

PostgreSQL 18

Оптимизация запросов



НОВАЯ ДЕБОМАЗА

Павел Толмачев

Оглавление

Введение	3
I. Демонстрационная база данных	7
1. Общая информация	7
2. Основные сущности	9
II. Планирование и выполнение	19
1. Простой протокол запросов	19
2. Расширенный протокол	26
3. Подробнее о планировании	30
III. Методы доступа	37
1. Последовательное сканирование	37
2. Сканирование индекса	43
3. Сканирование по битовой карте	50
4. Сканирование только индекса	59
5. Параллельный доступ	64
6. Сравнение эффективности методов доступа	72
IV. Сортировка и группировка	75
1. Сортировка	75
2. Группировка	83
V. Способы соединения	91
1. Что такое способы соединения	91
2. Соединение вложенными циклами	92
3. Хеш-соединение	97
4. Соединение слиянием	106

2	VI. Статистика	111
	1. Базовая статистика	111
	2. Расширенная статистика	126
	VII. Профилирование	137
	1. Что такое профилирование	137
	2. Что измерять	139
	3. Профиль приложения	140
	4. Профиль PL/pgSQL	140
	5. Профиль запросов	141
	6. Профиль одного запроса	147
	VIII. Материализация и функции	151
	1. Материализация	151
	2. Функции	161
	IX. Подходы к настройке	179
	1. Что настраивать?	179
	2. Настройка сервера	180
	3. Настройка приложения	187
	4. Запросы	191
	Заключение	205
	О компании	209

Введение

Эта книга посвящена основам оптимизации запросов PostgreSQL.

Образовательный отдел компании Postgres Professional разрабатывает учебные курсы для администраторов и прикладных разработчиков, среди которых есть и курс по оптимизации запросов. Внимательный читатель может задать вопрос — а в чем отличие книги и курса, если и там, и там речь идет об одном и том же — об оптимизации запросов?



На самом деле материал этой книги и есть переработанный курс «QPT. Оптимизация запросов». Поэтому и из книги, и из курса можно почерпнуть примерно одни и те же сведения. Но есть некоторые различия в форме изложения.

Курс QPT последовательно, связно и подробно раскрывает тему, начиная с базовых алгоритмов, которые используются при оптимизации в PostgreSQL. Курс состоит из нескольких модулей, а модули — из группы связанных тем. Каждая тема курса состоит из теоретической и практической частей. Теория — это в большинстве случаев не просто презентация, но также и демонстрация примеров на «живой» системе в виртуальной машине курса. Слушатели получают презентации с подробными комментариями к каждому слайду, результаты работы демонстрационных скриптов, решения практических заданий и дополнительные



- 4 справочные материалы. Курс можно пройти как с преподавателем, так и самостоятельно — все материалы, включая видеозаписи, доступны на нашем сайте всем желающим. Мы рекомендуем изучать курс, повторяя все приведенные демонстрации и выполняя практические задания.

Это обстоятельный и детальный подход. Но бывает так, что доставать ноутбук сложно, например, в метро или на остановке, а терять время не хочется. Чтобы материал курса всегда был под рукой, мы перенесли его в формат небольшой книги, которую удобно читать в любой обстановке, в которой можно заложить страницу закладкой и вернуться к ней без необходимости включать компьютер и запускать программы.

Но перенести курс в формат книги без изменений невозможно: ширина листа плохо подходит для длинных строк, а разделение на теоретические и демонстрационные блоки выглядит неестественно. Поэтому материал курса был серьезно переработан. В книге результаты демонстрационных запросов вплетены в повествование; некоторые из них претерпели изменения, а некоторые (обычно те, которые показывали какие-то дополнительные детали) были исключены для сокращения объема. С той же целью в книгу не попала тема «Типы индексов». Некоторые темы (например, «Методы доступа» и «Параллельный доступ») были объединены для упрощения подачи материала.

Мы рекомендуем приступить к этой книге, будучи знакомым с операционными системами семейства Unix, уверенно владея языком SQL (знакомство с PL/pgSQL не обязательно, но полезно) и изучив хотя бы один из курсов DBA1 или DEV1, потому что далее мы опускаем многие детали, связанные с конфигурированием сервера и его внутренним устройством, не относящиеся напрямую к оптимиза-

ции запросов. Эти знания необходимы и для прохождения курса QPT.

5

В книге мы показываем все примеры на демонстрационной базе данных «Авиаперевозки» от 01.09.2025. Обратите внимание, что в курсе QPT по 16-й версии PostgreSQL используется предыдущая версия демобазы (от 15.08.2017).



I. Демонстрационная база данных

1. Общая информация

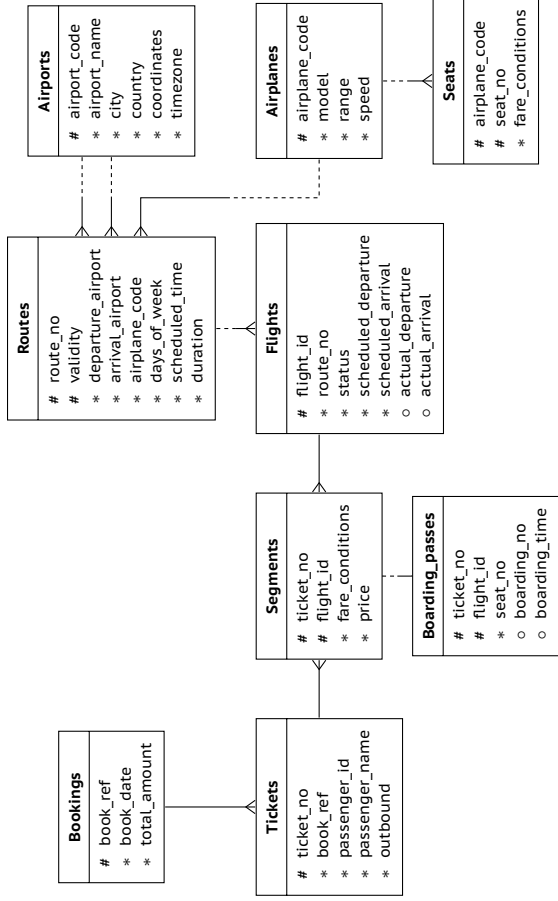
Чтобы научиться работать со сложными запросами, нам понадобится серьезная база данных — целых девять таблиц, наполненных правдоподобными сведениями.

В качестве предметной области мы выбрали авиаперевозки. Наша база данных содержит информацию о рейсах, совершенных воображаемой авиакомпанией за некоторый промежуток времени.



Схема нашей базы приведена на рисунке ниже. Мы старались сделать схему данных не слишком сложной, но и не слишком простой, чтобы обойтись без лишних деталей и в то же время отработать на ее примере осмысленные и интересные запросы.

На сайте база данных доступна в четырех вариантах, отличающихся размером. Например, в этой книге по оптимизации запросов используется база от 01.09.2025 объемом примерно 2,7 Гбайт, содержащая данные по полетам за шесть месяцев.



Проверить версию можно с помощью функции:

9
i

```
SELECT bookings.version();
               version
-----
PostgresPro 2025-09-01 (181 days)
(1 row)
```

2. Основные сущности

Основной сущностью является **бронирование** (bookings).

```
SELECT * FROM bookings b WHERE b.book_ref = '03RUKL';
 book_ref |          book_date          | total_amount
-----+-----+-----
 03RUKL   | 2026-01-04 08:01:43.367932+03 |      84850.00
(1 row)
```

Мы видим дату бронирования и общую сумму. Для нас эта дата с каждым днем уходит в прошлое:

```
SELECT now() - b.book_date
FROM bookings b
WHERE b.book_ref = '03RUKL';
      ?column?
-----
213 days 11:38:13.679144
(1 row)
```

Но для демобазы «текущий» момент зафиксирован:

```
SELECT bookings.now();
      now
-----
2026-03-01 03:00:00+03
(1 row)
```

- 10 Так что с точки зрения приложения дата бронирования от-
i стоит не так далеко:

```
SELECT bookings.now() - b.book_date
FROM bookings b
WHERE b.book_ref = '03RUKL';
      ?column?
```

```
-----
55 days 18:58:16.632068
(1 row)
```

В одно бронирование можно включить несколько пассажиров, каждому из которых выписывается отдельный **билет** (tickets) для путешествия в одну сторону и, если необходимо, еще один для возвращения.

Посмотрим, какие билеты включены в выбранное бронирование:

```
SELECT t.ticket_no, t.passenger_id, t.passenger_name
FROM bookings b
JOIN tickets t ON t.book_ref = b.book_ref
WHERE b.book_ref = '03RUKL';
```

ticket_no	passenger_id	passenger_name
0005435998770	RU 3761090132061	Evgeniy Kondratyev
0005435998767	RU 3761090132061	Evgeniy Kondratyev
0005435998776	RU 5595520101762	Oleg Golubev
0005435998774	RU 5595520101762	Oleg Golubev

(4 rows)

Билет включает в себя один **перелет** (segments), если между пунктами отправления и назначения есть прямой рейс, и несколько перелетов, если требуются пересадки.

Какие перелеты совершают наши пассажиры? Достаточно проверить одного, поскольку у всех пассажиров в одном бронировании пути следования совпадают:

```
SELECT s.*  
FROM tickets t  
JOIN segments s ON s.ticket_no = t.ticket_no  
WHERE t.ticket_no = '0005435998767';
```

ticket_no	flight_id	fare_conditions	price
0005435998767	17386	Economy	6750.00
0005435998767	17606	Economy	14500.00

(2 rows)

Это был билет в одну сторону. Сравним его с обратным:

ticket_no	flight_id	fare_conditions	price
0005435998770	26693	Comfort	3575.00
0005435998770	27013	Comfort	5525.00
0005435998770	27375	Economy	2750.00
0005435998770	27537	Economy	8250.00

(4 rows)

Как видим, пути следования не обязательно совпадают: обратно Евгений летит с тремя пересадками, причем на двух **рейсах** (flights) — в салоне класса комфорт.

Каждый рейс следует по некоторому **маршруту** (routes) из одного аэропорта в другой. Маршруты с одним номером имеют одинаковые пункты вылета и назначения, но конкретные рейсы будут отличаться датой отправления.

Чтобы разобраться, какие маршруты скрываются за выбранными рейсами, необходимо учесть темпоральную связь между этими таблицами: расписание периодически меняется, поэтому при соединении следует учитывать не только номер маршрута, но и соответствие его интервала действия времени отправления.

```

12 SELECT r.route_no, f.scheduled_departure, f.status,
   i   r.departure_airport dep, r.arrival_airport arr
FROM tickets t
  JOIN segments s ON s.ticket_no = t.ticket_no
  JOIN flights f ON f.flight_id = s.flight_id
  JOIN routes r ON r.route_no = f.route_no
              AND r.validity @> f.scheduled_departure
WHERE t.ticket_no = '0005435998767'
ORDER BY f.scheduled_departure;

```

route_no	scheduled_departure	status	dep	arr
PG0177	2026-01-05 15:05:00+03	Arrived	SVO	AER
PG0587	2026-01-06 19:00:00+03	Arrived	AER	LHR

(2 rows)

Избежать этих сложностей позволяет **расписание** (time-table): это представление, как табло в аэропорту, показывает необходимую информацию, скрывая соединение двух таблиц:

```

SELECT tt.route_no, tt.scheduled_departure, tt.status,
       tt.departure_airport dep, tt.arrival_airport arr
FROM tickets t
  JOIN segments s ON s.ticket_no = t.ticket_no
  JOIN timetable tt ON tt.flight_id = s.flight_id
WHERE t.ticket_no = '0005435998770'
ORDER BY tt.scheduled_departure;

```

route_no	scheduled_departure	status	dep	arr
PG0272	2026-02-27 06:25:00+03	Arrived	LHR	CDG
PG0781	2026-03-01 02:50:00+03	Departed	CDG	MUC
PG0358	2026-03-03 03:50:00+03	Scheduled	MUC	VIE
PG0165	2026-03-04 02:20:00+03	Scheduled	VIE	SVO

(4 rows)

Оба рейса «туда» и один «обратно» уже совершены (Arrived), а в настоящее время пассажиры летят из CDG в MUC (Departed). На следующие рейсы регистрация пока не открыта (Scheduled).

В качестве ключа для **аэропортов** (airports) используется общепринятый трехбуквенный код. Посмотрим полную информацию об одном аэропорте: 13 i

```
SELECT * FROM airports WHERE airport_code = 'CDG' \gx
```

```
-[ RECORD 1 ]+-----  
airport_code | CDG  
airport_name | Шарль-де-Голль  
city         | Париж  
country      | Франция  
coordinates  | (2.55,49.0128)  
timezone     | Europe/Paris
```

Взглянем на интересующий нас маршрут:

```
SELECT lower(r.validity) valid_from,  
       upper(r.validity) valid_to, r.airplane_code,  
       r.days_of_week, r.scheduled_time, r.duration  
FROM routes r  
WHERE r.route_no = 'PG0781'  
      AND r.validity @> '2026-03-01 02:50:00+03'::timestamptz \gx
```

```
-[ RECORD 1 ]--+-----  
valid_from    | 2026-02-01 03:00:00+03  
valid_to      | 2026-03-01 03:00:00+03  
airplane_code | CR7  
days_of_week | {1,2,3,4,5,6,7}  
scheduled_time | 00:50:00  
duration      | 01:25:00
```

В этот период действия маршрут выполняется семь дней в неделю и обслуживается **самолетом** (airplanes) CR7:

```
SELECT * FROM airplanes WHERE airplane_code = 'CR7';  
airplane_code | model          | range | speed  
-----+-----+-----+-----  
CR7           | Бомбардье CRJ700 | 3100  | 829  
(1 row)
```

- 14 Количество **мест** (seats) и их распределение по классам обслуживания зависит от модели самолета, выполняющего рейс. В демобазе все самолеты одной модели имеют одинаковую конфигурацию салона.

Например, посмотрим на первый ряд:

```
SELECT s.*
FROM airplanes a
  JOIN seats s ON s.airplane_code = a.airplane_code
WHERE a.airplane_code = 'CR7'
      AND s.seat_no ~ '^1.$';
```

airplane_code	seat_no	fare_conditions
CR7	1A	Business
CR7	1C	Business
CR7	1D	Business

(3 rows)

Это бизнес-класс. А вот общее число мест различных классов обслуживания в салоне нашего самолета:

```
SELECT s.fare_conditions, count(*)
FROM seats s
WHERE s.airplane_code = 'CR7'
GROUP BY s.fare_conditions;
```

fare_conditions	count
Business	6
Comfort	12
Economy	52

(3 rows)

Посмотрим теперь внимательнее на все столбцы одного рейса, следующего выбранным маршрутом. Реальное время может отличаться от времени по расписанию: обычно не сильно (в данном случае вылет задержался на шесть минут), но иногда и на несколько часов:

```
SELECT * FROM flights f
WHERE f.flight_id = 27013 \gx
```

15
i

```
-[ RECORD 1 ]-----+-----
flight_id      | 27013
route_no       | PG0781
status         | Departed
scheduled_departure | 2026-03-01 02:50:00+03
scheduled_arrival  | 2026-03-01 04:15:00+03
actual_departure  | 2026-03-01 02:56:02.8521+03
actual_arrival   |
```

При регистрации на первый рейс пассажиру выдаются **посадочные талоны** (boarding_passes) на все рейсы в его билете. В посадочном талоне указано место в самолете; комбинация рейса и места уникальна — на одном кресле не могут сидеть двое. При посадке в талон проставляются порядковый номер и время прохода через гейт.

На каких местах сидел или будет сидеть наш пассажир?

```
SELECT f.status, f.flight_id, bp.seat_no, bp.boarding_no
FROM tickets t
  JOIN segments s ON s.ticket_no = t.ticket_no
  JOIN flights f ON f.flight_id = s.flight_id
  LEFT JOIN boarding_passes bp
    ON bp.ticket_no = s.ticket_no
    AND bp.flight_id = s.flight_id
WHERE t.ticket_no IN ('0005435998767','0005435998770')
ORDER BY f.scheduled_departure;
```

status	flight_id	seat_no	boarding_no
Arrived	17386	50A	41
Arrived	17606	21D	16
Arrived	26693	11A	176
Departed	27013	7C	49
Scheduled	27375	45D	
Scheduled	27537	13B	

(6 rows)

- 16 В демобазе заложена возможность перевода названий
i аэропортов, городов, стран и самолетов на другие языки. Выбор языка управляется конфигурационным параметром, настроенным на уровне базы данных:

```
SHOW bookings.lang;
bookings.lang
-----
ru
(1 row)
```

Установим для текущего сеанса английский язык:

```
SET bookings.lang = 'en';
SET
SELECT * FROM airports a
WHERE a.airport_code = 'SV0' \gx
-[ RECORD 1 ]+-----
airport_code | SV0
airport_name | Sheremetyevo
city         | Moscow
country      | Russia
coordinates  | (37.4146,55.9726)
timezone     | Europe/Moscow
```

Реализация использует представления над базовыми таблицами, которые содержат переводы в формате JSON:

```
SELECT ml.airport_name, ml.country, ml.city
FROM airports_data ml
WHERE ml.airport_code = 'SV0' \gx
-[ RECORD 1 ]+-----
airport_name | {"en": "Sheremetyevo", "ru": "Шереметьево"}
country      | {"en": "Russia", "ru": "Россия"}
city         | {"en": "Moscow", "ru": "Москва"}
```

Подведем итог списком таблиц и представлений демобазы:

\dtv

17
i

List of relations			
Schema	Name	Type	Owner
bookings	airplanes	view	postgres
bookings	airplanes_data	table	postgres
bookings	airports	view	postgres
bookings	airports_data	table	postgres
bookings	boarding_passes	table	postgres
bookings	bookings	table	postgres
bookings	flights	table	postgres
bookings	routes	table	postgres
bookings	seats	table	postgres
bookings	segments	table	postgres
bookings	tickets	table	postgres
bookings	timetable	view	postgres

(12 rows)

II. Планирование и выполнение

1. Простой протокол запросов

Сначала рассмотрим, как выполняется запрос в простом случае — например, если в `psql` написать команду `SELECT`. Обработка выполняется в несколько этапов:

- 1) разбор;
- 2) переписывание (трансформация);
- 3) планирование (оптимизация);
- 4) выполнение.



Разбор

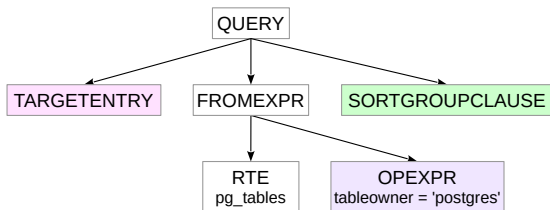
Синтаксический разбор

Сначала производится синтаксический разбор (`parse`): анализатор разбирает текст запроса на лексемы (идентификаторы, числа, строковые литералы и т. п.) и проверяет, что они складываются в синтаксически верные конструкции.

20 Рассмотрим простой пример:

ii

```
SELECT schemaname, tablename  
FROM pg_tables  
WHERE tableowner = 'postgres'  
ORDER BY tablename;
```



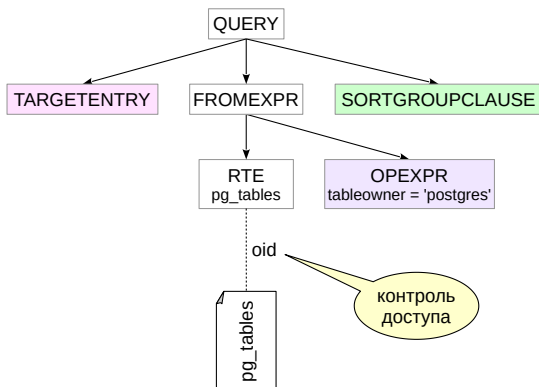
На этапе синтаксического разбора в памяти обслуживающего процесса будет построено дерево разбора, упрощенно показанное на рисунке ниже запроса. Цветом показано примерное соответствие частей текста запроса и узлов дерева.

(RTE — неочевидное сокращение от Range Table Entry. Этим именем внутри PostgreSQL называются таблицы, подзапросы, результаты соединений — иными словами, наборы строк, которыми может оперировать SQL.)

Семантический разбор

Затем выполняется семантический разбор (parse analysis), в ходе которого определяется, на какие объекты БД ссылается запрос и есть ли у пользователя доступ к ним.

```
SELECT schemaname, tablename  
FROM pg_tables  
WHERE tableowner = 'postgres'  
ORDER BY tablename;
```



На этапе семантического разбора анализатор сверяется с системным каталогом. В нашем примере он связывает имя `pg_tables` с представлением, имеющим определенный идентификатор (`oid`) в системном каталоге. Также будут проверены права доступа к этому представлению.

Переписывание

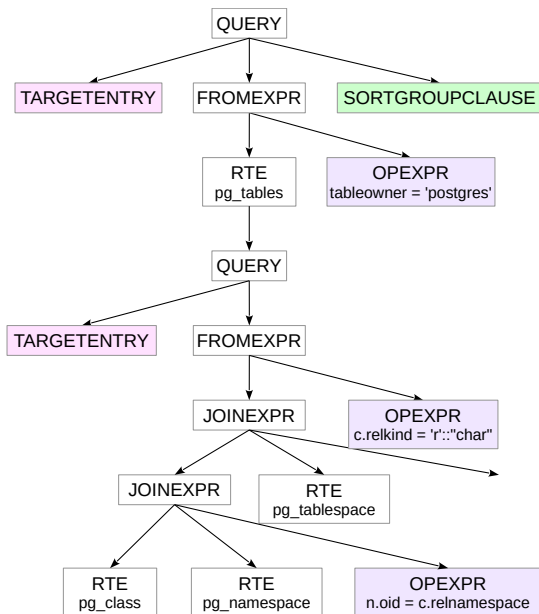
Далее запрос переписывается, или трансформируется (`rewrite`) с учетом правил (`rules`).

Важный частный случай переписывания — подстановка текста запроса вместо имени представления. Заметим, что текст представления опять необходимо разобрать, поэтому мы несколько упрощаем, говоря, что первые два этапа происходят друг за другом.

```

22 SELECT schemaname, tablename
ii  FROM (
      SELECT ...
      FROM pg_class c
           LEFT JOIN pg_namespace n
             ON n.oid = c.relnamespace
           LEFT JOIN pg_tablespace t
             ON t.oid = c.reltablespace
      WHERE c.relkind = 'r'::"char"
    )
    WHERE tableowner = 'postgres'
    ORDER BY tablename;

```



На рисунке приведен запрос с подставленным текстом представления. Это условность: вся работа по переписыванию происходит только с деревом запроса. В дереве узлы, соответствующие подзапросу, добавлены к узлу RTE (pg_tables).

23
ii

Планирование

Затем запрос планируется (plan).

Планировщик перебирает различные способы выполнения запроса и оценивает их с помощью некоторой математической модели, исходя из информации об обрабатываемых данных (статистики). Тот способ, для которого прогнозируется минимальная стоимость, представляется в виде дерева плана.

Sort

Sort Key: c.relname

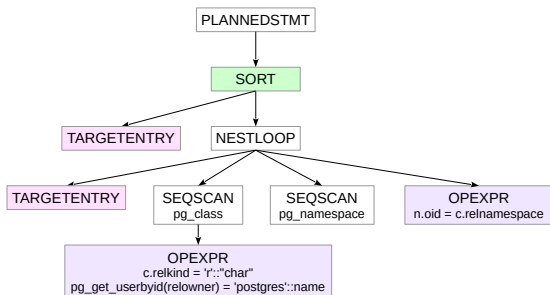
-> Nested Loop Left Join

Join Filter: (n.oid = c.relnamespace)

-> Seq Scan on pg_class c

Filter: ((relkind = 'r'::"char") AND pg_ge...

-> Seq Scan on pg_namespace n



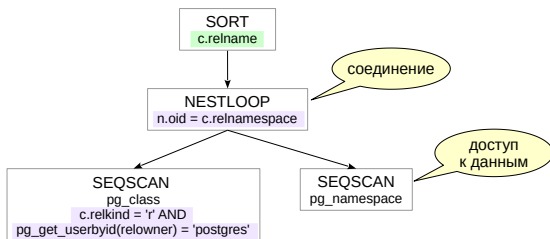
- 24 Здесь операция Seq Scan — это чтение соответствующих
ii таблиц, а Nested Loop — соединение двух таблиц. В виде
текста на рисунке приведен план выполнения в том виде,
как его показывает команда EXPLAIN. Подробно о методах
доступа к данным и о способах соединений мы будем го-
ворить в следующих главах.

Обратите внимание, что из трех таблиц осталось только две:
планировщик сообразил, что одна из таблиц не нужна для
получения результата и ее можно безболезненно удалить
из дерева плана.

Исполнение

Наконец, запрос выполняется (execute) в соответствии с вы-
бранным планом, и результат возвращается клиенту.

На этапе исполнения построенное ранее дерево плана
(на рисунке оно перерисовано так, чтобы показать только
самое основное) работает как конвейер.



Выполнение начинается с корня дерева. Корневой узел
(в нашем случае это операция сортировки SORT) обраща-

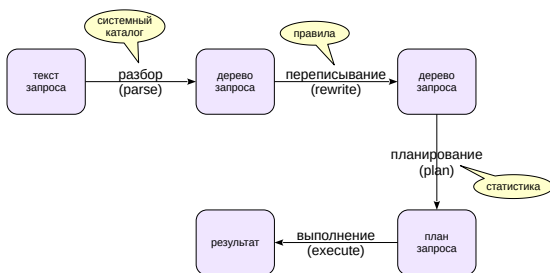
ется за данными к дочернему узлу; получив данные, он выполняет свою работу (сортировку) и отдает данные наверх (то есть клиенту).

25
ii

Такие узлы, как NESTLOOP на рисунке, соединяют данные, полученные из разных источников. Этот узел обращается по очереди к двум дочерним узлам (наборы строк всегда соединяются попарно), сопоставляет полученные от них строки и возвращает найденные пары наверх (то есть узлу сортировки).

Два нижних узла SEQSCAN представляют обращение к таблице за данными. Они читают строки из соответствующих таблиц и отдают наверх (то есть узлу соединения).

Некоторые узлы могут вернуть результат, только получив от дочерних узлов все данные. К таким узлам относится сортировка — нельзя отсортировать неполную выборку. Другие узлы могут возвращать данные по мере поступления. Например, доступ к таблице может отдавать данные по мере чтения (это позволяет быстро получить первую часть результата — например, для страничного отображения на сайте).



2. Расширенный протокол

По умолчанию применяется простой протокол запросов: мы отправляем на сервер оператор и (если это SELECT или команда с предложением RETURNING) получаем все строки результата. Например:

```
SELECT model FROM airplanes WHERE airplane_code = '77W';  
model  
-----  
Боинг 777-300ER  
(1 row)
```

В таком случае каждый запрос проходит все описанные этапы обработки, даже если один и тот же запрос выполняется несколько раз.



PostgreSQL также предусматривает расширенный протокол запросов, позволяющий более детально управлять исполнением. На практике это означает возможность подготавливать операторы и использовать курсоры.

Подготовленные операторы

Для подготовки оператора клиент передает запрос серверу (возможно, в параметризованном виде), а сервер разбирает и переписывает этот запрос и сохраняет подготовленное дерево разбора в локальную память обслуживающего процесса.

Чтобы выполнить подготовленный запрос, клиент называет его имя и указывает конкретные значения параметров. Сервер строит частный (custom) план запроса, учитывая значения переданных параметров, и выполняет его.

Подготовка позволяет избежать повторного разбора и переписывания одного и того же запроса, если он выполняется в одном сеансе неоднократно.

Если запрос не имеет параметров, серверу нет смысла перепланировать запрос при каждом выполнении. В этом случае он сразу запоминает общий (generic) план выполнения. Это позволяет еще больше экономить ресурсы.

Если же запрос имеет параметры, то сервер может перейти на общий план, если сочтет, что в среднем он получается не хуже частных планов. Подробнее о том, в каких случаях происходит переключение, говорится в главе VI «Статистика».

Еще один плюс подготовленных операторов — гарантия защиты от внедрения SQL-кода, если данные, полученные из ненадежного источника (например, из поля ввода на веб-форме), передаются запросу через параметры.

Подготовим запрос с параметром:

```
PREPARE model(varchar) AS
  SELECT model
  FROM airplanes
  WHERE airplane_code = $1;
PREPARE
```

Теперь мы можем вызывать оператор по имени:

```
EXECUTE model('CR7');
      model
-----
Бомбардье CRJ700
(1 row)
```

При этом происходит привязка значения к параметру (binding), перепланирование и исполнение.

Все подготовленные операторы видны в представлении:

```
SELECT * FROM pg_prepared_statements \gx
-[ RECORD 1 ]-----+-----
name          | model
statement     | PREPARE model(varchar) AS  +
               | SELECT model               +
               | FROM airplanes             +
               | WHERE airplane_code = $1;
prepare_time  | 2026-08-05 19:39:57.602451+03
parameter_types | {"character varying"}
result_types  | {text}
from_sql      | t
generic_plans | 0 ← выполнения с общим планом
custom_plans  | 1 ← выполнения с частными планами
```

Если подготовленный оператор больше не нужен, его можно удалить командой DEALLOCATE, но в любом случае оператор пропадет при завершении сеанса.

PREPARE, EXECUTE, DEALLOCATE — команды языка SQL. Аналогичные операции в других языках программирования будут выглядеть иначе, но драйвер базы данных в любом случае будет использовать тот же самый клиент-серверный протокол.

Курсоры

Курсоры дают возможность получать результат построчно. Они часто используются, поскольку обычно приложения создаются на императивных языках программирования, подразумевающих итеративную обработку данных.

Информация об открытом курсоре также хранится в локальной памяти обслуживающего процесса.

Объявляем курсор на языке SQL (при этом курсор сразу же открывается) и выбираем первую строку:

29
ii

```
BEGIN;  
DECLARE c CURSOR FOR SELECT * FROM airplanes;  
FETCH c;  


| airplane_code | model           | range | speed |
|---------------|-----------------|-------|-------|
| 32N           | Аэробус A320neo | 6500  | 830   |

  
(1 row)
```

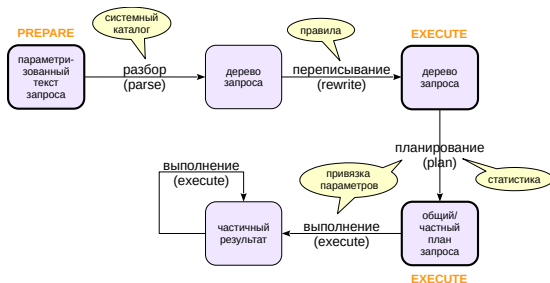
Читаем вторую строку результата (курсор закроется автоматически по окончании транзакции):

```
FETCH c;  


| airplane_code | model               | range | speed |
|---------------|---------------------|-------|-------|
| 339           | Аэробус A330-900neo | 13300 | 870   |

  
(1 row)  
COMMIT;
```

Все языки программирования позволяют определить, остались ли еще непрочитанные данные, благодаря чему можно обращаться к курсору в цикле.



3. Подробнее о планировании

Планирование запроса — очень важный и весьма сложный этап, поэтому остановимся на нем подробнее.

Процесс планирования

Оптимизатор перебирает и оценивает все возможные планы выполнения и выбирает среди них план с наименьшей стоимостью.

При расчете стоимости узла плана оптимизатор в первую очередь учитывает **тип узла** (понятно, что стоимость чтения данных напрямую из таблицы будет отличаться от стоимости чтения по индексу) и **объем данных**, обрабатываемых этим узлом.

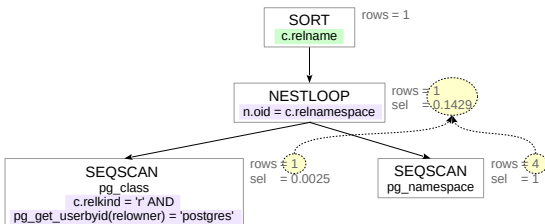
Для оценки объема данных важны два понятия:

- **кардинальность** — общее число строк на выходе узла;
- **селективность** — доля исходных строк, отбираемых условиями (предикатами) в узле.

