КСиТ

7. Транзакции, ACID, уровни изоляции

Базы данных

Хисамутдинов М.А.
кафедра №12 НИЯУ МИФИ
2026

Контекст данных

Учебная схема «Баланс склада»:

- `warehouse(warehouse_id, code, city, name)`
- `product(product_id, sku, name, category, unit, base_price, discontinued)`
- `stock_balance(warehouse_id, product_id, qty_on_hand, qty_reserved, updated_at)`
- `stock_move(move_id, moved_at, warehouse_id, product_id, qty_delta, reason, doc_no)`

Контекст данных

Добавим еще пару таблиц для примеров.

- `stock_reservation(reservation_id, created_at, warehouse_id, product_id, qty, status, created_by)`
- `stock_balance_audit(audit_id, changed_at, op, warehouse_id, product_id, old_qty_on_hand, new_qty_on_hand, old_qty_reserved, new_qty_reserved, actor)`
- `category_tree(category_id, parent_id, name)`

Далее используется синтаксис PostgreSQL.

Зачем нужны транзакции

Представим, что есть задача перевода денег между счетами.

Нужно выполнить два действия: списать деньги с одного счета и зачислить на другой.

Что будет, если после первого запроса сервер упадёт?

Деньги списались, но не зачислились. База данных в противоречивом состоянии.

Транзакция — это способ сказать базе данных: “эти несколько операций — одно целое. Либо все выполнятся, либо ни одна.”

Транзакции

Транзакция — группа SQL-операций, которая выполняется как единое целое.

Транзакции существуют не только для защиты от сбоев, но и для защиты от параллельных пользователей. Пока одна транзакция работает с данными, другая не должна видеть частичные изменения.

PostgreSQL поддерживает транзакции и с самого начала проектировался как надёжная СУБД с полной поддержкой их.

Все DDL операции выполняются внутри транзакций и могут быть откачены.

Что же могут гарантировать транзакции?

ACID - свойства транзакций

В 1983 году Андреас Рейтер и Тео Хердер сформулировали четыре свойства, которые должна обеспечивать любая транзакционная система. По первым буквам они складываются в аббревиатуру ACID.

ACID - свойства транзакций

- `A` - Atomicity - Атомарность. Транзакция либо выполняется целиком, либо не выполняется вообще. Нет понятия «выполнилась наполовину».

Пример: перевод денег. Если что-то пошло не так — оба изменения откатываются.

- `C` - Consistency - Согласованность. Транзакция переводит базу из одного корректного состояния в другое корректное. Все ограничения, правила и инварианты должны соблюдаться до и после.

Пример: нельзя перевести деньги, если это нарушит ограничение balance >= 0.

ACID - свойства транзакций

- `I` - Isolation - Изолированность. Параллельные транзакции не видят промежуточных состояний друг друга. Каждая транзакция работает так, как будто она одна в системе.

Пример: пока мы списываем деньги и ещё не зачислили — никто другой не должен видеть «дыру» в балансе.

- `D` - Durability - Долговечность. После того как транзакция зафиксирована (COMMIT), данные сохранены навсегда — даже если сервер упадёт прямо сейчас.

Как создавать транзакции?

```
BEGIN; -- Открывает транзакцию
  -- твои операции
COMMIT; -- Фиксирует изменения
BEGIN;
  -- что-то пошло не так
ROLLBACK; -- Отменяет всё что было сделано внутри
```

Ситуация	Что происходит
`COMMIT`	Изменения сохранены
`ROLLBACK`	Изменения отменены явно
Ошибка SQL внутри транзакции	PostgreSQL сам откатит при попытке COMMIT
Соединение оборвалось	PostgreSQL автоматически делает ROLLBACK

Базовый шаблон транзакции

```
BEGIN;

UPDATE stock_balance
SET qty_on_hand = qty_on_hand - 10
WHERE warehouse_id = 1
  AND product_id = 10;

UPDATE stock_balance
SET qty_on_hand = qty_on_hand + 10
WHERE warehouse_id = 2
  AND product_id = 10;

COMMIT;
```


Если что-то пошло не так: ROLLBACK

```
BEGIN;

UPDATE stock_balance
SET qty_on_hand = qty_on_hand - 10
WHERE warehouse_id = 1
  AND product_id = 10;

-- ошибка проверки или исключение
ROLLBACK;
```

После ROLLBACK изменения внутри транзакции не сохраняются.

