МИНИСТЕРСТВО ОБРАЗОВАНИЯ РЕСПУБЛИКИ БЕЛАРУСЬ
БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
ФАКУЛЬТЕТ ПРИКЛАДНОЙ МАТЕМАТИКИ И ИНФОРМАТИКИ
Кафедра информационных систем управления

ГУБКО Антон Павлович

ОПТИМИЗАЦИЯ ЗАПРОСОВ В ОБЪЕКТНОРЕЛЯЦИОННЫХ СУБД

Дипломная работа

Научный руководитель:

Кандитат физико-математических наук, доцен

А.Н. Исаченко

Допущена к защите

«____»_____20___г.

Заведующий кафедрой информационных систем управления

доктор технических наук, доцент А.М. Недзьведь

Минск, 2025

ОГЛАВЛЕНИЕ

РЕФЕРАТ

Структура и объём дипломной работы

45 страницы, 24 рисунка, 2 приложения, 13 источников

Ключевые слова: ОПТИМИЗАЦИЯ БАЗ ДАННЫХ, POSTGRESQL, ORACLE, ПЛАНИРОВЩИК ЗАПРОСОВ, ИНДЕКСЫ, АЛГОРИТМЫ JOIN.

Текст реферата

Объект исследования – планировщик запросов PostgreSQL и Oracle.

Предмет исследования – средства оптимизации баз данных PostgreSQL и Oracle, различные подходы к созданию индексов и использованию алгоритмов join.

Цель исследования – анализ и сравнение механизмов оптимизации запросов в объектно-реляционных системах управления базами данных PostgreSQL и Oracle, оценка эффективности применяемых методов планирования выполнения запросов, индексирования и алгоритмов соединения таблиц, а также выработка рекомендаций по написанию эффективных SQL-запросов.

Методы исследования – анализ учебного материала, разработка приложений, анализ изученного материала.

Полученные результаты и их новизна – представлен комплексный сравнительный анализ оптимизаторов запросов двух ведущих объектно-реляционных СУБД (PostgreSQL и Oracle) с акцентом на практическое поведение планировщиков при различных типах запросов.

Достоверность материалов и результатов дипломной работы – дипломная работа является завершенной, поставленные задачи выполнены.

в полной мере, присутствует возможность дальнейшего развития исследования.

Область возможного практического применения – разработка эффективных приложений.

РЭФЕРАТ

Структура і аб’ём дыпломнай працы

45 старонкі, 24 малюнка, 2 дадаткі, 13 крыніц

Ключавыя словы: АПТЫМІЗАЦЫЯ БАЗ ДАНЫХ, POSTGRESQL, ORACLE ПЛАНАВАЛЬНІК ЗАПЫТАЎ, ІНДЭКСЫ, АЛГАРЫТМ JOIN.

Змест працы

Аб’ект даследавання – планавальнік запытаў PostgreSQL і Oracle.

Прадмет даследавання – інструменты аптымізацыі баз даных PostgreSQL і Oracle, розныя падыходы да стварэння індэксаў і прымянення алгарытмаў join.

Мэта даследавання – аналіз і параўнанне механізмаў аптымізацыі запытаў у аб’екта-рэлейных СУБД PostgreSQL і Oracle, ацэнка эфектыўнасці метадаў планавання выканання запытаў, індэксавання і алгарытмаў злучэння табліц, а таксама распрацоўка рэкамендацый па напісанні прадуктыўных SQL-запытаў.

Метады даследавання – аналіз спецыялізаванай літаратуры, распрацоўка прататыпаў і прыкладанняў, аналіз атрыманых дадзеных.

Атрыманыя вынікі і іх навізна – прадстаўлены комплексны параўнальны аналіз аптымізатараў запытаў двух вядучых аб’екта-рэляцыйных СУБД (PostgreSQL і Oracle) з акцэнтам на рэальнае паводзіны планіроўшчыкаў пры розных тыпах запытаў.

Дакладнасць матэрыялаў і вынікаў дыпломнай працы – дыпломная праца завершаная, пастаўленыя задачы выкананы ў поўнай меры, ёсць перспектывы для далейшага развіцця даследавання.

Вобласць магчымага практычнага прымянення – распрацоўка праграмных рашэнняў.

SUMMARY

Structure and scope of the diploma work

45 pages, 24 figures, 2 appendixes, 13 references

Keywords: DATABASE OPTIMIZATION, POSTGRESQL, ORACLE, QUERY PLANNER, INDEXES, JOIN ALGORITHMS.

Summary text

The object of the research – PostgreSQL and Oracle query planner.

The subject of the research – The optimization tools in PostgreSQL and Oracle databases, including various approaches to index creation and the use of join algorithms.

The aim of the research – To analyze and compare the query optimization mechanisms in the object-relational database management systems PostgreSQL and Oracle; to evaluate the effectiveness of query planning methods, indexing strategies, and join algorithms; and to develop recommendations for writing efficient SQL queries.

Research methods – analysis educational material, developing applications, analyzing the studied material.

The results of the work and their novelty – A comprehensive comparative analysis of the query optimizers in two leading object-relational DBMSs (PostgreSQL and Oracle) was presented, focusing on the practical behavior of query planners across different query types.

Authenticity of the materials and results of the diploma work – The thesis is complete and all research objectives have been fully achieved. The study provides a solid foundation for further research development.

Recommendations on the usage – Development of efficient software applications.

ВВЕДЕНИЕ

На сегодняшний день с ростом количества информации, созданием все более масштабных информационных систем требуются хранить все большие объемы данных и, соответственно, возрастает потребность в разработке эффективных баз данных и написании быстрых SQL-запросов.

Фундаментальное отличие между созданием программ на различных императивных языках программирования, когда описывается последовательность действий для получения результата, и написанием SQL-запросов состоит в том, что язык SQL является декларативным, то есть описывается результат, который требуется получить, а не то, как его получать. Так два запроса, выдающие одинаковые результат, могут выполняться по-разному, использовать различные ресурсы и выполнятся не одинаково по времени. Поэтому для разработчиков и администраторов баз данных критическими становится навыки написания эффективных запросов и умения анализировать планы запросов.

В рамках данной дипломной работы будет проведен анализ выполнения запросов с помощью планировщика и рассмотрены некоторые способы повышения эффективности выполнения запросов.

Глава 1.
Оптимизаторы запросов PostgreSQL и Oracle

Оптимизатор является наиважнейшим частью любой СУБД. Так как именно на него ложиться обязанность по выбору оптимального плана выполнения любого запроса. Команды разработчиков PostrgeSQL и Oracle разошлись в своих подходах к реализации этого механизма.

В Oracle в настоящее время используется так называемый cost-based optimizer, который по своей сути ведет себя так же, как оптимизатор PostgreSQL, опираясь, как следует из названия, на стоимость операций. Он вычисляет стоимость всех возможных вариантов исполнения оператора SQL и выбирает один вариант с наименьшей стоимостью [13]. Данный вариант пришел на смену rule-based optimizer, суть которого в том, что планировщик выбирал планы согласно правилам, разработанным разработчиками СУБД. Однако и сейчас осталась возможность “подсказать” оптимизатору нужный план с помощью так называемых хинтов (дополнительных указаний в запросе в базу данных, позволяющих явным образом влиять на план выполнения запроса).

В отличие от некоторых других систем управления базами данных, PostgreSQL сознательно не предоставляет подсказок оптимизатору. Это не упущение, а принципиальная позиция ключевой команды разработчиков. Они придерживаются мнения, что инвестиции в развитие самого планировщика запросов — то есть компонента, отвечающего за выбор наиболее эффективного способа выполнения запроса — гораздо важнее. Цель состоит в создании системы, способной самостоятельно определять оптимальный путь выполнения без необходимости внешнего вмешательства. Такой подход привел к тому, что оптимизатор запросов PostgreSQL считается одним из лучших среди как коммерческих, так и открытых СУБД. Именно эта особенность привлекает многих опытных разработчиков баз данных к использованию Postgres. Более того, высокое качество оптимизатора стало одной из причин, по которой исходный код PostgreSQL был выбран в качестве основы для разработки нескольких коммерческих баз данных**.** В контексте PostgreSQL это означает повышенную важность декларативного написания SQL-инструкций. Пользователю следует сосредоточиться на том, что он хочет получить, а не на том, как это должно быть получено, полностью доверяя оптимизатору выполнение его задачи по выбору наиболее производительного плана.

1.1 Как оптимизатор создает план выполнения

Непосредственно в документации PostgreSQL прописана главная задача планировщика - построить наилучший план выполнения [11]. Если это не требует больших вычислений, оптимизатор запросов будет перебирать все возможные варианты планов, чтобы в итоге выбрать тот, который должен выполниться быстрее остальных. Несмотря на развитость оптимизаторов, для сложных запросов, включающих множество операций, количество возможных планов выполнения становится огромным. Полный перебор всех вариантов попросту невозможен, ведь время, затраченное на сам процесс выбора оптимального плана, напрямую добавляется к общему времени выполнения запроса. Именно поэтому оптимизаторы активно используют эвристики — практические правила и подходы, которые позволяют существенно сократить количество рассматриваемых планов, сохраняя при этом высокую вероятность нахождения близкого к оптимальному решения.

Для формирования плана выполнения оптимизатор оперирует набором правил преобразования, упомянутыми эвристиками и алгоритмами оптимизации на основе стоимости. Правила преобразования служат для трансформации одного плана в другой, предположительно более эффективный. Например, существует общее правило, согласно которому операции фильтрации и проекции должны выполняться как можно раньше в цепочке операций. Это связано с тем, что они уменьшают объем обрабатываемых данных, что, в свою очередь, снижает затраты последующих операций. Оптимизатор может переупорядочить операции в плане, чтобы обеспечить такое раннее выполнение. Конечная задача алгоритма оптимизации заключается в выборе того плана, который имеет наименьшую расчетную стоимость, основываясь на статистике данных и моделях затрат.

1.2 Причины большого количества планов выполния

Как уже упоминалось ранее, одна и та же SQL-команда может быть реализована множеством различных способов, каждый из которых представлен отдельным планом выполнения. Таких вариантов может существовать огромное количество – от сотен до миллионов – в зависимости от сложности запроса.

Различия между планами заключаются, в частности, в:

- Последовательности выполнения операций;
- Используемых алгоритмах соединения (например, вложенные циклы, хеш-соединения и т.д.);
- способах доступа к данным (например, через индексы или посредством полного сканирования таблиц).

Оптимизатор СУБД подбирает наиболее эффективный план сравнивая оценочную стоимость различных вариантов. Даже для относительно простого запроса, включающего соединение трех основных алгоритмов (см. Главу 3) и 12 вариантов порядка выполнения, можно получить 108 различных планов. С учетом вариантов доступа к данным к каждой таблицы количество планов возрастает в разы[2].

Тем не менее современные СУБД, такие как Oracle и PostgreSQL, не просматривают все возможные комбинации. Вместо этого они используют стратегию, основанную на принципе оптимальности: если определенный подзапрос имеет оптимальное решение, то его можно использовать как часть более крупного оптимального плана. План выполнения строится иерархически – от наиболее простых поддеревьев (например, доступа к одной таблице) к более сложным структурам, включающим соединения и фильтрации. На каждом этапе производится ограниченное сравнение стоимости между альтернативами.

Хотя сам алгоритм не рассматривает все существующие варианты, он остается исчерпывающим в рамках ограниченного пространства решений. Так, выбрав оптимальны способ извлечения данных одной таблицы, система игнорирует все планы, где он не используется. При этом общее число анализируемых планов все равно может быть значительным. Для снижения вычислительной нагрузки применяются эвристики, которые отсеивают маловероятные с точки зрения эффективности решения. Это ускоряет работу оптимизатора, но связано с риском пропуска наиболее выгодного плана, если он был исключен до стадии сравнения. Таким образом, система находит лучший вариант среди оставшихся, даже если глобально0оптимальный план был не учтен.

