Кафедра ЦТ
Институт информационных технологий
РТУ МИРЭА

Дисциплина «Разработка баз данных»

Практическая работа №7. Оптимизация запросов и управление транзакциями в PostgreSQL

Постановка задачи: основываясь на индивидуальной схеме данных, составьте необходимые запросы.

Задание №1: анализ и оптимизация (3 сценария)

Определить 3 медленных запроса к вашей БД, которые можно ускорить с помощью **разных** типов индексов.

При недостатке данных — предварительно сгенерируйте содержимое одной таблицы согласно примеру в методичке.

Для каждого из 3 сценариев выполните:

- **EXPLAIN ANALYZE** **без индекса** — получить фактический план.
- **Проанализировать план** «**до**» и указать, почему запрос медленный (например, Seq Scan).
- **Создать** подходящий **индекс**.
- Снова **выполнить** **EXPLAIN ANALYZE**.
- Привести **план «после»**, где виден Index Scan / Bitmap Index Scan.
- Подготовить **сравнительную таблицу** (до/после): тип плана, время выполнения.

(продолжение на следующем слайде)

Практическая работа №7. Оптимизация запросов и управление транзакциями в PostgreSQL

Постановка задачи: основываясь на индивидуальной схеме данных, составьте необходимые запросы.

Задание №2: демонстрация атомарной транзакции (COMMIT).

По примеру раздела 2.2 реализуйте в своей базе одну бизнес-операцию (минимум две связанные операции изменения данных) внутри транзакции BEGIN…COMMIT.

Задание №3: демонстрация отката транзакции (ROLLBACK).

Адаптировать приведённый в разделе 2.3 SQL-скрипт, моделирующий сбой операции, под свою предметную область, повторив описанные действия.

Задание №4: моделирование аномалии «Неповторяемое чтение».

Используя два редактора SQL, смоделировать проблему «неповторяемое чтение» на уровне изоляции по умолчанию (READ COMMITTED) по приведённому в разделе 2.4 образцу.

Задание №5: устранение аномалии «Неповторяемое чтение».

Повторить моделирование из Задания №4, но с использованием уровня изоляции REPEATABLE READ. В качестве образца использовать раздел 2.5.

(продолжение на предыдущем слайде)

ИНДЕКСЫ
(INDEXES)

Индексы (Indexes) – что такое индекс?

Индекс – это структура данных, созданная СУБД для быстрого поиска строк в таблице.

Он работает **как указатель**, позволяющий находить нужные значения **без обхода всей таблицы**.

Для этого он хранит **значение ключа** и **ссылку на строку**.

Индексы позволяют СУБД:

- ✓ **находить** данные **быстро**, а не просматривать всю таблицу;
- ✓ **выбирать строки** в **логарифмическом времени**, а не линейном;
- ✓ **ускорять** **JOIN**, **WHERE**, **ORDER BY** и **поиск** по диапазону;
- ✓ существенно **снижать нагрузку** на **диск** и **CPU** при **часто выполняемых запросах**.

ВАЖНО: индекс не дублирует всю таблицу, а хранит только ключи + ссылки в специальном, упорядоченном виде.

ВАЖНО: Индекс ускоряет чтение, но делает дороже вставку, обновление и удаление, потому что индекс тоже нужно поддерживать в актуальном состоянии.

Индексы (Indexes) – какие бывают индексы?

B-Tree – универсальный индекс, используется в большинстве случаев *(и создаётся **по умолчанию**)*.

- Подходит для условий **=**, **<**, **>**, **BETWEEN**, **сортировки**, поиска **по диапазонам**.
- Работает только если условия применяются к **исходному значению колонки**. `WHERE price > 100` ДА `price + 10 > 100` НЕТ

Индекс по выражению *(expression index)* – индексирует **результат функции**

- Используется в **WHERE** и применяется к функция – ***LOWER()***, ***date()***, ***substring()***, арифметическим и т.д.;
- Обычный **B-Tree** **не может** быть использован в таких случаях, потому что запрос сравнивает **вычисленное значение**, а **не исходное**.

Частичный индекс *(partial index)* – индексирует как **B-Tree**, но только **часть таблицы**.

- Пример: `WHERE is_active = TRUE`
- **Меньше** по размеру, **быстрее**, но работает **только** для запросов, **удовлетворяющих условию**.

