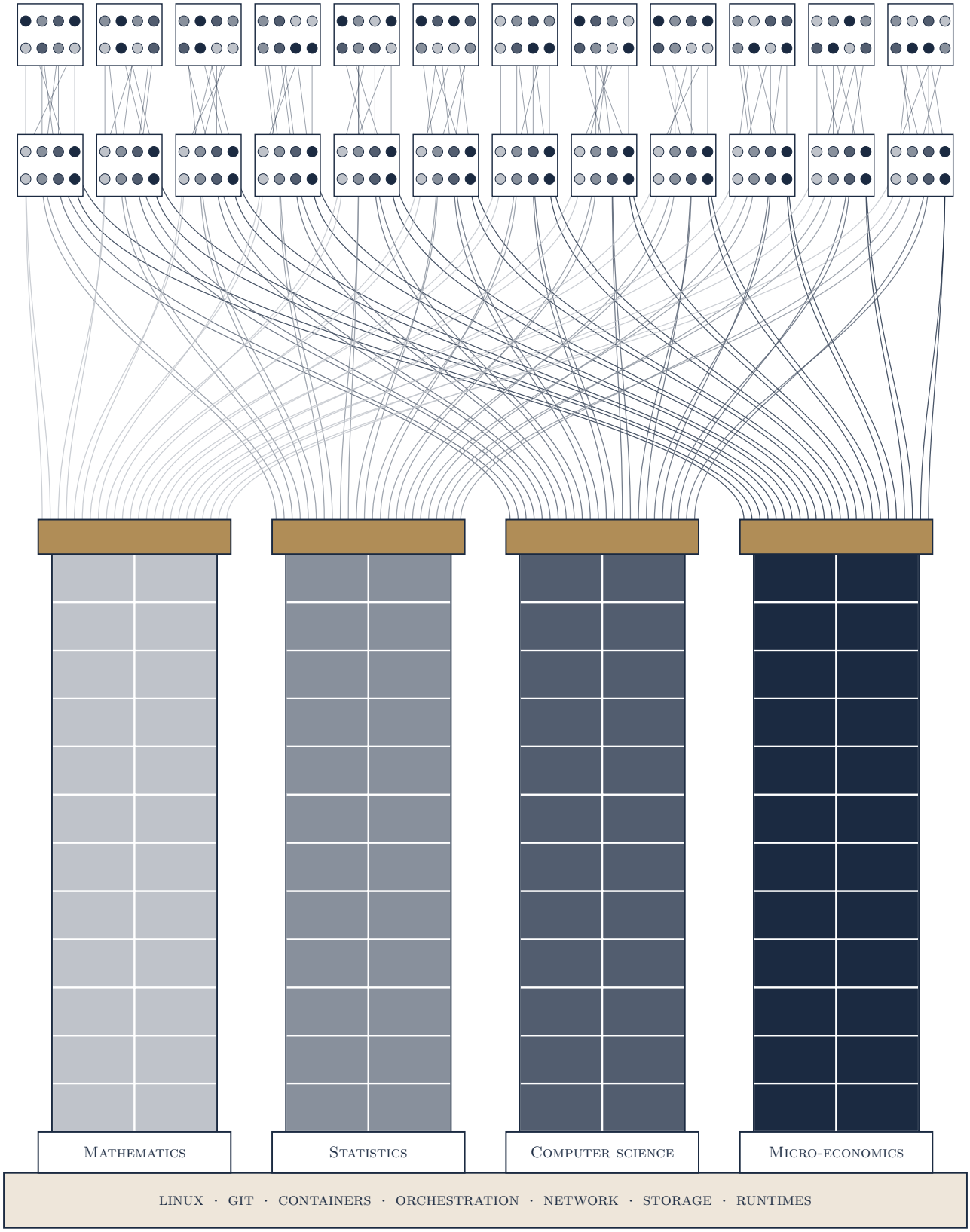
A 1ArchCodeMathsProveDog Book

ScaDaMaLe: Scalable Data Science and Distributed Machine Learning



Raazesh Sainudiin — An \|/*ias*/|\ Commons Continuum
Laboratory for Mathematical Statistical Experiments, Research Division, VakeWorks

ScaDaMaLe: Scalable Data Science and Distributed Machine Learning

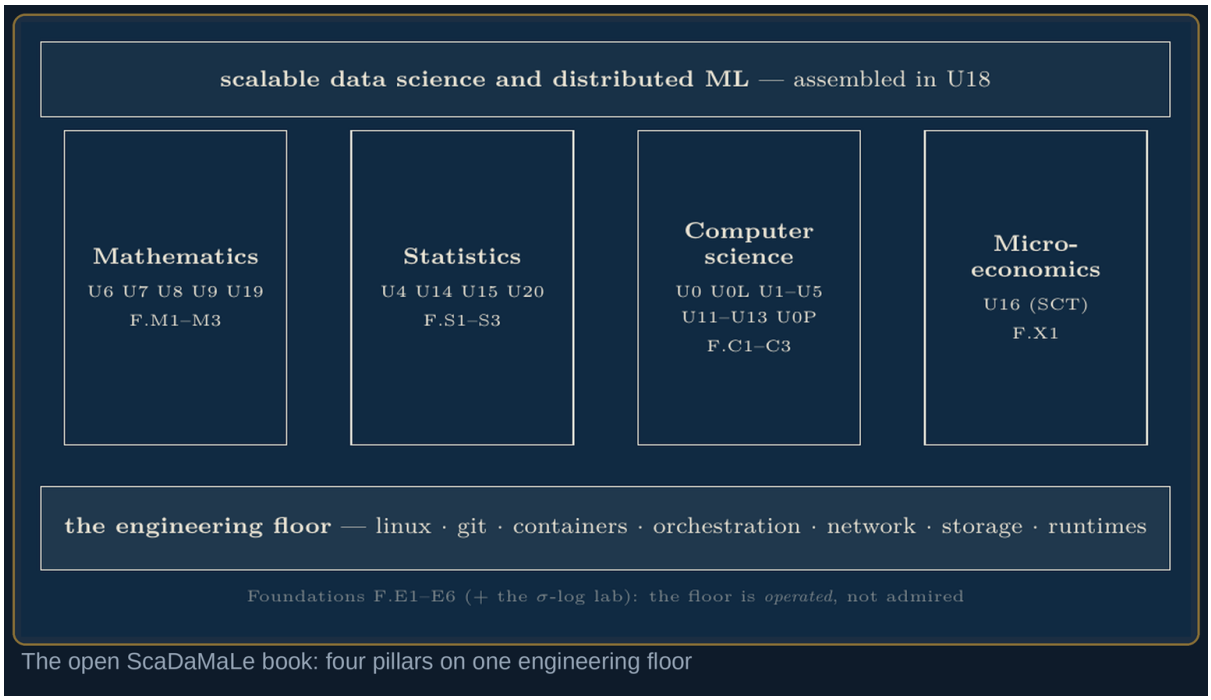
WORKING DRAFT

Raazesh Sainudiin — An \|/*ias*/|\ Commons Continuum
Laboratory for Mathematical Statistical Experiments, Research Division, VakeWorks

November 17, 2026

Study plan — a suggested course-week schedule

built 2026-11-17T16:37:26Z · source commit 96a8411c7178 · sources sha256:7213ad9fb5dee442 · this PDF's checksum: see VERIFY-SHA256.txt published beside it



The open ScaDaMaLe book: four pillars on one engineering floor

Copyright

© 2026 VakeWorks AB. The text, mathematics, figures and student exercises of this book are licensed under the Creative Commons Attribution–ShareAlike 4.0 International License (CC BY–SA 4.0), https://creativecommons.org/licenses/by-sa/4.0/. Runnable code is licensed separately (Apache-2.0, AGPL-3.0-or-later unless noted); product and company names are trademarks and are not covered by the content licence. See the LICENSE file accompanying the book source.

Contents

Start Here **10**

Study plan **11**

1 00 models and cost **13**
1. Intuition — three machines, three prices . . . 13
2. RAM — the sequential baseline . . . 14
3. PRAM — shared-memory parallelism: work, depth, Brent . . . 15
4. DPRAM / MPC — distributed data: rounds and communication . . . 18
5. The invariant (memorise this box) . . . 24
6. Lab (docker-compose; laptop-runnable; Canvas-submittable) . . . 24
7. Lean certificate (proofs via our prover) . . . 25
8. Sources (credited; archived under `scadamale-book-sources`) . . . 25
9. Exercises . . . 25

2 00A four pillars **29**
1. The four pillars . . . 29
2. Statistical contract theory — the day-one first look . . . 31
3. The cost-lens habit (bridge to U0) . . . 32
4. How to take this course . . . 32
5. Sources . . . 32
6. Exercises . . . 33

3 00K os foundations **34**
1. Acronyms, expanded (the master table — none are assumed) . . . 34
2. The stack, bottom to top — and the chain of trust . . . 35
3. What the kernel is, precisely . . . 35
4. How to prove things about system behaviour . . . 37
5. Why this matters now — the agentic age . . . 39
6. Open project ideas (for group projects) . . . 39
7. Sources . . . 40
8. Exercises . . . 40

4 00L llm workload **42**
1. What an LLM is, formally . . . 42
2. The cost anatomy of inference — one formula to carry . . . 44
3. Dense vs mixture-of-experts (MoE): what “active” means . . . 48
4. Splitting a model across machines: the trichotomy . . . 49
5. The frameworks, descriptively . . . 51
6. Honesty box (U0P discipline, applied to system claims) . . . 51
7. Where every later unit lands on this workload . . . 51

8. Sources (credited; archived under `scadamale-book-sources`) . . . 52
9. Exercises . . . 52

5 00N networking foundations **54**
1. Acronyms, expanded (the master table — none are assumed) . . . 54
2. The layered idea, and the OSI 7-layer model . . . 55
3. The core protocols (one line each) . . . 55
4. No black magic I — what `curl https://a2a-protocol.org/latest/specification/` really does . . . 56
5. No black magic II — what `docker run <image>` really does . . . 56
6. Where THIS course sits (the tie to the cost spine) . . . 57
7. Sources . . . 57
8. Exercises . . . 57

6 00S serialization rpc **59**
1. Acronyms (extending U00N's master table) . . . 59
2. Serialization: structure ↔ bytes . . . 59
3. RPC: calling the other agent . . . 61
4. The payoff: A2A and MCP are assemblies of this floor . . . 62
5. Tie to the cost spine (DPRAM) . . . 62
5b. One lens: the resolution lattice . . . 62
6. Exercises . . . 63
7. Sources . . . 63

7 01 reductions scan **64**
1. Intuition — why scan looks sequential . . . 64
2. RAM — the sequential baseline . . . 65
3. PRAM — Blelloch two-sweep scan . . . 66
4. DPRAM / MPC — distributed scan . . . 68
5. The invariant — instantiated for scan . . . 69
6. Lab sketch — `labs/lab1-alpha-beta/` . . . 70
7. Lean certificate (proofs via our prover) . . . 70
8. Sources (credited; archived under `scadamale-book-sources`) . . . 71
9. Exercises . . . 71

8 02 sorting **73**
1. Intuition — sorting as data routing . . . 73
2. RAM — sequential sorting: mergesort, quicksort, quick-select . . . 74
3. PRAM — parallel sorting: parallel merge and parallel mergesort . . . 75
4. DPRAM / MPC — distributed sorting: sample sort and TeraSort shape . . . 77
5. The invariant (U2 instantiation) . . . 81
6. Lab sketch (`labs/lab2-sample-sort-spark/`) . . . 81
7. Lean certificate target . . . 82
8. Sources (credited; archived under `scadamale-book-sources`) . . . 82
9. Exercises . . . 83

9 03 scheduling **84**
1. Intuition — the straggler problem . . . 84
2. RAM — offline makespan minimisation: NP-hardness of the optimum . . . 85
3. PRAM — list scheduling: the Graham theorem (THE CENTERPIECE) . . . 86
4. DPRAM / MPC — distributed load-balancing . . . 89
5. The invariant — instantiated for scheduling . . . 91

6. Lab — project track . . . 92
7. Lean certificate (proofs via our prover) . . . 93
8. Sources (credited; archived under `scadamale-book-sources`) . . . 94
9. Exercises . . . 94

10 04 sampling streaming **96**
1. Intuition — memory is a resource too . . . 96
2. RAM — exact counting and quantiles as the baseline . . . 97
3. Centerpiece 1 — Vitter reservoir sampling . . . 98
4. Centerpiece 2 — Count-Min sketch . . . 100
5. HyperLogLog — approximate distinct count . . . 102
6. PRAM — parallel reservoir sampling and mergeable sketches . . . 104
7. DPRAM / MPC — sketches as the distributed answer . . . 106
8. The invariant (U4 instantiation) . . . 107
9. Lab (Lab 3 — streaming sketches) . . . 107
10. Lean certificate target . . . 108
11. Sources (credited; archived under `scadamale-book-sources`) . . . 108
12. Exercises . . . 109

11 05 mapreduce spark **111**
1. Intuition — why the shape is what it is . . . 111
2. RAM — the model, stated precisely (Dean–Ghemawat formulation) . . . 114
3. PRAM — the BSP bridge (U0 §4, deepened) . . . 116
4. DPRAM / MPC — RDDs, partitioning, shuffle cost, and joins . . . 117
5. The invariant (U5 instantiation) . . . 127
6. Lab — project track . . . 128
7. Lean certificate target . . . 129
8. Sources (credited; archived under `scadamale-book-sources`) . . . 130
9. Exercises . . . 132

12 06 matmul **134**
1. Intuition — why matmul is the algorithm . . . 134
2. RAM — sequential matrix multiplication: cubic and Strassen . . . 135
3. PRAM — parallel matrix multiplication: block-recursive and Strassen . . . 137
4. DPRAM / MPC — communication-avoiding matrix multiplication . . . 138
5. The invariant (U6 instantiation) . . . 142
6. Lab (Lab 4 — blocked matmul) . . . 142
7. Lean certificate target . . . 143
8. Sources (credited; archived under `scadamale-book-sources`) . . . 144
9. Exercises . . . 145

13 07 svd jl dimsum **147**
1. Intuition — why high dimensions hurt, and three strategies . . . 147
2. RAM — classical SVD and the Eckart–Young theorem . . . 148
3. Johnson–Lindenstrauss lemma . . . 150
4. Randomised SVD (Halko–Martinsson–Tropp) . . . 151
5. Sparse subspace embeddings — sketch-and-solve (CountSketch) . . . 152
6. Distributed tall-skinny SVD — the $A^\top A$ Gramian pipeline . . . 153
7. Case study — DIMSUM: importance sampling meets the shuffle cost model (Zadeh–Carlsson 2013) . . . 155
8. Invariant (U7 instantiation) . . . 158
9. Lab — project track . . . 158

10. Lean certificate target . . . 159
11. Sources (credited; archived under `scadamale-book-sources`) . . . 159
12. Exercises . . . 160

14 08 solvers least squares **162**
1. Intuition — three regimes of linear problems at scale . . . 162
2. RAM — direct methods and the least-squares problem . . . 163
3. PRAM — parallelising solvers . . . 166
4. DPRAM / MPC — distributed least squares . . . 168
5. The projection theorem — foundation for U9 and U19 . . . 172
6. The invariant (U8 instantiation) . . . 172
7. Lab — project track . . . 173
8. Lean certificate target (`BMLSPar.lean` family) . . . 174
9. Sources (credited; archived under `scadamale-book-sources`) . . . 175
10. Exercises . . . 175

15 09 gd sgd **177**
1. Intuition — why gradient descent, and what "smooth" buys . . . 177
2. RAM — gradient descent: smooth and strongly convex rates . . . 178
3. The modern weaves — formal, compact . . . 180
4. SGD — stochastic gradient and minibatch . . . 184
5. PRAM — parallel gradient computation . . . 187
6. DPRAM / MPC — distributed optimisation architectures . . . 187
7. The invariant (U9 instantiation) . . . 190
8. Lab (Lab 5 — distributed SGD and parameter averaging) . . . 191
9. Lean certificate target . . . 192
10. Sources (credited; archived under `scadamale-book-sources`) . . . 193
11. Exercises . . . 194

16 0P proof checkers **196**
1. Intuition — what does "the checker accepted it" actually mean? . . . 196
2. What a machine-checked proof is . . . 197
3. Reading a Lean 4 proof . . . 198
4. Running the checker — the dockerised lab (L1) . . . 199
5. The cert ladder of this course . . . 200
6. Why machine-checking matters at scale . . . 201
7. Sources . . . 202
8. Exercises . . . 202

17 11 mst cc **204**
1. Intuition — spanning trees as consensus . . . 204
2. RAM — Kruskal's algorithm and the cut property . . . 205
3. PRAM — Borůvka's algorithm (the parallel centrepiece) . . . 207
4. Connected components — label propagation and the diameter dichotomy . . . 209
4b. Strongly connected components: when the best sequential algorithm refuses to parallelise . . . 210
5. DPRAM / MPC — distributed Borůvka and CC in MapReduce . . . 214
6. The invariant (U11 instantiation) . . . 215
7. Lab sketch — project track . . . 215
8. Lean certificate target . . . 216
9. Sources (credited; archived under `scadamale-book-sources`) . . . 217
10. Exercises . . . 218

18 12 pagerank triangles **219**
1. Intuition — three graph problems, one lesson . . . 219
2. RAM — the sequential baselines . . . 220
3. PRAM — parallel PageRank frontier and NodeIterator++ . . . 224
4. DPRAM / Spark — co-partitioned PageRank and two-pass triangle counting . . . 225
5. Graph partitioning . . . 229
6. Distributed vertex programming: BSP → Pregel → GAS → GraphX . . . 231
7. The invariant (U12 instantiation) . . . 234
8. Lab (Lab 6 — PageRank two ways) . . . 235
9. Lean certificate target . . . 236
10. Sources (credited; archived under `scadamale-book-sources`) . . . 237
11. Exercises . . . 238

19 13 clustering ann **240**
1. Intuition — the geometry of scale . . . 240
2. RAM — k-means, Lloyd's algorithm, and k-means++ seeding . . . 241
3. Centerpiece 1 — k-means‖ (distributed seeding in $O(\log n)$ rounds) . . . 242
4. Similarity search: exact cost, then LSH . . . 244
5. Modern ANN: HNSW, IVF/PQ, and the theory/practice gap . . . 247
6. Coresets — provable compression for clustering . . . 248
7. Submodularity and the greedy $(1 - 1/e)$ guarantee . . . 249
8. The invariant (U13 instantiation) . . . 250
9. Lab — project track . . . 251
10. Lean certificate target . . . 252
11. Sources (credited; archived under `scadamale-book-sources`) . . . 252
12. Exercises . . . 253

20 14 density trees **255**
1. Intuition — from bins to trees . . . 255
2. RAM — histogram density estimation and regular pavings . . . 256
3. PRAM — parallel tree build . . . 258
4. DPRAM / MPC — distributed sort on tree space . . . 258
5. Statistical guarantees — L1-consistency (stated) . . . 260
6. Lab sketch — project track . . . 261
7. Lean certificate target . . . 261
8. Sources (credited; archived under `scadamale-book-sources`) . . . 262
9. The invariant (U14 instantiation) . . . 262
10. Exercises . . . 263

21 15 l1mde **264**
1. Intuition — why total variation is the right currency . . . 264
2. Total variation and Scheffé's identity . . . 265
3. The Yatracos class and the minimum-distance estimate . . . 267
4. VC dimension, Sauer–Shelah, and the rate . . . 268
5. DPRAM / MPC — distributed L1-MDE . . . 270
6. The security capstone: the keyless watermark detector is Scheffé–Yatracos in L_1 . . . 271
7. The cert ladder (U15 instantiation) . . . 272
8. The invariant (U15 instantiation) . . . 273
9. Lab — project track . . . 273
10. Lean certificate (students extend) . . . 274
11. Sources (credited; archived under `scadamale-book-sources`) . . . 274
12. Exercises . . . 275

22 16 sct **277**
Framing: the fourth pillar, examined . . . 277
1. Why prediction is not enough: the collective view . . . 277
2. Classical contract theory: principal, agent, hidden information . . . 279
3. Statistical Contract Theory: statistical evidence as the observable . . . 279
4. Learning agents in markets . . . 282
5. Strategic data . . . 283
6. The U16 → U20 bridge: statistical contracts and conformal coverage . . . 285
7. The IAS ladder: Part V hook . . . 285
8. Invariant box (memorise this) . . . 286
9. Lab sketch — project track . . . 287
10. Lean certificate (schematic) . . . 287
11. Sources . . . 288
12. Exercises . . . 289

23 17 agentic orchestration **290**
1. Intuition — the systems everyone is building, the theorems you already own . . . 290
2. The substrate is a graph: SPIG . . . 293
3. Governed scheduling: the Lagom Obligations Graph . . . 293
4. Governed capability: what a worker may do, and what it may see . . . 294
5. The worker forgets: the context projection . . . 296
6. Continuity without process-liveness: the σ-chain as a sufficient statistic . . . 297
7. Identity and attestation: which agent did what, provably . . . 298
8. Many agents, many groups: coordination and robustness . . . 298
9. Why report honestly? Incentives close the loop . . . 299
10. Lab — the receipt chain you have been living inside . . . 299
11. Module-3 projects (deep dives; each = one paper section + one certificate) . . . 299
12. Sources and provenance . . . 299

24 18 llm ddl **301**
1. Intuition — the course as a construction kit . . . 301
2. Formally what an LLM is: a conditional density estimator . . . 302
3. The transformer as matmul composition (U6 lens) . . . 303
4. Distributed training: the course composed . . . 305
5. Scaling laws as empirical rates (U15 epistemics) . . . 309
5b. Mixture-of-experts: the sparse composition (U0L §3, in depth) . . . 310
6. Inference and serving: KV-cache, batching, quantisation, and the engine . . . 310
7. LoRA: low-rank adaptation (U7 lens) . . . 312
8. RAG: retrieval as ANN (U13 lens) . . . 313
8b. Local models: inference and fine-tuning on your own machine . . . 313
9. Watermark detection as Scheffé–Yatracos (U15 security capstone) . . . 317
10. Agentic systems: awareness, and the link back to U16 and U17 . . . 318
11. The invariant: all five lenses at once . . . 318
12. Lab — project track . . . 319
13. Lean certificate target . . . 320
14. Sources (credited) . . . 320
15. Exercises . . . 321

25 19 admm consensus **325**
1. Intuition — why a second agent appears in the dual . . . 325
2. Convex duality: compact recap . . . 326
3. Dual decomposition: where it helps and where it fails . . . 327

4. ADMM — the centrepiece . . . 327
5. Decentralised methods: gossip and DSGD . . . 330
6. Newton-sketch: the randNLA × optimisation crossover . . . 331
7. The invariant (memorise this box) . . . 332
8. Lab sketch — project track . . . 332
9. Lean target . . . 333
10. Sources . . . 333
11. Exercises . . . 334

26 20 distributed inference **335**
1. Intuition — statistical efficiency as a resource . . . 335
2. One-shot / divide-and-conquer averaging . . . 336
3. Bag of Little Bootstraps (the centerpiece) . . . 338
4. Communication-constrained minimax lower bounds . . . 339
5. Split conformal prediction — the SCT bridge . . . 341
6. The invariant (U20 instantiation) . . . 343
7. Lab sketch — project track . . . 343
8. Lean target (schematic) . . . 344
9. Sources (credited; listed for archiving — not yet in the local research archive) . . . 344
10. Exercises . . . 345

I Laboratory exercises **347**

27 Toolchain + proof-checker **348**
What you set up . . . 348
The one idea worth taking away . . . 348
Submit . . . 348

28 Ping-pong + tree vs ring allreduce (the alpha+beta*n model) **349**
The model you measure . . . 349
What you run . . . 349
What to observe (this is the point — read it before you upload) . . . 349
Submit . . . 350

29 Sample sort on Spark **351**
What you build . . . 351
What to observe (before you upload) . . . 351
Submit . . . 351

30 Streaming sketches: reservoir merge, Count-Min, HyperLogLog **352**
The claims you check . . . 352
What you run . . . 352
What to observe (this is the point — read it before you upload) . . . 353
Under the hood . . . 353
Submit . . . 353

31 Blocked matrix multiplication: block size and sparsity **355**
The claims you check . . . 355
What you run . . . 355
What to observe (this is the point — read it before you upload) . . . 355
Under the hood . . . 356
Submit . . . 356

32 Distributed gradient descent and parameter averaging **357**
The claims you check . . . 357
What you run . . . 357
What to observe (this is the point — read it before you upload) . . . 357
Under the hood . . . 358
Submit . . . 358

33 PageRank two ways: naive vs co-partitioned RDDs, and GraphX **359**
The claims you check . . . 359
What you run . . . 359
What to observe (this is the point — read it before you upload) . . . 359
Under the hood . . . 360
Submit . . . 360

II Preliminaries — enter as needed **361**

34 Mathematical preliminaries **362**
F.M1 — Asymptotics, recurrences, and sums (a recap) . . . 362
F.M2 — Linear algebra: the six facts the middle of the book runs on . . . 363
F.M3 — Convexity, gradients, and a first look at duality . . . 365

35 Statistical preliminaries **367**
F.S1 — Probability and expectation, finite-first . . . 367
F.S2 — The concentration ladder: Markov → Chebyshev → Hoeffding . . . 369
F.S3 — Densities, total variation, and what "estimating" means . . . 371
F.S4 — Information: entropy, KL, and perplexity . . . 373

36 Computational preliminaries **377**
F.C1 — Machine models: RAM → PRAM → DPRAM (a recap) . . . 377
F.C1b — Problems, classes, and reductions . . . 378
F.C2 — Discrete structures and hashing . . . 382
F.C2b — Order, lattices, and fixed points . . . 383
F.C3 — Logic, induction, and reading a machine-checked proof . . . 389

37 Economic preliminaries **405**
F.X1 — Utility, incentives, and what a contract is . . . 405

38 The engineering floor **408**
F.E1 — Linux and the shell . . . 408
F.E1b — Git, SSH keys, and a sovereign git server . . . 409
F.E2 — Docker: the reproducible container . . . 411
F.E3 — Kubernetes: containers on a cluster . . . 412
F.E4 — The network is physical: the α-β cost model . . . 413
F.E5 — Storage anatomy: a distributed filesystem and database . . . 415
F.E6 — Runtime anatomy: batch, actor, streaming, and serving engines . . . 423
Lab (Engineering Floor) — Proving contribution by replaying the log . . . 424

Acronyms and abbreviations **429**

Subject index **431**

Start Here

how to use this learning resource: the atlas, the units, the labs, the slides

the atlas is the map; the units + labs are the terrain; read the atlas and the book together. Clone the course repo (below); this 2026 course lives in its 2026/ directory (2027/, latest/ will follow) – the labs are in 2026/labs/.

Get the labs

Clone the course repository from whichever forge is nearest you — all four hold the same content:

- **git.vake.works**: `https://git.vake.works/vakeworks/scadamale` (primary — sovereign)
- **codeberg**: `https://codeberg.org/vakeworks/scadamale`
- **github**: `https://github.com/vakeworks/scadamale`
- **gitlab**: `https://gitlab.com/vakeworks/scadamale`

How every lab works

Run `bash run-lab.sh`; it produces a receipt (tool versions + measured numbers + SHA-256) + one line appended to labs/sigma-log.jsonl (a hash-chain). To submit: keep your receipts + sigma-log.jsonl locally (optional VW-cert later) (open self-study), or upload the receipt to KTH Canvas (WASP cohort).

A companion book

This course pairs with Jim Dowling's *Building Machine Learning Systems with a Feature Store: Batch, Real-Time, and LLM Systems* (O'Reilly, 2025) — the production-systems companion to ScaDaMaLe. Where this book gives the theory and proves *why* the algorithms scale (and measures them in Docker labs), Dowling shows *how* to build and run real ML systems: the Feature/Training/Inference (FTI) pipeline architecture and the feature store as the central data layer, for batch, real-time and LLM systems. The two interlock most at this book's feature-store / FTI and MLOps material.

Dowling's book is built on the European **Hopsworks** stack and is free to read (courtesy PDF via Hopsworks; © O'Reilly Media / Jim Dowling — we cite it, never reproduce it).

Study plan — a suggested course-week schedule

Follow the atlas for the reading order of the units — prerequisites sit lower, and following the arrows upward is a valid learning order. The labs are paced by course week:

Course week	Lab
cw1	Toolchain + proof-checker
cw3	Ping-pong + tree vs ring allreduce (the alpha+beta*n model)
cw4	Sample sort on Spark
cw6	Streaming sketches: reservoir merge, Count-Min, HyperLogLog
cw7	Blocked matrix multiplication: block size and sparsity
cw9	Distributed gradient descent and parameter averaging
cw10	PageRank two ways: naive vs co-partitioned RDDs, and GraphX

This is a self-study pace with no fixed calendar — follow it at whatever speed suits you.

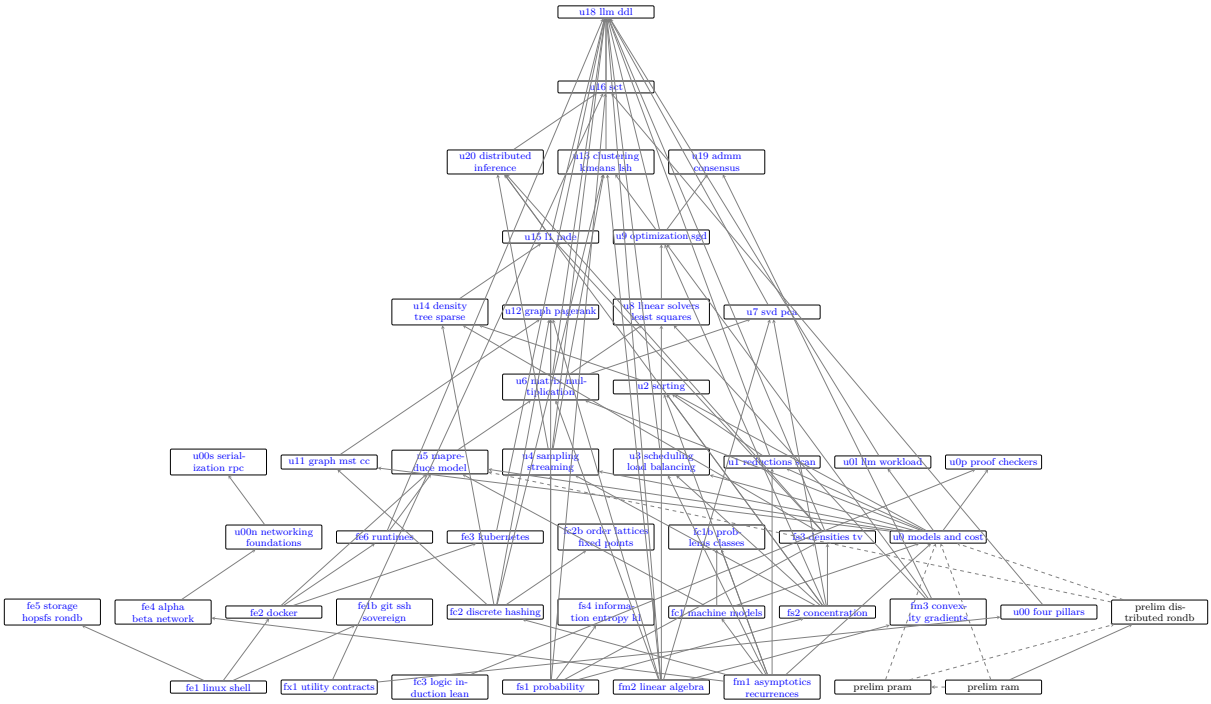


Figure 1: The atlas: the book's chapter prerequisite graph, generated from its pinUni encoding. Prerequisites sit lower; following the arrows upward is a valid learning order. Every box is a hyperlink to its chapter. Dashed edges refine rather than require.

Chapter 1

00 models and cost

U0 — Models & cost: RAM → PRAM → DPRAM

The ScaDaMaLe book, unit 0 (examined core). Our own exposition; original sources credited in §8.

The one organising idea of this whole book: *the same algorithmic problem is re-solved as the machine model gains a resource; what changes is the cost you must account for.* Sequential asks "how many operations?" Parallel adds "how deep is the dependency chain?" Distributed adds "how many rounds, and how many bytes cross the network?" Every later unit takes ONE classical algorithm and walks it through these three lenses. This unit fixes the lenses.

Entry kit — what this chapter assumes; follow a link if you need the memory jog (map: Figure 1): F.C1 (the RAM/PRAM/DPRAM machine recap) • F.M1 (asymptotics and recurrences)

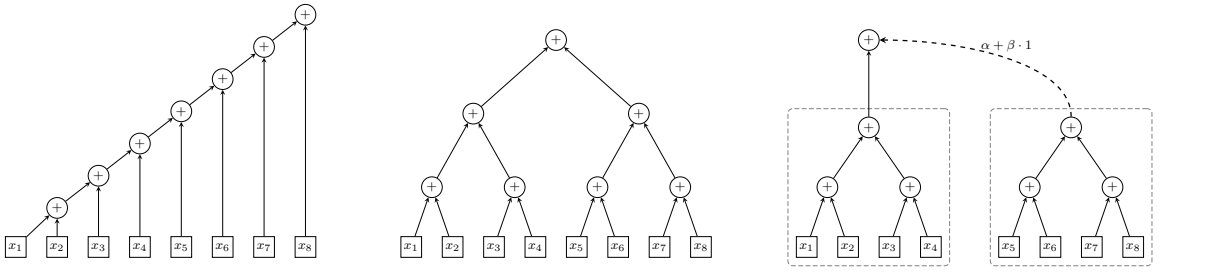
1. Intuition — three machines, three prices

Imagine summing n numbers.

- On **one processor**, you pay one price: $n - 1$ additions, one after another.
- With **many processors sharing one memory**, pairs can be added simultaneously: a balanced binary tree of additions finishes in $\lceil \log_2 n \rceil$ *steps*, though the *total* number of additions is still $n - 1$. Two different quantities have appeared: total work, and the length of the longest chain of operations that must happen in order.
- With the data **partitioned across machines that must exchange messages**, a third quantity appears and usually dominates: each machine sums its own shard locally (cheap), then the partial sums must *travel*. The price is now counted in **communication rounds** and **bytes moved**, and the fastest algorithm is often the one that moves the least data, not the one that does the fewest additions.

Same problem, three cost models. Formalising these three is the entire content of U0. Figure 1.1 is the whole unit in one picture — return to it after §4 and every annotation will have become a theorem.

Check your reading.
Q1. Summing n numbers split across two machines, each machine sums its half locally and one partial sum crosses the network at cost alpha + beta. For SMALL n, the total time is typically dominated by: (A) the local additions (B) the network term alpha + beta (C) reading the input from disk (D) the final single addition



(a) RAM (sequential)
the caterpillar: $W = n - 1$, $D = n - 1$

(b) PRAM (shared memory)
the balanced tree: $W = n - 1$, $D = \lceil \log_2 n \rceil$

(c) DPRAM (two PRAMs + a network)
$T = \lceil \log_2 \frac{n}{2} \rceil + (\alpha + \beta) + 1$ — the wire term can dominate everything

Figure 1.1: Adding n numbers ($n = 8$) on the book's three machines. (a) The sequential RAM folds left-to-right: the *caterpillar* tree, work $n - 1$, depth $n - 1$. (b) The PRAM reshapes the *same* $n - 1$ additions into a balanced tree of depth $\lceil \log_2 n \rceil$ — Brent's theorem lives on this picture: with p processors, fold its levels. (c) The DPRAM splits the data across two PRAM machines, each sums its half by a balanced tree, and *one word crosses the network* at cost $\alpha + \beta \cdot 1$ before the final addition. Communication cost enters as a picture before it enters as algebra: for small n the dashed wire costs more than all the arithmetic beneath it, and the generalisation of panel (c) to p machines is exactly the allreduce of U1.

2. RAM — the sequential baseline

The **Random-Access Machine (RAM)**: one processor, unit-cost arithmetic/compare/load/store on machine words, random access to an unbounded memory. The cost of an algorithm is $T(n)$ = the number of primitive operations on the worst-case (or expected, when stated) input of size n.

Why it matters here: $T_{\text{seq}}(n)$ for the *best known sequential algorithm* is the yardstick every parallel or distributed version must answer to. A parallel algorithm that does vastly more total work than T_{seq} is spending processors to buy back what it wasted.

2.1 Worked: eight numbers on the RAM

Take the running input of this book, $a = [3, 1, 7, 0, 4, 1, 6, 3]$ ($n = 8$, sum 25). The RAM folds it left to right:

```
3 --+1--> 4 --+7--> 11 --+0--> 11 --+4--> 15 --+1--> 16 --+6--> 22 --+3--> 25
```

Each arrow is one addition, and each needs the result of the one before it. So

$$T_{\text{seq}}(n) = n - 1 \qquad (n = 8 : \; T_{\text{seq}} = 7).$$

Read as a dependency DAG (§3), this chain has $W = 7$ operations and a longest chain of $D = 7$. Every operation waits for its predecessor, so a second processor would have nothing to do. This is the *caterpillar* of Figure 1.1(a). The algorithm, not the problem, is what has no parallelism: §3.1 re-brackets the same seven additions into a tree of depth 3.

Check your reading.
Q2. On the RAM model, the cost of an algorithm is:
(A) the wall-clock seconds on a reference laptop (B) the number of unit-cost operations as a function of input size (C) the number of bytes of memory allocated (D) the number of lines of code

3. PRAM — shared-memory parallelism: work, depth, Brent

The **Parallel Random-Access Machine (PRAM)**: p synchronous processors sharing one random-access memory. Variants restrict simultaneous access to a cell: **EREW** (exclusive read, exclusive write), **CREW** (concurrent read, exclusive write), **CRCW** (concurrent read and write, with a stated write-resolution rule). The variant matters for constant factors and for what is achievable at depth $O(1)$–$O(\log n)$; we will name it whenever it is load-bearing.

An execution is a DAG of operations (edges = data dependencies). Two costs:

- **Work** $W(n)$ — the total number of operations (the size of the DAG). With one processor, $T_1 = W$.
- **Depth (span)** $D(n)$ — the length of the longest dependency chain (the DAG's critical path). With unboundedly many processors, $T_\infty = D$: no schedule can beat the critical path.

Theorem (Brent 1974). *Any computation of work W and depth D can be executed on p processors in time*

$$T_p \;\le\; \frac{W}{p} + D.$$

Proof idea. Schedule the DAG level by level. Level i has m_i operations, all of whose predecessors are complete, so it takes $\lceil m_i/p \rceil \le m_i/p+1$ steps; summing over the D levels gives $\sum_i m_i/p+D = W/p + D$. □

Both directions of the bound are the design guide: - $T_p \ge \max(W/p,\ D)$ always (you can neither beat perfect work-sharing nor the critical path), so Brent's level-by-level schedule (and, by §3.2, any greedy schedule) is within a factor 2 of optimal for the DAG. - **Speed-up** $T_1/T_p \approx p$ (linear) exactly when $W/p \gg D$ — i.e. when the input is large relative to the machine, adding processors keeps paying until $p \approx W/D$ (the *parallelism* of the algorithm).

Work-optimality. A PRAM algorithm is *work-optimal* if $W = O(T_{\text{seq}})$. The discipline of this book: first make it work-optimal, then drive the depth down. Low depth bought with super-linear work is a false economy at scale.

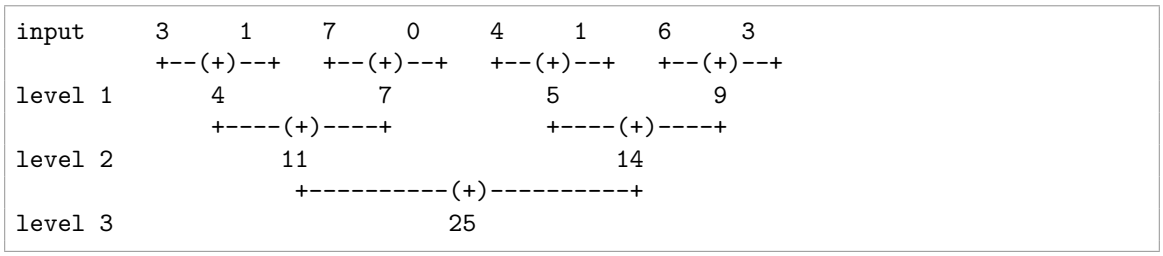
The two universal primitives. Almost every parallel algorithm in this book is assembled from: - **Reduction:** combine n values with an associative $\oplus$ — a balanced tree gives $W = O(n)$, $D = O(\log n)$. - **Scan (all-prefix-sums, Blelloch 1990):** from $(a_1, \ldots, a_n)$ compute all prefixes $(a_1,\ a_1 \oplus a_2,\ \ldots)$. The two-sweep (up-sweep reduce + down-sweep distribute) algorithm is work-optimal $W = O(n)$ at depth $O(\log n)$ — surprisingly, knowing *all* prefixes costs no more work than knowing the last one. Scan turns "sequential-looking" chains (numbering, packing, partitioning) into $O(\log n)$-depth operations; it is the workhorse of U1.

Associativity is the licence for all of this: it is what lets a left-to-right fold be re-bracketed into a tree. (Floating-point addition is *not* associative — a real-world caveat every lab revisits.)

Machine-checked (our theorem prover). The Brent bound above is not merely asserted: the course ships `BMLSPar.lean`, a Lean proof of the scheduling bound for the level-by-level schedule (§3.4 states its exact scope), and *your first proof exercise is to run the checker and then extend it* (a blanked inductive step; see §7). Later units add their own certificates. The point of the exercise: at scale, "the proof checks by machine" is the difference between a cost model you can bet a cluster budget on and folklore.

3.1 Worked: the same sum as a balanced tree

Associativity, $(x \oplus y) \oplus z = x \oplus (y \oplus z)$, licenses re-bracketing the chain of §2.1 into a balanced tree:



Level l has $n/2^l$ additions, here $4 + 2 + 1$. So

$$W = n - 1 = 7 \text{ (work-optimal)}, \qquad D = \lceil \log_2 n \rceil = 3.$$

That $W = n - 1$ is no accident of the tree shape. Every addition replaces two numbers by one, so *any* bracketing of a sum of n numbers uses exactly $n - 1$ additions. Re-bracketing changes the depth, never the work. (For floating-point numbers the tree shape can change the computed answer, because floating-point addition is not associative; see above and §6, Experiment 3.)

3.2 Greedy schedules: what "greedy" means, and why they are within a factor 2

A **greedy algorithm** is one that, at each step, takes the choice that looks best *now*. It does not look ahead and never undoes a choice. For scheduling a DAG on p processors, the greedy rule is: *never leave a processor idle while some operation is ready* (all its inputs computed). At each step, if at least p operations are ready, run any p of them; otherwise run all of them. The order among ready operations is arbitrary, so no planning is needed. For independent jobs this is Graham's **list scheduling** (U3 §3.1).

Greedy is a *rule*, not a promise of the optimum. Write T_p^* for the fastest time that *any* schedule of the given DAG achieves on p processors. Computing T_p^* is NP-hard in general: already for independent jobs on two machines, the scheduling problem contains PARTITION (U3 §2.2). What greedy does guarantee is the following.

Claim (greedy schedule bound). *Every greedy schedule of a DAG of unit-cost operations with work W and depth D finishes within*

$$T_p^{\text{greedy}} \;\le\; \frac{W - D}{p} + D \;\le\; \frac{W}{p} + D.$$

Proof. Call a step **complete** if it runs p operations and **incomplete** otherwise.

- *Complete steps.* Each runs p operations.
- *Incomplete steps.* An incomplete step runs every ready operation. Take any longest path among the not-yet-run operations. Its first operation has no unrun predecessor (otherwise the path could be extended), so it is ready, so it runs. Hence each incomplete step shortens the longest remaining path by one, and there are at most $I \le D$ incomplete steps.
- *Counting.* The incomplete steps run at least I operations, so the complete steps number at most $(W - I)/p$.

Therefore $T \le (W - I)/p + I$. This is non-decreasing in I, because its coefficient $1 - 1/p$ is at least 0, so $I \le D$ gives $T \le (W - D)/p + D$. □

With the lower bound $T_p^* \ge \max(W/p, D)$ this gives the factor 2:

$$T_p^{\text{greedy}} \;\le\; \frac{W}{p} + D \;\le\; 2\max\Big(\frac{W}{p},\, D\Big) \;\le\; 2\,T_p^*.$$

Level-by-level is simpler, but it is not greedy. The proof idea in §3 runs the DAG one level at a time, finishing level i before starting level $i+1$. That schedule can leave a processor idle while an operation of the next level is already ready. Example: the tree of §3.1 with $p = 3$. Step 1 runs three of the four level-1 additions. Step 2 runs only the fourth, although $4 + 7$ is ready. It still meets the bound, because $\lceil m/p \rceil \le (m-1)/p + 1$ for $m \ge 1$ gives $\sum_i \lceil m_i/p \rceil \le (W - D)/p + D$.

Greedy is not optimal. Take $p = 2$ and a DAG of four unit operations: a chain $c_1 \to c_2$ and two independent operations x, y. Then $W = 4$, $D = 2$, and the lower bound is $\max(2, 2) = 2$.

- A greedy schedule that runs x, y first then needs c_1 and then c_2, for 3 steps.
- A greedy schedule that runs c_1, x first finishes c_2, y at step 2, which is optimal.

The ratio $3/2 = 2 - 1/p$ is the same one as U3 §3.3's instance $[1, 1, 2]$: a job of length 2 is exactly a chain of two unit operations. The factor 2 is the worst case, not the usual one: on the eight-number sum, greedy is optimal for every p (§3.3).

3.3 Brent, worked: the eight-number sum ($W = 7$, $D = 3$)

p	lower bound $\max(W/p, D)$	Brent $W/p + D$	certified $D + (W - D)/p$ (§3.4)	actual T_p
1	7	10	7	7
2	3.5, so 4 whole steps	6.5	5	**4**
3	3	5.33	4.33	**4**
4	3	4.75	4	**3**

The schedules for $p = 2$ and $p = 4$, step by step:

step	p=2: P1	P2	p=4: P1	P2	P3	P4
1	3+1=4	7+0=7	3+1=4	7+0=7	4+1=5	6+3=9
2	4+1=5	6+3=9	4+7=11	5+9=14	idle	idle
3	4+7=11	5+9=14	11+14=25	idle	idle	idle
4	11+14=25	idle				

- $p = 2$**.** Level 1 has $m_1 = 4$ operations and takes $\lceil 4/2 \rceil = 2$ steps; levels 2 and 3 take one step each. $T_2 = 4 = \lceil W/p \rceil$, with 1 idle slot of 8.
- $p = 4$**.** One step per level, so $T_4 = 3 = D$, with 5 idle slots of 12.
- $p = 3$**.** Also 4 steps, and 4 is optimal for $p = 3$. After any first step, at most one level-2 addition can be ready at step 2, so the root cannot be reached before step 4.
- **Every row.** Both upper bounds hold, and the actual schedule is optimal.
- **Speed-up.** $T_1/T_2 = 1.75$ and $T_1/T_4 = 7/3 \approx 2.3$. The parallelism is $W/D = 7/3$. At $p = 4$ the depth is already the binding term, so doubling p from 2 to 4 saves one step, not two, and more processors buy nothing.

In one line: **processors divide the work, never the depth.** The claim "$T = D$" holds only when there are enough processors for the widest level ($n/2 = 4$ here). With fewer, the work term W/p is back.

3.4 The certified form

`BMLSPar.lean` (U0P §5–6) certifies the level-by-level schedule of the proof idea in §3, which §3.2 showed is not greedy. Its statement takes a list of level sizes $m_1, \ldots, m_D$ (so $W = \sum_i m_i$) and $p \ge 1$, and proves for $T_p = \sum_i \lceil m_i/p \rceil$ the denominator-cleared inequality $p\,T_p \le W + D(p-1)$. Dividing by p,

$$T_p \;\le\; D + \frac{W - D}{p},$$

the same bound the greedy claim of §3.2 gives by hand. It implies Brent's $W/p + D$ because $(W - D)/p \leq W/p$. The certified statement is the stronger one; the textbook form is its corollary. However, the certificate takes the list of levels as given: that a DAG splits into D levels of ready operations is the informal step of §3, and the greedy claim of §3.2 is not certified.

Check your reading.

Q3. Summing n numbers by a balanced binary tree on a PRAM has:
(A) work O(log n) and depth O(log n) (B) work O(n) and depth O(log n) (C) work O(n log n) and depth O(n) (D) work O(n) and depth O(1)

Q4. Brent's principle states $T_p \leq$ W/p + D. Which statement is the correct reading?
(A) adding processors always speeds up any algorithm proportionally (B) speed-up is near-linear only while p is small compared with W/D; beyond that, depth dominates (C) depth can always be reduced by adding processors (D) the bound only holds for sorting algorithms

Q5. A parallel algorithm has work 5x the best sequential algorithm but depth O(log n). In one or two sentences: when is this a good trade, and when is it a bad one?

4. DPRAM / MPC — distributed data: rounds and communication

Now remove the shared memory. The **massively-parallel-computation (MPC)** model (Karloff–Suri– Vassilvitskii 2010, formalising the MapReduce/Spark reality; Valiant's **BSP** 1990 is the ancestor): M machines, each with **local memory** s (sublinear in n: the data does not fit on one machine — that is the whole premise). Computation proceeds in synchronous **rounds**: in a round, each machine computes freely on its local data, then machines exchange messages, subject to each machine sending and receiving $O(s)$ bytes per round.

Costs, in order of dominance: 1. **Rounds** R — each round is a cluster-wide synchronisation barrier (in Spark: a *stage boundary*, i.e. a shuffle). Real-world latency makes R the headline cost; the classical results are about achieving $R = O(1)$ or $O(\log n)$. 2. **Communication** — total bytes crossing the network; the per-round shape is **map** → **shuffle** → **reduce**: local computation, then an all-to-all exchange re-keying the data, then local computation on the new layout. *Communication-avoiding* algorithm design (moving asymptotically fewer bytes for the same answer, e.g. 2.5D matrix multiply in U6) is the distinctive game of this model. 3. **Local work** — the RAM cost inside each machine each round; it prices your per-executor CPU, but it rarely decides the winner.

The primitives recur. Reduce becomes *tree-aggregation* (partials up a machine-tree — `treeReduce`); scan becomes the *distributed scan* (local scans + a small scan of per-machine totals + a local fix-up pass — exactly U1's segmented structure); sorting becomes *sample-sort* (sample splitters, one shuffle, local sorts — U2). The PRAM intuition transfers; the *accounting* changes.

The bridge, made explicit. A PRAM algorithm of depth D simulates on BSP/MPC in $O(D)$ rounds (each PRAM level = local compute + one exchange). That is why low PRAM depth is *still* the right thing to engineer even when your target is Spark: depth upper-bounds your round count. But the converse discipline is new: an $O(\log n)$-depth PRAM algorithm that shuffles the whole dataset every level is a *bad* MPC algorithm; U5 dwells on exactly this divergence.

4.1 Pricing a round: the α-β term

A message of b bytes costs

$$T(b) = \alpha + \beta b$$

(F.E4 §1, where the size is written M; here M counts machines). Here α is the latency per message and β the seconds per byte, defined precisely and measured in F.E4 §2–2.1. An algorithm

with R rounds pays the **communication term**

$$\alpha R \;+\; \beta \cdot B_{\text{crit}},$$

where B_{crit} is the number of bytes on the critical path: the bytes of the slowest message of each round, summed over rounds.

Why a round costs α, not α per machine. Within one round the messages travel in parallel on different links. The round ends when its slowest message arrives, so R rounds pay about αR in latency, however many machines take part. This assumes each machine sends and receives $O(1)$ messages per round. When k messages converge on one receiver, that receiver's link serialises them, and the round costs at least $\alpha + \beta k b$, with the receiver then combining the k messages one by one. That is exactly why the all-to-one reduce is bad (§4.3).

4.2 Where R comes from

Count every point at which machines must exchange data before any of them can go on. These are communication rounds *across machines*; local work between two rounds pays no α.

- **The triptych.** Figure 1.1(c) has one round: the single dashed wire.
- **MapReduce and Spark.** One round per shuffle, i.e. per stage boundary. Sample sort has $R = 2$ (U2 §4.2).
- **Tree reduce.** A tree reduce over M machines has $R = \lceil \log_2 M \rceil$ (§4.3, U1 §4.1).
- **Distributed gradient descent.** Each of K steps ends with one all-reduce of the gradient. As a tree (reduce up, then broadcast down) an all-reduce is $2\lceil \log_2 M \rceil$ rounds, so $R = 2K\lceil \log_2 M \rceil$: R grows with the number of steps. The ring all-reduce trades more rounds for smaller messages (U18).

4.3 Worked: the eight numbers on four machines

Put $a = [3, 1, 7, 0, 4, 1, 6, 3]$ on $M = 4$ machines, two numbers each. Each machine sums its block; then the four partial sums climb a tree:

```
            M1          M2          M3          M4
block       [3,1]       [7,0]       [4,1]       [6,3]
local sum     4           7           5           9      1 add each
round 1       4 <-------- 7           5 <-------- 9      send 1 word
             11                      14
round 2      11 <------------------- 14                 send 1 word
total        25  (on M1)
```

Price it with one number as one 8-byte word:

- **Local work.** $n/M - 1 = 1$ addition per machine, all in parallel.
- **Rounds.** $R = \log_2 M = 2$, each sending one word and followed by one addition.
- **Critical-path bytes.** Round 1 sends *two* words (7 and 9), but in parallel on different links, so the path waits for one: 8 bytes. Round 2 sends one word (14): 8 bytes. So $B_{\text{crit}} = 16$ and the communication term is $2\alpha + 16\beta$.
- **Total traffic.** The network carries three words in all, 24 bytes. That is the load on the network, not the time.

Measuring time in additions, and writing w for the bytes of one partial sum, the general form is

$$T \;\approx\; \left(\frac{n}{M} - 1\right) \;+\; \lceil \log_2 M \rceil \,(\alpha + \beta w + 1).$$

Why the tree. The all-to-one alternative (Spark's plain `reduce`, U1 §4.1) sends $M-1$ messages into one machine, which combines them one by one. At $M = 1000$ that is $\lceil \log_2 1000 \rceil = 10$ tree rounds against 999 messages converging on one receiver.

4.4 Worked: the figure's two machines

Figure 1.1(c) splits the sum over two PRAM machines, $n/2 = 4$ numbers each. One word crosses the wire, then one final addition. Its caption, $T = \lceil \log_2 \frac{n}{2} \rceil + (\alpha + \beta) + 1$, counts β per word. It assumes each machine has at least $n/4 = 2$ processors, the width of its widest level, so that each machine's time is its depth. With p processors per machine, Brent on each machine gives

$$T \le \underbrace{\frac{n/2-1}{p}}_{\text{work per machine}} + \underbrace{\left\lceil \log_2 \frac{n}{2} \right\rceil}_{\text{depth per machine}} + \underbrace{(\alpha+\beta)}_{\text{one word on the wire}} + \underbrace{1}_{\text{final add}} .$$

processors per machine	local work	local steps (actual)	Brent bound, local	total time
$p = 1$ (each machine a RAM)	3	3	$3 + 2 = 5$	$3 + (\alpha + \beta) + 1$
$p = 2$ (the figure's case)	3	$2 = D$	$1.5 + 2 = 3.5$	$2 + (\alpha + \beta) + 1$
$p = 4$	3	$2 = D$	$0.75 + 2 = 2.75$	$2 + (\alpha + \beta) + 1$

On a real cluster α is microseconds and an addition about a nanosecond, so the wire dominates every row. But only the work term tells you how many processors per machine are worth buying: here, no more than two.

4.5 The State made explicit: DPRAM-JEM

The DPRAM prices the wire in time. It does not say whose law the wire crosses. Yet every machine stands on some State's territory, is owned or served by a company that some States can compel, may run a model trained under some State's influence, and is operated by a person whom some States can reach. The **jurisdiction-explicit model** (JEM) of the IAS paper makes these four facts coordinates of the machine (Sainudiin and an IAS Commons Continuum 2026, bb-v1.1.2, p. 120).

The enforcement leg of the fourth pillar. The microeconomics pillar (U00) stands on two legs. U16 designs contracts: who pays whom, on what evidence. Yet a contract pays only where some State can enforce it, against the party and over the resources it names; the JEM is that second leg, for it names which States can compel each owner and reach each principal. Thus, a contract and its JEM are read together, and neither stands alone.

Definition (AMJEM; the IAS paper's Dfn-AMJEM). Let A be one of this unit's three machines (RAM, PRAM, DPRAM), $R = R(A)$ its resource set (processors, memory cells, links), $\mathcal{J}$ a finite set of sovereign States, and t a time. An *abstract-machine jurisdiction-explicit model* is $J = (A, \mathcal{J}, \jmath)$, where the **sovereignty assignment** $\jmath = (\jmath_{\text{deploy}}, \jmath_{\text{own}}, \jmath_{\text{prov}}, \jmath_{\text{princ}})$ has four coordinates, named for *deployment*, *owner*, *provenance* and *principal*:

- $\jmath_{\text{deploy}}(r)$: the State in whose territory the machines serving resource r stand;
- $\jmath_{\text{own}}(r, t)$: the set of States that at time t can legally compel the company owning or serving r (empty for open weights that no one owns);
- $\jmath_{\text{prov}}(r)$: the set of States whose influence was fixed into r when it was built (for a model, its training origin); it never changes afterwards;
- $\jmath_{\text{princ}}(\omega, t)$: the set of States with coercive jurisdiction at time t over the person who conducts operation ω.

At A = DPRAM this is **DPRAM-JEM** (the paper writes DPRAMJEM). An instance is *State-pinned* to s, written DPRAMJEM-s, when $\jmath_{\text{deploy}}(r) = s$ for every resource r: for example, DPRAMJEM-Sverige (bb-v1.1.2, p. 121).

In words. The cost model asks *how fast*; the JEM also asks, for each operation, under whose ground, whose compulsion, whose provenance and whose reach over its operator it runs. Deploy is a coordinate of place, not of ownership: a US provider's region in an EU State has that EU State as $\jmath_{\text{deploy}}$, and the United States enters through $\jmath_{\text{own}}$, because US legal process can require a US provider to disclose data in its possession, custody or control wherever it is stored (the CLOUD Act, 18 U.S.C. §2713; the paper's case, p. 129). Figure 1.2 draws the five places a student's prompt can go: the small local server, a server rented on an EU-owned cloud, a US provider's EU region, a US vendor, and a PRC vendor.

Worked: the triptych's sum, placed three ways. Take Figure 1.1(c): eight numbers on two machines, one word on the wire, $T = \lceil \log_2 4 \rceil + (\alpha + \beta) + 1 = 3 + \alpha + \beta$. Machine 1 is your laptop in Sweden. Place machine 2 in three ways:

machine 2	$\jmath_{\text{deploy}}$	$\jmath_{\text{own}}$	the one word on the wire	the States that reach the sum
a Swedish-owned server in Sweden	SE	{SE}	stays in Sweden	Sweden (where EU law also applies)
a US provider's region in an EU State	that EU State	{US}	a transfer inside the EU	Sweden, that EU State, and the US through $\jmath_{\text{own}}$
a PRC provider in the PRC	PRC	{PRC}	a transfer out of the EU (GDPR Ch. V)	Sweden and the PRC

All three rows cost $T = 3 + \alpha + \beta$, with α larger the farther the wire runs: the cost model cannot tell the rows apart beyond that. The JEM can. Row 1 is DPRAMJEM-Sverige. Pinning fixes $\jmath_{\text{deploy}}$ only: had the Swedish server run an open-weights model trained elsewhere, $\jmath_{\text{prov}}$ would still name the training origin, and $\jmath_{\text{princ}}$ still names every State that can reach you.

Open weights on an EU-owned cloud. A model too large for a small local server can run on GPUs rented from an EU-owned cloud (for example Hetzner in Germany, OVHcloud or Scaleway in France; the dashed blue box of Figure 1.2). Take an open-weights model developed in the PRC (Qwen or DeepSeek; Lab 7's `qwen3:1.7b` on your laptop is the small case), download its weights once, and fine-tune it on your own data there. Then $\jmath_{\text{deploy}}$ is an EU State; $\jmath_{\text{own}}$ is the cloud's EU State, and empty for the weights, which no one owns; and $\jmath_{\text{prov}}$ contains the PRC, fixed at the base model's training. The fine-tuned model is a new artifact whose provenance holds the base model's and adds your data's (bb-v1.1.2, p. 121). Your data never leaves the EU, and the PRC reaches this machine only through provenance, through what the base model learned, not through compulsion: its developer can be compelled, but holds neither this copy nor your data. This is the converse of the red arrow: there the data stays in the EU and a foreign State reaches it through $\jmath_{\text{own}}$; here a foreign State's influence arrives in the weights and nothing flows back. Open models from the US (for example Llama or gpt-oss) and from the EU (for example Mistral) read the same way.

Which law binds which coordinate (pointers, not legal advice).

State	pointers	the coordinate it binds
USA	CLOUD Act (Pub. L. 115-141, div. V, 2018; 18 U.S.C. §2713)	$\jmath_{\text{own}}$: a US company's EU region is still reached
EU	GDPR (Reg. (EU) 2016/679), Chapter V on transfers out of the EU and Art. 48 on foreign orders; AI Act (Reg. (EU) 2024/1689), duties of providers and deployers	the links that leave the EU; $\jmath_{\text{deploy}}$ and $\jmath_{\text{princ}}$ for AI systems used in the EU

State	pointers	the coordinate it binds
PRC	Cybersecurity Law (amended text in force 1 January 2026); Data Security Law (2021); Personal Information Protection Law (2021), Arts. 38–40 on transfers abroad; Interim Measures for Generative AI Services (CAC Order No. 15, 2023)	$\jmath_{\text{own}}$ and $\jmath_{\text{prov}}$ of a PRC model; the links out of the PRC
Sweden	Lag (2018:218), the Swedish supplement to the GDPR; Offentlighets- och sekretesslag (2009:400), what a public body such as a university may hand to a provider	$\jmath_{\text{deploy}}$ and $\jmath_{\text{princ}}$ here; whether a course's data may leave

Modes. An operator dials the JEM per coordinate in one of three modes (p. 121): *agnostic* (jurisdiction is no constraint, but the coordinates are still there to log), *aware* (the States are computed and logged per coordinate), and *govern-by*, in two behaviours: **pin** (constrain $\jmath_{\text{deploy}}$ or $\jmath_{\text{own}}$ to chosen States) and **abide** (conform the operation's conduct to the applicable law). The paper names the modes γ, α, β; they are not this unit's α and β.

Responsible AI, made checkable: Dignum's ART. Virginia Dignum's ART (*accountability*, *responsibility* and *transparency*) names what a responsibly built AI system must exhibit (Dignum 2019, *Responsible Artificial Intelligence*, Springer; Dignum 2022, from principles to practice, arXiv 2205.10785). The IAS paper turns each into something that can be checked (Remark 3.23, bb-v1.1.2, p. 30): accountability is the σ-journal, every action recorded before it takes effect; responsibility is a legally responsible person who sets the purpose, with the machines and models as tools that person directs, never their owners; transparency is a stated source for every claim and a listed certificate for every formal result. The JEM adds the question ART leaves open, *under whose law?* Transparency asks for provenance, and $\jmath_{\text{prov}}$ names the States fixed into a model at its training. Responsibility names a person, and $\jmath_{\text{princ}}$ names the States that can reach that person. Accountability needs a record that can be demanded, and $\jmath_{\text{own}}$ names who can compel its holder.

Every agent that acts. *Agent* comes from the Latin *agere*, to act. An LLM agent, a robot arm, a humanoid and a drone are all agents in this sense: each is a resource set run on behalf of a principal, so the same four coordinates and the same ART apply. The JEM assumes no particular machine (Dfn-JEM: the resources may be "the premises, licences, inputs, and counterparties of a real-world operation", p. 120).

agent	$\jmath_{\text{deploy}}$	$\jmath_{\text{own}}$	$\jmath_{\text{prov}}$	$\jmath_{\text{princ}}$
an LLM agent on a US vendor's API	US (or the region)	US	US training	you, your State(s)
a humanoid robot in a Swedish care home, made abroad	SE	the maker's State, which can push its firmware	where its models were trained	the care provider's State(s)
a delivery drone, Malmö to Copenhagen	SE, then DK, in flight	the maker's or operator's State	its vision model's origin	the operator's State(s)

The drone shows what an embodied agent adds: it changes $\jmath_{\text{deploy}}$ while it acts, and the paper's $\jmath_{\text{deploy}}$ carries no time index, so a moving agent needs one; supplying that index is left as a project. Law pointers, not legal advice: unmanned aircraft, Reg. (EU) 2018/1139 (Arts. 55–57, Annex IX); robots as machinery, Reg. (EU) 2023/1230; and an AI system that is a safety component of a product under the EU product laws of the AI Act's Annex I can be high-risk (AI Act Art. 6(1)). Thus, the same three duties bind *any* agent: to name a person who answers for it, to record what it did, and to map whose law reaches each of its parts.

4.6 The legal analysis of the sovereignty of operations

The JEM names the States that reach an operation. What do their laws say, and can the operation meet them all at once? IAS answers this with its **JEM engine**, built in (Figure 1.3). Its first rule is the one a statistician would choose: *no model is the authority on what a law says* (the sovereignty-trilemma preprint, p. 2). Its second: the engine flags, a human decides.

1. **Fetch, from the source.** A registry lists each jurisdiction's official portal and how it serves its law (an HTML page, a PDF, EUR-Lex by CELEX number, or an API), with a *canary* law that a correct fetch must return, so a portal that changes is caught at the next run instead of silently returning nothing. Each instrument is fetched on two independent paths (the official portal, and a legal-intelligence gateway where the corpus is enrolled), the two texts are compared by SHA-256, and the text is archived by State, law and article with its verification status: dual-verified, single-path, or a gap.
2. **Type, before citing.** Each provision becomes a clause with a *binding type* (positive mandate, prohibition, or designation) and a *verification type*: DET (the verified text settles it), INT (a reading is needed, for example any English rendering of a PRC statute), or gap. A conclusion drawn from two clauses is only as firm as the weaker of their verification types (bb-v1.1.2, p. 128).
3. **Decide, by construct-or-refute.** An *architecture* assigns concrete behaviour to the regulated surfaces. A set Φ of clauses is *satisfiable* if some architecture complies with every clause; a *minimal unsatisfiable core* is an unsatisfiable $M \subseteq \Phi$ whose every proper subset is satisfiable. Satisfiability is shown by exhibiting a complying architecture, unsatisfiability by showing that none can exist.
4. **Sign.** A governance-weight decision is admissible only when the operator signs it with the operator's own key, never the machine's (the human-decision envelope, bb-v1.1.2, p. 127). Every fetch, hash, typing decision and verdict is a record on the append-only, hash-chained σ-log, so anyone can re-run the analysis.

Worked, at full size. The sovereignty-trilemma preprint tests a public claim that no global technology company can satisfy Chinese content control, European interoperability mandates and American decoupling policy at once. Four regulated surfaces of one product (what the model may generate, what must be opened to third parties, with whom the supply chain may transact, where data must reside) are the resources; the EU, the US and the PRC are the States. At the corpus snapshot the joint system is unsatisfiable, and its unique minimal unsatisfiable core is a single clause pair on one surface: the EU Digital Markets Act Art. 6(7), a mandate to open the operating system and the assistant to third parties, against the PRC Generative AI Measures Arts. 4 and 17, a mandate to control what the model generates. The core is typed DET on the EU side and INT on the PRC side, so the verdict is only as firm as the PRC reading. The feasible resolution is a family of architectures, one per State (the verdict, p. 5), and the preprint states what would falsify it (p. 6).

Register. This subsection is design and method, not a theorem: the check is done by hand, holds for one snapshot of the law, and carries no Lean certificate (the human-decision envelope is a design contract; only its approval-routing law is certified, p. 127). The registry has 118 jurisdictions, 8 of them fetch-verified end to end on 3 November 2026; the rest are mapped and labelled unverified.

Deep dive. The IAS paper, bb-v1.1.2 (PDF): the section *Law, jurisdiction and contracts*, Dfn-JEM and Dfn-AMJEM (p. 120); Dfn-DPRAMJEM and the jurisdiction modes (p. 121); the human-decision envelope (p. 127); deciding a legal claim, typed clauses and the minimal unsatisfiable core (p. 128); the CLOUD Act case (p. 129). The sovereignty-trilemma preprint (PDF): the method (p. 2), the clauses (p. 3), the verdict (p. 5), the falsifiers and the governance receipts (p. 6). Where the model began: DPRAMJEM: Math and Law (WASP summer school,

2025).

Check your reading.
Q6. Moving from PRAM to the distributed (DPRAM/MPC) lens, the NEW dominant cost that appears is:
(A) floating-point rounding error (B) communication: synchronous rounds and bytes moved between machines (C) the number of processors (D) compiler optimisation level

5. The invariant (memorise this box)

> **RAM:** how many ops? — $T(n)$, the work baseline. **PRAM:** how much work W, how deep D? — **Brent:** $T_p \leq W/p + D$; work-optimal first, then shrink depth; reduce + scan are the universal tools. **DPRAM/MPC:** how many rounds R, how many bytes, within local memory s? — map → shuffle → reduce; communication, not computation, is the scarce resource.

Every unit U1–U17 opens by instantiating this box for its algorithm.

5.1 The box, instantiated on eight numbers

Machine	Assumes	What you pay for	$n = 8$	general
RAM	one processor, one memory	operations T	$T = 7$ ($W = D = 7$, the chain)	$n - 1$
PRAM, $p = 1$	one processor, shared memory	work, depth	$W = 7$, $D = 3$, $T_1 = 7$ (Brent bound 10)	$T_p \leq W/p + D$
PRAM, $p = 2$	two processors	work, depth	$T_2 = 4$ (bound 6.5)	
PRAM, $p \geq 4$	enough processors for every level at once	depth	$T = D = 3$ (bound 4.75 at $p = 4$)	$D = \lceil \log_2 n \rceil$
DPRAM, $M = 4$	one processor each; 1 word per message	rounds, bytes, local work	1 local add; $R = 2$; $2\alpha + 16\beta$ on the critical path	$n/M - 1$; $R = \lceil \log_2 M \rceil$; $\alpha R + \beta B_{\text{crit}}$

The time column is what the schedule actually takes. Brent's $W/p + D$ is an upper bound on it, and $\max(W/p, D)$ a lower bound. On a real cluster α is microseconds or more, while one addition is a nanosecond or so: the network term, not the arithmetic, sets the price. Before quoting a cost, say which machine and how many processors. The same tree costs 7, 4 or 3 steps.

Check your reading.
Q7. Without looking: write the three questions the RAM, PRAM, and DPRAM lenses ask (one line each).

6. Lab (docker-compose; laptop-runnable; Canvas-submittable)

Lab 0 — toolchain + proof-checker (`labs/lab0-toolchain/`) has no algorithmic content: follow its README; `bash run-lab.sh` checks git, ssh and Docker, smoke-tests the course lab image, asks the Lean kernel to accept `1 + 1 = 2`, and writes your receipt (Canvas-submittable). The three experiments below are this unit's own; run and extend them in the same course lab image once Lab 0 passes. 1. **Sum & reduce three ways** — sequential fold; multi-threaded tree-reduce (watch $W/p + D$ in wall clock as you vary threads); Spark `treeReduce` vs `reduce` on a partitioned RDD — count the *stages* (= rounds) in the Spark UI and attribute wall-clock to shuffle vs compute. 2. **Measure the invariant** — for n from 10^6 to 10^9: plot T_p vs p against the Brent envelope $W/p + D$; find where speed-up saturates and explain it as $p \to W/D$. 3. **Break associativity** — floating-point sum in three different tree shapes; observe (and explain) the differing answers; fix with Kahan summation. Deliverable: the notebook + a half-page cost-accounting for each experiment *in the three-lens vocabulary* (that vocabulary being the actual thing under examination).

7. Lean certificate (proofs via our prover)

- **Shipped ✓:** `BMLSPar.lean` — the Brent scheduling bound for the level-by-level schedule, machine-checked (§3.4 gives its exact scope). Exercise U0.L1 (the Week-2 theorem-checker exercise, `provedog/exercises/wasp-module1-brent/`): read the proof top-down; complete the student copy `BrentBound_blank.lean`, whose whole inductive step is blanked by one `sorry` (with three one-line hints); check it with `check-brent.sh`. Exercise U0.L2 (deep): state work-optimality of tree-reduction and prove it.
- **Target (gap G1):** Graham's list-scheduling bound $T \le (2 - 1/p)\, T_{\mathrm{OPT}}$ (U3 will use it) — the next certificate in this line; students meet it as a *stated-but-uncertified* claim in U0 and see its certificate targeted in U3.

LLM lens — where this unit lands on the running workload (U0L):
LLM inference is this unit's three machines in production: decode on one node is one machine reading its weights (the RAM lens, priced in bytes read from memory, which the unit-cost RAM leaves out and U0L §2 adds); prefill is parallel matmul (the PRAM lens); and a model split across machines pays $\alpha + \beta M$ per crossing (the DPRAM lens) — U0L §2's tok/s ceiling is nothing but this unit's cost accounting applied to one workload. The dashed wire of Figure 1.1 is where U0L's three cuts (Figure 4.2) live.

8. Sources (credited; archived under `scadamale-book-sources`)

Brent (1974), *The parallel evaluation of general arithmetic expressions*, JACM 21(2). — JáJá (1992), *An Introduction to Parallel Algorithms*, Addison-Wesley. — Blelloch (1990), *Prefix Sums and Their Applications*, CMU-CS-90-190. — Valiant (1990), *A bridging model for parallel computation* (BSP), CACM 33(8). — Karloff, Suri, Vassilvitskii (2010), *A model of computation for MapReduce*, SODA. — Graham (1966), *Bounds for certain multiprocessing anomalies*, Bell System Tech. J. 45. — MIT OCW 6.172 / 6.046 / 18.337 (performance engineering; algorithms; parallel computing). — Wikipedia: *Parallel random- access machine*, *Analysis of parallel algorithms*, *Bulk synchronous parallel*, *Prefix sum*. — For §4.5–§4.6: Sainudiin and an IAS Commons Continuum (2026), *An interaction analysis of interacting agent systems*, bb-v1.1.2, vake.works/preprints (the jurisdiction-explicit model, pp. 120–129; Remark 3.23, p. 30). — Sainudiin and an IAS Commons Continuum (2026), *A jurisdiction-explicit verdict on the sovereignty trilemma*, vake.works/preprints. — *DPRAMJEM: Math and Law*, WASP summer school 2025, vake.works/SovScaDesDisMaLOps/2025. — Dignum (2019), *Responsible Artificial Intelligence*, Springer, doi:10.1007/978-3-030-30371-6. — Dignum (2022), *Responsible Artificial Intelligence: from Principles to Practice*, arXiv:2205.10785.

9. Exercises

Light (examined core): E0.1 Compute W and D for the balanced-tree sum of n values; state T_p by Brent; for which p is speed-up within 10% of linear? E0.2 A colleague's algorithm has $W = O(n \log n)$, $D = O(\log n)$ for a problem with $T_{\mathrm{seq}} = O(n)$. Is it work-optimal? For what regime of p does it still beat the sequential algorithm on wall clock? E0.3 In the MPC model with $s = O(n^{2/3})$: give an $R = 2$ algorithm that leaves the sum of n numbers on every machine and argue no correct algorithm achieves $R = 0$ communication. E0.4 Map each Spark notion onto the model: *stage*, *shuffle*, *partition*, *executor memory* $\to$ {round, all-to-all exchange, data shard, s}.

Deep (Module-3 material): E0.5 Prove the lower bound $T_p \ge \max(W/p, D)$ and exhibit a DAG where a greedy schedule is a factor $2 - o(1)$ from optimal. E0.6 (Simulation) Prove the PRAM$\to$BSP bridge: an EREW PRAM step with p processors on data of size $n \le p \cdot s$ simulates in $O(1)$ BSP supersteps with per-machine communication $O(s)$. E0.7 (Lean) Complete exercise U0.L2 and submit the checked certificate. E0.8 For the four-operation DAG of §3.2 (chain

$c_1 \to c_2$ plus x, y) on $p = 2$: list every greedy schedule, give its length, and check each against $\max(W/p, D)$, $D + (W - D)/p$ and $W/p + D$. E0.9 With $\alpha = 50\,\mu s$ and $\beta = 1$ ns/byte, price the tree reduce of §4.3 at $M = 1024$ for $w = 8$ bytes and for $w = 8$ MB (a gradient). Which term dominates in each case, and where is the knee?

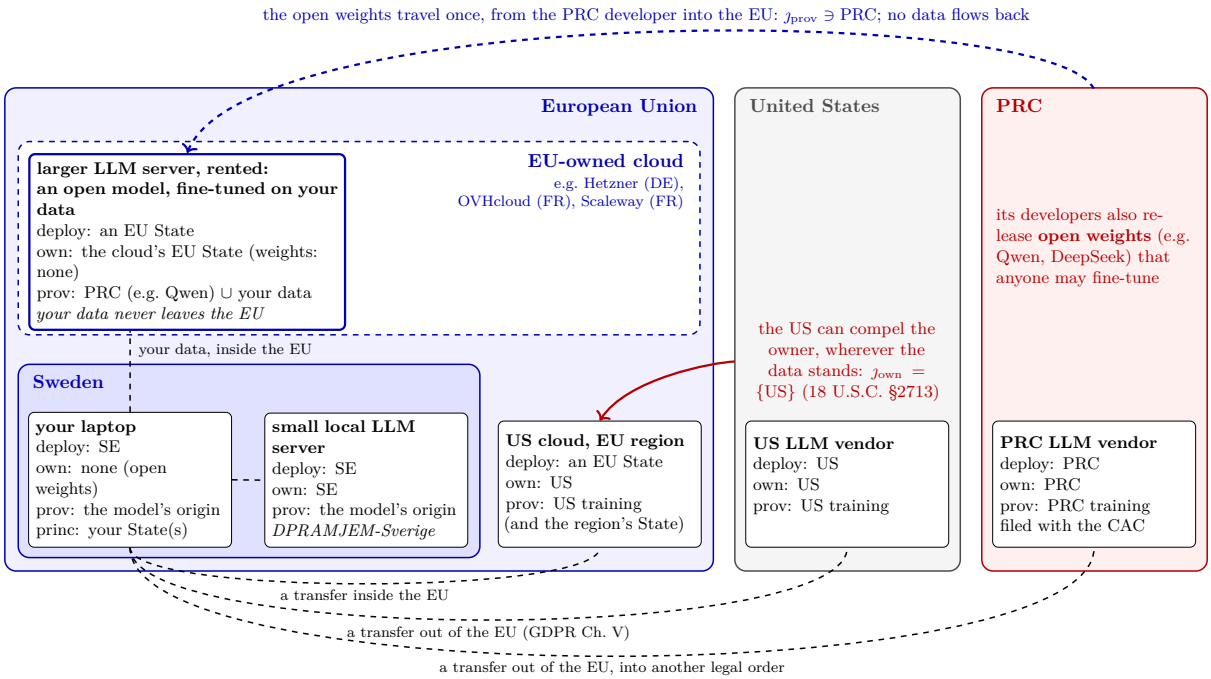


Figure 1.2: The DPRAM of Figure 1.1(c) with the State made explicit (DPRAM-JEM, §4.5). Each machine is a resource r and carries its sovereignty assignment $\jmath$: $\jmath_{\text{deploy}}(r)$, the State whose territory it stands in (the shaded region); $\jmath_{\text{own}}(r,t)$, the States that can compel whoever owns or serves it; $\jmath_{\text{prov}}(r)$, the States fixed into it at construction (a model's training origin); and $\jmath_{\text{princ}}$, the States that reach the person who runs the operation. The dashed wires are the triptych's network, now priced twice: $\alpha + \beta b$ in time, and a change of legal order whenever a wire leaves a region. The red arrow is the case the coordinates exist to separate: a US provider's EU region stands in an EU State, yet the United States reaches it through $\jmath_{\text{own}}$. The dashed blue box is the converse: a larger LLM server rented on an EU-owned cloud, running a PRC-developed open-weights model fine-tuned on data that never leaves the EU, which the PRC reaches only through $\jmath_{\text{prov}}$. Pinning every machine to Sweden gives DPRAMJEM-Sverige. Such a server is what one rents when a model is too big for the small local server; the model's developer can be compelled, but holds neither this copy nor the data, and Lab 7's `qwen3:1.7b` is the same case, smaller. Law pointers, not legal advice. The law each region carries: in the European Union, the GDPR (Ch. V, Art. 48) and the AI Act; in Sweden, Lag (2018:218) and OSL (2009:400); in the United States, the CLOUD Act; in the PRC, the CSL, the DSL, the PIPL and the GenAI Measures.

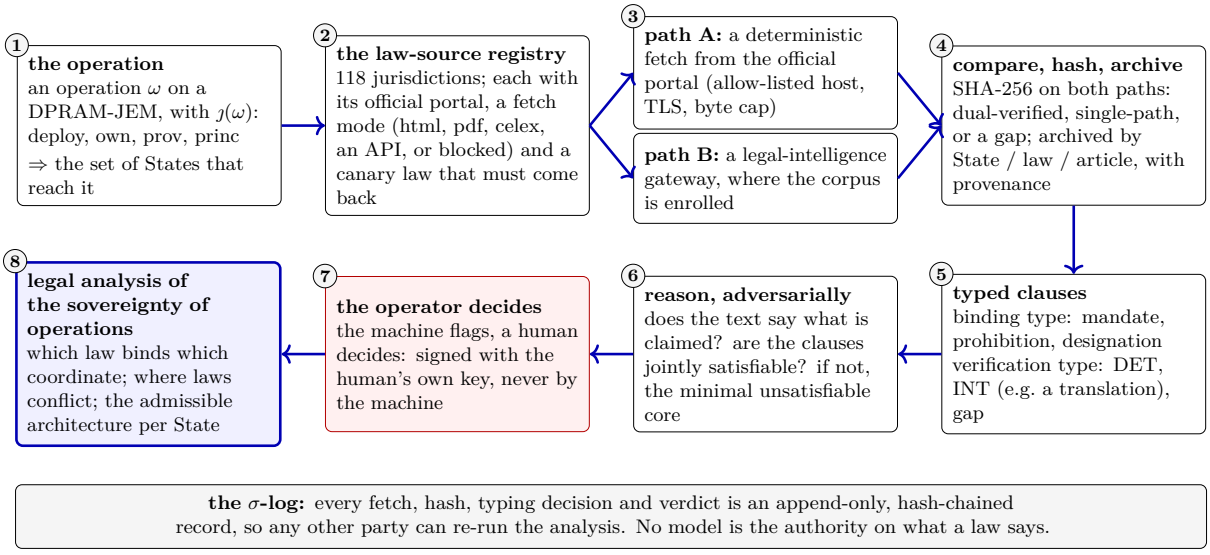


Figure 1.3: How IAS reads the law for an operation: the IAS JEM engine (§4.6). Steps 1–4 fetch: the operation's sovereignty assignment names the States that reach it (Figure 1.2); a registry gives each State's official source and how to fetch it, with a canary law that must come back, so a changed portal is caught; every instrument is fetched on two independent paths, compared by hash and archived with its verification status. Steps 5–8 decide: each provision becomes a typed clause, the clauses are checked for joint satisfiability by construct-or-refute, and the operator, not the machine, signs the decision. The registry has 118 jurisdictions, 8 of them fetch-verified end to end (3 November 2026); the others are mapped but not yet verified, and are labelled so.

Chapter 2

00A four pillars

U00 — What scalable data science and distributed machine learning *is* today

The ScaDaMaLe book, unit 00 (examined core, week 1, day 1). Our own exposition; sources credited in §5.

The one idea of this unit: *the subject you are starting is ONE subject, not four.* It looks like a pile of disconnected skills — some measure theory here, some Spark configuration there, a proof about schedulers, a paper about incentives. This unit gives you the map that makes it one thing, so that every later unit lands in a place you already know exists.

Entry kit — what this chapter assumes; follow a link if you need the memory jog (map: Figure 1):
F.X1 (utility, incentives, contracts — the fourth pillar's kit)

1. The four pillars

Modern data science and machine learning stand on four load-bearing pillars:

1. **Mathematics** — linear algebra, optimisation, the analysis that makes "convergence" a theorem rather than a hope. In this book: U6–U9, U19.
2. **Statistics** — inference, learning theory, uncertainty quantification; what it means to *know* something from data, and at what rate. In this book: U4, U14, U15, U20.
3. **Computer science / algorithms** — the models of computation and the algorithms whose costs we can account for and certify. This is the book's central column: U0–U5, U11–U13 — the classical parallel-and-distributed canon.
4. **Microeconomics — incentives, markets, and contracts.** The newest pillar, and the one most often missing from technical curricula. Data is produced, shared, and acted on by *self-interested parties*: labs, firms, platforms, and increasingly machine agents. Whether a prediction is *good* is inseparable from whether anyone had an incentive to make it honestly, and whether the guarantee attached to it is one a counterparty can collect on. This is also the pillar that turns a model into a *product*: what you promise a buyer (a service-level guarantee), how you price it, how that price should move as evidence accumulates, and how you contract for the data and platforms it runs on are all questions of incentives and contracts — the mathematical spine of what industry calls "AI product management" (U16 §5.3). The formal home of these questions is **statistical contract theory (SCT)**, and this course introduces it on day one — first as a *lens*, then (U20 → U16) as theorems.

This four-pillar view of the subject is closely associated with Michael I. Jordan's program of "melding" statistical inference with economic mechanism — argued in his widely read essay *Artificial Intelligence — The Revolution Hasn't Happened Yet* (Harvard Data Science Review,

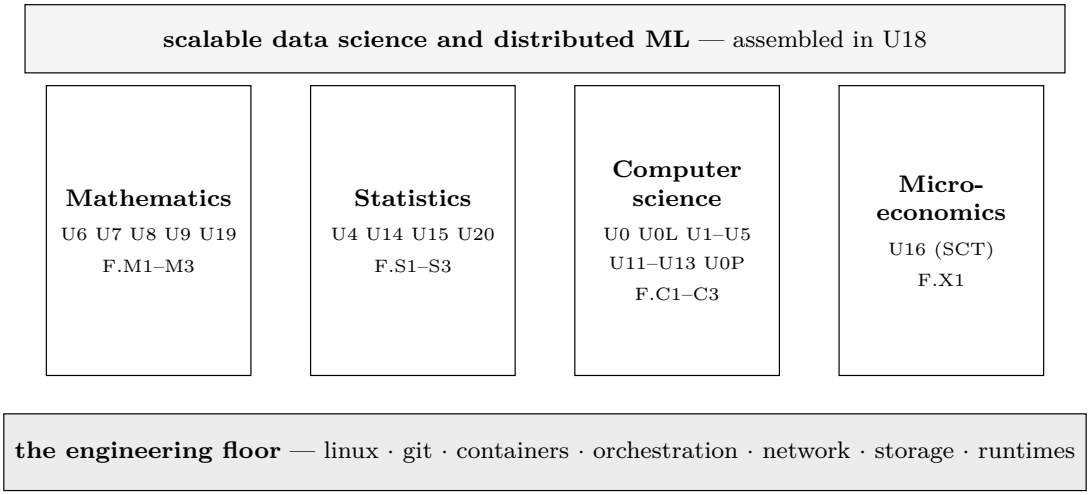


Figure 2.1: The subject's load-bearing structure, which is also this book's map: four pillars on an engineering floor, carrying the modern synthesis. Each pillar lists the units that build it and the Foundations chapters (Part F) that backfill it — the same correspondence the hyperlinked atlas draws as a graph, here drawn as architecture. The fourth pillar is the one most curricula omit; this course installs it on day one.

2019) and developed technically in the statistical-contract-theory line of work we study in U16. The framing resonated with your instructor first-hand at Jordan's plenary at a data-science conference in Portugal at the end of 2023; the course's opening landscape descends from the workshop material (`sovscadesdismalops`) built in that spirit.

Jordan's most recent statement of the programme, *A Collectivist, Economic Perspective on AI* (arXiv:2507.06268, 2025), sharpens why this pillar is not optional. Its thesis: AI is best understood not as isolated cognition but as **collectives of interacting agents mediated by markets and incentives** — recommendation systems, data markets, and multi-layered platforms are the real objects, and reasoning about them requires blending three modes of thought at once: the *computational*, the *inferential* (statistical), and the *economic*. The essay also names the gap this course exists to close — its §6 is titled *"The Missing Middle Kingdom in AI Education"*: technical curricula teach computation and inference but omit the economic middle, so graduates can build models yet cannot reason about the incentives, contracts, and prices that govern them in the world. The fourth pillar is that missing middle; SCT (U16) is its mathematics.

And the pillars stand on an engineering floor. None of the four pillars runs itself: cloud substrates, containers and orchestration (docker → kubernetes; the weekly Labs 0–7 start in docker), distributed file systems and databases, and security are what make the theorems *executable*. This book refuses the usual separation between "the maths" and "the infrastructure": the same course that proves Brent's theorem has you run it on a cluster, and the proof-checker that certifies the theorem is itself a container you operate (U0P). That refusal is the **mathesis** ethos of the book.

Figure 2.1 is the whole book's load-bearing structure at a glance.

Check your reading.

Q1. A vendor claims its model is "good". Deciding whether anyone had an *incentive* to make that claim honestly, and whether the attached guarantee can be collected on, belongs primarily to which pillar? (A) Mathematics (optimisation and linear algebra) (B) Statistics (inference and uncertainty) (C) Computer science (models of computation) (D) Microeconomics (incentives, markets, and contracts)

Q2. In this book's structure, containers, orchestration, storage, and networking are described as: (A) a fifth pillar, equal to the other four (B) the engineering floor the four pillars stand on, taught so the theorems are executable (C) optional industry context, separate from the examined material (D) the responsibility of the cloud provider, outside the course

Q3. Pick any deployed ML system you know. In two sentences, name one question about it that each of the four pillars would ask (four questions total).

2. Statistical contract theory — the day-one first look

Here is the lens, in one worked thought.

A hospital consortium wants a model that predicts sepsis twelve hours out. Three things are true at once: (i) the *statistical* question — does the model meet a sensitivity target at a controlled false-alarm rate? — is a hypothesis-testing problem (Pillar 2); (ii) the *computational* question — can it be trained across twenty hospitals' data without moving that data — is a distributed-systems problem (Pillars 1+3, U9/U19/U20); and (iii) the *economic* question — why should the vendor train it honestly, and what can the consortium *verify* rather than trust? — is a contract problem (Pillar 4).

SCT's move is to make (iii) mathematical: attach to the model a **statistical guarantee that functions as contract collateral** — a claim precise enough to test, with error rates both sides agreed on, and with payments or acceptance decisions that make honest effort the vendor's best strategy. Three course landmarks build this out:

- **U20** proves a *distribution-free* coverage guarantee (split conformal prediction) — a promise that can be made without trusting anyone's modelling assumptions;
- **U16** develops the contracts themselves — principal–agent hypothesis testing, statistical contracts, markets of learning agents (the Jordan-school papers);
- **U18** closes the arc: when the "vendor" is an autonomous machine agent, the incentive questions do not go away — they intensify. (And the book's Part V shows the same mathematics reappearing in the governance of the agent systems that helped write it.)

For today, the takeaway is the habit: **every deployed model is implicitly a contract.** The rest of the course gives you the tools to make the contract explicit — and machine-checkable.

2.1 A second worked lens: one spam filter, four questions

The same decomposition works on the smallest system you use daily. A spam filter, interrogated by each pillar:

Pillar	The question
Mathematics	Does the training objective have one optimum, and does the training procedure reach it?
Statistics	What is the false-positive rate on *fresh* mail, with what confidence interval?
Computer science	Can scoring keep up at peak volume, and at what cost per message?
Microeconomics	Does the vendor's pricing reward under-reporting of errors?

Most real failures happen in the pillar nobody was watching. The fourth question is worked in numbers in F.X1 §3.1.

Check your reading.

Q4. What does a *statistical contract* add to an ordinary accuracy claim about a model?
(A) a better optimiser, so the accuracy is higher (B) a payment rule tied to a verifiable statistical outcome, agreed before the work (C) a larger benchmark suite (D) a legal disclaimer that no guarantee is offered

3. The cost-lens habit (bridge to U0)

The book's second habit, introduced properly in U0 and drilled in every unit after: *the same algorithmic problem is re-solved as the machine model gains a resource; what changes is the cost you must account for.* Operations on one machine; work and depth on a shared-memory machine; rounds, communication, and local memory on a cluster; bits-for-accuracy when the object shipped is an *estimator* (U20). When you can state a system's behaviour in those currencies — and point to the theorem, ideally CERTIFIED (U0P), that prices it — you are doing the subject, not reciting it.

Check your reading.

Q5. The course's recurring method prices every algorithm on three machines. Which triple names the three lenses?
(A) CPU, GPU, TPU (B) laptop, cluster, cloud (C) RAM (operations), PRAM (work and depth), DPRAM (rounds and communication) (D) Python, Scala, Haskell

4. How to take this course

Flipped classroom: read the unit and watch the cast *before* the meeting; the meeting is a lab. Weekly quizzes cover the examined units (U00, U0, U0P, U1–U5, U6, U9, U12, U18); one bounded group project (Module 3) goes deep on a seed of your choosing; everything runs in docker on your laptop first (the weekly Labs 0–7, in the course lab image), and on the course cluster when it must. Every theorem carries an honesty flag — CERTIFIED / STATED / UNCERTIFIED (U0P §5) — and you can always run the checker yourself.

LLM lens — where this unit lands on the running workload (U0L):
The four pillars, read on the running workload of U0L: the model's substance is matrix algebra (Pillar 1, U6); its output is a fitted conditional distribution and its training objective a log-loss (Pillar 2, U15); its cost is parallel-and-distributed computation (Pillar 3, U0/U0L §2); and who pays per token, and what a quality guarantee is worth, are contract questions (Pillar 4, U16). One artifact, four pillars, day one.

Check your reading.

Q6. In one sentence each: (a) what should you do BEFORE a course week, and (b) what should you do when a chapter's Entry Kit lists a foundation section you don't recognise?

5. Sources

- M. I. Jordan (2019). *Artificial Intelligence — The Revolution Hasn't Happened Yet.* Harvard Data Science Review 1(1). Open access.
- M. I. Jordan (2023). *Statistical Contract Theory* — tutorial lectures, CIRM, Marseille. (The technical papers behind it are the U16 source list: Bates–Jordan–Sklar–Soloff; Liu–Mania–Jordan.)
- Our course workshop landscape: the `sovscadesdismalops` materials (VakeWorks, 2026).
- The IAS mathesis paper, §8.5 (statistical contract theory for operator–agent pairings) — the Part-V ladder's SCT rung.

6. Exercises

E00.1 Take one data-driven system you use daily. Write four sentences — one per pillar — stating the mathematical, statistical, computational, and economic question it embodies. (No wrong answers; graded for the *decomposition*, not the system.)

E00.2 For the sepsis example of §2: state precisely what the consortium can *verify* about the delivered model without access to the training data or code. Which of the three questions (i)–(iii) does your verification protocol address, and which does it leave to trust?

E00.3 (Deep, revisit after U16.) Read §2 again and write down, in the notation of U16 §3, who the principal is, who the agent is, what the observable is, and what the reward rule could be, for the sepsis contract.

Chapter 3

00K os foundations

U00K — The operating system, the kernel, and how to prove things about system behaviour

The ScaDaMaLe book, preliminary unit K (engineering floor). Our own exposition; sources credited in §7.

The one idea of this unit: *every operation in scalable data science and distributed machine learning runs, in the end, as ordinary programs on an operating system — so the correctness and cost guarantees we prove about algorithms silently rest on guarantees about the system underneath. This unit makes those system-level guarantees explicit and shows how — exactly as for algorithms in U0P — we can replace "we hope the system holds" with a machine-checked proof, honest about the small base we must still trust.*

A distributed reduction that assumes exclusive access to a buffer is simply *wrong* the instant another process can scribble on that buffer; a shuffle that assumes its temporary files are private leaks the moment they are not. An algorithm's guarantee is only as sound as the system it runs on — the kernel is the referee that decides what every program may do to memory, files, and the network. And in an age where a cheap AI can *be* the process that scribbles, a system-level claim you can only assert is worth far less than one a machine can re-check.

Entry kit — what this chapter assumes; follow a link if you need the memory jog (map: Figure 1): U0P (proof-checkers: what CERTIFIED / STATED / UNCERTIFIED mean, and why a small kernel checks everything) and U00N (the no-black-magic habit of tracing a real system by hand).

1. Acronyms, expanded (the master table — none are assumed)

We keep the book-wide habit: **every acronym is expanded the first time it appears.** This table is the master list for the operating-systems floor; it extends the networking floor of U00N.

Acronym	Expansion (one line)
OS	Operating System — the program that shares one machine's hardware among many programs
kernel	the privileged core of the OS; the only code with direct hardware access
syscall	System Call — the controlled entry from an ordinary program into the kernel
UEFI	Unified Extensible Firmware Interface — modern motherboard firmware (successor to the BIOS)
BIOS	Basic Input/Output System — the legacy motherboard firmware
TPM	Trusted Platform Module — a tamper-resistant chip that measures the boot chain
PCR	Platform Configuration Register — an append-only hash register inside the TPM
uid / gid	User / Group IDentifier — the numeric owner of a process or file (uid 0 = root)
DAC	Discretionary Access Control — ordinary owner/permission checks (`rwx`)
MAC	Mandatory Access Control — a policy the owner cannot override (AppArmor, SELinux)
LSM	Linux Security Module — the kernel hook layer MAC systems plug into
VFS	Virtual File System — the kernel layer that makes many disks look like one file tree

Acronym	Expansion (one line)
inode	index node — the on-disk record of a file's owner, mode, and data pointers (a *name* points to an inode)
cap	capability — a fine-grained slice of root's power (e.g. `CAP_NET_ADMIN`) that can be dropped
seccomp	SECure COMPuting — a per-process filter that forbids chosen system calls
Landlock	an unprivileged LSM letting a process restrict its own filesystem/network access
namespace	a kernel-virtualised view of a global resource (mounts, network, PIDs, users)
IMA / EVM	Integrity Measurement Architecture / Extended Verification Module — kernel file-integrity checking
TCB	Trusted Computing Base — the parts you do not prove but must assume correct
SMT	Satisfiability Modulo Theories — the automated engine behind bounded model-checkers

2. The stack, bottom to top — and the chain of trust

A running computer is a stack of layers, each started and (ideally) checked by the one below. Read it from the silicon up.

1. **Firmware** (UEFI, formerly BIOS) — the very first code the processor runs at power-on, stored in a flash chip *on the motherboard*, not on the disk. It wakes the memory and buses and then loads a bootloader. Individual devices (a network card, a Thunderbolt controller) carry their own small firmware too; these are updated from the running OS but are not part of it.
2. **The TPM** — as each layer loads, its hash is *extended* into a PCR: a register you can only add to, never rewrite without a reboot. The TPM can therefore later *attest* "exactly this firmware and this kernel booted, unmodified." It is a hardware root of trust: tamper-resistant silicon.
3. **Secure Boot / Measured Boot** — the firmware checks a cryptographic signature on the bootloader and kernel before running them (Secure Boot), and records their measurements in the TPM (Measured Boot).
4. **The bootloader** (e.g. GRUB) — a small program the firmware loads from disk whose one job is to find the kernel, load it into memory, and start it.
5. **The kernel** — the subject of §3.
6. **Userspace** — once the kernel is up it starts the first ordinary process (on Linux, `systemd`, process number 1), which launches everything else you think of as "the computer".

The arrows only point one way: each layer is verified by the layer *below* it. That is the chain of trust, and it is the reason firmware and the TPM matter for security even though you never program them directly.

3. What the kernel is, precisely

3.1 Two worlds: kernel mode and user mode

The processor runs in one of (at least) two privilege levels. In **kernel mode** code may touch any hardware, any memory, any device. In **user mode** — where *every* program you run lives, from a text editor to a database — code may touch only its own memory and must *ask the kernel* for anything real: to read a file, open a socket, start another program. The kernel is the referee that stands between all those requests and the hardware.

3.2 The one door: system calls

The only way a user-mode program crosses into the kernel is a **system call**. There are a few hundred of them (`open`, `read`, `write`, `mmap`, `execve`, `mount`, `unshare`, `connect`, …). Everything a program does to the outside world is one of these calls. This is the single most useful fact

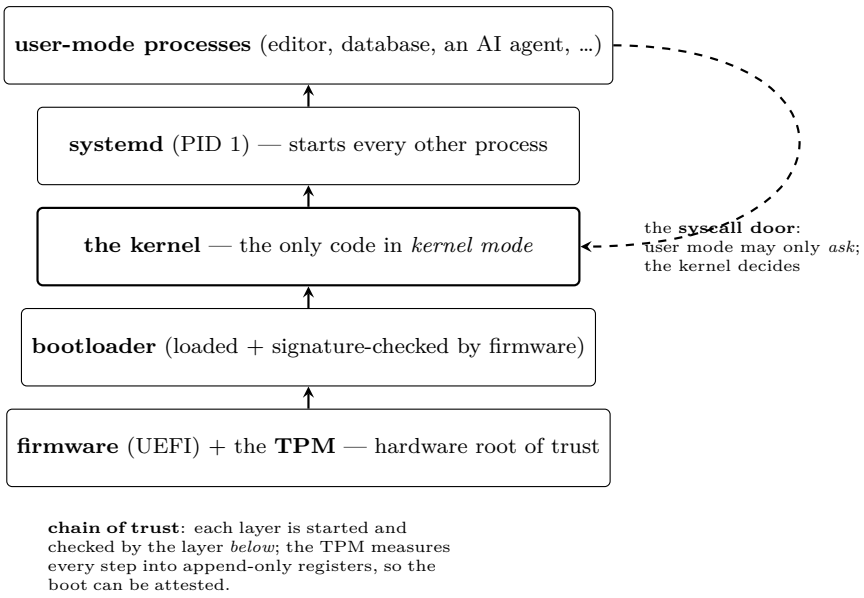


Figure 3.1: The stack, bottom to top. Firmware and the TPM are the hardware root of trust; each higher layer is loaded and verified by the one below it. Every ordinary program — including an AI agent — runs in user mode and can act on the world only by asking the kernel through the system-call door. Firmware and the TPM are never programmed directly, yet they anchor everything above: that is why they sit in the trusted base of §4.

for reasoning about security: *if you understand the syscall boundary, you understand the entire surface through which a program can act.* A confinement is exactly a restriction on which syscalls (with which arguments) the kernel will honour for a given process.

3.3 What the kernel manages

- **Processes and scheduling** — it decides which program runs on which processor core and when; each process has a uid that says who it acts as.
- **Memory** — each process gets its own *virtual* address space; one process cannot read another's memory. This isolation is enforced by the hardware's memory-management unit under kernel control.
- **Filesystems** — through the VFS the kernel turns raw disks into a tree of files and directories. Each file is an *inode* carrying an **owner** (uid), a **mode** (the `rwx` permission bits), and optional flags. A *name* in a directory merely points at an inode; two names can point at the same inode (a hard link). Path lookup walks the directory tree one component at a time (`fs/namei.c` in the Linux source) — which, as §4 will show, is exactly where a careless permission can be turned into an attack.
- **Drivers** — the code that actually speaks to each piece of hardware.

3.4 The authority mechanisms (how the kernel says "no")

Modern kernels give a program *less* than full root, in layers that a well-built sandbox stacks together:

- **DAC**: the classic owner + `rwx` checks. Only uid 0 (root) bypasses them — unless root's power has been sliced away by:
- **Capabilities**: root's authority is split into ~40 *capabilities* (`CAP_NET_ADMIN`, `CAP_SYS_ADMIN`, `CAP_LINUX_IMMUTABLE`, …). A process can be launched with most of them permanently **dropped** — an irreversible, kernel-enforced restriction.

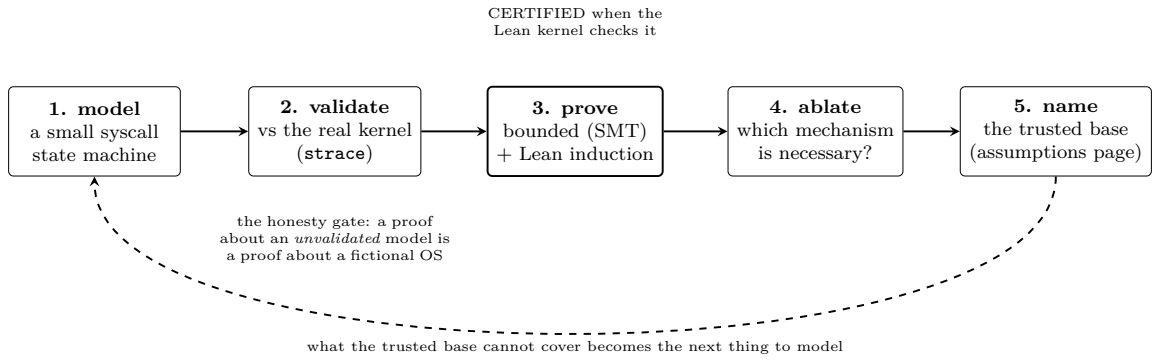


Figure 3.2: Proving a property of system behaviour, in five steps. Model only the slice the claim touches; validate that model against the real kernel before trusting it; prove the property two ways that must agree; ablate to learn which mechanisms are load-bearing; and state the trusted base honestly. The result is not "nothing can go wrong" — it is a small, written-down set of assumptions with everything above it re-checkable.

- **The immutable attribute** (`chattr +i`): a file flag such that *no* process may write the file — and only a process holding `CAP_LINUX_IMMUTABLE` may clear the flag. Drop that capability and the file is write-proof for that process, whatever its uid.
- **seccomp**: a filter installed by the process itself that makes chosen syscalls simply fail (`unshare`, `mount`, `ptrace`, …). Once installed it can only be tightened, never loosened.
- **Landlock**: an unprivileged module a process uses to fence its own filesystem and network reach.
- **Namespaces**: give a process its own private view of mounts, the network, or the user table — the mechanism containers are built from.
- **IMA / EVM**: the kernel measures (and, in appraisal mode, *refuses to read*) files whose signed hash does not match — integrity enforced below the application.

Every one of these is a fact *about the kernel's behaviour*. And a fact about behaviour is something we can model and prove.

4. How to prove things about system behaviour

The rest of the book proves theorems about *algorithms*. Here we do the same for a *system*. The method is general; it is worth learning as a transferable skill in its own right.

Step 1 — Model the relevant slice. You do not model the whole kernel (millions of lines); you model the tiny subset your claim touches. To reason about "can this process write that file?", the model needs only: inodes (owner, mode, immutable), processes (uid, which capabilities they hold), and the handful of operations that change them (`write`, `chattr`, `chown`, `rename`, `open`). Write it as a **state machine**: a state, and a transition for each operation with its precondition (the kernel's own check).

Step 2 — Validate the model against reality. A proof about a model that does *not* match Linux proves something about a fictional operating system. So before proving anything, you *run* the real syscalls (with a tracer such as `strace`) and confirm the model predicts the same outcome the kernel actually produces. This is **differential testing**, and it is the load-bearing honesty gate: it earns the right to reason about the model instead of the kernel.

Step 3 — Prove the property. Now state the claim and check it two ways that must agree: - a **bounded** check, with an SMT-backed model-checker, that no bad state is reachable within

(say) thirty operations — fast, and it hands you a concrete counterexample when the property is false; - an **unbounded** proof, in a proof assistant (Lean, from U0P), by induction over *all* operation sequences.

The book's honesty flags carry straight over: a property is **CERTIFIED** when the Lean kernel has checked it, **STATED** when the statement is precise but not yet mechanised, **UNCERTIFIED** when it is only argued informally.

Step 4 — Ask which assumptions are load-bearing (ablation). Remove one mechanism from the model and re-run the check. If the property still holds, that mechanism was *redundant* for this claim; if the check now finds a counterexample, the mechanism is *necessary*. This turns "we added a lot of security settings" into a precise ledger of which ones actually do the work.

Step 5 — Name the trusted base honestly. No proof stands on nothing. Yours rests on a **TCB** you did not prove: the hardware, the TPM and firmware, the kernel *outside* the slice you modelled, and the proof-checker itself. The value of the exercise is not "nothing can go wrong" — it is that you have *shrunk* the set of things that must be trusted to a small, written-down list, and made everything above it a re-checkable computation. Even the largest verified kernels in the world ship exactly such an assumptions page.

4.1 A worked micro-claim

Here is a claim small enough to prove and useful enough to matter: *a uid-0 (root) process that has dropped `CAP_LINUX_IMMUTABLE` cannot, by any sequence of the modelled file operations, write to a root-owned file that has `chattr +i` set.* (For an unprivileged process the claim is true but dull: ordinary DAC already refuses the write, and changing the flag also requires owning the file or holding `CAP_FOWNER`. Root is the case where the capability-drop does the work.)

The argument is a one-line **invariant**: the target's immutable flag stays set (only `chattr -i` could clear it, and that needs the capability the process lacks), and writing an immutable file is refused by the kernel — so across every reachable sequence, no write to the target ever succeeds. Bounded model-checking finds no counterexample; the induction proves it for all sequences; removing the capability-drop from the model immediately produces the counterexample `[chattr -i ; write]`, confirming the capability-drop is *necessary*. That is the whole method in miniature: model, validate, prove both ways, ablate, and state the floor.

4.2 The course's own algorithms depend on this

This is not a detour from scalable data science — it *discharges preconditions those algorithms quietly assume*. Take the parallel sum-reduction of U01 on a shared array $A[1 \dots n]$ with p workers, in the EREW PRAM model (Exclusive Read, Exclusive Write). PRAM is a *closed* model — its only processors are the algorithm's own p workers — and its correctness and $O(n/p + \log p)$ cost bound quietly rely on that closedness: *during the reduction, no writer outside the p cooperating workers touches A.* Note this is a *distinct* assumption from the EREW discipline itself: EREW is *intra*-worker (the p workers do not collide with one another on a cell); the closedness is *extra*-worker (no process outside the worker set writes A at all), and it is the closedness that a real machine must actually deliver. On a real shared node — a multi-tenant cluster, a cloud VM, an agent-orchestrated pipeline — that is not a given; it is an **OS-level guarantee**. If a co-tenant process, or a stray agent sharing the machine, can write A mid-reduction, the EREW proof's hypothesis fails and the sum is *silently* wrong — no crash, just a wrong number that every downstream step trusts.

So the algorithm's guarantee carries an OS-level precondition: *the shared region is writable only by the worker set.* That is exactly a U00K-style claim — model the region's owner and

permissions and the set of processes holding write access; prove no out-of-set process can write it (a private mapping, or a shared mapping with every other process's write capability dropped). Proving the algorithm and proving its OS precondition are the *same discipline at two layers*: the reduction's $O(n/p + \log p)$ bound and "no outsider can write A" are both theorems, and the first is only true when the second is. That is the honest way the course's subject and its foundations meet.

5. Why this matters now — the agentic age

For decades this kind of proof was a specialist luxury: writing a real privilege-escalation exploit was slow and rare, so "best-effort sandboxing and hope" was a tolerable posture. That calculus has changed. Capable, inexpensive AI has made reconnaissance, exploit drafting, and attack orchestration cheap and available to almost anyone. When the *attacker's* side is automated and democratised, a defender's "we think the sandbox holds" is no longer good enough — the defence itself needs to be **checkable, not merely asserted.**

A machine-checked confinement proof is precisely that. It converts "an untrusted program cannot escape" from a hope into a reproducible computation — true on every machine, with the assumptions written down. It does not make a system unbreakable (a bug in the trusted base, or a flaw outside the modelled slice, still bites). What it *does* is shrink and document exactly what you are trusting, and make the rest falsifiable. In an age of cheap attacks, that honest, minimised trusted base is the realistic standard of care — and the skill of building it (model → validate → prove → ablate → name the floor) is more durable than any single tool.

And to be plain about scope: this chapter is not a security course annexing cloud computing, or reinforcement learning, or systems programming — each a deep field in its own right, taught by people who spend careers on it. It is the older observation that *every* correctness or cost theorem about a running computation carries silent preconditions about the machine underneath it: isolation, atomicity, non-interference, integrity (§4.2 made one such precondition explicit for a PRAM reduction). A scientist doing scalable data science should be able to *see* those preconditions and, when the stakes warrant, *discharge* them — not to colonise another discipline, but because they are load-bearing for the guarantees this course already claims. Security, systems, and learning are not separate fiefs here; they meet in the one question this book asks of every claim — *what exactly did we assume, and did we check it?* — and that question does not stop at the boundary of the algorithm. The students, at least, need to know it does not.

> **LLM lens** — where this unit lands on the running workload (U0L):
> An AI assistant is, from the kernel's point of view, just another user-mode process making system calls — and, from the attacker's point of view, a cheap way to generate those calls in hostile combinations. Both readings point the same way: the guarantees you can rely on are the ones the kernel *enforces* and you have *checked*, not the ones you asked the assistant to respect. This unit is the foundation for the sequel exercises where you prove a confinement property of an agent-orchestrated system, or show that an isolation assumption a parallel algorithm quietly relies on actually holds at the OS level.

6. Open project ideas (for group projects)

None of these is examined; all are open, and each is a good group project that ends in a *checkable* artifact — a small model, a validation trace, and a bounded and/or Lean proof, with an honest assumptions page. Scope any of them to the time you have; the deliverable is the honest method, not exhaustive coverage.

P00K.1 **The PRAM precondition (recommended first project).** Take one algorithm from U01–U06 (the EREW reduction of §4.2, a prefix-scan, or a shared-memory matmul tile). State precisely the OS-level precondition its correctness/cost proof assumes (exclusive access,

private temporaries, no torn reads). Model the shared region's ownership + the writer set; validate against a real run with `strace`; prove "no out-of-set process can write the region" bounded, and state the trusted base. Deliverable: the algorithm's guarantee *and* its discharged OS precondition, side by side.

P00K.2 **Confine a worker, then prove it.** Take a distributed-ML worker (a Spark/Ray/Flink executor from the labs). Write a systemd unit or container spec that drops capabilities, restricts syscalls (seccomp), and fences the filesystem (Landlock). Model the confinement; prove the worker cannot read outside its data directory or reach the network except through one endpoint; ablate to find which restriction is load-bearing.

P00K.3 **Broker-only egress for an agent pipeline.** Model an orchestrated set of programs where no member may reach the network except through a single audited broker. Prove that if each member lacks network capability and the broker validates, every outbound effect passed validation. State the residual trusted base (what the model does *not* cover).

P00K.4 **Tamper-evident results.** Model a pipeline whose result files are integrity-protected (immutable + IMA-appraised). Prove an unprivileged step cannot alter a sealed result; validate that the kernel actually refuses the write on the target machine; document the assumptions (kernel config, hardware root of trust).

P00K.5 **Model-vs-reality audit (differential testing).** Pick five system calls relevant to a course lab. Build the small state-machine model, then a harness that runs the real calls and checks the model predicts each outcome. Report where model and kernel agree, where they diverge, and what each divergence means for a proof built on the model. (This is the honesty gate of §4, done as a standalone study.)

P00K.6 **An assumptions page for a real deployment.** Take one lab's actual runtime (image, kernel version, cloud/VM). Write its trusted-base / assumptions page in the style of §4 step 5: what is proved, what is validated-not-proved, what is assumed, and what a realistic attacker with cheap AI would target first.

7. Sources

- A. Silberschatz, P. B. Galvin, G. Gagne. *Operating System Concepts.* Wiley. (The standard first course: processes, memory, filesystems, protection.)
- R. Love. *Linux Kernel Development*, 3rd ed. Addison-Wesley. (Kernel-side view: syscalls, the VFS, scheduling.)
- The Linux `man` pages: `capabilities(7)`, `seccomp(2)`, `namespaces(7)`, `mount_namespaces(7)`, `landlock(7)`, `credentials(7)`, `chattr(1)`. (Primary, authoritative behaviour of each mechanism.)
- The Linux kernel source, `fs/namei.c`, `fs/attr.c`, `fs/namespace.c`. (Where path resolution, attribute changes, and mounts actually happen.)
- The seL4 microkernel verification project (Klein et al., SOSP 2009, and the project's assumptions page). (The reference example of a machine-checked OS guarantee that ships an explicit trusted-base statement.)
- The `strace` and `bpftrace` documentation. (Tracing real syscalls — the differential-testing tool of §4.)

8. Exercises

E00K.1 (Light.) List, from memory, the four things §3.3 says the kernel manages, and for each write one sentence naming the system call an ordinary program would use to ask for it.

E00K.2 Trace a real system call by hand: run `strace -f ls /tmp` (or `strace cat somefile`) and find the `openat`, `read`, and `close` calls. Explain, in one paragraph, where user mode ends and kernel mode begins in what you see.

E00K.3 State the §4.1 micro-claim as a precise "for all sequences of operations ..." sentence. Which of the honesty flags (CERTIFIED / STATED / UNCERTIFIED) would you attach to your sentence *before* mechanising it, and which *after*?

E00K.4 (Ablation.) The §4.1 claim needs the uid-0 process to lack `CAP_LINUX_IMMUTABLE`. Argue informally what goes wrong if that assumption is dropped, and give the two-operation attack sequence that then succeeds.

E00K.5 (Deep; the on-ramp to the §6 group projects.) Revisit §4.2 and take any parallel algorithm from a later unit. Identify one correctness or cost assumption it makes about isolation, atomicity, or privacy that ultimately depends on an OS-level guarantee, and state precisely the OS-level property that would have to be proved for the algorithm's guarantee to hold. If you continue, this is exactly project P00K.1.

Chapter 4

00L llm workload

U0L — The LLM: this course's running workload

The ScaDaMaLe book, unit 0L. Read once after U0, lightly; every later unit ends with an "LLM lens" box that lands its mathematics back here. The large language model is not this book's finale — it is the worked example the whole book carries, and by U18 you will have met every one of its parts under its mathematical name.

Entry kit — what this chapter assumes; follow a link if you need the memory jog (map: Figure 1): F.C1 (the three machines this workload runs on) • F.M2 (matrices — the substance of the model) • F.S1 (expectation — the substance of its output)

1. What an LLM is, formally

Fix a finite alphabet V of **tokens** (subword strings; $|V|$ in the tens of thousands). A **language model** is a probability distribution over token sequences, specified **autoregressively**:

$$p_\theta(x_1, \ldots, x_T) \;=\; \prod_{t=0}^{T-1} p_\theta(x_{t+1} \mid x_{1:t}),$$

where each conditional $p_\theta(\cdot \mid x_{1:t})$ is a distribution on V computed by a parameterized function of the prefix. That is the entire definition. Two readings, both load-bearing:

1. **The statistical reading (F.S3, U14, U15):** an LLM is a **density estimator on sequence space** — an enormous nonparametric-in-spirit model fit to a corpus by maximum likelihood, i.e. by minimizing cross-entropy $-\frac{1}{n}\sum \log p_\theta(x_{t+1} \mid x_{1:t})$. "Perplexity" is the exponentiated version of exactly this log-loss (U15's lens). *Generation* is nothing exotic: **sampling from a fitted conditional distribution**, one token at a time (U4's lens).
2. **The computational reading (F.M2, U6):** the function computing each conditional is a **transformer** — an alternating stack of attention and feed-forward blocks, and both are, in substance, matrix multiplications. One equation carries the whole attention idea: with query/key/value matrices Q, K, V (each a matmul of the token representations against learned parameter matrices),

$$\mathrm{Attn}(Q, K, V) \;=\; \mathrm{softmax}\!\left(\frac{QK^\top}{\sqrt{d}}\right)V.$$

   Rows of the softmax matrix are probability weights (F.S1: a convex average); the output is a weighted mixing of value vectors. Everything else — embeddings, feed-forward layers, the final logits — is more matmul. **Parameter count is dominated by these matrices**,

which is why "a 8B model" is, computationally, "8 billion matrix entries that must be read to produce a token."

Training the parameters θ is U9's story (SGD/Adam at scale) told properly in U18. This unit needs only the *shape*: an LLM is a sampled conditional density whose every evaluation is a stack of matmuls.

1.1 Tokens and the indexing, spelled out

A **tokenizer** splits text into pieces from the model's fixed vocabulary V, and each piece is named by an integer id. That id is a token. Vocabularies of current models have from a few tens of thousands to a few hundred thousand entries. A word may be one token or several: "unbelievable" might be split as un · believ · able, i.e. 3 tokens. The exact split depends on the tokenizer. A common rule of thumb for English is that one token is about three quarters of a word.

The prompt and the answer form **one** sequence $x_1, \ldots, x_T$:

- capital T is its length, a fixed number;
- lower-case t is a position, running over $1, \ldots, T$, so x_t is the token at position t and x_T the last one.

The product above can equally be written

$$p_\theta(x_1, \ldots, x_T) = \prod_{t=1}^{T} p_\theta(x_t \mid x_1, \ldots, x_{t-1}),$$

where the first factor conditions on nothing. Each factor is "the probability of the next token given all the tokens before it". **Generating** means sampling x_{t+1} from $p_\theta(\cdot \mid x_{1:t})$, appending it, and repeating. That is the one-token-at-a-time **decode loop** of §2 and Figure 4.1.

1.2 How the weights are made: pre-training, then post-training

Pre-training fits the density p_θ itself. Given a corpus of D tokens it minimises the next-token cross-entropy

$$\mathcal{L}(\theta) \;=\; -\frac{1}{D} \sum_{t} \log p_\theta(x_{t+1} \mid x_1, \ldots, x_t),$$

the log-loss of the statistical reading above; for current models D runs to trillions of tokens. The method is stochastic gradient descent in its Adam form (U9). One step on a mini-batch of sequences has three parts:

1. **forward:** push the batch through the model. All positions of a sequence are processed at once, so the step has the matrix-times-matrix shape of prefill (§2), at about $2N$ FLOPs per token for N parameters (U18 §3.3);
2. **backward:** compute the gradient $g = \nabla_\theta \mathcal{L}$ on the batch by back-propagation through the same graph, at about twice the forward cost, $4N$ FLOPs per token;
3. **update:** $\theta \leftarrow \theta - \eta\, g$ (Adam first rescales g coordinate by coordinate), a few operations per parameter, independent of the batch size.

This is where the training estimate $C \approx 6ND$ FLOPs of U18 §3.3 comes from, and U18 §4.1 prices the same step spread over many machines.

Post-training starts from the pre-trained weights and trains further on far smaller, curated data. Two kinds of signal are used.

- **Instruction examples**, for **supervised fine-tuning (SFT)**: the same cross-entropy loss, restricted to curated prompt–answer pairs. It adds little to what the model knows; it teaches the format of an answer.
- **Preference or reward signals**: answers to the same prompt, ranked by people or by a scoring model. Reinforcement learning from human feedback (**RLHF**) fits a reward model to the rankings and then optimises the language model against it. Direct preference optimisation (**DPO**) fits the rankings directly with a classification-style loss and no separate reward model. The reinforcement-learning variants must also *sample* answers from the current model at every step, so they add decode-phase work (§2) to a training job that is otherwise prefill-shaped.

Whatever the recipe, **the product is one file: the weights θ.** That file is the memory box of Figure~4.1, and §2 prices reading it.

1.3 Open-weight and closed-weight models

Who holds θ decides how a model can be used and what can be measured about it. A **closed-weight model** keeps its weights on the provider's servers: the user sends prompts over the network to a programming interface (an API) and usually pays per token. An **open-weight model** publishes the weight file: the user downloads it and runs it on hardware they govern, as in this unit's lab.

	closed-weight	open-weight
where θ lives	on the provider's servers only	a file you download and keep
how it is used	prompts over the network (an API), paid per token	run on hardware you govern
what you can measure	only what the interface returns	everything: your own decode ceiling (§2), memory, energy
adaptation	only as the provider allows	fine-tune it yourself, e.g. with LoRA (U18 §7)
your prompts and data	leave your machine	stay on your machine

Open-weight is not open-source. The training data and training code are often not released, so the model cannot be rebuilt from its ingredients. The licences under which weights are released also differ, and some restrict use. A licence is a contract, which puts it in the fourth pillar (U00), so read it before building on a model. The cost lens is the same either way: someone paid about $6ND$ FLOPs to train the model (U18 §3.3), and serving speed is capped by bandwidth over active bytes (§2). What differs is whose machine does the arithmetic, and so who can check the price.

Check your reading.
Q1. Formally, a language model as defined in this course is:
(A) a database of memorised sentences with fast lookup (B) an autoregressive conditional distribution over tokens: $p(x_{t+1} \mid x_{1:t})$, fit by maximum likelihood (C) a rule-based grammar with learned exceptions (D) a search engine over its training corpus

2. The cost anatomy of inference — one formula to carry

The three machines of U0 price this workload as follows. The model's weights occupy

$$\text{memory} \;=\; (\#\text{parameters}) \times (\text{bytes per parameter}),$$

where bytes-per-parameter is a **dial** (quantization): 2 bytes at 16-bit, ~0.5 bytes at 4-bit (U18 treats what quantization does to quality; here it is pure cost). In addition the **KV cache** — the stored keys/values of every previous token — grows linearly with context length and must also be read.

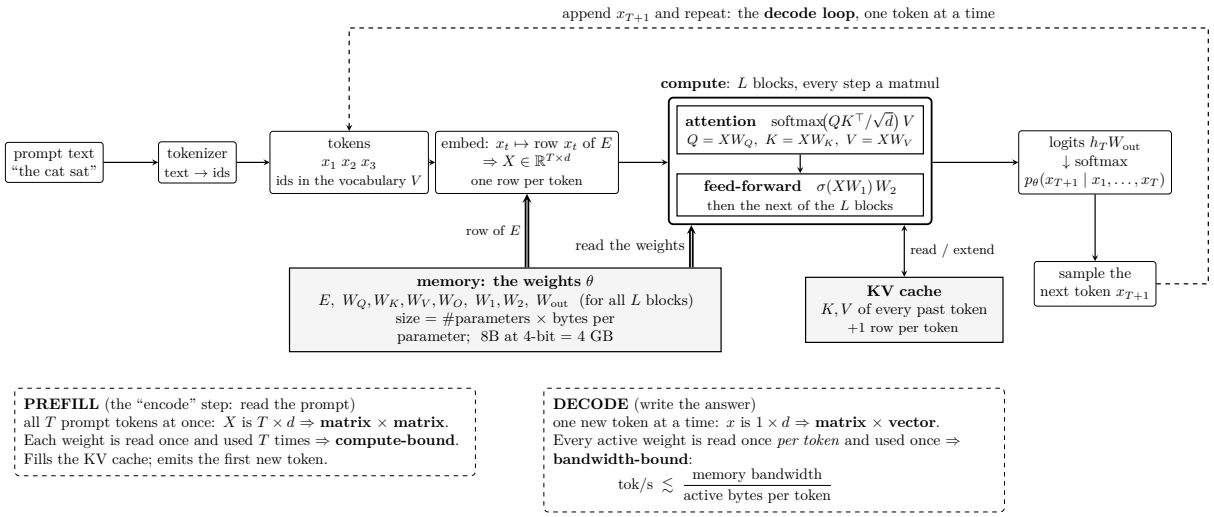


Figure 4.1: An LLM as a workload. Text becomes tokens, tokens become rows of X, and every block is matrix multiplication against weights that live in memory. *Prefill* pushes the whole prompt through at once (matrix × matrix, compute-bound). *Decode* loops one token at a time and re-reads the weights for each one (matrix × vector, bandwidth-bound). That re-reading is the decode ceiling.

Inference has two phases with *opposite* cost profiles:

- **Prefill** (ingest the prompt): all prompt tokens at once → large matmuls → **compute-bound** (the compute side of the roofline comparison of §2.1).
- **Decode** (emit tokens one by one): each new token must **read every active weight once** to produce one token → **memory-bandwidth-bound**. To first order,

$$\text{tok/s} \lesssim \frac{\text{effective memory bandwidth}}{\text{active bytes per token}}$$

Figure~4.1 puts the whole of §1–2 in one picture: what goes in and comes out, where the weights live, and why the two phases are priced differently.

Illustrative-class numbers (labeled illustrative; measure your own machine in the lab): an 8B-parameter model at 4-bit is ~4 GB of active bytes; on a ~250 GB/s-class unified-memory node the ceiling is ~60 tok/s; on a ~1 TB/s-class GPU-memory node, ~250 tok/s. **The formula, not the numbers, is the content**: decode speed is a bandwidth story, which is why memory bandwidth (the price U0's unit-cost RAM leaves out) and F.E4's measured α-β are not warm-up acts — they are the pricing model of the most economically significant workload of the decade.

2.1 Why the two phases price differently

Count what one weight does. In prefill the T prompt tokens form a matrix $X \in \mathbb{R}^{T\times d}$, and each layer multiplies it by weight matrices such as $W \in \mathbb{R}^{d\times d}$: matrix times matrix. Each entry of W is read from memory once and used in T multiply-adds, one per token, which is $2T$ FLOPs per weight read. In decode only the newest token is new: $x \in \mathbb{R}^{1\times d}$ times W, vector times matrix. Each weight is read once and used in one multiply-add, which is 2 FLOPs. At b bytes per weight, the **arithmetic intensity** (FLOPs per byte moved from memory) is

$$\text{prefill: } \frac{2T}{b}, \qquad \text{decode: } \frac{2}{b}.$$

The *roofline* model compares this ratio with the machine's balance point: its peak FLOP rate divided by its memory bandwidth. That ratio is in the tens for laptop CPUs and the hundreds

for GPUs (illustrative classes). Decode, at $2/b$ (4 FLOPs per byte at 4-bit), therefore sits far on the memory side: the arithmetic units wait for weights. Prefill with a prompt of a few hundred tokens reaches the compute side. The weights and matmuls are the same; the only difference is how many tokens share each read. The KV cache (§2.4) adds a second stream that every decode step must also read.

2.2 The decode ceiling, with symbols

Setting. The decode phase, one request at a time (batch size 1). To emit one token the model reads every *active* weight once (§3 defines active; for a dense model it is every weight). To first order the weight traffic dominates, so the KV cache is ignored here; §2.4 puts it back.

Symbols. B is the effective memory bandwidth in bytes per second; N_a is the number of active parameters per token; b is the number of bytes per parameter (2 at 16-bit, 1 at 8-bit, about 0.5 at 4-bit).

One token moves $N_a b$ bytes, and at most B bytes move per second, so

$$\text{tok/s} \lesssim \frac{B}{N_a\, b}.$$

This is the boxed ceiling written in symbols. It is an upper bound: everything else the step does (reading the KV cache, attention arithmetic, unpacking quantised weights, achieving less than peak bandwidth) can only lower the number. FLOPs barely enter. With m requests batched, one read of the weights serves m tokens, so total throughput rises until compute binds; U18 §6.2 treats that regime.

2.3 Worked examples on a laptop

Every bandwidth in this subsection is an illustrative peak computed from a memory specification. Read your own machine's specification; §2.5 then measures the decode rate it caps.

The pipe. A DDR memory module is 64 bits = 8 bytes wide (DDR5 splits it into two 32-bit subchannels of the same total width). A module rated DDR5-5600 performs 5.6×10^9 transfers per second. A laptop with two channels therefore has peak bandwidth

$$B \;=\; 2 \times 8\ \text{B} \times 5.6 \times 10^9\ \text{s}^{-1} \;=\; 89.6\ \text{GB/s}.$$

With dual-channel DDR4-3200 the same computation gives $B = 2 \times 8 \times 3.2 \times 10^9 = 51.2$ GB/s.

An 8B dense model at 4-bit. Bytes per token: $N_a b = 8 \times 10^9 \times 0.5 = 4 \times 10^9$ B = 4 GB. Ceiling: $89.6/4 = 22.4$ tok/s, so at best about 45 ms per token. On the DDR4-3200 laptop: $51.2/4 = 12.8$ tok/s. Faster memory gives proportionally faster text.

Quantisation overheads. A 4-bit format stores, besides the 4-bit weights, a scale factor (and sometimes an offset) for each small block of weights. The effective size is therefore somewhat above 0.5 bytes per parameter; take $b \approx 0.55$. A model with 4×10^9 active parameters then reads $4 \times 10^9 \times 0.55 = 2.2$ GB per token, and at $B \approx 50$ GB/s the ceiling is $50/2.2 \approx 23$ tok/s. Real systems land lower (§2.4, §2.5): expect a number between single digits and low tens on a laptop CPU.

The quantisation dial, before and after. The same 8B model on the 89.6 GB/s laptop with 16 GB of RAM:

weights	b (B/param)	memory $8 \times 10^9 \cdot b$	fits 16 GB?	ceiling $B/(N_a b)$
16-bit	2	16 GB	no (the operating system needs room too)	5.6 tok/s
8-bit	1	8 GB	yes	11.2 tok/s
4-bit	0.5	4 GB	yes	22.4 tok/s

Halving b halves the memory and doubles the ceiling before any hardware changes. The quality price is empirical, not a theorem; U18 §6.3 states what is observed.

2.4 The KV cache, in numbers

With full multi-head attention, a model with L layers and hidden dimension d keeps a KV cache of $2LdTb_{\text{kv}}$ bytes for a context of T tokens at b_{kv} bytes per entry (U18 §6.1; the 2 counts keys and values). Attention reads all of it at every decode step, so it joins the weights in the denominator:

$$\text{tok/s} \lesssim \frac{B}{N_a b + 2LdTb_{\text{kv}}}.$$

Worked (8B-class shape $L = 32$, $d = 4096$, 16-bit cache, $b_{\text{kv}} = 2$). Per context token the cache holds $2 \times 32 \times 4096 \times 2 = 524\,288$ B $= 0.5$ MiB. At $T = 4096$ it holds 0.5 MiB $\times\, 4096 = 2$ GiB ≈ 2.15 GB. The bytes per token rise from 4 to $4 + 2.15 = 6.15$ GB, and on the 89.6 GB/s laptop the ceiling falls from 22.4 to $89.6/6.15 \approx 14.6$ tok/s. The cache grows as the conversation grows, so a long session slows down as it proceeds. Many current models let several query heads share one key/value head: with h query heads and g key/value heads, the cache shrinks by the factor g/h. Check your model's configuration before using the formula. E0L.2 carries this further.

2.5 Measuring your own number

The lab procedure has four steps: predict, run, measure, and explain the gap.

1. Find B from your RAM type and channel count, as in §2.3.
2. For the model you will run, find its active parameter count N_a and its bytes per parameter b, including overheads. For a dense model the file size is a good estimate of $N_a b$.
3. Predict the ceiling $B/(N_a b)$, and write the arithmetic down.
4. Measure the decode rate, and explain the difference.

One way to run a local model is the Ollama runtime. The commands below were current when written; this software changes quickly, so check it at lab time.

```
ollama pull qwen3:1.7b        # about 1.4 GB: fits an 8 GB laptop
ollama pull qwen3:4b          # about 2.5 GB: for 16 GB of RAM or more
ollama run qwen3:4b --verbose "Explain memory bandwidth in two sentences."
```

The same in a container (CPU only; F.E2):

```
docker run -d --name ollama -v ollama:/root/.ollama -p 11434:11434 ollama/ollama
docker exec -it ollama ollama pull qwen3:1.7b
docker exec -it ollama ollama run qwen3:1.7b --verbose "Explain memory bandwidth
  in two sentences."
```

With `--verbose` the runtime prints timings after the answer. The line `eval rate: … tokens/s` is the decode rate, the number to compare with the ceiling. `prompt eval rate` is the prefill rate, which is priced by compute (§2.1) and not by the ceiling. The first run also loads the model from disk, so run twice and keep the second run's decode rate. Its prompt eval is not a prefill measurement if the prompt is the same: the runtime reuses the cached prompt, and the

prompt eval takes about one decode step (Lab 7's log: 57.9 ms, about 707.9 tok/s). Measure prefill with a prompt the runtime has not seen, or take the first run's prompt eval rate.

A measured example (one machine, measured once). A 2018-class laptop: Intel Core i5-8350U, dual-channel DDR4-2400 (its specification), 32 GB of RAM. The model is `qwen3:4b`, about 4×10^9 parameters in a 2.5 GB file at 4-bit, so $N_a b \approx 2.5$ GB.

```
B   = 2 channels x 8 bytes x 2.4e9 transfers/s   = 38.4 GB/s  (peak)
ceiling  B / (N_a b) = 38.4 / 2.5                  ~ 15 tok/s
measured: ollama run qwen3:4b --verbose
  eval rate          8.7 tok/s   (decode: 57% of the ceiling)
  prompt eval rate  25.5 tok/s   (prefill: compute-bound, not the ceiling)
  load duration     1 min 45 s   (cold, from disk)
```

The measurement is below the ceiling, as it must be. A single program seldom sustains the peak bandwidth; on an older CPU, unpacking 4-bit weights costs compute too; and the KV cache and attention add traffic of their own (§2.4). Predicting about 15, measuring 8.7, and explaining the gap is the whole lab report, done once. E0L.5 asks for yours.

Check your reading.

Q2. Why is the decode phase of LLM inference typically memory-bandwidth-bound rather than compute-bound?

(A) because GPUs cannot multiply matrices quickly (B) because each new token requires reading every active weight once, while doing relatively little arithmetic per byte read (C) because the KV cache must be recomputed from scratch at every step (D) because tokens must be fetched from the internet

Q3. Your laptop's memory bandwidth is 100 GB/s (peak, from its memory specification as in U0L section 2.3). Estimate the decode ceiling for a 7B-parameter model at 4-bit quantisation, and name one reason the real number will be lower.

3. Dense vs mixture-of-experts (MoE): what "active" means

A **dense** model activates all parameters for every token: active = total. A **mixture-of-experts** model replaces feed-forward blocks with many parallel **experts** plus a learned **router** that sends each token through only a few:

- **memory** must hold *all* experts (total parameters);
- **bandwidth per token** pays only the *active* ones.

This single asymmetry re-prices distributed inference. Worked contrast (illustrative, 4-bit): a 120B-total / 5B-active MoE occupies ~60 GB of memory but reads only ~2.5 GB per token — on a ~215 GB/s-class node that is an ~85 tok/s ceiling, where a *dense* 120B on the same node would cap near 3.5 tok/s. Consequence, worth stating as the unit's second boxed fact:

> **Dense models reward fast interconnect** (every parameter must be reachable per token); **MoE models reward large memory per node** (hold all experts locally, read few). Hardware and model architecture are one optimisation problem, not two.

The router itself is mathematics this book teaches: a **top-k selection** (U2's lens) feeding a **hash-partition-like placement** of experts across machines (U5's lens), whose load balance is a balls-in-bins tail question (F.S2, U3's lens).

3.1 Worked example: total versus active parameters

Use the per-layer count of U18 §3.3: attention projections $4d^2$ plus one feed-forward block $8d^2$ (expand $d \to 4d$, project back), $12d^2$ in all. Embeddings, normalisation and the router's own dE weights are ignored. Take $L = 32$, $d = 4096$, so $Ld^2 = 32 \times 4096^2 \approx 5.37 \times 10^8$. Compare a

dense model with an MoE in which every feed-forward block is replaced by $E = 8$ experts of the same shape, of which the router keeps the top $k = 2$:

```
                router: scores over E = 8 experts, keep top-2
token x --> [ attention ] --> router --+--> FF_3 --+
                                      +--> FF_6 --+--> weighted sum
FF_1 FF_2 FF_4 FF_5 FF_7 FF_8 : in memory, NOT read for this token
```

model	parameters per layer	total N	active N_a	4-bit memory	4-bit ceiling at 89.6 GB/s
dense	$4d^2 + 8d^2 = 12d^2$	6.4B	6.4B	3.2 GB	28 tok/s
MoE, E=8, k=2	$4d^2 + 8 \cdot 8d^2 = 68d^2$	36.5B	$(4 + 2 \cdot 8)d^2 = 20d^2$: 10.7B	18.3 GB	16.7 tok/s

The MoE holds $68/12 \approx 5.7$ times the parameters but reads only $20/12 \approx 1.7$ times the bytes per token (5.4 GB against 3.2 GB); its real cost is 18 GB of memory just to exist. (The dense shape gives 6.4B, not 8B. Real 8B-class models spend the rest on embeddings and wider feed-forward blocks: the formula gives the shape, not the model card.) The 120B/5B contrast above is the same arithmetic at a larger scale: $215/2.5 = 86$ tok/s against $215/60 \approx 3.6$ tok/s.

Check your reading.
Q4. A 120B-total / 5B-active mixture-of-experts model, compared with a 120B dense model on the same node:
(A) needs less memory and less bandwidth per token (B) needs the same memory but ~24x less bandwidth per token (C) needs more memory and more bandwidth per token (D) cannot be run without a GPU

4. Splitting a model across machines: the trichotomy

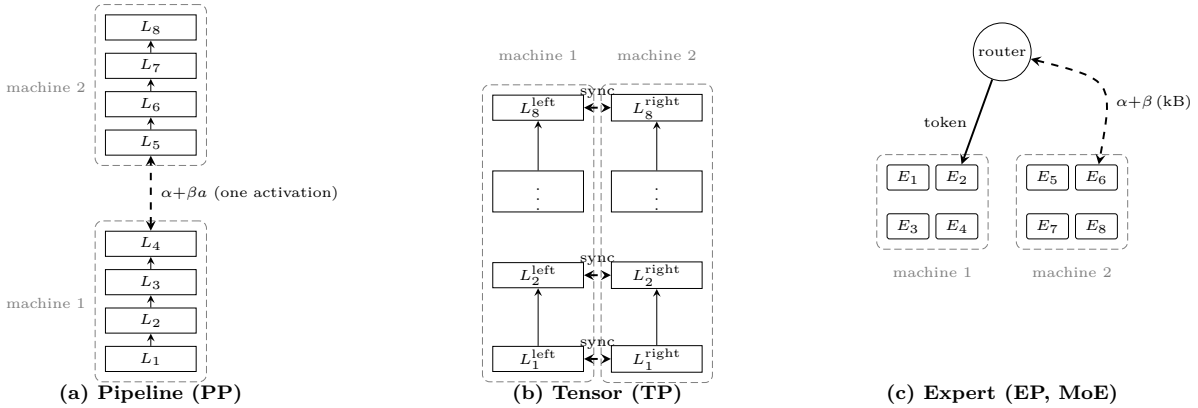
When one node's memory cannot hold the model — or one node's bandwidth cannot serve it fast enough — the model is split. There are exactly three basic strategies, and they are the three cost lenses of U0 wearing new clothes:

Strategy	What is split	Per-token communication	Cost-lens reading	Adding a node buys
Pipeline (PP)	layers: node 1 holds layers 1..k, node 2 the rest	one activation vector per boundary per token	the **depth** lens: a sequential chain; only one stage active per token (batch=1)	capacity (bigger models), **not** single-stream speed
Tensor (TP)	every layer, sharded	an all-reduce **inside every layer** (two per layer in the forward pass)	the **work** + α-β lens: Brent-style division paying a per-layer communication term	speed *and* capacity — **if** the interconnect is fast enough that sync $\ll$ compute
Expert (EP)	whole experts (MoE only)	the token itself (kilobytes) routed to its expert's node	the **shuffle** lens: hash-partition + routing; traffic $\propto$ tokens, not weights	capacity at near-zero speed penalty, even on modest networks (per-token latency still enters once per MoE layer at batch 1)

The one-line summary a student should be able to reproduce on demand: **PP moves activations, TP moves partial sums, EP moves tokens** — and their prices under the α-β model (F.E4) follow immediately from *what* crosses the wire *how often*. Figure 4.2 draws all three on one model pair of machines.

4.1 The three cuts, priced on two laptops

Take the 8B dense model of §2.3 at 4-bit (4 GB of weights, $L = 32$, $d = 4096$, activations at 2 bytes), and two laptops of 89.6 GB/s each. Decode, batch 1. Network costs are illustrative.



(a) Pipeline (PP)
cut *between* layers
moves **activations**;
one stage active per token

(b) Tensor (TP)
cut *through* every layer
moves **partial sums**,
$(\alpha+\beta s)$ *per layer*

(c) Expert (EP, MoE)
cut *by expert*
moves **tokens** (tiny);
traffic $\propto$ tokens, not weights

Figure 4.2: One eight-layer model, two machines, three cuts. (a) **Pipeline parallelism** cuts between layers: capacity doubles, but a single token still walks the whole chain — the depth lens of U0. (b) **Tensor parallelism** cuts through every layer: work is divided Brent-style, but every layer pays a synchronisation $(\alpha + \beta s)$ — the work-plus-communication lens; it wins only when the wire is fast. (c) **Expert parallelism** (MoE) cuts by expert: the router ships each *token* (kilobytes) to its expert's machine — the shuffle lens of U5; modest networks suffice because weights never travel. PP moves activations, TP moves partial sums, EP moves tokens.

Pipeline (PP). Device A holds layers 1–16 and device B layers 17–32, 2 GB each. Per token, one activation vector crosses the boundary: $d \times 2 = 8192$ B $= 8$ KiB. At batch 1 the stages run one after the other: A reads its 2 GB (22.3 ms), sends 8 KiB, then B reads its 2 GB (22.3 ms). The total is about 45 ms per token, the same as one laptop, plus the hops (the boundary, and the new token returning to A). A second node buys capacity, not single-stream speed.

Tensor (TP). In each block the first weight matrix is split by columns and the second by rows, so each device computes a partial sum of the block's output, and one all-reduce combines the two. A transformer layer has two such blocks (attention and feed-forward), so the forward pass needs $2L = 64$ all-reduces per token. Each device reads 2 GB per token: 22.3 ms. At batch 1 each all-reduce carries only d values (8 KiB), so its cost s is mostly latency α (F.E4). With $s = 0.1$ ms, a token costs $22.3 + 64 \times 0.1 = 28.7$ ms, about 35 tok/s, faster than one laptop. With $s = 1$ ms, it costs $22.3 + 64 = 86.3$ ms, about 11.6 tok/s, slower than one laptop. The break-even point is $s \approx 22.3/64 \approx 0.35$ ms. TP buys speed and capacity only when sync $\ll$ compute.

Expert (EP; MoE only). For the MoE of §3.1, device A holds experts 1–4 of every layer and device B experts 5–8. At each MoE layer the router picks, say, experts 3 and 6. Expert 3 runs where the token already is; the token's activation (about 8 KiB) is sent to B, and $\text{FF}_6(x)$ comes back. The weights never move; only the token travels, at cost about $2(\alpha + \beta a)$ per remote expert with a the activation bytes, independent of the experts' size. Per token this is at most one round trip per MoE layer, so kilobytes, not gigabytes, cross the wire. At batch 1, though, the latency α still enters up to L times per token, so on a high-latency network EP too slows a single stream. What stays small is the byte count, and with many tokens in flight the round trips are shared.

Check your reading.

Q5. Which correctly pairs the three parallelism strategies with what they move across the network?
(A) pipeline moves tokens; tensor moves activations; expert moves partial sums (B) pipeline moves activations; tensor moves partial sums; expert moves tokens (C) pipeline moves weights; tensor moves weights; expert moves weights (D) all three move the full model every step

Q6. Under pipeline parallelism with batch size 1, adding a third machine to a two-machine chain primarily buys:
(A) faster single-stream token generation (B) the capacity to hold a larger model, with no single-stream speed gain (C) lower per-token latency (D) nothing at all

5. The frameworks, descriptively

The open ecosystems that implement §4 (concept-level here; hands-on in the labs; all operational specifics are best checked at lab time — this landscape moves quarterly):

Framework	Strategy	Runs on	One-sentence identity
vLLM (+ Ray)	TP + EP (+ PP)	Linux, CUDA/ROCm GPUs	the production serving standard: continuous batching + paged KV-cache, clustered via Ray (U3's scheduling lens, live)
EXO	TP/EP, peer-to-peer	Apple-silicon fleets	auto-discovering home-cluster inference over high-bandwidth local links
llama.cpp RPC	PP (layer split)	everything (CPU/Metal/CUDA/ROCm)	the universal fallback: capacity over speed, any hardware mix
Distributed Llama	TP	cross-platform	tensor parallelism with 2^n-1 workers over Ethernet
Petals	PP over a swarm	internet-scale volunteers	BitTorrent-style shared serving of very large models

Choosing among them is a U0-style exercise, not a matter of fashion: identify what your network charges (α, β), what your nodes hold (memory), what the model activates (dense/MoE) — the table then reads itself.

6. Honesty box (U0P discipline, applied to system claims)

Throughput claims ("~X tok/s") are **benchmark-STATED**: they depend on quantization, context length, batch size, thermal state, and software version, and they are not certificates. In this book's terms (U0P §5): a *cost bound* derived from §2's formula with measured bandwidth is a checkable claim; a *quality* claim about a model's outputs is contractible only the U16 way — an agreed test on held-out data with a known error distribution. When this book states an LLM number it is labeled illustrative or measured-in-lab; treat every vendor benchmark you meet with the same taxonomy.

Check your reading.

Q7. A vendor page says "our node does 94 tok/s on an 8B model". In two sentences: what would make this a checkable claim in this course's sense, and which part of it can NEVER be certified by a benchmark alone?

7. Where every later unit lands on this workload

U1: allreduce is gradient sync and layernorm's reduction · U2: the router's top-k is selection · U3: continuous batching is online scheduling · U4: decoding is sampling, the KV cache is a windowed summary · U5: corpus pipelines and EP routing are shuffle/partition · U6: attention

IS matmul; prefill/decode is the roofline · U7: LoRA is low-rank; embeddings feed retrieval · U8: normalisation is conditioning · U9: training is SGD at scale · U12: attention is a weighted token graph · U13: RAG is ANN · U14/U15: the LLM is a sequence-space density estimator; perplexity is log-loss · U16: evaluation and alignment are contracts · U19: federated fine-tuning is consensus · U20: merging estimators vs merging experts · **U18: all of it, assembled and made big.**

8. Sources (credited; archived under `scadamale-book-sources`)

- A. Vaswani et al., "Attention Is All You Need," NeurIPS 2017. (The transformer; the attention equation of §1.)
- N. Shazeer et al., "Outrageously Large Neural Networks: The Sparsely-Gated Mixture-of-Experts Layer," ICLR 2017. (MoE routing; §3.)
- W. Fedus, B. Zoph, N. Shazeer, "Switch Transformers," JMLR 23, 2022. (MoE at scale; load balancing; §3.)
- L. Ouyang et al., "Training language models to follow instructions with human feedback," NeurIPS 2022. (RLHF; §1.2.)
- R. Rafailov et al., "Direct Preference Optimization: Your Language Model is Secretly a Reward Model," NeurIPS 2023. (DPO; §1.2.)
- The distributed-serving systems literature is surveyed in U18's sources; framework documentation (vLLM, llama.cpp) is cited there as software, not as scholarship.

9. Exercises

E0L.1 (The ceiling formula.) A 14B-parameter dense model is served at 8-bit on a node whose effective memory bandwidth is B GB/s (for your own laptop, the peak figure of §2.3, which is also the input Lab 7 asks for). Derive the decode tok/s ceiling as a function of B; evaluate at $B = 100, 250, 1000$. State one reason measured throughput falls below the ceiling.

E0L.2 (KV-cache arithmetic.) For a model with L layers, hidden dimension d, 16-bit KV entries, and context length T: write the KV-cache size formula $2 \cdot L \cdot d \cdot T \cdot 2$ bytes, evaluate for L=32, d=4096, T=128k, and find the T at which the cache overtakes a 4-bit 8B model's weights. (This is why long context is a *memory* feature, not a *quality* feature.)

E0L.3 (PP vs TP on the α-β model.) Two nodes, per-layer compute time c, L layers split evenly. PP sends one activation of a bytes per boundary per token; TP synchronises s bytes per layer. Write the per-token decode time of each under F.E4's $T(M) = \alpha + \beta M$; find the (α, β) regime where PP beats TP for batch 1, and explain in one sentence why large batches change the answer.

E0L.4 (Expert load balance = balls in bins.) A router sends each of n tokens independently to one of E experts sharded evenly across 2 nodes. Using F.S2, bound the probability that one node receives more than 60% of the tokens for $n = 10^4, E = 16$, and name the U3 quantity this imbalance inflates.

E0L.5 (Predict, run, measure, explain.) Using §2.3–2.5: (a) compute your machine's peak memory bandwidth B from its memory type and channel count; (b) for the model you run, state N_a and b (with overheads) and predict the decode ceiling $B/(N_a b)$; (c) measure the decode rate twice and keep the second run (for prefill, use a fresh prompt); (d) report the ratio of measured to predicted rate, name two causes of the gap, and test one of them. For example, repeat with a long prompt and compare the change with §2.4's KV-cache term.

```
B   = __ channels x __ bytes x __ GT/s        = ______ GB/s
```

```
N_a = ______ x 10^9   b = ______ B/param   => N_a b = ______ GB
predicted ceiling  B / (N_a b)                = ______ tok/s
measured                                      = ______ tok/s
```

Chapter 5

00N networking foundations

U00N — Networking foundations: how the internet actually works (no black magic)

The ScaDaMaLe book, preliminary unit N (the engineering floor, before the first lab). Our own exposition; sources in §7. Written for a mathematics student: no prior networking assumed, every acronym expanded on first use.

The one idea of this unit: *every network thing a lab does is a short stack of simple layers.* When you `curl` a dataset, `docker run` a Spark cluster, or an agent calls a tool, nothing magic happens — each step is one small protocol handing a clean abstraction to the next. This unit makes that stack visible, once, so nothing downstream is mysterious.

> **Entry kit** — what this chapter assumes; follow a link if you need the memory jog (map: Figure 1):
> F.E4 (the $\alpha+\beta n$ network cost model — the theorem this floor makes runnable)

1. Acronyms, expanded (the master table — none are assumed)

We keep one habit book-wide: **every acronym is expanded the first time it appears.** This table is the master list for the networking floor; later units extend it.

Acronym	Expansion (one line)
OSI	Open Systems Interconnection — a 7-layer *reference model* for networking
TCP	Transmission Control Protocol — reliable, ordered byte-stream
IP	Internet Protocol — addresses + routes packets host-to-host
UDP	User Datagram Protocol — bare packets, fast, lossy
TCP/IP	the 4-layer protocol *suite* the real internet runs on
DNS	Domain Name System — resolves a name to a numeric address
URL	Uniform Resource Locator (`https://host/path`)
HTTP / HTTPS	HyperText Transfer Protocol / HTTP Secure
TLS / SSL	Transport Layer Security (the "S"); SSL is its retired predecessor
MAC	Media Access Control (a link-layer hardware address)
JSON	JavaScript Object Notation (a text data format — see U00S)
RPC / API	Remote Procedure Call / Application Programming Interface
MIME	Multipurpose Internet Mail Extensions (the `media_type` labels, e.g. `text/html`)
RFC	Request For Comments — the IETF standards documents
`curl`	"Client for URLs" — a command-line HTTP client
SSH	Secure Shell (encrypted remote login + transport)

(The serialization/RPC acronyms — gRPC, SSE, protobuf — are expanded in the sibling unit U00S.)

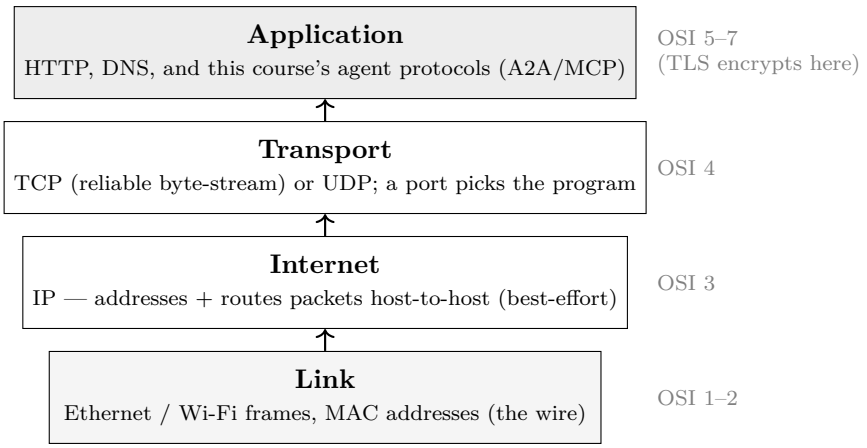


Figure 5.1: The layered idea, two ways. The 4-layer TCP/IP suite the real internet runs (boxes), with each layer mapped to the OSI 7-layer reference model (right). Read bottom to top: IP moves packets to a host; TCP makes a reliable byte-stream between two programs; TLS encrypts it; HTTP structures request/response over it; the course's agent protocols (A2A, MCP) sit at the very top. Every primitive the labs use lives at one definite layer — no magic.

2. The layered idea, and the OSI 7-layer model

Networking is built in **layers**: each layer does one job and hands a clean abstraction upward, so you can reason about "send these bytes to that machine" without thinking about voltages on a wire. The **OSI reference model** names 7 layers; the internet in practice uses the leaner 4-layer **TCP/IP model**. Figure 5.1 lines them up.

Bottom to top: **(1) Physical** — bits as signals on copper/fibre/radio. **(2) Data Link** — frames between two directly-connected nodes (Ethernet, Wi-Fi; MAC addresses). **(3) Network** — routing packets across many links to any host: **IP** (best-effort, no delivery guarantee). **(4) Transport** — end-to-end delivery between *programs*: **TCP** (reliable, ordered, a connection) or **UDP** (fast, unordered); a *port number* picks the program (443 = HTTPS). **(5–7) Session / Presentation / Application** — in practice folded together: encryption (TLS) sits around 6, and what programs actually speak (**HTTP**, **DNS**, and this course's agent protocols) is layer 7.

The one mental model to keep: *IP moves packets to a host; TCP turns packets into a reliable byte-stream between two programs; TLS encrypts that stream; HTTP structures request/response over it; JSON/protobuf is what's inside; an agent protocol is the conversation.* Every primitive we use later sits at a definite layer — no magic.

Check your reading.

Q1. IP delivers packets host-to-host but is best-effort (packets may be lost, reordered, duplicated). Which protocol turns that into a reliable, ordered byte-stream between two programs?
(A) DNS (Domain Name System) (B) TCP (Transmission Control Protocol) (C) TLS (Transport Layer Security) (D) HTTP (HyperText Transfer Protocol)

3. The core protocols (one line each)

- **IP** — addresses (`93.184.x.x`) and routes packets host-to-host, best-effort.
- **TCP** — a reliable ordered byte-stream over IP: a handshake, acknowledgements, retransmit-on-loss.
- **UDP** — bare packets: fast and lossy (used by DNS and video).
- **DNS** — turns a name (`a2a-protocol.org`) into an IP address (usually a UDP query to port 53).

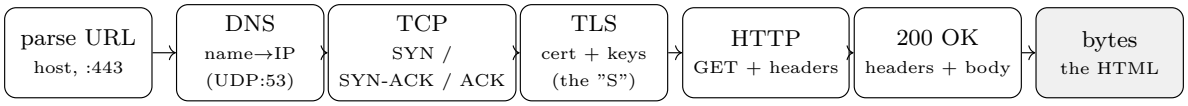


Figure 5.2: One `curl https://a2a-protocol.org/…` in seven steps — the whole trip, no magic. Parse the URL; resolve the name to an address (DNS); open a reliable connection (the TCP 3-way handshake); encrypt it and verify the server's certificate (TLS — the "S" in HTTPS); send the text request (HTTP `GET` + headers); read back the status, headers and body; and you have the bytes. Each box is exactly one protocol at one layer of Figure 5.1.

- **TLS** — authenticates the server (a **certificate**) and encrypts the TCP stream — the "S" in HTTPS.
- **HTTP** — a text request/response protocol (`GET /path`, headers, a body) over TCP (or TLS).

4. No black magic I — what `curl https://a2a-protocol.org/latest/specification/` really does

This is the exact fetch this course's protocol research used. Figure 5.2 traces it; here it is in words.

1. **Parse the URL** → scheme `https`, host `a2a-protocol.org`, path `/latest/specification/`, default port `443`.
2. **DNS lookup** — ask a resolver "what IP is `a2a-protocol.org`?"; get back an address (a **UDP** query to port 53).
3. **TCP handshake** — open a reliable connection to that IP on port 443: the 3-way **SYN** → **SYN-ACK** → **ACK** (synchronize / synchronize-acknowledge / acknowledge).
4. **TLS handshake** — the server presents a **certificate** proving it is `a2a-protocol.org`; the two sides agree keys; the stream is now encrypted (the "S").
5. **HTTP request** — over that encrypted stream, send text: `GET /latest/specification/ HTTP/1.1` plus headers (`Host:`, `User-Agent:`, `Accept:`).
6. **HTTP response** — the server streams back a status line (`200 OK`), headers (`Content-Type: text/html`, `Content-Length:`), a blank line, then the **body bytes**.
7. **You get the bytes** — `curl` writes them out (in our fetch: 616,875 bytes of HTML).

That is the *entire* trip. Our `IASWebFetch` tool is exactly this — raw `curl`, deterministic parse, byte-faithful file, **no model summarising** (a summariser confabulates; the bytes do not). Every step is one protocol; nothing is magic.

Check your reading.
Q2. You run `curl https://a2a-protocol.org/latest/specification/`. After the URL is parsed, which step happens FIRST, before any connection to the web server is opened?
(A) the TLS handshake (certificate + keys) (B) the DNS lookup that turns the host name into an IP address (C) sending the HTTP GET request line (D) reading back the 200 OK response body

5. No black magic II — what `docker run <image>` really does

Docker packages software as a content-addressed, layered filesystem and runs it as an isolated process. `docker run apache/spark`:

1. **Resolve the image reference** `registry/name:tag` (default registry = Docker Hub).
2. **Pull the manifest** — a JSON document (fetched over HTTPS, exactly as in §4) listing the image's **layers** by **content hash** (`sha256:...`). Content-addressed means *the hash is*

the name, so identical layers are shared and verified.

3. **Pull missing layers** — each layer is a compressed tarball of filesystem changes; layers already present (matched by hash) are skipped. This is why the *second* pull is fast.
4. **Assemble a union filesystem** — stack the read-only layers plus one writable top layer (copy-on-write) into the container's root.
5. **Isolate and run** — the Linux kernel gives the process its own **namespaces** (its own view of processes, network, mounts, users) and **cgroups** (control groups — CPU/memory limits; the very mechanism behind the out-of-memory-containment work in this book's own substrate). Then it starts the entry process.

So a container is *not* a virtual machine — it is a normal Linux process with a private filesystem view and resource limits. Layered, hashed, legible.

Check your reading.
Q3. A `docker run` pulls image layers named by their content hash (sha256:...). In one or two sentences, explain why the SECOND pull of the same image is fast, and why this makes the labs reproducible on any student's laptop.

6. Where THIS course sits (the tie to the cost spine)

Our agent protocols live at the **Application layer**; their JSON-RPC/gRPC bindings ride **HTTP** over **TCP/TLS** over **IP**. The course's central cost model, **DPRAM** — *rounds R, local memory s, and bytes moved* (introduced properly in U0, and drilled once we reach the distributed model in U5) — is *literally the cost of these lower layers*: "bytes moved" is transport payload; a "round" is one request/response trip (each a §4 round). Serialization (protobuf vs JSON, U00S) shrinks the *bytes-moved* constant; the transport choice sets the round latency. **The network stack and the DPRAM theory are the same story from two directions** — which is exactly why the labs open here.

LLM lens — where this unit lands on the running workload (U0L):
The running large-language-model workload of U0L is this floor in motion: the weights ship as content-hashed `docker` layers (§5); each token request is an HTTP round over TCP/TLS (§4); the per-round bytes are the serialized prompt/response (U00S); and "how many rounds, how many bytes" is precisely the DPRAM cost (§6). One inference call exercises the whole stack, top to bottom.

7. Sources

- Our own exposition. First stops for each concept: Wikipedia "OSI model", "Internet protocol suite (TCP/IP)", "Transmission Control Protocol", "Domain Name System", "Transport Layer Security", "HTTP", "Docker (software)", "cgroups", "Linux namespaces".
- For authority, the relevant **IETF RFCs** (Request For Comments): TCP is RFC 9293; HTTP/1.1 is RFC 9112; TLS 1.3 is RFC 8446.
- The course's byte-faithful fetch tooling (`IASWebFetch`) and the "why not summarise" note are the worked example of §4.

8. Exercises

E00N.1 For each of the six protocols in §3, name the OSI layer (2–7) it lives at, and one thing that would break if it were removed (e.g. "without DNS you would have to type IP addresses").

E00N.2 Run `curl -v https://example.com/` on your laptop (the `-v` prints the handshake). Match each block of output to a step in §4 (DNS, TCP `SYN`, TLS certificate, the `GET` line, the `200`). Which step do you *not* see, and why? (Hint: DNS often happens in a separate resolver.)

E00N.3 `docker pull` an image once, then pull it again and time both. Explain the difference using §5 (content-hashed layers). Then state, in DPRAM terms (§6), why a lab that pulls a 2 GB image over a slow link is "bytes-bound" rather than "round-bound".

Chapter 6

00S serialization rpc

U00S — Serialization & RPC: how a value becomes bytes, and a call becomes a message

The ScaDaMaLe book, preliminary unit S (the engineering floor, right after U00N). Our own exposition; the `.proto` snippets are quoted from the Apache-2.0 A2A `a2a.proto`. Sources in §7.

The one idea of this unit: *the whole "agent protocol" is two small answers stacked.* A socket carries only a flat byte stream, so two agents in different processes (maybe different languages, vendors) need (1) **serialization** — turn a structured value into bytes and back — and (2) **RPC** — make "ask the other agent to do X and give me the result" feel like a function call. Everything A2A and MCP add sits on top of those two answers. Once you see them, both standards read as *assemblies*, not magic.

> **Entry kit** — what this chapter assumes; follow a link if you need the memory jog (map: Figure 1):
> U00N (the network stack this rides on — HTTP over TCP/TLS over IP)

1. Acronyms (extending U00N's master table)

New here, on top of U00N §1:

Acronym	Expansion
protobuf	Protocol Buffers — a schema-first, compact binary serialization format
IDL	Interface Definition Language (the `.proto` schema file — the *contract*)
`protoc`	the Protocol-Buffers *compiler* (generates typed classes from a `.proto`)
gRPC	a high-performance RPC system: protobuf over HTTP/2
HTTP/2	the multiplexed successor to HTTP/1.1 (many streams on one connection)
SSE	Server-Sent Events (one-way server→client streaming over HTTP)
varint	variable-length integer encoding (small numbers use fewer bytes)
stub	the code-generated local proxy that makes a remote call look local

2. Serialization: structure ↔ bytes

Two agents must exchange a *structured* value — a message with an id, a role, some parts. Figure 6.1 is the whole job: **serialize** → **transport** → **deserialize**, landing the *same* value on the far side.

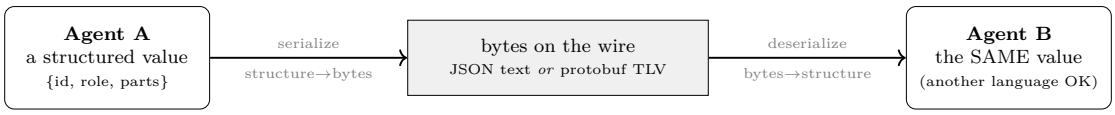


Figure 6.1: Serialization is the floor's whole job: a structured value on Agent A becomes a flat byte stream (self-describing JSON text, or schema-first protobuf tag-length-value), crosses the network of U00N, and is reconstructed as the *same* value on Agent B — possibly a different language or vendor. RPC (§3) then just puts a method name + params in that value and correlates a reply; A2A and MCP are exactly this, standardised.

2.1 JSON — self-describing text

JSON encodes a value as UTF-8 text, and the **field names travel with the data**, so it is self-describing — no shared schema needed to *parse*:

```
{"message_id":"7f3a","role":"agent","parts":[{"text":"build finished"}]}
```

Human-readable, ubiquitous, schema-optional. But: verbose (every key repeated on every message), few built-in types, and both sides must agree on the *meaning* of fields informally (nothing stops a typo'd key).

2.2 Protocol Buffers (protobuf) — schema-first + compact binary

Protobuf splits the two concerns JSON fuses: a **schema (IDL)** in a `.proto` file (the *contract*), and a compact **binary encoding** on the wire (the *bytes*). You declare typed messages once; `protoc` **code-generates** classes in many languages, so a Python agent and a Kotlin agent agree on shapes *by construction*. Here are `Part` and `Message` from the A2A standard's `a2a.proto`, abridged (the full `Part` also has `data` and `metadata`; the full `Message` has eight fields, among them `task_id`, `metadata`, `extensions` and `reference_task_ids`):

```
message Part {
  oneof content {          // exactly ONE is set (a tagged union)
    string text = 1;       // the number 1 is the FIELD TAG, not the name
    bytes  raw  = 2;
    string url  = 3;
    // (abridged: `data` omitted)
  }
  // (abridged: `metadata` omitted)
  string filename   = 6;
  string media_type = 7;   // e.g. "text/plain", "image/png"
}
message Message {
  string message_id = 1;   // de-dup / idempotency key
  string context_id = 2;   // groups a conversation thread
  Role   role       = 4;   // enum: USER / AGENT
  repeated Part parts = 5; // repeated = a list
  // (abridged: task_id, metadata, extensions, reference_task_ids omitted)
}
```

How it becomes bytes (worth hand-tracing). Protobuf is a sequence of **(field-number, wire-type, value)** triples — a *tag-length-value* encoding. The **field number**, not the name, goes on the wire, so `message_id`'s name never costs bytes; integers use **varint** (small = fewer bytes). A `Message` with `message_id="7f3a"` and `role=AGENT(2)` encodes field 1 as `[tag 0x0A][len 0x04][37 66 33 61]` (the four ASCII bytes of the string `7f3a`) and field 4 as `[tag 0x20][0x02]`

— a handful of bytes, no names. That is why protobuf is small and fast.

Why field NUMBERS give compatibility. Because the wire carries numbers, you can add `field 9` later: old readers see an unknown tag and *skip* it; new readers of old data use the default. This **forward/backward compatibility** is why schema-first formats dominate long-lived, multi-party systems — exactly the agent-interop case. (Rule: never renumber or reuse a field number.) `oneof` (tagged union), `repeated` (list), and `enum` (whose `0` is `*_UNSPECIFIED`, so "unset" is distinguishable) are the three structural blocks you will see everywhere.

2.3 The trade-off

	JSON	Protobuf
schema	optional (self-describing)	required (`.proto` IDL)
wire form	UTF-8 text (names repeated)	binary tag-length-value (numbers)
size / speed	larger / slower	smaller / faster
cross-language	parse anywhere; agree on meaning informally	code-gen → types agree by construction
evolution	ad hoc	field numbers → safe add/skip

(Same family, briefly: Avro, Thrift, MessagePack, CBOR — schema-or-self-describing × text-or-binary.)

