OTTERTUNE

The Part of PostgreSQL I Hate the Most

what are the drawbacks of **MVCC** and how to optimize it

Bohan Zhang
Cofounder, OtterTune

01. **Background**

02. **MVCC in PostgreSQL**

03. **Problems & Optimizations**

01. Background

01. We Love Postgres !

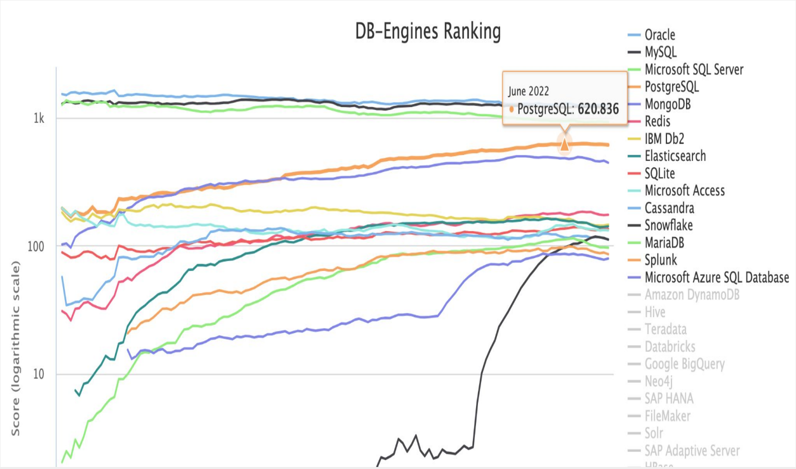
It's the **fourth** most popular database.

It's actually **a lot more popular** than you might expect.

> In OtterTune, we have **roughly the same** number of Postgres RDS and MySQL RDS under our management

Its popularity has been **significantly increasing** over the last five years. This trend will continue.

> It's open-source, feature-rich, extensible, and well-suited for complex queries.



source: DB-Engines Ranking

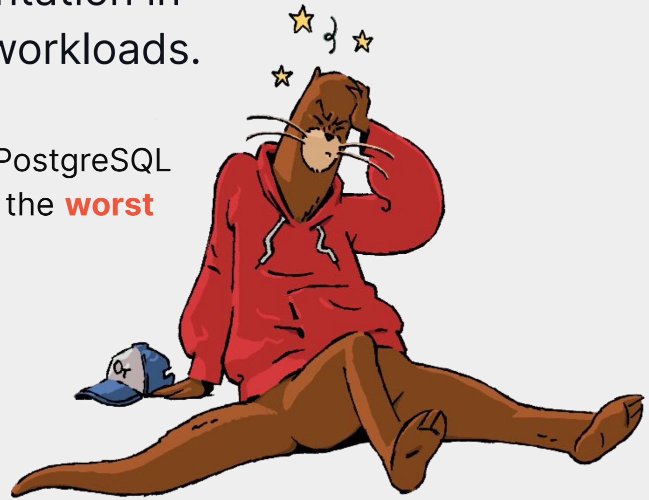
01. But certain aspects are not that great...

But as much as we love PostgreSQL at OtterTune, certain aspects are not that great.

> Instead of talking about how awesome everyone's favorite DBMS is, I want to discuss the one major thing that sucks

The **multi-version concurrency control (MVCC)** implementation in Postgres can cause severe performance issues for some workloads.

> Our research at Carnegie Mellon University and experience optimizing PostgreSQL databases in OtterTune have shown that their MVCC implementation is the **worst** among other widely used DBMSs



01. What is MVCC

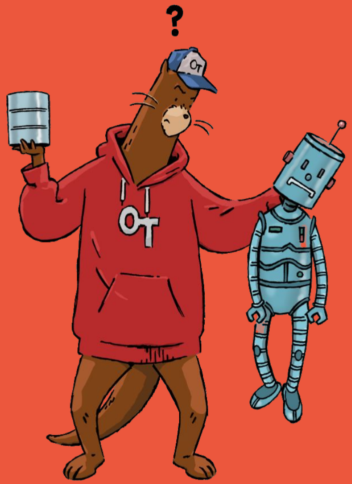
When a query updates an existing row in a table, the DBMS **makes a copy** of that row and **applies the changes** to this new version instead of overwriting the original version.

