Mvcc Unmasked

BRUCE MOMJIAN

This talk explains how Multiversion Concurrency Control (MVCC) is implemented in Postgres, and highlights optimizations which minimize the downsides of MVCC.

https://momjian.us/presentations

Last updated: February 2026

Unmasked: Who Are These People?

https://www.flickr.com/photos/danielsemper/

Unmasked: The Original Star Wars Cast

Left to right: Han Solo, Darth Vader, Chewbacca, Leia, Luke Skywalker, R2D2

Why Unmask MVCC?

- Predict concurrent query behavior
- Manage MVCC performance effects
- Understand storage space reuse

Outline

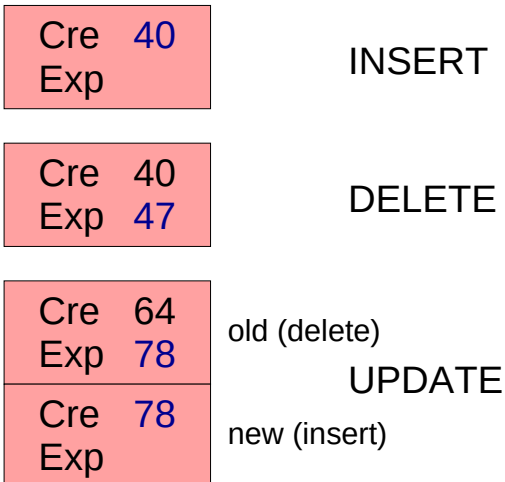
What is MVCC?

Multiversion Concurrency Control (MVCC) allows Postgres to offer high concurrency even during significant database read/write activity. MVCC specifically offers behavior where "readers never block writers, and writers never block readers".
This presentation explains how MVCC is implemented in Postgres, and highlights optimizations which minimize the downsides of MVCC.

Which Other Database Systems Support MVCC?

- Oracle
- DB2 (partial)
- MySQL with InnoDB
- Informix
- Firebird
- MSSQL (optional, disabled by default)

MVCC Behavior



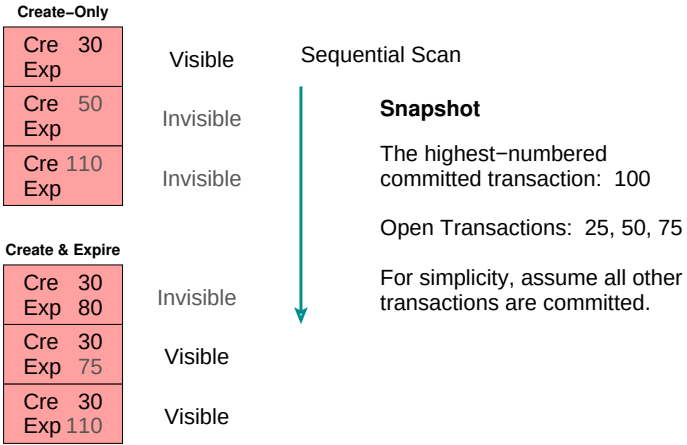
MVCC Snapshots

MVCC snapshots control which tuples are visible for SQL statements. A snapshot is recorded at the start of each SQL statement in READ COMMITTED transaction isolation mode, and only at transaction start in REPEATABLE READ and SERIALIZABLE transaction isolation modes. In fact, it is the frequency of taking new snapshots that controls the transaction isolation behavior.
When a new snapshot is taken, the following information is gathered:

- the highest-numbered committed transaction
- the transaction numbers currently executing

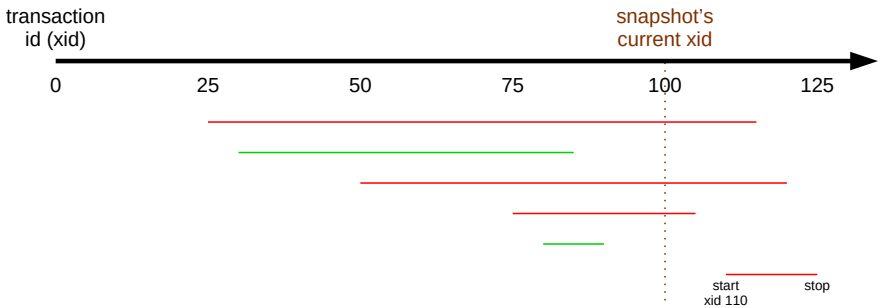
Using this snapshot information, Postgres can determine if a transaction's actions should be visible to an executing statement.

MVCC Snapshots Determine Row Visibility



Internally, the creation xid is stored in the system column 'xmin', and expire in 'xmax'.

MVCC Snapshot Timeline



Green is visible. Red is invisible.

Only transactions completed before transaction id 100 started are visible.

Confused Yet?

Source code comment in *src/backend/utils/time/tqual.c*:

```
((Xmin == my-transaction &&                inserted by the current transaction
  Cmin < my-command &&                     before this command, and
  (Xmax is null ||                         the row has not been deleted, or
   (Xmax == my-transaction &&              it was deleted by the current transaction
    Cmax >= my-command)))                  but not before this command,
||                                         or
 (Xmin is committed &&                     the row was inserted by a committed transaction, and
     (Xmax is null ||                      the row has not been deleted, or
      (Xmax == my-transaction &&           the row is being deleted by this transaction
       Cmax >= my-command) ||              but it's not deleted "yet", or
      (Xmax != my-transaction &&           the row was deleted by another transaction
       Xmax is not committed))))           that has not been committed
```

mao [Mike Olson] says 17 march 1993: *the tests in this routine are correct; if you think they're not, you're wrong, and you should think about it again. i know, it happened to me.*

Implementation Details

All queries were generated on an unmodified version of Postgres. The contrib module *pageinspect* was installed to show internal heap page information and *pg_freespacemap* was installed to show free space map information.

Setup

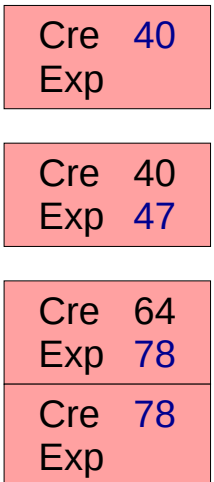
```
CREATE TABLE mvcc_demo (val INTEGER);

DROP VIEW IF EXISTS mvcc_demo_page0;

CREATE EXTENSION pageinspect;

CREATE EXTENSION pg_freespacemap;

CREATE VIEW mvcc_demo_page0 AS
        SELECT  '(0,' || lp || ')' AS ctid,
                CASE lp_flags
                        WHEN 0 THEN 'Unused'
                        WHEN 1 THEN 'Normal'
                        WHEN 2 THEN 'Redirect to ' || lp_off
                        WHEN 3 THEN 'Dead'
                END,
                t_xmin::text::int8 AS xmin,
                t_xmax::text::int8 AS xmax,
                t_ctid
        FROM heap_page_items(get_raw_page('mvcc_demo', 0))
        ORDER BY lp;
```

INSERT Using Xmin

```
DELETE FROM mvcc_demo;

INSERT INTO mvcc_demo VALUES (1);

SELECT xmin, xmax, * FROM mvcc_demo;
 xmin | xmax | val
------+------+-----
 5409 |    0 |   1
```

All queries used in this presentation are available at https://momjian.us/main/writings/pgsql/mvcc.sql.

