Engineering High-Performance Real-Time Leaderboards.

From $O(N^2)$ to $O(\log N)$.

Ivan Balashov

Tech Lead · SoftSwiss · 2026

Ivan Balashov.

Tech Lead at SoftSwiss.

What I work on.

Real-time tournament leaderboards and back-office systems serving 500+ casino operators — sub-millisecond hot paths inside a multi-country compliance envelope.

Where I focus.

High-load Go services, distributed systems, performance under burst load. Boring is the highest praise an engineer can give a system.

Background.

10+ years building backend infrastructure for regulated, latency-sensitive industries.

srb.balashov@gmail.com · linkedin.com/in/ivan-balashov-041b5614b

Fifty milliseconds.

Roughly how long our leaderboard took to process a single update with ten thousand users in it. Sounds fine. It is not fine.

CPU	pinned at 100%, continuously
OOM	killer running multiple times a day
Pods	restarting in a loop, losing in-memory state

The platform behind this story.

SoftSwiss. Regulated iGaming. Tournament leaderboards aren't a UX feature — they directly determine who wins money in real time.

—

500+	*casino clients*
300+	*game providers integrated*
40,000+	*games in catalog*
35,000	*concurrent sessions at peak*
10+	*country licenses · 6 compliance frameworks*

What we'll cover.

01 **Why this isn't "just sort a list"**
Conflicting demands of a regulated leaderboard

02 **The incident**
A feedback loop in production

03 **Why we didn't reach for Redis**
Four constraints that ruled it out

04 **The solution**
Skip lists, span counters, identity layer

05 **Production impact**
Before / after — and what metrics can't measure

06 **What it taught us**
Five takeaways for senior engineers

PART 01

Why this isn't "just sort a list."

Several requirements. All conflicting. All simultaneous.

Five demands. One data structure.

01	High write throughput	*Score updates arrive in bursts and bulk operations.*
02	Constant rank queries	*Players, operators all ask in parallel.*
03	Identity deduplication	*One person, multiple identifiers — must be one row.*
04	Multi-tenant isolation	*Per-operator leaderboards; no leaks across tenants.*
05	Reads can't block writes	*Under load, queues form and never drain.*

The textbook says "sorted set, done." The textbook didn't have these constraints.

Whiteboard-correct. Production-fatal.

leaderboard.go

```
mu.Lock()
defer mu.Unlock()

for i, e := range board {
    if e.UserID == id || e.Email == em {
        board[i].Score = newScore
        goto resort
    }
}
board = append(board, Entry{...})
resort:
sort.Slice(board, byScoreDesc)
```

What this costs at scale.

Identity lookup	`O(M · N)`	*linear scan per update*
Sort	`O(N log N)`	*every batch*
Rank query	`O(N)`	*no skips, no shortcuts*
When M ≈ N	**$O(N^2)$**	*the cliff*

100 users → fine. 1,000 → slow. 10,000 → falls off a cliff.

PART 02

The incident.

What the dashboards looked like at 100% CPU.

Production dashboard, the morning of.

100%	multi/day	+50%	p99
CPU pinned	OOM killer runs	traffic was enough	lock-wait spikes
continuously, not occasionally	*every restart loses in-memory state*	*to bring the whole service down*	*reads queued behind sort operations*

WHAT THE PROFILER SHOWED

The overwhelming majority of CPU time was spent inside the leaderboard merge logic — linear identity scans, slice reallocations, and re-sorting after every batch.

PART 03

Why we didn't reach for Redis.

Redis was the answer to a different question.

Why the obvious answer was the wrong one.

01

Time

Pods crashing in production. Standing up Redis to our regulated, multi-region standard is a multi-week project.

02

Latency budget

Hot-path operations had a sub-millisecond budget. A network round-trip plus serialization eats most of it.

03

Failure modes

Adding Redis means a new dependency: down, slow, partitioned, OOM. New blast radius. New audit surface.

04

Cost

Redundant, multi-region, with operational overhead — real money for what is, on inspection, a software issue.

Different question.
Different answer.

THE TEXTBOOK QUESTION

"How do I store and rank a sorted set across services?"

OUR ACTUAL QUESTION

"How do I make this in-process operation stop being O(N²)?"

PART 04

The solution.

A 1990 paper, ~300 lines of Go, and a profiler.

