Лекция 13.

Транзакции и блокировки в базах данных

Транзакции в СУБД

Транзакция – неделимая последовательность операторов манипулирования данными (чтения, удаления, вставки, модификации), приводящая к одному из двух возможных результатов:

- либо последовательность выполняется, если все операторы правильные,
- либо вся транзакция откатывается, если хотя бы один оператор не может быть успешно выполнен.

Транзакции в СУБД

Корректная поддержка транзакций:

- Основа надежности и целостности базы данных.
- Основа изолированности, параллельной работы нескольких пользователей. Параллелизм транзакций.

Команды для управления транзакцииями

1. `BEGIN TRANSACTION` **(или просто** `BEGIN` **в некоторых СУБД)**

- Начинает новую транзакцию.
- После этой команды все последующие DML-операции (INSERT, UPDATE, DELETE) входят в состав транзакции до её завершения.
- Пример:

sql

```sql
BEGIN TRANSACTION;
```

> В некоторых СУБД (например, PostgreSQL) используется `BEGIN;`, а в MySQL — `START TRANSACTION;`.

Команды для управления транзакцииями

2. `COMMIT`

- Фиксирует все изменения, сделанные в рамках текущей транзакции.
- После `COMMIT` изменения становятся постоянными и видны другим пользователям.
- Пример:

```sql
COMMIT;
```

---

3. `ROLLBACK`

- Отменяет все изменения, сделанные в текущей транзакции.
- База данных возвращается в состояние, которое было до начала транзакции.
- Пример:

```sql
ROLLBACK;
```

Команды для управления транзакцииями

4. `SAVEPOINT` и `ROLLBACK TO SAVEPOINT` **(опционально поддерживается)**

- Позволяют создавать точки сохранения внутри транзакции.
- Можно откатиться не ко всему началу транзакции, а только к определённой точке.
- Пример:

sql

```sql
SAVEPOINT sp1;
-- какие-то операции
ROLLBACK TO sp1;
```

> Поддержка `SAVEPOINT` зависит от СУБД (есть в PostgreSQL, Oracle, MySQL, но не во всех).

Команды для управления транзакцииями

Пример полной транзакции (в стиле PostgreSQL / SQL Standard):

sql

```sql
BEGIN;
UPDATE accounts SET balance = balance - 100 WHERE user_id = 1;
UPDATE accounts SET balance = balance + 100 WHERE user_id = 2;
COMMIT;
```

Если между `BEGIN` и `COMMIT` произойдёт ошибка, можно выполнить `ROLLBACK`, чтобы отменить оба изменения.

ACID-свойства транзакций

ACID описывает требования к транзакционной системе, обеспечивающие наиболее надёжную и предсказуемую её работу.

- **A**tomicity - атомарность
- **C**onsistency - согласованность
- **I**solation - изолированность
- **D**urability - долговечность

ACID-свойства транзакций

1. Atomicity (Атомарность)

> "Всё или ничего"

- Транзакция — **неделимая единица**: либо **все** её операции успешно завершаются и применяются, либо **ни одна** из них не оказывает эффекта.
- Обеспечивается с помощью **журнала транзакций (transaction log)** и механизма **отката (rollback)**.
- Пример:
  При переводе 1000 ₽ с одного счёта на другой должны выполниться **оба действия**: списание и зачисление. Если второе не удалось — первое откатывается.

ACID-свойства транзакций

2. Consistency (Согласованность)

> "База данных всегда остаётся в валидном состоянии"

- Транзакция должна **сохранять все инварианты** базы данных:
  — ограничения (`CHECK`, `UNIQUE`, `FOREIGN KEY`),
  — триггеры,
  — бизнес-правила.
- Важно: **СУБД не проверяет бизнес-логику**, но гарантирует, что если транзакция не нарушает правил — согласованность сохранится.
- Пример:
  Если в таблице `accounts` есть ограничение `balance >= 0`, то транзакция, делающая баланс отрицательным, будет отклонена — согласованность не нарушится.

