Transactions

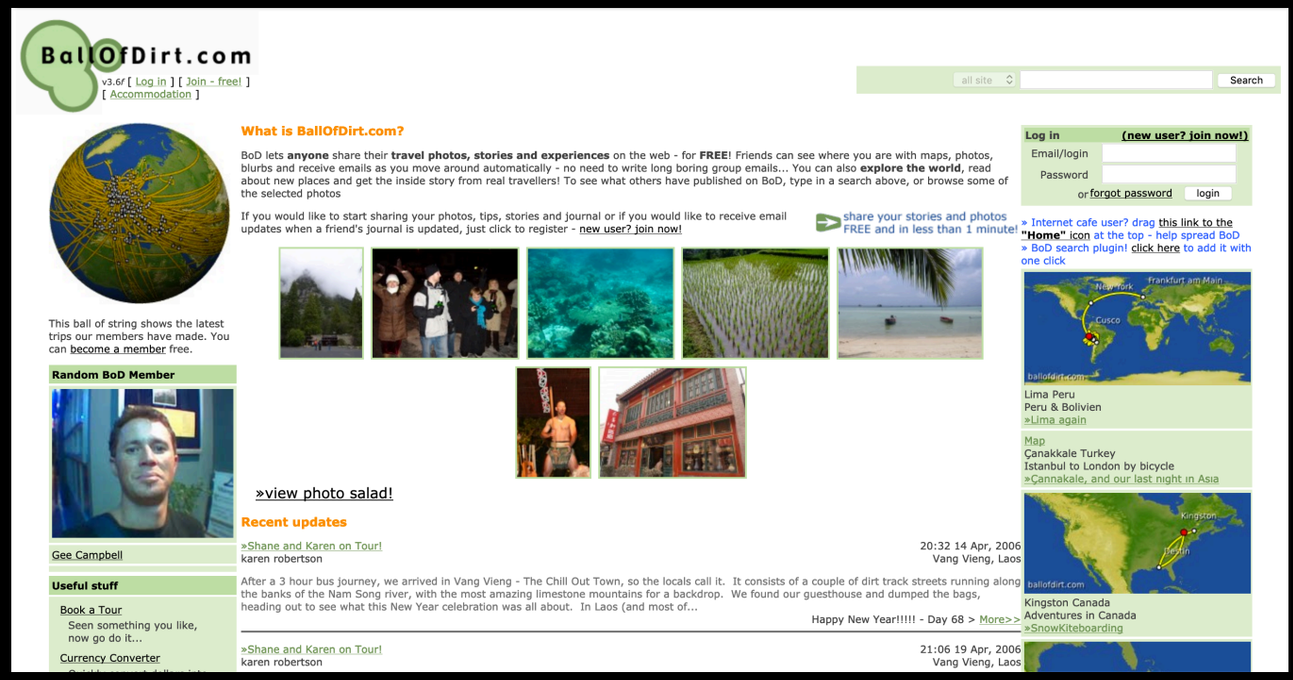
in PostgreSQL and other animals

PGCon 2019, Ottawa, Canada

Thomas Munro

My introduction to PostgreSQL

- 2001-2010 side project: travel journals, maps, photos
- Initially ran on a beige PC under the stairs, eventually on a lot of 19” wide heavy metal boxes with blinking lights
- Growing pains: Media coverage produced database meltdown scenarios induced by MySQL table-level locking
- Discovered PostgreSQL 7.3 and stayed up late learning and porting ASAP to access its MVCC writers-don’t-block-readers goodness



MySQL
planner, executor etc

MyISAM

InnoDB

buffer pool
transactions
recovery/rollback
indexes
tables
block storage
FKs
etc…

PostgreSQL
planner, executor etc

btree

gist

heap

zheap

xact

undo

buffer pool

xlog

clog

etc…

“Storage engines”

“Access methods”

Terminology

Access method: different ways of laying out and finding data. Originally used in the 1960s by IBM to model different kinds of hardware, then used in IBM System R (1970s) for different types of indexes and tables, adopted by INGRES (1970s) and POSTGRES (1980s) which were designed to support adding new kinds of indexes (eg GIST). Extended in PostgreSQL 12 to support new kinds of tables (“table AM”).

—> I don’t like to use “storage engine” or “pluggable storage” for table AMs


ACID

Atomicity. A transaction's changes to the state are atomic: either all happen or none happen. These changes include database changes, messages, and actions on transducers.

