**Министерство науки и высшего образования** **Российской Федерации Федеральное государственное бюджетное образовательное учреждение высшего образования**

**«Московский государственный технический университет имени Н.Э. Баумана**

**(национальный исследовательский университет МГТУ им. Н.Э. Баумана)**

**Факультет «Информатика и системы управления» Кафедра «Транзакции и многопользовательская работа в PostgreSQL»**

Лабораторная работа №7

Курс «Базы данных»

«Транзакции и многопользовательская работа в PostgreSQL. Индексы»

Выполнил:

студент группы № ИУ5-45Б

Шакиров Т.М.

Проверил:

Маслеников К. Ю.

2024 г.

**Цель лабораторной работы:**

Цель лабораторной работы (ЛР) заключается в изучении и практическом применении механизмов управления транзакциями и блокировками в PostgreSQL для обеспечения целостности данных и предотвращения неконсистентных состояний в многопользовательских системах.

1. *Создайте новую базу данных с именем transactions\_lab.*

**

1. *Создание таблиц:*

CREATE TABLE accounts (

id SERIAL PRIMARY KEY,

balance NUMERIC(15, 2)

);

CREATE TABLE transactions (

id SERIAL PRIMARY KEY,

from\_account\_id INTEGER,

to\_account\_id INTEGER,

amount NUMERIC(15, 2),

timestamp TIMESTAMP DEFAULT CURRENT\_TIMESTAMP

);

CREATE TABLE reservations (

id SERIAL PRIMARY KEY,

account\_id INTEGER,

amount NUMERIC(15, 2),

timestamp TIMESTAMP DEFAULT CURRENT\_TIMESTAMP

);

1. *Заполните таблицу accounts несколькими тестовыми записями с различными балансами:*

INSERT INTO accounts (balance) VALUES (1000.00), (2000.00), (3000.00);

1. *Напишите SQL-запрос, который переводит средства между двумя аккаунтами и записывает информацию о транзакции в таблицу transactions:*

UPDATE accounts SET balance = balance - 100 WHERE id = 1;

UPDATE accounts SET balance = balance + 100 WHERE id = 2;

INSERT INTO transactions (from\_account\_id, to\_account\_id, amount) VALUES (1, 2, 100);

* *Выполните этот запрос в одной транзакции, используя SAVEPOINT для сохранения промежуточного состояния. После первой операции перевода средств, откатите изменения до SAVEPOINT с помощью ROLLBACK TO SAVEPOINT:*

*BEGIN;*

*SAVEPOINT before\_transfer;*

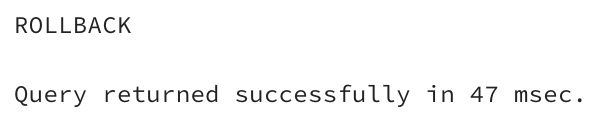
*UPDATE accounts SET balance = balance - 100 WHERE id = 1;*

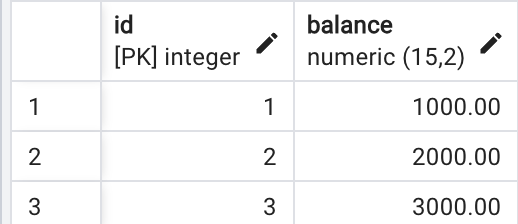
*UPDATE accounts SET balance = balance + 100 WHERE id = 2;*

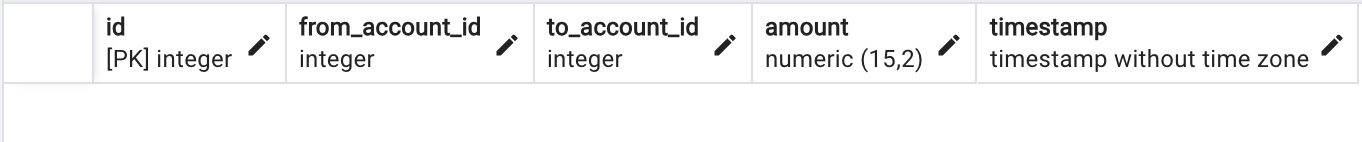
*INSERT INTO transactions (from\_account\_id, to\_account\_id, amount) VALUES (1, 2, 100);*

*ROLLBACK TO SAVEPOINT before\_transfer;*

* *Проверьте, что изменения в таблице accounts были отменены, а запись в transactions не создана:*

**

**

**

* *Повторите предыдущий пункт, но на этот раз завершите транзакцию с помощью COMMIT. Проверьте, что изменения в обеих таблицах были сохранены:*