Readers do not block writers, and writers do not block readers.

> Increase the DBMS throughput
> Reduce the query latency

No free lunch. It introduces **additional overhead and issues**.

> Maintain multiple versions in storage
> Find the latest version
> Clean up "expired" versions



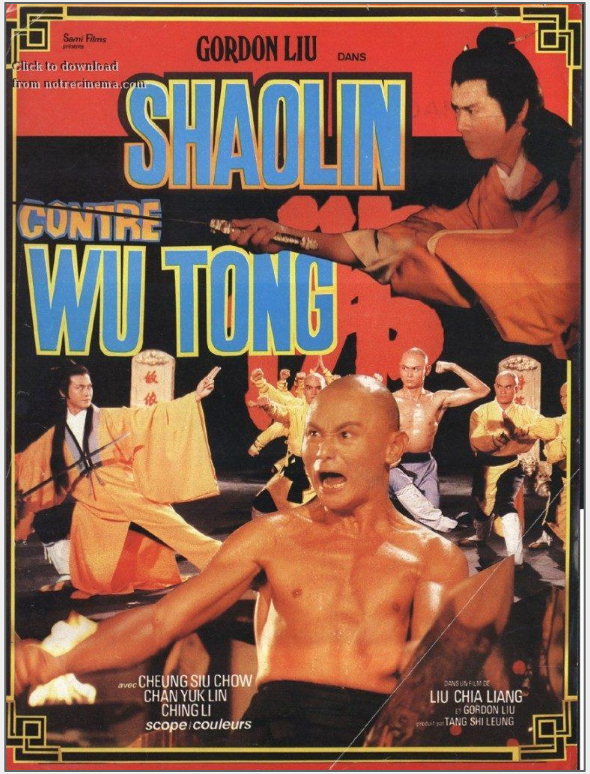
02. MVCC in PostgreSQL

03. Kung Fu Movies

Primary Index | Secondary Index | Secondary Index

id	name	year	director
1	Shaolin and Wu Tang	1985	Chia-Hui Liu
2	Executioners from Shaolin	1977	Chia-Liang Liu
3	Five Deadly Venoms	1978	Cheh Chang

```sql
CREATE TABLE movies (
  id SERIAL PRIMARY KEY,
  name TEXT,
  year INT,
  director VARCHAR(128)
);
CREATE INDEX idx_name ON movies (name);
CREATE INDEX idx_director ON movies (director);
```



03. Multi-Version Storage

```
UPDATE movies
   SET year = 1983
 WHERE name = 'Shaolin and Wu Tang'
```

id	name	year	director
1	Shaolin and Wu Tang	1985	Chia-Hui Liu
2	Executioners from Shaolin	1977	Chia-Liang Liu
3	Five Deadly Venoms	1978	Cheh Chang

id	name	year	director	
1	Shaolin and Wu Tang	1985	Chia-Hui Liu	◀ Old
2	Executioners from Shaolin	1977	Chia-Liang Liu	
3	Five Deadly Venoms	1978	Cheh Chang	
1	Shaolin and Wu Tang	1983	Chia-Hui Liu	◀ New

Postgres **makes a copy** of that row and **applies the changes** to this new version.

All row versions in a table are stored in the same storage space.

Known as **append-only** version storage schema.

