PostgresPro

PGMeetup.PERM 2025

Превращаем Postgres в OLAP СУБД

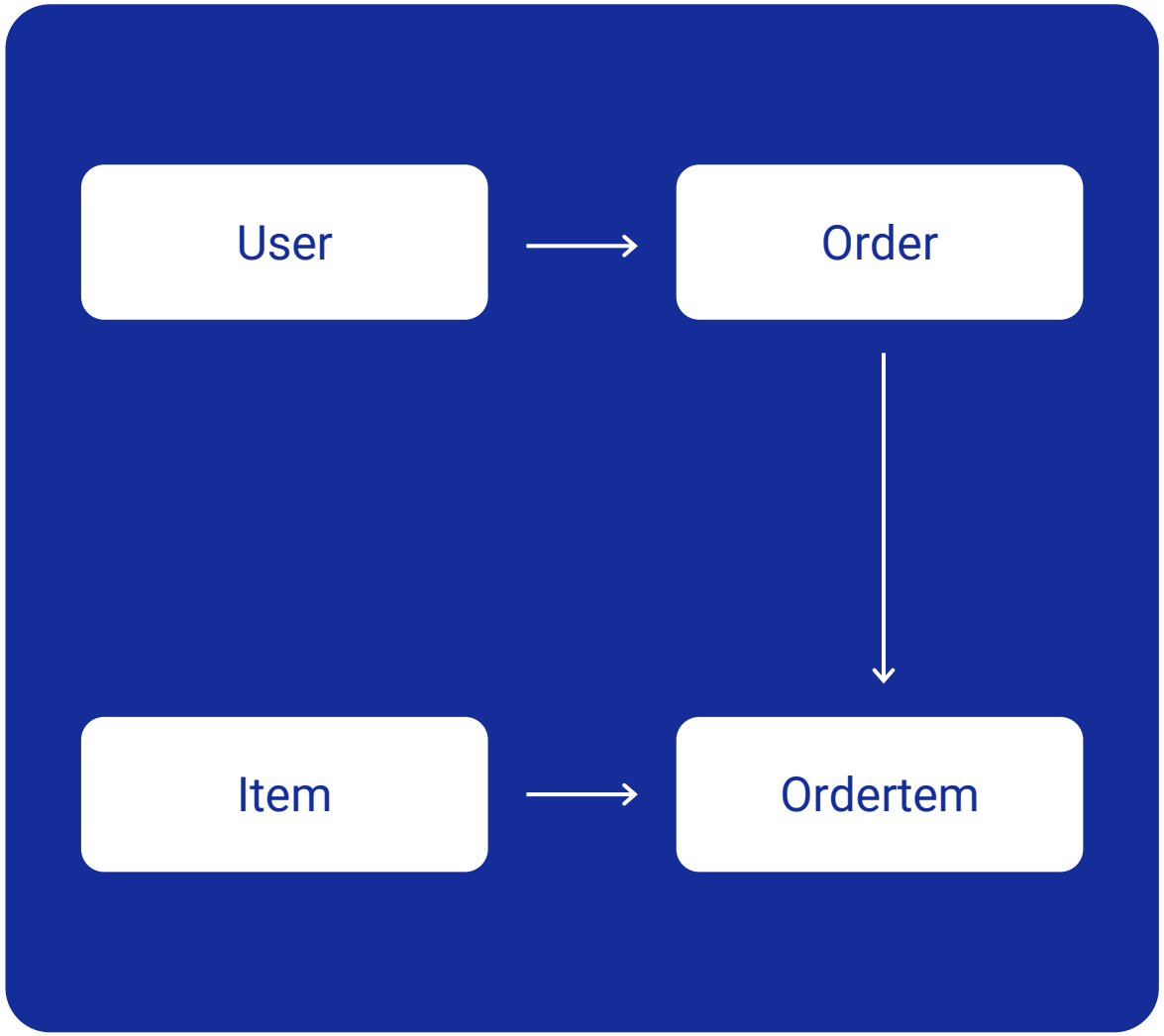
Алексей Гордеев
R&D engineer

OLTP

PostgresPro

Online Transaction Processing

- Большое количество пользователей
- Минимальное время отклика
- Точечная обработка данных
- Оптимизация выполнения небольших запросов

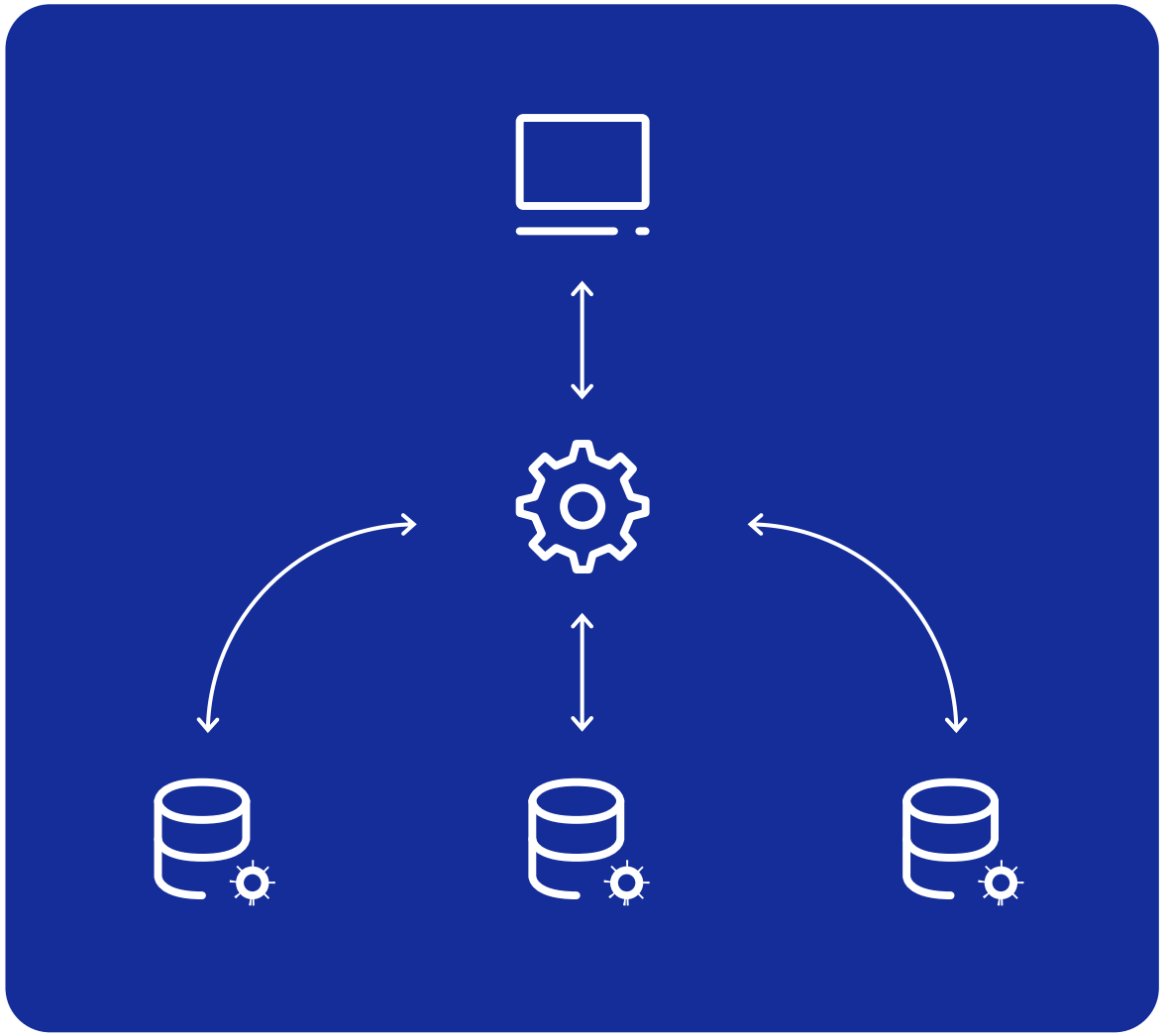


OLAP

PostgresPro

Online Analytical Processing

- Меньшее количество пользователей
- Большее время отклика
- Пакетная обработка данных
- Оптимизация выполнения больших запросов
- Нужен ли Sharding?



OLAP в Postgres?

- **Формат хранения данных**

  Используем подходящий для OLAP формат вместо heap

- **Проброс проекции**

  Читаем только колонки, используемые в запросе

- **Проброс предикатов**

  Используем условия фильтрации на уровне движка хранения

- **Пакетная вставка**

  Реализуем преимущества нового формата данных

- **Resource manager**

  Пишем в WAL новый формат данных

- **Поздняя материализация**

  Достаём значения строки на поздних этапах выполнения запроса

- **Векторное выполнение запросов**

  Обрабатываем по нескольку строк за раз

PostgresPro

Формат хранения данных

Строковое хранение

PostgresPro

N-ary storage model, NSM

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

heap1.dat

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

heap2.dat

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

Колоночное хранение

PostgresPro

Decomposition Storage Model, DSM

ID	Name	Birthdate
10	Ivan	14.08.1987
11	Peter	26.02.1984
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

