Bloat and Fragmentation in PostgreSQL

NTT Open Source Center
Masahiko Sawada

PGConf.ASIA 2018

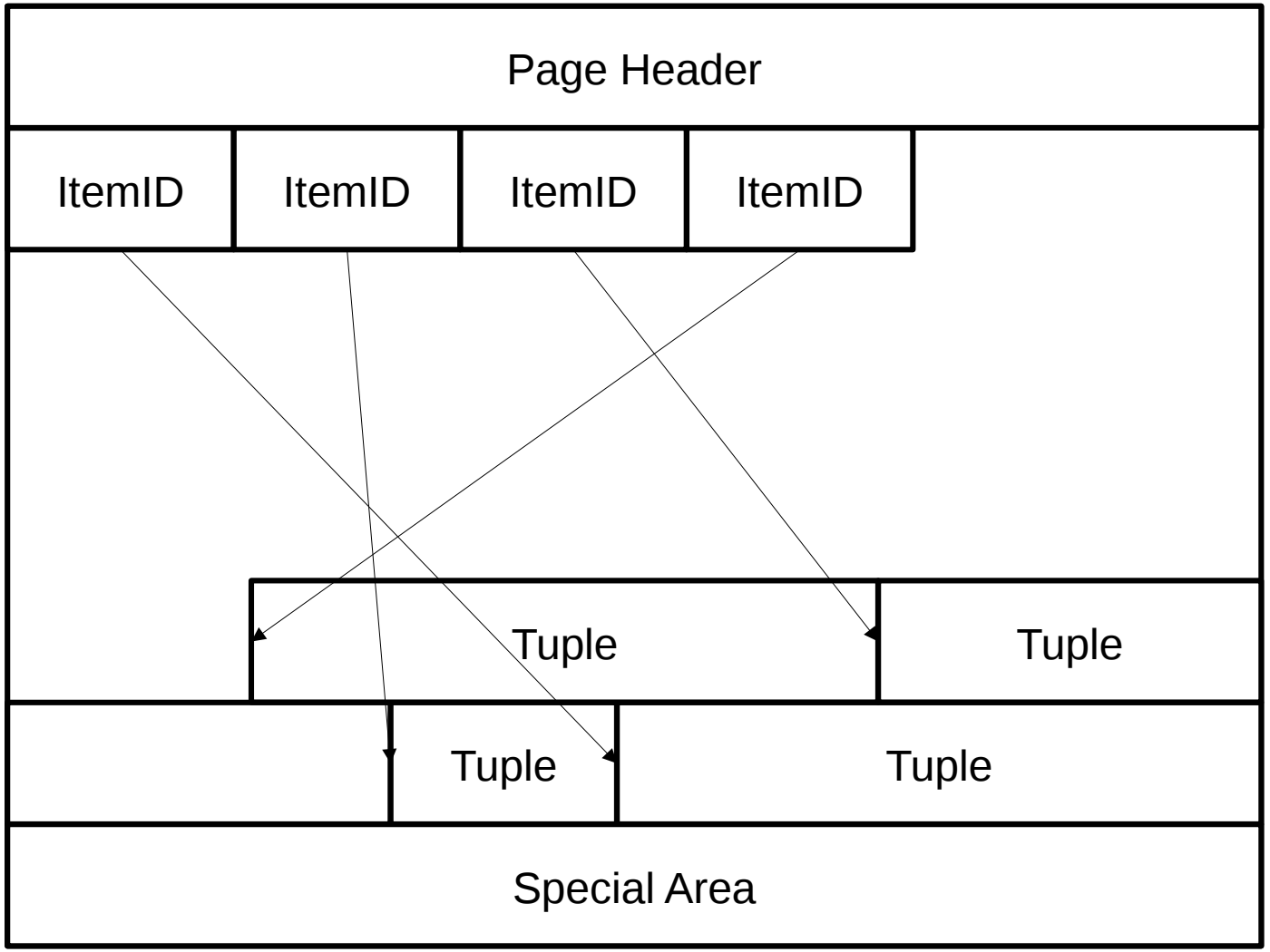
Index

- Heap and Index
- Fragmentation and Bloat
- VACUUM 、 HOT(Heap Only Tuple)
- Clustered Tables
- Future