03. O2N version chain

```
SELECT * FROM movies
 WHERE name = 'Shaolin and Wu Tang'
```

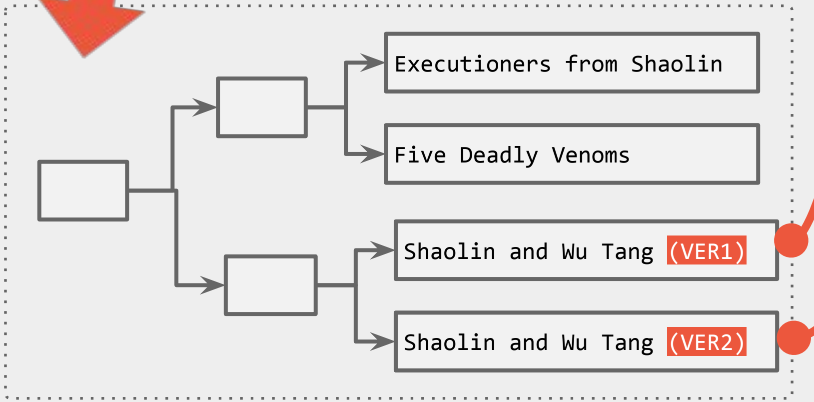
id	name	year	director	*next ver*
1	Shaolin and Wu Tang	1985	Chia-Hui Liu	
2	Executioners from Shaolin	1977	Chia-Liang Liu	-
3	Five Deadly Venoms	1978	Cheh Chang	-
1	Shaolin and Wu Tang	1983	Chia-Hui Liu	-

Oldest-to-Newest Version Chain

Each tuple version points to its new version, and the head is the oldest tuple version.
Known as **Oldest-to-Newest (O2N)** version chain
Postgres traverses the version chain to find the latest version.

03. Index

```
UPDATE movies
   SET year = 1983
 WHERE name = 'Shaolin and Wu Tang'
```



Index (movies.name)

id	name	year	director	*next ver*
1	Shaolin and Wu Tang	1985	Chia-Hui Liu	
2	Executioners from Shaolin	1977	Chia-Liang Liu	-
3	Five Deadly Venoms	1978	Cheh Chang	-

Table Page #1

id	name	year	director	*next ver*
1	Shaolin and Wu Tang	1983	Chia-Hui Liu	-
				-
				-

Table Page #2

PostgreSQL adds an entry to **every** table's index for **each** physical version of a row.

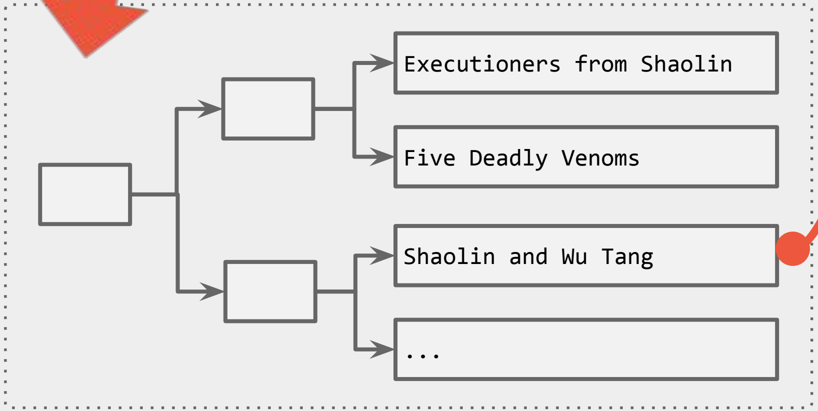
avoid having to traverse the entire version chain to get the latest version

03. HOT optimization

```sql
UPDATE movies
   SET year = 1983
 WHERE name = 'Shaolin and Wu Tang'
```

HOT (heap-only tuple) update:

> an update does not modify any columns referenced by table's indexes
> the new version is stored on the same data page as the old version



