PostgresPro

PGConf.Russia 2025

Реализация табличного движка с нуля. Новый TableAM для OLAP.

Алексей Гордеев
R&D engineer, Postgres Professional

PostgresPro

Зачем еще один табличный движок для OLAP?

- Citus Columnar
- Greenplum AO
- Timescale Hypertables
- Hydra (Citus Columnar fork)
- Zedstore (RIP)

- Производительность должна быть выше
- Собственные движки сложнее развивать
- Собственные форматы не подключить к сторонним исполнителям запросов

Часть 1.
Познавательная.

Разработка движка

- TableAmRoutine
- Ограничения для колоночного движка
    - Свой state для каждого INSERT
    - Нет проброса проекции
    - Нет проброса предикатов
    - Исполнение запроса не векторизовано
    - Исполнение применяет раннюю материализацию
- Полезное
    - Soft DELETE
    - Буферизация строк
    - WAL. Resource Manager.
    - Storage Manager
    - Использование аллокаторов Postgres в сторонней либе

Строковое хранение

N-ary storage model, NSM

	ID	Name	Birthdate
1	10	Ivan	14.08.1987
2	11	Peter	26.02.1984
3	12	Sidor	09.11.1995

heap1.dat

10	Ivan	14.08.1987	11
Peter	26.02.1984	...	...

heap2.dat

12	Sidor	09.11.1995	...
...	...	...	...

Postgres Heap

Колоночное хранение

Decomposition Storage Model, DSM

	ID	Name	Birthdate
1	10	Ivan	14.08.1987
2	11	Peter	26.02.1984
3	12	Sidor	09.11.1995

column1.dat

10	11	12	…

column2.dat

Ivan	Peter	Sidor	…

column3.dat

14.08.1987	26.02.1984	09.11.1995	…

Greenplum Append-Optimized

PAX

Partition Attributes Across

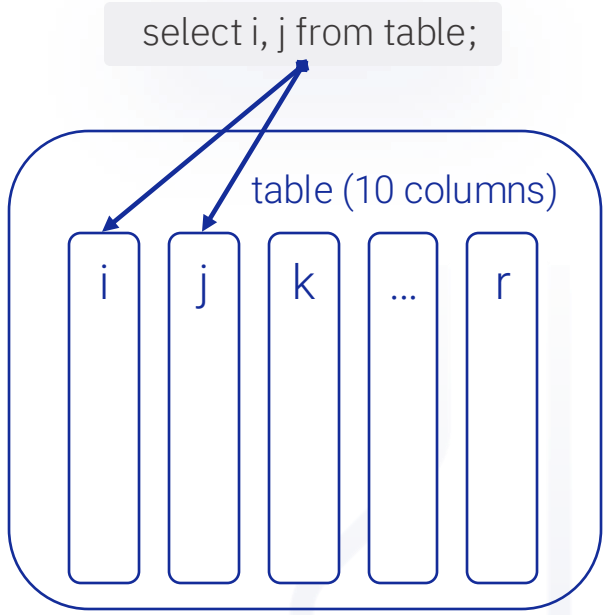
	ID	Name	Birthdate
1	10	Ivan	14.08.1987
2	11	Peter	26.02.1984
3	12	Sidor	09.11.1995

Citus Columnar

Строки vs Колонки

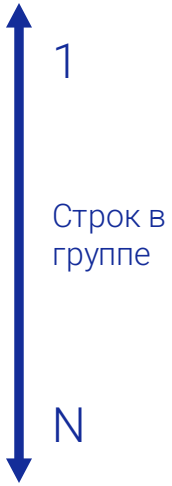
- **Строки -> OLTP**
  - SELECT одной строки
  - DML одной строки

- **Колонки -> OLAP**
  - Сжатие
  - Чтение только используемых колонок
  - +Кэш
  - Векторное исполнение, SIMD



PAX

- **Строки -> OLTP**
  - SELECT одной строки
  - DML одной строки



- **Колонки -> OLAP**
  - Сжатие
  - Чтение только используемых колонок
  - +Кэш
  - Векторное исполнение, SIMD

- **PAX**
  - Статистика и легковесные индексы (min/max, bloom)

PAX: Parquet, ORC, CarbonData ...

Начало

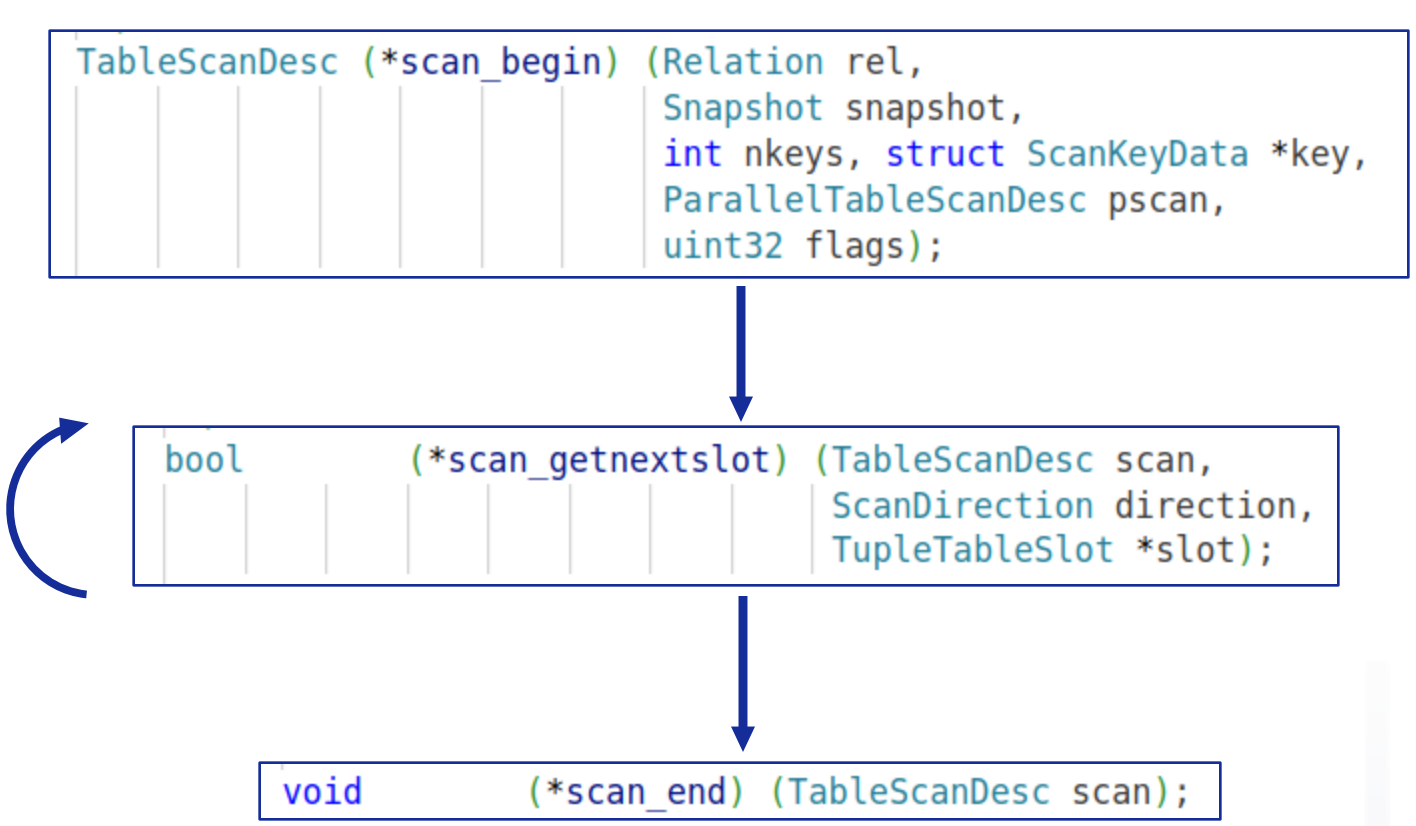
TableAmRoutine, 40+ функций

```
280  /*
281   * API struct for a table AM.  Note this must be allocated in a
282   * server-lifetime manner, typically as a static const struct, which then gets
283   * returned by FormData_pg_am.amhandler.
284   *
285   * In most cases it's not appropriate to call the callbacks directly, use the
286   * table_* wrapper functions instead.
287   *
288   * GetTableAmRoutine() asserts that required callbacks are filled in, remember
289   * to update when adding a callback.
290   */
      Alexander Korotkov, 11 months ago | 10 authors (Andres Freund and others)
291  typedef struct TableAmRoutine
292  {
293      /* this must be set to T_TableAmRoutine */
294      NodeTag     type;
295
```

Postgres: src/include/access/tableam.h; TableAmRoutine

PostgresPro

Чтение



`void (*scan_rescan)` пока не трогаем

Example. Citus Columnar: src/backend/columnar/columnar_tableam.c; columnar_beginscan; columnar_getnextslot; columnar_endscan

Postgres. ResourceReleaseCallback: src/include/utils/resowner.h

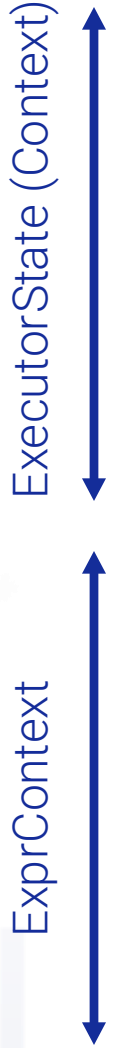
Запись

Single INSERT:

```
/* see table_tuple_insert() for reference about parameters */
void        (*tuple_insert) (Relation rel, TupleTableSlot *slot,
                             CommandId cid, int options,
                             struct BulkInsertStateData *bistate);
```

COPY:

```
/* see table_multi_insert() for reference about parameters */
void        (*multi_insert) (Relation rel, TupleTableSlot **slots, int nslots,
                             CommandId cid, int options, struct BulkInsertStateData *bistate);
```



Example. Citus Columnar: src/backend/columnar/columnar_tableam.c; columnar_tuple_insert; columnar_multi_insert

Хранение

```
/*
 * This callback needs to create new relation storage for `rel`, with
 * appropriate durability behaviour for `persistence`.
 *
 * Note that only the subset of the relcache filled by
 * RelationBuildLocalRelation() can be relied upon and that the relation's
 * catalog entries will either not yet exist (new relation), or will still
 * reference the old relfilelocator.
 *
 * As output *freezeXid, *minmulti must be set to the values appropriate
 * for pg_class.{relfrozenxid, relminmxid}. For AMs that don't need those
 * fields to be filled they can be set to InvalidTransactionId and
 * InvalidMultiXactId, respectively.
 *
 * See also table_relation_set_new_filelocator().
 */
void        (*relation_set_new_filelocator) (Relation rel,
                                             const RelFileLocator *newrlocator,
                                             char persistence,
                                             TransactionId *freezeXid,
                                             MultiXactId *minmulti);
```

Оценка стоимости

```
/*
 * See table_relation_estimate_size().
 *
 * While block oriented, it shouldn't be too hard for an AM that doesn't
 * internally use blocks to convert into a usable representation.
 *
 * This differs from the relation_size callback by returning size
 * estimates (both relation size and tuple count) for planning purposes,
 * rather than returning a currently correct estimate.
 */
void        (*relation_estimate_size) (Relation rel, int32 *attr_widths,
                                       BlockNumber *pages, double *tuples,
                                       double *allvisfrac);
```

Example. Citus Columnar: src/backend/columnar/columnar_tableam.c; columnar_estimate_rel_size

Example. csv_tam. https://github.com/InnerLife0/csv_tam/blob/main/src/impl.c; csv_relation_set_new_filelocator

PostgresPro

Слоты. TOAST.

```c
/*
 * Return slot implementation suitable for storing a tuple of this AM.
 */
const TupleTableSlotOps *(*slot_callbacks) (Relation rel);
```

```c
csv_slot_callbacks(Relation relation)
{
    /* use virtual slot implementation as is */
    return &TTSOpsVirtual;
}
```

```c
/*
 * This callback should return true if the relation requires a TOAST table
 * and false if it does not.  It may wish to examine the relation's tuple
 * descriptor before making a decision, but if it uses some other method
 * of storing large values (or if it does not support them) it can simply
 * return false.
 */
bool        (*relation_needs_toast_table) (Relation rel);
```

```c
bool
csv_relation_needs_toast_table(Relation relation)
{
    return false;
}
```

Example. Citus Columnar: src/backend/columnar/columnar_tableam.c; columnar_slot_callbacks; columnar_relation_needs_toast_table

Example. csv_tam. https://github.com/InnerLife0/csv_tam/blob/main/src/impl.c; csv_slot_callbacks; csv_relation_needs_toast_table