Для оценки селективности и кардинальности надо, в свою очередь, иметь сведения о данных: размер таблиц, распределение значений в столбцах и др. Таким образом, в итоге все сводится к **статистике**. Она собирается и обновляется процессом автоанализа или командой ANALYZE.

Если кардинальность оценена правильно, то и стоимость обычно рассчитывается довольно точно. Основные ошибки оптимизатора связаны именно с неправильной оценкой кардинальности (что может происходить из-за неадекватной статистики, невозможности ее использования или несовершенства моделей оптимизатора).

Оценку кардинальности удобно рассматривать как рекурсивный процесс. Чтобы оценить кардинальность узла, надо оценить кардинальности его дочерних узлов, а затем — зная тип узла — вычислить на их основе кардинальность самого узла.



Таким образом, сначала необходимо рассчитать кардинальности листовых узлов, в которых находятся методы доступа к данным. Для этого нам нужно знать размер таблицы и селективность условий, наложенных на нее. Как конкретно это делается, мы рассмотрим позже.

Пока отметим, что достаточно уметь оценивать селективность простых условий, а селективность условий, составленных с помощью логических операций, рассчитываются по простым формулам:

$$sel_{x \text{ and } y} = sel_x sel_y ;$$

$$sel_{x \text{ or } y} = 1 - (1 - sel_x)(1 - sel_y).$$

Эти формулы предполагают независимость условий x и y . В случае коррелированных предикатов такая оценка будет неточной (но ее можно улучшить с помощью расширенной статистики).

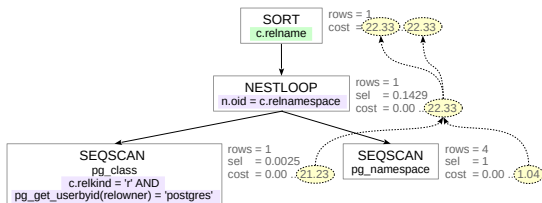
- 32 Затем можно рассчитать кардинальности соединений. Кар-
- ii диальности соединяемых наборов данных нам уже известны, осталось оценить селективность условий соединения. Пока мы не будем углубляться в это, но далее рассмотрим примеры такой оценки.

Аналогично можно поступить и с другими узлами, например, с сортировками или агрегациями.

Важно отметить, что ошибка расчета кардинальности, возникшая в нижних узлах, будет распространяться выше, приводя в итоге к неверной (обычно заниженной) оценке и выбору неудачного плана.

Оценка стоимости

Теперь рассмотрим общий процесс оценки стоимости. Он также рекурсивен по своей природе. Чтобы рассчитать стоимость поддерева, сначала надо вычислить стоимости дочерних узлов и сложить их, а затем добавить стоимость самого узла.



Стоимость работы узла определяется на основе математической модели, заложенной в планировщик, с учетом оценки кардинальности (которая уже рассчитана).

Стоимость состоит из двух компонент, которые оцениваются отдельно: **начальная** — стоимость подготовительной работы, и **полная** — стоимость получения всех строк.

Некоторые операции не требуют никакой подготовки; у таких узлов начальная стоимость будет равна нулю.

Другие операции, наоборот, требуют выполнения предварительных действий. Например, сортировка в приведенном примере должна сначала получить от дочернего узла все данные, чтобы начать работу. У таких узлов начальная стоимость будет отлична от нуля — эту стоимость придется «заплатить» независимо от того, сколько строк результата потребуется.

Важно понимать, что стоимость отражает оценку планировщика и может не коррелировать с реальным временем выполнения. Можно считать, что стоимость выражена в неких «условных единицах», которые сами по себе ни о чем не говорят. Стоимость нужна лишь для того, чтобы планировщик мог сравнивать разные планы одного и того же запроса.

Выбор лучшего плана

Оптимизатор старается перебрать все возможные планы выполнения запроса, чтобы выбрать из них лучший.

Для сокращения перебора используется алгоритм динамического программирования, но при большом количестве вариантов (в первую очередь из-за числа соединяемых таблиц) точное решение задачи оптимизации за разумное время становится невозможным. В таком случае планировщик сокращает количество перебираемых планов, либо рассматривая не все варианты попарных соединений,

- 34 либо переключаясь на генетический алгоритм оптимизации. Это может привести к тому, что оптимизатор выберет неудачный план не из-за ошибок в оценке, а просто потому, что лучший план не рассматривался. В таких случаях автор запроса может подсказать планировщику желаемый порядок соединений.
- ii

Что же считается «лучшим планом»? Для обычных запросов это план, минимизирующий время получения всех строк, то есть план с минимальной полной стоимостью.

Однако при использовании курсоров может быть важно как можно быстрее получить первые строки. Поэтому существует параметр *cursor_tuple_fraction* (значение по умолчанию 0.1), задающий долю строк, которую надо получить как можно быстрее. Чем меньше значение этого параметра, тем больше на выбор плана влияет начальная стоимость, а не полная.

Просмотр плана запроса



Для получения плана запроса служит команда EXPLAIN. Она выводит в текстовом виде информацию, которую мы показывали на рисунках. Команда EXPLAIN — незаменимый инструмент специалиста по оптимизации запросов, и его необходимо хорошо освоить.

Вот план запроса, который мы рассматривали:

```
EXPLAIN SELECT schemaname, tablename
FROM pg_tables
WHERE tableowner = 'postgres'
ORDER BY tablename;
```

```

Sort (cost=22.36..22.37 rows=1 width=128)
  Sort Key: c.relname
    -> Nested Loop Left Join
      (cost=0.00..22.35 rows=1 width=128)
      Join Filter: (n.oid = c.relnamespace)
        -> Seq Scan on pg_class c
          (cost=0.00..21.26 rows=1 width=72)
          Filter: ((relkind = ANY ('{r,p}':"char"[])) ...)
        -> Seq Scan on pg_namespace n
          (cost=0.00..1.04 rows=4 width=68)
(10 rows)

```

Если добавить параметр `analyze`, сервер выполнит запрос и покажет не только оценки, но и фактические значения:

```

EXPLAIN (analyze, buffers off) SELECT schemaname, tablename
FROM pg_tables
WHERE tableowner = 'postgres'
ORDER BY tablename;

```

QUERY PLAN

```

Sort (cost=22.36..22.37 rows=1 width=128)
  (actual time=0.101..0.103 rows=77.00 loops=1)
  Sort Key: c.relname
  Sort Method: quicksort  Memory: 35kB
    -> Nested Loop Left Join
      (cost=0.00..22.35 rows=1 width=128)
      (actual time=0.009..0.080 rows=77.00 loops=1)
      Join Filter: (n.oid = c.relnamespace)
      Rows Removed by Join Filter: 112
        -> Seq Scan on pg_class c
          (cost=0.00..21.26 rows=1 width=72)
          (actual time=0.006..0.039 rows=77.00 loops=1)
          Filter: ((relkind = ANY ('{r,p}':"char"[])) ...)
          Rows Removed by Filter: 382
        -> Seq Scan on pg_namespace n
          (cost=0.00..1.04 rows=4 width=68)
          (actual time=0.000..0.000 rows=2.45 loops=77)
Planning Time: 0.080 ms
Execution Time: 0.113 ms
(19 rows)

```

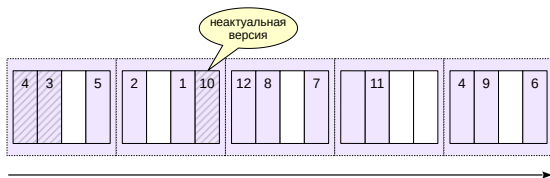
- 36 Будьте аккуратны с запросами, изменяющими данные!
- ii Вывод времени выполнения каждого шага, как в этом примере, может существенно замедлять выполнение запроса на некоторых платформах. Если такая информация не нужна, лучше ее отключить, добавив параметр `timing off`.

III. Методы доступа

1. Последовательное сканирование

В распоряжении оптимизатора имеется несколько способов доступа к данным. Самый простой из них – последовательное сканирование (Seq Scan) таблицы.

Файл (или файлы) основного слоя таблицы читается постранично от начала до конца. Напомним, что чтение происходит через буферный кеш (для временных таблиц – через локальный кеш сеанса).



Последовательное чтение файла позволяет использовать тот факт, что операционная система обычно читает данные порциями большими, чем размер страницы – с большой вероятностью несколько следующих страниц уже окажутся в кеше ОС.

Последовательное сканирование эффективно работает, когда надо прочитать всю таблицу или значительную ее часть (если селективность условия низка).

38 В плане выполнения запроса последовательное сканиро-
iii вание представлено узлом Seq Scan:

```
EXPLAIN SELECT * FROM flights;  
               QUERY PLAN  
-----  
Seq Scan on flights  
  (cost=0.00..744.71 rows=37671 width=51)  
(2 rows)
```

В скобках приведены важные значения:

- cost — оценка стоимости;
- rows — оценка числа строк, возвращаемых операцией;
- width — оценка среднего размера одной табличной строки в байтах.

Стоимость указывается в условных единицах и состоит из двух компонент:

- Первое число показывает **начальную стоимость** вычисления узла. Для последовательного сканирования это ноль — чтобы начать возвращать данные, никакой подготовки не требуется.
- Второе число показывает **полную стоимость** получения всех данных.

Оптимизатор PostgreSQL учитывает ввод-вывод и ресурсы процессора. Составляющая ввода-вывода рассчитывается как произведение числа страниц в таблице на условную стоимость чтения одной страницы:

```
SELECT relpages,  
       current_setting('seq_page_cost'),  
       relpages * current_setting('seq_page_cost')::real AS total  
FROM pg_class WHERE relname = 'flights';
```

relpages	current_setting	total
368	1	368

(1 row)

Составляющая ресурсов процессора складывается из стоимости обработки каждой строки:

```
SELECT reltuples,
       current_setting('cpu_tuple_cost'),
       reltuples * current_setting('cpu_tuple_cost')::real AS total
FROM pg_class WHERE relname = 'flights';
```

reltuples	current_setting	total
37671	0.01	376.71

(1 row)

Мертвые версии строк

При последовательном сканировании на каждой странице рассматриваются все версии строк — в том числе и неактуальные (мертвые) версии, которые еще не удалены процедурой очистки (процессом автоочистки). Если в файлах таблицы остается много мертвых, но еще не вычищенных версий строк, это будет приводить к замедлению скорости работы последовательного сканирования.



В частности поэтому своевременное выполнение очистки может существенно ускорить выполнение некоторых запросов.



Создадим копию таблицы `flights`, отключив для нее автоочистку, чтобы мертвые версии строк не удалялись автоматически, и проверим размер полученной таблицы:

```
40 CREATE TABLE flights_copy
iii WITH (autovacuum_enabled = false)
AS SELECT * FROM flights;

SELECT 37671

SELECT pg_size_pretty(
    pg_total_relation_size('flights_copy')
);

pg_size_pretty
-----
3024 kB
(1 row)
```

Удалим все строки из этой таблицы:

```
DELETE FROM flights_copy;

DELETE 37671
```

Мертвые версии строк останутся в файле таблицы до ближайшей очистки, поэтому размер таблицы не изменится:

```
SELECT pg_size_pretty(
    pg_total_relation_size('flights_copy')
);

pg_size_pretty
-----
3024 kB
(1 row)
```

Получим план запроса с параметром `analyze`. Подразумеваемый параметр `buffers` показывает количество страниц:

- `hit` — найденных в кеше;
- `read` — не найденных в кеше и запрошенных у операционной системы;
- `written` — записанных в файлы;
- `dirtied` — ставших грязными (то есть впервые измененных в кеше).

```
EXPLAIN (analyze, timing off, summary off)
SELECT count(*) FROM flights_copy;
```

41
iii

QUERY PLAN

```
-----
Aggregate  (cost=673.89..673.90 rows=1 width=8) (actual...
  Buffers: shared hit=377
    -> Seq Scan on flights_copy  (cost=0.00..614.51 rows=...
      Buffers: shared hit=377
Planning:
  Buffers: shared hit=3
(6 rows)
```

Прочитано более трехсот страниц. Очистим таблицу вручную и снова проверим размер таблицы:

```
VACUUM flights_copy;
SELECT pg_size_pretty(pg_total_relation_size('flights_copy'));
```

```
VACUUM
pg_size_pretty
-----
24 kB
(1 row)
```

Мертвые версии строк удалены, размер таблицы уменьшился. Снова получим план запроса:

```
EXPLAIN (analyze, timing off, summary off)
SELECT count(*) FROM flights_copy;
```

QUERY PLAN

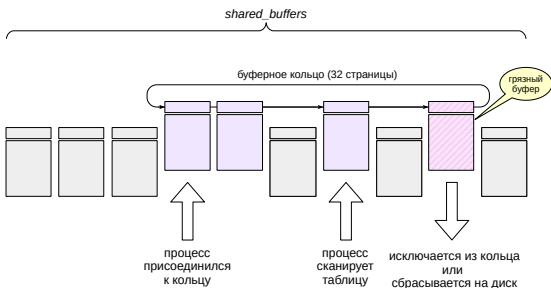
```
-----
Aggregate  (cost=0.00..0.01 rows=1 width=8) (actual row...
  -> Seq Scan on flights_copy  (cost=0.00..0.00 rows=1 ...
Planning:
  Buffers: shared hit=2
(4 rows)
```

Таблица пуста — не прочитано ни одной версии строки, стоимость плана минимальна. Время, затраченное на обработку неактуальных версий строк, может быть существенным для таблиц большего размера.

42 **DROP TABLE** flights_copy;
iii DROP TABLE

Массовое вытеснение

При последовательном сканировании через буферный кеш проходит большое количество страниц с «одноразовыми» данными, которые могут вытеснять полезные страницы из буферов. Чтобы этого не происходило, применяются так называемые буферные кольца. Для последовательного чтения большой таблицы (размер которой превышает четверть буферного кеша) из всего кеша используются только 32 страницы, в пределах которых действует вытеснение. При этом остальные данные в буфере не страдают.



Если страница, загруженная в кольцо, изменилась, появляется грязный буфер. Такой буфер отцепляется от кольца и будет вытеснен на общих основаниях, а в кольцо добавляется новый буфер. Такая стратегия рассчитана на то, что в основном данные читаются, а не меняются.

Если в процессе чтения другому процессу потребуется та же таблица, он не начинает читать ее с начала, а подклю-

чается к имеющемуся буферному кольцу. После окончания сканирования процесс дочитывает «пропущенное» начало таблицы.

43
iii

При работе с временными таблицами через локальный кеш механизм буферных колец не используется.

2. Сканирование индекса

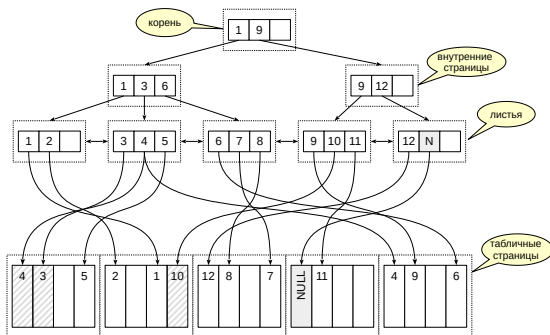
В этой главе мы будем рассматривать сканирование индекса (Index Scan) и только один из доступных в PostgreSQL типов индексов: В-дерево (B-tree). Это наиболее часто применяющийся на практике тип индекса.

Как и все индексы в PostgreSQL, В-дерево является вторичной структурой — индекс не содержит в себе никакой информации, которую нельзя было бы получить из самой таблицы, однако занимает дополнительное место на диске. Индекс можно удалить и пересоздать. Индексы нужны для ускорения операций, затрагивающих небольшую часть таблицы: выборки малого количества строк, а также поддержки ограничений целостности (первичных и уникальных ключей).

Индексы сопоставляют значения проиндексированных полей (ключи поиска) и идентификаторы строк таблицы. Для этого в индексе B-tree строится упорядоченное дерево ключей, в котором можно быстро найти нужный ключ, а вместе с ним — и ссылки на версии строк. Но проиндексировать можно только данные, допускающие сортировку (должны быть определены операции «больше», «меньше»). Например, числа, строки и даты могут быть проиндексированы, а точки на плоскости — нет (для них существуют другие типы индексов).

44
iii

Особенностями В-дерева является его сбалансированность (постоянная глубина) и сильная ветвистость. Хотя размер дерева зависит от проиндексированных столбцов, на практике деревья обычно имеют глубину не больше 4–5 уровней.



На рисунке приведен пример В-дерева. Его страницы состоят из индексных записей, каждая из которых содержит:

- ключ — значения столбцов, по которым построен индекс (такие столбцы называются ключевыми);
- ссылку на другую страницу индекса или ссылки на версии строк.

Внутри страницы ключи всегда упорядочены.

Листовые страницы ссылаются непосредственно на табличные версии строк, содержащие ключи индексирования. Эти страницы связаны двунаправленным списком, чтобы было удобно перебирать ключи в порядке возрастания или убывания.

Внутренние страницы ссылаются на нижележащие страницы индекса; значения ключей определяют диапазон значений, которые можно обнаружить, спустившись по ссылке.

Верхняя страница дерева, на которую нет ссылок, называется **корнем**.

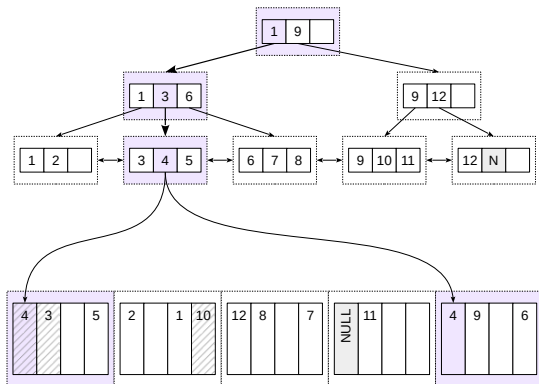
Индексная страница может быть заполнена частично. Свободное место используется для вставки в индекс новых записей. Если же на странице не хватает места — она разделяется: одна половина записей остается на старой странице, а другая переносится на новую. Разделившиеся страницы никогда не объединяются, и это в некоторых случаях может приводить к разрастанию индекса.

По умолчанию считается, что неопределенные значения «больше» обычных, и они хранятся в дереве справа. Этот порядок можно изменить при создании индекса с помощью предложений `NULLS LAST` и `NULLS FIRST`.

Поиск одного значения

Рассмотрим, как происходит поиск одного значения с помощью индекса. Например, нам нужно найти в таблице строку, в которой значение проиндексированного столбца равно четырем.

Начинаем с корня дерева. Индексные записи в корневой странице определяют диапазоны значений ключей в нижележащих страницах: «от 1 до 9» и «9 и больше». Нам подходит диапазон «от 1 до 9», что соответствует строке с ключом 1. Заметим, что ключи хранятся упорядоченно, следовательно, поиск внутри страницы выполняется очень эффективно.



Ссылка из найденной записи приводит нас к странице второго уровня. В ней мы находим диапазон «от 3 до 6» (ключ равен 3) и переходим к странице третьего уровня.

Эта страница является листовой. В ней мы находим ключ, равный 4, и переходим на страницу таблицы.

Заметьте, что ключи могут повторяться даже в уникальном индексе: из-за работы механизма многоверсионности могут возникать разные версии одной и той же строки. Для экономии места ключи хранятся в индексной странице в одном экземпляре.

Прежде чем вернуть найденные версии строк, необходимо проверить их видимость.

На иллюстрациях цветом выделены записи и страницы, которые потребовалось прочитать.

Возьмем таблицу бронирований `bookings`. В ней столбец `book_ref` является первичным ключом и для него автоматически был создан индекс `bookings_pkey`.

Проверим план запроса с поиском одного значения:

```
EXPLAIN SELECT * FROM bookings WHERE book_ref = 'CDEFD8';  
QUERY PLAN
```

```
-----  
Index Scan using bookings_pkey on bookings  
  (cost=0.43..8.45 rows=1 width=21)  
  Index Cond: (book_ref = 'CDEFD8'::bpchar)  
(3 rows)
```

Выбран метод доступа Index Scan, указано имя использованного индекса. Здесь обращение и к индексу, и к таблице представлено одним узлом плана. Строкой ниже указано условие доступа.

Начальная стоимость индексного доступа — оценка ресурсов для спуска к листовому узлу. Она зависит от высоты дерева. При оценке считается, что необходимые страницы окажутся в кеше, и оцениваются только ресурсы процессора: цифра получается небольшой.

Полная стоимость добавляет оценку чтения необходимых листовых страниц индекса и табличных страниц.

В данном случае, поскольку индекс уникальный, модель рассчитывает на то, что будет прочитана одна индексная страница и одна табличная. Стоимость каждого из чтений оценивается параметром *random_page_cost*:

```
SELECT current_setting('random_page_cost');  
current_setting  
-----  
4  
(1 row)
```

Его значение обычно больше, чем *seq_page_cost*, поскольку произвольный доступ стоит дороже (хотя для SSD-накопителей этот параметр следует существенно уменьшить).

48 Итого получаем 8, и еще немного добавляет оценка про-
iii цессорного времени на обработку строк.

В строке Index Cond плана указываются только те условия, по которым происходит обращение к индексу или которые могут быть проверены на уровне индекса.

Дополнительные условия, которые можно проверить только по таблице, отображаются в отдельной строке Filter:

```
EXPLAIN SELECT * FROM bookings
WHERE book_ref = 'CDEFD8' AND total_amount > 1000;
      QUERY PLAN
```

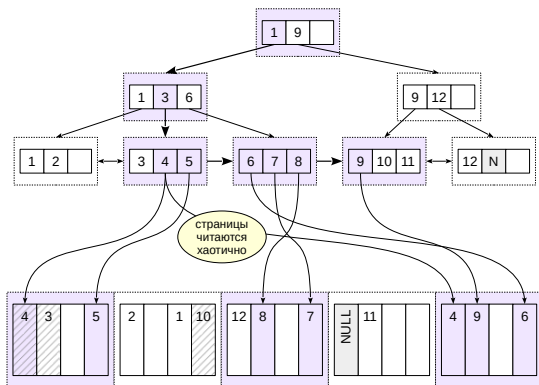
```
-----
Index Scan using bookings_pkey on bookings
  (cost=0.43..8.45 rows=1 width=21)
  Index Cond: (book_ref = 'CDEFD8'::bpchar)
  Filter: (total_amount > '1000'::numeric)
(4 rows)
```

Поиск диапазона значений

В-дерево позволяет эффективно искать не только отдельные значения, но и диапазоны значений (по условиям «меньше», «больше», «меньше или равно», «больше или равно», а также BETWEEN).

Сначала мы ищем крайний ключ условия. Например, для условия «от 4 до 9» мы можем выбрать значение 4 или 9, а для условия «меньше 9» надо взять 9. Затем спускаемся до листовой страницы индекса, как рассматривали в предыдущем примере, и получаем первое значение из таблицы.

Дальше остается двигаться по листовым страницам индекса вправо (или влево, в зависимости от условия), перебирая записи этих страниц до тех пор, пока мы не встретим ключ, выпадающий из диапазона.



На рисунке показан пример поиска значений по условию x BETWEEN 4 AND 9 или, что то же самое, $x \geq 4$ AND $x \leq 9$. Спустившись к значению 4, мы затем перебираем ключи 5, 6, и так далее до 9. Встретив ключ 10, прекращаем поиск.

Нам помогают два свойства: упорядоченность ключей на всех страницах и связанность листовых страниц двунаправленным списком. Результат поиска автоматически получается отсортированным.

Обратите внимание, что к одной и той же табличной странице нам пришлось обращаться несколько раз. Мы прочитали первую страницу таблицы (значение 4), затем последнюю (тоже 4), затем опять первую (5), затем опять последнюю (6) и так далее.

Мы получаем данные из индекса, спускаясь от корня дерева к левому листовому узлу и проходя по списку листовых страниц. Поэтому индексное сканирование всегда возвращает данные в том порядке, в котором они хранятся в дереве индекса и который был указан при его создании:

50 **EXPLAIN** (costs off)
iii **SELECT * FROM** bookings
 WHERE book_ref > '000CTL' **AND** book_ref < '000FHR'
 ORDER BY book_ref;

QUERY PLAN

Index Scan using bookings_pkey on bookings
 Index Cond: ((book_ref > '000CTL'::bpchar) **AND** (book_...
(2 rows)

Тот же самый индекс может использоваться и для получения строк в обратном порядке:

EXPLAIN (analyze, costs off, timing off, summary off)
SELECT * FROM bookings
WHERE book_ref > '000CTL' **AND** book_ref < '000FHR'
ORDER BY book_ref **DESC**;

QUERY PLAN

Index Scan **Backward** using bookings_pkey on bookings (ac...
 Index Cond: ((book_ref > '000CTL'::bpchar) **AND** (book_...
 Index Searches: 1
 Buffers: shared hit=3 read=5
 Planning:
 Buffers: shared hit=13
(6 rows)

В этом случае мы спускаемся от корня дерева к правому листовому узлу и проходим по списку листовых страниц в обратную сторону. Обратите внимание на количество страниц (Buffers), которое потребовалось прочитать, и на количество проходов по индексу (Index Searches).

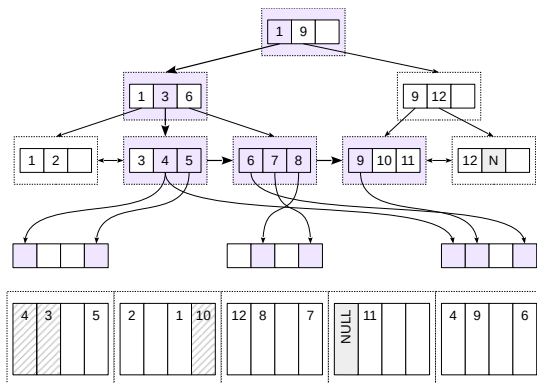
3. Сканирование по битовой карте

Многократный просмотр одних и тех же табличных страниц крайне неэффективен. Даже в лучшем случае, если нужная

страница осталась в буферном кеше, ее нужно найти и заблокировать.

51
iii

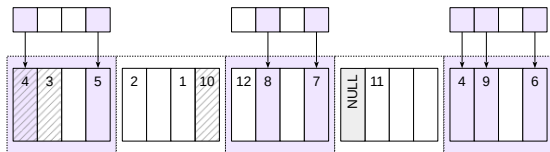
Чтобы не тратить ресурсы на повторный просмотр табличных страниц, применяется еще один способ доступа — сканирование по битовой карте (Bitmap Scan). Он похож на обычный индексный доступ, но происходит в два этапа.



Сначала сканируется индекс (Bitmap Index Scan) и в локальной памяти процесса строится битовая карта. Битовая карта состоит из фрагментов. Фрагменты соответствуют табличным страницам, а каждый бит фрагмента соответствует версии строки в этой странице. При построении битовой карты в ней отмечаются те версии строк, которые удовлетворяют условию и должны быть прочитаны. За счет деления битовой карты на фрагменты карта, в которой отмечено немного версий, будет занимать мало места.

Когда индекс просканирован и битовая карта готова, начинается сканирование таблицы (Bitmap Heap Scan).

- 52 На одной странице может проверяться несколько версий
iii строк, но каждая страница просматривается один раз.



Будем рассматривать таблицу бронирований bookings. Создадим на ней два дополнительных индекса:

```
CREATE INDEX ON bookings(book_date);  
CREATE INDEX ON bookings(total_amount);
```

Чтобы упорядочить сканирование большого числа страниц, планировщик выбрал сканирование по битовой карте:

```
EXPLAIN SELECT *  
FROM bookings WHERE total_amount < 5000;  
QUERY PLAN
```

```
-----  
Bitmap Heap Scan on bookings  
  (cost=6247.77..26270.44 rows=333334 width=21)  
    Recheck Cond: (total_amount < '5000'::numeric)  
    -> Bitmap Index Scan on bookings_total_amount_idx  
        (cost=0.00..6164.44 rows=333334 width=0)  
        Index Cond: (total_amount < '5000'::numeric)  
(6 rows)
```

Этот метод состоит из двух узлов:

- Bitmap Index Scan читает индекс и строит по нему битовую карту;
- Bitmap Heap Scan читает табличные страницы, используя построенную карту.

Обратите внимание на стоимости: карту нельзя использовать, пока она не построена полностью.

Кроме того, что битовая карта позволяет избежать повторных чтений табличных страниц, с ее помощью можно объединять несколько условий:

```
EXPLAIN (costs off)
SELECT * FROM bookings
WHERE total_amount < 5000 OR total_amount > 500000;
QUERY PLAN
```

```
-----
Bitmap Heap Scan on bookings
  Recheck Cond: ((total_amount < '5000'::numeric) OR (t...
-> BitmapOr
   -> Bitmap Index Scan on bookings_total_amount_idx
       Index Cond: (total_amount < '5000'::numeric)
   -> Bitmap Index Scan on bookings_total_amount_idx
       Index Cond: (total_amount > '500000'::numeric)
(7 rows)
```

Здесь сначала были построены две битовые карты — по одной на каждое условие, а затем объединены побитовой операцией «или» (узел BitmapOr).

В объединении могут участвовать и разные индексы:

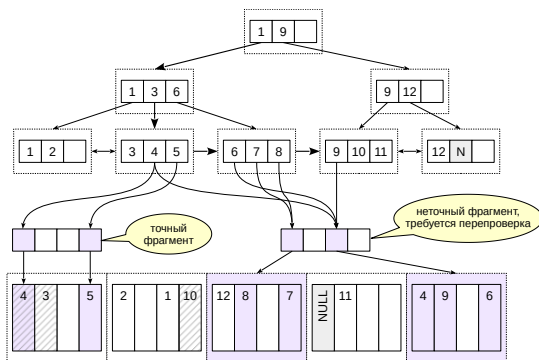
```
EXPLAIN (costs off)
SELECT * FROM bookings
WHERE total_amount < 5000
      OR book_date = bookings.now() - INTERVAL '1 day';
QUERY PLAN
-----
Bitmap Heap Scan on bookings
  Recheck Cond: ((total_amount < '5000'::numeric) OR (b...
-> BitmapOr
   -> Bitmap Index Scan on bookings_total_amount_idx
       Index Cond: (total_amount < '5000'::numeric)
   -> Bitmap Index Scan on bookings_book_date_idx
       Index Cond: (book_date = ('2026-03-01 03:00:0...
(7 rows)
```

Неточные фрагменты

Битовая карта хранится в локальной памяти обслуживающего процесса, и под ее хранение выделяется *work_mem* байт. Временные файлы никогда не используются.

Если карта перестает помещаться в *work_mem*, часть ее фрагментов «огрубляется» — каждый бит начинает соответствовать целой странице, а не отдельной версии строки (lossy bitmap). Стоимость обработки таких фрагментов увеличивается. Освободившееся место используется для того, чтобы продолжить строить карту.

В принципе, при сильно ограниченном *work_mem* и большой выборке битовая карта может не поместиться в памяти, даже если в ней совсем не останется информации на уровне версий строк. В таком случае ограничение *work_mem* нарушается — под карту будет дополнительно выделено столько памяти, сколько необходимо.



На рисунке показана битовая карта, состоящая из двух фрагментов. Первый фрагмент — точный, каждый его бит соответствует одной версии строки. Второй — неточный, в нем каждый бит соответствует целой странице.

Неточные фрагменты требуют перепроверки условий для всех версий строк в табличной странице, а это сказывается на производительности. Если две битовые карты объединяются, причем фрагмент хотя бы одной из карт неточен, то и результирующий фрагмент тоже вынужденно будет неточным. Поэтому размер *work_mem* играет большую роль для эффективности сканирования по битовой карте.

Узел Bitmap Heap Scan показывает условие перепроверки (Recheck Cond). Сама же перепроверка выполняется не всегда, а только при потере точности, когда битовая карта не помещается в память.

