Pluggable Table Storage in PostgreSQL

Andres Freund

PostgreSQL Developer & Committer

Email: andres@anarazel.de

Email: andres.freund@enterprisedb.com

Twitter: @AndresFreundTec

anarazel.de/talks/2019-05-30-pgcon-pluggable-table-storage/pluggable.pdf

Pluggable Table Storage / tableam

CREATE TABLE …(...) USING heap;

In collaboration with:

- Haribabu Kommi
- Alvaro Herrera
- Ashutosh Bapat
- Alexander Korotkov
- Amit Khandekar
- Dmitry Dolgov

What do you mean: table storage

- Contents of a `TABLE / MATERIALIZED VIEW`
- **NOT** contents of indexes
- Not purely a change of IO layer

What do you mean: pluggable

```
CREATE EXTENSION magic_storage;
CREATE TABLE something (…) USING magic_storage;
SET default_table_access_method = ‘magic_storage’;
CREATE TABLE else (…); -- still uses magic_storage
```

Why?

- ZHeap – UNDO based storage, address bloat and write amplification problems
- Columnar Storage
- Experiments

Why not?

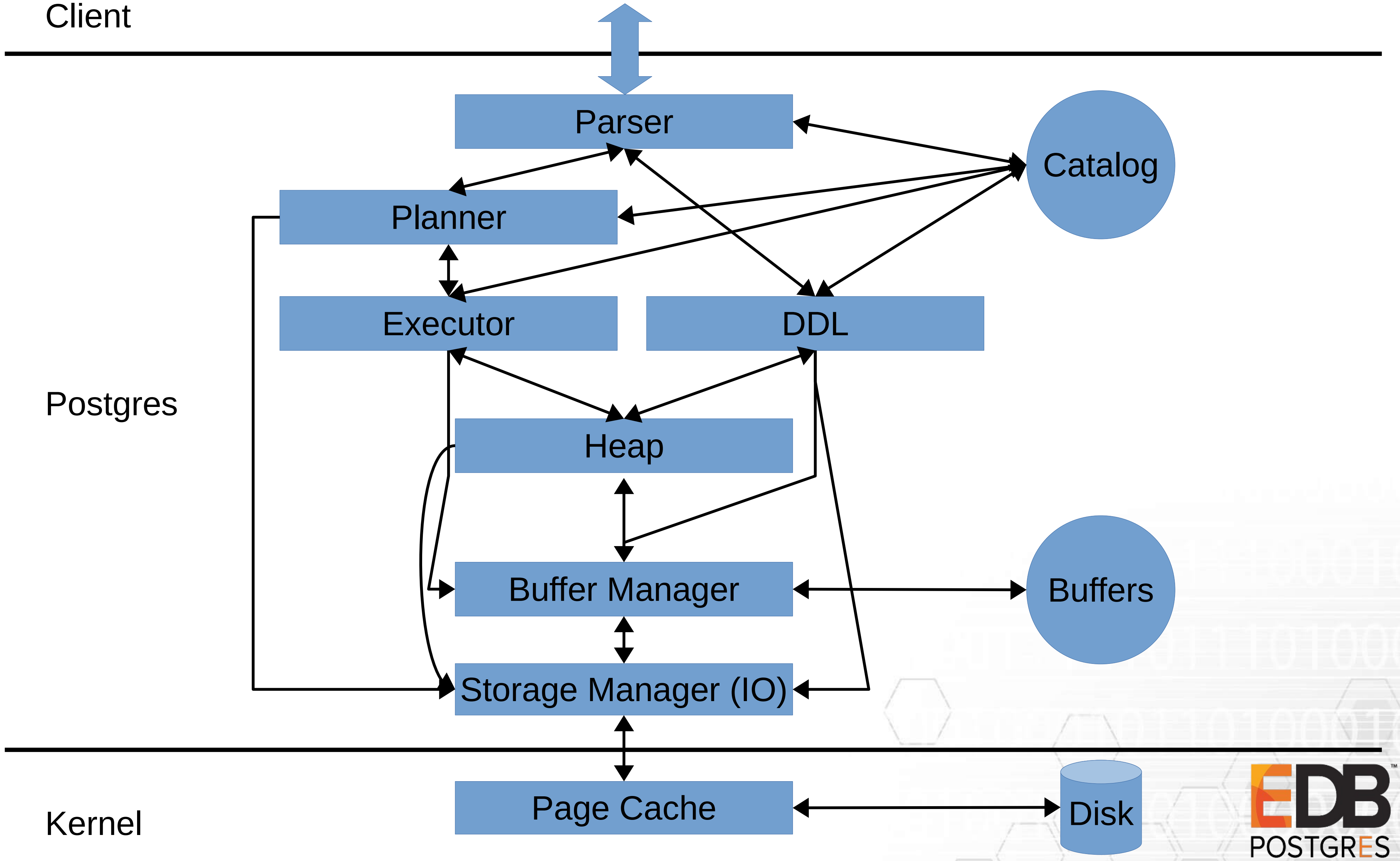
- Proliferation of half-baked storage engines, rather than one or two good ones
- Proliferation of closed & commercial storage engines
- Architectural impact

What?

