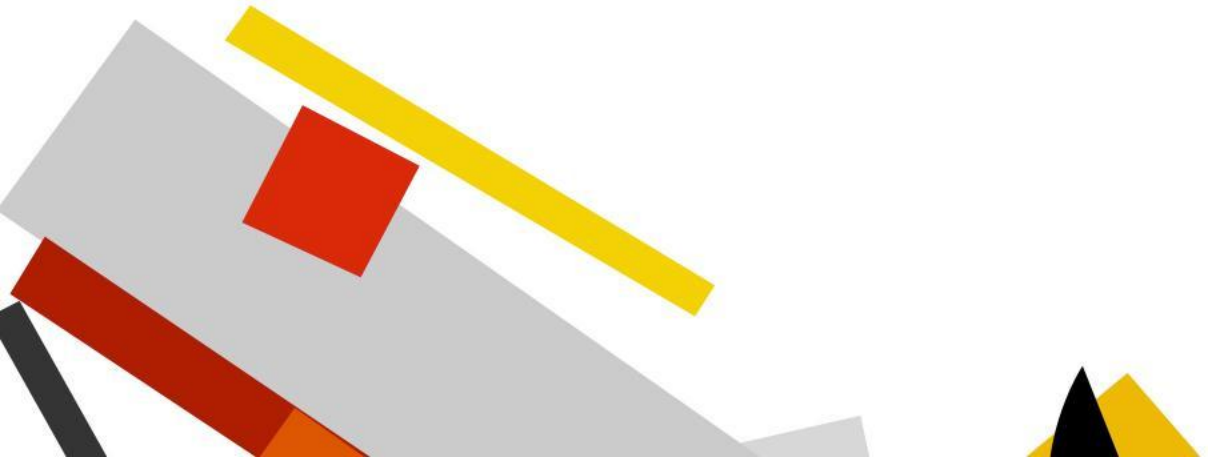
Долгожданный релиз pg_pathman

Дмитрий Иванов, Александр Коротков



Профессиональная конференция разработчиков высоконагруженных систем

Так зачем секционировать?

- Управление большими объемами данных
- Быстрые запросы к наиболее используемым секциям (локальность данных)
- Хранение старых данных на медленных носителях или отдельных серверах (FDW)
- Pagination без OFFSET + LIMIT
- Ротация данных при помощи секций

Когда нужно секционировать?

- Таблица содержит архивные данные, в последнюю секцию добавляются новые данные
- Содержимое таблицы должно быть распределено между дисками или серверами (шардинг)
- Хочется ускорить запросы к определенным срезам данных

Старый добрый метод

```
CREATE TABLE partitioned (val INT);

CREATE TABLE partitioned_1 (LIKE partitioned INCLUDING ALL)
INHERITS (partitioned);

ALTER TABLE partitioned_1 ADD CHECK (val >= 1 AND val < 100);
```

Минусы

- Много ручной работы (управление секциями)
- Полный перебор секций при планировании
- Отсутствие оптимизаций во время исполнения
- Нет встроенной поддержки HASH секционирования
- Не копируются foreign keys родителя
- “Интересные” проблемы с ACL (привилегии)

Решение

- Выбрать какое-нибудь расширение для автоматизации рутины (не решает проблему с планированием)
- Попробовать написать свое :)

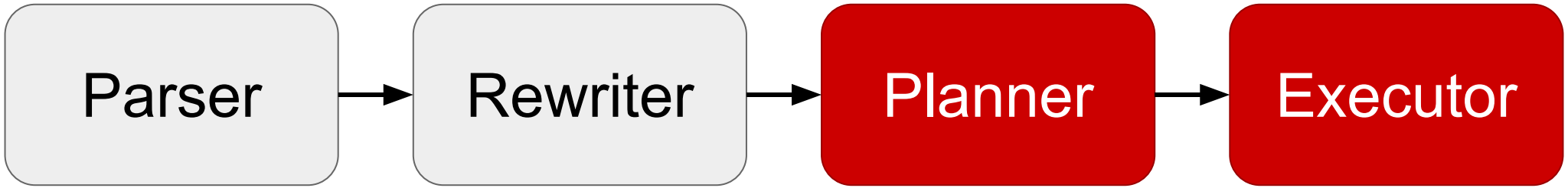
pg_pathman - это:

- Поддержка HASH и RANGE секционирования
- Автоматическое + ручное управление секциями
- Улучшенное планирование запросов
- **`RuntimeAppend`** - выбор секции во время исполнения
- **`PartitionFilter`** - INSERT без триггеров
- Перехват оператора **`COPY FROM/TO`**
- Неблокирующее конкурентное секционирование
- Поддержка FDW

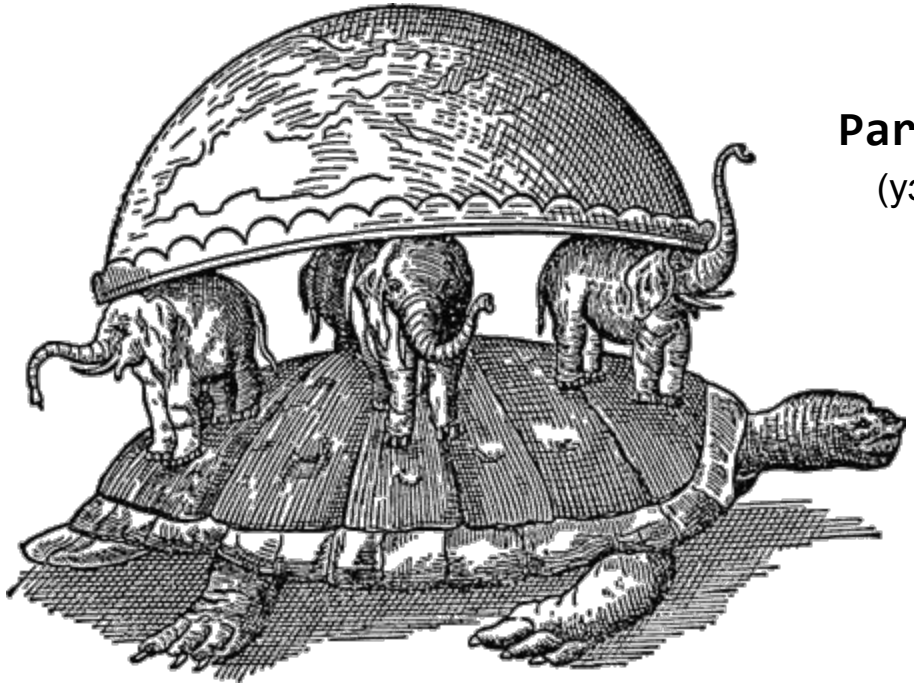
Основные элементы API

- Создание секций (add, attach, append, prepend)
- Управление созданными секциями (merge, split, drop)
- Генерация check constraints и триггеров для UPDATE
- Установка обработчиков создания секций
- Представление (view) с информацией о секциях
- Представление (view) с перечнем задач конкурентного секционирования
- Таблица для хранения опциональных настроек

Этапы выполнения запроса



RuntimeAppend
(узел исполнения)



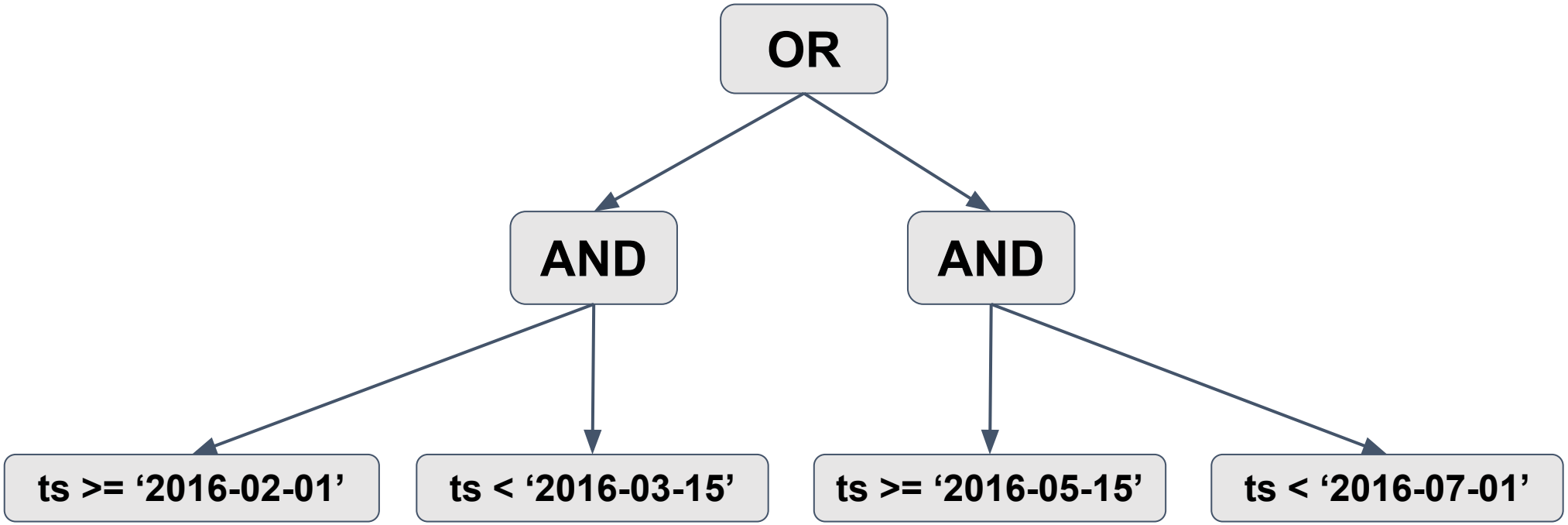
PartitionFilter
(узел исполнения)

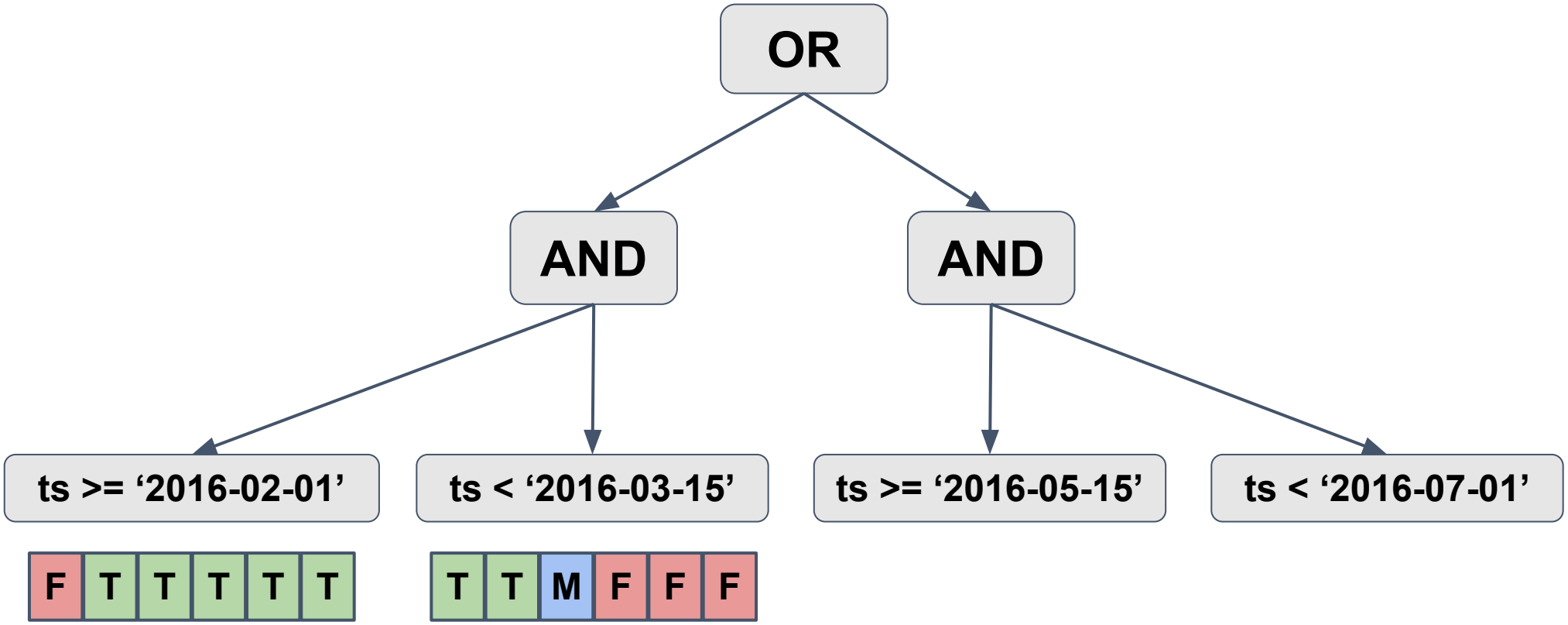
условия WHERE
(исключение секций)

