КАРАГАНДИНСКИЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ

Кафедра ИВС

Дисциплина «Проектирование и разработка приложений БД»

СВОЙСТВА ACID

Атомарность, Согласованность, Изоляция, Долговечность

ОП «Информационные системы», ОП «IT-медицина»

Авторы:
ст.преп. каф. ИВС Саданова Б.М.
ст.преп. каф. ИВС Олейникова А.В.

Atomicity и Consistency

Первые два свойства ACID: атомарность гарантирует целостность операции, согласованность — целостность данных

Atomicity (Атомарность)

«Всё или ничего» — транзакция либо выполняется полностью, либо не выполняется вовсе.

• Если транзакция завершается успешно (COMMIT) — все изменения сохраняются

• Если происходит сбой (ROLLBACK) — все изменения отменяются, БД возвращается к состоянию до начала транзакции

• Промежуточные состояния невидимы для других транзакций

Если сбой после шага 1 — оба изменения откатываются.

Механизмы реализации:

• WAL (Write-Ahead Logging) — журналирование

• Undo logs — откат незавершенных операций

• Savepoints — точки сохранения внутри транзакции

Consistency (Согласованность)

База данных переходит из одного согласованного состояния в другое.

• Все ограничения целостности (constraints) соблюдаются:

  — PRIMARY KEY, FOREIGN KEY

  — CHECK constraints

  — NOT NULL, UNIQUE

  — Триггеры

• Сумма балансов счетов остается постоянной

• Инварианты предметной области не нарушаются

Важно: Consistency — это свойство и приложения, и СУБД:

• СУБД обеспечивает constraints и triggers

• Приложение обеспечивает бизнес-логику (например, проверку достаточности средств)

• ACID-Consistency ≠ CAP-Consistency (это разные концепции!)

Isolation и Durability

Изоляция обеспечивает независимость транзакций, долговечность — сохранность данных при любых сбоях

Isolation (Изоляция)

Параллельно выполняющиеся транзакции не влияют друг на друга.

- Результат параллельного выполнения эквивалентен некоторому последовательному (serializable schedule)
- Транзакция работает в «пузыре» — не видит изменений других транзакций до их фиксации
- Уровни изоляции определяют степень защиты:
  - — READ UNCOMMITTED — минимальная
  - — READ COMMITTED — защита от dirty reads
  - — REPEATABLE READ — защита от non-repeatable reads
  - — SERIALIZABLE — максимальная

Механизмы реализации:

- Блокировки (2PL, Strict 2PL)
- MVCC — Multi-Version Concurrency Control
- Снимки (snapshots) базы данных
- SSI — Serializable Snapshot Isolation

Durability (Долговечность)

Зафиксированные изменения сохраняются навсегда, даже при сбоях.

- После COMMIT данные не теряются при:
  - — Отказе питания
  - — Сбое операционной системы
  - — Аварийном завершении процесса СУБД
  - — Повреждении оборудования (при репликации)

Механизмы реализации:

- WAL (Write-Ahead Logging):
  - — Сначала запись в журнал, потом в данные
  - — Журнал пишется на диск синхронно (fsync)
- Репликация:
  - — Физическая (streaming WAL)
  - — Логическая (row-level changes)
- Резервное копирование:
  - — PITR (Point-in-Time Recovery)
  - — Базовые бэкапы + WAL-архивы

Механизмы реализации ACID в СУБД

Современные СУБД используют комбинацию механизмов для обеспечения всех четырёх свойств ACID

Журналирование (WAL / Redo / Undo)

Обеспечивает: Atomicity + Durability

- WAL (Write-Ahead Log):
  - — Запись изменений в журнал ДО записи в данные
  - — Гарантирует восстановление после сбоя
- Redo Log:
  - — Повтор зафиксированных изменений
  - — Обеспечивает Durability
- Undo Log / Segments:
  - — Откат незавершенных транзакций
  - — Обеспечивает Atomicity
  - — Используется для MVCC (старые версии)
- Checkpoint:
  - — Синхронизация WAL с данными
  - — Ускорение восстановления

Блокировки (2PL / MVCC)

