Intro to LSM & B-Trees for Devs & DevOps

adyen

engineered
for ambition

Quick Intro to DB Storage (Devs)

Data Page (aka Heap) - UNORDERED

Data
Pages

Row Structure:

Primary Key

Field 1

Field 2

Field 3

...

B-tree Pages - ORDERED

PK

Secondary Indexes

B-Tree

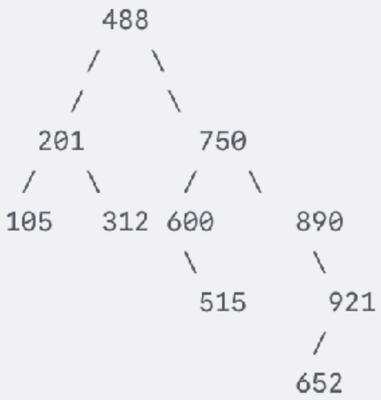
Leaf Structure (array)

(Key, DataPage Pointer)

Regular binary tree

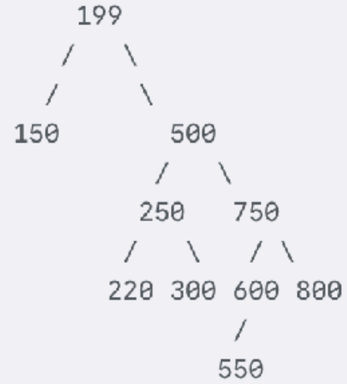
Example 1

Numbers: 488, 201, 750, 105, 312, 600, 890, 515, 652, 921



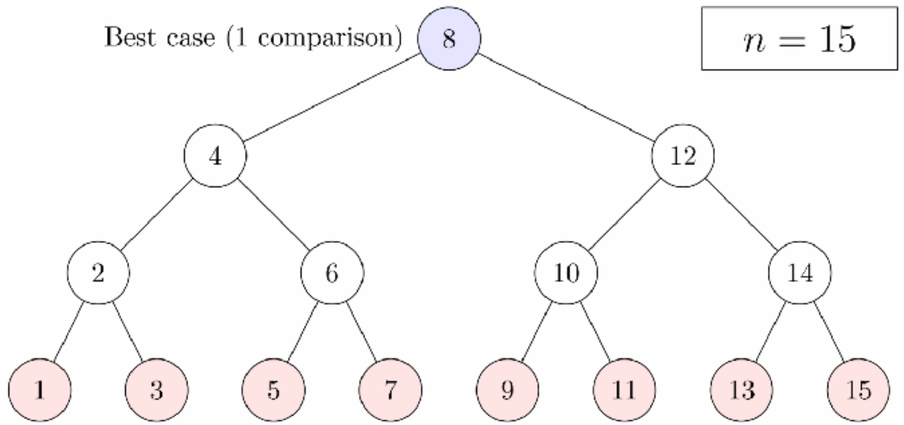
Example 2

Numbers: 199, 150, 500, 250, 750, 220, 300, 600, 800, 550



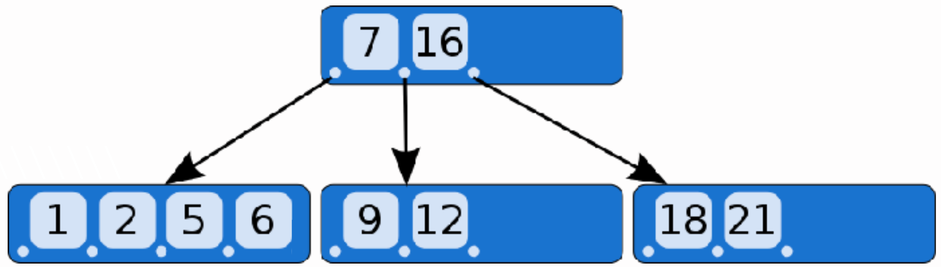
1970s balance tree fan-out (flat)

Binary-search tree



Worst case
$\lfloor \log_2(n) + 1 \rfloor$
comparisons)

Balanced Tree (f = fanout)



$\log_2(n)$ vs $\log_f(n)$
f = 4 or 200?

What The Fanout?

Imagine a tree whose canopy is the world itself! (F=200)

1. Fanout and Exponential Growth

- Suppose each B-tree node can branch to **F** children (the fanout).
- At depth 1 (the root), the tree covers at most **F** entries.
- At depth 2, it covers **$F \times F = F^2$** entries.
- At depth 3, it covers **$F \times F \times F = F^3$** entries.
- In general, at depth *d*, the tree can cover up to **F^d entries**.
- That's **exponential growth** — every time you add a level, the capacity multiplies.

How Deep Is My Balanced Tree?

Now some really fun maths (optional) with logs to base 200 (i.e. F=200)

2. Logarithms and Tree Height

- If you know the number of rows NN and fanout FF, you can ask:
  How many levels (depth) does the tree need to hold N rows?

  - Solve for *d* in:

$$F^d \geq N$$

  Take logarithms:

$$d \geq \log_F(N)$$

- That's why lookups in a B-tree are **O(log n)**: the tree height grows only logarithmically with the number of rows.

Postgres Example

Real-world application: One billion rows, just four levels of depth!