Just Like INSERT



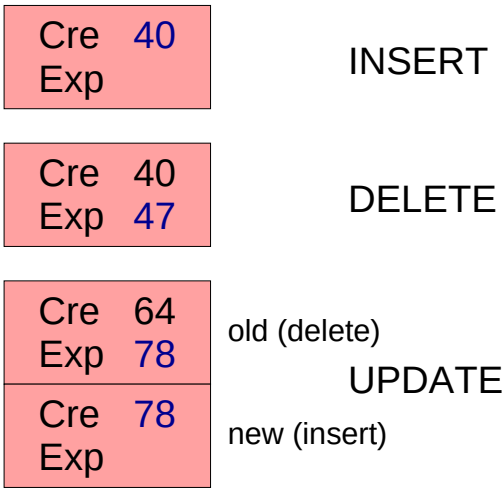
DELETE Using Xmax

```
DELETE FROM mvcc_demo;

INSERT INTO mvcc_demo VALUES (1);

SELECT xmin, xmax, * FROM mvcc_demo;
 xmin | xmax | val
------+------+-----
 5411 |    0 |   1

BEGIN WORK;

DELETE FROM mvcc_demo;
```

DELETE Using Xmax

```
SELECT xmin, xmax, * FROM mvcc_demo;
 xmin | xmax | val
------+------+-----


      SELECT xmin, xmax, * FROM mvcc_demo;
         xmin | xmax | val
        ------+------+-----
         5411 | 5412 |   1

SELECT txid_current();
 txid_current
--------------
         5412

COMMIT WORK;
```

Just Like DELETE



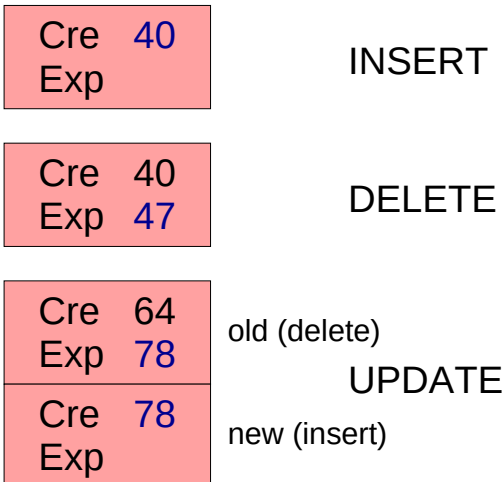
UPDATE Using Xmin and Xmax

```
DELETE FROM mvcc_demo;

INSERT INTO mvcc_demo VALUES (1);

SELECT xmin, xmax, * FROM mvcc_demo;
 xmin | xmax | val
------+------+-----
 5413 |    0 |   1

BEGIN WORK;

UPDATE mvcc_demo SET val = 2;
```

UPDATE Using Xmin and Xmax

```
SELECT xmin, xmax, * FROM mvcc_demo;
 xmin | xmax | val
------+------+-----
 5414 |    0 |   2

        SELECT xmin, xmax, * FROM mvcc_demo;
         xmin | xmax | val
        ------+------+-----
         5413 | 5414 |   1

COMMIT WORK;
```

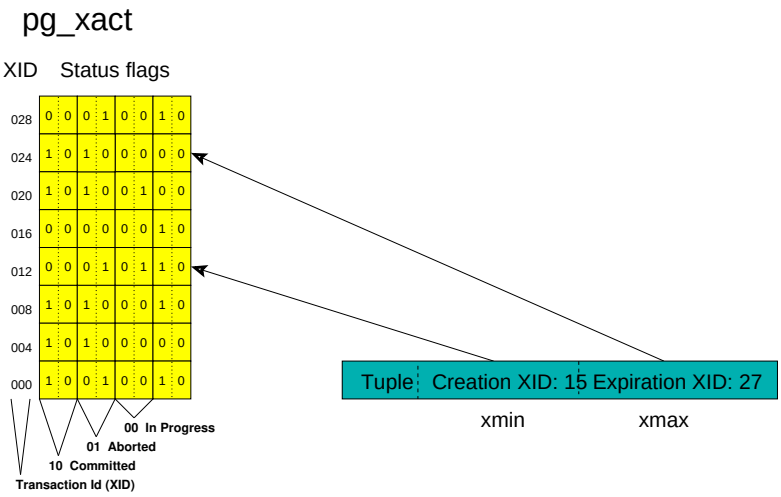
Just Like UPDATE



Aborted Transaction IDs Remain

```
DELETE FROM mvcc_demo;

INSERT INTO mvcc_demo VALUES (1);

BEGIN WORK;

DELETE FROM mvcc_demo;

ROLLBACK WORK;

SELECT xmin, xmax, * FROM mvcc_demo;
 xmin | xmax | val
------+------+-----
 5415 | 5416 |   1
```

Aborted IDs Remain, Transaction Status Is Recorded Centrally



Transaction roll back marks the transaction ID as aborted. All sessions will ignore such transactions; it is not necessary to revisit each row to undo the transaction.

Row Locks Using Xmax

```
DELETE FROM mvcc_demo;

INSERT INTO mvcc_demo VALUES (1);

BEGIN WORK;

SELECT xmin, xmax, * FROM mvcc_demo;
 xmin | xmax | val
------+------+-----
 5416 |    0 |   1

SELECT xmin, xmax, * FROM mvcc_demo FOR UPDATE;
 xmin | xmax | val
------+------+-----
 5416 |    0 |   1
```

Row Locks Using Xmax

```
SELECT xmin, xmax, * FROM mvcc_demo;
 xmin | xmax | val
------+------+-----
 5416 | 5417 |   1

COMMIT WORK;
```

The tuple bit HEAP_XMAX_EXCL_LOCK is used to indicate that xmax is a locking xid rather than an expiration xid.

Multi-Statement Transactions

Multi-statement transactions require extra tracking because each statement has its own visibility rules. For example, a cursor's contents must remain unchanged even if later statements in the same transaction modify rows. Such tracking is implemented using system command id columns cmin/cmax, which is internally actually is a single column.

