

## PostgreSQL

### Транзакции и обработка данных в PostgreSQL

Транзакции являются основным механизмом обеспечения **целостности, согласованности и отказоустойчивости** данных в PostgreSQL. Транзакция — это логическая единица работы с базой данных, которая может включать одну или несколько операций, таких как вставка, обновление или удаление данных. Эти операции выполняются в рамках транзакции и должны удовлетворять строгим условиям для обеспечения надёжности системы.

Пример: перевод средств с одного банковского счёта на другой. Операция состоит как минимум из двух шагов — списание и зачисление. Если один из этих шагов не выполнен, целостность системы нарушается. Поэтому в БД такая операция должна быть оформлена как транзакция.

### Свойства транзакций (ACID)

Чтобы гарантировать надёжность и корректность работы с данными, транзакции в реляционных базах данных должны соблюдать четыре ключевых свойства, объединённых в аббревиатуру **ACID**:

#### 1. **Atomicity (атомарность):**

Транзакция — это минимально неделимая единица работы. Либо все изменения в рамках транзакции выполняются, либо ни одно. В случае ошибки или сбоя все промежуточные изменения отменяются, обеспечивая целостность данных.

**Пример:** при переводе денег между счетами, если средства списались с

одного счёта, но не зачислились на другой из-за сбоя, вся операция откатывается.

## 2. **Consistency (согласованность):**

После завершения транзакции база данных должна оставаться в согласованном состоянии. Это означает, что все ограничения и правила (внешние ключи, уникальность, контроль целостности и т.д.) продолжают соблюдаться.

**Пример:** при списании товара со склада количество не должно стать отрицательным. Транзакция, нарушающая это правило, будет отменена.

## 3. **Isolation (изолированность):**

Параллельно выполняющиеся транзакции не должны мешать друг другу.

Результат их выполнения должен быть таким же, как если бы они выполнялись последовательно, одна за другой.

Однако на практике полная изоляция может приводить к снижению производительности, поэтому в стандарте SQL предусмотрены уровни изоляции, позволяющие контролировать допустимые степени взаимодействия между транзакциями. Подробнее о возникающих при этом проблемах — в следующем разделе.

## 4. **Durability (надёжность):**

После фиксации транзакции её результаты сохраняются даже в случае сбоя оборудования, отключения питания или программной ошибки.

**Пример:** если после подтверждённого перевода средств выключается

сервер, данные не теряются и будут восстановлены при перезапуске системы.

### **Феномены параллельного выполнения транзакций**

Хотя ACID требует изоляции, на практике системы управления базами данных реализуют её с разной степенью строгости. Это приводит к различным аномалиям при параллельном выполнении транзакций. Эти аномалии объясняют, зачем нужны уровни изоляции, описанные в SQL-стандарте, и какие последствия возможны при их нарушении:

#### **1. Потерянное обновление (Lost Update)**

Если разные транзакции меняют одни и те же данные, то может возникнуть ситуация при которой изменения одной транзакции перезапишутся другой транзакцией. Например есть две транзакции. Они производят обновление одной и той же строки. Вторая транзакция изменила строку и зафиксировала свои изменения. После этого зафиксировала свои изменения первая транзакция и перезаписала изменения второй транзакции.

#### **Пример:**

[Рисунок1]

**Решение:** блокировка строк или уровни изоляции REPEATABLE READ и выше.

#### **2. Грязное чтение (Dirty Read)**

Первая транзакция изменила данные, но еще не зафиксировала свои изменения. В это время вторая транзакция читает измененные данные. Если первая транзакция сделает

отмену своих изменений - ROLLBACK, то получится что вторая транзакция работает с данными, которых нет в базе данных.

**Пример:**

[Рисунок2]

**Разрешается только** на уровне изоляции READ UNCOMMITTED, который PostgreSQL не поддерживает.

### **3. Неповторяющееся чтение (Non-repeatable Read)**

Первая транзакция читает строку и получает одни значения. После этого вторая транзакция меняет ту же строку, которая прочитала первая транзакция или удаляет эту строку и фиксирует свои изменения. Первая транзакция снова читает строку и получает уже другие данные, либо не получает данных, если вторая транзакция удалила строку.