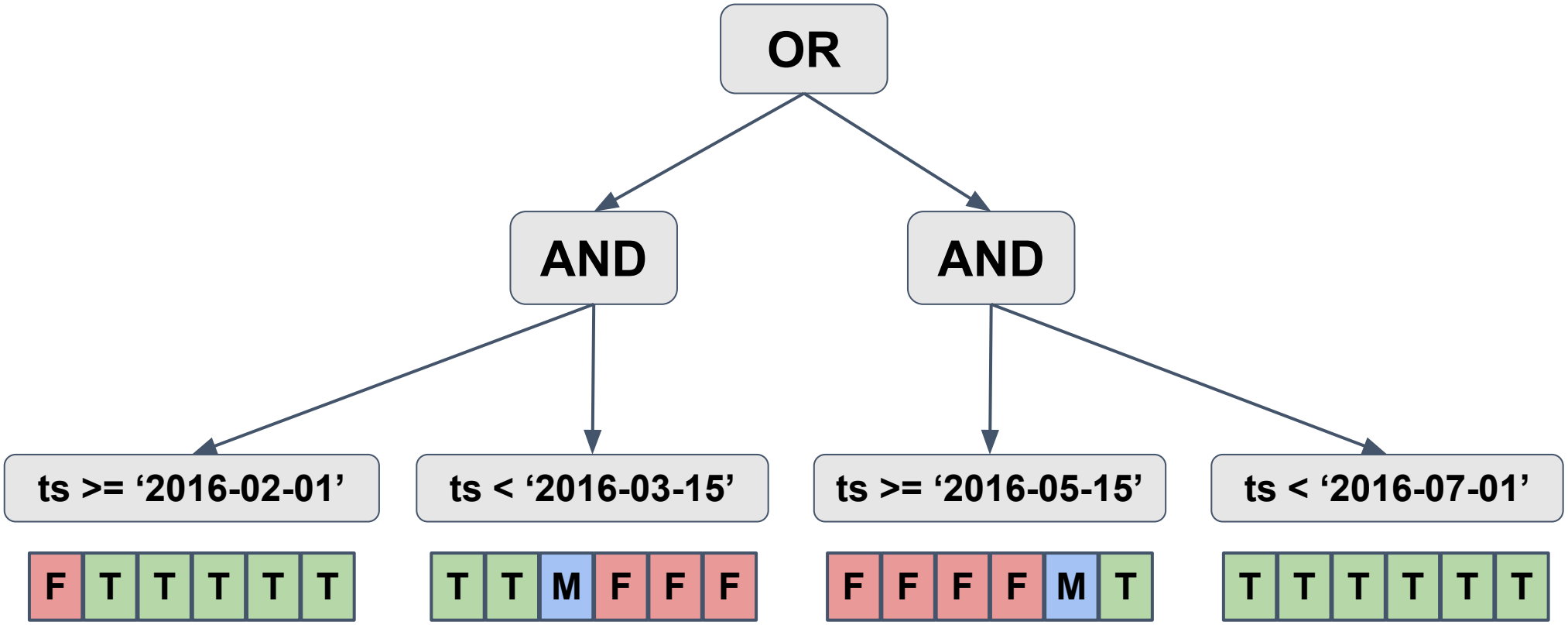
Обработка условий (`WHERE`)

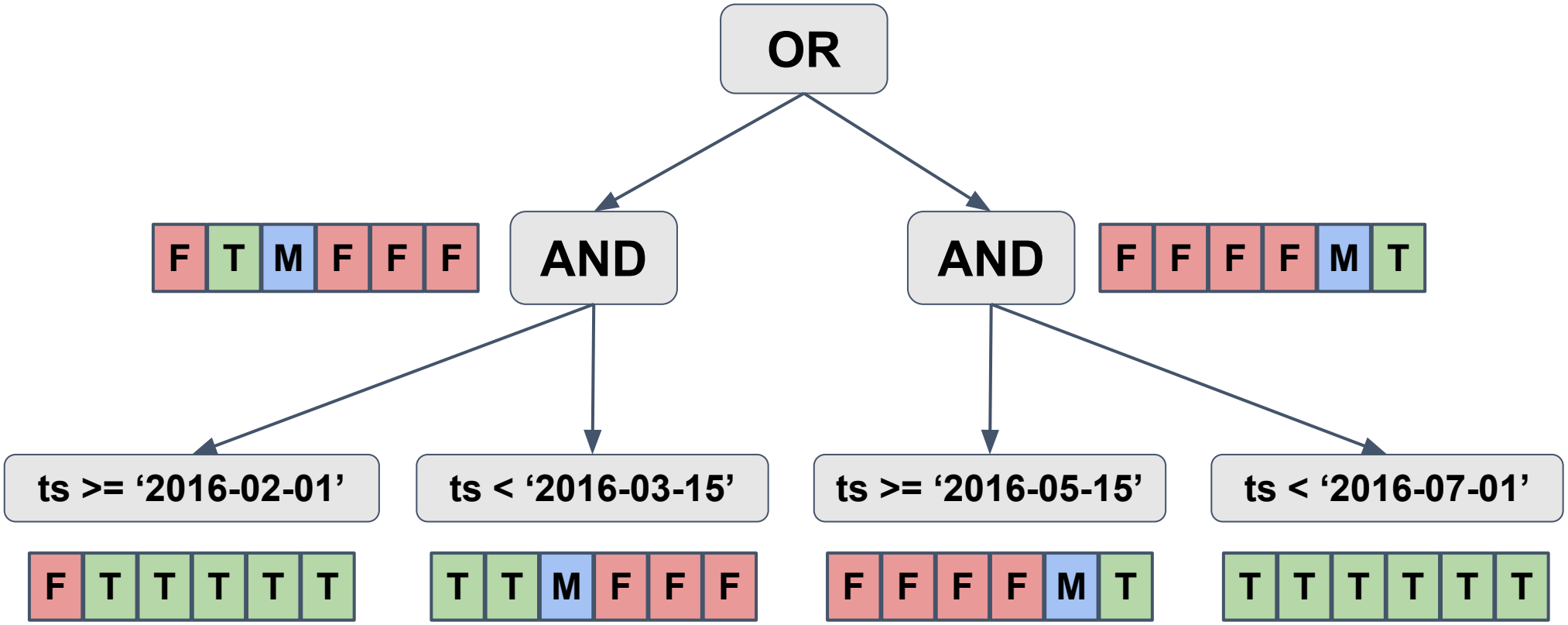
1. Механизм constraint exclusion в PostgreSQL не упрощает условия **`WHERE`**, которые попадают в секции. Они передаются “как есть”.
2. pg_pathman упрощает условия **`WHERE`**, которые попадают в каждую конкретную секцию.
3. Рассмотрим, как pg_pathman справляется с этим, на примере. Пусть данные разбиты на **6** секций по колонке **`ts`**. Каждая секция – один месяц начиная с 01.2016.

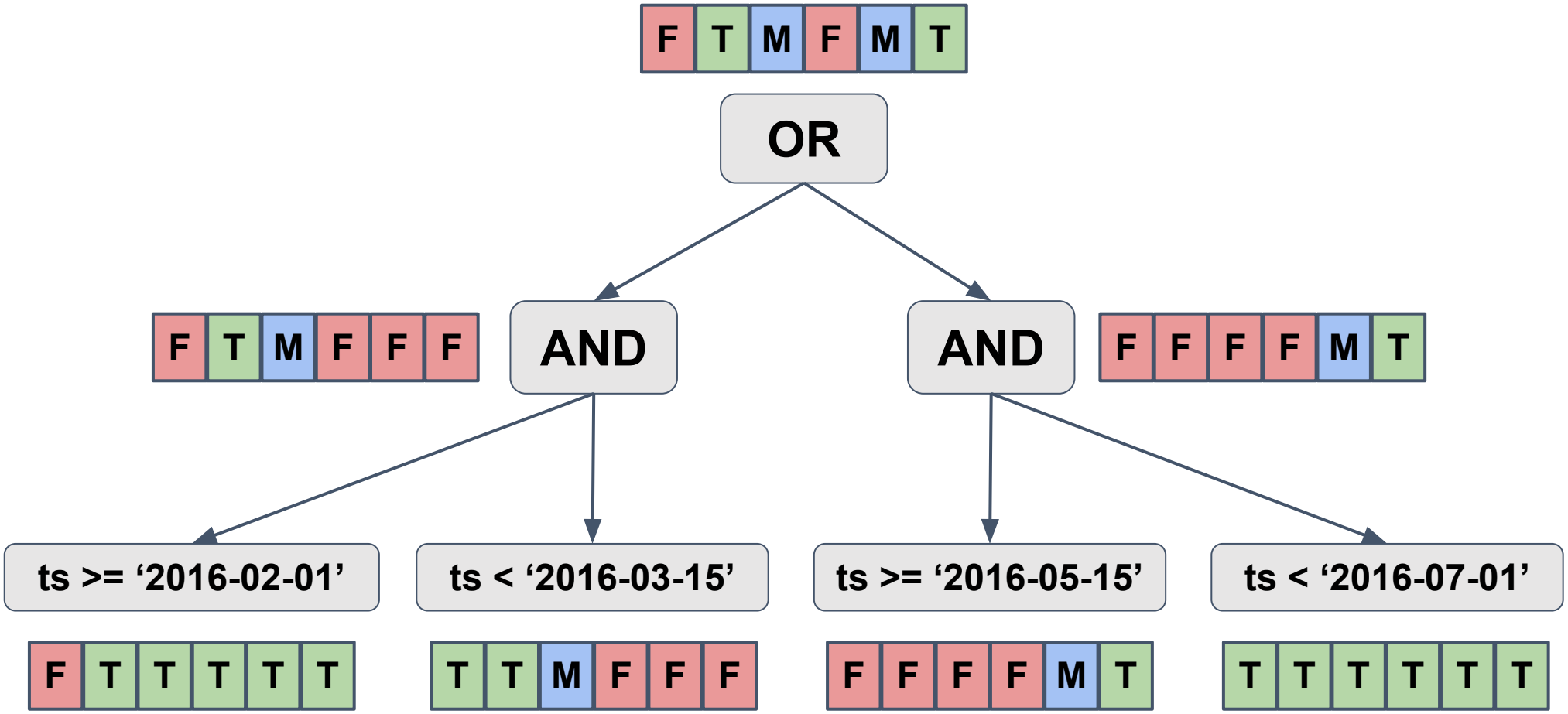
```
SELECT * FROM test WHERE (ts >= '2015-02-01' AND ts < '2015-03-15')
                      OR (ts >= '2015-05-15' AND ts < '2015-07-01');
```