Снова будем искать бронирования с суммой до 5 тысяч ₽:

```
EXPLAIN (analyze, costs off, buffers off, timing off)
SELECT * FROM bookings WHERE total_amount < 5000;
```

QUERY PLAN

```
-----
Bitmap Heap Scan on bookings (actual rows=332628.00 loo...
  Recheck Cond: (total_amount < '5000'::numeric)
  Heap Blocks: exact=15839
    -> Bitmap Index Scan on bookings_total_amount_idx (ac...
        Index Cond: (total_amount < '5000'::numeric)
        Index Searches: 1
  Planning Time: 0.042 ms
  Execution Time: 69.415 ms
(8 rows)
```

Строка «Heap Blocks: exact» говорит о том, что все фрагменты битовой карты построены с точностью до строк — перепроверка не выполняется.

56 Уменьшим размер выделяемой памяти и повторим тот же
iii запрос:

```
SET work_mem = '512kB';  
EXPLAIN (analyze, costs off, buffers off, timing off)  
SELECT * FROM bookings WHERE total_amount < 5000;
```

QUERY PLAN

```
-----  
Bitmap Heap Scan on bookings (actual rows=332628.00 loo...  
  Recheck Cond: (total_amount < '5000'::numeric)  
  Rows Removed by Index Recheck: 1121673  
  Heap Blocks: exact=7581 lossy=8258  
    -> Bitmap Index Scan on bookings_total_amount_idx (ac...  
        Index Cond: (total_amount < '5000'::numeric)  
        Index Searches: 1  
  Planning Time: 0.050 ms  
  Execution Time: 87.380 ms  
(9 rows)
```

Здесь появились lossy-фрагменты битовой карты – с точностью до страниц. Также указано, сколько строк не прошло перепроверку условия (Rows Removed by Index Recheck).

Восстановим значение параметра.

```
RESET work_mem;
```

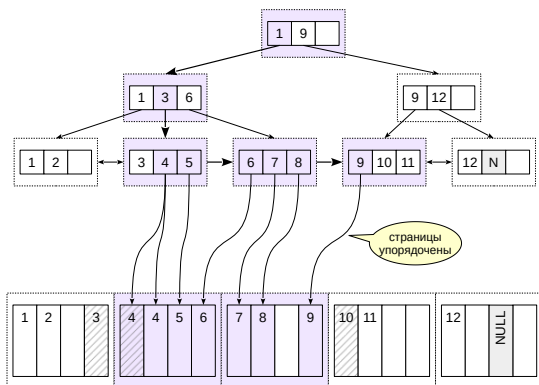
Кластеризация

Если данные в таблице физически упорядочены, обычное индексное сканирование не будет читать табличную страницу повторно.

В таком (нечастом на практике) случае метод сканирования по битовой карте теряет смысл, проигрывая обычному индексному сканированию. Разумеется, планировщик это тоже учитывает.

Если строки таблицы упорядочены так же, как и индекс, битовая карта становится излишней.

57
iii



Продemonстрируем это с помощью команды CLUSTER. Сейчас строки таблицы не имеют определенного физического порядка:

```
SELECT * FROM bookings LIMIT 10;
```

book_ref	book_date	total_amount
2EW1SQ	2025-09-01 03:00:12.557744+03	8125.00
3ZY305	2025-09-03 03:25:35.278453+03	7500.00
756UAS	2025-09-10 19:24:02.978559+03	6325.00
GC7I6S	2025-09-03 03:26:37.085293+03	18000.00
RBUIIQ	2025-09-01 03:00:33.126206+03	13000.00
GB7E53	2025-09-10 18:58:23.948989+03	11000.00
6TAS39	2025-09-01 03:00:06.265219+03	5000.00
UEX925	2025-09-03 03:27:39.815145+03	19550.00
GGY5EU	2025-09-01 03:00:07.565193+03	5750.00
MS148V	2025-09-03 03:29:12.70515+03	6750.00

(10 rows)

- 58 Переупорядочим строки в соответствии с индексом по
iii столбцу `total_amount`. Выполним код в транзакции, чтобы
кластеризация не повлияла на последующие команды:

```
BEGIN;  
CLUSTER bookings USING bookings_total_amount_idx;  
CLUSTER
```

Убедимся, что строки упорядочены по стоимости:

```
SELECT * FROM bookings LIMIT 10;
```

book_ref	book_date	total_amount
7ZHP9E	2025-09-10 19:44:37.68484+03	1750.00
PZ01NI	2025-11-12 03:45:47.817622+03	1750.00
6BC1L3	2025-09-03 05:32:45.971723+03	1750.00
YQ01XT	2025-09-24 03:35:58.864739+03	1750.00
EQXEG8	2025-11-05 03:52:11.774956+03	1750.00
BS0GMN	2025-09-17 03:36:35.340508+03	1750.00
9HLHGP	2026-02-18 04:09:04.979717+03	1750.00
SFR4P7	2025-10-15 03:37:58.510102+03	1750.00
XDD0AF	2025-11-26 04:06:02.505752+03	1750.00
BKVSNB	2025-10-02 07:44:39.551269+03	1750.00

(10 rows)

```
ANALYZE bookings;
```

```
EXPLAIN (costs off)
```

```
SELECT * FROM bookings WHERE total_amount < 5000;
```

QUERY PLAN

```
-----  
Index Scan using bookings_total_amount_idx on bookings  
Index Cond: (total_amount < '5000'::numeric)  
(2 rows)
```

```
ROLLBACK;
```

До кластеризации использовалось сканирование по битовой карте, но теперь проще и выгоднее сделать обычное индексное сканирование.

- Команда `CLUSTER` устанавливает исключительную блокировку, поскольку полностью перестраивает таблицу (как `VACUUM FULL`);
- Строки будут упорядочены, но далее в процессе работы кластеризация может ухудшиться — порядок строк поддерживаться не будет.

4. Сканирование только индекса

Index-Only Scan

Если в запросе требуются только проиндексированные данные, они уже есть в самом индексе и к таблице обращаться не надо. Такой индекс называется покрывающим для запроса.

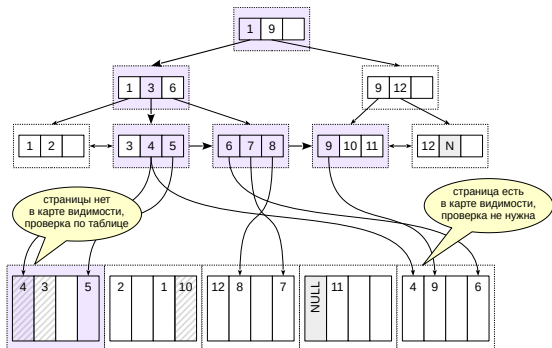
Это хорошая оптимизация, исключающая обращения к табличным страницам. Но, к сожалению, индексные страницы не содержат информацию о видимости строк — чтобы проверить, надо ли показывать найденную в индексе строку, мы вынуждены заглянуть и в табличную страницу, что сводит оптимизацию на нет.



Поэтому для эффективности сканирования только индекса критична карта видимости. Если табличная страница содержит гарантированно видимые данные и это отражено в карте видимости, к такой табличной странице обращаться не надо. Это одна из причин, по которым стоит выполнять

60
iii

очистку достаточно часто — именно этот процесс обновляет карту видимости.



Планировщик не знает точно, сколько табличных страниц потребуют проверки, но учитывает оценку этого числа. При плохом прогнозе планировщик может отказаться от использования сканирования только индекса.

При обработке каждой индексной записи листового узла сначала проверяется наличие табличной страницы в карте видимости, а при ее отсутствии читается сама табличная страница.

В плане запроса такой узел будет отображаться как Index-Only Scan. Но страницы, не отмеченные в карте видимости, посетить все-таки придется.

```
EXPLAIN (analyze, buffers off, costs off, timing off)
SELECT total_amount
FROM bookings
WHERE total_amount > 200000;
```

```
-----
Index Only Scan using bookings_total_amount_idx on book...
  Index Cond: (total_amount > '200000'::numeric)
    Heap Fetches: 0
    Index Searches: 1
  Planning Time: 0.122 ms
  Execution Time: 0.150 ms
(6 rows)
```

Строка Heap Fetches показывает, сколько версий строк было проверено с помощью таблицы. В данном случае карта видимости содержит актуальную информацию, обращаться к таблице не потребовалось.

Обновим первую строку таблицы:

```
UPDATE bookings
SET total_amount = total_amount
WHERE book_ref = '00EV9A';
UPDATE 1
```

Теперь пришлось проверять все версии, попавшие на измененную страницу:

```
EXPLAIN (analyze, buffers off, costs off, timing off)
SELECT total_amount
FROM bookings
WHERE total_amount > 200000;
```

QUERY PLAN

```
-----
Index Only Scan using bookings_total_amount_idx on book...
  Index Cond: (total_amount > '200000'::numeric)
    Heap Fetches: 3
    Index Searches: 1
  Planning Time: 0.041 ms
  Execution Time: 0.136 ms
(6 rows)
```

Include-индексы

Покрывающий индекс, как правило, увеличивает эффективность выборки. Добавление ключевых столбцов может сделать индекс покрывающим, хотя это и не всегда возможно:

- добавление столбца в уникальный индекс нарушит гарантию уникальности исходных столбцов;
- тип данных добавляемого столбца может не поддерживаться индексом.

В таких случаях можно добавить к индексу неключевые столбцы, указав их в предложении `INCLUDE`.

Значения таких столбцов не формируют дерево индекса, а просто хранятся как дополнительные сведения в индексных записях листовых страниц. Поиск по неключевым столбцам не работает, но их значения могут быть получены без обращения к таблице.

В настоящее время `include`-индексы поддерживаются только для индексов на основе B-деревьев, GiST и SP-GiST-индексов.

`Include`-индексы создаются, чтобы индекс стал покрывающим, но не стоит путать эти два термина. Индекс вполне может быть покрывающим для некоторого запроса, но не использовать предложение `INCLUDE`. А `include`-индекс может не быть покрывающим для каких-то запросов.

Индекс `tickets_pkey` не является покрывающим для приведенного запроса, поскольку требуется вернуть не только столбец `ticket_no`, который есть в индексе, но и `book_ref`, которого в индексе нет:

```
EXPLAIN (analyze, costs off, summary off)
SELECT ticket_no, book_ref
FROM tickets
WHERE ticket_no > '0005435990286';
```

QUERY PLAN

```
-----
Index Scan using tickets_pkey on tickets (actual time=0...
  Index Cond: (ticket_no > '0005435990286'::text)
  Index Searches: 1
  Buffers: shared hit=1496831 read=23275 written=16
Planning:
  Buffers: shared hit=45 read=7 dirtied=1
(6 rows)
```

Напомним, что строка Buffers показывает количество прочитанных страниц (hit+read).

Создадим include-индекс, добавив в него неключевой столбец book_ref, так как он требуется запросу:

```
CREATE UNIQUE INDEX ON tickets (ticket_no)
INCLUDE (book_ref);
CREATE INDEX
```

Повторим запрос:

```
EXPLAIN (analyze, costs off, summary off)
SELECT ticket_no, book_ref
FROM tickets
WHERE ticket_no > '0005435990286';
```

QUERY PLAN

```
-----
Index Only Scan using tickets_ticket_no_book_ref_idx on...
  Index Cond: (ticket_no > '0005435990286'::text)
  Heap Fetches: 0
  Index Searches: 1
  Buffers: shared hit=1 read=8033
Planning:
  Buffers: shared hit=18 read=1 dirtied=1
(7 rows)
```

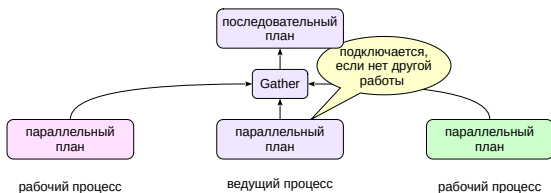
Теперь оптимизатор выбирает метод Index Only Scan и использует только что созданный индекс.

Поскольку карта видимости актуальна, обращаться к таблице не пришлось (Heap Fetches: 0). Соответственно, и количество прочитанных страниц сократилось.

Поскольку столбцы, добавленные в индекс с помощью предложения INCLUDE, никак не используются индексным методом, в include-индекс можно включать столбцы и с типами данных, которые не поддерживаются B-деревом (например, геометрические типы или xml).

5. Параллельный доступ

PostgreSQL может выполнять запросы параллельно. Ведущий процесс порождает несколько рабочих процессов, которые одновременно выполняют одну и ту же «параллельную» часть плана. Результаты их работы передаются ведущему процессу и собираются в узле Gather.



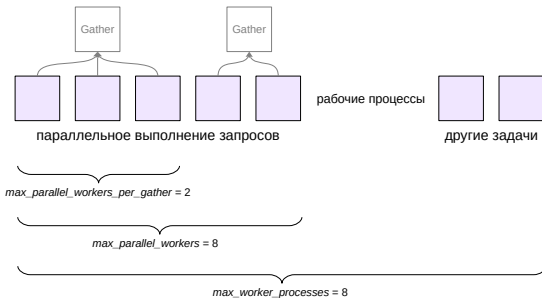
Если рабочие процессы не успевают загрузить ведущего данными, ведущий процесс подключается к выполнению того же самого параллельного плана.

Разумеется, запуск процессов и пересылка данных требуют определенных ресурсов, поэтому далеко не каждый запрос выполняется параллельно. Кроме того, даже при параллельном выполнении не все шаги плана запроса могут быть распараллелены. Часть операций может выполняться только последовательно ведущим процессом.



Размер пула процессов

Параллельным выполнением управляет довольно много параметров.



Вообще механизм рабочих процессов используется не только для параллельного выполнения запросов. Например, рабочие процессы задействованы в логической репликации, ими могут пользоваться расширения. Рабочие процессы можно использовать в прикладном коде (подробнее об этом — в теме «Фоновые процессы» курса DEV2). Общее число одновременно выполняющихся рабочих процессов ограничено параметром `max_worker_processes` (по умолчанию 8).

66
iii

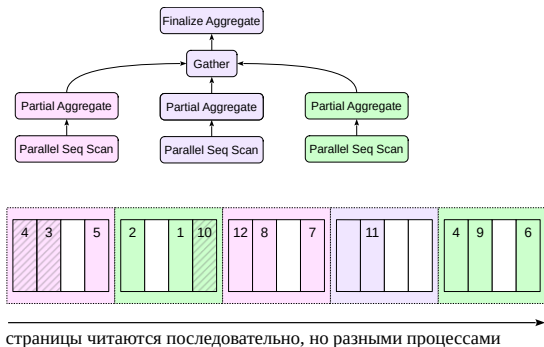
Число одновременно выполняющихся рабочих процессов, занимающихся параллельными планами, ограничено параметром `max_parallel_workers` (по умолчанию тоже 8).

Число одновременно выполняющихся рабочих процессов, обслуживающих один ведущий процесс, ограничено параметром `max_parallel_workers_per_gather` (по умолчанию 2).

Значения этих параметров следует изменить в зависимости от возможностей аппаратуры, объема данных и загруженности системы.

Parallel Seq Scan

Примером узла, выполняющегося в параллельном режиме является `Parallel Seq Scan` — «параллельное последовательное сканирование».



Название звучит противоречиво, но, тем не менее, отражает суть операции. Страницы таблицы читаются последова-

тельно, в том же самом порядке, в котором они читались бы при обычном последовательном сканировании. Однако запросы на чтение выдаются несколькими параллельно работающими процессами. Процессы синхронизируются между собой, чтобы их запросы шли в правильном порядке.

Преимущество такого сканирования проявляется в том, что параллельные процессы одновременно обрабатывают свои страницы. Чтобы эффект оправдал дополнительные накладные расходы на пересылку данных от процесса к процессу, обработка должна быть достаточно ресурсоемкой. Хорошим примером является агрегация данных, поскольку для нее необходимы ресурсы процессора, а пересылать требуется только одно итоговое число. В таком случае параллельное выполнение запроса может занять существенно меньше времени, чем последовательное.

Рассмотрим запрос с агрегатной функцией на небольшой таблице. Такой запрос выполняется последовательно:

```
EXPLAIN SELECT count(*) FROM seats;  
QUERY PLAN
```

```
-----  
Aggregate  (cost=31.76..31.77 rows=1 width=8)  
-> Seq Scan on seats  (cost=0.00..27.41 rows=1741 wid...  
(2 rows)
```

План состоит из двух узлов. Верхний узел Aggregate, в котором происходит вычисление count, получает данные от дочернего узла Seq Scan. Его начальная стоимость практически равна полной. Это означает, что узел не может выдать результат (состоящий из одной строки), пока не обработает все данные — что вполне логично.

Теперь рассмотрим такой же пример, но с большой таблицей. Здесь в плане запроса увидим параллельное последовательное сканирование:

```
EXPLAIN SELECT count(*) FROM bookings;  
QUERY PLAN
```

```
-----  
Finalize Aggregate (cost=29807.83..29807.84 rows=1 wid...  
-> Gather (cost=29807.61..29807.82 rows=2 width=8)  
    Workers Planned: 2  
    -> Partial Aggregate (cost=28807.61..28807.62 ro...  
        -> Parallel Seq Scan on bookings (cost=0.00....  
            (5 rows)
```

Все, что находится ниже узла Gather — параллельная часть плана. Она выполняется в каждом из рабочих процессов (которых запланировано два) и, возможно, в ведущем процессе.

Узел Gather и все узлы выше выполняются только в ведущем процессе. Это последовательная часть плана.

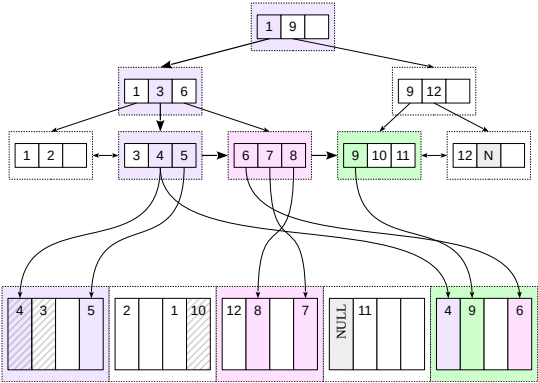


Не каждый запрос может выполняться в параллельном режиме. Например, не распараллеливаются запросы, изменяющие или блокирующие данные; запросы, содержащиеся в функциях, которые вызываются из распараллеленного запроса и другие варианты. Часть из этих ограничений может быть снята в следующих версиях PostgreSQL.

Parallel Index (Only) Scan

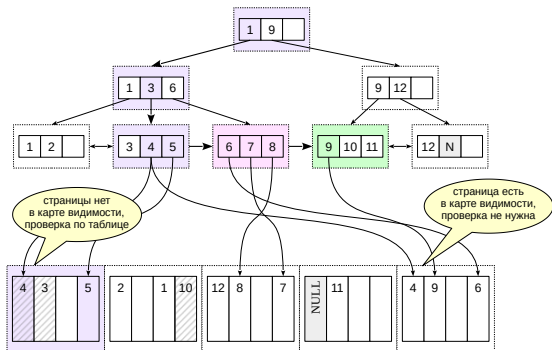
Индексный доступ тоже может выполняться в параллельном режиме. Это происходит в два этапа. Сначала ведущий процесс спускается от корня дерева к листовой странице. Затем рабочие процессы выполняют параллельное чтение листовых страниц индекса, двигаясь по списку.

Процесс, прочитавший индексную страницу, выполняет и чтение необходимых табличных страниц. При этом может получиться так, что одну и ту же табличную страницу прочитают несколько процессов (этот пример показан на рисунке: последняя табличная страница содержит строки, ссылки на которые ведут от нескольких индексных страниц, прочитанных разными процессами). Конечно, сама страница будет находиться в буферном кеше в одном экземпляре.



Параллельное сканирование только индекса происходит точно так же, как и при обычном индексном сканировании: ведущий процесс спускается от корня к листовой странице, а затем рабочие процессы параллельно сканируют листовые страницы индекса, обращаясь при необходимости к соответствующим страницам таблицы для проверки видимости.

70
iii



В качестве примера параллельного сканирования индекса найдем количество бронирований, сделанных за последний месяц:

```
EXPLAIN (costs off)
SELECT count(*)
FROM bookings
WHERE book_date > '2026-02-01 03:00:00+03';
```

QUERY PLAN

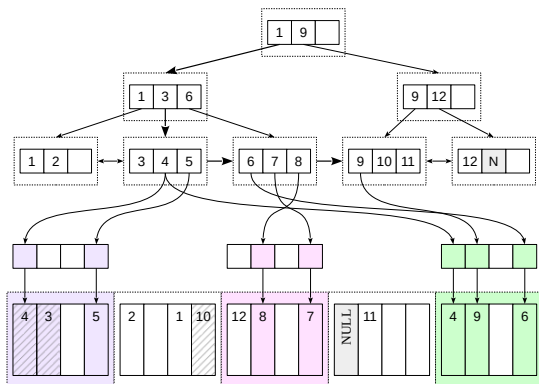
```
-----
Finalize Aggregate
  -> Gather
        Workers Planned: 2
        -> Partial Aggregate
              -> Parallel Index Only Scan using bookings_bo...
                  Index Cond: (book_date > '2026-02-01 03:0...
(6 rows)
```

Аналогичный план мы уже видели при параллельном последовательном сканировании, но теперь данные читаются с помощью индекса узлом Parallel Index Only Scan.

Parallel Bitmap Heap Scan

71
iii

Сканирование по битовой карте тоже может выполняться параллельно.



Первый этап — сканирование индекса и построение битовой карты — всегда выполняется последовательно ведущим процессом.

А второй этап — сканирование таблицы — выполняется рабочими процессами параллельно, как при параллельном последовательном сканировании.

Сколько бронирований сделано за последние три месяца на сумму до 5 тысяч ₽?

```
EXPLAIN (costs off)
SELECT count(*)
FROM bookings
WHERE total_amount < 5000
AND book_date > '2025-12-01 03:00:00+03';
```

```
-----  
Finalize Aggregate  
  -> Gather  
      Workers Planned: 2  
      -> Partial Aggregate  
          -> Parallel Bitmap Heap Scan on bookings  
              Recheck Cond: (total_amount < '5000'::num...  
              Filter: (book_date > '2025-12-01 03:00:00...  
          -> Bitmap Index Scan on bookings_total_am...  
              Index Cond: (total_amount < '5000'::n...  
  
(9 rows)
```

Ведущий процесс строит битовую карту в узле Bitmap Index Scan. Сканирование таблицы по построенной карте выполняется параллельно в узле Parallel Bitmap Heap Scan.

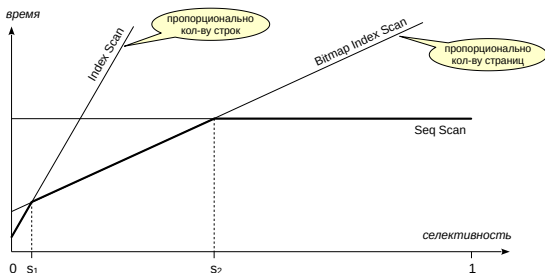
Удалим созданные индексы:

```
DROP INDEX bookings_book_date_idx;  
DROP INDEX bookings_total_amount_idx;  
DROP INDEX tickets_ticket_no_book_ref_idx;
```

6. Сравнение эффективности методов доступа

Индексное сканирование лучше всего работает при очень высокой селективности, когда по индексу выбирается одно или несколько значений.

При средней селективности обычно лучше всего показывает себя сканирование по битовой карте. Хотя этот способ требует предварительного построения битовой карты, он выигрывает у индексного сканирования, поскольку обходится без повторных чтений одних и тех же страниц (если только данные в таблице не расположены физически

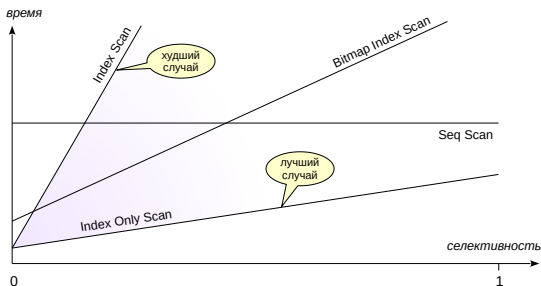


в нужном порядке, что бывает нечасто). В худшем случае производительность индексного доступа пропорциональна числу выбираемых строк, а сканирования по битовой карте — числу страниц.

При низкой селективности лучше всего работает последовательное сканирование: если надо выбрать все или почти все табличные строки, обращение к индексным страницам только увеличивает накладные расходы. Этот эффект усиливается в случае вращающихся дисков, где скорость произвольного чтения существенно ниже скорости чтения последовательно расположенных страниц.

Значение селективности, при котором становится выгодно переключиться на другой метод доступа, сильно зависит от конкретной таблицы и конкретного индекса. Планировщик учитывает множество параметров, чтобы выбрать наиболее подходящий способ.

Еще одно замечание: при индексном доступе результат всегда возвращается отсортированным, что в ряде случаев увеличивает привлекательность такого доступа даже при низкой селективности.



Эффективность сканирования только индекса сильно зависит от актуальности карты видимости и от того, сколько страниц действительно содержат только актуальные версии строк.

В лучшем случае этот метод доступа может быть эффективнее последовательного сканирования даже при низкой селективности (если индекс меньше, чем таблица, и особенно на SSD-накопителях). В худшем случае, когда требуется проверять видимость каждой строки, метод вырождается в обычное индексное сканирование. Поэтому планировщику приходится учитывать состояние карты видимости: при неблагоприятном прогнозе исключительно индексное сканирование не применяется из-за опасности получить замедление вместо ускорения.

IV. Сортировка и группировка

1. Сортировка

Индексный доступ (при использовании В-дерева) позволяет сразу же получить набор строк, отсортированный по ключам индексирования:

```
EXPLAIN (costs off)
SELECT * FROM flights
ORDER BY flight_id;

               QUERY PLAN
-----
Index Scan using flights_pkey on flights
(1 row)
```

Но если попросить сервер отсортировать данные по столбцу без индекса, потребуется два отдельных шага: получение данных и сортировка.

```
EXPLAIN (costs off)
SELECT * FROM flights
ORDER BY status;

               QUERY PLAN
-----
Sort
  Sort Key: status
  -> Seq Scan on flights
(3 rows)
```

Сортировка в памяти

В идеальном случае набор строк, подлежащий сортировке, целиком помещается в память, ограниченную параметром *work_mem*. В этом случае все строки просто сортируются алгоритмом быстрой сортировки (Sort Method: quicksort) и результат возвращается родительскому узлу. В той же строке плана указан объем использованной памяти:

```
EXPLAIN (analyze, timing off, summary off, buffers off)
SELECT *
FROM seats
ORDER BY seat_no;
```

QUERY PLAN

```
-----
Sort (cost=121.13..125.48 rows=1741 width=15) (actual ...
  Sort Key: seat_no
  Sort Method: quicksort Memory: 103kB
    -> Seq Scan on seats (cost=0.00..27.41 rows=1741 wid...
(4 rows)
```

Чтобы начать выдавать данные, узлу Sort нужно полностью отсортировать набор строк. Поэтому начальная стоимость узла включает в себя полную стоимость чтения таблицы. В общем случае сложность быстрой сортировки равна $O(M \log M)$, где M — число строк в исходном наборе.

Если нужно получить первые N строк отсортированного набора, планировщик может переключиться на частичную сортировку:

```
EXPLAIN (analyze, timing off, summary off, buffers off)
SELECT *
FROM seats
ORDER BY seat_no
LIMIT 100;
```

```

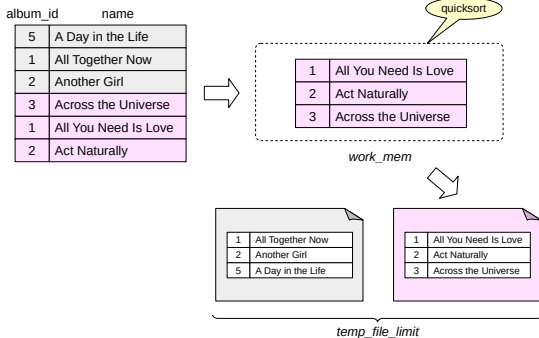
Limit (cost=93.95..94.20 rows=100 width=15) (actual ro...
  -> Sort (cost=93.95..98.30 rows=1741 width=15) (actu...
      Sort Key: seat_no
      Sort Method: top-N heapsort Memory: 30kB
      -> Seq Scan on seats (cost=0.00..27.41 rows=1741...
(5 rows)
    
```

Обратите внимание, что стоимость запроса снизилась и для сортировки потребовалось меньше памяти. В общем случае сложность алгоритма top-N heapsort ниже, чем для быстрой сортировки — она равна $O(M \log N)$.

Внешняя сортировка

Если набор строк велик, он не поместится в память целиком. Тогда применяется алгоритм внешней сортировки.

Набор строк читается в память, пока есть возможность, затем сортируется и записывается во временный файл.

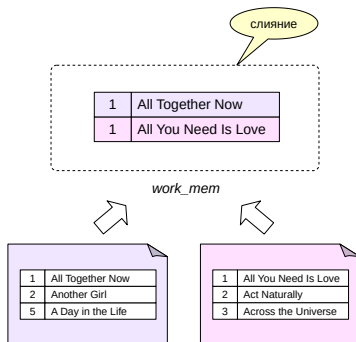


Эта процедура повторяется столько раз, сколько необходимо, чтобы записать все данные в файлы, каждый из которых по отдельности отсортирован.

Напомним, что общий размер временных файлов одного процесса (не включая временные таблицы) ограничен значением параметра *temp_file_limit*.

Далее несколько файлов (два или более) соединяются с сохранением порядка строк. Эта операция называется слиянием.

Для соединения не требуется много места в памяти, достаточно разместить по одной строке из каждого файла (как в примере на рисунке ниже). Среди этих строк выбирается минимальная (максимальная) и возвращается как часть результата, а на ее место читается новая строка из того же файла. На практике строки читаются не по одной, а пакетами, чтобы ускорить ввод-вывод.



Если оперативной памяти недостаточно, чтобы соединить сразу все файлы, сначала соединяется часть файлов и ре-

зультат записывается в новый временный файл, затем происходит соединение этих новых файлов и так далее.

79
iv

```
EXPLAIN (analyze, timing off, summary off)
SELECT *
FROM bookings
ORDER BY total_amount;
```

QUERY PLAN

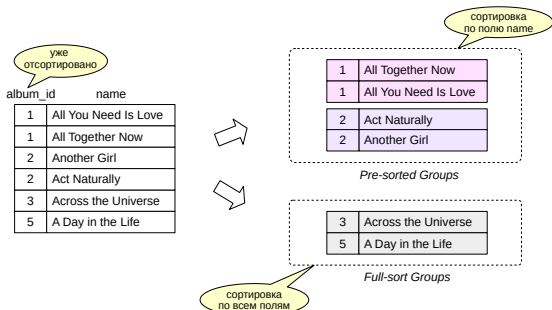
```
-----
Sort  (cost=406880.90..413097.68 rows=2486710 width=21)...
  Sort Key: total_amount
  Sort Method: external merge  Disk: 78176kB
  Buffers: shared read=15856, temp read=19539
            written=19578
  -> Seq Scan on bookings (cost=0.00..40723.10 rows=24...
        Buffers: shared read=15856
Planning:
  Buffers: shared hit=37 read=4
(8 rows)
```

Обратите внимание на то, что узел Sort записывает и читает временные данные (temp read и written).

Инкрементальная сортировка

Если набор данных требуется отсортировать по ключам $K_1 \dots K_m \dots K_n$ и при этом известно, что набор уже отсортирован по первым m ключам, можно не пересортировывать все данные заново. Достаточно разбить набор данных на группы, имеющие одинаковые значения начальных ключей $K_1 \dots K_m$ (значения таких групп следуют друг за другом), и затем отсортировать отдельно каждую из групп по оставшимся ключам $K_{m+1} \dots K_n$. Такой способ называется инкрементальной сортировкой.