[Рисунок3]

**Решение:** REPEATABLE READ и выше.

### **4. Фантомное чтение (Phantom Read)**

Этот феномен очень похож на неповторяющееся чтение, но здесь идет добавление одной или нескольких строк. Первая транзакция делает выборку набора строк. После этого приходит вторая транзакция и добавляет строки попадающие в эту выборку. Вторая транзакция фиксирует свои изменения. После этого первая транзакция снова делает ту же

самую выборку и получает другой набор строк, так как вторая транзакция добавила строки.

[Рисунок4]

**Решение:** SERIALIZABLE, или явные блокировки.

### **5. Аномалия сериализации (Serialization anomaly)**

Результат выполнения двух параллельных транзакции не совпадает ни с одним из результатов последовательного их выполнения. Если выполняются транзакции T1 и T2, то результат параллельного их выполнения не будет равен ни результату последовательного выполнения транзакции T1 затем T2 ни результату последовательного выполнения транзакции T2 затем T1.

[Рисунок5]

**Решается только при SERIALIZABLE.**

### **3. Уровни изоляции**

Чтобы контролировать допустимые взаимодействия параллельных транзакций и управлять феноменами, описанными выше, SQL и PostgreSQL определяют **уровни изоляции транзакций**. Каждый уровень задаёт допустимую степень «видимости» изменений, внесённых другими транзакциями:

1. **READ UNCOMMITTED:** согласно стандарту SQL, на этом уровне допускается грязное чтение — т.е. чтение данных, изменённых, но ещё не зафиксированных

другими транзакциями.

Однако в PostgreSQL грязное чтение невозможно, даже на этом уровне. Поэтому READ UNCOMMITTED фактически работает идентично уровню READ COMMITTED. Это означает, что данные, изменённые, но не зафиксированные другими транзакциями, никогда не будут видны.

2. **READ COMMITTED** (по умолчанию в PostgreSQL): при этом уровне каждая SQL-команда в пределах транзакции видит только те изменения, которые были зафиксированы до начала выполнения этой команды.

Это означает, что разные команды внутри одной транзакции могут видеть разное состояние данных. Возможны такие феномены, как **неповторяющееся чтение** и **фантомное чтение**, но **грязное чтение исключено**.

Благодаря MVCC в PostgreSQL чтение не блокирует запись, и наоборот — что делает этот уровень достаточно производительным при высокой конкуренции.

3. **REPEATABLE READ**: Все запросы внутри транзакции видят **одинаковое состояние данных**, зафиксированное в момент первого запроса. Это устраняет **неповторяющееся чтение** и **грязное чтение**.

В PostgreSQL этот уровень также предотвращает **фантомные чтения**, что достигается за счёт **снимка базы данных (snapshot isolation)**.

Транзакция не видит изменений, сделанных другими транзакциями после создания снимка. Если транзакция попытается изменить строки, которые были изменены и зафиксированы другой транзакцией после момента создания снимка, произойдёт конфликт записи, и операция завершится ошибкой.

4. **SERIALIZABLE**: Обеспечивает поведение, при котором результат выполнения параллельных транзакций эквивалентен их **последовательному выполнению**.

В PostgreSQL этот уровень реализован как **Serializable Snapshot Isolation (SSI)**, который позволяет добиться полной сериализуемости **без блокировок чтения**.

Система отслеживает возможные конфликты между транзакциями, и если сериализуемость нарушена, одна из транзакций откатывается. Это делает уровень строгим, но с сохранением высокой параллельности.

#### 4. MVCC (Многоверсионность)

В PostgreSQL параллельная работа с данными реализуется с помощью **MVCC (Multiversion Concurrency Control)** — многоверсионного контроля параллелизма. Это механизм, который позволяет транзакциям выполняться одновременно, без блокировок при чтении, и при этом сохранять корректность и изолированность выполнения.

##### Основные принципы работы MVCC

Вместо того чтобы блокировать строки, PostgreSQL создаёт новые версии строк при каждом их изменении. Старые версии сохраняются, чтобы они оставались доступными для других транзакций, которые начали работу до момента изменений. Таким образом, каждая транзакция видит «снимок» базы данных — её состояние на момент начала первого запроса.

Чтобы управлять версиями строк, PostgreSQL использует две системные колонки в каждой строке:

- **xmin** — идентификатор транзакции, которая создала эту версию строки;
- **xmax** — идентификатор транзакции, которая пометила строку как удалённую или заменённую.

