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.

Creative Commons Attribution License

http://momjian.us/presentations

Last updated: April, 2016

Unmasked: Who Are These People?



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

- 1. Introduction to MVCC
- 2. MVCC Implementation Details
- 3. MVCC Cleanup Requirements and Behavior

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

Cre 40 Exp INSERT

Cre 40 Exp 47 DELETE

Cre 64 Exp 78 old (delete) UPDATE

Cre 78 new (insert)

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 at transaction start in SERIALIZABLE transaction isolation mode. In fact, it is 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

Create-Only

Cre 30 Exp Cre 50 Exp Cre 110

Visible

Invisible

Invisible

Sequential Scan

Snapshot

The highest–numbered committed transaction: 100

Open Transactions: 25, 50, 75

For simplicity, assume all other transactions are committed.

Create & Expire

Exp

Cre 30 Exp 80 Cre 30 Exp 75 Cre 30 Exp 110

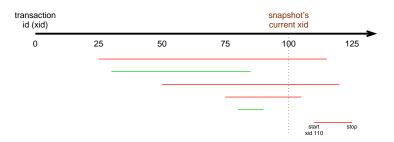
Invisible

Visible

Visible

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 == mv-transaction &&
                                         it was deleted by the current transaction
    Cmax >= my-command)))
                                         but not before this command,
 (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 >= mv-command) ||
                                         but it's not deleted "vet". 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);
CREATE TABLE
DROP VIEW IF EXISTS mvcc demo page0;
DROP VIFW
CREATE EXTENSION pageinspect;
CREATE EXTENSION
CREATE VIEW mvcc demo pageO AS
        SELECT '(0,' || 1p || ')' AS ctid,
                CASE lp flags
                        WHEN O THEN 'Unused'
                        WHEN 1 THEN 'Normal'
                        WHEN 2 THEN 'Redirect to ' || 1p 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 1p;
CREATE VIEW
```

INSERT Using Xmin

```
DELETE FROM mvcc_demo;
DELETE 0
INSERT INTO mvcc_demo VALUES (1);
INSERT 0 1
SELECT xmin, xmax, * FROM mvcc_demo;
xmin | xmax | val
----+-----
5409 | 0 | 1
(1 row)
```

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

Just Like INSERT

Cre 40 **INSERT** Exp

Cre 40 **DELETE** Exp 47

Cre 64 Exp 78

Cre 78

Exp

old (delete)

UPDATE

new (insert)

DELETE Using Xmax

```
DELETE FROM mvcc demo;
DELETE 1
INSERT INTO mvcc demo VALUES (1);
INSERT 0 1
SELECT xmin, xmax, * FROM mvcc demo;
xmin | xmax | val
____+
5411 | 0 | 1
(1 row)
BEGIN WORK;
BEGIN
DELETE FROM mvcc demo;
DELETE 1
```

DELETE Using Xmax

```
SELECT xmin, xmax, * FROM mvcc demo;
 xmin | xmax | val
----+----
(0 rows)
     SELECT xmin, xmax, * FROM mvcc demo;
        xmin | xmax | val
       ----+----
        5411 | 5412 | 1
       (1 row)
SELECT txid current();
 txid current
        5412
(1 row)
COMMIT WORK;
COMMIT
```

Just Like DELETE

Cre 40 Exp INSERT

Cre 40 Exp 47 DELETE

Cre 64 Exp 78 old (delete) UPDATE

Cre 78 new (insert)

UPDATE Using Xmin and Xmax

```
DELETE FROM mvcc demo;
DELETE 0
INSERT INTO mvcc demo VALUES (1);
INSERT 0 1
SELECT xmin, xmax, * FROM mvcc demo;
xmin | xmax | val
----+----
5413 | 0 | 1
(1 row)
BEGIN WORK;
BEGIN
UPDATE mvcc demo SET val = 2;
UPDATE 1
```

UPDATE Using Xmin and Xmax

```
SELECT xmin, xmax, * FROM mvcc demo;
xmin | xmax | val
5414 | 0 | 2
(1 row)
       SELECT xmin, xmax, * FROM mvcc demo;
        xmin | xmax | val
       ____+
        5413 | 5414 | 1
       (1 row)
COMMIT WORK;
COMMIT
```

Just Like UPDATE

Cre 40 Exp INSERT

Cre 40 Exp 47 DELETE

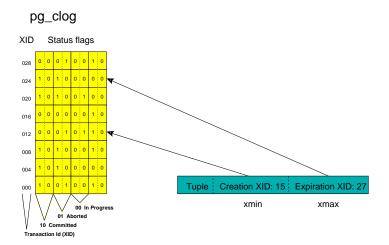
Cre 64 Exp 78 old (delete) UPDATE

Cre 78 new (insert)

Aborted Transaction IDs Remain

