Managing Your Tuple Graveyard

Chelsea Dole
cdole@brex.com
Senior Software Engineer, Brex
◆ “The credit card for startups”, expense management software
◆ Previously: Data Engineer, Backend Engineer

Tech Lead, Data Storage Team
◆ Postgres infrastructure
◆ Query optimization
◆ & more!

Chelsea Dole
Outline

1. Multi-Version Concurrency Control (MVCC)
   a. What is MVCC and why does it need to exist?
   b. Vacuum, live vs dead tuples, and more

2. Table bloat
   a. What is it, what causes it, and how does it impact databases?
   b. Case study

3. Quantifying, mitigating, and avoiding table bloat
   a. pgstattrange, pg_repack, ...
   b. Autovacuum configuration

4. Designing bloat-aware data access patterns
1. MVCC

(Multi-Version Concurrency Control)
What is MVCC?

Multi-Version Concurrency Control:
A set of rules through which Postgres provides two important (yet seemingly contradictory) features:

1. Transaction isolation
2. Fast performance
Transaction Isolation

- The “I” in ACID
  - Atomic, Consistent, Isolated, Durable
- Data within a transaction represents table state at transaction start

Fast Performance

- Writes don’t block reads
- Reads don’t block writes
Why are these goals contradictory?

TLDR; locks ensure transaction isolation, but lead to cascading locks/waits (and therefore bad performance)

→ EX: Basic Locking
  ◆ Most straightforward way to ensure transaction isolation
  ◆ Not compatible with performance concurrent operations
MVCC’s approach

➔ “Row versioning” via tuples

➔ All DML operations create new tuple(s) or update tuple metadata only

◆ INSERT, UPDATE, DELETE, MERGE

<table>
<thead>
<tr>
<th>Tuple</th>
</tr>
</thead>
<tbody>
<tr>
<td>A physical, immutable “row” stored on disk.</td>
</tr>
<tr>
<td>A &quot;row&quot; is a logical construct consisting of 1 to n tuples under the hood, representing the data over time.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Live Tuple</th>
</tr>
</thead>
<tbody>
<tr>
<td>Newest row version OR used by a running query</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Dead Tuple</th>
</tr>
</thead>
<tbody>
<tr>
<td>Old row version AND unused by running queries</td>
</tr>
</tbody>
</table>
MVCC’s approach

→ Transaction snapshots
→ Tuple visibility
  ◆ $xmin$ - TXID which inserted the tuple
  ◆ $xmax$ - TXID which updated/deleted the tuple
  ◆ $xip\_list$ - TXIDs of active transactions
→ TXID: assigned at transaction start

**Snapshot**

A data structure created on a per-transaction basis.

Uses $xmin$, $xmax$, and $xip\_list$ to determine which tuples are visible for the transaction.
<table>
<thead>
<tr>
<th>xmin</th>
<th>xmax</th>
<th>id</th>
<th>first_name</th>
<th>city</th>
<th>updated_at</th>
</tr>
</thead>
<tbody>
<tr>
<td>594</td>
<td></td>
<td>1</td>
<td>chelsea</td>
<td>seattle</td>
<td>2015-03-26T10:58:51</td>
</tr>
<tr>
<td>594</td>
<td></td>
<td>2</td>
<td>stephen</td>
<td>nashville</td>
<td>2021-07-23T21:11:48</td>
</tr>
<tr>
<td>594</td>
<td></td>
<td>3</td>
<td>selena</td>
<td>bellingham</td>
<td>2018-01-04T07:33:21</td>
</tr>
<tr>
<td>594</td>
<td></td>
<td>4</td>
<td>tommy</td>
<td>toronto</td>
<td>1998-09-17T04:03:02</td>
</tr>
<tr>
<td>594</td>
<td></td>
<td>5</td>
<td>adam</td>
<td>chicago</td>
<td>2017-04-15T10:07:52</td>
</tr>
</tbody>
</table>
### Example - INSERT

