zheap: an answer to postgresql bloat woes

- Amit Kapila | PGConf.EU 2018

Contents

- Why zheap?
- Purpose of undo
- Zheap
- TPD (Extended transaction data)
- Undo
- Indexing in zheap
- Performance data
- Benefits and Drawbacks

Bloat: Definition

- 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 updates 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 and there is an additional cost to remove bloat.
- Transaction-id wraparound can cause emergency freezing which could be very costly.

Purpose of undo

- The old versions of rows required for MVCC are stored in 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.
- 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.

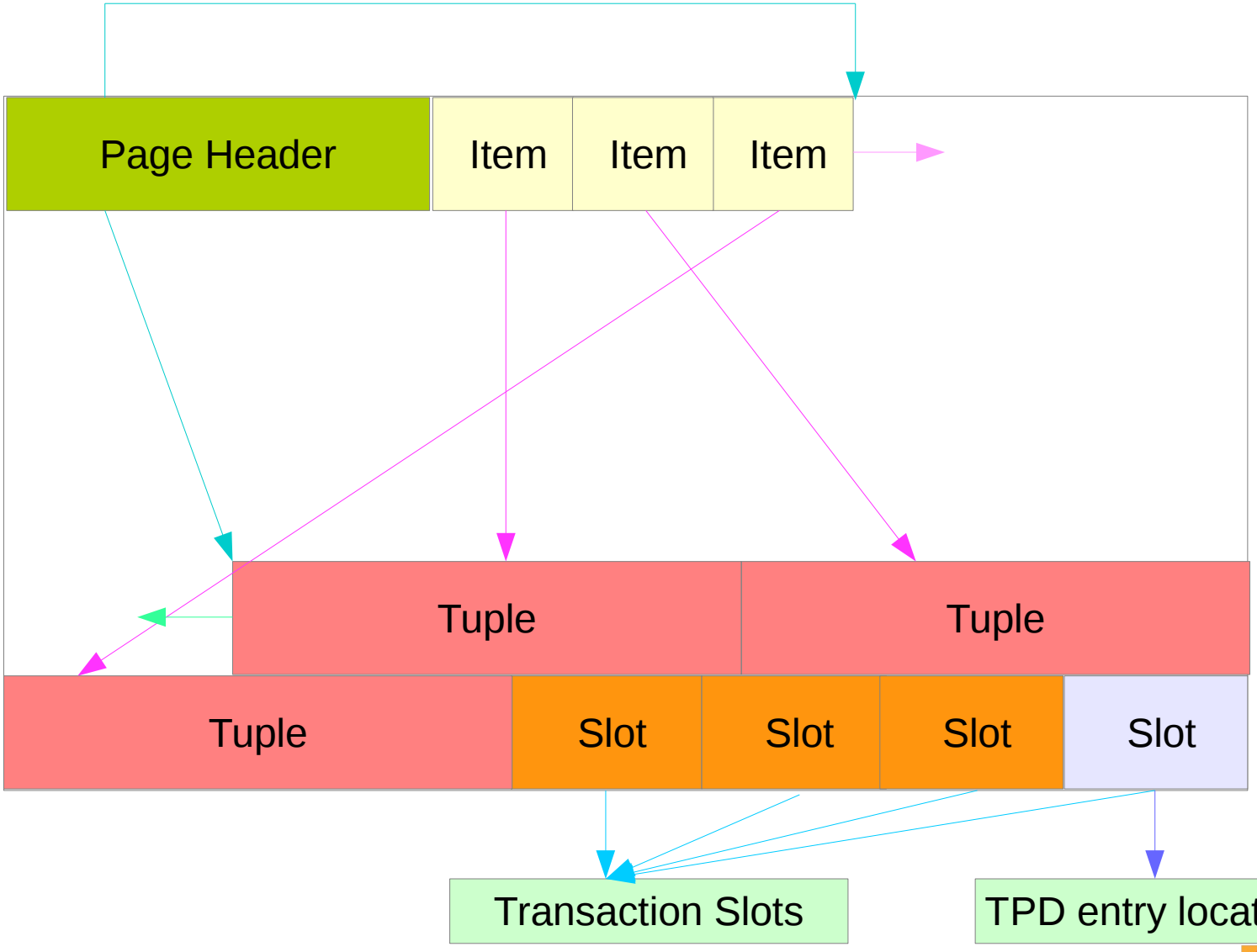
zheap: High-level goals

- 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 a data modification**.
  - 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. This will also help in containing the bloat.
- Smaller in size
  - Narrower tuple headers – most transactional information not stored within the page itself.
  - Eliminate most alignment padding.