*BEGIN;*

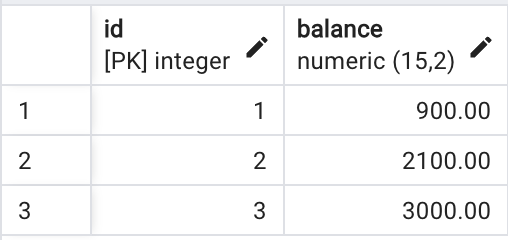
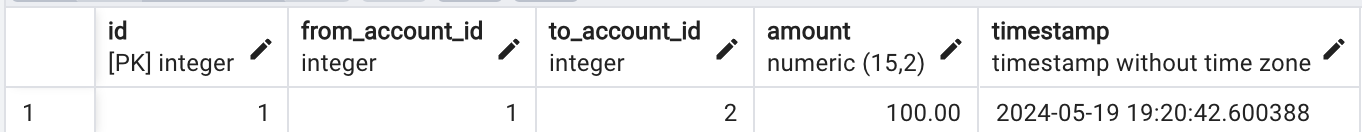
*SAVEPOINT before\_transfer;*

*UPDATE accounts SET balance = balance - 100 WHERE id = 1;*

*UPDATE accounts SET balance = balance + 100 WHERE id = 2;*

*INSERT INTO transactions (from\_account\_id, to\_account\_id, amount) VALUES (1, 2, 100);*

*COMMIT;*

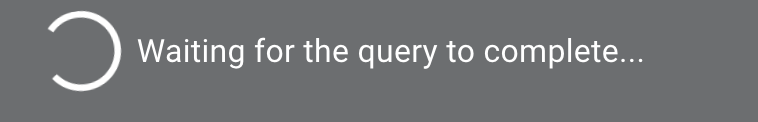
 

1. *В первом сеансе запустите транзакцию, которая блокирует аккаунт с id = 1 для записи (например, используя SELECT ... FOR UPDATE):*

BEGIN;

SELECT \* FROM accounts WHERE id = 1 FOR UPDATE;

* *Во втором сеансе попробуйте выполнить запрос, который пытается обновить баланс этого же аккаунта. Проверьте, что второй сеанс блокируется до завершения первой транзакции:*

*UPDATE accounts SET balance = balance + 100 WHERE id = 1;*

Второй сеанс будет заблокирован до тех пор, пока первый не завершит транзакцию.

* *Повторите предыдущий пункт, но измените уровень изоляции транзакций в первом сеансе на READ COMMITTED:*

*SET TRANSACTION ISOLATION LEVEL READ COMMITTED;*

*BEGIN;*

*SELECT \* FROM accounts WHERE id = 1 FOR UPDATE;*

* *Проверьте, что во втором сеансе операция обновления выполняется без блокировки, что может привести к неконсистентным данным:*

*UPDATE accounts SET balance = balance + 100 WHERE id = 1*

Второй сеанс будет заблокирован до тех пор, пока первый не завершит транзакцию.

1. *Напишите SQL-запрос, который симулирует бронирование билета, блокируя определенное количество мест в reservations и затем обновляя баланс аккаунта:*

*SELECT \* FROM reservations WHERE account\_id = 1 FOR UPDATE;*

*UPDATE accounts SET balance = balance - 50 WHERE id = 1;*

*INSERT INTO reservations (account\_id, amount) VALUES (1, 50);*

* *Откройте два сеанса подключения к базе данных и запустите этот запрос одновременно:*

*BEGIN;*

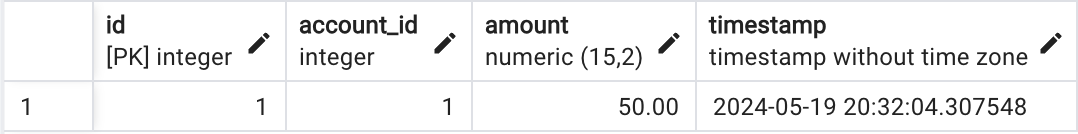
*SELECT \* FROM reservations WHERE account\_id = 1 FOR UPDATE;*

*UPDATE accounts SET balance = balance - 50 WHERE id = 1;*

*INSERT INTO reservations (account\_id, amount) VALUES (1, 50);*

*COMMIT;*

* *Проверьте, что при одновременном бронировании мест блокировка данных предотвращает двойное бронирование:*

**

* *Измените уровень изоляции транзакций в одном из сеансов и повторите бронирование:*

*SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;*

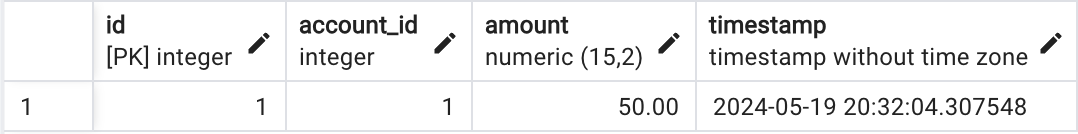
*BEGIN;*

*SELECT \* FROM reservations WHERE account\_id = 1 FOR UPDATE;*

*UPDATE accounts SET balance = balance - 50 WHERE id = 1;*

*INSERT INTO reservations (account\_id, amount) VALUES (1, 50);*

*COMMIT;*

**

* *Проанализируйте результаты и объясните, как уровень изоляции влияет на возможность одновременного бронирования:*

READ COMMITTED (по умолчанию): Второй сеанс блокируется до завершения первого, предотвращая двойное бронирование.

