PostgreSQL extendability: Origins and new horizonts

Towards pluggable storage engines

Alexander Korotkov

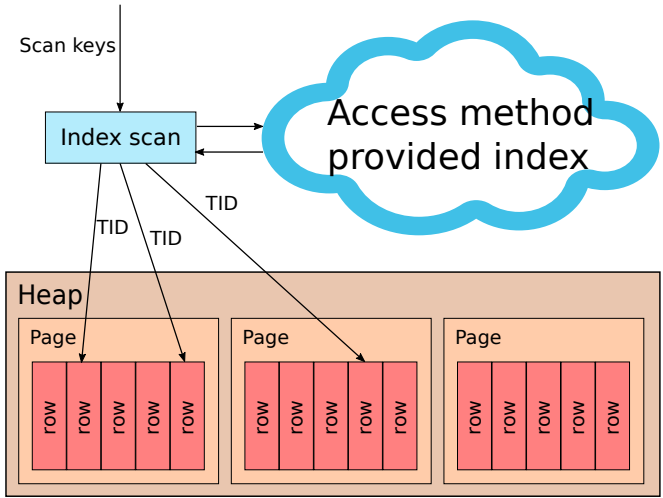
Postgres Professional

2016

How access method was designed by Berkeley?

- It is some abstraction which provides the way to scan the table. Initially heap was just one of access methods.
- Now heap is built-in too deep. In fact there is no abstraction: primary storage of table is always heap.
- Now there are two other ways to retrieve tuples: FDW and custom nodes. By the nature they could be access methods, but by design they aren't.

What is index access method?



What is index access method?

- It is some abstraction which provide us indexes using given documented API: `http://www.postgresql.org/docs/9.5/static/indexam.html`.
- Index is something that can provide us set of tuples TIDs satisfying some set of restrictions faster than sequential scan of heap can do this.
- Internally most of indexes are kind of trees. But it is not necessary so. HASH and BRIN are examples of in-core non-tree index AM.

Which non-index access methods could be?

- Sequential access methods: implement complex strategies for generation of distributed sequences.
  - http://www.postgresql.org/message-id/CA+U5nMLV3ccdzbqCvcedd-HfrE4dUmoFmTBPL_uJ9YjsQbR7iQ@mail.gmail.com
- Columnar access methods: implement columnar storage of data.
  - http://www.postgresql.org/message-id/20150611230316.GM133018@postgresql.org
  - http://www.postgresql.org/message-id/20150831225328.GM2912@alvherre.pgsql

Why access method extendability?

Postgres was designed so...

”It is imperative that a user be able to construct new access methods to provide efficient access to instances of nontraditional base types”

*M*ichael Stonebraker, Jeff Anton, Michael Hirohama.

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

How did we loose it?

- Other object of system catalog received CREATE/ALTER/DROP commands while access methods didn’t.
- When WAL was introduced, it came with fixed table of resource managers. Loaded module can’t add its own resource manager.

People want bleeding-edge features...

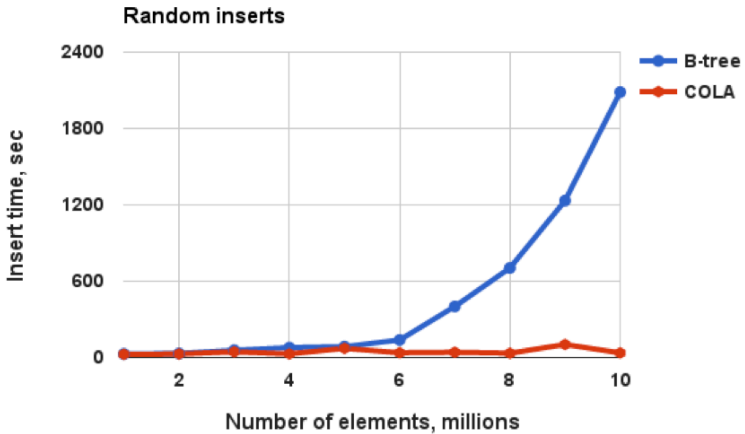
- Fast FTS was presented in 2012, but only 2 of 4 GIN improvements are committed yet.
- Fast-write indexes are arriving: LSM/Fractal Trees, COLA etc.

Fast FTS for 9.3...

	Without patch	With patch	Sphinx
Table size	6.0 GB	6.0 GB	
Index size	1.29 GB	1.27 GB	1.12 GB
Index build time	216 sec	303 sec	180 sec
Queries in 8 hours	**3,0 mln.**	**42.7 mln.**	**32.0 mln**

Only 2 of 4 GIN improvements are committed yet. GIN isn't yet as cool as we wish yet.

