Access method extendability in PostgreSQL or back to origin

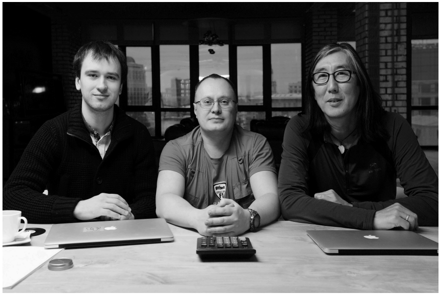
Alexander Korotkov, Oleg Bartunov, Teodor Sigaev

Postgres Professional

2015

Russian developers of PostgreSQL:

Alexander Korotkov, Teodor Sigaev, Oleg Bartunov



PostgreSQL CORE

- Locale support
- PostgreSQL extendability: GiST(KNN), GIN, SP-GiST
- Full Text Search (FTS)
- NoSQL (hstore, jsonb)
- Indexed regexp search
- VODKA access method (WIP)

Extensions

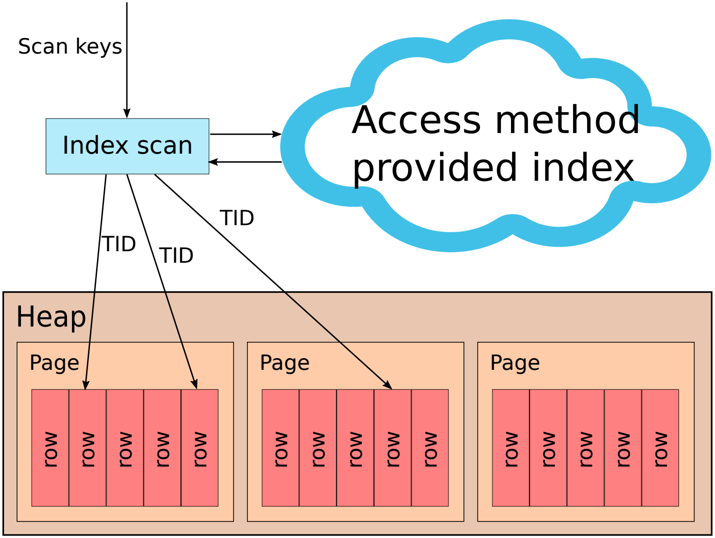
- intarray
- pg_trgm
- ltree
- hstore
- plantuner
- jsquery

- Speakers at PGCon, PGConf: 20+ talks
- GSoC mentors
- PostgreSQL committers (1+1 in progress)
- Conference organizers
- 50+ years of PostgreSQL expertship: development, audit, consulting
- Novartis, Raining Data, Heroku, Engine Yard, WarGaming, Rambler, Avito, 1C

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...

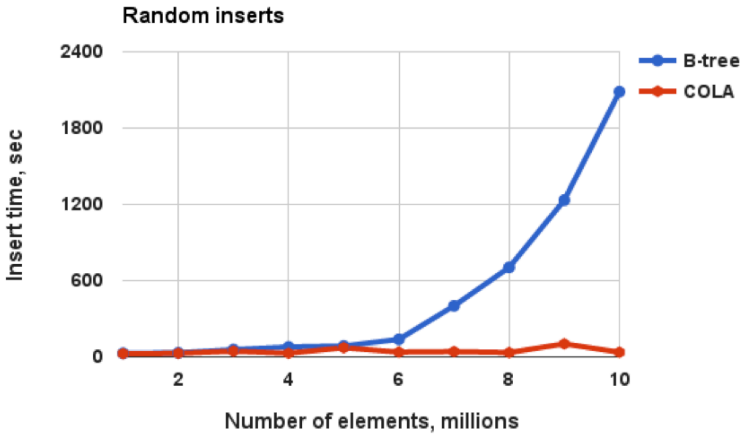
- Fast FTS was presented in 2012, but only 2 of 4 GIN improvements are committed yet.
- Fast-write indexes are arriving: LSM/Fractal Trees, COLA etc.

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->ExplainForeignScan = fileExplainForeignScan;
    fdwroutine->BeginForeignScan = fileBeginForeignScan;
    fdwroutine->IterateForeignScan = fileIterateForeignScan;
    fdwroutine->ReScanForeignScan = fileReScanForeignScan;
    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

- Patch is on the commitfest `https://commitfest.postgresql.org/6/336/`.
- Got pretty much of review.
- Hopefully will be committed soon.

What are limitations of access method interface?