<table>
<thead>
<tr>
<th>xmin</th>
<th>xmax</th>
<th>id</th>
<th>first_name</th>
<th>city</th>
<th>updated_at</th>
</tr>
</thead>
<tbody>
<tr>
<td>594</td>
<td></td>
<td>1</td>
<td>chelsea</td>
<td>seattle</td>
<td>2015-03-26T10:58:51</td>
</tr>
<tr>
<td>594</td>
<td></td>
<td>2</td>
<td>stephen</td>
<td>nashville</td>
<td>2021-07-23T21:11:48</td>
</tr>
<tr>
<td>594</td>
<td></td>
<td>3</td>
<td>selena</td>
<td>bellingham</td>
<td>2018-01-04T07:33:21</td>
</tr>
<tr>
<td>594</td>
<td></td>
<td>4</td>
<td>tommy</td>
<td>toronto</td>
<td>1998-09-17T04:03:02</td>
</tr>
<tr>
<td>594</td>
<td></td>
<td>5</td>
<td>adam</td>
<td>chicago</td>
<td>2017-04-15T10:07:52</td>
</tr>
<tr>
<td>600</td>
<td></td>
<td>6</td>
<td>john</td>
<td>new york</td>
<td>2002-03-13T11:15:14</td>
</tr>
</tbody>
</table>

1. INSERT new tuple
   - xmin = current txid

TUPLE COUNT: 1
CURRENT TXID: 600
Example - UPDATE

<table>
<thead>
<tr>
<th>xmin</th>
<th>xmax</th>
<th>id</th>
<th>first_name</th>
<th>city</th>
<th>updated_at</th>
</tr>
</thead>
<tbody>
<tr>
<td>594</td>
<td>1</td>
<td>1</td>
<td>chelsea</td>
<td>seattle</td>
<td>2015-03-26T10:58:51</td>
</tr>
<tr>
<td>594</td>
<td>2</td>
<td>2</td>
<td>stephen</td>
<td>nashville</td>
<td>2021-07-23T21:11:48</td>
</tr>
<tr>
<td>594</td>
<td>3</td>
<td>3</td>
<td>selena</td>
<td>bellingham</td>
<td>2018-01-04T07:33:21</td>
</tr>
<tr>
<td>594</td>
<td>4</td>
<td>4</td>
<td>tommy</td>
<td>toronto</td>
<td>1998-09-17T04:03:02</td>
</tr>
<tr>
<td>594</td>
<td>5</td>
<td>5</td>
<td>adam</td>
<td>chicago</td>
<td>2017-04-15T10:07:52</td>
</tr>
<tr>
<td>600</td>
<td>605</td>
<td>6</td>
<td>john</td>
<td>new york</td>
<td>2002-03-13T11:15:14</td>
</tr>
<tr>
<td>605</td>
<td>6</td>
<td>6</td>
<td>john</td>
<td>seattle</td>
<td>2023-03-10T14:07:52</td>
</tr>
</tbody>
</table>

1. Soft DELETE existing tuple
   a. xmax = current txid
2. INSERT new tuple with updated values
   a. xmin = current txid
### Example - **DELETE**

<table>
<thead>
<tr>
<th>xmin</th>
<th>xmax</th>
<th>id</th>
<th>first_name</th>
<th>city</th>
<th>updated_at</th>
</tr>
</thead>
<tbody>
<tr>
<td>594</td>
<td>1</td>
<td>1</td>
<td>chelsea</td>
<td>seattle</td>
<td>2015-03-26T10:58:51</td>
</tr>
<tr>
<td>594</td>
<td>2</td>
<td>2</td>
<td>stephen</td>
<td>nashville</td>
<td>2021-07-23T21:11:48</td>
</tr>
<tr>
<td>594</td>
<td>3</td>
<td>3</td>
<td>selena</td>
<td>bellingham</td>
<td>2018-01-04T07:33:21</td>
</tr>
<tr>
<td>594</td>
<td>4</td>
<td>4</td>
<td>tommy</td>
<td>toronto</td>
<td>1998-09-17T04:03:02</td>
</tr>
<tr>
<td>594</td>
<td>5</td>
<td>5</td>
<td>adam</td>
<td>chicago</td>
<td>2017-04-15T10:07:52</td>
</tr>
<tr>
<td>600</td>
<td>605</td>
<td>6</td>
<td>john</td>
<td>new york</td>
<td>2002-03-13T11:15:14</td>
</tr>
<tr>
<td>605</td>
<td>609</td>
<td>6</td>
<td>john</td>
<td>seattle</td>
<td>2023-03-10T14:07:52</td>
</tr>
</tbody>
</table>