```
DELETE FROM mvcc demo;
DFLFTF 1
INSERT INTO mvcc demo VALUES (1);
INSFRT 0 1
BEGIN WORK;
BFGIN
DELETE FROM mvcc demo;
DFLFTF 1
ROLLBACK WORK;
ROLLBACK
SELECT xmin, xmax, * FROM mvcc demo;
 xmin | xmax | val
----+----
5415 | 5416 | 1
(1 row)
```

Aborted IDs Can Remain Because Transaction Status Is Recorded Centrally



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

Row Locks Using Xmax

```
DELETE FROM mvcc demo;
DELETE 1
INSERT INTO mvcc demo VALUES (1);
INSFRT 0 1
BEGIN WORK;
BEGIN
SELECT xmin, xmax, * FROM mvcc demo;
xmin | xmax | val
5416 | 0 | 1
(1 row)
SELECT xmin, xmax, * FROM mvcc demo FOR UPDATE;
 xmin | xmax | val
5416 | 0 | 1
(1 row)
```

Row Locks Using Xmax

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;
DELETE 1
BEGIN WORK;
BFGIN
INSERT INTO mvcc demo VALUES (1);
INSERT 0 1
INSERT INTO mvcc demo VALUES (2);
INSERT 0 1
INSERT INTO mvcc demo VALUES (3);
INSERT 0 1
```

INSERT Using Cmin

DELETE Using Cmin

```
DELETE FROM mvcc demo;
DELETE 3
BEGIN WORK;
BFGIN
INSERT INTO mvcc demo VALUES (1);
INSERT 0 1
INSERT INTO mvcc demo VALUES (2);
INSERT 0 1
INSERT INTO mvcc demo VALUES (3);
INSERT 0 1
```

DELETE Using Cmin

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

DELETE Using Cmin

```
DELETE FROM mvcc demo;
DFLFTF 3
SELECT xmin, cmin, xmax, * FROM mvcc demo;
 xmin | cmin | xmax | val
   ---+----
(0 rows)
FETCH ALL FROM c mvcc demo;
 xmin | xmax | cmax | val
   ___+__

    5421 | 5421 | 0 | 1

    5421 | 5421 | 1 | 2

    5421 | 5421 | 2 | 3

(3 rows)
COMMIT WORK;
COMMIT
```

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;
DFLFTF 0
BEGIN WORK;
BFGIN
INSERT INTO mvcc demo VALUES (1);
INSERT 0 1
INSERT INTO mvcc demo VALUES (2);
INSERT 0 1
INSERT INTO mvcc demo VALUES (3);
INSERT 0 1
SELECT xmin, cmin, xmax, * FROM mvcc demo;
 xmin | cmin | xmax | val

    5422 |
    0 |
    0 |
    1

    5422 |
    1 |
    0 |
    2

    5422 |
    2 |
    0 |
    3

(3 rows)
DECLARE c mvcc demo CURSOR FOR
SELECT xmin, xmax, cmax, * FROM mvcc demo;
```

UPDATE Using Cmin

```
UPDATE mvcc demo SET val = val * 10;
UPDATE 3
SELECT xmin, cmin, xmax, * FROM mvcc demo;
xmin | cmin | xmax | val
5422 | 3 |
                0 | 10
5422 | 3 | 0 | 20
5422 | 3 | 0 |
                   30
(3 rows)
FETCH ALL FROM c mvcc demo;
xmin | xmax | cmax | val
5422 | 5422 | 0 |
5422 | 5422 | 1 | 2
5422 | 5422 | 2 |
(3 rows)
COMMIT WORK;
COMMIT
```

Modifying Rows From Different Transactions

```
DELETE FROM mvcc demo;
DFLFTF 3
INSERT INTO mvcc demo VALUES (1);
INSFRT 0 1
SELECT xmin, xmax, * FROM mvcc demo;
 xmin | xmax | val
----+----
5424 | 0 | 1
(1 row)
BEGIN WORK:
BEGIN
INSERT INTO mvcc demo VALUES (2);
INSERT 0 1
INSERT INTO mvcc demo VALUES (3);
INSERT 0 1
INSERT INTO mvcc demo VALUES (4);
INSERT 0 1
```

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
(4 rows)
UPDATE mvcc_demo SET val = val * 10;
UPDATE 4
```