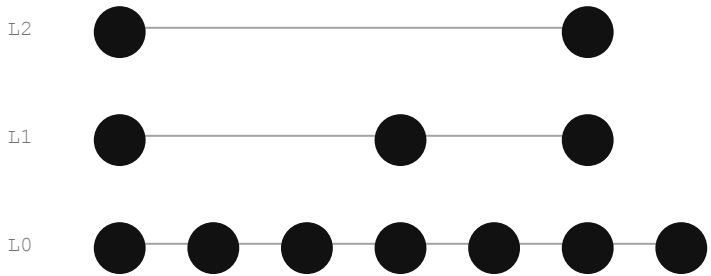
A 1990 paper, in production in 2026.

Pugh, W. (1990).

Random heights, no rotations.

Take a sorted linked list. On top of it, build a sparser one — every other node. Then a sparser one on top of that. To search, start at the top, walk forward, drop a level, repeat.

Skip list · 3 levels



Search path: skip forward at the top, drop down, repeat.

O(log N)

insert, delete, search

No rotations

no rebalancing, no parent pointers

Simpler

than red-black or AVL trees

A coin flip. Literally.

$$P(level \geq k) = p^k$$

Insert a node. Flip a weighted coin. Heads → bump it up a level. Heads again → bump it once more.

We use **p = 1/4**, same as Redis. Lower p → less memory, slightly more comparisons. Right trade for our access patterns.

MAX LEVEL CAP

`32` — *Bounds memory in pathological cases.*

EXPECTED POINTERS/NODE

`≈ 1.33` — *Very low memory overhead per node.*

SEARCH COMPLEXITY

$\approx 1.5 \cdot \log_4 N$ — *Small constant penalty for the win.*

REBALANCE CODE

`0 lines` — *No rotations. No parent pointers.*

A node — in code, and in memory.

One allocation per node. Score, identity, height, and per-level forward pointers all in one contiguous struct.

```
// node.go

type Level struct {
  Forward *Node
  Span    uint32
}

type Node struct {
  UserID    uint32
  Score     float64
  Timestamp int64
  Height    uint8
  Levels    []Level
}
```

P2 in memory · one allocation

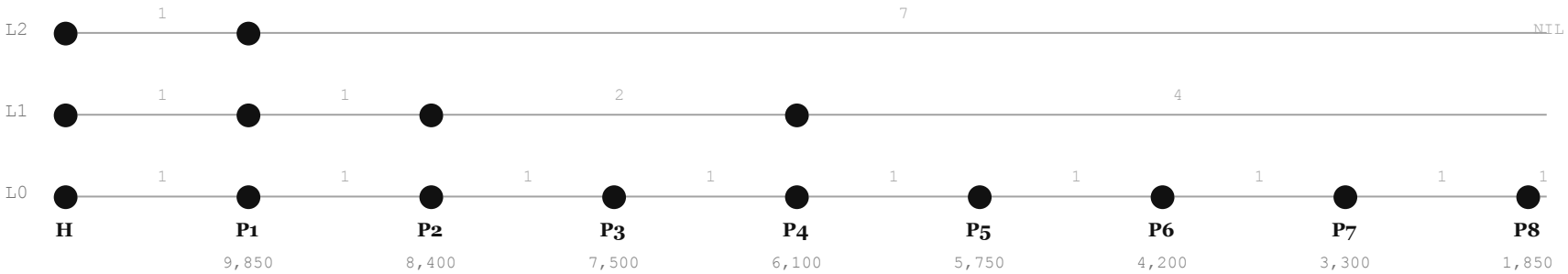
UserID	uint32	hash('player_2')
Score	float64	8400
Timestamp	int64	$1730 \cdot 10^6$
Height	uint8	2
Levels []Level	*contiguous — one slice, two entries*	
Levels[0]	Forward: → P3	Span: 1
Levels[1]	Forward: → P4	Span: 2

~ 80 B per node.
uint32 user_id.
Levels stored contiguously.

Score, TS, ID inline. Levels slice header inline; backing array is one allocation.
Identity layer hashes any incoming identifier to uint32 — no strings on the hot path.
Following Forward[0] then Forward[1] is one cache line, not two.

Eight players, three-level skip list.

Tournament after round 5. Each node's height was chosen by coin flip on insert.