— Transaction Processing: Concepts and Techniques, Jim Gray & Andreas Reuter

Traditional approach (System R, DB2, Sybase, …):

- Lock everything you modify
- If you roll back, put everything back the way you found it! This requires keeping an **undo log**, inside or alongside the redo log (WAL)
- After committing or rolling back, everything is either done or undone, so atomicity is achieved
- After crash recovery, in-progress transactions still have to be rolled back.
- Concurrent transactions are a special case: more soon (that comes under isolation)
- Implementations vary on whether there is one undo undo per connection, some other number, or just one integrated with the WAL

"CLOG" scheme of PostgreSQL:

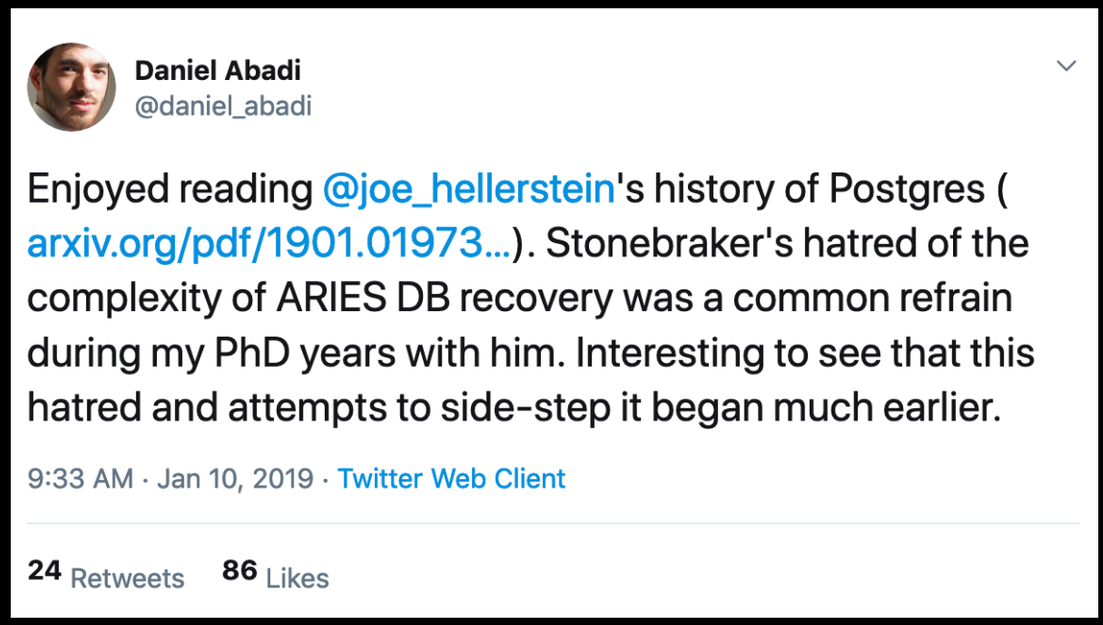
- Copy-on-write tuples forming an update chain
- Tuples are marked in such a way that readers can tell which transaction wrote it
- Maintain a commit log showing which transactions committed
- Whenever reading tuples, readers use a "snapshot" that decides whether they can see it
- A cost must be paid: the extra copies of the data — aborted transactions must eventually be garbage collected to reclaim space, and trim the CLOG and other data structures

Why this difference?

- Preceding project Ingres used WAL/REDO and UNDO logs.

- POSTGRES had neither. “When considering the POSTGRES storage system, we were guided by a missionary zeal to do something different. All current commercial systems use a storage manager with a write-ahead log (WAL), and we felt that this technology was well understood. Moreover, the original Ingres prototype from the 1970s used a similar storage manager, and we had no desire to do another implementation. [SK91]”

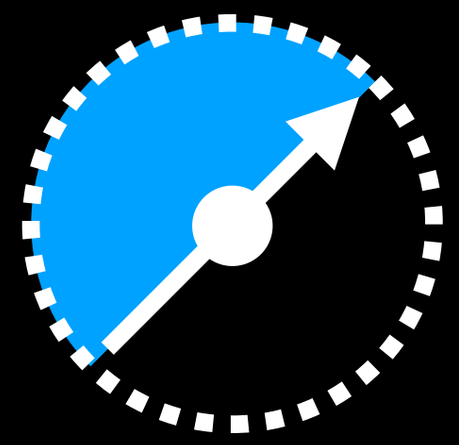
- PostgreSQL 7.1 (2001): “In this first release of WAL, UNDO operation is not implemented, because of lack of time.”