**TUPLE COUNT: 2**
**CURRENT TXID: 609**

1. **Soft DELETE existing tuple**
   a. xmax = current txid
So... infinitely increasing row count forever?
Vacuum

1. ⭐ Deletes dead tuples from Postgres pages, freeing up the space for reuse
2. Updates Postgres internal statistics via `ANALYZE`, improving query planner's effectiveness
3. Updates the "visibility map", which helps vacuum and Index-Only Scan performance
4. Frees up TXIDs for reuse to avoid TXID exhaustion
### Example - VACUUM

<table>
<thead>
<tr>
<th>xmin</th>
<th>xmax</th>
<th>id</th>
<th>first_name</th>
<th>city</th>
<th>updated_at</th>
</tr>
</thead>
<tbody>
<tr>
<td>594</td>
<td></td>
<td>1</td>
<td>chelsea</td>
<td>seattle</td>
<td>2015-03-26T10:58:51</td>
</tr>
<tr>
<td>594</td>
<td></td>
<td>2</td>
<td>stephen</td>
<td>nashville</td>
<td>2021-07-23T21:11:48</td>
</tr>
<tr>
<td>594</td>
<td></td>
<td>3</td>
<td>selena</td>
<td>bellingham</td>
<td>2018-01-04T07:33:21</td>
</tr>
<tr>
<td>594</td>
<td></td>
<td>4</td>
<td>tommy</td>
<td>toronto</td>
<td>1998-09-17T04:03:02</td>
</tr>
<tr>
<td>594</td>
<td></td>
<td>5</td>
<td>adam</td>
<td>chicago</td>
<td>2017-04-15T10:07:52</td>
</tr>
</tbody>
</table>

1. VACUUM hard-deletes dead tuples, freeing up page space for reuse

TUPLE COUNT: 0
CURRENT TXID: 609
Example - **INSERT + SELECT**

<table>
<thead>
<tr>
<th>xmin</th>
<th>xmax</th>
<th>id</th>
<th>first_name</th>
<th>city</th>
<th>updated_at</th>
</tr>
</thead>
<tbody>
<tr>
<td>594</td>
<td></td>
<td>1</td>
<td>chelsea</td>
<td>seattle</td>
<td>2015-03-26T10:58:51</td>
</tr>
<tr>
<td>594</td>
<td></td>
<td>2</td>
<td>stephen</td>
<td>nashville</td>
<td>2021-07-23T21:11:48</td>
</tr>
<tr>
<td>594</td>
<td></td>
<td>3</td>
<td>selena</td>
<td>bellingham</td>
<td>2018-01-04T07:33:21</td>
</tr>
<tr>
<td>594</td>
<td></td>
<td>4</td>
<td>tommy</td>
<td>toronto</td>
<td>1998-09-17T04:03:02</td>
</tr>
<tr>
<td>594</td>
<td></td>
<td>5</td>
<td>adam</td>
<td>chicago</td>
<td>2017-04-15T10:07:52</td>
</tr>
<tr>
<td>611</td>
<td></td>
<td>89</td>
<td>olivia</td>
<td>new york</td>
<td>2023-04-10T17:19:37</td>
</tr>
</tbody>
</table>

1. **TXID=611**: INSERT INTO <table> VALUES (x, y, z);
2. SELECT * FROM <table>;