INSERT Using Cmin

```
DELETE FROM mvcc_demo;

BEGIN WORK;

INSERT INTO mvcc_demo VALUES (1);
INSERT INTO mvcc_demo VALUES (2);
INSERT INTO mvcc_demo VALUES (3);
```

INSERT Using Cmin

```
SELECT xmin, cmin, xmax, * FROM mvcc_demo;
 xmin | cmin | xmax | val
------+------+------+-----
 5419 |    0 |    0 |   1
 5419 |    1 |    0 |   2
 5419 |    2 |    0 |   3

COMMIT WORK;
```

DELETE Using Cmin

```
DELETE FROM mvcc_demo;

BEGIN WORK;

INSERT INTO mvcc_demo VALUES (1);
INSERT INTO mvcc_demo VALUES (2);
INSERT INTO mvcc_demo VALUES (3);
```

DELETE Using Cmin

```
SELECT xmin, cmin, xmax, * FROM mvcc_demo;
 xmin | cmin | xmax | val
------+------+------+-----
 5421 |    0 |    0 |   1
 5421 |    1 |    0 |   2
 5421 |    2 |    0 |   3

DECLARE c_mvcc_demo CURSOR FOR
SELECT xmin, xmax, cmax, * FROM mvcc_demo;
```

DELETE Using Cmin

```
DELETE FROM mvcc_demo;

SELECT xmin, cmin, xmax, * FROM mvcc_demo;
 xmin | cmin | xmax | val
------+------+------+-----

FETCH ALL FROM c_mvcc_demo;
 xmin | xmax | cmax | val
------+------+------+-----
 5421 | 5421 |    0 |   1
 5421 | 5421 |    1 |   2
 5421 | 5421 |    2 |   3

COMMIT WORK;
```

A cursor had to be used because the rows were created and deleted in this transaction and therefore never visible outside this transaction.

UPDATE Using Cmin

```
DELETE FROM mvcc_demo;

BEGIN WORK;

INSERT INTO mvcc_demo VALUES (1);
INSERT INTO mvcc_demo VALUES (2);
INSERT INTO mvcc_demo VALUES (3);
SELECT xmin, cmin, xmax, * FROM mvcc_demo;
 xmin | cmin | xmax | val
------+------+------+-----
 5422 |    0 |    0 |   1
 5422 |    1 |    0 |   2
 5422 |    2 |    0 |   3

DECLARE c_mvcc_demo CURSOR FOR
SELECT xmin, xmax, cmax, * FROM mvcc_demo;
```

UPDATE Using Cmin

```
UPDATE mvcc_demo SET val = val * 10;

SELECT xmin, cmin, xmax, * FROM mvcc_demo;
 xmin | cmin | xmax | val
------+------+------+-----
 5422 |    3 |    0 |  10
 5422 |    3 |    0 |  20
 5422 |    3 |    0 |  30

FETCH ALL FROM c_mvcc_demo;
 xmin | xmax | cmax | val
------+------+------+-----
 5422 | 5422 |    0 |   1
 5422 | 5422 |    1 |   2
 5422 | 5422 |    2 |   3

COMMIT WORK;
```

Modifying Rows From Different Transactions

```
DELETE FROM mvcc_demo;

INSERT INTO mvcc_demo VALUES (1);

SELECT xmin, xmax, * FROM mvcc_demo;
 xmin | xmax | val
------+------+-----
 5424 |    0 |   1

BEGIN WORK;

INSERT INTO mvcc_demo VALUES (2);
INSERT INTO mvcc_demo VALUES (3);
INSERT INTO mvcc_demo VALUES (4);
```

Modifying Rows From Different Transactions

```
SELECT xmin, cmin, xmax, * FROM mvcc_demo;
 xmin | cmin | xmax | val
------+------+------+-----
 5424 |    0 |    0 |   1
 5425 |    0 |    0 |   2
 5425 |    1 |    0 |   3
 5425 |    2 |    0 |   4

UPDATE mvcc_demo SET val = val * 10;
```

Modifying Rows From Different Transactions

```
SELECT xmin, cmin, xmax, * FROM mvcc_demo;
 xmin | cmin | xmax | val
------+------+------+-----
 5425 |    3 |    0 |  10
 5425 |    3 |    0 |  20
 5425 |    3 |    0 |  30
 5425 |    3 |    0 |  40
```

```
SELECT xmin, xmax, cmax, * FROM mvcc_demo;
  xmin | xmax | cmax | val
 ------+------+------+-----
  5424 | 5425 |    3 |   1
```

```
COMMIT WORK;
```

Combo Command Id

Because *cmin* and *cmax* are internally a single system column, it is impossible to simply record the status of a row that is created and expired in the same multi-statement transaction. For that reason, a special combo command id is created that references a local memory hash that contains the actual cmin and cmax values.

UPDATE Using Combo Command Ids

```
-- use TRUNCATE to remove even invisible rows
TRUNCATE mvcc_demo;

BEGIN WORK;

DELETE FROM mvcc_demo;
DELETE FROM mvcc_demo;
DELETE FROM mvcc_demo;
INSERT INTO mvcc_demo VALUES (1);
INSERT INTO mvcc_demo VALUES (2);
INSERT INTO mvcc_demo VALUES (3);
```

UPDATE Using Combo Command Ids

```
SELECT xmin, cmin, xmax, * FROM mvcc_demo;
 xmin | cmin | xmax | val
------+------+------+-----
 5427 |    3 |    0 |   1
 5427 |    4 |    0 |   2
 5427 |    5 |    0 |   3

DECLARE c_mvcc_demo CURSOR FOR
SELECT xmin, xmax, cmax, * FROM mvcc_demo;

UPDATE mvcc_demo SET val = val * 10;
```

UPDATE Using Combo Command Ids

```
SELECT xmin, cmin, xmax, * FROM mvcc_demo;
 xmin | cmin | xmax | val
------+------+------+-----
 5427 |    6 |    0 |  10
 5427 |    6 |    0 |  20
 5427 |    6 |    0 |  30

FETCH ALL FROM c_mvcc_demo;
 xmin | xmax | cmax | val
------+------+------+-----
 5427 | 5427 |    0 |   1
 5427 | 5427 |    1 |   2
 5427 | 5427 |    2 |   3
```

UPDATE Using Combo Command Ids

```
SELECT  t_xmin AS xmin,
        t_xmax::text::int8 AS xmax,
        t_field3::text::int8 AS cmin_cmax,
        (t_infomask::integer & X'0020'::integer)::bool AS is_combocid
FROM heap_page_items(get_raw_page('mvcc_demo', 0))
ORDER BY 2 DESC, 3;
 xmin | xmax | cmin_cmax | is_combocid
------+------+-----------+-------------
 5427 | 5427 |         0 | t
 5427 | 5427 |         1 | t
 5427 | 5427 |         2 | t
 5427 |    0 |         6 | f
 5427 |    0 |         6 | f
 5427 |    0 |         6 | f

COMMIT WORK;
```

The last query uses /contrib/pageinspect, which allows visibility of internal heap page structures and all stored rows, including those not visible in the current snapshot. (Bit *0x0020* is internally called HEAP_COMBOCID.)