Transaction IDs

- PostgreSQL 7.2 (2002): “There is no longer a problem with installations that exceed four billion transactions.” Before that, you had to dump and restore if you ran out.

- Firebird faced the same problem but moved to 48 bit transaction IDs.



- To support circular 32 bit XIDs, we have to “freeze” old tuples (= remove old transaction IDs) so that the range of active XIDs never exceeds 2^31. Sometimes the resulting “wrap-around vacuums” cause significant grief.

- PostgreSQL 12 takes a very small step towards 64 bit “full” transaction IDs, for use by future AMs*. Not yet used in many places but…

Pixabay free clipart

*Full transaction IDs will run out after ~11.7 million years of doing 50,000 TPS. After that, you’ll probably have to dump and restore. But you’ll probably run out of LSNs first.

A hypothetical heap with no need to freeze

- People have proposed schemes for adding a “reference” full transaction ID to heap page headers, so that old committed tuples don’t have to be frozen, even though tuples carry only the lower 32 bits in their xmin and xmax as now.

- That leaves only the uncommitted transaction IDs to worry about. We have to somehow get rid of them before the CLOG can be truncated, so that we don’t suddenly think that ancient aborted transactions committed.

- Hmm, if only we had a reliable technology to do arbitrary jobs when rolling back a transaction…

*And something something multixacts. “We loved vacuum freeze so much we invented a whole new kind.”

A simpler use for undo logs

- Even though the traditional heap AM uses the CLOG strategy to deal with atomicity at the level of tuples, there are other transaction effects that we could track better

- Case in point: orphaned file clean-up, for the relation files used by heap, btree, gist, gin, hash. If you create a relation and then crash before committing, we forget to unlink the files!

```
postgres=# begin;
BEGIN
postgres=# create table t as select generate_series(1, 1000000);
SELECT 1000000
postgres=# select pg_backend_pid(), relfilenode from pg_class where relname = 't';
 pg_backend_pid | relfilenode
----------------+-------------
          74817 |       24576
(1 row)


$ kill -9 74817
$ ls -slaph pgdata/base/13643/24576
70928 -rw-------  1 munro  staff    35M 29 May 18:45 pgdata/base/13643/24576
```

- Undo-aware RMGRs (AMs or other subsystem) can define their own types of undo record, and store them. They must generate the exact same undo record at redo time. In this case it's src/backend/catalog/storage.c, which is responsible for creating and unlinking regular relation files.

- If the transaction commits, all associated undo data is efficiently discarded by a "discard worker".

- If the transaction aborts, the registered callback is invoked to execute any cleanup actions. Small transactions' undo records are executed in the foreground, and large transactions' undo records are pushed to a background "undo apply worker". This happens automatically after crash recovery.

```
postgres=# begin;
BEGIN
postgres=# create table t1 ();
CREATE TABLE
postgres=# create table t2 ();
CREATE TABLE
postgres=# select * from undoinspect();
     urecptr      |  rmgr   | flags | xid |                  description
------------------+---------+-------+-----+-----------------------------------------------
 0000000000003452 | Storage | P     |     | CREATE dbid=13643, tsid=1663, relfile=24588
 0000000000003404 | Storage | P,T   | 497 | CREATE dbid=13643, tsid=1663, relfile=24585
(2 rows)
```

```
bool
smgr_undo(UndoRecInfo *urp_array,
          int first_idx,
          int last_idx,
          Oid reloid,
          FullTransactionId full_xid,
          BlockNumber blkno,
          bool blk_chain_complete)
{
    int i;

    for (i = first_idx; i <= last_idx; ++i)
    {
        UndoRecInfo *urec_info = &urp_array[i];
        UnpackedUndoRecord *uur = urec_info->uur;

        if (uur->uur_type == UNDO_SMGR_CREATE)
        {
            SMgrRelation srel;
            RelFileNode *rnode;
            xl_smgr_drop xlrec;

            Assert(uur->uur_payload.len == sizeof(RelFileNode));
            rnode = (RelFileNode *) uur->uur_payload.data;
            srel = smgropen(SMGR_MD, *rnode, InvalidBackendId);
            smgrdounlink(srel, false);
            smgrclose(srel);

            xlrec.rnode = *rnode;
            XLogBeginInsert();
            XLogRegisterData((char *) &xlrec, sizeof(xlrec));
            XLogInsert(RM_SMGR_ID, XLOG_SMGR_DROP);
        }
        else
            elog(PANIC,
                 "smgr_undo: unknown op code %d", uur->uur_type);
    }

    return true;
}
```

To give it a chance to work efficiently, the callback receives batches of undo records relating to the same page of a relation. In this simple case the record is not page oriented.