Автокоммит

В PostgreSQL каждый запрос по умолчанию — это отдельная транзакция.

Если написать просто `UPDATE` без `BEGIN` — PostgreSQL сам обернёт его в транзакцию и сразу закоммитит.

Это называется autocommit.

Поэтому `BEGIN` нужен только когда есть необходимость объединить несколько операций в одну транзакцию.

SAVEPOINT

`SAVEPOINT` нужен, когда внутри транзакции есть шаг, который можно откатить отдельно.

Например:

- основное изменение обязательно
- служебная запись в журнал - нет

SAVEPOINT

```
BEGIN;

UPDATE stock_balance
SET qty_reserved = COALESCE(qty_reserved, 0) + 5,
    updated_at = now()
WHERE warehouse_id = 1
  AND product_id = 10;

SAVEPOINT before_audit;

INSERT INTO stock_balance_audit(
  changed_at, op, warehouse_id, product_id, actor
)
VALUES (now(), 'RESERVE', 1, 10, current_user);
--Проверяем все ли ок, но может откатиться
ROLLBACK TO SAVEPOINT before_audit;
COMMIT;
```


Аномалии конкурентного доступа

Это типовые проблемы, которые возникают при параллельной работе транзакций:

- dirty read (грязное чтение)
- non-repeatable read (неповторяемое чтение)
- phantom read (фантомные строки)
- lost update (потерянное обновление)

Dirty Read — грязное чтение

Транзакция читает данные, которые другая транзакция изменила, но еще не закоммитила.

t	Транзакция A	Транзакция B
1	`UPDATE products SET price = 500 WHERE id = 1`	
2		`SELECT price FROM products WHERE id = 1 → видит 500`
3	`ROLLBACK — передумал менять цену`	
4		Покупатель оформляет заказ по цене 500, реальная цена 999

Dirty Read — грязное чтение

Покупатель прочитал цену которой никогда не существовало.
Магазин получает заказ по неверной цене.

Реальные сценарии где это критично:

- финансовые операции
- цены
- остатки на складе
- баланс счёта

Non-repeatable Read — неповторяемое чтение

Транзакция читает одну и ту же строку дважды и получает разные значения, потому что другая транзакция успела её изменить и закоммитить между двумя чтениями.

Отличие от Dirty Read — здесь другая транзакция успела закоммитить. Данные настоящие, но транзакция видит разное в рамках одной своей работы.

Non-repeatable Read — бухгалтерский отчёт

t	Транзакция A	Транзакция B
1	`SELECT balance FROM accounts WHERE id = 1` → 10 000	
2		`UPDATE accounts SET balance = 3000 WHERE id = 1`
3		`COMMIT`
4	`SELECT balance FROM accounts WHERE id = 1` → 3 000	
5	Бухгалтер считает отчёт — данные внутри одного отчёта противоречат друг другу	

Бухгалтер в рамках одного отчёта видит разные значения одного счёта. Отчёт несогласован.

Non-repeatable Read — бухгалтерский отчёт

Реальные сценарии:

- формирование отчётов
- аналитика
- любые вычисления которые делают несколько обращений к одним данным

Phantom Read — фантомное чтение

Транзакция делает один и тот же запрос с условием дважды и получает разное количество строк, потому что другая транзакция вставила или удалила строки между этими двумя чтениями.

Отличие от Non-repeatable Read — там меняется значение в строке, здесь меняется количество строк.

Phantom Read — бронирование мест

t	Транзакция A	Транзакция B
1	`SELECT COUNT(*) FROM seats WHERE free = true` → 1 свободное место	
2		`INSERT INTO bookings ...` — занял последнее место
3		`COMMIT`
4	Система решает: место есть, можно бронировать	
5	`SELECT COUNT(*) FROM seats WHERE free = true` → 0 мест	
6	Два человека получили подтверждение на одно место	

Phantom Read

Реальные сценарии:

- бронирование билетов
- номеров в отеле
- запись к врачу
- любые ситуации с ограниченным ресурсом

Lost Update — потерянное обновление

Две транзакции читают одно значение, обе его изменяют на основе прочитанного, и одно из обновлений перезаписывает другое — как будто его не было.