PostgresPro

Partition Attributes Across

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

pax1.dat

Group 1

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

Group 2

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

Citus Columnar
PAX: https://www.pdl.cmu.edu/PDL-FTP/Database/pax.pdf

Строки vs Колонки

PostgresPro

Строки → OLTP

- SELECT одной строки
- DML одной строки

Колонки → OLAP

- Сжатие
- Чтение только используемых колонок (+ кэш)
- Векторное исполнение

Строки vs Колонки

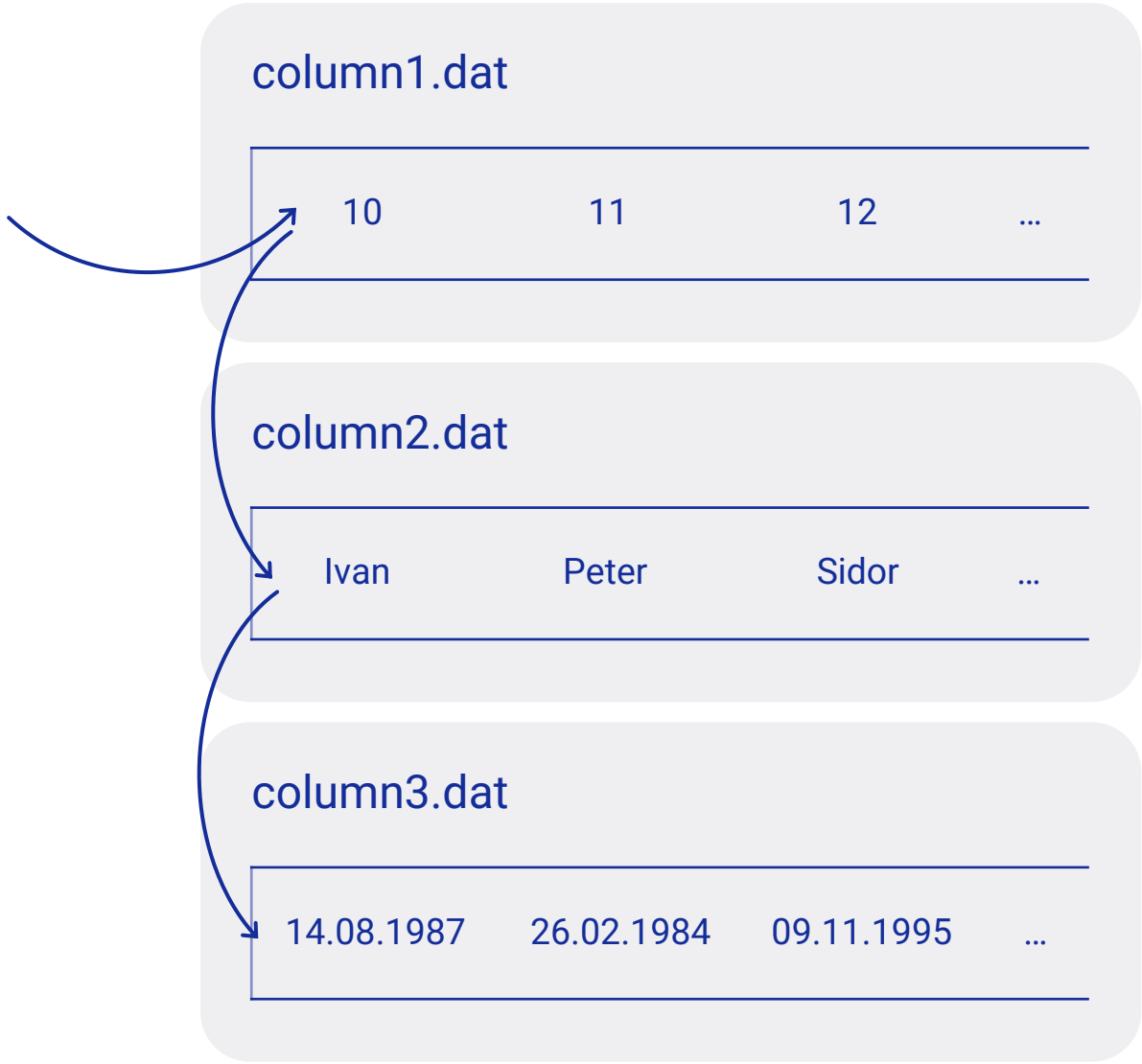
PostgresPro

Строки → OLTP

- SELECT одной строки
- DML одной строки

Колонки → OLAP

- Сжатие
- Чтение только используемых колонок (+ кэш)
- Векторное исполнение



Строки vs Колонки

PostgresPro

Строки → OLTP

- SELECT одной строки
- DML одной строки

Колонки → OLAP

- Сжатие
- Чтение только используемых колонок (+ кэш)
- Векторное исполнение

column1.dat

10	11	12	...

10 → 1000010
(1 million of sorted primary keys)

Delta Encoding

column1.dat

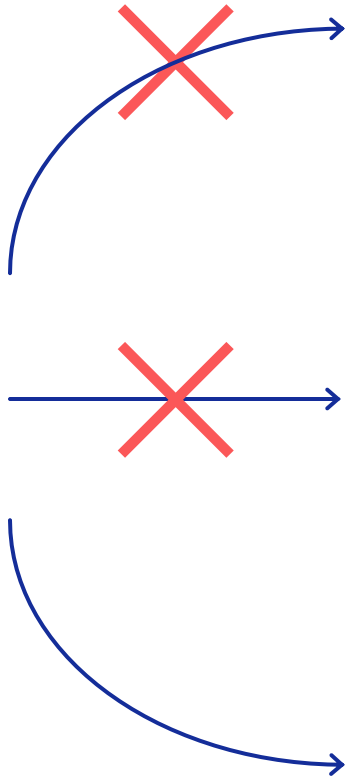
Start	Count	Delta
10	1000000	1

Строки vs Колонки

PostgresPro

select max(birthdate) from users;

Проброс проекции



column1.dat

10	11	12	...

column2.dat

Ivan	Peter	Sidor	...

column3.dat

14.08.1987	26.02.1984	09.11.1995	...

Строки vs Колонки

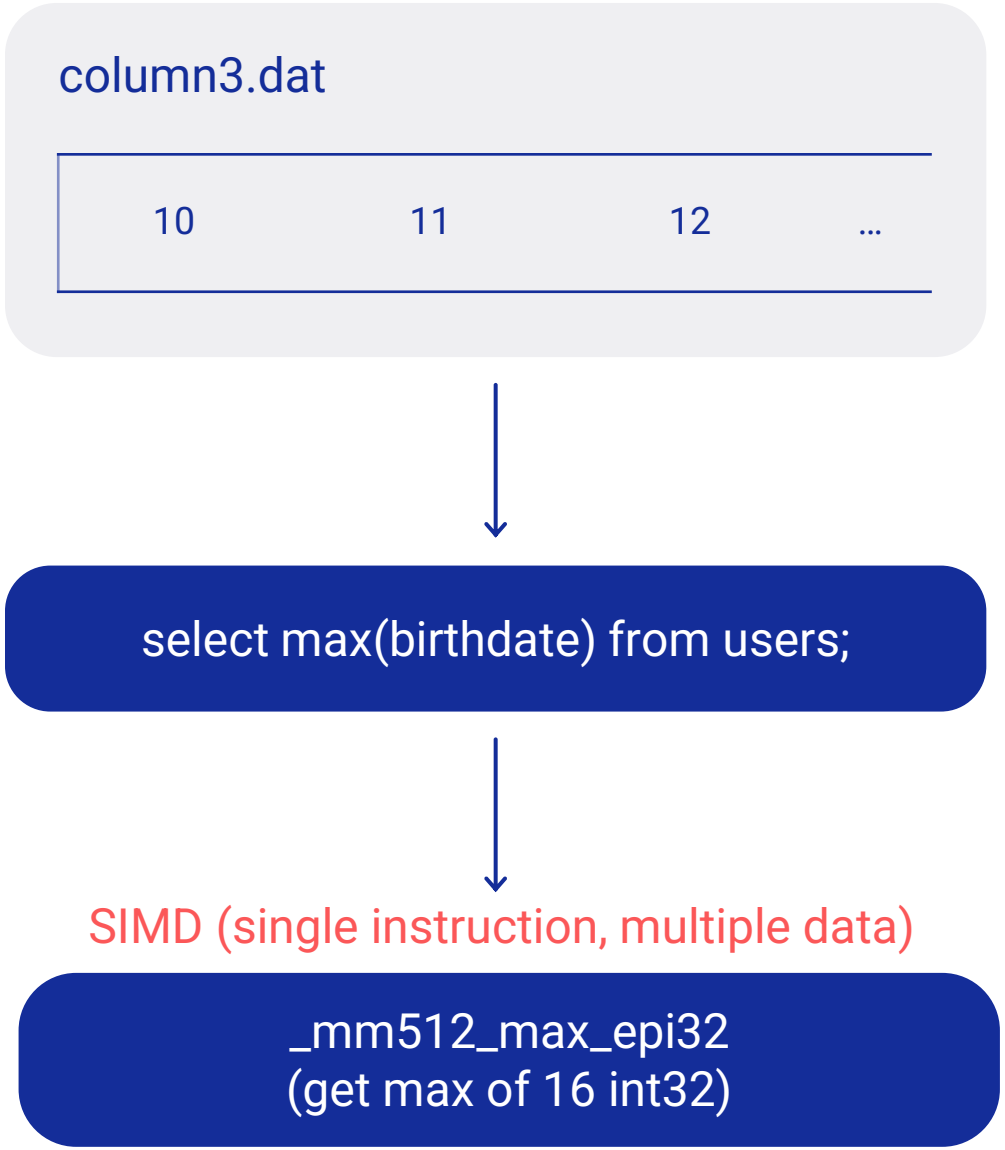
PostgresPro

Строки → OLTP

- SELECT одной строки
- DML одной строки

Колонки → OLAP

- Сжатие
- Чтение только используемых колонок (+ кэш)
- Векторное исполнение



PAX

PostgresPro

Строки → OLTP

- SELECT одной строки
- DML одной строки

Колонки → OLAP

- Сжатие
- Чтение только используемых колонок (+ кэш)
- Векторное исполнение

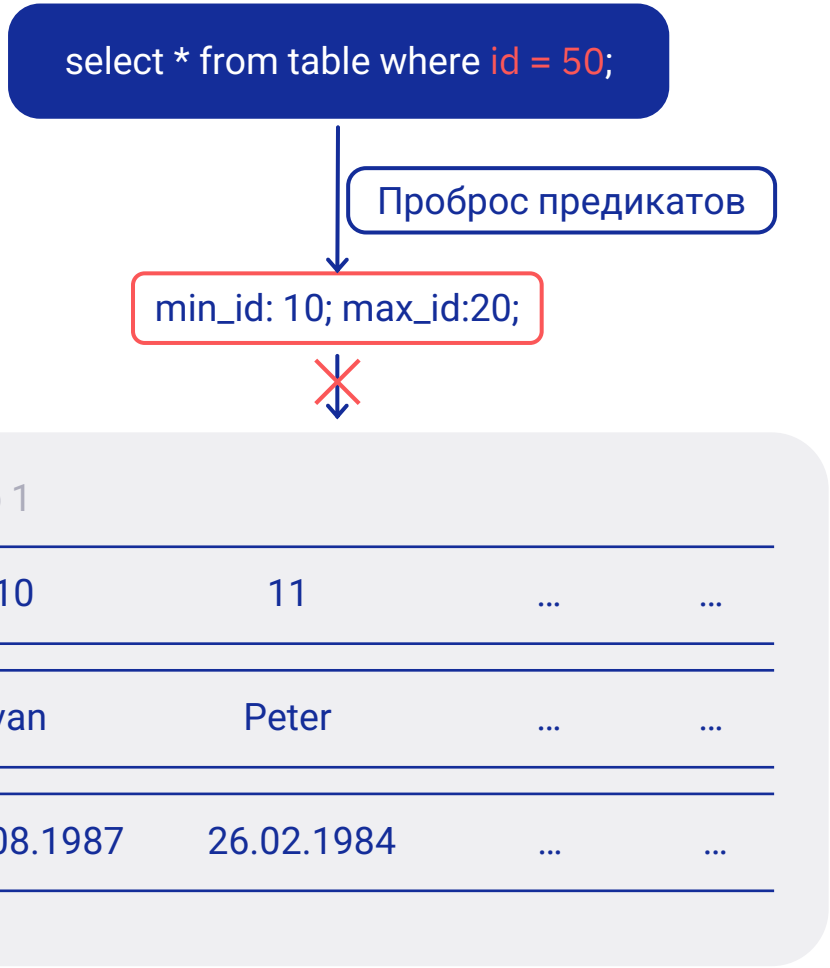
1

Строк
в группе

N

PAX

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



Расширяем возможности Postgres

PostgresPro

Postgres Extensions

1. Языки программирования
   plpython, plperl, ...
2. Агрегатные функции
   tdigest, hll, ...
3. Индексы
   bloom, btree_gin, ...
4. Мониторинг и отладка
   pg_stat_statements, auto_explain, ...
5. Табличные движки
   citus_columnar, zheap, ...
6. Подключение к внешним данным
   mysql_fdw, file_fdw, ...
7. И многое другое
   postgis, pgcrypto, citus, ...

```
postgres=# create extension bloom;
CREATE EXTENSION
postgres=# create table test_bloom(i int, j int);
CREATE TABLE
postgres=# insert into test_bloom select i, i from generate_series(0,100000) i;
INSERT 0 100001
postgres=# create index idx1_bloom on test_bloom using bloom(i, j);
CREATE INDEX
postgres=# explain (analyze, costs off) select * from test_bloom where i = 1000 and j = 1000;
                                 QUERY PLAN
-----------------------------------------------------------------------------
 Bitmap Heap Scan on test_bloom (actual time=0.947..0.948 rows=1 loops=1)
   Recheck Cond: ((i = 1000) AND (j = 1000))
   Heap Blocks: exact=1
   ->  Bitmap Index Scan on idx1_bloom (actual time=0.926..0.926 rows=1 loops=1)
         Index Cond: ((i = 1000) AND (j = 1000))
 Planning Time: 0.268 ms
 Execution Time: 0.979 ms
(7 rows)
```

