DB Storage Structures

The old (B-tree & Columnar) & the new (LSM-trees)

Agenda

Intro & visualisation of the core DB structures

Deep dive into traditional b-trees (bread & butter)

Review of columnar (still traditional)

Why LSM Trees and possible edge cases

Conclusions

Who am I?

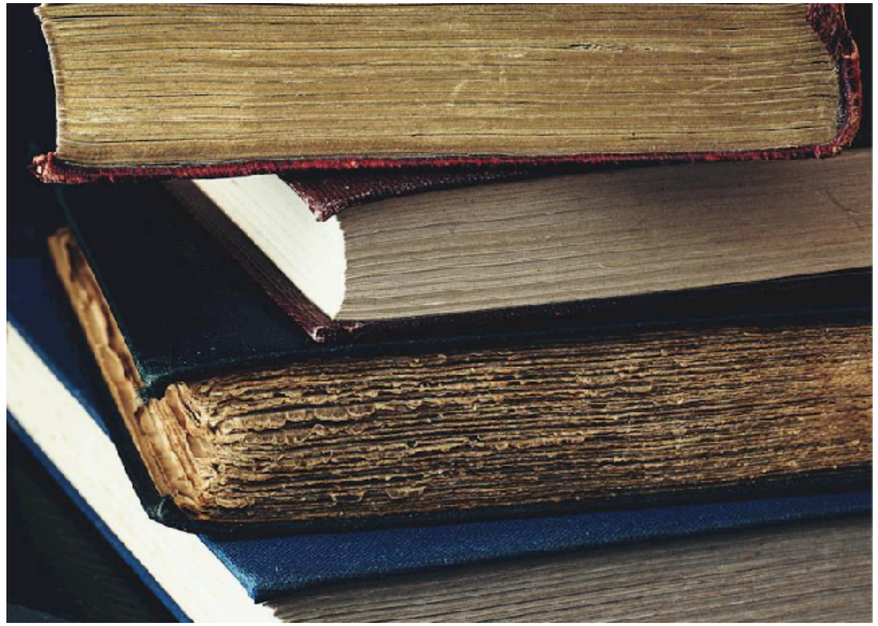
Dave Pitts - Database Engineer - Adyen

adyen

engineered
for ambition

DB Storage Engine Models

Paged (aka Heap)



Source: https://www.pexels.com/photo/stacked-books-1333742/

Log Based (key-value)

DB Storage Engine Models

Paged (aka Heap)

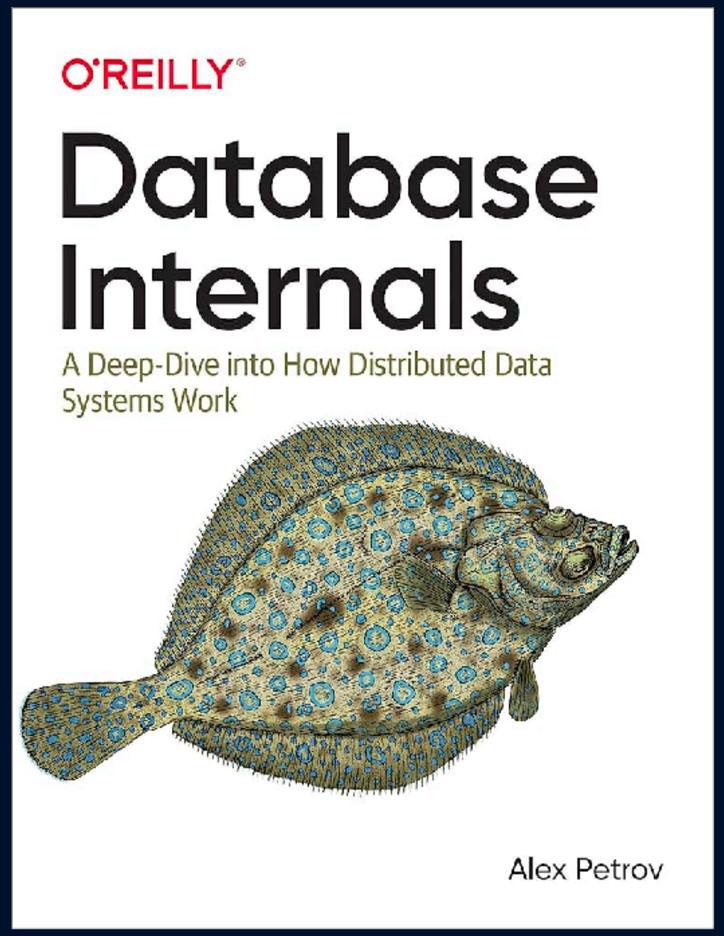
Log Based (key-value)

Source: https://www.deviantart.com/pickgameru/art/Futuristic-sci-fi-Egyptian-pyramids-1022318331

Immutable Log Structures

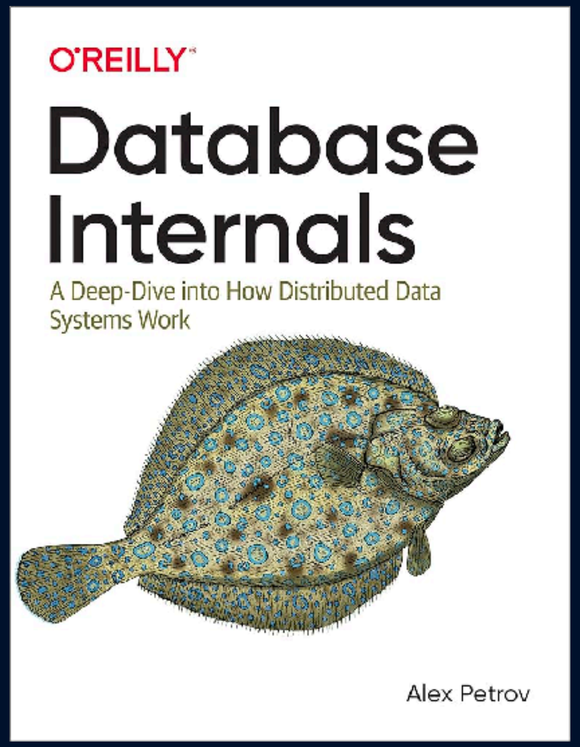
"Accountants don't use erasers or they end up in jail"

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



Database Internals

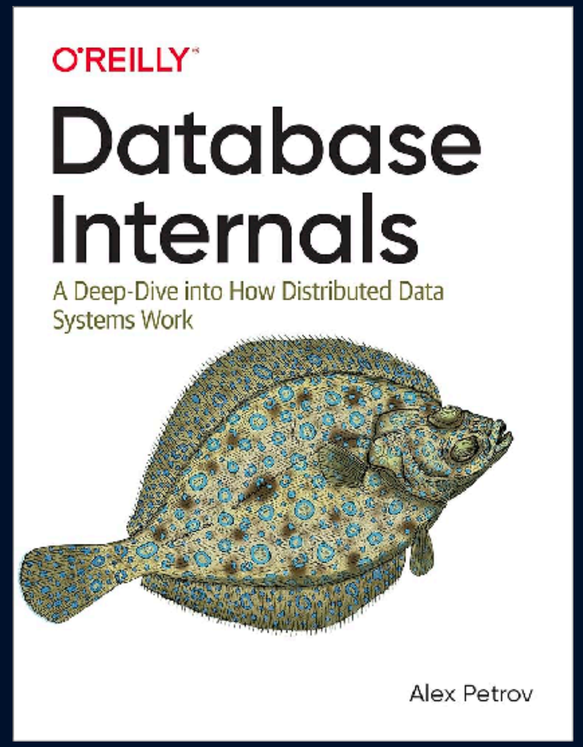
"A very fine line between indexes and tables"



- 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

Practice tops Standards

Standards

Ansi SQL

MVCC

4 Transaction Isolation Levels

Read Committed

Serializable

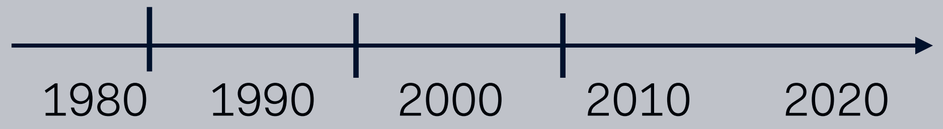
Implementation

Snapshot based:

UNDO Pages

Dead Rows

UNDO Logs



Page Locks:

Practice tops Standards

Standards

Ansi SQL

MVCC

4 Transaction Isolation Levels

Read Committed

Serializable

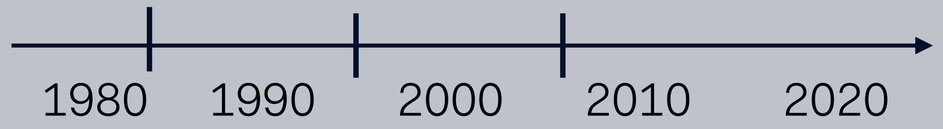
Implementation

Snapshot based:

UNDO Pages

Dead Rows

UNDO Logs



Page Locks:

Practice tops Standards

Standards

Ansi SQL

MVCC

4 Transaction Isolation Levels

Read Committed

Serializable

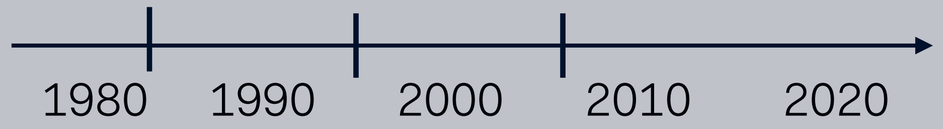
Implementation

Snapshot based:

UNDO Pages

Dead Rows

UNDO Logs



Page Locks:

Practice tops Standards

Standards

Ansi SQL

MVCC

4 Transaction Isolation Levels

Read Committed

Serializable

Implementation

Snapshot based:

UNDO Pages

Dead Rows

UNDO Logs

Page Locks:

Practice tops Standards

Standards

Ansi SQL

MVCC

4 Transaction Isolation Levels

Read Committed

Serializable

Implementation

Snapshot based:

UNDO Pages

Dead Rows

UNDO Logs

Page Locks:

Practice tops Standards

Standards

Ansi SQL

MVCC

4 Transaction Isolation Levels

Read Committed

Serializable

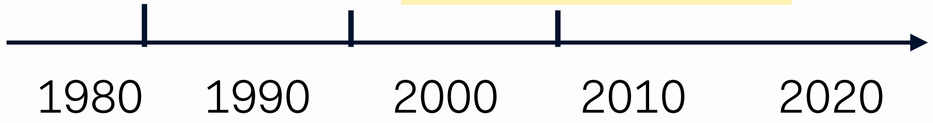
Implementation

Snapshot based:

UNDO Pages

Dead Rows

UNDO Logs



Page Locks:

Practice tops Standards

Standards

Ansi SQL

MVCC

4 Transaction Isolation Levels

Read Committed

Serializable

Implementation

Snapshot based:

UNDO Pages

Dead Rows

UNDO Logs

Page Locks:

Practice tops Standards

Standards

Ansi SQL

MVCC

4 Transaction Isolation Levels

Read Committed

Serializable

Implementation

Snapshot based:

UNDO Pages

Dead Rows

UNDO Logs

Page Locks:

Practice tops Standards

Standards

Ansi SQL

MVCC

4 Transaction Isolation Levels

Read Committed

Serializable

Implementation

Snapshot based:

UNDO Pages

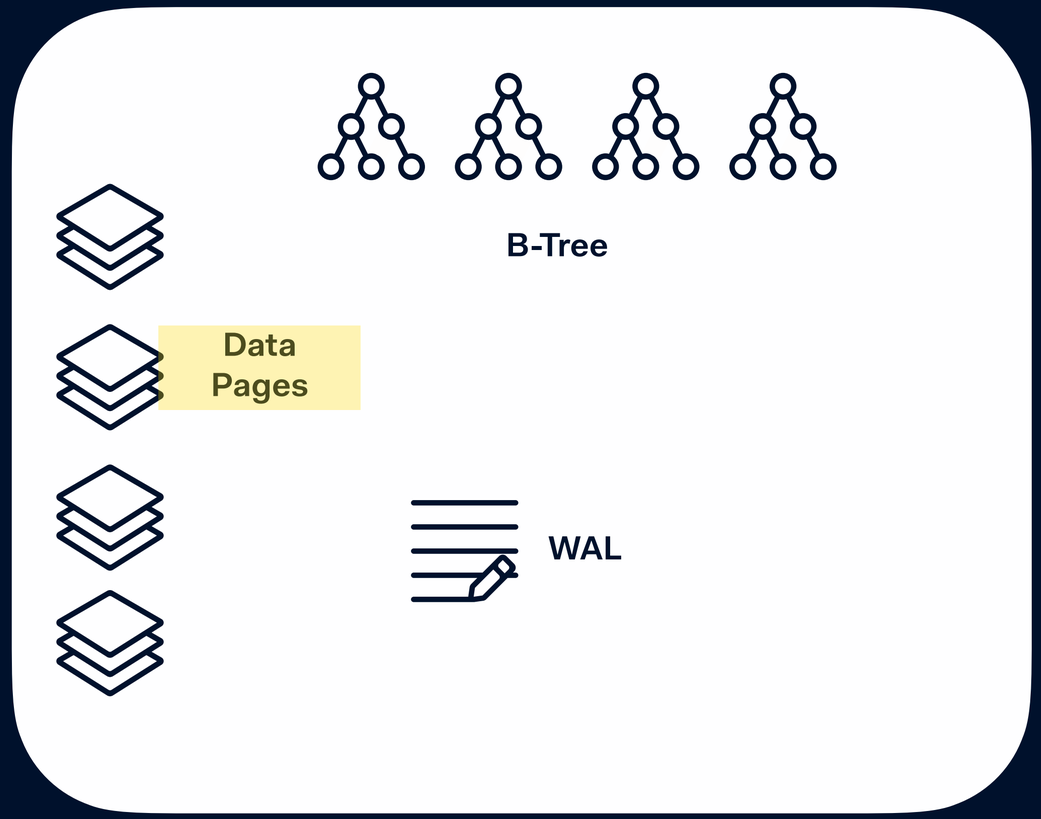
Dead Rows

UNDO Logs

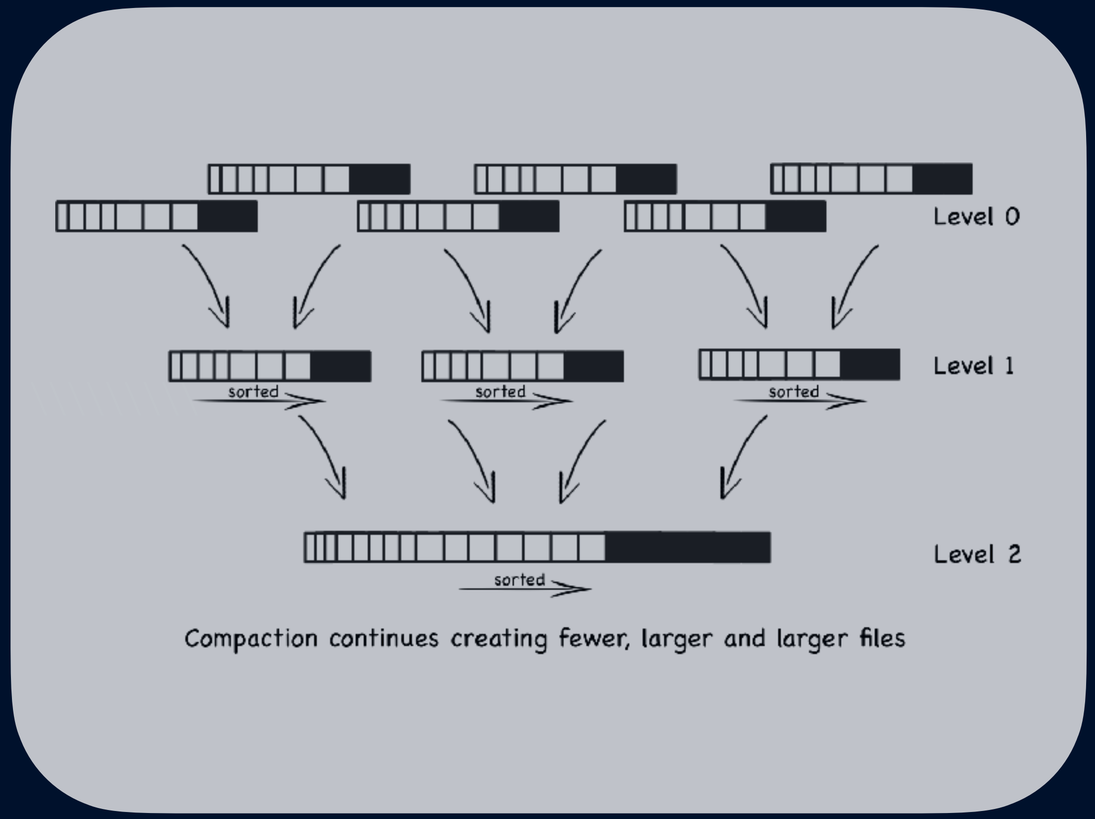
Page Locks:

DB Storage Engine Models

Paged (aka Heap)

