Уровни изоляции транзакций: системный подход

Уровни изоляции транзакций представляют собой одно из наиболее фундаментальных понятий в системах баз данных, встречающееся практически во всех ресурсах по базам данных и архитектуре. Однако поверхностное запоминание таблиц уровней изоляции без понимания лежащих в их основе механизмов может привести к проблемам в работе параллельных систем.

Понимание на основе фундаментальных принципов

В данном анализе используется подход, основанный на фундаментальных принципах, для понимания изоляции транзакций. Вместо запоминания уровней изоляции и их свойств, этот систематический метод исследует, как транзакции работают в современных RDBMS, изучает возникающие в результате аномалии параллелизма и выводит гарантии, предоставляемые каждым уровнем изоляции.

Этот подход доказал свою эффективность, поскольку понимание основных причин каждой аномалии естественным образом показывает, почему определенные уровни изоляции предотвращают определенные проблемы. Всестороннее понимание устраняет необходимость запоминания абстрактных таблиц сравнения.

Основное противоречие: последовательное выполнение против производительности

Идеальная изоляция

Транзакции выполняются последовательно, одна за другой

Реальная производительность

Параллельное выполнение для достижения приемлемой скорости

Компромисс

Различные уровни изоляции как баланс между корректностью и эффективностью

Гарантии транзакций

Транзакции обеспечивают гарантии ACID (атомарность, согласованность, изолированность, долговечность) для групп операций базы данных. В контексте уровней изоляции критически важным ожиданием является семантика последовательного выполнения: разработчики ожидают, что транзакции будут вести себя так, как если бы они выполнялись последовательно, без вмешательства со стороны параллельных транзакций.

Настоящее последовательное выполнение обеспечивало бы строгий порядок: не допускалось бы перекрытие транзакций, каждая транзакция завершалась бы полностью, прежде чем начиналась следующая.