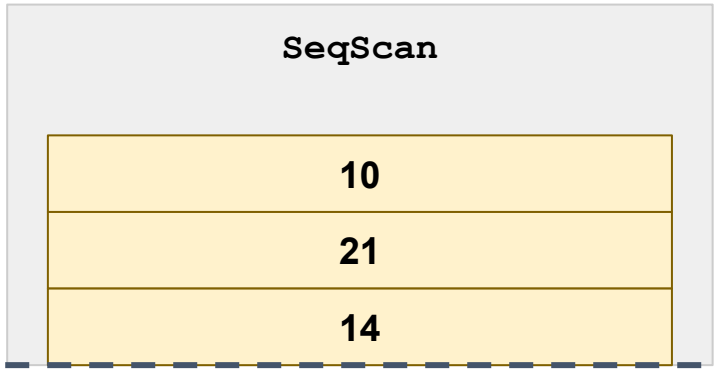


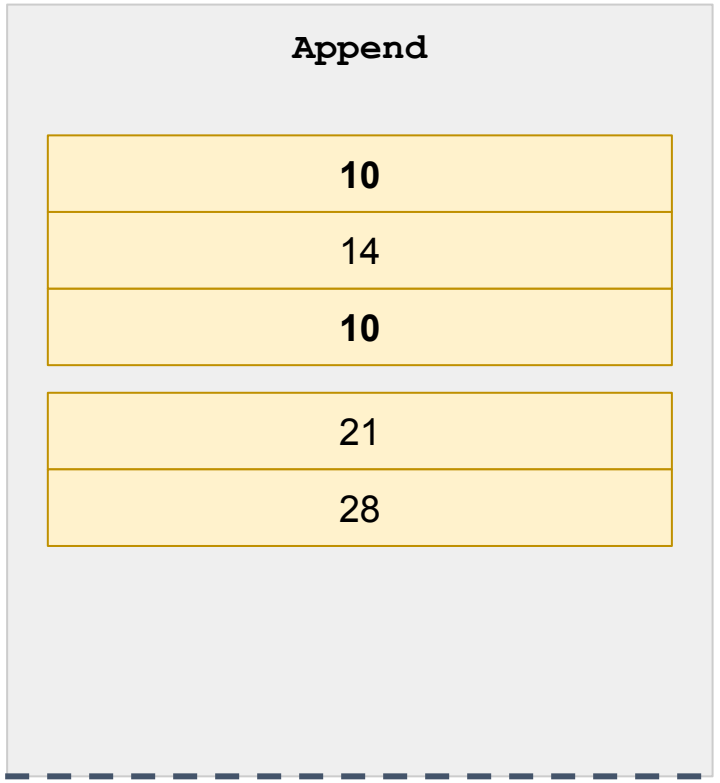


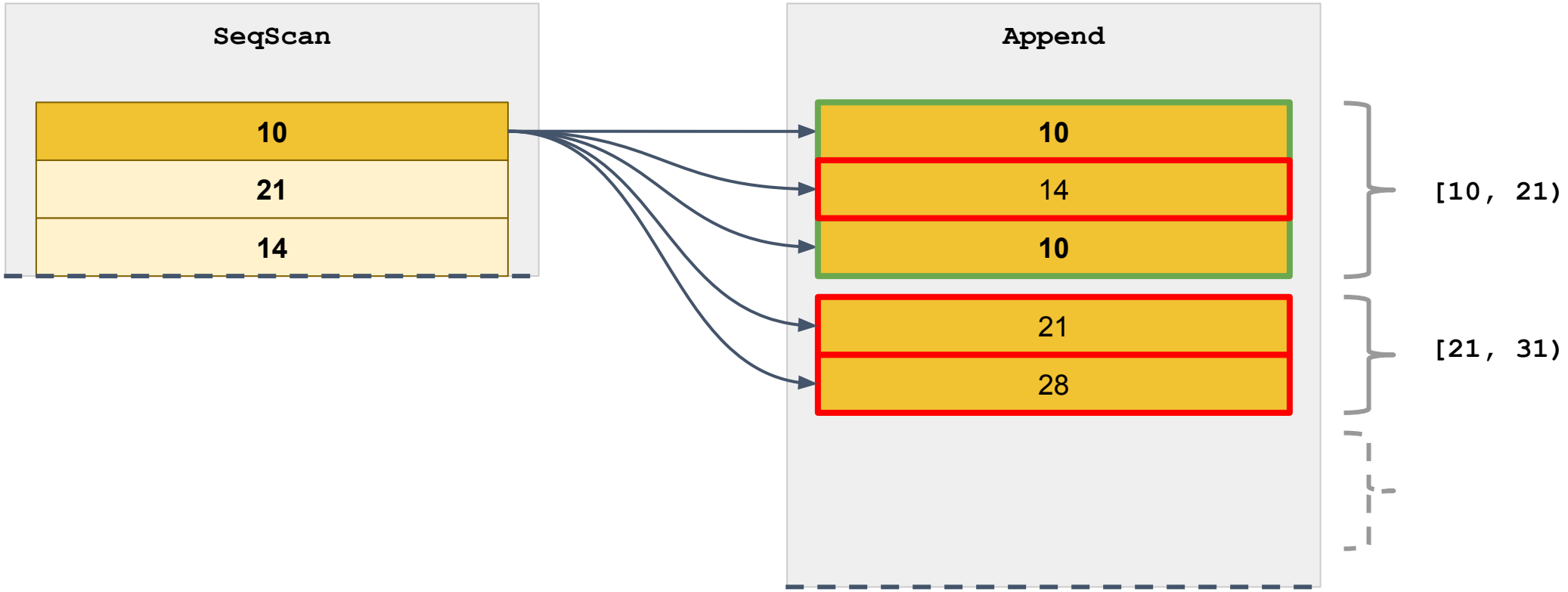


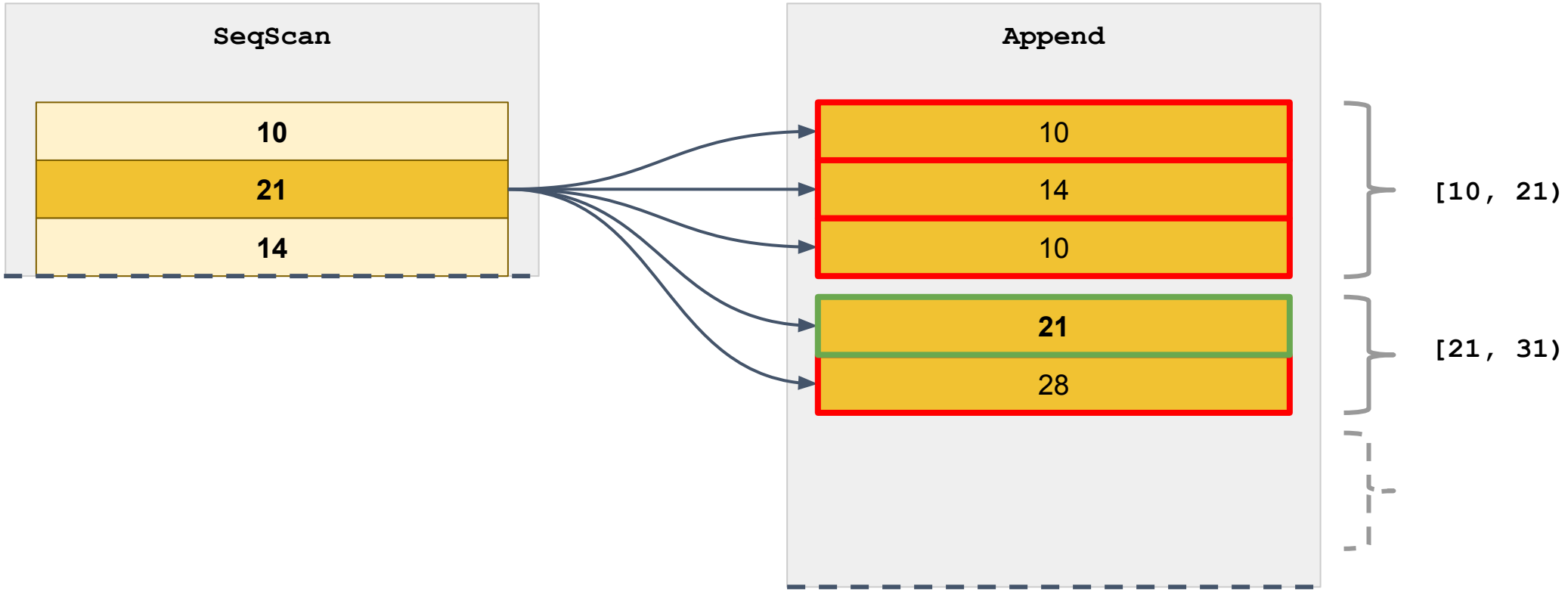


```
EXPLAIN SELECT * FROM test WHERE (ts >= '2015-02-01' AND ts < '2015-03-15')
                              OR (ts >= '2015-05-15' AND ts < '2015-07-01');
                                 QUERY PLAN
-------------------------------------------------------------------------------
 Append  (cost=0.00..3248.59 rows=0 width=0)
   ->  Seq Scan on test_2  (cost=0.00..780.20 rows=0 width=0)
   ->  Index Scan using test_3_ts_idx on test_3  (cost=0.29..767.99 rows=0 width=0)
         Index Cond: (ts < '2015-03-15 00:00:00'::timestamp without time zone)
   ->  Seq Scan on test_5  (cost=0.00..864.40 rows=0 width=0)
         Filter: (ts >= '2015-05-15 00:00:00'::timestamp without time zone)
   ->  Seq Scan on test_6  (cost=0.00..836.00 rows=0 width=0)
(7 rows)
```