1.3 Команды EXPLAIN и EXPLAIN PLAN

Однако оптимизатор также может допускать ошибки при выборе плана, точность вычисления оптимального плана напрямую зависит от доступа к статистическим данным системы [1]. Именно поэтому для разработчика критически важно умение читать планы и анализировать планы запроса. Для того, чтобы узнать какой план запроса был выбран в PostgreSQL необходимо воспользоваться командой EXPLAIN. Согласно официальной документации PostgreSQL, команда EXPLAIN служит для отображения плана выполнения запроса, разработанного планировщиком. Этот план подробно описывает, как будут сканироваться таблицы, задействованные в операции (например, с помощью простого последовательного сканирования или сканирования по индексу). Если запрос обращается к нескольким таблицам, план также укажет, какие алгоритмы соединения будут использоваться для объединения необходимых строк из каждой таблицы.

Параметры команды EXPLAIN:

Команда EXPLAIN поддерживает ряд параметров, позволяющих получить более детальную или специализированную информацию о плане выполнения:

ANALYZE:

При использовании этого параметра EXPLAIN фактически выполняет запрос и отображает реальное время работы, количество обработанных строк и другую статистику. По умолчанию этот параметр отключен (FALSE).

VERBOSE:

Активация этого параметра включает дополнительную информацию о плане. В частности, это включает перечень выходных столбцов для каждого узла в дереве плана, полные имена таблиц и функций со схемой, а также идентификатор запроса, если он был сгенерирован. По умолчанию параметр выключен.

COSTS:

Этот параметр (включен по умолчанию — TRUE) отображает информацию о предполагаемой начальной и общей стоимости каждого узла плана, а также ожидаемое количество строк и их среднюю ширину.

SETTINGS:

Если этот параметр включен, EXPLAIN показывает информацию о параметрах конфигурации, влияющих на планирование запроса, особенно тех, чьи значения отличаются от встроенных по умолчанию. По умолчанию отключен.

GENERIC_PLAN:

Позволяет команде содержать параметризованные заполнители (например, $1) и генерировать общий план, который не зависит от конкретных значений этих параметров. Важно отметить, что GENERIC_PLAN нельзя использовать в сочетании с ANALYZE. По умолчанию отключен.

BUFFERS:

Включает подробную информацию об использовании буферного кеша. Это включает данные о количестве попаданий (когда блок найден в кеше), прочитанных, загрязненных (измененных) и записанных общих, локальных и временных блоков. Также указывается время, затраченное на операции ввода/вывода (если track_io_timing включен). Общие блоки относятся к обычным таблицам и индексам, локальные — к временным таблицам, а временные — к рабочим данным для сортировок или хешей. Количество блоков, отображаемое для верхнего узла, агрегирует данные со всех его дочерних узлов. В текстовом формате выводятся только ненулевые значения. По умолчанию параметр выключен.

WAL:

Если включен ANALYZE, этот параметр предоставляет информацию о записях журнала упреждающей записи (WAL). Он показывает количество

записей, количество полных изображений страниц (FPI) и объем сгенерированного WAL в байтах. В текстовом выводе также отображаются только ненулевые значения. По умолчанию отключен.

TIMING:

При включенном ANALYZE этот параметр добавляет в вывод фактическое время запуска и время, затраченное на выполнение каждого узла плана. Отключение этого параметра может быть полезно, если точное время на уровне узлов не критично, поскольку многократное чтение системных часов может замедлять выполнение запроса. Общее время выполнения запроса измеряется всегда, независимо от этого параметра. По умолчанию включен (TRUE).

SUMMARY:

Включает сводную информацию о запросе (например, общее время) после основного плана. Сводка автоматически включается при использовании ANALYZE, но может быть включена и отдельно. Время планирования для EXPLAIN EXECUTE включает время извлечения плана из кеша и, при необходимости, время перепланирования.

FORMAT:

Определяет формат вывода плана: TEXT (по умолчанию), XML, JSON или YAML. Нетекстовые форматы удобны для программного анализа[12].

В Oracle получить предполагаемый план выполнения запроса можно с помощью команды EXPLAIN PLAN со следующим синтаксисом:

```
EXPLAIN PLAN
   [ SET STATEMENT_ID = string ]
   [ INTO [ schema. ] table [ @ dblink ] ]
FOR statement ;
```

рисунок 1.1 Синтаксис команды EXPLAIN PLAN

Однако отметим, что команда EXPLAIN предоставляет план запроса, который предполагается планировщиком, для того чтобы узнать, по какому плану в действительности выполняется запрос необходимо, как описано выше, использовать параметр ANALYZE, хотя зачастую эти планы совпадают.

В Oracle такая возможность отсутствует, так как команды EXPLAIN с параметром ANALYZE, по сути, собирает статистику в реальном времени, в то время как в Oracle статистка собирается всегда вручную (DBMS_STATS).

```
EXPLAIN (COSTS, BUFFERS) SELECT first_name, last_name from users
```

рисунок 1.2 Пример команды EXPLAIN

1.4 Хинты

Как уже упоминалось выше, Oracle предоставляет пользователем возможность подсказывать предпочитаемый вариант исполнения запроса с помощью так называемых хинтов (см. Приложение В). То есть с помощью применения подсказки можно добиться изменения плана выполнения запроса [10]. Причем имеется широкий спектр, приведем лишь здесь лишь основные:

— общие цели оптимизатора

```
/*+ RULE */
/*+ ALL_ROWS */
/*+ FIRST_ROWS */
/*+ FIRST_ROWS(n) */
```

— порядок доступа

```
/*+ LEADING */
/*+ ORDERED */
```

— методы соединения

```
/*+ USE_HASH */
/*+ USE_NL */
/*+ USE_MERGE */
/*+ USE_HASH_AGGREGATION */
/*+ NATIVE_FULL_OUTER_JOIN */
/*+ INDEX_JOIN */
/*+ INDEX_COMBINE */
/*+ NUM_INDEX_KEYS */
```

— способы выполнения [под]запроса

```
/*+ DRIVING_SITE */
/*+ MATERIALIZE */
/*+ INLINE */
/*+ PRECOMPUTE_SUBQUERY */
```
[8]

Стоит также отметить важный факт, касающийся синтаксиса: в контексте использования подсказок оптимизатору в SQL, критически важно соблюдать синтаксис. Конкретно, наличие пробела между символом + и первой буквой самой подсказки имеет значение. Например, запись /*+ ALL_ROWS */ является корректной и будет воспринята оптимизатором, тогда как /*+ALL_ROWS */ без этого пробела считается неправильной и может быть проигнорирована системой. Это тонкий, но важный аспект синтаксиса, влияющий на применение оптимизационных инструкций[8].

1.5 Чтение планов выполнения

Выполним следующий запрос на рисунке 1.3

```
SELECT events.id, events.professor, events.place, events.time, events.time, usrs.first_name, usrs.last_name
FROM events
        JOIN attendance ON events.id = attendance.event_id
        JOIN (SELECT users.id, users.first_name, users.last_name
              FROM users
                       JOIN user_group_roles ON users.id = user_group_roles.user_id
              WHERE user_group_roles.group_id = 'b1de91a5-e9af-4f77-96c6-54a692df72f7'::uuid) AS usrs
             ON usrs.id = attendance.user_id
```

рисунок 1.3 Запрос с информацией о посещенных студентом занятиях

Причем таблицы имеют следующую структуру (с данными таблиц можно ознакомится в приложении А):

Таблица "users"

Поле	Тип данных	Ограничения
id	UUID	PRIMARY KEY
first_name	VARCHAR(100)	
last_name	VARCHAR(100)	NOT NULL
username	VARCHAR(100)	NOT NULL, UNIQUE
email	VARCHAR(100)	NOT NULL, UNIQUE
password	VARCHAR(1000)	NOT NULL
role	ENUM(RoleEnum)	NOT NULL

Таблица "groups"

Поле	Тип данных	Ограничения
id	UUID	PRIMARY KEY
number	INTEGER	NOT NULL
course	ENUM(CourseEnum)	NOT NULL

Таблица "user_group_roles"

Поле	Тип данных	Ограничения
id	INTEGER	PRIMARY KEY
user_id	UUID	FOREIGN KEY → users.id
group_id	UUID	FOREIGN KEY → groups.id
role	ENUM(GroupRoleEnum)	NOT NULL

Таблица "events"

Поле	Тип данных	Ограничения
id	UUID	PRIMARY KEY
professor	VARCHAR(30)	
name	VARCHAR(50)	NOT NULL
place	VARCHAR(20)	
week	ENUM(WeekEnum)	NOT NULL
weekday	ENUM(WeekDayEnum)	NOT NULL
time	TIME (с таймзоной)	NOT NULL
date	DATETIME (с таймзоной)	NOT NULL

Таблица "attendance"

Поле	Тип данных	Ограничения
id	UUID	PRIMARY KEY
user_id	UUID	FOREIGN KEY → users.id
event_id	UUID	FOREIGN KEY → events.id
attended	BOOLEAN	
promised	BOOLEAN	
important_skip	BOOLEAN	

	QUERY PLAN text
1	Nested Loop (cost=25.36..62.74 rows=124 width=612)
2	Join Filter: (users.id = user_group_roles.user_id)
3	-> Nested Loop (cost=25.22..56.03 rows=31 width=196)
4	-> Hash Join (cost=25.07..49.55 rows=31 width=48)
5	Hash Cond: (attendance.user_id = user_group_roles.user_id)
6	-> Seq Scan on attendance (cost=0.00..20.30 rows=1030 width=32)
7	-> Hash (cost=25.00..25.00 rows=6 width=16)
8	-> Seq Scan on user_group_roles (cost=0.00..25.00 rows=6 width...
9	Filter: (group_id = 'b1de91a5-e9af-4f77-96c6-54a692df72f7'::uu...
10	-> Index Scan using ix_events_id on events (cost=0.14..0.21 rows=1 widt...
11	Index Cond: (id = attendance.event_id)
12	-> Index Scan using ix_users_id on users (cost=0.14..0.20 rows=1 width=452)
13	Index Cond: (id = attendance.user_id)

рисунок 1.4 План выполнения запроса

Как видно из рисунка, в плане указывается каким алгоритмом производится операция и дополнительные сведения, так, например, в 4 строке указано, что слияние выполнятся по хешу, а ниже в строке 5 указано условие для этого соединения – совпадение хешей ID таблицт attendance и user_group_role. Кроме того, помимо наименований используемых алгоритмов, каждая строка в плане выполнения содержит дополнительные оценочные параметры, имеющие ключевое значение для анализа эффективности исполнения запроса. К ним относятся предполагаемая стоимость выполнения операции, ожидаемое количество строк, возвращаемых на выходе, а также прогнозируемая средняя ширина этих строк. Все эти показатели вычисляются на основании статистической информации, накопленной системой управления базами данных.

Следует отметить, что значения стоимости учитывают совокупные затраты, включая все предыдущие операции, лежащие ниже по иерархии плана. Для каждой операции формируются два показателя: первый отражает стоимость, необходимую для извлечения первой строки результата, в то время как второй оценивает затраты, связанные с получением полного набора выходных данных.

Прогнозируемые объёмы и характеристики выходных строк играют принципиальную роль при дальнейшей оценке стоимости вышестоящих

операций, поскольку именно на этих данных базируется выбор оптимальных методов обработки на следующих этапах построения плана выполнения.

Отметим также, что в некоторых инструментах, например, pgAdmin (инструмент с графическим интерфейсом для управления базами данных PostgreSQL. Он позволяет управлять серверами, выполнять SQL-запросы, визуализировать структуры данных и следить за производительностью базы) существует возможность графического отображения плана выполнения запроса. Тем не менее данное представление может быть трудночитаемым.