Pages

- **Almost all objects such as tables and indexes that PostgreSQL manages consist of pages**
- **Page size is 8kB by default**
  - Changing by the configure script
- **Block (on disk) = Page (on memory)**

Page Layouts (Heaps and Indexes)



TID - Tuple ID

- **Combination of block number and offset number**
  - TID = (123, 86)
- **Unique with in a table**

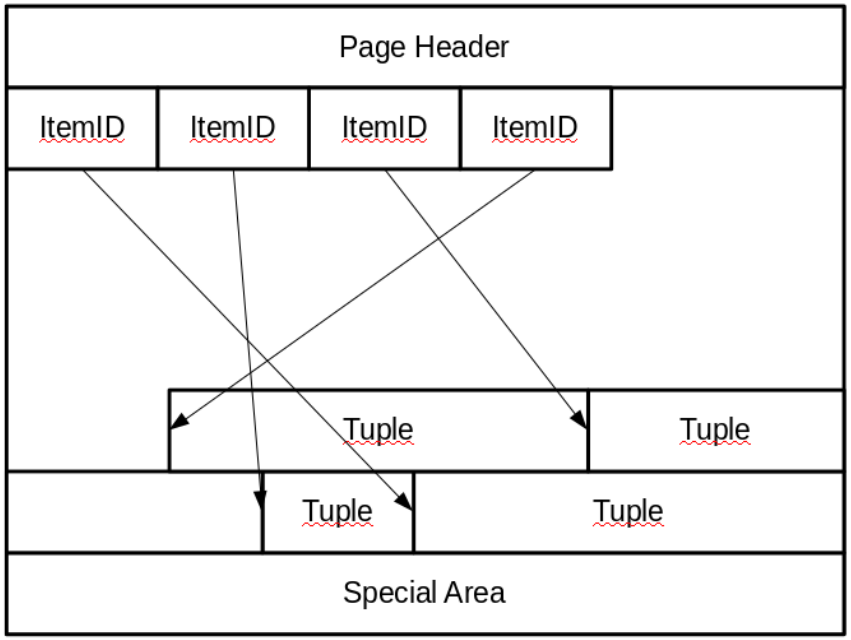
```
 =# SELECT ctid, c FROM test ORDER BY c LIMIT 10;
      ctid      |  c
----------------+----
 (88495,130)|  1
 (0,1)          |  1
 (88495,131)|  2
 (44247,179)|  2
 (0,2)          |  2
 (88495,132)|  3
 (44247,180)|  3
 (0,3)          |  3
 (88495,133)|  4
 (44247,181)|  4
(10 rows)
```

MVCC

- **MVCC = Multi Version Concurrency Control**
  - Concurrency control using multiple versions of row
  - Reads don't block writes
- **PostgreSQL's approach:**
  - **INSERT and UPDATE create the new version of the row**
  - **UPDATE and DELETE don't immediately remove the old version of the row**
  - **Tuple visibility is determined by a 'snapshot'**
- **Other DBMS might use UNDO log instead**
- **Old versions of rows have to be removed eventually**
  - VACUUM

Heap

- Heap ≒ Table
- User created table, materialized view as well as system catalogs use heap
- Optimized sequential access and accessed by index lookup
- Heaps have one free space map and one visibility map