Первый запуск

Пример Table Access Method на базе CSV

```
static const TableAmRoutine routine = {
    .type = T_TableAmRoutine,

    .slot_callbacks = csv_slot_callbacks,

    .scan_begin = csv_beginscan,
    .scan_end = csv_endscan,
    .scan_getnextslot = csv_getnextslot,

    .tuple_insert = csv_tuple_insert,
    .multi_insert = csv_multi_insert,

    .relation_set_new_filelocator = csv_relation_set_new_filelocator,
    .relation_needs_toast_table = csv_relation_needs_toast_table,
    .relation_estimate_size = csv_estimate_rel_size,
```

https://github.com/InnerLife0/csv_tam/

Индексы и доступ по TID

```
struct IndexFetchTableData *(*index_fetch_begin) (Relation rel);

/*
 * Reset index fetch. Typically this will release cross index fetch
 * resources held in IndexFetchTableData.
 */
void        (*index_fetch_reset) (struct IndexFetchTableData *data);

/*
 * Release resources and deallocate index fetch.
 */
void        (*index_fetch_end) (struct IndexFetchTableData *data);
```

```
bool        (*index_fetch_tuple) (struct IndexFetchTableData *scan,
                                  ItemPointer tid,
                                  Snapshot snapshot,
                                  TupleTableSlot *slot,
                                  bool *call_again, bool *all_dead);
```

TID = 6 байт (-ограничения)

```
typedef struct ItemPointerData
{
    BlockIdData ip_blkid;
    OffsetNumber ip_posid;
}
```

2x uint16

1x uint16

Citus Columnar. About TID limitations: src/include/columnar/columnar_tableam.h

Parallel Workers

```
/*
 * Estimate the size of shared memory needed for a parallel scan of this
 * relation. The snapshot does not need to be accounted for.
 */
Size        (*parallelscan_estimate) (Relation rel);

/*
 * Initialize ParallelTableScanDesc for a parallel scan of this relation.
 * `pscan` will be sized according to parallelscan_estimate() for the same
 * relation.
 */
Size        (*parallelscan_initialize) (Relation rel,
                                        ParallelTableScanDesc pscan);

/*
 * Reinitialize `pscan` for a new scan. `rel` will be the same relation as
 * when `pscan` was initialized by parallelscan_initialize.
 */
void        (*parallelscan_reinitialize) (Relation rel,
                                          ParallelTableScanDesc pscan);
```

Example. Postgres: src/backend/access/table/tableam.c; table_block_parallelscan_initialize

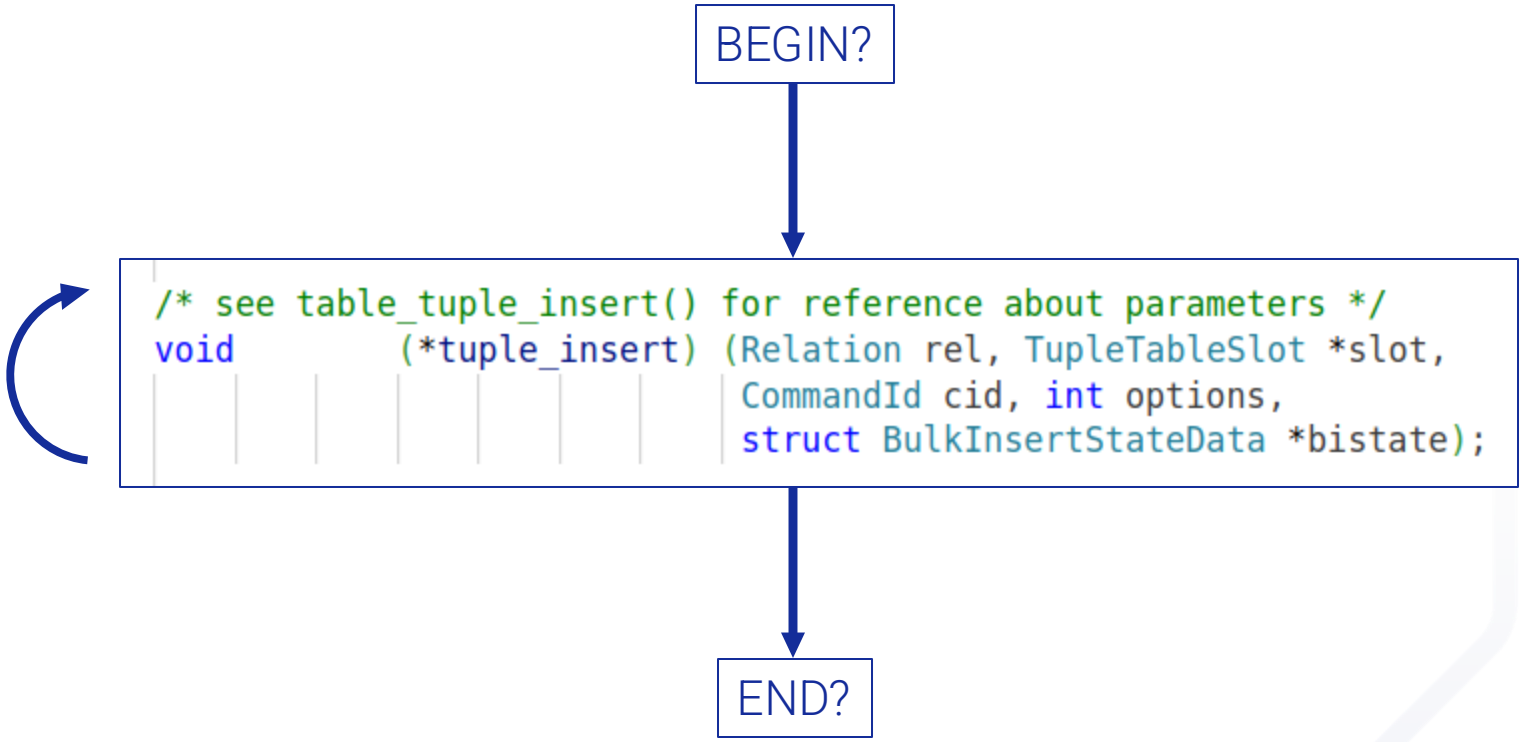
PostgresPro

Ограничения для колоночного движка

- Свой state для каждого INSERT
- Нет проброса проекции
- Нет проброса предикатов
- Исполнение запроса не векторизовано
- Исполнение применяет раннюю материализацию

Ограничения. Stateless Write.

- Нет методов begin и end



PostgresPro

Ограничения. Stateless Write.

Решение:

1. Хранить state в глобальной переменной
2. Освободить state при завершении транзакции

Плюсы:

1. Избегаем повторной инициализации
2. Буферизируем строки внутри транзакции на каждый INSERT. Записываем данные пакетно.

```c
/*
 * Mapping from relfilenode to WriteStateMapEntry. This keeps write state for
 * each relation.
 */
static HTAB *WriteStateMap = NULL;

/* memory context for allocating WriteStateMap & all write states */
static MemoryContext WriteStateContext = NULL;
```

```c
void
RegisterSubXactCallback(SubXactCallback callback, void *arg)
```

```c
void
RegisterXactCallback(XactCallback callback, void *arg)
```

Custom Context under TopTransactionContext

Example. Citus Columnar: src/backend/columnar/write_state_management.c

Postgres: src/backend/access/transam/xact.c

PostgresPro

Ограничения. Нет проброса проекции.

- Список используемых колонок не передаётся в scan

```
329     TableScanDesc (*scan_begin) (Relation rel,
330                                  Snapshot snapshot,
331                                  int nkeys, struct ScanKeyData *key,
332                                  ParallelTableScanDesc pscan,
333                                  uint32 flags);
```


“Bitmapset *attr_needed” ?

Ограничения. Нет проброса проекции.

Решение:
Использовать CustomScan

Методы CustomScan и CustomExec открывают доступ к большему числу структур ядра, в том числе к ScanState.

```
explain select i from c_test where i = 1;
                        QUERY PLAN
-------------------------------------------------------------------------
 Custom Scan (ColumnarScan) on c_test  (cost=0.00..0.00 rows=1 width=4)
   Filter: (i = 1)
   Columnar Projected Columns: i
   Columnar Chunk Group Filters: (i = 1)
(4 rows)
```

```
typedef struct CustomScan
{
    Scan         scan;
    uint32       flags;
    List        *custom_plans;
    List        *custom_exprs;
    List        *custom_private;
    List        *custom_scan_tlist;
    Bitmapset   *custom_relids;
    const struct CustomScanMethods *methods;
} CustomScan;
```

Example. Citus Columnar: src/backend/columnar/columnar_customscan.c; ColumnarAttrNeeded
Postgres: src/include/nodes/extensible.h

Ограничения. Нет проброса предикатов.

- Проброс условий фильтрации только внутри некоторых методов ядра и только для условий AND
- Для SeqScan отсутствует полностью

```
scandesc = table_beginscan(node->ss.ss_currentRelation,
                           estate->es_snapshot,
                           0, NULL);
node->ss.ss_currentScanDesc = scandesc;      Robert Haas
```

```
TableScanDesc (*scan_begin) (Relation rel,
                             Snapshot snapshot,
                             int nkeys, struct ScanKeyData *key,
                             ParallelTableScanDesc pscan,
                             uint32 flags);
```

Postgres: src/backend/executor/nodeSeqscan.c

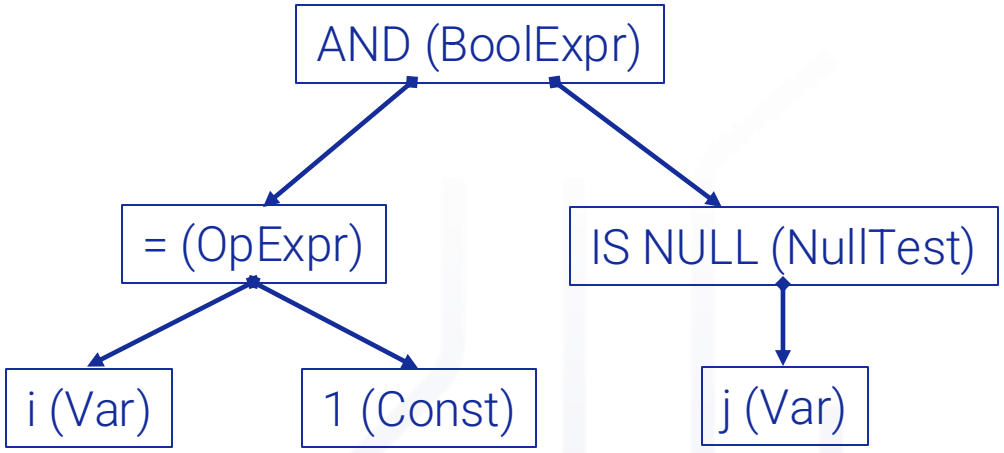
PostgresPro

Ограничения. Нет проброса предикатов.

Решение:
Использовать CustomScan

Плюс/минус:
Ядро повторно проверит условия для строки из движка

```
explain select i from c_test where i = 1 and j is null;
                          QUERY PLAN
---------------------------------------------------------------------
 Custom Scan (ColumnarScan) on c_test  (cost=0.00..0.00 rows=1 width=4)
   Filter: ((j IS NULL) AND (i = 1))
   Columnar Projected Columns: i, j
   Columnar Chunk Group Filters: (i = 1)
(4 rows)
```



Example. Citus Columnar: src/backend/columnar/columnar_customscan.c; FilterPushdownClauses

PostgresPro

Ограничения. Нет векторизации.

Per tuple execution:

Vectorized execution:

Ограничения. Нет векторизации.

Решение:

1) ~~.Патчи.~~ ПАТЧИ в ядро. 
2) Сторонний executor (самописный или готовый)

Ограничения. Ранняя материализация.

```
explain (analyze, verbose, costs off) select t0.j
from (
   select i, j
   from test0
   where exists (select 1 from test1 where test0.i=test1.k)
) t0
offset 1000000 limit 10;
```

```
 Limit (actual time=11660.533..11660.540 rows=10 loops=1)
   Output: test0.j
   ->  Hash Semi Join (actual time=516.460..11621.005 rows=1000010 loops=1)
         Output: test0.j
         Hash Cond: (test0.i = test1.k)
         ->  Seq Scan on public.test0 (actual time=0.016..4427.402 rows=30100000 loops=1)
               Output: test0.j, test0.i
         ->  Hash (actual time=516.134..516.134 rows=2020000 loops=1)
               Output: test1.k
               Buckets: 262144  Batches: 16  Memory Usage: 6492kB
               ->  Seq Scan on public.test1 (actual time=0.010..198.298 rows=2020000 loops=1)
                     Output: test1.k
 Planning Time: 0.162 ms
 Execution Time: 11755.489 ms
(14 rows)
```

А используем только 10

Достали миллионы значений из столбца j

PostgresPro

Ограничения. Ранняя материализация.

