Министерство науки и образования Российской Федерации
Федеральное государственное бюджетное образовательное учреждение высшего образования
«Московский государственный технический университет имени Н.Э. Баумана
(национальный исследовательский университет)»
(МГТУ им. Н.Э. Баумана)

МЕТОДИЧЕСКИЕ УКАЗАНИЯ К ЛАБОРАТОРНЫМ РАБОТАМ ПО КУРСУ «БАЗЫ ДАННЫХ»

Лабораторная работа №4
«Оптимизация процессов в PostgresSQL»

Авторы:
Скворцова М.А., magavrilova@bmstu.ru
Лапшин А.В.

Москва, 2022

Общие сведения

Сокращения

SQL – Structured Query Language («язык структурированных запросов»).

БД – База данных.

СУБД – Система управления базами данных.

ЯП – язык программирования.

PL/pgSQL – Procedural Language/PostGres Structured Query Language.

CTE – Общие табличные выражения.

Лабораторная работа посвящена упрощению работы аналитика с помощью создания функций на языке PL/pgSQL.

PL/pgSQL – процедурное расширение языка SQL, используемое в СУБД PostgreSQL. Этот язык предназначен для написания функций, триггеров и правил и обладает следующими особенностями:

- Добавляет управляющие конструкции к стандарту SQL;
- Допускает сложные вычисления;
- Может использовать все объекты БД, определенные пользователем;
- Прост в использовании.

При работе с данными скорость запросов – один из главных показателей эффективности. Чтобы повысить эту скорость, нужно знать не только как оптимизировать сами запросы, но и как конфигурация самой базы влияет на скорость выполнения запроса.

Данная лабораторная работа призвана сформировать у студента понимание создания алгоритмов для автоматизации процессов в БД и оптимизация запросов.

Лабораторная работа №4. Оптимизация процессов в PostgresSQL

Работа состоит из 2 взаимосвязанных разделов. В каждом разделе есть теоретическая и практическая части лабораторной работы. В конце каждой части есть задания для выполнения в рамках лабораторной работы.

Цель:

- Сформировать у студента понимание методов упрощения работы аналитика с БД.

Задачи:

- Получить теоретические знания о функция.
- Подробнее узнать о языке PL/pgSQL.
- Узнать основу синтаксиса языка.
- Ознакомится с операторами, управляющими конструкциями.
- Получить знания о курсорах и обработки исключений.
- Научится использовать вышеописанные навыки для написания собственной функции.
- Анализ эффективности запросов.
- Узнать о индексах и их влияния на оптимизацию.
- Изучить функции для полнотекстового поиска.

Часть 1. Функции и язык PL/pgSQL

1.1 Теоретическая часть и практические примеры

1.1.1 Функция

С темой облегчения запросов проносится через все части лабораторных работ, одно из таких решений было в создании триггеров для действий с таблицами в БД, там в части с создание функции для работы триггера можно было наблюдать синтаксис языка PL/pgSQL. А сейчас стоит более подробно разобрать эту тему, начнём с разбора примера триггерной функции из предыдущей работы для налогообложения билетов:

```
CREATE OR REPLACE FUNCTION add_tax()
RETURNS TRIGGER AS
$$
BEGIN
    UPDATE ticket_flights SET amount = amount + (amount * 0.36)
    WHERE flight_id = NEW.flight_id;
    RETURN NEW;
END;
$$
LANGUAGE 'plpgsql';
```

Последовательность начинается с ключевого слова CREATE OR REPLACE, можно писать и просто CREATE, но тогда будет невозможно изменять уже созданную, а точнее блок с «$$», то есть часть с алгоритмом функции. (Таким способом нельзя изменить возвращаемое значение или параметры функции, нужно воспользоваться оператором DROP FUNCTION)

Далее после имени пишутся круглые скобки в них при надобности указываются нужные для работы алгоритма переменные (или же параметры, рассмотрим их позже).

Следом описывается возвращаемое значение с помощью RETURNS, в этом случае функция является триггерной, поэтому она возвращает новую запись таблицы (с помощью NEW).

Дальнейшие действия были рассмотрены в предыдущий работе, вкратце создается блок кода (между символами «$$») с алгоритмом функции, который в конце помечается на каком языке это написано (LANGUAGE).

1.1.2 PL/pgSQL и основные операторы, управляющие структуры

Использование процедурных языков на сервере дает ряд преимуществ в дополнение к возможностям чистого SQL.

Можно значительно повысить производительность за счет:

- Меньше команд: исключаются дополнительные обращения между клиентом и сервером
- Меньше пересылки данных: промежуточные ненужные результаты не передаются между сервером и клиентом
- Меньше разборов команд: автоматическое использование подготовленных операторов позволяет избежать многочисленных разборов одного и того же запроса. Однако использование процедурных языков в тех случаях, когда достаточно чистого SQL, может существенно снизить производительность работы приложения.
- Хранимые функции позволяют создавать API для работы с объектами БД. Клиентское приложение, вызывая хранимую функцию, не обязано знать о том, какие действия и с какими объектами нужно выполнить. А права доступа можно настроить так, что пользователи смогут выполнять функции, но не иметь доступа к объектам, с которыми эти функции работают.
- Недоверенные языки позволяют получить доступ к внешним данным из хранимых функций.

Обратная сторона в том, что перенос логики приложения на сервер базы данных усиливает зависимость от используемой СУБД.

PL/pgSQL – это блочно-структурированный язык. В структуре блока можно выделить:

- Необязательную секцию, предназначенную для объявления локальных переменных и курсоров.
- Основную секцию исполнения, в которой располагаются операторы.
- Необязательную секцию обработки ошибок.

В качестве операторов можно использовать команды PL/pgSQL, а также большинство команд SQL. Причем важно, что SQL и PL/pgSQL интегрированы бесшовно: команды SQL используются в блоке напрямую. В качестве оператора может выступать и другой (вложенный) PL/pgSQL блок.

Каждому блоку можно присвоить метку, которая поможет отличить переменную блока от столбца таблицы с таким же именем. Или во вложенном блоке отличить переменные с одинаковыми именами для внешнего и вложенного блоков.