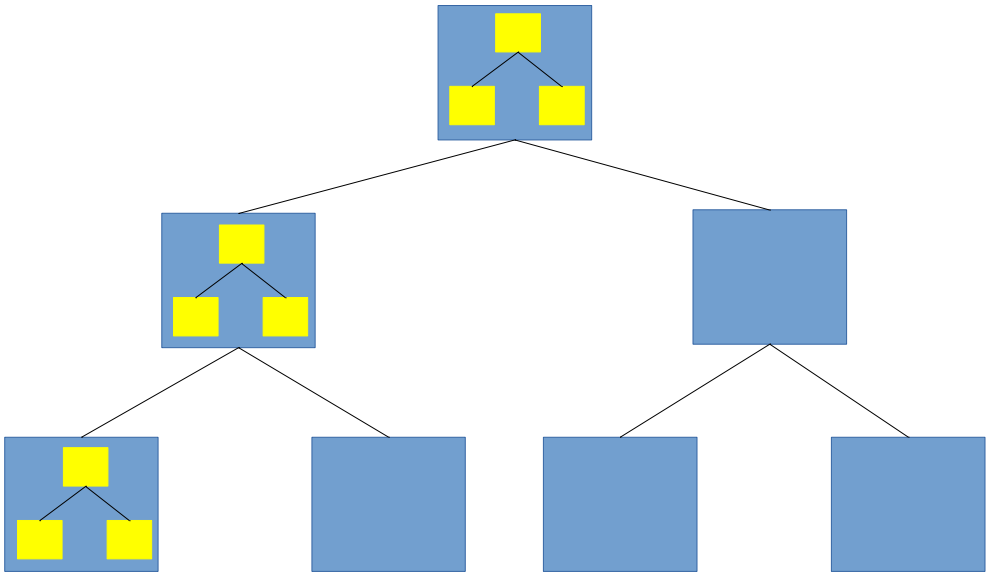


Visibility Map

- **Tracks visibility information per blocks**
- **1234_vm**
- **2 bits / block**
  - all-visible bit : Used for Index Only Scans and vacuums
  - all-frozen bit : Used for aggressive vacuums
- **Isn’t relevant with bloating**

Free Space Map

- **Manage available space in tables or indexes**
- **2-level binary tree**
  - Bottom level store the free space on each page
  - Upper levels aggregate information from the lower levels
- **Record free space at a granularity of 1/256th of a page**
- **1234_fsm**



Index

- **Fast lookup a row in the table**
- **Various types of index**
  - B-tree
  - Hash
  - GIN
  - Gist
  - SP-Gist
  - BRIN
  - Bloom
- **Indexes eventually points to somewhere in heap**

B-tree

- One of the most popular type of index
- PostgreSQL has B+Tree
- One B+tree node is one page
- Leaf nodes have TIDs pointing to the heap
  - Need new version index tuple when a new version is created

FRAGMENTATION AND BLOAT

Fragmentation and Bloat

- **Fragmentation**
  - As pages split to make room to added to a page, there might be excessive free space left on the pages
- **Bloat**
  - Tables or indexes gets bigger than its actual size
  - Less utilization efficiency of each pages

Vacuum

- Garbage collection
- Doesn’t block DELETE, INSERT and UPDATE (of course and SELECT)
- VACUUM command
- VACUUM FULL is quite different feature
- Batch operation

Vacuum Processing

1. **Scan heap and collect dead tuples**
    1. Scanning page can skip using its visibility map
2. **Recover dead space in all indexes**
3. **Recover dead space in heap**
4. **Loop 1 to 3 until the end of table**
5. **Truncate the tail of table if possible**

Batch Operation

- Always start at the beginning of table
- Always visit all indexes (at multiple times)

Auto Vacuum

- **Threshold-based, automatically vacuum execution**
- **Could be cancelled by a concurrent conflicting operation**
  - e.g. ALTER TABLE 、 TRUNCATE
- **Vacuum delay is enabled by default**

Vacuum Delay

- **Cost-based vacuum delay**
- **Sleep vacuum_cost_delay(10 msec) time whenever the vacuum cost goes above vacuum_cost_limit(200)**