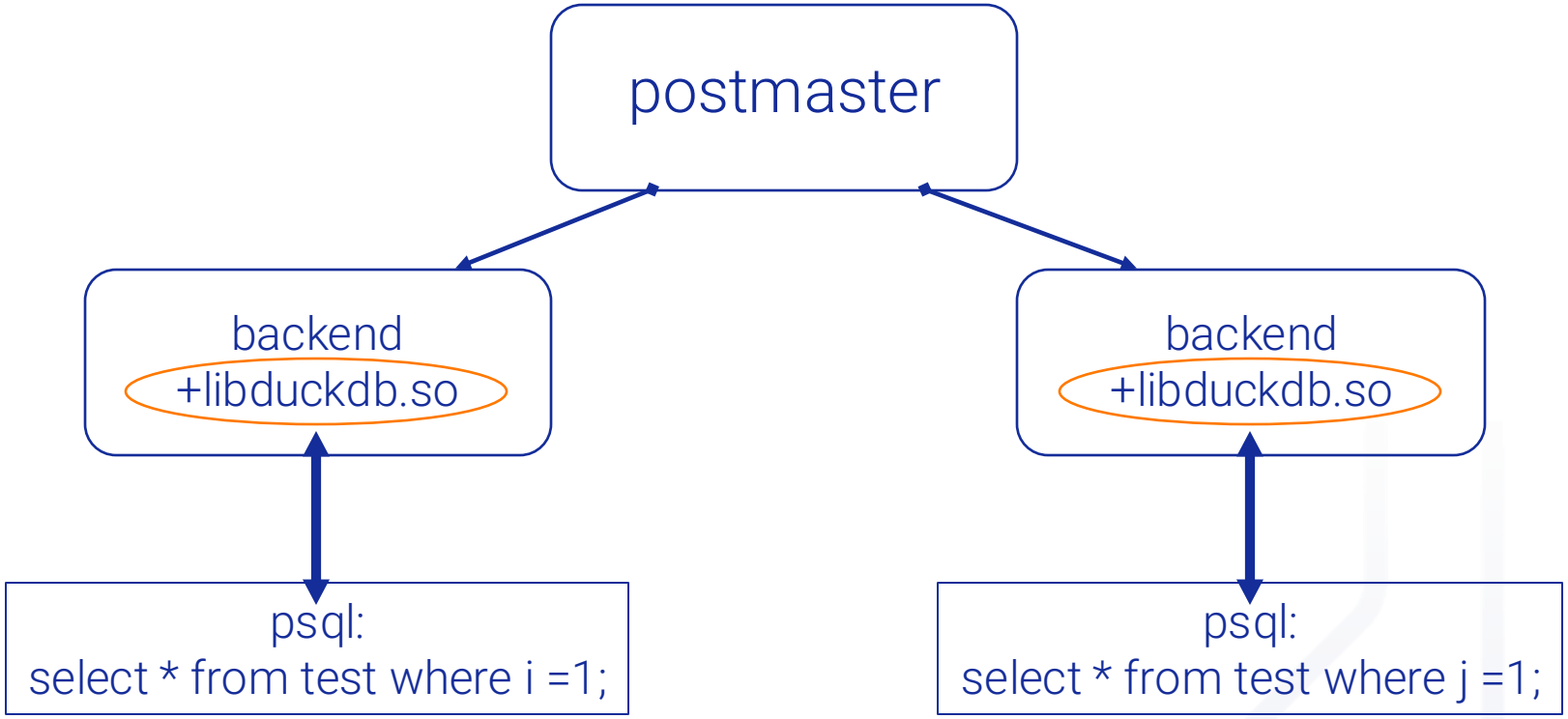
Решение:

1) Еще ПАТЧИ в ядро.
2) Сторонний executor (самописный или готовый)

Сторонний исполнитель. Duck Tales.

DuckDB is an analytical in-process SQL database management system.

pg_duckdb is a Postgres extension that embeds DuckDB's columnar-vectorized analytics engine and features into Postgres.

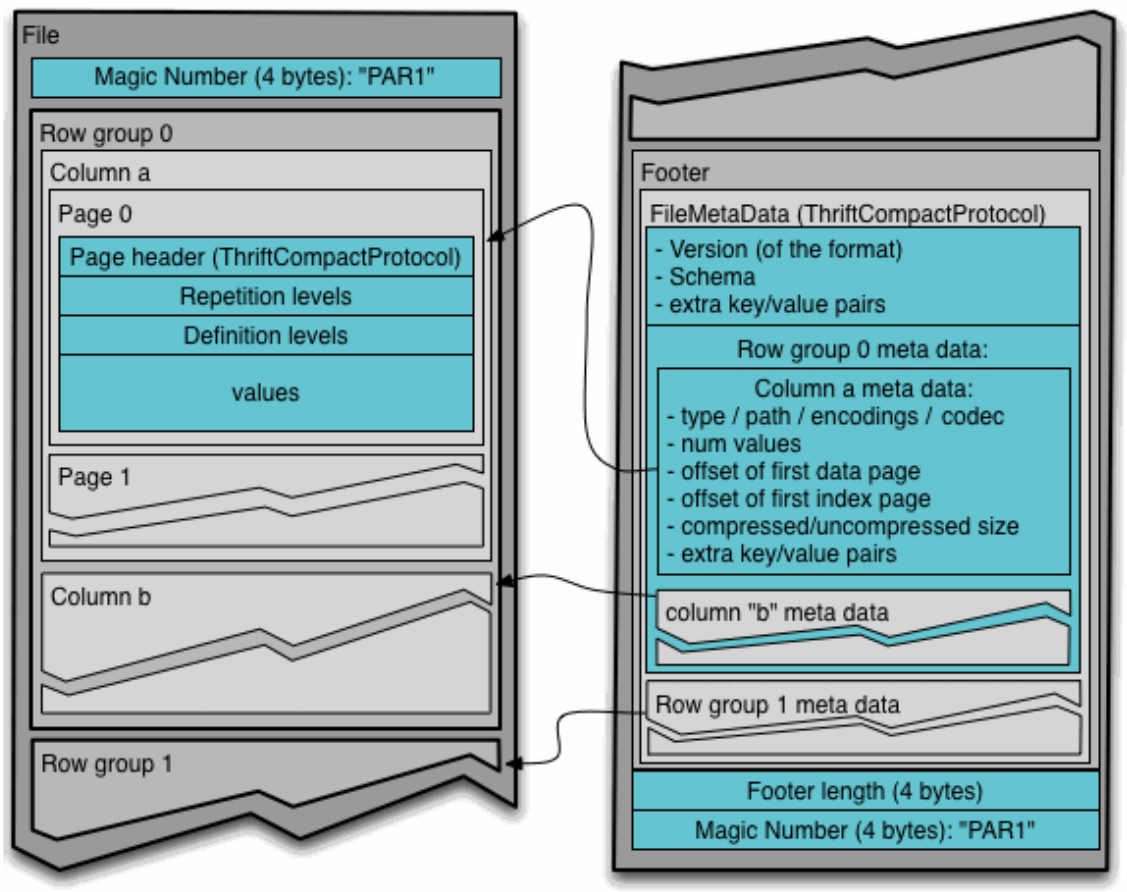


https://github.com/duckdb/duckdb

https://github.com/duckdb/pg_duckdb

Parquet

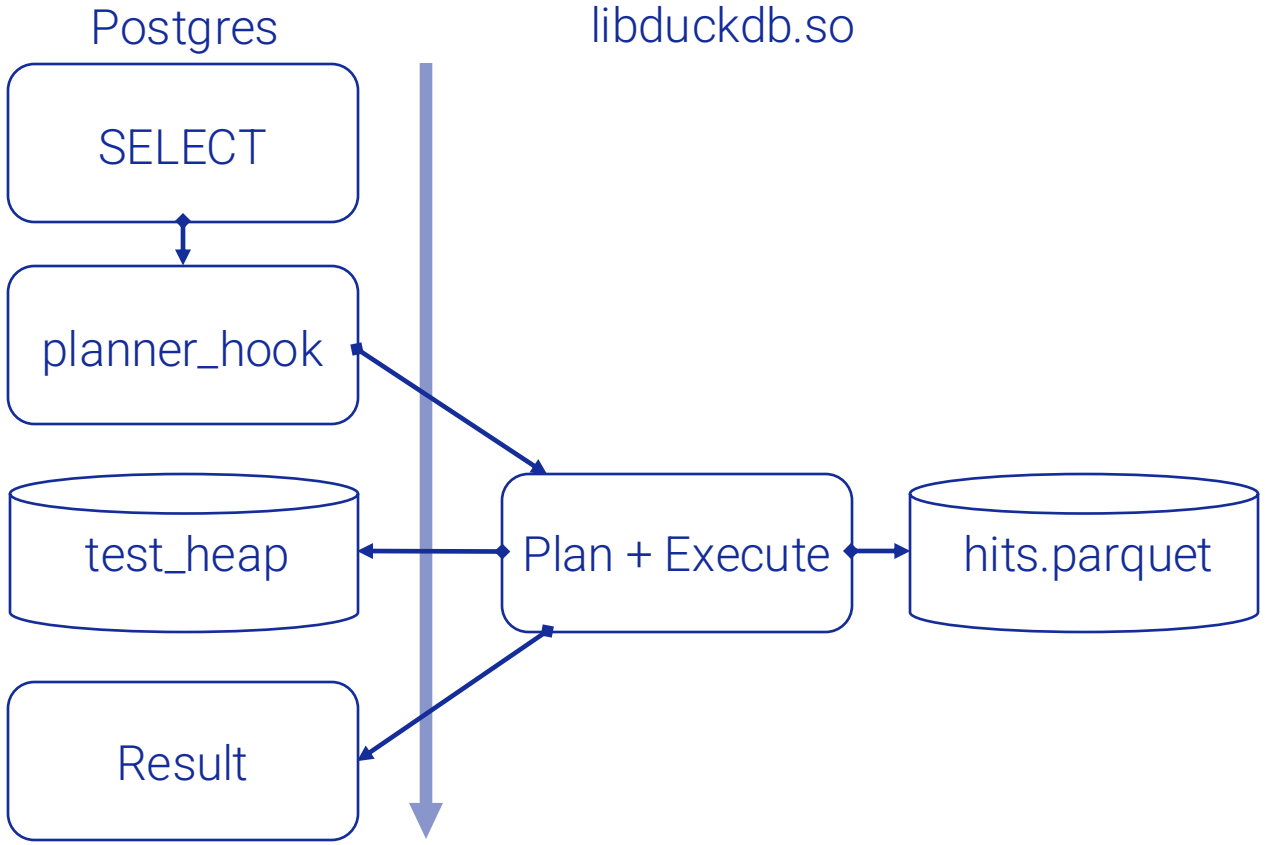
Формат основанный на идеях PAX



- Compression (SNAPPY, GZIP, LZO, BROTLI, ZSTD, LZ4)
- Encoding (PLAIN/RLE_DICTIONARY, RLE, DELTA*, BYTE_STREAM_SPLIT)
- Checksumming
- MIN/MAX, bloom and others

DuckDB + Parquet + Postgres

```
explain select r['watchid']
from read_parquet('/tmp/hits.parquet') r
where r['watchid'] not in (select i from test_heap);
```




```
                          QUERY PLAN
-------------------------------------------------------------
Custom Scan (DuckDBScan)  (cost=0.00..0.00 rows=0 width=0)
  DuckDB Execution Plan:

┌─────────────────────────────┐
│         PROJECTION          │
│    ─────────────────────    │
│             #0              │
│                             │
│       ~19999499 Rows        │
└──────────────┬──────────────┘
┌──────────────┴──────────────┐
│           FILTER            │
│    ─────────────────────    │
│       (NOT SUBQUERY)        │
│                             │
│       ~19999499 Rows        │
└──────────────┬──────────────┘
┌──────────────┴──────────────┐                ┌─────────────────────────────┐
│         HASH_JOIN           │                │                             │
│    ─────────────────────    │                │                             │
│     Join Type: MARK         │                │                             │
│                             │                │                             │
│         Conditions:         │                │                             │
│ WatchID = CAST(#0 AS BIGINT)│                │                             │
│                             │                │                             │
│       ~99997497 Rows        │                │                             │
└──────────────┬──────────────┘                └──────────────┬──────────────┘
┌──────────────┴──────────────┐                ┌──────────────┴──────────────┐
│        READ_PARQUET         │                │        POSTGRES_SCAN        │
```

PostgresPro

Полезное

- Soft DELETE
- Буферизация строк
- WAL. Resource Manager.
- Storage Manager
- Использование аллокаторов Postgres
- в сторонней либе

Soft DELETE

Parquet

Row Group #1

10	Ivan	14.08.1987	1
11	Peter	26.02.1984	2
12	Sidor	09.11.1995	3

Row Group #2

…	…	…	4
…	…	…	5
…	…	…	6

deleted_tid (heap, MVCC)