ACID-свойства транзакций

3. Isolation (Изолированность)

> "Параллельные транзакции не мешают друг другу"

- Даже если несколько транзакций выполняются **одновременно**, результат должен быть таким же, как если бы они выполнялись **последовательно** (в каком-то порядке).
- Реализуется через:
  - **Блокировки (locking)** — традиционный подход.
  - **Многоверсионность (MVCC — Multi-Version Concurrency Control)** — современный подход (PostgreSQL, Oracle, InnoDB).
- Уровни изоляции (SQL стандарт):
  - `READ UNCOMMITTED`
  - `READ COMMITTED`
  - `REPEATABLE READ`
  - `SERIALIZABLE`
- Более высокий уровень → больше изоляции, но ниже производительность.

ACID-свойства транзакций

4. Durability (Долговечность)

> "Зафиксированное — навсегда"

- После того как транзакция завершена (`COMMIT`), её результат **гарантированно сохраняется**, даже если сразу после этого произойдёт:
  - сбой питания,
  - падение сервера,
  - сбой диска (при наличии резервирования).
- Обеспечивается с помощью:
  - **Write-Ahead Logging (WAL)**: изменения сначала пишутся в **журнал на диске**, и только потом — в основные данные.
  - Синхронной записи (`fsync`) для гарантии фиксации на физическом носителе.
- Пример:
  Вы получили подтверждение перевода денег → даже при перезагрузке базы эти деньги **не исчезнут**.

ACID-свойства транзакций

🧩 Как ACID связаны между собой?

- **Атомарность** + **Долговечность** → надёжность при сбоях.
- **Изолированность** → корректность при параллелизме.
- **Согласованность** — **результат** корректного применения других свойств плюс соблюдение правил базы.

Распределенные (NoSQL) системы могут не поддерживать одновременно все ACID-свойства.

Теорема CAP

Распределенная система при возможности сетевых сбоев не может гарантировать одновременного выполнения следующих трёх свойств:

- **Consistency**. **Согласованность** - все пользователи в любой момент времени видят одинаковые данные.
- **Availability**. **Доступность** – при выходе из строя каких-либо узлов оставшиеся узлы продолжают функционировать. Система всегда отвечает на запросы, даже если часть серверов не работает.
- **Partition Tolerance**. **Устойчивость к разделению** – при распадении системы на отдельные группы узлов из-за сбоя сети каждая группа продолжает работать.

Теорема CAP

Если **сеть разделена (P выполняется)**, то система **не может одновременно быть и полностью согласованной, и полностью доступной**.

Простой смысл доказательства:

При сетевом разделении два узла не могут обмениваться данными.

- Если система требует согласованности — она должна отказать в записи или чтении, чтобы не нарушить согласованность → теряет доступность.
- Если система требует доступности — она должна принять запрос даже при разделении, но тогда данные могут быть несогласованными → теряет согласованность.

Примеры CAP

🧑‍💼 Пример 1: Банковский счёт (выбираем CP — согласованность и устойчивость к сбоям)

Представь, у тебя есть счёт в банке, и ты пытаешься снять деньги через банкомат. Если банкомат **не может связаться с центральным сервером** (сетевое разделение), он **откажет в выдаче денег**, чтобы не дать тебе снять больше, чем есть на счёте.

- ✅ Согласованность: все операции видят актуальный баланс.
- ✅ Устойчивость к разделению: система знает, что сеть может отвалиться.
- × Доступность: ты не получил деньги, хотя банкомат физически работает.

Примеры CAP

🛒 **Пример 2: Интернет-магазин (выбираем AP — доступность и устойчивость)**

Ты добавляешь товар в корзину на сайте, а сервер, где хранится твоя корзина, временно не отвечает.
Сайт всё равно **позволяет тебе продолжить покупку**, сохраняя данные локально или на другом сервере.

- ✅ Доступность: ты можешь работать дальше.
- ✅ Устойчивость к разделению: система работает даже при сбоях.
- × Согласованность: возможно, позже выяснится, что товара нет на складе, или корзина "сбросилась".

Примеры CAP

🏥 **Пример 3: Медицинская система (может выбирать CA, но только если нет разделения)**

Представь локальную больничную базу данных, работающую **в одной сети без репликации**. Тогда можно быть и согласованной, и доступной — но **только пока сеть не разделится**.

Если сеть надёжна и всегда цела, **P не требуется**, и можно иметь CA.

Но в распределённых (облачных) системах это нереалистично.

Резюме по CAP

CAP-теорема — **не рецепт**, а **осознание компромиссов**:

> «При сетевом сбое вы **не можете** быть одновременно полностью доступны и полностью согласованны. Выберите, что важнее для вашего сценария».

