Министерство образования и науки Российской Федерации
Санкт-Петербургский политехнический университет Петра Великого
Институт компьютерных наук и технологий
Высшая школа киберфизических систем и управления

Работа допущена к защите

Руководитель ОП

___________ А.А. Ефремов

«___»_______________2018 г.

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА БАКАЛАВРА

Анализ и модификация информационной системы складского учета с целью повышения скорости выполнения запросов

по направлению 09.03.02 Информационные системы и технологии

Выполнил
студент гр. в43503/6 П.Ю. Алексеев

Руководитель
к.т.н., доц. С.А. Нестеров

Санкт-Петербург

2018

РЕФЕРАТ

На 46 с., 3 рис., 3 табл., 1 прил.

Ключевые слова: БАЗА ДАННЫХ, ЗАПРОС, ИНДЕКС, Б-ДЕРЕВО, ОРГАНИЗАЦИЯ ПОИСКА, POSTGRESQL

Объектом исследования является система складского учёта, использующая PostgreSQL 10 в качестве СУБД. Целью работы является иллюстрация основных методов сбора и анализа статистики СУБД, проведения анализа медленных запросов, а также анализ и использование индексов для ускорения поиска. В процессе работы были подробно рассмотрены планы выполнения запросов, создания индекса на основе планов и была ускорена работы системы.

ABSTRACT

46 pages, 3 pictures, 3 tables, 1 appendix

Key words: DATABASE, QUERY, INDEX, B-TREE, SEARCH ORGANIZATION, POSTGRESQL

The object of research is inventory management system that uses PostgreSQL 10 as DBMS. The aim of work is to illustrate main methods of collecting and analyzing DBMS usage statistics, analysis of slow queries, also analysis and usage of indexes to increase search queries. In the course of work, query plans were analyzed, indexes were created based on those plans and system speed was increased.

Оглавление

Введение.. 4

Глава 1. Анализ предметной области.. 5

1.1.Постановка задачи 5

1.2.Поиск возможных инструментов 5

1.2.1.Сбор статистики 5

1.2.2.Анализ статистики 7

1.2.3.Анализ запросов 8

1.2.4.Индексы 13

1.3.Выводы и уточненная постановка задачи 21

Глава 2. Анализ системы. 23

2.1. Описание складской системы. 23

2.2. Проведения тестирования и анализ результатов 24

2.3. Запрос степени выполнения документов. 26

2.4. Запрос остатка незарезервированного товара на складе 34

2.5. Выводы 38

Глава 3. Модификация системы и анализ результата 39

3.1. Создание индексов 39

3.2. Тестирование системы после модификации 40

Заключение 43

Список литературы 44

Приложение 1 45

Введение

В настоящее время базы данных используются очень широко. Если у приложения есть необходимость сохранять и читать данные во время работы, то чаще всего наиболее оправданным решением будет использование базы данных. Существуют несколько типов баз данных, и много решений от разных компаний под разные потребности. Наиболее используемыми являются реляционные базы данных, основанные на реляционной модели баз данных, для работы с которыми используют реляционные СУБД.

Целью данной работы является анализ и модификация складской системы с целью повышения скорости выполнения аналитических запросов.

В рамках данной работы будут решены следующие задачи:

- обзор возможностей СУБД по предоставлению статистики её быстродействия

- обзор инструментов анализа статистики быстродействия с целью поиска медленных аналитических запросов

- анализ планировщика запросов СУБД для разбора причины медленного выполнения запроса

- обзор и анализ индексов для быстрого поиска информации в СУБД

- экспериментальная часть, иллюстрирующая анализ и модификацию системы на основе указанных методов.

Глава 1. Анализ предметной области

1.1.Постановка задачи

В данной работе будет рассмотрена и модифицирована система складского учёта, использующая в качестве СУБД PostgreSQL 10. Данная система обрабатывает все необходимые действия, проводимые с содержимым на складе, в том числе: поставка товара на склад, проверка наличия, резервирование, отгрузка товара, оптимизация использования свободных ячеек и отчёты по использованию склада.

Необходимо ускорить создание отчётов по использованию склада, а в этой проблеме подавляющей количество времени занимает выполнение соответствующего аналитического запроса. При этом необходимо чтобы меры предпринятые для ускорения аналитических запросов не привели к критическим замедлениям в той части системы, которая отвечает за обработку складских операций в реальном времени.

Соответственно необходимо проанализировать возможные решения, которые могут ускорить аналитические запросы, оценить их применимость в данной системе, эффективность и побочные эффекты.

Целью работы является выбор одного или комбинации решений, которые дадут наибольший прирост в скорости выполнения аналитических запросов с наименьшим числом побочных эффектов.

1.2.Поиск возможных инструментов

Для поиска нужных инструментов в СУБД обратимся к официальной документации.

1.2.1.Сбор статистики

В первую очередь необходимо собрать статистику.

Для этого необходимо включить логирование событий в настройках сервера.[1]

Чтобы сделать это, нужно установить значения

```
logging_collector = on
log_directory = '<путь для сохранения логов>'
log_destination = 'csvlog'
```

Первая настройка включает логирование, вторая указывает путь для сохранения файлов логов. PostgreSQL поддерживает несколько форматов логирования, включая stderr, csvlog и syslog. На Windows системах также доступен eventlog. Мы выберем csvlog как формат, который можно с удобством прочитать множеством программ, за это отвечает третья по порядку настройку.

Затем нужно включить логирование именно запросов.

```
log_min_duration_statement = 0
log_statement = 'mod'
```

Первая настройка отвечает за то, какой минимальной длительности запросы нужно записывать в лог. Значение -1 выключает логирование запросов, значение 0 включает логирование всех выполняемых запросов, значение больше нуля это длительность в миллисекундах, дольше которого должен выполняться запрос чтобы он был записан в лог. Для проведения нагрузочного тестирования во время которого мы проведём анализ, установим это значение в 0, чтобы увидеть полную картину выполнения запросов. Для уже оптимизированной системы это значение рекомендуется устанавливать более 1000, чтобы не заполнять лог множеством уже оптимизированных запросов, а только видеть медленные запросы, оптимизацию которых не провели. Вторая настройка отвечает за тип запросов, которые будут логироваться. Возможные варианты это ddl, mod, all. ddl это только DDL запросы, mod включает в себя ещё и DML, all это все запросы. Нам в первую очередь интересны DML запросы, поэтому выбираем mod.

1.2.2.Анализ статистики

После сбора статистики нужно её интерпретировать. Лог последовательных событий хоть и содержит всю исходную информацию, но не удобен для понимания. Мы сохранили лог в формате csv, поэтому одна из возможностей это создать таблицу в базе данных и загрузить туда лог. Это даёт возможность удобного поиска по нему, но не даёт возможность агрегировать статистику, из-за минимальных отличий в запросах.

Поэтому обратимся к стороннему программному обеспечению. Существуют несколько программ, которые в качестве входных данных принимают лог PostgreSQL и на выходе могут сгенерировать отчёт, удобный для анализа специалисту. Такие как pglogcheck, PgFouine, pgBadger. Выберем pgBadger – он является лидирующим анализатором логов для PostgreSQL, активно развивается и имеет открытый исходный код.[2]

pgBadger представляет собой консольную утилиту, написанную на Perl и использует JavaScript библиотеки для построения графиков, поэтому установка дополнительных модулей не нужна. Чтобы построить отчёт с параметрами по умолчанию, достаточно передать pgBadger путь к лог-файлу в качестве параметра, и pgBadger запишет результат в файл out.html, который можно посмотреть в любом актуальном браузере с поддержкой JavaScript.