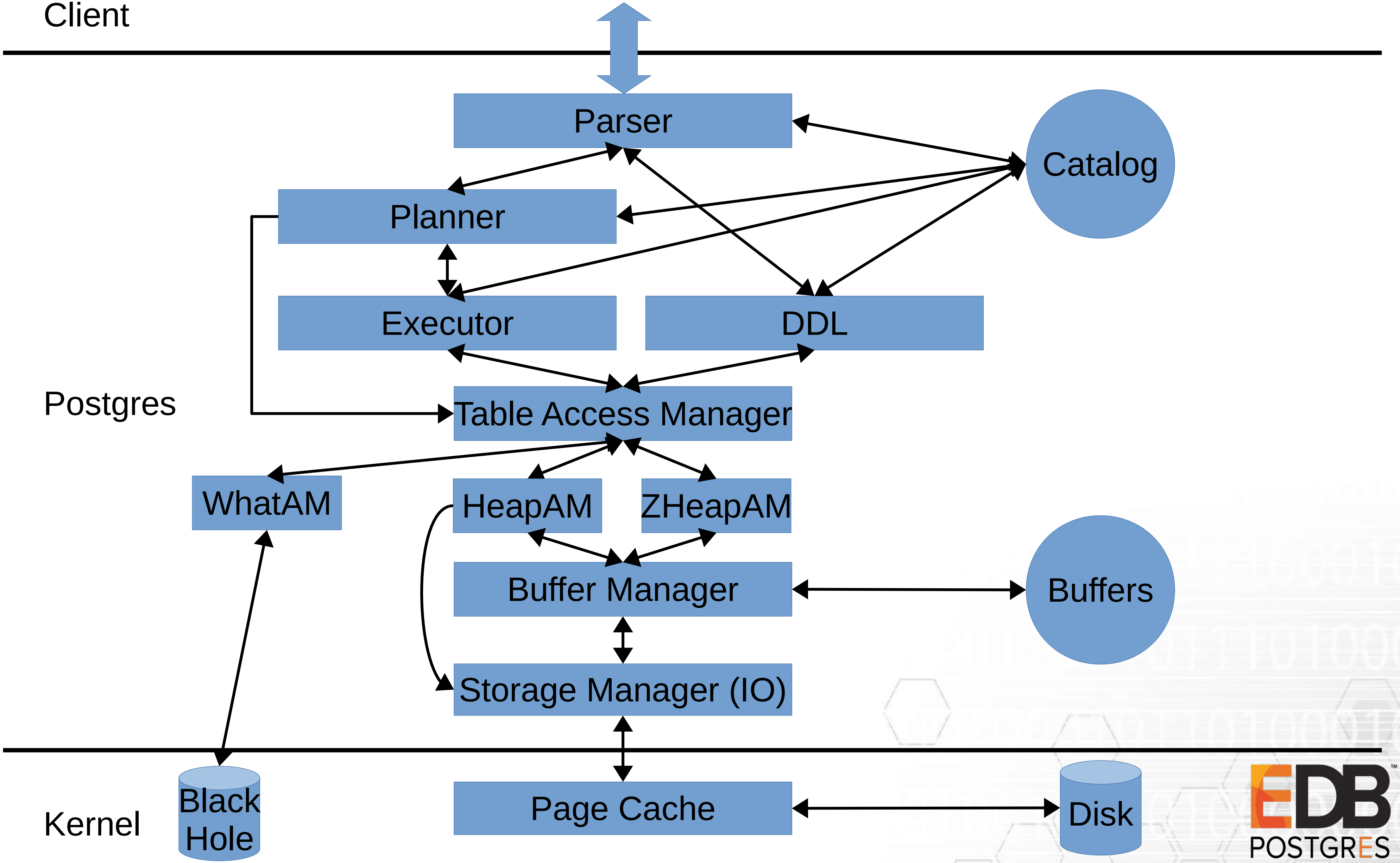
- Multiple table AMs should be able to exist at compile time
- new table AMs can be added at runtime (i.e. `CREATE EXTENSION new_am;`)
- Indexes: Should work across different table AMs
- Planner: Should work largely unmodified against different AMs
- NOT: stable API for near-term future
- NOT: Fully extensible WAL logging
- NOT: non-heap catalog tables
- NOT: Executor/Planner magic to make every storage method super fast

Contrast to Foreign Data Wrapper API

- FDWs basically hook in at the planner level
- FDWs not intended to locally store data
- Transactional Integration
- FDWs do not really support DDL
- Foreign Keys not supported (and it doesn’t really make sense to support)
- Different goals, but some overlap exists



EDB POSTGRES



Client

Parser

Catalog

Planner

Executor

DDL

Postgres

Table Access Manager

WhatAM

HeapAM

ZHeapAM

Buffer Manager

Buffers

Storage Manager (IO)