Алгоритм обрабатывает относительно крупные группы строк отдельно, а небольшие объединяет и сортирует полностью. Сортируемые наборы данных уменьшаются, так что



уменьшается и требование к доступному объему памяти. Но увеличение количества наборов приводит к росту накладных расходов.

Наборы могут обрабатываться как в оперативной памяти, так и на диске — если не хватает объема *work_mem*. Инкрементальная сортировка позволяет выдавать результаты уже после обработки первой группы, не дожидаясь сортировки всего набора.

Отключить использование инкрементальной сортировки можно параметром *enable_incremental_sort*.

Для иллюстрации создадим индекс на таблице *bookings*:

```
CREATE INDEX ON bookings(total_amount);
```

Посмотрим на пример, увеличив значение *work_mem*:

```
SET work_mem = '8 MB';  
EXPLAIN (analyze, costs off, timing off, buffers off)  
SELECT * FROM bookings  
ORDER BY total_amount, book_date;
```

```
-----
Incremental Sort (actual rows=2486710.00 loops=1)
  Sort Key: total_amount, book_date
  Presorted Key: total_amount
  Full-sort Groups: 3132  Sort Method: quicksort
  Average Memory: 27kB  Peak Memory: 27kB
  Pre-sorted Groups: 2449  Sort Method: quicksort
  Average Memory: 6442kB  Peak Memory: 6464kB
  -> Index Scan using bookings_total_amount_idx on book...
    Index Searches: 1
  Planning Time: 0.095 ms
  Execution Time: 938.554 ms
(11 rows)
```

Здесь данные, полученные из таблицы `bookings` по только что созданному индексу `bookings_total_amount_idx`, уже отсортированы по столбцу `total_amount` (Presorted Key), поэтому остается доупорядочить строки по столбцу `book_date`.

Строка `Pre-sorted Groups` относится к крупным группам, которые досортировывались по столбцу `book_date`, а строка `Full-sort Groups` — к небольшим группам, которые были объединены и отсортированы полностью.

В примере все группы поместились в выделенную память и применялась быстрая сортировка.

Теперь уменьшим `work_mem`:

```
SET work_mem = '128 kB';
```

И повторим запрос:

```
EXPLAIN (analyze, costs off, timing off, buffers off)
SELECT *
FROM bookings
ORDER BY total_amount, book_date;
```

iv

```
-----
Incremental Sort (actual rows=2486710.00 loops=1)
  Sort Key: total_amount, book_date
  Presorted Key: total_amount
  Full-sort Groups: 3132  Sort Method: quicksort
  Average Memory: 27kB  Peak Memory: 27kB
  Pre-sorted Groups: 2449  Sort Method: external merge
  Average Disk: 2662kB  Peak Disk: 2672kB
  -> Index Scan using bookings_total_amount_idx on book...
      Index Searches: 1
  Planning Time: 0.038 ms
  Execution Time: 920.843 ms
(11 rows)
```

Теперь, при нехватке памяти, для некоторых крупных групп пришлось применить внешнюю сортировку с использованием временных файлов (Pre-sorted Groups ... external merge).

Восстановим значение параметра и удалим индекс:

```
RESET work_mem;
DROP INDEX bookings_total_amount_idx;
```

Построение В-дерева

При построении индекса сервер мог бы добавлять записи в пустой индекс по одной, обрабатывая последовательно строки таблицы. Но такой способ крайне неэффективен.

Поэтому при создании и перестроении индекса используется сортировка: все строки таблицы сортируются, и формируются записи, которые раскладываются по листовым индексным страницам. Затем достраиваются верхние уровни

дерева, состоящие из ссылок на элементы страниц нижележащего уровня, до тех пор, пока на очередном уровне не получится одна страница — она и будет корнем дерева.

Сортировка устроена точно так же, как рассматривалось выше. Однако размер памяти ограничен не *work_mem*, а *maintenance_work_mem*, поскольку операция создания индекса не слишком частая и имеет смысл выделить для нее больше памяти.

Построение индекса может выполняться параллельно. Количество рабочих процессов выбирается так же, как и при выполнении параллельных запросов (в зависимости от размера таблицы), но ограничено значением параметра *max_parallel_maintenance_workers*.

2. Группировка

Применение группировки

В SQL-запросе набор строк можно сгруппировать с помощью предложения GROUP BY, в сочетании с которым обычно используются агрегатные функции, обрабатывающие все строки группы. Похожую возможность для оконных функций дает предложение PARTITION BY.

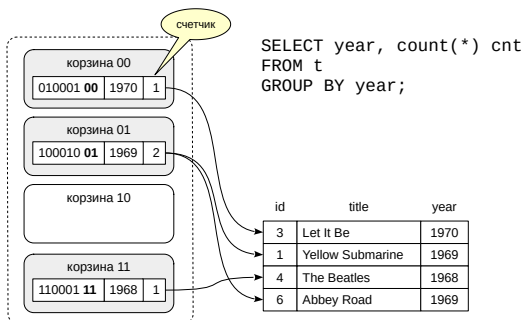
Группировка может выполняться и неявно при устранении дубликатов в запросе со словом DISTINCT или с командами обработки двух наборов строк (UNION, INTERSECT, EXCEPT без слова ALL), которые удаляют дубликаты.

Сервер выбирает один из двух способов группировки — хешированием или сортировкой, — учитывая имеющиеся ресурсы, возможность получить отсортированные данные для ключа группировки и другие факторы.

Группировка хешированием

Идея хеширования состоит в том, что значения любого типа данных равномерно распределяются по ограниченному количеству корзин хеш-таблицы с помощью функции хеширования. Если хеш-таблица имеет достаточный размер, чтобы в одну корзину в среднем попадало одно значение (хеш-код), поиск значения в хеш-таблице выполняется за константное время. Для этого:

1. Вычисляется хеш-функция от заданного значения.
2. По нескольким битам полученного хеш-кода определяется номер корзины.
3. Корзина просматривается в поисках хеш-кода. Если данные распределены неравномерно, в одну корзину могут попасть много значений. В этом случае эффективность поиска будет страдать.



Узел плана, отвечающий за агрегацию методом хеширования, называется HashAggregate:

```
EXPLAIN
SELECT airplane_code, count(*)
FROM seats
GROUP BY airplane_code;
```

QUERY PLAN

```
-----
HashAggregate (cost=36.12..36.20 rows=8 width=12)
  Group Key: airplane_code
    -> Seq Scan on seats (cost=0.00..27.41 rows=1741 wid...
(3 rows)
```

Обратите внимание на начальную стоимость: результаты начинают выдаваться только после построения хеш-таблицы по всем строкам.

Выполнив запрос, можно увидеть объем памяти, выделенной под хеш-таблицу:

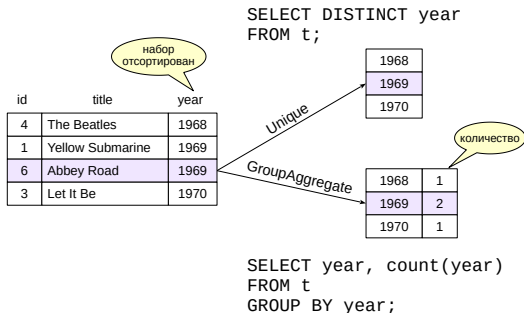
```
EXPLAIN (analyze, timing off, costs off, buffers off)
SELECT airplane_code, count(*)
FROM seats
GROUP BY airplane_code;
```

QUERY PLAN

```
-----
HashAggregate (actual rows=8.00 loops=1)
  Group Key: airplane_code
    Batches: 1 Memory Usage: 32kB
    -> Seq Scan on seats (actual rows=1741.00 loops=1)
Planning Time: 0.028 ms
Execution Time: 0.157 ms
(6 rows)
```

Группировка сортировкой

Исключение дубликатов и группировка могут выполняться не только путем хеширования, но и с помощью предварительной сортировки набора данных по полям группировки.



Узел Unique устраняет дубликаты, добавляя к результату очередное значение, если оно не совпадает с предыдущим.

Узел GroupAggregate возвращает агрегированные данные. В примере на рисунке узел строит список различных значений поля year, подсчитывая, сколько раз встретилось каждое из них в наборе данных.

Как было рассмотрено выше, упорядоченный набор строк может быть получен с помощью индексного доступа или в результате сортировки в узле Sort. В первом случае узлы Unique и GroupAggregate обычно более выгодны, чем HashAggregate, поскольку выполняются быстро и не требуют много памяти. Однако во втором случае построение хеш-таблицы, как правило, эффективнее сортировки.

В отличие от HashAggregate, оба узла Unique и GroupAggregate возвращают упорядоченные данные.

Пример плана с узлом GroupAggregate, в котором происходит группировка отсортированного набора:

```
EXPLAIN (costs off)
SELECT ticket_no, count(ticket_no)
FROM segments
GROUP BY ticket_no;
```

QUERY PLAN

```
-----
GroupAggregate
  Group Key: ticket_no
    -> Index Only Scan using segments_pkey on segments
(3 rows)
```

В этом запросе сортировку набора данных обеспечивает индексный доступ.

Запрос с DISTINCT использует для устранения дубликатов узел Unique:

```
EXPLAIN (costs off)
SELECT DISTINCT ticket_no
FROM segments
ORDER BY ticket_no;
```

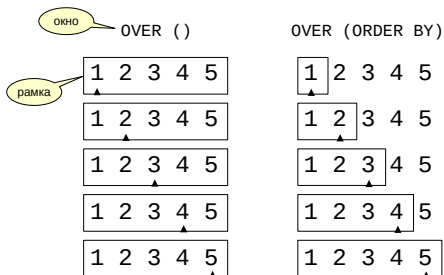
QUERY PLAN

```
-----
Unique
  -> Index Only Scan using segments_pkey on segments
(2 rows)
```

Поскольку узел возвращает результаты, упорядоченные так же, как требует предложение ORDER BY, дополнительная сортировка не требуется.

В оконных функциях

В отличие от обычных агрегатных функций, оконные функции вычисляются для каждой строки набора, не уменьшая



его размер. После имени функции в предложении `OVER` указывается окно: правило, определяющее для каждой строки результата запроса выборку, которую должна обработать оконная функция. Такая выборка называется рамкой.

Если окно указано как `OVER()`, то рамка одинакова для каждой строки и состоит из всех строк результата.

Если же в определении окна указано предложение `ORDER BY`, упорядочивающее строки выборки, рамка окна по умолчанию охватывает строки от самой первой до текущей.

Посмотрим план запроса, вычисляющего сумму бронирований нарастающим итогом:

EXPLAIN

```
SELECT *, sum(total_amount) OVER (ORDER BY book_date)
FROM bookings;
```

QUERY PLAN

```
-----
WindowAgg (cost=406880.92..450398.33 rows=2486710 widt...
  Window: w1 AS (ORDER BY book_date)
  -> Sort (cost=406880.90..413097.68 rows=2486710 widt...
    Sort Key: book_date
    -> Seq Scan on bookings (cost=0.00..40723.10 row...
      (5 rows)
```

Оконная функция вычисляется в узле WindowAgg, который получает отсортированные данные от дочернего узла Sort. При этом может использоваться любой метод сортировки, рассмотренный выше. Например, быстрая сортировка:

89
iv

```
EXPLAIN (analyze, timing off, summary off, buffers off)
SELECT *, count(*) OVER (ORDER BY seat_no)
FROM seats;
```

QUERY PLAN

```
-----
WindowAgg (cost=121.18..151.59 rows=1741 width=23) (ac...
  Window: w1 AS (ORDER BY seat_no)
  Storage: Memory Maximum Storage: 17kB
  -> Sort (cost=121.13..125.48 rows=1741 width=15) (ac...
    Sort Key: seat_no
    Sort Method: quicksort Memory: 103kB
    -> Seq Scan on seats (cost=0.00..27.41 rows=1741...
(7 rows)
```

Или внешняя:

```
EXPLAIN (analyze, timing off, summary off, buffers off)
SELECT *, sum(total_amount) OVER (ORDER BY book_date)
FROM bookings;
```

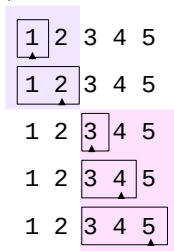
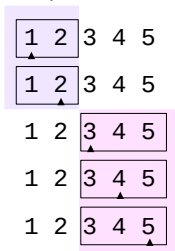
QUERY PLAN

```
-----
WindowAgg (cost=406880.92..450398.33 rows=2486710 widt...
  Window: w1 AS (ORDER BY book_date)
  Storage: Memory Maximum Storage: 17kB
  -> Sort (cost=406880.90..413097.68 rows=2486710 widt...
    Sort Key: book_date
    Sort Method: external merge Disk: 75760kB
    -> Seq Scan on bookings (cost=0.00..40723.10 row...
(7 rows)
```

Для узлов, использующих внутреннее хранилище строк (WindowAgg, CTE Scan, Recursive Union и др.), строка Storage показывает максимальное количество выделенной памяти и ее тип: Memory (хватило оперативной памяти) или Disk (потребовались временные файлы).

OVER (PARTITION BY)

OVER (PARTITION BY ORDER BY)



В оконных функциях используется и группировка. Если определение окна включает конструкцию `PARTITION BY`, оконная функция вычисляется на основе групп строк (аналогично группировке `GROUP BY`).

Для каждой группы строк границы рамки меняются в пределах этой группы. Если при этом нет других указаний, для всех строк одной группы рамка будет совпадать с этой группой.

Как и в примере с сортировкой, указание `ORDER BY` упорядочивает строки в каждой группе; в этом случае рамка охватывает строки от первой до текущей.

В отличие от обычной группировки `GROUP BY`, группировка в определении окна всегда реализуется с помощью сортировки: план строится так, чтобы на вход узлу `WindowAgg` пришел набор строк, отсортированный сначала по ключам группировки `PARTITION BY`, а затем — по ключам сортировки `ORDER BY`. Хеширование не применяется даже в отсутствие конструкции `ORDER BY`.

V. Способы соединения

1. Что такое способы соединения

Мало получать данные с помощью рассмотренных методов доступа, надо еще и уметь соединять их.

Способы соединения представляют собой алгоритмы для соединения двух наборов строк. С их помощью реализуются не только соединения (JOIN), но и другие конструкции языка SQL, например, EXISTS. Не стоит путать одно с другим: *SQL-соединения* – это логические операции над двумя множествами; *способы соединения* – это возможные реализации таких соединений, учитывающие эффективность.

Часто говорят, что соединяются таблицы. Это удобное упрощение, но в общем случае соединяются *наборы строк*. Эти наборы могут быть получены как непосредственно из таблицы (с помощью одного из методов доступа), так и, например, в результате соединения других наборов строк.

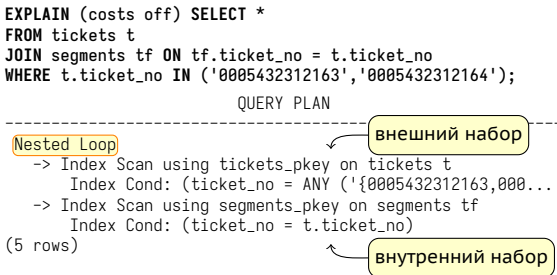
Наборы строк всегда соединяются попарно. Порядок, в котором соединяются наборы, в большинстве случаев не влияет на результат запроса (например, `(a join b) join c` или `(b join c) join a`), но очень важен с точки зрения производительности. Как мы увидим дальше, важен и порядок, в котором соединяются два набора строк (`a join b` или `b join a`).

2. Соединение вложенными циклами

Начнем с соединения вложенными циклами. Его алгоритм таков: для каждой строки одного из наборов перебираем и возвращаем соответствующие ему строки второго набора. По сути, это два вложенных цикла, отсюда и название способа.

Заметим, что ко второму (внутреннему) набору мы будем обращаться столько раз, сколько строк в первом (внешнем) наборе. Если нет эффективного метода доступа для поиска соответствующих строк во втором наборе (то есть, попросту говоря, индекса на таблице), придется неоднократно перебирать большое количество строк, не относящихся к делу. Очевидно, это будет не лучший выбор, хотя для небольших наборов алгоритм и в этом случае может оказаться достаточно эффективным.

Так выглядит план для соединения вложенным циклом, которое оптимизатор обычно предпочитает для небольших выборок (смотрим перелеты, включенные в два билета). Мы видим узел Nested Loop с двумя дочерними узлами:



Это *параметризованное* соединение. Для каждой строки внешнего набора (узел Index Scan по таблице билетов) выбираются строки из внутреннего набора (узел Index Scan по таблице перелетов), отвечающие условию соединения. На каждой итерации внешнего цикла условие индексного доступа для внутреннего набора имеет вид `ticket_no = константа`.

Процесс повторяется до тех пор, пока внешний набор не исчерпает все строки.

Вот еще один пример соединения вложенным циклом. Здесь мы выводим самолеты, способные обслуживать перелеты заданной протяженности:

```
EXPLAIN (costs off) SELECT *  
FROM ( VALUES (1000), (10000) ) d(range)  
JOIN airplanes a ON a.range >= d.range;
```

QUERY PLAN

```
-----  
Nested Loop  
  Join Filter: (m1.range >= "*VALUES".column1)  
    -> Seq Scan on airplanes_data m1  
    -> Materialize  
      -> Values Scan on "*VALUES*"  
(5 rows)
```

Это *непараметризованное* соединение: для каждой строки внешнего набора читается весь внутренний набор (про узел Materialize мы будем говорить позже; в данном случае это просто значения из VALUES) и для полученных пар строк проверяется условие Join Filter. По сути, это декартово произведение двух наборов строк с фильтрацией.

Обратите внимание, что соединение вложенным циклом позволяет соединять строки по любому условию, не обязательно по равенству значений.

Мемоизация

Если внутренний набор сканируется много раз с повторяющимися значениями параметров, иногда имеет смысл закешировать результат, чтобы не читать много раз одни и те же данные. Эта операция называется мемоизацией. Она выполняется в узле Memoize, который встраивается между Nested Loop и узлом, поставляющим данные.

(Если планировщик ошибается в расчетах, мемоизацию можно отключить, установив `enable_memoize = off`.)

Посмотрим план запроса, в котором соединяются таблицы маршрутов и аэропортов:

```
EXPLAIN (analyze, costs off, timing off, buffers off)
SELECT * FROM routes r
  JOIN airports a ON a.airport_code = r.departure_airport
WHERE r.departure_airport in ('BOM', 'SVO');
```

QUERY PLAN

```
Nested Loop (actual rows=108.00 loops=1)
  -> Bitmap Heap Scan on routes r (actual rows=108.00 l...
      Recheck Cond: (departure_airport = ANY
        ('{BOM,SVO}'::bpchar[]))
      Heap Blocks: exact=33
  -> Bitmap Index Scan on routes_departure_airport_...
      Index Cond: (departure_airport = ANY
        ('{BOM,SVO}'::bpchar[]))
      Index Searches: 2
  -> Memoize (actual rows=1.00 loops=108) ← кеширование
      Cache Key: r.departure_airport
      Cache Mode: logical
      Hits: 106 Misses: 2 Evictions: 0 Overflows: 0
      Memory Usage: 1kB
  -> Index Scan using airports_data_pkey on airport...
      Index Cond: (airport_code =
        r.departure_airport)
      Index Searches: 2
(18 rows)
```

↑ внутренний набор

Пример подобран таким образом, что из внутреннего набора (таблица `airports_data`) будет получено только две строки по ключу `r.departure_airport` (Cache Key), они и будут закешированы в узле Memoize.

Для кеширования используется хеш-таблица, ключом в которой служит параметр или несколько параметров, с которыми соединение обращается к внутреннему набору.

Обратите внимание:

- в первый раз нужные строки пришлось прочитать из таблицы (Misses: 2);
- все повторные обращения к данным обслуживаются кешем (Hits: 106), для этого хватило 1 КБайта памяти;
- вытеснений из кеша не было (Evictions: 0);
- строки внутреннего набора всегда умещались в выделенную память (Overflows: 0).

Вычислительная сложность

Если принять за N число строк во внешнем наборе данных, а за M — среднее число строк внутреннего набора, получаемых на одной итерации, то общая сложность соединения будет пропорциональна произведению $N \times M$.

Для непараметризованного соединения M будет в точности равно числу строк внутреннего набора; для параметризованного — M может быть значительно меньше.

Метод соединения вложенным циклом эффективен, когда внешний набор невелик, а обращение к строкам внутреннего обходится дешево. В частности, такой метод (в сочетании с индексным доступом) характерен для OLTP-запросов, в которых надо очень быстро вернуть мало строк.

Nested Loop в параллельных планах

Соединение вложенным циклом может использоваться в параллельных планах.

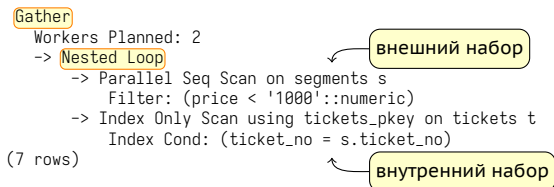
Внешний набор может сканироваться несколькими рабочими процессами параллельно. Получив строку из внешнего набора, процесс затем перебирает соответствующие ему строки внутреннего набора последовательно.

Чтобы получить очередную страницу из внешнего набора, процессы синхронизируются. К внутреннему набору данных каждый процесс обращается независимо от остальных.

Найдем билеты с недорогими перелетами:

```
EXPLAIN (costs off)
SELECT t.ticket_no, s.price
FROM tickets t
JOIN segments s ON s.ticket_no = t.ticket_no
WHERE s.price < 1000;
```

QUERY PLAN



В этом запросе используется параллельный план. Каждый из процессов читает свою часть таблицы перелетов (segments) и соединяет ее с билетами (tickets) с помощью вложенного цикла.

Однопроходное соединение

На первом этапе в памяти строится хеш-таблица по строкам первого (внутреннего) набора данных.

Строки этого набора читаются последовательно, и для каждой из них вычисляется хеш-функция от значений полей, входящих в условие соединения.

В корзину хеш-таблицы помещаются вычисленный хеш-код и все поля, которые входят в условие соединения или используются в запросе.

Размер хеш-таблицы в памяти ограничен значением $work_mem \times hash_mem_multiplier$. Наилучшая эффективность достигается, если вся хеш-таблица помещается в этот объем памяти целиком — такое соединение выполняется *за один проход*.

Поэтому планировщик обычно выбирает в качестве внутреннего меньший из двух наборов строк. Также не стоит использовать в запросе лишние поля, в том числе «звездочку», чтобы не перегружать хеш-таблицу ненужной информацией.

На втором этапе последовательно читается второй (внешний) набор строк. По мере чтения вычисляется хеш-функция от значений полей, участвующих в условии соединения. Если в соответствующей корзине хеш-таблицы обнаруживается строка с таким же хеш-кодом и со значениями полей, подходящими под условие соединения, то мы нашли пару.

98
v

Проверки одного только хеш-кода недостаточно. Во-первых, не все условия соединения, перечисленные в запросе, могут быть учтены при выполнении соединения хешированием: из-за того, что хеш-функция не сохраняет порядок значений, поддерживаются только эквисоединения. Во-вторых, возможны коллизии, при которых разные значения получают одинаковые хеш-коды. Вероятность этого мала, но исключать такой исход нельзя.

Рассмотрим пример. При низкой селективности оптимизатор предпочитает соединение хешированием:

```
EXPLAIN (costs off)
SELECT *
FROM airplanes a
      JOIN seats s ON a.airplane_code = s.airplane_code;
                        QUERY PLAN
```

```
-----
Hash Join
  Hash Cond: (s.airplane_code = m1.airplane_code)
    -> Seq Scan on seats s
    -> Hash  ← хеш-таблица
        -> Seq Scan on airplanes_data m1
(5 rows)
```

Узел Hash Join начинает работу с обращения к дочернему узлу Hash. Тот получает от своего дочернего узла (здесь — Seq Scan) весь набор строк и строит хеш-таблицу.

Затем Hash Join обращается ко второму дочернему узлу и соединяет строки, постепенно возвращая полученные результаты.

Проверим, как выполняется этот запрос:

```
EXPLAIN (analyze, costs off, timing off, buffers off)
SELECT *
FROM airplanes a
      JOIN seats s ON a.airplane_code = s.airplane_code;
```

```
-----  
Hash Join (actual rows=1741.00 loops=1)  
  Hash Cond: (s.airplane_code = m1.airplane_code)  
    -> Seq Scan on seats s (actual rows=1741.00 loops=1)  
    -> Hash (actual rows=10.00 loops=1)  
        Buckets: 1024  Batches: 1  Memory Usage: 10kB  
    -> Seq Scan on airplanes_data m1 (actual rows=10....  
Planning Time: 0.078 ms  
Execution Time: 1.311 ms  
(8 rows)
```

Хеш-таблица поместилась в память (Batches: 1). Она состояла из 1024 корзин (Buckets) и заняла 10 Кбайт (Memory Usage).

Обратите внимание, что хеш-таблица строилась по меньшему набору строк.

Двухпроходное соединение

Двухпроходное соединение применяется, когда хеш-таблица не помещается в объем памяти, ограниченный $work_mem \times hash_mem_multiplier$.

Первый (внутренний) набор строк разбивается на отдельные пакеты, используя несколько битов хеш-кода (поэтому число пакетов всегда является степенью числа 2). В идеале в каждый пакет попадает примерно одинаковое количество строк, но, если значения в строках повторяются, возможен перекос.

При планировании запроса заранее вычисляется минимально необходимое число пакетов так, чтобы хеш-таблица для каждого отдельного пакета помещалась в памяти. Это число не уменьшается, даже если оптимизатор ошибся с оценками, но при необходимости может динамически увеличиваться.

Хеш-таблица для первого пакета остается в памяти, а строки, принадлежащие другим пакетам, сбрасываются на диск во временные файлы — каждый пакет в свой файл.

Далее читается второй (внешний) набор строк. Если строка принадлежит первому пакету, она сопоставляется с хеш-таблицей в памяти, которая как раз содержит первый пакет. С другими пакетами строку сопоставлять не надо — в них не может найтись соответствие, поскольку хеш-коды заведомо будут отличаться.

Если же строка принадлежит другому пакету, она сбрасывается на диск — каждый пакет в свой файл. Таким образом, при N пакетах используются $2(N - 1)$ временных файлов.

Следует учитывать, что общий объем временных файлов одного сеанса ограничен параметром *temp_file_limit* (буферы временных таблиц в это ограничение не входят).

Далее по очереди обрабатываются все пакеты, начиная со второго. Из временного файла в хеш-таблицу считываются строки внутреннего набора, затем из другого временного файла считываются строки внешнего набора и сопоставляются с хеш-таблицей.

Процедура повторяется для всех оставшихся $N - 1$ пакетов. После обработки последнего пакета соединение завершается и временные файлы освобождаются.

Возьмем теперь большие таблицы и выведем статистику использования буферного кеша. Теперь хеш-таблица не поместилась в память и используется двухпроходное соединение, для которого потребовалось 32 пакета (Batches). Видно, что узел Hash записывает пакеты во временные файлы (temp written), а узел Hash Join и записывает, и читает (temp read и written):

```
EXPLAIN (analyze, costs off, timing off, summary off)
SELECT * FROM bookings b
JOIN tickets t ON b.book_ref = t.book_ref;
```

101
v

QUERY PLAN

```
Hash Join (actual rows=5598473.00 loops=1)
  Hash Cond: (t.book_ref = b.book_ref)
  Buffers: shared hit=15856 read=57816, temp read=60337
  written=60337
  -> Seq Scan on tickets t (actual rows=5598473.00 loop...
    Buffers: shared read=57816
  -> Hash (actual rows=2486710.00 loops=1)
    Buckets: 131072 Batches: 32 Memory Usage: 5155kB
    Buffers: shared hit=15856, temp written=12364
    -> Seq Scan on bookings b (actual rows=2486710.00...
      Buffers: shared hit=15856
Planning:
  Buffers: shared hit=34 read=14
(12 rows)
```

Сравним с таким же запросом, который выводит только одно поле:

```
EXPLAIN (analyze, costs off, timing off, summary off)
SELECT b.book_ref FROM bookings b
JOIN tickets t ON b.book_ref = t.book_ref;
```

QUERY PLAN

```
Hash Join (actual rows=5598473.00 loops=1)
  Hash Cond: (t.book_ref = b.book_ref)
  Buffers: shared hit=15950 read=57722, temp read=25850
  written=25850
  -> Seq Scan on tickets t (actual rows=5598473.00 loop...
    Buffers: shared hit=94 read=57722
  -> Hash (actual rows=2486710.00 loops=1)
    Buckets: 262144 Batches: 32 Memory Usage: 5002kB
    Buffers: shared hit=15856, temp written=7927
    -> Seq Scan on bookings b (actual rows=2486710.00...
      Buffers: shared hit=15856
Planning:
  Buffers: shared hit=18
(12 rows)
```

Расход памяти уменьшился, так как в хеш-таблице теперь только одно поле (вместо трех, составляющих строку таблицы bookings).

Таким образом, при нехватке оперативной памяти алгоритм соединения становится *двухпроходным*: каждый пакет (кроме первого) требуется записать на диск и затем прочитать повторно. Разумеется, это сказывается на эффективности соединения.

Поэтому важно, чтобы:

- в хеш-таблицу попадали только действительно нужные поля (обязанность автора запроса);
- хеш-таблица строилась по меньшему набору строк (обязанность планировщика).

Вычислительная сложность

Общая сложность соединения хешированием пропорциональна сумме числа строк в одном и другом наборах данных: $N + M$. Поэтому метод соединения хешированием гораздо эффективнее вложенного цикла при большом числе строк.

Однако, чтобы начать соединение, требуется построить хеш-таблицу: из-за этого при небольшом числе строк вложенный цикл более эффективен.

Соединение хешированием (в сочетании с полным сканированием таблиц) характерно для OLAP-запросов, в которых надо обработать большое число строк, причем общая пропускная способность важнее времени отклика.

В отличие от других способов соединения, хеш-соединение не только может участвовать в параллельных планах, но и имеет отдельный эффективный параллельный алгоритм. Он позволяет распараллеливать оба этапа соединения: и построение хеш-таблицы по первому (внутреннему) набору строк, и сопоставление с ней строк второго (внешнего) набора.

Параллельное хеш-соединение включается параметром *enable_parallel_hash* и по умолчанию разрешено.

Как и у последовательного алгоритма, у параллельного есть два варианта: однопроходный при достаточном количестве оперативной памяти и двухпроходный.

Однопроходный алгоритм используется, если хеш-таблица помещается в суммарный объем памяти, выделенный всем участвующим процессам. То есть размер хеш-таблицы ограничен значением $work_mem \times hash_mem_multiplier \times количество\ процессов$. Процессы параллельно читают первый набор строк (например, используя узел *Parallel Seq Scan*) и строят общую хеш-таблицу в разделяемой памяти, где каждый из них имеет к ней доступ.

После того как хеш-таблица полностью построена, рабочие процессы приступают к параллельному чтению второго набора и сопоставляют прочитанные ими строки с общей хеш-таблицей. Таким образом, каждый из процессов проверяет по хеш-таблице только часть данных.

Увеличим объем памяти, выделяемый процессу для построения хеш-таблицы, и выполним запрос с агрегацией:

104
v

```
SET work_mem = '32MB';  
EXPLAIN (analyze, costs off, timing off, buffers off)  
SELECT count(*)  
FROM bookings b  
JOIN tickets t ON b.book_ref = t.book_ref;  
QUERY PLAN
```

```
-----  
Finalize Aggregate (actual rows=1.00 loops=1)  
-> Gather (actual rows=3.00 loops=1)  
Workers Planned: 2  
Workers Launched: 2  
-> Partial Aggregate (actual rows=1.00 loops=3)  
-> Parallel Hash Join (actual rows=1866157.67...  
Hash Cond: (t.book_ref = b.book_ref)  
-> Parallel Seq Scan on tickets t (actual...  
-> Parallel Hash (actual rows=828903.33 l...  
Buckets: 4194304 Batches: 1 Memory  
Usage: 130112kB  
-> Parallel Seq Scan on bookings b (a...  
Planning Time: 0.110 ms  
Execution Time: 639.463 ms  
(14 rows)
```

Обратите внимание: объем выделенной памяти (Memory Usage) превышает ограничение, установленное для одного рабочего процесса ($64\text{MB} = \text{work_mem} \times \text{hash_mem_multiplier}$), но в общую память трех процессов хеш-таблица помещается. Поэтому выполняется однопроходное соединение (Batches: 1).