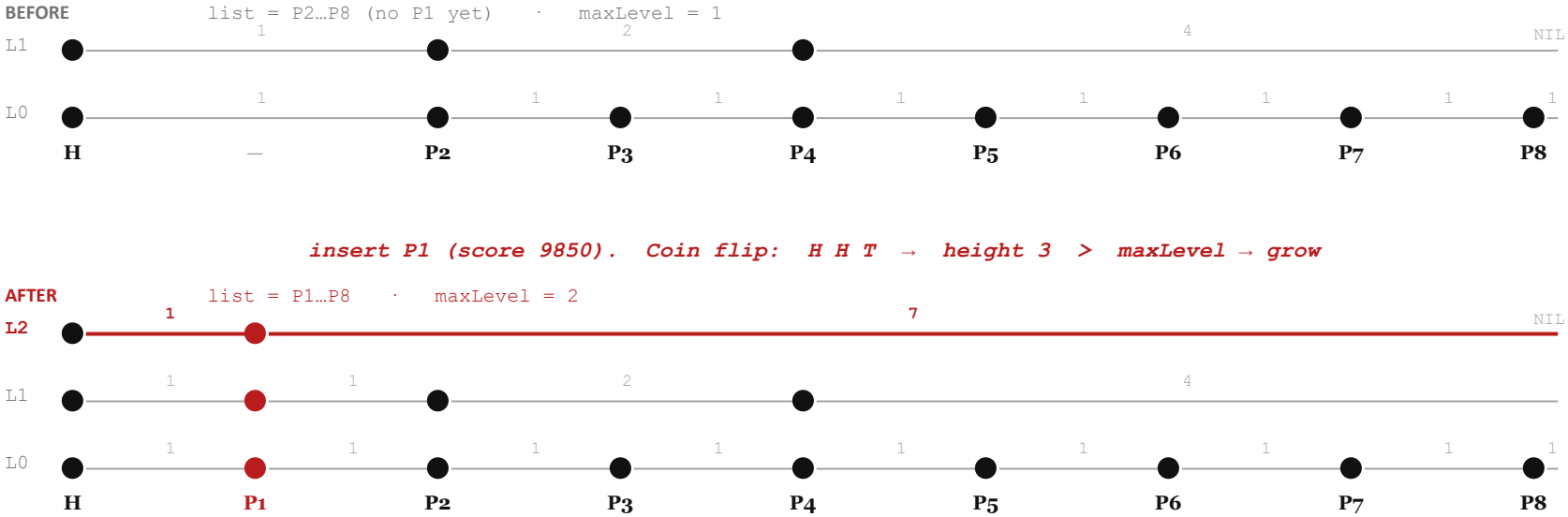


Span = how many L0 positions this arrow advances over.

Every L0 arrow has span 1. On L1, P2 → P4 has span 2 (skips P3). On L2, P1 → NIL has span 7 (covers all 7 players after P1).

How did L2 come to exist on the previous slide?

Before P1 was inserted, the structure had only L0 and L1. P1's coin flip gave it height 3 — exceeding maxLevel — so the head sprouted a new L2 pointer.

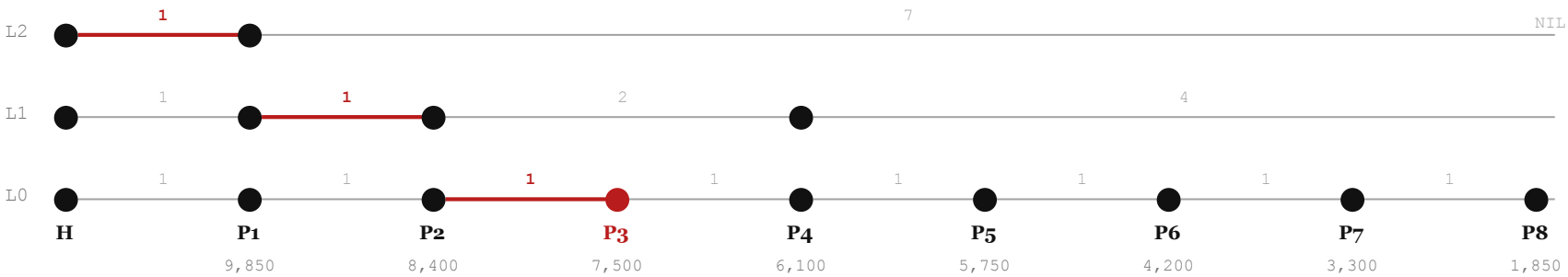


If h > maxLevel, the head sprouts (h – maxLevel) new top pointers, each pointing straight at the new node.