Ознакомится с переменными, операторами и управляющими структурами предлагаю на примере простой задачки для нахождения факториала, выглядеть он будет как-то так:

```
CREATE OR REPLACE FUNCTION factorial(N integer)
RETURNS INTEGER AS
$$
DECLARE -- блок с переменными
	R integer = 1;
	BEGIN
		IF N < 0 THEN -- если пользователь ввел отрицательное число
			RETURN 0; -- возвращаем ноль
		END IF;
		IF N = 0 THEN -- если пользователь ввел ноль,
			RETURN 1; -- то возвращаем единицу
		ELSE -- для остальных случаев
			WHILE N > 0 LOOP
				R = R * N;
				N = N - 1;
			END LOOP;
			RETURN R;
		END IF;
	END;
$$
LANGUAGE 'plpgsql';
```

Первым же отличием от триггерной функции описанной выше является то, что в данном случае присутствует принимаемое значение N, помимо этого поменялось и возвращаемое значение на INTEGER.

Так же появились и новые конструкции такие как DECLARE, он пишется в начале блока с «$$» ниже него определяются (если надо просто выделить память) и инициализируются (если надо задать значение) переменные. (Первым пишется имя, а вторым тип данных)

Ниже идёт сам алгоритм подсчёта факториала, сначала идёт проверка входного значения на отрицательные значения и ноль с помощью условного оператора IF (если) THEN (тогда) ELSE (то). Конструкция RETURN останавливает выполнение функции и возвращает значение.

Далее пользуемся ветвлением и проверяем значение на ноль, если не равно нулю, то пользуемся ключевым слово WHILE с условием что N больше нуля, иначе цикл останавливается, также указываем начало и конец блока цикла с помощью LOOP и END LOOP по аналогии с IF. Для этого на каждой итерации цикла уменьшаем значение N на единицу. А чтобы подчитать факториал умножаем значение R на переменную N. И в конце возвращаем накопившиеся значение R через RETURN.

Чтобы увидеть результат можно воспользоваться таким запросом:

```
SELECT * FROM factorial(8);
```

1.1.3 Курсоры

Курсор в SQL – это временная выборка записей в процессе выполнения функции, над которой могут выполняться необходимые Вам действия, данная выборка является указателем на область памяти.

Курсоры могут быть очень полезны, например, если Вам в функции необходимо выполнять определенные действия с каким-то набором строк, при этом до начала выполнения функции Вы даже не знаете, сколько строк будет при обработке той или иной записи. Если проще курсор – это просто запрос, который запускается в процессе выполнения функции.

Курсор определяется и инициализируется в том же месте, где и обычные переменнные.

```
DECLARE
curs1 refcursor;
```

Прежде чем курсор можно будет использовать для извлечения строк, он должен быть открыт. (Это эквивалентное действие команде SQL ОБЪЯВИТЬ КУРСОР.) PL/pgSQL имеет три формы инструкции OPEN, две из которых используют несвязанные переменные курсора, а третья использует связанную переменную курсора.

```
OPEN curs1 FOR SELECT * FROM ticket_flights WHERE key = mykey;
```

После открытия курсора мы можем манипулировать им с помощью инструкций FETCH, MOVE, UPDATE или DELETE. К примеру, напишем функцию, которая в результате своей работы возвращает курсор:

```
CREATE OR REPLACE FUNCTION return_cursor(mykey integer)
RETURNS refcursor AS
$$
DECLARE -- блок с переменными
        curs1 refcursor := 'curs1';
BEGIN
        OPEN curs1 FOR SELECT * FROM ticket_flights WHERE flight_id = mykey;
        RETURN curs1;
END;
$$
LANGUAGE 'plpgsql';
```

А далее через консоль вызовем её и получим весь результат внутри неё, как на рисунке 1.

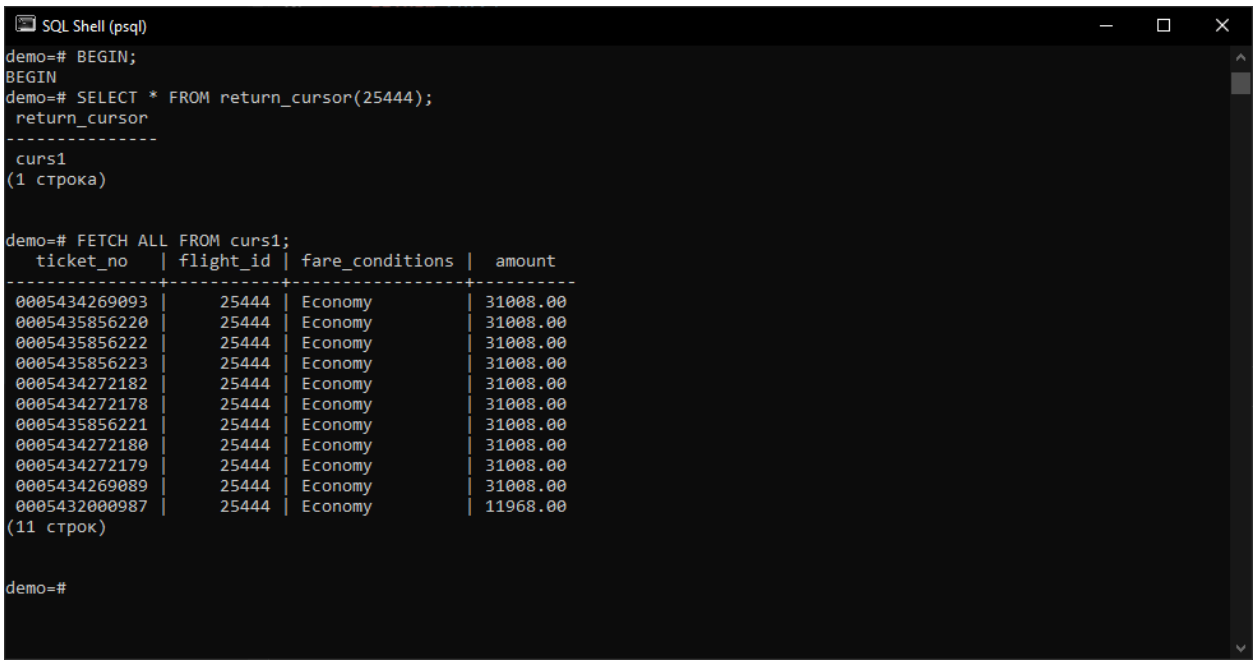

Рисунок 1 – Данные из курсора

1.1.4 Исключения

Во встроенном процедурном языке PL/pgSQL для СУБД PostgreSQL отсутствуют привычные операторы из других ЯП «TRY / CATCH» для перехвата исключений, возникающих в коде во время выполнения. Аналогом является оператор EXCEPTION, который используется в конструкции:

```
BEGIN
-- код, в котором может возникнуть исключение
EXCEPTION WHEN OTHERS -- аналог catch
THEN
-- код, обрабатывающий исключение
END
```