01

Атомарность

Все операции выполняются или откатываются

02

Согласованность

База данных остается в корректном состоянии

03

Изолированность

Транзакции не влияют друг на друга

04

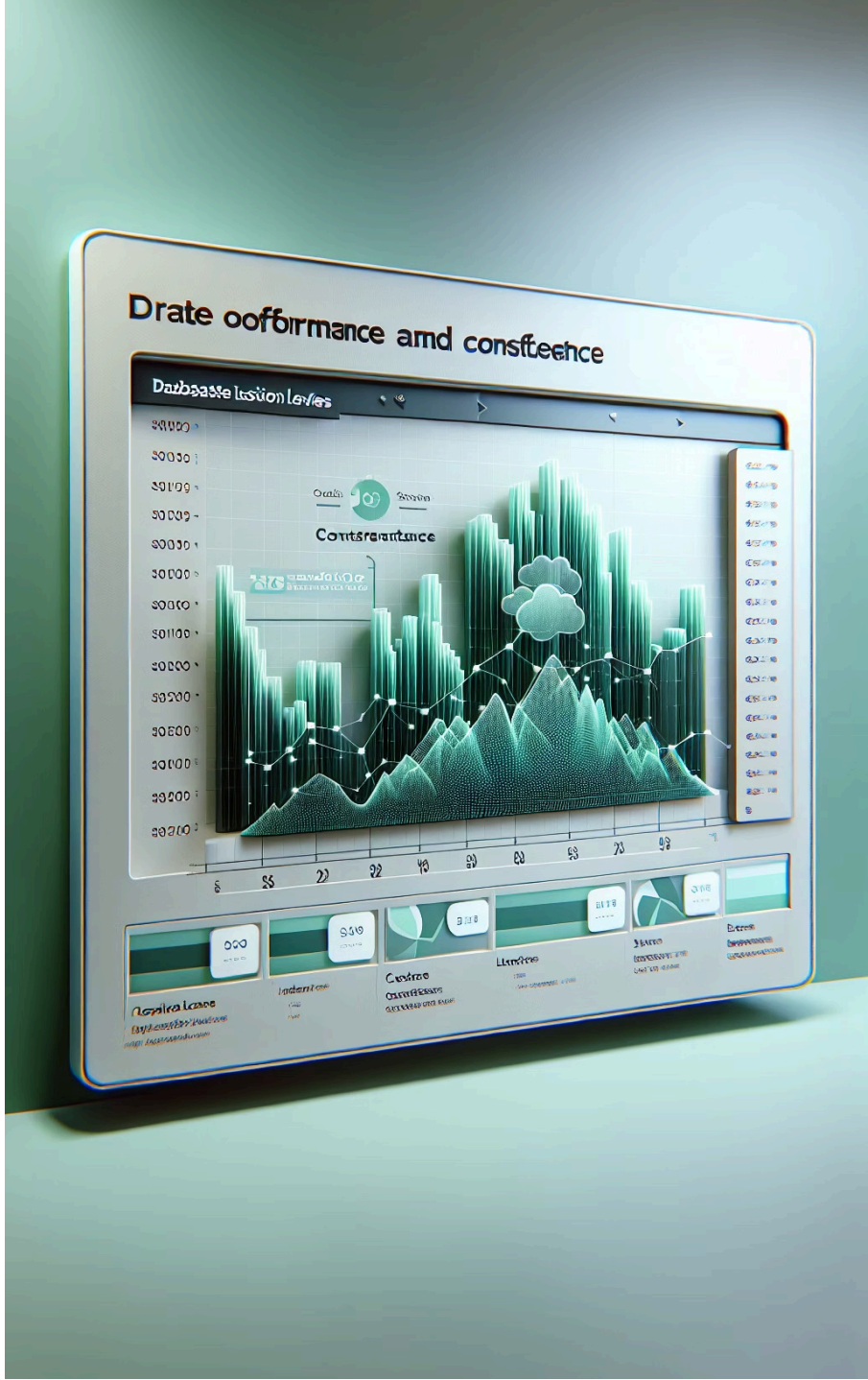
Долговечность

Зафиксированные изменения сохраняются

Реальность: компромисс между производительностью и требованиями

В то время как некоторые базы данных реализуют истинную изоляцию SERIALIZABLE (PostgreSQL, SQL Server, CockroachDB), другие идут на компромисс, реализуя более слабые гарантии под тем же названием (Oracle использует изоляцию моментальных снимков, а не истинную сериализуемость). Последовательное выполнение оказывается слишком ограничивающим для требований к производительности, особенно при длительных транзакциях.

Современные базы данных позволяют выполнять транзакции параллельно для достижения приемлемой производительности, принимая в качестве компромисса случайные аномалии согласованности. Это противоречие между производительностью и корректностью создает спектр уровней изоляции, где каждый уровень представляет собой определенный баланс между гарантиями согласованности и эффективностью выполнения.



Краткий справочник: уровни изоляции и предотвращение аномалий

Прежде чем углубиться в каждый уровень, вот таблица, к которой вы можете обращаться по мере изучения деталей:

Уровень изоляции	Грязное чтение	Неповторяемое чтение	Фантомное чтение	Потеря обновления *	Смещение записи
Незафиксированное чтение	❌	❌	❌	❌	❌
Зафиксированное чтение	✅	❌	❌	❌	❌
Повторяемое чтение	✅	✅	❌**	варьируется ***	❌
Сериализуемое	✅	✅	✅****	✅****	✅****

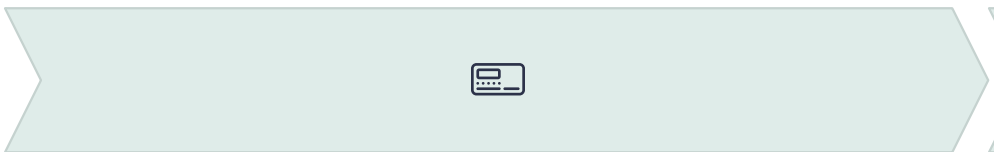
** Предотвращение потерянных обновлений зависит от реализации базы данных*

*** Некоторые базы данных, такие как PostgreSQL, предотвращают фантомные чтения даже при повторяемом чтении*

**** PostgreSQL допускает потерю обновлений при повторяемом чтении, MySQL предотвращает их*

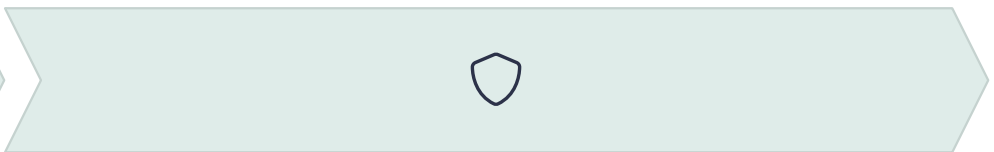
**** «SERIALIZABLE» Oracle не предотвращает сдвиг записи; некоторые базы данных NoSQL вообще не предлагают настоящую SERIALIZABLE