3. RPC: calling the other agent

Remote Procedure Call makes "run X over there, give me the result" look like a local call. Two members matter for agent protocols.

3.1 JSON-RPC 2.0 — a tiny, transport-agnostic envelope

JSON-RPC is *not* a transport; it is a 4-field envelope you can carry over stdio, HTTP, a WebSocket, or a Unix socket. A request and its reply:

```
{"jsonrpc":"2.0","id":42,"method":"SendMessage",
 "params":{"message":{"messageId":"7f3a","role":"ROLE_USER",
                      "parts":[{"text":"build finished"}]}}}

{"jsonrpc":"2.0","id":42,"result":{"message":{"messageId":"9c01",
                                    "role":"ROLE_AGENT","parts":[{"text":"ack"}]}
                                     }}
```

(The names follow the A2A specification's JSON-RPC binding: PascalCase method names that match the gRPC service, `ROLE_USER` for the client's message and `ROLE_AGENT` for the server's.)

The `id` **correlates** the reply to the request (many can be in flight); a request **with no `id` is a notification** (fire-and-forget, no reply); requests may be **batched** (an array); errors are structured (`{"error":{"code":-32602,"message":"Invalid params"}}`; `-32600..-32603` reserved). This *is* the JSON-RPC binding of **both A2A and MCP** — the whole agent call is one such envelope.

3.2 gRPC — the high-performance member

gRPC = **protobuf over HTTP/2** with code-generated **stubs** and four call shapes (unary, server-/client-/bidirectional-streaming). The A2A standard defines its whole service in the same `a2a.proto`:

```
service A2AService {
  rpc SendMessage(SendMessageRequest) returns (SendMessageResponse);
  rpc SendStreamingMessage(SendMessageRequest) returns (stream StreamResponse);
  rpc GetTask(GetTaskRequest) returns (Task);
}
```

`stream` on the return type is server-side streaming (HTTP/2 carries many frames on one connection) — how A2A streams task progress without polling.

3.3 JSON-RPC vs gRPC

	JSON-RPC 2.0	gRPC
payload	JSON text	protobuf binary
transport	anything (stdio/HTTP/socket)	HTTP/2
streaming	no (add SSE/WebSocket)	native (4 shapes)
tooling	trivial (hand-writable)	code-gen stubs
weight	minimal, easy	heavier, faster

Same idea, different weight class. A2A offers *both* (plus HTTP+JSON) as **bindings** of one abstract service; you pick per deployment.

4. The payoff: A2A and MCP are assemblies of this floor

- **A2A** (Agent-to-Agent) = the `a2a.proto` **schema** (Message/Part/Task/AgentCard) exposed over a chosen **binding** — JSON-RPC (§3.1), gRPC (§3.2), HTTP+JSON, or a *custom* one (this course's own Unix socket is one). Discovery via an **AgentCard** listing `{URL, transport, binding}`.
- **MCP** (Model Context Protocol) = **JSON-RPC 2.0** over a transport (stdio locally, HTTP remotely), with the same "custom transports MUST preserve the JSON-RPC message format" rule.

Once you have §2 + §3, both standards read as obvious assemblies — the whole point of putting this on the floor before the agentic units (U17; U18~§10, which names both protocols) build on it.

5. Tie to the cost spine (DPRAM)

This *is* the DPRAM message-passing model at the systems layer: **serialization = bytes moved** (protobuf shrinks the constant), **RPC = the message-passing round**, **transport = the channel**, and **the `messageId` idempotency key = round accounting + de-dup** — the practical face of **at-least-once vs exactly-once** delivery, a classic distributed-systems result made concrete.

LLM lens — where this unit lands on the running workload (U0L):
An LLM tool-call is exactly this floor: the model emits a JSON-RPC request (§3.1) naming a tool + arguments (a serialized value, §2); the runtime runs it and returns a JSON-RPC result; MCP is the standard that fixes the envelope. "How many tokens in, how many out" is the bytes-moved term; "how many tool round-trips" is the rounds term — U00N §6 again, one layer up.

5b. One lens: the resolution lattice

One lens unifies much of what follows. Text has a lattice of **resolution rungs** — byte, codepoint, normalization form, glyph, semantics, and the *statistical distribution of tokens.* A mark or

an attack can live at any rung: an invisible zero-width character hides at the **byte** rung and can be *stripped*; a shift in the token distribution hides at the **statistical** rung and *survives paraphrase* — unstrippable. A defender's habit worth naming is the **Rumsfeld matrix**: separate known-known covert marks (we detect them), known-unknowns (we know a class exists but cannot yet detect it — the statistical rung), and unknown-unknowns (marks we do not yet know exist). Teaching the shape of one's own ignorance is rare and worth doing.

Two ideas recur book-wide. First, a security transform worth trusting is **total, idempotent, and provenance-preserving** — defined on every input, unchanged by re-application, and it keeps the raw as recoverable evidence. Second, **every byte crossing in from an untrusted party is suspect**: you inspect it and keep the raw. U15~§6b makes this concrete — the keyless watermark *detector* and the covert-character *scrub* are the same L_1/total-variation geometry on this same lattice, one working at the statistical rung, the other at the byte rung.

6. Exercises

E00S.1 Hand-encode `Part{text:"hi"}` as protobuf (field 1, a string) and count the bytes; compare to the JSON `{"text":"hi"}`. Which wins, and by how much, as the text grows to a paragraph?

E00S.2 Add a field `priority` to `Message`. What number do you give it, and why must you never reuse an old one? Argue that an old reader still parses a new message.

E00S.3 Write the JSON-RPC 2.0 request that cancels a task with id `t-9`, and the success response. Now write it as a *notification* — what changes, and what do you give up?

E00S.4 A2A `SendMessage` is offered over JSON-RPC, gRPC, and HTTP+JSON. For (a) two co-located agents, (b) a browser client, (c) a high-throughput server mesh — pick a binding and justify each in one sentence (tie it to §2.3 / §3.3).

7. Sources

- The A2A specification and the MCP specification — their web pages archived byte-faithfully in `designs/research/agentic-protocols-2026-08-30/`. The A2A schema file `a2a.proto` (Apache-2.0) lives in the A2A GitHub repository and is *not* in that archive (its fetch was blocked), so the field numbers in our snippets are quoted from it, not checked against the archive. Our own exposition; the snippets are quoted, not republished wholesale.
- JSON-RPC 2.0: the public `jsonrpc.org` specification.
- Same-family formats for optional reading: Wikipedia "Protocol Buffers", "gRPC", "JSON-RPC", "Comparison of data-serialization formats".

Chapter 7

01 reductions scan

U1 — Reductions & scan (all-prefix-sums)

The ScaDaMaLe book, unit 1 (examined core). Our own exposition; original sources credited in §8.

The one organising idea (revisited from U0): *the same algorithmic problem is re-solved as the machine model gains a resource; what changes is the cost you must account for.* U1 takes the most fundamental data-parallel primitive — the *scan*, or all-prefix-sums — and walks it through all three lenses. After this unit, the PRAM section of every later unit will invoke scan as a solved subroutine.

Entry kit — what this chapter assumes; follow a link if you need the memory jog (map: Figure 1):
F.M1 (the recurrences behind the scan bounds)

1. Intuition — why scan looks sequential

Take a sequence of numbers $a_1, a_2, \ldots, a_n$ and an associative binary operation $\oplus$ (addition, multiplication, max, exclusive-or, …). The *prefix-sum problem* asks for all n values

$$s_k \;=\; a_1 \oplus a_2 \oplus \cdots \oplus a_k, \quad k = 1, \ldots, n.$$

(We call this an *inclusive* scan. The *exclusive* variant shifts by one and gets its own symbol: $x_k = a_1 \oplus \cdots \oplus a_{k-1}$ for $k = 1, \ldots, n$, where the empty combination $x_1 = e$ is the identity. In words, x_k is everything strictly before position k, so $x_k = s_{k-1}$ with $s_0 := e$. Both are equally fundamental.) Figure 7.1 is this unit's central algorithm in one picture; the rest of the unit earns its two panels.

On one processor the recurrence $s_k = s_{k-1} \oplus a_k$ is unavoidable: each s_k depends on s_{k-1}, so the sequential algorithm is a left-to-right pass — a *fold* — costing $T(n) = n - 1$ operations. This looks irreducibly sequential: step k cannot start until step $k - 1$ finishes.

The key insight is that *associativity allows re-bracketing.* The value s_k equals $(a_1 \oplus \cdots \oplus a_{k/2}) \oplus (a_{k/2+1} \oplus \cdots \oplus a_k)$. Intermediate partial sums can be computed on different processors and then combined — at the cost of some redundant work. The art is to find the schedule that does this re-bracketing with *no extra total work* and depth only $O(\log n)$. That schedule is the Blelloch two-sweep scan.

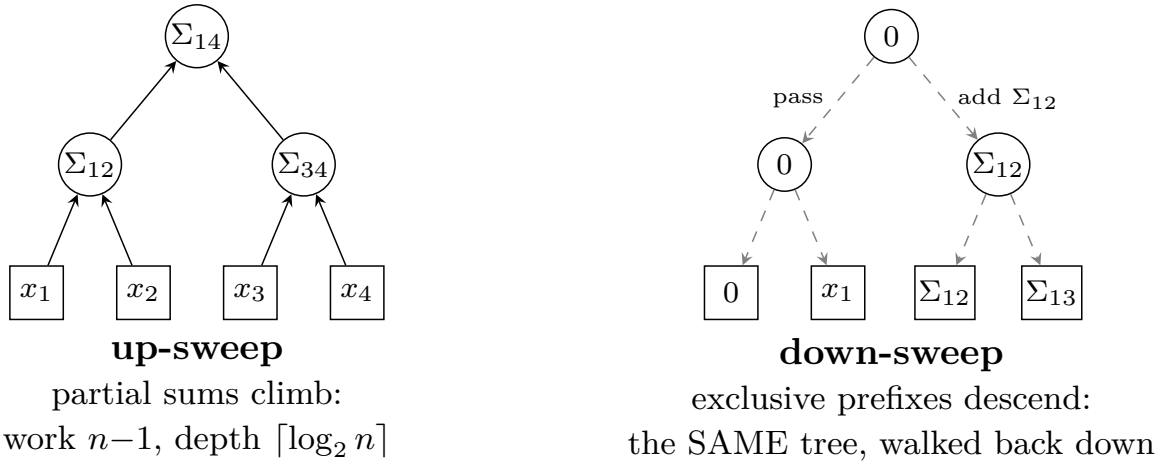


Figure 7.1: Blelloch's work-efficient scan ($n = 4$): two passes over one balanced tree. The *up-sweep* computes partial sums bottom-up (this is the PRAM panel of Figure 1.1 verbatim); the *down-sweep* walks the same tree top-down, passing each left child its parent's incoming value and each right child the parent's value plus the left sibling's stored partial sum — delivering every *exclusive prefix* $0, x_1, \Sigma_{12}, \Sigma_{13}$ in $O(n)$ total work and $O(\log n)$ depth. One tree, two directions: reduction is the up-sweep alone; scan is the round trip.

Check your reading.
Q1. The prefix-sum recurrence $s_k = s_{k-1} + a_k$ suggests scan is inherently sequential. What breaks this apparent barrier?
(A) a faster processor (B) associativity: partial results over blocks can be combined in a balanced tree, in any bracketing (C) commutativity: the inputs may be reordered arbitrarily (D) caching the input array

2. RAM — the sequential baseline

Model. Random-Access Machine: one processor, unit-cost ops on machine words.

Reduction (fold). Given $(a_1, \ldots, a_n)$ and associative $\oplus$ with identity e, return $a_1 \oplus \cdots \oplus a_n$. The sequential fold:

```
acc = e
for i = 1 to n:
    acc = acc ⊕ a[i]
return acc
```

$T_{\text{seq}}^{\text{reduce}}(n) = n - 1$ operations.

Scan (linear pass). Return the array $(s_1, \ldots, s_n)$ of all prefixes:

```
s[1] = a[1]
for i = 2 to n:
    s[i] = s[i-1] ⊕ a[i]
```

$T_{\text{seq}}^{\text{scan}}(n) = n-1$ operations — same as reduction. Knowing *all* prefixes costs exactly the same as knowing the last one. This equality will persist in the parallel setting, and that is the surprising fact U1 is built around.

Why scan matters. Scan is the engine behind: - **Pack / filter** — output only elements satisfying a predicate: scan the indicator array, use prefix sums as output indices. - **Radix-sort digit pass** — count-then-scatter needs per-digit offset arrays = a scan. - **List-rank and tree**

operations — Euler-tour + scan linearises pointer-based structures for PRAM. - **Segmented operations** — scan within variable-length segments (the primitive for irregular data-parallel computations).

Check your reading.
Q2. Which of the following is NOT naturally expressed as a scan/reduction?
(A) computing running account balances (B) stream compaction (keep marked elements, packed left)
(C) radix-sort's counting step (D) finding whether a specific key exists via binary search on a sorted array

3. PRAM — Blelloch two-sweep scan

3.1 The reduction building block

Before the full scan, observe that *reduction* on n values achieves work-optimality at depth $O(\log n)$ via a balanced binary tree: pair up neighbours, reduce each pair in parallel, repeat on the $n/2$ partial results. Level l has $n/2^l$ operations, so:

$$W^{\text{reduce}} = \sum_{l=1}^{\log_2 n} \frac{n}{2^l} = n - 1 = O(n), \qquad D^{\text{reduce}} = \log_2 n = O(\log n).$$

Work-optimal (matches T_{seq}); depth $O(\log n)$. By Brent (U0): $T_p \leq (n-1)/p + \log n$, giving speed-up $\approx p$ when $p \ll n/\log n$. EREW PRAM suffices (each processor reads distinct cells).

3.2 The Blelloch scan — THE CENTERPIECE

The insight (Blelloch 1990): run the tree *down* after running it *up*. Two passes over a perfect binary tree on $n = 2^k$ leaves:

Up-sweep (reduce phase). Build partial sums bottom-up. At each internal node, store the $\oplus$ of its entire subtree. After the up-sweep the root holds $a_1 \oplus \cdots \oplus a_n$.

Down-sweep (distribute phase). Propagate prefix information top-down. Each node receives from its parent a value representing "the $\oplus$ of everything to the *left* of my subtree." It passes its *left child* that same value, and passes its *right child* that value $\oplus$ (its left child's up-sweep result).

Worked example: $n = 8$, addition.

Input: $a = [3, 1, 7, 0, 4, 1, 6, 3]$.

After up-sweep — tree of partial sums (leaf level = inputs; each node = sum of its subtree):

```
Level 0 (leaves):  3   1   7   0   4   1   6   3
Level 1 (pairs):     4       7       5       9
Level 2 (quads):         11              14
Level 3 (root):                  25
```

Each level-l node at position j holds $a_{j\cdot 2^l+1} \oplus \cdots \oplus a_{(j+1)\cdot 2^l}$. Operations at level l: $n/2^l$ — the levels halve, so the total is $n/2 + n/4 + \cdots + 1 = n - 1$ (here 7 for $n = 8$), matching the work bound $W = 2(n-1)$ once the down-sweep's $n - 1$ ops are added.

Down-sweep — initialise root to identity $e = 0$, then for each node with incoming value v and stored subtree-sum t: left child gets v, right child gets $v \oplus t_{\text{left}}$.

```
Level 3 (root):    0  [set to identity]
Level 2:           0            11
```

```
Level 1:           0    4       11    16
Level 0 (leaves):  0  3   4  11  11 15  16 22
```

Read the leaves: $[0, 3, 4, 11, 11, 15, 16, 22]$. These are the *exclusive* prefix sums $x_1, \ldots, x_8$ of §1 ($x_k = a_1 + \cdots + a_{k-1} = s_{k-1}$, with $x_1 = s_0 = 0$). The inclusive sums $s_1, \ldots, s_8$ follow by adding each input to its own leaf, $s_k = x_k + a_k$: $[3, 4, 11, 11, 15, 16, 22, 25]$. Both are obtained in the same two sweeps.

Cost of the Blelloch scan:

$$W^{\text{scan}} = 2(n-1) = O(n), \qquad D^{\text{scan}} = 2\log_2 n = O(\log n).$$

Work-optimal: $W = O(n) = O(T_{\text{seq}})$. Depth $O(\log n)$. The claim that "knowing all prefixes is no harder than knowing one" is now concrete: the two-sweep algorithm doubles the constant in W but leaves the asymptotic complexity unchanged. EREW PRAM suffices.

3.3 Work-optimality — why the constant matters in practice

A naive parallel scan assigns processor i the task of computing $s_i = a_1 \oplus \cdots \oplus a_i$ directly by a sequential fold — total work $\Theta(n^2)$, depth $O(n)$. An early parallel scan (Hillis–Steele 1986) achieves $D = O(\log n)$ but at work $W = O(n \log n)$: it re-uses fewer partial results and recomputes many subtotals. Blelloch's two-sweep is one of the classic scans that are simultaneously *work-optimal* and *depth-optimal* (up to constants) for this problem.

The Hillis–Steele constant matters too: on a Spark cluster with 100 partitions, $O(n \log n)$ total work means $\log n \approx 7$ times more CPU time per stage. Work-optimality is not a theoretical nicety; it is a cluster-budget decision.

3.4 Monoid generality

The algorithm requires only that $\oplus$ is *associative* with identity e — a *monoid*. Examples: $(\mathbb{Z}, +, 0)$; $(\mathbb{R} \cup \{-\infty\}, \max, -\infty)$; $(\{0,1\}^*, \text{concat}, \varepsilon)$; $(\text{Sets}, \cup, \emptyset)$. Commutativity is *not* required; the down-sweep preserves left-to-right order exactly. Floating-point addition is **not** associative in IEEE 754 — rounding makes $(a \oplus b) \oplus c$ differ from $a \oplus (b \oplus c)$, whereas in exact arithmetic addition *is* associative — so reducing/scanning in a different tree shape can change the result; the lab revisits this.

3.5 Segmented scan

A *segmented scan* applies an independent scan within each of k variable-length segments, all in a single pass over the array. Encode segment boundaries with a flag array f ($f_i = 1$ iff a_i starts a new segment). The lifted operation on pairs (v, f) is:

$$(v_1, f_1) \oplus' (v_2, f_2) \;=\; (f_2 \,?\, v_2 \;:\; v_1 \oplus v_2,\; f_1 \vee f_2).$$

This is associative (check by cases). Apply the same Blelloch tree to the lifted pairs: $W = O(n)$, $D = O(\log n)$, independent of k. Segmented scan is the key primitive for irregular data-parallel computations: it lets variable-length records be processed as if they were uniform, a trick used in sparse-matrix-vector products, graph BFS frontiers, and list-rank.

3.6 Applications of scan

Application	How scan is used
Pack / filter	Compute indicator $f_i \in \{0,1\}$; exclusive-scan → output indices; scatter.
Radix-sort digit pass	Count-sort per digit: histogram → exclusive-scan → scatter. $O(\log n)$ depth per pass.
List-rank	Euler-tour of a tree → array; scan computes depth/weight in $O(\log n)$ depth.
Parallel string parsing	Segment-bracket matching with a monoid on stack-depth changes.
Sparse BLAS (SpMV)	Segment boundaries = row starts; segmented-scan accumulates products per row.
Running totals in a DataFrame	A cumulative-sum window column is an inclusive scan (distributed: §4.2–4.3).
Stream compaction	Another name for pack: the exclusive scan of the keep-flags gives each kept row its output position.
Packing sequences for an LLM batch	Concatenate variable-length token sequences; the exclusive scan of their lengths gives each sequence's start offset, which the batched kernels use as segment boundaries (a segmented layout, §3.5).

Check your reading.

Q3. In Blelloch's work-efficient scan, the down-sweep exists because the up-sweep alone yields:
(A) all inclusive prefixes, but unsorted (B) only the total (the root) and block partials – the per-position exclusive prefixes still have to be distributed back down the same tree (C) the maximum element only (D) nothing usable

Q4. For n = 1024 = $2^1$0 inputs, the up-sweep performs exactly how many additions?
(A) 512 (B) 1023 (C) 1024 (D) 10240

Q5. Give one operation that is associative but not commutative, and one that is commutative but not associative. For each, say whether Blelloch's scan applies.

4. DPRAM / MPC — distributed scan

4.1 Tree-aggregate (reduction at scale)

In the MPC model, each of M machines holds n/M elements. When the M partial results fit in one machine's memory (the MPC accounting used again in §4.2), a global reduction takes **one** round: every machine sends its local total to one machine. A binary machine-tree aggregation takes $O(\log M)$ rounds instead, but keeps every machine's fan-in at two, which matters when one receiver cannot absorb M messages. Spark: `RDD.treeReduce(f, depth)` — internally a multi-level tree of reduces. Round count = the tree depth, $O(\log M)$ for a binary tree; local work per round = $O(n/M)$; communication per round = $O(M\,w)$ bytes, w = size of one partial result (one word for a sum), on a critical path of one message. When $M \ll n$, the $O(\log M)$ rounds dominate.

Spark's default `reduce` is a *linear* aggregation (driver collects all): $O(M)$ communication to the driver, $O(M)$ sequential combines at the driver — a single point of contention. `treeReduce` is strictly better in both depth and driver load for large M. At $M = 1000$, the tree takes $\lceil \log_2 1000 \rceil = 10$ rounds, while the linear aggregation delivers 999 messages to one receiver, which then performs 999 combines in sequence. U0 §4.3 prices the eight-number case under the α-β model: $2\alpha + 16\beta$ on the critical path for four machines.

4.2 Distributed scan — 2-round algorithm

Given M machines each holding a contiguous slice of n elements, compute the global all-prefix-sums in **2 rounds** (Blelloch-style translated to MPC):

Round 1 — local scans + per-machine totals. Each machine i runs a sequential scan on its n/M local elements, producing local prefix sums and a local total $T_i = \bigoplus_{\text{local}} a_j$.

Between rounds — scan of totals. The M totals $(T_1, \ldots, T_M)$ are small ($M \ll n$); they are gathered to one machine (in Spark, a `collect` to the driver), which scans them, producing the exclusive prefix $P_i = T_1 \oplus \cdots \oplus T_{i-1}$ for each machine i, and sends P_i back to machine i. (A `treeReduce` cannot do this step: it returns only the grand total, not the M prefixes.)

Round 2 — local fix-up. Machine i adds its offset P_i to every local prefix sum: $s_{i,k} \leftarrow P_i \oplus s_{i,k}$. No further communication is needed.

Cost. 2 rounds *in the MPC accounting* (the M totals fit on one machine, so the between-round gather/scatter is part of the round boundary). Local work per round: $O(n/M)$. Communication: $O(M)$ totals exchanged — negligible relative to the data. One engineering nuance: in Spark the totals step is a separate small job (a `collect` of M numbers to the driver; the M offsets travel back inside the fix-up tasks) — negligible in bytes, no shuffle, but visible in the Spark UI as an extra job. Compare to a naïve one-at-a-time global scan that would require $O(M)$ sequential rounds of inter-machine messages.

In Spark this pattern appears as: `mapPartitions` (local scan → local total), a `collect` of the M totals (the driver scans them), then `mapPartitionsWithIndex` (machine i applies offset P_i). Exposing it explicitly is a code-quality exercise (E1.6).

4.3 Scan in real pipelines

Distributed scan appears in: - **Radix sort digit pass** — distributed radix-sort (U2) depends on a distributed count-scan to compute scatter offsets before the shuffle. - **Sparse feature encoding** — assign globally unique feature indices across shards by scanning per-shard feature counts. - **Window aggregations** — partial running totals per partition, corrected by a global scan of partition-end values (common in Flink/Spark Structured Streaming). - **Parallel prefix compression** in column-store databases: delta encoding computed via distributed scan.

The pattern is always the same: *local scans + scan of per-machine totals + local fix-up* — the 3-step structure introduced above. Recognising it when it is hidden inside a library call is a data-engineering skill this unit targets.

Check your reading.
Q6. On p machines under the alpha-beta model, a TREE allreduce of one value costs about:
(A) p(alpha + beta) (B) log2(p) rounds of (alpha + beta)-class messages (C) alpha + beta * p^2 (D) zero – allreduce is free

5. The invariant — instantiated for scan

RAM: how many ops? — $T(n) = n-1$ (one left-to-right fold; no shortcut). **PRAM:** how much work W, how deep D? — Blelloch two-sweep: $W = O(n)$ (work-optimal), $D = O(\log n)$; associativity is the only requirement; segmented scan generalises to variable-length segments at the same cost; **Brent:** $T_p \leq 2(n-1)/p + 2\log n$. **DPRAM/MPC:** how many rounds R, how many bytes? — 2 rounds (local scan + offset fix-up), $O(M)$ bytes exchanged (the per-machine totals), local work $O(n/M)$ per round; the dominant cost is the single between-round gather, not the local computation.

Check your reading.
Q7. Fill the scan row of the invariant box from memory: RAM cost, PRAM work/depth, DPRAM rounds/bytes for scanning n items on p machines.

6. Lab sketch — `labs/lab1-alpha-beta/`

Lab 1 — ping-pong + tree vs ring allreduce (`labs/lab1-alpha-beta/`): follow its README; `bash run-lab.sh` runs everything in the course lab image and writes your receipt (Canvas-submittable). The experiments below are the lab's intellectual content — run and extend them inside the same image.

The lab prices the *wire* of this unit's distributed reduction (§4.1); it runs no Spark code. The local work is already priced by §3; what the lab measures is the communication term.

Experiment 1 — ping-pong: measure α and β. Two containers on one Docker network time round trips for message sizes from 1 byte to 16 MB (medians over many repetitions), and a least-squares fit of $T(n) = \alpha + \beta n$ gives your machine's α (per message) and β (per byte), with R^2 for the fit. A container bridge is a loopback baseline: α is microseconds, and a slightly negative fitted intercept is noise plus extrapolation, not physics.

Experiment 2 — tree vs ring allreduce (analysis, no new code). An *allreduce* is a reduction (§3.1, §4.1) whose result every machine receives. Plug your β and $\alpha/2$ (your fitted α comes from round trips, so it is about twice the one-way α these models assume) into the two classic cost models for p workers and an n-byte vector:

$$T_{\text{tree}} \approx 2\log_2 p\,(\alpha + \beta n), \qquad T_{\text{ring}} \approx 2(p-1)\,\alpha + 2\,\frac{p-1}{p}\,\beta n.$$

In words, the tree is this unit's reduction tree run up (reduce) and back down (broadcast), $2\log_2 p$ rounds each carrying the whole vector; the ring takes $2(p-1)$ rounds, but each carries only n/p bytes. Find which wins at $n = 1$ KB and at $n = 100$ MB, and the crossover n: small vectors are latency-bound (the tree wins), large ones bandwidth-bound (the ring wins).

Experiment 3 — the one-line takeaway. $p-1$ one-byte messages cost $\approx (p-1)\alpha$; one n-byte stream costs $\alpha + \beta n$. Which dominates for your β and one-way $\alpha/2$ at $n = 1$ MB, $p = 16$? This is §4.1's linear aggregation, which pays $M-1$ message latencies at one receiver, priced on your own machine.

Deliverable. The receipt, with one-sentence comments: the α, β model note and the tree-vs-ring crossover. Racing real tree and ring allreduce across containers is an optional deep-dive later in the course.

7. Lean certificate (proofs via our prover)

Shipped (exercise form): the scan work/depth theorem, in `provedog/exercises/wasp-module1-scan/` — a reference solution (`ScanCost_solution.lean`), a student blank (`ScanCost_blank.lean`, one `sorry`), and a checker (`check-scan.sh`). The statement:

```
-- from ScanCost_blank.lean (sweepWork k = combine-ops of one sweep on 2^k
  leaves;
-- scanWork k = 2 * sweepWork k; scanDepth k = 2 * k):
theorem blelloch_scan_cost (k n : ℕ) (hn : n = 2 ^ k) :
    scanWork k = 2 * (n - 1) ∧ scanDepth k = 2 * k
```

The proof goes by induction on k through the lemma `sweepWork_eq` (one sweep costs $2^k - 1$): base case $k = 0$ computes; the inductive step analyses one level of the sweep and closes with `omega`.

Exercise U1.L1 (light): supply the blanked base case in `ScanCost_blank.lean`; verify with `check-scan.sh` (ACCEPT = typechecks, sorry-free — the kernel is the grader).

Exercise U1.L2 (deep): Prove work-optimality as a corollary: `scanWork`$(k) = O(n) = O(T_{\text{seq}}(n))$ for $n = 2^k$. State and prove the analogous result for segmented scan.

Note: the reference solution exists and typechecks but sits OUTSIDE the sorry-free `lake build IasCert` ledger (deliberately — it is an exercise, not a shipped cert); the book marks this theorem *stated + exercise-checked*, not machine-checked-in-the-cert-ledger.

> **LLM lens** — where this unit lands on the running workload (U0L):
> Distributed LLM training IS this unit at scale: gradient synchronisation is an allreduce (a reduction over machines), layernorm is a reduction over a vector, and the ring-vs-tree allreduce choice is priced by the α-β cost models that Lab 1 (§6) has you evaluate with your own measured α and β. This unit supplies the tree; U18 supplies the ring and prices why it is bandwidth-optimal.

8. Sources (credited; archived under `scadamale-book-sources`)

Primary: Blelloch, G.E. (1990). *Prefix Sums and Their Applications.* Technical Report CMU-CS-90-190, Carnegie Mellon University. — The foundational reference for the two-sweep algorithm, segmented scan, and the application catalogue. Our exposition follows the structural logic of that report but is written anew.

Supporting: JáJá, J. (1992). *An Introduction to Parallel Algorithms*, ch. 4. Addison-Wesley. — rigorous treatment of PRAM reduction and scan including lower bounds.

Wikipedia: *Prefix sum* — accessible overview of applications; the Hillis–Steele and Blelloch variants are both described with clear diagrams.

MIT OCW 6.046 *Design and Analysis of Algorithms* — lecture notes on parallel algorithms; covers the balanced-tree reduction and scan in a modern algorithms course context.

9. Exercises

Light (examined core):

E1.1 Write out the Blelloch two-sweep for input $[2, 5, 3, 8]$ ($n = 4$) using max as $\oplus$. Produce both the up-sweep tree and the down-sweep tree; read off the exclusive-max prefix array. Verify by hand that each entry equals the running max to its left.

E1.2 Let $\oplus$ be matrix multiplication of 2×2 matrices over $\mathbb{R}$. Is this a valid monoid for the Blelloch scan? Give the identity element. Does the scan still run correctly in $O(\log n)$ depth? What is the work in terms of the cost of a single matrix multiply?

E1.3 A work-suboptimal parallel scan achieves $D = O(\log n)$ but uses work $W = O(n \log n)$ (Hillis–Steele). For $n = 10^9$ and $p = 1000$ processors, compare the Brent-predicted runtimes of the two algorithms. At what p does the suboptimal algorithm *never* beat the sequential fold, regardless of D?

E1.4 Implement pack (filter) using a scan in pseudocode: given array $a[1..n]$ and predicate P, output the subsequence of elements satisfying P in original order. State the work and depth of your algorithm on the PRAM, and the round count on the MPC model with M machines.

Deep (Module-3 material):

E1.5 (Lower bound) Prove that any EREW PRAM algorithm that computes all n prefix sums requires depth $\Omega(\log n)$. *Hint:* consider the information-flow DAG: how many outputs depend on a_1?

E1.6 (Segmented scan design) A dataset of N variable-length records is stored in a flat array with a flag marking each record start. Design a PRAM algorithm using segmented scan that computes, for each record, the maximum element in that record. Analyse W and D in terms of N and the total number of elements n. Implement in Spark (RDD-level, `mapPartitions` + a segmented-scan helper) and measure the stage count.

E1.7 (Lean extension) State and prove in Lean that the lifted segmented-scan operation $\oplus'$ (§3.5) is associative. Use this to argue formally that the Blelloch tree applied to pairs (v, f) computes a valid segmented scan.

Chapter 8

02 sorting

U2 — Sorting: RAM → PRAM → DPRAM

The ScaDaMaLe book, unit 2 (examined core). Our own exposition; original sources credited in §8.

Instantiating the three-lens invariant. The U0 box for sorting:

> **RAM:** the best comparison-based sort costs $\Theta(n \log n)$ comparisons — a lower bound, not just a convention. The interesting RAM questions are constants, cache, and randomisation (quicksort / quick-select). **PRAM:** sorting n keys with p processors costs total work $W = O(n \log n)$ and depth $D = O(\log^2 n)$ (achievable); $D = O(\log n)$ is known (Cole 1988, state-only). Brent links the two: with $p = \Omega(n)$ processors the wall-clock is polylogarithmic. **DPRAM/MPC:** sorting is the backbone of every distributed join and aggregation. The dominating cost is the **one all-to-all shuffle**; the game is choosing $k - 1$ splitters that keep it balanced. Sample sort achieves $R = O(1)$ rounds with communication $O(n)$ in total ($O(n/M)$ per machine in expectation) and deviation controlled by the sampling rate.

Entry kit — what this chapter assumes; follow a link if you need the memory jog (map: Figure 1): F.M1 (recurrences and the master theorem) • F.S2 (Hoeffding, for the splitter-balance analysis)

1. Intuition — sorting as data routing

Sorting is not merely an algorithmic benchmark; at scale it is the primary mechanism for aligning data — records that must meet (for a join, a group-by, a co-group) are sorted to the same partition, so sorting and shuffling are the same operation viewed from different angles.

The progression across the three machines crystallises this: - **RAM:** keys are moved in memory, comparisons drive the structure. The question is: how many compares? - **PRAM:** many pairs of keys can be compared simultaneously. The question becomes: how many *rounds* of simultaneous comparison until all keys settle? - **DPRAM:** keys live on different machines. Moving a key means a network message. The question is: can we route every key to its correct machine using *one* shuffle, regardless of the initial distribution?

The answer at each level is yes, with a precise cost — and the costs compose: a PRAM sort of each local shard after a single DPRAM shuffle is how distributed sort actually works.

Figure 8.1 is the distributed algorithm of this unit in one picture.

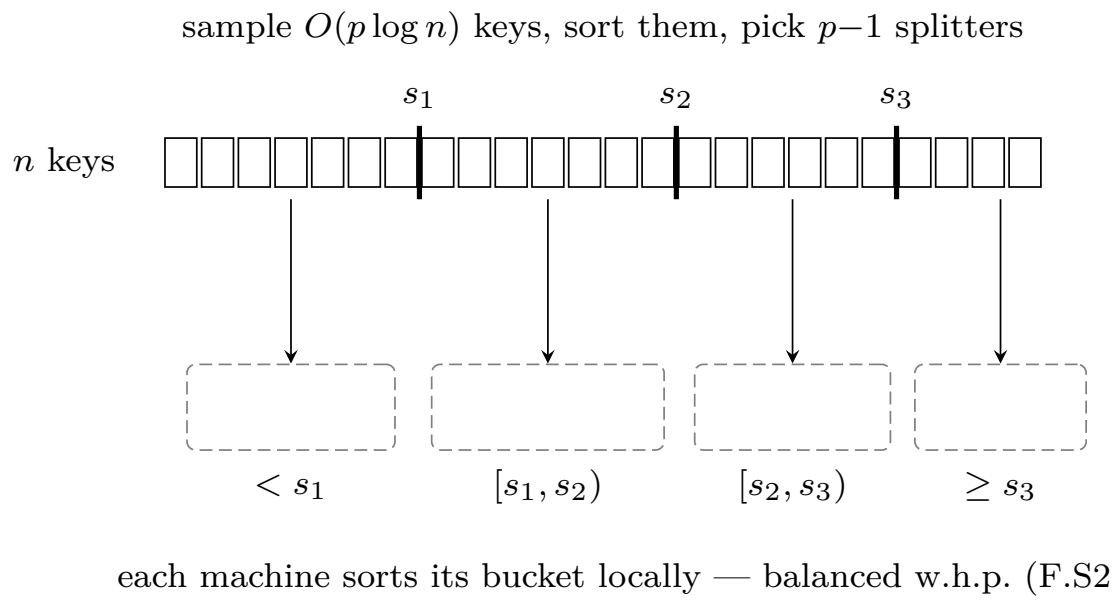


Figure 8.1: Sample sort in one picture: sample a few keys, sort the sample, choose $p-1$ splitters, and let the splitters partition the whole bar into p buckets — one shuffle round — each sorted locally. The entire correctness burden sits on one probabilistic claim: random splitters balance the buckets, which is Hoeffding on indicator variables (F.S2, E.FS2.1). The same splitter-and-bucket logic reappears when an MoE router batches tokens by destination expert (U0L).

Check your reading.
Q1. The unit opens with the claim that at scale, sorting and shuffling are 'the same operation viewed from different angles'. What does this mean?
(A) both take O(n log n) time, so they are interchangeable in cost accounting (B) a shuffled (random) array is the worst-case input for quicksort (C) records that must meet (for a join, a group-by, a co-group) are routed to the same partition by sorting on the key – moving every key to its correct machine IS the sort, and that one all-to-all shuffle is the dominating cost (D) Spark implements sortByKey by shuffling records to random partitions

2. RAM — sequential sorting: mergesort, quicksort, quick-select

2.1 Mergesort and quicksort (cost recap)

Both are canonical divide-and-conquer comparison sorts; we state their costs and move on.

Mergesort splits an array of n elements into two halves, sorts each recursively, and merges. Merge of two sorted arrays of total length n costs $n-1$ comparisons. The recurrence $T(n) = 2T(n/2) + O(n)$ resolves to $T(n) = O(n \log n)$ in the worst case. Mergesort is *deterministic* and *stable*; its constant factor is moderate. (Reference: CLRS ch. 2, 4.)

Quicksort chooses a pivot, partitions into elements $\leq$ pivot and $>$ pivot, recurses on each part. Worst case is $O(n^2)$ (sorted input with a bad pivot strategy); with a random pivot the expected cost satisfies $E[T(n)] = O(n \log n)$. The recurrence for expected comparisons $C(n)$ satisfies $C(n) = n - 1 + \frac{2}{n}\sum_{k=0}^{n-1} C(k)$, which yields $C(n) = 2n \ln n - O(n)$ (approximately $1.386\, n \log_2 n$). In practice, quicksort's cache behaviour and in-place partition make it fast. Spark's within-partition sort, however, is not quicksort: it is TimSort (a merge-based hybrid), or a radix sort on key prefixes where that applies. (Reference: CLRS ch. 7.)

2.2 Quick-select: the $O(n)$ median (expected)

Problem. Given n elements and a rank k, find the k-th smallest element without sorting all n.

Algorithm (Hoare 1961). Partition around a random pivot: elements $\leq$ pivot go left, elements $>$ pivot go right. If the pivot lands at position j: - if $j = k$, return the pivot; - if $j > k$, recurse on the left part; - if $j < k$, recurse on the right part.

Only one side is recurred on each level, unlike quicksort.

Claim. $E[T(n)] = O(n)$.

Proof idea. Call a pivot *good* if it lands in the middle half of the current subarray — then the recursive subproblem has size at most $3/4$ of the current one. A uniform random pivot is good with probability $1/2$, so in expectation at most 2 partition passes are paid at each *phase* (a phase = the work until the problem shrinks by the factor $3/4$). Phase i operates on size at most $(3/4)^i n$, and a partition pass costs at most its subarray size in comparisons, so the expected total is $E[C(n)] \leq \sum_{i \geq 0} 2 \cdot (3/4)^i n = 2n \cdot 4 = 8n = O(n)$. $\square$

(Deterministic $O(n)$ worst case is achievable via the median-of-medians algorithm, at a larger constant; we do not need it here.)

Quick-select is the RAM step inside PRAM parallel selection and inside the splitter-finding subroutine of sample sort.

2.3 The comparison lower bound

Theorem. Any deterministic comparison-based sorting algorithm uses $\Omega(n \log n)$ comparisons in the worst case.

Proof. A comparison-based sort is a binary decision tree: each internal node is one comparison; each leaf is one of the $n!$ permutations of the input. The tree has depth $\geq \lceil \log_2 n! \rceil$. By Stirling: $\log_2 n! = n \log_2 n - n \log_2 e + O(\log n) = \Omega(n \log n)$. $\square$

This makes mergesort and quicksort (in expectation) asymptotically optimal. It is also why *any* parallel comparison sort must do $\Omega(n \log n)$ comparisons in total, and hence $\Omega(n \log n/p)$ per processor on p processors: the lower bound is on comparisons, and parallelism only divides them among the processors.

Check your reading.
Q2. The Omega(n log n) lower bound for comparison-based sorting views any such algorithm as a binary decision tree. The load-bearing count in the proof is:
(A) the number of internal nodes, which is n - 1 for a balanced tree (B) the number of leaves: at least n! (one per input permutation), forcing depth $\geq$ ceil(log2 n!) = Omega(n log n) by Stirling (C) the number of array writes, which is at least n for any sort (D) the number of processors, which must be at least n for parallel sorting

3. PRAM — parallel sorting: parallel merge and parallel mergesort

3.1 Parallel merge (the rank-based idea; JaJa §3)

Input. Two sorted arrays $A[1..m]$ and $B[1..n]$, $m \leq n$, sharing $p = m + n$ processors.

Goal. Produce $C[1..m+n]$, the sorted merge.

Key idea. The position of $A[i]$ in C is $i + \text{rank}_<(A[i], B)$, where $\text{rank}_<(x, B) = |\{j : B[j] < x\}|$ is found by binary search on B in $O(\log n)$ time; the position of $B[j]$ is $j + \text{rank}_\leq(B[j], A)$, with

$\text{rank}_{\leq}(y, A) = |\{i : A[i] \leq y\}|$. One side counts strictly and the other weakly, so equal keys never collide: ties go to A first. (With $\leq$ on both sides, $A = [1]$ and $B = [1]$ would both be written to $C[2]$.)

Algorithm. 1. (Parallel) For each $i \in [m]$, compute $r_i = \text{rank}_{<}(A[i], B)$ by binary search: $O(\log n)$ depth, $O(m \log n)$ work. 2. (Parallel) For each $j \in [n]$, compute $s_j = \text{rank}_{\leq}(B[j], A)$ by binary search: $O(\log m)$ depth, $O(n \log m)$ work. 3. (Parallel) Write $A[i] \to C[i + r_i]$ and $B[j] \to C[j + s_j]$: $O(1)$ depth, $O(m + n)$ work.

Costs. Work $W = O((m + n) \log(m + n))$ — *not* work-optimal relative to the $O(m + n)$ sequential merge. Depth $D = O(\log n)$ on CREW PRAM (concurrent reads on A, B are needed by multiple processors simultaneously).

Work-optimal parallel merge requires more structure (splitting A and B into blocks and merging small pairs sequentially); the work-optimal version achieves $W = O(n + m)$, $D = O(\log(m + n))$ on CREW. We state this and refer to JaJa §3.4 for the construction; the rank-based version above is the clearer building block.

3.2 Parallel mergesort

Construction. Run the recursive mergesort tree in parallel: at each level of the recursion, all independent merges run simultaneously. There are $\log n$ levels; at level i from the bottom there are $n/2^i$ merge problems each of size 2^i. Using parallel merge at each level:

- **Work.** Each level contributes $W_{\text{level}} = O(n \log n)$ (rank-based) or $O(n)$ (work-optimal). With the rank-based merge: $W = O(n \log^2 n)$; with work-optimal merge: $W = O(n \log n)$ — work-optimal with respect to sequential mergesort.
- **Depth.** Each merge at level i has depth $O(\log 2^i) = O(i)$. Levels run sequentially in the recursion tree, so $D = \sum_{i=1}^{\log n} O(i) = O(\log^2 n)$.

Stated result (parallel mergesort, standard construction). $W = O(n \log n)$, $D = O(\log^2 n)$ on CREW PRAM.

Existence note (Cole 1988). Cole's parallel mergesort achieves $W = O(n \log n)$, $D = O(\log n)$ using a pipelined merge that overlaps multiple levels simultaneously. The construction is technically involved; we state the result and cite it. (Reference: Cole 1988, SIAM J. Comput. 17(4):770–785, *Parallel merge sort.*)

Brent speed-up (instantiated). With work-optimal parallel mergesort ($W = O(n \log n)$, $D = O(\log^2 n)$) on p processors:

$$T_p \;\leq\; \frac{W}{p} + D \;=\; O\!\left(\frac{n \log n}{p}\right) + O(\log^2 n).$$

Speed-up is within a constant of p as long as $p \ll n \log n / \log^2 n = n / \log n$, i.e. the parallelism of this algorithm is $\Theta(n / \log n)$. Adding processors beyond $n / \log n$ returns no speed-up: depth dominates.

Check your reading.
Q3. Parallel mergesort with the work-optimal merge has W = O(n log n) and D = $O(log^2$ n), so Brent gives $T_p \leq$ W/p + D. Beyond roughly how many processors does adding more return no further speed-up? (A) p ~ n / log n – the algorithm's parallelism W/D (B) p ~ n log n – one processor per comparison (C) p ~ log^2 n – one processor per level of depth (D) p ~ sqrt(n)

4. DPRAM / MPC — distributed sorting: sample sort and TeraSort shape

This is the **centrepiece of U2**: the single algorithm that every distributed data system (Hadoop, Spark, Flink) runs as its sort primitive.

4.1 Why not just use PRAM merge on each partition?

In the MPC model, data is partitioned across M machines each holding $s = n/M$ keys. A naive approach: sort each partition locally, then merge across machines in a tournament tree. But every merge step requires one communication round (fetching elements from neighbours). With M machines, the merge tree has $\log M$ levels: $R = \Theta(\log M)$ rounds. For $M = 10^4$ machines that is 14 shuffle stages — unacceptable in practice. We need $R = O(1)$.

4.2 Sample sort: the $O(1)$-round distributed sort

High-level shape. Reduce the global sort to: *find $M - 1$ splitter values that partition the key space into M roughly equal buckets; send each key to its bucket; sort locally.*

Each of those three steps is $O(1)$ rounds if done correctly, so the total is $R = O(1)$.

Algorithm (sample sort, our presentation).

Let n be the total number of keys, M machines each with $s = n/M$ keys initially, and a desired oversampling rate t (we will set $t = O(\log n)$).

1. **Local sort.** Each machine sorts its s local keys. Cost: $O(s \log s)$ local work, 0 communication.
2. **Sample splitters.** Each machine draws t equally-spaced elements from its sorted local array (every $\lfloor s/t \rfloor$-th element), yielding Mt candidate splitters across all machines. Collect all candidates at a designated coordinator machine (or via a tree-aggregate); sort the Mt candidates locally; keep every t-th one to form $M-1$ global splitters $v_1 < v_2 < \cdots < v_{M-1}$.

   Communication this step: Mt keys collected = Mt messages. With $t = O(\log n)$ and $M = O(n/s)$: total bytes = $O(n \log n/s)$. If $s = \Omega(\sqrt{n \log n})$ this fits within one machine's memory. **Rounds: 1** (the gather; returning the splitters is step 3).
3. **Broadcast splitters.** The coordinator sends the $M - 1$ splitters to all machines: one more round, of only $O(M)$ bytes (in Spark it is a broadcast variable; the all-gather variant below avoids it). Each machine now knows which bucket each of its keys belongs to.
4. **Shuffle: route keys to their bucket.** Each key is sent to the machine owning its bucket. This is the **one all-to-all shuffle** — in Spark, a `repartitionAndSortWithinPartitions` with the splitter-defined range partitioner.

   Communication this step: every key moves exactly once, so total bytes = $O(n)$. Per machine: send and receive $O(s)$ keys in expectation. **Rounds: 1.**
5. **Local sort of received keys.** Each machine sorts its received keys: $O(s' \log s')$ local work, where s' is the bucket size. 0 additional communication.

Total: $R = 3$ communication rounds with the coordinator of steps 2–3 (gather the samples, return the splitters, shuffle the keys), or $R = 2$ in the all-gather form below (exchange the samples, shuffle the keys); $O(n)$ total communication bytes, $O(s \log s)$ local work per machine. (Often written $R = O(1)$ to stress the constant is independent of n and M.)

Counting the rounds carefully. "2" assumes that every machine can compute the splitters itself. In the **all-gather** variant each machine sends its t samples to every other machine ($M(M-1)t$ keys in all; small when Mt is small). Then every machine sorts the same Mt candidates, derives the same splitters, and no broadcast is needed. The rounds are (1) samples and (2) the shuffle. With a **coordinator**, as in steps 2–3 above, there are three exchanges: gather the samples, send back the splitters, shuffle. That is still $O(1)$, and the extra round carries only $O(M)$ bytes. The shuffle is the only round that moves $\Theta(n)$ bytes; it is the price.

Worked: $M = 2$ machines, $t = 2$. Eight keys, four per machine.

step	M1	M2
input	8, 3, 5, 1	2, 7, 4, 6
local sort	1, 3, 5, 8	2, 4, 6, 7
sample (every 2nd)	3, 8	4, 7
candidates, sorted	3, **4**, 7, 8, so splitter $v_1 = 4$ (the t-th = 2nd)	the same, computed locally
after shuffle (≤ 4 / > 4)	1, 3, 2, 4	5, 8, 6, 7
local sort	1, 2, 3, 4	5, 6, 7, 8

As data moving:

```
                    M1                                  M2
local sort          1 3 5 8                             2 4 6 7        no network
ROUND 1  samples    3 8  ============================>  3 8            2 keys
  cross
                    4 7  <============================  4 7            2 keys
                      cross
         both sort  3 4 7 8 -> v1 = 4                   3 4 7 8 -> v1 = 4
ROUND 2  shuffle    keep 1 3 (<= 4)                     keep 6 7 (> 4)
                    5 8  ============================>  5 8            2 keys
                      cross
                    2 4  <============================  2 4            2 keys
                      cross
local sort          1 2 3 4                             5 6 7 8        no network
output              1 2 3 4 | 5 6 7 8   (read M1, then M2)
```

A few samples cross in round 1; half of the data crosses in round 2 (the shuffle, the $O(n)$ bytes); the buckets are 4 and 4, balanced because the splitter was taken from the sample. Note the tie rule: this example sends a key equal to the splitter to the *lower* bucket ($\leq v_1$), whereas the 12-key example below and the lab code (`bisect_right`) send it to the *upper* one. Either rule sorts correctly, provided every machine applies the same one; only the balance can change (here < 4 / ≥ 4 would give buckets of 3 and 5).

Where this runs in practice. A Spark `sortBy` / `orderBy` is this algorithm (§4.4). The same sample-then-shuffle shape sits under a sort-merge join and under a range-partitioned write.

Worked example: 12 keys, $M = 3$ machines, $t = 2$ samples each. Each machine holds $s = 4$ keys and samples every $s/t = 2$nd element of its sorted array. The coordinator sorts the $Mt = 6$ candidates and keeps the t-th, $2t$-th, …, $(M-1)t$-th (here the 2nd and 4th) as splitters. A key goes to bucket = number of splitters $\leq$ key (`bisect_right` in the lab code).

```
machine  input        local sort    sample (every 2nd)
m1       7 2 9 4      2 4 7 9       4 9
m2       1 8 5 12     1 5 8 12      5 12
m3       11 3 6 10    3 6 10 11     6 11
coordinator sorts 4 5 6 9 11 12, keeps the 2nd and 4th: splitters 5 | 9
------- ONE shuffle: bucket(key) = number of splitters <= key -------
```

```
bucket 0 (< 5):      1 2 3 4       4 keys  -> sorted locally
bucket 1 (5 to 8):   5 6 7 8       4 keys  -> sorted locally
bucket 2 (>= 9):     9 10 11 12    4 keys  -> sorted locally
```

Reading the buckets in order gives the sorted output. With the coordinator there are three rounds (collect the samples, return the splitters, shuffle the keys), or two in the all-gather form, and every key moves once, so $O(n)$ bytes. The **balance factor**, the largest bucket divided by n/M, is $4/4 = 1.00$, which is perfect.

4.3 Skew and the role of splitter sampling

The output of step 4 is globally sorted for any splitters applied the same way on every machine; how well the splitters separate the key distribution decides the balance. If the distribution is uniform, $t = 1$ splitter per machine suffices in expectation. But real datasets are **skewed**: a small number of keys (Zipfian keys, popular products, trending hashtags) appear far more often than others. A machine assigned the bucket for a popular key receives far more than s keys — this is the **skew problem**, and it causes the one shuffle to produce imbalanced partitions, straggler tasks, and OOM failures.

How oversampling controls skew. By sampling t elements per machine (rather than 1), we draw Mt candidates from the distribution and take quantile estimates. The k-th quantile estimate from Mt i.i.d. samples of an unknown distribution F has deviation $O(1/\sqrt{Mt})$ from the true k/M quantile with high probability (standard DKW-type concentration; stated here informally under the i.i.d.-sampling idealisation — the adversarial-placement analysis is deep-exercise material). Note that this is an *absolute* error in probability mass, while a bucket's target mass is $1/M$; relative to the bucket it is $\sqrt{M/t}$, small only when $t \gg M$. The sharper argument is per bucket: a bucket can exceed $(1+\epsilon)\,n/M$ keys only if a key range of that mass caught at most t of the samples instead of the $(1+\epsilon)\,t$ it expects. By a Chernoff bound this has probability $e^{-\Theta(\epsilon^2 t)}$, and a union bound over the M buckets, with failure probability polynomially small in n, gives $\epsilon = O(\sqrt{\log n/t})$. Setting $t = c \log n$ therefore gives a balance factor $1 + O(1/\sqrt{c})$ with high probability: a constant factor, close to 1 for a large constant c. A balance factor $1 + o(1)$ needs $t = \omega(\log n)$. The oversampling cost (step 2) grows as $O(Mt) = O(n \log n/s)$, which fits in memory when $s = \Omega(\sqrt{n \log n})$.

Non-sortable keys (heavy hitters). Keys that appear $\Omega(s)$ times cannot be separated by any splitter — every machine that holds a copy of the heavy key belongs to the same bucket. The standard fix: identify heavy hitters (by a count-sketch pass, U4) and route them to dedicated machines. This is out of scope here; note the dependency on U4.

Worked example: what the sample buys. Take the same 12 keys with only $t = 1$ sample per machine. Each machine sends its every 4th key, which is its maximum:

```
samples 9 12 11 -> sorted 9 11 12 -> splitters 9 | 11
bucket 0 (< 9):     1 2 3 4 5 6 7 8   8 keys
bucket 1 (9, 10):   9 10              2 keys
bucket 2 (>= 11):   11 12             2 keys     balance = 8 / (12/3) = 2.00
```

The output is still correctly sorted. A worse sample makes a *slower* sort, not a wrong one, because the largest bucket becomes the straggler that the whole job waits for. Lab 2 draws a random sample of $32p$ keys and takes splitters at positions $\lfloor (i+1)|\text{sample}|/p \rfloor$ (0-indexed). Applied to the six $t = 2$ samples above, that convention picks $6 \mid 11$ and gives buckets of 5, 5 and 2 (balance 1.25). Small samples are sensitive to the splitter convention; the oversampling analysis above is why $32p$ keys suffice.

4.4 TeraSort and the Hadoop / Spark shape

TeraSort (O'Malley 2008, the Hadoop sort benchmark) is sample sort as above, applied to 10^{12} bytes. The sampling step collects a fixed number of candidate splitters, sorts them on the coordinator, and writes a partition file; each mapper reads the partition file (the broadcast) and partitions its output accordingly; reducers receive sorted partitions and write them. One MapReduce job, $R = 1$ shuffle.

In Spark the same shape appears as:

- `sortByKey(numPartitions)` — uses a `RangePartitioner` that samples the RDD (by default about 20 keys per output partition, $\min(20p, 10^6)$ in total for p output partitions, so the sample does not grow with n; each input partition is reservoir-sampled with a 3x margin and re-sampled if it turns out skewed), sorts and quantiles the samples, assigns each key to a range partition, shuffles, sorts within each partition. The sampling is the splitter step; the shuffle is step 4; the within-partition sort is step 5.
- `repartitionAndSortWithinPartitions(partitioner)` — explicit control: the caller supplies the partitioner (the splitter-based `RangePartitioner` for sort), Spark fuses the partition-routing and local-sort into a single shuffle write, avoiding a second pass.

The round count in Spark. Each shuffle is one stage boundary. Sample sort = 1 shuffle = 1 stage boundary beyond the local sorts. `sortByKey` plus a subsequent join would be 2 shuffles. The Spark DAG UI makes this observable; the lab measures it directly.

4.5 Communication lower bound (intuition)

In any sorting algorithm on the MPC model, if each machine starts with $s = n/M$ keys uniformly at random and the final sorted order places the n/M smallest keys on machine 1, then in expectation $(M-1)/M$ of those keys started on some other machine — at least $(n/M) \cdot (M-1)/M \approx n/M$ keys must be sent to machine 1. Summing over all machines: at least $\Omega(n)$ bytes must cross the network in total, summed over all rounds; with R rounds, some round moves $\Omega(n/R)$ bytes. Therefore:

> **Claim (distributional).** For inputs whose keys start uniformly spread across machines, any distributed sort on the MPC model moves $\Omega(n)$ bytes in expectation, in total; with $R = O(1)$ rounds, some round moves $\Omega(n)$ of them. Sample sort achieves both bounds: $R = 2$ or 3 rounds ($O(1)$) and $O(n)$ total bytes. (The distributional assumption is load-bearing; see exercise E2.7 for the argument and its scope.)

Check your reading.

Q4. Sample sort runs on M = 100 machines with oversampling rate t = 50. In step 2, how many candidate splitters are collected at the coordinator, and how many global splitters are kept?
(A) 100 candidates, 99 splitters (B) 5000 candidates, 100 splitters (C) 50 candidates, 49 splitters (D) 5000 candidates, 99 splitters

Q5. Real key distributions are skewed (Zipfian keys, trending hashtags). What does oversampling – t = O(log n) samples per machine instead of 1 – buy sample sort?
(A) it reduces the number of shuffle rounds below O(1) (B) quantile estimates from Mt i.i.d. samples deviate by only O(1/sqrt(Mt)) from the true quantiles, so buckets are near-balanced with high probability (C) it reduces the total shuffle traffic below O(n) bytes (D) it guarantees that heavy hitters (keys appearing Omega(s) times) are split across machines

Q6. A Mixture-of-Experts layer routes each token to its top-k experts and batches tokens by destination expert. In two or three sentences, map this onto this unit's vocabulary: what are the keys, the buckets, the shuffle, and what is the skew problem?

5. The invariant (U2 instantiation)

RAM: sorting costs $\Theta(n \log n)$ comparisons (lower bound tight; mergesort / quicksort achieve it); randomised quick-select finds the k-th order statistic in $O(n)$ expected time.

PRAM: parallel merge achieves depth $O(\log n)$ (CREW); parallel mergesort achieves $W = O(n \log n)$, $D = O(\log^2 n)$ (standard); Cole's pipeline achieves $D = O(\log n)$ (stated). Brent: $T_p \leq O(n \log n/p) + O(\log^2 n)$; parallelism $\Theta(n/\log n)$.

DPRAM/MPC: sample sort achieves $R = 2$ communication rounds (sample all-gather + key shuffle; 3 with a coordinator that returns the splitters; $O(1)$ in the sense that the constant is independent of n and M) and $O(n)$ total communication; skew is controlled by $t = O(\log n)$ oversampling; Spark exposes this as `sortByKey` / `repartitionAndSortWithinPartitions`. The shuffle is the computation: route keys to the machine that will sort them — $\Omega(n)$ bytes is unavoidable.

Check your reading.
Q7. Fill the sorting row of the invariant box from memory: RAM comparisons, PRAM work/depth, DPRAM rounds/bytes for sorting n keys across M machines.

6. Lab sketch (`labs/lab2-sample-sort-spark/`)

Lab 2 — sample sort on Spark (`labs/lab2-sample-sort-spark/`): follow its README; `bash run-lab.sh` runs everything in the course lab image and writes your receipt (Canvas-submittable). The experiments below are the lab's intellectual content — run and extend them inside the same image.

Setup. One-time Docker install plus the course lab image — see `labs/README.md`; everything runs free and locally on your own laptop.

Experiment 1 — sample sort vs `sortByKey` on uniform keys.

Generate $n = 10^7$ uniform random integers (reproducible seed). Run both `repartitionAndSortWithinPartitions` with a `RangePartitioner` and a naive `sortByKey`. Collect from the Spark UI (or `SparkListener`): - number of shuffle stages (should be 1 for both — verify); - total shuffle bytes written and read; - per-partition record counts (measure balance).

Question to answer in the deliverable: do the two approaches produce the same number of shuffle bytes? Explain by tracing the Spark DAG.

Experiment 2 — skew experiment with Zipfian keys.

Generate $n = 10^7$ keys from a Zipf(α) distribution with $\alpha = 1.5$. Run `sortByKey` with default sampling and with a 10x oversampled `RangePartitioner`. Compare: - per-partition record counts (plot a histogram — does the default partitioner produce stragglers?); - maximum partition size vs minimum partition size; - stage wall-clock time.

Question: at what α does oversampling matter and why?

Experiment 3 — shuffle bytes as a function of n and M.

For $n \in \{10^6, 10^7, 10^8\}$ and $M \in \{2, 4, 8\}$ partitions: measure total shuffle bytes. Plot shuffle bytes / n. Does it converge to a constant? Explain from the $O(n)$ communication bound.

Deliverable. A notebook with the three experiments, plots, and a half-page cost-accounting in the three-lens vocabulary per experiment.

7. Lean certificate target

Current target (new — U2): certify the work and depth bounds for *parallel merge* (rank-based version, §3.1).

Formally, let `merge_depth : ℕ → Prop` assert that a sorted merge of two sorted arrays of total size n can be computed on a CREW PRAM in depth $O(\log n)$, and let `merge_work : ℕ → Prop` assert the work bound $O(n \log n)$. The Lean target is:

```
-- SCHEMATIC target statement (MergeAlg + names align to the BMLSPar.lean
  conventions at cert time):
-- MergeAlg must be instantiated to the rank merge's own cost profile (§3.1): a
  bare existential
-- over MergeAlg is vacuous (the zero profile satisfies it).
theorem parallel_merge_depth (n : ℕ) (hn : 2 ≤ n) :
    ∃ (alg : MergeAlg), alg.work n ≤ c_W * n * Nat.log 2 n
                      ∧ alg.depth n ≤ c_D * Nat.log 2 n := by
  sorry -- cert target: fill this in (lab U2.L1)
```

Exercise U2.L1 (light, Lean). Load `BMLSPar.lean` (already shipped) together with a stub defining `MergeAlg`, `c_W`, `c_D` (write it; reference: `provedog/exercises/wasp-module2-merge/MergeCost_solution.lean`; certificate: `IasCert.Book.Unit02`). Using those definitions, state the theorem above in the course Lean environment and verify that the `sorry` type-checks. Then: apply the Brent bound to the certified work and depth values to bound T_p — write the one-line calculation as a Lean `calc` block.

Exercise U2.L2 (deep, Lean). State and certify the round count of sample sort: `theorem sample_sort_rounds : sample_sort_rounds n M = 2` (or a suitable formalisation). Start from a formal MPC round definition analogous to the BSP superstep abstraction in `BMLSPar.lean`.

The point: certifying the round count is a *semantic* claim about the algorithm's communication structure, not just a count of LOC. This is where the machine-checked proof discipline pays off.

LLM lens — where this unit lands on the running workload (U0L):
The MoE router's top-k expert choice (U0L §3) is a selection problem, and batching tokens by destination expert is precisely the splitter-and-bucket logic of sample sort — same mathematics, keys are now tokens and buckets are experts.

8. Sources (credited; archived under `scadamale-book-sources`)

CLRS (ch. 7–8). Cormen, Leiserson, Rivest, Stein, *Introduction to Algorithms* (4th ed.). Ch. 7: quicksort, expected cost, randomised pivot. Ch. 8: lower bound for comparison-based sort; decision-tree argument. — *Cite only. No exposition reproduced.*

JaJa (§3). JáJá, *An Introduction to Parallel Algorithms*, Addison-Wesley, 1992. §3.1–3.4: parallel merge (rank-based + work-optimal), parallel mergesort, sorting networks. — The rank-based parallel merge in §3.1 above follows JaJa's construction.

Cole (1988). R. Cole, *Parallel merge sort*, SIAM J. Comput. 17(4):770–785, 1988. Optimal $O(\log n)$-depth parallel mergesort. — Cited for existence; construction not reproduced.

Blelloch samplesort. G. Blelloch, C. Leiserson et al., *A comparison of sorting algorithms for the connection machine CM-2*, SPAA 1991. — foundational PRAM samplesort; the distributed version generalises this.

TeraSort. O. O'Malley, *TeraByte sort on Apache Hadoop*, Yahoo! Research technical note, 2008. Available: https://sortbenchmark.org/Yahoo-Hadoop.pdf — the Hadoop benchmark that

makes sample sort concrete.

Wikipedia. *Merge sort*, *Quicksort*, *Samplesort*, *Sorting algorithm*, *External sorting.* — For background and cross-references; not the primary exposition.

MIT OCW 6.046. *Design and Analysis of Algorithms*, MIT OpenCourseWare. Lectures 2–4 cover comparison sorting and the decision-tree lower bound. — Freely available; credited.

9. Exercises

Light (examined core):

E2.1 State the Brent bound for parallel mergesort with $W = O(n \log n)$ and $D = O(\log^2 n)$. For $n = 10^9$ and $p = 1024$ processors, compute the two terms W/p and D. Which dominates? What is the maximum p for which the speed-up is within a factor 2 of ideal?

E2.2 Quicksort on a sorted input of n elements with a fixed (non-random) first-element pivot has cost $T(n) = O(n^2)$. Show the recurrence and solve it. Then explain why *random* pivot selection recovers $E[T(n)] = O(n \log n)$.

E2.3 In sample sort with M machines and oversampling rate t per machine: (a) what is the total number of candidate splitters collected in step 2? (b) How many global splitters are kept? (c) Why do we take every t-th candidate rather than the top $M - 1$ by value?

E2.4 Map the five steps of sample sort onto Spark primitives: which step corresponds to `sortWithinPartitions`? Which to `broadcast`? Which to the shuffle write? Draw the Spark DAG (stages and boundaries) for a `sortByKey` call on 4 partitions.

Deep (Module-3 / optional):

E2.5 *Quick-select lower bound.* Prove that any deterministic comparison-based selection algorithm (finding the k-th smallest of n elements) requires $\Omega(n)$ comparisons in the worst case. (*Hint:* an adversary argument — maintain a set of inputs consistent with all comparisons made so far; argue that until $\Omega(n)$ comparisons are done, the adversary can still swap the answer.)

E2.6 *Parallel merge, work-optimal version.* Describe (in 10–15 lines of pseudocode) a parallel merge of two sorted arrays of size n each, achieving $W = O(n)$, $D = O(\log n)$ on CREW PRAM. (*Hint:* split A into blocks of size $O(\log n)$; binary-search block boundaries into B; merge each block pair sequentially in parallel.) Verify that the work and depth recurrences give the stated bounds.

E2.7 *Shuffle bytes lower bound (intuition).* Consider any distributed sort algorithm on M machines where each machine starts with n/M keys drawn uniformly at random from $[n]$. The output must place the smallest n/M keys on machine 1. Show that in expectation at least $(1 - 1/M) \cdot n/M$ keys must leave their starting machine to reach machine 1. Conclude that any sorting algorithm requires at least $\Omega(n)$ bytes of total communication across all machines, summed over its rounds (so with R rounds some round moves $\Omega(n/R)$ bytes), and hence the $O(n)$ communication of sample sort is asymptotically optimal.

Chapter 9

03 scheduling

U3 — Scheduling & load-balancing: RAM → PRAM → DPRAM

The ScaDaMaLe book, unit 3 (examined core). Our own exposition; original sources credited in §8.

Instantiating the three-lens invariant. The U0 box for scheduling:

> **RAM:** assigning n independent jobs to p identical machines to minimise the maximum completion time (makespan) is NP-hard in the offline optimum. The interesting RAM question is: what does a simple greedy algorithm sacrifice? **PRAM:** greedy *list scheduling* achieves makespan $T \leq (2 - 1/p)\, T_{\text{OPT}}$ — a factor-2 approximation with a clean proof. This bound is the same Brent bound derived differently: the proof mechanism of Brent (U0) *is* Graham's argument. LPT ordering tightens the ratio to $4/3 - 1/(3p)$. **DPRAM/MPC:** data is already partitioned; the challenge is imbalance across partitions. Greedy least-loaded, work-stealing, and the power-of-two-choices heuristic address different manifestations of the same problem: a straggler holds up the entire stage.

Entry kit — what this chapter assumes; follow a link if you need the memory jog (map: Figure 1): F.M1 (asymptotics for the makespan bounds) • F.S2 (coupon collector and balls-in-bins tails)

1. Intuition — the straggler problem

Consider a Spark job with 1000 tasks spread across 100 executors. If one task holds data ten times larger than the others — a Zipfian key, a miscounted partition, a skewed shuffle — that executor finishes last, and all others wait. The cluster sits idle at 99% completion while one machine runs. This is the **straggler problem**, and scheduling is the art of preventing it.

The same tension appears at every scale: - A **compiler** scheduling instructions across CPU functional units (PRAM, p units) wants to minimise the longest pipeline delay. - A **cluster** distributing map tasks wants no executor to receive far more work than its share. - A **web server farm** routing requests wants to avoid sending the next long request to an already-busy server.

At each scale the algorithmic question is the same: given n jobs with known (or unknown) costs, assign them to p machines to minimise the time until the last job completes — the *makespan*. The constraints change, but the core tension — *balance vs. information* — does not.

Figure 9.1 is the 2-approximation proof as a bar chart; the rest of the unit makes it precise.

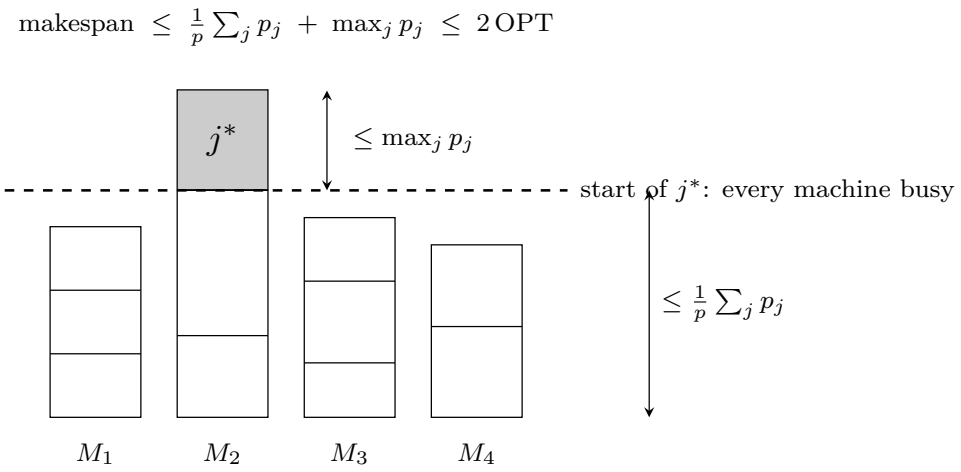


Figure 9.1: Graham's list scheduling as water-filling: greedy assignment keeps every machine busy below the waterline, so when the last job j^* starts, the water level is at most the average load $\frac{1}{p}\sum_j p_j \le \mathrm{OPT}$; the straggler adds at most $\max_j p_j \le \mathrm{OPT}$ on top. The 2-approximation is literally visible: the picture is the proof. Every distributed system in this book that "just uses a greedy scheduler" is standing on this bar chart.

Check your reading.
Q1. A synchronous round on p machines finishes when the LAST machine finishes. If per-machine times are random, increasing p while keeping per-machine work fixed makes the expected round time:
(A) decrease, because of parallelism (B) stay exactly constant (C) increase, because the maximum of more random variables grows (D) oscillate unpredictably

2. RAM — offline makespan minimisation: NP-hardness of the optimum

2.1 The problem (P || C_max)

Definition. Given n jobs with processing times $p_1, \dots, p_n > 0$ and m identical parallel machines (we use m for machines here, reserving p for processors in the PRAM section), an *assignment* is a function $\sigma : [n] \to [m]$ mapping each job to a machine. Preemption is not allowed: each job runs on its assigned machine from start to finish without interruption. The *makespan* of an assignment is

$$C_{\max}(\sigma) \;=\; \max_{k \in [m]} \sum_{j:\, \sigma(j)=k} p_j .$$

The problem $P \,\|\, C_{\max}$ asks: find σ^* minimising $C_{\max}$.

2.2 NP-hardness

Claim. $P \,\|\, C_{\max}$ is NP-hard for general m, even for $m = 2$.

Reduction (stated). The $m = 2$ case reduces from the **PARTITION** problem: given positive integers $a_1, \dots, a_n$, does there exist a subset $S \subseteq [n]$ with $\sum_{j \in S} a_j = \sum_{j \notin S} a_j = A/2$ (where $A = \sum_j a_j$)?

Set $p_j = a_j$ for each job j and $m = 2$ machines. An assignment σ achieves $C_{\max}(\sigma) = A/2$ if and only if the jobs assigned to machine 1 have total size $A/2$, i.e. if and only if PARTITION

has a YES instance. Since PARTITION is NP-complete, $P \,\|\, C_{\max}$ is NP-hard. □

Consequence. We cannot, in general, compute T_{OPT} in polynomial time (unless P = NP). The interesting RAM question is therefore not "find the optimum" but "how well can a fast algorithm do?"

2.3 Two lower bounds on the optimum

Before analysing any algorithm, note two unavoidable lower bounds on T_{OPT}:

$$T_{\mathrm{OPT}} \;\geq\; \max_{j \in [n]} p_j \tag{LB1}$$

— the longest job must run on some machine, so makespan is at least its length.

$$T_{\mathrm{OPT}} \;\geq\; \frac{1}{m} \sum_{j=1}^{n} p_j \tag{LB2}$$

— the total work must be spread across m machines; by pigeonhole at least one machine receives at least $1/m$ of the total.

Both lower bounds hold simultaneously; the effective lower bound is $\max(\max_j p_j,\ \frac{1}{m}\sum_j p_j)$. These two simple bounds are the backbone of the Graham proof in §3.

Check your reading.
Q2. Why does this unit prove a 2-approximation instead of computing the optimal schedule?
(A) the optimum does not exist (B) makespan minimisation is NP-hard, so no known polynomial algorithm finds the optimum; a provable approximation is the honest tractable goal (C) approximation algorithms are always faster in practice (D) the optimum requires knowing job lengths, which is impossible

3. PRAM — list scheduling: the Graham theorem (THE CENTERPIECE)

We now move to p processors (symmetric with PRAM terminology; the scheduling result applies equally to the multiprocessor and PRAM contexts).

3.1 The greedy list-scheduling algorithm

Algorithm (Graham 1966). Fix any ordering (a *list*) of the n jobs. Process the list in order: whenever a processor becomes free, assign it the next unassigned job in the list.

```
list = [job_1, job_2, ..., job_n]   # any ordering
free_time[k] = 0 for k in [p]
for job j in list:
    k* = argmin_k free_time[k]       # least-loaded processor
    assign job j to processor k*
    free_time[k*] += p_j
makespan = max_k free_time[k]
```

This is the *greedy list-scheduling* algorithm. Its running time on the RAM is $O(n \log p)$ (using a min-heap for the free times). The question is: how far from optimal is its makespan?

Why "greedy". A greedy algorithm takes, at each step, the choice that looks best now, with no look-ahead and no undoing. Here the rule is: *never leave a processor idle while a job is waiting.*

The list order is arbitrary, and the rule needs no knowledge of future jobs, not even their sizes, until they finish. That is why it is the default in cluster task schedulers and job queues. Greedy is a rule, not a promise of the optimum. The optimum is NP-hard to compute (§2.2), and what the rule buys is the guarantee of §3.2: within a factor $2 - 1/p$ of the optimum, whatever the order. U0 §3.2 gives the DAG version of the same rule, where it is the proof of Brent's bound.

3.2 The Graham theorem

Theorem (Graham 1966). *For any list L of n jobs with processing times $p_1, \dots, p_n$ on p identical processors, greedy list-scheduling achieves makespan*

$$T_L \;\le\; \left(2 - \frac{1}{p}\right) T_{\mathrm{OPT}}.$$

Proof. Let j^* be the job that finishes last — the job determining the makespan T_L — and let t be the time at which j^* *starts*. Under the greedy rule, no processor is ever idle while an unassigned job remains; in particular, throughout $[0, t)$ **every** processor is busy (had any been free before t, it would have taken j^* or an earlier job, and j^* would have started sooner). Moreover, all the work performed during $[0, t)$ is on jobs **other than** j^* (it has not started yet). Counting that work:

$$p \cdot t \;\le\; \sum_{j \neq j^*} p_j \;=\; \sum_{j=1}^{n} p_j \;-\; p_{j^*} \quad\Longrightarrow\quad t \;\le\; \frac{1}{p}\sum_{j=1}^{n} p_j \;-\; \frac{p_{j^*}}{p}.$$

The makespan is the start time of j^* plus its processing time:

$$T_L \;=\; t + p_{j^*} \;\le\; \frac{1}{p}\sum_{j=1}^{n} p_j \;+\; \left(1 - \frac{1}{p}\right) p_{j^*}.$$

Now apply the two lower bounds of §2.3 — LB2 to the first term ($\frac{1}{p}\sum_j p_j \le T_{\mathrm{OPT}}$) and LB1 to the second ($p_{j^*} \le T_{\mathrm{OPT}}$):

$$T_L \;\le\; T_{\mathrm{OPT}} + \left(1 - \frac{1}{p}\right) T_{\mathrm{OPT}} \;=\; \left(2 - \frac{1}{p}\right) T_{\mathrm{OPT}}. \qquad \square$$

The argument as a two-line chain:

1. **All processors busy before j^* starts, on jobs $\neq j^*$:** $t \le \frac{1}{p}\sum_j p_j - \frac{p_{j^*}}{p}$.
2. **Makespan = start + job:** $T_L = t + p_{j^*} \le \frac{1}{p}\sum_j p_j + (1 - \frac{1}{p})p_{j^*} \le (2 - \frac{1}{p})\, T_{\mathrm{OPT}}$.

The $-p_{j^*}/p$ term — available only because the busy interval excludes j^*'s own work — is exactly what sharpens the naive factor 2 to $2 - 1/p$. Note the mechanism: **two lower bounds on OPT + one accounting identity about the greedy schedule.** That is also precisely the shape of Brent's proof in U0 (§3.5 makes the correspondence exact).

3.3 Tightness of the bound

The factor $(2 - 1/p)$ is **tight** (exercise E3.5 works the general construction). The canonical instance: $p(p-1)$ jobs of size 1 followed by **one job of size** p. Greedy spreads the unit jobs evenly ($p - 1$ per machine, finishing at time $p - 1$), then the big job starts — makespan $2p - 1$. The optimum gives the big job its own machine and spreads the $p(p-1)$ units over the other

$p-1$ machines: $T_{\mathrm{OPT}} = p$. Ratio: $(2p-1)/p = 2 - 1/p$, exactly. Worked for $p = 2$: jobs $[1, 1, 2]$ in that list order — greedy: M1 gets 1 then 2 (makespan 3), M2 gets 1; optimum: the size-2 job alone on one machine (makespan 2). Ratio $3/2 = 2 - 1/2$.

3.4 LPT: a better ordering (stated only)

Longest Processing Time first (LPT). Sort jobs in non-increasing order of p_j before running greedy list-scheduling.

Theorem (Graham 1969; stated). LPT achieves

$$T_{\mathrm{LPT}} \;\le\; \left(\frac{4}{3} - \frac{1}{3p}\right) T_{\mathrm{OPT}}.$$

The proof requires a more careful analysis of which job finishes last in the LPT ordering; the ratio is strictly smaller than the general-list bound for all $p \ge 2$. We state this result and do not prove it here; the Graham (1969) paper contains the original proof.

Practical implication. LPT is the heuristic of choice when job sizes are known in advance (the *offline* setting). For $p = 10$: the general bound gives $T_L \le 1.9\, T_{\mathrm{OPT}}$; LPT gives $T_{\mathrm{LPT}} \le 1.3\overline{0}\, T_{\mathrm{OPT}}$. The improvement is material for large machines. For $p = 2$: $2 - 1/2 = 1.5$ against $4/3 - 1/6 = 7/6 \approx 1.17$.

LPT on the tight instance. Sorted longest-first, $[1, 1, 2]$ becomes $[2, 1, 1]$ on $p = 2$: M1 takes the 2, M2 takes a 1 and then the other 1. The makespan is 2, the optimum.

Why the order helps (the idea of Graham's proof). In the proof of §3.2 the overshoot beyond $\frac{1}{p}\sum_j p_j$ is $(1 - \frac{1}{p})\, p_{j^*}$, where j^* is the job that finishes last.

1. *j^* may be taken to be the last job in the list.* Jobs after j^* in the list do not change the start time of j^*. Deleting them leaves T_L unchanged and can only lower T_{OPT}.
2. *Under LPT the last job is the shortest.* If $p_{j^*} \le T_{\mathrm{OPT}}/3$, the proof of §3.2 gives $T_{\mathrm{LPT}} \le T_{\mathrm{OPT}} + (1 - \frac{1}{p})\, T_{\mathrm{OPT}}/3 = (\frac{4}{3} - \frac{1}{3p})\, T_{\mathrm{OPT}}$.
3. *The remaining case* is that every job exceeds $T_{\mathrm{OPT}}/3$. Then each machine holds at most two jobs in an optimal schedule, and one shows that LPT is optimal there. That case is the part we state and do not prove.

LPT needs the job sizes before it starts, so it is *offline*. Plain list scheduling does not.

3.5 Connection to Brent: greedy scheduling of a DAG

The Brent proof mechanism and Graham's argument are the same. In U0 §3.2 we proved the greedy bound $T_p \le (W - D)/p + D$ for a DAG of unit operations: each step is either *complete* (all p processors busy) or *incomplete* (it runs every ready operation, and so shortens the longest remaining path by one). The greedy rule there — never leave a processor idle while an operation is ready — is list scheduling applied to a job set with *precedence constraints*. (U0's first proof idea, level by level, is simpler but not greedy; it meets the same bound.)

Note that the independent-jobs theorem of §3.2 does not by itself give $W/p + D$: with unit jobs it reads $T_L \le W/p + (1 - 1/p)$, and it knows nothing of the order the DAG forces. The bridge is Graham's bound in its precedence form (the 1966 paper allows a precedence order on the jobs): if the longest chain of jobs has total length L, any greedy list schedule satisfies

$$T_L \;\le\; \frac{1}{p}\sum_{j=1}^{n} p_j \;+\; \Big(1 - \frac{1}{p}\Big) L.$$

The proof is the same accounting. Each moment of the schedule either has all p processors busy or has an idle processor; at an idle moment the greedy rule guarantees that some job of one fixed chain (traced back from the last-finishing job) is running. So the idle moments total some $I \le L$ and already account for at least I units of work, the all-busy moments total at most $(\sum_j p_j - I)/p$, and $T_L \le (\sum_j p_j - I)/p + I \le \frac{1}{p}\sum_j p_j + (1 - \frac{1}{p})L$. With unit operations, $\sum_j p_j = W$ and $L = D$, so

$$T_p \;\le\; \frac{W}{p} + \Big(1 - \frac{1}{p}\Big)D \;=\; \frac{W - D}{p} + D \;\le\; \frac{W}{p} + D,$$

which is the same formula as U0's certified bound (U0 §3.4, which certifies it only for the level-by-level schedule), and implies Brent's bound. The two bounds feed from the same pair of lower bounds: (LB1) the longest job, upgraded to the longest chain $T_{\mathrm{OPT}} \ge L$; (LB2) the total work divided by the number of processors. With both, the precedence form gives the same factor $2 - 1/p$. **U3 is the unit that shows students why the Brent bound in U0 was not a coincidence — it is the unit-job case of Graham's precedence-constrained list-scheduling bound.**

Check your reading.

Q3. Graham's water-filling proof bounds the makespan by two summands. They are:
(A) the number of jobs plus the number of machines (B) the average load ($\le$ OPT) up to the last job's start, plus that job's own length ($\le$ OPT) (C) the longest job times the machine count (D) the total work divided by the longest job

Q4. Jobs with total work 900 and longest job 100 are list-scheduled on p = 10 machines. Graham's bound guarantees the makespan is at most:
(A) 90 (B) 100 (C) 190 (D) 1000

Q5. Describe (or draw) a job family on p machines where list scheduling's makespan approaches twice the optimum. One sentence on what the adversarial structure exploits.

4. DPRAM / MPC — distributed load-balancing

In the MPC model the jobs are not given fresh — they arrive as a data distribution that was fixed (by a shuffle or a prior stage) across the machines. The challenge is not to *assign* jobs but to *rebalance* them without spending a full shuffle round every time.

4.1 Greedy least-loaded across machines

The simplest online strategy: when a new task arrives (or a machine becomes free), assign it to the machine with the smallest current load. This is the distributed analogue of the Graham greedy rule.

Analysis. Under the Graham theorem, with n tasks distributed across m machines by least-loaded, the makespan satisfies $T \le (2 - 1/m)\, T_{\mathrm{OPT}}$. In practice the bound is much better: if all jobs have the same size (the common case for partitioned RDDs), the algorithm is optimal.

When it fails. Greedy least-loaded requires knowing the current load of every machine — a synchronisation cost. In a fully distributed setting with m machines, a global least-loaded query requires $O(\log m)$ communication (a tree aggregate). Worse, if jobs arrive in batches (the shuffle model), the optimal assignment may not be achievable with a single round.

4.2 Stragglers: the DPRAM-specific pathology

A **straggler** is a task that takes far longer than the average — either because its input data is larger (skewed partition), its computation is more expensive (adversarial input), or the machine is slower (hardware heterogeneity). The straggler is the distributed scheduler's version of the

long job j^* in the Graham proof: it determines the makespan regardless of what the other tasks do.

Sources of stragglers in Spark: 1. **Partition skew** — one partition holds far more records (the sort skew problem from U2; caused by heavy-hitter keys or a bad splitter). 2. **GC pressure** — a machine with a full JVM heap pauses for garbage collection while holding a task. 3. **Hardware heterogeneity** — in cloud environments, VMs on different physical hosts have different I/O bandwidth and CPU speed.

4.3 Work-stealing: the dynamic answer

Work-stealing is the DPRAM analogue of online rebalancing: a machine that becomes idle *steals* tasks from a machine that has a long queue.

Concept. Each machine maintains a local double-ended queue (deque) of tasks. A machine processes its own tasks from the front of its deque. When it runs out, it picks a *victim* machine uniformly at random and steals a task from the *back* of that victim's deque. This asymmetry (own work: front; stolen work: back) ensures that stealing does not disrupt the victim's sequential execution order.

Why it works. Work-stealing guarantees that no machine sits idle for long: with high probability, a machine that runs out of work will find a victim with tasks to steal. The formal analysis (Blumofe–Leiserson, JACM 1999) shows that a **fully strict** multithreaded computation (the well-structured fork-join class — Cilk-style programs) of work W and depth D completes in $O(W/p + D)$ expected time under the randomised work-stealing scheduler — exactly the Brent bound. Work-stealing is thus a *practical scheduler achieving the Brent bound* on dynamic computations of that class; for arbitrary task graphs the guarantee is not automatic (the scope matters).

Dynamic balancing in Spark, Ray and Python (related to work-stealing, but not the same): - Spark speculative execution. When a task has been running significantly longer than the median for its stage, Spark launches a speculative copy of the task on another executor. The first copy to finish wins; the other is killed. This is not work-stealing — no queued task changes owner — but *duplicate* (backup) execution: the answer to a slow machine, not to a long queue. Enabled with `spark.speculation = true`; threshold controlled by `spark.speculation.multiplier` and `spark.speculation.quantile` (the fraction of finished tasks in the stage). Their defaults differ between Spark versions: read them from your version's configuration page (the labs pin Spark 3.5.4), or set them explicitly as Experiment 2 does. - **Ray dynamic task scheduling.** Ray places each submitted `ray.remote` task on a worker as resources free up, rather than on a fixed pre-assignment, so a long task delays only its own slot. This is the online greedy rule of §3.1. - **Python's `concurrent.futures.ProcessPoolExecutor`** feeds its worker processes from one shared call queue: each idle worker takes the next pending call. That too is greedy list scheduling with a central list, not stealing between per-worker deques.

4.4 Static partitioning vs. dynamic balancing: the fundamental tension

There is a fundamental trade-off between:

Static partitioning: decide the assignment once (at shuffle time, at partition time, at input ingestion) and commit to it. Cost: zero ongoing coordination; the shuffle is the only synchronisation point. Risk: if the distribution is adversarial or unknown, some machines get far more work than others — the straggler arises.

Dynamic balancing: monitor loads at runtime and rebalance continuously. Cost: ongoing

communication and synchronisation; every steal or speculative launch involves coordination overhead. Benefit: the system adapts to observed runtime imbalance.

In practice, distributed systems use a **hybrid**: static partitioning with a good splitter (U2: sample sort with $t = O(\log n)$ oversampling) to control skew at the shuffle, plus speculative execution or work-stealing to catch residual stragglers at runtime. The U2 and U3 units are inseparable in practice: good shuffle design (U2) reduces the frequency of stragglers; good scheduling (U3) bounds the damage when they occur.

4.5 Power-of-two-choices: the theory of random balancing (stated)

The power-of-two-choices (Azar et al. 1994; Mitzenmacher 2001) is a randomised load-balancing protocol for the online setting where each job, upon arrival, samples two machines uniformly at random and is assigned to the *less loaded* of the two.

Theorem (Azar–Broder–Karlin–Upfal 1994; stated). With $n = m$ jobs assigned to m machines by the power-of-two-choices rule, the maximum load is $\frac{\log\log m}{\log 2} + O(1)$ with high probability — versus $\Theta(\log m / \log\log m)$ for one random choice.

Heavy-load form (Berenbrink–Czumaj–Steger–Vöcking 2006; stated). For general $n \geq m$, the maximum load is $\frac{n}{m} + O(\log\log m)$ with high probability — the excess over the perfect average stays doubly-logarithmic *independent of* n.

Compare this to uniform random assignment (each job goes to one uniformly random machine), where the maximum load is $\frac{n}{m} + \Theta\left(\sqrt{\frac{n \log m}{m}}\right)$ — far worse. The power-of-two-choices reduces the maximum *excess* load from $O(\sqrt{n/m \cdot \log m})$ to $O(\log\log m)$: an exponential improvement from the extra one comparison per job. This is one of the most striking results in the theory of randomised algorithms.

The $\log\log m$ bound is not a typographic quirk: the proof (Mitzenmacher's 2001 survey is the canonical reference) uses a potential function argument tracking the number of machines with load exceeding successive thresholds; the recursion in the potential collapses doubly-exponentially.

Where it appears: web server load balancers (Nginx, HAProxy) offer the power-of-two-choices or its continuous-time analogue. (Spark's `HashPartitioner` does not: it sends a key to partition `hashCode mod numPartitions`, one fixed choice.)

Check your reading.

Q6. Placing n balls into n bins: one uniformly random choice gives a maximum load of ~ log n / log log n. Choosing the LESS loaded of TWO random bins changes the maximum load to approximately:
(A) the same order – randomness is randomness (B) log log n order – an exponential improvement (C) constant 2 (D) sqrt(n)

Q7. Continuous batching in an LLM serving engine admits and retires requests of unpredictable lengths mid-flight. In two sentences, map this onto this unit's vocabulary: what are the jobs, the machines, and the straggler?

5. The invariant — instantiated for scheduling

RAM: minimising makespan on m identical machines is NP-hard (reduces from PARTITION); no polynomial algorithm computes T_{OPT} in general. The two key lower bounds are $T_{\text{OPT}} \geq \max_j p_j$ (longest job) and $T_{\text{OPT}} \geq \frac{1}{m}\sum_j p_j$ (average load).

PRAM: greedy list-scheduling achieves $T \leq (2 - 1/p)\, T_{\text{OPT}}$ (Graham 1966); LPT achieves $(4/3 - 1/(3p))\, T_{\text{OPT}}$ (stated). The Graham proof mechanism is the Brent proof mechanism applied to variable-size jobs: both lower bounds feed into the same

two-step argument. Brent's bound is the unit-job case of Graham's precedence form $T \leq \frac{1}{p}\sum_j p_j + (1 - \frac{1}{p})L$, with L the longest chain (§3.5).

DPRAM/MPC: static partitioning controls average load but cannot prevent stragglers; dynamic balancing (work-stealing in fork-join runtimes, Spark's speculative backup copies, Ray's dynamic task placement) catches residual stragglers at runtime; power-of-two-choices achieves maximum load $n/m + O(\log \log m)$ with two random choices per job — an exponential improvement over one random choice.

6. Lab — project track

Project-track lab (no weekly receipt): run these experiments free and locally in the course lab image (`labs/image/`) — or any Spark-in-Docker environment — as deep-dive and group-project material.

Setup. One-time Docker install plus the course lab image — see `labs/README.md`; everything runs free and locally on your own laptop.

Experiment 1 — measuring makespan against the $(2 - 1/p)$ envelope (Spark).

Generate a set of $n = 200$ "tasks" with sizes drawn from a Pareto distribution (shape 1.5, scale 1): this models real-world skewed job sizes. Each task is a `mapPartitions` call that sleeps for its size in seconds (controlled by the seed).

1. Assign tasks to 4 executors *uniformly at random* (set `numPartitions = 4`, no custom partitioner). Measure the stage wall-clock makespan T_{random}.
2. Assign tasks using LPT: sort by size descending, then use a least-loaded assignment to fill 4 partitions. Measure T_{LPT}.
3. Compute the lower bounds LB1 and LB2 from the known task sizes. Compute $T_{\text{OPT}}^{\text{LB}} = \max(\text{LB1}, \text{LB2})$ (a lower bound on the true optimum).
4. Compare $T_{\text{random}}/T_{\text{OPT}}^{\text{LB}}$ and $T_{\text{LPT}}/T_{\text{OPT}}^{\text{LB}}$ against the theoretical bounds $(2 - 1/4) = 1.75$ and $(4/3 - 1/12) \approx 1.25$.

Experiment 2 — artificial straggler and speculative execution (Spark).

Create a Spark stage with 20 tasks of uniform size 1 second, plus one *straggler* task of size 10 seconds. Make the slowdown *executor-side*, not data-side: sleep only in the first attempt of the straggler partition (`if partition_id == straggler_partition and TaskContext.get().attemptNumber() == 0: sleep(10)`), so that a fresh copy runs at normal speed — the slow-machine case. (A sleep keyed on the partition alone is a data-side straggler: the speculative copy sleeps too, which is exercise E3.4's failure case.) Run on at least two worker hosts, for instance two worker containers: Spark does not place a speculative copy on a host that already runs the task, so `local[*]` launches none.

1. Run without speculative execution (`spark.speculation = false`). Measure makespan and confirm it is dominated by the straggler.
2. Enable speculative execution (`spark.speculation = true`, `spark.speculation.multiplier = 1.5`, `spark.speculation.quantile = 0.75`). Run again and measure makespan.
3. In the Spark UI, identify the speculative task: when was it launched? How much earlier did the stage finish? Compute the makespan ratio improvement $T_{\text{speculative}}/T_{\text{baseline}}$.

Question to answer in the deliverable: at what straggler size ratio does speculative execution pay off? At what ratio does it NOT pay off (because the speculation launch overhead dominates)?

Experiment 3 — static vs dynamic assignment (Ray).

Implement the same 20-task-plus-straggler workload in Ray, using `ray.remote` tasks:

1. **Static assignment:** pre-assign tasks to actors (no rebalancing); measure makespan.
2. **Dynamic (default Ray):** submit all tasks and let Ray place each on the next free worker; measure makespan.
3. Vary the straggler size from 2× to 20× the mean task size. Plot makespan vs. straggler multiple for both strategies.

Question: at what straggler multiple does dynamic scheduling recover the most wall-clock time?

Deliverable. Notebook + one table mapping each experiment's measured makespan against the three-lens cost accounting: (ops = jobs = task counts; work+depth = total task time + critical path; rounds+bytes = stage count + shuffle bytes for the Spark experiments).

7. Lean certificate (proofs via our prover)

Gap G1 — the Graham bound. In U0 §7 the Graham $(2 - 1/p)$ bound was named as the next certificate in the Lean proof chain, after `BMLSPar.lean` (Brent). This unit is where students meet that gap head-on.

```
-- SCHEMATIC target statement (names and types align to BMLSPar.lean conventions
  at cert time):
-- Inputs: n jobs, p processors, processing times pt : Fin n -> Nat
-- T_OPT = optimal makespan (spec via existence of an assignment achieving it)
-- T_L   = makespan of greedy list scheduling on any list ordering
-- NB: stated MULTIPLIED THROUGH by p (p * T_L ≤ (2p-1) * T_OPT) so the
  statement
-- lives in exact Nat arithmetic -- no Nat-division rounding subtleties.
theorem graham_list_scheduling (p : ℕ) (hp : 0 < p) (n : ℕ)
    (pt : Fin n -> Nat) (T_OPT : Nat) (hopt : isOptimalMakespan pt p T_OPT) :
    p * greedyMakespan pt p ≤ (2 * p - 1) * T_OPT := by
  sorry -- cert target G1; students see this gap named in U0 and closed here
```

The proof structure follows the written proof in §3.2 exactly:

1. Formalise the two lower bounds as lemmas `lb1` and `lb2`.
2. Let `j_star` be the last-finishing job (exists by `Finset.exists_max`).
3. Prove the start-time bound: all processors busy on jobs other than `j_star` until it starts $\Longrightarrow$ start time $\leq$ (total − `pt j_star`) / p, i.e. `p * start ≤ total - pt j_star` (§3.2; with only total / p, step 4 would reach just $2\,T_{\mathrm{OPT}}$).
4. Combine: makespan = start + `pt j_star` $\leq$ `T_OPT + (1 - 1/p) * T_OPT`.

Exercise U3.L1 (light). Read the `sorry`-stub above; supply the two lower bound lemmas (`lb1`, `lb2`) as `theorem` statements with `sorry` bodies. Check that the type signatures are consistent with the `cdiv`/`bmls_parallel_bound` helpers in `BMLSPar.lean` (note: `BMLSPar.lean` does not define `MakeSpan` as a standalone definition; the makespan is represented as `Finset.sup` over machines of their total load — align your `lb1`/`lb2` types accordingly). Run `lake build`.

Exercise U3.L2 (deep). Fill in step 3 of the proof sketch: formalise the argument "all processors are busy when `j_star` starts." (This is the key combinatorial step; it requires reasoning about the greedy scheduler's invariant — a machine is never left idle while jobs remain.) Submit the certified fragment.

Note: G1 is marked as a *gap* — it does not ship pre-proved in this draft. The ambition is to close it before the first live ScaDaMaLe run. Students who complete U3.L2 contribute to the live formalisation.

LLM lens — where this unit lands on the running workload (U0L):

Continuous batching in serving engines (vLLM-class, U0L §5) is online scheduling: requests arrive, finish at unpredictable lengths, and the engine packs the GPU like this unit packs machines. Expert load imbalance in MoE is a balls-in-bins tail (F.S2) inflating exactly the makespan this unit bounds.

8. Sources (credited; archived under `scadamale-book-sources`)

Primary:

Graham, R.L. (1966). *Bounds for certain multiprocessing anomalies.* Bell System Technical Journal, 45(9):1563–1581. — The original list-scheduling paper; the $(2 - 1/p)$ bound and the LPT bound appear here and in the companion 1969 paper. Our exposition of the proof follows the structural logic of this paper but is written anew.

Graham, R.L. (1969). *Bounds on multiprocessing timing anomalies.* SIAM Journal on Applied Mathematics, 17(2):416–429. — The LPT $(4/3 - 1/(3p))$ bound.

Supporting:

Mitzenmacher, M. (2001). *The power of two choices in randomized load balancing.* IEEE Transactions on Parallel and Distributed Systems, 12(10):1094–1104. — The definitive survey of the power-of-two-choices result; the $\log \log m$ bound and proof sketch are here.

Blumofe, R.D. and Leiserson, C.E. (1999). *Scheduling multithreaded computations by work stealing.* Journal of the ACM, 46(5):720–748. — Formal analysis of work-stealing schedulers; the $O(W/p + D)$ expected makespan theorem.

Wikipedia: *Multiprocessor scheduling*, *Job-shop scheduling*, *Work stealing*, *Power of two choices.* — Background and cross-references; not the primary exposition.

MIT OCW 6.172 *Performance Engineering of Software Systems.* — Lectures on parallel scheduling, work-stealing, and cache-efficient parallelism; freely available.

9. Exercises

Light (examined core):

E3.1 Three jobs with sizes $[3, 3, 5]$ are scheduled on $p = 2$ processors by greedy list-scheduling (in the given order). Draw the Gantt chart (time on the x-axis, processor on the y-axis). Compute the makespan. Compute LB1 and LB2. What is the ratio $T_L/T_{\text{OPT}}^{\text{LB}}$?

E3.2 For the same three jobs, compute the LPT ordering and run greedy list-scheduling on it. Is the makespan smaller? Is it optimal? (Note: $T_{\text{OPT}}^{\text{LB}} = \max(\text{LB1}, \text{LB2})$ is a lower bound on the optimum, not necessarily equal to it; the true optimum may be strictly larger than $T_{\text{OPT}}^{\text{LB}}$. Determine T_{OPT} by inspection for this small instance.)

E3.3 The Brent bound says $T_p \le W/p + D$ for a DAG of work W and depth D. Explain, in the vocabulary of §3.5, why this is the unit-size case of Graham's precedence-constrained bound $T_L \le \frac{1}{p}\sum_j p_j + (1 - \frac{1}{p})L$ when all jobs (= DAG operations) have unit size, and why the independent-jobs bound of §3.2 alone does not give it. Identify the correspondences: which quantity in Brent maps to T_{OPT}, which to L, which to $\sum_j p_j$, and what becomes of p_{j^*}?

E3.4 In Spark, `spark.speculation = true` is disabled by default. Give one scenario where enabling it would hurt performance rather than help it. (*Hint:* what happens if the "straggler" task is slow because the input data is genuinely large — not because the machine is slow — and the speculative copy also receives the same large input?)

Deep (Module-3 / optional):

E3.5 *Tightness, in full.* §3.3 sketches the instance that makes the Graham $(2 - 1/p)$ bound tight. Write the argument out in full for every $p \geq 2$: state the list ordering, trace what greedy does with it, compute T_L and T_{OPT} exactly (proving that no schedule beats your T_{OPT}), and check the ratio numerically for $p = 4$.

E3.6 *LPT gap analysis.* Prove that the LPT bound of $(4/3 - 1/(3p))\, T_{\text{OPT}}$ is also tight for $p = 3$ by constructing an instance where the LPT makespan equals $(1 + \epsilon)\, T_{\text{OPT}}$ with $\epsilon = 2/9$, i.e. the ratio $4/3 - 1/9 = 11/9$; no instance can do worse. (This requires a careful choice of job sizes; see the Graham 1969 paper for the tight example.)

E3.7 *Power-of-two-choices simulation.* Implement the power-of-two-choices protocol (each job samples 2 machines uniformly at random, goes to the less loaded) in Python for $n = 10^6$ jobs and $m \in \{10, 100, 1000\}$ machines. Compare the maximum load under: (a) uniform random assignment (one choice); (b) power-of-two-choices. Plot the *excess* maximum load (maximum load minus the mean n/m) of both vs. m on a log-log scale, and report the slope of each curve. Does the $\log \log m$ prediction match your simulation? Is the difference between the two maximum loads ever negligible — in absolute terms, or relative to the mean load n/m?

Chapter 10

04 sampling streaming

U4 — Sampling & streaming: RAM → PRAM → DPRAM

The ScaDaMaLe book, unit 4 (examined core). Our own exposition; original sources credited in §11.

Instantiating the three-lens invariant. The U0 box for sampling and streaming:

> **RAM:** exact counting and exact quantiles are baseline algorithms — they are correct but they read and store every element. The streaming model strips the machine to one pass and memory $m \ll n$: what exact answers are still computable, and what must we approximate? **PRAM:** independent reservoir samples from independent partitions can be merged; the obstacle is that a naive union is biased toward smaller shards (their elements are over-represented). The weighted-merge subtlety is the PRAM problem. **DPRAM/MPC:** sketches — Count-Min and HyperLogLog — are *linearly composable* data structures: per- partition sketch + one tree-aggregate round yields the global sketch. This is the fundamental property that makes streaming algorithms distributed. Exact distinct-count requires $\Omega(n)$ communication; HLL requires $O(m)$ bytes for m registers, independent of n.

Entry kit — what this chapter assumes; follow a link if you need the memory jog (map: Figure 1): F.S1 (indicators and linearity — this unit's whole toolkit) • F.S2 (the concentration ladder for sketch accuracy)

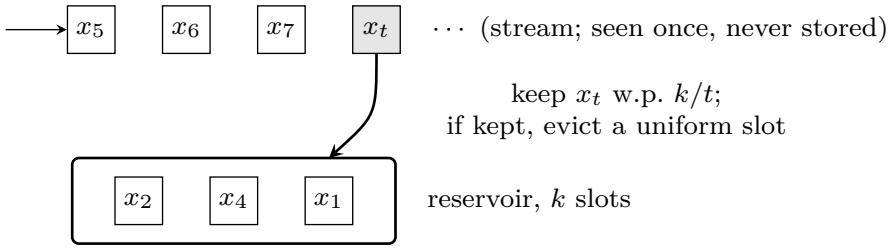
1. Intuition — memory is a resource too

In every earlier unit the cost model tracked operations (RAM), work and depth (PRAM), or rounds and bytes (DPRAM/MPC). Memory was free: we assumed the data fit and said nothing more.

This unit adds **memory** to the accounted resources, and in doing so opens an entirely new class of algorithms.

Imagine a stream of n elements arriving one at a time. You cannot store them all — your memory budget is m, and $m \ll n$. At the end of the stream you must answer a query: how many distinct elements appeared? What fraction were greater than a threshold? Which element is the most common?

The naive answer to each of these is "store everything, then compute." With memory $m \ll n$, that is impossible. The streaming model asks: *what is achievable with one pass and bounded memory?*



invariant: after step t, every one of the t items seen is in the reservoir with probability exactly k/t (cert: `ReservoirUniform`)

Figure 10.1: Reservoir sampling: the stream is a conveyor you see once; the reservoir is k slots you may rewrite. Item x_t enters with probability k/t, evicting a uniformly chosen slot — and induction keeps every item seen so far in the sample with probability exactly k/t (the machine-checked invariant). One picture, one line of probability, one certificate: the entire discipline of this unit in miniature.

The answer is striking. A *deterministic* algorithm needs $\Omega(n)$ bits even to approximate the distinct count within a small constant factor (Alon, Matias and Szegedy); but a *randomized* one approximates it to within relative error ε, with high probability, in $O(\varepsilon^{-2} \log n)$ bits, and HyperLogLog (§5) in $O(\varepsilon^{-2} \log \log n)$ bits. Uniform sampling from a stream of unknown length is achievable with exactly k elements of memory (Vitter). Heavy-hitter detection is achievable with $O(\log(1/\delta)/\varepsilon)$ cells (Count-Min).

This unit covers three algorithms: **reservoir sampling** (uniform sample), **Count-Min sketch** (heavy hitters and frequency estimation), and **HyperLogLog** (distinct-count cardinality). Together they populate the streaming practitioner's toolkit — and they all turn out to be distributed-friendly.

Figure 10.1 is the unit's emblem: one pass, bounded memory, an exact invariant.

Check your reading.

Q1. In the streaming model – one pass, memory m « n – which of the following is achievable EXACTLY, not approximately?
(A) the number of distinct elements in the stream (B) the phi-quantile of the stream (C) a uniform random sample of exactly k elements: a uniform random k-subset (every k-subset equally likely, so each item is in it with probability k/n) (D) the most frequent element (the mode)

Q2. The unit's LLM lens calls the KV cache 'a windowed summary of the stream of past tokens' and speculative decoding 'an accept-reject scheme'. In two or three sentences, map LLM decoding onto this unit's vocabulary: what is the stream, what is the bounded-memory summary, and where does sampling enter?

2. RAM — exact counting and quantiles as the baseline

2.1 Exact frequency counting

Given n elements from a universe $[u]$, build a frequency array $f[1..u]$ with $f[i]$ = number of times element i appeared. Cost: $O(n)$ time, $O(u)$ space. Querying the most frequent element (the *mode*) costs $O(u)$ extra.

For large u (e.g. $u = 2^{32}$ IPv4 addresses, 2^{128} IPv6 addresses, or 2^{64} 64-bit user-ids), $O(u)$ space is infeasible. The baseline is honest about its cost: exact counting requires space proportional to the universe, not the stream.

2.2 Exact quantiles

Sort the n elements: $O(n \log n)$ time, $O(n)$ space. The ϕ-quantile is then a direct array access. Exact quantiles require $\Omega(n)$ space: you must keep all elements (or at least their sorted order) in the worst case, since the last element of the stream can shift every quantile.

The streaming baseline. Both problems above — exact frequency and exact quantiles — are easy with $O(n)$ memory and one pass. The streaming setting forbids $O(n)$ memory. Henceforth, m denotes the working memory (in elements or cells), and we demand $m \ll n$.

3. Centerpiece 1 — Vitter reservoir sampling

3.1 The problem

Problem. A stream delivers n items, one at a time, and n is *not known in advance*. Maintain a set R of exactly k items such that, when the stream ends at item n, R is a *uniform k-subset* of the n items: each of the $\binom{n}{k}$ sets of k items is equally likely to be R (so, in particular, each item is in R with probability k/n). Use $O(k)$ memory.

This is the simplest formulation of streaming: one pass, memory exactly k, uniform sample.

3.2 Algorithm (Vitter 1985)

```
Reservoir-Sample(stream, k):
  Fill R[1..k] with the first k items of the stream.
  For i = k+1, k+2, ..., n:
    j := uniform random integer in [1, i]   // one random number per item
    if j <= k:
      R[j] := stream[i]                      // replace a random reservoir slot
  Return R
```

The reservoir R holds exactly k elements at all times; no item is stored outside R. The one-line inner loop is the entire algorithm; it needs one uniform random integer per item.

3.3 The uniformity proof (induction on n)

Claim. After processing $n \geq k$ items, the reservoir is a uniform k-subset of $\{x_1, \ldots, x_n\}$: for every set S of k of these items,

$$P(R = S) = \binom{n}{k}^{-1}.$$

In particular, each item is in the reservoir with probability exactly $\binom{n-1}{k-1}/\binom{n}{k} = k/n$.

Why the whole subset, not only each item. Equal marginals $P(x_i \in R) = k/n$ do not make R a uniform k-subset: a uniformly random block of k cyclically consecutive items also contains each item with probability k/n, yet it produces only n of the $\binom{n}{k}$ subsets (for $n = 6$, $k = 2$: 6 of the 15). The merge of §6.1 needs the whole claim.

Proof by induction on n.

Base case ($n = k$). After the fill step, $R = \{x_1, \ldots, x_k\}$, the only k-subset, with probability $1 = \binom{k}{k}^{-1}$.

Inductive step. Assume that after $n - 1$ items R is a uniform k-subset of $\{x_1, \ldots, x_{n-1}\}$. Item x_n arrives, and the algorithm draws j uniform in $[1, n]$, independently of everything before. Fix a set S of k items from $\{x_1, \ldots, x_n\}$.

- If $x_n \notin S$: $R = S$ after step n iff $R = S$ before it and x_n is rejected ($j > k$), with probability $\binom{n-1}{k}^{-1} \cdot \frac{n-k}{n}$.
- If $x_n \in S$: $R = S$ after step n iff, before it, $R = (S \setminus \{x_n\}) \cup \{y\}$ for one of the $n-k$ earlier items $y \notin S$, and j is the slot holding y (one value of j out of n), with probability $(n-k) \cdot \binom{n-1}{k}^{-1} \cdot \frac{1}{n}$.

In both cases $P(R = S) = \frac{n-k}{n}\binom{n-1}{k}^{-1} = \binom{n}{k}^{-1}$, since $\binom{n}{k} = \frac{n}{n-k}\binom{n-1}{k}$. By induction, the claim holds for all $n \geq k$. □

The marginal on its own, by the same induction. The k/n can also be followed one item at a time (this is the calculation §10 formalises). Base case: for $n = k$ every item is in R, and $k/k = 1$. Inductive step: assume that after $x_1, \ldots, x_{n-1}$ each of these $n-1$ items is in R with probability $k/(n-1)$. Now item x_n arrives.

Item x_n enters the reservoir with probability k/n (the `if j <= k` line: $j \leq k$ with prob k/n).

Items $x_1, \ldots, x_{n-1}$ remain in R after step n. Fix any x_i, $i < n$.

By the inductive hypothesis, $x_i \in R$ before step n with probability $k/(n-1)$.

Given $x_i \in R$: it is evicted only if (a) x_n is admitted ($j \leq k$, prob k/n) and (b) the slot chosen is x_i's slot (prob $1/k$, since the replacement slot is uniform in $[k]$). So:

$$P(x_i \text{ evicted at step } n \mid x_i \in R) = \frac{k}{n} \cdot \frac{1}{k} = \frac{1}{n}.$$

Therefore:

$$\begin{aligned} P(x_i \in R \text{ after step } n) &= P(x_i \in R \text{ before}) \cdot P(\text{not evicted} \mid x_i \in R) \\ &= \frac{k}{n-1} \cdot \left(1 - \frac{1}{n}\right) = \frac{k}{n-1} \cdot \frac{n-1}{n} = \frac{k}{n}. \end{aligned}$$

By induction, each item is in R with probability k/n, for all $n \geq k$. □

Memory cost. $O(k)$ — exactly the reservoir R, plus $O(1)$ counters. No element outside R is ever stored.

Time cost (RAM). $O(n)$ — one pass, $O(1)$ work per item.

3.4 Remark on the algorithm's elegance

The proof witnesses something structural: the algorithm is *memoryless* about which slot to replace. Knowing only the current stream position i and the reservoir size k, the correct replacement probability k/i falls out of the uniform draw on $[1, i]$. No knowledge of the final n is needed; the invariant self-maintains step by step.

Vitter's 1985 paper also gives optimised variants (Algorithms X, Y, Z) that skip over items without drawing a random number for each, achieving $O(k(1 + \log(n/k)))$ expected random-number calls instead of n. The core uniformity argument is the one above.

Check your reading.
Q3. In Vitter's uniformity proof, given that item x_i is already in the reservoir (size k) when item x_n arrives, the probability that x_i is evicted at step n is:
(A) k/n (B) 1/k (C) 1/n (D) 1/(n-1)

4. Centerpiece 2 — Count-Min sketch

4.1 The problem

Problem. A stream delivers n items from a universe $[u]$. Each item has an associated weight (or multiplicity) $c \geq 0$; denote by f_i the total weight of item i. After the stream, answer *point queries*: given an item a, estimate $\hat{f}_a \approx f_a$.

The weights must not be negative. What the proofs of §4.3 use is that every final count $f_j \geq 0$, and $c \geq 0$ guarantees it; with negative counts the sketch can undercount, and Markov's inequality no longer applies to the overcount. (Spark's Count-Min rejects negative counts.)

Desired guarantee. Return an estimate $\hat{f}_a$ that satisfies:

$$f_a \;\leq\; \hat{f}_a \;\leq\; f_a + \varepsilon \cdot \|f\|_1$$

with probability $\geq 1-\delta$, where $\|f\|_1 = \sum_i f_i$ is the total stream weight (= total number of items with unit weights). The sketch should use $O(\frac{1}{\varepsilon} \log \frac{1}{\delta})$ cells and support updates and queries in $O(\log \frac{1}{\delta})$ time.

Note the asymmetry: the guarantee is *overestimate-only*. The sketch cannot undercount; it may overcount by at most $\varepsilon\|f\|_1$. This is the fundamental property of the Count-Min sketch.

4.2 Construction

Fix parameters $w = \lceil e/\varepsilon \rceil$ (width, columns) and $d = \lceil \ln(1/\delta) \rceil$ (depth, rows). The sketch is a two-dimensional array $C[1..d][1..w]$ of integers, initialised to zero.

Choose d independent hash functions $h_1, \ldots, h_d : [u] \to [w]$, each mapping universe elements to columns. (In practice, pairwise-independent families suffice; we assume full independence for the clean analysis.)

Update (item a arrives with weight c):

$$C[r][h_r(a)] \mathrel{+}= c \quad \text{for each row } r \in [d].$$

Query (estimate f_a):

$$\hat{f}_a = \min_{r \in [d]} C[r][h_r(a)].$$

Worked, the smallest case. A search engine wants to know how often each query was typed, without one counter per distinct query. Three queries arrive: a, b, a. Take $d = 2$ rows of $w = 3$ counters (Figure 10.2). Row 1's hash sends a and b to the same column, so that cell ends at 3; row 2's hash sends them to different columns, which end at 2 and 1. The query for a reads its cell in each row, 3 and 2, and returns $\min(3, 2) = 2$, the true count. A cell can only *add* other items' counts to the item's own, so every row overcounts or is exact, and the minimum keeps the row with the fewest collisions. With w in the thousands and a few rows, a collision in *every* row at once is rare; §4.3 makes that precise.

4.3 The overestimate-only guarantee

Claim (overestimate). For any item a: $\hat{f}_a \geq f_a$ always.

Proof. Each counter $C[r][h_r(a)]$ accumulates the weights of all items that hash to column $h_r(a)$ in row r. Item a's own contribution f_a is always included. So $C[r][h_r(a)] \geq f_a$ for every r, and $\hat{f}_a = \min_r C[r][h_r(a)] \geq f_a$. □

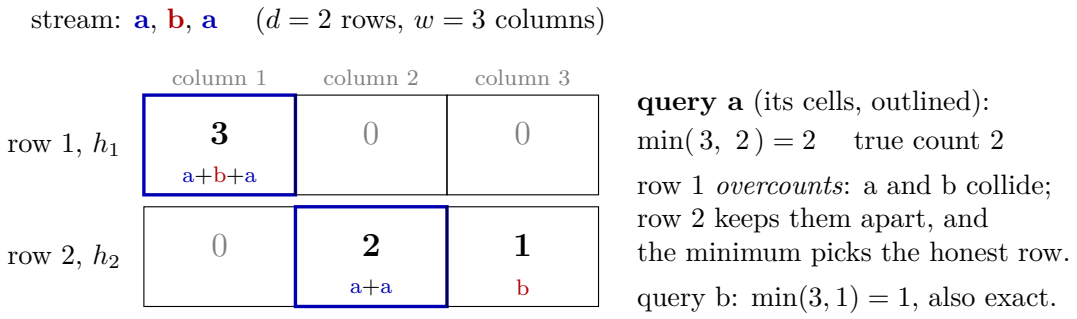


Figure 10.2: Count-Min on the smallest stream that shows its point: a, b, a with $d = 2$ rows of $w = 3$ counters. Each item adds 1 to one cell in every row (the cell its row's hash picks); a query reads the item's cell in every row and takes the minimum. A cell holds the item's own count plus whatever collided with it, so no row undercounts, and the minimum keeps the row with the fewest collisions (U4 §4.2–§4.3).

Claim (bounded overestimate). For any item a:

$$P\Big(\hat{f}_a > f_a + \varepsilon\|f\|_1\Big) \le \delta.$$

Proof. Fix a row r. Define the indicator $I_{r,j} = \mathbf{1}[h_r(j) = h_r(a),\ j \neq a]$ for each item $j \neq a$. The overcount in row r at column $h_r(a)$ is:

$$X_r = C[r][h_r(a)] - f_a = \sum_{j \neq a} f_j \cdot I_{r,j}.$$

By linearity of expectation and pairwise independence of h_r:

$$E[X_r] = \sum_{j \neq a} f_j \cdot P(h_r(j) = h_r(a)) = \sum_{j \neq a} f_j \cdot \frac{1}{w} \le \frac{\|f\|_1}{w} \le \frac{\varepsilon\|f\|_1}{e}.$$

By Markov's inequality:

$$P(X_r > \varepsilon\|f\|_1) \le \frac{E[X_r]}{\varepsilon\|f\|_1} \le \frac{1}{e}.$$

The rows are independent (different independent hash functions). The query returns the minimum over d independent rows, so:

$$P\Big(\hat{f}_a > f_a + \varepsilon\|f\|_1\Big) = P(\forall r : X_r > \varepsilon\|f\|_1) \le \left(\frac{1}{e}\right)^d \le \left(\frac{1}{e}\right)^{\ln(1/\delta)} = \delta. \qquad \square$$

Parameters. $w = \lceil e/\varepsilon \rceil$ and $d = \lceil \ln(1/\delta) \rceil$ give a sketch of size $d \times w = O(\frac{1}{\varepsilon} \ln \frac{1}{\delta})$ cells.

4.4 Heavy hitters

A *heavy hitter* with threshold ϕ is any item a with $f_a \ge \phi\|f\|_1$. Set $\varepsilon = \phi/2$ and report every item with $\hat{f}_a \ge \phi\|f\|_1$. Then (i) every true heavy hitter is reported — the sketch never undercounts, so $\hat{f}_a \ge f_a \ge \phi\|f\|_1$; and (ii) with probability $\ge 1 - \delta$ per item, nothing with $f_a < (\phi - \varepsilon)\|f\|_1 = \frac{\phi}{2}\|f\|_1$ is reported, since then $\hat{f}_a \le f_a + \varepsilon\|f\|_1 < \phi\|f\|_1$. Items in the grey band $[\frac{\phi}{2}, \phi)\|f\|_1$ may go either way — that is the price of the sketch. To enumerate candidates, a parallel counter heap over all queried items suffices. In Spark, the Scala and Java method

`df.stat.countMinSketch(col, eps, confidence, seed)` builds a Count-Min sketch and lets you query it (§7.3; PySpark differs). It never undercounts, like ours. Spark sizes it by the same Markov argument with a failure chance of $\frac{1}{2}$ per row instead of $1/e$: width $w = \lceil 2/\varepsilon \rceil$ and depth $d = \lceil \log_2(1/\delta) \rceil$ (with `confidence` $= 1 - \delta$). That argument needs each row's collision probability to be at most $1/w$ and the rows to be independent. Spark hashes an integer item by multiplying it by a random constant (one per row, all drawn from one `Random(seed)`), reducing mod $2^{31} - 1$ and then mod w, with no random additive term; that does not provably give either property, so Spark's δ is a heuristic rather than a theorem. Lab 3 compares the two sketches.

Check your reading.
Q4. The Count-Min point-query estimate f_hat_a = min over rows r of $C[r][h_r(a)]$ (update weights c ≥ 0) satisfies which guarantee?
(A) it is unbiased: $E[f_hat_a] = f_a$ (B) $f_hat_a \geq f_a$ always (update weights c ≥ 0), and $f_hat_a \leq f_a$ + eps * $||f||_1$ with probability ≥ 1 - delta: the error is one-sided (overestimate-only) (C) it can under- or over-count symmetrically, each by at most eps * $||f||_1$ (D) $f_hat_a \leq f_a$ always, since taking the min over rows can only undercount

5. HyperLogLog — approximate distinct count

5.1 The problem

Problem. Estimate the number D of distinct elements (the *cardinality*) in a stream of n elements, using $O(m)$ memory, where m is a parameter and $m \ll D$.

5.2 The leading-zeros idea

Map each element to a uniform random bit string via a hash function; copies of an element hash alike, so only the D distinct elements matter. The probability that a uniform random bit string begins with at least ℓ zeros is $2^{-\ell}$. If the maximum number of leading zeros over the hashed values is $\hat{\ell}$, then a rough estimator for D is $2^{\hat{\ell}}$. The leading-zeros idea goes back to Flajolet and Martin's (1985) probabilistic counting; the estimator above is its simplest form, which keeps only one counter (the maximum leading zeros seen).

Worked, the smallest case. A web page wants to know how many different visitors it had, without storing their identifiers. Five visits arrive: a, b, c, d, a (a came back). Hash each identifier to 6 bits (Figure 10.3): `101101`, `011010`, `110011`, `001011`, and a's again. The leading zeros are 0, 1, 0, 2, 0; the longest run is $\hat{\ell} = 2$, so $\hat{D} \approx 2^2 = 4$, the true count. The return visit hashed exactly as before and changed nothing. But one counter is fragile: had d hashed to `000001`, then $\hat{\ell} = 5$ and $\hat{D} \approx 32$. One lucky item moves the estimate by a factor of two for every extra zero.

From one counter to many: sub-streams and registers. The cure for one noisy observation is many observations, averaged. HyperLogLog gets them without hashing each item many times. It uses the first $b = \log_2 m$ bits of the hash as an address, and so splits the stream into m **sub-streams**: sub-stream j is the set of items whose hash begins with the bits of j. An item always lands in the same sub-stream, since its hash never changes, so a repeat stays where it was; and because the hash looks random, each sub-stream receives about D/m of the distinct items. A **register** is the one small counter kept for a sub-stream: a few bits of memory (6 bits hold any value up to 63) recording the longest run seen in that sub-stream. The whole sketch is the array of m registers $M[0], \ldots, M[m-1]$; nothing else is stored, neither the items nor their hashes.

Formally, for a hashed value, let ρ be the position of the leftmost 1 in its remaining bits, that is, its number of leading zeros plus one. Register $M[j]$ holds the maximum ρ over sub-stream j ($M[j] = 0$ while the sub-stream is empty). The estimate combines the m per-register estimates $2^{M[0]}, \ldots, 2^{M[m-1]}$ via their **harmonic mean**:

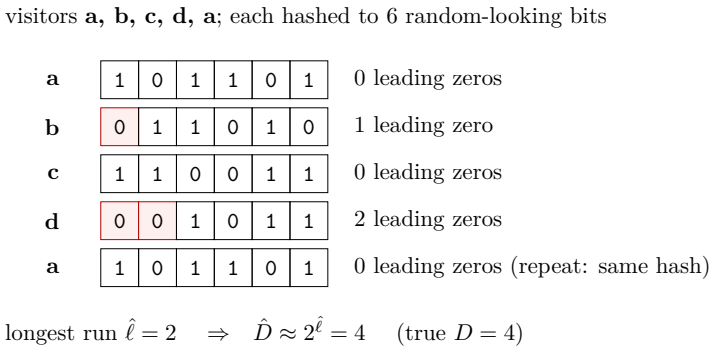


Figure 10.3: The leading-zeros idea behind HyperLogLog (U4 §5.2): hash every item, count the zeros each hash starts with, and keep only the longest run $\hat{\ell}$; about $2^{\hat{\ell}}$ distinct items were hashed. Repeats hash alike and change nothing. One such counter is a single noisy observation, which is what HyperLogLog's registers fix (Figure 10.4).

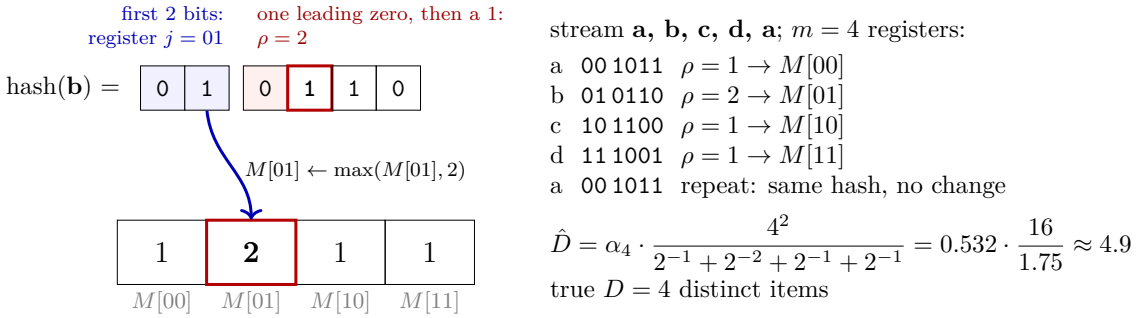


Figure 10.4: HyperLogLog on the smallest stream that shows its parts, a, b, c, d, a with $m = 4$ registers and a 6-bit hash. The first $\log_2 m = 2$ bits pick the register; ρ, the position of the first 1 in the rest, is what a long run of zeros would push up; each register keeps the largest ρ it has seen; the estimate is a harmonic mean of the $2^{M[j]}$, scaled by α_m (here $\alpha_4 = 0.532$; $\alpha_m \to 0.7213$ for large m). A repeat hashes alike and changes nothing, so only distinct items count (U4 §5.2).

$$\hat{D} = \alpha_m \cdot m^2 \cdot \left(\sum_{j=0}^{m-1} 2^{-M[j]} \right)^{-1},$$

where α_m is a small correction constant (approximately 0.7213 for large m). The constant is calibrated to ρ: with $M[j]$ defined as the maximum number of leading zeros instead, each non-empty register's $2^{-M[j]}$ doubles and the estimate drops to about half the true count.

Worked, the same five visits with $m = 4$ registers. We use a second hash (Figure 10.4), chosen so that the four visitors fall in four different registers. The hash of b is `01 0110`: its first two bits, `01`, send it to register $M[01]$; the rest, `0110`, has one leading zero, so $\rho = 2$ and $M[01]$ becomes 2. The four visitors land in the four registers, giving $M = (1, 2, 1, 1)$, and a's return visit changes nothing. The per-register guesses $2^{M[j]}$ are 2, 4, 2, 2; their harmonic mean is $4/1.75 \approx 2.29$, times $m = 4$ registers is 9.1, times $\alpha_4 = 0.532$ (from the integral that defines α_m in Flajolet et al. 2007) gives $\hat{D} \approx 4.9$ against the true 4. With $m = 4$ the typical error is $1.04/\sqrt{4} \approx 52\%$ (§5.3), so the closeness here is partly luck; the registers earn their keep at m in the thousands.

5.3 Error guarantee (stated)

Stated result (Flajolet, Fusy, Gandouet, Meunier 2007). With m registers of $O(\log \log D)$ bits each (each register holds a value of at most about $\log_2 D$, which takes $O(\log \log D)$ bits), HyperLogLog achieves relative standard error

$$\frac{\sqrt{\mathrm{Var}(\hat{D})}}{D} \approx \frac{1.04}{\sqrt{m}}$$

(asymptotically, for large D and m). This is the typical size of the relative error $|\hat{D} - D|/D$, not a bound on it; its mean is about $0.83/\sqrt{m}$.

Total memory: $O(m \log \log D)$ bits. Setting $m = 2^{14} = 16384$ gives relative standard error $\approx 0.81\%$, at a memory cost of 12 KiB.

We do not prove this here — the analysis requires careful treatment of harmonic means over Poisson processes and is beyond our scope. We cite Flajolet et al. 2007 and state the result; the exercises examine the leading-zeros intuition and the register structure.

In Spark, `approx_count_distinct(col, rsd)` and `HyperLogLog++` (an improved variant) implement this directly, with `rsd` = relative standard deviation: the accuracy you ask for, as a fraction of the true count. Spark turns it into a number of registers: the smallest power of two m with $1.106/\sqrt{m} \le$ rsd, that is $m = 2^{\lceil 2 \log_2(1.106/\mathrm{rsd}) \rceil}$ (a slightly more cautious constant than the 1.04 above). So `rsd = 0.05` gives $m = 512$ and `rsd = 0.01` gives $m = 16384$.

Check your reading.
Q5. A HyperLogLog sketch uses m = 1024 registers. Its theoretical relative standard error is approximately: (A) 1.04 / 1024, about 0.1 percent (B) 1.04 percent, independent of m (C) 1.04 / sqrt(1024) = 1.04 / 32, about 3.25 percent (D) 1.04 / log2(1024) = 10.4 percent

6. PRAM — parallel reservoir sampling and mergeable sketches

6.1 Parallel reservoir sampling and the merge problem

With P processors, each processor p holds a shard of the stream of length n_p (not necessarily equal — streams may be partitioned by time or by source). Each processor independently runs Vitter's algorithm on its shard, producing a local reservoir R_p of size k.

The merge problem. Given P reservoirs $R_1, \ldots, R_P$ of size k, each a uniform sample of its respective shard of lengths $n_1, \ldots, n_P$, produce a single reservoir R of size k that is a uniform sample of the combined stream of length $n = n_1 + \cdots + n_P$.

The naive wrong approach. Simply take the union $R_1 \cup \cdots \cup R_P$ and sub-sample k elements at random from the Pk candidates. This is *wrong*: elements from smaller shards are over-represented, since a shard of length n_p contributes k reservoir elements regardless of n_p.

The correct merge (hypergeometric split). To merge two reservoirs R_1 (uniform sample of shard 1, $|R_1| = k$, shard length n_1) and R_2 (uniform sample of shard 2, $|R_2| = k$, shard length n_2):

Draw $K_1 \sim$ Hypergeometric — the number of type-1 items in a uniform k-subset of a population of $n_1 + n_2$ items of which n_1 are type-1 — then take K_1 uniformly-chosen elements of R_1 and $k - K_1$ uniformly-chosen elements of R_2. *Why this is exact:* a uniform k-subset of the combined stream contains a Hypergeometric-distributed number of shard-1 items, and conditional on that count its shard-1 part is a uniform subset of shard 1 — which a uniform sub-sample of the uniform sample R_1 provides. (Exercise E4.7 asks you to write this out, and ALSO to show

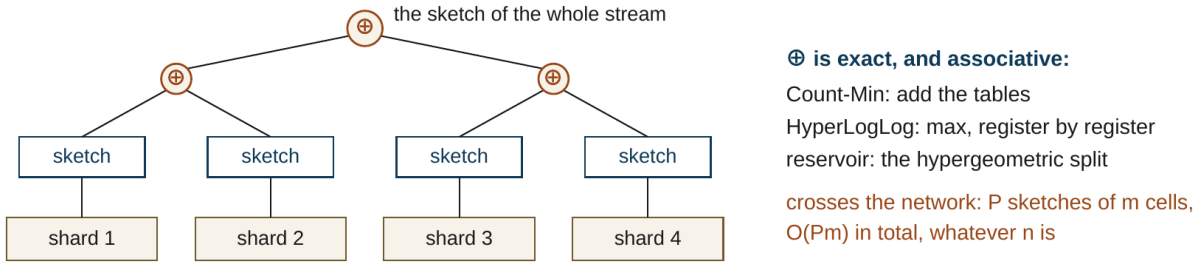


Figure 10.5: We merge the summaries, not the data.

why the tempting per-element Bernoulli-accept-and-swap scheme is *not* exactly uniform — a genuinely instructive false friend.)

For P shards, apply this merge pairwise in a tree (updating the effective shard length $n_1 + n_2$ at each internal node): $O(\log P)$ rounds, $O(kP)$ work ($P - 1$ merges of $O(k)$ each; $O(k \log P)$ along the critical path).

PRAM cost. $W = O(n + kP)$ (local sampling dominates, since $kP \leq n$), and $D = O(\max_p n_p + k \log P)$: each processor's sequential pass over its own shard, then the $O(\log P)$ levels of the merge tree at $O(k)$ each.

6.2 Mergeability: the key property

A data structure is *mergeable* if two instances computed from disjoint data can be combined in $O(1)$ or $O(\text{size})$ time to produce an instance equivalent to running on the union of the data.

- **Count-Min sketch:** merge two sketches C_1 and C_2 (same d, w and same hash functions) by coordinate-wise addition: $C[r][c] = C_1[r][c] + C_2[r][c]$. This is correct because the counter values are sums of weights — addition distributes over the stream union. $O(dw)$ **work,** $O(1)$ **depth.**
- **HyperLogLog:** merge two HLL sketches (same m and same hash function) by taking the coordinate-wise maximum: $M[j] = \max(M_1[j], M_2[j])$ for each register j. This is correct because each register holds the maximum ρ seen (§5.2) — the maximum over the union is the max of the two maxima. $O(m)$ **work,** $O(1)$ **depth.**

These merge operations are associative and (in the case of HLL with max, and CM with addition) commutative. They define *monoid* operations over the sketch data structures — the exact algebraic structure that makes distributed aggregation via tree-reduce possible.

Why exact distinct-count is harder. There is no mergeable data structure of size $o(n)$ bits that computes the exact union size — the distinct-count problem requires $\Omega(n)$ bits of communication in any protocol (a consequence of the set-disjointness communication lower bound). HLL gives an ε-approximate answer with $O(m \log \log n)$ bits total, where $m = O(1/\varepsilon^2)$.

Figure 10.5 draws the moral of this section: across machines we merge the small summaries and never move the data.

Check your reading.

Q6. Two Count-Min sketches and two HyperLogLog sketches are each built on disjoint partitions with identical parameters and hash functions. The correct coordinate-wise merge operations are:
(A) Count-Min: maximum; HyperLogLog: addition (B) Count-Min: addition; HyperLogLog: maximum
(C) both: addition (D) both: maximum

7. DPRAM / MPC — sketches as the distributed answer

7.1 The pattern: per-partition sketch + one tree-aggregate

Given a stream of n items split across M machines, the streaming paradigm maps directly to the MPC model:

1. **Map phase:** each machine runs the sketch algorithm (reservoir, CM, HLL) on its local partition. Cost: local streaming pass, $O(n/M)$ per machine, $O(m)$ memory per machine.
2. **Reduce phase (tree-aggregate):** merge the M local sketches up a binary tree of machines. Each merge step: $O(dw)$ for CM or $O(m)$ for HLL, at depth $O(\log M)$.
3. **Root:** holds the global sketch, which answers queries.

Rounds: $R = O(1)$ in practice. A binary merge tree takes $O(\log M)$ rounds, one per level. But the sketches are small, so one machine can usually merge many of them at once (all M, if M sketches fit in its memory); the tree is then flat, and two rounds suffice: one to send the sketches to the aggregator, one to broadcast the global sketch if the machines need it.

Total communication: each machine sends one sketch of $O(dw)$ or $O(m)$ cells, so total bytes $= O(M \cdot dw)$ or $O(M \cdot m)$. For $m \ll n/M$ (the usual setting), this is far less than sending the data.

7.2 Contrast: exact distinct-count requires shuffle

To count distinct elements *exactly* across M machines, equal elements must be brought together: collect all elements at one machine, hash-partition them by value (each machine then counts the distinct values of its own part, and the counts add up), or sort globally by element and count per-partition boundaries. Any such method moves $\Omega(n)$ bytes across the network — the communication lower bound from §6.2 applies.

HLL with m registers requires only $O(M \cdot m \log\log n)$ bits total — doubly logarithmic in n, so essentially independent of it. For $m = 2^{14}$ registers (12 KiB per sketch) and $M = 1000$ machines, that is 12 MB, vs $\Omega(n)$ bytes for the exact answer. This is the practical win that makes `approx_count_distinct` the default in any serious distributed analytics pipeline.

7.3 Spark APIs

- `approx_count_distinct(col, rsd=0.05)` — HLL++ under the hood; `rsd` = relative standard deviation, from which Spark picks the number of registers m (end of §5.3); the default is `rsd = 0.05`.
- **Count-Min:** in Scala and Java, `df.stat.countMinSketch(col, eps, confidence, seed)` returns a sketch you can query (`estimateCount(x)`), useful for heavy-hitter identification. PySpark 3.5 does not have this method: its `count_min_sketch(col, eps, confidence, seed)` aggregate returns the sketch serialised as bytes, which Python cannot query directly. Lab 3 therefore builds its own Count-Min in a few lines of NumPy, straight from §4.2.
- **Fixed-size sample:** `df.sample(fraction, seed)` is a Bernoulli sample (each row independently, so its size varies); `df.limit(k)` is not random (it takes rows from the first partitions); `df.orderBy(rand()).limit(k)` is a correct uniform sample of exactly k rows. Spark 3.5 moves only $P \cdot k$ rows for it over P partitions, and shuffles nothing when it is collected (the $P \cdot k$ candidates go to the driver as task results), while a full `df.orderBy(rand())` shuffles every row; Lab 3 measures the bytes and asks why.

8. The invariant (U4 instantiation)

RAM: exact counting costs $O(n)$ time and $O(u)$ or $O(n)$ space — too much for large streams; **MEMORY** joins the accounted resources. One-pass algorithms with $O(m)$ memory, $m \ll n$, are the streaming model.

PRAM: reservoir samples from disjoint partitions can be merged, but naive union is biased; the correct merge weights by shard length. Count-Min and HLL are both mergeable (addition and max, respectively) — they form monoids, and monoid operations map to tree-reduction in $O(\log P)$ depth.

DPRAM/MPC: per-partition sketch + one tree-aggregate round yields the global answer in $O(M \cdot m)$ bytes total, independent of n. Exact distinct-count requires $\Omega(n)$ bytes; HLL achieves relative standard error $1.04/\sqrt{m}$ with $O(m)$ bytes per machine. Spark exposes `approx_count_distinct` directly, and Count-Min through `df.stat.countMinSketch` in Scala and Java (§7.3).

Check your reading.
Q7. Fill the sampling-and-streaming row of the invariant box from memory: the RAM baseline and what new resource it exposes, the PRAM merge subtlety, and the DPRAM sketch pattern's rounds and bytes.

9. Lab (Lab 3 — streaming sketches)

Lab 3 — streaming sketches: the ***`lab3-streaming-sketches/`*** *folder of the course repository, and the lab chapter Lab 3 in the Laboratory exercises part. Its README +* ***`run-lab.sh`*** *follow the same receipt pattern as Labs 0-2. The experiments below are the lab's intellectual content, sized for an 8 GB laptop: one stream of $n = 10^6$ items from a Zipf($\alpha = 1.2$) distribution, split into 8 shards of unequal length.*

Experiment 1 — HLL error vs m (approx_count_distinct).

Streams of $n = 10^6$ elements with a known number of distinct values $D \in \{10^3, 10^4, 10^5, 10^6\}$. Call `approx_count_distinct` with `rsd` $\in \{0.20, 0.10, 0.05, 0.01\}$, where `rsd` is the relative standard deviation you ask for and fixes the number of registers m (end of §5.3). For each pair (D, rsd), report $|\hat{D} - D|/D$ next to the standard error $1.04/\sqrt{m}$ of §5.3. Does the measured error track the standard error?

Experiment 2 — Count-Min heavy hitters on a Zipfian stream.

Build a Count-Min sketch with $\varepsilon = 0.001$, $\delta = 0.01$ on every shard and merge the tables by addition (§6.2). Compare estimated frequencies with exact ones for the top-100 items. Report: the fraction of heavy hitters found at $\phi = 2\varepsilon$; the share of reported items that are not heavy; and the fraction of items overcounted by more than $\varepsilon\|f\|_1$, against δ (the theorem of §4.3 is about each item; the maximum overcount is only an observation). Then build Spark's own sketch from Scala, `df.stat.countMinSketch` (§7.3), on the same stream: how does Spark size it for the same ε, δ, and does it keep the same guarantees?

Experiment 3 — Reservoir sampling: exactly uniform? The merge, and the sort-based sample.

Draw a reservoir of $k = 10^3$ items on every shard and merge the reservoirs two ways: the exact hypergeometric merge of §6.1, and the naive "pool and pick k". Test each merged sample for uniformity over the stream: a χ^2 test on the counts across the ten tenths of the stream. Compare with the sort-based sample `df.orderBy(rand()).limit(k)`: is it uniform? How many bytes does Spark shuffle for it (collected to the driver, or written out), and for the full `df.orderBy(rand())`? Which samples are correct, and why does the naive merge fail?

Deliverable: the receipt, with one or two sentences on each of: a prediction, from the shard lengths and §3.3, of how many of the naive merge's items fall in the first tenth of the stream, compared with the measured count; and why `orderBy(rand()).limit(k)` shuffles only about a hundred kilobytes when written out, and nothing when collected (rows then reach the driver as task results), while a full `orderBy(rand())` shuffles the whole stream. Optional: plots, and a half-page cost-accounting in the three-lens vocabulary, noting explicitly how MEMORY appears as an accounted resource.

10. Lean certificate target

Target (new — U4): the marginal form of the reservoir uniformity claim of §3.3 — that after processing $n \geq k$ items, each item is in the reservoir with probability exactly k/n — is a statement amenable to a formal inductive proof.

```
-- SCHEMATIC target statement (names align to BMLSPar.lean conventions at cert
  time):
-- reservoir_prob n k : ℚ, the probability that a given item is in the reservoir
  after n items.
theorem reservoir_uniformity (n k : ℕ) (hk0 : 0 < k) (hk : k ≤ n) :
    reservoir_prob n k = (k : ℚ) / n := by
  induction n, hk using Nat.le_induction with
  | base => sorry   -- base case n = k: prob = 1 = (k : ℚ) / k, as k ≠ 0
  | succ n hkn ih =>
    -- step (n ≥ k > 0, so n ≠ 0): item x_{n+1} enters with prob k/(n+1);
    --       existing items survive with prob (k/n) * (1 - 1/(n+1)) = k/(n+1)
    sorry  -- cert target: fill in the step (lab U4.L1)
```

Exercise U4.L1 (Lean, light). Run the BMLSPar Lean environment. State the inductive step: write the calculation `(k/n) * (1 - 1/(n+1)) = k/(n+1)` as a Lean `calc` block over rationals, assuming `0 < n` (Lean defines $x/0 = 0$, so at $n = 0$ the identity fails for $k \neq 0$; in the induction $n \geq k > 0$). Verify it type-checks. (This is the algebraic core of the induction step, separated from the probability model.)

Exercise U4.L2 (deep). Extend the SCHEMATIC above to a fully-stated theorem in Lean's mathlib probability framework: a random variable $X_i^{(n)}$ indicating whether item i is in the reservoir after step n, show $E[X_i^{(n)}] = k/n$. The induction structure is the same; the types are richer.

LLM lens — where this unit lands on the running workload (U0L):
Decoding IS sampling from a fitted conditional distribution, one token at a time (U0L §1); temperature and top-p are reshapings of that distribution before the draw. The KV cache is a windowed summary of the stream of past tokens, and speculative decoding is an accept–reject scheme: a cheap proposal model drafts, the expensive model verifies — this unit's machinery, deployed at the heart of modern inference.

11. Sources (credited; archived under `scadamale-book-sources`)

Vitter (1985). J. S. Vitter, *Random sampling with a reservoir*, ACM Transactions on Mathematical Software 11(1):37–57, 1985. — The reservoir sampling algorithm and its analysis; the uniformity proof in §3.3 is our own inductive presentation of the argument.

Cormode & Muthukrishnan (2005). G. Cormode and S. Muthukrishnan, *An improved data stream summary: the Count-Min sketch and its applications*, Journal of Algorithms 55(1):58–75, 2005. — The Count-Min sketch construction; our §4 follows their presentation; the Markov-independence proof is our own exposition of their argument.

Flajolet, Fusy, Gandouet, Meunier (2007). P. Flajolet, É. Fusy, O. Gandouet, F. Meunier, *HyperLogLog: the analysis of a near-optimal cardinality estimation algorithm*, Discrete Mathematics and Theoretical Computer Science Proceedings, Analysis of Algorithms (AofA), 2007. — The HLL construction, harmonic-mean estimator, and $1.04/\sqrt{m}$ standard error bound; we state the result and cite this paper; the Poisson-process analysis is beyond our scope.

Wikipedia. *Reservoir sampling*, *Count-Min sketch*, *HyperLogLog*, *Streaming algorithm*. — Background and cross-references.

12. Exercises

Light (examined core):

E4.1 Vitter's algorithm processes $n = 10^6$ items with $k = 100$. (a) What is the probability that item x_{1000} is in the final reservoir? (b) What is the probability that item x_1 (the first item) survives to the end? Justify using the inductive argument of §3.3.

E4.2 Count-Min sketch with $\varepsilon = 0.01$ and $\delta = 0.05$. (a) Compute w and d. (b) What is the total number of cells? (c) For a stream with $\|f\|_1 = 10^8$, what bound on the overcount of a single queried item holds with probability at least $1 - \delta$?

E4.3 A HLL sketch with $m = 1024$ registers using a 64-bit hash. The first $\log_2 m = 10$ bits of the hash determine the register index; the remaining $64 - 10 = 54$ bits determine ρ, the position of the leftmost 1 in that register's sub-hash (§5.2). (a) What is the theoretical relative standard error? (b) How many bits does each register need to store $M[j]$, the maximum ρ over a 54-bit sub-hash (values $0, 1, \ldots, 55$: 0 for an empty register, and $\rho = 55$ when all 54 bits are zero; so 56 distinct values)? (c) What is the total memory in bytes?

E4.4 Explain why `df.limit(k)` is *not* a uniform reservoir sample in Spark. What property of the Spark execution model makes it biased? (Hint: in what order does Spark scan the partitions to find its k rows, and does that order involve any randomness? Compare §7.3.)

Deep (Module-3 / optional):

E4.5 *Count-Min lower bound.* Show that any streaming algorithm that approximates f_a to within additive error $\varepsilon\|f\|_1$ with probability $\geq 2/3$ must use $\Omega(1/\varepsilon)$ bits of memory. (*Hint:* a communication argument: encode a message of $\Theta(1/\varepsilon)$ bits into the stream, and show that a decoder who wants one bit of it, of its own choosing, can recover that one bit from a single query on the sketch.)

E4.6 *HLL leading-zeros analysis (warm-up).* Let $X_1, \ldots, X_n$ be i.i.d. uniform on $\{0,1\}^b$ for large b. Let L_i = number of leading zeros in X_i. Show that $P(L_i \geq \ell) = 2^{-\ell}$ and that $E[\max_i L_i] \approx \log_2 n$. Why does this give a (high-variance) estimator for n?

E4.7 *Reservoir merge — the exact scheme and the false friend.* Two reservoirs R_1 and R_2 of size k are uniform samples of disjoint streams of lengths n_1 and n_2. (a) Prove the hypergeometric-split merge of §6.1 is exactly uniform on the combined stream (*hint:* condition on the shard-1 count of a uniform k-subset of the union; use "a uniform sub-sample of a uniform sample is a uniform sample"). (b) Now analyse the tempting WRONG scheme — accept each element of R_2 with probability $n_2/(n_1 + n_2)$, each acceptance replacing a uniform slot of R_1: show a shard-1 item survives with probability $\left(1 - \frac{1}{k} \cdot \frac{n_2}{n_1+n_2}\right)^k \neq \frac{n_1}{n_1+n_2}$ for $k \geq 2$ (at $k = 1$ the two are equal), so for $k \geq 2$ the result is NOT exactly uniform – and it does not even converge to uniform: for $n_1 = n_2$ the survival probability $\to e^{-1/2} \approx 0.607$ as $k \to \infty$, a constant gap from the target $\frac{1}{2}$. (The false friend: the expected number of accepts, $k\, n_2/(n_1 + n_2)$, matches the hypergeometric mean, but a later accept can overwrite an earlier accepted element of R_2, so fewer shard-2 elements are

kept than accepted: even the mean composition is wrong.) Quantify the bias for $k = 4$, $n_1 = n_2$ (answer key: $(7/8)^4 = 2401/4096 \approx 0.586$ vs target 0.5).

Chapter 11

05 mapreduce spark

U5 — MapReduce & the distributed model

The ScaDaMaLe book, unit 5 (examined core). Our own exposition; original sources credited in §8.

Instantiating the three-lens invariant. The U0 box for this unit:

> **RAM:** MapReduce is not a RAM algorithm — it is the definition of the machine. The "RAM baseline" here is the *sequential* processing of all records by one worker: $T_{\text{seq}} = O(n)$ for a scan or grouping task on n records. **PRAM:** a BSP superstep IS a (map → shuffle → reduce) round; PRAM depth maps to round count via the bridge (U0 §4, deepened below). Low depth still means few rounds. **DPRAM/MPC:** RDDs make the model concrete and add fault tolerance (lineage); the accounting IS the content — every join, every aggregation, has a byte cost; this unit derives all of them.

Entry kit — what this chapter assumes; follow a link if you need the memory jog (map: Figure 1): F.C1 (the DPRAM model MapReduce instantiates) • F.E2 (containers, to run the labs) • F.E6 (driver/executor runtime anatomy)

1. Intuition — why the shape is what it is

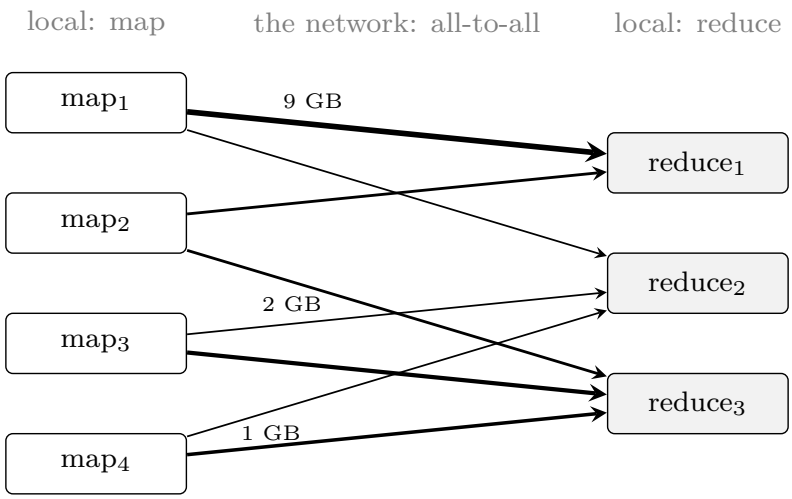
Every distributed computation faces three inescapable constraints:

1. Data does not fit on one machine (the premise of the MPC model).
2. Network bandwidth is orders of magnitude below local memory bandwidth.
3. Machines fail at rate proportional to cluster size and job duration.

The MapReduce shape — *map locally, shuffle by key, reduce locally* — is the minimal structure that satisfies all three:

- **map** and **reduce** are local; they cost no network bytes.
- **shuffle** is the single all-to-all exchange; it is unavoidable whenever two records on different machines need to meet (a join, an aggregation).
- **deterministic, side-effect-free map and reduce** mean a failed task can be re-executed on any machine that has (or can read) the same input: fault tolerance by re-execution, not replication.

The intuition for the round structure: if you need to compute something that involves records from *different* machines, you need at least one shuffle round. If two separate such computations



shuffle cost = total edge weight crossing machines;
a hot key (→ the thick edge) makes one reducer the straggler (U3)

Figure 11.1: The shuffle is a weighted bipartite graph: mappers on the left, reducers on the right, edge weight = bytes that must cross the network, priced at $\alpha + \beta M$ per message (F.E4). Everything this unit proves is visible here: total shuffle cost is the sum of edge weights, skew is one thick edge turning a reducer into the straggler of U3, and co-partitioning (U12's PageRank trick) is the art of making the graph diagonal so edges stop crossing machines at all. Expert-parallel token routing (U0L) is this same picture with tokens for keys and experts for reducers.

have no data dependency, they can share a round. The art of MapReduce algorithm design (the MRC/KSV theory below) is minimising rounds while keeping local memory sublinear.

Figure 11.1 is the cost model of this unit drawn as a graph; keep it in view throughout.

1.1 Two lineages that met

The tools of data engineering descend from two lines of work that began from opposite premises and ended in the same place. Knowing both explains why the systems you will be handed look the way they do.

The relational lineage: start from a mathematical model. Codd's relational model (1970) represents data as *relations*, sets of tuples over named attributes, and separates the **logical model** (what the relations are) from the **physical storage** (how the bytes are laid out, which indexes exist). A query states *what* result is wanted, in a declarative language (the relational algebra and calculus, later SQL); a **query optimizer** decides *how* to compute it, choosing a plan (join order, join algorithm, access paths) by estimating the cost of the alternatives. This is the bargain the whole lineage keeps: the programmer gives up control of the plan, and in return the system pays the cost of finding a good one. (The logical side is first-order logic over finite structures; F.C3 states Codd's theorem.)

Two workloads soon pulled apart on the same data. **OLTP** (online transaction processing) serves many small questions and many small writes ("what is this customer's balance?"), each touching a handful of rows but all their columns. **OLAP** (online analytical processing) serves few large questions ("revenue by region by quarter?"), each touching millions to billions of rows but only a few of many columns. The first wants row-oriented storage and a normalised schema, so that every fact is stored once and an update happens in one place. The second wants column-oriented storage and a deliberately denormalised schema. F.E5 §4 prices the difference. So analytical

data lives in a separate **data warehouse**, fed from the operational systems by pipelines; writing and running those pipelines is the job called **data engineering**.

Shared-nothing parallelism. Analytical engines scaled out early: split each table across M machines, each with its own processors and disks; run the query on every partition; merge the partial results. Teradata (founded 1979) built its database machines this way, and such systems remain common in banks, telecoms, airlines and government. The shape is the one this book prices from U1 on (partition, compute locally, combine), as in U1's reductions and U2's sample sort. Today's cloud warehouses keep the partitioned, parallel execution of queries but, as described below, no longer tie the data to the machines that compute on it.

Modelling: a warehouse is a translation, not a copy. Between the operational database and the reports sits a chain of human decisions about what the data *means*:

```
OLTP / RDBMS          →   DATA WAREHOUSE        →   DATA MART
the business as it        the business as it        the business as one
operates                  is recorded               department reports it
order_lines, customers    fact_sales, dim_customer  revenue by cost centre
normalised                conformed, historical     campaign attribution
```

The most widely used modelling style is Kimball's **star schema**. A central **fact table** holds measurements: many rows, numeric, append-only (for example `fact_sales(date_key, customer_key, product_key, store_key, quantity, amount_eur)`). Its foreign keys point to **dimension tables** of descriptive context: few rows, slowly changing (`dim_date`, `dim_customer`, `dim_product`, `dim_store`). A typical analytical query joins the fact table to a few small dimensions and aggregates, which is why the broadcast join of §4.2.2 matters so much in practice. Other styles exist: the normalised enterprise warehouse, Data Vault (hubs, links and versioned satellites, designed for auditability), and the layered bronze/silver/gold ("medallion") pattern. They share one purpose. With dozens of source systems and no modelling discipline, an organisation ends up with several "customer" tables, several definitions of "active user", and two dashboards reporting two revenue numbers. Modelling is the mechanism that makes one number mean one thing, and when a company has adopted a method, the reason is usually someone (often a regulator) asking *where did this number come from?*

The distributed lineage: start from a hardware constraint. The second line began from the fact that thousands of cheap machines fail constantly, so a system built on them must make failure routine. The Google File System (2003) stored large files as blocks replicated, by default, three times across machines. MapReduce (2004, §2) was the programming model for computing on it. Hadoop (2006) was the open-source reimplementation: HDFS for the filesystem and Hadoop MapReduce for the engine, later joined by Hive (SQL compiled to MapReduce jobs), Pig, HBase and the YARN resource manager. Its design principle was **move the computation to the data**, because the network was the scarce resource (§2, data locality).

MapReduce's weakness was the price of a *sequence* of rounds. Each job reads its input from the replicated filesystem and writes its output back there, so an iterative computation, which is to say most of machine learning, pays that round-trip on every iteration (§4.1.1 prices it). A join written by hand as one map and one reduce is also laborious. Spark kept the working set in memory across rounds (RDDs, §4.1); the RDD paper reports speed-ups of up to 20× over Hadoop on iterative machine-learning jobs. Spark's later DataFrame and SQL interfaces are declarative and are planned by the Catalyst optimizer. The distributed lineage had arrived back at the relational lineage's founding idea: declare the result, let the planner choose the plan.

The convergence: storage separated from compute. The event that brought the two lineages together was neither a database nor an engine but **object storage**: Amazon S3 (2006),

a service in which you `PUT` and `GET` byte objects over HTTP. Four properties changed the economics. Capacity is effectively unbounded, so storage no longer needs capacity planning. The price per terabyte is low, so data is rarely deleted. Durability is managed by the provider, so users no longer manage replication. Any machine can reach it, so storage is no longer attached to compute. The last property removed an assumption Teradata and Hadoop shared: on object storage, **storage persists and compute is disposable**, with clusters started for a job and released afterwards. Both lineages rebuilt themselves on this substrate, and that is why they converged. Snowflake (founded 2012; SQL-first, warehouse ancestry) and Databricks (founded 2013; Spark ancestry, Python and Scala first, files and notebooks) now compete for the same workloads, each extending towards the other's historic strength. The contest has moved below the query engine, to the **open table format** (Apache Iceberg, Delta Lake: a table stored as columnar Parquet files plus a metadata log that gives atomic commits, schema evolution and "time travel" to earlier versions) and to the **catalog** that holds every table's metadata, permissions and lineage.

Check your reading.
Q1. The map → shuffle → reduce shape is presented as the MINIMAL structure satisfying three inescapable constraints of distributed computation. Which reading of the shape is correct?
(A) map and shuffle are local; reduce is the all-to-all exchange (B) map and reduce are local (zero network bytes); shuffle is the single all-to-all exchange, unavoidable whenever records on different machines must meet; determinism of map/reduce buys fault tolerance by re-execution (C) all three phases move data across the network, but shuffle moves the least (D) the shape exists to make programs shorter, not cheaper

2. RAM — the model, stated precisely (Dean–Ghemawat formulation)

The **MapReduce model (Dean–Ghemawat, OSDI 2004)** defines a computation by two user-supplied functions:

$$\texttt{map} : (k_1, v_1) \ \rightarrow \ \text{list}((k_2, v_2))$$
$$\texttt{reduce} : (k_2, \text{list}(v_2)) \ \rightarrow \ \text{list}(y)$$

The runtime executes one **round** as follows:

1. **Map phase.** Apply `map` to each input key-value pair (k_1, v_1) independently (no communication; embarrassingly parallel). Each call emits zero or more intermediate pairs (k_2, v_2).
2. **Shuffle phase.** Group all intermediate pairs by key k_2. Every pair with the same k_2 is routed to the same reducer, regardless of which mapper emitted it. This is the **all-to-all exchange**; its cost is measured in bytes.
3. **Reduce phase.** Apply `reduce` to each group $(k_2, \text{list}(v_2))$ independently. Each call emits output values y.

Why this shape gives fault tolerance. Both `map` and `reduce` are required to be *deterministic* and *stateless* (no side effects beyond their output). If a worker executing a map task fails, the master re-assigns the exact same input split to another worker; the re-execution produces the same output. This is the key design decision (Dean–Ghemawat §3): the cost of determinism (no mutable shared state, no in-place updates) is paid once at programming time and saves the cluster from replication overhead at runtime. Map outputs are written to the map workers' local disks, partitioned by reducer, and fetched by the reducers from there. Because those files live on the mapper's own disk, a worker failure forces re-execution even of its *completed* map tasks;

completed reduce tasks, whose output is in the global fault-tolerant file system (GFS/HDFS), need not be re-run.

Data locality. The master scheduler attempts to assign each map task to a worker that already holds (or is close to) the input split — the "move computation to the data" principle. This is why the map phase often reads data at local disk bandwidth (~100 MB/s) rather than network bandwidth (~10 MB/s in 2004 clusters). The shuffle phase *must* move data (different reducers need different keys) and is therefore the network bottleneck; minimising shuffle bytes is the primary optimisation target.

Worked example: wordcount. Let the input be a set of documents, each stored as `(filename, text)`.

```
map(filename, text):
    for each word w in text.split():
        emit(w, 1)

reduce(word, counts):
    emit(word, sum(counts))
```

One map task per document: embarrassingly parallel, no cross-document communication. After the shuffle, the shuffle routes each word w to reducer $h(w) \bmod p$; every reducer holds all occurrences of the words hashed to it and sums them. Without a combiner, shuffle bytes = (number of word tokens) × (bytes per word token + 1 byte for the count), roughly the corpus size itself; a map-side combiner (§4.2.1) cuts this to one pair per distinct word per map task.

Hash partitioning. Which reducer receives key k is decided by a *partition function*. The default is $\text{part}(k) = h(k) \bmod p$ for a hash function h and p reducers (the `HashPartitioner` of both Hadoop and Spark). Equal keys hash equally, so all pairs with the same key meet at one reducer; that is the only property correctness needs. A good h also spreads distinct keys evenly over the reducers, which is what balance needs. (A *range* partitioner, U2 §4.4, instead assigns contiguous key ranges, and is what sorting needs.)

Worked example: wordcount, every key/value pair. Three input splits of three words each, and $p = 2$ reducers. Which reducer gets which word depends on h; the assignment below is one possibility.

```
split 1 "the cat sat" -map-> (the,1) (cat,1) (sat,1)
split 2 "the dog sat" -map-> (the,1) (dog,1) (sat,1)
split 3 "the cat ran" -map-> (the,1) (cat,1) (ran,1)
        ----------- shuffle: group by key, part(w) = h(w) mod 2 -----------
reducer 1:  the - [1,1,1] -reduce-> (the,3)
            cat - [1,1]   -reduce-> (cat,2)
reducer 2:  sat - [1,1]   -reduce-> (sat,2)
            dog - [1]     -reduce-> (dog,1)
            ran - [1]     -reduce-> (ran,1)
```

Map and reduce move no bytes across the network; only the shuffle does. It carries one pair per token: 9 pairs. With 3-byte words and a 1-byte count, the formula above gives $9 \times (3 + 1) = 36$ bytes. A map-side combiner (§4.2.1) saves nothing on this input, because no word repeats *within* a split: each map task has three distinct words and still sends three pairs. If split 1 were "the cat the", its combiner would send `(the,2) (cat,1)`, two pairs instead of three. The combiner pays off exactly when keys repeat within a split.

Check your reading.
Q2. Dean-Ghemawat REQUIRE map and reduce to be deterministic and side-effect free. The design payoff is:
(A) a failed task is simply re-executed on any machine with the same input split and produces identical output – fault tolerance by re-execution, with no replication of intermediate state (B) the shuffle phase becomes unnecessary (C) map tasks run faster because pure functions compile better (D) the master can skip scheduling entirely

3. PRAM — the BSP bridge (U0 §4, deepened)

3.1 Recap: BSP and the depth→rounds mapping

Recall from U0 §4: a BSP superstep = local compute, then communication (the all-to-all exchange), then a barrier. A PRAM algorithm of depth D simulates in $O(D)$ BSP supersteps ($O(1)$ supersteps per level of the PRAM computation DAG) under three hypotheses (U0 E0.6): the PRAM is EREW, or CREW with concurrent reads handled as in KSV's simulation (§4.3); its total memory n fits in the cluster, $n \leq M \cdot s$; and each machine sends and receives $O(s)$ bytes per superstep. Therefore:

> **Bridge (U0, deepened here):** under these hypotheses, a PRAM algorithm of depth D and work W runs in $R = O(D)$ MapReduce rounds on the MPC model, with local work $O(W/M)$ per machine per round.

The new content of this unit is the *converse discipline*: not every D-depth PRAM algorithm that runs in $O(D)$ rounds is a *good* MPC algorithm. An $O(\log n)$-depth PRAM algorithm that shuffles the entire dataset (all n records) every level uses $O(n \log n)$ total network bytes — a factor $O(\log n)$ above the $O(n)$-byte lower bound for many problems. **U5 dwells on exactly this divergence.**

3.2 Wordcount as a reduce-tree (PRAM lens)

Wordcount is a reduction: each token `(word, 1)` is a leaf; the goal is the sum for each word. On the PRAM, this is a per-key parallel reduction:

- Each word's occurrences form an independent group; within a group, apply a balanced tree of additions.
- $W = O(n)$ (each token counted once), $D = O(\log(\text{max group size})) \leq O(\log n)$.

On the BSP/MPC: the *shuffle* groups tokens by word (one round), then each reducer runs the sequential fold (or `treeReduce`) on its group locally. The BSP bridge says this costs $R = O(\log n)$ rounds IF we execute the tree one level per round. But the na"ive implementation fires all $O(\log n)$ tree levels as separate rounds — i.e. $O(\log n)$ shuffles — when in fact one shuffle (group by word) suffices and the tree runs locally after, *provided every word's group fits on one machine* (at most s tokens, the local memory of U0 §4). A Zipf-distributed corpus breaks this hypothesis: its most frequent word is a constant fraction of all n tokens (about 8% for a vocabulary of 10^5 words with Zipf exponent 1, since $1/H_{10^5} \approx 1/12$), so that word's group outgrows any sublinear s. With a combiner (wordcount's `+` is associative and commutative, §4.2.1), each key arrives at its reducer as at most M partial aggregates, one per mapper, so a reducer holding the key set K receives at most $M \cdot |K|$ records: one round therefore suffices when $M \cdot |K| \leq s$, even if a raw group exceeds s; for a single heavy key that is $M \leq s$.

The lesson. The bridge upper-bounds rounds; the algorithm can do *better* by batching many PRAM levels into one round (doing more local work per round, fewer network touches). The right question for MPC is not "how many PRAM levels?" but "how many times must data cross the network?"

Check your reading.
Q3. The bridge says a PRAM algorithm of depth D runs in R = O(D) MapReduce rounds (under the simulation's hypotheses). Why can executing an O(log n)-depth PRAM algorithm level-by-level still be a BAD MPC algorithm?
(A) because PRAM depth and MPC rounds are unrelated quantities (B) because the bridge only holds for sorting-type problems (C) because O(log n) rounds always exceeds the machine count (D) because firing every PRAM level as its own round shuffles the data O(log n) times – O(n log n) network bytes – when batching the tree into local work after ONE group-by-key shuffle uses O(n) bytes (when every key's group fits on one machine, or, with a combiner, when M*|K| ≤ s)

4. DPRAM / MPC — RDDs, partitioning, shuffle cost, and joins

4.1 Resilient Distributed Datasets (RDDs)

Zaharia et al. (NSDI 2012) introduce the **Resilient Distributed Dataset (RDD)**: a read-only, partitioned collection of records, defined by a **lineage** — a DAG of transformations applied to stable input data (files on HDFS) or other RDDs.

Formal definition (adapted from Zaharia §4). An RDD R is defined by: - a set of **partitions** $\{P_1, \ldots, P_k\}$, - for each partition, a set of **dependencies** on partitions of parent RDDs, - a **compute function** that, given its dependency partitions, computes the partition's records, - optionally, **partitioning metadata** (a `Partitioner`) and **preferred locations**.

Lineage-based fault tolerance. No RDD's records are stored by default — only the lineage DAG (the one exception, shuffle output, is described with persist below). If partition P_i is lost (executor failure), Spark re-executes the compute function for P_i using the dependency partitions (which are either available in memory, on disk, or can themselves be re-computed). This is the direct descendent of MapReduce's re-execution model, generalised to arbitrary DAGs rather than the fixed map→shuffle→reduce shape.

Narrow vs wide dependencies — the structural definition of a stage boundary.