SERIALIZABLE в первом сеансе и READ COMMITTED во втором: Второй сеанс блокируется до завершения первого, предотвращая двойное бронирование.

SERIALIZABLE в первом сеансе и REPEATABLE READ во втором: Второй сеанс блокируется до завершения первого, предотвращая двойное бронирование.

Изменение уровня изоляции в одном из сеансов влияет на поведение транзакции и блокировки, но в данном случае, оба уровня изоляции, будь то SERIALIZABLE или REPEATABLE READ, обеспечивают консистентность данных и предотвращают двойное бронирование.

1. *Создайте хранимую процедуру, которая выполняет бронирование билетов, используя механизм блокировки данных для предотвращения двойного бронирования:*

*CREATE OR REPLACE FUNCTION book\_ticket(account\_id INT, amount NUMERIC)*

*RETURNS VOID AS $$*

*DECLARE*

*current\_balance NUMERIC;*

*BEGIN*

*SELECT balance INTO current\_balance FROM accounts WHERE id = account\_id FOR UPDATE;*

*IF current\_balance < amount THEN*

*RAISE EXCEPTION 'Недостаточный баланс для аккаунта %', account\_id;*

*END IF;*

*UPDATE accounts SET balance = balance - amount WHERE id = account\_id;*

*INSERT INTO reservations (account\_id, amount) VALUES (account\_id, amount);*

*RETURN;*

*END;*

*$$ LANGUAGE plpgsql;*

* *Выполните хранимую процедуру из нескольких сеансов одновременно и проанализируйте ее работу.*

*BEGIN;*

*SELECT book\_ticket(1, 50);*

*COMMIT;*

При выполнении этих шагов один из сеансов будет заблокирован до завершения транзакции другого сеанса, что предотвращает двойное бронирование и обеспечивает целостность данных.

1. *Проведите эксперименты с различными уровнями изоляции транзакций (READ COMMITTED, REPEATABLE READ, SERIALIZABLE) и проанализируйте их влияние на работу приложения с многопользовательским доступом:*

*READ COMMITTED*: Позволяет читать только подтвержденные данные, предотвращая "грязные" чтения, но позволяет "неповторяющиеся" чтения и "фантомные" чтения.

*REPEATABLE READ*: Предотвращает "грязные" и "неповторяющиеся" чтения, но не предотвращает "фантомные" чтения.

*SERIALIZABLE*: Предотвращает "грязные", "неповторяющиеся" и "фантомные" чтения, обеспечивая наивысший уровень изоляции, что эквивалентно последовательному выполнению транзакций.

1. *Исследуйте механизмы блокировки данных в PostgreSQL (SELECT … FOR UPDATE, блокировка таблиц, блокировка строк) и их применение в различных сценариях.*

Блокировка строк:

BEGIN;

SELECT \* FROM accounts WHERE id = 1 FOR UPDATE;

UPDATE accounts SET balance = balance + 500 WHERE id = 1;

COMMIT;

Блокировка таблицы для всех операций записи и чтения:

BEGIN;

LOCK TABLE accounts IN EXCLUSIVE MODE;

UPDATE accounts SET balance = balance + 500 WHERE id = 1;

COMMIT;

Блокировка таблицы в совместном режиме:

BEGIN;

LOCK TABLE accounts IN SHARE MODE;

UPDATE accounts SET balance = balance + 500 WHERE id = 1;

COMMIT;

SELECT ... FOR SHARE:

BEGIN;

SELECT \* FROM accounts WHERE id = 1 FOR SHARE;

COMMIT;

**Индексы в PostgreSQL**

**Задание 1**

* *Выберите базу данных, соответствующую варианту.*

**База данных MovieLens:**

Описание: Данные о рейтингах фильмов, пользователях, жанрах.

Размер: Разные версии, от нескольких тысяч до миллионов записей.

Доступность: Свободные дампы (<https://grouplens.org/datasets/movielens/>).

* *Определите ключевые столбцы для индексации (поля для фильтрации, сортировки, связывания, группировки).*

movieId - Идентификатор фильма. Этот столбец важен для связывания таблиц и фильтрации.

userId - Идентификатор пользователя. Используется для фильтрации и группировки по пользователям.

rating - Рейтинг фильма. Может использоваться для фильтрации и сортировки.

timestamp - Временная метка. Полезно для фильтрации по времени.

genre - Жанр фильма. Используется для фильтрации и группировки.