Это особенная аномалия — данные не «грязные» и не «фантомные», просто одно реальное изменение бесследно исчезает.

Lost Update — склад

t	Транзакция A	Транзакция B
1	`SELECT stock FROM products WHERE id = 1 → 100`	
2		`SELECT stock FROM products WHERE id = 1 → 100`
3	`Продал 10 единиц: UPDATE ... SET stock = 90`	
4	`COMMIT`	
5		`Продал 5 единиц: UPDATE ... SET stock = 95`
6		`COMMIT`
7	В базе: 95. Реально продано: 15. Должно быть: 85	

Lost Update

Обновление первого кладовщика потеряно — второй перезаписал его своим значением, потому что читал те же исходные данные.

Реальные сценарии:

- счётчики просмотров
- остатки на складе
- количество лайков
- баланс бонусных баллов — любое место где несколько пользователей могут одновременно менять одно значение

Уровни изоляции

Если существуют аномалии, может просто запретить их все?

Можно. Но цена — скорость.

Чем строже изоляция, тем больше транзакции мешают друг другу, тем больше ожиданий и блокировок, тем ниже производительность.

В реальных системах не всегда нужна максимальная защита — иногда достаточно частичной, зато быстрой.

Поэтому стандарт SQL определяет четыре уровня изоляции — каждый запрещает разные аномалии.

Уровни изоляции

Уровень изоляции показывает, насколько транзакция защищена от параллельных изменений других транзакций.

Удобно запоминать уровни как лестницу: каждый следующий уровень запрещает больше проблем, чем предыдущий.

- `READ UNCOMMITTED`: почти нет защиты; можно увидеть даже незакоммиченные изменения другой транзакции
- `READ COMMITTED`: читаем только закоммиченные данные; dirty read уже невозможен
- `REPEATABLE READ`: повторное чтение той же строки в рамках транзакции не должно “прыгать”; dirty read и non-repeatable read исключаются
- `SERIALIZABLE`: самый строгий режим; СУБД старается вести себя так, будто транзакции выполнились по очереди

Уровни изоляции

Уровень	Dirty Read	Non-repeatable Read	Phantom Read	Lost Update
Read Uncommitted	✅	✅	✅	✅
Read Committed	❌	✅	✅	✅
Repeatable Read	❌	❌	✅	❌
Serializable	❌	❌	❌	❌

Это стандарт SQL. PostgreSQL реализует его немного иначе.

Как установить уровень изоляции в PostgreSQL

```
BEGIN TRANSACTION ISOLATION LEVEL READ COMMITTED;

-- или после BEGIN:
BEGIN;
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;
```

Уровень изоляции устанавливается на одну транзакцию. Изменить его после первого запроса внутри транзакции уже нельзя.

Можно также изменить уровень по умолчанию для всей сессии:

```
SET default_transaction_isolation = 'repeatable read';
```


Read Committed

Уровень по умолчанию в PostgreSQL

Что гарантирует: транзакция видит только закоммиченные данные. Грязного чтения нет.

Что не гарантирует: одна и та же строка может вернуть разные значения в рамках одной транзакции — если другая транзакция успела закоммитить изменение между двумя чтениями.

Как это работает в PostgreSQL: каждый отдельный запрос внутри транзакции видит снимок данных на момент начала этого запроса, а не на момент начала транзакции. Поэтому два одинаковых SELECT в одной транзакции могут вернуть разные данные.

Read Committed

```
-- Транзакция A
BEGIN;
SELECT balance FROM accounts WHERE id = 1;  -- видит 1000

-- В этот момент транзакция Б делает UPDATE + COMMIT, balance стал 500

SELECT balance FROM accounts WHERE id = 1;  -- видит 500
COMMIT;
```

Когда использовать: подходит для большинства операций где не нужна гарантия что данные не изменятся в процессе.

Repeatable Read

Что гарантирует: транзакция видит снимок данных на момент начала транзакции и этот снимок не меняется до конца. Одна и та же строка всегда вернёт одно и то же значение.