TID
1
5
…

```
select * from parquet;
```

Прозрачно для пользователя

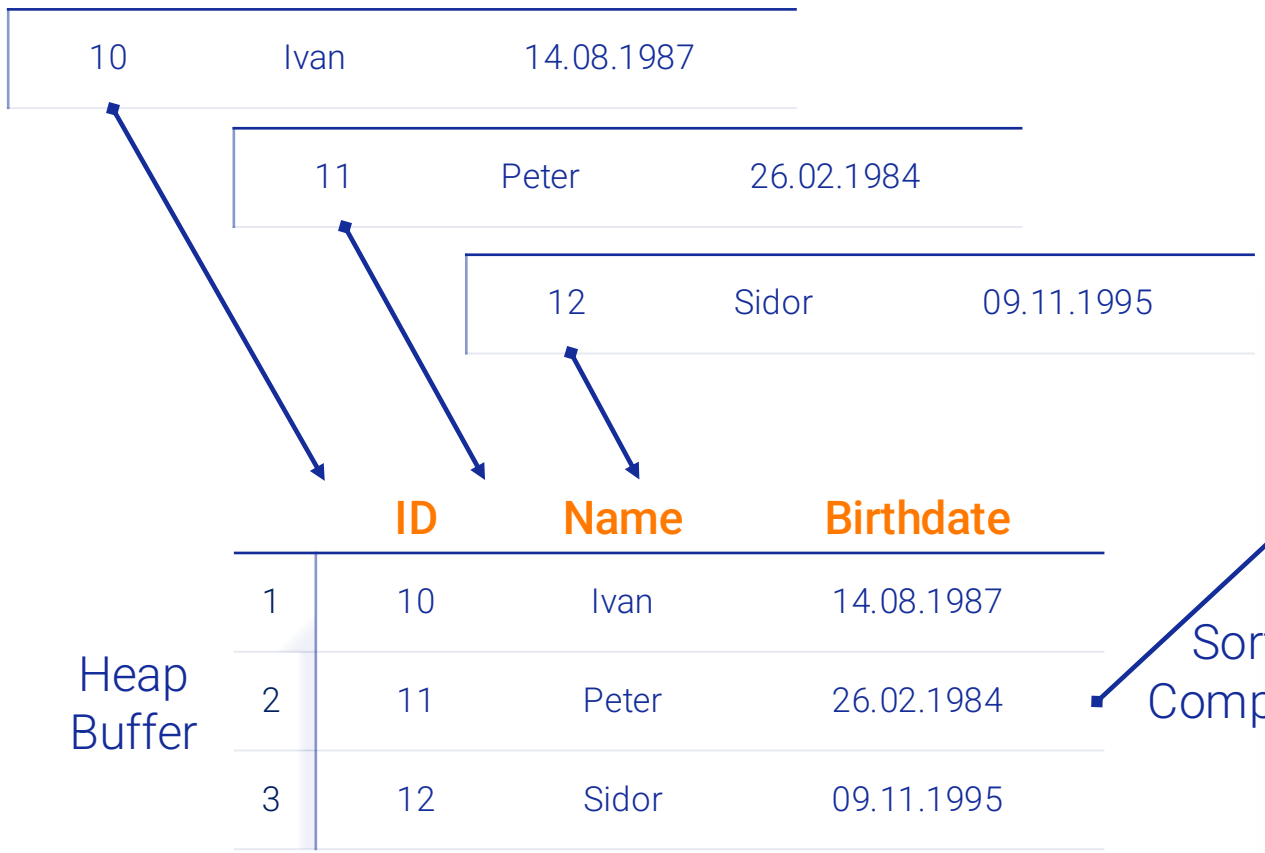
```
select *
from parquet
where tid not in
    (select tid from deleted_tid);
```

Example. Hydra: columnar/src/backend/columnar/columnar_metadata.c; UpdateRowMask; “row_mask” table

Example. Greenplum: src/backend/access/appendonly/appendonly_visimap.c; AppendOnlyVisimapDelete_Hide; “pg_aovisimap_*” tables

PostgresPro

Буферизация строк



Example. TimescaleDB: tsl/src/compression/api.c; tsl_compress_chunk_wrapper

Storage Manager

smgr.c

```
typedef struct f_smgr
{
    void        (*smgr_init) (void);    /* may be NULL */
    void        (*smgr_shutdown) (void);    /* may be NULL */
    void        (*smgr_open) (SMgrRelation reln);
    void        (*smgr_close) (SMgrRelation reln, ForkNumber forknum);
    void        (*smgr_create) (SMgrRelation reln, ForkNumber forknum,
                                bool isRedo);
    bool        (*smgr_exists) (SMgrRelation reln, ForkNumber forknum);
    void        (*smgr_unlink) (RelFileLocatorBackend rlocator, ForkNumber forknum,
                                bool isRedo);
    void        (*smgr_extend) (SMgrRelation reln, ForkNumber forknum,
                                BlockNumber blocknum, const void *buffer, bool skipFsync);
    void        (*smgr_zeroextend) (SMgrRelation reln, ForkNumber forknum,
                                    BlockNumber blocknum, int nblocks, bool skipFsync);
    bool        (*smgr_prefetch) (SMgrRelation reln, ForkNumber forknum,
                                  BlockNumber blocknum, int nblocks);
    void        (*smgr_readv) (SMgrRelation reln, ForkNumber forknum,
                               BlockNumber blocknum,
                               void **buffers, BlockNumber nblocks);
    void        (*smgr_writev) (SMgrRelation reln, ForkNumber forknum,
                                BlockNumber blocknum,
                                const void **buffers, BlockNumber nblocks,
                                bool skipFsync);
```

Storage Manager. Зачем Custom?

```
void
mdwritev(SMgrRelation reln, ForkNumber forknum, BlockNumber blocknum,
         const void **buffers, BlockNumber nblocks, bool skipFsync)
{
    /* This assert is too expensive to have on normally ... */
#ifdef CHECK_WRITE_VS_EXTEND
    Assert((uint64) blocknum + (uint64) nblocks <= (uint64) mdnblocks(reln, forknum));
#endif

    while (nblocks > 0)
    {
        struct iovec iov[PG_IOV_MAX];
        int          iovcnt;
        off_t        seekpos;
        int          nbytes;
        MdfdVec     *v;
        BlockNumber nblocks_this_segment;
        size_t      transferred_this_segment;
        size_t      size_this_segment;

        v = _mdfd_getseg(reln, forknum, blocknum, skipFsync,
                         EXTENSION_FAIL | EXTENSION_CREATE_RECOVERY);

        seekpos = (off_t) BLCKSZ * (blocknum % ((BlockNumber) RELSEG_SIZE));
```

```
...;...;...
10;Ivan;1987-08-14
11;Peter\x00\x05\x00\x04;1984-02-26
12;Sidor;1995-11-09
```

- Хидеры каждые 8кб
- Выравнивание по 8кб – пустые данные в конце

Postgres. Useful functions for custom SMGR: src/include/storage/fd.h

PostgresPro

WAL. Generic XLog vs Custom RMGR.

Generic XLog:

- Если формат укладывается в блоки
- Меньше кода
- Операции через буферный пул

```
/* API for construction of generic xlog records */
extern GenericXLogState *GenericXLogStart(Relation relation);
extern Page GenericXLogRegisterBuffer(GenericXLogState *state, Buffer buffer,
                                      int flags);
extern XLogRecPtr GenericXLogFinish(GenericXLogState *state);
extern void GenericXLogAbort(GenericXLogState *state);
```

Custom Resource Manager:

- Если формат не укладывается в блоки
- Больше возможностей
- Больше кода

```
typedef struct RmgrData
{
    const char *rm_name;
    void        (*rm_redo) (XLogReaderState *record);
    void        (*rm_desc) (StringInfo buf, XLogReaderState *record);
    const char *(*rm_identify) (uint8 info);
    void        (*rm_startup) (void);
    void        (*rm_cleanup) (void);
    void        (*rm_mask) (char *pagedata, BlockNumber blkno);
    void        (*rm_decode) (struct LogicalDecodingContext *ctx,
                              struct XLogRecordBuffer *buf);
} RmgrData;
```

Postgres: src/include/access/generic_xlog.h

Postgres: src/include/access/xlog_internal.h

Example. Citus Columnar: src/backend/columnar/columnar_storage.c; WriteToBlock

Example. Postgres: src/test/modules/test_custom_rmgrs/test_custom_rmgrs.c

Аллокаторы

```
server closed the connection unexpectedly
        This probably means the server terminated abnormally
        before or while processing the request.
The connection to the server was lost. Attempting reset: Succeeded.
postgres=# select 1;
 ?column?
----------
        1
(1 row)
```

```
server closed the connection unexpectedly
        This probably means the server terminated abnormally
        before or while processing the request.
The connection to the server was lost. Attempting reset: Failed.
The connection to the server was lost. Attempting reset: Failed.
!?> select 1;
You are currently not connected to a database.
```

Аллокаторы Postgres в сторонней либе

Как подключить?

1. 
2. Использовать возможности либы (включая custom memory pool)
3. Переписать исходники
4. LD_PRELOAD
5. Global Offset Table patch

Аллокаторы. LD_PRELOAD

main.c

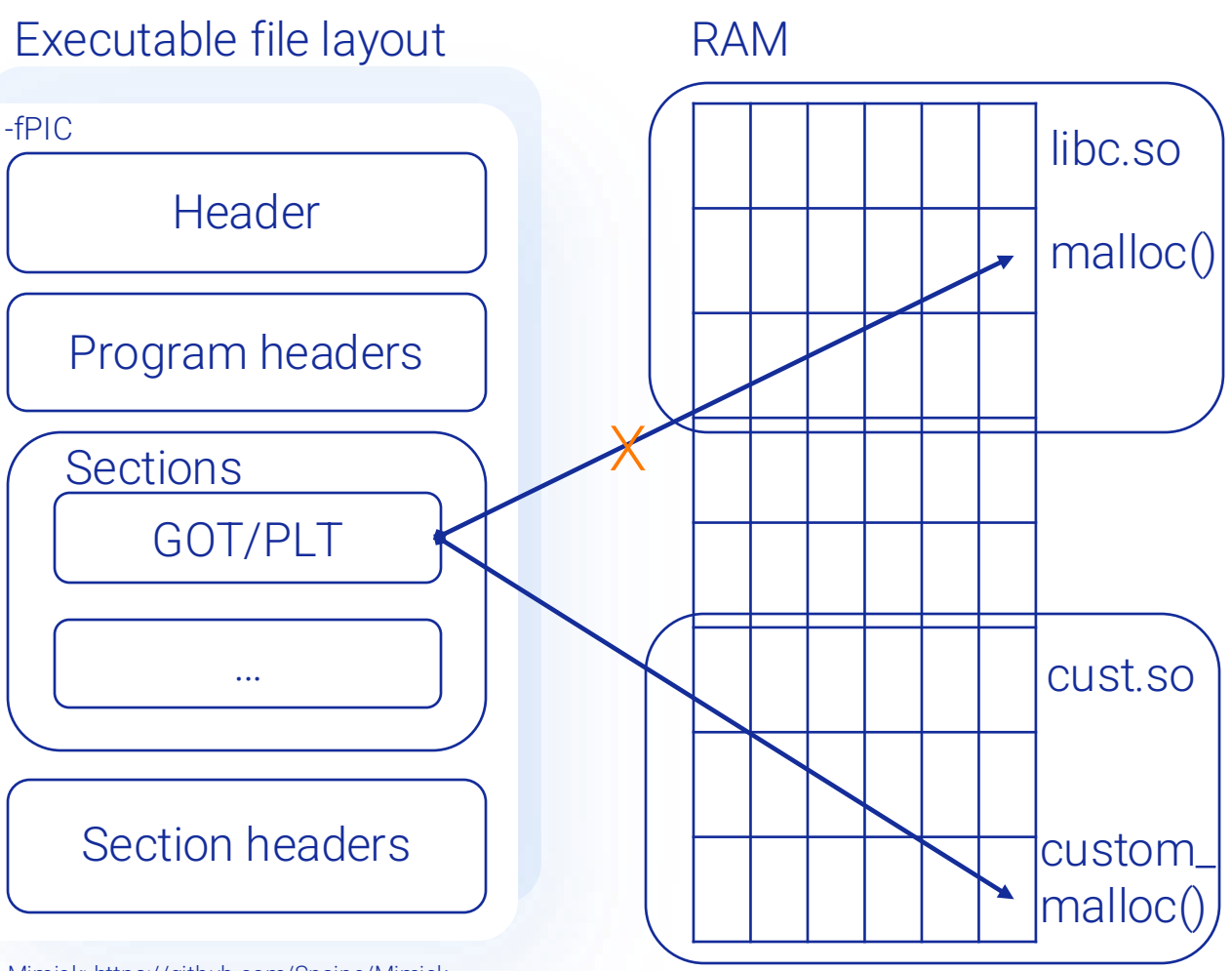
```c
#include <stdlib.h>
int main()
{
        void *p = malloc(10);
        free(p);
        return 0;
}
```

preload.c

```c
#include <dlfcn.h>
#include <unistd.h>
void *malloc(size_t size)
{
        void* (*malloc_ptr)(size_t) =
                dlsym(RTLD_NEXT, "malloc");
        write(1, "Hi from preload!\n", 17);
        return malloc_ptr(size);
}
```

```
> LD_PRELOAD=$PWD/preload.so ./main
Hi from preload!
```

Аллокаторы. Global Offset Table patch.



Mimick: https://github.com/Snaipe/Mimick

Manual Example: https://stackoverflow.com/questions/27137527/overload-symbols-of-running-process-ld-preload-attachment

Modified manual example was tested on Apache Arrow lib.

