Oriole
data base

The Future of PostgreSQL

Fostering Adoption Through Extensibility

Alexander Korotkov

2023

Initial Postgres Concept

- Complex datatypes support
- **Extensibility**
- Rules
- Heap as replacement of undo/redo logs
- Object-oriented elements

Stonebraker M., Rowe L. A., Hirohama M. The implementation of POSTGRES //IEEE Transactions on Knowledge & Data Engineering. - 1990. – #. 1. – pp. 125-142.

Extendability – the core idea

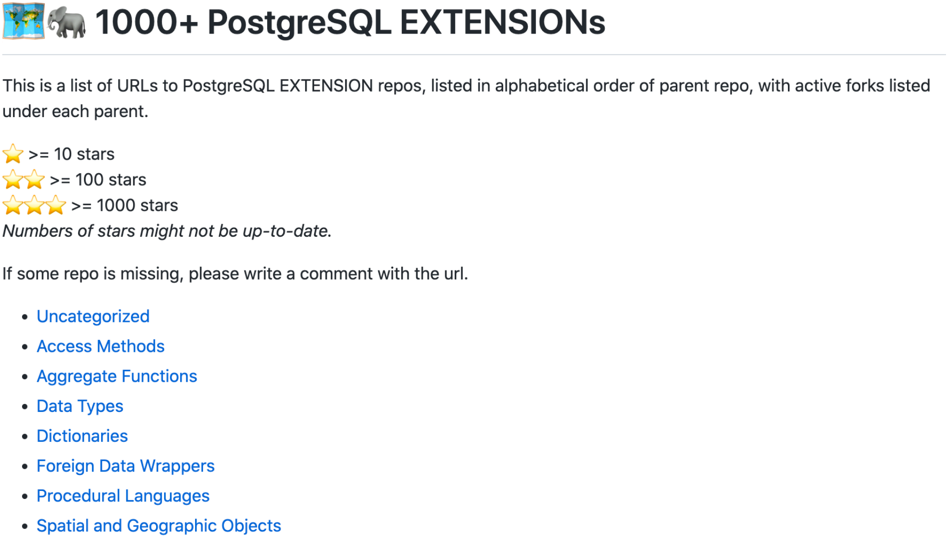
Extensibility

It is imperative that a user be able to construct new access methods to provide efficient access to instances of nontraditional base types Michael Stonebraker, Jeff Anton, Michael Hirohama.

Extendability in POSTGRES , IEEE Data Eng. Bull. 10 (2) pp.16-23, 1987

- Data types
- Functions
- Procedural languages
- Operators
- Operator classes
- **Access Methods (index & table)**
- **Hooks, custom shared memory, custom workers**

Tons of extensions



🗺️🐘 1000+ PostgreSQL EXTENSIONs

This is a list of URLs to PostgreSQL EXTENSION repos, listed in alphabetical order of parent repo, with active forks listed under each parent.

⭐ >= 10 stars
⭐⭐ >= 100 stars
⭐⭐⭐ >= 1000 stars
Numbers of stars might not be up-to-date.

If some repo is missing, please write a comment with the url.

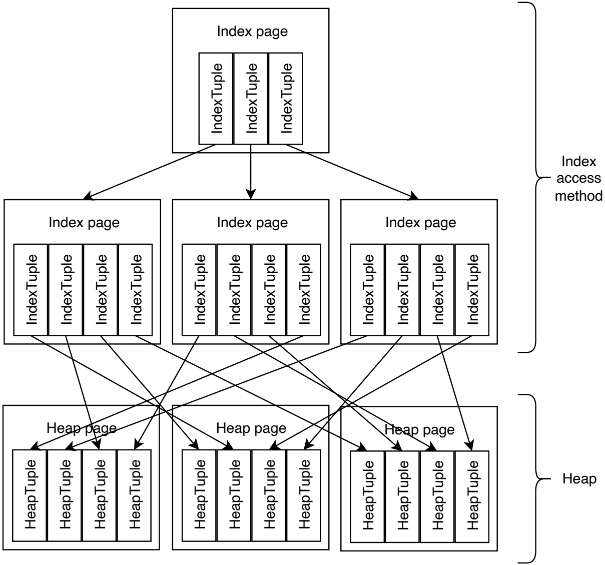
- Uncategorized
- Access Methods
- Aggregate Functions
- Data Types
- Dictionaries
- Foreign Data Wrappers
- Procedural Languages
- Spatial and Geographic Objects