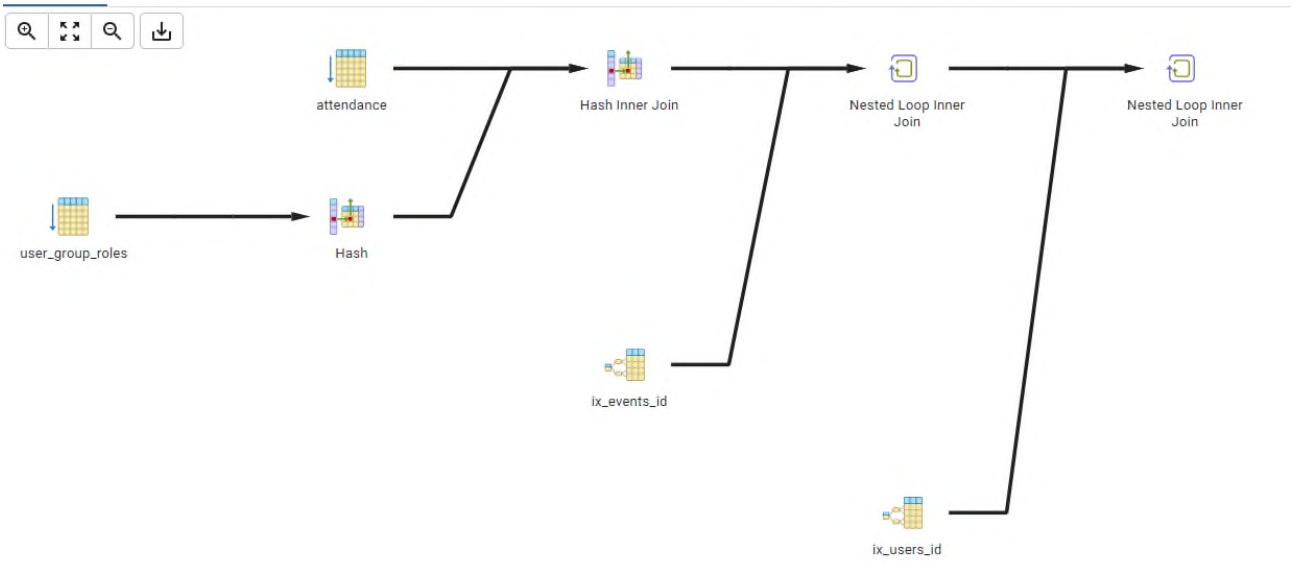


рисунок 1.5 Графическое отображение плана выполнения запроса

Глава 2.
Индексные структуры

Индексы являются важным вспомогательным механизмом, предназначенным для оптимизации доступа к данным в таблицах базы данных. По сути, индекс представляет собой отдельный сегмент, в котором сохраняются значения одного или нескольких столбцов таблицы, а также указатели на физическое местоположение соответствующих строк — так называемые идентификаторы строк (ROWID).

Индексы могут создаваться как для отдельных таблиц, так и для кластеров. При этом возможно наличие нескольких индексов на одну таблицу, каждый из которых характеризуется уникальной конфигурацией — будь то выбор столбцов или порядок их следования. Такие различия позволяют гибко адаптировать индексные структуры под различные типы запросов.

Современные системы управления базами данных поддерживают несколько типов индексов, каждый из которых имеет свои особенности применения. Наиболее часто используются B-деревья, которые обеспечивают эффективный доступ при произвольных значениях столбцов. Кроме них, применяются кластерные индексы, способствующие упорядоченному хранению данных, а также битовые (или масочные) индексы, которые оказываются особенно эффективными в случае столбцов с ограниченным числом уникальных значений.

Применение индексов значительно ускоряет выполнение операций выборки, позволяя системе не обращаться ко всей таблице, а находить требуемые записи точечно, что особенно важно при работе с большими объёмами данных и высокой нагрузке на систему[5].

Общую стоимость сканирования с применением индекса можно выразить следующей формулой:

Стоимость индексного сканирования = Стоимость обхода индекса + Стоимость чтения данных

Где:

Стоимость обхода индекса = Высота дерева индекса × Стоимость чтения страницы

Стоимость чтения данных = Количество найденных строк × Стоимость чтения строки

Сравнение с полным сканированием таблицы:

Стоимость полного сканирования таблицы=Количество страниц таблицы × Стоимость чтения страницы

2.1 B-деревья

Самая распространенная индексная структура – это B-дерево. Структура B-дерева организована в виде иерархии узлов, каждый из которых ассоциирован с определённым блоком, физически размещённым на диске. Листовые узлы содержат полные записи индекса — пары, состоящие из индексного ключа и указателя на соответствующую строку в таблице. Внутренние (не листовые) узлы, расположенные на всех уровнях дерева, за исключением нижнего, содержат ключи, представляющие минимальные значения, встречающиеся в соответствующих дочерних блоках, а также указатели на эти блоки. Все элементы в индексных блоках строго упорядочены, что обеспечивает эффективность навигации по дереву. Кроме того, важно отметить, что каждый блок в структуре дерева заполняется как минимум наполовину, что способствует поддержанию сбалансированной формы дерева и, как следствие, равномерной глубины.

Поиск конкретного ключа K в B-дереве начинается с корневого узла. На каждом уровне осуществляется поиск наибольшего ключа P, не превышающего K, после чего переходит к дочернему узлу, на который указывает соответствующий указатель. Этот процесс продолжается до тех пор, пока не будет достигнут листовой уровень, где можно либо найти ключ K, либо определить положение, в котором он мог бы находиться при наличии в индексе. Количество операций при поиске пропорционально глубине дерева.

Следует также подчеркнуть, что данная структура эффективно поддерживает диапазонный поиск — например, в контексте SQL-операции BETWEEN. После нахождения нижней границы диапазона последующее извлечение значений осуществляется последовательным чтением листовых узлов до тех пор, пока не будет достигнута верхняя граница. Эта же процедура используется в случае неуникальных индексов, где одному ключу может соответствовать множество строк: требуется последовательный обход всех указателей в пределах листовых блоков.

2.2 Битовые карты

Битовая карта представляет собой вспомогательную структуру данных, применяемую для оптимизации доступа к другим компонентам хранения, состоящим из нескольких блоков. Её можно рассматривать как особый тип индекса, предназначенный для компактного отображения свойств табличных данных.

Чаще всего битовая карта содержит по одному биту на каждый блок (обычно размером 8192 байта). Если блок обладает заданным свойством, соответствующий бит устанавливается в единицу; в противном случае — в ноль. Такая схема позволяет эффективно определять, какие блоки необходимо просматривать при выполнении запроса, особенно при использовании нескольких индексов. На рисунок ниже продемострировано, как битовые карты используются для доступа к данным по нескольким индексам.

Номер блока	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Индекс 1	0	1	0	0	0	1	0	0	1	0	0	0	0	0	0	1
Индекс 2	0	0	1	0	0	1	0	0	0	1	0	0	1	0	0	1
Логическое «и» ⇓																
	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	1

рисунок 2.1 Использование битовых карт для доступа к таблицам по нескольким индексам

При выполнении запроса с несколькими условиями отбора, механизм базы данных начинает с последовательного сканирования соответствующих индексов и формирования для каждого из них отдельной битовой карты. Эти карты отражают, какие блоки содержат строки, удовлетворяющие условиям конкретного индекса.

После построения битовых карт система выполняет побитовую операцию «И», позволяющую выделить те блоки, которые одновременно соответствуют обоим условиям. Далее происходит обращение лишь к тем блокам, для которых в результирующей битовой карте установлен бит в значение 1. Это позволяет избежать чтения блоков, содержащих строки, удовлетворяющие лишь одному из условий.

Следует, однако, учитывать, что наличие установленного бита не гарантирует присутствие нужной строки в блоке — он может содержать только частично релевантные данные. Тем не менее такой подход позволяет существенно сократить объём ненужных обращений к диску. Несмотря на компактность, битовые карты в случае очень крупных таблиц могут занимать значительный объём памяти, распределяясь на множество блоков.

Битовые карты отлично подходят для запросов к таблицам с малой кардинальностью данных, то есть данным с малым количеством уникальных записей.

PostgreSQL в отличие от Oracle не поддерживает битовые карты, однако предоставляет технику Bitmap Scan, которая на самом деле лишь симулирует использования битовой карты, на самом деле используя просто несколько индексов при возможности.

	QUERY PLAN text
1	Bitmap Heap Scan on orders (cost=4271.64..27316.56 rows=381834 width=13)
2	Recheck Cond: ((order_status)::text = 'Pending'::text)
3	-> Bitmap Index Scan on status_indx (cost=0.00..4176.18 rows=381834 width...
4	Index Cond: ((order_status)::text = 'Pending'::text)

рисунок 2.2 пример использования битовой карты оптимизатором

2.3 Другие виды индексов

Однако Следует отметить, что PostgreSQL обладает широким спектром индексных структур, предназначенных для работы с различными типами данных и классов поисковых выражений, многие из которых отсутствуют в системе Oracle.

В частности, хеш-индекс представляет собой структуру, основанную на применении хеш-функции, преобразующей значение ключа в адрес соответствующего индексного блока. Такая схема обеспечивает крайне эффективный доступ при проверке равенства — в этом случае производительность хеш-индекса может превосходить традиционные B-деревья.

Однако важно подчеркнуть, что хеш-индексы не пригодны для запросов, связанных с диапазонным поиском, поскольку хеш-функция полностью утрачивает информацию о порядке значений. Ещё одной отличительной особенностью является то, что стоимость обращения к хеш-индексу не зависит от его объёма, в отличие от B-деревьев, где эффективность поиска логарифмически связана с размером структуры. В Oracle же хеш-индекс можно лишь эмулировать с помощью партиций.

Индексы GIN(Generalized Inverted Index) — это обобщённый инвертированный индекс в базе данных PostgreSQL. Он предназначен для ускорения поиска в структурах данных, содержащих составные типы, например массивы или JSONB [4]. По своей сути, это обратный индекс, где для каждого уникального элемента хранится список указателей на записи, в которых он встречается. Данный индекс работает с неатомарными данными, индексируя при этом отдельные элементы (элементы ссылаются на значения, в которых они содержатся) [7]. Это даёт возможность быстро находить записи, соответствующие запросу. В Oracle для последней цели можно лишь использовать контексты.

Индексы GiST (Generalized Search Tree) — обобщённое поисковое дерево в PostgreSQL. Это базовый шаблон, на основе которого могут реализовываться

произвольные схемы индексации. GiST позволяет распределить данные любого типа по сбалансированному дереву и использовать это дерево для поиска по разным условиям [9]. Применяется для специфических типов данных: геометрия, сетевые адреса, диапазоны.

2.4. Создание индекса

Синтаксис создания индекса согласно официальной документации Postgresql:

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

рисунок 2.3 Синтаксис команды create index

Параметры:

UNIQUE:

Данный модификатор обязывает систему проверять уникальность значений в индексируемых столбцах. Проверка происходит как при создании индекса (для существующих данных), так и при каждой последующей операции добавления или изменения данных. Попытки внести или обновить данные, которые приведут к нарушению уникальности, будут отклонены с ошибкой. Отдельные ограничения действуют для уникальных индексов на секционированных таблицах.

CONCURRENTLY:

Использование этой опции позволяет PostgreSQL построить индекс без установки блокировок, препятствующих одновременным операциям вставки, обновления или удаления данных в целевой таблице. В отличие от стандартной сборки индекса, которая блокирует запись (но не чтение) до завершения процесса. Для временных таблиц CREATE INDEX всегда выполняется непараллельно, поскольку к ним нет одновременного доступа из других сессий, а непараллельное создание индекса является менее ресурсоемким.