Cache Oblivious Lookahead Array (COLA)



New access method interface

How does access method interface look like?

In the docs

```
IndexBuildResult *ambuild (Relation heapRelation, Relation indexRelation,
                           IndexInfo *indexInfo);
void ambuildempty (Relation indexRelation);
bool aminsert (Relation indexRelation, Datum *values,
               bool *isnull, ItemPointer heap_tid,
               Relation heapRelation, IndexUniqueCheck checkUnique);
IndexBulkDeleteResult *ambulkdelete (IndexVacuumInfo *info,
       IndexBulkDeleteResult *stats, IndexBulkDeleteCallback callback,
       void *callback_state);
.........................................
```

In the system catalog

```
internal btbuild(internal, internal, internal)
void btbuildempty(internal)
boolean btinsert(internal, internal, internal, internal, internal, internal)
internal btbulkdelete(internal, internal, internal, internal)
.........................................
```

What is the problem with access method interface?

- Most of datatypes used in arguments and return values are C-structures and pointers. These datatypes don't have SQL-equivalents. This is why they are declared as ”internal”.
- None of interface functions are going to be SQL-callable. None of them are going to be implemented not in C.
- Once we have extendable access methods, interface may change. We could have extra difficulties with, for instance, additional ”internal”which is to be added to function signature.

Another approach: handlers

Handler hide all guts from SQL.

```
CREATE FOREIGN DATA WRAPPER file HANDLER file_fdw_handler;
```

```
Datum
file_fdw_handler(PG_FUNCTION_ARGS)
{
    FdwRoutine *fdwroutine = makeNode(FdwRoutine);

    fdwroutine->GetForeignRelSize = fileGetForeignRelSize;
    fdwroutine->GetForeignPaths = fileGetForeignPaths;
    fdwroutine->GetForeignPlan = fileGetForeignPlan;
.........................................
    fdwroutine->EndForeignScan = fileEndForeignScan;
    fdwroutine->AnalyzeForeignTable = fileAnalyzeForeignTable;

    PG_RETURN_POINTER(fdwroutine);
}
```

Access method handlers

If we would have access method handlers like this

```
Datum
bthandler(PG_FUNCTION_ARGS)
{
    IndexAmRoutine *amroutine = makeNode(IndexAmRoutine);
    amroutine->amstrategies = 5;
    amroutine->amsupport = 2;
    amroutine->amcanorder = true;
.................................
    amroutine->aminsert = btinsert;
    amroutine->ambeginscan = btbeginscan;
    amroutine->amgettuple = btgettuple;
.................................
    PG_RETURN_POINTER(amroutine);
}
```

then it would be easy to define new access method

```
CREATE ACCESS METHOD btree HANDLER bthandler;
```

pg_am

Before:

Column	Type	Modifiers
amname	name	not null
amstrategies	smallint	not null
amsupport	smallint	not null
amcanorder	boolean	not null
amcanorderbyop	boolean	not null
amcanbackward	boolean	not null
amcanunique	boolean	not null
amcanmulticol	boolean	not null
amoptionalkey	boolean	not null
amsearcharray	boolean	not null
amsearchnulls	boolean	not null
.....20 more columns.....		

After:

Column	Type	Modifiers
amname	name	not null
amhandler	regproc	not null

pg_am becomes suitable to store other access methods: sequential, columnar etc.

Access method procedures signatures

Before:

```
Datum
btinsert(PG_FUNCTION_ARGS)
```

After:

```
bool
btinsert(Relation rel, Datum *values, bool *isnull,
         ItemPointer ht_ctid, Relation heapRel,
         IndexUniqueCheck checkUnique)
```

Signatures of access method procedures becomes more meaningful.

Operator classes validation

There were some regression tests which rely on exposing index access method in pg_am.

```
-- Cross-check amprocnum index against parent AM

SELECT p1.amprocfamily, p1.amprocnum, p2.oid, p2.amname
FROM pg_amproc AS p1, pg_am AS p2, pg_opfamily AS p3
WHERE p1.amprocfamily = p3.oid AND p3.opfmethod = p2.oid AND
    p1.amprocnum > p2.amsupport;
```

Now opclasses validation is up to index access method.

```
/* validate oplass */
typedef void
(*amvalidate_function) (OpClassInfo *opclass);
```

New access method interface status

Committed!

http://git.postgresql.org/gitweb/?p=postgresql.git;a=commit;h=65c5fcd353a859da9e61bfb2b92a99f12937de3b

Why do we need CREATE ACCESS METHOD command?