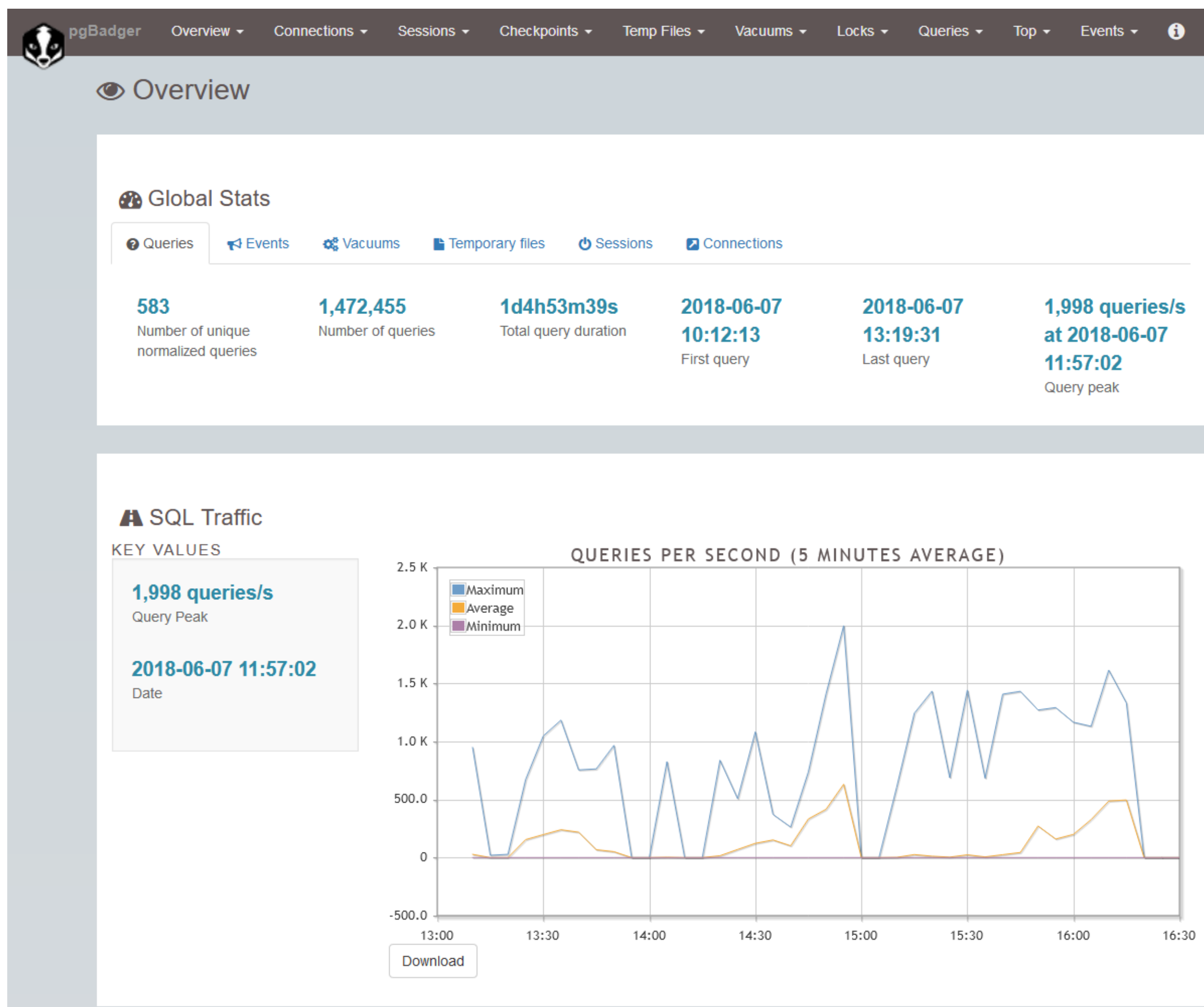

Рисунок 1.1 Пример главной страницы отчёта pgBadger

Нас больше всего будет интересовать вкладка «Top» - там находится статистика наиболее часто выполняемых запросов, самых долго выполняющихся запросов, запросов, которые суммарно выполнялись наибольшее число времени.

1.2.3.Анализ запросов

Для анализа запросов в СУБД есть команда, предоставляющая для запроса план запроса, по которому СУБД будет выполнять запрос. PostgreSQL тому не исключение. Хороший планировщик запросов это критически важная часть СУБД, потому что чем лучше выбран план, тем быстрее выполнится запрос.[3]

Команда EXPLAIN возвращает предполагаемый план запроса, который система использовала бы в данный момент при выполнении этого запроса. Ре-

зультат может возвращаться как в текстовом виде, предназначенном для чтения специалистом, так и в отформатированном виде (XML, JSON, YAML) для передачи плана другой программе.

План состоит из дерева узлов (plan nodes). Узлы нижнего уровня это узлы сканирования (scan nodes), они возвращают необработанные данные из таблицы. Есть несколько главных видов узлов сканирования для разных видов доступа к данным из таблицы.[4]

Последовательное сканирование (sequential scan) – это последовательное чтение строк данных из таблицы

Сканирование с индексом (index scan) – это сканирование с использованием индекса. Сначала данные фильтруются с использованием индекса, затем по индексу выбираются необходимые строки из таблицы.

Сканирование с битовой картой по индексам (bitmap index scan) – это сканирование, при котором на основе индекса (или индексов) формируется битовая карта, и уже по ней выбираются строки таблицы.

Каким видом сканирования пользоваться, планировщик запросов решает на основе автоматически собираемой о таблице статистике. Главными критериями является предполагаемое число строк, которое должен вернуть узел и наличие или отсутствие индексов.

Если запрос требует соединения с другими таблицами, агрегацию, сортировку или другие операции над полученными строками, то над узлами сканирования будут узлы соответствующие этим операциями. Эти операции тоже можно сделать по-разному, поэтому разные типы узлов могут быть в плане для одних операций. Вывод команды EXPLAIN состоит из одной строчки на каждый узел. В строчке указан базовый тип узла и дополнительная информация, такая как предположительная оценка ресурсоёмкости на выполнение этого узла.

Оценка ресурсоёмкости (cost) производится по шкале, которая определяется в зависимости от разных факторов. По умолчанию, она напрямую связана со

ресурсоёмкостью необходимой на получение последовательных блоков страниц БД. Этот параметр задаётся в настройках и называется seq_page_cost, значение по умолчанию установлено в 1.0. При желании оценку ресурсоёмкости можно перевести на зависимость от другой шкалы.

Дополнительная информация состоит из следующих факторов:

- предположительная оценка начальной трудоёмкости узла, например сортировка в узле сортировки

- предположительная оценка общей трудоёмкости узла. Она рассчитывается с учётом полного выполнения плана узла, т.е. возврата всех строк. В некоторых случаях выполнения может преждевременно прекратится, например, если в запросе есть ограничение строк, заданное командой LIMIT

- предположительное количество строк, которое вернёт узел. Опять же, с учётом полного выполнения плана узла.

- предположительная средняя длина возвращаемых строк в байтах

Необходимо упомянуть следующие моменты.

Во-первых, оценка трудоёмкости узла, включает в себя оценку на выполнение нижестоящих узлов. Во-вторых, оценка трудоёмкости отражает только те вещи, которые важны с точки зрения построения плана запроса. Например, она не включает в себя оценку времени, которая необходимо на передачу строк клиенту, а это время может быть важным фактором общего времени выполнения запроса. Но т.к. планировщик не может повлиять на это, то он игнорирует эти затраты.

Также оценка количества строк это не оценка количества обработанных или отсканированных строк, а оценка строк, которые узел должен вернуть. Это количество зачастую меньше чем количество отсканированных строк из-за фильтрации WHERE условий, применяемых в этом узле. Условия из WHERE отображаются в плане как условия фильтрования (Filter).

Есть возможность проверить точность оценок планировщика, используя опцию ANALYZE команды EXPLAIN. С этой опцией EXPLAIN после построение плана выполняет запрос по этому плану и дополнительно выводит реальное количество строк и время, потраченное на выполнение действий узла. Потраченное время СУБД выводит в миллисекундах, поэтому сравнить его с оценкой обычно не представляется возможным. Но можно сравнить предполагаемое количество строк и реально выбранное. В идеале они должны быть близки. Если это не так, то возможно план выбран не оптимально.[5]

Для некоторых узлов, таких как узлы сортировки и узлы хэш соединения (Hash Join) ANALYZE показывает дополнительную информацию: используемый метод сортировки и память, или структуру хэш-таблицы и используемую память. Также он может показывать количество отфильтрованных записей фильтром.

У EXPLAIN есть ещё опция BUFFERS, которую можно использовать вместе с ANALYZE для получения ещё более подробной статистики выполнения запроса. Это может оказаться полезным, в случае если необходимо оценить части запроса наиболее требовательные к записи/чтению на диск.

Опция ANALYZE выполняет запрос, поэтому нужно понимать, что запросы вставки, изменения или удаления могут модифицировать содержимое таблицы. Если это нежелательно, то можно откатывать транзакцию командой ROLLBACK.

Разберём пару примеров команды EXPLAIN.