MVCC Implementation Summary

xmin:	creation transaction number, set by INSERT and UPDATE
xmax:	expire transaction number, set by UPDATE and DELETE; also used for explicit row locks
cmin/cmax:	used to identify the command number that created or expired the tuple; also used to store combo command ids when the tuple is created and expired in the same transaction, and for explicit row locks

Traditional Cleanup Requirements

Traditional single-row-version (non-MVCC) database systems require storage space cleanup:

- deleted rows
- rows created by aborted transactions

MVCC Cleanup Requirements

MVCC has additional cleanup requirements:

- The creation of a new row during UPDATE (rather than replacing the existing row); the storage space taken by the old row must eventually be recycled.
- The *delayed* cleanup of deleted rows (cleanup cannot occur until there are no transactions for which the row is visible)

Postgres handles both traditional and MVCC-specific cleanup requirements.

Cleanup Behavior

Fortunately, Postgres cleanup happens automatically:

- On-demand cleanup of a single heap page during row access, specifically when a page is accessed by SELECT, UPDATE, and DELETE
- In bulk by an autovacuum processes that runs in the background

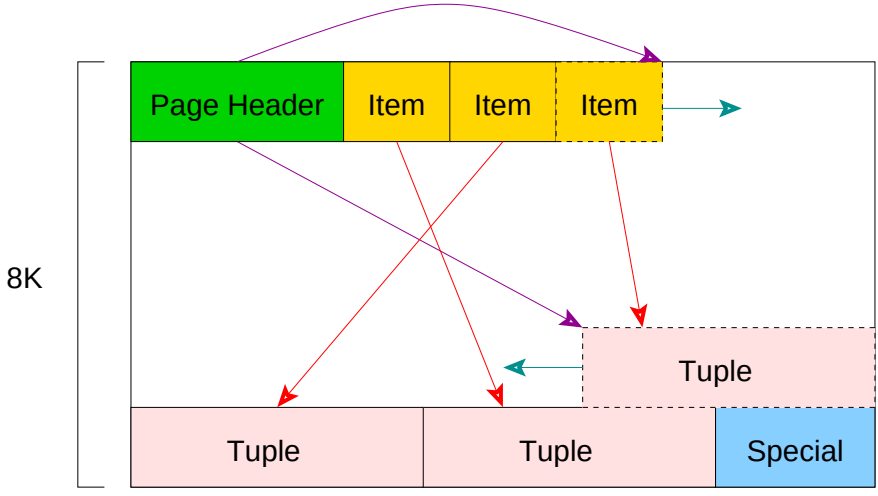
Cleanup can also be initiated manually by VACUUM.

Aspects of Cleanup

Cleanup involves recycling space taken by several entities:

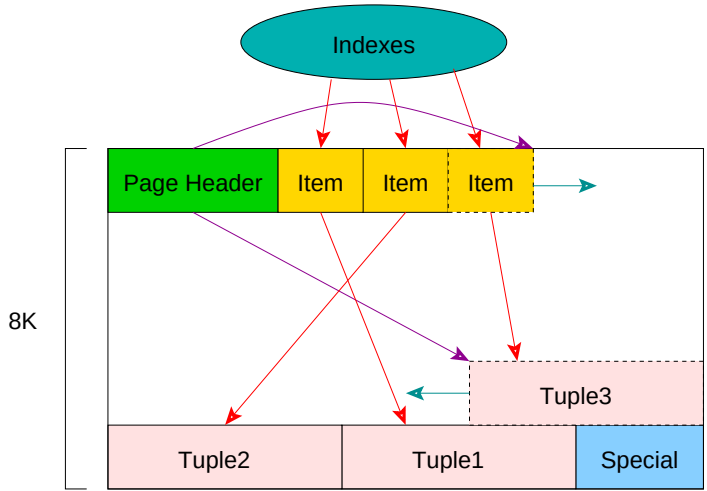
- heap tuples/rows (the largest)
- heap item pointers (the smallest)
- index entries

Internal Heap Page

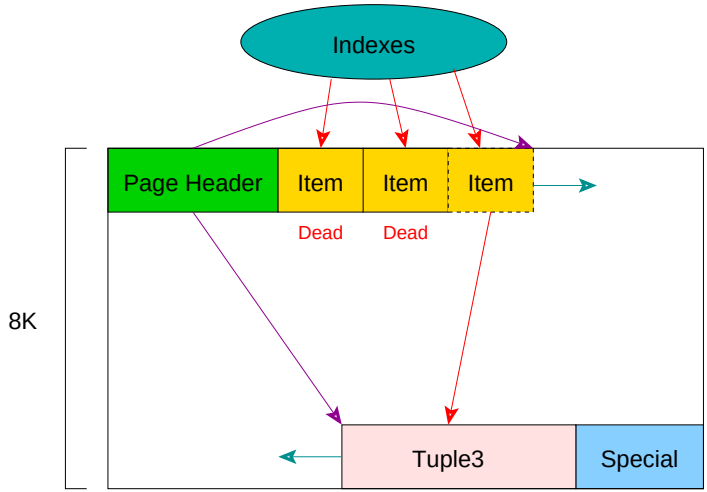


https://stormatics.tech/blogs/postgresql-internals-part-2-understanding-page-structure

Indexes Point to Items, Not Tuples

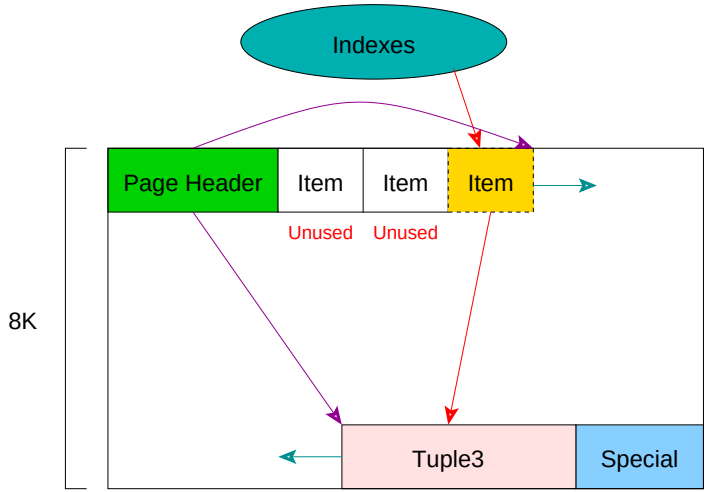


Heap Tuple Space Recycling



Indexes prevent item pointers from being recycled.