Could we be satisfied with this?

```
INSERT INTO pg_am (
    amname,
    amhandler
) VALUES (
    'bloom',
    'blhandler'
);
```

No, because pg_upgrade will wash that away. We need command like this with pg_dump support.

```
-- Access method
CREATE ACCESS METHOD bloom HANDLER blhandler;
```

Reliability problems of custom access methods

- AM can crash during index search, build or insert. Opclass can behave the same, not AM-specific problem.
- AM can corrupt index and/or give wrong answers to queries. Opclass can behave the same, not AM-specific problem.
- AM can crash during vacuum. Autovacuum could run into cycle of crashes. **That is AM-specific problem**.
- AM can crash in WAL replay during recovery or replication. **That is AM-specific problem**.

Vacuum problem

- Vacuum crash isn’t any worse than crash during index search, build or insert.
- Cycle autovacuum crash is worse because it doesn’t require explicit user actions.
- We can mark custom indexes with some flag on crash in vacuum. Then autovacuum will skip it until user explicitly unset this flag.

WAL redo functions

src/include/access/rmgrlist.h

```
PG_RMGR(RM_XLOG_ID, ”XLOG”, xlog_redo, xlog_desc, xlog_identify, NULL, NULL)
PG_RMGR(RM_XACT_ID, ”Transaction”, xact_redo, xact_desc, xact_identify, NULL, NU
PG_RMGR(RM_SMGR_ID, ”Storage”, smgr_redo, smgr_desc, smgr_identify, NULL, NULL)
PG_RMGR(RM_CLOG_ID, ”CLOG”, clog_redo, clog_desc, clog_identify, NULL, NULL)
PG_RMGR(RM_DBASE_ID, ”Database”, dbase_redo, dbase_desc, dbase_identify, NULL, N
PG_RMGR(RM_TBLSPC_ID, ”Tablespace”, tblspc_redo, tblspc_desc, tblspc_identify, N
PG_RMGR(RM_MULTIXACT_ID, ”MultiXact”, multixact_redo, multixact_desc, multixact_
PG_RMGR(RM_RELMAP_ID, ”RelMap”, relmap_redo, relmap_desc, relmap_identify, NULL,
PG_RMGR(RM_STANDBY_ID, ”Standby”, standby_redo, standby_desc, standby_identify,
.....................................................................
```

WAL problem

- WAL replay is critical for reliability because it is used for both recovery, continuous archiving and streaming replication. This is why making WAL replay depend on custom extension is not an option.
- Universal **generic WAL format** could be an option. It should do maximum checks before writing WAL-record in order to exclude error during replay.

Generic WAL interface

Custom access method in extension should make generic WAL records as following.

- `GenericXLogStart(index)` – start usage of generic WAL for specific relation.
- `GenericXLogRegister(buffer, false)` – register specific buffer for generic WAL record. Second argument indicating new buffer.
- `GenericXLogFinish()` or `GenericXLogAbort()` – write generic WAL record or abort with reverting page state.

Generic xlog takes care about critical section, unlogged relation, setting lsn, making buffer dirty. User code is just simple and clear.

Generic WAL usage example (1/2): Init bloom metapage

```
/* initialize the meta page */
metaBuffer = BloomNewBuffer(index);
GenericXLogStart(index);
GenericXLogRegister(metaBuffer, true);
BloomInitMetabuffer(metaBuffer, index);
GenericXLogFinish();
UnlockReleaseBuffer(metaBuffer);
```

Generic WAL usage example (2/2): Try to insert new item to the page

```
buffer = ReadBuffer(index, blkno);
LockBuffer(buffer, BUFFER_LOCK_EXCLUSIVE);
GenericXLogStart(index);
GenericXLogRegister(buffer, false);
if (BloomPageAddItem(&blstate, BufferGetPage(buffer), itup))
    /* Item was successfully added: finish WAL record */
    GenericXLogFinish();
else
    /* Item wasn't added: abort WAL record */
    GenericXLogAbort();
UnlockReleaseBuffer(buffer);
```

