2ndQuadrant®
PostgreSQL

Recent Advances in MVCC

Pavan Deolasee

PGConf India 2020

Agenda

- Basic of transactions
- What is MVCC?
- Challenges posed by MVCC
- VACUUM - why do I need this
- Past improvements
- What's coming in near future

Transactions

- Logical unit of work
- Work done must be atomic, consistent, isolated and durable (ACID)
- Each write transaction is assigned a transaction identifier (XID)
- XID is a 32-bit unsigned integer, so yes it can wrap-around
  - 10 txns/sec - 6.8 years
  - 1000 txns/sec - 24.5 days

Multi Version Concurrency Control (MVCC)

- Mechanism to provide transactional guarantees, especially **A**tomicity and **I**solation.
- Multiple versions of a single row
- A given transaction (or query) can see at most one version of the row
- All versions are stamped with identifiers of the creating and destroying transactions
- A MVCC snapshot is used to decide which version is visible to a given query.

UPDATE Example

```
BEGIN;
UPDATE mytable
   SET somecol = 'newval'
   WHERE pkey = 'x';
COMMIT;

BEGIN;
UPDATE mytable
   SET somecol = 'newval'
   WHERE pkey = 'x';
ABORT;
```

UPDATE Example

Option 1 - Write in place, but keep enough information to undo the change (Oracle)

('x', 'newval')	('x', 'oldval')

Option 2 - Create new version of row (PostgreSQL)

('x', 'oldval')	('x', 'newval')

UPDATE Example

`xid1` - Transaction that inserted old tuple
`xid2` - Transaction that updated the old tuple and created new version

The old version has xmin set to `xid1` and xmax to `xid2`
The new version has xmin set to `xid2` and xmax is invalid

UPDATE Example - COMMIT

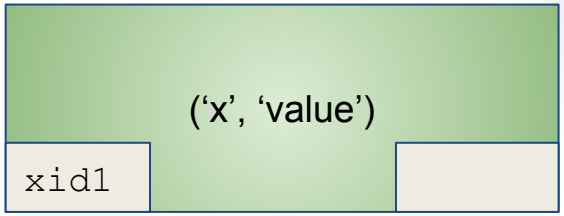
`xid1` - transaction that inserted old version is COMMITTED
`xid2` - transaction that deleted old version and inserted new version is COMMITTED

UPDATE Example - ABORT

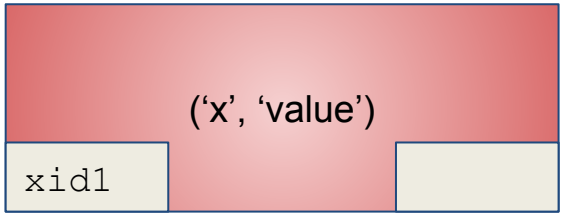
`xid1` - transaction that inserted old version is COMMITTED
`xid2` - transaction that deleted old version and inserted new version is ABORTED