Пусть у нас есть таблица tenk1, в которой есть 10000 строк. Выполним команду

```
EXPLAIN SELECT * FROM tenk1;
```

Результатом будет:

```
Seq Scan on tenk1  (cost=0.00..458.00 rows=10000
width=244)
```

Этот план означает, что планировщик предполагает последовательное чтение всех страниц таблицы. Т.к. мы запрашиваем все данные таблицы, то быстрее чем просто прочитать все данные, варианта нет. Продемонстрируем, откуда берётся оценка в 458, выполнив следующий запрос:

```
SELECT relpages, reltuples FROM pg_class WHERE
relname = 'tenk1';
```

Обнаружим что таблица tenk1 занимает 358 дисковых страниц, и в ней 10000 строк. Оценка трудоёмкости cost рассчитывается как

$$cost = P * seq_page_cost + \mathrm{R} * cpu_tuple_cost,$$

где P - число прочитанных страниц, а R - число отсканированных строк

Как уже было замечено выше, значение seq_page_cost по умолчанию равно 1.0, а значение по умолчанию cpu_tuple_cost равно 0.01. Поэтому предполагаемая оценка равна (358 * 1.0) + (10000 * 0.01) = 458.

Добавим в запрос условие в WHERE блоке.

```
SELECT * FROM tenk1 WHERE unique1 < 100;
```

Результат EXPLAIN будет таким

```
Seq Scan on tenk1  (cost=0.00..483.00 rows=7001
width=244)
   Filter: (unique1 < 7000)
```

Видно, что условие из WHERE блока появилась в плане как условие фильтра, прикреплённое к узлу последовательного сканирования таблицы. Однако оценка стоимости меньше не стала, потому что нужно проверить все строки этим условием. Более того, она выросла на трудоёмкость проверки каждой строки. Эта дополнительная трудоёмкость считается как:

$$cost_{add} = R * \mathrm{cpu_operator_cost},$$

где R - число строк, а cpu_operator_cost это оценка выполнения оператора или функции во время запроса. Значение по умолчанию 0.0025.

Предположим у нас есть B-tree индекс на поле unique1. Проанализируем следующий запрос:

```
SELECT * FROM tenk1 WHERE unique1 < 100;
```

Результат EXPLAIN будет следующим

```
 Bitmap Heap Scan on tenk1  (cost=5.07..229.20 rows=101 width=244)
      Recheck Cond: (unique1 < 100)
      ->  Bitmap Index Scan on tenk1_unique1 (cost=0.00..5.04 rows=101 width=0)
            Index Cond: (unique1 < 100)
```

В этот раз планировщик решил использовать двухступенчатый план. Сначала происходит поиск значений по индексу удовлетворяющих условию, затем на основе отобранных значений по индексу, ищутся соответствующие строки в таблице. Поиск значений в таблице через индекс это более затратная операция чем просто последовательное чтение, но планировщик предположил, что строк достаточно мало чтобы оправдать это. Однако строк достаточно много чтобы, планировщик первым шагом решил собрать в битовую карту адрес строк таблицы, и отсортировать их, чтобы на шаге чтения данных из таблицы минимизировать количество запросов страниц таблицы.

1.2.4.Индексы

Индексы - это объекты базы данных, предназначенные для ускорения поиска данных в таблице. Они существуют нескольких видов и могут быть построены от одного или нескольких столбцов таблицы, или от какого-либо выражения включающего эти столбцы. Они могут быть использованы при любых операциях, включающих в себя чтение данных из таблиц, если планировщик решит что использование индекса ускоряет выполнение. После создания индекса СУБД должна его поддерживать в актуальном состоянии, поэтому операции манипуляции данных в таблице, на которую построен индекс, занимают большее время, чем без индексов, чтобы при необходимости обновить индексы.

Для создания индекса существует команда CREATE INDEX с указанием столбца (или столбцов) и таблицы.

По умолчанию создаваемый индекс будет типа Б-дерева (B-tree). С внешней точки зрения это сбалансированное дерево с большой ветвистостью. Сбалансированность значит, что до листьев от корня длина пути не может отличаться более чем на единицу. Ветвистость - это свойство узла ссылаться на множество узлов-потомков. Обычно на физическом уровне Б-дерево представляет собой мульти списочную структуру страниц памяти. Т.е. каждому узлу дерева соответствует блок памяти (страница).[6]

Страница – это основная структура, в которой хранятся данные в БД. Она же является основной единицей для чтения и записи данных на диск. У страниц есть фиксированная структура, заголовок, строки данных и дескрипторы строк. В заголовке хранятся номер страницы, номер предыдущей страницы, номер следующей страницы, информация о свободном месте на странице. В строках данных находится данные, хранимые в БД. Дескрипторы же описывают длину строки и её сдвиг на странице.

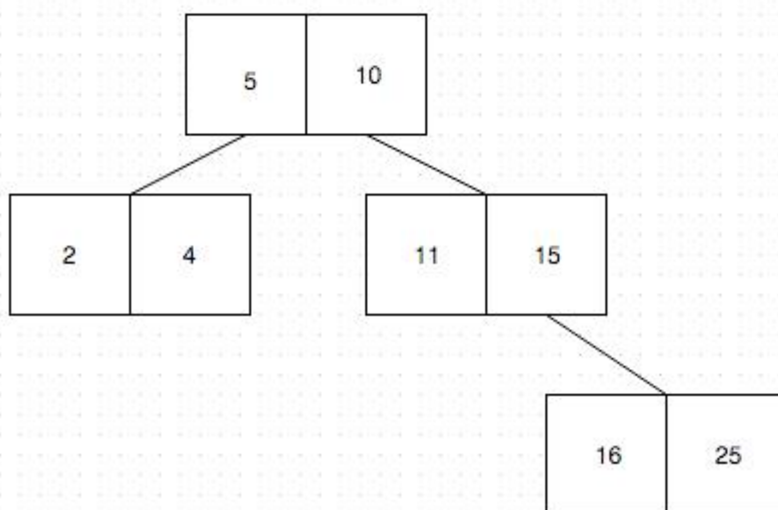

Рисунок 1.0.1.Пример Б-дерева

В PostgreSQL используются Б-деревья, основанные на работе 1981 года Лемана и Яо. Их модификация Б-дерева позволяет во время многопоточной работы обнаружить, что лист дерева был разделён, что в свою очередь позволяет проводить поиск по дереву без удержания блокировки на нём (за исключением блокировки читаемого листа, чтобы его прочесть без модификаций).

Рассмотрим скорость поиска данных с использований Б-дерева индекса в PostgreSQL. Характерным для таких деревьев является их малая высота h. В Б-дереве узел может иметь много потомков, на практике до нескольких тысяч. Количество потомков (максимальное) определяет степень Б-дерева. Узел x, хранящий n[x] ключей, имеет n[x]+1 сыновей. Хранящиеся в x ключи служат границами, разделяющими всех потомков узла на n[x]+1 групп. За каждую группу отвечает один из сыновей x. При поиске в Б-дереве искомый ключ сравнивается с границами в узле x и выбирает один из n[x]+1 путей для дальнейшего поиска. Чтобы уменьшить количество операций чтения-записи на диск при поиске элемента в Б-дереве, нужно максимально уменьшить его высоту. Для этого степень Б-дерева делается максимальной. В PostgreSQL каждый узел индекса занимает одну страницу, соответственно степень зависит от размера ключа. Обычно степень оказывается от 100 до 2000. Например, Б-дерево степени 1000 и высоты 2 может хранить более миллиарда ключей. В таком случае узел-корень дерева обычно всегда хранится в оперативной памяти, а для доступа к остальным ключам требуется всего 2 операции чтения с диска. [7]

Пусть n это число элементов в дереве, а m это максимальное число потомков, которое может иметь. Тогда в узле может быть не более m-1 ключей.

В таком случае если все узлы полностью заполнены, то такое дерево хранит m^h-1 ключей. Соответственно минимальная высота дерева

$$h = (\log_m n + 1) - 1$$

Пусть d это минимальное число потомков, которое может уметь узел. Для обычного Б-дерева d = m/2. Наибольшая высота дерева это

$$h \le \log_d \frac{n+1}{2}$$

Так как в PostgreSQL узел индекса всегда занимает одну страницу, то для поиска нужного ключа нужно прочитать не более h – 1 страниц, с учётом того что корень обычно находится в оперативной памяти.

Сложность поиска (в страницах) по Б-дереву соразмерна его высоте, а поиск чтением по таблице соразмерен количеству страниц на всю таблицу.

Проведём мысленный эксперимент. Предположим, что в каждый узел в среднем помещается 100 ключей. В таком случае для таблицы, в которой менее двух миллионов ключей, высота не превысит 2 (высота корня принимается за 0). Соответственно для поиска нужного ключа нужно прочитать не более двух страниц с диска. Пусть таблица состоит только из этого ключа (такое невозможно, но сделаем такое допущение), и на каждую страницу помещается в два раза больше ключей чем в страницу с индексом. В таком случае таблица занимает 10000 страниц. Тогда поиск по индексу требует чтения трёх страниц, а поиск по таблице 10000.