Прогрессивное путешествие через уровни изоляции



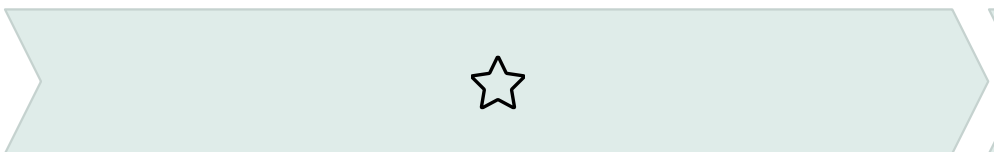
Read Uncommitted

Максимальная скорость



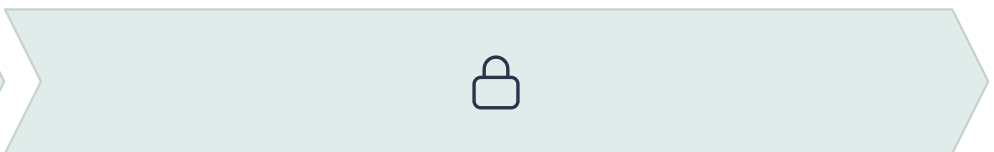
Read Committed

Базовая защита



Repeatable Read

Согласованность



Serializable

Полная изоляция

Незафиксированное чтение: максимальная скорость, минимальная безопасность

Цель: Добиться максимальной производительности, отказавшись от любых гарантий изоляции.

Решение: Очевидно, что если мы откажемся от любой изоляции, то накладные расходы на транзакции будут нулевыми. Транзакции всегда будут видеть любые изменения, внесенные другими транзакциями, независимо от того, были ли они зафиксированы или отменены, и порядок просмотра значений не гарантируется.

Компромисс: **грязное чтение** — чтение данных, которые могут быть откачены. Это означает, что вы можете принимать бизнес-решения на основе данных, которые на самом деле никогда не существовали в зафиксированном состоянии.

Реальный пример: В реальности есть много случаев, когда это приемлемо: например, некоторые метрики, которые должны быть приблизительными или изменчивыми, такие как количество лайков в социальных сетях или количество посетителей в реальном времени и т. д.

Примечания по реализации: Не все БД поддерживают эту функцию; PostgreSQL официально поддерживает ее, но на практике обрабатывает так же, как Read Committed.

Read Committed: скрытие нефиксированных изменений

Цель: решить проблему грязного чтения, сохранив хорошую производительность.

Решение: скрыть все изменения, внесенные транзакцией, до их фиксации. Это позволяет избежать «грязных» чтений, т. е. изменения, которые потенциально могут быть отменены, не видны другим транзакциям до тех пор, пока эти изменения не будут зафиксированы. Есть два способа реализовать эту гарантию:

- Исторически (как в старых версиях PostgreSQL или некоторых текущих конфигурациях MySQL/MariaDB) это достигалось путем установки и снятия блокировок чтения и записи на строки. Блокировки записи устанавливаются на строки на время транзакции, а блокировки чтения устанавливаются только на время операции чтения (чтобы гарантировать блокировку, если установлена блокировка записи).
- Большинство современных систем используют MVCC (многоверсионный контроль параллелизма). Каждая транзакция начинается со снимка базы данных на момент ее запуска, но для каждой операции снимок обновляется: используется снимок из последней зафиксированной транзакции.

Компромисс: **Неповторимое чтение** — один и тот же запрос может возвращать разные значения в рамках одной транзакции.

> **Пример:** Сценарий бронирования отеля: транзакция A считывает room_price = $100 и отображает это значение клиенту. Во время заполнения формы транзакция B обновляет цену до $150 и фиксирует изменения. Когда транзакция A завершает бронирование, она считывает обновленную цену $150. Клиент получает счет на $150 за номер, стоимость которого была указана как $100, что приводит к нарушению бизнес-логики.

Примечания по реализации: Это гораздо лучше, чем Read Uncommitted, и фактически является уровнем изоляции по умолчанию в PostgreSQL. Существует много случаев, когда вышеупомянутые Dirty Reads неприемлемы, но Non-repeatable Reads допустимы. Например, в электронной коммерции мы хотим, чтобы клиенты видели только подтвержденные продукты, или мы не хотим отправлять электронные письма на адрес, который изменяется другой транзакцией, но может быть отменен.