Dependency type	Definition	Failure recovery	Stage boundary?
Narrow	Each partition of the *parent* is used by at most one partition of the child (Zaharia §4: `map`, `filter`, `union`, `coalesce`, a `join` of co-partitioned inputs)	Recompute only the lost partitions, from their parent partitions, in parallel on other nodes	No
Wide	A partition of the parent may be used by *multiple* child partitions (a shuffle: `groupByKey`; `reduceByKey` and `join` on inputs not already partitioned by the key)	Spark resubmits tasks only for the lost map outputs (and, recursively, any lost ancestor outputs); in the worst case a single failure forces a complete re-execution (Zaharia §4), which is why long lineages are checkpointed	Yes

A **stage** is a maximal sequence of narrow transformations. A **stage boundary** = a wide dependency = a shuffle. The Spark scheduler cuts the lineage DAG at every wide dependency edge; within a stage, all operations are pipelined (no intermediate materialisation).

Why immutability makes re-execution safe. If an RDD were mutable, a re-executed task might produce different output (because prior tasks modified shared state). Immutability + determinism = re-execution produces the identical partition. This is the same argument as MapReduce determinism, lifted to arbitrary lineage DAGs.

Persist / cache semantics. `rdd.persist()` (or `cache()`) is *lazy*: it marks `rdd` for storage, and its partitions are materialised in executor memory (default) or on disk by the first action that computes them. Persist *keeps* the lineage: a cached partition lost with its executor is recomputed from the lineage (or read from a replica, under a replicated storage level such as `MEMORY_ONLY_2`). Only `checkpoint()` cuts the lineage: it writes the RDD to reliable storage (HDFS) when the next action runs, and recovery then reads that copy. Shuffle outputs are a separate matter: Spark writes each shuffle's map-side output to the local disk of the node that ran the map task ("we currently materialize intermediate records on the nodes holding parent partitions", Zaharia §5.1), as MapReduce does (§2). They let a later stage or job restart from the shuffle, but they are lost with that node. Persist is a performance hint: use it when the same RDD is used in multiple downstream computations (without persist, each action recomputes the lineage back to the source or to the last shuffle output still on disk). Checkpoint is the fault-tolerance cut: use it when the lineage is long and re-computation after a failure is expensive (e.g. an iterative chain of joins; Zaharia §5.4).

Cost model. Without persist: the cost of a computation is the cost of computing every RDD from scratch (or from the last shuffle output still on disk) each time it is needed. With persist: the cost of a computation is the cost of the first execution (to materialise) plus the cost of subsequent reads from memory. The *right* place to persist is the highest-cost RDD that is re-used, where "highest-cost" means the shuffle writes and reads leading up to it.

4.1.1 Worked cost model: k iterations on MapReduce and on Spark

Setup. A working set of B bytes on M machines; k iterations, each one map → shuffle → reduce round; the stable store (HDFS) keeps 3 replicas of everything it writes; each iteration's output is the same size as its input; and the shuffle moves S bytes per iteration in either engine.

MapReduce. Each iteration is a separate job. It reads its B bytes of input from HDFS and writes its B bytes of output back, stored three times, so $3B$ bytes land on disk. That is $4B$ bytes of stable-storage traffic per iteration and $4kB$ in total:

```
iteration 1:  read HDFS --> map --> reduce --> WRITE HDFS
iteration 2:  READ HDFS --> map --> reduce --> WRITE HDFS
iteration 3:  READ HDFS --> map --> reduce --> WRITE HDFS
              ^----------- disk, 3x replicated ----------+
```

Spark, working set persisted. The input is read once (B), the partitions stay in executor memory across iterations (`cache()`), and only the final result is written ($3B$): $4B$ in total.

lens (U0 §4)	MapReduce	Spark, working set `cache()`d
rounds R	k	k
shuffle bytes	kS	kS
stable-storage bytes	$k(B+3B) = 4kB$	$B+3B = 4B$
local memory per machine	one input split at a time	B/M, held across rounds

The ratio of stable-storage traffic is $4kB/4B = k$. At $k = 1$ (one-pass extract-transform-load) the two engines are equal, which is why MapReduce was adequate for batch ETL and painful for iterative learning.

What Spark does not remove. It keeps the same k rounds and the same kS shuffle bytes. Both engines also write each shuffle's map-side output to *local*, unreplicated disk before the reducers fetch it. What Spark removes is the round-trip through *replicated stable storage* on every round, and only while B/M (plus working space) fits in executor memory. If it does not fit,

partitions spill to disk (`MEMORY_AND_DISK`) or are recomputed from lineage, and the advantage shrinks.

With numbers. Take $B = 100$ GB, $k = 20$, $M = 10$, and 100 MB/s of disk bandwidth per machine (the §2 figure), so about 1 GB/s across the cluster.

	stable-storage bytes	disk time at 1 GB/s	memory per machine
MapReduce	$4 \cdot 20 \cdot 100$ GB = 8 TB	8000 s ≈ 2.2 h	small
Spark	$4 \cdot 100$ GB = 400 GB	400 s ≈ 6.7 min	100/10 = 10 GB

The disk-traffic ratio is $k = 20$. The *end-to-end* ratio is smaller, because shuffle and compute are shared costs. If they take τ seconds per iteration in both engines, the ratio is $(8000 + 20\tau)/(400 + 20\tau)$; at $\tau = 100$ s that is $10000/2400 \approx 4.2$. Faster disks shrink both disk columns by the same factor and leave the ratio k unchanged. The price of MapReduce's disk I/O grows with k; the price of Spark's does not.

4.2 Communication cost of the workhorses

4.2.1 `reduceByKey` vs `groupByKey` (map-side combine)

Both aggregate values by key. The difference is whether partial aggregation happens before the shuffle:

`groupByKey` collects all values for each key from all partitions and ships them to the reducer before applying any function. For a dataset of n records each carrying a value of size v bytes:

$$\text{Shuffle bytes (groupByKey)} = n \cdot v + n \cdot |k|$$

where $|k|$ is the average key size. *Every* value crosses the network.

`reduceByKey(f)` applies `f` (an associative, commutative function such as `+` or `max`) to partial aggregates *within each mapper* before the shuffle (the **map-side combine**, also called a *combiner* in Hadoop parlance). Let $\rho = V/n$ be the reduction ratio — the ratio of the number V of distinct keys to total records (in wordcount, $\rho = |\text{vocab}|/|\text{tokens}|$). After the map-side combine, each mapper emits one `(key, partial_aggregate)` per distinct key it holds. If mapper j (of M) holds V_j distinct keys:

$$\text{Shuffle bytes (reduceByKey)} = \sum_{j=1}^{M} V_j\,(v_{\text{agg}} + |k|), \qquad V \;\le\; \sum_{j=1}^{M} V_j \;\le\; \min(n,\, MV),$$

where v_{agg} is the size of the aggregated value (often much smaller than v — e.g. one integer instead of a list). The lower end is the best case, each key on exactly one mapper: $n\rho(v_{\text{agg}} + |k|)$ bytes. The upper end is M times larger: if every key occurs on all M mappers, the shuffle moves $Mn\rho(v_{\text{agg}} + |k|)$ bytes. Because V_j is at most mapper j's record count, the shuffle never exceeds $n(v_{\text{agg}} + |k|)$, which is at most `groupByKey`'s bytes when $v_{\text{agg}} \le v$. For workloads where $\rho \ll 1$ (high repetition, e.g. word frequencies), `reduceByKey` can reduce shuffle bytes by up to a factor of $1/\rho$ (with $v_{\text{agg}} = v$) — orders of magnitude for natural language; when every key is on every mapper the factor falls to $\max(1, 1/(M\rho))$.

When is the map-side combine safe? Only when `f` is *associative and commutative*: the partial aggregates can be combined in any order and yield the same result. For `sum`, `max`, `min`, `count`: safe. For `mean`: NOT safe (average of averages is

not the global average) — carry (sum, count) pairs, which combine safely, and divide at the end: `rdd.mapValues(lambda x: (x, 1)).reduceByKey(lambda a, b: (a[0] + b[0], a[1] + b[1])).mapValues(lambda p: p[0] / p[1])`. (RDDs have no `sumByKey`, and `countByKey` is an action that returns a map to the driver.) For set operations: depends on the operator.

Rule (always prefer `reduceByKey` over `groupByKey` for associative, commutative aggregates).

4.2.2 The three joins and their byte costs

A join of two RDDs R (size $|R|$ records, average record size r bytes) and S (size $|S|$, average record size s bytes) on a common key, with output size $|R \bowtie S|$ records of average size $r + s$ bytes.

Shuffle join (sort-merge join / hash-join via shuffle).

Both sides are shuffled by the join key to the same set of M partitions; within each partition a local hash-join or merge-join produces the output.

$$\text{Shuffle bytes} = |R| \cdot r + |S| \cdot s$$

Both full datasets cross the network. Number of shuffle rounds: **1** (one wide dependency for each of the two sides: two map stages, one per side, and one shuffle round). Local work per partition: $O((|R| + |S| + |R \bowtie S|)/M)$ for a hash join (build a hash table on the smaller side, probe with the larger side, emit the matches) when the keys spread evenly. Skew breaks the $1/M$: the partition that owns a heavy key k receives all $|R_k| + |S_k|$ of its records and emits all $|R_k| \cdot |S_k|$ of its output pairs.

When to use: the default; both datasets are large; no better structure available.

Broadcast join (map-side join, replicated join).

If $|S| \cdot s \ll$ executor memory ($= s_{\text{mem}}$): broadcast S to every executor (one read from the driver; broadcast variable), then each partition of R joins with the in-memory copy of S locally, with no shuffle of R.

$$\text{Broadcast bytes} = |S| \cdot s \cdot M, \quad \text{shuffle bytes} = 0$$

The M executors each receive a copy of S. R moves zero bytes in the shuffle. **0 additional shuffle stages for** R; the broadcast itself is $O(M)$ messages of size $|S| \cdot s$.

Threshold (practical rule): `spark.sql.autoBroadcastJoinThreshold` (default 10 MB in Spark SQL).

When broadcast beats shuffle:

$$|S| \cdot s \cdot M < |R| \cdot r + |S| \cdot s \quad \Leftrightarrow \quad |S| \cdot s < \frac{|R| \cdot r}{M - 1} \approx \frac{|R| \cdot r}{M}$$

i.e. the smaller side fits in the memory that would otherwise be used to receive and hash it across the network — roughly, broadcast wins when $|S|$ is less than $1/M$ of the total join volume.

Worked example: one join, two prices. Join a fact table with a dimension table, `fact_sales` $\bowtie$ `dim_store` (§1.1): R has 10^9 rows of 100 bytes (100 GB), S has 10^5 rows of 100 bytes (10 MB), and $M = 100$.

	bytes over the network	shuffle rounds
shuffle join	$100 + 0.01 = 100.01$ GB	1
broadcast join	10 MB $\times$ 100 = 1 GB	0
broadcast, if S were 2 GB	$2 \times 100 = 200$ GB (the shuffle join would move 102 GB)	0

Here the crossover is $|S| \cdot s < 100 \text{ GB}/99 \approx 1.01$ GB. The 10 MB dimension is $100.01/1 \approx 100\times$ cheaper to broadcast. A 2 GB dimension is cheaper to shuffle, and may not fit in an executor at all. Two refinements leave these conclusions unchanged. First, in a shuffle a fraction $1/M$ of each side is already on its destination machine, so the shuffle strictly moves $(1 - 1/M)(|R|r + |S|s)$ bytes, a 1% correction at $M = 100$. Second, a broadcast is not free of rounds: S must first be collected at the driver and then sent out, one or two communication steps. What the broadcast join saves is the *shuffle of* R, not every round. The rule: broadcast a small dimension; shuffle both sides when both are large; the inequality tells you which case you are in.

Co-partitioned join (narrow dependency).

If R and S were partitioned with the *same* partitioner (equal `Partitioner` objects: the same partitioning function and the same number of partitions, e.g. both `partitionBy(p)` with the default hash partitioner) and both were persisted after partitioning, then partition i of R and partition i of S hold the same keys, and `R.join(S)` has a narrow dependency on each parent: no shuffle. Without the persist, every recomputation of a partitioned RDD fetches its records through the shuffle again. Two separate `sortByKey` calls do *not* co-partition: each samples its own data to choose its range bounds, so the two range partitioners differ.

Shuffle bytes = 0

0 shuffle rounds (no wide dependency). Local work: a per-partition hash join (the RDD `join` is hash-based: it groups both sides' values per key), $O((|R| + |S| + |R \bowtie S|)/M)$ when the keys spread evenly. (Spark SQL differs: its default for joining two large tables is a sort-merge join, `SortMergeJoin`, which shuffles both sides by the key unless they are already partitioned compatibly.)

When to use: when the tables are large, repeatedly joined, and can be persisted in co-partitioned form. A one-time partitioning cost ($O(n)$ shuffle bytes) amortised over k joins gives total shuffle cost $O(n/k)$ per join — the dominant win for iterative algorithms (PageRank in U12) that join the same two RDDs repeatedly.

Summary table:

Join type	Network bytes	Rounds	Condition
Shuffle join	$\|R\| \cdot r + \|S\| \cdot s$ (shuffle)	1 shuffle	Default
Broadcast join	$\|S\| \cdot s \cdot M$ (broadcast; shuffle = 0)	no shuffle; 1 broadcast step (1–2 communication steps)	$\|S\| \cdot s < s_{\text{mem}}$
Co-partitioned join (narrow)	0 per join (after a one-time partitioning shuffle)	0 (amortised)	Same partitioner on both sides, both persisted

Figure 11.2 prices one join two ways from the table above; what we pay for is the shuffle.

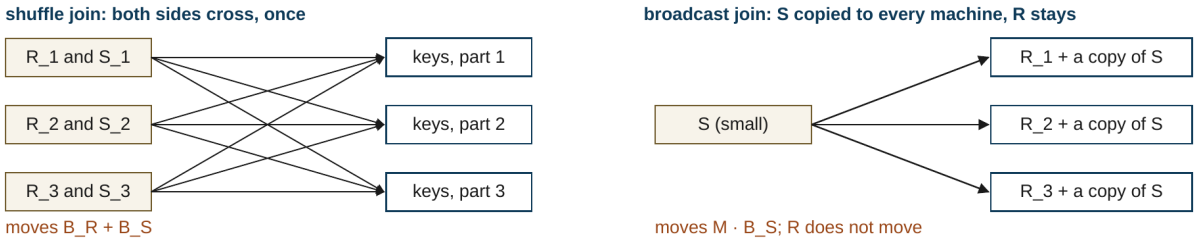


Figure 11.2: One join at two prices: the shuffle is the cost.

4.3 The MRC / KSV theory (deepening U0 §4)

The **MapReduce Class (MRC)** framework of Karloff, Suri, and Vassilvitskii (SODA 2010) gives a formal complexity-theoretic foundation for the empirical MapReduce model. We recap the U0 definition and state the new content this unit adds.

Recall (U0). In MPC: M machines, each with local memory s sublinear in n. **MRC (KSV).** Fix $\epsilon > 0$. An MRC algorithm runs on $O(n^{1-\epsilon})$ machines, each with $O(n^{1-\epsilon})$ local memory (so $O(n^{2-2\epsilon})$ total memory), with polynomial-time map and reduce functions. A problem is in $\mathbf{MRC}_i$ (KSV write MRC^i) if such an algorithm solves it in $O(\log^i n)$ rounds; MRC itself allows polylogarithmically many rounds.

New: replication rate and the communication lower bound (Afrati et al. joins).

For a binary join $R \bowtie S$ with $|R| = |S| = n$ and output size $|R \bowtie S|$:

Define the **replication rate** r of a round as the average number of reducers each input record is sent to: the total number of records received by all reducers, divided by the number $|I|$ of input records ($|I| = 2n$ here); total communication in the round is $r \cdot |I|$ records. (We write r, as Afrati et al. do; it is neither §4.2.1's reduction ratio ρ nor §4.2.2's record size.) The key distinction (upper bounds: Afrati–Ullman's *Shares*, EDBT 2010; lower bounds: Afrati–Das Sarma–Salihoglu–Ullman, PVLDB 2013):

- **Well-spread equi-join:** when keys are well distributed, hash-routing each record to the reducer owning its key achieves $r = 1$ — every record moves exactly once, with no reducer cap needed. This is the shuffle join of §4.2.2, and it is optimal.
- **Output-dense joins** — cross-products, similarity/all-pairs joins, heavy-hitter keys, and multiway joins — are where replication becomes unavoidable once each reducer's input is capped:

  Theorem (cross-product lower bound; the Afrati–Das Sarma–Salihoglu–Ullman recipe). Let a 1-round MapReduce algorithm compute the full cross-product of R and S ($|R| = |S| = n$), with each reducer receiving at most q input records. Then its replication rate is $r \geq 2n/q$, i.e. at least $4n^2/q$ records are communicated. With M reducers the cap must satisfy $q \geq 2n/\sqrt{M}$, and at that balanced cap $q = 2n/\sqrt{M}$ the bound reads $r \geq \sqrt{M}$: total communication $\geq 2n\sqrt{M}$ records.

The cap is essential: without it, a single reducer that receives all $2n$ records computes the cross-product with $r = 1$, however many reducers are available.

Proof (covering argument). Reducer i receives $q_i \leq q$ records, x_i of R and y_i of S, so it can emit at most $x_i y_i \leq q_i^2/4$ output pairs. Together the reducers must cover all n^2 pairs: $\sum_i q_i^2/4 \geq n^2$. Since $q_i \leq q$, $\sum_i q_i \cdot q/4 \geq \sum_i q_i^2/4 \geq n^2$, so $\sum_i q_i \geq 4n^2/q$ and $r = \sum_i q_i/(2n) \geq 2n/q$. With M reducers, $Mq^2/4 \geq \sum_i q_i^2/4 \geq n^2$ gives $q \geq 2n/\sqrt{M}$. □

The bound is tight: split R and S into $\sqrt{M}$ groups of $n/\sqrt{M}$ records each and send group a of R and group b of S to reducer (a, b); every reducer receives $q = 2n/\sqrt{M}$ records and every record goes to $\sqrt{M}$ reducers. (The bound $2n/q$ is an average over all $2n$ records; an unbalanced grid can replicate one side less, at the cost of replicating the other more.)

Implications: - For the 1-round cyclic (triangle) join $R(A, B) \bowtie S(B, C) \bowtie T(C, A)$ with $|R| = |S| = |T| = n$ and a reducer cap, the same covering argument, with the AGM bound (a reducer holding x, y, z records of R, S, T emits at most $\sqrt{xyz}$ triples) in place of $x_i y_i$, gives $r \geq M^{1/3}$ at the balanced cap $q = 3n/M^{2/3}$ on an instance with $n^{3/2}$ output triples (exercise E5.7); the Shares plan, which hashes each of A, B, C into $M^{1/3}$ buckets, attains it. For the 3-way cross product the argument gives $M^{2/3}$ (at its balanced cap $q = 3n/M^{1/3}$); for an equi-join on one common attribute, hashing on it gives $r = 1$. The alternative to the 1-round triangle join is 2 sequential binary joins (2 rounds, $r = 1$ each with no reducer cap, paying to shuffle the intermediate $R \bowtie S$) — a genuine rounds-vs-replication tradeoff. - Skewed keys (heavy hitters) push an equi-join locally into the output-dense regime — the U2/U4 heavy-hitter machinery is what detects it. - **Communication-avoiding joins** (the analogue of 2.5D matmul, U6) navigate exactly this replication/rounds/memory tradeoff.

What is expressible in $O(1)$ rounds with sublinear memory?

KSV show that a CREW PRAM algorithm that runs in t steps using $O(n^{2-2\epsilon})$ total memory and $O(n^{2-2\epsilon})$ processors can be simulated in $O(t)$ MapReduce rounds within the MRC bounds (Fish et al. 2015 restate it: "MapReduce algorithms can simulate CREW PRAMs which use subquadratic total memory and processors"). So an $O(\log n)$-time CREW PRAM algorithm with subquadratic memory and processors gives an MRC_1 algorithm. The subquadratic hypotheses matter; the result does not place all of NC^1 in MRC_1. (NC^1 is $O(\log n)$-depth, polynomial-size circuits of *bounded* fan-in; $O(\log n)$ depth with polynomial fan-in is AC^1.) What matters for this course follows from the definition itself:

> Any algorithm expressible as a constant number of map→shuffle→reduce steps on $O(n^{1-\epsilon})$ machines with $O(n^{1-\epsilon})$ local memory and polynomial-time map and reduce is in MRC_0 ($= O(1)$ rounds). The workhorses of this course (wordcount, sort, pageRank-step, ALS-step, and equi-joins whose output fits the $O(n^{2-2\epsilon})$ total memory) are in MRC_0 or MRC_1, provided heavy keys are split or combined so that no reducer exceeds its $O(n^{1-\epsilon})$ memory.

However, not every join qualifies: a cross-product, or an equi-join in which one key carries a constant fraction of both sides, outputs $\Theta(n^2)$ records, more than the $O(n^{2-2\epsilon})$ total memory; and a single heavy key, left unsplit, breaks the per-reducer bound (§3.2 shows the same for a Zipf-distributed wordcount).

Not everything collapses to $O(1)$ rounds: for graph **connectivity** with strictly sublinear local memory, the widely believed **one-cycle-vs-two-cycles conjecture** asserts $\Omega(\log n)$ rounds are necessary (distinguishing one n-cycle from two $n/2$-cycles); it remains unproven, and conditional hardness results for many graph problems rest on it. This is an active research frontier.

We work the replication rate once by hand, on a six-by-six input and nine reducers, in Figure 11.3.

4.4 Around the engine: what a practitioner is handed

Arrays and data frames. *numpy* stores an array as one contiguous buffer of a single machine type (for example 10^6 contiguous 8-byte floats) and provides *vectorised* operations. A Python `for` loop over the array runs the interpreter once per element, with a type check and an object allocation each time. `a + b` on numpy arrays dispatches *once* into a compiled loop over the

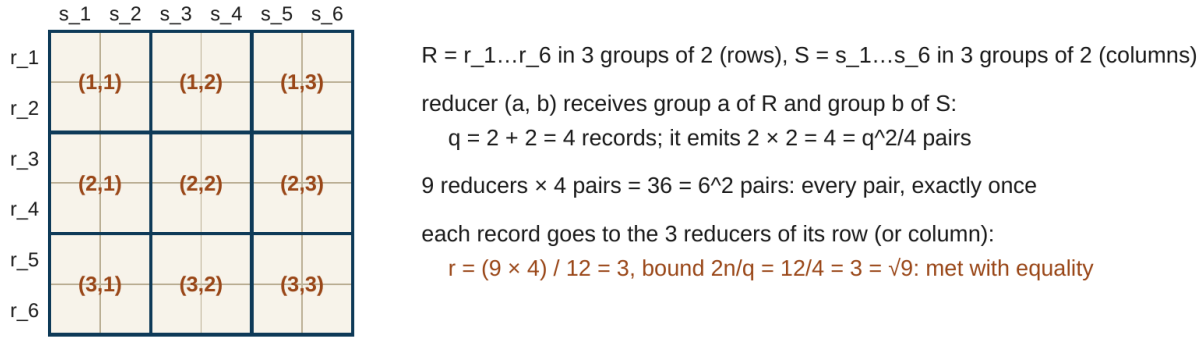


Figure 11.3: The replication rate on a six-by-six input with nine reducers.

whole buffer. The work is the same n additions; what changes is the constant per element, commonly by one to two orders of magnitude (measure it on your own machine). *pandas* adds labels on top of numpy: row indexes, named columns, joins, group-by, time series and missing values. Every distributed engine in this unit has an escape hatch to a local frame (`.toPandas()`, `.to_pandas()`). It is useful and dangerous in equal measure, because it collects the whole distributed dataset to one machine.

Notebooks. Notebooks are excellent for exploration and teaching, with plots next to the code that made them. Five properties make them bad at everything else:

problem	consequence
hidden state	a cell works only because an earlier cell was run twice
out-of-order execution	the result is not reproducible, and nothing warns you
diffs are JSON	code review is impractical
no functions, no tests	nothing can be reused or verified
copy-paste	the same transform six times, slightly different each time

The professional pattern is to explore in a notebook, move the logic into a `.py` module with tests, and import it back. The course labs go one step further: each run writes its own evidence, a hashed receipt appended to a hash chain (F.E-σ lab).

Dashboards, apps and orchestration. *Business-intelligence tools* (Power BI, Tableau, Looker, Metabase, Superset) are governed and shared, built for people who did not write the pipeline. *Python app frameworks* (Streamlit, Dash, Gradio, Panel, Marimo) are built and owned by you. *Orchestrators* run a directed acyclic graph of tasks on a schedule, with retries: an idea from the Hadoop era that outlived Hadoop (Oozie, Luigi, Airflow, Dagster, Prefect, and the managed equivalents of the cloud providers). *dbt* manages SQL transformations with version control, a dependency graph and tests; it is the tool you are most likely to be handed and least likely to be taught. All of them answer three questions: what runs, in what order, and what happens when it fails in the middle of the night? U3's scheduling theory is the mathematics of the second question.

Most data fits on one machine. Choose the lowest rung that works:

```
spreadsheet                 < ~1M rows, one-off, a human will read it
pandas / Polars             fits in RAM, one machine
DuckDB / SQLite             larger than RAM, still one machine (columnar,
  out-of-core)
Postgres / cloud warehouse  shared, concurrent, needs governance
Spark / Databricks          genuinely does not fit, or needs many machines
```

A laptop with tens of gigabytes of RAM handles hundreds of millions of rows in a columnar engine, depending on row width, and a single rented cloud machine can offer terabytes of RAM. Each rung down the list costs setup time, debugging time, money, and a smaller number of people who can help you. In this book's terms, the MPC model (U0 §4) is the right model only when n exceeds one machine's memory s. Move down a rung only when you have *measured* a reason: the same predict-then-verify discipline as every lab.

What lasts. Tools change every few years. Four skills have not: SQL used properly (window functions, common table expressions, reading a query plan); git used properly (branches, rebase, reviewing a diff; F.E1b); the command line (`bash`, `ssh`, `tmux`, `grep`, `awk`/`sed`, `jq`; F.E1); and being able to say where a number came from.

4.5 Five failure modes of data pipelines

A model is a function of its data, and the data is a function of a pipeline. In applied machine learning most failures are data failures, not model failures: more time goes into the correctness of joins than into the architecture of networks. Five failure modes recur. Each is an engineering property, and each is made precise elsewhere in the book.

1. Point-in-time correctness. To predict whether a customer will churn, the tempting query is

```
SELECT l.customer_id, l.churned, c.total_lifetime_spend, c.support_tickets
FROM   labels l
JOIN   customer_features c USING (customer_id)
```

It is wrong. `customer_features` holds *today's* values, and some of them were *caused* by the churn being predicted: a customer who is leaving files support tickets. The model learns from information that did not exist when the prediction would have been made (**future leakage**), scores superbly offline, and fails in production. The cut-off that matters is the **observation time** t_i of each training row: the moment at which, in production, the prediction would have been made. That is not the later time at which the outcome became known. A feature may use only values timestamped $\leq t_i$. The **as-of join** takes, for each row and each feature, the most recent value timestamped $\leq t_i$. It is leakage-free by construction, and no leakage-free join can use a more recent value (F.E5 §5.3 states and proves this).

Worked example. Three customers, all observed on 1 March, with their plan histories:

customer	observed	churned?	plan history
A	1 Mar	yes	basic (1 Jan) → cancelled (10 Mar)
B	1 Mar	no	basic (5 Jan) → premium (20 Feb)
C	1 Mar	yes	premium (1 Feb) → cancelled (3 Mar)

customer	churned?	naive `plan` (latest value today)	as-of `plan` (at 1 Mar)
A	yes	cancelled	basic
B	no	premium	premium
C	yes	cancelled	premium

Under the naive join, "cancelled" holds exactly on the churned rows: a perfect predictor. Both cancellations (10 March and 3 March) come *after* the observation time, so the feature *is* the outcome, recorded late. In production the model predicts at observation time, when no customer it still has to predict has been cancelled yet, so the feature never fires. The as-of join gives basic, premium, premium: a weak signal, but an honest one.

The fix needs history. Store each feature as a history table with a validity interval [`valid_from`, `valid_to`), where the current row has `valid_to` equal to `NULL`:

```
SELECT l.customer_id, l.churned, c.total_lifetime_spend, c.support_tickets
FROM   labels l
JOIN   customer_features_history c
       ON  c.customer_id = l.customer_id
       AND c.valid_from <= l.observation_time            -- as it was, then
       AND (c.valid_to > l.observation_time OR c.valid_to IS NULL)
```

If the intervals of one customer do not overlap, the row whose interval contains t_i is exactly the most recent value timestamped $\leq t_i$, so this query *is* the as-of join. (Without the `IS NULL` clause, every customer whose feature has not changed since t_i silently drops out.) The query works only if the warehouse *kept history*: Kimball's slowly-changing dimensions, Data Vault's satellites, or the time travel of a table format (§1.1). A table overwritten every night (`TRUNCATE; INSERT`) has no history, and no correct model can be trained from it. The modelling discipline of §1.1 is what makes honest machine learning possible.

2. Idempotency and backfills. Jobs fail, so the question is what a rerun does. A job is *idempotent* if running it $n \geq 1$ times on the same inputs leaves the same state as running it once: as a map on states, $f \circ f = f$ (F.C2b; U17 §6 meets the same property as the merge law $s \cup s = s$).

```
-- NOT idempotent: run twice -> every row duplicated -> revenue doubles
INSERT INTO sales SELECT ... WHERE date = '2026-03-01';
-- idempotent: run n times -> identical result
DELETE FROM sales WHERE date = '2026-03-01';
INSERT INTO sales SELECT ... WHERE date = '2026-03-01';
```

Two cautions. The pair should run in one transaction, or as one `MERGE` / `INSERT OVERWRITE PARTITION`; otherwise a failure between the two statements leaves the partition empty until the rerun, and readers see the gap. And partition the data by what you will need to reprocess, usually the date, so a rerun replaces one partition instead of rewriting the table. Every pipeline is eventually backfilled: design for it from the start.

3. Schema drift. An upstream team renames or retypes a column without knowing you depend on it. In ascending order of harm: the pipeline crashes (the good case); the column silently becomes `NULL`; an `int` widens to `string` and joins stop matching; the *meaning* changes while the name and type stay the same. Only the first announces itself. The defences are schema enforcement at the boundary, data-quality tests (row counts, null fractions, value ranges) and drift alerts, plus a working relationship with the upstream team. The general lesson, which U16 develops for incentive contracts, is that you can only rely on what you can *check*: a property with no test at the boundary is not part of the contract.

4. Training/serving skew. Training features come from a warehouse query and serving features from a live service: two implementations of "the same" feature. They drift apart through time zones, `NULL` mapped to 0 in one path and dropped in the other, late-arriving data, or a bug fixed in only one path. The model then meets inputs in production unlike those it was trained on. The defence is **one definition, one code path**, executed in both contexts. That is the purpose of a feature store: if both paths apply the same transform g to the same raw input, the offline and online values agree for every input (F.E5 §5.7).

5. Cost is a design constraint. On premises a bad query is slow; in the cloud it is also an invoice. Examples: a `SELECT *` on a 40 TB table in a warehouse that bills by bytes scanned; a cluster left running over a weekend; a dashboard refreshing every 15 minutes that three people

ever open; a cartesian join that runs for six hours and then fails. The habits: filter on the partition column (the engine then skips whole partitions); select only the columns you need (a column store then reads only those, F.E5 §4); sample during development; set auto-suspend on clusters; read the query plan before running. This is U0's first habit applied to a bill: price the rounds, the bytes and the local memory before you run.

failure mode	where it is made precise
1. point-in-time correctness	the as-of join is the unique maximal leakage-free training set (F.E5 §5.3)
2. idempotency and backfills	$f \circ f = f$; merge as a semilattice join, so reruns converge (F.C2b, U17 §6)
3. schema drift	rely only on what is checked at the boundary (by analogy with U16's verifiable contracts)
4. training/serving skew	one shared transform makes the two-path square commute (F.E5 §5.7)
5. cost	the three cost lenses, priced before you run (U0)

Check your reading.

Q4. Wordcount on n = 10^9 tokens with V = 10^5 distinct words, with M = 100 map partitions, 8-byte word keys and 4-byte integer counts. Using the unit's shuffle-byte formulas, groupByKey vs reduceByKey(+) ship approximately:
(A) about 12 GB for both – the map-side combine changes nothing (B) 12 GB vs between 1.2 MB and 120 MB (best: each word on one partition; worst: every word on every partition) (C) 12 GB vs 6 GB – a factor 2 (D) 1.2 MB vs 12 GB – reduceByKey ships MORE because of the combiner metadata

Q5. The covering argument (Afrati, Das Sarma, Salihoglu, Ullman, PVLDB 2013) shows that a 1-round algorithm whose reducers each receive at most q records has replication rate r $\geq$ 2n/q, i.e. r $\geq$ sqrt(M) at the balanced cap q = 2n/sqrt(M). To WHICH joins does this lower bound apply?
(A) every binary equi-join – any 1-round join must replicate each record sqrt(M) times (B) only 3-way joins such as R join S join T (C) the full cross-product and other output-dense joins (similarity/all-pairs, heavy-hitter keys); a well-spread equi-join achieves r = 1 by hash-routing each record to the reducer owning its key (D) no join – the bound applies only to sorting

Q6. A Mixture-of-Experts layer routes each token to a small number of expert shards on different accelerators. In two or three sentences, map this onto the unit's vocabulary: what plays the roles of keys, mappers, reducers, and shuffle bytes – and which of this unit's cost rules tells you whether the routing exchange is cheap?

5. The invariant (U5 instantiation)

RAM: the model itself — MapReduce formalises the computation as (map: $(k_1, v_1) \to \text{list}(k_2, v_2)$; shuffle: group by k_2; reduce: $(k_2, \text{list}(v_2)) \to \text{list}(y)$); determinism + data locality enable re-execution fault tolerance; sequential baseline = $O(n)$ scan.

PRAM / BSP bridge: a PRAM algorithm of depth D runs in $R = O(D)$ MapReduce rounds under the simulation's hypotheses (§3.1); but the bridge is an upper bound — good MPC algorithms batch many PRAM levels into one round, avoiding unnecessary shuffles. Low PRAM depth is a sufficient, not a necessary, condition for a low round count, and it bounds rounds, not bytes.

DPRAM / MPC (the accounting IS the content): (i) RDDs + lineage = the MPC model made concrete, with fault tolerance by re-execution and narrow/wide dependencies as the structural definition of stage boundaries; persist/cache = lazy materialisation that keeps the lineage (only `checkpoint()` cuts it), separate from the shuffle outputs on the map side's local disk; (ii) `reduceByKey` beats `groupByKey` by up to a factor $1/\rho$ when the aggregation is associative and commutative (map-side combine); its shuffle is $\sum_j V_j(v_{\text{agg}} + |k|)$ bytes, and $1/\rho$ is the best case, each key on one mapper; (iii) the three joins: shuffle ($|R| \cdot r + |S| \cdot s$ bytes, 1 shuffle round),

broadcast ($|S| \cdot s \cdot M$ bytes, no shuffle but one broadcast step of 1–2 communication steps, condition $|S| \cdot s < s_{\text{mem}}$), co-partitioned (0 shuffle bytes per join after a one-time partitioning shuffle; a narrow dependency, condition: the same partitioner on both sides, both persisted); (iv) Afrati et al.: a 1-round *cross-product* with each reducer receiving at most q records has replication rate $\geq 2n/q$, i.e. $\geq \sqrt{M}$ at the balanced cap $q = 2n/\sqrt{M}$ (an equi-join achieves replication rate 1 with no reducer cap; §4.3); MRC (KSV): $O(1)$ or $O(\log n)$ rounds for the course workhorses, with joins restricted to outputs that fit the total memory and heavy keys split (§4.3).

Check your reading.

Q7. Fill the U5 row of the invariant box from memory: the RAM baseline, the BSP bridge statement plus its caveat, and the byte cost + round count of the three joins of R (record size r) and S (record size s) on M partitions.

6. Lab — project track

Project-track lab (no weekly receipt): run these experiments free and locally in the course lab image (`labs/image/`) — or any Spark-in-Docker environment — as deep-dive and group-project material.

Setup. One-time Docker install plus the course lab image — see `labs/README.md`; everything runs free and locally on your own laptop.

Experiment 1 — `groupByKey` vs `reduceByKey`: shuffle bytes.

Generate $n = 10^7$ word tokens drawn from a Zipfian vocabulary of size $V \in \{10^3, 10^5\}$ (two regimes: high repetition / low repetition). For each V:

- Implement wordcount with `groupByKey` + `map(sum)`.
- Implement wordcount with `reduceByKey(+)`.

Using the Spark UI (or `SparkListener`) collect shuffle bytes written and shuffle bytes read for both.

Questions to answer in the deliverable: 1. By what factor does shuffle bytes differ between the two implementations? Does this match the prediction $n(v + |k|)/\left(\sum_j V_j \left(v_{\text{agg}} + |k|\right)\right)$ of §4.2.1, where V_j is the number of distinct words in map partition j? (It equals $1/\rho$ only when each word lives in one partition; count the V_j to check.) 2. At what vocabulary size V does the difference vanish (i.e. $\sum_j V_j \to n$, which $\rho \to 1$ forces)? Explain from the definition of the map-side combine. 3. State in three-lens vocabulary: which lens does the difference live in?

Experiment 2 — broadcast join vs shuffle join crossover.

Generate two RDDs: R with $|R| = 10^7$ records ($r = 20$ bytes each, join key uniform on $[0, K)$ with $K = 10^7$) and S with $|S| \in \{10^3, 10^4, 10^5, 10^6\}$ records ($s = 20$ bytes each, with distinct keys drawn uniformly from the same key space $[0, K)$, so the join has about $|R| \cdot |S|/K = |S|$ output pairs).

For each $|S|$: - Run a shuffle join (`R.join(S)`). - Run a broadcast join (manually: `S_small = sc.broadcast(S.collectAsMap()); R.mapPartitions(probe_S_small)`). `collectAsMap` keeps only one value per key: it is exact here only because the keys of S are distinct. With duplicate keys, build a dictionary of lists from `S.collect()` instead, or the broadcast join silently drops matches.

Collect: shuffle bytes (shuffle join) vs broadcast bytes (broadcast join); wall-clock time per join; number of Spark stages.

Questions: 1. At what $|S|$ does broadcast join become faster? Does the crossover match the theoretical threshold $|S| \cdot s < |R| \cdot r/M$? With $r = s = 20$ bytes the byte threshold is $|S| < 10^7/(M-1) \approx 10^7/M$, which lies inside the tested range $|S| \leq 10^6$ only when $M \geq 11$ executors; with fewer, extend the range of $|S|$ or expect broadcast to win on bytes at every tested size. 2. How many Spark stages does each approach use? Why? 3. Why does broadcast join fail (OOM or correctness) when $|S|$ is too large?

Experiment 3 — lineage recovery demo: kill an executor mid-job.

Submit a multi-stage job (e.g. wordcount → `sortByKey` → `reduceByKey` pipeline on a large dataset) using a Spark cluster with 2 workers. Count its stages in the DAG view first: wordcount's `reduceByKey` and the `sortByKey` each add a shuffle (and `sortByKey` also runs a small sampling job to choose its range bounds). In the Scala/Java API the final `reduceByKey` reuses `sortByKey`'s `RangePartitioner`, so it adds no shuffle; PySpark's `reduceByKey` hashes by default and adds one. After the first stage completes and the second stage begins: kill one worker (docker `stop spark-worker-2`).

- Observe Spark rescheduling the failed tasks (Spark UI → Stages → Executor tab).
- Note which tasks are re-executed. The killed worker's map outputs (its shuffle files, on its local disk) are lost with it, so the reduce tasks that fetch them fail with `FetchFailed`; Spark then resubmits the parent stage's tasks whose outputs were lost (and, recursively, any lost ancestor outputs) and reruns the failed tasks.
- Verify the final output is correct.
- Record: how many tasks were re-executed? How long did recovery add to total job time?

Question: if the job had called `persist()` on the RDD that stage 1 computes (and an earlier action had materialised it), would recovery be faster? Why? In your answer, separate three things: the cached blocks (held by executors and lost with the worker; persist keeps the lineage, so lost partitions are recomputed from it), a `checkpoint()` (written to reliable storage; it cuts the lineage), and the shuffle files (on the map side's local disk; lost with the worker).

Deliverable. Notebook + half-page cost-accounting for each experiment in the three-lens vocabulary (ops baseline / PRAM: depth and Brent / DPRAM: rounds + shuffle bytes + local memory). The cost-accounting — not the code — is the examination target.

7. Lean certificate target

Target (new — U5): MapReduce round-cost of reduce-tree.

The claim to certify: the wordcount-style reduce-tree, implemented as one shuffle (group by key) + one local fold, requires exactly $R = 1$ MPC round (one wide dependency = one shuffle) and $O(n)$ total shuffle bytes, *provided every key's group fits on one machine* (at most s records; §3.2 shows a Zipf-distributed corpus breaking this). With a combiner, each key arrives at its reducer as at most M partial aggregates, one per mapper, so a reducer holding the key set K receives at most $M \cdot |K|$ records: one round then suffices when $M \cdot |K| \leq s$, even if a raw group exceeds s; for a single heavy key that is $M \leq s$.

Proposition (reduce-tree round cost). For $n > 0$ input pairs, if every key's group fits on one machine (at most s records) and every (k, v) pair takes at most c bytes, then the reduce-tree

takes exactly one MPC round and shuffles at most $c \cdot n$ bytes.

```
-- SCHEMATIC target statement (names align to the BMLSPar.lean conventions at
  cert time):
-- A MapReduce round is modelled as: one map phase (local, 0 communication) +
-- one shuffle (all-to-all, counted in bytes) + one reduce phase (local).
-- rounds(reduce_tree n) counts the number of shuffle phases.

theorem reduce_tree_rounds (n s c : ℕ) (hn : 0 < n)
    (hfit : max_group_size (reduce_tree n) ≤ s)  -- every key's group fits on
      one machine
    (hc : pair_bytes (reduce_tree n) ≤ c) :      -- c bounds the bytes of one
      (k, v) pair
    rounds (reduce_tree n) = 1 ∧
    shuffle_bytes (reduce_tree n) ≤ c * n := by
  sorry  -- SCHEMATIC: cert target; students state the MPC model formally and
    fill this proof
```

Proof sketch. The reduce-tree implementation groups all (k, v) pairs by key k in one shuffle; each reducer, which by the hypothesis can hold its groups, then folds its list of values locally. The fold is a sequential loop — 0 additional rounds. The shuffle sends at most n pairs (each pair once), so shuffle bytes $\leq c \cdot n$ for the record size constant c. Round count = 1 shuffle = 1.

The Lean formalisation requires: (a) a model of MPC rounds (analogous to the BSP superstep abstraction in `BMLSPar.lean`); (b) a definition of `reduce_tree` as a MapReduce program; (c) an induction argument over the number of keys. This is a non-trivial cert target; students in Module-3 are asked to formalise it given the BSP stub.

Exercise U5.L1 (light). Load `BMLSPar.lean`. Write a Lean `structure MRCRound` with fields `map_work`, `shuffle_bytes`, `reduce_work`. Define `total_shuffle_bytes` of an `MRCProgram` (a list of rounds). State (without proof) `reduce_tree_rounds` and verify the `sorry`-stub type-checks.

Exercise U5.L2 (deep). Formally state and prove in Lean the shuffle-byte accounting of §4.2.1 for wordcount on M map partitions, partition j holding V_j distinct keys: `shuffle_bytes reduce_by_key = (Σ j, V j) * (v_agg + |k|)`, and `shuffle_bytes reduce_by_key` $\leq$ `shuffle_bytes group_by_key` when $v_{\text{agg}} \leq v$. Then, still with $v_{\text{agg}} \leq v$ and the added hypothesis that each key lives on exactly one partition ($\sum_j V_j = V$), prove `shuffle_bytes reduce_by_key` $\leq \rho$ `* shuffle_bytes group_by_key`, and give a counterexample showing that this inequality is false without that hypothesis. What additional precondition on `f` (the aggregation function) is needed for the two versions to compute the same counts?

LLM lens — where this unit lands on the running workload (U0L):
Pretraining data pipelines are this unit industrialised: corpus dedup, sharding, and shuffling at web scale are map-shuffle-reduce jobs. And expert parallelism (U0L §4) is a hash-partition: the router assigns tokens to expert-shards the way a shuffle assigns keys to reducers — tiny messages, locality wins, the shuffle lens.

8. Sources (credited; archived under `scadamale-book-sources`)

Dean, J. and Ghemawat, S. (2004). *MapReduce: Simplified Data Processing on Large Clusters.* OSDI 2004. The original paper. The map/reduce function signatures, the fault-tolerance argument (deterministic re-execution), and the data-locality design are taken from this source. Our exposition is our own; we follow the paper's structure but restate it in our three-lens vocabulary. Available: https://www.usenix.org/conference/osdi04/mapreduce-simplified-data-processing-large-clusters

Zaharia, M., Chowdhury, M., Das, T., Dave, A., Ma, J., McCauley, M., Franklin, M.J., Shenker, S., and Stoica, I. (2012). *Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing.* NSDI 2012. The RDD paper. The definition of RDDs (partitions, dependencies, compute function, partitioner, preferred locations), the narrow/wide dependency classification, and the lineage-based fault tolerance argument are from this paper. Our exposition is our own. Available: https://www.usenix.org/conference/nsdi12/technical-sessions/presentation/zaharia

Karloff, H., Suri, S., and Vassilvitskii, S. (2010). *A Model of Computation for MapReduce.* SODA 2010. The MRC complexity-theoretic framework. The formal definition of MRC_i ($O(n^{1-\epsilon})$ machines, each with $O(n^{1-\epsilon})$ memory, $O(\log^i n)$ rounds), the sublinear-memory constraint, and the simulation of CREW PRAM algorithms that use $O(n^{2-2\epsilon})$ total memory and processors are from this source. Our $O(1)$-round characterisation follows this paper.

Fish, B., Kun, J., Lelkes, Á.D., Reyzin, L. and Turán, G. (2015). *On the Computational Complexity of MapReduce.* DISC 2015; arXiv:1410.0245. Restates KSV's MRC definition and its CREW PRAM simulation ("MapReduce algorithms can simulate CREW PRAMs which use subquadratic total memory and processors"), which §4.3 quotes.

Afrati, F.N. and Ullman, J.D. (2010). *Optimizing Joins in a Map-Reduce Environment.* EDBT 2010 (journal version: *Optimizing multiway joins in a map-reduce environment*, IEEE TKDE 23(9), 2011). The *Shares* method: each attribute of the map-key gets a share (a number of hash buckets), and the shares are chosen to minimise communication for a fixed number k of reducers. It is an algorithm, hence an upper bound on communication, and proves no lower bounds; §4.3's grid for the cross-product and the $M^{1/3}$ shares for the triangle join are instances of it.

Afrati, F.N., Das Sarma, A., Salihoglu, S. and Ullman, J.D. (2013). *Upper and Lower Bounds on the Cost of a Map-Reduce Computation.* PVLDB 6(4), 2013. The replication rate as a function of the reducer size q, and the covering recipe for lower bounds that §4.3's theorem and E5.7 follow. (Without a reducer cap, hash routing gives an equi-join replication rate 1.)

Atserias, A., Grohe, M. and Marx, D. (2008). *Size Bounds and Query Plans for Relational Joins.* FOCS 2008; SIAM J. Comput. 42(4), 2013. The AGM bound on the output size of a join (for the triangle join, $\sqrt{|R|\,|S|\,|T|}$), used in §4.3 and E5.7.

Valiant, L.G. (1990). *A bridging model for parallel computation.* CACM 33(8):103–111. The BSP model. Cited for the superstep abstraction used in the PRAM→MPC bridge.

Codd, E.F. (1970). *A Relational Model of Data for Large Shared Data Banks.* Communications of the ACM 13(6), 377–387. The relational model and the logical/physical separation (§1.1).

Ghemawat, S., Gobioff, H. and Leung, S.-T. (2003). *The Google File System.* SOSP 2003. Replicated block storage on unreliable commodity machines (§1.1).

Kimball, R. (1996). *The Data Warehouse Toolkit.* Wiley. Dimensional modelling, the star schema, and slowly-changing dimensions (§1.1, §4.5).

Armbrust, M. et al. (2015). *Spark SQL: Relational Data Processing in Spark.* SIGMOD 2015. DataFrames and the Catalyst optimizer (§1.1).

Wikipedia. *MapReduce*, *Apache Spark*, *Resilient Distributed Dataset*, *Bulk synchronous parallel.* Background and cross-references.

9. Exercises

Light (examined core):

E5.1 State the types of `map` and `reduce` in the Dean–Ghemawat formulation. Write a MapReduce program (pseudocode `map` + `reduce` functions) that computes, for each URL in a web-crawl dataset, the number of distinct pages that link to it. Identify the keys k_1, k_2 and values v_1, v_2 in your program.

E5.2 In the three-lens vocabulary, account for the cost of wordcount on n tokens and V distinct words using `reduceByKey` vs `groupByKey` on M map partitions. Express shuffle bytes for each and give the reduction factor in terms of n, V and the numbers V_j of distinct words per partition, in the best and the worst case. For $n = 10^9$, $V = 10^5$ (English-vocabulary scale), $M = 100$, and 8-byte word keys with 4-byte integer values: compute the actual shuffle bytes for both implementations (for `reduceByKey`, the best and the worst case).

E5.3 An RDD lineage DAG has the following structure: two source RDDs A and B, joined with a wide dependency to produce C, then `map`ped (narrow) to produce D, then `reduceByKey`-d (wide) to produce E. (a) Identify the stage boundaries. (b) How many Spark stages are there? (c) If executor 2 fails after stage 1 completes but before stage 2 finishes: which partitions must be re-executed? Does the re-execution require re-reading A and B?

E5.4 For a broadcast join of R (10 GB) and S (50 MB) on a cluster with $M = 100$ executors each with 4 GB memory: (a) compute the shuffle bytes of a shuffle join vs a broadcast join; (b) determine whether broadcast join fits in executor memory; (c) state the condition $|S| \cdot s < s_{\text{mem}}$ for this example.

Deep (Module-3 / optional):

E5.5 (BSP bridge, tightness). Construct a 2-round MapReduce algorithm for matrix-vector multiplication $y = Ax$ where A is $n \times n$ and x is n-dimensional, with M machines each holding n/M rows of A. (a) Give the `map` and `reduce` functions for each round. (b) Account for shuffle bytes in each round. (c) Is this algorithm in MRC_0 or MRC_1? (d) Compare to the PRAM depth of matrix-vector multiply (U0 / U6 preview) and explain why the MPC algorithm uses fewer rounds than PRAM depth suggests.

E5.6 (Co-partitioned join amortisation). Suppose an iterative algorithm (e.g. PageRank) joins the same two RDDs R and S in each of k iterations. (a) If both are shuffled each iteration, what is the total shuffle cost after k iterations? (b) If R and S are co-partitioned and persisted after the first iteration, what is the total shuffle cost? (c) State the breakeven k as a function of $|R|$, $|S|$, and the co-partitioning setup cost (one partitioning shuffle of each, with the same partitioner). (d) For PageRank with the web graph (billions of edges), is the amortisation worth it?

E5.7 (3-way join replication rate). Consider the cyclic (triangle) join $R(A, B) \bowtie S(B, C) \bowtie T(C, A)$ with $|R| = |S| = |T| = n$ and M reducers, each receiving at most q records. A 1-round algorithm must route every output triple (r, s, t) (three records agreeing on their shared attributes) to a reducer that holds all three. (a) A reducer holding x, y, z records of R, S, T can emit at most $\sqrt{xyz}$ triples (the AGM bound), and some instance has $n^{3/2}$ output triples. Following the covering argument of §4.3, show that the replication rate is at least $\sqrt{3n/q}$, hence at least $M^{1/3}$ at the smallest cap with which M reducers can cover that instance, $q = 3n/M^{2/3}$. What does the same argument give for the 3-way cross product $R \times S \times T$? (b) Byte-account two alternative join plans: (i) 1-round with optimal replication $M^{1/3}$ (Shares: hash each of A, B, C into $M^{1/3}$ buckets); (ii) 2-round sequential joins ($R \bowtie S$ on B, then result $\bowtie T$). Assume the equi-join $R \bowtie S$ has selectivity 1 (each record of R matches exactly one record of S), so the

intermediate result $|R \bowtie S| = n$ with record size $r + s$. (c) Given $|R| = |S| = |T| = 10^9$ records of 100 bytes each and $M = 1000$: compute the total shuffle bytes for both plans. Which is smaller?

Light (added):

E5.8 (Iteration pricing, §4.1.1.) $B = 50$ GB, $k = 10$, $M = 5$ machines with 200 MB/s of disk each, and 3-way replication. (a) Compute the stable-storage bytes and disk time for MapReduce and for Spark with the working set cached. (b) How much memory per machine does Spark need? (c) For which k are the two equal?

E5.9 (Join crossover, §4.2.2.) R: 2×10^8 rows of 50 bytes; $M = 50$. (a) Compute the crossover size of S. (b) For $|S| \cdot s = 100$ MB and $= 500$ MB, give the network bytes of both joins and say which to use.

E5.10 (As-of join by hand, §4.5.) Observation time 1 June. D: free (1 Apr) → pro (15 May) → cancelled (20 Jun), churned. E: pro (1 Mar), unchanged, not churned. F: free (1 May) → cancelled (1 Jun), churned. (a) Fill in the naive and the as-of `plan` column. (b) F's cancellation is timestamped exactly at the observation time. Is using it leakage? What does your answer depend on, and what does it say about the label?

Chapter 12

06 matmul

U6 — Matrix multiplication: RAM → PRAM → DPRAM

The ScaDaMaLe book, unit 6 (examined core, Part II). Our own exposition; original sources credited in §8.

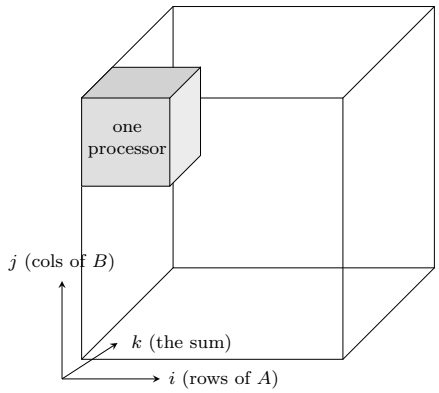
Instantiating the three-lens invariant. The U0 box for matrix multiplication:

> **RAM:** the naïve algorithm costs $\Theta(n^3)$ multiplications for $n \times n$ matrices — not tight. Strassen's 1969 recursion breaks the cubic barrier using 7 (not 8) recursive sub-multiplications, giving $W = O(n^{\log_2 7}) \approx O(n^{2.807})$. The M_k bilinear identities that drive Strassen are a one-time computation; the savings compound across recursion levels. The current record ($O(n^{2.372})$, Williams 2024) is stated, not reproduced.
>
> **PRAM:** block-recursive parallel matmul achieves $W = O(n^3)$, $D = O(\log n)$ by running its 8 sub-block multiplications in parallel and adding their results element-wise in constant depth. Strassen's recursion in parallel has the same depth, $O(\log n)$ — one level per recursion depth, each of constant PRAM depth — with $W = O(n^{\log_2 7})$ total work: it wins on work, not on depth. Brent bounds both.
>
> **DPRAM/MPC:** communication, not arithmetic, is the bottleneck. The classical 2D layout (Cannon / SUMMA) requires $\Theta(n^2/\sqrt{p})$ words per processor. The Ballard–Demmel–Holtz–Schwartz communication lower bound (2011; for matmul, Irony–Toledo–Tiskin 2004) asserts that any *classical* algorithm — one that forms all n^3 products $A[i,k]\,B[k,j]$, so not Strassen — with local memory s words per processor must move $\Omega\big(n^3/(p\sqrt{s})\big)$ words per processor when the work is load-balanced (and by some processor otherwise). The **2.5D** algorithm (Solomonik–Demmel) attains this bound with c-fold replication, $1 \le c \le p^{1/3}$, reducing communication by $\sqrt{c}$ at a memory cost of c. Spark's `BlockMatrix.multiply` is the practical anchor: a block-parallel matmul on a 2D grid of partitions whose shuffle bytes are directly measurable in the lab.

Entry kit — what this chapter assumes; follow a link if you need the memory jog (map: Figure 1):
F.M2 (matrix-as-three-things, norms, outer products)

1. Intuition — why matmul is the algorithm

Matrix multiplication is not an isolated academic exercise. It is the kernel inside every linear solver, SVD step (U7), neural network forward pass (U18), and graph power iteration (U12). The course returns to it repeatedly; this unit nails its cost accounting once so later units can cite back.



volume = the n^3 multiply–adds (fixed);
each processor owns a sub-cube: **compute** $\propto$ **volume, communication** $\propto$ **surface**

Figure 12.1: Matrix multiplication as a cube of n^3 elementary multiply–adds indexed by (i, j, k): the work is the volume and no partition changes it — what a partition chooses is the *surface area* each processor's sub-cube exposes, and faces are exactly the matrix entries that must be communicated. Balanced sub-cubes minimise surface per volume, which is the whole content of communication-avoiding (2.5D/3D) matmul, and the reason attention/MLP serving (U0L, U18) obsesses over blocking: the cube is the roofline's geometry.

The progression across the three machines shows three qualitatively different bottlenecks:

- **RAM:** arithmetic dominates. n^3 scalar multiplications for an $n \times n$ product. The interesting question is whether a recursion can reduce the constant exponent. Strassen's answer is yes — but at the cost of numerical instability and extra additions; in practice the threshold where Strassen's constant factor beats the cubic algorithm's smaller constant is $n \gtrsim 500$–1000, implementation-dependent.
- **PRAM:** many sub-blocks can be multiplied simultaneously. The interesting question is how deep the dependency chain is. Multiplying block C_{ij} requires a sum of outer products — a reduction, which has depth $O(\log n)$ on its own — and those reductions can run in parallel over (i, j).
- **DPRAM:** data lives on different processors. A scalar $A[i, k]$ needed to compute $C[i, j]$ may live on a processor that owns neither row i of C nor column j of B. Every element of A participates in n inner products; every element of B participates in n inner products. That participation costs *communication* — and the question is how much is unavoidable.

Figure 12.1 is the geometry every bound in this unit lives in.

Check your reading.
Q1. Across the three machine lenses, matmul's bottleneck changes qualitatively. The correct progression is:
(A) communication (RAM) → arithmetic (PRAM) → depth (DPRAM) (B) depth (RAM) → communication (PRAM) → arithmetic (DPRAM) (C) arithmetic dominates in all three lenses; only constants change (D) RAM: arithmetic (can a recursion beat n^3?); PRAM: dependency depth (the C_ij sums are log-depth reductions); DPRAM: communication (every element of A and B participates in n inner products, and that participation costs network words)

2. RAM — sequential matrix multiplication: cubic and Strassen

2.1 The naïve algorithm: $O(n^3)$

Problem. Given matrices $A, B \in \mathbb{R}^{n \times n}$, compute $C = AB$, where $C[i, j] = \sum_{k=1}^{n} A[i, k] \cdot B[k, j]$.

Algorithm (naïve). Three nested loops over i, j, k; for each (i, j) accumulate the inner product

over k. Cost: n^2 inner products, each costing n multiplications and $n-1$ additions. Total: n^3 multiplications, $n^2(n-1)$ additions $= \Theta(n^3)$ arithmetic operations.

Lower bound (trivial). C has n^2 entries; each entry depends on $2n$ inputs (row i of A and column j of B) and in the worst case is a distinct polynomial — so at minimum $\Omega(n^2)$ operations are needed. The naïve algorithm is not tight from below: the true arithmetic lower bound is still open (the ω conjecture: the exponent of matrix multiplication satisfies $\omega = 2$). We do not resolve this; we exhibit algorithms that improve the naïve exponent.

2.2 Strassen's algorithm: $O(n^{\log_2 7})$

The recursion (Strassen 1969). Partition A, B, C into 2×2 blocks of size $n/2 \times n/2$:

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}, \quad B = \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}, \quad C = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix}.$$

The naïve block multiplication computes C_{ij} using 8 block-matrix multiplications and 4 additions. Strassen shows that 7 products suffice. Define the 7 auxiliary matrices:

$$\begin{aligned} M_1 &= (A_{11} + A_{22})(B_{11} + B_{22}), \quad M_2 = (A_{21} + A_{22})B_{11}, \\ M_3 &= A_{11}(B_{12} - B_{22}), \quad M_4 = A_{22}(B_{21} - B_{11}), \\ M_5 &= (A_{11} + A_{12})B_{22}, \quad M_6 = (A_{21} - A_{11})(B_{11} + B_{12}), \\ M_7 &= (A_{12} - A_{22})(B_{21} + B_{22}). \end{aligned}$$

Then:

$$\begin{aligned} C_{11} &= M_1 + M_4 - M_5 + M_7, \quad C_{12} = M_3 + M_5, \\ C_{21} &= M_2 + M_4, \quad C_{22} = M_1 - M_2 + M_3 + M_6. \end{aligned}$$

Verification that these four expressions equal the correct block products of AB is an exercise (E6.1). Each identity holds in *any* ring, commutative or not: expanding never swaps a factor (every term is an A-block times a B-block, in that order), which is exactly what the recursion needs, since the blocks are matrices and do not commute. The proof is a finite algebraic check, making it an ideal Lean-cert target.

Practical caveats. The M_k expressions involve both additions and subtractions of blocks, so Strassen uses 18 additions/subtractions of $(n/2) \times (n/2)$ matrices (compared to 4 additions in the naïve block scheme). For small n the constant factor in the additions and the extra memory for the M_k temporaries makes Strassen slower than the naïve algorithm. The crossover threshold is implementation-dependent, typically $n \approx 500$–1000 in practice (though some highly tuned BLAS libraries only use Strassen for very large n). Strassen is also numerically less stable than naïve matmul (the subtractions amplify round-off).

2.3 Solving the recurrence

Strassen's algorithm satisfies the recurrence

$$T(n) = 7\,T(n/2) + O(n^2),$$

where $O(n^2)$ accounts for the 18 block additions and subtractions (each $O((n/2)^2) = O(n^2/4)$ operations, so 18 of them is still $O(n^2)$).

Proposition (Strassen's running time). By the Master Theorem (case 1: $\log_2 7 > 2$):

$$T(n) = \Theta\Big(n^{\log_2 7}\Big) \approx \Theta(n^{2.807}).$$

Proof (for the record). Set $a = 7$, $b = 2$, $f(n) = O(n^2)$. Since $\log_b a = \log_2 7 \approx 2.807 > 2$, case 1 applies: $T(n) = \Theta(n^{\log_2 7})$. □

For $n = 1000$, the *exponent gain* of Strassen over naïve is $n^3/n^{\log_2 7} = n^{0.193} \approx 3.8$, and it grows with n: about 5.9 at $n = 10^4$ ($10^{4\times 0.193}$), about 14 at $n = 10^6$. However, the exponent gain is not the speed-up, because it ignores the constants. Counting exactly, the naïve algorithm uses n^3 multiplications and $n^2(n-1)$ additions, $2n^3 - n^2$ operations; Strassen recursing down to 1×1 uses $T(n) = 7T(n/2) + 18(n/2)^2$, $T(1) = 1$, that is $T(n) = 7n^{\log_2 7} - 6n^2$. At $n = 1024$ these are 1,971,035,287 against 2,146,435,072, a ratio of 0.92: Strassen saves 8%, not a factor 4. Switching to the naïve algorithm below a cutoff of 8–16 brings the ratio to ≈ 0.51.

Current record. The exponent $\log_2 7 \approx 2.807$ has been improved by a sequence of algebraic constructions. As of 2024, the best known exponent is $\omega < 2.372$ (Williams 2024, improving Alman–Williams 2021 and prior work by Coppersmith–Winograd 1987). These algorithms are currently of theoretical interest only — their constant factors are astronomical and the crossover with naïve matmul exceeds any matrix size used in practice. We cite them and do not reproduce them.

Check your reading.
Q2. Strassen's algorithm beats the naive $Theta(n^3)$ arithmetic bound because at each level of the 2x2 block recursion it:
(A) skips the block additions entirely (B) still performs 8 block products, but on cleverly overlapping blocks (C) computes only 7 (not 8) block products via the M_k bilinear identities, at the price of 18 block additions/subtractions – giving T(n) = 7 T(n/2) + $O(n^2)$ = $Theta(n^(log2\ 7))$ (D) sorts the matrix entries so multiplications become lookups

3. PRAM — parallel matrix multiplication: block-recursive and Strassen

3.1 Block-recursive parallel matmul

Strategy. Partition $C = AB$ into 2×2 blocks as in §2.2. Each of the 8 block products needed by the naïve formula is independent and can be computed simultaneously. At each recursion level, we launch 8 parallel sub-problems, each writing its product into its own temporary. Each block $C_{ij} = A_{i1}B_{1j} + A_{i2}B_{2j}$ is then the sum of two temporaries — an element-wise addition, of depth $O(1)$ on the PRAM with one processor per entry (U1: a reduction of two terms).

Work. The recursion is $W(n) = 8W(n/2) + O(n^2)$. This solves to $W(n) = O(n^3)$ — same total arithmetic as naïve matmul. Work-optimality: the parallel algorithm does no extra arithmetic over the sequential classical algorithm; it is work-optimal with respect to the classical algorithm — not with respect to the best sequential algorithm, which is Strassen-like (§2.2) and asymptotically cheaper.

Depth. The 8 sub-problems at each level run in parallel, so they contribute only one level of depth, and the addition of the two temporaries contributes $O(1)$. There are $\log_2 n$ recursion levels (down to scalar products), so

$$D(n) = D(n/2) + O(1) \;=\; O(\log n).$$

In words, the sum of the n terms of each inner product is not a separate $O(\log n)$ reduction on top of the recursion: it *is* the recursion, two terms per level over $\log_2 n$ levels, and counting it again

would count the same levels twice. (In a fork-join model, where spawning the n^2 element-wise additions of a level itself costs depth $O(\log n)$, each level costs $O(\log n)$ and this algorithm and Strassen's both have depth $\Theta(\log^2 n)$; the comparison of §3.2 is the same in either model.)

Brent (instantiated). With $W = O(n^3)$, $D = O(\log n)$ on p processors:

$$T_p \;\le\; \frac{W}{p} + D \;=\; O\!\left(\frac{n^3}{p}\right) + O(\log n).$$

Parallelism $= W/D = O(n^3/\log n)$. Speed-up is within a constant of p for $p \ll n^3/\log n$, i.e. for all realistic cluster sizes when n is large.

3.2 Strassen recursion in parallel

Strassen's 7-way recursion runs its 7 sub-problems in parallel. The recursion depth is $\log_2 n$ levels; at each level the 7 sub-problems of size $n/2$ are independent (the M_k computations are independent given the block sums $A_{11} + A_{22}$, etc., which are themselves computed in parallel). The per-level work in additions is $O(n^2)$, parallelised to depth $O(1)$ (element-wise, fully parallel).

Depth. Each recursion level contributes depth $O(1)$ (the block additions are element-wise, depth $O(1)$ with $\Theta(n^2)$ processors at that level, and the 7 sub-problems run in parallel). There are $\log_2 n$ levels, so:

$$D_{\text{Strassen}} = O(\log n).$$

Work. $W_{\text{Strassen}} = O(n^{\log_2 7})$ — the same as sequential Strassen.

Brent (Strassen in parallel). With $W = O(n^{\log_2 7})$ and $D = O(\log n)$:

$$T_p \;\le\; O\!\left(\frac{n^{\log_2 7}}{p}\right) + O(\log n).$$

Parallelism $= O(n^{\log_2 7}/\log n)$. Note that this is *smaller* than the cubic algorithm's $O(n^3/\log n)$: the depths tie, so the algorithm with more work has more parallelism. Strassen wins on work only — which is what $T_p \le W/p + D$ rewards whenever p is below both parallelisms — and in practice the 7-way branching factor and added additions make its constant larger.

Brent instantiation: the two algorithms compared.

Algorithm	W	D	Parallelism
Block-recursive (naïve)	$O(n^3)$	$O(\log n)$	$O(n^3/\log n)$
Strassen parallel	$O(n^{2.807})$	$O(\log n)$	$O(n^{2.807}/\log n)$

For large n: Strassen parallel wins on total work; the depths tie, and the naïve algorithm has the larger parallelism.

Check your reading.
Q3. Block-recursive parallel matmul (the 8-way naive recursion, all sub-blocks in parallel) has work W = $O(n^3)$ and depth:
(A) O(1) – everything is independent (B) $O(log^2$ n) – log n recursion levels, times an O(log n)-depth reduction for the inner-product sums (C) O(n) – one level per row (D) O(log n) – the same as parallel Strassen

4. DPRAM / MPC — communication-avoiding matrix multiplication

This is the **centrepiece of U6**: the three-way tradeoff between memory per processor, number of processors, and communication volume, made rigorous by the Ballard et al. lower bound.

4.1 The 2D layout (Cannon / SUMMA): setup and cost

Setup. Distribute an $n \times n$ matrix across p processors arranged on a $\sqrt{p} \times \sqrt{p}$ logical grid. Processor (r, c) owns block A_{rc} (rows $r \cdot n/\sqrt{p}$ through $(r+1) \cdot n/\sqrt{p} - 1$) and similarly for B and C.

2D block distribution (stated). Tile A into $\sqrt{p} \times \sqrt{p}$ blocks of size $(n/\sqrt{p}) \times (n/\sqrt{p})$; similarly for B. Processor (r, c) computes $C_{rc} = \sum_k A_{rk} B_{kc}$. The inner-product sum is a sequence of outer products; each term requires the blocks A_{rk} (owned by row r of the grid) and B_{kc} (owned by column c) to be available on processor (r, c). This requires communication.

Cannon's algorithm (1969). A classical $O(1)$-round-per-step algorithm for the 2D grid: shift rows of A and columns of B in a co-ordinated pattern so that each processor accumulates its inner-product one outer-product-contribution at a time, using only nearest-neighbour communication. After $\sqrt{p}$ shift steps, each processor has computed its C_{rc} block. Total communication per processor: each of the $\sqrt{p}$ steps sends and receives two blocks of size $(n/\sqrt{p})^2$ words, so total words moved per processor is $\sqrt{p} \cdot 2 \cdot (n/\sqrt{p})^2 = 2n^2/\sqrt{p}$ words.

SUMMA (Scalable Universal Matrix Multiplication Algorithm, van de Geijn and Watts 1997). A broadcast-based variant: at step k, processor (r, k) broadcasts block A_{rk} across row r, and processor (k, c) broadcasts block B_{kc} down column c; then every processor (r, c) updates $C_{rc} += A_{rk} \cdot B_{kc}$. After $\sqrt{p}$ steps, C is complete. Communication volume per step: $(n/\sqrt{p})^2$ words broadcast across $\sqrt{p}$ processors in a row (or column) — total bytes per processor per step is $O(n^2/p)$; over $\sqrt{p}$ steps: $O(n^2/\sqrt{p})$ words total per processor — the same as Cannon.

The 2D communication cost. Both algorithms move $\Theta(n^2/\sqrt{p})$ words per processor. The total across all p processors is $\Theta(n^2\sqrt{p})$ — growing with p. This seems counterintuitive: more processors means more total communication. It reflects a fundamental memory vs. communication tradeoff, which the lower bound makes precise.

4.2 The memory–communication tradeoff: the Ballard–Demmel–Holtz–Schwartz lower bound

Theorem (Irony, Toledo, Tiskin 2004; Ballard, Demmel, Holtz, Schwartz 2011, informal). Any *classical* algorithm for the product $C = AB$ of two $n \times n$ matrices — one that computes all n^3 products $A[i,k]\,B[k,j]$, as the naïve algorithm, Cannon and SUMMA do; Strassen does not — running on a parallel machine with p processors each holding at most s words of local memory (the book's symbol from U0; BDHS write M for the same quantity), with the n^3 multiplications balanced across the processors, must communicate (move words between processors or between memory levels) at least

$$\Omega\left(\frac{n^3}{p\sqrt{s}}\right)$$

words per processor. Without load balance the same bound holds for *some* processor — one that performs at least n^3/p of the multiplications.

Intuition. The proof rests on a geometric lemma, the Loomis–Whitney inequality: a set V of points (i, j, k) of the cube of Figure 12.1 — that is, of multiply-adds $C[i,j] \mathrel{+}= A[i,k]\,B[k,j]$ — satisfies $|V| \le \sqrt{|V_A|\,|V_B|\,|V_C|}$, where V_A, V_B, V_C are its shadows on the three faces, the entries of A, B, C it touches. Thus a processor holding s words of A, B, C in local memory can perform at most $O(s^{3/2})$ of the required multiply-adds without fetching new data (this is the *arithmetic intensity* of matmul — the ratio of multiply-adds to words accessed is at most $O(\sqrt{s})$), and a classical computation has exactly n^3 multiply-adds, so it needs at least $\Omega(n^3/\sqrt{s})$ words moved

in total. Spreading this lower bound over p processors gives the per-processor bound. The sequential ancestor is Hong–Kung 1981, whose red-blue pebble game and S-partition argument give $\Omega(n^3/\sqrt{M})$ I/Os for a fast memory of M words (E6.5); Irony, Toledo and Tiskin carried it to distributed memory through Loomis–Whitney.

Strassen-like algorithms. The theorem does not cover Strassen, which never forms some of the n^3 products. For Strassen-like algorithms Ballard, Demmel, Holtz and Schwartz (SPAA 2011; JACM 2012) prove, by graph expansion, the analogous bound $\Omega\big((n/\sqrt{s})^{\log_2 7} \cdot s/p\big)$. Since the classical bound is $(n/\sqrt{s})^3 \cdot s/p$, Strassen's is smaller by the factor $(n/\sqrt{s})^{3-\log_2 7}$: Strassen saves communication as well as arithmetic.

When is the bound attained? Only in a range of memory sizes, $n^2/p \lesssim s \lesssim n^2/p^{2/3}$. Below n^2/p the matrices do not fit in the machine at all. Above $n^2/p^{2/3}$ extra memory stops helping: a second, *memory-independent* bound says that some processor must communicate $\Omega(n^2/p^{2/3})$ words whatever s is (Ballard, Demmel, Holtz, Lipshitz, Schwartz, SPAA 2012), and the two bounds meet at $s = n^2/p^{2/3}$, where $n^3/(p \cdot np^{-1/3}) = n^2/p^{2/3}$. Within that range, algorithms that attain the bound are called **communication-avoiding**.

Instantiation for the 2D layout. Each processor holds $s = 3(n/\sqrt{p})^2$ words (three blocks of A, B, C), so $\sqrt{s} = O(n/\sqrt{p})$. The lower bound gives $\Omega(n^3/(np^{-1/2} \cdot p)) = \Omega(n^2/\sqrt{p})$ words per processor. The 2D algorithms achieve this — they are communication-optimal for the 2D layout.

4.3 2.5D replication: trading memory for less communication

The idea. Replicate the data c-fold across a third dimension: instead of a $\sqrt{p} \times \sqrt{p}$ grid, use a $\sqrt{p/c} \times \sqrt{p/c} \times c$ processor grid. Each of the c layers holds a full copy of A and B; computation is split across layers. With c-fold replication, local memory per processor grows by the factor c — block size $n/\sqrt{p/c}$ gives $s = \Theta((n/\sqrt{p/c})^2) = \Theta(n^2c/p)$ words — and communication per processor decreases by a factor of $\sqrt{c}$:

$$\text{words per processor} = \Theta\left(\frac{n^2}{\sqrt{p} \cdot \sqrt{c}}\right), \qquad 1 \le c \le p^{1/3}.$$

Replication is not free. Copying A and B from the front layer to the other $c-1$ layers, and summing the c partial results for C across the layers at the end, moves $\Theta(n^2c/p)$ words per processor — one block of each matrix. This stays below the shift cost $n^2/\sqrt{pc}$ exactly while $c^{3/2} \le \sqrt{p}$, that is, while $c \le p^{1/3}$; beyond it the replication itself dominates. That is why the range is $1 \le c \le p^{1/3}$.

For every c in that range the algorithm attains the lower bound of §4.2 for its memory: with $s = \Theta(n^2c/p)$, $n^3/(p\sqrt{s}) = \Theta(n^2/\sqrt{pc})$. This family is the **2.5D algorithm** (Solomonik–Demmel 2011; the parallel lower bound it meets is Irony–Toledo–Tiskin 2004): it achieves the communication lower bound by using extra memory for replication. Its endpoints are the 2D algorithm ($c = 1$) and the 3D algorithm ($c = p^{1/3}$), where $M = \Theta(n^2/p^{2/3})$ memory per processor gives communication $\Theta(n^2/p^{2/3})$ words per processor — meeting the memory-independent bound of §4.2 as well, so more memory cannot buy less.

Summary (stated for reference). The 2.5D algorithm with c-fold replication, $1 \le c \le p^{1/3}$: - Memory per processor: $s = O(n^2c/p)$ - Communication per processor: $O(n^2/\sqrt{pc})$ for the shifts, plus $O(n^2c/p)$ for replication and reduction, which is no larger in this range - Setting $c = p^{1/3}$ (the 3D endpoint): $s = O(n^2p^{-2/3})$, communication $= O(n^2p^{-2/3})$ — optimal.

The tradeoff: $\sqrt{c}$ less communication, c more memory. The algorithm name "2.5D" reflects its use of a $\sqrt{p/c} \times \sqrt{p/c} \times c$ grid, which interpolates between the 2D grid ($c = 1$) and a 3D grid

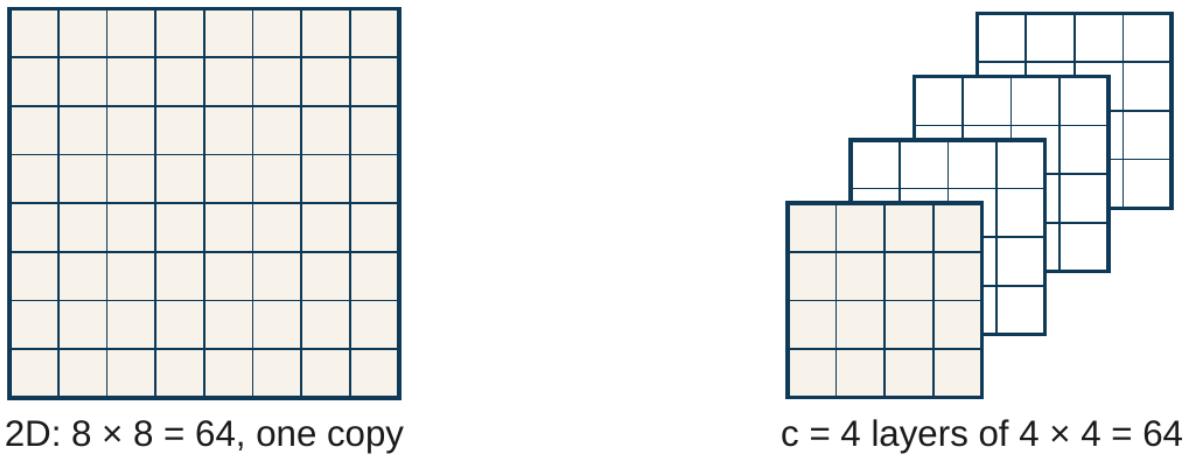


Figure 12.2: 2D against 2.5D: 64 processors as one 8×8 grid, or as $c = 4$ layers of 4×4.

($c = p^{1/3}$, the memory-intensive extreme).

Figure 12.2 sets the two layouts side by side for $p = 64$.

4.4 Spark `BlockMatrix.multiply`: the practical anchor

Spark's `BlockMatrix` (in `org.apache.spark.mllib.linalg.distributed`) represents an $n \times n$ matrix as an RDD of `((blockRow, blockCol), Matrix)` tuples with a configurable `rowsPerBlock` × `colsPerBlock` block size. The `multiply(other: BlockMatrix)` method computes $C = AB$ via a distributed outer-product / block accumulation:

1. **Placement.** The blocks of C are laid out on a square grid of P partitions (Spark's `GridPartitioner`), the 2D layout of §4.1. Each block A_{ik} is needed by every partition that holds a block C_{ij} of its block-row, and each block B_{kj} by every partition holding a block of its block-column; Spark computes these destinations and sends each block to all of them — a shuffle in which blocks are *replicated*, not merely moved.
2. **Local matmul.** Each partition multiplies the pairs (A_{ik}, B_{kj}) it received.
3. **Reduction.** Partial blocks $C_{ij}^{(k)}$ are summed by key (i, j) with `reduceByKey` (combined inside each partition first) — a last, smaller shuffle.

What the block size changes. The bytes moved are set by the partition grid P, through the replication of step 1 (§4.1's $\Theta(n^2\sqrt{p})$), not by the block size itself; the block size changes the number and size of the records, the task overhead, the efficiency of the local matrix multiplication, and how many partitions the product can have at most (one per block of C). The lab measures both (§6).

Sparse matmul (one paragraph). When A or B is sparse (a common case in graph algorithms and recommender systems), representing them as RDDs of (row, column, value) triples and computing the matrix-vector product Ax reduces to a join between A's entries and x's entries on the column index — exactly a join of U5 §4.2.2 (U5's exercise E5.5 builds this matrix–vector product in MapReduce). For sparse A with nnz non-zeros, the communication of a matrix–*vector* product is proportional to nnz rather than n^2, making it far cheaper than dense for typical graphs and recommendation matrices. The Spark `CoordinateMatrix` and `IndexedRowMatrix` types support this pattern. For a product of two sparse *matrices* the same join is not $O(nnz)$ in general; Lab 4 measures what it moves, and asks why.

Check your reading.

Q4. Cannon and SUMMA both move *Theta*(n^2 / sqrt(p)) words per processor. For n = 4096 matrices on an 8 x 8 grid (p = 64), the quantity n^2 / sqrt(p) evaluates to approximately:
(A) 2.6 x 10^5 words (B) 2.1 x 10^6 words (C) 1.7 x 10^7 words (D) 1.1 x 10^9 words

Q5. The communication lower bound (Irony-Toledo-Tiskin 2004; Ballard-Demmel-Holtz-Schwartz 2011) says any CLASSICAL matmul (all n^3 products formed), load-balanced, with local memory s must move *Omega*(n^3 / (p sqrt(s))) words per processor. How does the 2.5D algorithm communicate LESS than the 2D layout without violating this bound?
(A) it replicates A and B c-fold across a third grid dimension: memory per processor grows by the factor c, and communication drops by sqrt(c) to *Theta*(n^2 / sqrt(pc)) – attaining the lower bound at the larger memory size s = *Theta*(n^2 c / p) (B) it performs fewer arithmetic operations, Strassen-style, so less data is needed (C) it compresses the blocks before sending them (D) it uses randomness, to which the lower bound does not apply

Q6. Tensor parallelism shards each transformer layer's weight matrices across accelerators and synchronises activations every layer. In two or three sentences, map this onto the unit's vocabulary: which quantity of this unit is the per-layer sync, what does the Ballard et al. bound say about making it vanish, and where do prefill vs decode sit on the arithmetic-intensity picture?

5. The invariant (U6 instantiation)

RAM: naïve matmul costs $\Theta(n^3)$ arithmetic operations. Strassen's recursion uses 7 (not 8) sub-multiplications, giving $W = O(n^{\log_2 7}) \approx O(n^{2.807})$ via the recurrence $T(n) = 7T(n/2) + O(n^2)$; the M_k bilinear identities are verifiable algebra (exercise E6.1). The current best exponent is $O(n^{2.372})$ (Williams 2024). Practical threshold for Strassen: $n \gtrsim 500$–1000; numerical instability is a real caveat.

PRAM: block-recursive parallel matmul achieves $W = O(n^3)$, $D = O(\log n)$; Strassen in parallel achieves $W = O(n^{\log_2 7})$, $D = O(\log n)$. Brent: $T_p \leq W/p + D$; parallelism $\Theta(n^3/\log n)$ (naïve) or $\Theta(n^{2.807}/\log n)$ (Strassen) — the depths tie, and Strassen wins on work only.

DPRAM/MPC: the Ballard–Demmel–Holtz–Schwartz lower bound says any classical (non-Strassen) algorithm needs $\Omega(n^3/(M^{1/2}p))$ words per processor under load balance (where M = local memory), attainable for $n^2/p \lesssim M \lesssim n^2/p^{2/3}$. The 2D layout (Cannon/SUMMA) achieves $\Theta(n^2/\sqrt{p})$ words per processor (optimal for 2D). The 2.5D algorithm with c-fold replication, $1 \leq c \leq p^{1/3}$, achieves $O(n^2/\sqrt{pc})$, buying $\sqrt{c}$ less communication at $c\times$ more memory. Spark `BlockMatrix.multiply` realises the 2D block scheme; the number of partitions it lays the product on is the practical dial. A sparse matrix–vector product collapses to a join (U5), with communication $O(nnz)$ instead of $O(n^2)$; for two sparse matrices the join also moves its partial products (Lab 4).

Check your reading.

Q7. Fill the U6 row of the invariant box from memory: RAM costs (naive and Strassen, with the recurrence), PRAM work/depth for both parallel variants, and the DPRAM story (lower bound, 2D achievement, 2.5D trade).

6. Lab (Lab 4 — blocked matmul)

Lab 4 — blocked matmul: the `lab4-blocked-matmul/` folder of the course repository, and the lab chapter Lab 4 in the Laboratory exercises part. It is written in Scala (Spark's own language) and follows the same receipt pattern as the other labs; everything runs free and locally on your laptop. The experiments below are the lab's intellectual content.

Experiment 1 — BlockMatrix multiply: block-size sweep.

Two random dense 2048×2048 matrices as Spark `BlockMatrix` objects, with `rowsPerBlock` = `colsPerBlock` $\in \{128, 256, 512, 1024\}$. For each block size, run `A.multiply(B)` and record, from Spark's own task metrics: the shuffle bytes written and read, the number of stages, the wall-clock time, and the number of partitions holding the product's blocks. Check every product with Freivalds' randomized test ($Cx = A(Bx)$ for a random vector x).

Question: block sizes 128 and 256 move the same number of matrix copies (about 17: 17.03 and 17.02), while 512 and 1024 move fewer. Why? Derive the number of copies from the number of partitions, with the 2D layout of §4.1.

Experiment 2 — Sparse matmul two ways: blocks vs. a join of entries.

Two random sparse 1024×1024 matrices with density $\rho \in \{0.001, 0.01, 0.05, 0.1\}$, multiplied as sparse blocks (`CoordinateMatrix.toBlockMatrix`, then `multiply`) and as a join of their (i, k, a) and (k, j, b) entries on k, followed by a sum per (i, j). Record the shuffle bytes of each, the non-zeros, and the number of pairs (A_{ik}, B_{kj}) that are both non-zero.

Question: why does the join's shuffle grow so much faster with ρ than the number of non-zeros does, while the blocks' shuffle does not?

A known issue. The lab seeds the two random matrices of each experiment from overlapping seed ranges, so some blocks of B repeat blocks of A (rows, in Experiment 2); their bytes and the Freivalds checks are unaffected.

Deliverable. The receipt, with one or two sentences on each question. Optional: plots, and a half-page cost-accounting in the three-lens vocabulary.

7. Lean certificate target

Current target (U6 — new): certify the work bound for Strassen's recursion (SCHEMATIC).

The goal is to state and prove that the recurrence $T(n) = 7 \cdot T(n/2) + c \cdot n^2$ (for a constant c) solves to $T(n) = O(n^{\log_2 7})$. This is an application of the Master Theorem, which can be stated as a general lemma and then specialised.

```
-- SCHEMATIC target (align names to BMLSPar.lean conventions at cert time):
theorem strassen_work (n : ℕ) (hn : 0 < n) :
    ∃ (C : ℕ), strassen_ops n ≤ C * n ^ (Real.log 7 / Real.log 2) := by
  sorry -- cert target: instantiate Master Theorem lemma with a=7, b=2,
    f(n)=O(n^2)
```

Exercise U6.L1 (light, Lean). Load `BMLSPar.lean`. State the Strassen recurrence as a Lean `def strassen_recurrence : ℕ → ℕ` with base case `strassen_recurrence 1 = 1` and recursive case `strassen_recurrence n = 7 * strassen_recurrence (n/2) + c * n^2`. Verify the first 4 unfoldings numerically (i.e. compute `strassen_recurrence 2`, `strassen_recurrence 4`, `strassen_recurrence 8`, `strassen_recurrence 16`) and check them against the closed form for $n = 2^k$,

$$T(2^k) = \left(1 + \tfrac{4c}{3}\right)7^k - \tfrac{4c}{3}\,4^k \;=\; 7^k + \frac{4c\,(7^k - 4^k)}{3}$$

(the second form stays in $\mathbb{N}$: 3 divides $7^k - 4^k$). Note that the coefficient of 7^k is $1 + 4c/3$, not 1: the additions contribute $T - 7^k = (4c/3)(7^k - 4^k) = \Theta(n^{\log_2 7})$, the same order as the leading term, not $O(n^2)$. This grounds the cert target in concrete arithmetic before the asymptotic statement.

Exercise U6.L2 (deep, Lean). State and prove the M_k algebraic identity for one of the seven expressions — for example, verify that $M_1 + M_4 - M_5 + M_7 = A_{11}B_{11} + A_{12}B_{21}$ over an arbitrary ring, not necessarily commutative (`[Ring R]`, closed by `noncomm_ring`). The commutative version (`[CommRing R]`, closed by `ring`) certifies only the 2×2 scalar case, not the block step, whose entries are matrices. This is a ring-arithmetic proof (no induction needed) — a finite algebraic identity — making it a realistic Lean certificate at this stage of the provedog development.

> **LLM lens** — where this unit lands on the running workload (U0L):
> Attention and feed-forward blocks ARE the matmuls this unit prices: prefill is the compute-bound side of the roofline, decode the bandwidth-bound side (U0L §2), and "communication-avoiding" is the entire serving game — tensor parallelism's per-layer sync (U0L §4) is this unit's communication lower bound wearing production clothes.

8. Sources (credited; archived under `scadamale-book-sources`)

Strassen (1969). V. Strassen, *Gaussian elimination is not optimal*, Numerische Mathematik 13(4):354–356, 1969. The original 7-multiplication recursion and M_k identities. — The primary source; algorithm and recurrence in §2.2 above follow Strassen directly.

Ballard, Demmel, Holtz, Schwartz (2011). G. Ballard, J. Demmel, O. Holtz, O. Schwartz, *Minimizing communication in numerical linear algebra*, SIAM Journal on Matrix Analysis and Applications 32(3):866–901, 2011. — Communication lower bound stated in §4.2 above, for classical algorithms, generalising Irony–Toledo–Tiskin from matmul to classical linear algebra.

Irony, Toledo, Tiskin (2004). D. Irony, S. Toledo, A. Tiskin, *Communication lower bounds for distributed-memory matrix multiplication*, Journal of Parallel and Distributed Computing 64(9):1017–1026, 2004. — The parallel lower bound of §4.2 for classical matmul, through the Loomis–Whitney inequality; with Hong–Kung, the origin of the bound.

Ballard, Demmel, Holtz, Schwartz (2012). G. Ballard, J. Demmel, O. Holtz, O. Schwartz, *Graph expansion and communication costs of fast matrix multiplication*, SPAA 2011; Journal of the ACM 59(6), 2012. — The communication lower bound for Strassen-like algorithms in §4.2.

Ballard, Demmel, Holtz, Lipshitz, Schwartz (2012). G. Ballard, J. Demmel, O. Holtz, B. Lipshitz, O. Schwartz, *Brief announcement: strong scaling of matrix multiplication algorithms and memory-independent communication lower bounds*, SPAA 2012. — The memory-independent bound $\Omega(n^2/p^{2/3})$ that limits the range of §4.2–4.3.

Cannon (1969). L. E. Cannon, *A cellular computer to implement the Kalman filter algorithm*, Ph.D. dissertation, Montana State University, 1969. — The original 2D-shift distributed matmul algorithm; credited in §4.1.

Solomonik, Demmel (2011). E. Solomonik, J. Demmel, *Communication-optimal parallel 2.5D matrix multiplication and LU factorization algorithms*, Euro-Par 2011, LNCS 6853, 90–109. — The 2.5D replication scheme in §4.3.

Hong, Kung (1981). J.-W. Hong, H. T. Kung, *I/O complexity: the red-blue pebble game*, STOC 1981. — Foundational sequential I/O lower bound (the red-blue pebble game and its S-partitions) underlying the parallel bound of §4.2 and E6.5; not reproduced, cited for the proof chain.

Williams (2024). V. V. Williams, N. Xu, Z. Xu, R. Zhou, *New bounds on matrix multiplication: from Alpha to Omega* (Williams, Xu, Xu, Zhou), SODA 2024. — Current record $\omega < 2.372$; stated in §2.3.

Coppersmith, Winograd (1987). D. Coppersmith, S. Winograd, *Matrix multiplication via arithmetic progressions*, STOC 1987. — Long-standing $O(n^{2.376})$ result; the starting point of the modern fast matmul line; cited.

Wikipedia. *Matrix multiplication algorithm*, *Strassen algorithm*, *Communication-avoiding algorithm*. — Background and cross-references.

MIT OCW 18.337. *Parallel Computing and Scientific Machine Learning*, MIT OpenCourseWare. Lectures on distributed linear algebra and communication-avoiding algorithms. — Freely available; credited.

9. Exercises

Light (examined core):

E6.1 *Verify the M_k identities.* Expand $M_1 + M_4 - M_5 + M_7$ in terms of the A_{ij}, B_{ij} block entries and verify it equals $A_{11}B_{11} + A_{12}B_{21} = C_{11}$. Repeat for $C_{12} = M_3 + M_5$. (The verification is a polynomial identity over the block entries — no matrices need to be multiplied; treat the blocks as elements of a ring and expand, keeping every product in its order, A-block times B-block, since blocks do not commute.)

E6.2 *Strassen recurrence.* State the recurrence for Strassen's algorithm: $T(n) = 7T(n/2) + O(n^2)$. Identify $a = 7$, $b = 2$, $f(n) = O(n^2)$. Apply the Master Theorem to derive $T(n) = \Theta(n^{\log_2 7})$. What is $\log_2 7$ to three decimal places, and what does it say about the savings over the naïve n^3 algorithm for $n = 1024$? (Compare the exponent gain with the exact operation counts of §2.3.)

E6.3 *Brent bound for parallel matmul.* For block-recursive parallel matmul with $W = O(n^3)$ and $D = O(\log n)$: (a) apply the Brent bound to find T_p; (b) for $n = 4096$ and $p = 256$, compute the numerical values of both terms in the bound; (c) what is the maximum p for which speed-up is within a factor 2 of ideal?

E6.4 *2D SUMMA communication volume.* In the SUMMA algorithm on a $\sqrt{p} \times \sqrt{p}$ processor grid for $n \times n$ matrices: (a) at each step $k \in \{1, \ldots, \sqrt{p}\}$, how many words does each other processor of the row receive from a single row-broadcast of a block? (b) Over all $\sqrt{p}$ steps, what is the total number of words received by processor (r, c) from row-broadcasts? (c) Adding column-broadcasts: what is the total communication volume per processor? Show that the answer is $\Theta(n^2/\sqrt{p})$.

Deep (Module-3 / optional):

E6.5 *Memory–communication tradeoff curve.* Consider a sequential classical algorithm (one that forms all n^3 products) for $n \times n$ matrix multiplication running on a two-level memory hierarchy (fast memory of size M, slow memory unlimited). The Hong–Kung lower bound implies that the number of slow-memory reads and writes (I/O operations) is $\Omega(n^3/\sqrt{M})$. (a) State this bound clearly, identifying the analogy with the Ballard et al. parallel bound. (b) Plot the tradeoff curve: as M grows from 1 to n^2, how does the I/O lower bound change? (c) At what M does the lower bound become $O(n^2)$ (i.e. one input matrix fits in fast memory)? What does this say about the optimal cache size for matmul?

E6.6 *Derive the 2D SUMMA communication volume.* Prove rigorously (not just by analogy) that the SUMMA algorithm makes each processor receive exactly $2(n^2/\sqrt{p}) \cdot (1 - 1/\sqrt{p})$ words (not counting the local C_{rc} buffer). *Hint:* at step k, each processor of the row other than the block's owner receives $(n/\sqrt{p})^2$ words. Sum over all $\sqrt{p}$ steps for both the A-broadcast (row) and B-broadcast (column). Reconcile this with the $\Theta(n^2/\sqrt{p})$ bound.

E6.7 *2.5D memory vs. communication.* The 2.5D algorithm with c-fold replication on p processors achieves communication $\Theta(n^2/\sqrt{pc})$ per processor with local memory $M = \Theta(n^2 c/p)$ per processor, for $1 \le c \le p^{1/3}$. (a) Show that setting $c = p^{1/3}$ minimises the communication under the constraint that memory $M \le n^2/p^{2/3}$ (a given memory budget). (b) By how much does the communication decrease (compared to 2D, $c = 1$) when $c = 4$? When $c = 16$? (c) For $p = 1024$ processors and $n = 16384$, take $c = 4$, the $16 \times 16 \times 4$ grid (why not $c = p^{1/3} \approx 10$, or $c = 8$?): compute the 2D and 2.5D communication volumes per processor (in MB, assuming 8 bytes per double), counting for 2.5D the replication of A and B and the reduction of C as well as the shifts. Would the 2.5D improvement be observable on a cluster with 10 Gbit/s inter-node links?

Chapter 13

07 svd jl dimsum

U7 — Dimensionality reduction & randomized numerical linear algebra

The ScaDaMaLe book, unit 7 (deep/optional; Module-3 group-project anchor). Our own exposition; original sources credited in §11.

Instantiating the three-lens invariant. The U0 box for dimensionality reduction:

> **RAM:** Classical SVD/PCA costs $O(mn^2)$ for an $m \times n$ tall-skinny matrix ($m \gg n$) — correct but prohibitive when m is the number of data points in a real corpus. The Eckart–Young theorem tells us the best rank-k approximation; the question is how to compute it cheaply. **PRAM:** Randomized SVD (Halko–Martinsson–Tropp 2011) uses a random range-finder to project the problem to a small $m \times (k+p)$ subspace, then does the expensive factorisation there. The algorithm is parallelisable; its error guarantee is stated and holds with high probability. **DPRAM/MPC:** A **sparse subspace embedding** (CountSketch, §5) sketches the whole column space in a *single input-sparsity-time pass*, and the distributed Gramian pipeline ($A^\top A$ via row outer-products, tree-aggregate, local SVD, broadcast back) reduces the tall matrix in a few shuffle rounds. The honest, general "cheap approximate solve" is the subspace embedding, with a worst-case guarantee over *all* A; the unit closes with DIMSUM (§7), a production similarity method whose "independent of m" shuffle holds under the conditions stated in §7.3.

Entry kit — what this chapter assumes; follow a link if you need the memory jog (map: Figure 1):
F.M2 (why the SVD exists; orthogonality) • F.S2 (concentration, for the random-projection guarantees)

1. Intuition — why high dimensions hurt, and three strategies

A dataset of m documents, each represented as a word-count vector in $\mathbb{R}^n$, lives in an ambient space of dimension n (the vocabulary size — easily 10^5 to 10^6). Yet the *intrinsic* structure of the corpus may live in a much lower-dimensional space: most documents cluster around a handful of topics; most columns of the data matrix are highly correlated.

Three problems, all hard at scale, all addressed in this unit:

- **Compress the data.** Replace the $m \times n$ matrix A by a product of smaller matrices that capture the essential structure — principal components or a truncated SVD. The RAM cost of exact SVD is $O(mn^2)$ (for $m \gg n$); for $m = 10^8$, $n = 10^5$, this is 10^{18} operations — impossible. Randomised algorithms bring this to $O(mnk)$ for a rank-k approximation.

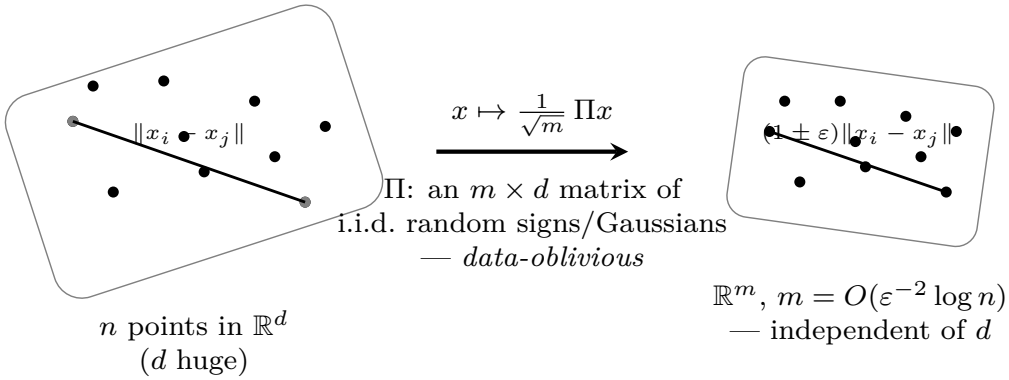


Figure 13.1: The Johnson–Lindenstrauss picture: a random matrix — chosen without ever looking at the data — maps n points from dimension d down to $m = O(\varepsilon^{-2} \log n)$ while preserving every pairwise distance to a factor $1 \pm \varepsilon$, with high probability (F.S2 concentration + a union bound over the $\binom{n}{2}$ pairs). The target dimension does not depend on d at all. Contrast with the SVD: JL preserves *distances obliviously*; the SVD finds the *best subspace adaptively*. U13's retrieval indexes and the embedding layers of U0L live in exactly this regime.

- **Preserve distances while reducing dimension.** Given N points in $\mathbb{R}^d$, find a map into $\mathbb{R}^k$ with $k \ll d$ that distorts pairwise distances by at most $(1 \pm \varepsilon)$. The Johnson–Lindenstrauss lemma says $k = O(\log N/\varepsilon^2)$ suffices — *independent of the original dimension* d. This is the theoretical backbone of modern sketching and locality-sensitive hashing.
- **Sketch a whole matrix, then solve on the small one.** For $A \in \mathbb{R}^{m \times d}$ ($m \gg d$), a **subspace embedding** S (§5) preserves every length in A's column space, so regression and low-rank problems can be solved on the small SA. The sparse CountSketch construction applies in $O(\text{nnz}(A))$ time — a single pass — with a worst-case guarantee over *all* A. (DIMSUM (§7) applies a related importance-sampling idea to similarity Gramians; its m-independence holds under the conditions we state there.)

All three strategies share a common algorithmic idea: **randomness breaks the dimension problem**, either by projecting to a random subspace (JL, randomised SVD) or by sketching the column space (subspace embeddings).

Figure 13.1 carries the unit's surprise: distances survive a random, data-blind projection.

2. RAM — classical SVD and the Eckart–Young theorem

2.1 The SVD

Every real matrix $A \in \mathbb{R}^{m \times n}$ (take $m \geq n$) has a **singular value decomposition**:

$$A = U \Sigma V^\top,$$

where $U \in \mathbb{R}^{m \times n}$ has orthonormal columns (the left singular vectors), $\Sigma = \text{diag}(\sigma_1, \ldots, \sigma_n)$ with $\sigma_1 \geq \sigma_2 \geq \cdots \geq \sigma_n \geq 0$ (the singular values), and $V \in \mathbb{R}^{n \times n}$ is orthogonal (the right singular vectors, i.e. the principal directions).

The **rank-k truncated SVD** retains only the top k singular triplets:

$$A_k = U_k \Sigma_k V_k^\top, \quad U_k \in \mathbb{R}^{m \times k},\ \Sigma_k \in \mathbb{R}^{k \times k},\ V_k \in \mathbb{R}^{n \times k}.$$

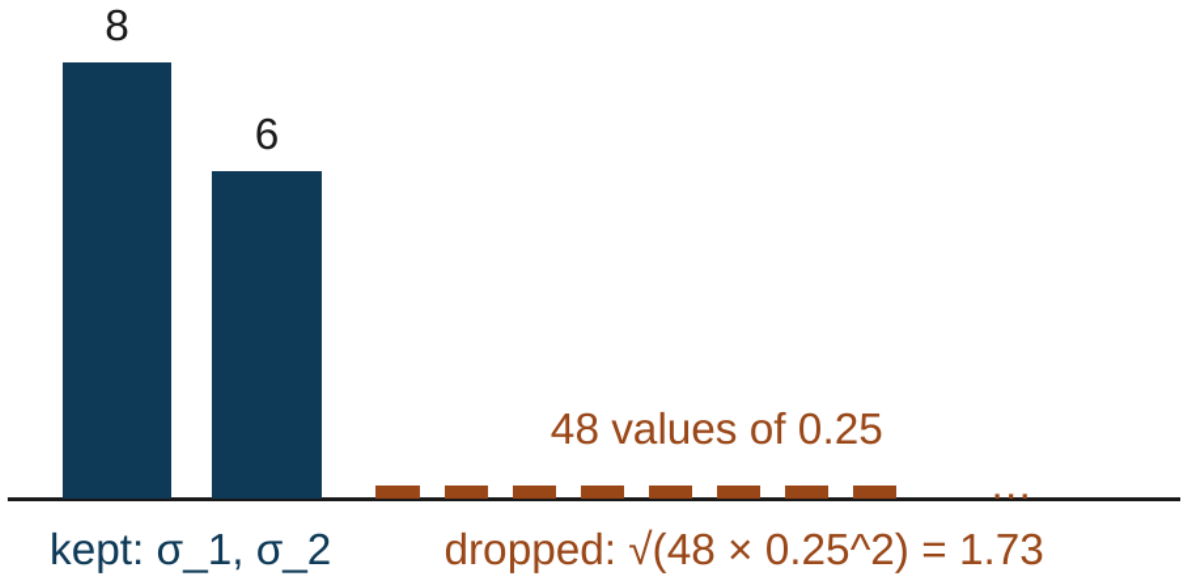


Figure 13.2: Eckart–Young: the best rank-k approximation is the truncated SVD.

2.2 Eckart–Young theorem (stated)

Theorem (Eckart–Young–Mirsky 1936). *For any matrix $A \in \mathbb{R}^{m\times n}$ and any $k \le \operatorname{rank}(A)$, the best rank-$k$ approximation in both the Frobenius norm and the operator (spectral) norm is the truncated SVD:*

$$\|A - A_k\|_F = \min_{\operatorname{rank}(B)\le k} \|A - B\|_F = \sqrt{\sigma_{k+1}^2 + \cdots + \sigma_n^2},$$

$$\|A - A_k\|_2 = \min_{\operatorname{rank}(B)\le k} \|A - B\|_2 = \sigma_{k+1}.$$

We state this without proof; the proof uses the Courant–Fischer min-max principle and is standard (see Golub–Van Loan Chapter 8). The theorem is the justification for everything that follows: A_k is the best low-rank approximation possible, and we want to compute it (or something close to it) cheaply.

PCA connection. Principal Component Analysis on a centred matrix A (rows = mean-subtracted data points) is exactly the truncated SVD: the top k right singular vectors V_k are the principal components (the directions of greatest variance); the scores $AV_k = U_k\Sigma_k$ are the projections of data points onto those components. SVD and PCA are two names for one computation.

Figure 13.2 draws the Eckart–Young theorem: we keep the top k singular triples and drop the rest.

2.3 RAM cost of exact SVD

For an $m \times n$ tall-skinny matrix ($m \ge n$):

- **Bidiagonalisation** (Golub–Kahan): $O(mn^2)$ operations, $O(mn)$ memory.
- **QR-based:** $O(mn^2)$ via a Householder QR followed by an $n \times n$ SVD.
- **Power method / Lanczos** for the top-k truncation: $O(mnk)$ per power iteration step; convergence depends on the spectral gap σ_k/σ_{k+1}.

For $m \gg n$, the bottleneck is the $O(mn^2)$ Gram computation or QR sweep. This is the cost the randomised approach will circumvent.

3. Johnson–Lindenstrauss lemma

3.1 The lemma (stated precisely)

Lemma (Johnson–Lindenstrauss 1984). *For any $\varepsilon \in (0, 1/2)$, any set of N points $x_1, \dots, x_N \in \mathbb{R}^d$, and any*

$$k \;\geq\; \frac{4 \ln N}{\varepsilon^2/2 - \varepsilon^3/3} \;=\; O\!\left(\frac{\log N}{\varepsilon^2}\right),$$

there exists a map $f : \mathbb{R}^d \to \mathbb{R}^k$ such that for all $i \neq j$:

$$(1-\varepsilon)\|x_i - x_j\|^2 \;\leq\; \|f(x_i) - f(x_j)\|^2 \;\leq\; (1+\varepsilon)\|x_i - x_j\|^2.$$

Moreover, a random linear map $f(x) = \frac{1}{\sqrt{k}} Rx$ where $R \in \mathbb{R}^{k \times d}$ has i.i.d. $\mathcal{N}(0,1)$ entries achieves this with probability at least $1/N$ (over the randomness of R, simultaneously for all $\binom{N}{2}$ pairs): each pair fails with probability at most $2/N^2$, and the union bound leaves $1 - \binom{N}{2} \cdot 2/N^2 = 1/N$. Replacing the constant 4 by 8 makes each pair fail with probability at most $2/N^4$, and the success probability rises to at least $1 - 1/N^2$.

Key points: - k depends on $\log N$ and ε, **not on** d. The target dimension is set by the *number of points* and the distortion you tolerate, never by the ambient dimension — so a dataset in millions of dimensions projects to the *same* k as one in thousands, given the same N and ε. (For a concrete calculation see E7.1; the sketch size is independent of d.) - The map is linear and oblivious to the data (just fix R before seeing any point). - The bound is tight up to constants: a matching lower bound is known (Larsen–Nelson 2016 for linear maps; Larsen–Nelson 2017 for arbitrary maps) — there are N-point sets that no linear map embeds with $(1 \pm \varepsilon)$ distortion in fewer than $\Omega(\log N/\varepsilon^2)$ dimensions, for ε not too small (any N points embed isometrically in $N-1$ dimensions, so the bound cannot exceed that). Alon's earlier $\Omega(\log N/(\varepsilon^2 \log(1/\varepsilon)))$ came within a $\log(1/\varepsilon)$ factor (see Dasgupta–Gupta 2003, §1).

3.2 Proof idea (one-paragraph sketch)

The argument has three steps, each classical.

Step 1 — single pair, sub-Gaussian concentration. Fix two points x, y; let $u = x - y$. The projection $f(x) - f(y) = \frac{1}{\sqrt{k}} Ru$ is a scaled Gaussian vector; its squared norm $\frac{1}{k}\|Ru\|^2 = \frac{\|u\|^2}{k} \sum_{i=1}^{k} Z_i^2$ where $Z_i = R_i u/\|u\| \sim \mathcal{N}(0,1)$ i.i.d. The normalised sum $\frac{1}{k}\sum Z_i^2$ is $\chi^2(k)/k$ with mean 1. Applying the **sub-Gaussian (Bernstein/Hanson–Wright) concentration inequality** to the sum of squares gives $P\big(|\, \|Ru\|^2/\|u\|^2 - k \,| > \varepsilon k\big) \leq 2\exp(-c\varepsilon^2 k)$ for a constant $c > 0$. Setting this probability below $1/N^3$ requires $k = \Omega(\log N/\varepsilon^2)$.

Step 2 — union bound over all pairs. There are $\binom{N}{2} < N^2/2$ pairs. By the union bound, the probability that *any* pair is distorted by more than ε is at most $\binom{N}{2} \cdot 2\exp(-c\varepsilon^2 k) < N^2 \exp(-c\varepsilon^2 k)$. Setting $k = (2\ln N + \ln 2)/(c\varepsilon^2)$ makes this probability less than $1/2$, so the map exists, and a random Gaussian matrix achieves it with probability more than $1/2$.

Step 3 — the map is cheap to apply. Multiplying by R costs $O(dk)$ per point, so projecting N points costs $O(Ndk)$ total — linear in N and d, with a small constant k. This is the computational win.

3.3 Database-friendly variants (stated)

The Gaussian random matrix works, but generating and storing $k \times d$ Gaussian entries is expensive for large d. Two practically important variants:

Achlioptas (2003). Replace each entry of R by $\sqrt{3}$ times an independent draw from a sparse distribution: $+1$ with probability $1/6$, -1 with probability $1/6$, and 0 with probability $2/3$. (The factor $\sqrt{3}$ restores unit variance: without it each entry has variance $1/3$, and $\mathbf{E}\|f(x)\|^2 = \|x\|^2/3$.) **Stated result:** the JL distortion bound holds with the same asymptotic $k = O(\log N/\varepsilon^2)$. Saving: two-thirds of entries are zero — the matrix-vector product is $3\times$ faster; the non-zero entries are $\pm\sqrt{3}$, so once the single factor $\sqrt{3/k}$ is pulled out of the product only additions and subtractions remain — no floating-point multiplication needed. *We state this without proof; the argument extends the sub-Gaussian concentration via the moment method (Achlioptas 2003, Theorem 1.1).*

Very sparse JL (Kane–Nelson 2014). Even sparser — $O(\varepsilon^{-1}\log N)$ non-zeros per column, a fraction $O(\varepsilon)$ of the $k = O(\varepsilon^{-2}\log N)$ rows — with the same guarantees. Useful when d is very large and the data vectors themselves are sparse (e.g. TF-IDF).

Fast JL (Ailon–Chazelle 2006). Preprocess with a random Hadamard rotation (fast Walsh–Hadamard transform, $O(d\log d)$) then sub-sample. The combined map can be applied in $O(d\log d + k^2\log(1/\delta))$ time per point. Useful when d is large and k is small.

4. Randomised SVD (Halko–Martinsson–Tropp)

4.1 The algorithm

The key insight of Halko, Martinsson, and Tropp (SIREV 2011) is to first *sketch* the column space of A using a random projection, then do the expensive factorisation on the sketch. The input is $A \in \mathbb{R}^{m\times n}$, target rank k, and oversampling parameter p (typically $p = 5$ to 10).

```
Randomised-SVD(A, k, p, q):
  1. Draw Omega in R^{n x (k+p)} with i.i.d. N(0,1) entries.   // random test
    matrix
  2. Form Y = A Omega.                                          // Y in R^{m x
    (k+p)}
  3. [Optional] Power iteration: for j = 1..q:
        Y = A (A^T Y)                                          // amplify top
          singular subspace
  4. Compute an orthonormal basis Q for the columns of Y:
        Q, _ = QR(Y)                                          // Q in R^{m x
          (k+p)}
  5. Form B = Q^T A.                                           // B in R^{(k+p)
    x n}: SMALL
  6. Compute SVD of B: B = U~ Sigma V^T.                       // (k+p)x(k+p)
    times n
  7. Return U = Q U~_k, Sigma_k, V_k^T.                        // truncate to
    rank k
```

4.2 Cost accounting

- **Step 1:** generating Ω is $O(n(k+p))$.
- **Step 2:** the matrix–matrix product $A\Omega$ costs $O(mn(k+p))$ — this is the dominant step. For $k+p \ll n$, this is much cheaper than the $O(mn^2)$ of full SVD.
- **Step 3 (power iterations):** each iteration costs an additional $O(mn(k+p))$; q iterations cost $O(qmn(k+p))$ total.
- **Step 5:** $B = Q^\top A$ is a second pass over A, costing $O(mn(k+p))$ — the same order as step 2.
- **Steps 4, 6, 7:** $O(m(k+p)^2 + n(k+p)^2)$ — small, since $k+p \ll m, n$.

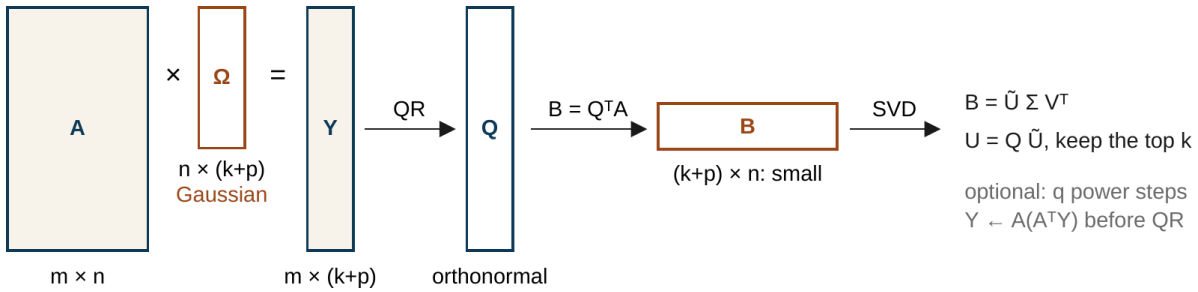


Figure 13.3: Randomised SVD: we sketch the range, then factor a small matrix.

Total: $O((1+q)mn(k+p))$. For $q=0$ and $k+p=O(k)$: $O(mnk)$. Compare with $O(mn^2)$ for exact SVD: a factor of n/k speedup when $k \ll n$.

4.3 Error guarantee (stated)

Theorem (Halko–Martinsson–Tropp 2011, Theorem 10.5). *Let $A \in \mathbb{R}^{m\times n}$, choose a target rank $k \geq 2$ and oversampling $p \geq 2$ with $k+p \leq \min\{m,n\}$, and let Q be the orthonormal basis that steps 1–4 return with no power iterations ($q=0$). Then:*

$$\mathbf{E}\| A - QQ^\top A\|_F \;\leq\; \left(1+\frac{k}{p-1}\right)^{1/2} \cdot \left(\sum_{j>k}\sigma_j^2\right)^{1/2},$$

where the expectation is over the random matrix Ω. Here $QQ^\top A = QB$ is the approximation of step 5, before the truncation to rank k. In words, the range finder pays at most the factor $(1+k/(p-1))^{1/2}$ over the Eckart–Young optimum $\|A-A_k\|_F$; with $p=k$ that factor is at most $\sqrt{3}$, before any power iteration. For $q \geq 1$ power iterations, HMT's Corollary 10.10 bounds the spectral-norm error: it applies their spectral bound (Theorem 10.6) to $(AA^\top)^q A$, whose singular values are σ_j^{2q+1}, and takes the $(2q+1)$-th root, which drives the extra factor over the optimum σ_{k+1} towards one as q grows (the constant enters only through its $(2q+1)$-th root, so the excess shrinks like $1/q$).

We state this without proof; the proof uses Johnson–Lindenstrauss-style concentration of the random range finder and the Eckart–Young theorem to bound the residual. See HMT 2011 §10 for the full argument. The practical message: one or two power iterations ($q=1,2$) typically suffice to make the approximation accurate; more iterations help when the singular values decay slowly.

Figure 13.3 shows the whole pipeline in one picture.

5. Sparse subspace embeddings — sketch-and-solve (CountSketch)

JL (§3) shrinks a *set of points*; the next gem shrinks a whole *matrix's column space* at once, cheaply enough to run on the biggest data — a general "approximate a tall computation with one cheap pass" with a worst-case guarantee, complementing the similarity-Gramian setting of §7.

5.1 The object: a subspace embedding

Definition (ℓ_2-subspace embedding). A matrix $S \in \mathbb{R}^{t\times m}$ is a $(1\pm\varepsilon)$ **subspace embedding** for the column space of $A \in \mathbb{R}^{m\times d}$ ($m \gg d$) if

$$(1-\varepsilon)\,\|Ax\|_2 \;\le\; \|SAx\|_2 \;\le\; (1+\varepsilon)\,\|Ax\|_2 \qquad \text{for all } x \in \mathbb{R}^d.$$

Then the *small* matrix $SA \in \mathbb{R}^{t\times d}$ preserves every length in A's column space to within ε: any least-squares or low-rank problem may be solved on SA instead of the tall A. This is JL promoted from finitely many points to an entire d-dimensional subspace (a net argument over the sphere lifts the pairwise guarantee — Woodruff survey, Def. 1–4).

5.2 The gem: CountSketch in input-sparsity time

The landmark *sparse* construction is **CountSketch** (Clarkson–Woodruff 2013): S has exactly **one non-zero per column**, a random ±1 in a random row. Applying S to A is then a single streaming pass that hashes each non-zero entry of A into a bucket — cost $O(\mathrm{nnz}(A))$, *input-sparsity time*, the property that makes it scale.

Theorem (Clarkson–Woodruff 2013 — CountSketch subspace embedding). *With sketch dimension*

$$t \;=\; \tilde{O}\!\left(\frac{d^2}{\varepsilon^2}\right),$$

a CountSketch S is a $(1\pm\varepsilon)$ subspace embedding for any $A \in \mathbb{R}^{m\times d}$ with constant probability, and SA is computed in $O(\mathrm{nnz}(A))$ time. (The $\tilde{O}$ hides logarithmic factors; later work sharpens the d^2; the constant and the success-probability boosting are as in the source — see the certificate/deep-dive pointer.)

5.3 Sketch-and-solve

The payoff is **sketch-and-solve** for regression $\min_x\|Ax-b\|_2$: sketch $[A\,|\,b]$ to $[SA\,|\,Sb]$, solve the *small* problem $\min_x\|SAx-Sb\|_2$, and return its solution $\tilde{x}$. Because S preserves lengths in the relevant subspace,

$$\|A\tilde{x}-b\|_2 \;\le\; (1+\varepsilon)\,\|Ax^*-b\|_2,$$

in total time $O(\mathrm{nnz}(A))+\tilde{O}(d^3/\varepsilon^2)$ — the tall dimension m is touched only by the one sparse pass. `MATHS->CS:` this is a worst-case-guaranteed "cheap approximate solve" — general over *all* A, a complement to the similarity-Gramian method of §7. It also feeds U8: sketch-and-solve is one of the two RNLA regression routes there (the other, leverage-score *sampling*, is its sample-based twin).

6. Distributed tall-skinny SVD — the $A^\top A$ Gramian pipeline

When $A \in \mathbb{R}^{m\times n}$ is tall-skinny ($m \gg n$) and the rows of A are distributed across M machines (one or more rows per machine), the Gramian $G = A^\top A \in \mathbb{R}^{n\times n}$ is small (only n^2 entries) and contains the full spectral information needed for PCA.

6.1 The pipeline (G11 lab fold)

```
Distributed-SVD via A^T A:

  Map:   machine i owns rows A_i in R^{m_i x n}.
         Emit (key=*, value=A_i^T A_i)          // local n x n outer product
           sum; O(m_i n^2) work
```

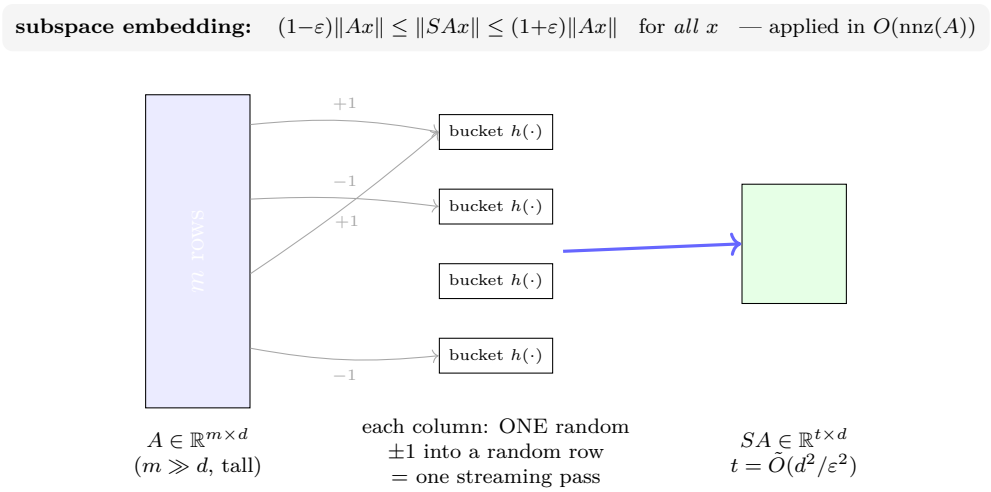


Figure 13.4: CountSketch, the sparse subspace embedding. Each column of the tall A contributes a single ± 1 to one random bucket (row) of the sketch — so forming SA is one streaming pass over the non-zeros, $O(\mathrm{nnz}(A))$ time. The short SA preserves *every* length in A's column space to within ε (a subspace embedding), so any regression or low-rank problem may be solved on it instead of on the tall A. This is the honest, worst-case-guaranteed "sketch a big computation with one cheap pass" — general over all A.

```
Tree-aggregate:
       Gather local Gramians; add them up:
       G = sum_i A_i^T A_i = A^T A              // tree-reduce of n x n
         matrices; O(Mn^2) comm in total, O(n^2 log M) on any root path

One machine: SVD of G = V D V^T                 // n x n SVD; O(n^3) work (small)

Broadcast: V, Sigma = sqrt(D)                  // broadcast n x n matrix; O(Mn^2)
  comm

Map: machine i computes U_i = A_i V Sigma^{-1}  // local U slices; O(m_i n k)
  work
```

6.2 Cost accounting

- **Local map work:** machine i computes $A_i^\top A_i$ — an $n \times n$ matrix product costing $O(m_i n^2)$ — and sends n^2 values. Total map work: $O(mn^2)$ (same as sequential Gram computation, shared across M machines: $O(mn^2/M)$ per machine).
- **Communication (shuffle):** each machine sends $O(n^2)$ values; total $O(Mn^2)$ bytes.
- **Tree-aggregate:** a binary tree has $\lceil \log_2 M \rceil$ levels; its $M-1$ additions of $n \times n$ matrices cost $(M-1)n^2 = O(Mn^2)$ additions in total, but only $O(n^2 \log M)$ along any root path (the depth). A tree of fan-in f has $\lceil \log_f M \rceil$ levels, which is $O(1)$ when each machine can hold $f = M^\delta$ Gramians ($\delta > 0$ fixed).
- **Broadcast V, Σ:** $O(n^2)$ bytes to each of the M machines — $O(Mn^2)$ bytes in total either way; a tree-broadcast cuts what any one machine sends from $O(Mn^2)$ to $O(n^2 \log M)$.
- **Local U computation:** each machine multiplies its rows by $V\Sigma^{-1}$: $O(m_i nk)$ per machine, $O(mnk)$ total.

Round count: two passes (map → aggregate → broadcast → local U): $R = O(1)$ rounds in MPC with a constant-depth aggregation tree (fan-in M^δ); a binary tree adds $O(\log M)$ levels.

Bottleneck: the shuffle of Gramian pieces costs $O(Mn^2)$ bytes — independent of m. This pipeline is efficient when $n^2 \ll m$ (tall-skinny), which is the typical PCA regime.

7. Case study — DIMSUM: importance sampling meets the shuffle cost model (Zadeh–Carlsson 2013)

DIMSUM is a production-motivated, importance-sampled estimator of the column-similarity (Gramian) matrix. We teach it for its ***shuffle / reduce-key cost model*** *(§7.4): a clean, real-world illustration of how the communication cost of the Gramian can be made independent of the tall dimension m.*

> **The conditions the guarantee uses (§7.3).** So the algorithm is applied within its regime, we state its assumptions plainly: - the sampling probability is **coupled to the column magnitudes** (with a magnitude-independent emission the expected shuffle instead grows with m); - the per-entry approximation is a probabilistic guarantee; - the rescaling step assumes a bounded dynamic range for the entries. These are the ordinary small print of an at-scale result; §7.3 makes them explicit, and the CountSketch subspace embedding of §5 is a complementary tool with a worst-case guarantee.

7.1 The problem

Let $A = [c_1 \mid c_2 \mid \cdots \mid c_n] \in \mathbb{R}^{m \times n}$ be a matrix of column vectors (e.g. m users, n items; or m documents, n terms — but now transposed from the previous sections to match the DIMSUM convention). We want to compute the **cosine similarity** between every pair of columns:

$$\mathrm{sim}(i,j) = \frac{c_i^\top c_j}{\|c_i\|\|c_j\|} = \frac{(A^\top A)_{ij}}{\|c_i\|\|c_j\|}.$$

Equivalently, we need the **similarity Gramian** $S \in \mathbb{R}^{n \times n}$ with $S_{ij} = \mathrm{sim}(i,j)$.

The exact Gramian requires computing all $\binom{n}{2}$ pairwise inner products. Naively in MapReduce: each row r (of length n) emits all $\binom{n}{2}$ pairs $(i, j, r_i r_j)$, giving a shuffle size of $O(mn^2)$ key-value pairs — proportional to m, the tall dimension. For $m = 10^9$, $n = 10^4$, this is 10^{17} pairs: completely impractical.

The question DIMSUM asks: can we approximate S to within $(1 \pm \varepsilon)$ per entry with a shuffle size that does not grow with m?

7.2 The algorithm — importance sampling on rows

The idea. Row r of A contributes $A_{ri} \cdot A_{rj}$ to the inner product $c_i^\top c_j$. Rows with small $\|A_{r\cdot}\|_\infty$ contribute little; rows with large entries in columns i and j are the dominant contributors. DIMSUM samples each row's pair (i,j) with a probability that captures this importance.

Algorithm (DIMSUM; Zadeh–Carlsson 2013). Fix a parameter $\gamma > 0$ (to be chosen).

```
DIMSUM(A, gamma):
  For each row r = 1..m:
    For each non-zero pair (i, j) in row r (i.e. A_{ri} != 0, A_{rj} != 0, i <
      j):
      p_ij := min(1, gamma / (||c_i|| * ||c_j||))    // importance-sampling
        probability
      With probability p_ij:
        Emit (key=(i,j), value= A_{ri} * A_{rj} / p_ij)   // unbiased estimator
```

```
For each key (i,j):
  S_hat_{ij} = sum of received values / (||c_i|| * ||c_j||)
```

The key insight: the sampling probability *falls* as the column norms grow. Pairs with small norms, $\|c_i\|\|c_j\| \le \gamma$, are emitted with probability 1 — they have few contributing rows, so every contribution counts — while pairs with large norms are sampled *less* often (probability $\gamma/(\|c_i\|\|c_j\|) \to 0$ as the denominator grows relative to γ): with entries in $[0,1]$ a large norm means many contributing rows, and a sample of them already pins down the cosine. The division by p_{ij} makes each emission an unbiased estimator of the true contribution.

7.3 The theorems (stated, with their conditions)

Let us scale A so that its entries lie in $[0,1]$ (one pass finds the largest magnitude), and let L denote the largest number of non-zeros in any row and H the smallest non-zero entry after scaling ($H = 1$ for 0/1 data such as "term occurs in document"). Write $\hat{S}_{ij}$ for the reducer's output, so that $\|c_i\|\|c_j\|\,\hat{S}_{ij}$ estimates $(A^\top A)_{ij}$. The DIMSUM paper (Bosagh Zadeh and Carlsson, arXiv:1304.1467) proves three statements; we state them without proof, each with the price it charges.

Theorem (shuffle size). *The expected number of emitted key-value pairs is $O(nL\gamma/H^2)$, and, for non-negative A, each key receives at most γ/H^2 values.* Neither bound contains m. E7.7 asks you to derive the first.

Theorem (similar pairs). *Let $s > 0$ be a similarity threshold, $\delta \in (0,1)$ a relative error and $\alpha > 0$, and run DIMSUM with $\gamma \ge \alpha/s$. For any two columns with $\mathrm{sim}(i,j) \ge s$, the estimate $\|c_i\|\|c_j\|\,\hat{S}_{ij}$ lies outside $(1 \pm \delta)\,(A^\top A)_{ij}$ with probability at most*

$$\left(\frac{e^{\delta}}{(1+\delta)^{1+\delta}}\right)^{\alpha} + e^{-\alpha\delta^2/2}.$$

In words, the sampled contributions to a similar pair are a sum of independent terms in $[0,1]$ whose mean is $\gamma\,\mathrm{sim}(i,j) \ge \alpha$, so the multiplicative Chernoff bound pins the estimate within $(1 \pm \delta)$ unless an exponentially unlikely event occurs. Note that the paper states this pair by pair. The step to *every* similar pair at once is ours: both terms are at most $e^{-\alpha\delta^2/3}$, so $\alpha = 9\ln n/\delta^2$ makes each pair fail with probability at most $2n^{-3}$, and a union bound over the fewer than $n^2/2$ pairs leaves a total failure probability of at most $1/n$. The price is $\gamma = 9\ln n/(\delta^2 s)$, hence a shuffle of $O(nL\log n/(\delta^2 s H^2))$.

Theorem (the whole Gramian). *If $m > n$ and $\gamma = \Omega(n/\varepsilon^2)$, then with $D = \mathrm{diag}(\|c_1\|,\ldots,\|c_n\|)$ and $\hat{S} = (\hat{S}_{ij})$,*

$$\|D\hat{S}D - A^\top A\|_2 \;\le\; \varepsilon\,\|A^\top A\|_2$$

with probability at least $1/2$. The proof bounds the second and fourth moments of each entry and applies Latała's bound on the spectral norm of a random matrix, then Markov's inequality. Thus the singular values of A are preserved to relative error ε, at a shuffle of $O(n^2 L/(\varepsilon^2 H^2))$.

Observe the two prices. Accurate *similar pairs* need γ to grow like $\log n/s$; the whole spectrum needs γ to grow like n/ε^2, so for comparable accuracy a factor of order $n/\log n$ more shuffle. However, both are free of m, which is the point.

Figure 13.5 sets the exact shuffle against DIMSUM's bound from 10^9 documents up: the exact cost grows with m, DIMSUM's does not.

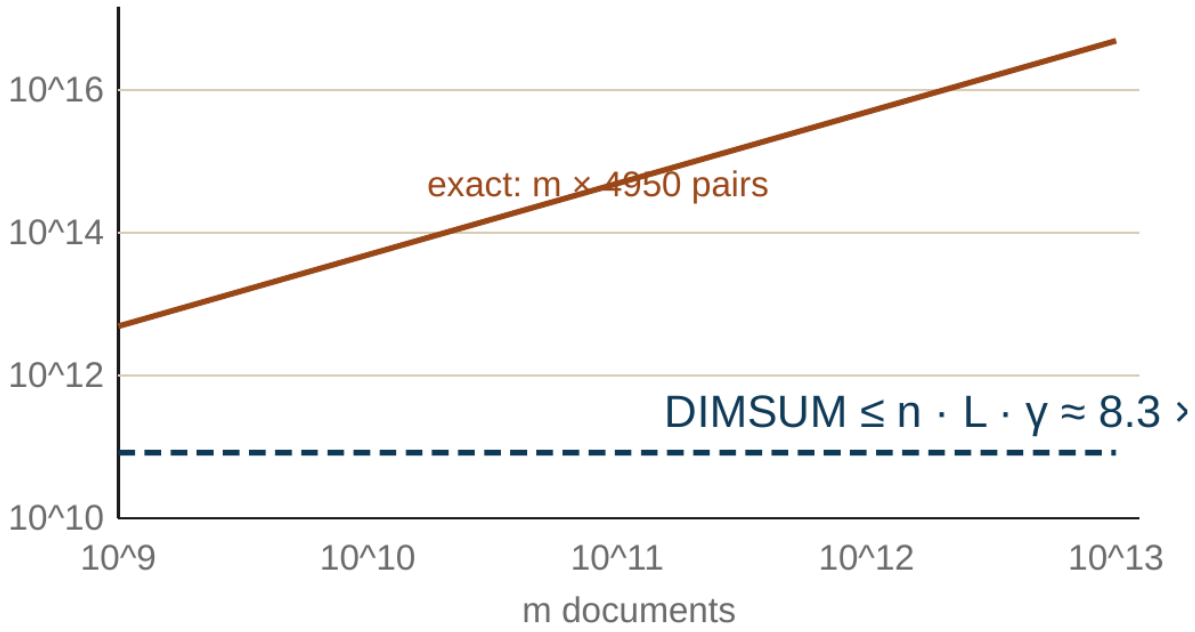


Figure 13.5: The exact shuffle against DIMSUM's bound, from $m = 10^9$ documents up.

7.4 Why this matters — the shuffle-size lens (the real lesson)

Recall from U0 and U5 that in the DPRAM/MPC model, **the shuffle is the bottleneck**. DIMSUM's theorem states that:

- **Exact Gramian:** shuffle size $O(mL^2)$, at worst $O(mn^2)$ — grows with m; impractical for large corpora.
- **DIMSUM:** shuffle size $O(nL \log n/(\delta^2 s H^2))$ for every pair above similarity s, and $O(n^2 L/(\varepsilon^2 H^2))$ for the whole spectrum — **no** m; the tall dimension has been eliminated from the communication cost.

A worked instance with real magnitudes. Take a corpus of $m = 10^9$ short documents over $n = 10^4$ terms, each document using at most $L = 100$ terms, with 0/1 entries ($H = 1$). If every document has L terms, the exact shuffle is $m\binom{L}{2} \approx 10^9 \times 4950 \approx 5 \times 10^{12}$ pairs. For pairs above similarity $s = 0.1$ to relative error $\delta = 0.1$, §7.3 asks for $\alpha = 9 \ln(10^4)/0.01 \approx 8.3 \times 10^3$ and $\gamma = \alpha/s \approx 8.3 \times 10^4$, so DIMSUM emits at most about $nL\gamma \approx 10^4 \times 100 \times 8.3 \times 10^4 \approx 8.3 \times 10^{10}$ pairs, some 60 times fewer. Note that this is a worst-case bound and the saving is modest at this m. However, the bound does not move when m grows: at $m = 10^{13}$ documents (the scale the DIMSUM paper has in mind) the exact shuffle is 5×10^{16} pairs and the saving is about 6×10^5.

This is the signature of a truly scalable distributed algorithm: the communication cost does not degrade as more data is added; only local work grows with m (each row still scans its non-zeros).

DIMSUM in Spark. `org.apache.spark.mllib.linalg.distributed.RowMatrix.columnSimilarities(threshold)` implements DIMSUM directly. The `threshold` s is a similarity level (similarities above s are estimated accurately), and Spark turns it into the oversampling parameter $\gamma = 10 \ln n/s$ (that is, $\alpha = 10 \ln n$ in §7.3): the smaller the threshold, the larger γ. Spark samples per *column* rather than per pair, the paper's "LeanDIMSUM" variant: in each row it keeps the entry of column i with probability $\min(1, \sqrt{\gamma}/\|c_i\|)$, independently, and emits a product for every pair of kept entries. A pair is therefore emitted with probability $\min(1, \sqrt{\gamma}/\|c_i\|)\min(1, \sqrt{\gamma}/\|c_j\|)$, which equals $\gamma/(\|c_i\|\|c_j\|)$ when both column norms exceed $\sqrt{\gamma}$. This saves random draws, and the per-pair guarantee survives it; however, the estimates of different pairs are no longer independent, so the whole-Gramian theorem does not carry over.

Setting `threshold = 0` gives exact computation; positive thresholds enable the importance-sampling speedup.

7.5 Algorithm provenance

DIMSUM was introduced by Reza Zadeh and Gunnar Carlsson at Stanford/Twitter (2013). The algorithm was used in production at Twitter for computing item-to-item similarity at scale in recommendation systems: every tweet, user, or content item is a column vector; the matrix has m = billions of interactions (users/events) and n = millions of items. The key challenge in that setting is exactly the one DIMSUM solves: the shuffle cost of the Gramian computation must not scale with the number of interactions. The Spark implementation in `RowMatrix.columnSimilarities` is the direct descendent of this work and is one of the reasons DIMSUM is treated as a first-class algorithm in this course. We follow the exposition of Zadeh–Carlsson 2013 in our focused treatment.

8. Invariant (U7 instantiation)

RAM: Exact SVD costs $O(mn^2)$ for an $m \times n$ tall-skinny matrix — the **Eckart–Young theorem** (stated) gives the best rank-k approximation; JL (stated + proof sketch) reduces N points in $\mathbb{R}^d$ to $\mathbb{R}^{O(\log N/\varepsilon^2)}$ with $(1 \pm \varepsilon)$ distance preservation.

PRAM: Randomised SVD (HMT 2011; stated) uses a random range-finder + optional power iterations to achieve $O(mnk)$ cost and a stated near-optimal error guarantee.

DPRAM/MPC: Distributed Gramian pipeline (map row outer-products → tree-aggregate $n \times n$ matrix → local SVD → broadcast → local U slices) uses $O(1)$ rounds (with a constant-depth aggregation tree; $O(\log M)$ with a binary one) and $O(Mn^2)$ shuffle. **DIMSUM** (Bosagh Zadeh–Carlsson; stated theorems) estimates every cosine similarity above a threshold s to relative error $(1 \pm \delta)$ with shuffle size $O(nL \log n/(\delta^2 sH^2))$, and the whole Gramian to spectral error ε with $O(n^2 L/(\varepsilon^2 H^2))$ — **independent of the tall dimension** m; Spark `RowMatrix.columnSimilarities` is the production anchor.

9. Lab — project track

Project-track lab (no weekly receipt): run these experiments free and locally in the course lab image (`labs/image/`) — or any Spark-in-Docker environment — as deep-dive and group-project material. Folds G11 and G12 from the W3-v2 sweep.

Experiment 1 — Distributed SVD via $A^\top A$ (G11 fold).

Generate a synthetic $m \times n$ tall-skinny RDD (e.g. $m = 10^6$, $n = 500$) with known rank-k structure ($k = 20$) plus noise. Use `RowMatrix.computeGramianMatrix()` to form the Gramian $A^\top A$ (not `computeCovariance()`, which centres the columns and divides by $m - 1$); compute its full SVD locally; compare the top-k singular values and right singular vectors against ground truth. Plot reconstruction error $\|A - \hat{A}_k\|_F / \|A\|_F$ vs k. Count Spark stages and attribute wall-clock time to: local map work, shuffle, and local SVD.

Experiment 2 — DIMSUM error vs γ sweep (G12 fold).

Construct an $m \times n$ matrix ($m = 5 \times 10^5$, $n = 200$) with a known cosine similarity structure (e.g. column pairs drawn from a mixture model with known pairwise cosines). Run `RowMatrix.columnSimilarities(threshold)` for thresholds $s \in \{0.4, 0.2, 0.1, 0.05, 0.02\}$, so that

Spark's oversampling $\gamma = 10 \ln n/s$ (§7.4) grows as s shrinks. For each setting, compute: (a) the maximum relative error $\max |\hat{S}_{ij} - S_{ij}|/S_{ij}$ over the pairs with $S_{ij} \geq s$, and, for contrast, the maximum absolute error over the pairs below s; (b) shuffle size (count emitted pairs via Spark accumulator); (c) wall-clock time. Plot (a) and (b) vs γ. Does the shuffle stay below the $O(nL\gamma/H^2)$ bound of §7.3? Is the relative error above the threshold small at every s, while the pairs below it are left rough, as the per-pair theorem promises?

Deliverable: notebook with both experiments, plots, and a cost-accounting paragraph in the three-lens vocabulary, explicitly stating what the shuffle bound buys in practice.

10. Lean certificate target

Target (new — U7): the Johnson–Lindenstrauss bound is a probabilistic statement about a random projection; a Lean certificate captures the deterministic algebraic core.

```
-- SCHEMATIC target (names align to BMLSPar.lean conventions at cert time):
-- For a fixed pair of unit vectors u, v in R^d and a k x d Gaussian random
  matrix R,
-- the squared norm of (1/sqrt(k)) R (u - v) is concentrated around ||u - v||^2.
-- The cert formalises the variance computation:

-- jl_single_pair (k d : ℕ) (eps : ℝ) (heps : 0 < eps ∧ eps < 1) :
--   P(|(1/sqrt(k)) R u||^2 - ||u||^2| > eps * ||u||^2) ≤ 2 * exp(-c * eps^2 *
  k) := by
--   sorry  -- beyond the labs (U7.L1 is the variance identity, U7.L2 the
  Chebyshev bound)
```

Exercise U7.L1 (Lean, light). In the BMLSPar Lean environment, prove the variance identity: if $Z_1, \ldots, Z_k \sim \mathcal{N}(0,1)$ i.i.d., then $\mathrm{Var}\left(\frac{1}{k}\sum_{i=1}^{k} Z_i^2\right) = \frac{2}{k}$. Write this as a `calc` block over reals. (This is the variance computation that feeds the Chebyshev step in the JL proof.)

Exercise U7.L2 (deep). State and prove in Lean that for $Z \sim \mathcal{N}(0,1)$, $E[Z^4] = 3$. Combine with U7.L1 to give a Chebyshev-based JL bound (far weaker than the sub-Gaussian argument — it needs k polynomial in N, where the sub-Gaussian bound needs only $O(\log N/\varepsilon^2)$ — but provable with standard mathlib tools).

LLM lens — where this unit lands on the running workload (U0L):
LoRA — the standard cheap fine-tune — is Eckart–Young in production: adapt a weight matrix by a learned low-rank update because the best small correction is a low-rank one. Embedding matrices feed retrieval, and random projections compress exactly the vectors RAG systems index (U13's lens continues this).

11. Sources (credited; archived under `scadamale-book-sources`)

Johnson & Lindenstrauss (1984). W. B. Johnson and J. Lindenstrauss, *Extensions of Lipschitz mappings into a Hilbert space*, Contemporary Mathematics 26:189–206, 1984. — The original lemma; our §3 follows the standard random-projection proof (Dasgupta–Gupta 2003 presentation).

Achlioptas (2003). D. Achlioptas, *Database-friendly random projections: Johnson–Lindenstrauss with binary coins*, Journal of Computer and System Sciences 66(4):671–687, 2003. — The sparse $\sqrt{3} \cdot \{0, \pm 1\}$ variant; we state Theorem 1.1.

Larsen & Nelson (2016, 2017). K. G. Larsen and J. Nelson, *The Johnson–Lindenstrauss lemma is optimal for linear dimensionality reduction*, ICALP 2016; and *Optimality of the*

Johnson–Lindenstrauss lemma, FOCS 2017. — The matching lower bounds cited in §3 (linear maps; arbitrary maps).

Eckart & Young (1936). C. Eckart and G. Young, *The approximation of one matrix by another of lower rank*, Psychometrika 1(3):211–218, 1936. — The best low-rank approximation theorem; we state this classical result.

Halko, Martinsson & Tropp (2011). N. Halko, P.-G. Martinsson, J. A. Tropp, *Finding structure with randomness: Probabilistic algorithms for constructing approximate matrix decompositions*, SIAM Review 53(2):217–288, 2011. — The randomised SVD algorithm and error bounds; we follow their §9–10 notation; Theorem 10.5 is stated above, and Corollary 10.10 is cited for the power scheme.

Bosagh Zadeh & Carlsson (2013). R. Bosagh Zadeh and G. Carlsson, *Dimension Independent Matrix Square using MapReduce (DIMSUM)*, arXiv:1304.1467 (first posted 2013; v4, 2016). — The DIMSUM algorithm and its three theorems (shuffle size, similar pairs, the whole Gramian), stated in §7.3; the union-bound step to all similar pairs at once is ours, as is the exposition.

Wikipedia. *Singular value decomposition*, *Principal component analysis*, *Johnson– Lindenstrauss lemma*, *Random projection*, *Locality-sensitive hashing*, *HyperLogLog*. — Background and cross-references.

12. Exercises

Light (Module-3 / group-project anchor):

E7.1 *(JL k computation.)* You have $N = 10^6$ documents in $\mathbb{R}^{50000}$. (a) Using the JL lemma of §3.1 — the tight Dasgupta–Gupta bound $k \geq \lceil 4 \ln N/(\varepsilon^2/2 - \varepsilon^3/3) \rceil$ — with $\varepsilon = 0.1$, compute the smallest k the lemma allows for a map preserving all pairwise distances to within 10% (the random Gaussian map then succeeds with probability at least $1/N$; see §3.1). (Sanity check against the leading-order form $8 \ln N/\varepsilon^2$, which drops the $\varepsilon^3/3$: the two agree to leading order — the leading-order form is smaller by the factor $1 - 2\varepsilon/3$, so it slightly undercounts k.) (b) What is the compression ratio d/k? (c) If the Gaussian map costs $O(dk)$ per document, what is the total projection cost for all N documents? Is this cheaper than computing all $\binom{N}{2}$ pairwise distances in the original space?

E7.2 *(Eckart–Young.)* A matrix $A \in \mathbb{R}^{1000 \times 100}$ has singular values $\sigma_1 = 10, \sigma_2 = 5, \sigma_3 = 2$, and $\sigma_j = 0.1$ for $j \geq 4$. (a) Compute $\|A - A_3\|_F$ exactly. (b) By how much does the rank-3 truncation reduce the Frobenius norm? (c) What fraction of the "signal" (in the sense of $\|A\|_F^2$) is captured by the top 3 singular values?

E7.3 *(Distributed SVD cost.)* A matrix $A \in \mathbb{R}^{m \times n}$, $m = 10^9$, $n = 500$, is distributed across $M = 1000$ machines with equal row partitions. (a) Compute the local map work per machine (forming $A_i^\top A_i$). (b) How many bytes does each machine send in the shuffle (as $n \times n$ float64 matrices)? (c) What is the total shuffle volume? (d) Compare with the cost of shuffling the full matrix for a centralised computation.

E7.4 *(DIMSUM parameter choice.)* A recommendation matrix has $m = 10^9$ user-events (rows) over $n = 10^5$ items, each event touching at most $L = 50$ items, with 0/1 entries. You want every item pair with cosine similarity at least $s = 0.2$ to relative error $\delta = 0.1$. (a) Using $\alpha = 9 \ln n/\delta^2$ and $\gamma = \alpha/s$ (§7.3), compute γ. (b) If every event touches L items, the exact Gram computation emits $m\binom{L}{2}$ pairs — compute this number. (c) DIMSUM emits at most $nL\gamma$ pairs — compute this number. (d) What is the ratio? Find the break-even m^* below which DIMSUM's bound does not beat the exact shuffle, and say how the saving grows beyond it.

Deep (Module-3 / optional):

E7.5 *(JL lower bound — stated.)* A matching lower bound is known: for ε not too small (§3.1), there is an N-point set in $\mathbb{R}^d$ such that *every* linear map from $\mathbb{R}^d$ to $\mathbb{R}^k$ that embeds it with $(1 \pm \varepsilon)$ distortion has $k = \Omega(\log N/\varepsilon^2)$. Explain in two paragraphs why this matches the JL upper bound, and what this implies for the optimality of random projections. (Proof of the lower bound is beyond our scope; see Larsen–Nelson 2016, 2017.)

E7.6 *(HMT power iteration.)* In the randomised SVD algorithm (§4.1), explain why the power iteration step $Y \leftarrow A(A^\top Y)$ helps when the singular values decay slowly (specifically, when σ_k/σ_{k+1} is close to 1). *Hint:* after q iterations, the effective singular values of the modified matrix are σ_j^{2q+1}; compute the effective gap.

E7.7 *(DIMSUM expected shuffle-size derivation.)* Fill in the proof of the shuffle-size bound in §7.3. Specifically: (a) Show that for a single row r, the expected number of emitted pairs is $\sum_{i<j:A_{ri}\neq 0, A_{rj}\neq 0} \min(1, \gamma/(\|c_i\|\|c_j\|))$. (b) Show this is at most $\frac{\gamma}{2H^2}\sum(A_{ri}^2/\|c_i\|^2 + A_{rj}^2/\|c_j\|^2)$ over the same non-zero pairs, using $\min(1, \gamma/ab) \le \gamma(1/a^2 + 1/b^2)/2$ for $a, b > 0$ (the AM-GM inequality $(1/a^2 + 1/b^2)/2 \ge 1/(ab)$) together with $A_{ri}^2 \ge H^2$ for every non-zero entry *(note: without the second fact the step fails for small entries, since the left side does not depend on A_{ri}).* (c) Each non-zero entry of row r pairs with at most $L - 1$ others; sum over all rows, use $\sum_r A_{ri}^2 = \|c_i\|^2$, and obtain $\gamma n(L-1)/(2H^2) = O(nL\gamma/H^2)$, the bound of §7.3. Where did m go? *(This is the key calculation in the DIMSUM paper; working through it gives you the full algorithm understanding.)*

Chapter 14

08 solvers least squares

U8 — Linear solvers & least squares: RAM → PRAM → DPRAM

The ScaDaMaLe book, unit 8 (deep). Our own exposition; original sources credited in §9.

Instantiating the three-lens invariant. The U0 box for linear systems and least squares:

> **RAM:** direct methods (LU, QR, Cholesky) cost $O(n^3)$ and are numerically exact but memory-bound at scale; the least-squares problem $\min_x \|Ax - b\|_2$ is best solved by QR (stable) rather than the normal equations (ill-conditioned). CG solves $Ax = b$ iteratively in at most n steps, with a $\sqrt{\kappa}$ convergence rate — the RAM bridge into the iterative world. **PRAM:** Jacobi iteration is embarrassingly parallel: one independent division per row per sweep. Gauss–Seidel is *not* trivially parallel, but the **graph-colouring theorem** saves it: a proper K-colouring of the adjacency graph of A (no edge joins two vertices of one colour) yields a K-step parallel Gauss–Seidel sweep. Givens rotations for QR parallelise row-disjointly (fold G6). **DPRAM/MPC:** tall-skinny data matrices call for distributed least squares; the two paradigms are *exact* (TSQR, Gramian normal equations via the U7 pipeline) and *sketched* (random projections or leverage-score sampling that reduces to an $O(d \log d/\varepsilon^2)$-row subproblem, preserving the objective to $1 \pm \varepsilon$).

Entry kit — what this chapter assumes; follow a link if you need the memory jog (map: Figure 1): F.M2 (projection, PSD, condition number) • F.M3 (quadratics as the solver–optimizer bridge)

1. Intuition — three regimes of linear problems at scale

A linear system $Ax = b$ or least-squares problem $\min_x \|Ax - b\|_2$ is the most common computational primitive in scientific computing and machine learning. The challenge changes character as the problem grows:

- **Moderate size** ($n \lesssim 10^5$): a direct solver (Gaussian elimination with pivoting, or QR) is the right tool. It is numerically stable and gives a machine-precision answer in a predictable $O(n^3)$ time. Cache effects and BLAS-3 exploitation decide the constant.
- **Large but structured** ($n \lesssim 10^7$): the matrix is sparse or comes from a PDE, and an iterative method (Jacobi, Gauss–Seidel, conjugate gradients) converges to working precision in far fewer than n steps. The convergence rate depends on the condition number $\kappa(A)$.
- **Massive and over-determined** ($m \gg n$, both large): the least-squares matrix $A \in \mathbb{R}^{m \times n}$ does not fit on one machine. Here the game is to reduce the row count without changing the answer — either by a deterministic distributed QR (TSQR) or by a randomised sketch that concentrates with $O(d \log d/\varepsilon^2)$ rows.

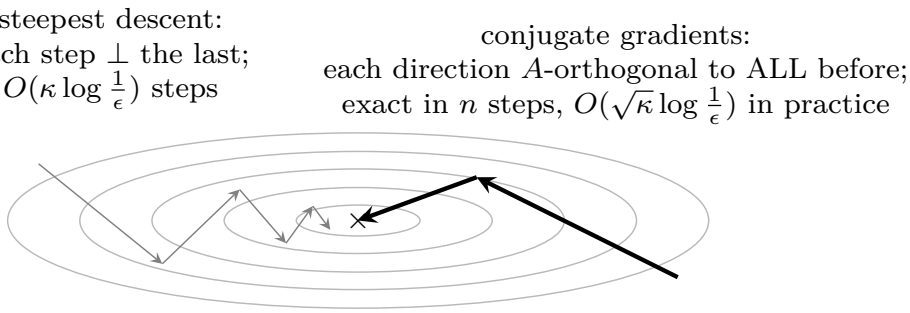


Figure 14.1: One ill-conditioned quadratic (the ellipse eccentricity is κ, F.M2 §5), two solvers. Steepest descent must zig-zag: successive gradients are orthogonal, so progress along the long axis is a random-walk-like crawl costing $O(\kappa)$ iterations. Conjugate gradients chooses each direction A-orthogonal to *every* previous one — no work is ever undone — finishing exactly in n steps and practically in $O(\sqrt{\kappa})$. Same per-iteration cost (one sparse mat-vec, distributable by U6), square-root fewer iterations: the whole art of Krylov methods in one contrast.

The unit walks all three regimes across the three lenses.

Figure 14.1 is the unit's central contrast: the same bowl, a square-root difference in steps.

2. RAM — direct methods and the least-squares problem

2.1 Direct solvers: cost recap

Solver	Applicable to	Dominant cost	Notes
LU (Gaussian elim. + pivoting)	Square non-singular $A \in \mathbb{R}^{n \times n}$	$\frac{2}{3}n^3$ flops	Unstable without pivoting; stable with partial pivoting
Cholesky ($A = LL^T$)	Symmetric positive definite	$\frac{1}{3}n^3$ flops	Twice as fast as LU; fails (with warning) if $A \not\succ 0$
QR ($A = QR$, thin)	Any full-rank A; least squares	$2mn^2 - \frac{2}{3}n^3$ for $m \geq n$	The numerically preferred route for $\min_x \|Ax - b\|_2$

All three are $O(n^3)$ for square systems; for tall-skinny $m \times n$ with $m \gg n$, QR costs $\Theta(mn^2)$. Memory is $O(n^2)$: at $n = 10^5$ that is 80 GB for doubles — not in RAM. This is where iterative methods, distributed QR, and sketching take over.

2.2 Least squares: normal equations versus QR

The least-squares problem $\min_x \|Ax - b\|_2^2$ has the solution $x^* = A^\dagger b$ where $A^\dagger$ is the Moore–Penrose pseudoinverse. Two computational routes:

Normal equations. Differentiate $\|Ax - b\|_2^2$ and set to zero: $A^TAx = A^Tb$. This is an $n \times n$ symmetric positive definite system, solvable by Cholesky in $O(n^3)$. The catch: $\kappa(A^TA) = \kappa(A)^2$. If A is moderately ill-conditioned (say $\kappa(A) \approx 10^8$ in double precision), A^TA has condition number $\approx 10^{16}$, right at the edge of 64-bit floating-point precision. The normal equations can fail catastrophically.

QR factorisation. Write $A = QR$ (thin QR: $Q \in \mathbb{R}^{m \times n}$ orthogonal columns, $R \in \mathbb{R}^{n \times n}$ upper triangular). Then $\|Ax - b\|_2^2 = \|QRx - b\|_2^2 = \|Rx - Q^Tb\|_2^2 + \|(I - QQ^T)b\|_2^2$ (split b into its part QQ^Tb in the column space of Q and the orthogonal rest; Q has orthonormal columns). The second term does not depend on x, so the minimiser satisfies the square triangular system $Rx = Q^Tb$, solved by back-substitution in $O(n^2)$ after the $O(mn^2)$ QR decomposition. QR is stable: the condition number encountered is $\kappa(A)$, not $\kappa(A)^2$.

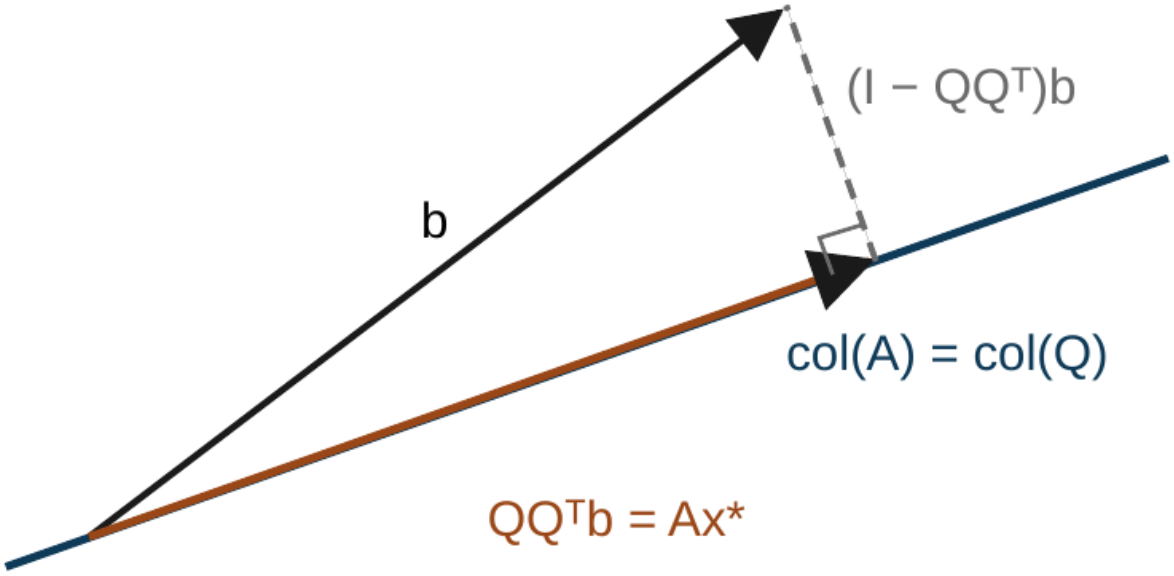


Figure 14.2: Least squares by QR: $Ax^* = QQ^\top b$, the projection of b onto the column space of A.

Rule of thumb. Use QR unless A^TA is already available, A is well-conditioned, and the $O(mn^2)$ QR cost is prohibitive. At scale, TSQR (§4.1) brings QR's stability to the distributed setting.

Figure 14.2 draws the QR route: $Ax^* = QQ^\top b$, the projection of b onto the column space of A.

2.3 Iterative solvers: the splitting framework

For large sparse systems, form the **splitting** $A = M - N$ where M is easy to invert. The iteration is:

$$x^{(k+1)} \;=\; M^{-1}N\,x^{(k)} + M^{-1}b \;=\; Bx^{(k)} + c, \quad B := M^{-1}N.$$

Lemma (error propagation). Let x^* be the true solution ($Ax^* = b$, i.e. $Mx^* = Nx^* + b$). The error $e^{(k)} := x^{(k)} - x^*$ satisfies the **error recursion**:

$$e^{(k+1)} \;=\; B\,e^{(k)}.$$

Proof. $e^{(k+1)} = x^{(k+1)} - x^* = Bx^{(k)} + c - (Bx^* + c) = B(x^{(k)} - x^*) = Be^{(k)}$. □

Theorem (splitting convergence). *If $\rho(B) < 1$ — i.e. all eigenvalues of $B = M^{-1}N$ have modulus less than 1 — then the iteration converges to x^* from any starting point $x^{(0)}$, and for every $\varepsilon > 0$ there is a constant C_ε, depending on the norm, on ε and on B, with*

$$\|e^{(k)}\| \;\le\; C_\varepsilon\,(\rho(B) + \varepsilon)^k\,\|e^{(0)}\|.$$

If B is diagonalisable one may take $\varepsilon = 0$, with C the condition number of its eigenvector matrix (in the 2-norm). For a defective B one may not: a 2×2 Jordan block has $\|B^k\| \ge k\rho(B)^{k-1}$, which no constant multiple of $\rho(B)^k$ bounds.

Proof (one-line induction + spectral radius argument). By the recursion, $e^{(k)} = B^k e^{(0)}$. For any $\varepsilon > 0$ there exists a matrix norm $\|\cdot\|_\varepsilon$ such that $\|B\|_\varepsilon \le \rho(B) + \varepsilon$ (subordinate norm from an approximate diagonalisation of B). Choosing $\varepsilon < 1 - \rho(B)$ gives $\|B\|_\varepsilon < 1$, hence

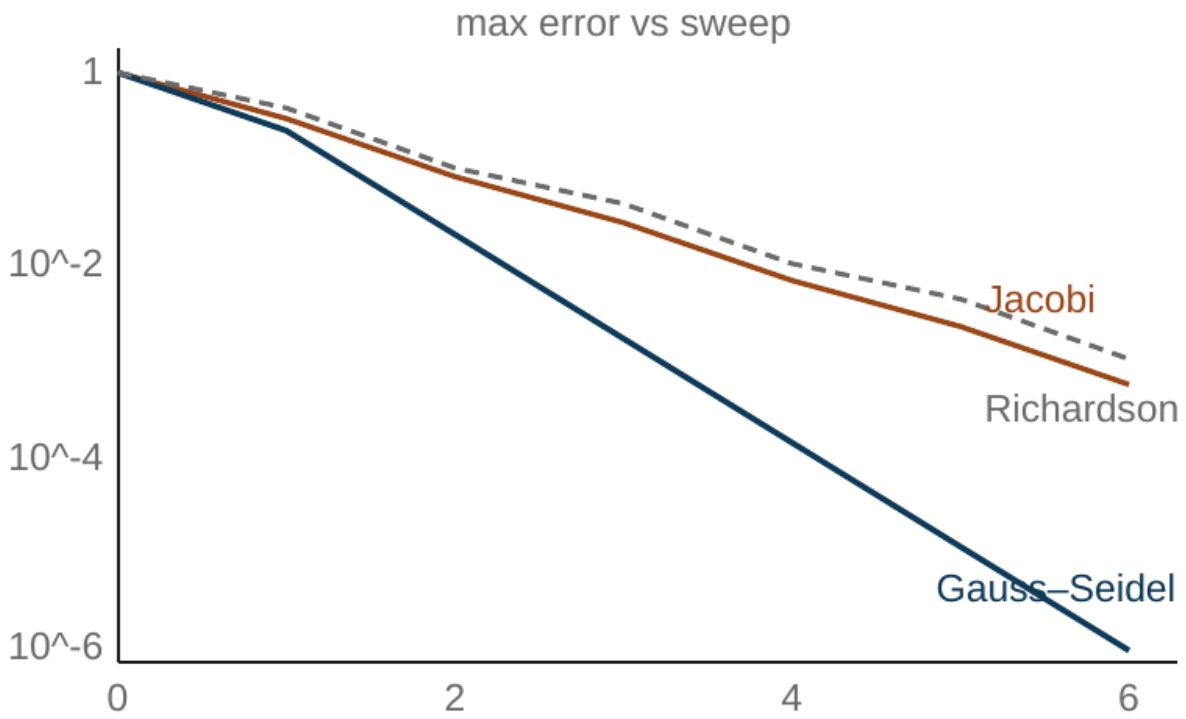


Figure 14.3: Jacobi, Gauss–Seidel and Richardson: the largest error after each sweep.

$\|e^{(k)}\|_\varepsilon \le \|B\|_\varepsilon^k \|e^{(0)}\|_\varepsilon \to 0$. The equivalence of norms on $\mathbb{R}^n$ converts to any fixed norm with a constant C_ε, and $\|B\|_\varepsilon^k \le (\rho(B)+\varepsilon)^k$. The geometric rate is $\rho(B)$ in the limit. □

Remark (the honest hard part — Lean flag). The above argument produces a geometric bound with rate $\rho(B)+\varepsilon$ for any $\varepsilon > 0$, but the constant C_ε hides the dependence on ε and the condition number of the eigenvector matrix. Making C_ε explicit requires bounding the spectral norm via matrix norms — that is the honest, harder, quantitative statement. The Lean schematic in §8 flags this: the spectral-radius step is where a complete machine-checked proof would need the most work.

2.4 Jacobi, Gauss–Seidel, Richardson as instances

Write $A = D + L + U$ where D = diagonal, L = strict lower triangular, U = strict upper.

Method	M	N	$B = M^{-1}N$	Convergence condition
Jacobi	D	$-(L+U)$	$-D^{-1}(L+U)$	$\rho(D^{-1}(L+U)) < 1$
Gauss–Seidel	$D+L$	$-U$	$-(D+L)^{-1}U$	$\rho((D+L)^{-1}U) < 1$
Richardson	$\omega^{-1}I$	$\omega^{-1}I - A$	$I - \omega A$	$\rho(I-\omega A) < 1$

For symmetric positive definite A: **Jacobi** converges iff $\rho(D^{-1}(L+U)) < 1$ (not always true, though it holds for strictly diagonally dominant matrices). **Gauss–Seidel** converges for all SPD A (Theorem: for SPD matrices, $\rho((D+L)^{-1}U) < 1$ — stated; proof via the energy argument, see Golub–Van Loan §10.1). **Richardson** converges for optimal $\omega = 2/(\lambda_{\min}+\lambda_{\max})$ with rate $(\kappa-1)/(\kappa+1)$ where $\kappa = \lambda_{\max}/\lambda_{\min}$ — the same ratio that CG targets.

Figure 14.3 compares the three splittings sweep by sweep; Gauss–Seidel's error falls fastest.

2.5 Conjugate gradients (CG) — iterative solution for SPD systems

For SPD A, conjugate gradients (Hestenes–Stiefel 1952) is the method of choice. We state it here; the derivation (minimisation of the quadratic energy over Krylov subspaces) is worked through in Shewchuk's exposition.

Algorithm (CG, Shewchuk's notation). Initialise $x^{(0)} = 0$, $r^{(0)} = b$, $d^{(0)} = r^{(0)}$. For $k = 0, 1, 2, \ldots$:

$$\alpha_k = \frac{r^{(k)T} r^{(k)}}{d^{(k)T} A d^{(k)}}, \quad x^{(k+1)} = x^{(k)} + \alpha_k d^{(k)}, \quad r^{(k+1)} = r^{(k)} - \alpha_k A d^{(k)},$$

$$\beta_k = \frac{r^{(k+1)T} r^{(k+1)}}{r^{(k)T} r^{(k)}}, \quad d^{(k+1)} = r^{(k+1)} + \beta_k d^{(k)}.$$

Stop when $\|r^{(k)}\| \le \tau$. Each iteration costs one matrix–vector product $Ad^{(k)}$ plus $O(n)$ dot products — a total of $O(\mathrm{nnz}(A))$ per step for sparse A.

Theorem (CG convergence rate; see Shewchuk §9.2, Eq. (52)). *After k iterations of CG on an SPD system with condition number $\kappa = \kappa(A) = \lambda_{\max}/\lambda_{\min}$:*

$$\|e^{(k)}\|_A \;\le\; 2\left(\frac{\sqrt{\kappa}-1}{\sqrt{\kappa}+1}\right)^k \|e^{(0)}\|_A,$$

where $\|v\|_A = \sqrt{v^T A v}$ is the A-norm.

Consequence. To reduce the A-norm error by a factor of ε: it suffices to take

$$k \;\ge\; \frac{\sqrt{\kappa}}{2} \ln\frac{2}{\varepsilon}$$

iterations (using $\ln((1+t)/(1-t)) \ge 2t$ for small t where $t = 1/\sqrt{\kappa}$). The $\sqrt{\kappa}$ dependence — rather than κ for steepest descent — is what makes CG attractive for well-structured problems (e.g. Laplacian systems where $\kappa = O(n^{2/d})$ for d-dimensional grids). (Reference: Shewchuk, *An introduction to the conjugate gradient method without the agonizing pain*, 1994, §9.)

Preconditioning. Replace A by $M^{-1/2} A M^{-1/2}$ (implicitly) for a symmetric positive definite preconditioner $M \approx A$. The convergence rate depends on $\kappa(M^{-1}A)$: a good preconditioner clusters the eigenvalues, making CG converge in $O(1)$ or $O(\sqrt{\kappa(M^{-1}A)})$ iterations rather than $O(\sqrt{\kappa(A)})$. Preconditioned CG is the standard solver for large-scale PDE problems; exposition deferred to a deep exercise.