Теперь проведём эксперимент на нашей базе данных. Создадим таблицу из одного столбца, заполненную числами от 1 до 10000000, создадим индекс на этот столбец.

```
create table test_tbl as select generate_series(1,10000000) as col_a;
create index on test_tbl(col_a);
```

Узнаем сколько страниц занимают наши созданные объекты:

```
SELECT relname, relpages FROM pg_class WHERE relname LIKE 'test_tbl%';
```

Таблица - 44248 страниц, индекс – 27422.

Узнаем структуру индекса, выполнив следующий запрос:

```
SELECT * FROM pgstatindex('test_tbl_col_a_idx');
```

Высота – 2, внутренних страниц – 98, страниц листьев – 27323.

В таком случае мы можем ожидать при поиске одного значения с индексом четыре прочитанных страницы (три индекса и одна таблицы), а для таблицы 44248. Воспользуемся ранее изученной командой EXPLAIN (ANALYZE, BUFFERS)

```
explain (analyze, buffers)
select * from test_tbl where col_a = 60000
Результат:
Index Only Scan using test_tbl_col_a_idx on test_tbl (cost=0.44..8.45 rows=1 width=4) (actual time=0.012..0.012 rows=1 loops=1)
  Index Cond: (col_a = 60000)
  Heap Fetches: 1
  Buffers: shared hit=4
```

Теперь, установим флаги enable_indexscan и enable_bitmapscan в off, чтобы планировщик не использовал индекс, а последовательно отсканировал таблицу и вызовем EXPLAIN (ANALYZE, BUFFERS) опять. Результат:

```
Seq Scan on test_tbl  (cost=0.00..169248.60 rows=1 width=4) (actual time=5.536..881.463 rows=1 loops=1)
  Filter: (col_a = 60000)
  Rows Removed by Filter: 9999999
  Buffers: shared hit=2177 read=42071
```

Часть страниц уже находилась в кэше, поэтому они не были прочитаны с диска, но суммарно планировщик прочитал 44248 страниц как мы и предполагали. Значит мы подтвердили экспериментом ранее выведенные математически заключения. Эффективность индекса при точечном поиске невозможно отрицать.

Б-дерево индексы могут использоваться для сравнений равенства и сравнений меньше, больше, меньше или равно, больше или равно. Эквивалентные этим операции BETWEEN и IN соответственно тоже могут быть реализованы поиском по дереву. Это возможно, потому что Б-дерево это упорядоченная структура. Также возможен поиск для операции сравнения по шаблону LIKE и

~, но только если шаблон с которым сравнивается, начинается с константы. Т.е. сравнение LIKE ‘foo%’ может использовать индекс, а LIKE ‘%foo’ не может.

Также т.к. Б-дерево упорядочено по порядку, то оно может быть использовано, если необходимо вернуть данные в упорядоченном виде. Но это не всегда быстрее чем простое чтение таблицы без использования индекса, это зависит от того какую часть строк таблицы необходимо вернуть.

Также PostgreSQL поддерживает хэш индексы (hash index). Хэш индексы представляют собой хэш-таблицу и поддерживают только сравнения равенства. Также в старых версиях PostgreSQL они не записываются в область WAL, поэтому их нужно будет перестраивать после критической ошибки системы. Но у них есть и неоспоримые плюсы перед Б-дерево индексами – они занимают меньше места на диске, и поиск в них зачастую быстрее. Для таблиц с сотнями миллионов записей, выигрыш в сэкономленном пространстве может достигать свыше 10 Гб, и количество чтений с диска для такого индекса в среднем меньше.[8]

Как было упомянуто выше, индекс может быть определён не для одного столбца таблицы, а для нескольких. Может быть использовано до 32 столбцов, но это предел может быть увеличен в настройках сервера. Хэш-индексы не поддерживают индексы для нескольких столбцов.

Индекс на несколько столбцов может быть использован для условий, которые включают любое подмножество столбцов множества столбцов индекса, но наиболее эффективен, когда условие ограничения распространяется на самые первые (левые) столбцы такого индекса. Точным определением работы является следующее: будут использованы сравнения равенства на первые столбцы плюс неточные сравнения на первый столбец по порядку, на который нет сравнения равенства.

Индексы нескольких столбцов занимают больше места, чем индексы единственного столбца, и их следует использовать осторожно. Также их перестрое-

ние занимает большее время, поэтому если нет однозначной причины использовать такой индекс, то стоит их избегать. [9]

Индексы также могут быть использованы, если в запросе есть ORDER BY. Это позволяет избежать дополнительного узла сортировки в плане запроса. Только Б-дерево индексы позволяют это, т.к. только они являются сортированными по порядку, остальные типы индексов возвращают соответствующие строки в порядке, зависящем от реализации алгоритма. По умолчанию Б-дерево индексы хранят значения в возрастающем порядке с null значениями в конце. Это значит, что прямое сканирование индекса выдаёт результат, подходящий для ORDER BY x (или в более полной форме ORDER BY x ASC NULLS LAST). Сканирование в обратном порядке соответственно выдаёт результат, подходящий для ORDER BY x DESC NULLS FIRST, или более кратко ORDER BY x DESC.

При необходимости при создании индекса можно задать параметры сортировки опциями ASC, DESC, NULLS FIRST, NULLS LAST. Это редко необходимо, но в определённых ситуациях может предоставить значительный прирост в скорости. Стоит ли поддерживать такой индекс, определяется тем, насколько часто используются запросы с такой сортировкой.

Также индексы могут быть частичными. Частичные индексы покрывают только часть таблицы, ограниченную условием (предикатом) заданным при создании индекса. Это специализированная опция, но есть некоторые ситуации, в которых они полезны. [10]

Одной важной ситуацией является создание индекса, которые не индексирует часто встречающиеся значения в таблице. Т.к. запросы по этим значениям, которые запрашивают более 1-10% всех строк таблицы (в зависимости от настроек СУБД) обычно не используют индекс, то не имеет смысла забивать индекс такими значениями. Это уменьшает размер индекса и увеличивает скорость его использования. Также такие индексы реже нужно обновлять для операций обновления таблицы.

Рассмотрим подробнее процесс создания индексов

```
CREATE [ UNIQUE ] INDEX [ CONCURRENTLY ] [ [ IF NOT EXISTS ] name ] ON table_name [ USING method ]
    ( { column_name | ( expression ) } [ COLLATE collation ] [ opclass ] [ ASC | DESC ] [ NULLS { FIRST | LAST } ] [, ...] )
    [ WITH ( storage_parameter = value [, ... ] ) ]
    [ TABLESPACE tablespace_name ]
    [ WHERE predicate ]
```

Ключевое слово UNIQUE указывает, что нужно создать уникальный индекс. Это означает, что система будет проверять уникальность значений указанных столбцов в таблице. Операции модификации данных, которые приведут к дублирующимся значениям, завершатся с ошибкой. Если в таблице есть дублирующиеся значения на момент создания индекса, то индекс не будет создан.

Ключевое слово CONCURRENTLY указывает СУБД, что индекс нужно создавать в режиме без блокировок табличных данных на чтение, удаление и изменение. По умолчанию индекс создаётся в режиме, в котором данные блокируются. Создание индекса в CONCURRENTLY режиме требует в 2-3 раза больше времени, но незаменимо когда необходимо создать индекс на базе данных, где данные в таблице активно модифицируются. [11]

name – имя индекса, схема указываться не должна, она совпадает со схемой таблицы. Если имя пропущено, то PostgreSQL выберет имя на основании имени таблицы и имён столбцов, на которых создаётся индексов

Выражение IF NOT EXISTS указывает, что индекс необходимо создать, только если он не существует. По умолчанию команда попробует создать индекс, и если индекс с таким именем на той же таблице существует, то команда завершится с ошибкой. IF NOT EXISTS удобно для скриптов обновления базы, которые можно запускать несколько раз без появления непредвиденных эффектов.

table_name – имя таблицы. Может включать, или не включать в себя схему.

method – тип индекса. Тип по умолчанию – btree (Б-дерево).

column_name – название столбца

expression – выражение на основе одного или нескольких столбцов. В большинстве случаев должно быть обёрнуто в круглые скобки.

ASC – сортировать в возрастающем порядке. Режим по умолчанию.

DESC – сортировать в убывающем порядке.

NULLS FIRST – Указывает что null значения должны быть в начале индекса. Вариант по умолчанию, если выбрана сортировка по убыванию.

NULLS LAST – Указывает что null значения должны быть в конце индекса. Вариант по умолчанию, если сортировка по убыванию не выбрана.

predicate – условие (предикат) для создания частичного индекса.