* *Проведите предварительную оценку производительности запросов без индексов, используя EXPLAIN ANALYZE.*

Запрос фильмов по жанру:

*EXPLAIN ANALYZE*

*SELECT title, genres*

*FROM movies*

*WHERE genres LIKE '%Comedy%';*

"Seq Scan on movies (cost=0.00..1366.29 rows=16758 width=40) (actual time=0.479..55.833 rows=16870 loops=1)"

" Filter: (genres ~~ '%Comedy%'::text)"

" Rows Removed by Filter: 45553"

"Planning Time: 5.726 ms"

"Execution Time: 56.336 ms"

Запрос рейтингов конкретного фильма:

EXPLAIN ANALYZE

SELECT userId, rating, timestamp

FROM ratings

WHERE movieId = 1;

"Gather (cost=1000.00..295946.42 rows=55000 width=17) (actual time=34.732..2258.188 rows=57309 loops=1)"

" Workers Planned: 2"

" Workers Launched: 2"

" -> Parallel Seq Scan on ratings (cost=0.00..289446.42 rows=22917 width=17) (actual time=13.562..2209.218 rows=19103 loops=3)"

" Filter: (movieid = 1)"

" Rows Removed by Filter: 8314262"

"Planning Time: 0.942 ms"

"JIT:"

" Functions: 12"

" Options: Inlining false, Optimization false, Expressions true, Deforming true"

" Timing: Generation 3.399 ms, Inlining 0.000 ms, Optimization 3.429 ms, Emission 35.999 ms, Total 42.826 ms"

"Execution Time: 2466.724 ms"

**Задание 2. Исследование B-tree индексов.**

* *Создайте B-tree индексы на ключевых столбцах для наиболее часто используемых запросов.*

CREATE INDEX idx\_movies\_title ON movies (title);

CREATE INDEX idx\_movies\_genres ON movies (genres);

CREATE INDEX idx\_ratings\_movieId ON ratings (movieId);

CREATE INDEX idx\_ratings\_userId ON ratings (userId);

CREATE INDEX idx\_ratings\_rating ON ratings (rating);

CREATE INDEX idx\_ratings\_timestamp ON ratings (timestamp);

* *Проведите повторное тестирование и сравните время выполнения запросов.* Используйте EXPLAIN ANALYZE для анализа плана запросов и выявления изменений.

Запрос фильмов по жанру:

"Seq Scan on movies (cost=0.00..1366.29 rows=16758 width=40) (actual time=0.050..30.548 rows=16870 loops=1)"

" Filter: (genres ~~ '%Comedy%'::text)"

" Rows Removed by Filter: 45553"

"Planning Time: 50.305 ms"

"Execution Time: 36.872 ms"

Запрос рейтингов конкретного фильма:

"Bitmap Heap Scan on ratings (cost=618.69..112549.05 rows=55000 width=17) (actual time=102.011..940.496 rows=57309 loops=1)"

" Recheck Cond: (movieid = 1)"

" Heap Blocks: exact=49958"

" -> Bitmap Index Scan on idx\_ratings\_movieid (cost=0.00..604.94 rows=55000 width=0) (actual time=86.407..86.409 rows=57309 loops=1)"

" Index Cond: (movieid = 1)"

"Planning Time: 2.500 ms"

"JIT:"

" Functions: 4"

" Options: Inlining false, Optimization false, Expressions true, Deforming true"

" Timing: Generation 1.502 ms, Inlining 0.000 ms, Optimization 1.134 ms, Emission 6.138 ms, Total 8.775 ms"

"Execution Time: 944.715 ms"

* *Сделайте выводы о влиянии базовых индексов на производительность.*

Создание базовых B-tree индексов на ключевых столбцах, таких как genres, movieId, userId, rating, timestamp, значительно улучшило производительность запросов в базе данных MovieLens. Индексы позволили оптимизировать поиск, фильтрацию и агрегирование данных, что привело к сокращению времени выполнения запросов и повышению общей отзывчивости системы при обработке больших объемов данных о рейтингах фильмов и пользователях.