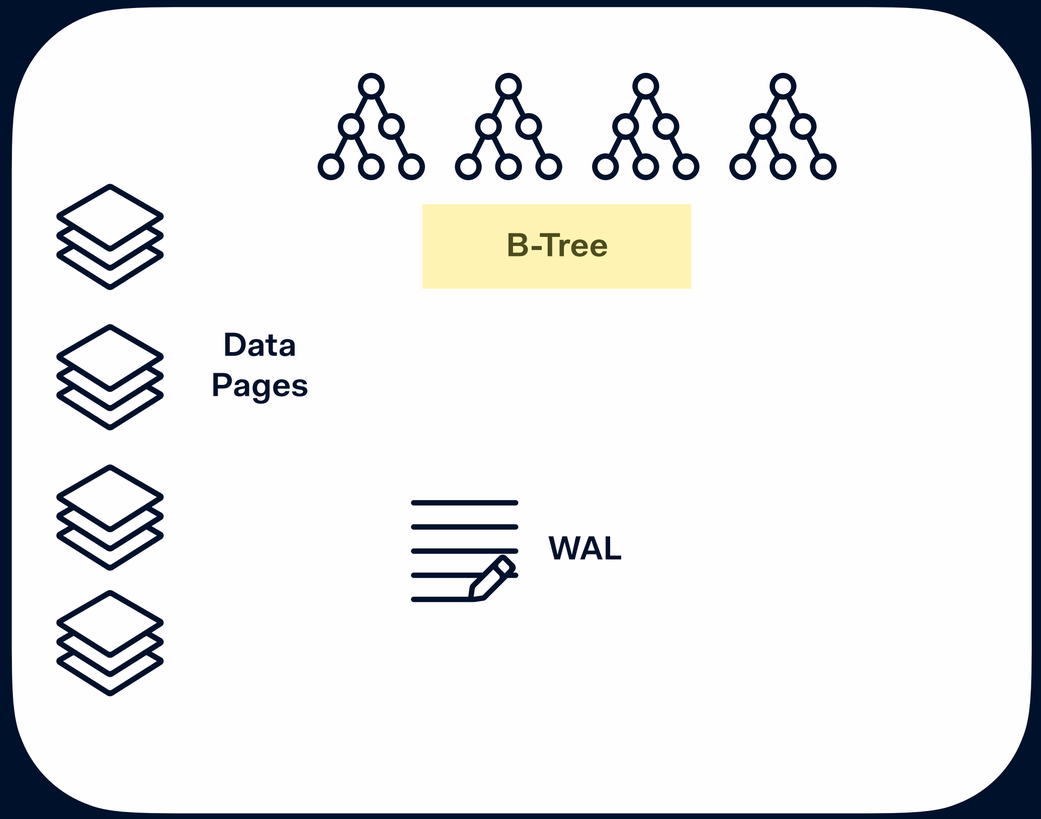


Log Based (key-value)

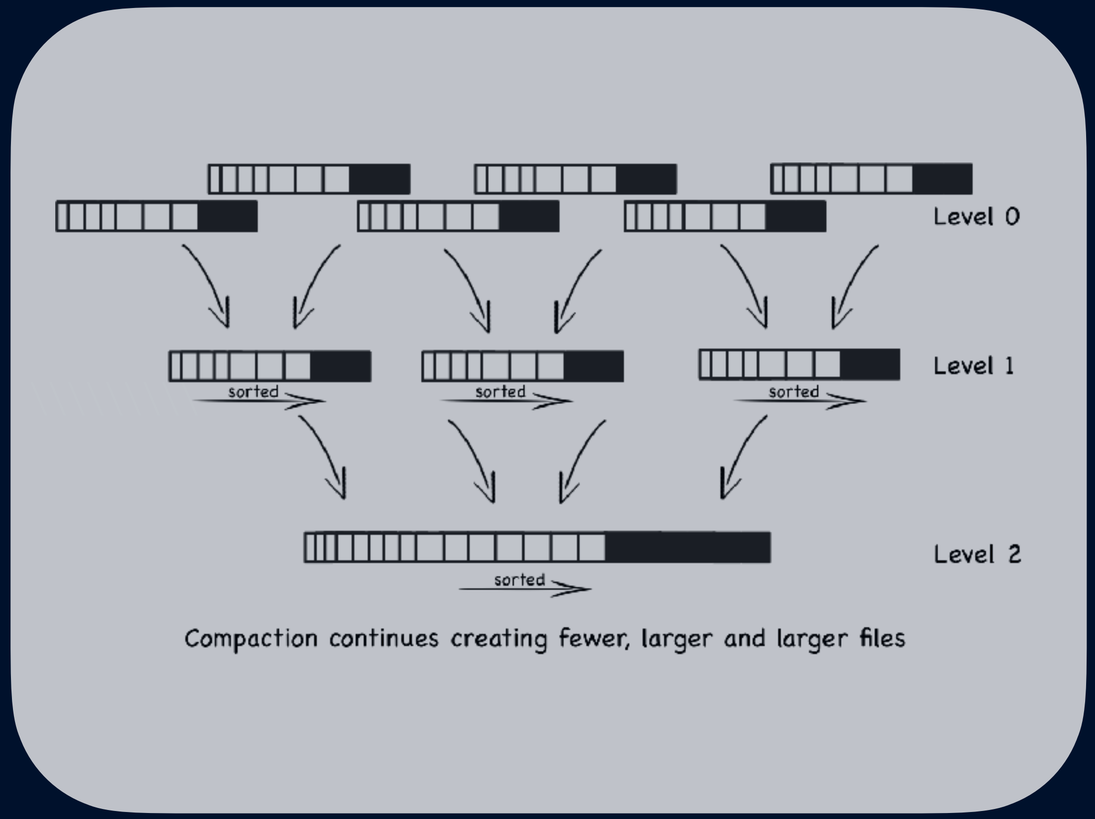


DB Storage Engine Models

Paged (aka Heap)

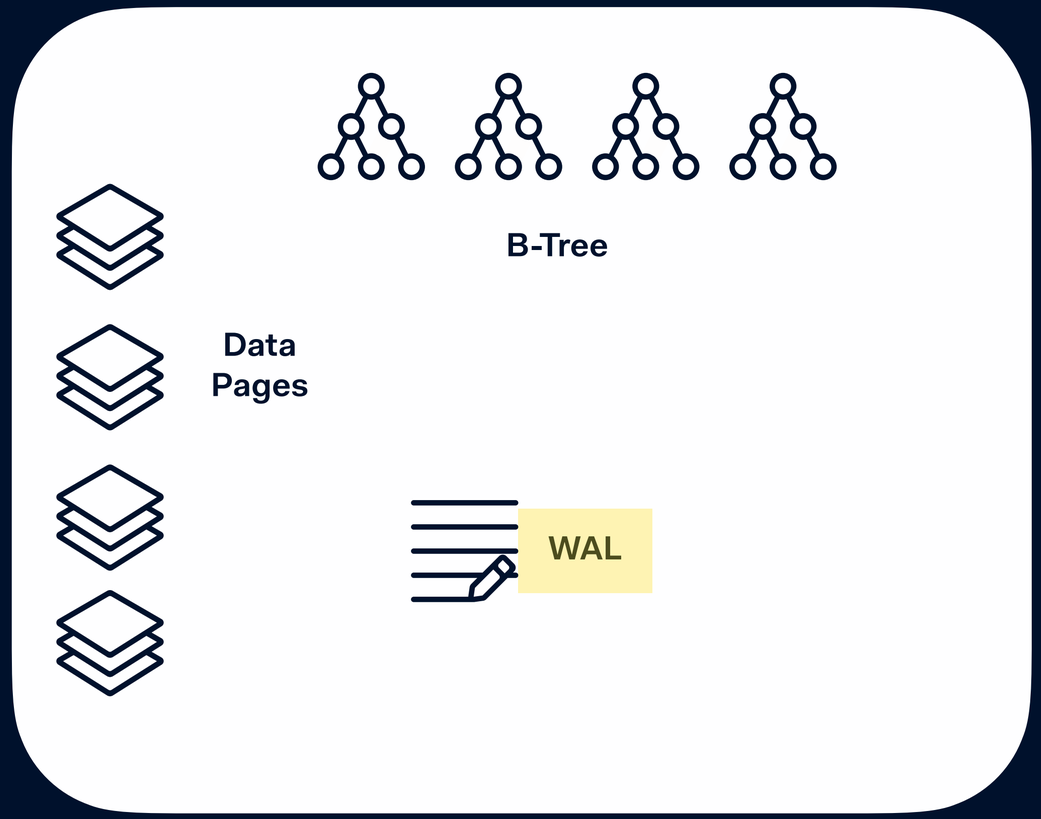


Log Based (key-value)

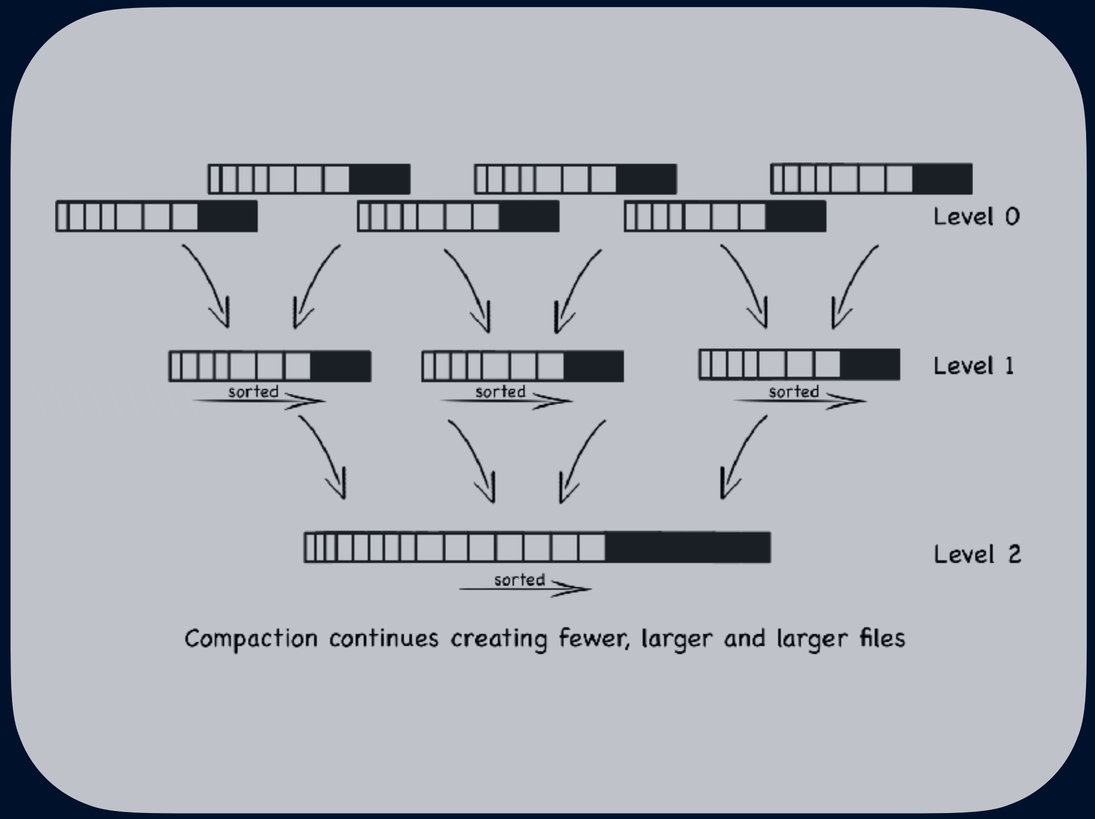


DB Storage Engine Models

Paged (aka Heap)

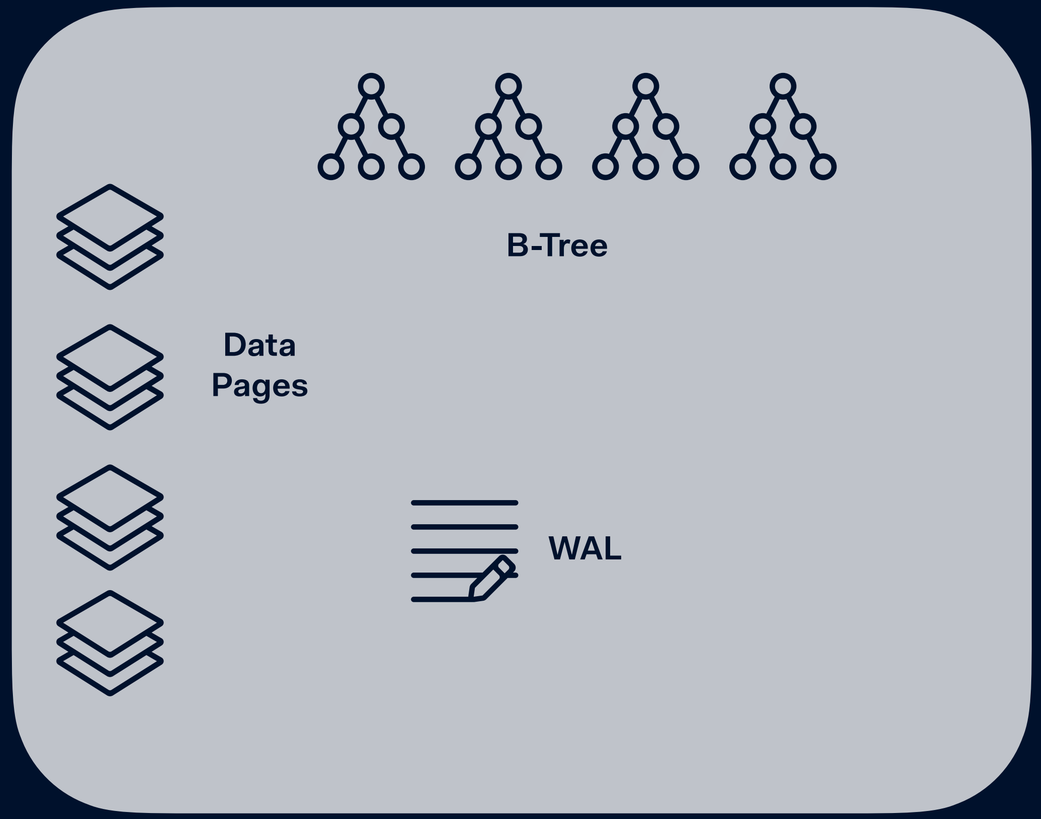


Log Based (key-value)

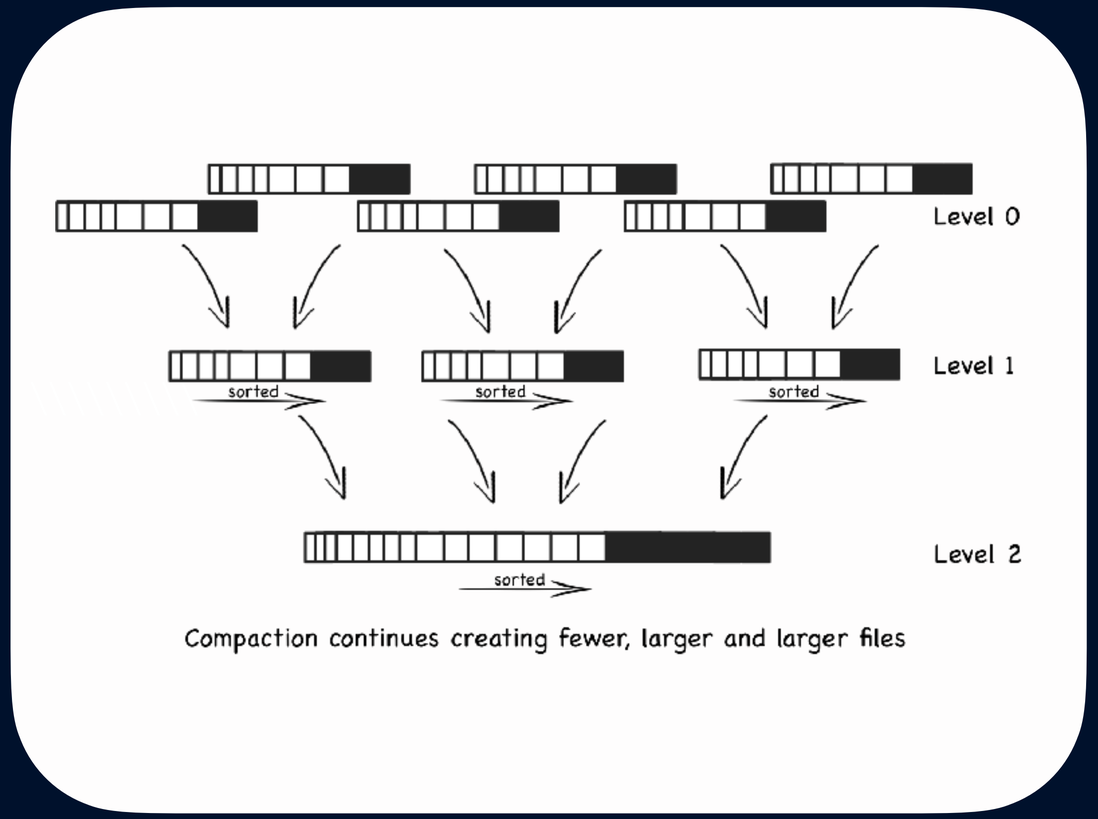


DB Storage Engine Models

Paged (aka Heap)