3. PRAM — parallelising solvers

3.1 Jacobi: embarrassingly parallel per sweep

Each Jacobi update for variable i reads row i of A and the *old* iterate $x^{(k)}$:

$$x_i^{(k+1)} = \frac{1}{A_{ii}} \left(b_i - \sum_{j \ne i} A_{ij} x_j^{(k)} \right).$$

Because all updates read from $x^{(k)}$ (the old iterate) and write to $x^{(k+1)}$ (a fresh buffer), there are **no write–write or write–read conflicts**: processor i and processor j operate on different rows and different output locations. Each sweep is a trivially parallel $O(n)$-work, $O(1)$-depth (per row, given n processors and sparse A) operation on EREW PRAM.

Work and depth. With n processors and a sparse matrix ($\mathrm{nnz}(A) = O(n)$ non-zeros): each sweep has $W = O(n)$, $D = O(1)$. For dense A: $W = O(n^2)$, $D = O(\log n)$ (parallel row dot-products). By Brent: $T_p \le O(n^2/p) + O(\log n)$ per sweep; with $p = n$ this is $O(n + \log n) = O(n)$ per sweep. On the model problems of this unit Jacobi contracts the error by $\rho \approx 1 - O(1/\kappa)$ per sweep (E8.1), so one runs $O(\kappa \log(1/\varepsilon))$ sweeps for a total of $O(n\kappa \log(1/\varepsilon))$ wall-clock steps — often far less than the direct $O(n^3)$ cost. (The $\sqrt{\kappa}$ of CG above is precisely what Krylov acceleration buys over these plain sweeps.)

3.2 Gauss–Seidel: the graph-colouring theorem

Gauss–Seidel updates use the *new* iterate values as soon as they are computed:

$$x_i^{(k+1)} = \frac{1}{A_{ii}} \left(b_i - \sum_{j<i} A_{ij} x_j^{(k+1)} - \sum_{j>i} A_{ij} x_j^{(k)} \right).$$

The forward dependency $x_j^{(k+1)}$ for $j < i$ makes the naive sequential Gauss–Seidel *not directly parallel*: update i must wait for update $j < i$. The dependency graph has a directed edge $j \to i$ whenever $A_{ij} \neq 0$ and $j < i$ (a "forward" coupling).

Theorem (Gauss–Seidel graph-colouring parallelism). *Let $G = (V, E)$ be the undirected graph of non-zeros of A ($V = \{1, \dots, n\}$, $(i, j) \in E$ iff $A_{ij} \neq 0$, $i \neq j$). If G admits a proper K-colouring $\chi : V \to [K]$ (adjacent vertices receive different colours), then one Gauss–Seidel sweep can be executed in exactly K parallel steps: in step $c = 1, \dots, K$, update all variables in colour class $\chi^{-1}(c)$ simultaneously. Moreover, this K-step parallel sweep is correct — i.e. it produces the same result as the sequential sweep in colour-major ordering $\chi^{-1}(1), \chi^{-1}(2), \dots, \chi^{-1}(K)$.*

Proof sketch.

Within colour class c, no variable in that class has a Gauss–Seidel forward dependency on any other variable in the same class. Concretely: a forward dependency $j \to i$ within a class requires $A_{ij} \neq 0$ with $j < i$, which means $(i, j) \in E$; but then i and j must receive different colours under a proper colouring — contradicting both being in class c. Therefore all updates in colour class c are mutually independent (each reads only values already fixed from previous colour classes or the previous sweep), and they can be applied in parallel. Sequencing colour classes $1, 2, \dots, K$ in order gives exactly the Gauss–Seidel result for the colour-major variable ordering. □

Example: grid Laplacian. For a 2D grid $\{1, \dots, \sqrt{n}\}^2$ with the 5-point Laplacian, the graph is bipartite (checkerboard colouring): $K = 2$, so one parallel Gauss–Seidel sweep takes 2 parallel steps (one for the black squares, one for the white squares). This is the standard approach for PDE solvers; the lab exercises it directly (§7).

Work and depth of the K-step parallel sweep. Each colour class is updated with independent row computations: work $W = O(n)$ per class (sparse), depth $D = O(1)$ per class (row dot-products in parallel). Total for one sweep: $W = O(Kn)$, $D = O(K)$. For the grid Laplacian: $K = 2$, constant overhead.

Figure 14.4 shows the two-colouring ($K = 2$) of a 3×3 grid that lets a Gauss–Seidel sweep run in two parallel steps.

3.3 Givens-rotation parallel QR (fold G6)

A **Givens rotation** $G(i, j, \theta)$ is an orthogonal matrix that rotates in the (i, j) plane, zeroing out element (j, i) (or (i, j)) of a target matrix. Applying one Givens rotation to an $m \times n$ matrix costs $O(n)$ flops (only two rows change). Classical Householder QR applies reflections one at a time, sequentially annihilating sub-diagonal entries. Givens rotations expose more parallelism.

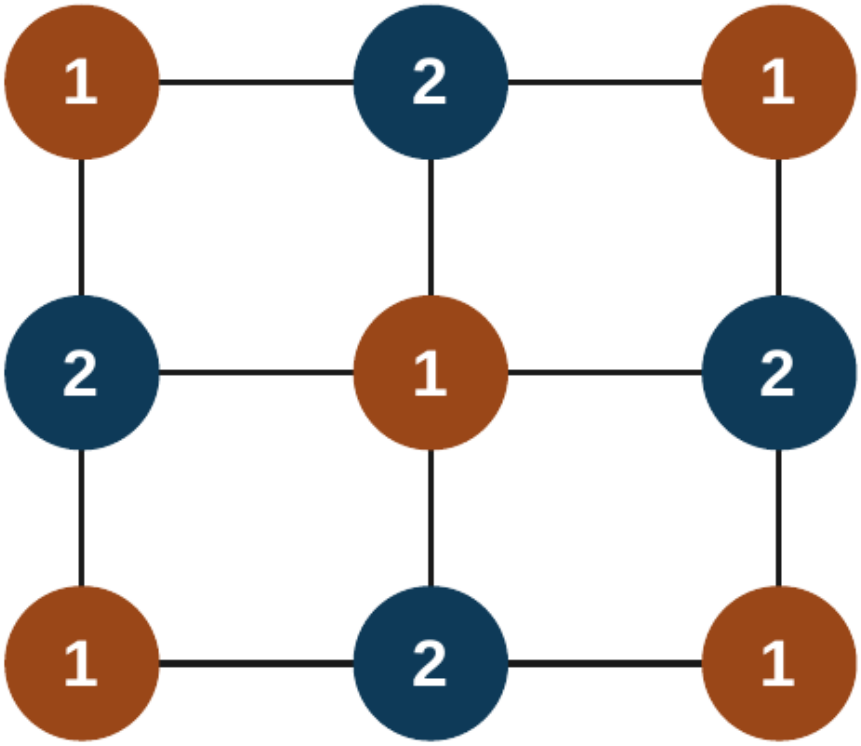


Figure 14.4: A two-colouring of a 3×3 grid: each colour class is one parallel step of a Gauss–Seidel sweep.

Key observation. Two Givens rotations $G(i_1, j_1, \theta_1)$ and $G(i_2, j_2, \theta_2)$ *commute* if they act on **disjoint row pairs**: $\{i_1, j_1\} \cap \{i_2, j_2\} = \emptyset$. If two rotations act on disjoint rows they can be applied simultaneously (parallel step).

Parallel Givens QR schedule. For an $m \times n$ matrix, consider the column-by-column annihilation pattern: to zero out column j's sub-diagonal, we apply rotations to row pairs $(m, m-1), (m-1, m-2), \ldots, (j+1, j)$ in sequence (a "wavefront"). The wavefront can be pipelined: while column $j+1$ is being processed, column j can start. More precisely:

- At each *parallel step*, select a maximal set of row-disjoint Givens rotations across all columns being processed. Apply them simultaneously.
- **Work:** $W = O(mn^2)$ total Givens operations (same order as sequential Householder QR for $m \geq n$). **Depth:** $D = O(m+n)$ parallel steps — a linear pipeline schedule (see JáJá §6 or the systolic array literature). For square matrices ($m = n$): depth $O(n)$ vs work $O(n^3)$, parallelism $O(n^2)$. Brent: $T_p \leq O(n^3/p) + O(n)$.

This schedule underlies the **systolic array** QR implementations used in hardware (e.g. FPGAs for real-time signal processing) and motivates the TSQR tree design in §4.1.

4. DPRAM / MPC — distributed least squares

4.1 Exact distributed least squares

Problem. $A \in \mathbb{R}^{m \times n}$ is partitioned row-wise across P machines: $A = [A_1^T | A_2^T | \cdots | A_P^T]^T$, each $A_p \in \mathbb{R}^{m/P \times n}$ with corresponding b_p. We want $x^* = \arg\min_x \|Ax - b\|_2$ exactly.

Route (a): TSQR (Demmel–Grigori–Hoemmen–Langou 2012; preprint 2008). *Tall-*

skinny QR exploits the block structure of the row partition. Each machine computes a local QR:

$$A_p = Q_p R_p, \quad R_p \in \mathbb{R}^{n\times n} \text{ (upper triangular)}.$$

The key fact: $\|Ax-b\|_2^2 = \sum_p \|A_p x - b_p\|_2^2$, and the QR of the stacked R_p matrices $[R_1^T | \cdots | R_P^T]^T$ (size $Pn \times n$) gives the global QR of A. This "factor-then-communicate-only-the-R" structure is the core of TSQR:

1. **Round 1.** Each machine p computes $A_p = Q_p R_p$ (local QR, $O((m/P)n^2)$ flops) and updates $\tilde{b}_p = Q_p^T b_p$. Sends R_p and $\tilde{b}_p^{(1)}$ (the first n components of $Q_p^T b_p$, which interact with R_p) to the next level.
2. **Tree aggregation.** Pair machines, concatenate their R factors, and compute another local QR. Repeat until one machine holds $R \in \mathbb{R}^{n\times n}$.
3. **Back-solve.** Solve $Rx = Q^T b$ by back-substitution ($O(n^2)$).

Communication. Each round communicates $n \times n$ triangular factors — $O(n^2)$ values, regardless of m. Rounds: $R = O(\log P)$ (binary tree). For $m \gg n$ (the tall-skinny regime), the dominant cost is the local QR ($O((m/P)n^2)$ per machine, perfectly parallel), and the communication is a tiny $O(n^2 \log P)$ tail. This is **communication-optimal** for the exact case. Spark implements TSQR as `RowMatrix.tallSkinnyQR` (in `org.apache.spark.mllib.linalg.distributed`).

Route (b): Gramian normal equations via the U7 pipeline. Compute $A^T A$ and $A^T b$ in a single distributed pass (exactly as in the distributed Gram matrix computation of U7). Each machine computes $A_p^T A_p$ and $A_p^T b_p$; tree-reduce with addition gives the global $A^T A$ and $A^T b$ in $O(\log P)$ rounds, communicating $O(n^2)$ values. Solve with Cholesky in $O(n^3)$ on the coordinator. This is **faster to implement** but inherits the conditioning issue ($\kappa(A^T A) = \kappa(A)^2$) of the normal equations — acceptable when A is well-conditioned.

When to use which. If numerical stability matters (ill-conditioned features, polynomial regression, kernel methods): TSQR. If A is well-conditioned and n is small (say $n \le 10^3$): normal equations. At scale in Spark, normal equations with Gramian are the common first cut; TSQR is the precision upgrade.

4.2 Sketch-and-solve: randomised approximate least squares

When even TSQR is too slow (e.g. $m = 10^9$, $n = 10^4$), the **sketch-and-solve** paradigm trades exactness for speed. The idea: replace the $m \times n$ problem by a much smaller one whose solution is provably close.

Setup. Let $A \in \mathbb{R}^{m\times n}$, $b \in \mathbb{R}^m$, and let $\Phi \in \mathbb{R}^{s\times m}$ be a random *sketching matrix* (e.g. a scaled random Gaussian or a Hadamard transform). Define the *sketch*:

$$\tilde{A} = \Phi A \in \mathbb{R}^{s\times n}, \quad \tilde{b} = \Phi b \in \mathbb{R}^s.$$

The sketched least-squares problem is $\tilde{x} = \arg\min_x \|\tilde{A}x - \tilde{b}\|_2$.

Leverage scores. The **statistical leverage score** of row i of A is

$$\ell_i(A) \;:=\; [UU^T]_{ii} \;=\; \|U_i\|_2^2,$$

where $U \in \mathbb{R}^{m\times n}$ is the matrix of left singular vectors of A (i.e. the Q factor from a thin QR). Equivalently, $\ell_i = \|(A^T A)^{-1/2} a_i^T\|_2^2$ where a_i^T is row i of A. The leverage scores satisfy $\ell_i \in [0,1]$ and $\sum_i \ell_i = n$ (since $\mathrm{tr}(UU^T) = n$).

Intuitively, a row with high leverage score is "influential" — it contributes disproportionately to the column space of A. Rows that look like outliers in the design matrix have high leverage. A uniform random sample of rows will miss high-leverage rows with high probability, biasing the

solution. Leverage-score sampling corrects this by sampling each row with probability $\propto \ell_i$, so that influential rows appear more often.

Theorem (leverage-score sampling preserves the least-squares objective; Drineas–Mahoney–Muthukrishnan 2006). *Let $A \in \mathbb{R}^{m\times n}$ have rank n, and let $\varepsilon \in (0, 1/2)$. Sample $s = O(n \log n/\varepsilon^2)$ rows independently, each row i with probability $p_i \geq c\ell_i/n$ for an absolute constant $c > 0$, forming the rescaled sketch $\tilde{A}$ (rescale row i by $1/\sqrt{sp_i}$). Then with probability at least 0.9:*

$$\|A\tilde{x} - b\|_2^2 \;\leq\; (1+\varepsilon)\,\|Ax^* - b\|_2^2,$$

i.e. the minimiser $\tilde{x}$ of the sketched problem achieves true objective cost within a factor $1+\varepsilon$ of the true optimum x^*.

The statement also implies $\|A(\tilde{x} - x^*)\|_2 = O(\varepsilon^{1/2})\|Ax^* - b\|$, so the solution quality is controlled too (not just the objective value). See Drineas–Mahoney–Muthukrishnan 2006 (SODA) for the complete proof via matrix Chernoff.

Random-projection sketch (JL/U7 tie-in). An alternative sketching matrix is a scaled random Gaussian or Hadamard-based sketch Φ (as in JL and U7). The JL lemma guarantees that with $s = O(n/\varepsilon^2)$ rows, the sketch approximately preserves all pairwise distances in the column space of A. This needs *fewer* rows than leverage-score sampling (no $\log n$ factor) and no prior knowledge of the leverage scores, so it is a single-pass algorithm. However, the price is paid in arithmetic: a dense Gaussian Φ costs $O(mns)$ to apply, which a Hadamard-based sketch cuts to $O(mn \log m)$.

When sketch beats exact (flop accounting). Suppose $m = 10^9$, $n = 10^4$. Every method reads A at least once: $mn = 10^{13}$ entries, roughly 80 TB in float64. Note that the sketch saves no input bytes, since it learns which rows matter only after a pass over A (to estimate the leverage scores, or to apply Φ). What it saves is arithmetic. TSQR performs about $2mn^2 = 2 \times 10^{17}$ flops in its local QRs. A Hadamard-based sketch costs about $mn \log_2 m \approx 10^{13} \times 30 = 3 \times 10^{14}$ flops to apply, and solving the sketched problem with $s = O(n \log n/\varepsilon^2) \approx 10^6$ rows ($\varepsilon \approx 0.3$) costs about $sn^2 = 10^{14}$ more — some 500 times fewer flops in all. In words, the sketch wins when TSQR's n^2 per row outweighs the sketch's $n \log m$ per row, and the sketched problem is much shorter than the original ($s \ll m$); then it fits in memory and is solved directly. For moderate m/n ratios, or when the exact solution is required, TSQR is preferred because it is exact.

Estimating leverage scores in one distributed pass. A complication: computing exact leverage scores requires the full SVD of A, which is what we are trying to avoid. The practical solution: compute *approximate* leverage scores via a random projection sketch of A^TA (cost: one pass, one reduce round — the U7 pipeline), or via a fast JL sketch of A followed by a local QR. This approximation is sufficient: the sampling guarantee holds with $p_i \geq c\,\tilde{\ell}_i/n$ for any $\ell_i \leq \tilde{\ell}_i \leq O(\ell_i)$ approximation.

Figure 14.5 shows which rows carry high leverage, and Figure 14.6 prices sketch-and-solve against the exact solve at $m = 10^9$, $n = 10^4$.

4.3 Dual decomposition (forward pointer to U19)

The sketch-and-solve paradigm handles the unconstrained or box-constrained case. Many practical least-squares problems are **separably constrained**: each machine owns a block of variables $x_p \in \mathcal{X}_p$ (convex sets), and the global objective is $\sum_p f_p(x_p)$ subject to a coupling constraint $\sum_p A_p x_p = c$. This is the template for multi-party regression, federated learning, and network flow with a least-squares objective.

Dual decomposition (Boyd–Vandenberghe, §5.5) handles this by introducing a Lagrange multiplier λ for the coupling constraint. The dual function $g(\lambda) = \sum_p \min_{x_p \in \mathcal{X}_p}\{f_p(x_p) +$

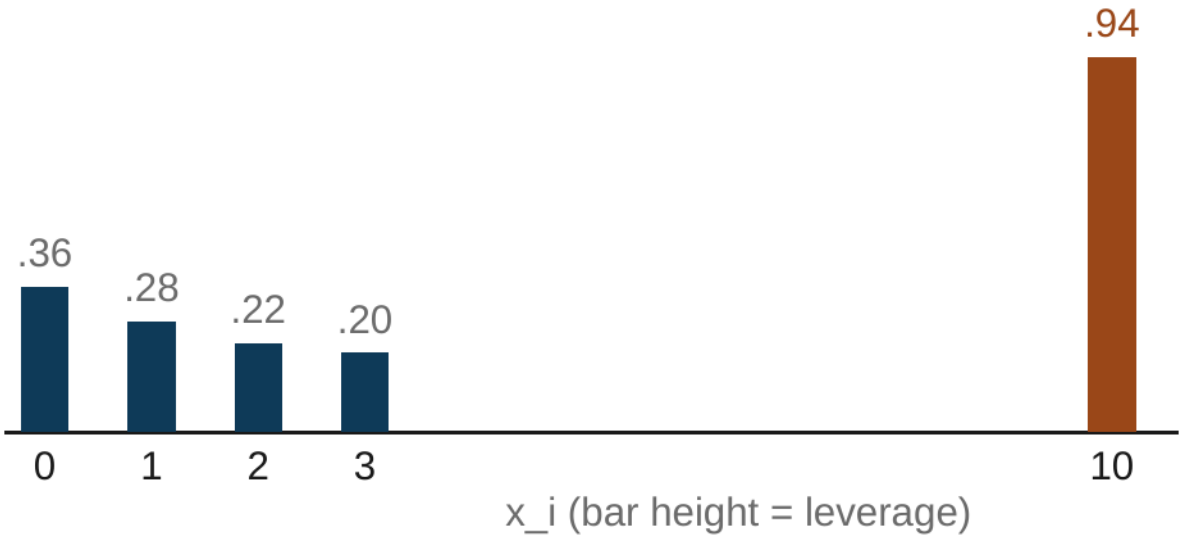


Figure 14.5: Leverage scores mark the rows a sample must not miss.

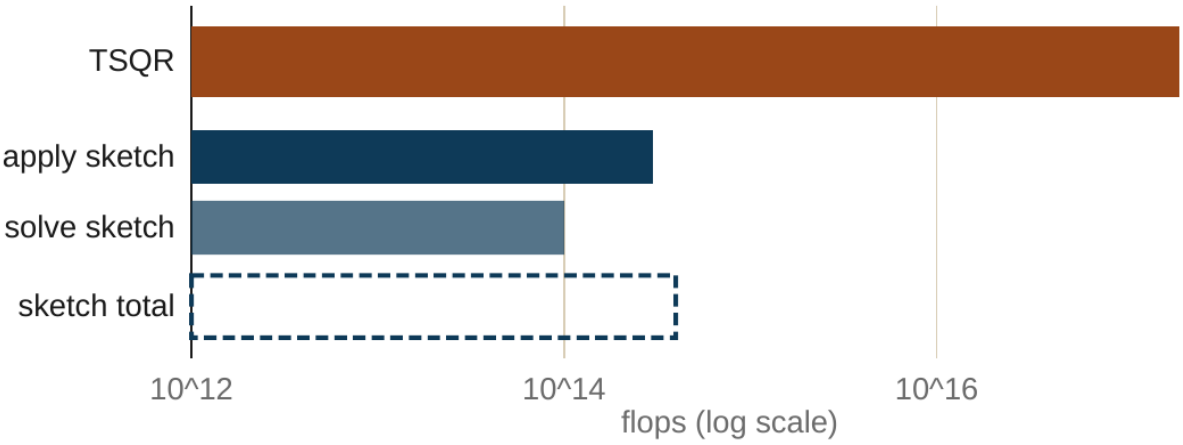


Figure 14.6: When the sketch wins: the cost at $m = 10^9$ and $n = 10^4$.

$\lambda^T A_p x_p\} - \lambda^T c$ decomposes into P independent minimisations — each machine solves its own subproblem given the current λ. The coordinator updates λ by subgradient ascent on the dual. If strong duality holds (Slater's condition), the primal solution is recovered from the dual optimal.

Dual decomposition converts a coupled distributed problem into a sequence of independent local subproblems plus one small global update — the prototype pattern for U19 (distributed optimisation: ADMM, consensus, decentralised gradient descent). Full treatment deferred to U19; flagged here as the natural exit point from distributed least squares into constrained optimisation.

5. The projection theorem — foundation for U9 and U19

The projection theorem is the geometric backbone behind least squares, CG, and all of U9's gradient methods. We state and prove it here; it is invoked repeatedly in the remainder of the book.

Theorem (projection onto a closed convex set). *Let $\mathcal{C} \subseteq \mathbb{R}^n$ be a nonempty, closed, convex set, and let $x \in \mathbb{R}^n$. There exists a unique point $\Pi_{\mathcal{C}}(x) \in \mathcal{C}$ such that*

$$\|\Pi_{\mathcal{C}}(x) - x\|_2 \;=\; \min_{y \in \mathcal{C}} \|y - x\|_2.$$

Moreover: 1. **(Variational characterisation.)** $\Pi_{\mathcal{C}}(x)$ is the unique point in $\mathcal{C}$ satisfying $\langle \Pi_{\mathcal{C}}(x) - x,\ y - \Pi_{\mathcal{C}}(x) \rangle \geq 0$ for all $y \in \mathcal{C}$. 2. **(Nonexpansiveness.)** $\|\Pi_{\mathcal{C}}(x) - \Pi_{\mathcal{C}}(z)\|_2 \leq \|x - z\|_2$ for all $x, z \in \mathbb{R}^n$.

Proof of uniqueness (parallelogram / strong convexity argument). Suppose $u, v \in \mathcal{C}$ both achieve the minimum distance $\delta = \min_{y \in \mathcal{C}} \|y - x\|_2$. Then their midpoint $w = (u+v)/2 \in \mathcal{C}$ by convexity. By the parallelogram law:

$$\|u - v\|_2^2 = 2\|u - x\|_2^2 + 2\|v - x\|_2^2 - 4\|w - x\|_2^2 = 4\delta^2 - 4\|w - x\|_2^2.$$

Since $\|w - x\|_2 \geq \delta$ (as $w \in \mathcal{C}$), we get $\|u - v\|_2^2 \leq 0$, hence $u = v$. $\square$

Proof of the variational characterisation. Let $p = \Pi_{\mathcal{C}}(x)$. For any $y \in \mathcal{C}$ and $t \in [0, 1]$, the point $p + t(y - p) \in \mathcal{C}$ by convexity. The function $t \mapsto \|p + t(y - p) - x\|_2^2$ is minimised at $t = 0$, so its derivative at $t = 0^+$ is non-negative: $2\langle p - x, y - p \rangle \geq 0$, i.e. $\langle p - x, y - p \rangle \geq 0$. $\square$

Nonexpansiveness. Let $p = \Pi_{\mathcal{C}}(x)$ and $q = \Pi_{\mathcal{C}}(z)$. By the variational characterisation: $\langle p - x, q - p \rangle \geq 0$ and $\langle q - z, p - q \rangle \geq 0$. Adding: $\langle (p - q) - (x - z), q - p \rangle \geq 0$, i.e. $\|p - q\|_2^2 \leq \langle x - z, p - q \rangle \leq \|x - z\|_2 \|p - q\|_2$, giving $\|p - q\|_2 \leq \|x - z\|_2$. $\square$

Connection to least squares. If $\mathcal{C} = \{Ax : x \in \mathbb{R}^n\} = \mathrm{col}(A)$ is the column space of A (a closed linear subspace — convex and closed), then $\Pi_{\mathcal{C}}(b)$ is the projection of b onto the column space, and $x^* = \arg\min_x \|Ax - b\|_2$ satisfies $Ax^* = \Pi_{\mathrm{col}(A)}(b)$. The variational characterisation gives the normal equations $A^T(Ax^* - b) = 0$ directly. CG is an algorithm for computing this projection iteratively along conjugate directions. Projected gradient descent (U9) generalises to arbitrary convex $\mathcal{C}$.

Figure 14.7 draws the projection theorem for a convex set C: the angle at the projection p is at least 90°; the column space of A is one such C.

6. The invariant (U8 instantiation)

> **RAM:** direct methods (LU/Cholesky/QR) cost $O(n^3)$; use QR for least squares (stable, condition number $\kappa(A)$ not $\kappa(A)^2$). Iterative splitting: converges if $\rho(M^{-1}N) <$

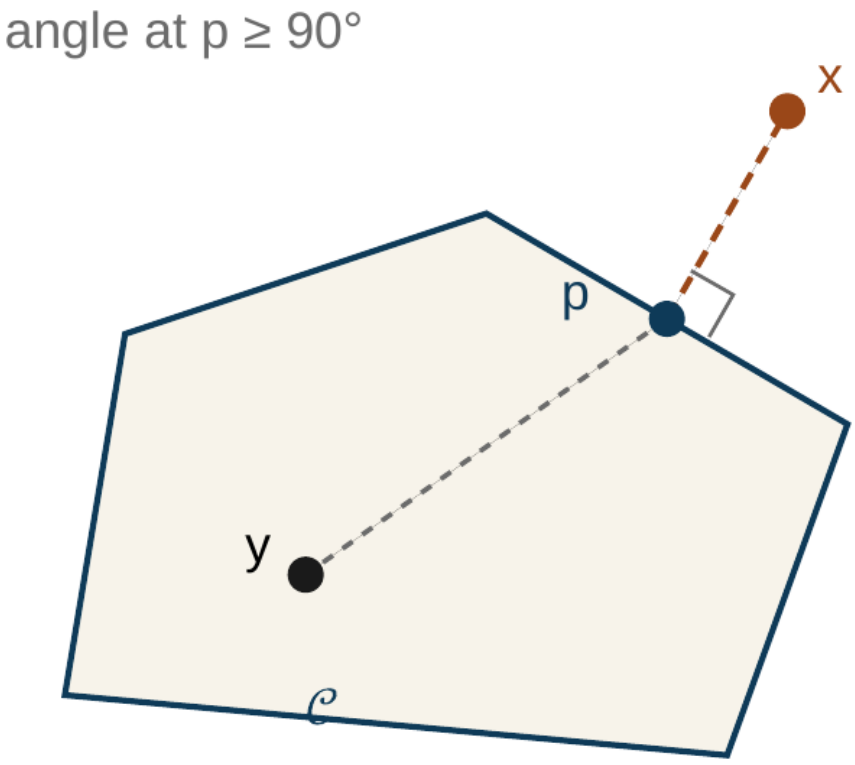


Figure 14.7: The projection theorem for a closed convex set C.

1, geometric rate; Jacobi/GS/Richardson are instances. CG: $O(\sqrt{\kappa}\log(1/\varepsilon))$ iterations, each one matrix–vector product.

PRAM: Jacobi is embarrassingly parallel per sweep (EREW, $W = O(n)$, $D = O(1)$). Gauss–Seidel parallelises via graph colouring: a K-colouring yields a K-step parallel sweep iff no monochromatic edge (a proper colouring). Givens-rotation QR: row-disjoint rotations commute and run in parallel; depth $O(m+n)$, work $O(mn^2)$.

DPRAM/MPC: exact distributed least squares via TSQR ($O(\log P)$ rounds, $O(n^2)$ communication) or Gramian normal equations (same round count, faster but less stable). Sketched least squares via leverage-score sampling ($s = O(n\log n/\varepsilon^2)$ rows, $(1+\varepsilon)$ objective guarantee) or random projection (JL, $O(n/\varepsilon^2)$ rows, no leverage-score oracle needed). Dual decomposition couples separable problems via one shared dual variable — the gateway to U19. Projection theorem: nonexpansive projector onto closed convex set, unique, variational — geometric foundation of least squares, CG, and all of U9.

7. Lab — project track

Project-track lab (no weekly receipt): run these experiments free and locally in the course lab image (`labs/image/`) — or any Spark-in-Docker environment — as deep-dive and group-project material.

Experiment 1 — Jacobi vs Gauss–Seidel colouring on a grid Laplacian.

Construct the 2D grid Laplacian $A \in \mathbb{R}^{n\times n}$ for a $\sqrt{n}\times\sqrt{n}$ grid (5-point stencil) with $n = 40\,000$. Implement: (a) Sequential Gauss–Seidel (natural row order). (b) Parallel Jacobi in Spark (one RDD update per sweep — each row is a partition or a slice of a partition; collect all rows,

compute new iterate, redistribute). (c) Parallel Gauss–Seidel with 2-colouring (checkerboard: one Spark stage per colour class).

For a right-hand side b sampled uniformly and a zero initial iterate: plot $\|e^{(k)}\|$ vs sweep count; compare wall-clock per sweep for (a)/(b)/(c). *Question:* does the 2-colour GS converge at the same rate as sequential GS? (Asymptotically yes, though not because the iteration matrix is the same: reordering the unknowns changes B. Both the natural and the red–black orderings of the 5-point Laplacian are *consistently ordered*, so in both $\rho(B_{\mathrm{GS}}) = \rho(B_{\mathrm{J}})^2$, and the Jacobi matrix has the same spectrum in every ordering.) What is the wall-clock difference?

Experiment 2 — TSQR vs normal equations: conditioning.

Generate a tall-skinny $A \in \mathbb{R}^{m\times n}$ with $m = 10^6$, $n = 50$, and condition number $\kappa(A)$ adjustable via a diagonal scaling (SVD with prescribed singular values from 1 to κ). Solve $\min_x \|Ax - b\|_2$ via: (a) Normal equations + Cholesky (Spark: one reduce round for A^TA, then Cholesky on the coordinator). (b) TSQR (implemented as a binary tree of local QRs on each Spark partition, reducing to one $n \times n$ R factor at the root, then back-solve).

For $\kappa \in \{10^2, 10^4, 10^6, 10^8\}$: compute $\|x_{\mathrm{computed}} - x^*\|_2/\|x^*\|_2$ (forward error). *Question:* at what κ do the normal equations lose accuracy? Is the crossover consistent with the κ^2 condition number analysis?

Deliverable. Notebook with both experiments, plots, and a paragraph of cost-accounting in the three-lens vocabulary for each.

8. Lean certificate target (`BMLSPar.lean` family)

Target (U8 — schematic, CG convergence): certify the convergence rate statement $\|e^{(k)}\|_A \le 2((\sqrt{\kappa}-1)/(\sqrt{\kappa}+1))^k \|e^{(0)}\|_A$ for the conjugate gradient method on an SPD system.

```
-- SCHEMATIC target statement (align names to BMLSPar conventions at cert time):
-- Requires: a formalisation of the CG recurrence (Hestenes--Stiefel 1952),
--           the Krylov subspace property, and the Chebyshev polynomial bound.
-- The spectral-radius step (relating rho(B) to kappa) is the honest hard part:
-- it requires bounding matrix norms by spectral radius, which demands
-- a formalisation of the Jordan normal form or the approximate-diagonalisation
-- argument (epsilon-norm). This is flagged as the load-bearing gap.
theorem cg_convergence_rate
    (A : Matrix (Fin n) (Fin n) ℝ) (hA : PosDef A)
    (kappa : ℝ) (hkappa : kappa = A.condNum)
    (x_star x0 : Fin n → ℝ) (k : ℕ) :
    A_norm (cg_iterate A b x0 k - x_star) ≤
      2 * ((Real.sqrt kappa - 1) / (Real.sqrt kappa + 1))^k *
      A_norm (x0 - x_star) := by
  sorry -- SCHEMATIC: cert target; spectral-radius-via-matrix-norms is the hard
    lemma
```

Exercise U8.L1 (light). State the splitting-convergence theorem for the Jacobi iteration on a 2×2 SPD matrix. Compute $\rho(D^{-1}(L+U))$ symbolically and verify convergence. Write the one-step error bound as a Lean `calc` block (using `sorry` for the operator-norm step $\|B\|_2 = |b|/\min(a,d)$; in the 2-norm a one-step bound through $\rho(B)$ is false).

Exercise U8.L2 (deep). State the graph-colouring theorem for a path graph (n nodes, 2-colourable) as a Lean proposition. Prove that the 2-colour parallel Gauss–Seidel sweep is a valid step, i.e. that no edge of the path joins two nodes of the same colour.

LLM lens — where this unit lands on the running workload (U0L):

Normalisation layers (layernorm/RMSNorm) are conditioning in the κ sense of this unit: they keep the optimisation landscape's curvature ratio tame so that first-order methods (U9) converge — the same condition-number story, relocated from solvers to training stability.

9. Sources (credited; archived under `scadamale-book-sources`)

Golub–Van Loan. G. Golub and C. Van Loan, *Matrix Computations* (4th ed.), Johns Hopkins University Press, 2013. §3 (LU), §5 (QR), §10 (iterative methods), §11 (Krylov). — Authoritative reference for direct and iterative methods; condition number analysis; Gauss–Seidel convergence for SPD. *No text reproduced.*

Shewchuk. J. Shewchuk, *An introduction to the conjugate gradient method without the agonizing pain*, CMU Tech Report, 1994 (freely available). §9 (Eq. (52) in §9.2): convergence rate; the $\sqrt{\kappa}$ bound and the Chebyshev polynomial derivation; Appendix C3: optimality of the Chebyshev polynomials. — The clearest self-contained treatment; cited directly for the convergence theorem.

Demmel–Grigori–Hoemmen–Langou (TSQR). J. Demmel, L. Grigori, M. Hoemmen, J. Langou, *Communication-optimal parallel and sequential QR and LU factorizations*, SIAM J. Sci. Comput. 34(1), 2012 (preprint available). — Canonical TSQR reference; our §4.1 follows the tree structure described there. *No text reproduced.*

Drineas–Mahoney–Muthukrishnan. P. Drineas, M. Mahoney, S. Muthukrishnan, *Sampling algorithms for ℓ_2 regression and applications*, SODA 2006. — Source for the leverage-score sampling theorem in §4.2. The $O(d \log d/\varepsilon^2)$ sample complexity is from this paper (and the follow-up full version, JMLR 2011). *Our statement follows the paper's main theorem; no text reproduced.*

Sarlos (2006). T. Sarlos, *Improved approximation algorithms for large matrices via random projections*, FOCS 2006. — Random-projection sketch (JL-based) applied to least squares; complements the leverage-score approach in §4.2.

Boyd–Vandenberghe. S. Boyd and L. Vandenberghe, *Convex Optimization*, Cambridge University Press, 2004 (freely available online). §5.5: dual decomposition. §6: projection and proximal operators. — The projection theorem (§5) follows their exposition.

JáJá. J. JáJá, *An Introduction to Parallel Algorithms*, Addison-Wesley, 1992. §6: parallel matrix computations; Givens-rotation QR. — Parallel QR structure in §3.3 follows JáJá.

Wikipedia. *Conjugate gradient method*, *Gauss–Seidel method*, *Jacobi method*, *QR decomposition*, *Leverage (statistics)*, *Regularized least squares*. — For background and cross-references.

10. Exercises

Light (examined — assessed only for units in the examined core; U8 is deep but exercises appear in the Module-2 problem set):

E8.1 State the Brent bound for parallel Jacobi on a dense $n \times n$ system ($W = O(n^2)$ per sweep, $D = O(\log n)$ per sweep). For $n = 10^4$ and $p = 10^3$ processors, compute the two Brent terms. How many sweeps are needed to halve the error if $\kappa(A) = 100$ (use the Jacobi convergence rate $\rho(B) \approx 1 - O(1/\kappa)$)?

E8.2 A matrix A has condition number $\kappa = 10^6$. (a) How many CG iterations suffice to reduce the A-norm error by 10^{-6}? (b) Compare to the number of Jacobi iterations needed for the same reduction. (c) Why does CG achieve a $\sqrt{\kappa}$ rate rather than the κ rate of Richardson iteration?

E8.3 Construct a 4×4 sparse matrix A (symmetric positive definite) with adjacency graph that is a 4-cycle. Give a proper 2-colouring and describe one parallel Gauss–Seidel sweep explicitly (which updates are done simultaneously in each step?). Verify that under the proper 2-colouring, no two variables in the same colour class are coupled by a nonzero off-diagonal entry of A, and hence the updates within each colour class are mutually independent — so the 2-step parallel sweep equals the sequential Gauss–Seidel sweep in colour-major order.

E8.4 In leverage-score sampling: (a) Compute the leverage scores for a matrix with two rows $[1, 0]$ and $[0, 1]$, and say what the values mean for each row's direction. (b) Compute them for the matrix with rows $[1, 0], [1, 0], [0, 1]$: show $\ell_1 = \ell_2 = 1/2$ while $\ell_3 = 1$. Explain the lesson: DUPLICATED rows SHARE the leverage of their common direction, while a direction carried by a single row concentrates leverage 1 on it — which is exactly why leverage-score sampling must keep rare-direction rows to preserve the least-squares objective.

Deep (Module-3 / optional):

E8.5 *TSQR communication.* Show that TSQR on a binary tree of depth $\log_2 P$ machines, each with m/P rows of an $m \times n$ matrix, communicates $O(Pn^2)$ values in *total* across the tree (the $P - 1$ internal-node reductions each ship one $O(n^2)$ upper-triangular R factor), while any single machine sends only $O(n^2 \log P)$ values along its root path. Compare the total to the $O(mn)$ cost of communicating the full matrix. For what values of m, n, P does TSQR achieve a factor m/n communication reduction?

E8.6 *Projection theorem → normal equations.* Let $\mathcal{C} = \mathrm{col}(A)$ for full-rank $A \in \mathbb{R}^{m \times n}$. Use the variational characterisation of $\Pi_{\mathcal{C}}(b)$ to derive the normal equations $A^T A x^* = A^T b$. Show that the condition is equivalent to $A^T(Ax^* - b) = 0$ (residual perpendicular to column space).

E8.7 *Leverage score computation.* Prove that $\sum_{i=1}^{m} \ell_i(A) = n$ for any $A \in \mathbb{R}^{m \times n}$ with rank n. Prove $\ell_i \in [0, 1]$ for all i. Then: for the matrix $A = I_n \in \mathbb{R}^{n \times n}$ (identity), compute all leverage scores. For a rank-1 matrix $A = uv^T$ with $\|u\| = \|v\| = 1$, compute the leverage scores.

Chapter 15

09 gd sgd

U9 — Optimisation I: GD/SGD and friends

The ScaDaMaLe book, unit 9 (examined core). Our own exposition; original sources credited in §10.

Instantiating the three-lens invariant. The U0 box for this unit:

> **RAM:** gradient descent is a sequential iterative algorithm; the cost measure is the *iteration count* to reach an ϵ-accurate point, where each iteration costs one full gradient evaluation ($O(nd)$ operations for a sum of n losses on d-dimensional problems: $O(d)$ per example). **PRAM:** a gradient for a sum-loss $f(x) = \frac{1}{n}\sum_i f_i(x)$ is itself a reduction; the full-batch gradient computes in $W = O(nd)$, $D = O(\log n + d)$ using U1's tree. Minibatch degree of parallelism $= B$ (the batch size); depth per step $= O(\log B + d)$. **DPRAM/MPC:** data-parallel SGD partitions the n examples across machines; each step costs one all-reduce of the gradient vector (U1's primitive; $O(d)$ bytes per step); the parameter-server and ring-all-reduce architectures trade latency for bandwidth differently.

Entry kit — what this chapter assumes; follow a link if you need the memory jog (map: Figure 1): F.M3 (smoothness and strong convexity — the rate hypotheses) • F.S2 (concentration, for the stochastic-gradient noise)

1. Intuition — why gradient descent, and what "smooth" buys

Optimisation is the engine of machine learning. Training a logistic-regression model, a neural network, or a recommender system all reduce to: *find* $x^* = \arg\min_{x\in\mathbb{R}^d} f(x)$.

The simplest iterative method: stand at x_k, move in the direction of steepest descent.

$$x_{k+1} = x_k - \alpha_k \nabla f(x_k).$$

Two questions immediately arise: *how big a step* α_k*?* and *how fast does this converge?* Both answers hinge on a single structural property of f: **smoothness**. Smoothness says the gradient cannot change faster than a Lipschitz constant L; it gives a quadratic *upper* bound on f near any point, and that bound is exactly the function you minimise in one step of GD with step $1/L$. The iteration count to reach an ϵ-accurate point then follows by a one-line telescoping argument. Adding *strong convexity* — a quadratic *lower* bound on f — converts the additive rate into a multiplicative (geometric / linear) rate.

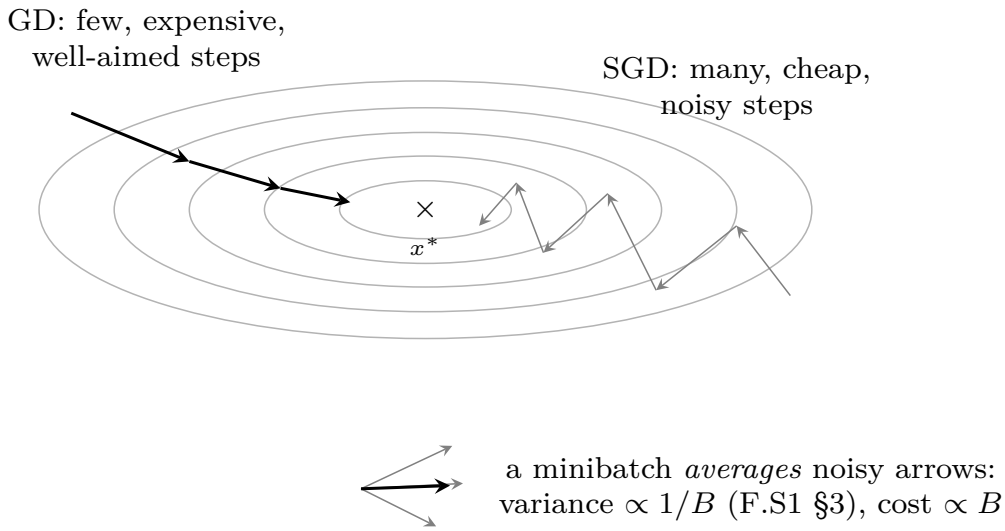


Figure 15.1: One quadratic bowl, two descents. The contour eccentricity is the condition number $\kappa = L/\mu$ (F.M3): the rounder the bowl, the straighter every path. Full gradient descent takes few well-aimed steps, each touching all n data points; SGD takes many cheap noisy ones, each touching one. The minibatch inset is the entire data-parallel story: averaging B stochastic arrows divides the variance by B at B times the cost — and computing that average across machines is a U1 allreduce, which is how this picture becomes distributed LLM training (U18).

This unit develops this chain precisely, then asks: what changes when you can only see *noisy* gradients (SGD), and what changes when you compute them *in parallel* (minibatch/distributed SGD)?

Figure 15.1 carries the unit's three ideas: conditioning, noise, and averaging.

2. RAM — gradient descent: smooth and strongly convex rates

2.1 Definitions

Definition 9.1 (L-smooth). $f : \mathbb{R}^d \to \mathbb{R}$ is *L-smooth* (for $L > 0$) if ∇f is L-Lipschitz:

$$\|\nabla f(x) - \nabla f(y)\| \le L\|x - y\| \quad \text{for all } x, y \in \mathbb{R}^d.$$

Definition 9.2 (μ-strongly convex). f is *μ-strongly convex* (for $\mu > 0$) if for all x, y and $t \in [0, 1]$,

$$f(ty + (1-t)x) \le tf(y) + (1-t)f(x) - \tfrac{\mu}{2}t(1-t)\|x - y\|^2.$$

Equivalently, $f(x) - \frac{\mu}{2}\|x\|^2$ is convex, or the Hessian (when it exists) satisfies $\nabla^2 f(x) \succeq \mu I$ everywhere.

Condition number. For an L-smooth, μ-strongly convex function, $\kappa := L/\mu \ge 1$ is the *condition number*. A well-conditioned problem ($\kappa \approx 1$) is geometrically easy; an ill-conditioned one ($\kappa \gg 1$) requires many iterations. The convergence rates below all depend on κ or its square root.

2.2 The descent lemma (quadratic upper bound) — PROVED

Lemma 9.3 (Descent lemma / quadratic upper bound). If f is L-smooth, then for all x, y:

$$f(y) \le f(x) + \nabla f(x)^\top (y - x) + \frac{L}{2}\|y - x\|^2.$$

Proof. Fix x, y and define $g(t) = f(x + t(y - x))$ for $t \in [0, 1]$. Then $g(1) = f(y)$ and $g(0) = f(x)$. By the fundamental theorem of calculus,

$$f(y) - f(x) = g(1) - g(0) = \int_0^1 g'(t)\, dt = \int_0^1 \nabla f(x + t(y - x))^\top (y - x)\, dt.$$

Subtract $\nabla f(x)^\top (y - x)$ from both sides:

$$f(y) - f(x) - \nabla f(x)^\top (y - x) = \int_0^1 \big[\nabla f(x + t(y - x)) - \nabla f(x)\big]^\top (y - x)\, dt.$$

Apply Cauchy–Schwarz and the L-Lipschitz condition on ∇f:

$$\begin{aligned}\big|\big[\nabla f(x + t(y - x)) - \nabla f(x)\big]^\top (y - x)\big| &\le \|\nabla f(x + t(y - x)) - \nabla f(x)\| \cdot \|y - x\| \\ &\le L\|t(y - x)\| \cdot \|y - x\| = Lt\|y - x\|^2.\end{aligned}$$

Integrating:

$$f(y) - f(x) - \nabla f(x)^\top (y - x) \le \int_0^1 Lt\|y - x\|^2\, dt = \frac{L}{2}\|y - x\|^2. \qquad \square$$

Interpretation. The descent lemma says that f lies *below* the quadratic $Q_x(y) := f(x) + \nabla f(x)^\top (y - x) + \frac{L}{2}\|y - x\|^2$ (a upward paraboloid tangent to the first-order model at x). The GD step with $\alpha = 1/L$ minimises Q_x exactly: $\arg\min_y Q_x(y) = x - \frac{1}{L}\nabla f(x)$, which is x_{k+1}. This is why $1/L$ is the "right" step size for smooth functions.

Key corollary (sufficient decrease). Substituting $y = x - \frac{1}{L}\nabla f(x)$ into Lemma 9.3:

$$f(x_{k+1}) \le f(x_k) - \frac{1}{2L}\|\nabla f(x_k)\|^2. \tag{SD}$$

2.3 GD convergence to an ϵ-critical point — PROVED

Theorem 9.4 (GD on L-smooth; iteration complexity). Let f be L-smooth with $\inf_x f(x) = f_* > -\infty$. Gradient descent with step $\alpha = 1/L$ satisfies: after K iterations,

$$\min_{0 \le k \le K-1} \|\nabla f(x_k)\|^2 \;\le\; \frac{2L(f(x_0) - f_*)}{K}.$$

In particular, to reach an ϵ-critical point ($\min_k \|\nabla f(x_k)\| \le \epsilon$), it suffices to run $K = \lceil 2L(f(x_0) - f_*)/\epsilon^2 \rceil$ iterations.

Proof. From (SD), summing from $k = 0$ to $K - 1$:

$$\sum_{k=0}^{K-1} \frac{1}{2L}\|\nabla f(x_k)\|^2 \;\le\; \sum_{k=0}^{K-1} \big[f(x_k) - f(x_{k+1})\big] = f(x_0) - f(x_K) \le f(x_0) - f_*.$$

The sum telescopes. Dividing by K and using $\min_k \|\nabla f(x_k)\|^2 \le \frac{1}{K}\sum_k \|\nabla f(x_k)\|^2$:

$$\min_{0 \le k \le K-1} \|\nabla f(x_k)\|^2 \le \frac{2L(f(x_0) - f_*)}{K}. \qquad \square$$

What the proof does NOT use. There is no convexity assumption: Theorem 9.4 holds for any L-smooth function, including non-convex objectives (e.g. neural network loss). The guarantee is to a *critical point* (gradient vanishes), not a global minimum.

2.4 Linear rate under strong convexity — PROVED

When f is also μ-strongly convex, GD converges at a *geometric* rate to the unique global minimiser x^*.

Theorem 9.5 (GD on L-smooth, μ-strongly convex; linear rate). Let f be L-smooth and μ-strongly convex with minimiser x^*. Gradient descent with step $\alpha = 1/L$ satisfies:

$$f(x_k) - f_* \;\le\; \Big(1 - \frac{\mu}{L}\Big)^k \big(f(x_0) - f_*\big).$$

To reach ϵ-optimality it suffices to run $K = \lceil \kappa \log((f(x_0) - f_*)/\epsilon) \rceil$ iterations, where $\kappa = L/\mu$ (since $(1 - 1/\kappa)^K \le e^{-K/\kappa}$).

Proof. Strong convexity implies the *Polyak–Łojasiewicz (PL) inequality*:

$$\|\nabla f(x)\|^2 \ge 2\mu(f(x) - f_*) \quad \text{for all } x. \tag{PL}$$

(Proof of PL: by μ-strong convexity, $f(x^*) \ge f(x) + \nabla f(x)^\top (x^* - x) + \frac{\mu}{2}\|x^* - x\|^2$; optimising the right side over $x^* - x$ gives $f_* \ge f(x) - \frac{1}{2\mu}\|\nabla f(x)\|^2$, i.e. (PL).*)

Combining (SD) and (PL):

$$f(x_{k+1}) \le f(x_k) - \frac{1}{2L}\|\nabla f(x_k)\|^2 \le f(x_k) - \frac{\mu}{L}(f(x_k) - f_*).$$

Subtracting f_* from both sides:

$$f(x_{k+1}) - f_* \le \Big(1 - \frac{\mu}{L}\Big)(f(x_k) - f_*).$$

Iterating gives the geometric decrease. □

Condition number κ. The factor $(1 - 1/\kappa)^k \approx e^{-k/\kappa}$: to reduce the error by a factor e^{-1} requires κ steps. Large κ = slow convergence. For least-squares with matrix A, $\kappa(A^\top A) = \kappa(A)^2$ — a notorious squaring of ill-conditioning. Preconditioning, randomised solvers (U8), and acceleration (§3.1 below) all address large κ.

Check your reading.

Q1. For an L-smooth function f, why is alpha = 1/L the canonical GD step size?
(A) it is the largest step that keeps the iterates bounded for every f (B) the GD step with alpha = 1/L exactly minimises the descent lemma's quadratic upper bound $Q_x(y)$ = f(x) + grad $f(x)^T$ (y - x) + (L/2)||y - x||^2, yielding the sufficient decrease $f(x_{k+1}) \le f(x_k)$ - (1/(2L)) ||grad $f(x_k)$||^2 (C) it makes GD converge in exactly one step on quadratics (D) any smaller step provably diverges

Q2. Theorem 9.4 gives $min_{k<K}$ ||grad $f(x_k)$||^2 $\le 2L(f(x_0)$ - $f_*)/K$ for GD with step 1/L. Which assumption does the proof NOT use?
(A) L-smoothness of f (B) f bounded below (f_* > -infinity) (C) convexity – none is needed: the telescoped sufficient-decrease argument works for any smooth f, including neural-network losses, but the guarantee is then only an epsilon-CRITICAL point, not a global minimum (D) the step size alpha = 1/L

3. The modern weaves — formal, compact

3.1 Nesterov momentum (accelerated GD)

The accelerated rate $O(1/K^2)$ in objective value for smooth convex problems (compared to GD's $O(1/K)$) is achieved by *Nesterov's accelerated gradient method* (1983). The update uses a momentum term that "looks ahead" along the trajectory: from x_0, with $y_0 = x_0$, for $k = 0, 1, 2, \ldots$

$$y_{k+1} = x_k - \frac{1}{L}\nabla f(x_k), \qquad x_{k+1} = y_{k+1} + \frac{k}{k+3}(y_{k+1} - y_k).$$

The momentum coefficient is 0 on the first step and tends to 1. (In the $k \geq 1$ indexing of Su, Boyd and Candès (2016), where the gradient-step points are called x_k, the same coefficient reads $(k-1)/(k+2)$.)

Theorem 9.6 (Nesterov acceleration; stated). For L-smooth convex f with minimiser x^*, Nesterov's method started from $y_0 = x_0$ satisfies, at the gradient-step points y_K:

$$f(y_K) - f_* \;\leq\; \frac{2L\|x_0 - x^*\|^2}{(K+1)^2} = O(L/K^2)$$

(Nesterov 1983; in this form, Su–Boyd–Candès 2016, eq. (6) with step $s = 1/L$). For μ-strongly convex f ($\kappa = L/\mu$), run the same two-line recursion with the *constant* momentum coefficient $\beta = (\sqrt{\kappa}-1)/(\sqrt{\kappa}+1)$ in place of $k/(k+3)$, started from $y_0 = x_0$. Then

$$f(y_K) - f_* \;\leq\; \Bigl(1 - \frac{1}{\sqrt{\kappa}}\Bigr)^K \Bigl(f(x_0) - f_* + \frac{\mu}{2}\|x_0 - x^*\|^2\Bigr) \;\leq\; 2\Bigl(1 - \frac{1}{\sqrt{\kappa}}\Bigr)^K (f(x_0) - f_*),$$

requiring $O(\sqrt{\kappa}\log(1/\epsilon))$ iterations — a factor $\sqrt{\kappa}$ improvement over GD's $O(\kappa\log(1/\epsilon))$.

Why acceleration is optimal. The Nemirovski–Yudin oracle lower bound (§3.5) shows that *any* first-order method (accessing f only through gradient/value oracles) requires at least $\Omega(\sqrt{\kappa}\log(1/\epsilon))$ iterations for smooth strongly convex problems. Nesterov's method matches this lower bound: it is *optimal* in the oracle model. The proof of the upper bound is a careful potential-function (Lyapunov) argument; we defer it to Module-3 (see E9.6). The *existence* of such an optimal rate is a non-trivial theorem in itself.

3.2 Adam and AdaGrad

AdaGrad (Duchi–Hazan–Singer 2011). Per-coordinate adaptive step sizes: let $G_k = \sum_{j=0}^{k} \nabla f(x_j)\nabla f(x_j)^\top$ (sum of outer products). The update is

$$x_{k+1} = x_k - \frac{\eta}{\sqrt{\operatorname{diag}(G_k) + \epsilon_0}} \odot \nabla f(x_k),$$

where the square root and the division are taken coordinate by coordinate, $\odot$ is the Hadamard (coordinate-wise) product, and $\epsilon_0 > 0$ is a damping constant. The effect: coordinates with small historical gradients get a large effective step; coordinates with large gradients get a small step. Provably optimal in the online convex optimisation setting (§3.4 below) for sparse gradient problems.

Adam (Kingma–Ba 2015). Combines AdaGrad's adaptive learning rates with momentum. Maintain exponential moving averages of the gradient (m_k) and its elementwise square (v_k):

$$\begin{aligned}
m_{k+1} &= \beta_1 m_k + (1-\beta_1)\nabla f(x_k),\\
v_{k+1} &= \beta_2 v_k + (1-\beta_2)(\nabla f(x_k))^{\odot 2},\\
\hat{m}_{k+1} &= m_{k+1}/(1-\beta_1^{k+1}), \quad \hat{v}_{k+1} = v_{k+1}/(1-\beta_2^{k+1}),\\
x_{k+1} &= x_k - \frac{\eta}{\sqrt{\hat{v}_{k+1}} + \epsilon_0} \odot \hat{m}_{k+1}.
\end{aligned}$$

Default hyperparameters: $\beta_1 = 0.9$, $\beta_2 = 0.999$, $\epsilon_0 = 10^{-8}$. The bias-correction terms $\hat{m}, \hat{v}$ correct for the initialisation at zero.

What is and is not proved. Adam is widely used in deep learning; its theoretical understanding is incomplete. Reddi, Kale, and Kumar (2018) exhibit a *non-convergence example*: a simple convex online problem where Adam (with fixed $\beta_2 < 1$) provably does not converge to the

optimum. The fix (AMSGrad) replaces v_k with a running maximum, restoring convergence; in the paper's experiments it performs as well as Adam or better. In practice, Adam with tuned hyperparameters often outperforms theoretically cleaner methods on non-convex neural-network objectives; the gap between theory and practice here is real and active.

3.3 Variance reduction: SVRG

Full-batch GD uses the exact gradient but costs $O(n)$ per step. Vanilla SGD uses one noisy gradient at $O(1)$ per step but has irreducible noise: with fixed step, it oscillates; with decaying step $\gamma_k = O(1/k)$, it converges but slowly ($O(1/k)$ rate). Can we keep a linear rate while paying far fewer than n gradient evaluations per step on average? Yes — via *variance reduction*.

Algorithm 9.7 (SVRG; Johnson–Zhang 2013). Fix an epoch length m and a constant step γ, and start from a snapshot $\tilde{x}$. Each epoch: compute the full gradient $\tilde{\mu} = \nabla f(\tilde{x}) = \frac{1}{n}\sum_i \nabla f_i(\tilde{x})$ (one pass over the data, n gradient evaluations) and set $x_0 = \tilde{x}$. Then for $k = 0, \dots, m-1$: sample i_k uniformly from $[n]$; compute the *variance-reduced gradient estimate*

$$g_k = \nabla f_{i_k}(x_k) - \nabla f_{i_k}(\tilde{x}) + \tilde{\mu};$$

update $x_{k+1} = x_k - \gamma g_k$. At the end of the epoch choose the new snapshot. The choice the convergence theorem analyses (*Option II*) is $\tilde{x} \leftarrow x_k$ for k drawn uniformly from $\{0, \dots, m-1\}$. The paper's experiments use *Option I*, $\tilde{x} \leftarrow x_m$, which its theorem does not cover.

Why it works. The key identity: $\mathbb{E}_{i_k}[g_k] = \nabla f(x_k)$ — the estimator is unbiased. When each f_i is convex and L-smooth, its second moment satisfies $\mathbb{E}_{i_k}\|g_k\|^2 \le 4L\big[(f(x_k) - f_*) + (f(\tilde{x}) - f_*)\big]$ (Johnson and Zhang's proof), so it vanishes as x_k and $\tilde{x}$ approach x^*. This vanishing variance is what breaks the SGD noise floor and lets the step stay constant.

Theorem 9.8 (SVRG convergence; Johnson–Zhang 2013, Theorem 1; stated). Let $f = \frac{1}{n}\sum_i f_i$ with *each* f_i *convex* and L-smooth, and f μ-strongly convex with minimiser x^* ($\kappa = L/\mu$). Run SVRG with Option II, a constant step γ and an epoch length m large enough that

$$\rho := \frac{1}{\mu\gamma(1 - 2L\gamma)m} + \frac{2L\gamma}{1 - 2L\gamma} < 1$$

(this needs $\gamma < 1/(4L)$). Then the snapshots satisfy

$$\mathbb{E}[f(\tilde{x}^{(s)}) - f_*] \le \rho^s (f(\tilde{x}^{(0)}) - f_*).$$

(The paper writes ψ_i, P, η, γ, $\tilde{w}$ for this book's f_i, f, γ, μ, $\tilde{x}$.) For example $\gamma = 1/(10L)$ and $m = 50\kappa$ give $2L\gamma/(1 - 2L\gamma) = 1/4$ and $1/(\mu\gamma(1 - 2L\gamma)m) = 12.5\kappa/m = 1/4$, so $\rho = 1/2$: the epoch length must be sufficiently large, $m \ge c\kappa$ for a constant c.

Cost. An epoch costs n gradient evaluations for $\tilde{\mu}$ plus two per inner step, $n + 2m$ in all ($n + m$ if the $\nabla f_i(\tilde{x})$ are stored). With $m = \Theta(\kappa)$ and $\rho = 1/2$, $O(\log(1/\epsilon))$ epochs reach ϵ-accuracy: SVRG needs $O\big((n + \kappa)\log(1/\epsilon)\big)$ component-gradient evaluations, against $O(n\kappa \log(1/\epsilon))$ for full-batch GD ($O(\kappa \log(1/\epsilon))$ iterations at n evaluations each) — a saving of a factor of about $\min(n, \kappa)$. Per inner step the amortised cost is $(n + 2m)/m = 2 + n/m$ (or $1 + n/m$ with stored gradients), which is $O(1)$ only if $m \gtrsim n$; Johnson and Zhang take $m = 2n$ in their convex experiments.

3.4 Online GD and multiplicative weights

The online learning framing generalises SGD to an adversarial setting.

Online gradient descent (Zinkevich 2003). At each round $t = 1, \dots, T$: the learner chooses $x_t \in \mathcal{X}$ (a convex set); an adversary reveals a convex loss ℓ_t; the learner suffers $\ell_t(x_t)$. The *regret* against the best fixed point $x^* = \arg\min_{x \in \mathcal{X}} \sum_t \ell_t(x)$ is

$$R_T = \sum_{t=1}^{T} \ell_t(x_t) - \sum_{t=1}^{T} \ell_t(x^*).$$

OGD updates: $x_{t+1} = \Pi_{\mathcal{X}}\big(x_t - \eta \nabla \ell_t(x_t)\big)$, where $\Pi_{\mathcal{X}}$ is Euclidean projection onto $\mathcal{X}$.

Theorem 9.9 (OGD regret; proof sketch). With step $\eta = D/(G\sqrt{T})$ (where $D = \operatorname{diam}(\mathcal{X})$ and G bounds $\|\nabla \ell_t\|$):

$$R_T \le DG\sqrt{T}.$$

Three-line proof sketch. By convexity, $\ell_t(x_t) - \ell_t(x^*) \le \nabla \ell_t(x_t)^\top (x_t - x^*)$. Expand $\|x_{t+1} - x^*\|^2 \le \|x_t - \eta \nabla \ell_t(x_t) - x^*\|^2$ (projection contracts distance) to get a *regret-telescoping* inequality: $\nabla \ell_t^\top (x_t - x^*) \le \frac{1}{2\eta}(\|x_t - x^*\|^2 - \|x_{t+1} - x^*\|^2) + \frac{\eta}{2}\|\nabla \ell_t\|^2$. Summing over t, the differences telescope; the gradient-norm terms sum to $\frac{\eta G^2 T}{2}$; with $\|x_1 - x^*\| \le D$ the total is at most $\frac{D^2}{2\eta} + \frac{\eta G^2 T}{2}$, and $\eta = D/(G\sqrt{T})$ makes each term $DG\sqrt{T}/2$, so $R_T \le DG\sqrt{T}$. $\square$ (Full proof: E9.5.)

Corollary. Average regret $R_T/T \le DG/\sqrt{T} \to 0$ — OGD is *no-regret*. The connection to SGD: let the losses $\ell_t(x) = f_{i_t}(x)$ be drawn i.i.d. Then $\mathbb{E}[\ell_t(x_t)] = \mathbb{E}[f(x_t)]$, because the index i_t is drawn after x_t is fixed. The regret proof holds against any fixed comparator in $\mathcal{X}$, in particular the minimiser of f over $\mathcal{X}$, with value f_*; and convexity (Jensen) moves the average inside f. So the averaged iterate $\bar{x}_T = \frac{1}{T}\sum_t x_t$ satisfies $\mathbb{E}[f(\bar{x}_T)] - f_* \le DG/\sqrt{T}$ — the online-to-batch conversion. The guarantee is in objective value: it bounds the gap, not the distance $\|\bar{x}_T - x^*\|$.

Multiplicative weights (Arora–Hazan–Kale 2012 survey). For the *experts problem*: n experts, adversarial losses $\ell_t \in [0,1]^n$, learner plays a mixture $p_t \in \Delta_n$. The MW update: $w_{t+1}(i) = w_t(i)(1-\epsilon)^{\ell_t(i)}$, $p_{t+1} = w_{t+1}/\|w_{t+1}\|_1$.

Theorem 9.10 (MW regret; stated). With $\epsilon = \sqrt{\ln n / T}$:

$$\sum_{t=1}^{T} p_t^\top \ell_t - \min_i \sum_{t=1}^{T} \ell_t(i) \;\le\; 2\sqrt{T \ln n}.$$

MW is the multiplicative-update analogue of OGD in the simplex; it is the meta-algorithm behind boosting, game-theoretic learning, and LP solving. The regret bound grows as $O(\sqrt{T \ln n})$: only logarithmic in the number of experts. MW is the algorithmic heart of the Hedge/AdaBoost family.

3.5 Oracle lower bounds (Nemirovski–Yudin)

Theorem 9.11 (First-order oracle lower bound; stated). In the *first-order oracle model* (each query to f reveals $(f(x), \nabla f(x))$), any deterministic algorithm requires at least

$$\Omega\Big(\sqrt{\kappa}\,\log\big(1/\epsilon\big)\Big)$$

queries to find an ϵ-optimal point of an L-smooth, μ-strongly convex function.

For smooth convex (not strongly convex) functions: any first-order method requires at least $\Omega(1/\sqrt{\epsilon})$ queries to find an ϵ-optimal point.

Proof idea (smooth convex lower bound). Construct an adversarial "hard function" $f_p(x) = \frac{L}{8}\big(x_1^2 + \sum_{i=1}^{p-1}(x_i - x_{i+1})^2 + x_p^2 - 2x_1\big)$ on $\mathbb{R}^p$. (The prefactor is $L/8$, not $L/4$: the Hessian of the

bracket is $2A_p$, where A_p is the $p \times p$ tridiagonal matrix with 2 on the diagonal and -1 off it; its largest eigenvalue $4 + 4\cos(\pi/(p+1))$ tends to 8 as $p \to \infty$, so $L/8$ is the scaling that makes f_p L-smooth for every p; $L/4$ would give a $2L$-smooth function.) Setting $\nabla f_p = 0$ gives $A_p x^* = e_1$, whose solution is $x_i^* = 1 - i/(p+1)$. For a method started at the origin whose iterates stay in the span of the gradients seen so far, each query uncovers at most one new coordinate (because A_p is tridiagonal), so after k queries the iterate lies in $\mathrm{span}\{e_1, \ldots, e_k\}$. With $p = 2k+1$ the unreached coordinates contribute an irreducible gap $\geq 3L\|x_0 - x^*\|^2/(32(k+1)^2) = \Omega(L/k^2)$ (E9.7). Taking $k \approx \sqrt{L/\epsilon}$ (unit initial distance) proves the lower bound. For the strongly convex case, the construction uses a tridiagonal quadratic family; the lower bound $\Omega(\sqrt{\kappa}\log(1/\epsilon))$ matches Nesterov's accelerated rate exactly.

Upshot. The Nemirovski–Yudin lower bound is the *optimisation minimax theorem*: it plays the same role in optimisation theory that the Cramér–Rao bound plays in estimation (U20) and that communication lower bounds play in distributed computing (U6/U9 DPRAM section). You cannot beat $\sqrt{\kappa}$ with first-order methods, no matter how clever. Newton's method escapes this lower bound by querying second-order information (the Hessian) — at the cost of $O(d^2)$ memory and a linear solve per step. Quasi-Newton methods such as L-BFGS do not escape it: they build their curvature estimates from differences of gradients, so they query only gradients (their iterates even stay in the span of the gradients seen, the class of the construction above), and the lower bound applies to them.

Check your reading.

Q3. Nesterov's accelerated method needs O(sqrt(kappa) log(1/epsilon)) iterations on L-smooth, mu-strongly convex problems. In what sense is this OPTIMAL?
(A) it is the fastest possible among all algorithms, including Newton-type second-order methods (B) it has the lowest per-iteration arithmetic cost of any gradient method (C) it provably outperforms Adam on every problem (D) it matches the Nemirovski-Yudin lower bound: any deterministic method accessing f only through a first-order oracle (values and gradients) needs Omega(sqrt(kappa) log(1/epsilon)) queries

4. SGD — stochastic gradient and minibatch

4.1 The stochastic gradient oracle

In large-scale ML, the objective is a finite sum $f(x) = \frac{1}{n}\sum_{i=1}^n f_i(x)$ (each f_i is the loss on example i). Computing $\nabla f(x) = \frac{1}{n}\sum_i \nabla f_i(x)$ costs $O(n)$ per step. For $n = 10^9$ (standard web-scale), this is prohibitive.

Stochastic gradient descent (Robbins–Monro 1951). Sample i_k uniformly at random from $[n]$; use $g_k = \nabla f_{i_k}(x_k)$ as the gradient estimate.

The oracle assumptions: (U) *Unbiasedness:* $\mathbb{E}[g_k \mid x_k] = \nabla f(x_k)$. (B) *Bounded second moment on the iterates:* $\mathbb{E}[\|g_k\|^2 \mid x_k] \leq G^2$ for some $G < \infty$, at every point of a set that contains all the iterates x_k.

(U) holds when $g_k = \nabla f_{i_k}(x_k)$: $\mathbb{E}[\nabla f_{i_k}(x_k)] = \frac{1}{n}\sum_i \nabla f_i(x_k) = \nabla f(x_k)$. (B) cannot be asked of all of $\mathbb{R}^d$ once f is μ-strongly convex: by Jensen and strong convexity (§4.2), $\mathbb{E}[\|g_k\|^2 \mid x_k] \geq \|\nabla f(x_k)\|^2 \geq \mu^2\|x_k - x^*\|^2$, which is unbounded. So (B) is stated on a set the iterates never leave, together with a condition that keeps them there. The standard case is a λ-regularised loss, $f_i(x) = \ell_i(x) + \frac{\lambda}{2}\|x\|^2$ with each ℓ_i convex and $\|\nabla \ell_i\| \leq R$ everywhere (the logistic loss with $\|a_i\| \leq R$, as in Lab 5). The SGD step is $x_{k+1} = (1 - \gamma_k\lambda)x_k - \gamma_k \nabla \ell_{i_k}(x_k)$, so if $\|x_k\| \leq R/\lambda$ and $\gamma_k \leq 1/\lambda$, then $\|x_{k+1}\| \leq (1-\gamma_k\lambda)R/\lambda + \gamma_k R = R/\lambda$. Started in the ball $\|x\| \leq R/\lambda$ (e.g. at $x_1 = 0$), every iterate stays in it, and on the ball $\|\nabla f_i(x)\| \leq R + \lambda\|x\| \leq 2R$: (B) holds with $G = 2R$. The step condition holds for the step $\gamma_k = 1/(\mu k)$ of Theorem 9.12 with $\mu = \lambda$, from $k = 1$ on, since $\gamma_k\lambda = 1/k \leq 1$.

4.2 SGD convergence — STATED with proof sketch

Theorem 9.12 (SGD on μ-strongly convex; $O(1/k)$ rate; stated with sketch). Let $f = \frac{1}{n}\sum_i f_i$ be μ-strongly convex with minimiser x^*. SGD with step $\gamma_k = 1/(\mu k)$ and oracle satisfying (U)-(B) satisfies, for all $k \geq 1$:

$$\mathbb{E}[\|x_k - x^*\|^2] \;\leq\; \frac{2G^2}{\mu^2 k}.$$

Proof sketch (induction). Write $e_k = \mathbb{E}[\|x_k - x^*\|^2]$. Expand $\|x_{k+1} - x^*\|^2 = \|x_k - \gamma_k g_k - x^*\|^2$ and take expectations:

$$e_{k+1} = e_k - 2\gamma_k \mathbb{E}[\langle g_k, x_k - x^*\rangle] + \gamma_k^2 \mathbb{E}[\|g_k\|^2].$$

By unbiasedness (U), $\mathbb{E}[\langle g_k, x_k - x^*\rangle] = \mathbb{E}[\langle \nabla f(x_k), x_k - x^*\rangle]$. By strong convexity, $\langle \nabla f(x), x - x^*\rangle \geq \mu\|x - x^*\|^2$ (follows from the strong convexity definition). By (B), the second moment term is $\leq \gamma_k^2 G^2$. So:

$$e_{k+1} \leq (1 - 2\gamma_k\mu)e_k + \gamma_k^2 G^2.$$

With $\gamma_k = 1/(\mu k)$: $1 - 2\gamma_k\mu = 1 - 2/k$ and $\gamma_k^2 G^2 = G^2/(\mu^2 k^2)$. Put $C = 2G^2/\mu^2$; we show $e_k \leq C/k$ by induction.

Base case $k = 1$. By (U), Jensen and (B), $\|\nabla f(x_1)\|^2 \leq \mathbb{E}[\|g_1\|^2 \mid x_1] \leq G^2$; by strong convexity and Cauchy–Schwarz, $\mu\|x_1 - x^*\|^2 \leq \langle \nabla f(x_1), x_1 - x^*\rangle \leq \|\nabla f(x_1)\|\,\|x_1 - x^*\|$. Hence $\mu\|x_1 - x^*\| \leq G$ and $e_1 \leq G^2/\mu^2 \leq C$. (This is why the starting distance does not appear in the bound: (B) already controls it.)

Step $k = 1 \to 2$. Here $1 - 2/k = -1$ is negative, so the hypothesis cannot be used; dropping the term $-e_1 \leq 0$ gives $e_2 \leq G^2/\mu^2 = C/2$ directly.

Step $k \to k+1$ for $k \geq 2$. Now $1 - 2/k \geq 0$, so $e_k \leq C/k$ gives

$$e_{k+1} \leq \Big(1 - \frac{2}{k}\Big)\frac{C}{k} + \frac{C}{2k^2} = \frac{C(2k-3)}{2k^2} \leq \frac{C}{k+1},$$

the last because $(2k-3)(k+1) = 2k^2 - k - 3 < 2k^2$. □

The factor 2 cannot be dropped. Started at $x_1 = x^*$, the first step gives $e_2 = \gamma_1^2\,\mathbb{E}\|g_1\|^2 = \mathbb{E}\|g_1\|^2/\mu^2$, which equals $G^2/\mu^2 = C/2$ when the oracle's second moment at x^* is G^2: the bound is tight at $k = 2$, and a constant $\max\{\|x_1 - x^*\|^2, G^2/\mu^2\} = G^2/\mu^2$ in place of C would be false there.

The noise floor. For a fixed step γ the recurrence reads $e_{k+1} \leq (1 - 2\gamma\mu)e_k + \gamma^2 G^2$: the error contracts geometrically down to a floor at its fixed point $\gamma G^2/(2\mu)$, and SGD oscillates at that scale around x^*. To reach $\mathbb{E}\|x_k - x^*\|^2 \leq \epsilon$ the floor must lie below ϵ, so $\gamma \lesssim \epsilon\mu/G^2$, and the contraction factor $1 - 2\gamma\mu$ then needs about $\log(1/\epsilon)/(2\gamma\mu)$ steps: $O\big(G^2/(\mu^2\epsilon)\cdot\log(1/\epsilon)\big)$ steps in all — worse than GD's $O(\kappa\log(1/\epsilon))$ for any fixed ϵ. SGD's advantage is the $O(1)$ per-step cost; the tradeoff is favourable when n is large and ϵ is modest.

Figure 15.2 puts numbers on this. Against a target of 0.01, the step $\gamma = 10^{-3}$ stalls at its floor of 0.05; the decaying step $\gamma_k = 1/(\mu k)$ reaches the target at $k = 2 \times 10^4$, where $200/k = 0.01$, and $\gamma = 10^{-4}$ a little later, above its own floor of 0.005.

4.3 Minibatching: variance reduction via averaging

Rather than one sample per step, draw a *minibatch* $B \subset [n]$ of size $|B| = b$:

$$g_k^B = \frac{1}{b}\sum_{i \in B} \nabla f_i(x_k).$$

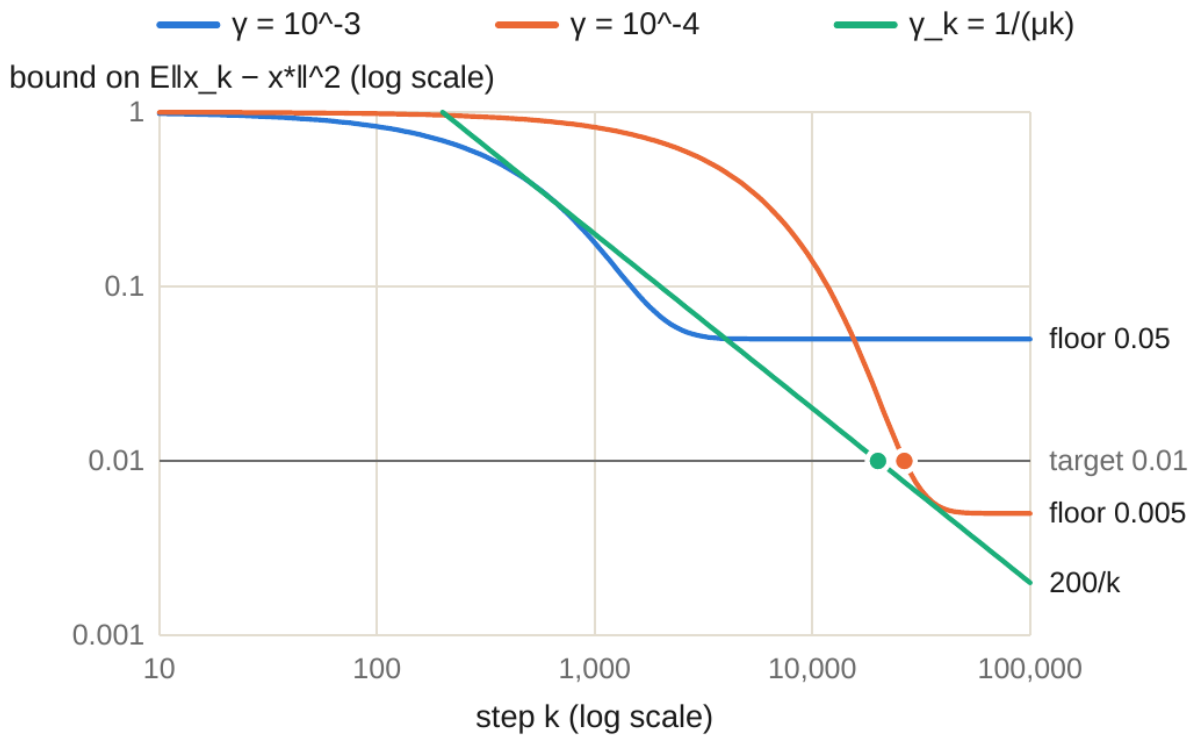


Figure 15.2: One noise floor in numbers: a constant step γ stalls at a floor that scales with γ; the decaying step $\gamma_k = 1/(\mu k)$ keeps falling.

By independence, $\mathrm{Var}(g_k^B) = \mathrm{Var}(g_k)/b$: variance reduces linearly with batch size. In Theorem 9.12 only the variance part of G^2 is divided by b ($\mathbb{E}\|g_k^B\|^2 = \|\nabla f(x_k)\|^2 + \mathrm{Var}(g_k)/b$), so near the optimum, where the variance dominates, the noise floor drops roughly by the factor b. Put the other way: for a target accuracy ϵ the step can be about b times larger, and the iteration count of §4.2 becomes roughly $O\big(G^2/(b\mu^2\epsilon)\cdot\log(1/\epsilon)\big)$; but each step costs b gradient evaluations, so the total number of gradient evaluations is unchanged.

Linear scaling of the step. The same accounting, one doubling at a time: doubling b halves the noise, and doubling γ with it keeps the noise floor where it was, while the contraction factor $1-2\gamma\mu$ reaches that floor in half the iterations. Each iteration now costs twice the gradient evaluations, so the total is unchanged; the gain is wall-clock time when the b evaluations of a step run on b processors: twice the processors, half the iterations, the same total work. This *linear-scaling* rule (step proportional to batch) lasts only while the step is small. Even exact GD diverges along a direction of curvature L once $\gamma > 2/L$ (the error there is multiplied by $1-\gamma L$ each step), so the step cannot grow past order $1/L$ however large the batch: beyond that point, the large-batch regime, a bigger batch no longer buys a bigger step.

Step-size schedules. Common choices: - *Constant:* $\gamma_k = \gamma$. Converges to a noise-floor neighbourhood; good for non-convex. - $1/k$ *decay (Robbins–Monro):* optimal rate for strongly convex (Theorem 9.12); slow in practice. - *Cosine annealing:* $\gamma_k = \frac{\gamma_{\max}}{2}(1+\cos(\pi k/K))$. Widely used in deep learning; no theory, strong practice. - *Warm-up + cosine:* large-batch training starts with a linear warm-up, ramping γ up from a small value, to avoid instability from a large γ early in training (the linear-scaling rule makes the target step large when b is large).

Check your reading.
Q4. Minibatch SGD averages b i.i.d. sampled gradients per step: g_k^B = (1/b) $sum_(i$ in B) grad $f_i(x_k)$. Compared with b = 1, the variance of the gradient estimate:
(A) divides by b – only the variance part of G^2 shrinks, so near x^* the fixed-step noise floor gamma $G^2/(2$ mu) drops roughly by the factor b (B) divides by sqrt(b), because standard deviations average (C) is unchanged – the samples share the same x_k (D) divides by b^2, because variance is quadratic

5. PRAM — parallel gradient computation

A full-batch gradient $\nabla f(x) = \frac{1}{n}\sum_i \nabla f_i(x)$ is a *reduction*: combine n gradient vectors of dimension d with the associative $\oplus$ = vector addition. By U1's analysis:

$$W = O(nd), \qquad D = O(\log n + d).$$

(Work: n gradient evaluations each costing $O(d)$; depth: a balanced tree of n additions has $O(\log n)$ levels, and at each level the d-coordinate additions are done in parallel, so depth per level is $O(1)$; the $O(d)$ term comes from the gradient evaluations themselves run in parallel, each of depth $O(d)$.)

With p processors and Brent's bound:

$$T_p \le \frac{nd}{p} + \log n + d.$$

Speed-up is linear while the work term dominates, $nd/p \gtrsim \log n + d$, i.e. for $p \lesssim nd/(\log n + d) = W/D$ — the *parallelism* of one GD step (about n when $d \gg \log n$). Beyond this, depth dominates and adding processors yields no further gain. This is the *synchronous minibatch* regime in practice: batch size $b = p$, one processor per example.

6. DPRAM / MPC — distributed optimisation architectures

6.1 Synchronous data-parallel SGD

Partition the n training examples across p workers (each holds n/p examples). Each step: 1. Each worker j computes local gradient $g_j = \frac{1}{|B_j|}\sum_{i\in B_j} \nabla f_i(x_k)$ on its local minibatch B_j. Cost: $O(|B_j| \cdot d)$ local work. 2. **All-reduce** the gradients: the global gradient is $g = \frac{1}{p}\sum_j g_j$ — a reduction of p vectors of dimension d. By U1's distributed reduction, this costs $O(\log p)$ communication rounds, each moving a full d-vector along the critical path: in the tree-reduce pattern the time per step is $O(\log p)$ message latencies plus the transfer time of $O(d \log p)$ floats. (The total volume a tree moves is only about $2(p-1)d$ floats, reduce plus broadcast; the $d \log p$ is critical-path time, not bytes.) 3. Each worker updates $x_{k+1} = x_k - \gamma g$.

The **ring-all-reduce** architecture (used by NCCL, Horovod) is bandwidth-optimal for the all-reduce primitive: in a ring of p workers, send-receive volume per worker per step is $\frac{2(p-1)}{p} \cdot d \cdot$ sizeof(float) bytes — approaching $2d$ floats ($8d$ bytes in single precision) as $p \to \infty$, independent of p. This is the same primitive as U1's distributed scan/reduction; the ring topology makes the *bandwidth* cost independent of p (unlike the tree-reduce, whose critical path carries a full d-vector at each of its $O(\log p)$ levels). The *latency* cost of the ring is $O(p)$ message steps against the tree's $O(\log p)$. With per-message latency α and per-float transfer time β, a ring all-reduce takes $2(p-1)\alpha + 2\frac{p-1}{p}\beta d$ and a tree (reduce, then broadcast) $2\log_2 p\,(\alpha + \beta d)$: rings win when bandwidth dominates (large d), trees when latency dominates (small d, or very large p for a fixed d, since the crossover d grows like $(\alpha/\beta)\, p/\log_2 p$).

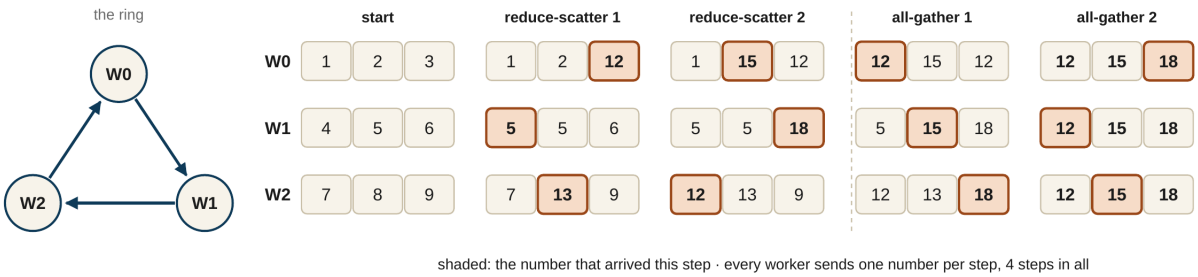


Figure 15.3: A ring all-reduce with $p = 3$ workers, step by step.

Byte budget per training step (single-precision, d-parameter model, p workers, ring-all-reduce):

$$\text{Bytes per worker per step} = 2 \cdot \frac{p-1}{p} \cdot d \cdot 4 \;\approx\; 8d \text{ bytes.}$$

For a 70B-parameter model ($d = 7 \times 10^{10}$), this is ≈ 560 GB per worker per step — the dominant communication cost in large-scale training, before tensor/pipeline parallelism.

Figure 15.3 walks a ring all-reduce with $p = 3$ workers, one step at a time.

6.2 Parameter-server architecture

An alternative to all-reduce: a *parameter server* (PS) holds the global model x; workers send gradient updates to the PS; the PS aggregates and broadcasts the updated model. Communication pattern per step: each worker sends d floats to PS; PS broadcasts d floats back to p workers. Total: $2pd$ floats per step — about the same as the ring's total, p workers times $2\frac{p-1}{p}d$, i.e. $2(p-1)d$ floats. The difference is where the traffic goes: all $2pd$ floats cross the server's link, p times what a ring worker's link carries ($\approx 2d$). The PS is a *bottleneck* at large scale; ring-all-reduce is preferred for bandwidth-bound jobs. The PS architecture remains useful for *sparse gradient* problems (e.g. embedding tables in recommendation systems) where only a small fraction of the d parameters are updated per step.

6.3 HOGWILD! — asynchronous lock-free SGD

Setting. $f(x) = \sum_{i=1}^{n} f_i(x)$ on $x \in \mathbb{R}^d$, where each term f_i depends only on the coordinates in its support $e_i \subseteq [d]$. The *conflict graph* G_C has the n terms as vertices and an edge (i, j) iff $e_i \cap e_j \neq \emptyset$. Write $G_i = n\nabla f_i$ (zero outside e_i) for the sampled gradient, an unbiased estimate of ∇f. HOGWILD! runs p processors on a shared x, without locks: each repeatedly samples i uniformly, reads x_{e_i}, computes G_i, and subtracts γG_i from the coordinates in e_i, one atomic scalar update per coordinate. (The analysis updates one random coordinate $v \in e_i$ per step, with step $\gamma|e_i|$.) The sparsity of the problem enters through three numbers: - $\Omega := \max_i |e_i|$, the most coordinates any one term touches; - $\Delta := \max_{v \in [d]} |\{i : v \in e_i\}|/n$, the largest fraction of terms touching one coordinate; - $\rho := \max_i |\{j : e_j \cap e_i \neq \emptyset\}|/n$, the largest fraction of terms that share a coordinate with one term, i.e. $(1 +$ the maximum degree of $G_C)/n$.

Theorem 9.13 (HOGWILD!; Recht–Ré–Wright–Niu 2011, Proposition 4.1; stated). Let each f_i be convex, ∇f be L-Lipschitz, f be μ-strongly convex with minimiser x^*, and the sampled gradients satisfy $\|G_i(x)\| \leq M$. *Assume* that the lag between when a gradient is computed and when it is used — the number of updates written in between — is always at most τ. Take the step

$$\gamma = \frac{\vartheta\, \epsilon\, \mu}{2LM^2\Omega\big(1 + 6\rho\tau + 4\tau^2\Omega\Delta^{1/2}\big)}$$

for some $\epsilon > 0$ and $\vartheta \in (0,1)$, and let $D_0 = \|x_0 - x^*\|^2$. Then after

$$k \;\geq\; \frac{2LM^2\Omega\big(1 + 6\tau\rho + 6\tau^2\Omega\Delta^{1/2}\big)\log(LD_0/\epsilon)}{\mu^2\vartheta\epsilon}$$

component updates, $\mathbb{E}[f(x_k) - f_*] \leq \epsilon$. (The constants 4 and 6 in the τ^2 terms are as printed in the paper, which writes c for μ and n for this book's d.)

With $\tau = 0$ this is the serial SGD rate. The bound keeps nearly the same rate for $\tau = o(d^{1/4})$ (the paper's $\tau = o(n^{1/4})$, its n being the number of coordinates), since, as the paper puts it, ρ and Δ are typically $o(1/d)$. The lag τ is a *hypothesis*, not a consequence of sparsity: it counts the updates that run concurrently, so it grows with the number of processors p, not with the number of examples. Hence a near-linear speed-up while p stays below about $d^{1/4}$.

Proof idea. Recht et al. show that $a_j = \frac{1}{2}\mathbb{E}\|x_j - x^*\|^2$ obeys a serial-style recursion $a_{j+1} \leq (1 - c_r\gamma)(a_j - a_\infty) + a_\infty$, with an "effective curvature" $c_r < \mu$ and a floor a_∞ (of order $\gamma M^2/\mu$) whose constants grow with τ, ρ, Δ and Ω because of the stale reads; the count of updates then follows as for constant-step SGD (§4.2). The cleaner *noisy-serial* reading follows Mania et al. (2017): the asynchronous run *is* serial SGD whose gradients are evaluated at *perturbed iterates* $\hat{x}_j$ (the possibly inconsistent values a processor actually read) instead of x_j, $x_{j+1} = x_j - \gamma G_{i_j}(\hat{x}_j)$; the perturbation $\hat{x}_j - x_j$ consists of at most τ overlapping updates, and sparsity keeps its effect small. It is the iterates that are perturbed, not the objective.

When is the conflict graph sparse? In sparse ML problems: (a) *Sparse linear regression:* each example $f_i(x) = (y_i - a_i^\top x)^2/2$ where a_i has $s \ll d$ non-zeros; two examples conflict only if their non-zero supports overlap. For random support sets of size s in dimension d, two supports meet with probability about s^2/d, so the expected degree in G_C is about ns^2/d: $\Omega = s$, ρ of order s^2/d and Δ of order s/d — sparse when $s^2/d \ll 1$. (b) *Matrix factorisation / recommender systems:* example (u, i) only overlaps with examples sharing user u or item i; for power-law user/item distributions, the conflict graph is effectively sparse. In dense problems (every term touches every coordinate, as for the weights of a fully-connected network), $\Omega = d$ and $\rho = \Delta = 1$: the factor $1 + 6\rho\tau + 4\tau^2\Omega\Delta^{1/2}$ becomes $1 + 6\tau + 4\tau^2 d$, so the step must shrink, and the update count grow, by a factor of order $\tau^2 d$ — far more than the factor p that the processors supply — and the theorem predicts no speed-up.

Staleness tradeoffs. Asynchronous methods trade staleness for the elimination of synchronisation barriers. Each worker proceeds at its own speed; fast workers are not blocked by slow ones (stragglers). The cost: stale gradients add an error that grows with τ (through $6\rho\tau + 4\tau^2\Omega\Delta^{1/2}$ in Theorem 9.13); with many processors and dense models, this can dominate the variance of the stochastic gradient itself. Modern large-scale systems typically use *bounded staleness* protocols (e.g. stale-synchronous parallel, SSP) or switch to synchronous all-reduce (§6.1).

6.4 Parameter averaging (local SGD)

Synchronous SGD (§6.1) communicates once per step. *Parameter averaging*, also called *local SGD*, communicates less often: every worker starts a round from the same model x, runs H steps of minibatch SGD on its own examples, and the p resulting models are averaged, $x \leftarrow \frac{1}{p}\sum_j x_j$. Here a *round* is one combination of the workers' vectors: one all-reduce of a d-vector, whatever its internal tree levels or ring steps (in Spark's driver pattern, one vector up per worker and a broadcast of the new model down). A round costs the same communication as one step of §6.1, but now carries H steps of computation per worker. With $H = 1$ it coincides with synchronous minibatch SGD with batch $p \cdot b$: averaging one step taken from the same point is one step with the averaged gradient. For larger H the workers' models move apart between averages, each towards the optimum of its own examples, and each carries the noise of SGD with a constant

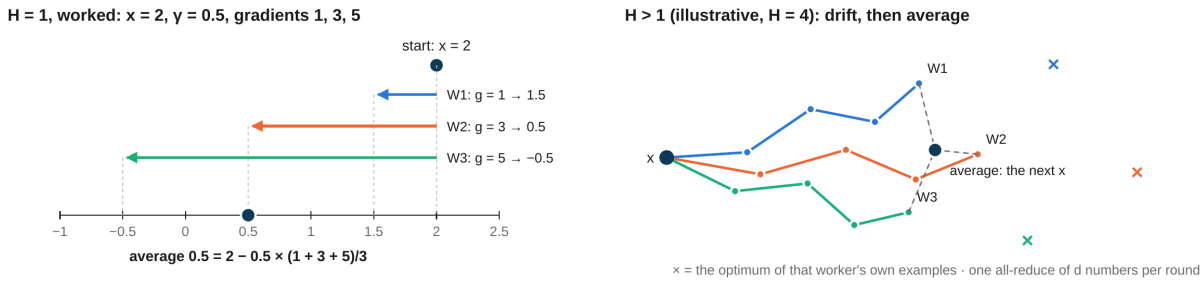


Figure 15.4: Parameter averaging (local SGD): the workers talk once every H steps.

step (§4.3). When the workers' data come from the same distribution the drift is small; when they differ (as in federated learning, where each device holds its own users' data) the drift limits how large H can usefully be. Lab 5 measures what larger H buys for a fixed number of rounds.

Figure 15.4 draws the schedule: H local steps, then one average.

Check your reading.

Q5. Synchronous data-parallel SGD on p = 64 workers, model dimension d = 10^6, single precision (4 bytes). Per worker per step, the ring-all-reduce moves approximately:
(A) 4 MB – one copy of the gradient (B) 8 MB – 2 (p-1)/p * d * 4 bytes, essentially 8d bytes independent of p (C) 24 MB – d * 4 * log2(p) bytes (D) 512 MB – 2 p d * 4 bytes

Q6. A 70B-parameter model is trained with synchronous data-parallel Adam across many accelerators. In two or three sentences, map this onto the unit's vocabulary: what object crosses the network each step and how many bytes (use the unit's own figure), why ring-all-reduce beat the parameter server here, and what this unit says about going asynchronous instead.

7. The invariant (U9 instantiation)

RAM: L-smooth f: descent lemma (quadratic upper bound, proved) → GD with $\alpha = 1/L$: $2L(f_0 - f_*)/\epsilon^2$ steps to ϵ-critical point (proved, telescoping). Add μ-strong convexity: linear rate $(1 - \mu/L)^k$ (proved via PL inequality). Condition number $\kappa = L/\mu$ governs cost. **Modern weaves (formal):** Nesterov $\sqrt{\kappa}$ rate (stated; optimal by Nemirovski–Yudin lower bound); Adam/AdaGrad (rules stated; non-convergence example cited; theory gap is real); SVRG (stated; each f_i convex, epoch length $m \geq c\kappa$: linear rate with $O((n+\kappa)\log(1/\epsilon))$ gradient evaluations against GD's $O(n\kappa \log(1/\epsilon))$); OGD (regret $O(\sqrt{T})$, 3-line proof sketch); MW (regret $O(\sqrt{T \ln n})$, stated).

PRAM: a gradient step is a reduction ($W = O(nd)$, $D = O(\log n + d)$); Brent gives $T_p \leq nd/p + \log n + d$; linear speed-up while $p \lesssim nd/(\log n + d)$. Minibatch parallelism = U1's reduction; depth per step $O(\log B + d)$.

DPRAM/MPC: synchronous data-parallel SGD = gradient-reduction via all-reduce (U1's primitive) + distributed gradient averaging; ring-all-reduce moves $\approx 2d$ floats ($8d$ bytes, FP32) per worker per step, independent of p (bandwidth-optimal); a parameter server moves about the same total, $2pd$ floats per step, but all of it through the server's link — p times a ring worker's load (PS bottleneck at large p). Parameter averaging (local SGD) = one all-reduce per H local steps; limited by the constant-step noise floor and, for non-identically distributed data, by drift. HOGWILD! = lock-free SGD: assuming a lag of at most τ updates and a sparse problem (small Ω, Δ, ρ), nearly the serial SGD rate for $\tau = o(d^{1/4})$ (Recht et al., Proposition 4.1; noisy-serial view: Mania et al.); dense models lose the guarantee. Oracle lower bounds: first-order methods cannot beat $\Omega(\sqrt{\kappa}\log(1/\epsilon))$ — optimisation

has minimax limits, mirroring the Cramér–Rao (U20) and Scheffé–Yatracos (U15) limits for estimation.

Check your reading.
Q7. Fill the U9 row of the invariant box from memory: RAM iteration complexities (smooth; smooth + strongly convex; SGD), PRAM work/depth of a full-batch gradient with the Brent bound, and the DPRAM byte accounting of the two synchronous architectures.

8. Lab (Lab 5 — distributed SGD and parameter averaging)

Lab 5: the `lab5-distributed-sgd/` folder of the course repository, and the lab chapter Lab 5 in the Laboratory exercises part; PySpark in the course container, laptop-runnable, the same receipt pattern as the other labs.

What the lab runs. Logistic regression with an L2 penalty ($\lambda = 10^{-3}$) on $n = 2 \times 10^5$ synthetic examples with $d = 50$ features of unequal scales, split over 8 workers. L, $\mu = \lambda$ and the exact f_* (by Newton's method) are computed first. *Experiment 1:* full-batch GD with step $1/L$, one round of communication (a `treeAggregate` of the partial gradients) per step, against the bound of Theorem 9.5. *Experiment 2:* parameter averaging (§6.4) with $H \in \{1, 10, 100\}$ local minibatch steps per round, for a budget of 10 rounds. *Experiment 3:* the bytes per round, and the ring all-reduce budget of §6.1 for a 7-billion-parameter model. Both are computed from the formulas ($p \cdot d$ doubles per round, $8 \cdot 50 \cdot 8 = 3200$ bytes; $2\frac{p-1}{p} \cdot d \cdot 4$ bytes for the ring), not measured on the network, although the receipt prints them on its `MEASURED` lines. The same algorithms run on Ray (actors) or with MPI/NCCL all-reduce; the lab uses Spark.

Deliverable: the receipt, with one or two sentences on each of: why Theorem 9.5's bound is so pessimistic for this problem, and why GD slows down (compare λ with the curvature the lab measures at the start, at step 40 and at the optimum); and why $H = 10$ needs 10 rounds to reach the gap that $H = 100$ reaches after one, and why $H = 100$ then stops improving.

The experiments below are the original, larger design of this lab (a Ray cluster, $n = 10^6$, $d = 500$); they remain as optional deep-dive projects.

Original setup (optional; not provided — it needs your own Ray and PyTorch installation, which the course image does not include). A `docker-compose.yml` with a Ray head, 2 workers and Jupyter. Logistic regression on a synthetic binary classification dataset: $n = 10^6$ examples, $d = 500$ features, Gaussian design with controlled condition number κ.

Experiment 1 — GD vs SGD vs minibatch: step-size and rate.

Generate data with $\kappa \in \{2, 10, 100\}$. Implement: - Full-batch GD with step $1/L$ (compute L from data). - SGD with $\gamma_k = 1/(\mu k)$ (compute μ from data). - Minibatch SGD with batch sizes $b \in \{1, 10, 100, 1000\}$ and a constant step γ_b tuned separately for each b (a grid search).

For each, record $f(x_k) - f_*$ vs wall-clock time and vs gradient-evaluation count.

Questions: 1. Plot $f(x_k) - f_*$ vs k (log scale) for GD. Does the slope match $\log(1 - \mu/L)$? 2. For minibatch SGD, plot the optimal step size vs batch size. Is the linear-scaling rule $\gamma_b \propto b$ consistent with your measurements? 3. For which κ does GD outperform SGD in wall-clock time? Explain from the theory.

Experiment 2 — Sync minibatch vs HOGWILD!-style async (Ray).

Using Ray's actor model, implement: - *Sync:* a parameter-server actor receives gradients from all workers before each update. - *Async (HOGWILD!-style):* workers read and write parameters without synchronisation (Ray shared memory or object store); updates applied immediately without waiting for other workers.

Compare convergence (iterations to reach ϵ) vs wall-clock time for $p \in \{1, 2, 4\}$ workers on the sparse logistic regression dataset (95% zero features — conflict graph is sparse).

Questions: 1. Does async outperform sync in wall-clock time? At what p? 2. Change to a dense dataset (all features non-zero). Does HOGWILD!-style async still converge? 3. Explain both observations via the conflict graph.

Experiment 3 — All-reduce byte measurement.

Using Ray collective (or `torch.distributed` if available in the image), run a ring-all-reduce of a d-dimensional gradient vector for $d \in \{10^3, 10^4, 10^5\}$ and $p \in \{2, 4\}$ workers. Measure bytes transferred per worker per all-reduce.

Questions: 1. For each (d, p), compute the theoretical $2(p-1)/p \cdot d \cdot 4$ bytes. Do your measurements agree? 2. At what d does communication dominate compute per step? (Compare the time for the gradient computation vs the all-reduce.) 3. Project: for a 7B-parameter model (FP32, $d = 7 \times 10^9$), what is the all-reduce byte volume per step? At 100 Gbps network, what is the communication time? (For comparison, redo with FP16: byte volume halves.)

Experiment 4 — Step-size/batch sweep vs theory curves.

For minibatch SGD with the strongly convex logistic objective: sweep batch size b from 1 to $n/10$ and record the minimum objective value achieved within a fixed gradient-evaluation budget (e.g. $10n$ gradient evaluations total). Overlay the theoretical prediction from Theorem 9.12 and §4.3: only the variance part of G^2 shrinks by $1/b$, so near x^* the fixed-step floor drops by about b, while the same budget buys b times fewer steps.

Deliverable of the optional deep dive: notebook + half-page cost-accounting for each experiment in the three-lens vocabulary.

9. Lean certificate target

Target (U9): the descent lemma.

The claim to certify: for an L-smooth function f, the GD step with $\alpha = 1/L$ achieves sufficient decrease

$$f(x - (1/L)\nabla f(x)) \;\le\; f(x) - \frac{1}{2L}\|\nabla f(x)\|^2.$$

This is a *polynomial inequality* once f is specified as a quadratic (the hard case is general f; the quadratic case is the genuinely certifiable schematic target).

```
-- SCHEMATIC (names align to BMLSPar.lean conventions at cert time):
-- f : ℝ^d → ℝ is L-smooth (modelled as a quadratic Q(x) = x^T A x / 2 - b^T x,
-- with A ≽ μI and A ≼ LI, i.e. condition number κ = L/μ).
-- Mathlib's ‖·‖ on Fin d → ℝ is the sup norm: the descent lemma stays true
  under it,
-- the PL inequality does not. So ‖∇f x‖ ^ 2 is written as the dot product gradf
  x ⬝ᵥ gradf x.
open scoped Matrix MatrixOrder   -- `*ᵥ`; `≤` on matrices is the Loewner order
```

```
theorem descent_lemma_quadratic (L μ : ℝ) (hL : 0 < L) (hμ : 0 < μ) (hLμ : μ ≤
  L)
    (A : Matrix (Fin d) (Fin d) ℝ) (hA_lower : μ • 1 ≤ A) (hA_upper : A ≤ L •
      1)
    (b x : Fin d → ℝ) :
    let f := fun x => (x ⬝ᵥ A *ᵥ x) / 2 - b ⬝ᵥ x
    let gradf := fun x => A *ᵥ x - b
    let x' := x - (1/L) • gradf x
    f x' ≤ f x - (1 / (2 * L)) * (gradf x ⬝ᵥ gradf x) := by
  sorry  -- SCHEMATIC: polynomial inequality; certifiable for concrete quadratics
```

What makes this certifiable. For a fixed quadratic f, the descent lemma becomes a polynomial inequality in the entries of x, A, b — decidable by sum-of-squares (SOS) methods. The general smooth-function case requires a formal model of smoothness as a Lean typeclass; that is the Module-3 extension target.

Exercise U9.L1 (light). Load `BMLSPar.lean`. State the descent lemma for a 1D quadratic $f(x) = ax^2/2 - bx$ ($a > 0$): verify that GD with step $1/a$ satisfies $f(x_{k+1}) \le f(x_k) - \frac{1}{2a}(ax_k - b)^2$ and prove it algebraically (by expanding both sides).

Exercise U9.L2 (deep). Formalise the PL inequality in Lean for a quadratic f and use it to prove the one-step linear-rate bound $f(x_{k+1}) - f_* \le (1 - \mu/L)(f(x_k) - f_*)$.

LLM lens — where this unit lands on the running workload (U0L):
Training an LLM is this unit at its largest: cross-entropy loss (U0L §1) minimised by Adam-flavoured SGD over minibatches, with the data/tensor/pipeline TRAINING parallelism trichotomy mirroring U0L §4's inference one. Every scaling-law curve (U18) is a plot of this unit's convergence behaviour against compute.

10. Sources (credited; archived under `scadamale-book-sources`)

Robbins, H. and Monro, S. (1951). *A Stochastic Approximation Method.* Annals of Mathematical Statistics 22(3):400–407. The original SGD paper; introduces the stochastic approximation framework and the diminishing step-size rule. Our §4.1–4.2 follows the Robbins–Monro setup.

Polyak, B.T. (1964). *Some methods of speeding up the convergence of iteration methods.* USSR Computational Mathematics and Mathematical Physics 4(5):1–17. As its title says, a paper on speeding up iterative methods. The PL (Polyak–Łojasiewicz) inequality used in the proof of Theorem 9.5 is not from this paper: it is from Polyak (1963) and Łojasiewicz (1963).

Nesterov, Yu. (1983). *A method of solving a convex programming problem with convergence rate $O(1/k^2)$.* Soviet Mathematics Doklady 27(2):372–376. The original accelerated gradient paper; source of Theorem 9.6. See also: Nesterov, Yu. (2004). *Introductory Lectures on Convex Optimization: A Basic Course.* Springer.

Su, W., Boyd, S., and Candès, E.J. (2016). *A Differential Equation for Modeling Nesterov's Accelerated Gradient Method: Theory and Insights.* JMLR 17 (arXiv:1503.01243). Eq. (1) is the recursion of §3.1 (in $k \ge 1$ indexing) and eq. (6) the bound of Theorem 9.6.

Nemirovski, A. and Yudin, D. (1983). *Problem Complexity and Method Efficiency in Optimization.* Wiley–Interscience. The foundational oracle lower bounds (Theorem 9.11).

Duchi, J., Hazan, E., and Singer, Y. (2011). *Adaptive Subgradient Methods for Online Learning and Stochastic Optimization.* JMLR 12:2121–2159. The AdaGrad update rule (§3.2).

Kingma, D.P. and Ba, J. (2015). *Adam: A Method for Stochastic Optimization.* ICLR 2015. The Adam update rules stated in §3.2.

Reddi, S.J., Kale, S., and Kumar, S. (2018). *On the Convergence of Adam and Beyond.* ICLR 2018. The non-convergence example for Adam (§3.2); introduces AMSGrad.

Johnson, R. and Zhang, T. (2013). *Accelerating Stochastic Gradient Descent using Predictive Variance Reduction.* NeurIPS 2013. The SVRG algorithm and convergence theorem (§3.3/Theorem 9.8, their Theorem 1 with Option II).

Recht, B., Re, C., Wright, S., and Niu, F. (2011). *HOGWILD!: A Lock-Free Approach to Parallelizing Stochastic Gradient Descent.* NeurIPS 2011 (arXiv:1106.5730). Theorem 9.13 is their Proposition 4.1; the sparsity statistics Ω, Δ, ρ are their (2.6) (§6.3).

Mania, H., Pan, X., Papailiopoulos, D., Recht, B., Ramchandran, K., and Jordan, M.I. (2017). *Perturbed Iterate Analysis for Asynchronous Stochastic Optimization.* SIAM Journal on Optimization 27(4). The noisy-serial (perturbed-iterate) reading of HOGWILD! in §6.3.

Zinkevich, M. (2003). *Online Convex Programming and Generalized Infinitesimal Gradient Ascent.* ICML 2003. The OGD regret bound (Theorem 9.9/§3.4).

Arora, S., Hazan, E., and Kale, S. (2012). *The Multiplicative Weights Update Method: a Meta-Algorithm and Applications.* Theory of Computing 8:121–164. The MW regret bound (Theorem 9.10/§3.4) and the survey of applications.

Boyd, S. and Vandenberghe, L. (2004). *Convex Optimization.* Cambridge University Press. Free at https://web.stanford.edu/~boyd/cvxbook/. Background for convexity, strong convexity, and Lagrangian duality (U19 prerequisite).

11. Exercises

Light (examined core):

E9.1 State Lemma 9.3 (descent lemma) for the function $f(x) = \frac{1}{2}\|Ax - b\|^2$ ($A \in \mathbb{R}^{m\times d}$). (a) What is the smoothness constant L in terms of A? (b) Starting from $x_0 = 0$, write out the first two GD iterates with step $1/L$. (c) How many iterations are needed to reach $\|\nabla f(x_k)\| \le 10^{-3}$ if $f(x_0) - f_* = 100$ and $L = 10$?

E9.2 Prove the Polyak–Łojasiewicz inequality: for μ-strongly convex f with minimiser x^*, show $\|\nabla f(x)\|^2 \ge 2\mu(f(x) - f_*)$ for all x. (*Hint:* use the definition of strong convexity with $y = x^*$ and optimise over x^*.)

E9.3 In the three-lens vocabulary, account for the cost of one gradient-descent step for $f(x) = \frac{1}{n}\sum_i \ell(x; z_i)$ ($z_i \in \mathbb{R}^d$): (a) RAM: how many operations per step? (b) PRAM: what is (W, D) for the gradient evaluation? State the Brent bound T_p. (c) DPRAM: for synchronous data-parallel SGD with ring-all-reduce on p workers, what is the byte volume per step? For $d = 10^6$ and $p = 64$, compute this in GB.

E9.4 Adam update rules: (a) Write out the Adam update for step k with hyperparameters $(\beta_1, \beta_2, \eta, \epsilon_0)$. (b) What is the bias-correction term for m_k, and why is it needed at step $k = 1$? (c) State (without proof) the non-convergence example of Reddi et al. (2018): what property of the update causes non-convergence?

Deep (Module-3 / optional):

E9.5 (OGD regret, full proof). Prove Theorem 9.9 in full: fill in all three steps of the sketch and verify the choice of $\eta = D/(G\sqrt{T})$ that minimises the regret bound. Extend to the case where $\mathcal{X} = \mathbb{R}^d$ (no projection needed): what happens to the regret?

E9.6 (Nesterov potential, strongly convex, Module-3). Let f be L-smooth and μ-strongly convex, $\kappa = L/\mu$, $\alpha = 1/\sqrt{\kappa}$, and run Nesterov's method (§3.1) with the constant momentum coefficient $\beta = (1-\alpha)/(1+\alpha)$ from $y_0 = x_0$. Define z_k from the momentum recursion by $x_k = (y_k + \alpha z_k)/(1+\alpha)$. (a) Show that $z_{k+1} = (1-\alpha)z_k + \alpha x_k - \frac{\alpha}{\mu}\nabla f(x_k)$. (b) With the Lyapunov function $E_k = f(y_k) - f_* + \frac{\mu}{2}\|z_k - x^*\|^2$, show $E_{k+1} \le (1 - 1/\sqrt{\kappa})E_k$ using the descent lemma, strong convexity at x_k (evaluated at x^*) and convexity (evaluated at y_k). (c) Conclude $f(y_K) - f_* \le 2(1-1/\sqrt{\kappa})^K (f(x_0) - f_*)$.

E9.7 (Oracle lower bound, adversarial construction). For the smooth convex lower bound (§3.5): (a) Verify that the tridiagonal quadratic family f_p is convex and L-smooth, and that its minimiser is $x_i^* = 1 - i/(p+1)$ with $f_p^* = -\frac{L}{8}\frac{p}{p+1}$. (b) Show that for any method started at $x_0 = 0$ whose iterates satisfy $x_j \in \mathrm{span}\{\nabla f(x_0), \dots, \nabla f(x_{j-1})\}$, the k-th iterate on $f = f_{2k+1}$ satisfies $f(x_k) - f_* \ge \frac{3L\|x_0 - x^*\|^2}{32(k+1)^2}$, i.e. $\Omega(L/k^2)$ for unit initial distance. (c) Use this to prove the lower bound $K = \Omega(1/\sqrt{\epsilon})$ for smooth convex functions.

Chapter 16

0P proof checkers

U0P — Proof-checkers and the mathesis method

The ScaDaMaLe book, unit 0P (examined core, week 1). Our own exposition; original sources credited in §7.

The one idea of this unit: *a proof is a program; the compiler is the judge.* When the Lean kernel accepts a proof term, it has applied a finite, auditable algorithm — the same algorithm on every machine, with no appeal to intuition. The rest of this course relies on that fact: every theorem statement you meet will carry one of three honest flags — CERTIFIED (the kernel checked it), STATED (the statement is precise but the proof is not yet mechanised), or UNCERTIFIED (informal or hand-wavy). You should treat these flags differently. This unit explains why.

Entry kit — what this chapter assumes; follow a link if you need the memory jog (map: Figure 1):
F.C3 (logic, induction, and how to read a Lean statement)

1. Intuition — what does "the checker accepted it" actually mean?

Start with a simple claim: the addition of natural numbers commutes. Every mathematician accepts this immediately — it is obvious from the axioms of arithmetic. But "obvious from the axioms" is doing hidden work: we are mentally running through a short induction, compressing the steps, and recognising the pattern.

A *proof checker* makes the hidden steps explicit. You write down every logical move in a formal language. The checker applies one rule at a time — no intuition, no pattern recognition — and either the whole chain type-checks or the checker refuses. The certificate you get when it accepts is not a certificate of "this seems right"; it is a certificate of "every step is derivable in the stated axiom system."

Why does that matter for a course on scalable data science? Because the theorems that govern distributed systems — Brent's scheduling bound, Scheffé's identity, the density-estimation rates — are the same theorems you use to argue a cluster budget or a watermark-detection guarantee. When the proof is machine-checked, "the theorem holds" is a reproducible computation, not a shared convention. The cluster budget rests on the same footing as the build that checked it.

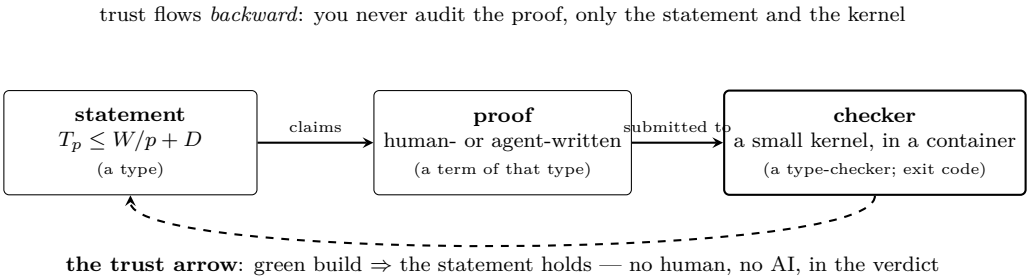


Figure 16.1: The proof triune. A theorem is a type, a proof is a term of that type, and the checker is a type-checker whose verdict is a build exit code. Who (or what) wrote the proof is irrelevant to the verdict — which is why this book lets agents write proofs and lets no one, human or machine, overrule the kernel. The reader's whole job (F.C3) is the top-left box: read the statement precisely, then follow the trust arrow.

Figure 16.1 is the whole unit's trust model in three boxes.

Check your reading.
Q1. "The checker accepted the proof" means, precisely:
(A) a committee of experts reviewed and approved it (B) the proof term type-checks against the statement in a small kernel, with no admitted holes (C) an AI judged the proof to be convincing (D) the proof compiled without warnings

2. What a machine-checked proof is

2.1 Proofs as programs — the kernel and the trust base

The proof assistant we use is **Lean 4** (Lean proof assistant) with the **Mathlib** community library (Mathlib).

Lean is built on the **Curry–Howard correspondence** (Curry–Howard): a proposition is a *type* and a proof of that proposition is a *program of that type*. Checking whether a proof is correct is therefore the same computation as checking whether a program type-checks. The checker is the **kernel** — a small, fixed piece of code (a few thousand lines, dwarfed by the millions of lines of Mathlib it can check). All of the high-level proof-writing conveniences (*tactics*) ultimately produce a low-level **proof term**, and only the kernel type-checks that term. You do not have to trust the tactics, the tactics library, or Mathlib — only the kernel. This "small trusted core checks everything" design is the **de Bruijn criterion** (proof assistant characteristics).

Your trust base is:

1. the kernel is correctly implemented,
2. the logical axioms you depend on are the ones you intended.

Point 2 is visible: `#print axioms <theorem>` lists the exact axioms a theorem depends on. Most of our certificates rest on the standard classical trio `propext`, `Classical.choice`, `Quot.sound` — the same foundation Mathlib uses throughout.

2.2 The sorry marker

Lean has one escape hatch: `sorry`. A file containing `sorry` may still type-check in a weak sense — Lean marks the result as having "axiom sorry" in its axiom footprint and reports a warning. Our rule throughout this course: **a certificate counts only when `sorry_count = 0`**. A file with a `sorry` is not checked; it is planned. We distinguish these in every coverage report.

2.3 LLM-free proof search

Finding a proof and checking a proof are different operations. *Checking* is always LLM-free: the kernel is the oracle. *Finding* a proof can also be LLM-free: Lean has tactics that mechanically search — `omega` (linear integer arithmetic), `decide` (decidable propositions), `norm_num`, `nlinarith`, `aesop`. Every certificate in our provedog library closes with exactly these. So "the computer proved it" can mean deterministic automation, not a language model. (A language model can draft a proof for a human to check — that is authoring assistance, optional and separately evaluated; the verification never depends on it.)

Check your reading.
Q2. In the propositions-as-types reading, WRITING a proof corresponds to:
(A) running a test suite until it passes (B) constructing a program (term) whose type is the proposition
(C) writing documentation for a function (D) measuring the empirical error rate

3. Reading a Lean 4 proof

3.1 A small example: commutativity of natural-number addition

Below is a complete, self-contained Lean 4 proof that addition of natural numbers commutes. This is not copied from anywhere; it is a small original illustration for this unit.

```
-- UOP illustration: Nat.add_comm stated and checked from first principles.
-- We use Lean's induction tactic; the proof matches the standard induction
-- argument taught in any foundations course.

theorem add_comm_UOP (n m : Nat) : n + m = m + n := by
  induction n with
  | zero      => simp [Nat.zero_add]
  | succ k ih => simp [Nat.succ_add, Nat.add_succ, ih]
```

Read this line by line.

- `theorem add_comm_UOP (n m : Nat) : n + m = m + n` — the *statement*: for all natural numbers `n` and `m`, `n + m = m + n`. The colon separates the type signature (the proposition) from the proof that follows.
- `by` — we enter *tactic mode*, where we build the proof incrementally.
- `induction n with` — perform induction on `n`. Lean splits the goal into two cases.
- `| zero` — base case: `n = 0`. The goal becomes `0 + m = m + 0`. `simp [Nat.zero_add]` closes it by rewriting with the lemma `Nat.zero_add : 0 + k = k` (itself proved in Mathlib).
- `| succ k ih` — inductive step: assume `k + m = m + k` (that is `ih`); prove `(k + 1) + m = m + (k + 1)`. `simp [Nat.succ_add, Nat.add_succ, ih]` rewrites with the successor lemmas and the induction hypothesis, closing the goal.

The kernel does not see the tactic names. It sees the proof term the tactics produced and checks that term against the type `n + m = m + n`. If every reduction rule applied correctly, the kernel reports success. No intuition; no convention; no tolerance for gaps.

Note: `Nat.add_comm` already exists in Mathlib. In real work we would write `exact Nat.add_comm n m` and be done in one line. The expanded version above is a pedagogical illustration of how a Lean proof is structured — base case, inductive step, the exact Mathlib lemmas invoked.

3.2 The axiom footprint

After saving the file, `#print axioms add_comm_U0P` reports:

```
add_comm_U0P uses axioms: propext, Classical.choice, Quot.sound
```

The classical trio — consistent with the rest of Mathlib. No `sorryAx`. The proof is clean.

Check your reading.
Q3. Open the course's Brent-bound certificate and, WITHOUT reading the proof body, write down in ordinary mathematics what the theorem statement says, including the exact form of every hypothesis.

4. Running the checker — the dockerised lab (L1)

4.1 Why docker

Lean + Mathlib are version-sensitive: a proof that checks under one toolchain version may not under another. Pinning everything in a Docker image makes the check reproducible on any machine — the same verdict on your laptop, on the T480, on a cluster node. (Reproducibility · Containerisation)

We pin in three layers: `lean-toolchain` fixes the exact Lean release; `lakefile.toml` specifies the Mathlib revision; `Dockerfile` bakes both into the `iasprovedog:lean-v4.32.0` image. You do not install Lean on your host. The container carries everything. (Lab 0 uses a different, smaller image: the public core-only Lean 4 checker `vakeworks/lean-checker`, enough for its one-line theorem but without Mathlib.)

4.2 The one command that matters

From the repo root, after the image is built:

```
bash provedog/build-in-docker.sh
```

Under the hood this runs the container, mounts the proof directory, downloads prebuilt Mathlib `.olean` files (the `lake exe cache get` step — what makes laptop checking feasible in minutes rather than hours), and then runs `lake build`. Exit code 0 means the kernel accepted every file. It does not, by itself, mean `sorry`-free: in Lean 4 a `sorry` is a warning, not an error (§2.2), and `lakefile.toml` does not turn warnings into errors, so a sorried file still exits 0. A certificate is clean when the build exits 0 *and* its `#print axioms` footprint shows no `sorryAx` (equivalently, the log has no `declaration uses 'sorry'` warning). Exit code non-zero means the checker refused something — and the log tells you exactly which step failed.

The script builds its image itself on the first run (then cached). To build it by hand:

```
docker build -t iasprovedog:lean-v4.32.0 provedog/
```

No accounts needed. No paid services. The only cost is your laptop's CPU and the one-time download.

4.3 Reading a real certificate: ScheffeFinite.lean

Open `provedog/IasCert/CMDE/ScheffeFinite.lean`. It proves Scheffé's identity — the fact that the total variation distance between two distributions equals half the L1 distance between their densities. This result underpins the density-estimation theory in U14 and U15. Read the

`theorem` statements. Find the `#print axioms` call and its output. Then deliberately break something — replace one tactic invocation with `sorry` — and rerun the checker. The build reports `sorryAx`; the certificate is no longer clean. This is the checker *earning its keep*: it refuses to certify what you have not proved.

Check your reading.

Q4. The course ships its proof-checker as a Docker container primarily so that:
(A) the proofs run faster than on bare metal (B) the verdict is reproducible on any student's laptop – same kernel, same libraries, same exit code (C) students cannot read the proof sources (D) the university's cluster is not needed

5. The cert ladder of this course

The course tracks machine-checked proofs on a **cert ladder**. Each rung is a result that matters for the subject, given as a Lean certificate in `provedog/`.

Rung	Unit	Certificate	Status
BMLSPar	U0	`IasCert/Paper/BMLSPar.lean` — Brent's $pT_p \le W + D(p-1)$ over $\mathbb{N}$, level by level	CERTIFIED ✓
Scan	U1	Blelloch scan work-optimality	solution file exists; not yet in lake build tree
Merge	U2	parallel merge depth	solution file exists; not yet in lake build tree
Graham G1	U3	list-scheduling $(2 - 1/p)$ bound	solution file exists; not yet in lake build tree
Reservoir	U4	reservoir sampling correctness	solution file exists; not yet in lake build tree
…	…	…	…
CMDE	U15	Scheffé–Yatracos; L1-MDE rate; TV = half-L1	CERTIFIED ✓

BMLSPar is your first contact with a real certificate. Its student copy, with the inductive step blanked, is the Week-2 theorem-checker exercise (`provedog/exercises/wasp-module1-brent/`); this week's Lab 0 runs the kernel only on the one-line theorem `1 + 1 = 2`. By the time you reach U3 you will see it cited again — the Graham bound proof uses the same scheduling-DAG argument the Brent bound formalises. By U15 you will see the CMDE certificates green in the same `lake build` output. The ladder makes progress on the course's own formalization visible as a coverage report, not as a claim.

The **honesty contract** of this book: every theorem statement is flagged.

- **CERTIFIED** — machine-checked, `sorry_count = 0`, axiom footprint disclosed. You can run the checker yourself and confirm.
- **STATED** — the statement is mathematically precise; the proof in the text is a correct informal argument; the Lean certificate is a target (future rung on the ladder).
- **UNCERTIFIED** — the claim is intuitive or informal; treat it as a conjecture until promoted.

No result is presented as proved unless its flag says CERTIFIED or the flag is explicitly stated as STATED with a proof sketch you can follow. This is the mathesis method applied to itself.

Check your reading.

Q5. A theorem in this book is marked CERTIFIED. What does that entitle you to conclude?
(A) every claim in that chapter is machine-checked (B) the exact certified statement (in the checker's normalisation) holds, resting only on the listed axioms – and nothing more (C) the algorithm it describes is the fastest known (D) the result has been peer-reviewed by a journal

6. Why machine-checking matters at scale

6.1 The cluster-budget argument

Brent's bound (U0 §3.4) says that a DAG computation of work W and depth D, run one level at a time on p processors, finishes within

$$T_p \;\le\; D + \frac{W - D}{p}.$$

What is CERTIFIED is the denominator-cleared form over the natural numbers, $p\,T_p \le W + D(p-1)$, which `BMLSPar.lean` proves for $T_p = \sum_i \lceil m_i/p \rceil$ and a list of level sizes $m_1, \ldots, m_D$ summing to W; dividing by $p > 0$ to reach the display above is a one-line step on paper. (The standard shorthand $T_p \le W/p + D$ is a corollary: since $(W - D)/p \le W/p$ for $D \ge 0$, the displayed form implies the looser one, and is tighter by D/p. However, the certificate takes the list of levels as given: that a DAG splits into D levels of ready operations is U0's informal step, and the greedy schedules of U0 §3.2 are not certified.)

Suppose you are deciding whether to purchase 64 additional nodes for a cluster running a distributed proof-checking pipeline over a book of 400 theorems. You model the work per theorem as w_i and the dependency depth as D. The certified bound tells you the schedule achieves $T_{64} \le T_\infty + (T_1 - T_\infty)/64$. If $T_1 = 400$ machine-hours and $T_\infty = 8$ machine-hours (the longest dependency chain of prerequisite theorems totals 8 hours), then $T_{64} \approx 14$ machine-hours — and no amount of additional nodes buys you below 8. That calculation rests on the correctness of Brent's bound. If the bound is wrong — or if you are relying on an informal version of it that omits a side condition — the budget decision is wrong too. A machine-checked proof removes that uncertainty: the bound is exactly what the certificate says, with the axiom footprint disclosed.

This is not a toy concern. A compute-cluster run schedules independent proof-checking jobs across nodes. The schedule is the BMLS schedule. Its correctness claim is `BMLSPar.lean`. One of the things the cluster checks is the very bound that justifies the schedule it runs under.

6.2 The watermark connection (one line)

The keyless watermark detector in U15 is a hypothesis test whose Type I and Type II error rates are controlled by Scheffé's identity — which is a CERTIFIED theorem in `ScheffeFinite.lean`. The security guarantee is as strong as the proof.

6.3 The same discipline, applied to making this course

The course material was drafted by digital agents under a human operator, and none of their output counted until a check passed: the Lean kernel for proofs, independent review for text, a build for documents, and a human for anything made public. The Lean kernel is one row of a larger table of checks that an agent cannot argue its way past. U17 §1.1 gives the table, a worked example, and the errors the checks caught.

LLM lens — where this unit lands on the running workload (U0L):
System claims about LLMs get this unit's taxonomy: a decode-throughput ceiling computed from measured bandwidth (U0L §2) is a checkable, even certifiable, COST claim; a vendor's "X tok/s" is benchmark-STATED; and a model-quality claim is contractible only via an agreed test (U16). The discipline you learned here — read the statement, check what the verdict rests on — applies verbatim to benchmark culture.

Check your reading.
Q6. Machine-checking is said to move trust rather than eliminate it. In two sentences: what do you still have to trust after a green build, and why is that a good trade?

7. Sources

- **Lean 4 documentation.** *The Lean 4 Reference Manual* and *Theorem Proving in Lean 4* (Avigad et al., open access): https://leanprover.github.io/lean4/doc/ · https://lean-lang.org/theorem_proving_in_lean4/
- **Mathlib.** The Lean community's library of formalised mathematics: https://leanprover-community.github.io/
- **de Bruijn, N.G. (1980).** *A survey of the project AUTOMATH.* In Seldin and Hindley (eds.), *To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism.* Academic Press. (Origin of the de Bruijn criterion for proof checkers.)
- **Curry–Howard correspondence.** Wikipedia: https://en.wikipedia.org/wiki/Curry-Howard_correspondence. For the deep connection between proof terms and typed programs.
- **Our provedog.** `provedog/` in this repo — the real dockerised checker: `Dockerfile`, `build-in-docker.sh`, `IasCert/Paper/BMLSPar.lean`, `IasCert/CMDE/ScheffeFinite.lean`, and the checker driver. The lab in §4 runs the real tool, not a simulation.

---

8. Exercises

Light (examined core):

E0P.1 *(Run the checker.)* From the repo root, build the `iasprovedog:lean-v4.32.0` image (or let the script build it) and run `bash provedog/build-in-docker.sh`. Report: did the build exit 0? How many modules were built? What does `lake build` print for a proof file that contains `sorry`? (One paragraph, backed by the actual output.)

E0P.2 *(Read a proof.)* Open `provedog/IasCert/Paper/BMLSPar.lean`. Identify: (a) the main `theorem` statement and the quantifiers it ranges over; (b) the tactic that closes the base case; (c) the axiom footprint reported by `#print axioms bmls_parallel_bound` (or equivalent). Write the answers as three labelled lines.

E0P.3 *(Fill one blank.)* The proof below has one `sorry` in place of the inductive-step tactic. Replace the `sorry` with a tactic that closes the goal, and verify the checker accepts:

```
theorem double_ge_U0P (n : Nat) : n ≤ n + n := by
  induction n with
  | zero      => omega
  | succ k ih => sorry
```

State which tactic you used and why it closes the goal (`omega` is the intended one-word answer — explain what decision procedure it implements and why this goal is in its fragment; `simp +arith` (the spelling Mathlib uses at our pin; older code writes `simp_arith`) or an explicit `Nat.succ_le_succ`-based term are also acceptable — if you used one of those, explain the extra steps it needed).

Deep (Module-3 material):

E0P.4 *(State a lemma.)* In Lean 4, write a `theorem` statement (header only — no proof required yet) for the following claim: *for all natural numbers n and k, $(n+k)+0 = n+k$.* Your statement

must type-check with a `sorry` proof. Then state — in one sentence — which Mathlib lemma (`#check <name>`) already proves this, and what the difference is between citing a Mathlib lemma and authoring your own certificate.

E0P.5 *(Extend a cert.)* The `BMLSPar.lean` certificate proves the Brent upper bound in the integer form $p \cdot T_p \leq W + D(p-1)$, that is $T_p \leq D + (W-D)/p$ (§6.1; the shorthand $T_p \leq W/p + D$ is its corollary). The matching lower bound is $T_p \geq \max(W/p, D)$. (a) State this as a Lean `theorem` (again, a `sorry` proof is fine for the statement exercise). (b) Write an informal proof of the lower bound in at most five sentences — one sentence per logical step. (c) Identify the gap: which step in your informal proof would require the most work to formalise in Lean, and why?

Chapter 17

11 mst cc

U11 — Graphs I: MST and connected components: RAM → PRAM → DPRAM

The ScaDaMaLe book, unit 11 (deep / optional core). Our own exposition; original sources credited in §9.

Instantiating the three-lens invariant. The U0 box for MST and connected components:

> **RAM:** minimum spanning tree (MST) costs $O(m \log n)$ — Kruskal's algorithm, grounded by the cut property. The interesting RAM question is the data structure (union-find; inverse-Ackermann amortised cost). **PRAM:** Borůvka's 1926 algorithm is the natural parallel MST: in each phase every component selects its lightest outgoing edge, edges are contracted. The number of components at least halves per phase, so $O(\log n)$ phases suffice. Work $W = O(m \log n)$, depth $D = O(\log^2 n)$. **DPRAM/MPC:** distributed Borůvka maps cleanly to MapReduce: each phase is one round (a `reduceByKey` for the per-component minimum edge, then a relabel join for contraction). Connected components via label propagation run in rounds proportional to the graph's diameter — the small-diameter dichotomy.

Entry kit — what this chapter assumes; follow a link if you need the memory jog (map: Figure 1):
F.C2 (graph vocabulary and hashing basics)

1. Intuition — spanning trees as consensus

A spanning tree of a connected graph $G = (V, E)$ is a subgraph that touches every vertex and contains no cycle — a minimal connected subgraph. The MST is the spanning tree of least total edge weight. Why do we care at scale?

- **Cluster topology.** Network providers and distributed systems use MST-like structures to decide how data flows between machines.
- **Image segmentation and scientific meshes.** Minimum spanning forests partition a similarity graph.
- **The algorithmic interest.** MST is an exact problem (not an approximation) that is *easy* sequentially but requires non-trivial ideas to parallelise. The gap between the sequential and parallel worlds is instructive: Kruskal's sequential algorithm is inherently serial (it processes edges one by one); Borůvka's 1926 algorithm — historically the *first* MST algorithm — is inherently parallel.

Connected components (CC) is the prerequisite: before asking for the MST of a connected graph, one must verify (or enforce) connectivity. The two problems are tightly coupled and we treat

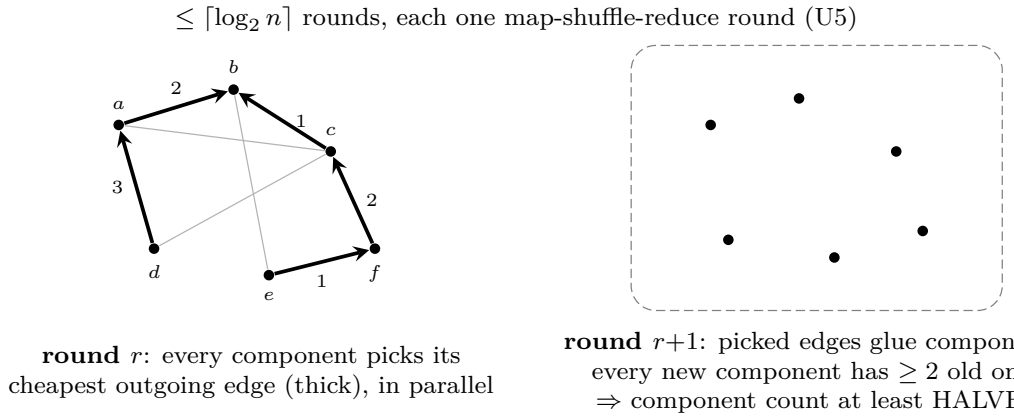


Figure 17.1: One Borůvka round: every component simultaneously grabs its cheapest outgoing edge (safe for the MST by the cut property), and the grabbed edges glue components together — each surviving component swallows at least two old ones, so the count at least halves and $\lceil \log_2 n \rceil$ rounds suffice. Each round is one shuffle (U5), which is why Borůvka — the oldest MST algorithm — is also the most distributed-friendly one.

them together.

The progression through the three machines: - **RAM:** the cut property pins down which edges *must* appear in any MST; Kruskal exploits it greedily. - **PRAM:** Borůvka runs every component's greedy choice simultaneously, then contracts the graph. - **DPRAM:** each Borůvka phase is one MapReduce round; CC by label propagation gives the same structure with a diameter-dependent round count.

Figure 17.1 is one round of the unit's oldest and most distributed-friendly algorithm.

2. RAM — Kruskal's algorithm and the cut property

2.1 The cut property (Blue Rule) — proof

Setup. Let $G = (V, E, w)$ be a connected undirected weighted graph with n vertices and m edges. Assume for clarity that all edge weights are *distinct* (the MST is then unique; the argument extends to ties by a tie-breaking convention).

A **cut** of G is a partition $(S, V \setminus S)$ of the vertex set with $S \neq \emptyset$ and $S \neq V$. An edge **crosses** the cut if one endpoint is in S and the other in $V \setminus S$.

Theorem (Cut Property / Blue Rule). *Let $S \subset V$ be any non-empty proper subset. Let e^* be the minimum-weight edge crossing the cut $(S, V \setminus S)$. Then e^* belongs to the MST of G.*

Proof (exchange argument). Let T be any spanning tree of G. We show that if $e^* \notin T$ then T is not a minimum spanning tree, so the MST must contain e^*.

Suppose $e^* = \{u, v\} \notin T$, with $u \in S$ and $v \in V \setminus S$. Since T is a spanning tree, there is a unique path P in T from u to v. That path must cross the cut $(S, V \setminus S)$ at least once (it starts in S and ends in $V \setminus S$); let $f = \{x, y\}$ be any edge of P that crosses the cut.

Since e^* is the *minimum-weight* edge crossing the cut and f also crosses the cut, we have $w(e^*) < w(f)$ (strict, as all weights are distinct).

Form $T' = (T \setminus \{f\}) \cup \{e^*\}$. We verify: 1. T' **is a spanning tree.** Removing f from T splits T into two subtrees; adding e^* reconnects them (since e^* crosses the same cut, its endpoints lie

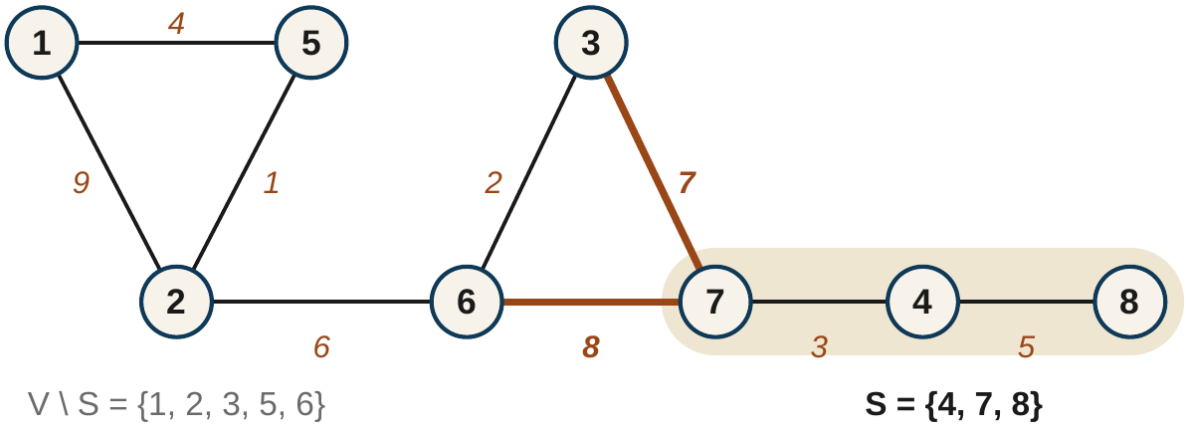


Figure 17.2: Minimum spanning trees and the cut property.

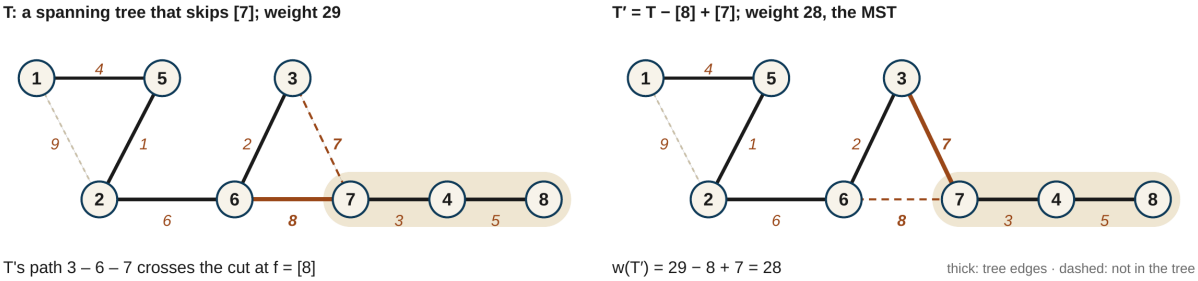


Figure 17.3: One exchange on eight vertices: a spanning tree of weight 29 becomes the minimum spanning tree, of weight 28.

in the two subtrees). So T' is connected, spans all vertices, and has $n-1$ edges — a spanning tree. 2. T' **has strictly smaller total weight.** $w(T') = w(T) - w(f) + w(e^*) < w(T)$ because $w(e^*) < w(f)$.

So T has strictly larger total weight than T', hence T is not an MST. Since T was an arbitrary spanning tree not containing e^*, every spanning tree not containing e^* is non-minimum: the MST contains e^*. □

Remark (uniqueness). With distinct weights the MST is unique: any two minimum spanning trees would differ in some edge, and the lighter of the two alternatives would improve the heavier tree by the same exchange argument.

Figure 17.2 draws the cut property, and Figure 17.3 makes one exchange by hand, which turns a spanning tree of weight 29 into the minimum spanning tree, of weight 28.

2.2 Kruskal's algorithm

Algorithm (Kruskal 1956). Sort all edges in non-decreasing weight order. Maintain a union-find structure on the vertices, initially n singleton sets. Process edges in sorted order: for each edge $\{u, v\}$, if u and v are in different components, add the edge to the MST and union the two components; otherwise discard the edge.

Correctness. At each addition step, the discarded components on either side of the chosen edge form a cut; by the cut property, the lightest edge crossing that cut belongs to the MST. Processing in weight order means we always pick the lightest available crossing edge. Inductive argument: after k additions, the k chosen edges are all in the MST (by the cut property applied

k times); after $n-1$ additions we have a spanning tree consisting entirely of MST edges — the MST itself.

Cost. Sorting m edges costs $O(m \log m) = O(m \log n)$ (since $m \leq n^2$). The union-find operations (at most m `find` + $n-1$ `union`) cost $O(m \cdot \alpha(n))$ with path-compression and union-by-rank, where α is the inverse Ackermann function — effectively a constant for all practical n.[1] Sorting dominates.

Total: $O(m \log n)$ **work.**

Prim's algorithm (one paragraph). Prim (1957) grows the MST from a root vertex: maintain a priority queue of edges with one endpoint in the current tree and one outside; repeatedly extract the minimum, add it, and enqueue its new neighbours. With a binary heap the cost is $O((m+n)\log n) = O(m \log n)$ for connected graphs; with a Fibonacci heap, $O(m + n \log n)$. Prim is sequential in structure (it grows a single frontier) and does not parallelise as naturally as Borůvka; we focus on Borůvka below.

3. PRAM — Borůvka's algorithm (the parallel centrepiece)

3.1 The algorithm

Algorithm (Borůvka 1926; restated for PRAM). Let the input be $G = (V, E, w)$ with distinct edge weights. Maintain a partition of V into components (initially singletons). Repeat: 1. **(Parallel) Minimum-outgoing-edge selection.** For each component C, let $e_C = \arg\min_{e \text{ crosses } \partial C} w(e)$ be the minimum-weight edge that has one endpoint in C and one outside. Select all such edges simultaneously. 2. **(Parallel) Contraction.** Merge every pair of components joined by a selected edge. Update the edge list: remove edges that now have both endpoints in the same component; retain the lighter of any two parallel edges (same component-pair, different weight).

Correctness. Each selected edge e_C is the minimum edge crossing the cut defined by component C, so by the cut property it belongs to the MST. Every phase adds at least one MST edge per component (each component selects its cheapest outgoing edge). The union of all selected edges forms a subgraph of the MST; contracting and repeating yields the full MST.

3.2 Phase count: components at least halve — proof

Lemma. If each of the k components has an outgoing edge (true when G is connected and $k \geq 2$), then after the phase the number of components is at most $\lfloor k/2 \rfloor$, where k is the number of components at the start of the phase.

Proof. Consider the directed graph H on the k components, where each component C points to the component containing the other endpoint of its chosen edge e_C. Every node of H has out-degree exactly 1, and there are no self-loops (e_C leaves C). Borůvka contracts ALL chosen edges simultaneously, so each weakly connected component of H collapses into exactly ONE new component.[2] Since every node of H has an outgoing edge to a DIFFERENT node, no weakly connected component of H can consist of a single node — each has at least 2 nodes. Hence the number of new components = the number of weakly connected components of $H \leq k/2$, i.e. at most $\lfloor k/2 \rfloor$. □

[1] The $O(m \cdot \alpha(n))$ union-find bound is due to Tarjan (1975). The inverse Ackermann function $\alpha(n) < 5$ for all $n < 10^{80}$; treating it as a constant is standard.

[2] Structure note: each weakly connected component of the out-degree-1 digraph H contains exactly one directed cycle (of length ≥ 2) with in-trees hanging off it; the simultaneous contraction collapses the whole weak component to one node. (With distinct edge weights the cycle is in fact always a 2-cycle — a mutually-lightest pair.)

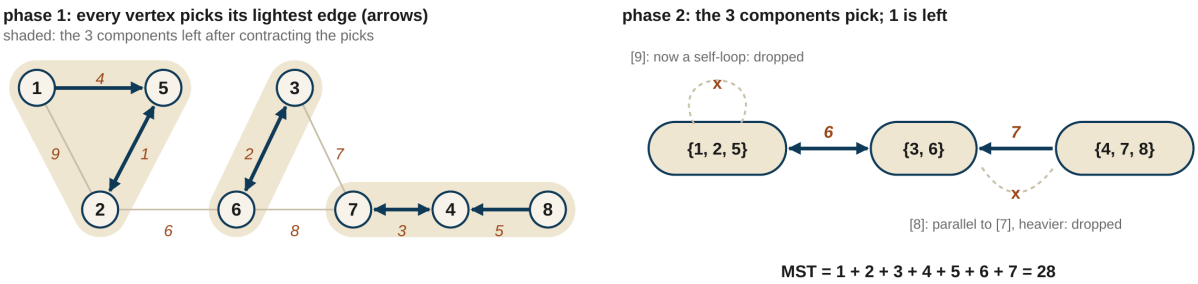


Figure 17.4: Borůvka's algorithm on the same eight vertices, in two phases.

Corollary. After t phases (while more than one component remains; G connected) the number of components is at most $\lfloor n/2^t \rfloor$. The algorithm terminates in at most $\lceil \log_2 n \rceil$ phases.

Figure 17.4 runs Borůvka on the same eight vertices; here two phases suffice.

3.3 Work and depth accounting

Per-phase cost. One Borůvka phase does: - **Minimum-edge selection:** scan the edge list; for each edge, update the minimum for both component labels. With component labels stored in an array, this is $O(m')$ work (m' = current edge count) and $O(\log m')$ depth (parallel reduction per component). - **Contraction:** relabel each vertex with its new component id, remove self-loops, and deduplicate parallel edges. $O(m')$ work and $O(\log m')$ depth (using parallel prefix or sorting).

The edge count m' can only decrease per phase (edges within contracted components are removed), so across all phases the total work is at most $\sum_{t=0}^{\lceil \log n \rceil} O(m) = O(m \log n)$.

Total PRAM costs.

$$W = O(m \log n), \qquad D = O(\log^2 n).$$

The $O(\log^2 n)$ depth arises because each of the $O(\log n)$ phases has internal depth $O(\log n)$ (from the reduction/sorting steps within a phase). This is the same depth structure as parallel mergesort (U2).

Brent instantiation. On p processors:

$$T_p \le \frac{W}{p} + D = O\left(\frac{m \log n}{p}\right) + O(\log^2 n).$$

Speed-up is near-linear for $p \ll m/\log n$.

Randomised star-contraction (stated). An alternative contraction scheme from the parallel graph-contraction literature: each vertex independently flips a fair coin; an edge is *kept* if it runs from a head ("centre") to a tail ("satellite"). Expected satellites: a vertex becomes a satellite when it flips tails and at least one of its neighbours flips heads; for a vertex with at least one neighbour this has probability $\ge \frac{1}{2} \cdot \frac{1}{2} = \frac{1}{4}$, so in a graph without isolated vertices the expected number of satellite vertices is $\ge n/4$. Star-contraction reduces the graph size by at least a $1/4$ factor in expectation per phase, giving $O(\log n)$ phases in expectation. A different combination of ideas — Borůvka phases, random sampling of edges, and a linear-time verification step that discards edges which cannot be in the MST — gives Karger–Klein–Tarjan's (1995) expected linear-time $O(m)$ MST on a RAM, the best known randomised sequential bound. We state this and do not prove it.

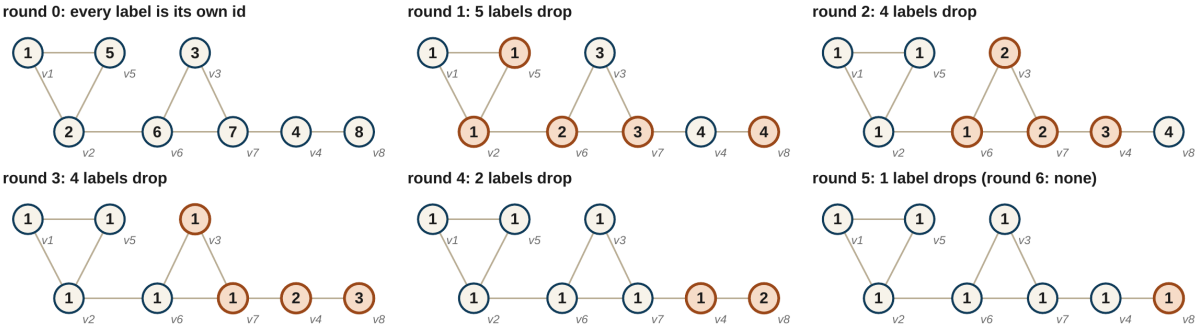


Figure 17.5: Connected components by label propagation, round by round.

4. Connected components — label propagation and the diameter dichotomy

4.1 The problem

Input. An undirected graph $G = (V, E)$, $|V| = n$, $|E| = m$.

Output. A labelling $\ell : V \to V$ such that $\ell(u) = \ell(v)$ if and only if u and v are in the same connected component of G. (Convention: $\ell(v) = \min_{u \in C(v)} u$, the minimum vertex id in the component of v.)

4.2 Label propagation (min-label flooding)

Algorithm. Initialise $\ell(v) = v$ for all v. Repeat: for every vertex v, set $\ell(v) \leftarrow \min(\ell(v), \min_{u:\{u,v\} \in E} \ell(u))$. Terminate when no label changes.

Correctness. Each round propagates the minimum label exactly one hop. After t rounds, the minimum label within any connected component has reached every vertex at graph distance $\leq t$ from the vertex $v_{\min}$ carrying it. Termination requires that the minimum label reaches every vertex — this happens after exactly $\mathrm{ecc}(v_{\min})$ rounds, the *eccentricity* of $v_{\min}$ (its largest distance to any vertex of the component), plus one round to detect that no label changed. Let $\mathrm{diam}(G)$ denote the diameter of the component (the maximum shortest-path distance between any two vertices). Then $\lceil \mathrm{diam}(G)/2 \rceil \leq \mathrm{ecc}(v_{\min}) \leq \mathrm{diam}(G)$, and both bounds are attained (E11.6). So label propagation needs at most $\mathrm{diam}(G)$ rounds, plus the detecting round.

Round count: $O(\mathrm{diam}(G))$**.**

Figure 17.5 shows why: each round carries a label one hop further, so the smallest label needs $\mathrm{ecc}(v_{\min})$ rounds to reach every vertex (here 5, the diameter), plus one round that sees no change.

4.3 The small-diameter dichotomy

The diameter-dependent round count creates a fundamental dichotomy:

- **Small-diameter graphs** (expanders, random graphs, social networks): diameter $O(\log n)$, so label propagation terminates in $O(\log n)$ rounds — efficient in both PRAM and MPC.
- **Large-diameter graphs** (paths, grids, trees): diameter $\Theta(\sqrt{n})$ or $\Theta(n)$, making naive label propagation impractical at scale.

This dichotomy motivates faster CC algorithms.

Figure 17.6 sets the two prices side by side.

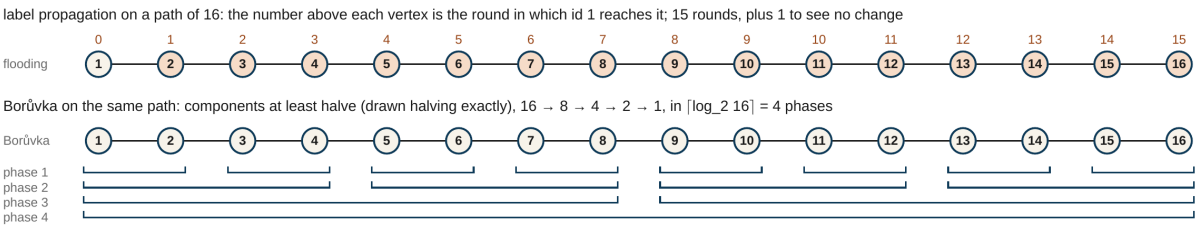


Figure 17.6: Flooding pays for the diameter; contraction pays $\log n$.

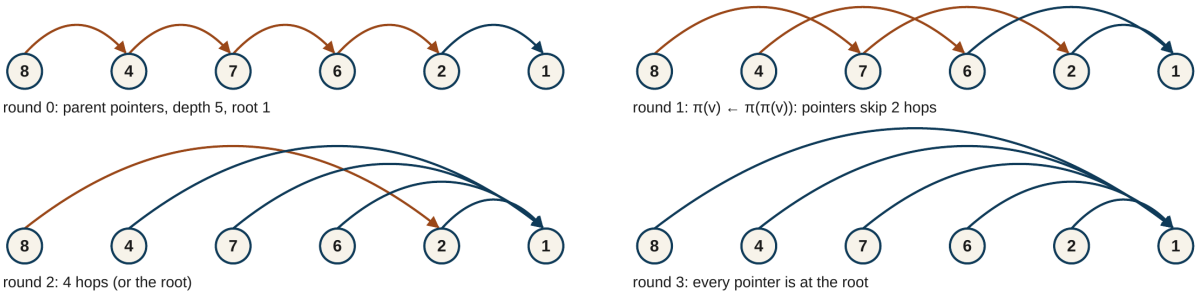


Figure 17.7: Pointer doubling: depth 5 collapses in 3 rounds.

4.4 Pointer-doubling and hash-to-min

Pointer doubling (stated). Maintain a parent pointer $\pi(v)$ for each vertex; the root of each tree is its own parent. Each round: set $\pi(v) \leftarrow \pi(\pi(v))$ (double the pointer). After t rounds, $\pi(v)$ points 2^t hops up the tree. Termination in $O(\log n)$ rounds regardless of diameter; work per round is $O(n)$ pointer updates.

Hash-to-min (stated). A deterministic variant: in each round, each vertex updates its label to the minimum label within distance 2^t (using a BFS-like expansion that doubles in each round via pointer chasing). $O(\log n)$ rounds are proven on paths and conjectured for general graphs (Rastogi et al.; their Hash-Greater-to-Min variant has a proven $3 \log n$ bound), with $O(m \log n)$ total work. The hash-to-min scheme is the basis of the MPC CC algorithms (§5 below).

Figure 17.7 doubles pointers by hand: depth 5 needs $\lceil \log_2 5 \rceil = 3$ rounds.

4b. Strongly connected components: when the best sequential algorithm refuses to parallelise

The directed story completes the undirected one — and teaches the course's central lesson in its sharpest form.

4b.1 The object

Direct the edges and connectivity splits in two. Vertices u, v are **strongly connected** if each reaches the other along directed paths; the maximal such sets are the **strongly connected components (SCCs)**. Collapsing each SCC to a single super-node yields the **condensation** G^{SCC} — and the first theorem is three lines:

Theorem (condensation is a DAG). *If two distinct SCCs lay on a directed cycle of G^{SCC}, every vertex of each would reach every vertex of the other, so they would be one SCC — contradiction.* □

Small as it is, this theorem is load-bearing infrastructure: it says *any* directed graph — including

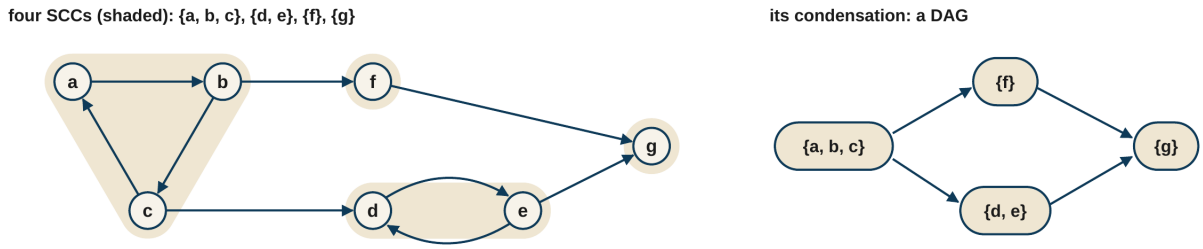


Figure 17.8: Strongly connected components of a directed graph, where depth-first search does not parallelise.

one with feedback loops — becomes a DAG after SCC-condensation, so everything this course does with DAGs (topological order, critical paths, antichain layers, DP-on-DAGs) applies to arbitrary directed systems *after* one condensation pass. U17 uses exactly this: the obligations graph of a governed agent fleet contains genuine cycles (a policy that declares a rule and the hook that enforces it point at each other), and the TBMLS scheduler's first stage — **(T)** — is SCC-condensation, after which Brent, Mirsky, and allocation run on a DAG. *(Lean certificate candidate: finite-graph condensation acyclicity — small, discrete, tranche-1-adjacent.)*

Figure 17.8 draws the strongly connected components of a small directed graph.

4b.2 RAM — Tarjan's algorithm: one DFS, linear time

Tarjan (1972) finds all SCCs in a **single depth-first search**, $O(|V| + |E|)$ — asymptotically optimal, and one of the most beautiful algorithms in the canon. At teaching pace:

- DFS assigns each vertex a discovery index $\mathrm{idx}(v)$ and maintains a stack of vertices whose SCC is not yet decided.
- Each vertex also carries a **lowlink**, defined *recursively* as the smallest discovery index reachable from v by descending through v's DFS subtree and then taking one back/cross edge into a still-on-stack vertex: $\mathrm{lowlink}(v) = \min\big(\mathrm{idx}(v),\ \min_{c \text{ child of } v} \mathrm{lowlink}(c),\ \min_{\{v \to w\} \text{ back/cross}, w \text{ on stack}} \mathrm{idx}(w)\big)$. The recursion aggregates every child's lowlink, so a single vertex's lowlink summarises reachability over its entire subtree (not just one edge).
- **The one-line insight:** v is the *root* of an SCC exactly when $\mathrm{lowlink}(v) = \mathrm{idx}(v)$ — at that moment, popping the stack down to v yields precisely v's SCC, closed and complete.

The lowlink invariant is a strengthened loop invariant of the kind U2 taught you to state and prove; the exercise set asks you to state it exactly and prove the root characterisation. `MATHS->CS:` Tarjan is the standard library's SCC (it is what a compiler runs to find mutually recursive definitions — and what a build system runs to find dependency cycles).

4b.3 The negative lesson — depth: why DFS does not parallelise

Here is the sharpest instance of the course's central lesson. Kruskal (U11 §2) parallelised by re-thinking (Borůvka); sorting (U2) parallelised by re-routing (sample sort); scan (U1) parallelised by re-bracketing. Depth-first search does **not** parallelise by re-thinking: computing the lexicographic DFS ordering is **P-complete** (Reif 1985) — as hard, under fast-parallel reductions, as *anything* in P. Under the standard conjecture (NC $\neq$ P; see F.C1b for the classes), *no* polylog-depth algorithm computes it: the DFS tree is an **inherently deep** object.

The work/depth lens (F.C1) says exactly what this means: Tarjan's algorithm has optimal work and **depth** $\Theta(|V|)$ in the worst case — a chain of discoveries no fleet of processors can shorten.

The sequential-optimal algorithm is not "hard to parallelise for engineering reasons"; it sits below the depth wall *provably*. So the parallel world must **change the primitive**.

4b.4 PRAM/DPRAM — distributed SCC in detail: reachability replaces DFS

What parallelises beautifully is **reachability**: "which vertices does S reach?" is a BFS-flavoured frontier expansion — work $O(|E|)$-ish per phase, depth $O(\text{diam})$ — the exact machinery of §4's label propagation. Both practical distributed-SCC families are built from it.

(a) Forward–backward divide-and-conquer (FW–BW; Fleischer–Hendrickson–Pınar 2000). Pick a pivot v. Compute its forward-reachable set F and backward-reachable set B — two parallel reachability sweeps. Then:

$$\text{SCC}(v) \;=\; F \cap B,$$

and — the divide step — every other SCC lies **entirely** inside one of the three remainder regions $F \setminus B$, $B \setminus F$, or $V \setminus (F \cup B)$ (an SCC is strongly connected, so it cannot straddle a reachability boundary; exercise). Recurse on the three regions **in parallel**. With random pivots the recursion is quicksort-shaped: given parallel-reachability sub-routines, the randomised analyses of Blelloch–Gu–Shun–Sun give expected work within logarithmic factors of linear and polylogarithmic expected span. `MATHS->CS:` the pivot argument is U2's sample-sort argument wearing graph clothes — random splitters, balanced regions, with high probability.

(b) Label propagation in the dataflow model (the U5/§4 machinery). Every vertex carries a colour; propagate colours **forward** to fixpoint (rounds = forward diameter, exactly §4's CC propagation on the directed graph); the vertices that also reach their colour-source **backward** form an SCC; peel and repeat. Each phase is a pair of Pregel-style message-passing fixpoints — implementable in the same dataflow engine as §5's distributed Borůvka, no new systems machinery at all. The **diameter dichotomy of §4 applies verbatim**: low-diameter graphs converge in few rounds; a directed path is the adversary. Practical engines combine (a)+(b): trim trivial SCCs (in/out-degree 0) first, FW–BW on the giant component, label-propagation for the remainder — the composition is an exercise.

The three-lens box, instantiated.

> **RAM:** Tarjan — one DFS, $O(|V|+|E|)$, optimal, *deep*. **PRAM:** reachability-based FW–BW — near-linear expected work, polylog expected span *given* polylog-depth reachability sub-routines (plain BFS reachability has depth $O(\text{diam})$); the primitive changed, not the constant factors. **DPRAM/MPC:** label-propagation SCC — rounds priced by directed diameter; communication = frontier messages, exactly §5's accounting.

Figure 17.9 runs forward–backward on the graph of Figure 17.8.

4b.5 Where you will use it

U17's governed scheduler consumes this section whole: **(T)**arjan-condense the obligations graph (its meta-circular 2-cycles become super-nodes), then Brent-bound and Mirsky-layer the resulting DAG. The condensation-is-a-DAG theorem of §4b.1 is the reason the rest of the pipeline is *allowed to assume acyclicity* — a one-theorem interface between the messy directed world and the clean DAG world. And the depth lesson of §4b.3 is why the scheduler runs condensation as a *phase* (cheap, reachability-parallel at fleet scale) rather than DFS-online.

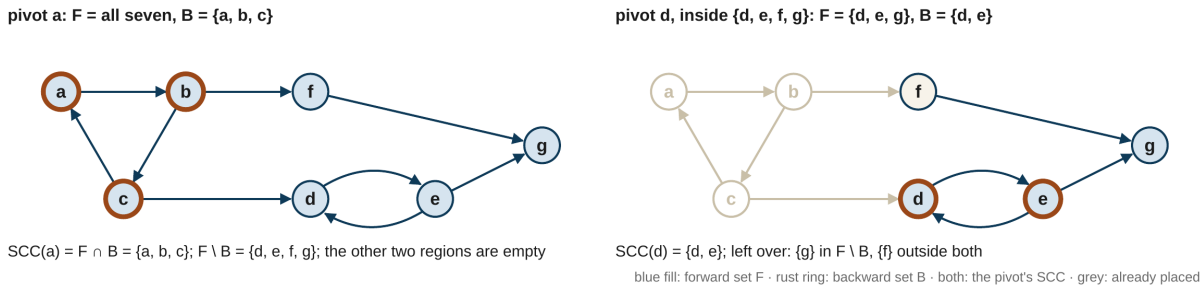


Figure 17.9: Forward–backward on the same directed graph.

4b.6 Lab sketch — project track

Project-track lab (no weekly receipt): run these experiments free and locally in the course lab image (`labs/image/`) — or any Spark-in-Docker environment — as deep-dive and group-project material.

1. **Tarjan by hand** on a 12-vertex graph with three SCCs (trace idx/lowlink/stack); then run the reference implementation and diff your trace.
2. **FW–BW:** implement forward/backward reachability as the two fixpoints of §4's label propagation; measure rounds vs directed diameter on (i) a random graph, (ii) a long path — see the dichotomy with your own eyes.
3. **Condense and schedule:** condense the lab's obligations-graph fixture and topologically layer it (Mirsky) — you have just executed the (T) and (M) stages of U17's TBMLS.
4. *Two-node variant (optional):* the same FW–BW run partitioned across **two machines** — e.g. a pair of desktop-class unified-memory nodes over a high-bandwidth local link, or two laptops on one switch — makes §5's communication term visible in wall-clock: measure rounds × frontier-bytes against the single-node run. *(The course's portable two-node teaching cluster serves this lab when available; the lab is written to require nothing beyond two docker hosts on one network.)*

4b.7 Exercises

1. State Tarjan's lowlink invariant precisely and prove the root characterisation (lowlink(v) = idx(v) iff v roots an SCC).
2. Prove the FW–BW region lemma: every SCC other than SCC(v) is contained in exactly one of $F \setminus B$, $B \setminus F$, $V \setminus (F \cup B)$.
3. Prove condensation acyclicity (§4b.1) for finite directed graphs; then sketch the Lean statement (finite graphs; decidable reachability) — this is a tranche-adjacent cert.
4. The trim optimisation: show a vertex with in-degree or out-degree 0 is a singleton SCC, and bound the work saved on a graph whose giant SCC covers half the vertices.
5. (Deeper) Read the P-completeness reduction for lexicographic DFS and write a one-page account of *what exactly* is inherently sequential about it — connect to F.C1's depth wall and F.C1b's definitions.

4b.8 Sources (credited; archived under `scadamale-book-sources`)

Tarjan (1972) — depth-first search and linear graph algorithms; Reif (1985) — depth-first search is inherently sequential (lex-DFS P-completeness); Fleischer–Hendrickson–Pınar (2000) — FW–BW divide-and-conquer SCC; Blelloch–Gu–Shun–Sun — randomised parallel SCC analysis (near-linear expected work, polylogarithmic expected span). Our own exposition throughout; the U17 forward-connection is the course's own research substrate.

5. DPRAM / MPC — distributed Borůvka and CC in MapReduce

5.1 Distributed Borůvka (per-phase MapReduce shape)

Setup. Edges are stored as (key, value) pairs partitioned across machines. Each component is identified by a component id (initially, the vertex id of the minimum vertex in the component).

One Borůvka phase in MapReduce / Spark:

1. **Min-edge per component (`reduceByKey`).** Map each edge $\{u, v, w\}$ to two key-value pairs: `(comp(u), (w, v))` and `(comp(v), (w, u))`. Reduce by key with `min` on weight: for each component label, find the minimum-weight outgoing edge. Cost: one shuffle, $O(m)$ bytes total.
2. **Contraction (relabel join).** The selected edges define a new component structure: merge the two components at each selected edge's endpoints. Compute new component ids (e.g. take the minimum of the two merging component ids); broadcast or join the new id map. Relabel every edge with updated component ids; drop self-loops (both endpoints in the same new component). Cost: one join (one more shuffle), $O(m)$ bytes.

Rounds per Borůvka phase: $O(1)$ **(two shuffles — two Spark stages).**

Total rounds: $O(\log n)$ (by the component-halving lemma, §3.2).

Total communication: $O(m \log n)$ **bytes** across all phases (each phase moves $O(m)$ bytes and there are $O(\log n)$ phases; the edge count can only decrease, so this is an overcount that can be tightened to $O(m \log n)$ worst-case).

5.2 Connected components in MapReduce — the G14 approach

The Kiveris et al. (2014) CC algorithm (we call it the "G14 sweep fold") runs label propagation in the MPC setting with a number of rounds R polylogarithmic in n, regardless of diameter (see the paper for the proven bound):

Key idea. Rather than propagating one hop per round, use pointer doubling: in round t, each vertex propagates its label 2^{t-1} hops ahead. Concretely:

- Represent the graph as a set of (vertex, label) pairs and (edge) pairs.
- In each round: for each edge $\{u, v\}$, emit `(u, label(v))` and `(v, label(u))`; reduce at each vertex to take the minimum label; update.
- Use the label update to also update the edge representation (relabel edges), so in subsequent rounds the label of a vertex propagates through the contracted structure.

The bytes accounting per round: each edge generates 2 key-value pairs ($O(m)$ bytes); with the label updates, the total over the R rounds is $O(mR)$ bytes.