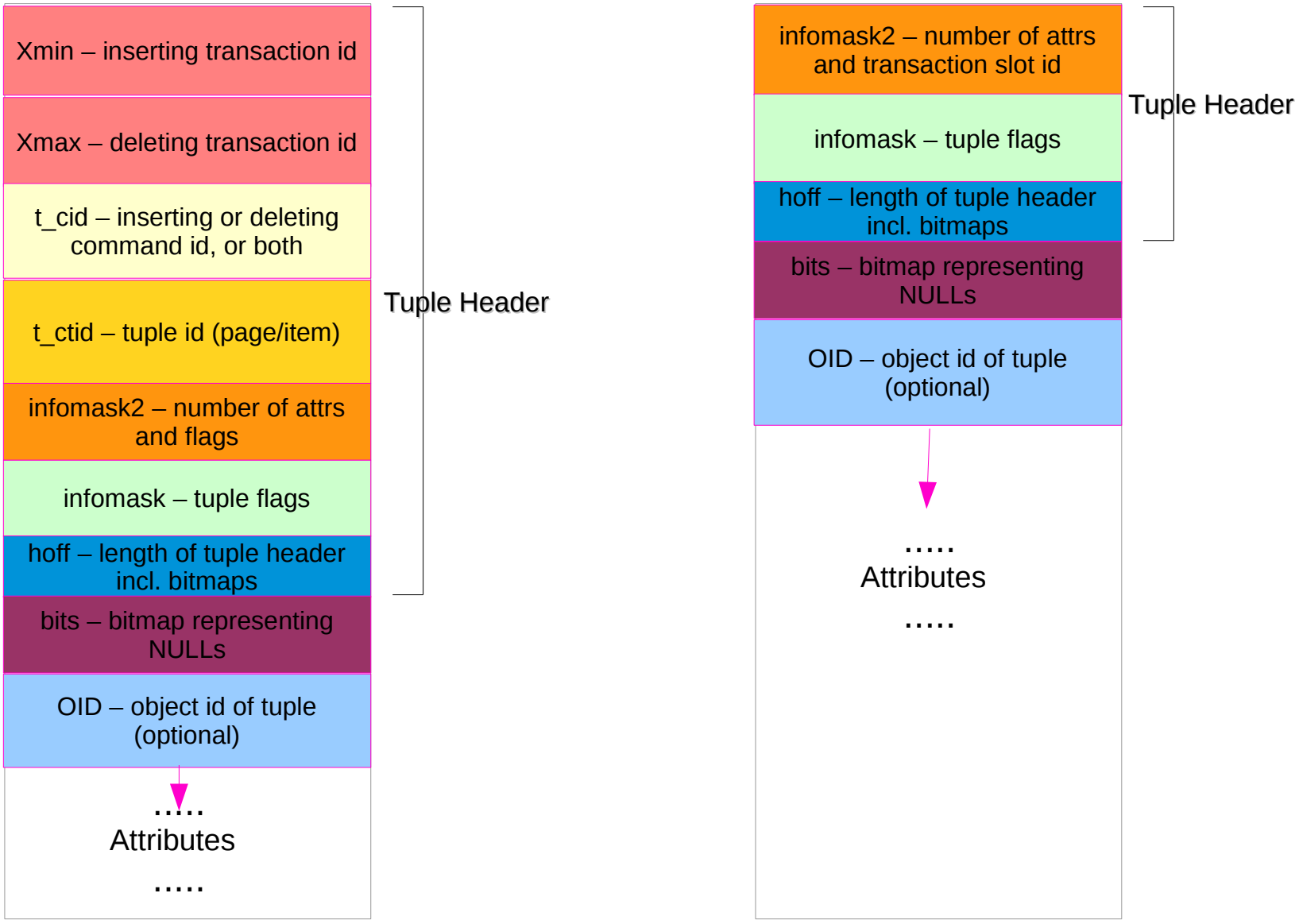
zheap: Page Format

- 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.
- Tuples are placed in an itemid order to allow faster scans.