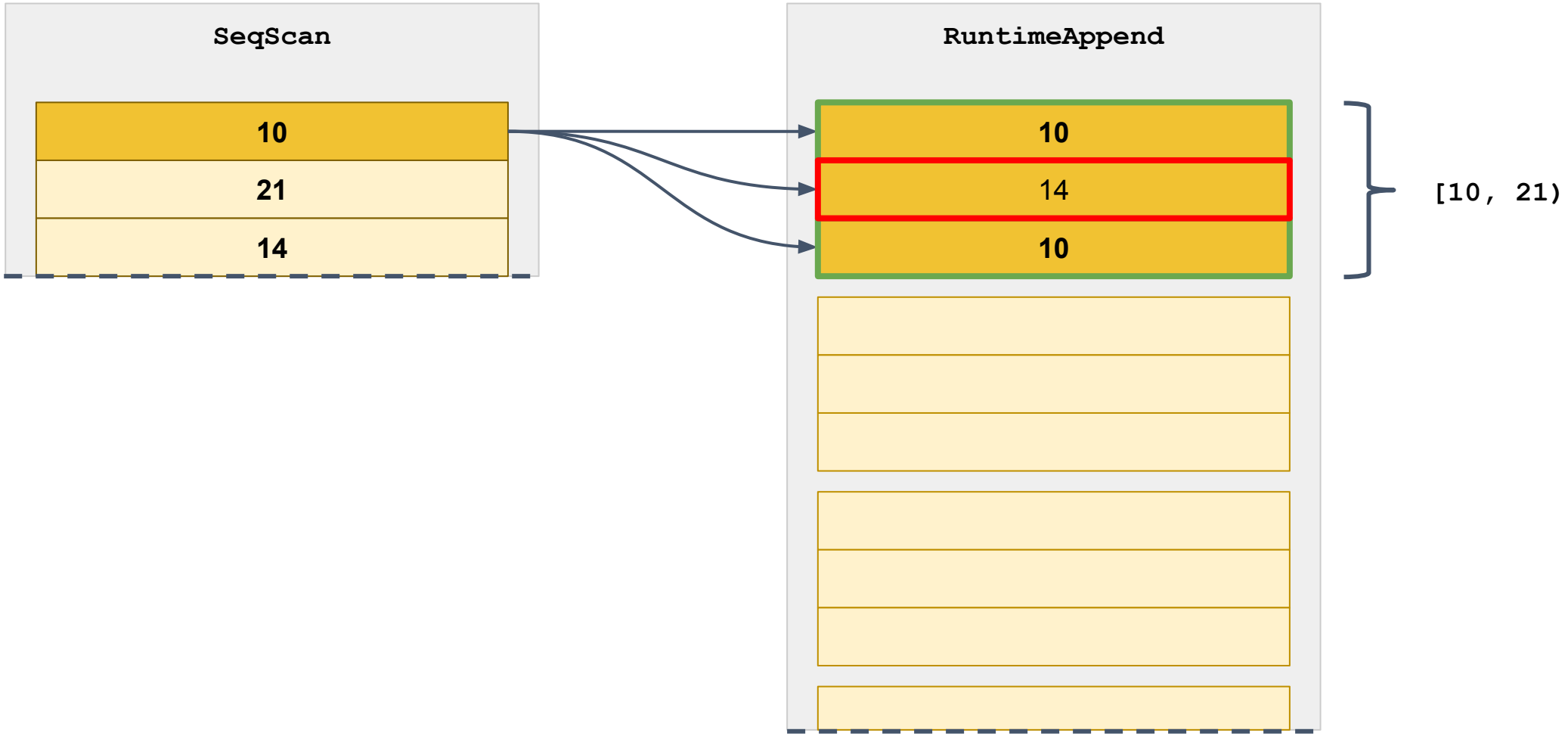


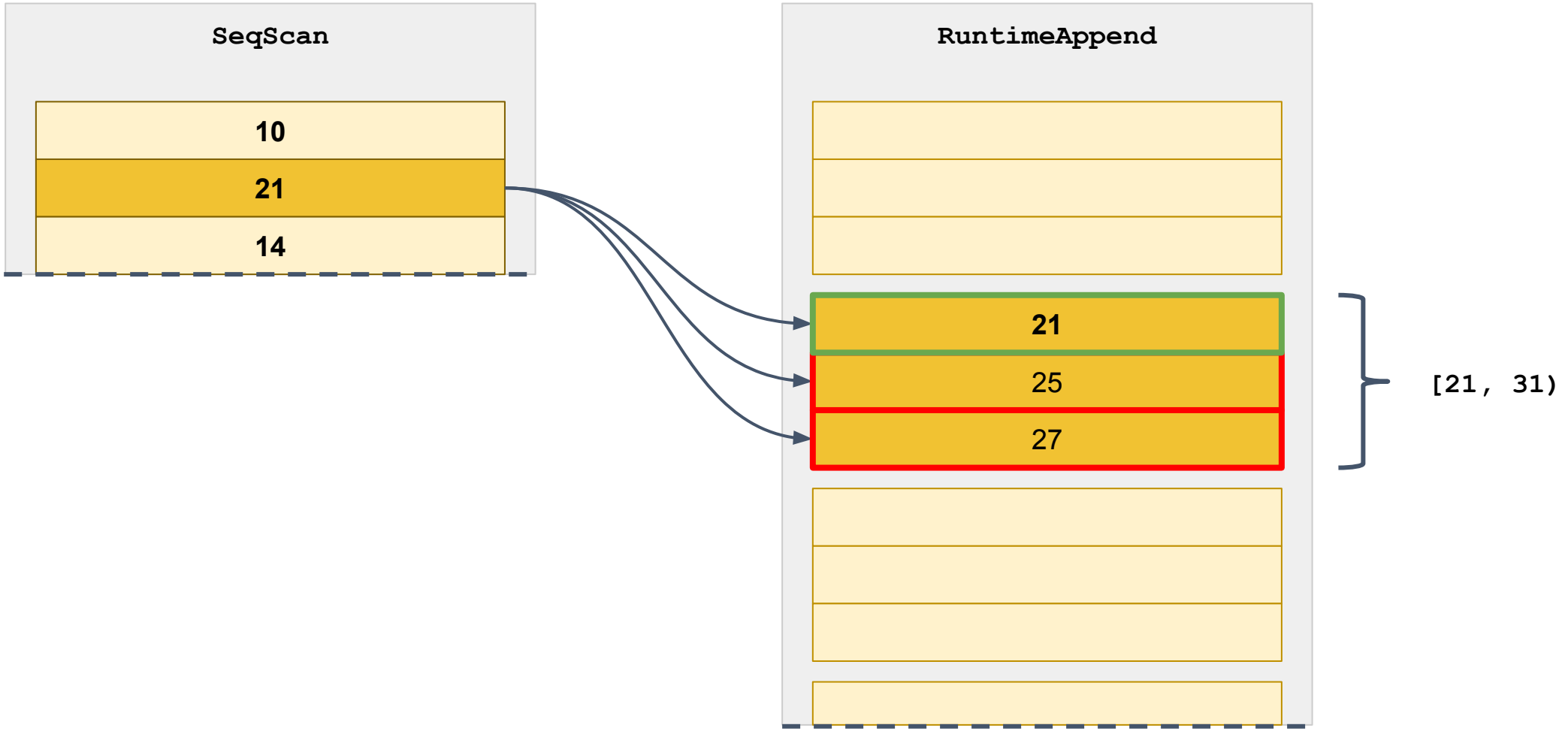


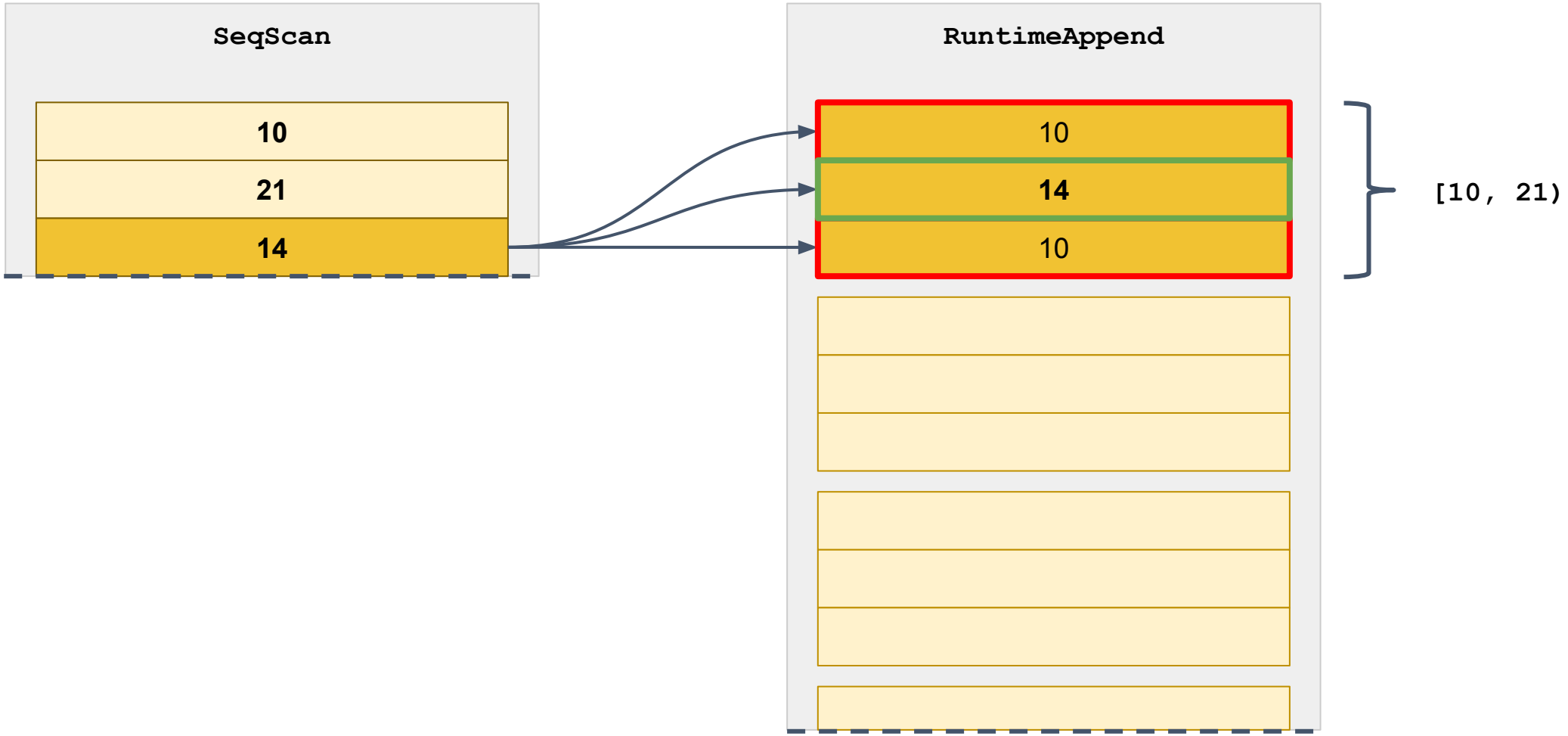


RuntimeAppend

- Выбирает только те секции, которые подходят под условия (**`WHERE`**) в данный момент времени (на этапе исполнения)
- Умеет вычислять условия с параметрами (**`$N`**)
- **Побочный эффект: `EXPLAIN`** (без **`ANALYZE`**) показывает всех детей, так как мы должны запланировать все сканы до стадии исполнения







```
SELECT * FROM partitioned_table
WHERE id = (SELECT * FROM some_table LIMIT 1);

SELECT * FROM partitioned_table
WHERE id = ANY (SELECT * FROM some_table LIMIT 4);

SELECT * FROM partitioned_table
JOIN some_table USING (id);
```