Обеспечивают: Isolation

- Two-Phase Locking (2PL):
  - — Growing phase: получение locks
  - — Shrinking phase: освобождение locks
  - — Strict 2PL: X-lock до commit
- MVCC (Multi-Version CC):
  - — Несколько версий строк
  - — Читатели не блокируют писателей
  - — Снимки (snapshots) на момент времени
- Next-Key Locks (InnoDB):
  - — Предотвращение phantom reads
  - — Record lock + gap lock
- SSI (PostgreSQL):
  - — Serializable Snapshot Isolation
  - — Отслеживание rw-конфликтов

Ограничения целостности (Constraints / Triggers)

Обеспечивают: Consistency

- PRIMARY KEY:
  - — Уникальность идентификаторов
- FOREIGN KEY:
  - — Ссылочная целостность
- CHECK constraints:
  - — Диапазоны значений
  - — Бизнес-правила
- NOT NULL / UNIQUE:
  - — Обязательность и уникальность
- Триггеры (Triggers):
  - — Автоматические проверки
  - — Каскадные операции
- Хранимые процедуры:
  - — Инкапсуляция бизнес-логики

ACID vs BASE: классические СУБД vs NoSQL

ACID (Классические СУБД)

- Atomicity — атомарность операций
- Consistency — согласованность данных
- Isolation — изоляция транзакций
- Durability — долговечность изменений

Характеристики:

- Строгая согласованность данных
- Надёжность и предсказуемость
- Высокая latency на запись (fsync)
- Сложность масштабирования
- Примеры: PostgreSQL, MySQL, Oracle, SQL Server

Применение:

- Финансовые системы
- Банковские операции
- Бронирование и учет
- Любые системы, где потеря данных недопустима

BASE (NoSQL / Distributed)

- Basically Available — базовая доступность
- Soft state — мягкое состояние (данные могут быть временно несогласованы)
- Eventually consistent — согласованность в конечном счете

Характеристики:

- Высокая доступность и масштабируемость
- Согласованность не гарантируется мгновенно
- Низкая latency на запись
- Простота горизонтального масштабирования
- Примеры: Cassandra, MongoDB, DynamoDB, Redis

Применение:

- Социальные сети (лайки, комментарии)
- Аналитика и big data
- Кэширование
- Системы, где доступность важнее строгой согласованности

Теорема CAP: невозможно одновременно обеспечить Consistency, Availability и Partition Tolerance. ACID выбирает C, BASE выбирает A.

Практическое применение и выводы

Реализация ACID в СУБД

- PostgreSQL:
  - — MVCC + WAL + SSI
  - — Полная поддержка ACID
  - — SERIALIZABLE через SSI
  - — PITR для восстановления
- MySQL (InnoDB):
  - — Redo Log + Undo Log
  - — MVCC + Next-Key Locks
  - — Strict 2PL для блокировок
  - — Crash recovery через redo
- Oracle:
  - — Undo Segments + Redo Logs
  - — MVCC через SCN
  - — Flashback для time-travel
  - — RAC для кластеризации
- SQL Server:
  - — Transaction Log (WAL)
  - — MVCC через snapshot isolation
  - — Row-versioning для чтения
  - — Always On Availability Groups

Ключевые выводы

- ACID — фундамент надежности транзакций в СУБД
- Atomicity — «всё или ничего» через WAL и Undo
- Consistency — переход между согласованными состояниями
- Isolation — параллельные транзакции как последовательные
- Durability — сохранность данных при любых сбоях
- Механизмы:
  - — WAL + Redo/Undo для A и D
  - — Блокировки + MVCC для I
  - — Constraints + Triggers для C
- ACID vs BASE — выбор зависит от требований:
  - — Финансы → ACID
  - — Соцсети / аналитика → BASE
- Теорема CAP: Consistency, Availability, Partition Tolerance — выбери два

Феномены аномалий параллельного выполнения

ANSI SQL-92 определяет три классических феномена + расширенные аномалии (Berenson et al., 1995)

Dirty Read (Грязное чтение)

Транзакция читает данные, измененные другой транзакцией, которая еще не зафиксировала изменения (uncommitted).

