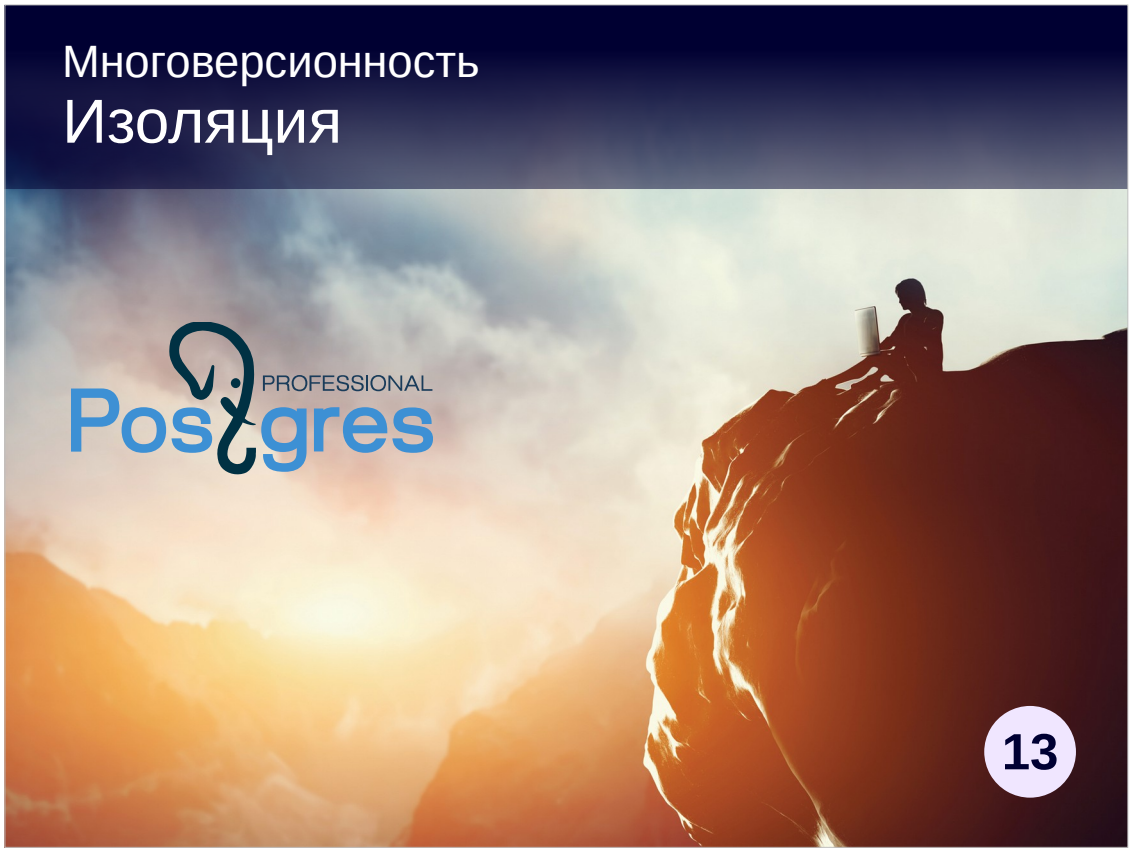

Авторы: Егор Рогов, Павел Лузанов, Илья Баштанов

Обратная связь
Отзывы, замечания и предложения направляйте по адресу:
edu@postgrespro.ru

Темы

Транзакции и их свойства

Уровни изоляции в стандарте SQL

Особенности реализации PostgreSQL

Транзакции и их свойства

Транзакция — последовательность операций,
которые переводят базу данных из одного корректного состояния в другое корректное (Согласованность)
при условии, что транзакция выполнена полностью (Атомарность)
и без помех со стороны других транзакций (Изоляция)

- атомарность: транзакция либо выполняется полностью, либо не оставляет никаких следов
- согласованность определяется ограничениями целостности и семантикой приложения
- неполная изоляция приводит к нарушению корректности (аномалиям) при конкурентном выполнении транзакций

Транзакцией называется последовательность операций, выполняемая приложением, которая переводит базу данных из одного корректного состояния в другое корректное состояние (*согласованность*) при условии, что транзакция выполнена полностью (*атомарность*) и без помех со стороны других транзакций (*изоляция*).

Корректное (согласованное) состояние определяется как ограничениями целостности, заданными на уровне базы данных, так и семантикой приложения.

Идеальная изоляция транзакций гарантирует, что на работу приложения не окажут влияния никакие другие конкурентно выполняющиеся транзакции. Однако если ослабить требования к изоляции, то транзакция, которая корректно выполнялась в базе данных в одиночестве, может выдавать некорректные результаты при наличии других транзакций — из-за того, что при выполнении операторы разных транзакций могут чередоваться. Такие некорректные ситуации называются *аномалиями*.