- Assume:
  - Fanout ≈ **200** (typical for 8 KB pages with small keys).
  - Rows indexed: **1 billion (10^9)**.
- Tree depth needed:

$$d \approx \log_{200}(10^9)$$

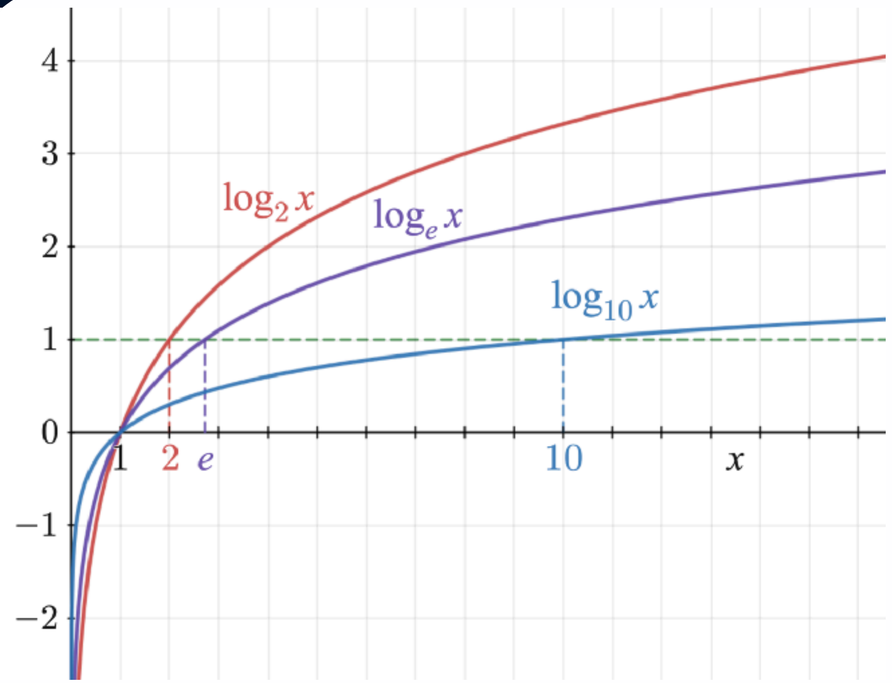
- Change base:

$$d = \frac{\log_{10}(10^9)}{\log_{10}(200)} = \frac{9}{2.3} \approx 4$$

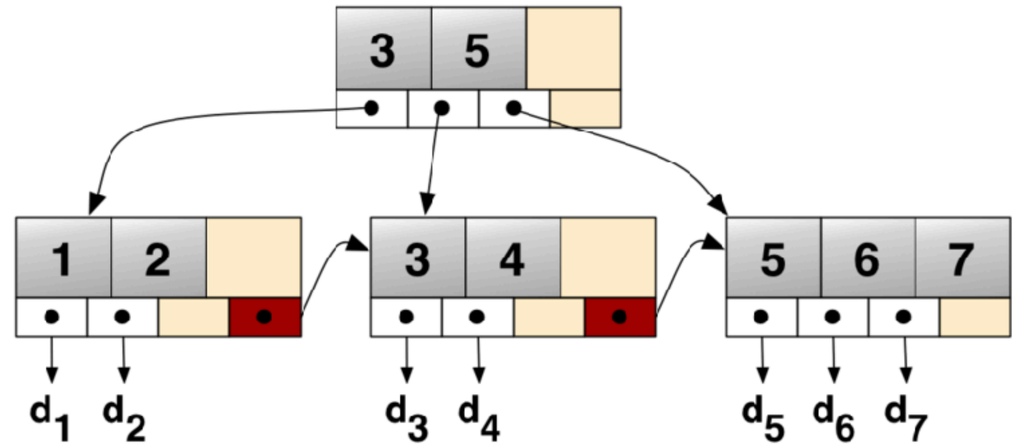
- Just **4 levels deep**!
  That means even with a **billion rows**, a lookup requires at most 4 page reads (root → internal → leaf → row).

B+ trees - double linked list

$\log_2(x)$ vs $\log_f(x)$



B+Tree



So what are B minus trees?

Mythical! A bit of DBA humor ...

1. **Q:** Why did the B-minus tree fail its data structures exam?
   A: Because it could never find the right balance and its performance was just barely passing.
2. A B+ Tree and a B-minus Tree are in a database. The B+ Tree says, "I love how efficiently I can retrieve data ranges with my linked leaves!"
   The B-minus Tree sighs and says, "I keep all my data in a sorted array and just hope no one asks for anything from the middle."
3. **Q:** What's the time complexity of a search operation in a B-minus tree?
   A: In theory, it's $O(\log n)$. In practice, it's $O(\text{procrastination})$.

Btree Reads-per-Select

Scale Factor 4 (400,000 rows) has 4 Read

```
bench_ff100_sf4=# explain (analyze,buffers) select * from pgbench_accounts where aid = 32928;
                                                          QUERY PLAN
-------------------------------------------------------------------------------------------------------------------------------
 Index Scan using pgbench_accounts_pkey on pgbench_accounts  (cost=0.42..8.44 rows=1 width=97) (actual time=0.527..0.528 rows=1 loops=1)
   Index Cond: (aid = 32928)
   Buffers: shared read=4
 Planning:
   Buffers: shared hit=62 read=8 dirtied=1
 Planning Time: 17.107 ms
 Execution Time: 0.573 ms
(7 rows)
```

Btree Reads-per-Select

Scale Factor 8 (800,000 rows) has 4 Read

```
bench_ff100_sf8=# explain (analyze,buffers) select * from pgbench_accounts where aid = 32928;
                                                          QUERY PLAN
-------------------------------------------------------------------------------------------------------------------------------
 Index Scan using pgbench_accounts_pkey on pgbench_accounts  (cost=0.42..8.44 rows=1 width=97) (actual time=0.709..0.711 rows=1 loops=1)
   Index Cond: (aid = 32928)
   Buffers: shared read=4
 Planning:
   Buffers: shared hit=62 read=8 dirtied=3
 Planning Time: 10.082 ms
 Execution Time: 0.978 ms
(7 rows)
```

Btree Reads-per-Select

Keep on doubling

bench_ff100_sf16, bench_ff100_sf32, bench_ff100_sf64 ….