IF NOT EXISTS:

Этот параметр предотвращает возникновение ошибки, если индекс с указанным именем уже существует. В таком случае будет выдано только уведомление. Важно отметить, что этот параметр не гарантирует идентичность существующего индекса с тем, который предполагалось создать. При использовании IF NOT EXISTS имя индекса должно быть указано явно.

INCLUDE:

Необязательное условие INCLUDE позволяет указать список столбцов, которые будут включены в индекс как неключевые. Неключевые столбцы не используются для квалификации поиска при сканировании индекса и игнорируются при проверке ограничений уникальности или исключения. Однако их наличие делает возможным сканирование только по индексу (index-only scan), позволяя запросам получать содержимое этих столбцов непосредственно из записи индекса без необходимости обращения к основной таблице (heap). Следует проявлять осторожность при включении неключевых столбцов, особенно если они содержат широкие данные, так как это увеличивает размер индексных кортежей и индекса в целом, потенциально замедляя операции поиска и вставки. Превышение максимального размера кортежа, разрешенного для типа индекса, приведет к ошибкам вставки. Кроме того, дедупликация B-дерева не применяется к индексам с неключевыми столбцами. Столбцы, указанные в INCLUDE, не требуют наличия соответствующих классов операторов. Выражения не поддерживаются в качестве включенных столбцов. В настоящее время данная функциональность поддерживается методами доступа B-tree, GiST и SP-GiST, где значения включенных столбцов сохраняются в конечных кортежах, но не участвуют в навигации по дереву верхнего уровня.

name: Имя создаваемого индекса. Указание имени схемы здесь не допускается; индекс всегда создается в той же схеме, что и его родительская таблица. Имя индекса должно быть уникальным в пределах схемы среди всех отношений. Если имя не указано, PostgreSQL автоматически генерирует подходящее имя на основе имени таблицы и индексируемых столбцов.

ONLY:

Данный параметр указывает, что индексы не должны рекурсивно создаваться для секций, если целевая таблица является секционированной. По умолчанию рекурсивное создание индексов на секциях включено.

table_name: Имя индексируемой таблицы, возможно, дополненное именем схемы.

method:

Указывает используемый индексный метод. Доступные варианты включают btree (метод по умолчанию), hash, gist, spgist, gin, brin, а также пользовательские методы доступа, такие как Bloom.

column_name:

Имя столбца таблицы, который будет индексироваться.

expression:

Выражение, основанное на одном или нескольких столбцах таблицы. Обычно выражение должно быть заключено в круглые скобки, но их можно опустить, если выражение представляет собой вызов функции.

collation:

Имя правила сортировки, которое будет использоваться для индекса. По умолчанию индекс наследует параметры сортировки, объявленные для индексируемого столбца или результата индексируемого выражения. Индексы с нестандартными параметрами сортировки полезны для запросов, использующих эти же параметры.

opclass:

Имя класса операторов, определяющего поведение индекса для конкретного типа данных и операторов.

opclass_parameter:

Имя параметра класса оператора.

ASC:

Указывает порядок сортировки по возрастанию (по умолчанию).

DESC:

Указывает порядок сортировки по убыванию.

NULLS FIRST:

Указывает, что значения NULL сортируются до не-NULL значений. Это поведение по умолчанию, если указано DESC.

NULLS LAST:

Указывает, что значения NULL сортируются после не-NULL значений. Это поведение по умолчанию, если DESC не указано.

NULLS DISTINCT / NULLS NOT DISTINCT:

Определяет, будут ли нулевые значения считаться различными (не равными) для целей уникального индекса. По умолчанию они считаются различными, что позволяет уникальному индексу содержать несколько NULL значений в столбце.

storage_parameter:

Имя параметра хранения, специфичного для выбранного индексного метода.

tablespace_name:

Имя табличного пространства, в котором будет создан индекс. Если не указано, используется default_tablespace или temp_tablespaces для индексов временных таблиц.

predicate: Выражение ограничения для создания частичного индекса.

```
create index status_indx on orders (order_id) where orders.order_status = 'Pending';
```

рисунок 2.4. Пример создания индекса с условием

В Oracle синтаксис создания индекса совпадает с PostgreSQL согласно [6]:

```
create index <index_name> on <table_name> ( <column1>, <column2>, … );
```

рисунок 2.5. Синтаксис создания индекса в Oracle

2.5 Когда не используются индексы

Несмотря на очевидную пользу индексов, не стоит создавать их при любой возможности. Индекс может только мешать, а последовательное сканирование таблицы будет более приемлемым. Существуют по меньшей мере две обоснованные ситуации, при которых применение индексных структур может оказаться неэффективным:

- Рассматриваемая таблица обладает малым объёмом, она может быть целиком загружена в оперативную память. В таком случае затраты на использование индекса могут превысить выгоду от него, поскольку последовательное чтение таблицы выполняется крайне эффективно;
- Результатом запроса становится значительная часть строк таблицы, преимущества индексного доступа нивелируются: система в любом случае будет вынуждена обращаться ко многим блокам данных, что делает полное сканирование более рациональным вариантом;
- К таблице часто применяются не только операции извлечения данных, но также вставки и удаления.

Последний пункт можно объяснить тем, что необходимо затрачивать ресурсы на обновление индекса, либо же индекс вовсе будет не актуальным.

Возникает вопрос: существует ли возможность избежать применения уже созданных индексов? В подавляющем большинстве случаев оптимизатор запросов способен самостоятельно определить целесообразность использования индексных структур, корректно выбирая между индексным доступом и полным сканированием.

Тем не менее, в отдельных ситуациях возможны ошибки в оценке, приводящие к неэффективному плану выполнения. В подобных случаях разработчик может повлиять на выбор плана косвенно — например, путём модификации условий фильтрации, что способно изменить оценки стоимости операций и побудить оптимизатор отказаться от использования определённого индекса.

Глава 3.
Алгоритмы JOIN

Одним из важнейших преимуществ реляционных баз данных является возможность сочетать данные из нескольких таблиц. Для этой цели используется оператор join, который позволяет объединять колонки из двух и нескольких строк, и, хотя существует много разновидностей этого оператора, все они использует одни и те же алгоритмы.

3.1 Вложенные циклы

Данный алгоритм реализует операцию декартова произведения, при которой каждая строка одной таблицы сопоставляется с каждой строкой другой [3]. Наиболее прямолинейный способ выполнения этой операции заключается в последовательном переборе всех строк первой таблицы R, для каждой из которых осуществляется полный проход по строкам второй таблицы S. Таким образом формируется полное множество всех возможных комбинаций строк из R и S. Псевдокод этого алгоритма представлен ниже:

```
FOR row1 IN table1 LOOP
    FOR row2 IN table2 LOOP
        Output(row)
    END LOOP
END LOOP
```

При рассмотрении фундаментальных операций в реляционных базах данных, таких как декартово произведение (R×S), его вычислительная сложность является ключевым аспектом. Время выполнения данного алгоритма напрямую пропорционально произведению мощностей входных множеств (таблиц), то есть O(rows(R)*rows(S)).Этот показатель представляет собой теоретическую нижнюю границу сложности для любых алгоритмов, реализующих декартово произведение, что означает принципиальную невозможность достижения более высокой производительности. Хотя различные реализации могут демонстрировать вариации в константах и коэффициентах, общая асимптотическая сложность остаётся неизменной.

Операция соединения, являясь более сложной, может быть концептуально представлена как декартово произведение, за которым следует операция фильтрации по определенному условию. Простейший и наиболее интуитивно понятный алгоритм её реализации – соединение вложенными циклами – по своей природе схож с декартовым произведением, поскольку он также оперирует всеми возможными парами строк из входных таблиц. Следовательно, его

вычислительная сложность также составляет O(rows(R)*rows(S)). несмотря на то что итоговый размер результирующей выборки, как правило, значительно меньше.

В контексте практического применения, особенно когда входные таблицы являются хранимыми данными, алгоритмы соединения могут быть оптимизированы путем интеграции с подсистемой доступа к данным. Несмотря на сохранение базовой асимптотической сложности, существуют модификации алгоритма вложенных циклов, направленные на минимизацию количества операций ввода-вывода. Примером такой оптимизации является блочная обработка, когда внешний цикл итерирует по блокам первой таблицы, а внутренний – по содержащимся в них строкам. Более продвинутые подходы ориентированы на загрузку нескольких блоков внешней таблицы в оперативную память и последующую обработку соответствующих строк из внутренней таблицы за один проход, что существенно снижает накладные расходы на дисковые операции и повышает общую эффективность выполнения запроса.

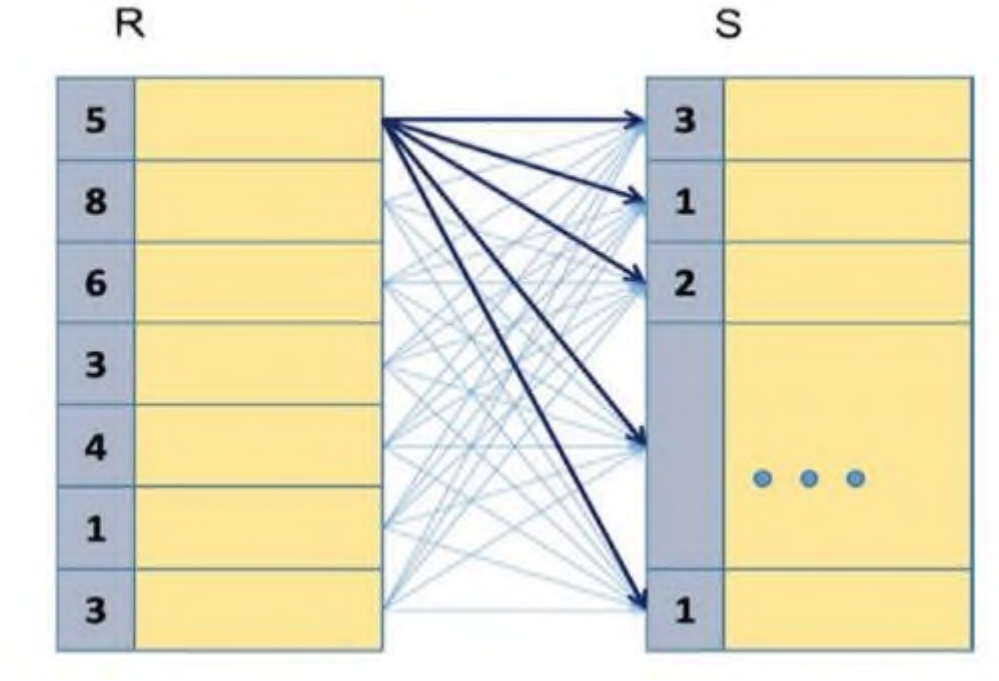


рисунок 3.1 Алгоритм вложенного цикла

Рассмотренные ранее алгоритмы соединения демонстрируют универсальность, работая с произвольными условиями. Однако на практике чаще всего встречаются естественные соединения, где условие требует равенства соответствующих атрибутов в обеих таблицах (R и S). В таких случаях эффективность алгоритма соединения вложенными циклами можно значительно повысить за счет использования индексированного доступа к данным таблицы S.

Если по атрибутам, участвующим в условии соединения, в таблице S создан индекс, то для каждой строки из таблицы R внутренний цикл алгоритма существенно сокращается, поскольку система может напрямую найти нужные строки в S, используя индекс. В идеальном случае, если индекс на атрибуте соединения в таблице S является уникальным (например, это первичный ключ), то внутренний цикл может фактически исчезнуть, так как для каждой строки R будет найдена максимум одна соответствующая строка в S.

Алгоритм соединения вложенными циклами с индексным доступом обычно предпочтителен, когда число строк в таблице R (внешнем цикле) невелико. Однако его эффективность снижается по мере увеличения объема данных, подлежащих обработке, поскольку накладные расходы на индексированный доступ начинают превышать выгоду.