Effects of rolling back must be WAL logged. These are called “compensation records” in the literature.

On success, the passed-in undo records can be discarded. If we fail, or crash before reaching this, they’ll be retried.

Monitoring undo logs

- The meta-data used for space management within each undo log is: discard <= insert <= end. Discard and insert we have met; end shows unused space that has been allocated on disk.

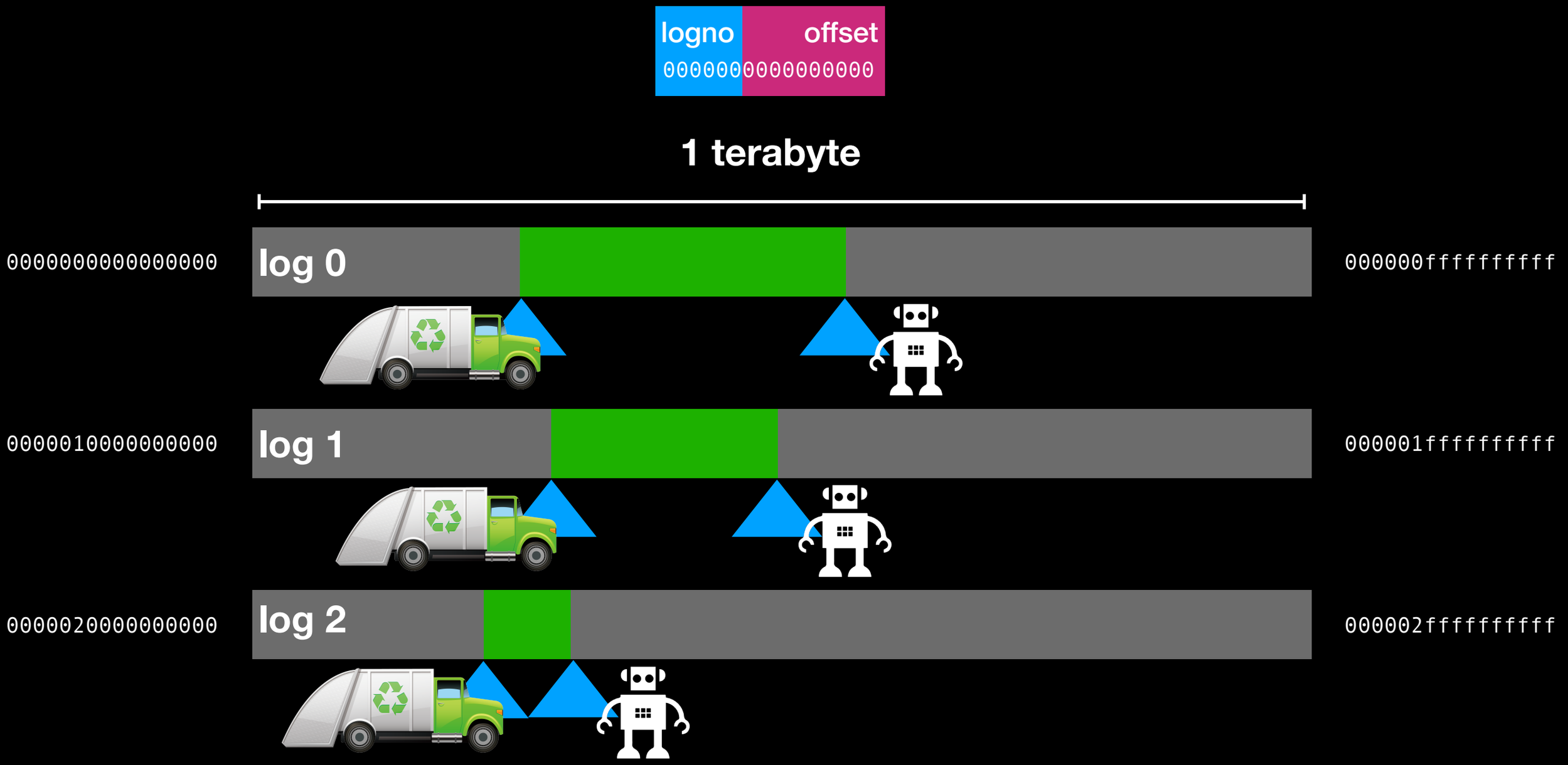
- We also track the currently attached backend and xid, if there is one. These are visible in the pg_stat_undo_logs view.