Btree Reads-per-Select

Scale Factor 512 (5,120,000 rows) has 5 Read

```
bench_ff100_sf512=# explain (analyze,buffers) select * from pgbench_accounts where aid = 32928;
                                                          QUERY PLAN
-------------------------------------------------------------------------------------------------------------------------------
 Index Scan using pgbench_accounts_pkey on pgbench_accounts  (cost=0.56..8.58 rows=1 width=97) (actual time=0.080..0.083 rows=1 loops=1)
   Index Cond: (aid = 32928)
   Buffers: shared hit=5
 Planning Time: 0.256 ms
 Execution Time: 0.212 ms
(5 rows)
```

DB Storage Engine (DevOps)

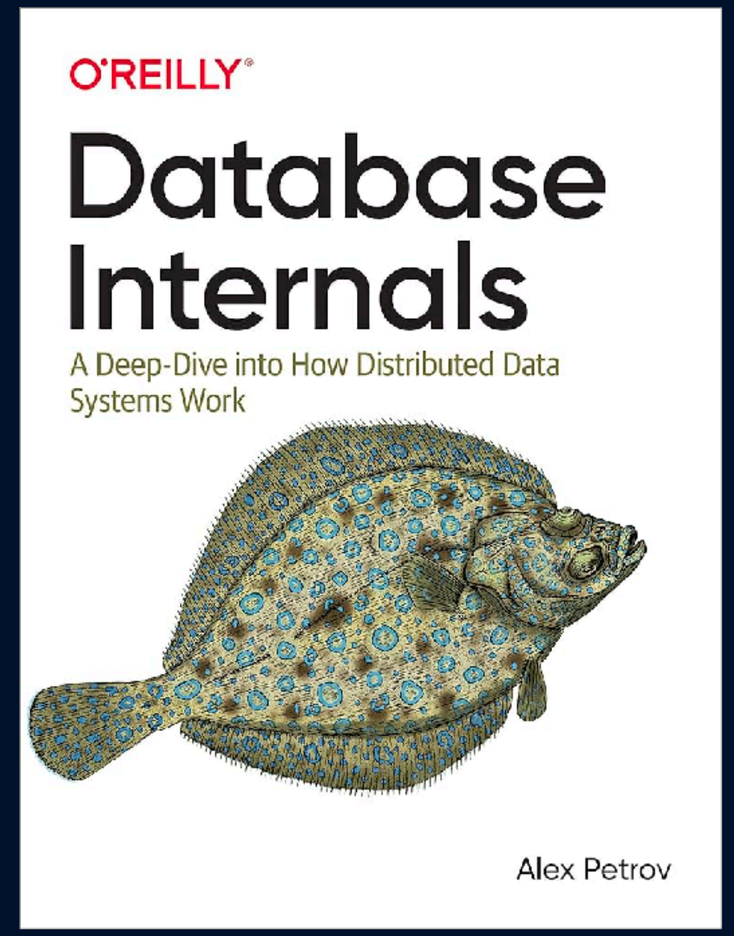
Heap (Un-Ordered)

LSM (Ordered)

Immutable Log Structures

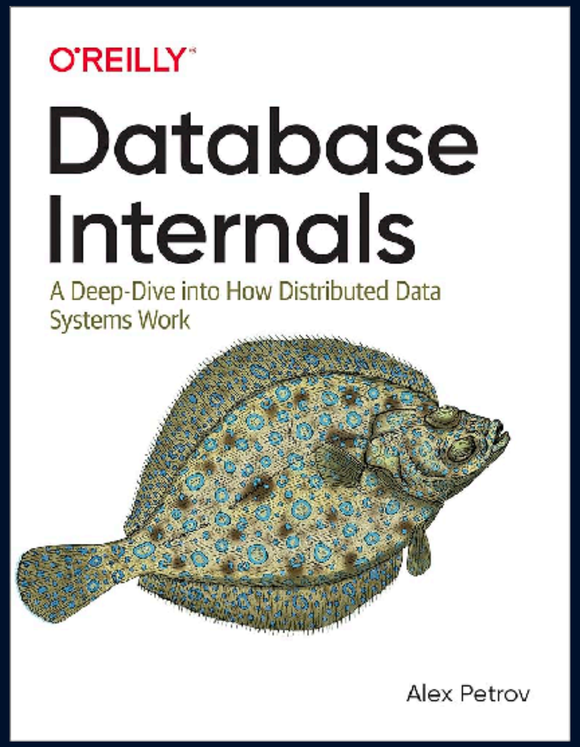
"Accountants don't use erasers or they end up in jail"

Key	Value
K1	AAA,BBB,CCC
K2	AAA,BBB
K3	AAA,DDD
K4	AAA,2,01/01/2015
K5	3,ZZZ,5623



Database Internals

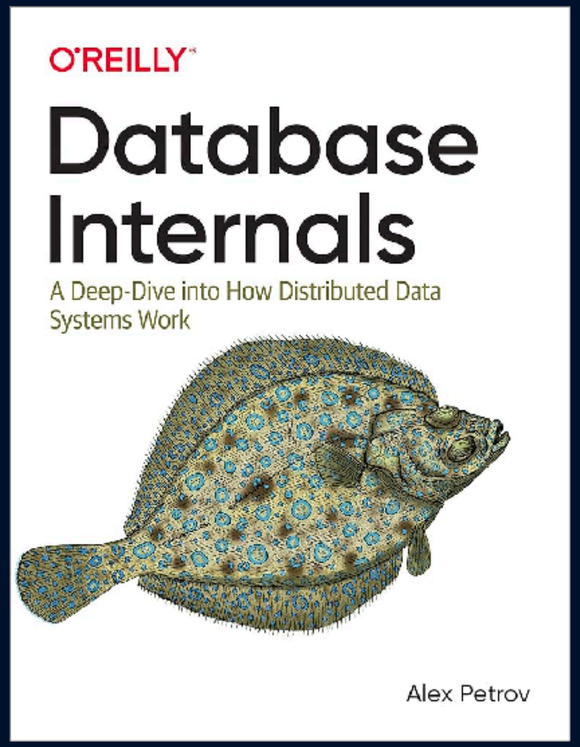
"A very fine line between indexes and tables”