- **Финансовые системы** → чаще выбирают **CP** (лучше недоступность, чем ошибочные транзакции).
- **Соцсети, IoT, логирование** → чаще выбирают **AP** (лучше устаревшие данные, чем отказ в записи).

Понимание CAP помогает принимать **архитектурные решения** при проектировании масштабируемых и отказоустойчивых систем.

Согласованность в конечном счете

- **Что такое «согласованность в конечном счёте»?**

Eventual Consistency (EC) — это **ослабленная форма согласованности**, при которой:

> Если **никто больше не меняет данные**, то **все узлы системы со временем придут к одному и тому же значению**.

То есть:

- **Сразу после записи** другие узлы **могут видеть старые данные** — это нормально.
- Но **спустя какое-то время** (секунды, минуты — зависит от системы) **все узлы синхронизируются**, и данные становятся одинаковыми.

💡 Главное: **гарантия только «в конечном счёте»**, а не мгновенно.

Согласованность в конечном счете

- **Пример из жизни**

Представь, что ты редактируешь документ в Google Docs на телефоне, а твой коллега — на компьютере.

- Ты добавил строку, но у него она **пока не появилась** (сеть медленная, сервера не успели синхронизироваться).
- Через пару секунд он **всё же видит твою строку**.

Это **eventual consistency**: данные не синхронизированы мгновенно, но **в итоге становятся одинаковыми**.

Другой пример — рассылка постов в соцсети. Иногда ты публикуешь пост, а друзья в другой стране видят его с задержкой. Через минуту — уже все видят.

Согласованность в конечном счете

- **Когда подходит eventual consistency?**

✅ Когда:

- Важна **высокая доступность** (сайт не должен падать).
- Небольшая **задержка в синхронизации допустима** (например, лайки, комментарии, рекомендации).
- Система **масштабируется на тысячи серверов**.

× Когда **НЕ подходит**:

- Финансовые переводы (нельзя, чтобы баланс "временно не совпадал").
- Резервирование билетов (нельзя продать один и тот же билет дважды).
- Любые операции, где **мгновенная точность критична**.