INSERT Example

COMMIT Case

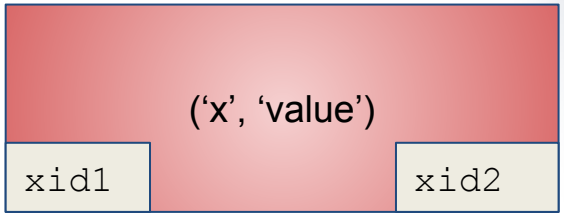


ABORT Case

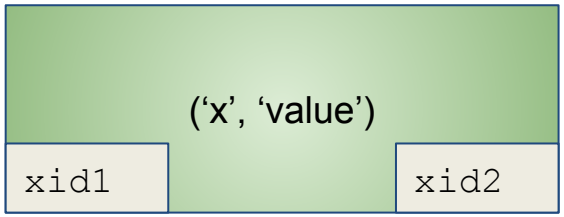


DELETE Example

COMMIT Case



ABORT Case

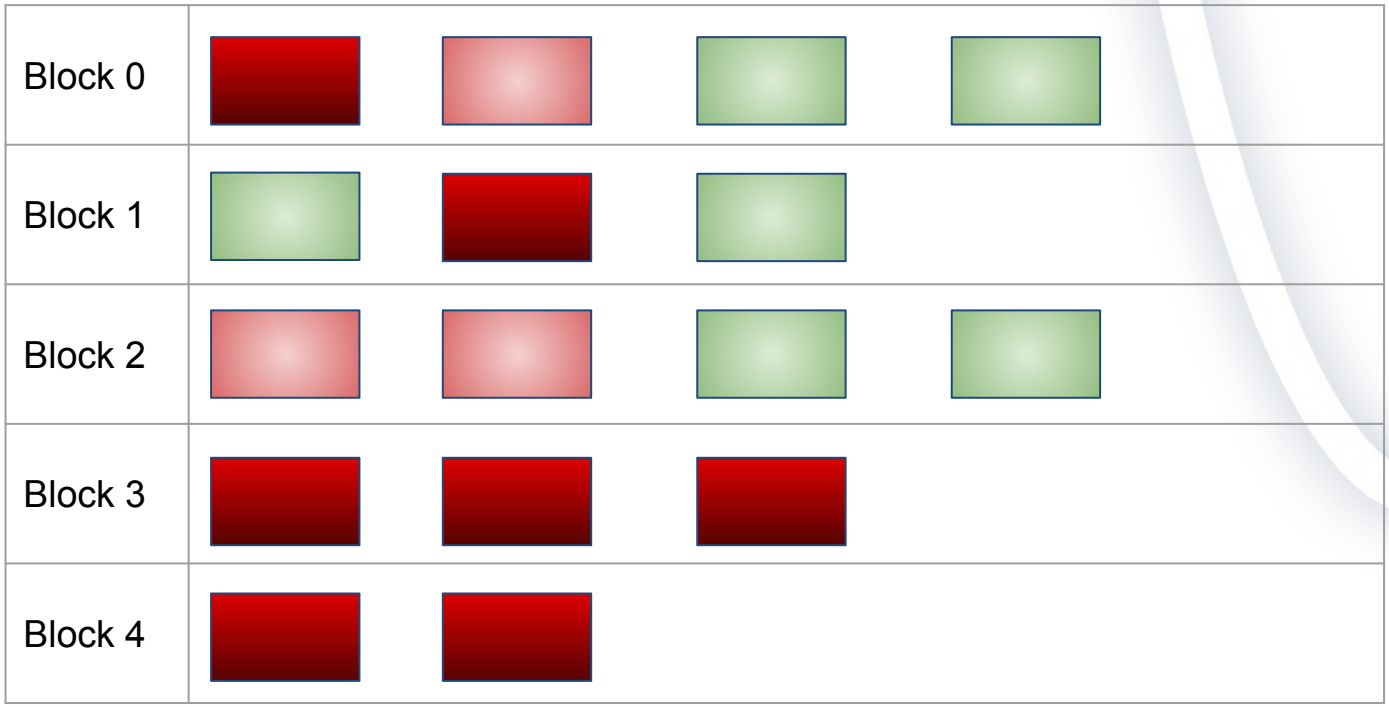


READ Using MVCC Snapshot

Snapshot tells you what's visible and what's not. It includes information about

- Transactions that are completed (either aborted or committed) when the snapshot is taken
- Transactions that are running when the snapshot is taken
- Transactions that are not even started

READ Using MVCC Snapshot



MVCC Challenges

1. UPDATEs, DELETEs and failed INSERTs eventually create dead versions, leading to heap bloat.
2. Every UPDATE inserts a new index tuple in all indexes, leading to index bloat
3. MVCC scans must check every row version for visibility
4. Transaction ID wrap-around issues
5. Row versions must store two transaction identifiers

(Auto)Vacuuming

Challenge #1 - Heap Bloat

- Remove dead rows from the table (heap)
- Remove dead pointers from indices
- Truncate heap when possible
- Normal VACUUM can’t shrink the heap by defragmentation (VACUUM FULL can)