- SE Radio 417: Alex Petrov: Distributed Databases

Database Internals

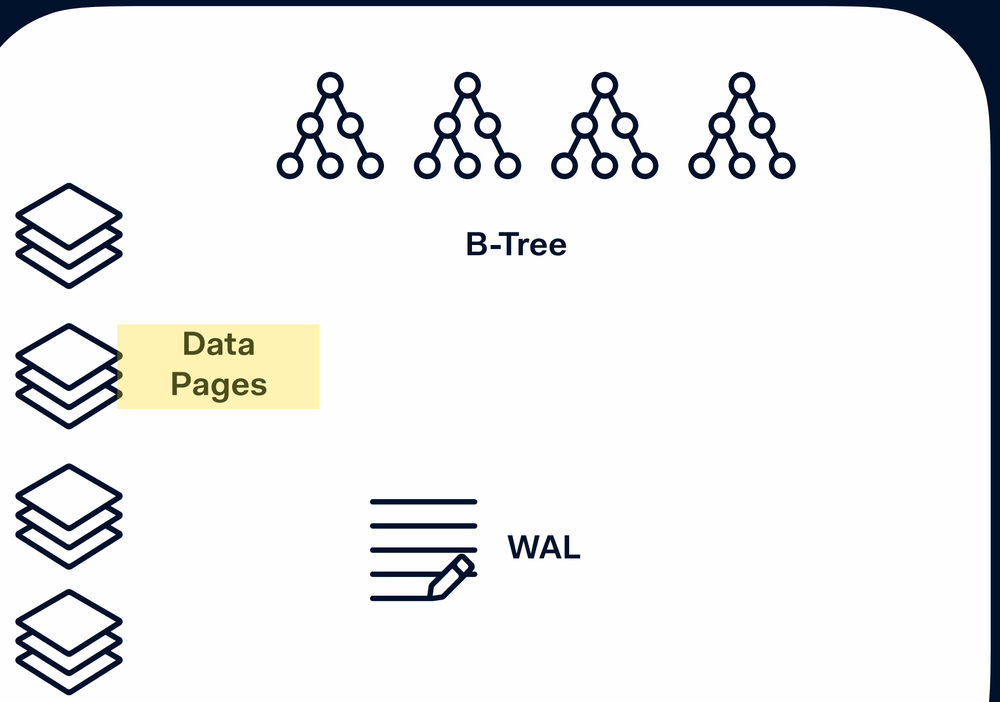
"A very fine line between indexes and tables”
"b-trees are heavily read optimized & lsm-trees are heavily write optimized"



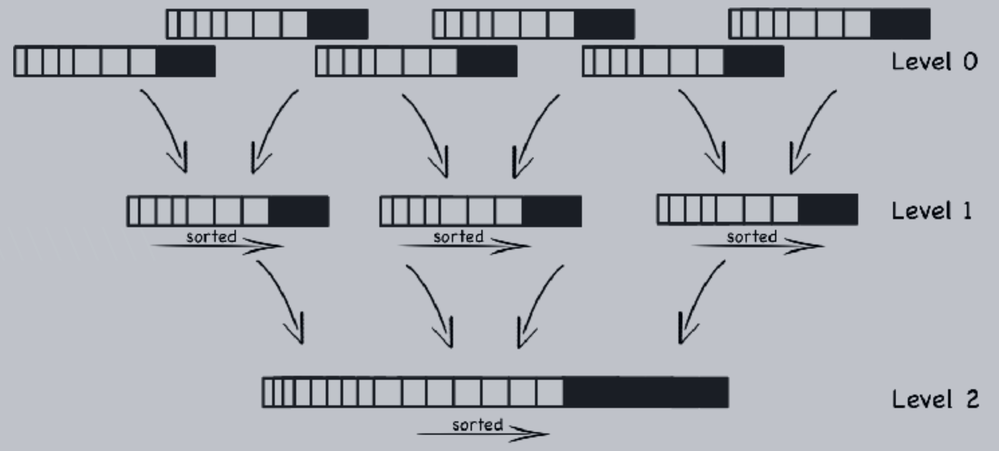
- SE Radio 417: Alex Petrov: Distributed Databases

DB Storage Engine (DevOps)

Paged (aka Heap)



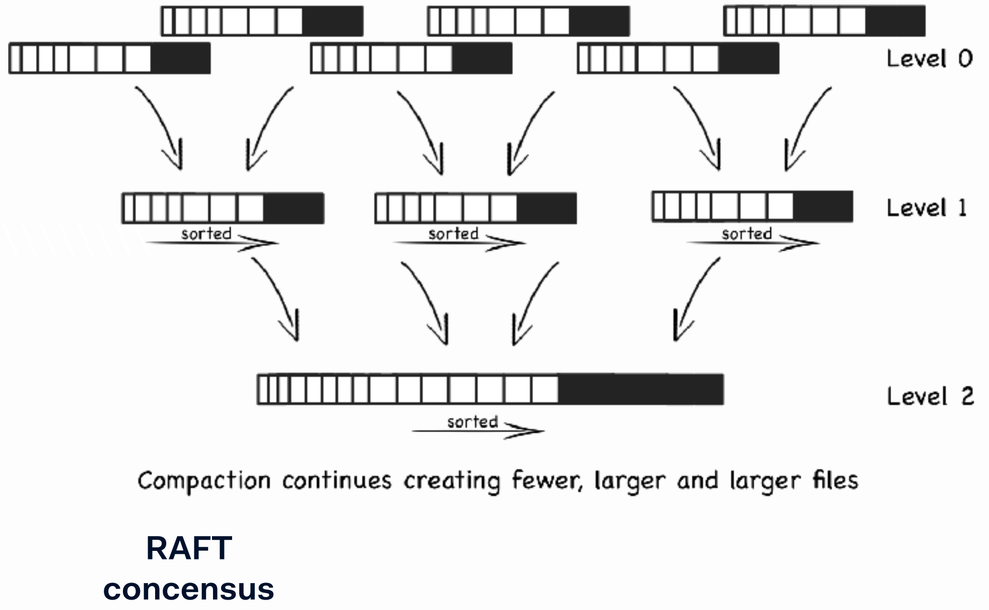
Log Based (key-value)