Составной индекс *(multi-column)* – индекс **по нескольким колонкам** сразу.

- Важен **порядок полей**: идёт слева, т.е. индекс по полям (***a***, ***b***, ***c***) проверит колонку **a**, затем (**a, b**) и (**a, b, c**). Не сработает для (**b**) или (**c**).
- Полезен, если запросы фильтруют **одновременно по нескольким полям**.

Индексы (Indexes) – цена индексов...

Индекс **ускоряет чтение**, но может **удорожить изменения** данных при не правильном использовании.

- При **INSERT** ключи новой строки **добавляются** во все индексы.
- При **UPDATE** индексируемых полей **запись удаляется** из старого положения в индексе и **вставляется в новое**.
- При **DELETE** запись нужно **убрать из индекса**.

ВАЖНО: Чем больше индексов, тем медленнее массовые вставки и обновления, больше файлов на диске, больше работы для **ANALYZE**. Будьте внимательны при работе с ними!

Грамотная оптимизация – это **минимальный** набор индексов, которые дают ощутимую пользу для **реальных запросов**.

Индексы (Indexes) – ANALYZE & EXPLAIN

ANALYZE пересчитывает статистику по **таблицам** и **столбцам** для планировщика запросов.

Актуальная статистика влияет на выбор **Seq Scan** / **Index Scan**, порядок **JOIN**, **сортировку** и **агрегацию**.

Устаревшая статистика влечёт **ошибки плана** в **оценке числа строк**, и **запросы** «вдруг» начинают **работать медленнее**.

Обычно **ANALYZE** запускается автоматически, но после **массовой загрузки** или **сильного изменения данных** его полезно **вызвать вручную для ключевых таблиц**, а потом уже сравнивать планы и время выполнения.

EXPLAIN показывает, как PostgreSQL планирует выполнить запрос *(оценочные cost и rows)*, а **EXPLAIN ANALYZE** – как запрос действительно выполняется:

- Тип плана (***Index Scan*** – *использование индекса* / ***Seq Scan*** – *прямой перебор всех строк*);
- Точное количество строк **проверенное** на каждом шаге;
- Итоговый **Execution Time**.

ТРАНЗАКЦИИ (TRANSACTIONS)

Транзакции (Transactions) – что такое транзакции?

Транзакция – это **единый логический блок операций** над данными, который **выполняется целиком** или **не выполняется вовсе**.

Это механизм, который гарантирует:

- ✓ **все запросы** внутри **транзакции** рассматриваются как **одна неделимая операция**;
- ✓ **изменения становятся видимыми** для других только **после COMMIT**;
- ✓ при **ошибке или отмене** все **изменения откатываются** командой **ROLLBACK**.

Транзакция – основа свойств **ACID** (атомарность, согласованность, изолированность, долговечность).

Транзакции (Transactions) – ACID: что важно знать?

В контексте практического использования **достаточно помнить**:

- **A** (***Atomicity*** / ***Атомарность***) – либо **все** изменения транзакции видны после **COMMIT**, либо **ни одно** не сохраняется (при **ROLLBACK**).
- **C** (***Consistency*** / ***Согласованность***) – транзакция **не должна** оставлять БД в состоянии, **нарушающем** ограничения и бизнес-правила; **при нарушении завершение** должно быть с **ошибкой** и **откатом**.
- **I** (***Isolation*** / ***Изолированность***) – **параллельные транзакции не должны** случайно **портить результаты друг друга**; детально задаётся уровнем изоляции.
- **D** (***Durability*** / ***Долговечность***) – после успешного **COMMIT данные не теряются** при сбоях.

Транзакции (Transactions) – когда используются?

Транзакции обеспечивают корректность данных в ситуациях, когда **одна бизнес-операция** состоит **из нескольких отдельных** SQL-команд.

Транзакции применяются:

- ✓ при выполнении **нескольких связанных действий**;
- ✓ при любых **операциях**, **затрагивающих несколько таблиц**;
- ✓ при **денежных** и **финансовых действиях**: *переводы, платежи, списания*;
- ✓ при **сложных бизнес-операциях**, где **ошибка** в одном шаге **делает недействительными все остальные**;
- ✓ при **параллельной работе** пользователей, чтобы **исключить конфликты** и «**гонки**».

Транзакции – **обязательная часть** всех сценариев, где важно, чтобы данные были строго согласованными.