1.3.Выводы и уточненная постановка задачи

В предыдущем разделе данной главы были рассмотрены инструменты, которые могут помочь разработчику решить задачу анализа и модификации структуры СУБД для ускорения запросов.

При описании были отмечены и общие положения, и конкретные, которые особенно применимы для данной задачи.

Задача анализа была разделена на задачу сбора данных для анализа, задачу предобработки этих данных, и на задачу собственно анализа медленных частей системы.

Задача сбора данных для анализа решается с помощью встроенной в СУБД PostgreSQL системы логирования событий. Она глубоко конфигурируема и позволяет записывать только те события, которые нужны администратору.

Задача предобработки собранного лога необходима, потому что интерпретация лога человеком затруднительна. К счастью, эту задачу нет необходимости решать самостоятельно, потому что инструменты для этого уже были созданы. В данной работе было принято решение использовать ПО pgBadger.

Решив предыдущие задачи, у нас будут данные, какие запросы в системе медленно выполняются. Используя подробный планировщик запросов, мы сможет определить причину медленного выполнения.

Наконец задача модификации сводится к тому что, изучив планы запросов, мы сможем создать, изменить или удалить индексы, которых не хватает для более быстрого выполнения запроса, или наоборот, которые замедляют работу системы.

Глава 2. Анализ системы.

2.1. Описание складской системы.

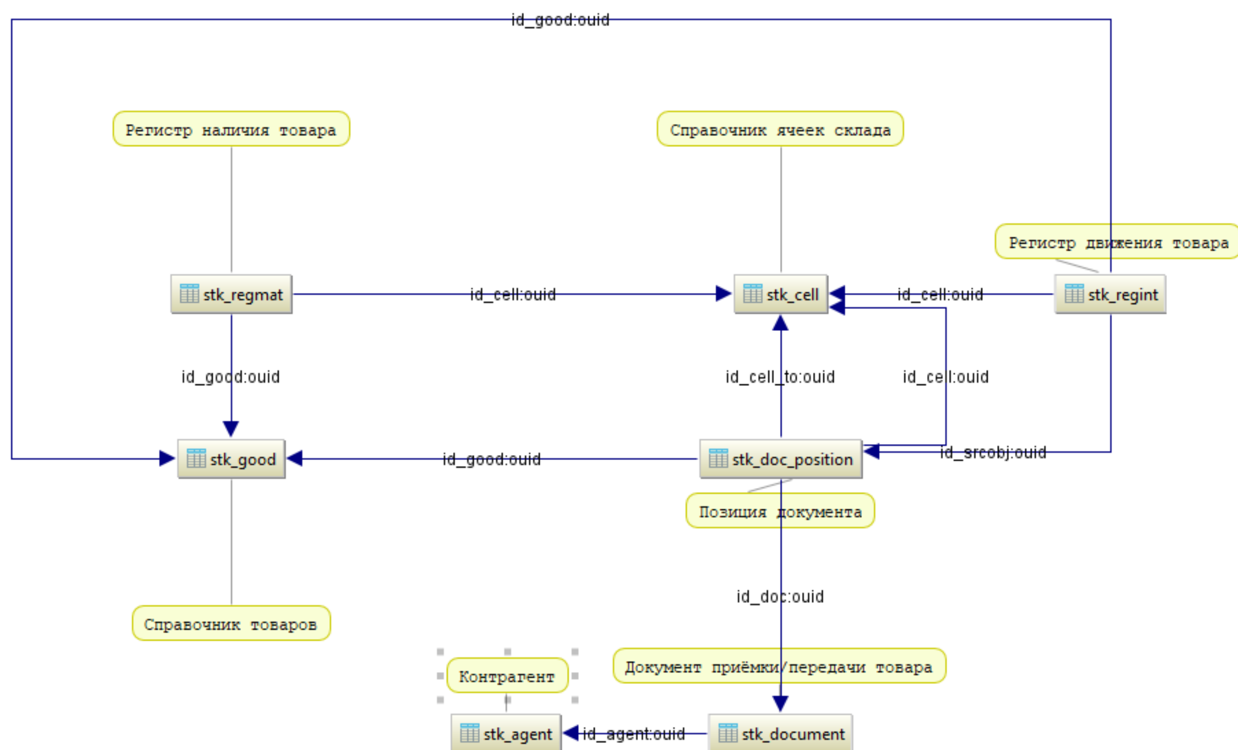

Рисунок 2.1. Упрощённая схема БД

Базовый принцип работы системы:

Создаются документы приёмки/поставки для работы с определённым контрагентом.

В документах есть позиции, в которых указаны товар, количество и ячейка склада, куда товар придет или откуда будет взят. У позиций есть поля, отвечающие за степень выполнения позиции.

Регистр движения товара заполняется от выполнения позиций документов. От каждой позиции документа создаётся строка в регистре с той же ячейкой, товаром и количеством. Если товар приходит, то количество положительное, если товар отгружают, то количество отрицательное. Есть поле «источник», указывающее на позицию. Если позиция отменяется после выполнения, то соответствующая строка в регистре движения удаляется.

Регистр наличия товара представляет собой свёртку всего регистра движения товара, и показывает наполненность склада на текущий момент. Он изменяется при каждом изменении регистра движения товара. Стоят ограничения на то, что реальное количество товара не может стать отрицательным.

2.2. Проведения тестирования и анализ результатов

На основании данных из технического задания, проводим над системой нагрузочное тестирование с включённым логированием. По окончанию тестирования, собираем результаты и обрабатываем с помощью pgBadger.

Общая статистика тестирования:

- 583 уникальных нормализованных запроса
- 1,472,455 запросов
- 1 день 4 часа 53 минуты суммарная длительность выполнения всех запросов
- Пик запросов – 1998 в секунду

Считаем полученные данные достаточными, чтобы проводить анализ и модификацию системы на их основе. Посмотрим на статистику запросов подробнее.

Начнём с типов запроса.

Таблица 2.1. Количество выполненных запросов по типам

Тип запроса	Количество	Длительность выполнения
SELECT	1 366 642	1d1h10m47s
INSERT	22 977	3h37m27s
UPDATE	15 895	1m29s
DELETE	819	205ms

SELECT запросов по количеству более 90%. Это было ожидаемо, т.к. работа со складской системой подразумевает постоянное получение данных склада и складских документов. По времени выполнения они занимают меньше в относительном соотношении, потому что операции чтения гораздо быстрее, чем операции записи на диск.

Теперь подробнее рассмотрим, какие именно запросы выполнялись дольше и чаще всего.

Таблица 2.2. Запросы выполнявшиеся дольше всего

Ранг	Мин. длит.	Макс. длит.	Средняя длит.	Раз выполнен	Общ. длит.	Листинг запроса
1	1с 629мс	1м2с	23с495мс	2 496	12ч17м24с	В разделе 2.3
2	0мс	7с629мс	1с145мс	19 347	6ч9м15с	В разделе 2.4
3	1с644мс	40с962мс	29с797мс	415	3ч26м5с	Приложение 1, запрос 1
4	0мс	4с904мс	1с265мс	7 184	2ч31м33с	Приложение 1, запрос 2
5	0мс	5с688мс	1s561ms	415	10м48с	Приложение 1, запрос 3

Запросы на шестом месте и дальше, суммарно выполнялись менее трёх минут, и каждый из них был вызван значительное число раз. В рамках целей этой работы считаем, что они у них нет критичных проблем в быстродействии, поэтому не занимаемся их анализом. Теперь проведём подробный анализ запросов, на которые ушло наибольшее число времени и второе наибольшее.

2.3. Запрос степени выполнения документов.