DB Storage Engine (DevOps)

Paged (aka Heap)

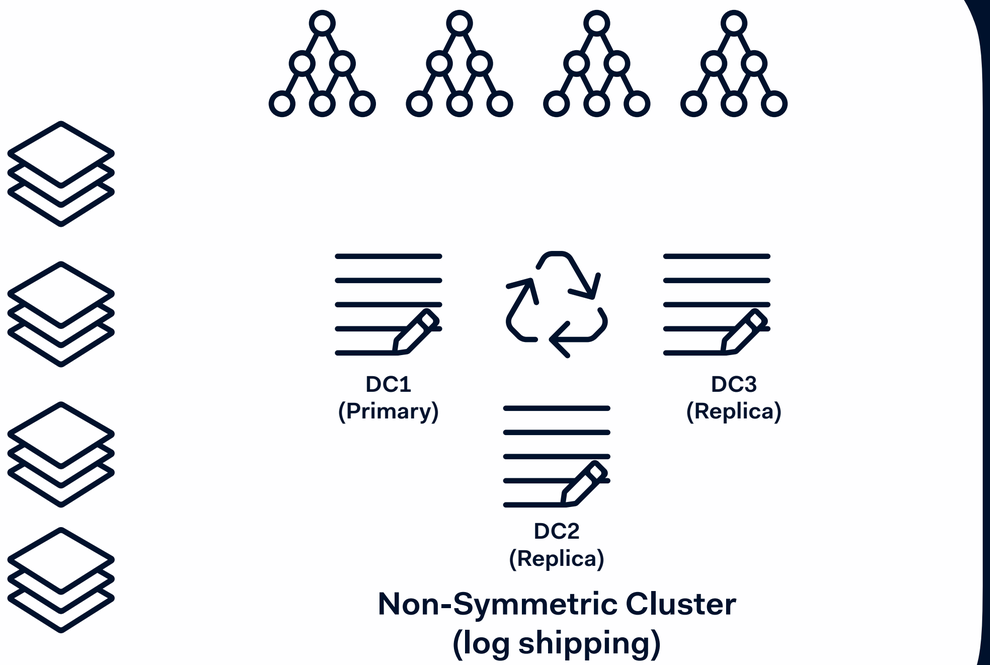
Log Based (key-value)



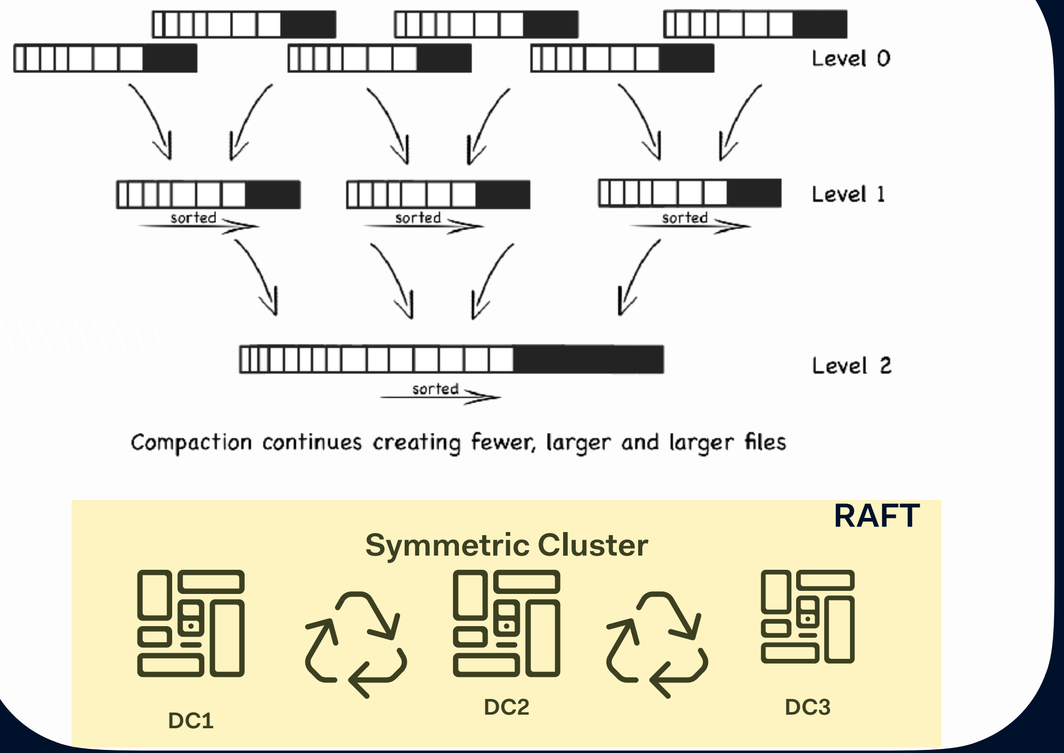
RAFT
concensus

Replication Models

Paged (aka Heap)



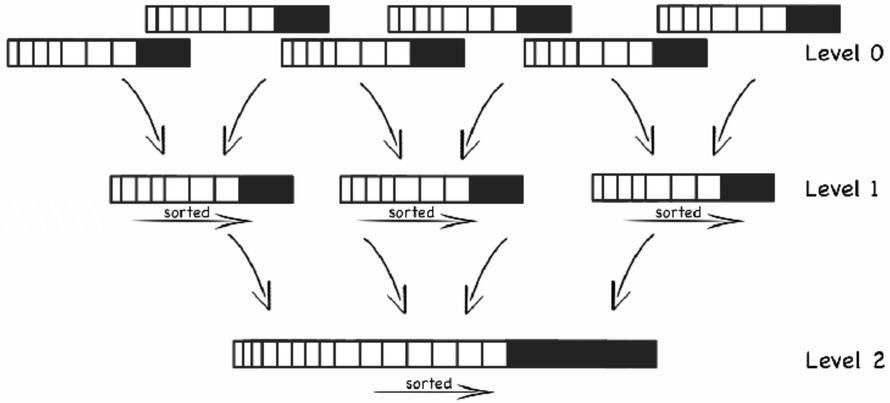
Log Based (Quorum)