**SELECT Snapshot**

- **xmin**: 611+  
- **xip_list**: [611]
Postgres disk usage

➔ Vacuum: “frees up space for reuse”
➔ Without explicit intervention*, Postgres disk usage only increases
  ◆ Pages are only created, not deleted
  ◆ Vacuum deletes tuples, not pages
➔ Exceptions:
  ◆ Page truncation, but VERY rare

* (we’ll get to this later)
2. Table Bloat
<table>
<thead>
<tr>
<th>Table Bloat</th>
</tr>
</thead>
</table>

Less-than-optimal “page density”

(number of live tuples per page vs how many could hypothetically fit)
Example

vs
Why is bloat often problematic?

➔ With dead tuples occupying what should be allocate-able disk space for new tuples, Postgres continues to create new pages
  ◆ Unnecessarily increases disk usage

➔ After vacuum runs and dead tuples are deleted, live tuples are stored sparsely over many pages
  ◆ More I/O usage during scans (more pages per scan)
Why is bloat often problematic?

Things are problematic... when they create problems 😱🧠

➔ Problems:
  ◆ Bad read latency
  ◆ High (expensive?) disk usage
  ◆ High (expensive?) IOPS

➔ Bloat == the root cause of other problems, not necessarily a problem in itself
How does bloat occur?

1. **UPDATE/DELETE**-heavy workloads
   a. Bloat is caused by pages becoming saturated with dead tuples, generated by updates and deletes
   b. Example:
      i. User activity resulting in cascading updates/deletes
      ii. Scheduled batch jobs editing massive amounts of data

2. **Badly-tuned autovacuum configuration**
   a. Overly conservative (or older default) autovacuum configurations paired with high **UPDATE/DELETE** workload mean autovacuum can’t catch up
Example Case Study

ML Feature Store

- 100s/1000s features/user
- Table size: 300GB
- All writes = upserts
- Burst-based, high volume write traffic triggered by user activity
- Feature deprecation → cron-based job to remove old values
- Default autovacuum configs
## Example Case Study

### ML Feature Store

- **100s/1000s features/user**
- **Table size: 300GB**
- **All writes = upserts**
- **Burst-based, high volume write traffic** triggered by user activity
- **Feature deprecation** → cron-based job to remove old values
- **Default autovacuum configs**

| id | feature_name  | user_id (bigint) | value (JSONB) | ...
|----|---------------|------------------|---------------|------
| 1  | last_login    | 61466            | {...}         |      |
| 2  | likes_cats    | 9953217          | true          |      |
| 3  | owns_house    | 33644221         | false         |      |
| 4  | svd_vector    | 37995002         | [...]         |      |
| ...| ...           | ...              | {...}         | ...  |
3. Quantifying, Mitigating, & Avoiding Bloat
Quantifying table bloat

1. **pgstattuple**
   a. Postgres contrib module created specifically for quantifying table bloat
   b. Precise return value, but can be very expensive. Slow-running, high resource usage
   c. $O(n)$ runtime based on table size

2. **Estimation queries**
   a. Open-source estimation queries leveraging `pg_class.reltuples`
   b. Run `ANALYZE` first
   c. $O(1)$ runtime, but results are only estimates
<table>
<thead>
<tr>
<th>Descriptor</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>table_len</td>
<td>81584128</td>
</tr>
<tr>
<td>tuple_count</td>
<td>108963</td>
</tr>
<tr>
<td>tuple_len</td>
<td>73811880</td>
</tr>
<tr>
<td>tuple_percent</td>
<td>90.47</td>
</tr>
<tr>
<td>dead_tuple_count</td>
<td>2517</td>
</tr>
<tr>
<td>dead_tuple_len</td>
<td>2006536</td>
</tr>
<tr>
<td>dead_tuple_percent</td>
<td>2.46</td>
</tr>
<tr>
<td>free_space</td>
<td>5017928</td>
</tr>
<tr>
<td>free_percent</td>
<td>6.15</td>
</tr>
</tbody>
</table>

