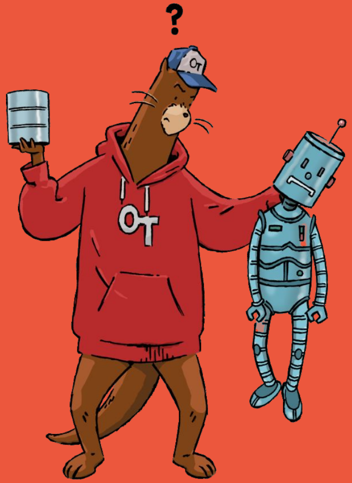
Everything you need to know about autovacuum

Bohan Zhang
Cofounder, OtterTune



01. MVCC in PostgreSQL

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**.

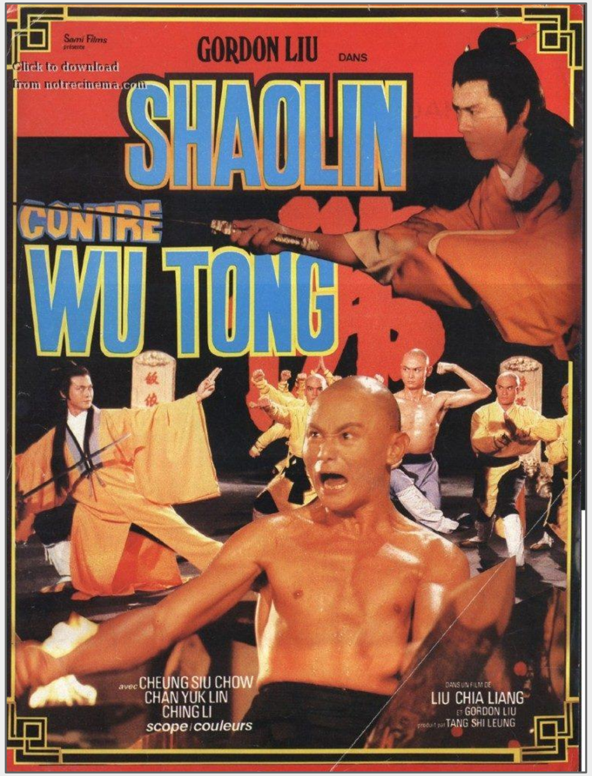
> Maintain multiple versions in storage
> Find the latest version
> Clean up "expired" versions (autovacuum)

01. Kung Fu Movies

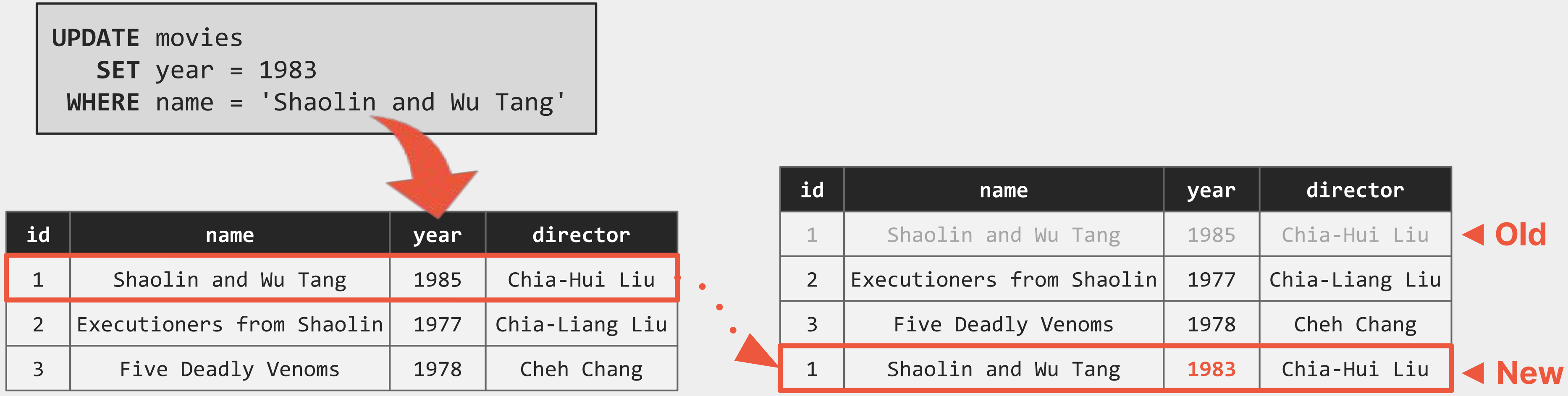
Primary Index → id
Secondary Index → name
Secondary Index → director

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

```
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);
```



01. 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
2	Executioners from Shaolin	1977	Chia-Liang Liu
3	Five Deadly Venoms	1978	Cheh Chang
1	Shaolin and Wu Tang	1983	Chia-Hui Liu