Visualizing LSM trees

Pyramid Build Blocks

SSD table - triangulate



Implementations - old & new

Paged/Heap (old world)

Postgres 1996

SQL Server 1988

Oracle 1979

LSM (new world)

"Postgres Compatible"
(last few 5-10 years)

pgrocks
(fdw)

Stream broker 2015

RocksDB
Facebook
(2012 MySQL)

Google
BigTable
(Memtable
2008)

Google
Chrome
(Memtable
2008)

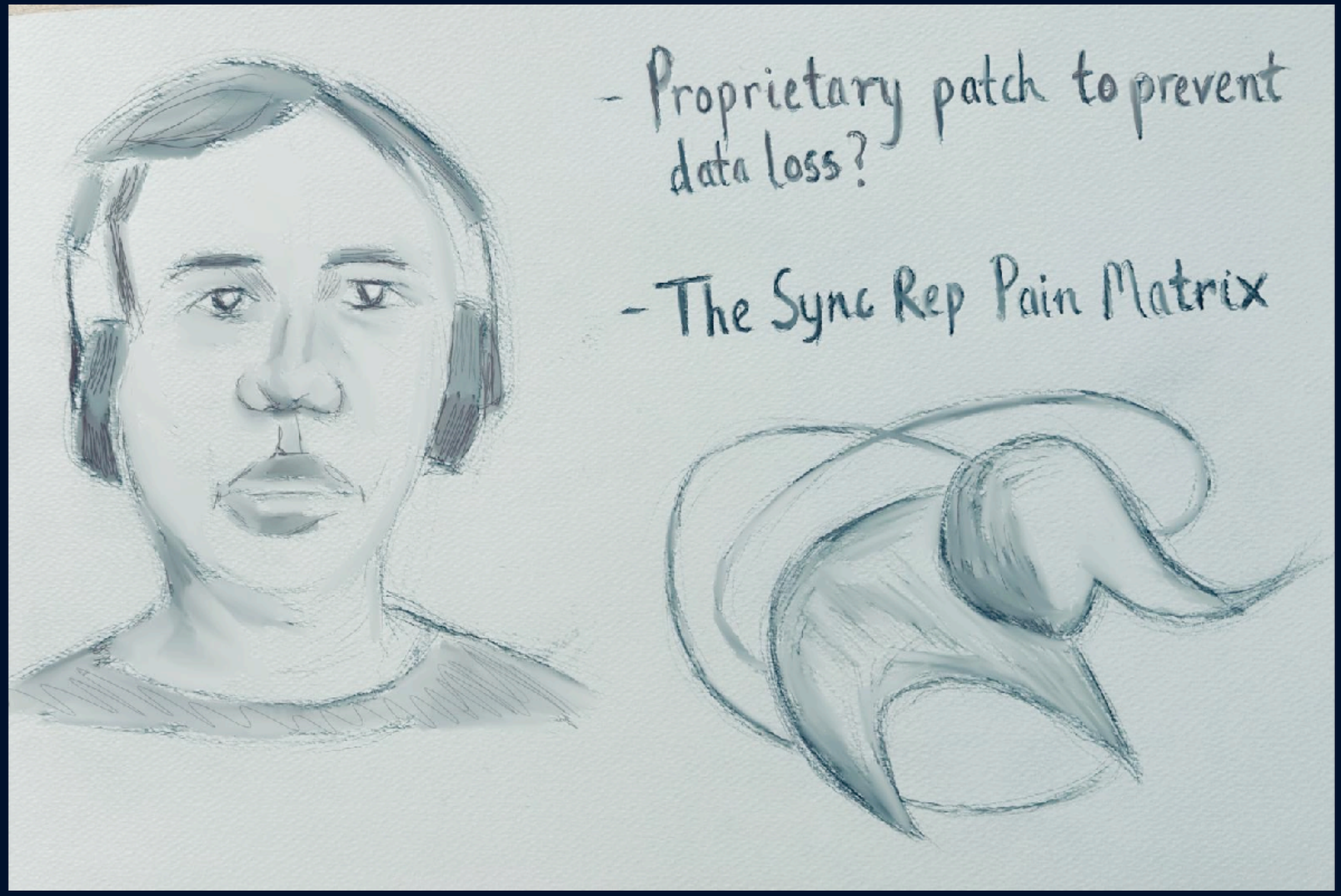
Facebook
(2007)

Flink LSM-tree synergy?