Если же размер хеш-таблицы превышает ограничение объема (что может выясниться и на этапе выполнения соединения), применяется **двухпроходный алгоритм**. Он существенно отличается и от двухпроходного последовательного, и от однопроходного параллельного.

Сначала рабочие процессы параллельно читают первый набор данных, разбивают его на пакеты и записывают пакеты во временные файлы. При этом каждый процесс запи-

сывает строки *во все* временные файлы, поэтому запись синхронизируется. Первый пакет тоже попадает в файл; хеш-таблица в памяти не строится.

Затем рабочие процессы параллельно читают второй набор данных и также разбивают его на пакеты и записывают в файлы. Таким образом, при N пакетах на диск записываются $2N$ временных файлов.

После этого каждый рабочий процесс выбирает себе по одному пакету. Процесс загружает выбранный пакет первого набора в хеш-таблицу в памяти. В этом алгоритме у каждого процесса своя хеш-таблица размером $work_mem \times hash_mem_multiplier$, но располагаются эти таблицы в общей памяти, то есть доступ к каждой таблице есть у всех рабочих процессов.

После заполнения хеш-таблицы процесс читает пакет второго набора и сопоставляет строки.

Когда процесс завершает обработку одного пакета, он выбирает следующий, еще не обработанный.

Когда необработанные пакеты заканчиваются, освободившийся процесс подключается к обработке одного из еще не завершенных пакетов, пользуясь тем, что все хеш-таблицы находятся в разделяемой памяти.

Несколько хеш-таблиц работают лучше, чем одна большая: так проще организовать совместную работу и меньше ресурсов тратится на синхронизацию.

Уменьшим объем памяти:

```
SET work_mem = '16MB';
```

Выполним тот же запрос с агрегацией:

106
v

```
EXPLAIN (analyze, costs off, timing off, buffers off)
SELECT count(*)
FROM bookings b
JOIN tickets t ON b.book_ref = t.book_ref;
```

QUERY PLAN

```
-----
Finalize Aggregate (actual rows=1.00 loops=1)
  -> Gather (actual rows=3.00 loops=1)
    Workers Planned: 2
    Workers Launched: 2
    -> Partial Aggregate (actual rows=1.00 loops=3)
      -> Parallel Hash Join (actual rows=1866157.67...
        Hash Cond: (t.book_ref = b.book_ref)
        -> Parallel Seq Scan on tickets t (actual...
        -> Parallel Hash (actual rows=828903.33 l...
          Buckets: 1048576 Batches: 8 Memory
          Usage: 20384kB
          -> Parallel Seq Scan on bookings b (a...
      Planning Time: 0.114 ms
      Execution Time: 637.746 ms
    (14 rows)
```

Соединение стало двухпроходным с восемью пакетами.

4. Соединение слиянием

Третий, и последний, способ соединения — соединение слиянием (Merge Join). Он основан на том, что два предварительно отсортированных набора данных нетрудно объединить в один общий — и точно так же отсортированный — набор. Похожим образом работает узел Gather Merge.

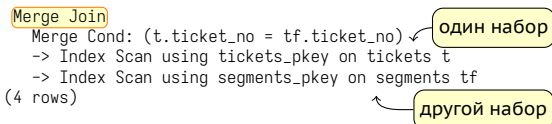
Подготовительным этапом для соединения слиянием служит сортировка обоих наборов строк. Сортировка — дорогая операция, она имеет сложность $O(N \log N)$. Но иногда этого этапа удастся избежать, если можно сразу получить отсортированные по нужным столбцам наборы строк, например за счет индексного доступа к таблице.

Общий алгоритм таков: читаем следующую строку того набора, для которого значение поля, по которому происходит соединение, меньше (один набор «догоняет» другой). Если же значения одинаковы, то читаем следующую строку второго набора.

Рассмотрим пример. Здесь дочерние узлы сразу поставляют отсортированные данные:

```
EXPLAIN (costs off) SELECT *  
FROM tickets t  
JOIN segments tf ON tf.ticket_no = t.ticket_no  
ORDER BY t.ticket_no;
```

QUERY PLAN



Важно, что алгоритм слияния возвращает результат соединения в отсортированном виде. В частности, полученный набор строк можно использовать для следующего соединения слиянием без дополнительной сортировки. Или, как в примере, сразу вернуть результат соединения клиенту.

В следующем примере с двумя соединениями слиянием один узел Merge Join получает отсортированный набор от другого узла Merge Join:

```
EXPLAIN (costs off)  
SELECT t.ticket_no, bp.flight_id, bp.seat_no  
FROM tickets t  
JOIN segments tf ON t.ticket_no = tf.ticket_no  
JOIN boarding_passes bp ON bp.ticket_no = tf.ticket_no  
AND bp.flight_id = tf.flight_id  
ORDER BY t.ticket_no;
```

v

```

Merge Join
Merge Cond: ((t.ticket_no = tf.ticket_no) AND (bp.fli...
-> Merge Join
    Merge Cond: (bp.ticket_no = t.ticket_no)
    -> Index Scan using boarding_passes_pkey on board...
    -> Index Only Scan using tickets_pkey on tickets t
-> Index Only Scan using segments_pkey on segments tf
(7 rows)

```

Здесь соединяются перелеты (segments) и посадочные талоны (boarding_passes), и с этим, уже отсортированным по номерам билетов, набором строк соединяются билеты (tickets).

Вычислительная сложность

В случае, когда не требуется сортировать данные, общая сложность соединения слиянием пропорциональна сумме чисел строк в обоих наборах данных: $N + M$. Но, в отличие от соединения хешированием, здесь не требуются накладные расходы на построение хеш-таблицы. Поэтому соединение слиянием может успешно применяться как в OLTP, так и в OLAP-запросах.

Однако если сортировка нужна, стоимость становится пропорциональной $N \log N$, и на больших объемах такой способ скорее всего проигрывает соединению хешированием.

Merge Join в параллельных планах

Соединение слиянием может использоваться в параллельных планах.

Так же, как и при соединении вложенным циклом, сканирование одного набора строк выполняется рабочими процессами параллельно, но другой набор строк каждый рабочий процесс читает полностью самостоятельно — от начала и до того момента, как станет понятно, что соответствий больше не будет.

Поэтому при соединении больших объемов строк в параллельных планах гораздо чаще используется соединение хешированием, имеющее эффективный параллельный алгоритм.

Для демонстрации нам потребуется индекс, позволяющий уменьшить стоимость соединения слиянием за счет получения строк в необходимом порядке:

```
CREATE INDEX ON tickets(book_ref);
```

Вот пример параллельного плана, в котором используется соединение слиянием:

```
EXPLAIN (costs off) SELECT count(*)  
FROM bookings b  
  JOIN (  
    SELECT book_ref FROM tickets GROUP BY book_ref  
  ) t ON b.book_ref = t.book_ref;  
          QUERY PLAN
```

```
-----  
Finalize Aggregate  
  -> Gather  
      Workers Planned: 2  
      -> Partial Aggregate  
          -> Merge Join  
              Merge Cond: (b.book_ref = tickets.book_ref)  
                  -> Parallel Index Only Scan using booking...  
                      -> Group  
                          Group Key: tickets.book_ref  
                              -> Index Only Scan using tickets_book...  
  
(10 rows)
```

110 Здесь индекс на таблице бронирований bookings_pkey
V сканируется параллельно, а результат группировки из под-
запроса t — каждым процессом полностью.

```
DROP INDEX tickets_book_ref_idx;
```

```
DROP INDEX
```

VI. Статистика

1. Базовая статистика

Базовая статистика собирается на уровне всей таблицы и на уровне отдельных столбцов.

Статистика уровня таблицы

К статистике таблицы относится информация о размере объекта (поля `reltuples` и `relpages` в таблице `pg_class`). Поскольку такая статистика крайне важна, она обновляется некоторыми DDL-операциями (`CREATE INDEX`, `CREATE TABLE AS SELECT`) и уточняется при очистке и анализе.



Кроме того, планировщик масштабирует количество строк в соответствии с отклонением реального размера файла данных от значения `relpages`.

При анализе просматривается случайная выборка строк. Математически установлено, что размер выборки, обеспечивающий хорошую точность оценок, практически не зависит от размера таблицы. В качестве размера выборки используется ориентир статистики, заданный параметром `default_statistics_target` (значение по умолчанию 100), умноженный на 300:

```

112 SELECT current_setting('default_statistics_target')::int
vi    * 300 AS rows_in_sample;
      rows_in_sample
-----
              30000
(1 row)

```

Поэтому для более или менее крупных таблиц оценки не будут точными. Но статистика и не должна быть *абсолютно* точной, чтобы планировщик мог выбрать приемлемый план; часто достаточно попадания в порядок.

Для примера возьмем простой запрос без предикатов:

```

EXPLAIN SELECT * FROM flights;
              QUERY PLAN
-----
Seq Scan on flights
  (cost=0.00..744.71 rows=37671 width=51)
(2 rows)

```

Оптимизатор получает оценку из pg_class:

```

SELECT reltuples, relpages FROM pg_class
WHERE relname = 'flights';
 reltuples | relpages
-----+-----
    37671 |     368
(1 row)

```

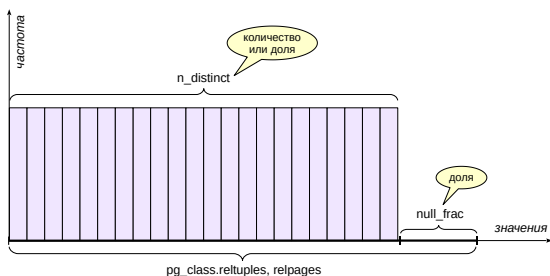
В данном случае оценка совпадает с точным значением:

```

SELECT count(*) FROM flights;
 count
-----
    37671
(1 row)

```

Вся остальная статистика собирается при анализе таблицы отдельно для каждого столбца. Статистика уровня столбцов хранится в таблице `pg_statistic`, но смотреть проще в представление `pg_stats`, которое показывает информацию в более удобном виде.



Доля неопределенных значений

Поле `null_frac` содержит долю строк с неопределенными значениями в столбце (от 0 до 1).

Часть рейсов еще не отправились, поэтому время вылета для них не определено:

```
EXPLAIN SELECT *
FROM flights WHERE actual_departure IS NULL;
QUERY PLAN
```

```
-----
Seq Scan on flights
  (cost=0.00..744.71 rows=10781 width=51)
  Filter: (actual_departure IS NULL)
(3 rows)
```

114 Оценка оптимизатора получена как общее число строк,
vi умноженное на долю NULL-значений:

```
SELECT round(37671 * null_frac)
FROM pg_stats
WHERE tablename = 'flights'
      AND attname = 'actual_departure';
round
-----
 10781
(1 row)
```

Оценка немного отличается от точного значения:

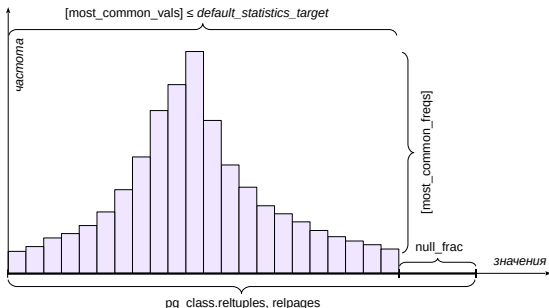
```
SELECT count(*) FROM flights
WHERE actual_departure IS NULL;
count
-----
 10806
(1 row)
```

Число уникальных значений

Поле `n_distinct` хранит количество уникальных значений в столбце. Если это значение отрицательно, то его модуль показывает долю уникальных значений. Например, `-1` означает, что все значения уникальны (типичный случай для первичного ключа).

Наиболее частые значения

Если бы данные всегда были распределены равномерно, то есть все значения встречались бы с одинаковой частотой, этой информации было бы почти достаточно (нужен еще минимум и максимум).



Но в реальности неравномерные распределения встречаются очень часто. Поэтому собирается еще и следующая информация:

- массив наиболее частых значений — поле `most_common_vals`;
- массив частот этих значений — поле `most_common_freqs`.

Частота непосредственно служит оценкой селективности для поиска конкретного значения. Такой способ прекрасно работает, если число различных значений не очень велико. Максимальный размер каждого из массивов ограничен параметром `default_statistics_target`. Это значение можно переопределять на уровне отдельного столбца; в таком случае размер анализируемой выборки будет определяться по максимальному значению для таблицы.

Чтобы не увеличивать размер `pg_statistic` и не нагружать планировщик бесполезной работой, значения, превышающие 1 Кбайт, исключаются из статистики и анализа. Ведь такие большие значения, скорее всего, уникальны и все равно не имеют шансов попасть в `most_common_vals`.

116 Для эксперимента ограничим размер списка наиболее частых значений на уровне столбца:

```
ALTER TABLE routes
ALTER COLUMN arrival_airport SET STATISTICS 10;
ANALYZE routes;
```

Если значение попало в список наиболее частых, селективность можно узнать непосредственно из статистики. Пример — поиск маршрутов в Мумбаи:

```
EXPLAIN SELECT *
FROM routes WHERE arrival_airport = 'BOM';
               QUERY PLAN
-----
Seq Scan on routes
  (cost=0.00..59.38 rows=55 width=103)
  Filter: (arrival_airport = 'BOM'::bpchar)
(3 rows)
```

Точное значение:

```
SELECT count(*) FROM routes WHERE arrival_airport = 'BOM';
 count
-----
      55
(1 row)
```

Вот как выглядит список наиболее частых значений и частоты их встречаемости:

```
SELECT unnest(most_common_vals::text::text[]) AS mcv,
       unnest(most_common_freqs::text::float[]) AS mcf
FROM pg_stats
WHERE tablename = 'routes'
  AND attname = 'arrival_airport'
ORDER BY 2 DESC;
```

```

mcv |      mcf
-----+-----
BOM | 0.027093597
FRA | 0.026108375
SVO | 0.026108375
DEL | 0.021182265
ATL | 0.020689655
BLR | 0.020689655
CAN | 0.020689655
TFU | 0.020197045
CDG | 0.019211823
HND | 0.019211823
(10 rows)

```

Кардинальность вычисляется как число строк в таблице routes, умноженное на частоту значения:

```

SELECT round(
    2030 * most_common_freqs[
        array_position((most_common_vals::text::text[]), 'BOM')
    ]
)
FROM pg_stats
WHERE tablename = 'routes'
    AND attname = 'arrival_airport';
round
-----
      55
(1 row)

```

Наиболее частые значения применяются и для оценки селективности неравенств. Для этого в `most_common_vals` надо найти все значения, удовлетворяющие неравенству, и просуммировать частоты соответствующих элементов из `most_common_freqs`.

Если же указанного значения нет в списке наиболее частых, оно вычисляется исходя из предположения, что все данные (кроме наиболее частых) распределены равномерно.

118 Например, в списке частых значений нет аэропорта Сантья-
vi го в Чили. Проверим оценку:

```
EXPLAIN SELECT *
FROM routes WHERE arrival_airport = 'SCL';
               QUERY PLAN
-----
Seq Scan on routes
  (cost=0.00..59.38 rows=25 width=103)
  Filter: (arrival_airport = 'SCL'::bpchar)
(3 rows)
```

Она, однако, не сильно отличается от точного значения:

```
SELECT count(*) FROM routes
WHERE arrival_airport = 'SCL';
 count
-----
      18
(1 row)
```

Для получения оценки планировщик сначала вычисляет сумму частот наиболее частых значений:

```
SELECT sum(f)
FROM pg_stats s, unnest(s.most_common_freqs) f
WHERE s.tablename = 'routes'
      AND s.attnname = 'arrival_airport';
      sum
-----
0.22118227
(1 row)
```

На менее частые значения приходятся оставшиеся строки. Поскольку мы исходим из предположения о равномерности распределения менее частых значений, селективность будет равна $1/\text{число уникальных значений}$:

```

SELECT n_distinct
FROM pg_stats
WHERE tablename = 'routes' AND attname = 'arrival_airport';
   n_distinct
-----
              73
(1 row)

```

Учитывая, что из этих значений 10 входят в список наиболее частых, и нет неопределенных значений, получаем оценку:

```

SELECT round( 2030 * (1 - 0.22118227) / (73 - 10) );
   round
-----
        25
(1 row)

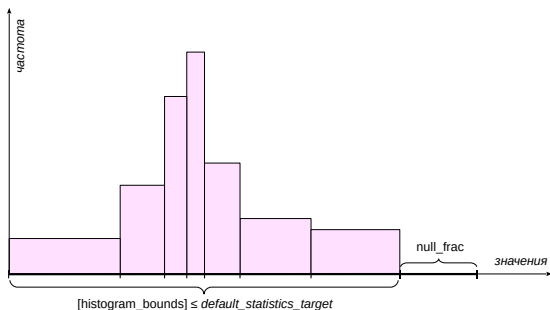
```

Гистограмма

Если число различных значений слишком велико, чтобы записать их в массив, на помощь приходит гистограмма. Гистограмма состоит из нескольких корзин, в которые помещаются значения. Количество корзин ограничено тем же параметром `default_statistics_target`.

Ширина корзин выбирается так, чтобы в каждую попало примерно одинаковое число значений (на рисунке это выражается в одинаковой площади прямоугольников). При таком построении достаточно хранить только массив крайних значений каждой корзины — поле `histogram_bounds`. Частота одной корзины равна $1/\text{число корзин}$.

Оценить селективность условия `поле < значение` можно как $N/\text{общее число корзин}$, где N — число корзин, лежащих



слева от значения. Оценку можно улучшить, добавив часть корзины, в которую попадает само значение.

При этом гистограмма строится так, чтобы не включать наиболее частые значения и NULL:

```
SELECT histogram_bounds
FROM pg_stats
WHERE tablename = 'routes'
AND attname = 'arrival_airport';
      histogram_bounds
```

```
-----
{AER, CKG, DFW, HEL, KZN, LHR, MXP, PKX, SVX, YEG, YYZ}
(1 row)
```

Поэтому при условиях «больше» и «меньше» для оценки будет использоваться и гистограмма, и список наиболее частых значений.

Рассмотрим пример:

EXPLAIN SELECT *

FROM routes WHERE arrival_airport <= 'DFW';

QUERY PLAN

Seq Scan on routes
 (cost=0.00..59.38 rows=579 width=103)
 Filter: (arrival_airport <= 'DFW'::bpchar)
(3 rows)

121

vi

Точное значение:

SELECT count(*) FROM routes WHERE arrival_airport <= 'DFW';

count

609
(1 row)

Как получена оценка?

Учтем частоту наиболее частых значений, попадающих в указанный интервал:

**SELECT sum(s.most_common_freqs[
 array_position((s.most_common_vals::text::text[]),v)
])**

FROM pg_stats s,

unnest(s.most_common_vals::text::text[]) v

WHERE s.tablename = 'routes'

AND s.attname = 'arrival_airport'

AND v <= 'DFW';

sum

0.12955666
(1 row)

Указанный интервал занимает ровно 2 корзины гистограммы из 10, а неопределенных значений в данном столбце нет. Получаем следующую оценку:

122
vi

```
SELECT round(  
    2030 * ( 0.12955666 + (1 - 0.22118227) * (2.0 / 10.0) )  
);  
round  
-----  
579  
(1 row)
```

В общем случае учитываются и не полностью занятые корзины (с помощью линейной аппроксимации).

Для оценки селективности условия *поле = значение* гистограмма не годится; приходится довольствоваться предположением о равномерном распределении и брать в качестве оценки $1/\text{число уникальных значений}$.

Дополнительные поля

Сервер хранит еще несколько показателей статистики.



В поле `pg_stats.correlation` записывается показатель физической упорядоченности значений столбца. Если значения хранятся строго по возрастанию, показатель будет близок к единице; если по убыванию — к минус единице. Чем более хаотично расположены данные на диске, тем ближе значение показателя к нулю. Именно это поле использует оптимизатор, когда выбирает между сканированием битовой карты и обычным индексным сканированием.

Поле `pg_class.relallvisible` хранит количество страниц таблицы, которые содержат только актуальные версии строк (эта информация обновляется вместе с картой видимости). Если количество недостаточно велико, планировщик может отказаться от сканирования только индекса в пользу сканирования по битовой карте.

В поле `pg_stats.avg_width` сохраняется средний размер значений в данном столбце в байтах для расчета необходимого для операции объема памяти.

123
vi

Частные и общие планы

При простом протоколе запросов (вспомните главу II «Планирование и выполнение») каждый запрос планируется заново с учетом значений параметров.

Расширенный протокол позволяет подготавливать операторы, причем операторы могут иметь параметры. Подготовка всегда включает разбор и переписывание запроса, а дерево разбора сохраняется в локальной памяти обслуживающего процесса.

При выполнении подготовленного оператора есть варианты. Запрос может, как обычно, планироваться каждый раз заново с учетом значений параметров. Такой план называется *частным* (custom).

Это имеет смысл при неравномерном распределении, ведь для разных значений оптимальные планы могут отличаться. Например, для высокоселективных условий может лучше работать индексный доступ, а если значение встречается часто — последовательный.

Но запрос можно спланировать и без учета параметров. Это позволяет сохранить не только дерево разбора, но и сам план, и не тратить время на повторное планирование. Такой план называется *общим* (generic).

Общий план отлично работает в случае равномерного распределения, когда селективность условия не зависит от

124 конкретное значения. Но в случае неравномерного рас-
vi пределения общий план может хорошо работать для одних значений и плохо — для других.

Разберемся, как планировщик решает, какой вариант использовать.

Подготовим запрос и создадим индекс:

```
PREPARE f(text) AS  
SELECT * FROM flights WHERE status = $1;  
CREATE INDEX ON flights(status);
```

Поиск отмененных рейсов будет использовать индекс, поскольку статистика говорит о том, что таких рейсов мало:

```
EXPLAIN EXECUTE f('Cancelled');  
QUERY PLAN
```

```
-----  
Index Scan using flights_status_idx on flights  
  (cost=0.29..56.50 rows=202 width=51)  
  Index Cond: (status = 'Cancelled'::text)  
(3 rows)
```

А поиск прибывших рейсов — нет, поскольку их много:

```
EXPLAIN EXECUTE f('Arrived');  
QUERY PLAN
```

```
-----  
Seq Scan on flights  
  (cost=0.00..838.89 rows=26878 width=51)  
  Filter: (status = 'Arrived'::text)  
(3 rows)
```

Однако планировщик строит и общий план без учета конкретных значений параметров. Если в какой-то момент оказывается, что стоимость общего плана не превышает среднюю стоимость уже построенных ранее частных планов, планировщик начинает пользоваться общим планом,

больше не выполняя планирование. Но первые пять раз 125
используются частные планы, чтобы накопить статистику. vi

Выполним оператор еще три раза:

```
EXECUTE f('Arrived');  
EXECUTE f('Arrived');  
EXECUTE f('Arrived');
```

Количество выполнений общего и частных планов можно посмотреть в представлении pg_prepared_statements:

```
SELECT name, generic_plans, custom_plans  
FROM pg_prepared_statements;  
name | generic_plans | custom_plans  
-----+-----+-----  
f | 0 | 5  
(1 row)
```

В следующий раз планировщик переключится на общий план. Вместо конкретного значения в плане будет указан номер параметра:

```
EXPLAIN EXECUTE f('Arrived');  
QUERY PLAN  
-----  
Index Scan using flights_status_idx on flights (cost=0...  
Index Cond: (status = ($1))  
(2 rows)
```

```
SELECT name, generic_plans, custom_plans  
FROM pg_prepared_statements;  
name | generic_plans | custom_plans  
-----+-----+-----  
f | 1 | 5  
(1 row)
```

126 **vi** Переход на общий план может быть нежелателен в случае неравномерного распределения значений. Параметр *plan_cache_mode* позволяет отключить использование частных планов (или наоборот, с самого начала использовать общий план):

```
SHOW plan_cache_mode;
      plan_cache_mode
-----
      auto
(1 row)

SET plan_cache_mode = 'force_custom_plan';
EXPLAIN EXECUTE f('Arrived');
                                QUERY PLAN
-----
Seq Scan on flights (cost=0.00..838.89 rows=26878 width...)
  Filter: (status = 'Arrived'::text)
(2 rows)

RESET plan_cache_mode;
```

2. Расширенная статистика

Базовой статистики, которая собирается автоматически, может не хватать для точных оценок кардинальности и селективности.

PostgreSQL позволяет администратору вручную определить, какая дополнительная — расширенная — статистика требуется. Можно собрать статистику, охватывающую не один столбец, а сразу несколько (многовариантная статистика), или статистику по произвольному выражению.

Заметьте, что базовая статистика собирается для таблиц и их столбцов, но не для индексов (за исключением индек-

сов по выражениям). Поэтому и индекс, построенный по нескольким столбцам, не приводит сам по себе к появлению многовариантной статистики.

Расширенная статистика создается командой `CREATE STATISTICS`. После того, как объект создан, соответствующая статистика будет собираться автоматически в фоновом режиме или командой `ANALYZE`.

Собранная информация хранится в таблицах `pg_statistic_ext` и `pg_statistic_ext_data`; доступная пользователю статистика видна в представлениях `pg_stats_ext` и `pg_stats_ext_exprs`.

Существует несколько видов многовариантной статистики, которые можно указать при создании объекта расширенной статистики.

Зависимые столбцы

Функциональная зависимость между столбцами показывает, насколько данные в одном столбце определяются значением другого столбца.



city	index
Самара	443000
Казань	420000
Нижний Новгород	603000
Великий Новгород	173000

128 В примере на рисунке почтовый индекс однозначно
vi определяет город, поэтому селективность условия city =
'Самара' and index = '443000' определяется селективно-
стью предиката index = '443000'. Такие предикаты, селек-
тивность которых нельзя рассчитывать отдельно друг от
друга, называются *коррелированными*.

Рассмотрим запрос с двумя условиями:

```
SELECT count(*) FROM routes
WHERE route_no = 'PG0085' AND arrival_airport = 'FCO';

count
-----
      7
(1 row)
```

Оценка оказывается сильно заниженной:

```
EXPLAIN SELECT * FROM routes
WHERE route_no = 'PG0085' AND arrival_airport = 'FCO';

               QUERY PLAN
-----
Bitmap Heap Scan on routes
  (cost=4.20..22.77 rows=1 width=103)
  Recheck Cond: (route_no = 'PG0085'::text)
  Filter: (arrival_airport = 'FCO'::bpchar)
  -> Bitmap Index Scan on routes_route_no_validity_excl
      (cost=0.00..4.20 rows=7 width=0)
      Index Cond: (route_no = 'PG0085'::text)
(7 rows)
```

Причина в том, что планировщик полагается на то, что пре-
дикаты независимы, и считает итоговую селективность как
произведение селективностей условий, объединенных ло-
гическим «и». Это хорошо видно в приведенном плане:
оценка в узле Bitmap Index Scan (условие на flight_no)
верная, а после фильтрации в узле Bitmap Heap Scan (усло-
вие на departure_airport) — заниженная.

Однако мы понимаем, что номер рейса однозначно определяет аэропорт отправления. Фактически, второе условие избыточно (считая, что аэропорт указан правильно).

129
vi

Это можно объяснить планировщику с помощью статистики по функциональной зависимости:

```
CREATE STATISTICS routes_dep(dependencies)
ON route_no, arrival_airport FROM routes;
ANALYZE routes;
```

Собранная статистика хранится в следующем виде (сначала идут порядковые номера атрибутов, а после двоеточия — коэффициент зависимости):

```
SELECT dependencies FROM pg_stats_ext
WHERE statistics_name = 'routes_dep';
dependencies
-----
{"1 => 4": 1.000000}
(1 row)
```

Проверим:

```
EXPLAIN SELECT * FROM routes
WHERE route_no = 'PG0085' AND arrival_airport = 'FCO';
QUERY PLAN
-----
Bitmap Heap Scan on routes
  (cost=4.20..22.78 rows=7 width=103)
  Recheck Cond: (route_no = 'PG0085'::text)
  Filter: (arrival_airport = 'FCO'::bpchar)
    -> Bitmap Index Scan on routes_route_no_validity_excl
      (cost=0.00..4.20 rows=7 width=0)
      Index Cond: (route_no = 'PG0085'::text)
(7 rows)
```

Теперь оценка улучшилась.

Команда \d показывает объекты расширенной статистики для конкретной таблицы. В разделе *Statistics objects* показаны имена, столбцы и нестандартные целевые значения объектов статистики.

```
...
Statistics objects:
  "bookings.routes_dep" (dependencies) ON route_no,
  arrival_airport FROM routes
```

Комбинации значений



Список наиболее частых комбинаций значений позволяет сохранить несколько комбинаций значений и их частоту.

На рисунке показаны возможные пары город — река. Очевидно, что предикаты со столбцами *city* и *river* коррелированы, но, в отличие от предыдущего примера, ни один из этих столбцов не определяет другой (между городами и реками отношение «многие ко многим»). В этом случае статистика по функциональным зависимостям не поможет улучшить оценки.

city	river
Самара	Волга
Казань	Волга
Нижний Новгород	Ока
Нижний Новгород	Волга
Великий Новгород	Волхов

Столбцы могут быть коррелированы, но не всегда между ними есть прямая функциональная зависимость. Выполним такой запрос:

```
EXPLAIN (analyze, timing off, summary off, buffers off)
SELECT * FROM routes
WHERE departure_airport = 'BOM' AND airplane_code = '7M7';

QUERY PLAN
```

```
-----
Bitmap Heap Scan on routes
  (cost=4.69..40.71 rows=9 width=103)
  (actual rows=17.00 loops=1)
  Recheck Cond: (departure_airport = 'BOM'::bpchar)
  Filter: (airplane_code = '7M7'::bpchar)
  Rows Removed by Filter: 38
  Heap Blocks: exact=27
-> Bitmap Index Scan on routes_departure_airport_lowe...
    (cost=0.00..4.69 rows=55 width=0)
    (actual rows=55.00 loops=1)
    Index Cond: (departure_airport = 'BOM'::bpchar)
    Index Searches: 1
(12 rows)
```

Планировщик ошибается почти в два раза. Учет функциональной зависимости недостаточно исправит ситуацию. В этом случае можно добавить расширенную статистику по частым комбинациям значений нескольких столбцов:

```
CREATE STATISTICS (mcv)
ON departure_airport, airplane_code FROM routes;
ANALYZE routes;
EXPLAIN SELECT * FROM routes
WHERE departure_airport = 'BOM' AND airplane_code = '7M7';

QUERY PLAN
```

```
-----
Bitmap Heap Scan on routes
  (cost=4.69..40.72 rows=17 width=103)
  Recheck Cond: (departure_airport = 'BOM'::bpchar)
  Filter: (airplane_code = '7M7'::bpchar)
-> Bitmap Index Scan on routes_departure_airport_lowe...
    (cost=0.00..4.69 rows=55 width=0)
    Index Cond: (departure_airport = 'BOM'::bpchar)
(7 rows)
```

Оценка исправилась.

132 Статистику по наиболее частым комбинациям можно по-
vi смотреть так:

```
SELECT values, frequency
FROM pg_statistic_ext
JOIN pg_statistic_ext_data ON oid = stxoid,
pg_mcv_list_items(stxdmccv) m
WHERE stxname = 'routes_departure_airport_airplane_code_stat'
LIMIT 3;
```

values	frequency
{FRA,77W}	0.009852216748768473
{BOM,7M7}	0.008374384236453201
{PVG,77W}	0.008374384236453201

(3 rows)

Уникальные комбинации



Число уникальных комбинаций значений позволяет улучшить оценку кардинальности при группировке по нескольким столбцам.