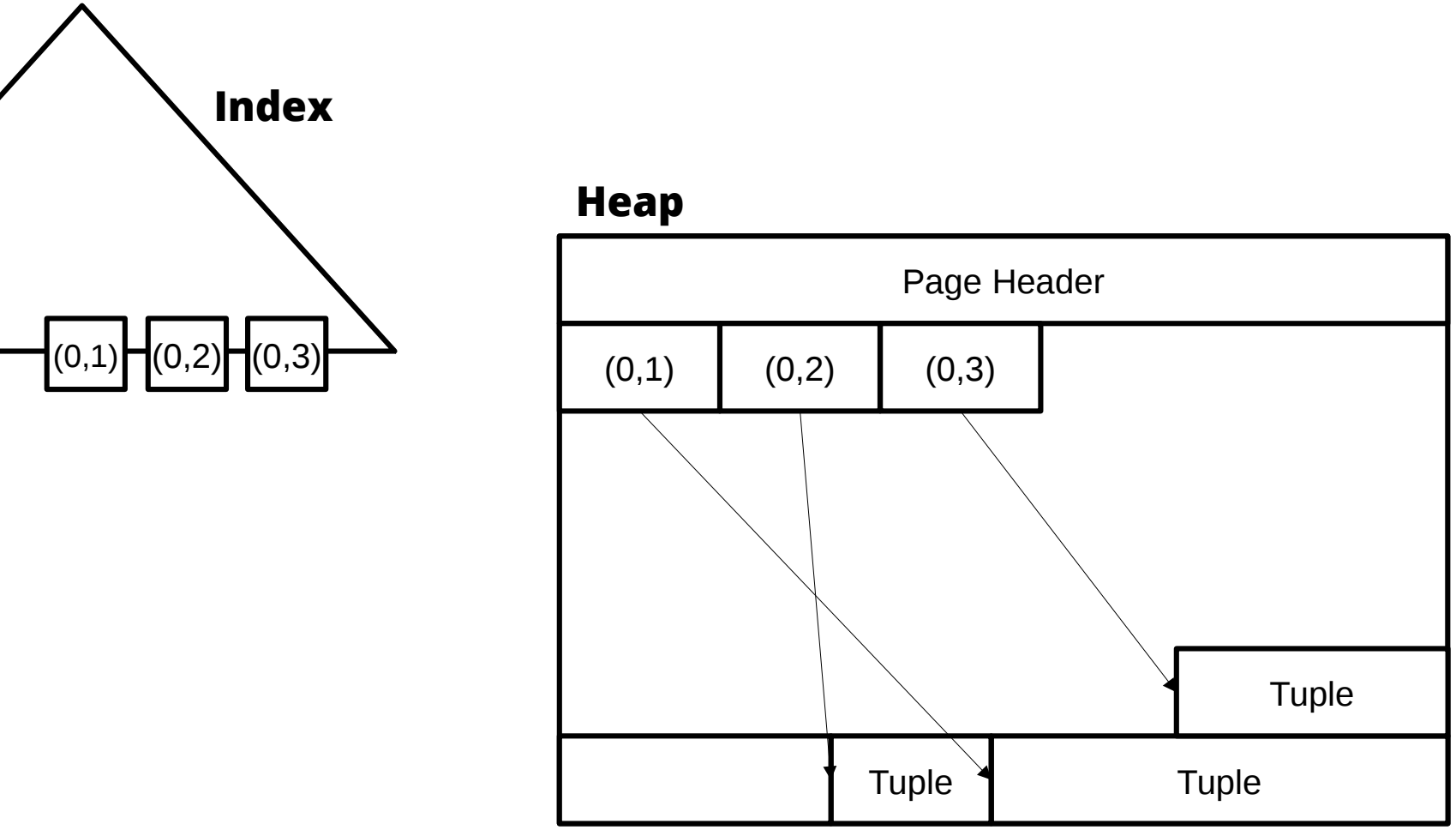
- **Cost configurations**
  - vacuum_cost_page_hit (1)
  - vacuum_cost_page_miss (10)
  - vacuum_cost_page_dirty (20)

- **By default, vacuum processes 16GB/h at maximum in case where every pages don't hit**

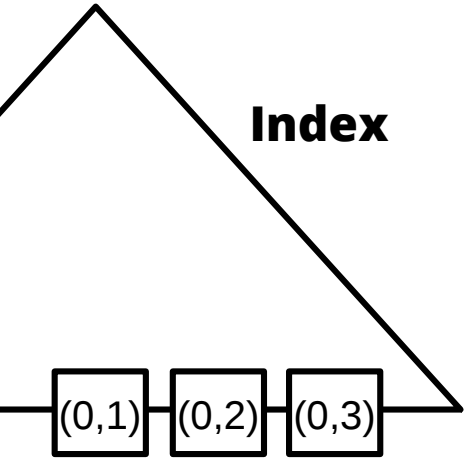
HOT(Heap Only Tuple) Update

- Avoid creating new version of index entries
- Depending on updated column being updated
- Chain heap tuples
- Heap tuple chain is pruned opportunistically