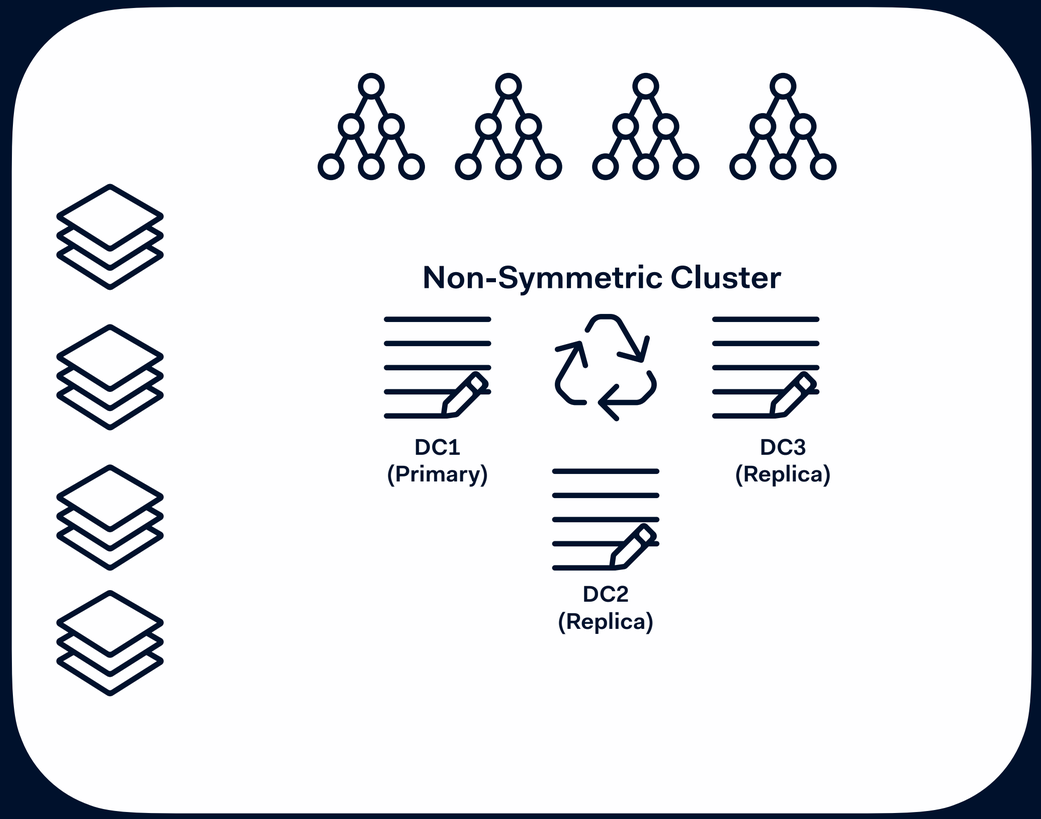


Log Based (key-value)

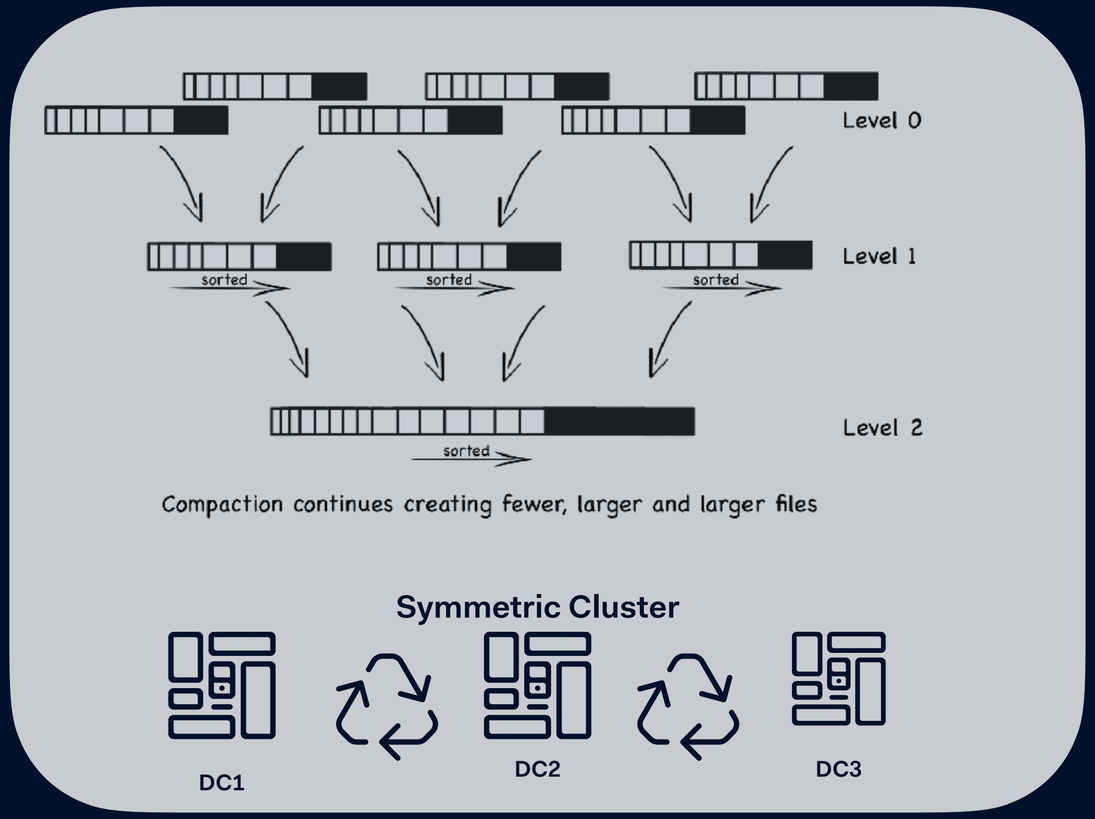


Replication Models

Paged (aka Heap)

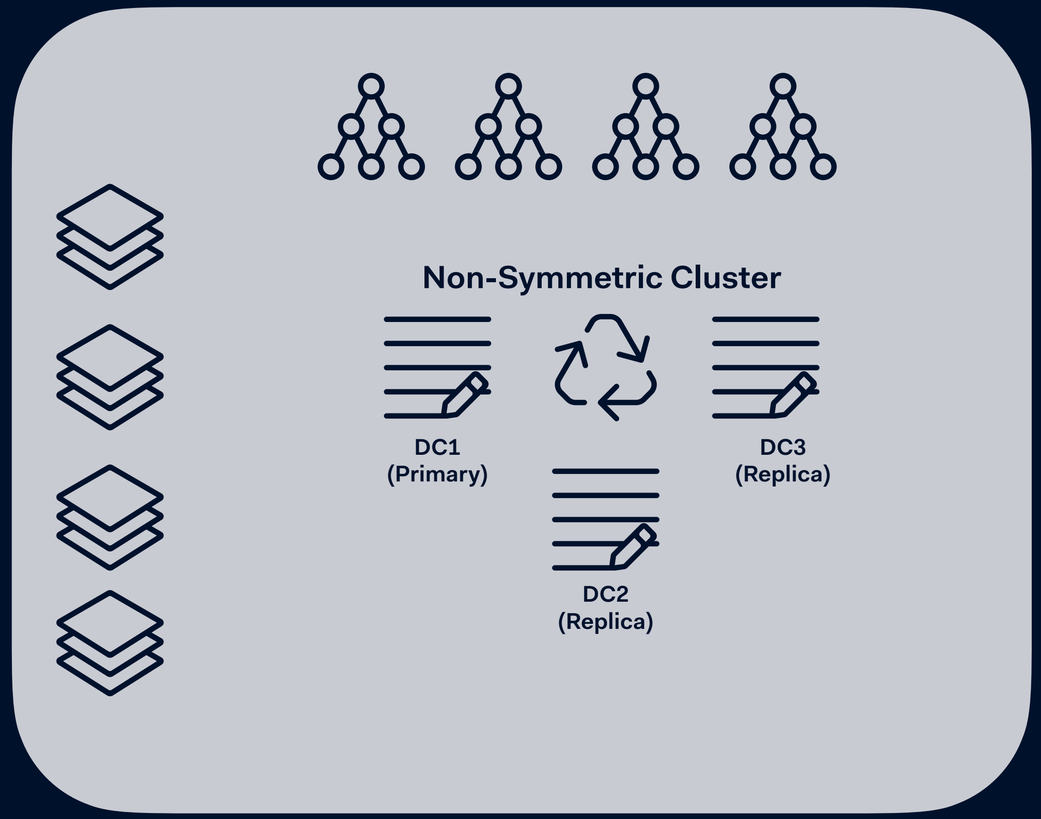


Log Based (Quorum)

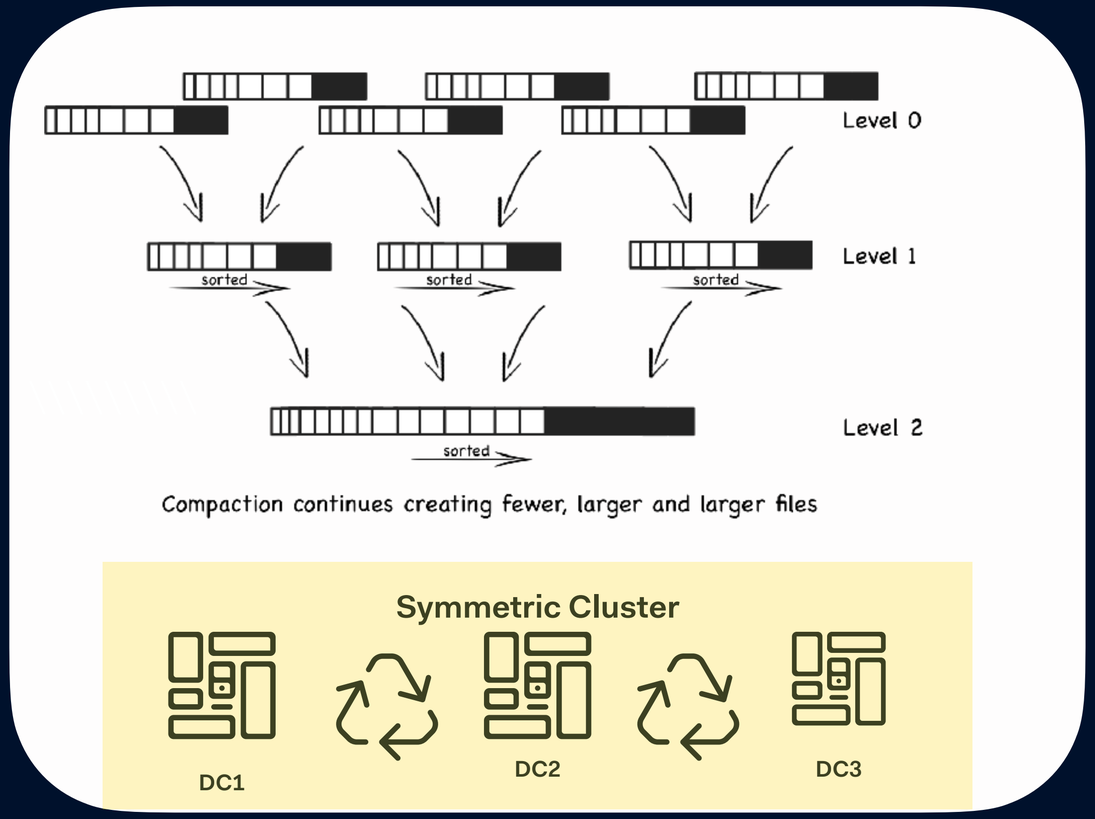


Replication Models

Paged (aka Heap)

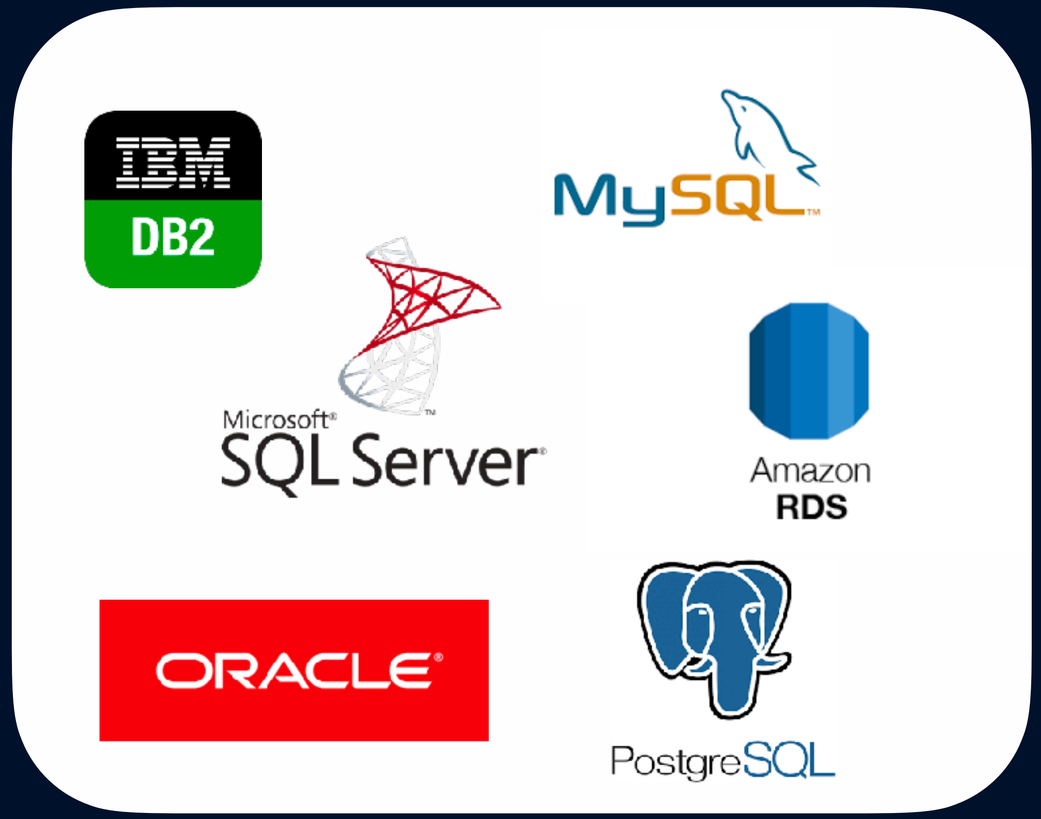


Log Based (Quorum)



Implementations - old & new

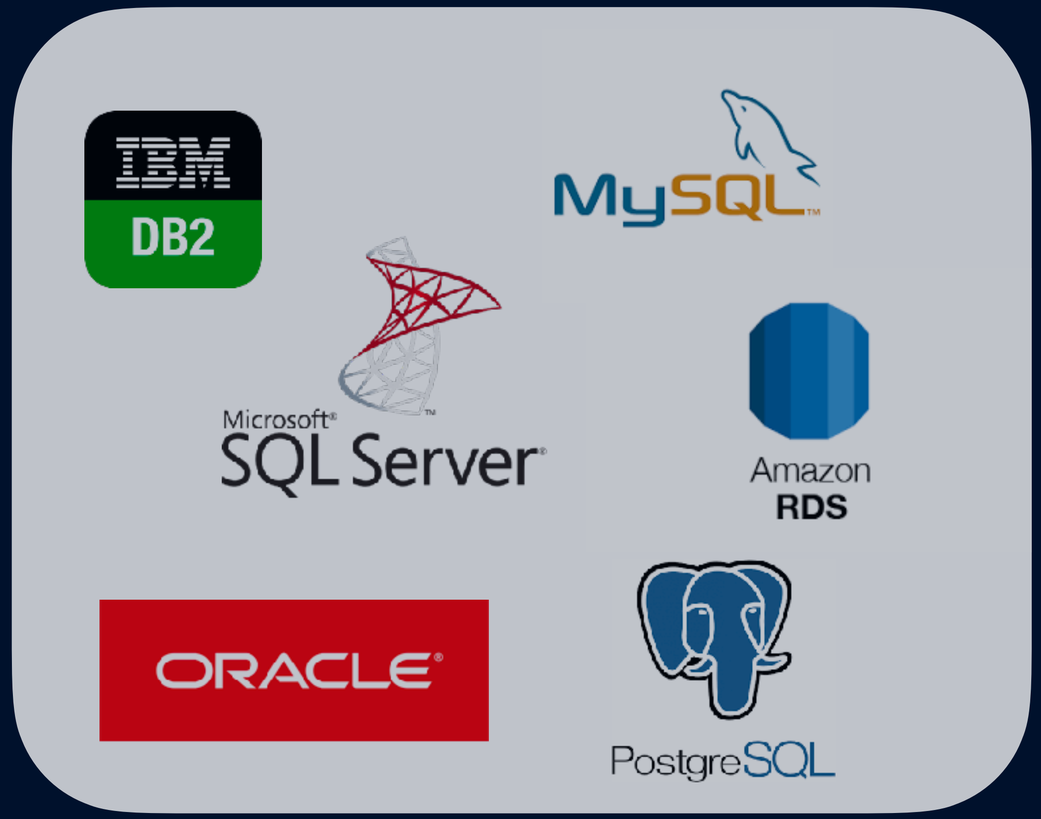
Paged/Heap (older)



Log Structured (newer)

Implementations - old & new

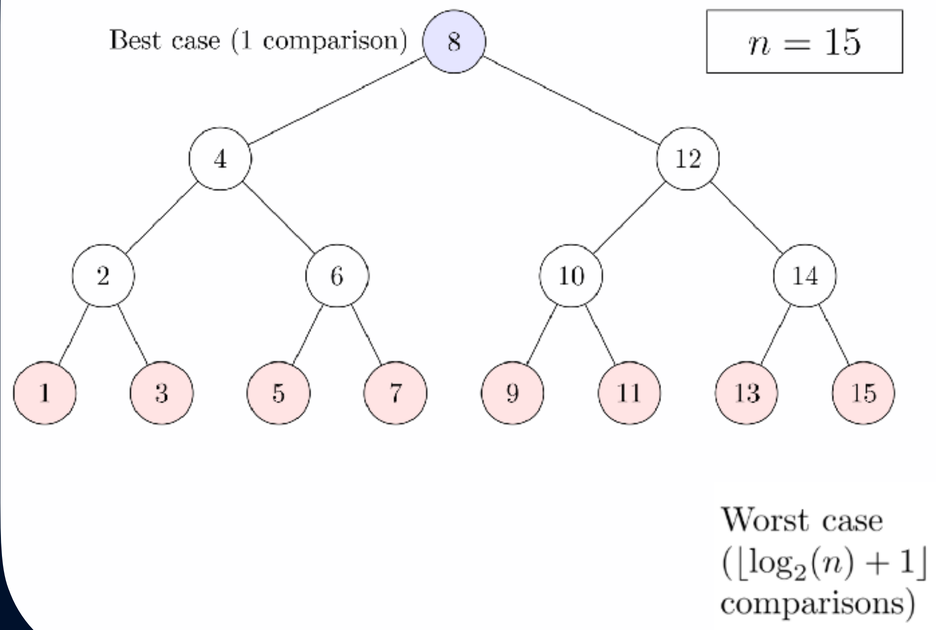
Paged/Heap (older)