With p = 1/4 this happens maybe 5–10 times across millions of inserts.

First, a simpler one: where is 7,500?

Walk top-down. When the next forward would overshoot, don’t take it — drop a level instead.

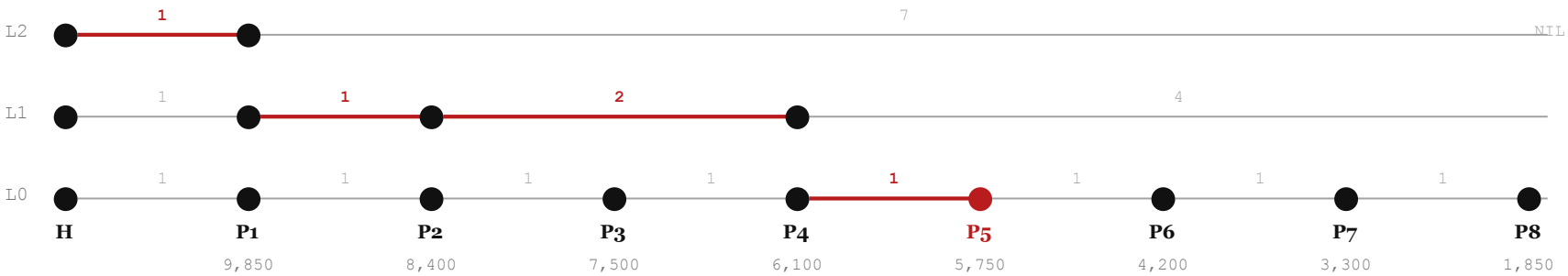


THE TRACE

L2	`H → P1`	*9850 > 7500. Take.*	+ 1	**`rank = 1`**
L2	`drop`	*P1.Forward[2] = NIL.*	–	**`rank = 1`**
L1	`P1 → P2`	*8400 > 7500. Take.*	+ 1	**`rank = 2`**
L1	`drop`	*P2.Forward[1] = P4 (6100). Would overshoot — don’t take.*	–	**`rank = 2`**
L0	`P2 → P3`	*7500 is the target. Arrived.*	+ 1	**`rank = 3`**

Where is 5,750?

Walk top-down. Every forward move adds its span to a running rank. Done in 4 moves.



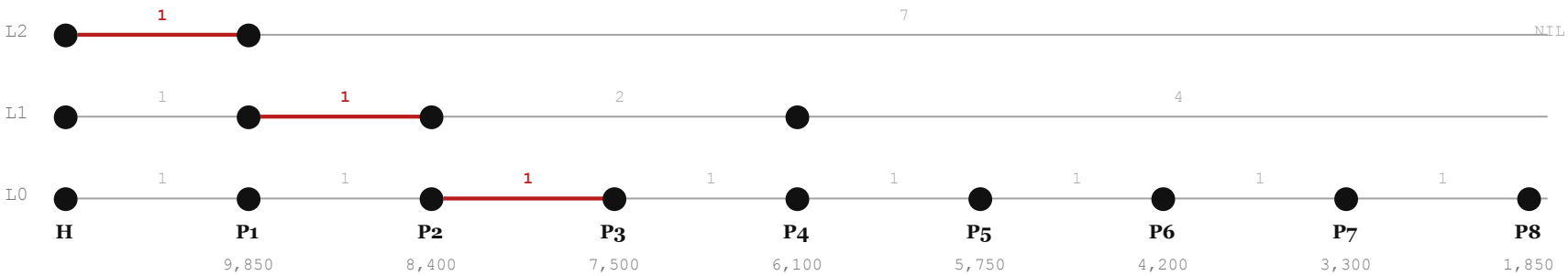
THE TRACE

L2	`H → P1`	*9850 > 5750. Take.*	`+ 1`	**`rank = 1`**
L2	`drop`	*P1.Forward[2] = NIL.*	`—`	**`rank = 1`**
L1	`P1 → P2`	*8400 > 5750. Take.*	`+ 1`	**`rank = 2`**
L1	`P2 → P4`	*6100 > 5750. Take.*	`+ 2`	**`rank = 4`**
L1	`drop`	*P4.Forward[1] = NIL.*	`—`	**`rank = 4`**
L0	`P4 → P5`	*P5 is the target. Arrived.*	`+ 1`	**`rank = 5`**

4 forward moves vs. 5 for a linear scan. At N = 100,000 the ratio is ≈ 8 vs. 100,000.