Пример:

T1: UPDATE balance=90

T2: SELECT balance → 90

T1: ROLLBACK

T2: работает с неверными данными (90 вместо 100)

Опасность: решения на основе временных, несохраненных данных.

Non-Repeatable Read (Неповторяемое чтение)

Транзакция дважды читает одни и те же данные и получает разные значения, потому что другая транзакция зафиксировала изменения между чтениями.

Пример:

T1: SELECT balance → 100

T2: UPDATE balance=90, COMMIT

T1: SELECT balance → 90

Проблема: несогласованность внутри одной транзакции, ошибки в отчетах и расчетах.

Phantom Read (Фантомное чтение)

Транзакция повторно выполняет запрос с условием и получает другой набор строк, потому что другая транзакция вставила или удалила строки, удовлетворяющие условию.

Пример:

T1: SELECT COUNT(*) WHERE price>100 → 3

T2: INSERT order(price=150), COMMIT

T1: SELECT COUNT(*) WHERE price>100 → 4

Проблема: «фантомные» строки появляются/исчезают между запросами.

Расширенные аномалии и сводная таблица

В PostgreSQL REPEATABLE READ предотвращает и Phantom Read

Расширенные аномалии (Berenson, 1995)

Lost Update (Потерянное обновление):

• Две транзакции читают одно значение, модифицируют и записывают.

• Первая запись перезаписывается второй — одно обновление теряется.

Read Skew (Чтение с перекосом):

• Транзакция читает два связанных объекта в разные моменты.

• Между чтениями другая транзакция изменяет один из объектов.

• Результат: несогласованная пара значений.

Write Skew (Запись с перекосом):

• Две транзакции читают пересекающееся множество строк.

• Каждая обновляет разные строки, основываясь на прочитанном.

• Вместе они нарушают ограничение, которое каждая проверила.

• Пример: две транзакции переводят деньги, каждая видит достаточный баланс, но в сумме баланс становится отрицательным.

Уровни изоляции и аномалии

	Dirty Read	Non-Rep. Read	Phantom Read	Serial. Anomaly
READ UNCOMMITTED	✔	✔	✔	✔
READ COMMITTED	✗	✔	✔	✔
REPEATABLE READ	✗	✗	✔*	✔
SERIALIZABLE	✗	✗	✗	✗

Ключевое правило:
Чем выше уровень изоляции — тем меньше аномалий, но ниже параллелизм и производительность.

PostgreSQL: READ UNCOMMITTED = READ COMMITTED (MVCC архитектура).

Уровни изоляции: от READ UNCOMMITTED к SERIALIZABLE

READ UNCOMMITTED

- Самый низкий уровень изоляции
- Транзакции видят НЕзафиксированные изменения других
- Допускает: Dirty Read, Non-Repeatable Read, Phantom Read
- Не использует shared locks при чтении
- Максимальная производительность, минимальная согласованность
- Практически не применяется в production
- Исключение: аналитические запросы, где точность менее важна скорости

READ COMMITTED (по умолчанию в большинстве СУБД)

- Транзакция видит только зафиксированные данные
- Предотвращает Dirty Read
- Допускает: Non-Repeatable Read, Phantom Read
- Каждый запрос видит снимок на момент начала запроса
- Хороший баланс для большинства приложений
- PostgreSQL, Oracle, SQL Server — default level

REPEATABLE READ

- Один снимок данных на ВСЮ транзакцию
- Предотвращает: Dirty Read, Non-Repeatable Read
- Допускает: Phantom Read (в ANSI SQL)
- В PostgreSQL: Phantom Read ТОЖЕ предотвращается (MVCC)
- Транзакция видит данные на момент BEGIN
- При конфликте обновления — ошибка serialization failure
- Необходим retry-логика в приложении

SERIALIZABLE (максимальная изоляция)

- Транзакции выполняются как будто последовательно
- Предотвращает ВСЕ аномалии, включая Write Skew
- В PostgreSQL: SSI (Serializable Snapshot Isolation)
  - — отслеживает rw-конфликты через SIREAD locks
  - — при обнаружении аномалии — abort одной транзакции
- Максимальная согласованность, минимальный параллелизм
- Критически важно для финансовых операций