PartitionFilter

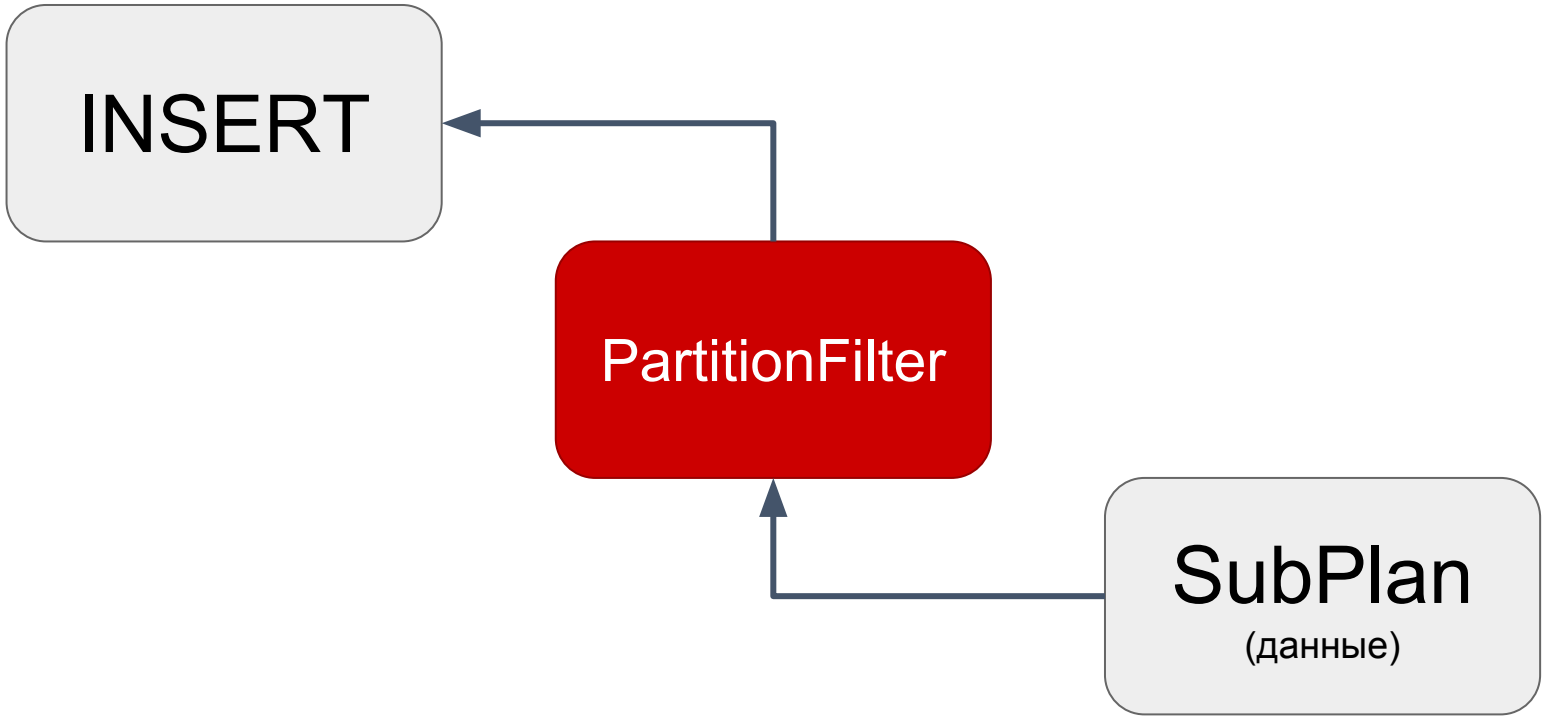
Было:

```
EXPLAIN (COSTS OFF)
INSERT INTO partitioned_table
SELECT generate_series(1, 10), random();
               QUERY PLAN
----------------------------------------
 Insert on partitioned_table
   ->  Subquery Scan on "*SELECT*"
         -> Result
(3 rows)
```

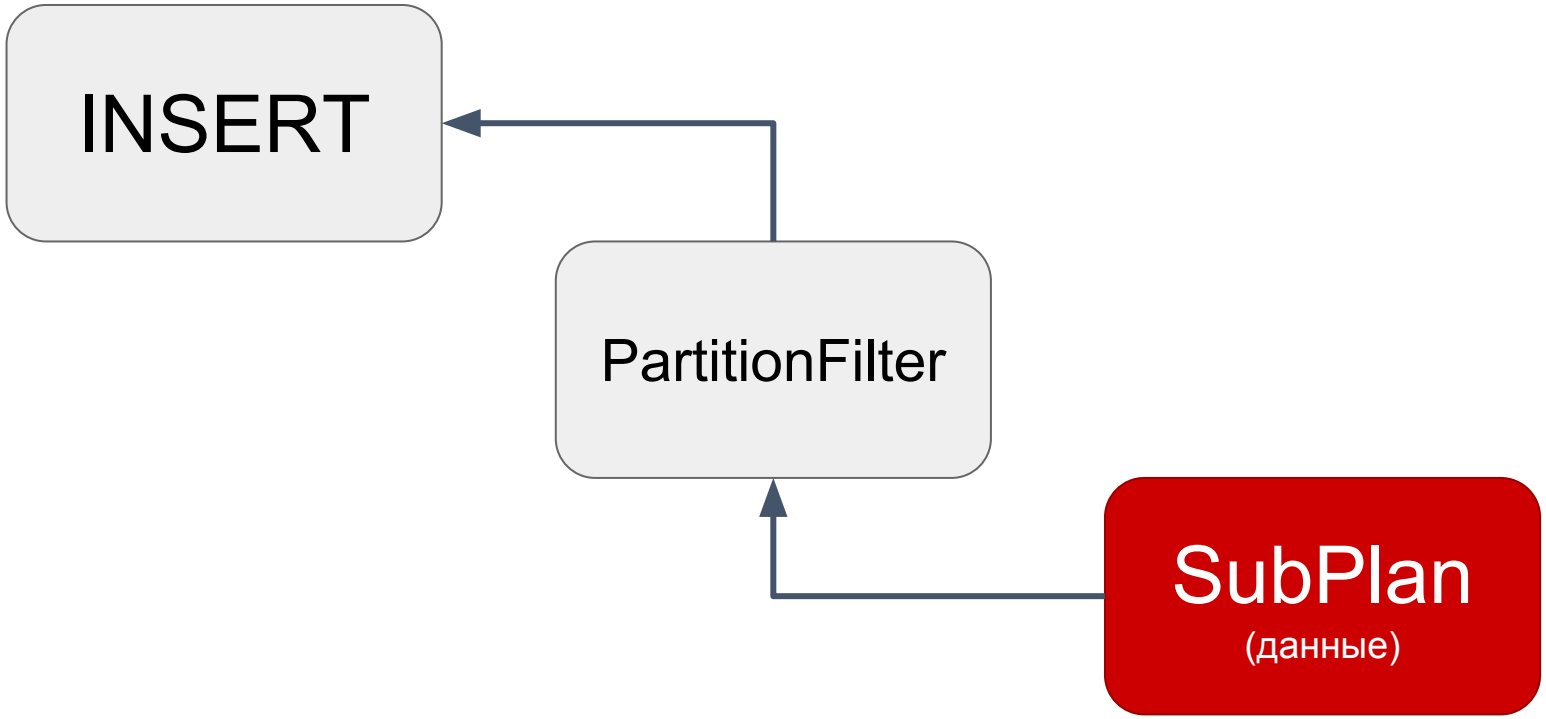
Стало:

```
EXPLAIN (COSTS OFF)
INSERT INTO partitioned_table
SELECT generate_series(1, 10), random();
               QUERY PLAN
----------------------------------------
 Insert on partitioned_table
   ->  Custom Scan (PartitionFilter)
         ->  Subquery Scan on "*SELECT*"
               -> Result
(4 rows)
```