Важно отметить, что для операций декартова произведения и соединений с произвольными условиями теоретически доказано, что алгоритмы вложенных циклов являются наиболее производительными. Тем не менее, для специфических типов условий соединения всегда остается открытым вопрос о существовании более подходящих, специализированных алгоритмов, способных превзойти общие подходы.

3.2 Алгоритмы на основе хеширования

В рамках естественного соединения (natural join) результатом являются пары строк из таблиц R и S, для которых значения соответствующих атрибутов равны. Принципиальная идея алгоритма соединения хешированием (hash join) базируется на свойстве хеш-функций: если значения атрибутов идентичны, то и их хеш-значения будут совпадать. Суть алгоритма хеширования заключается в следующем: обе входные таблицы разделяются на группы (корзины) на основе применения хеш-функции к значениям атрибутов, по которым осуществляется соединение. После этого соединение строк выполняется независимо внутри каждой сформированной корзины. Это позволяет существенно сократить объем данных, подлежащих сравнению, поскольку строки из разных корзин гарантированно не будут соответствовать условию соединения.

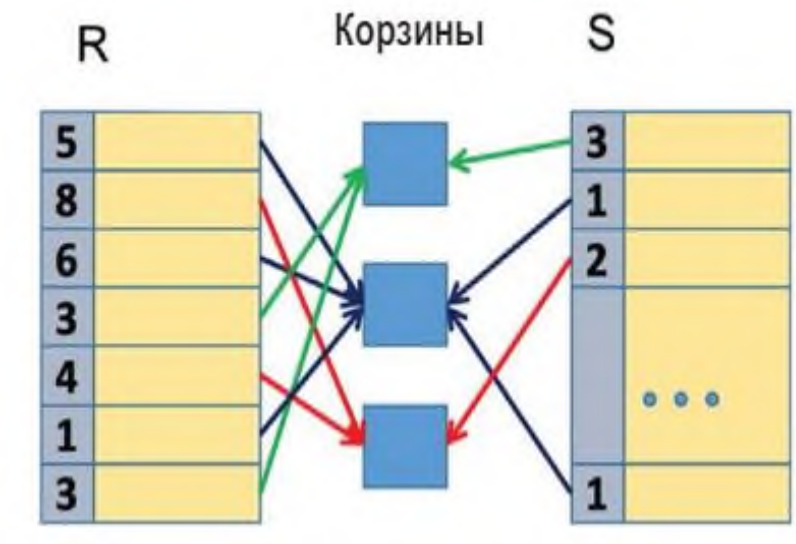


рисунок 3.2 Алгоритм соединения хешированием

Базовый алгоритм хеш-соединения выполняется в две основные фазы:

1. Фаза построения: На этом этапе все записи из "меньшей" входной таблицы (обычно обозначаемой как R) хешируются по значению атрибута, используемого для соединения. Результатом являются хеш-корзины, где каждая корзина содержит записи с одинаковым хеш-значением.
2. Фаза проверки: После построения хеш-таблицы, каждая запись из "большей" входной таблицы (обычно S) также хешируется по тому же атрибуту. Затем эта запись направляется в соответствующую хеш-корзину. Если в этой корзине обнаруживаются совпадающие записи из таблицы R, то формируются выходные строки, соответствующие условию соединения. Простейший подход для поиска совпадений внутри корзины – это перебор (фактически, мини-вложенный цикл) всех записей в ней.

На плане выполнения запроса эти две фазы обычно отображаются как отдельные физические операции. Приближенная оценка стоимости хеш-соединения может быть выражена формулой:

cost(hash,R,S) = size(R) + size(S) + size(R)*size(S)/size(JA),

где JA обозначает атрибут, по которому осуществляется соединение. Первые два слагаемых в этой формуле отражают затраты на однократное сканирование обеих таблиц R и S. Последнее слагаемое, size(JA)/size(R)·size(S), представляет собой примерный размер генерируемого результирующего набора данных. Важно отметить, что стоимость вывода результата присуща всем алгоритмам соединения, но в оценке стоимости алгоритма вложенных циклов ею часто пренебрегают, так как она значительно меньше, чем его основные затраты.

Эта формула наглядно демонстрирует, что алгоритм хеш-соединения демонстрирует значительное преимущество перед алгоритмом вложенных циклов при работе с большими таблицами, особенно когда количество уникальных значений атрибута соединения велико. Например, если атрибут соединения уникален в одной из таблиц (например, является первичным ключом), то последнее слагаемое в формуле будет примерно равно размеру другой таблицы, что указывает на высокую эффективность. Базовая версия хеш-соединения эффективно работает при условии, что все хеш-корзины, созданные на этапе построения, полностью помещаются в доступную оперативную память. Для сценариев, когда входные таблицы слишком велики и не могут быть целиком загружены в память, применяется модифицированный подход, известный как гибридное хеш-соединение. Этот вариант алгоритма сначала делит обе таблицы на более мелкие разделы (партиции) таким образом, чтобы каждый отдельный раздел мог поместиться в память. Затем базовый алгоритм хеш-соединения последовательно применяется к каждой соответствующей паре разделов. Стоимость гибридного хеш-соединения выше, чем у базового, так как требуется временное хранение разделов на диске и двукратное сканирование обеих таблиц. Однако ключевым преимуществом остается то, что общая стоимость по-прежнему пропорциональна сумме размеров входных таблиц, а не их произведению, что делает его значительно эффективнее вложенных циклов для больших объемов данных.

3.3 Сортировка слиянием

Еще один алгоритм для естественных соединений, который называют сортировкой слиянием, показан на рисунке 3.3.

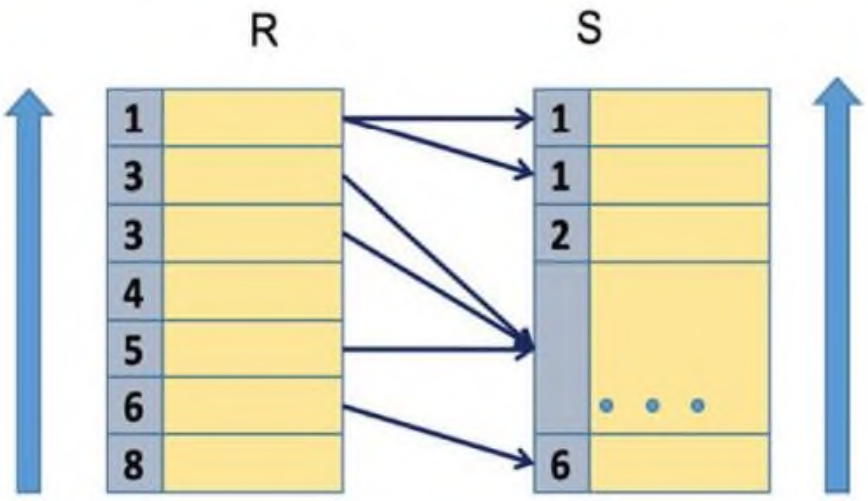


рисунок 3.3 Алгоритм сортировки слиянием

На первом этапе алгоритма обе входные таблицы сортируются в порядке возрастания по атрибутам соединения. После того как таблицы правильно упорядочены, на этапе слияния обе таблицы сканируются один раз и для каждого значения атрибута соединения вычисляется декартово произведение строк, содержащих это значение. Отметим, что это произведение является необходимой частью результата соединения. Новые строки с таким же значением атрибута не могут появиться в оставшейся части входных данных, потому что таблицы упорядочены.

Стоимость фазы слияния можно выразить той же формулой, что и для соединения хешированием, то есть она пропорциональна сумме размеров входных и выходных данных. Фактическая стоимость несколько ниже, потому что нет необходимости в этапе построения хеш-таблицы. Стоимость сортировки можно оценить следующей формулой:

size(R)*log(size(R)) + size(S)*log(size(S))

Алгоритм сортировки слиянием особенно эффективен, если одна или обе входные таблицы уже отсортированы. Это может произойти в серии соединений по одним и тем же атрибутам.

3.4 Сравнение алгоритмов

Как и в случае с методами доступа к данным, при выборе алгоритма соединения нет универсально лучшего варианта. Всё зависит от конкретных обстоятельств. Алгоритм вложенных циклов более гибок и оптимален для небольших соединений, особенно если есть возможность использовать индексы. А вот сортировка слиянием и хеширование показывают себя гораздо эффективнее при работе с большими таблицами, при условии, что они вообще применимы в данном сценарии. Следует так же помнить, что лишь алгоритм вложенного цикла может быть использован вместе с условиями типа неравенства.

Операцию соединения таблиц можно выразить формулой:

Общая стоимость соединения:

Стоимость соединения = Стоимость построения + Стоимость поиска

Для каждого алгоритма имеем:

Hash Join:

Стоимость построения = Количество строк в таблице построения × Стоимость обработки строки

Стоимость поиска = Количество строк в таблице поиска × Стоимость поиска в хеше

Merge Join:

Стоимость Merge Join = Стоимость сортировки таблицы A + Стоимость сортировки таблицы B + Стоимость слияния

Nested Loops:

Стоимость Nested Loops = Количество строк во внешней таблице × Количество строк во внутренней таблице × Стоимость сравнения строк

Можно так же рассмотреть использование обеих техник, а именно применение алгоритма хеширования с использованием индекса. Стоимость данной операции можно описать формулами:

Расчет селективности соединения:

Селективность соединения = (| A| ×| B|) / Количество строк, удовлетворяющих условию

Кардинальность соединения:

Кардинальность соединения =| A| ×| B| ×Селективность соединения

Стоимость построения индекса:

Стоимость построения индекса = Количество строк в таблице × Стоимость добавления строки в индекс

Стоимость использования индекса:

Стоимость индексного сканирования = Обход индекса + Чтение данных по индексу

Стоимость соединения (Hash Join):

Стоимость объединения по хешу = стоимость построения + стоимость поиска

3.5 Сравнение реализаций алгоритмов в PostgreSQL и Oracle

Обе СУБД поддерживают и активно используют перечисленные выше алгоритмы. Различия заключаются лишь в особенностях их использования. Так в Oracle активно применяется параллелизм при обработке больших таблиц, также пользователям позволяется влиять на решение планировщика запросов используя хинты, тогда как в PostgreSQL имеется возможность лишь полностью отключить использование какого-либо из алгоритмов с использованием команд типа enable_mergejoin=False.

Глава 4.
Использование индексных структур и алгоритмов join на практике

4.1 Использование индексных структур

Выполним следующий запрос в СУБД PostgreSQL

```
1. SELECT order_id, order_status
2.     FROM public.orders
3.     where orders.order_status ='Pending';
```

Данный запрос извлекает данные из таблицы заказов со статусом «в обработке». Отметим также, кардинальность извлекаемых данных составляет 0.3 при общем объеме таблицы в 1 000 000 записей со следующей структурой (с данными таблицы можно ознакомится в приложении Б):

Таблица "orders"

Поле	Тип данных	Ограничения
order_id	UUID	PRIMARY KEY
customer_id	UUID	FOREIGN KEY → customers.id
order_status	ENUM	NOT NULL
order_date	TIMESTAMP	NOT NULL
total_amount	DECIMAL(10,2)	
payment_status	ENUM	

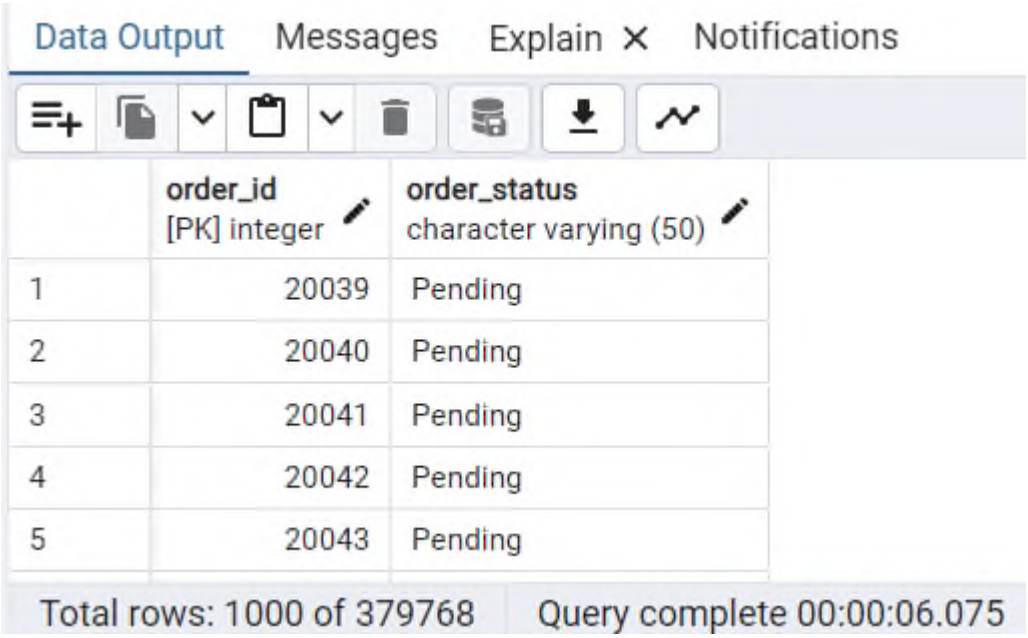


рисунок 4.1 Результат выполнения запроса без индексов

Видим, что запрос выполнился за 6.075 секунд. Рассмотрим план данного запроса с помощью команды EXPLAIN.

