NEON
Serverless Postgres

Shaving off bytes at any scale

Space savings in various subsystems of PostgreSQL

Matthias van de Meent

Matthias van de Meent

- Frontend Dev -> Full-stack ('17) -> Infra/DBA ('18) -> PG hacker ('20)
- Employed full-time on PG-related job since 2021

Overview

1: Catalogs

2: Table storage

3: WAL

4: TOAST

1: Catalogs

> pg_node_tree

> pg_attribute, pg_type information duplication

> aggressive toast_tuple_target

1: Catalogs > pg_node_tree

Used any time we need to store expressions

- Serialized nodes: Storage & debugging
- Extremely verbose
- Not even a very useful output format:
  - → Based on our own misinterpretation of Lisp syntax
  - → Quite difficult to parse, process

1: Catalogs > pg_node_tree

Used any time we need to store expressions

- Replace storage with binary format (WIP)
  - → Create improved node read/write infrastructure
  - → Add binary read-write format, omit default values.

  Initial tests show a 50+% reduction in storage
- Using said infrastructure, create a JSON writer for Node*/pg_node_tree (future)
  - → Improved debugging experience of catalog data

1: Catalogs > pg_attribute

Stores attributes (duh!) of relations

- (HEAD) min size of 104 bytes, each
- Name
  - → NAMEDATALEN bytes, = 64B
- Duplicated data from pg_type
  - → attlen, attbyval, attalign
- Boolean flag bytes x5
  - → single flags field?

1: Catalogs > aggressive toast_tuple_target

How many TOAST table accesses do we want in our catalog?

2: Table storage

> Visibility information

> Physical column order

> Columnar, compression

> Index data

2: Table storage > Visibility information

Every live tuple must be updatable

- 18 bytes (+ 15 infomask bits) on visibility info:
  - → t_xmin/t_xmax/t_cid/t_ctid
  - → Kind of wasteful for all-visible frozen tuples
- Put all that visibility info in a separate fork, e.g.
  - → specialized btree ordered by ctid
  - → drop visibility info for frozen tuples
  - → efficient VACUUM scans
-

2: Table storage > Physical column order

```
/* Column Tetris */ CREATE TABLE (
  c1 bool, c2 bigint, ...
);
```

- Alignment padding can be expensive
- Reorder columns in table creation, new column creation
  → Logical vs physical order
- ALTER TABLE ... ADD COLUMN support?
  → use HEAP_NATTS as layout version

2: Table storage > Columnar, compression

Some data is more equal than other data

- Some data can be very compressible
  - → Time series data, orderlines, ...
- Various compression schemes make sense
  - → Even *MySQL* has (optional) page-level compression.
- Also applies to indexes (or, *especially* to indexes)

2: Table storage > Index size

- Data in indexes is often co-located with similar data
  → btree, gist, ...
- BTree prefix compression
- BRIN range bound suffix truncation

3: WAL

> Record overhead

> Compression scope

3: WAL > Record overhead

Empty WAL record

3: WAL > Record overhead

Empty WAL record: 24 bytes

- xl_tot_len: 4 B
- xl_xid: 4 B
- xl_prev: 8 B
- xl_info: 1 B
- xl_rmid: 1 B
- <padding: 2 B>
- xl_crc: 4 B

3: WAL > Record overhead

Empty WAL record: 24 bytes

- xl_prev, xl_info, xl_rmid, xl_crc: 14 B
  - → No comments
- xl_tot_len: 4 B
  - → value essentially always $< 2^{16}$
- xl_xid: 4 B
  - → but no index AM uses this, so…
  - → No comments
- <padding: 2 B>
  - → *Huh?*

3: WAL > Record overhead

Empty WAL record: 24 bytes
Modify a single data page?

3: WAL > Record overhead

Empty WAL record: 24 bytes
Modify a single data page? 44 bytes

- WAL record header (24 B)
- blkid (1B)
- fork+flags (1B)
- length (2B)
- RelFileLocator (12 B)
- BlockNo (4B)

3: WAL > Record overhead

Empty WAL record: 24 bytes
Modify a single data page? 44 bytes

- blkid (1B), fork+flags (1B)
- length (2B)
  - → regularly 0/empty
- RelFileLocator (12 B)
  - → 3x OID, can be anything
  - → … but often small
- BlockNo (4B)
  - → varint coding for smaller tables?

3: WAL > Compression scope

FPIs are not the only compressible data in WAL records

- Records with multiple FPIs
  - → e.g. GIN bulk creation
- Compress full WAL record data
  - → smaller total WAL, but higher CPU overhead...

4: TOAST

> Compression

> Updates

4: TOAST > Compression

From intra-value to inter-value

- Compression dictionaries
  - → Analysis of existing dataset, or hand-crafted dictionaries.
  - → Can't be dropped without full table scan, or versioning horizon approach
- Datatype-aware compression
  - → int[] -> differential encoding; etc.
  - → CREATE TYPE hooks...

4: TOAST > Updates

UPDATE tab SET col_200MB_bytea =
col_200MB_bytea || '\x00'::bytea;

- WAL volume is huge

  OID churn is huge

  TOAST table bloat is huge

- ... why not have a specific API for bytea 'append'

  operations? Or jsonb 'update' operations? Or ...

Thank you!

@mmeent_pg