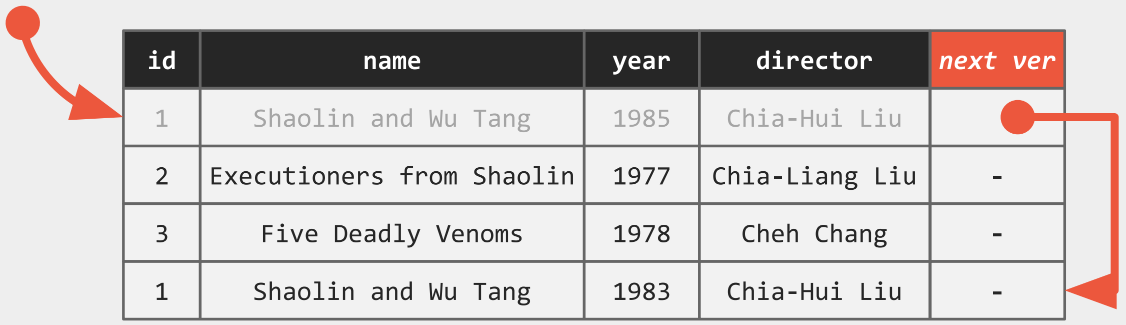
Postgres **makes a copy** of that row and **applies the changes** to this new version.

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

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

01. 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.

01. Index

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

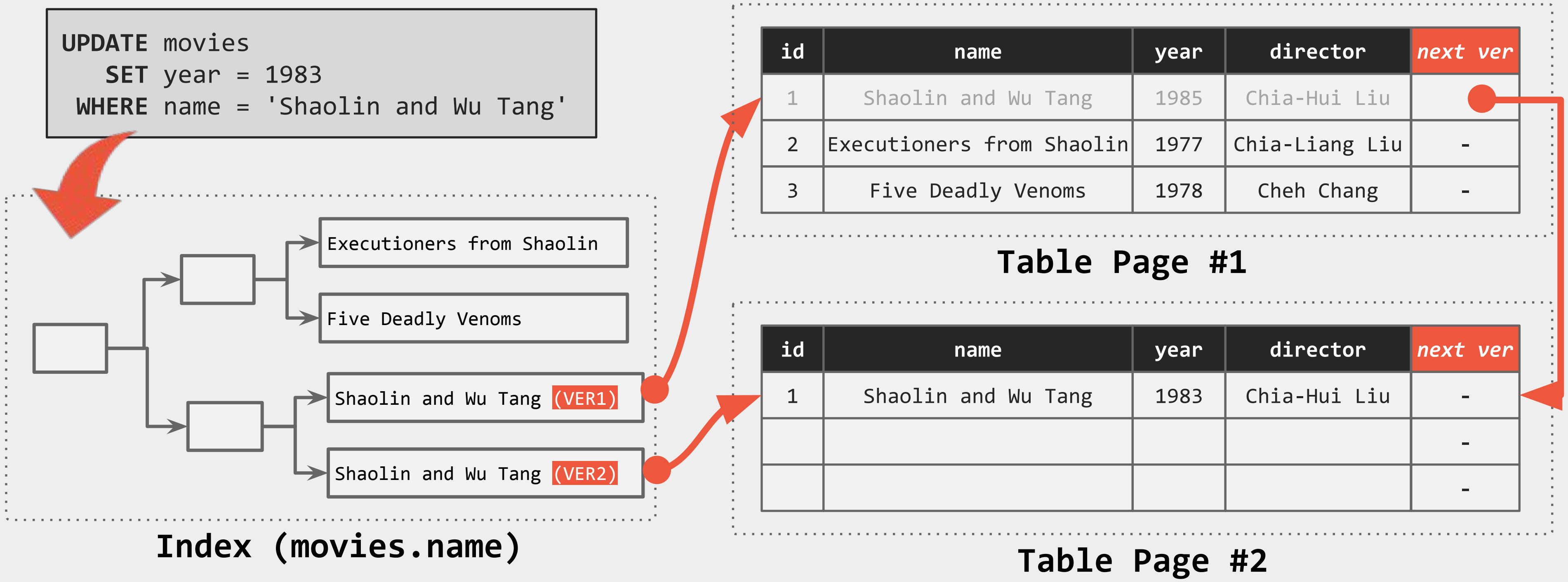

Executioners from Shaolin
Five Deadly Venoms
Shaolin and Wu Tang (VER1)
Shaolin and Wu Tang (VER2)
Index (movies.name)
Table Page #1
Table Page #2