Важное отличие PostgreSQL от стандарта SQL: в стандарте Repeatable Read не защищает от Phantom Read. В PostgreSQL — защищает. PostgreSQL использует MVCC (механизм версионирования строк) и снимок данных захватывается на всю транзакцию, поэтому фантомов тоже не возникает.

Repeatable Read

```
-- Транзакция А
BEGIN TRANSACTION ISOLATION LEVEL REPEATABLE READ;
SELECT balance FROM accounts WHERE id = 1;  -- видит 1000

-- Транзакция Б делает UPDATE + COMMIT, balance стал 500

SELECT balance FROM accounts WHERE id = 1;  -- всё равно видит 1000
COMMIT;
```

Когда использовать: отчёты и аналитика где важно что данные не изменятся в процессе вычислений. Например, формируете финансовый отчёт и делаете десяток запросов — все они должны видеть одно и то же состояние базы.

Serializable

Что гарантирует: транзакции выполняются так, как будто они идут строго одна за другой, а не параллельно. Все аномалии исключены, включая сложные зависимости между транзакциями которые не покрывают предыдущие уровни.

Как работает в PostgreSQL: PostgreSQL использует SSI (Serializable Snapshot Isolation) — он отслеживает зависимости между транзакциями. Если обнаруживает что параллельное выполнение дало бы результат невозможный при последовательном — одна из транзакций получает ошибку и должна повториться.

Serializable

Это значит что приложение должно быть готово к ошибке и уметь повторить транзакцию:

```
-- Транзакция может завершиться с ошибкой:
-- ERROR: could not serialize access due to read/write dependencies
-- В этом случае нужно повторить BEGIN ... COMMIT заново
```

Когда использовать: финансовые операции где критична абсолютная точность, сложные бизнес-правила которые зависят от нескольких таблиц одновременно. Использовать по умолчанию не стоит — это самый медленный уровень.

Read Uncommitted — почему его нет в PostgreSQL

Формально этот уровень есть в стандарте SQL и PostgreSQL его принимает как команду — но внутри всё равно работает как Read Committed.

PostgreSQL принципиально не читает незакоммиченные данные, это заложено в архитектуру MVCC.

На практике: если напишете `SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED` — PostgreSQL молча применит Read Committed.

Блокировки

Уровни изоляции описывают что видят транзакции. Блокировки — это механизм который это обеспечивает, плюс инструмент для явного контроля доступа к данным.

В PostgreSQL чтение не блокирует запись, а запись не блокирует чтение — это заслуга MVCC.

Блокировки в основном возникают когда несколько транзакций хотят изменить одно и то же.

Row-level locks - блокировки на уровне строк

Самый частый вид блокировок. Возникают автоматически при изменении строк.

```
BEGIN;
UPDATE accounts SET balance = balance - 1000 WHERE id = 1;
-- строка с id=1 заблокирована до COMMIT или ROLLBACK
-- другая транзакция которая попытается UPDATE эту же строку — будет ждать
COMMIT;
```

Другая транзакция не получит ошибку сразу — она будет ждать пока первая не завершится.

Row-level locks

SELECT FOR UPDATE — явная блокировка строки:

Иногда нужно заблокировать строку ещё на этапе чтения — чтобы никто не успел её изменить до того как ты сделаешь UPDATE.

```
BEGIN;
SELECT balance FROM accounts WHERE id = 1 FOR UPDATE;
-- строка заблокирована, хотя мы только читали
-- другая транзакция не сможет ни изменить её, ни тоже сделать FOR UPDATE

UPDATE accounts SET balance = balance - 1000 WHERE id = 1;
COMMIT;
```

Это решение для Lost Update — читаем и сразу говорим «я буду это менять, никого не пускать».

Row-level locks

SELECT FOR SHARE — мягкая блокировка:

```
SELECT * FROM accounts WHERE id = 1 FOR SHARE;
-- другие транзакции тоже могут читать с FOR SHARE
-- но никто не сможет сделать UPDATE или FOR UPDATE
```

Используется когда хочешь гарантировать что данные не изменятся, но готов пустить других читателей.

Table-level locks — блокировки на уровне таблицы

Возникают при операциях которые затрагивают всю таблицу.
PostgreSQL берёт их автоматически.