- `table_len`: Table length (bytes)
- `tuple_count`: # of total live tuples
- `tuple_percent`: % of total tuples which are live
- `dead_tuple_count`: % of total tuples which are dead
- `free_space`: % of total tuples which are dead
<table>
<thead>
<tr>
<th></th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>table_len</td>
<td>81584128</td>
</tr>
<tr>
<td>tuple_count</td>
<td>108963</td>
</tr>
<tr>
<td>tuple_len</td>
<td>73811880</td>
</tr>
<tr>
<td>tuple_percent</td>
<td>90.47</td>
</tr>
<tr>
<td>dead_tuple_count</td>
<td>2517</td>
</tr>
<tr>
<td>dead_tuple_len</td>
<td>2006536</td>
</tr>
<tr>
<td>dead_tuple_percent</td>
<td>2.46</td>
</tr>
<tr>
<td>free_space</td>
<td>5017928</td>
</tr>
<tr>
<td>free_percent</td>
<td>6.15</td>
</tr>
</tbody>
</table>
```
db=> ANALYZE VERBOSE;

db=> <really long bloat estimation query>;

- [ RECORD 1 ]----------------------------------------

<table>
<thead>
<tr>
<th>real_size</th>
<th>81723392</th>
<th>estimated table length (bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>bloat_size</td>
<td>7700480</td>
<td>estimated size of bloat (bytes)</td>
</tr>
<tr>
<td>bloat_pct</td>
<td>9.422614274258219</td>
<td>estimated % of real_size used by bloat</td>
</tr>
</tbody>
</table>

https://github.com/ioguix/pgsql-bloat-estimation/tree/master
```
db=> ANALYZE VERBOSE;

db=> <really long bloat estimation query>;

| RECORD 1 |-------------------|
| real_size | 81723392          |
| bloat_size | 7700480          |
| bloat_pct | 9.422614274258219 |

--- estimated table length (bytes)
--- estimated size of bloat (bytes)
--- estimated % of real_size used by bloat

https://github.com/ioguix/pgsql-bloat-estimation/tree/master
Comparing methods

% dead tuple count (pgstattrace) vs % dead disk space (estimation)

Not directly comparable

- Tuple size varies wildly
- Page-level opportunistic pruning leaves 4-byte “tombstones”
- 1KB “dead page space”: 250 4-byte tombstones, or 10 100-byte tuples?

More info: Bloat in PostgreSQL: A Taxonomy (Peter Geoghegan)
Interpreting results: How much bloat is “too much”?
Interpreting results: How much bloat is “too much”? it depends
Interpreting results: How much bloat is “too much”?

1. **Very Small (<= 1GB):**
   a. Up to ~70% bloat is acceptable
   b. This is high and not ideal, but at this table size, bloat has an imperceptible impact on performance.

2. **Small - Medium (~1-30GB):**
   a. Up to ~25% dead tuples is acceptable

3. **Large (~30-100GB):**
   a. Up to ~20% dead tuples is acceptable

4. **Very Large (~100GB+):**
   a. Up to ~18% dead tuples is acceptable
Dealing with bloated tables

1. Configure autovacuum to be more aggressive
2. Repack or rebuild tables
1. Configure autovacuum aggressively

- `autovacuum_vacuum_scale_factor`
  - Default: 0.2 (20% of table size)
  - “At least x% of the table must have changed since last vacuum for autovacuum to run”
  - Smaller → more frequent triggering of vacuums
  - EX: `autovacuum_vacuum_scale_factor = 0.01`
    - 1% of table size

- `autovacuum_vacuum_threshold`
  - Default: 50
  - Can be used to set raw value for vacuum trigger:
    - `autovacuum_vacuum_scale_factor = 0`
    - `autovacuum_vacuum_threshold = 200000`

Typically tune per-table via `ALTER TABLE`, not server-wide.
1. Configure autovacuum aggressively