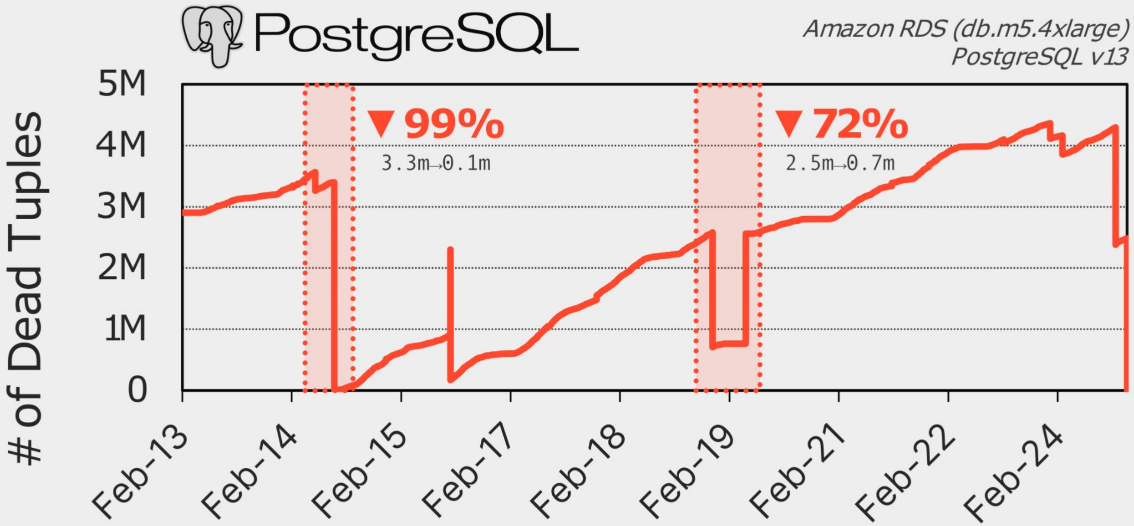
Index (movies.name)

id	name	year	director	*next ver*
1	Shaolin and Wu Tang	1985	Chia-Hui Liu	
2	Executioners from Shaolin	1977	Chia-Liang Liu	-
3	Five Deadly Venoms	1978	Cheh Chang	-
1	Shaolin and Wu Tang	1983	Chia-Hui Liu	-

Table Page #1

The index still points to the old version. Do not need to maintain indexes.
During normal operation, Postgres removes old versions to prune the version chain.

03. Autovacuum



PostgreSQL uses a vacuum procedure to clean up dead tuples from tables.
PostgreSQL automatically executes this vacuum procedure at regular intervals.

03. Problems & Optimizations

03. Version Copying

When a query updates a tuple, **all of its columns** are copied into the new version.

> Regardless of whether the query updates **a single column** or all columns.
> What if a table has 1000 columns?

id	name	year	director	*next ver*
1	Shaolin and Wu Tang	1985	Chia-Hui Liu	
2	Executioners from Shaolin	1977	Chia-Liang Liu	-
3	Five Deadly Venoms	1978	Cheh Chang	-
1	Shaolin and Wu Tang	1983	Chia-Hui Liu	-

This results in **massive data duplication** and **increased storage requirements**.

https://github.com/orioledb

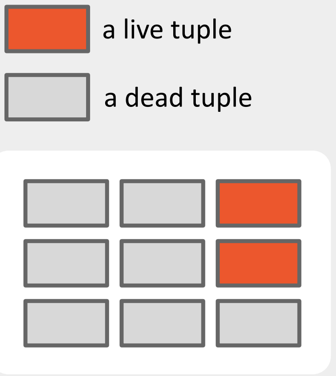
03. Table Bloat

The DBMS has to **load dead tuples** into memory during query execution.

> It intermingles dead tuples with live tuples in pages
> Page is the smallest unit when fetching data into memory

This causes the DBMS to **incur more IOPS** and **consume more memory** than necessary during table scans.

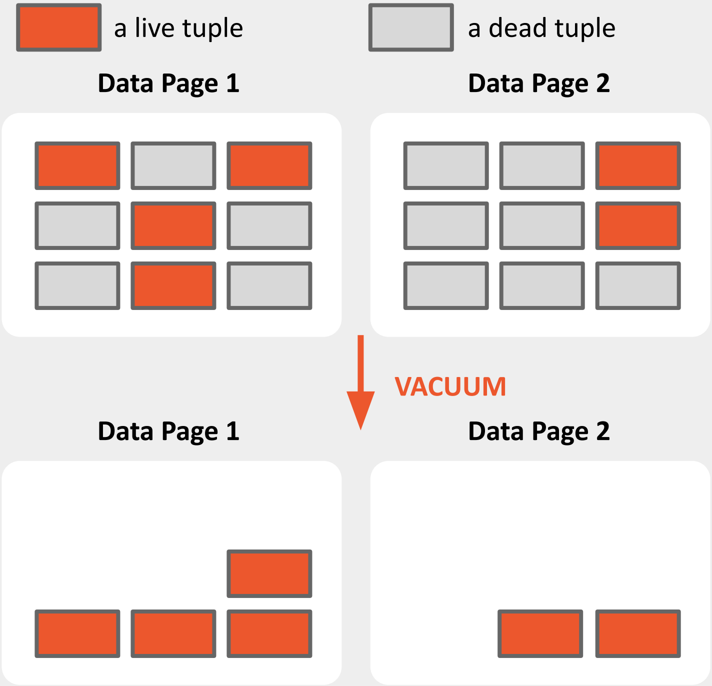
Inaccurate optimizer statistics caused by dead tuples can lead to poor query plans.