Реализация уровней изоляции в СУБД

PostgreSQL

- READ UNCOMMITTED = READ COMMITTED (MVCC)
- READ COMMITTED — default, snapshot на запрос
- REPEATABLE READ — snapshot на транзакцию, предотвращает phantom read
- SERIALIZABLE — SSI (Serializable Snapshot Isolation)
    - Отслеживает rw-конфликты
    - При обнаружении: ERROR 40001 (serialization_failure)
    - Требует retry в приложении
- MVCC — основной механизм, не блокировки
- Нет dirty reads по архитектуре

MySQL (InnoDB)

- READ UNCOMMITTED — dirty reads возможны
- READ COMMITTED — default в Oracle-режиме
- REPEATABLE READ — default в MySQL
    - MVCC + Next-Key Locks
    - Предотвращает phantom reads через gap locks
- SERIALIZABLE — полные shared locks на SELECT
- Блокировки: record locks, gap locks, next-key locks
- MVCC + 2PL гибрид
- innodb_lock_wait_timeout

Oracle

- READ COMMITTED — default
- SERIALIZABLE — поддерживается
- READ ONLY — оптимизированный режим
- MVCC через Undo Segments
- SCN (System Change Number) для версий
- Нет REPEATABLE READ как отдельного уровня
- Flashback Query — time-travel до SCN
- Snapshot Isolation вместо true serializable
- Нет SSI — phantom reads возможны в SERIALIZABLE

Выбор уровня изоляции: баланс и выводы

Рекомендации по выбору

- READ COMMITTED — для большинства OLTP-приложений
  - Баланс согласованности и производительности
  - Подходит для web-приложений, CRM, ERP
- REPEATABLE READ — для финансовых расчетов
  - Гарантия стабильных данных внутри транзакции
  - Отчеты, агрегации, инвентаризация
- SERIALIZABLE — для критических операций
  - Переводы денег, бронирование, учет запасов
  - Требует retry-логики в приложении
- READ UNCOMMITTED — почти никогда
  - Только аналитика, где точность не критична

Ключевые выводы

- Четыре уровня изоляции ANSI SQL: READ UNCOMMITTED → READ COMMITTED → REPEATABLE READ → SERIALIZABLE
- Три классических феномена:
  - Dirty Read — чтение незафиксированных данных
  - Non-Repeatable Read — изменение данных между чтениями
  - Phantom Read — появление/исчезновение строк
- Расширенные аномалии: Lost Update, Read Skew, Write Skew
- Компромисс: высокая изоляция = низкий параллелизм
- MVCC (PostgreSQL, MySQL) vs Locking (SQL Server)
- SERIALIZABLE в PostgreSQL — это SSI, не Strict 2PL
- Правильный выбор уровня изоляции — ключ к надежности приложения

Типы блокировок: S-lock и X-lock

Блокировки — механизм ограничения доступа к данным при параллельном выполнении транзакций

Shared Lock (S-lock)

Разделяемая (совместная) блокировка для операций ЧТЕНИЯ.

- Несколько транзакций могут одновременно удерживать S-lock на один и тот же объект
- Транзакция с S-lock может только ЧИТАТЬ данные
- Запрещает модификацию данных другими транзакциями
- Обозначается: lock-S(A) или S(A)

Пример:

```
lock-S(Account_A)
read(Account_A)
unlock(Account_A)
```

Совместимость:

- S-lock + S-lock = ✔ (допустимо)
- S-lock + X-lock = ✗ (конфликт)

Exclusive Lock (X-lock)

Эксклюзивная (монопольная) блокировка для операций ЗАПИСИ.

