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: August, 2019
Unmasked: Who Are These People?

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

<table>
<thead>
<tr>
<th>Cre</th>
<th>Exp</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>40</td>
<td></td>
<td>INSERT</td>
</tr>
<tr>
<td></td>
<td></td>
<td>DELETE</td>
</tr>
<tr>
<td>40</td>
<td>47</td>
<td>UPDATE</td>
</tr>
<tr>
<td>64</td>
<td>78</td>
<td>old (delete)</td>
</tr>
<tr>
<td>78</td>
<td></td>
<td>new (insert)</td>
</tr>
</tbody>
</table>
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

<table>
<thead>
<tr>
<th>Cre</th>
<th>Exp</th>
</tr>
</thead>
<tbody>
<tr>
<td>30</td>
<td></td>
</tr>
<tr>
<td>50</td>
<td></td>
</tr>
<tr>
<td>110</td>
<td></td>
</tr>
</tbody>
</table>

Create & Expire

<table>
<thead>
<tr>
<th>Cre</th>
<th>Exp</th>
</tr>
</thead>
<tbody>
<tr>
<td>30</td>
<td>80</td>
</tr>
<tr>
<td>30</td>
<td>75</td>
</tr>
<tr>
<td>30</td>
<td>110</td>
</tr>
</tbody>
</table>

Visible

Invisible

Sequential Scan

**Snapshot**

The highest–numbered committed transaction: 100

Open Transactions: 25, 50, 75

For simplicity, assume all other transactions are committed.

Internally, the creation xid is stored in the system column ’xmin’, and expire in ’xmax’.
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`:

```c
((Xmin == my-transaction &&
  Cmin < my-command &&
  (Xmax is null ||
   (Xmax == my-transaction &&
    Cmax >= my-command)))
 ||
 (Xmin is committed &&
  (Xmax is null ||
   (Xmax == my-transaction &&
    Cmax >= my-command) ||
   (Xmax != my-transaction &&
    Xmax is not committed))))
```

inserted by the current transaction
before this command, and
the row has not been deleted, or
it was deleted by the current transaction
but not before this command,
or
the row was inserted by a committed transaction, and
the row has not been deleted, or
the row is being deleted by this transaction
but it's not deleted "yet", or
the row was deleted by another transaction
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.
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.
CREATE TABLE mvcc_demo (val INTEGER);

DROP VIEW IF EXISTS mvcc_demo_page0;

CREATE EXTENSION pageinspect;

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;
DELETE FROM mvcc_demo;

INSERT INTO mvcc_demo VALUES (1);

SELECT xmin, xmax, * FROM mvcc_demo;

<table>
<thead>
<tr>
<th>xmin</th>
<th>xmax</th>
<th>val</th>
</tr>
</thead>
<tbody>
<tr>
<td>5409</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Cre</th>
<th>Exp</th>
</tr>
</thead>
<tbody>
<tr>
<td>40</td>
<td></td>
</tr>
</tbody>
</table>

INSERT

<table>
<thead>
<tr>
<th>Cre</th>
<th>Exp</th>
</tr>
</thead>
<tbody>
<tr>
<td>40</td>
<td>47</td>
</tr>
</tbody>
</table>

DELETE

<table>
<thead>
<tr>
<th>Cre</th>
<th>Exp</th>
</tr>
</thead>
<tbody>
<tr>
<td>64</td>
<td>78</td>
</tr>
</tbody>
</table>

old (delete)

<table>
<thead>
<tr>
<th>Cre</th>
<th>Exp</th>
</tr>
</thead>
<tbody>
<tr>
<td>78</td>
<td></td>
</tr>
</tbody>
</table>

UPDATE

new (insert)
DELETE FROM mvcc_demo;

INSERT INTO mvcc_demo VALUES (1);

SELECT xmin, xmax, * FROM mvcc_demo;

<table>
<thead>
<tr>
<th>xmin</th>
<th>xmax</th>
<th>val</th>
</tr>
</thead>
<tbody>
<tr>
<td>5411</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

BEGIN WORK;

DELETE FROM mvcc_demo;
DELETE Using Xmax

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

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

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

```
COMMIT WORK;
```
### Just Like DELETE

<table>
<thead>
<tr>
<th>Cre</th>
<th>Exp</th>
</tr>
</thead>
<tbody>
<tr>
<td>40</td>
<td></td>
</tr>
<tr>
<td>40</td>
<td>47</td>
</tr>
<tr>
<td>64</td>
<td>78</td>
</tr>
<tr>
<td>78</td>
<td></td>
</tr>
</tbody>
</table>