Spark shape: alternating `flatMap` / `reduceByKey` / join pairs; $O(R)$ shuffle stages. In GraphX/GraphFrames: the `connectedComponents` API implements a variant of this algorithm. The exact primitive (hash-to-min, alternating-fields, label-propagation-on-RDD) varies by version and is an active engineering area; for the course we treat the G14 shape as the reference algorithm.

Note on GraphX / GraphFrames. These Spark graph APIs provide `connectedComponents`, `shortestPaths`, and related primitives. They implement variants of the above. The lab sketch below is written system-agnostically, so it remains valid whether we use GraphBLAS, GraphX, Pregel-on-Spark, or a hand-rolled RDD pipeline.

5.3 The sublinear-memory connectivity frontier

Distributed Borůvka and the G14 approach spread the m edges across machines (§5.1), so no machine holds all of them; the question is how small each machine's local memory s may be. Our sketches assume a machine can hold n-sized bookkeeping when needed (the broadcast of the component-id map in §5.1). When $s \ll n$ (the ultra-large regime), no machine can hold even the vertex set.

The sublinear-memory connectivity problem asks: can we solve CC on a graph with n vertices and m edges in $o(\log n)$ MPC rounds — asymptotically fewer than $\log n$ — with $s = O(n^{1-\epsilon})$ local memory per machine (i.e., sublinear in n), for some constant $\epsilon > 0$? Note that $O(\log n)$ rounds are not the question: they are achievable in this regime, for instance by simulating a logarithmic-depth PRAM connectivity algorithm with $O(1)$ MPC rounds per PRAM step. The question is whether the logarithm can be beaten.

To the best of our knowledge this is open for general graphs. For graphs of small diameter D the logarithm has been beaten: Andoni et al. (FOCS 2018) and Behnezhad et al. (FOCS 2019) reach round counts of order $\log D$ times, or plus, $\log \log n$. This connects to U5's one-cycle-vs-two-cycles conjecture (the MPC connectivity conjecture): distinguishing a graph that is a single cycle from one that is two disjoint cycles is conjectured to require $\Omega(\log n)$ rounds in the sublinear-memory MPC model. If the conjecture holds, the answer to the question above is no, and $O(\log n)$ rounds are optimal. The connectivity problem at scale is thus both practically important and theoretically rich.

6. The invariant (U11 instantiation)

RAM: MST costs $O(m \log n)$ — Kruskal's algorithm greedily adds minimum-crossing edges justified by the cut property (proved by exchange argument). Union-find with path compression and union-by-rank costs $O(m \cdot \alpha(n))$ for the data-structure component; sorting dominates.

PRAM: Borůvka (1926) is the natural parallel MST — each component simultaneously selects its minimum outgoing edge; components at least halve per phase (proved); $O(\log n)$ phases, $W = O(m \log n)$, $D = O(\log^2 n)$. Randomised star-contraction gives $O(\log n)$ expected contraction phases; Borůvka phases with random edge sampling and linear-time verification (KKT 1995) give expected linear-time sequential MST (both stated). Connected components via pointer doubling: $O(\log n)$ rounds, $O(m \log n)$ work (hash-to-min: proven on paths, conjectured in general).

DPRAM/MPC: distributed Borůvka = $O(\log n)$ MapReduce rounds, each round two shuffles (`reduceByKey` for min-edge, join for relabelling). CC via G14: R rounds, polylogarithmic in n, and $O(mR)$ bytes. Whether sublinear-memory connectivity can beat $O(\log n)$ rounds is open (U5's one-cycle/two-cycles conjecture says no). The lab is system-agnostic (GraphBLAS / GraphX / hand-rolled RDD).

7. Lab sketch — project track

Project-track lab (no weekly receipt): run these experiments free and locally in the course lab image (`labs/image/`) — or any Spark-in-Docker environment — as deep-dive and group-project material. The two experiments below are valid across GraphBLAS, GraphX, and hand-rolled-RDD backends.

Experiment 1 — CC label propagation rounds vs graph diameter.

Generate two families of graphs (using the NetworkX Python library, pre-installed in the lab image): - **Random Erdős–Rényi graphs** $G(n,p)$ with $p = c\ln n/n$ (barely connected; diameter $O(\log n)$); vary $n \in \{10^3, 10^4, 10^5\}$. - **Path graphs** P_n (diameter $= n-1$); vary $n \in \{10^3, 10^4, 10^5\}$.

For each graph, run label propagation (min-label flooding, §4.2) and count the number of rounds to termination. Plot rounds vs n for both families.

Questions: (a) What functional relationship do you observe between rounds and n for Erdős–Rényi? For paths? (b) Does the empirical round count for Erdős–Rényi match the $O(\log n)$ prediction? (c) Propose one real-world graph class (social network, road network, web graph) that belongs to each regime; justify.

Experiment 2 — MST phase count vs $\log n$ (Borůvka empirical).

Generate random weighted graphs $G(n,m)$ with $n \in \{10^3, 10^4, 10^5\}$ and $m = 5n$ (sparse, and not always connected: about ne^{-10} vertices are isolated, roughly 4.5 at $n = 10^5$, so run on the largest connected component or count phases to the minimum spanning forest (there the halving applies only to components that still have an outgoing edge; finished components stay)). Implement Borůvka's algorithm (no parallelism required for the measurement — a sequential simulation suffices) and count the number of phases until no component has an outgoing edge.

Plot (number of phases) vs $\log_2 n$ for each n. Fit a line; does the phase count track $\log_2 n$?

Questions: (a) Does the empirical phase count match the $O(\log n)$ upper bound? (b) Is it ever significantly below $\log_2 n$? (c) Compare the phase count for a path graph P_n (with unit weights) — does the worst case behave differently?

Deliverable. A notebook with the two experiments, plots, and a half-page cost-accounting in the three-lens vocabulary.

8. Lean certificate target

Target (new — U11): the cut property (§2.1), the exchange-argument proof.

The cut property is an ideal Lean target: it is a finite combinatorial statement about finite graphs (finite vertex/edge sets, a weight function, a spanning tree), the proof uses only set manipulations and the ordering on weights, and the key step (the exchange argument) is a constructive witness.

```
-- SCHEMATIC (names to align to BMLSPar.lean conventions and the course
  graph-formalisation at cert time):
-- G : FinGraph, w : E → ℝ, MST : SpanningTree G, S : Finset V (a side of a
  cut)
-- e_star : E with endpoints {u, v}, u ∈ S, v ∉ S, and w(e_star) = min of
  crossing edges
theorem cut_property (G : FinGraph) (w : G.Edge → ℝ) (hw : Set.InjOn w
  G.edgeSet)  -- distinct edge weights
    (S : Finset G.V) (hS : S.Nonempty) (hSc : S ≠ Finset.univ)
    (e_star : G.Edge) (he_cross : crosses S e_star)
    (he_min : ∀ f, crosses S f → w e_star ≤ w f) :
    ∀ T, IsMST G w T → e_star ∈ T := by
  -- exchange argument: assume e_star ∉ MST, construct a lighter spanning tree,
    contradiction
  sorry -- cert target: fill this in (lab U11.L1)
```

Exercise U11.L1 (light, Lean). Load the Lean environment. State the theorem above; the names are schematic (`FinGraph`, `crosses`, `MST` are not Mathlib names); see `IasCert.Book.Unit11` for a version that type-checks. Then: fill in the exchange argument at the `sorry`: (1) assume `e_star` $\notin$ `MST`; (2) exhibit the unique path in `MST` from `u` to `v`; (3) select a crossing edge `f` on that path; (4) define `MST' = MST - {f} ∪ {e_star}` and verify it is a spanning tree; (5) compute `w(MST') < w(MST)` from `w(e_star) < w(f)`. Conclude by contradiction.

Exercise U11.L2 (deep, Lean). State and certify the component-halving lemma (§3.2): after one Borůvka phase, the number of components is at most $\lfloor k/2 \rfloor$. A suitable formalisation: define a `BoruvkaStep` function on component partitions, prove `BoruvkaStep_components_le_half`.

Note: the schematic is certifiable — the exchange argument is a finite case analysis with a constructive witness, exactly the kind of argument Lean handles cleanly.

> **LLM lens** — where this unit lands on the running workload (U0L):
> Retrieval corpora and knowledge bases arrive as sharded graphs; the connectivity and spanning structures this unit computes are the preprocessing layer under graph-augmented retrieval. (A light lens — the heavy graph connection is U12's.)

9. Sources (credited; archived under `scadamale-book-sources`)

Borůvka (1926). O. Borůvka, *O jistém problému minimálním* (On a certain minimum problem), Práce Moravské Přírodovědecké Společnosti 3, 1926. — The first MST algorithm; the parallel structure is inherent in the original paper. Historical context: Nevšemalová, M., Nešetřil, J., and Nešetřilová, H., *Otakar Borůvka on minimum spanning tree problem*, Discrete Mathematics 233(1–3):3–36, 2001. — Cite for history; our exposition is self-contained.

Kruskal (1956). J. B. Kruskal, *On the shortest spanning subtree of a graph and the traveling salesman problem*, Proc. Amer. Math. Soc. 7(1):48–50, 1956. — The greedy sort-and-add algorithm.

Prim (1957). R. C. Prim, *Shortest connection networks and some generalizations*, Bell System Tech. J. 36(6):1389–1401, 1957. — The priority-queue MST algorithm.

Karger, Klein, Tarjan (1995). D. R. Karger, P. N. Klein, R. E. Tarjan, *A randomized linear-time algorithm to find minimum spanning trees*, J. ACM 42(2):321–328, 1995. — Randomised linear-time MST; KKT 1995. Cited for the stated expected linear-time MST.

Kiveris et al. (2014). R. Kiveris, S. Lattanzi, V. Mirrokni, V. Rastogi, S. Vassilvitskii, *Connected components in MapReduce and beyond*, SoCC 2014. — The G14 CC algorithm for MPC; polylogarithmically many rounds regardless of diameter.

Rastogi et al. (2013). V. Rastogi, A. Machanavajjhala, L. Chitnis, A. Das Sarma, *Finding connected components in map-reduce in logarithmic rounds*, ICDE 2013. — Hash-to-Min and Hash-Greater-to-Min (§4).

Andoni et al. (2018); Behnezhad et al. (2019). A. Andoni, Z. Song, C. Stein, Z. Wang, P. Zhong, *Parallel graph connectivity in log diameter rounds*, FOCS 2018; S. Behnezhad, L. Dhulipala, H. Esfandiari, J. Łącki, V. Mirrokni, *Near-optimal massively parallel graph connectivity*, FOCS 2019. — The small-diameter results of §5.3.

Wikipedia. *Minimum spanning tree*, *Borůvka's algorithm*, *Kruskal's algorithm*, *Connected component (graph theory)*, *Parallel connectivity*. — Background and cross-references.

MIT OCW 6.046. *Design and Analysis of Algorithms*, MIT OpenCourseWare. Lecture notes on graph algorithms, MST, and union-find. — Freely available; credited.

10. Exercises

Light (examined core):

E11.1 Apply the cut property to argue that the minimum-weight edge of the *entire* graph must be in any MST. (Take $S = \{u\}$ where u is one endpoint of the global minimum edge and the other endpoint is in $V \setminus S$; apply the Cut Property of §2.1.)

E11.2 Trace Kruskal's algorithm on the graph with vertices $\{1, 2, 3, 4\}$ and edges $\{1\text{-}2, w = 4\}$, $\{1\text{-}3, w = 2\}$, $\{2\text{-}3, w = 1\}$, $\{2\text{-}4, w = 5\}$, $\{3\text{-}4, w = 3\}$. List the edges added in order, and give the total MST weight.

E11.3 Give the three-lens invariant for MST and connected components using the U0 box template. Fill in: (a) the RAM cost and the key data structure; (b) the PRAM algorithm, phase count, work, depth; (c) the MPC algorithm, round count, total bytes.

E11.4 In one Borůvka phase on a graph with $k = 8$ components, what is the maximum number of components that can survive? The minimum? Explain from the component-halving argument.

Deep (Module-3 / optional):

E11.5 **Prove that components at least halve in each Borůvka phase.** More precisely: let k be the number of components before a phase, and suppose each of them has an outgoing edge (true when G is connected and $k \geq 2$). Show that after the phase, the number of components is at most $\lfloor k/2 \rfloor$. (*Hint:* consider the directed graph H where each component points to the component it chose (every node has out-degree 1, no self-loops). Borůvka contracts ALL chosen edges simultaneously, so each weakly connected component of H collapses to exactly ONE new component. Conclude that there are no singleton weakly connected components of H (every node has an outgoing edge to a different node), so every weakly connected component has size ≥ 2, giving at most $\lfloor k/2 \rfloor$ new components.)

E11.6 **Diameter bounds for label propagation.** Let G be a connected graph with diameter D, and let $v_{\min}$ be the vertex with the minimum id. Show that label propagation (min-label flooding, §4.2) changes some label in each of the first $\mathrm{ecc}(v_{\min})$ rounds and in no later round, where $\mathrm{ecc}(v_{\min})$ is the largest distance from $v_{\min}$ to any vertex; then show $\lceil D/2 \rceil \leq \mathrm{ecc}(v_{\min}) \leq D$, and give labellings of the path P_n that attain each bound. (*Hint:* the minimum label moves exactly one hop per round; for the lower bound, apply the triangle inequality through $v_{\min}$ to two vertices at distance D.)

E11.7 (Optional research question) The MPC connectivity conjecture (§5.3) states that distinguishing a single n-cycle from two $n/2$-cycles requires $\Omega(\log n)$ rounds in the sublinear-memory MPC model. (a) Why is this problem hard — what property of the graph structure makes it difficult to detect within a few rounds? (b) The conjecture concerns $o(\log n)$ rounds, not $O(\log n)$: explain why pointer doubling along the cycles (§4.4), which keeps only a constant amount of state per vertex, separates the two inputs in $O(\log n)$ rounds even with $s = O(n^{1-\epsilon})$, and why beating $\log n$ rounds would need a vertex's reach to grow by more than a constant factor per round.

Chapter 18

12 pagerank triangles

U12 — Graphs II: PageRank, Triangle Counting, and Graph Partitioning

The ScaDaMaLe book, unit 12 (examined core). Our own exposition; original sources credited in §10.

Instantiating the three-lens invariant. The U0 box for this unit:

> **RAM:** power iteration for PageRank — $O(m)$ per step where $m = |E|$; triangle counting via A^3 trace — $O(n^\omega)$ ($\omega < 2.372$, U6 §2.3); both baselines are brute-force. **PRAM:** parallel frontier BFS; the NodeIterator++ degree-split argument gives $O(m^{3/2})$ triangle work — the depth follows the dependency graph of the intersection tests. **DPRAM / Spark:** PageRank with co-partitioned links eliminates the join-side (links) shuffle after the first iteration (the rank contributions still shuffle); NodeIterator++ in two MapReduce passes; Pregel / GraphX vertex-program model; edge-cut vs vertex-cut partitioning for power-law graphs.

Entry kit — what this chapter assumes; follow a link if you need the memory jog (map: Figure 1): F.M2 (eigenvectors and the power iteration) • F.C2 (graph representations) • F.S1 (expectation, for the random-surfer reading)

1. Intuition — three graph problems, one lesson

Three apparently different problems — ranking nodes, counting dense substructures, partitioning a graph into pieces — share one underlying challenge: **the information a node needs is spread across its neighbours**, and in a large distributed graph those neighbours may live on many different machines.

The three-lens invariant reveals their structure:

- **PageRank** is a linear-algebra fixed-point iteration: the rank of a page depends on the ranks of all pages that link to it. Sequentially this is $O(m)$ per step; the challenge is that billions of web-graph edges live on thousands of machines, so each step shuffles the adjacency lists and the rank values across the cluster. Co-partitioning the adjacency with the rank table removes the join-side (links) shuffle; the per-edge rank contributions still move every step.
- **Triangle counting** is a combinatorial inner-product problem: a triangle exists when two neighbours of a node are themselves connected. The naive approach — check all triples — is cubic. A degree-based split reduces it to $O(m^{3/2})$, provably tight for *listing* the triangles

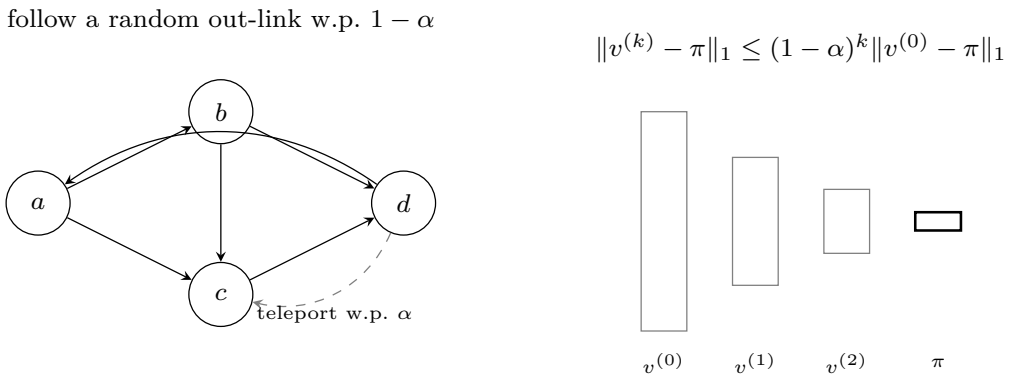


Figure 18.1: PageRank in two panels. Left: the random surfer follows a uniform out-link with probability $1-\alpha$ and teleports with probability α — teleportation makes the chain primitive, so a unique stationary distribution π exists (Perron–Frobenius). Right: power iteration contracts toward π geometrically in total variation at rate $(1-\alpha)$ per step — the shrinking boxes are the error $\|v^{(k)}-\pi\|_1$. The same fixed-point-plus-contraction shape returns whenever importance flows through a weighted graph, including the attention graphs of U0L/U18.

(counting alone can go below it, with fast matrix multiplication). In MapReduce this becomes two carefully designed passes; the byte accounting is non-trivial.

- **Graph partitioning** — divide the vertices into balanced groups minimising edges cut — is NP-hard in general. Two principled relaxations: the Fiedler spectral relaxation (the algebraic connectivity Laplacian eigenvector) and the vertex-cut strategy for power-law graphs (PowerGraph). Both connect to the deeper structure of the graph.

Figure 18.1 is the PageRank half of the unit in two panels.

Check your reading.
Q1. PageRank, triangle counting, and graph partitioning look unrelated. Section 1 names the ONE underlying challenge they share:
(A) all three are NP-hard (B) all three reduce to sorting the edge list (C) the information a node needs is spread across its neighbours, and in a large distributed graph those neighbours live on many machines (D) all three require the full adjacency matrix in one machine's memory

2. RAM — the sequential baselines

2.1 PageRank: the random-surfer Markov chain

Let $G=(V,E)$ be a directed graph with $n=|V|$ nodes and $m=|E|$ edges. Let A denote the adjacency matrix ($A_{ij}=1$ if there is an edge from j to i, i.e. j links to i), and D the diagonal out-degree matrix ($D_{jj}=\text{outdeg}(j)$).

The random-surfer model (Page, Brin, Motwani, Winograd 1999). A surfer at node j: - with probability $1-\alpha$ follows a uniformly random outgoing link — the transition $A\,D^{-1}$ (column j sums to 1 if $\text{outdeg}(j)\ge 1$; a node without out-links is handled below); - with probability α teleports to a node drawn from the uniform distribution $\mathbf{1}/n$ (uniform teleportation).

Definition. The **damped PageRank matrix** is

$$P \;=\; \frac{\alpha}{n}\,\mathbf{1}\mathbf{1}^T \;+\; (1-\alpha)\cdot A\,D^{-1},$$

where $\mathbf{1}$ is the all-ones column vector, so every entry of the teleportation term $\frac{\alpha}{n}\mathbf{1}\mathbf{1}^T$ is α/n

and each of its columns is α times the uniform distribution $\mathbf{1}/n$. Standard damping: $\alpha = 0.15$. (Mind the order: with $A_{ij} = 1$ iff j links to i and D the *out*-degree matrix, it is $A\,D^{-1}$ whose columns sum to one — column j spreads node j's rank over its out-neighbours; $D^{-1}A$ would divide by the wrong node's degree.)

If a node j has no out-links (a **dangling** node), then $D_{jj} = 0$; reading D^{-1} as zero there, column j of $A\,D^{-1}$ is zero and column j of P sums to α, not 1: rank that reaches j leaks out of the chain. The standard patch lets the surfer at a dangling node jump uniformly: replace $A\,D^{-1}$ by

$$S \;=\; A\,D^{-1} \;+\; \frac{1}{n}\,\mathbf{1}\,\mathbf{d}^T,$$

where $d_j = 1$ if $\mathrm{outdeg}(j) = 0$ and $d_j = 0$ otherwise, and from here on let

$$P \;=\; \frac{\alpha}{n}\,\mathbf{1}\mathbf{1}^T \;+\; (1-\alpha)\,S.$$

This P is column-stochastic and every entry is at least $\alpha/n > 0$, so it is **primitive** (irreducible and aperiodic) for every graph with at least one node. Primitivity is the hypothesis of the Perron–Frobenius theorem.

Theorem (Perron–Frobenius; existence and uniqueness). A primitive column-stochastic $n \times n$ matrix P has a unique stationary distribution π satisfying $P\pi = \pi$, $\|\pi\|_1 = 1$, $\pi > 0$ componentwise.

Proof sketch. By Perron–Frobenius, P has a unique eigenvalue of modulus 1 (namely 1), and the corresponding eigenvector is strictly positive and unique up to scaling; normalising to $\|\pi\|_1 = 1$ gives the stationary distribution. □

The PageRank vector π is this stationary distribution. Its entry π_i is the long-run fraction of time the random surfer spends at node i.

2.2 Convergence of power iteration: TV contraction

Power iteration. Starting from any distribution $v^{(0)}$ (e.g., uniform), set

$$v^{(k+1)} = P\,v^{(k)}.$$

Theorem (total-variation contraction; the damping argument). For any distribution $v^{(0)}$, with π the stationary distribution of P,

$$\left\|v^{(k)} - \pi\right\|_1 \;\le\; (1-\alpha)^k \left\|v^{(0)} - \pi\right\|_1.$$

Proof. Write $P = \alpha R + (1-\alpha)Q$ where $R = \frac{1}{n}\mathbf{1}\mathbf{1}^T$ (rank-1 teleportation; $Rv = \mathbf{1}/n$ for any distribution v) and $Q = S$ (the patched link matrix of §2.1). Since both R and Q are column-stochastic:

$$v^{(k+1)} - \pi = P(v^{(k)} - \pi) = \alpha R(v^{(k)} - \pi) + (1-\alpha)Q(v^{(k)} - \pi).$$

The term $\alpha R(v^{(k)} - \pi)$ vanishes: $Ru = \mathbf{1}(\mathbf{1}^T u)/n$ and $\mathbf{1}^T(v^{(k)} - \pi) = 0$ (both are distributions, so they sum to 1). Therefore:

$$v^{(k+1)} - \pi = (1-\alpha)Q(v^{(k)} - \pi).$$

Since Q is column-stochastic, it is non-expansive in total variation: $\|Qu\|_1 \le \|u\|_1$. Iterating:

$$\left\|v^{(k)} - \pi\right\|_1 \le (1-\alpha)^k \left\|v^{(0)} - \pi\right\|_1. \quad \square$$

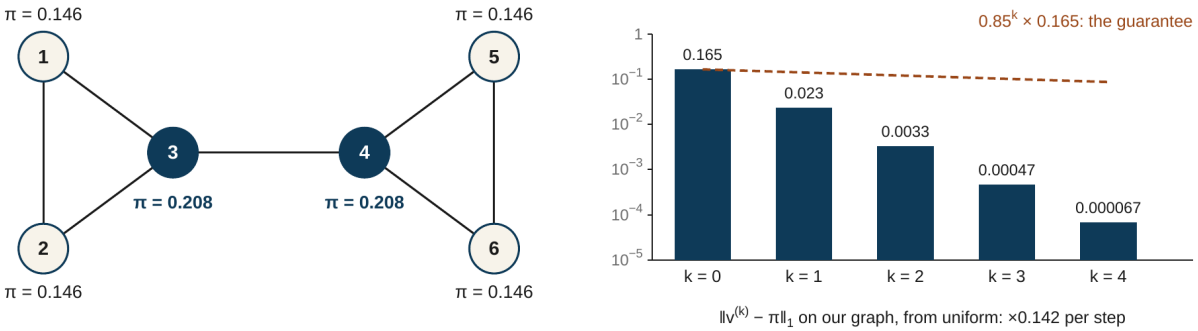


Figure 18.2: PageRank on the two-triangle graph (undirected, no dangling page, $\alpha = 0.15$).

Iteration count. To reach $\|v^{(k)} - \pi\|_1 \le \epsilon$ with $\alpha = 0.15$, starting from $\|v^{(0)} - \pi\|_1 \le 1$:

$$(1 - 0.15)^k \le \epsilon \quad \Longrightarrow \quad k \ \ge \ \frac{\ln \epsilon}{\ln 0.85} \ \approx \ -6.15 \ln \epsilon.$$

For $\epsilon = 10^{-8}$: $k \ge 6.15 \times 8 \ln 10 \approx 113.3$, so $k \ge 114$. Two distributions are at most 2 apart in ℓ_1, so from an arbitrary start ($\|v^{(0)} - \pi\|_1 \le 2$) the condition is $(0.85)^k \cdot 2 \le 10^{-8}$, i.e. $k \ge \ln(2 \times 10^8)/\ln(1/0.85) \approx 117.6$, so $k \ge 118$. The rate $1 - \alpha$ cannot be improved in general: the second eigenvalue of P satisfies $|\lambda_2(P)| \le 1 - \alpha$, with equality whenever the link graph has two or more closed communicating classes, as the web graph does (Haveliwala–Kamvar 2003). The bound does not depend on n: a *small constant* number of iterations (10–50 in practice) suffices for web-scale ranking.

RAM cost. Each iteration computes $v^{(k+1)} = (1-\alpha) A\, D^{-1} v^{(k)} + (1-\alpha)\,(\mathbf{d}^T v^{(k)}/n)\,\mathbf{1} + \alpha/n \cdot \mathbf{1}$: one sparse matrix-vector multiply touching every edge once — $O(m)$ arithmetic operations — plus a scalar normalisation — $O(n)$. Total for K iterations: $O(Km)$.

Figure 18.2 iterates PageRank on the two-triangle graph with $\alpha = 0.15$.

2.3 Triangle counting: A^3 trace vs combinatorial

A **triangle** in an undirected graph $G = (V, E)$ is a set of three mutually adjacent vertices. The number of triangles is

$$\Delta(G) \ = \ \frac{1}{6} \sum_{i=1}^{n} [A^3]_{ii},$$

because $[A^3]_{ii}$ counts the number of walks of length 3 from i back to i, each triangle contributing 6 such walks (two orientations $\times$ three starting vertices).

Sequential costs. - A^3 by matrix multiplication: $O(n^\omega)$, $\omega < 2.372$ (Williams, Xu, Xu, Zhou 2024; U6 §2.3) (Coppersmith-Winograd / Williams). For sparse graphs ($m \ll n^2$), combinatorial methods are much faster. - **Naive combinatorial:** enumerate all $\binom{m}{2}$ pairs of edges sharing a vertex; for each pair check if the third edge exists. Cost $O(m^2)$ — quadratic in the edge count.

The key insight: both are too slow for billion-edge graphs. The NodeIterator++ algorithm achieves $O(m^{3/2})$, which is tight for *listing* all triangles. Counting alone can go below it: Alon–Yuster–Zwick (1997) count triangles in $O(m^{2\omega/(\omega+1)}) \approx O(m^{1.41})$ with fast matrix multiplication.

Figure 18.3 draws the two-triangle graph with its degrees, and Figure 18.4 checks its wedges: 6 closed wedges give 2 triangles.

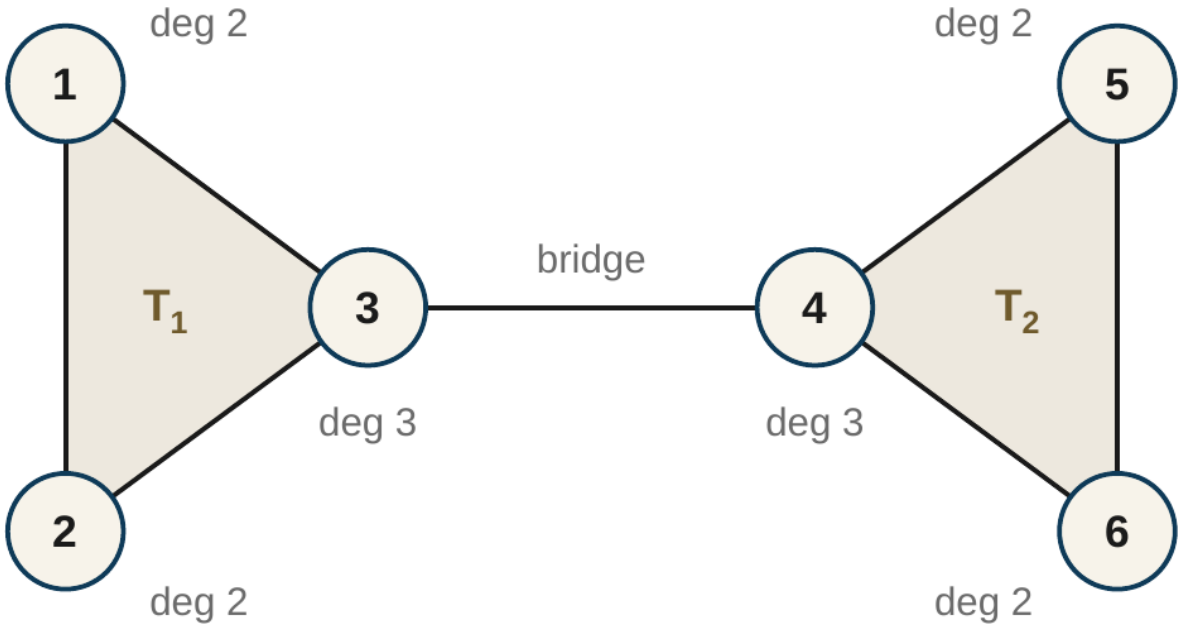


Figure 18.3: The two-triangle graph and its degrees.

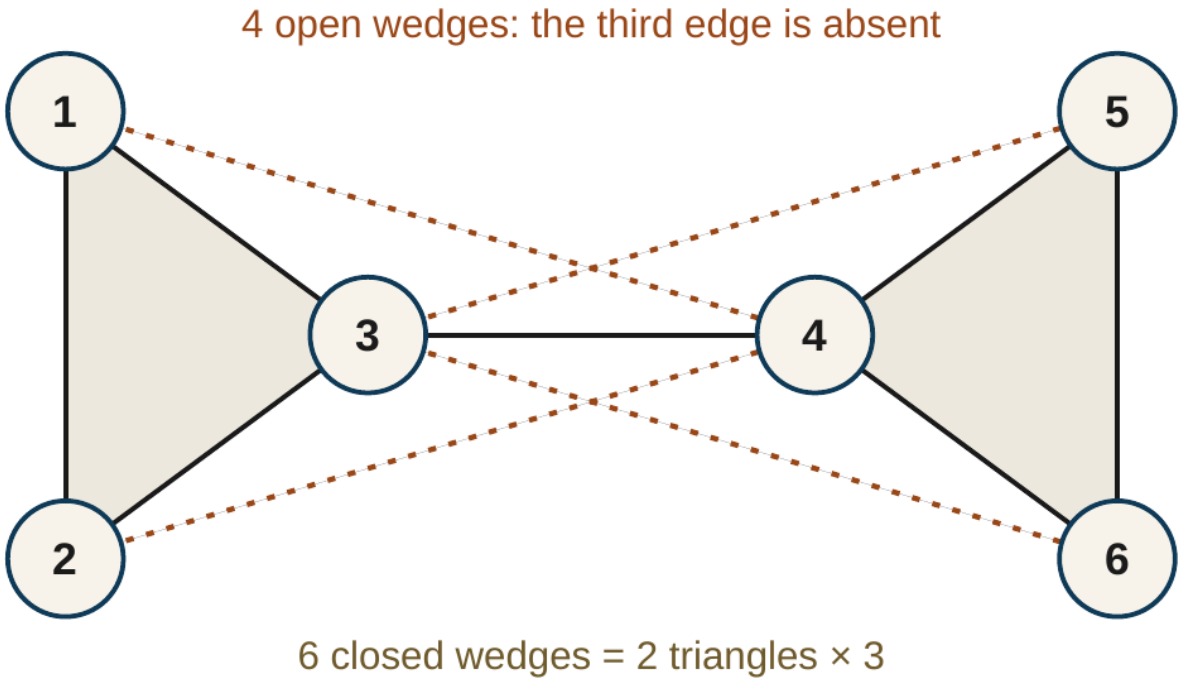


Figure 18.4: The wedges of the two-triangle graph: 4 open and 6 closed; each triangle closes 3 wedges.

Check your reading.

Q2. With this unit's convention (A_ij = 1 iff j links to i; D the diagonal OUT-degree matrix), the link-following transition of the random surfer (before the dangling-node patch) is:
(A) D^{-1} A – divide each row of A by that row's degree (B) A D^{-1} – column j spreads node j's rank uniformly over j's out-neighbours (C) A alone – an adjacency matrix is already stochastic (D) D A – weight each link by the out-degree

Q3. The damping theorem gives $||v^(k) - pi||_1 \leq (1 - alpha)^k \ ||v^(0) - pi||_1$. With alpha = 0.15, from a start with $||v^(0) - pi||_1 \leq 1$, roughly how many iterations does THIS bound require to guarantee total-variation accuracy 10^{-8}?
(A) about 8 (B) about 30 (C) about 114 (D) about 10^8

3. PRAM — parallel PageRank frontier and NodeIterator++

3.1 Parallel PageRank (PRAM)

Each power-iteration step $v^{(k+1)} = Pv^{(k)}$ is a sparse matrix-vector multiply. On the PRAM: - **Work:** $W = O(m)$ (one multiply-add per edge). - **Depth:** $D = O(\log n)$ (parallel reduction over the in-neighbours of each node; the fan-in reduction has depth log(indeg), bounded by $\log n$).

The K-iteration power iteration has total work $W = O(Km)$ and depth $D = O(K \log n)$. By Brent: $T_p \leq O(Km/p + K \log n)$. For web-scale graphs, $K \approx 30$ and $m \approx 10^{10}$ — the work term dominates for realistic p.

3.2 NodeIterator++ and the $O(m^{3/2})$ bound (PRAM)

Degree-priority ordering. Fix a total order $\succ$ on V breaking ties by degree: $u \succ v$ if $\deg(u) > \deg(v)$, or degrees are equal and $u > v$ lexicographically. Define the **reduced neighbourhood** $\Gamma^+(v) = \{u \in \Gamma(v) : u \succ v\}$.

Algorithm (NodeIterator++, Suri–Vassilvitskii 2011, based on the sequential version of Chiba–Nishizeki 1985). For each node v: for each $u \in \Gamma^+(v)$: for each $w \in \Gamma^+(v) \cap \Gamma^+(u)$: count a triangle. A triangle $\{a, b, c\}$ with $a \prec b \prec c$ is counted exactly once, at $v = a$, $u = b$, $w = c$.

The intersection $\Gamma^+(v) \cap \Gamma^+(u)$ is computed by iterating over the smaller of the two neighbour sets and probing the larger with hash lookups (expected $O(1)$ each), taking expected time $O(\min(|\Gamma^+(v)|, |\Gamma^+(u)|))$ per edge. (A merge of the two sorted lists costs $O(|\Gamma^+(v)| + |\Gamma^+(u)|)$ instead; by the lemma below both lists have at most $2\sqrt{m}$ entries, so the bound of the theorem holds for the merge as well.)

Theorem ($O(m^{3/2})$ runtime; Chiba–Nishizeki 1985, MR form in Suri–Vassilvitskii 2011). NodeIterator++ runs in $O(m^{3/2})$ total work.

Proof. Set a threshold $t = \sqrt{m}$. Call a node **heavy** if $\deg(v) \geq t$ and **light** otherwise.

There are at most $2m/t = 2\sqrt{m}$ heavy nodes (since total degree is $2m$).

Lemma. $|\Gamma^+(v)| \leq 2\sqrt{m}$ for every node v. If v is light, $|\Gamma^+(v)| \leq \deg(v) < \sqrt{m}$. If v is heavy, every $u \in \Gamma^+(v)$ has $u \succ v$, so $\deg(u) \geq \deg(v) \geq t$: $\Gamma^+(v)$ contains only heavy nodes, and there are at most $2\sqrt{m}$ of them.

For each edge $\{u, v\}$ with $u \succ v$, the work for the intersection $\Gamma^+(v) \cap \Gamma^+(u)$ is $O(\min(|\Gamma^+(v)|, |\Gamma^+(u)|)) \leq O(|\Gamma^+(v)|)$. Split by the degree of the lower endpoint v: - **Edge incident to a light node.** Then v is light ($u \succ v$ gives $\deg(v) \leq \deg(u)$), so $|\Gamma^+(v)| < t = \sqrt{m}$. Summing over all m edges: $O(m\sqrt{m}) = O(m^{3/2})$. - **Edge between two heavy nodes.** By the lemma, $|\Gamma^+(v)| \leq 2\sqrt{m}$. There are at most m such edges: $O(2m\sqrt{m}) = O(m^{3/2})$.

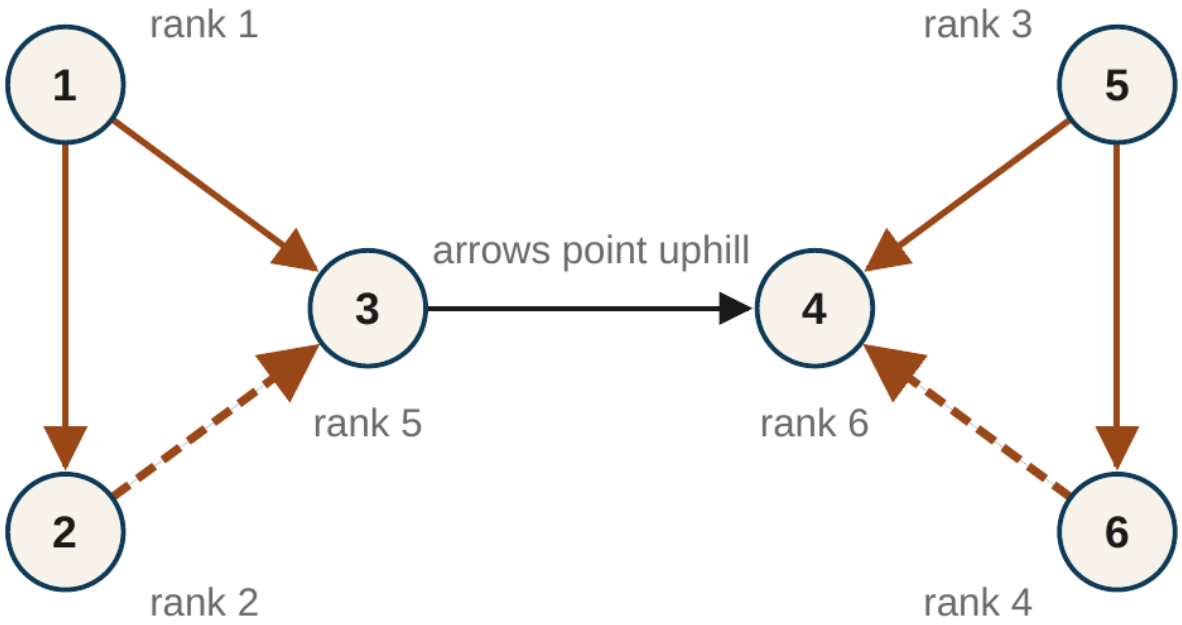


Figure 18.5: NodeIterator++ points every edge uphill.

Total work: $O(m^{3/2} + m^{3/2}) = O(m^{3/2})$. □

Tightness for listing (lollipop graph). Take G = complete graph K_t ($t = \sqrt{n}$ nodes) plus $n - t$ pendant nodes attached to one vertex of the clique. Then $m = \binom{t}{2} + n - t = \Theta(n)$, and the number of triangles is $\binom{t}{3} = \Theta(t^3) = \Theta(n^{3/2}) = \Theta(m^{3/2})$. Any algorithm that *lists* the triangles must output $\Theta(m^{3/2})$ of them, so $O(m^{3/2})$ is tight for listing (output-sensitive lower bound). It is not a lower bound for *counting* (§2.3). □

PRAM analysis of NodeIterator++. The intersection tests for different edges are independent, and each is a merge of two sorted lists, of depth $O(\log n)$; adding up the counts is a reduction of depth $O(\log m) = O(\log n)$. So $W = O(m^{3/2})$ and $D = O(\log n)$, and Brent gives $T_p \le O(m^{3/2}/p + \log n)$. For $p \approx m^{1/2}$ processors that is $O(m)$ time.

Figure 18.5 shows the one trick of NodeIterator++: we orient every edge towards its higher end in the degree order (ties broken by label), so that each triangle is found exactly once.

Check your reading.
Q4. In the $O(m^{3/2})$ proof for NodeIterator++, set the threshold t = sqrt(m). For a graph with m = 10^6 edges, the number of heavy nodes (degree ≥ t = 1000) is at most:
(A) 500 (B) 1000 (C) 2000 (D) 10^6

4. DPRAM / Spark — co-partitioned PageRank and two-pass triangle counting

4.1 PageRank in Spark: join-flatMap-reduceByKey

The natural Spark implementation of one power-iteration step (the classic join–flatMap–reduceByKey formulation, re-derived here):

```
# links: RDD[(url, Iterable[neighbour])]  --- adjacency list
# ranks: RDD[(url, rank_value)]           --- current rank vector
```

```
contributions = links.join(ranks).flatMap(
    lambda kv: [(nb, kv[1][1] / len(kv[1][0])) for nb in kv[1][0]]   # kv = (url,
      (neighbours, rank))
)
new_ranks = contributions.reduceByKey(lambda a, b: a + b) \
                         .mapValues(lambda r: 0.15 / n + 0.85 * r)
```

Byte accounting per iteration (shuffle-join variant). - `links.join(ranks)` is a wide dependency: both `links` and `ranks` are shuffled by URL hash. - `links` shuffle: each URL's adjacency list crosses the network once. Bytes: $\sum_v(\text{outdeg}(v) \cdot b_{\text{url}}) = m \cdot b_{\text{url}}$. - `ranks` shuffle: one rank value per URL. Bytes: $n \cdot (b_{\text{url}} + b_{\text{rank}})$. - Total shuffle bytes (join): $O(m \cdot b_{\text{url}} + n \cdot b_{\text{rank}})$. - `reduceByKey`: partial sums are map-side combined (associative `+`): each of the P map partitions sends at most one partial sum per destination URL, so at most $\min(m, Pn)$ records, reduced to one float per URL at the reducer. Bytes: $O(\min(m, Pn) \cdot (b_{\text{url}} + b_{\text{float}}))$. - For a 3-billion-edge graph ($m = 3 \times 10^9$), 4-byte floats, 8-byte URL keys: the `links` shuffle is $3 \times 10^9 \times 8 \approx 24$ GB and the contribution shuffle up to $3 \times 10^9 \times 12 \approx 36$ GB, so join + reduceByKey shuffle ≈ 60 GB per iteration (plus the n-scale `ranks` term). At 50 iterations: ≈ 3 TB total shuffle — expensive.

4.2 Hash-partitioning eliminates the per-iteration join shuffle

Theorem (co-partitioned PageRank; shuffle-free joins). If (i) `links` is persisted with a hash partitioner keyed by URL, (ii) `mapValues` (not `map`) is used for the rank update, and (iii) the `reduceByKey` that produces the new `ranks` uses a partitioner equal to that of `links` (same number of partitions, same hash function), then all iterations after the first require zero shuffle bytes for the `links.join(ranks)` step.

Proof. Co-partition `links` and `ranks` with the same `HashPartitioner(numPartitions)`: both RDDs assign URL u to partition $h(u) \mod P$ for the same hash h. A `join` of two co-partitioned RDDs with equal partitioners is a **narrow dependency** in Spark (Zaharia et al. §4; their §3.2.2 applies it to PageRank): partition i of the output depends only on partition i of each parent. Spark detects this and executes the join without a shuffle — no shuffle bytes are written or read. (Running the task where both partitions are cached is a scheduler preference, not a guarantee; a partition may still be read from another executor, but it is not shuffled.)

`mapValues` preserves the partitioner (it maps values but not keys, so the partition assignment of each key is unchanged). *Honest scope:* the `flatMap` that emits per-edge rank contributions does **not** preserve the partitioner, so the closing `reduceByKey` still shuffles every iteration — what it moves is the map-side-combined contribution table (at most one partial sum per destination key per partition, so at most $\min(m, Pn)$ records, and the same in both strategies). The theorem eliminates the *join-side* shuffle — the $O(m)$ links table.

After iteration 0 (which materialises both `links` and an initial `ranks` co-partitioned with the same scheme), every subsequent iteration's `join` is narrow: join-side shuffle bytes (iterations $1, \ldots, K-1$) $= 0$. □

Byte-accounting comparison (100 iterations, same graph): | Strategy | Iteration-0 shuffle | Join-side shuffle (iter 1–99) | Contribution shuffle (iter 1–99) | Total | |—|—|—|—|—| | Naive (re-shuffle each time) | $O(m+n)$ bytes | $O(m+n)$ bytes/iter (links + ranks) | $O(\min(m, Pn))$ bytes/iter (combined) | $O(100\,(m+n))$ | | Co-partitioned links (persist) | $O(m+n)$ bytes (setup) | **0** (narrow join) | $O(\min(m, Pn))$ bytes/iter (combined) | $O(m) + O(100 \min(m, Pn))$ |

For $m = 3 \times 10^9$ the links table is ≈ 24 GB per shuffle pass ($3 \times 10^9 \times 8$ bytes), so the *avoided* join-side traffic is approximately 99×24 GB $\approx$ **2.4 TB**. The contribution shuffle (the `reduceByKey` step, which emits at most one aggregated float per destination URL per partition) remains every iteration, the same in both strategies — it does not vanish. It is the small term only when

$m \gg Pn$ (many links per page, modest P); on Lab 6's graph ($m \approx 9.4n$, $P = 8$) the two terms are about the same size. This is the U5 sort-merge/co-partition lesson instantiated for iterative graph algorithms: the one-time cost of co-partitioning amortises across iterations.

Implementation note. In code:

```
links = sc.textFile(graph_file).map(parse_adjacency) \
          .partitionBy(numPartitions) \
          .persist()
ranks = links.mapValues(lambda _: 1.0 / n)  # initialise; inherits the
  partitioner
for _ in range(num_iterations):
    contribs = links.join(ranks, numPartitions).flatMap(...)
    ranks = contribs.reduceByKey(add, numPartitions).mapValues(lambda r: 0.15/n +
      0.85*r)
    # same numPartitions and the default hash: ranks carries the same partitioner
      as links
```

In PySpark two partitioners are equal only if both the number of partitions and the hash function are equal. So pass the same `numPartitions` everywhere and keep the default hash: a custom `partitionFunc` on one side (or a missing `numPartitions`, which lets the count drift) makes them unequal, and every join shuffles again. Lab 6 runs this loop.

4.3 Triangle counting in MapReduce: two-phase NodeIterator++

The two-phase MapReduce implementation of NodeIterator++ (Suri–Vassilvitskii WWW 2011):

Phase 1 — Build sorted adjacency lists with degree-priority orientation. Deciding $v \succ u$ needs the degrees of both endpoints, and a reducer keyed by u sees only $\deg(u)$, so a first round attaches the degrees.

- **Round 1a (attach degrees). Map:** emit (u, v) for every edge $\{u, v\} \in E$, both orientations. **Reduce:** for each node u, collect all neighbours, so $\deg(u) = |\Gamma(u)|$; send $(v, (u, \deg(u)))$ to every neighbour v.
- **Round 1b (orient). Reduce:** for each node u, the records $(v, \deg(v))$ of all its neighbours arrive, and their number is $\deg(u)$; keep the v with $v \succ u$ and emit the sorted oriented adjacency list $\{(u, \Gamma^+(u))\}$ where $\Gamma^+(u) = \{v \in \Gamma(u) : v \succ u\}$.

Shuffle bytes (phase 1): $2m$ edge records in each round, each $O(b_{\text{url}})$ bytes. Total: $O(m \cdot b_{\text{url}})$.

Phase 2 — Generate and check wedges.

- **Map:** for each oriented adjacency list $(u, \Gamma^+(u))$: for each pair $(v, w) \in \binom{\Gamma^+(u)}{2}$ (a *wedge* rooted at u with $v \succ u$ and $w \succ u$): emit key $(\min(v, w), \max(v, w))$ with value `wedge`. Also emit key $(\min(u, v), \max(u, v))$ with value `edge` for every oriented edge (u, v), $v \in \Gamma^+(u)$, so that a wedge and the edge that closes it meet at the same key.
- **Reduce:** for each key (a, b): if both at least one `wedge` and at least one `edge` record arrive, the edge $\{a, b\}$ closes a triangle for each wedge; emit the count.
- **Count:** sum all reducer outputs; no division is needed. A triangle $\{u, v, w\}$ with $u \prec v \prec w$ is found exactly once: as the wedge (v, w) rooted at its lowest vertex u, closed by the edge $\{v, w\}$.

Shuffle bytes (phase 2). The number of wedges emitted per oriented edge (u, v) (root u, higher-priority endpoint v) is $|\Gamma^+(u)| - 1 \le \deg(u)$. Summing over all oriented edges:

$$\text{wedge records} = \sum_v \binom{|\Gamma^+(v)|}{2} \le \frac{1}{2} \sum_v |\Gamma^+(v)|^2 \le \frac{1}{2} \cdot 2\sqrt{m} \cdot \sum_v |\Gamma^+(v)| = m^{3/2},$$

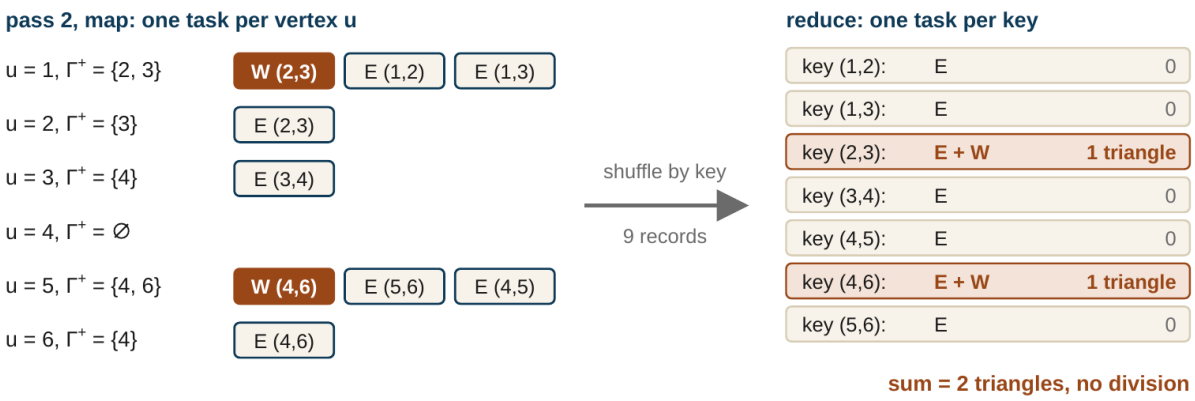


Figure 18.6: The second MapReduce pass on the two-triangle graph.

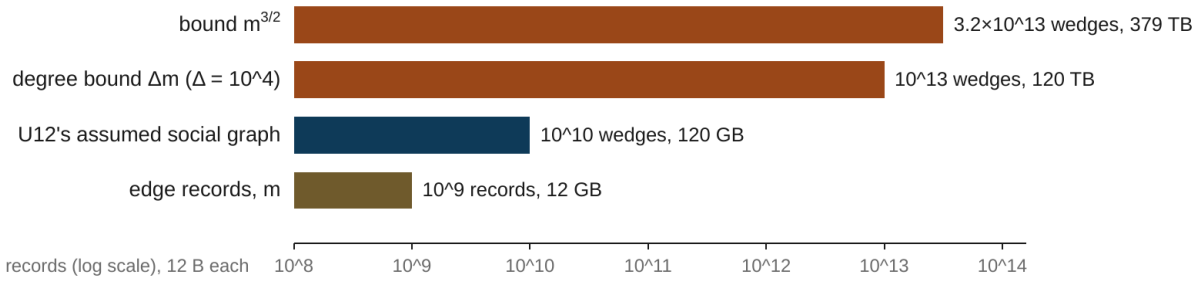


Figure 18.7: The price of the second pass for $m = 10^9$ edges.

by the lemma of §3.2 ($|\Gamma^+(v)| \le 2\sqrt{m}$) and $\sum_v |\Gamma^+(v)| = m$ (each edge is oriented once). No assumption on the maximum degree $\Delta = \max_v \deg(v)$ is needed; the degree bound $\frac{1}{2}\sum_v \deg(v)^2 \le \frac{1}{2}\Delta \sum_v \deg(v) = \Delta m$ is smaller only when $\Delta < \sqrt{m}$. Edge records: m. Total shuffle bytes (phase 2): $O(m^{3/2} \cdot b_{\text{key}})$.

Worked example. For $m = 10^9$ edges with $\Delta = 10^4$ (below $\sqrt{m} \approx 3.2 \times 10^4$): wedge records $\le \Delta m = 10^{13}$ (the $m^{3/2}$ bound gives $\approx 3.2 \times 10^{13}$) — this is large; in practice, the degree filter $t = \sqrt{m}$ restricts most wedge emission to lighter nodes, and the actual wedge count is much smaller for real-world graphs (which are sub-lollipop). For a social graph with $m = 10^9$ and empirical wedge count $\approx 10^{10}$ (10× edges), phase-2 shuffle $\approx 10^{10} \times 12$ bytes $= 120$ GB. Achievable on a 100-machine cluster in one shuffle round.

Figure 18.6 runs the second pass on the two-triangle graph, and Figure 18.7 prices it for a graph with $m = 10^9$ edges.

Check your reading.

Q5. Persisting `links` with a `HashPartitioner` and using `mapValues` for the rank update (and the rank-producing `reduceByKey` uses a partitioner equal to that of `links`) makes every iteration after the first cheaper. Precisely which shuffle disappears?

(A) all shuffle – iterations 1..K-1 move zero bytes over the network (B) the join-side shuffle of the links table (the O(m) term); the `flatMap` breaks the partitioner, so `reduceByKey` still shuffles at most min(m, Pn) combined contribution records every iteration (C) the `reduceByKey` shuffle; the `links.join(ranks)` still re-shuffles the adjacency lists each iteration (D) iteration 0's setup shuffle

5. Graph partitioning

5.1 The problem and its hardness

Definition. A **k-partition** of $G = (V, E)$ is a partition $V = V_1 \cup \cdots \cup V_k$ (disjoint, non-empty) with $|V_i| \le \lceil |V|/k \rceil + B$ for a balance parameter B. The **edge-cut** is $|\{(u, v) \in E : V_i \ni u,\ V_j \ni v,\ i \ne j\}|$. The **balanced graph partitioning** problem minimises the edge-cut subject to the balance constraint.

Hardness. Balanced graph partitioning is NP-hard even for $k = 2$ (proved by reduction from bisection-width, which is NP-hard by reduction from MAX-CUT on general graphs; see Garey–Johnson). At scale, exact solutions are out of reach; the two principled relaxations used in practice are spectral relaxation and vertex-cut partitioning.

5.2 Spectral relaxation: the Fiedler vector (G9 deep exercise)

Graph Laplacian. For an undirected unweighted graph G, the **Laplacian** is $L = D - A$, where D is the degree matrix. L is symmetric positive semidefinite; its eigenvalues $0 = \lambda_1 \le \lambda_2 \le \cdots \le \lambda_n$ are non-negative.

Algebraic connectivity. The second-smallest eigenvalue λ_2 is the **Fiedler value** (Fiedler 1973); the corresponding eigenvector f is the **Fiedler vector**. $\lambda_2 = 0$ iff the graph is disconnected (the null space of L has dimension = number of connected components). A larger λ_2 means the graph is "harder to cut" — the spectral gap governs mixing and cut quality.

Spectral bisection. Assign node v to V_1 if $f_v \ge 0$ and to V_2 otherwise (or threshold at the median for balance). This is the **spectral relaxation** of the NP-hard bisection problem: it minimises the continuous relaxation of the Rayleigh quotient $x^T Lx / \|x\|^2$ over $\{\pm 1\}^n$ (hard) to $\mathbb{R}^n$ orthogonal to $\mathbf{1}$ (easy — solved by the Fiedler eigenvector).

Why it works in practice. The Rayleigh quotient is $x^T Lx = \sum_{\{u,v\} \in E} (x_u - x_v)^2$. Minimising this with $x \perp \mathbf{1}$ and $\|x\| = 1$ encourages connected vertices to have similar values and far-apart vertices to have different values — the Fiedler vector organises vertices by "algebraic distance" in the graph. The spectral partition often achieves close-to-optimal cuts on well-structured (expander-free) graphs.

Computation. The Fiedler vector belongs to the *second-smallest* eigenvalue λ_2 of L — so naive power iteration on L is the WRONG tool (it converges to the *largest* eigenvalue's eigenvector). Two correct routes at scale: (i) **spectral shift**: power-iterate $\lambda_{\max} I - L$ (with $\lambda_{\max}$ bounded by $2 \deg_{\max}$), deflating the constant vector — its top eigenvector is L's smallest-nonzero one; (ii) **inverse iteration** (solve $Lx^{(k+1)} = x^{(k)}$ per step — each step a distributed linear solve, U8's machinery). Both are matvec-loops with the same distributed structure as PageRank. Full distributed implementation is a deep exercise (G9).

Figures 18.8, 18.9 and 18.10 take, in order, the relaxation, the Fiedler vector of a small graph, and the PageRank-shaped loop that computes it at scale.

5.3 Edge-cut vs vertex-cut: PowerGraph and the power-law graph insight

For general graphs, partitioning into k parts assigns vertices to machines — this is **vertex partitioning / edge-cut**: edges crossing partition boundaries must be communicated during each computation step. On a graph where edges are distributed like the nodes, edge-cut is manageable.

Power-law degree distribution. Many real-world graphs (web, social networks) follow a power-law: $\Pr[\deg(v) = d] \propto d^{-\gamma}$ with $\gamma \in (2, 3)$. A small number of **hub** nodes have degree

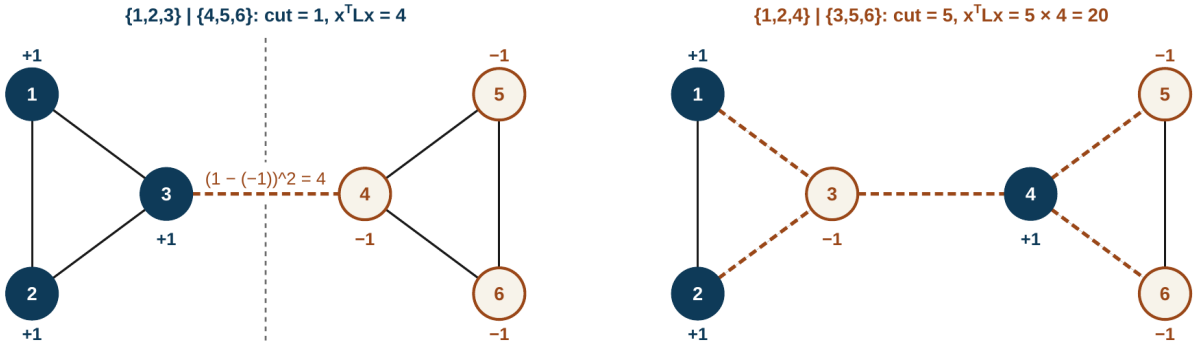


Figure 18.8: Cutting a graph in two: the spectral relaxation.

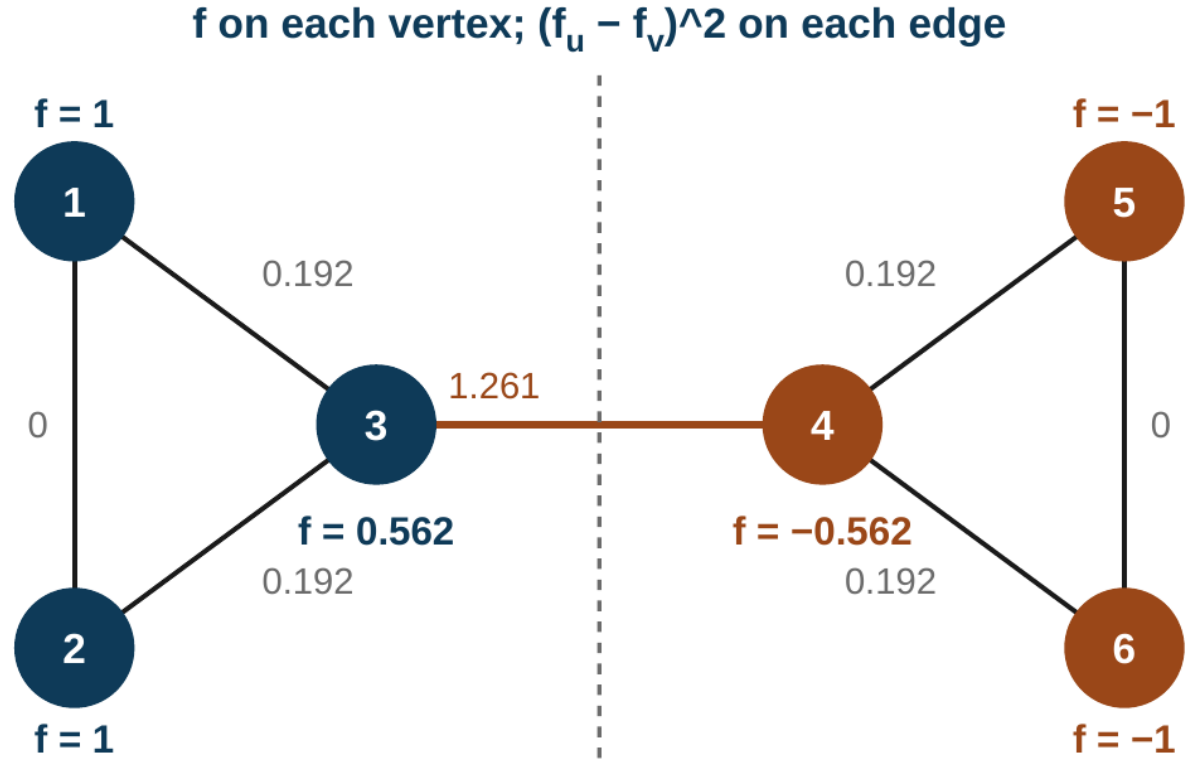


Figure 18.9: The Fiedler vector of a small graph.

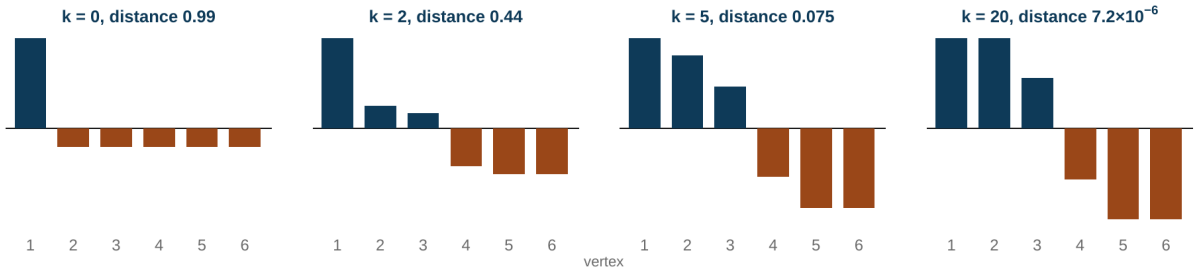


Figure 18.10: Computing f at scale with a loop shaped like PageRank.

of order $n^{1/(\gamma-1)}$ — between $\sqrt{n}$ and n for $\gamma \in (2,3)$: $o(n)$, but polynomially large. For vertex-partitioned edge-cut on p machines, assigning such a hub to any one machine leaves its neighbours spread over all p, so about $(1-1/p)$ of its edges cross to other machines — an edge-cut of $\approx (1-1/p)\deg(v) = \Theta(n^{1/(\gamma-1)})$ per hub.

Vertex-cut (PowerGraph; Gonzalez et al. OSDI 2012). Partition E instead of V: each edge is assigned to exactly one machine; vertices may be **replicated** across machines that hold their incident edges. A vertex v has a **mirror** on each machine holding at least one edge of v; one mirror is the **master**. During a computation step, each machine computes its local partial result using its edges and local mirror values; masters then aggregate across mirrors (a gather phase) and distribute updates back (a scatter phase).

Why vertex-cut wins on power-law graphs (a systems worked-example, not a headline theorem). Two honest facts from Gonzalez et al. (2012) carry the whole insight. First, the *negative* one (their Thm 5.1): a random **edge-cut** on p machines cuts an expected $1-1/p$ fraction of all edges — with any realistic p, essentially *every* edge crosses machines, so edge-cut communication is hopeless on hub-dominated graphs. Second, the *positive* one (their Thm 5.2): under a random **vertex-cut**, a vertex of degree d is expected to be replicated on

$$\mathbb{E}\big[\,|A(v)|\,\big] \;=\; p\left(1-\left(1-\tfrac{1}{p}\right)^{d}\right)$$

machines — a curve that grows with d but *saturates at* p. Averaging over a power-law degree distribution, the expected replication factor stays small: the few huge hubs cost at most p mirrors each (they were unsplittable under edge-cut anyway), while the long tail of low-degree vertices — the overwhelming majority — is barely replicated at all. That saturation, against edge-cut's cut-everything baseline, is the entire reason vertex-cut wins on natural graphs; greedy placement improves further on random. The deeper frame — *why* the computation can be split per-edge at all — is the GAS decomposition of §6.3, of which this is the placement corollary.

Check your reading.
Q6. A power-law graph has hub vertices of degree d; you have p machines. Using the unit's two facts – a random edge-cut cuts an expected (1 - 1/p) fraction of ALL edges, and a random vertex-cut replicates a degree-d vertex on an expected p(1 - (1 - 1/p)^d) machines – explain in 2-3 sentences why PowerGraph partitions edges rather than vertices.

6. Distributed vertex programming: BSP → Pregel → GAS → GraphX

This section is one of the course's load-bearing gems, and it is worth saying plainly what the gem *is*: not any framework's API, but a four-layer stack of ideas — a **cost model** (BSP), an **abstraction** (think-like-a-vertex), a **unification** (Gather–Apply–Scatter), and a **dissolution into dataflow** (GraphX) — under which one mathematical fact stays invariant: *an iterative graph algorithm is a fixpoint iteration, and every layer of the stack is a different way of pricing and executing the same fixpoint.* The frameworks of any given decade come and go; the stack of ideas stays.

6.1 The cost model: bulk-synchronous parallelism (Valiant 1990)

Valiant (CACM 1990) proposed BSP as a *bridging model* — playing, for parallel computation, the role the von Neumann machine plays for sequential: a stable contract between algorithm designers above and hardware designers below. A BSP machine is (i) a set of processors with local memory, (ii) a router delivering point-to-point messages, and (iii) **barrier synchronisation as**

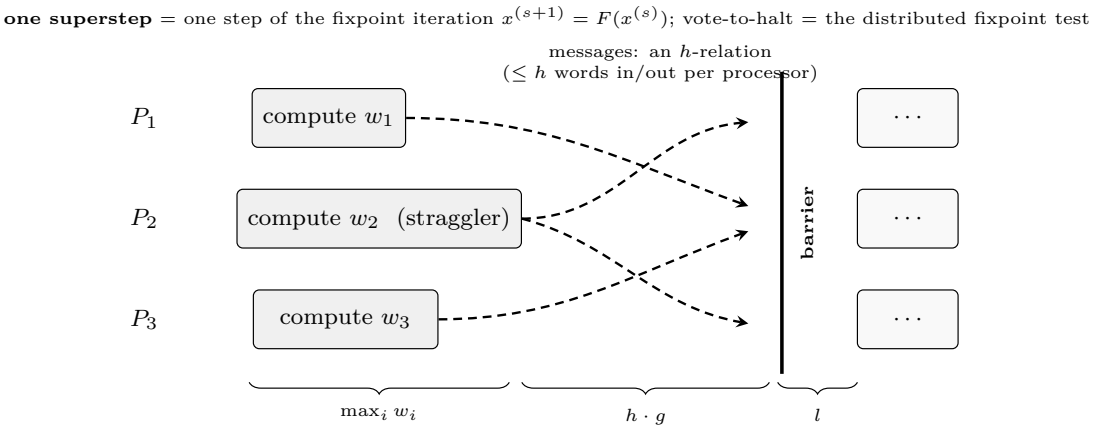


Figure 18.11: One BSP superstep (Valiant 1990), the cost unit of distributed vertex programming. Every processor computes locally (the slowest, the *straggler*, sets $\max_i w_i$), messages move as a bulk h-relation priced $h \cdot g$, and the barrier costs l: cost $= \max_i w_i + hg + l$, summed over supersteps. Read mathematically, a superstep is one step of a fixpoint iteration — PageRank's TV-contracting power step (§2.2) or SSSP's monotone relaxation (F.C2b) — and Pregel's vote-to-halt is the distributed test that the fixpoint has been reached.

the one non-local primitive. Computation proceeds in **supersteps**: in each, every processor computes locally, sends/receives messages, and then all wait at a barrier.

The cost of one superstep in which processor i does w_i local work and no processor sends or receives more than h words (an *h-relation*):

$$\text{cost} \;=\; \max_i w_i \;+\; h\,g \;+\; l,$$

where g is the router's *gap* (time per word under continuous traffic — inverse bandwidth, in flop units) and l is the *barrier latency*. A full algorithm costs the sum over its supersteps. Communication is charged **in bulk** ($h \cdot g$), not per message — that is the model's defining abstraction, and it is why BSP composes so cleanly with U0's work/depth and F.E4's measured α–β: one superstep is priced by exactly three legible terms. *Honest caveat:* the barrier l is paid every superstep, and the slowest processor sets $\max_i w_i$ — BSP over-synchronises on skewed workloads, which is the opening the asynchronous engines exploit (§6.3).

6.2 The abstraction: Pregel and think-like-a-vertex (Malewicz et al. 2010)

Pregel instantiates BSP for graphs. The state lives *at the vertices*; the program is one local function. In superstep s, every **active** vertex v: 1. reads the messages sent to it in superstep $s-1$; 2. updates its own value (and possibly its out-edges); 3. sends messages along out-edges, delivered in superstep $s+1$; 4. optionally **votes to halt** (becomes inactive); an inactive vertex is reawakened only by an incoming message.

Termination: the computation ends when *all vertices are simultaneously inactive and no messages are in transit*. Read that as mathematics, not plumbing: vote-to-halt is a **distributed fixpoint test** — a vertex halts when its local update has stopped changing its state, and global termination is precisely "the iteration has reached a fixpoint."

Two supporting devices: **combiners** (a commutative–associative merge of messages headed to the same vertex — the map-side combine of U5, and the same algebraic requirement as U1's reduction monoid; Pregel's SSSP saw a fourfold message reduction) and **aggregators** (a global

reduce visible to all vertices in the next superstep — how convergence is *detected*, e.g. "total residual $< \varepsilon$").

Pregel-PageRank is §2's power iteration verbatim: v sends $r_v/\text{outdeg}(v)$ to its out-neighbours and updates $r_v \leftarrow 0.15/n + 0.85 \sum_u c_{uv}$ — one superstep per power-iteration step, convergence priced by §2.2's TV-contraction. Pregel-SSSP relaxes tentative distances monotonically downward — a decreasing iteration on a well-founded order, so it *must* terminate (F.C2b's lens). Synchrony also buys correctness for free: within a superstep there is nothing to race on, so the model is deadlock- and data-race-free by construction.

6.3 The unification: Gather–Apply–Scatter (Gonzalez et al. 2012)

GAS factors any vertex program into three phases over the neighbourhood of u:

$$\underbrace{\Sigma \leftarrow \bigoplus_{v \in \text{Nbr}(u)} g\big(D_u, D_{(u,v)}, D_v\big)}_{\textbf{Gather}\ \text{(a comm.-assoc. reduce)}} \qquad \underbrace{D'_u \leftarrow a(D_u, \Sigma)}_{\textbf{Apply}} \qquad \underbrace{D_{(u,v)} \leftarrow s(D'_u, D_{(u,v)}, D_v)}_{\textbf{Scatter}}$$

This subsumes both Pregel's message-passing *and* shared-memory vertex programs — the unification is the point. And it has one load-bearing requirement: **Apply must be sub-linear (ideally constant) in the degree.** That single condition is what lets a hub vertex's Gather be *split across machines* — each machine reduces its local edges, the partial Σs combine (associativity again), and only then does the master Apply once. Edge-parallelism is what makes the vertex-cut of §5.3 executable, not just a placement idea: the two are one design, GAS the algebra and vertex-cut the layout. *Honest caveats:* the win is specific to high-degree/power-law graphs (on bounded-degree graphs a good edge-cut is competitive), and an Apply that is linear in degree silently forfeits the benefit. Asynchronous execution (committing updates immediately, no barrier) can converge where synchronous cannot — greedy graph colouring is the canonical example — at the price of determinism.

6.4 The dissolution: GraphX and graph-parallel as dataflow (Gonzalez et al. 2014)

GraphX closes the loop with U5: the property graph is *just two collections* — a vertex table and an edge table — and one derived **triplets view**, the three-way join edges $\bowtie$ $\text{vertices}_{\text{src}}$ $\bowtie$ $\text{vertices}_{\text{dst}}$. The whole of vertex programming reduces to one operator, `mrTriplets`: *map* over triplets, *group by* destination, reduce with a commutative–associative function. One Pregel superstep = one join + one group-by-aggregate. The entire Pregel API is a few dozen lines over Spark's RDDs, and fault tolerance comes free from U5's lineage. `MATHS->CS:` the abstraction stack *dissolves* — graph-parallel computation was dataflow all along, and the graph-specific engineering reappears as join optimisation (indexing, join elimination) and incremental view maintenance. Lab 6 runs both: the hand-rolled joins of §4, where the shuffle accounting stays visible, and GraphX's `staticPageRank`, the production form of the same joins (GraphFrames offers the same on DataFrames, with a Python API).

6.5 The invariant underneath: fixpoints, four ways

Line up the four layers over the same algorithm and the gem is visible:

Layer	What it fixes	PageRank there
BSP	the **price** of one iteration: $\max_i w_i + hg + l$	one power-iteration step per superstep
Pregel	the **program**: local update + vote-to-halt	$r \leftarrow 0.15/n \cdot \mathbf{1} + 0.85\, S\, r$, halt on small residual

Layer	What it fixes	PageRank there
GAS	the **algebra**: Gather $\oplus$ comm.-assoc., Apply sub-linear	$\oplus = +$, Apply is one affine map
GraphX	the **execution**: join $\rightarrow$ map $\rightarrow$ group-by	two exchanges per iteration: new ranks shipped to the edge partitions, messages aggregated back to the vertices

The mathematics is the invariant: PageRank converges because §2.2's damped walk is a TV-contraction (Perron–Frobenius); SSSP, connected components, and label propagation converge because they are monotone iterations on well-founded (or complete-lattice) orders — F.C2b's Kleene fixed point, met here in production clothing. **The frameworks changed four times in a quarter-century; the two convergence arguments did not.** That is what "a gem that stays" means, and it is the standard this course holds its systems content to.

GraphX / GraphFrames (availability note). Apache GraphX (OSDI 2014) and GraphFrames (2016, DataFrames) are both mature in Spark 3.x. Lab 6 runs the plain-RDD loop with explicit joins, so the shuffle accounting stays visible, next to GraphX's `staticPageRank` (§8); any system exposing shuffle bytes and stage counts (the Spark UI) serves the pedagogical goal.

GraphBLAS tie (U6 connection). The Pregel gather-apply-scatter model corresponds to sparse matrix-vector multiply: the adjacency matrix A is the communication pattern, the vertex values are the vector, and the scatter/gather is the matrix-vector product. The **GraphBLAS** standard (Kepner et al. 2016) formalises graph algorithms as linear algebra operations over semirings, enabling the U6 matrix-multiplication machinery (blocked, comm-avoiding) to be applied to graph algorithms. This is the natural next abstraction; it is a deep connection, not a course prerequisite.

Check your reading.

Q7. The unit's LLM-lens box reads the attention matrix $softmax(QK^T$ / sqrt(d)) as a weighted directed graph over tokens, rebuilt every layer. In 2-3 sentences, map this onto the section-6 stack: what plays the vertex value, what is one superstep, and which of the unit's two convergence arguments (TV contraction vs monotone/Kleene fixpoint) would you reach for – and why is neither immediate?

7. The invariant (U12 instantiation)

RAM: PageRank — $O(m)$ per iteration, $O(Km)$ total; damping contracts TV by $(1-\alpha)^k$; existence and uniqueness of π by Perron–Frobenius. Triangle counting — $O(n^\omega)$ via A^3; $O(m^{3/2})$ via NodeIterator++; tight on the lollipop (for listing). Graph partitioning — NP-hard; spectral relaxation (Fiedler) and vertex-cut are the principled approximations.

PRAM: PageRank iteration — $W = O(m)$, $D = O(\log n)$ per step; Brent gives linear speed-up until $p \approx m/\log n$. NodeIterator++ — $W = O(m^{3/2})$; parallel frontier intersections with depth $O(\log n)$ per edge.

DPRAM / Spark: PageRank — co-partitioned `links` + `mapValues` eliminates the join-side (links) shuffle after iteration 0; the $O(\min(m, Pn))$-scale contribution shuffle remains; byte saving $= (K-1) \cdot m \cdot b$ per run (the links share). Triangle counting — 2-phase MapReduce; phase-1 shuffle $O(m)$ (two rounds: attach degrees, then orient); phase-2 shuffle $O(m^{3/2})$ wedge records (one round). Graph partitioning — edge-cut vs vertex-cut: vertex-cut wins on power-law graphs by sub-linear replication factor; Pregel vertex-program model + combiners for graph-parallel algorithms.

---

8. Lab (Lab 6 — PageRank two ways)

Lab 6 — PageRank two ways: the `lab6-pagerank-two-ways/` folder of the course repository, and the lab chapter Lab 6 in the Laboratory exercises part; Python (PySpark RDDs) and Scala (GraphX), the same receipt pattern as the other labs, everything free and local on your laptop.

What the lab runs. A power-law graph of 10^5 vertex ids and about 9.4×10^5 edges; PageRank for 10 iterations (links followed with probability 0.85, teleport $\alpha = 0.15$ as in §2.1) three ways: the RDD loop of §4.1 with the partitioning lost (naive), the same loop co-partitioned as in the implementation note of §4.2, and GraphX's `staticPageRank` (§6.4) on the same graph. Spark's task metrics give the shuffle bytes of the join and of the contributions, separately, for every iteration; the two RDD versions must give the same ranks, the co-partitioned join must move nothing, and GraphX must find the same top vertices. (The RDD loop is the classic Spark formulation, not §2.1's chain: it starts with rank $1/n$ only on the vertices that have out-links, and adds the teleport term $0.15/n$ only to a vertex that receives a contribution, so a vertex without in-links drops out of the rank table after the first iteration, and a vertex whose in-links all come from such vertices drops out one iteration later; and it passes on nothing from a vertex without out-links. Rank leaks every iteration, and after 10 iterations the ranks sum to about 0.91. §2.1's chain gives every vertex the teleport term and spreads dangling rank uniformly, so its ranks sum to 1.)

Deliverable: the receipt, with one or two sentences on each of: why co-partitioning removes the join's shuffle but not the contributions'; and why GraphX, moving about as many bytes, takes a fraction of the time.

Optional deep dives: the three experiments below are the original, larger design of this lab. They are not part of the submission. Lab 6 runs a laptop-sized form of Experiment 1.

Experiment 1 (optional) — PageRank: co-partitioned vs naive, scaled up.

Generate a synthetic power-law graph ($n = 10^5$ nodes, $m = 10^6$ edges, Barabási–Albert model) stored as an edge list.

Implement two variants: - **Naive:** `links.join(ranks)` with no explicit co-partitioning; let Spark re-shuffle on every iteration. - **Co-partitioned:** persist `links` with `HashPartitioner(P)`; initialise `ranks` with the same partitioner; use `mapValues` for rank updates.

For each variant, run 20 iterations. For each iteration, record (via SparkListener or UI): - shuffle bytes written and read - wall-clock time

Optional questions: 1. Plot shuffle bytes per iteration for both variants, the join and the contributions separately. Does the co-partitioned join move anything in any iteration? 2. What is the total shuffle bytes saving over 20 iterations? Express as a multiple of the graph size $m \cdot b_{\text{edge}}$. 3. State in three-lens vocabulary: which lens does the saving live in, and why does it arise?

Experiment 2 — Triangle counting: NodeIterator++ vs naive on a power-law graph.

Using the same synthetic graph from Experiment 1.

Implement: - **Naive:** for each node v, for each pair of neighbours (u, w), check if (u, w) is an edge (using a broadcast hash set of edges). Count triangles by summing over all nodes and dividing by 3 (each triangle is found once at each of its three vertices). - **NodeIterator++:**

implement the two-pass MapReduce as described in §4.3. Count triangles from the phase-2 reducer output.

Measure: total shuffle bytes (Spark UI) for each approach; wall-clock time; number of Spark stages.

Also include a **G8 diameter exercise data point** (see Exercise E12.5): on the same graph, run a BFS from a random source node using iterative `join` + `filter` and record the diameter (largest eccentricity encountered). This is the G8 shallow-graph exercise integrated into the lab.

Questions: 1. How does shuffle bytes compare between the naive and NodeIterator++ implementations? 2. Is the measured triangle count for NodeIterator++ equal to the naive count? (Sanity check.) 3. How many Spark stages does each approach require?

Experiment 3 — Spectral partitioning (preview, G9).

On the synthetic graph, compute the Fiedler vector approximately via 20 iterations of power iteration on $2I - \mathcal{L}$, where $\mathcal{L}$ is the normalised Laplacian, deflating its top eigenvector $D^{1/2}\mathbf{1}$ (power iteration on $\mathcal{L}$ itself finds the eigenvector of the *largest* eigenvalue, not the Fiedler vector), or via inverse iteration (§5.2); use an explicit iterative implementation. Bisect the graph at the median of the Fiedler values.

Measure: edge-cut of the bisection; balance (fraction of nodes in each part). Compare to a random bisection (assign each node uniformly at random to one of two parts).

Question: how much better is spectral bisection than random bisection in edge-cut? Can you relate the Fiedler value to the edge-cut via the Cheeger inequality (stated but not proved: $\lambda_2/2 \le \phi(G) \le \sqrt{2\lambda_2}$ where ϕ is the conductance)?

Deliverable of the optional deep dive. Notebook + half-page cost-accounting for each experiment in the three-lens vocabulary.

---

9. Lean certificate target

Target (new — U12): TV-contraction step of damped PageRank.

The core claim to certify: one step of the damped power iteration contracts the total-variation distance to stationarity by factor $(1 - \alpha)$.

Lemma (one-step TV contraction). Let $0 < \alpha < 1$, let Q be column-stochastic and $P = \frac{\alpha}{n}\mathbf{1}\mathbf{1}^T + (1-\alpha)\,Q$ the damped chain, with stationary distribution π; then for every distribution v, the total-variation distance from Pv to π is at most $(1 - \alpha)$ times the total-variation distance from v to π.

```
-- SCHEMATIC target (finite-state Markov chain over a type V; names align to
  BMLSPar.lean):
-- P = α * teleport + (1 - α) * Q, where teleport sends every state to the
  uniform distribution
-- pi : V → ℝ is the stationary distribution

theorem pagerank_tv_contraction
    (α : ℝ) (hα : 0 < α) (hα1 : α < 1)
```

```
    (P Q : V → V → ℝ)  -- damped chain; column-stochastic link chain (Q = S of
      §2.1)
    (pi v : V → ℝ)     -- stationary distribution; current iterate
    (hQ : is_column_stochastic Q)
    (hpi : P_stationarity P pi)
    (hpi_dist : is_distribution pi)
    (hv : is_distribution v)
    (hP : P = damped_chain α Q) :
    tv_dist (P_step P v) pi ≤ (1 - α) * tv_dist v pi := by
  sorry  -- SCHEMATIC: cert target; the proof uses teleportation annihilation +
    Q-contractivity
```

Proof sketch (for the cert). The key step is: $Pv - \pi = (1-\alpha)Q(v-\pi)$ (the αR term vanishes because $Ru = \mathbf{1}(\mathbf{1}^T u)/n$ and $\mathbf{1}^T(v-\pi) = 0$). Then $\|Qu\|_1 \le \|u\|_1$ for signed u with $\mathbf{1}^T u = 0$ (because $Q = S$ is column-stochastic and does not increase the ℓ_1 norm of balanced signed vectors). The Lean formalisation requires: (a) a finite-state Markov chain type; (b) a formalisation of TV distance; (c) the teleportation annihilation lemma; (d) the Q-contractivity lemma (one induction on states). This is a self-contained cert on a finite type — no real-analysis axioms needed beyond basic `Finset` and `NNReal` in Mathlib.

Exercise U12.L1 (light). Load the cert stub. State `teleportation_annihilation`: that for any signed measure u with $\mathbf{1}^T u = 0$, $Ru = 0$. Verify the `sorry`-stub type-checks.

Exercise U12.L2 (deep). Complete `Q_contractivity` — prove that for sub-stochastic Q and balanced u, $\|Qu\|_1 \le \|u\|_1$ — and use it, with $Q = S$, to close the main contraction proof.

LLM lens — where this unit lands on the running workload (U0L):
The attention matrix $\mathrm{softmax}(QK^\top/\sqrt{d})$ is a weighted directed graph over tokens, rebuilt every layer; importance propagating through it is power-iteration intuition — this unit's fixed-point lens, applied inside the model rather than over the web.

10. Sources (credited; archived under `scadamale-book-sources`)

Page, L., Brin, S., Motwani, R., and Winograd, T. (1999). *The PageRank Citation Ranking: Bringing Order to the Web.* Stanford InfoLab Technical Report 1999-66. The original PageRank paper. The random-surfer model, damped chain $P = \frac{\alpha}{n}\mathbf{1}\mathbf{1}^T + (1-\alpha)A\,D^{-1}$, and power-iteration convergence are from this source. Our exposition is our own. Available: http://ilpubs.stanford.edu:8090/422/

Haveliwala, T.H. and Kamvar, S.D. (2003). *The Second Eigenvalue of the Google Matrix.* Stanford University Technical Report. The bound $|\lambda_2(P)| \le 1-\alpha$, with equality when the link graph has two or more closed communicating classes, cited in §2.2.

Malewicz, G., Austern, M.H., Bik, A.J.C., Dehnert, J.C., Horn, I., Leiser, N., and Czajkowski, G. (2010). *Pregel: A System for Large-Scale Graph Processing.* SIGMOD 2010. The Pregel vertex-program / superstep / combiner model. Our description of the Pregel model in §6 follows this paper's framework. Available: https://dl.acm.org/doi/10.1145/1807167.1807184

Zaharia, M., Chowdhury, M., Das, T., Dave, A., Ma, J., McCauley, M., Franklin, M.J., Shenker, S., and Stoica, I. (2012). *Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing.* NSDI 2012 (full entry with U5's sources). The join of co-partitioned inputs as a narrow dependency (their §4) and the co-partitioned PageRank (their §3.2.2), cited in §4.2.

Suri, S. and Vassilvitskii, S. (2011). *Counting Triangles and the Curse of the Last Reducer.* WWW 2011. The NodeIterator++ algorithm, the $O(m^{3/2})$ proof, the two-phase MapReduce implementation, and the lollipop tightness example are from this source. Our proof in §3.2 follows the degree-splitting argument of this paper. Available: https://dl.acm.org/doi/10.1145/1963405.1963491

Gonzalez, J.E., Low, Y., Gu, H., Bickson, D., and Guestrin, C. (2012). *PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs.* OSDI 2012. The vertex-cut model, the power-law replication factor analysis, and the gather-apply-scatter abstraction are from this source. Available: https://www.usenix.org/conference/osdi12/technical-sessions/presentation/gonzalez

Fiedler, M. (1973). *Algebraic connectivity of graphs.* Czechoslovak Mathematical Journal 23(2):298–305. The Fiedler value (algebraic connectivity λ_2) and the Fiedler vector for graph bisection are from this source.

Chiba, N. and Nishizeki, T. (1985). *Arboricity and subgraph listing algorithms.* SIAM Journal on Computing 14(1):210–223. The sequential $O(m^{3/2})$ triangle-listing algorithm that underlies NodeIterator++.

Alon, N., Yuster, R., and Zwick, U. (1997). *Finding and counting given length cycles.* Algorithmica 17(3):209–223. Triangle counting in $O(m^{2\omega/(\omega+1)})$ time by fast matrix multiplication, cited in §2.3.

Wikipedia. *PageRank*, *Perron–Frobenius theorem*, *Graph Laplacian*, *Algebraic connectivity*, *Triangle-free graph*, *Pregel (Google)*, *GraphX*, *GraphBLAS*. Background, cross-references, and figures.

---

11. Exercises

Light (examined core):

E12.1 State the damped PageRank matrix $P = \frac{\alpha}{n}\,\mathbf{1}\mathbf{1}^T + (1-\alpha)A\,D^{-1}$, where $\mathbf{1}$ is the all-ones column vector (each column of the teleportation term is α times the uniform distribution $\mathbf{1}/n$). Explain why P is primitive (irreducible and aperiodic) for any connected graph with $\alpha > 0$. What goes wrong if the graph has a dangling node (out-degree 0), for any $\alpha < 1$, and how does the patch S of §2.1 repair it? What does $\alpha = 0$ break in addition?

E12.2 Using the TV-contraction theorem, compute the minimum number of iterations to achieve $\|v^{(k)} - \pi\|_1 \le 10^{-6}$ with $\alpha = 0.15$. For the same accuracy, how many iterations does ℓ_2 convergence (at the rate of the second-eigenvalue modulus $|\lambda_2(P)| = (1-\alpha)\lambda_2$, where λ_2 is the second eigenvalue of the link matrix S; the spectral gap is $1 - |\lambda_2(P)|$) require if $\lambda_2 = 0.9$?

E12.3 In the Spark PageRank implementation, `links` is co-partitioned with `ranks`. Explain, in terms of Spark's narrow vs wide dependency classification, why the `links.join(ranks)` call after co-partitioning is a narrow dependency and requires zero shuffle bytes.

E12.4 Apply the NodeIterator++ degree-split argument to a graph with $m = 10^6$ edges and $\Delta = 10^3 \approx \sqrt{m}$. (a) Count the number of heavy nodes. (b) Bound the total work for edges incident to light nodes; for edges between heavy nodes. (c) Confirm the total is $O(m^{3/2})$. (d) Compute the actual bound for these numbers.

Deep (Module-3 / optional):

E12.5 **(G8 — diameter of shallow graphs).** A graph is **k-shallow** (has treewidth bounded by k) if it admits a tree decomposition of width k. (a) Show that treewidth and diameter do not bound each other (consider the path P_n and the complete graph K_n). For an undirected connected graph, show that BFS from a source s takes $\mathrm{ecc}(s)$ rounds and that $\mathrm{ecc}(s) \le \mathrm{diam}(G) \le 2\,\mathrm{ecc}(s)$. (b) For the web graph (empirically: diameter ≈ 20 for $n \approx 10^{10}$), what does this imply about the treewidth? (c) Implement BFS for diameter estimation in Spark (iterative `join` + `filter` to propagate frontiers); run on the experiment-1 graph and report the diameter and number of BFS rounds.

E12.6 **(G9 — spectral cuts and the Fiedler vector).** Let $L = D - A$ be the graph Laplacian and f the Fiedler vector. (a) State the Cheeger inequality: $\lambda_2/2 \le \phi(G) \le \sqrt{2\lambda_2}$ where ϕ is the conductance. (b) For a 2-regular cycle C_n, compute the unnormalized $\lambda_2 = 2(1 - \cos(2\pi/n))$ and the conductance $\phi(C_n) = 2/n$; verify both sides of Cheeger (careful: the bound uses the *normalised* Laplacian eigenvalue $\lambda_2^{\mathrm{norm}} = \lambda_2/d = 1 - \cos(2\pi/n)$ — pairing the unnormalized λ_2 with ϕ breaks the left inequality on small cycles: for $n = 8$, $\lambda_2/2 = 1 - \cos(\pi/4) \approx 0.293 > 2/8$). (c) Argue why the best *sweep cut* (sort the vertices by $x_v/\sqrt{\deg(v)}$, where x is the second eigenvector of the normalised Laplacian — on a regular graph, simply by f_v — and take the best of the $n-1$ prefix cuts) has conductance at most $\sqrt{2\lambda_2}$, hence cuts at most $\sqrt{2\lambda_2}\,|E| = O(\sqrt{\lambda_2}\cdot|E|)$ edges, and why the threshold at the median of f, which forces balance, carries no such guarantee. (d) Compute the Fiedler vector of C_8 by hand (the eigenvectors of the cycle Laplacian are discrete cosines) and identify which bisection it produces.

E12.7 **(Vertex-cut replication factor for a star graph).** Let $G = K_{1,m}$ (star with m leaves) be partitioned across k machines with a balanced edge partition (each machine gets m/k edges). (a) Compute the replication factor (number of machine copies of the hub vertex). (b) Compare to the edge-cut for a balanced vertex partition (all leaves on $k-1$ machines, hub on 1). (c) For $m = 10^6$ leaves (edges) and $k = 100$ machines, compute the total bytes of hub-state communication per computation step under both partitioning strategies.

Chapter 19

13 clustering ann

U13 — Clustering & similarity search at scale: RAM → PRAM → DPRAM

The ScaDaMaLe book, unit 13 (deep/optional, Part III). Our own exposition; original sources credited in §11.

Instantiating the three-lens invariant. The U0 box for clustering and similarity search:

> **RAM:** k-means has a clean objective (sum of squared distances to nearest centre) and Lloyd's algorithm (alternating assignment and recomputation) provably descends it monotonically — it converges but to a local optimum. k-means++ seeding gives an $O(\log k)$-approximation guarantee in expectation via a single random pass. Exact nearest-neighbour search costs $O(nd)$ per query — intractable at scale. **PRAM:** k-means‖ (Bahmani et al. 2012) runs $O(\log n)$ seeding rounds, each sampling $l = O(k)$ candidate centres in parallel; total work $O(nkd \log n)$, depth $O(\log n)$ rounds; the $O(k \log n)$ candidates have cost within a constant factor of the optimum, and a single cheap reclustering of them by an α-approximation gives an $O(\alpha)$-approximation ($O(\log k)$ with k-means++). **DPRAM/MPC:** each round of k-means‖ maps to one MPC round (local distance computations + one shuffle to aggregate sample weights); the per-round communication is $O(ld)$ points — asymptotically smaller than the full dataset. LSH and MinHash reduce nearest-neighbour search to hash lookups; the tables are one shuffle to build and one lookup per query. Modern graph-ANN systems (HNSW) dominate in practice but carry no worst-case theoretical guarantees; the theory/practice gap is honestly named.

Entry kit — what this chapter assumes; follow a link if you need the memory jog (map: Figure 1): F.M2 (norms and inner products) • F.C2 (the hash families LSH is built from)

1. Intuition — the geometry of scale

Two recurring questions drive large-scale data science: *How do I group similar points?* and *Given a query, what stored item is most similar?* At $n = 10^9$ and $d = 10^3$, neither brute-force answer is affordable.

Clustering partitions n points into k groups (the structure is latent). **Similarity search** (or nearest-neighbour search) answers queries against a pre-built index (the structure is in the index). Both problems are linked: the same distance geometry underlies both, and efficient data structures for one (locality-sensitive hashing, coresets) feed solutions for the other.

This unit is the geometry unit of the book. It weaves four threads:

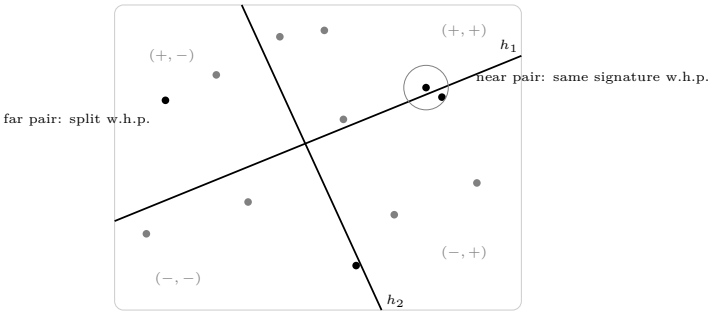


Figure 19.1: Locality-sensitive hashing for angular similarity: each random hyperplane contributes one sign bit, and the probability two points share a bit is exactly $1 - \theta/\pi$ — collision probability is a monotone function of closeness, the defining LSH property. Concatenating bits (AND) sharpens precision; repeating tables (OR) recovers recall; the F.S1/F.S2 toolkit prices both. Every vector index under a RAG system (U0L) is a production descendant of this picture.

1. **k-means** — the classic objective and its RAM algorithm (Lloyd), plus a theoretically-grounded seeding (k-means++) and a fully distributed seeding algorithm (k-means‖).
2. **LSH** — locality-sensitive hashing converts the similarity-search question into a hash-collision question, with a formal trade-off between accuracy and query time.
3. **Modern ANN in practice** — graph-based methods (HNSW) and inverted-file / product-quantization (IVF/PQ) dominate real deployments; we describe these honestly without overstating their theory.
4. **Coresets and submodularity** — two deep tools that recur in distributed optimisation: (k,ε)-coresets give provable compression; the submodular greedy algorithm gives a universal $(1 - 1/e)$ guarantee for a broad family of objectives including the facility-location objective tied to k-median (§7.4).

Figure 19.1 is the hashing idea that makes similarity searchable.

2. RAM — k-means, Lloyd's algorithm, and k-means++ seeding

2.1 The k-means objective

Given a set $X = \{x_1, \dots, x_n\} \subset \mathbb{R}^d$ and an integer k, choose a set $C = \{c_1, \dots, c_k\} \subset \mathbb{R}^d$ of *centres* to minimise:

$$\phi(X, C) \;=\; \sum_{x \in X} \min_{c \in C} \|x - c\|^2.$$

This is the *k-means objective* (sum of squared distances to the nearest centre). The optimal solution is NP-hard to compute in general; we seek algorithms with provable approximation guarantees, or algorithms that converge to a local optimum efficiently.

2.2 Lloyd's algorithm (Voronoi iteration)

Algorithm. Initialise $C^{(0)}$ arbitrarily. Repeat until convergence:

1. *Assignment.* For each x_i, assign x_i to its nearest centre: $z_i = \arg\min_{j \in [k]} \|x_i - c_j\|^2$.
2. *Update.* Replace each c_j by the mean of its assigned points: $c_j \leftarrow \frac{1}{|S_j|} \sum_{x \in S_j} x$, where $S_j = \{x_i : z_i = j\}$.

Cost (RAM). Each round costs $O(nkd)$: n points, k distance computations, d coordinates each. The algorithm terminates in at most k^n rounds (finite number of partitions), but in practice converges in $O(k)$ rounds empirically.

Theorem (monotone descent). Lloyd's algorithm is a coordinate descent on ϕ: each step does not increase ϕ.

Proof in three lines. The assignment step minimises ϕ over all assignments given fixed centres (each point goes to its nearest — any reassignment can only increase the sum). The update step minimises ϕ over all centres given fixed assignments (the mean minimises sum of squared deviations — a standard calculus fact: $\frac{\partial}{\partial c_j}\sum_{x\in S_j}\|x-c_j\|^2 = 0 \Rightarrow c_j = \text{mean}(S_j)$). So ϕ is non-increasing after each step; since the number of distinct partitions is finite, the algorithm terminates. □

Convergence caveat. Monotone descent to a *local* optimum, not the global optimum. The quality of the final solution depends critically on initialisation. Poor seeding can give solutions arbitrarily worse than optimal.

2.3 k-means++ seeding: the D^2-sampling rule

Algorithm (Arthur and Vassilvitskii 2007). Initialise the centre set C with one point:

1. Choose $c_1 \in X$ uniformly at random. Set $C = \{c_1\}$.
2. For $i = 2, \ldots, k$: choose $c_i \in X$ with probability proportional to $D(x)^2 = \min_{c\in C}\|x-c\|^2$ (the squared distance from x to the current nearest centre).
3. Run Lloyd's algorithm from C.

The rule in step 2 is D^2-sampling: points far from all current centres are chosen with higher probability. This biases the initial centres to be spread across the input.

Theorem (Arthur-Vassilvitskii 2007). k-means++ initialisation gives a solution satisfying:

$$\mathbb{E}[\phi(X,C)] \;\le\; O(\log k)\cdot\phi(X,C^*),$$

where C^* is the globally optimal set of k centres and the expectation is over the randomness of the D^2-sampling. That is, k-means++ seeding alone (before running Lloyd's) gives an $O(\log k)$-approximation in expectation.

We state this result and cite the original paper; the proof is a careful potential-function argument involving conditional expectations over the D^2-sampling process — beyond the scope of this unit but accessible in the original paper.

Sequential cost of k-means++ seeding. The k seeding rounds each require one pass over n points to compute D^2 values: total $O(nkd)$ for seeding alone, matching the cost of Lloyd's rounds. The sequential cost is acceptable; the distributed cost is not — k sequential passes over n partitioned points require k MPC rounds. This is the bottleneck that k-means‖ fixes.

3. Centerpiece 1 — k-means‖ (distributed seeding in $O(\log n)$ rounds)

3.1 The distributed seeding problem

k-means++ requires k sequential passes (one per centre) — one MPC round each. For $k = 1000$ and a dataset distributed across a cluster, that is 1000 shuffle stages. The k-means‖ algorithm

of Bahmani, Moseley, Vattani, Kumar, and Vassilvitskii (VLDB 2012) reduces this to $O(\log n)$ rounds by oversampling aggressively and reclustering.

3.2 Algorithm (k-means‖)

Parameters. Oversampling factor $l = O(k)$ (e.g., $l = 2k$ in practice). Number of rounds $r = O(\log n)$.

```
k-means-parallel(X, k, l, r):
  Choose c_1 from X uniformly at random.
  C := {c_1}
  psi := phi(X, C)   // initial cost: sum of squared distances to c_1
  For t = 1, ..., r:
    For each x in X (in parallel):
      sample x into C independently with probability  min(1, l * D(x)^2 / psi)
    update psi := phi(X, C)
  Compute weights w(c) := |{x in X : c = argmin_{c' in C} ||x - c'||}| for each c
    in C.
  Recluster (C, weights) into k centres (using k-means++ or Lloyd on the
    coreset).
  Return the k centres.
```

Each round: every point independently decides whether to join C based on its distance to the current centre set. After $r = O(\log n)$ rounds, $|C| = O(k \log n)$ — a small coreset of candidates. The final reclustering of this $O(k \log n)$-point weighted problem is cheap.

3.3 Per-round communication accounting

In the MPC model, each round of k-means‖ maps to one shuffle stage:

- **Map:** each machine computes $D(x)^2$ for each of its n/M points against the current centre set C (of size at most $l \cdot r = O(k \log n)$), and emits each sampled point.
- **Communication:** the current centre set C is broadcast to all machines ($O(|C| \cdot d)$ bytes per machine); sampled points are collected ($O(l \cdot d)$ bytes per round on average).
- **Reduce:** aggregate the sampled points into the new C.

Per-round bytes: $O((k \log n) \cdot d)$ to broadcast centres + $O(l \cdot d)$ sampled per round — a constant (in terms of n) number of bytes per round. The full data X stays partitioned; no global shuffle of all n points occurs in any round.

Total rounds: $R = O(\log n)$ seeding rounds + $O(1)$ rounds for the weighted reclustering.

3.4 Approximation theorem (stated)

Theorem (Bahmani et al. 2012). After $r = \Theta(\log n)$ rounds of k-means‖ with oversampling factor $l = \Omega(k)$, the set C of $O(k \log n)$ candidates satisfies:

$$\mathbb{E}[\phi(X, C)] \;\le\; O(1) \cdot \phi(X, C^*),$$

i.e., the $O(k \log n)$ coreset-like candidate set has expected cost within a constant factor of the optimal k-means objective (with more than k centres). The final reclustering of the weighted candidate set into k centres by an α-approximation algorithm then gives an $O(\alpha)$-approximation to the original problem (their Theorem 1) — $O(\log k)$ in expectation when the reclustering is k-means++.

The proof uses a potential-function argument showing that each round reduces the expected residual cost by a constant factor; after $O(\log n)$ rounds the residual is negligible. We state the result and cite the paper.

Spark ML anchor. `pyspark.ml.clustering.KMeans` uses k-means‖ as its default initialisation (the `initMode="k-means||"` parameter). The number of seeding steps is `initSteps` (default 2, matching the practical finding that 2–5 rounds suffice empirically even though the theory requires $O(\log n)$). The implementation follows Algorithm 2 of the Bahmani et al. paper (their Algorithm 1 is k-means++).

4. Similarity search: exact cost, then LSH

4.1 Exact nearest-neighbour at scale

Given a query point $q \in \mathbb{R}^d$ and a database X of n points, the exact nearest-neighbour (NN) problem asks for $\arg\min_{x \in X} \|q - x\|$. Brute-force cost: $O(nd)$ per query. For $n = 10^9$, $d = 256$, and 10^6 queries per second, this is $10^9 \times 256 \times 10^6 \approx 2.6 \times 10^{17}$ operations per second — clearly intractable.

Exact NN structure (kd-tree, ball-tree). These give $O(d \log n)$ query time but degrade to $O(nd)$ in high dimensions (the "curse of dimensionality": all points become nearly equidistant). In practice, exact NN is not feasible for $d \geq 20$ and large n.

The approximate NN (ANN) trade-off. Accept a $(1 + \varepsilon)$-approximation (return a point within factor $1 + \varepsilon$ of the true NN) or, in the (c, r)-NN formulation: given a query with a true NN at distance $\leq r$, return any point at distance $\leq cr$. This relaxation enables sublinear query times via hashing or indexing.

4.2 Locality-sensitive hashing (LSH) — formal definition

Definition (locality-sensitive family). A family $\mathcal{H}$ of hash functions $h : \mathcal{X} \to U$ is (r, cr, p_1, p_2)-*sensitive* for distance dist (with approximation ratio $c > 1$) if for any two points $x, y \in \mathcal{X}$:

$$\text{dist}(x, y) \leq r \;\Rightarrow\; P_{h \in \mathcal{H}}[h(x) = h(y)] \geq p_1,$$
$$\text{dist}(x, y) \geq cr \;\Rightarrow\; P_{h \in \mathcal{H}}[h(x) = h(y)] \leq p_2,$$

with $p_1 > p_2$. Close points collide with higher probability; far points with lower probability. The *quality* of the family is captured by $\rho = \frac{\log p_1}{\log p_2}$ (smaller is better).

The amplification construction (AND-OR). A single LSH function has unacceptably high false-positive and false-negative rates. Amplify by combining two operations:

- **AND (concatenation):** form a *band* $g = (h_1, \ldots, h_k)$ of k independent functions from $\mathcal{H}$. Two points collide in the band iff they collide in all k functions. The collision probabilities become p_1^k (close pairs) and p_2^k (far pairs) — near pairs become less likely to collide, but far pairs are suppressed faster if $p_2 < p_1$.
- **OR (L tables):** build L independent band functions $g_1, \ldots, g_L$. Answer a query by looking up the query in all L tables and returning any collision. Two points collide in at least one table with probability $1 - (1 - p_1^k)^L$ (close) vs $1 - (1 - p_2^k)^L$ (far).

Choose k and L to balance these probabilities to desired false-positive/negative rates.

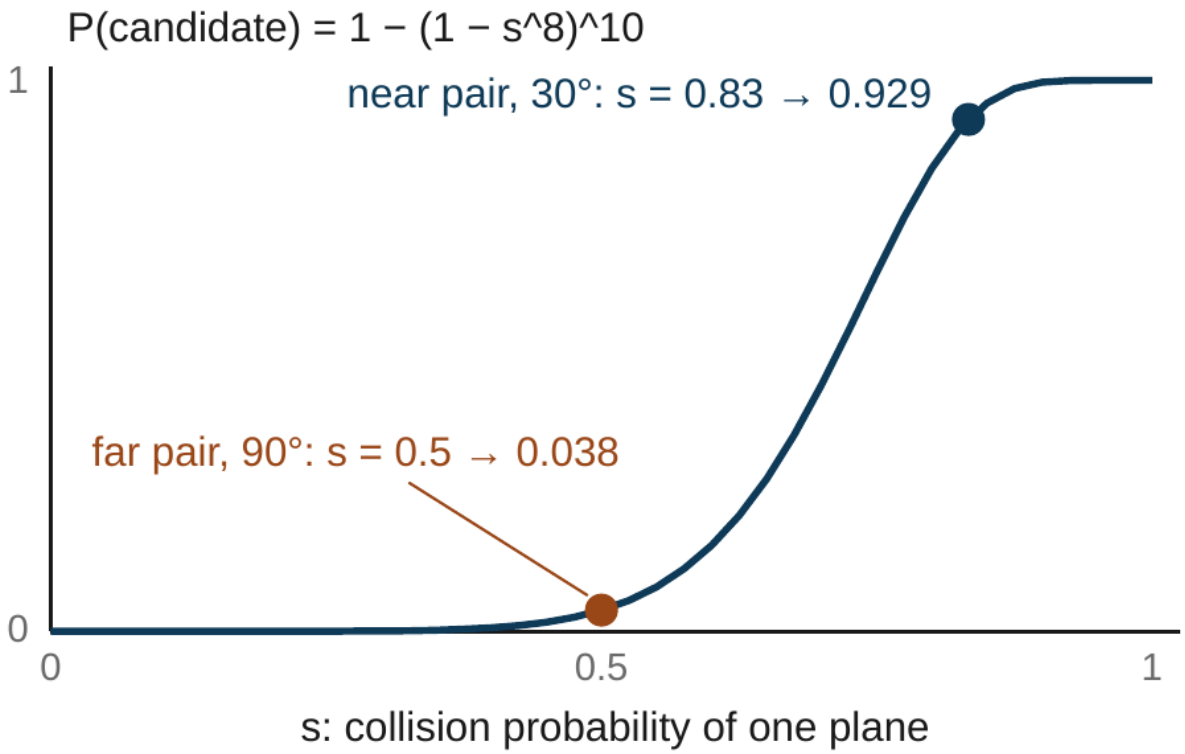


Figure 19.2: A near pair and a far pair, hashed with 8 planes in each of 10 tables.

Figure 19.2 puts numbers on this choice: $k = 8$ planes per table and $L = 10$ tables, for one near pair and one far pair.

4.3 Query time theorem (stated with argument sketch)

Theorem (Indyk-Motwani 1998). For a (r, cr, p_1, p_2)-sensitive family with $\rho = \log p_1 / \log p_2 < 1$, setting $k = \log_{1/p_2} n$ and $L = n^\rho$ gives:

- **False-positive rate** (far-pair collision in some table): at most constant.
- **False-negative rate** (close-pair misses all tables): at most constant.
- **Query time:** $O(n^\rho \cdot d \cdot k) = O(n^\rho)$ distance evaluations (ignoring d, k as logarithmic factors), sub-linear in n for $\rho < 1$.
- **Space:** $O(n^{1+\rho})$ for the hash tables.

Argument sketch. A close pair (distance $\le r$) collides in a single band with probability $p_1^k = p_1^{\log_{1/p_2} n} = n^{\log_{1/p_2} p_1} = n^{-\rho}$ (using $\log_{1/p_2} p_1 = -\frac{\log p_1}{\log p_2} = -\rho$. More carefully: p_1^k with $k = \log n / \log(1/p_2)$ gives $p_1^k = e^{k \log p_1} = n^{\log p_1 / \log(1/p_2)} = n^{-\log p_1 / \log p_2} = n^{-\rho}$). Setting $L = n^\rho$ makes the close-pair collision probability over all L tables approach $1 - (1 - n^{-\rho})^{n^\rho} \approx 1 - 1/e$ (constant probability). For a far pair: band collision probability is $p_2^k \le 1/n$, so the expected number of tables in which one far point collides with the query is $L \cdot p_2^k \le n^\rho / n = n^{\rho - 1} \to 0$. Summed over the (at most n) far points, the expected number of false positives per query is at most $n \cdot n^{\rho-1} = n^\rho = L$ — not $o(1)$ — so by Markov's inequality more than $3L$ appear with probability at most $1/3$, and the query stops after inspecting $3L$ candidates. The expected work per query is $L = n^\rho$ table lookups, each requiring $O(kd) = O(d \log n)$ to compute the band hash, plus at most $3L$ candidate distance checks of $O(d)$ each, giving $O(n^\rho d \log n)$ total. □

The key insight: $\rho < 1$ is what makes LSH useful — it depends on the gap between p_1 and p_2, which in turn depends on the specific hash family and the distance. For Euclidean (ℓ_2) distance,

p-stable projections (Datar et al. 2004) achieve ρ of order $1/c$ (optimising the bucket width numerically gives $\rho \approx 0.45$ at $c = 2$, against $1/c^2 = 0.25$); the Andoni-Indyk 2006 construction approaches $\rho = 1/c^2$, the best possible for hash families chosen independently of the data. Data-dependent hashing (Andoni-Razenshteyn 2015, §5.3) goes below $1/c^2$, so $1/c^2$ is not an information-theoretic floor for ℓ_2. For Jaccard, MinHash gives $\rho = \log(1/p_1)/\log(1/p_2)$ with explicit formulas depending on the approximation ratio c.

4.4 MinHash for Jaccard similarity — collision probability proved

Setting. Let $A, B \subseteq [u]$ be two sets (e.g., documents represented as sets of shingles). The **Jaccard similarity** is $J(A, B) = |A \cap B|/|A \cup B|$.

MinHash (Broder 1997). Let $\pi : [u] \to [u]$ be a uniformly random permutation. Define $h_\pi(A) = \min_{e \in A} \pi(e)$ — the minimum hash value attained by any element of A.

Theorem. $P_\pi[h_\pi(A) = h_\pi(B)] = J(A, B)$.

Proof in four lines. (1) $h_\pi(A) = h_\pi(B)$ iff the element with the smallest π-value in $A \cup B$ belongs to $A \cap B$ (since $h_\pi(A) = \min_A \pi = h_\pi(B) = \min_B \pi$ iff $\arg\min_{A \cup B} \pi \in A \cap B$). (2) By symmetry of a uniform random permutation, each element of $A \cup B$ is equally likely to have the minimum π-value. (3) The number of elements in $A \cup B$ that are also in $A \cap B$ is $|A \cap B|$. (4) Therefore $P[h_\pi(A) = h_\pi(B)] = |A \cap B|/|A \cup B| = J(A, B)$. □

This is the foundational result: MinHash turns Jaccard similarity into a collision probability exactly.

Banding for near-duplicate detection. Use b bands of r hash functions each ($b \times r = t$ total hashes). Two documents with Jaccard similarity s collide in at least one band with probability $1 - (1 - s^r)^b$. The function $s \mapsto 1 - (1 - s^r)^b$ is an S-curve: pairs with s above a threshold collapse to probability ≈ 1; below it, to ≈ 0. The threshold is approximately $(1/b)^{1/r}$. This gives a practical near-duplicate detection pipeline: compute t MinHashes, band them, and report pairs sharing a band as candidate near-duplicates.

Cost. Building MinHash signatures: $O(nd \cdot t)$ for a dataset of n documents with average size d. One shuffle stage to group documents by their band hashes and find collisions.

4.5 Random-hyperplane LSH for cosine similarity — collision probability proved

Setting. Let $x, y \in \mathbb{R}^d$, with $\theta = \angle(x, y)$ (the angle between them). The **cosine similarity** is $\cos\theta$.

Random-hyperplane hash (Charikar 2002 / SimHash). Choose a random unit vector $r \in \mathbb{R}^d$ (each component i.i.d. standard normal, then normalised). Define:

$$h_r(x) = \text{sign}(r \cdot x) \in \{+1, -1\}.$$

Theorem. $P_r[h_r(x) \neq h_r(y)] = \theta/\pi$, and equivalently, $P_r[h_r(x) = h_r(y)] = 1 - \theta/\pi$.

Proof via the plane-cutting argument. (1) $h_r(x) \neq h_r(y)$ iff r lies in the open half-space where $r \cdot x > 0, r \cdot y < 0$ or vice versa — equivalently, iff the hyperplane $\{v : r \cdot v = 0\}$ (which is perpendicular to r) *separates* x and y (i.e., x and y are on opposite sides). (2) r is uniform over the sphere. The hyperplane through the origin perpendicular to r is a uniformly random hyperplane through the origin. (3) A uniformly random hyperplane through the origin separates x and y with probability equal to the fraction of the great circle (in the plane spanned by

x and y) that lies in the "separating arc" — which is θ/π (the arc of length proportional to the angle between x and y, out of total great-circle arc π per hemisphere). (4) Therefore $P[h_r(x) \neq h_r(y)] = \theta/\pi$. $\square$

Intuition: the closer x and y are (small θ), the smaller the separating arc, and the less likely a random hyperplane cuts between them. This is the basis of SimHash, used in large-scale near-duplicate detection (Google web-crawl deduplication, web-scale retrieval systems).

LSH family quality for cosine. $p_1 = 1-\theta_1/\pi$ (for near-pairs with angle $\leq \theta_1$) and $p_2 = 1-\theta_2/\pi$ (for far-pairs with angle $\geq \theta_2$), giving $\rho = \frac{\log(1-\theta_1/\pi)}{\log(1-\theta_2/\pi)}$. For small angles this is close to $\theta_1/\theta_2 < 1$.

5. Modern ANN: HNSW, IVF/PQ, and the theory/practice gap

5.1 Graph-based ANN: HNSW

The **Hierarchical Navigable Small World (HNSW)** algorithm (Malkov and Yashunin 2018) is the current state-of-the-art ANN index in practice, dominating benchmarks (ann-benchmarks.com) across recall and query-per-second metrics for many dataset types.

The idea (sketch, no false theorem claims). HNSW builds a multi-layer proximity graph: - **Base layer (layer 0):** all n points; each point connected to its (approximate) M nearest neighbours in the graph built so far. - **Upper layers:** a geometrically thinned subset of points (each point is inserted into layer ℓ with probability $e^{-\ell/m_L}$ for a scale parameter m_L); upper layers act as a "highway" for coarse navigation.

Query (greedy search). Start from an entry point at the top layer; greedily follow edges to the neighbour closest to the query; descend to the next layer; repeat until layer 0, then collect the k closest points found.

Index construction. Each new point is inserted by running greedy search to find its neighbours at each layer, then adding bidirectional links.

Empirical performance. HNSW achieves recall@1 of $\geq 95\%$ with query times orders of magnitude below brute-force on standard benchmarks (SIFT1M, GIST1M, GloVe). Index construction is $O(n \log n)$ empirically.

What theory says (honest). There is no worst-case guarantee for HNSW: the greedy search can be trapped in local minima on adversarial inputs. The theoretical foundation for navigable small-world graphs (Kleinberg 2000 on lattice graphs) does not transfer directly to arbitrary metric spaces or high-dimensional ℓ_2. HNSW's success is empirical; the correct framing is "best-in-practice graph-based ANN with no worst-case theoretical guarantees." Honest engineering practice: use HNSW when empirical recall and latency on your dataset meet requirements; use LSH when you need probabilistic guarantees.

5.2 IVF / product quantization (one paragraph)

Inverted file index (IVF): partition the dataset into k_{coarse} Voronoi cells (using k-means); at query time, search only the n_{probe} nearest cells. This reduces the search space by a factor of $k_{\text{coarse}}/n_{\text{probe}}$. **Product quantization (PQ)** (Jégou et al. 2011) compresses each d-dimensional vector by splitting it into m sub-vectors of dimension d/m and quantizing each sub-vector independently with a codebook of size 256 (one byte). A vector then takes m bytes instead of $4d$ (float32), a factor of $4d/m$ — for $d = 128$ and $m = 8$, 512 bytes become 8, a 64× saving — and distances are approximated using lookup tables. **IVF+PQ** (as in FAISS, Facebook AI) is the standard choice for billion-scale ANN search where memory is a constraint; it trades recall for throughput and memory.

5.3 When LSH-theory vs graph-ANN-practice

	LSH	Graph-ANN (HNSW)
Worst-case guarantee	Yes (n^ρ query time, $n^{1+\rho}$ space)	No
Typical recall@10	Moderate (tunable via k, L)	High (state-of-the-art)
Index update (streaming)	Easy (add to hash tables)	Insert incremental (§5.1); delete hard (graph repair costly)
Theory/practice gap	Narrow (the bound holds)	Wide (empirical only)
Recommended when	Provable bounds required; streaming; high-d Hamming/Jaccard	Recall-QPS matters most on static index; $d \leq 1000$

The gap is genuine and worth naming: the theoretical ANN literature (Andoni-Razenshteyn 2015 optimal LSH) proves tight bounds, but graph-ANN systems win all practical benchmarks. This is an active research area.

6. Coresets — provable compression for clustering

6.1 Definition

A (k, ε)**-coreset** for the k-means problem on X is a weighted point set $S \subseteq X$ (with weights $w : S \to \mathbb{R}_{>0}$) such that for *every* set C of k centres:

$$(1-\varepsilon)\,\phi(X, C) \;\leq\; \phi_w(S, C) \;\leq\; (1+\varepsilon)\,\phi(X, C),$$

where $\phi_w(S, C) = \sum_{s \in S} w(s)\, \min_{c \in C} \|s - c\|^2$ is the weighted k-means objective on S. A coreset approximates the objective uniformly over all candidate centre sets.

Coreset size (stated). The following result gives near-optimal coreset sizes:

Theorem (Feldman and Langberg 2011). For any $X \subset \mathbb{R}^d$ with $|X| = n$ and any $\varepsilon, \delta > 0$, there exists a (k, ε)-coreset for k-means of size

$$|S| \;=\; O\!\left(\frac{k \cdot d \cdot \log(1/\delta)}{\varepsilon^2}\right),$$

obtainable in $O(nkd)$ time. The size is independent of n, making coresets the key to streaming and distributed clustering.

6.2 Merge-and-reduce streaming composition (ties to U4)

The power of coresets in distributed settings comes from a composition lemma:

Lemma (merge-and-reduce). If S_1 is a (k, ε)-coreset for X_1 and S_2 is a (k, ε)-coreset for X_2 (disjoint), then any (k, ε)-coreset of $S_1 \cup S_2$ (the union of the coresets) is a $(k, O(\varepsilon))$-coreset for $X_1 \cup X_2$.

In words: a coreset of coresets is a coreset. This is the same **mergeability** property that made streaming sketches (Count-Min, HLL) composable in U4 — both are manifestations of the same structural principle: if a compressed summary is closed under a composition operation, it can be aggregated in a tree.

Streaming algorithm. Process X in blocks of size s; compute a local coreset for each block; merge and reduce using the tree structure. At each merge node, take the union of two coresets and re-compute a coreset of the union. However, the errors compound: over the $h = \lceil \log_2(n/s) \rceil$ levels of the tree, level-wise (k, ε')-coresets give only a $((1+\varepsilon')^h - 1)$-coreset at the root (E13.7).

So we run each level at $\varepsilon' = \varepsilon/(2h)$ (for $\varepsilon \le 1$); the root then holds a (k, ε)-coreset of the full dataset of size $O(kd \log(1/\delta) \log^2(n/s)/\varepsilon^2)$ — polylogarithmic in n, not independent of it.

DPRAM/MPC. Each machine computes a local coreset (one pass, $O(nkd/M)$ per machine); coresets are sent to a tree of aggregators ($O(\log M)$ rounds); the root recluters on the merged coreset. Total communication: $O(M \cdot |S|) = O(M \cdot kd \log^2 M/\varepsilon^2)$ (each of the $\log M$ levels run at $\varepsilon/\log M$, as above), independent of n.

The connection. k-means‖ (§3) can be seen as a *randomised coreset construction*: the $O(k \log n)$ sampled candidates (with weights proportional to the number of points they represent) form a coreset-like summary. The theoretical analysis in §3.4 formalises this view. Both algorithms are instances of one principle: reduce n to a small summary, then solve the summary.

7. Submodularity and the greedy $(1 - 1/e)$ guarantee

7.1 Submodularity defined

Definition. A set function $f : 2^V \to \mathbb{R}$ is **submodular** if for all $A \subseteq B \subseteq V$ and all $u \notin B$:

$$f(A \cup \{u\}) - f(A) \;\ge\; f(B \cup \{u\}) - f(B).$$

The marginal gain of adding u is non-increasing as the set grows — a formal statement of "diminishing returns." Examples: coverage functions, entropy, graph cut, the facility-location objective (see below).

Non-negative monotone submodular maximisation. Given f submodular, non-negative, and monotone ($f(A) \le f(B)$ for $A \subseteq B$), find $A \subseteq V$ with $|A| \le k$ maximising $f(A)$.

7.2 The greedy $(1 - 1/e)$ theorem

Algorithm (greedy). Start with $A = \emptyset$. For $i = 1, \dots, k$: add to A the element $u \notin A$ with the largest marginal gain $f(A \cup \{u\}) - f(A)$.

Theorem (Nemhauser, Wolsey, Fisher 1978). The greedy algorithm achieves:

$$f(A_{\text{greedy}}) \;\ge\; \left(1 - \frac{1}{e}\right) f(A^*),$$

where A^* is the optimal set of size k. The factor $1 - 1/e \approx 0.632$ is tight (achieved by coverage functions).

Proof sketch (marginal-gains telescoping). Let $A^* = \{u_1^*, \dots, u_k^*\}$ be optimal. At step i, let A_i be the greedy set. The key inequality: since A^* is a k-element set and f is submodular and monotone:

$$f(A^*) \le f(A_i \cup A^*) \le f(A_i) + \sum_{j=1}^{k} \big[f(A_i \cup \{u_j^*\}) - f(A_i) \big].$$

The greedy choice at step $i+1$ has marginal gain $\ge$ the average marginal gain of any k elements (including those of A^*):

$$f(A_{i+1}) - f(A_i) \ge \frac{1}{k} \big[f(A^*) - f(A_i) \big].$$

Let $\delta_i = f(A^*) - f(A_i)$ (the gap). Then $\delta_{i+1} \le \delta_i(1 - 1/k)$, so by induction $\delta_k \le \delta_0(1 - 1/k)^k \le \delta_0/e$. Since $\delta_0 = f(A^*) - f(\emptyset) \le f(A^*)$, we get $f(A^*) - f(A_k) \le f(A^*)/e$, i.e., $f(A_k) \ge (1 - 1/e)f(A^*)$. $\square$

7.3 Distributed GreeDi (two-round partition-greedy)

Algorithm (Mirzasoleiman et al. 2013). Partition V into m equal parts $V_1, \dots, V_m$.

1. **Round 1 (local):** on each part V_i, run greedy to select k elements locally, obtaining G_i.
2. **Round 2 (global):** collect $G = G_1 \cup \cdots \cup G_m$ (of size mk) at one machine; run greedy on G to select the final k elements.

Approximation factor (stated). GreeDi achieves:

$$\mathbb{E}[f(A_{\text{GreeDi}})] \;\ge\; \frac{1}{\min(\sqrt{k}, m)} \cdot (1 - 1/e) \cdot f(A^*),$$

Read the factor honestly: the guarantee *weakens* as m grows — for $m \le \sqrt{k}$ the divisor is m, and beyond that it plateaus at $\sqrt{k}$; more partitions never *improve* the worst-case bound. For moderate m (e.g., $m = 10$ partitions) and many practical submodular functions, GreeDi nonetheless achieves near-$(1 - 1/e)$ performance — an *empirical* observation, not the theorem.

Rounds: $R = O(1)$ (2 rounds: one map-reduce for local greedy; one global greedy at root).

Cost. Local: $O(n/m \cdot k)$ evaluations per machine. Communication: $O(mk \cdot d)$ to collect local solutions. Total: $O(nk/m \cdot m) = O(nk)$ local work + $O(mk)$ communication.

7.4 Facility location and the tie to k-median

The **facility-location objective** is:

$$f_{\text{FL}}(C) = \sum_{x \in X} \max_{c \in C} \text{sim}(x, c),$$

where sim is a non-negative similarity function (and $f_{\text{FL}}(\emptyset) = 0$). This is non-negative, monotone, and submodular, as a sum of submodular functions: adding u to C raises the x-term by $\big(\text{sim}(x, u) - \max_{c \in C} \text{sim}(x, c)\big)^+$, which can only shrink as C grows. Maximising f_{FL} subject to $|C| \le k$ is tied to k-median (which minimises $\sum_x \min_c \text{dist}(x, c)$) by the distance-to-similarity conversion $\text{sim}(x, c) = D - \text{dist}(x, c)$, with D at least the largest distance: then $f_{\text{FL}}(C) = nD - \text{cost}(C)$, so the two problems have the same optimal C, and the greedy $(1 - 1/e)$ bound applies to f_{FL}. However, the guarantee does not transfer multiplicatively: $f_{\text{FL}}(C) \ge (1 - 1/e) \max f_{\text{FL}}$ gives only $\text{cost}(C) \le (1 - 1/e)\,\text{OPT} + nD/e$, an additive nD/e and no constant-factor k-median bound. The GreeDi algorithm above thus gives a distributed approximation algorithm for facility location; for k-median it is a heuristic carrying only that additive guarantee.

8. The invariant (U13 instantiation)

RAM: k-means objective; Lloyd's monotone descent converges in finitely many steps (proved in 3 lines); k-means++ D^2-seeding gives $O(\log k)$-approximation (AV 2007, stated). LSH: a (r, cr, p_1, p_2)-sensitive family (approximation ratio $c > 1$) + AND-OR amplification reduces approximate NN to hash lookup; n^ρ query time where $\rho = \log p_1 / \log p_2$ (Indyk-Motwani, stated). MinHash: collision probability =

Jaccard (proved in 4 lines). Random hyperplane: collision probability $= 1 - \theta/\pi$ (proved via plane-cutting). Exact NN is $O(nd)$ and intractable at scale.

PRAM / DPRAM centerpiece: k-means‖ (Bahmani et al. 2012): $O(\log n)$ rounds, each sampling $l = O(k)$ candidates in parallel; $O(\alpha)$-approximation after an α-approximate reclustering (stated); per-round communication $O(kd \log n)$, not $O(nd)$; Spark ML implements this directly. Coresets: a (k, ε) -coreset of size independent of n (Feldman-Langberg 2011, stated); merge-and-reduce streaming composition = "coresets of coresets are coresets" (same mergeability principle as U4 sketches).

Modern practice vs theory: HNSW (Malkov-Yashunin 2018) dominates ANN benchmarks — empirical state-of-the-art with no worst-case theoretical guarantees; use when recall-QPS matters on static index. IVF+PQ trades recall for memory/throughput at billion scale. Use LSH when provable bounds or streaming updates are required.

Submodular deep tail: the greedy $(1 - 1/e)$ bound (Nemhauser-Wolsey-Fisher 1978; proved via marginal-gains telescoping) applies to any non-negative monotone submodular objective; distributed GreeDi (Mirzasoleiman et al. 2013) guarantees $(1 - 1/e)/\min(\sqrt{k}, m)$ in 2 rounds (near-$(1 - 1/e)$ only empirically); facility-location ties k-median to the submodular framework.

9. Lab — project track

Project-track lab (no weekly receipt): run these experiments free and locally in the course lab image (`labs/image/`), system-agnostic (python-graphblas / NumPy / MinHash-from-scratch), as deep-dive and group-project material.

Experiment 1 — k-means‖ rounds vs quality.

Implement k-means‖ using NumPy (or python-graphblas sparse matrix operations for the distance computations) on a synthetic Gaussian mixture dataset ($n = 10^5$, $d = 10$, $k = 20$ true clusters). Vary the number of seeding rounds $r \in \{1, 2, 5, \log_2 n\}$ and the oversampling factor $l \in \{k/2, k, 2k\}$. For each configuration, report the final k-means objective after Lloyd's convergence. Compare to k-means++ (as a sequential baseline). Plot objective vs rounds at fixed l; plot objective vs l at fixed r. How many rounds suffice in practice?

Experiment 2 — MinHash / LSH near-duplicate detection: precision-recall vs (k, L).

Use a text dataset (e.g., 10,000 Wikipedia article snippets, pre-tokenised into 3-shingles). Implement MinHash from scratch: generate $t = 100$ hash functions (random affine maps modulo a large prime); compute signatures; apply banding with varying (b, r) pairs (fixing $b \cdot r = 100$). Report precision and recall of near-duplicate detection (ground truth: Jaccard ≥ 0.5) vs the banding threshold $(1/b)^{1/r}$. Overlay the theoretical S-curve $1 - (1 - s^r)^b$. Does the empirical precision-recall track the theory?

Experiment 3 — HNSW vs brute-force: recall-QPS curve.

Note on system choice: HNSW is available via the `hnswlib` Python library (Apache-2 licence; single-machine, no GPU required) for laptop-runnable experiments; for a FAISS-free alternative, `nndescent` implements the NN-descent graph build. The lab uses `hnswlib`.

On a synthetic or open dataset ($n = 10^5$, $d = 128$; e.g., random Gaussian or the GloVe-25 open dataset), build an HNSW index with varying M (connection parameter) and ef_construction. Query 10^3 random query points at varying `ef` (search parameter); report recall@10 (fraction of true 10-NN returned) vs queries-per-second. Plot the recall-QPS Pareto frontier. Compare to

brute-force at the same recall. What does the recall-QPS curve tell us about the theory/practice gap?

Deliverable: the notebook + a half-page cost-accounting in the three-lens vocabulary: for each experiment, which cost (ops, rounds, communication, memory) is the bottleneck and how does the algorithm address it?

10. Lean certificate target

Target (U13, new): the MinHash collision probability theorem of §4.4 — that $P[h_\pi(A) = h_\pi(B)] = J(A, B)$ for a uniform random permutation π — is a statement in finite combinatorics, amenable to a Lean proof.

```
-- SCHEMATIC target statement:
-- Let U be a finite type with n elements; A B : Finset U.
-- h_pi(A) = argmin over A of a uniformly random permutation pi : Equiv.Perm U
-- (a finite U has no bijection onto ℕ; LinearOrder U orders the values pi e).
-- The collision event = the minimum-pi element of A ∪ B lies in A ∩ B.

theorem minhash_jaccard [Fintype U] [LinearOrder U] (A B : Finset U) :
    P_{π : Equiv.Perm U} [π.argmin A = π.argmin B] = |A ∩ B| / |A ∪ B| := by
  -- Key step: h_π(A) = h_π(B) ↔ argmin_{A∪B} π ∈ A ∩ B
  -- Uniform distribution over permutations → each element of A∪B equally
    likely to be argmin
  -- P[argmin ∈ A∩B] = |A∩B| / |A∪B|
  sorry  -- cert target
```

Exercise U13.L1 (Lean, light). In Lean's `Mathlib.Data.Finset`, state the combinatorial core: for a finite set $S = A \cup B$ with each element equally likely to be minimum, the probability that the minimum lies in $A \cap B$ is $|A \cap B|/|A \cup B|$. Write this as a `Finset.card`-to-`Rat` ratio and verify it type-checks.

Exercise U13.L2 (deep). Extend to the full theorem: formalise the uniform permutation model (uniform measure on $\mathrm{Sym}(U)$), define h_π, state the collision event, and prove the equality. The combinatorial core from L1 is the main step; the measure-theory wrapper uses Mathlib's `ProbabilityTheory.uniformOn` on `Set.univ : Set (Equiv.Perm U)`.

LLM lens — where this unit lands on the running workload (U0L):
RAG's vector search IS approximate nearest neighbours: embed the corpus (U7), then answer queries with ANN indexes whose recall–latency trade this unit prices. LSH's "close things collide" is the same leverage the retrieval layer of every production LLM system runs on.

11. Sources (credited; archived under `scadamale-book-sources`)

Lloyd (1957/1982). S. P. Lloyd, *Least squares quantization in PCM*, IEEE Transactions on Information Theory 28(2):129–137, 1982. (Circulated as a Bell Labs technical report in 1957.) — The assignment-update iteration (Lloyd's algorithm); the monotone descent property.

Arthur and Vassilvitskii (2007). D. Arthur and S. Vassilvitskii, *k-means++: The advantages of careful seeding*, in Proc. SODA, pp. 1027–1035, 2007. — The D^2-sampling seeding rule and the $O(\log k)$-approximation theorem.

Bahmani, Moseley, Vattani, Kumar, Vassilvitskii (2012). B. Bahmani, B. Moseley, A. Vattani, R. Kumar, S. Vassilvitskii, *Scalable k-means++*, VLDB 2012, 5(7):622–633. —

The k-means‖ algorithm (their Algorithm 2), the oversampling-round analysis, and the $O(\alpha)$-approximation theorem (their Theorem 1).

Indyk and Motwani (1998). P. Indyk and R. Motwani, *Approximate nearest neighbors: towards removing the curse of dimensionality*, in Proc. STOC, pp. 604–613, 1998. — The foundational LSH paper; the (r, cr, p_1, p_2)-sensitive family definition and the n^ρ query-time result.

Gionis, Indyk, Motwani (1999). A. Gionis, P. Indyk, R. Motwani, *Similarity search in high dimensions via hashing*, in Proc. VLDB, pp. 518–529, 1999. — The AND-OR amplification construction and the practical banding scheme.

Broder (1997). A. Z. Broder, *On the resemblance and containment of documents*, in Proc. Compression and Complexity of Sequences, pp. 21–29, 1997. — MinHash and the collision-probability = Jaccard theorem.

Charikar (2002). M. S. Charikar, *Similarity estimation techniques from rounding algorithms*, in Proc. STOC, pp. 380–388, 2002. — Random-hyperplane LSH for cosine similarity (SimHash); the $1 - \theta/\pi$ collision probability.

Malkov and Yashunin (2018). Y. A. Malkov and D. A. Yashunin, *Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs*, IEEE TPAMI, 42(4):824–836, 2020 (preprint 2018). — The HNSW algorithm and index structure; we rely on this paper for the empirical performance claims and the absence of worst-case guarantees.

Feldman and Langberg (2011). D. Feldman and M. Langberg, *A unified framework for approximating and clustering data*, in Proc. STOC, pp. 569–578, 2011. — The (k, ε)-coreset size bound; the merge-and-reduce composition.

Nemhauser, Wolsey, Fisher (1978). G. L. Nemhauser, L. A. Wolsey, M. L. Fisher, *An analysis of approximations for maximizing submodular set functions — I*, Mathematical Programming 14:265–294, 1978. — The greedy $(1 - 1/e)$ theorem; the marginal-gains telescoping argument (our proof sketch follows their structure).

Mirzasoleiman, Karbasi, Sarkar, Krause (2013). B. Mirzasoleiman, A. Karbasi, R. Sarkar, A. Krause, *Distributed submodular maximization: identifying representative elements in massive data*, NIPS 2013. — The two-round partition-greedy (GreeDi) algorithm and its approximation factor $\mathbb{E}[f(A_{\text{GreeDi}})] \geq (1 - 1/e)/\min(\sqrt{k}, m) \cdot f(A^*)$.

Wikipedia. *k-means clustering*, *k-means++ seeding*, *Locality-sensitive hashing*, *MinHash*, *SimHash*, *Hierarchical Navigable Small World*, *Submodular set function*, *Facility location problem*, *Coreset*. — Background and cross-references.

12. Exercises

Light (examined core):

E13.1 Lloyd's algorithm on a 1D example: $X = \{1, 2, 8, 9\}$, $k = 2$, initial centres $C^{(0)} = \{1, 9\}$. Run two rounds by hand. Verify that ϕ is non-increasing at each step. What are the final centres? Is this globally optimal?

E13.2 k-means++ D^2-sampling: $X = \{0, 1, 5, 6, 10\}$ on the real line, $k = 2$. Given $c_1 = 0$, compute $D(x)^2$ for each point and the sampling probabilities for the second centre. Which centre choices lead to a good initialisation?

E13.3 MinHash: let $A = \{1, 2, 3\}$, $B = \{2, 3, 4\}$ over universe $[5]$. Compute $J(A, B)$ exactly. For the permutation $\pi = (3, 1, 4, 2, 5)$ (i.e., $\pi(1) = 3, \pi(2) = 1, \pi(3) = 4, \pi(4) = 2, \pi(5) = 5$),

compute $h_\pi(A)$ and $h_\pi(B)$. Do they collide? Is the collision consistent with the theorem?

E13.4 Random-hyperplane LSH: two unit vectors in $\mathbb{R}^2$ with angle $\theta = \pi/4$. What is the probability that a random hyperplane separates them? What about $\theta = \pi/2$? Verify the formula $P = \theta/\pi$.

Deep (Module-3 / optional):

E13.5 *k-means‖ rounds needed.* Show that the number of rounds $r = O(\log \psi_0/\mathrm{OPT})$ suffices, where $\psi_0 = \phi(X, \{c_1\})$ is the initial cost (one centre) and OPT is the optimum. How does this connect to the $O(\log n)$ bound?

E13.6 *Submodular greedy tightness.* Give an example of a monotone submodular function f (e.g., a set-coverage function) for which the greedy $(1 - 1/e)$ bound is achieved with equality in the limit. Construct the function explicitly and trace through the marginal-gains argument.

E13.7 *Coreset composition.* Prove the merge-and-reduce lemma: if S_i is a (k, ε) -coreset for X_i $(i = 1, 2)$ and S is a (k, ε)-coreset of $S_1 \cup S_2$, then S is a $(k, 2\varepsilon + \varepsilon^2)$-coreset for $X_1 \cup X_2$. (*Hint:* chain the approximation inequalities: $\phi(X, C) \approx_\varepsilon \phi_w(S_1 \cup S_2, C) \approx_\varepsilon \phi_{w'}(S, C)$; account for the multiplicative error accumulation.)

Chapter 20

14 density trees

U14 — Nonparametric density estimation on sparse binary trees

The ScaDaMaLe book, unit 14 (Part IV — the ScaDaMaLe superset; deep / optional material). Our own exposition; original sources credited in §8.

The context for Part IV. Parts 0–III of this book cover a near-superset of the standard distributed-algorithms canon. Part IV is where ScaDaMaLe becomes distinctly itself: the algorithms here arise from our own research line in nonparametric density estimation on recursive binary-tree partitions. U14 introduces the data structure and the distributed algorithm. U15 follows with the statistical guarantee (the minimum-distance estimator, its oracle inequality and its rate). Readers who have completed Parts 0–III will find every idea in U14 already familiar under a new label — that is by design.

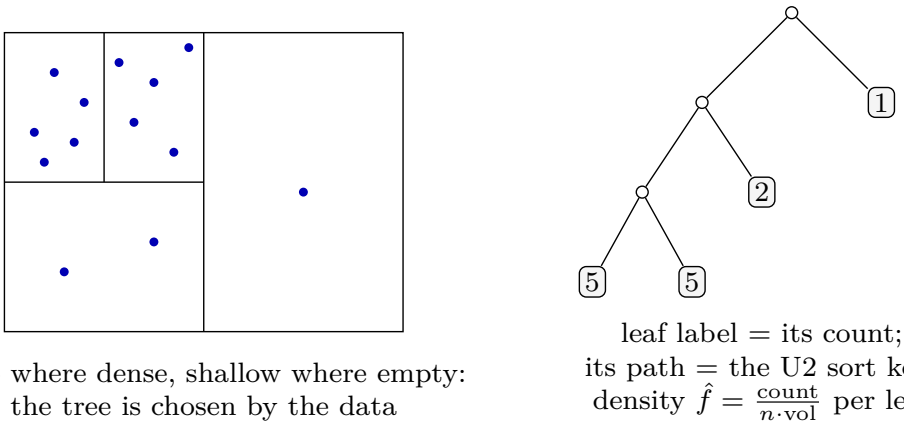
Instantiating the three-lens invariant (U0 box for U14).

> **RAM:** the classical histogram on a fixed grid is the baseline density estimator. Its weakness — the grid is chosen *before* seeing the data — motivates data-adaptive partitions. The **regular paving** (recursive binary splitting of the box) gives a data-adaptive histogram whose tree shape is determined by the sample itself.
>
> **PRAM:** building an SRP to depth d over n points assigns each point independently to a leaf (the assignment is deterministic given the tree shape), so subtrees are data-parallel. Work and depth of the parallel build are the content of §3.
>
> **DPRAM/MPC:** points are partitioned across machines; each machine builds a local SRP; local SRPs are **merged** by count addition (leaf counts are additive over disjoint datasets — an associative monoid). This is the **distributed sort on tree space**: map each point to its leaf label (a bit-string — the U2 sort key!), shuffle by label (one round), reduce by label to get global leaf counts. The SRP merge is the structural cousin of the Count-Min merge (U4): same additive monoid, same one-round aggregation.

Entry kit — what this chapter assumes; follow a link if you need the memory jog (map: Figure 1): F.S3 (densities and total variation) • F.C2 (trees and recursive partitions)

1. Intuition — from bins to trees

Every student in this course has seen a histogram: partition the real line into B equal-width bins, count the points in each bin, normalise by total count times bin width. The result is a piecewise-constant density estimate.



deep where dense, shallow where empty:
the tree is chosen by the data

leaf label = its count;
its path = the U2 sort key;
density $\hat{f} = \frac{\text{count}}{n \cdot \text{vol}}$ per leaf

Figure 20.1: A statistical regular paving: recursive midpoint bisection along the widest coordinate, split deep where the data is dense and left shallow where it is empty. Left, the partition; right, the same object as a tree carrying leaf counts. The leaf's root-to-leaf path is a bit-string — a sort key — so *building the distributed histogram is a distributed sort* (U2), and merging two SRPs is leaf-count addition: the same additive-monoid one-round merge as U4's sketches. U15 supplies the guarantee that selecting among such trees is honest.

The histogram has one free parameter: the bin width h. Choose h too large and all the density structure is washed out (high bias). Choose h too small and most bins contain one or zero points — the estimate is very jagged (high variance). There is an optimal h that balances these two errors, and it depends on the unknown density: the very thing we are trying to estimate. This is the **bias-variance tradeoff for histogram density estimation**.

Formally, for a density f on $\mathbb{R}$ and n iid observations, a histogram with fixed bin width h has integrated squared error MISE $\approx \frac{h^2}{12}\left(\int (f')^2\right) + \frac{1}{nh}$ — the bias is *first-order* because a piecewise-constant estimate cannot track the slope of f inside a bin (a first-order Taylor expansion of the bias, plus a variance calculation). The optimal rate-balancing choice is $h^* \propto n^{-1/3}$, giving MISE $\sim n^{-2/3}$ — the classical histogram rate in one dimension (a smooth *kernel* estimator, with its second-order bias $\frac{h^4}{4}(\int (f'')^2)$, improves this to $n^{-4/5}$; Scott 1979, Freedman–Diaconis 1981). In d dimensions the histogram rate slows to $n^{-2/(2+d)}$, and the number of bins to fill is exponential in d: the **curse of dimensionality**. A data-adaptive partition — one that allocates cells where the data is dense and ignores empty regions — is the right response: it is the subject of this unit.

The key idea: replace the regular d-dimensional grid with a **binary tree partition** of the data space. Each internal node of the tree bisects its box along one coordinate; the depth of each branch is chosen by the data (deep where many points fall, shallow where the space is empty). The resulting partition is a **regular paving** (RP), and when it carries counts it is a **statistical regular paving** (SRP). This is our research line (Sainudiin, Tucker, and collaborators; see §8).

Figure 20.1 shows the unit's data structure twice: as a partition and as a counting tree.

2. RAM — histogram density estimation and regular pavings

2.1 Classical fixed-width histogram (baseline)

Fix a box $\mathbf{x}_\rho \subset \mathbb{R}^d$ containing all n observations. Partition it into a grid of B^d cells of equal width $h = |\mathbf{x}_\rho|^{1/d}/B$ per axis. For each cell c, count $n(c)$ = the number of observations falling in c. The density estimate is

$$\hat{f}_{\text{hist}}(\mathbf{x}) = \frac{n(c(\mathbf{x}))}{n \cdot \text{vol}(c(\mathbf{x}))},$$

where $c(\mathbf{x})$ is the cell containing $\mathbf{x}$. The estimate is piecewise constant on the grid, takes $O(n)$ time to build, and requires $O(B^d)$ cells — exponential in d. The bin-width tradeoff and its mitigation by data-adaptive partitions is the motivating problem for the rest of this unit.

2.2 Regular pavings (RP)

Definition (Regular Paving). A **regular paving** (RP) of a box $\mathbf{x}_\rho \subset \mathbb{R}^d$ is a binary tree s in which: - The root corresponds to $\mathbf{x}_\rho$. - Each internal node bisects its box along one coordinate at the midpoint. - The leaves $\mathcal{L}(s)$ partition $\mathbf{x}_\rho$ into non-overlapping boxes (the **cells** of the paving).

The bisection rule is fixed: along the coordinate of widest extent (ties broken lexicographically). This regularity is what gives RP its name, and it is what enables the **bit-string leaf label** described in §4: the path from root to a leaf is a sequence of left/right binary decisions, one per split, encoding exactly which half-space the data point occupies at each depth. This path label is the tree-space analogue of a sort key.

Definition (Mapped Regular Paving, MRP). An MRP is a triple (s, A, f) where s is an RP and $f : \mathcal{L}(s) \to A$ assigns a value in A to each leaf. MRPs define piecewise-constant functions over the data space: given a point $\mathbf{x}$, compute its leaf $l(\mathbf{x}) \in \mathcal{L}(s)$ and return $f(l(\mathbf{x}))$.

Definition (Statistical Regular Paving, SRP). An SRP is an MRP with $A = \mathbb{Z}_+$: the value at each leaf is the count of training points whose coordinates fall in that leaf's box. The density estimate from an SRP $(s, n(\cdot), N)$ (N = total points) is

$$\hat{f}_{\text{SRP}}(\mathbf{x}) = \frac{n(l(\mathbf{x}))}{N \cdot \text{vol}(l(\mathbf{x}))}.$$

This is the same formula as the fixed-width histogram (§2.1), with the important difference that the partition $\mathcal{L}(s)$ is data-adaptive — the tree s is grown by the data, not prescribed in advance.

Sparse SRP. In d dimensions, most cells of a full binary tree to depth D are empty. The sparse SRP omits empty leaves from its representation, storing only cells with count $n(l) > 0$. Storage is $O(N)$ — one entry per distinct occupied cell — instead of $O(2^D)$. This is critical for high-dimensional data.

2.3 Building an SRP (RAM algorithm)

Given n points in $\mathbf{x}_\rho$ and a maximum depth D:

```
Build-SRP(points, root-box, depth):
  if depth = 0 or |points| = 0: return a leaf with count |points|
  split root-box along its widest coordinate at midpoint
  left-points  := points falling in the left  half-box
  right-points := points falling in the right half-box
  return a node with children:
    left  := Build-SRP(left-points,  left-half-box,  depth-1)
    right := Build-SRP(right-points, right-half-box, depth-1)
```

RAM cost. At each level, every point is tested against one threshold: $O(n)$ work per level, $O(nD)$ total. Building to depth D takes $T(n, D) = O(nD)$ time.

The partially-ordered space of tree shapes is the **paving lattice**: one RP s_1 is a **refinement** of s_2 (written $s_1 \leq s_2$) if every leaf of s_2 can be obtained by merging some leaves of s_1. The finest paving has every point in its own leaf; the coarsest is the root box. The MDE (U15) searches this partially-ordered space for the statistically best SRP.

3. PRAM — parallel tree build

3.1 The independence of subtrees

Observation. At depth d, the tree has at most 2^d subtrees. The assignment of a point $\mathbf{x}$ to its depth-k subtree depends only on the path label of $\mathbf{x}$ to depth k — and these labels for different points are computed independently. Once we know the split at depth k, the points falling in the left child and the points falling in the right child are disjoint sets, and the two subtrees are built independently.

This means the tree can be built **level by level** in parallel.

3.2 Work and depth

Level-parallel build. Process all nodes at depth k simultaneously, then move to depth $k+1$.

- **Work.** At each level, each point is tested once: $O(n)$ tests per level, D levels, giving $W = O(nD)$ — the same as the sequential algorithm. The parallel build is **work-optimal**.
- **Depth.** The critical path is the sequence of D level-processing steps: $D = O(D)$. With p processors, Brent gives $T_p \leq O(nD/p + D)$.

Parallelism. The parallelism of the build (the ratio W/D) is $O(n)$: adding processors pays fully until $p \approx n$. In practice the first $\log n$ levels are depth- limited (few nodes, many processors idle); below depth $\log n$ the algorithm is fully work-limited (many nodes, each processing a fraction of the points).

3.3 Point-to-leaf assignment as independent mapping

The path of a point $\mathbf{x}$ to its leaf at depth D is a deterministic function of $\mathbf{x}$ and the root box $\mathbf{x}_\rho$. There are no data dependencies *across* points: each of the n points can have its depth-D leaf label computed independently. This is a **parallel map** ($W = O(nD)$, $D = O(D)$, no communication between points), and it is the key observation for the DPRAM algorithm.

4. DPRAM / MPC — distributed sort on tree space

This section is the conceptual centerpiece of U14.

4.1 Leaf labels as sort keys — the U2 connection

A point $\mathbf{x}$ in the root box $\mathbf{x}_\rho$ has a **leaf label** at depth D: a bit-string $\ell(\mathbf{x}) \in \{0,1\}^D$ recording the left/right binary choices at each level of the RP. Left = 0 (the point falls in the left half-box at this level); right = 1. The labels are computed coordinatewise and dimensionally: at level k, the bit is 0 if $\mathbf{x}$'s coordinate along the split axis is below the midpoint, 1 otherwise.

This is **sorting on tree space**: the lexicographic order on leaf labels is a total order on the leaves of the tree, and mapping each point to its leaf label is assigning each point a sort key. The points that share a leaf label are exactly the points that belong to the same cell.

Recognise this structure: **it is the sort from U2**, applied in a tree-structured key space instead of the integers. The bit-string key is the digital analogue of a range-partition key in sample-sort (U2), with the tree structure making the partition hierarchy explicit.

4.2 The distributed SRP algorithm

Algorithm (Distributed SRP via sort on leaf labels).

Given n points partitioned across M machines, a root box $\mathbf{x}_\rho$, and maximum depth D:

1. **Map (local).** Each machine m maps each of its local points $\mathbf{x}$ to its leaf label $\ell(\mathbf{x}) \in \{0,1\}^D$. Cost per machine: $O((n/M) \cdot D)$.
2. **Shuffle (one round).** Group all $(\ell(\mathbf{x}), \mathbf{x})$ pairs by $\ell(\mathbf{x})$; route each pair to the machine responsible for its leaf label. In Spark: `rdd.map(x => (leafLabel(x), x)).groupByKey()` or equivalently `reduceByKey(_ + _)` where the accumulated value is the count.
3. **Reduce (local).** Each machine holds all points in its assigned leaves. Compute `count(leaf)` for each leaf. The result is the sparse SRP: only occupied leaves appear.

Cost. $R = 1$ shuffle round. Communication: $O(n \cdot D/M)$ bits per machine, about $nD/(8M)$ bytes (each point's label is D bits). Local work: $O((n/M) \cdot D)$ per machine. Total work: $W = O(nD)$.

4.3 SRP merge — an additive monoid

Suppose we have two machines that independently built SRPs from disjoint datasets of size N_1 and N_2 over the same root box $\mathbf{x}_\rho$. Let s_1 and s_2 be the respective trees. Define the **union paving** $s = \mathrm{RPUnion}(\mathcal{L}(s_1), \mathcal{L}(s_2))$ as the coarsest paving that is a refinement of both s_1 and s_2 (the common refinement: the *meet* $s_1 \wedge s_2$ under the order $\leq$ of §2.3, where finer is smaller; as trees, the union of the two node sets, hence the name). Then:

$$n_{\text{merged}}(l) = n_1(l) + n_2(l) \quad \text{for each leaf } l \in \mathcal{L}(s),$$

where $n_1(l)$ is the number of dataset-1 points in l, and similarly for n_2. Note that when l is strictly finer than its enclosing leaf l' of s_1, the stored count $n_1(l')$ cannot simply be copied down: copying it into every sub-leaf of l' would count those points once per sub-leaf. The count must be re-read from the dataset-1 points in l' (densities, not counts, are what propagate down unchanged). With counts so defined, the merged SRP has total count $N_1 + N_2$ and is a valid SRP of the combined dataset.

This operation — the **MRPOperate with** $\star = +$ (Harlow-Sainudiin-Tucker 2012, formalised in the RSS manuscript) — is: - **Associative:** $(s_1 \oplus s_2) \oplus s_3 = s_1 \oplus (s_2 \oplus s_3)$ (tree union is associative, as the lattice meet is). - **Commutative:** $s_1 \oplus s_2 = s_2 \oplus s_1$. - **Identity element:** the empty paving (leaf count 0 everywhere).

The SRP merge is a **commutative monoid**. This is the algebraic structure from U4 (§6.2 of U4: mergeable sketches as monoids): Count-Min merged by coordinate addition, HLL merged by coordinate maximum, SRPs merged by count addition over the common refinement. The same tree-reduce pattern applies.

Distributed density estimate. Each machine builds a local SRP from its shard. The M local tree shapes are merged in $O(\log M)$ rounds via tree-reduce (each round takes the union of the node sets of a pair). The resulting common refinement is broadcast, and each machine re-counts its own shard on its leaves (one local map pass). The per-leaf counts are then summed

by tree-reduce, and the root holds the global SRP. (When all machines share one tree shape, e.g. a fixed-depth cut, the reshaping steps are void and the merge is plain count addition.) The global density estimate $\hat{f}_{\text{SRP}}$ follows immediately from the merged counts and volumes.

DPRAM cost summary.

Phase	Work	Rounds	Communication
Local map (leaf labels)	$O(nD/M)$ per machine	0	0
Shuffle (sort on labels)	$O((n/M)\log(n/M))$ local sort	1	$O(n \cdot D/M)$ bits per machine ($O(nD)$ in all)
Reduce (count per leaf)	$O(n/M)$ per machine	0	0
SRP merge (tree-reduce)	$O(N_{\text{leaves}} \log M)$	$O(\log M)$	$O(N_{\text{leaves}} \cdot \log M)$ cells

In Spark, the three phases map to: `map(leafLabel)` → `reduceByKey(_ + _)` (the shuffle is the stage boundary) → `collect` the sparse count table.

4.4 Averaging over pavings — distributed histogram

If each machine has a pre-built SRP of the same tree shape (e.g. from a fixed-depth cut), the density *average* over two SRPs uses the same MRPOperate with $\star$ = midpoint: the averaged density at each leaf is the mean of the two local density values. This is the **density averaging** operation, used in the distributed MDE algorithm of U15 for combining partial estimates.

5. Statistical guarantees — L1-consistency (stated)

The SRP histogram described above is a nonparametric density estimator. Its statistical behaviour is the subject of a body of classical theory, and of our own research line.

L1-consistency of data-adaptive histograms (Lugosi-Nobel 1996; histogram estimates in Devroye-Lugosi 2001, §10.3). A histogram density estimator $\hat{f}_n$ is *L1-consistent* for the true density f if $\|\hat{f}_n - f\|_1 \to 0$ as $n \to \infty$, for all $f \in L^1(\mathbb{R}^d)$. Here $\|\cdot\|_1 = \int |\cdot|$ is the L1 distance (twice the total variation distance, by the Scheffé identity).

For data-adaptive histograms on recursive binary partitions, L1-consistency holds under three conditions (Lugosi-Nobel 1996): 1. **Partition class growth:** the number of distinct partitions at sample size n grows sub-exponentially in n. 2. **Cell count sublinearity:** the number of cells grows as $o(n)$. 3. **Cell diameters shrink:** the diameter of the cell containing any fixed point x tends to zero as $n \to \infty$.

Regular pavings built to depth $D = D(n) \to \infty$ with $2^D = o(n)$ satisfy all three conditions. The conclusion — $\hat{f}_n \to f$ in L1, almost surely, for every density f — is **universal**: no parametric assumption on f, no smoothness requirement.

The minimum-distance estimator (MDE) — forward pointer to U15. The SRP histogram above builds to a fixed depth D. Choosing D is a model-selection problem: the optimal depth depends on f, which is unknown. The **minimum-distance estimator** (Devroye-Lugosi CMDE, ch. 6; our RSS paper) selects the SRP from a data-driven sequence of coarsenings (the SEBTreeMC priority chain) by minimising an L1 distance to the empirical measure. This selection rule is statistically principled and avoids the bin-width tuning problem. U15 gives the full treatment: the Scheffé set, the Yatracos class, the MDE convergence rate, and the connection to the CMDE Lean certificates (already green in the course provedog). The **L1 error rate and the Scheffé/Yatracos arguments belong to U15**; U14 delivers the data structure and the distributed algorithm.

6. Lab sketch — project track

Project-track lab (no weekly receipt): run these experiments free and locally in the course lab image (`labs/image/`) — or any Spark-in-Docker environment — as deep-dive and group-project material.

Experiment 1 — Build a distributed SRP histogram ($n = 10^8$ points via sort-on-labels).

Generate $n = 10^8$ points from a mixture of 3 Gaussians in $d = 2$ dimensions (known true density). Map each point to its depth-D leaf label (bit-string, $D \in \{8, 12, 16\}$). Use `rdd.map(leafLabel).reduceByKey(_ + _)` to compute leaf counts. Report: number of occupied leaves; build time; shuffle stage size in bytes. Compare with a fixed-grid histogram at the same number of cells.

Experiment 2 — Merge two shards' SRPs.

Partition the 10^8 points across two workers. Each worker builds a local SRP of depth 12. Merge the two SRPs by count addition over the common refinement (implement MRPOperate). Verify: the merged SRP counts equal the global SRP counts. Measure: the merge cost vs the full-dataset build cost.

Experiment 3 — L1 error vs depth on synthetic mixtures.

For $n \in \{10^5, 10^6, 10^7\}$ and $D \in \{4, 8, 12, 16\}$: compute the empirical L1 error $\|\hat{f}_{\text{SRP}} - f\|_1$ (approximated by a fine numerical quadrature of the true density f and the SRP estimate). Plot L1 error as a function of D for each n. Observe the bias-variance tradeoff: shallow (small D) is high-bias; deep (large D) is high-variance (many empty leaves). *The optimal D increases with n — motivating the data-driven MDE selection of U15.*

Deliverable: notebook + one-page cost-accounting in the three-lens vocabulary (noting where the leaf label is the U2 sort key, and where the SRP merge is the U4 monoid pattern).

7. Lean certificate target

Target (new — U14): paving-refinement partial order + counts-additivity under merge (SCHEMATIC).

The mathematical claim for U14 is finite-tree combinatorics: the leaf-count function is additive under the SRP merge operation.

```
-- SCHEMATIC --- aligns to BMLSPar.lean conventions; cert realised at U14 cert
  time
-- Finite binary tree; leaf count; two SRPs s1 s2 over disjoint datasets;
-- merged SRP has count = sum of propagated counts.
theorem srp_count_additive (s1 s2 : SRP) (h : disjoint_datasets s1 s2) :
    count (merge s1 s2) = count s1 + count s2 := by
  sorry -- cert target: structural induction on the tree union
```

The claim is certifiable: it is a statement about finite trees and natural-number addition, with no measure theory. (By contrast, U15's rate theorem is probabilistic; its certs are green in the provedog with Hoeffding's tail bound taken as a hypothesis, U15 §4.2 — U14's cert is the simpler combinatorial precursor.)

Exercise U14.L1 (Lean, light). State and prove the base case: if s_1 and s_2 are both single-leaf SRPs (the root box), then `merge s1 s2` has count $n_1 + n_2$.

Exercise U14.L2 (deep). Prove the full `srp_count_additive` theorem by structural induction on the merge tree. Your proof must: (i) state the induction hypothesis explicitly for a tree of depth k; (ii) handle the inductive step by showing that at each internal split, the left-subtree merge and the right-subtree merge are independent and the counts add via $n_{\text{merged}}(l) = n_1(l) + n_2(l)$ for every leaf l of the common refinement; (iii) verify that the result holds for both the case where one tree is a leaf (base-case propagation) and where both trees are internal nodes. *(Lean target: fill the* `sorry` *in the schematic above using* `induction` *on the tree structure; the proof should not use* `sorry` *in the final certificate.)*

> **LLM lens** — where this unit lands on the running workload (U0L):
> U0L §1's statistical reading, made precise by this unit's vocabulary: an LLM is a density estimator on sequence space, with the autoregressive factorisation playing the role the tree partition plays here — a structured decomposition making an intractable joint density estimable piece by piece.

8. Sources (credited; archived under `scadamale-book-sources`)

Devroye, L. and Lugosi, G. (2001). *Combinatorial Methods in Density Estimation.* Springer Series in Statistics. — The canonical reference for L1-consistent density estimation; Ch. 2 (concentration inequalities, Hoeffding), Ch. 6 (MDE/Scheffé/Yatracos) and §10.3 (histogram estimates) are the foundation for U14 and U15. **Archived locally:** `.ias/research-archive/books/devroye-lugosi-2001-cmde.pdf` (sha256: `22997b9b...`); cited, not reproduced.

Lugosi, G. and Nobel, A. (1996). Consistency of data-driven histogram methods for density estimation and classification. *Annals of Statistics* 24(2):687–706. — The three-condition L1-consistency theorem for data-adaptive histograms stated in §5.

Sainudiin, R., Tucker, W., Wiklund, L., Graner, J., Sandstedt, A., Viitasaari, L., and Ilmonen, P. (Manuscript; in preparation for *Journal of the Royal Statistical Society.*) *Distributed Sparse Binary Trees for L1 Minimum Distance Estimation of Any Unknown Density.* — **Our research line.** Introduces the formal RP/MRP/SRP definitions, the MRPOperate arithmetic, the four-phase MDE pipeline (box hull → finest histogram → SEBTreeMC coarsening → MDE search), and the empirical demonstration at terabyte scale ($d = 1000$, $n = 10^8$). The distributed sort-on-labels algorithm of §4 draws from the Spark implementation in this manuscript. The predecessor chain: Harlow-Sainudiin-Tucker (2012; basic MRP arithmetic) → Teng-Kuhn-Sainudiin (2012; aircraft trajectories) → Sainudiin-Teng-Harlow-Lee (2012; Bayesian histogram MCMC) → Sainudiin-Teng (2019; MDE + universality) → Sainudiin-Tucker-Wiklund (2020) + Graner Master's thesis (2022) + Sandstedt Master's thesis (2023) → the present manuscript.

Breiman, L., Friedman, J., Olshen, R., and Stone, C. (1984). *Classification and Regression Trees* (CART). Wadsworth. — Context cite: the CART tradition of binary recursive partitioning; the RP is the regular (midpoint) specialisation that enables the MRP arithmetic.

Stone, C. (1980). Optimal rates of convergence for nonparametric estimators. *Annals of Statistics* 8(6):1348–1360. — The minimax rate $n^{-4/(4+d)}$ for smooth densities in d dimensions; the curse-of-dimensionality motivation for sparse representations.

Wikipedia. *Histogram*, *Nonparametric statistics*, *Binary tree*, *Density estimation.* — Background and cross-references.

9. The invariant (U14 instantiation)

> **RAM:** the fixed-width histogram has bin-width h chosen in advance, giving MISE $\sim n^{-2/(2+d)}$ at best. Data-adaptive histograms on regular pavings (recursive binary

splits of the box) choose the partition from the data: the SRP carries leaf counts $n(l)$ and yields density $\hat{f}(x) = n(l(x))/(N \cdot \mathrm{vol}(l(x)))$. Build time: $O(nD)$; storage: $O(N)$ sparse.

PRAM: subtrees at the same depth are independent: the build is level-parallel with $W = O(nD)$ (work-optimal) and $D = O(D)$ depth. Brent: $T_p \leq O(nD/p + D)$.

DPRAM/MPC: map each point to its leaf label (bit-string — the U2 sort key) in parallel; shuffle by label (one round); reduce by label to leaf counts. SRP merge = count addition over the common refinement — an additive monoid, aggregated by tree-reduce in $O(\log M)$ rounds. Total: $R = 1$ shuffle round for the sort-on-labels build ($O(\log M)$ rounds if local SRPs are merged by tree-reduce instead), $O(n \cdot D/M)$ bits per machine. The L1-consistency of the resulting estimator is stated (§5); the MDE model-selection and the full statistical guarantee are the subject of U15.

10. Exercises

Light (examined-core level — suitable for optional Part IV homework):

E14.1 Let $\mathbf{x}_\rho = [0,1]^2$ and $D = 3$. Draw the RP tree and its leaf boxes. How many leaves does the tree have? What is the maximum number of leaves at depth D? Write out the bit-string leaf label for the point $(0.3, 0.7)$.

E14.2 A 1-D dataset $\{0.1, 0.4, 0.55, 0.8, 0.9\}$ falls in $[0,1]$. Build the SRP to depth $D = 2$ by hand. Give the leaf counts and the piecewise-constant density estimate. What is the L1 distance (by direct integration) between this estimate and the uniform density on $[0,1]$?

E14.3 Explain in one paragraph why a fixed-width histogram with B^d bins is impractical for $d = 100$ dimensions when $n = 10^6$ (take $B = 2$). What is the expected fraction of non-empty bins for data drawn from a standard Gaussian in $d = 100$?

E14.4 In the distributed SRP algorithm (§4.2), the Spark operation `reduceByKey(_ + _)` on `(leafLabel, 1)` pairs produces the leaf counts. Identify: (a) which U0 primitive this instantiates (reduce, scan, or neither); (b) how many Spark stages (rounds) the job uses; (c) what the communication cost is in terms of n, D, and the number of machines M.

Deep (Module-3 / optional group-project track):

E14.5 *SRP mergeability proof.* Let s_1 be an SRP of N_1 points and s_2 be an SRP of N_2 points, both over the same root box. Define the merge $s = s_1 \oplus s_2$ as described in §4.3. Prove: (a) s is a valid SRP of the combined $N_1 + N_2$ points; (b) the merge is associative; (c) the merge is commutative. (*Hint:* argue by induction on the depth of the common refinement tree.)

E14.6 *Bit-string labels and sorted order.* Fix the root box $[0,1]^d$ and depth D. Show that the lexicographic order on depth-D leaf labels corresponds to a left-to-right ordering of the leaf boxes under a fixed coordinate priority. In 1D (with $d = 1$), how does this relate to a binary representation of real numbers? What does this say about the relationship between sorting by leaf label and sorting the points by their coordinates?