Транзакции (Transactions) – состояние прерванной транзакции

Прерванная транзакция – это состояние, в которое транзакция попадает **после любой ошибки**.

PostgreSQL **блокирует** выполнение **следующих команд**, чтобы не допустить **неконсистентного состояния данных**.

Причины перехода в aborted:

- ✓ **ошибка UNIQUE, FOREIGN KEY, CHECK**;
- ✓ попытка **изменить данные** в режиме **READ ONLY**;
- ✓ **ошибка функции/триггера**;
- ✓ любая другая **ошибка внутри транзакции**.

Что происходит в aborted-состоянии:

- ➢ **Любая новая команда** вызывает **ошибку**
  «`current transaction is aborted, commands ignored`»
- ➢ **Транзакцию нельзя подтвердить** (**COMMIT**).
- ➢ Ни одна **новая операция не выполняется**.

Единственно **верное действие** – выполнить `ROLLBACK;`, **отменив все действия транзакции** в базе данных.

Если нужна возможность «**частичных откатов**» –нужно **заранее использовать SAVEPOINT**.

Транзакции (Transactions) – аномалия «Неповторяемое чтение»

Аномалия «Неповторяемое чтение» ***(Non-Repeatable Read)*** возникает, когда **одна и та же транзакция** **дважды** **читает одни и те же** **строки** и **получает разные значения**, потому что **другая транзакция** **успела изменить данные** и зафиксировать их (**COMMIT**) между чтениями.

Что происходит в транзакции:

- ✓ **первый** **запрос** видит **старое значение** *(например,* ***цена 120****)*;
- ✓ другая транзакция **изменяет эти данные**;
- ✓ **второй** **запрос** видит **новое значение** *(например,* ***цена 500****)*;

Единственный **надёжный способ** **избежать этого** – использовать **более строгий** **уровень изоляции** **REPEATABLE READ** / **SERIALIZABLE**, где транзакция работает с **фиксированным «снимком» данных**, и повторные чтения **не увидят изменения** других транзакций до конца нашей.

Транзакции (Transactions) – уровни изоляции

READ COMMITTED *(уровень по умолчанию)* – каждая **команда** видит только **зафиксированные изменения на момент начала** конкретной **команды**. **Возможна аномалия** «**Неповторяемое чтение**».

REPEATABLE READ – транзакция работает **со «снимком» данных на момент первого чтения**. Повторные **SELECT** внутри транзакции **не увидят** новые **COMMIT** других транзакций *(**нет** «**Неповторяемого чтения**»)*.

SERIALIZABLE – самый строгий уровень, имитирует **последовательное выполнение транзакций** – при конфликте возможна ошибка сериализации, такую транзакцию нужно повторить. **Практически не используется**.

В PostgreSQL **уровень изоляции** можно **задавать** – для одной транзакции и для всей сессии.

- `SHOW transaction_isolation;`
  < **Показать** уровень **текущей изоляции**
- `SET SESSION CHARACTERISTICS AS TRANSACTION ISOLATION LEVEL READ COMMITTED;`
  < **Задать** уровень для **всей сессии**
-
```
BEGIN;
   SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;
   -- здесь ваши запросы
COMMIT;
```
  < **Задать** уровень только для **одной транзакции**

Транзакции (Transactions) – основные операторы

- **BEGIN** – **начинает транзакцию**. Все последующие операции выполняются как единый логический блок.
- **COMMIT** – **фиксирует изменения**. Делает их видимыми для других транзакций.
- **ROLLBACK** – **отменяет все изменения** с момента **BEGIN**. **Возвращает данные в исходное состояние**.
- **SAVEPOINT <имя>** – создаёт **точку отката** внутри транзакции. Позволяет откатывать только **часть операций**, а не все.
- **ROLLBACK TO SAVEPOINT <имя>** – **откатывает изменения** до указанного **SAVEPOINT**,оставляя транзакцию активной.
- **RELEASE SAVEPOINT <имя>** – удаляет **SAVEPOINT**. После этого откат к нему невозможен.
- **SET TRANSACTION** – устанавливает параметры текущей транзакции:
  - **уровень изоляции – ISOLATION LEVEL** [**READ COMMITTED** | REPEATABLE READ | SERIALIZABLE];
  - **режим доступа – READ WRITE** - разрешены любые операции **/ READ ONLY** – только чтение (**SELECT**).

Спасибо за внимание