**INSERT**

**DELETE**

old (delete)

**UPDATE**

new (insert)
UPDATE Using Xmin and Xmax

DELETE FROM mvcc_demo;

INSERT INTO mvcc_demo VALUES (1);

SELECT xmin, xmax, * FROM mvcc_demo;

<table>
<thead>
<tr>
<th>xmin</th>
<th>xmax</th>
<th>val</th>
</tr>
</thead>
<tbody>
<tr>
<td>5413</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

BEGIN WORK;

UPDATE mvcc_demo SET val = 2;
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

<table>
<thead>
<tr>
<th>Cre</th>
<th>Exp</th>
<th>Old (delete)</th>
<th>New (insert)</th>
</tr>
</thead>
<tbody>
<tr>
<td>40</td>
<td>40</td>
<td>INSERT</td>
<td></td>
</tr>
<tr>
<td>47</td>
<td>47</td>
<td></td>
<td>DELETE</td>
</tr>
<tr>
<td>64</td>
<td>78</td>
<td>UPDATE</td>
<td></td>
</tr>
<tr>
<td>78</td>
<td>78</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
DELETE FROM mvcc_demo;

INSERT INTO mvcc_demo VALUES (1);

BEGIN WORK;

DELETE FROM mvcc_demo;

ROLLBACK WORK;

SELECT xmin, xmax, * FROM mvcc_demo;

<table>
<thead>
<tr>
<th>xmin</th>
<th>xmax</th>
<th>val</th>
</tr>
</thead>
<tbody>
<tr>
<td>5415</td>
<td>5416</td>
<td>1</td>
</tr>
</tbody>
</table>
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 necessary to revisit each row to undo the transaction.
DELETE FROM mvcc_demo;

INSERT INTO mvcc_demo VALUES (1);

BEGIN WORK;

SELECT xmin, xmax, * FROM mvcc_demo;

<table>
<thead>
<tr>
<th>xmin</th>
<th>xmax</th>
<th>val</th>
</tr>
</thead>
<tbody>
<tr>
<td>5416</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

SELECT xmin, xmax, * FROM mvcc_demo FOR UPDATE;

<table>
<thead>
<tr>
<th>xmin</th>
<th>xmax</th>
<th>val</th>
</tr>
</thead>
<tbody>
<tr>
<td>5416</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>
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 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.
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;

<table>
<thead>
<tr>
<th>xmin</th>
<th>cmin</th>
<th>xmax</th>
<th>val</th>
</tr>
</thead>
<tbody>
<tr>
<td>5419</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>5419</td>
<td>1</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>5419</td>
<td>2</td>
<td>0</td>
<td>3</td>
</tr>
</tbody>
</table>

COMMIT WORK;
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.
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;

<table>
<thead>
<tr>
<th>xmin</th>
<th>cmin</th>
<th>xmax</th>
<th>val</th>
</tr>
</thead>
<tbody>
<tr>
<td>5422</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>5422</td>
<td>1</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>5422</td>
<td>2</td>
<td>0</td>
<td>3</td>
</tr>
</tbody>
</table>

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;

<table>
<thead>
<tr>
<th>xmin</th>
<th>cmin</th>
<th>xmax</th>
<th>val</th>
</tr>
</thead>
<tbody>
<tr>
<td>5422</td>
<td>3</td>
<td>0</td>
<td>10</td>
</tr>
<tr>
<td>5422</td>
<td>3</td>
<td>0</td>
<td>20</td>
</tr>
<tr>
<td>5422</td>
<td>3</td>
<td>0</td>
<td>30</td>
</tr>
</tbody>
</table>

FETCH ALL FROM c_mvcc_demo;

<table>
<thead>
<tr>
<th>xmin</th>
<th>xmax</th>
<th>cmax</th>
<th>val</th>
</tr>
</thead>
<tbody>
<tr>
<td>5422</td>
<td>5422</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>5422</td>
<td>5422</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>5422</td>
<td>5422</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>

COMMIT WORK;
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);
SELECT xmin, cmin, xmax, * FROM mvcc_demo;