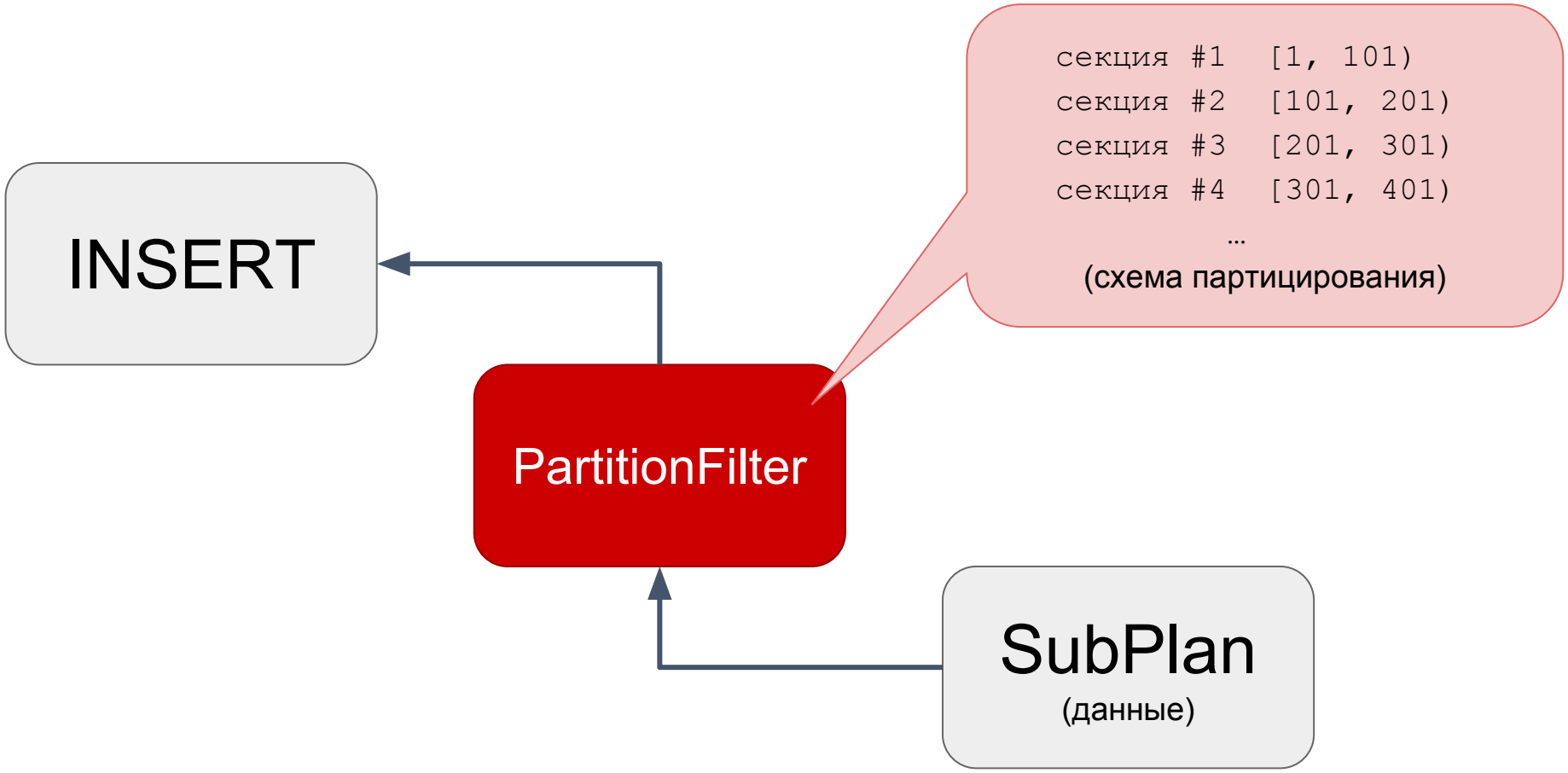
PartitionFilter



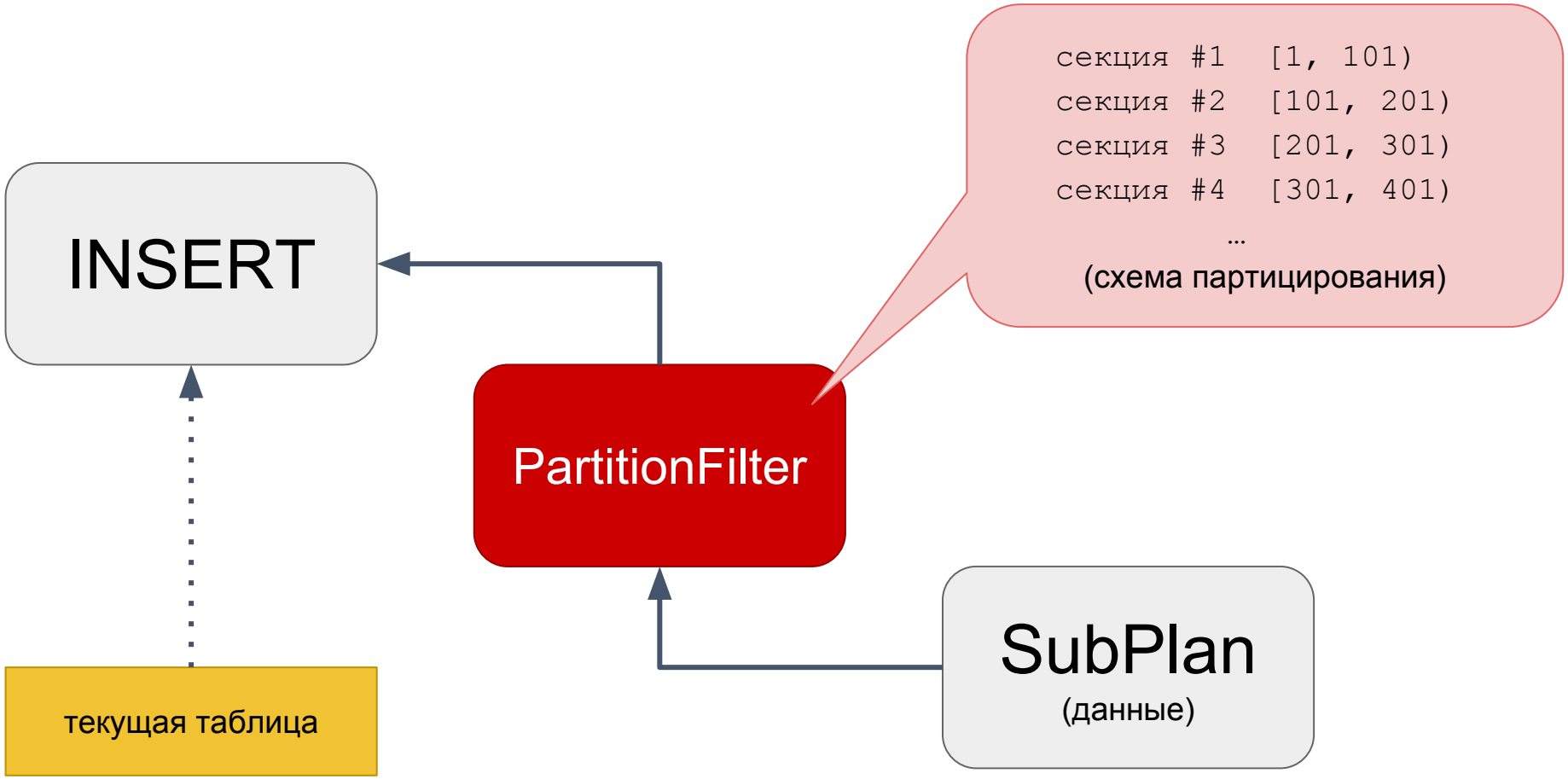
PartitionFilter



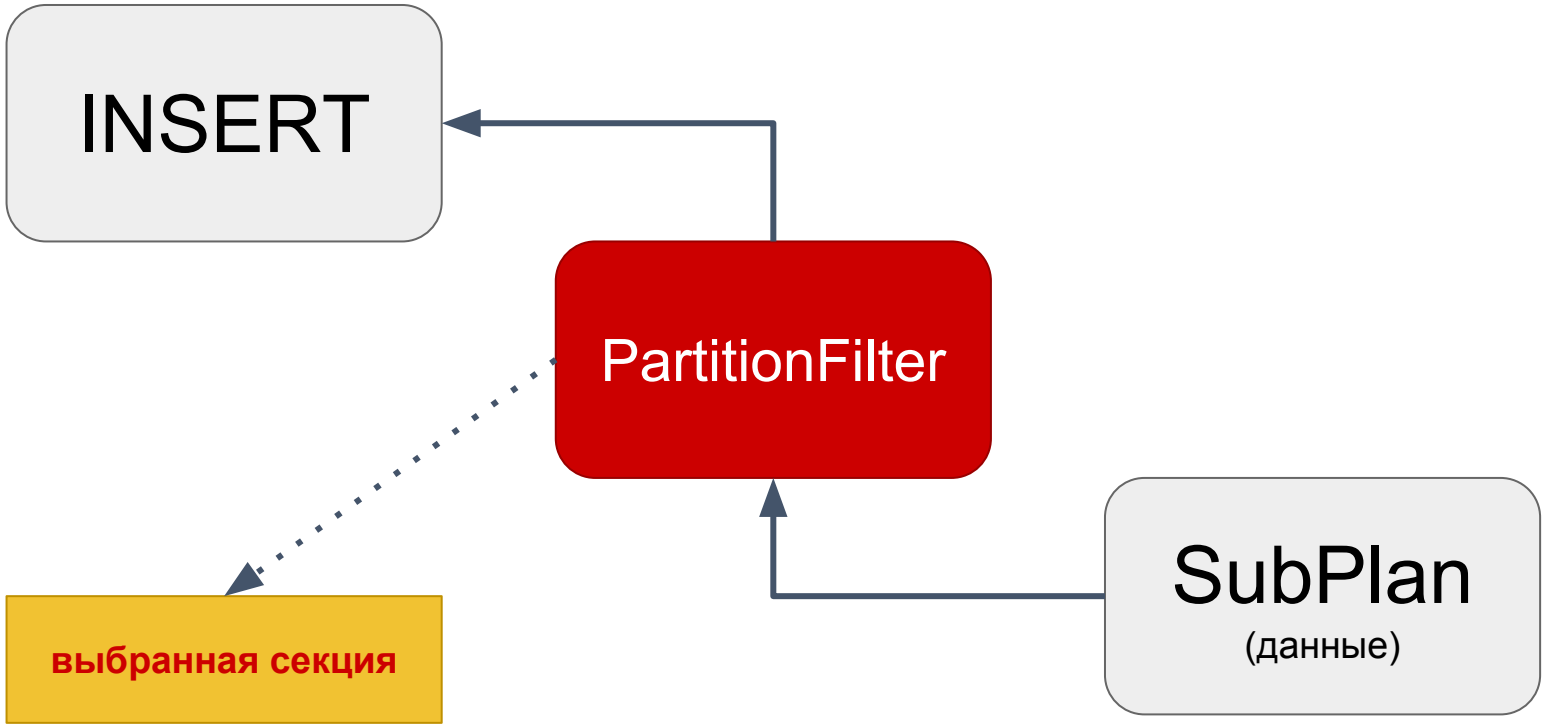
PartitionFilter



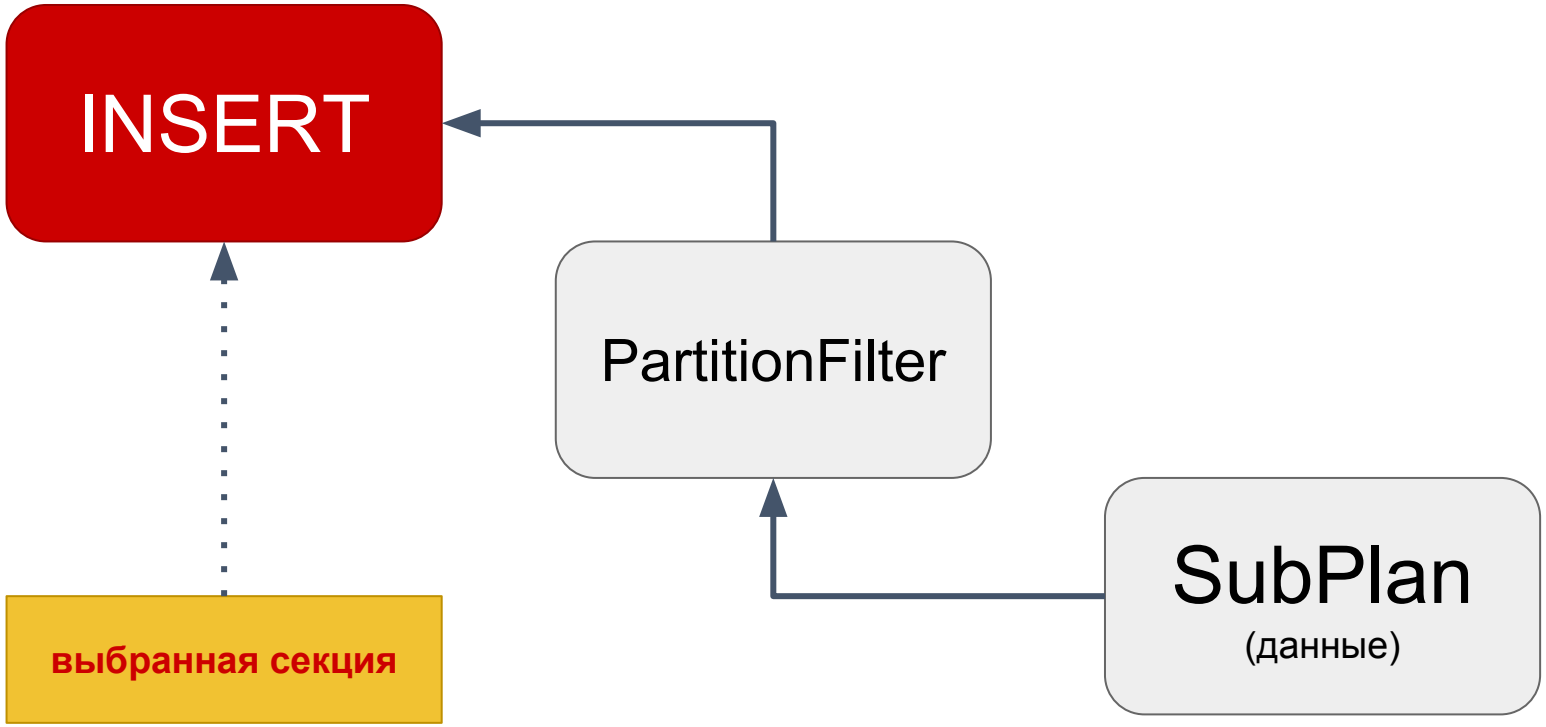
PartitionFilter



PartitionFilter



PartitionFilter



Плюсы PartitionFilter

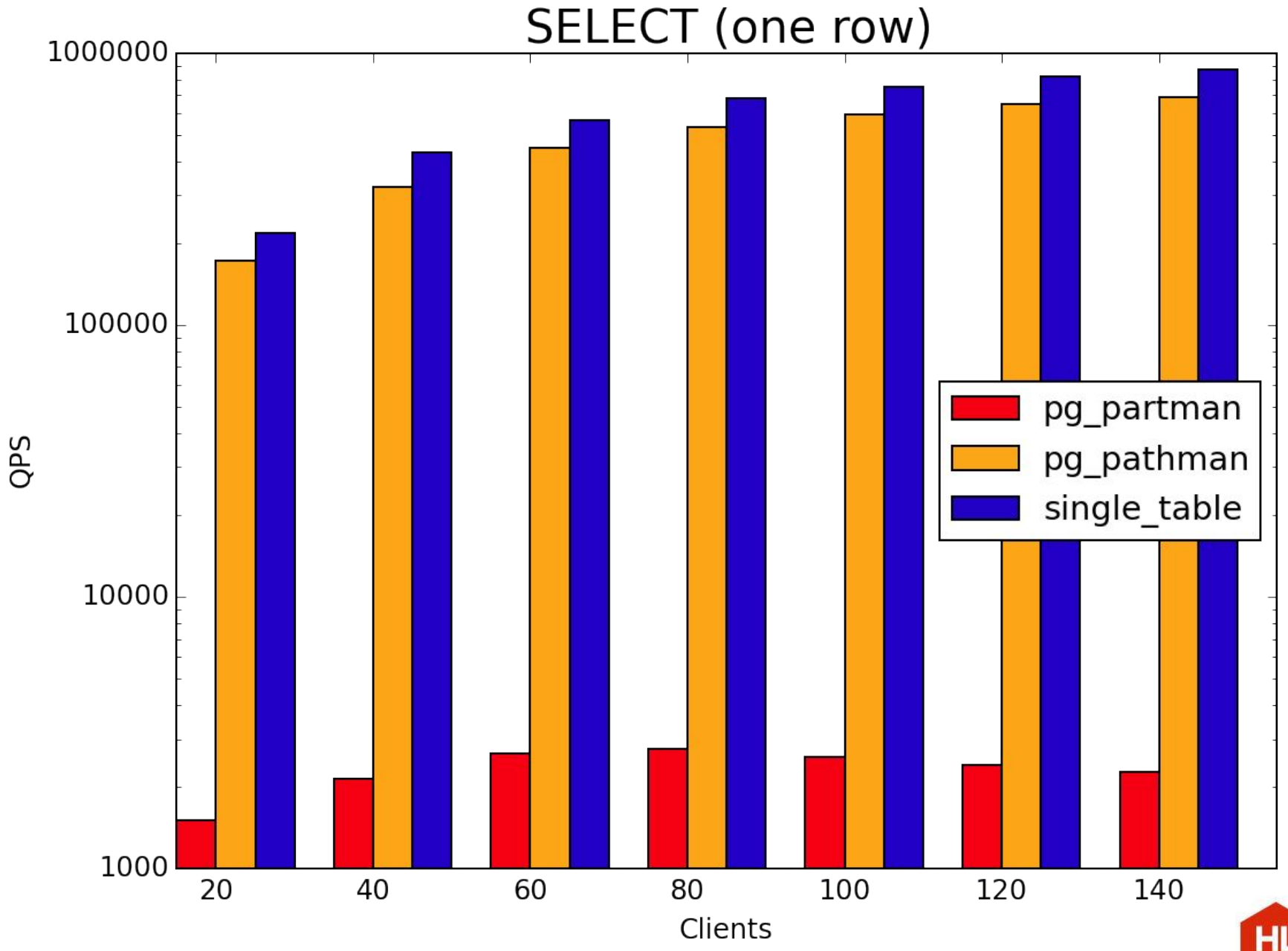
- Быстрая вставка данных (сравнимо с обычной таблицей)
- Поддержка предложения **`RETURNING *`**
- Как следствие, корректно отображается число вставленных строк
- Поддержка триггеров