Table Page #1

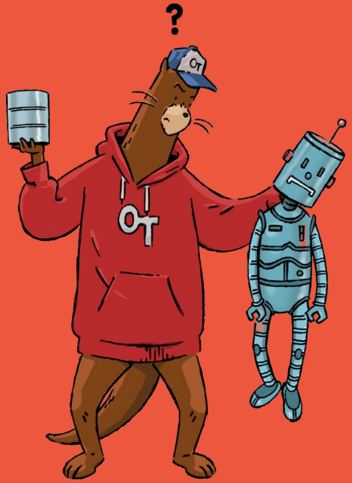
id	name	year	director	next ver
1	Shaolin and Wu Tang	1985	Chia-Hui Liu	●
2	Executioners from Shaolin	1977	Chia-Liang Liu	-
3	Five Deadly Venoms	1978	Cheh Chang	-

Table Page #2

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

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



02. Autovacuum in PostgreSQL

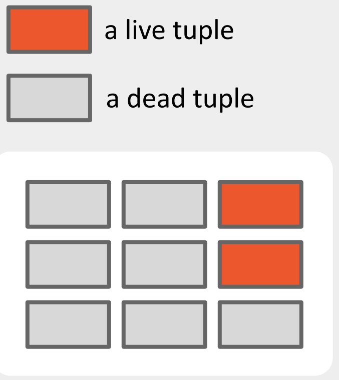
02. Table Bloat

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.

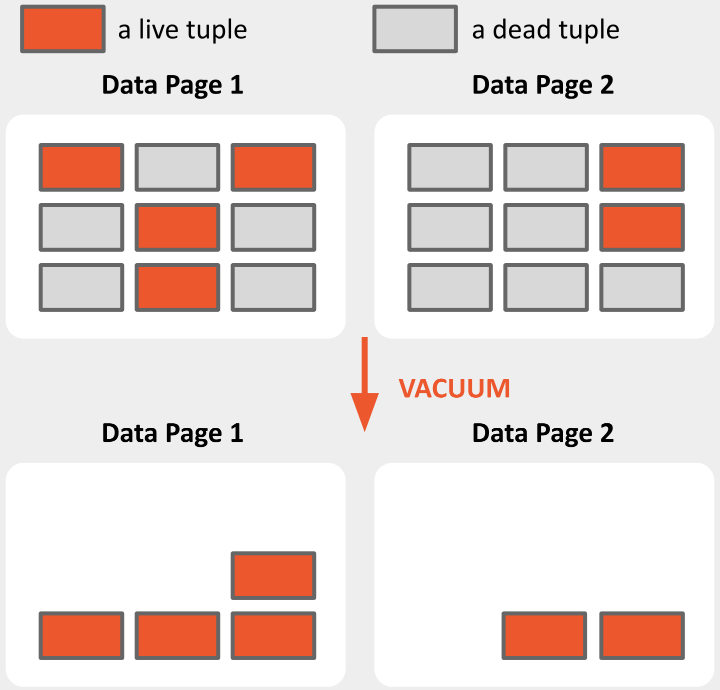
Inaccurate optimizer statistics caused by dead tuples can lead to poor query plans.



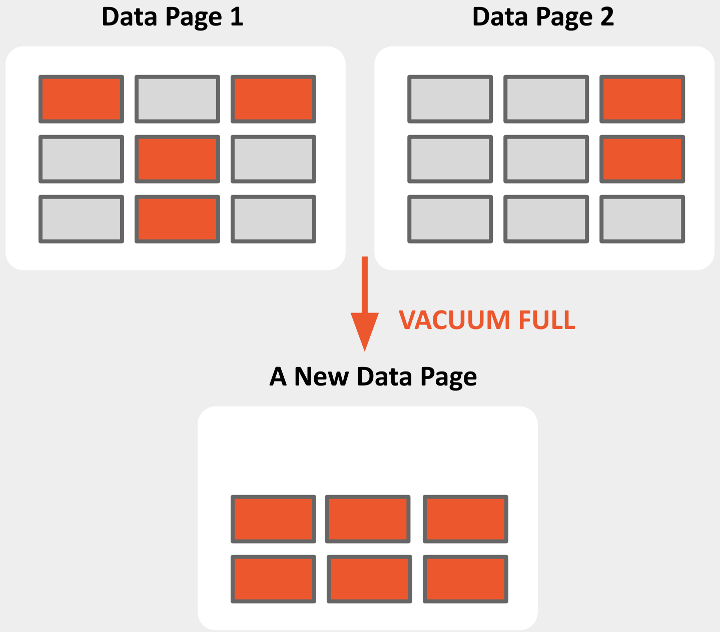
Data Page

(2 live tuples, 7 dead tuples)

02. 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**

02. Vacuum

Vacuum: reclaims storage occupied by dead tuples and makes it available for reuse

Vacuum process has 3 main phases:

- **Scan Heap**: scan a target table to build a list of dead tuples stored in the local memory
- **Vacuum Index**: remove index tuples by referring to the dead tuple list
- **Vacuum Heap**: remove dead tuples from the heap

02. Analyze

Analyze: updates statistics used by the planner to determine the most efficient way to execute a query

A user had an ETL workload running scripts that **bulk-loaded** data, and then ran queries to aggregate values

```
Seq Scan on mrt_contributions (cost=0.00..242575.00 rows=1 width=4)
(actual time=0.006..1955.159 rows=3474601 loops=1)
```

The optimizer was choosing **seq scan** in the query plan, because it thought there was only one row. After manually running Analyze on the table, it used indexes for **hash** lookup. The job time went from **52 minutes to 34 seconds**

Run ANALYZE. Run ANALYZE. Run ANALYZE.