<table>
<thead>
<tr>
<th>xmin</th>
<th>cmin</th>
<th>xmax</th>
<th>val</th>
</tr>
</thead>
<tbody>
<tr>
<td>5424</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>5425</td>
<td>0</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>5425</td>
<td>1</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>5425</td>
<td>2</td>
<td>0</td>
<td>4</td>
</tr>
</tbody>
</table>

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;
```
Because \textit{cmin} and \textit{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);
SELECT xmin, cmin, xmax, * FROM mvcc_demo;

<table>
<thead>
<tr>
<th>xmin</th>
<th>cmin</th>
<th>xmax</th>
<th>val</th>
</tr>
</thead>
<tbody>
<tr>
<td>5427</td>
<td>3</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>5427</td>
<td>4</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>5427</td>
<td>5</td>
<td>0</td>
<td>3</td>
</tr>
</tbody>
</table>

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;
```

<table>
<thead>
<tr>
<th>xmin</th>
<th>cmin</th>
<th>xmax</th>
<th>val</th>
</tr>
</thead>
<tbody>
<tr>
<td>5427</td>
<td>6</td>
<td>0</td>
<td>10</td>
</tr>
<tr>
<td>5427</td>
<td>6</td>
<td>0</td>
<td>20</td>
</tr>
<tr>
<td>5427</td>
<td>6</td>
<td>0</td>
<td>30</td>
</tr>
</tbody>
</table>

```
FETCH ALL FROM c_mvcc_demo;
```

<table>
<thead>
<tr>
<th>xmin</th>
<th>xmax</th>
<th>cmax</th>
<th>val</th>
</tr>
</thead>
<tbody>
<tr>
<td>5427</td>
<td>5427</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>5427</td>
<td>5427</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>5427</td>
<td>5427</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>
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;

<table>
<thead>
<tr>
<th>xmin</th>
<th>xmax</th>
<th>cmin_cmax</th>
<th>is_combocid</th>
</tr>
</thead>
<tbody>
<tr>
<td>5427</td>
<td>5427</td>
<td>0</td>
<td>t</td>
</tr>
<tr>
<td>5427</td>
<td>5427</td>
<td>1</td>
<td>t</td>
</tr>
<tr>
<td>5427</td>
<td>5427</td>
<td>2</td>
<td>t</td>
</tr>
<tr>
<td>5427</td>
<td>0</td>
<td>6</td>
<td>f</td>
</tr>
<tr>
<td>5427</td>
<td>0</td>
<td>6</td>
<td>f</td>
</tr>
<tr>
<td>5427</td>
<td>0</td>
<td>6</td>
<td>f</td>
</tr>
</tbody>
</table>

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

Page Header Item Item Item

Tuple

Tuple

Tuple

Special

8K
Indexes Point to Items, Not Tuples
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, 240);

-- 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
----------
   6

INSERT INTO mvcc_demo VALUES (1);
Cleanup of Deleted Rows

```
SELECT * FROM mvcc_demo_page0
OFFSET 240;
  ctid | case | xmin | xmax | t_ctid
---------+--------+------+------+---------
(0,241) | Normal | 5430 | 0 | (0,241)

DELETE FROM mvcc_demo WHERE val > 0;

INSERT INTO mvcc_demo VALUES (2);

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)
```
DELETE FROM mvcc_demo WHERE val > 0;

INSERT INTO mvcc_demo VALUES (3);

SELECT * FROM mvcc_demo_page0 OFFSET 240;

<table>
<thead>
<tr>
<th>ctid</th>
<th>case</th>
<th>xmin</th>
<th>xmax</th>
<th>t_c tid</th>
</tr>
</thead>
<tbody>
<tr>
<td>(0,241)</td>
<td>Dead</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(0,242)</td>
<td>Normal</td>
<td>5432</td>
<td>5433</td>
<td>(0,242)</td>
</tr>
<tr>
<td>(0,243)</td>
<td>Normal</td>
<td>5434</td>
<td>0</td>
<td>(0,243)</td>
</tr>
</tbody>
</table>