Modifying Rows From Different Transactions

```
SELECT xmin, cmin, xmax, * FROM mvcc demo;
 xmin | cmin | xmax | val
 5425 |
                              10

    5425 |
    3 |
    0 |
    20

    5425 |
    3 |
    0 |
    30

    5425 |
    3 |
    0 |
    40

(4 rows)
         SELECT xmin, xmax, cmax, * FROM mvcc demo;
           xmin | xmax | cmax | val
             ____+_
           5424 | 5425 | 3 | 1
          (1 row)
COMMIT WORK;
COMMIT
```

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.

```
-- use TRUNCATE to remove even invisible rows
TRUNCATE mvcc demo;
TRUNCATE TABLE
BEGIN WORK;
BEGIN
DELETE FROM mvcc demo;
DFLFTF 0
DELETE FROM mvcc demo;
DFLFTF 0
DELETE FROM mvcc demo;
DELETE 0
INSERT INTO mvcc demo VALUES (1);
INSERT 0 1
INSERT INTO mvcc demo VALUES (2);
INSERT 0 1
INSERT INTO mvcc demo VALUES (3);
INSERT 0 1
```

```
SELECT xmin, cmin, xmax, * FROM mvcc demo;
 xmin | cmin | xmax | val
5427 | 3 | 0 |
5427 | 4 | 0 | 2
5427 | 5 | 0 | 3
(3 rows)
DECLARE c mvcc demo CURSOR FOR
SELECT xmin, xmax, cmax, * FROM mvcc demo;
DECLARE CURSOR
UPDATE mvcc demo SET val = val * 10;
UPDATE 3
```

```
SELECT xmin, cmin, xmax, * FROM mvcc demo;
 xmin | cmin | xmax | val
 5427 | 6 |
                          10
 5427 | 6 | 0 | 20
5427 | 6 | 0 | 30
(3 rows)
FETCH ALL FROM c mvcc demo;
 xmin | xmax | cmax | val

    5427 | 5427 | 0 | 1

    5427 | 5427 | 1 | 2

 5427 | 5427 | 2 |
(3 rows)
```

```
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 |

        5427
        5427
        1
        t

        5427
        5427
        2
        t

        5427
        0
        6
        f

        5427
        0
        6
        f

        5427
        0
        6
        f

        5427
        0
        6
        f

(6 rows)
COMMIT WORK:
COMMIT
```

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

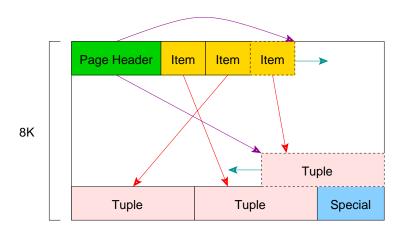
Cleanup can also be initiated manually by VACUUM.

Aspects of Cleanup

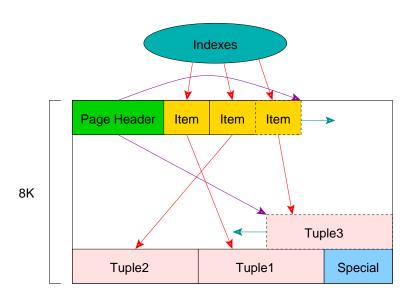
Cleanup involves recycling space taken by several entities:

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

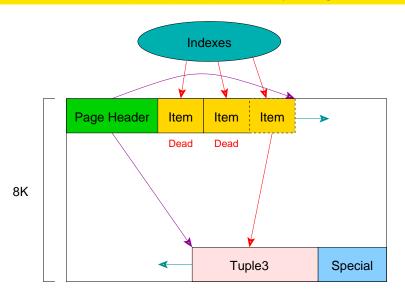
Internal Heap Page



Indexes Point to Items, Not Tuples

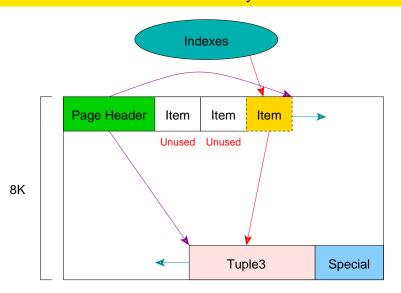


Heap Tuple Space Recycling



Indexes prevent item pointers from being recycled.