Super extensions

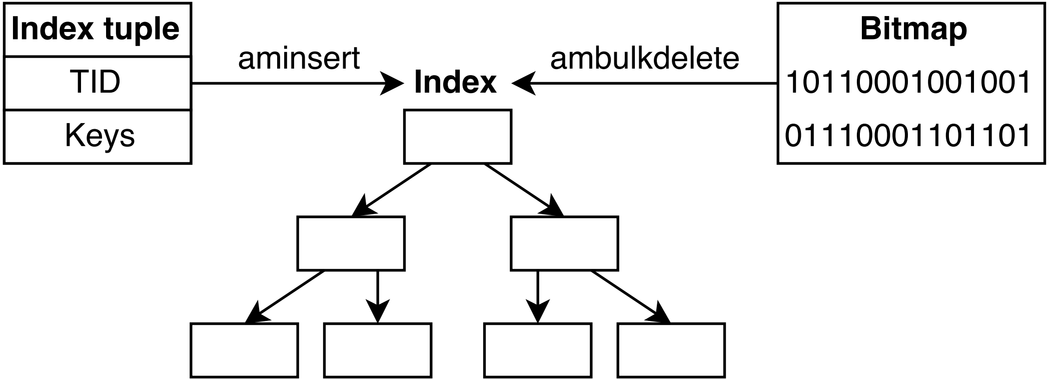
- PostGIS – defines new datatypes, functions & operators on them, operator classes for indexing.
- Citus – heavily uses hooks, custom shmem, and custom workers to provide distributed database for dashboarding. Provides columnar table AM.
- TimescaleDB – heavily uses hooks, custom shmem, and custom workers to provide distributed database for timeseries.