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 240;
  ctid | case | xmin | xmax | t_ctid
  --------+--------+------+------+---------
  (0,241) | Dead | | | |
  (0,242) | Dead | | | |
  (0,243) | Normal | 5434 | 0 | (0,243)
```
Same as this Slide
Cleanup of Deleted Rows

SELECT pg_freespace('mvcc_demo');

VACUUM mvcc_demo;

SELECT * FROM mvcc_demo_page0
OFFSET 240;

<table>
<thead>
<tr>
<th>ctid</th>
<th>case</th>
<th>xmin</th>
<th>xmax</th>
<th>t_ctid</th>
</tr>
</thead>
<tbody>
<tr>
<td>(0,241)</td>
<td>Unused</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(0,242)</td>
<td>Unused</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(0,243)</td>
<td>Normal</td>
<td>5434</td>
<td>0</td>
<td>(0,243)</td>
</tr>
</tbody>
</table>
Same as this Slide

Indexes

Page Header  Item  Item  Item

Unused  Unused

8K

Tuple3  Special
SELECT pg_freespace('mvcc_demo');

pg_freespace
--------------
(0,416)

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
--------------
60 / 90
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
--------------
   (0,8096)
```
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.
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.
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

Indexes

Page Header | Item | Item | Item

Unused | Unused

Tuple, v1 | Special

8K
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

[Indexes]

[Page Header] [Item] [Item] [Item]

[Redirect]

[8K]

[Tuple, v4] [Tuple, v3] Special
All Old UPDATE Row Versions Eventually Removed

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

INSERT INTO mvcc_demo SELECT 0 FROM generate_series(1, 240);

INSERT INTO mvcc_demo VALUES (1);

SELECT * FROM mvcc_demo_page0
OFFSET 240;

+----------------+--------+-------+-------+--------+
| ctid | case   | xmin  | xmax  | t_c tid|
|-------+--------+-------+-------+--------|
| (0,241) | Normal | 5437 | 0 | (0,241) |
Cleanup of Old Updated Rows

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

SELECT * FROM mvcc_demo_page0
OFFSET 240;

<table>
<thead>
<tr>
<th>ctid</th>
<th>case</th>
<th>xmin</th>
<th>xmax</th>
<th>t_ctid</th>
</tr>
</thead>
<tbody>
<tr>
<td>(0,241)</td>
<td>Normal</td>
<td>5437</td>
<td>5438</td>
<td>(0,242)</td>
</tr>
<tr>
<td>(0,242)</td>
<td>Normal</td>
<td>5438</td>
<td>0</td>
<td>(0,242)</td>
</tr>
</tbody>
</table>
Cleanup of Old Updated Rows

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

SELECT * FROM mvcc_demo_page0
OFFSET 240;

<table>
<thead>
<tr>
<th>ctid</th>
<th>case</th>
<th>xmin</th>
<th>xmax</th>
<th>t_ctid</th>
</tr>
</thead>
<tbody>
<tr>
<td>(0,241)</td>
<td>Redirect to 242</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(0,242)</td>
<td>Normal</td>
<td>5438</td>
<td>5439</td>
<td>(0,243)</td>
</tr>
<tr>
<td>(0,243)</td>
<td>Normal</td>
<td>5439</td>
<td>0</td>
<td>(0,243)</td>
</tr>
</tbody>
</table>

No index entry added because indexes only point to the head of the HOT chain.
Same as this Slide

Indexes

Page Header  Item  Item  Item

Redirect

8K

Tuple, v2  Tuple, v3  Special
UPDATE mvcc_demo SET val = val + 1 WHERE val > 0;

SELECT * FROM mvcc_demo_page0
OFFSET 240;

<table>
<thead>
<tr>
<th>ctid</th>
<th>case</th>
<th>xmin</th>
<th>xmax</th>
<th>t_ctid</th>
</tr>
</thead>
<tbody>
<tr>
<td>(0,241)</td>
<td>Redirect to 243</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(0,242)</td>
<td>Normal</td>
<td>5440</td>
<td>0</td>
<td>(0,242)</td>
</tr>
<tr>
<td>(0,243)</td>
<td>Normal</td>
<td>5439</td>
<td>5440</td>
<td>(0,242)</td>
</tr>
</tbody>
</table>
Same as this Slide

Indexes

Page Header Item Item Item

Redirect

8K

Tuple, v4 Tuple, v3 Special
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 240;
```

```
| ctid   | case                     | xmin | xmax | t_ctid       |
|--------+--------------------------+------+------+--------------|
| (0,241) | Redirect to 242          |      |      |              |
| (0,242) | Normal                   | 5440 | 0    | (0,242)      |
| (0,243) | Unused                   |      |      |              |
```
Same as this Slide
VACUUM Does Not Remove the Redirect

```sql
VACUUM mvcc_demo;

SELECT * FROM mvcc_demo_page0
OFFSET 240;