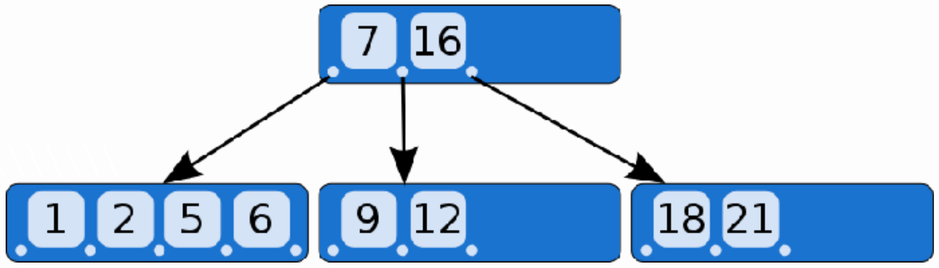
Log Structured (newer)

1970s balance tree fan-out (flat)

Binary Sort



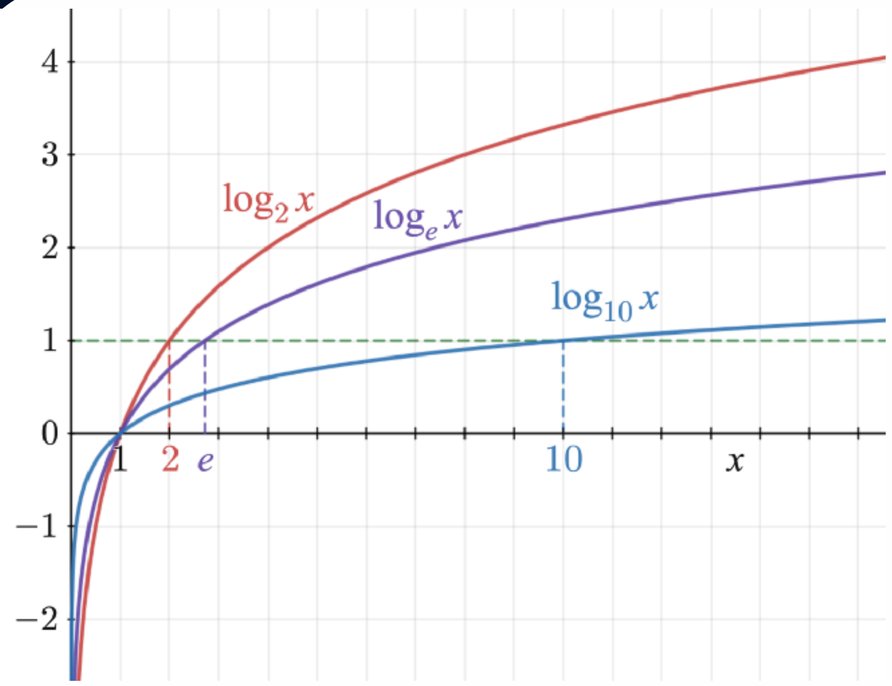
Balanced Tree



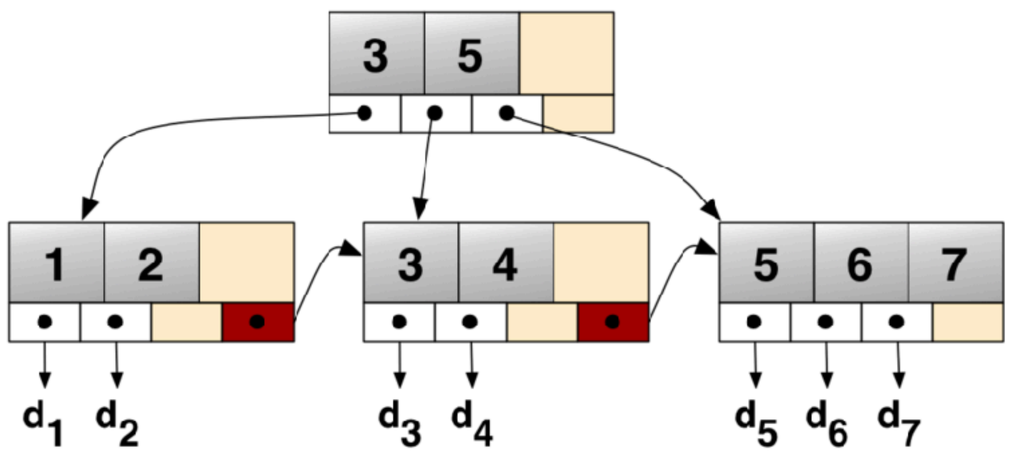
$\log_2(n)$ vs $\log_p(n)$

Further Research 1970s ...

$\log_2(x)$ vs $\log_p(x)$



B+Tree



Btree Reads-per-Select - Flat

Scale Factor 1 (100,000 rows) has 3 Read

```
bench_ff100_sf1=# explain (analyze,buffers) select * from pgbench_accounts where aid = 32928;
                                                QUERY PLAN
-------------------------------------------------------------------------------------------------------------------------------------
 Index Scan using pgbench_accounts_pkey on pgbench_accounts  (cost=0.29..8.31 rows=1 width=97) (actual time=0.724..0.726 rows=1 loops=1)
   Index Cond: (aid = 32928)
   Buffers: shared read=3
 Planning:
   Buffers: shared hit=69 read=1 dirtied=1
 Planning Time: 2.568 ms
 Execution Time: 0.787 ms
(7 rows)
```

Btree Reads-per-Select

Scale Factor 1 (100,000 rows) has 3 Read

```
bench_ff100_sf1=# explain (analyze,buffers) select * from pgbench_accounts where aid = 32928;
                                                QUERY PLAN
-------------------------------------------------------------------------------------------------------------------------------------
 Index Scan using pgbench_accounts_pkey on pgbench_accounts  (cost=0.29..8.31 rows=1 width=97) (actual time=0.724..0.726 rows=1 loops=1)
   Index Cond: (aid = 32928)
   Buffers: shared read=3
 Planning:
   Buffers: shared hit=69 read=1 dirtied=1
 Planning Time: 2.568 ms
 Execution Time: 0.787 ms
(7 rows)
```

Btree Reads-per-Select

Scale Factor 1 (100,000 rows) has 3 Read

```
bench_ff100_sf1=# explain (analyze,buffers) select * from pgbench_accounts where aid = 32928;
                                                QUERY PLAN
-------------------------------------------------------------------------------------------------------------------------------
 Index Scan using pgbench_accounts_pkey on pgbench_accounts  (cost=0.29..8.31 rows=1 width=97) (actual time=0.724..0.726 rows=1 loops=1)
   Index Cond: (aid = 32928)
   Buffers: shared read=3
 Planning:
   Buffers: shared hit=69 read=1 dirtied=1
 Planning Time: 2.568 ms
 Execution Time: 0.787 ms
(7 rows)
```

Btree Reads-per-Select

Scale Factor 2 (200,000 rows) has 4 Read

```
bench_ff100_sf2=# 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=0.023..0.025 rows=1 loops=1)
   Index Cond: (aid = 32928)
   Buffers: shared hit=4
 Planning:
   Buffers: shared hit=70
 Planning Time: 0.791 ms
 Execution Time: 0.078 ms
(7 rows)
```