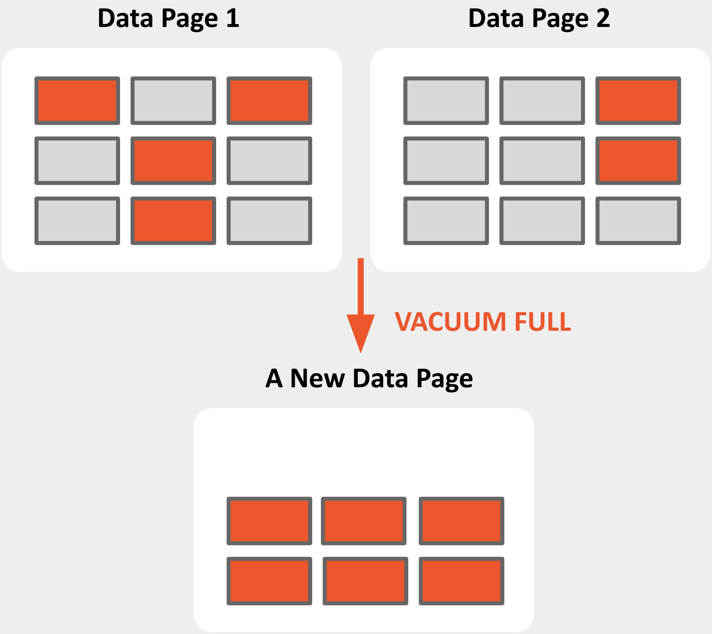
Data Page

(2 live tuples, 7 dead tuples)

03. Table Bloat



(a). VACUUM



(b). VACUUM FULL

VACUUM does not return unused space to OS

VACUUM FULL can return unused space, but it's **time-consuming** and **resource-intensive**

OPTIMIZATION: Monitor the database bloat ([pgstattuple]) and reclaim unused space ([pg_repack]).

03. Index Maintenance

For non-HOT updates, PostgreSQL needs to modify **ALL** of indexes in the table for each update.