Retail Vacuuming - HOT

Challenge #2 - Index Bloat

- (Auto)vacuum removes index bloat
- Tracking and reusing dead index tuples.
- HOT UPDATE reduces index bloat by avoiding duplicate index entries when none of the indexed keys are changed

Marking Page All-Visible

Challenge #3 - Checking every row for visibility

- If all tuples in a page are visible to all snapshots, then the page is marked as all-visible
- No need to check visibility for any tuple in such a page

Freezing

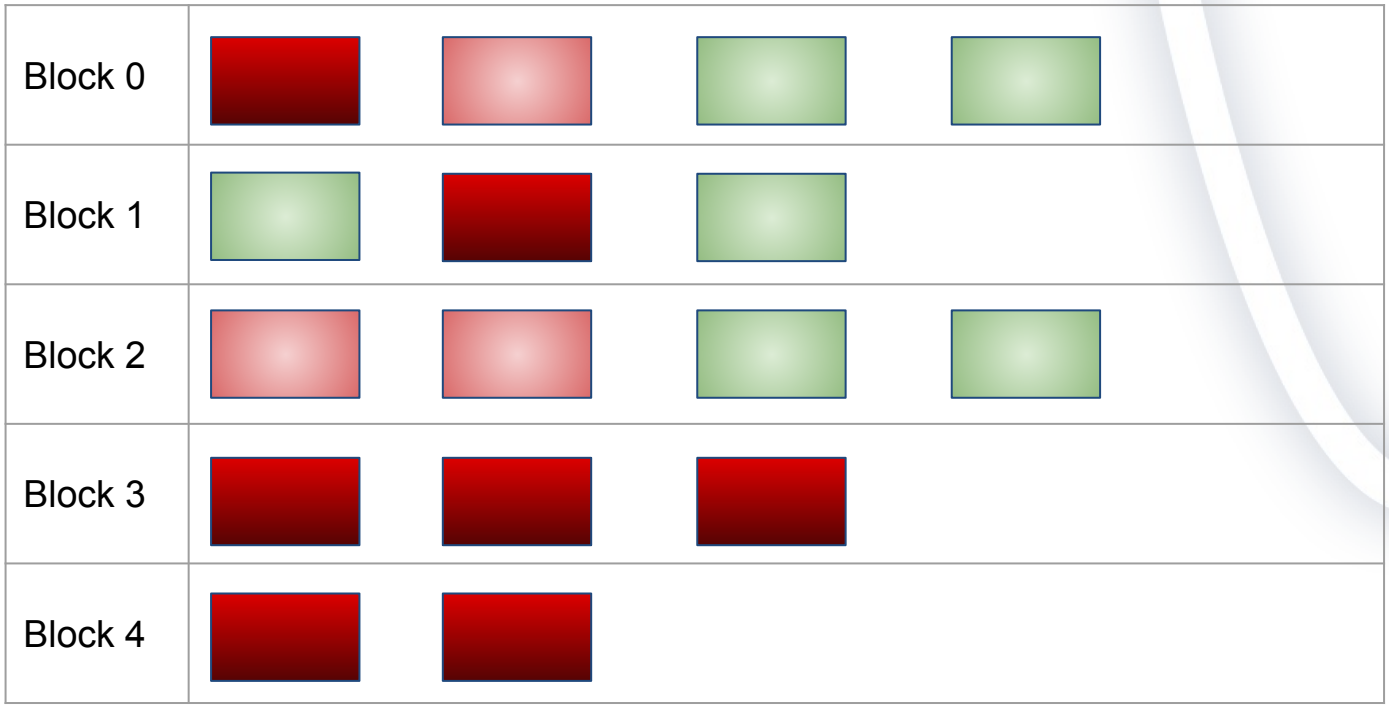
Challenge #4 - XID Wraparound

- XID wrap-around can confuse MVCC checks
- Old XIDs must be cleaned up and removed from the database before they can be reused (actually much before that)
- Inserting XID (xmin) is frozen and deleting XID (xmax, if any) is invalidated