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)
```

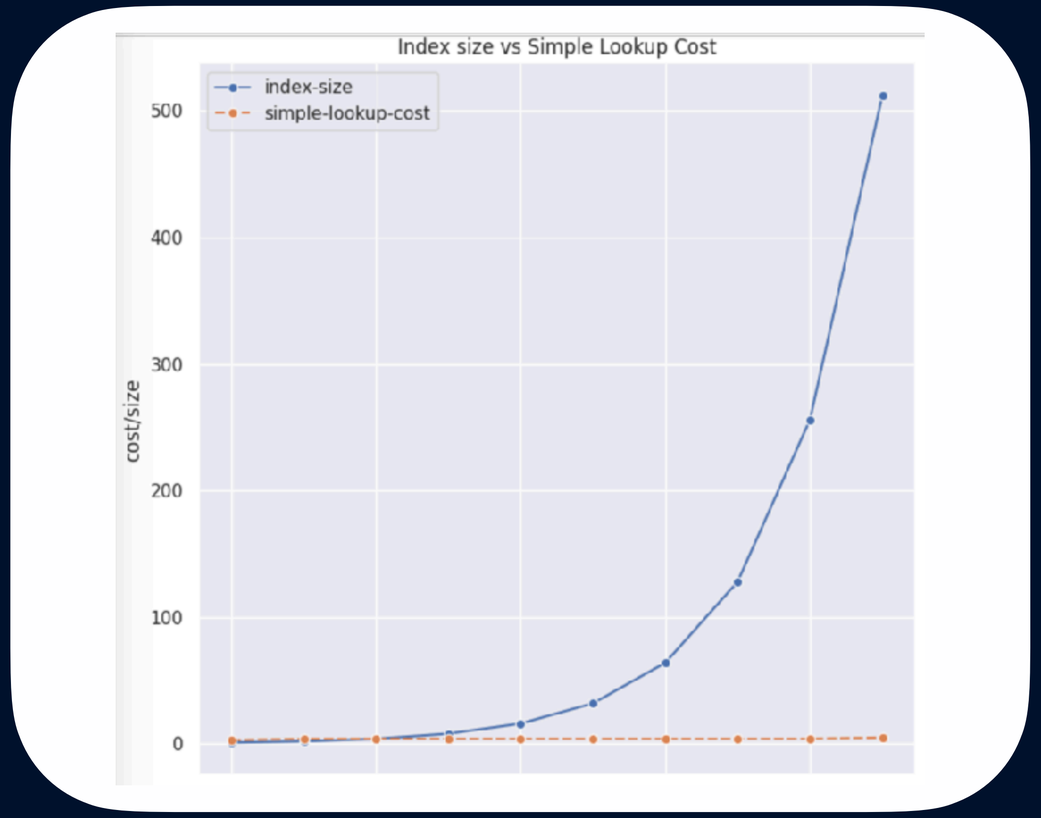
Btree Reads-per-Select

Keep on doubling (still getting 5 reads)

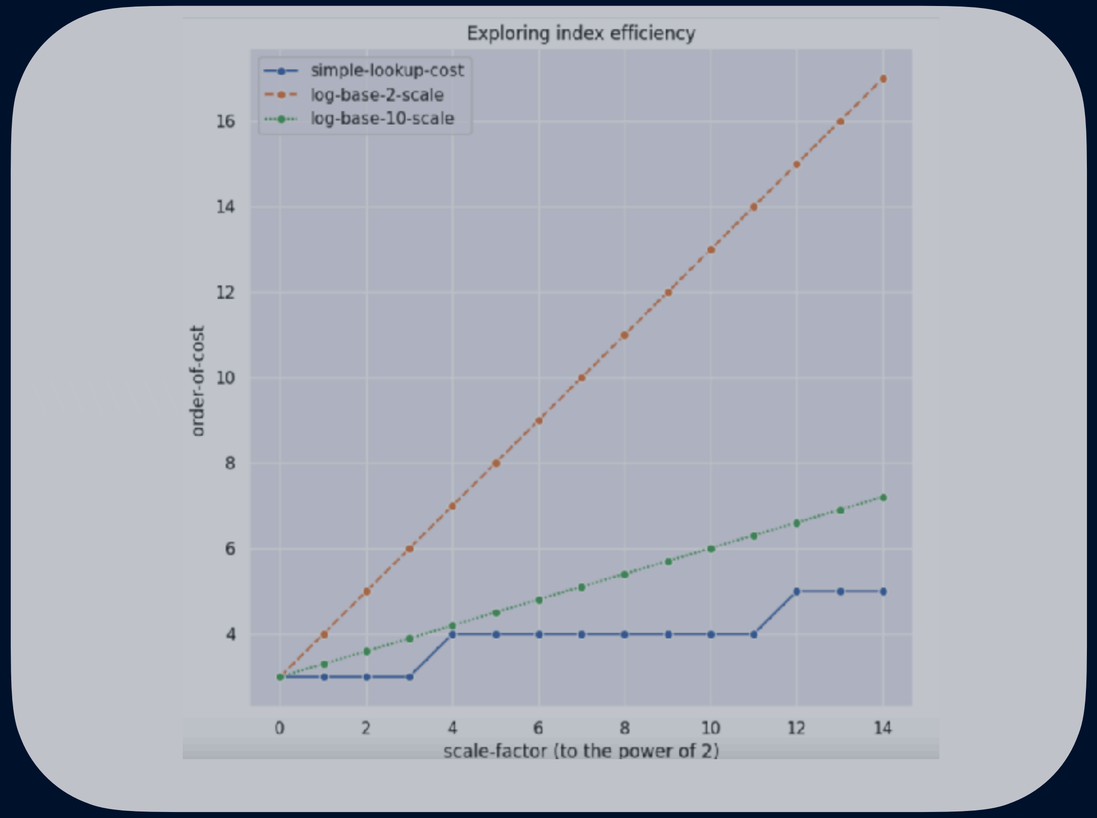
bench_ff100_sf1024, bench_ff100_sf2048, bench_ff100_sf4096 ....

Further Research 1970s ...

Size vs Lookups

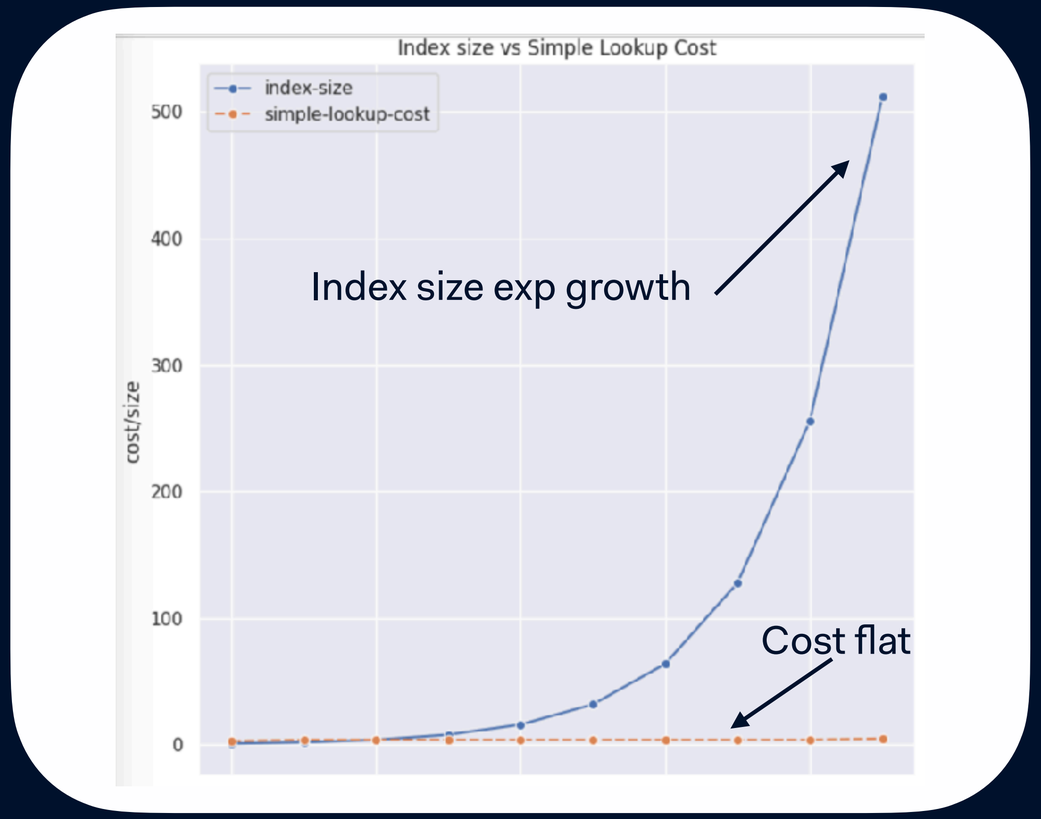


index efficiency (log-scale)

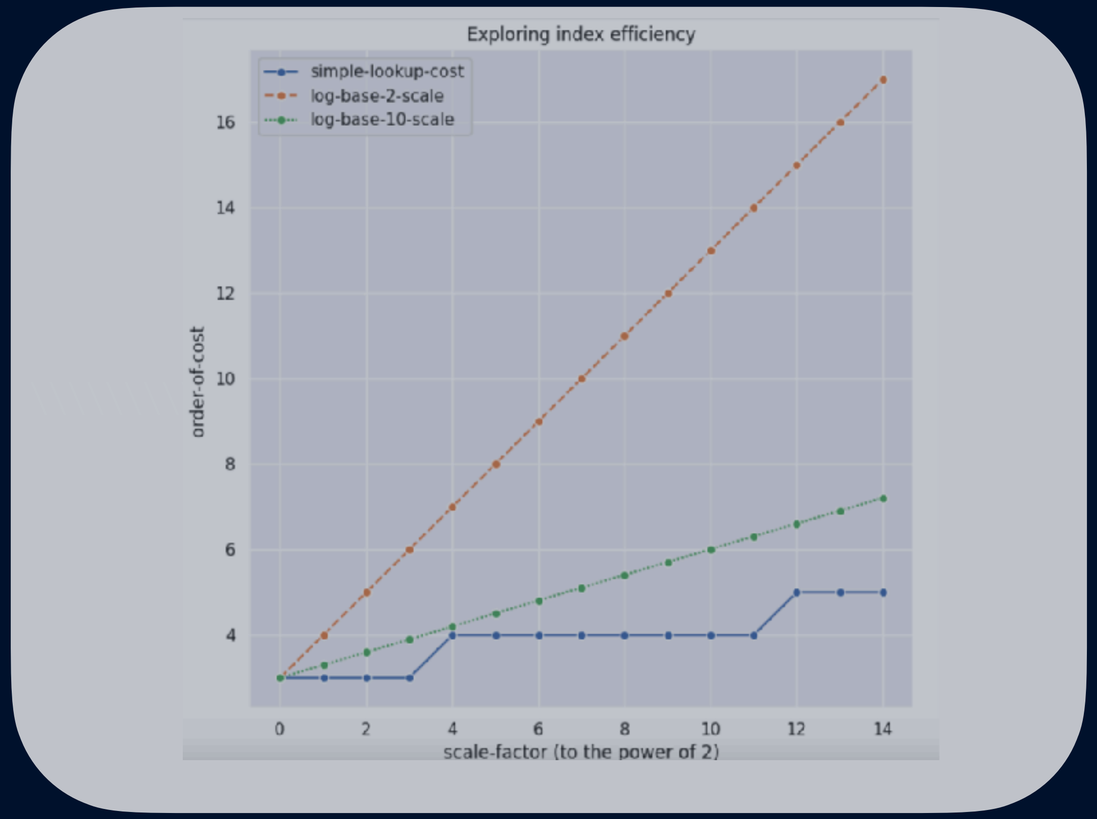


Further Research 1970s ...

Size vs Lookups

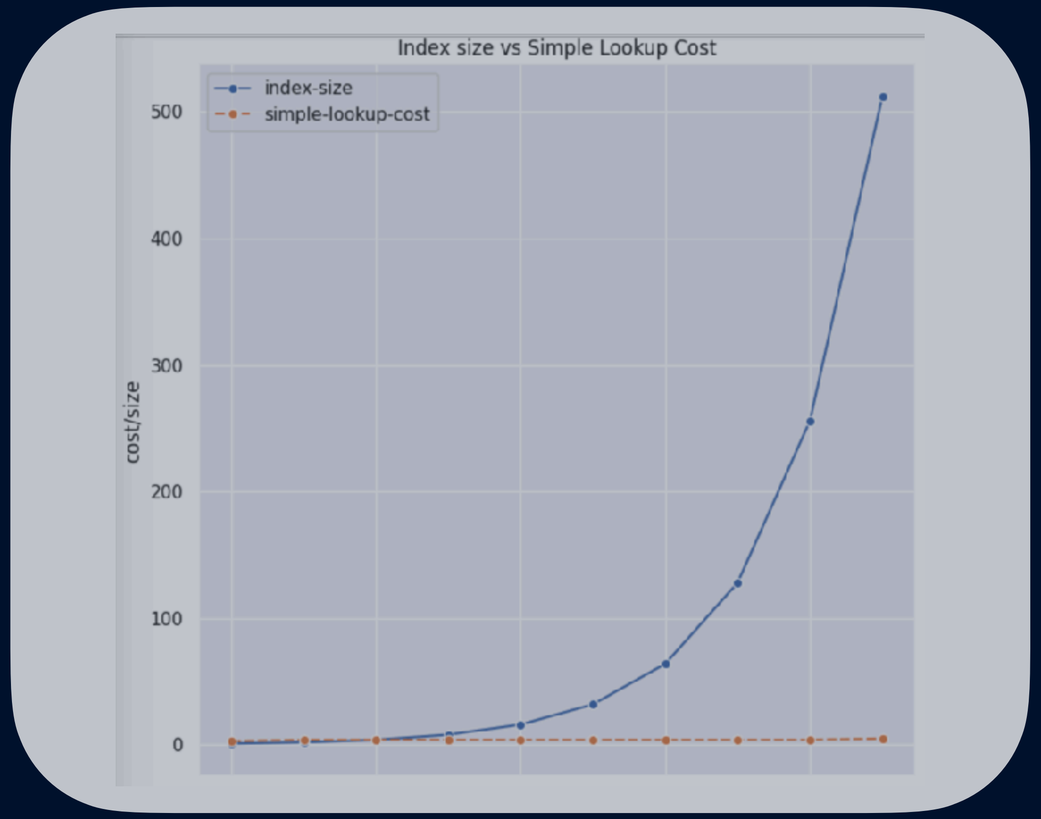


index efficiency (log-scale)

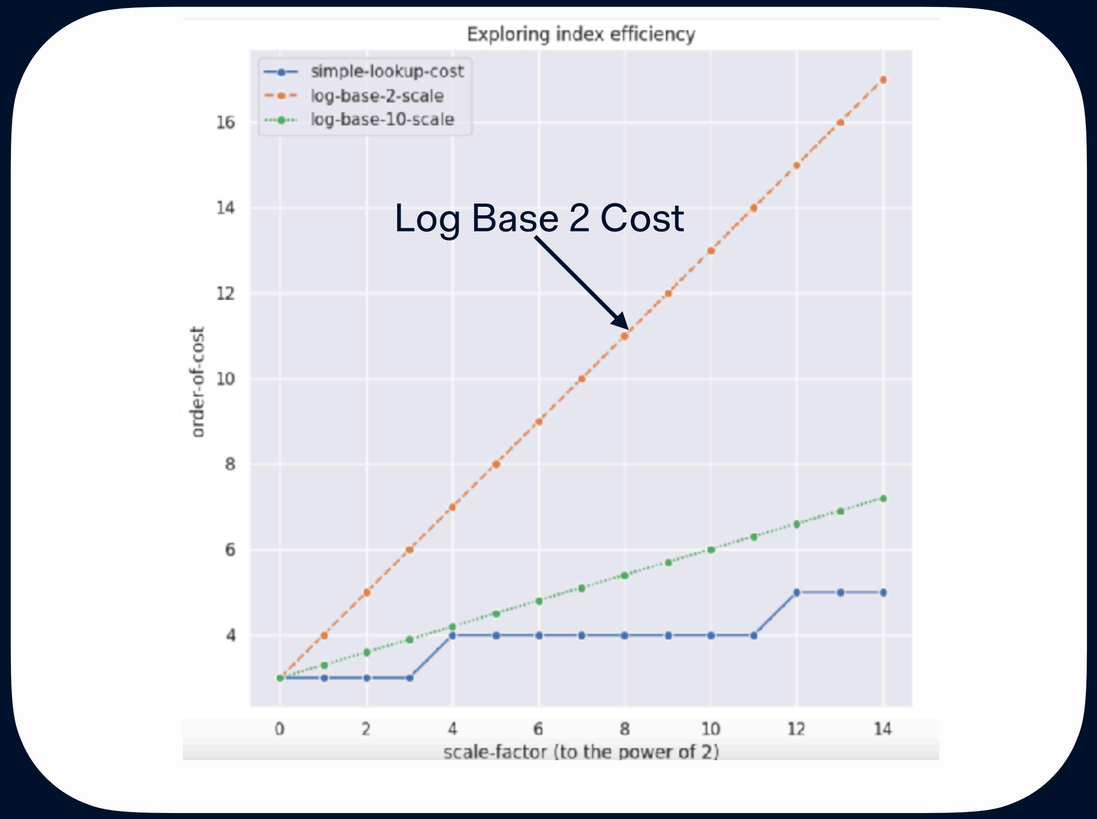


Further Research 1970s ...

Size vs Lookups

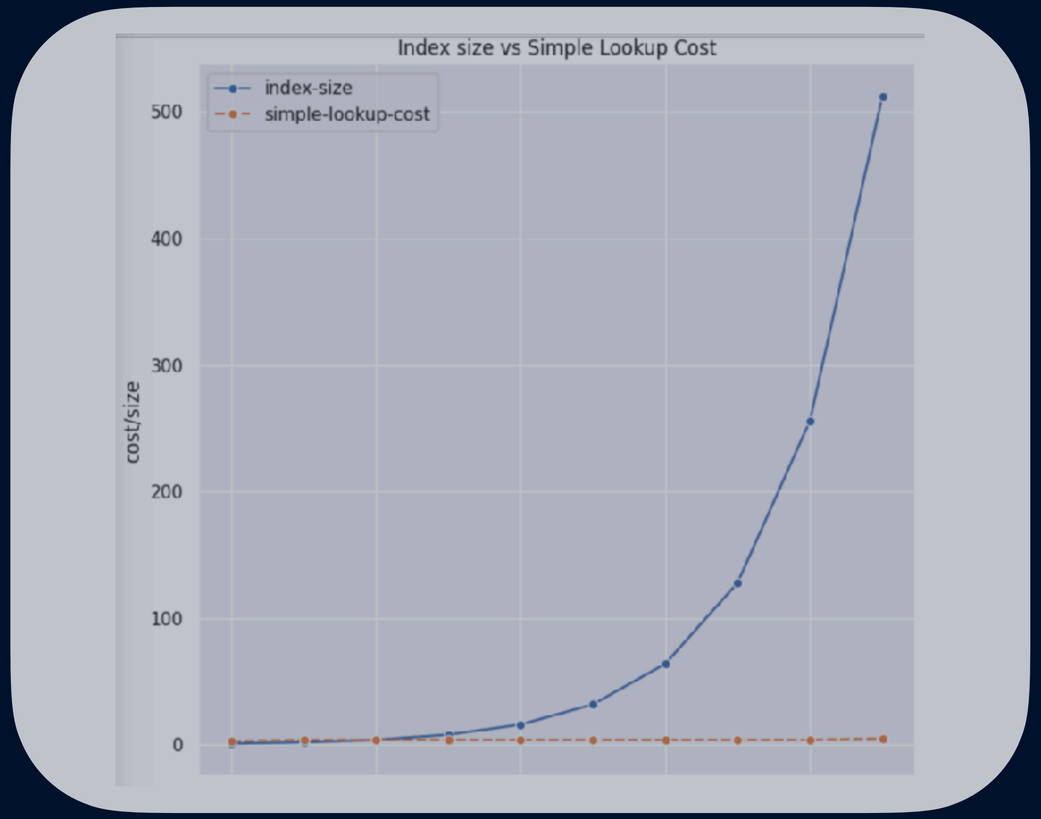


index efficiency (log-scale)

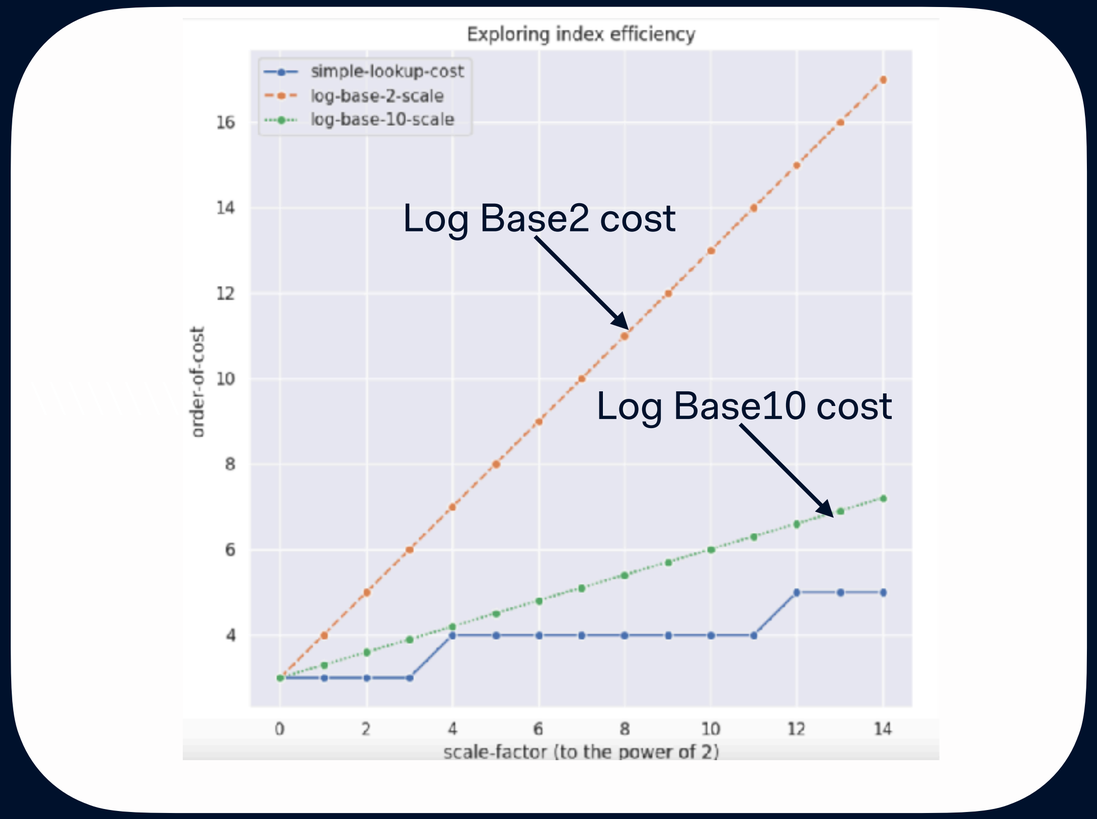


Further Research 1970s ...

Size vs Lookups

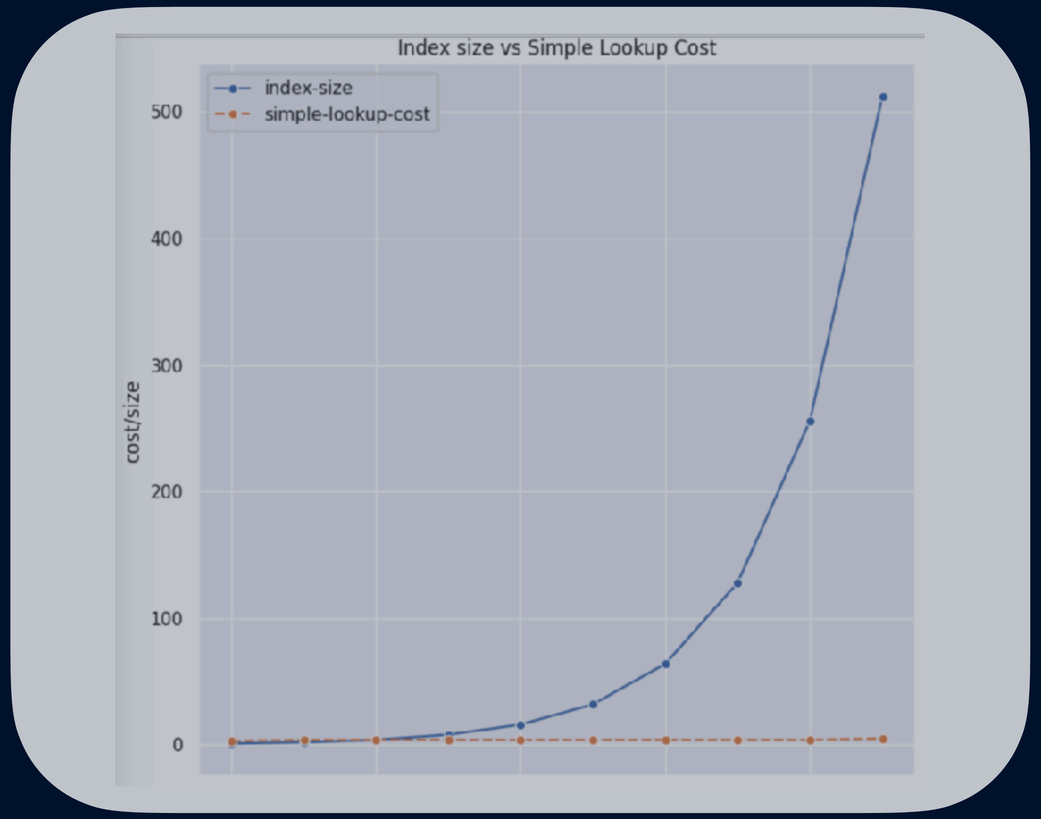


index efficiency (log-scale)

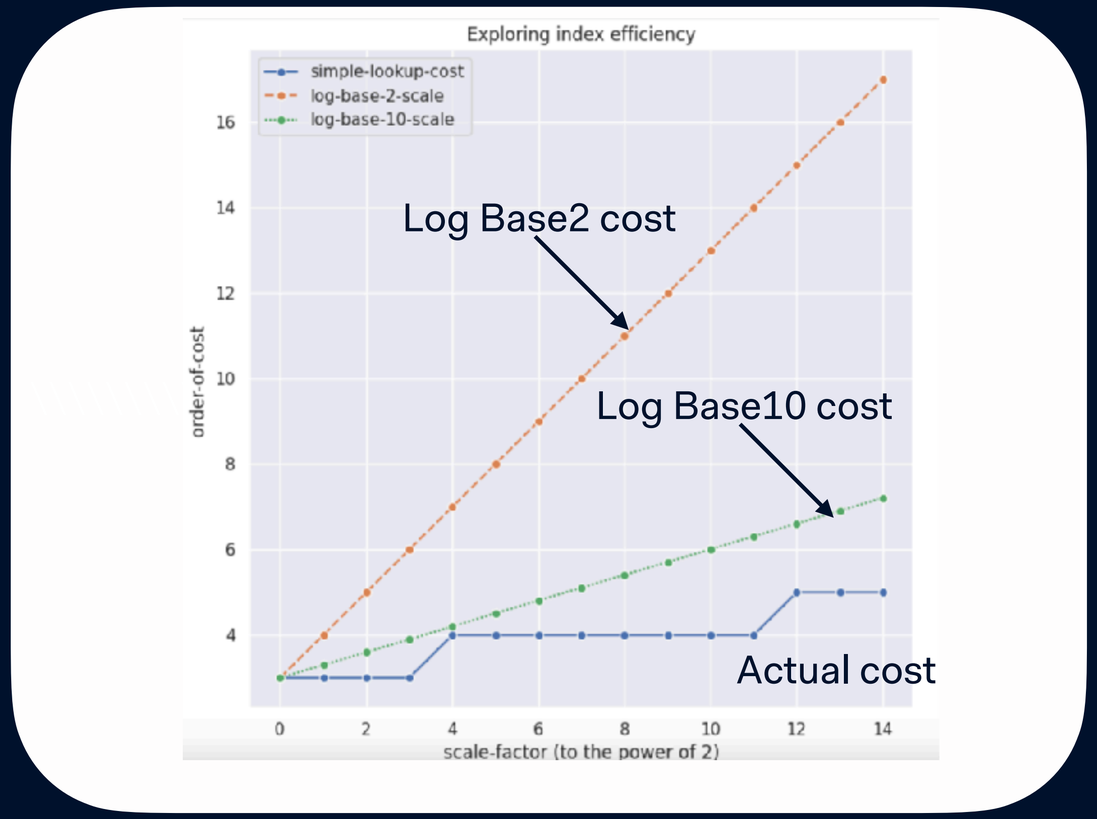


Further Research 1970s ...

Size vs Lookups



index efficiency (log-scale)



pg16 random_normal()

Returns a random value from the normal distribution with the given parameters; mean defaults to 0.0 and stddev defaults to 1.0

https://www.postgresql.org/docs/current/functions-math.html

pg16 random_normal()

Returns a random value from the normal distribution with the given parameters; mean defaults to 0.0 and stddev defaults to 1.0

https://www.postgresql.org/docs/current/functions-math.html

Note: 68% of the data falls within one standard deviation, 95% percent within two standard deviations, and 99.7% within three standard deviations from the mean.

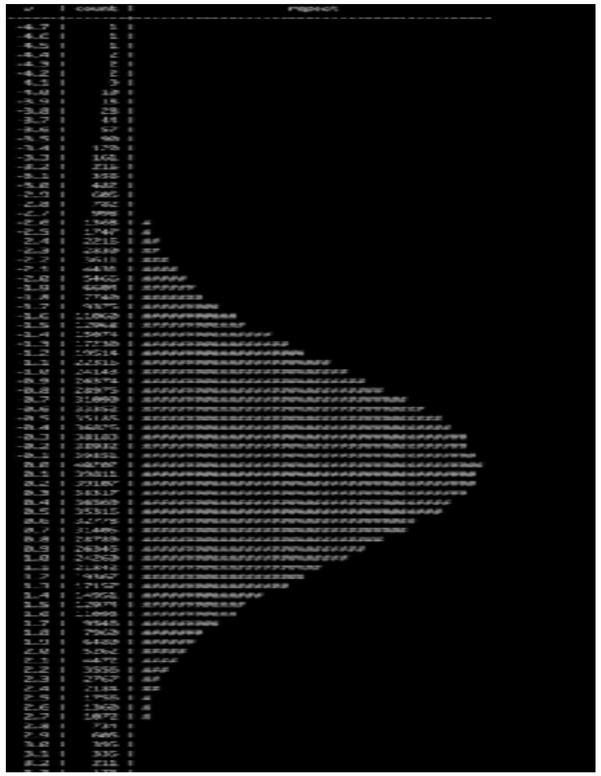
pg16 random_normal()