Which clauses can be given to index scan?

- `WHERE column OPERATOR value` – represented as array of ScanKeys which are implicitly ANDed.
- `ORDER BY column OPERATOR value` – represented as array of ScanKeys, one for each clause in sequence.
- `ORDER BY column (ASC|DESC)` – index assumed to return ordered data when amcanorder is true.

How did we workaroung this in the past?

In historical order:

- query_int
- tsquery
- lquery
- jsquery

What are drawbacks of custom query types?

- Closed for planner
  - JsQuery has its own rule-based mini-planner without any statistics.
  - Other haven't any kind of planner/optimizer.
  - That causes problems with frequent/rare values.
  - Partially fixed in GIN in 9.4.
- No extensibility
  - Jsquery – could not add new operators/types.
  - query_int – could not be extended to support new types (intarray $\rightarrow$ anyarray)

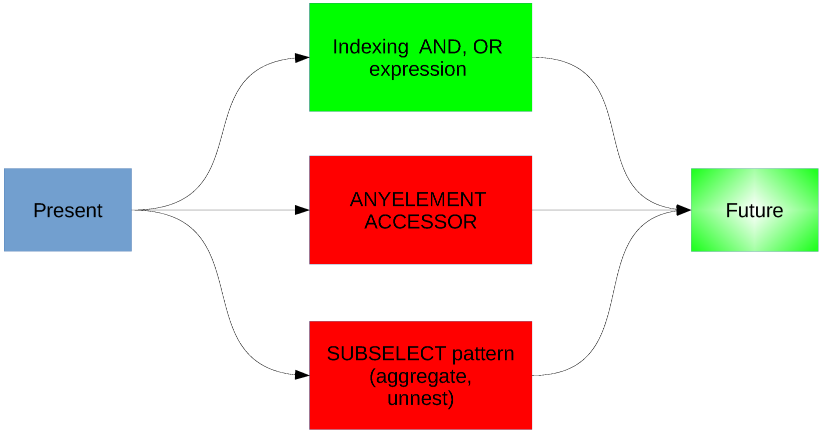
```
SELECT ...
WHERE
  ANY ELEMENT OF
    array_or_json_column AS element SATISFIES
  ( expression_over_element )
```

$$\Downarrow$$

```
SELECT ... WHERE  (
  SELECT bool_or(true)
  FROM unnest(array_or_json_column) AS element
  WHERE expression_over_element )
```

We need index support for subselects!

How do we see the future?


Present
Indexing AND, OR expression
ANYELEMENT ACCESSOR
SUBSELECT pattern (aggregate, unnest)
Future

ScanKeys for index: now

Scan keys of index:

```
An index scan has zero or more scan keys,
which are implicitly ANDed.
```

ScanKeys for index: patched

Polish notation for simple stack machine. Examples:

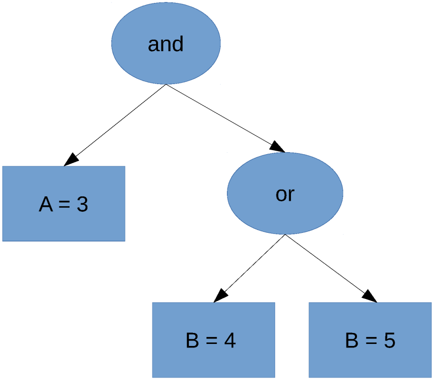
- `(a=3)(b=4)(b=5)(OR)(AND)`
- `(b=4)(b=5)(OR)(a=3)(AND)`

Now supported:

- GiST
- GIN
- BRIN

Potentially supported:

- Btree (difficult)
- SP-GiST
- Hash



GIN vs GIN

```
# EXPLAIN ANALYZE
SELECT count(*) FROM tbl WHERE (a && ’{1}’ OR a && ’{2}’) AND
                               (a && ’{99901}’ OR a && ’{99902}’);
                                                                   QUERY PLAN
-------------------------------------------------------------------------------------------------------------
 Aggregate  (cost=4503.28..4503.29 rows=1 width=0) (actual time=0.546..0.547 rows=1 loops=1)
   ->  Bitmap Heap Scan on tbl  (cost=522.79..4500.61 rows=1065 width=0) (actual time=0.541..0.542
         Recheck Cond: (((a && ’{99901}’::integer[]) OR (a && ’{99902}’::integer[])) AND ((a && ’{
         Heap Blocks: exact=1
         ->  Bitmap Index Scan on idx  (cost=0.00..522.79 rows=1097 width=0) (actual time=0.537..0
               Index Cond: (((a && ’{99901}’::integer[]) OR (a && ’{99902}’::integer[])) AND ((a &&
 Planning time: 0.151 ms
 Execution time: 0.603 ms
(8 rows)
                                                                 QUERY PLAN
-------------------------------------------------------------------------------------------------------------
 Aggregate  (cost=5139.85..5139.86 rows=1 width=0) (actual time=19.212..19.212 rows=1 loops=1)
   ->  Bitmap Heap Scan on tbl  (cost=1159.37..5137.19 rows=1065 width=0) (actual time=19.174..19.
         Recheck Cond: (((a && ’{99901}’::integer[]) OR (a && ’{99902}’::integer[])) AND ((a && ’{
         Rows Removed by Index Recheck: 2
         Heap Blocks: exact=3
         ->  BitmapAnd  (cost=1159.37..1159.37 rows=1097 width=0) (actual time=19.118..19.118 rows=
               ->  BitmapOr  (cost=131.49..131.49 rows=9995 width=0) (actual time=0.023..0.023 row
                     ->  Bitmap Index Scan on idx  (cost=0.00..65.48 rows=4998 width=0) (actual ti
                           Index Cond: (a && ’{99901}’::integer[])
                     ->  Bitmap Index Scan on idx  (cost=0.00..65.48 rows=4998 width=0) (actual ti
                           Index Cond: (a && ’{99902}’::integer[])
               ->  BitmapOr  (cost=1027.62..1027.62 rows=109745 width=0) (actual time=18.948..18.9
                     ->  Bitmap Index Scan on idx  (cost=0.00..522.79 rows=55839 width=0) (actual 
                           Index Cond: (a && ’{1}’::integer[])
                     ->  Bitmap Index Scan on idx  (cost=0.00..504.30 rows=53906 width=0) (actual 
                           Index Cond: (a && ’{2}’::integer[])
 Planning time: 0.141 ms
 Execution time: 19.274 ms
(18 rows)
```

GIN vs BTREE

```
EXPLAIN ANALYZE
SELECT count(*) FROM tst WHERE id = 5 OR id = 500 OR id = 5000;
                                                             QUERY PLAN
--------------------------------------------------------------------------------------------------------------
 Aggregate  (cost=10000004782.31..10000004782.32 rows=1 width=0)
            (actual time=0.279..0.279 rows=1 loops=1)
   ->  Bitmap Heap Scan on tst  (cost=10000000057.50..10000004745.00 rows=14925 width=0)
                                (actual time=0.080..0.267 rows=173 loops=1)
         Recheck Cond: ((id = 5) OR (id = 500) OR (id = 5000))
         Heap Blocks: exact=172
         ->  Bitmap Index Scan on idx_gin  (cost=0.00..57.50 rows=15000 width=0)
                                           (actual time=0.059..0.059 rows=147 loops=1)
               Index Cond: ((id = 5) OR (id = 500) OR (id = 5000))
 Planning time: 0.077 ms
 Execution time: 0.308 ms

EXPLAIN ANALYZE
SELECT count(*) FROM tst WHERE id = 5 OR id = 500 OR id = 5000;
                                                            QUERY PLAN
--------------------------------------------------------------------------------------------------------------
 Aggregate  (cost=51180.53..51180.54 rows=1 width=0)
            (actual time=796.766..796.766 rows=1 loops=1)
   ->  Index Only Scan using idx_btree on tst  (cost=0.42..51180.40 rows=55 width=0)
                                               (actual time=0.444..796.736 rows=173 loops=1)
         Filter: ((id = 5) OR (id = 500) OR (id = 5000))
         Rows Removed by Filter: 999829
         Heap Fetches: 1000002
 Planning time: 0.087 ms
 Execution time: 796.798 ms
```