PostgreSQL Core principles

Heap table access method



Index access method API



Index tuple

TID

Keys

aminsert

Index

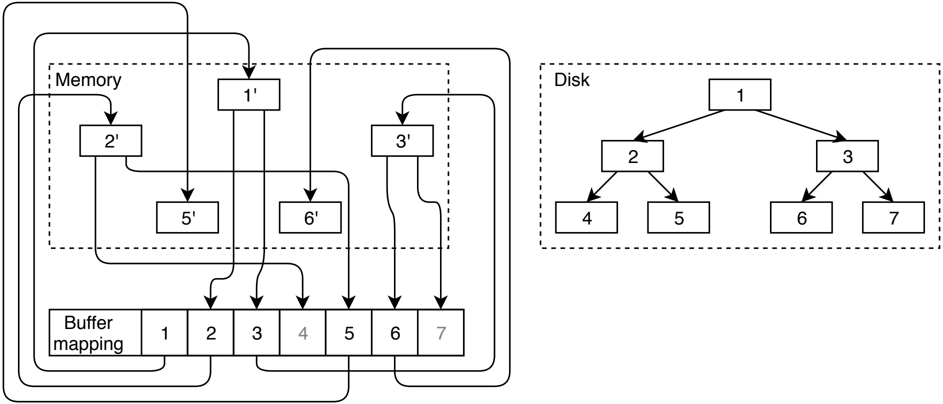
ambulkdelete

Bitmap

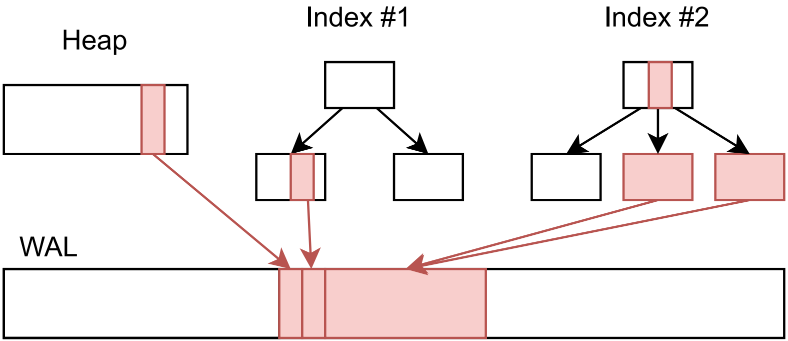
10110001001001

01110001101101

Buffer manager

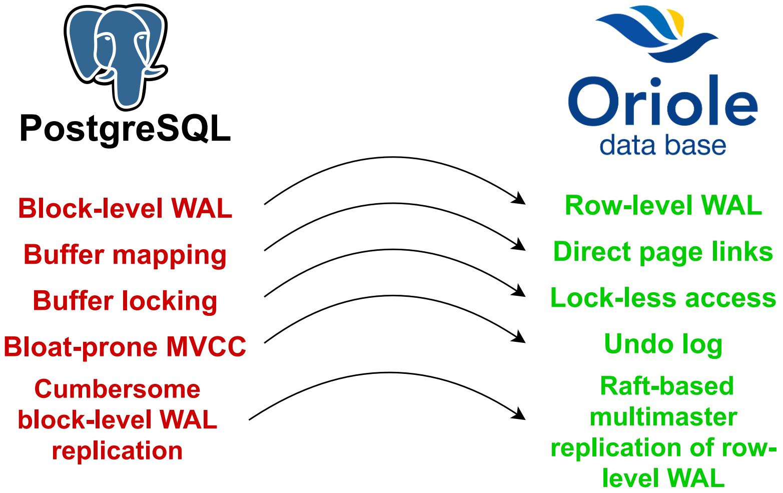


Block-level WAL



Could we have a new core?

OrioleDB – new core for PostgreSQL



PostgreSQL

Oriole data base

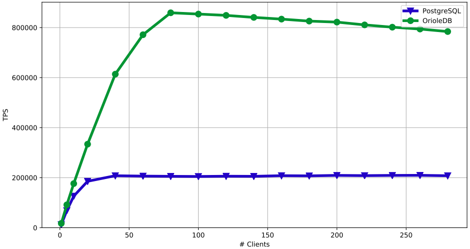
Block-level WAL → Row-level WAL

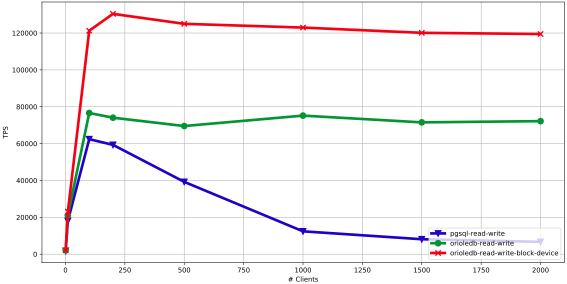
Buffer mapping → Direct page links

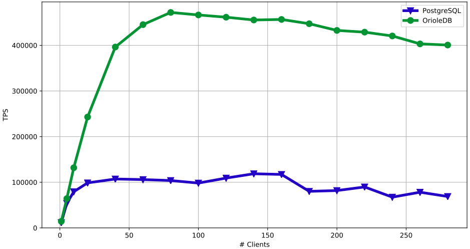
Buffer locking → Lock-less access

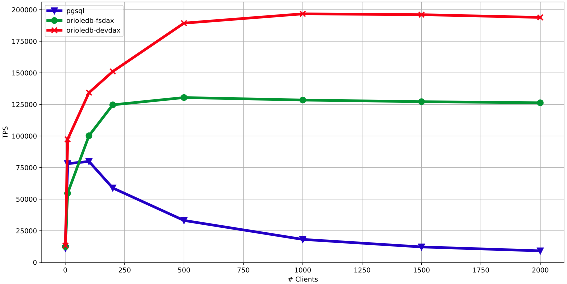
Bloat-prone MVCC → Undo log

Cumbersome block-level WAL replication → Raft-based multimaster replication of row-level WAL


Read-only scalability test PostgreSQL vs OrioleDB
1 minute of pgbench script reading 9 random values of 100M


pgbench -s 20000 -j $n -c $n -M prepared on odb-node02
mean of 3 3-minute runs with shared_buffers = 32GB(128GB), max_connections = 2500


Read-write scalability test PostgreSQL vs OrioleDB
1 minute of pgbench TPC-B like transactions wrapped into stored procedure


pgbench -s 20000 -j $n -c $n -M prepared -f read-write-proc.sql on node03
5-minute run with shared_buffers = 32GB, max_connections = 2500

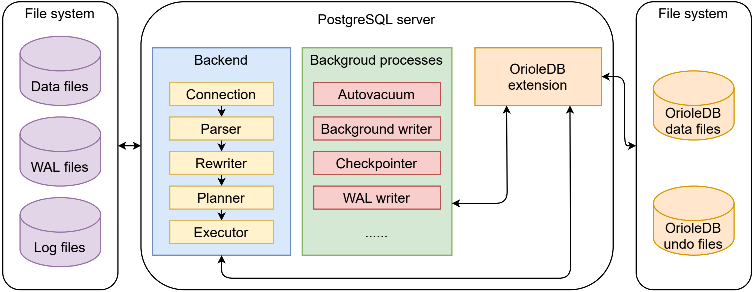
OrioleDB’s answer to 10 wicked problems of PostgreSQL [1]