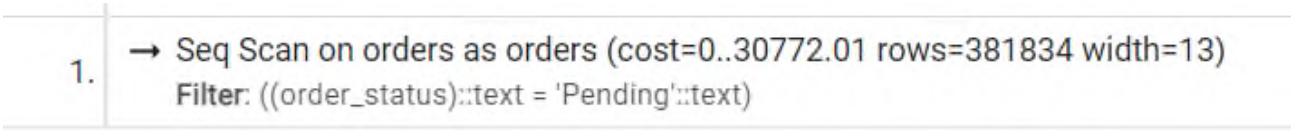


рисунок 4.2 План запроса без индекса

Запрос выполняется последовательным сканированием, стоимость получения первой строки составляет 0, последней - 30772

Теперь создадим индекс по столбцу order_status выполнив команду

```
create index status_indx on orders (order_status)
```

и выполним тот же запрос.

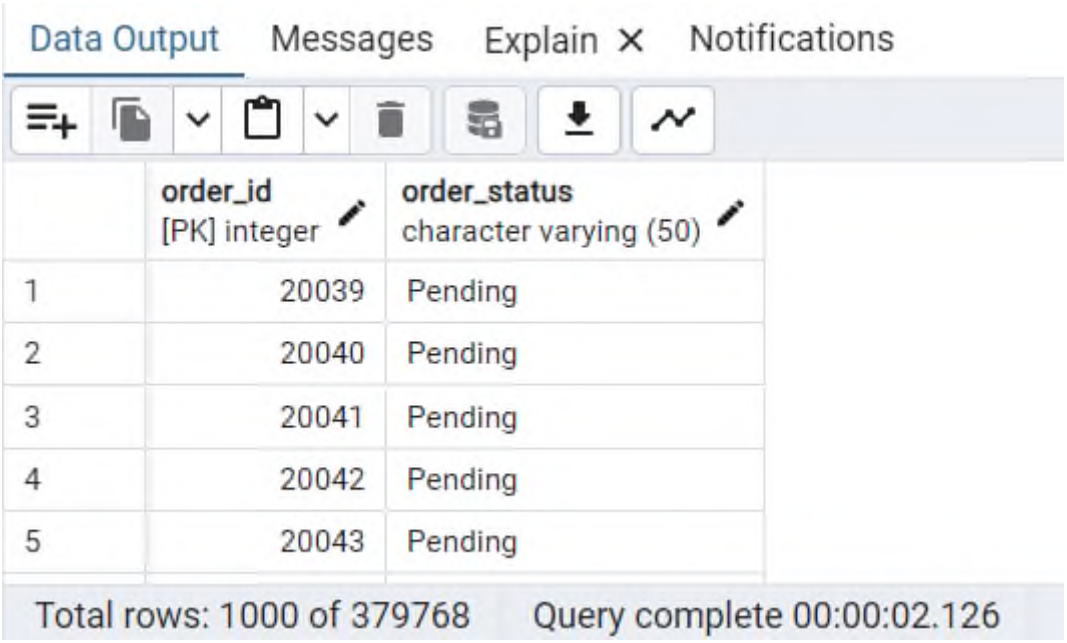


рисунок 4.3 Результат выполнения запроса с индексом

Время выполнения запроса сократилась до 2.126 секунд, а план запроса теперь выглядит, как показано н рисунке 4.4.

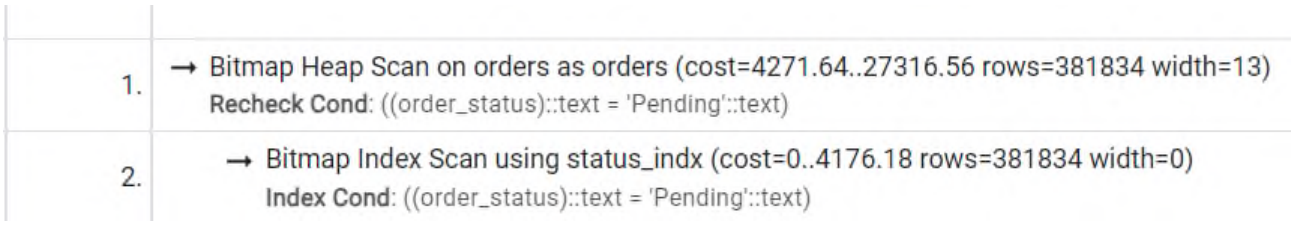


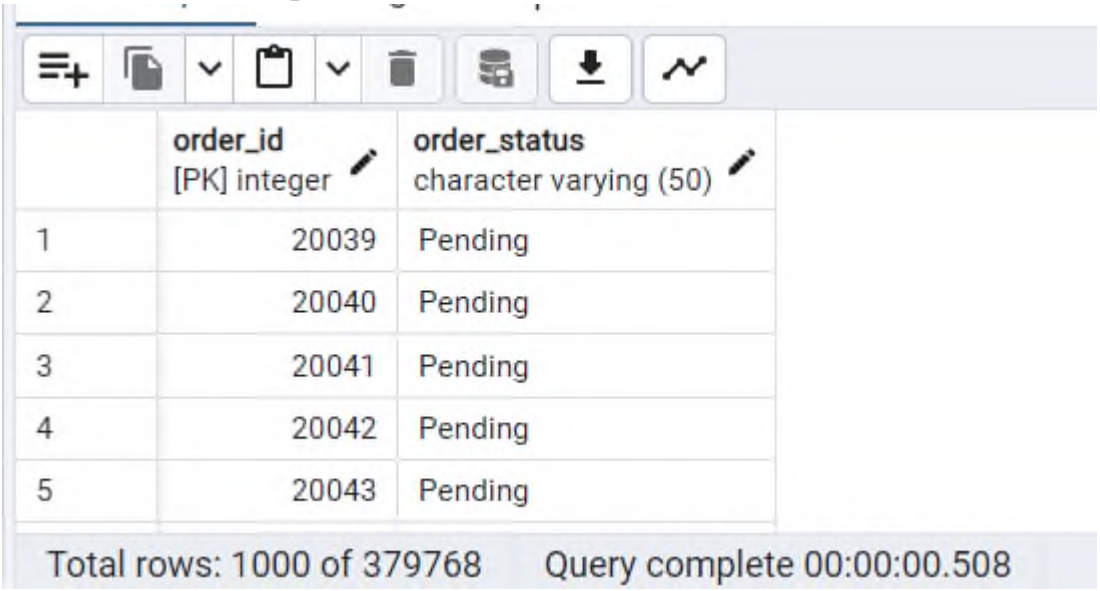
рисунок 4.4 План запроса с индексом

Теперь для выполнения данного запроса используется битовая карта, обновились и стоимости получения записей: с одной стороны, стоимость возросла для получения первой записи - 4271, однако в целом запрос выполняется быстрее, так как получить последнюю строку стоит - 27316.

Удалим теперь данный индекс, создадим индекс по первичному ключу таблицы с условием на столбец order_status

```
create index stutus_cond_indx on orders (order_id) where order_status = 'Pending';
```

и вновь выполним запрос.

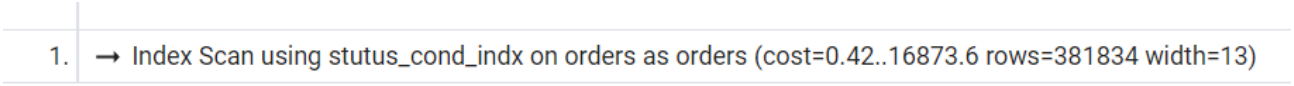


	order_id [PK] integer	order_status character varying (50)
1	20039	Pending
2	20040	Pending
3	20041	Pending
4	20042	Pending
5	20043	Pending

Total rows: 1000 of 379768 Query complete 00:00:00.508

рисунок 4.5 Результат выполнения запроса с условным индексом

Запрос выполнился всего за полсекунды. Теперь для получения результирующей выборки используется сканирование по индексу



1. → Index Scan using stutus_cond_indx on orders as orders (cost=0.42..16873.6 rows=381834 width=13)

рисунок 4.6 План запроса с условным индексом

Из плана видно, что стоимость получения первой строки вновь составляет 0, но и стоимость получения последней еще больше уменьшилась - 16873.

Создадим вновь индекс по столбцу order_status, выполним запрос и посмотрим, какой план выберет оптимизатор.

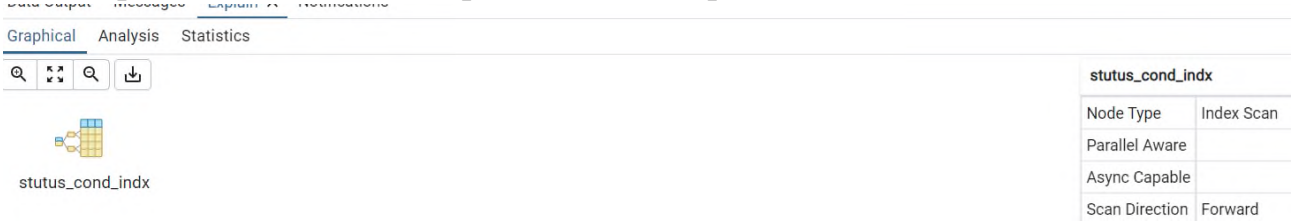


рисунок 4.7 План запроса при созданных двух индексах

Планировщик, действительно, выбрал оптимальный план с использованием индекса по первичному ключу.

4.2 Использование алгоритмов join

Выполним запрос из рисунка 1.3 в СУБД PostgreSQL без предварительных приготовлений и рассмотрим его план.