Insert P9, score 7000.

Coin flip gives P9 height 2 (pointers at L0 and L1). First, find the spot. Track update[] and rank[].



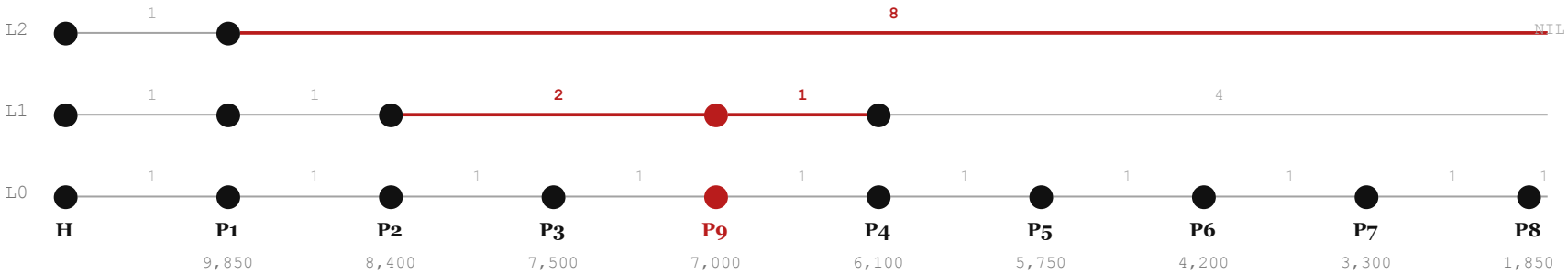
THE WALK

L2	`H → P1.`	*9850 > 7000. Move. rank[2] = 1.*	**update[2] = P1**
L2	`P1.Forward[2] = NIL.`	*Drop. rank[1] = 1.*	
L1	`P1 → P2.`	*8400 > 7000. Move. rank[1] = 2.*	
L1	`P2 → P4 too far.`	*6100 < 7000. Stop. update[1] = P2. Drop. rank[0] = 2.*	**update[1] = P2**
L0	`P2 → P3.`	*7500 > 7000. Move. rank[0] = 3.*	
L0	`P3 → P4 too far.`	*6100 < 7000. Stop.*	**update[0] = P3**

After the walk: update = [P3, P2, P1] rank = [3, 2, 1] (indexed by level).

Splice P9 in, fix the spans.

Two cases. At levels where P9 has a pointer: split the old span. Above P9's height: + 1.



THE ARITHMETIC

L0 *P9 has a pointer here.*

```
P9.Span[0] = P3.Span[0] - (rank[0]-rank[0]) = 1 - 0 = 1.
P3.Span[0] = (rank[0]-rank[0]) + 1 = 1.
```

L1 *P9 has a pointer here.*

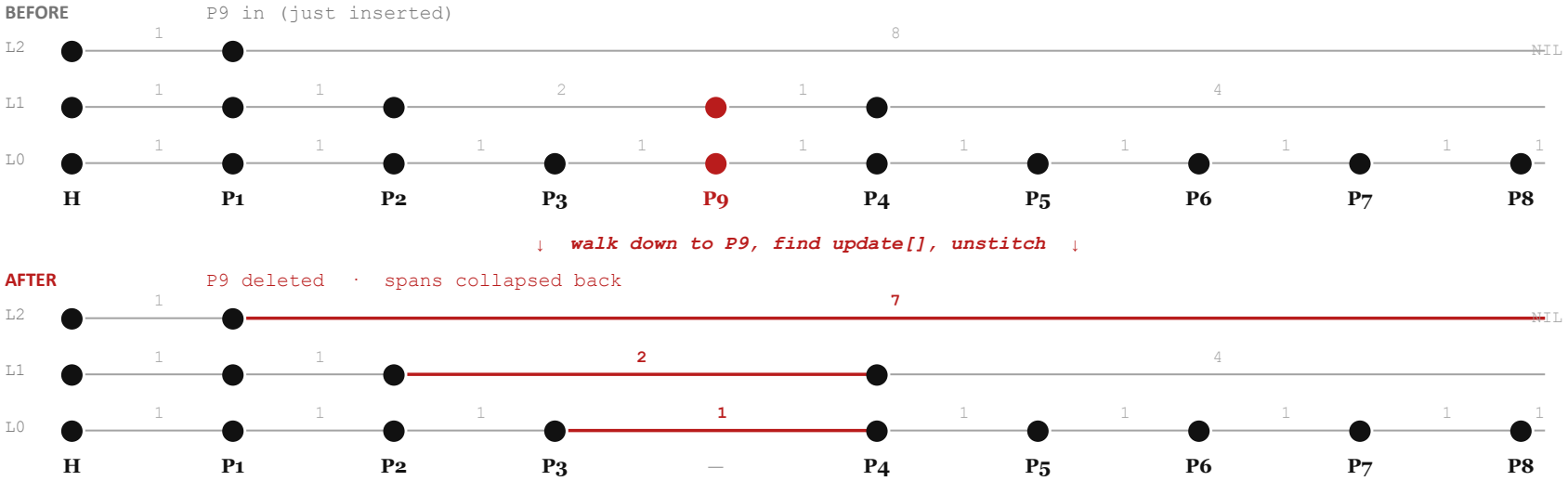
```
P9.Span[1] = P2.Span[1] - (rank[0]-rank[1]) = 2 - 1 = 1.
P2.Span[1] = (rank[0]-rank[1]) + 1 = 2.
```