VACUUM Later Recycle Items



VACUUM performs index cleanup, then can mark “dead” items as “unused”.

Cleanup of Deleted Rows

```
TRUNCATE mvcc_demo;

-- force page to < 10% empty
INSERT INTO mvcc_demo SELECT 0 FROM generate_series(1, 220);

-- compute free space percentage
SELECT (100 * (upper - lower) /
        pagesize::float8)::integer AS free_pct
FROM page_header(get_raw_page('mvcc_demo', 0));
 free_pct
----------
        3

INSERT INTO mvcc_demo VALUES (1);
```

Cleanup of Deleted Rows

```
SELECT * FROM mvcc_demo_page0
OFFSET 220;
  ctid   |  case  | xmin | xmax | t_ctid
---------+--------+------+------+---------
 (0,221) | Normal | 5430 |    0 | (0,221)

DELETE FROM mvcc_demo WHERE val > 0;

INSERT INTO mvcc_demo VALUES (2);

SELECT * FROM mvcc_demo_page0
OFFSET 220;
  ctid   |  case  | xmin | xmax | t_ctid
---------+--------+------+------+---------
 (0,221) | Normal | 5430 | 5431 | (0,221)
 (0,222) | Normal | 5432 |    0 | (0,222)
```

Cleanup of Deleted Rows

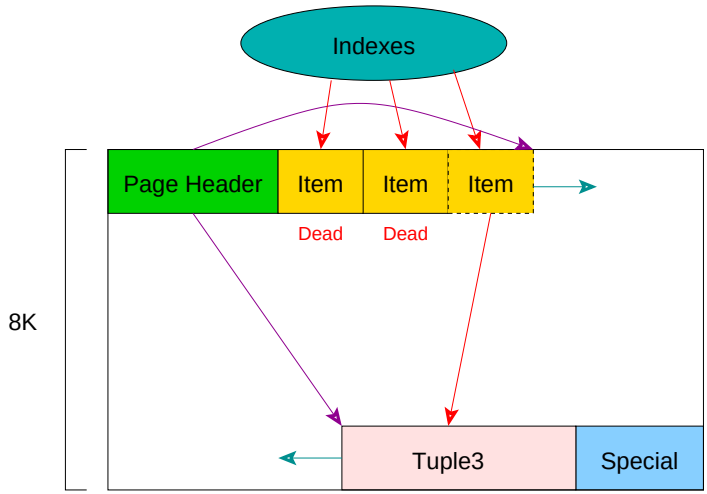
```
DELETE FROM mvcc_demo WHERE val > 0;

INSERT INTO mvcc_demo VALUES (3);

SELECT * FROM mvcc_demo_page0
OFFSET 220;
  ctid   |  case  | xmin | xmax | t_ctid
---------+--------+------+------+---------
 (0,221) | Dead   |      |      |
 (0,222) | Normal | 5432 | 5433 | (0,222)
 (0,223) | Normal | 5434 |    0 | (0,223)
```

In normal, multi-user usage, cleanup might have been delayed because other open transactions in the same database might still need to view the expired rows. However, the behavior would be the same, just delayed.

Cleanup of Deleted Rows

```
-- force single-page cleanup via SELECT
SELECT * FROM mvcc_demo
OFFSET 1000;
 val
-----


SELECT * FROM mvcc_demo_page0
OFFSET 220;
  ctid   |  case  | xmin | xmax | t_ctid
---------+--------+------+------+---------
 (0,221) | Dead   |      |      |
 (0,222) | Dead   |      |      |
 (0,223) | Normal | 5434 |    0 | (0,223)
```

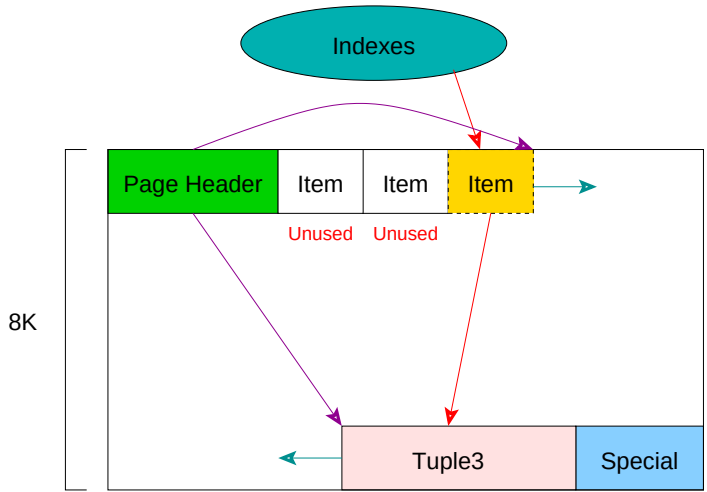
Same as this Slide



Cleanup of Deleted Rows

```
SELECT pg_freespace('mvcc_demo');
 pg_freespace
--------------
 (0,0)

VACUUM mvcc_demo;

SELECT * FROM mvcc_demo_page0
OFFSET 220;
  ctid   |  case  | xmin | xmax | t_ctid
---------+--------+------+------+---------
 (0,221) | Unused |      |      |
 (0,222) | Unused |      |      |
 (0,223) | Normal | 5434 |    0 | (0,223)
```

Same as this Slide



Free Space Map (FSM)

```
SELECT pg_freespace('mvcc_demo');
 pg_freespace
--------------
 (0,192)
```

VACUUM also updates the free space map (FSM), which records pages containing significant free space. This information is used to provide target pages for INSERTs and some UPDATEs (those crossing page boundaries). Single-page cleanup does not update the free space map.

Another Free Space Map Example

```
TRUNCATE mvcc_demo;

VACUUM mvcc_demo;

SELECT pg_freespace('mvcc_demo');
 pg_freespace
--------------
```

Another Free Space Map Example

```
INSERT INTO mvcc_demo VALUES (1);

VACUUM mvcc_demo;

SELECT pg_freespace('mvcc_demo');
 pg_freespace
--------------
 (0,8128)

INSERT INTO mvcc_demo VALUES (2);

VACUUM mvcc_demo;

SELECT pg_freespace('mvcc_demo');
 pg_freespace
--------------
 (0,8064)
```

Another Free Space Map Example

```
DELETE FROM mvcc_demo WHERE val = 2;

VACUUM mvcc_demo;

SELECT pg_freespace('mvcc_demo');
 pg_freespace
--------------
 (0,8128)
```

VACUUM Also Removes End-of-File Pages

```
DELETE FROM mvcc_demo WHERE val = 1;

VACUUM mvcc_demo;

SELECT pg_freespace('mvcc_demo');
 pg_freespace
--------------


SELECT pg_relation_size('mvcc_demo');
 pg_relation_size
------------------
                0
```

VACUUM FULL shrinks the table file to its minimum size, but requires an exclusive table lock.