Новый табличный движок

PostgresPro

TableAmRoutine, 40+ функций

```
typedef struct TableAmRoutine
{
    NodeTag     type;

    const TupleTableSlotOps *(*slot_callbacks) (Relation rel);

    TableScanDesc (*scan_begin) (Relation rel,
                                 Snapshot snapshot,
                                 int nkeys, struct ScanKeyData *key,
                                 ParallelTableScanDesc pscan,
                                 uint32 flags);
    void        (*scan_end) (TableScanDesc scan);
    void        (*scan_rescan) (TableScanDesc scan, struct ScanKeyData *key,
                                bool set_params, bool allow_strat,
                                bool allow_sync, bool allow_pagemode);
    bool        (*scan_getnextslot) (TableScanDesc scan,
                                     ScanDirection direction,
                                     TupleTableSlot *slot);
    void        (*scan_set_tidrange) (TableScanDesc scan,
                                      ItemPointer mintid,
                                      ItemPointer maxtid);
    bool        (*scan_getnextslot_tidrange) (TableScanDesc scan
```

Postgres: src/include/access/tableam.h; TableAmRoutine
https://pgconf.ru/talk/2032287

Citus Columnar

Citus Columnar — расширение для Postgres, которое добавляет поддержку колоночного (PAX) хранения данных в таблицах.

Citus — расширение для Postgres, превращающее его в распределённую СУБД (шардированную кластерную систему).

Плюсы:

- Проброс проекции
- Проброс предикатов
- Сжатие

Минусы:

- Собственный формат данных
- Потенциал не раскрыть построчным экзекьютором Postgres

Другие движки:

- Greenplum AO (с патчами ядра)
- Timescale Hypertables
- Hydra (Citus Columnar fork)
- ZHeap (RIP)
- Zedstore (RIP)

Parquet

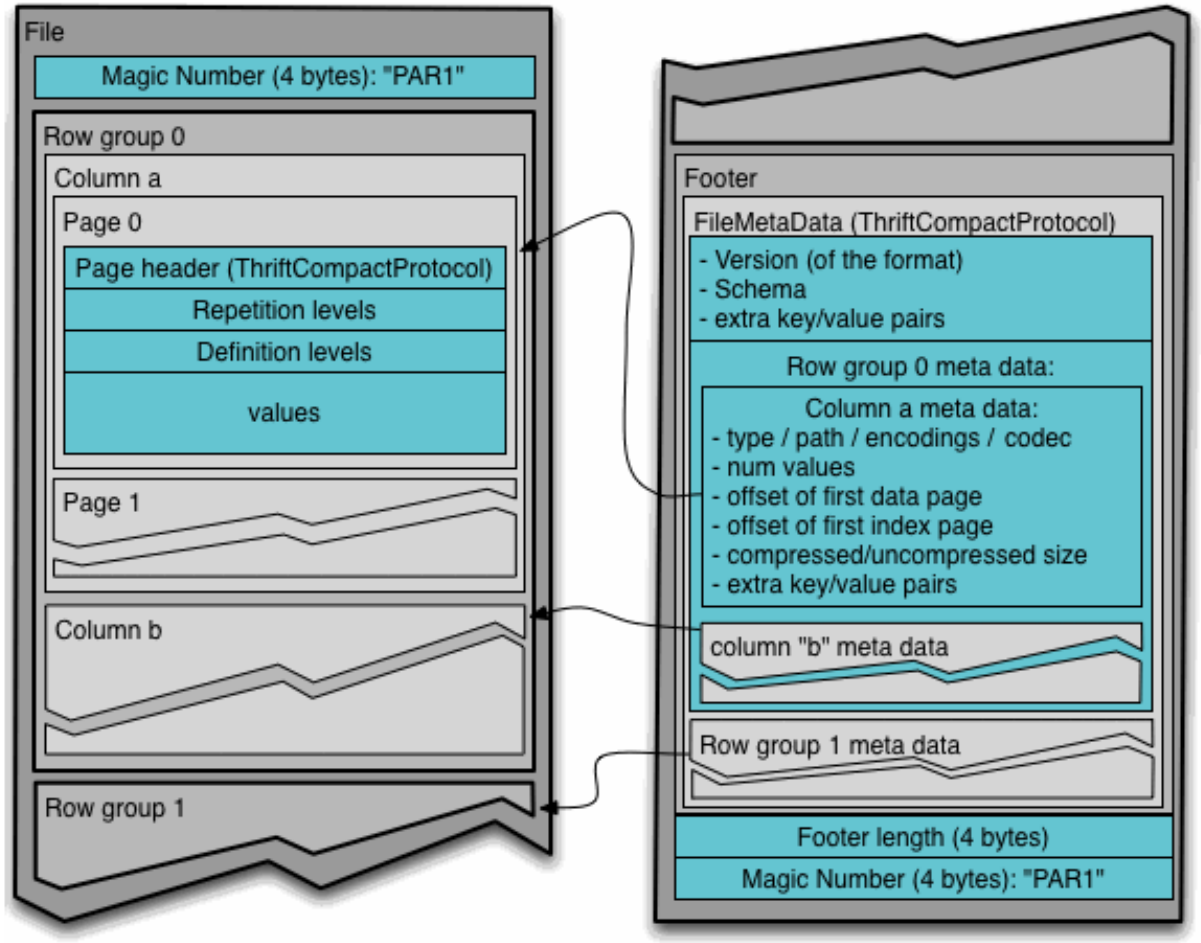
PostgresPro

Формат основанный на идеях PAX

Фичи:

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

Для работы с форматом внутри Postgres можно использовать libarrow



https://parquet.apache.org/docs/file-format/
https://github.com/apache/arrow

PostgresPro

Проброс проекции

Нет проброса проекции

PostgresPro

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» ?

Проброс проекции через CustomScan

PostgresPro

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

PostgresPro

Проброс предикатов

Нет проброса предикатов

PostgresPro

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;
```

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

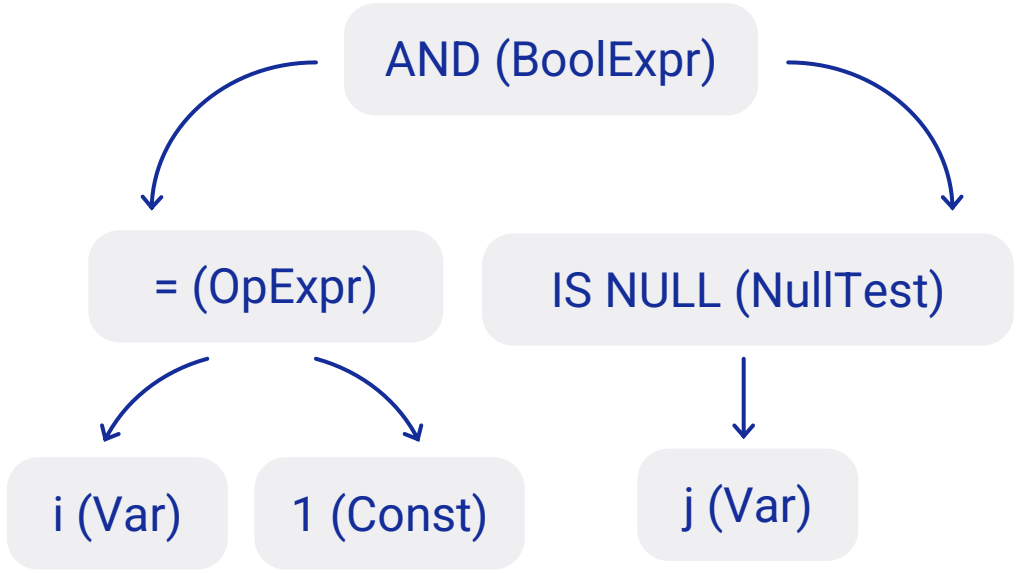
Postgres: src/backend/executor/nodeSeqscan.c

Проброс предикатов через CustomScan

PostgresPro

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

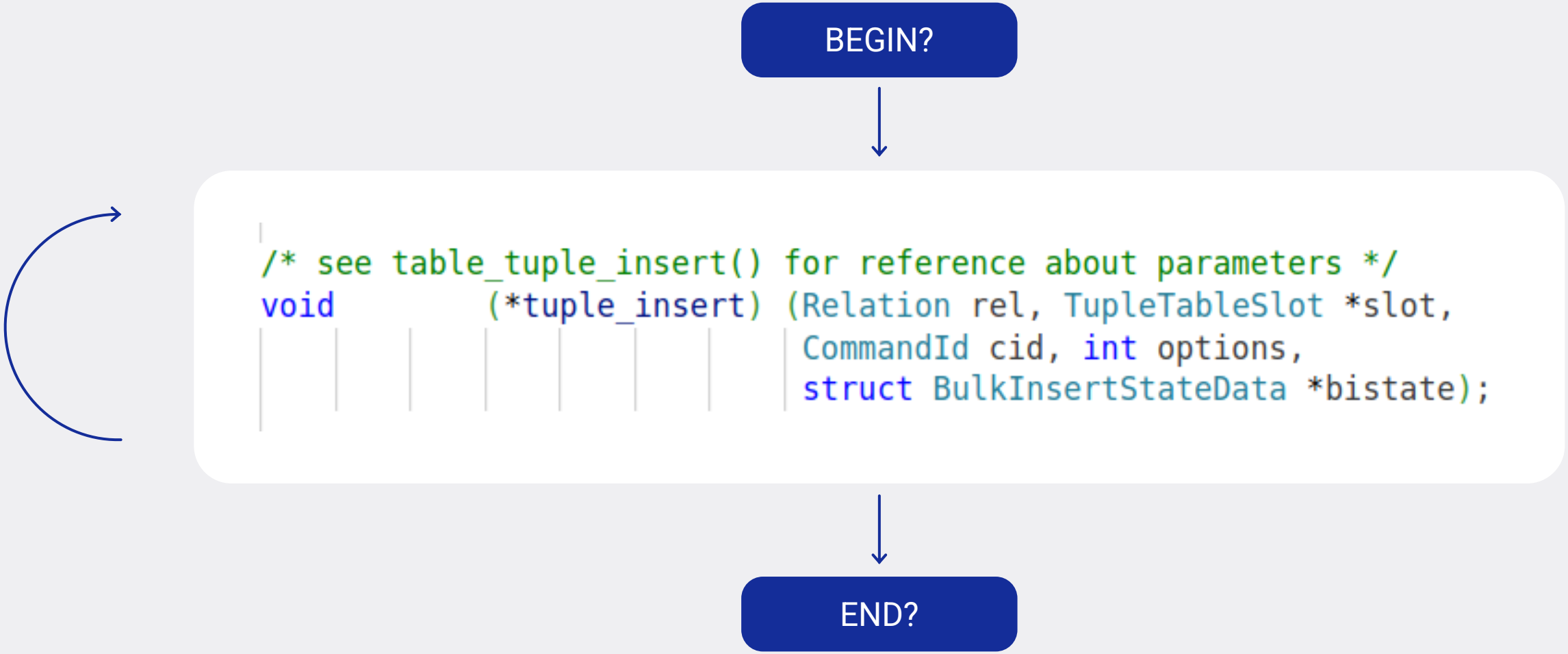
PostgresPro

Пакетная вставка

Stateless Write

PostgresPro

Нет методов begin и end



Запись через буфер + Transaction Callback

PostgresPro

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.