```
CREATE table normal_dist  as
SELECT round (random_normal()::pg_catalog.numeric,1) as v,
          'Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do … dolore magna aliqua' as filler
from generate_series( 1, 1000000 );
```

pg16 random_normal()

```
CREATE table normal_dist  as
SELECT round (random_normal()::pg_catalog.numeric,1) as v,
        'Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do … dolore magna aliqua' as filler
from generate_series( 1, 1000000 );
```

pg16 random_normal()



ASCII barchart for million row normal_dist table

```
select v, count(*),
        repeat('#', (COUNT(*) / 1000)::int4)
from normal_dist
GROUP BY 1 ORDER BY 1;
```

pg16 random_normal()

v	count	repeat
-4.7	1	
-4.6	1	
-4.5	1	
-4.4	2	
-4.3	2	
-4.2	2	
-4.1	9	
-4.0	10	
-3.9	18	
-3.8	28	
-3.7	44	
-3.6	57	
-3.5	90	
-3.4	120	
-3.3	161	
-3.2	216	
-3.1	356	
-3.0	482	
-2.9	605	
-2.8	782	
-2.7	998	
-2.6	1368	#
-2.5	1747	#
-2.4	2216	##

-1.9	6604	######
-1.8	7740	#######
-1.7	9375	########
-1.6	11060	##########
-1.5	12968	###########
-1.4	15074	##############
-1.3	17230	################
-1.2	19514	##################
-1.1	22316	#####################
-1.0	24143	#######################
-0.9	26374	#########################
-0.8	28975	###########################
-0.7	31090	##############################
-0.6	33352	################################
-0.5	35185	##################################
-0.4	36826	###################################
-0.3	38183	#####################################
-0.2	38932	######################################
-0.1	39851	#######################################
0.0	40202	##
0.1	39811	#######################################

Columnar storage

```
CREATE TABLE normal_dist_columnar(LIKE normal_dist3) USING COLUMNAR;

INSERT into normal_dist_columnar (select * from normal_dist);
```

Columnar storage

```
CREATE TABLE normal_dist_columnar(LIKE normal_dist3) USING COLUMNAR;

INSERT into normal_dist_columnar (select * from normal_dist);
```

```
SELECT pg_total_relation_size('normal_dist')::numeric/
pg_total_relation_size('normal_dist_columnar') AS compression_ratio;

  compression_ratio
----------------------
 70.5898437500000000
```

Columnar comparison

Default Table Scan

```
postgres=# explain (analyze,buffers) select count(*) from normal_dist3 where v = '2.2';
                                        QUERY PLAN
-----------------------------------------------------------------------------------------------------------------
 Aggregate  (cost=79197.16..79197.17 rows=1 width=8) (actual time=999.460..999.464 rows=1 loops=1)
   Buffers: shared hit=2112 read=64576
   ->  Seq Scan on normal_dist3  (cost=0.00..79187.41 rows=3900 width=0) (actual time=0.917..997.464 rows=3563 loops=1)
         Filter: (v = 2.2)
         Rows Removed by Filter: 996437
         Buffers: shared hit=2112 read=64576
 Planning:
   Buffers: shared hit=7
 Planning Time: 3.519 ms
 Execution Time: 999.907 ms
(10 rows)
```

Columnar comparison

Columnar based query