В примере на рисунке число возможных комбинаций всех полей, очевидно, нельзя определить, просто перемножив количество уникальных значений для каждого столбца по отдельности.

city	region	index
Самара	Самарская обл.	443000
Казань	Респ. Татарстан	420000
Нижний Новгород	Нижегородская обл.	603000
Великий Новгород	Новгородская обл.	173000

Другая ситуация, в которой планировщик ошибается, связана с группировкой. Найдем количество пар аэропортов, связанных прямыми рейсами:

133
vi

```
SELECT count(*) FROM (  
  SELECT DISTINCT departure_airport, arrival_airport  
  FROM routes  
);  
count  
-----  
    932  
(1 row)
```

Планировщик сильно ошибается:

```
EXPLAIN SELECT DISTINCT departure_airport, arrival_airport  
FROM routes;
```

QUERY PLAN

```
-----  
HashAggregate  
  (cost=64.45..66.48 rows=203 width=8)  
  Group Key: departure_airport, arrival_airport  
  -> Seq Scan on routes  
      (cost=0.00..54.30 rows=2030 width=8)  
(5 rows)
```

Расширенная статистика может исправить и эту оценку:

```
CREATE STATISTICS  
ON departure_airport, arrival_airport FROM routes;  
ANALYZE routes;  
EXPLAIN SELECT DISTINCT departure_airport, arrival_airport  
FROM routes;
```

QUERY PLAN

```
-----  
HashAggregate  
  (cost=64.45..73.77 rows=932 width=8)  
  Group Key: departure_airport, arrival_airport  
  -> Seq Scan on routes  
      (cost=0.00..54.30 rows=2030 width=8)  
(5 rows)
```

134 Мы не указали вид статистики; в этом случае в создаваемый
vi объект будут включены все поддерживаемые виды.

Статистику по уникальным комбинациям можно увидеть следующим образом:

```
SELECT n_distinct
FROM pg_stats_ext
WHERE statistics_name =
      'routes_departure_airport_arrival_airport_stat';
      n_distinct
-----
{"3, 4": 932}
(1 row)
```

Посмотреть список всех объектов расширенной статистики можно командой \dX. Сами значения при этом надо смотреть в представлении pg_statistic_ext_data.

Статистика по выражению

Расширенная статистика по выражению позволяет собрать всю базовую статистику, которая была бы собрана, если бы в таблице был определен столбец, вычисляемый по этому выражению.

city	index	address
Самара	443000	443000, г. Самара
Казань	420000	420000, г. Казань
Нижний Новгород	603000	603000, г. Нижний Новгород
Великий Новгород	173000	173000, г. Великий Новгород

Заметим, что выражения также можно использовать вместо имен столбцов в многовариантной статистике любого типа.

Если в предикате слева или справа от оператора стоит не имя столбца, а выражение, планировщик использует фиксированную оценку селективности и часто ошибается:

135
vi

```
EXPLAIN (analyze, timing off, summary off, buffers off)
SELECT * FROM flights
WHERE extract( month FROM scheduled_departure
               AT TIME ZONE 'Europe/Moscow' ) = 1;
```

QUERY PLAN

```
Seq Scan on flights
  (cost=0.00..1027.24 rows=188 width=51)
  (actual rows=5529.00 loops=1)
  Filter: (EXTRACT(month FROM (scheduled_departure AT T...
  Rows Removed by Filter: 32142
(5 rows)
```

Можно исправить ситуацию, построив индекс по выражению, но индексы требуют ресурсов для хранения и постоянной синхронизации. Можно поступить иначе — создать расширенную статистику по выражению:

```
CREATE STATISTICS
ON extract( month FROM scheduled_departure
            AT TIME ZONE 'Europe/Moscow' )
FROM flights;
ANALYZE flights;
EXPLAIN
SELECT * FROM flights
WHERE extract( month FROM scheduled_departure
               AT TIME ZONE 'Europe/Moscow' ) = 1;
```

QUERY PLAN

```
Seq Scan on flights
  (cost=0.00..1027.24 rows=5582 width=51)
  Filter: (EXTRACT(month FROM (scheduled_departure AT T...
(3 rows)
```

Оценка стала корректной.

136 **Расширенная статистика по выражениям хранится в отдельной таблице. Вот несколько столбцов:**

```
SELECT statistics_name, left(expr,40) AS expr,  
       n_distinct, most_common_vals  
FROM pg_stats_ext_exprs \gx  
-[ RECORD 1 ]-----+-----  
statistics_name | flights_expr_stat  
expr            | EXTRACT(month FROM (scheduled_departure  
n_distinct      | 7  
most_common_vals | {1,12,10,3,11,4,2}
```

Удалим созданный индекс и расширенную статистику:

```
DROP INDEX flights_status_idx;  
DROP INDEX  
SELECT 'DROP STATISTICS ' || statistics_name  
FROM pg_stats_ext \gexec  
DROP STATISTICS  
DROP STATISTICS  
DROP STATISTICS  
DROP STATISTICS
```

VII. Профилирование

1. Что такое профилирование

В предыдущих главах мы разобрались с тем, как работают запросы, из каких «кирпичиков» строится план выполнения и что влияет на выбор того или иного плана. Это самое сложное и важное. Поняв механизмы, с помощью логики и здравого смысла можно разобраться в любой возникшей ситуации и понять, эффективно ли выполняется запрос и что можно сделать, чтобы улучшить показатели.

Но как найти тот запрос, который имеет смысл оптимизировать?

В принципе, решение любой задачи оптимизации (не только в контексте СУБД) начинается с профилирования, хоть этот термин и не всегда употребляется явно. Мы должны разбить задачу, вызывающую нарекания, на подзадачи и измерить, какую часть общего времени они занимают. Также полезна информация о числе выполнений каждой из подзадач.

Чем больше доля подзадачи в общем времени, тем больше выигрыш от оптимизации именно этой подзадачи. На практике приходится учитывать и ожидаемые затраты на оптимизацию: получить потенциальный выигрыш может оказаться нелегко.

Бывает так, что подзадача выполняется быстро, но часто (это, например, характерно для запросов, генерируемых ORM-ами). Стоит задаться вопросом: должна ли подзадача выполняться *так* часто? Изменение архитектуры решения никогда не дается просто, но может привести к существенному приросту производительности.

Профиль общей активности может многое сказать администратору, который занят не решением конкретной проблемы, а поиском наиболее ресурсоемких задач, оптимизация которых, вероятно, снизит нагрузку на систему. Такой профиль должна показывать система мониторинга.



Можно также воспользоваться расширением `pg_profile` Андрея Зубкова. Оно основано на статистических представлениях PostgreSQL, информация из которых сохраняется в хранилище снимков активности. Изучая и сравнивая между собой снимки, можно выявлять проблемы и причины их возникновения.



Детальное обсуждение того, как именно строить систему мониторинга, какие инструменты для этого использовать и какие метрики собирать, выходит за рамки книги, но порекомендуем труд Алексея Лесовского «**Мониторинг PostgreSQL**».

Для решения конкретной проблемы необходимо строить профиль так, чтобы измерения затрагивали только действия, необходимые для воспроизведения именно этой проблемы. Если, например, пользователь жалуется на то, что «окно открывается целую минуту», бессмысленно смотреть на всю активность в базе данных за эту минуту: в эти цифры попадут действия, не имеющие никакого отношения к открытию окна.

В каких единицах измерять ресурсоемкость подзадач в профиле? Самая осмысленная для конечного пользователя характеристика — это **время отклика**: сколько секунд прошло от «нажатия на кнопку» до «получения результата».

Однако с технической точки зрения смотреть на время не всегда удобно. Оно сильно зависит от массы внешних факторов: от наполненности кеша, от текущей загруженности сервера. Если поиском проблемы занимаются не на основном, а на тестовом сервере с иными характеристиками, то добавляется и разница в аппаратном обеспечении, в настройках, в профиле нагрузки.

В этом смысле может оказаться удобнее смотреть на **объем ввода-вывода**, то есть на число прочитанных и записанных страниц. Этот показатель более стабилен. Он не зависит от характеристик «железа» и от нагрузки, то есть будет совпадать на разных серверах с одинаковыми данными и равными значениями параметров конфигурации.

Поэтому очень важно анализировать проблемы на полном наборе данных. Для этого, например, можно использовать Database Lab Engine — средство быстрого создания тонких клонов, разработанное Postgres.ai.



Ввод-вывод, как правило, адекватно отражает объем работы, необходимый для выполнения запроса, поскольку основное время уходит на чтение и обработку страниц с данными. Хотя такой показатель не имеет смысла для конечного пользователя.

3. Профиль приложения

Часто проблемы с производительностью кроются именно в СУБД, поскольку один неадекватно построенный план запроса может увеличивать время на порядки. Но это не всегда так. Проблема может оказаться в том, что у клиента медленное соединение с сервером, что клиентская часть долго отрисовывает полученные данные и т. п. Это означает, что в профиль должна входить не только СУБД, но и клиентская часть, и сервер приложений, и передача данных по сети.

К сожалению, получить такой полный профиль достаточно трудно. Для этого все компоненты информационной системы должны быть снабжены подсистемами мониторинга. Но как минимум стоит измерить общее время отклика (хотя бы и секундомером) и сравнить с общим временем работы СУБД; убедиться в отсутствии существенных сетевых задержек и т. п. Иначе может оказаться, что ключи ищут там, где светлее, а не там, где их потеряли. Собственно, весь смысл профилирования и состоит в том, чтобы выяснить, что именно нуждается в оптимизации.

Далее мы будем считать установленным, что проблема именно в СУБД.

4. Профиль PL/pgSQL

При выполнении пользователем некоторого действия выполняется обычно не один, а несколько запросов. Как определить тот запрос, который имеет смысл оптимизировать? Для этого нужно построить профиль, детализированный до запросов.

Однако серверная часть приложения, выполняющаяся внутри СУБД, может состоять не только из SQL-запросов, но и содержать процедурный код.

Если код написан на языке PL/pgSQL, то для его профилирования можно воспользоваться расширением PL Profiler Яна Вика. Расширение позволяет профилировать как отдельно выполняемые скрипты, так и снимать профиль работающего сеанса. По результатам формируется html-файл, содержащий необходимую информацию. В том числе он включает изображение вызовов в виде «языков пламени» (flame graph).



Также можно порекомендовать расширение plpgsql_check Павла Стехуле. Это расширение проверяет достоверность идентификаторов SQL, используемых в коде PL/pgSQL, пытается выявить проблемы с производительностью, имеет встроенный профилировщик кода.



5. Профиль запросов

Для получения профиля по выполняемым SQL-запросам есть два основных средства, встроенных в PostgreSQL: журнал сообщений сервера и статистика выполнения запросов.

Журнал сообщений сервера

В журнале сообщений с помощью конфигурационных параметров можно включить вывод информации о запросах и времени их выполнения. Обычно для этого используется параметр `log_min_duration_statement`.

Как локализовать в журнале именно те запросы, которые относятся к действиям пользователя? Имело бы смысл включать и выключать параметр для конкретного сеанса, но штатных средств для этого не предусмотрено. Можно фильтровать общий поток сообщений, выделяя только нужные; для этого удобно в параметр *log_line_prefix* вывести дополнительную идентифицирующую информацию. Использование пула соединений еще больше услож-



няет ситуацию. Для анализа вложенных запросов (когда в запросе вызывается функция, содержащая другие запросы) потребуется модуль *auto_explain*.

Следующая проблема — анализ журнала, который может достигать громадных размеров. Для этого приходится применять внешние средства, из которых стандартом де-факто является расширение *pgBadger*.



Разумеется, можно выводить в журнал и собственные сообщения, если они могут помочь делу.

Статистика



Второй способ получения профиля запросов — статистика выполнения запросов, собираемая расширением *pg_stat_statements* и отображаемая в одноименном представлении. В главе VI «Статистика» мы уже встречали термин «статистика», но тогда говорили про распределение данных в таблицах и их столбцах, то есть про статистику для оптимизатора. Статистика выполнения запросов — это обобщенная информация о реальном исполнении, получаемая уже после завершения запросов.

В состав статистики выполнения запросов входят сведения о произведенном вводе-выводе в терминах страниц.

Поскольку число различных запросов может быть очень большим, размер хранилища ограничен конфигурационным параметром; в нем остаются наиболее часто используемые запросы. При этом «одинаковыми» считаются запросы, имеющие одинаковое (с точностью до констант) дерево разбора. Надо иметь в виду, что такие запросы могли иметь разные планы выполнения и выполняться разное время.

К сожалению, есть сложности и с идентификацией: запросы можно отнести к конкретному пользователю и базе данных, но не к сеансу.

При установке конфигурационного параметра `compute_query_id` в значения `auto` или `on` идентификаторы запросов будут вычисляться ядром СУБД. Это позволяет связывать информацию, полученную из разных расширений, по унифицированным идентификаторам.



Подключим расширение `pg_stat_statements`, с помощью которого будем строить профиль:

```
CREATE EXTENSION pg_stat_statements;  
ALTER SYSTEM SET shared_preload_libraries =  
    'pg_stat_statements';
```

Требуется перезапустить сервер:

```
postgres$ pg_ctl restart -l /home/postgres/logfile  
postgres$ /usr/local/pgsql/bin/psql demo
```

Настроим расширение так, чтобы собиралась информация о всех запросах, в том числе вложенных:

144 SET pg_stat_statements.track = 'all';
vii

Создадим представление, чтобы смотреть только на часть полей и выводить только общее число страниц кеша:

```
CREATE VIEW statements_v AS
SELECT
    queryid,
    toplevel,
    substring(regexp_replace(query, ' +', ' ', 'g') FOR 55)
        AS query,
    calls,
    round(total_exec_time)/1000 AS time_sec,
    shared_blks_hit + shared_blks_read +
        shared_blks_written AS shared_blks
FROM pg_stat_statements
ORDER BY total_exec_time DESC;
```

Рассмотрим задачу. Требуется построить отчет, выводящий сводную таблицу количества перевезенных пассажиров: строками отчета должны быть модели самолетов, а столбцами — категории обслуживания.

Сначала создадим функцию, возвращающую число пассажиров для заданной модели и категории обслуживания:

```
CREATE FUNCTION qty(
    airplane_code text, fare_conditions text
) RETURNS bigint
AS $$
    SELECT count(*)
    FROM timetable tt
        JOIN boarding_passes bp ON bp.flight_id = tt.flight_id
        JOIN seats s ON s.airplane_code = tt.airplane_code
            AND s.seat_no = bp.seat_no
    WHERE tt.airplane_code = qty.airplane_code
        AND s.fare_conditions = qty.fare_conditions;
$$ STABLE LANGUAGE sql;
```

Для отчета создадим табличную функцию:

```

CREATE FUNCTION report()
RETURNS TABLE(
    model text,
    economy bigint, comfort bigint, business bigint
)
AS $$
DECLARE
    r record;
BEGIN
    FOR r IN
        SELECT a.airplane_code, a.model
        FROM airplanes a
        ORDER BY a.model
    LOOP
        report.model := r.model;
        report.economy := qty(r.airplane_code, 'Economy');
        report.comfort := qty(r.airplane_code, 'Comfort');
        report.business := qty(r.airplane_code, 'Business');
        RETURN NEXT;
    END LOOP;
END;
$$ STABLE LANGUAGE plpgsql;

```

Теперь исследуем работу предложенной реализации, сбросив предварительно статистику:

```

SELECT pg_stat_statements_reset();
SELECT * FROM report();

```

model	economy	comfort	business
Аэробус A320neo	120803	0	24494
Аэробус A330-900neo	337923	42242	43775
Аэробус A350-1000	736214	0	115244
Аэробус A350F	0	0	0
Боинг 737 MAX 7	556344	0	61743
Боинг 767-300F	0	0	0
Боинг 777-300ER	2238243	329509	205873
Боинг 787-9 Dreamliner	613540	68505	156712
Бомбардье CRJ700	91626	21181	10604
Эмбраэр E170	191837	0	16006

(10 rows)

Посмотрим, какую статистику мы получили:

```
SELECT * FROM statements_v \gx
-[ RECORD 1 ]-----
queryid      | 4328082630277697223
toplevel     | t
query        | SELECT * FROM report()
calls        | 1
time_sec     | 26.368
shared_blks  | 8427310
-[ RECORD 2 ]-----
queryid      | -3792749405278051211
toplevel     | f
query        | SELECT count(*)              +
              | FROM timetable tt            +
              | JOIN boarding_passe
calls        | 30
time_sec     | 26.349
shared_blks  | 8426084
-[ RECORD 3 ]-----
queryid      | 947065213111747660
toplevel     | t
query        | SELECT pg_stat_statements_reset()
calls        | 1
time_sec     | 0
shared_blks  | 0
-[ RECORD 4 ]-----
queryid      | 704897965492952962
toplevel     | f
query        | SELECT a.airplane_code, a.model +
              | FROM airplanes a              +
              | ORDE
calls        | 1
time_sec     | 0
shared_blks  | 4
```

Первым идет запрос верхнего уровня (toplevel), который детализируется ниже. Второй — запрос из функции qty: он вызывался 30 раз и именно на него приходится основная работа.

Затем следуют запрос, по которому работает цикл, и запрос сброса статистики.

6. Профиль одного запроса

147
vii

Так или иначе, среди выполненных запросов мы находим тот, который будем оптимизировать. Как работать с самим запросом? В этом тоже поможет профиль, который выдает команда EXPLAIN ANALYZE.

Подзадачами такого профиля являются узлы плана (план — не плоский список, а дерево). Продолжительность выполнения узла и число его повторений покажут `time` и `loops` из блока значений `actual`.

С помощью параметра `buffers` можно получить объем ввода-вывода (и время, потраченное на операции ввода-вывода, если включен параметр `track_io_timing`).

План выполнения содержит также крайне важную информацию об ожидании оптимизатора относительно кардинальности каждого шага. Как правило, если нет серьезной ошибки в прогнозе кардинальности, то и план будет построен адекватный (если нет, то надо менять глобальные настройки). Поэтому стоит обращать внимание не только на наиболее ресурсоемкие узлы, но и на узлы с большим (на порядок или выше) расхождением между `rows` и `actual rows`. Смотреть надо на наиболее вложенный проблемный узел, поскольку ошибка будет распространяться от него выше по дереву.

Бывают ситуации, когда запрос выполняется так долго, что выполнить EXPLAIN ANALYZE не представляется возможным. В таком случае придется довольствоваться простым EXPLAIN и пытаться разобраться в причине неэффективности без полной информации.

Попробуем теперь вывести тот же отчет одним запросом SQL и посмотрим на его выполнение командой EXPLAIN

148 с параметрами `analyze` и `buffers`, чтобы в плане отобра-
vii жалось количество фактически прочитанных и записанных
страниц файлов данных и временных файлов:

```
EXPLAIN (analyze, costs off, timing off)
WITH rf AS MATERIALIZED (
  SELECT f.flight_id, r.airplane_code
  FROM flights f
       JOIN routes r ON r.route_no = f.route_no
                  AND r.validity @> f.scheduled_departure
), t AS (
  SELECT rf.airplane_code,
         count(*) FILTER
           (WHERE s.fare_conditions = 'Economy') AS economy,
         count(*) FILTER
           (WHERE s.fare_conditions = 'Comfort') AS comfort,
         count(*) FILTER
           (WHERE s.fare_conditions = 'Business') AS business
  FROM rf
       JOIN boarding_passes bp ON bp.flight_id = rf.flight_id
       JOIN seats s ON s.airplane_code = rf.airplane_code
                  AND s.seat_no = bp.seat_no
  GROUP BY rf.airplane_code
)
SELECT a.model,
       coalesce(t.economy, 0) AS economy,
       coalesce(t.comfort, 0) AS comfort,
       coalesce(t.business, 0) AS business
FROM airplanes a
   LEFT JOIN t ON a.airplane_code = t.airplane_code
ORDER BY a.model;
```



Как легко заметить, работать с большими планами в текстовом виде не всегда удобно. Для лучшей наглядности можно пользоваться сторонними инструментами, из которых отметим explain.tensorflow.org.

Обратите внимание, как уменьшилось время выполнения запроса и насколько меньше страниц пришлось прочитать благодаря устранению избыточных чтений.

```

-----
Sort (actual rows=10.00 loops=1)
  Sort Key: ((ml.model ->> current_setting('bookings.la...
  Sort Method: quicksort Memory: 25kB
  Buffers: shared hit=11806 read=38407
  CTE rf
    -> Hash Join (actual rows=37671.00 loops=1)
      Hash Cond: (f.route_no = r.route_no)
      Join Filter: (r.validity @> f.scheduled_departure)
      Rows Removed by Join Filter: 115016
      Buffers: shared hit=402
      -> Seq Scan on flights f (actual rows=37671.00 ...
        Buffers: shared hit=368
      -> Hash (actual rows=2030.00 loops=1)
        Buckets: 2048 Batches: 1 Memory Usage: 145kB
        Buffers: shared hit=34
        -> Seq Scan on routes r (actual rows=2030.0...
          Buffers: shared hit=34
    ...
      Buffers: shared hit=11806 read=38406
      -> Hash Join (actual rows=5982418.00 ...
        Hash Cond: ((rf.airplane_code = s...
        Buffers: shared hit=11806 read=38406
        -> Hash Join (actual rows=5982418...
          Hash Cond: (bp.flight_id = rf...
          Buffers: shared hit=11804 rea...
          -> Seq Scan on boarding_passe...
            Buffers: shared hit=11402...
          -> Hash (actual rows=37671.00...
            Buckets: 65536 (originall...
            Buffers: shared hit=402
            -> CTE Scan on rf (actual...
              Storage: Memory Maxi...
              Buffers: shared hit=402
          -> Hash (actual rows=1741.00 loop...
            Buckets: 2048 Batches: 1 Me...
            Buffers: shared hit=2 read=8
            -> Seq Scan on seats s (actua...
              Buffers: shared hit=2 read=8

Planning:
  Buffers: shared hit=6
Planning Time: 0.293 ms
Execution Time: 1365.147 ms
(54 rows)

```

150 Можно посчитать общее количество страниц во всех заде-
vii ствованных таблицах:

```
SELECT sum(relpages)
FROM pg_class
WHERE relname IN ('flights', 'routes', 'boarding_passes',
                  'seats', 'airplanes');
```

```
sum
-----
50212
(1 row)
```

Это число может служить грубой оценкой сверху для запроса, которому требуются все строки: обработка существенно большего числа страниц может говорить о том, что данные перебираются по несколько раз.

Видно, что в данном случае план близок к оптимальному. Его тоже можно улучшить, но уже не так радикально.

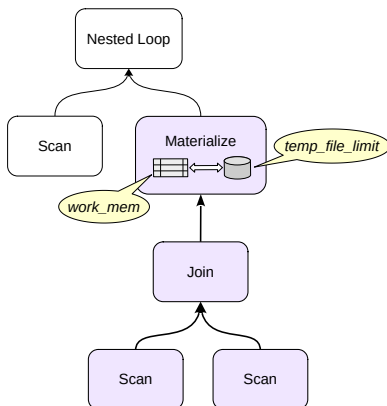
VIII. Материализация и функции

1. Материализация

Материализацией называется сохранение промежуточного набора строк с целью повторного использования, как правило, многократного. Сохранять набор можно на разных уровнях: для конкретного запроса, на уровне сеанса или на уровне базы данных.

Обычно узлы запроса передают друг другу данные по принципу конвейера: когда алгоритму, выполняющемуся в узле плана, требуется очередная строка набора, узел обращается к одному из дочерних узлов за следующей порцией данных. Однако в некоторых случаях исполнителю запроса имеет смысл (а иногда и необходимо) сразу получить все строки, сохранить их и иметь возможность обращаться к сохраненному результату повторно. Такое сохранение выполняет узел *Materialize*.

Строки сохраняются в оперативной памяти обслуживающего процесса, пока их объем укладывается в ограничение *work_mem*. При превышении этого ограничения все строки сбрасываются во временный файл и при необходимости читаются оттуда. Объем всех временных файлов одного процесса ограничен значением параметра *temp_file_limit*.



На рисунке верхний узел *Nested Loop* выполняет соединение вложенным циклом, но к внутреннему набору данных нет эффективного доступа: он вычисляется с помощью другого соединения. Чтобы не повторять многократно вложенное соединение, его результат можно материализовать.

После того как результат вложенного соединения материализован, узел *Nested Loop* будет обращаться к уже готовому набору строк.

Для примера найдем пары аэропортов, один из которых в часовом поясе Москвы, а другой — за полярным кругом:

```
EXPLAIN (costs off)
SELECT a1.city, a2.city
FROM airports a1, airports a2
WHERE a1.timezone = 'Europe/Moscow'
      AND abs(a2.coordinates[1]) > 66.652;
```

```
Nested Loop
-> Seq Scan on airports_data ml_1
    Filter: (abs(coordinates[1]) > '66.652'::double p...
-> Materialize
    -> Seq Scan on airports_data ml
        Filter: (timezone = 'Europe/Moscow'::text)
(6 rows)
```

Здесь для предиката внутреннего набора строк нет подходящего индекса, поэтому план с материализацией оказывается выгодным.

Материализация CTE

Общие табличные выражения (common table expression, CTE), они же подзапросы WITH, — прекрасный способ структурировать запрос и сделать его более понятным. В отличие от обычных подзапросов, использование CTE не приводит к большой вложенности.

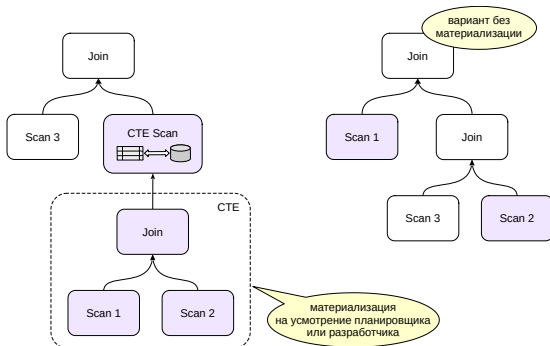


По возможности планировщик раскрывает (не материализует) подзапросы CTE. Это позволяет ему выбирать оптимальный порядок соединений. По умолчанию подзапрос материализуется в следующих ситуациях:

- Основной запрос обращается к подзапросу несколько раз — чтобы не повторять вычисления. В этом случае материализацию можно отменить, указав предложение AS NOT MATERIALIZED.
- Подзапрос имеет побочные эффекты (изменяет данные) — чтобы изменение произошло ровно один

раз. (К побочным эффектам относится также обращение к изменчивым функциям, см. следующий раздел «Функции».) В этом случае материализацию отменить невозможно.

Материализацию всегда можно принудительно включить, указав предложение AS MATERIALIZED.



Итак, оптимизатор старается не материализовать подзапросы в WITH без надобности:

```
EXPLAIN (costs off)
WITH q AS (
  SELECT r.route_no, a.airplane_code
  FROM routes r
  JOIN airplanes a ON a.airplane_code = r.airplane_code
)
SELECT *
FROM q
  JOIN seats s ON s.airplane_code = q.airplane_code
WHERE s.seat_no = '1A';
```

```

Hash Join
  Hash Cond: (ml.airplane_code = s.airplane_code)
    -> Hash Join
      Hash Cond: (r.airplane_code = ml.airplane_code)
        -> Seq Scan on routes r
        -> Hash
          -> Seq Scan on airplanes_data ml
    -> Hash
      -> Seq Scan on seats s
      Filter: (seat_no = '1A'::text)
(10 rows)

```

Но явное указание заставляет оптимизатор планировать подзапрос отдельно от основного запроса:

```

EXPLAIN (costs off)
WITH q AS MATERIALIZED (
  SELECT r.route_no, a.airplane_code
  FROM routes r
  JOIN airplanes a ON a.airplane_code = r.airplane_code
)
SELECT *
FROM q
JOIN seats s ON s.airplane_code = q.airplane_code
WHERE s.seat_no = '1A';

```

QUERY PLAN

```

Hash Join
  Hash Cond: (q.airplane_code = s.airplane_code)
  CTE q
    -> Hash Join
      Hash Cond: (r.airplane_code = ml.airplane_code)
        -> Seq Scan on routes r
        -> Hash
          -> Seq Scan on airplanes_data ml
    -> CTE Scan on q
  -> Hash
    -> Seq Scan on seats s
    Filter: (seat_no = '1A'::text)
(12 rows)

```

156 Если подзапрос используется в запросе несколько раз, пла-
viii нировщик выбирает материализацию, чтобы не выполнять
одни и те же действия многократно:

```
EXPLAIN (analyze, costs off, buffers)
WITH b AS (
  SELECT * FROM bookings
)
SELECT *
FROM b AS b1
JOIN b AS b2 ON b1.book_ref = b2.book_ref
WHERE b2.book_ref = '000112';
```

QUERY PLAN

```
Nested Loop (actual time=228.849..228.850 rows=0.00 loo...
  Buffers: shared read=15856, temp written=9746
  CTE b
    -> Seq Scan on bookings (actual time=0.371..70.590 ...
      Buffers: shared read=15856
    -> CTE Scan on b b1 (actual time=228.848..228.849 row...
      Filter: (book_ref = '000112'::bpchar)
      Rows Removed by Filter: 2486710
      Storage: Disk Maximum Storage: 77968kB
      Buffers: shared read=15856, temp written=9746
    -> CTE Scan on b b2 (never executed)
      Filter: (book_ref = '000112'::bpchar)
      Storage: Disk Maximum Storage: 77968kB
Planning:
  Buffers: shared hit=22 read=2
Planning Time: 0.095 ms
Execution Time: 232.877 ms
(17 rows)
```

Обычно это оправдано, особенно если в подзапросе происходят затратные вычисления. Но в некоторых случаях (как в этом примере) план без материализации может оказаться эффективнее, и тогда ее можно отменить.

Сравните значение `buffers` с предыдущим вариантом:

```
EXPLAIN (analyze, costs off, buffers)
WITH b AS NOT MATERIALIZED (
  SELECT * FROM bookings
)
SELECT *
FROM b AS b1
JOIN b AS b2 ON b1.book_ref = b2.book_ref
WHERE b2.book_ref = '000112';
```

QUERY PLAN

```
-----
Nested Loop (actual time=0.024..0.025 rows=0.00 loops=1)
  Buffers: shared read=3
    -> Index Scan using bookings_pkey on bookings (actual...
        Index Cond: (book_ref = '000112'::bpchar)
        Index Searches: 1
        Buffers: shared read=3
    -> Index Scan using bookings_pkey on bookings booking...
        Index Cond: (book_ref = '000112'::bpchar)
        Index Searches: 0
Planning:
  Buffers: shared hit=5
Planning Time: 0.079 ms
Execution Time: 0.033 ms
(13 rows)
```

Однако если подзапрос изменяет данные, материализация будет выполнена в любом случае: изменение обязано произойти только один раз.

Временные таблицы

Для получения промежуточных данных может использоваться сложный алгоритм, например, написанный на процедурном языке. В таком случае данные не получится посчитать в СТЕ, но их можно поместить во временную таблицу и использовать в нескольких запросах.



Временные таблицы удобнее для промежуточных данных, чем обычные, поскольку существуют только в пределах сеанса или транзакции (в зависимости от указанного при создании режима) и автоматически удаляются вместе с данными и зависимыми объектами (представлениями и индексами). К тому же временные таблицы не журналируются и кешируются в локальной памяти процесса, обслуживающего сеанс — это делает работу с ними более быстрой.

Однако для временных таблиц создаются записи в системном каталоге и файлы на диске. Поэтому при массовой работе с временными таблицами — например, в системе 1С — будет разрастаться системный каталог и увеличится нагрузка на файловую систему. Так что для 1С используют специальные патчи, нивелирующие эти нежелательные эффекты.

Память для кеша выделяется по необходимости и ограничена для сеанса параметром *temp_buffers* (после первого обращения к временной таблице изменить ограничение уже нельзя).