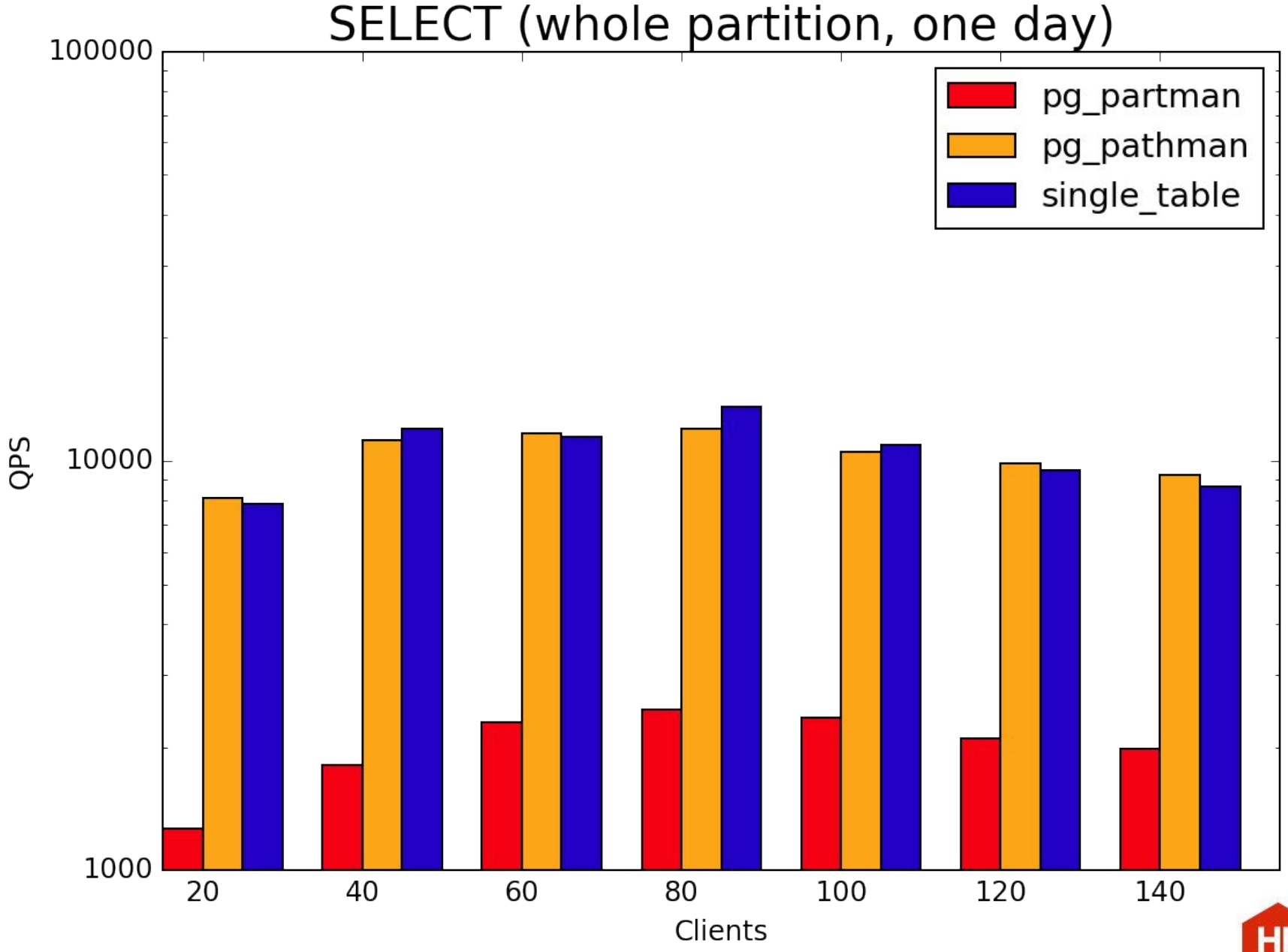
```
INSERT INTO journal (dt, level, msg)
VALUES ('2016-12-31', random(), 'test')
RETURNING *;
  id     |          dt           | level | msg
---------+-----------------------+-------+------
 1051202 | 2016-12-31 00:00:00   |     0 | test
(1 row)

INSERT 0 1
```

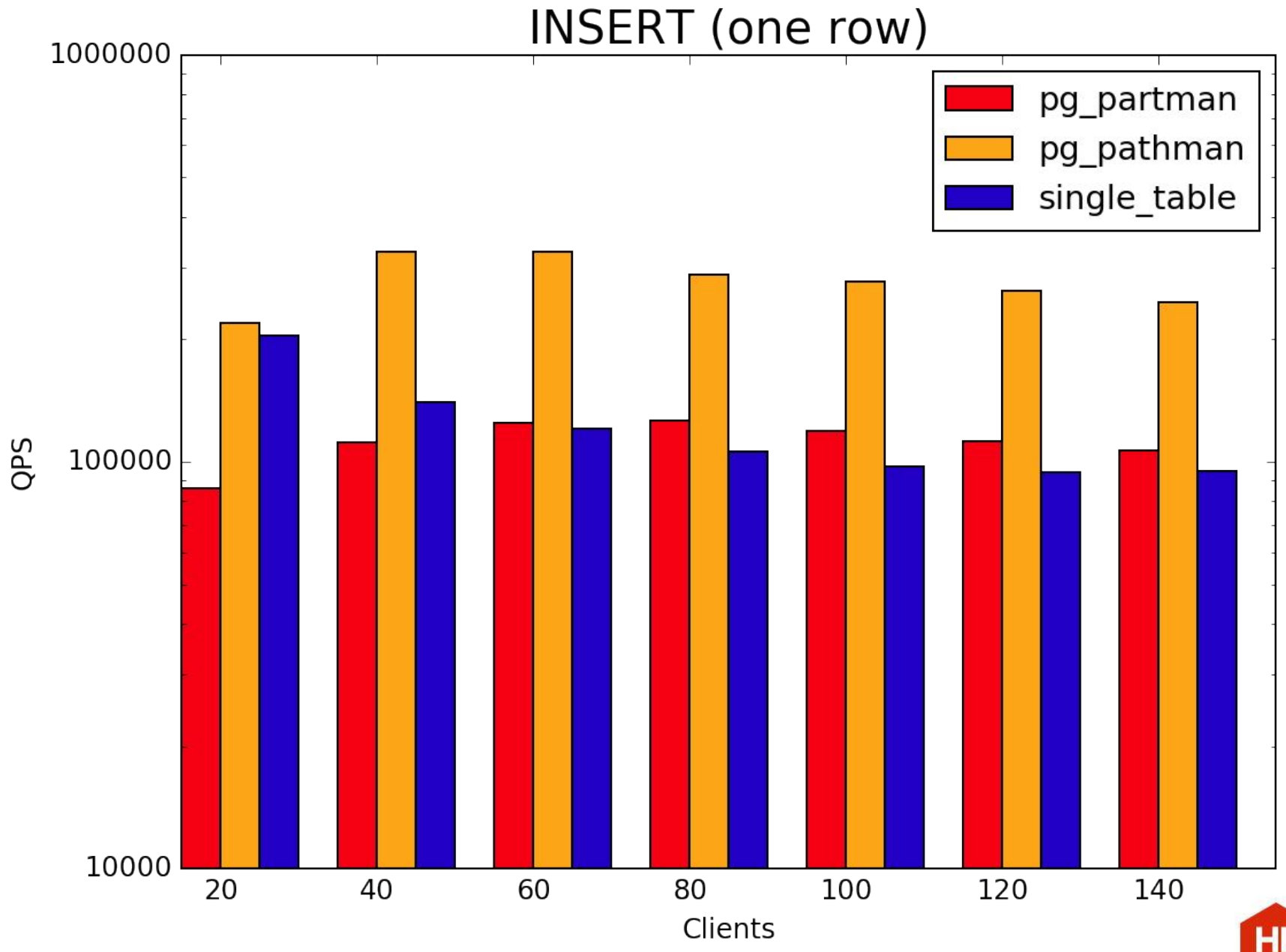
```
COPY journal TO stdout;
1051203  2016-12-31 00:00:00   1  test

COPY journal FROM '/home/dmitry/journal.sql';
PATHMAN COPY 1

SELECT * FROM ONLY journal;
 id | dt | level | msg
----+----+-------+-----
(0 rows)
```