- Undo record pointers are 64 bit numbers and the address space is never reused, but much like FullTransactionIds, 64 bits ought to be enough for anyone*

```
postgres=# select * from pg_stat_undo_logs;
 log_number | persistence | tablespace |     discard      |      insert      |       end        | xid |  pid
------------+-------------+------------+------------------+------------------+------------------+-----+-------
          0 | permanent   | pg_default | 000000000000004A | 000000000000004A | 0000000000400000 | 559 | 56156
          1 | permanent   | pg_default | 00000100009C1908 | 00000100009C1908 | 0000010001000000 | 562 | 56163
          2 | permanent   | pg_default | 000002000000004A | 000002000000004A | 0000020000400000 | 563 | 56174
(3 rows)
```

*Or, at least, LSNs will run out first

Why “logs” plural?
Address space arbitrarily chopped up into 16.7m x 1TB ranges to reduce contention



What determines the amount of undo data?

- You always need to keep undo data that might be needed to roll back a transaction.
- For MVCC purposes, it depends on your tolerance. In our current code, we keep it as long as any snapshot needs it, but in theory we could have an adjustable retention policy and cause errors for old snapshots so that we can put a cap on the amount of undo data.

Consistency. A transaction is a correct transformation of the state. The actions taken as a group do not violate any of the integrity constraints associated with the state. This requires that the transaction be a correct program.

— Transaction Processing: Concepts and Techniques, Jim Gray & Andreas Reuter

I’m not going to talk about constraints and locking, but please see my colleague Kuntal Gosh’s talk “Tuple Locking Redesigned” at 4pm for a detailed overview of the locking system in zheap.

Isolation. Even though transactions execute concurrently, it appears to each transaction, T, that others executed either before T or after T, but not both.

— Transaction Processing: Concepts and Techniques, Jim Gray & Andreas Reuter

Isolation levels

SQL Standard:

- ~~UNCOMMITTED READ~~ — challenge: got a non-performance use for this?
- READ COMMITTED — most people's default
- REPEATABLE READ
- SERIALIZABLE — true isolation, lofty ideal, and the Standard's default

Also:

- SNAPSHOT ISOLATION (PostgreSQL's REPEATABLE READ)
- SERIALIZABLE SNAPSHOT ISOLATION (PostgreSQL's SERIALIZABLE)

Traditional approach:

- UNCOMMITTED READ: All data is fair game, results arbitrarily screwy, but it's *fast*

- READ COMMITTED: If you manage to share-lock it, you can see it! Therefore writers block readers, until they either roll back (old version restored) or commit (become visible).

- REPEATABLE* READ: Same, but share locks are held for whole transaction (preventing writers); lock escalation and deadlocks become more likely

- SERIALIZABLE: Similar, but also locking things that aren't there ("predicates", "gaps", "next key")

Writers block readers. The problem escalates more quickly at higher isolation levels.

PostgreSQL approach:

- UNCOMMITTED READ: maps to READ COMMITTED

- READ COMMITTED: Take a new snapshot (= set of visible tansactions) and used it to decide which tuples you can see

- REPEATABLE READ = SI: Same, but only take new snapshot one per transaction.

- SERIALIZABLE = SSI: Same, but add optimistic predicate locking scheme

Writers don't block readers! But you can't update in place, and there is a garbage collection problem.