Повторяемое чтение: согласованные моментальные снимки в рамках транзакций

Цель: обеспечить, чтобы значения строк возвращали одно и то же значение независимо от того, какие другие транзакции могут быть подтверждены в то же время.

Решение: в этом случае проблема неповторимых чтений смягчается, поскольку в течение всей транзакции будет использоваться одно и то же значение цены номера. Совершенно логично, что такой уровень изоляции называется **повторяемым чтением**.

Опять же, это можно реализовать двумя способами:

- опять же, устаревший способ — использование блокировок: установка блокировок на чтение или запись на время транзакции. Это гарантирует, что если значение изменяется, то другие транзакции увидят его только после фиксации/отката.
- использование MVCC: транзакции сохраняют один и тот же снимок на протяжении всей транзакции, но они остаются фиксированными и не обновляются при каждой операции, как в Read Committed.

Компромиссы Repeatable Read

Компромиссы: Хотя Repeatable Read предотвращает грязные чтения и неповторимые чтения, он все же допускает несколько аномалий:

Потерянные обновления: Интересно отметить, что PostgreSQL использует чистый MVCC для этого уровня изоляции, в то время как MySQL использует комбинацию MVCC и блокировок. Поэтому PostgreSQL допускает **потерянные обновления** (нарушая стандарт), а MySQL — нет. Это один из примеров, когда граница между уровнями изоляции транзакций становится размытой, а реализации отличаются от стандарта.

> **Пример потерянного обновления:** две параллельные транзакции считывают общий баланс банковского счета в размере 1000 долларов. Транзакция A вычисляет депозит: 1000 долларов + 200 долларов = 1200 долларов. Транзакция B выполняет тот же расчет: 1000 долларов + 200 долларов = 1200 долларов. Последняя зафиксированная транзакция перезаписывает первую, в результате чего окончательный баланс составляет 1200 долларов вместо правильных 1400 долларов. Один депозит фактически потерян.

Примечание: этого не произошло бы, если бы счет был обновлен одним оператором, поскольку все операции в большинстве баз данных являются атомарными (например, UPDATE accounts SET balance = balance + 200 WHERE id = 1;), но если баланс считывается, а затем обновляется в двух операциях, то это будет проблемой.

Фантомное чтение: эта проблема связана с запросами по диапазону. Если мы запустим запрос типа SELECT * FROM accounts WHERE balance > 0 несколько раз во время транзакции, ни блокировки, ни MVCC не помогут нам получить одинаковый результат, поскольку строки, удовлетворяющие этому критерию, могут быть добавлены или удалены другими транзакциями.

Смещение записи: Это происходит, когда две транзакции читают пересекающиеся наборы данных, принимают решения на основе прочитанного, а затем записывают в непересекающиеся наборы данных, но записи нарушают ограничение, которое должно соблюдаться в обоих наборах.

> **Пример несинхронной записи:** Система дежурств требует, чтобы в любое время дежурил по крайней мере один врач. Две одновременные транзакции считывают текущее состояние (2 врача на дежурстве). На основе этой информации каждая транзакция разрешает соответствующему врачу уйти с дежурства. Результат: на дежурстве не остается ни одного врача, что нарушает бизнес-ограничение.