Problem name	Solution
1. Wraparound	**Native 64-bit transaction ids**
2. Failover Will Probably Lose Data	**Multimaster replication**
3. Inefficient Replication That Spreads Corruption	**Row-level replication**
4. MVCC Garbage Frequently Painful	**Non-persistent undo log**
5. Process-Per-Connection = Pain at Scale	**Migration to multithread model**
6. Primary Key Index is a Space Hog	**Index-organized tables**
7. Major Version Upgrades Can Require Downtime	**Multimaster + per-node upgrade**
8. Somewhat Cumbersome Replication Setup	**Simple setup of raft-based multimaster**
9. Ridiculous No-Planner-Hints Dogma	**In-core planner hints**
10. No Block Compression	**Block-level compression**

*** Scalability on modern hardware**

[1] `https://gist.github.com/akorotkov/f5e98ba5805c42ee18bf945b30cc3d67`

PostgreSQL extendability improvements



- Extended table AM.
- Custom toast handlers.
- Custom row identifiers.
- Custom error cleanup.
- Recovery & checkpointer hooks.

- Snapshot hooks.
- Some other miscellaneous hooks total 3K lines patch to PostgreSQL Core

TableAM improvements

- Custom RowId
- INSERT ... ON CONFLICT ... patch
- tuple_is_current() patch
- Custom analyze table
- Control on the index definition
- Control both table and index options
- Allow complex rd_amcache

rowId issue

- Currently the row must be identified by ctid (32-bits block number, 16-bits offset number). Some code paths still assumes that row version is also identified by ctid.
- Show stopper for index-organized tables.

rowId patch (1/2)

```
typedef enum RowRefType
{
    ROW_REF_TID,
    ROW_REF_ROWID, /* bytea! */
    ROW_REF_COPY   /* used for FDWs */
} RowRefType;
...........................
typedef struct TableAmRoutine
{
...........................
    RowRefType (*get_row_ref_type) (Relation rel);
...........................
```

rowId patch (2/2)

```
typedef struct TableAmRoutine
{
...........................
    /* see table_tuple_delete() for reference about parameters */
    TM_Result   (*tuple_delete) (Relation rel, Datum tupleid,
...........................
    /* see table_tuple_update() for reference about parameters */
    TM_Result   (*tuple_update) (Relation rel, Datum tupleid,
...........................
    /* see table_tuple_lock() for reference about parameters */
    TM_Result   (*tuple_lock) (Relation rel, Datum tupleid,
...........................
    TupleTableSlot *(*tuple_insert) (Relation rel, TupleTableSlot *slot,
...........................
```

RowId patch summary

- Allow bytea row identifier, which could potentially hold anything.
- Bitmap scans should be replaced or disabled (currently we do this with hooks).

INSERT ... ON CONFLICT ...

```
typedef struct TableAmRoutine
{
...........................
    /* see table_tuple_insert_speculative() for reference about parameters */
    void (*tuple_insert_speculative) (Relation rel, TupleTableSlot *slot,
                                      CommandId cid, int options,
                                      struct BulkInsertStateData *bistate,
                                      uint32 specToken);
    /* see table_tuple_complete_speculative() for reference about parameters */
    void (*tuple_complete_speculative) (Relation rel, TupleTableSlot *slot,
                                        uint32 specToken, bool succeeded);
```

Seems like **very** implementation-depended API.

INSERT ... ON CONFLICT ... patch

```
typedef struct TableAmRoutine
{
...........................
    TupleTableSlot *(*tuple_insert_with_arbiter) (
                 ResultRelInfo *resultRelInfo,
                 TupleTableSlot *slot, CommandId cid, int options,
                 struct BulkInsertStateData *bistate, List *arbiterIndexes,
                 EState *estate, LockTupleMode lockmode,
                 TupleTableSlot *lockedSlot, TupleTableSlot *tempSlot);
```

- Inserts new tuple or locks the conflicting tuple.
- Leaves the TableAM with enough of freedom on how to do this.
- Still need some work to evade passing EState to TableAM.

Check is tuple was created in the current transaction

```
bool
RI_FKey_fk_upd_check_required(Trigger *trigger, Relation fk_rel,
                              TupleTableSlot *oldslot,
                              TupleTableSlot *newslot)
................................
    xminDatum = slot_getsysattr(oldslot, MinTransactionIdAttributeNumber,
                                &isnull);
    Assert(!isnull);
    xmin = DatumGetTransactionId(xminDatum);
    if (TransactionIdIsCurrentTransactionId(xmin))
        return true;
```

This also seems implementation-depended. Who said we have xmin?