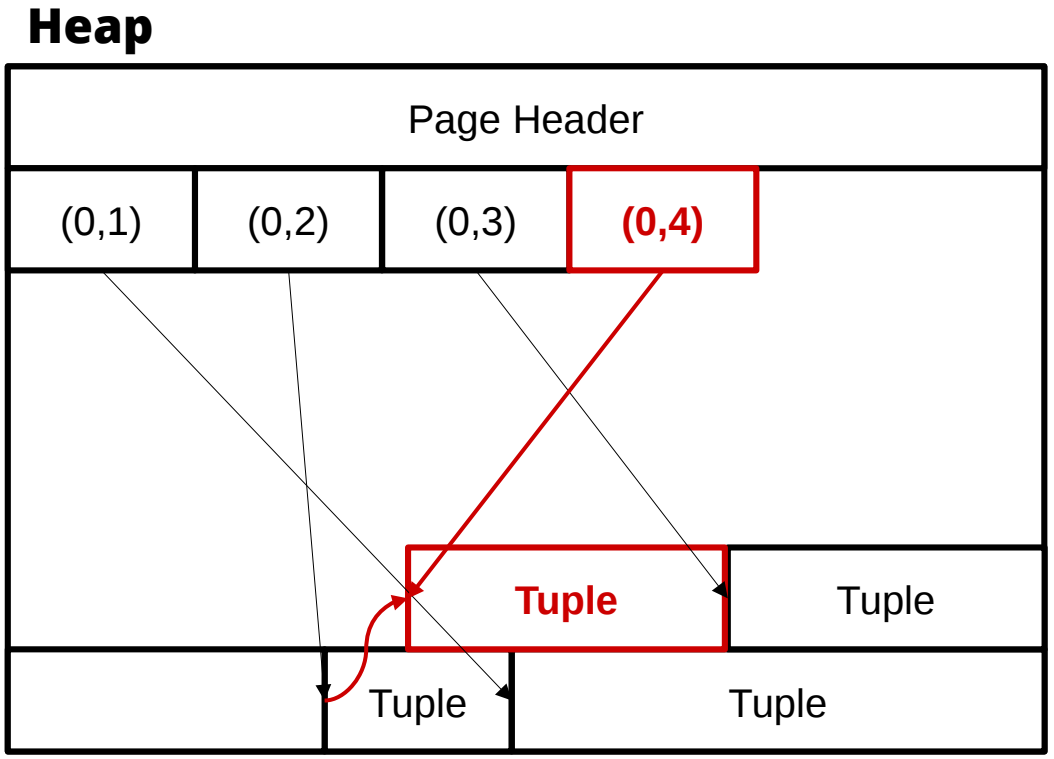
HOT Update & HOT pruning



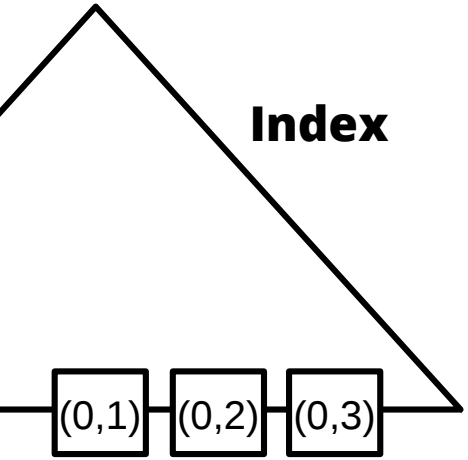
HOT Update & HOT pruning



1. Update TID = (0,2) → (0,4)

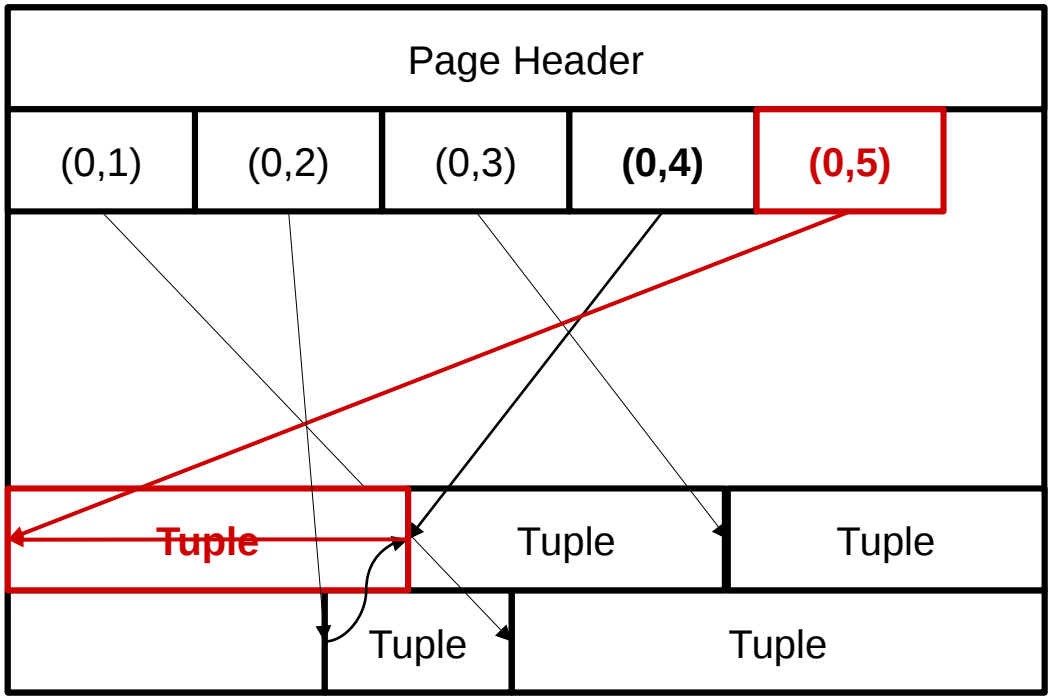


HOT Update & HOT pruning

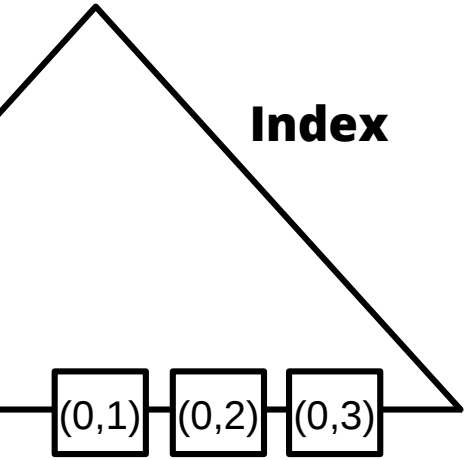


2. Update TID = (0,4) → (0,5)