```
postgres=# explain (analyze,buffers) select count(*) from normal_dist3_columnar  where v = '2.2';
                                                    QUERY PLAN
-----------------------------------------------------------------------------------------------------------------------------------
 Aggregate  (cost=30.52..30.53 rows=1 width=8) (actual time=384.268..384.273 rows=1 loops=1)
   Buffers: shared hit=836
   ->  Custom Scan (ColumnarScan) on normal_dist3_columnar  (cost=0.00..18.02 rows=5000 width=0) (actual time=8.996..383.011 rows=3563
loops=1)
         Filter: (v = 2.2)
         Rows Removed by Filter: 996437
         Columnar Projected Columns: v
         Columnar Chunk Group Filters: (v = 2.2)
         Columnar Chunk Groups Removed by Filter: 0
         Buffers: shared hit=836
 Planning:
   Buffers: shared hit=36
 Planning Time: 6.261 ms
 Execution Time: 384.793 ms
```

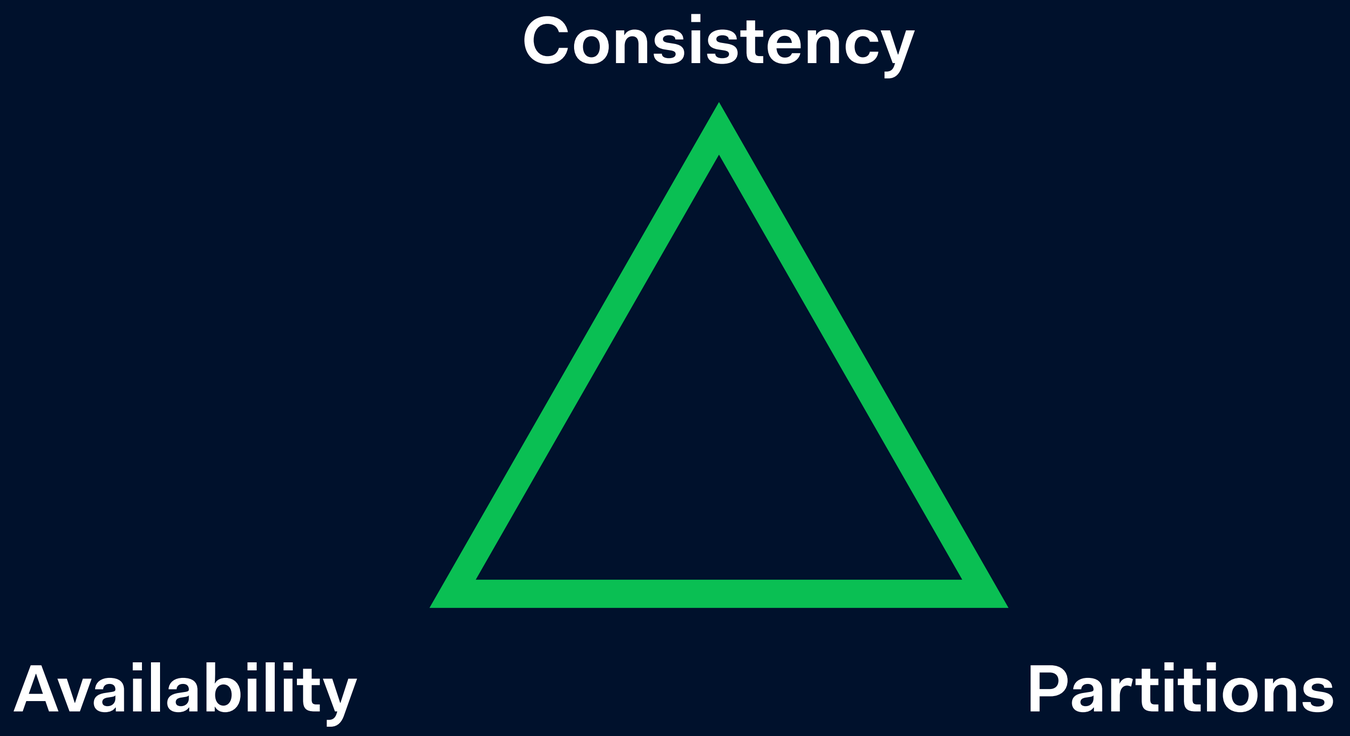
How radical is Columnar?

Columnar data stores can co-existing with traditional tables.

To me Columnar is orthogonal, storing by columns instead of rows, so different but not conceptually difficult.

and pause ...

CAP Theorem



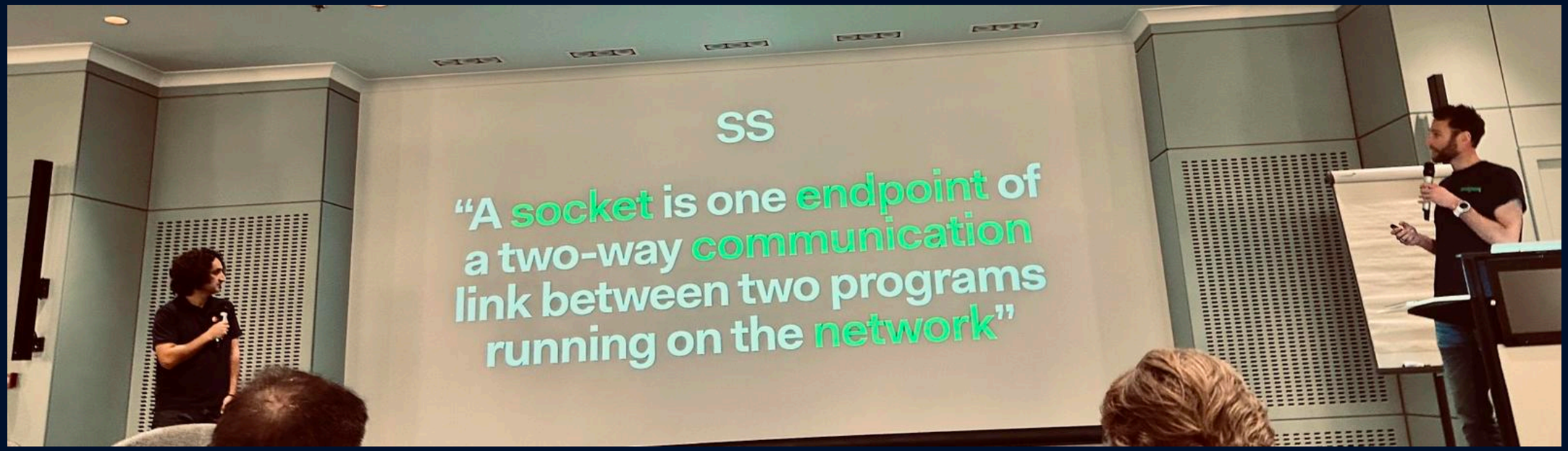
CAP Theorem Simplied

* In the presence of the network Partition, a distributed system is either Available or Consistent

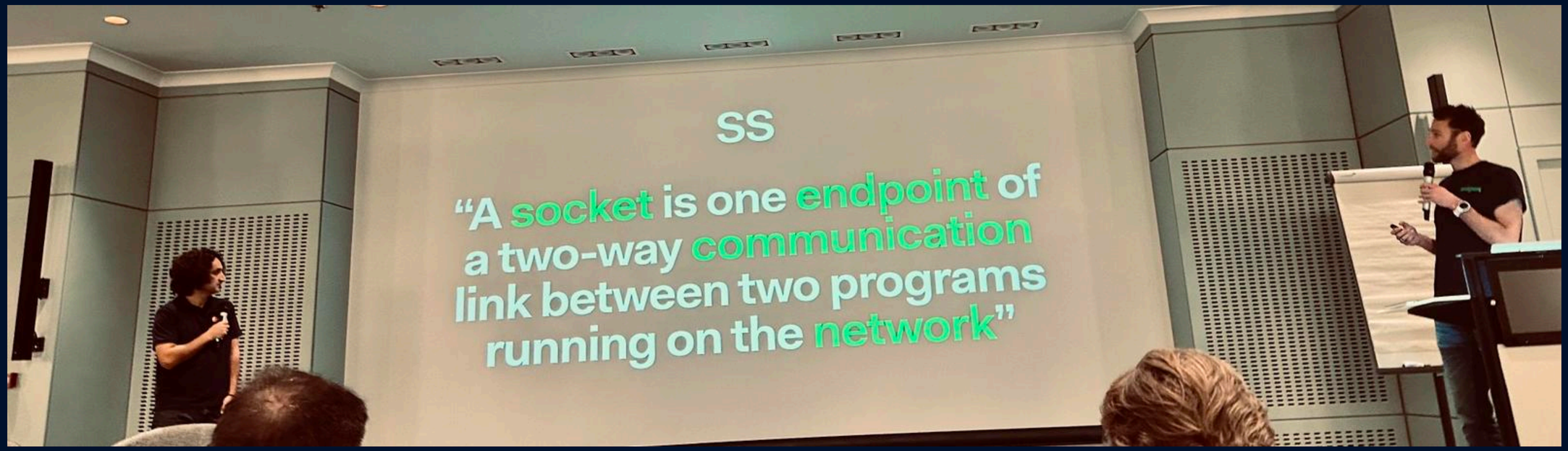
* Strong parallels with problems in network fundamentals (split-brain)

- SE Radio 227: Eric Brewer: The CAP Theorem, Then and Now

Postgres SyncRep Gotchas

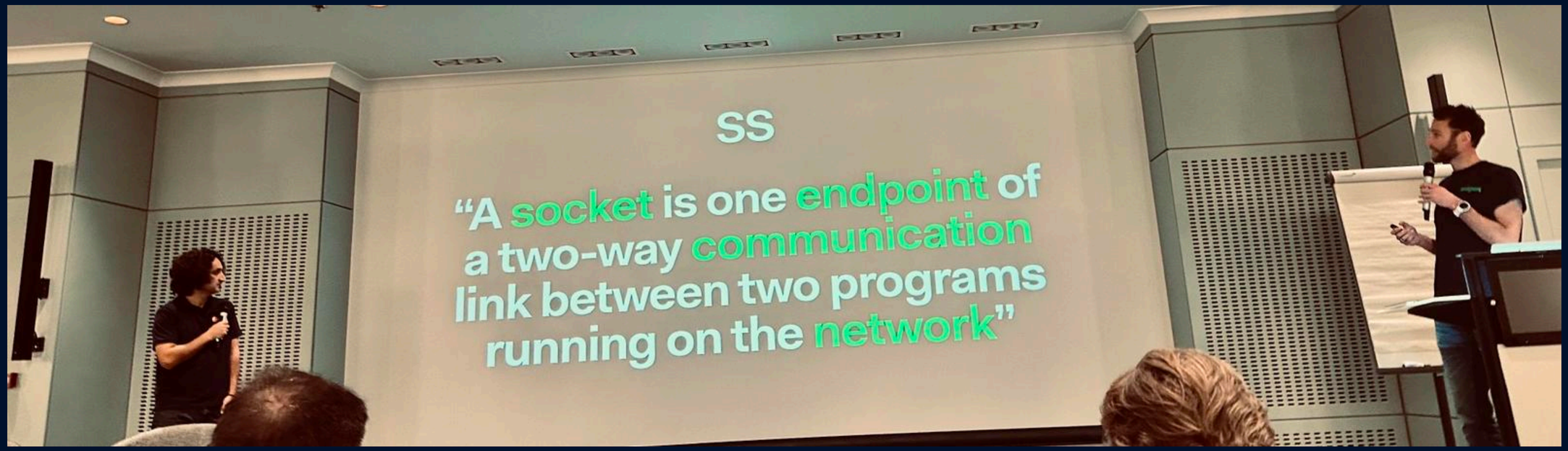


Postgres SyncRep Gotchas



Performance and debugging

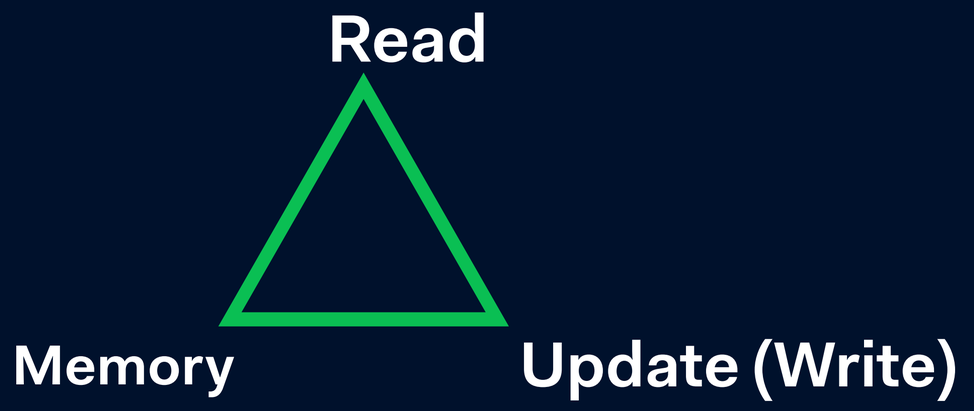
Postgres SyncRep Gotchas



Eventual Consistency & In Doubt Transactions?

RUM Conjecture

RUM Conjecture



* Traditional databases are read optimized (via btree structures)

* LSM Trees are Write Optimized and read can be expensive (bloom filters)

RocksDB

A little history ...

* Facebook started 2012 (LevelDB fork)
* By 2017 big success - Mark Callaghan's FOSDEM presentation
* Multiple subsequent RocksDB projects (several are "postgresql wire compatible")

RocksDB

A little history ...

* Facebook started 2012 (LevelDB fork)
* By 2017 big success - Mark Callaghan's FOSDEM presentation
* Multiple subsequent RocksDB projects (several are "postgresql wire compatible")

FB (2017) caveats

* Limiting max rows/memory for large transactions
* Performance tuning is hard

Recap - Why LSM Trees?

Pros

* Optimized for Sequential Writes

* Quorum & Symmetric Distributed Databases

Cons

* Read Amplification
* Distributed Read
* Very Long Transaction
* Very Large Inserts & Updates

Recap - Why LSM Trees?

Pros

* Optimized for Sequential Writes
* Quorum & Symmetric Distributed Databases

Cons

* Read Amplification
* Distributed Read
* Very Long Transaction
* Very Large Inserts & Updates

CAP Theorem and LSM Trees

* **Cassandra** prioritize **Availability** and eventual consistency

* **CockroachDB** (Serializable) and **Yugabyte** (Read Committed) prioritize **Consistency**

MVCC and Eventual Consistency

Nice 👍

Mum has posted a new cat..

MVCC and Eventual Consistency

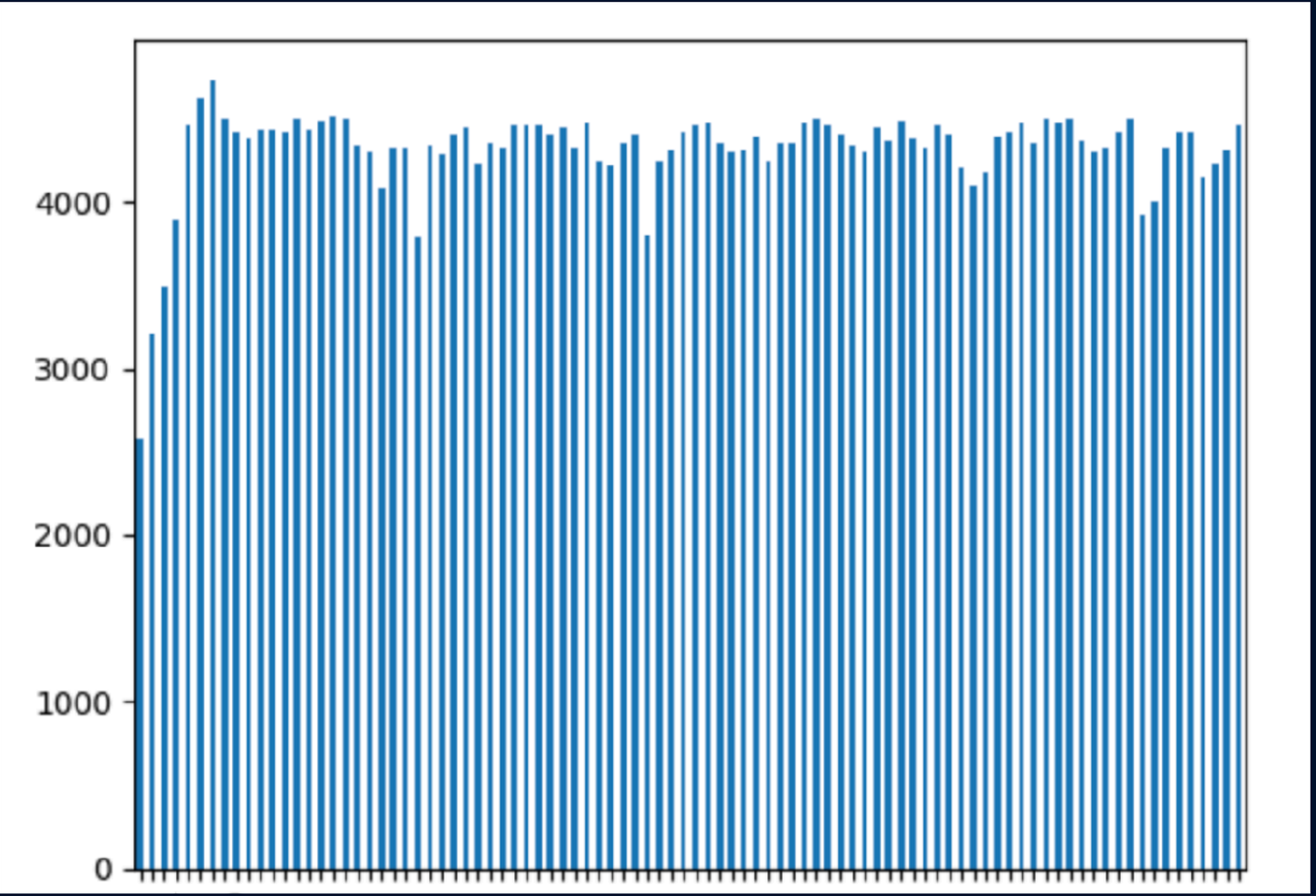
Hey … Now it gone …

MVCC and Eventual Consistency

Nice 👍

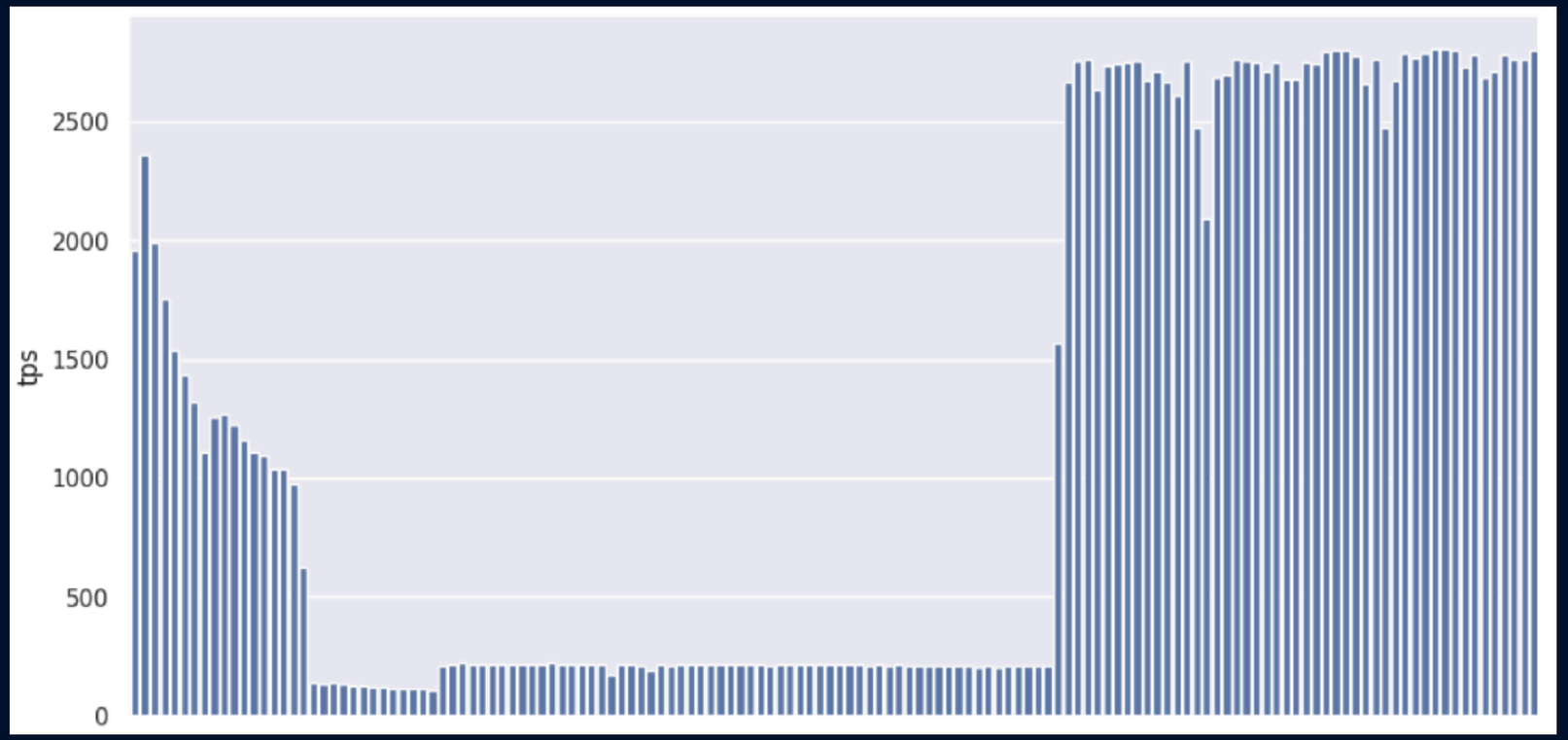
The cat photo is back ❤️

Pgbench - pgrocks (fdw)



Postgres Long TXNs - Berlin

Long TXNs - pgrocks (fdw)



Long TXNs - pgrocks (fdw)

Session One - Hot Block

```
pgbench -P 1 -c 10  -t 100000
-f update-abalance-for-aid-1.sql
```

Session Two - Long TXN

```
begin;
select txid_current();
select pg_sleep(${sleepSeconds});
end
```

Long TXNs - pgrocks (fdw)

Session One - Hot Block

```
pgbench -P 1 -c 10  -t 100000
-f update-abalance-for-aid-1.sql
```

Session Two - Long TXN

```
begin;
select txid_current();
select pg_sleep(${sleepSeconds});
end
```

Big Table Tests

Setup 10 million row table with id & filler

```
create table big_table (id int, fil1, text default md5(random()::text));
insert into big_table(id) SELECT generate_series(1,10000000);
analyze big_table;
```

Analyze YugabyteDB

```
bench1=# analyze big_table;
analyze big_table;
WARNING:  'analyze' is a beta feature!
LINE 1: analyze big_table;
        ^
HINT:  Set 'ysql_beta_features' yb-tserver gflag to true to suppress the warning for all beta features.
ANALYZE
Time: 5743.785 ms (00:05.744)
```

Actual 4 vs Expected 17,103!

```
bench1=# analyze big_table;
analyze big_table;
WARNING:  'analyze' is a beta feature!
LINE 1: analyze big_table;
        ^
HINT:  Set 'ysql_beta_features' yb-tserver gflag to true to suppress the warning for all beta features.
ANALYZE
Time: 5743.785 ms (00:05.744)
```

```
bench1=# explain (analyze,buffers) select filler from big_table where id = 80000;
explain (analyze,buffers) select filler from big_table where id = 80000;
                                                        QUERY PLAN
____________________________________________________________________

 Index Scan using big_table_id on big_table  (cost=0.00..2099.12 rows=17103 width=218)
                                                                  (actual time=8.996..9.050 rows=4 loops=1)
   Index Cond: (id = 80000)
 Planning Time: 7.983 ms
 Execution Time: 10.065 ms
 Peak Memory Usage: 8 kB
(5 rows)
```

adyen

engineered
for ambition

YugabyteDB - RBO vs CBO

In OLTP database applications, a rule-based optimizer can suffice for many queries, provided that the data model includes appropriate indexes, and the queries remain straightforward.

Franck Pachot for YugabyteDB

https://dev.to/yugabyte/yugabytedb-cost-based-optimizer-and-cost-model-for-distributed-lsm-tree-1hb4

Drop Column - pgrocks (fdw)

Session One - pgbench

```
pgbench -P 1  -T 300
```

Session Two - Long TXN

```
# alter table big_table drop
column fil1;

ALTER TABLE

Time: 15.719 ms
```

Drop Column - pgrocks (fdw)

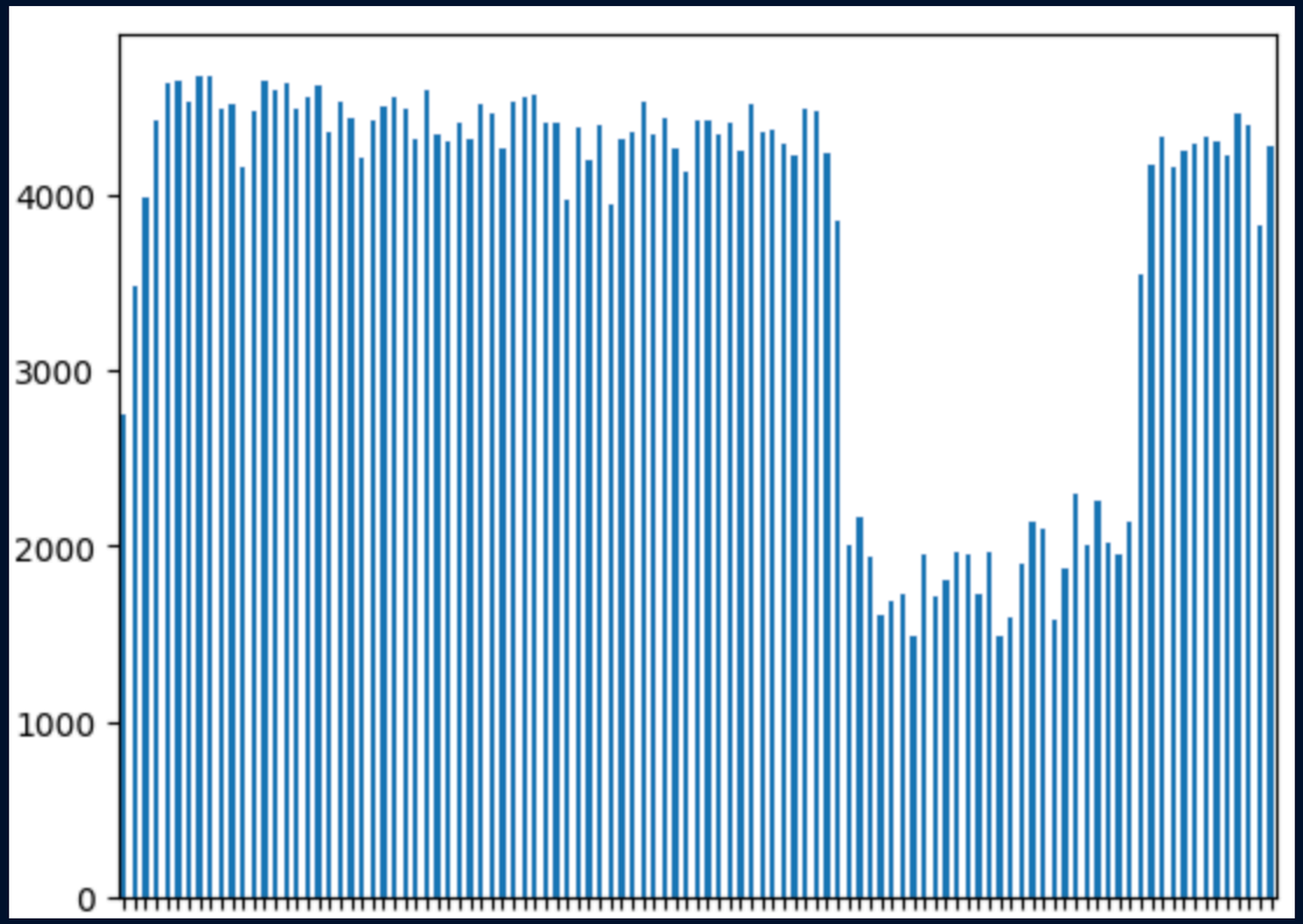
Session One - pgbench