```
typedef void (*XactCallback) (XactEvent event, void *arg);
```

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

```
typedef enum
{
    XACT_EVENT_COMMIT,
    XACT_EVENT_PARALLEL_COMMIT,
    XACT_EVENT_ABORT,
    XACT_EVENT_PARALLEL_ABORT,
    XACT_EVENT_PREPARE,
    XACT_EVENT_PRE_COMMIT,
    XACT_EVENT_PARALLEL_PRE_COMMIT,
    XACT_EVENT_PRE_PREPARE,
} XactEvent;
```

Example. Citus Columnar: src/backend/columnar/write_state_management.c
Postgres: src/backend/access/transam/xact.c

PostgresPro

Resource manager

WAL. Generic XLog.

PostgresPro

- 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);
```

Postgres: src/include/access/generic_xlog.h
Example. Citus Columnar: src/backend/columnar/columnar_storage.c; WriteToBlock

WAL. Custom RMGR.

PostgresPro

- 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/xlog_internal.h
Example. Postgres: src/test/modules/test_custom_rmgrs/test_custom_rmgrs.c

PostgresPro

Поздняя материализация

Нет поздней материализации

PostgresPro

```
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

Поздняя материализация в Postgres

PostgresPro

Есть 2 решения

~~Патчи~~ ПАТЧИ в ядро

Сторонний executor
(самописный или готовый)

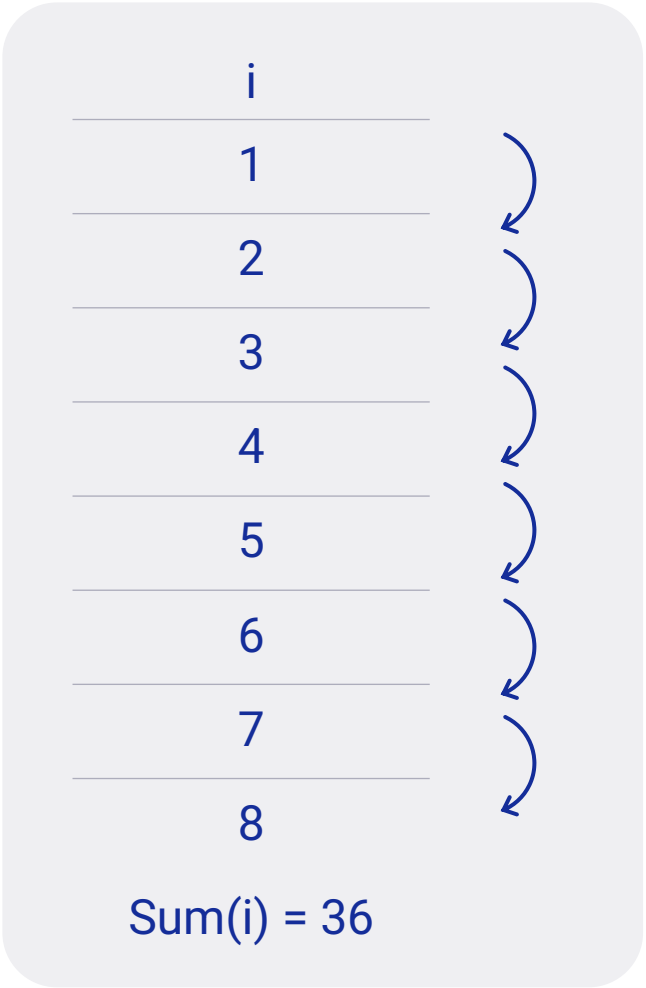
PostgresPro

Векторное выполнение запросов

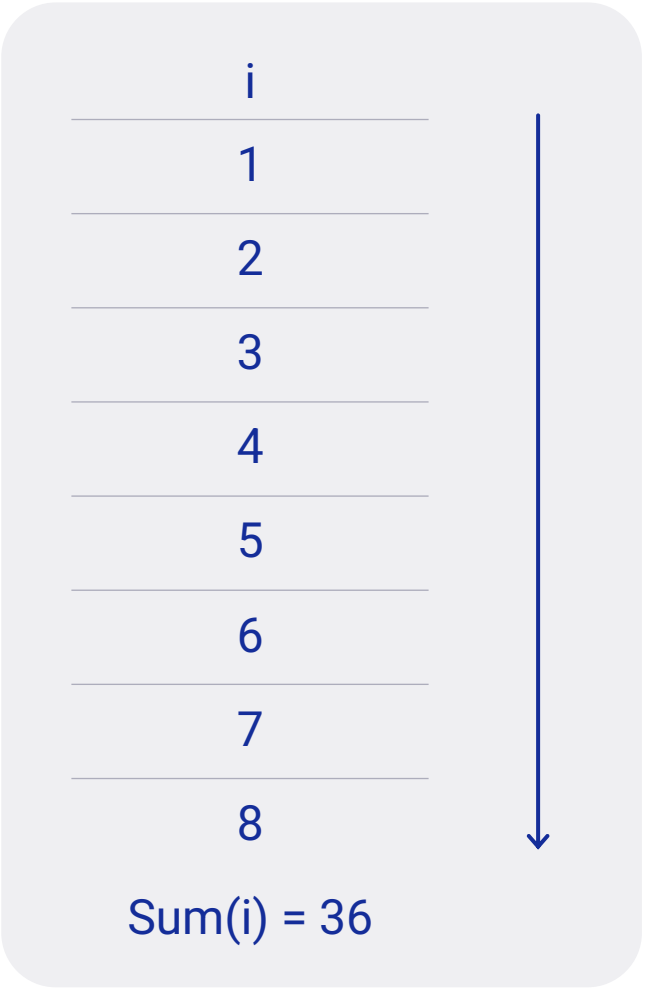
Нет векторизации

PostgresPro

Per tuple execution



Vectorized execution



Нет векторизации

PostgresPro

```
postgres=# explain
select sum(test1.i + test2.i)
from test1 join test2 on test1.j = test2.j
where test1.i > 10
and test2.i > 100;
                                 QUERY PLAN
-----------------------------------------------------------------------------
 Aggregate  (cost=6508.28..6508.29 rows=1 width=8)
 ->  Hash Join  (cost=2941.85..6008.80 rows=99897 width=8)
       Hash Cond: (test1.j = test2.j)
       ->  Seq Scan on test1  (cost=0.00..1693.01 rows=99991 width=8)
             Filter: (i > 10)
       ->  Hash  (cost=1693.01..1693.01 rows=99907 width=8)
             ->  Seq Scan on test2  (cost=0.00..1693.01 rows=99907 width=8)
                   Filter: (i > 100)
(8 rows)
```

Векторное выполнение в Postgres

PostgresPro

Есть 2 решения

Еще ПАТЧИ в ядро

Сторонний executor

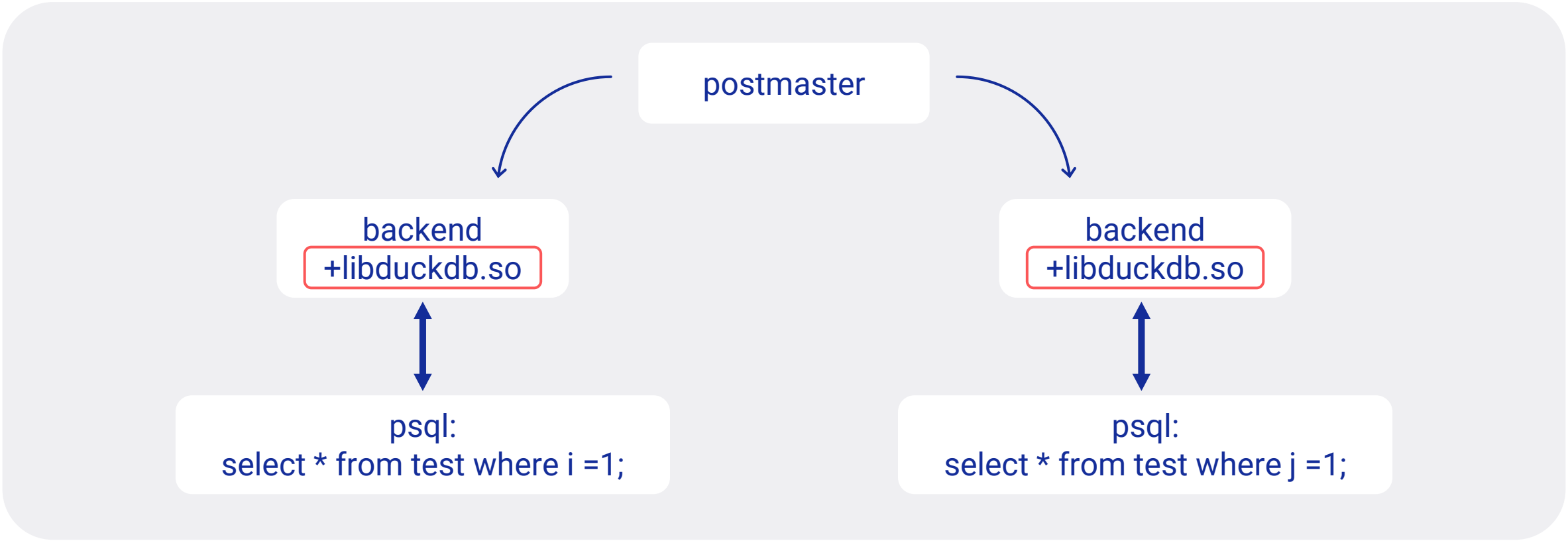
(самописный или готовый)

Сторонний исполнитель. Duck Tales.

PostgresPro

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.

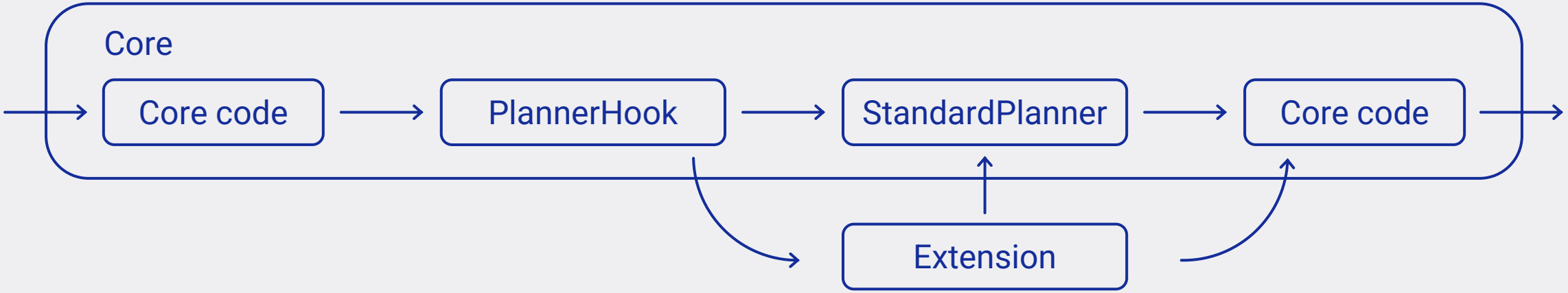


https://github.com/duckdb/duckdb
https://github.com/duckdb/pg_duckdb

Сторонний исполнитель. Planner Hook.

PostgresPro