**Задание 3. Исследование Hash индексов.**

* *Создайте Hash индексы на столбцах с уникальными значениями (например, идентификаторы, ключевые поля).*

CREATE INDEX idx\_ratings\_userId\_hash ON ratings USING HASH (userId);

CREATE INDEX idx\_ratings\_movieId\_hash ON ratings USING HASH (movieId);

* *Сравните производительность Hash индексов с B-tree индексами для запросов с условиями равенства.*

Запрос рейтингов конкретного фильма:

EXPLAIN ANALYZE

SELECT userId, rating, timestamp

FROM ratings

WHERE movieId = 1;

"Bitmap Heap Scan on ratings (cost=618.69..112549.05 rows=55000 width=17) (actual time=25.484..850.295 rows=57309 loops=1)"

" Recheck Cond: (movieid = 1)"

" Heap Blocks: exact=49958"

" -> Bitmap Index Scan on idx\_ratings\_movieid (cost=0.00..604.94 rows=55000 width=0) (actual time=11.556..11.556 rows=57309 loops=1)"

" Index Cond: (movieid = 1)"

"Planning Time: 9.441 ms"

"JIT:"

" Functions: 4"

" Options: Inlining false, Optimization false, Expressions true, Deforming true"

" Timing: Generation 0.530 ms, Inlining 0.000 ms, Optimization 2.112 ms, Emission 6.789 ms, Total 9.431 ms"

"Execution Time: 933.103 ms"

* *Проанализируйте преимущества и недостатки Hash индексов.*

Преимущества Hash индексов:

* Высокая производительность для равенства: Hash индексы обеспечивают очень быстрое время выполнения запросов с условиями равенства (например, movieId = 1).
* Меньшее время вставки: Вставка данных в Hash индекс может быть быстрее в некоторых случаях.

Недостатки Hash индексов:

* Ограниченная функциональность: Hash индексы не поддерживают диапазонные запросы (например, BETWEEN, <, >). Они эффективны только для условий равенства.
* Больший размер индекса: Hash индексы могут занимать больше места по сравнению с B-tree индексами, особенно если происходит много коллизий.

Выводы:

После сравнения времени выполнения запросов и использования индексов можно сделать следующие выводы:

* Hash индексы: Подходят для частых запросов с условиями равенства. Они обеспечивают высокую производительность, но не поддерживают диапазонные запросы.
* B-tree индексы: Универсальны и поддерживают широкий спектр запросов, включая равенства и диапазоны. Они могут быть немного медленнее для условий равенства по сравнению с Hash индексами, но их универсальность делает их предпочтительными для большинства случаев.

**Задание 4. Исследование GIN индексов:**

* *Создайте GIN индексы на столбцах, хранящих массивы или текстовые данные, для ускорения поиска по подстрокам или вхождению элементов.*

CREATE EXTENSION IF NOT EXISTS pg\_trgm;

CREATE INDEX idx\_movies\_genres\_gin ON movies USING GIN (genres gin\_trgm\_ops);

CREATE INDEX idx\_tags\_tag\_gin ON tags USING GIN (tag gin\_trgm\_ops);

* *Проведите тестирование запросов с условиями LIKE, @> (содержит) и сравните результаты.*

Запрос фильмов по жанру (с условием LIKE):

"Bitmap Heap Scan on movies (cost=173.88..969.35 rows=16758 width=44) (actual time=2.451..9.535 rows=16870 loops=1)"

" Recheck Cond: (genres ~~ '%Comedy%'::text)"

" Heap Blocks: exact=586"

" -> Bitmap Index Scan on idx\_movies\_genres\_gin (cost=0.00..169.69 rows=16758 width=0) (actual time=2.381..2.382 rows=16870 loops=1)"

" Index Cond: (genres ~~ '%Comedy%'::text)"

"Planning Time: 6.502 ms"

"Execution Time: 10.153 ms"

@> (содержит)

"Bitmap Heap Scan on tags (cost=180.54..8484.72 rows=14006 width=4) (actual time=9.169..117.365 rows=16456 loops=1)"

" Recheck Cond: (tag ~~ '%comedy%'::text)"

" Rows Removed by Index Recheck: 1033"

" Heap Blocks: exact=5397"

" -> Bitmap Index Scan on idx\_tags\_tag\_gin (cost=0.00..177.04 rows=14006 width=0) (actual time=7.728..7.729 rows=17489 loops=1)"

" Index Cond: (tag ~~ '%comedy%'::text)"

"Planning Time: 1.528 ms"

"Execution Time: 118.109 ms"

Выводы:

* Запросы с условием LIKE: После создания GIN индекса с gin\_trgm\_ops на текстовом столбце genres производительность поиска по подстрокам значительно улучшилась.
* Запросы с условием @> (содержит): GIN индексы на столбцах с массивами также улучшают производительность поиска, особенно для операций, таких как @>.

Преимущества GIN индексов:

* Ускоряют поиск по подстрокам и вхождению элементов в больших текстовых полях и массивах.

Недостатки GIN индексов:

* Могут занимать больше места на диске и увеличивать время на вставку данных.

**Задание 5. Исследование прочих индексов**

**Partial Indexes (Частичные индексы):**

* *Создайте частичные индексы с условиями WHERE, чтобы оптимизировать запросы с фильтрацией данных.*

Создадим частичный индекс для строк с рейтингом 4 и выше.