Примеры когда PostgreSQL берёт табличную блокировку:

```
-- ALTER TABLE блокирует всю таблицу на время выполнения
ALTER TABLE accounts ADD COLUMN currency varchar(3);

-- TRUNCATE тоже
TRUNCATE TABLE logs; -- Удаляет все данные из таблицы logs

-- обычный UPDATE берёт только ROW EXCLUSIVE на таблицу
-- это мягкая блокировка, не мешает другим UPDATE/SELECT
UPDATE accounts SET balance = 0 WHERE id = 1;
```

Table-level locks

Явная табличная блокировка:

```
BEGIN;
LOCK TABLE accounts IN EXCLUSIVE MODE;
-- теперь никто другой не может ни читать ни писать в таблицу
-- используется для критических операций с целой таблицей
COMMIT;
```

`ALTER TABLE` на большой таблице в продакшене — это риск.

Пока он выполняется, таблица заблокирована и все запросы к ней ждут.

Advisory locks — пользовательские блокировки

Это блокировки которые PostgreSQL не берёт сам — их берёт приложение явно для своей логики. База данных просто хранит информацию о том что блокировка взята.

Когда нужны: когда необходимо синхронизировать не строки в таблице, а какой-то процесс в приложении. Например, гарантировать что определённую задачу выполняет только один воркер.

```
-- взять блокировку по произвольному числу (ключу)
SELECT pg_try_advisory_lock(12345);
-- вернёт true если удалось взять, false если уже занята

-- отпустить
SELECT pg_advisory_unlock(12345);
```

Advisory locks

Очередь задач:

```
BEGIN;
-- пытаемся взять задачу, сразу блокируя её
SELECT * FROM jobs
WHERE status = 'pending'
AND pg_try_advisory_lock(id)
LIMIT 1;

-- если получили задачу — обрабатываем
-- другой воркер не возьмёт эту же задачу
COMMIT;
```

Advisory locks часто используют в системах с воркерами, планировщиками задач, и везде где нужно «только один процесс делает это в один момент времени».

Как резервируют товар

Обычно шаги такие:

- заблокировали строку
- проверили остаток
- обновили данные
- проверили, что изменилась ровно одна строка

Как резервируют товар

```sql
BEGIN;
SELECT
  COALESCE(qty_on_hand, 0) - COALESCE(qty_reserved, 0) AS qty_available
FROM stock_balance
WHERE warehouse_id = 1
  AND product_id = 10
FOR UPDATE;
-- проверка: qty_available >= 5
UPDATE stock_balance
SET qty_reserved = COALESCE(qty_reserved, 0) + 5,
    updated_at = now()
WHERE warehouse_id = 1
  AND product_id = 10
  AND COALESCE(qty_on_hand, 0) - COALESCE(qty_reserved, 0) >= 5;
-- далее контролируем, что обновилась одна строка
COMMIT;
```

Deadlock’и

Deadlock возникает когда две транзакции ждут друг друга и ни одна не может продолжить. Каждая держит блокировку на ресурс который нужен другой — и обе ждут бесконечно. Без внешнего вмешательства они так и будут стоять.

Deadlock’и

t	Транзакция A	Транзакция B
1	`UPDATE accounts SET balance = balance - 1000 WHERE id = 1` — заблокировала строку 1	
2		`UPDATE accounts SET balance = balance - 500 WHERE id = 2` — заблокировала строку 2
3	`UPDATE accounts SET balance = balance + 1000 WHERE id = 2` — ждёт, строка 2 занята	
4		`UPDATE accounts SET balance = balance + 500 WHERE id = 1` — ждёт, строка 1 занята

Deadlock’и

t	Транзакция A	Транзакция B
5	🔴 Обе транзакции ждут друг друга — deadlock	

Как PostgreSQL обнаруживает deadlock

PostgreSQL не ждёт вечно. Есть параметр `deadlock_timeout` — по умолчанию 1 секунда.

Когда транзакция ждёт блокировку дольше этого времени, PostgreSQL запускает алгоритм обнаружения deadlock — строит граф ожидания и проверяет есть ли в нём цикл.

```
А ждёт Б → Б ждёт А → цикл найден → deadlock
```


Как PostgreSQL разрешает deadlock

Когда цикл найден — PostgreSQL выбирает одну из транзакций как жертву и убивает её с ошибкой:

```
ERROR: deadlock detected
DETAIL: Process 1234 waits for ShareLock on transaction 5678;
        blocked by process 5678.
        Process 5678 waits for ShareLock on transaction 1234;
        blocked by process 1234.
HINT: See server log for query details.
```

Транзакция-жертва получает ROLLBACK, вторая — продолжает работу.