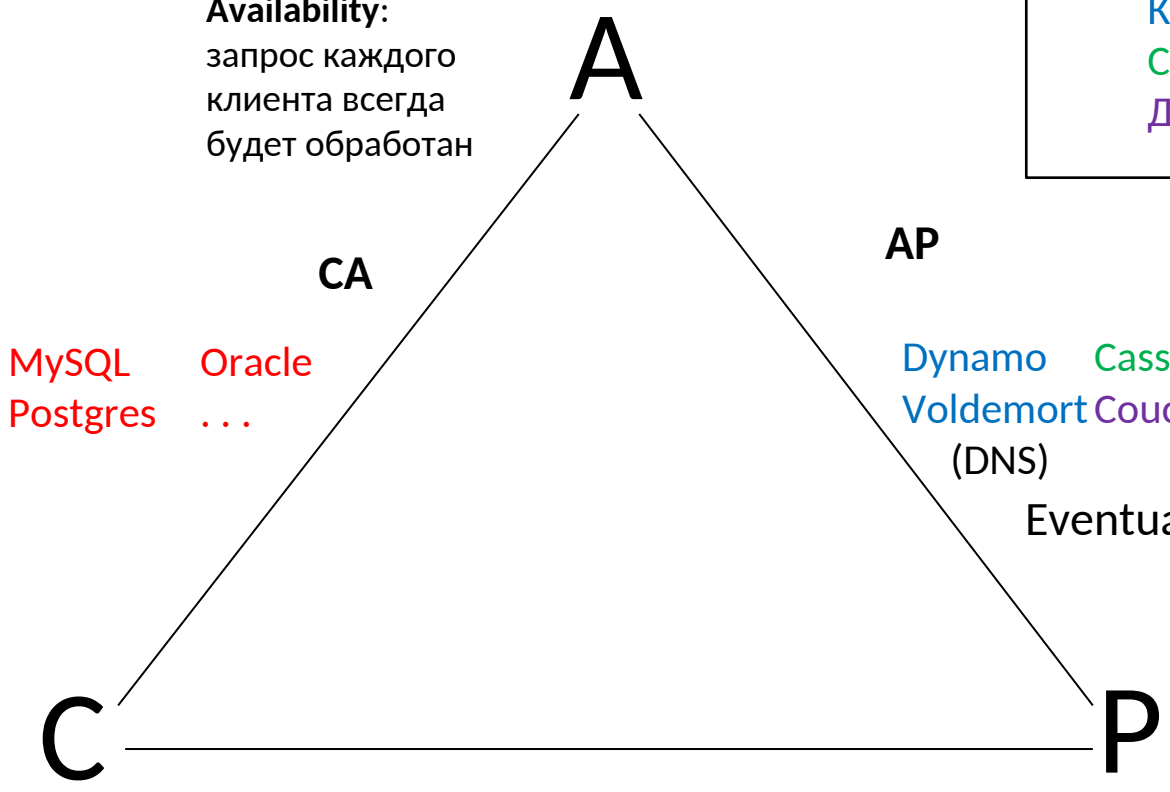
Согласованность в конечном счете

- **Eventual consistency** — это **практический компромисс**, позволяющий строить **масштабируемые и доступные системы**.
- Она **не нарушает CAP-теорему**, а **работает в рамках её ограничений**, выбирая **AP** и отказываясь от строгой **C**.
- Это **не «поломка» системы**, а **осознанный выбор модели данных** под бизнес-требования.

Теорема CAP

Модели данных:
- Реляционная
- Ключ-значение
- Столбцовая
- Документная

Availability:
запрос каждого клиента всегда будет обработан



CA

MySQL Oracle
Postgres . . .

AP

Dynamo Cassandra Riak
Voldemort CouchDB SimpleDB
(DNS)

Eventual consistency

Consistency:
все всегда видят одни и те же данные

CP

BigTable MongoDB Redis
Hbase TerraStore MemcacheDB

Partition tolerance:
система продолжает работать после физического разделения

Журнал транзакций

Журналы транзакций - фундаментальный механизм для выполнения требований A, I, Д к транзакциям.

- **Atomicity**. Транзакция **неделима**.
- **Isolation**. Транзакция **изолирована**, ее результаты самодостаточны. Незаконченная (неподтвержденная) транзакция должна быть невидима извне.
- **Durability**. Транзакция **устойчива (долговечна)**, ее действие постоянно даже при сбое системы. После завершения транзакции изменения должны стать доступны всем и не быть потеряны.

Основные типы журналов в СУБД

1. **Журнал транзакций (Transaction Log / Write-Ahead Log — WAL)**
   Записывает все изменения, вносимые транзакциями, **до** их фактического применения к данным на диске. Это ключевой компонент для обеспечения **ACID-свойств**, особенно **durability** (устойчивости).
2. **Журнал отката (Undo Log)**
   Содержит информацию, необходимую для **отката** транзакций (например, исходные значения изменённых данных). Используется при отмене транзакции или при восстановлении после сбоя.
3. **Журнал повторного выполнения (Redo Log)**
   Содержит информацию, необходимую для **повторного применения** завершённых транзакций при восстановлении после сбоя.

Некоторые СУБД используют комбинацию redo/undo, другие — только WAL.

Реализация журналов в СУБД

1. PostgreSQL — Write-Ahead Logging (WAL)

- Все изменения сначала записываются в WAL-файл, затем применяются к данным.
- Используется для:
  - Восстановления после сбоя (crash recovery)
  - Репликации (логической и физической)
  - Point-in-time recovery (PITR)
- WAL-файлы хранятся в каталоге `pg_wal` (ранее `pg_xlog`).

Реализация журналов в СУБД

2. MySQL / InnoDB — Redo Log + Undo Log

- **Redo Log**: фиксирует физические изменения страниц данных. Используется при восстановлении после сбоя.
- **Undo Log**: хранит старые версии строк для отката транзакций и реализации MVCC (многоверсионного управления конкурентным доступом).
- Оба журнала хранятся в табличном пространстве или в отдельных файлах (`ib_logfile0`, `ib_logfile1` для redo).