Mimick

Unix Build Status build passing coverity passed license MIT

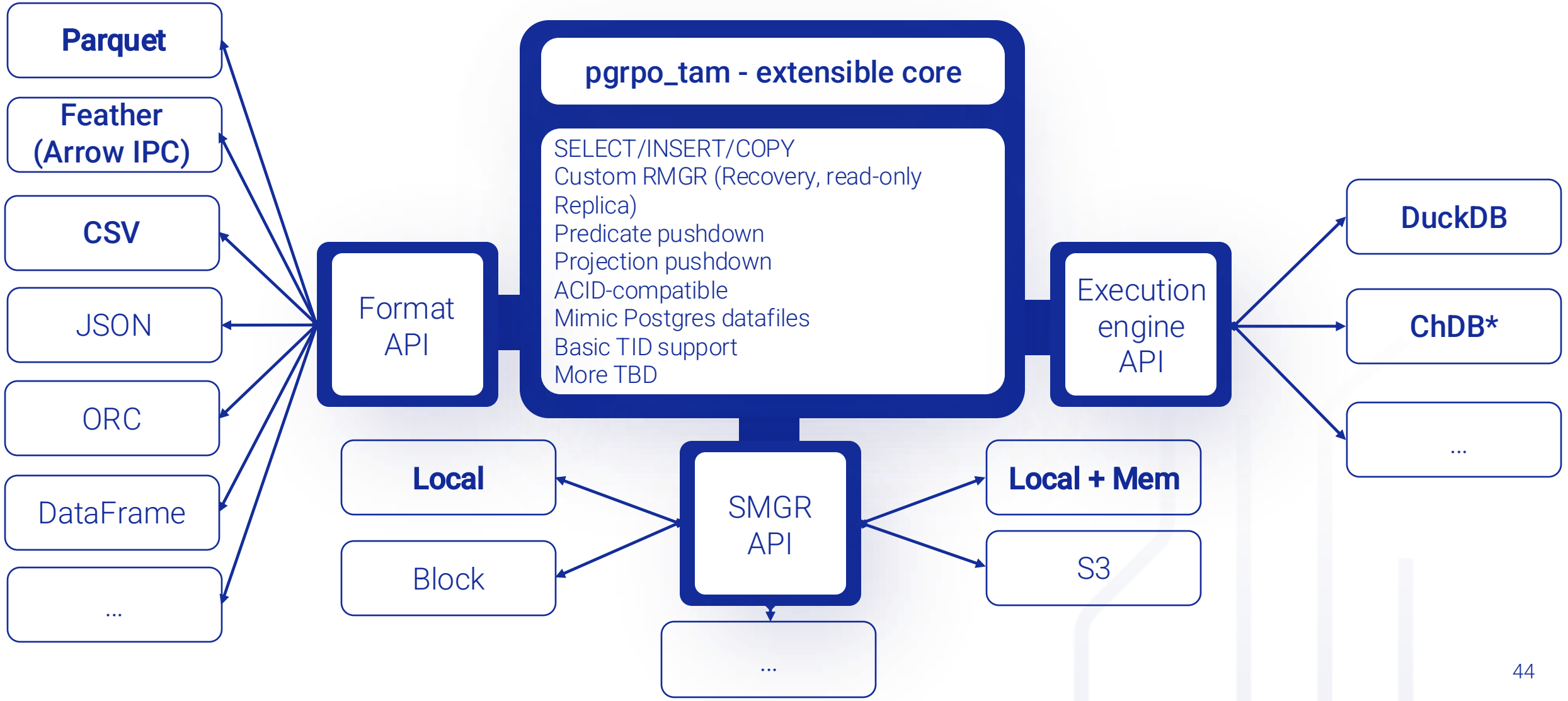
A KISS, cross-platform C Mocking library

✓: Supported.
×: Unsupported.
?: Not Tested, but is expected to work.
∄: Does not exist / not applicable.
~: Works on some conditions.

Arch	Linux	OS X/iOS	FreeBSD	Windows
x86	✓	✓	✓	✓
x86_64	✓	✓	✓	✓
ARM	✓	?	✓	×
ARM64	✓	?	?	∄
PPC	×	×	×	∄

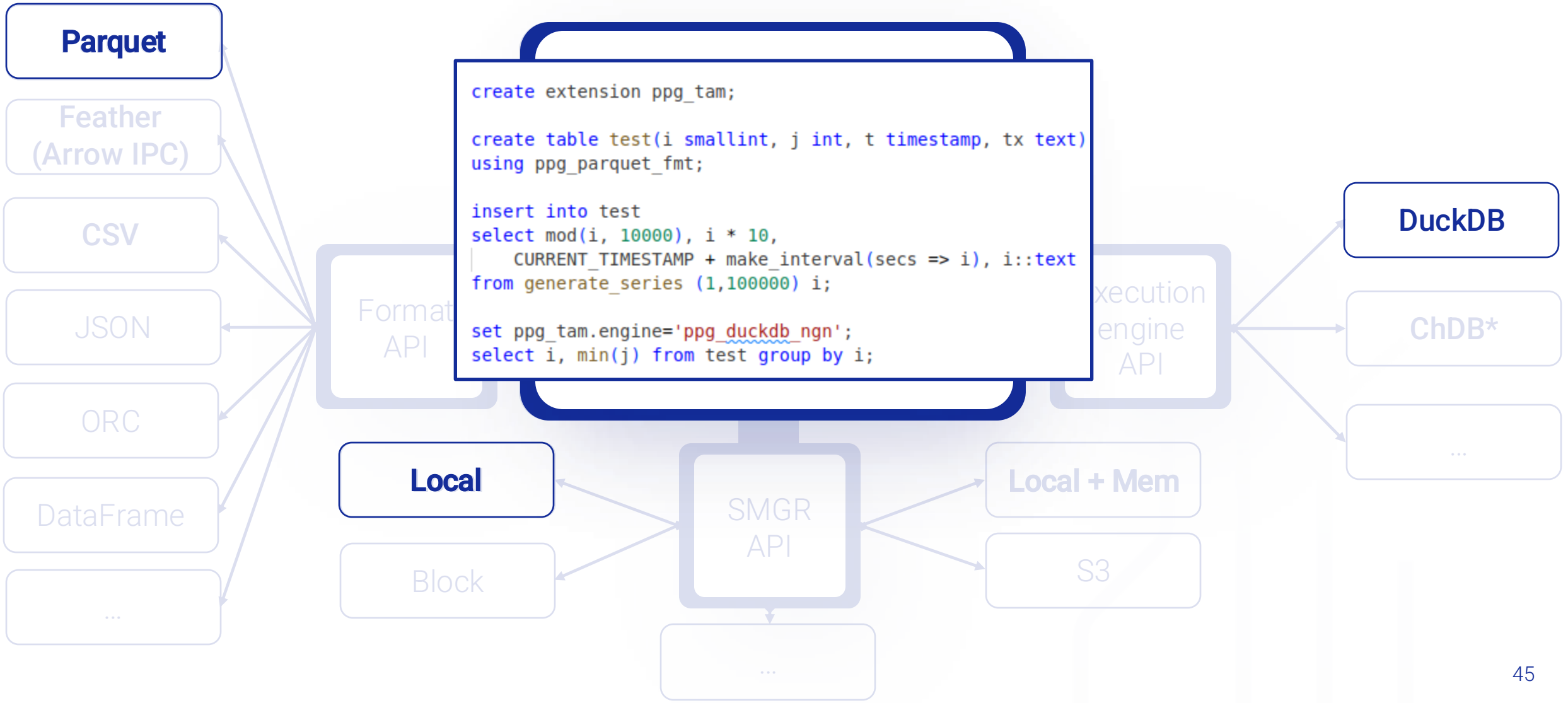
Часть 2.
Практическая.

pgpro_tam. Extension for vanilla Postgres.



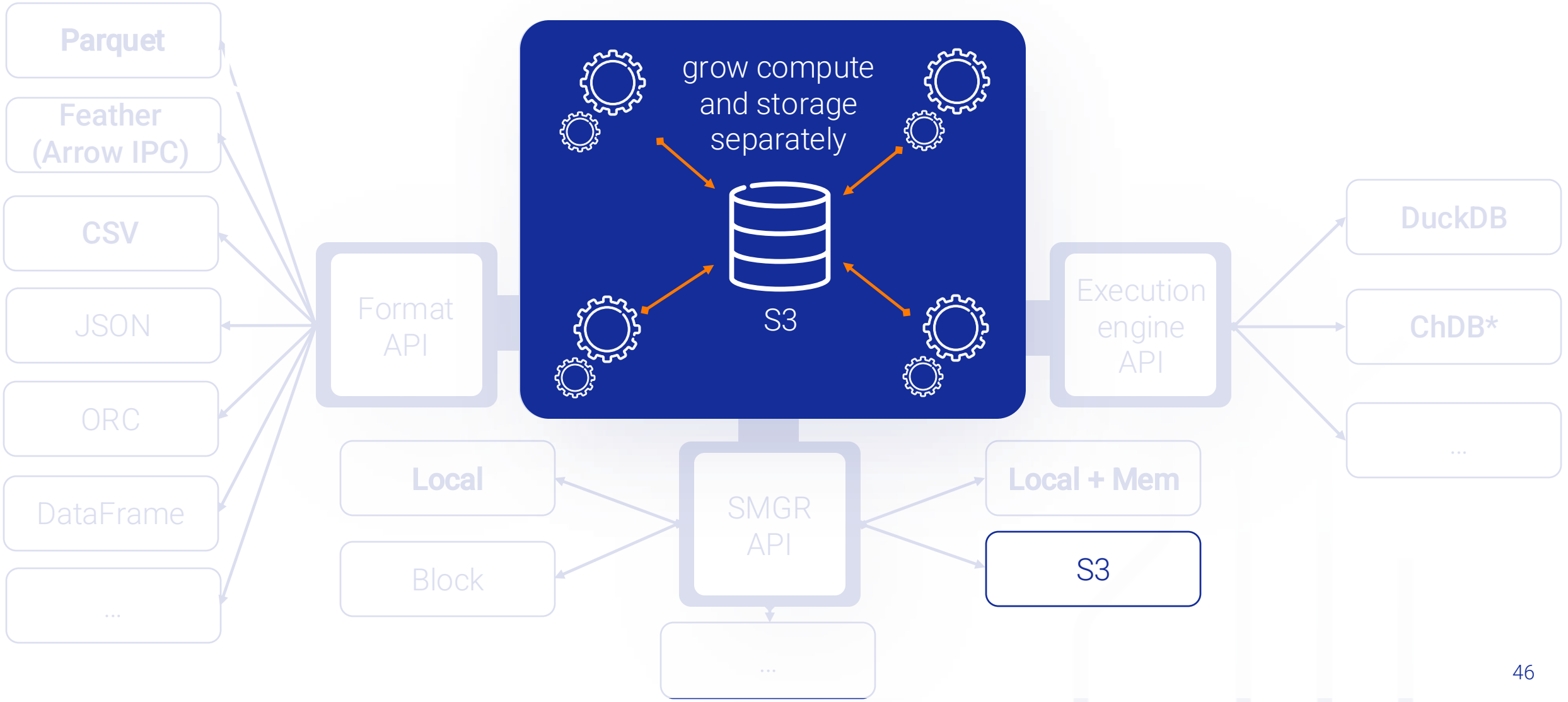
PostgresPro

pgpro_tam. Universal analytic engine.



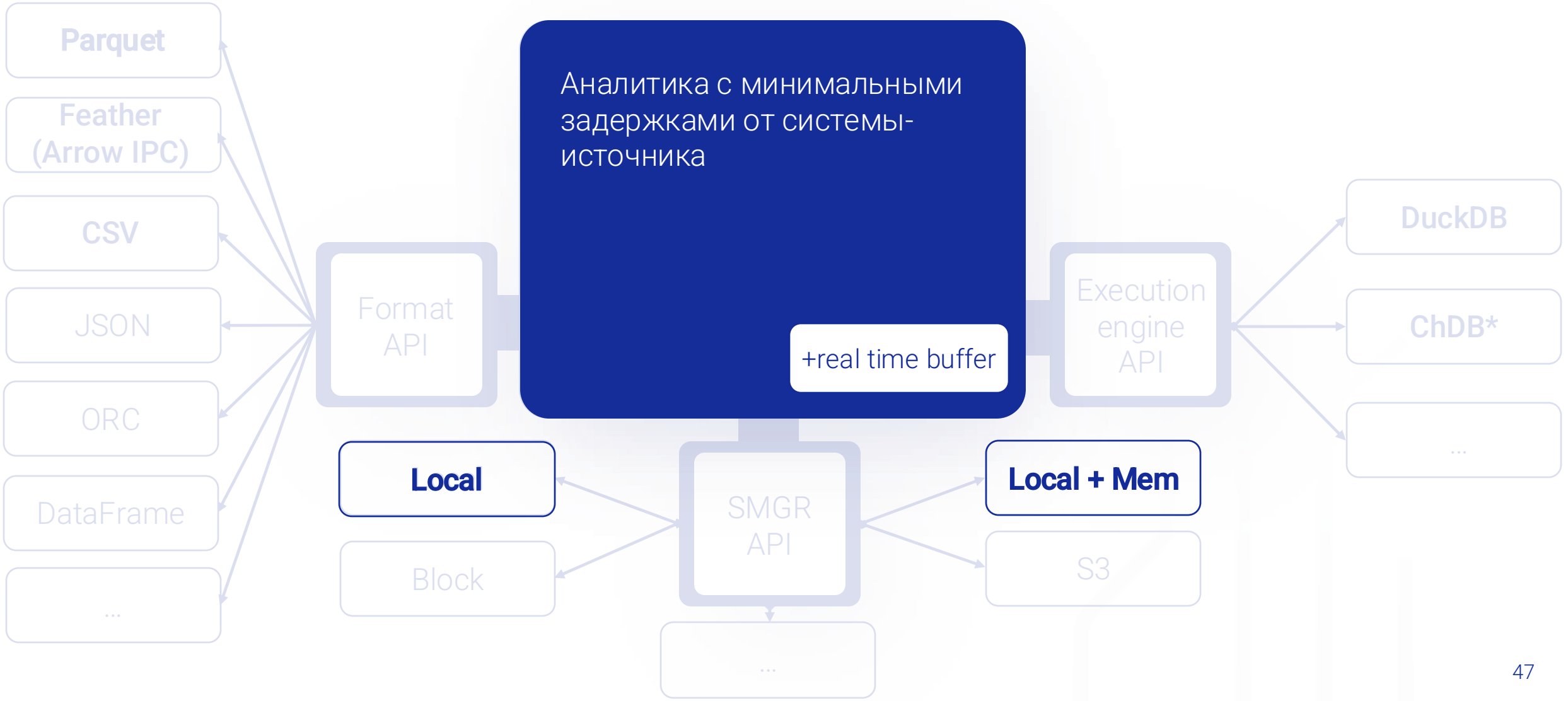
PostgresPro

pgpro_tam. Compute-storage.