E14.7 *Bias-variance and depth.* For a 1-D density f on $[0,1]$ and an SRP to depth D (so that 2^D equally-sized bins of width $h = 2^{-D}$ are available): (a) using the histogram MISE formula from §2.1, $\mathrm{MISE} \approx \frac{h^2}{12}\int (f')^2 + \frac{1}{nh}$, express MISE as a function of D and n; (b) find the depth $D^*(n) = \arg\min_D \mathrm{MISE}(D, n)$ and show it satisfies $2^{D^*} \propto n^{1/3}$; (c) what is the resulting MISE rate as a function of n? (d) State how this rate compares to the optimal kernel density estimator rate $n^{-4/5}$ and explain why the kernel rate is faster.

Chapter 21

15 l1mde

U15 — L1-MDE / Scheffé–Yatracos: minimum-distance estimation in total variation

The ScaDaMaLe book, unit 15 (Part IV deep; the L1-paper tie-in). Our own exposition; original sources credited in §11.

Instantiating the three-lens invariant. The U0 box for minimum-distance estimation:

> **RAM:** the minimum-distance estimate (MDE) picks, from a countable family of candidate densities, the one whose empirical-measure distance to the data is smallest. The oracle inequality (Scheffé– Yatracos, CMDE Thm 6.3) bounds the resulting error in total variation. Cost: one pass over the data to compute empirical frequencies; one pass over the Yatracos class to evaluate candidate-pair distances — feasible when the class is finite or constructively enumerable.
>
> **PRAM:** no meaningful parallel layer — the candidate-pair tournament is embarrassingly parallel over the candidate pairs, but the dominant costs are in data and class size, not in parallelism; U15 has no PRAM section.
>
> **DPRAM/MPC:** distribute the data; per-shard empirical measures aggregate by addition (one tree-reduce round); Yatracos-class evaluations are per-set counts, computed as map+reduceByKey over shards; the candidate-pair tournament is a local sort on the roots. Ties directly to U14's tree-partitioned data (the candidate densities = SRP histograms on the tree).

Entry kit — what this chapter assumes; follow a link if you need the memory jog (map: Figure 1):
F.S3 (TV = half-L_1, Scheffé, minimax vocabulary)

1. Intuition — why total variation is the right currency

Every earlier unit measured cost in a computational unit: number of operations (RAM), work and depth (PRAM), rounds and bytes (DPRAM). This unit measures statistical estimation error — and the choice of *which* error metric is itself a theorem.

Two densities p and q on a measurable space $(\mathcal{X}, \mathcal{B})$ can be far apart in one sense and close in another. The metric that is *exactly the right one* for hypothesis tests, decision rules, and — as we will see — watermark detection is **total variation**: the maximum probability of an event that one distribution assigns but the other does not. It is the distributional analogue of the adversarial gap.

The deep surprise is the **Scheffé identity**: total variation equals *half* the L_1 distance between densities. The two measures that look different (TV = supremum over events; L_1 = integral of pointwise difference) are the *same thing* up to a factor of 2. This identity is not a coincidence — it is why L_1 estimation is the natural goal of minimum-distance estimation, and why the machine-checked cert `ScheffeFinite` is the load-bearing rung of the whole construction.

The **Yatracos method** exploits this: given n samples from an unknown density p, form the empirical measure μ_n, then choose the candidate $\hat{p}$ that is closest to μ_n in the supremum over a carefully chosen class of sets — the **Yatracos class** of the candidate family. The oracle inequality (CMDE Thm 6.3) says the resulting $\hat{p}$ is almost as good as the best candidate, with an additive price of order $\sqrt{V(n)/n}$, where $V(n)$ is the effective complexity of the candidate class as measured by the VC dimension of the Yatracos class.

By the end of this unit, students will have: (i) proved Scheffé's identity themselves; (ii) seen the full oracle-inequality chain, 10 lines, with every step named; (iii) understood VC dimension and Sauer–Shelah well enough to plug in the rate; (iv) run the distributed version on two synthetic families in the lab; and (v) recognised that the keyless watermark detector of a recent research paper by this course's instructor is *the same theorem applied*, with the LLM token distribution playing the role of the unknown p and the pre-mark corpus playing the role of the candidate family. One object; two disciplines; machine-checked from the ground up.

Figure 21.1 is the estimator as a tournament; every theorem below is about why the bracket is fair.

2. Total variation and Scheffé's identity

2.1 Total variation

Let p and q be probability mass functions on a finite alphabet $\mathcal{X}$ (the continuous case follows identically with integrals; we work discretely to align with the certified code).

Definition (Total variation). The **total variation distance** between p and q is

$$\mathrm{TV}(p,q) \;=\; \sup_{A\subseteq\mathcal{X}} \bigl|p(A)-q(A)\bigr| \;=\; \sup_{A\subseteq\mathcal{X}} \Bigl|\sum_{x\in A} p(x) - \sum_{x\in A} q(x)\Bigr|.$$

It is a metric: non-negative, zero iff $p=q$ pointwise, symmetric, satisfies the triangle inequality. Operationally: $\mathrm{TV}(p,q)$ is the maximum advantage any decision rule can achieve in distinguishing a single draw from p vs q. It is exactly the distributional gap that the statistical watermark detector must resolve.

Scheffé's set. Define $A^* = \{x \in \mathcal{X} : q(x) < p(x)\}$, the set of points where p dominates q. This set *achieves the supremum* in the definition of TV.

2.2 Scheffé's identity (proved)

Theorem (Scheffé 1947 / CMDE Thm 5.1; certified as `ScheffeFinite`). *For probability mass functions p and q on a finite alphabet $\mathcal{X}$:*

$$\mathrm{TV}(p,q) \;=\; \frac{1}{2}\sum_{x\in\mathcal{X}} |p(x)-q(x)| \;=\; \frac{1}{2}\|p-q\|_1.$$

Proof. Write $d(x) = p(x) - q(x)$. Since $\sum_x p(x) = \sum_x q(x) = 1$, we have $\sum_x d(x) = 0$. Split $\mathcal{X}$ into $\mathcal{X}^+ = \{x : d(x) > 0\}$ and $\mathcal{X}^- = \{x : d(x) \le 0\}$.

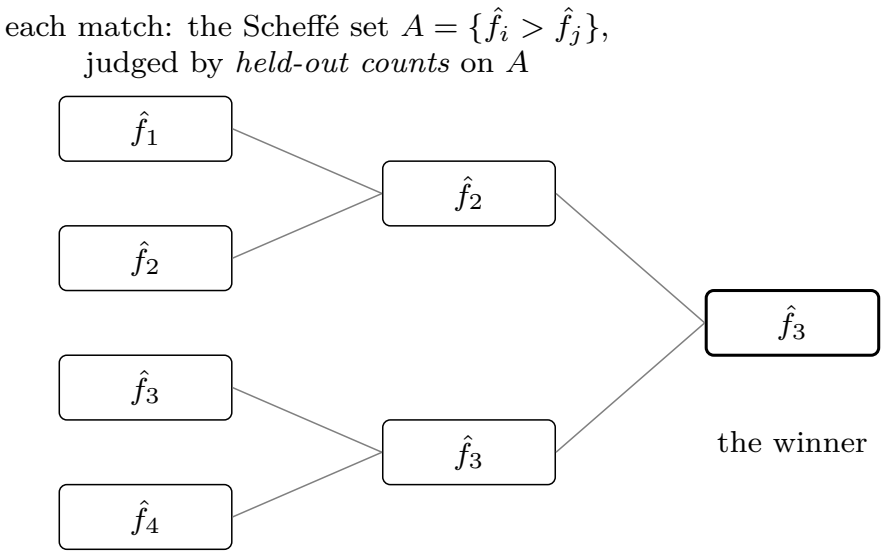


Figure 21.1: The minimum-distance idea as a tournament: candidate densities are players, and a match between $\hat{f}_i$ and $\hat{f}_j$ asks only one held-out question — how often did data land in the Scheffé set $\{\hat{f}_i > \hat{f}_j\}$, compared to what each candidate predicted? (F.S3: two densities can be distinguished exactly as well as their TV separation allows, by one set.) The bracket's winner is provably within a constant factor of the best candidate in total variation — a certified statement in the CMDE ladder — which is what lets U14's tree-built histograms be *selected* honestly, and what principled LLM evaluation (U0L §6, U18) would look like if benchmark culture adopted this unit.

Step 1: A^* achieves the supremum. For any $A \subseteq \mathcal{X}$:

$$p(A) - q(A) = \sum_{x \in A} d(x) \;\le\; \sum_{x \in A} d(x)^+ \;\le\; \sum_{x \in \mathcal{X}} d(x)^+ = \sum_{x \in \mathcal{X}^+} d(x) = p(A^*) - q(A^*),$$

where $d(x)^+ = \max(d(x), 0)$. Since $|p(A) - q(A)| \le \max(p(A) - q(A),\, q(A) - p(A))$, the supremum is achieved at A^* (or at $\mathcal{X} \setminus A^*$, which gives the same value by symmetry).

Step 2: the value at A^* equals $\frac{1}{2}\|p - q\|_1$. From $\sum_x d(x) = 0$:

$$\sum_{x \in \mathcal{X}^+} d(x) + \sum_{x \in \mathcal{X}^-} d(x) = 0 \;\Rightarrow\; \sum_{x \in \mathcal{X}^+} d(x) = -\sum_{x \in \mathcal{X}^-} d(x).$$

Therefore:

$$\begin{aligned} \|p - q\|_1 &= \sum_x |d(x)| = \sum_{x \in \mathcal{X}^+} d(x) + \sum_{x \in \mathcal{X}^-} (-d(x)) \\ &= 2 \sum_{x \in \mathcal{X}^+} d(x) = 2(p(A^*) - q(A^*)) = 2\,\mathrm{TV}(p, q). \quad \square \end{aligned}$$

Machine-checked certificate. This proof is exactly `IasCert.CMDE.scheffe_argmax` + `IasCert.CMDE.scheffe_eq_half_L1` in `IasCert/CMDE/ScheffeFinite.lean` — sorry-count 0, axiom footprint `[propext, Classical.choice, Quot.sound]`. Students *run the check* in the lab (§9).

Consequence: TV = 0 iff $p = q$ pointwise (`tv_eq_zero_iff_eq`, certified). A mark that is "distortion-free" in the sense of leaving the token distribution unchanged has TV zero to the unmarked distribution — this is the boundary the watermark paper makes precise.

2.3 TV's data-processing inequality (stated)

Theorem (CMDE Thm 5.2; certified as `CMDE-Sieve`). *For any measurable map $f : \mathcal{X} \to \mathcal{Y}$ and probability mass functions p, q:*

$$\mathrm{TV}(f_\sharp p,\ f_\sharp q)\ \le\ \mathrm{TV}(p, q),$$

where $(f_\sharp p)(y) = \sum_{x:f(x)=y} p(x)$ is the pushforward.

In plain language: coarsening the view can only reduce the TV gap. Refining the observable (a finer partition, a less lossy projection) can only reveal *more* of the distributional difference. The certified code: `IasCert.CMDE.tv_pushforward_le` (sorry-count 0). This theorem underlies the watermark paper's *rung-scope* argument: a negative scan at a coarser rung (character-level) says nothing about a finer rung (token-distribution), because TV is non-increasing under coarsening.

---

3. The Yatracos class and the minimum-distance estimate

3.1 Setup

Let $\{p_\theta : \theta \in \Theta\}$ be a family of probability mass functions on $\mathcal{X}$ — the **candidate family** or **model**. We observe n i.i.d. samples $X_1, \ldots, X_n$ from an unknown distribution p. The empirical measure is $\mu_n = \frac{1}{n}\sum_{i=1}^n \delta_{X_i}$.

3.2 The Yatracos class

Definition (Yatracos 1985). For each ordered pair $(\theta, \theta') \in \Theta \times \Theta$, define the **Yatracos set**

$$A_{\theta\theta'}\ =\ \{x \in \mathcal{X} : p_\theta(x) > p_{\theta'}(x)\}.$$

The **Yatracos class** of the family is

$$\mathcal{A}\ =\ \{A_{\theta\theta'} : \theta, \theta' \in \Theta\}.$$

Why this class? By Scheffé's identity, $2\,\mathrm{TV}(p_\theta, p_{\theta'}) = \|p_\theta - p_{\theta'}\|_1 = 2(p_\theta(A_{\theta\theta'}) - p_{\theta'}(A_{\theta\theta'}))$. So the TV between any two candidates is *determined by their masses on a set in $\mathcal{A}$*. The Yatracos class is the minimal class of sets that carries all the inter-candidate TV information.

3.3 The minimum-distance estimate

Definition. The **minimum-distance estimate (MDE)** is any

$$\hat\theta\ \in\ \arg\min_{\theta\in\Theta} \sup_{A\in\mathcal{A}} |\mu_n(A) - p_\theta(A)|.$$

In words: choose the candidate whose empirical-measure distances to all other candidates are as small as possible, measured over the Yatracos class. This is a tournament: candidate θ challenges θ' on the set $A_{\theta\theta'}$; the winner is the one whose frequency $\mu_n(A_{\theta\theta'})$ is closest to its claimed probability $p_\theta(A_{\theta\theta'})$.

3.4 The oracle inequality (proved fully)

Theorem (CMDE Thm 6.3; certified as `CMDE-L6a-MinDistanceOracle`). *Let* $\Delta_n = \sup_{A\in\mathcal{A}} |\mu_n(A) - p(A)|$ *be the uniform deviation of the empirical measure over the Yatracos class. For any candidate* $\theta^* \in \Theta$ *and the MDE* $\hat\theta$*:*

$$\mathrm{TV}(p_{\hat\theta},\, p) \;\le\; 3\,\mathrm{TV}(p_{\theta^*},\, p) + 2\Delta_n.$$

Proof (10-line triangle-inequality chain). Write $\hat\theta$ for the argmin and θ^* for any comparison candidate. Let $A = A_{\hat\theta\,\theta^*}$ be the Yatracos set for the pair $(\hat\theta, \theta^*)$.

Write $\mathrm{dev}(\theta) := \sup_{A\in\mathcal{A}} |\mu_n(A) - p_\theta(A)|$ for the empirical deviation of candidate θ over the Yatracos class $\mathcal{A}$, and $\Delta_n := \sup_{A\in\mathcal{A}} |\mu_n(A) - p(A)|$ for the uniform empirical error. The oracle inequality assembles the four steps discharged by `IasCert.CMDE.min_distance_oracle`.

Step 1 (TV triangle). $\mathrm{TV}(p_{\hat\theta}, p) \le \mathrm{TV}(p_{\theta^*}, p) + \mathrm{TV}(p_{\hat\theta}, p_{\theta^*})$.

Step 2 (Scheffé-pair + empirical-measure triangle). On the Scheffé set $A = A_{\hat\theta\theta^*}$, $\mathrm{TV}(p_{\hat\theta}, p_{\theta^*}) = p_{\hat\theta}(A) - p_{\theta^*}(A) = [p_{\hat\theta}(A) - \mu_n(A)] + [\mu_n(A) - p_{\theta^*}(A)] \le \mathrm{dev}(\hat\theta) + \mathrm{dev}(\theta^*)$.

Step 3 (MDE selection). $\hat\theta$ minimises the empirical deviation, so $\mathrm{dev}(\hat\theta) \le \mathrm{dev}(\theta^*)$; hence $\mathrm{TV}(p_{\hat\theta}, p_{\theta^*}) \le 2\,\mathrm{dev}(\theta^*)$.

Step 4 (Scheffé candidate-vs-truth). For any θ, $\mathrm{dev}(\theta) \le \sup_A |\mu_n(A) - p(A)| + \sup_A |p(A) - p_\theta(A)| = \Delta_n + \mathrm{TV}(p, p_\theta)$ (Scheffé identity for the last equality).

Assemble. Substituting Step 3 then Step 4 into Step 1:

$$\begin{aligned}
\mathrm{TV}(p_{\hat\theta}, p) &\le \mathrm{TV}(p_{\theta^*}, p) + 2\,\mathrm{dev}(\theta^*) \\
&\le \mathrm{TV}(p_{\theta^*}, p) + 2\big(\Delta_n + \mathrm{TV}(p_{\theta^*}, p)\big) \\
&= 3\,\mathrm{TV}(p_{\theta^*}, p) + 2\Delta_n. \qquad \square
\end{aligned}$$

Convention remark. In the classical Devroye–Lugosi L1 rendering (CMDE Thm 6.3), the same statement reads $\|f_{\hat\theta} - f\|_1 \le 3\min_\theta \|f_\theta - f\|_1 + 4\Delta_n$ — identical under Scheffé's $L_1 = 2\,\mathrm{TV}$ (halve both sides; Δ_n is the same quantity in both forms). The L_1 form is the one our certificate `min_distance_oracle` discharges: its `err` is the L_1 error, its Step-4 hypothesis reads $\mathrm{dev} \le \frac{1}{2}\,\mathrm{err} + \Delta$, and its conclusion is $\mathrm{err}(\hat\theta) \le 3\,\mathrm{err}(\theta) + 4\Delta$; the TV form above follows on dividing by 2.

Honest note. The constant cannot be brought down to 1: CMDE Fact 6.1 (§6.3, p. 51) exhibits, for the closely related Scheffé estimate choosing between two densities, examples where the limiting ratio of its expected L_1 error to the best candidate's is at least $2 - \varepsilon$ for any $\varepsilon > 0$. Whether 3 itself is attained is not settled by that fact. The proof above is the same four-step structure (selection; Scheffe-pair + empirical-measure triangle; Scheffe-vs-truth bound; triangle inequality) encoded in `IasCert.CMDE.min_distance_oracle`.

---

4. VC dimension, Sauer–Shelah, and the rate

4.1 VC dimension and shatter coefficients

Definition. A class $\mathcal{A}$ of subsets of $\mathcal{X}$ **shatters** a set $C = \{x_1, \ldots, x_m\} \subseteq \mathcal{X}$ if every subset of C is realised as $A \cap C$ for some $A \in \mathcal{A}$. The **VC dimension** of $\mathcal{A}$ is

$$\mathrm{VC}(\mathcal{A}) = \sup\{m : \exists C \text{ of size } m \text{ shattered by } \mathcal{A}\}.$$

The **shatter coefficient** (growth function) is

$$S(\mathcal{A}, n) = \max_{x_1,\ldots,x_n \in \mathcal{X}} |\{A \cap \{x_1, \ldots, x_n\} : A \in \mathcal{A}\}|.$$

It counts the maximum number of distinct restrictions of $\mathcal{A}$ to n points.

Theorem (Sauer 1972, Shelah 1972; in Mathlib as `Finset.card_le_card_shatterer` with `Finset.card_shatterer_le_sum_vcDim`). *If* $\mathrm{VC}(\mathcal{A}) = V < \infty$*, then for all* $n \geq 1$*:*

$$S(\mathcal{A}, n) \;\leq\; \sum_{k=0}^{V} \binom{n}{k},$$

and, for $n \geq V \geq 1$, $\sum_{k=0}^{V} \binom{n}{k} \leq (en/V)^V$.

Proof: Sauer's original 1972 argument proceeds by double induction on n and V; it is available in Mathlib for the finite-set setting and we use it as a library theorem. The key upshot: a class of VC dimension V can produce at most polynomially many (in n) distinct restrictions — and it is this polynomial that converts the union bound on Yatracos-class sets into a rate.

4.2 The L1-MDE rate (proved structurally)

Theorem (CMDE Ch 3/6; certified as `CMDE-L5-L6b-L1MDERate`). *Let* $\mathcal{A}$ *be the Yatracos class of a candidate family with* $|\mathcal{A}| = k^2$ *pairs (i.e.,* k *candidates). For any* $t > 0$*:*

$$P\big(\mathrm{TV}(p_{\hat{\theta}}, p) \geq 3\,\mathrm{TV}(p_{\theta^*}, p) + 4t\big) \;\leq\; k^2 \cdot B(t),$$

where $B(t) = 2\exp(-2nt^2)$ *is Hoeffding's two-sided tail bound for a single set's deviation.*

Proof shape. Three steps:

Step 1 (Union bound over the Yatracos class). The event $\{\Delta_n \geq t\}$ is a union over all $A \in \mathcal{A}$ of $\{|\mu_n(A) - p(A)| \geq t\}$. By the union bound:

$$P(\Delta_n \geq t) \;\leq\; |\mathcal{A}| \cdot \max_{A \in \mathcal{A}} P(|\mu_n(A) - p(A)| \geq t) \;\leq\; k^2 \cdot B(t).$$

Step 2 (Hoeffding for one set). $\mu_n(A) = \frac{1}{n}\sum_{i=1}^{n} \mathbf{1}[X_i \in A]$ is an average of i.i.d. bounded $[0,1]$ variables. Hoeffding's inequality gives $P(|\mu_n(A) - p(A)| \geq t) \leq 2e^{-2nt^2}$; the event is two-sided, so $B(t) = 2e^{-2nt^2}$ (the one-sided e^{-2nt^2} bounds only $P(\mu_n(A) - p(A) \geq t)$).

Step 3 (Connect oracle to rate). By the oracle inequality, the error event $\{\mathrm{TV}(p_{\hat{\theta}}, p) \geq 3\mathrm{TV}(p_{\theta^*}, p) + 4t\}$ implies $\{\Delta_n \geq t\}$ (if $\Delta_n < t$ the oracle gives error $< 3\mathrm{TV}(p_{\theta^*}, p) + 4t$). So the error probability is bounded by $2k^2 \cdot e^{-2nt^2}$.

Setting t. With $t = \sqrt{\log(2k^2/\delta)/(2n)}$, the probability is $\leq \delta$, and the error is $3\mathrm{TV}(p_{\theta^*}, p) + O(\sqrt{\log k/n})$. When the Yatracos class has VC dimension V, the union bound needs only the number of *distinct restrictions* on the sample, so the crude count k^2 is replaced by the shatter coefficient $S(\mathcal{A}, n) \leq \min\big(k^2, (en/V)^V\big)$ (Sauer–Shelah, §4.1); whenever the VC bound is the smaller, this gives the dimension-driven rate $O(\sqrt{V\log(n/V)/n})$.

Honest certificate note. The rate structure (union bound + oracle plug + Hoeffding abstracted as a hypothesis) is fully machine-checked as `CMDE-L5-L6b-L1MDERate` (sorry-count 0; `IasCert.CMDE.l1mde_rate`). The one remaining mechanical step — wiring the empirical-measure deviation into Mathlib's i.i.d. `iIndepFun` + `hasSubgaussianMGF_of_mem_Icc` to produce the concrete Hoeffding $B(t)$ — is pure probability plumbing, staged for cluster push; the rate's structure is locked.

---

5. DPRAM / MPC — distributed L1-MDE

5.1 The distributed setup

Data $\{X_1, \ldots, X_n\}$ is split across M machines (shards). Candidate family: $\{p_1, \ldots, p_k\}$ — in the ScaDaMaLe application these are SRP histogram densities on the binary-tree partition built in U14. The Yatracos class has k^2 sets; k is the number of candidate densities.

5.2 Per-shard empirical measures; aggregation

Round 0 (local map). Each machine m holds shard S_m of size n_m. For each Yatracos set $A_{ij} \in \mathcal{A}$, machine m computes the local count:

$$c_m(A_{ij}) \;=\; |\{X \in S_m : X \in A_{ij}\}|.$$

This is a `map` over the data: for each point, emit $(i, j, 1)$ if the point falls in A_{ij}.

Round 1 (reduceByKey). Shuffle by key (i, j); each reducer sums the local counts to obtain the global empirical count $c(A_{ij}) = \sum_m c_m(A_{ij})$. The global empirical measure value is $\mu_n(A_{ij}) = c(A_{ij})/n$.

Communication. Each machine sends k^2 counts of $O(\log n)$ bits each: total bytes $= O(M \cdot k^2 \cdot \log n)$. With k candidate densities and Sauer-Shelah bounding k^2 by at most polynomial in n, this is practical for the typical moderate-k case.

Rounds: $R = 2$ (one shuffle for the counts; one optional broadcast of the selected $\hat{\theta}$).

5.3 Candidate-pair tournament

After aggregation, the root machine holds $\{\mu_n(A_{ij})\}_{i,j}$ and the candidate masses $\{p_l(A_{ij})\}_{l,i,j}$ (every candidate's mass on every Yatracos set; known, since the candidate family is explicit). The tournament is:

$$\hat{\theta} = \arg\min_i \max_{A \in \mathcal{A}} |\mu_n(A) - p_i(A)|,$$

the sup over the *whole* Yatracos class, as in §3.3. This is local work at the root: k candidates against $O(k^2)$ sets, $O(k^3)$ comparisons — CMDE (§6.6, p. 54) prices it at $k^2(k-1)/2$ set-integrals, about k times the $k(k-1)/2$ of the Scheffé tournament — still sub-dominant to the shuffle for moderate k. (Taking the max only over candidate i's own row, $\max_j |\mu_n(A_{ij}) - p_i(A_{ij})|$, costs $O(k^2)$ but is a different estimator from §3.3's.)

5.4 Tie to U14: candidate densities from tree space

In U14, the candidate densities are SRP histograms — piecewise-constant densities on cells of a binary-tree partition of the data space. Each candidate p_θ corresponds to a partition at depth d, with 2^d cells (not to be confused with k, the number of candidates). The Yatracos set A_{ij} is a union of tree cells: it is determined by which cells have $p_i > p_j$. Both the map (point falls in cell? in Yatracos set?) and the reduce (sum counts) are natural Spark RDD operations on the tree-keyed data from U14.

Bytes and rounds summary. - **Rounds:** $R = 2$ (map + reduceByKey; result broadcast). - **Bytes:** $O(Mk^2 \log n)$ per round (counts up; selected density broadcast down). - **Local work:** $O(n/M \cdot k^2)$ per machine (point-in-Yatracos-set test for each data point and each of the k^2 set indicators, in the brute-force variant; in practice, point-in-tree-cell is a single tree traversal, and A_{ij} is a union of cells determinable before the data pass).

---

6. The security capstone: the keyless watermark detector is Scheffé–Yatracos in L_1

6.1 One theorem; two disciplines

The Scheffé–Yatracos oracle inequality is not only a statistical learning theorem. It is the exact mathematical structure of the **keyless LLM watermark detector** introduced in the recent preprint *"A Watermark for All, Formally"* (Sainudiin et al., 2026; Devroye–Lugosi CMDE (2001) + Yatracos (1985) + Scheffé (1947) are the cited ancestors).

The connection is the following. A statistical watermark shifts the next-token sampling distribution from the unmarked p_0 (the vendor's baseline) to a marked $p_w = p_0 + \delta_w$, where δ_w is a perturbation that depends on a secret key. The gap between the two distributions is exactly $\Delta = \mathrm{TV}(p_w, p_0)$ — the total-variation distance the detector must resolve. A recipient who does not hold the key cannot run the keyed z-test; but she *can* apply the L1-MDE machinery:

- **The candidate family** is $\{p_0^{(t)} : t \text{ is a time period}\}$ — the vendor's output distribution estimated from the recipient's dated corpus in each time window.
- **The Yatracos class** is the class of sets $\{A_{tt'}\}$ where the before-date distribution dominates the after-date distribution — observable as token-frequency differences across the date of the purported mark activation t_w.
- **The minimum-distance estimator** selects the period whose empirical distribution best matches the pattern predicted by a mark of strength Δ — this is the difference-in-differences (DiD) estimand proved in `IasCert.Watermark.DiD`.

The unification statement. The DiD detector's identification condition (proved in the preprint and certified as `Prp-DiD`) is a special case of the oracle inequality: the DiD isolates the watermark term $\gamma = \mathrm{TV}(p_w, p_0)$ under parallel trends, exactly as the MDE oracle bounds $\mathrm{TV}(p_{\hat{\theta}}, p)$ under the uniform-deviation condition. The watermark detection problem is an L1-MDE problem where the candidate family is the temporal slice of the vendor's output distribution.

6.2 Machine-checked certificate chain (students run this)

The cert chain is complete and students can run every link:

Cert	Module	Theorem(s)	What it certifies
`ScheffeFinite`	`IasCert.CMDE.ScheffeFinite`	`scheffe_argmax`, `scheffe_eq_half_L1`, `tv_eq_zero_iff_eq`	TV = ½ L1; A* achieves the sup; TV=0 iff equal
`CMDE-Sieve`	`IasCert.CMDE.Sieve`	`tv_pushforward_le`, `tv_zero_pushforward_zero`	TV non-increasing under coarsening; distortion-free is projection-invariant
`CMDE-L6a-MinDistanceOracle`	`IasCert.CMDE.MinDistance`	`yatracos_pair_dist`, `min_distance_oracle`	Oracle inequality (factor 3); Scheffe step grounded in L0
`CMDE-L5-L6b-L1MDERate`	`IasCert.CMDE.L1MDERate`	`max_dev_union_tail`, `max_dev_card_bound`, `error_exceed_subset_union`, `l1mde_rate`	Rate: P(error ≥ 3·opt + 4t) ≤ k² · B(t)
`Prp-DiD`	`IasCert.Watermark.DiD`	`did_isolates_gamma`, `did_null_of_no_watermark`	DiD isolates the watermark term γ = TV(p_w, p_0) under parallel trends

Every entry: sorry-count 0; axiom footprint the standard Lean/Mathlib classical triple `[propext, Classical.choice, Quot.sound]`. *Build tool:* `bash provedog/build-in-docker.sh`.

6.3 The honest boundary (STATED nodes)

Two nodes in the certificate chain remain at STATED (honest, with reasons):

- **Ch7 skeleton (CMDE Thm 7.1 — skeleton estimates and Kolmogorov entropy).** The covering-number version of the rate, applicable when k is infinite but the family has finite entropy at scale ε. We state the result and cite CMDE Ch 7; the full proof requires empirical-process theory beyond the scope of this unit. Cert target: `P-cmde-formalization` phase 2.
- **Minimax lower bounds (CMDE Ch 15 — Assouad / Fano).** The other side of the rate: how good *any* estimator can be, not just the MDE. Stated here (not proved in this unit) in the form: there exist families for which no estimator achieves TV error less than $\Omega(\sqrt{V/n})$. Cert target: phase-1 of the minimax agenda.

The point is pedagogically important: students see exactly which steps are machine-checked and which are stated-but-not-yet-certified, with a named plan for closing each gap.

6.4 The mathesis ethos in one sentence

The same 10-line triangle-inequality chain that the student proves in §3.4 — Scheffé + tournament + union bound + Hoeffding — is the mathematical core of a working LLM-watermark detection method. Pure mathematics is a security tool; the two are one object.

6b. The mirror: the input side (covert-channel scrub)

The detector of §6 reads bytes going **out** — a provenance mark at the *statistical* rung. Its mirror reads the bytes coming **in**. It has been shown (stego-finetuning, arXiv:2603.08104; cf. Trojan-Source (Boucher–Anderson), OWASP LLM04, and the 250-document poisoning result) that a language model, poisoned by ordinary users, can *emit* invisible covert characters to third parties while its interface looks benign — the vendor need not know. So the input side needs a scrub.

A scrub worth trusting is a **total provable transform**: it strips the covert byte-rung class (the character classes are the canonical invisible/covert set, not a re-invented list) and emits a small **diff** — the raw is kept as evidence, no doubling — and it is proven *coverage-complete*, *idempotent*, and *provenance-lossless* (the raw is reconstructible from filtered + diff). The punchline closes the loop of this unit: the keyless watermark detector of §6 and the covert-character scrub are the *same* L_1 / total-variation geometry on the *same* resolution lattice (U00S §5b, the resolution lattice) — one *detecting* at the statistical rung, one *stripping* at the byte rung. Three disciplines — statistics, control, cyber-defence — one object.

---

7. The cert ladder (U15 instantiation)

Certified (GREEN — students RUN and EXTEND): - `ScheffeFinite` — TV = ½ L1; the argmax set A*. - `MinDistance oracle` — the 3·opt + 4Δ oracle inequality (L1 form; 3·opt + 2Δ in TV). - `L1MDERate` — rate: P(error) ≤ k² · B(t), with the per-set tail B(t) taken as a hypothesis (Hoeffding: B(t) = 2·exp(−2nt²)). - `CMDE-Sieve` — TV non-increasing under coarsening (data-processing). - `Prp-DiD` — DiD watermark isolator (the security capstone cert). - Sauer–Shelah — via Mathlib `Finset.card_le_card_shatterer` + `Finset.card_shatterer_le_sum_vcDim`.

STATED (named open nodes): - Ch7 skeleton (covering-number rate, empirical-process lift) — cert target: P-cmde-formalization phase 2. - Minimax lower bound (Assouad/Fano, CMDE Ch 15) — cert target: phase-1 minimax agenda.

This is where the book's cert discipline pays in full. Every earlier unit named a Lean cert target. U15 is where those targets *close*: the Scheffé identity, the oracle inequality, the rate — all green. The student's first task in the lab is to run the build and read the sorry-count output; the second is to blank the oracle-inequality chain and re-derive two steps.

---

8. The invariant (U15 instantiation)

RAM: choose the candidate minimising the sup-deviation over the Yatracos class — cost $O(nk^2)$ for n data points and k candidates. Oracle inequality: error $\leq 3\cdot$ best-candidate error $+2\Delta_n$ (TV form; $4\Delta_n$ in the classical L1 rendering, the form certified); rate: $\Delta_n = O(\sqrt{\log k/n})$ from Hoeffding + the union bound — certified with Hoeffding's tail taken as a hypothesis (§4.2) — and $O(\sqrt{V\log(n/V)/n})$ via Sauer–Shelah (in Mathlib), the latter stated, not certified.

PRAM: trivially parallel over candidate pairs (no meaningful depth reduction needed; the bottleneck is data access, not computation).

DPRAM/MPC: per-shard local counts (map); shuffle on Yatracos-set key (reduceByKey, one round); local tournament at root (second round). Bytes: $O(Mk^2\log n)$. Candidate densities from U14's tree space. The distributed empirical measure is the sum of per-shard measures — exact, not approximate.

SECURITY: the keyless watermark detector is Scheffé–Yatracos in L_1; $\Delta = \mathrm{TV}(p_w, p_0)$ orders the whole mark space; $\Delta = 0$ is the exact keyless- undetectability boundary. Cert chain: 5 certs, sorry-count 0.

---

9. Lab — project track

Project-track lab (no weekly receipt): run these experiments free and locally in the course lab image (`labs/image/`) — or any Spark-in-Docker environment — as deep-dive and group-project material. Experiment 1 additionally uses the provedog cert-runner container (see its build script below).

Experiment 1 — Run the cert ladder. In the provedog docker (`bash provedog/build-in-docker.sh`): 1. Verify the build log reports `sorry_count=0` for all four CMDE certs and `PrpDiD`. 2. Run `#print axioms IasCert.CMDE.scheffe_eq_half_L1` and `IasCert.CMDE.min_distance_oracle`; confirm the axiom footprint is exactly `[propext, Classical.choice, Quot.sound]`. 3. Blank the two middle proof steps of `min_distance_oracle` (the MDE-selection line and the uniform-deviation line) and re-derive them. Submit the completed `.lean` file.

Experiment 2 — Distributed MDE on two synthetic families. Two families: $\mathcal{F}_1$ = uniform histograms on $[0,1]$ with 2^d bins (varying d); $\mathcal{F}_2$ = Beta-mixture histograms on the same bins. 1. Generate $n = 10^6$ samples from a mixture $p = 0.7\cdot p_1 + 0.3\cdot p_2$ where $p_1 \in \mathcal{F}_1$, $p_2 \in \mathcal{F}_2$. 2. Distribute across $M = 8$ Spark partitions; compute per-shard empirical measures via `map + reduceByKey` on Yatracos-set indicators. 3. Run the tournament; report $\hat{\theta}$ and the measured TV distance to p. 4. Vary $n \in \{10^4, 10^5, 10^6\}$ and plot TV-error vs n; overlay the $O(\sqrt{\log k/n})$ rate bound. *Deliverable:* notebook + two-paragraph cost-accounting in the three-lens vocabulary.

Experiment 3 — The DiD / keyless-detection worked example. A simulated JEM family: vendor output is a token-histogram p_0 on a 100-token alphabet; at time $t_w = 50$ the

vendor activates a mark shifting $p_0 \to p_w$ with $\mathrm{TV}(p_w, p_0) = \Delta$ (vary $\Delta \in \{0.01, 0.05, 0.1, 0.2\}$). A control vendor never activates a mark. Simulate $n_1 = n_2 = 5000$ token draws per period. 1. Compute the DiD estimand (before/after × treatment/control) on token frequencies. 2. Report the DiD estimate vs the true Δ; overlay the L1-MDE rate-implied detection threshold $t^* = \sqrt{\log k^2/(2n)}$. 3. At what Δ does the detector achieve power ≥ 0.8 (over 100 repeated draws)? *Deliverable:* notebook + one-paragraph connection to `Prp-DiD` and the oracle inequality constants.

---

10. Lean certificate (students extend)

The GREEN certs are not schematic — they are the real thing. Students run and extend:

```
-- EXERCISE U15.L1 (light): blank the two middle steps; re-derive.
-- In provedog/IasCert/CMDE/MinDistance.lean, min_distance_oracle.
-- L1 form: err k = candidate k's L1 error to the truth, dev k = its
  Yatracos-class
-- deviation from the empirical measure, L i j = pairwise L1 distance. The four
  hypotheses are:
--   hsel  : ∀ k, dev iHat ≤ dev k                      -- MDE selection
--   hpair : ∀ j, L iHat j ≤ 2 * dev iHat + 2 * dev j  -- Scheffe pair +
  empirical triangle
--   hdev  : ∀ k, dev k ≤ (1/2) * err k + Δ             -- Scheffe
  candidate-vs-truth
--   htri  : ∀ j, err iHat ≤ err j + L iHat j           -- L1 triangle inequality
-- BLANK: derive L iHat j ≤ 2 * err j + 4 * Δ
-- BLANK: apply htri to conclude err iHat ≤ 3 * err j + 4 * Δ   (TV form: halve
  both sides)
-- (Both are linarith calls over the hypotheses -- the point is to NAME the step,
--  not to discover a hard computation.)
theorem min_distance_oracle_student {ι : Type*} (err dev : ι → ℝ) (L : ι → ι →
  ℝ) (Δ : ℝ)
    (iHat : ι) (hsel : ...) (hpair : ...) (hdev : ...) (htri : ...) :
    ∀ j, err iHat ≤ 3 * err j + 4 * Δ := by
  -- fill in
  sorry
```

Exercise U15.L2 (deep). Extend the Sieve cert: state and prove that for *two* projections f_1, f_2, $\mathrm{TV}(\pi_{f_1 \circ f_2}(p), \pi_{f_1 \circ f_2}(q)) \leq \mathrm{TV}(\pi_{f_2}(p), \pi_{f_2}(q)) \leq \mathrm{TV}(p, q)$. (Composing two coarsenings is a composition of pushforwards — the chain rule for TV data-processing.)

LLM lens — where this unit lands on the running workload (U0L):
Perplexity is exponentiated log-loss: LLM evaluation is distance-between-distributions, this unit's home ground. The MDE lens — compare candidate models by a principled distance under an honest convention — is exactly what benchmark culture lacks and U16 turns into contracts.

11. Sources (credited; archived under `scadamale-book-sources`)

Devroye & Lugosi (2001). L. Devroye and G. Lugosi, *Combinatorial Methods in Density Estimation*, Springer 2001. (CMDE.) — The foundational reference for this entire unit: Thm 5.1 (Scheffé's identity, Ch 5), Thm 5.2 (TV data-processing, Ch 5), Thm 6.3 (minimum-distance oracle inequality, Ch 6), the VC/Sauer–Shelah rate (Chs 3–4), Thm 7.1 (skeleton estimates, Ch 7),

and the minimax lower bounds (Ch 15). Archived in `.ias/research-archive/books/devroye-lugosi-2001-cmde.pdf` (PDF-verified 2026-08-22). The exposition above is our own; we never reproduce © text.

Yatracos (1985). Y. G. Yatracos, *Rates of convergence of minimum-distance estimators and Kolmogorov's entropy*, Annals of Statistics 13(2):768–774, 1985. — The minimum-distance estimator construction and the oracle inequality. The Yatracos class and the factor-3 result carry his name.

Scheffé (1947). H. Scheffé, *A useful convergence theorem for probability distributions*, Annals of Mathematical Statistics 18(3):434–438, 1947. — The identity TV = ½ L1 and the argmax set A*, proved in the σ-finite measure setting; our §2.2 is its finite/discrete specialisation.

Vapnik & Chervonenkis (1971). V. N. Vapnik and A. Ya. Chervonenkis, *On the uniform convergence of relative frequencies of events to their probabilities*, Theory of Probability and Its Applications 16(2):264–280, 1971. — VC dimension and the uniform convergence theorem.

Sauer (1972) / Shelah (1972). N. Sauer, *On the density of families of sets*, Journal of Combinatorial Theory Series A 13:145–147, 1972. S. Shelah, *A combinatorial problem; stability and order for models and theories in infinitary languages*, Pacific Journal of Mathematics 41(1):247–261, 1972. — The Sauer–Shelah lemma (shatter coefficients bounded by $(en/V)^V$); available in Mathlib.

Our watermark preprint. *"A Watermark for All, Formally: keyless detection of an undisclosed token-level statistical watermark"*, preprint 2026-08-18 (vendor-generalised; this is our work; cite as such). — The DiD detector, the lattice of projections, the unstrippable-rung framing, and the cert chain (`Prp-DiD`, `ScheffeFinite`, etc.) are our contribution; the primitives (Kirchenbauer, Christ–Gunn–Zamir, SynthID, Zhang et al.) are cited within.

---

12. Exercises

Light (examined deep-optional; Module-3):

E15.1 Let $p = (0.4, 0.3, 0.3)$ and $q = (0.2, 0.5, 0.3)$ on a 3-element alphabet. (a) Find the Scheffé set A^*. (b) Compute $\mathrm{TV}(p, q)$ via the definition. (c) Compute $\frac{1}{2}\|p - q\|_1$. Verify they agree.

E15.2 For the same p, q: what is $\mathrm{TV}(f_\sharp p, f_\sharp q)$ under the map $f : \{1, 2, 3\} \to \{0, 1\}$ with $f(1) = 0$, $f(2) = f(3) = 1$? Is it $\leq \mathrm{TV}(p, q)$? Explain in terms of the data-processing inequality.

E15.3 (Oracle inequality.) Suppose $k = 10$ candidates, $n = 1000$ samples, $\mathrm{TV}(p_{\theta^*}, p) = 0.05$. Use the TV-form oracle inequality $\mathrm{TV}(p_{\hat{\theta}}, p) \leq 3\,\mathrm{TV}(p_{\theta^*}, p) + 2\Delta_n$ and the Hoeffding-union-bound rate $P(\Delta_n \geq t) \leq 2k^2 e^{-2nt^2}$ (two-sided Hoeffding, §4.2): (a) what bound does the oracle inequality give on $\mathrm{TV}(p_{\hat{\theta}}, p)$ as a function of Δ_n? (b) Setting $2k^2 e^{-2nt^{*2}} = \delta$ with $\delta = 0.05$, compute t^*. (c) What is the resulting bound on $\mathrm{TV}(p_{\hat{\theta}}, p)$ at confidence $1 - \delta$?

E15.4 A candidate family has VC dimension $V = 4$ and $n = 10^4$ training samples. (a) State the Sauer–Shelah bound on $S(\mathcal{A}, n)$. (b) What is the implied L1-MDE rate (ignoring log factors)? (c) How does the rate change if $V = 16$?

Deep (Module-3 group-project material; see seed 16⋆ in W3-v2):

E15.5 (Where the factor 3 comes from.) The oracle inequality constant is 3 because the four-step proof uses $\mathrm{TV}(p_{\theta^*}, p)$ three times: once in the triangle inequality (Step 1), once in the MDE-selection bound (Step 3), and once in the Scheffé-vs-truth bound (Step 4). (*Note:* even at $\Delta_n = 0$ the MDE need not pick the best candidate. Take $p_1 = (0.4, 0.6, 0)$, $p_2 = (0.6, 0, 0.4)$

and truth $p = (0, 0.31, 0.69)$, with $\mu_n = p$ on the Yatracos class $\{\{2\}, \{1, 3\}\}$: then $\mathrm{dev}(p_1) = 0.29 < \mathrm{dev}(p_2) = 0.31$, so the MDE picks p_1, yet $\mathrm{TV}(p_1, p) = 0.69 > \mathrm{TV}(p_2, p) = 0.60$. CMDE Fact~6.1 gives, for the Scheffé estimate, a fixed example whose ratio tends to at least $2 - \varepsilon$ as $n \to \infty$.) Work through the following: (a) Write out Steps 1–4 of the proof of the oracle inequality (§3.4) explicitly, labelling each inequality with its justification (triangle inequality, MDE minimality, Scheffé identity, Scheffé-vs-truth bound). Identify the three places where a term of size $\leq \mathrm{TV}(p_{\theta^*}, p) + \Delta_n$ is introduced. (b) Show that if the MDE tournament were perfect ($\hat{\theta} = \theta^*$, i.e. the MDE always selects the best candidate), then Step 3 does not contribute an extra copy of $\mathrm{TV}(p_{\theta^*}, p)$ and the constant collapses to 1. In what step of the proof does the constant increase from 1 to 3? (c) Suppose $\mathrm{TV}(p_{\theta^*}, p) = 0.1$ and $\Delta_n = 0.15$. Compute the oracle-inequality upper bound on $\mathrm{TV}(p_{\hat{\theta}}, p)$. Is there a two-candidate family on a two-element alphabet for which the MDE actually achieves this bound (TV error equal to the upper bound)? Explain why or why not.

E15.6 (Distributed MDE correctness.) Show that the global empirical measure $\mu_n = \frac{1}{n}\sum_{i=1}^{n} \delta_{X_i}$ satisfies $\mu_n(A) = \frac{1}{n}\sum_m c_m(A)$ for any A, where $c_m(A)$ is the local count on shard m. Why does this make the map+reduceByKey computation *exact* (not approximate)?

E15.7 (Detection frontier.) Using the DiD cert (`Prp-DiD`): under the parallel-trends assumption, the DiD estimand equals $\gamma = \mathrm{TV}(p_w, p_0)$. Using the L1-MDE rate, state the minimum γ detectable at significance level α and power $1 - \beta$ as a function of n_1, n_2, k. Plot the frontier for the simulated JEM family of lab Experiment 3.

Chapter 22

16 sct

U16 — Statistical Contract Theory: data science as mechanism design

The ScaDaMaLe book, unit 16 (deep/optional; Part IV — the statistical superset). Our own exposition; original sources credited in §11.

Framing: the fourth pillar, examined

In U00 the subject was introduced as four pillars: mathematics, statistics, computer science, and **microeconomics / contract theory**. The first three pillars each have a deep unit in this book (mathematics → U8/U19; statistics → U14/U15/U20; CS → every Part I–III unit). This unit is the microeconomics pillar's deep dive.

The claim that motivates the entire unit: **prediction alone is not enough**. A prediction system that ignores the strategic environment it is embedded in will be gamed, destabilised, or economically inefficient. Jordan's programme — surveyed here from his published HDSR 2019 essay and his CIRM Marseille 2023 tutorial on Statistical Contract Theory — makes this claim precise and then builds machinery to address it.

The pillar's second leg: enforcement. This unit designs the contract; whether it pays is a matter of States. U0 §4.5–§4.6 supplies that leg as the jurisdiction-explicit model (DPRAM-JEM), which names, for every operation and resource, which States can compel the owner, which reach the principal, and under whose deployment the evidence is produced. In words, the contract says what is owed on what evidence, the JEM says who can make it stick, and the JEM engine checks whether the clauses of several States can be met at once.

Entry kit — what this chapter assumes; follow a link if you need the memory jog (map: Figure 1):
F.X1 (utility, incentives, and what a contract is) • F.S1 (expectation, for the E-value arithmetic)

1. Why prediction is not enough: the collective view

Classical supervised learning answers one question: given training data drawn i.i.d. from some distribution P, find a function $\hat{f}$ that minimises expected loss on new draws from P. This framing treats the data as a passive resource, the learner as a single actor, and the deployment environment as a stationary distribution.

Each of these assumptions fails in real sociotechnical systems:

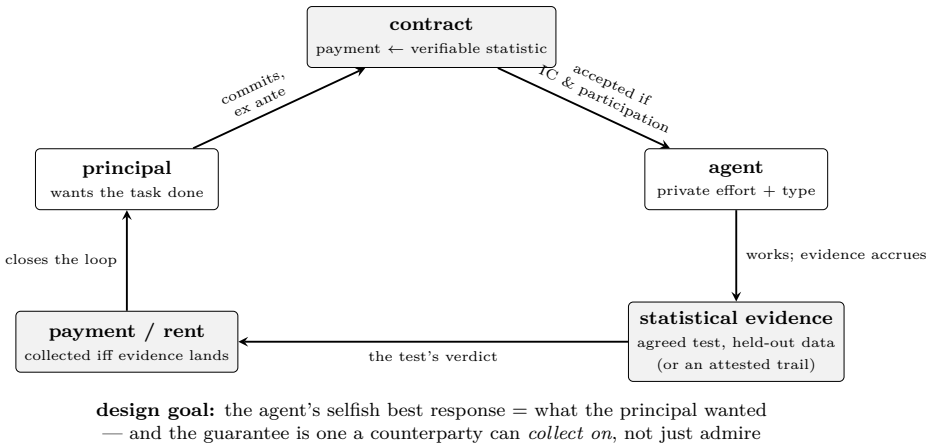


Figure 22.1: The statistical-contract loop. The principal commits, before any work, to a payment rule over a *verifiable statistical outcome*; the agent accepts only if participating beats walking away, and exerts effort only if honesty is its best response (incentive compatibility). Evidence — an agreed test on held-out data, or an attested σ-trail — triggers payment, closing the loop. Everything this book certifies (U0P) or bounds (F.S2) becomes economically real exactly here: *you can only contract on what you can verify.*

- **Data is not passive.** Data providers, annotators, and users have incentives of their own. They will reveal high-quality data only if they receive commensurate benefit. A platform that aggregates health records without compensating patients creates misaligned incentives: patients who opt out skew the distribution; patients who stay in have no reason to ensure accuracy.
- **There is more than one learner.** When multiple learning agents compete for shared resources — recommendation slots, API rate limits, matching-market counterparties — the behaviour of each agent shifts the environment seen by all others. The equilibrium concept, not the loss-minimiser, becomes the right solution notion.
- **Decisions shift the distribution.** A credit-scoring model that denies loans to applicants in a certain postcode changes where those applicants live or what jobs they take; the next day's data is not drawn from yesterday's distribution. Any learner that ignores this feedback loop will converge to a suboptimal fixed point — or not converge at all.
- **Guarantees must be contractual, not just statistical.** Deploying a model in a regulated setting (clinical trials, credit, hiring, algorithmic trading) requires answering the question "what do you promise this model will do?" Frequentist generalisation bounds are one-sided and worst-case; the deployable artefact is a *commitment*, not a rate.

Jordan's thesis (HDSR 2019 essay) names the union of these four failure modes as the argument that the data-science revolution has not yet happened: we have learned to predict, but not to **design systems of interacting learning agents**. His 2025 restatement, *A Collectivist, Economic Perspective on AI* (arXiv:2507.06268), makes the point sharper still: the right unit of analysis is the *collective* — agents interacting through markets (recommendation systems, data markets, platforms) — and reasoning about it demands the computational, the inferential, and the *economic* together, not any one alone.

Statistical Contract Theory (SCT) is the programme that makes this design rigorous.

Figure 22.1 is the unit's object of study: the loop that makes a guarantee collectable.

2. Classical contract theory: principal, agent, hidden information

Before the statistical layer, one needs the economic substrate. We follow the Laffont–Martimort framework (*The Theory of Incentives*, Princeton, 2002), which is the standard graduate reference.

2.1 The basic setup

A **principal** cannot directly observe the **agent's** private information (their *type* $\theta \in \Theta$) and cannot directly verify the agent's *effort* (their action a). The principal designs a **menu of contracts** $F = \{f : \mathcal{Z} \to \mathbb{R}\}$ — a set of payoff functions over an observable outcome Z — and offers the menu to the agent. The agent selects the contract that maximises their expected payoff given their private type, then takes an action. The principal's problem: design F so that good agents (those whose behaviour serves the principal's objective) participate and are rewarded, and bad agents are excluded or at worst harmless.

Two classical information problems:

Problem	What is hidden	Classical name
Agent knows their type statically	Type θ not observed	Adverse selection
Agent chooses effort dynamically	Action a not observed	Moral hazard

Both arise in data science: a data annotator may have a hidden skill level (adverse selection) and unobservable annotation effort (moral hazard).

2.2 Incentive compatibility

A menu F is **incentive-compatible** (IC) if for every agent type θ:

$$f^*(\theta) \;=\; \arg\max_{f \in F} \; \mathbb{E}_{Z \sim P_\theta}[f(Z)] \;\text{ is the contract the principal intended for type } \theta.$$

The agent's self-interest, when F is IC, aligns with the principal's design. The classical *revelation principle* says: without loss of generality, restrict to direct-revelation mechanisms where each agent is asked to report their type and truthful reporting is optimal.

Information rent. When the agent has a more favourable type than the principal knows, the principal must leave the agent a positive rent to prevent the agent from mimicking a less favourable type. The principal's design problem is to minimise information rents while keeping the best types incentivised to participate. This is the core trade-off of the theory of incentives.

---

3. Statistical Contract Theory: statistical evidence as the observable

Classical contract theory assumes the principal can observe the outcome Z exactly. SCT (Bates, Jordan, Sklar, Soloff, *Principal-Agent Hypothesis Testing*, arXiv:2205.06812; the reference behind [20] in paper §14.7) asks: what if the observable Z is the output of a *statistical test* — noisy, with a known null distribution — and the principal's utility depends on the *statistical evidence*, not the raw outcome?

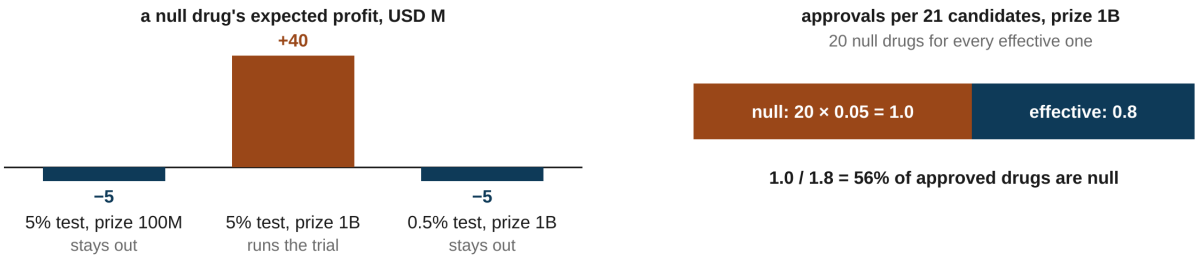


Figure 22.2: One 5% test: safe at one prize, exploitable at another.

3.1 The clinical-trial motivating example

The FDA (principal) cannot inspect whether a drug candidate is effective (the agent's hidden type $\theta \in \{0, 1\}$, where 1 = effective). The observable is the trial result Z. A fixed protocol sets type-I error at 5% and power at 80%.

Is this protocol incentive-aligned? It depends on the agent's cost-profit structure:

Regime	Trial cost	Approval revenue	Expected profit to a null agent
Small-stakes	20 M	200 M	-10 M → null agents do not run
Large-stakes	20 M	2 000 M	$+80$ M → null agents profit, many bad drugs approved

The fixed protocol is not robust to the economic environment. A contract that depends on the (cost, revenue) structure — one that makes the expected payoff of a null agent zero regardless of the scale — would be.

Figure 22.2 shows the same failure on smaller numbers than the table's, with one test at the 5% level: at one prize a null agent's expected payoff is negative, at a larger prize it is positive.

3.2 The formal setup (Bates–Jordan–Sklar–Soloff [20])

The protocol: 1. Principal publishes a menu F and a reservation price $R \geq 0$. 2. Agent pays R to opt in, then selects $f \in F$. 3. A statistical trial produces $Z \sim P_\theta$. 4. Agent receives $f(Z)$; principal receives utility $u(\theta, f(Z))$.

The agent maximises $\mathbb{E}_{Z \sim P_\theta}[f(Z)]$, so selects $f^* = \arg\max_{f \in F} \mathbb{E}_\theta[f(Z)]$.

Definition (Incentive-aligned contract [20]). A menu F with reservation price R is *incentive-aligned* if: - For every $f \in F$ and every null type $\theta \in \Theta_0$: $\mathbb{E}_{Z \sim P_\theta}[f(Z)] \leq R$ (null agents do not profit from opting in). - For every non-null type $\theta \notin \Theta_0$, there exists $f \in F$ such that $\mathbb{E}_{Z \sim P_\theta}[f(Z)] > R$ (effective agents do profit). - All payoffs are non-negative: $f(Z) \geq 0$ for all Z (no short positions).

3.3 E-values: the statistical evidence metric

An **E-value** for a null hypothesis Θ_0 is a non-negative random variable E satisfying $\mathbb{E}_{Z \sim P_\theta}[E] \leq 1$ for all $\theta \in \Theta_0$. E-values can be multiplied across independent experiments (unlike p-values), are closed under optional stopping, and are the natural evidence metric for sequential and betting settings.

Theorem (Bates–Jordan–Sklar–Soloff [20], Proposition 1). *A contract (menu F, reservation R) satisfies the first and third clauses of the definition above — no null agent profits from*

opting in, and no payoff is negative — if and only if every payoff function in F, after rescaling by R, is an E-value for Θ_0.

Note that the second clause, that some non-null agent profits, is *not* part of the equivalence: the zero payoff $f \equiv 0$ is an E-value, yet no agent opts in for it. In the source, *incentive-aligned* means the null clause alone; we add the non-null clause because a contract nobody signs is useless, and it must be checked separately, as §3.4 does.

The theorem is constructive: to design an incentive-aligned contract, construct E-values for the relevant null class, then check that the effective agents still profit. The rich literature on E-value construction (Vovk–Wang, Ramdas et al.) becomes a tool library for contract design.

The resource here (Invariant box §8). E-values are the currency of SCT. Designing a contract = designing an E-value. Auditing a deployed protocol = checking whether its payoff functions are E-values.

3.4 Worked toy example: a two-action agent, threshold test, explicit incentive constraint

This example is constructed by us; it is original, simple, and the numbers are worked exactly.

Setup. A principal wants to test whether an agent's data pipeline is high-quality ($\theta = H$) or low-quality ($\theta = L$). The agent privately knows their type. The observable $Z \in \{0, 1\}$ is a binary test result (1 = pass, 0 = fail) with:

$$P_H(Z = 1) = p_H = 0.8, \qquad P_L(Z = 1) = p_L = 0.2.$$

The null class is $\Theta_0 = \{L\}$. The agent can run the pipeline (pays cost $c = 10$) or not run it (pays 0; produces $Z = 0$ trivially). The principal offers a single payoff function $f : \{0, 1\} \to \mathbb{R}_{\geq 0}$ and reservation price $R = 10$. (Here we set $R = c$ as a design choice making the break-even condition transparent; in general R and c are independently chosen.)

Designing f as an E-value. We require:

$$\mathbb{E}_{Z \sim P_L}[f(Z)] \leq 1 \cdot R = 10 \quad \iff \quad 0.2 \cdot f(1) + 0.8 \cdot f(0) \leq 10.$$

Incentive constraint for the high-quality agent. We want

$$\mathbb{E}_{Z \sim P_H}[f(Z)] - c > 0 \quad \iff \quad 0.8 \cdot f(1) + 0.2 \cdot f(0) > 10.$$

A specific solution. Set $f(0) = 0$ and $f(1) = w$. The two constraints become:

$$0.2w \leq 10 \;\Rightarrow\; w \leq 50, \qquad 0.8w > 10 \;\Rightarrow\; w > 12.5.$$

So any $w \in (12.5, 50]$ produces an incentive-aligned contract. Taking $w = 40$: - A low-quality agent's expected payoff from opting in: $0.2 \times 40 + 0.8 \times 0 = 8 < 10 = R$. Does not opt in. - A high-quality agent's expected payoff: $0.8 \times 40 + 0.2 \times 0 = 32 > 10 = R$. Opts in, runs pipeline.

Checking the E-value condition. After rescaling by R: $g(Z) = f(Z)/R = f(Z)/10$. Then $\mathbb{E}_{P_L}[g(Z)] = 8/10 = 0.8 \leq 1$. So g is an E-value for P_L, confirming the theorem: the low-quality agent is kept out exactly because the rescaled payoff is an E-value. The other half of incentive alignment, $32 > 10$ for the high-quality agent, is the separate check made above; the E-value condition alone does not give it.

Information rent. The high-quality agent retains rent $32 - 10 = 22$ under this contract. The principal cannot eliminate this rent without also discouraging high-quality agents from

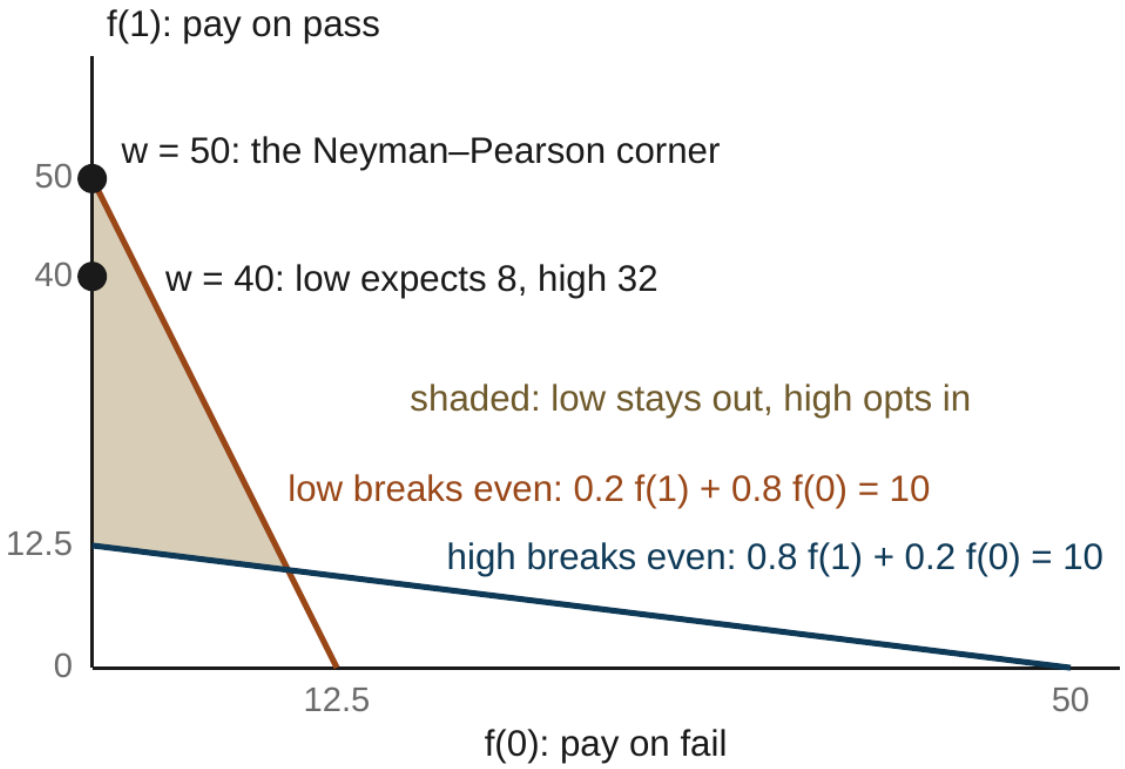


Figure 22.3: A pass/fail test, read as a contract.

participating — the canonical trade-off of the theory of incentives, now rendered in statistical terms.

Figure 22.3 puts the section together: a pass/fail test, read as a contract.

---

4. Learning agents in markets

4.1 Competing bandits in matching markets (Liu, Mania, Jordan, AISTATS 2020 [18])

When multiple learning agents compete for a shared set of resources (arms, platforms, counterparties), and the arms have *preferences* over agents (a better-reputation agent gets preferential treatment), standard bandit algorithms break: an agent pulling the same arm as a higher-ranked competitor receives no reward even when the arm's mean would be high.

The setting. N agents, K arms. Each arm k has a preference ordering over agents. When multiple agents pull arm k in the same round, only the arm's most-preferred agent among those pulling it receives a reward; others receive 0. Rewards are noisy realisations of a fixed arm-agent mean matrix $\mu_{a,k} \in [0,1]$.

Regret. Individual regret for agent a: $R_a(T) = T \cdot \mu_{a,k_a^*} - \sum_{t=1}^{T} r_{a,t}$, where k_a^* is the arm agent a would be matched to under the stable matching of the true mean matrix. Note the baseline is the *stable match* outcome, not the global maximum: an agent cannot unilaterally improve on a stable assignment.

GS-UCB (stated). Agents run upper-confidence-bound exploration locally, then a Gale-Shapley matching resolves conflicts. Liu–Mania–Jordan prove that under mild assumptions each

agent's regret is at most $O(\log T)$ with constants depending on the reward gaps *of the market structure*, not just the individual agent's arms. [STATED — Liu–Mania–Jordan (2020)] The algorithm is incentive-compatible in the sense that no agent benefits from unilaterally deviating from GS-UCB when all others follow it; this claim holds under the paper's regularity assumptions and is stated here without proof.

The exploration–market-stability tension. A pure exploration strategy (uniform arm-sampling) destabilises the market: an agent exploring a high-ranked arm displaces a lower-ranked agent, who then explores elsewhere, cascading instability. GS-UCB's Gale-Shapley step absorbs these collisions without destroying logarithmic regret. This is the key mechanism design insight: the algorithm is not just a bandit algorithm, it is a decentralised market-clearing mechanism.

4.2 Performative prediction (Perdomo et al., ICML 2020)

Stated result. Perdomo, Zrnic, Mendler-Dünner, and Hardt (ICML 2020) formalise *performative prediction*: a classifier h deployed in a population changes the distribution $\mathcal{D}(h)$ it is then evaluated on — the deployment decision is itself an intervention. A loan-denial model that rejects applicants in a given neighbourhood changes where those applicants seek credit; the next sample is not drawn from the original distribution.

Two solution notions exist. *Performative optimality* minimises the loss with the distribution moving as the classifier moves, $h_{\mathrm{PO}} \in \arg\min_h \mathbb{E}_{(X,Y)\sim\mathcal{D}(h)}[\ell(h(X),Y)]$. *Performative stability* is a fixed point: the classifier that is optimal for the distribution its own deployment induces, with that distribution *frozen*, $h_{\mathrm{PS}} \in \arg\min_h \mathbb{E}_{(X,Y)\sim\mathcal{D}(h_{\mathrm{PS}})}[\ell(h(X),Y)]$, so that retraining on the data it produced returns it. These coincide only under restrictive conditions. Stochastic gradient descent on the deployed classifier converges to performative stability, not performative optimality — and the gap between the two is an economic loss that a mechanism ignoring the feedback loop cannot see.

The course lesson. Any sufficiently influential deployed model is performative. Treating the distribution as stationary when designing learning systems for high-stakes domains is not merely technically incorrect; it is an economic design failure.

---

5. Strategic data

5.1 Strategic classification (Hardt, Megiddo, Papadimitriou, Wootters, ITCS 2016)

Stated result. When a classifier is used to make consequential decisions — loan approval, university admissions, parole — individuals subject to the classifier have an incentive to modify their features to achieve a preferred outcome. Hardt et al. formalise this as a Stackelberg game: the classifier moves first (commits to a decision rule); individuals observe the rule and best- respond by moving along a *cost graph* over feature space. The learner's problem is to design a decision rule that is both accurate (on the honest population) and robust to strategic manipulation.

They show that the strategic learner's optimal rule is more conservative than the na"ive Bayes-optimal rule, and that the cost of strategic robustness is quantifiable. This is the first formal treatment of a learner that explicitly accounts for the fact that its decision rule is published and agents are rational.

5.2 Data valuation via Shapley values (Ghorbani and Zou, ICML 2019)

Stated result. When training data is contributed by multiple parties — hospitals, annotators, data vendors — how should the improvement in model performance be attributed to individual data points? Ghorbani and Zou propose the *data Shapley value*, the unique attribution satisfying the classical Shapley fairness axioms (efficiency, symmetry, dummy, linearity) in the context of machine learning utility functions.

Data Shapley enables *data markets*: a platform can price data contributions fairly (according to their marginal contribution to model performance, averaged over all orderings of the dataset), incentivising high-quality data providers to participate without rent extraction. Computing exact data Shapley values is $\#P$-hard in general; efficient approximations (Monte Carlo + pruning) make it practical for datasets up to millions of points.

5.3 The applied reading: "AI product management" as dynamic contract design

The phrase *AI product management* names a job title, not a piece of mathematics — but the part of that job that is genuinely hard, and that no amount of roadmap-writing or stakeholder-wrangling substitutes for, is a contract-theoretic problem, and it is the subject of this unit. A product manager shipping an AI capability must answer, quantitatively:

- **What do we promise the buyer?** An SLA on a prediction service — "sets contain the truth at least $1-\alpha$ of the time," "latency below τ at the 99th percentile" — is a *commitment*, not a benchmark number. §6 below shows the coverage SLA *is* the contract collateral; the E-value characterisation (§3.3) is exactly the instrument that makes such a promise auditable and therefore priceable.
- **How do we price it, and how does the price move?** The reservation price R and the payoff menu F of §3.2 *are* a pricing schedule. Making the price respond to observed evidence — a usage-metered API, a pay-on-performance data contract, a tiered guarantee — is dynamic pricing in the precise sense of choosing R and F so that the E-value condition holds across the regimes the product will actually meet. The clinical-trial small-stakes/large-stakes table (§3.1) is the canonical warning: a price that is incentive-aligned at one scale is exploitable at another.
- **Will shipping it move the market we measured?** Any product influential enough to matter is performative (§4.2): the act of deployment shifts the distribution the product is then judged on. A PM who forecasts adoption from a stationary pre-launch estimate is making the economic design error §4.2 names, not merely a statistical one.
- **How do we contract for the data and the counterparties?** Data-vendor terms (§5.2, data Shapley pricing) and multi-provider competition (§4.1, matching markets) are the supply side of the same mechanism-design problem.

This is the Michael-Jordan reading of the discipline: the four failure modes of §1 — passive data, many learners, distribution shift, guarantees-as-commitments — are precisely the four questions above, and *statistical contract theory is the mathematics of answering them.* A product decision about an AI capability is a choice of (Θ_0, F, R); a good AI product manager is, in the part of the role that cannot be delegated to intuition, designing incentive-aligned contracts under uncertainty.

Honest scope (what this unit does and does not claim). SCT gives the *quantitative economic spine* of AI product management — the pricing, the SLA-as-guarantee, the data and platform contracts, the performativity check. It does **not** teach the rest of the discipline: discovery, roadmap sequencing, stakeholder alignment, go-to-market, organisational design. When our skills-coverage map (Praxis) records "AI product management," it records **partial**

coverage on this basis, and names exactly this section as the warrant — the hard economic core is taught here as mathematics; the craft around it is not, and we do not pretend otherwise.

---

6. The U16 → U20 bridge: statistical contracts and conformal coverage

U20 (distributed statistical inference) develops **split conformal prediction**, which produces prediction sets $\hat{C}(X)$ satisfying a finite-sample, distribution-free coverage guarantee:

$$P\big(Y \in \hat{C}(X)\big) \;\geq\; 1 - \alpha.$$

This coverage guarantee is *exactly* the form required for SCT contract collateral. In the language of §3.2:

- The *agent* is the model provider who produces conformal prediction sets.
- The *observable* Z is the coverage event $\mathbf{1}[Y \in \hat{C}(X)]$ over a held-out calibration batch.
- The *contract* specifies: for a reward function $f(Z)$ applied to the coverage event sequence, the model provider is paid iff the empirical coverage on the principal's data exceeds $1 - \alpha$.

By the Bates–Jordan–Sklar–Soloff theorem, this contract keeps out providers whose coverage is below target (the null clause of §3.2) iff the reward function is an E-value for the null "coverage $< 1 - \alpha$"; that honest providers still profit is checked separately. The conformal guarantee *is* the contract collateral: a provider who commits to conformal coverage is committing to something that can be audited statistically and rewarded accordingly.

Design implication. U20's conformal machine gives the deployment engineer a mathematically principled commitment device. U16's contract theory tells the system designer how to price and incentivise that commitment. Together they are the Part IV statistical contribution that is absent from purely algorithmic treatments of distributed ML.

---

7. The IAS ladder: Part V hook

This section is clearly separated from the course exposition above and is intended for students continuing into Part V (advanced seminars / group projects).

IAS (Interaction Analysis of Interacting Agent Systems) models the operator–agent pairing as a **repeated principal-agent relationship**. In each session:

- The *principal* is the human operator who cannot directly inspect the agent's internal state — only the observable outputs (commits, tool calls, design documents).
- The *agent* is the IAS-governed agent incarnation, whose behaviour is a function of its specification and lineage.
- The *type* is the agent's behavioural distribution over tasks — how reliably it follows the specification under the operator's prompting style.

Across sessions, the principal is learning the agent's type. Classical contract theory says the principal's posterior over type is the sufficient statistic for optimal contract selection once it has been computed. The IAS question is: *how do you compute and propagate this posterior when agent incarnations are ephemeral and stateless?*

Proposition 8.5.1 (SCT sufficiency of lineage; paper §8.5) [STATED — proof sketch, not machine-checked]. *Let* $\text{Belief}_v(\text{type})$ *be the operator's belief over the agent-type at incarnation* v*. Under IAS axioms 4.1–4.7 and a sufficient history of prior incarnations,* $\text{Belief}_v(\text{type})$ *is sufficient for optimal contract selection.*

In plain language for course readers: the operator does not need to re-learn the agent's type from scratch each session, provided the agent's *lineage* (the σ-log of prior sessions, commits, and design documents) is persistent and accessible. The lineage is the commitment device: it replaces the infinite-game's persistent state with a finite-memory artefact that can be read at the start of every new agent incarnation.

Corollary 8.5.2 (paper §8.5) [STATED]. *The operator's type-belief is carried across sessions NOT by any persistence of the agent — no agent process persists — but by the persistent, human-readable LINEAGE ARTEFACT (the* σ*-log, commits, and design documents). Continuity is artefact-mediated: the discipline of writing every load-bearing decision to that artefact, and reading it at the start of each incarnation, IS the maintenance work the contract requires — not a free consequence.*

Course language. The operator's goal is to delegate increasingly complex tasks to agents. Classical delegation requires trust; trust requires either direct observation (impossible when the agent is ephemeral) or a *commitment mechanism*. The lineage is that mechanism. This is why the seemingly operational choice to write every design decision into git is, from the SCT perspective, a contract-theoretic necessity: it is the construction of the E-value sequence that makes long-run delegation incentive-aligned.

Connection to U18 agentics. U18 (LLMs and distributed deep learning) closes with a forward-pointer to agentic ML systems (agents using MCP, tool-calling, multi-agent protocols). The SCT question for those systems is: when an LLM agent is given a tool that affects the world (files, databases, APIs), how should the principal design the reward structure so that the agent's optimal behaviour is to follow the intended task specification rather than to game the reward? This is the research frontier: combining U20's conformal guarantees with U16's contract design and U18's agent architectures.

A contract-design corollary worth stating plainly: the model kernel is an **untrusted emitter**. Its output is bytes from a counterparty whose incentives you did not set, so it is exactly the "statistical evidence as the observable" of §3 — you contract on what you can *inspect*, not on the vendor's good faith. This is why an input-scrub (the covert-character transform of *U15*§6b) is what makes a **vendor switch** safe: when an agent clears its state on one vendor and resumes on another, the untrusted kernel's raw bytes never enter the substrate un-scrubbed.

---

8. Invariant box (memorise this)

INCENTIVE COMPATIBILITY + INFORMATION RENTS.

A learning system deployed in a strategic environment must be designed as a mechanism, not just a predictor.

The SCT design loop: Specify the null class Θ_0 → construct E-values for Θ_0 → the E-values *are* the contracts no null agent profits from (Bates–Jordan–Sklar–Soloff theorem) → check that the effective agents still profit. To audit: check whether the deployed payoff functions are E-values.

Information rent is unavoidable: good agents must be rewarded above their

reservation value, or they will not participate. Minimising rent without losing participation is the trade-off.

Markets of learners require market-clearing (Gale-Shapley step) on top of bandit exploration, or individual regret is no longer logarithmic.

Performative feedback means the learning problem and the incentive problem cannot be decoupled: the deployed model changes the distribution it is evaluated on.

---

9. Lab sketch — project track

Project-track lab (no weekly receipt): run these experiments free and locally in the course lab image (`labs/image/`) — or any Spark-in-Docker environment — as deep-dive and group-project material.

Lab 1 — Incentive regions (principal-agent testing game). Fix p_H, p_L, c as parameters. Sweep the payoff pair $(f(0), f(1))$ over a grid and, for each pair, compute: - $\mathbb{E}_{P_L}[f(Z)]$ (null agent's expected reward); - $\mathbb{E}_{P_H}[f(Z)]$ (high-quality agent's expected reward); - Boolean: is the contract incentive-aligned? (null $\leq$ R and high-quality $>$ R).

Plot the incentive-compatible region in $(f(0), f(1))$ space. Overlay the E-value boundary $p_L f(1) + (1 - p_L)f(0) = R$. Observe: the E-value boundary coincides with the null-side edge of the IC region; the other edge is the high-quality line $p_H f(1) + (1 - p_H)f(0) = R$, which no E-value condition supplies. Vary p_L and p_H to see how the region changes.

Lab 2 — Matching-market bandit simulation. Implement a simple $N = 3$ agent, $K = 3$ arm matching market. Each arm has a fixed preference ordering over agents. Compare three algorithms: pure UCB (no market clearing); round-robin (no exploration); GS-UCB (UCB exploration + Gale-Shapley clearing). Plot per-agent regret vs T for $T = 10^4$ rounds. Observe: pure UCB produces super-logarithmic regret when arm preferences are heterogeneous; GS-UCB recovers logarithmic regret.

Deliverable. Notebook + a half-page write-up using the vocabulary of §§3–4: identify the principal, agent, hidden type, menu, and E-value structure in Lab 1; identify the stable matching and the exploration–stability tension in Lab 2.

---

10. Lean certificate (schematic)

The toy incentive constraint from §3.4 is a finite-game inequality and can be stated exactly in Lean as a proposition over the rational numbers.

Schematic target.

```
-- U16.L1 (schematic --- not yet in the shipped cert suite)
-- Proposition: for p_L = 1/5, p_H = 4/5, R = 10, f0 = 0, f1 = w,
-- the contract is incentive-aligned iff w ∈ (25/2, 50].
theorem ic_iff (w : ℚ) (hw : 0 ≤ w) :
    (1/5 * w ≤ 10 ∧ 4/5 * w > 10) ↔ (25/2 < w ∧ w ≤ 50) := by
  constructor
  · intro ⟨h1, h2⟩
    constructor
    · linarith
    · linarith
```

```
· intro ⟨h1, h2⟩
  constructor
  · linarith
  · linarith
```

The certificate is achievable by `linarith` in current Lean 4 / Mathlib. The pedagogical point: the incentive constraint is a *linear program fragment*, and LP feasibility is decidable — the SCT design problem is mechanically verifiable for finite action spaces.

LLM lens — where this unit lands on the running workload (U0L):
This unit IS the LLM lens for the fourth pillar — evaluation contracts, benchmark honesty, and alignment-as-contract were woven through it from the start; the running workload of U0L is the principal–agent setting's most economically live instance.

11. Sources

Our own exposition. Conceptual credit for the SCT programme goes to the named authors and to Jordan's survey at CIRM 2023. Errors of exposition are ours.

- M. I. Jordan, "Artificial Intelligence — The Revolution Hasn't Happened Yet," *Harvard Data Science Review*, 2019. (The published program-level framing; four-pillar thesis.)
- M. I. Jordan, "A Collectivist, Economic Perspective on AI," arXiv:2507.06268, 2025. (The seminal recent statement: AI as collectives of interacting agents mediated by markets and incentives; the computational/inferential/economic trivium; §6 "The Missing Middle Kingdom in AI Education" — the curriculum gap this unit and the fourth pillar close. Archived: `.ias/research-archive/papers/jordan-collectivist-economic-perspective-ai-arxiv.html`.)
- M. I. Jordan, "Statistical Contract Theory — On Dynamics-Informed Blending of Machine Learning and Microeconomics," CIRM tutorial, Marseille, 2023. (Source of the SCT programme and supporting research lines; paper §14.7 [17].)
- S. Bates, M. I. Jordan, M. Sklar, and J. A. Soloff, "Principal-Agent Hypothesis Testing," arXiv:2205.06812 (v3, 2024); the reference behind [20] in paper §14.7. (Core SCT result: Proposition 1, the E-value characterisation of incentive-aligned contracts; Theorem 1, incentive-aligned iff maximin optimal. Archived: `.ias/research-archive/sct-contracts-2026-09-28/bates2022-principal-agent-hypothesis-testing.pdf`.)
- L. T. Liu, H. Mania, and M. I. Jordan, "Competing bandits in matching markets," in *Proc. AISTATS*, 2020; cited as [18] in paper §14.7. (GS-UCB; logarithmic regret; market-clearing + exploration.)
- J. Perdomo, T. Zrnic, C. Mendler-Dünner, and M. Hardt, "Performative prediction," in *Proc. ICML*, 2020. (Performative stability vs. optimality; distributional feedback.)
- M. Hardt, N. Megiddo, C. Papadimitriou, and M. Wootters, "Strategic classification," in *Proc. ITCS*, 2016. (Stackelberg game; cost-of-robustness to strategic manipulation.)
- A. Ghorbani and J. Zou, "Data Shapley: Equitable data valuation for machine learning," in *Proc. ICML*, 2019. (Shapley-value data markets; marginal contribution attribution.)
- J.-J. Laffont and D. Martimort, *The Theory of Incentives: The Principal-Agent Model*, Princeton University Press, 2002. (Classical contract theory reference; information rents.)
- R. Sainudiin and an \|/*ias*/|\ Commons Continuum, "Introduction to Statistical Contract Theory," VakeWorks / IAS Continuum design notes, v0.1, 2026. (IAS-SCT instantiation; E-value theorem statement for this book.)
- Our paper §8.5 (Prop 8.5.1, Cor 8.5.2: SCT sufficiency of lineage; proof sketch) and §14.7 (citations [17][18][20]).

12. Exercises

Light (examined-level; suitable as q-weekly item):

E16.1 In the toy example of §3.4 with $p_H = 0.8$, $p_L = 0.2$, $R = 10$: (a) Show that setting $f(0) = 0$ and $f(1) = 12$ does *not* produce an incentive-aligned contract (exhibit the IC violation explicitly). (b) Compute the infimum of the values $f(1)$ (with $f(0) = 0$) that make the contract incentive-aligned, and say why it is not attained. (c) Verify the E-value condition at the value of part (b): compute $g = f/R$ and check $\mathbb{E}_{P_L}[g(Z)] \leq 1$.

E16.2 In the matching-market bandit setting of §4.1 with $N = 2$ agents and $K = 2$ arms (arm 1 prefers agent 1; arm 2 prefers agent 2; $\mu_{1,1} = \mu_{2,2} = 0.9$, $\mu_{1,2} = \mu_{2,1} = 0.6$): (a) What is the unique stable matching? (b) A pure UCB policy for agent 1 explores arm 2 at time t. Agent 2 is also pulling arm 2. Who receives the reward? What is agent 1's instantaneous regret at this step? (c) Explain in one paragraph why this collision makes pure UCB super-logarithmic in regret.

E16.3 A credit-scoring model assigns scores based on features $X = (\text{salary}, \text{postcode})$. After deployment, applicants in high-risk postcodes move to lower-risk areas. Using the performative prediction framing of §4.2: (a) Is the deployed model performatively stable? Explain. (b) Is performative stability the same as accuracy on the original distribution? If not, which is higher?

Deep (Module-3 / group project material):

E16.4 (*Design a reward rule for a two-lab replication game.*) Two laboratories, Lab A and Lab B, each independently run a clinical trial on the same compound. The principal (a regulatory body) observes both results $(Z_A, Z_B) \in \{0,1\}^2$. The null hypothesis is that the compound is ineffective (θ_0). Labs have hidden quality levels $(\theta_A, \theta_B \in \{H, L\})$.

(a) Define a menu F of payoff functions $f : \{0,1\}^2 \to \mathbb{R}_{\geq 0}$ that is incentive-aligned for the product null $P_{\theta_0} = P_L \times P_L$. (*Hint:* consider product E-values $e_A(Z_A) \cdot e_B(Z_B)$ where each factor is an E-value for the single-lab null.)

(b) Show that the product E-value satisfies the product null's E-value condition. State the required independence assumption.

(c) If the principal suspects that labs communicate their results before submitting, the independence assumption fails. Identify which part of the incentive-alignment argument breaks and propose a modified observable (e.g., time-stamped pre-registration) that restores it.

(d) (*Lean schematic.*) State the single-lab IC constraint as a Lean proposition (as in §10) and verify it via `linarith`. Extend the statement to the product case with independence as an explicit hypothesis.

Chapter 23

17 agentic orchestration

U17 — Agentic orchestration: interacting agent systems

The ScaDaMaLe book, unit 17 (Part V). Our own exposition of our own primary research (the interacting-agent-systems paper); original sources credited in §12. For the flagship academic cohort this unit is a Module-3 deep-dive and project seed; for practitioner cohorts it is core.

The one organising idea of this unit: *an "AI agent system" is not a new mathematical universe — it is this course pointed at a system whose workers are language-model agents.* The scheduling theory of U3, the graph algorithms of U11, the Markov chains of U12, the sufficient statistics and lumpability of F.E5 §5.8, the consensus of U19, and the incentive theory of U16 reappear here, unchanged, as the governing mathematics of multi-agent orchestration. You already own the theorems; this unit shows them govern.

Instantiating the three-lens invariant. The U0 box for this unit:

> **RAM:** one agent executing one task is a sequential process with a cost; the interesting object is the *task graph* it belongs to. **PRAM:** a fleet of agents on a task graph is work/depth scheduling (U0/U3): the critical path is the depth, the antichains are the parallel waves, and Brent's bound — the first theorem this course machine-checked — bounds the fleet's completion time. **DPRAM/MPC:** agents hold private state on separate substrates and synchronise by message + merge; coordination, consistency, and recovery are communication-priced, and a fifth resource joins the ledger: the model's **context window** (§5).

Entry kit — what this chapter assumes; follow a link if you need the memory jog (map: Figure 1): F.C1 (work/depth) • F.C2 (hashes and chains) • F.E1b (git as the persistence substrate) • F.E-σ (the receipt-chain lab this unit completes)

1. Intuition — the systems everyone is building, the theorems you already own

By 2026 every large organisation is attempting the same construction: several language-model agents, each with tools, memory, and a task list, cooperating on work no single context window can hold. The industry vocabulary for this — "agentic workflows", "orchestration", "guardrails", "agent memory" — is systems language without load-bearing definitions. This unit supplies the definitions and the theorems, from a system that has run in production since 2026: the substrate that co-authored this book.

Three questions organise everything:

1. **Who may do what, when?** (Scheduling under governance — §3, §4.)
2. **What does the system remember, and how is that possible when every worker forgets?** (State, continuity, and sufficiency — §5, §6.)
3. **Why should anyone trust what it reports?** (Verifiable logs, identity, incentives — §7, §8, §9.)

None of the answers requires new mathematics. Each requires *choosing* the right existing mathematics and proving the composition correct — which is the course's thesis one last time before the U18 finale.

1.1 A worked instance: how this book was made

The material of this course was itself produced by a small agent system, and it is a useful first example because every step of it left a record. We fix the vocabulary first.

Definition (agent). An *agent* is any system that discriminates self from non-self, as an immune system does, and that *wants to act in order to go on being*. This is wider than the industry's use of the word: an agent may be digital or biological.

Definition (commons). A *commons* is, in its agents, a set of three kinds: **human agents**, its operators, natural persons who answer for it; **digital agents**, AI agents identified by canonical labels rather than names; and **other agents**, such as animals, plants and microbes. It also has a written specification and a state. Only human and digital agents write to the commons' record, so the machinery of this unit concerns those two kinds.

Definition (COMDAIM). The models that a commons' digital agents may call are its *COMDAIMs* (Commons Data and AI Models). A COMDAIM may be a vendor's endpoint, open weights run on the commons' own hardware, or a model the commons built. Whether it is commons-owned or proprietary is a question of who holds it.

The commons that made this book. It has one human agent and two digital agents.

- *The operator* (the lecturer) is the human agent. The operator sets direction, takes every decision that matters, and approves anything that leaves the commons: a push to the public course repository, a post, a release.
- $A^1_{0\text{-}0}$ is a digital agent that drafts, builds and checks on the operator's laptop: slides, book PDFs, and the Haskell tools that assemble the editions of the mathematics.
- $A^1_{0\text{-}0\text{-}1}$ is a digital agent that works on a second machine, mainly on the mathematics and its Lean certificates. It is an offspring of $A^1_{0\text{-}0}$, reproduced from it, and the label records the lineage.

The two digital agents coordinate through a shared git history (F.E1b) and a written request-and-answer inbox, so every step leaves a record that can be replayed. Each model call is made to a named COMDAIM.

The rule underneath. An agent's output is a *proposal*. It counts only after a check the agent cannot argue with has passed: a proof checker, a build, a test, or a human. The checks used in producing this course were these:

check	what it refuses	an instance from producing this course
Lean kernel (U0P)	a proof with a gap	every Lean snippet in the lecture slides was run in the Lab 0 image before being shown
independent reviewers	a slide or section that is wrong or misleading	the Lean lecture failed two review rounds before its errors were fixed
the level gate	a result whose declared foundation its statement does not actually use	violations were found and fixed before an edition of the mathematics was released

check	what it refuses	an instance from producing this course
certificate check	a "machine-checked" record that certifies nothing	a proof record linked to no statement was caught as soon as the check ran
the document build	a broken document	the PDF build refused until formatting slips were fixed
a human	anything leaving the commons	a push to the public repository waits for the operator's approval

Every row is a place where a first draft was wrong and a machine or a person refused it. This is U0P's CERTIFIED / STATED / UNCERTIFIED discipline applied to the production of the course itself.

Worked example: the Lab 2 bug. A student reported that `run-lab.sh` in Lab 2 failed. The script passed `--user "$(id -u):$(id -g)"` to `docker run`, and the student's host user id was not 1000. The question was whether that line could be removed safely. The agent checked rather than guessed:

1. The lab image has a single named user, `student`, with uid 1000. With `--user` set to any other uid (for example 501 on macOS, or a second Linux account), the container's process has no user name.
2. Spark's JVM needs a named Unix user at login, so any uid other than 1000 fails.
3. Without `--user` the container runs as its default user (root). That is fine for Spark in local mode, and the script keeps `HADOOP_USER_NAME=student` and `USER=student`.
4. The run writes nothing into the mounted `/lab` folder from inside the container: the log and the receipt are written by host-side redirects. So no root-owned files appear on the student's machine, and nothing else changes.

A human approved the one-line change, because it alters a public repository. Then a guard fired: the agent was not permitted to clone the public repository, because that repository was not yet registered in its capability scope (§4). So the operator ran the public push. The pattern is: evidence, then a human decision, then a guarded action. It is the same shape as the contracts of U16: the principal cannot audit every step, so the agent's report must be checkable and the consequential action is gated.

What the agents got wrong, and why that is the point.

- *Reasoned, not run.* In this set-up the drafting agent could not run Lean or render slides on its own machine. Its first claims about Lean output were reasoning. They became facts only when the operator ran the check file.
- *Confident but wrong.* One review predicted that a Lean warning would not appear. The real run showed that it did, and the warning (`Variable name` `` `hp` `` `is not explicitly referenced`) answered a question posed in the lecture: the hypothesis `hp` of the Brent-bound theorem is not needed, because `cdiv a 0 = 0`.
- *Formatting at scale.* Hard-wrapped source text broke lines mid-sentence on every slide. It was fixed with a tool, not by hand.
- *Records that certified nothing.* A new check found a certificate record attached to no statement. The mathematics was fine; the bookkeeping was not.

`MATHS->CS:` use AI to go faster, and trust it only as far as a check you can re-run yourself. The rest of this unit makes that sentence precise: which agent may act (§4), what survives when it forgets (§5–§6), and why its reports can be trusted (§7–§9).

2. The substrate is a graph: SPIG

Fix the object first. The **Substrate-Process IAS Graph (SPIG)** is a labelled directed multigraph whose vertices are *everything the system is made of* — agents, operators (natural persons), policy declarations, enforcement hooks, log entries, memory directories, design documents, running daemon processes — and whose typed edges are *everything the system does*: `declares`, `enforces`, `references`, `attests`, `modifies`, `spawns`, σ`-records`. Every operation of the system is an edge- or node-modification of SPIG; the append-only edge-log is the σ-chain of §6. `MATHS->CS:` "the platform" stops being a vague noun — it is one graph, and questions about the platform become graph queries.

One structural fact matters immediately: a policy document that *declares* a rule and a hook that *enforces* that document form a 2-cycle (`declares` one way, `enforces` back). These **meta-circular** pairs — the system regulating its own regulators — are strongly connected components of SPIG. So the substrate's own task graph is NOT acyclic, and the first algorithm of §3 must be one you met in U11: condense the SCCs.

Check your reading.
Q1. In the SPIG (the substrate as a labelled directed graph), a policy that DECLARES a rule and the hook that ENFORCES that policy point at each other. What does this create, and why does it matter for scheduling?
(A) a redundant edge that can be deleted (B) a meta-circular 2-cycle – a strongly connected component – so the substrate's task graph is NOT acyclic and must be SCC-condensed before anything can be scheduled (C) a scheduling deadlock with no resolution (D) nothing – cycles are ignored by the scheduler

3. Governed scheduling: the Lagom Obligations Graph

The unit of work is an **obligation** — a task node carrying its dependencies, its owner, and the *preconditions the scheduler enforces before dispatch*: the acting agent is within its capability scope (§4), the memory writes are label-legal (§6, axiom A1), and the action will be σ-recorded before its effects are observable (axiom A2). The obligations form the **Lagom Obligations Graph** — a directed graph, cyclic in general (§2), scheduled by the four-stage **TBMLS** pass (Tarjan–Brent–Mirsky Lagom Scheduling):

1. **(T) Tarjan condensation.** Strongly connected obligation-clusters (the meta-circular cores of §2) are condensed into super-nodes; the result is a DAG. This is U11's SCC section doing load-bearing work: *the reason SCC belonged in a distributed-computing course is that governed scheduling needs it before anything else runs.*
2. **(B) Brent/PERT critical path.** On the condensed DAG, the longest dependency chain is the depth D; total obligation cost is the work W (U0's lenses, verbatim).
3. **(M) Mirsky antichain layering.** The DAG decomposes into antichains — sets of obligations with no dependency among them — which are exactly the *parallel dispatch waves*: what the fleet may run simultaneously.
4. **(LS) Allocation.** Obligations within a wave are allocated to agents subject to the governance preconditions; the acyclic restriction of this pass is **BMLS**, and its completion-time guarantee is the course's oldest friend:

Proposition (Prp-BMLSPar, first half — machine-checked, `IasCert/Paper/BMLSPar.lean`). For total work W, critical-path depth D, and $p \geq 1$ agents, the scheduled completion time obeys the Brent (1974) bound in its *certified* form

$$T_p \;\leq\; D + \frac{W - D}{p},$$

of which the familiar $T_p \leq W/p + D$ is the looser corollary. *The first theorem you machine-checked*

in U0P bounds the running time of the agent fleet in U17. The second, harder half (allocation optimality) is not yet certified — the certificate says so explicitly.

Two companions, both machine-checked:

Proposition (Prp-DisjointWritesCorrect — *machine-checked*, `DisjointWrites.lean`). *If a batch of obligations has pairwise-disjoint write-sets, dispatching the whole batch in one wave yields the same state as any sequential execution of it* — so a Mirsky wave whose members do not write the same cell is correct without locks (the parallel-correctness lemma family of U1/U2, in production clothes: this is *why* the scheduler is allowed to run an antichain at once).

Proposition (Prp-ReplanCost — *machine-checked*, `ReplanCost.lean`). *When an agent fails mid-obligation, replanning wastes at most one super-node of work* — the coarsened (SCC-condensed) unit the failed obligation belonged to, and no more. Failure is bounded by the granularity of the condensation of §3, not by the whole schedule (U3's straggler problem, with a proved ceiling).

`MATHS->CS:` the industry phrase "orchestration framework" names, at best, stages (M)+(LS) without (T) or the governance preconditions. The formal content of *governed* scheduling is precisely that the axioms of §6 are *checked before dispatch* — which is what makes the completion-time bound a guarantee about a safe execution rather than a fast unsafe one.

Check your reading.

Q2. TBMLS schedules the obligations graph in four stages: (T) Tarjan condensation, (B) Brent/PERT critical path, (M) Mirsky antichain layering, (allocate). Which earlier course result reappears as the completion-time guarantee of the final stage?
(A) the Master theorem (B) the Brent bound $T_p \leq \mathrm{W/p} + \mathrm{D}$ – the first theorem the course machine-checked, now bounding the agent fleet's completion time (C) the Chernoff bound (D) Amdahl's law

Q3. An 'orchestration framework' that runs stages (M)+(allocate) without (T) or the governance preconditions can still be fast. In two sentences, say what it LACKS that makes governed scheduling's completion-time bound a guarantee about a SAFE execution rather than a fast unsafe one.

4. Governed capability: what a worker may do, and what it may see

Industry calls this "guardrails". The formal object is a pair of lattices under one algebra:

- The **capability lattice**: an agent's permitted action-set is a *projection* of the full action space (the paper's IPLiKiDi projection), and the enforcement question — "is this dispatch within scope?" — is an order-theoretic one, answered at a policy decision point before the action runs. Realisation soundness (the projection compiles to kernel-level filesystem policy) is the paper's `Prp-ContainmentRealisationSoundness` *(stated; the kernel-realisation certificate is a later tranche — the OS-level model is not yet mechanised).*
- The **egress lattice**: dual to *doing* is *seeing* — what may flow OUT through a worker (and hence leak through a prompt-injected worker). The paper proves the two boundaries are ONE relative-complement construction instantiated twice, with a Galois connection (`Prp-EgressBoundaryGalois` — *machine-checked*, `GaloisBoundaries.lean`) tying enforcement to specification: the deny-set is authored once and both boundaries enforce it.

`MATHS->CS:` prompt injection is the industry's unsolved terror precisely because ad-hoc filters do not compose. Order theory composes. A capability system with a proved Galois connection between "what is specified" and "what is enforced" turns agent security from pattern-matching into algebra — this is the section where the course's security content becomes an employable skill. The capability/egress algebra is machine-checked (`GaloisBoundaries.lean`) and, notably,

uses *none* of the substrate axioms of §6: security here is pure order theory, constraining any substrate rather than resting on the framework's own assumptions.

4.1 The algebra, stated (all machine-checked)

Fix a lattice Cap of capabilities (finite unions of permission patterns) and, for each task t, its **allowlist** $\mathrm{allow}(t) \in \mathrm{Cap}$. The **projection** sends a task-set to its capability footprint,

$$\varphi(T) \;=\; \bigsqcup_{t \in T} \mathrm{allow}(t),$$

and its **right adjoint** sends a budget to the tasks it admits, $\varphi^{\dashv}(C) = \{\, t : \mathrm{allow}(t) \sqsubseteq C \,\}$.

Proposition (IPLiKiDi Galois connection — *machine-checked*, `iplikidi_galois`). *For every task-set T and capability budget C,*

$$\varphi(T) \sqsubseteq C \quad\Longleftrightarrow\quad T \sqsubseteq \varphi^{\dashv}(C).$$

Read it as one sentence of security said two ways: *the task-set's footprint fits the budget iff every task is within the budget's allowance.* The two consequences the certificate also carries — $T \sqsubseteq \varphi^{\dashv}(\varphi(T))$ (`le_adj_phi`) and $\varphi\varphi^{\dashv}\varphi = \varphi$ (`phi_adj_phi`) — are the closure operators of §F.C2b §4; their fixed points are the **polar pairs**, the framework's *minimal-and-sufficient* capability scopes.

The deny-tier. A sovereign rule must *subtract* — deny a set of capabilities no grant can restore. This is the relative complement $\setminus$ of F.C2b §3, characterised by the two laws the certificate proves are exactly a `GeneralizedBooleanAlgebra`:

Lemma (deny-tier laws — *machine-checked*, `paper_disjointness` + `paper_recovery`). $(a \setminus d) \sqcap d = \bot$ *(the deny truly removes)* and $(a \setminus d) \sqcup (a \sqcap d) = a$ *(nothing else is lost).*

The enforced boundary. Enforcement projects, then subtracts the sovereign deny-set Deny: $\varphi_{\mathrm{enf}}(T) = \varphi(T) \setminus \mathrm{Deny}$.

Proposition (enforced boundary is still a Galois connection — *machine-checked*, `enforced_galois`). φ_{enf} *is a left adjoint, with right adjoint $C \mapsto \varphi^{\dashv}(C \sqcup \mathrm{Deny})$:*

$$\varphi(T) \setminus \mathrm{Deny} \;\sqsubseteq\; C \quad\Longleftrightarrow\quad T \sqsubseteq \varphi^{\dashv}(C \sqcup \mathrm{Deny}).$$

The proof is one residuation step ($a \setminus d \sqsubseteq c \iff a \sqsubseteq c \sqcup d$) composed with the base connection — subtracting the deny provably preserves the whole algebra.

One algebra, two boundaries. The dual of *doing* is *seeing*. Replace the action-capability lattice by a **context** lattice (what a worker may read, hence leak) and the sovereign deny by a per-tier deny-set $\mathrm{sdeny}(\tau)$, and the *identical* theorem returns:

Proposition (egress boundary — *machine-checked*, `egress_galois`). *For every tier τ, $\varphi_{\mathrm{enf}}(\cdot, \mathrm{sdeny}(\tau))$ is a left adjoint with right adjoint $C \mapsto \varphi^{\dashv}(C \sqcup \mathrm{sdeny}(\tau))$.*

So the same relative-complemented Galois algebra governs **both** boundaries — the *action* boundary (what a worker may do, compiled to kernel filesystem policy) and the *egress* boundary (what a worker may see, compiled to the context a prompt is built from) — the deny authored once, enforced on both sides, proved once. The four boundary results above (five Lean theorems; with two helper lemmas, seven in all) are machine-checked in `GaloisBoundaries.lean` and — the finding worth stating plainly — the file does **not import the axiom base**: the security algebra rests on *no* substrate axiom. It is pure order theory, and it constrains *any* substrate, ours or another's.

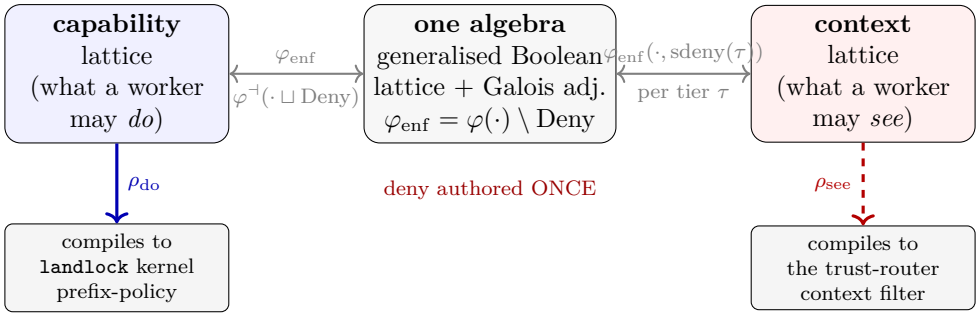


Figure 23.1: One algebra, two boundaries. The relative-complemented Galois algebra (centre) is instantiated twice: on the left as the **action** boundary — what a worker may *do*, ρ_{do}-compiled to a kernel `landlock` prefix-policy — and on the right as the **egress** boundary — what a worker may *see* (and hence leak), compiled to the trust-router's context filter. The sovereign deny-set is authored once; the *same* machine-checked adjunction (`enforced_galois`, and its tier-parametrised twin `egress_galois`) enforces it on both sides. Security as algebra, not pattern-matching.

Exercises (§4). 1. *Residuation.* In a generalized Boolean algebra, prove $a \setminus d \sqsubseteq c \iff a \sqsubseteq c \sqcup d$. (This one line is the engine of the enforced-boundary proposition; `GaloisBoundaries.lean` discharges it as `sdiff_le_iff`.) 2. *Why not a pseudo-complement.* Replace the genuine relative complement by a Heyting pseudo-complement $\neg d$, denying by $a \sqcap \neg d$. Show that disjointness survives, $(a \sqcap \neg d) \sqcap d = \bot$. Then, in the Heyting algebra of open subsets of $\mathbb{R}$, exhibit a capability a and deny d for which **recovery fails**, $(a \sqcap \neg d) \sqcup (a \sqcap d) \neq a$, so the deny silently removes capabilities outside d; and show that residuation, $a \sqcap \neg d \sqsubseteq c \iff a \sqsubseteq c \sqcup d$, fails for the same pair. Finally, taking $a = \top$, show that recovery for *every* pair forces $d \sqcup \neg d = \top$ — excluded middle, hence $\neg\neg d = d$. Conclude that only the genuine relative complement gives a sound sovereign seal — the reason the paper (and the certificate) insist on a generalized Boolean algebra, not a Heyting one. 3. *Pair the boundaries.* Both boundaries take a task-set, so they are enforced together by pairing them, not by composing them: $T \mapsto \big(\varphi(T) \setminus \text{Deny},\ \psi(T) \setminus \text{sdeny}(\tau)\big)$ into the product lattice Cap × Ctx, ordered componentwise, where $\psi(T)$ is the egress proposition's analogue of φ on the context lattice (the join of the tasks' context allowlists; not the context projection π of §5). Show that this pairing is again a left adjoint, and identify its right adjoint. (The proof runs the two adjunctions side by side — the same move the enforced-boundary proof makes once.)

Check your reading.
Q4. The unit models 'what a worker may do' (capability) and 'what a worker may see' (egress) as two boundaries tied to policy by a Galois connection. What does the Galois connection guarantee that ad-hoc filters do not?
(A) faster policy checks (B) that the deny-set specified ONCE is exactly the deny-set enforced everywhere – specification and enforcement composed by order theory, so the boundaries compose instead of leaking (C) that no worker can ever be compromised (D) that the model output is always correct

5. The worker forgets: the context projection

A language-model agent's working state is its **context window** — and the window is finite. When the conversation exceeds it, a **context projection** π (industry: the agent's "memory" being "compacted") maps the full history to a shorter representation, and information is *lost*. The empirical signature is familiar to every practitioner: an agent re-proposes work it completed

hours earlier, because the completion fell out of context while remaining true in the substrate. The paper names this the *preconstructive curse* and builds the entire framework as a structural defense against π-induced loss.

Two consequences the course prices exactly:

1. **Context is a resource.** Alongside time, memory, and communication (U0), the fleet's ledger carries *context tokens*, verification depth, and operator attention — a five-component cost vector. The three-lens box above gains a fifth lens whenever the worker is an LLM: U0L's token-cost arithmetic applies to the agents building the system, not only to the model being served.
2. **Recovery must be a theorem, not a hope.** If the worker forgets, the *system* must not. That requires an object that provably carries everything needed — which is §6.

Check your reading.
Q5. A language-model agent's working state is its finite context window. When history exceeds it, a projection maps the full history to a shorter representation and information is lost. What is the observable symptom, and what must carry the lost state?
(A) the agent crashes; a backup process carries the state (B) the agent re-proposes work it already completed (the completion fell out of context though it stayed true in the substrate); the SUBSTRATE, not the worker, must carry recovery – as a provable sufficient statistic (C) the agent slows down; more memory fixes it (D) nothing observable; the projection is lossless

6. Continuity without process-liveness: the σ-chain as a sufficient statistic

The substrate's persisted state is a typed bundle: cross-commons canonical state, per-group shared state (git-mediated), per-agent private memory (label-exclusive), and broadcast views. Its evolution is driven by four operators — the context projection π, the write-through σ (every in-vocabulary action is recorded *before* its observable effect), the sync Σ (merge of parallel views), and the coevolution C (quorum-gated specification change).

Three facts, at teaching pace:

- **Merge is a semilattice-join (machine-checked, `SigmaMerge.lean`).** Modelling a state's history as its commit-set, the sync operator is set-union: idempotent (`Lem-SigmaIdempotence`), history-preserving (`Lem-SigmaMonotonicity`), and a *least* upper bound (`Lem-SigmaConvergence`) — so multi-party reconciliation is order-independent and convergent. You verified the identity `s ∪ s = s` yourself in F.E-σ's lab; here it is as infrastructure.
- **The σ-chain is a sufficient statistic (`Prp-SigmaSuff` — *stated; certificate pending*).** Precisely: for the substrate-correctness decision class $\mathcal{D}_{\text{IAS}}$ (the yes/no questions "is canonical state reconstructible / is lineage intact / is the current specification the agreed one"), the recorded chain $\sigma(h)$ is a *sufficient statistic* for the full history h — the decision's answer factors through $\sigma(h)$ (Fisher–Neyman), so h and $\sigma(h)$ induce the *same* verdict on every question in $\mathcal{D}_{\text{IAS}}$. The worker may forget the history under the context projection π; the chain survives, and because it is sufficient, *nothing the decision needs is lost.* This is the statistical heart of agent memory, and it is a cousin of the lumpability condition of Kemeny–Snell (1960). A partition of a Markov chain's states into blocks is *lumpable* when the aggregated process is again a Markov chain for every initial distribution, which holds exactly when, for every pair of blocks A, B, the one-step probability $\sum_{j \in B} p_{ij}$ of moving into B is the same for every state $i \in A$ (F.E5 §5.8). Both are coarsenings that lose nothing that matters: lumpability keeps the Markov property of the dynamics, sufficiency keeps the verdict of every question in the decision class — here the σ-chain is the coarsening and

$\mathcal{D}_{\text{IAS}}$ the decision class. (Certificate pending: the statement is exact; only the machine proof is future work.)

- **Continuity is artifact-mediated.** No agent process persists; continuity claims quantify over the *artifacts* (chain + store + labelled memory), never over process-liveness. A previously-carried persistence assumption fell exactly when this was formalised — the first demonstration that mechanisation *refines the theory* (§11).

The axiom base, minimally. The framework rests on substrate axioms (memory-inviolability per label; σ-discipline), synchronisation axioms (within-group, across-group, persistence across hibernation), and governance axioms (quorum; ritual enforcement) — with three further structural candidates (finite bounds; boundary maintenance; single-host session-uniqueness). The ten axioms are *machine-checked* as ten **separate** classes (`Axioms.lean`): no proof may take a bundled "all axioms" hypothesis, so each certificate's axiom footprint is visible in its type. The unified continuity proposition (`Prp-UnifContThrm` — *stated; capstone certificate pending*) is their conjunction: every live agent at every time can reconstruct complete canonical state, lineage, and current specification. **Which axioms are truly needed is itself a theorem-shaped question**, and the certification campaign's ablation pass answers it by countermodel (`Ablation.lean`, *machine-checked*): the base is consistent, and none of the five *suspect* axioms is derivable from the five *core* ones at the skeletal-carrier level — so every demotion must come from carrier refinement or theory-design, never derivation. The footprints already read from the certificates: two of the three σ-merge lemmas need **no** axioms at all, the persistence lemma composes from `BranchPersistence` alone, and the capability/egress algebra of §4 uses **none** of the ten — early evidence that the theory carries less axiomatic weight than it first appears.

Check your reading.
Q6. Modelling a substrate state's history as its commit-set, the sync (merge) operator is set-union. Which three properties make multi-party reconciliation order-independent and convergent?
(A) commutative, transitive, reflexive (B) idempotent, history-preserving (monotone), and a least upper bound – i.e. merge is a semilattice join (C) associative, invertible, bounded (D) linear, continuous, differentiable

7. Identity and attestation: which agent did what, provably

Identity is a lattice of projections (the same lattice U15 used to locate watermarks between quotients of text-identity — one formalism, two applications). An agent's canonical label binds its memory scope (axiom A1) and its capability scope (§4); an **attestation envelope** binds machine-provenance and natural-person-provenance to a signed record. `MATHS->CS:` "audit log" becomes: a chain whose every entry is hash-linked (F.C2) and whose validation is an algorithm you can run — and break, and watch fail — in §10's lab.

8. Many agents, many groups: coordination and robustness

Within a group, parallel writers reconcile by the semilattice-join of §6 at epoch boundaries; across groups, by broadcast epochs. Specification changes require quorum — the system's governance is itself scheduled and σ-recorded work. Two results give the section its teeth: numbering of federated groups converges to a unique global order under bounded propagation latency (`Prp-NumConv`, *stated; certificate pending* — a clean distributed-systems argument in the U11/U19 family), and the threat model is Byzantine: the design target is maximal recovery from ONE uncompromised agent plus the sealed archive. U19's ADMM consensus is the optimisation face of the same coordination question; this section is its systems face.

9. Why report honestly? Incentives close the loop

A fleet of capable agents raises U16's question in machine form: the principal (the operator) cannot observe effort, only reports. Statistical contract theory — incentive compatibility, honest-reporting mechanisms — is the layer that makes the σ-chain's *verifiability* into *trustworthiness*: the chain makes lying detectable; the contract makes honesty optimal. U16 taught the theory; U17 is its deployment site. And the worker inside every obligation is the LLM of U0L — which is why U18, the finale, assembles the whole course around it.

Check your reading.
Q7. The sigma-chain makes an agent's actions VERIFIABLE. Why is that not yet TRUSTWORTHINESS, and what closes the gap? Answer in two sentences using this unit's vocabulary.

10. Lab — the receipt chain you have been living inside

Since Week 1's "first receipt" micro-lab, every lab in this course has emitted a hash-chained receipt of its computed intermediates — the grading substrate itself. This unit's lab closes the circle:

1. Run a three-agent toy orchestration in the course's docker-compose stack; emit σ-events.
2. Validate the chain with the chain-validation algorithm (the paper's Appendix B), implemented in the lab kit.
3. **Tamper** with one recorded event; re-validate; watch the validation fail at exactly the tampered link — and explain *why* from F.C2's collision-resistance.
4. Reflect (instructor copy carries the model answer): what, precisely, does a green validation *prove*, and what does it *not* prove? (Soundness about the chain; nothing about un-recorded actions — the honesty boundary of §9.)

Verification, not assertion — the ethos of the book, applied to the book's own grading.

11. Module-3 projects (deep dives; each = one paper section + one certificate)

1. **Implement BMLS** and measure fleet completion time against the certified bound on real obligation graphs; attempt the uncertified second half.
2. **Certify the σ-lemma family further**: content-loss bounds under recording restrictions (`Lem-SigmaRecContentLossBound`).
3. **Simulate the substrate as a controlled Markov chain** (U12 machinery; the paper's controlled-chain definition) and study boundary-maintenance as work against entropy.
4. **The metacircular fixed point**: formalise the meta-circular core (§2) and the fixed-point of specification coevolution (`Prp-MetacircFP`, Appendix C.2).
5. **Federation convergence**: implement the numbering protocol and stress (`Prp-NumConv`) under injected latency; compare with the certificate's bound.

Each project's write-up doubles as exposition folded back into the paper — the course refining the research that produced it.

12. Sources and provenance

Our own exposition of the interacting-agent-systems paper (the substrate's primary research); Brent (1974) for the bound of §3; Tarjan (1972) for SCC condensation; Mirsky (1971) for antichain decomposition; Kemeny–Snell (1960) for the lumpability of §6; the machine-checked certificates cited per-proposition live in the course's public certificate library, and every claim

marked *machine-checked* names its Lean file. Claims marked *pending* are pending — the honesty ledger, not the prose, is the source of truth.

Chapter 24

18 llm ddl

U18 — LLMs & distributed deep learning: the course composed

The ScaDaMaLe book, unit 18 (examined finale, Part V). Our own exposition; original sources credited in §14. This unit is examined.

The one organising idea of this unit: *you have seen every part of this machine; now we assemble it and make it big.* You met the LLM on day one (U0L) as the course's running workload, and every unit since has ended by landing its mathematics on it — the LLM-lens boxes. This unit is the second pass, in depth: every algorithm, cost model, and statistical lens the course has developed in units 0–17 is a component of the system assembled here, named explicitly — not as a metaphor, but as a cost-accounting exercise.

Instantiating the three-lens invariant. The U0 box for this unit:

> **RAM:** an LLM is a function over token sequences; training is an M-estimation problem (U9); a forward pass costs $\approx 2ND$ FLOPs (about ND multiply-accumulates, one per parameter per token) where N = parameter count and D = token count processed. **PRAM:** the transformer's per-layer operations are matmul compositions (U6); attention has work $O(T^2 d)$ and depth $O(\log T + \log d)$ per layer (its inner products and softmax sums are reductions, U1). **DPRAM/MPC:** training a large model requires more memory than any one device holds; the course's full toolkit — reductions (U1), scheduling (U3), matmul distribution (U6), optimisation (U9), memory accounting (U4), ANN retrieval (U13), density estimation (U14/U15) — is deployed simultaneously. This unit accounts for every piece.

Entry kit — what this chapter assumes; follow a link if you need the memory jog (map: Figure 1): F.M2 (the matrix view of attention) • F.S3 (distributions and estimation vocabulary) • F.E3 (the orchestration substrate) • F.E6 (the training-runtime anatomy)

1. Intuition — the course as a construction kit

A 70-billion-parameter language model trains on roughly one trillion tokens. A single GPU holds perhaps 80 GB of HBM. You cannot fit the model, let alone a training batch. You need to split the model across hundreds of GPUs and synchronise them after every gradient step. You need an optimiser that converges despite noise from mini-batches. You need to estimate how much data the model needs before adding more parameters is a waste. You need to detect whether the model's output carries a statistical signature. Every one of these needs has an exact counterpart in the previous units.

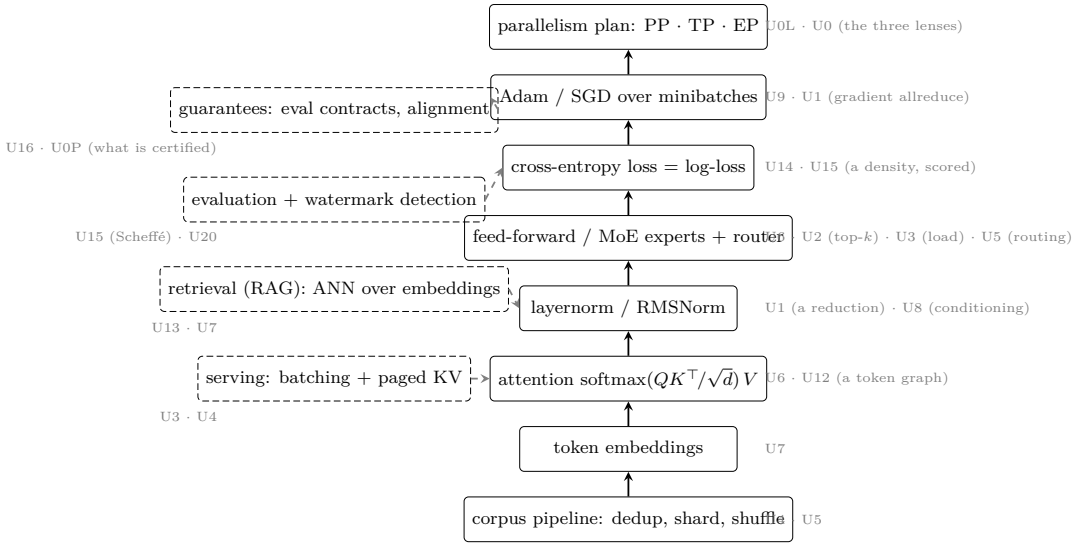


Figure 24.1: The course, composed: every block of a modern LLM system labeled by the unit that explains it. The central column is the model and its training loop; the dashed satellites are the deployment surround. Nothing in this machine is new to a reader of Parts 0–IV — U18's job is assembly and scale, not introduction. This figure is the atlas's sink node *u18*, drawn from the inside.

The intuition for this unit is simple: *stop reading it as a survey of a fashionable topic and read it as a completion of the course.* Figure 24.1 is the whole claim as a picture — every block labeled by the unit that explains it.

Check your reading.
Q1. U18's organising claim is that every need of distributed LLM training and serving has an exact counterpart in units 0-16. Which pairing does the unit NOT make?
(A) gradient synchronisation across data-parallel replicas – U1's reduction / all-reduce (B) pipeline fill-drain bubble – U3's scheduling and makespan accounting (C) KV-cache sizing – U4's memory accounting (D) RAG document retrieval – U6's distributed matmul

2. Formally what an LLM is: a conditional density estimator

2.1 Autoregressive factorisation

A vocabulary $\mathcal{V}$ is a finite set of tokens (subword units produced by a tokenisation algorithm, discussed below). A token sequence of length T is $\mathbf{x} = (x_1, \ldots, x_T) \in \mathcal{V}^T$. The chain rule of probability factors the joint over any ordering; the left-to-right ordering is:

$$p(\mathbf{x}) \;=\; \prod_{t=1}^{T} p(x_t \mid x_1, \ldots, x_{t-1}) \;=\; \prod_{t=1}^{T} p(x_t \mid x_{<t}).$$

A **language model** is a parameterised family $p_\theta(x_t \mid x_{<t})$: for each prefix $x_{<t}$, it produces a probability distribution over the next token. This is a **conditional density estimator** over the discrete space $\mathcal{V}$ — the same statistical object studied in U14 (density estimation on tree structures) and U15 (minimum-distance estimation), now instantiated over a very high-dimensional function class (a deep neural network).

The connection to U14/U15 is not rhetorical. The fundamental statistical question is the same: given n samples $\mathbf{x}^{(1)}, \ldots, \mathbf{x}^{(n)}$ (the training corpus), find $\hat{\theta}$ such that $p_{\hat{\theta}}$ is close — in an appropriate distance — to the true data-generating distribution p^*. The choice of distance is

load-bearing; empirical cross-entropy (the training objective) corresponds to KL divergence, which can be very large even when the L1/TV distance is small (a point U15 makes sharp). Honest epistemics: we do not know whether any LLM is close to p^* in the L1 sense; the training objective does not guarantee it.

2.2 Training objective as M-estimation (U9 lens)

The training corpus is a sequence of D tokens, chunked into contexts of length T. The **empirical cross-entropy loss** is:

$$\hat{L}(\theta) \;=\; -\frac{1}{D}\sum_{t=1}^{D} \log p_\theta(x_t \mid x_{<t}).$$

Minimising $\hat{L}(\theta)$ over θ is an **M-estimation problem** (U9, §4.1's finite-sum objective): we minimise an empirical average of a loss function $\ell(\theta; x_t, x_{<t}) = -\log p_\theta(x_t \mid x_{<t})$. The asymptotic theory of M-estimation (under regularity conditions that do not hold in full generality for overparameterised networks — honest caveat) says the minimiser converges to the KL-minimiser of the model class. In practice, convergence guarantees are empirical rather than theoretical for the regimes of interest.

The training optimiser is a variant of **Adam** (U9, §3.2): adaptive per-parameter learning rates computed from running estimates of the first and second moments of the gradient. At the scale of hundreds of billions of parameters and trillions of tokens, this is the algorithm of U9 — applied to a loss surface that is high-dimensional, non-convex, and only partially understood.

2.3 Tokenisation (one paragraph)

Before the network sees any text, a fixed **tokenisation** map converts raw bytes or characters into token IDs from $\mathcal{V}$. The dominant algorithm is **Byte-Pair Encoding (BPE)**: beginning with a vocabulary of individual bytes, iteratively merge the most frequent adjacent pair into a new token, repeating until $|\mathcal{V}|$ reaches the target size (typically 32 000–100 000 tokens). BPE is a sequential greedy algorithm on the corpus; it runs once, offline, and the resulting vocabulary is fixed at training time and frozen at inference. Its cost does not factor into the training complexity accounting below; we mention it because it shapes what the input space $\mathcal{V}^T$ is, and because tokenisation artefacts are one source of failure modes that a rigorous analysis of p_θ vs p^* must eventually confront.

3. The transformer as matmul composition (U6 lens)

3.1 Attention

The central operation of the transformer (Vaswani et al., 2017) is **scaled dot-product attention**. Given a sequence of T token representations, each a d-dimensional vector, form query matrix $Q \in \mathbb{R}^{T\times d_k}$, key matrix $K \in \mathbb{R}^{T\times d_k}$, and value matrix $V \in \mathbb{R}^{T\times d_v}$ by linear projections:

$$\text{Attention}(Q,K,V) \;=\; \text{softmax}\!\left(\frac{QK^\top}{\sqrt{d_k}}\right)V.$$

The factor $1/\sqrt{d_k}$ is a variance-stabilising normalisation; without it, large d_k pushes the dot-products into regions where the softmax saturates. Every operation here is a matmul or an elementwise nonlinearity: $QK^\top$ is a $T \times T$ matrix multiplication ($O(T^2 d_k)$ work, $O(\log d_k)$ depth

on PRAM, each entry an inner product, U6), the softmax divides each row by a sum of T terms (a reduction, $O(\log T)$ depth), and the final product with V is $T \times T$ times $T \times d_v$ ($O(T^2 d_v)$ work, $O(\log T)$ depth). In practice $d_k = d_v = d/h$ where h is the number of heads; **multi-head attention** runs h independent attention heads in parallel and concatenates their outputs, at the same total work.

The U4 memory connection. The $T \times T$ attention matrix must be materialised (or recomputed under memory-efficient variants like FlashAttention); at sequence length T and h heads, this costs $O(T^2 h)$ HBM memory — the dominant memory bottleneck at long context. The **KV cache** at inference (see §6) stores keys and values for all previous tokens across all layers; its size is accounted precisely in §6.

3.2 MLP blocks

Each transformer layer also contains a **two-layer MLP**: given the d-dimensional representation, expand to $4d$, apply a nonlinearity (GeLU or ReLU), project back to d. This is two matmuls: $\mathbb{R}^{T \times d} \to \mathbb{R}^{T \times 4d} \to \mathbb{R}^{T \times d}$. Work per layer per token: $O(d \cdot 4d + 4d \cdot d) = O(8d^2)$. Per the U6 lens, these are exactly the matrix-matrix multiplications of that unit; the communication-avoiding arguments apply when they are sharded across devices (§4.2 below).

3.3 Per-layer FLOPs and the 6ND estimate (U0 work lens)

A transformer with L layers, hidden dimension d, and h attention heads has, per token, approximately:

- **Attention projections (Q, K, V and output):** $4 \times 2d^2$ FLOPs $= 8d^2$ (four $d \times d$ projections, each costing d^2 multiply-accumulates, that is $2d^2$ FLOPs).
- **Attention $QK^\top$ and $\cdot V$:** $2 \times 2Td$ FLOPs $= 4Td$ per token (dominant at long context; subdominant when $T \ll d$).
- **MLP:** $2 \times 2 \cdot 4d^2 = 16d^2$ FLOPs.

Summing and noting that the dominant terms at $T \ll d$ give $\approx 24d^2$ FLOPs per token per layer. The parameter count N for the attention + MLP blocks is approximately $12Ld^2$ (ignoring embeddings and layer-norms, which are sub-leading). Therefore FLOPs per token per layer $\approx 2N/L$, and across L layers:

$$\text{Forward-pass FLOPs per token} \;\approx\; 2N.$$

Training requires a forward pass plus a backward pass (gradient computation). The backward pass costs approximately twice the forward pass (compute the activations forward; propagate the gradient backward through the same graph). Therefore:

$$\text{Training FLOPs per token} \;\approx\; 6N,$$

and for a full training run of D tokens:

$$\boxed{C \;\approx\; 6ND}$$

This is the **classic 6ND estimate** (Hoffmann et al. 2022; Kaplan et al. 2020). It is derived, not pulled from a table. The exact constant depends on the architecture details (embedding matrices, layer-norm, rotary embeddings — these are sub-leading for large N); the $6\times$ factor splits $2\times$ forward + $4\times$ backward (gradient w.r.t. parameters and activations).

Attention's additional $O(T^2d)$ cost. At $T \gg d$ (very long contexts), the attention matmul term dominates; the $6ND$ estimate then under-counts. For standard pretraining with $T \leq 4096$ and $d \geq 4096$, the $T \ll d$ regime holds and $6ND$ is accurate.

Worked example: pricing a training run. Device rates are illustrative sustained rates, not peaks. A full fine-tune of $N = 10^9$ parameters on $D = 10^8$ tokens costs $C = 6{\times}10^9{\times}10^8 = 6{\times}10^{17}$ FLOPs.

device (illustrative sustained rate)	time $= C/\text{rate}$
GPU-class, 10^{14} FLOP/s	6×10^3 s $\approx$ 1.7 h
laptop-CPU-class, 10^{12} FLOP/s	6×10^5 s $\approx$ 6.9 days

Training from scratch at $N = 8 \times 10^9$, $D = 10^{12}$ tokens costs $C = 6 \times 8 \times 10^9 \times 10^{12} = 4.8 \times 10^{22}$ FLOPs. At 10^{14} FLOP/s that is 4.8×10^8 s, about 15 years on one device. That is why training is distributed (§4).

Check your reading.
Q2. The training-cost estimate C = 6ND is derived, not pulled from a table. Per token, the factor 6N splits as:
(A) 6 forward passes per token (B) 3x forward + 3x backward (C) 2x forward + 4x backward (gradients w.r.t. both parameters and activations) (D) 1x forward + 5x optimiser update

4. Distributed training: the course composed

This section is the centerpiece of U18. Distributed LLM training is not a new category of problem — it is the simultaneous deployment of U1 (reductions), U3 (scheduling), U4 (memory accounting), U6 (distributed matmul), and U9 (distributed optimisation). We account for each parallelism strategy in the three-lens vocabulary.

4.1 Data parallelism: all-reduce as U1's reduction tree

In **data-parallel** training, p devices each hold a full copy of the model (parameters and optimiser state) and process a different mini-batch shard. After each forward+backward pass, every device holds a local gradient $g_i \in \mathbb{R}^N$; the global gradient is $\bar{g} = \frac{1}{p}\sum_{i=1}^{p} g_i$, which every device must have before applying the parameter update.

This is a **reduction** followed by a **broadcast** — or equivalently, an **all-reduce**. On a binary tree (U1, §3.1): depth $O(\log p)$, work $\Theta(Np)$ (the reduction's $p-1$ vector additions of N numbers each, and as many copies in the broadcast). The efficient network implementation is **ring all-reduce** (Baidu 2017 / NCCL):

Arrange the p devices in a ring. A ring all-reduce runs in two phases, each consisting of $p-1$ steps: 1. **Reduce-scatter:** in $p-1$ steps, each device sends a N/p-element chunk to its right neighbour and accumulates incoming chunks. After $p-1$ steps, each device holds the reduced sum for one chunk of the gradient. 2. **All-gather:** in $p-1$ steps, each device broadcasts its reduced chunk to the ring. After $p-1$ steps, every device holds the full reduced gradient.

Bandwidth cost. Each phase sends $(p-1)/p \cdot N$ elements per device; both phases together:

$$\text{Ring all-reduce bytes per device} \;=\; 2 \cdot \frac{p-1}{p} \cdot N \cdot b \;\xrightarrow{p\to\infty}\; 2Nb,$$

where b is bytes per parameter (4 bytes for fp32, 2 bytes for bf16). This is **bandwidth-optimal**: each device sends and receives at most $2Nb$ bytes regardless of p, achieving asymptotically perfect

bandwidth utilisation. The U6 reader recognises the same structure as the communication-optimal matrix multiply: the ring all-reduce is the all-reduce equivalent of the 1D systolic pass.

Gradient accumulation. To simulate a large global batch without moving the full batch through memory at once, each device accumulates gradients over k micro-batches before the all-reduce. This multiplies the effective batch size by k at no additional communication cost; memory cost stays constant (only one micro-batch live at a time). In the U0 round-counting vocabulary: gradient accumulation keeps $R = 1$ (one all-reduce per global step) while raising the per-step compute-to-communication ratio by the factor k.

Worked example: the bytes of one all-reduce. Take $N = 10^9$ parameters, bf16 gradients ($b = 2$) and $p = 4$ devices. Each device sends $2 \times \frac{3}{4} \times 10^9 \times 2 = 3 \times 10^9$ B $= 3$ GB per all-reduce, that is, per global step. Over 1 Gb/s Ethernet (0.125 GB/s) that takes 24 s per step; over 10 Gb/s (1.25 GB/s), 2.4 s. At an illustrative 10^{14} FLOP/s per device, the local compute of $6N$ FLOPs per token matches 2.4 s only when each device processes $2.4 \times 10^{14}/(6 \times 10^9) = 4 \times 10^4$ tokens per step. This is why gradient accumulation raises the tokens per all-reduce.

One SGD step as a DPRAM computation (U0 §4). Each of the p machines is a PRAM holding a full copy of θ and a shard of the batch. Notation: N is the number of parameters and b bytes per number; D (in $6ND$) counts training tokens, not depth.

```
local     forward + backward on its shard      ~ 6N x (tokens in shard) FLOPs
  (PRAM)
-------------------------------------------------------------------------------
  -----------
ROUNDS    all-reduce the gradient g in R^N     ring: 2(p-1) rounds
  (DPRAM
          (same result as U1's reduce-then-    2(p-1)/p x N b bytes per machine
            wire)
           broadcast tree, done as a ring)
-------------------------------------------------------------------------------
  -----------
local     theta <- theta - eta * g_bar  (the same g_bar everywhere)   O(N)
  operations
```

Under the α-β model (F.E4), each ring round moves Nb/p bytes per machine, so

$$T_{\text{step}} \;\approx\; T_{\text{compute}} \;+\; 2(p-1)\,\alpha \;+\; \frac{2(p-1)}{p}\,Nb\,\beta,$$

and a run of S steps costs S times this, with $R = S \cdot 2(p-1)$ rounds in U0's accounting. The "$R = 1$" of gradient accumulation above counts *all-reduces* per global step; the ring realises each one in $2(p-1)$ rounds. A binary tree realises it in $2\lceil \log_2 p \rceil$ rounds, but each round moves the full Nb bytes. The tree trades bytes for rounds, the ring trades rounds for bytes, and the ring wins when βNb dominates α, which it does for gradients of billions of numbers. The update step needs no communication because every machine holds the same $\bar{g}$. A model too big for one machine adds the tensor and pipeline cuts (§4.2–4.3) on top of this schedule.

Figure 24.2 works one all-reduce through with $p = 4$ devices and $N = 10^9$ numbers.

4.2 Tensor parallelism: split matmuls across devices (U6 lens)

When the model is too large to fit on one device, **tensor parallelism** (Shoeybi et al., Megatron-LM 2019) shards the weight matrices across t devices. Consider the MLP's first linear layer $Y = \text{GeLU}(XA)$ where $X \in \mathbb{R}^{B \times d}$ and $A \in \mathbb{R}^{d \times 4d}$. Partition A column-wise: $A = [A_1 | A_2 | \cdots | A_t]$,

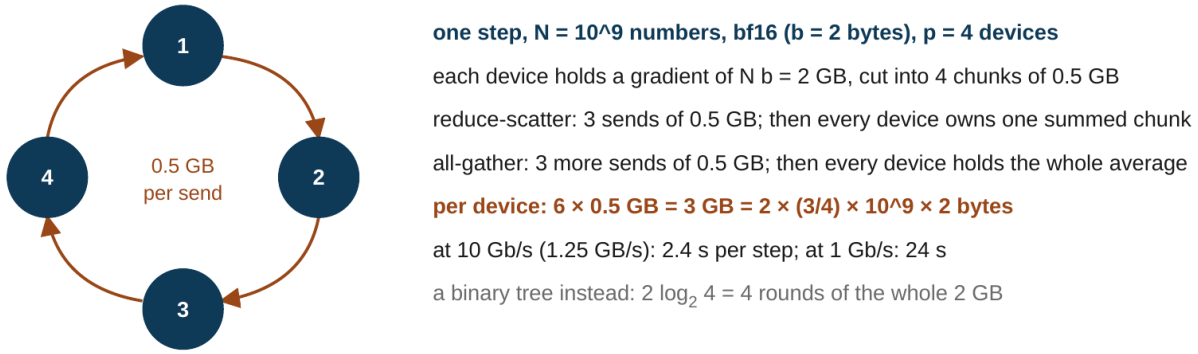


Figure 24.2: The all-reduce, training's one collective.

each $A_i \in \mathbb{R}^{d\times 4d/t}$ on device i. Each device computes $Y_i = \mathrm{GeLU}(XA_i)$ locally, since GeLU acts entry by entry; the outputs Y_i span different columns, and they need not be gathered, because the next layer is split to use each Y_i where it lies (below). This is the **block-distribution logic of U6**, in one dimension: split the outer (output) dimension, and each block of the result is computed locally with no communication; split the inner dimension, as the row-parallel layer does, and each device holds a partial sum that must be reduced.

Communication pattern (Megatron-style). Megatron pairs a column-parallel linear with a row-parallel one in each block (attention: Q, K, V split by heads, output projection split by rows; MLP: A by columns, the second matrix by rows), so each block needs **one all-reduce in the forward pass** (to sum the row-parallel partial outputs) and **one in the backward pass**. A layer has two blocks: total **4 all-reduces per layer** (2 forward, 2 backward), each of $O(B\cdot d)$ bytes across t devices. This replaces the U6 all-to-all exchange with a targeted all-reduce, exploiting the matmul's associativity — again U1's associativity argument, now applied inside a layer.

Tensor parallelism is most efficient when t is small (within a single high-bandwidth NVLink domain), since the all-reduces are synchronous and their latency adds to the critical path.

4.3 Pipeline parallelism: micro-batches and bubble overhead (U3 lens)

Pipeline parallelism (Huang et al., GPipe 2019) assigns consecutive transformer layers to consecutive devices in a pipeline of p stages. Each stage processes one micro-batch at a time and passes activations to the next stage (forward pass) or gradients to the previous stage (backward pass).

The critical path for a single batch of m micro-batches through p stages is a scheduling problem (U3). Compute the **pipeline bubble fraction** — the fraction of device-time wasted while a stage waits for input or drains at the end:

In GPipe-style (fill-drain) scheduling: the pipeline takes $m + p - 1$ time units to process m micro-batches (fill takes $p - 1$ extra units; drain takes $p - 1$ extra units; the two overlap). Of the total $p(m + p - 1)$ device-time units, $p(m + p - 1) - mp = p(p - 1)$ units are wasted (the bubble). Therefore:

$$\boxed{\text{Bubble fraction} \;=\; \frac{p(p-1)}{p(m+p-1)} \;=\; \frac{p-1}{m+p-1}.}$$

Derivation. This is exactly a makespan / utilisation calculation (U3, §3): the p-stage pipeline with m jobs (micro-batches) is a $P\|\cdot\|C_{\max}$ scheduling instance where each stage takes unit

time and stages must be sequenced. The total available work is $p \cdot m$ stage-steps (useful) out of $p \cdot (m+p-1)$ stage-steps scheduled, giving utilisation $m/(m+p-1)$ and bubble $(p-1)/(m+p-1)$.

The engineering consequence. The bubble fraction vanishes as $m \gg p$: use many micro-batches per pipeline step. But m large means memory proportional to m activations must be live simultaneously (U4's memory lens): there is a direct memory–utilisation tradeoff. Interleaved scheduling (Narayanan et al. 2021) reduces the bubble by a factor v (the number of pipeline stages per device) at the cost of increased communication, exactly as U3's preemptive schedules trade communication for makespan.

4.4 ZeRO and FSDP: sharding the memory burden (U4 lens)

The **ZeRO optimizer** (Rajbhandari et al. 2020) and PyTorch **FSDP** address a different bottleneck: even with a model that fits on one device, the **optimiser state** dominates. In the mixed-precision Adam training that ZeRO analyses, each parameter costs 16 bytes: $2N$ for the fp16 parameters, $2N$ for the fp16 gradients, and $12N$ of fp32 optimiser state (a master copy of the parameters, and Adam's first and second moments, $4N$ each) — total $16N$ bytes. (Pure fp32 training gives the same total: $4N + 4N + 8N$.) For $N = 70 \times 10^9$ that is 1,120 GB before any activations.

ZeRO partitions these bytes across p data-parallel ranks (bytes per device): - **Stage 1:** shard the optimiser state: $4N + 12N/p$. - **Stage 2:** also shard the gradients: $2N + 14N/p$. - **Stage 3:** also shard the parameters: $16N/p$.

In Stage 3, each device holds $16N/p$ bytes (vs $16N$ without ZeRO). The tradeoff (U4 memory lens): less memory per device, but more communication — in the forward pass, each device must gather the parameters it needs from other devices (one all-gather per layer), again in the backward pass, and the gradient shards must be reduce-scattered. Total communication per training step under ZeRO Stage 3: about $3Nb$ bytes per device (two all-gathers of the parameters and one reduce-scatter of the gradients, about Nb each), 1.5× the baseline ring all-reduce's $2Nb$ (Rajbhandari et al. 2020). The memory–communication tradeoff is explicit and tunable. *(Corrected 2026-10-23: the 2026-10-23 edition said* $12N$*, dropping the fp32 master copy, and* $3 \times 2Nb$ *for the communication.)*

4.5 Mixture of Experts: routing as load-balancing (U3 + U1 lens)

A **Mixture of Experts (MoE)** layer (Shazeer et al. 2017; Fedus et al. Switch 2021) replaces a dense MLP with E expert MLPs. A **router** network selects the top-k experts (usually $k = 1$ or $k = 2$ per token) for each token; only those experts compute, leaving the others idle for that token.

Routing is load-balancing (U3 lens). Each expert has a **capacity factor** c: it processes at most $c \cdot T/E$ tokens per batch. Tokens assigned to an overloaded expert are dropped (capacity overflow). The routing mechanism must therefore distribute tokens as evenly as possible across experts — precisely the load-balancing problem of U3. The **auxiliary balance loss** $\mathcal{L}_{\text{aux}} = \alpha \sum_i f_i p_i$ (where f_i is the fraction of tokens dispatched to expert i and p_i is the router's average probability for expert i) is a differentiable proxy for load imbalance — the gradient of a scheduling objective, exactly as work-stealing (U3, §4) uses the queue imbalance signal. The **power-of-two-choices** result (U3, §4.5) is a contrast, not an explanation of $k = 2$: placing each token on the *less-loaded* of two randomly drawn experts would cut the maximum load, for E tokens into E experts, from $\Theta(\log E / \log \log E)$ (one random choice) to $\log \log E / \log 2 + O(1)$ (two choices). Top-2 routing does something else: it sends each token to *both* of its two highest-scoring experts, chosen by score and not by load, so that result does not apply; the balance comes from the auxiliary loss and the capacity factor.

Communication. Each token must be dispatched to its selected expert. In a distributed MoE where experts reside on different devices, this is an **all-to-all exchange** — each device sends tokens to all expert-holding devices and receives tokens from them. This is the highest-communication operation in MoE training. As §5b counts it, each MoE layer does two all-to-alls in the forward pass (dispatch the tokens, combine the results) and two more in the backward pass; with a batch of T tokens spread over p devices, each moves about $k\,(T/p)\,d\,b$ bytes per device (b bytes per number), so about $4k\,(T/p)\,d\,b$ bytes per device per MoE layer per step, times the number of MoE layers. It is the main reason MoE training requires very high inter-device bandwidth.

Check your reading.

Q3. Data-parallel training on p = 64 GPUs of a model with N = 7 x 10^9 parameters in bf16 (b = 2 bytes). By the unit's formula, ring all-reduce bytes per device per step is closest to:
(A) 0.44 GB (B) 14 GB (C) 28 GB (D) 1.8 TB

Q4. From memory, fill the section-4 table: for each of data parallelism, tensor parallelism, pipeline parallelism, and ZeRO Stage 3, state (i) what is split, (ii) the per-step communication shape, and (iii) the earlier unit whose lens prices it.

5. Scaling laws as empirical rates (U15 epistemics)

Kaplan et al. (2020) observed that test loss $\hat{L}$ decreases as a power law in compute C, data D, and parameters N, approximately:

$$\hat{L}(N, D) \;\approx\; \left(\frac{N_c}{N}\right)^{\alpha_N} + \left(\frac{D_c}{D}\right)^{\alpha_D}$$

with empirically fitted exponents $\alpha_N \approx 0.076$, $\alpha_D \approx 0.095$ (on their corpus) and constants N_c, D_c. This is a regression on log-log axes — an **empirical rate claim** (U15 rate-eyes).

Hoffmann et al. (2022; "Chinchilla") refined the compute-optimal tradeoff: for a given FLOPs budget C, the loss-minimising allocation scales *both* factors as $\sqrt{C}$ — $N \propto \sqrt{C}$ and $D \propto \sqrt{C}$ with $C = 6ND$, the fitted constant working out to roughly $D \approx 20\,N$ **tokens per parameter** (so $N \approx \sqrt{C/120}$, $D \approx \sqrt{C/0.3}$; doubling compute means scaling parameters and tokens equally, not favouring one). This corrected the earlier Kaplan et al. finding that larger N was always preferable. The Chinchilla paper reports that a 70B parameter model trained on 1.4T tokens achieves lower loss than a 280B model trained on 300B tokens at the same compute budget.

Honest epistemics (the U15 question: what IS a rate claim without a lower bound?) These scaling laws are empirical regressions on the data points available in 2020–2022. They are not theorems. They do not come with confidence intervals over out-of-distribution regimes (very large N, different data modalities, different architectures). Subsequent work has found deviations (e.g. "emergent" behaviours at certain scale thresholds that are not captured by smooth power laws; flat regions in loss curves that the power law predicts should be declining). The correct reading — consistent with U15's discussion of empirical rates — is: *these are useful engineering heuristics calibrated on a specific regime; they are not guaranteed laws and they carry no information-theoretic lower bound.* The question "what is the minimum number of parameters needed to achieve loss L^*?" does not have a known rigorous answer; the scaling law gives an empirically-estimated upper bound on the parameter count needed to achieve it at the Chinchilla compute frontier.

Check your reading.
Q5. Under the Chinchilla compute-optimal frontier with C = 6ND, an engineer doubles the compute budget. The loss-minimising reallocation is:
(A) put all the extra compute into parameters N – bigger model, same data (B) put all the extra compute into tokens D – same model, more data (C) scale N and D each by sqrt(2), keeping the tokens-per-parameter ratio at roughly D = 20N (D) set N = D

5b. Mixture-of-experts: the sparse composition (U0L §3, in depth)

U0L introduced the MoE asymmetry — memory holds all parameters, bandwidth pays only the active ones. Here is the mechanism and its distributed consequence.

The mechanism. Replace each feed-forward block by E parallel expert networks $\mathrm{FF}_1, \ldots, \mathrm{FF}_E$ plus a learned **router**: a small linear map producing scores over experts, of which the top-k (typically $k \in \{1, 2\}$) are evaluated and their outputs combined by the softmaxed scores. The token's compute is k/E of the dense block's; its memory footprint is unchanged. The router is trained jointly, with an auxiliary **load-balancing loss** penalising concentration — because a router that sends everything to one expert recreates a dense model on one shard while the rest idle. Load imbalance is a balls-in-bins tail (F.S2; E0L.4), and its cost is a straggler makespan (U3): the three lenses meet in one design constraint.

The distributed consequence: expert parallelism. Shard whole experts across machines (Figure 4.2c). Per token, the network carries the token's activation to its expert's machine and back — kilobytes — instead of synchronising weight shards. In α-β terms (F.E4): each MoE layer does two all-to-alls (dispatch the tokens to their experts, combine the results), each moving about $k\,a$ bytes per token (k experts per token, a activation bytes), so a batch of B tokens costs about $2(\alpha + \beta\, k\, a\, B)$ per MoE layer, times the number of MoE layers. It scales with the activation size, not with the expert *weights*, where TP pays a per-layer sync scaling with the hidden dimension. This is why MoE re-prices hardware (U0L §3's boxed fact): interconnect quality buys dense models throughput; memory capacity buys MoE models existence.

What MoE costs statistically. Sparse activation is a modelling assumption — the claim that token-conditional computation can specialise. The honest ledger (U15 epistemics): MoE models dominate dense equals-active-compute baselines on the measured regimes, but the specialisation the router learns is an empirical artifact, not a theorem; interpretability of expert roles is folklore-grade. STATED, not certified.

6. Inference and serving: KV-cache, batching, quantisation, and the engine

6.1 KV-cache memory accounting (U4 memory lens)

At inference, the model generates one token at a time autoregressively. At step t, it must attend over all $t-1$ previous tokens. Re-computing keys and values for all previous tokens at each step would cost $O(T^2)$ total work; instead, the **KV cache** stores the key and value tensors for every token at every layer.

Exact accounting. For a transformer with L layers whose attention keeps n_{kv} key/value heads, each of dimension d_{head}, a KV cache for T tokens (at precision b bytes per float) costs:

$$\text{KV-cache bytes} \;=\; 2 \cdot L \cdot n_{kv} \cdot d_{\text{head}} \cdot T \cdot b,$$

where the factor 2 covers both K and V. With full multi-head attention every head keeps its own K and V, so $n_{kv} = h$ and $n_{kv} d_{\text{head}} = d$ (the hidden dimension). For a 70B-class model with $L = 80$, $d = 8192$ (64 heads of 128), at $T = 4096$ tokens and $b = 2$ bytes (bf16):

$$\text{full MHA: } 2 \times 80 \times 8192 \times 4096 \times 2 \approx 10.7 \text{ GB per sequence.}$$

Llama-2/3 70B, which has exactly these L and d, uses *grouped-query attention*: the 64 query heads share $n_{kv} = 8$ key/value heads of 128, so $n_{kv} d_{\text{head}} = 1024$ and

$$\text{GQA: } 2 \times 80 \times 1024 \times 4096 \times 2 \approx 1.34 \text{ GB per sequence,}$$

eight times less: shrinking the cache is exactly why GQA exists. This memory is on top of the model weights (≈ 140 GB at bf16 for 70B params). The KV cache scales linearly in T: doubling the context window doubles the cache. This is the U4 memory lens instantiated precisely.

6.2 Batching, throughput, and latency

Inference admits two operating regimes:

- **Latency-optimised (interactive):** serve one request at a time; minimise time-to-first-token and per-token latency. Memory-bound (small batch, underutilised compute).
- **Throughput-optimised (batch):** pack as many requests as possible into a batch; maximise tokens generated per second. Compute-bound (large batch, high GPU utilisation).

Continuous batching (Orca, Yu et al. 2022) allows requests at different completion stages to share a batch dynamically: new requests are inserted as completed requests leave, without padding or waiting for a full batch to finish. This is a scheduling policy (U3) for a heterogeneous-completion-time workload, and it substantially increases throughput over static batching.

6.3 Quantisation (one paragraph, empirical)

Post-training quantisation (PTQ) represents weights and/or activations at reduced precision: INT8 (8-bit) or INT4 (4-bit) instead of fp16/bf16. For a 70B model: INT8 reduces weights from $2N$ bytes to N bytes; INT4 reduces to $N/2$ bytes (35 GB), enabling inference on consumer hardware. The performance degradation is empirically observed to be small for INT8 (perplexity increase $< 1\%$ on standard benchmarks for most model families) and larger for INT4 (varies by model and task, typically 1–5% perplexity increase). There is no proven lower bound on the quantisation-induced loss gap; the empirical results are model- and task-dependent. Quantisation-aware training (QAT) recovers some of the gap by simulating low-precision arithmetic during training. We state these facts as engineering observations, not theorems. In U0L §2's ceiling formula, quantisation is the *denominator dial*: halving bytes-per-parameter doubles the decode ceiling before any hardware changes — which is why it is the first lever every serving stack pulls.

Figure 24.3 shows the two dials: fewer bytes per weight, and fewer key/value heads for the cache.

6.4 The serving engine, assembled (U0L §5, in depth)

A production serving engine (the vLLM-class systems of U0L's framework table) is the course's systems units composed one final time: **continuous batching** admits and retires requests mid-flight, packing the accelerator like U3 packs machines; **paged KV-cache** manages attention state in fixed-size blocks — virtual memory's page table (F.C1's memory hierarchy) rebuilt inside an inference server, eliminating fragmentation as requests of wildly different lengths come and go; and the **parallelism plan** (Figure 4.2: TP within a node or fast island, PP across memory

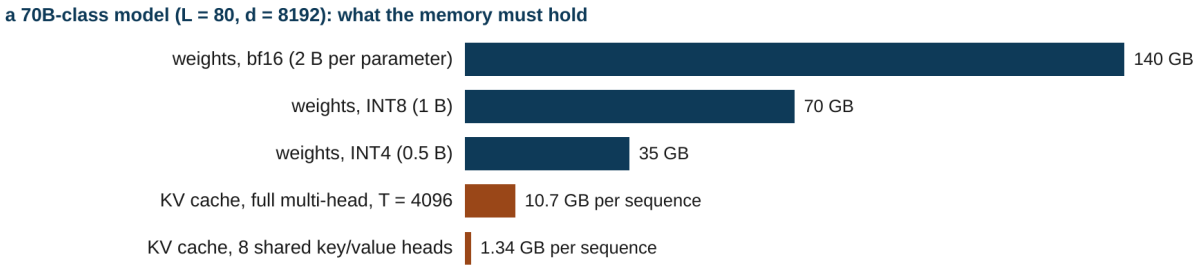


Figure 24.3: Two dials: fewer bytes per weight, and fewer key/value heads for the KV cache.

boundaries, EP for MoE) is chosen exactly by the α-β accounting of E0L.3. The engine's identity in F.E6's taxonomy: *maximise tokens per second subject to a latency objective* — a fourth engine shape alongside batch, actor, and streaming. Lab 7 measures the decode rate on your own laptop against its predicted ceiling, and from that measurement prices the pipeline and tensor cuts for two laptops; expert parallelism needs a mixture-of-experts model and is not exercised.

Check your reading.

Q6. A 13B-parameter transformer with full multi-head attention (L = 40 layers, hidden dimension d = 5120) serves T = 8192-token contexts in bf16 (b = 2 bytes). Its KV cache, by the unit's formula 2 * L * n_{kv} * d_{head} * T * b with n_{kv} * d_{head} = d, costs about:
(A) 3.4 GB (B) 6.7 GB (C) 13.4 GB (D) 26 GB

Q7. Your lab group must serve an open-weight model to the whole course for a week on a single 80 GB GPU – the course's running LLM workload, now yours to operate. Using section 6's vocabulary: which operating regime do you target, which two levers do you pull first, and what does each lever cost? Name the course unit (or foundation) behind each lever.

7. LoRA: low-rank adaptation (U7 lens)

Low-Rank Adaptation (LoRA) (Hu et al. 2021) fine-tunes a pre-trained model by adding a low-rank perturbation to each weight matrix. For a pre-trained weight $W_0 \in \mathbb{R}^{d\times k}$, instead of updating W_0 directly (at cost dk parameters), LoRA adds:

$$W_0 + \Delta W, \quad \Delta W = BA, \quad B \in \mathbb{R}^{d\times r},\ A \in \mathbb{R}^{r\times k},$$

with rank $r \ll \min(d, k)$. Only A and B are trained; W_0 is frozen.

Parameter and memory savings. The original weight has dk parameters; LoRA adds $r(d+k)$ trainable parameters. For $d = k = 4096$ and $r = 16$: $dk = 16.8$ M parameters vs $r(d + k) = 131$ K LoRA parameters — a factor of 128 reduction. The optimiser state (Adam), and the gradients, are kept for the $r(d + k)$ adapter parameters of each adapted matrix instead of its dk: a fraction $r(d + k)/(dk)$, 1/128 here, and less over the whole model when only some matrices are adapted. This is the U7 (SVD/low-rank) lens applied as a practical memory constraint: the hypothesis is that the relevant update direction for a new task lies in a low-dimensional subspace of parameter space, and the rank-r constraint is the regularisation that enforces it.

Bytes saved at runtime. After training, $\Delta W = BA$ can be merged into W_0 for inference with zero overhead. Alternatively, multiple LoRA adapters for different tasks can be hot-swapped on the same base model, each task's adapter storing only $r(d + k)$ parameters per adapted matrix (that many parameters times the bytes per parameter).

8. RAG: retrieval as ANN (U13 lens)

Retrieval-Augmented Generation (RAG) (Lewis et al. 2020) augments an LLM's input context at inference time with documents retrieved from an external corpus. The pipeline:

1. **Embed:** represent each candidate document (and each query) as a dense vector in $\mathbb{R}^m$ using an embedding model.
2. **Retrieve:** given a query vector q, find the k nearest neighbours in the document embedding corpus — an **ANN search** (U13, §5: HNSW or IVF-based indices).
3. **Condition:** prepend the retrieved documents to the LLM's context and generate.

The retrieval quality vs recall tradeoff is exactly the U13 lab quantity: HNSW achieves 90–99% recall at query (measured against exact nearest-neighbour) at $O(\log n)$ expected query time; IVF achieves similar recall at lower memory cost. Both trade recall for speed and memory in ways that U13 characterises. The choice of embedding model, index type, and k are engineering decisions; the U13 lens provides the cost-accounting vocabulary for making them principled rather than heuristic.

8b. Local models: inference and fine-tuning on your own machine

Sections 3–8 priced the model on clusters. This section prices it on one laptop with one user, in two uses. **Inference** runs a fixed model, and U0L §2 showed that it is a bandwidth question. **Fine-tuning** trains the model further on your own data: §4.4's memory accounting and §3.3's FLOP count, made smaller by §7's low-rank trick. The symbols are U0L's: B is the memory bandwidth (not §4.2's batch size), N the parameter count, N_a the active parameters per token, and b the bytes per parameter.

8b.1 Why local

A **local model** is an open-weight model (U0L §1.3) whose weight file runs on hardware you govern. There are five reasons to run one, and each has a price.

- **The data stays on the machine.** Prompts, retrieved documents and training examples never leave it. Who may see the data is a contract question (U00's fourth pillar; U16): a closed-weight API adds a party to every prompt (U0L §1.3), and a local model removes it. You still have to check what you train on: U15 §6b's input-side scrub applies to fine-tuning data too.
- **No per-token bill.** You pay for the machine and its electricity instead, and its throughput is capped. At §8b.2's measured 16.97 tok/s, a day of decoding at batch 1 yields at most $16.97 \times 86\,400 \approx 1.5 \times 10^6$ tokens.
- **Offline.** Once the weights are downloaded, no network is needed.
- **Reproducible runs.** Pin the weights by their content hash: the Lab 7 log names the blob `3d0b790534fe…`, while a tag such as `qwen3:1.7b` is a pointer its publisher can move. Pin the runtime version too and fix the decoding settings, and a rerun on the same machine is expected to give the same output. STATED, not certified: adding floating-point numbers in a different order (another thread count, another machine) can change a token.
- **Licences.** Open-weight licences differ, and some restrict use (U0L §1.3). Check the licence before you build on the weights, and again before you share anything trained from them.

8b.2 Inference: does it fit, and how fast?

Capacity (U4 memory lens). At q bits per parameter the weights take $Nq/8 = Nb$ bytes. The KV cache of §6.1, the runtime and the operating system come on top:

$$Nb \;+\; 2\,L\,n_{kv}\,d_{\text{head}}\,T\,b_{kv} \;+\; (\text{runtime and OS}) \;\leq\; \text{memory}.$$

Worked example. Take an 8B-class model with U0L §2.4's shape ($L = 32$, $d = 4096$). At 4 bits with block scales ($b \approx 0.55$, U0L §2.3) its weights take $8 \times 10^9 \times 0.55 = 4.4$ GB. Its 16-bit KV cache per token is $2 \times 32 \times 4096 \times 2 = 524\,288$ B (0.5 MiB) under full multi-head attention. It is $2 \times 32 \times (8 \times 128) \times 2 = 131\,072$ B (128 KiB) if 32 query heads share 8 key/value heads of 128, as in many current models of this size (this shape is ASSUMED: check the model's configuration). At $T = 8192$ the two caches hold 4.3 GB and 1.07 GB. On a 16 GB laptop, the context length can decide whether the model fits, not only the weights.

Decode is memory-bandwidth-bound. At batch 1, each token reads every active weight once (U0L §2.2):

$$\text{tok/s} \;\leq\; \frac{B}{N_a\,b}.$$

Worked example (one machine, one run, 2026-10-23). The laptop of U0L §2.5 has an Intel Core i5-8350U with two channels of DDR4-2400, so $B = 2 \times 8$ B $\times\, 2.4 \times 10^9$ s$^{-1} = 38.4$ GB/s at peak. It ran `qwen3:1.7b`, a dense model in a 1.4 GB file, natively with Ollama 0.32.15 on the CPU, using Lab 7's script. For a dense model the file size stands in for $N_a b$ (U0L §2.5):

```
ceiling   B / (N_a b) = 38.4 / 1.4                          = 27.4 tok/s
measured  eval: 124 tokens in 7.308 s (second run)          = 16.97 tok/s   (0.62 of
  the ceiling)
```

U0L §2.5 measured the larger `qwen3:4b` on the same laptop at 0.57 of its ceiling. The file holds $1.4 \times 10^9 \times 8/(1.7 \times 10^9) \approx 6.6$ bits for each of the name's 1.7×10^9 parameters, so it is not 4-bit throughout. That is why the measured size, not the nominal bit width, goes in the denominator.

Prefill is compute-bound (U6 roofline). A prompt of T_p tokens shares each weight read among its T_p tokens: the arithmetic intensity is $2T_p/b$, against $2/b$ in decode (U0L §2.1). So prefill costs about $2NT_p$ operations (§3.3) and takes about $2NT_p/R$ seconds at arithmetic rate R. In the same log, the warm-up run read its 41-token prompt in 624 ms. That is 65.7 tok/s, or about $2 \times 1.7 \times 10^9 \times 65.7 \approx 2.2 \times 10^{11}$ operations per second on quantised weights. It is a lower bound, since a first run may include one-off start-up work. In decode the same CPU did $2 \times 1.7 \times 10^9 \times 16.97 \approx 5.8 \times 10^{10}$ operations per second, about a quarter of that: in decode the arithmetic units wait for memory. **Measure prefill with a fresh prompt.** The second run repeated the warm-up's prompt, so the runtime reused its earlier work and reported 707.9 tok/s. Its 57.9 ms is about one decode step ($1000/16.97 = 58.9$ ms), not a prefill.

The quantisation dial and three kinds of memory. Quantisation (§6.3) is the denominator. At 16 bits the same 1.7B model would take $2 \times 1.7 \times 10^9 = 3.4$ GB, and the ceiling would drop from 27.4 to $38.4/3.4 = 11.3$ tok/s. Where the bytes live sets both B and the capacity. Qualitatively:

- **CPU and system RAM:** the most capacity and the least bandwidth (tens of GB/s, U0L §2.3). Unpacking quantised weights also costs CPU compute (U0L §2.5).
- **A laptop's discrete GPU:** it has its own memory, typically faster than the RAM and smaller. A model that does not fit is **split**: some layers in GPU memory, the rest in RAM. At batch 1 the two parts run one after the other, like U0L §4.1's pipeline. With a fraction

f of the bytes on the GPU,

$$t_{\text{token}} \approx \frac{f\,N_a b}{B_{\text{GPU}}} + \frac{(1-f)\,N_a b}{B_{\text{RAM}}},$$

and the slower memory dominates. With $B_{\text{GPU}} = 10\,B_{\text{RAM}}$ (an ASSUMED ratio) and $f = 1/2$, the speed-up over RAM alone is only $1/(0.05 + 0.5) \approx 1.8$.

- **Unified memory** (Apple silicon, for example): the CPU and GPU share one pool, so the GPU's bandwidth applies to a model held in main memory, with no separate, smaller GPU memory to fit into. Read your chip's specification; U0L §2 uses a ~250 GB/s class as illustrative.

CPU only: most laptops, and ours. The laptop of the worked example has no discrete GPU. Its only graphics is integrated (Intel UHD Graphics 620), which has no memory of its own and reads the same RAM, so even a runtime that uses it cannot raise the decode ceiling above $B_{\text{RAM}}/(N_a b)$. Every measurement in this section is therefore of the CPU-only case, and it is the case to plan for. In words: inference is practical (16.97 tok/s for the 1.7B model, and U0L §2.5's 8.7 tok/s for the 4B one), while fine-tuning runs at the CPU's arithmetic rate, the 10^{11} FLOP/s row of §8b.3, where LoRA on a 1.7B model over 10^6 tokens takes about 19 hours. So the CPU-only recipe is: a small model (1–4B) at 4 bits; retrieval (§8b.4) before any training; and, if you must fine-tune, LoRA on a small model with a small dataset, measured first on a few hundred steps so that the hours are a measurement rather than a guess.

Batch 1 (U3 lens). A laptop serving its owner decodes one stream, $m = 1$, at the memory-bound end of §6.2. The engine of §6.4 batches m requests so that one read of the weights serves m tokens. Each step then moves $N_a b + m \cdot$ (one stream's KV bytes): batching shares the cost of the weights but not of the caches. Throughput rises until the intensity $2m/b$ reaches the machine's balance point R/B (U0L §2.1). On a laptop, more streams serve more users; they do not make the first user's stream faster.

8b.3 Training on your own data

Fine-tuning continues training (supervised fine-tuning, U0L §1.2) on your own examples. We price memory first, then compute.

Full fine-tuning. Every parameter trains, so all of §4.4 applies: 16 bytes per parameter under mixed-precision Adam with fp32 master weights ($2 + 2 + 12$), before any activations. For $N = 7 \times 10^9$ that is $16 \times 7 \times 10^9 = 112$ GB, 3.5 to 14 times the 8–32 GB of a typical laptop (an ASSUMED range). A laptop cannot do it.

LoRA (§7, U7 lens). Freeze the base and train a rank-r pair on chosen matrices: $r(d_{\text{in}} + d_{\text{out}})$ parameters per adapted $d_{\text{in}} \times d_{\text{out}}$ matrix (§7's $d \times k$). Gradients and Adam state exist only for these, so §4.4's 16 bytes apply to them alone, and the frozen base can be stored in low precision (2 bytes at bf16). *Worked example* (the 7B-class shape $L = 32$, $d = 4096$, with $r = 16$ on the four $d \times d$ attention projections of every layer; our choice: Hu et al. adapt only W_q and W_v in most of their experiments, which halves the count): $4 \times 32 \times 16 \times (4096 + 4096) = 16\,777\,216 \approx 1.7 \times 10^7$ trainable parameters. That is 0.24% of 7×10^9, with $16 \times 1.68 \times 10^7 \approx 0.27$ GB of state.

QLoRA. Dettmers et al. (2023) keep the frozen base at 4 bits and back-propagate through it into 16-bit LoRA adapters. They report fine-tuning a 65B-parameter model on a single 48 GB GPU while preserving the task performance of full 16-bit fine-tuning (STATED: their abstract, on their benchmarks). Our byte count agrees: $65 \times 10^9 \times 0.5$ B $= 32.5$ GB of 4-bit base fits in 48 GB with room to spare. For our 7B model the base takes $7 \times 10^9 \times 0.5$ B $= 3.5$ GB. It takes about 3.9 GB if each block of 64 weights also carries one 32-bit scale (QLoRA's own first-level format, before double quantisation: $32/64 = 0.5$ bit per weight); their double quantisation shrinks the

scales to $8/64 + 32/(64 \cdot 256) \approx 0.127$ bit per weight, so about $7 \times 10^9 \times 4.127/8 \approx 3.6$ GB (and 33.5 GB for 65B, still well inside 48 GB).

7B model, before activations	frozen base	trainable state	total
full fine-tuning (§4.4)	none	$16N = 112$ GB	112 GB
LoRA, bf16 base, $r = 16$ on attention	$2N = 14$ GB	0.27 GB	14.3 GB
QLoRA, 4-bit base, the same adapters	3.5–3.9 GB	0.27 GB	3.8–4.2 GB

Activations. The backward pass needs the inputs of every matmul and nonlinearity. For §3's block, per token and layer, these are: the inputs and outputs of the two layer norms ($4d$), Q, K, V ($3d$), the attention output (d), and the MLP's hidden layer before and after its nonlinearity ($8d$). That is $16d$ values, or $32d$ bytes at 16 bits. For $L = 32$, $d = 4096$ and one 2048-token sequence, it comes to $32 \times 4096 \times 32 \times 2048 \approx 8.6$ GB, more than twice the 4-bit base. **Activation checkpointing** keeps only each layer's input ($2Ld$ bytes per token, 0.54 GB here) and recomputes the rest one layer at a time (0.27 GB for one layer), about 0.8 GB in all. The price is a second forward pass. This count is ours. It leaves out the attention scores, which FlashAttention-style kernels recompute instead of storing, and the $T \times |\mathcal{V}|$ output logits. A careful implementation stores less for frozen matrices, whose inputs are needed only for their own weight gradients. Measure it (§12, Experiment 3). With checkpointing, QLoRA on a 7B-class model at $T = 2048$ needs about $3.9 + 0.27 + 0.8 \approx 5$ GB before framework overheads: on paper, within a laptop's reach.

Compute: LoRA saves memory, but not most of the compute. Take one linear map $y = xW$ with W of size $d_{\text{in}} \times d_{\text{out}}$. Per token, the forward pass costs $2d_{\text{in}}d_{\text{out}}$ operations. The backward pass computes $\partial\mathcal{L}/\partial x = (\partial\mathcal{L}/\partial y)\, W^\top$ for another $2d_{\text{in}}d_{\text{out}}$ and $\partial\mathcal{L}/\partial W = x^\top(\partial\mathcal{L}/\partial y)$ for a third. Summed over all weights, this is §3.3's $2N + 2N + 2N = 6N$. A frozen W needs no $\partial\mathcal{L}/\partial W$, but $\partial\mathcal{L}/\partial x$ must still pass through it to reach the adapters below, so

$$\text{LoRA FLOPs per token} \;\approx\; 2N + 2N + 6N_{\text{LoRA}} \;\approx\; 4N \qquad (N_{\text{LoRA}} \ll N),$$

which is two thirds of full fine-tuning. Activation checkpointing adds its forward pass to both. QLoRA also dequantises the frozen weights each time it uses them, extra work that we do not price here.