GIN vs SeqScan

```
EXPLAIN ANALYZE
SELECT count(*) FROM tst WHERE id = 5 OR id = 500 OR id = 5000;
                                                             QUERY PLAN
------------------------------------------------------------------------------------------------------------
 Aggregate  (cost=10000004782.31..10000004782.32 rows=1 width=0)
            (actual time=0.279..0.279 rows=1 loops=1)
   ->  Bitmap Heap Scan on tst  (cost=10000000057.50..10000004745.00 rows=14925 width=0)
                                (actual time=0.080..0.267 rows=175 loops=1)
         Recheck Cond: ((id = 5) OR (id = 500) OR (id = 5000))
         Heap Blocks: exact=172
         ->  Bitmap Index Scan on idx_gin  (cost=0.00..57.50 rows=15000 width=0)
                                           (actual time=0.059..0.059 rows=147 loops=1)
               Index Cond: ((id = 5) OR (id = 500) OR (id = 5000))
 Planning time: 0.077 ms
 Execution time: 0.308 ms
                                                  QUERY PLAN
------------------------------------------------------------------------------------------------------------
 Aggregate  (cost=21925.63..21925.64 rows=1 width=0)
            (actual time=160.412..160.412 rows=1 loops=1)
   ->  Seq Scan on tst  (cost=0.00..21925.03 rows=237 width=0)
                        (actual time=0.535..160.362 rows=175 loops=1)
         Filter: ((id = 5) OR (id = 500) OR (id = 5000))
         Rows Removed by Filter: 999827
 Planning time: 0.459 ms
 Execution time: 160.451 ms
```

GiST OR-ed + KNN (1/1)

```
FIND 10 closest to (0,45) POI from Antarctica and Arctica.

Total: 7240858 POI
```

```
EXPLAIN ANALYZE SELECT name,point FROM geo
WHERE point <@ circle('(90,90)',10) OR
      point <@ circle('(-90,90)',10)
ORDER BY point <-> '(0,45)' LIMIT 10;
```

Individual queries are blazing fast, thanks to KNN !

```
select name,point from geo where point <@ circle('(90,90)',10)
order by point <-> '(0,45)' limit 10;
0.561 ms
```

```
select name,point from geo where point <@ circle('(-90,90)',10)
order by point <-> '(0,45)' limit 10;
0.454 ms
```

GiST OR-ed + KNN (2/2)

NEW:

```
 Limit  (cost=0.41..322.73 rows=1 width=34)
        (actual time=0.123..0.147 rows=10 loops=1)
   ->  Index Scan using geo_idx on geo  (cost=0.41..322.73 rows=0 width=34)
                                        (actualtime=0.121..0.144 rows=10 loops=1)
         Index Cond: ((point <@ ’<(90,90),10>’::circle) OR (point <@ ’<(-90,90),10>’::circle))
         Order By: (point <-> ’(0,45)’::point)
 Planning time: 0.097 ms
 Execution time: 0.192 ms
```

OLD:

```
 Limit  (cost=0.41..662.97 rows=10 width=30)
        (actual time=7228.580..7266.604 rows=10 loops=1)
   ->  Index Scan using geo_idx on geo  (cost=0.41..958987.91 rows=14474 width=30)
                                        (actual time=7228.570..7266.585 rows=10 loops=1)
         Order By: (point <-> ’(0,45)’::point)
         Filter: ((point <@ ’<(90,90),10>’::circle) OR (point <@ ’<(
90,90),10>’::circle))
         Rows Removed by Filter: 3956498
 Planning time: 0.300 ms
 Execution time: 7266.673 ms
```

~ 38 000 FASTER !

Index-only Scan

Index-only scan now works with OR-ed conditions:

```
Index: ”id_idx_gist” gist (id)
EXPLAIN ANALYZE SELECT id FROM geo WHERE id=1 OR id=2000;
                                                            QUERY PLA
-----------------------------------------------------------------------
 Index Only Scan using id_idx_gist on geo
      (cost=0.41..4.42 rows=0 width=4)
      (actual time=0.099..0.131 rows=2 loops=1)
   Index Cond: ((id = 1) OR (id = 2000))
   Heap Fetches: 0
 Planning time: 0.263 ms
 Execution time: 0.163 ms
(5 rows)
```

FTS+KNN works !

```
Index: ”geo_point_fts_idx” gist (point, fts)
EXPLAIN ANALYZE SELECT name, point FROM geo WHERE fts @@ to_tsq


-----------------------------------------------------------------
 Limit  (cost=0.67..34.25 rows=10 width=30)
        (actual time=0.594..2.922 rows=10 loops=1)
   ->  Index Scan using geo_point_fts_idx on geo
            (cost=0.67..136789.20 rows=40735 width=30)
            (actual time=0.593..2.921 rows=10 loops=1)
          Index Cond: (fts @@ to_tsquery(’town’) OR
                       fts @@ to_tsquery(’city’))
          Order By: (point <-> ’(0,45)’::point)
 Planning time: 0.271 ms
 Execution time: 2.955 ms
(6 rows)
```

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 something like this.

```
-- 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 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.

Thanks for attention!