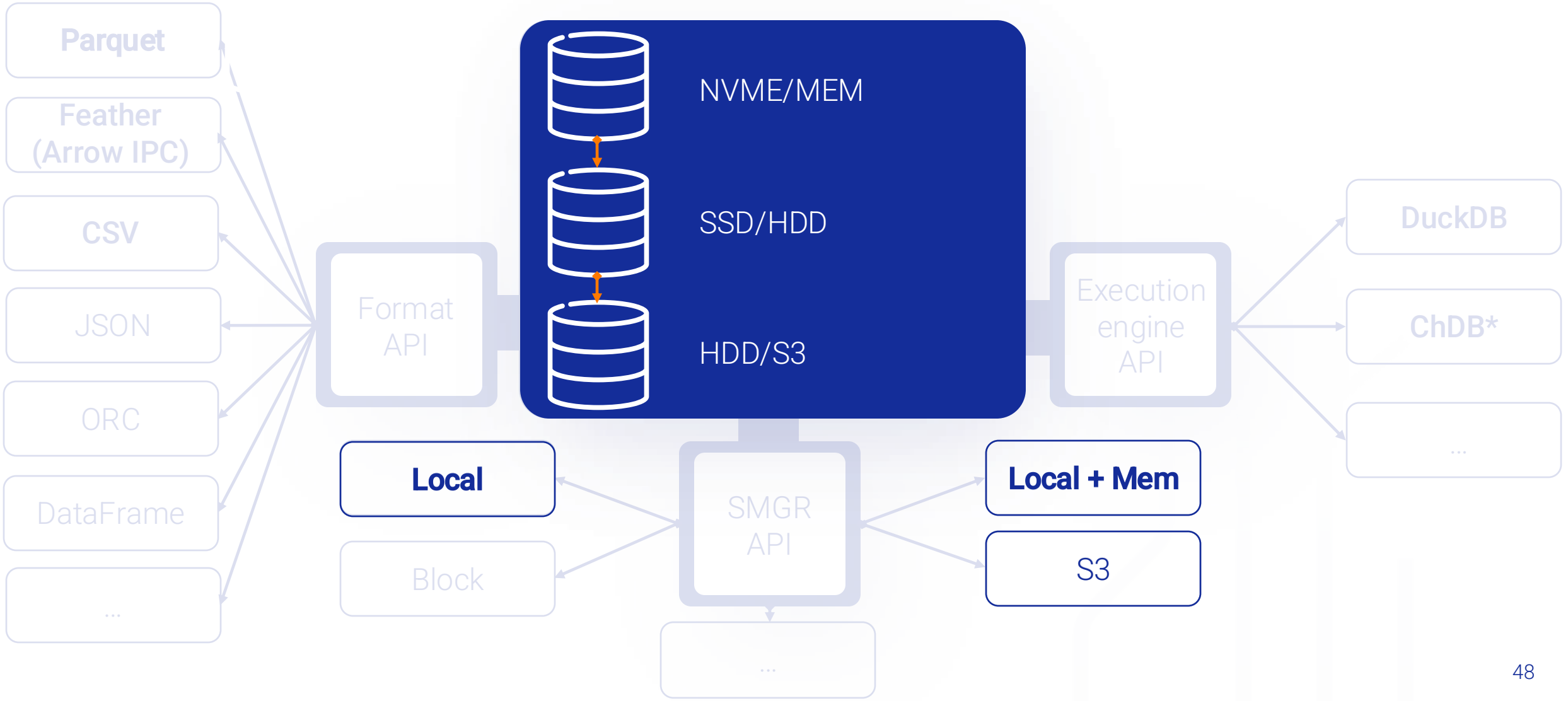
PostgresPro

pgpro_tam. Real-Time analytics.



PostgresPro

pgpro_tam. Data Tiering.



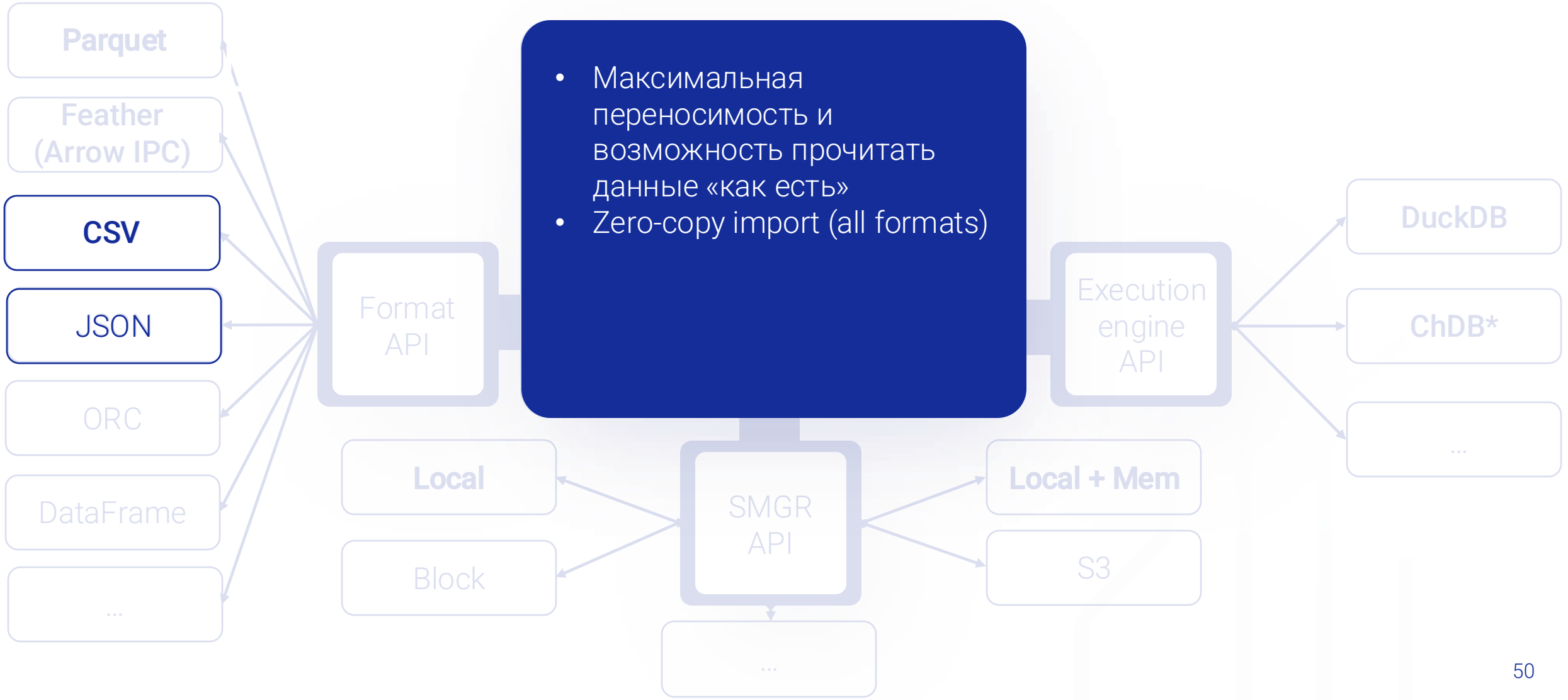
PostgresPro

pgpro_tam. Программно-аппаратный комплекс.

- Использование блочного устройства минуя ФС (128k block direct access)
- Проброс предикатов в контроллер диска

PostgresPro

pgpro_tam. Human-readable database.



- Максимальная переносимость и возможность прочитать данные «как есть»
- Zero-copy import (all formats)

PostgresPro

pgpro_tam. Machine learning.

PostgresPro

pgpro_tam. ClickBench. Load time.

Tinybird (Free Trial) (serverless):	stateless
ClickHouse (web) (c5n.4xlarge, 500gb gp2):	1s (×1.00)
ClickHouse (web) (c6a.metal, 500gb gp2):	1s (×1.00)
Polars (DataFrame) (c6a.metal, 500gb gp2):	2s (×2.30)
DuckDB (memory) (c6a.metal, 500gb gp2):	30s (×29.57)
MotherDuck (Jumbo):	69s (×69.39)
Databend (c6a.metal, 500gb gp2):	70s (×70.00)
MotherDuck (Standard):	77s (×76.80)
ClickHouse (tuned, memory) (c6a.metal, 500gb gp2):	85s (×84.73)
pg_duckdb (MotherDuck enabled) (Jumbo):	119s (×118.62)
DuckDB (c6a.metal, 500gb gp2):	124s (×123.60)
ClickHouse (c6a.metal, 500gb gp2):	128s (×128.43)
chDB (c6a.metal, 500gb gp2):	133s (×133.28)
Hydra (XL):	136s (×136.00)
ClickHouse (tuned) (c6a.metal, 500gb gp2):	136s (×136.44)
QuestDB (c6a.metal, 500gb gp2)†:	157s (×157.00)
ClickHouse Cloud (azure) (120GiB, 3 replica(s)):	173s (×173.24)
ClickHouse Cloud (aws) (120GiB, 3 replica(s)):	174s (×173.91)
• • •	
ClickHouse Cloud (aws) (120GiB, 2 replica(s)):	227s (×226.85)
ClickHouse Cloud (gcp) (32GiB, 3 replica(s)):	227s (×227.27)
pgpro_tam (parquet, local, parallel) (16 vCPU 32GB):	231s (×231.00)

Additional test:
~80Gb of heap
to pgpro_tam
30 vCPU
67% avg CPU usage
Result:
34.6 seconds

https://benchmark.clickhouse.com/

35 из 144 (без stateless)

pgpro_tam. ClickBench. Hot and Cold run.