zheap: Page Format



zheap: Tuple Format



zheap: Operations

- In this new heap, we need to emit an UNDO record for each of the Insert, Delete and Update operations.
- For an INSERT, we will insert the new write and emit UNDO which will remove it.
- For Delete, we will emit an UNDO which will put back the row.

zheap: Operations

- Update can be done in two ways:
  - An in-place update in which old row can be maintained in UNDO and new row in heap.
  - Cases where in-place update is not possible like
    - when the new row is bigger than old row and can't fit in same page, we need to perform Delete combined with Insert and emit an UNDO to reverse both the operations.
    - Index column is updated.
- First strategy is preferable as that doesn't bloat the heap, but I think we can't avoid to have non-in-place updates in some cases.

zheap: Space Reclaim

- Space can be reclaimed for
  - deletes
  - Non-in-place updates
  - updates that update to a smaller value.
- We can reuse the space when the transaction that has performed the operation is committed unlike heap where we need to wait till the deleted tuple becomes all-visible.
- We can immediately reclaim the space for inserts that are rolled-back.

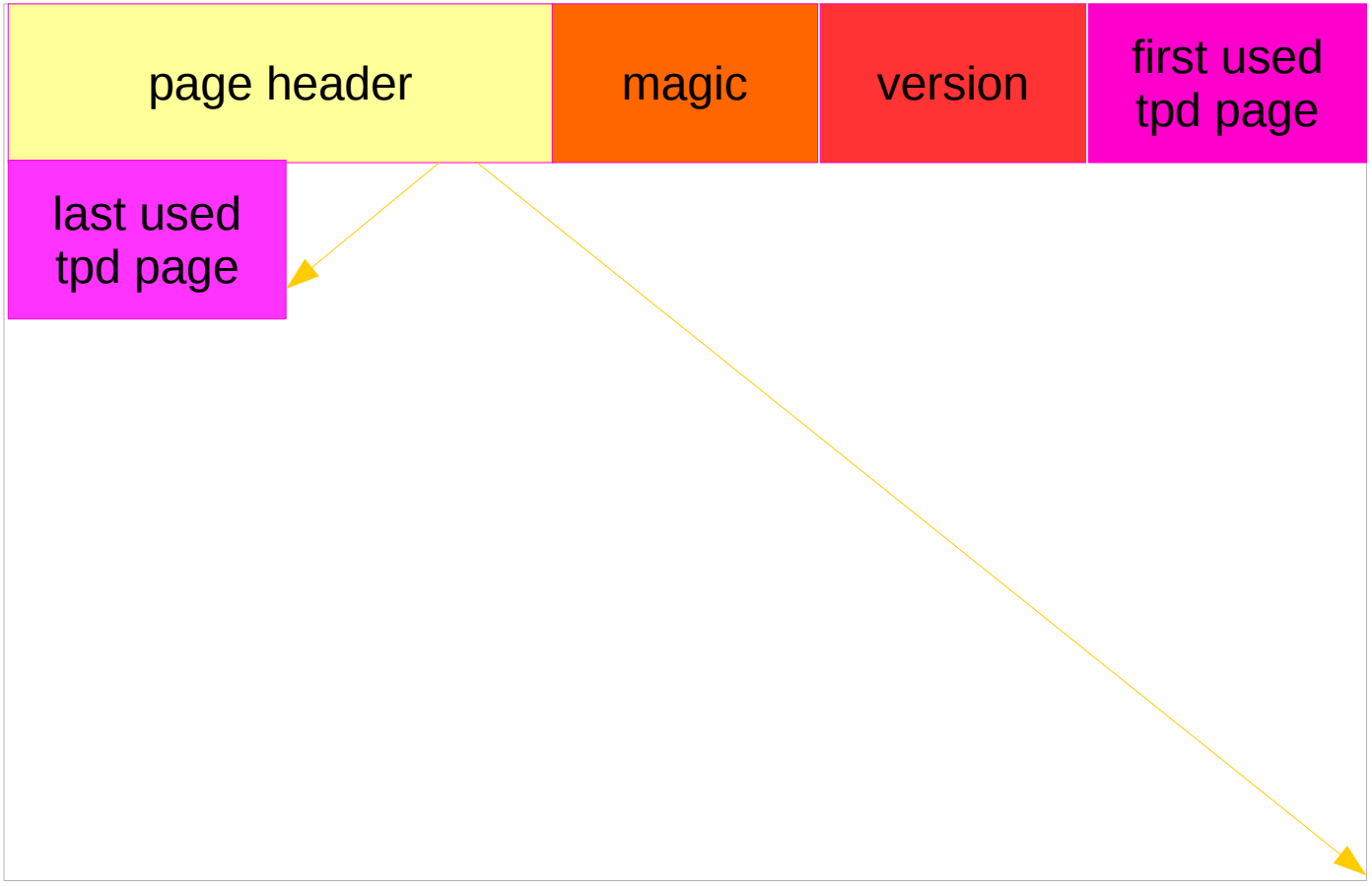
Undo chains and visibility

- The undo chain is formed at page level for each transaction.
- 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).