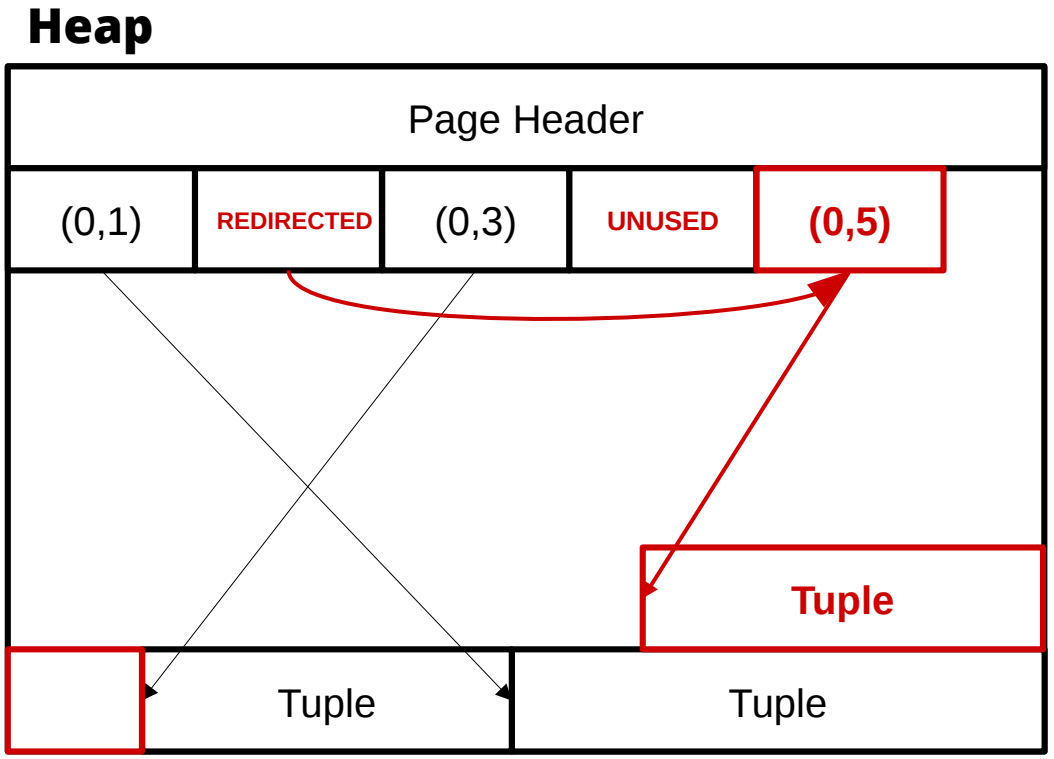
Heap



HOT Update & HOT pruning



3. VACUUM



Using HOT Update

- Two conditions to use HOT updating
  - Enough free space in the same page
  - No Indexed column is not updated
- Perform HOT-pruning when the page utilization goes above 90%
- IMO, the most important feature to prevent the table bloat