Создадим временную таблицу:

```
CREATE TEMP TABLE airports_nh -- северное полушарие
ON COMMIT PRESERVE ROWS -- по умолчанию
AS SELECT *
FROM airports
WHERE coordinates[1] > 0;
SELECT 4328
```

Она создается во временной схеме *pg_temp*.

По умолчанию таблица существует до конца сеанса; можно указать, чтобы при завершении транзакции удалялись строки (*ON COMMIT DELETE ROWS*) или сама таблица (*ON COMMIT DROP*).

Зачастую есть смысл проанализировать только что заполненную временную таблицу, прежде чем использовать ее в запросах. Локальная память процесса недоступна процессам автоочистки, поэтому очистку и анализ следует проводить вручную.

Сравните оценки кардинальности до и после анализа:

```
EXPLAIN SELECT *
FROM airports_nh;
               QUERY PLAN
-----
Seq Scan on airports_nh
  (cost=0.00..183.04 rows=5504 width=160)
(2 rows)

ANALYZE airports_nh;
EXPLAIN SELECT *
FROM airports_nh;
               QUERY PLAN
-----
Seq Scan on airports_nh
  (cost=0.00..171.28 rows=4328 width=91)
(2 rows)
```

Материализованные представления

Материализованное представление — это имеющийся результат запроса, сохраненный на уровне базы данных. К материализованному представлению можно обращаться как к обычной таблице, доступной только для чтения.



По материализованному представлению можно построить индексы (но нельзя добавить ограничение целостности — ограничения должны проверяться в базовых таблицах).

Для материализованного представления собирается такая же статистика, как и для обычных таблиц.

В отличие от обычных представлений, строки материализованного представления не изменяются при изменении базовых таблиц. Синхронизация выполняется вручную.



Полная синхронизация материализованного представления может быть слишком затратной. Штатное инкрементальное обновление (отдельных строк по мере изменения базовых таблиц) не доступно, но такую возможность дает расширение `pg_ivm` Юго Нагата.

Создадим материализованное представление:

```
CREATE MATERIALIZED VIEW airports_mv AS
SELECT * FROM airports;
SELECT 5501
```

И построим индекс:

```
CREATE UNIQUE INDEX ON airports_mv(airport_code);
EXPLAIN (costs off) SELECT *
FROM airports_mv
ORDER BY airport_code
LIMIT 3;
```

QUERY PLAN

```
-----
Limit
  -> Index Scan using airports_mv_airport_code_idx on a...
(2 rows)
```

При изменении содержимого базовых таблиц строки материализованного представления не изменяются:

```
UPDATE airports_data
SET airport_code = 'BSZ'
WHERE airport_code = 'FRU';
```

```
UPDATE 1
```

```
SELECT count(*)
FROM airports_mv
WHERE airport_code = 'BSZ';
```

```
count
-----
      0
(1 row)
```

Синхронизацию нужно проводить явно. Команда REFRESH полностью блокирует материализованное представление на время перестроения, что может быть нежелательным. В данном случае этого можно избежать, указав CONCURRENTLY, поскольку на материализованном представлении создан уникальный индекс:

```
REFRESH MATERIALIZED VIEW CONCURRENTLY airports_mv;
SELECT count(*) FROM airports_mv
WHERE airport_code = 'BSZ';
```

```
count
-----
      1
(1 row)
```

2. Функции

Категории изменчивости

Каждой функции сопоставлена категория изменчивости, которая определяет свойства возвращаемого значения при одинаковых значениях входных параметров.



Категория `Volatile` говорит о том, что возвращаемое значение может произвольно меняться. Такие функции будут выполняться каждый раз при каждом вызове (для каждой строки результата запроса). Если при создании функции категория не указана, назначается именно эта категория.

Категория `Stable` используется для функций, возвращаемое значение которых не меняется в пределах одного SQL-оператора. В частности, такие функции не могут менять состояние БД. Такая функция может быть выполнена один раз во время выполнения запроса, а затем будет использоваться вычисленное значение.

Категория `Immutable` еще более строгая: возвращаемое значение не меняется никогда. Такую функцию можно выполнить на этапе планирования запроса, а не во время выполнения.

Можно — не означает, что всегда происходит именно так, но планировщик вправе выполнить такие оптимизации и сэкономить на вызовах функции.

Для экспериментов создадим функцию, возвращающую случайное число:

```
CREATE FUNCTION rnd() RETURNS float
LANGUAGE sql VOLATILE
RETURN random();
```

Проверим план выполнения следующего запроса:

```
EXPLAIN (costs off)
SELECT * FROM generate_series(1,10) WHERE rnd() > 0.5;
          QUERY PLAN
-----
Function Scan on generate_series
  Filter: (random() > '0.5'::double precision)
(2 rows)
```

В плане мы видим обращение к табличной функции `generate_series` в узле `Function Scan`. Каждая строка результата сравнивается со случайным числом и при необходимости отбрасывается фильтром, в котором вычисляется скалярная функция `random`.

В этом можно убедиться и воочию:

```
SELECT * FROM generate_series(1,10) WHERE rnd() > 0.5;
generate_series
-----
          1
          3
          4
          9
         10
(5 rows)
```

Здесь с разной вероятностью получаем от 0 до 10 строк.

Функция с категорией изменчивости `Stable` будет вызвана всего один раз — поскольку мы фактически указали, что ее значение не может измениться в пределах оператора:

```
ALTER FUNCTION rnd() STABLE;
EXPLAIN (costs off)
SELECT * FROM generate_series(1,10) WHERE rnd() > 0.5;
          QUERY PLAN
-----
Result
  One-Time Filter: (rnd() > '0.5'::double precision)
    -> Function Scan on generate_series
(3 rows)
```

Узел `Result` формирует строку выборки, а выражение `One-Time Filter` вычисляется один раз, так что результатом запроса будет либо 0, либо 10 строк.

```
164 SELECT * FROM generate_series(1,10) WHERE rnd() > 0.5;
viii generate_series
-----
(0 rows)
```

Наконец, категория `Immutable` позволяет узнать значение функции еще на этапе планирования, поэтому во время выполнения условие фильтра уже не вычисляется. Выполним запрос два раза:

```
ALTER FUNCTION rnd() IMMUTABLE;
EXPLAIN (costs off)
SELECT * FROM generate_series(1,10) WHERE rnd() > 0.5;
      QUERY PLAN
-----
Result
One-Time Filter: false
(2 rows)

EXPLAIN (costs off)
SELECT * FROM generate_series(1,10) WHERE rnd() > 0.5;
      QUERY PLAN
-----
Function Scan on generate_series
(2 rows)
```

Для `Immutable` получаем случайный план!

Ответственность «за дачу заведомо ложных показаний» о назначении категории лежит на разработчике.

Подстановка кода функций



Оптимизатор PostgreSQL умеет подставлять (inline) тело функции в SQL-запрос. Это работает как со скалярными, так и с табличными функциями.

В обоих случаях есть много ограничений: функция должна быть написана на языке SQL, использовать единственный оператор SELECT и т. д. Скалярная функция к тому же не должна обращаться к таблицам базы данных и вызывать функции, имеющие менее строгую категорию, а табличная должна быть стабильной или постоянной.

Важным преимуществом подстановки тела функции в запрос является то, что функция становится прозрачной для планировщика. Например, дополнительные условия в теле основного запроса могут быть применены к запросу из тела функции, что позволит как можно раньше отфильтровать лишние строки.

Тело очень простых скалярных функций на языке SQL может быть подставлено прямо в основной SQL-оператор непосредственно перед планированием. В этом случае время на вызов функции не тратится.

Подстановку тела скалярной функции мы, на самом деле, уже видели на примере rnd. Проверим план запроса в случае, когда категория изменчивости функции rnd (Stable) не соответствует категории изменчивости функции random (Volatile):

```
ALTER FUNCTION rnd() STABLE;  
EXPLAIN (costs off)  
SELECT * FROM generate_series(1,10) WHERE rnd() > 0.5;  
QUERY PLAN
```

```
-----  
Result  
  One-Time Filter: (rnd() > '0.5'::double precision)  
    -> Function Scan on generate_series  
(3 rows)
```

В фильтре упоминается функция rnd, подстановка не произошла.

Поменяем категорию изменчивости:

```
ALTER FUNCTION rnd() VOLATILE;  
EXPLAIN (costs off)  
SELECT * FROM generate_series(1,10) WHERE rnd() > 0.5;  
QUERY PLAN  
-----  
Function Scan on generate_series  
  Filter: (random() > '0.5'::double precision)  
(2 rows)
```

Теперь в фильтре упоминается функция `random`. Она будет вызываться напрямую, минуя «обертку» в виде функции `rnd`.

Вызов табличных функций

Когда табличная функция вызывается в предложении `FROM`, за ее вычисление отвечает узел `Function Scan`. При этом сначала все строки, возвращаемые функцией, материализуются, и только затем передаются родительскому узлу плана. Таково текущее ограничение реализации.

Если же функция вызывается в предложении `SELECT`, она выполняется в узле `ProjectSet`. В этом случае функция может воспользоваться интерфейсом возвращения строк «по требованию» (`value-per-call`), без материализации.

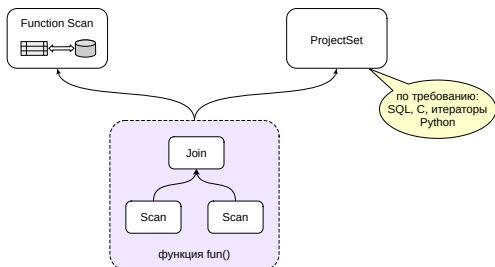
Так работают функции на языке SQL, большинство встроенных функций (написанных на C) и функции на PL/Python, возвращающие итератор. Функции на других языках программирования тоже могут использовать этот интерфейс, если такая возможность в них реализована.

Вызовем функцию `generate_series` в предложении `FROM` с ограничением на количество строк (`LIMIT`):

SELECT * FROM fun()

SELECT fun()

167
viii



```
EXPLAIN (analyze, costs off, buffers off)
SELECT * FROM generate_series(1,10_000_000)
LIMIT 10;
```

QUERY PLAN

```
Limit (actual time=287.450..287.452 rows=10.00 loops=1)
  -> Function Scan on generate_series (actual time=287....
Planning Time: 0.013 ms
Execution Time: 293.738 ms
(4 rows)
```

А теперь повторим запрос, вызвав функцию из предложения SELECT:

```
EXPLAIN (analyze, costs off, buffers off)
SELECT generate_series(1,10_000_000)
LIMIT 10;
```

QUERY PLAN

```
Limit (actual time=0.001..0.002 rows=10.00 loops=1)
  -> ProjectSet (actual time=0.001..0.001 rows=10.00 lo...
    -> Result (actual time=0.000..0.000 rows=1.00 loo...
Planning Time: 0.021 ms
Execution Time: 0.010 ms
(5 rows)
```

В первом случае в плане запроса видим узел Function Scan — сначала были получены все строки из функции, и только потом наложено ограничение LIMIT.

Во втором случае в плане появился узел ProjectSet, который формирует десять строк выборки — ровно столько, сколько требуется узлу Limit. Время выполнения запроса сократилось на порядки. Узел Result здесь представляет опущенное в запросе предложение FROM — он передает родительскому узлу ровно одну строку.

Однако не все функции могут возвращать строки по одной. Например, обращение к любой функции на языке PL/pgSQL возвращает все строки результата:

```
CREATE FUNCTION plpgsql_rows() RETURNS SETOF integer
AS $$
BEGIN
    RETURN QUERY
        SELECT * FROM generate_series(1,10_000_000);
END
$$ LANGUAGE plpgsql;

CREATE FUNCTION

EXPLAIN (analyze, costs off, buffers off)
SELECT plpgsql_rows() LIMIT 10;

                                QUERY PLAN
```

```
-----
Limit (actual time=651.055..651.057 rows=10.00 loops=1)
  -> ProjectSet (actual time=651.054..651.055 rows=10.0...
    -> Result (actual time=0.000..0.001 rows=1.00 loo...
Planning Time: 0.038 ms
Execution Time: 658.696 ms
(5 rows)
```

В плане — узел ProjectSet, но теперь серверу приходится получить результат функции полностью, а затем наложить ограничение LIMIT. Время выполнения это хорошо показывает.

Обычно (если не удалось подставить тело функции в запрос) оптимизатор не имеет возможности анализировать код функции и воспринимает ее как «черный ящик».

Однако можно дать оптимизатору приблизительную информацию о стоимости вызова функции и числе возвращаемых строк. Параметр `COST` задает стоимость пользовательской функции в единицах `cpu_operator_cost`. По умолчанию C-функции получают оценку 1, а на других языках — 100.



Параметр `ROWS` указывает примерное число возвращаемых строк. `COST` и `ROWS` можно указывать как при создании функции, так и для уже существующих функций.

CREATE FUNCTION fun()

```
SELECT * FROM fun( )
```

Function Scan (rows=1000 cost=100)

оценки
по умолчанию

CREATE FUNCTION fun() ROWS 12 COST 123

```
SELECT * FROM fun( )
```

Function Scan (rows=12 cost=123)

ROWS

COST

Напишем табличную функцию на языке PL/pgSQL, выводящую дни недели:

```

170 CREATE FUNCTION days_of_week() RETURNS SETOF text
viii AS $$
BEGIN
    FOR i IN 7 .. 13 LOOP
        RETURN NEXT to_char(to_date(i::text, 'J'), 'TMDy');
    END LOOP;
END;
$$ LANGUAGE plpgsql;

CREATE FUNCTION
SELECT * FROM days_of_week();

days_of_week
-----
Пн
Вт
Ср
Чт
Пт
Сб
Вс
(7 rows)

```

План запроса:

```

EXPLAIN SELECT * FROM days_of_week();

          QUERY PLAN
-----
Function Scan on days_of_week
    (cost=0.25..10.25 rows=1000 width=32)
(2 rows)

```

Стоимость выполнения функции считается постоянной, для пользовательских функций по умолчанию она равна стоимости 100 операторов:

```

SELECT 100 * current_setting('cpu_operator_cost')::float;

?column?
-----
0.25
(1 row)

```

Но это значение можно изменить:

171

viii

```
ALTER FUNCTION days_of_week COST 10000;  
EXPLAIN SELECT * FROM days_of_week();  
QUERY PLAN
```

```
-----  
Function Scan on days_of_week  
  (cost=25.00..35.00 rows=1000 width=32)  
(2 rows)
```

Сервер оценивает кардинальность результата этой функции как 1000, хотя фактическое значение равно семи.

С помощью указания ROWS можно подсказать серверу ориентировочное количество строк, которое должна вернуть функция:

```
ALTER FUNCTION days_of_week ROWS 10;  
EXPLAIN SELECT * FROM days_of_week();  
QUERY PLAN
```

```
-----  
Function Scan on days_of_week  
  (cost=25.00..25.10 rows=10 width=32)  
(2 rows)
```

Теперь сервер считает, что функция вернет 10 строк, поэтому уменьшилась и полная стоимость узла:

```
SELECT 10000 * current_setting('cpu_operator_cost')::float  
  + 10 * current_setting('cpu_tuple_cost')::float;  
?column?  
-----  
      25.1  
(1 row)
```

172 Измененные значения можно посмотреть в системном ка-
viii талоге:

```
SELECT procost, prorows
FROM pg_proc WHERE proname='days_of_week';
procost | prorows
-----+-----
100000 |      10
(1 row)
```

Или с помощью метакоманды psql:

```
\sf days_of_week
CREATE OR REPLACE FUNCTION bookings.days_of_week()
RETURNS SETOF text
LANGUAGE plpgsql
COST 100000 ROWS 10
AS $function$
BEGIN
    FOR i IN 7 .. 13 LOOP
        RETURN NEXT to_char(to_date(i::text, 'J'), 'TMDy');
    END LOOP;
END;
$function$
```

Вспомогательные функции



Параметры `COST` и `ROWS` позволяют задать стоимость и количество строк, возвращаемых функцией, как постоянные величины. Но константы не всегда дают желаемый результат.

В PostgreSQL есть возможность для функции написать вспомогательную функцию, которая предоставляет планировщику информацию, зависящую от значений аргументов основной (целевой) функции.

CREATE FUNCTION fun(x) ROWS 18

173
viii

SELECT * FROM fun(1)

Function Scan (rows=18)

ROWS

SELECT * FROM fun(2)

Function Scan (rows=18)

CREATE FUNCTION fun(x) SUPPORT fun_support

SELECT * FROM fun(1)

Function Scan (rows=12)

fun_support(1)

SELECT * FROM fun(2)

Function Scan (rows=24)

fun_support(2)

Вспомогательная функция может по значениям аргументов целевой функции выдавать:

- оценку ее стоимости;
- оценку числа возвращаемых строк;
- выражение, эквивалентное вызову функции.

Дополнительно для функций, возвращающих boolean:

- оценку селективности;
- эквивалентный предикат с индексируемым оператором.

Вспомогательная функция может быть написана только на языке C.

Пометки параллельности

Ранее уже было сказано, что не каждый запрос может выполняться в параллельном режиме.



Поскольку оптимизатор не может проанализировать тело функции, он рассчитывает на пометки параллельности, определяя по ним возможность параллельной обработки. При создании функции (или позже) можно указать одну из трех пометок:

- UNSAFE — запрещаются параллельные планы, если в запросе есть вызов функции;
- RESTRICTED — разрешаются параллельные планы, но запрещено вызывать функцию в параллельной части плана;
- SAFE — безопасна для параллельной обработки.

По умолчанию используется пометка UNSAFE.

Пометки параллельности указываются также для пользовательских агрегатных функций.

Эти пометки можно увидеть в столбце `proparallel` таблицы `pg_proc` (`r` = `restricted`, `s` = `safe`, `u` = `unsafe`):

```
SELECT proparallel, count(*)
FROM pg_proc
GROUP BY proparallel;
```

proparallel	count
r	202
s	3374
u	109

(3 rows)

Все основные стандартные функции безопасны.

Пометки также показывает метакоманда `\df+` утилиты `psql` (поле `Parallel`):

```
\x \df+ random numeric \x
```

175

Expanded display is on.

viii

List of functions

```
-[ RECORD 1 ]-----+-----  
Schema          | pg_catalog  
Name            | random  
Result data type | numeric  
Argument data types | min numeric, max numeric  
Type            | func  
Volatility       | volatile  
Parallel         | restricted  
Owner            | postgres  
Security         | invoker  
Leakproof?       | no  
Access privileges |  
Language         | internal  
Internal name    | numeric_random  
Description      | random numeric in range  
Expanded display is off.
```

Проверим, как пометка параллельности влияет на план выполнения запроса.

Напишем функцию, вычисляющую стоимость билета. Она помечена как безопасная для параллельного выполнения:

```
CREATE FUNCTION ticket_price(ticket_no char(13))  
RETURNS numeric  
LANGUAGE plpgsql STABLE PARALLEL SAFE  
AS $$  
BEGIN  
    RETURN (SELECT sum(price)  
            FROM segments s  
            WHERE s.ticket_no = ticket_price.ticket_no  
    );  
END;  
$$;  
CREATE FUNCTION
```

Запрос проверяет, что общая стоимость бронирований совпадает с общей стоимостью билетов:

176
viii

```
EXPLAIN (costs off) SELECT
(SELECT sum(ticket_price(ticket_no)) FROM tickets) =
(SELECT sum(total_amount) FROM bookings);
QUERY PLAN
```

```
-----
Result
InitPlan 1
-> Finalize Aggregate
    -> Gather
        Workers Planned: 2
    -> Partial Aggregate
        -> Parallel Seq Scan on tickets

InitPlan 2
-> Finalize Aggregate
    -> Gather
        Workers Planned: 2
    -> Partial Aggregate
        -> Parallel Seq Scan on bookings

(13 rows)
```

План запроса состоит из двух частей: в узле InitPlan 1 выполняется подзапрос с агрегацией по tickets, а подзапрос в узле InitPlan 2 выполняет агрегацию по bookings.

Сейчас оба подзапроса выполняются параллельно. Помечая пометку параллельности на UNSAFE:

```
ALTER FUNCTION ticket_price PARALLEL UNSAFE;
EXPLAIN (costs off) SELECT
(SELECT sum(ticket_price(ticket_no)) FROM tickets) =
(SELECT sum(total_amount) FROM bookings);
QUERY PLAN
```

```
-----
Result
InitPlan 1
-> Aggregate
    -> Seq Scan on tickets

InitPlan 2
-> Aggregate
    -> Seq Scan on bookings

(7 rows)
```

Теперь оба подзапроса выполняются последовательно — пометка запрещает параллельные планы.

177
viii

А теперь пометим функцию как ограниченно распараллеливаемую (RESTRICTED):

```
ALTER FUNCTION ticket_price PARALLEL RESTRICTED;  
EXPLAIN (costs off) SELECT  
  (SELECT sum(ticket_price(ticket_no)) FROM tickets) =  
  (SELECT sum(total_amount) FROM bookings);  
                                QUERY PLAN
```

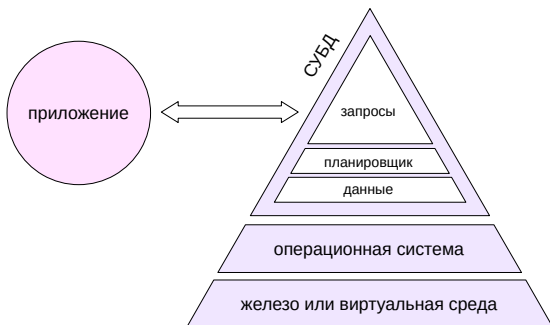
```
-----  
Result  
  InitPlan 1  
    -> Aggregate  
        -> Seq Scan on tickets  
  InitPlan 2  
    -> Finalize Aggregate  
        -> Gather  
            Workers Planned: 2  
            -> Partial Aggregate  
                -> Parallel Seq Scan on bookings  
(10 rows)
```

Подзапрос с функцией выполняется последовательно ведущим процессом, для второго подзапроса выбран параллельный план.

IX. Подходы к настройке

1. Что настраивать?

Производительность работы информационной системы обеспечивается на разных уровнях.



Мы совсем не рассматриваем два важных уровня: оборудование (которое может быть виртуализировано, и тогда к картине добавляется еще один слой) и операционную систему. В частности, работа СУБД с данными происходит через файловую систему, а эффективность и надежность операций ввода-вывода зависит от дисковой подсистемы.

Поэтому прежде всего нужно обеспечить необходимые ресурсы и настроить систему для их максимального использования.

Сервер баз данных настраивается так, чтобы служебные задачи не занимали больше ресурсов, чем необходимо, и чтобы большинство запросов работали приемлемо. На этом уровне мы можем не только настраивать конфигурационные параметры (из которых рассмотрим лишь малую часть), но и управлять расположением данных. Наша задача состоит в том, чтобы обеспечить разумное распределение ресурсов и адекватную работу планировщика.

Далее дело доходит до настройки отдельных запросов. В главе VII «Профилирование» мы говорили, что разработчик обычно думает над оптимизацией запросов, которые в данный момент пишет, а администратор — над оптимизацией запросов, оказывающих наибольшее влияние на сервер в целом. В этой главе мы рассмотрим некоторые приемы оптимизации запросов.

Но запросы, составляющие рабочую нагрузку, инициируются приложением. Поэтому и настройка приложения не менее важна, чем настройка СУБД.

2. Настройка сервера

Ресурсы: CPU и ввод-вывод

Начнем с настроек, которые определяют, какими ресурсами может распоряжаться СУБД.

PostgreSQL может использовать несколько ядер процессоров при параллельной обработке. Для этого запускаются

дополнительные фоновые рабочие процессы. Управлять их плановым количеством можно с помощью параметров:

181
ix

- *max_parallel_workers_per_gather* = 2;
- *max_parallel_workers* = 8;
- *max_worker_processes* = 8.

Подробно эти настройки были рассмотрены в главе III «Методы доступа», а использование фоновых процессов для прикладной разработки — в теме «Фоновые процессы» курса DEV2.

Из всех настроек ввода-вывода принципиальными являются две: количество операций, которые могут выполняться одновременно (параметр *effective_io_concurrency* для обычных задач и *maintenance_io_concurrency* для служебных), и количество идущих подряд страниц, которые могут читаться одной операцией (параметр *io_combine_limit*). Доступно несколько реализаций асинхронного ввода-вывода (параметр *io_method*) и большое количество более тонких настроек. При их использовании необходимо учитывать возможности подсистемы ввода-вывода сервера и профиль нагрузки.

Ресурсы: память

Ряд настроек влияет на выделение и использование оперативной памяти.

Параметр *work_mem* определяет размер доступной памяти для отдельных операций. Он рассматривался в соответствующих главах.

Этот параметр влияет и на выбор плана. Например, при небольших значениях предпочтение отдается сортировке,

которая лучше работает при недостатке памяти, чем хеширование. Поэтому для узлов, использующих хеширование, размер рабочей памяти можно увеличить с помощью параметра-мультипликатора *hash_mem_multiplier*.

Параметр *maintenance_work_mem* влияет на скорость построения индексов и на работу служебных процессов.

Параметр *shared_buffers* определяет размер буферного кеша экземпляра.

В любой OLTP-базе, даже очень большого размера, можно выделить более или менее компактный набор активно используемых данных. При недостаточном значении параметра *shared_buffers* эти данные не поместятся в буферный кеш, страницы будут постоянно вытесняться и создавать нагрузку на систему ввода-вывода (хотя этот эффект и сглаживается за счет кеширования данных операционной системой). Но и сильно увеличивать буферный кеш не стоит, поскольку при этом растут накладные расходы на его поддержание.

Поэтому оптимальный размер буферного кеша зависит от многих факторов: не существует какой-то универсальной формулы определения *shared_buffers*, подходящего любым системам. Настройка этого параметра рассматривается в курсе DBA2, но точно можно сказать, что значение по умолчанию крайне мало.

Параметр *effective_cache_size* подсказывает PostgreSQL общий объем кешируемых данных (как в буферном кеше, так и в кеше ОС). Чем больше его значение, тем более предпочтительным будет индексный доступ. Этот параметр не влияет на реальное выделение памяти.

Физическая организация данных может сильно влиять на производительность.

С помощью **табличных пространств** можно управлять размещением объектов по физическим устройствам ввода-вывода. Например, активно используемые данные хранить на SSD-накопителях, а архивные — на более медленных HDD.

Часть параметров сервера, которые зависят от характеристик носителя, (например, *random_page_cost*) можно устанавливать на уровне табличных пространств.

Секционирование позволяет организовать работу с данными очень большого объема. Основная выгода для производительности состоит в замене полного сканирования всей таблицы сканированием отдельной секции. Заметим, что секции тоже можно распределять по различным табличным пространствам.



Еще один вариант — размещение секций на разных серверах (**шардирование**) с возможностью выполнения распределенных запросов.

Стандартный PostgreSQL содержит только базовые механизмы, необходимые для организации шардирования: секционирование и расширение `postgres_fdw`. Более эффективно и полноценно шардинг реализуется с помощью внешних решений. Обзорно этот вопрос рассматривается в последней теме курса DBA3.



Оптимизатор: статистика

Планировщик опирается в своих оценках на статистику, как рассматривалось в главе VI «Статистика». Поэтому статистика должна собираться достаточно часто. Достигается это настройкой процесса автоочистки и автоанализа. Детали настройки подробно рассматриваются в курсе DBA2 «Настройка и мониторинг».

Кроме того, статистика должна быть достаточно точной. Абсолютной точности достичь не получится (и не нужно), но погрешности не должны приводить к построению некорректных планов. Признаком неактуальной, неточной статистики будет серьезное несоответствие ожидаемого и реального числа строк в листовых узлах плана.

Для увеличения точности может потребоваться изменить значение *default_statistics_target* (глобально или для отдельных столбцов таблиц). Иногда может оказаться полезным индекс по выражению, обладающий собственной статистикой. В отдельных случаях можно использовать расширенную статистику.

Оптимизатор: стоимость

Имеется большое число настроек для стоимости элементарных операций, из которых, как мы уже видели, в итоге складывается стоимость плана запроса. Такие настройки имеет смысл изменять, если запросы, несмотря на точную оценку кардинальности, выполняются не самым эффективным образом.

Параметры *seq_page_cost* и *random_page_cost* связаны со вводом-выводом и определяют относительную стои-

мость чтения одной страницы при последовательном и при произвольном доступе.

185
ix

Значение *seq_page_cost* равно единице и его не стоит изменять. Высокое значение по умолчанию параметра *random_page_cost* (4) отражает реалии HDD-дисков. Для SSD-накопителей (а также в случаях, когда все данные с большой вероятностью будут закешированы), значение этого параметра необходимо значительно уменьшать, например, до 1.1.

Параметры времени процессора определяют веса для учета стоимости обработки прочитанных данных. Обычно эти параметры не меняют, поскольку трудно спрогнозировать эффект изменений.

Но, как мы видели в главе VIII «Материализация и функции», иногда полезно задать стоимость пользовательской функции в единицах *cpu_operator_cost*.



Пример запроса с корректной оценкой кардинальности:

```
EXPLAIN (analyze, timing off, buffers off)
SELECT * FROM bookings
WHERE book_ref < 'E00000';
```

QUERY PLAN

```
-----
Seq Scan on bookings
  (cost=0.00..46939.88 rows=994464 width=21)
  (actual rows=967421.00 loops=1)
  Filter: (book_ref < 'E00000'::bpchar)
  Rows Removed by Filter: 1519289
  Planning Time: 0.088 ms
  Execution Time: 102.417 ms
(7 rows)
```

Планировщик выбрал последовательное сканирование, и сделал это на основе полной информации.

Так ли удачно это решение? Проверим, скомандовав планировщику не использовать последовательное сканирование, если есть другие способы:

```
SET enable_seqscan = off; -- действует до конца сеанса
```

Аналогичные enable-параметры есть и для прочих способов соединения, методов доступа и многих других операций. Эти параметры довольно грубо вмешиваются в работу планировщика, но весьма полезны для отладки и экспериментов.

```
EXPLAIN (analyze, timing off, buffers off)
SELECT * FROM bookings
WHERE book_ref < 'E00000';
```

QUERY PLAN

```
-----
Bitmap Heap Scan on bookings
  (cost=18619.53..46906.33 rows=994464 width=21)
  (actual rows=967421.00 loops=1)
  Recheck Cond: (book_ref < 'E00000'::bpchar)
  Heap Blocks: exact=15839
    -> Bitmap Index Scan on bookings_pkey
        (cost=0.00..18370.91 rows=994464 width=0)
        (actual rows=967422.00 loops=1)
        Index Cond: (book_ref < 'E00000'::bpchar)
        Index Searches: 1
  Planning Time: 0.045 ms
  Execution Time: 85.134 ms
(12 rows)
```

```
RESET enable_seqscan; -- отменяем
```

Результат, скорее всего, окажется в пользу индексного сканирования, поскольку все данные закешированы и произвольный доступ выполняется быстро. Такая же ситуация возможна при использовании быстрых SSD-накопителей. Делать выводы на основании одного запроса неправильно, но систематическая ошибка должна послужить пово-

дом к изменению глобальных настроек. В таком случае стоит либо уменьшить значение *random_page_cost*, чтобы планировщик не завышал стоимость индексного доступа, либо увеличить *effective_cache_size*, чтобы сделать индексное сканирование более привлекательным.

187
ix

3. Настройка приложения

Клиенты и соединения

Если приложение открывает слишком много соединений или устанавливает их слишком часто и на короткое время, может потребоваться установить между приложением и СУБД пул соединений. Однако пул накладывает на приложение определенные ограничения. Подробно этот вопрос рассмотрен в курсе DEV2.

Курсоры следует использовать, только если нужно получить небольшую часть выборки или ее размер зависит от действий пользователя. Параметр *cursor_tuple_fraction* подскажет планировщику, что нужно оптимизировать получение первых строк выборки.

Если приложение выполняет слишком большое количество мелких запросов (каждый из которых сам по себе выполняется эффективно), общая эффективность будет очень невысока. Это часто происходит при использовании средств объектно-реляционного отображения (ORM). В главе VII «Профилирование» мы рассматривали похожий пример с функцией, которая вызывалась в цикле.