Реализация журналов в СУБД

5. SQLite — WAL Mode или Journal Mode

- По умолчанию использует **rollback journal** (журнал отката): перед изменением создаётся резервная копия изменяемых страниц.
- В **WAL-режиме** (включается явно) изменения записываются в отдельный WAL-файл, что улучшает конкурентность и производительность.
- WAL позволяет читать данные, пока идёт запись.

```sql
PRAGMA journal_mode = WAL;
```

Упреждающая журнализация, Write-Ahead Log (WAL)

1. В памяти формируются **новые версии изменённых страниц**.
2. Эти новые страницы **добавляются в WAL-файл в памяти**.
3. При `COMMIT`:
   - WAL-файл **принудительно сбрасывается на диск** (`fsync`).
   - В WAL записывается **запись о коммите** (commit record).
4. Только **после успешной записи WAL на диск** СУБД возвращает клиенту «OK».
5. **Основной файл `.db` остаётся неизменным** — чтение теперь может брать данные из WAL.
6. Позже (в фоне или по команде) — **checkpoint** переносит данные из WAL в `.db`.

Для чего нужен журнал транзакций

1. Восстановление после сбоев

Проблема:
Представьте, что СУБД обрабатывает транзакцию: например, переводит 1000 рублей со счёта A на счёт B. В середине выполнения (после списания с A, но до зачисления на B) происходит сбой — отключилось электропитание или аварийно завершился процесс СУБД.

Решение через журналирование:
Благодаря журналу транзакций (например, WAL или redo/undo-логам), СУБД при перезапуске может:

- **Определить**, какие транзакции были полностью завершены (и гарантированно записаны на диск через журнал) — и **применить** их повторно (redo).
- **Выявить**, какие транзакции были прерваны — и **откатить** их изменения (undo), чтобы база данных осталась в согласованном состоянии.

2. Обеспечение ACID-свойств

Как журнал помогает:

- **Атомарность**: если транзакция прервана, журнал позволяет откатить её частично сделанные изменения.
- **Долговечность**: как только транзакция «закоммичена», её запись в журнал гарантированно сбрасывается на диск (`fsync`). Это означает, что даже если СУБД упадёт сразу после `COMMIT`, при перезапуске изменения будут восстановлены из журнала.
- **Согласованность**: достигается совместно с другими механизмами (ограничения, триггеры), но журнал гарантирует, что нарушенные транзакции не оставят «полуобновлённые» данные.

> Журналирование — ключевой механизм реализации **атомарности** и **долговечности**.

3. Поддержка резервного копирования "на лету" (hot backup / online backup)

Проблема:
Если делать резервную копию «просто скопировав файлы базы данных», пока СУБД работает, можно получить **неконсистентную копию**: данные на диске могут быть частично обновлены, а частично — нет.

Решение через журнал:

- СУБД делает **снимок данных** (например, копирует файлы), а **параллельно продолжает писать изменения в журнал**.
- При восстановлении из резервной копии СУБД:
  1. Загружает снимок.
  2. «Прокручивает» (replays) журнал с момента начала копирования до нужного момента времени.
- Такой подход называется **архивным журналированием** (archive logging) или **point-in-time recovery (PITR)**.

4. Репликация и высокая доступность

Современные СУБД часто используют **репликацию** — копирование данных на другие серверы для масштабирования, отказоустойчивости или балансировки нагрузки.

Как журнал участвует:

- **Физическая репликация**: реплика получает **байт-в-байт копии журнала** и применяет те же изменения, что и основной сервер.
  - Пример: PostgreSQL **streaming replication**, Oracle **Data Guard**, SQL Server **Log Shipping**.
- **Логическая репликация**: из журнала извлекаются **логические операции** (`INSERT`, `UPDATE` и т.д.), которые затем применяются на реплике.
  - Пример: логическая репликация в PostgreSQL, MySQL Row-Based Replication.

Почему журнал идеален для этого:

- Он содержит **все изменения** в строгом порядке.
- Его можно **передавать по сети** в реальном времени.
- Реплика всегда может «догнать» основной сервер, даже если была отключена.

Журнал транзакций

Итог:

Журнал транзакций служит **единственным источником истины** о том, что изменилось в базе — что делает его идеальной основой для репликации и обеспечения высокой доступности.

Блокировки

При выполнении транзакций СУБД накладывает на данные блокировки.

Блокировка - временное ограничение на выполнение некоторых операций обработки данных. Два вида:

- **блокировка записи** – транзакция блокирует строки в таблицах таким образом, что запрос другой транзакции к этим строкам будет отменен;
- **блокировка чтения** – транзакция блокирует строки так, что запрос со стороны другой транзакции на блокировку записи этих строк будет отвергнут, а на блокировку чтения – принят.