Для таблиц, имеющих следующую структуру (с данными таблиц можно ознакомится в приложении А):

Таблица "users"

Поле	Тип данных	Ограничения
id	UUID	PRIMARY KEY
first_name	VARCHAR(100)	
last_name	VARCHAR(100)	NOT NULL
username	VARCHAR(100)	NOT NULL, UNIQUE
email	VARCHAR(100)	NOT NULL, UNIQUE
password	VARCHAR(1000)	NOT NULL
role	ENUM(RoleEnum)	NOT NULL

Таблица "groups"

Поле	Тип данных	Ограничения
id	UUID	PRIMARY KEY
number	INTEGER	NOT NULL
course	ENUM(CourseEnum)	NOT NULL

Таблица "user_group_roles"

Поле	Тип данных	Ограничения
id	INTEGER	PRIMARY KEY
user_id	UUID	FOREIGN KEY → users.id
group_id	UUID	FOREIGN KEY → groups.id
role	ENUM(GroupRoleEnum)	NOT NULL

Таблица "events"

Поле	Тип данных	Ограничения
id	UUID	PRIMARY KEY
professor	VARCHAR(30)	
name	VARCHAR(50)	NOT NULL
place	VARCHAR(20)	
week	ENUM(WeekEnum)	NOT NULL
weekday	ENUM(WeekDayEnum)	NOT NULL
time	TIME (с таймзоной)	NOT NULL
date	DATETIME (с таймзоной)	NOT NULL

Таблица "attendance"

Поле	Тип данных	Ограничения

id	UUID	PRIMARY KEY
user_id	UUID	FOREIGN KEY → users.id
event_id	UUID	FOREIGN KEY → events.id
attended	BOOLEAN	
promised	BOOLEAN	
important_skip	BOOLEAN	

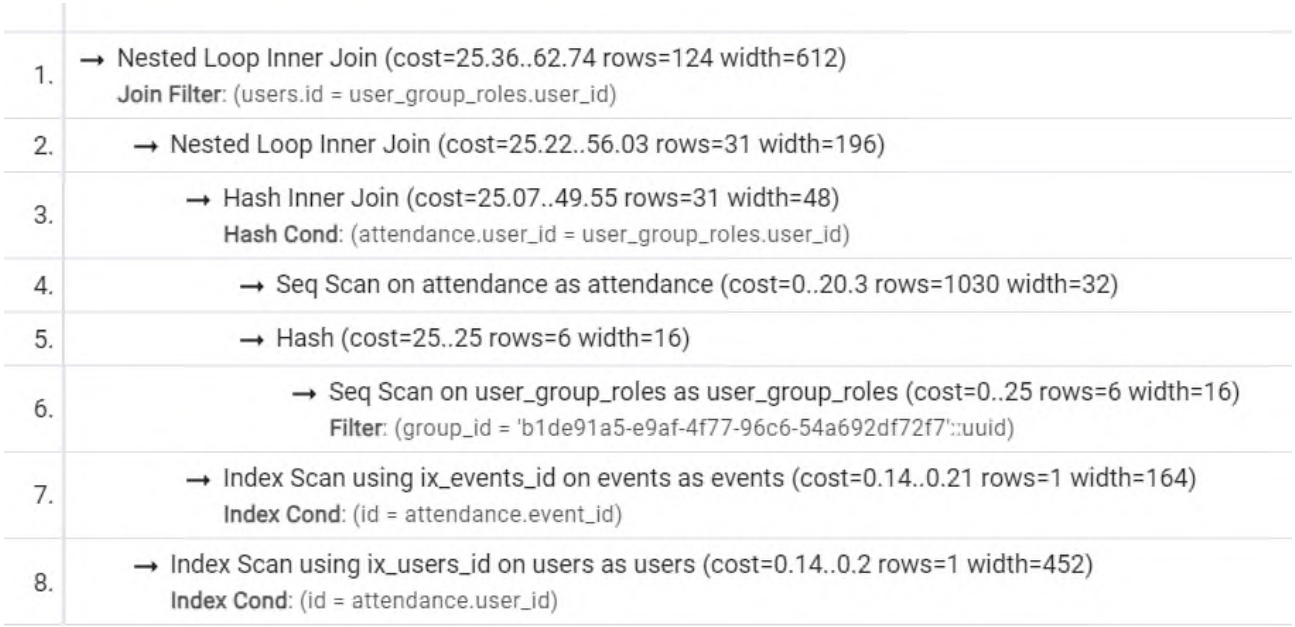


рисунок 4.8 План запроса с включенными алгоритмами хеширования и слияния

Заметно, что при наличии выбора из всех трех алгоритмов оптимизатор выбирает как алгоритм хеширования, так и вложенный цикл для двух операторов join.

Теперь отключим возможность использования алгоритма хеширования и посмотрим, как измениться план.

#	Node
1.	→ Nested Loop Inner Join (cost=97.21..115.61 rows=124 width=612) **Join Filter**: (users.id = user_group_roles.user_id)
2.	→ Nested Loop Inner Join (cost=97.07..108.89 rows=31 width=196)
3.	→ Merge Inner Join (cost=96.92..102.41 rows=31 width=48)
4.	→ Sort (cost=25.08..25.09 rows=6 width=16)
5.	→ Seq Scan on user_group_roles as user_group_roles (cost=0..25 rows=6 width=16) **Filter**: (group_id = 'b1de91a5-e9af-4f77-96c6-54a692df72f7'::uuid)
6.	→ Sort (cost=71.84..74.42 rows=1030 width=32)
7.	→ Seq Scan on attendance as attendance (cost=0..20.3 rows=1030 width=32)
8.	→ Index Scan using ix_events_id on events as events (cost=0.14..0.21 rows=1 width=164) **Index Cond**: (id = attendance.event_id)
9.	→ Index Scan using ix_users_id on users as users (cost=0.14..0.2 rows=1 width=452) **Index Cond**: (id = attendance.user_id)

рисунок 4.9 План запроса с отключенным алгоритмом хеширования

Из рисунка 4.9 видно, что там, где раньше использовался алгоритм хеширования теперь применятся алгоритм слияния с предшествующей сортировкой. Так же заметно, что и стоимость получения записей возросла значительно, как для первой, так и для последней записи.

Отключим еще и алгоритм слияния и оценим результаты.

#	Node
1.	→ Nested Loop Inner Join (cost=0.29..151.21 rows=124 width=612) **Join Filter**: (users.id = user_group_roles.user_id)
2.	→ Nested Loop Inner Join (cost=0.14..144.5 rows=31 width=196)
3.	→ Nested Loop Inner Join (cost=0..138.01 rows=31 width=48) **Join Filter**: (attendance.user_id = user_group_roles.user_id)
4.	→ Seq Scan on attendance as attendance (cost=0..20.3 rows=1030 width=32)
5.	→ Materialize (cost=0..25.03 rows=6 width=16)
6.	→ Seq Scan on user_group_roles as user_group_roles (cost=0..25 rows=6 width=16) **Filter**: (group_id = 'b1de91a5-e9af-4f77-96c6-54a692df72f7'::uuid)
7.	→ Index Scan using ix_events_id on events as events (cost=0.14..0.21 rows=1 width=164) **Index Cond**: (id = attendance.event_id)
8.	→ Index Scan using ix_users_id on users as users (cost=0.14..0.2 rows=1 width=452) **Index Cond**: (id = attendance.user_id)

рисунок 4.10 План запроса с выключенными алгоритмами хеширования и слияния

Действительно, теперь все соединения происходят с помощью вложенных циклов. Отсюда следует, что стоимость получения первой строки - 0, однако получение всех записей в таком случае является наиболее затратным.

Таким образом оптимизатор изначально выбрал наиболее эффективный план.

В Oracle похожего поведения можно добиться используя хинты(см рисунок 4.11) используя /*+ USE_HASH */ /*+ USE_NL */ /*+ USE_MERGE */ для соединения по хешу, вложенными циклами, слиянием соответственно.

```
SELECT /*+ USE_HASH(a usrs) */ e.id, e.professor, e.place, e.time, u.first_name, u.last_name
FROM events e
        JOIN attendance a ON e.id = a.event_id
        JOIN (SELECT /*+ FULL(u) */ u.id, u.first_name, u.last_name
              FROM users u
                      JOIN user_group_roles ugr ON u.id = ugr.user_id
              WHERE ugr.group_id = 'b1de91a5-e9af-4f77-96c6-54a692df72f7') usrs ON usrs.id = a.user_id;
```

рисунок 4.11 Запрос с применением хинта в Oracle

Стоит отметить, что, хотя и существует возможность выполнить команду отключения алгоритма вложенных циклов, на самом деле полостью отключить его не получиться, потому что как было описано выше только этот алгоритм может работать с условиями типа равенства, и выполнение этой команды может лишь подсказать оптимизатору не использовать данный алгоритм, однако последнее слово остается за планировщиком.

ЗАКЛЮЧЕНИЕ

Исходя из проведенного анализа и проделанных экспериментов, следует, что создание подходящих индексов и выбор правильного алгоритма соединения таблиц являются ключевыми навыками для написания оптимального SQL-запроса. Для определения оптимальности плана СУБД представляют возможность анализа, построенных оптимизатором, планов. При этом заметны различия в подходах ключевых групп к данному вопросу. Так разработчики PostgreSQL делают ставку на максимальную оптимизацию планировщика запросов, отбирая у пользователей возможность вмешиваться в процесс выбора плана. Oracle, напротив, предоставляет данную возможность с помощью механизма хинтов и в целом более широким возможностям ручной настройки поведения базы данных.

Индексы – структуры, позволяющие ускорить процесс извлечения данных таблицы, являются мощным инструментом для ускорения выполнения запроса. Грамотно построенный индекс позволяет уменьшить время выполнения запроса в несколько раз. Кроме того, PostreSQL предлагает широкий выбор специализированных индексов, которые наилучшем образом подходят для нестандартных данных, таких как геоданные, JSON-объекты, полносвязанный текст, в этом аспекте Oracle уступает. Однако создание индексов может иметь и обратный эффект, так для таблиц, к которым часто применяются операции вставки или обновления данных, наличие индекса лишь увеличит время выполнения запроса, так как будут затрачены ресурсы на его обновление.

В обеих рассмотренных СУБД подходы к организации слияния таблиц не отличаются и представлены тремя видами алгоритмов: соединение вложенными циклами, соединение по хешу и сортировки слиянием. Каждый из алгоритмов служат своей цели, так даже самый просто и на первый взгляд неоптимальный алгоритм соединения вложенными циклами может быть намного эффективнее для небольших таблиц, особенно при наличии подходящего индекса.

В рамках практической части были подготовлены примеры, демонстрирующие влияние рассмотренных техник на скорость выполнения запроса: создание индекса, причем продемонстрирована разница между покрывающим и непокрывающим индексами и выбор оптимального алгоритма join, причем так же была продемонстрирована точность выбора плана оптимизатором запросов.