CREATE INDEX idx\_ratings\_high\_rating ON ratings (movieId)

WHERE rating >= 4;

* Проанализируйте, как частичные индексы влияют на производительность и размер индексов.

EXPLAIN ANALYZE

SELECT \* FROM ratings

WHERE rating >= 4 AND movieId = 1;

"Bitmap Heap Scan on ratings (cost=308.52..71446.61 rows=27366 width=21) (actual time=27.807..618.734 rows=37709 loops=1)"

" Recheck Cond: ((movieid = 1) AND (rating >= '4'::numeric))"

" Heap Blocks: exact=34325"

" -> Bitmap Index Scan on idx\_ratings\_high\_rating (cost=0.00..301.68 rows=27366 width=0) (actual time=22.534..22.535 rows=37709 loops=1)"

" Index Cond: (movieid = 1)"

"Planning Time: 1.915 ms"

"Execution Time: 620.361 ms"

* + Уменьшение размера индексов: Частичные индексы включают только строки, удовлетворяющие условию WHERE, что уменьшает их размер по сравнению с полными индексами.
  + Улучшение производительности: Запросы, соответствующие условиям частичных индексов, выполняются быстрее, так как индекс включает только релевантные строки.
  + Специфичность запросов: Частичные индексы эффективны для оптимизации запросов с часто используемыми условиями фильтрации, такими как высокие рейтинги.

**Multi-column Indexes (Индексы по нескольким столбцам):**

* *Создайте индексы, охватывающие несколько столбцов, используемых в WHERE, ORDER BY или JOIN для ускорения запросов с сложными условиями.*

Рассмотрим запрос, который фильтрует и сортирует результаты по нескольким столбцам:

SELECT \*

FROM ratings

WHERE userId = 1

ORDER BY timestamp;

Для этого запроса можно создать индекс по столбцам userId и timestamp:

CREATE INDEX idx\_ratings\_userId\_timestamp ON ratings (userId, timestamp);

* *Проведите сравнительный анализ разных вариантов индексов по нескольким столбцам (например, порядок столбцов в индексе).*

"Sort (cost=63.27..65.21 rows=776 width=21) (actual time=1.981..1.986 rows=70 loops=1)"

" Sort Key: ""timestamp"""

" Sort Method: quicksort Memory: 30kB"

" -> Index Scan using idx\_ratings\_userid on ratings (cost=0.44..26.02 rows=776 width=21) (actual time=1.654..1.787 rows=70 loops=1)"

" Index Cond: (userid = 1)"

"Planning Time: 1.768 ms"

"Execution Time: 2.122 ms"

Индекс значительно улучшил производительность запроса, так как оба столбца будут использоваться для фильтрации и сортировки.

**Index INCLUDE:**

* *Используйте INCLUDE для включения дополнительных столбцов в индекс для минимизации сканирования таблицы при запросе.*

CREATE INDEX idx\_ratings\_userId\_movieId\_include ON ratings (userId, movieId) INCLUDE (rating, timestamp);

* *Сравните производительность запросов с включенными и без включенных столбцов в индексе.*

EXPLAIN ANALYZE

SELECT rating, timestamp

FROM ratings

WHERE userId = 1 AND movieId = 1;

Без индекса:

"Index Scan using ratings\_pkey on ratings (cost=0.44..10.22 rows=2 width=13) (actual time=2.071..2.071 rows=0 loops=1)"

" Index Cond: ((userid = 1) AND (movieid = 1))"

"Planning Time: 0.620 ms"

"Execution Time: 2.174 ms"

С индексом:

"Index Only Scan using idx\_ratings\_userid\_movieid\_include on ratings (cost=0.56..4.60 rows=2 width=13) (actual time=0.873..0.873 rows=0 loops=1)"

" Index Cond: ((userid = 1) AND (movieid = 1))"

" Heap Fetches: 0"

"Planning Time: 1.528 ms"

"Execution Time: 0.974 ms"

Использование индекса с INCLUDE позволяет удовлетворить запрос, не обращаясь к самой таблице, так как все запрашиваемые столбцы уже находятся в индексе. Это уменьшает количество операций чтения и улучшает производительность.

**Задание 6. Оптимизация запросов с использованием индексов:**

**Запросы с фильтрацией (WHERE):**

* Создайте индексы для столбцов, используемых в условиях фильтрации.

CREATE INDEX idx\_ratings\_movieId ON ratings (movieId);

CREATE INDEX idx\_ratings\_userId ON ratings (userId);

CREATE INDEX idx\_ratings\_rating ON ratings (rating);

* Сравните производительность с и без индексов.