<table>
<thead>
<tr>
<th>ctid</th>
<th>case</th>
<th>xmin</th>
<th>xmax</th>
<th>t_ctid</th>
</tr>
</thead>
<tbody>
<tr>
<td>(0,241)</td>
<td>Redirect to 242</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(0,242)</td>
<td>Normal</td>
<td>5440</td>
<td>0</td>
<td>(0,242)</td>
</tr>
<tr>
<td>(0,243)</td>
<td>Unused</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```
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;

<table>
<thead>
<tr>
<th>ctid</th>
<th>xmin</th>
<th>xmax</th>
</tr>
</thead>
<tbody>
<tr>
<td>(0,1)</td>
<td>5442</td>
<td>0</td>
</tr>
<tr>
<td>(0,2)</td>
<td>5443</td>
<td>0</td>
</tr>
<tr>
<td>(0,3)</td>
<td>5444</td>
<td>0</td>
</tr>
</tbody>
</table>

DELETE FROM mvcc_demo;
## Cleanup Using Manual VACUUM

```sql
SELECT ctid, xmin, xmax
FROM mvcc_demo_page0;
```

<table>
<thead>
<tr>
<th>ctid</th>
<th>xmin</th>
<th>xmax</th>
</tr>
</thead>
<tbody>
<tr>
<td>(0,1)</td>
<td>5442</td>
<td>5445</td>
</tr>
<tr>
<td>(0,2)</td>
<td>5443</td>
<td>5445</td>
</tr>
<tr>
<td>(0,3)</td>
<td>5444</td>
<td>5445</td>
</tr>
</tbody>
</table>

-- too small to trigger autovacuum

```sql
VACUUM mvcc_demo;
```

```sql
SELECT pg_relation_size('mvcc_demo');
```

<table>
<thead>
<tr>
<th>pg_relation_size</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
</tr>
</tbody>
</table>
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.
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, 240);
INSERT INTO mvcc_demo VALUES (1);

SELECT * FROM mvcc_demo_page0
OFFSET 240;
  ctid    | case    | xmin  | xmax  | t_ctid
-----------------------------+----------+-------+-------+--------------------------
  (0,241) | Normal   | 5449  | 0     | (0,241)
UPDATE of an Indexed Column

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

SELECT * FROM mvcc_demo_page0
OFFSET 240;
   ctid | case   | xmin | xmax | t_ctid
---------+--------+------+------+---------
(0,241) | Normal | 5449 | 5450 | (0,242)
(0,242) | Normal | 5450 | 0    | (0,242)

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

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)
UPDATE of an Indexed Column

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

SELECT * FROM mvcc_demo_page0
OFFSET 240;

<table>
<thead>
<tr>
<th>ctid</th>
<th>case</th>
<th>xmin</th>
<th>xmax</th>
<th>t_ctid</th>
</tr>
</thead>
<tbody>
<tr>
<td>(0,241)</td>
<td>Dead</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(0,242)</td>
<td>Dead</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(0,243)</td>
<td>Normal</td>
<td>5451</td>
<td>5452</td>
<td>(0,244)</td>
</tr>
<tr>
<td>(0,244)</td>
<td>Normal</td>
<td>5452</td>
<td>0</td>
<td>(0,244)</td>
</tr>
</tbody>
</table>
UPDATE of an Indexed Column

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

SELECT * FROM mvcc_demo_page0
OFFSET 240;
  ctid | case  | xmin | xmax | t_ctid
  ---------------------------------------
   (0,241) | Dead  | | |
   (0,242) | Dead  | | |
   (0,243) | Dead  | | |
   (0,243) | Dead  | | |
   (0,244) | Normal| 5452 | 0  | (0,244)
```
**UPDATE of an Indexed Column**

VACUUM mvcc_demo;

SELECT * FROM mvcc_demo_page0
OFFSET 240;

<table>
<thead>
<tr>
<th>ctid</th>
<th>case</th>
<th>xmin</th>
<th>xmax</th>
<th>t_ctid</th>
</tr>
</thead>
<tbody>
<tr>
<td>(0,241)</td>
<td>Unused</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(0,242)</td>
<td>Unused</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(0,243)</td>
<td>Unused</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(0,244)</td>
<td>Normal</td>
<td>5452</td>
<td>0</td>
<td>(0,244)</td>
</tr>
</tbody>
</table>
Cleanup Summary

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

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

http://momjian.us/presentations