Проблема параллелизма

- Транзакция считывает строку таблицы → накладывает на нее блокировку чтения.
- Транзакция модифицирует строку данных → накладывает на нее блокировку записи.
- Запрашиваемая блокировка на строку отвергается из-за уже имеющейся блокировки → транзакция переводится в режим ожидания до снятия блокировки.
- Блокировка записи сохраняется до конца выполнения транзакции.

Deadlock

Ситуация в СУБД, при которой несколько транзакций ожидают освобождения ресурсов, занятых друг другом и ни одна из них не может продолжить свое выполнение.

- **Транзакция T1**:

```sql
UPDATE счета SET баланс = баланс - 100 WHERE id = 1;  -- блокирует строку 1
-- ... (делает что-то ещё)
UPDATE счета SET баланс = баланс + 100 WHERE id = 2;  -- хочет строку 2
```

- **Транзакция T2**:

```sql
UPDATE счета SET баланс = баланс - 50 WHERE id = 2;  -- блокирует строку 2
-- ... (делает что-то ещё)
UPDATE счета SET баланс = баланс + 50 WHERE id = 1;  -- хочет строку 1
```

Проблемы одновременного доступа к данным

Потерянное обновление – при одновременном изменении одного блока данных разными транзакциями одно из изменений теряется.

Транзакция 1	**Транзакция 2**
UPDATE tbl1 SET f2=f2+20 WHERE f1=1;	UPDATE tbl1 SET f2=f2+25 WHERE f1=1;

1. Обе транзакции одновременно читают текущее состояние поля.
2. Обе транзакции вычисляют новое значение поля.
3. Транзакции пытаются записать результат вычислений обратно в поле f2. Физически одновременно две записи выполнить невозможно, одна из операций записи будет выполнена раньше, другая позже. При этом вторая операция записи перезапишет результат первой. Первая транзакция «пропадет».

Проблемы одновременного доступа к данным

«Грязное» чтение – чтение данных, добавленных или изменённых транзакцией, которая впоследствии не подтвердится (откатится).

Транзакция 1

```
UPDATE tbl1 SET f2=f2+1 WHERE f1=1;

ROLLBACK;
```

Транзакция 2

```
SELECT f2 FROM tbl1 WHERE f1=1;
```

В транзакции 1 изменяется значение поля f2, а затем в транзакции 2 выбирается значение этого поля. После этого происходит откат транзакции 1. В результате значение, полученное второй транзакцией, будет отличаться от значения, хранимого в базе данных.

Проблемы одновременного доступа к данным

Неповторяющееся чтение – при повторном чтении в рамках одной транзакции ранее прочитанные данные оказываются изменёнными.

Транзакция 1	**Транзакция 2**
	SELECT f2 FROM tbl1 WHERE f1=1;
UPDATE tbl1 SET f2=f2+1 WHERE f1=1;	
COMMIT;	
	SELECT f2 FROM tbl1 WHERE f1=1;

В транзакции 2 выбирается значение поля f2, затем в транзакции 1 изменяется значение поля f2. При повторной попытке выбора значения из поля f2 в транзакции 2 будет получен другой результат.
Эта ситуация особенно неприемлема, когда данные считываются с целью их частичного изменения и обратной записи в базу данных.

Проблемы одновременного доступа к данным

Фантомное чтение – при повторном чтении в рамках одной транзакции одна и та же выборка дает разные множества строк.

Транзакция 1	**Транзакция 2**
	SELECT SUM(f2) FROM tbl1;
INSERT INTO tbl1 (f1,f2) VALUES (15,20);	
COMMIT;	
	SELECT SUM(f2) FROM tbl1;

В транзакции 2 выполняется SQL-оператор, использующий все значения поля f2. Затем в транзакции 1 выполняется вставка новой строки, приводящая к тому, что повторное выполнение SQL-оператора в транзакции 2 выдаст другой результат.

Проблемы одновременного доступа к данным

Аномалии сериализации – ситуация, когда параллельное выполнение транзакций приводит к результату, невозможному при последовательном выполнении тех же транзакций.

Транзакция 1

```
INSERT INTO tbl1 (f1,f2) VALUES (15,20);
COMMIT;
```

Транзакция 2

```
SELECT SUM(f2) FROM tbl1;

SELECT SUM(f2) FROM tbl1;
```

Уровни изоляции транзакций

Уровни изоляции транзакций — это механизм, с помощью которого СУБД управляет **видимостью изменений**, вносимых одной транзакцией, для других параллельно выполняющихся транзакций. Они позволяют находить баланс между **согласованностью данных** и **производительностью/параллелизмом**.