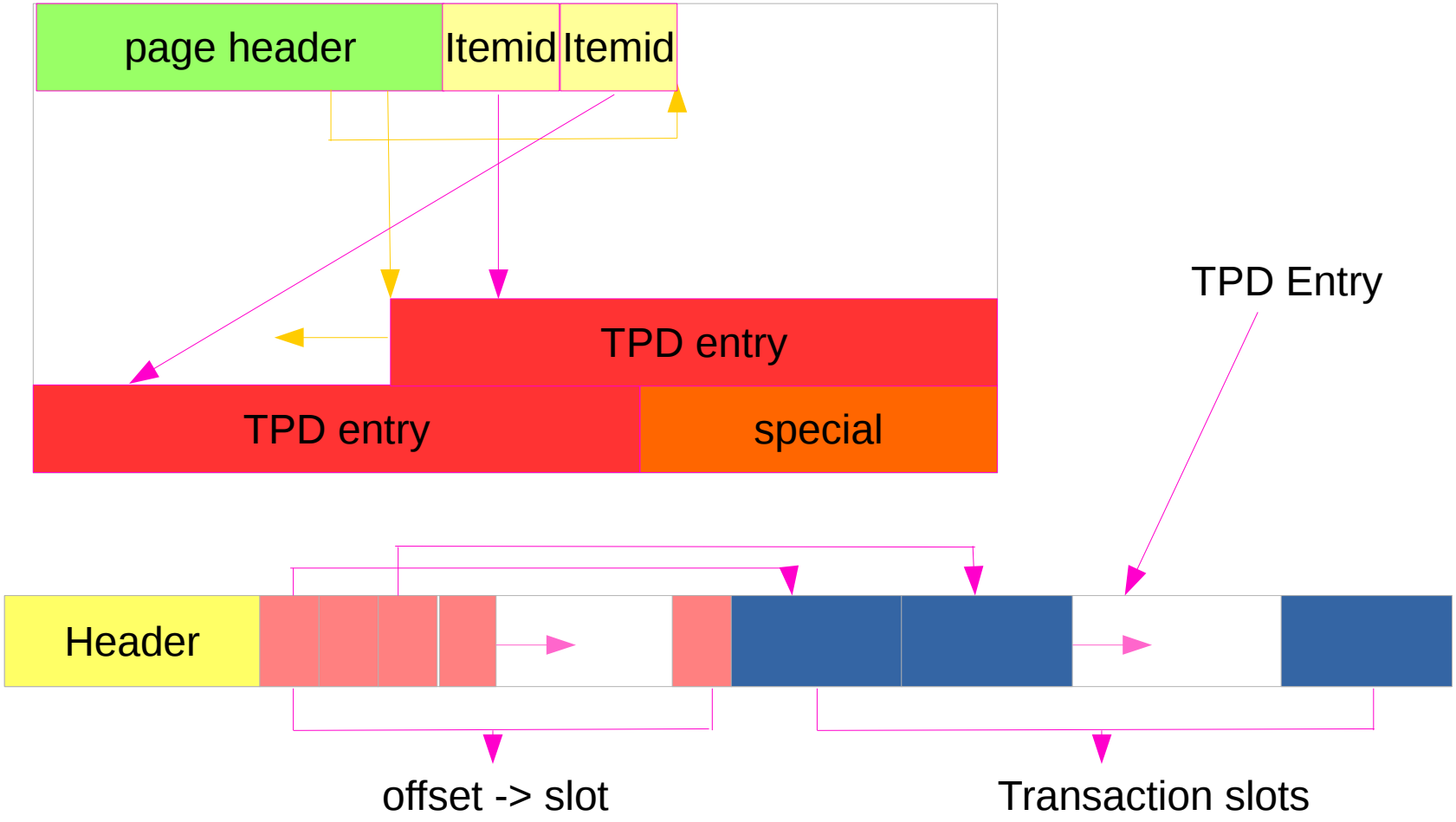
TPD

- TPD is nothing but temporary data page consisting of extended transaction slots from heap pages.
- Why we need TPD?
  - In the heap page we have fixed number of transaction slots which can lead to deadlock.
  - To support cases where a large number of transactions acquire SHARE or KEY SHARE locks on a single page.
- The TPD overflow pages will be stored in the zheap itself, interleaved with regular pages.
- We have a meta page in zheap from which all overflow pages are tracked.
- The idea of putting TPD in heap was of **Andres Freund**

Metapage



TPD page



- TPD Entry acts like an extension of the transaction slot array in heap page.
- Tuple headers normally point to the transaction slot responsible for the last modification, but since there aren't enough bits available to do this in the case where a TPD is used, an offset -> slot mapping is stored in the TPD entry itself.