- Только ОДНА транзакция может удерживать X-lock на объект
- Транзакция с X-lock может и ЧИТАТЬ, и ПИСАТЬ данные
- Полностью блокирует доступ других транзакций
- Обозначается: lock-X(A) или X(A)

Пример:

```
lock-X(Account_A)
read(Account_A)
write(Account_A)
unlock(Account_A)
```

Совместимость:

- X-lock + S-lock = ✗ (конфликт)
- X-lock + X-lock = ✗ (конфликт)

Двухфазный протокол 2PL

Двухфазный протокол (2PL)

Транзакция проходит через две фазы:

1. Growing Phase (Фаза роста):
   - Транзакция только ПОЛУЧАЕТ блокировки
   - Может upgrade S → X
   - Нельзя освободить ни одной блокировки
   - Заканчивается в Lock Point

2. Shrinking Phase (Фаза сжатия):
   - Транзакция только ОСВОБОЖДАЕТ блокировки
   - Может downgrade X → S
   - Нельзя запросить новые блокировки

Lock Point — момент, когда транзакция получила все необходимые блокировки. После него начинается фаза сжатия.

Гарантия: 2PL обеспечивает conflict serializability — любое расписание, полученное по 2PL, эквивалентно некоторому сериальному расписанию.

Варианты двухфазного протокола

От Basic 2PL к более строгим вариантам, устраняющим каскадные откаты и грязное чтение

Strict 2PL

- X-lock удерживается до конца транзакции
- S-lock можно освобождать в фазе сжатия
- Предотвращает грязное чтение (dirty reads)
- Обеспечивает возможность отката одной транзакции
- Тупики возможны

Пример:

lock-X(A), read(A), write(A)
lock-X(B), read(B), write(B)
COMMIT → unlock(A), unlock(B)

Rigorous 2PL

- ВСЕ блокировки (S и X) удерживаются до конца транзакции
- Самый строгий вариант
- Полностью исключает каскадные откаты
- Прерывание легко отменить — восстановить старые значения
- Максимально ограничивает конкурентность
- Тупики возможны

Пример:

lock-S(A), read(A)
lock-X(B), read(B), write(B)
COMMIT → unlock(A), unlock(B)

Conservative 2PL

- Все блокировки запрашиваются В НАЧАЛЕ транзакции
- Если не получены все — транзакция не стартует
- Полностью предотвращает тупики
- Требует знания всех данных заранее
- Может удерживать блокировки дольше, чем нужно
- На практике редко используется

Пример:

lock-X(A), lock-X(B)
read(A), write(A)
read(B), write(B)
COMMIT → unlock(A), unlock(B)

Проблемы 2PL: тупики и каскадные откаты

Тупики (Deadlocks)

Тупик возникает, когда транзакции ждут друг друга в цикле.

Классический пример:

T1: lock-X(A) → lock-X(B) → ...

T2: lock-X(B) → lock-X(A) → ...

Хронология:

- T1 блокирует A (X-lock)
- T2 блокирует B (X-lock)
- T1 пытается заблокировать B → ждет T2
- T2 пытается заблокировать A → ждет T1
- Цикл: T1 → T2 → T1 → ТУПИК!

Решения:

- Обнаружение через граф ожидания (WFG)
- Предотвращение: Wait-Die, Wound-Wait
- Таймауты (deadlock_timeout)
- Conservative 2PL — полностью исключает тупики

Каскадные откаты (Cascading Aborts)

Каскадный откат — когда откат одной транзакции вынуждает откат других.

Сценарий (Basic 2PL):

- T1: lock-X(A), write(A), unlock(A), ...
- T2: lock-S(A), read(A) ← dirty read!
- T3: lock-S(A), read(A) ← тоже dirty read!
- T1 abort → T2 и T3 должны откатиться!

Проблема: T2 могла уже зафиксировать изменения, зависящие от T1.

Решения:

- Strict 2PL — X-lock до commit (нет dirty reads)
- Rigorous 2PL — все locks до commit (максимальная защита)
- MVCC — чтение старых версий без блокировок

Strict 2PL и Rigorous 2PL гарантируют:

- Recoverable schedules
- Cascadeless schedules

Тупик (Deadlock): понятие и условия

Ситуация, когда две или более транзакций бесконечно блокируют друг друга, ожидая освобождения ресурсов

Классический пример тупика

Транзакция T1:

```
UPDATE Account_A SET balance -= 100;
UPDATE Account_B SET balance += 100;
```

Транзакция T2:

```
UPDATE Account_B SET balance -= 50;
UPDATE Account_A SET balance += 50;
```

Хронология:

- T1 блокирует Account_A (X-lock)
- T2 блокирует Account_B (X-lock)
- T1 пытается заблокировать Account_B → ждет T2
- T2 пытается заблокировать Account_A → ждет T1
- Цикл ожидания → ТУПИК!