Уровни изоляции транзакций

Степень обеспечиваемой внутренними механизмами СУБД защиты от всех или некоторых видов несогласованностей данных, возникающих при параллельном выполнении транзакций.

Стандарт SQL-92 определяет четыре уровня изоляции:

1. **Read uncommitted** – чтение незафиксированных данных
2. **Read committed** – чтение фиксированных данных
3. **Repeatable read** – повторяемость чтения
4. **Serializable** – упорядочиваемость

Первый – самый слабый, последний — самый сильный, каждый последующий включает в себя все предыдущие.

		Эффекты				
		Потерянное обновление	Грязное чтение	Неповтор-ся чтение	Фантомное чтение	Аномалии сериализации
Уровни изоляции	Read uncommitted	Нет	Допускается	Возможно	Возможно	Возможно
	Read committed	Нет	Нет	Возможно	Возможно	Возможно
	Repeatable read	Нет	Нет	Нет	Допускается	Возможно
	Serializable	Нет	Нет	Нет	Нет	

Повышение уровня изоляции: точность и согласованность данных растет, количество параллельно выполняемых транзакций уменьшается.

Установка уровня изоляции транзакций

MySQL:

```
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
SET SESSION TRANSACTION ISOLATION LEVEL SERIALIZABLE;
SET GLOBAL TRANSACTION ISOLATION LEVEL SERIALIZABLE;
```

Уровни изоляции транзакций

Read uncommitted – чтение незафиксированных данных

Гарантирует только отсутствие потерянных обновлений. Если несколько параллельных транзакций пытаются изменять одну и ту же строку таблицы, то в окончательном варианте строка будет иметь значение, определенное всем набором успешно выполненных транзакций.

При этом возможно считывание не только логически несогласованных данных, но и данных, изменения которых ещё не зафиксированы.

Уровни изоляции транзакций

Read committed – чтение зафиксированных данных

Если транзакция начала изменение данных, то никакая другая транзакция не сможет прочитать их до завершения первой.

Тем не менее, в процессе работы одной транзакции другая может быть успешно завершена и сделанные ею изменения зафиксированы. В итоге первая транзакция будет работать с другим набором данных.

Уровни изоляции транзакций

Repeatable read (snapshot isolation) – повторяемость чтения

Если транзакция считывает данные, то никакая другая транзакция не сможет их изменить. При повторном чтении они будут находиться в первоначальном состоянии.

Однако другие транзакции могут вставлять новые строки, соответствующие условиям поиска инструкций, содержащихся в текущей транзакции. При повторном запуске инструкции текущей транзакцией будут извлечены новые строки, что приведёт к фантомному чтению.

MultiVersion Concurrency Control (MVCC)

MVCC — это аббревиатура от **Multiversion Concurrency Control** («многоверсионное управление конкурентным доступом»).
Это **механизм**, который используют многие современные СУБД (PostgreSQL, Oracle, MySQL/InnoDB, SQLite в WAL-режиме и др.), чтобы **разрешить параллельное выполнение транзакций без блокировок при чтении** и **сохранить изоляцию**.

- **Главная идея MVCC — простыми словами:**

> **Каждая транзакция видит «снимок» (snapshot) базы данных на момент её начала.**
> При этом **новые изменения не мешают читающим транзакциям**, потому что они работают со *старыми версиями данных*, а не ждут, пока кто-то закончит писать.

То есть:

- **Чтение не блокирует запись.**
- **Запись не блокирует чтение.**

Уровни изоляции транзакций

Repeatable read (snapshot isolation) – повторяемость чтения

MultiVersion Concurrency Control (MVCC)

- Разные пользователи могут одновременно работать с одними и теми же данными;
- Каждый пользователь видит свой изолированный срез данных;
- Изменения, вносимые пользователем, никому не видны до завершения транзации.

MVCC

Представь, что в таблице есть строка:

id	имя	возраст
1	Алиса	30

Теперь:

1. **Транзакция T1** начинается и читает строку → видит `возраст = 30`.
2. **Транзакция T2** начинается, **обновляет** возраст Алисы на `31` и **фиксирует** изменения.
3. **Транзакция T1** снова читает ту же строку.