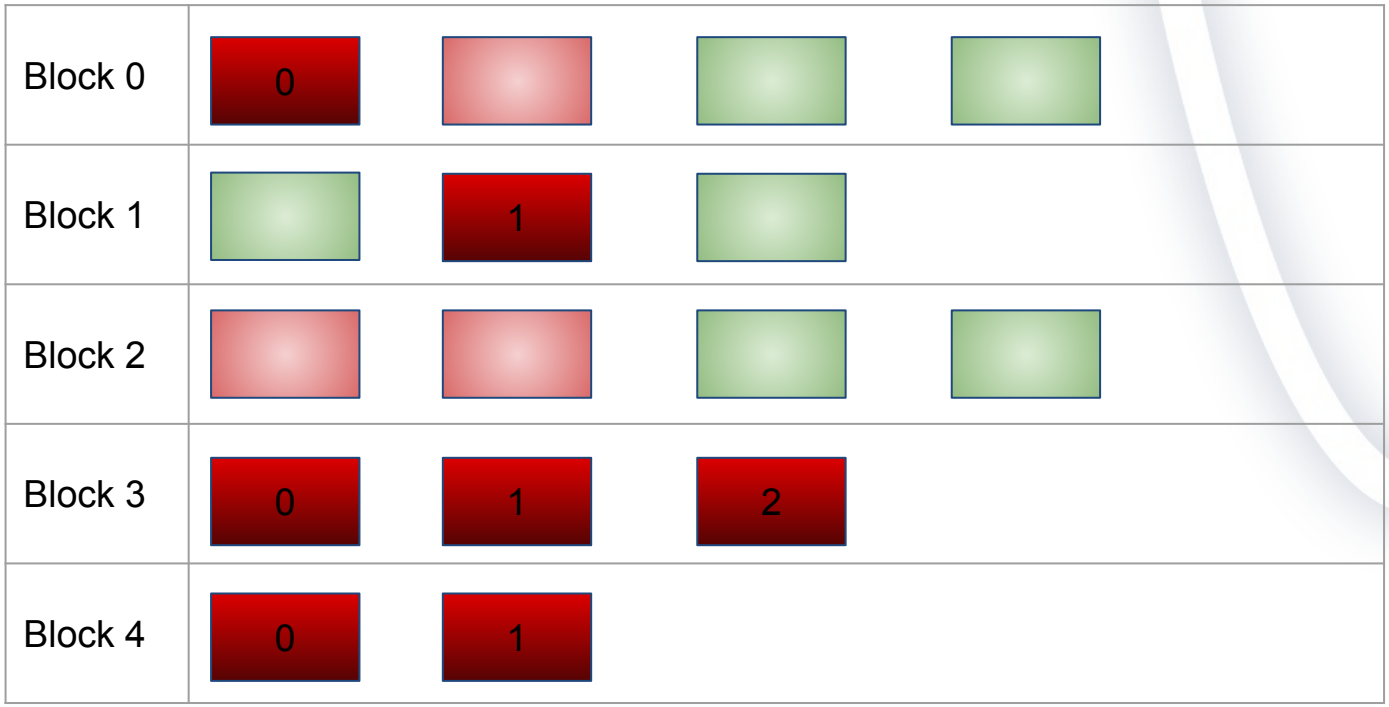
More On VACUUM

- Four stage process
  - Scan heap for dead rows, record them
  - Scan indexes and remove index pointers to dead heap rows
  - Scan heap and remove dead rows
  - Optionally truncate heap if all pages to the end are empty
- Second and third phase may need to run multiple times
- Autovacuum, manual VACUUM and vacuumdb command