VACUUM Later Recycle Items



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

```
TRUNCATE mvcc demo;
TRUNCATE TABLE
-- force page to < 10% empty
INSERT INTO mvcc demo SELECT O FROM generate series(1, 240);
INSFRT 0 240
-- compute free space percentage
SELECT (100 * (upper - lower) / pagesize::float8)::integer AS free
FROM page header(get raw page('mvcc demo', 0));
 free pct
        6
(1 row)
INSERT INTO mvcc demo VALUES (1);
INSFRT 0 1
```

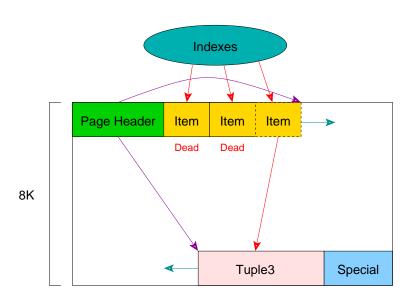
```
SELECT * FROM mvcc demo page0
OFFSET 240;
 ctid | case | xmin | xmax | t ctid
 (0,241) | Normal | 5430 | 0 | (0,241)
(1 row)
DELETE FROM mvcc demo WHERE val > 0;
DFIFTE 1
INSERT INTO mvcc demo VALUES (2);
INSFRT 0 1
SELECT * FROM mvcc demo page0
OFFSET 240:
 ctid | case | xmin | xmax | t ctid
 (0,241) | Normal | 5430 | 5431 | (0,241)
 (0,242) | Normal | 5432 | 0 | (0,242)
(2 rows)
```

```
DELETE FROM mvcc demo WHERE val > 0;
DFLFTF 1
INSERT INTO mvcc demo VALUES (3);
INSERT 0 1
SELECT * FROM mvcc demo page0
OFFSET 240:
 ctid | case | xmin | xmax | t ctid
-----+-----+-----+-----+-----
(0,241) | Dead | |
(0,242) | Normal | 5432 | 5433 | (0,242)
(0,243) | Normal | 5434 | 0 | (0,243)
(3 rows)
```

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.

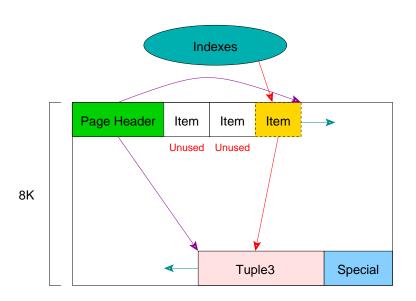
```
-- force single-page cleanup via SELECT
SELECT * FROM mvcc demo
OFFSET 1000;
val
(0 rows)
SELECT * FROM mvcc demo page0
OFFSET 240;
      | case | xmin | xmax | t ctid
 ctid
 (0,241) | Dead |
 (0,242) | Dead |
 (0,243) | Normal | 5434 | 0 | (0,243)
(3 rows)
```

Same as this Slide



```
SELECT pg freespace('mvcc demo');
 pg freespace
 (0,0)
(1 row)
VACUUM mvcc demo;
VACUUM
SELECT * FROM mvcc demo page0
OFFSET 240:
 ctid | case | xmin | xmax | t ctid
 (0,241) | Unused |
 (0,242) | Unused |
 (0,243) | Normal | 5434 | 0 |
                                 (0,243)
(3 rows)
```

Same as this Slide



Free Space Map (FSM)

```
SELECT pg_freespace('mvcc_demo');
  pg_freespace
-----(0,416)
(1 row)
```

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

Another Free Space Map Example

```
INSERT INTO mvcc demo VALUES (1);
INSERT 0 1
VACUUM mvcc demo;
VACUUM
SELECT pg freespace('mvcc demo');
pg_freespace
 (0,8128)
(1 row)
INSERT INTO mvcc demo VALUES (2);
INSERT 0 1
VACUUM mvcc demo;
VACUUM
SELECT pg freespace('mvcc demo');
 pg freespace
 (0,8096)
(1 row)
```

Another Free Space Map Example

VACUUM Also Removes End-of-File Pages

```
DELETE FROM mvcc demo WHERE val = 1;
DELETE 1
VACUUM mvcc demo;
VACUUM
SELECT pg freespace('mvcc demo');
 pg freespace
(0 rows)
SELECT pg relation size('mvcc demo');
 pg relation size
(1 row)
```

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

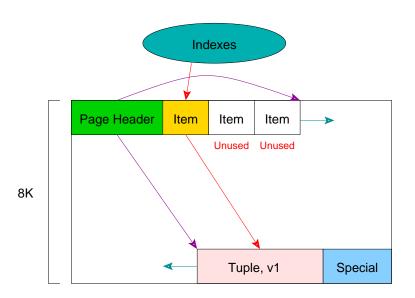
Optimized Single-Page Cleanup of Old UPDATE Rows

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.