L2 *Above P9's height.*

```
P1.Span[2] += 1.    7 → 8.
(One more L0 node now passes under this arrow.)
```

Delete P9: walk and unstitch.

Same walk as insert. Splice neighbors back together. Spans collapse instead of splitting.



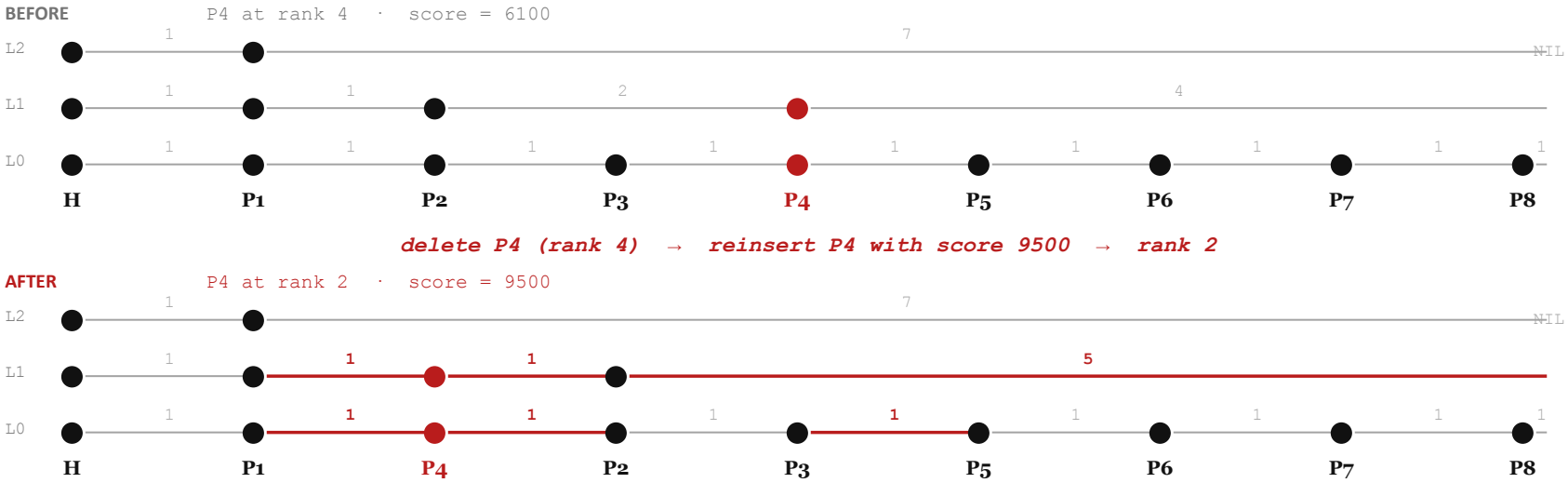
DELETE RULES

At levels where node had a pointer: `prev.Span = prev.Span + node.Span − 1.` L0: 1+1−1 = 1. L1: 2+1−1 = 2.

At levels above node's height: `prev.Span −= 1.` L2: 8 → 7.