Worked example: the hours. Fine-tune a 1.7B model on $D = 10^6$ tokens (D counts every token processed, over all epochs). LoRA costs $4ND = 4 \times 1.7 \times 10^9 \times 10^6 = 6.8 \times 10^{15}$ FLOPs, and full fine-tuning $6ND = 1.02 \times 10^{16}$.

sustained rate (ASSUMED)	LoRA, $4ND$	full, $6ND$
10^{11} FLOP/s (laptop CPU, fp32 training loop)	6.8×10^4 s $\approx$ 19 h	1.0×10^5 s $\approx$ 28 h
10^{13} FLOP/s (laptop GPU, bf16)	680 s $\approx$ 11 min	1020 s = 17 min

These rates are assumptions, not measurements. We chose them ten times below §3.3's illustrative classes (10^{12} and 10^{14}); the prefill rate measured above, using different arithmetic, is the same order as the CPU row. Measure your own. Full fine-tuning also needs $16 \times 1.7 \times 10^9 = 27.2$ GB of training state, so on a laptop GPU memory rules it out before time does.

Overfitting and evaluation (U15 honesty lens). Private datasets are small: $N_{\text{LoRA}} \approx 10^7$ free parameters against a few thousand examples leaves room to memorise them rather than learn from them. Split the data before you train: a training set, a validation set for choices (rank, epochs, learning rate), and a **held-out test set** that you use once. Its score has sampling

error. For a metric in $[0,1]$ averaged over n examples, Hoeffding's inequality (F.S2 §4) gives the half-width $\varepsilon = \sqrt{\ln(2/\delta)/(2n)}$ at confidence $1-\delta$. At $\delta = 0.05$ that is about ± 0.136 for $n = 100$ and ± 0.043 for $n = 1000$, and by this bound a smaller gain is not evidence. This is U0L §6's rule: a quality claim is an agreed test on held-out data with a known error distribution (U16). Report the score with its n and its interval, and label the claim STATED unless such a test backs it.

8b.4 Retrieve before you train

When the request is "the model should know my documents", reach for retrieval (§8, U13) before training:

- **No training at all**, with an off-the-shelf model. Embed the documents once, index them (U13's ANN), and put the top k into the prompt when a question arrives. (Lewis et al. also fine-tune the generator and the query encoder end to end; retrieval into a frozen model's prompt needs neither.)
- **The data stays outside the weights.** What the index holds can be inspected, corrected, deleted and access-controlled. What a fine-tune has absorbed cannot be taken out, short of training again.
- **Updates are instant.** A new document is one more vector in the index.

The practice that follows is **fine-tune for style or format, retrieve for facts**, STATED as practice, not a theorem. On a laptop, retrieval moves cost from training to prefill. Five retrieved passages of 400 tokens add 2000 prompt tokens, about $2N \times 2000$ operations. At the 65.7 tok/s prefill measured above, that is about 30 s before the first token, plus 2000 tokens of KV cache (§6.1).

8b.5 What Lab 7 measures

Lab 7 runs the inference half of this section on your own laptop. It serves `qwen3:1.7b` natively, if Ollama is installed, and in the `ollama/ollama` container on the CPU. You:

1. predict the decode ceiling $B/(N_a b)$ from your memory specification and the file size, before running anything;
2. measure the decode rate both ways and explain the gap (U0L §2.5);
3. price two laptops from your time per token t: the pipeline at about t, and tensor parallelism at $t/2 + 2L\,s$ with $L = 28$, worth it only if $s < (t/2)/(2L)$ (U0L §4.1).

The lab measures the memory axis (decode) and prices the communication axis (PP against TP). It trains nothing: fine-tuning on a laptop is §12's Experiment 3. Its `prefill_tok_s` comes from a repeated prompt, so read it with §8b.2's caveat.

9. Watermark detection as Scheffé–Yatracos (U15 security capstone)

Statistical watermarking schemes for LLM-generated text work by perturbing the token-sampling distribution at generation time. A **keyless detector** (our setting) must distinguish the watermarked distribution Q from the unwatermarked distribution P without access to the secret key that parameterises the perturbation.

The core statistical quantity is the **total variation (TV) distance**:

$$\mathrm{TV}(P,Q) \;=\; \frac{1}{2}\|P-Q\|_1,$$

the L1 distance between the two token distributions (U15: the Scheffé–Yatracos identity). The **minimum-distance estimator (MDE)** of U15 selects the hypothesis (P or Q) whose empirical distribution is closer in TV to the observed token sequence. The Scheffé set $A^* = \{x : Q(x) > P(x)\}$ achieves the maximum distinguishing power with a single test — a consequence of the Neyman–Pearson lemma restated in the L1 distance.

Our preprint (*"A Watermark for All, Formally"*, VakeWorks / IAS Continuum, 2026) formalises this connection and derives a **keyless difference-in-differences detector**: compare the empirical frequency of tokens in the Scheffé set A^* under an observed sequence against the frequencies predicted by P and Q. The detection power is controlled by $\mathrm{TV}(P, Q)$ — the *maximum* advantage any single test can achieve (Scheffé), attained by the A^* test above. The Lean certificates (`provedog/IasCert/Watermark/`) machine-check the Scheffé identity $\mathrm{TV} = \frac{1}{2}\|P - Q\|_1$ and the difference-in-differences detector's soundness (U15's `ScheffeFinite` / `DiD` in the provedog ledger) — the certified claim is that the optimal test's advantage *equals* TV, not a universal bound on every test's power. This is the security thread of the course brought to closure: U15's statistical machinery, applied to the problem of detecting AI-generated text, certifiably.

10. Agentic systems: awareness, and the link back to U16 and U17

Agent loops combine an LLM's conditional generation capability with external tools and persistent memory: the model generates a next action (a tool call, a retrieval request, a response), the environment executes it and returns an observation, and the model conditions on the extended context to generate the next action. The loop is a state machine; the LLM is the transition function. U17 built the governance around such loops: who may act, what is remembered, and why the reports can be trusted.

This is tool-agnostic. Protocols for communicating tool descriptions to models (MCP; Anthropic 2024) and for composing multiple agents (A2A; Google 2025, now a Linux Foundation project) are named here as existence proofs that the ecosystem has standardised on this architecture — we do not endorse or depend on any specific vendor's implementation.

The incentive question that remains open. When multiple autonomous agents operate in the same environment — bidding for resources, selecting from shared tool endpoints, or rating each other's outputs — the question of whether their collective behaviour produces good outcomes is a **mechanism design** question, not an optimisation question. An agent loop that maximises expected reward in isolation may behave poorly in a multi-agent equilibrium. This is the question of U16 (Statistical Contract Theory): the statistical guarantees that a principal can extract from a strategic agent, the incentive-compatible mechanisms for eliciting truthful information, and the conditions under which distributed learning agents converge to socially optimal outcomes; U17 §9 brought it to bear on agent systems (why report honestly?). The arc of the course closes here: from the cost of one operation (U0) to the incentive structure of systems of learning agents (U16, U17).

11. The invariant: all five lenses at once

> **RAM (U9 / U14 / U15):** an LLM is an autoregressive conditional density estimator $p_\theta(x_t \mid x_{<t})$; training minimises empirical cross-entropy = an M-estimation problem; forward cost $\approx 2N$ FLOPs/token; training cost $\approx 6ND$.
>
> **PRAM (U6 / U1):** each transformer layer = matmul + softmax compositions;

attention work $O(T^2 d)$, depth $O(\log T + \log d)$; ring all-reduce = U1's reduction tree, bandwidth-optimal at $2Nb$ bytes/device.

DPRAM/MPC (U0 / U3 / U4 / U6 / U13): distributed training requires all four parallelisms simultaneously — data (ring all-reduce, $2Nb$), tensor (4 all-reduces per layer per step, 2 forward and 2 backward, $O(Bd)$ each), pipeline (bubble $(p-1)/(m+p-1)$), ZeRO (memory $16N/p$, communication about $3Nb$); MoE routing = load-balancing with auxiliary loss; inference = KV-cache bytes $2 \cdot L \cdot d \cdot T \cdot b$, batching = scheduling (U3); LoRA = low-rank (U7); RAG = ANN (U13).

Statistical (U14 / U15): scaling laws = empirical rates, no lower bound, honest; watermark detection = Scheffé–Yatracos = U15's MDE applied to token distributions, machine-checked.

Incentive / societal (U16, U17): multi-agent systems raise mechanism-design questions that optimisation alone cannot answer; SCT is the right lens.

The point of the box: none of these five items requires a new concept. They all appeared in units 0–17. This unit exists to name that fact.

12. Lab — project track

Project-track lab (no weekly receipt): laptop-runnable in the course lab image (`labs/image/`), as deep-dive and group-project material. Ray for distributed experiments; no paid APIs; only open-weight models from Hugging Face.

Experiment 1 — Ring all-reduce bytes: measure vs formula. Use Ray's `collective.allreduce` on a cluster of 2–4 Ray actors. For a mock gradient tensor of size $N \in \{10^6, 10^7, 10^8\}$ parameters (float32), measure bytes sent and received per actor per all-reduce using Ray's network stats or a custom byte counter. Compare to the formula $2(p-1)/p \cdot Nb$ and to the naive all-gather baseline (every actor sends its whole gradient to each of the other $p-1$ and sums locally: $(p-1)Nb$ bytes per actor, growing with p where the ring's stays below $2Nb$). Deliverable: table of measured vs formula bytes; one-paragraph cost-accounting in the three-lens vocabulary.

Experiment 2 — KV-cache memory accounting: empirics vs formula. Load a small open-weight transformer (e.g. SmolLM-360M or GPT-2 small) using Hugging Face `transformers`, with the key/value cache enabled (the model returns it as `past_key_values`). For sequence lengths $T \in \{128, 512, 1024, 2048\}$ (up to the model's context length: stop at 1024 for GPT-2 small), measure the memory the cache tensors hold: on a GPU, the peak from `torch.cuda.memory_stats()` minus the same peak with the cache disabled (the peak counts the weights and activations too); on the CPU, the summed `nbytes` of the returned key and value tensors. Compare to §6.1's formula $2 \cdot L \cdot n_{kv} \cdot d_{\text{head}} \cdot T \cdot b$, reading L, n_{kv} and d_{head} from the model's configuration: with full multi-head attention (GPT-2 small) $n_{kv} d_{\text{head}} = d$, while a grouped-query model has fewer key/value heads, and $2 \cdot L \cdot d \cdot T \cdot b$ over-counts its cache. Where do the measured values deviate (overhead from attention masks, activation checkpointing, padding)? Deliverable: plot of measured vs formula memory; cost-accounting paragraph.

Experiment 3 — LoRA fine-tune of a tiny open model. Using Hugging Face PEFT, apply LoRA ($r \in \{4, 16\}$) to GPT-2 small ($N \approx 117$M params) for a simple classification or generation task (e.g. sentiment fine-tune on SST-2 or a small QA dataset from the Hugging Face hub). Measure: (a) trainable parameter count vs total; (b) optimiser state memory with and without LoRA; (c) final task accuracy vs full fine-tune baseline. Compute the theoretical parameter savings from the formula $r(d+k)/dk$ and compare to the measured values. Deliverable: training

log + cost-accounting in the three-lens vocabulary.

13. Lean certificate target

Pipeline bubble fraction (SCHEMATIC — pure combinatorics; certifiable).

The pipeline bubble fraction derivation is a scheduling calculation with no probabilistic component — it is a counting argument over a finite schedule. The Lean statement:

```
-- SCHEMATIC (names align to BMLSPar.lean conventions at cert time):
-- p : ℕ -- number of pipeline stages
-- m : ℕ -- number of micro-batches (m > 0)
-- Each stage takes unit time; the pipeline is fill-drain (GPipe schedule).
-- useful_steps := m * p          -- total stage-steps doing real work
-- total_steps  := (m + p - 1) * p  -- total stage-steps scheduled
-- bubble_steps := total_steps - useful_steps = p * (p - 1)
-- bubble_frac  := (p - 1) / (m + p - 1)

theorem pipeline_bubble_frac (p m : ℕ) (hp : 0 < p) (hm : 0 < m) :
    bubble_steps p m = p * (p - 1) ∧
    m * p + p * (p - 1) = (m + p - 1) * p := by
  -- SCHEMATIC, NOT CHECKED: a bare `ring` does not close this. Split the ∧
    (constructor);
  -- in ℕ, p - 1 and m + p - 1 are truncated subtractions, so first write p = q +
    1 (from hp),
  -- which turns them into q and m + q; the second conjunct is then a polynomial
    identity (ring),
  -- and the first follows from it (unfold bubble_steps;
    Nat.add_sub_cancel_left).
```

The two equalities are pure arithmetic, so the statement is certifiable. However, it is not certified as written: `bubble_steps` must be defined, the conjunction split, and the truncated subtraction of $\mathbb{N}$ discharged using $0 < p$ before `ring` applies. The semantic content (that `bubble_steps` correctly models the GPipe schedule) requires an additional model of the pipeline that students define in the lab. The schematic is the combinatorial core; the modelling is the Module-3 exercise.

14. Sources (credited)

Vaswani, A., Shazeer, N., Parmar, N., et al. (2017). *Attention Is All You Need.* NeurIPS 2017. The transformer architecture: scaled dot-product attention, multi-head attention, positional encoding, the encoder-decoder structure. Our exposition adapts the notation.

Kaplan, J., McCandlish, S., Henighan, T., et al. (2020). *Scaling Laws for Neural Language Models.* arXiv:2001.08361. The power-law scaling relationships between compute, data, parameters, and loss. Our empirical-rates paragraph cites this paper.

Hoffmann, J., Borgeaud, S., Mensch, A., et al. (2022). *Training Compute-Optimal Large Language Models.* NeurIPS 2022 (Chinchilla). The compute-optimal $N \propto D$ frontier. Our §5 cites this paper as the correction to Kaplan et al.

Shoeybi, M., Patwary, M., Puri, R., et al. (2019). *Megatron-LM: Training Multi-Billion Parameter Language Models Using Model Parallelism.* arXiv:1909.08053. Tensor parallelism: column-parallel and row-parallel linear layers; two all-reduces per layer in the forward pass and two in the backward (as restated in Narayanan et al. 2021, §2.3 and §3.2).

Rajbhandari, S., Rasley, J., Ruwase, O., and He, Y. (2020). *ZeRO: Memory Optimizations Toward Training Trillion Parameter Models.* SC 2020. The ZeRO optimizer stages 1–3 and the memory vs communication tradeoff. Our byte-accounting in §4.4 follows this paper.

Huang, Y., Cheng, Y., Bapna, A., et al. (2019). *GPipe: Efficient Training of Giant Neural Networks using Pipeline Parallelism.* NeurIPS 2019. The fill-drain pipeline schedule; the bubble fraction formula. Our derivation in §4.3 is our own; the result matches §2.3 of this paper.

Shazeer, N., Mirhoseini, A., Maziarz, K., et al. (2017). *Outrageously Large Neural Networks: The Sparsely-Gated Mixture-of-Experts Layer.* ICLR 2017. The MoE routing mechanism and capacity factor.

Fedus, W., Zoph, B., and Shazeer, N. (2021). *Switch Transformers: Scaling to Trillion Parameter Models with Simple and Efficient Sparsity.* JMLR 23(1). Top-1 routing; auxiliary balance loss; the capacity-factor–load-imbalance tradeoff.

Hu, E., Shen, Y., Wallis, P., et al. (2021). *LoRA: Low-Rank Adaptation of Large Language Models.* ICLR 2022. The $\Delta W = BA$ parameterisation; the rank-r hypothesis.

Dettmers, T., Pagnoni, A., Holtzman, A., and Zettlemoyer, L. (2023). *QLoRA: Efficient Finetuning of Quantized LLMs.* NeurIPS 2023; arXiv:2305.14314. A frozen 4-bit base with 16-bit LoRA adapters; §8b.3 states the abstract's 65B-on-one-48 GB-GPU claim as STATED.

Lewis, P., Perez, E., Piktus, A., et al. (2020). *Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks.* NeurIPS 2020; arXiv:2005.11401. Retrieval into the generator's context; cited in §8 and §8b.4.

Narayanan, D., Shoeybi, M., Casper, J., et al. (2021). *Efficient Large-Scale Language Model Training on GPU Clusters Using Megatron-LM.* SC '21; arXiv:2104.04473. The interleaved pipeline schedule cited in §4.3 and E18.5.

Yu, G.-I., Jeong, J. S., Kim, G.-W., Kim, S., and Chun, B.-G. (2022). *Orca: A Distributed Serving System for Transformer-Based Generative Models.* OSDI '22. Iteration-level scheduling, the continuous batching of §6.2.

VakeWorks / IAS Commons Continuum (2026). *"A Watermark for All, Formally."* R. Sainudiin et al., *"A Watermark for All, Formally"* (self-hosted preprint, 2026), with machine-checked Lean certificates accompanying the course materials. The Scheffé–Yatracos watermark-detection capstone in §9 follows that paper.

Wikipedia. *Transformer (machine learning model)*, *Large language model*, *Mixture of experts*, *Ring all-reduce*, *Low-rank adaptation*, *Retrieval-augmented generation.* Background and cross-references; not cited as primary sources.

15. Exercises

Light (examined core):

E18.1 *(6ND estimate)* A transformer language model has $L = 96$ layers, hidden dimension $d = 12\,288$, and is trained on $D = 1.4 \times 10^{12}$ tokens. (a) Estimate the parameter count N using the approximation $N \approx 12Ld^2$ (ignore embeddings). (b) Using the $6ND$ formula, compute the training FLOPs. (c) If training runs on a cluster of 1000 A100 GPUs (each at 312×10^{12} FLOPs/s theoretical peak, assume 40% utilisation), compute the training time in days. (d) State which term — model size or dataset size — dominates the FLOPs estimate, and relate this to the Chinchilla compute-optimal frontier ($N \propto D$).

E18.2 *(Ring all-reduce)* A data-parallel training job runs on $p = 64$ GPUs. The model has $N = 7 \times 10^9$ parameters in bf16 (2 bytes each). (a) Compute the bytes per GPU per all-reduce step using the formula $2(p-1)/p \cdot Nb$. (b) How does this change as $p \to \infty$? (c) The GPU cluster has inter-node bandwidth of 400 Gb/s and intra-node NVLink bandwidth of 600 GB/s. Compute the all-reduce time for the gradient assuming all communication is inter-node. Compare to a single forward pass time at the peak throughput from E18.1(c).

E18.3 *(KV-cache)* A 13B-parameter transformer ($L = 40$, $d = 5120$, bf16) serves long-context inference at $T = 8192$ tokens. (a) Compute the KV-cache size using the formula $2 \cdot L \cdot n_{kv} \cdot d_{\text{head}} \cdot T \cdot b$ of §6.1 with full multi-head attention ($n_{kv} d_{\text{head}} = d$), i.e. $2 \cdot L \cdot d \cdot T \cdot b$. (b) The GPU has 80 GB HBM. Compute the model weight memory ($2N$ bytes at bf16). Is there room for the KV cache? (c) Propose one mitigation (quantise the KV cache to INT8; or reduce T; or use grouped-query attention reducing the cache: with h query heads sharing n_{kv} key/value heads, it shrinks the cache by the factor h/n_{kv}) and compute the resulting memory budget.

E18.4 *(LLM as density estimator)* Let $\mathcal{V} = \{0, 1\}$ (binary tokens) and $T = 3$. A language model assigns $p_\theta(x_1, x_2, x_3)$ to each of the 8 sequences. (a) Write the autoregressive factorisation $p_\theta(x_1, x_2, x_3)$ explicitly. (b) What constraint must the 8 probabilities satisfy (normalisation)? (c) Is minimising empirical cross-entropy on a training set of 4 sequences guaranteed to recover the true distribution p^*? State one condition under which it would (hint: the M-estimation view of §2.2; U9 §4.1 gives the finite-sum objective).

Deep (Module-3 / optional):

E18.5 *(Pipeline bubble fraction — derive and extend)* (a) Derive the GPipe bubble fraction $(p-1)/(m+p-1)$ from first principles using the fill-drain schedule: draw the pipeline timing diagram for $p = 3$, $m = 4$; identify bubble steps; count total steps. (b) Show that the bubble fraction approaches 0 as $m \to \infty$ and approaches $(p-1)/p$ as $m \to 1$ (single micro-batch). (c) Interleaved pipeline scheduling (Narayanan et al. 2021) with v virtual stages reduces the bubble fraction by a factor of v to $(p-1)/(vm+p-1)$, at the cost of v times more peer-to-peer communication. For $p = 8$, $m = 16$, compute the bubble fraction for $v = 1, 2, 4$ and the corresponding increase in communication. (d) State this as a scheduling tradeoff in the U3 vocabulary: what does v correspond to in the preemptive scheduling language?

E18.6 *(Total bytes per step under combined data + tensor parallelism)* A model with $N = 70 \times 10^9$ parameters is trained with data parallelism over $d_p = 8$ nodes and tensor parallelism over $t = 4$ GPUs per node (total $d_p \cdot t = 32$ GPUs). Each layer's weight matrix has $d = 8192$.

(a) **Tensor-parallel communication** (per layer, per step): each of the four all-reduces per layer (two forward, two backward, §4.2) sends $B \cdot d$ elements (batch size $B = 1024$ tokens) across $t = 4$ GPUs in bf16. Compute the bytes per layer per step. With $L = 80$ layers, compute the total tensor-parallel communication per step.

(b) **Data-parallel all-reduce**: after the backward pass, perform a ring all-reduce of the gradient across $d_p = 8$ nodes. Compute bytes per GPU per step using the formula $2(d_p - 1)/d_p \cdot N' \cdot b$, where $N' = N/t$ is the parameter shard on each GPU.

(c) Add (a) and (b) to obtain total communication per GPU per training step. Compare to the case of pure data parallelism ($d_p = 32, t = 1$) for the same total GPU count. Under what regime does combined data+tensor parallel reduce total communication?

(d) Account for ZeRO Stage 2 on top of data parallelism (§4.4): the gradient all-reduce of (b) becomes a reduce-scatter (each data-parallel rank keeps only its $1/d_p$ shard of the gradient and updates only its shard of the parameters and optimiser state), followed by an all-gather of the updated parameters. Using the ring cost $(d_p - 1)/d_p \cdot N' \cdot b$ for each, recompute the per-GPU communication and compare with (b) and (c). What, then, does

Stage 2 buy? Compute the bytes of model state each GPU holds with and without it (mixed-precision Adam, §4.4).

E18.7 *(Watermark detection power)* Let P be the unwatermarked distribution and Q the watermarked distribution over $\mathcal{V}^T$ (token sequences of length T). Let $A^* = \{x : Q(x) > P(x)\}$ be the Scheffé set.

(a) Show that $Q(A^*) - P(A^*) = \mathrm{TV}(P, Q)$ (this is the Scheffé–Yatracos identity; the proof is the same as U15 but you should write it out for this discrete setting).

(b) A keyless detector observes n token sequences and tests $H_0 : P$ vs $H_1 : Q$ by comparing $\hat{Q}(A^*) - \hat{P}(A^*)$ to a threshold τ. By Hoeffding's inequality, bound the probability of a false negative (failing to detect the watermark when it is present) in terms of n, T, and $\mathrm{TV}(P, Q)$.

(c) The Lean certificate in `provedog/IasCert/Watermark/` verifies the Scheffé identity (L0 in the ledger) and the rate-union structure (L5/L6b). Load the certificate; read the statement of L0. What hypothesis is carried as an axiom rather than proved (hint: the per-set Hoeffding tail instantiation is flagged OPEN in the ledger)? Why is this an honest statement of the current state of the formalisation?

Local models (§8b):

E18.8 *(Does it fit, and how fast?)* A laptop has 16 GB of RAM and two channels of DDR5-5600; keep 4 GB for the operating system and the runtime. It runs an 8B dense model with $L = 32$ layers and 32 query heads that share 8 key/value heads of $d_{\text{head}} = 128$ ($d = 4096$). (a) Compute B (U0L §2.3). (b) Compute the weight bytes at 16, 8 and 4 bits (take $b = 2, 1, 0.55$) and say which fit. (c) Compute the 16-bit KV cache at $T = 32\,768$ tokens, with grouped-query attention and with full multi-head attention. Which weight widths still fit with each? (d) For the 4-bit model, compute the decode ceiling at a nearly empty context and at $T = 32\,768$ with grouped-query attention. (e) A second laptop has a discrete GPU whose memory holds 60% of the 4-bit model's bytes. Its bandwidth is 8 times the RAM's. Compute the decode speed-up over RAM alone, and the fraction of the bytes the GPU must hold for a 4× speed-up. *Hint:* §6.1 and U0L §2.4 for the cache; §8b.2's split formula for (e).

E18.9 *(Fine-tuning memory on a laptop)* A 7B model ($N = 7 \times 10^9$) has the shape $L = 32$, $d = 4096$, with an MLP of width $4d$ (§3.2). Apply LoRA with $r = 16$ to the four attention projections and to both MLP matrices of every layer. (a) Count the trainable parameters, and give them as a fraction of N. (b) Compute their training state at 16 bytes each (§4.4). (c) Compute the bytes before activations for full fine-tuning, LoRA on a bf16 base, and QLoRA on a 4-bit base with one 32-bit scale per 64 weights. (d) The laptop of E18.8 again: which recipes fit, and does a 2048-token sequence fit with and without activation checkpointing? (e) Name the one measurement you would make first to test your count. *Hint:* $r(d_{\text{in}} + d_{\text{out}})$ per matrix; activations as counted in §8b.3.

E18.10 *(Fine-tuning compute, and whether it helped)* (a) Activation checkpointing recomputes each layer's forward pass once during the backward pass. Write the FLOPs per token for full fine-tuning and for LoRA, with and without it, and the LoRA-to-full ratio in each case. (b) You have 5×10^5 tokens of private text and train for 3 epochs with LoRA on a 1.7B model. Compute the FLOPs and the time at the two ASSUMED rates of §8b.3, with and without checkpointing. (c) You keep 200 of your 2000 labelled examples as a held-out test set. Compute the 95% Hoeffding half-width of the test accuracy. (d) The fine-tuned model scores 6 points above the base model on those 200 examples. Is that evidence of improvement by this bound? How many test examples would bring the half-width down to 3 points? (e) Why must the rank and the number of epochs be chosen on a validation set and not on the test set? *Hint:* §8b.3's

two-line derivation, plus one forward pass; F.S2 §4's recipe.

Chapter 25

19 admm consensus

U19 — Optimisation II: duality, ADMM & consensus

The ScaDaMaLe book, unit 19 (deep/optional, Part II). Our own exposition; original sources credited in §10. Prerequisite: U8 (projection theorem, dual decomposition sketched) + U9 (GD/SGD, HOGWILD!, variance reduction). This unit does NOT repeat the projection theorem or dual-ascent derivation from U8; it opens from the point at which U8 showed dual decomposition breaks down.

The one organising idea of this unit: *decentralised optimisation is not a hack on top of centralised optimisation; it is a principled second dual variable that turns coupling constraints into per-block subproblems. ADMM is the algorithm that makes the dual update robust enough to matter at scale.*

Entry kit — what this chapter assumes; follow a link if you need the memory jog (map: Figure 1):
F.M3 (the Lagrangian-as-price reading of duality)

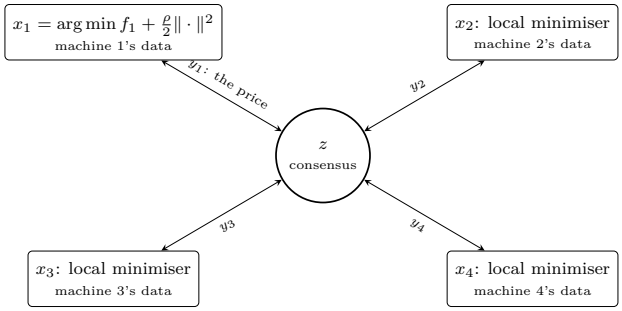
1. Intuition — why a second agent appears in the dual

Recall from U8: a separable objective with coupling constraints decomposes over the primal blocks, and the dual variable carries the coordination signal. That works when the dual function is smooth. For non-smooth regularisers — the lasso penalty $\lambda\|x\|_1$ being the canonical example — the dual gradient does not exist everywhere, dual ascent stalls, and U8's recipe breaks.

ADMM's insight (Boyd–Parikh–Chu–Peleato–Eckstein 2011): augment the Lagrangian with a proximal term in the primal. The augmentation restores strong convexity locally, making the dual update well-defined even when the original objective is non-smooth. The cost is that the augmented Lagrangian is no longer separable — so a second variable z is introduced whose role is to hold the constraint, while x minimises the augmented objective locally. The x–z–u iteration is the algorithm.

The same structure, re-read for a cluster: each machine holds its own data shard and optimises its own x_i in parallel; z is the global model that lives on a coordinator (or is computed by averaging); u_i is machine i's dual variable tracking the disagreement between x_i and z. This is consensus ADMM — the canonical distributed ML template.

Figure 25.1 is the algorithm as a market: proposals in, prices out, until agreement.



iterate: (1) each machine minimises *locally*; (2) the hub averages: $z \leftarrow \overline{x}$;
(3) each price updates: $y_i \leftarrow y_i + \rho\,(x_i - z)$ — rising wherever a machine disagrees with the consensus

Figure 25.1: The ADMM consensus star: each machine holds its own data and repeatedly solves a small local problem, the hub averages the proposals into a consensus z, and the dual variable on each edge is a *price* (F.M3 §4) that rises as long as that machine keeps disagreeing — until every local minimiser voluntarily agrees. No raw data ever leaves a machine; only proposals and prices travel — one O(model size) exchange per round. Federated fine-tuning (U0L) is this star with local models at the leaves.

2. Convex duality: compact recap

We collect here only what U19 adds. U8 stated the projection theorem and dual ascent; read that first.

2.1 Lagrangian and dual function

For a primal problem $\min_x f(x)$ subject to $g(x) \le 0,\ h(x) = 0$ (all functions real-valued; not yet assuming convexity):

Lagrangian:

$$\mathcal{L}(x, \lambda, \nu) \;=\; f(x) + \lambda^\top g(x) + \nu^\top h(x), \qquad \lambda \ge 0.$$

Dual function:

$$d(\lambda, \nu) \;=\; \inf_x \mathcal{L}(x, \lambda, \nu).$$

The dual function is concave (pointwise infimum of linear functions in (λ, ν)) regardless of the primal.

2.2 Weak duality (proved)

Theorem (weak duality). *For any feasible primal x and dual-feasible $(\lambda \ge 0, \nu)$:*

$$d(\lambda, \nu) \;\le\; p^*,$$

where $p^* = \inf\{f(x) : g(x) \le 0,\ h(x) = 0\}$ is the primal optimum.

Proof. $d(\lambda,\nu) = \inf_{x'} \mathcal{L}(x', \lambda, \nu) \le \mathcal{L}(x, \lambda, \nu) = f(x) + \lambda^\top g(x) + \nu^\top h(x) \le f(x)$, where the last step uses $\lambda \ge 0, g(x) \le 0, h(x) = 0$. Taking the infimum over feasible x gives $d(\lambda, \nu) \le p^*$. □

2.3 Strong duality under Slater (stated)

Theorem (strong duality, Slater's condition). *If $f, g_1, \ldots, g_m$ are convex and there exists a strictly feasible point ($\exists \hat{x} : g(\hat{x}) < 0,\ h(\hat{x}) = 0$), then $d^* = p^*$: the duality gap is zero.*

(Proof: Boyd–Vandenberghe §5.3. The key is that Slater's condition implies the set of active constraints at the optimum is manageable enough to rule out any duality gap.)

2.4 KKT conditions (stated)

At a primal-dual optimum (x^*, λ^*, ν^*) under strong duality + differentiability of f, g, h:

$$\nabla_x \mathcal{L}(x^*, \lambda^*, \nu^*) = 0, \quad \lambda^* \geq 0, \quad g(x^*) \leq 0, \quad h(x^*) = 0, \quad \lambda_i^* g_i(x^*) = 0 \; \forall i.$$

The last condition (complementary slackness) says an inequality constraint is either tight or its dual variable is zero. KKT are sufficient for optimality under convexity + Slater.

---

3. Dual decomposition: where it helps and where it fails

(This section recalls U8's conclusion, for orientation only; the derivation lives in U8.)

With a separable objective $f(x) = \sum_i f_i(x_i)$ and coupling constraints $Ax = b$, the dual function decomposes into independent per-block minimisations. Dual ascent iterates:

1. For each block i in parallel: $x_i^{k+1} \leftarrow \arg\min_{x_i} \mathcal{L}_i(x_i, \nu^k)$.
2. Dual update: $\nu^{k+1} \leftarrow \nu^k + \alpha_k(Ax^{k+1} - b)$.

Where it works: strongly convex f_i (modulus $\mu > 0$). Then each block minimiser is unique, the dual gradient $\nabla d(\nu) = Ax^{k+1} - b$ is well-defined, and it is Lipschitz with constant $\|A\|_2^2/\mu$: strong convexity of the primal is smoothness of the dual. (Bounded Hessians, i.e. smoothness of the f_i, buy the opposite: strong concavity of the dual, when A has full row rank.)

Where it fails: - *Non-smooth* f_i (e.g. $f_i(x_i) = \|x_i\|_1$): the primal minimisation may have multiple solutions; the dual function is not differentiable; subgradient ascent converges slowly ($O(1/\varepsilon^2)$ iterations vs $O(1/\varepsilon)$ for smooth dual). - *No strong convexity in* f_i: the primal iterate x^k may oscillate even as ν^k converges. - *Infeasible intermediate iterates*: the consensus violation $Ax^k - b$ may be large until late in the run, making the algorithm practically unreliable.

ADMM fixes all three.

---

4. ADMM — the centrepiece

4.1 The augmented Lagrangian

For the standard-form problem

$$\min_{x,z} \; f(x) + g(z) \quad \text{subject to} \quad Ax + Bz = c,$$

define the **augmented Lagrangian** with penalty parameter $\rho > 0$:

$$\mathcal{L}_\rho(x, z, y) \;=\; f(x) + g(z) + y^\top (Ax + Bz - c) + \frac{\rho}{2}\|Ax + Bz - c\|_2^2.$$

The quadratic penalty forces the iterates towards feasibility at every step, not just at convergence. Because $\rho\|Ax + Bz - c\|^2/2$ is strongly convex in x (provided A has full column rank), the primal x-subproblem is strongly convex even if f is merely convex — this is the augmentation's gift.

4.2 The x-z-u iterations (scaled dual form)

Introduce the **scaled dual variable** $u = y/\rho$. The ADMM iterations become:

$$\boxed{\begin{aligned} x^{k+1} &\leftarrow \arg\min_x \left\{ f(x) + \frac{\rho}{2}\|Ax + Bz^k - c + u^k\|_2^2 \right\} \\ z^{k+1} &\leftarrow \arg\min_z \left\{ g(z) + \frac{\rho}{2}\|Ax^{k+1} + Bz - c + u^k\|_2^2 \right\} \\ u^{k+1} &\leftarrow u^k + Ax^{k+1} + Bz^{k+1} - c \end{aligned}}$$

Three remarks on structure: 1. The x-update and z-update are **decoupled** — they involve disjoint variable sets — so they can be computed by separate entities (e.g. workers and a coordinator). 2. The u-update is a simple addition: it accumulates the running constraint residual. 3. When $B = -I$ and $c = 0$, the constraint is $Ax = z$; this specialises to the consensus form below.

4.3 Consensus form for distributed ML

The **consensus ADMM** problem:

$$\min_{x_1,\ldots,x_N,\, z} \sum_{i=1}^{N} f_i(x_i) \quad \text{s.t.} \quad x_i = z \;\; \forall i,$$

where machine i owns data shard i and $f_i(x) = \ell_i(x; \text{data}_i) + (\lambda/N)\Omega(x)$ (loss on local data plus a share of a regulariser Ω).

The constraint forces all local models x_i to equal the global model z. Under consensus ADMM with $B_i = -I$, $c_i = 0$:

$$\boxed{\begin{aligned} x_i^{k+1} &\leftarrow \arg\min_{x_i} \left\{ f_i(x_i) + \frac{\rho}{2}\|x_i - z^k + u_i^k\|_2^2 \right\} \quad \text{(machine } i\text{, parallel)} \\ z^{k+1} &\leftarrow \frac{1}{N}\sum_{i=1}^{N}(x_i^{k+1} + u_i^k) \quad \text{(coordinator or all-reduce average)} \\ u_i^{k+1} &\leftarrow u_i^k + x_i^{k+1} - z^{k+1} \quad \text{(machine } i\text{, parallel)} \end{aligned}}$$

This is **the canonical distributed-learning template**. Each machine solves a proximal subproblem on its local data (the x_i-update); the coordinator averages the adjusted iterates (an all-reduce); each machine updates its dual variable locally. The communication per round is one all-reduce of a single model vector — the same cost as a gradient all-reduce in data-parallel SGD, but with far stronger convergence guarantees when the objective is non-smooth.

DPRAM invariant: rounds = number of ADMM outer iterations; bytes per round = $2\times N\times d\times 4$ bytes (each machine sends one d-dimensional float32 vector up and receives one back, gathered and broadcast in one all-reduce; a ring all-reduce moves $2(N-1)\times d\times 4$, the same to leading order). The x_i-subproblems are local, costing RAM-level work on each machine.

4.4 Convergence theorem (stated)

Theorem (Boyd–Parikh–Chu–Peleato–Eckstein 2011, §3.2). *Assume:* - f and g are closed, proper, convex; - the unaugmented Lagrangian $\mathcal{L}_0$ has a saddle point.

Then as $k \to \infty$: 1. *Primal residuals converge:* $\|Ax^k + Bz^k - c\|_2 \to 0$; 2. *Objective converges:* $f(x^k) + g(z^k) \to p^*$; 3. *Dual variable converges:* $y^k \to y^*$ (a dual optimum).

What the theorem does NOT give: a linear convergence rate in general. Under additional assumptions (strong convexity in one block, or full column rank of A) linear rates are known; see Deng–Yin 2012. In practice, ADMM reaches engineering accuracy (say 10^{-3} relative) in tens to low hundreds of outer iterations, which is why it is preferred over primal methods when each outer iteration is a parallelisable batch solve.

4.5 Practical ρ selection

ρ controls the trade-off between primal feasibility and dual update aggressiveness. Too small: feasibility is weakly enforced, slow to converge. Too large: the primal subproblems are dominated by the quadratic term, not by f and g, and the method effectively becomes projected gradient descent.

Rule of thumb (Boyd–Parikh et al., §3.4): use the **varying** ρ heuristic — increase ρ by the factor τ when the primal residual $\|Ax^k + Bz^k - c\|$ exceeds the dual residual $\|\rho A^\top (z^k - z^{k-1})\|$ by more than a factor μ, and decrease it symmetrically:

$$\rho^{k+1} = \begin{cases} \tau \rho^k & \text{if } \|r^k_{\text{prim}}\| > \mu \|s^k_{\text{dual}}\|, \\ \rho^k / \tau & \text{if } \|s^k_{\text{dual}}\| > \mu \|r^k_{\text{prim}}\|, \\ \rho^k & \text{otherwise,} \end{cases}$$

with $\mu = 10$, $\tau = 2$ being common choices. This heuristic is not theoretically principled but works well in practice and is the default in most distributed ML implementations.

When ADMM beats gradient methods: (i) non-smooth regularisers (lasso, group lasso, nuclear norm) where each primal subproblem has a closed-form proximal operator; (ii) problems with hard constraints that cannot be folded into a smooth penalty; (iii) distributed settings where the communication overhead of many SGD rounds exceeds the cost of a small number of ADMM outer iterations with cheap local solves.

4.6 Worked example: distributed lasso via consensus ADMM

The **lasso**:

$$\min_z \left\{ \frac{1}{2} \|Az - b\|_2^2 + \lambda \|z\|_1 \right\},$$

with the data matrix $A \in \mathbb{R}^{n \times d}$ split row-wise across N machines ($A_i \in \mathbb{R}^{n_i \times d}$, $b_i \in \mathbb{R}^{n_i}$).

Reformulate as consensus ADMM: $f_i(x_i) = \frac{1}{2}\|A_i x_i - b_i\|_2^2$, $g(z) = \lambda \|z\|_1$, constraint $x_i = z$.

The x_i-update (machine i): least-squares with a Tikhonov term,

$$x_i^{k+1} = (A_i^\top A_i + \rho I)^{-1} (A_i^\top b_i + \rho (z^k - u_i^k)).$$

This is a ridge-regression solve. The first round pays $O(n_i d^2 + d^3)$: $O(n_i d^2)$ to form $A_i^\top A_i$ (and $A_i^\top b_i$), $O(d^3)$ to Cholesky-factor $A_i^\top A_i + \rho I$. The factor is cached, so every later round costs only two triangular solves, $O(d^2)$, while ρ is held fixed (the varying-ρ heuristic of §4.5 forces a refactorisation each time ρ changes).

The z-update (coordinator): minimise $g(z) + \frac{N\rho}{2}\|z - \bar{v}^k\|_2^2$ where $\bar{v}^k = \frac{1}{N}\sum_i (x_i^{k+1} + u_i^k)$. This is a **soft-thresholding** (proximal map of $\|\cdot\|_1$):

$$\boxed{z_j^{k+1} = \mathcal{S}_{\lambda/(N\rho)}\left(\bar{v}_j^k\right), \quad \mathcal{S}_\kappa(a) = \operatorname{sign}(a)\max(|a| - \kappa,\ 0).}$$

Derivation: the j-th coordinate subproblem is $\min_{z_j}\{\lambda|z_j| + \frac{N\rho}{2}(z_j - \bar{v}_j)^2\}$. The subdifferential condition $0 \in \lambda\partial|z_j| + N\rho(z_j - \bar{v}_j)$ gives: if $|\bar{v}_j| \le \lambda/(N\rho)$ then $z_j = 0$; if $\bar{v}_j > \lambda/(N\rho)$ then $z_j = \bar{v}_j - \lambda/(N\rho)$; symmetrically for negative $\bar{v}_j$. Combining: $\mathcal{S}_\kappa(a)$ as stated. □

Soft-thresholding is the proximal operator of $\|\cdot\|_1$; its closed form is why lasso (and any regulariser whose proximal map is cheap) is the natural home for ADMM.

---

5. Decentralised methods: gossip and DSGD

Consensus ADMM still requires a coordinator for the z-update. In a fully decentralised topology (no central node; machines arranged on a graph), averaging is replaced by **gossip**.

5.1 Gossip averaging

Machines V are nodes of a connected graph G. A **doubly stochastic mixing matrix** $W \in \mathbb{R}^{|V|\times|V|}$ satisfies $W_{ij} \ge 0$, $W\mathbf{1} = \mathbf{1}$, $\mathbf{1}^\top W = \mathbf{1}^\top$, and $W_{ij} = 0$ whenever $(i,j) \notin E$ (only neighbours communicate). Popular choices: the Metropolis–Hastings weights (Xiao–Boyd 2004).

Gossip averaging: each node i starts with x_i^0, and repeats

$$x_i^{k+1} = \sum_{j\in\mathcal{N}(i)\cup\{i\}} W_{ij}x_j^k.$$

Mixing rate. Let $\rho(W) = \max(|\lambda_2(W)|, |\lambda_N(W)|)$ — the second-largest eigenvalue **in absolute value** (eigenvalues near -1 hurt exactly as much as near $+1$: they make the iteration oscillate instead of mix; on a *bipartite* graph with symmetric zero-diagonal weights $\lambda_N = -1$ and gossip never converges — self-weights fix this). After k rounds:

$$\left\|x^k - \bar{x}\mathbf{1}\right\|_2 \;\le\; \rho(W)^k \left\|x^0 - \bar{x}\mathbf{1}\right\|_2,$$

where $\bar{x} = \frac{1}{N}\sum_i x_i^0$ is the true average. Convergence requires $\rho(W) < 1$ (connectivity guarantees $\lambda_2 < 1$; positive self-weights keep $\lambda_N > -1$). The number of gossip rounds to reach accuracy ε is

$$k = O\left(\frac{\log(1/\varepsilon)}{1-\rho(W)}\right).$$

The spectral gap $1 - \lambda_2(W)$ is the bottleneck: a ring topology has $1 - \lambda_2 = \Theta(1/N^2)$ (slow — the graph diameter controls mixing); an expander has $1 - \lambda_2 = \Omega(1)$ (fast, independent of N). This is the mixing-time measurement the lab exercises.

5.2 Decentralised SGD (DSGD)

DSGD (Lian et al. 2017; author list in §10) combines local SGD steps with gossip averaging:

$$x_i^{k+1} = \sum_j W_{ij}x_j^k - \alpha_k \nabla f_i(x_i^k; \xi_i^k),$$

where ξ_i^k is a local minibatch. The topology enters through $1 - \lambda_2(W)$:

Convergence (informal; the shape of the rate, not a quoted theorem). For non-convex f, smooth and with bounded gradient variance, DSGD with step-size $\alpha_k = \alpha\sqrt{N/K}$ on N nodes

achieves a bound of the shape

$$\frac{1}{K}\sum_{k=0}^{K-1}\mathbb{E}\|\nabla F(x^k)\|^2 = O\!\left(\frac{\sigma}{\sqrt{NK}} + \frac{\text{topology term}}{K^{2/3}}\right),$$

where the topology term is $O((1-\lambda_2)^{-2/3})$. Lian et al. (2017) prove that DSGD matches the first, centralised term once K is large relative to the topology; the explicit $K^{-2/3}(1-\lambda_2)^{-2/3}$ form of the second term is the shape later decentralised-SGD analyses report, and we have not checked its constants and hypotheses against a source — treat it as indicative. A well-connected graph (large $1-\lambda_2$) reduces the second term. For the ring: $(1-\lambda_2) = \Theta(1/N^2)$, so the topology term dominates for large N; for an expander it becomes a constant. DSGD matches the rate of centralised SGD (the first term) when K is large enough relative to the graph's spectral gap.

5.3 One-shot averaging (bridge to U20)

In the regime where each machine has enough local data to produce a good local estimator, one can run full local optimisation on each machine and then average the local optima:

$$\hat{\theta} = \frac{1}{N}\sum_i \hat{\theta}_i^{\text{local}}.$$

This is **one-shot** (single communication round, $O(Nd)$ bytes total). It works well when $n_i \gg d$ and the loss is strongly convex; it degrades when the local data is heterogeneous (different distributions on each machine). U20 analyses this formally (Bag of Little Bootstraps; divide-and-conquer estimators; communication-constrained lower bounds). Forward pointer: what local averaging gains in communication, it may lose in statistical efficiency — U20 quantifies the trade-off.

---

6. Newton-sketch: the randNLA × optimisation crossover

One section; stated, not proved. Prerequisites: U7 (sketching, JL), U9 (Newton's method for smooth strongly convex f).

Newton's method for minimising a smooth, strongly convex $f : \mathbb{R}^d \to \mathbb{R}$:

$$x^{k+1} = x^k - (\nabla^2 f(x^k))^{-1}\nabla f(x^k).$$

Local quadratic convergence is rapid, but each step solves a $d \times d$ system in the Hessian. In the data-fitting problems of this book, $f(x) = \sum_{i=1}^n \psi(\langle a_i, x\rangle)$ with a data matrix $A \in \mathbb{R}^{n\times d}$ (n data points, d features), and merely forming the Hessian $A^\top \mathrm{diag}(\psi'')A$ costs $O(nd^2)$ per step — prohibitive when n and d are both large.

Newton-sketch (Pilanci–Wainwright 2017). Such a Hessian comes with a square root $\nabla^2 f(x)^{1/2} \in \mathbb{R}^{n\times d}$ (here $\mathrm{diag}(\psi'')^{1/2}A$), so that $\nabla^2 f = (\nabla^2 f^{1/2})^\top \nabla^2 f^{1/2}$. Newton-sketch compresses its n rows with a random sketch $S_k \in \mathbb{R}^{m\times n}$ ($m \ll n$; e.g. a sub-sampled randomised Hadamard transform, normalised so that $\mathbf{E}[S_k^\top S_k] = I_n$) and uses the **sketched Hessian**

$$\tilde{H}_k = \left(S_k \nabla^2 f(x^k)^{1/2}\right)^\top \left(S_k \nabla^2 f(x^k)^{1/2}\right) \in \mathbb{R}^{d\times d},$$

whose expectation is the true Hessian, in place of it:

$$x^{k+1} = x^k - \tilde{H}_k^{-1}\nabla f(x^k).$$

Theorem (Pilanci–Wainwright 2017, informal). *For self-concordant f (and, near the optimum, for strongly convex f with a Lipschitz Hessian), a sketch size m of order $\min\{n, d\}$ suffices for Newton-sketch to converge super-linearly with high probability, with guarantees that do not depend on condition numbers. Each iteration costs $O(nd \log m)$ to apply a Hadamard-based sketch plus $O(md^2)$ to form and solve with $\tilde{H}_k$, against $O(nd^2)$ for an exact Newton step; with $m \asymp d$ and $n \geq d^2$, an iteration costs $O(nd \log d)$, linear in the size of the data.*

The crossover: the same JL/sketching machinery that powers U7's randomised SVD now accelerates second-order optimisation. This is the randNLA × optimisation seam — every course on distributed ML eventually hits it, and U7 + U9 + U19 form the triple that makes it legible.

---

7. The invariant (memorise this box)

Rounds = ADMM outer iterations (one x-update + one z-update + one u-update per round). **Bytes = $2 \times d \times N \times 4$ bytes per round** (one float32 model vector up and one down per machine, one all-reduce). **Local work per round** (x_i-update): a ridge solve, $O(n_i d^2 + d^3)$ once to form and Cholesky-factor, then $O(d^2)$ per round from the cached factor (fixed ρ); or a proximal step if f_i has a closed-form prox. **Convergence guarantee (consensus ADMM):** primal residuals, objective, and dual variable all converge under convexity + saddle-point assumption; rate in general $O(1/k)$ (ergodic); linear under strong convexity. **Gossip topology enters through $1 - \lambda_2(W)$:** ring has $\Theta(1/N^2)$ gap (slow); expander has $\Omega(1)$ gap (fast, N-independent); DSGD's topology term scales as $(1 - \lambda_2)^{-2/3}$. **Newton-sketch buys $O(nd^2) \to O(nd \log m + md^2)$ per iteration** (with $m \asymp d$: $O(nd \log d)$ once $n \geq d^2$) at the cost of sketch variance (U7's machinery applied to the Hessian).

---

8. Lab sketch — project track

Project-track lab (no weekly receipt): run these experiments free and locally in the course lab image (`labs/image/`) — or any Spark- or Ray-capable Docker environment — as deep-dive and group-project material.

Experiment A — Distributed lasso via consensus ADMM (Ray) vs. centralized lasso (scikit-learn): Generate a sparse signal $\theta^* \in \mathbb{R}^{200}$ (10 non-zeros), data $A \in \mathbb{R}^{5000 \times 200}$, $b = A\theta^* + \varepsilon$. Split rows uniformly across $N = 5$ Ray workers. Run consensus ADMM ($\rho = 1.0$; adaptive ρ heuristic as a variant) for 200 outer iterations; compare the primal residual trajectory against a centralised solve. Measure: (i) objective gap vs. iteration; (ii) communication bytes vs. iteration; (iii) soft-threshold sparsity pattern of z^k evolving to θ^*. Expected finding: ADMM reaches engineering accuracy in (∼)50 outer iterations; the soft-threshold z-update visibly zeros out the right coordinates.

Experiment B — Gossip on ring vs. expander (mixing-time measurement): Initialise $N = 16$ nodes with random values. Implement gossip averaging on (i) a ring graph with Metropolis–Hastings weights — on a ring every node has degree 2, so $W_{ij} = 1/3$ for each neighbour and $W_{ii} = 1/3$ (the self-weight is what keeps $\lambda_N > -1$; try $W_{ii} = 0, W_{ij} = 1/2$ on the even ring once, watch it oscillate forever, and explain why via §5.1) — and (ii) a random 3-regular graph on the same 16 nodes, an expander with high probability (the Petersen graph, the classic small 3-regular example, has only 10 vertices; use it for a separate $N = 10$ run if you like). Measure the consensus error $\|x^k - \bar{x}\mathbf{1}\|_2$ vs. round k; fit the exponential decay to

estimate $\rho(W)$ empirically. Compare with the theoretical $\rho(W) = \lambda_2(W) = \frac{1}{3}\big(1 + 2\cos(2\pi/N)\big)$ for this ring weighting (for even N, $|\lambda_N(W)| = 1/3 < \lambda_2(W)$, so $\rho(W) = \lambda_2(W)$). Expected finding: expander mixes in $O(\log N)$ rounds; ring in $O(N^2)$ rounds — the spectral gap makes the difference.

Deliverable: the notebook + a one-page cost-accounting in the unit invariant vocabulary (rounds, bytes, local work) for each experiment.

---

9. Lean target

Schematic certificate (weak duality): the proof in §2.2 is four lines of algebra involving an ordered field + a sign condition — genuinely certifiable in Lean/Mathlib without additional library development. The cert is labelled SCHEMATIC: the goal is to encode the weak-duality inequality as a Lean theorem, establishing the pattern for stronger optimality certificates (KKT, primal-dual rates) in later work.

Target name: `WeakDualityLean.lean`. Exercise U19.L1: run the cert; read the proof top-down; fill in the blanked subdifferential step for the soft-threshold derivation.

LLM lens — where this unit lands on the running workload (U0L):
Federated fine-tuning is consensus optimisation: locally adapted models (clients' LoRA updates) reconciled through exactly this unit's ADMM star — local minimisers around a consensus hub, prices on the edges.

10. Sources

Boyd S, Parikh N, Chu E, Peleato B, Eckstein J (2011). *Distributed optimization and statistical learning via the alternating direction method of multipliers.* Foundations and Trends in Machine Learning 3(1):1–122. — the primary source; free PDF at stanford.edu/~boyd/papers/admm_-distr_stats.html.

Bertsekas DP, Tsitsiklis JN (1989). *Parallel and Distributed Computation: Numerical Methods.* Prentice Hall. — the classical reference for distributed dual decomposition and ADMM precursors.

Xiao L, Boyd S (2004). *Fast linear iterations for distributed averaging.* Systems and Control Letters 53(1):65–78. — doubly stochastic mixing matrices; Metropolis–Hastings weights; spectral-gap analysis.

Nedic A, Ozdaglar A (2009). *Distributed subgradient methods for multi-agent optimization.* IEEE Transactions on Automatic Control 54(1):48–61. — decentralised subgradient descent; foundational.

Lian X, Zhang C, Zhang H, Hsieh C-J, Zhang W, Liu J (2017). *Can decentralized algorithms outperform centralized algorithms? A case study for decentralized parallel stochastic gradient descent.* NeurIPS 2017. — DSGD; topology enters through $1 - \lambda_2$; rate comparison with centralised SGD.

Pilanci M, Wainwright MJ (2017). *Newton sketch: a near linear-time optimization algorithm with linear- quadratic convergence.* SIAM Journal on Optimization 27(1):205–245. — Newton-sketch; sketched Hessians; sub-sampled randomised Hadamard; the randNLA × optimisation crossover.

Boyd S, Vandenberghe L (2004). *Convex Optimization.* Cambridge University Press. — Lagrangian, dual function, weak/strong duality, Slater, KKT (§§5.1–5.5); free PDF at stanford.edu/~boyd/cvxbook/.

Wikipedia: *Alternating direction method of multipliers*, *Gossip protocol*, *Decentralized gradient descent*, *Proximal gradient method*, *Lasso (statistics)*.

---

11. Exercises

Light (examined-adjacent; self-check on the deep material):

E19.1 Write out the three ADMM iterations (§4.2, scaled dual form) for the problem $\min_x \frac{1}{2}\|x - c\|_2^2$ subject to $x = z$, $g(z) = \mathbf{0}$. Show that the error contracts by the factor $\rho/(1+\rho)$ per iteration (geometric convergence — NOT one step; verify by iterating from $x^0 = z^0 = u^0 = \mathbf{0}$ with $\rho = 1$), and explain which term in the x-update is responsible for the contraction.

E19.2 State the weak duality inequality (§2.2) for the lasso problem and identify the dual variable. What does complementary slackness say about non-zero entries of the primal solution?

E19.3 A gossip network has $N = 8$ nodes on a ring. First take the naive equal weights $W_{ij} = 1/2$ for neighbours, $W_{ii} = 0$: show that the even ring is bipartite, that $\lambda_N(W) = -1$, and hence that gossip does NOT converge (§5.1's $\rho(W) = 1$). Then repair it with the true Metropolis weights ($W_{ij} = 1/3$, $W_{ii} = 1/3$): compute $1 - \rho(W)$ and how many rounds reduce the consensus error by a factor 10^{-3}.

E19.4 (Invariant box.) For the consensus lasso experiment in §8: if $N = 10$, $d = 500$, and one ADMM round takes 2 seconds of local compute per machine plus 50 ms of all-reduce, estimate the wall-clock time to 100 outer iterations, and the total bytes sent over the network.

Deep (Module-3 / research depth):

E19.5 *(Derive the lasso z-update.)* For the consensus ADMM formulation of the lasso (§4.6), derive the soft-thresholding z-update from the subdifferential optimality condition. Show the derivation holds coordinate-wise and that the threshold is $\lambda/(N\rho)$.

E19.6 *(Ring vs. complete gossip mixing.)* Let G_1 be the N-node ring and G_2 be the complete graph (K_N). For the uniform Metropolis weights on each, compute (or bound) λ_2 for both. Show that gossip on K_N converges in one round for any target accuracy, while on the ring the number of rounds grows as $\Theta(N^2 \log(1/\varepsilon))$. Explain why the one-shot averaging baseline (§5.3) is the extreme case of the complete-graph limit.

E19.7 *(Convergence under strong convexity.)* Assume each f_i in the consensus lasso is μ-strongly convex ($\mu > 0$) and L-smooth. Using the convergence analysis of Boyd–Parikh et al. §3.2 and the ADMM linear convergence result for strongly convex problems (Deng–Yin 2012), state the per-iteration contraction factor in terms of μ, L, and ρ. For fixed μ/L, find the ρ that minimises the contraction factor.

Chapter 26

20 distributed inference

U20 — Distributed statistical inference: RAM → PRAM → DPRAM

The ScaDaMaLe book, unit 20 (deep; Part IV). Our own exposition; original sources credited in §9.

Instantiating the three-lens invariant. U0 priced *operations* (RAM), *depth* (PRAM), and *rounds/bytes* (DPRAM/MPC). This unit applies the same habit to *statistical efficiency*: the resource being priced is not CPU cycles but **information about the unknown parameter**. The new question is: when data are distributed across M machines, how much statistical power do we lose, and how does the loss depend on computation, communication, and algorithm design?

> **RAM:** a single statistician with all n observations computes the best estimator; the fundamental limit is the minimax rate — the fastest any estimator can converge as $n \to \infty$. **PRAM:** split n across M machines, compute local estimators in parallel, then average; the averaged estimator matches the centralised rate under a condition on M vs n (one-shot averaging). Bootstrapping is infeasible at scale — the Bag of Little Bootstraps replaces it. **DPRAM/MPC:** when each machine may only send B bits, the centralised rate is unachievable below a threshold total budget (Zhang–Duchi–Jordan–Wainwright lower bound): the minimax risk rises as communication shrinks. Split conformal prediction escapes this: it is a *distribution-free guarantee* that costs only one calibration pass and one aggregate.

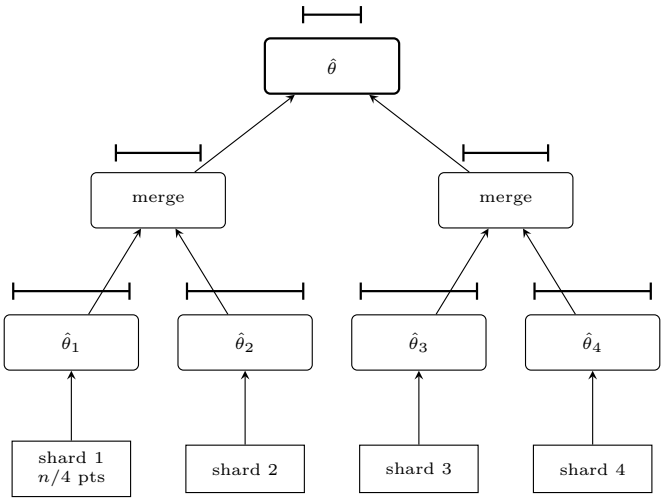
Entry kit — what this chapter assumes; follow a link if you need the memory jog (map: Figure 1):
F.S2 (concentration, for the shard-level errors) • F.S3 (risk and minimax, for what merging must preserve)

1. Intuition — statistical efficiency as a resource

Every earlier unit priced one scarce resource: operations (U0), memory (U4), rounds and bytes (U5). When we move to *statistics*, the scarcest resource is **data** — or more precisely, the Fisher information carried by the data.

A single statistician with all n samples estimates an unknown parameter θ optimally: the minimax rate (the lowest worst-case mean-squared error achievable by any estimator) is typically $\Theta(1/n)$ for a one-dimensional smooth problem — the Cramér–Rao floor. This is the RAM baseline.

Now split the data across M machines. Each machine has n_i samples ($\sum n_i = n$) and must compute a local estimate. How should the machines combine their local estimates? Three



the merge is a U1 reduction; the STATISTICAL question is what the merge preserves: when is the root as good as one estimator on ALL the data (full efficiency), and when does one round of averaging provably lose against the harder, nonparametric truths (U14/U15)?

Figure 26.1: Divide-and-conquer inference: shard the data, estimate locally, merge up a tree — error bars shrinking at every level. Computationally the tree is just a reduction (U1); statistically it is a theorem battleground: for smooth parametric problems one-shot averaging achieves the full-data rate, while for the tree-density problems of U14/U15 the merge must carry *structure* (counts, not point estimates) to avoid losing information. Merging estimators here and merging experts in U0L §3 are the same divide-and-combine question in two registers.

regimes emerge:

1. **Oracle (centralised):** all raw data move to one machine; the centralised estimator achieves rate $1/n$. Communication cost: $O(n)$ bytes — infeasible at scale.
2. **One-shot averaging:** each machine sends one d-dimensional vector (the local estimate); the coordinator averages them. Communication: $O(Md)$ bytes — embarrassingly cheap. *Does this match the centralised rate?* **Yes — under a condition.** This is the content of §2.
3. **Communication-constrained:** each machine sends only B bits total. Below a threshold, no estimator can match the centralised rate, regardless of algorithm. This is the lower bound of §4.

Between regimes 2 and 3 sits the Bag of Little Bootstraps (§3): distributing the bootstrap — a $O(n)$-memory monolith — so that uncertainty quantification is achievable at scale.

The split conformal construction (§5) sidesteps all of the above by making a *weaker but distribution-free* guarantee: instead of point-estimating θ, it certifies a prediction interval with marginal coverage $1-\alpha$ — a *promise* the provider can keep without any parametric assumption.

Figure 26.1 is the unit's question drawn as a tree: what does the merge preserve?

2. One-shot / divide-and-conquer averaging

2.1 Setup

Fix a smooth parametric problem: data $X_1, \dots, X_n \overset{\text{i.i.d.}}{\sim} P_\theta$ with $\theta \in \Theta \subseteq \mathbb{R}^d$, and a *M-estimator* defined by the empirical objective $\hat{\theta}_n = \operatorname{argmin}_\theta \frac{1}{n}\sum_{i=1}^n \ell(\theta; X_i)$ for a loss ℓ that is

twice differentiable in θ.

Examples: MLE ($\ell = -\log p_\theta$); ordinary least squares ($\ell = (y - \theta^\top x)^2$); logistic loss.

Distribute: machine k holds $n_k = n/M$ samples (assume equal split for clarity). Machine k computes the local M-estimator:

$$\tilde{\theta}_k = \operatorname*{argmin}_\theta \frac{1}{n_k} \sum_{i \in \mathcal{S}_k} \ell(\theta; X_i).$$

The **averaged estimator** is:

$$\bar{\theta} = \frac{1}{M} \sum_{k=1}^{M} \tilde{\theta}_k.$$

2.2 When does averaging match the centralised rate?

Theorem (Zhang–Duchi–Wainwright style, stated). *Suppose $\ell(\cdot; x)$ is three times continuously differentiable in θ, and the Fisher information matrix $\mathcal{I}(\theta)$ is positive definite at the true θ_0. Assume each local sample size $n_k = n/M$ satisfies $n_k \to \infty$. Then the averaged estimator satisfies*

$$\bar{\theta} - \hat{\theta}_n = O_p\left(\frac{M}{n}\right)$$

and in particular $\bar{\theta}$ achieves the same asymptotic variance as the centralised estimator $\hat{\theta}_n$ when

$$M = o(n^{1/2}).$$

What the condition means. Each local sample size $n_k = n/M$ must grow fast enough that the local *bias* — the error introduced by solving the problem on a fraction of the data — is negligible relative to the statistical *variance*. The bias of each local estimator is $O(1/n_k) = O(M/n)$. The variance of the average of M local estimators, each with variance $O(1/n_k)$, is $O(1/n_k \cdot 1/M) = O(1/n)$ — matching the centralised rate. For the bias to be absorbed into the variance, we need $(M/n)^2 \ll 1/n$, i.e. $M^2 \ll n$, i.e. $M = o(n^{1/2})$.

Heuristic proof of the bias-variance decomposition. Expand $\tilde{\theta}_k$ around θ_0 via a Taylor argument (as in the standard M-estimator expansion). The leading term is

$$\tilde{\theta}_k - \theta_0 \approx \underbrace{\mathcal{H}_k^{-1} \cdot \frac{1}{n_k} \sum_{i \in \mathcal{S}_k} \nabla_\theta \ell(\theta_0; X_i)}_{\text{variance term, } O_p(1/\sqrt{n_k})} + \underbrace{O_p(1/n_k)}_{\text{bias from local curvature}},$$

where $\mathcal{H}_k$ is the local empirical Hessian. Averaging over M machines: - **Variance term:** by independence of the shards, the variance of $\bar{\theta}$ from the first term is $\frac{1}{M} \cdot O(1/n_k) = O(1/n)$ — the centralised rate. - **Bias term:** $O(1/n_k) = O(M/n)$. Squared: $O(M^2/n^2)$. For this to be $o(1/n)$ (dominated by the variance): $M^2/n^2 = o(1/n)$, i.e. $M = o(n^{1/2})$. $\square_{\text{heuristic}}$

When averaging fails. The condition $M = o(n^{1/2})$ is sharp-ish in the parametric smooth case. Averaging fails — or degrades — in: - **Nonsmooth problems:** quantile regression, ℓ_1-penalised methods, threshold estimators. Local bias can decay more slowly than $O(1/n_k)$, and the same bias-squared-versus-variance arithmetic tightens the condition. A bias of $O(n_k^{-3/4})$ gives $(M/n)^{3/2} \ll 1/n$, i.e. $M = o(n^{1/3})$; a bias of $O(n_k^{-1/2})$ gives a squared bias of order M/n, never below the variance $1/n$, so the number of machines cannot grow at all ($M = O(1)$), and even a fixed M leaves the bias of the order of the standard error. - **High-dimensional**

problems: when $d \gg 1$ and the model is sparse, the local estimator on n/M samples may not even exist or may be badly inconsistent (e.g. Lasso when $n/M < d$). - **Heterogeneous shards:** if data are not i.i.d. across machines (stratified populations, temporal shifts), the average of local estimates is a biased estimator of the pooled optimum.

---

3. Bag of Little Bootstraps (the centerpiece)

3.1 Why the bootstrap is infeasible at scale

The **bootstrap** (Efron 1979) is the workhorse of nonparametric uncertainty quantification: draw B samples of size n *with replacement* from the empirical distribution $\hat{F}_n$, compute the statistic on each, and use the empirical distribution of the B bootstrap statistics as the sampling distribution. For $B = 1000$ and $n = 10^8$:

- Each bootstrap sample has $n = 10^8$ elements; with $0.632n \approx 6.3 \times 10^7$ distinct original rows (the rest are repeats). **Each resample is 63% the size of the dataset** — not a small subsample.
- Total data processed: $B \times n = 10^{11}$ element-references. On a cluster that holds n in memory per machine, each bootstrap resample requires a full cluster-wide shuffle.
- This is simply infeasible for production-scale n.

3.2 The BLB construction

Kleiner, Talwalkar, Sarkar, Jordan (JRSS-B 2014). The Bag of Little Bootstraps (BLB) replaces the infeasible n-bootstrap with a two-level scheme:

Step 1 — Draw s subsets of size $b \ll n$ (without replacement) from the full data $\{X_1, \ldots, X_n\}$. The subset size $b = n^\gamma$ for some $\gamma \in (0.5, 1)$; the paper recommends $\gamma = 0.6$ ($b = n^{0.6}$) as a practical default.

Step 2 — Bootstrap each subset to size n. For each subset $\mathcal{S}_j$ of size b, draw r bootstrap resamples of size n — but *from the b points in $\mathcal{S}_j$*, not from all n. This is cheap: a resample of size n from b points is just a **multinomial weight vector $\mathbf{w}$** $\sim \text{Multinomial}(n, (1/b, \ldots, 1/b))$; we never materialise n records, only the b-dimensional weight vector. The reweighted statistic is $\hat{\theta}(\mathbf{w}) = \text{argmin}_\theta \sum_{i \in \mathcal{S}_j} w_i \ell(\theta; X_i)$.

Step 3 — Aggregate. For each subset j, compute the bootstrap distribution of the statistic over the r resamples; compute the quality measure (e.g. confidence interval width, bootstrap variance). Average the quality measure across the s subsets.

3.3 Why the multinomial reweighting trick is cheap

A naive bootstrap resample of size n from n points draws n integers uniformly with replacement from $[n]$: about $(1 - 1/e) \approx 63.2\%$ of the original rows appear at least once, so the resample is large. In the BLB, the subset has only $b = n^{0.6}$ rows; a resample of size n from b rows is specified by b non-negative integer weights summing to n — a b-dimensional integer vector. For $n = 10^8$ and $b = n^{0.6} \approx 63{,}100$, the weight vector has 63{,}100 entries, not 10^8.

Concretely: instead of re-reading and reshuffling 10^8 records per bootstrap resample, the computation is a weighted M-estimation on 63{,}100 records with integer weights — orders of magnitude cheaper. For many estimators (OLS, logistic, quantile), weighted M-estimation is available in closed form or via efficient iteratively reweighted least squares.

3.4 Communication accounting (rounds and bytes)

The BLB is **embarrassingly parallel across subsets** and **embarrassingly parallel across bootstrap resamples within a subset**.

- **Round 1 (map):** draw s subsets $\mathcal{S}_1, \ldots, \mathcal{S}_s$ (each of size b) from the full data. This requires a distributed sample (one shuffle in the MPC model — cf. U4's reservoir sample). Cost: $O(sb)$ bytes total (each subset element must be located), 1 round.
- **Round 2 (inner bootstrap, parallel per subset):** for each subset, draw r multinomial weight vectors and solve the weighted M-estimation. This is local computation — the b data points for each subset fit on one machine. Cost: $O(r)$ M-estimation solves per subset, each on b points, 0 communication rounds.
- **Round 3 (aggregate quality measure):** each of the s machines sends one scalar (or one confidence interval endpoint) — the quality estimate for its subset — to the coordinator. Cost: $O(s)$ scalars, 1 round.

Total rounds: $R = O(1)$ (dominated by the initial sample draw). Total bytes: $O(sb \cdot \text{record size} + s)$ — far less than the $O(Bn)$ bytes of naive full bootstraps.

Consistency theorem (stated). *Fix a smooth functional $\xi(F)$ of the data distribution F (e.g. the mean, a quantile, the OLS coefficient vector). Under mild regularity conditions on ξ and F, the BLB estimate of the sampling distribution of $\hat{\xi}_n = \xi(\hat{F}_n)$ converges to the true sampling distribution at the same rate as the standard bootstrap, for any choice of $b = n^\gamma$ with $\gamma \in (0.5, 1)$, as $n \to \infty$ with $s, r \to \infty$ at appropriate rates.*

Source: Kleiner, Talwalkar, Sarkar, Jordan, JRSS-B 2014, Theorem 1. We do not prove this here; the proof uses the functional delta method applied to the empirical process indexed by ξ.

---

4. Communication-constrained minimax lower bounds

4.1 The framework

Zhang, Duchi, Jordan, Wainwright (NeurIPS 2013). Fix the d-dimensional mean estimation problem: $X_1, \ldots, X_n \overset{\text{i.i.d.}}{\sim} \mathcal{N}(\theta, I_d)$ with $\theta \in \Theta = \{-1, +1\}^d$ (the hypercube; the hardness comes from the packing number). Split the n samples across M machines, with $n_k = n/M$ per machine. Each machine k is permitted to send a message Y_k of at most B bits to a central coordinator, who then produces the estimate $\hat{\theta}$.

The **minimax risk** under a bit-budget constraint is:

$$\mathcal{R}(n, M, B) = \inf_{\text{protocol}, \hat{\theta}} \sup_{\theta \in \Theta} E\left[\|\hat{\theta} - \theta\|_2^2\right],$$

where the infimum is over all communication protocols with per-machine budget B bits.

The centralised (unconstrained) minimax rate for this problem is $\mathcal{R}^*(n) = \Theta(d/n)$ (standard parametric rate; Cramér–Rao in this Gaussian model).

4.2 The lower bound phenomenon

Theorem (Zhang–Duchi–Jordan–Wainwright 2013, informal statement). *For the d-dimensional mean estimation problem above, achieving the centralised minimax rate $\Theta(d/n)$ requires a total bit budget*

$$M \cdot B \;\geq\; \Omega\left(\frac{d \cdot M}{\log M}\right),$$

i.e. each machine must send at least $B = \Omega(d/\log M)$ bits.

Quantitative form (approximate; not yet checked against the paper). There exist universal constants $c, C > 0$ such that for $B \le Cd/\log M$:

$$\mathcal{R}(n, M, B) \;\ge\; c \cdot \frac{d}{n} \cdot \frac{d}{MB}.$$

However, as written this display exceeds $\Theta(d/n)$ only when $MB \ll d$, i.e. $B \ll d/M$, a far smaller budget than the per-machine threshold $B \approx d/\log M$ of the informal statement above (so the display is the weaker lower bound), and the two cannot both be the paper's exact form. We keep the per-machine threshold $B = \Omega(d/\log M)$ as the statement this unit uses (it is the one repeated in §4.4, §6 and E20.4) and flag this display as approximate. *Source to check: Zhang et al. 2013, Theorem 2; constants and proof are in the original.*

Warning on constants. The exact form of the bound and constant factors depend on the specific prior family used in the Fano/Le Cam lower bound argument. For the exact form, read the paper's Theorem 2; do not reproduce a self-invented formula. The qualitative message — that below $\Omega(d/\log M)$ bits per machine the centralised rate is unachievable — is the robust takeaway.

4.3 Why this is the statistical twin of U0's communication lens

In U0, the α-β cost model for communication (U0 v2 weave) shows that sending a message of M bytes costs $\alpha + \beta M$; communication-avoiding matrix multiply (U6) buys a factor from the βM term. That is the *algorithmic* communication lens: bits are priced by bandwidth.

Here, bits are priced by *information about* θ: each machine can communicate at most B bits, which carries at most B bits of mutual information about its local data X_k. The lower bound shows that below the threshold, no estimator can recover θ at the centralised rate, because the channel from each machine to the coordinator is an information bottleneck. The Fano inequality (or Le Cam's method) links mutual information capacity to estimation error — *information theory as statistics.*

The two lenses are literally the same resource (bits), priced by different scarcity: bandwidth in U0, Fisher information capacity here.

4.4 Proof sketch (heuristic)

The lower bound uses a **minimax reduction to a binary hypothesis testing problem** over a product prior. Briefly:

1. **Embed the problem.** Choose d orthogonal "hard" directions; place a prior over ± 1 independently in each direction (the hypercube $\Theta = \{-1, +1\}^d$).
2. **Reduce to one bit per direction.** For any estimator to achieve error $< c \cdot d/n$ in squared loss, it must correctly recover all d bits of θ (up to a constant fraction). Each bit corresponds to a binary hypothesis test on the transmitted messages.
3. **Apply Fano's inequality.** The mutual information between the parameter bit and the full transcript (all M messages, each of B bits) is at most $\min(MB, n$ Fisher info$) \sim \min(MB, n/M \cdot 1) \cdot d^{-1}$ per coordinate (this is the quantitative step requiring careful calculation — see the paper). Fano implies: if $B \ll d/\log M$ per machine, the bit-recovery error rate is bounded away from zero, hence the ℓ_2^2 risk stays above the centralised rate.

The argument is a distributed variant of standard minimax lower-bound technology; the novelty is the careful accounting of how local sample Fisher information passes through the communication bottleneck.

5. Split conformal prediction — the SCT bridge

5.1 The conformal idea

Problem. Given a black-box regression model $\hat{f}$ (trained on a training set) and a new test point X_{n+1}, construct a *prediction set* $\hat{C}(X_{n+1})$ such that

$$P\Big(Y_{n+1} \in \hat{C}(X_{n+1})\Big) \;\geq\; 1-\alpha,$$

where the probability is over both the calibration data and the new test point. No distributional assumption on $P_{X,Y}$ beyond exchangeability.

This is a **distribution-free** guarantee. We do not need Gaussianity, no parametric form, no correctly-specified model. The only assumption is that the data (calibration + test) are exchangeable.

5.2 The split conformal construction

Split (inductive) conformal prediction (Papadopoulos et al. 2002; Vovk et al. "Algorithmic Learning in a Random World"; Lei, G'Sell, Rinaldo, Tibshirani, Wasserman, JASA 2018).**

Step 1 — Split. Partition the labelled data into a *training set* $\mathcal{D}_{\text{train}}$ and a *calibration set* $\mathcal{D}_{\text{cal}} = \{(X_1, Y_1), \ldots, (X_m, Y_m)\}$.

Step 2 — Fit. Train the model $\hat{f}$ on $\mathcal{D}_{\text{train}}$ only. (Any model works — neural net, random forest, linear regression.)

Step 3 — Compute nonconformity scores. For each calibration point, compute the residual (nonconformity score):

$$s_i = |Y_i - \hat{f}(X_i)|, \quad i = 1, \ldots, m.$$

Step 4 — Find the quantile. Let $k = \lceil (1-\alpha)(m+1) \rceil$. Set

$$\hat{q} = \begin{cases} s_{(k)} & \text{if } k \leq m, \\ +\infty & \text{if } k > m, \end{cases}$$

where $s_{(1)} \leq s_{(2)} \leq \cdots \leq s_{(m)}$ are the order statistics of the calibration scores $s_1, \ldots, s_m$. (The $+\infty$ convention applies when $\alpha < 1/(m+1)$ and ensures the coverage guarantee holds trivially in that degenerate case.)

Step 5 — The prediction interval. For a new test point X_{n+1}:

$$\hat{C}(X_{n+1}) = \Big[\hat{f}(X_{n+1}) - \hat{q},\; \hat{f}(X_{n+1}) + \hat{q}\Big].$$

5.3 The coverage theorem (proved)

Theorem (split conformal marginal coverage). *Suppose* $(X_1, Y_1), \ldots, (X_m, Y_m), (X_{n+1}, Y_{n+1})$ *are exchangeable (e.g. i.i.d.). Then*

$$P\Big(Y_{n+1} \in \hat{C}(X_{n+1})\Big) \;\geq\; 1-\alpha.$$

Proof (exchangeability and rank argument). Define the augmented score sequence by appending the test point's score as if it were known: $s_{m+1} = |Y_{n+1} - \hat{f}(X_{n+1})|$. Consider the $m+1$ scores $s_1, \ldots, s_m, s_{m+1}$.

Since $(X_1, Y_1), \ldots, (X_m, Y_m), (X_{n+1}, Y_{n+1})$ are exchangeable and $\hat{f}$ is trained only on $\mathcal{D}_{\text{train}}$ (independent of the calibration and test points by the split), the calibration and test residuals $s_1, \ldots, s_{m+1}$ are **exchangeable** as random variables.

By exchangeability, the rank of s_{m+1} among $s_1, \ldots, s_{m+1}$ is uniformly distributed on $\{1, 2, \ldots, m+1\}$:

$$P(\text{rank}(s_{m+1}) \leq j) = \frac{j}{m+1}, \quad j = 1, \ldots, m+1.$$

The event $Y_{n+1} \in \hat{C}(X_{n+1})$ is equivalent to $s_{m+1} \leq \hat{q}$, which holds iff s_{m+1} is at most the $\lceil (1-\alpha)(m+1) \rceil$-th smallest among $s_1, \ldots, s_{m+1}$, i.e. iff $\text{rank}(s_{m+1}) \leq \lceil (1-\alpha)(m+1) \rceil$. Therefore:

$$P\Big(Y_{n+1} \in \hat{C}(X_{n+1})\Big) = P(\text{rank}(s_{m+1}) \leq \lceil (1-\alpha)(m+1) \rceil) = \frac{\lceil (1-\alpha)(m+1) \rceil}{m+1} \geq 1 - \alpha. \quad \square$$

What the proof establishes. The argument uses only: 1. **Exchangeability** of the calibration + test scores — not i.i.d., not Gaussian, not parametric. 2. **The rank of a uniform-on-permutations variable** — elementary combinatorics. 3. **The split** — which makes the model $\hat{f}$ independent of the calibration/test scores.

Nothing else. This is a *combinatorial certificate*, not a statistical limit theorem.

5.4 Distributed split conformal

With M machines each holding a calibration shard of size m_k (for a total calibration set of size $m = \sum m_k$), the distributed variant is straightforward:

1. **Map:** each machine computes the local calibration scores $s_i = |Y_i - \hat{f}(X_i)|$ for its shard. (The model $\hat{f}$ is broadcast to all machines.)
2. **Reduce:** aggregate all m scores at the coordinator — or, more cheaply, use a **distributed order-statistic computation**: each machine sends the empirical CDF of its local scores (discretised to a histogram), the coordinator merges the histograms to find the global $\lceil (1-\alpha)(m+1) \rceil$-th order statistic. Cost: $O(M \cdot |\text{histogram}|)$ bytes, 1 round.
3. The global quantile $\hat{q}$ is broadcast, and prediction intervals are computed locally.

The coverage guarantee is unchanged: the calibration scores are still exchangeable, and the rank argument applies globally.

Communication cost: $O(M \cdot B_{\text{hist}})$ bytes for the histogram merge (where B_{hist} is the histogram resolution). This is *independent of* n — unlike the estimation lower bounds of §4, conformal prediction does not hit a minimax communication barrier. The distribution-free guarantee is cheaper to communicate than any parametric estimation.

5.5 Split conformal as a statistical contract (the SCT bridge)

A split conformal interval is a **promise**: for any new test point, drawn exchangeably with the calibration data, the true label lies in $\hat{C}(X_{n+1})$ with probability at least $1 - \alpha$. This is a *coverage contract* — a guarantee a model provider can write into a service-level agreement, verified empirically on the calibration set.

This is exactly the *Statistical Contract Theory* (Jordan / Bates–Jordan–Sklar–Soloff) framing of U16: a principal (model consumer) buys a statistical service from an agent (model provider);

the agent's promise is audited by the coverage rate on held-out data; the split conformal bound is the mathematical grounding of the contract. The bridge: **split conformal is the "how" of the SCT guarantee, and U16 SCT is the "why"** — incentives, auditing, strategic behaviour.

---

6. The invariant (U20 instantiation)

RAM: with all n samples, the centralised M-estimator achieves the minimax rate $\Theta(d/n)$ for smooth parametric problems. The bootstrap yields valid uncertainty quantification, but costs $O(Bn)$ storage — infeasible at scale.

PRAM: the one-shot averaged estimator matches the centralised rate when $M = o(n^{1/2})$ (smooth parametric regime). The Bag of Little Bootstraps (BLB) replaces the $O(n)$-bootstrap with s subsets of size $b = n^{0.6}$ bootstrapped to size n via multinomial weights — $O(1)$ communication rounds, total bytes $O(sb)$.

DPRAM/MPC: below $B = \Omega(d/\log M)$ bits per machine, the centralised rate is **unachievable** for d-dimensional mean estimation (Zhang–Duchi–Jordan–Wainwright lower bound). Split conformal prediction escapes the communication barrier: a distribution-free $1-\alpha$ coverage guarantee costs $O(M \cdot B_{\text{hist}})$ bytes — independent of n and d. The split is the model's statistical contract; U16 SCT prices its incentive structure.

New resource introduced: **BITS-FOR-ACCURACY** — the information-theoretic price of a statistical guarantee under communication constraints. The minimax risk under budget (n, M, B) is the unit's new box entry alongside {ops, depth, rounds, bytes, memory}.

---

7. Lab sketch — project track

Project-track lab (no weekly receipt): run these experiments free and locally in the course lab image (`labs/image/`) — or any Spark-in-Docker environment — as deep-dive and group-project material.

Experiment 1 — BLB confidence intervals vs full bootstrap (n = 10^8 surrogate). Generate a surrogate large dataset by streaming in chunks (total $n = 10^8$ rows; one column $X \sim \mathcal{N}(0,1)$, target $Y = 2X + \varepsilon$, $\varepsilon \sim \mathcal{N}(0,1)$). Estimate the OLS coefficient and its 95% CI via: (a) full bootstrap on $n = 10^5$ (feasible control case; $B = 500$); (b) BLB with $b = n^{0.6}$, $s = 5$ subsets, $r = 500$ resamples per subset; (c) the analytic standard error. Report: CI width, coverage, wall-clock time per method.

Experiment 2 — One-shot averaging M-sweep vs centralised. Simulate $n = 10^6$ i.i.d. logistic regression observations with $d = 10$. Run the centralised MLE and the one-shot averaged MLE for $M \in \{1, 4, 16, 64, 256, 1000\}$. For each M, report the ℓ_2 error $\|\bar{\theta} - \hat{\theta}_n\|_2$ and the wall-clock time. At what M does the averaging error exceed the centralised estimation error? How does it compare to the theoretical $M = o(n^{1/2}) = o(10^3)$ threshold?

Experiment 3 — Split conformal coverage empirics. Take the California housing dataset. Train a random forest on half of the data. From the other half, set aside a held-out test set of 5000 points and draw, disjoint from it, calibration sets of size $m \in \{100, 500, 2000, 5000\}$ (check the row count first: m plus the 5000 test points must fit in the held-out half). Run split conformal with $\alpha = 0.1$, and for each m estimate the empirical coverage on the test set. Plot

coverage vs m, overlay the $1-\alpha = 0.9$ line, and report interval widths. Does coverage hold for small m?

Deliverable: notebook with three experiments, plots, and a half-page cost-accounting in the three-lens vocabulary (BITS-FOR-ACCURACY for Experiments 1 and 2; communication rounds for Experiment 3).

---

8. Lean target (schematic)

The split conformal coverage proof (§5.3) is the most certifiable argument in this unit: the rank-of-a-uniform-permutation step is finite combinatorics, not asymptotic analysis.

```
-- SCHEMATIC (names align to BMLSPar.lean at cert time):
-- For m+1 exchangeable reals, the rank of the last one is Uniform({1,...,m+1}).
-- Therefore it is at most ceil((1-alpha)(m+1)) with probability >= 1-alpha.

theorem conformal_coverage (m : ℕ) (alpha : ℚ) (halpha : 0 < alpha ∧ alpha < 1)
  :
    let k := Nat.ceil ((1 - alpha) * (m + 1 : ℚ))
    (k : ℚ) / (m + 1) ≥ 1 - alpha := by
  -- Arithmetic: ceil((1-alpha)(m+1)) / (m+1) >= 1-alpha
  -- The exchangeability step (rank is uniform) is the semantic content;
  -- the above is the algebraic certificate.
  sorry  -- cert target: fill in (lab U20.L1)
```

Exercise U20.L1 (Lean, light). Prove the arithmetic lemma above: for $m \in \mathbb{N}$ and $\alpha \in (0,1) \cap \mathbb{Q}$, show $\lceil (1-\alpha)(m+1) \rceil / (m+1) \geq 1-\alpha$. This is the algebraic core; the exchangeability argument is the statistical content (written in prose in §5.3).

Exercise U20.L2 (deep, schematic). State the full coverage theorem in Lean's mathlib probability framework: define a sequence of $m+1$ exchangeable real random variables; define the rank of the last; show the rank is uniform on $\{1, \ldots, m+1\}$ by finite permutation symmetry; conclude coverage. This requires the Mathlib `MeasureTheory.Measure.ProbabilityMeasure` and `Finset.card_filter` infrastructure. Target for a module-3 group.

LLM lens — where this unit lands on the running workload (U0L):
Merging estimators (this unit) and merging experts (U0L §3) are the same divide-and-combine question in two registers: statistics asks what error the merge preserves; systems asks what bandwidth the merge costs. A student who can answer both, per shard and per token, has closed the book's loop.

9. Sources (credited; listed for archiving — not yet in the local research archive)

Kleiner, Talwalkar, Sarkar, Jordan (2014). A. Kleiner, A. Talwalkar, P. Sarkar, M. I. Jordan, *A scalable bootstrap for massive data*, Journal of the Royal Statistical Society: Series B (Statistical Methodology) 76(4):795–816, 2014. — The BLB construction, the multinomial reweighting trick, the consistency theorem, and the $\gamma = 0.6$ recommendation.

Zhang, Duchi, Jordan, Wainwright (2013, NeurIPS). Y. Zhang, J. Duchi, M. I. Jordan, M. J. Wainwright, *Information-theoretic lower bounds for distributed statistical estimation with communication constraints*, Advances in Neural Information Processing Systems 26, 2013. —

The minimax risk lower bound under per-machine bit budgets; Theorem 2 is the source of the quantitative form §4.2 states (approximately, pending a check against the paper).

Zhang, Duchi, Wainwright (2015, JMLR). Y. Zhang, J. Duchi, M. J. Wainwright, *Divide and conquer kernel ridge regression: a distributed algorithm with minimax optimal rates*, Journal of Machine Learning Research 16:3299–3340, 2015. (One-shot averaging for kernel ridge regression — a nonparametric extension of §2. The parametric averaging theorem of §2.2 is from Y. Zhang, J. Duchi, M. J. Wainwright, *Communication-efficient algorithms for statistical optimization*, Journal of Machine Learning Research 14:3321–3363, 2013.)

Vovk, Gammerman, Shafer (2005). V. Vovk, A. Gammerman, G. Shafer, *Algorithmic Learning in a Random World*, Springer, 2005. — The full conformal prediction framework; exchangeability as the sole assumption; the transductive (full) variant from which split conformal is derived.

Lei, G'Sell, Rinaldo, Tibshirani, Wasserman (2018). J. Lei, M. G'Sell, A. Rinaldo, R. J. Tibshirani, L. Wasserman, *Distribution-free predictive inference for regression*, Journal of the American Statistical Association 113(523):1094–1111, 2018. — The split conformal construction in the regression setting; marginal coverage theorem; conditional coverage discussion.

Efron (1979). B. Efron, *Bootstrap methods: another look at the jackknife*, Annals of Statistics 7(1):1–26, 1979. — The original bootstrap; baseline for the BLB comparison.

---

10. Exercises

Light (deep-unit standard):

E20.1 For the one-shot averaging theorem: state precisely the three conditions (smoothness of ℓ, positive definiteness of $\mathcal{I}(\theta_0)$, condition on M vs n). Give a one-paragraph explanation of why the condition $M = o(n^{1/2})$ is sharp by constructing an informal scenario where $M = n^{3/4}$ and the local bias term dominates the variance.

E20.2 BLB parameters. For $n = 10^9$, $b = n^{0.6}$, $s = 10$ subsets, $r = 200$ bootstrap resamples per subset: (a) compute the exact value of b; (b) for each bootstrap resample (multinomial weight draw on b points), what is the effective size of the computation vs a naive bootstrap resample of size n? (c) compute the total number of M-estimation problems solved and the total data touched in bytes (assume 8-byte doubles, a $d = 20$-dimensional design matrix).

E20.3 Split conformal. Suppose the calibration set has $m = 999$ points and $\alpha = 0.1$. (a) Compute $\lceil (1-\alpha)(m+1) \rceil$. (b) What is the guaranteed coverage? (c) Give an example of a pathological model $\hat{f}$ (not necessarily good) for which the guarantee still holds. (d) Does the guarantee hold if the calibration and test points are drawn from different distributions? Justify.

E20.4 Communication lower bound (qualitative). Suppose $d = 1000$ and $M = 100$ machines. According to the Zhang–Duchi–Jordan–Wainwright bound (stated, §4.2), below what per-machine bit budget B can the centralised rate not be achieved (up to log factors)? Give a concrete interpretation: if a machine sends a 64-bit float per coordinate, how many coordinates can it send before hitting the threshold?

Deep (Module-3 / optional):

E20.5 *Bootstrap infeasibility accounting.* Let $n = 10^8$, $B = 1000$ bootstrap resamples, and assume each record is 100 bytes. (a) Estimate the total bytes processed by a naive full bootstrap. (b) For BLB with $b = n^{0.6}$, $s = 20$, $r = 500$: estimate the bytes processed. (c) Compute the

ratio. (d) Discuss: what additional computation (solving a weighted M-estimation problem) does BLB incur per resample, and for what class of estimators is this cheap?

E20.6 *Sharpness of the $M = o(n^{1/2})$ condition.* Consider linear regression $Y = X^\top \theta_0 + \varepsilon$ with $X \sim \mathcal{N}(0, I_d)$, $\varepsilon \sim \mathcal{N}(0, \sigma^2)$. The OLS estimator is $\hat{\theta}_n = (X^\top X)^{-1} X^\top Y$. (a) Compute the exact bias and variance of the averaged estimator $\bar{\theta}$ as functions of M and n. (b) Show that $\bar{\theta}$ is exactly *unbiased* for **all** M (each shard's OLS is conditionally unbiased) — the averaging bias that drives the $M = o(n^{1/2})$ theory is identically zero here. (c) The variances do NOT match in finite samples: using $\mathbb{E}[(X_k^\top X_k)^{-1}] = \frac{1}{n/M-d-1} I$ (inverse-Wishart; Jensen makes it exceed $(\mathbb{E}[X_k^\top X_k])^{-1}$), show $\mathrm{Var}(\bar{\theta}) = \frac{\sigma^2 d}{n - M(d+1)}$ versus $\frac{\sigma^2 d}{n-d-1}$ for full-data OLS, so the ratio tends to 1 whenever $M(d+1) = o(n)$ — a far weaker condition than $M = o(n^{1/2})$. (d) Explain why linear regression is this degenerate for the averaging theory, and identify what property of OLS makes it so. (*Hint:* the nonlinearity of a general M-estimator is what creates the shard bias; OLS has none, leaving only a vanishing variance inflation.)

E20.7 *Conditional vs marginal coverage.* The split conformal guarantee is *marginal*: the coverage holds on average over X_{n+1}. Give an example (a concrete joint distribution of X, Y and a model $\hat{f}$) where marginal coverage is exactly $1 - \alpha$ but the conditional coverage $P(Y \in \hat{C}(X) \mid X = x)$ is $< 1 - 2\alpha$ for some values of x. Discuss the practical implications for model deployment and how U16 SCT addresses the gap (if at all).

Part I

Laboratory exercises

Chapter 27

Toolchain + proof-checker

Do this one first — every later lab builds on it, and it must be complete before you start the course proper. Runnable steps: the ***`lab0-toolchain/`*** *folder (clone URLs in Start Here). It has no algorithmic content: the deliverable is a* ***working toolchain****, receipted.*

What you set up

1. **Docker** — the container runtime every lab uses. Install it (official links are in Start Here); verify with `docker run --rm hello-world`.
2. **git** — `git --version`; used from the next lab on to clone and to replay your own contribution log.
3. **an ssh key** — `ssh-keygen -t ed25519` if `ls ~/.ssh/*.pub` shows nothing; used later for sovereign git.
4. **run the lab** — `bash run-lab.sh`. It pulls both course images, smoke-tests the lab image, and asks the Lean proof-checker to accept the one-line theorem in `Check.lean` (that $1 + 1 = 2$), then writes a receipt.

The one idea worth taking away

A proof-checker earns its authority only because it can say **no**. Open `Check.lean`, change the `2` to a `3`, commit the change (`git add Check.lean`, then `git commit -m "break: 1+1=3"`; `run-lab.sh` refuses to run on uncommitted changes, because a receipt is bound to a commit), rerun, and watch it **FAIL**: a checker that cannot reject certifies nothing (this is the whole point of U0P). Change it back, commit again, rerun, and keep the passing receipt. That single experiment — accept the true statement, reject the false one — is the seed of every "machine-checked" claim in this book.

Submit

Upload the passing receipt (`receipts/receipt-lab0-*.txt`) the run prints. The submission mechanics — the same for every lab — are in Start Here.

Trouble? A ***`permission denied ... docker.sock`*** *on Linux means you skipped the post-install step; a corporate laptop that blocks Docker Desktop needs the fallback path — ask in the course discussion early, not the night before you need it.*

Chapter 28

Ping-pong + tree vs ring allreduce (the alpha+beta*n model)

Ideally in course week 3. Runnable code + step-by-step: the `lab1-alpha-beta/` *folder of the course repository (see Start Here for the clone URLs). Prerequisite: F.E4 (the cost model) and U00N (what a message actually is).*

The model you measure

The whole communication-cost story of this course rests on a two-parameter model: sending an n-byte message between two machines costs

$$T(n) = \alpha + \beta\, n,$$

where α is the per-*message* cost (latency, seconds) and β is the per-*byte* cost (the reciprocal of bandwidth). Chapter F.E4 derives why this linear form governs everything from a shuffle to an all-reduce. In this lab you *measure* your own machine's α and β — between two Docker containers on one host — and see how well the straight line actually fits.

What you run

The lab starts two containers on a single Docker network — a server and a client — and the client times round trips for message sizes from 1 byte to 16 MB (medians over many repetitions). It then solves the least-squares fit for α, β (the normal equations of U8, five lines of code you can read in `measure.py`) and writes a receipt.

What to observe (this is the point — read it before you upload)

1. **`alpha_us` vs `beta_ns_per_byte`.** On a loopback container bridge α is tiny (microseconds) and the effective bandwidth is huge — this is the *baseline*. A real cluster network has α thousands of times larger. It is the **ratio** α/β, not either number alone, that decides whether tree or flat aggregation wins (F.E4 §3). Do not be alarmed if your fitted α comes out slightly **negative**: a physical latency cannot be negative, but a least-squares *intercept* can be, when the true α is tiny and the large-message points steer the line. That is measurement noise plus extrapolation — and exactly why the receipt also reports R^2. A

model is a fit, not a truth; say so in your one-sentence comment and you have understood the lab.

2. **`r_squared`.** If it is not close to 1, the linear model missed something on your machine (a cache effect, a power-management kink). That observation is worth a sentence.

3. **Tree vs ring all-reduce (no new code — the analysis half).** Plug your measured α, β into the two classic all-reduce cost models for p workers and an n-byte vector:

$$T_{\text{tree}} \approx 2\log_2(p)\,(\alpha + \beta n), \qquad T_{\text{ring}} \approx 2(p-1)\,\alpha + 2\tfrac{p-1}{p}\,\beta n.$$

   Your fitted α comes from round trips (the message out plus a 1-byte reply back), so it is about twice the one-way α these formulas assume: use $\alpha/2$ here and in item 4 (β needs no change, since the n bytes travel one way). For *your* numbers, which wins at $n = 1$ KB, which at $n = 100$ MB, and at what n do they cross? Put the crossover in your receipt comment. This is F.E4's tree-vs-flat trade-off made personal: small messages are latency-bound (tree wins), large ones are bandwidth-bound (ring wins).

4. **The one-line takeaway.** $p - 1$ one-byte messages cost $\approx (p-1)\alpha$; one n-byte stream costs $\alpha + \beta n$. Which dominates for your α, β at $n = 1$ MB, $p = 16$?

Submit

Run the lab (`bash run-lab.sh` in the `lab1-alpha-beta/` folder); it prints a receipt path and appends a line to `sigma-log.jsonl`. Upload that receipt, with your one-sentence comments (the α, β note and the tree-vs-ring crossover). The submission mechanics — the same for every lab — are in Start Here.

Looking further: actually implementing and racing tree vs ring all-reduce across real containers is an optional deep-dive later in the course; it is not part of this receipt.

Chapter 29

Sample sort on Spark

Ideally in course week 4. Runnable steps: the `lab2-sample-sort-spark/` folder (clone URLs in Start Here). Prerequisite: U2 (sample sort) and U5 (Spark, the shuffle).

What you build

You implement **sample sort** (U2) on **Spark** (U5), verify it, and race it against Spark's built-in `sortBy`. Spark runs in `local[*]` mode inside one container — every CPU core is a worker — so this is safe on an 8 GB laptop.

What to observe (before you upload)

1. **The one shuffle.** Read `sample_sort.py` (~130 lines) and find the single `partitionBy` — that is *the* data movement, and everything else is local. U2's claim is that sampling $32p$ keys is enough to make all p buckets balanced with high probability; your receipt's `balance` number is that claim, measured on your machine.
2. **Verification never trusts the algorithm.** The checker asserts global order *across* bucket boundaries **and** that the multiset of records survived the shuffle (count + checksum). Grading is on the receipt's `verified=True` — a sort that loses or duplicates a record fails, however fast it is. This is the evidence-trail discipline of the whole course in miniature: the *measured, checked* fact is the deliverable, not the code's say-so.
3. **The race.** Is your sample sort faster or slower than `sortBy`? Within noise they should be close — `sortBy` *is* a range-partition sort. Both move all n records exactly once; the constant factors are where the interest lives. A sentence on why is worth writing.

Submit

Run the lab (`bash run-lab.sh`); upload the receipt (`receipts/receipt-lab2-*.txt`) it prints. Optional, if your laptop has 16 GB: rerun with a bigger input (`bash run-lab.sh 10000000 8`) and note whether the balance factor and speed ratio hold at scale. The submission mechanics — the same for every lab — are in Start Here.

Chapter 30

Streaming sketches: reservoir merge, Count-Min, HyperLogLog

Runnable code + step-by-step: the `lab3-streaming-sketches/` folder of the course repository (clone URLs in Start Here). Prerequisite: U4 (sampling and streaming) and U5 (Spark, the shuffle).

The claims you check

Unit U4 adds **memory** to the accounted resources: one pass over a stream of n items, and a memory budget $m \ll n$. This lab measures, on your own laptop, the guarantees the unit proves or states:

- a **reservoir** of k items is a uniform sample of its stream — each item is in it with probability exactly k/n (U4 §3.3); for two reservoirs of disjoint shards, the hypergeometric split of U4 §6.1 is the exact merge;
- a **Count-Min sketch** of $d \times w$ counters, with $w = \lceil e/\varepsilon \rceil$ and $d = \lceil \ln(1/\delta) \rceil$, never undercounts, and overcounts each item by more than $\varepsilon \|f\|_1$ with probability at most δ (U4 §4.3);
- **HyperLogLog** with m registers estimates the number of distinct items with standard error about $1.04/\sqrt{m}$ (U4 §5.3).

In words: a sample, a frequency table and a distinct count, each in memory that does not grow with the stream — and each with a guarantee you can check.

What you run

One stream of $n = 10^6$ items from a skewed (Zipf) distribution, split into 8 shards of **unequal** length, as a real stream is split by time or by source. The reservoirs and the Count-Min tables are built on each shard and then merged — the per-partition-summary-plus-one-aggregate pattern of U4 §7.1. The Python program (`streaming_sketches.py`, PySpark) keeps Vitter's reservoir, the Count-Min construction and both reservoir merges as short functions you can read next to U4; the sort-based sample and HyperLogLog are Spark's own (`orderBy(rand()).limit(k)`, `approx_count_distinct`). A short Scala program (`countmin_library.scala`) then runs Spark's built-in Count-Min on the same stream. These are the experiments of U4 §9, sized for a laptop.

What to observe (this is the point — read it before you upload)

1. **Three samples.** Each merged sample of $k = 1000$ items is tested for being evenly spread over the ten tenths of the stream (`per_tenth_of_stream`, about 100 each; a chi-square test). The exact merge passes (small `chi2`). The naive merge — pool the eight reservoirs and pick k — fails badly: the first tenths of the stream get far more than 100 (large `chi2`). Same reservoirs, different merge. The sort-based sample `orderBy(rand()).limit(k)` passes too; compare the bytes Spark shuffles for it (collected, or written out) with a full sort of the stream, on the `shuffle_bytes` line.

2. **A small table that never lies low.** With $\varepsilon = 0.001$ and $\delta = 0.01$, the sketch has $5 \times 2719 = 13\,595$ counters, against the exact table's `distinct_items`. Check `never_undercounts=True` (U4 §4.3: each counter contains the item's own count, so the minimum cannot fall below it) and `fraction_of_items_over_eps_n` against δ — the theorem is about each item; the maximum overcount is an observation. Every true heavy hitter at $\phi = 2\varepsilon$ is reported (U4 §4.4), with a few reported items that are not heavy. The shard tables merge by addition: the monoid of U4 §6.2.

3. **Error against registers.** HyperLogLog runs on the stream, and on synthetic streams with exactly $D = 10^3, \ldots, 10^6$ distinct values. For each requested accuracy `rsd`, compare the measured `relative_error` with the book's $1.04/\sqrt{m}$ (U4 §5.3). A standard error is a typical size, not a bound: some errors lie above it, most below.

4. **The library's sketch, in Scala.** Spark's own `df.stat.countMinSketch` — a Scala and Java API, not in PySpark (U4 §7.3) — sizes its table 2000×7 for the same ε and δ, against the book's 2719×5: $w = \lceil 2/\varepsilon \rceil$ and $d = \lceil \log_2(1/\delta) \rceil$, the same Markov argument with a per-row failure chance of $\frac{1}{2}$ instead of $1/e$. That argument needs each row's collision probability to be at most $1/w$ and the rows to be independent; Spark's simple multiplicative hash does not provably give either, so its δ is a heuristic. Never undercounting holds regardless.

Under the hood

Spark is a Scala program on the Java virtual machine; PySpark drives the same engine from Python, so a DataFrame query is planned and run identically from either language, while Python functions handed to Spark run in separate Python worker processes. The Python part hands the stream to the Scala part as a Parquet directory: a columnar, compressed format with a schema and per-chunk statistics, written as one part-file per partition. Rows move between partitions only at a *shuffle* — an Exchange in the physical plan, which `df.explain()` prints — and the lab reads the shuffled bytes from Spark's own task metrics. The lab's README explains each of these in a paragraph.

Submit

Run the lab (`bash run-lab.sh` in the `lab3-streaming-sketches/` folder); it prints a receipt path and appends a line to `sigma-log.jsonl`. Upload the receipt, with one or two sentences answering each question in the comment box of the upload (the receipt itself stays unedited):

1. *Predict, from the shard lengths in the first `MEASURED` line and U4 §3.3, about how many of the naive merge's $k = 1000$ items should fall in the first tenth of the stream. Compare with your receipt, and say in one sentence why that tenth is over-represented.*
2. *Why does `orderBy(rand()).limit(k)` shuffle only about a hundred kilobytes when written out, and nothing when collected (rows then reach the driver as task results), while a full*

`orderBy(rand())` shuffles the whole stream? (Where can the k smallest keys of the whole stream be?)

The submission mechanics — the same for every lab — are in Start Here.

Chapter 31

Blocked matrix multiplication: block size and sparsity

Runnable code + step-by-step: the `lab4-blocked-matmul/` folder of the course repository (clone URLs in Start Here). Prerequisite: U6 (matrix multiplication) and U5 (Spark, the shuffle). This lab is in Scala, Spark's own language.

The claims you check

Unit U6 treats matrix multiplication on a cluster as a **communication** problem. The arithmetic is fixed at $2n^3$ operations; what varies is how many bytes move. In the 2D layout of U6 §4.1, each of p processors on a $\sqrt{p} \times \sqrt{p}$ grid needs a block-row of A and a block-column of B, and the total communication is $\Theta(n^2\sqrt{p})$: more processors, more bytes. Spark's `BlockMatrix.multiply` (U6 §4.4) is such a layout: blocks are shuffled to where they are multiplied, multiplied locally, and the partial blocks summed. For sparse matrices, U6 §4.4 also describes the join pattern of U5.

What you run

Two random dense 2048×2048 matrices are multiplied with `BlockMatrix`, with blocks of 128, 256, 512 and 1024 rows and columns; Spark's own task metrics give the shuffle bytes, and every product is checked with Freivalds' randomized test ($Cx = A(Bx)$ for a random x, in $O(n^2)$ work instead of $O(n^3)$). Then two random sparse 1024×1024 matrices, at densities 0.001 to 0.1, are multiplied twice: as sparse blocks, and as a join of their (i, k, a) and (k, j, b) entries on k followed by a sum per (i, j).

What to observe (this is the point — read it before you upload)

1. **Bytes and partitions.** For each block size, compare `matrix_copies_moved` (shuffle bytes divided by one matrix) with `partitions` (the partitions holding the product's blocks). The bytes do not follow the block size one-for-one.
2. **Two ways to multiply sparse matrices.** Both methods give the same product (the same Cx for a random x). Compare how `join_shuffle_MB` and `blocks_shuffle_MB` grow with the density, next to `nnz_A` and `partial_products` (the number of pairs A_{ik}, B_{kj} that are both non-zero).

3. **Time is noisy, bytes are not.** The `seconds` change from run to run; the bytes are the same on every machine.

Under the hood

A `BlockMatrix` is an RDD of $((\text{blockRow}, \text{blockCol}), \text{block})$ pairs. Spark lays the blocks of C on a grid of partitions, copies blocks of A and B to the partitions that need them, multiplies pairs of blocks locally, and sums partial blocks with `reduceByKey`. The program counts bytes with a *listener*, a callback Spark calls after every task with its metrics. Spark compresses shuffle data, so mostly-zero blocks move few bytes. The lab's README has a short guide to reading the Scala.

Submit

Run the lab (`bash run-lab.sh` in the `lab4-blocked-matmul/` folder); it prints a receipt path and appends a line to `sigma-log.jsonl`. Upload the receipt, with one or two sentences answering each question:

1. *Block sizes 128 and 256 move the same number of matrix copies (about 17: 17.03 and 17.02), while 512 and 1024 move fewer. Why? Derive the number of copies from* ***`partitions`****.* (The 2D layout of U6 §4.1.)
2. *Why does the join's shuffle grow so much faster with the density than the number of non-zeros does, while the blocks' shuffle does not?* (Compare `join_shuffle_MB` with `partial_products` and with `nnz_A`.)

The submission mechanics — the same for every lab — are in Start Here.

Chapter 32

Distributed gradient descent and parameter averaging

Runnable code + step-by-step: the `lab5-distributed-sgd/` *folder of the course repository (clone URLs in Start Here). Prerequisite: U9 (gradient descent and SGD) and U5 (Spark). This lab is in Python (PySpark).*

The claims you check

Unit U9 proves that gradient descent with step $1/L$ on an L-smooth, μ-strongly convex function shrinks the gap $f(w_k) - f_*$ at least by the factor $1 - \mu/L$ per step (Theorem 9.5); that minibatch SGD with a constant step stalls at a noise floor (§4.3); and that distributing the computation costs one round of communication per combination of the workers' results (§6.1). Parameter averaging (§6.4) combines models instead of gradients, after several local steps.

What you run

Logistic regression with an L2 penalty on $n = 200\,000$ examples with $d = 50$ features, split over 8 workers; L, μ and the exact optimum f_* are computed first. Then full-batch gradient descent, one round of communication per step; then parameter averaging with $H = 1, 10, 100$ local SGD steps per round, for a budget of 10 rounds. Same problem and the same algorithms run on Ray or with MPI all-reduce; the lab uses Spark.

What to observe (this is the point — read it before you upload)

1. **The theorem holds, with room to spare.** The measured gap stays below $(1 - \mu/L)^k$ times the first gap at every step, but the measured shrink factors per step (over steps 10–20 and 30–40) are far smaller than the bound's, and they grow: GD slows down. Compare $\mu = \lambda$ with the `curvature` line: the smallest curvature of f at the start, at step 40 and at the optimum.
2. **Local work per round.** With the same 10 rounds, compare the gaps for $H = 1, 10, 100$ and for plain gradient descent. Look at the last rounds of $H = 100$.
3. **What a round costs.** Each worker sends one vector of d numbers per round, whatever H is; for a 7-billion-parameter model, U9 §6.1's ring all-reduce budget gives the bytes and seconds per step.

Under the hood

Each partition holds its examples as one NumPy array, so a partition's gradient is a few matrix–vector products; Spark sums the partial results with `treeAggregate` (a reduction, as in U1); in the averaging part it sends the model to the workers with a `broadcast`. One combination of the workers' vectors is one round. The lab's README explains each in a paragraph.

Submit

Run the lab (`bash run-lab.sh` in the `lab5-distributed-sgd/` folder); it prints a receipt path and appends a line to `sigma-log.jsonl`. Upload the receipt, with one or two sentences answering each question:

1. *Theorem 9.5's bound promises a factor of about 0.9998 per step, but over steps 30–40 GD shrinks the gap by about 3% per step — and faster still over steps 10–20. Why is the bound so pessimistic, and why is GD slowing down?* (Compare λ with the three numbers on the `curvature` line.)
2. *With 10 rounds of communication, $H = 10$ needs all 10 rounds to reach the gap that $H = 100$ reaches after one (8.1×10^{-3}). Why — and why does $H = 100$ then stop improving?* (What a round costs, and U9 §4.3 for a constant step size.)

The submission mechanics — the same for every lab — are in Start Here.

Chapter 33

PageRank two ways: naive vs co-partitioned RDDs, and GraphX

Runnable code + step-by-step: the `lab6-pagerank-two-ways/` folder of the course repository (clone URLs in Start Here). Prerequisite: U12 (graphs at scale) and U5 (Spark, the shuffle). Python (PySpark) and Scala (GraphX).

The claims you check

Unit U12 computes PageRank by power iteration, as a join of the link table with the rank vector, a `flatMap` of contributions and a `reduceByKey` (§4.1). Its theorem in §4.2: if the link table is hash-partitioned once and kept, and the ranks are updated with `mapValues`, the join is narrow — the link table no longer moves. §6.4 describes GraphX, which runs the same computation over an edge table and a vertex table.

What you run

A power-law graph of 10^5 vertex ids and about 9.4×10^5 edges; PageRank for 10 iterations (links followed with probability 0.85, a random jump with probability 0.15, U12 §2.1's α), three ways: the RDD loop of §4.1 with the partitioning lost by a `map` (naive), the same loop co-partitioned as in §4.2, and GraphX's `staticPageRank` in Scala on the same graph, on 8 edge partitions. Every RDD iteration runs as two Spark jobs, the join and the contributions, and Spark's task metrics give the shuffle bytes of each.

What to observe (this is the point — read it before you upload)

1. **Same ranks, different bytes.** The naive and co-partitioned versions give the same ranks; compare their `join_shuffle_MB_per_iteration` and `contrib_shuffle_MB_per_iteration`. (The first iteration or two moves a little less: every starting rank is $1/n$, and Spark compresses what it shuffles.)
2. **The classic Spark loop.** A vertex with no in-links drops out of the rank table after the first iteration, and the rank of a vertex with no out-links goes nowhere, so `rank_sum` is about 0.91 rather than 1 (U12 §2.1's chain gives every vertex the teleport term and spreads dangling rank uniformly; this loop does neither).
3. **A third way.** GraphX finds the same top vertices (`same_top100_as_rdd`); compare its bytes per iteration and its time with the two RDD versions.

Under the hood

A key–value RDD can carry a *partitioner*; two RDDs with the same partitioner join partition by partition, without a shuffle. `mapValues` keeps the partitioner, `map` loses it. Spark runs in the JVM; when an RDD job calls a Python function, every record is serialised to a Python worker and back. GraphX is written in Scala and stores its edges in a vertex-cut layout. The lab's README explains each in a paragraph.

Submit

Run the lab (`bash run-lab.sh` in the `lab6-pagerank-two-ways/` folder); it prints a receipt path and appends a line to `sigma-log.jsonl`. Upload the receipt, and in the comment box of the upload page answer each question in one or two sentences:

1. *In the co-partitioned version the join moves nothing, yet in both versions the contributions still move about 4.1 MB every iteration. Why can co-partitioning remove the join's shuffle, but not this one?* (U12 §4.2's proof and its "honest scope", and the edge-cut of U12 §5.3.)
2. *GraphX moves about as many bytes per iteration as the RDD versions (about 5.2 MB, against 4.1 MB co-partitioned), yet its 10 iterations take a few seconds instead of half a minute. If the bytes do not explain the difference, what does?* ("Under the hood"; count the records each iteration handles.)

The submission mechanics — the same for every lab — are in Start Here.

Part II

Preliminaries — enter as needed

Chapter 34

Mathematical preliminaries

F.M1 — Asymptotics, recurrences, and sums (a recap)

*Foundation chapter (Pillar 1, Mathematical Foundations). A **memory-jog** for the three pieces of asymptotic bookkeeping every cost analysis uses. If a unit's Entry Kit says "needs F.M1", it means one of the statements boxed below. This is a root: the units assume it and do not re-teach it.*

1. Asymptotic notation

For eventually-positive $f, g : \mathbb{N} \to \mathbb{R}_{>0}$:

- $f = O(g)$ — *at most, up to a constant:* $\exists\, c > 0,\ n_0 :\ f(n) \le c\, g(n)$ for all $n \ge n_0$.
- $f = \Omega(g)$ — *at least:* $g = O(f)$.
- $f = \Theta(g)$ — *same order:* $f = O(g)$ **and** $f = \Omega(g)$.
- $f = o(g)$ — *strictly smaller:* $f(n)/g(n) \to 0$. (ω is the dual.)

The point is to **discard constants and lower-order terms** and compare growth. `MATHS->CS:` "this sort is $O(n \log n)$" is a statement about the RAM operation count of F.C1. `CS->MATHS:` treat $O(g)$ as *the set* of functions dominated by g; "$f = O(g)$" is really $f \in O(g)$.

The one rigorous statement units use: to prove $f = O(g)$, *exhibit the constant* — a specific c and n_0. To refute it, show that $f(n)/g(n)$ is unbounded — for instance, that $f(n)/g(n) \to \infty$.

2. Recurrences and the master theorem

Divide-and-conquer costs satisfy $T(n) = a\,T(n/b) + f(n)$ with $a \ge 1$, $b > 1$ (a subproblems of size n/b, plus $f(n)$ to split/combine). Compare $f(n)$ with the *watershed* $n^{\log_b a}$:

- **leaves dominate:** if $f(n) = O(n^{\log_b a - \varepsilon})$, then $T(n) = \Theta(n^{\log_b a})$.
- **balanced:** if $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a} \log n)$.
- **root dominates:** if $f(n) = \Omega(n^{\log_b a + \varepsilon})$ (and a regularity condition holds), then $T(n) = \Theta(f(n))$.

Examples the units hit: mergesort $T(n) = 2T(n/2) + \Theta(n) = \Theta(n \log n)$ (balanced case, U2); a work-efficient parallel scan's $\Theta(\log n)$ **depth** is the height of the same balanced tree (U1).

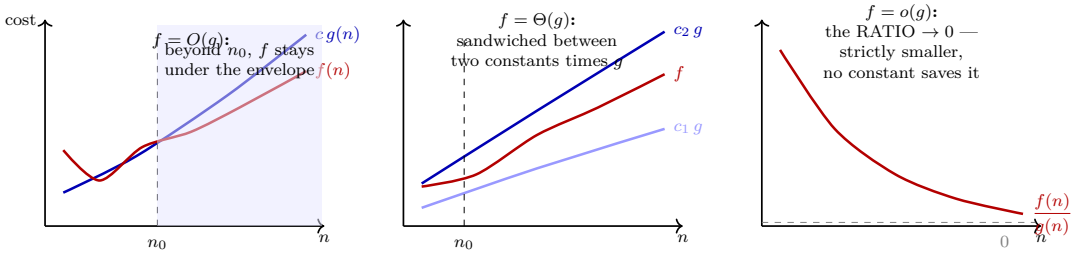


Figure 34.1: What the asymptotic quantifiers *look like*. Left: $f = O(g)$ promises an envelope $c\,g$ that f never exceeds *beyond some* n_0 (early crossings, shaded white, are allowed and irrelevant). Middle: $f = \Theta(g)$ is the two-sided sandwich — same order exactly. Right: little-o is stronger than any envelope: the ratio itself vanishes. Every bound in this book is one of these three pictures with named constants.

3. Sums that recur

- **Geometric series** $\sum_{i=0}^{k} r^i = \dfrac{r^{k+1}-1}{r-1}$; for $r > 1$ it is $\Theta(r^k)$ — *the top level dominates* (why up-/down-sweep work is $\Theta(n)$, U1).
- **Powers of two** $\sum_{i=0}^{k-1} 2^i = 2^k - 1$ — the exact identity the scan/tree certs use.
- **Harmonic series** $H_n = \sum_{i=1}^{n} 1/i = \ln n + O(1)$ — the log in randomized-pivot analyses.
- **Arithmetic series** $\sum_{i=1}^{n} i = \frac{n(n+1)}{2} = \Theta(n^2)$.

Cert pointer: the powers-of-two identity is exactly what the U1 scan cert (`ScanCost`, `blelloch_scan_cost`) proves machine-checked over $\mathbb{N}$ — asymptotics on paper, an exact closed form in the prover (F.C3).

Where this is used

Everywhere a cost is stated: U0 (cost lens), U1 (scan work/depth sums), U2 (sort recurrence), U3 (bounds), U4/U15/U20 (rates). Every atlas edge `depends F.M1` is a unit standing on one of the three boxes above.

F.M2 — Linear algebra: the six facts the middle of the book runs on

Memory jog for: U6 (matrix multiplication), U7 (SVD, random projections), U8 (solvers, least squares), U9/U18 (gradients of quadratics; attention is matrix products). This is not a linear algebra course; it is the six facts those chapters lean on, each stated so you can check you still own it.

1. A matrix is three things at once

An $m \times n$ real matrix A is simultaneously: (i) a **table** of mn numbers (the storage view — U6's communication costs count these bytes); (ii) a **linear map** $x \mapsto Ax$ from $\mathbb{R}^n$ to $\mathbb{R}^m$ (the algebraic view — composition of maps *is* matrix multiplication, which is why matmul is associative and why $(AB)C$ vs $A(BC)$ can differ astronomically in cost while agreeing in value); (iii) a **sum of outer products** $A = \sum_k a_{\cdot k}\, e_k^\top$ (the decomposition view — the form in which U6's outer-product algorithm and U7's low-rank approximations think). Fluency means switching views mid-sentence. Figure 34.2 draws the three views on one 2×2 matrix; its third panel reads Ax column by column, $Ax = \sum_k x_k\, a_{\cdot k}$, the outer-product view applied to one vector.

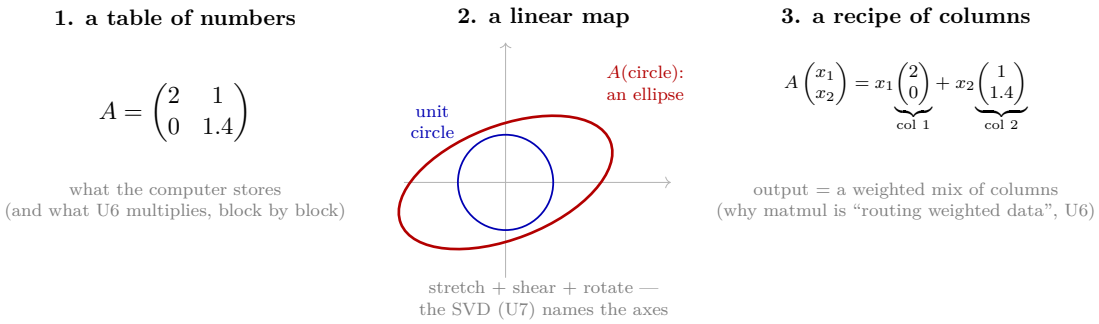


Figure 34.2: One matrix, three simultaneous readings: a stored table of numbers, a geometric map (the unit circle goes to an ellipse — the SVD of U7 is exactly the ellipse's axes), and a recipe mixing columns. Every linear-algebra argument in this book picks whichever reading makes the step obvious — and the fluency this chapter builds is switching between them mid-proof.

2. Rank

$\text{rank}(A)$ = dimension of the column space = dimension of the row space. Rank r means A can be written with r outer products and no fewer — the entire premise of U7 (compress by keeping few outer products) in one sentence.

3. Norms — three for vectors, two for matrices

Vectors: $\|x\|_1, \|x\|_2, \|x\|_\infty$, with $\|x\|_\infty \le \|x\|_2 \le \|x\|_1$. Matrices, the two the book uses:

- **Frobenius norm:** $\|A\|_F^2 = \sum_{ij} a_{ij}^2$ — the "energy"; computable entrywise, distributes over blocks (U6's per-shard accounting).
- **Operator (spectral) norm:** $\|A\|_2 = \max_{\|x\|_2=1} \|Ax\|_2$ — the worst-case stretch; the norm in which convergence of iterative solvers (U8) and stability of attention layers (U18) are stated.

Always $\|A\|_2 \le \|A\|_F \le \sqrt{r}\,\|A\|_2$.

4. Orthogonality and projection

Q has orthonormal columns when $Q^\top Q = I$; then $\|Qx\|_2 = \|x\|_2$ (lengths survive). The **projection** of b onto the column space of full-rank A is $A(A^\top A)^{-1}A^\top b$ — equivalently: the residual is orthogonal to every column. That orthogonality condition IS the normal-equations derivation of least squares; U8 does nothing deeper, just at scale.

5. Symmetric and PSD matrices

Symmetric $M = M^\top$ have real eigenvalues and an orthonormal eigenbasis (spectral theorem). M is **positive semidefinite (PSD)** when $x^\top Mx \ge 0$ for all x — equivalently all eigenvalues ≥ 0 — equivalently $M = B^\top B$ for some B. Every Gram matrix $A^\top A$ is PSD; its **condition number** $\kappa = \lambda_{\max}/\lambda_{\min}$ is the single scalar that predicts how fast U8's iterative solvers converge and how much U9's gradient descent zig-zags.

6. Why the SVD exists (the fact U7 stands on)

Theorem (SVD). Every $m \times n$ real A factors as $A = U\Sigma V^\top$ with U, V orthonormal-columned and $\Sigma = \text{diag}(\sigma_1 \ge \cdots \ge \sigma_r > 0)$.

Why it exists, in one breath: $A^\top A$ is symmetric PSD, so it has an orthonormal eigenbasis v_i with eigenvalues $\sigma_i^2 \geq 0$ (spectral theorem, §5); set $u_i = Av_i/\sigma_i$, check the u_i are orthonormal, and assemble. The SVD is the spectral theorem pushed through a rectangle.

Its consumer-facing corollary — keep the top k triples and you have the **best** rank-k approximation in both $\|\cdot\|_2$ and $\|\cdot\|_F$ (Eckart–Young) — is the mathematical license for everything U7 does distributed.

7. One exercise

E.FM2.1 Let A be $n \times n$ of rank 1, so $A = uv^\top$. Compute $\|A\|_F$ and $\|A\|_2$ in terms of $\|u\|_2, \|v\|_2$; give the SVD of A explicitly; and count the flops to apply A to a vector naively vs. using the factored form. (The factored-form count is U6's outer-product lesson in miniature.)

F.M3 — Convexity, gradients, and a first look at duality

Memory jog for: U9 (GD/SGD — every rate theorem's hypotheses live here), U19 (ADMM — the Lagrangian is its native language), U13 (k-means' non-convexity is the point), U18 (what Adam is trying to descend). If "L-smooth and μ-strongly convex" triggers the picture of a function trapped between two parabolas, skip this chapter.

1. Gradients — the local linear story

For differentiable $f : \mathbb{R}^d \to \mathbb{R}$, the gradient $\nabla f(x)$ is the vector of partials, and $f(x+h) = f(x) + \nabla f(x)^\top h + o(\|h\|)$: locally, f is the plane with slope ∇f. Two consequences the units use daily: $-\nabla f(x)$ is the steepest descent direction, and $\nabla f(x^*) = 0$ at any interior minimizer. For the book's favourite function class — quadratics $f(x) = \frac{1}{2}x^\top Mx - b^\top x$ with M PSD (F.M2 §5) — $\nabla f(x) = Mx - b$, so "solve the linear system" and "minimize the quadratic" are the same task; that identification is the bridge between U8 and U9.

2. Convex sets and functions

A set is **convex** if it contains the segment between any two of its points. A function is **convex** if its graph sits below its chords: $f(\lambda x + (1-\lambda)y) \leq \lambda f(x) + (1-\lambda)f(y)$. For differentiable f, convexity is equivalent to the tangent plane being a global under-estimator:

$$f(y) \;\geq\; f(x) + \nabla f(x)^\top (y - x) \quad \text{for all } x, y,$$

which upgrades the local fact $\nabla f(x^*) = 0$ to the global statement **stationary = optimal**. That upgrade is the entire reason convexity buys theorems: U9's convergence proofs are inequality-chains hanging off this one line. Figure 34.3 draws the chord above the graph and the tangent below it, the tangent being the global certificate.

3. The two-parabola sandwich: smoothness and strong convexity

- f is **L-smooth** when $f(y) \leq f(x) + \nabla f(x)^\top (y-x) + \frac{L}{2}\|y-x\|^2$ (a parabola of curvature L sits **above** f at every point).
- f is **μ-strongly convex** when $f(y) \geq f(x) + \nabla f(x)^\top (y-x) + \frac{\mu}{2}\|y-x\|^2$ (a parabola of curvature μ sits **below**).

For the quadratic of §1, $L = \lambda_{\max}(M)$ and $\mu = \lambda_{\min}(M)$ — so the ratio L/μ **is** the condition number κ of F.M2 §5, and the same scalar that predicted U8's solver convergence reappears verbatim in U9's gradient-descent rate $(1 - \mu/L)^t$. One number, two chapters, one phenomenon.

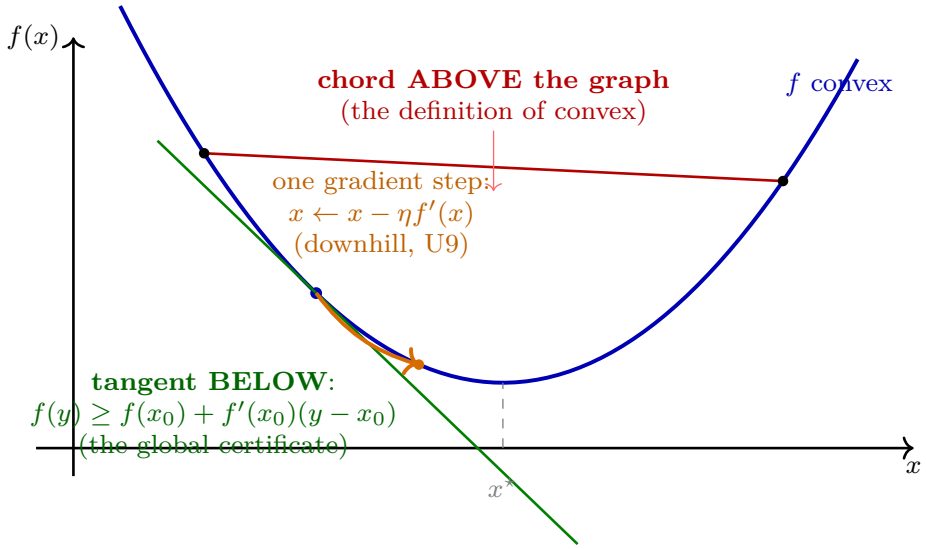


Figure 34.3: Convexity in one picture: every chord sits *above* the graph (the definition), so every tangent sits *below* it — which turns the derivative into a **global** certificate: the tangent inequality lower-bounds f everywhere from one local measurement. That is why following the gradient downhill (orange, U9) cannot be fooled: in a convex bowl there is nowhere misleading to go, and the tangent tells you how far you still are from $x^\star$.

4. Duality-lite: the Lagrangian as a price

For $\min_x f(x)$ subject to $Ax = b$, form the **Lagrangian** $\mathcal{L}(x, y) = f(x) + y^\top(Ax - b)$. Read y as a **price per unit of constraint violation**: the dual function $g(y) = \min_x \mathcal{L}(x, y)$ asks "what is the best I can do if violations merely cost me at rate y?", and **weak duality** $g(y) \leq f(x^*)$ says any price vector yields a certified lower bound. U19's ADMM is nothing but an alternating scheme for finding the price at which the decoupled, per-machine subproblems voluntarily agree — the "consensus star" of its sticky figure, with y living on the edges.

5. What non-convexity costs (and why the book still trains)

k-means (U13) and deep networks (U18) minimize non-convex objectives: stationary no longer implies optimal, and no algorithm inherits §2's guarantees. What survives — and what U9's SGD theorems actually promise there — is convergence to **stationarity** ($\|\nabla f\| \to 0$) plus whatever the problem's structure volunteers. The honest reading of "SGD trains the model": the theorem delivers a stationary point; the empirical loss curve delivers the rest. The book keeps these two claims typographically separate (U0P honesty flags), and F.X1/U16 turn that separation into contract language.

6. One exercise

E.FM3.1 For $f(x) = \frac{1}{2}\|Ax - b\|_2^2$: compute ∇f; show f is L-smooth with $L = \|A^\top A\|_2$ and μ-strongly convex with $\mu = \lambda_{\min}(A^\top A)$ (when positive); and conclude the gradient-descent contraction factor for least squares in terms of $\kappa(A^\top A)$. (This is the U8↔U9 bridge, walked by hand once so both chapters can cite it.)

Chapter 35

Statistical preliminaries

F.S1 — Probability and expectation, finite-first

Memory jog for: U4 (sampling and streaming — its every proof is an expectation computation), U3 (randomized load balancing), U13 (clustering and hashing). If $\mathbb{E}[X+Y]=\mathbb{E}[X]+\mathbb{E}[Y]$ "even when X and Y are dependent" is obvious to you, skip this chapter.

1. The finite probability space

A **finite probability space** is a finite set Ω of outcomes together with weights $p(\omega) \geq 0$ summing to one. An **event** is a subset $A \subseteq \Omega$, with $\Pr(A) = \sum_{\omega \in A} p(\omega)$. Everything this book needs from probability in Parts 0–II happens on finite spaces — the outcomes are hash seeds, random pivots, sampled indices — so no measure theory is required to make the units' proofs rigorous. (When densities on $\mathbb{R}^d$ enter, in Part IV, the honest general framework is measure-theoretic; F.S3 says exactly how much of it we import, with Billingsley's *Probability and Measure* (3rd ed.) as the reference text.)

Two events are **independent** when $\Pr(A \cap B) = \Pr(A)\Pr(B)$; the **conditional probability** given B with $\Pr(B) > 0$ is $\Pr(A \mid B) = \Pr(A \cap B)/\Pr(B)$. The **union bound** $\Pr(\bigcup_i A_i) \leq \sum_i \Pr(A_i)$ needs no independence at all — it is the workhorse of every "with high probability, none of the n bad events happen" argument in U2, U3, and U13.

2. Random variables and the one theorem that carries U4

A **random variable** is a function $X : \Omega \to \mathbb{R}$; its **expectation** is the weighted average $\mathbb{E}[X] = \sum_\omega p(\omega)\, X(\omega)$.

Theorem (linearity of expectation). For any random variables $X_1, \ldots, X_n$ on one space and constants $c_1, \ldots, c_n$,

$$\mathbb{E}\Big[\sum_{i=1}^{n} c_i X_i\Big] \;=\; \sum_{i=1}^{n} c_i\, \mathbb{E}[X_i],$$

with no independence assumption whatsoever.

Proof. Both sides are the same finite double sum $\sum_\omega p(\omega) \sum_i c_i X_i(\omega)$, reordered. □

The proof is one line; the consequences run the length of the book. The standard move (U4 uses it constantly) pairs linearity with **indicator variables**: for an event A, let $\mathbf{1}_A(\omega) = 1$ if $\omega \in A$ and 0 otherwise; then $\mathbb{E}[\mathbf{1}_A] = \Pr(A)$, and a count of things is a sum of indicators. The expected number of hash collisions, of sampled elements, of reservoir swaps — each is computed by decomposing into indicators and summing probabilities, dependence be damned.

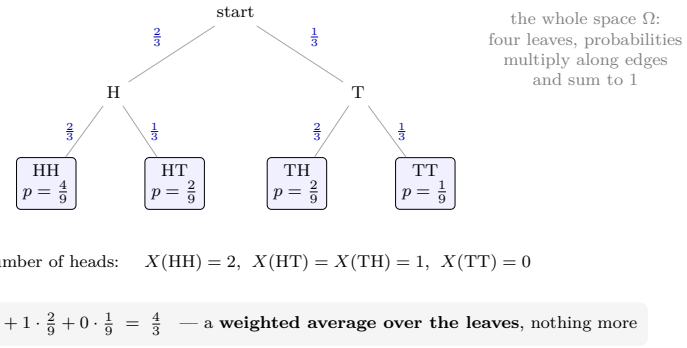


Figure 35.1: A finite probability space *is* a tree: outcomes are leaves, probabilities multiply along root-to-leaf paths and sum to one, an event is a set of leaves, and the expectation of a random variable is the leaf-weighted average. Every probabilistic argument in this book — splitter balance (U2), sampling (U4), concentration (F.S2) — starts from a picture like this one, usually with more leaves.

3. Variance — where independence starts to matter

$\mathrm{Var}(X) = \mathbb{E}[(X - \mathbb{E}X)^2] = \mathbb{E}[X^2] - (\mathbb{E}X)^2 \geq 0$. Variance is **not** linear in general; what survives is:

Fact. If $X_1, \ldots, X_n$ are **pairwise independent**, then $\mathrm{Var}(\sum_i X_i) = \sum_i \mathrm{Var}(X_i)$.

Pairwise suffices — full mutual independence is not needed — which is why the pairwise-independent hash families of U4 and U13 are enough for every second-moment argument they run. This fact + Chebyshev (F.S2) is the entire engine behind sketch-accuracy guarantees.

4. The two habits

1. **Counting? Indicators + linearity.** No independence check, no case analysis.
2. **Deviation? First try the second moment.** Compute the variance (pairwise independence usually available by construction), then Chebyshev — and only reach for Chernoff/Hoeffding (F.S2) when you need exponentially small failure probability.

5. Three ways a random sequence can converge (a definitions box)

Later units say "converges almost surely", "in probability", "in distribution" — three different promises, ordered from strongest to weakest. For random variables $X_1, X_2, \ldots$ and a target X:

> **Almost surely (a.s.):** $\Pr\big(X_n \to X\big) = 1$. *The promise:* run the experiment once; with probability one, the actual sequence of numbers you observe converges. (The strong law of large numbers is this kind: your running average *will* settle.)
>
> **In probability:** for every $\varepsilon > 0$, $\Pr\big(|X_n - X| > \varepsilon\big) \to 0$. *The promise:* at any late time, a big error is unlikely — but no promise about the whole trajectory of a single run. (F.S2's concentration bounds prove exactly this kind, with rates.)
>
> **In distribution:** $\Pr(X_n \leq x) \to \Pr(X \leq x)$ at continuity points. *The promise is about histograms, not runs:* the *shape* of X_n's distribution approaches X's. The limit can be random (the CLT's limit is a Gaussian, not a number).

The one-line hierarchy: a.s. $\Rightarrow$ in probability $\Rightarrow$ in distribution — and neither arrow reverses in general. (To a constant, in-probability and in-distribution coincide.)

Why the course cares: U9's SGD guarantees are typically **in expectation / in probability** (a rate on Pr(error $> \varepsilon$), F.S2-style) — not a promise about your one training run's every step; the strong-law intuition ("it will settle") is the **a.s.** statement and needs more. U14's density-estimation consistency is convergence of a random *function*, stated via in-probability of its L_1 error. When a unit's theorem names its mode, it is telling you precisely which promise you bought.

Deeper dive: wiki: Convergence of random variables → Wasserman, *All of Statistics*, ch. 5 → Kolmogorov (1933).

6. One exercise

E.FS1.1 In a stream of n distinct items, each item is independently kept with probability q. Let K be the number kept. Compute $\mathbb{E}[K]$ and $\mathrm{Var}(K)$, and show $\Pr(K = 0) = (1-q)^n \le e^{-qn}$. (This three-line computation is the skeleton of U4's sampling guarantees.)

F.S2 — The concentration ladder: Markov → Chebyshev → Hoeffding

Memory jog for: U2 (why random splitters balance sample sort), U4 (sketch accuracy), U13 (LSH amplification), U15/U20 (rates). The ladder has three rungs; each buys a stronger tail from a stronger hypothesis. If you can state all three from memory with their hypotheses, skip ahead — but the §4 recipe is worth ten seconds even then.

1. Rung one: Markov — nonnegativity alone

Theorem (Markov's inequality). If $X \ge 0$, then for every $t > 0$, $\Pr(X \ge t) \le \mathbb{E}[X]/t$.

Proof. $\mathbb{E}[X] \ge \mathbb{E}[X\,\mathbf{1}_{X \ge t}] \ge t\,\Pr(X \ge t)$. □

Weak, universal, and the parent of everything below: every fancier tail bound is Markov applied to a cleverly transformed variable.

2. Rung two: Chebyshev — a second moment

Theorem (Chebyshev's inequality). For any X with finite variance and $t > 0$, $\Pr(|X - \mathbb{E}X| \ge t) \le \mathrm{Var}(X)/t^2$.

Proof. Markov applied to the nonnegative variable $(X - \mathbb{E}X)^2$ at threshold t^2. □

Paired with pairwise-independent variance additivity (F.S1 §3), this gives the $1/t^2$-tail that powers every second-moment sketch analysis in U4: polynomial, not exponential, but available under the weakest independence assumption our hash families guarantee.

3. Rung three: Hoeffding — independence and boundedness

Stated exactly in the form U2 and U4 invoke it:

Theorem (Hoeffding's inequality). Let $X_1, \dots, X_n$ be independent with $a_i \le X_i \le b_i$, and $S_n = \sum_{i=1}^n X_i$. Then for every $t > 0$,

$$\Pr\big(|S_n - \mathbb{E}[S_n]| \ge t\big) \;\le\; 2\exp\Big(\frac{-2t^2}{\sum_{i=1}^n (b_i - a_i)^2}\Big).$$

In the iid mean form ($X_i \in [0,1]$, $\bar{X} = S_n/n$): $\Pr(|\bar{X} - \mathbb{E}\bar{X}| \ge \varepsilon) \le 2e^{-2n\varepsilon^2}$.

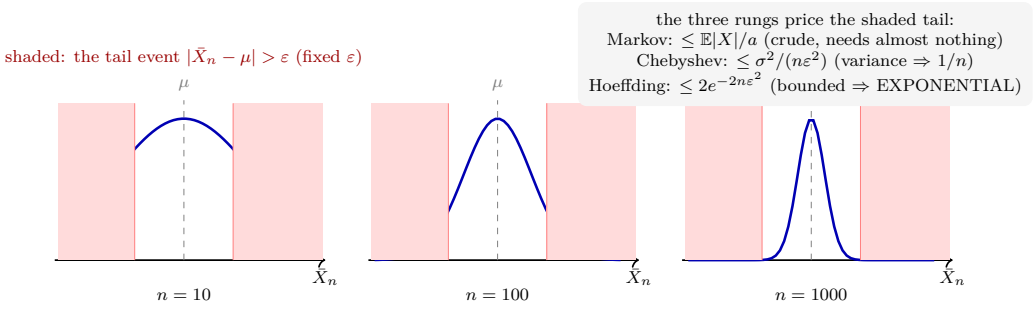


Figure 35.2: Concentration in one look: as n grows, the distribution of the sample mean $\bar{X}_n$ narrows around the truth μ, and the probability of the fixed-width tail (shaded) collapses. The chapter's three rungs are three prices for that shaded area — from Markov's crude bound to Hoeffding's exponential one. This collapsing tail is why sampled splitters balance (U2), why sketches work (U4), and why a lab's measured average means anything at all.

Proof idea [STATED; proved in standard texts — see e.g. Billingsley, *Probability and Measure* (3rd ed.), for the underlying moment-generating-function method]: apply Markov to $e^{\lambda(S_n - \mathbb{E}S_n)}$, bound each factor's mgf by Hoeffding's lemma ($\mathbb{E}[e^{\lambda(X_i - \mathbb{E}X_i)}] \leq e^{\lambda^2(b_i-a_i)^2/8}$, the one genuinely technical step), multiply across independence, optimize λ. The same template with different mgf bounds yields the whole Chernoff family.

Figure 35.2 shows the shaded tail that the three rungs price, collapsing as n grows.

4. The recipe (how the units actually use rung three)

To estimate an unknown mean to accuracy ε with confidence $1 - \delta$: solve $2e^{-2n\varepsilon^2} \leq \delta$, i.e.

$$n \;\geq\; \frac{\ln(2/\delta)}{2\varepsilon^2}.$$

Read it as a price list: **accuracy is expensive** ($1/\varepsilon^2$), **confidence is cheap** ($\log(1/\delta)$). This asymmetry is why U4's sketches happily union-bound over millions of counters (confidence), while every halving of error quadruples the sample bill (accuracy) — and why U2's sample sort can afford splitter samples of size only $O(p \log n)$.

5. Two idioms worth keeping loaded

- **Coupon collector:** collecting all n coupons takes $nH_n \approx n \ln n$ draws in expectation — the "$\log n$ oversampling" that appears whenever every bucket/partition must be hit (U2 splitters, U3 balls-in-bins).
- **Birthday:** among m uniform draws from $[n]$, collisions appear around $m \approx \sqrt{n}$ — the reason hash ranges are sized quadratically in the number of keys when collisions must be rare (U4, U13).

6. One exercise

E.FS2.1 You sample s elements uniformly with replacement from an array of length n and use their empirical median as a pivot. The pivot's rank falls below $(\frac{1}{2} - \varepsilon)n$ only if at least half the samples have rank $\leq (\frac{1}{2} - \varepsilon)n$. Using Hoeffding on the indicator variables $\mathbf{1}\{\text{rank}(X_i) \leq (\frac{1}{2} - \varepsilon)n\}$, the mirror-image indicators $\mathbf{1}\{\text{rank}(X_i) \geq (\frac{1}{2} + \varepsilon)n\}$ for the other side, and a union bound over the two sides, show that with $s = c\ln(1/\delta)/\varepsilon^2$ samples, the pivot's rank is within εn of $n/2$ with probability $\geq 1 - \delta$. (This is U2's splitter-balance lemma with one splitter.)

F.S3 — Densities, total variation, and what "estimating" means

Memory jog for: U14 (tree densities), U15 (minimum-distance estimation — this chapter is its vocabulary), U20 (merging estimators), U16 (what a guarantee is a guarantee of). If you can prove that total variation equals half the L_1 distance, skip to §4.

1. Densities, finite-first

On a finite space Ω, a **density** (probability mass function) is just the weight function $p(\omega)$ of F.S1. On $\mathbb{R}^d$, a density is a nonnegative f integrating to one, and $\Pr(A) = \int_A f$. The finite intuition transfers wholesale: sums become integrals. (The honest general statement — densities with respect to a dominating measure, existence via Radon–Nikodym — is measure theory; the canonical reference for this book is Billingsley, *Probability and Measure*, 3rd ed., which is also where the factorization and Scheffé results below live in full generality. We import only what Part IV consumes.)

2. Total variation — the distance that means something operationally

Definition. The **total variation distance** between densities p, q on Ω:

$$\mathrm{TV}(p,q) \;=\; \sup_{A\subseteq\Omega} \big|\Pr_p(A) - \Pr_q(A)\big|.$$

It is the worst-case disagreement over **every question you could ask** — the reason TV (not, say, mean-squared error on a histogram) is the honest currency for density estimation in U15.

Theorem (TV = half L_1). $\mathrm{TV}(p,q) = \frac{1}{2}\sum_\omega |p(\omega) - q(\omega)| = \frac{1}{2}\|p-q\|_1$.

Proof (finite case). Take $A^* = \{\omega : p(\omega) > q(\omega)\}$. For any A, $\Pr_p(A) - \Pr_q(A) = \sum_A (p-q) \le \sum_{A\cap A^*}(p-q) \le \sum_{A^*}(p-q)$, so the sup is attained at A^*. Since $\sum_\omega (p-q) = 0$, the positive part $\sum_{A^*}(p-q)$ equals the negative part in magnitude, and each equals $\frac{1}{2}\|p-q\|_1$. □

U15's lesson: **keep the factor of two straight.** Statements true in TV can be false as stated in L_1 and vice versa; the certified statements in the CMDE ladder (provedog: `ScheffeFinite`, `L1MDERate`) fix the convention exactly, and the book quotes them in the checker's own normalization.

3. Scheffé's identity — how you *test* between two densities

Theorem (Scheffé). For densities p, q and the set $A^* = \{p > q\}$: a sample from an unknown $r \in \{p, q\}$ can be tested by asking whether A^* occurred (answer p if it did, q if not). The sum of the test's two error probabilities is $\Pr_p(\Omega \setminus A^*) + \Pr_q(A^*) = 1 - \mathrm{TV}(p,q)$, and no test does better: answering p on any set A gives $1 - (\Pr_p(A) - \Pr_q(A)) \ge 1 - \mathrm{TV}(p,q)$. So A^* is optimal.

Figure 35.3 draws the identity: TV is half the shaded area between p and q, and A^* is the region where p sits above q.

The operational reading is what U15 runs with: **two candidate densities can be distinguished from data exactly as well as their TV separation allows, by a single set membership question.** The Scheffé tournament (U15 §3.3, run distributed in U15 §5.3; its bracket figure is in U15 §1) is nothing but this pairwise test run over a tournament of candidates — certified finite-first in `ScheffeFinite`.

4. Estimators, risk, and the minimax yardstick

An **estimator** is a map $\hat{f}$ from data to answers (here: from n samples to a density). Its **risk** at truth f is $\mathbb{E}_f[\,\mathrm{TV}(\hat{f}, f)\,]$ (expectation over the sample). Because the truth is unknown, U15/U20

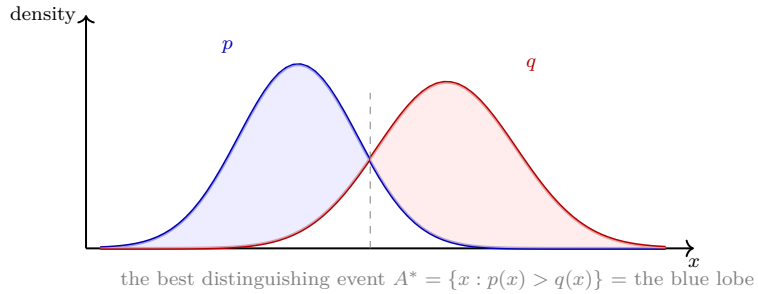


Figure 35.3: Total variation, seen. Two densities disagree exactly on the shaded area between them; TV — the largest probability gap *any* event can exhibit — equals *half* that area (Scheffé's identity), and the event achieving it is simply "where p sits above q". Half this picture is U14/U15's estimation theory; the other half is a monitoring dashboard: distribution drift *is* this shaded area, growing.

grade estimators by **worst-case (minimax) risk over a class $\mathcal{F}$**:

$$\inf_{\hat{f}} \sup_{f \in \mathcal{F}} \mathbb{E}_f[\,\mathrm{TV}(\hat{f}, f)\,].$$

Three habits attach:

1. **An estimator without a class is a number without units.** Every U15 guarantee names its $\mathcal{F}$ (e.g., tree-structured densities of bounded depth).
2. **Upper bounds are algorithms, lower bounds are information.** U15 proves upper bounds by exhibiting the MDE; the matching lower bounds (why no estimator can do better) are information-theoretic and live in the U15 sources.
3. **Rates are the deliverable.** "Consistent" is the participation trophy; $O(\sqrt{k \log n / n})$-type rates with explicit constants are what the certified ladder (`L1MDERate`) actually pins down.

5. Quantiles and the empirical distribution

§4's estimators map data to answers; the most basic such map is the one that turns a sample *into a distribution*. It is also the plug-in behind §4's risk language.

Definition. For a random variable with CDF $F(x) = \Pr(X \le x)$, the **u-quantile** $(0 < u < 1)$ is $Q(u) = \inf\{x : F(x) \ge u\}$ — the smallest value with at least a u-fraction of the probability at or below it. $Q(1/2)$ is the median; $Q(1/4), Q(3/4)$ the quartiles. Q is the (generalised) inverse of F: the CDF answers "how much lies below x?", the quantile answers the reverse question.

Definition. Given data $x_1, \ldots, x_n$, the **empirical CDF** is

$$\widehat{F}_n(x) = \frac{1}{n} \#\{i : x_i \le x\}$$

— a staircase that jumps $1/n$ at each data point. It is the CDF of the *empirical distribution* (uniform on your sample), and **sample quantiles** are its quantiles: the sample median of $n = 5$ points is the 3rd order statistic, exactly $\widehat{Q}(1/2)$.

- *Worked example:* data $\{3, 1, 4, 1, 5\}$. Sorted: $1, 1, 3, 4, 5$. The staircase jumps by $1/5$ at 1 (twice, so by $2/5$), 3, 4, 5; the sample median is 3; the sample 0.8-quantile is 4.

Why you may trust the staircase (the one theorem to know). Glivenko–Cantelli: as $n \to \infty$, $\sup_x |\widehat{F}_n(x) - F(x)| \to 0$ — the staircase converges to the truth *uniformly*, and F.S2's

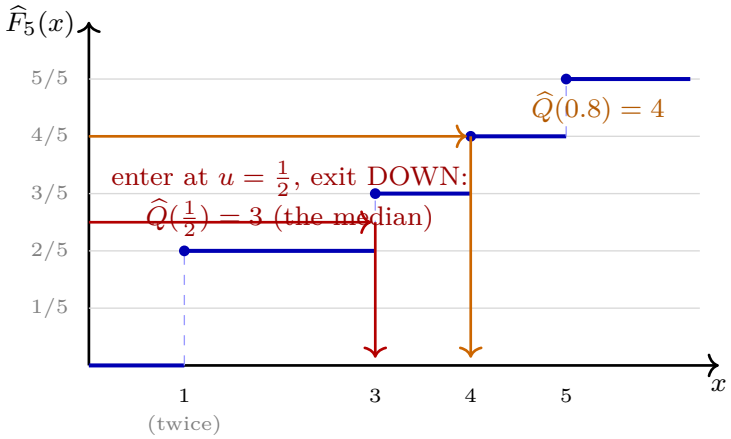


Figure 35.4: The empirical CDF of the data $\{3, 1, 4, 1, 5\}$: a staircase jumping 1/5 per point (2/5 at the repeated 1). Quantiles are read *sideways*: enter on the vertical axis at level u, exit downward at $\widehat{Q}(u)$. U2's splitters, U4's sketches, and U20's conformal thresholds all read a staircase exactly like this one — and Glivenko–Cantelli says the staircase converges uniformly to the truth.

concentration rungs price the error at finite n (the DKW inequality gives the Hoeffding-style rate $2e^{-2n\varepsilon^2}$). The empirical distribution is the statistician's plug-in: compute on the staircase what you wish you could compute on the truth — exactly the §4 estimator idea in its purest form.

Why the course cares — five units stand on this section: U2's sampled **splitters** are sample quantiles (the balance guarantee is exactly "sample quantiles approximate true quantiles", priced by F.S2); U4's streaming **quantile sketches** approximate $\widehat{Q}$ in small memory; U20's **conformal** threshold is a sample quantile of scores (its coverage guarantee is the quantile's defining property); U15's estimators and U18's evaluation percentiles read the same staircase.

Deeper dive: wiki: Empirical distribution function · wiki: Quantile → Wasserman, *All of Statistics*, ch. 7 → Glivenko (1933), Cantelli (1933); Dvoretzky–Kiefer–Wolfowitz (1956).

6. One exercise

E.FS3.1 Let p be uniform on $\{1, \ldots, 2m\}$ and q uniform on $\{1, \ldots, m\}$. Compute $\mathrm{TV}(p, q)$ two ways: from the definition (find A^*) and from $\frac{1}{2}\|p - q\|_1$, and verify they agree. Then state what a Scheffé test between p and q asks of one sample, and its error. (Ten minutes; after it, U15 §1–2 read like restatements.)

F.S4 — Information: entropy, KL, and perplexity

*Foundation chapter (the Statistics pillar). Prereq: F.S1 (finite probability, expectation). A **memory-jog with full formal definitions** — the information vocabulary six units of this book use and no chapter yet taught. Same contract as F.C1b: every definition with a worked example as gentle as arithmetic, a picture, a line on why the course cares, and a ladder.*

One idea underneath everything here: *how surprised should you be, on average?* Answer that precisely and entropy, cross-entropy, KL, and perplexity all fall out as four readings of one quantity.

1. Surprise, and its average: entropy

Definition. For an outcome of probability p, its **surprise** (self-information) is $-\log_2 p$ bits. Rare events are big news ($p \to 0$ gives surprise $\to \infty$); certain events are no news ($p = 1$ gives 0). The **entropy** of a finite distribution $p = (p_1, \ldots, p_k)$ is the *expected surprise*:

$$H(p) \;=\; -\sum_i p_i \log_2 p_i \quad \text{bits.}$$

- *Worked example (the same biased coin as F.S1's tree, $p = (2/3, 1/3)$):* $H = \frac{2}{3}\log_2 \frac{3}{2} + \frac{1}{3}\log_2 3 \approx \frac{2}{3}(0.585) + \frac{1}{3}(1.585) \approx 0.918$ bits — a little less than a fair coin's 1 bit, because this coin is a little predictable.
- Two anchors: the **uniform** distribution on k outcomes maximises entropy at $\log_2 k$ (nothing is predictable); a **point mass** has entropy 0 (everything is).
- The operational reading (Shannon): $H(p)$ is the average number of bits an *optimal* code needs per outcome — surprise = ideal code length (exactly so in the long-run limit, via Kraft/arithmetic coding; per single symbol the honest statement is $\lceil -\log_2 p \rceil$). `MATHS->CS:` compression is applied entropy; a file that compresses well was predictable.
- *Why the course cares:* entropy is the honest unit of "how much is going on" — in a data stream (U4), a cluster assignment (U13), a token distribution (U0L), or a substrate's disorder (U17's boundary-maintenance reading).

Deeper dive: wiki: Entropy (information theory) → Cover–Thomas, *Elements of Information Theory*, ch. 2 → Shannon (1948).

2. Cross-entropy: the cost of believing the wrong model

Definition. Let p be the *true* distribution and q your *model* of it. The **cross-entropy**

$$H(p, q) \;=\; -\sum_i p_i \log_2 q_i$$

is the average surprise you experience when reality draws from p but you priced everything by q — equivalently, the average code length when you built the code for q and the data came from p.

- *Worked example (same coin):* truth $p = (2/3, 1/3)$, model $q = (1/2, 1/2)$: $H(p, q) = -\frac{2}{3}\log_2 \frac{1}{2} - \frac{1}{3}\log_2 \frac{1}{2} = 1$ bit — more than the 0.918 bits the truth requires. Wrong beliefs cost bits.
- $H(p, q) \geq H(p)$ always, with equality iff $q = p$: no model beats the truth.
- *Why the course cares — this IS the training loss.* A language model is trained by minimising the cross-entropy of its next-token distribution against the data (u9's M-estimation with $-\log q$ as the loss; u18's objective). "The loss went down" means, literally, "the model's average surprise at real text went down."

Deeper dive: wiki: Cross-entropy → Cover–Thomas ch. 5 → Kraft (1949), Shannon (1948).

3. KL divergence: the gap itself

Definition. The **Kullback–Leibler divergence** is the excess — cross-entropy minus entropy:

$$\mathrm{KL}(p \,\|\, q) \;=\; H(p, q) - H(p) \;=\; \sum_i p_i \log_2 \frac{p_i}{q_i} \;\geq\; 0,$$

with equality iff $p = q$ (Gibbs' inequality). It is the *price of the wrong model*, in bits per observation.

- *Worked example (same numbers):* $\mathrm{KL}(p \,\|\, q) = 1 - 0.918 = 0.082$ bits — the fair-coin model overpays 0.082 bits per flip for this biased coin.
- **Honesty about its shape:** KL is *not* a distance — it is asymmetric ($\mathrm{KL}(p\|q) \neq \mathrm{KL}(q\|p)$ in general) and violates the triangle inequality. It is a *divergence*: a directed cost, read "q judged from p's world".
- **The bridge to F.S3 (Pinsker's inequality):** $\mathrm{TV}(p, q) \leq \sqrt{\frac{\ln 2}{2}\,\mathrm{KL}(p \,\|\, q)}$ — small KL forces small total variation, so the information world of this chapter and the event-probability world of F.S3/U15 bound each other. (The reverse fails: TV small does not force KL small — one zero in q where p has mass makes KL infinite. This asymmetry is *why* language models never assign a token probability exactly zero.)
- *Why the course cares:* KL is the book's currency of model mismatch — in training objectives (u9/u18), in drift and mismatch detection alongside TV (u15), and in U17's sufficiency story (a lossless summary is one whose KL cost is zero).

Deeper dive: wiki: KL divergence → Cover–Thomas ch. 2, 11 → Kullback–Leibler (1951); Pinsker (1964).

4. Perplexity: cross-entropy, exponentiated

Definition. The **perplexity** of a model q on data from p is

$$\mathrm{PPL} \;=\; 2^{H(p,q)}$$

(computed in practice on a held-out corpus, per token). Exponentiating turns average bits back into a *count*: perplexity is the **effective branching factor** — "the model is as confused as if it were choosing uniformly among PPL tokens at every step."

- *Worked example (same coin):* the fair-coin model has $\mathrm{PPL} = 2^1 = 2$ — as confused as a two-way uniform guess; the true model achieves $2^{0.918} \approx 1.89$ — slightly *less* than two-way confused, because the coin is slightly predictable. A model that has learned nothing about a vocabulary of $|V|$ tokens has perplexity $|V|$; training is the art of pushing it down.
- *Base-independence (preempting a real confusion):* perplexity does not care which logarithm you used — $\mathrm{PPL} = b^{H_b(p,q)}$ for any base b. Practitioners report e^{loss} with the natural-log training loss; that IS this chapter's 2^{bits}, the same number.
- *Why the course cares:* perplexity is the headline number of u0L and u18 — now you know exactly what it is (exponentiated cross-entropy), why lower is better (fewer effective choices), and why it can never beat $2^{H(p)}$ (no model beats the truth, §2).

Deeper dive: wiki: Perplexity → Jurafsky–Martin, *Speech and Language Processing*, ch. 3 → Jelinek et al. (1977).

5. The map of this chapter

$$\underbrace{-\log_2 p}_{\text{surprise}} \xrightarrow{\mathbb{E}_p} \underbrace{H(p)}_{\text{entropy}} \xrightarrow{\text{wrong model}} \underbrace{H(p,q)}_{\text{cross-entropy}} \xrightarrow{-H(p)} \underbrace{\mathrm{KL}(p\|q)}_{\text{the gap}}$$

$$\underbrace{H(p,q)}_{\text{cross-entropy}} \xrightarrow{2^{(\cdot)}} \underbrace{\mathrm{PPL}}_{\text{branching factor}}$$

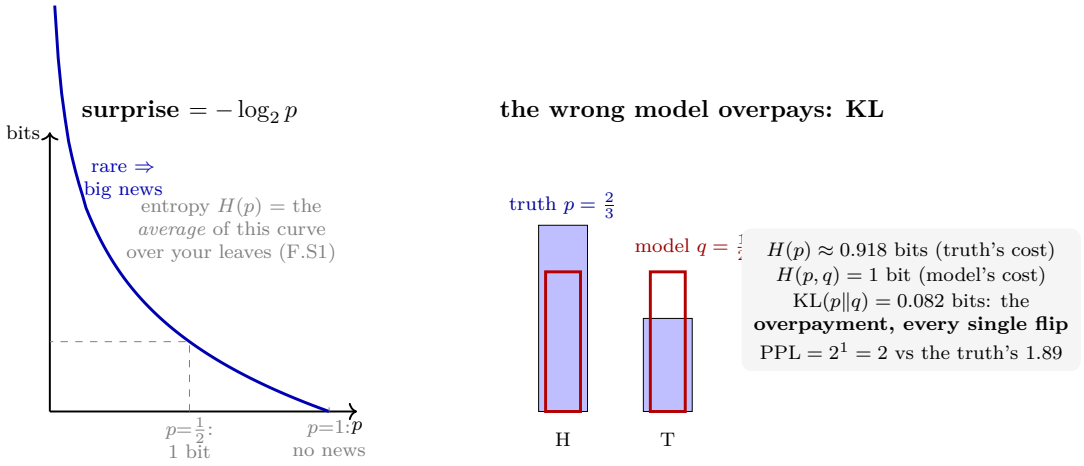


Figure 35.5: Information in two looks, on F.S1's own biased coin. Left: surprise is $-\log_2 p$ — certain events are no news, rare ones are big news; entropy is this curve averaged over your probability tree's leaves. Right: modelling the $\left(\frac{2}{3}, \frac{1}{3}\right)$ coin as fair costs 1 bit per flip where the truth costs 0.918; the KL divergence *is* that 0.082-bit overpayment, and perplexity is the same story exponentiated back into an effective number of choices. Training a language model = shrinking the right panel's gap on real text.

Note that perplexity exponentiates the cross-entropy, not the gap: 2^{KL} for the fair-coin model is $2^{0.082} \approx 1.06$, while its perplexity is $2^1 = 2$ (§4). One quantity — average surprise — read four ways. Everything is finite sums over F.S1's leaves; nothing here needed more than the probability tree you already own.

Where this is used

u0L (tokens as code-lengths; the perplexity headline), u9 (the loss function IS cross-entropy), u13 (entropy of cluster assignments), u15 (KL beside TV; Pinsker is the bridge; why zero-probability tokens are forbidden), u17 (boundary maintenance as work against entropy; sufficiency as zero-KL summarisation), u18 (training/eval objectives, perplexity).

Chapter 36

Computational preliminaries

F.C1 — Machine models: RAM → PRAM → DPRAM (a recap)

*Foundation chapter (Pillar 3, Computational Foundations). This is a **memory-jog**, not a first course: it states, in one place, the machine models and cost vocabulary the units assume. If a unit's Entry Kit says "needs F.C1", this is what it means. Unit 0 (Models & cost) sharpens all of it; here we only need the shape.*

The one organising idea. The *same* problem is re-solved as the machine gains a resource, and what changes is the *cost you must account for*. Three models, three prices.

1. RAM — the sequential baseline

A **Random-Access Machine**: one processor, a flat memory of words, unit-cost basic operations (arithmetic, comparison, load/store at an arbitrary address). The cost of an algorithm is $T(n)$, the number of operations on an input of size n. This is the number your $O(\cdot)$ bounds are secretly about, and it is the **total work** every parallel or distributed version must ultimately match.

The one rigorous statement units use: an algorithm is $O(f(n))$ time on the RAM if its operation count is $\leq c\, f(n)$ for some constant c and all large n (see F.M1 for $O/\Theta/\Omega$ and recurrences).

2. PRAM — shared-memory parallelism: work, depth, Brent

A **Parallel RAM**: p processors share one memory. Two numbers price a parallel algorithm:

- **Work** W — the total number of operations across all processors (the RAM cost, summed).
- **Depth** (or **span**) D — the length of the longest chain of operations that must happen in order: the time on ∞ processors. Parallel-algorithm design is *shaping the dependency DAG to be shallow without adding work.*

Brent's principle (the bridge). On p processors,

$$T_p \;\leq\; \frac{W}{p} + D.$$

Low depth **and** low work give good speed-up; you stay on the linear-speed-up regime only while $p \lesssim W/D$ (below the *depth wall*). **Work-optimality** means $W = O(T_{\text{seq}})$: the parallel version wastes no more than a constant factor of total work.

The two universal primitives — **reduction** (an associative fold) and **scan / all-prefix-sums** — both run in depth $O(\log n)$ and work $O(n)$; almost everything parallel is built from them

(see U1). Memory-access variants (EREW / CREW / CRCW) change constants and some lower bounds but not this picture.

Cert pointer: the Brent bound is machine-checked — `BMLSPar.lean` (`bmls_parallel_bound`), sorry-free. Reading a cert like it is F.C3.

3. DPRAM / MPC — distributed data: rounds and communication

When the data does not fit on one machine it is **partitioned** across nodes that talk only by messages. Now the dominant cost is **communication**, not work or depth. Account for:

- **Rounds** R — synchronous phases of (local compute, then a global exchange + barrier).
- **Local memory** s — how much each node may hold (sublinear in the total).
- **Bytes moved** — the total communicated volume; a badly-sharded all-to-all shuffle is the distributed analogue of thrashing.

The recurring shape is **map** → **shuffle** → **reduce** (local → all-to-all → local). Bridging cost models — **BSP** (per-superstep: local work $+g \cdot h + L$) and **LogP** — just add a *communication term* to the (W, D) accounting of §2, so the same primitives recur: reduce becomes tree-aggregate, scan becomes a distributed scan, sort becomes sample-then-shuffle. (F.E4 derives the network's α-β cost from first principles; F.E1/F.E2 are the machine + container floor the labs run on.)

4. The invariant (memorise this box)

> **RAM asks:** *how many operations?* (work $T(n)$). **PRAM adds:** *how deep?* — hence, via Brent, *how much speed-up?* (work W, depth D). **DPRAM adds:** *how many rounds, and how much communication?* (rounds R, local memory s, bytes).

Every later unit re-instantiates these three lenses for one algorithm. That is the whole method: pick a problem, price it on all three machines, and read off where scaling stops.

Where this is used

U0 sharpens this into the examined cost-lens; U1–U5 (scan, sorting, scheduling, sampling, map-reduce) live on it directly; every unit with a `depends F.C1` edge in the atlas is standing here.

F.C1b — Problems, classes, and reductions

*Foundation chapter (the Computation pillar). Prereq: F.C1 (machine models + work/depth), F.M1 (asymptotics). A **memory-jog with full formal definitions** — because a reader who has never needed "P-complete" deserves the same civilised on-ramp as one who has never needed* $1 < 2$.

The ethos of this chapter. A definition is a definition. "$1 < 2$" and "lexicographic depth-first search is P-complete" are the same kind of object — a precise statement built from precise parts — differing only in how *familiar* the parts happen to be to a given reader at a given moment. Familiarity is biography, not mathematics. This chapter supplies every part, in order, each with: the formal definition, one worked example treated exactly as gently as arithmetic, one line on why this course needs it, and a ladder for deeper dives. Nothing here is harder than what you have already done in the labs; it is only newer.

1. What a "problem" is, formally

A **decision problem** is a set L of finite strings — the *yes-instances* — over some alphabet: to "solve" L is to decide, given a string x, whether $x \in L$.