Optimized Single-Page Cleanup of Old UPDATE Rows

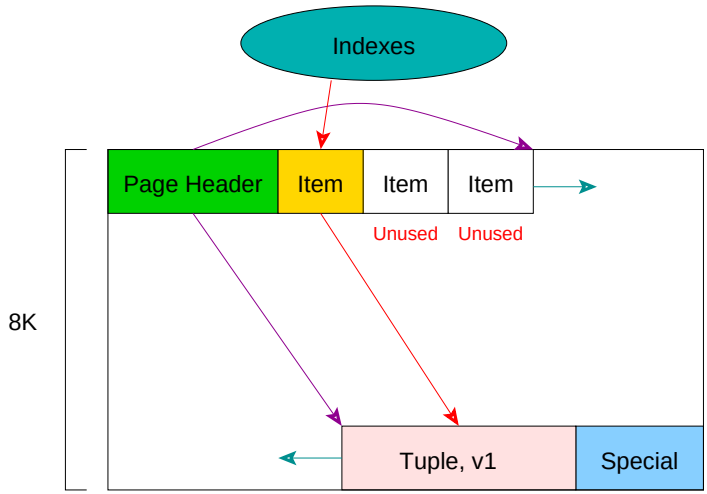
The storage space taken by old UPDATE tuples can be reclaimed just like deleted rows.
However, certain UPDATE rows can even have their items reclaimed, i.e., it is possible to reuse certain old UPDATE items, rather than marking them as “dead” and requiring VACUUM to reclaim them after removing referencing index entries.
Specifically, such item reuse is possible with special HOT update (heap-only tuple) chains, where the chain is on a single heap page and all indexed values in the chain are identical.

Single-Page Cleanup of HOT UPDATE Rows

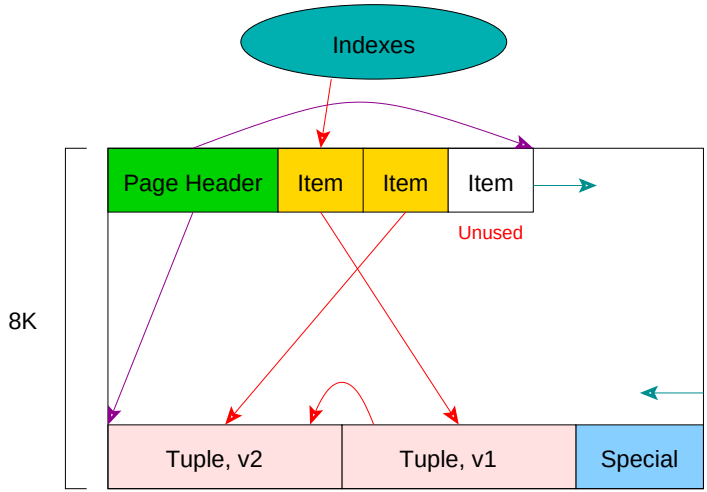
HOT update items can be freed (marked "unused") if they are in the middle of the chain, i.e., not at the beginning or end of the chain. At the head of the chain is a special "Redirect" item pointers that are referenced by indexes; this is possible because all indexed values are identical in a HOT/redirect chain.

Index creation with HOT chains is complex because the chains might contain inconsistent values for the newly indexed columns. This is handled by indexing just the end of the HOT chain and allowing the index to be used only by transactions that start after the index has been created. (Specifically, post-index-creation transactions cannot see the inconsistent HOT chain values due to MVCC visibility rules; they only see the end of the chain.)

Initial Single-Row State

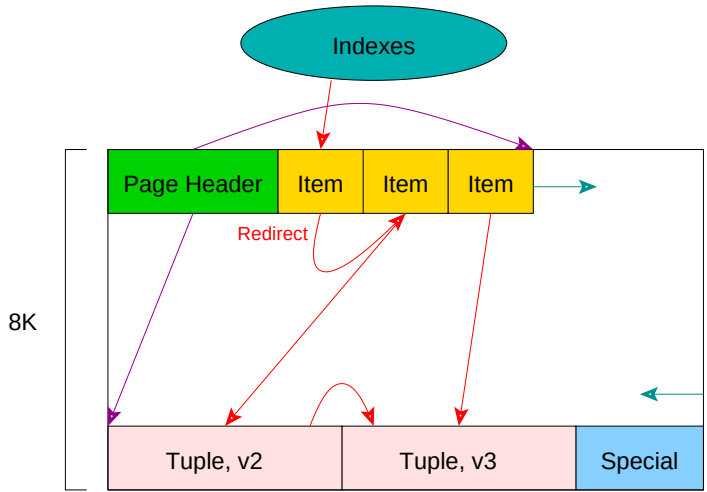


UPDATE Adds a New Row

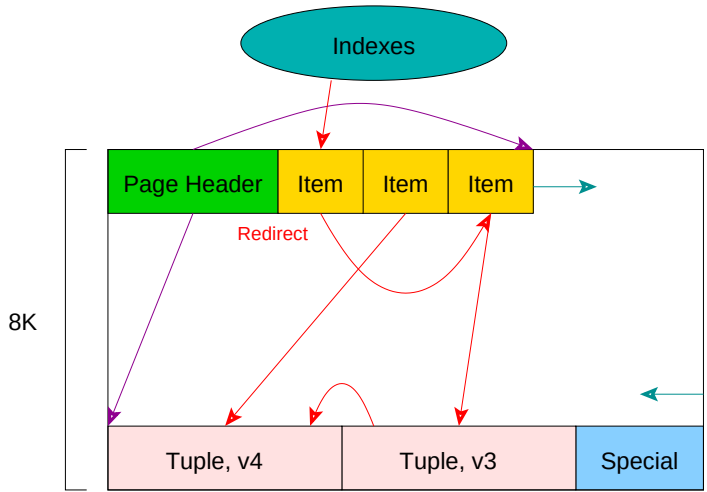


No index entry added because indexes only point to the head of the HOT chain.