02. Autovacuum

Automate the execution of VACUUM and ANALYZE commands

Autovacuum checks for tables that have had a large number of dead tuples. If it exceeds the vacuum(or analyze) threshold, VACUUM (or ANALYZE) is triggered

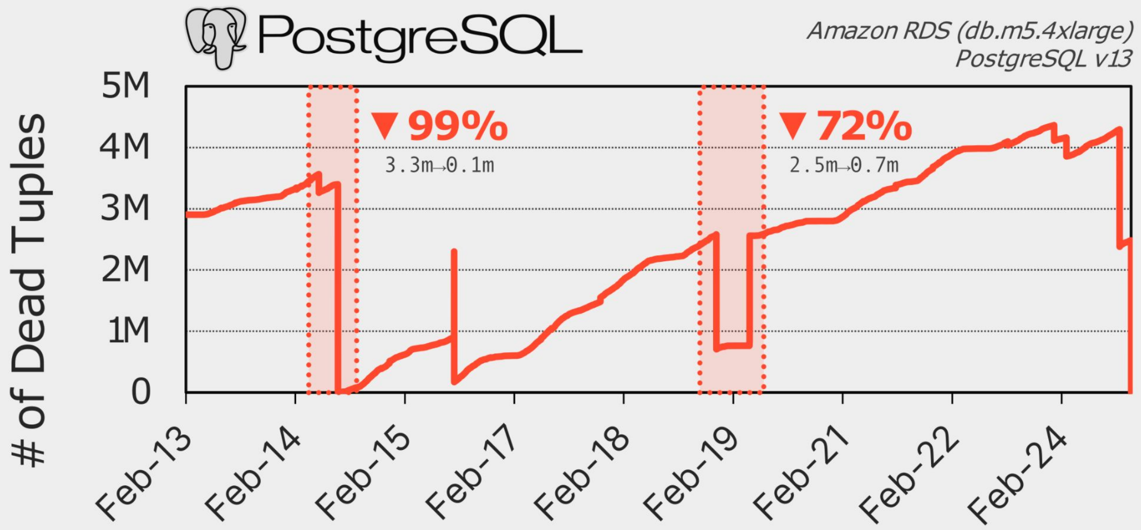
vacuum threshold = vacuum base threshold + vacuum scale factor * number of tuples

⇧ ⇧

autovacuum_vacuum_threshold (50) **autovacuum_vacuum_scale_factor** (0.2)

Similar for analyze threshold: autovacuum_analyze_threshold, autovacuum_analyze_scale_factor

02. Autovacuum



PostgreSQL automatically executes the vacuum procedure to clean up dead tuples

02. Table-Level Tuning

vacuum threshold = base threshold (50) + scale factor (0.2) * number of tuples

If a table has 1000 tuples, vacuum threshold = 50 + 0.2 * 1000 = 250

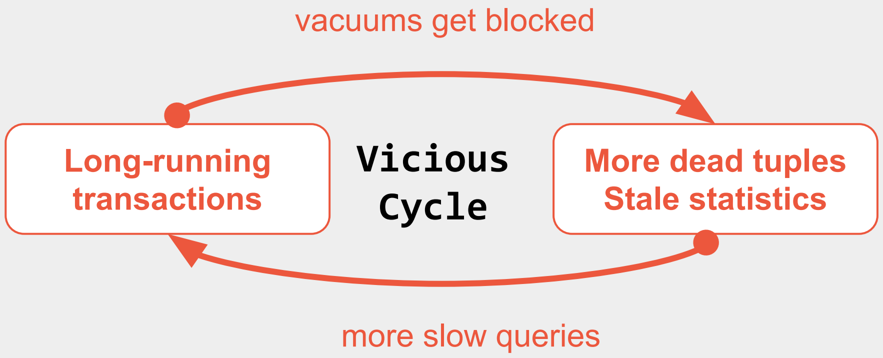
If a table has 1 billion tuples, vacuum threshold is roughly 200 million tuples

Default settings are not suitable for very **large tables**, you should set the scale factor knob for large tables to a smaller value

```
ALTER TABLE table_name SET (autovacuum_ vacuum_scale_factor = 0.05);
```