Traditional heap

old version

row

old version

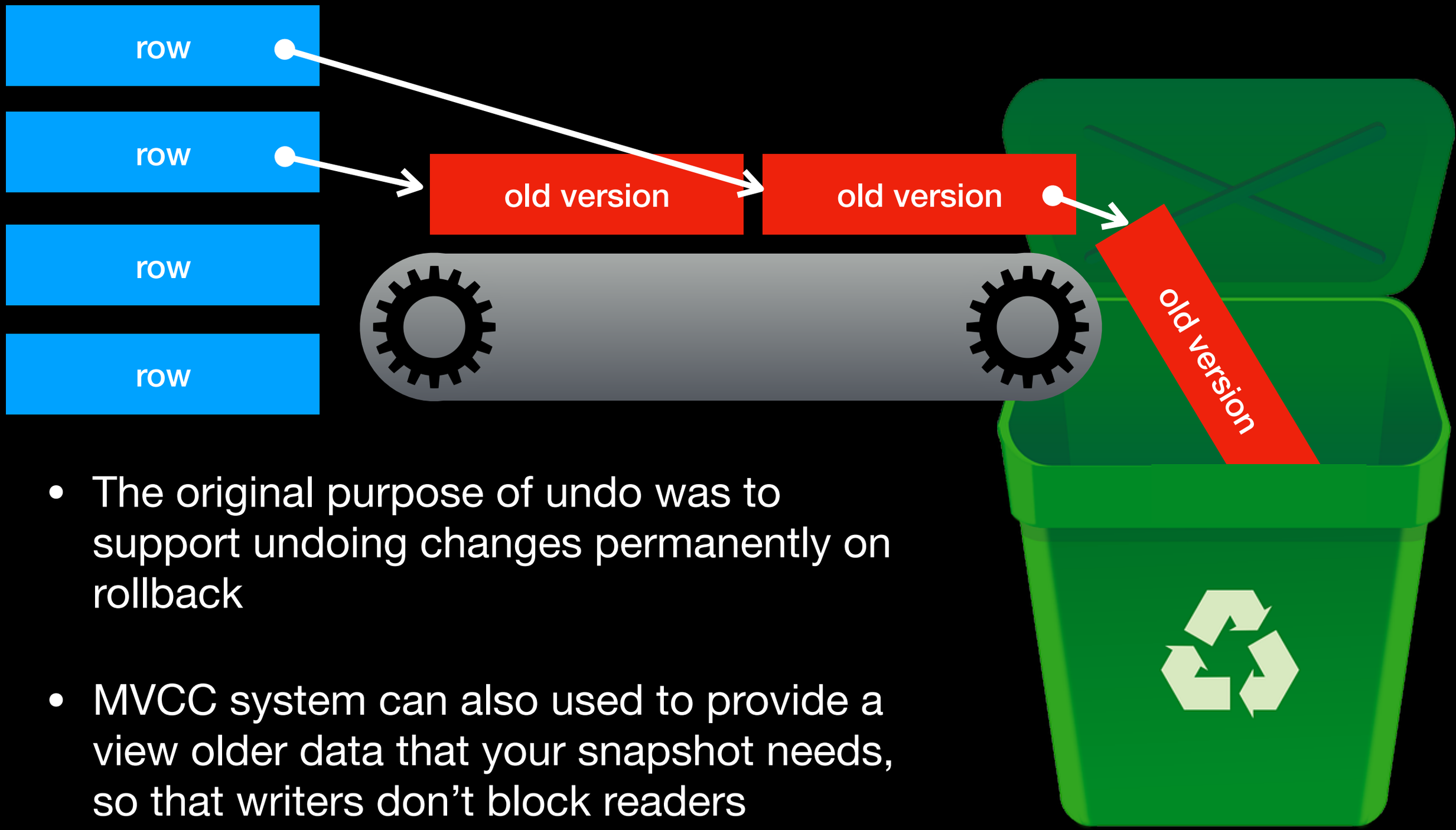
old version

old version

free space

row

Undo-based MVCC



- The original purpose of undo was to support undoing changes permanently on rollback
- MVCC system can also used to provide a view older data that your snapshot needs, so that writers don’t block readers

Durability. Once a transaction completes successfully (commits), its changes to the state survive failures.

— Transaction Processing: Concepts and Techniques, Jim Gray & Andreas Reuter

University POSTGRES approach to durability:

- COMMIT flushes all dirty buffers buffers (“force”)
- CLOG is a plain relation, and is access through shared buffers
- Behold antique 4.2 code (1994)

```
/* ----------------------------------
 *  RecordTransactionCommit
 *
 *  Note: the two calls to BufferManagerFlush() exist to ensure
 *        that data pages are written before log pages.  These
 *        explicit calls should be replaced by a more efficient
 *        ordered page write scheme in the buffer manager
 *        -cim 3/18/90
 * ----------------------------------
 */
void
RecordTransactionCommit()
{
    TransactionId xid;
    int leak;

    /* ----------------
     *  get the current transaction id
     * ----------------
     */
    xid = GetCurrentTransactionId();

    /* ----------------
     *  flush the buffer manager pages.  Note: if we have stable
     *  main memory, dirty shared buffers are not flushed
     *  plai 8/7/90
     * ----------------
     */
    leak = BufferPoolCheckLeak();
    FlushBufferPool(!TransactionFlushEnabled());
    if (leak) ResetBufferPool();

    /* ----------------
     *  have the transaction access methods record the status
     *  of this transaction id in the pg_log / pg_time relations.
     * ----------------
     */
    TransactionIdCommit(xid);

    /* ----------------
     *  Now write the log/time info to the disk too.
     * ----------------
     */
    leak = BufferPoolCheckLeak();
    FlushBufferPool(!TransactionFlushEnabled());
    if (leak) ResetBufferPool();
}
```

Traditional approach, and PostgreSQL 7.1 (2001):

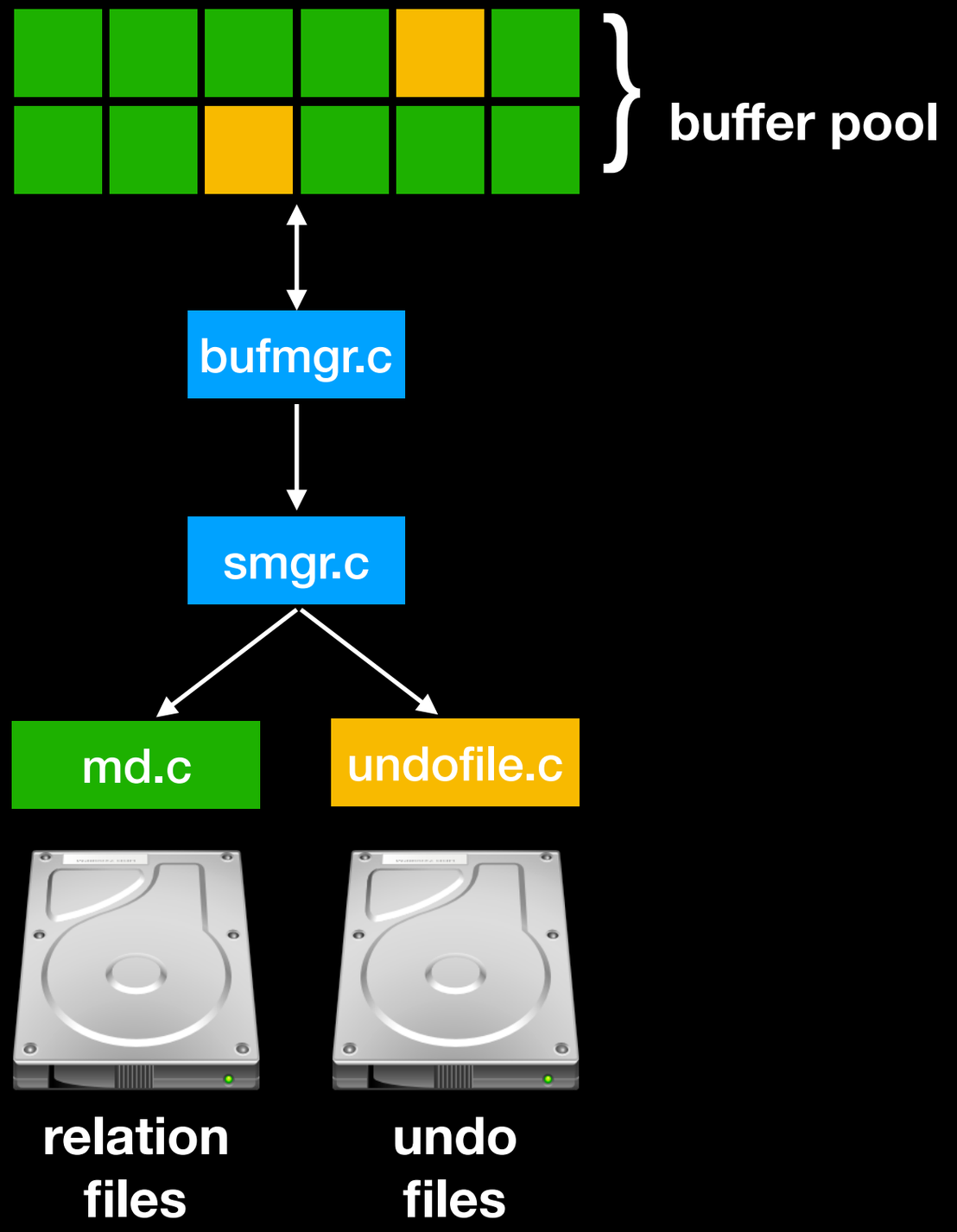
- COMMIT flushes WAL records only (“no force”)
- Checkpoints and background writers flush buffers, but the last WAL to touch a page must always be flushed before any buffer it modifies
- Regular backends only have to write pages out if they have to evict one and flush it first (“steal”)