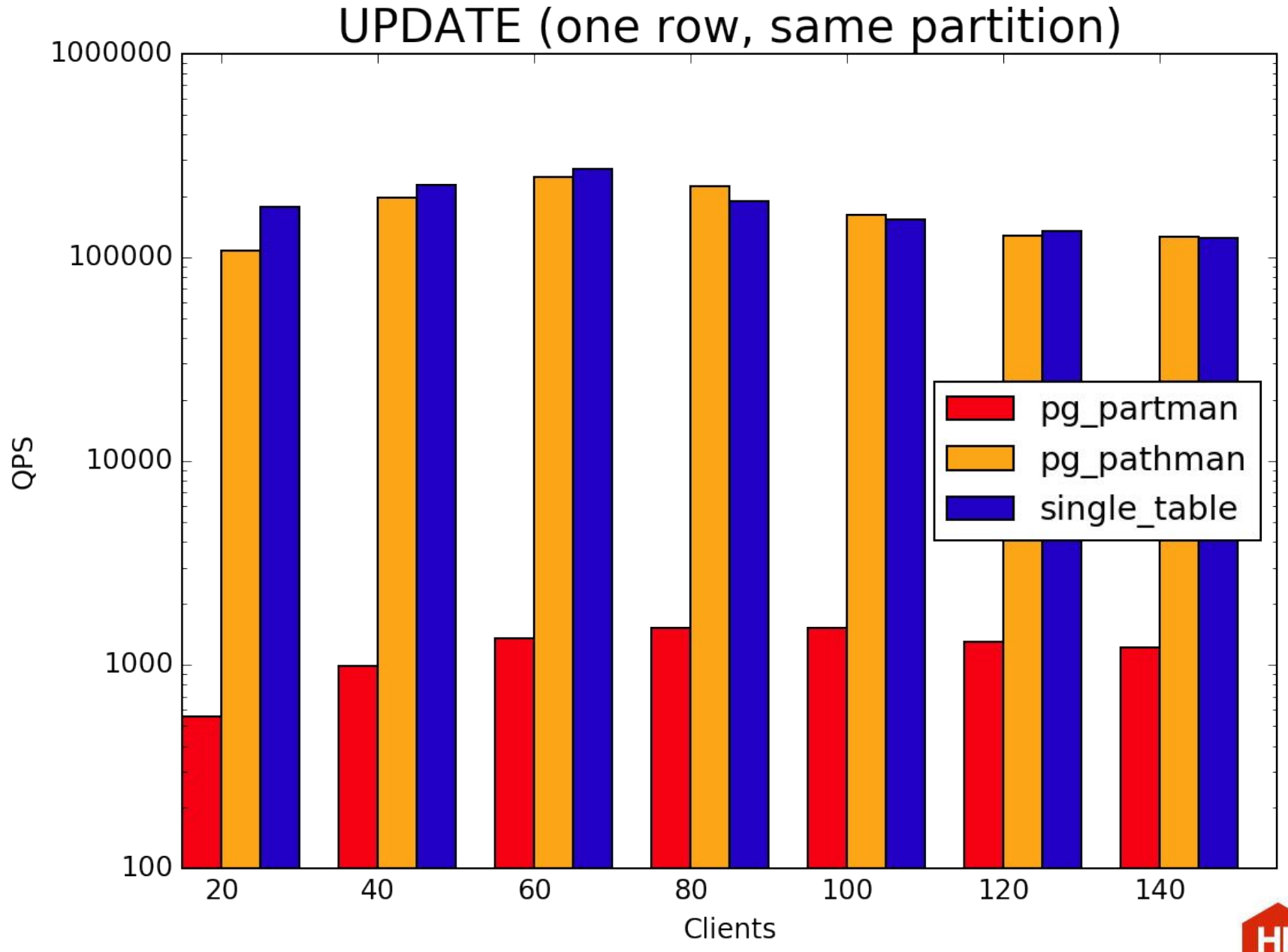
Бенчмарки

```
CREATE TABLE journal (
    id      SERIAL PRIMARY KEY,
    dt      TIMESTAMP NOT NULL,
    level   INTEGER,
    msg     TEXT);

CREATE INDEX journal_dt_idx ON journal (dt);

/* разбиваем на 366 секций, по 1 на день, затем заполняем данными: */
INSERT INTO journal (dt, level, msg)  SELECT g, random() * 6, md5(g::TEXT)
FROM generate_series('2016-01-01'::DATE, '2016-12-31'::DATE, '30 seconds') as g;
```



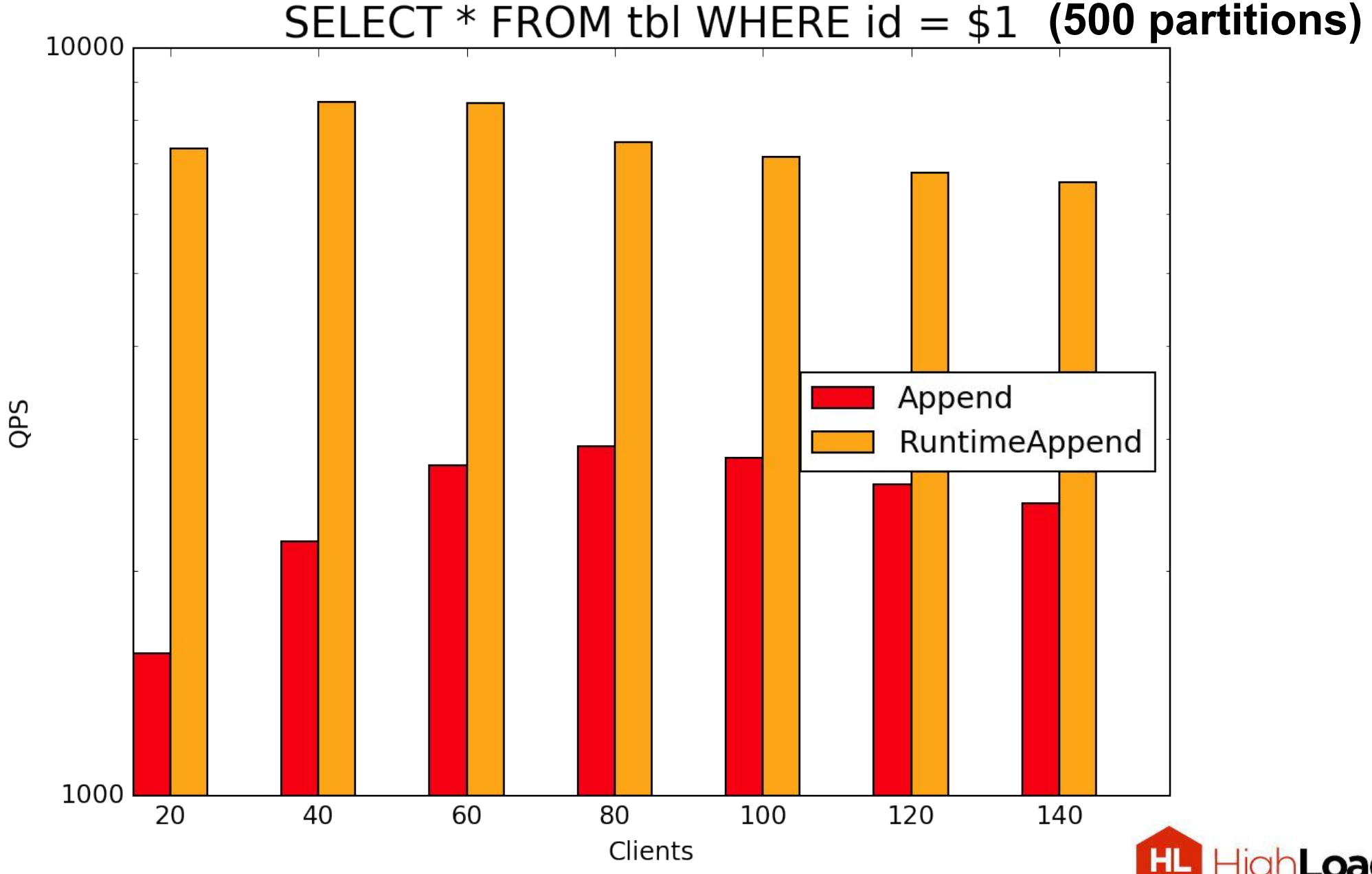

SELECT (whole partition, one day)
100000
10000
1000
QPS
20
40
60
80
100
120
140
Clients
pg_partman
pg_pathman
single_table

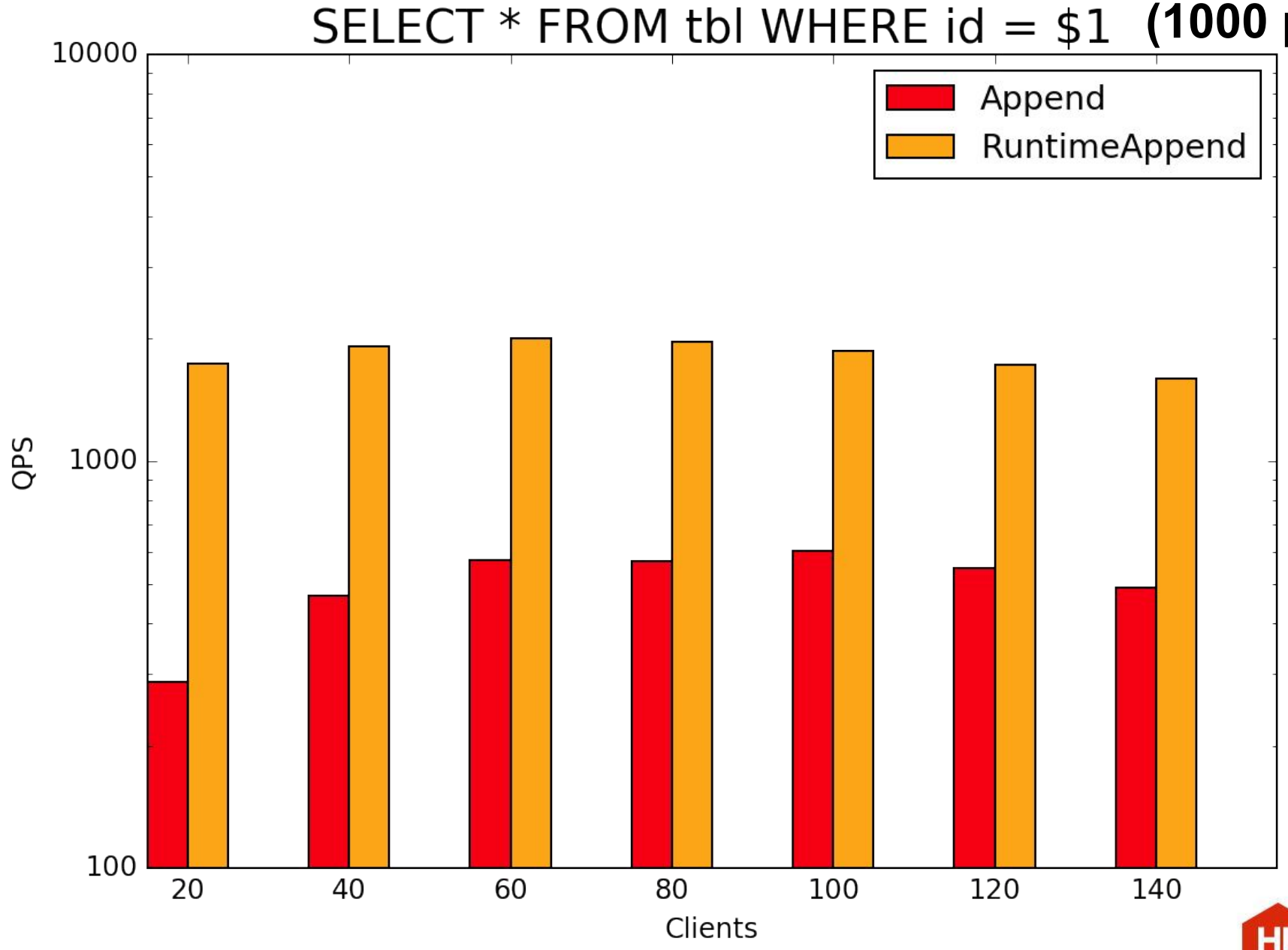




Бенчмарки - RuntimeAppend

```
CREATE TABLE rappend_test(
    id       INT NOT NULL,
    val      REAL,
    comment  TEXT);

CREATE INDEX ON rappend_test (id, comment);

/* разбиваем, заполняем данными */
SELECT create_range_partitions('rappend_test', 'id', 1, W, N);
INSERT INTO rappend_test select g, random(), g:: TEXT FROM generate_series(1, (1E8))  AS g;
```





Выводы

- pg_pathman даёт достаточно богатую функциональность. И гораздо более высокую производительность, чем все другие расширения, основанные на constraint exclusion.
- pg_pathman’ом можно пользоваться уже сейчас. Мы знаем, что то там, то тут баги, но реагируем оперативно!
- Декларативный синтаксис будет в 10 (очень надеемся!). Но всех фич pg_pathman он достигнет (самое раннее) к 11 (2018 г.).
- Как только с декларативным синтаксисом будут решены основные вопросы, мы будем портировать свои фичи туда.