02. Long Running Transactions

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

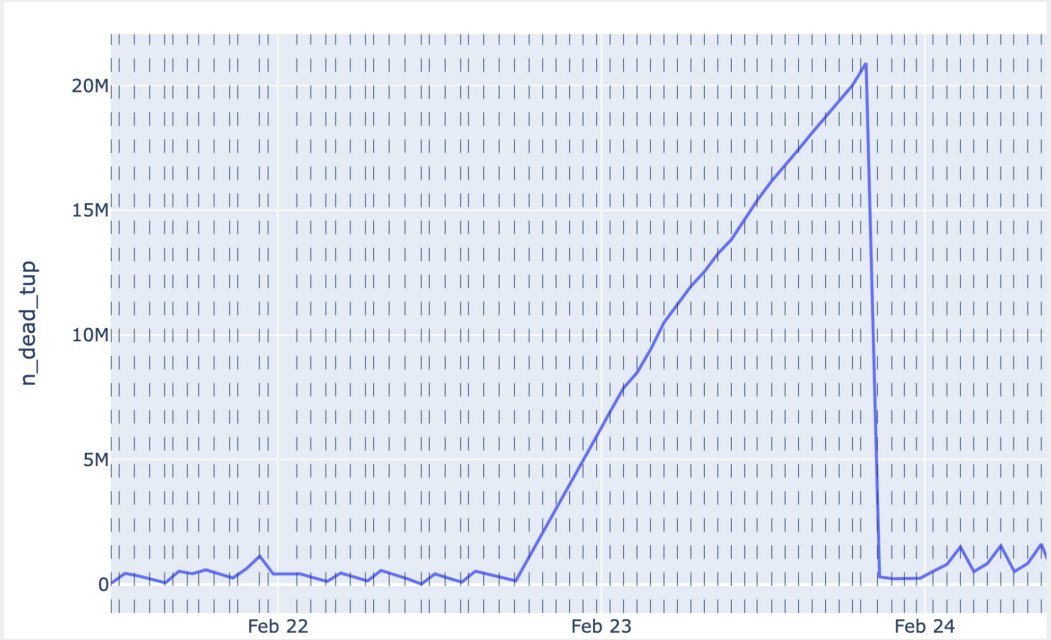


Identify and resolve long-running transactions promptly. pg_stat_activity
Identify and optimize prolonged vacuum processes. pg_stat_progress_vacuum

03. Case Studies

03. Case Study

Is autovacuum really vacuuming the table?



Dash lines show the timestamps when the table was auto-vacuumed

last_autovacuum: Last time this table was vacuumed by the autovacuum daemon (pg_stat_all_tables

The last_autovacuum keeps updated, but the tables are not successfully vacuumed

03. Case Study

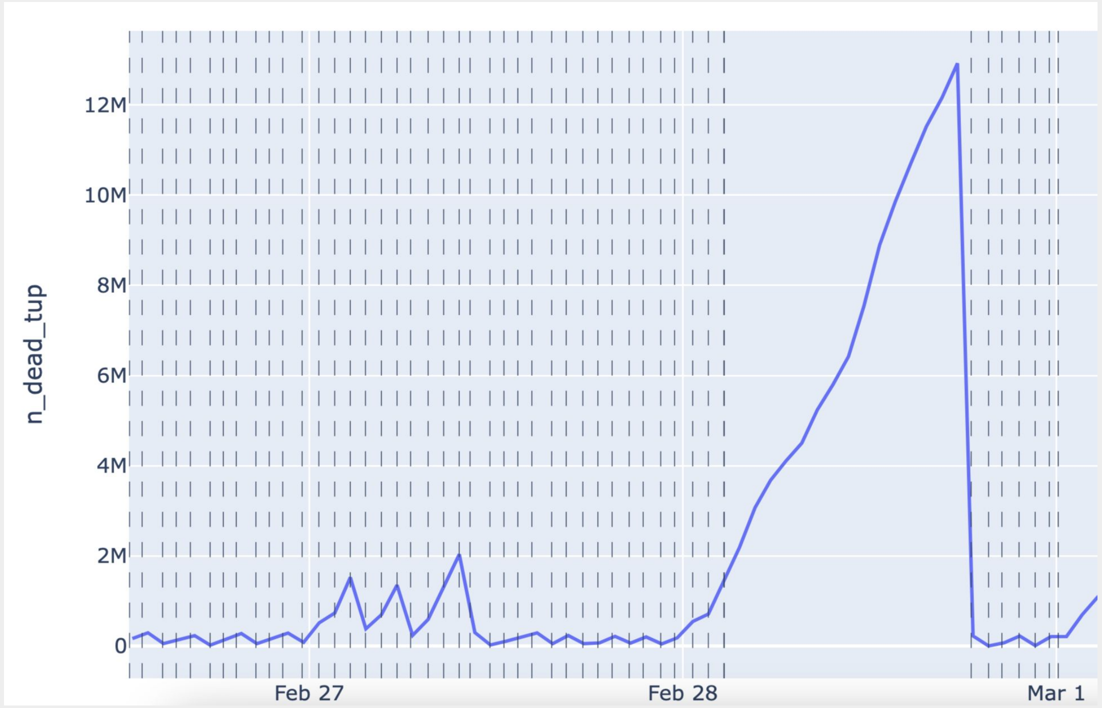
Is autovacuum really vacuuming the table?

```
LOG: automatic vacuum of table "postgres.public.test": index scans: 0
pages: 0 removed, 27 remain, 1 skipped due to pins, 0 skipped frozen
tuples: 0 removed, 5856 remain, 3870 are dead but not yet removable, oldest xmin: 963
index scan not needed: 0 pages from table (0.00% of total) had 0 dead item identifiers removed
I/O timings: read: 0.000 ms, write: 0.000 ms
avg read rate: 0.000 MB/s, avg write rate: 0.000 MB/s
buffer usage: 98 hits, 0 misses, 0 dirtied
WAL usage: 1 records, 0 full page images, 188 bytes
system usage: CPU: user: 0.00 s, system: 0.00 s, elapsed: 0.00 s
```