```
/* Hook for plugins to get control in planner() */
Fujii Masao, 5 years ago | 2 authors (Tom Lane and one other)
typedef PlannedStmt *(*planner_hook_type) (Query *parse,
                                           const char *query_string,
                                           int cursorOptions,
                                           ParamListInfo boundParams);
extern PGDLLIMPORT planner_hook_type planner_hook;
```

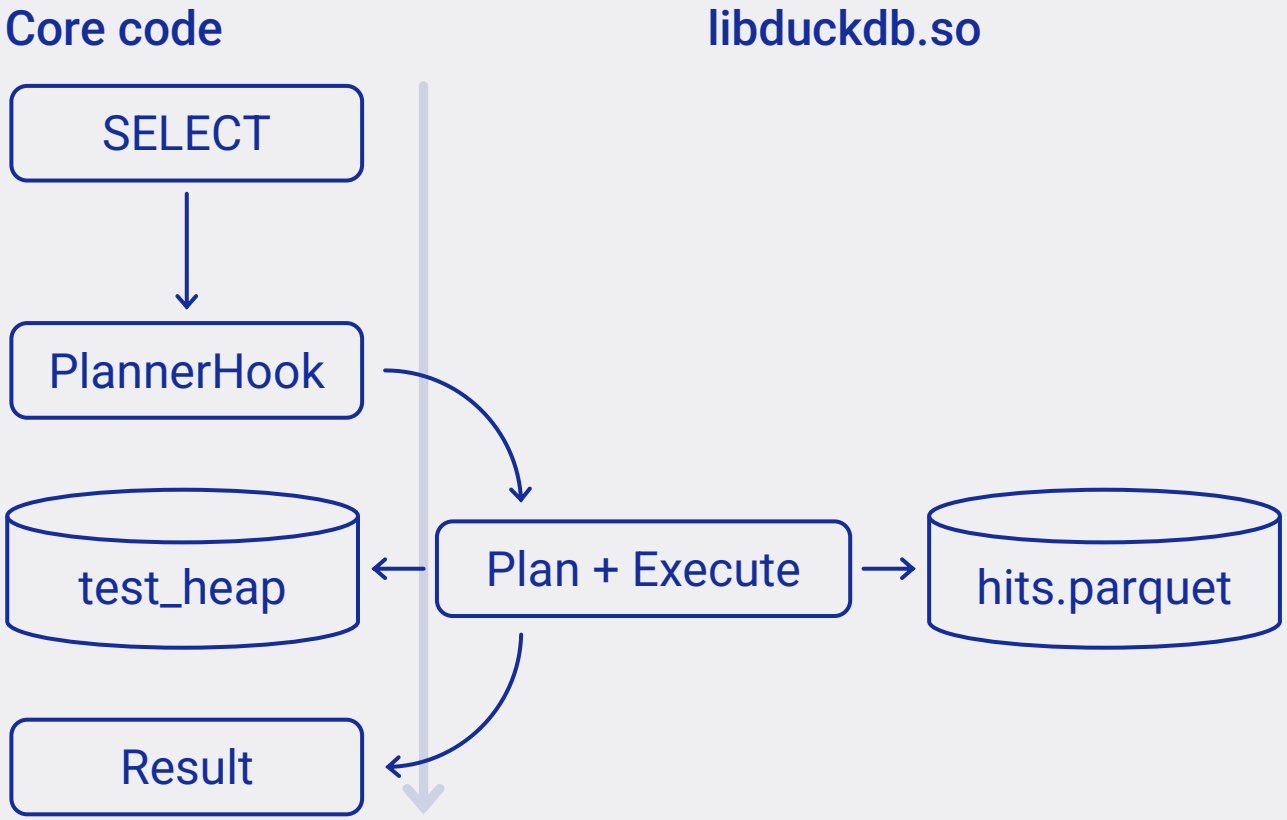


Postgres: src/include/optimizer/planner.h

DuckDB + Parquet + Postgres

PostgresPro