Если необходимо обработать только конкретную ошибку, то в условии WHEN нужно указать идентификатор или код конкретной ошибки:

```
BEGIN
  -- код, в котором может возникнуть исключение
EXCEPTION WHEN '<идентификатор_или_код_ошибки>'
THEN
    -- код, обрабатывающий исключение
END
```

Внутри секции EXCEPTION код ошибки можно получить из переменной SQLSTATE, а текст ошибки из переменной SQLERRM:

```
BEGIN
  -- код, в котором может возникнуть исключение
EXCEPTION WHEN OTHERS
THEN
   RAISE NOTICE 'ERROR CODE: %. MESSAGE TEXT: %', SQLSTATE, SQLERRM;
END
```

К примеру, напишем функцию деления двух чисел, всем известно, что деление на ноль не допустимо, поэтому будем вылавливать это исключение и возвращать значение «-1»:

```
CREATE OR REPLACE FUNCTION exception_func(a INTEGER, b INTEGER)
RETURNS NUMERIC AS
$$
BEGIN
        RETURN a / b;
        EXCEPTION WHEN OTHERS -- аналог catch
                THEN
                        RETURN -1;
END;
$$
LANGUAGE 'plpgsql';
```

В описанной выше функции при делении на ноль будет отрабатывать арифметическая ошибка, что приводит программу на строку с «RETURN -1;», вместо её невыполнения.

1.2 Практическое задание для первой части ЛР

Первая часть заключается в использовании функций. При выполнении задания необходимо:

- Составить SQL-скрипты для создания нескольких функций, упрощающих манипуляции с данными.
- Продемонстрировать полученные знания о возможностях языка PL/pgSQL. В скриптах должны использоваться:
    - Циклы.
    - Ветвления.
    - Переменные.
    - Курсоры.
    - Исключения.
- Обосновать преимущества механизма функций перед механизмом представлений.

Часть 2. Оптимизация запросов. Индексы.

2.1 Теоретическая часть и практические примеры

2.1.1 EXPLAIN и ANALYZE

Чтобы достичь хорошей производительности, этот план должен учитывать свойства данных. Планированием занимается специальная подсистема – планировщик (planner). Просмотреть план выполнения любого запроса можно с помощью команды EXPLAIN. Для детального понимания планов выполнения сложных запросов требуется опыт. Мы изложим лишь основные приемы работы с этой командой.

Структура плана запроса представляет собой дерево, состоящее из так называемых узлов плана (plan nodes). Узлы на нижних уровнях дерева отвечают за просмотр и выдачу строк таблиц, которые осуществляются с помощью методов доступа, описанных выше. Если конкретный запрос требует выполнения операций агрегирования, соединения таблиц, сортировки, то над узлами выборки строк будут располагаться дополнительные узлы дерева плана. Например, для соединения наборов строк будут использоваться способы, которые мы только что рассмотрели. Для каждого узла дерева плана команда EXPLAIN выводит по одной строке, при этом выводятся также оценки стоимости выполнения операций на каждом узле, которые делает планировщик. В случае необходимости для конкретных узлов могут выводиться дополнительные строки. Самая первая строка плана содержит общую оценку стоимости выполнения данного запроса

Запустим командную строку (psql) и введём простой запрос, результат можно наблюдать на рисунке 2:

```
EXPLAIN SELECT * FROM aircrafts;
```

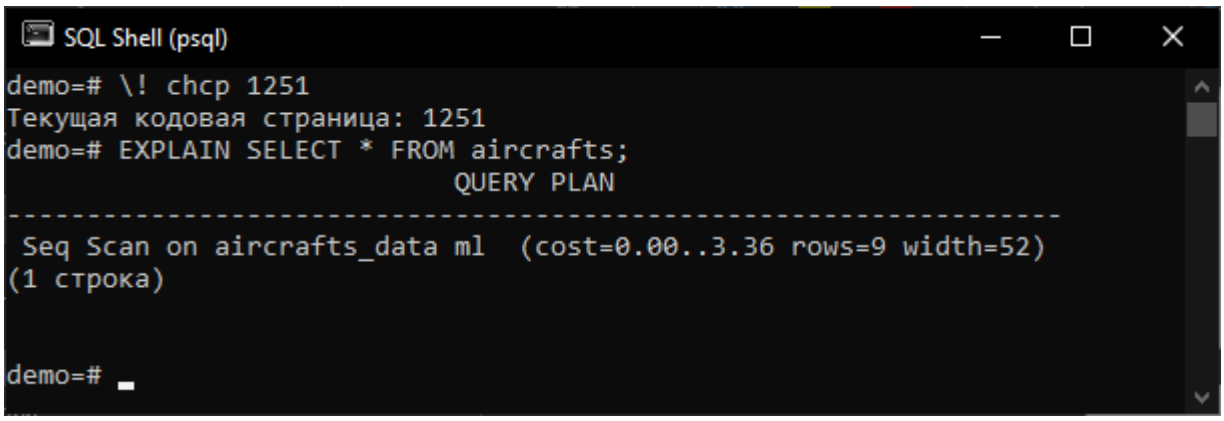

Рисунок 2 – Ответ на запрос

Поскольку в этом запросе нет предложения WHERE, он должен просмотреть все строки таблицы, поэтому планировщик выбирает последовательный просмотр (sequential scan). В скобках приведены важные параметры плана.

Первое число означает оценку ресурсов, требуемых для того, чтобы приступить к выводу данных. В нашем примере эта оценка равна нулю, поскольку никакие дополнительные операции с выбранными строками не предполагаются, и PostgreSQL может сразу же выводить прочитанные строки. Второе число – это оценка общей стоимости выполнения запроса. Формируя эту оценку, планировщик исходит из предположения, что данный узел плана запроса выполняется до конца, т. е. извлекаются все имеющиеся строки таблицы. Однако в ряде случаев на практике это может оказаться и не так, если узел-родитель прекратит свою работу досрочно, например, в случае использования в запросе SELECT предложения LIMIT, которое ограничивает выборку записей из таблицы конкретным их числом. Обе оценки стоимости выполнения выражаются в неких условных единицах, которые вычисляются на основе ряда параметров сервера баз данных. При этом не важно, в каких конкретно единицах производится измерение стоимости: важны соотношения стоимостей.