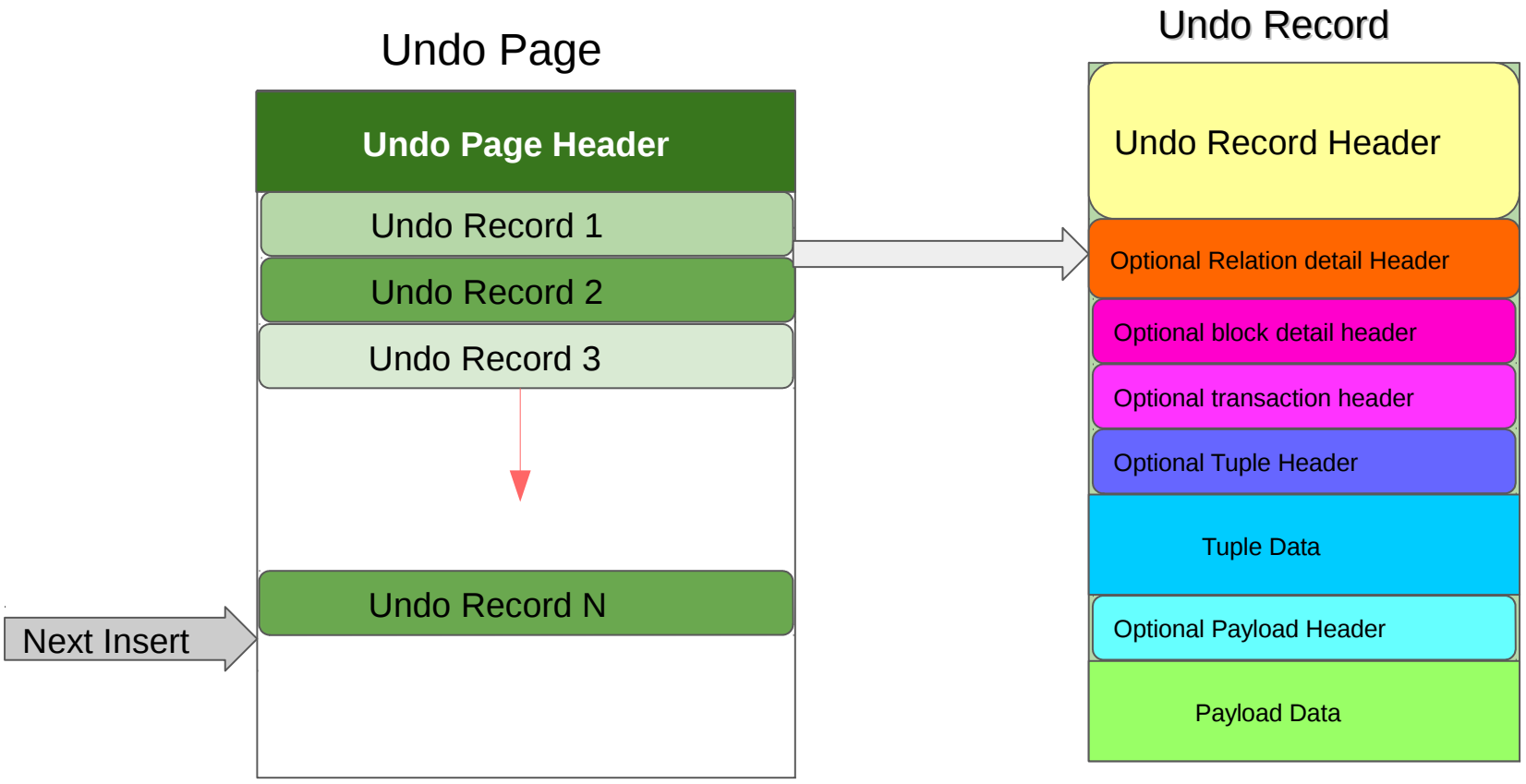
UNDO storage

- Single file per backend
  - It allows efficient use of space.
  - Cleanup can be performed efficiently by keeping the tail and head (insertion) pointer and keep the tail pointer moving and once it reaches insertion point, reclaim the space for file.
  - Once the backend exits, the same file can be used by another backend.

Undo

- undo_tablespaces
  - This variable specifies the tablespaces in which to store undo data.
  - The value is a list of names of tablespaces. When there is more than one name in list, we choose an arbitrary one.
  - The default value is an empty string, which results in all temporary objects being created in the default tablespace.
  - The variable can only be changed before the first statement is executed in a transaction.
- This allows undo data to be stored in separate space than the actual data.

UNDO page format



WAL considerations for undo data

- One important consideration is that we don't need to have full page images for data in undo logs (except when data checksums are enabled) as the undo logs are always written serially, so there shouldn't be any torn page issue.
- Unlike heap,
  - we don't need to rely on the existing state of page to perform operation in the undo logs.
  - undo logs doesn't have any operations that move data, like heap page compaction/pruning.