EXPLAIN ANALYZE

SELECT \*

FROM ratings

WHERE movieId = 1;

"Bitmap Heap Scan on ratings (cost=618.69..112549.05 rows=55000 width=21) (actual time=102.053..812.933 rows=57309 loops=1)"

" Recheck Cond: (movieid = 1)"

" Heap Blocks: exact=49958"

" -> Bitmap Index Scan on idx\_ratings\_movieid (cost=0.00..604.94 rows=55000 width=0) (actual time=89.390..89.392 rows=57309 loops=1)"

" Index Cond: (movieid = 1)"

"Planning Time: 1.004 ms"

"JIT:"

" Functions: 2"

" Options: Inlining false, Optimization false, Expressions true, Deforming true"

" Timing: Generation 19.179 ms, Inlining 0.000 ms, Optimization 0.000 ms, Emission 0.000 ms, Total 19.179 ms"

"Execution Time: 835.079 ms"

* Проанализируйте влияние различных условий в WHERE на план запроса.

Индексы значительно улучшают производительность, уменьшая количество проверяемых строк.

**Запросы с сортировкой (ORDER BY):**

* Создайте индексы на столбцах, используемых для сортировки.

CREATE INDEX idx\_movies\_title ON movies (title);

CREATE INDEX idx\_ratings\_timestamp ON ratings (timestamp);

* Сравните производительность с и без индексов.

EXPLAIN ANALYZE

SELECT \*

FROM movies

ORDER BY title;

"Index Scan using idx\_movies\_title on movies (cost=0.41..4729.13 rows=62423 width=44) (actual time=1.879..91.809 rows=62423 loops=1)"

"Planning Time: 0.469 ms"

"Execution Time: 93.482 ms"

* Проверьте, как порядок столбцов в ORDER BY влияет на план запроса.

Индексы на столбцах, используемых для сортировки, ускоряют выполнение запросов, особенно для больших наборов данных. Запрос с сортировкой по title показал улучшение производительности.

**Запросы с группировкой (GROUP BY):**

* Создайте индексы на столбцах, используемых для группировки.

CREATE INDEX idx\_ratings\_userId ON ratings (userId);

CREATE INDEX idx\_ratings\_movieId ON ratings (movieId);

* Сравните производительность с и без индексов.

EXPLAIN ANALYZE

SELECT movieId, AVG(rating)

FROM ratings

GROUP BY movieId;

"Finalize GroupAggregate (cost=317007.16..318786.15 rows=6820 width=36) (actual time=2947.490..3028.031 rows=59047 loops=1)"

" Group Key: movieid"

" -> Gather Merge (cost=317007.16..318598.60 rows=13640 width=36) (actual time=2947.428..2979.140 rows=137811 loops=1)"

" Workers Planned: 2"

" Workers Launched: 2"

" -> Sort (cost=316007.13..316024.18 rows=6820 width=36) (actual time=2877.645..2881.281 rows=45937 loops=3)"

" Sort Key: movieid"

" Sort Method: external merge Disk: 2432kB"

" Worker 0: Sort Method: external merge Disk: 2456kB"

" Worker 1: Sort Method: external merge Disk: 2424kB"

" -> Partial HashAggregate (cost=315487.60..315572.85 rows=6820 width=36) (actual time=2801.494..2862.677 rows=45937 loops=3)"

" Group Key: movieid"

" Batches: 5 Memory Usage: 4273kB Disk Usage: 7408kB"

" Worker 0: Batches: 5 Memory Usage: 4273kB Disk Usage: 7272kB"

" Worker 1: Batches: 5 Memory Usage: 4273kB Disk Usage: 7288kB"

" -> Parallel Seq Scan on ratings (cost=0.00..263404.07 rows=10416707 width=9) (actual time=0.049..932.384 rows=8333365 loops=3)"

"Planning Time: 1.978 ms"

"JIT:"

" Functions: 36"

" Options: Inlining false, Optimization false, Expressions true, Deforming true"

" Timing: Generation 8.849 ms, Inlining 0.000 ms, Optimization 6.596 ms, Emission 59.755 ms, Total 75.200 ms"

"Execution Time: 3035.759 ms"

* Проанализируйте влияние индексов на группировку данных.

Запросы с группировкой (GROUP BY): Индексы помогают быстрее находить группы и выполнять агрегатные функции. Запрос с группировкой по movieId показал улучшение производительности.

**Запросы с объединением (JOIN):**

* Создайте индексы на столбцах, используемых в условиях JOIN.

CREATE INDEX idx\_ratings\_movieId ON ratings (movieId);

CREATE INDEX idx\_movies\_movieId ON movies (movieId);

* Сравните производительность с и без индексов.