Таким образом при разработке высоконагруженных информационных систем ключевым фактором к их производительности является способность разработчика спроектировать базу данных и написать оптимальные SQL-запросы. В рамках данной дипломной работы предложены основные подходы к оптимизации запросов. Однако, следует помнить, что для каждой ситуации стратегия будет отличаться, поэтому решения требуется принимать на основе требований, предъявляемых к системе. Тоже касается и выбора СУБД, в качестве общей рекомендации можно предложить использовать PostgreSQL для более простых задач, полагаясь на оптимальные настройки СУБД со стороны

разработчика. Если же требуется более тонкая настройка, то Oracle будет лучшем выбором.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Алгазали С.М.М., Айвазов В.Г., Кузнецова А.В., Совершенствование процесса поиска неэффективных SQL-запросов в СУБД Oracle // Инженерный вестник Дона. - 2017. № 4.
2. Домбровская Г., Новиков Б., Бейликова А., Оптимизация запросов в PostgreSQL - СПб.: ДМК Пресс, 2022 г. - 278 с.
3. Исаченко А. Н., Бондаренко С. П., Модели данных и системы управления базами данных - Минск: БГУ, 2007 г. - 220с.
4. Казаков А., GIN индексы в PostgreSQL [Электронный ресурс] – https://medium.com/@kazakov.andr.v/gin-postgresql
5. Каримов Т.Ш., Оптимизация запросов в реляционных базах данных: подходы и инструменты // Вестник науки, 2025 г. №4, том 3 – С. 780 – 785
6. Льюис Дж., Oracle. Основы стоимостной оптимизации - СПб.: Питер, 2006г. - 528с.
7. Морозов С. В., Нестеров С. А., Сравнительный анализ типов индексов в субд Sql Server и Postgresql // Системный анализ в проектировании и управлении, 2024 г. №2, том 27 – С. 485 – 491.
8. Подсказки (Oracle Hints) [Электронный ресурс]. - https://iusoltsev.wordpress.com/profile/individual-sql-and-cbo/cbo-hints
9. Сушков В., Индексы в PostgreSQL [Электронный ресурс] - https://tproger.ru/articles/indeksy-v-postgresql
10. Унковская Г.А., Анализ и выбор режима оптимизатора для получения оптимального плана выполнения запроса в СУБД ORACLE // Computational nanotechnology 2024. № 3, том 10 - С. 92 – 100.
11. PostgreSQL: Документация [Электронный ресурс]. - https://postgrespro.ru/docs/postgresql
12. PostgreSQL official Documentation [Электронный ресурс]. - https://www.postgresql.org/docs
13. Oracle official Documentation [Электронный ресурс]. - https://docs.oracle.com/en/database/oracle/oracle-database

ПРИЛОЖЕНИЕ А

ФРАГМЕНТЫ ТАБЛИЦ, ИСПОЛЬЗУЮЩИХСЯ ДЛЯ ДЕМОСТРАЦИИ АЛГОРИТМОВ JOIN

Таблица attendance

	id [PK] uuid	user_id uuid	event_id uuid	attended boolean	promised boolean	important_skip boolean
1	8a8b17e9-2fc3-4200-a04f-9ac7f68b8421	a1f2133a-e1fd-41e9-9003-52a7d1d4f1...	6d7b3f52-1c7d-4b2b-90d0-9e4fef8fae...	true	true	false
2	fb64792b-f5d6-4f69-9aef-89369a8e72d7	bb2f6f6b-fd39-4020-8d7c-97b9b5129d...	f273ad30-8c79-4623-9b83-fb63ff2893...	true	false	false
3	7ec1c4b3-0f2e-4f00-80c6-f6d4cf7a71ad	be50d293-c1ed-45bc-857d-243fc49a5...	bd7e7ce4-c0e0-4b17-8447-cf45b1885...	false	true	true
4	9a42124b-9a86-4932-921e-27c779e9a...	df31ef17-11f7-4890-b826-0d1178017a...	0ab6f1bc-274e-42a4-bb99-1f01a6f2f7...	true	true	false
5	b4c2636d-f95e-437f-82be-6eac0983cbfb	cfcc5604-b316-4b82-82ce-1977e0e86...	4d73e7e7-6fc6-4b6a-a8c0-c25dcbf5e5...	false	false	false
6	e68f1f5e-8398-4623-86a4-3d319991e9...	dff5b7c1-ec67-45fc-8df8-2b5a173bb5b3	c9f46a43-b7c4-4dbb-a7db-8c6ef1d5fd...	true	true	true
7	3a429835-7bc3-40dc-81a0-b0a45251a...	eaf11461-c6f2-4470-89e8-e3e9ef25aa...	dfe10cc3-e6f6-44a7-b80c-7d42a91f54...	true	false	false
8	043ee390-d9b1-4681-8a49-f27d6ac42f...	fb12b470-9bb0-41a3-8799-d2c33c0cb...	889f37b2-567a-4c56-b9cd-38e7c127c...	false	true	false
9	fac79cd5-b104-4b3c-b166-4f0fd32fe469	af832b63-bbdf-4e1f-8f75-17cf72975b1c	3c2ed3bc-14df-4c2f-a567-f89b4f1c4b...	false	false	true

Таблица events

	id [PK] uuid	professor character varying (30)	name character varying (50)	place character varying (20)	week character varying (10)	weekday character varying (10)	time time without time zone	date timestamp with time zon
1	6d7b3f52-1c7d-4b2b-90d0-9e4fef8fae...	Петров И.И.	Математика	Ауд. 101	ЧТ	ПН	09:00:00	2024-09-02 09:00:00+00
2	f273ad30-8c79-4623-9b83-fb63ff2893...	Сидоров С.С.	Физика	Ауд. 102	НЕЧ	ВТ	11:00:00	2024-09-03 11:00:00+00
3	bd7e7ce4-c0e0-4b17-8447-cf45b1885...	Кузнецова А.А.	Химия	Ауд. 103	ЧТ	СР	13:00:00	2024-09-04 13:00:00+00
4	0ab6f1bc-274e-42a4-bb99-1f01a6f2f7...	Иванов В.В.	История	Ауд. 104	НЕЧ	ЧТ	15:00:00	2024-09-05 15:00:00+00
5	4d73e7e7-6fc6-4b6a-a8c0-c25dcbf5e5...	Смирнов Н.Н.	Биология	Ауд. 105	ЧТ	ПТ	10:00:00	2024-09-06 10:00:00+00
6	c9f46a43-b7c4-4dbb-a7db-8c6ef1d5fd...	Морозова Е.Е.	Информатика	Ауд. 106	НЕЧ	ПН	14:00:00	2024-09-09 14:00:00+00
7	dfe10cc3-e6f6-44a7-b80c-7d42a91f54...	Гусев Р.Р.	География	Ауд. 107	ЧТ	ВТ	16:00:00	2024-09-10 16:00:00+00
8	889f37b2-567a-4c56-b9cd-38e7c127c...	Тарасов Ю.Ю.	Литература	Ауд. 108	НЕЧ	СР	08:00:00	2024-09-11 08:00:00+00
9	3c2ed3bc-14df-4c2f-a567-f89b4f1c4b...	Белов А.А.	Астрономия	Ауд. 109	ЧТ	ЧТ	12:00:00	2024-09-12 12:00:00+00
10	3cf442ce-8f19-4638-a707-cae761e8fafd	Зайцева И.И.	Обществознание	Ауд. 110	НЕЧ	ПТ	10:00:00	2024-09-13 10:00:00+00

Таблица groups

	id [PK] uuid	number integer	course character varying (20)
1	1f8c7f5b-27e2-4a9c-891f-98a8020a9fc7	101	math
2	2f8c7f5b-27e2-4a9c-891f-98a8020a9fc8	102	physics
3	3f8c7f5b-27e2-4a9c-891f-98a8020a9fc9	103	cs
4	4f8c7f5b-27e2-4a9c-891f-98a8020a9fc1	104	bio
5	5f8c7f5b-27e2-4a9c-891f-98a8020a9fc2	105	math
6	6f8c7f5b-27e2-4a9c-891f-98a8020a9fc3	106	cs
7	7f8c7f5b-27e2-4a9c-891f-98a8020a9fc4	107	physics
8	8f8c7f5b-27e2-4a9c-891f-98a8020a9fc5	108	bio
9	9f8c7f5b-27e2-4a9c-891f-98a8020a9fc6	109	cs

Таблица user_group_roles

	id [PK] integer	user_id uuid	group_id uuid	role character varying (20)
1	54	a1f2133a-e1fd-41e9-9003-52a7d1d4f1...	1f8c7f5b-27e2-4a9c-891f-98a8020a9fc7	student
2	55	bb2f6f6b-fd39-4020-8d7c-97b9b5129d...	1f8c7f5b-27e2-4a9c-891f-98a8020a9fc7	student
3	56	be50d293-c1ed-45bc-857d-243fc49a5...	2f8c7f5b-27e2-4a9c-891f-98a8020a9fc8	teacher
4	57	df31ef17-11f7-4890-b826-0d1178017a...	3f8c7f5b-27e2-4a9c-891f-98a8020a9fc9	student
5	58	cfcc5604-b316-4b82-82ce-1977e0e86...	4f8c7f5b-27e2-4a9c-891f-98a8020a9fc1	teacher
6	59	dff5b7c1-ec67-45fc-8df8-2b5a173bb5b3	5f8c7f5b-27e2-4a9c-891f-98a8020a9fc2	student
7	60	eaf11461-c6f2-4470-89e8-e3e9ef25aa...	6f8c7f5b-27e2-4a9c-891f-98a8020a9fc3	student
8	61	fb12b470-9bb0-41a3-8799-d2c33c0cb...	7f8c7f5b-27e2-4a9c-891f-98a8020a9fc4	admin
9	62	af832b63-bbdf-4e1f-8f75-17cf72975b1c	8f8c7f5b-27e2-4a9c-891f-98a8020a9fc5	student

Таблица users

	id [PK] uuid	first_name character varying (100)	last_name character varying (100)	username character varying (100)	email character varying (100)	password character varying (1000)	role character varying (20)
1	a1f2133a-e1fd-41e9-9003-52a7d1d4f1...	Иван	Иванов	ivanov	ivanov@example.com	hashedpwd1	student
2	bb2f6f6b-fd39-4020-8d7c-97b9b5129d...	Ольга	Смирнова	smirnova	smirnova@example.com	hashedpwd2	student
3	be50d293-c1ed-45bc-857d-243fc49a5...	Пётр	Петров	petrov	petrov@example.com	hashedpwd3	teacher
4	df31ef17-11f7-4890-b826-0d1178017a...	Анна	Кузнецова	kuznecova	kuznecova@example.com	hashedpwd4	student
5	cfcc5604-b316-4b82-82ce-1977e0e86...	Сергей	Сидоров	sidorov	sidorov@example.com	hashedpwd5	teacher
6	dff5b7c1-ec67-45fc-8df8-2b5a173bb5b3	Татьяна	Фролова	frolova	frolova@example.com	hashedpwd6	student
7	eaf11461-c6f2-4470-89e8-e3e9ef25aa...	Николай	Орлов	orlov	orlov@example.com	hashedpwd7	student
8	fb12b470-9bb0-41a3-8799-d2c33c0cb...	Мария	Соколова	sokolova	sokolova@example.com	hashedpwd8	admin
9	af832b63-bbdf-4e1f-8f75-17cf72975b1c	Людмила	Морозова	morozova	morozova@example.com	hashedpwd9	student
10	b6e28eaf-1688-4f23-b85c-2f292c7f9a20	Алексей	Воробьёв	vorobiev	vorobiev@example.com	hashedpwd10	student

ПРИЛОЖЕНИЕ Б

ФРАГМЕНТ ТАБЛИЦЫ, ИСПОЛЬЗОВАВШЕЙСЯ ДЛЯ ДЕМОСТРАЦИИ СОЗДАНИЯ ИНДЕКСОВ

Таблица orders

	order_id [PK] uuid	customer_name character varying (100)	order_date timestamp with time zone	order_status character varying (20)	total_amount numeric (10,2)
1	bffbd0f8-c8cf-4b38-958a-8f5fadc7f201	Алексей Иванов	2024-05-01 07:30:00+00	Pending	1590.00
2	4e4cb1f2-5d29-44fc-b1a2-449ccbd918...	Мария Смирнова	2024-05-02 11:15:00+00	Completed	2290.50
3	7d2360f5-61b3-4e5f-90b7-e368ab3998...	Иван Петров	2024-05-03 06:45:00+00	Pending	1200.00
4	3377b839-8261-4292-a18c-72f5a80dc...	Ольга Сидорова	2024-05-04 10:05:00+00	Cancelled	890.00
5	ec84b26f-2846-4049-bc46-4ae4e8016e...	Сергей Кузнецов	2024-05-05 08:20:00+00	Pending	3100.75
6	9f1870d9-1d86-429f-88bb-6fa62dbfdcde	Елена Орлова	2024-05-06 14:10:00+00	Shipped	1975.20
7	3a6ebfd1-5bd7-4ce4-80be-bdcf3cf0a76d	Павел Морозов	2024-05-07 12:40:00+00	Completed	2840.99
8	83c2e9c9-9609-4b04-b85b-34d8c11c4...	Анна Белова	2024-05-08 05:55:00+00	Pending	1075.00
9	d83f4901-0131-4fa9-9d82-9205e7e8b5...	Дмитрий Волков	2024-05-09 09:25:00+00	Processing	1640.45