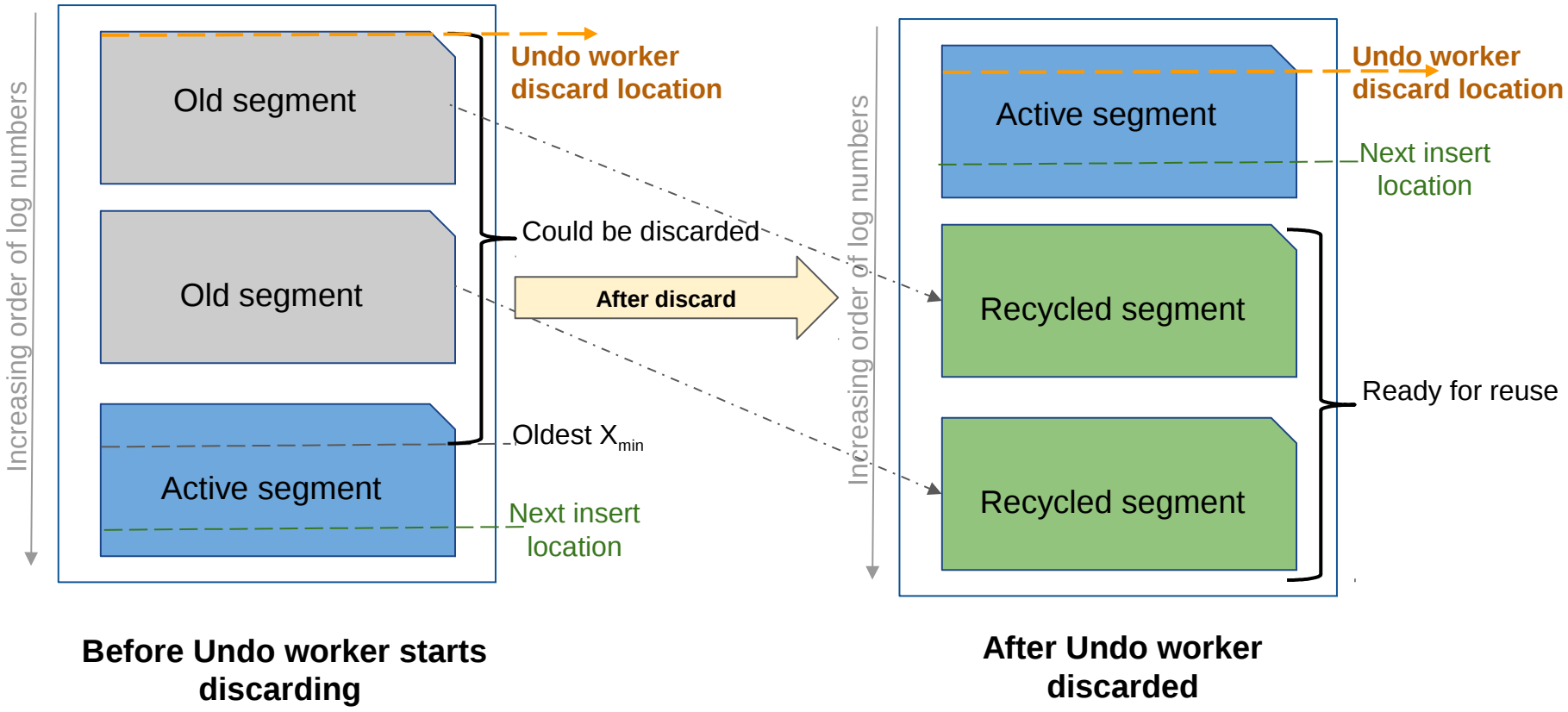
Rollbacks

- We need to apply undo actions during Rollback, Rollback To Savepoint and on an error.
- On an error, we apply undo actions in a new transaction.
- We do try to combine and apply the undo actions of a page
  - to cut down the effort for locking-unlocking the page and
  - to reduce the amount of WAL
- If the size of undo for a particular transaction is greater than certain threshold (configurable), then we push the rollback request to background undo worker.

Undo retention

- UNDO data needs to be retained till the active transactions needs to see old versions
  - All transactions which are in-progress
  - For aborted transactions till the time UNDO actions have been performed
  - For committed transactions till the time they are all-visible
- We could reduce the time period for which UNDO needs to be retained in category 3 by implementing “snapshot too old”.
- We consider undo for a transaction to be discardable once its XID is smaller than oldestXmin.

Undo log processing



Undo discard mechanism performed by discard worker

Undo log processing

- The job of discarding the undo logs is performed by discard worker.
- It process all the undo logs.
  - Discard unwanted undo segments from undo logs.
  - Forget all the buffers corresponding to discarded undo to avoid I/O (required to flush those buffers).
  - Identify the aborted transaction and add the request for its rollback in rollback_hash_table.
- Undo launcher checks the rollback_hash_table periodically and spawn new undo workers to perform the rollback.
- Each spawned undo worker processes the rollback requests for a particular database.

Indexing & zheap

- Current version of zheap works without any changes to index access methods.
- We plan to continue supporting the use of unmodified index access methods with zheap.
- However, if indexes are modified to support “delete-marking”, we could do in-place updates even when indexed columns are modified.
- When performing an in-place update, mark the old index entry as possibly-deleted, and insert a new one. No change to indexes where the columns aren’t modified.
    - Instead of an insert into every index, we incur an insert into only those indexes where there’s a change, plus we delete-mark an entry for each insert.