Complete example: bloom filter index (1/2)

```
# CREATE TABLE tst AS (
      SELECT (random()*100)::int AS i,
             substring(md5(random()::text), 1, 2) AS t
      FROM generate_series(1, 1000000));

# EXPLAIN (ANALYZE, BUFFERS) SELECT * FROM tst
                             WHERE i = 16 AND t = 'af';
 Seq Scan on tst  (cost=0.00..19425.00 rows=25 width=36)
                  (actual time=0.285..74.322 rows=31 loops=1)
   Filter: ((i = 16) AND (t = 'af'::text))
   Rows Removed by Filter: 999969
   Buffers: shared hit=192 read=4233
 Planning time: 0.156 ms
 Execution time: 74.354 ms
```

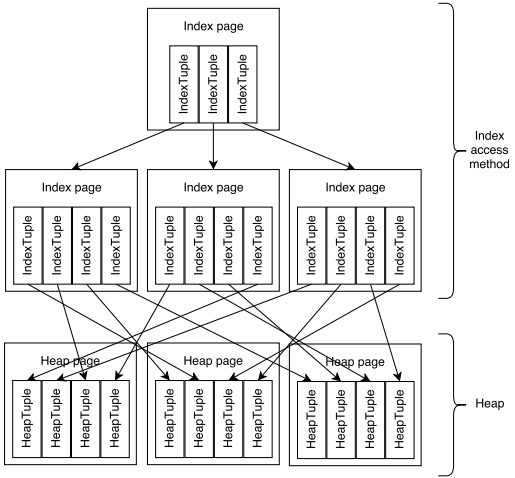
Complete example: bloom filter index (2/2)

```
# CREATE INDEX tst_i_t_idx ON tst USING bloom (i, t)
                           WITH (col1 = 5, col2 = 11);

# EXPLAIN (ANALYZE, BUFFERS) SELECT * FROM tst
                             WHERE i = 16 AND t = 'af';

 Bitmap Heap Scan on tst  (cost=17848.01..17942.74 rows=25 width=36)
                          (actual time=4.705..4.948 rows=31 loops=1)
   Recheck Cond: ((i = 16) AND (t = 'af'::text))
   Heap Blocks: exact=31
   Buffers: shared hit=1962 read=30
   ->  Bitmap Index Scan on tst_i_t_idx
           (cost=0.00..17848.00 rows=25 width=0)
           (actual time=4.650..4.650 rows=31 loops=1)
         Index Cond: ((i = 16) AND (t = 'af'::text))
         Buffers: shared hit=1961
 Planning time: 0.211 ms
 Execution time: 5.000 ms
```

CREATE ACCESS METHOD + generic WAL + bloom contrib

- Patch is on the commitfest `https://commitfest.postgresql.org/6/353/`.
- Got some review.
- Hopefully will be committed to 9.6.

Pluggable heap?



Could we replace heap a well?

Pluggable table engines concept

Owns

- Ways to scan and modify.
- Access methods implementations.

Other wise it wouldn't be distinct pluggable table engines.

Shares

- Transactions, snapshots.
- WAL.

Other wise it wouldn't be part of PostgreSQL.

Why table engine is not just FDW?

- AM – one can’t CREATE INDEX on access method.
- WAL – FDWs are not WAL-logged.
- VACUUM – FDWs don’t need VACUUMing.
- File node – FDWs don’t have regular way to associate files with them.

Index access methods and table engines

Table engines have their own index access methods implementations. But sharing opclasses would be useful. Everything related to opclass validation leaves in pg_am. Everything related to scan, build, insert etc goes to pg_am_impl – only table engine deals with that.

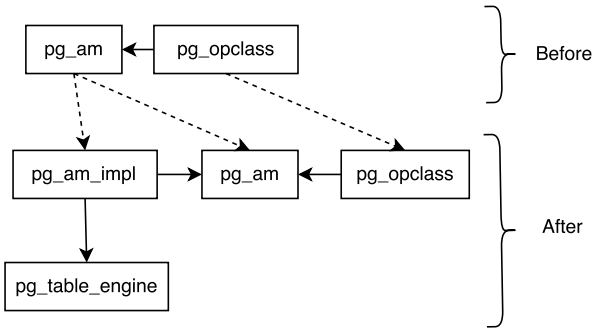


Table engines and WAL

- Generic WAL records could be solution for some cases.
- In other cases, custom redo functions are definitely needed. For instance, in-memory tables with persistence. On-disk representation: snapshot + logical deltas(in WAL).

Table engines and VACUUM

- Not mandatory. Some table engines wouldn’t need VACUUM.
- Track relminmxid and relfrozenxid if xids are used.
- Table engines are responsible for its indexes VACUUM as well.

Pluggable everything!

Postgres PROFESSIONAL

Thank you for attention!