В таких случаях со стороны СУБД практически нет средств для оптимизации (кроме подготовленных операторов, если запросы повторяются). Действенный метод — избавление от процедурного кода в приложении и перенос его на

сервер БД в виде небольшого числа крупных SQL-команд. Это дает планировщику возможность применить более эффективные способы доступа и соединений и избавляет от многочисленных пересылок данных от клиента к серверу и обратно. К сожалению, если приложение уже написано, переделать его бывает очень затратно.

Схема данных

На логическом уровне база данных должна быть **нормализованной**: неформально, в хранимых данных не должно быть избыточности. Если это не так, мы имеем дело с ошибкой проектирования: будет сложно проверять согласованность данных, возможны различные аномалии при изменении данных.

Однако на уровне хранения некоторое дублирование может дать существенный выигрыш в производительности — ценой того, что избыточные данные необходимо синхронизировать с основными.

Самый частый способ денормализации — индексы (хотя о них обычно не думают в таком контексте). Индексы обновляются автоматически.

Можно дублировать некоторые данные (или результаты расчета на основе этих данных) в столбцах таблиц. Можно использовать генерируемые столбцы или синхронизировать данные с помощью триггеров. Так или иначе, за денормализацию в этом случае также отвечает база данных.

Другой пример — материализованные представления. Их необходимо обновлять вручную, например, по расписанию или другим способом. Подробно это рассматривалось в главе VIII «Материализация и функции».

Дублировать данные можно и на уровне приложения, кешируя результаты выполнения запросов. Это популярный способ, но часто к нему прибегают из-за неправильной работы приложения с базой данных (например, в случае использования ORM). На приложение ложится задача по своевременному обновлению кеша, обеспечению разграничения доступа к данным кеша и т. п.

Важен правильный выбор **типов данных** из всего многообразия, предлагаемого PostgreSQL. Например, представление интервалов дат не двумя отдельными столбцами, а с помощью диапазоновых типов (`daterange`, `tstzrange`) позволяет использовать индексы GiST и SP-GiST для таких операций, как пересечение интервалов.

В ряде случаев эффект дает использование сложных типов (таких как массивы или JSON) вместо классического подхода с созданием отдельной таблицы. Это позволяет сэкономить на соединении и не требует хранения большого количества служебной информации в заголовках версий строк. Но такое решение следует принимать с большой осторожностью, поскольку оно имеет свои минусы.

Ограничения целостности важны сами по себе, но в некоторых случаях планировщик может учитывать их и для оптимизации:

- Ограничения первичного ключа и уникальности сопровождаются построением уникальных индексов и гарантируют, что все значения столбцов ключа различны (точная статистика). Такие гарантии позволяют и более эффективно выполнять соединения.
- Наличие внешнего ключа и ограничений NOT NULL дает гарантии, которые позволяют в ряде случаев устранять лишние соединения (что особенно важно

ет оценку селективности в случае, если соединение происходит по нескольким столбцам.

- Ограничение СНЕСК с использованием параметра *constraint_exclusion* позволяет не сканировать таблицы (или секции), если в них гарантированно нет требуемых данных.

Вот пример точной оценки кардинальности при соединении по составному ключу:

```
EXPLAIN SELECT *  
FROM segments s  
    JOIN boarding_passes bp ON s.flight_id = bp.flight_id  
        AND s.ticket_no = bp.ticket_no;  
  
      QUERY PLAN
```

```
Hash Join (cost=275309.45..625096.95 rows=5982418 width...)
Hash Cond: ((s.flight_id = bp.flight_id) AND (s.ticke...
-> Seq Scan on segments s
    (cost=0.00..134757.97 rows=7350197 width=31)
-> Hash (cost=109624.18..109624.18 rows=5982418 width...)
    -> Seq Scan on boarding_passes bp
        (cost=0.00..109624.18 rows=5982418 width=80)
(7 rows)
```

Однако если удалить внешний ключ, оценка становится неадекватной. Это еще одно проявление проблемы коррелированных предикатов:

```
BEGIN;  
ALTER TABLE boarding_passes  
DROP CONSTRAINT boarding_passes_ticket_no_flight_id_fkey;  
EXPLAIN SELECT *  
FROM segments s  
JOIN boarding_passes bp ON s.flight_id = bp.flight_id  
AND s.ticket_no = bp.ticket_no;
```

```

Hash Join (cost=275309.45..625096.95 rows=564 width=111)
Hash Cond: ((s.flight_id = bp.flight_id) AND (s.ticke...
-> Seq Scan on segments s
    (cost=0.00..134757.97 rows=7350197 width=31)
-> Hash (cost=109624.18..109624.18 rows=5982418 widt...
    -> Seq Scan on boarding_passes bp
        (cost=0.00..109624.18 rows=5982418 width=80)
(7 rows)
ROLLBACK;
```

4. Запросы

Пути оптимизации

Цель оптимизации запроса — получить адекватный план выполнения. Есть разные пути, которыми можно идти к этой цели.

Можно посмотреть в план запроса, понять причину неэффективного выполнения и сделать что-то, исправляющее ситуацию. К сожалению, проблема не всегда бывает очевидной, а исправление часто сводится к борьбе с оптимизатором.

Если идти таким путем, то хочется иметь возможность целиком или частично отключить планировщик и самому создать план выполнения. Такая возможность называется подсказками (хинтами) и в явном виде отсутствует в PostgreSQL.

Другой подход состоит в том, чтобы добиться корректного расчета кардинальности в каждом узле плана. Для этого, конечно, необходима аккуратная статистика, но ее одной

зачастую бывает недостаточно. Если при правильно оцененной кардинальности планировщик все равно строит неэффективный план, это повод заняться настройкой глобальных конфигурационных параметров.

Если идти вторым путем, то мы не боремся с планировщиком, а помогаем ему принять верное решение. К сожалению, во многих случаях это оказывается слишком сложной задачей.

Обычно имеет смысл применять оба способа, смотря по ситуации и сообразуясь со здравым смыслом.

Короткие запросы

Запросы можно (довольно условно) разделить на две группы, имеющие свои особенности и требующие разных подходов к оптимизации: «короткие» и «длинные».

К первой относятся запросы, характерные для OLTP-систем. Такие запросы читают немного данных и выдают одну или несколько строк. Они могут обращаться и к большим таблицам, но в этом случае имеют условия с высокой селективностью, позволяющие читать лишь малую часть данных.

Обычно короткие запросы обслуживают пользовательский интерфейс, и поэтому важно, чтобы они выполнялись как можно быстрее. Приоритет отдается времени отклика.

Это обеспечивается тем, что необходимые запросам данные читаются либо из очень небольших таблиц (одна-две страницы), либо по индексу. Соединения, как правило, выполняются методом вложенного цикла, который не требует подготовительных действий (в отличие от хеш-соединения) и не вызывает проблем при небольшом количестве соединяемых строк.

Рассмотрим запрос, который выводит список перелетов, 193
забронированных в определенный час (мы отключили па- ix
раллельное выполнение, бесполезное для коротких за-
просов):

```
SET max_parallel_workers_per_gather = 0;  
EXPLAIN (analyze, timing off)  
SELECT t.*  
FROM bookings b  
JOIN tickets t ON t.book_ref = b.book_ref  
WHERE date_trunc('hour',b.book_date) =  
      '2025-11-01 12:00:00+03';
```

QUERY PLAN

```
-----  
Hash Join (cost=53312.08..181808.82 rows=27993 width=52)  
  (actual rows=1338.00 loops=1)  
  Hash Cond: (t.book_ref = b.book_ref)  
  Buffers: shared hit=12523 read=61149  
  -> Seq Scan on tickets t  
    (cost=0.00..113800.73 rows=5598473 width=52)  
    (actual rows=5598473.00 loops=1)  
    Buffers: shared read=57816  
  -> Hash (cost=53156.65..53156.65 rows=12434 width=7)  
    (actual rows=567.00 loops=1)  
    Buckets: 16384 Batches: 1 Memory Usage: 150kB  
    Buffers: shared hit=12523 read=3333  
    -> Seq Scan on bookings b  
      (cost=0.00..53156.65 rows=12434 width=7)  
      (actual rows=567.00 loops=1)  
      Filter: (date_trunc('hour'::text, book_date) ...  
      Rows Removed by Filter: 2486143  
      Buffers: shared hit=12523 read=3333  
  
Planning:  
  Buffers: shared hit=174 read=16  
Planning Time: 2.899 ms  
Execution Time: 368.919 ms  
(22 rows)
```

Этот запрос можно отнести к разряду коротких: из двух больших таблиц требуется менее процента данных. Однако в плане мы видим, что таблицы читаются полностью. Кроме

того, наблюдается большая разница между прогнозируемой и фактической кардинальностью.

Начнем оптимизацию с самого вложенного узла, Seq Scan по таблице бронирований bookings. Из-за чего планировщик ошибается в оценке кардинальности? Все дело в функции date_trunc: для нее нет вспомогательной функции планировщика, поэтому берется фиксированная оценка селективности 0,5 %. Для исправления ситуации достаточно переписать условие так, чтобы слева от оператора находилось поле таблицы:

```
EXPLAIN (analyze, timing off, buffers off)
SELECT *
FROM bookings b
WHERE b.book_date >= '2025-11-01 12:00:00+03'
      AND b.book_date < '2025-11-01 13:00:00+03';
```

QUERY PLAN

```
Seq Scan on bookings b
  (cost=0.00..53156.65 rows=580 width=21)
  (actual rows=567.00 loops=1)
  Filter: ((book_date >= '2025-11-01 12:00:00+03'::time...
    Rows Removed by Filter: 2486143
  Planning Time: 0.057 ms
  Execution Time: 53.402 ms
(7 rows)
```

Теперь оценка похожа на правду, но по-прежнему выполняется полное сканирование с отбрасыванием большей части строк. Создадим индекс и проверим наш основной запрос. Заодно посмотрим на количество страниц, которые потребовалось прочитать, с помощью параметра buffers:

```
CREATE INDEX ON bookings(book_date);
CREATE INDEX
```

```
EXPLAIN (analyze, timing off)
SELECT t.*
FROM bookings b
JOIN tickets t ON t.book_ref = b.book_ref
WHERE b.book_date >= '2025-11-01 12:00:00+03'
      AND b.book_date < '2025-11-01 13:00:00+03';
```

QUERY PLAN

```
-----
Nested Loop (cost=0.99..11837.88 rows=1306 width=52) (...)
  Buffers: shared hit=2630 read=782
  -> Index Scan using bookings_book_date_idx on booking...
      Index Cond: ((book_date >= '2025-11-01 12:00:00+0...
      Index Searches: 1
      Buffers: shared hit=542 read=5
  -> Index Scan using tickets_book_ref_passenger_id_out...
      Index Cond: (book_ref = b.book_ref)
      Index Searches: 567
      Buffers: shared hit=2088 read=777
Planning:
  Buffers: shared hit=46 read=1
Planning Time: 0.152 ms
Execution Time: 135.279 ms
(14 rows)
```

Желаемое достигнуто: планировщик использует индексный доступ, данные соединяются методом вложенного цикла.

Возможным улучшением будет исключение доступа к таблице bookings за счет сканирования только индекса. Добавим к индексу столбец, который нужен для соединения:

```
DROP INDEX bookings_book_date_idx;
CREATE INDEX ON bookings(book_date) INCLUDE (book_ref);
EXPLAIN (analyze, timing off)
SELECT t.*
FROM bookings b
JOIN tickets t ON t.book_ref = b.book_ref
WHERE b.book_date >= '2025-11-01 12:00:00+03'
      AND b.book_date < '2025-11-01 13:00:00+03';
```

ix

```
Nested Loop (cost=0.99..11819.17 rows=1306 width=52) (...
  Buffers: shared hit=2866 read=6
    -> Index Only Scan using bookings_book_date_book_ref...
        Index Cond: ((book_date >= '2025-11-01 12:00:00+0...
        Heap Fetches: 0
        Index Searches: 1
        Buffers: shared hit=1 read=6
    -> Index Scan using tickets_book_ref_passenger_id_out...
        Index Cond: (book_ref = b.book_ref)
        Index Searches: 567
        Buffers: shared hit=2865
  Planning:
    Buffers: shared hit=42 read=1
  Planning Time: 0.144 ms
  Execution Time: 1.922 ms
(15 rows)
```

Мы выиграли еще около 15 % чтений. Это может быть важно для запросов, которые выполняются часто, но вряд ли это наш случай. Скорее всего, включение в индекс всех полей таблицы tickets будет избыточным — накладные расходы перевесят выигрыш в производительности. Остановимся на достигнутом.

Длинные запросы

Длинные запросы встречаются в OLAP-системах. Они характеризуются тем, что для получения результата должны прочитать большое количество данных. При этом не важно, сколько строк возвращает запрос: в результате агрегации может остаться и одна.

Такие запросы могут читать большие таблицы, а условия обычно низкоселективны. Поэтому вместо индексного доступа более эффективным становится последовательное

сканирование, а вместо соединения вложенным циклом — хеш-соединение. Тем более, что для длинного запроса важно выдать весь результат за разумное время, а не первые строки как можно раньше.

Очень важно следить, чтобы одни и те же данные не читались в запросе несколько раз. Это может происходить по самым разным причинам (из-за коррелированных подзапросов, использования функций и т. п.), которые приводят к появлению вложенных циклов, явных или неявных.

Разумеется, в плане длинного запроса могут встречаться и индексный доступ (при наличии высокоселективных условий), и соединение вложенным циклом (при соединении небольшого количества строк).

Благодаря тому, что длинные запросы обрабатывают много данных и обычно агрегируют их (пользователям редко бывают нужны миллионы строк), они могут выигрывать от параллельного выполнения.

Вычислим количество пассажиров, летевших рейсами, и вылет, и прибытие которых состоялись с опозданием от одной минуты до четырех часов:

```
\timing on
SELECT count(*)
FROM flights f
      JOIN segments tf ON tf.flight_id = f.flight_id
      JOIN boarding_passes bp ON bp.flight_id = tf.flight_id
                              AND bp.ticket_no = tf.ticket_no
WHERE f.actual_departure - f.scheduled_departure
      BETWEEN interval '1 min' AND interval '4 hours'
      AND f.actual_arrival - f.scheduled_arrival
      BETWEEN interval '1 min' AND interval '4 hours';

count
-----
5871849
(1 row)
```

198 Time: 41361.784 ms (00:41.362)
ix \timing off

Запрос выдает правильный результат, но время выполнения нас не устраивает.

Прежде чем приступить к оптимизации, выясним, короткий это запрос или длинный, а для этого узнаем селективность условий. Всего строк в таблице рейсов:

```
SELECT count(*) FROM flights f;  
count  
-----  
37671  
(1 row)
```

Из них условия отбирают:

```
SELECT count(*) FROM flights f  
WHERE f.actual_departure - f.scheduled_departure  
      BETWEEN interval '1 min' AND interval '4 hours'  
      AND f.actual_arrival - f.scheduled_arrival  
      BETWEEN interval '1 min' AND interval '4 hours';  
count  
-----  
26364  
(1 row)
```

Селективность имеющихся условий равна примерно 70 %, на остальные таблицы условий нет, причем все таблицы в запросе большие. Очевидно, что это длинный запрос.

В плане запроса мы видим индексный доступ и соединения вложенным циклом — операции, совсем не подходящие для длинного запроса:

```

Aggregate (cost=1711.10..1711.11 rows=1 width=8)
-> Nested Loop
    (cost=0.86..1710.70 rows=159 width=0)
    -> Nested Loop
        (cost=0.43..1624.34 rows=195 width=22)
        -> Seq Scan on flights f
            (cost=0.00..1498.13 rows=1 width=4)
            Filter: (((actual_departure - scheduled_d...
        -> Index Scan using segments_flight_id_idx on...
            (cost=0.43..123.58 rows=263 width=18)
            Index Cond: (flight_id = f.flight_id)
    -> Index Only Scan using boarding_passes_pkey on ...
        (cost=0.43..1.99 rows=150 width=36)
        Index Cond: ((ticket_no = tf.ticket_no) AND (...
(14 rows)

```

Почему так получилось? Проблема в ошибке оценки селективности условий. Планировщик считает, что из таблицы flights будет выбрана всего одна строка. Попробуем быстро проверить гипотезу, что переход на другой способ соединения поможет делу.

Попросим планировщик не использовать вложенный цикл, если это возможно:

```

SET enable_nestloop = off;
SET
EXPLAIN (analyze, timing off, costs off)
SELECT count(*)
FROM flights f
    JOIN segments tf ON tf.flight_id = f.flight_id
    JOIN boarding_passes bp ON bp.flight_id = tf.flight_id
                        AND bp.ticket_no = tf.ticket_no
WHERE f.actual_departure - f.scheduled_departure
    BETWEEN interval '1 min' AND interval '4 hours'
    AND f.actual_arrival - f.scheduled_arrival
    BETWEEN interval '1 min' AND interval '4 hours';

```

QUERY PLAN

```

-----
Aggregate (actual rows=1.00 loops=1)
  Buffers: shared hit=11393 read=100031, temp read=8496...
  -> Hash Join (actual rows=5871849.00 loops=1)
    Hash Cond: ((bp.flight_id = tf.flight_id) AND (bp...
    Buffers: shared hit=11393 read=100031, temp read=...
    -> Seq Scan on boarding_passes bp (actual rows=59...
      Buffers: shared hit=2632 read=47168
    -> Hash (actual rows=5871849.00 loops=1)
      Buckets: 262144 (originally 1024) Batches: 6...
      Buffers: shared hit=8761 read=52863, temp wri...
      -> Hash Join (actual rows=5871849.00 loops=1)
        Hash Cond: (tf.flight_id = f.flight_id)
        Buffers: shared hit=8761 read=52863
        -> Seq Scan on segments tf (actual rows=7...
          Buffers: shared hit=8393 read=52863
        -> Hash (actual rows=26364.00 loops=1)
          Buckets: 32768 (originally 1024) Bat...
          Buffers: shared hit=368
          -> Seq Scan on flights f (actual rows...
            Filter: (((actual_departure - sch...
            Rows Removed by Filter: 11307
            Buffers: shared hit=368

Planning:
  Buffers: shared hit=14
Planning Time: 0.201 ms
Execution Time: 3360.557 ms
(26 rows)

RESET enable_nestloop;

```

Обращения к таблицам автоматически сменились на последовательное чтение, выполнение ускорилось. Если проверить план с параметром `buffers`, можно увидеть, что запросу потребовалось прочитать примерно 110 тысяч страниц. Мы не смотрели план выполнения первой версии с параметром `buffers`, но она — проверьте! — читает почти в 200 раз больше данных.

Нам осталось более аккуратно объяснить планировщику, какой план стоит использовать. Самый мягкий способ —

предоставить более точную статистику. В данном случае подойдет статистика по выражению, которую мы рассматривали в главе VI «Статистика»:

201
ix

```
CREATE STATISTICS
ON (actual_departure - scheduled_departure) FROM flights;
CREATE STATISTICS
ON (actual_arrival - scheduled_arrival) FROM flights;
ANALYZE flights;
EXPLAIN SELECT count(*)
FROM flights f
  JOIN segments tf ON tf.flight_id = f.flight_id
  JOIN boarding_passes bp ON bp.flight_id = tf.flight_id
                        AND bp.ticket_no = tf.ticket_no
WHERE f.actual_departure - f.scheduled_departure
      BETWEEN interval '1 min' AND interval '4 hours'
  AND f.actual_arrival - f.scheduled_arrival
      BETWEEN interval '1 min' AND interval '4 hours';
QUERY PLAN
```

```
-----
Aggregate  (cost=517139.54..517139.55 rows=1 width=8)
  -> Hash Join
        (cost=247827.57..509803.06 rows=2934594 width=0)
        Hash Cond: ((tf.flight_id = bp.flight_id) AND (tf...
        -> Hash Join
              (cost=1729.12..155783.52 rows=3605540 width=22)
              Hash Cond: (tf.flight_id = f.flight_id)
              -> Seq Scan on segments tf
                    (cost=0.00..134757.97 rows=7350197 width=18)
              -> Hash
                    (cost=1498.13..1498.13 rows=18479 width=4)
                    -> Seq Scan on flights f
                          (cost=0.00..1498.13 rows=18479 width=4)
                          Filter: (((actual_departure - schedul...
        -> Hash
              (cost=109624.18..109624.18 rows=5982418 width=...)
              -> Seq Scan on boarding_passes bp
                    (cost=0.00..109624.18 rows=5982418 width=36)

(18 rows)
```

Оценки исправились, теперь во всех узлах они довольно точно соответствуют реальным цифрам.

Однако зададимся вопросом: не читаем ли мы лишние данные? Знание предметной области позволяет нам прийти к выводу, что таблица `segments` вовсе не нужна в запросе. Таблицу `flights` можно соединить непосредственно с `boarding_passes` — внешние ключи проверяют корректность данных, но не требуются для соединения. Добавление `segments` никак не ограничивает данные (не может быть ситуации, при которой посадочный талон есть, а перелет отсутствует).

Вообще, переформулирование запроса — самый разнообразный и вариативный способ влияния на производительность. Некоторые эквивалентные преобразования планировщик умеет выполнять сам, но мы можем, например:

- переписать условия так, чтобы они могли (или наоборот, не могли) использовать индекс;
- заменить коррелированные подзапросы (которые, по сути, являются вложенным циклом) соединениями;
- устранить лишние сканирования (как в этом примере, или, скажем, за счет применения оконных функций);
- использовать недоступные планировщику трансформации, такие, как преобразование условий `OR` в `UNION`.

Итак, упрощаем запрос:

```
EXPLAIN (analyze, timing off)
SELECT count(*)
FROM flights f
  JOIN boarding_passes bp ON bp.flight_id = f.flight_id
WHERE f.actual_departure - f.scheduled_departure
      BETWEEN interval '1 min' AND interval '4 hours'
  AND f.actual_arrival - f.scheduled_arrival
      BETWEEN interval '1 min' AND interval '4 hours';
```

```

Aggregate (cost=134395.40..134395.41 rows=1 width=8) (...)
  Buffers: shared hit=2194 read=47974
    -> Hash Join (cost=1729.12..127058.91 rows=2934594 w...
      Hash Cond: (bp.flight_id = f.flight_id)
      Buffers: shared hit=2194 read=47974
        -> Seq Scan on boarding_passes bp (cost=0.00..10...
          Buffers: shared hit=1826 read=47974
        -> Hash (cost=1498.13..1498.13 rows=18479 width=...
          Buckets: 32768 Batches: 1 Memory Usage: 1183kB
          Buffers: shared hit=368
        -> Seq Scan on flights f (cost=0.00..1498.13...
          Filter: (((actual_departure - scheduled_d...
          Rows Removed by Filter: 11307
          Buffers: shared hit=368
    Planning Time: 0.097 ms
    Execution Time: 447.913 ms
(16 rows)

```

Потребовалось прочитать около 50 тысяч страниц.

Проверим, сколько всего страниц в таблицах:

```

SELECT sum(relpages) FROM pg_class
WHERE relname IN ('flights','boarding_passes');

sum
-----
50168
(1 row)

```

Теперь мы уверены, что не читаем лишних данных. Соединение хешированием также выполняется эффективно, за один проход.

При наличии в системе свободных ядер стоило бы проверить паралельный режим, зачастую полезный для длинных запросов, но мы остановимся и сочтем оптимизацию завершенной.

Удалим созданный индекс и расширенную статистику:

```
DROP INDEX bookings_book_date_book_ref_idx;  
DROP INDEX  
SELECT 'DROP STATISTICS ' || statistics_name  
FROM pg_stats_ext \gexec  
DROP STATISTICS  
DROP STATISTICS
```

Подсказки оптимизатору



Еще один (традиционный для других СУБД) способ влияния — подсказки оптимизатору — отсутствует в PostgreSQL 18, но в 19-й версии появится расширение `pg_plan_advice`, которое вместе с `pg_stash_advice` позволит управлять подсказками без необходимости изменять текст запроса.

Пока же можно пользоваться неявными подсказками в виде конфигурационных параметров и других средств.



Кроме того, давно существуют сторонние расширения, например `pg_hint_plan`, автор — Китаро Хоригучи.

Но нельзя забывать, что использование подсказок, сильно ограничивающих свободу планировщика, может навредить в будущем, когда распределение данных изменится.

Заключение

Оптимизация запросов — это бесконечный процесс, который требует постоянного мониторинга и настройки. Нельзя рассчитывать что оптимально выполняющийся запрос будет оптимальным всегда: может измениться объем данных в таблицах, могут измениться версии СУБД и ОС, может измениться оборудование. Чтобы поддерживать высокую производительность, ей все время необходимо уделять внимание.

Книга задумывалась как введение в тему оптимизации запросов. В ней мы знакомим читателя с основами работы планировщика PostgreSQL — рассматриваем методы доступа к данным, способы соединения, использование статистики и основные приемы оптимизации.

Описанные в книге аспекты оптимизации запросов формируют основу для понимания механизмов работы планировщика. Но не останавливайтесь только на этой книге, изучайте новинки каждой версии PostgreSQL, повышайте свою квалификацию.

Из дополнительных источников в первую очередь рекомендуем официальную документацию PostgreSQL (как перевод на русский, так и оригинал). Читая документацию, вы получите емкую и точную информацию из первых рук — она написана самими разработчиками и всегда аккуратно поддерживается в актуальном состоянии.





Но читать документацию подряд, от первой до последней страницы — сложное и не всегда нужное дело. В том числе и потому, что логически связанная информация обычно распределена по разным главам. Поэтому мы и создаем разные учебные курсы, в которых стараемся изложить определенную тему пусть не столь полно, но зато понятно и последовательно.

Так что, если хотите изучить что-то новое, обратите внимание и на другие наши курсы:

- **DBA1. Базовый курс по администрированию PostgreSQL**, в котором рассказаны общие сведения об архитектуре, установке, настройке и администрировании сервера;
- **DBA2. Администрирование PostgreSQL. Настройка и мониторинг**, а также обновление сервера, локализация и работа с расширениями;
- **DBA3. Администрирование PostgreSQL. Резервное копирование и репликация**, а также представление о способах построения кластеров;
- **DBS. Администрирование PostgreSQL. Разграничение доступа**, а также настройка аутентификации, в том числе внешней;
- **DEV1. Разработка серверной части приложений PostgreSQL. Базовый курс** о серверном программировании на языках SQL и PL/pgSQL;
- **DEV2. Разработка серверной части приложений PostgreSQL. Расширенный курс** про возможности реализации логики приложений на стороне сервера;

- **PGPRO. Возможности Postgres Pro Enterprise**, которые отличают продукт нашей компании от свободно распространяемой СУБД PostgreSQL. 207

О компании

Компания Postgres Professional была основана в 2015 году и объединила ключевых российских разработчиков, вклад которых в развитие PostgreSQL признан мировым сообществом. Здесь готовят квалифицированные отечественные кадры в области разработки СУБД. В настоящее время в ней работает около 400 программистов, архитекторов, инженеров и других специалистов.

Postgres Professional выпускает несколько версий системы Postgres Pro, построенной на основе PostgreSQL, выполняет разработки на уровне ядра СУБД и расширений, оказывает услуги по проектированию и поддержке прикладных систем и миграции на PostgreSQL.

Компания уделяет большое внимание образовательной деятельности, организует крупнейшую ежегодную международную конференцию PgConf.Russia в Москве и принимает участие в конференциях по всему миру.

Контактная информация:

117036, г. Москва, ул. Дмитрия Ульянова, д. 7А

+7 495 150-06-91

info@postgrespro.ru



Postgres Pro компании Postgres Professional — российская коммерческая СУБД, основанная на свободно распространяемой СУБД PostgreSQL и разработанная согласно с требованиями корпоративных заказчиков. Входит в реестр российского программного обеспечения.

Postgres Pro Standard содержит все функциональные возможности PostgreSQL и дополнен различными расширениями и патчами, в том числе еще не принятыми сообществом. Клиент получает доступ к полезному функционалу и выигрывает в производительности, не дожидаясь очередного релиза PostgreSQL.

Postgres Pro Enterprise — это глубоко переработанная версия СУБД, благодаря большей надежности и повышенной производительности пригодная для решения серьезных промышленных задач.

Обе версии Postgres Pro, дополненные необходимыми средствами защиты информации, прошли **сертификацию ФСТЭК**.

Для использования любой версии Postgres Pro необходимо приобрести лицензию. Можно бесплатно получить интересующую вас версию СУБД для тестирования, изучения возможностей СУБД и разработки прикладного программного обеспечения.

Для образовательных организаций предоставляется бесплатная академическая лицензия на СУБД Postgres Pro Standard.

Postgres Pro Enterprise Manager (PPEM) — интегрированная панель управления СУБД Postgres Pro Enterprise.

PPEM предоставляет единую консоль с простым интерфейсом мониторинга и управления, позволяет выполнять основные административные действия из окна браузера, обеспечивает централизованный доступ ко всем экземплярам и базам данных. PPEM повышает продуктивность группы сопровождения БД и облегчает выполнение повседневных административных задач.



Postgres Pro Backup Enterprise

Postgres Pro Backup Enterprise (pg_probackup) — утилита для управления резервным копированием и восстановлением кластеров баз данных PostgreSQL.



Утилита работает с централизованным каталогом копий, который может размещаться как локально, так и удаленно, в том числе в хранилище S3.

Поддерживаются инкрементальное резервное копирование на уровне страниц данных, политики хранения резервных копий, проверка целостности данных и возможности восстановления на заданный момент времени, параллельное резервирование и восстановление, сжатие данных и файловая система CFS.



Shardman — это распределенная реляционная СУБД, основанная на PostgreSQL, но при этом обладающая расширенными возможностями СУБД Postgres Pro.

Shardman совместима с PostgreSQL, предоставляет строгие гарантии изоляции и согласованности данных, обеспечивает горизонтальное масштабирование и обладает встроенной отказоустойчивостью.

Карьера

Мы активно ищем специалистов, готовых разрабатывать продукты экосистемы Postgres и развивать открытое программное обеспечение.

В команде Postgres Professional работает большинство российских постгресистов. Мы занимаемся разработкой на уровне ядра PostgreSQL и его расширений, создаем экосистемные продукты для мониторинга и администрирования СУБД. Наши специалисты разрабатывают сервис DBaaS, проводят технический аудит и миграцию данных, оказывают техническую поддержку, занимаются исследованиями и создают инновационные продукты.



Мы открыты для сотрудничества с системными программистами, backend- и frontend-разработчиками, специалистами в области DevOps, QA-инженерами, администраторами баз данных и профессионалами других направлений.



Отказоустойчивые решения для Postgres

Проектирование и участие в создании высоконагруженных, высокопроизводительных и отказоустойчивых промышленных систем; консалтинговые услуги. Внедрение СУБД Postgres и оптимизация конфигурации.

Вендорская техническая поддержка

Техподдержка Postgres Pro и PostgreSQL круглосуточно и без выходных. Мониторинг, восстановление работоспособности, анализ непредвиденных обстоятельств, повышение производительности, исправление ошибок в СУБД и расширениях.

Миграция прикладных систем на СУБД Postgres

Оценка сложности миграции с других СУБД на Postgres. Разработка архитектуры нового решения и необходимых доработок. Миграция прикладных систем на СУБД Postgres и поддержка в процессе миграции.

Обучение Postgres

Обучение администраторов баз данных, разработчиков и архитекторов прикладных систем особенностям СУБД Postgres и эффективному использованию ее преимуществ.

Аудит СУБД

Экспертная оценка состояния СУБД. Аудит информационной безопасности систем на основе Postgres.

Толмачев Павел Владимирович
PostgreSQL 18. Оптимизация запросов

Авторы курса: Павел Лузанов,
Егор Рогов,
Павел Толмачев,
Илья Баштанов

Дизайнер обложки: Мария Лукашенко
postgrespro.ru/education/books/qptbook

© ООО «ППГ», 2026

Москва, Постгрес Профессиональный, 2026