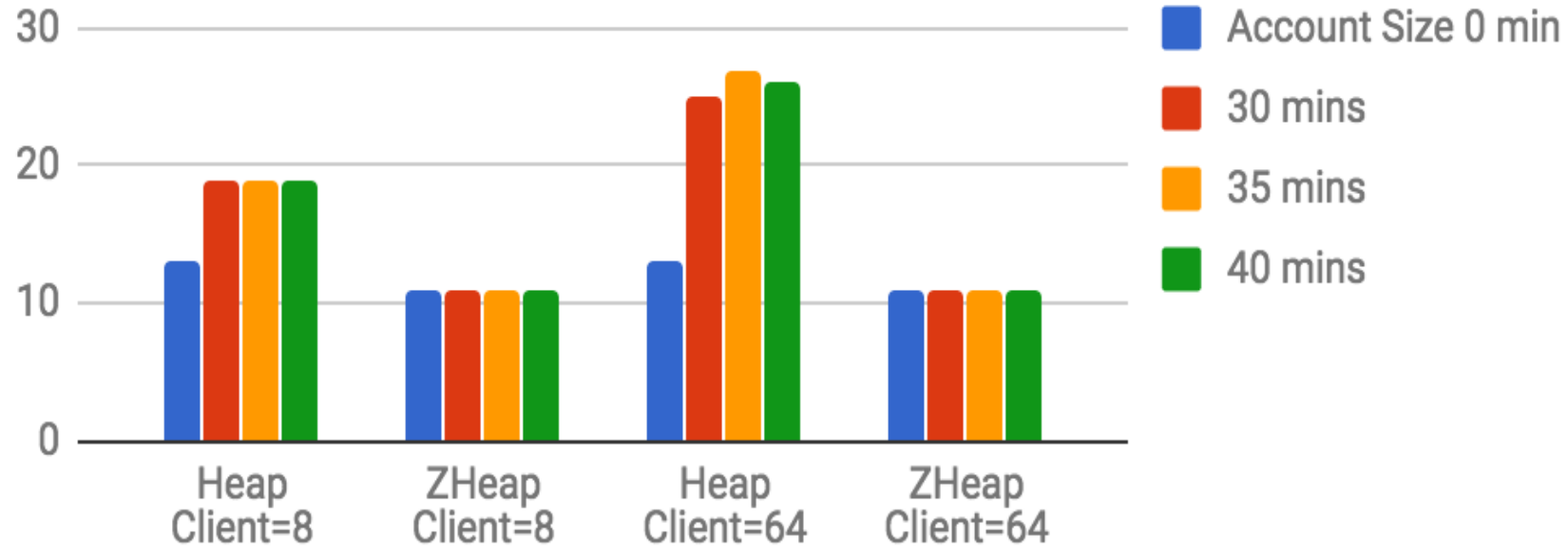
Eliminating VACUUM

- If all indexes on the table support delete-marking, maybe we don’t need VACUUM any more.
- Remember, zheap pages don’t need to be hinted, frozen, etc. If there are leftover tuples, we can remove them when we want to reuse the space, rather than doing it in advance.
- Delete-marked index tuples can be removed when the pages are scanned, or perhaps when they are evicted from shared buffers. Index pages that are never accessed again might be bloated, but that might not matter very much.
- If we don’t VACUUM, we can’t ever “lose” free space!
- Could still be an option for users wanting to clean up more aggressively.

Performance data (Test setup)

- Size and TPS comparison of heap and zheap
- We have used pgbench
  - to initialize the data (at scale factor 1000) and
  - then use the simple-update test (which comprises of one-update, one-select, one-insert) to perform updates.
- Machine details: x86_64 architecture, 2-sockets, 14-cores per socket, 2-threads per-core and has 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, synchoronous_commit = off;

Accounts Size in GB (1 trans open for 30 mins)



- 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% more than heap in above tests at 8 client-count. In some other high-end machines, we have seen up to ~100% improvement for similar test.

Benefits

- Performing updates in-place wherever possible prevents bloat from being created.
- Old tuple versions are removed eagerly from the heap (as soon as the transaction ends).
- The toast table data will be stored in zheap which means that the space for old rows in them will also be reclaimed eagerly.
- Most things that could cause a page to be rewritten multiple times are eliminated. Tuples no longer need to be frozen; instead, pages are implicitly frozen by the removal of associated UNDO.
- Because zheap is smaller on-disk, we get a small performance boost.
- In workloads where the heap bloats and zheap only bloats the undo, we get a massive performance boost.

Drawbacks

- Reading a page will be more expensive when there are active transactions operating on a page.
- Delete marking will have some overhead, but we will still win if there are many indexes on the table and only few of them got updated.
- Transaction abort can be lengthy.

Who?

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

Who?

- A special thanks to Robert Haas, Andres Freund and Thomas Munro who have provided a lot of valuable design inputs.
- Robert Haas has written an initial high-level design document for this project.
- Thomas Munro has implemented undo storage module in this project.

Thanks!