Even if an autovacuum fails to remove any dead tuple, ***last_autovacuum*** will still be updated
We cannot depend solely on ***last_autovacuum*** to know whether dead tuples have been removed

03. Case Study

Why autovacuum is not triggered?



Dash lines show the timestamps when the table was auto-vacuumed (*last_autovacuum*)

Autovacuum was not triggered on the table

03. Case Study

Why autovacuum is not triggered?

autovacuum_max_workers: specifies the maximum number of autovacuum processes that may be running at any one time. The default is three.

All autovacuum workers are busy with vacuuming other tables

Increase the maximum number of autovacuum workers

Speedup the autovacuum process

> it estimates the cost of autovacuum operations; if it surpasses a threshold, it will sleep some time.
> decrease **autovacuum_vacuum_cost_delay** to make the sleeps shorter
> increase **autovacuum_vacuum_cost_limit** to make the sleeps happen less frequently

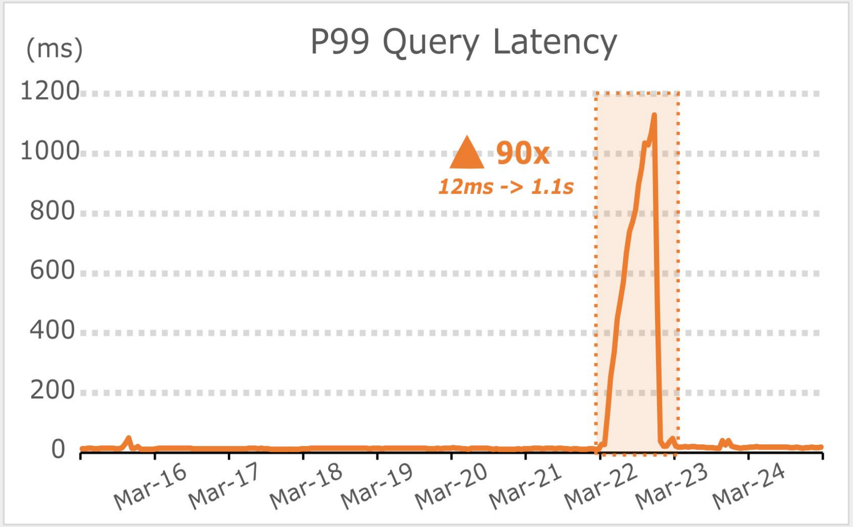
Try OtterTune for free:
https://ottertune.com/try