Buffered access

- Undo logs are accessed via regular shared buffers, so we need to extend that slightly to allow for new types of data

- Such buffered access allows for efficient access when supporting MVCC

- Plans are afoot to access other kinds of things through bufmgr.c and smgr.c too



- To handle the possibility of torn pages (power failure that leaves you with a fraction of an old page and a fraction of a new page), we have a “full page write” scheme, where the first WAL record that dirties a page after a checkpoint redo point must include a complete image of the page in the WAL
- Undo log data pages are treated the same way, and have a standard header that include a checksum field

- An alternative double-writing scheme is used by MySQL: if you write the page and flush it twice in succession, only one copy can possibly be torn. This has various trade-offs but so far has been rejected for inclusion in PostgreSQL.

Files

- The name of each 1MB file is the UndoRecPtr address of the first byte in the file, with a dot inserted to separate the undo log number from the rest

- When discarding files, we usually just rename them into position, so that they become new space (similar to what we do for WAL segments); this usually happens in the undo worker

- This means that foreground processes usually avoid having to do slow filesystem operations

```
$ ls -slaph base/undo/ | head -7
total 139264
   0 drwx------  70 munro  staff   2.2K 26 Mar 09:35 ./
   0 drwx------   7 munro  staff   224B 26 Mar 09:33 ../
2048 -rw-------   1 munro  staff   1.0M 26 Mar 09:38 000000.0000600000
2048 -rw-------   1 munro  staff   1.0M 26 Mar 09:33 000000.0000700000
2048 -rw-------   1 munro  staff   1.0M 26 Mar 09:38 000001.0000600000
2048 -rw-------   1 munro  staff   1.0M 26 Mar 09:33 000001.0000700000
```

Tablespaces

- GUC “undo_tablespaces” controls where your session writes undo data (similar to “temp_tablespaces”)
- Tablespace can only be dropped when contained undo logs are empty (no attached transactions in progress, fully discarded); attached sessions will be forcibly detached

```
postgres=# create tablespace ts1 location '/tmp/ts1';
CREATE TABLESPACE
postgres=# set undo_tablespaces = ts1;
SET
postgres=# insert into foo values (42);
INSERT 0 1
postgres=# select * from pg_stat_undo_logs where tablespace = 'ts1';
 log_number | persistence | tablespace |     discard      |      insert      |       end        |  xid   |  pid
------------+-------------+------------+------------------+------------------+------------------+--------+-------
         60 | permanent   | ts1        | 00003C0000000018 | 00003C0000000018 | 00003C0000100000 | 189257 | 46137
(1 row)
postgres=# drop tablespace ts1;
DROP TABLESPACE


 2018-03-28 15:44:50.265 NZDT [46137] LOG:  created undo segment "pg_tblspc/16416/PG_11_201802061/undo/00003C.0000000000"
```

Summary

- Zheap and other table AMs in development can now use the 64 bit XIDs so that they don't need "freezing" or "wrap-around vacuums"

- Support is proposed for buffered, checkpointed, checksummed, torn-page-proof access to new kinds of data files (undo being one example)

- An undo infrastructure is proposed to support future in-place-update AMs that need rollback actions and MVCC

- A fix the problem of orphaned files that is also a simple test case for the the undo infrastructure

Some references

- Looking Back at Postgres (Joseph M Hellerstein, 2019)
- System R: Relational Approach to Database Management (Astrahan, Blasgen, Chamberlin, Gray et al, 1975)
- ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging (Mohan et al, 1992)
- A Critique of ANSI SQL Isolation Levels (Berenson, Bernstein, Gray, Melton, O’Neil, O’Neil, 1995)