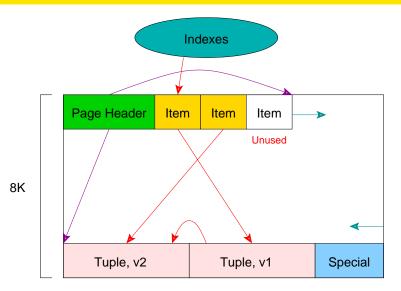
Single-Page Cleanup of HOT UPDATE Rows

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

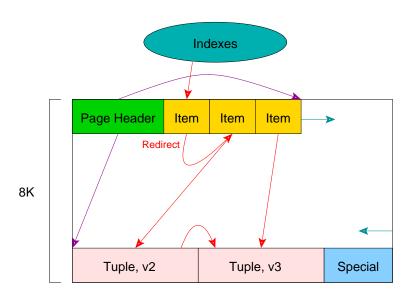


UPDATE Adds a New Row

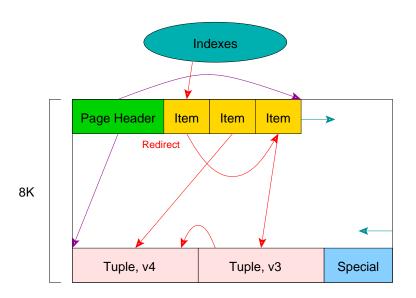


No index entry added because indexes only point to the head of the HOT chain.

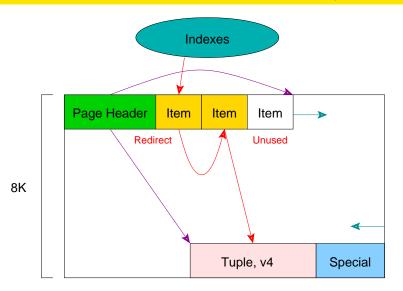
Redirect Allows Indexes To Remain Valid



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

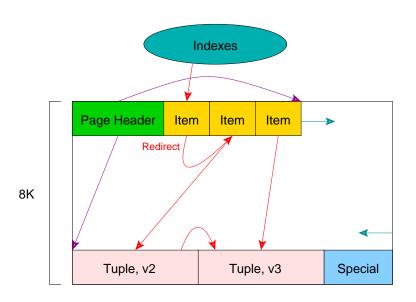
```
TRUNCATE mvcc demo;
TRUNCATE TABLE
INSERT INTO mvcc demo SELECT O FROM generate series(1, 240);
INSFRT 0 240
INSERT INTO mvcc demo VALUES (1);
INSFRT 0 1
SELECT * FROM mvcc demo page0
OFFSET 240:
 ctid | case | xmin | xmax | t ctid
-----+----+-----+-----+-----
 (0,241) | Normal | 5437 | 0 | (0,241)
(1 row)
```

Cleanup of Old Updated Rows

Cleanup of Old Updated Rows

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

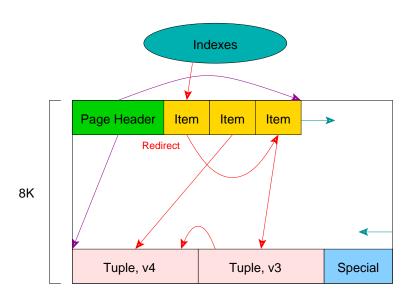
Same as this Slide



Cleanup of Old Updated Rows

```
UPDATE mvcc demo SET val = val + 1 WHERE val > 0;
UPDATE 1
SELECT * FROM mvcc demo page0
OFFSET 240:
 ctid | case | xmin | xmax | t ctid
(0,243) | Normal | 5439 | 5440 | (0,242)
(3 rows)
```