Реализация полной изоляции — сложная задача, сопряженная с уменьшением пропускной способности системы. На практике как правило применяется именно ослабленная изоляция, и поэтому важно понимать, какие это влечет последствия.

Четвертое свойство ACID — *долговечность* — мы рассмотрим позже, в теме про журнал предзаписи.

Read Uncommitted

Должен предотвращать аномалии

потерянные обновления:
транзакция может перезаписать изменения других транзакций, зафиксированные после ее начала

Не реализован в PostgreSQL

работает как Read Committed

Стандарт SQL определяет четыре уровня изоляции, описывая аномалии, которые допускаются при конкурентном выполнении транзакций на этом уровне.

Самый слабый уровень — Read Uncommitted, который (как следует из названия) позволяет увидеть даже незафиксированные данные.

Согласно стандарту на этом уровне должна предотвращаться единственная аномалия:

- **Потерянные обновления** (lost updates). Если после начала транзакции T1 другая транзакция T2 изменила и зафиксировала строки, транзакция T1 может перезаписать эти изменения.

Уровень Read Uncommitted вводился для минимизации накладных расходов, но PostgreSQL может без потери эффективности работать на более строгом уровне. Поэтому в PostgreSQL такой уровень не реализован и говорить о нем мы больше не будем.

Read Committed

Должен предотвращать аномалии

потерянные обновления

грязное чтение:
запрос видит незафиксированные изменения других транзакций

Реализация в PostgreSQL

используется по умолчанию

Каждый следующий уровень изоляции строже, чем предыдущий. Поэтому уровень Read Committed должен предотвращать не только потерянные обновления, но и еще одну аномалию:

- **«Грязное» чтение** (dirty read). Транзакция T1 может читать строки, измененные, но еще не зафиксированные транзакцией T2. Отмена изменений (ROLLBACK) в T2 приведет к тому, что T1 прочитает данные, которых никогда не существовало.

Уровень Read Committed допускает множество других аномалий. Разработчик должен всегда помнить о возможных проблемах и при необходимости вручную использовать блокировки.

В PostgreSQL (как и во многих других СУБД) именно этот уровень изоляции используется по умолчанию — как компромисс между строгостью изоляции и эффективностью.

Заметим, что в PostgreSQL на уровне Read Committed можно потерять изменения — такой пример будет показан в демонстрации.

Управление уровнем изоляции

Для демонстрации мы будем использовать отдельную базу данных для каждой темы.

```
=> CREATE DATABASE arch_isolation;
```

```
CREATE DATABASE
```

```
=> \c arch_isolation
```

```
You are now connected to database "arch_isolation" as user "student".
```

Уровни изоляции посмотрим на примере с таблицей, представляющей состояние светофора:

```
=> CREATE TABLE lights(
  id integer GENERATED ALWAYS AS IDENTITY,
  lamp text,
  state text
);
```

```
CREATE TABLE
```

Это будет пешеходный светофор с двумя лампочками:

```
=> INSERT INTO lights(lamp,state) VALUES
    ('red', 'on'), ('green', 'off');
```

```
INSERT 0 2
```

```
=> SELECT * FROM lights ORDER BY id;
```

```
 id | lamp  | state
----+-------+-------
  1 | red   | on
  2 | green | off
(2 rows)
```

Один способ установить уровень изоляции — команда SET TRANSACTION, выполненная в начале транзакции:

```
=> BEGIN;
```

```
BEGIN
```

```
=> SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
```

```
SET
```

Проверить текущий уровень можно, посмотрев значение параметра:

```
=> SHOW transaction_isolation;
```

```
 transaction_isolation
-----------------------
 read committed
(1 row)
```

```
=> COMMIT;
```

```
COMMIT
```

А можно указать уровень прямо в команде BEGIN:

```
=> BEGIN ISOLATION LEVEL READ COMMITTED;
```

```
BEGIN
```

```
=> COMMIT;
```

```
COMMIT
```

По умолчанию используется уровень Read Committed:

```
=> SHOW default_transaction_isolation;
```

```
 default_transaction_isolation
-------------------------------
 read committed
(1 row)
```

Так что если этот параметр не менялся, можно не указывать уровень явно.

```
=> BEGIN;
```

```
BEGIN
```

```
=> SHOW transaction_isolation;
```

```
 transaction_isolation
-----------------------
 read committed
(1 row)
```

```
=> COMMIT;
```

```
COMMIT
```

Read Committed и грязное чтение

Попробуем прочитать «грязные» данные. В первой транзакции гасим красный свет:

```
=> BEGIN;
```

```
BEGIN
```

```
=> UPDATE lights SET state = 'off' WHERE lamp = 'red';
```

```
UPDATE 1
```

Начнем второй сеанс.

```
student$ psql arch_isolation
```

В нем откроем еще одну транзакцию с тем же уровнем Read Committed.

```
=> BEGIN;
```

```
BEGIN
```

```
=> SELECT * FROM lights ORDER BY id;
```

```
 id | lamp  | state
----+-------+-------
  1 | red   | on
  2 | green | off
(2 rows)
```

Вторая транзакция не видит незафиксированных изменений.

