Zheap: The Next Generation Storage Engine for Postgres

Ken Rugg, EnterpriseDB
May 29, 2019
The world leader in Enterprise Postgres software and services

PROVEN
- Recognized RDBMS leader by Gartner
- 2013-2018 Member of Gartner Magic Quadrant

COMMITTED
- Founded in 2004
- Largest PostgreSQL contributor—40% of core team

GLOBAL
- Customer global base > 4000
- 300+ Employees world-wide
- Offices in 16 countries
ZHEAP: WHAT IS IT?

• New storage format being developed by EnterpriseDB
• Work in progress is already released on github under PostgreSQL license
• Basics work, but much remains to be done
• Goal is to get it integrated into PostgreSQL
BLOAT: MOTIVATION AND DEFINITION

• Original motivation for zheap: In some workloads, PostgreSQL tables tend to bloat, and when they do, it’s hard to get rid of the bloat.

• Bloat occurs when the table and indexes grow even though the amount of real data being stored has not increased.

• Bloat is caused mainly by updates, because we must keep both the old and new row versions for a period of time.

• Bloat can be a concern because of increased disk consumption, but typically a bigger concern is performance loss – if a table is twice as big as it “should be”, scanning it takes twice as long.
BLOAT: WHY A NEW STORAGE FORMAT?

• All systems that use MVCC must deal with multiple row versions, but they store them in different places.
  • PostgreSQL and Firebird put all row versions in the table.
  • Oracle and MySQL put old row versions in the undo log.
  • SQL Server puts old row versions in tempdb.

• Leaving old row versions in the table makes cleanup harder – sometimes need to use CLUSTER or VACUUM FULL.

• Improving VACUUM helps contain bloat, but can’t prevent it completely.
ZHEAP: HIGH-LEVEL GOALS

• Better Bloat Control
  • Perform updates “in place” to avoid creating bloat (when possible)
  • Reuse space right after COMMIT or ABORT – little or no need for VACUUM

• Fewer Writes
  • Eliminate hint-bits, freezing and anything else that could dirty a page (other than an update)
  • Allowing in-place updates when index column is updated by providing delete-marking in index
  • Indexes are not touched if the indexed columns are not changed

• Smaller in Size
  • Narrower tuple headers – most transactional info not stored on the page itself
  • Eliminate most alignment padding
ZHEAP: TRANSACTION SLOTS

• Each zheap page has fixed set of transaction slots containing transaction info (transaction id, epoch and the latest undo record pointer of that transaction).

• As of now, the number of slots are configurable and default value of same is four.

• Each transaction slot occupies 16 bytes.

• We allow the transaction slots to be reused after the transaction becomes too old to matter (older than oldest xid having undo), committed or rolled back. This allows us to operate without having too many slots.
ZHEAP: PAGE FORMAT
### ZHEAP: TUPLE FORMAT

<table>
<thead>
<tr>
<th><strong>Field</strong></th>
<th><strong>Description</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Xmin</td>
<td>- inserting transaction id</td>
</tr>
<tr>
<td>Xmax</td>
<td>- deleting transaction id</td>
</tr>
<tr>
<td>t_cid</td>
<td>- inserting or deleting command id, or both</td>
</tr>
<tr>
<td>t_ctid</td>
<td>- tuple id (page/item)</td>
</tr>
<tr>
<td>infomask2</td>
<td>- number of attrs and flags</td>
</tr>
<tr>
<td>infomask</td>
<td>- tuple flags</td>
</tr>
<tr>
<td>hoff</td>
<td>- length of tuple header incl. bitmaps</td>
</tr>
<tr>
<td>bits</td>
<td>- bitmap representing NULLs</td>
</tr>
<tr>
<td>OID</td>
<td>- object id of tuple (optional)</td>
</tr>
</tbody>
</table>

**Tuple Header**

- infomask2 – number of attrs and transaction slot id
- infomask – tuple flags
- hoff – length of tuple header incl. bitmaps
- bits – bitmap representing NULLs
- OID – object id of tuple (optional)

**Attributes**

- heap Tuple
- zheap Tuple
ZHEAP: HOW DOES IT WORK?

• Whenever a transaction performs an operation on a tuple, the operation is also recorded in an undo log.

• If the transaction aborts, the undo log can be used to reverse all of the operations performed by the transaction.

• The undo log also contains most of the data we need for MVCC purposes, so little transactional data needs to be stored in the table itself. This, and a reduction in alignment padding, mean that zheap is smaller on disk.