Результат: обе транзакции заблокированы навсегда, если СУБД не вмешается.

Условия Коффмана (Coffman)

Для возникновения тупика необходимы ВСЕ четыре условия:

1. Взаимное исключение (Mutual Exclusion)

   Ресурс может быть занят только одной транзакцией

2. Удержание и ожидание (Hold and Wait)

   Транзакция удерживает ресурсы, ожидая дополнительные

3. Отсутствие вытеснения (No Preemption)

   Ресурсы нельзя принудительно забрать — только добровольно освободить

4. Циклическое ожидание (Circular Wait)

   Замкнутая цепочка транзакций, где каждая ждет ресурс, удерживаемый следующей

Вывод: для предотвращения тупика достаточно нарушить хотя бы одно условие!

Обнаружение тупика: граф ожидания

Wait-For Graph — ориентированный граф, где вершины = транзакции, ребро $T_i \rightarrow T_j$ означает «T_i ждет освобождения ресурса, удерживаемого T_j»

Построение графа ожидания:

• Для каждой транзакции, ожидающей блокировку, добавляем ребро к транзакции, удерживающей эту блокировку

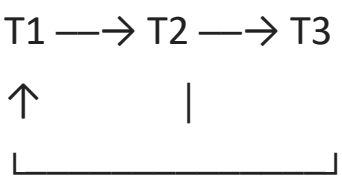
• Периодически (или при превышении таймаута ожидания) СУБД проверяет граф на наличие циклов

Алгоритм обнаружения цикла (DFS):

• Обходим граф в глубину от каждой вершины

• Если при обходе встречаем уже посещенную вершину в текущем пути — ЦИКЛ найден!

Пример графа ожидания:



Цикл: T1 → T2 → T3 → T1 → ТУПИК!

Предотвращение тупиков: Непрерывный (Wait-Die) и Прерываемый (Wound-Wait)

Оба алгоритма используют временную метку (timestamp) транзакций для предотвращения циклического ожидания

Wait-Die (Непрерывный)

Непрерывная (non-preemptive) схема:

- T_i запрашивает ресурс, удерживаемый T_j
- Если T_i СТАРШЕ (timestamp меньше) → T_i ЖДЕТ (wait)
- Если T_i МЛАДШЕ (timestamp больше) → T_i УМИРАЕТ (die/abort)

Принцип: старшие транзакции ждут младших, младшие не ждут старших.

Свойства:

- Старшие транзакции никогда не откатываются
- Младшие могут умирать многократно
- При перезапуске временная метка сохраняется
- Больше прерываний, чем в Wound-Wait

Wound-Wait (Прерывание)

Прерывная (preemptive) схема:

- T_i запрашивает ресурс, удерживаемый T_j
- Если T_i СТАРШЕ (timestamp меньше) → T_j РАНИТ (wound/abort), T_i получает ресурс
- Если T_i МЛАДШЕ (timestamp больше) → T_i ЖДЕТ (wait)

Принцип: старшие транзакции забирают ресурсы у младших.

Свойства:

- Младшие транзакции никогда не откатываются
- Старшие могут ранить младших
- Меньше прерываний, чем в Wait-Die
- Старшие транзакции получают приоритет — полезно для OLTP

Таймауты и разрешение тупиков

Выбор «жертвы» (Victim Selection)

Критерии выбора транзакции для отката:

- Минимальная работа — транзакция с наименьшим количеством log-записей, CPU time или измененных строк
- Возраст — самая молодая транзакция (меньше потерь) или самая старая (предотвращение starvation)
- Приоритет — низкоприоритетные транзакции жертвуются первыми
- Стоимость отката — транзакция с минимальной стоимостью rollback
- Количество ресурсов — транзакция, удерживающая наименьшее число блокировок

Таймауты (Timeout-based)

Простейший метод разрешения тупиков:

- Каждая транзакция имеет максимальное время ожидания блокировки
- Если время ожидания превышено — транзакция считается «подозрительной»
- СУБД проверяет, участвует ли транзакция в тупике
- Если да — транзакция откатывается (abort)

Параметры СУБД:

- PostgreSQL: deadlock_timeout = 1s (по умолчанию)
  - — при превышении запускается проверка WFG
- MySQL: innodb_lock_wait_timeout = 50s
  - — транзакция откатывается по таймауту
- SQL Server: LOCK_TIMEOUT

MVCC: Многоверсионный контроль параллелизма

Multi-Version Concurrency Control — читатели не блокируют писателей, писатели не блокируют читателей

Концепция MVCC

Вместо перезаписи строки при UPDATE система создает новую версию, сохраняя старую.

- Readers видят старую версию — не ждут завершения записи
- Writers создают новую версию — не блокируют читателей
- Каждая транзакция работает со снимком (snapshot) БД на момент старта

Метаданные версии:

- xmin — ID транзакции, создавшей версию
- xmax — ID транзакции, удалившей версию (или NULL)
- cmin/cmax — ID команд внутри транзакции

MVCC vs Strict 2PL

Strict Two-Phase Locking:

- Читатели блокируют писателей (shared locks)
- Писатели блокируют читателей (exclusive locks)
- Высокая конкуренция, тупики

MVCC:

- Чтение без блокировок — читатели не ждут
- Писатели создают версии — не конфликтуют с читателями
- Только write-write конфликты требуют разрешения
- Поддержка time-travel queries (запросы к состоянию БД в прошлом)

Результат: значительно более высокая пропускная способность при смешанных read/write нагрузках.

Снимки (Snapshots) и видимость версий

Snapshot Isolation — каждая транзакция видит согласованный снимок базы данных на момент своего старта

Структура снимка (PostgreSQL):

- xmin — нижняя граница: все транзакции с txid < xmin завершены и видимы
- xmax — верхняя граница: txid следующей транзакции, еще не стартовавшей
- xip[] — список активных (не завершенных) транзакций в момент снимка

Правила видимости версии строки:

1. Строка видима, если xmin создана транзакцией, которая:

- txid < snapshot.xmin (завершена до старта текущей транзакции) ИЛИ
- txid в snapshot.xip и уже committed (завершена, но после старта)

2. Строка НЕ видима, если:

- xmax указывает на активную или прерванную транзакцию
- xmax указывает на транзакцию, завершившуюся после старта текущей

Snapshot Flow
BEGIN → create snapshot → read only committed versions visible in snapshot → COMMIT/ROLLBACK

Уровни изоляции и сборка мусора (Garbage Collection)

Уровни изоляции в MVCC

Read Committed (по умолчанию):

• Новый снимок для КАЖДОГО SQL-запроса

• Видны изменения, зафиксированные до начала запроса

• Допускает неповторяемое чтение

Repeatable Read:

• Один снимок на ВСЮ транзакцию

• Видны только данные, зафиксированные до BEGIN

• Предотвращает неповторяемое чтение

и фантомное чтение

Сериализуемость:

• Повторяемое чтение (Repeatable Read) + SSI (Serializable Snapshot Isolation)

• Отслеживает rw-конфликты через SIREAD locks

• Обнаруживает аномалии ассимметрии записи и ассиметрии при чтении

• При обнаружении конфликта — одна транзакция прерывается

Garbage Collection (VACUUM)

Проблема: множественные версии раздувают БД

PostgreSQL VACUUM:

• Определяет «мертвые» кортежи (dead tuples) — версии, невидимые всем активным транзакциям

• Физически удаляет мертвые кортежи, освобождая место

• Обновляет Visibility Map (страницы без мертвых кортежей)

• VACUUM FULL — полная реорганизация таблицы

HOT Updates (Heap-Only Tuples):

• UPDATE без изменения индексируемых колонок

• Новая версия вставляется в ту же страницу

• Индекс НЕ обновляется — снижает нагрузку

Steam GC (HyPer, in-memory):

• Удаление устаревших версий при доступе к странице

• Очистка буферов версий при завершении транзакции