```
pgbench -P 1  -T 300
```

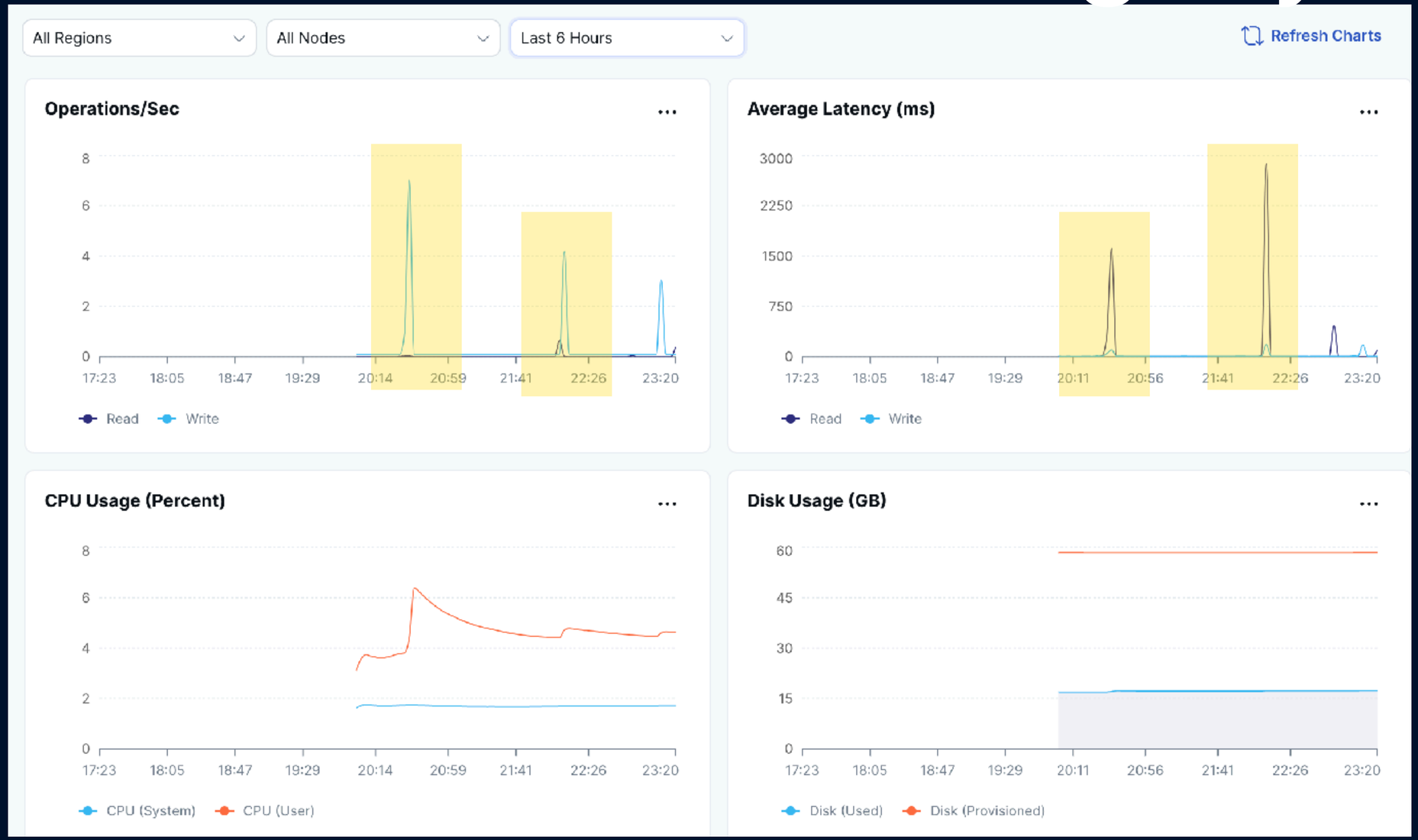
Session Two - Long TXN

```
# alter table big_table drop
column fil1;

ALTER TABLE

Time: 15.719 ms
```

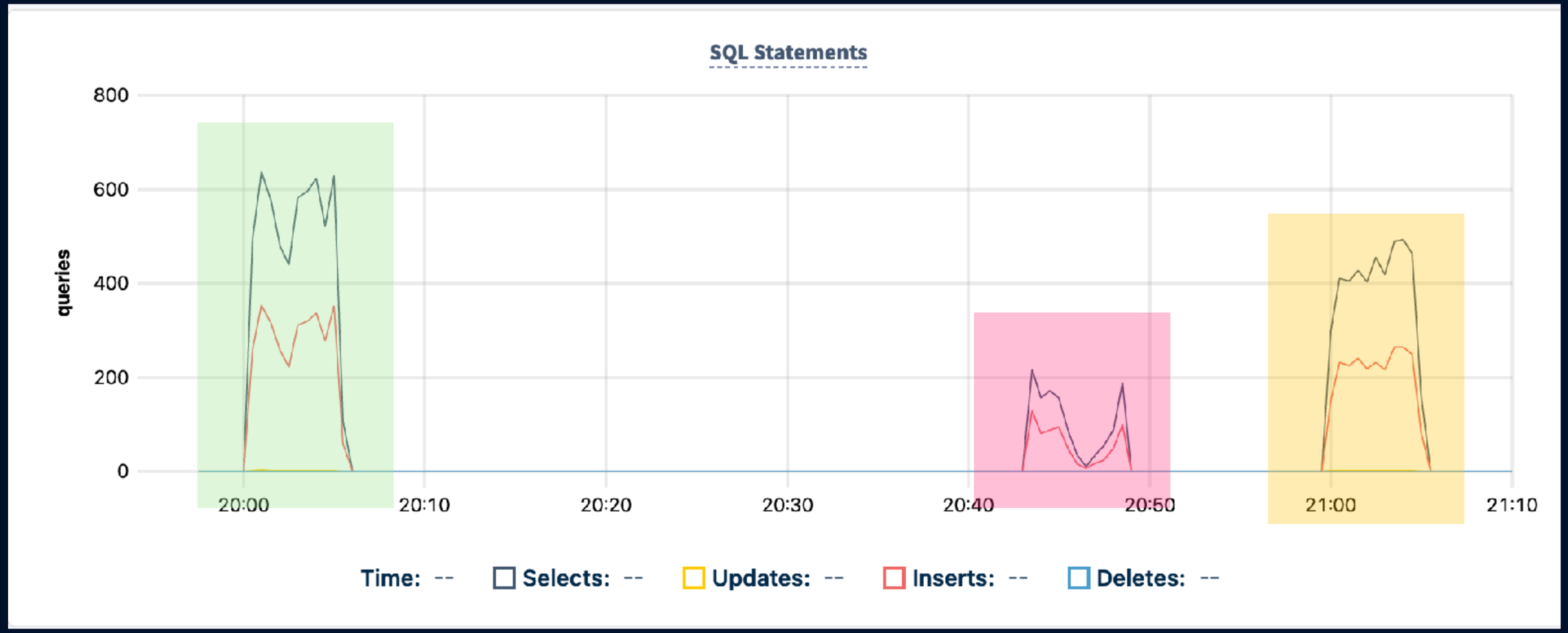
Drop Column - pgrocks (fdw)



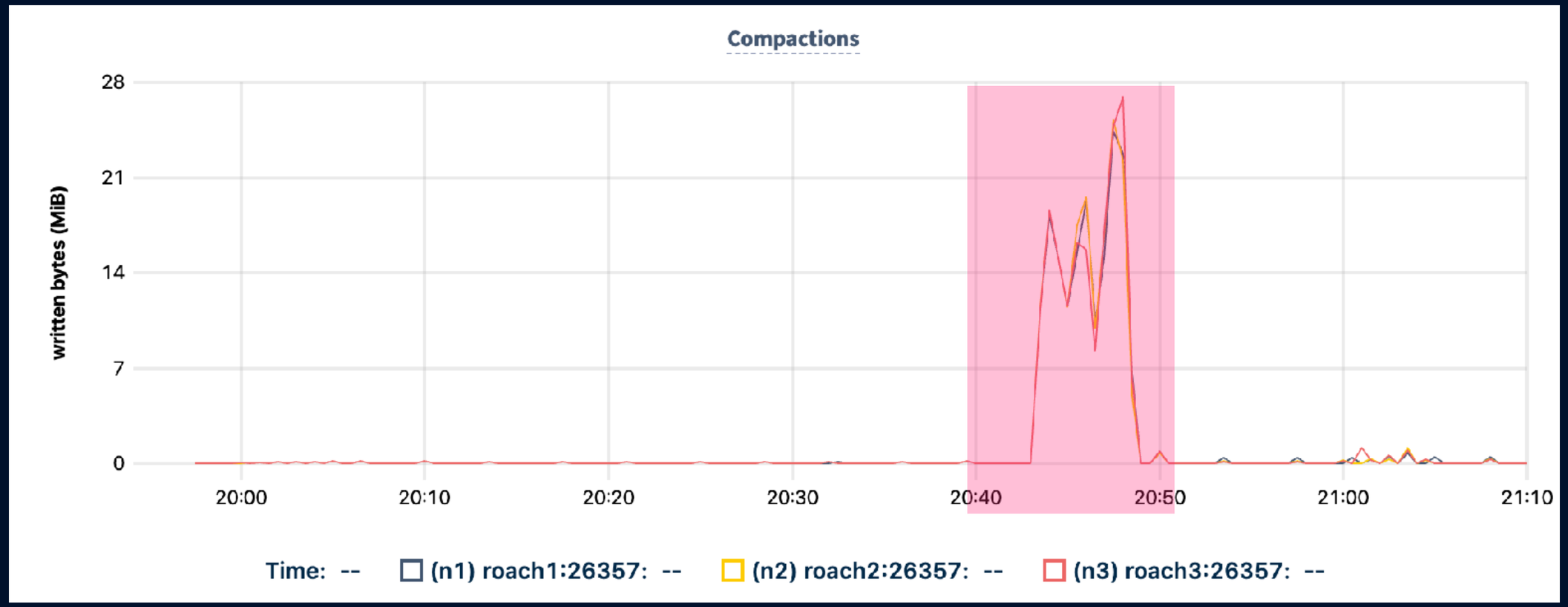
Nice Dashboards ... YugabyteDB



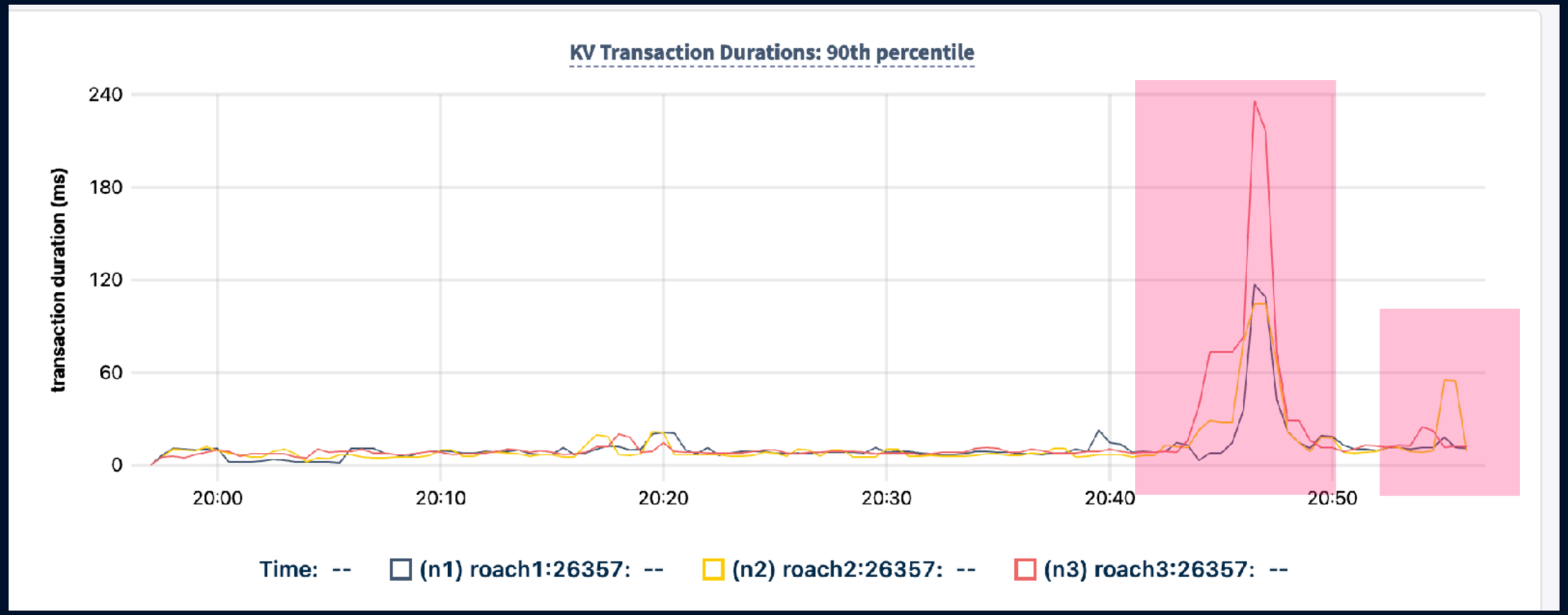
CockroachDB Introspection



CockroachDB Introspection



Time to play ... CockroachDB



Truncate Table - pgrocks (fdw)

Session One - pgbench

```
pgbench -P 1  -T 300
```

Session Two - Truncate

```
# truncate table big_table;

TRUNCATE TABLE

Time: 75.829 ms
```

Truncate Table - pgrocks (fdw)

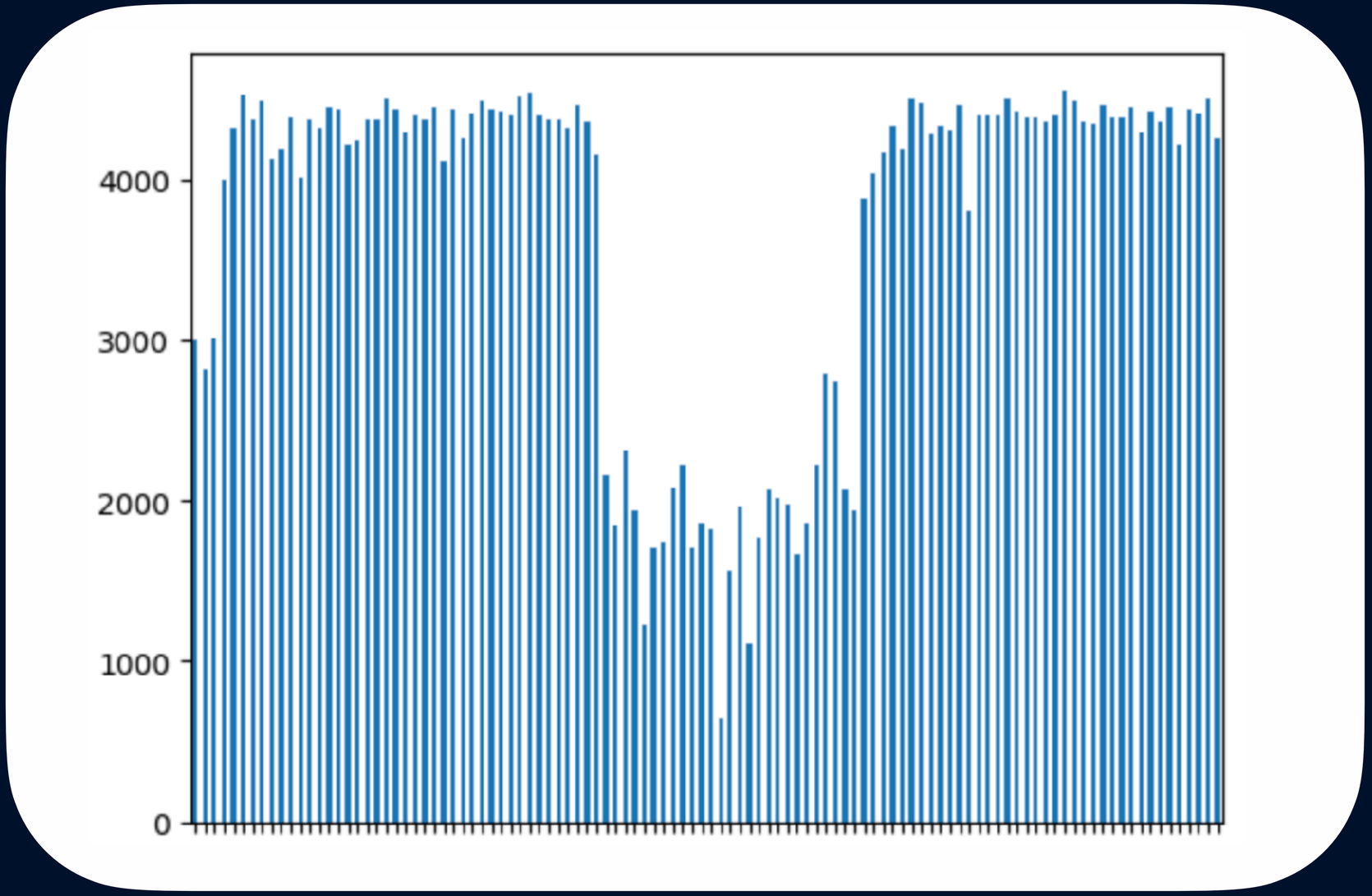
Session One - pgbench

```
pgbench -P 1  -T 300
```

Session Two - Truncate

```
# truncate table big_table;

TRUNCATE TABLE

Time: 75.829 ms
```

Truncate Table - pgrocks (fdw)



Conclusion

LSM trees are going to be very useful for certain workloads and distributed databases

There are some interesting edge cases (e.g. drop column, truncate, long transactions, read amplification, distributed reads ..)

Beware being sold easy fixes to problems like traditional vacuum

It may look like the Postgres we know and love, but under the hood it quite different

Q & A

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