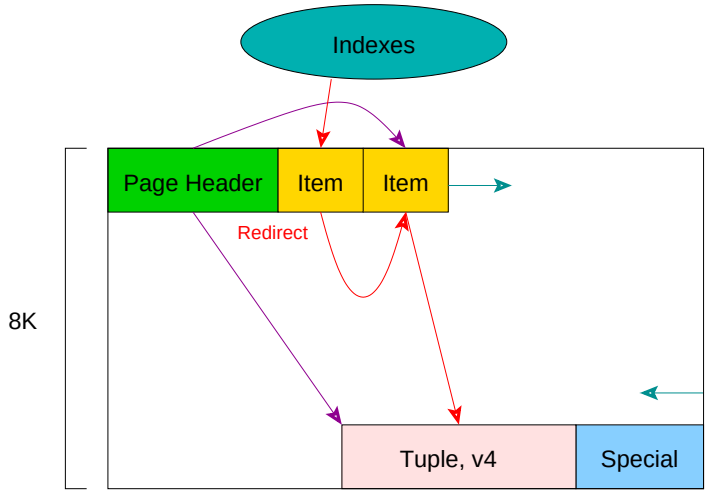
Redirect Allows Indexes To Remain Valid



UPDATE Replaces Another Old Row



All Old UPDATE Row Versions Eventually Removed



This cleanup was performed by another operation on the same page.

Cleanup of Old Updated Rows

```
TRUNCATE mvcc_demo;

INSERT INTO mvcc_demo SELECT 0 FROM generate_series(1, 220);

INSERT INTO mvcc_demo VALUES (1);

SELECT * FROM mvcc_demo_page0
OFFSET 220;
  ctid   |  case  | xmin | xmax | t_ctid
---------+--------+------+------+---------
 (0,221) | Normal | 5437 |    0 | (0,221)
```

Cleanup of Old Updated Rows

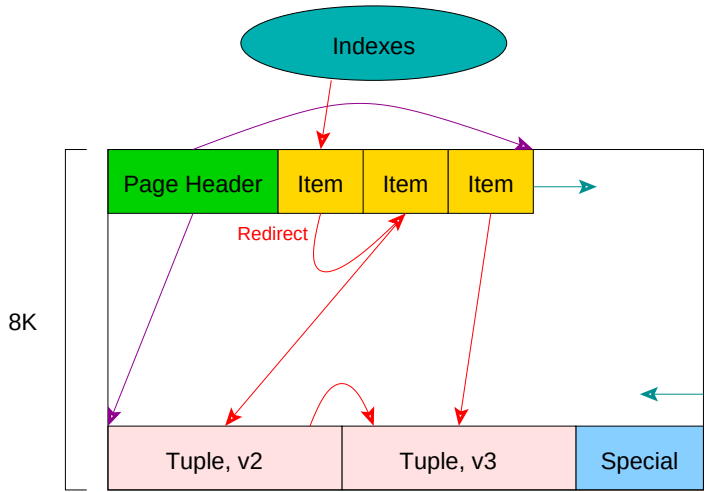
```
UPDATE mvcc_demo SET val = val + 1 WHERE val > 0;

SELECT * FROM mvcc_demo_page0
OFFSET 220;
  ctid   |  case  | xmin | xmax | t_ctid
---------+--------+------+------+---------
 (0,221) | Normal | 5437 | 5438 | (0,222)
 (0,222) | Normal | 5438 |    0 | (0,222)
```

Cleanup of Old Updated Rows

```
UPDATE mvcc_demo SET val = val + 1 WHERE val > 0;

SELECT * FROM mvcc_demo_page0
OFFSET 220;
  ctid   |      case       | xmin | xmax | t_ctid
---------+-----------------+------+------+---------
 (0,221) | Redirect to 222 |      |      |
 (0,222) | Normal          | 5438 | 5439 | (0,223)
 (0,223) | Normal          | 5439 |    0 | (0,223)
```

No index entry added because indexes only point to the head of the HOT chain.

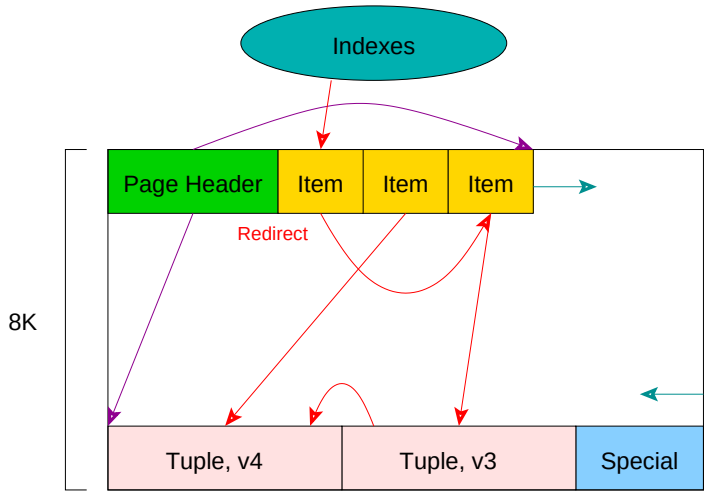
Same as this Slide



Cleanup of Old Updated Rows

```
UPDATE mvcc_demo SET val = val + 1 WHERE val > 0;

SELECT * FROM mvcc_demo_page0
OFFSET 220;
  ctid   |      case       | xmin | xmax | t_ctid
---------+-----------------+------+------+---------
 (0,221) | Redirect to 223 |      |      |
 (0,222) | Normal          | 5440 |    0 | (0,222)
 (0,223) | Normal          | 5439 | 5440 | (0,222)
```

Same as this Slide

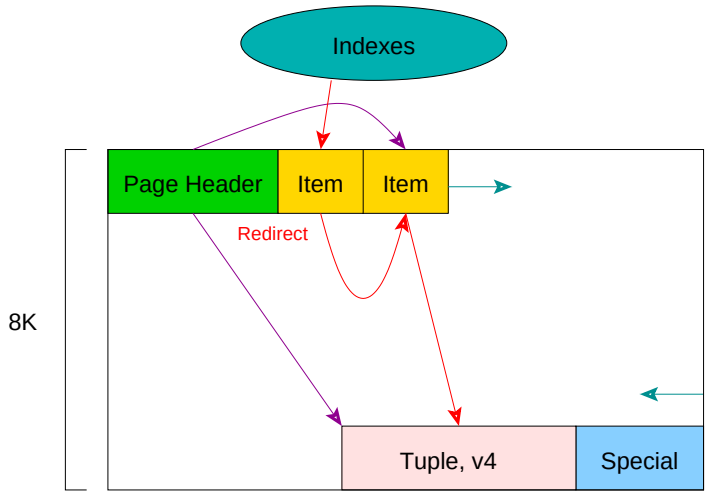


Cleanup of Old Updated Rows

```
-- transaction now committed, HOT chain allows tid
-- to be marked as "Unused"
SELECT * FROM mvcc_demo
OFFSET 1000;
 val
-----


SELECT * FROM mvcc_demo_page0
OFFSET 220;
  ctid   |      case       | xmin | xmax | t_ctid
---------+-----------------+------+------+---------
 (0,221) | Redirect to 222 |      |      |
 (0,222) | Normal          | 5440 |    0 | (0,222)
```

Trailing `Unused` item pointers like `(0,223)` are removed starting in Postgres 15.