Далее в выводе идет общее число строк, которые должны быть извлечены (возвращены) на данном узле плана, также при условии выполнения этого узла до полного завершения. В нашем примере число строк

равно 9. Это число является оценкой, которую планировщик получает на основе статистики, накапливаемой в специальных системных таблицах. Последним параметром узла плана идет оценка среднего размера строк, которые выводятся на данном узле плана запроса. В нашем примере размер (ширина) строки данных оценивается в 52 байта.

В том случае, когда нас не интересуют численные оценки, можно воспользоваться параметром COSTS OFF:

```
EXPLAIN ( COSTS OFF ) SELECT * FROM aircrafts;
```

Сформируем запрос с предложением WHERE:

```
EXPLAIN SELECT * FROM aircrafts WHERE model ~ 'Air';
```

Поскольку наложено дополнительное условие на строки, выбираемые из таблицы, то ниже узла плана, отвечающего за их последовательную выборку, добавляется еще один узел, описывающий критерий отбора строк.

Обратите внимание, что по своей форме вывод команды EXPLAIN также является выборкой, поэтому в конце выборки, как обычно, выводится информация о числе строк в ней, т. е. в дереве плана. Это не число строк, которые будут выбраны из таблицы

Теперь усложним запрос, добавив в него сортировку данных:

```
EXPLAIN SELECT * FROM aircrafts ORDER BY aircraft_code;
```

Дополнительный узел обозначен на плане символами «->». Хотя по столбцу aircraft_code создан индекс (для поддержки первичного ключа), планировщик предпочел не использовать этот индекс, а прибегнуть к последовательному сканированию (Seq Scan) таблицы, о чем говорит нам нижний узел плана. На верхнем узле выполняется сортировка выбранных строк. Поскольку для выполнения сортировки требуется время, отличное от нуля, то этот факт и отражен в первой числовой оценке – 1,23. Это оценка времени, которое потребуется для того, чтобы приступить к выводу отсортированных строк. Но времени непосредственно на саму сортировку

потребуется меньше: ведь в оценку 1,23 входит и оценка стоимости получения выборки – 1,09. Когда таблица очень маленькая, то обращение к индексу не даст выигрыша в скорости, а лишь добавит к операциям чтения страниц, в которых хранятся строки таблиц, еще и операции чтения страниц с записями индекса.

Для управления планировщиком предусмотрен целый ряд параметров. Их можно изменить на время текущего сеанса работы с помощью команды SET. Конечно, изменять параметры в производственной базе данных следует только в том случае, когда вы обоснованно считаете, что планировщик ошибается. Однако для того, чтобы научиться видеть ошибки планировщика, нужен большой опыт. Поэтому следует рассматривать приведенные далее команды управления планировщиком лишь с позиции изучения потенциальных возможностей управления им, а не как рекомендацию к бездумному изменению этих параметров в реальной работе.

В команде EXPLAIN можно указать опцию ANALYZE, что позволит выполнить запрос и вывести на экран фактические затраты времени на выполнение запроса и число фактически выбранных строк. При этом, хотя запрос и выполняется, сами результирующие строки не выводятся.

Сначала разрешим планировщику использовать метод соединения слиянием:

```
SET enable_mergejoin = on;
```

И напишем такой запрос:

```
EXPLAIN SELECT t.ticket_no, t.passenger_name, tf.flight_id, tf.amount
FROM tickets t
JOIN ticket_flights tf ON t.ticket_no = tf.ticket_no
ORDER BY t.ticket_no;
```

Фактические затраты времени измеряются в миллисекундах, а оценки стоимости – в условных единицах, поэтому плановые оценки и фактические

затраты совпасть не могут. Важнее обратить внимание на то, насколько точно планировщик оценил число обрабатываемых строк, а также на фактическое число повторений того или иного узла дерева плана – это параметр loops. В данном запросе каждый узел плана был выполнен ровно один раз, поскольку выбор строк из обоих соединяемых наборов производился по индексу, поэтому достаточно одного прохода по каждому набору. Число выбираемых строк было оценено точно, поскольку таблицы связаны по внешнему ключу, и в выборку включаются все их строки (нет предложения WHERE).

Если модифицировать запрос, добавив предложение WHERE, то точного совпадения оценки числа выбираемых строк и фактического их числа уже не будет.

```
EXPLAIN ANALYZE
SELECT t.ticket_no, t.passenger_name, tf.flight_id, tf.amount
FROM tickets t
JOIN ticket_flights tf ON t.ticket_no = tf.ticket_no
WHERE amount > 50000
ORDER BY t.ticket_no;
```

План выполнения запроса изменился. Метод соединения наборов строк остался прежним – слияние. Но выборка строк в нижнем узле дерева плана теперь выполняется с помощью последовательного сканирования и сортировки. Обратите внимание, что при включении опции ANALYZE может выводиться дополнительная информация о фактически использовавшихся методах, о затратах памяти и др. В частности, сказано, что была использована внешняя сортировка на диске (Sort Method), приведены затраты памяти на ее выполнение, приведено число строк, удаленных при проверке условия их отбора (Rows Removed by Filter).

Обратимся еще раз к запросу, который мы уже рассматривали выше, и выполним его с опциями ANALYZE и COSTS OFF (для сокращения вывода). В плане этого запроса нас будет интересовать фактический параметр loops.

```
EXPLAIN (ANALYZE, COSTS OFF)
SELECT a.aircraft_code, a.model, s.seat_no, s.fare_conditions
FROM seats s
JOIN aircrafts a ON s.aircraft_code = a.aircraft_code
WHERE a.model ~ '^Air'
ORDER BY s.seat_no;
```

Как видно из плана, значение параметра loops для узла, выполняющего сканирование таблицы seats по индексу с построением битовой карты, равно трем. Это объясняется тем, что из таблицы aircrafts были фактически выбраны три строки, и для каждой из них выполняется поиск в таблице seats. Для подсчета общих затрат времени на выполнение операций сканирования по индексу за три цикла нужно значение параметра actual time умножить на значение параметра loops. Таким образом, для узла дерева плана Bitmap Index Scan получим 0,064 × 3 = 0,192.