```
explain select watchid
from hits
where 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│
```

OLAP в Postgres!

- **Формат хранения данных**

  Parquet через libarrow -> TableAmRoutine

- **Проброс проекции**

  Для стандартного исполнителя через CustomScan

- **Проброс предикатов**

  Для стандартного исполнителя через CustomScan

- **Пакетная вставка**

  State + Buffer + Transaction Callback

- **Resource manager**

  Custom RMGR – пишем Parquet

- **Поздняя материализация**

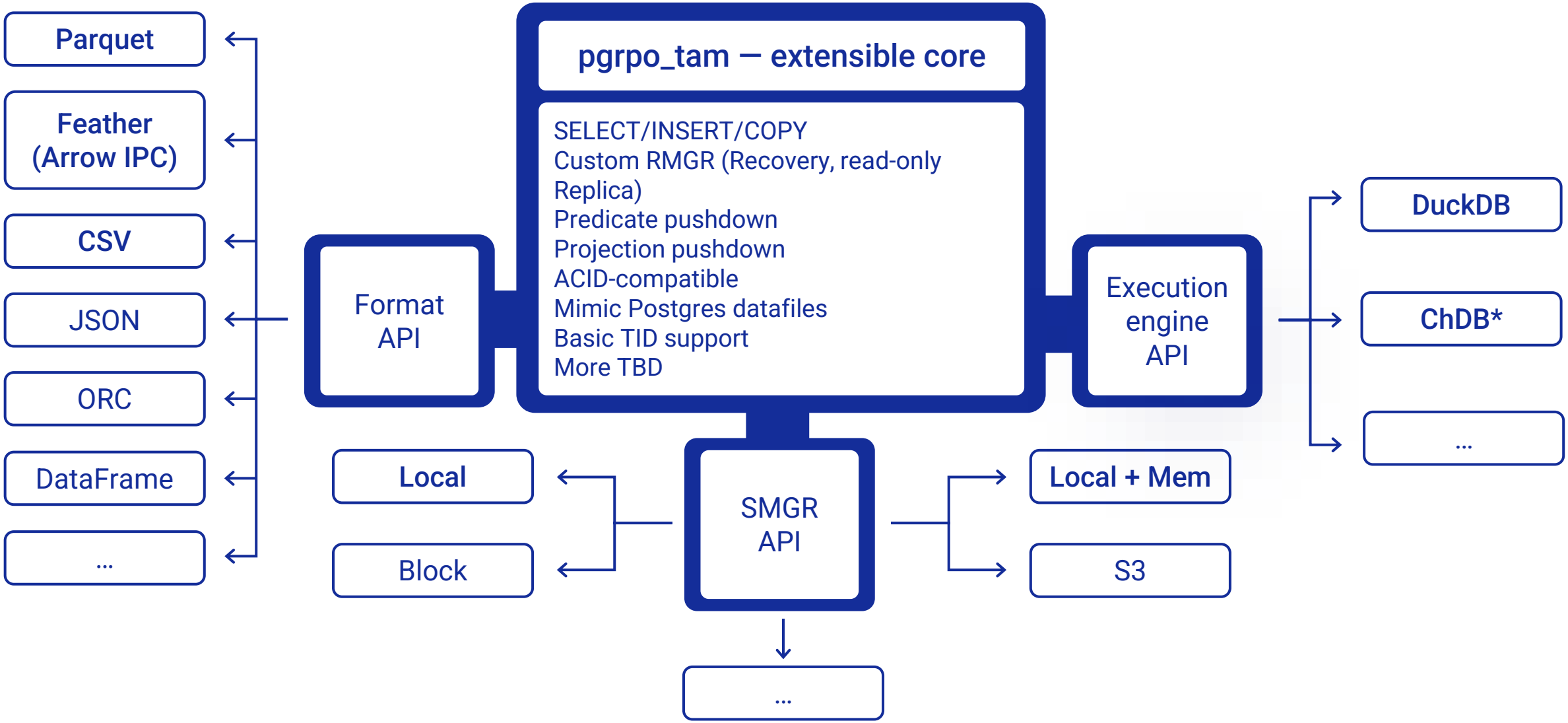
  Не поддерживается в DuckDB при работе с parquet, да и…

- **Векторное выполнение запросов**

  DuckDB (+ проброс проекции и предикатов)

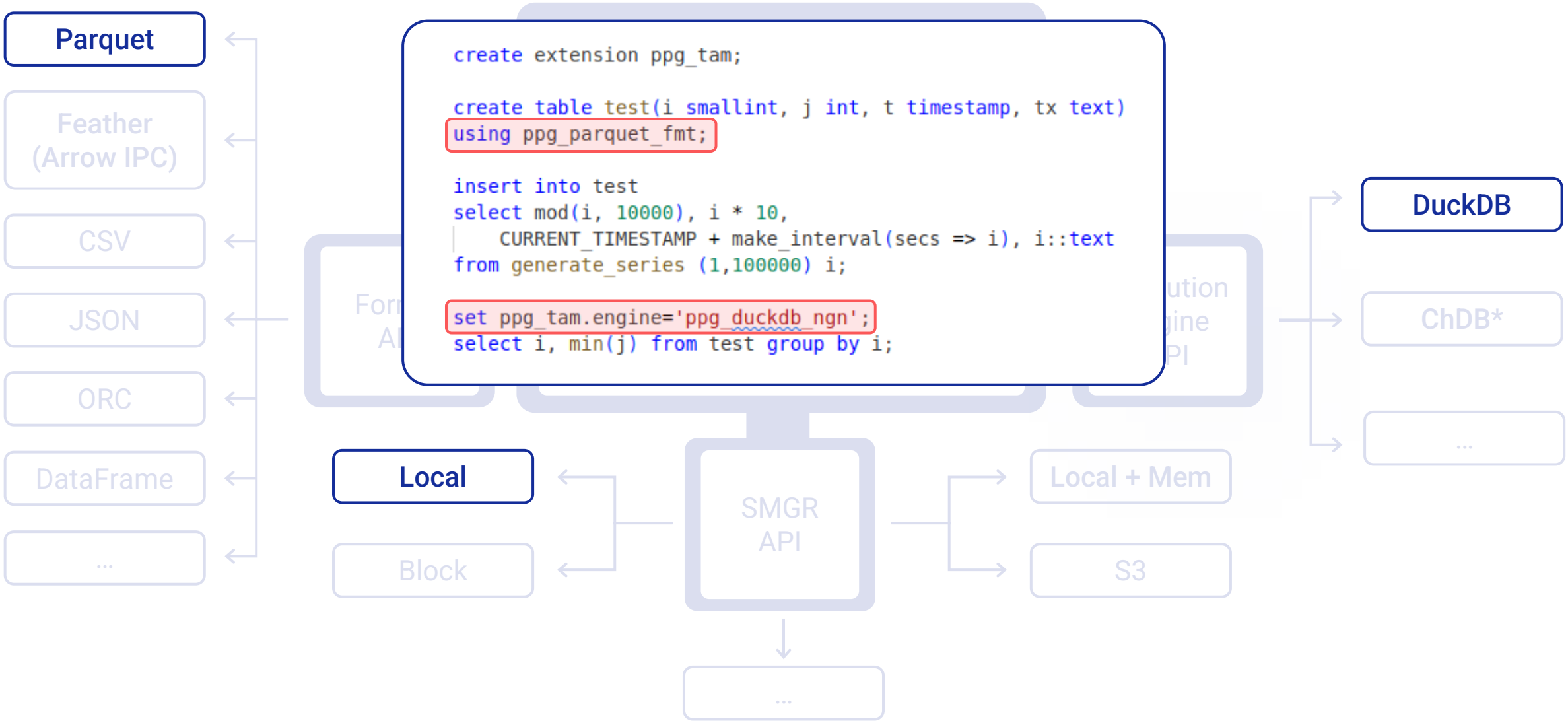
pgpro_tam. Extension for vanilla Postgres.

PostgresPro



pgpro_tam. Universal analytic engine.

PostgresPro

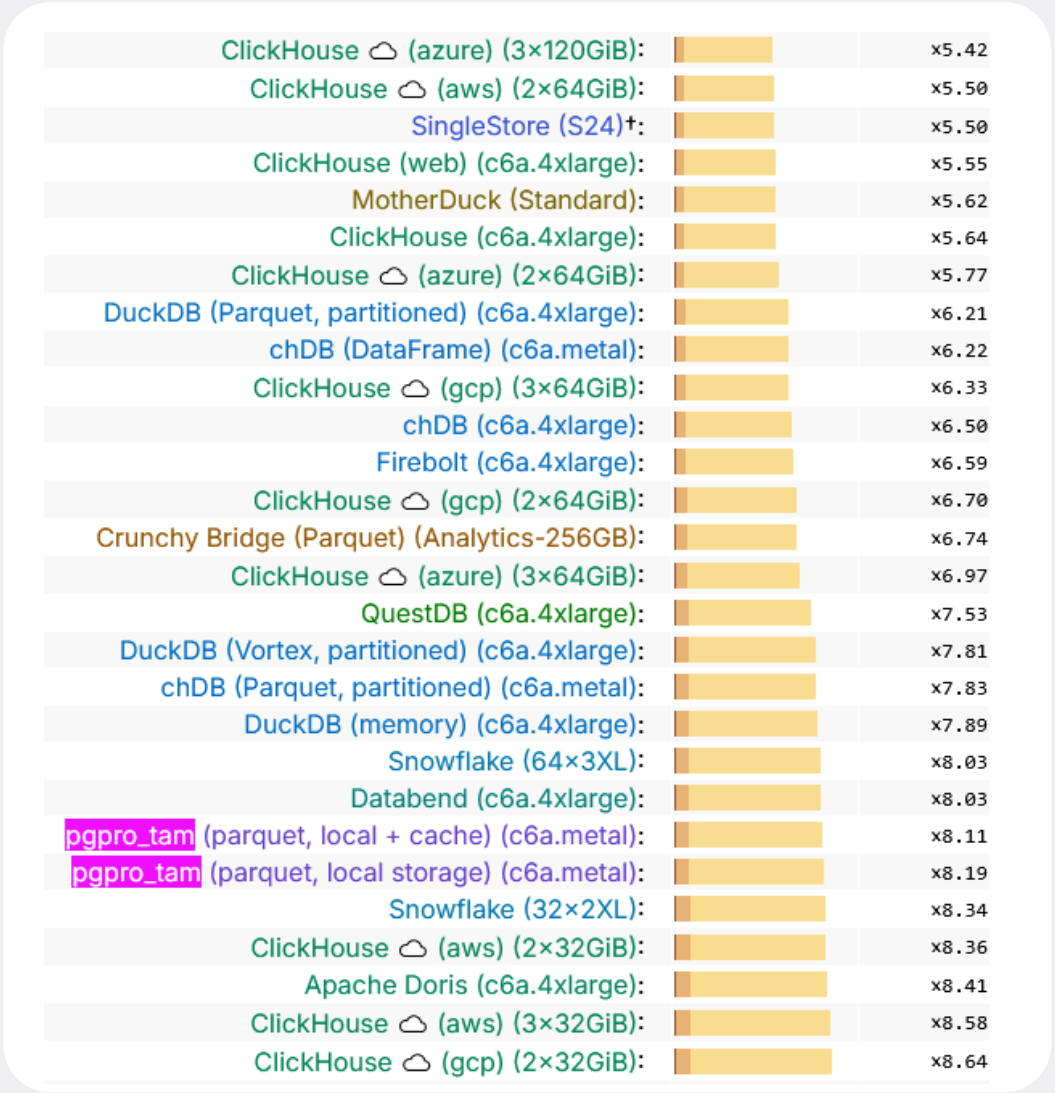


PostgresPro

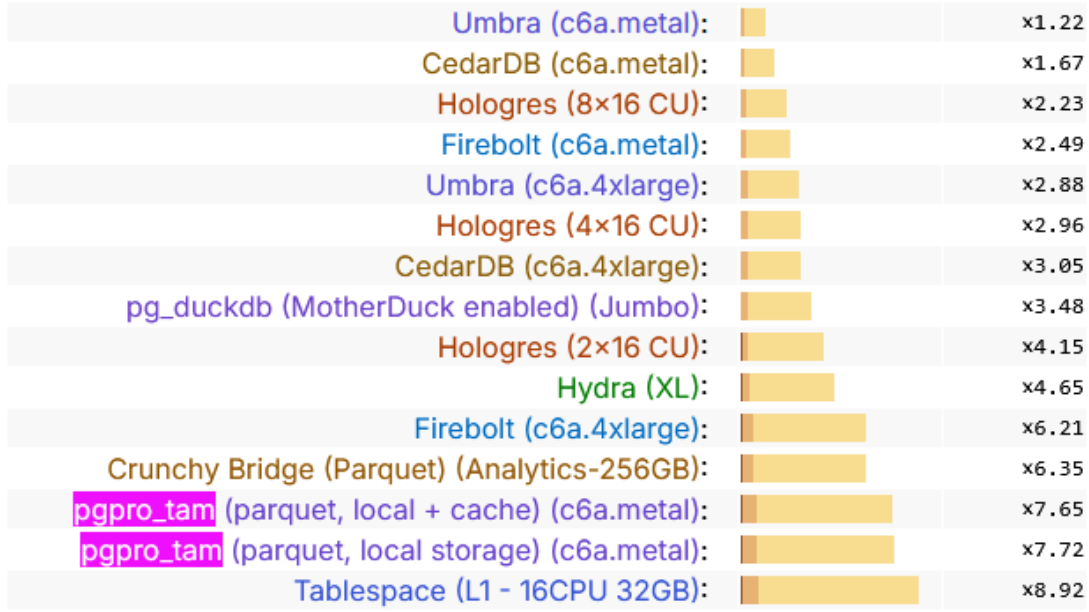
Benchmarks

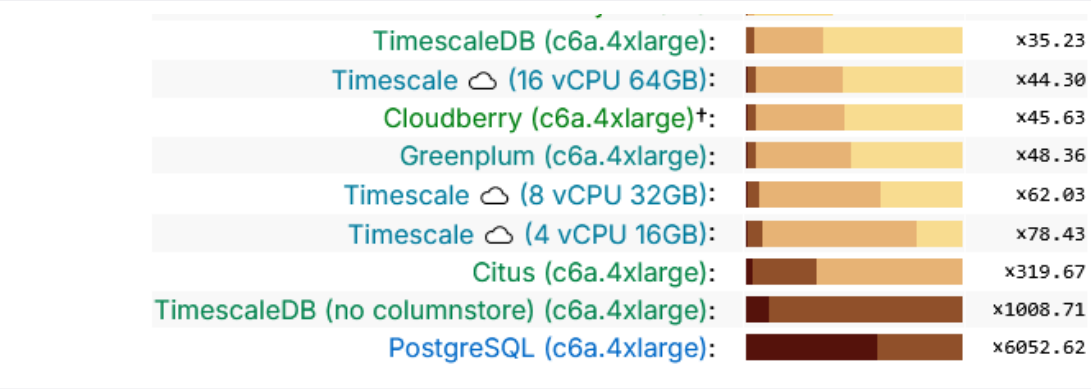
pgpro_tam. ClickBench. Hot run.

PostgresPro



63 из ~210

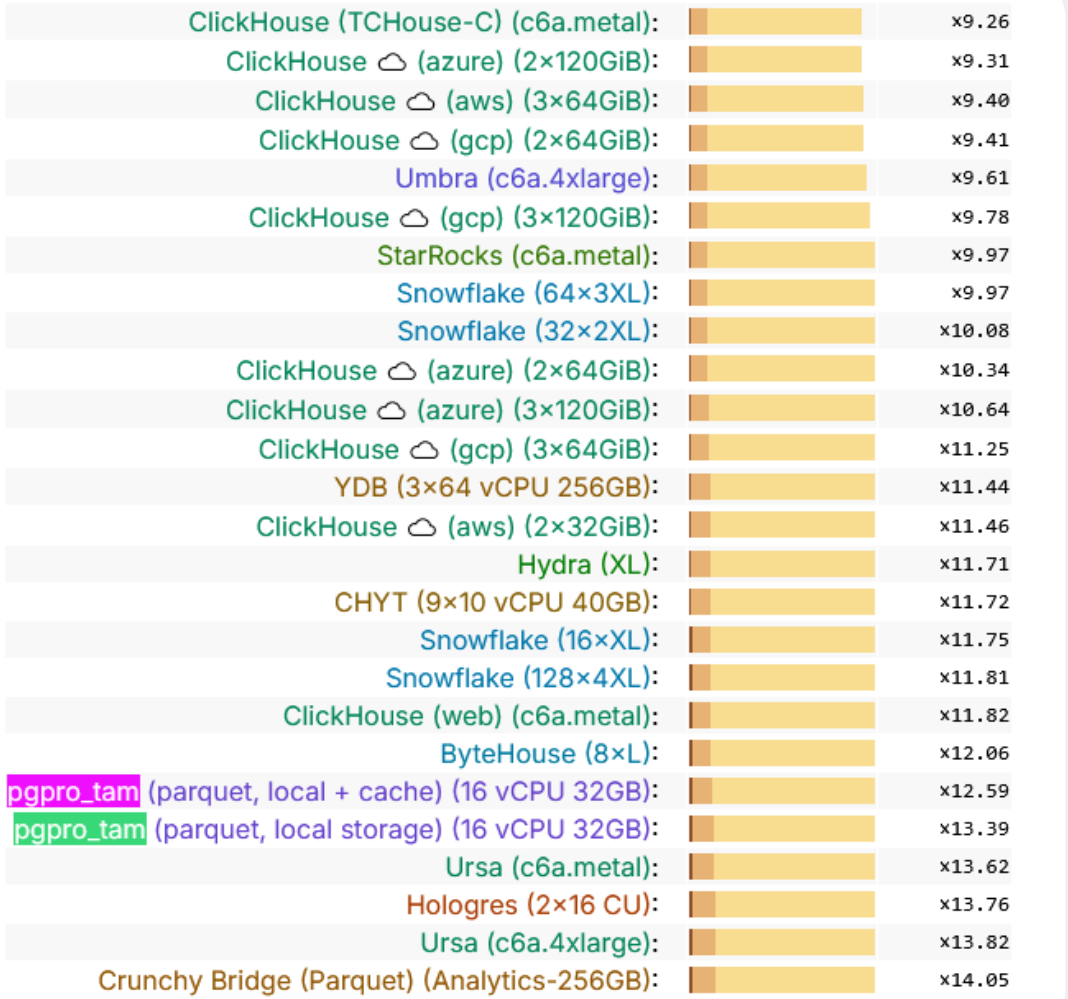




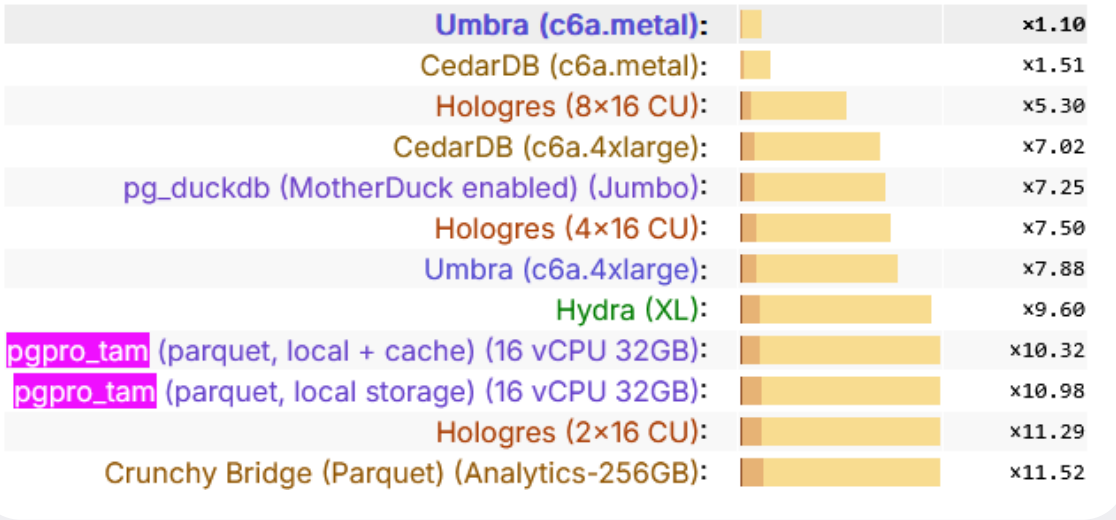
13 из ~42 (Postgres compatible)

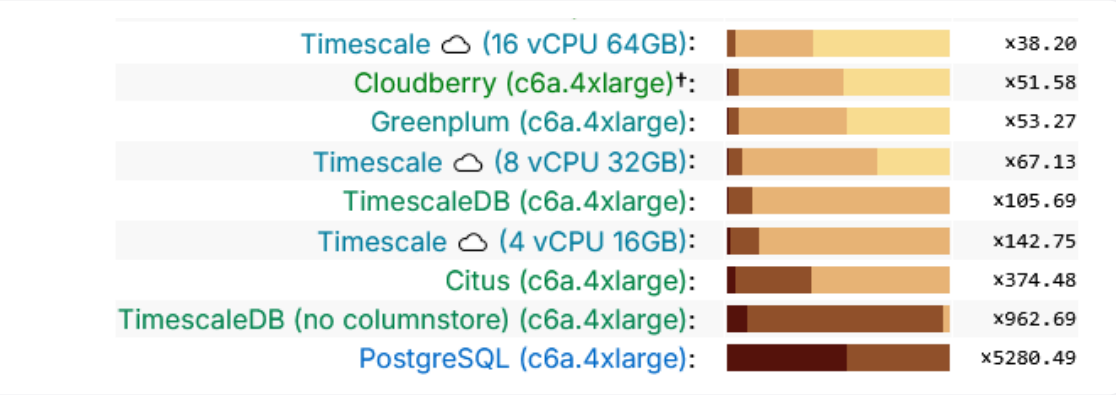
pgpro_tam. ClickBench. Cold run.

PostgresPro



41 из ~210





9 из ~42 (Postgres compatible)

pgpro_tam. ClickBench. Storage Size.

PostgresPro

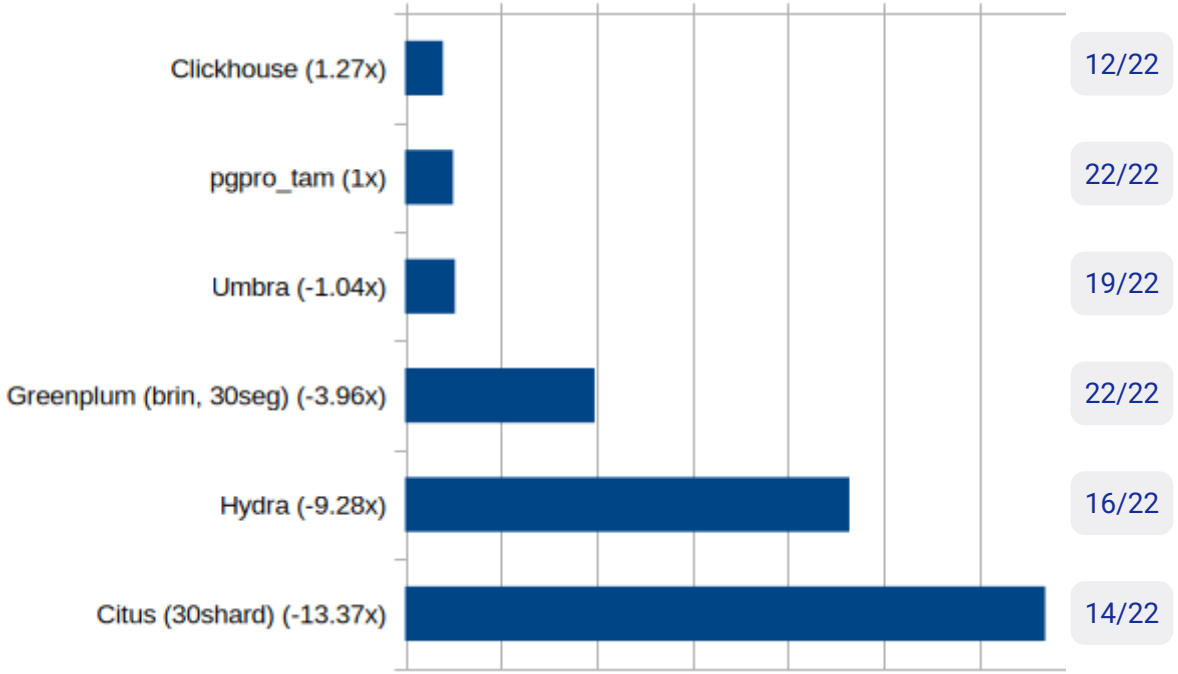
Bigquery (serverless):	8.16 GiB (x1.00)
pgpro_tam (parquet, local, parallel) (16 vCPU 32GB):	**8.66 GiB (x1.06)**
pgpro_tam (parquet, local, parallel) (c6a.metal):	8.66 GiB (x1.06)