У складской системы есть интерфейс мониторинга для руководителей, в котором можно отслеживать разные параметры рабочего процесса. Одна из таблиц представляет собой список документов, которые находятся в работе, степень завершённости документа и время последнего изменения позиций в документе.

```
select doc_title
```

```
          ,floor(100*positions_done/positions_in_work) as
percentage_done
          ,last_doc_update
     from (
   select dt.name || ' '|| d.docnum || ' от ' ||
(d.docdate :: varchar) as doc_title
          ,count(case when p.isinwork = 't' then 1 end)
as positions_in_work
          ,count(case when p.isdone = 't' then 1 end) as
positions_done
          ,to_char(max(p.time_done),'HH24:mm:ss
dd.MM.YYYY') as last_doc_update
          ,d.docdate
     from (
   select distinct t.id_doc
     from stk_doc_position t
      where t.isinwork = 't'
          ) r
     join stk_document d on r.id_doc = d.ouid
     join stk_doc_position p on d.ouid = p.id_doc
     join stk_doc_type dt on d.id_type = dt.ouid
    group by d.ouid
     ,dt.name
     ,d.docnum
     ,d.docdate
    ) tt
    order by tt.docdate asc;
```

План этого запроса, полученный с помощью EXPLAIN (ANALYZE, BUFFERS)

```
   Sort  (cost=373260.36..374915.30 rows=661975
width=80) (actual time=464.091..464.094 rows=50 loops=1)
     Sort Key: tt.docdate
     Sort Method: quicksort  Memory: 32kB
     Buffers: shared hit=14138 read=21767
```

```
      I/O Timings: read=84.541
      ->  Subquery Scan on tt  (cost=209057.80..250431.24 rows=661975 width=80) (actual time=455.467..464.069 rows=50 loops=1)
            Buffers: shared hit=14138 read=21767
            I/O Timings: read=84.541
            ->  GroupAggregate (cost=209057.80..237191.74 rows=661975 width=127) (actual time=455.464..464.051 rows=50 loops=1)
                  Group Key: d.ouid, dt.name
                  Buffers: shared hit=14138 read=21767
                  I/O Timings: read=84.541
                  ->  Sort  (cost=209057.80..210712.74 rows=661975 width=57) (actual time=455.326..455.702 rows=4594 loops=1)
                        Sort Key: d.ouid, dt.name
                        Sort Method: quicksort  Memory: 551kB
                        Buffers: shared hit=14138 read=21767
                        I/O Timings: read=84.541
                        ->  Nested Loop (cost=74698.46..120164.68 rows=661975 width=57) (actual time=445.470..447.556 rows=4594 loops=1)
                              Join Filter: (t.id_doc = p.id_doc)
                              Buffers: shared hit=14138 read=21767
                              I/O Timings: read=84.541
                              ->  Hash Join (cost=74698.03..76113.67 rows=4897 width=51) (actual time=445.455..445.491 rows=50 loops=1)
                                    Hash Cond: (d.ouid = t.id_doc)
                                    Buffers: shared hit=12589 read=21767
```

```
                                    I/O Timings:
read=84.541
                                    ->  Hash Join
(cost=29.13..1245.16 rows=40170 width=47) (actual
time=0.019..13.397 rows=40170 loops=1)
                                          Hash Cond:
(d.id_type = dt.ouid)
                                          Buffers: shared
hit=263
                                          ->  Seq Scan on
stk_document d  (cost=0.00..663.70 rows=40170 width=19)
(actual time=0.008..3.116 rows=40170 loops=1)
                                                Buffers:
shared hit=262
                                          ->  Hash
(cost=18.50..18.50 rows=850 width=36) (actual
time=0.006..0.006 rows=3 loops=1)
                                                Buckets:
1024  Batches: 1  Memory Usage: 9kB
                                                Buffers:
shared hit=1
                                                ->  Seq
Scan on stk_doc_type dt  (cost=0.00..18.50 rows=850
width=36) (actual time=0.002..0.003 rows=3 loops=1)

Buffers: shared hit=1
                                    ->  Hash
(cost=74607.69..74607.69 rows=4897 width=4) (actual
time=428.733..428.733 rows=50 loops=1)
                                          Buckets: 8192
Batches: 1  Memory Usage: 66kB
                                          Buffers: shared
hit=12326 read=21767
                                          I/O Timings:
read=84.541
                                          ->  Unique
(cost=74531.98..74558.72 rows=4897 width=4) (actual
time=428.196..428.718 rows=50 loops=1)
```

```
                                                    Buffers:
shared hit=12326 read=21767
                                                    I/O
Timings: read=84.541
                                                    ->  Sort
(cost=74531.98..74545.35 rows=5348 width=4) (actual
time=428.196..428.500 rows=4594 loops=1)

Sort Key: t.id_doc

Sort Method: quicksort  Memory: 408kB

Buffers: shared hit=12326 read=21767
                                                          I/O
Timings: read=84.541
                                                          ->
Seq Scan on stk_doc_position t  (cost=0.00..74200.81
rows=5348 width=4) (actual time=426.682..427.581
rows=4594 loops=1)

Filter: isinwork

Rows Removed by Filter: 4005125

Buffers: shared hit=12326 read=21767

I/O Timings: read=84.541
                                  ->  Index Scan using
idx_pos_doc on stk_doc_position p  (cost=0.43..7.31
rows=135 width=14) (actual time=0.004..0.023 rows=92
loops=50)
                                        Index Cond: (id_doc =
d.ouid)
                                        Buffers: shared
hit=1549
```

Начнём разбирать план, начиная с нижних узлов.

Сначала последовательно сканируется таблица stk_doc_position, и применяется фильтр по условию isinwork = ‘t’. Прочитывается 33 тысячи страниц. Читается нам необходимый столбец id_doc. Стоимость этого узла – 74191

Затем идёт узел Sort. СУБД проводит сортировку методом quicksort. Эта операция не читает дополнительных строк из базы данных, не уменьшает число строк, которое она прияла на вход, но использует 408 Кбайт оперативной памяти. По умолчанию лимит памяти на сортировку – 1 Мбайт. Если используется больше, то PostgreSQL пишет на диск и использует другой алгоритм сортировки. В запросе в этот момент сортировки нет, но в плане она появляется потому, что планировщик запроса решает, что выполнить операцию distinct быстрее всего через сортировку значений. Стоимость этого узла начинается с 74522 – сортировка производится в предварительной стадии. Заканчивается на стоимости 74536.

После чего следует узел Unique. Из уже отсортированного списка значений, выбираются только уникальные значения. Количество ожидаемых строк падает с 5346 до 4895. Однако здесь планировщик сильно ошибается. В реальном запросе уникальных строк является только 50. Причиной этому может быть устаревшая статистика на таблице. Благодаря тому, что строки уже отсортированы, удаление повторяющихся строк это относительно лёгкая операция. Стоимость возрастает до 74549.

Следующий узел Hash. В нём подготавливается хэш-таблица, чтобы провести соединение двух таблиц. Количество вёдер (Buckets) представляет собой степень двойки. Это число больше числа предполагаемых строк, чтобы они влезли. В данном случае на основе предыдущего числа в 4895 строк, планировщик выбирает 8192 ведра.

Посмотрим теперь на ветвь узлов, с которой будем проводить Hash Join выше в плане.

Последовательное сканирование таблицы stk_doc_type. Планировщик считает, что там 850 строк, в то время как на самом деле там их 3. В реальности читается одна страница. Стоимость этого узла 18.

В узле Hash подготавливается хэш-таблица с 1024 вёдрами.

Происходит последовательное чтение таблицы stk_document, читается вся таблица, выбираются все 40170 строк.

Теперь алгоритмом Hash Join в узле с одноимённым названием, происходит соединение строк из таблиц stk_document и stk_doc_type. При таком алгоритме одна таблица (ожидаемо меньшая, в нашем случае stk_doc_type) хэшируется предварительно, а от другой таблицы (stk_document) выбранные столбцы соединения построчно хэшируются и сравниваются на попадание в хэш-таблицу stk_doc_type. Если совпадают, то строка из stk_document отправляется в результирующий набор строк, вместе с соответствующей строкой из stk_doc_type. Соответственно такой алгоритм соединения возможен только для условия равенства. Наше условие строки не множит, на выходе остаются 40170 строк. Стоимость была оценена в 1245.

Узел Hash Join в котором мы соединяем таблицу из предположительно 40170 строк, состоящих из таблицы документов и типов, и захэшированную таблицу из предположительно 4895 строк позиций документов. Общая стоимость доходит до 76104.

Далее мы сканируем таблицу stk_doc_position, используя сканирование с индексом, благодаря наличию индекса на поле id_doc у этой таблицы.

Затем мы, используя алгоритм Nested Loop, соединяем две таблицы. Этот алгоритм для каждой строки одной таблицы (в данном случае результирующей таблицы от stk_document) проверяет условие для каждой строки другой таблицы. В данном случае благодаря наличию вышеупомянутого индекса, условие проверялось не для таблицы, а для индекса, что очень сэкономило время. Nested Loop это самый базовый вид соединения таблиц, его плюсом является то, что

он очень быстр на небольших объёмах, потому что не нужно подготавливать данные. Строк стало больше, в количестве 488619 предполагаемых и 4594 в реальности, за счёт того что мы «размножили» строки документы на число строк их позиций.

Затем планировщик выполняет сортировку, чтобы агрегация выполнялась проще. Памяти потребовалось 551 Кбайт, всё ещё в пределах, разрешённых для сортировки в оперативной памяти. Предполагаемая стоимость выросла более чем на 100000, до 185901, потому что планировщик считает, что нужно отсортировать почти полмиллиона строк.