Подобные вычисления общих затрат времени на промежуточных уровнях дерева плана могут помочь выявить наиболее ресурсоемкие операции. Попутно заметим, что, согласно этому плану, сортировка на верхнем уровне плана выполнялась в памяти с использованием метода quicksort.

2.1.2 Индексы

Индексы позволяют повысить производительность базы данных. PostgreSQL поддерживает различные типы индексов. Мы ограничимся рассмотрением только индексов на основе B-дерева. Индекс — специальная структура данных, которая связана с таблицей и создается на основе данных, содержащихся в ней. Основная цель создания индексов — повышение производительности функционирования базы данных.

Строки в таблицах хранятся в неупорядоченном виде. При выполнении операций выборки, обновления и удаления СУБД должна отыскать нужные строки. Для ускорения этого поиска и создается индекс. В принципе он организован таким образом: на основе данных, содержащихся в конкретной строке таблицы, формируется значение элемента (записи) индекса, соответствующего этой строке. Для поддержания соответствия между элементом индекса и строкой таблицы в каждый элемент помещается указатель на строку. Индекс является упорядоченной структурой. Элементы (записи) в нем хранятся в отсортированном виде, что значительно ускоряет поиск данных в индексе. После отыскания в нем требуемой записи СУБД переходит к соответствующей строке таблицы по прямой ссылке. Записи индекса могут формироваться на основе значений одного или нескольких полей соответствующих строк таблицы. Значения этих полей могут комбинироваться и преобразовываться различными способами. Все это определяет разработчик базы данных при создании индекса.

Для того чтобы увидеть индексы, созданные для данной таблицы, нужно воспользоваться командой утилиты psql, где «boarding_passes» имя таблицы, рисунок 3:

```
\d boarding_passes
```

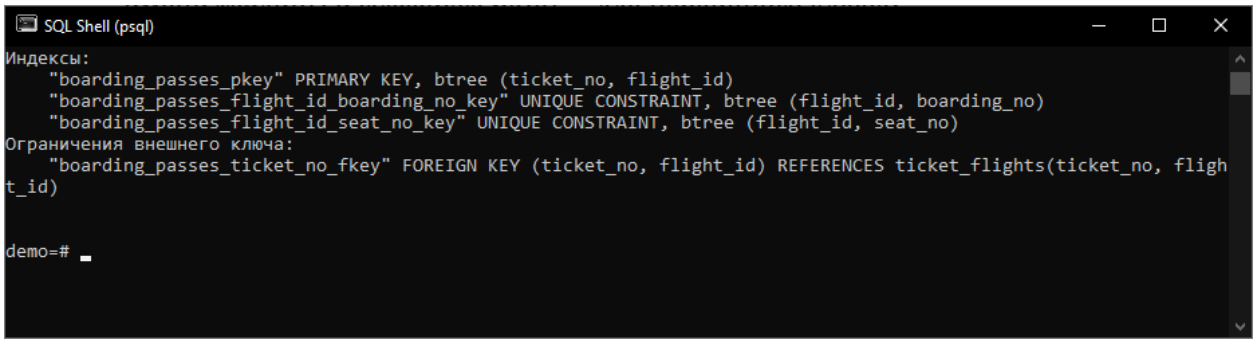

Рисунок 3 – Индексы

Каждый индекс, который был создан самой СУБД, имеет типовое имя, состоящее из следующих компонентов: – имени таблицы и суффикса pkey —

для первичного ключа; – имени таблицы, имен столбцов, по которым создан индекс, и суффикса key — для уникального ключа.

В описании также присутствует список столбцов, по которым создан индекс, и тип индекса — в данном случае это btree, т. е. B-дерево. PostgreSQL может создавать индексы различных типов, но по умолчанию используется так называемое B-дерево. Такой индекс подходит для большинства типовых задач. В этой главе мы будем рассматривать только индексы на основе B-дерева. Наличие индекса может ускорить выборку строк из таблицы, если он создан по столбцам, на основе значений которых и производится выборка. Поэтому, как правило, при разработке и эксплуатации баз данных не ограничиваются только индексами, которые автоматически создает СУБД, а создают дополнительные индексы с учетом наиболее часто выполняющихся выборок. Для создания индексов предназначена команда, на примере таблицы с аэропорта:

```
CREATE INDEX ON airports_data ( airport_name );
```

Посмотрим описание нового индекса, рисунок 4:

```
\d airports
```

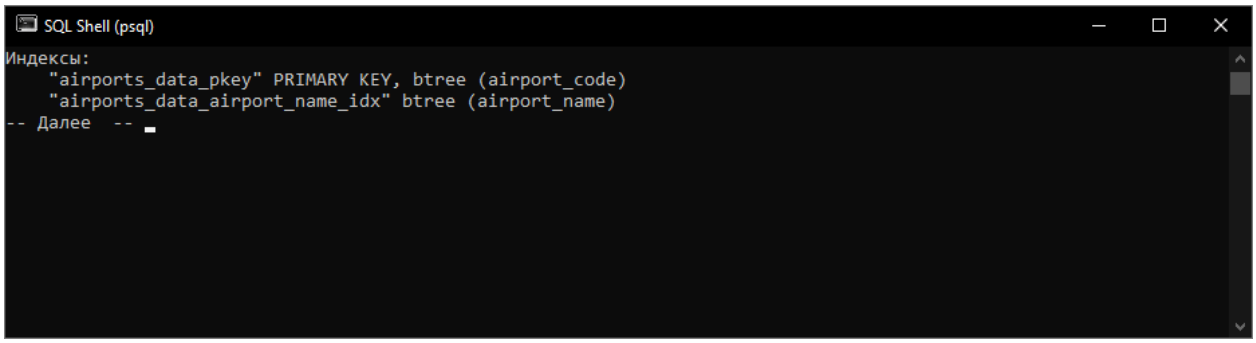

Рисунок 4 – Новый индекс

Обратите внимание, что имя индекса, сформированное автоматически, включает имя таблицы, имя столбца и суффикс idx.