Score update: P4 climbs from rank 4 to rank 2.

Not in-place. Delete from the old position + insert at the new one. Each phase is O(log N) — no slice resort.



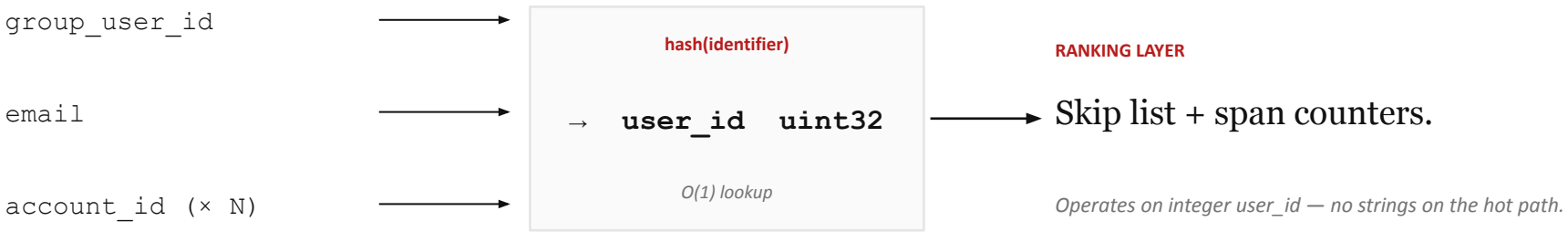
Two phases. *Walk + splice (delete), then walk + splice (insert). O(log N) each.*

Height reused. *P4 keeps its old height (2). No new coin flip — saves an allocation.*

No-op shortcut. *If new score == old score, skip both phases. Just update payload in place.*

Pull identity out of ranking.

Same player, multiple identifiers. The original code matched on ID OR email — duplicates everywhere.



Identity is an architectural concern, not a coding problem.

Two layers, two contracts. The skip list doesn't care that a user has 15 email addresses — it sees one uint32.

Three layers of testing.

In a regulated environment, you can't deploy a custom data structure and hope.

---

01 Unit tests

FAST · EVERY COMMIT

After every insert/delete, assert: spans correct, forward pointers consistent, base level sorted, size matches count.

02 Integration tests

BEHAVIOR · PRE-MERGE

Identity + ranking together. Multiple identifiers per user. Out-of-order updates. Staging trace replays.

03 Manual + perf

STAGING · PRE-SHIP

Tournament-scale scenarios replayed against staging. Benchmarks across N = 1k / 10k / 100k — validated the O(log N) asymptote, not just correctness.

Flat line for the new implementation. Exponential curve for the old. Shipped.

PART 05

Production impact.

Two days of work. One hundred times faster.

What changed when we deployed.

METRIC	BEFORE	AFTER	Δ
Concurrent users sustained	~60k (crashing)	**100k+ (stable)**	*10× capacity*
Update latency @ 10k users	50–100 ms	**< 20 ms**	*≈ 5×*
CPU under +10% load	100% / crash	**30–40% stable**	*headroom*
OOM incidents	multi / day	**0**	*since deploy*
Time to deliver	≥ 2 weeks*	**2 days**	**Redis path*

The system became

boring.

Boring is the highest praise an engineer can give a system.

- *Operators stopped seeing degradation during their busy hours.*
- *Compliance stopped seeing duplicate entries.*
- *On-call stopped getting paged at night for leaderboard CPU.*

PART 06

Lessons we learned

Five things to take with you.

For senior engineers in 2026.

01 The standard answer is not always the right answer.

Redis is excellent. It was the wrong tool here. Senior judgment is knowing the difference.

02 Custom data structures are not a 2010s relic.

300 well-tested lines beat "use a service" when the library doesn't fit your access pattern.

03 Profile before you optimize.

We were wrong about the hot path on at least two theories. Without profiling, nothing would have changed.

Thank you.

Questions?

Ivan Balashov · Tech Lead, SoftSwiss

Further reading.

Pugh, W. (1990). Skip Lists: A Probabilistic Alternative to Balanced Trees.

`Communications of the ACM`

Redis Documentation. Sorted Sets.

`redis.io/docs/data-types/sorted-sets/`

LevelDB. MemTable and Skip Lists.

`github.com/google/leveldb`

Herlihy & Shavit. The Art of Multiprocessor Programming.

`Skip Lists chapter`