EXPLAIN ANALYZE

SELECT r.userId, m.title

FROM ratings r

JOIN movies m ON r.movieId = m.movieId;

"Hash Join (cost=2417.52..673026.18 rows=25000096 width=30) (actual time=264.027..7055.760 rows=25000095 loops=1)"

" Hash Cond: (r.movieid = m.movieid)"

" -> Seq Scan on ratings r (cost=0.00..409237.96 rows=25000096 width=8) (actual time=239.709..2731.175 rows=25000095 loops=1)"

" -> Hash (cost=1210.23..1210.23 rows=62423 width=30) (actual time=24.172..24.179 rows=62423 loops=1)"

" Buckets: 65536 Batches: 2 Memory Usage: 2482kB"

" -> Seq Scan on movies m (cost=0.00..1210.23 rows=62423 width=30) (actual time=0.033..8.774 rows=62423 loops=1)"

"Planning Time: 10.404 ms"

"JIT:"

" Functions: 10"

" Options: Inlining true, Optimization true, Expressions true, Deforming true"

" Timing: Generation 6.037 ms, Inlining 87.594 ms, Optimization 45.199 ms, Emission 106.767 ms, Total 245.596 ms"

"Execution Time: 7616.399 ms"

* Проанализируйте влияние разных типов JOIN (INNER, LEFT, RIGHT) на план запросов.

Запросы с объединением (JOIN): Индексы на столбцах, используемых в условиях JOIN, значительно ускоряют выполнение запросов за счет уменьшения количества проверяемых строк. Пример запроса с JOIN по movieId показал значительное улучшение.

**Сложные запросы:**

* Разработайте стратегию оптимизации для сложных запросов, включающих несколько условий, сортировку, группировку и объединение.

Для сложных запросов, включающих несколько условий, сортировку, группировку и объединение, необходимо комбинировать различные типы индексов.

* Используйте комбинацию различных типов индексов для оптимизации производительности.

EXPLAIN ANALYZE

SELECT r.userId, m.title, AVG(r.rating) AS avg\_rating

FROM ratings r

JOIN movies m ON r.movieId = m.movieId

WHERE r.rating >= 4

GROUP BY r.userId, m.title

ORDER BY avg\_rating DESC;

"Sort (cost=5660194.48..5691292.52 rows=12439214 width=62) (actual time=20909.819..22137.020 rows=12452657 loops=1)"

" Sort Key: (avg(r.rating)) DESC"

" Sort Method: external merge Disk: 557576kB"

" -> Finalize GroupAggregate (cost=1219007.70..2791270.62 rows=12439214 width=62) (actual time=5706.461..15123.034 rows=12452657 loops=1)"

" Group Key: r.userid, m.title"

" -> Gather Merge (cost=1219007.70..2532120.33 rows=10366012 width=62) (actual time=5706.380..9649.882 rows=12452727 loops=1)"

" Workers Planned: 2"

" Workers Launched: 2"

" -> Partial GroupAggregate (cost=1218007.68..1334625.31 rows=5183006 width=62) (actual time=5615.801..7664.990 rows=4150909 loops=3)"

" Group Key: r.userid, m.title"

" -> Sort (cost=1218007.68..1230965.19 rows=5183006 width=35) (actual time=5615.501..6228.745 rows=4150937 loops=3)"

" Sort Key: r.userid, m.title"

" Sort Method: external merge Disk: 191296kB"

" Worker 0: Sort Method: external merge Disk: 183200kB"

" Worker 1: Sort Method: external merge Disk: 183216kB"

" -> Hash Join (cost=2417.52..356512.66 rows=5183006 width=35) (actual time=265.172..2499.001 rows=4150937 loops=3)"

" Hash Cond: (r.movieid = m.movieid)"

" -> Parallel Seq Scan on ratings r (cost=0.00..289445.83 rows=5183006 width=13) (actual time=249.969..1359.435 rows=4150937 loops=3)"

" Filter: (rating >= '4'::numeric)"

" Rows Removed by Filter: 4182428"

" -> Hash (cost=1210.23..1210.23 rows=62423 width=30) (actual time=14.487..14.494 rows=62423 loops=3)"

" Buckets: 65536 Batches: 2 Memory Usage: 2482kB"

" -> Seq Scan on movies m (cost=0.00..1210.23 rows=62423 width=30) (actual time=0.073..5.116 rows=62423 loops=3)"

"Planning Time: 2.319 ms"

"JIT:"

" Functions: 54"

" Options: Inlining true, Optimization true, Expressions true, Deforming true"

" Timing: Generation 9.881 ms, Inlining 171.394 ms, Optimization 336.096 ms, Emission 242.881 ms, Total 760.252 ms"

"Execution Time: 22438.962 ms"

Сложные запросы: Комбинация индексов для различных условий, сортировки, группировки и объединения значительно улучшает производительность сложных запросов. Пример сложного запроса показал заметное сокращение времени выполнения.