Прежде чем приступить к экспериментам с индексами, нужно включить в утилите psql секундомер с помощью следующей команды:

```
\timing on
```

Когда необходимость в использовании секундомера отпадет, для его отключения нужно будет сделать так:

```
\timing off
```

Теперь psql будет сообщать время, затраченное на выполнение всех команд. Для практической проверки влияния индекса на скорость выполнения выборок сначала выполним следующий запрос:

```
SELECT count( * ) FROM tickets WHERE passenger_name = 'IVAN IVANOV';
```

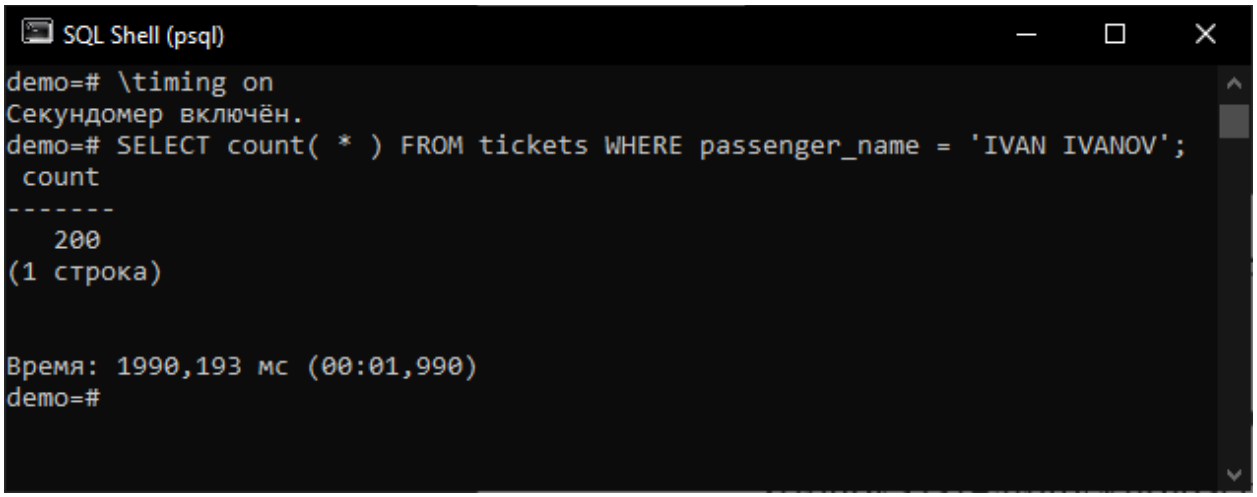

Рисунок 5 – Время без индекса

Показатели времени, полученные на вашем компьютере, конечно, будут отличаться от значений, приведенных в книге, и — возможно — значительно. Эти показатели нужно рассматривать лишь как качественные ориентиры.

Создадим индекс по столбцу passenger_name, при этом никакого суффикса в имени индекса использовать не будем, поскольку его наличие не является обязательным:

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

Теперь выполним ту же выборку из таблицы tickets:

```
SELECT count( * ) FROM tickets WHERE passenger_name = 'IVAN IVANOV';
```

Теперь видно, что время выполнения выборки при наличии индекса оказалось значительно меньше, рисунок 6.

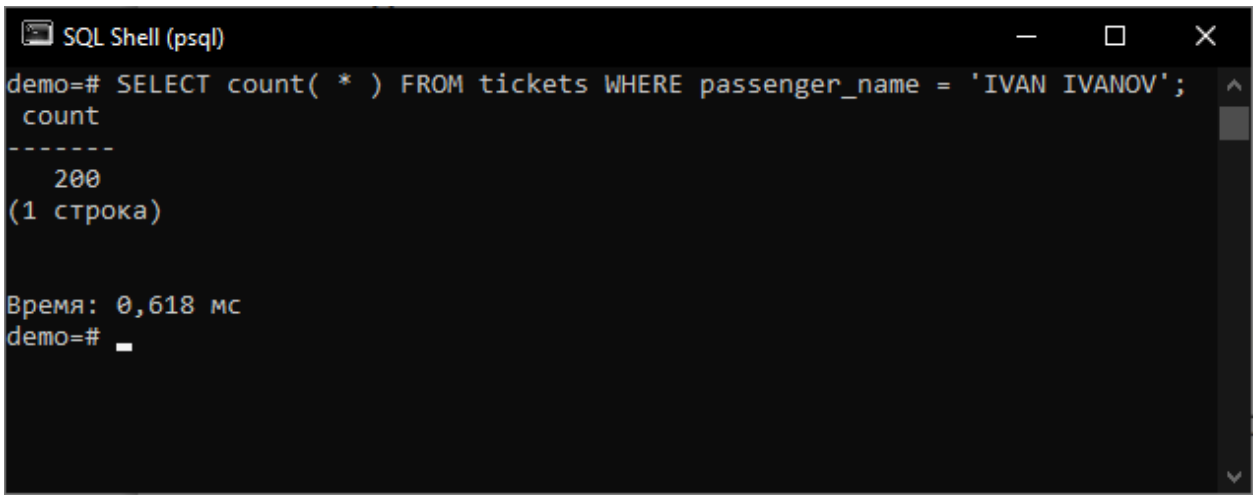

Рисунок 6 – Время с индексом

Для удаления индекса используется команда:

```
DROP INDEX passenger_name;
```

Когда индекс уже создан, о его поддержании в актуальном состоянии заботится СУБД. Конечно, следует учитывать, что это требует от СУБД затрат ресурсов и времени. Индекс, созданный по столбцу, участвующему в соединении двух таблиц, может позволить ускорить процесс выборки записей из таблиц. При выборке записей в отсортированном порядке индекс также может помочь, если сортировка выполняется по тем столбцам, по которым индекс создан.

2.1.3 Полнотекстовый поиск

Полнотекстовый поиск (или просто поиск текста) — это возможность находить документы на естественном языке, соответствующие запросу, и, возможно, дополнительно сортировать их по релевантности для этого запроса.

Наиболее распространённая задача — найти все документы, содержащие слова запроса, и выдать их отсортированными по степени соответствия запросу. Понятия запроса и соответствия довольно расплывчаты и зависят от конкретного приложения. В самом простом случае запросом считается набор слов, а соответствие определяется частотой слов в документе. В PostgreSQL есть операторы ~, ~*, LIKE и ILIKE, но их возможностей недостаточно.