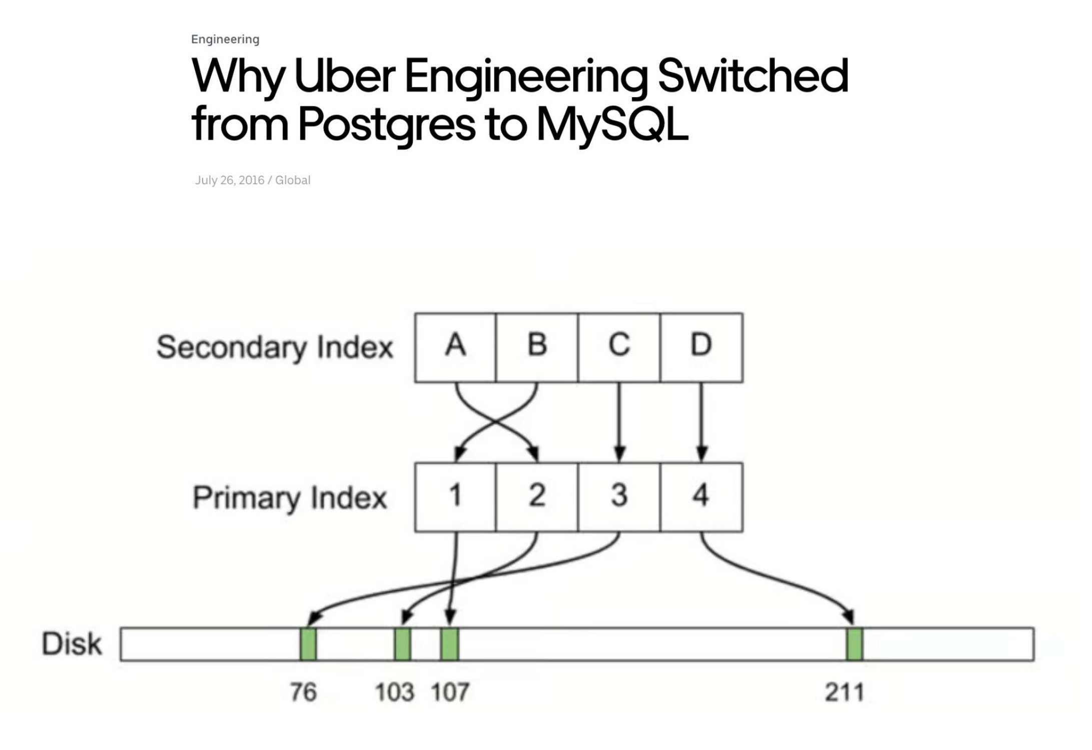
> What if a table has dozens of indexes?
> Significant index maintenance overhead and write amplification

OtterTune customers' PostgreSQL databases shows that roughly **46% of updates** use the HOT optimization on average.

OPTIMIZATION: Drop duplicate and unused indexes in tables. pg_stat_all_indexes

03. Index Maintenance

Uber migrate from Postgres to MySQL



https://www.uber.com/blog/postgres-to-mysql-migration/

Side Comment:
Oldest-to-Newest (O2N)
version chain,

Not N2O version chain erroneously stated in blog

03. Vacuum Management

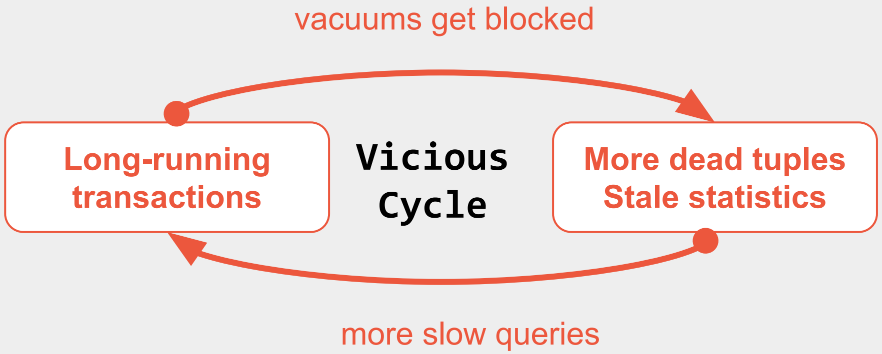
Making sure that PostgreSQL's autovacuum is running as best as possible is **difficult** due to its complexity.

> Default settings for tuning the autovacuum are not ideal for all tables, particularly for large ones
> **autovacuum_vacuum_scale_factor** default value is 20%
> If a table has 100 million tuples, it needs 20 million dead tuples before the autovacuum kicks in

OPTIMIZATION:
Fine-tune the autovacuum settings at the table level, particularly for large tables. pg_stat_all_tables

03. Vacuum Management

Autovacuum can be blocked by long-running transactions, requiring humans to intervene manually.



Case Study: [ANALYZE after bulk insertions]. The long query's execution time went from **52 minutes to just 34 seconds** after optimization.

OPTIMIZATION:

Identify and resolve long-running transactions promptly. pg_stat_activity

Identify and optimize prolonged vacuum processes. pg_stat_progress_vacuum

Try OtterTune for free:
https://ottertune.com/try

END

amazon

MySQL

PostgreSQL

bohan@ottertune.com