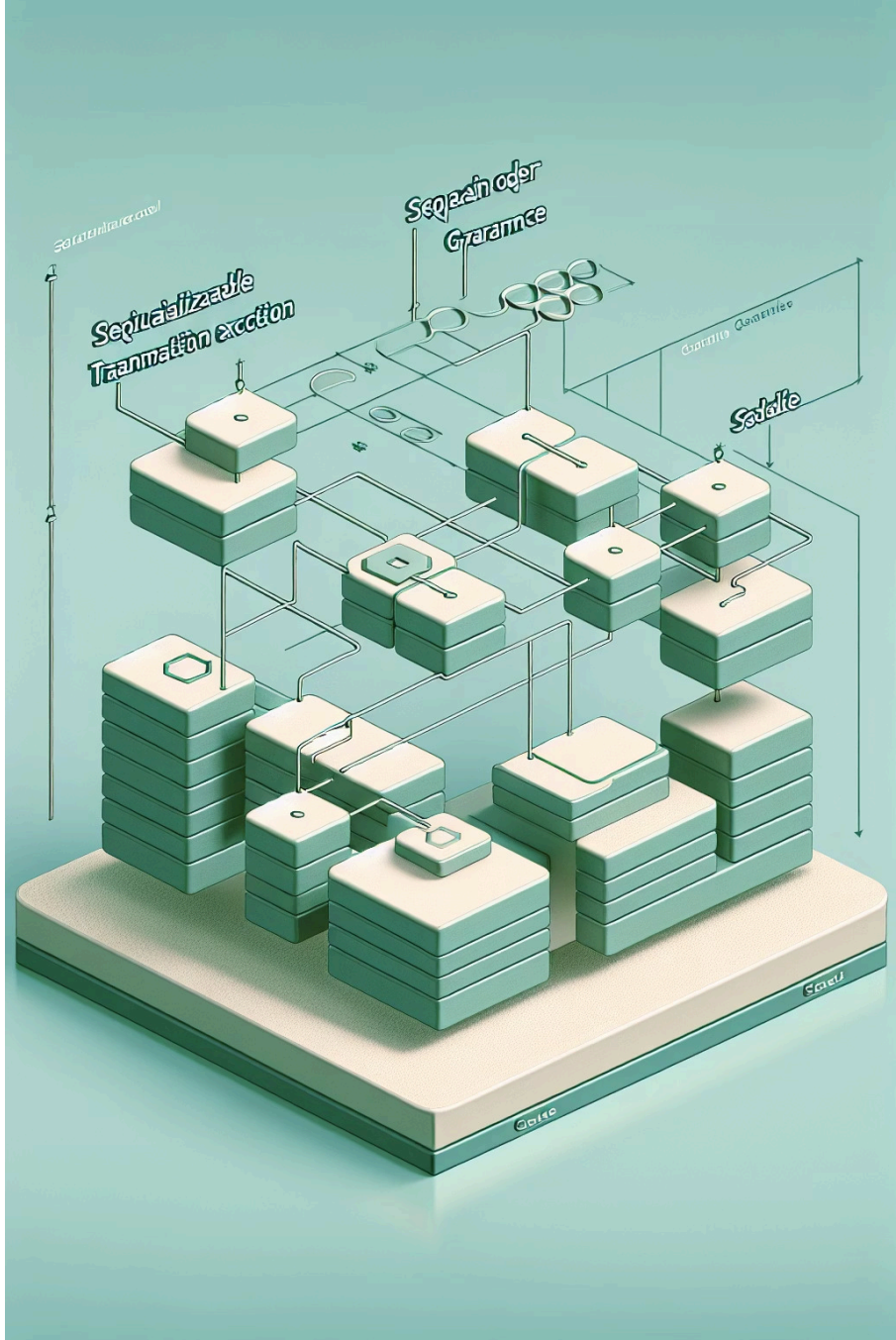
Примечания по реализации: Таким образом, решением здесь является использование блокировок (с помощью чего-то вроде SELECT FOR UPDATE для смягчения потери обновлений или блокировки другой специальной таблицы блокировок или использования консультативных блокировок для смягчения **фантомных чтений** и **разнородности записей**). В качестве альтернативы мы можем использовать самый строгий уровень изоляции, который мы рассмотрим ниже.

Serializable: окончательное решение

Цель: обеспечить выполнение транзакций так, как если бы они выполнялись одна за другой в последовательном порядке.

Решение: как обсуждалось выше, окончательным решением всех этих аномалий является обеспечение выполнения транзакций так, как если бы они выполнялись одна за другой в последовательном порядке. Именно это обещает изоляция **Serializable** — она предотвращает все аномалии, включая фантомные чтения, сдвиг записи и любые другие нарушения согласованности, которые могут возникнуть при одновременном выполнении.

Но здесь дело становится интересным и потенциально запутанным: не все базы данных, предлагающие уровень изоляции «SERIALIZABLE», на самом деле обеспечивают истинную сериализуемую изоляцию.