ClickHouse (web) (c5n.4xlarge, 500gb gp2):	×6.03
Databend (c5.4xlarge, 500gb gp2):	×6.05
ClickHouse Cloud (gcp) (64GiB, 3 replica(s)):	×6.32
ClickHouse Cloud (gcp) (64GiB, 2 replica(s)):	×6.45
QuestDB (c6a.4xlarge, 500gb gp2):	×6.74
chDB (Parquet, partitioned) (c6a.metal, 500gb gp2):	×6.99
Apache Doris (c6a.4xlarge, 500gb gp2):	×7.03
StarRocks (c6a.4xlarge, 500gb gp2):	×7.08
Snowflake (64×3XL):	×7.15
Parseable (Parquet, partitioned) (c6a.metal, 500gb gp2)†:	×7.23
pgpro_tam (parquet, local + cache) (c6a.metal, 500gb gp2):	×7.25
pgpro_tam (parquet, local storage) (c6a.metal, 500gb gp2):	×7.32
Snowflake (32×2XL):	×7.43
Snowflake (128×4XL):	×7.90
Redshift (serverless):	×8.18
DuckDB (Parquet, partitioned) (c6a.4xlarge, 500gb gp2):	×8.28
ClickHouse Cloud (azure) (32GiB, 2 replica(s)):	×8.46
Tablespace (L1 - 16CPU 32GB):	×8.48
Snowflake (16×XL):	×8.71
pgpro_tam (parquet, local, parallel) (c6a.metal, 500gb gp2):	×8.83
Oxla (c6a.4xlarge, 500gb gp2)†:	×8.92
ClickHouse Cloud (aws) (32GiB, 3 replica(s)):	×8.98
pgpro_tam (feather, local + cache) (c6a.metal, 500gb gp2):	×9.43

57 из 360

Snowflake (64×3XL):	×7.32
Snowflake (32×2XL):	×7.40
ClickHouse (web) (c6a.metal, 500gb gp2):	×7.48
ClickHouse Cloud (gcp) (120GiB, 3 replica(s)):	×7.49
Hydra (XL):	×8.61
Snowflake (16×XL):	×8.63
Snowflake (128×4XL):	×8.67
Oxla (c6a.4xlarge, 500gb gp2)†:	×8.72
ByteHouse (8×L):	×8.87
ClickHouse Cloud (aws) (32GiB, 2 replica(s)):	×8.90
ClickHouse Cloud (aws) (32GiB, 3 replica(s)):	×8.98
pgpro_tam (parquet, local + cache) (16 vCPU 32GB):	×9.25
Ursa (c6a.4xlarge, 500gb gp2):	×9.39
pgpro_tam (parquet, local storage) (16 vCPU 32GB):	×9.85
Apache Doris (c6a.4xlarge, 500gb gp2):	×10.10
ClickHouse Cloud (gcp) (32GiB, 2 replica(s)):	×10.13
ClickHouse Cloud (azure) (32GiB, 2 replica(s)):	×10.22
for Analytics (Parquet) (Analytics-256GB (64 vCores, 256 GB)):	×10.33
ClickHouse Cloud (azure) (16GiB, 3 replica(s)):	×10.34
pgpro_tam (parquet, local + cache) (c6a.metal, 500gb gp2):	×10.44
ClickHouse Cloud (azure) (32GiB, 3 replica(s)):	×10.55
ClickHouse (c6a.metal, 500gb gp2):	×10.58
pgpro_tam (parquet, local, parallel) (16 vCPU 32GB):	×11.33
SingleStore (S24)†:	×11.35
Snowflake (8×L):	×11.59

35 из 360

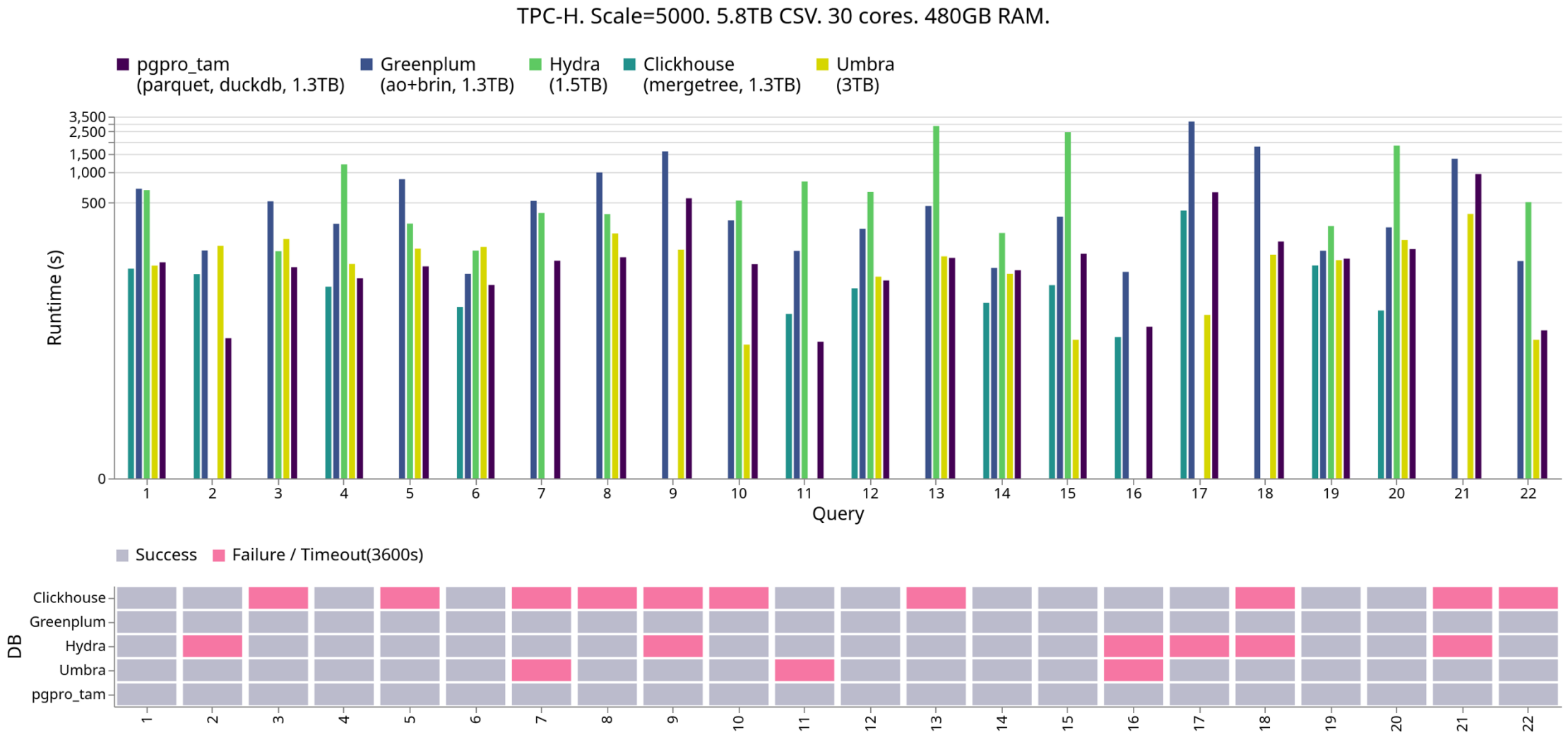
pgpro_tam. ClickBench. Storage Size.

System & Machine	Relative size (lower is better)
Tinybird (Free Trial) (serverless):	0.00 GiB (×0.00)
pgpro_tam (parquet, local, parallel) (16 vCPU 32GB):	8.66 GiB (×1.00)
pgpro_tam (parquet, local, parallel) (c6a.metal, 500gb gp2):	8.66 GiB (×1.00)
pgpro_tam (parquet, local + cache) (c6a.metal, 500gb gp2):	9.01 GiB (×1.04)
pgpro_tam (parquet, local storage) (16 vCPU 32GB):	9.03 GiB (×1.04)
pgpro_tam (parquet, local + cache) (16 vCPU 32GB):	9.03 GiB (×1.04)
pgpro_tam (parquet, local storage) (c6a.metal, 500gb gp2):	9.03 GiB (×1.04)
AlloyDB (16 vCPU 128GB):	9.26 GiB (×1.07)
AlloyDB (8 vCPU 64GB):	9.26 GiB (×1.07)
AlloyDB (tuned) (8 vCPU 64GB):	9.26 GiB (×1.07)
ClickHouse Cloud (aws) (12GiB, 1 replica(s)):	9.26 GiB (×1.07)
ClickHouse Cloud (aws) (12GiB, 3 replica(s)):	9.26 GiB (×1.07)
ClickHouse Cloud (gcp) (12GiB, 2 replica(s)):	9.26 GiB (×1.07)
ClickHouse Cloud (aws) (16GiB, 2 replica(s)):	9.26 GiB (×1.07)
ClickHouse Cloud (gcp) (12GiB, 1 replica(s)):	9.26 GiB (×1.07)
ClickHouse Cloud (gcp) (16GiB, 3 replica(s)):	9.26 GiB (×1.07)
ClickHouse Cloud (aws) (12GiB, 2 replica(s)):	9.26 GiB (×1.07)

1(?) из ~180

PostgresPro

pgpro_tam. TPC-H. Native formats.

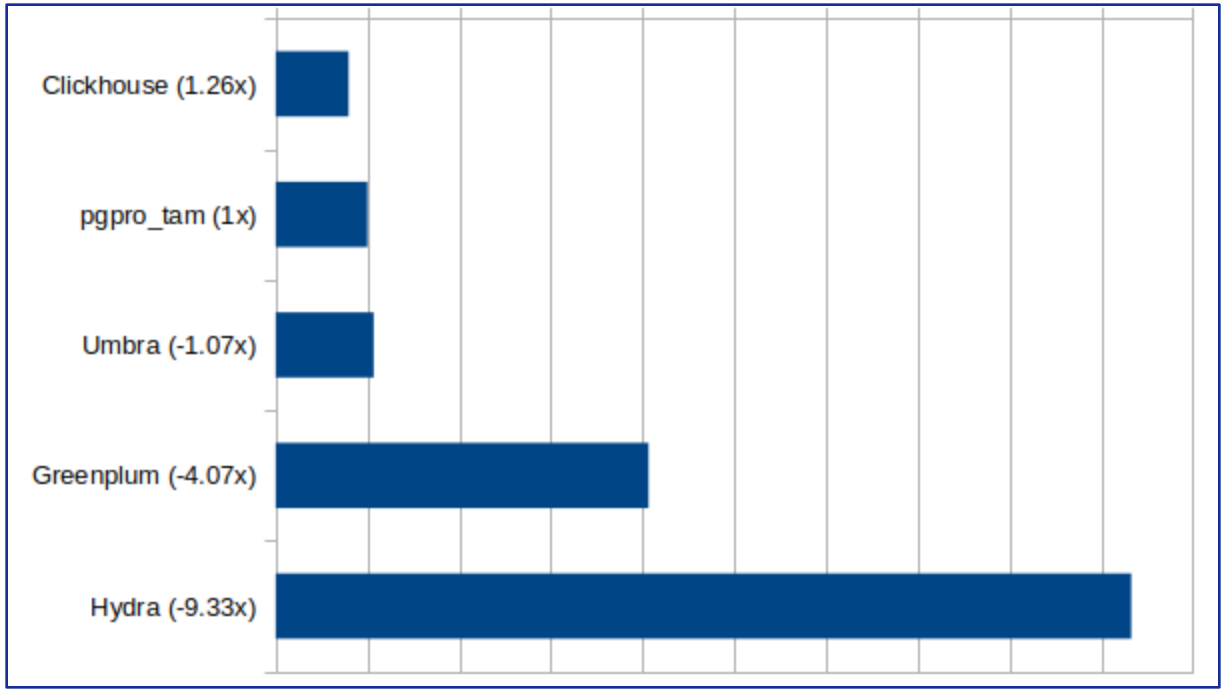


Дополнительно. Хорошо показали себя на малых объёмах, но не протестированы полноценно: SingleStore, StarRocks. Хорошо показали себя в ClickBench, но плохо в TPC-H: Oxla.

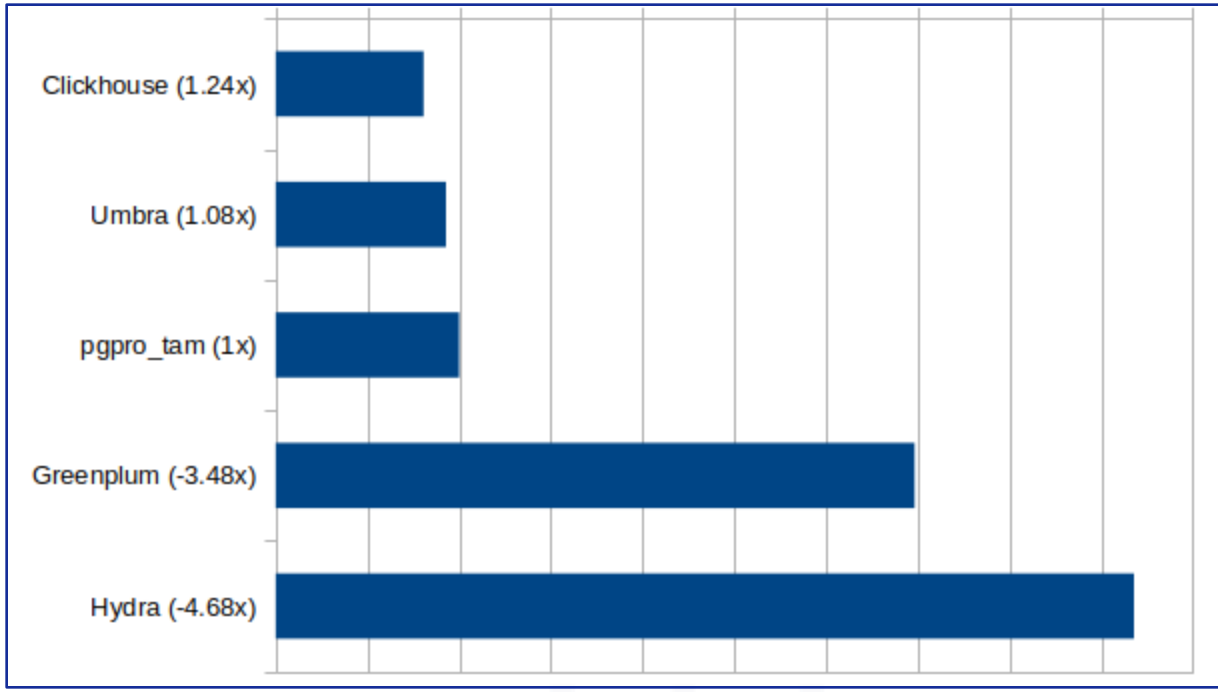
PostgresPro

pgpro_tam. TPC-H. Native formats. Среднее.