Causes of Bloat

- **Fragmentation**
  - Free space map is not up-to-date
- **Shortage of vacuum**
- **Concurrent long-transaction**

Shortage of Vacuum

- **Garbage collection speed < Generating garbage speed**

- **Solution : making vacuums run more faster**
  - Decrease vacuum delays
  - Increase maintenance_work_mem
    - Ideally perform the index vacuum only once

Vacuum and Snapshot

- “Due to long-running transaction dead spaces are not reclaimed” by a user

- That’s correct in a sense but this is actually inaccurate:
  - Vacuum doesn’t remove row if there is a concurrent snapshot that can see it, even for a one snapshot!
  - This also means these row is NOT dead yet
- A row is dead when no one can see it

CLUSTERD TABLES

Clustered Table

- An order of value on a column matches physical block order
- pg_stats.correlation
- CLUSTER command

Correlations

```
=# SELECT * FROM tt LIMIT 10;
 a |   b    |   c
---+--------+--------
 1 | 9988   | 100000
 2 |  176   |  99999
 3 | 1066   |  99998
 4 | 1980   |  99997
 5 | 2966   |  99996
 6 | 5732   |  99995
 7 | 1751   |  99994
 8 | 3813   |  99993
 9 | 3031   |  99992
10 | 5332   |  99991
(10 rows)

=# SELECT attname, correlation FROM pg_stats WHERE tablename = 'tt';
 attname | correlation
---------+-------------
    a    |           1
    b    |  0.00493127
    c    |          -1
(3 rows)
```

Performance Impact

```
=# EXPLAIN (buffers on, analyze on) SELECT * FROM tt WHERE a BETWEEN 200 AND 1000 LIMIT 1000;

                                                        QUERY PLAN
---------------------------------------------------------------------------------------------------------------------------
 Limit  (cost=0.29..38.07 rows=889 width=12) (actual time=0.063..1.153 rows=801 loops=1)
   Buffers: shared hit=9
   ->  Index Scan using tt_a on tt  (cost=0.29..38.07 rows=889 width=12) (actual time=0.060..0.942
rows=801 loops=1)
         Index Cond: ((a >= 200) AND (a <= 1000))
         Buffers: shared hit=9
 Planning Time: 0.388 ms
 Execution Time: 1.380 ms
(7 rows)

=# EXPLAIN (buffers on, analyze on) SELECT * FROM tt WHERE b BETWEEN 200 AND 1000 LIMIT 1000;

                                                        QUERY PLAN
---------------------------------------------------------------------------------------------------------------------------
 Limit  (cost=0.29..306.76 rows=1000 width=12) (actual time=0.052..3.176 rows=1000 loops=1)
   Buffers: shared hit=992
   ->  Index Scan using tt_b on tt  (cost=0.29..2435.18 rows=7945 width=12) (actual time=0.048..2.797
rows=1000 loops=1)
         Index Cond: ((b >= 200) AND (b <= 1000))
         Buffers: shared hit=992
 Planning Time: 0.749 ms
 Execution Time: 3.493 ms
(7 rows)
```

FUTURE

The Future of Vacuum

- **More faster**
  - Parallelism
- **Eager vacuum (retail index deletion)**
  - **Non-batch operation**

Good bye Vacuum ...?

- Pluggable Storage Engine

- zheap - an UNDO log based new storage engine
  - Prevent bloating by UPDATE

Conclusion

- **PostgreSQL creates multiple versions of rows within the same table**
- **Has to eventually get rid of the unnecessary row versions**
- **Vacuum and auto vacuum**
  - Batched garbage collection
- **HOT pruning**
  - Opportunistic garbage collection
- **Clustered Table**
- **Future**
  - Parallelism and eager vacuum
  - zheap

THANK YOU!