Истинные реализации сериализуемости

Истинные реализации сериализуемости:

Современные базы данных реализуют сериализуемую изоляцию умными способами, которые позволяют избежать снижения производительности при фактическом последовательном выполнении:

PostgreSQL (9.1+)

Использует Serializable Snapshot Isolation (SSI), оптимистичный подход, который позволяет транзакциям проходить без блокировки, но отслеживает зависимости между транзакциями. Если он обнаруживает паттерн, который может привести к аномалии сериализации, он прерывает одну из транзакций. Это чрезвычайно эффективно — часто только на 10-30% медленнее, чем Read Committed для типичных рабочих нагрузок.

SQL Server и MySQL InnoDB

Используют более традиционный подход, используя блокировку диапазона и блокировку следующего ключа соответственно. При выполнении команды SELECT * FROM accounts WHERE balance > 1000 они блокируют не только существующие строки, но и «промежутки» между строками, чтобы предотвратить появление фантомов. Это работает, но может привести к значительной блокировке.

CockroachDB и FoundationDB

Были разработаны с нуля с сериализуемым уровнем изоляции в качестве основного, используя распределенные версии оптимистичного управления параллелизмом.

Исключение Oracle

Важное примечание по реализации: Уровень изоляции SERIALIZABLE Oracle не обеспечивает истинных гарантий сериализуемости. Он реализует изоляцию моментальных снимков, которая предотвращает большинство аномалий, но остается уязвимой для сценариев Write Skew. Это различие в реализации подчеркивает важность понимания реализации уровней изоляции, специфичных для баз данных.

Компромиссы:

Serializable имеет смысл, когда:

- У вас есть сложные бизнес-инварианты, охватывающие несколько строк (как в примере с дежурными врачами)
- Правильность важнее производительности
- Вы хотите упростить рассуждения о параллельном поведении

Однако это сопряжено с некоторыми издержками:

- Снижение пропускной способности (зависит от реализации)
- Увеличение числа прерываний транзакций, требующих повторной попытки
- Возможность ложных срабатываний, когда транзакции прерываются без необходимости

Практическая структура для понимания уровней изоляции

Систематическая концептуальная структура для анализа уровней изоляции и их компромиссов:

01

Определите проблемы

Начните с понимания конкретных аномалий параллелизма

02

Сопоставьте решения

Свяжите каждый уровень изоляции с предотвращаемыми проблемами

03

Оцените компромиссы

Взвесьте производительность против гарантий согласованности

04

Выберите подходящий уровень

Примите решение на основе требований приложения

Начните с проблем, а не с уровней

1. **Грязное чтение**

   чтение данных, которые могут быть отменены

2. **Неповторяемое чтение**

   один и тот же запрос возвращает разные значения в рамках одной транзакции

3. **Фантомное чтение**

   запросы по диапазону возвращают разные наборы строк в рамках одной транзакции

4. **Потеря обновления**

   одновременные изменения перезаписывают друг друга

5. **Смещение записи**

   транзакции нарушают ограничения при записи в не связанные наборы данных

Затем сопоставьте проблемы с решениями

Чтение не зафиксированных данных

допускает все проблемы (используйте только для приблизительных данных)

Чтение зафиксированных данных

предотвращает грязное чтение (хороший вариант по умолчанию для большинства приложений)

Повторяемое чтение

предотвращает грязные + неповторяющиеся чтения (хорошо подходит для отчетов, вычислений)

Сериализуемое

предотвращает все проблемы (используйте для критически важной бизнес-логики)

Практическая структура принятия решений

Начните с Read Committed

это оптимальный вариант для большинства приложений

Перейдите на Повторяемое чтение

если вам нужны согласованные вычисления в рамках одной транзакции

Используйте Сериализуемое

только в том случае, если это требует бизнес-логика и вы можете обработать повторные попытки

Никогда не используйте Чтение без фиксации

если вам не требуется грязное чтение для повышения производительности

Вспомогательные средства для памяти

Примеры для запоминания

- Вспомните пример с бронированием отеля для неповторяемого чтения
- Вспомните пример с банковским счетом для утраченных обновлений
- Вспомните пример с дежурным врачом для искажения записи
- Помните: «больше изоляции = меньше аномалий = более низкая производительность»

Ключевое правило

> **Важное замечание:** Детали реализации, специфичные для баз данных, часто имеют приоритет над теоретическими стандартами. Повторяемое чтение в PostgreSQL допускает потерю обновлений, а реализация MySQL предотвращает их. «Serializable» в Oracle не обеспечивает истинную сериализуемость. Эти различия в реализации имеют значительные последствия для производства, выходящие за рамки теоретических спецификаций.