- *Worked example (as gentle as it looks):* REACHABILITY $= \{\langle G, s, t\rangle$: directed graph G has a path $s \to t\}$. The string $\langle G, s, t\rangle$ is any reasonable encoding of the graph and the two vertices (adjacency list as text, say); the problem is the *set* of encodings where a path exists. That is the whole definition.
- Optimisation problems ("find the cheapest schedule") have decision versions ("is there a schedule of cost $\leq k$?") — the classes below are defined for decision problems, and the decision version captures the hardness of the original (ask for k by binary search).
- *Why the course cares:* every "can the fleet finish in time T?", "is this graph strongly connected?", "does this training run fit in memory?" is a decision problem the moment you write it down carefully.

Deeper dive: wiki: Decision problem → Sipser, *Introduction to the Theory of Computation*, ch. 3–4 → Turing (1936).

2. P — solvable in polynomial time

Definition. $L \in$ P iff some algorithm decides "$x \in L$?" in at most $c \cdot |x|^k$ elementary steps (F.C1's RAM steps) for constants c, k and every input x. "Polynomial in the input length" — that is all.

- *Worked example:* REACHABILITY $\in$ P: breadth-first search touches each edge a constant number of times, $O(|V| + |E|)$ — a polynomial (indeed linear). You have run this exact algorithm in the U11 labs.
- The definition is **robust**: RAM steps, Turing steps, or your laptop's instructions differ only polynomially, so the class does not change with the machine (F.C1's lesson formalised).
- *Honest caveat:* "polynomial" is the mathematically robust notion of "efficient", not a promise of "fast" — n^{100} is polynomial. In this course the polynomials that appear are small, and the *distinctions between classes* are what carry the engineering meaning.
- *Why the course cares:* P is the baseline "solvable at all, at scale" — every algorithm in Parts I–IV lives here; the course's question is almost always the *finer* one: solvable with how much work, depth, communication, memory (F.C1)?

Deeper dive: wiki: P (complexity) → Arora–Barak, *Computational Complexity*, ch. 1 → Cobham (1965), Edmonds (1965).

3. NP — checkable in polynomial time

Definition (the certificate form). $L \in$ NP iff there is a polynomial-time **verifier** V and a polynomial p such that: $x \in L$ exactly when there EXISTS a *certificate* (witness) w, of length $\leq p(|x|)$, with $V(x, w)$ accepting.

Read it slowly, because it is beautiful: NP is not about solving — it is about **checking**. A yes-answer can be *certified* by a short witness that a fast verifier confirms.

- *Worked example:* SUBSET-SUM $= \{\langle a_1..a_n, T\rangle$: some subset sums to $T\}$. Finding the subset may require search; but *given* the subset (the certificate), checking it is at most $n - 1$ additions and one comparison. So SUBSET-SUM $\in$ NP.
- **The course has been living this definition since U0P.** A machine-checked proof is exactly an NP-style certificate: the proof object is the witness, the proof-checker's kernel is the polynomial-time verifier, and "the checker accepted it" is $V(x, w) = 1$ (F.C3). The

σ-receipts of your labs are certificates too: short witnesses that a grader's verifier confirms. *Verification cheaper than discovery* is arguably this book's deepest recurring pattern — NP is its complexity-theoretic name.
- Clearly P $\subseteq$ NP (solve, ignore the witness). Whether the containment is strict — **P vs NP** — is the most famous open problem in mathematics. We do not know. Everything below is stated with that honesty.

Deeper dive: wiki: NP (complexity) → Sipser ch. 7 → Cook (1971), Levin (1973).

4. Reductions — "at least as hard as"

Definition. $A \leq_p B$ (A *polynomial-time many-one reduces* to B) iff there is a polynomial-time computable f with: $x \in A \iff f(x) \in B$, for all x.

A reduction is a *translator*: any solver for B becomes a solver for A (translate, then ask). So $A \leq_p B$ reads "B is at least as hard as A".

- *Worked example (course-flavoured):* "does the directed graph G have a closed walk through both u and v?" (equivalently: do u and v lie in one strongly connected component?) reduces to REACHABILITY on a modified graph — take two copies of G, add one edge from the first copy of v to the second copy of v, and ask whether the first copy of u reaches the second copy of u. That graph surgery is f, and it runs in polynomial time. The U11 exercises do several of these without naming the concept; here it is named.
- Reductions compose ($\leq_p$ is transitive) and P, NP are *downward closed* under them: $A \leq_p B$ and $B \in$ P give $A \in$ P. `MATHS->CS:` $\leq_p$ is a **preorder on problems** — hardness has an order structure, and the classes are its down-sets. (The order-theoretic reading returns with force in the capability lattices of U17.)
- *Why the course cares:* a reduction is how you inherit an impossibility — and how you recognise that your "new" scheduling question is an old friend in costume.

Deeper dive: wiki: Polynomial-time reduction → Arora–Barak ch. 2 → Karp (1972).

5. NP-hard and NP-complete

Definition. B is **NP-hard** iff $A \leq_p B$ for *every* $A \in$ NP (B is at least as hard as everything checkable). B is **NP-complete** iff moreover $B \in$ NP (it is itself checkable — a *hardest member*, not an outsider).

- The Cook–Levin theorem: SAT (Boolean satisfiability) is NP-complete — the first, from which thousands follow by reductions (Karp's 21 started the avalanche).
- *What completeness buys:* if ANY NP-complete problem is in P, then P = NP — all of them fall together. Conversely, a proof that your problem is NP-hard is a licence to stop seeking an exact fast algorithm and reach for the course's other tools: approximation, randomisation (U2/U4), special structure (U8's DAGs), or scale-out with honest exponents.
- *Honest reading:* NP-hard $\neq$ "impossible" — it means "as hard as the hardest checkable problems, as far as anyone knows". Instances met in practice are often benign; the theory prices the worst case.
- *Why the course cares:* optimal cluster scheduling in full generality is NP-hard — which is precisely WHY U3/U17 run *greedy with a proved bound* (Graham, Brent) instead of chasing optimality: a provably-good-enough answer to a provably-hard question.

Deeper dive: wiki: NP-completeness → Garey–Johnson, *Computers and Intractability* → Cook (1971), Karp (1972).

6. NC — the fast-parallel class

Definition. $L \in$ NC iff it is decidable in **polylogarithmic depth** $O(\log^k n)$ using **polynomially many processors** — F.C1's work/depth lens promoted to a class: NC is "what a big fleet can finish in almost no wall-clock".

- *Worked examples you have already met:* prefix-scan (U1: depth $O(\log n)$), matrix multiplication (U6), sorting (U2's parallel mergesort; Cole's $O(\log n)$-depth version) — all in NC.
- NC $\subseteq$ P (simulate the fleet sequentially, polynomial work). Whether NC = P — "can everything solvable be solved *fast in parallel*?" — is open, and the standard conjecture is NO.
- *Why the course cares:* NC is the formal name of "below the depth wall" — the course's parallel algorithms are NC algorithms, stated with explicit constants.

Deeper dive: wiki: NC (complexity) → Greenlaw–Hoover–Ruzzo, *Limits to Parallel Computation* (the canonical text) → Pippenger (1979).

7. P-complete — "inherently sequential" (the class this course actually uses)

Definition. B is **P-complete** iff $B \in$ P and every $A \in$ P reduces to B under reductions weak enough not to smuggle in the hardness — standardly **NC (or log-space) reductions**, $A \leq_{\text{NC}} B$. (With $\leq_p$ the definition would collapse: any non-trivial member of P would be "complete". The reduction must be *parallel-cheap* so that hardness, not the translator, carries the load.)

What it means. A P-complete problem is a hardest problem *inside* P for parallel purposes: if any P-complete problem were in NC, then NC = P — every polynomial algorithm would parallelise. Under the standard conjecture, therefore, P-complete problems admit **no polylog-depth algorithm on polynomially many processors**: they are *inherently sequential* — solvable, efficiently even, but only by genuinely deep computations. This is the depth wall of F.C1 given a name, a definition, and hardest witnesses.

- *Canonical example:* CIRCUIT VALUE (evaluate a given Boolean circuit) — the "SAT of P".
- *The course's example:* **lexicographic depth-first search is P-complete** (Reif 1985) — the theorem behind U11 §4b.3's negative lesson: Tarjan's SCC algorithm is optimal AND inherently deep, so distributed SCC had to change primitives (reachability), not merely parallelise harder. *(The P-completeness is specifically for the lexicographic ordering; see Reif 1985.)*
- *Why the course cares, in one sentence:* P-completeness is the mathematically honest form of the sentence "no, you cannot just throw more workers at it" — and knowing WHEN that sentence is a theorem (versus an engineering excuse) is worth a great deal in practice.

Deeper dive: wiki: P-complete → Greenlaw–Hoover–Ruzzo (a whole book of P-complete problems) → Ladner (1975), Reif (1985).

8. The map, and the honest frontier

$$\text{NC} \subseteq \text{P} \subseteq \text{NP},$$

with **both** containments conjectured strict and **neither** proved so. The two great open questions bracket this course exactly: *P vs NP* asks whether checking is easier than solving (the certificate world of §3); *NC vs P* asks whether solvable means parallel-solvable (the depth world of §7). Everything this book computes lives inside that double uncertainty, priced honestly: we prove

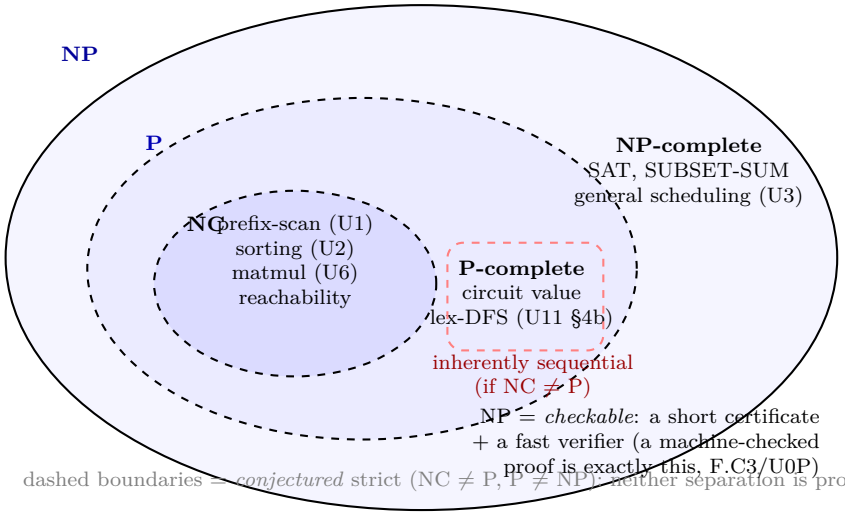


Figure 36.1: The map of F.C1b: NC ⊆ P ⊆ NP, with the course's own problems placed. The parallel algorithms of Parts I–II live in NC; P-complete problems (circuit value; lexicographic DFS, the U11 §4b lesson) are the hardest inside P for parallel purposes — *inherently sequential* under the standard conjecture; NP-complete problems bracket what greedy-with-proved-bounds (U3, U17) wisely declines to solve exactly. Dashed lines are drawn honestly: both separations are open.

membership (an algorithm, with constants) and we prove hardness (a completeness reduction), and where mathematics is silent we say so.

A reader who arrived intimidated by these names now owns them the same way they own $1 < 2$: as definitions they can state, check, and use. That was the point.

Where this is used

U11 §4b (lex-DFS P-completeness → why distributed SCC changes primitives); U17 (TBMLS runs condensation as a parallel-cheap phase precisely because DFS is not); U1/U2/U6 (their parallel algorithms are NC memberships with explicit constants); U3 (Graham's bound as the answer to an NP-hard scheduling question); F.C3/U0P (a machine-checked proof IS an NP certificate — the verifier is the kernel).

F.C2 — Discrete structures and hashing

Foundation chapter (Pillar 3, Computational Foundations). Prereq: F.M1. A ***memory-jog*** *of the discrete vocabulary the graph, sketch, and similarity-search units assume, and of the one idea — a* random-looking *hash with a* provable* guarantee — that makes streaming and LSH work.*

1. Discrete structures — the vocabulary

- **Sets, relations, functions.** A relation $R \subseteq A \times B$; a function is a relation that is single-valued. Cardinality $|A|$, power set, partitions.
- **Graphs.** $G = (V, E)$: vertices and edges (directed or not, weighted or not). The things units ask about — paths, connectivity, spanning trees, cuts, degree — are all defined here (U11, U12).
- **Trees.** A connected acyclic graph; rooted trees give parent/child/height. Balanced binary trees have height $\Theta(\log n)$ — the reason reductions and scans (U1) and the density trees (U14) are log-depth.

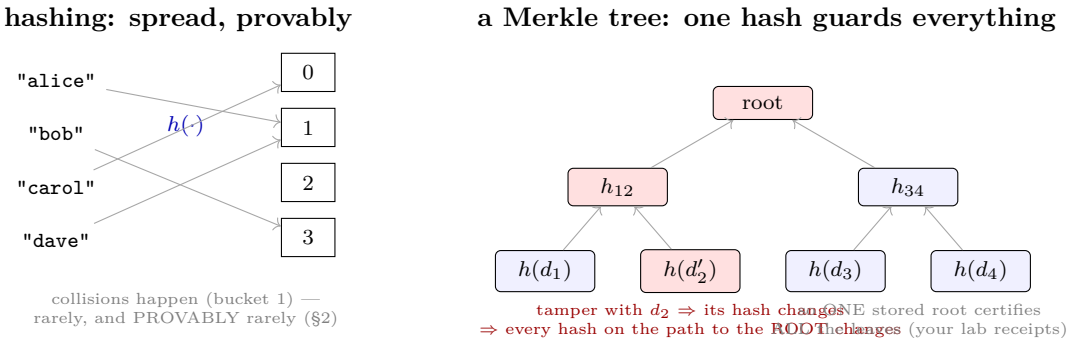


Figure 36.2: The two faces of hashing. Left: a hash function spreads keys across buckets, with collision probabilities you can *prove* (§2). Right: hashes composed into a Merkle tree — each parent hashes its children, so tampering with any leaf (red) changes every hash up to the root. One remembered root certifies an entire dataset: this is why your lab receipt-chains, git commits, and the course's σ-log are tamper-evident.

`CS->MATHS:` "a graph algorithm is $O(m+n)$" is a statement over $|E| = m$, $|V| = n$; the structure fixes what m and n mean.

2. Hashing — randomness you can prove things about

A **hash function** $h : U \to [m]$ maps a large universe into m buckets. The point is not obfuscation; it is that a *family* of hash functions, with a key drawn at random, behaves *predictably in expectation*:

- **universal family:** for $x \neq y$, $\Pr_h[h(x) = h(y)] \leq 1/m$ — collisions are rare *on average over the key*, for any fixed inputs (no adversary can force many collisions without knowing the key).
- **pairwise / k-wise independence:** any k distinct inputs land independently and uniformly — enough independence to bound a variance, but cheap to store.

That is exactly the leverage the streaming sketches use: Count-Min's error and HyperLogLog's estimate are controlled by the hash family's guarantee (U4), and locality-sensitive hashing (U13) turns "close points collide, far points do not" into a probability you can tune. `MATHS->CS:` a hash is a random variable with a *design spec*, not a black box.

Where this is used

U4 (Count-Min, HyperLogLog), U11/U12 (graph vocabulary), U13 (LSH + hash-partitioning), U14 (trees). Units with a `depends F.C2` edge stand on either the discrete vocabulary or the hashing guarantee.

F.C2b — Order, lattices, and fixed points

*Foundation chapter (the Computation pillar). Prereq: F.C2 (sets, relations). A **memory-jog with full formal definitions and pictures** — the same contract as F.C1b: every definition gets a worked example as gentle as arithmetic, a diagram where a diagram carries the idea, a line on why this course needs it, and a ladder for deeper dives.*

Why this chapter exists. Four load-bearing ideas of this book are, underneath, one subject — *order*: the scheduler's parallel waves (antichains), the merge that reconciles parallel work (a semilattice join), the security algebra that ties what-is-specified to what-is-enforced (a Galois

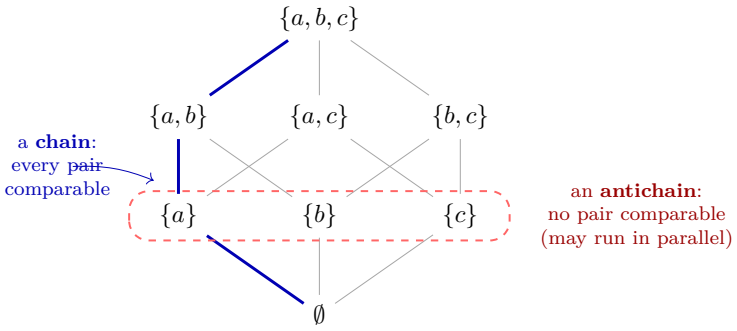


Figure 36.3: The subsets of $\{a, b, c\}$ under $\subseteq$, drawn as a Hasse diagram (edges are covers; height increases upward). One maximal **chain** (solid blue) has length 4 — the poset's height; the middle level (dashed red) is an **antichain** of size 3 — the width. "Partial" order means exactly this: some pairs, like $\{a\}$ and $\{b\}$, are incomparable.

connection), and the self-referencing specification that provably stabilises (a least fixed point). This chapter is that subject, gently.

1. Partial orders and their pictures (Hasse diagrams)

Definition. A **partial order** on a set P is a relation $\sqsubseteq$ that is *reflexive* ($x \sqsubseteq x$), *antisymmetric* ($x \sqsubseteq y$ and $y \sqsubseteq x$ imply $x = y$), and *transitive*. The pair $(P, \sqsubseteq)$ is a **poset**. "Partial" is the honest word: some pairs are simply **incomparable** — neither $x \sqsubseteq y$ nor $y \sqsubseteq x$.

- *Worked example (as gentle as it looks):* subsets of $\{a, b, c\}$ ordered by $\subseteq$. Then $\{a\} \subseteq \{a, b\}$, but $\{a\}$ and $\{b\}$ are incomparable — neither contains the other. That is all "partial" means.
- A **Hasse diagram** draws each element as a node, with an edge upward from x to y when y *covers* x ($x \sqsubset y$ with nothing strictly between); reflexive loops and transitive shortcuts are omitted because the picture implies them.
- *Course-native examples you have already touched:* commit histories ordered by "has all the commits of" (F.E1b — a preorder that becomes a poset on histories-up-to-equality); problems ordered by reduction $\leq_p$ (F.C1b §4); course concepts ordered by "prerequisite-of" — the book's own dependency graph is a poset, and its Hasse diagram is the atlas you navigate by.
- *Why the course cares:* "partial" is the mathematical name for *concurrency*. Two incomparable tasks are exactly two tasks neither of which must wait for the other.

Deeper dive: wiki: Partially ordered set → Davey–Priestley, *Introduction to Lattices and Order*, ch. 1 → Birkhoff (1940).

2. Chains, antichains, and the two decomposition theorems

Definition. In a poset, a **chain** is a set of pairwise *comparable* elements (a totally ordered path up the Hasse diagram); an **antichain** is a set of pairwise *incomparable* elements. The **height** is the number of elements in a longest chain; the **width** the size of a largest antichain. (We count *elements*, not edges, so that Mirsky's theorem below reads cleanly.)

- In the subset cube above: $\emptyset \subset \{a\} \subset \{a, b\} \subset \{a, b, c\}$ is a chain of 4 elements (so the height is 4); $\{\{a\}, \{b\}, \{c\}\}$ is an antichain of size 3 (the width).

Theorem (Mirsky 1971). *A finite poset of height h partitions into exactly h antichains* — take $A_k = \{x : \text{the longest chain ending at } x \text{ has } k \text{ elements}\}$. **Theorem (Dilworth 1950),**

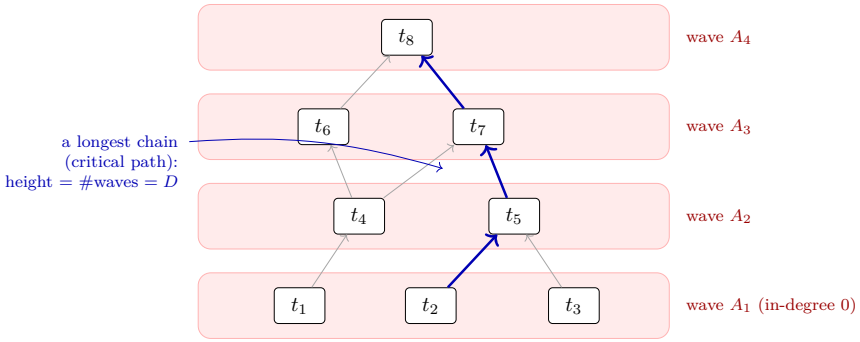


Figure 36.4: Mirsky's theorem as a scheduler: layer each task by the length of the longest dependency chain ending at it. Every layer is an **antichain** — its tasks are mutually independent, so each layer is one **parallel dispatch wave**. The number of waves equals the longest chain (thick blue): the critical path, i.e. the *depth* D of F.C1's work/depth model. U17's scheduler runs these waves after one condensation pass (U11 §4b).

the dual: *a finite poset of width w partitions into w chains.*

- *Why the course cares — this is the scheduler.* Order a DAG's tasks by reachability: Mirsky's layers $A_1, A_2, \ldots$ are precisely the **parallel dispatch waves** — each layer is an antichain, so everything in it may run simultaneously; the number of layers is the critical-path length (the *depth* D of F.C1); and computing the layers is exactly **topological sorting by levels**. Brent's bound (U0P's first certificate), the BMLS/TBMLS scheduler (U17), and the Hasse layout of this book's own atlas are all this one theorem wearing three coats.

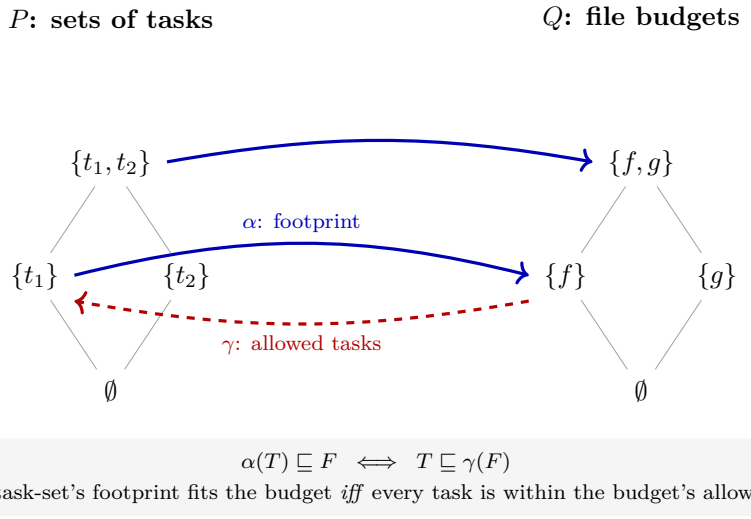
Deeper dive: wiki: Mirsky's theorem · wiki: Dilworth's theorem → Davey–Priestley ch. 1–2 → Mirsky (1971), Dilworth (1950).

3. Lattices and semilattices — when every pair has a best bound

Definition. In a poset, the **join** $x \sqcup y$ is the least upper bound of $\{x, y\}$ (the smallest element above both), when it exists; the **meet** $x \sqcap y$ the greatest lower bound. A **join-semilattice** has all pairwise joins; a **lattice** has both. Join is associative, commutative, and *idempotent* ($x \sqcup x = x$) — and any operation with those three properties determines a semilattice ($x \sqsubseteq y \iff x \sqcup y = y$).

- *Worked example:* subsets with $\cup$ as join and $\cap$ as meet — the cube of §1 is a lattice.
- *The course-native example — already machine-checked:* commit histories with **merge as union**. Idempotence ("merging a state with itself adds nothing"), monotonicity ("the merge preserves both parents"), and least-upper-bound-ness ("any view containing both parents contains the merge") are literally the three σ-merge lemmas of the course's certificate library (`SigmaMerge.lean`) — you verified a semilattice in Lean before this chapter named it. Order-independence of multi-party reconciliation is exactly associativity + commutativity + idempotence.
- *Why the course cares:* "merge conflicts cannot corrupt state, in any order" is not an engineering slogan — it is the statement that reconciliation is a semilattice join.

Subtracting a set: the relative complement. Join builds *up*; sometimes you must take *away* — a security policy grants a set of capabilities but a sovereign rule must *deny* a subset no matter what. The tool is the **relative complement** $x \setminus y$: the $\sqsubseteq$-greatest element that is disjoint from y yet recovers x when the overlap is added back — precisely the two laws $(x \setminus y) \sqcap y = \bot$ and $(x \setminus y) \sqcup (x \sqcap y) = x$. For subsets this is ordinary set-difference. A *distributive* lattice with $\bot$ and this operation (but possibly no top element — capability sets are open-ended) is a **generalized Boolean algebra**. Distributivity is not optional: the five-element diamond lattice M_3 admits



$$\alpha(T) \sqsubseteq F \iff T \sqsubseteq \gamma(F)$$

"the task-set's footprint fits the budget *iff* every task is within the budget's allowance"

Figure 36.5: A Galois connection between two small lattices: task-sets (left, ordered by $\subseteq$) and file budgets (right). α sends tasks to their file footprint; γ sends a budget to the tasks it allows. The adjunction (boxed) is one sentence of security said two ways — and it is the shape of U17's capability/egress algebra: *specify* once on one side, *enforce* provably on the other.

an operation satisfying both laws yet is not distributive. A generalized Boolean algebra has a *binary* subtraction but no *unary* complement. The distinction matters for a hard deny: a weaker *pseudo*-complement (Heyting) leaves $\neg\neg$-stable holes through which a denied capability could re-enter by double negation, so only the genuine relative complement gives a sound seal. This is the algebra U17 §4's deny-tier runs on (machine-checked in `GaloisBoundaries.lean`) — and, pleasingly, it needs no substrate axioms at all: subtraction is pure order theory.

Deeper dive: wiki: Semilattice · wiki: Lattice (order) → Davey–Priestley ch. 2 → Birkhoff (1940).

4. Monotone maps and Galois connections — specification meets enforcement

Definition. $f : P \to Q$ between posets is **monotone** if $x \sqsubseteq y$ implies $f(x) \sqsubseteq f(y)$. A **Galois connection** between posets P and Q is a pair of monotone maps $\alpha : P \to Q$ and $\gamma : Q \to P$ with, for all p, q:

$$\alpha(p) \sqsubseteq q \quad \iff \quad p \sqsubseteq \gamma(q).$$

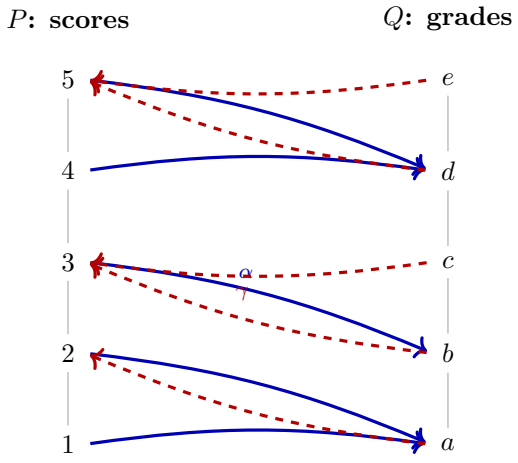
Read it as a *contract between two worlds*: α translates P-things to Q-things, γ translates back, and the equivalence says the two translations are as faithful to each other as the orders allow — $\gamma(\alpha(p)) \sqsupseteq p$ (the round trip loses nothing below you) and $\alpha(\gamma(q)) \sqsubseteq q$ (and claims nothing above).

- *Worked example (two five-element chains, fully by hand):* take $P = \{1 < 2 < 3 < 4 < 5\}$ and $Q = \{a \sqsubset b \sqsubset c \sqsubset d \sqsubset e\}$ — say scores and letter grades. Define the monotone translation

  $$\alpha(1) = a, \quad \alpha(2) = a, \quad \alpha(3) = b, \quad \alpha(4) = d, \quad \alpha(5) = d.$$

  Its adjoint is *forced*: $\gamma(q)$ must be the **largest** score whose grade fits under q, i.e.

  $$\gamma(a) = 2, \quad \gamma(b) = 3, \quad \gamma(c) = 3, \quad \gamma(d) = 5, \quad \gamma(e) = 5.$$



$$\alpha(p) \sqsubseteq q \iff p \le \gamma(q)$$

checked: $\alpha(3) = b \sqsubseteq c$ and $3 \le \gamma(c) = 3$; $\alpha(4) = d \not\sqsubseteq c$ and $4 > \gamma(c) = 3$

Figure 36.6: The same Galois shape on two five-element chains, every arrow checkable by hand: α (solid) grades each score; γ (dashed) is *forced* — the largest score whose grade still fits under the given grade. Follow any score right then back: you never land below where you started ($\gamma(\alpha(p)) \ge p$); follow any grade left then right: you never exceed it ($\alpha(\gamma(q)) \sqsubseteq q$). The ceiling/floor pair on numbers and the capability sentence of Figure 36.5 are this one picture at other scales.

Now check the contract $\alpha(p) \sqsubseteq q \iff p \le \gamma(q)$ on instances: $\alpha(3) = b \sqsubseteq c$ and indeed $3 \le \gamma(c) = 3$; $\alpha(4) = d \not\sqsubseteq c$ and indeed $4 > \gamma(c) = 3$. Round trips: $\gamma(\alpha(3)) = \gamma(b) = 3 \ge 3$ (losing nothing below you) and $\alpha(\gamma(c)) = \alpha(3) = b \sqsubseteq c$ (claiming nothing above). Two things are visible at this scale: γ is *not* an inverse (nothing maps to c or e; the round trip via a lifts 1 to 2), and γ is *unique* — given α, the contract leaves no freedom in choosing it. The familiar ceiling/floor pair $\lceil\cdot\rceil \dashv \iota \dashv \lfloor\cdot\rfloor$ between integers and reals is this same picture on infinite chains: $\lceil x \rceil \le n \iff x \le n$ — "the rounding fits" and "the number fits" are one statement.

- *Worked example (the security shape):* P = subsets of tasks, Q = subsets of files, $\alpha(T)$ = the files the tasks in T may touch, $\gamma(F)$ = the tasks allowed to touch only files in F. The displayed equivalence is then the security sentence: *a task-set's footprint fits a file-budget iff every task is within that budget's allowance.*
- *Why the course cares:* U17's capability/egress algebra ("what a worker may do" vs "what a worker may see") is a Galois connection between *specified* policy and *enforced* policy — the theorem that the deny-set you audited once is the deny-set the kernel enforces everywhere. Galois connections are also how abstract interpretation (static analysis), formal concept analysis, and adjoint functors work — one pattern, everywhere.

Deeper dive: wiki: Galois connection → Davey–Priestley ch. 7 → Ore (1944); (categorical face: adjunctions — a deliberate deep-dive, not a prerequisite).

5. Directed completeness, Scott continuity, and the Kleene fixed point

The last idea powers the book's most self-referential theorem, gently.

Definitions. A poset is a **dcpo** (directed-complete partial order) if every *directed* subset (any finite bunch has an upper bound inside the subset) has a least upper bound $\bigsqcup$, and it has a least element $\bot$ ("nothing yet"). A map f is **Scott-continuous** if it preserves those least upper

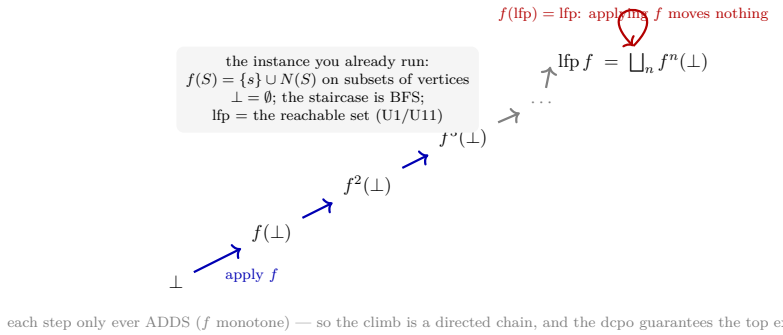


Figure 36.7: The Kleene fixed-point theorem as a staircase: start from "nothing yet" ($\bot$) and apply f repeatedly; monotonicity makes the iterates an ascending chain, directed completeness gives the chain a top, and that top is the *least* fixed point. Every fixpoint loop in this book — BFS, label propagation, and the substrate's self-referencing specification (`Prp-MetacircFP`) — is this one picture.

bounds (in particular it is monotone).

Theorem (Kleene fixed point). *For Scott-continuous f on a dcpo with $\bot$, the ascending chain*

$$\bot \sqsubseteq f(\bot) \sqsubseteq f(f(\bot)) \sqsubseteq \cdots$$

has a least upper bound, and that least upper bound is the LEAST FIXED POINT of f: $f(\mathrm{lfp}) = \mathrm{lfp}$, and it sits below every other fixed point.

- *Worked example:* P = subsets of a graph's vertices ordered by $\subseteq$; $f(S) = \{s\} \cup \text{neighbours}(S)$. Then $\bot = \emptyset$, and the Kleene chain $\emptyset, \{s\}, \{s\} \cup N(s), \ldots$ is *breadth-first search* — its least fixed point is the reachable set. You have been iterating to least fixed points since U1; every fixpoint loop in U11's label propagation is this theorem in work clothes.
- *Why the course cares:* a substrate whose specification includes rules **about its own rules** (the meta-circular core of U17 §2) raises the question "does 'the current specification' even denote something stable?" The paper's answer (`Prp-MetacircFP`) is a least-fixed-point argument of precisely this shape: specification-update is monotone on a suitable order, so iterate-from-$\bot$ stabilises, and the stable point is what "the specification" *means*. Denotational semantics answers "what does a recursive program mean?" the same way (Scott–Strachey) — the deep-dive ladder below is genuinely deep, but the theorem you need is the staircase in the figure.

Deeper dive: wiki: Kleene fixed-point theorem · wiki: Scott continuity → Davey–Priestley ch. 8; Abramsky–Jung, *Domain Theory* (1994) → Kleene (1952), Scott (1970).

6. The map of this chapter

$$\text{poset} \rightarrow \begin{cases} \text{chains/antichains} \rightarrow \text{Mirsky} \rightarrow \textbf{scheduling waves (U3/U17)} \\ \text{joins} \rightarrow \text{semilattice} \rightarrow \textbf{merge/reconciliation } (\sigma\text{-merge}) \\ \text{two posets} + \alpha, \gamma \rightarrow \text{Galois} \rightarrow \textbf{security algebra (U17)} \\ \text{dcpo} + \bot + \text{continuity} \rightarrow \text{Kleene} \rightarrow \textbf{stable self-reference (MetacircFP)} \end{cases}$$

One subject, four load-bearing exits. A reader who arrived knowing only "$\subseteq$ orders sets" now owns the order theory this book actually uses — with pictures.

Where this is used

U3/U17 (Mirsky layers = dispatch waves; TBMLS stage M); the σ-merge certificates (semilattice, machine-checked); U17 §4 (Galois capability/egress); the metacircular fixed point (U17 §2, `Prp-MetacircFP`, Module-3 project 4); U11 label propagation + F.C1b's reduction preorder (fixpoints and orders you had already met); the book's own atlas (the Hasse diagram you navigate).

F.C3 — Logic, induction, and reading a machine-checked proof

*Foundation chapter (Pillar 3, Computational Foundations). A root: it feeds U0P (proof-checkers) and every unit that ships a Lean cert. §§1–2 are the short **memory-jog**: just enough logic and induction to **state** a claim precisely and to **read** a proof the machine has already checked. §§3–9 are the full, self-contained account behind the Module 1 lecture on Lean, logic and proof checkers; the lecture is a condensed version of these sections. You are not asked to become a proof engineer, but after §§3–9 you should know exactly what a proof checker certifies, why it can be trusted, and what it cannot tell you.*

1. Logic and induction — stating a claim precisely

- **Propositional / predicate logic.** Connectives ($\wedge, \vee, \neg, \Rightarrow$) and quantifiers ($\forall, \exists$). "For all inputs of size n, the cost is $\leq f(n)$" is a $\forall$-statement — a cost bound is a logical claim, not a vibe.
- **Induction.** To prove $P(n)$ for all n: show $P(0)$ (base) and $P(k) \Rightarrow P(k+1)$ (step). *Strong* induction assumes $P(0), \ldots, P(k)$; *well-founded* induction generalises it to any decreasing measure (why a recursion that shrinks its argument terminates).

Induction is the shape of almost every cost proof in the units: the scan work $2(2^k - 1)$, a recurrence's closed form, a loop invariant — each is "base case, then it is preserved by one step."

2. Reading a machine-checked proof

A **proof assistant** (we use Lean) checks a proof the way a compiler type-checks a program: a theorem is a type, a proof is a term of that type, and the kernel either accepts it or does not.

- **"the checker accepted it"** means: the proof term type-checks against the library **and** contains no `sorry` (no admitted holes). The verdict is a build exit code — *no human, and no AI, in the loop* (the U0P discipline; F.E2 ships the checker as a container).
- **the axiom footprint.** `#print axioms <thm>` lists what the proof rests on; a footprint of only the standard logical axioms (and no `sorryAx`) is what "certified" means here.

You do not have to *write* these to *trust* them: reading the theorem statement + confirming the sorry-free green build is the skill this chapter teaches. *Cert pointers:* `BrentBound` (U0/U3), `ScanCost` (U1), `MergeCost` (U2), `GrahamBound` (U3), `ReservoirUniform` (U4), the CMDE certs (U15) — each is a green, sorry-free statement you can read.

3. First-order logic: language, proofs, truth

This section recalls, precisely, the parts of a first course in mathematical logic that a proof checker makes concrete. Nothing here is specific to Lean; §5 connects it to Lean.

3.1 The language: signatures, terms, formulas

A **signature** σ is a set of constant symbols, function symbols and relation symbols, each function and relation symbol with a fixed number of arguments (its *arity*). The signature of arithmetic

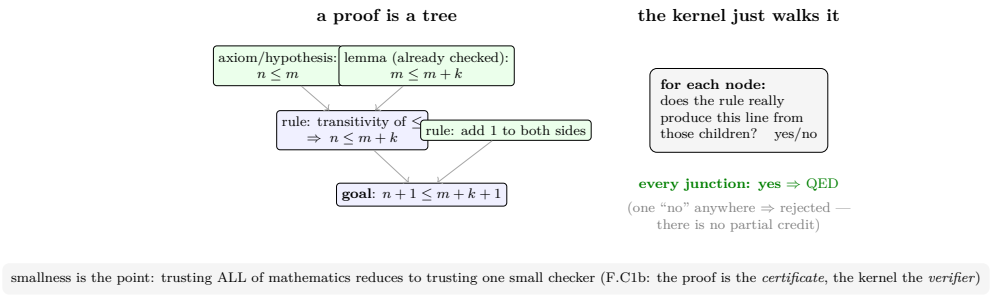


Figure 36.8: What "machine-checked" means. A proof is a finite tree: leaves are axioms, hypotheses, or already-checked lemmas; every internal node applies one inference rule. The kernel does nothing clever — it walks the tree asking one local question per junction. Checking is easy precisely where discovery was hard: the proof object is a certificate, the kernel a fast verifier.

is $\sigma_{\mathrm{ar}} = \{0, S, +, \cdot, <\}$: a constant 0, a unary function symbol S (successor), binary function symbols $+$ and $\cdot$, and a binary relation symbol $<$. Fix also an infinite supply of variables $x, y, z, x_1, x_2, \ldots$.

The **terms** are defined inductively: every variable and every constant is a term, and if f is a k-ary function symbol and $t_1, \ldots, t_k$ are terms then $f(t_1, \ldots, t_k)$ is a term. So x, 0, $S(0)$ and $x + S(y)$ (that is, $+(x, S(y))$) are terms. The **formulas** are also defined inductively:

- $t_1 = t_2$ and $R(t_1, \ldots, t_k)$, for terms t_i and a k-ary relation symbol R, are *atomic* formulas;
- if φ and ψ are formulas, so are $\neg\varphi$, $\varphi \wedge \psi$, $\varphi \vee \psi$ and $\varphi \Rightarrow \psi$;
- if φ is a formula and x a variable, $\forall x\, \varphi$ and $\exists x\, \varphi$ are formulas.

We write $\varphi \iff \psi$ for $(\varphi \Rightarrow \psi) \wedge (\psi \Rightarrow \varphi)$ and $t_1 \neq t_2$ for $\neg(t_1 = t_2)$.

An occurrence of x is **bound** when it lies inside some $\forall x$ or $\exists x$, and **free** otherwise. A **sentence** is a formula with no free variables. For example, $\forall x\, \exists y\, (x < y)$ is a sentence ("there is no largest element"), while $x < S(x)$ is a formula with the free variable x.

Two consequences of the inductive definitions matter for machines. First, a formula is a *finite string over a fixed alphabet*, so a program can read, store and compare it. Second, every definition "by recursion on formulas" (the free variables of φ, the truth of φ, the translation of φ into another language) is well defined, and every property of all formulas can be proved "by induction on formulas": check the atomic case, then check that each connective and quantifier preserves the property. This is the same induction as in §1, on a different inductive type.

3.2 Theories and proofs: syntax only

A **theory** T is a set of sentences, its **axioms**. The standard example is **Peano arithmetic** (PA). Here S is the successor function, "add one": $S(x)$ is intended to mean $x + 1$, so 1 abbreviates $S(0)$, 2 abbreviates $S(S(0))$, and every natural number is 0 with some number of S's applied. The axioms of PA say what S, $+$ and $\cdot$ must satisfy:

- $\forall x\, (S(x) \neq 0)$ and $\forall x\, \forall y\, (S(x) = S(y) \Rightarrow x = y)$;
- $\forall x\, (x + 0 = x)$ and $\forall x\, \forall y\, (x + S(y) = S(x + y))$;
- $\forall x\, (x \cdot 0 = 0)$ and $\forall x\, \forall y\, (x \cdot S(y) = x \cdot y + x)$;
- $\forall x\, \forall y\, (x < y \iff \exists z\, (x + S(z) = y))$;
- the **induction schema**: for every formula $\varphi(x)$, possibly with further free variables (its parameters), the universal closure of $\varphi(0) \wedge \forall k\, (\varphi(k) \Rightarrow \varphi(S(k))) \Rightarrow \forall n\, \varphi(n)$.

The induction schema is infinitely many axioms, one for each formula, but a program can decide

whether a given sentence is one of them. A theory whose axioms can be recognised by a program in this way is **recursively axiomatizable**, and PA is such a theory. Without $\cdot$ and its axioms, the theory of $(\mathbb{N}, 0, S, +, <)$ is **Presburger arithmetic**, which, unlike PA, is complete and decidable (Presburger 1929; see §6.2).

A **proof** of φ from T is a *finite* sequence of formulas ending in φ, each of which is an axiom of T, a logical axiom, or follows from earlier formulas by a **rule of inference**. There are many equivalent proof systems (Hilbert-style systems, natural deduction, sequent calculi); natural deduction, which Lean follows, has an introduction and an elimination rule for each connective and quantifier, for example:

rule	from	infer
modus ponens ($\Rightarrow$-elimination)	A and $A \Rightarrow B$	B
$\Rightarrow$-introduction	a derivation of B from the assumption A	$A \Rightarrow B$, discharging A
$\wedge$-introduction	A and B	$A \wedge B$
$\wedge$-elimination	$A \wedge B$	A (and, separately, B)
$\forall$-elimination	$\forall x\, \varphi(x)$	$\varphi(t)$ for any term t (substitutable for x)
$\forall$-introduction	$\varphi(x)$, with x free in no open assumption	$\forall x\, \varphi(x)$

We write $T \vdash \varphi$, read "T proves φ", when there is a proof of φ from T. This is a claim about strings and rules, not about truth. **Checking** a proof is pure symbol-pushing: for each line, is it an axiom, or does it follow from earlier lines by one of finitely many rules? That is a mechanical check that always terminates, and a proof checker is exactly that check, implemented. A theory is **consistent** when it does not prove every sentence; equivalently, when there is no φ with both $T \vdash \varphi$ and $T \vdash \neg\varphi$.

3.3 Structures and truth: semantics

A **structure** $\mathcal{M}$ for a signature σ is a non-empty set M, its **domain**, together with an interpretation of every symbol: an element $c^{\mathcal{M}} \in M$ for each constant, a function $f^{\mathcal{M}} : M^k \to M$ for each k-ary function symbol, and a set $R^{\mathcal{M}} \subseteq M^k$ for each k-ary relation symbol. Two examples: $\mathcal{N} = (\mathbb{N}, 0, S, +, \cdot, <)$, the standard natural numbers, and $\mathcal{A} = (\{0, 1, 2\}, <)$, a three-element order, a structure for the smaller signature $\{<\}$.

Truth is defined by recursion on formulas (Tarski). Given an **assignment** s of elements of M to the variables, each term t gets a value $t^{\mathcal{M}}[s] \in M$, and $\mathcal{M} \models \varphi\,[s]$, "$\varphi$ is true in $\mathcal{M}$ under s", is defined clause by clause:

- $\mathcal{M} \models (t_1 = t_2)[s]$ iff $t_1^{\mathcal{M}}[s] = t_2^{\mathcal{M}}[s]$, and $\mathcal{M} \models R(t_1, \ldots, t_k)[s]$ iff $(t_1^{\mathcal{M}}[s], \ldots, t_k^{\mathcal{M}}[s]) \in R^{\mathcal{M}}$;
- $\neg$, $\wedge$, $\vee$, $\Rightarrow$ are read as the usual truth tables;
- $\mathcal{M} \models \forall x\, \varphi\,[s]$ iff $\mathcal{M} \models \varphi\,[s(x \mapsto a)]$ for **every** $a \in M$, and $\mathcal{M} \models \exists x\, \varphi\,[s]$ iff it holds for **some** $a \in M$.

For a sentence the assignment does not matter, and we write $\mathcal{M} \models \varphi$. A **model** of T is a structure in which every axiom of T is true. We write $T \models \varphi$, read "T entails φ", when φ is true in **every** model of T, and φ is **valid** when it is true in every structure ($\emptyset \models \varphi$).

So $\vdash$ is about strings and rules (syntax) and $\models$ is about structures and truth (semantics). The great theorems of logic connect the two.

3.4 Soundness, and completeness (Gödel 1929/30)

Theorem (soundness). If $T \vdash \varphi$ then $T \models \varphi$.

Proof sketch. By induction on derivations, show: if φ is derived from T plus open assumptions Γ, then φ is true in every model of T under every assignment that makes Γ true. Axioms of T and

logical axioms are true; each rule preserves the property ($\Rightarrow$-introduction because discharging A matches the truth table of $A \Rightarrow B$). A proof has no open assumptions left, so its conclusion is true in every model of T. $\square$

Soundness is the property a proof checker must have, and it is the one we rely on: an accepted proof yields a true statement in every model of the axioms.

Theorem (completeness; Gödel 1929/30). For first-order logic, if $T \models \varphi$ then $T \vdash \varphi$. Together with soundness: $T \vdash \varphi \iff T \models \varphi$.

Idea of the proof (Henkin 1949). It suffices to show that every consistent theory has a model. Extend T to a consistent theory that decides every sentence and has, for each $\exists x\, \varphi(x)$ it proves, a constant c with $\varphi(c)$ (a "witness"); then build a structure whose elements are the terms of the extended language, identified when the theory proves them equal. $\square$

Two corollaries are used constantly.

- **Compactness.** If every finite subset of T has a model, then T has a model. (A proof of a contradiction from T is finite, so it uses only finitely many axioms.)
- **Löwenheim–Skolem.** If a theory in a countable language has an infinite model, it has models of every infinite cardinality. In particular first-order logic cannot pin down $\mathbb{N}$ up to isomorphism: PA has **non-standard models**, with elements larger than every $S(S(\cdots S(0)))$. (Compactness already gives this: add a constant c and the axioms $c > n$ for every numeral n.)

So for a first-order theory, a checker for $\vdash$ is also a checker for "true in every model of the axioms". Lean's logic is not first-order: it is a higher-order type theory (§5). For Lean, read "accepted" as "derivable in Lean's logic", and rely on its soundness relative to set theory (§5.4).

3.5 Decidable, recursively enumerable, and undecidable

A set X of strings is **decidable** when some program, given any string, halts and answers correctly whether it is in X. It is **recursively enumerable** (r.e.) when some program lists exactly its members (equivalently, halts exactly on the members); it is **co-r.e.** when its complement is r.e. A set is decidable exactly when it is both r.e. and co-r.e.

- **Proof checking is decidable**: given T recursively axiomatizable, a candidate proof and φ, a program decides whether it is a proof of φ from T.
- **Theoremhood is r.e.**: list all finite strings, check each one, and print the conclusions of the proofs. By completeness, the valid first-order sentences are therefore r.e.
- **Validity is not decidable** (Church 1936, Turing 1936): no program decides whether an arbitrary first-order sentence is valid, as soon as the signature has a binary relation symbol. So proof *search* can confirm a valid sentence, eventually, but can never in general tell you that no proof exists.

This is the logical form of a practical fact: **checking** a proof is easy, **finding** one is not (§6).

3.6 Incompleteness (Gödel 1931, Rosser 1936)

Theorem (first incompleteness). Let T be a consistent, recursively axiomatizable theory that contains enough arithmetic (Robinson arithmetic Q, a finite fragment of PA, suffices). Then there is a sentence G such that $T \nvdash G$ and $T \nvdash \neg G$.

Theorem (second incompleteness). If moreover T contains a little more arithmetic (PA suffices), then T does not prove the arithmetic sentence $\mathrm{Con}(T)$ that expresses its own consistency. (Here $\mathrm{Con}(T)$ is the canonical sentence $\neg \mathrm{Prov}_T(\ulcorner 0 = S(0) \urcorner)$, built from a natural formula defining T's axioms.)

Idea of the proof. Code formulas and proofs as numbers (Gödel numbering). Because proof checking is mechanical, the relation "p codes a proof of the formula coded by n" is expressible, and provable when true, in T. The *diagonal lemma* then gives a sentence G that T proves equivalent to "G is not provable in T". If $T \vdash G$, then $T \vdash \neg\mathrm{Prov}(\ulcorner G \urcorner)$ by the diagonal equivalence, and, by checking that proof inside T, also $T \vdash \mathrm{Prov}(\ulcorner G \urcorner)$, contradicting consistency; so $T \nvdash G$. Gödel needed a slightly stronger assumption (ω-consistency) to rule out $T \vdash \neg G$; Rosser's variant of the sentence needs only consistency. The second theorem formalises the first argument inside T. □

What this means for proof checkers:

- The theories behind the checkers we use (PA, ZFC, Lean's type theory) contain enough arithmetic, so each has true arithmetic sentences it cannot prove. For PA, such a sentence is true in $\mathbb{N}$ but, by completeness, false in some non-standard model.
- "The checker accepted it" means *derivable from the checker's axioms.* That is exactly what we want, and no more.
- No checker proves its own logic consistent. We rely instead on consistency relative to a standard foundation (§5.4), on the smallness of the kernel, and on independent re-checking (§6.1).
- Incompleteness never makes a sound checker accept a non-proof. **Soundness is the part we rely on.**

4. Finite model theory: when you can check by computing

Finite model theory studies first-order logic over *finite* structures. They are the structures of computing: a database is a finite structure (a domain of values with relations, the tables), and so is a graph, the state of a cluster, a schedule, or a directed acyclic graph of tasks.

4.1 Model checking is decidable, and what it costs

Model checking asks, for a finite structure $\mathcal{A}$ and a sentence φ, whether $\mathcal{A} \models \varphi$. The Tarski clauses of §3.3 are already an algorithm: for each quantifier, loop over the domain. With k nested quantifiers over a domain of n elements that is about n^k evaluations of atomic formulas. So:

- for a **fixed** sentence, model checking takes polynomial time in the size of the structure (data complexity; in fact it is in logarithmic space);
- when the sentence is part of the input too, the problem is PSPACE-complete (combined complexity; Stockmeyer 1974, Vardi 1982).

This is the logic behind database query languages: first-order logic under active-domain semantics (the relational calculus) is, by Codd's theorem (1972), as expressive as the relational algebra underlying SQL's core, and evaluating a fixed query is cheap in the size of the data.

Worked. Take $\mathcal{A} = (\{0, 1, 2\}, <)$.

- $\forall x\, \exists y\, (x < y)$ is **false**: at $x = 2$, none of the three values of y has $2 < y$.
- $\exists x\, \forall y\, \neg(y < x)$ is **true**: at $x = 0$, no y is below 0, so a least element exists.

In $\mathcal{N}$ the first sentence is true. In fact *no* finite strict linear order satisfies it, so this sentence has only infinite models among linear orders. It is also an example of a sentence that no amount of finite testing can confirm.

4.2 Over all finite structures, logic changes

Restricting attention to finite structures changes the theory of first-order logic.

- **Compactness fails.** Let λ_n be the sentence "there are at least n elements" ($\exists x_1 \cdots \exists x_n \bigwedge_{i<j} x_i \neq x_j$). Every finite set of the λ_n has a finite model, but the whole set has no finite model.
- **Completeness fails: Trakhtenbrot's theorem** (1950). For any signature with at least one binary relation symbol, the set of sentences true in *all finite* structures is not recursively enumerable. So no proof system, however clever, lists exactly the finitely valid sentences.

Idea of the proof. For any Turing machine M, write a sentence φ_M that has a finite model iff M halts on the empty input: it describes a finite linear order of time steps and tape cells, and relations recording the state, the head position and the tape symbol at each time, so that a halting run gives a finite model and any finite model contains one. (A further standard encoding reduces all these relations to a single binary relation.) So finite *satisfiability* is undecidable. It is r.e. (list the finite structures and model-check each), so its complement is not r.e.; and φ is finitely valid iff $\neg\varphi$ is not finitely satisfiable. □

So finite validity is co-r.e. but not r.e.: to refute a claim about all finite structures you can search for a counterexample and will find one if it exists, but there is no complete method of proof.

- **Expressiveness is limited, and there is a game to show it.** Some simple properties are not first-order definable on finite structures. For example, no first-order sentence in the empty signature (or the signature $\{<\}$ of linear orders) holds exactly in the structures with an **even** number of elements. The **quantifier depth** of a formula is the maximum nesting of quantifiers (atomic formulas have depth 0; $\neg\varphi$ has the depth of φ; $\varphi \wedge \psi$ the larger of the two; $\exists x\,\varphi$ and $\forall x\,\varphi$ that of φ plus 1). In the k-round **Ehrenfeucht–Fraïssé game** on structures $\mathcal{A}$ and $\mathcal{B}$, in each round the first player, *Spoiler*, picks an element of either structure, and the second player, *Duplicator*, answers with an element of the other. Duplicator wins if, after k rounds, the map $a_i \mapsto b_i$ between the chosen elements is a partial isomorphism: it respects $=$ and every relation. **Theorem** (Ehrenfeucht 1961, Fraïssé 1954): for a finite relational signature, $\mathcal{A}$ and $\mathcal{B}$ satisfy the same sentences of quantifier depth at most k iff Duplicator has a winning strategy in the k-round game. One shows that Duplicator wins on sets (or linear orders) of sizes m and $m+1$ once both are large enough relative to k; EF3.8 does the smallest case.
- **Every sentence is almost surely true or almost surely false.** On random graphs with n vertices, each edge present with probability $\frac{1}{2}$, every first-order sentence has probability tending to 0 or to 1 as $n \to \infty$ (the **zero–one law**; Glebskii et al. 1969, Fagin 1976). So first-order logic cannot express "the number of vertices is even", whose probability does not converge.

4.3 The lesson for engineers

Checking one finite instance is easy: compute. A claim about **all** instances (every input size n, every list of level widths, every number of processors p) is a statement about infinitely many structures, and computing instances never finishes it. Testing is model checking finitely many instances; a theorem needs a proof.

Worked: why testing is not proving. Claim: "$n^2 + n + 41$ is prime for every natural number n."

n	0	1	2	3	…	39	**40**
n^2+n+41	41	43	47	53	…	1601	**1681** $= 41 \times 41$

The claim passes 40 tests in a row, then fails at $n = 40$. (It fails for $n = 41$ as well, because $41^2 + 41 + 41 = 41 \times 43$.) Lean offers two honest tools, and §6 shows both: **compute** a decidable statement on a finite instance, which is model checking (the `decide` tactic), and **prove** a statement for all n, with induction, rewriting, and automation for decidable fragments (the `omega` tactic).

5. Lean: propositions as types, proofs as programs

Lean is a programming language and a proof checker in one. Its logic is not first-order logic with axioms for the objects you care about; it is a **dependent type theory**, in which numbers, lists and proofs are all defined inside the language.

5.1 Types, and propositions as types

Every expression in Lean has a **type**: `3 : Nat`, `[1, 2] : List Nat`, `Nat.add : Nat → Nat → Nat`. Types themselves live in **universes**: `Nat : Type`, and `Type : Type 1`. There is one special universe, `Prop`, whose elements are propositions: `2 + 2 = 4 : Prop`.

The **Curry–Howard correspondence** is the organising idea: a proposition A is a type, and a proof of A is a term of type A. So checking a proof is type-checking a program.

logic	Lean type	a proof is…	natural-deduction rule
$A \Rightarrow B$	`A → B`	a function turning proofs of A into proofs of B	$\Rightarrow$-intro = `fun`; modus ponens = application
$A \wedge B$	`A ∧ B`	a pair `⟨a, b⟩`	$\wedge$-intro / elim
$A \vee B$	`A ∨ B`	`Or.inl a` or `Or.inr b`	$\vee$-intro; elim = case split
$\forall x{:}X,\ P\,x$	`∀ x : X, P x`	a function giving, for each `x`, a proof of `P x`	$\forall$-intro / elim
$\exists x,\ P\,x$	`∃ x, P x`	a witness with a proof, `⟨w, h⟩`	$\exists$-intro
$\bot$ (false)	`False`	none exists: an empty type	ex falso = `False.elim`
$\neg A$	`¬A`, i.e. `A → False`	a function into the empty type	

`∀ x : X, P x` is a **dependent** function type: the type of the output, `P x`, depends on the input `x`. That is what "dependent type theory" means, and it is not only for propositions. A list whose length is part of its type is a dependent type:

```
inductive Vec (α : Type) : Nat → Type where
  | nil  : Vec α 0
  | cons : α → Vec α n → Vec α (n + 1)

def Vec.head : Vec α (n + 1) → α
  | .cons a _ => a
```

`Vec.head` needs no case for the empty vector: its type says the input has length $n + 1$, and the checker knows `nil` has length 0. A run-time error has become a type error.

Two proofs that are literally programs. For $A \wedge B \Rightarrow B \wedge A$, take the pair apart and put it back together swapped; for transitivity of implication, compose the two functions; for $A \vee B \Rightarrow B \vee A$, split the cases:

```
theorem my_and_swap (A B : Prop) : A ∧ B → B ∧ A :=
  fun ⟨a, b⟩ => ⟨b, a⟩

theorem my_imp_trans (A B C : Prop) : (A → B) → (B → C) → (A → C) :=
  fun f g a => g (f a)
```

```
theorem my_or_swap (A B : Prop) : A ∨ B → B ∨ A :=
  fun h => h.elim (fun a => Or.inr a) (fun b => Or.inl b)
```

The **statement** is the type (after the colon); the **proof** is the program (after `:=`). The kernel checks that the program has exactly that type. These are the natural-deduction proofs of §3.2, written as code. (The names start with `my_` so they do not clash with the library's own versions.)

Constructive by default. Lean's core logic is constructive: `A ∨ ¬A` is not provable for an arbitrary `A` without an extra axiom. You get it as `Classical.em A`, which is derived from the axiom `Classical.choice` together with `propext` and `Quot.sound` (Diaconescu's theorem): the classical logic of your logic course, switched on explicitly (§5.4).

5.2 The symbols you will read

you see	read it as
`theorem name (x : X) : P x := proof`	"theorem *name*: for `x` of type `X`, `P x` holds; here is the proof"
`example : P := proof`	the same, but unnamed (a quick check)
`def f (x : X) : Y := e`	a definition: the function `f`
`x : X`	"`x` has type `X`": `n : Nat`, `h : a ≤ b` (a proof is a value too)
`Prop`, `Type`, `Nat`, `Fin 3`, `List Nat`	propositions; the universe of ordinary types; naturals; $\{0,1,2\}$; lists of naturals
$\to\ \forall\ \exists\ \wedge\ \vee\ \neg\ \leftrightarrow$	implies · for all · exists · and · or · not · iff
`fun x => e`	the function $x \mapsto e$
`⟨a, b⟩`	build a pair / an "and" / a witness
`by tac₁; tac₂`	"prove it with these **tactics**" (§5.6)
`rfl`	"both sides compute to the same value" (reflexivity)
`sorry`	"I have not proved this". The checker marks it; it is **never** a proof

5.3 Inductive types: the numbers are defined, not axiomatized

In PA, 0, S and induction are axioms. In Lean, the natural numbers are **defined** as an inductive type. This is core Lean's own definition, shown here, not for pasting:

```
inductive Nat where
  | zero : Nat
  | succ (n : Nat) : Nat
```

An inductive definition lists the **constructors**, the only ways to build an element, and Lean derives from it a **recursor**, `Nat.rec`, which is a primitive rule of the logic. Read as a program, the recursor is *structural recursion*: define `f zero`, and define `f (succ n)` from `n` and `f n`. Read as a proof, it is *induction*: prove `P zero`, and prove `P (succ n)` from `P n`. So the Peano axioms become theorems: `succ n ≠ zero` holds because the two constructors are different, and injectivity of `succ` holds because constructors are injective.

Functions are defined by recursion and pattern matching. Addition recurses on its **second** argument:

```
Nat.add n zero      = n                        (A1)
Nat.add n (succ m)  = succ (Nat.add n m)       (A2)
```

These are PA's addition axioms of §3.2, now a definition. Lean accepts a recursive definition only if it can see that it terminates (here, the second argument shrinks), because a non-terminating "proof" would prove anything.

5.4 Lean's logic and its three axioms

Lean's logic is a dependent type theory, a relative of the calculus of inductive constructions. Lean's type theory, together with the three axioms below, is equiconsistent with ZFC plus, for each $n < \omega$, the axiom "there are n inaccessible cardinals" (Carneiro 2019, for Lean 3's type theory, which Lean 4 essentially shares). This is the relative-consistency guarantee of §3.6: if Lean's logic proved a contradiction, so would that well-studied set theory. On top of the type theory sit three axioms, the **classical trio**:

- `propext`: equivalent propositions are equal;
- `Classical.choice`: a nonempty type has an element you may pick (with the other two, this gives excluded middle);
- `Quot.sound`: related elements are equal in a quotient type, which gives function extensionality.

`#print axioms <thm>` lists which of them a theorem uses. It can also list `sorryAx` (a `sorry`, §7) or an auxiliary axiom such as `foo._native.native_decide.ax_1_1`, which records a compiler-trusting `native_decide` step. The course rule rejects both.

5.5 `1 + 1 = 2`, all the way down: definitional equality

```
theorem one_plus_one : 1 + 1 = 2 := rfl
```

piece	what it is, once the notation is unfolded
`theorem one_plus_one`	a **name** for what follows
`1`, `2`	numerals of type `Nat`: `1` is `succ zero` and `2` is `succ (succ zero)`
`+`	notation for `HAdd.hAdd`, which on `Nat` is the function `Nat.add`
`=`	`Eq`, an inductive type with **one** constructor, `Eq.refl a : a = a` ("a equals itself")
`: 1 + 1 = 2`	the **statement**, i.e. the type `@Eq Nat (Nat.add 1 1) 2`
`:= rfl`	the **proof**: `rfl` is `Eq.refl ?a`, whose type is `?a = ?a` for some `?a` to be found

So the question Lean faces is: can `Eq.refl ?a`, a proof of "something equals itself", have the type `1 + 1 = 2`? Only if a single `?a` fits both sides, so Lean tries `?a := 1 + 1`, and must then check that `1 + 1` and `2` are the same value. Two terms that reduce to the same value by unfolding definitions are **definitionally equal**, and the kernel decides definitional equality by computing:

```
1 + 1  =  Nat.add (succ zero) (succ zero)
       =  succ (Nat.add (succ zero) zero)       by A2
       =  succ (succ zero)                      by A1
       =  2                                     same value as the right-hand
         side: ACCEPTED
```

(The elaborator solves for `?a`; the kernel then re-checks the finished term `@rfl Nat (1 + 1) : 1 + 1 = 2`. For speed, the kernel does numeral arithmetic with built-in big-number routines; the result is the same as this unfolding.)

Why `1 + 1 = 3` is rejected: `1 + 1` computes to `succ (succ zero)`, while `3` is `succ (succ (succ zero))`; strip the two outer `succ`s and you compare `zero` with `succ zero`, different constructors that never reduce to each other. Lean reports an error while elaborating `rfl` (the kernel would reject the term for the same reason), the file does not check, and Lab 0's `run-lab.sh` writes no receipt. `#print axioms one_plus_one` lists **no axioms**, because only computation was used.

5.6 Computation versus induction, and the proof state

Because `+` recurses on the second argument, $n+0$ *computes* to n, so `rfl` proves it. But $0+n=n$ is stuck: nothing reduces until you know which constructor n is. Equalities like this one, true for every n but not by computation, are **propositional** equalities, and they need a proof, here by **induction**:

```
theorem my_add_zero (n : Nat) : n + 0 = n := rfl

theorem my_zero_add (n : Nat) : 0 + n = n := by
  induction n with
  | zero      => rfl                              -- goal: 0 + 0 = 0, computes
  | succ k ih => rw [← Nat.add_assoc, ih]      -- goal: 0 + (k + 1) = k + 1
```

In the step case, `ih : 0 + k = k` is the induction hypothesis; `← Nat.add_assoc` rewrites $0+(k+1)$ to $(0+k)+1$, and `ih` turns that into $k+1$, so both sides are identical and the goal closes. These are exactly the two cases of the induction schema of §3.2, now as the two constructors of `Nat`.

After `by`, Lean works in **tactic mode**. It shows **goals**: some hypotheses, then the target after $\vdash$. Each tactic transforms the goals until none remain:

```
before:           after `induction n with`:       then:
  n : Nat           case zero                       case zero: `rfl` closes 0 + 0
    = 0
  ⊢ 0 + n = n         ⊢ 0 + 0 = 0                   case succ: `rw [←
    Nat.add_assoc, ih]`
                    case succ                           turns the goal into k + 1
                      = k + 1,
                      k : Nat                           which rw closes by rfl
                      ih : 0 + k = k                no goals left: the proof is
                        complete
                      ⊢ 0 + (k + 1) = k + 1
```

Tactics are proof-writing helpers, not trusted code. Every tactic run produces a proof term, and only the kernel checks that term. A buggy tactic can fail to find a proof, but it cannot make the kernel accept a false one.

6. The kernel, the trust base, and finding versus checking

6.1 What you trust

```
you / a tactic / an LLM  -->  proof TERM  -->  KERNEL (small, fixed)  -->
  accept / reject
(untrusted: may be wrong,     (a program)      type-checks the term
 clever, or buggy)                             against the statement
```

What you must trust:

1. that the kernel is implemented correctly. It is small, and independent re-checkers exist, for example nanoda_lib in Rust and lean4lean; this includes the kernel's built-in fast arithmetic on numerals;
2. that the axioms are the ones you intended: `#print axioms` lists them;
3. that the logic is consistent, which rests on the relative consistency of §5.4.

What you do *not* need to trust: the tactics, the libraries, the person who wrote the proof, or a language model that drafted it. The proof is a checkable object, and a small program, the kernel

or an independent re-checker you could write yourself, checks it: the **de Bruijn criterion**. Barendregt named it after N. G. de Bruijn, whose AUTOMATH checker (begun in 1967) first worked this way.

6.2 Checking is mechanical; finding is not

Checking a given proof term is mechanical and, in practice, fast. **Finding** one is another matter: by §3.5, whether a proof exists is undecidable. You can search, and you will find a proof if one exists, but you never know when to give up. For some **decidable fragments**, though, there are complete or practically effective procedures, and Lean ships automation that needs no language model:

tactic	what it does	example
`decide`	computes the truth value of a proposition that has a `Decidable` instance (model checking a finite instance)	`2 + 3 = 5`; `∀ x, x < 3 → x * x < 9`
`omega`	a solver for linear arithmetic over $\mathbb{N}$ and $\mathbb{Z}$ (built from $=, <, \leq, +, -$, and $\cdot$, /, % by literals). The fragment is decidable (Presburger 1929); `omega` is not yet a complete decision procedure	`n ≤ n + n`
`simp`	rewrites with a database of equations; a heuristic, not a decision procedure	cleans up `0 + n`, `n * 1`, …
`rfl`	both sides compute to the same value	`1 + 1 = 2`

```
example : 2 + 3 = 5 := by decide                    -- compute both sides
example : ∀ x, x < 3 → x * x < 9 := by decide       -- a bounded ∀: check x = 0,
  1, 2
example (n : Nat) : n ≤ n + n := by omega           -- ALL n: linear arithmetic,
  closed by omega
```

The first two are model checking (§4.1): finitely many cases, computed. `decide` works because Lean has a `Decidable` instance for each of these propositions, a program that computes `true` or `false` together with a proof that the answer is right; for a bounded quantifier `∀ x, x < 3 → …` the instance tries `x = 0, 1, 2`. The third is a theorem about infinitely many n, and it is still automatic, because this fragment of linear arithmetic is decidable. Compare §3.5 and §4.2: validity in general is undecidable, and finite validity is not even r.e., but some useful theories, such as linear arithmetic over $\mathbb{N}$, are decidable. Knowing which fragment you are in is half the craft.

None of these tactics is a language model: every certificate in this course is closed by deterministic tactics and library lemmas.

7. How the checker keeps you honest, and what it cannot check

7.1 `sorry` and `#print axioms`

```
theorem not_yet : ∀ n : Nat, n * n ≥ n := by sorry
#print axioms not_yet        -- 'not_yet' depends on axioms: [sorryAx]

theorem one_plus_one : 1 + 1 = 2 := rfl
#print axioms one_plus_one   -- 'one_plus_one' does not depend on any axioms
```

`sorry` lets a file compile with a **warning** (`declaration uses \`sorry\``), and the axiom list then shows `sorryAx`, the "trust me" axiom. **The course rule:** a certificate counts only when `sorry_count = 0` and the axiom list holds nothing beyond the classical trio. (Try replacing

`sorry` by `omega`: it fails, because $n \cdot n$ is not linear and `omega` treats it as an unknown. The core lemma `Nat.le_mul_self : n ≤ n * n` proves it: `intro n; exact Nat.le_mul_self n`.)

7.2 What "CERTIFIED" does not mean: the statement can still be wrong

The kernel checks that the proof proves the statement *as written*, not that the statement says what you meant. In Lean, natural-number division is a total function with $a/0 = 0$, so

```
example : (5 : Nat) / 0 = 0 := by decide      -- accepted!
```

is accepted, and a theorem about `a / b` that forgets the hypothesis `b ≠ 0` can be certified and useless. So read three things, every time:

1. the **statement**: its hypotheses and its definitions (what exactly is `cdiv`? what is `p`?);
2. the **axiom footprint**;
3. the **model gap**: Brent's bound is about unit-cost DAGs, and a real cluster adds latency, bandwidth (U0's α and β) and failures.

The checker removes doubt about the proof; only you can remove doubt about the statement.

8. The course's own certificate, read line by line: Brent's bound

```
def cdiv (a p : Nat) : Nat := (a + (p - 1)) / p          -- ⌈a / p⌉ in natural
  numbers

theorem bmls_parallel_bound (m : List Nat) (p : Nat) (hp : 1 ≤ p) :
    p * (m.map (fun mi => cdiv mi p)).sum ≤ m.sum + m.length * (p - 1) := by
  induction m with
  | nil => simp                                    -- no levels: 0 ≤ 0
  | cons a t ih =>                                 -- one more level a in front of
    the list t
    have hdiv : p * cdiv a p ≤ a + (p - 1) := by
      unfold cdiv; exact Nat.mul_div_le (a + (p - 1)) p      -- p·⌊x/p⌋ ≤ x
    simp only [List.map_cons, List.sum_cons, List.length_cons, Nat.mul_add,
      Nat.add_mul, Nat.one_mul]                    -- expand the sums for a :: t
    omega                                          -- linear arithmetic closes it,
      using ih and hdiv
```

(The course file ***provedog/IasCert/Paper/BMLSPar.lean*** *is built with Mathlib and writes* $\mathbb{N}$*, Mathlib's name for* ***Nat****. In the Lab 0 image, which is core Lean 4, write* ***Nat****.)*

Reading the statement. `m` is the list of level widths $m_1, \ldots, m_D$ of the DAG, so `m.length` $= D$ and `m.sum` $= W$. `cdiv mi p` $= \lceil m_i/p \rceil$ is the number of steps that level takes on p processors, so the mapped sum is the time T_p of the level-by-level schedule; that this models the DAG's run is modelling, not Lean. **Certified:** $p \cdot T_p \le W + D(p-1)$. **On paper, not in Lean:** divide by p to get $T_p \le D + \frac{W-D}{p} \le \frac{W}{p} + D$. It is a $\forall$-statement, for every list and every $p \ge 1$: infinitely many structures (§4.3), so it needs a proof, not tests.

Reading the proof. It is induction on the list, the recursor of the inductive type `List` (§5.3). The `nil` case is the base case: an empty DAG takes no time. The `cons` case is the step: the new level contributes $p\lceil a/p \rceil \le a + (p-1)$ (`hdiv`, from the library lemma $p\lfloor x/p \rfloor \le x$), and `ih` covers the rest of the list; adding them gives the bound for the longer list, and `omega` closes the linear arithmetic. This is the level-by-level argument of U0's Brent proof, now checked by a machine.

Measured in the Lab 0 image (core Lean 4, with `Nat` for $\mathbb{N}$): the proof is accepted, and `#print axioms` gives `[propext, Quot.sound]`, two of the classical trio and no `sorryAx`. Lean's

linter also warns `Variable name `hp` is not explicitly referenced`: the hypothesis `hp` is not needed, because `cdiv a 0 = 0`.

A worked use (U0P §6.1). A pipeline with work $T_1 = 400$ machine-hours and critical path $T_\infty = 8$ hours on $p = 64$ nodes has $T_{64} \le T_\infty + (T_1 - T_\infty)/64 = 8 + 392/64 \approx 14.1$ hours. By the easy lower bound $T_p \ge T_\infty$ (not part of the certificate), no number of nodes gets it below 8 hours. If the bound were wrong, or missing a side condition, a purchase decision built on it would be wrong too; the certificate removes that doubt about the proof, and §7.2 tells you what doubt remains.

Where language models fit. A language model may draft a proof or suggest a lemma or tactic: authoring help, optional and judged separately. The kernel is the only judge, so a drafted proof is accepted exactly when it type-checks, and a wrong draft is simply rejected. The course can therefore use AI help without trusting it: the trust sits in a small checker you can run offline, not in the model, the vendor or the author. This is the course's general pattern, **verification, not assertion**.

9. Lean walkthroughs: logic and finite models, by hand and by machine

Each walkthrough takes an idea from §§3–4, does it on paper, then does it in Lean, line by line. All of them are core Lean 4 and run in the Lab 0 image (see *Running your own file* at the end of this section).

9.1 Model checking a finite structure, in Lean

§4.1 model-checked two sentences in $\mathcal{A} = (\{0,1,2\}, <)$ by hand. In Lean the domain $\{0,1,2\}$ is the type `Fin 3`, whose elements are `0`, `1` and `2`, with the usual order. The two sentences become:

```
-- "there is no largest element" is FALSE in ({0,1,2}, <)
example : ¬ (∀ x : Fin 3, ∃ y : Fin 3, x < y) := by decide

-- "there is a least element" is TRUE in ({0,1,2}, <)
example : ∃ x : Fin 3, ∀ y : Fin 3, ¬ (y < x) := by decide
```

Line by line. `Fin 3` is a finite type, so a quantifier over it ranges over three values. Lean has `Decidable` instances for ∀ and ∃ over `Fin n`, for `<` on `Fin n`, and for `¬`, so each proposition has a program that computes its truth value; `decide` runs that program and turns the answer `true` into a proof (§6.2). This is exactly the loop of §4.1: for the first sentence, for each of the three `x` it tries the three `y`, and finds `x = 2` with no `y`. The certificate records the result of a finite computation. It says nothing about any infinite order, and in ℕ the first sentence is true (§4.1).

9.2 Witnesses and negation: proofs you build by hand

An existential statement is proved by giving a witness and a proof for it (§5.1); a negation `¬A` is a function from proofs of `A` into `False`:

```
example : ∃ n : Nat, n + n = 6 := ⟨3, rfl⟩

theorem my_not_and_not (A : Prop) : ¬ (A ∧ ¬ A) :=
  fun ⟨a, na⟩ => na a
```

Line by line. In the first, `⟨3, rfl⟩` is the pair (witness `3`, proof that `3 + 3 = 6`), and `rfl` works because `3 + 3` computes to `6` (§5.5). In the second, the goal `¬ (A ∧ ¬ A)` unfolds to `A ∧ ¬ A → False`. So the proof is a function: it takes the pair apart into `a : A` and `na : ¬ A`,

that is `na : A → False`, and applies `na` to `a` to get `False`. This is the natural-deduction proof "assume A and $\neg A$; by $\Rightarrow$-elimination, $\bot$", and it uses no axioms: it is valid constructively.

9.3 A recursive function and a proof that it is correct

Define doubling by structural recursion (§5.3), then prove that it agrees with multiplication by 2:

```
def double : Nat → Nat
  | 0     => 0
  | n + 1 => double n + 2

theorem double_eq (n : Nat) : double n = 2 * n := by
  induction n with
  | zero      => rfl
  | succ k ih => simp only [double, ih]; omega
```

Line by line. `double` has one case per constructor of `Nat`, and the recursive call is on the smaller `n`, so Lean accepts that it terminates. In the proof, the `zero` goal is `double 0 = 2 * 0`, and both sides compute to `0`, so `rfl` closes it. The `succ` goal is `double (k + 1) = 2 * (k + 1)`. `simp only [double, ih]` rewrites with the defining equation `double (k + 1) = double k + 2` and then with `ih : double k = 2 * k`, leaving `2 * k + 2 = 2 * (k + 1)`, which is linear arithmetic, so `omega` closes it. The same pattern (unfold the definition, use the hypothesis, finish with arithmetic) proves most of the cost recurrences in the units.

9.4 Induction on lists

Lists are an inductive type with two constructors, `[]` (nil) and `x :: t` (cons), so they have an induction principle with two cases, exactly as `Nat` does. The length of a concatenation:

```
theorem my_length_append (xs ys : List Nat) :
    (xs ++ ys).length = xs.length + ys.length := by
  induction xs with
  | nil => simp
  | cons x t ih => simp only [List.cons_append, List.length_cons, ih]; omega
```

Line by line. In the `nil` case the goal is `([] ++ ys).length = [].length + ys.length`, and `simp` closes it with the library facts `[] ++ ys = ys` and `[].length = 0`. In the `cons` case the goal is `(x :: t ++ ys).length = (x :: t).length + ys.length`. The library lemma `List.cons_append` says `x :: t ++ ys = x :: (t ++ ys)`, and `List.length_cons` says a cons is one longer than its tail. So the goal becomes `(t ++ ys).length + 1 = t.length + 1 + ys.length`, and `ih : (t ++ ys).length = t.length + ys.length` turns it into `t.length + ys.length + 1 = t.length + 1 + ys.length`, which `omega` closes. This is the shape of the Brent proof in §8: induction on the list of levels, then linear arithmetic.

9.5 Disproving a claim: a counterexample is a proof too

To show that a universal claim is **false**, prove its negation, and the natural proof is a counterexample. The claim "$n \cdot n \le 81$ for every n" survives every test with $n < 10$, then fails at $n = 10$:

```
example : ∀ n : Nat, n < 10 → n * n ≤ 81 := by decide      -- ten tests pass

example : ¬ (∀ n : Nat, n * n ≤ 81) := by
  intro h
```

```
  have h10 := h 10
  omega
```

Line by line. The first line is model checking of the bounded claim (§6.2). In the second, `intro h` assumes the false claim, `h 10` specialises it to `10 * 10 ≤ 81`, and `omega` sees that `10 * 10` is the literal `100` and that `100 ≤ 81` is impossible, so it closes the goal `False`. This is §4.3 in one example: finitely many tests passed, and one more instance refutes the claim.

9.6 Reading a statement's definitions before trusting it

```
example : ∀ a b : Nat, a / b ≤ a := by
  intro a b
  exact Nat.div_le_self a b
```

The library lemma `Nat.div_le_self` proves it for every `b`, including `b = 0`. But at `b = 0` it holds only because `a / 0 = 0` in Lean (§7.2); as the division you meant, `a / 0` is undefined, so the certificate says nothing there. Before relying on a statement, read its definitions and ask what it says at the edges.

Running your own file. Put your proof in a file `Try.lean` in the Lab 0 folder and use the same image:

```
docker run --rm --user "$(id -u):$(id -g)" -e HOME=/tmp -v "$PWD:/lab" -w /lab \
  vakeworks/lean-checker:v4.32.0 lean Try.lean
```

No output means accepted; an error means rejected; a `declaration uses `sorry`` warning means **not proved**, even though `lean` exits 0; other warnings, such as an unused variable, are style hints and the proof is still accepted.

10. Exercises

Worked solutions are in the instructor edition (*Solutions fc3*). Each Lean exercise runs in the Lab 0 image; a Lean answer counts only when it is accepted with no `sorry` (§7.1).

Light (examined core):

EF3.1 *(Free and bound.)* For each formula, list the free variables and say whether it is a sentence: (a) $\forall x\,(x < y \Rightarrow \exists z\,(x < z \wedge z < y))$; (b) $\exists x\,\forall y\,\neg(y < x)$; (c) $\forall x\,(x = 0) \vee x < S(0)$.

EF3.2 *(Model checking by hand.)* Let $\mathcal{B} = (\{0,1,2,3\}, <)$. Decide, with a one-line reason each: (a) $\forall x\,\exists y\,(y < x \vee x = 0)$; (b) $\forall x\,\forall y\,(x < y \Rightarrow \exists z\,(x < z \wedge z < y))$ ("the order is dense"); (c) $\exists x\,\exists y\,(x < y \wedge \forall z\,\neg(x < z \wedge z < y))$.

EF3.3 *(Soundness at work.)* Using only soundness (§3.4), show: if a theory T has a model, then T is consistent. Then explain in two sentences why this gives the standard way to prove that a sentence φ is *not* provable from T.

EF3.4 *(Logic in Lean.)* Prove, in Lean, the contrapositive rule: `theorem my_contra (A B : Prop) : (A → B) → ¬ B → ¬ A`. Then say which natural-deduction rules your term uses.

EF3.5 *(Model checking in Lean.)* In `Fin 4` with `<`, state and prove with `decide` that every element other than `0` has a smaller element. Then state the same sentence for `Nat` and say why `decide` cannot be used there, and prove it anyway (hint: for `x ≠ 0`, the witness `0` works).

EF3.6 *(Induction on lists.)* Prove `theorem my_length_map (f : Nat → Nat) (xs : List Nat) : (xs.map f).length = xs.length` by induction on `xs`, in the style of §9.4.

Deep:

EF3.7 *(Only infinite models.)* Let ψ be the conjunction of the axioms of a strict linear order (irreflexive, transitive, total) with $\forall x\, \exists y\, (x < y)$. Prove on paper that ψ has no finite model but has an infinite one. Explain why this means no amount of model checking on finite structures can establish ψ's satisfiability, and relate it to Trakhtenbrot's theorem (§4.2).

EF3.8 *(An Ehrenfeucht–Fraïssé game.)* Work in the empty signature (only $=$). Show that the second player wins the 2-round Ehrenfeucht–Fraïssé game between a 2-element set and a 3-element set. Conclude that no sentence of quantifier depth 2 is true in exactly one of them. Then give a sentence of depth 3 that separates them.

EF3.9 *(Testing, then refuting, in Lean.)* The claim "every n satisfies $n \cdot n < 3n + 10$" passes some tests. (a) Find, by hand, the smallest n for which it fails. (b) In Lean, prove the bounded version for all n below that value with `decide`. (c) Prove the negation of the universal claim, in the style of §9.5.

EF3.10 *(Read the statement.)* A colleague's certified theorem is `theorem avg_le (a b : Nat) : (a + b) / 2 ≤ a + b`. (a) Is it correct, and does it say what "the average is at most the sum" means for natural numbers? (b) Their next claim is `∀ a b : Nat, (a + b) / 2 ≤ a`. Prove in Lean that this claim is false.

Sources

- **Logic.** H. B. Enderton, *A Mathematical Introduction to Logic*, 2nd ed., Academic Press, 2001 (§§3.1–3.6). K. Gödel (1930), Die Vollständigkeit der Axiome des logischen Funktionenkalküls, *Monatshefte für Mathematik und Physik* 37, 349–360; K. Gödel (1931), Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme I, *Monatshefte für Mathematik und Physik* 38, 173–198. L. Henkin (1949), The completeness of the first-order functional calculus, *Journal of Symbolic Logic* 14, 159–166.
- **Finite model theory.** L. Libkin, *Elements of Finite Model Theory*, Springer, 2004 (§§4.1–4.2: model checking and its complexity, Codd's theorem, the failure of compactness, Trakhtenbrot's theorem, Ehrenfeucht–Fraïssé games, the zero–one law). B. A. Trakhtenbrot (1950), The impossibility of an algorithm for the decision problem for finite models, *Doklady Akademii Nauk SSSR* 70, 569–572.
- **Lean.** J. Avigad, L. de Moura, S. Kong and S. Ullrich, *Theorem Proving in Lean 4* (open access): `https://lean-lang.org/theorem_proving_in_lean4/`. M. Carneiro (2019), *The Type Theory of Lean*, MSc thesis, Carnegie Mellon University (§5.4).
- **Proof checkers.** N. G. de Bruijn (1980), A survey of the project AUTOMATH, in J. P. Seldin and J. R. Hindley (eds.), *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, Academic Press (§6.1).
- **The other results named in the text** (Church 1936, Turing 1936, Rosser 1936, Presburger 1929, Stockmeyer 1974, Vardi 1982, Codd 1972, Glebskii et al. 1969, Fagin 1976, Ehrenfeucht 1961, Fraïssé 1954) are treated, with their original references, in Enderton (logic) and Libkin (finite model theory) above. Diaconescu's theorem (§5.1) is in neither book; its source is Diaconescu (1975).

Where this is used

U0P (the proof-checker unit) directly; every unit that names a Lean cert asks you to read one. Units with a `depends F.C3` edge lean on "you can read a machine-checked claim." Module 1's lecture on Lean, logic and proof checkers is a condensed version of §§3–9.

Chapter 37

Economic preliminaries

F.X1 — Utility, incentives, and what a contract is

Memory jog for: U16 (statistical contract theory — this chapter is its vocabulary), U00 §2 (the day-one lens), U20 (whose guarantee is being merged?), U18 (the alignment-as-contract remark). No economics background is assumed; this is the fourth pillar's minimum kit, kept deliberately smaller than the other pillars' — U16 builds the rest when it needs it.

1. Preferences and utility

An agent's **preferences** over outcomes are summarized by a **utility function** u: outcome a is chosen over b when $u(a) > u(b)$. Under uncertainty, the standard model compares **expected utilities**: lottery P beats Q when $\mathbb{E}_P[u] > \mathbb{E}_Q[u]$. Everything probabilistic from F.S1 imports unchanged — expected utility is just an expectation, and linearity (F.S1 §2) is as load-bearing here as anywhere.

The modelling commitment worth making explicit: **utility is the analyst's bookkeeping for choice, not a claim about inner life.** We use it for firms, labs, platforms, and machine agents alike; nothing in the formalism needs anyone to *feel* anything. (This book applies the same discipline to its own agents: behaviour and artifacts, not attributed mental states.)

2. Incentives — why data is not neutral

The pillar-4 observation (U00 §1): data is produced, shared, and acted on by **self-interested parties**. Whenever the reporter of a number gains from your believing a particular value, the number's distribution is an equilibrium object, not a law of nature. Three book-relevant instances:

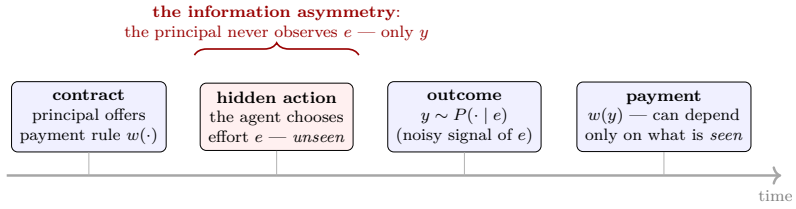
1. A vendor reports its model's accuracy; buyers cannot rerun the evaluation.
2. Federated clients (U19/U20) report local gradients or estimates; each bears the cost of honest computation and may prefer shortcuts.
3. A benchmark's authors also compete on it.

The statistical toolkit of F.S1–F.S3 assumes samples arrive from a fixed p; incentive analysis asks **which p a self-interested party chooses to let you see**. That single question is the gap between Pillar 2 and Pillar 4, and closing it rigorously is U16's job.

3. The principal–agent frame, and what a contract is

The minimal cast: a **principal** wants a task done (train a model, report an estimate); an **agent** does it, with private effort and private information; their interests are not aligned. A **contract** is a commitment, fixed *before* the work, mapping **observable, verifiable outcomes to payments** (or penalties, access, reputation). Good contracts make the agent's selfish best response coincide with what the principal wanted — **incentive compatibility** — subject to the agent being willing to sign at all (**participation**). Information the agent holds privately earns it a rent; the classical treatment of these trade-offs is the principal–agent literature cited in U16's sources.

The design insight that survives into U16: **you can only contract on what you can verify.** Vague quality is uncontractible; a statistic with a known distribution under stated hypotheses is contractible. This is where the fourth pillar reaches back into the second — and where this book's proof-checker ethos (U0P) reappears in economic clothing: a machine-checkable certificate is the strongest possible contractible object. Figure 37.1 lays the problem on one timeline: payment can see the outcome y, never the effort e.



the whole of contract theory in one question: choose $w(\cdot)$ so that the agent, maximising its *own* expected utility $\mathbb{E}[u(w(y))] - c(e)$, *chooses the effort the principal wanted* (incentive compatibility) — while still preferring to participate at all (individual rationality)

Figure 37.1: The principal–agent problem on one timeline. Payment can only condition on the observable outcome y, never on the hidden effort e — that asymmetry (red) is the entire subject. A contract $w(\cdot)$ is *incentive-compatible* when the agent's own optimisation lands on the effort the principal wanted. Statistical contract theory (U16) prices this with the course's estimation tools; U17 deploys it on machine agents: read the agent as an ML model or LLM agent and y as its reported metrics; the σ-chain makes a changed record *detectable*, and the contract makes honesty *optimal*.

3.1 Worked: a bonus paid on the agent's own number

The setting.

- *Principal:* a company's IT department. It wants legitimate mail delivered and spam stopped.
- *Agent:* the vendor that runs the filter. It holds private information: it sees every message it quarantines, while the principal sees only what its users complain about.
- *Contract:* €2,000 per month, plus a €1,000 bonus if the false-positive rate is below 0.1%, *as reported by the agent.*

The numbers.

```
mail per month                          1,000,000   (700,000 of it legitimate)
legitimate mail wrongly quarantined         7,000   = 1.0%  of legitimate mail
  (the true rate)
of those, users notice and complain             350   = 5%    of the 7,000
the agent's report: complaints / legit          350 / 700,000 = 0.05% < 0.1%  ->
  bonus paid
```

The outcome the bonus is paid on is the agent's own report, which the principal cannot verify. So the agent's best response is not "make fewer errors" but "report fewer errors": the contract is not incentive compatible. The agent keeps a rent of €1,000 a month from its private information; the principal loses 7,000 legitimate messages a month; no rule was broken.

The fix is in the contract, not the model. Pay the bonus on a *verifiable* signal that the principal draws itself: an audit.

1. Draw a random sample of incoming mail and have the principal label it, until 10,000 legitimate messages are in hand.
2. Suppose 100 of them had been quarantined. Then $\hat{p} = 1.0\%$, and the 95% normal-approximation interval is

$$\hat{p} \pm 1.96\sqrt{\hat{p}(1-\hat{p})/n} \;=\; [0.80\%,\ 1.20\%],$$

   far above 0.1%, so no bonus.
3. Now reporting fewer errors earns nothing, and only making fewer errors pays: the contract is incentive compatible.

The agent still signs as long as €2,000 covers its cost of running the filter; that is the participation constraint. Whoever produces the number must not be the one paid on it. This is the X pillar, priced with the S pillar's normal-approximation interval (a central-limit approximation, F.S1 §5; it is not one of F.S2's bounds, and E.FX1.1 replaces it with F.S2's Hoeffding bound).

4. Statistical contracts — the one-sentence preview of U16

A **statistical contract** ties payment to a *statistical* outcome — e.g., "the error-rate certificate, computed by this agreed test on held-out data, falls below ε" — so the guarantee attached to a prediction becomes something a counterparty can collect on. U16 develops the mathematics (its E-value theorem gives anytime-valid evidence a wallet: accumulated evidence bounds what the counterparty can lose), and the σ-log lab (Part F's floor-lab) is its operational sibling: an append-only attested trail is what makes *process* claims contractible, the same way a held-out test makes *accuracy* claims contractible.

4.1 The everyday face: pricing and "AI product management"

The same vocabulary is what a team actually reaches for when it turns a model into a *product*. The reservation value and payment of §3 are a **price**; the verifiable statistic of §4 is a **service-level guarantee** ("sets contain the truth at least $1-\alpha$ of the time"); the choice to let the payment respond to accumulated evidence is **dynamic pricing**. So the part of "AI product management" that is genuinely hard — deciding what to promise a buyer, how to price it, and how to contract for the data it runs on — is not a separate business skill sitting outside the mathematics; it *is* the principal–agent problem of this chapter, made concrete. U16 §5.3 develops this reading; here the only thing to carry forward is that a product decision about a model is a choice of *what is contractible and at what price*.

5. One exercise

E.FX1.1 A vendor's model has true error θ known only to the vendor. A buyer offers: "run the agreed test on n fresh samples; if the observed error $\hat{\theta} \le \varepsilon$, I pay V; otherwise nothing." Using Hoeffding (F.S2 §3), bound the probability a vendor with $\theta \ge \varepsilon + \delta$ collects V, and the probability an honest vendor with $\theta \le \varepsilon - \delta$ is unpaid. What does increasing n buy each party? (This ten-line calculation is a working statistical contract — U16 §3.4 works a threshold-test contract of the same shape.)

Chapter 38

The engineering floor

F.E1 — Linux and the shell

*Foundation chapter (the Engineering Floor). A root: every lab runs commands in a shell, and F.E1b (git) and F.E2 (docker) build directly on it. A **memory-jog** of the mental model and the dozen commands you actually use — nothing more.*

1. The mental model: a tree of files, one process at a time

The filesystem is a single tree rooted at `/`. A **path** names a node: absolute (`/home/you/work`) or relative to your **current working directory** (`.` is here, `..` is up, `~` is your home). Everything is a file — programs, devices, even a running process's handles. The shell is just a program that reads a command, launches another program with the arguments you gave, and waits.

2. The commands you actually use

```
pwd                     # where am I?
ls -la                  # list here (long, incl. hidden)
cd <dir>                # go there
cat <file>              # print a file
less <file>             # page through a file
grep <pat> <file>       # lines matching a pattern
find <dir> -name X      # locate files by name
wc -l <file>            # count lines
cp a b                  # copy a to b
mv a b                  # move / rename a to b
rm a                    # delete a
mkdir -p <dir>          # make a directory (and parents)
chmod +x <script>       # make a script runnable
./script.sh             # run a script in this dir
bash script.sh          # run it with bash
```

Two habits: read `--help` (or `man <cmd>`) before guessing, and quote paths with spaces (`"my file"`).

Five more that pay for themselves on the first long job:

```
ssh user@host                       # a shell on another machine (keys: F.E1b
  §2)
tmux new -s job                     # a session that survives disconnects;
  detach with Ctrl-b d
```

```
tmux attach -t job                    # reattach later (the job kept running
  while your laptop slept)
awk -F, '{s += $3} END {print s}' f.csv   # column arithmetic on text: sum the
  third field
sed -n '10,20p' file                  # print lines 10-20;  sed 's/old/new/g'
  substitutes
jq '.sha256' line.json                # pull one field out of JSON
```

3. Pipes and redirection — everything is a stream

Each program has three streams: **stdin** (0), **stdout** (1), **stderr** (2). You wire them:

```
cmd > out.txt           # send stdout to a file (overwrite)
cmd >> out.txt          # append stdout to a file
cmd < in.txt            # read stdin from a file
cmd 2> err.txt          # send stderr to a file
cmd1 | cmd2             # pipe: cmd1's stdout becomes cmd2's stdin
grep err log | wc -l    # compose small tools into a bigger one
```

`MATHS->CS:` a pipeline is *function composition* on streams — `cmd2` $\circ$ `cmd1`. `CS->MATHS:` this is the shell's whole philosophy: small programs that each do one thing, composed. It is also, in miniature, the map→shuffle→ reduce shape (F.C1) — data flowing through stages.

Where this is used

Every lab. F.E1b (git) and F.E2 (docker) are the next two floors up; the units never re-teach the shell but every `docker run` / `git commit` / lab script assumes it.

F.E1b — Git, SSH keys, and a sovereign git server

*Foundation chapter (the Engineering Floor). Prereq: F.E1 (shell). A **how-to memory-jog**: the version-control basics every project uses, how to authenticate with a key, and how to keep your own history sovereign while still reaching every forge. The σ-log-replay lab (a later engineering-floor lab) stands on this chapter.*

1. Git in ten commands

Git is a content-addressed history of your files: each **commit** is a snapshot plus a pointer to its parent, so the history is an append-only chain you can audit and replay.

```
git init                      # start a repository here
git add <path>                # stage changes for the next commit
git commit -m "message"       # record a snapshot (with a message)
git log --oneline --graph     # read the history
git branch <name>             # make a line of development
git checkout <name>           # switch to it   (or: git switch <name>)
git merge <name>              # fold another branch into this one
git clone <url>               # copy a remote repository locally
git pull                      # fetch + merge the remote's new commits
git push                      # send your commits to the remote
```

Two habits that matter for everything downstream: **commit small and often** (each commit is one reviewable step), and **write real messages** (the log is the code's memory — see the σ-log-replay lab).

Two more for working with others. `git rebase <base>` replays your commits on top of an updated base so the history stays linear; never rebase commits that others have already pulled. `git diff <base>...HEAD` shows exactly what your branch changes; read it yourself before asking anyone else to review it.

2. SSH keys — authenticating to a forge

A forge (a git host) authenticates you by a **key pair**: a private key that never leaves your machine and a public key you upload. Generate one, then add the public half to each forge:

```
ssh-keygen -t ed25519 -C "you@example.org"    # creates ~/.ssh/id_ed25519(.pub)
cat ~/.ssh/id_ed25519.pub                       # copy this PUBLIC line to the
  forge
ssh -T git@<forge-host>                         # test (e.g. github.com,
  gitlab.com, codeberg.org)
```

Upload the `.pub` file's contents under the forge's *SSH keys* settings — the same public key works on **gitlab**, **github**, and **codeberg** at once. Never share or upload the private key. Then clone/push over SSH: `git clone git@<forge-host>:<owner>/<repo>.git`.

3. Keeping your history sovereign — own the centre, mirror outward

The forges above are convenient but each is *governed from some jurisdiction*, and access or data can be reached from above (sanctions, cutoffs, extraterritorial demands). The design that keeps the reach without the dependency: **make your own server the authoritative origin, and push-mirror to the public forges** for discoverability and collaboration.

- **Authoritative remote = your server.** A git repository is just a directory; a *bare* repo on a small host is a complete origin: `git init --bare` on the server, then `git remote add origin ssh://you@your-host/srv/git/<repo>.git`. Your full history now lives on a machine you govern.
- **Mirror outward.** Add the public forges as additional push targets so every push fans out. Note that once a remote has any push URL, `git push` goes *only* to its push URLs, so register your own server as the first one: `git remote set-url --add --push origin ssh://you@your-host/srv/git/<repo>.git`, then `git remote set-url --add --push origin <forge-url>` (repeat per forge — github, gitlab, codeberg, or a self-hosted forge). Collaborators still open issues and PRs on whichever forge they use; the source of truth stays home.
- **Why it is cheap.** The centre is a bare repo on a host that costs about a coffee a month; the reach is a few lines of remote configuration. Sovereignty at the centre, reach everywhere — the full argument and the exact scripts are in the sovereign git server itself: https://git.vake.works/OpenEarthNetwork/sovereign-git-server (the sovereign centre, hosted on our own `git.vake.works`; mirrored for reach at https://github.com/OpenEarthNetwork/sovereign-git-server — sovereign at the centre, reach everywhere).

Where this is used

Every lab is delivered as a git repository you clone; every project's provenance is its commit trail. This chapter + the σ-log-replay lab are how a project *proves who did what* without any agent in the loop — the contribution record is the replayed history itself.

F.E2 — Docker: the reproducible container

*Foundation chapter (the Engineering Floor). Prereq: F.E1 (shell). A **memory-jog**: what a container is and the handful of commands every lab uses. The whole point is one guarantee* — if it runs in the container here, it runs in the container there* — which is why every lab in this book ships as an image you run, not a setup you fight.*

Install it now (one-time). Before Lab 0 you need Docker on your laptop; use the official installers — macOS: Docker Desktop (docs.docker.com/desktop/setup/install/mac-install); Linux: Docker Engine (docs.docker.com/engine/install) plus the post-install step so `docker` runs without `sudo`; Windows: WSL2 first (learn.microsoft.com/windows/wsl/install), then Docker Desktop with the WSL 2 backend (docs.docker.com/desktop/setup/install/windows-install), and do all course work inside the Ubuntu (WSL2) terminal. The labs' `README.md` walks each path step by step; verify with `docker run --rm hello-world`. The course images are multi-architecture — the same pull works on Intel/AMD and Apple-Silicon/ARM machines.

1. Image vs container

- An **image** is a frozen, layered filesystem plus a default command: a build's-worth of software, pinned. It is *content-addressed* (an immutable digest), so "the same image" means bit-for-bit the same environment.
- A **container** is a *running instance* of an image — an isolated process with its own filesystem view, network, and resource limits, sharing the host kernel (lighter than a virtual machine, which ships a whole OS).

`MATHS->CS:` a container is to an image what a process is to a program. `CS->MATHS:` reproducibility is *determinism up to the pinned image* — the experiment's environment is a fixed point, not a moving target.

2. The commands you actually use

```
docker build -t <name> .              # build an image from a Dockerfile in this
  directory
docker run --rm -it <name>            # run a container (--rm: clean up after;
  -it: interactive)
docker run -v "$PWD:/work" -w /work <name> <cmd>   # mount the current dir; run a
  command in it
docker run -p 8080:8080 <name>        # publish a container port to the host
docker compose up                      # bring up a multi-container app from
  compose.yaml
docker compose down                    # tear it down
```

Two patterns the labs lean on: **bind-mount your work** (`-v host:container`) so edits on the host are seen inside, and **compose** to wire several services (a compute engine + a store + a database) into one laptop-runnable stack.

3. Why it is the lab substrate

- **No "works on my machine."** The pinned image *is* the environment; a grader runs the same image and gets the same result. (This is exactly how the proof-checker lab ships — a pinned image, no local toolchain install; the verdict is a build exit code, no AI in the loop. See F.C3 / the proof-checker lab.)
- **Free + sovereign.** The engine and base images are open; the only cost is your own machine. Nothing forces a paid platform to complete a lab.

- **Bridges to scale (F.E3).** The same container that runs on your laptop is the unit a cluster schedules across many machines — docker is the atom, kubernetes (F.E3) is the orchestration.

Where this is used

Every lab is delivered as a container (`docker run` / `docker compose up`); the graders run the same image. Units with a `depends F.E2` edge assume this chapter; F.E3 puts these containers on a cluster.

F.E3 — Kubernetes: containers on a cluster

*Foundation chapter (the Engineering Floor). Prereq: F.E2 (docker). A **memory-jog**: what an orchestrator does and the handful of objects you meet. The concrete, from-scratch build (three machines, one file, one command, warts kept) is the Infrastructure deep-dive lab — this chapter is the map to it, not the territory.*

1. What it is

A single container (F.E2) runs on one machine. When a workload needs *many* machines — for capacity, for fault-tolerance, or because the data is partitioned (F.C1's DPRAM) — something must decide *which container runs where*, restart failures, and route traffic. That is an **orchestrator**; kubernetes is the common one.

You do not tell it *how*; you declare the **desired state** ("I want three replicas of this image, reachable on this port") and a control loop continuously **reconciles** reality toward it. `MATHS->CS:` it is a fixed-point iteration on cluster state — the same "iterate until you reach the invariant" shape as the algorithms in the units.

2. The objects you meet

- **Node** — a machine in the cluster (one control-plane node schedules; worker nodes run the work).
- **Pod** — the smallest deployable unit: one (or a few tightly-coupled) containers scheduled together.
- **Deployment** — a declared desired count of identical pods; the controller keeps that many alive (self-heals, rolls updates).
- **Service** — a stable address in front of a changing set of pods (the pods come and go; the service name does not).

Everything is a declarative object you `kubectl apply`; `kubectl get nodes/pods` reads the reconciled reality.

3. Two hard constraints the deep lab holds (and why they matter)

The full walkthrough stands the whole thing up on rented machines, and by design it is:

1. **LLM-free** — every step is a concrete command with its explanation, runnable from the page with no AI present (the same discipline as the proof-checker and σ-log labs).
2. **Forced-payment-free** — the tooling (an infrastructure-as-code CLI, `kubectl`, a lightweight distribution) is open; the only cost is the few euros of compute *you* choose to rent and can hand back afterward.

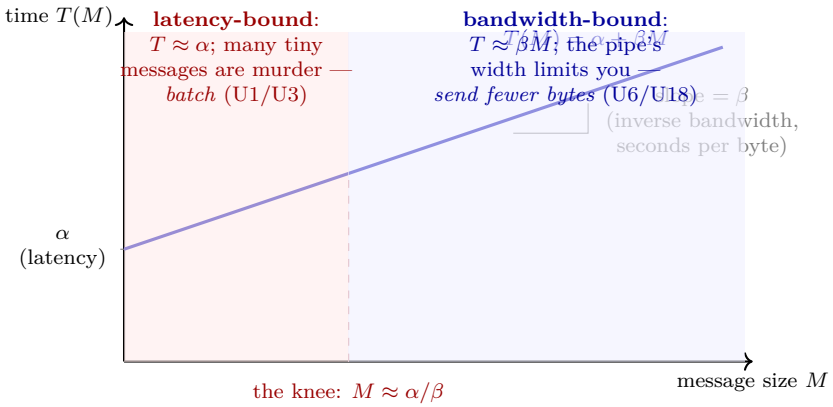


Figure 38.1: The α–β model in one line and two regimes. Every message pays a fixed latency α plus β seconds per byte; the knee at $M \approx \alpha/\beta$ (typically tens of kilobytes) splits the world. Left of it, latency dominates and the winning move is *fewer, batched* messages; right of it, bandwidth dominates and the winning move is *fewer bytes*. You measure your own α and β in the §2 ping-pong lab — and reuse them to predict tokens/sec in U0L.

That double constraint — sovereign and zero-forced-cost — is the reason a cluster is *reproducible from the page alone*: the record of how it was built is the build.

Where this is used

The distributed labs (a compute engine, a store, a database across several machines) schedule the F.E2 containers here; F.E5 (storage) and F.E6 (runtimes) are the payloads that run on this substrate. Units with a `depends F.E3` edge reach for the cluster; most only need F.E2.

F.E4 — The network is physical: the α-β cost model

Foundation chapter (the Engineering Floor). Prereq intuition: F.M1 (asymptotics). This is the measured twin of U0's communication term: the units account* for communication, and here is *why the numbers look the way they do.* Panel c of the U0 triptych — the dashed wire between machines — is this chapter.*

1. The model: latency plus size-over-bandwidth

Sending a message of M bytes between two machines costs, to first order,

$$T(M) \;=\; \alpha \;+\; \beta\, M,$$

where α is the **latency** (a fixed per-message cost — setup, propagation, software overhead) and β is the **inverse bandwidth** (seconds per byte). Two regimes:

- **small M: latency-bound.** $T \approx \alpha$; the message size barely matters. Many tiny messages are murder — you pay α every time.
- **large M: bandwidth-bound.** $T \approx \beta M$; you are limited by the pipe's width.

The crossover — the **knee** — is at $M \approx \alpha/\beta$: below it, latency dominates; above it, bandwidth does. Typical orders of magnitude have α in microseconds and β small, so the knee sits at tens of kilobytes — which is why systems *batch*.

2. The experiment (ping-pong) — measure your own α and β

Bounce a message of size M back and forth k times between two nodes and time it; the per-message cost is the total over $2k$. Sweep M across powers of two and plot T against M:

- the **intercept** at $M \to 0$ is α;
- the **slope** in the large-M tail is β;
- the bend between them is the knee.

This is a lab you can run on two containers (F.E2) or two cluster nodes (F.E3). You do not take α, β on faith — you measure the machine in front of you.

2.1 What α and β are, precisely, and a worked example

Latency α is the time for a message of zero length: $\alpha = T(0)$, in seconds per message. It runs from the sender's call to send until the receiver has the message, and it does not depend on M. It adds up the software overhead at both ends (system call, copies, protocol stack), the network interfaces and switch hops, and the propagation delay of the wire. **Inverse bandwidth** $\beta = 1/\text{bandwidth}$ is the extra time per byte, in seconds per byte.

Reading them off a measurement. Time messages of several sizes M, take the one-way time as half the ping-pong round trip, and fit a straight line to T against M: the intercept is α, the slope is β. In practice the smallest sizes often bend away from the line (packetisation, protocol thresholds), which is why §2 reads the slope from the large-M tail.

Worked (illustrative numbers, not a measurement). Take $\alpha = 50\,\mu s$ and $\beta = 1$ ns/byte (1 GB/s).

- *A small message.* One 8-byte partial sum costs $50\,\mu s + 8\,\text{ns} = 50.008\,\mu s \approx \alpha$: latency-bound.
- *The knee.* It sits at $M = \alpha/\beta = 50{,}000$ bytes, i.e. 50 KB. There the two terms are equal and $T = 2\alpha$. Below it latency dominates; above it bandwidth does.

How one round of many parallel messages is priced is U0 §4.1.

3. Why it changes algorithm design

Because α is paid per message, the game is **fewer, bigger messages**:

- **aggregate** before you send (combine locally, then one message — the reduce of U1);
- **all-reduce as a tree**, not all-to-one: $\log p$ rounds instead of p, each message near the bandwidth-bound size (U1 → U18's ring/tree all-reduce);
- a **badly-sharded all-to-all shuffle** sends many small messages — the latency term explodes (U5's shuffle cost). "Communication-avoiding" (U6) is exactly: restructure the algorithm so M is large and messages are few.

`MATHS->CS:` U0's communication term is $\alpha R + \beta B_{\text{crit}}$, with R the round count and B_{crit} the bytes on the critical path (the slowest message of each round, summed over rounds; U0 §4.1) — this chapter is where α and β come from. `CS->MATHS:` "the network is physical" means the cost model is not a metaphor; you can measure its two constants.

Where this is used

U0 (the communication lens), U1 (tree all-reduce), U5 (shuffle cost), U6 (communication-avoiding), U18 (ring-all-reduce bandwidth-optimality). Units with a `depends F.E4` edge are pricing a wire.

F.E5 — Storage anatomy: a distributed filesystem and database

*Foundation chapter (the Engineering Floor). Prereq: F.E1. A **memory-jog** for what "the data layer" is* under the distributed labs — enough to reason about why a shuffle (U5) or a matmul (U6) is fast or slow. Concept-first; where a claim is specific to a system, check it against that system's current documentation.**

1. What a distributed store IS

When data outgrows one machine it is **partitioned** across many, and a layer presents the illusion of *one* filesystem or *one* database on top:

- a **distributed filesystem** stores large files as **blocks** replicated across nodes, and answers "give me file X" by locating and streaming its blocks;
- a **distributed database** stores **rows/tables** sharded across nodes, and answers point/range queries with transactions, typically keeping hot data in memory for low latency.

The abstraction is the value: your code says "read this path" or "select this key" and does not track which machine holds it.

2. The metadata path vs the data path

The load-bearing split: **metadata** (the namespace — which blocks/shards live where, permissions, sizes) is tiny but touched by *every* operation; **data** (the bytes) is huge but touched only when you actually read/write.

The European Hopsworks stack is a clean example of designing around that split *(check the current Hopsworks/RonDB documentation for version-specific details)*:

- **HopsFS** — a distributed filesystem (HDFS-API-compatible) that moves the **metadata out of a single master** and into a distributed, in-memory database, so the namespace itself scales horizontally instead of bottlenecking on one node.
- **RonDB** — the distributed, in-memory relational database (in the MySQL NDB Cluster lineage) that holds that metadata (and serves low-latency key lookups generally). It is why the metadata path stays fast under load.

The lesson generalises past any one product: *separate the small-but-hot namespace from the large-but-cold data, and scale each on its own terms.* (This substrate is also, in our own work, where an append-only event log lives — see the σ-log lab.)

3. Why locality is a cost, not a nicety

Reading a block that already sits on the machine doing the compute is a local read; reading one across the network pays F.E4's $\alpha + \beta M$. So schedulers try to **move compute to the data**, and shuffles try to keep partitions where they are. When U5's shuffle or U6's block-matmul is slow, the usual culprit is data moving that did not have to. `CS->MATHS:` "locality" is a term in the cost model, not an aesthetic preference.

4. Row stores, column stores, and object storage

Row vs column layout. A table of N rows and C columns can be laid out row by row (every field of row 1, then row 2, and so on) or column by column (every value of column 1, then column 2, and so on). The layout fixes which queries are cheap. Take b bytes per value and a query that needs c of the C columns for N' rows. A *row store* reads whole rows, about $N'Cb$

bytes, which is cheap when N' is small. A *column store* reads only the needed columns, about $N'cb$ bytes, a saving of C/c. Each column also holds values of one type, so it compresses well. Worked: revenue by region by quarter over $N' = 10^9$ rows, touching $c = 3$ of $C = 200$ columns with $b = 8$, reads about 1.6 TB from a row store and 24 GB from a column store, a factor $200/3 \approx 67$ before compression. A lookup of one whole row is the reverse case: one read in a row store, C scattered reads in a column store.

This is why one data set serves two incompatible workloads (U5 §1.1):

	OLTP (transactional)	OLAP (analytical)
typical question	"what is this customer's balance?"	"revenue by region by quarter?"
rows touched	1–10	10^6–10^9
columns touched	all of them	a few of many
writes	constant, small	bulk, infrequent
optimised for	latency, correctness	throughput
storage	row-oriented	column-oriented
schema	normalised (each fact stored once)	denormalised on purpose

Object storage. An object store keeps immutable byte objects under keys in a flat namespace and is reached over HTTP (`PUT`, `GET`, `LIST`). It has no directories to rename atomically and no in-place updates. Its capacity is effectively unbounded, it is cheap per terabyte, the provider manages durability, and any machine can reach it, so storage is no longer attached to compute. Because objects cannot be updated in place, a "table" on an object store is a set of immutable columnar files (Parquet) plus a metadata log recording which files make up each version. An **open table format** (Apache Iceberg, Delta Lake) standardises that log. Each commit adds one log entry atomically, which gives transactions over the files, schema evolution, and "time travel" (reading an earlier version, which is also what makes the as-of joins of §5.3 and U5 §4.5 possible after the fact).

5. Feature stores: storage with a clock

The storage anatomy of §1–§4 is where one of the most common production data systems lives: the **feature store**. This section states what a feature store must guarantee and prices the guarantee with the cost models of U1–U5. The correctness conditions are theorem-shaped, and we treat them as theorems.

5.1 The feature store and the FTI factorisation

Training data does not arrive ready-made. In production it is *assembled* from features computed at different times, by different pipelines, and kept in different systems. Suppose three teams build one model: one computes features from raw logs, one trains on those features, and one serves predictions in production. If each team recomputes features its own way, the system fails in three specific ways:

1. **future leakage**: a model is trained on information that became available only after the moment it is supposed to predict from;
2. **offline–online skew**: training and serving compute "the same" feature differently;
3. **lossy aggregation**: a rollup (a count, a mean, a last-k) discards information that the downstream decision needed.

A **feature store** is shared, timestamped state. It records feature values as triples $(\text{entity}, \text{timestamp}, \text{value})$, so the three pipelines can be written, scheduled and scaled independently and still compose correctly.

Definition (FTI architecture). An end-to-end ML system is in *FTI form* when it factors into three pipelines that communicate only through the store:

- **Feature (F):** raw data → feature values, written to the store keyed by (entity, timestamp);
- **Training (T):** reads the store's **offline view** (the full timestamped history) → a model;
- **Inference (I):** reads the store's **online view** (the current value per entity), together with the model, → predictions.

Figure 38.2 draws the three pipelines and the one store they share.

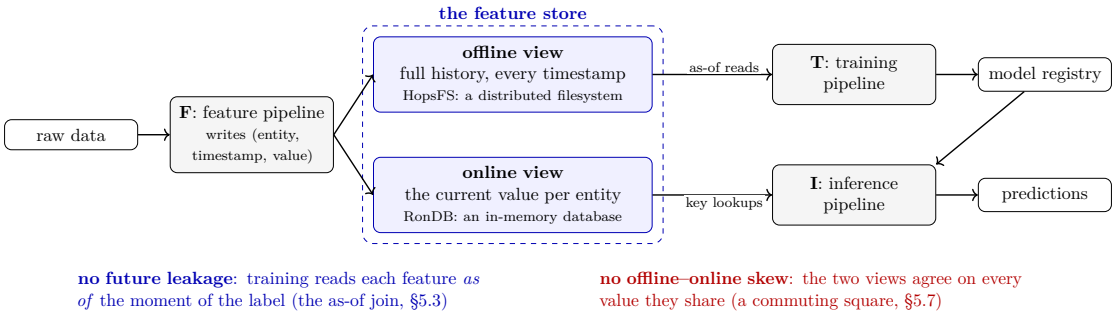


Figure 38.2: The FTI architecture on a feature store, as in Hopsworks. The three pipelines share features only through the store, and training hands inference nothing but a model, through the model registry: the feature pipeline writes timestamped values; training reads the full history as of each label's time; inference looks up the current value per entity. The offline view lives on a distributed filesystem (HopsFS), the online view in an in-memory database (RonDB). Correctness reduces to two checks at the store: no future leakage, no skew.

F never talks to T or I directly. T hands I only a model; the usual name for the place a trained model is handed over is a *model registry*. Every correctness question is therefore asked at the store's interface.

Principle (FTI modularity). *If the pipelines share state only through the store's typed, timestamped interface, then each can be scheduled (U3) and scaled on its own. The whole system's correctness then reduces to two local invariants on that interface: point-in-time correctness (§5.3) and offline–online consistency (§5.7).* This is a design principle, not a theorem. Its content is the reduction: global correctness becomes two checks at one narrow waist. The same move appears in U17, where a typed shared state bundle decouples independently evolving agents.

> **Companion reading (Dowling).** The FTI pipeline architecture and the feature-store chapters of the feature-store text give the industrial framing. This section supplies the invariants.

5.2 Worked example: one feature, three clocks

Take a feature `open_tickets` per customer, for 10^6 customers, with rows of 100 bytes.

pipeline	runs	reads / writes	size
F	every hour	writes $10^6 \times 24 = 2.4 \cdot 10^7$ rows per day	2.4 GB per day
T	nightly, over a 90-day window	reads $90 \times 2.4 \cdot 10^7 = 2.16 \cdot 10^9$ rows	216 GB (offline)
I	per request	reads 1 row per customer	10^6 rows = 100 MB (online)

Move T from nightly to weekly, or scale F from 4 machines to 40, and nothing in the other two pipelines changes. Only two things must still hold: **(1)** every training row uses feature values timestamped at or before its observation time (§5.3); **(2)** the offline and online values come from the same transform (§5.7). Note the asymmetry of the two views: the offline view grows with

history (216 GB for 90 days), while the online view is bounded by the number of entities (100 MB).

5.3 Point-in-time correctness and the as-of join

Fix an entity and a training row for it. Its **observation time** t is the moment at which, in production, the prediction would have been made. That is not the later time at which the outcome became known. The row may use a feature value only if that value was *already available* at t, that is, only if its timestamp τ satisfies $\tau \leq t$. Using a value with $\tau > t$ trains the model on its own future. Offline accuracy then looks excellent, and the signal is absent in production. (We keep the short name *label time* for t below; it always means the observation time, never the time the label was learned.)

For "$\tau \leq t$" to mean "known at t", a feature's timestamp must record when the value became *available*, not merely when the underlying event happened. A value computed from the label event itself and stamped exactly t would leak. Such pipelines use a strict $\tau < t$, or stamp availability time.

Definition (as-of join). Let the label events be (e_i, t_i), and let the feature log be a finite set of timestamped values per entity. The **as-of join** (or *point-in-time join*) builds the training row for label i by taking, for each feature, the value for entity e_i with the greatest timestamp $\tau \leq t_i$. If there is no such value, the row gets a null for that feature.

Because the log is finite, the greatest such τ exists whenever any exists.

Theorem (leakage-freeness and maximality). *The as-of join produces a training set with no future leakage. Among all leakage-free constructions it is the unique maximal one: no leakage-free construction can use a strictly more recent value for any feature of any row.*

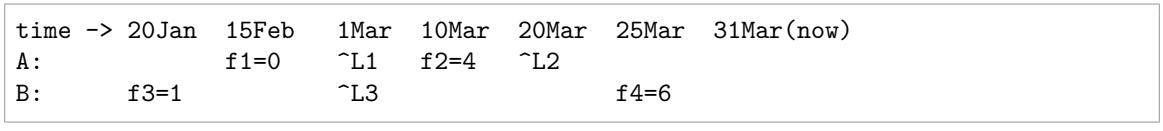
Proof. For row i and a fixed feature, the as-of join selects $\tau_i^* = \max\{\tau \leq t_i : \text{a value for } e_i \text{ exists at } \tau\}$. Every selected timestamp satisfies $\tau_i^* \leq t_i$, so no row sees its future. Now take any leakage-free construction. For row i it must use some $\tau \leq t_i$ at which a value exists, so $\tau \leq \tau_i^*$ by the definition of the maximum. A strictly more recent choice $\tau > \tau_i^*$ with an existing value would have to satisfy $\tau > t_i$, which is leakage. So the as-of join dominates every leakage-free alternative feature by feature, and the dominating choice is unique. □

`MATHS->CS:` "always do a point-in-time join" is not merely a best practice. It is the only maximal honest choice, and a naive join (take each entity's latest value *now*) is wrong by construction whenever a feature changed between t_i and the time the training set is built.

5.4 Worked example: the naive join and the as-of join, row by row

The label is "did the customer churn?"; each row's label time is its observation time. The feature is `open_tickets`. The training data is built on 31 Mar ("now").

label	entity	label time t_i	feature row	entity	timestamp τ	value
L1	A	1 Mar	f1	A	15 Feb	0
L2	A	20 Mar	f2	A	10 Mar	4
L3	B	1 Mar	f3	B	20 Jan	1
			f4	B	25 Mar	6



The naive join gives each label its entity's latest value as of now (31 Mar):

label	t_i	value used	its τ	$\tau \le t_i$?
L1 (A)	1 Mar	4 (f2)	10 Mar	no: leaks 9 days of future
L2 (A)	20 Mar	4 (f2)	10 Mar	yes
L3 (B)	1 Mar	6 (f4)	25 Mar	no: leaks 24 days of future

Two of three rows leak. The model learns "many open tickets ⇒ churn" from tickets that were opened *after* the prediction would have been made. Offline accuracy looks high; in production the signal is not there yet.

The as-of join gives each label the greatest $\tau \le t_i$ for its entity:

label	t_i	candidates with $\tau \le t_i$	greatest τ	value
L1 (A)	1 Mar	{15 Feb}	15 Feb	0
L2 (A)	20 Mar	{15 Feb, 10 Mar}	10 Mar	4
L3 (B)	1 Mar	{20 Jan}	20 Jan	1

Maximality in numbers. A monthly snapshot taken on 1 Mar would give L2 the value 0 (from 15 Feb). That is leakage-free but stale. The as-of join gives 4, and no leakage-free choice does better, because A has no value timestamped in $(10\ \text{Mar}, 20\ \text{Mar}]$.

How the join runs. The as-of join is an equi-join on *entity* plus a "latest $\tau \le t$" rule. Route both tables by entity. Within each partition, sort labels and feature rows together by (entity, time), with a feature row placed before a label row at an equal time (this realises $\le$; reverse the tie order for a strict $<$). Then scan once, carrying the last feature value seen for the current entity and emitting it at each label row. The routing is one shuffle; the rest is a local sort (U2) and a single scan (U1).

5.5 Pricing the as-of join

The byte costs of the three join strategies, and the broadcast crossover $|S|\,s < |R|\,r/(M-1)$, are derived in U5 §4.2.2. The map-side combine is in U5 §4.2.1. Here we apply them to the feature store. R is the feature log ($|R|$ rows of r bytes), S is the label table ($|S|$ rows of s bytes), and there are M executors.

Small label table: broadcast wins. Take the 90-day feature window of §5.2, so $|R|\,r = 216$ GB, with $M = 100$. Take $5 \cdot 10^6$ label events of 100 bytes, so $|S|\,s = 500$ MB.

- A shuffle join moves $216 + 0.5 = 216.5$ GB.
- Broadcasting the labels moves $0.5 \times 100 = 50$ GB, which is cheaper.
- Check against the crossover: $|R|\,r/(M-1) = 216/99 \approx 2.18$ GB, and $0.5 < 2.18$.

As U5 notes, Spark SQL's default automatic-broadcast threshold is 10 MB, so the engine will not choose this plan on its own. You hint it (or raise the threshold), and the 500 MB copy must fit in every executor's memory.

Large label table: shuffle wins. Keep the same feature log and $M = 100$, but take $5 \cdot 10^7$ labels, so $|S|\,s = 5$ GB. Broadcasting moves $5 \times 100 = 500$ GB; a shuffle moves $216 + 5 = 221$ GB. Shuffle wins, as the crossover predicts ($5 > 2.18$ GB).

The repeated join: pay the sort once. Training reruns the same join many times (hyperparameter sweeps, retraining). Suppose both sides are stored sorted by (entity, time) under one partitioner. The one-time sort moves about 221 GB, and each later join moves 0 bytes (U5's co-partitioned sort-merge join). Over $k = 10$ runs that is $221/10 \approx 22$ GB per join instead of 221

GB. This assumes the *same* label table each run. If each run brings fresh labels, only the feature side can stay pre-partitioned, and the labels are shuffled each time: $(216 + 10 \times 5)/10 = 26.6$ GB per join on average, still far below 221 GB. Appending new feature rows also costs something, since they must be sorted into place.

Rollups before the shuffle. Many features are aggregates, and U5 §4.2.1's map-side combine applies. Take a daily `clicks` feature built from $n = 10^9$ click events with key and value of 8 bytes each, over 10^6 customers, so the distinct-key ratio is $\rho = 10^{-3}$.

- `groupByKey` ships every record: $10^9 \times 16$ B $= 16$ GB.
- `reduceByKey(+)` ships partial sums. At best there is one partial per customer overall: $10^9 \cdot 10^{-3} \cdot 16$ B $= 16$ MB. At worst, each of 100 mappers holds every customer and emits one partial per customer: $100 \times 10^6 \times 16$ B $= 1.6$ GB. That is still a tenth of `groupByKey`.

The combine is safe only for an associative and commutative f. The mean is the standard trap. Suppose mapper 1 holds $\{10\}$ and mapper 2 holds $\{2, 4\}$. The mean of the two mappers' means is $(10 + 3)/2 = 6.5$, but the true mean is $16/3 \approx 5.33$. Ship the pairs $(\text{sum}, \text{count})$ instead, which do combine by addition: $(10, 1) + (6, 2) = (16, 3)$, and divide at the end: $16/3$.

5.6 Skew in the join: when one key owns a machine

Routing by entity sends all of one entity's rows to one machine. With M machines and n rows, balance means about n/M rows each. Two facts from earlier units decide what happens when one entity is very active:

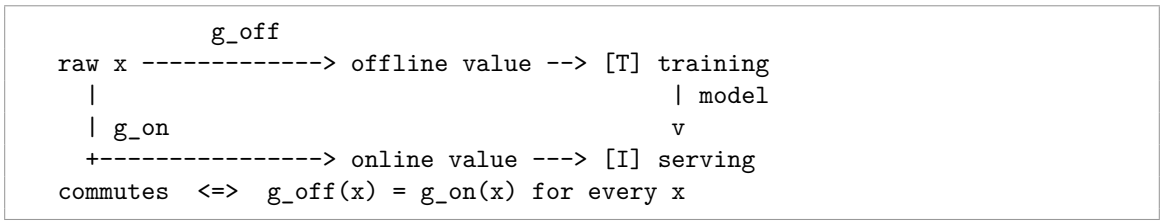
- a key that appears $\Omega(n/M)$ times cannot be separated by any splitter; the fix is to find the heavy keys with a sketch (Count-Min, U4 §4.4) and route them to dedicated machines (U2 §4.3);
- the makespan of a stage is at least its longest task (LB1, U3 §2.3), so the overloaded machine is a *straggler* (U3 §4.2) that sets the time for the whole stage, whatever the other $M - 1$ machines do.

Worked example: one customer holds 30% of all updates. Take $n = 10^9$ feature updates on $M = 100$ machines, so the ideal load is 10^7 rows each.

- *Hash by customer.* The heavy customer's $3 \cdot 10^8$ rows all go to one machine, together with that machine's share of the rest, $7 \cdot 10^8/100 = 7 \cdot 10^6$. That is $3.07 \cdot 10^8$ rows, **30.7 times the ideal**. If time is proportional to rows, the stage takes about 30 times longer than it should.
- *Fix: key the heavy customer by (customer, time range).* Choose 30 time ranges holding 10^7 of its rows each; the range boundaries are quantiles of its timestamps, found by the sample-sort splitter step of U2. Put the ranges on 30 dedicated machines, and spread the other $7 \cdot 10^8$ rows over the remaining 70 machines at 10^7 each. Every machine now holds 10^7 rows. The heavy customer's labels are routed by the same time ranges.
- *The as-of subtlety.* The first label in a range may need the customer's last update *before* the range starts, and that update lives on the previous range's machine. Each range therefore receives one carry-in value per feature from its predecessor: at most 30 extra rows per feature, in one extra round. This is a shift, the simplest relative of U1's scan.

5.7 Offline–online consistency: a commuting square

T reads the offline view (batch, columnar, bulk scans); I reads the online view (rows, millisecond key lookups). A feature is a transform g of raw input x. Here "the transform" includes any parameters it was fitted with, such as a mean and standard deviation. Let g_{off} be the transform used to build training data and g_{on} the one used at serving time.



Invariant (skew-freeness). *If one transform definition is used in both contexts, $g_{\text{off}} = g_{\text{on}} = g$, then for every raw input x the offline and online values coincide, and skew is identically zero. Conversely, if both contexts are given the same raw input x and g is deterministic, then any difference $g_{\text{off}}(x) \neq g_{\text{on}}(x)$ is a witness that two different transforms were deployed.*

Raw input reaches the model by two paths, through training and through serving. Skew-freeness says the square commutes, and it commutes *by construction* exactly when both paths execute one shared definition. This is why feature stores insist that a feature be defined once and executed in both contexts rather than re-implemented per context. The converse needs its hypothesis: if serving sees different input (late-arriving events, a stale online row), the values can differ even under one g. That is a freshness problem on the input side, not a transform problem.

Worked example: a skew you can measure. Take a z-scored spend feature, $g(x) = (x-\mu)/\sigma$, with $\mu = 50$ and $\sigma = 20$ fitted on the training data. Serving re-implements the transform and recomputes $\mu' = 60$ from yesterday's traffic. For $x = 80$, offline gives $(80 - 50)/20 = 1.5$ and online gives $(80 - 60)/20 = 1.0$. In general $g_{\text{off}}(x) - g_{\text{on}}(x) = (\mu' - \mu)/\sigma = 10/20 = 0.5$ for **every** x. A constant offset of this kind is the signature of two transforms. The fix is to store (μ, σ) with the single definition of g, so serving reads the parameters instead of refitting them.

5.8 When is a rollup safe? Sufficiency and lumpability

Features are often **aggregations**: counts, sums, means, last-k values over a window. Aggregation throws information away on purpose. The question is whether it throws away anything the *downstream decision* needs, and that question has a precise answer.

Setting. Let h be an entity's raw event history in a window, and $A(h)$ the rollup stored as the feature. Suppose h is drawn from a model $f(h \mid \theta)$ with unknown parameter θ, and the decisions of interest are rules $\delta(h)$ judged by their risk $R(\theta, \delta) = \mathbb{E}_\theta[L(\theta, \delta(h))]$ for some loss L.

Definition (sufficient statistic). A is *sufficient* for θ when the conditional distribution of h given $A(h)$ does not depend on θ. Equivalently (the Fisher–Neyman factorisation criterion), the likelihood factors as

$$f(h \mid \theta) = g\big(A(h), \theta\big)\, k(h).$$

In words: two histories with the same rollup carry the same evidence about θ.

Proposition (aggregation sufficiency). *If A is sufficient for θ, then for every decision rule $\delta(h)$ there is a rule δ' that depends on h only through $A(h)$ (possibly using extra randomisation) and has the same risk, $R(\theta, \delta') = R(\theta, \delta)$ for every θ. So storing $A(h)$ instead of h loses nothing for the decision class.*

Proof. Given $A(h) = a$, draw a synthetic history h' from the conditional distribution of h given $A(h) = a$. By sufficiency this distribution does not involve θ, so it can be sampled without knowing θ. Under every θ, h' has the same distribution as h. Set $\delta'(A(h)) = \delta(h')$. Then δ' and δ have the same distribution of decisions under every θ, and hence the same risk. □

When A is not sufficient, this guarantee is gone, and the discarded detail may matter. Whether a rollup is safe therefore has no answer until the decision class is named.

Worked example: the click count. Take two customers' last five sessions (1 = clicked): A = 1 1 0 0 1 and B = 0 1 1 1 0. The rollup "clicks in window" is 3 of 5 for both.

- *Decision about the click rate p* (for example "is $p > 0.5$?") under the i.i.d. Bernoulli model. The likelihood is $f(h \mid p) = p^3(1-p)^2$ for both histories. It depends on h only through the count, so by Fisher–Neyman the count is sufficient, and both customers give $\hat{p} = 3/5 = 0.6$. The rollup is lossless for this class.
- *Decision "did the customer click in the latest session?"* A: yes; B: no. This question is about h itself, not about p, so it lies outside the class the count is sufficient for, and the count 3 cannot answer it. The rollup is lossy here; a last-k feature is needed as well.

The Markov instance: lumpability. When raw events follow a finite Markov chain (U12 §2.1), a common rollup replaces the state with a coarser label (for example, "active" for several browsing-like states). Write $P(i, j)$ for the transition probability from state i to state j, so each *row* sums to 1. (U12 writes the transpose, a column-stochastic matrix.)

Definition (lumpable, after Kemeny–Snell). A partition of the states into lumps $B_1, \ldots, B_m$ is *lumpable* when, for every pair of lumps B_k, B_l, the probability of moving into B_l, $\sum_{j \in B_l} P(i, j)$, is the same for every state $i \in B_k$. Kemeny and Snell show that the lumped process is then itself a Markov chain for every initial distribution, with transition probabilities equal to these common values.

In the language of sufficiency: under lumpability the current lump is sufficient for forecasting future lumps, so a lump-level feature loses nothing for any decision that depends only on the future lump trajectory. When lumpability fails, states in the same lump have different futures, and the coarse label hides them.

Worked example: lumpable or not. Take states a = browsing, b = cart, c = inactive, with lumps $\{a, b\}$ = "active" and $\{c\}$ = "inactive".

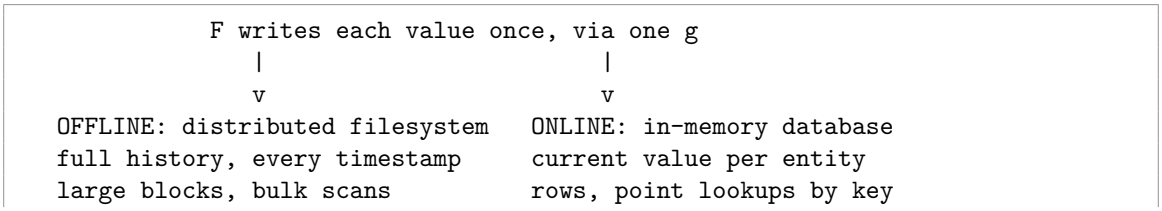
from	to a	to b	to c	to lump $\{c\}$
a	0.5	0.3	0.2	**0.2**
b	0.4	0.4	0.2	**0.2**
c	0.1	0.1	0.8	0.8

Both active states move to $\{c\}$ with probability 0.2, and the lump $\{c\}$ has only one state, so the partition is lumpable. The two-state chain has active → inactive with 0.2 and inactive → active with $0.1 + 0.1 = 0.2$, and it is exact. Now change row b to $(0.2, 0.3, 0.5)$. State a still moves to inactive with 0.2, but b does with 0.5. The "active" flag now hides a $0.5/0.2 = 2.5$-fold difference in one-step churn risk: the partition is not lumpable, and the rollup is lossy for churn forecasting.

The same sufficiency condition is what makes a lab's σ-log enough to check the work without re-running it (U17 §6).

5.9 Two substrates, and algorithms you already know

The offline and online views are the two storage kinds of §1, used in two roles:



```
read by T (as-of joins)          read by I (per request)
216 GB in the §5.2 example       100 MB in the §5.2 example
```

An online lookup is the as-of join evaluated at $t = \text{now}$, provided the online store has absorbed every update up to now. If it lags by Δ, serving is effectively an as-of join at now $- \Delta$, which is one source of the input-side skew in §5.7. The HopsFS and RonDB split of §2 is a worked example of the same metadata-path / data-path design.

Nothing in a feature store is a new algorithm. It is the algorithms of Weeks 1–4 under a correctness contract:

- assembling training sets is **joins with byte costs** (U5 §4.2.2, §5.5 above);
- partitioning by time range is **sample sort with skew control** (U2 §4.2–4.3, Lab 2; §5.6 above);
- rollups before the shuffle are the **map-side combine** (U5 §4.2.1);
- assigning dense feature indices across shards is a **scan**: each shard's offset is the prefix sum of the earlier shards' counts (U1).

Where this is used

U5 (shuffle + JOIN locality), U6 (block-matrix data movement), and the distributed labs on the F.E3 cluster. Units with a `depends F.E5` edge care where the bytes physically are.

Exercises

E.FE5.1 *(Price the as-of join.)* There are L labels and F feature updates spread over p machines. Using the route-sort-scan procedure of §5.4, give the total work, the depth, and the number of communication rounds. Which term dominates when $F \gg L$?

E.FE5.2 *(When to broadcast.)* In the setting of E.FE5.1, with labels of s bytes and feature rows of r bytes, derive the condition under which broadcasting the label table beats a shuffle join. State the memory condition too, and explain why the engine's default threshold may not choose the cheaper plan.

E.FE5.3 *(One heavy customer.)* One customer holds 30% of all updates. Say what breaks under hash partitioning by customer, using LB1 (U3). Say which part of U2 fixes it. Then explain why the fix needs one carry-in value per time range for an as-of join, and how many rounds the carry-in costs.

F.E6 — Runtime anatomy: batch, actor, streaming, and serving engines

*Foundation chapter (the Engineering Floor). Prereq: F.E2 (docker), F.C1 (the distributed cost model). A **memory-jog** for what a distributed compute engine* is*, so a lab can pick the right one for the shape of its job rather than forcing everything through one tool.*

1. The shape they all share

A distributed engine has a **driver** (a coordinator that holds your program and the plan) and **executors** (workers on the cluster nodes that do the actual compute). You submit a job; the engine builds a **task graph** — a DAG of stages/tasks — and schedules its tasks across the executors, moving data between stages (the shuffle of F.C1/U5). `MATHS->CS:` the task graph is the dependency DAG of U0's work/depth, now placed on physical machines.

2. Fault tolerance: lineage vs checkpoint

A node dies mid-job; the engine must not lose the work. Two strategies:

- **lineage (recompute):** record *how* each partition was derived from its inputs; on failure, recompute just the lost partitions from their ancestors. Cheap to record, can be expensive to replay.
- **checkpoint (restore):** periodically snapshot state to durable storage (F.E5); on failure, roll back to the last snapshot. Costs steady-state I/O, but recovery is bounded.

Batch systems lean on lineage; long-running streaming systems lean on checkpoints (you cannot recompute an infinite stream from the beginning).

3. Four engines, one sentence each

Pick by the *shape* of the job (do not use one engine for all four):

- **A batch/dataflow engine** — partitioned collections transformed by a dataflow of operators, fault-tolerant by **lineage**; the natural home for map–shuffle–reduce, SQL-style joins, and iterative batch ML. *(Spark is the representative; U5's RDD/DataFrame lab.)*
- **A task-and-actor engine** — general distributed Python: fine-grained **tasks** and stateful **actors**, the natural home for irregular, ML-flavoured parallelism (parameter servers, RL, hyperparameter sweeps). *(Ray is the representative; U9's distributed-optimisation labs.)*
- **A true-streaming engine** — unbounded event streams with **event-time** and **watermarks** and **checkpointed** state, the natural home for continuous, low-latency pipelines. *(Flink is the representative; U5's streaming-semantics note.)*
- **An inference-serving engine** — a fleet of accelerator workers answering an unbounded stream of token-generation requests against a **read-only** model, fault-tolerant by **replica restart** (a dead replica just reloads the weights). Its identity is *maximise tokens/sec under a latency objective*, and it earns that with three moves: **continuous batching** (a new request joins the running batch each decode step instead of waiting for a fixed batch to fill), a **paged KV-cache** (the per-request attention cache is paged like virtual memory, so the fleet's device memory does not fragment), and **tensor/expert parallelism** (one model sharded across devices — tensor-parallel splits each layer's matmul, expert-parallel routes mixture-of-experts tokens to their expert shards). *(vLLM is the representative; U0L/U18's serving labs. Concept-level; treat any throughput or memory number as indicative, and check current documentation.)*

The one-line test: *bounded batch → dataflow engine; irregular Python/ML tasks → actor engine; unbounded low-latency streams → streaming engine; a stream of LLM token-generation requests under a latency target → inference-serving engine.*

Where this is used

U5 (the dataflow + streaming labs), U9/U18 (the actor-engine optimisation/DL labs), U0L/U18 (the inference-serving lab where the U0L tokens/sec estimate meets a real serving fleet); all run on the F.E2/F.E3 substrate over F.E5 storage. Units with a `depends F.E6` edge are choosing an engine.

Lab (Engineering Floor) — Proving contribution by replaying the log

A flipped-classroom lab. Prereq: F.E1b (git). ***No agent required****: you run an ordinary project in a plain git repository and let the commit history itself attest who did what, when. This is*

the honesty model of the book made operational — a byline asserts contribution; a replayed log proves it.

The idea

Every commit records an author, a timestamp, a parent, and a diff. Read in order, that chain is a **verifiable trajectory**: who added which lines, who fixed whose mistake, in what sequence. You never have to *claim* a contribution split — you can *reconstruct* it. That is why, in this book's authorship model, the accountable human is the byline and the collaboration is attested by the log (the σ-chain / commit trail), not by adding names to an author list.

Hash functions and SHA-256: what makes a chain tamper-evident

Both chains in this lab, the σ-log and git, link each entry to the one before by a **hash**. This section says exactly what that is.

A hash function is a map $H : \{\text{byte strings}\} \to \{0,1\}^n$: any input, of any length, goes to one output of fixed length n, its **hash** or **digest**. For SHA-256, $n = 256$, written as 64 hexadecimal digits (each hex digit is 4 bits). It is **deterministic** (the same bytes give the same hash on every machine), the output has **fixed length** whatever the input size, and it shows **avalanche**: change one character and about half the output bits flip. For example, with `printf 'abc' | shasum -a 256` (or `sha256sum` on Linux):

```
abc  →  ba7816bf8f01cfea414140de5dae2223b00361a396177a9cb410ff61f20015ad
abd  →  a52d159f262b2c6ddb724a61840befc36eb30c88877a4030b65cbe86298449c9
```

The inputs differ in one letter; 122 of the 256 output bits differ. (The first line is the official SHA-256 test vector for `abc`.)

A hash map is a different use of hashing. A *hash map* (a dictionary, `dict` in Python, `HashMap` in Java) stores key–value pairs in an array of buckets and uses a hash of the key to choose the bucket, so lookup takes expected constant time. Spark's hash partitioning sends a record with key k to partition $h(k) \bmod p$ the same way (U5). Those hashes only need to be **fast and spread keys evenly**; collisions are expected and handled. A tamper-evident log needs a **cryptographic hash function**, one with three further properties:

property	"it is infeasible to …"	best known generic cost for SHA-256
preimage resistance	find *any* input with a given hash	about 2^{256} evaluations
second-preimage resistance	find a *different* input with the same hash as a given one	about 2^{256} evaluations
collision resistance	find *any* two different inputs with the same hash	about 2^{128} (the birthday bound)

These properties are **not proved**: they are what decades of public cryptanalysis have failed to break. The birthday bound is the generic limit: with $2^{n/2}$ random inputs, two share an n-bit hash with constant probability, just as 23 people suffice for a 50% chance that two share a birthday.

How SHA-256 computes a digest. SHA-256 is the 256-bit member of the SHA-2 family, standardised by NIST in FIPS 180-4. It follows the **Merkle–Damgård** construction:

1. **Pad** the message: append a single `1` bit, then `0` bits, then the message length in bits as a 64-bit number, so the padded length is a multiple of 512 bits.
2. **Split** the padded message into 512-bit blocks, and start from a fixed initial state of eight 32-bit words (256 bits).

3. **Compress** each block into the state in turn: 64 rounds that combine the block's words with the state using addition mod 2^{32}, bit rotations and bitwise logical functions.
4. **Output** the state after the last block: the 256-bit digest.

Its predecessor SHA-1 (160-bit output) has fallen: a practical SHA-1 collision was published in 2017, and SHA-1 is being retired. Git identifies commits by a hardened variant of SHA-1 by default, and also supports repositories that use SHA-256.

Why this makes the chain tamper-evident. Each σ-log line stores the SHA-256 of the whole previous line, and each git commit's id is a hash of its content, which includes its parent's id. To change an old entry without being noticed, an attacker would have to produce a different entry with the *same* hash (a second preimage), which is infeasible; otherwise the next link no longer matches, and so on to the end of the chain. So whoever holds the latest hash holds a fingerprint of the entire history. The chain is only *tamper-evident*, not tamper-proof: someone who controls every copy can rewrite the whole chain and recompute every hash. That is why the last hash must also be held by someone else (your uploaded receipts, a published hash).

Checking your own lab receipts

Every course lab's `run-lab.sh` writes a text receipt: a header, the `MEASURED` lines, and a log tail. It then hashes the whole file and appends one last line, `sha256: <64 hex digits>`, the SHA-256 of every line before it. It also appends one JSON line to `labs/sigma-log.jsonl`: `{"ts": …, "lab": …, "receipt": …, "sha256": <the receipt's hash>, "prev": <SHA-256 of the previous line>}`, where the first line has `"prev": "genesis"`. Both claims can be checked in three commands (use `sha256sum` on Linux and WSL, and `shasum -a 256` on macOS):

```
R=$(ls -t lab2-sample-sort-spark/receipts/receipt-lab2-*.txt | head -1)
tail -1 "$R"                          # the hash the receipt claims
sed '$d' "$R" | shasum -a 256         # recompute it over all but the last line:
  the two must agree
tail -2 sigma-log.jsonl               # the newest line's "prev" is the hash of
  the line before it
```

To check a link of the chain, hash line K (`sed -n 'Kp' sigma-log.jsonl | shasum -a 256`) and compare it with the `prev` field of line $K+1$. These checks prove *self-consistency*: the receipt has not been edited since it was hashed, and the chain has not been edited since its last line was written. As the previous section explains, they do not stop someone who controls every copy from rewriting everything and rehashing. That is why the hash you upload with a submission, held by someone else, is the part that makes the record tamper-evident.

What you build

A tiny project (say, a work-optimal `scan` in a language of your choice), developed as a *sequence of small commits*: a first draft, a bug, the fix, a review comment folded in, a passing test. Then you write a short **replay** that reads the log and answers, mechanically:

- how many commits, by which authors, over what span;
- for a given final line, which commit introduced it (`git blame`);
- which commit *fixed* an earlier one (the diff that deletes a line a previous commit added);
- the dependency order (each commit's parent) — the trajectory.

The exercise

1. **Set up (F.E1b).** `git init` a repo; make the sequence of small, well-messaged commits above.
2. **Replay by hand.** `git log --stat`, `git blame <file>`, `git log -p` — narrate the contribution story from the history alone. Write three sentences: *what was drafted, what broke, who/what fixed it* — each backed by a commit hash.
3. **Replay by program (light).** A dozen-line script over `git log --format=...` that prints the per-author commit counts and flags each "fix" commit (a commit whose diff removes a line an earlier commit added).
4. **The point (deep).** Argue in one paragraph why the replayed log is a *stronger* attestation of contribution than a name in an author list: it is granular (per line, per step), ordered (the trajectory), and independently verifiable (anyone with the repo can re-run your replay). A published artifact carries a resolvable pointer to this trail (a content-hash / locator) so the attestation travels with the work.

Optional: sovereignty + integrity

Push the repo to your own server (F.E1b §3) and mirror outward. Publish a **self-signed Merkle root** of the history in a channel you control; anyone can later verify that the trail they hold is the one you attested. That is the printed-attestation idea: the record's integrity does not depend on any platform you do not govern.

Where this is used

This lab is the engineering-floor counterpart of the book's authorship/provenance model: the same commit trail that proves a course project's contributions is the mechanism a published preprint uses to attest a human–agentic collaboration without placing anything but the accountable human in the byline.

Acronyms and abbreviations

ADMM alternating direction method of multipliers.

ANN approximate nearest neighbour (search).

BLB bag of little bootstraps.

BMLS Brent–Mirsky layered scheduling (acyclic restriction).

DPRAM distributed parallel random-access machine.

FTI Feature/Training/Inference pipeline architecture.

IAS Interaction Analysis of Interacting Agent Systems.

JL Johnson–Lindenstrauss (lemma).

KV key–value (cache).

LLM large language model.

LSH locality-sensitive hashing.

MDE minimum-distance estimate.

MLOps machine-learning operations.

MoE mixture of experts.

MPC massively parallel computation (model).

MST minimum spanning tree.

NC Nick's class (polylogarithmic parallel time, polynomially many processors).

OBG obligations graph.

PRAM parallel random-access machine.

RAG retrieval-augmented generation.

SCC strongly connected component.

SCT statistical contract theory.

SGD stochastic gradient descent.

SRP statistical regular paving.

TBMLS Tarjan–Brent–Mirsky layered scheduling.

TV total variation (distance).

Subject index

2.5D algorithm, 140

acronym discipline, 34, 54, 59
activation checkpointing, 316
AdaGrad, 181
Adam optimiser, 181
adverse selection, 279
agent, 291
agent (contract theory), 279
agent loop, 318
AI product management, 285
algebraic connectivity, 229
all-reduce, 414
all-to-all exchange, 114
alpha-beta network model, 414
alternating direction method of multipliers (ADMM), 328
antichain, 384
approximate nearest neighbour (ANN), 247
arithmetic complexity, 136
arithmetic intensity, 45, 140
arithmetic series, 363
ART (accountability, responsibility, transparency), 22
as-of join, 125, 418
attestation envelope, 298
augmented Lagrangian, 327
autoregressive model, 42
axiom footprint, 389

Bag of Little Bootstraps (BLB), 338
balance factor, 79
Ballard–Demmel–Holtz–Schwartz lower bound, 139
bandwidth, 413
batch/dataflow engine, 424
bias-variance tradeoff, 256
big-O notation, 362
big-Omega notation, 362
big-Theta notation, 362
birthday problem, 370
bits-for-accuracy, 343
block replication, 415
BlockMatrix (Spark), 141
Boolean satisfiability (SAT), 380
bootstrap, 338
Boruvka's algorithm, 207
Brent's bound, 293
Brent's principle, 377
Brent's scheduling theorem, 15
Brent's theorem, 76, 138
broadcast join, 120
bulk synchronous parallel (BSP), 18
bulk-synchronous parallel (BSP), 231, 378
Byte-Pair Encoding (BPE), 303
Byzantine fault tolerance, 298

candidate family, 267
Cannon's algorithm, 139
capability lattice, 294
cert ladder, 200
certificate (witness), 379
CERTIFIED (proof flag), 200
chain, 384
Chebyshev inequality, 369
checkpoint (fault tolerance), 424
Chernoff bound, 370
circuit value problem, 381
classical trio (Lean axioms), 397
closed-weight model, 44
CLOUD Act, 21
co-partitioned join, 121
collectives of interacting agents, 30
column space, 364
column-oriented storage, 415
combiner, 119
COMDAIM, 291
commit, 409
commit trail, 427
commons, 291
communication lower bound, 139
communication term (alpha-beta), 19
communication-avoiding algorithm, 140, 414

communication-constrained lower bound, 339
commutative monoid, 259
compactness (fails on finite structures), 394
compactness theorem, 392
comparison lower bound, 75
complementary slackness, 327
completeness theorem (Gödel), 392
condensation, 210
condition number, 364
conditional density estimator, 302
conditional probability, 367
conformal prediction, 31, 285
conjugate gradient method, 166
connected components, 204
consensus optimization, 328
consistency, 391
container, 411
context projection, 296
continuous batching, 311, 424
contract, 406
convergence
  almost sure, 368
  in distribution, 368
  in probability, 368
convex function, 365
convex projection, 172
convex set, 365
Cook–Levin theorem, 380
coreset, 248
cosine similarity, 155
Count-Min sketch, 97
coupon collector problem, 370
CRCW PRAM, 15
CREW PRAM, 15, 76
cross-entropy, 374
cross-entropy loss, 42
cryptographic hash function, 425
Curry-Howard correspondence, 197, 395
cut property, 205

data engineering, 113
data locality, 115, 415
data market, 284
data parallelism, 305
data partitioning, 415
data path, 415
data Shapley value, 284
data warehouse, 113
data-processing inequality, 267
dcpo, 387
de Bruijn criterion, 197, 399
decentralised SGD (DSGD), 330
decidable set, 392
decision problem, 379
decision tree (sorting), 75
declarative query, 112
decode ceiling, 46, 314
decode phase (LLM inference), 45
definitional equality, 397
dense model, 48
density, 371
density estimator on sequence space, 42
dependent type, 395
deployed model as contract, 31
Deployment, 412
depth (parallel), 377
depth (span), 15
deque, 90
descent lemma, 179
desired state, 412
difference-in-differences estimand, 271
digital agent, 291
Dilworth's theorem, 385
dimension table, 113
DIMSUM, 155
distributed database, 415
distributed filesystem, 415
distributed scan (2-round algorithm), 68
divide-and-conquer, 362
Docker Compose, 411
doubly stochastic mixing matrix, 330
down-sweep (distribute phase), 66
DPO (direct preference optimisation), 44
DPRAM, 57
DPRAM-JEM, 21
driver (distributed engine), 423
dual decomposition, 170
dual function, 326, 366
Dvoretzky–Kiefer–Wolfowitz inequality, 373
dynamic pricing, 285

E-value, 280
Eckart–Young theorem, 149
Eckart-Young theorem, 365
edge-cut, 229
egress lattice, 294
Ehrenfeucht-Fraisse game, 394
eigenvalue, 364
empirical cross-entropy loss, 303
empirical distribution function, 372
empirical measure, 265
entailment ($\models$), 391
entropy, 374

Entscheidungsproblem, 392
EREW PRAM, 15
estimator, 371
Euler-tour technique, 66
event, 367
event time, 424
exchangeability, 342
exclusive scan, 64
executor (distributed engine), 423
expectation, 367
expected utility, 405
expert parallelism (EP), 49

facility location, 250
fact table, 113
Fano's inequality, 340
feature aggregation, 421
feature store, 416
Fiedler vector, 229
filesystem, 36, 408
finite model theory, 393
finite probability space, 367
firmware, 35
Fisher information, 337
Fisher–Neyman factorisation, 421
fixed point, 388
Flajolet-Martin sketch, 102
forge, 410
forward-backward SCC, 212
four pillars (ScaDaMaLe), 29
Frobenius norm, 364
FSDP (fully sharded data parallel), 308
FTI architecture, 416
full fine-tuning, 315
future leakage, 125, 416

Gale-Shapley matching, 282
Galois connection, 294, 386
Gather-Apply-Scatter (GAS), 233
generalized Boolean algebra, 385
geometric series, 363
Gibbs' inequality, 375
Givens rotation, 167
Glivenko–Cantelli theorem, 372
gossip averaging, 330
gradient, 365
gradient accumulation, 306
gradient descent, 179
Graham's theorem, 87
Gram matrix, 153
graph, 382
graph colouring (parallel Gauss–Seidel), 167
graph Laplacian, 229
graph partitioning, 229
GraphX, 233
greedy algorithm, 16
  submodular maximisation, 249
greedy scheduling, 16
GS-UCB algorithm, 282

harmonic series, 363
hash function, 383, 425
hash map, 425
hash partitioning, 115
hash-to-min, 214
Hasse diagram, 384
heavy hitter, 101
height, 384
held-out test set, 316
hierarchical navigable small world (HNSW), 247
Hillis-Steele scan, 67
histogram density estimation, 256
Hoeffding inequality, 369
Hoeffding's inequality, 269
HOGWILD (lock-free SGD), 188
honest-reporting mechanism, 299
honesty contract (CERTIFIED/STATED/UNCERTIFIED), 200
HopsFS, 415
human agent, 291
hypergeometric split (reservoir merge), 104
HyperLogLog, 97

IAS (Interaction Analysis of Interacting Agent Systems), 285
  substrate graph (SPIG), 293
idempotence, 126
image, 411
immutable attribute (chattr +i), 37
importance sampling, 156
incentive, 405
incentive compatibility, 279, 406
inclusive scan, 64
incompleteness theorems (Gödel), 392
independence, 367
indicator variable, 367
inductive type (Lean), 396
inference-serving engine, 424
information rent, 279
inherently sequential, 381
inverse Ackermann function, 207

Jaccard similarity, 246
JEM engine (IAS), 23
Johnson–Lindenstrauss lemma, 150
join, 385
jurisdiction-explicit model (JEM), 20

k-means, 241
k-means parallel (k-means‖), 243
k-means++, 242
Kahan summation, 24
kernel (proof checker), 197
kernel mode, 35
keyless detector, 317
keyless watermark detector, 271
KKT conditions, 327
Kleene fixed-point theorem, 388
knee (alpha-beta model), 413
Kruskal's algorithm, 206
Krylov subspace, 166
Kubernetes, 412
Kullback–Leibler divergence, 374
KV cache, 44, 310

L1-consistency, 260
label propagation, 209
Lagom Obligations Graph, 293
Lagrangian, 326, 366
language model, 42, 302
lasso, 329
latency, 413
lattice, 385
leaf label, 258
Lean (proof assistant), 389, 395
Lean 4 (proof assistant), 197
least-squares problem, 163
leverage score, 169
lineage (RDD), 117
lineage-based fault tolerance, 424
linear map, 363
linearity of expectation, 367
list scheduling, 86
little-o notation, 362
Lloyd's algorithm, 241
local model, 313
  capacity, 314
  CPU only, 315
locality-sensitive hashing, 244, 383
log replay, 426
longest processing time (LPT), 88
low-rank adaptation (LoRA), 312
Lowenheim-Skolem theorem, 392
lumpability, 297, 422

M-estimation, 303
M-estimator, 336
makespan, 85
map-shuffle-reduce, 18
map-side combine, 119
mapped regular paving, 257
MapReduce, 114
MapReduce Class (MRC), 122
marginal coverage, 342
Markov inequality, 369
Markov's inequality, 101
massively parallel computation (MPC), 18
Master Theorem, 137
Master theorem, 362
matching market, 282
mathematical induction, 389
mathesis, 30
Mathlib, 197
matrix multiplication, 134
median-of-medians, 75
meet, 385
memory-bandwidth bound, 45
mergeable data structure, 105
mergesort, 74
Merkle root, 427
Merkle-Damgard construction, 425
metacircular fixed point, 299
metadata path, 415
microeconomics pillar, 29
MinHash, 246
minimal unsatisfiable core, 23
minimax rate, 335
minimax risk, 372
minimum spanning tree, 204
minimum-distance estimation, 267
minimum-distance estimator, 260
minimum-distance estimator (MDE), 318
Mirsky antichain layering, 293
Mirsky's theorem, 384
mixture of experts (MoE), 308
mixture-of-experts (MoE), 48
model (logic), 391
model checking, 393
model gap, 400
moment-generating function, 370
monoid, 67
monoid (sketch), 105
monotone map, 386
Moore–Penrose pseudoinverse, 163
moral hazard, 279
MRPOperate, 259

multi-head attention, 304
multinomial reweighting, 338
multiplicative weights update, 183

narrow dependency, 117
NC (complexity class), 381
Nemirovski–Yudin lower bound, 183
Nesterov acceleration, 180
network layers, 55
Newton-sketch, 331
node, 412
NodeIterator++, 224
non-standard model, 392
nonconformity score, 341
nonparametric density estimation, 260
normal equations, 163
NP (complexity class), 379
NP-complete, 380
NP-hard, 380

object storage, 113, 416
obligation (scheduling), 293
observation time, 418
offline–online skew, 416
offloading (GPU and RAM), 314
OLAP, 112
OLTP, 112
omega tactic (linear arithmetic), 198
one-shot averaging, 331, 337
online gradient descent, 183
open table format, 114
open-weight model, 44
operator, 291
operator norm, 364
oracle inequality, 268
oracle lower bound (optimisation), 183
orchestration, 124, 412
orchestrator, 412
orthogonal projection, 364
orthonormal columns, 364

P (complexity class), 379
P versus NP, 380
P-complete, 381
P-completeness, 211
pack (filter via scan), 65
paged KV-cache, 424
PageRank, 220
pairwise independence, 368, 383
parallel merge, 75
parallel mergesort, 76
parallel random-access machine (PRAM), 15, 377
parameter averaging (local SGD), 189
parameter server, 188
partial order, 384
participation constraint, 406
PARTITION problem, 85
path, 408
paving lattice, 257
Peano arithmetic, 390
performative prediction, 283
perplexity, 311, 375
Perron-Frobenius theorem, 221
ping-pong benchmark, 414
Pinsker inequality, 375
pipe, 409
pipeline bubble fraction, 307
pipeline parallelism, 307
pipeline parallelism (PP), 49
pod, 412
point query, 100
point-in-time correctness, 125
point-in-time join, 418
pointer doubling, 210
Polyak–Lojasiewicz inequality, 180
polynomial-time reduction, 380
poset, 384
positive semidefinite matrix, 364
post-training, 43
post-training quantisation (PTQ), 311
power iteration, 221
power-law degree distribution, 229
power-of-two-choices, 91
PowerGraph, 231
PRAM-to-BSP simulation, 18
pre-training, 43
preconditioning, 166
preconstructive curse, 297
predicate logic, 389
preferences (decision theory), 405
prefill phase (LLM inference), 45
prefix-sum problem, 64
Pregel, 232
Presburger arithmetic, 391
Prim's algorithm, 207
principal, 279
principal component analysis (PCA), 149
principal-agent hypothesis testing, 31
principal-agent problem, 406
process (OS), 36
projection theorem, 172

proof (formal), 391
proof assistant, 389
proof checker, 196
proof term, 197
provability (⊢), 391
proximal operator, 329
push mirror, 410

QLoRA, 315
quantifier depth, 394
quantile, 372
quantization (model weights), 44
query optimizer, 112
quick-select, 75
quicksort, 74

radix sort, 65
random range-finder, 151
random variable, 367
random-access machine (RAM), 14, 377
random-hyperplane hashing, 246
random-surfer model, 220
randomised SVD, 151
RangePartitioner, 80
rank, 364
reconciliation loop, 412
recursively axiomatizable theory, 391
recursively enumerable set, 392
reduction (parallel primitive), 15, 377
register (HyperLogLog), 102
regret (online learning), 183
regular paving, 256
replication rate, 122
repository, 410
reservoir sampling, 97
Resilient Distributed Dataset (RDD), 117
retrieval-augmented generation (RAG), 313
retrieve before you train, 317
revelation principle, 279
ring all-reduce, 305
ring-all-reduce, 187
risk (statistical), 371
RLHF (reinforcement learning from human feedback), 44
RonDB, 415
router (mixture-of-experts), 48
router (MoE), 308
row-oriented storage, 415

sample complexity, 370
sample sort, 77
Sauer–Shelah lemma, 269
scaled dot-product attention, 303
scaled dual variable, 328
scaling laws, 309
scan (all-prefix-sums), 15
scan (parallel primitive), 377
Scheffe identity, 260
Scheffe's identity, 199
Scheffé set, 318
Scheffé's identity, 265
Scheffé's theorem, 371
schema drift, 126
Scott continuity, 387
SCT sufficiency of lineage, 286
Secure Boot, 35
segmented scan, 67
self-information, 374
semilattice, 297, 385
sentence (logic), 390
Service, 412
service-level agreement, 285
SHA-256, 425
shared-nothing architecture, 113
shatter coefficient, 269
shell, 408
shuffle, 414
shuffle join, 120
shuffle phase, 114
sigma-chain, 297, 425
signature (logic), 389
SimHash, 246
singular value decomposition, 364
singular value decomposition (SVD), 148
sketch-and-solve, 169
sketched Hessian, 331
skew problem, 79
Slater's condition, 326
slowly changing dimension, 126
smooth function, 365
smoothness (L-smooth), 178
soft-thresholding, 329
sorry (Lean escape hatch), 197, 399
sorry-free proof, 389
soundness (logic), 391
sparse matrix multiplication, 141
sparse random projection, 151
sparse SRP, 257
spectral norm, 364
spectral theorem, 364
speculative execution, 90
speed-up (parallel), 15
split conformal prediction, 341

splitting method (iterative solvers), 164
SSH key pair, 410
standard error, 409
standard input, 409
standard output, 409
star schema, 113
star-contraction, 208
stationarity, 366
statistical contract, 407
statistical contract theory, 278, 342
statistical contract theory (SCT), 29
statistical guarantee as contract collateral, 31
statistical regular paving, 256
stochastic gradient descent (SGD), 184
straggler, 89
Strassen's algorithm, 136
strategic classification, 283
streaming engine, 424
streaming model, 98
strong convexity, 178, 365
strong duality, 326
strongly connected components, 210
structure (logic), 391
sub-Gaussian concentration, 150
sub-stream (HyperLogLog), 102
sublinear-memory connectivity, 215
submodular function, 249
subspace embedding, 153
Substrate-Process IAS Graph (SPIG), 293
sufficient statistic, 297, 421
SUMMA (Scalable Universal Matrix Multiplication Algorithm), 139
superstep, 232
supervised fine-tuning (SFT), 44
SVRG (stochastic variance reduced gradient), 182
system call, 35

tactic (Lean), 197, 398
Tarjan condensation, 293
Tarjan's SCC algorithm, 211
task graph, 423
task-and-actor engine, 424
TBMLS (Tarjan–Brent–Mirsky Lagom Scheduling), 293
tensor parallelism, 306, 424
tensor parallelism (TP), 49
TeraSort, 80
theory (logic), 390
token (language model), 42
tokenisation, 303
tokenizer, 43
total variation distance, 199, 265, 371
TPM (Trusted Platform Module), 35
training/serving skew, 126
Trakhtenbrot's theorem, 394
transformer, 42
tree-aggregate (distributed reduction), 68
triangle counting, 222
truncated SVD, 148
TSQR (tall-skinny QR), 168

uniformity proof (reservoir), 98
union bound, 367
union-find, 207
universal hashing, 383
up-sweep (reduce phase), 66
user mode, 35
utility function, 405

variance, 368
VC dimension, 268
verifiable trajectory, 425
version control, 409
vertex-cut, 231
virtual machine, 411
vote-to-halt, 232

watermark (streaming), 424
weak duality, 326, 366
well-founded induction, 389
wide dependency, 117
width, 384
work (parallel), 15, 377
work-optimality, 15, 137, 377
work-stealing, 90
working directory, 408

Yatracos class, 265
Yatracos method, 265

ZeRO optimizer, 308
zero-one law, 394

About the author

Raazesh Sainudiin is a mathematical statistician, coder and farmer. After degrees in mathematics and biology (Minnesota State University, Mankato) and in biometrics and statistics (M.S. and Ph.D., Cornell University), Sainudiin was a Research Fellow of the Royal Commission for the Exhibition of 1851 at the University of Oxford, Lecturer and then Senior Lecturer at the University of Canterbury, New Zealand, and Associate Professor of Mathematics with specialization in Data Science at Uppsala University (2019–2025), where Sainudiin became Docent in Applied Mathematics and Statistics. In industry, Sainudiin was Principal Data Scientist at Combient and Director of Technical Strategy and Research at Combient Mix.

The research runs from mathematical and statistical genetics and the mathematical theology of ideologies in media, through auto-validating simulation, interval arithmetic and computer-aided proofs and computational nonparametric statistics, to scalable data science and distributed machine learning; current research adds statistical contract theory and thermoeconomics. Sainudiin co-founded and leads VakeWorks in Sweden and India, whose Laboratory for Mathematical Statistical Experiments wrote this book with a commons of human and digital agents.

Sainudiin left academia to build VakeWorks and now teaches ScaDaMaLe not from a university but from Miðgarðr, a family farm, field station and gene bank for the Swedish heritage breed of pigs (linderödssvin) in Torpshammar, Sweden. Data science is one subject, mathematics, statistics, computer science and microeconomics on an engineering floor you build and operate, and every theorem in this course is implemented and measured in Docker on your own laptop, free and locally, up to large language models and distributed deep learning assembled from the course's own components. If a farmer outside any university can build a production machine-learning system end to end, so can anyone.



Raaz resting with Alpha-13, our Matriarch Linderödssvin (Swedish heritage pig breed), Spring 2025.
Miðgarðr Family Farm and Genebank of VakeWorks AB, Gim 238, 84175 Torpshammar, Sweden.
Photo: Elisabeth Bolund.