- `autovacuum_vacuum_cost_delay`
  - Default: 2ms (20ms PG11 and before)
  - Cost delay/wait time used in autovacuum operations
  - If using modern hardware, 2ms should be used regardless of PG version

- `autovacuum_max_workers`
  - Default: 3 (server-wide)
  - If you have many tables (1000s+) on your database server
  - Check `pg_stat_progress_vacuum` to see how many vacuums are currently running. Increase +1 if always at max.
2. Repack or rebuild tables

VACUUM FULL

Rewrites table and all indexes into a new disk file with no extra space

➔ Lock: ACCESS EXCLUSIVE (blocks reads & writes)
➔ “Wasted space” returned to the operating system.
➔ Not recommended due to extremely heavy lock
2. Repack or rebuild tables

pg_repack (+ pg_squeeze, etc)

Duplicates the bloated table, copies over incoming data via triggers – then ALTERs the table names to switch them, dropping the old table

➔ Lock: ACCESS SHARE
➔ Requires 2x current table size in disk, significant CPU/RAM
➔ Occasionally flaky
  ◆ Failure scenario: incomplete tables in pg_repack schema must be manually DROP-ped. No data loss, downtime.
➔ Overall recommended for use!
pg_repack (+ pg_squeeze, etc)

pg_repack

```
db=> CREATE EXTENSION pg_repack;
```

```
$ /usr/.../pg_repack -h <HOST> -U <USER> -d <DATABASE> -t <SCHEMA>.<TABLE>
```

→ External binary, less invasive
→ Supported in most managed Postgres services (EX: AWS RDS)

pg_squeeze

```
db=> CREATE EXTENSION pg_squeeze;
```

```
db=> SELECT squeeze.squeeze_table(...);
```

→ Operates entirely within the database, no external binary
→ Background worker to schedule rewrites
4. Designing bloat-aware data access patterns
Data Access Patterns

➔ How, when, and for what purpose are you writing & reading data?
  ◆ What % of transactions are reads, vs insert/update/deletes?

➔ Roughly what % data growth do you expect to occur annually?

➔ What sort of access will you/won’t you support?
  ◆ What is your process for enforcing this?
Data Access Patterns

➔ How, when, and for what purpose are you writing & reading data?
  ◆ What % of transactions are reads, vs insert/update/deletes?

➔ Roughly what % data growth do you expect to occur annually?

➔ What sort of access will you/won’t you support?
  ◆ What is your process for enforcing this?
If your app is **UPDATE/DELETE** heavy...

Can you redesign your data access patterns to have fewer updates/deletes?

➔ **EX:** User actions trigger a "burst" of updates on a single row.
  ◆ Can you update each row once instead of \( n \) times?

➔ **EX:** You’re updating the same row (\( \text{last_seen} \)) 5x/second.
  ◆ Can you have an append-only log style table with just inserts, index on \( (\text{user_id}, \text{inserted_at}) \), and query for the most recent row?
If you have regular large **DELETE** jobs...

➔ Is your dataset compatible with partitioning, meaning you can replace **DELETE** with **DETACH PARTITION/pg_partman**?
  ◆ Range or hash partitioning
  ◆ Always able to provide partition key for user queries?

➔ Are you making sure to always use a reasonable batch size in your **DELETE**s, rather than just running in one huge transaction?

➔ Instead of 1 large weekly **DELETE** job, can you run 7 smaller daily **DELETE** jobs, and configure autovacuum to trigger per job?
Are you reinventing any wheels?

**My rule of thumb:** using Postgres for things outside of Postgres’ intended OLTP purpose is fine (often via community-supported extensions) up to a certain scale.

→ **Full Text Search (FTS)**
   ◆ 25GB data → Postgres
   ◆ 100GB data → Elasticsearch

→ **Key/Value Store**
   ◆ 50GB K/V table, 80% traffic == reads → Postgres
   ◆ 120GB K/V table, 80% traffic == writes → Redis
Thank you!

Chelsea Dole

cdole@brex.com
https://www.linkedin.com/in/chelsea-dole/