Эти значения не видны пользователю, но PostgreSQL использует их для определения того, должна ли строка быть доступна в рамках конкретной транзакции. Например, если `xmin` больше идентификатора текущей транзакции, то эта строка была создана позже и ещё не должна быть видна. Если же `xmax` указывает на транзакцию, которая уже завершилась до начала текущей, и при этом строка была удалена, то она также считается невидимой.

Таким образом, каждая транзакция работает в логически изолированной среде, не видя незавершённых изменений других транзакций. Это обеспечивает изоляцию без использования блокировок на чтение.

Важно понимать, что строки, которые перестали быть видимыми для всех активных транзакций, остаются в таблице и продолжают занимать место. Для их удаления используется автоматический фоновый процесс — **autovacuum**. Он регулярно очищает таблицы от так называемых «мертвых» строк, восстанавливая дисковое пространство и предотвращая чрезмерный рост размера таблиц.

Эта многоверсионная архитектура позволяет PostgreSQL обеспечить чтение без блокировок. В отличие от систем, где чтение блокирует запись или наоборот, PostgreSQL позволяет транзакциям читать данные, даже если в это время они изменяются другими транзакциями. Конфликты выявляются лишь на момент фиксации изменений: если другая транзакция уже изменила те же данные и завершилась раньше, текущая транзакция получит ошибку сериализации и будет вынуждена откатиться.

Таким образом, MVCC в PostgreSQL не только реализует гибкие и безопасные уровни изоляции, но и делает возможной высокопроизводительную параллельную работу без избыточных блокировок. Это один из важнейших элементов архитектуры PostgreSQL,



обеспечивающий как корректность, так и масштабируемость в многопользовательской среде.

#### **Условная аналогия:**

Представьте, что в магазине нет очереди, потому что каждому вошедшему выдают **персональную копию магазина**. Каждый покупатель выбирает товары в своей версии и не мешает другим.

Однако при выходе все проходят через одного **кассира — менеджера транзакций**, который проверяет: можно ли применить эти действия в реальном магазине? Если какой-то товар уже "куплен" другой транзакцией, текущая операция будет отклонена.

Таким образом:

- **Первый покупатель**, который забрал бутылку молока, успешно завершает транзакцию — молоко списывается.
- **Второй покупатель**, выбравший ту же бутылку в своей изолированной копии, получает отказ при попытке завершения — молока в реальности уже нет.

### **5. Конфликты и блокировки при использовании MVCC**

Хотя MVCC (многоверсионность) позволяет PostgreSQL эффективно обрабатывать параллельные запросы без конфликтов при чтении, это не отменяет необходимости в блокировках — особенно когда дело касается изменений данных. PostgreSQL использует систему блокировок для защиты от конфликтов при одновременном доступе к одним и тем же данным со стороны разных транзакций.

**При чтении данных** обычным SELECT блокировки почти не применяются: каждая транзакция работает со своей "снимком" данных, видимым только ей. Это главное преимущество MVCC — чтения не блокируют записи и наоборот.

Однако **при изменении данных (INSERT, UPDATE, DELETE)** PostgreSQL применяет блокировки. Например, если одна транзакция изменила строку, другая не сможет изменить её до завершения первой. Это называется блокировкой на уровне строки. Такой подход позволяет избежать конфликтов, но в случае одновременного изменения одних и тех же строк одна из транзакций должна подождать.

Кроме строковых блокировок, PostgreSQL использует **блокировки на уровне таблицы**. Они применяются автоматически для защиты от одновременного выполнения операций, которые могли бы помешать друг другу — например, TRUNCATE или ALTER TABLE. Также можно явно указать нужный режим блокировки с помощью команды LOCK.

Существует несколько режимов таких блокировок, отличающихся строгостью и уровнем допуска к таблице. Например:

- простое чтение (SELECT) вызывает слабую блокировку, которая не мешает другим транзакциям;
- команды, изменяющие данные (UPDATE, DELETE), используют более строгие режимы;
- операции изменения структуры таблицы или очистки (как TRUNCATE, DROP TABLE) требуют самой строгой блокировки, исключающей любой другой доступ.

Если две транзакции пытаются получить несовместимые блокировки одновременно, одна из них будет ждать. Если же возникает **взаимоблокировка** (две транзакции ждут друг друга), PostgreSQL сам определяет проблему и завершает одну из транзакций с ошибкой. Это защищает систему от "зависаний".