Average:



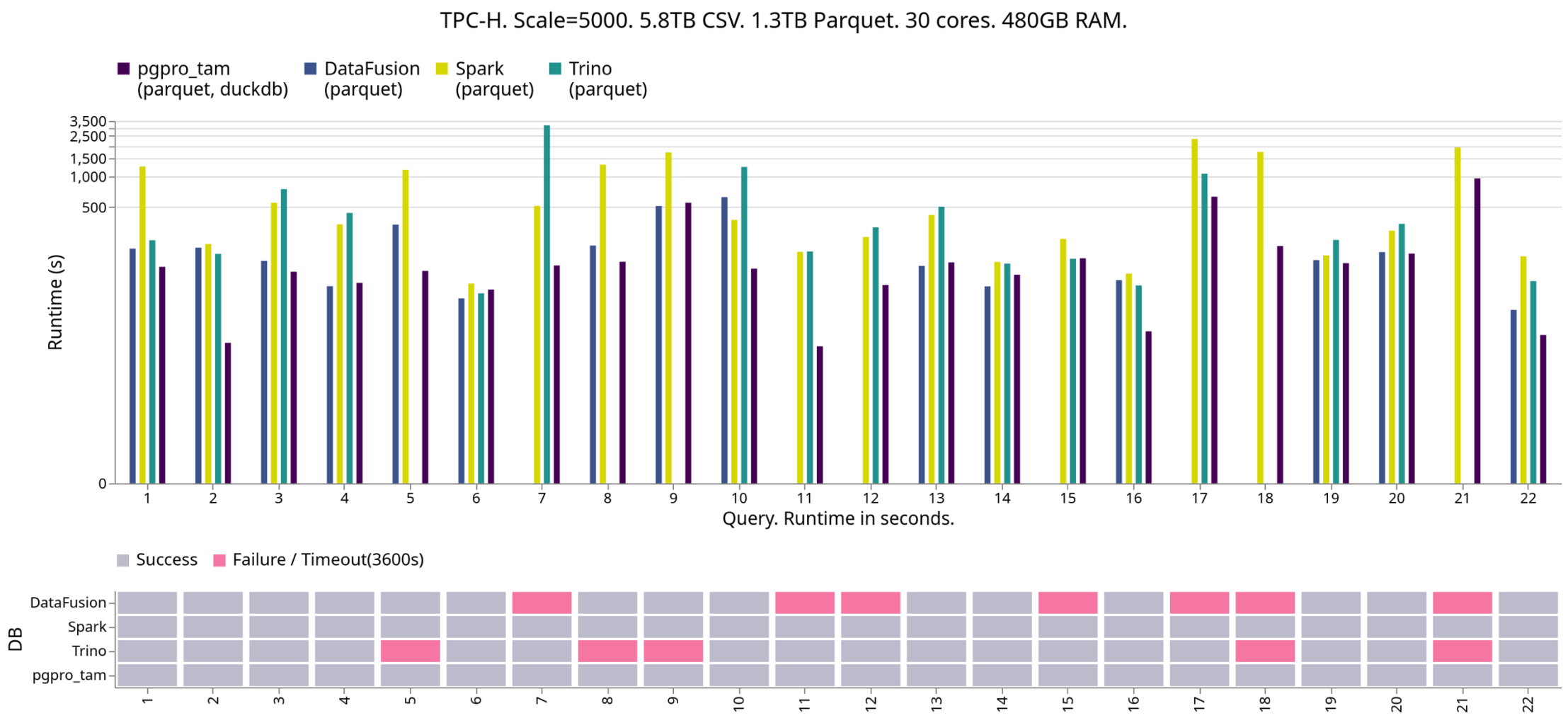
Median:



1. pgpro_tam: 22/22 queries
2. Greenplum: 22/22 queries
3. Umbra: 19/22 queries
4. Hydra: 16/22 queries
5. Clickhouse: 12/22 queries

PostgresPro

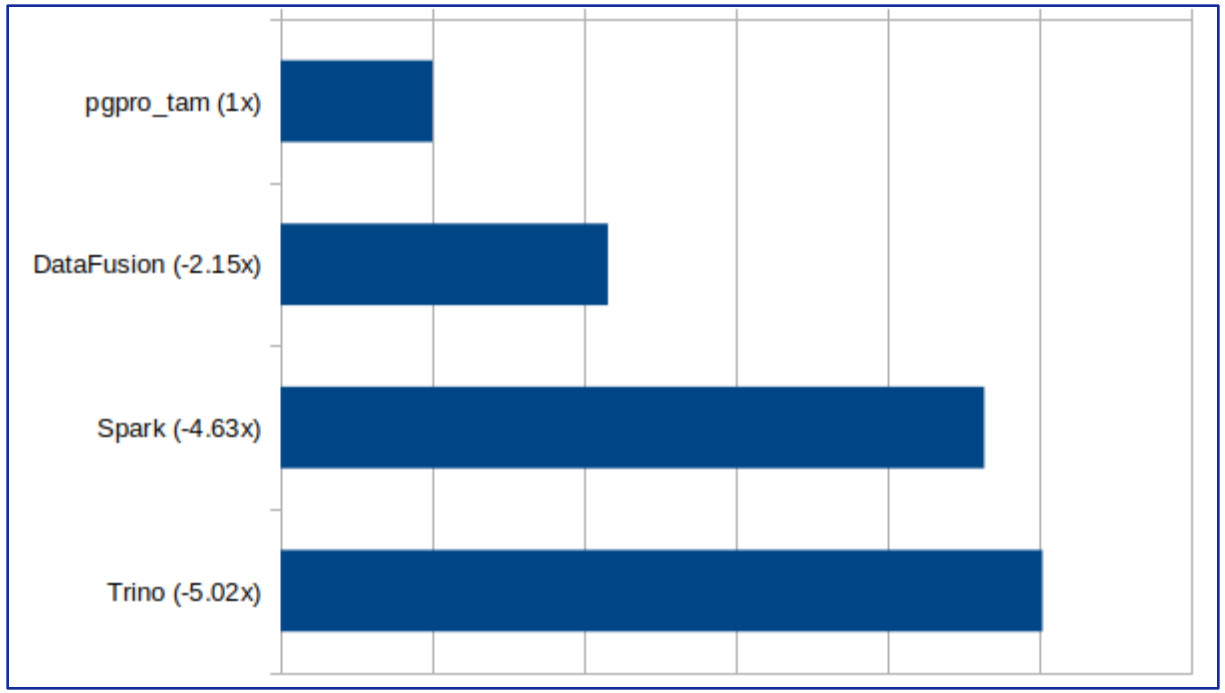
pgpro_tam. TPC-H. Parquet.



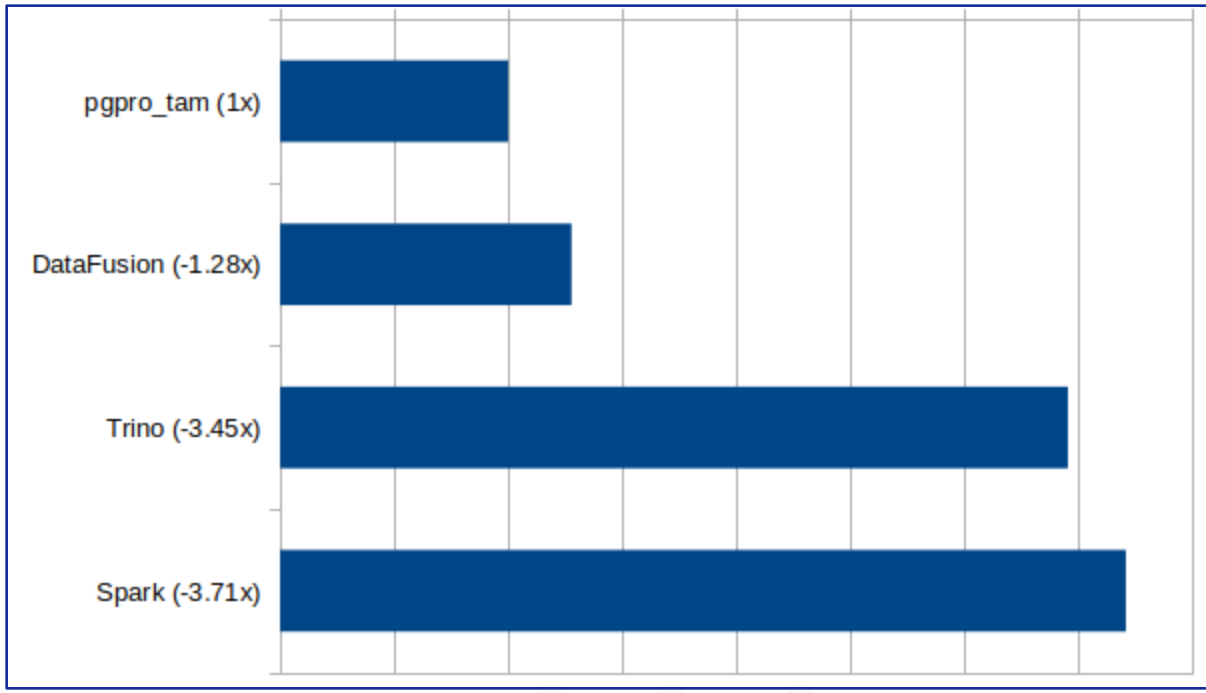
AMD EPYC 9354 30 vCores; 480GB RAM; 4x INTEL SSDPF2KX038XZ RAID10

pgpro_tam. TPC-H. Parquet. Среднее.

Average:



Median:

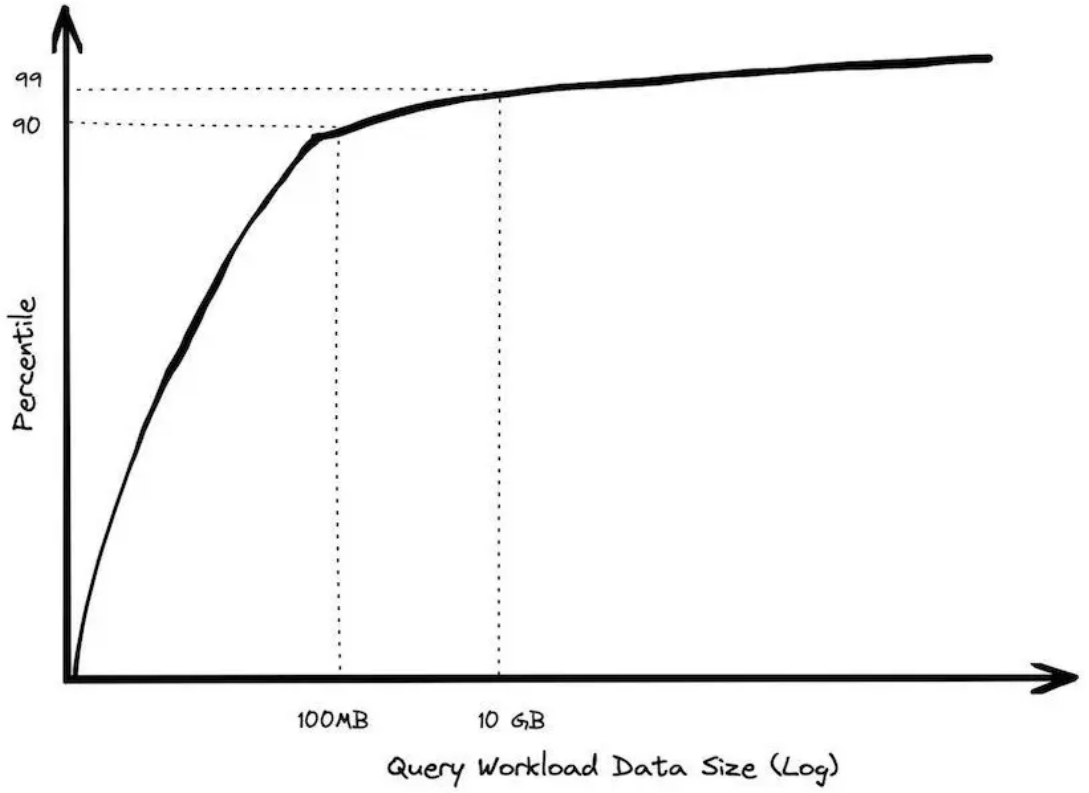


1. pgpro_tam: 22/22 queries
2. Spark: 22/22 queries
3. Trino: 17/22 queries
4. DataFusion: 15/22 queries

BIG DATA IS DEAD (?)

WORKLOAD SIZES ARE SMALLER THAN OVERALL DATA SIZES

MOST DATA IS RARELY QUERIED



https://motherduck.com/blog/big-data-is-dead/

Итоги

- Написать базовый движок – просто
- Написать полноценный движок – сложно, но вы знаете как
- Упал – вставай, встал – упай

- Postgres может в аналитику

pgpro_tam docker image:
https://hub.docker.com/r/innerlife/pgpro_tam

a.gordeev@postgrespro.ru
@innerlife0

PostgresPro

Спасибо за внимание!