Полнотекстовая индексация заключается в предварительной обработке документов и сохранении индекса для последующего быстрого поиска. • Предварительная обработка включает следующие операции:

- Разбор документов на фрагменты – числа, слова, словосочетания, почтовые адреса и т. д., которые будут обрабатываться по-разному;
- Преобразование фрагментов в лексемы. Лексема — это нормализованный фрагмент, в котором разные словоформы приведены к одной (например, буквы верхнего регистра приводятся к нижнему, а из слов обычно убираются окончания, исключаются стоп-слова). Для выполнения этого шага в PostgreSQL используются словари;
- Хранение документов в форме, подготовленной для поиска. Помимо лексем часто желательно хранить информацию об их положении для ранжирования по близости;

Для хранения подготовленных документов в PostgreSQL предназначен тип данных tsvector, а для представления обработанных запросов — тип tsquery. Поиск и ранжирование документов выполняется исключительно с представлением документа в формате tsvector. Исходный текст потребуется извлечь, только когда документ будет отобран для вывода пользователю.

Функция to_tsvector может разобрать и нормализовать текстовое содержимое документа, принцип работы изображен на рисунке 7.

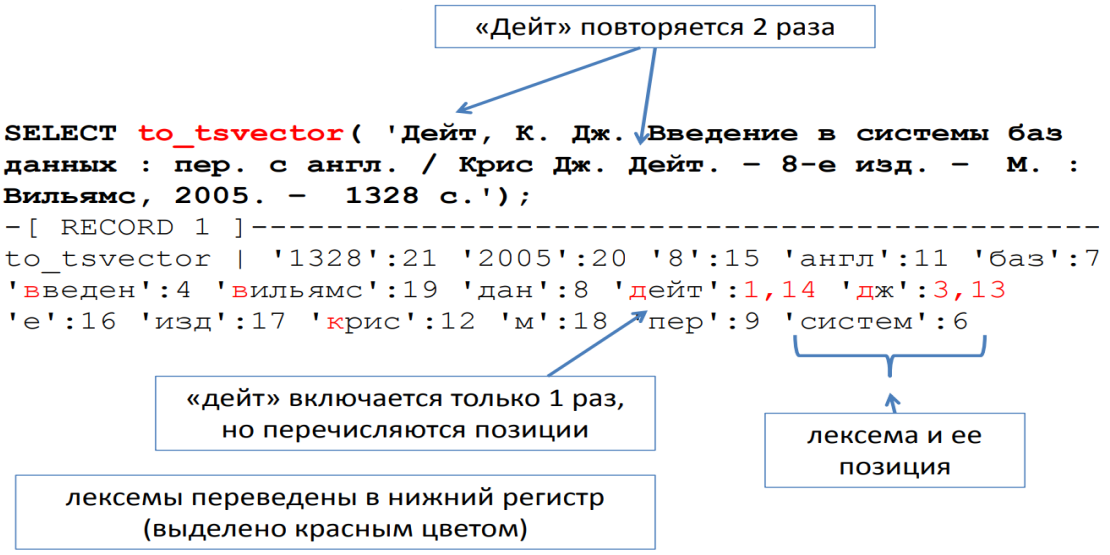

Рисунок 7 – to_tsvector

Можно воспользоваться функциями to_tsquery и plainto_tsquery для преобразования заданного пользователем текста, который содержит слова для поиска, в значение tsquery. Они нормализуют слова в этом тексте, рисунок 8.

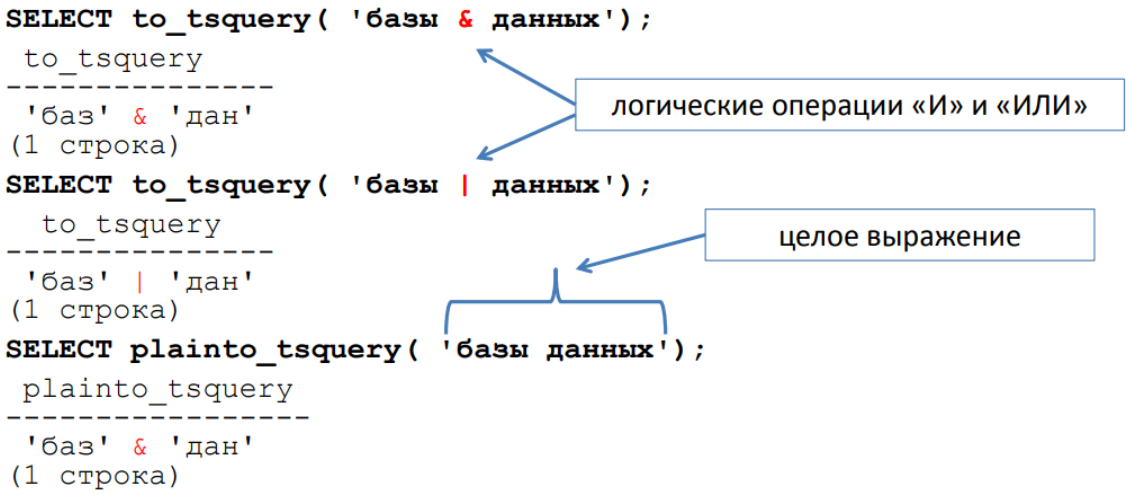

Рисунок 8 – to_tsquery, plainto_tsquery

Используя to_tsvector напишем выборку с общим табличным выражением:

```
WITH books ( description ) AS
( VALUES ( 'Дейт, К. Дж. Введение в системы баз данных :
 пер. с англ. / Крис Дж. Дейт. – 8-е изд. –
 М. : Вильямс, 2005. – 1328 с.' ),
 ( 'Грофф, Дж. SQL. Полное руководство : пер. с
 англ. / Джеймс Р. Грофф, Пол Н. Вайнберг,
 Эндрю Дж. Оппель. – 3-е изд. – М. : Вильямс,
 2015. – 960 с.' ),
 ( 'Лузанов, П. PostgreSQL для начинающих /
 П. Лузанов, Е. Рогов, И. Лёвшин ; Postgres
 Professional. – М., 2017. – 146 с.' )
),
books_2 ( description, ts_description ) AS
( SELECT description, to_tsvector( description )
 FROM books )
SELECT * FROM books_2;
```