MVCC

Без MVCC:

T1 либо увидит новое значение (нарушая изоляцию), либо будет заблокирована до завершения T2.

С MVCC:

- СУБД **сохраняет старую версию строки** (`возраст = 30`), пока есть активные транзакции, которым она нужна.
- T1 **продолжает видеть свою версию** — `30`.
- T2 работает с новой версией — `31`.

Таким образом, **обе транзакции работают независимо**, и изоляция соблюдается — например, на уровне **Repeatable Read** или **Snapshot Isolation**.

Уровни изоляции транзакций

Serializable – упорядочиваемость (блокирует чтение)
Если транзакция обращается к данным, то никакая другая транзакция не сможет добавить новые или удалить имеющие строки, которые могут быть считаны при выполнении транзакции.

Такая блокировка накладывается не на конкретные строки таблицы, а на строки, удовлетворяющие определенному логическому условию.

Как выбрать уровень изоляции транзакций?

- **Read Committed** — подходит для большинства OLTP-приложений.
- **Repeatable Read** — если в транзакции важно, чтобы повторные запросы возвращали одни и те же данные (например, при подготовке отчёта).
- **Serializable** — когда критична логическая целостность и нельзя допустить даже редкие аномалии (например, в финансовых системах).
- **Read Uncommitted** — почти не используется в production, разве что для «грязной» аналитики, где допустима неточность.

Оптимистичные и пессимистичные блокировки

Оптимистичная блокировка — это стратегия управления конкурентным доступом к данным, при которой:

> **Система предполагает, что конфликты между транзакциями — редкость**, и **не блокирует данные при чтении**, а **проверяет наличие конфликта только в момент записи**.

Если во время выполнения транзакции данные **не изменились**, запись разрешается.
Если **данные изменились** — транзакция **отклоняется**, и приложению нужно **повторить операцию**.

Оптимистичная блокировка

Часто используется **версионный контроль**:

- В таблице есть специальное поле: `version` (целое число) или `updated_at` (время).
- При чтении записывается текущая версия.
- При обновлении проверяется: **не изменилась ли версия с момента чтения?**

Оптимистичная блокировка

```sql
-- Читаем данные
SELECT id, name, version FROM accounts WHERE id = 1;
-- Получили: id=1, name='Алиса', version=5

-- Пытаемся обновить
UPDATE accounts
SET name = 'Ева', version = 6
WHERE id = 1 AND version = 5;  -- ← ключевое условие!
```

- Если `UPDATE` затронул **1 строку** → обновление успешно.
- Если затронул **0 строк** → кто-то уже изменил данные → конфликт → нужно **перечитать и повторить**.

Пессимистичная блокировка

Это **противоположный подход**:

> **Система предполагает, что конфликты — норма,**
> и **блокирует данные сразу при чтении**, чтобы никто другой не мог их изменить.

Пример (SQL):

sql

```sql
BEGIN;
SELECT * FROM accounts WHERE id = 1 FOR UPDATE;  -- ← блокирует строку
-- ... делаем логику ...
UPDATE accounts SET balance = balance - 100 WHERE id = 1;
COMMIT;  -- блокировка снимается
```

- Пока транзакция не завершится, **никто не может читать с `FOR UPDATE` или писать** в эту строку.
- Гарантирует **отсутствие конфликтов**, но **снижает параллелизм**.

Оптимистичная vs пессимистичная

Критерий	Оптимистичная	Пессимистична
Предположение	Конфликты редки	Конфликты часты
Блокировка при чтении	Нет	Да (`FOR UPDATE`)
Производительность при низкой конкуренции	Выше	Ниже
Производительность при высокой конкуренции	Ниже (много откатов)	Стабильна
Сложность приложения	Нужно обрабатывать откаты	Проще логика
Deadlock'и	Почти невозможны	Возможны

Оптимистичная vs пессимистичная

- **Оптимистичная блокировка** — «сначала делай, потом проверь». Отлично подходит для систем с **редкими конфликтами** (web-приложения, распределённые сервисы).
- **Пессимистичная блокировка** — «сначала заблокируй, потом делай». Надёжна для **критичных операций** (финансы, инвентарь).
- Современные СУБД (PostgreSQL, Oracle, InnoDB) **комбинируют оба подхода**:
  — MVCC + оптимистичная проверка для чтения,
  — пессимистичные блокировки при `SELECT ... FOR UPDATE`.