pgpro_tam (parquet, local + cache) (c6a.metal):	9.01 GiB (x1.10)
pgpro_tam (parquet, local storage) (16 vCPU 32GB):	9.03 GiB (x1.11)
pgpro_tam (c6a.4xlarge):	9.03 GiB (x1.11)
pgpro_tam (parquet, local + cache) (16 vCPU 32GB):	9.03 GiB (x1.11)
pgpro_tam (parquet, local storage) (c6a.metal):	9.03 GiB (x1.11)
ClickHouse ☁ (gcp) (3×16GiB):	9.26 GiB (x1.13)
ClickHouse ☁ (aws) (12GiB):	9.26 GiB (x1.13)
ClickHouse ☁ (azure) (2×12GiB):	9.26 GiB (x1.13)
ClickHouse ☁ (aws) (3×12GiB):	9.26 GiB (x1.13)
AlloyDB (16 vCPU 128GB):	9.26 GiB (x1.13)
AlloyDB (8 vCPU 64GB):	9.26 GiB (x1.13)
ClickHouse ☁ (azure) (3×16GiB):	9.26 GiB (x1.13)
ClickHouse ☁ (gcp) (12GiB):	9.26 GiB (x1.13)
ClickHouse ☁ (aws) (3×16GiB):	9.26 GiB (x1.13)
ClickHouse ☁ (azure) (12GiB):	9.26 GiB (x1.13)
ClickHouse ☁ (azure) (2×16GiB):	9.26 GiB (x1.13)
ClickHouse ☁ (aws) (2×12GiB):	9.26 GiB (x1.13)
ClickHouse ☁ (gcp) (2×12GiB):	9.26 GiB (x1.13)
ClickHouse ☁ (aws) (8GiB):	9.26 GiB (x1.14)
ClickHouse ☁ (gcp) (2×16GiB):	9.26 GiB (x1.14)

pgpro_tam. TPC-H. Native formats.

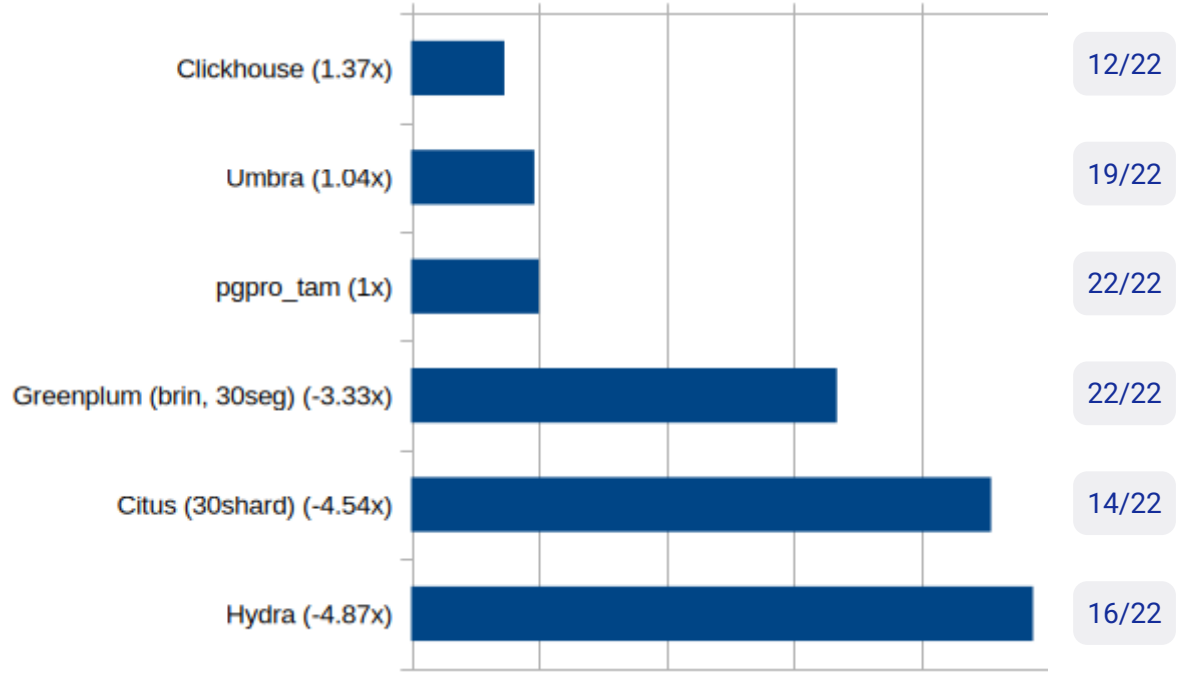
PostgresPro

Scale = 5000. 5.8TB CSV. 30 cores. 480GB RAM. Timeout = 1h.

Average



Median



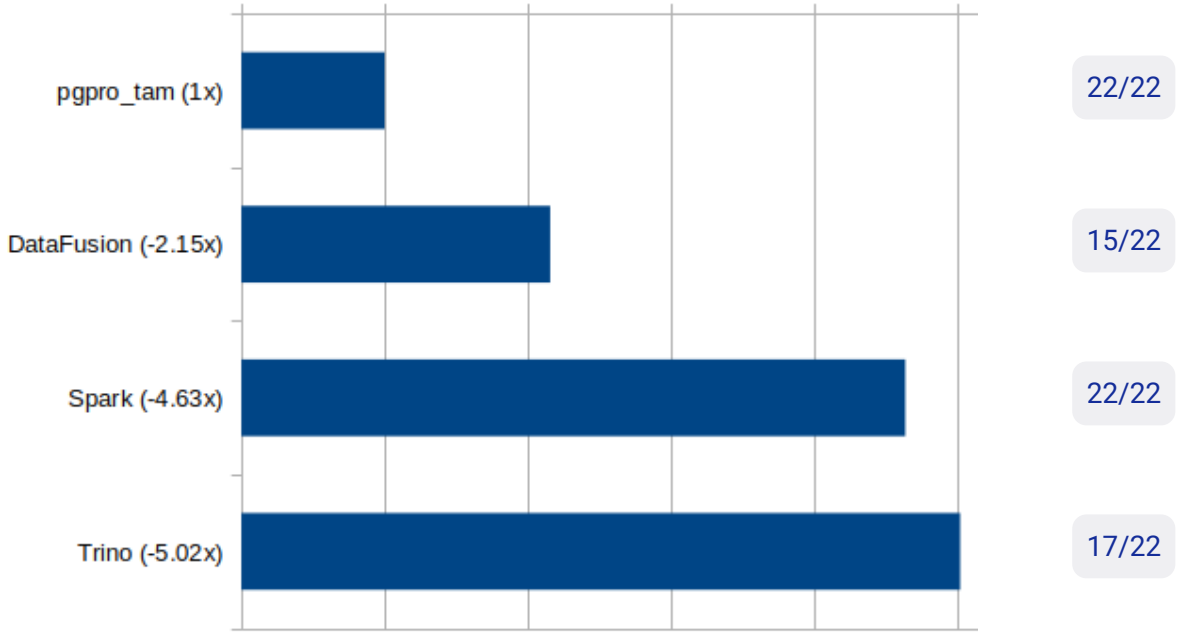
AMD EPYC 9354 30 vCores; 480GB RAM; 4x INTEL SSDPF2KX038XZ RAID10

pgpro_tam. TPC-H. Parquet.

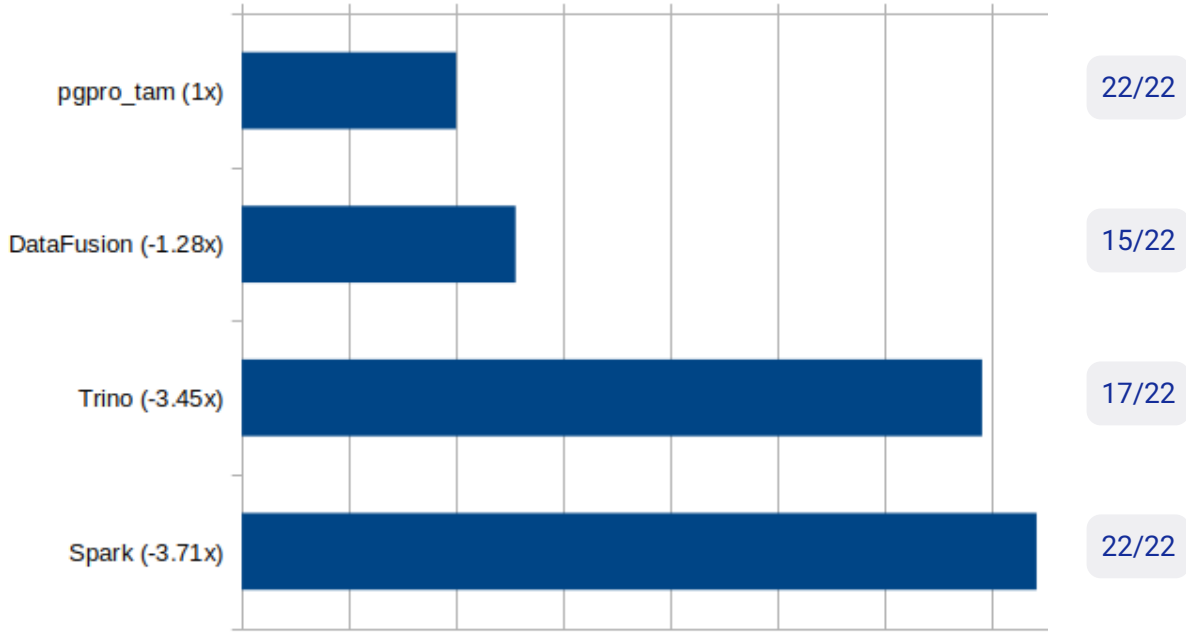
PostgresPro

Scale = 5000. 5.8TB CSV. 30 cores. 480GB RAM. Timeout = 1h.

Average



Median



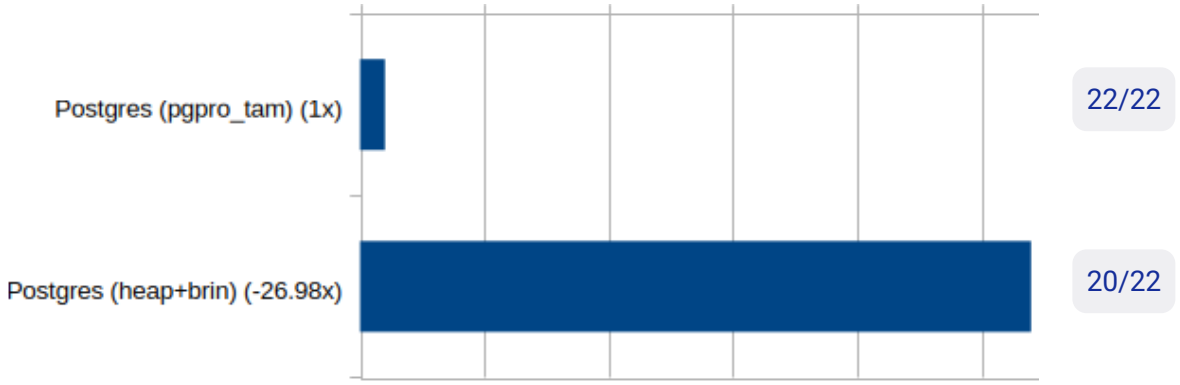
AMD EPYC 9354 30 vCores; 480GB RAM; 4x INTEL SSDPF2KX038XZ RAID10

pgpro_tam. TPC-H. Heap.

PostgresPro

Scale = 500. 540GB CSV. 30 cores. 480GB RAM. Timeout = 1h.