Для демонстрации «to_tsquery» изменим нижнюю часть запроса, поскольку запрос в CTE не изменяется:

```
WITH books ( description ) AS
( VALUES ( 'Дейт, К. Дж. Введение в системы баз данных :
 пер. с англ. / Крис Дж. Дейт. – 8-е изд. –
 М. : Вильямс, 2005. – 1328 с.' ),
 ( 'Грофф, Дж. SQL. Полное руководство : пер. с
 англ. / Джеймс Р. Грофф, Пол Н. Вайнберг,
 Эндрю Дж. Оппель. – 3-е изд. – М. : Вильямс,
 2015. – 960 с.' ),
 ( 'Лузанов, П. PostgreSQL для начинающих /
 П. Лузанов, Е. Рогов, И. Лёвшин ; Postgres
 Professional. – М., 2017. – 146 с.' )
),
books_2 ( description, ts_description ) AS
( SELECT description, to_tsvector( description )
 FROM books )
SELECT description
FROM books_2
WHERE ts_description @@ to_tsquery( 'SQL' );
```

Выборка с помощью функции plainto_tsquery может выглядить так:

```
WITH books ( description ) AS
( VALUES ( 'Дейт, К. Дж. Введение в системы баз данных :
 пер. с англ. / Крис Дж. Дейт. – 8-е изд. –
 М. : Вильямс, 2005. – 1328 с.' ),
 ( 'Грофф, Дж. SQL. Полное руководство : пер. с
 англ. / Джеймс Р. Грофф, Пол Н. Вайнберг,
 Эндрю Дж. Оппель. – 3-е изд. – М. : Вильямс,
 2015. – 960 с.' ),
```

```
 ( 'Лузанов, П. PostgreSQL для начинающих /
 П. Лузанов, Е. Рогов, И. Лёвшин ; Postgres
 Professional. – М., 2017. – 146 с.' )
),
books_2 ( description, ts_description ) AS
( SELECT description, to_tsvector( description )
 FROM books )
SELECT description
FROM books_2
WHERE ts_description @@ plainto_tsquery( 'базы данных' );
```

2.2 Практическое задание для второй части ЛР

Девятое практическое задание посвящено ускорению выполнения запросов. Для этого могут быть использованы механизмы секционирования, наследования и индексов. Для выполнения задания необходим достаточно большой объем данных, чтобы оптимизация была целесообразной (порядка 1 млн. строк в каждой таблице).

Необходимо подготовить два запроса:

- Запрос к одной таблице, содержащий фильтрацию по нескольким полям.
- Запрос к нескольким связанным таблицам, содержащий фильтрацию по нескольким полям.

Для каждого из этих запросов необходимо провести следующие шаги:

- Получить план выполнения запроса без использования индексов.
- Получить статистику (IO и Time) выполнения запроса без использования индексов.
- Создать нужные индексы, позволяющие ускорить запрос.
- Получить план выполнения запроса с использованием индексов и сравнить с первоначальным планом.

- Получить статистику выполнения запроса с использованием индексов и сравнить с первоначальной статистикой.
- Оценить эффективность выполнения оптимизированного запроса.

Также необходимо продемонстрировать полезность индексов для организации полнотекстового поиска, фильтрации с использованием массива и json-формата.

Для таблицы объемом больше 100 млн. записей произвести оптимизацию, позволяющую быстро удалять старые данные, ускорить вставку и чтение данных.

3 Темы для самостоятельной проработки

- Функции

https://postgrespro.ru/docs/postgrespro/11/xfunc-sql

- PL/PgSQL

https://postgrespro.ru/docs/postgrespro/11/plpgsql

- Основные операторы

https://postgrespro.ru/docs/postgrespro/11/plpgsql-statements

- Управляющие структуры

https://postgrespro.ru/docs/postgrespro/11/plpgsql-control-structures

- Курсоры

https://postgrespro.ru/docs/postgrespro/11/plpgsql-cursors

- Полное описание синтаксиса встретившихся команд

https://postgrespro.ru/docs/postgrespro/11/sql-commands

- EXPLAIN

https://postgrespro.ru/docs/postgrespro/11/using-explain

https://postgrespro.ru/docs/postgrespro/11/planner-stats

https://postgrespro.ru/docs/postgrespro/11/explicit-joins

- ANALYZE

https://postgrespro.ru/docs/postgrespro/11/routine-vacuuming#VACUUM-FOR-STATISTICS

- Индексы

https://postgrespro.ru/docs/postgrespro/11/indexes

- Полнотекстовый поиск

https://postgrespro.ru/docs/postgrespro/11/textsearch

- Наследование таблиц

https://postgrespro.ru/docs/postgrespro/11/ddl-inherit

- Секционирование таблиц

https://postgrespro.ru/docs/postgrespro/11/ddl-partitioning

- Полное описание синтаксиса встретившихся команд

https://postgrespro.ru/docs/postgrespro/11/sql-commands

4 Вопросы для самостоятельного изучения

- Объяснить, как работают написанные запросы.
- Исправить неверно работающий запрос (запросы).
- Упростить один или несколько запросов.
- Написать или модифицировать запрос по сформулированному заданию.
- Описать в каких случаях целесообразно создавать функции.
- Рассказать о курсорах, как и зачем используются.
- Рассказать о работе с циклами.
- В чем отличие первичного ключа и уникального индекса?
- В каких случаях имеет смысл создавать индексы? Какие колонки следует включать в индекс и почему?
- Какие существуют способы внутренней организации индексов?
- Рассказать о проблеме фрагментации индексов. Как бороться с фрагментацией?
- Имеет ли значение порядок указания колонок при создании индекса?
- В чем разница между Index Scan и Index Seek?
- В чем разница между секционированием и наследованием?
- Зачем нужен ANALYZE?
- Исправить ошибки в подготовленных выборках.
- Могут ли индексы ухудшить производительность? Если да, то продемонстрировать это.
- На что влияет порядок сортировки (ASC\DESC) при создании индекса? Продемонстрировать это.
- Продемонстрировать полезность индекса по выражению.
- Продемонстрировать полезность частичного индекса.

5 СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Postgres Professional / Егор Рогов – М., 2017 – 5 с
2. PostgreSQL. Основы языка SQL: учеб. пособие / Е. П. Моргунов; под ред. Е. В. Рогова, П. В. Лузанова. – СПб.: БХВ-Петербург, 2018. – 296-302 с.
3. Язык SQL Лекция 10 Полнотекстовый поиск / Е. П. Моргунов – М., 2018 – 1-15 с.
4. Info-comp [Электронный ресурс] – Режим доступа: URL: https://info-comp.ru/obucheniest/254-cursor-in-functions.html
5. Habr [Электронный ресурс] – Режим доступа: URL: https://habr.com/ru/post/657667/