Затем проходить агрегация и финальная сортировка. За счёт неверных данных, по плану агрегация оставит столько же строк, сколько и было до неё. Финальная сортировка опять предполагается очень трудоёмкой.

Разобрав план запроса, нам теперь видны проблемы.

Самый нижний узел поиска строк stk_doc_position проходит через полное сканирование таблицы. Мы знаем что он возвращает ~0.01% строк таблицы, а значит его можно ускорить используя индекс. Во-вторых, мы наблюдаем что планировщик считает что вернётся 4500 записей, в то время как вернётся только 50. Поэтому планировщик полностью сканирует stk_document чтобы провести Hash Join. Если бы планировщик предполагал бы, что вернётся 50 строк, то он мог бы провести упрощённый Nested Loop с таблицей stk_document, потому что ouid это ключ, а на главный ключ обязательно есть индекс.

Далее видно, что планировщик сильно ошибся в числе предполагаемых строк в stk_doc_type. Это, скорее всего показатель неверно собранной и не обновлённой статистики по этой таблице.

Рекомендуется – создать индекс на таблице stk_doc_positions на поле isinwork. Известно что большая часть значений в таблице будет иметь значение этого поля false. Соответственно индексировать их бесполезно, так как, во-первых запрос только по этому полю вернёт 99% записей, во-вторых, даже ес-

ли такой запрос необходим, он не будет использовать индекс, а будет напрямую сканировать таблицу. Поэтому индекс рекомендуется создавать частичный, с предикатом isinwork = ‘t’.

Также рекомендуется выполнить команду ANALYZE на таблицу stk_doc_type и проверить настройки СУБД на предмет наличия проблем с периодическим автоматическим анализом таблиц.

2.4. Запрос остатка незарезервированного товара на складе

Второй запрос, который мы будем анализировать, это запрос остатка незарезервированного товара на складе.

```
select
 c.ouid as stk_cell_ouid
,c.name as stk_cell_name
,g.ouid as stk_good_ouid
,g.name as stk_good_name
,sum(case when r.ismov = 't' then r.count when
r.isres = 't' then -r.count end) as stk_good_left
from stk_regmat r
join stk_good g on r.id_good = g.ouid
join stk_cell c on r.id_cell = c.ouid
where id_good = $1
group by
r.id_good
,c.ouid
,c.name
,g.ouid
,g.name;
```

Посмотрим на его план, выполнив команду EXPLAIN (ANALYZE, BUFFERS), выбрав в качестве параметра значение 1000.

```
GroupAggregate  (cost=45877.37..46064.94 rows=1672 width=67) (actual time=236.258..238.184 rows=1724 loops=1)
  Group Key: r.id_good, c.ouid, g.ouid
  Buffers: shared hit=1188 read=22988
  I/O Timings: read=89.265
  ->  Merge Join  (cost=45877.37..46027.32 rows=1672 width=65) (actual time=236.250..237.452 rows=1724 loops=1)
        Merge Cond: (c.ouid = r.id_cell)
        Buffers: shared hit=1188 read=22988
        I/O Timings: read=89.265
        ->  Index Scan using stk_cell_pkey on stk_cell c  (cost=0.29..2745.94 rows=45000 width=18) (actual time=0.004..0.329 rows=2001 loops=1)
              Buffers: shared hit=32
        ->  Sort  (cost=45877.08..45881.26 rows=1672 width=51) (actual time=236.243..236.469 rows=1724 loops=1)
              Sort Key: r.id_cell
              Sort Method: quicksort  Memory: 291kB
              Buffers: shared hit=1156 read=22988
              I/O Timings: read=89.265
              ->  Nested Loop  (cost=0.42..45787.56 rows=1672 width=51) (actual time=0.278..235.405 rows=1724 loops=1)
                    Buffers: shared hit=1156 read=22988
                    I/O Timings: read=89.265
                    ->  Index Scan using stk_good_pkey on stk_good g  (cost=0.42..8.44 rows=1 width=37) (actual time=0.005..0.008 rows=1 loops=1)
                          Index Cond: (ouid = 1000)
                          Buffers: shared hit=4
                    ->  Seq Scan on stk_regmat r  (cost=0.00..45762.40 rows=1672 width=14) (actual time=0.271..235.056 rows=1724 loops=1)
```

```
                              Filter: (id_good = 1000)
                              Rows Removed by Filter:
1728068
                              Buffers: shared hit=1152
read=22988
                              I/O Timings: read=89.265
```

Разберём его план.

Первым выполняется узел индексного поиска в таблице товаров stk_good по одному значение. Хоть и в запросе нет прямого равенства значения столбца таблицы stk_good, планировщик проводит цепочку на основе условия соединения, и поэтому применяет условие к этой таблице тоже. Ожидается одна строка, приходит одна строка, индекс в данной ситуации отрабатывает максимально эффективно.

«Параллельно» с этим выполняется полное сканирование регистра наличия товаров stk_regmat, фильтруя по идентификатору товара. Статистика на таблице верна, предполагаемое число строк близко к реальному числу строк.

Типом соединения эти таблиц планировщик выбирает Nested Loop, потому что с одной стороны соединения только одна строка. Это очень эффективно, оценка растёт с 45763 + 8 до 45789 после соединения.

Следующим типом соединения планировщик выбрал Merge Join, который требует отсортированных таблиц. Поэтому сначала идёт узел сортировки по значению id_cell. Памяти хватает, поэтому используется quicksort. Т.к. мы соединяем с полем, являющимся главным ключом таблицы STK_Cell, то поиск в ней происходит по индексу. Т.к. индекс упорядочен, то строки из этой таблицы уже приходят отсортированными, поэтому нет отдельного узла сортировки.

Узел с Merge Join по плану выполняется достаточно быстро, потому что строк ожидается немного.

Наконец план заканчивается агрегацией, которая по предложенному плану запроса вернёт столько же записей. Так и происходит.

В этом запросе проблема наблюдается только одна – полное сканирование регистра STK_Regmat. Планировщик выбрал вариант сканирования таблицы, в которой более 2-ух миллионов записей, и это привело к чтению более 20000 блоков. Посмотрим на то, какие на таблице есть индексы, выполнив запрос

```
    select *  from pg_indexes  where tablename =
'stk_regmat';
```

Вывод запроса:

```
    "public";"stk_regmat";"stk_regmat_pkey";"";"CREATE
UNIQUE INDEX stk_regmat_pkey ON public.stk_regmat USING
btree (ouid)"
    "public";"stk_regmat";"stk_regmat_id_cell_id_good_idx
";"";"CREATE INDEX stk_regmat_id_cell_id_good_idx ON
public.stk_regmat USING btree (id_cell, id_good)"
```

По результатам этого запроса видно, что у таблицы, помимо уникального индекса на первичный ключ, ещё есть Б-дерево индекс из нескольких столбцов, на столбцах (id_cell, id_good). Т.к. в нашем запросе фигурирует только условие по столбцу id_good (товар), и отсутствует условие по ячейке, где он хранится, то использование этого индекса не является целесообразным с точки зрения планировщика, потому что индекс для нескольких столбцов упорядочен по столбцам слева направо.

Как мы уже исследовали, индекс может не использоваться, если количество строк в таблице, которое он вернёт, весьма значительно. Поэтому прежде чем решать, что нам нужен новый индекс, исследуем предположительное количество строк, которое вернёт поиск по одному товару относительно общего числа строк в таблице. Для id_good, который мы проверяли это соотношение 1700 к ~1.8 млн., т.е. примерно 0.1%. Зная, что у системы нет какого-либо однозначного предпочтения каким-либо конкретным товарам, мы может оценивать эту цифру как относительно верную в общем случае. Для объёма данных в 0.1% строк таблицы создание индекса однозначно ускорит быстродействие узла запроса данных из таблицы. Остаётся выбрать тип индекса. B-tree это подходящий вариант, и он был бы однозначным выбором в версиях PostgreSQL 9 и ра-

нее. В 10 версии, которую использует система, которую мы анализируем, есть безопасная возможность использовать hash-индекс. Т.к. сравнение идентификатора товара у нас может быть только на равенство, то упорядоченность Б-дерева, которая позволяет искать диапазон значений, нам не нужна.

Предварительный вывод: если нет других запросов, которые были бы ускорены созданием индекса на несколько столбцов для этой таблицы, который начинается с id_good, то рекомендуется создать hash-индекс на поле id_good.

2.5. Выводы

Было проведено нагрузочное тестирование складской системы. Был получен отчёт на основе логов.