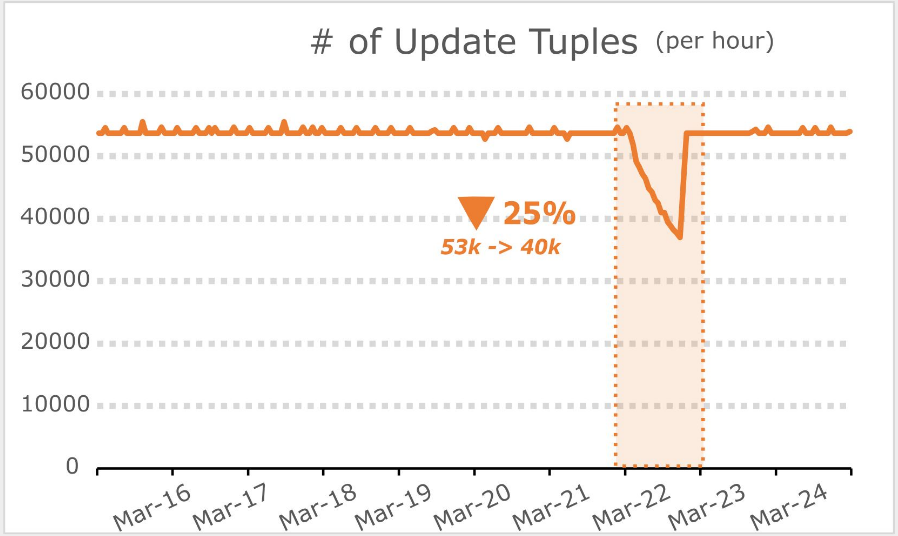
END

---

bohan@ottertune.com

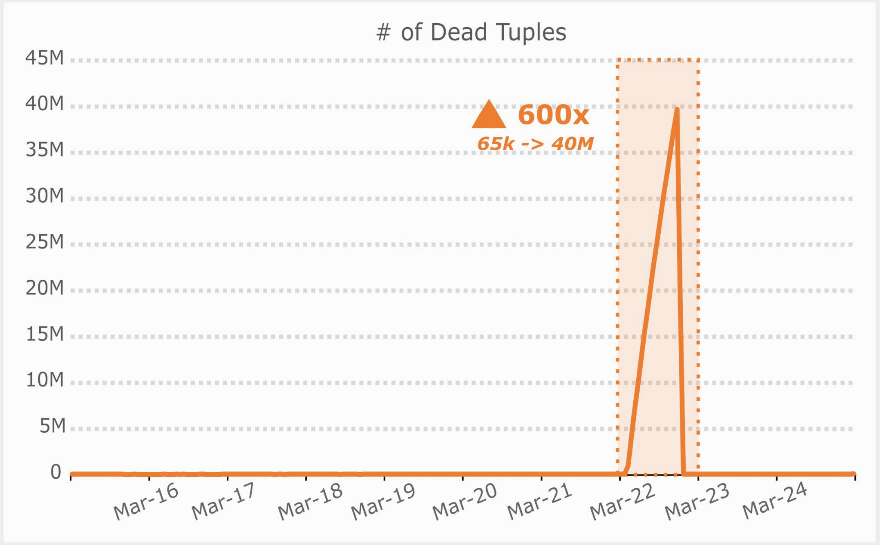
03. Case Study

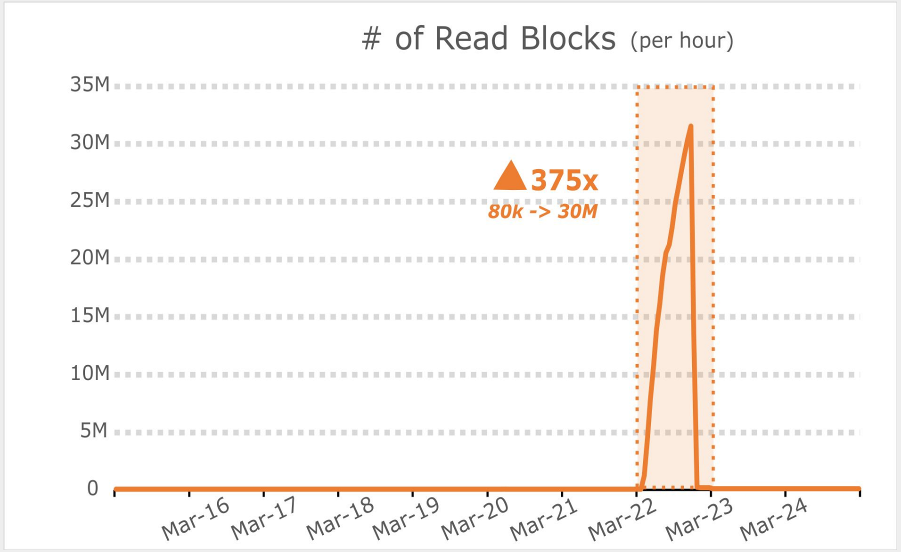




- Massive latency spikes
- Update-heavy workload; Only few insert/delete queries
- For update queries: latency increased **from 12ms to 710ms** (60x), throughput dropped by 25% during the spike