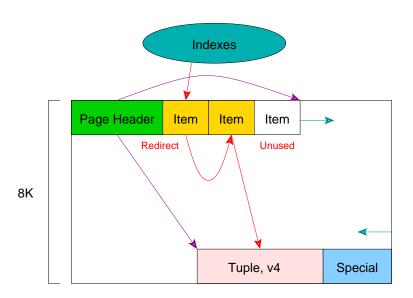
Same as this Slide



Cleanup of Old Updated Rows

```
-- transaction now committed, HOT chain allows tid
-- to be marked as ''Unused''
SELECT * FROM mvcc demo
OFFSET 1000;
val
(0 rows)
SELECT * FROM mvcc demo page0
OFFSET 240;
 ctid | case | xmin | xmax | t_ctid
(0,243) | Unused
(3 rows)
```

Same as this Slide



VACUUM Does Not Remove the Redirect

```
VACUUM mvcc demo;
VACUUM
SELECT * FROM mvcc demo page0
OFFSET 240;
 ctid | case | xmin | xmax | t ctid
(0,243) | Unused
(3 rows)
```

Cleanup Using Manual VACUUM

```
TRUNCATE mvcc demo;
TRUNCATE TABLE
INSERT INTO mvcc demo VALUES (1);
INSFRT 0 1
INSERT INTO mvcc demo VALUES (2);
INSERT 0 1
INSERT INTO mvcc demo VALUES (3);
INSERT 0 1
SELECT ctid, xmin, xmax
FROM mvcc demo page0;
 ctid | xmin | xmax
 (0,1) \mid 5442 \mid
 (0,2) | 5443 |
 (0,3) \mid 5444 \mid
(3 rows)
DELETE FROM mvcc demo;
DELETE 3
```

Cleanup Using Manual VACUUM

```
SELECT ctid, xmin, xmax
FROM mvcc demo page0;
 ctid | xmin | xmax
 (0,1) \mid 5442 \mid 5445
 (0,2) \mid 5443 \mid 5445
 (0,3) \mid 5444 \mid 5445
(3 rows)
-- too small to trigger autovacuum
VACUUM mvcc demo;
VACUUM
SELECT pg relation size('mvcc demo');
 pg relation size
(1 row)
```

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); CREATE INDEX
```

```
TRUNCATE mvcc demo;
TRUNCATE TABLE
INSERT INTO mvcc demo SELECT O FROM generate series(1, 240);
INSFRT 0 240
INSERT INTO mvcc demo VALUES (1);
INSFRT 0 1
SELECT * FROM mvcc demo page0
OFFSET 240:
 ctid | case | xmin | xmax | t ctid
-----+----+----+-----+----
(0,241) | Normal | 5449 | 0 | (0,241)
(1 row)
```

```
UPDATE mvcc demo SET val = val + 1 WHERE val > 0;
UPDATE 1
SELECT * FROM mvcc demo page0
OFFSET 240:
       | case | xmin | xmax | t ctid
 (0,241) | Normal | 5449 | 5450 | (0,242)
 (0,242) | Normal | 5450 | 0 | (0,242)
(2 rows)
UPDATE mvcc demo SET val = val + 1 WHERE val > 0;
UPDATE 1
SELECT * FROM mvcc demo page0
OFFSET 240:
 ctid
       case | xmin | xmax | t ctid
 (0,241) | Dead |
 (0,242) | Normal | 5450 | 5451 | (0,243)
 (0,243) | Normal | 5451 | 0 | (0,243)
(3 rows)
```

```
UPDATE mvcc demo SET val = val + 1 WHERE val > 0;
UPDATE 1
SELECT * FROM mvcc demo page0
OFFSET 240;
 ctid
         | case | xmin | xmax | t ctid
 (0,241) | Dead
 (0,242) | Dead
 (0,243) | Normal | 5451 | 5452 | (0,244)
 (0,244) | Normal | 5452 | 0 | (0,244)
(4 rows)
```

```
SELECT * FROM mvcc demo
OFFSET 1000;
val
(0 rows)
SELECT * FROM mvcc demo page0
OFFSET 240;
 ctid
         | case | xmin | xmax | t ctid
 (0,241) | Dead
 (0,242) | Dead
 (0,243) | Dead
 (0,244) | Normal | 5452 | 0 |
                                  (0,244)
(4 rows)
```

```
VACUUM mvcc demo;
VACUUM
SELECT * FROM mvcc demo page0
OFFSET 240;
 ctid
         | case | xmin | xmax | t ctid
 (0,241) | Unused
 (0,242) | Unused
 (0,243) | Unused
 (0,244) | Normal | 5452 | 0 |
                                 (0,244)
(4 rows)
```

Cleanup Summary

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

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



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

http://momjian.us/presentations