Same as this Slide



VACUUM Does Not Remove the Redirect

```
VACUUM mvcc_demo;

SELECT * FROM mvcc_demo_page0
OFFSET 220;
  ctid   |      case       | xmin | xmax | t_ctid
---------+-----------------+------+------+---------
 (0,221) | Redirect to 222 |      |      |
 (0,222) | Normal          | 5440 |    0 | (0,222)
```

Cleanup Using Manual VACUUM

```
TRUNCATE mvcc_demo;

INSERT INTO mvcc_demo VALUES (1);
INSERT INTO mvcc_demo VALUES (2);
INSERT INTO mvcc_demo VALUES (3);

SELECT  ctid, xmin, xmax
FROM mvcc_demo_page0;
 ctid  | xmin | xmax
-------+------+------
 (0,1) | 5442 |    0
 (0,2) | 5443 |    0
 (0,3) | 5444 |    0

DELETE FROM mvcc_demo;
```

Cleanup Using Manual VACUUM

```
SELECT  ctid, xmin, xmax
FROM mvcc_demo_page0;
 ctid  | xmin | xmax
-------+------+------
 (0,1) | 5442 | 5445
 (0,2) | 5443 | 5445
 (0,3) | 5444 | 5445

-- too small to trigger autovacuum
VACUUM mvcc_demo;

SELECT pg_relation_size('mvcc_demo');
 pg_relation_size
------------------
                0
```

The Indexed UPDATE Problem

The updating of any indexed columns prevents the use of “redirect” items because the chain must be usable by all indexes, i.e., a redirect/HOT UPDATE cannot require additional index entries due to an indexed value change.
In such cases, item pointers can only be marked as “dead”, like DELETE does.
No previously shown UPDATE queries modified indexed columns.

Index mvcc_demo Column

```
CREATE INDEX i_mvcc_demo_val on mvcc_demo (val);
```

UPDATE of an Indexed Column

```
TRUNCATE mvcc_demo;

INSERT INTO mvcc_demo SELECT 0 FROM generate_series(1, 220);
INSERT INTO mvcc_demo VALUES (1);

SELECT * FROM mvcc_demo_page0
OFFSET 220;
  ctid   |  case  | xmin | xmax | t_ctid
---------+--------+------+------+---------
 (0,221) | Normal | 5449 |    0 | (0,221)
```

UPDATE of an Indexed Column

```
UPDATE mvcc_demo SET val = val + 1 WHERE val > 0;

SELECT * FROM mvcc_demo_page0
OFFSET 220;
  ctid   |  case  | xmin | xmax | t_ctid
---------+--------+------+------+---------
 (0,221) | Normal | 5449 | 5450 | (0,222)
 (0,222) | Normal | 5450 |    0 | (0,222)

UPDATE mvcc_demo SET val = val + 1 WHERE val > 0;

SELECT * FROM mvcc_demo_page0
OFFSET 220;
  ctid   |  case  | xmin | xmax | t_ctid
---------+--------+------+------+---------
 (0,221) | Dead   |      |      |
 (0,222) | Normal | 5450 | 5451 | (0,223)
 (0,223) | Normal | 5451 |    0 | (0,223)
```

UPDATE of an Indexed Column

```
UPDATE mvcc_demo SET val = val + 1 WHERE val > 0;

SELECT * FROM mvcc_demo_page0
OFFSET 220;
  ctid   |  case  | xmin | xmax | t_ctid
---------+--------+------+------+---------
 (0,221) | Dead   |      |      |
 (0,222) | Dead   |      |      |
 (0,223) | Normal | 5451 | 5452 | (0,224)
 (0,224) | Normal | 5452 |    0 | (0,224)
```

UPDATE of an Indexed Column

```
SELECT * FROM mvcc_demo
OFFSET 1000;
 val
-----
```

```
SELECT * FROM mvcc_demo_page0
OFFSET 220;
  ctid   |  case  | xmin | xmax | t_ctid
---------+--------+------+------+---------
 (0,221) | Dead   |      |      |
 (0,222) | Dead   |      |      |
 (0,223) | Dead   |      |      |
 (0,224) | Normal | 5452 |    0 | (0,224)
```

UPDATE of an Indexed Column

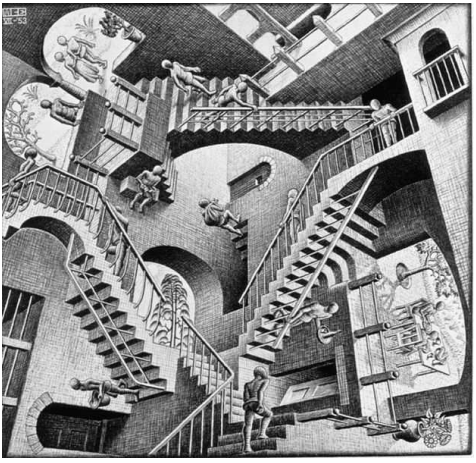
```
VACUUM mvcc_demo;

SELECT * FROM mvcc_demo_page0
OFFSET 220;
  ctid   |  case  | xmin | xmax | t_ctid
---------+--------+------+------+---------
 (0,221) | Unused |      |      |
 (0,222) | Unused |      |      |
 (0,223) | Unused |      |      |
 (0,224) | Normal | 5452 |    0 | (0,224)
```

Cleanup Summary

Cleanup Method	Triggered By	Scope	Reuse Heap Tuples?	Non-HOT Item State	HOT Item State	Clean Indexes?	Update FSM
Single-Page	SELECT, UPDATE, DELETE	single heap page	yes	dead	unused	no	no
VACUUM	autovacuum or manually	all potential heap pages	yes	unused	unused	yes	yes

Cleanup is possible only when there are no active transactions for which the tuples are visible.
HOT items are UPDATE chains that span a single page and contain identical indexed column values.
In normal usage, single-page cleanup performs the majority of the cleanup work, while VACUUM reclaims “dead” item pointers, removes unnecessary index entries, and updates the free space map (FSM).

Conclusion



https://momjian.us/presentations

M.C Escher, Relativity