VACUUM Processing



VACUUM - Heap Scan

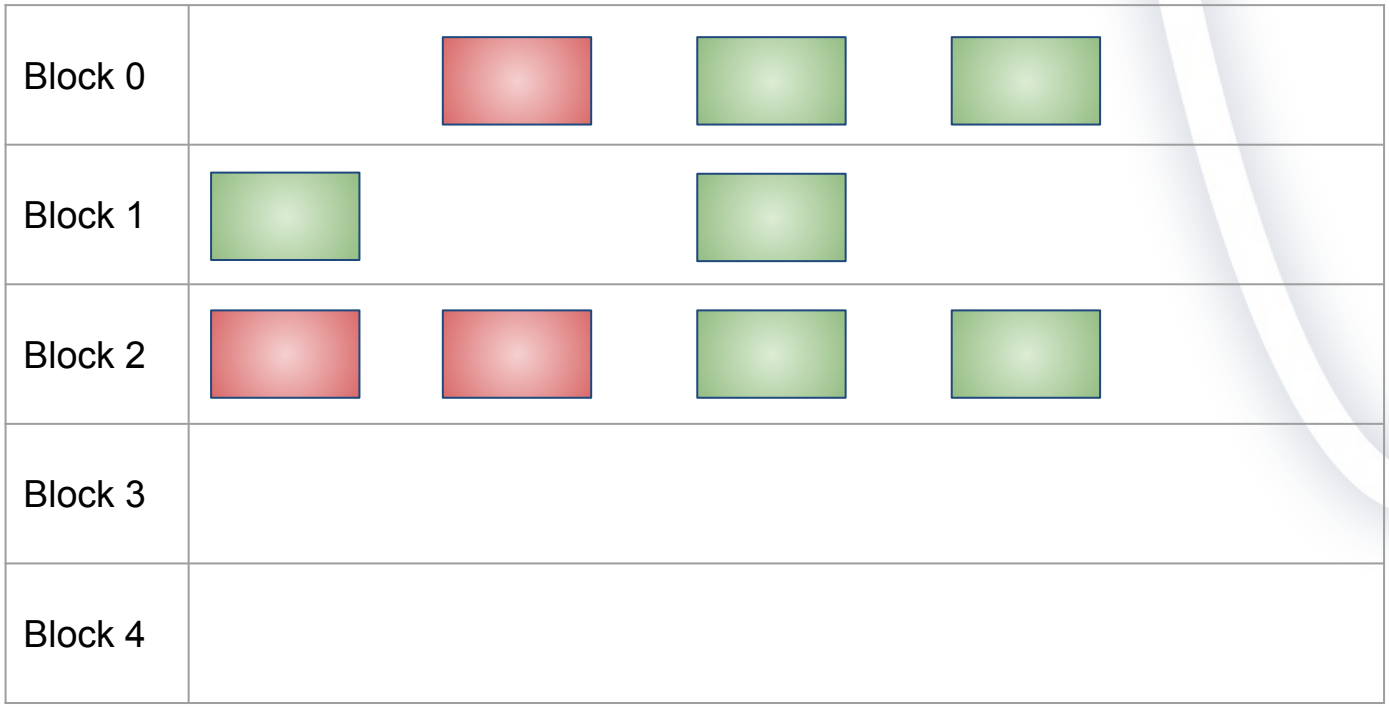


Collect DEAD tuples - (0,0), (1,1), (3,0), (3,1), (3,2), (4,0), (4,1)

VACUUM - Index Cleanup

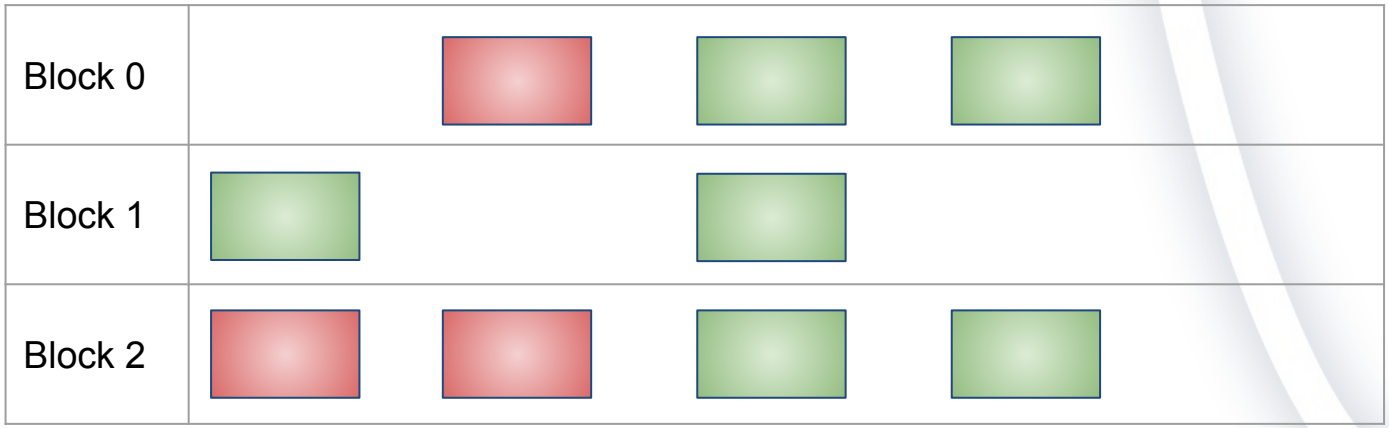
- Get list of dead tuples from heap scan
- For EACH index
  - Scan each index block
  - Check if index pointer is for one of dead tuples
  - Remove index pointer if so
- Can be quite expensive if there are large number of indexes on the table