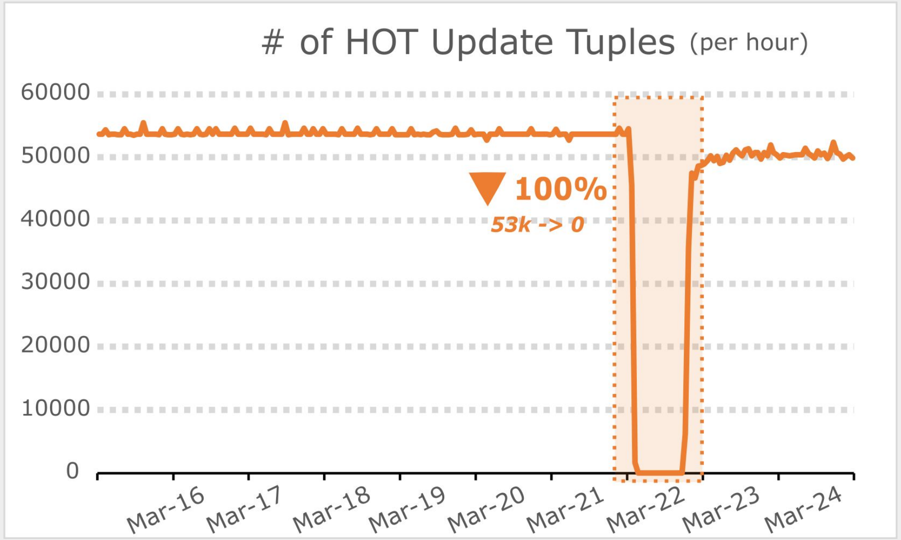
03. Case Study

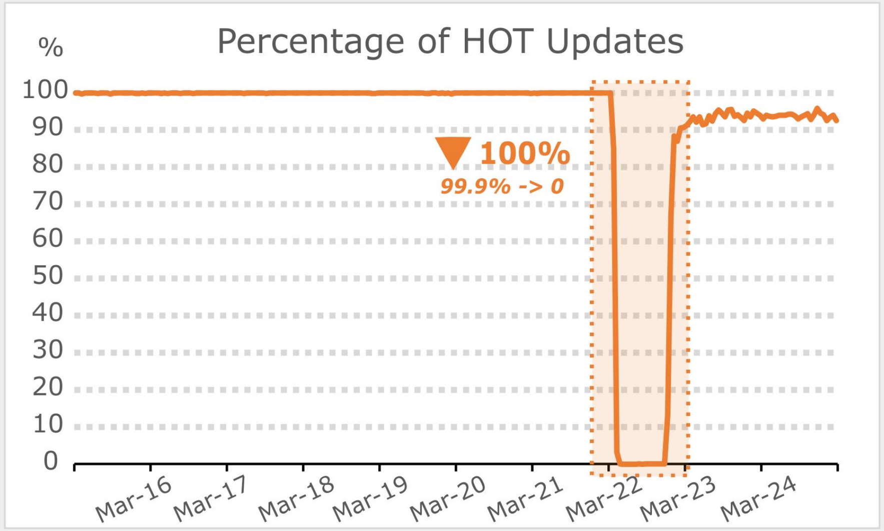




- The number of dead tuples increased a lot. Tables were not vacuumed successfully
- Consequently, more data were read from memory and disk

03. Case Study





- The number of HOT (heap-only tuple) updates dropped to 0
- Updates were more expensive during the spike time because HOT optimization was not applied

03. Case Study

Long running queries block autovacuum

↓

Dead tuples accumulate (**600x**)

↓

Significant increase in blocks read (**375x**) and non-HOT updates (**100%**)

↓

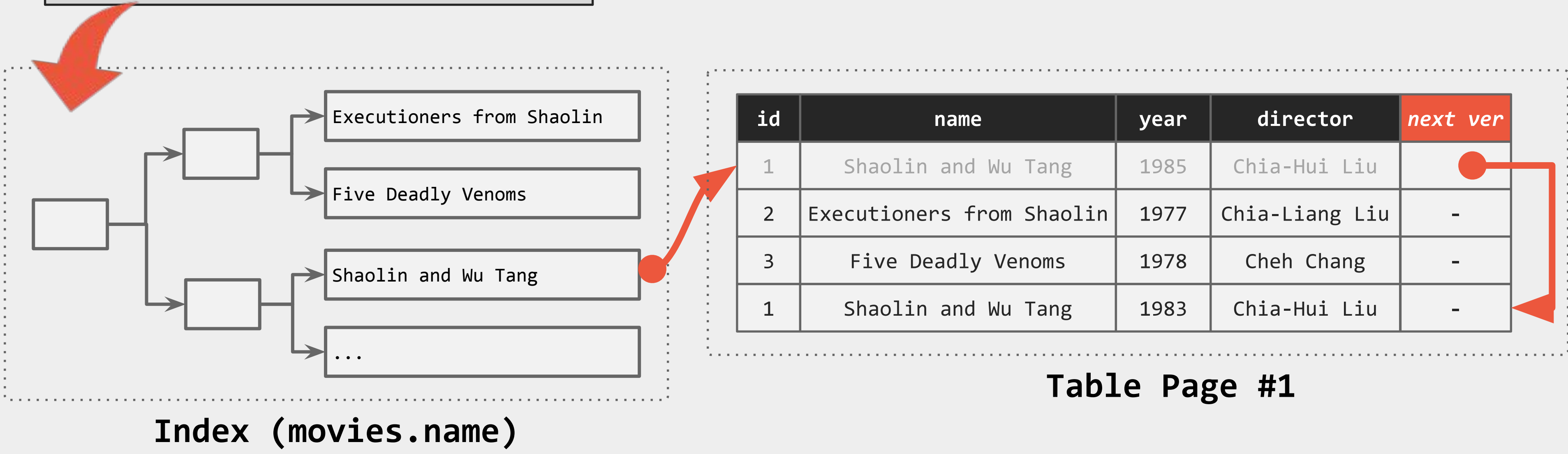
Query latency spikes (**90x**)

02. HOT (Heap-only Tuple) optimization

```
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



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	-

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.