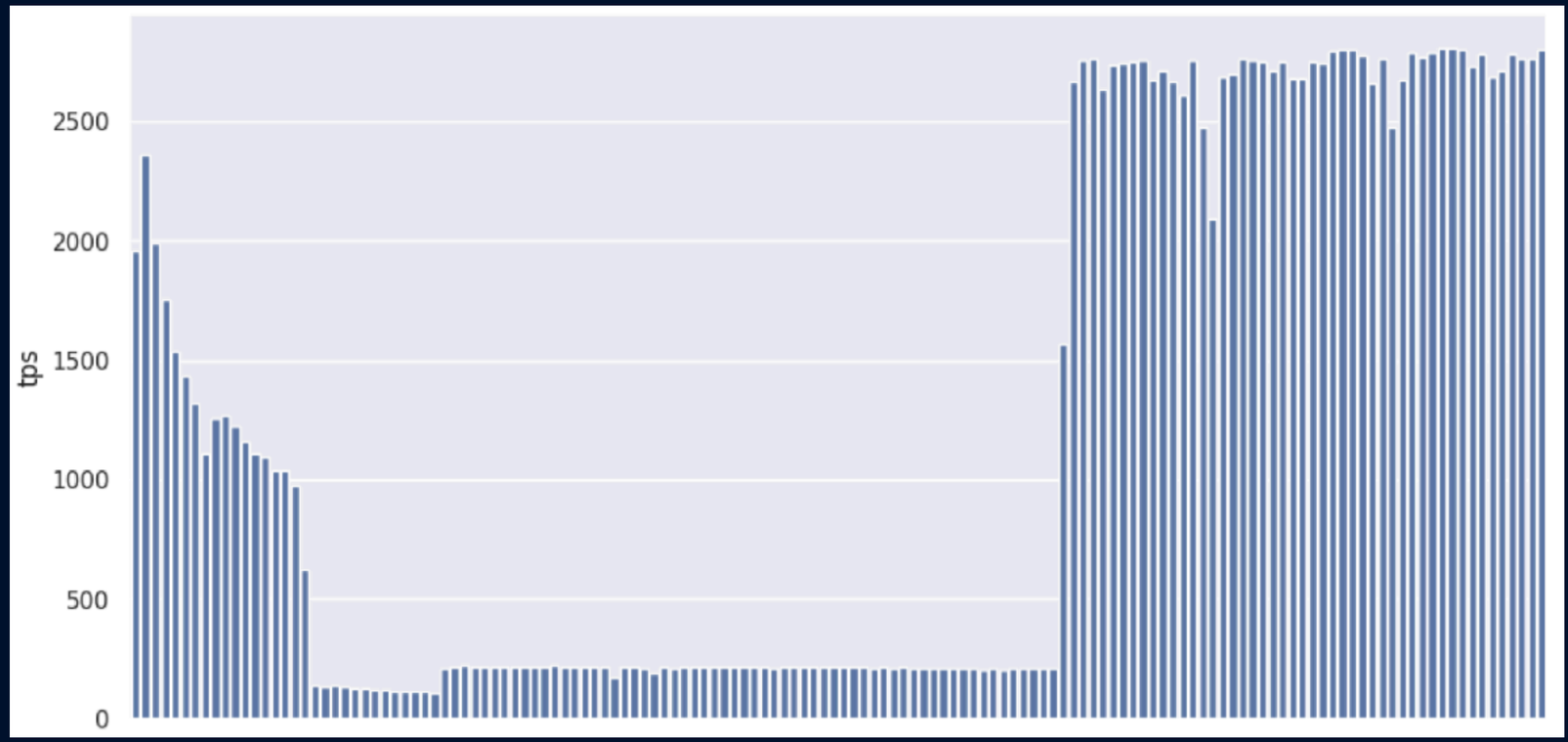
Flink's state access pattern is *append + point lookup* dominated:

Operation type	Frequency	Pattern
INSERT / PUT	Very frequent	New keys (e.g., new user/session/window).
UPDATE	Frequent but small	Overwrites of existing keys (e.g., count += 1). LSM handles this by appending a new version — not in-place modification.
DELETE	Moderate	Expiring state, old windows — handled via tombstones and compaction.
READ	Moderate	Random point lookups (e.g., lookup current counter).