Мы разобрали полученный отчёт, узнали соотношения запросов чтения и модификации. Увидели запросы, которые больше всех остальных замедляют систему.

Воспользовавшись планировщиком PostgreSQL, мы провели пошаговый анализ нескольких медленных запросов, и на основе нашего анализа мы вывели рекомендуемые действия для ускорения системы. Теперь необходимо будет провести эти действия, после чего провести контрольные измерения работы системы, чтобы определить верными ли были наши выводы о необходимых изменениях.

Глава 3. Модификация системы и анализ результата

3.1. Создание индексов

Для запроса 2.3 создадим hash-индекс, используя команду

```
CREATE INDEX IDX_STK_REGMAT_GOOD ON STK_REGMAT USING HASH(ID_GOOD);
```

Также для обновления статистики выполним команду

```
ANALYZE STK_DOC_TYPE;
```

Для запроса 2.4 создадим B-tree индекс, используя команду

```
CREATE INDEX IDX_STK_DOC_POSITION_IN_WORK ON STK_DOC_POSITION (ISINWORK) WHERE (ISINWORK = TRUE);
```

Индексы созданы успешно. Посмотрим, используя системные таблицы, сколько наши новые индексы занимают места на диске.

```
SELECT
    t.tablename,
    indexname,
pg_size_pretty(pg_relation_size(quote_ident(t.tablename)::text)) AS table_size,
pg_size_pretty(pg_relation_size(quote_ident(indexrelname)::text)) AS index_size
FROM pg_tables t
LEFT OUTER JOIN
    ( SELECT c.relname AS ctablename, ipg.relname AS indexname, indexrelname FROM pg_index x
        JOIN pg_class c ON c.oid = x.indrelid
        JOIN pg_class ipg ON ipg.oid = x.indexrelid
        JOIN pg_stat_all_indexes psai ON x.indexrelid = psai.indexrelid )
    AS foo
    ON t.tablename = foo.ctablename
```

```
WHERE t.schemaname='public'
 and foo.indexname in
('idx_stk_regmat_good','idx_stk_doc_position_in_work');
```

Индекс idx_stk_regmat_good занимает 95 Мб, когда сама таблица занимает 189 Мб. И это соотношение будет оставаться примерно одинаковым по мере роста таблицы. Но на момент решения этой задачи это не является проблемой.

Индекс idx_stk_doc_position_in_work занимает 120 Кб. Таблица занимает 266 Мб. С точки зрения места на диске это крайне эффективный индекс. Его обновление во время операций модификации данных практически не будет заметно.

3.2. Тестирование системы после модификации

Проведём нагрузочное тестирование опять, чтобы оценить наши модификации. Будем пользоваться теми же инструментами, и такой же нагрузкой как и до изменений.

Таблица 3.1. Распределение времени выполнения запросов по типам

Тип запроса	Количество	Длительность выполнения
SELECT	1 360 971	13ч10м47с
INSERT	22 817	3ч55м12с
UPDATE	15 343	1м45с
DELETE	780	205мс

Таблица 3.2. Запросы, выполнявшиеся дольше всего после тестирования

Ранг	Мин. длит.	Макс. длит.	Средняя длит.	Раз выполнен	Общ. длит.	Листинг запроса
1	1с644мс	40с962мс	29с797мс	406	3ч19м5с	Приложение 1, запрос 1
2	1мс	4с904мс	1с265мс	7 103	2ч51м03с	Приложение 1, запрос 2
3	0мс	5с00мс	1s561ms	409	10м48с	Приложение 1, запрос 3

Тестирование после проведения модификаций, показывает что система в целом стала быстрее. Запросы которые мы модифицировали, более не попадают

в ту границу, в пять минут, которую мы оценили как границу для запросов, которые стоит анализировать и модифицировать в первую очередь.

Наблюдается замедление скорости выполнения запроса вставки данных в таблицу STK_Regmat, которое выражается в увеличении выполнения времени запроса вставки в эту таблицу (листинг запроса указан в приложении 1, запрос 2) на 20%

Однако это замедление принимается как приемлемое, с учётом того что мы добились, оптимизировав запросы на выборку данных, которые в том числе, выполняются в том же бизнес процессе, так что общее время ожидание было уменьшено.

Можно сделать вывод, что модификация была проведена в целом успешно.

Заключение

В данной работе была проведено исследование возможностей ускорения работы СУБД на примере складкой системы. Были изучены инструменты анализа работы системы. Были рассмотрены настройки системы логирования PostgreSQL 10. Были изучены важнейшие элементы любой СУБД – планировщик запросов и структура индексов.

Время выполнения запросов поиска данных снизилось на 48%. Время выполнения запросов вставки и модификации увеличилось на 17%, но общее время выполнения запросов уменьшилось на 40%.

Можно сделать следующие выводы:

- Встроенный планировщик запросов является важнейшим инструментом разработчика, пишущего запросы для базы данных. Он показывает пошаговую схему построение запроса, что даёт возможность досконально понять действия системы. Это в свою очередь позволяет писать более оптимизированные запросы.

- Индексы это неотделимая часть базы данных. Они могут на порядок и выше ускорить выполнение запросов. Однако индексы несут и минусы, такие как замедление обновления данных. Понимание этого позволит создавать только необходимые и эффективно используемые индексы.

Поставленная задача модификации системы с целью ускорения выполнения аналитических запросов была выполнена.

Список литературы

1. PostgreSQL: Documentation: 10: 19.8. Error Reporting and Logging [Электронный ресурс]. – URL: https://www.postgresql.org/docs/10/static/runtime-config-logging.html
2. pgBadger:: A fast PostgreSQL log analyzer Logging [Электронный ресурс]. – http://dalibo.github.io/pgbadger/
3. Крёнке Д. Теория и практика построения баз данных 8-е изд. – СПб: Питер, 2003.(Серия “Классика computer science”). – 800 с.
4. Саймон Р., Кросинг Х. Администрирование PostgreSQL 9. Книга рецептов – ДМК Пресс, 2015 – 364 с.
5. PostgreSQL: Documentation: 10: Using EXPLAIN [Электронный ресурс]. – URL: https://www.postgresql.org/docs/10/static/using-explain.html
6. Карпова Т. С. Базы данных: модели, разработка, реализация – СПб: Питер, 2002 – 304 с.
7. Дейт К. Введение в системы баз данных. – Вильямс, 2017 – 1328 с.
8. Уилсон Р. Д., Редмонд Э. Семь баз данных за семь недель. Введение в современные базы данных и идеологию NoSQL – ДМК Пресс, 2018 – 384 с.
9. PostgreSQL: Documentation: 10: Chapter 11. Indexes [Электронный ресурс]. – URL: https://www.postgresql.org/docs/10/static/indexes.html
10. Диго С. М. Проектирование и использование баз данных: учебник для вузов по направлению и специальности “Информ. Системы в экономике”. – М.: Финансы и статистика, 1995. – 207 с.
11. PostgreSQL: Documentation: 10: CREATE INDEX [Электронный ресурс]. – URL: https://www.postgresql.org/docs/current/static/sql-createindex.html

Приложение 1

Запросы, занявшие больше всего времени во время первичного нагрузочного тестирования.

Запрос 1:

```
select a.name
      ,sum(r.count) as sum_count
  from stk_agent a
  join stk_document d on a.ouid = d.id_agent
  join stk_doc_position p on d.ouid = p.id_doc
  join stk_regint r on p.ouid = r.id_srcobj
  join stk_good g on p.id_good = g.ouid
  where d.ouid = $1
    and r.ismov = 't'
    and r.ts between $2 and $3
group by a.name
 ,g.ouid
 ,g.name;
```

Запрос 2:

```
insert into stk_regint (id_good, id_cell, count,
id_srcobj, ismov, isres)
  select
    sdp.id_good,
    sdp.id_cell,
    sdp.count,
    sdp.ouid,
    'f',
    't'
  from stk_document d
    join stk_doc_position sdp on d.ouid = sdp.id_doc
```

```
    join stk_doc_type sdt on d.id_type = sdt.ouid
  where d.ouid = $1;
```

Запрос 3:

```
select
  t.ouid  as stk_cell_ouid,
  t.name  as stk_cell_name,
  s2.ouid as stk_stock_ouid,
  s2.name as stk_stock_name
from stk_cell t
  join stk_stock s2 on t.id_stock = s2.ouid
  left join stk_regmat sr on t.ouid = sr.id_cell
                             and sr.ismov = 't'
where sr.ouid is null
group by t.ouid
  , t.name
  , s2.ouid
  , s2.name;
```