• We avoid dirtying the page except when the data has been modified, or after an abort. *Long-term goal: No VACUUM or other routine maintenance of any kind.*
PURPOSE OF UNDO

- Undo is responsible for reversing the effects of aborted transactions.
- When a transaction performs an operation, it also writes it to the write-ahead log (REDO) and records the information needed to reverse it (UNDO). If the transaction aborts, UNDO is used to reverse all changes made by the transaction.
- It stores old versions of rows required for MVCC.
- Independent of avoiding bloat, having undo can provide systematic framework for cleaning.
  - For example, if a transaction creates a table and, while that transaction is still in progress, there is an operating system or PostgreSQL crash, the storage used by the table is permanently leaked. This could be fixed by undo.
ZHEAP: BASIC OPERATIONS

• Requires an UNDO record for each INSERT, DELETE or UPDATE

• INSERT – Same as current heap + undo
  • Insert the new row
  • Emit an UNDO to remove it

• Delete - Same as current heap + undo
  • Emit an UNDO to put back the row

• Update – Very different than current heap
  • And slightly more complicated…
ZHEAP: BASIC OPERATIONS

• Update can be done in 2 ways
  • In place update – *whenever possible!*
    • Old row placed in UNDO
    • New row replaces it in heap
  • When in-place update is not possible…
    • Perform DELETE combined with INSERT
    • Emit an UNDO to reverse both the operations

• In place updates can **not** be done when…
  • New row is bigger than old row and can't fit in same page
  • Index column is updated

• In place vs HOT
  • HOT requires 1) space on the page for the new row and 2) no index changes
UNDO CHAINS AND VISIBILITY

• The undo chain is formed at page level for each transaction

• Each undo record header contains the location of previous undo record of the transaction on the same page

• When the current tuple is not visible to the scan snapshot, we can traverse undo chain to find the version which is visible (if any).
**UNDO CHAINS**

- Insert (1, 'abc')
- Update tbl set c2=true where c1=1
- Update tbl set c2='abc' where c1=100
- Update tbl set c2='ghi' where c1=1
PERFORMANCE DATA (TEST SETUP)

• Size and TPS comparison of heap and zheap

• We have used pgbench
  ➢ Initialize the data (at scale factor 1000)
  ➢ Simple-update test (which comprises of one-update, one-select, one-insert)

• Machine details:
  • x86_64 architecture, 2-sockets, 14-cores per socket, 2-threads per-core, 64-GB RAM

• Non-default parameters:
  • shared_buffers=32GB, min_wal_size=15GB, max_wal_size=20GB, checkpoint_timeout=1200, maintenance_work_mem=1GB, checkpoint_completion_target=0.9, synchronous_commit = off;
The Initial size of accounts table is 13GB in heap and 11GB in zheap.

The size in heap grows to 19GB at 8-client count test and to 26GB at 64-client count test.

The size in zheap remains at 11GB for both the client-counts at the end of test.

All the undo generated during test gets discarded within a few seconds after the open transaction is ended.

The TPS of zheap is ~40% higher than heap in above tests at 8 client-count.
BENEFITS

• Because zheap is smaller on-disk, we get a small performance boost.
• No worries about VACUUM kicking in unexpectedly.
• Undo bloat is self-healing – good for cloud or other “unattended” workloads.
• In workloads where the heap bloats and zheap only bloats the undo, we get a massive performance boost
• Discarding undo happens in the background and is cheaper than HOT pruning; that helps, too!
DRAWBACKS

• Transaction abort will be more expensive.
• Deletes might not perform as well.
• Could be slow if most/all indexed columns are updated at the same time.
• On small tables, contention for transaction slots can limit performance.
• Huge amount of development work.
PLUGGABLE STORAGE: PLAN

• Allow PostgreSQL to support pluggable storage formats.

• Allows innovation – major changes to the heap are impossible because everyone relies on it. Can’t go backwards for any use case!

• Allows for user choice – if there are multiple storage formats available, pick the one that is best for your use case.
PLUGGABLE STORAGE: EXAMPLES

• **Columnar storage**
  • Most queries don’t need all columns

• **Write-once read-many (WORM)**
  • No support UPDATE, DELETE, or SELECT FOR UPDATE/SHARE

• **Index-organized storage**
  • One index is more important than all of the others

• **In-memory storage**
  • No need to spill to disk

• **Non-transactional storage**
  • No MVCC
WHEN?

- Pluggable Storage in PostgreSQL 12
- Undo in PostgreSQL 12
- First version of zheap targeted for PostgreSQL 13
- There will still be much more to do for “v2”
WHO?

zheap & undo
- Amit Kapila (development lead)
- Dilip Kumar
- Kuntal Ghosh
- Mithun CY
- Ashutosh Sharma
- Rafia Sabih
- Beena Emerson
- Amit Khandekar
- Thomas Munro
- Neha Sharma (QA)

Pluggable Storage
- Haribabu Kommi (Fujitsu)
- Alexander Korotkov (Postgres Pro)
- Andres Freund
- Amit Khandekar
QUESTIONS & DISCUSSION
KEYNOTE SPEAKER

World Wide Web Inventor, Sir Tim Berners-Lee to Deliver Keynote at Postgres Vision 2019

LEARN MORE
THANK YOU

info@enterprisedb.com
www.enterprisedb.com