Kernel

Black Hole

Page Cache

Disk

Table AM Handlers

```
postgres[28850][1]=# SELECT * FROM pg_am WHERE amtype = 't';
```

amname	amhandler	amtype
heap	heap_tableam_handler	t

```
(1 row)
```

```
postgres[28850][1]=# \df heap_tableam_handler
                                  List of functions
```

Schema	Name	Result data type	Argument data types	Type
pg_catalog	heap_tableam_handler	table_am_handler	internal	func

```
(1 row)
```

Table AM Handlers

```
Datum
heap_tableam_handler(PG_FUNCTION_ARGS)
{
    PG_RETURN_POINTER(&heapam_methods);
}

static const TableAmRoutine heapam_methods = {
    .type = T_TableAmRoutine,

    .slot_callbacks = heapam_slot_callbacks,
…
};
```

Table AM API – DML & DDL

```
/*
 * API struct for a table AM.  Note instances of this this must be
 * allocated in a server-lifetime manner, typically as a static const struct.
 */
typedef struct TableAmRoutine
{
…
    void     (*tuple_insert) (Relation rel, TupleTableSlot *slot,
                              CommandId cid, int options,
                              struct BulkInsertStateData *bistate);
…
    void     (*relation_set_new_filenode) (Relation rel,
                                           const RelFileNode *newrnode,
                                           char persistence,
                                           TransactionId *freezeXid,
                                           MultiXactId *minmulti);
    void     (*relation_vacuum) (Relation onerel,
                                 struct VacuumParams *params,
                                 BufferAccessStrategy bstrategy);
…
    double   (*index_build_range_scan) (…);
…
}            TableAmRoutine;
```

Table AM API – Scans

```
typedef struct TableAmRoutine
{
…
    TableScanDesc (*scan_begin) (Relation rel,
                                 Snapshot snapshot,
                                 int nkeys, struct ScanKeyData *key,
                                 ParallelTableScanDesc pscan,
                                 uint32 flags);
    TableScanDesc (*scan_begin) (Relation rel,
                                 Snapshot snapshot,
                                 int nkeys, struct ScanKeyData *key,
                                 ParallelTableScanDesc pscan,
                                 uint32 flags);
…
    bool     (*scan_bitmap_next_block) (TableScanDesc scan,
                                        struct TBMIterateResult *tbmres);
    bool     (*scan_bitmap_next_tuple) (TableScanDesc scan,
                                        struct TBMIterateResult *tbmres,
                                        TupleTableSlot *slot);
…
} TableAmRoutine;
```

Infrastructure Changes

- Remove WITH OIDs support
- Generalize tuple slots
  - different types of tuples have different storage requirements
  - lots of rote changes
  - More complex slot changes:
    - Triggers
    - EvalPlanQual
    - Fix discrepancies between “declared” type of slot, and actually returned slot types
    - COPY
- Non-trivial changes to route things through tableam:
  - Executor:
    - Bitmap Scan
    - Sample Scan
  - DDL
    - ALTER TABLE SET TABLESPACE
    - ANALYZE
    - CLUSTER / VACUUM FULL
    - VACUUM
- Other changes
  - error checks in extensions like pageinspect

What’s Bad

- Function naming very confused (heap_ in a lot of functions unrelated to heap)
- Unnecessary conversions to/from HeapTuple
  - particularly around triggers
- Index only & bitmap scan nodes access visibilitymap
- pg_relation_size() looks at filesystem, rather than go through AM

Limits: WAL Logging

- introducing new WAL record types not possible without patching core code
- `access/generic_xlog.h` isn’t fast / small / capable enough
- need proper extensible WAL
  - dynamic registry problem repeatedly debated, not easy
  - static registry in core?

Limits: TID Format

```
typedef struct ItemPointerData
{
   BlockIdData ip_blkid;  // 4 bytes
   OffsetNumber ip_posid; // 2 bytes
}
```

- limits table size
- limits type of storage (no index organized table)
- What’s needed to fix:
  - Invent smart variable width encoding
  - Have space for wider TIDs in indexes
    - also helps global indexes and indirect indexes
  - Change lots of functions to accept variable width functions

Limits: Planner / Executor Integration

- Improvements particularly needed for efficiency for some storage types (columnar)
  - can partially be addressed via planner hooks + custom executor nodes
- Scans need to know the to-be-accessed columns
  - very important for columnar, but even interesting for heap
  - should be integrated with both AM and various slot types
- Costing improvements for individual AMs
  - can partially be “addressed” by skewing returned planner estimates

Limit: relation filenodes & forks

- relation forks not necessarily a good idea / set of forks not enough for all AMs
- only one relation fork for each pg_class entry
- to be integrated into base backups, files have to be in the traditional directories

Limit: Catalog on non-heap AM

- lots of rote code changes needed to not assume heap
- struct / table content mapping
- assumptions about precise transactional behavior (e.g. cache invalidation via xmin checks)

Limit: AMs assumed to be block based

- Planner cost estimates in blocks
- Analyze sampling is block based