Приложение должно быть готово к этой ошибке и повторить транзакцию.

Как избежать deadlock

1. Всегда захватывать ресурсы в одном порядке

```
-- Обе транзакции делают так:
BEGIN;
UPDATE accounts SET balance = balance - 1000 WHERE id = 1;  -- сначала
меньший id
UPDATE accounts SET balance = balance + 1000 WHERE id = 2;  -- потом больший
COMMIT;
```

Как избежать deadlock

2 Использовать SELECT FOR UPDATE заранее

```
BEGIN;
-- сначала блокируем всё что понадобится
SELECT * FROM accounts WHERE id IN (1, 2) ORDER BY id FOR UPDATE;

-- теперь делаем изменения
UPDATE accounts SET balance = balance - 1000 WHERE id = 1;
UPDATE accounts SET balance = balance + 1000 WHERE id = 2;
COMMIT;
```

ORDER BY id здесь важен — гарантирует что блокировки берутся в одном порядке даже если транзакции запустились параллельно.

3 Держать транзакции короткими

Чем дольше транзакция держит блокировки, тем выше шанс deadlock.

pg_locks — смотрим активные блокировки

pg_locks — это системное представление (view) в PostgreSQL которое показывает все активные блокировки в реальном времени. Это главный инструмент когда нужно понять почему транзакция зависла, кто кого блокирует и что вообще происходит в базе.

```
SELECT * FROM pg_locks;
```

```
SELECT
    locktype,       -- что блокируем: relation, tuple, transactionid...
    relation::regclass,  -- имя таблицы (если блокировка на таблицу/строку)
    pid,            -- id процесса который держит или ждёт блокировку
    mode,           -- режим блокировки: RowExclusiveLock, ShareLock...
    granted         -- true = держит блокировку, false = ждёт
FROM pg_locks;
```

`granted = false` — самое важное. Это транзакция которая ждёт — именно она сигнализирует о проблеме.

pg_stat_activity — смотрим что делают процессы

pg_locks часто используют вместе с pg_stat_activity — там видно текущие запросы всех подключений:

```
SELECT
    pid,
    state,          -- active, idle, idle in transaction
    query,
    now() - query_start AS duration
FROM pg_stat_activity
WHERE state != 'idle'
ORDER BY duration DESC;
```

`idle in transaction` — транзакция открыта но ничего не делает. Это опасно — она держит блокировки пока приложение думает или забыло закрыть транзакцию. В продакшене это частая причина проблем.

Как убить зависшую транзакцию

```
-- мягко — проспросить завершить текущий запрос
SELECT pg_cancel_backend(pid);

-- жёстко — убить процесс полностью
SELECT pg_terminate_backend(pid);
```

pg_cancel_backend посылает сигнал отменить текущий запрос — транзакция получит ошибку и откатится. pg_terminate_backend убивает соединение полностью.

Реальные примеры транзакций

```python
import psycopg2
conn = psycopg2.connect("dbname=bank")
try:
    with conn:  # это автоматический BEGIN + COMMIT/ROLLBACK
        with conn.cursor() as cur:
            cur.execute(
                "UPDATE accounts SET balance = balance - %s WHERE id = %s",
                (1000, 1)
            )
            cur.execute(
                "UPDATE accounts SET balance = balance + %s WHERE id = %s",
                (1000, 2)
            )
    # если исключения не было — автоматический COMMIT
except Exception as e:
    # автоматический ROLLBACK
    print(f"Транзакция откатилась: {e}")
```

Реальные примеры транзакций

```
from sqlalchemy.orm import Session

def transfer_money(session: Session, from_id: int, to_id: int, amount: int):
    try:
        from_account = session.get(Account, from_id)
        to_account = session.get(Account, to_id)

        from_account.balance -= amount
        to_account.balance += amount

        session.commit()
    except Exception:
        session.rollback()
        raise
```