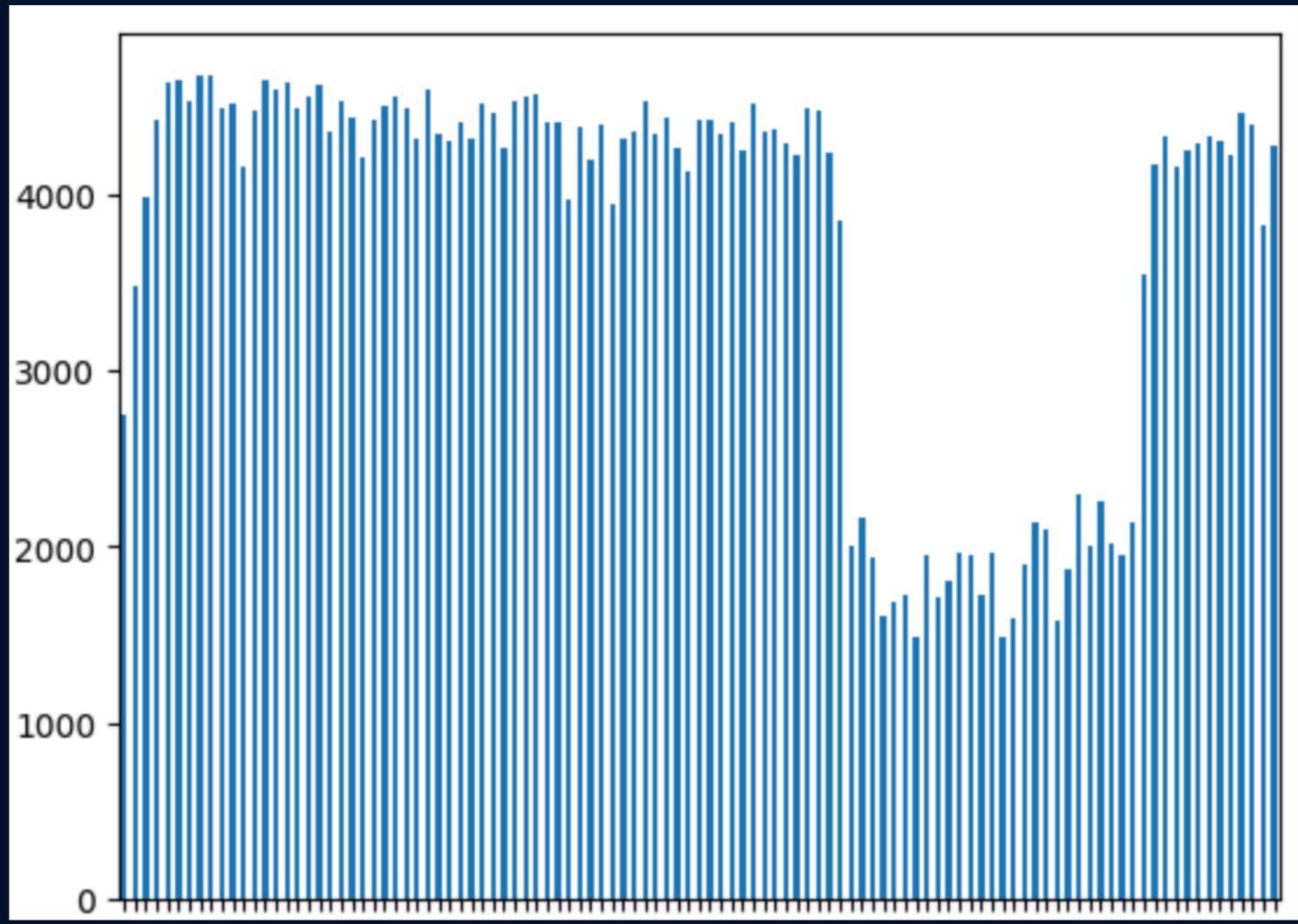
Kukushkin & PG Community Pain



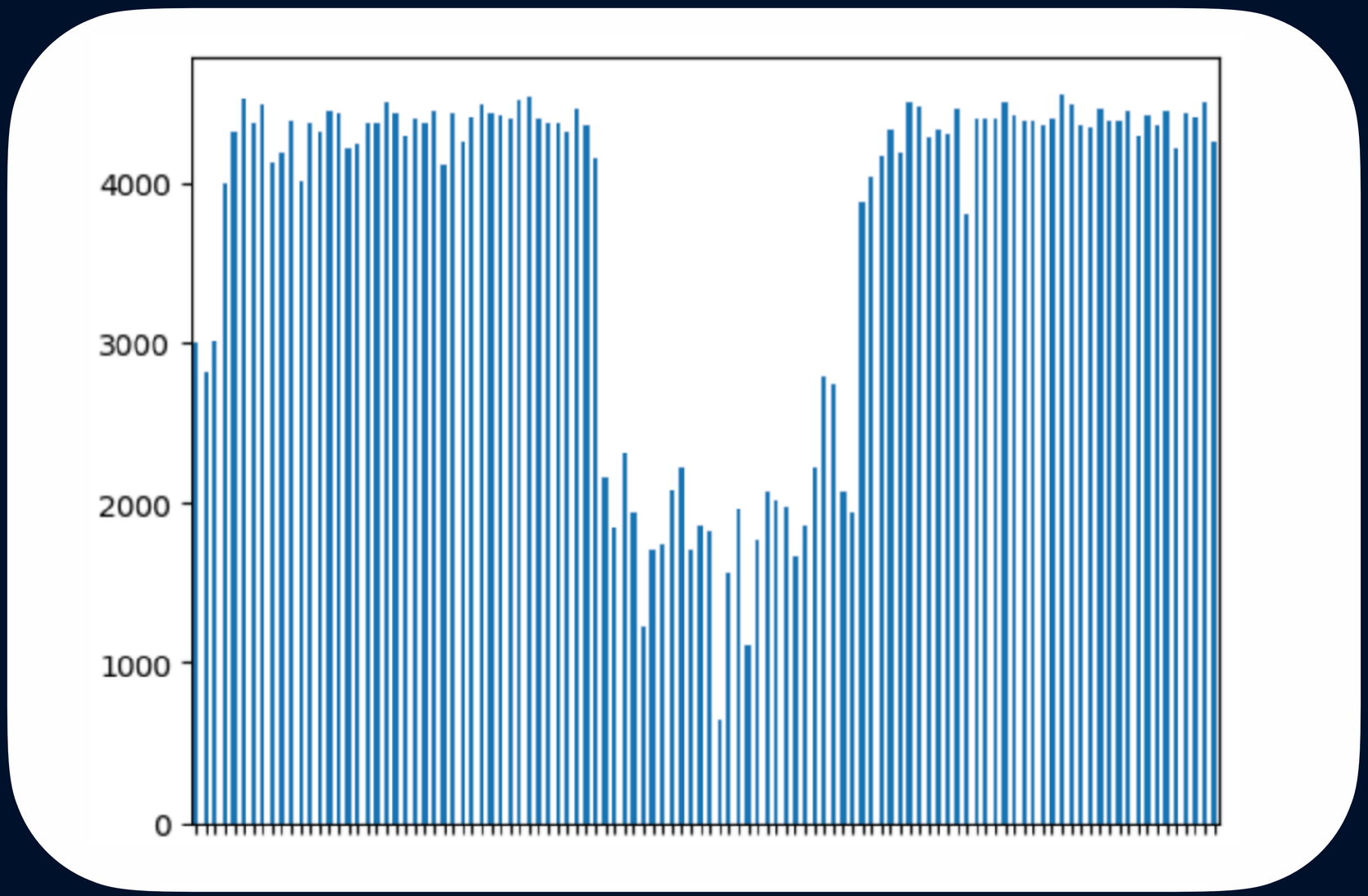
Long TXNs - pgrocks (fdw)



Drop Column - pgrocks (fdw)



Truncate Table - pgrocks (fdw)



Further Tests

* Update Heavy 0%, 20%, 40%, 80%, 100% analysis
* Hot Key ranges, frequent updated / key customers
* Very Long Duration, do not just test daily jobs but monthly and annual too!

Links

“PostgreSQL High Availability Poker” Adyen training (from pgDay Lowlands)
https://www.youtube.com/watch?v=oEjj6ofjxpo

pgDay Paris and Amsterdam Open Source Data Infra meetup
https://www.youtube.com/watch?v=TMKKZwWYY6A

Microsoft Posette “Myths and Truths about Synchronous Replication in PostgreSQL” Alexander Kukushkin
https://www.youtube.com/watch?v=PFn9qRGzTMc

Software Engineering Radio E417 - Alex Petrov on LSMs trees (contributor to Cassandra)
https://se-radio.net/2020/07/episode-417-alex-petrov-on-database-storage-engines/

Software Engineering Radio E417 - Eric-Brewer The-CAP-Theorem Then and Now
https://www.youtube.com/watch?v=_zHJAvq8xIA

Q & A

https://github.com/dgapitts/pgday-paris-btree-lsm-demo

adyen

engineered
for ambition