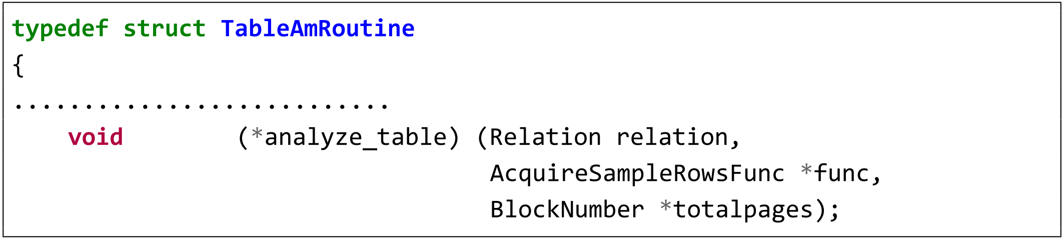
tuple_is_current() patch

```
typedef struct TableAmRoutine
{
...........................
    bool (*tuple_is_current) (Relation rel, TupleTableSlot *slot);
```

```
bool
RI_FKey_fk_upd_check_required(Trigger *trigger, Relation fk_rel,
                              TupleTableSlot *oldslot,
                              TupleTableSlot *newslot)
...........................
    if (table_tuple_is_current(fk_rel, oldslot))
        return true;
```

Checks if tuple was created in current transaction without implication on how this check should work.

Custom analyze table



```
typedef struct TableAmRoutine
{
...........................
    void        (*analyze_table) (Relation relation,
                                  AcquireSampleRowsFunc *func,
                                  BlockNumber *totalpages);
```

I think this is must have...

Bring more control on how we use indexes to TableAM

```
typedef struct TableAmRoutine
{
...........................
    TupleTableSlot *(*tuple_insert) (Relation rel, TupleTableSlot *slot,
                                 CommandId cid, int options,
                                 struct BulkInsertStateData *bistate,
                                 bool *insert_indexes);
    void        (*multi_insert) (Relation rel, TupleTableSlot **slots,
                                 int nslots, CommandId cid, int options,
                                 struct BulkInsertStateData *bistate,
                                 bool *insert_indexes);
```

Bring more control on how we use indexes to TableAM

```
typedef struct TableAmRoutine
{
...........................
    bool        (*define_index_validate) (Relation rel, IndexStmt *stmt,
                                          bool skip_build, void **arg);

    bool        (*define_index) (Relation rel, Oid indoid, bool reindex,
                                 bool skip_constraint_checks, bool skip_build,
                                 void *arg);
```

Heavily needed if we override the index implementations.

Control both table and index options

```
typedef struct TableAmRoutine
{
...........................
    bytea       *(*reloptions) (char relkind, Datum reloptions,
                               bool validate);

    bytea       *(*indexoptions) (amoptions_function amoptions, char relkind,
                                 Datum reloptions, bool validate);
```

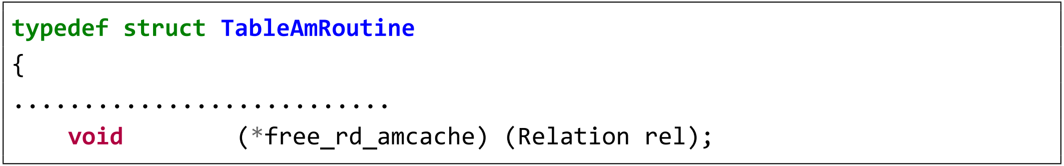
TableAM might need to add more options for indexes.

rd_amcache

```
typedef struct RelationData
{
...........................
    /*
     * If used, it must point to a single memory chunk palloc’d in
     * CacheMemoryContext ..........
     */
    void        *rd_amcache;    /* available for use by index/table AM */
```

- Too restrictive...

rd_amcache patch



```
typedef struct TableAmRoutine
{
...........................
    void        (*free_rd_amcache) (Relation rel);
```

- Custom free method
- Allows usage of complex data structures in cache

OrioleDB roadmap

- Basic engine features ✔
- Table AM interface implementation ✔
- Data compression ✔
- Undo log ✔
- TOAST support ✔
- Parallel row-level replication ✔
- Partial and expression indexes ✔
- Parallel scan ✔
- **Beta release** ✔
- pg_rewind
- Production quality
- Vectorization
- GiST/GIN analogues
- Multimaster

OrioleDB status

- Beta version is released;
- Extensibility patches: `https://github.com/orioledb/postgres/commits/patches16`;
- OrioleDB extension: `https://github.com/orioledb/orioledb`;
- Try it, test it, benchmark it, report issues;
- Sponsor it `https://github.com/sponsors/orioledb`