Average



Median

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

Duck Tales 2

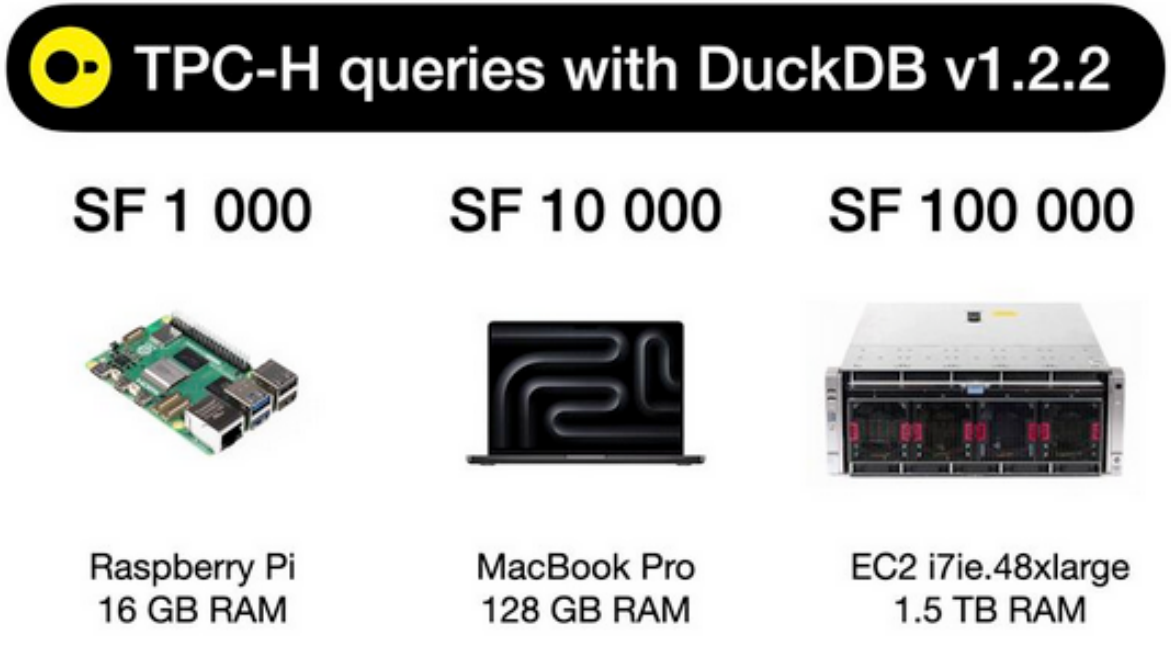
PostgresPro

DuckDB
42,443 followers
3mo •

✓ Following ···

📈 How large can DuckDB scale with its out-of-core capabilities?

We have been testing DuckDB with the TPC-H queries on increasingly large scale factors and found that all 22 queries completed on the scale factor 100 000 data set – equivalent to 100 TB in CSV format – on a single AWS EC2 instance. The queries were executed without any manual query optimization.



Итоги

PostgresPro

Аналитика в Postgres?

Формат хранения данных
Колонки, PAX

Вычитывание меньшего количества данных
Пробросы, поздняя материализация

Запись в WAL различных форматов

Встраивание стороннего исполнителя

Расширяемость Postgres
API, Hook, Callback, CustomScan

Аналитика в Postgres!

Без модификаций ядра

Не ломая user experience

Используя ту же инфраструктуру

На уровне других аналитических решений

Single instance

Итоги

PostgresPro

Аналитика в Postgres?

- Формат хранения данных
  Колонки, PAX
- Вычитывание меньшего количества данных
  Пробросы, поздняя материализация
- Запись в WAL различных форматов
- Встраивание стороннего исполнителя 
- Расширяемость Postgres
  API, Hook, Callback, CustomScan

Аналитика в Postgres!

- Без модификаций ядра
- Не ломая user experience
- Используя ту же инфраструктуру
- На уровне других аналитических решений
- Single instance

PostgresPro

Q&A

Алексей Гордеев
a.gordeev@postgrespro.ru
@innerlife0

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