VACUUM - Heap Cleanup



DELETE dead heap tuples

VACUUM - Truncate Heap



Truncate Heap towards the end and reclaim space

Challenges in VACUUM

- Multiple passes over the heap and the indexes
- Excessive read and write IO
- Excessive WAL generation
- Same pages scanned multiple times even if nothing changed in between
- Truncate at the end locks the table in AccessExclusiveLock mode, blocking all reads and writes
- No provision for defragmentation unless VACUUM FULL is used

Visibility/Freeze Map

- Two bits per page in the heap to indicate whether it may require vacuuming/freezing
- All-visible page has all tuples visible to all snapshots
- All-visible pages can be safely skipped from regular VACUUM
- As a bonus, also supports index-only scans
- All-frozen page has all tuples already frozen i.e. all references to XID has been removed
- VACUUM FREEZE can safely skip all-frozen pages

Parallel `vacuumdb`

```
vacuumdb --jobs
```

- Use specified number of concurrent connections
- Each connection to process one table at a time
- Make use of available compute and IO resources

VACUUM Progress Report

- Reports various phases of VACUUM
  - Initialising
  - Scanning heap
  - Vacuuming indexes
  - Vacuuming heap
  - Truncating heap
  - Performing final cleanup
- VACUUM also prints much more information than it used to

New Options to VACUUM

```
DISABLE_PAGE_SKIPPING [ boolean ]

SKIP_LOCKED [ boolean ]

INDEX_CLEANUP [ boolean ]

TRUNCATE [ boolean ]
```

DISABLE_PAGE_SKIPPING [boolean]

- Visibility map is used to skip all-visible pages during normal vacuum
- Use this option to disable that optimisation
- Useful for fixing corruption in visibility map

SKIP_LOCKED [boolean]

- Specifies that VACUUM should not wait for any conflicting locks to be released when beginning work on a relation
- If a relation cannot be locked immediately without waiting, the relation is skipped
- Useful to ensure that vacuum is not blocked

```
postgres=# VACUUM (SKIP_LOCKED) tab;
WARNING:  skipping vacuum of "tab" --- lock
not available
```

INDEX_CLEANUP [boolean]

- Specifies that VACUUM should attempt to remove index entries pointing to dead tuples.
- Setting to false may be useful when it is necessary to make vacuum run as quickly as possible, for example to avoid imminent transaction ID wraparound
- Must be used carefully or else the indexes may get bloated

TRUNCATE [boolean]

- Specifies that VACUUM should attempt to truncate off any empty pages at the end of the table
- This is normally the desired behavior and is the default
- Setting this option to false may be useful to avoid ACCESS EXCLUSIVE lock on the table that the truncation requires.

What's coming up in PG13 and beyond?

- VACUUM PARALLEL
  - Multiple background worker processes per table
  - Only used for index vacuum
- Resume interrupted vacuum
- Zheap
  - A new storage engine being developed which promises to reduce some of the pain points about MVCC
  - Leverages the work done on pluggable storage engine

2ndQuadrant

Thank You