Отменим изменение.

```
=> ROLLBACK;
```

```
ROLLBACK
```

```
=> ROLLBACK;
```

```
ROLLBACK
```

Read Committed и чтение зафиксированных изменений

Мы проверили, что транзакция не видит незафиксированных изменений. Посмотрим, что будет при фиксации.

```
=> BEGIN;
```

```
BEGIN
```

```
=> UPDATE lights SET state = 'off' WHERE lamp = 'red';
```

```
UPDATE 1
```

```
=> BEGIN;
```

```
BEGIN
```

```
=> SELECT * FROM lights ORDER BY id;
```

```
 id | lamp  | state
----+-------+-------
  1 | red   | on
  2 | green | off
(2 rows)
```

Пока не видно.

```
=> COMMIT;
```

```
COMMIT
```

А теперь?

```
=> SELECT * FROM lights ORDER BY id;
```

```
 id | lamp  | state
----+-------+-------
  1 | red   | off
  2 | green | off
(2 rows)
```

```
=> COMMIT;
```

```
COMMIT
```

Итак, в режиме Read Committed операторы одной транзакции видят зафиксированные изменения других транзакций.

Заметьте, что при этом один и тот же запрос в одной и той же транзакции может выдавать разные результаты.

Можно ли увидеть изменения, зафиксированные в процессе выполнения одного оператора? Проверим.

Сейчас все лампочки погашены. Запустим долгий запрос и, пока он работает, включим свет во втором сеансе:

```
=> SELECT *, pg_sleep(2) FROM lights ORDER BY id;
```

```
=> UPDATE lights SET state = 'on';
```

```
UPDATE 2
```

```
 id | lamp  | state | pg_sleep
----+-------+-------+----------
  1 | red   | off   |
  2 | green | off   |
(2 rows)
```

Итак, если во время выполнения оператора другая транзакция успела зафиксировать изменения, то они не будут видны. Оператор видит данные в таком состоянии, в котором они находились на момент начала его выполнения.

Однако если в запросе вызывается функция с категорией изменчивости volatile, выполняющая собственный запрос, то такой запрос внутри функции будет возвращать данные, не согласованные с данными основного запроса.

```
=> CREATE FUNCTION get_state(lamp text) RETURNS text AS $$
BEGIN
    RETURN (SELECT l.state FROM lights l WHERE l.lamp = get_state.lamp);
END;
$$ LANGUAGE plpgsql VOLATILE;
```

```
CREATE FUNCTION
```

Повторим эксперимент, но теперь запрос будет использовать функцию:

```
=> SELECT *, get_state(lamp), pg_sleep(2) FROM lights ORDER BY id;
```

```
=> UPDATE lights SET state = 'off';
```

```
UPDATE 2
```

```
 id | lamp  | state | get_state | pg_sleep
----+-------+-------+-----------+----------
  1 | red   | on    | on        |
  2 | green | on    | off       |
(2 rows)
```

Правильный вариант — объявить функцию с категорией изменчивости stable.

```
=> ALTER FUNCTION get_state STABLE;
```

```
ALTER FUNCTION
```

```
=> SELECT *, get_state(lamp), pg_sleep(2) FROM lights ORDER BY id;
```

```
=> UPDATE lights SET state = 'on';
```

```
  UPDATE 2
```

```
 id | lamp  | state | get_state | pg_sleep
----+-------+-------+-----------+----------
  1 | red   | off   | off       |
  2 | green | off   | off       |
(2 rows)
```

Вывод: внимательно следите за категорией изменчивости функций на уровне изоляции Read Committed. Со значениями «по умолчанию» легко получить несогласованные данные.

Read Committed и потерянные изменения

Что происходит при попытке изменения одной и той же строки двумя транзакциями? Сейчас все лампочки включены.

```
=> BEGIN;
```

```
BEGIN
```

```
=> UPDATE lights SET state = 'off' WHERE lamp = 'red';
```

```
UPDATE 1
```

```
  => BEGIN;
```

```
  BEGIN
```

```
  => UPDATE lights
  SET state = CASE WHEN state = 'on' THEN 'off' ELSE 'on' END
  WHERE lamp = 'red';
```

Вторая транзакция ждет завершения первой.

```
=> COMMIT;
```

```
COMMIT
```

```
  UPDATE 1
```

```
  => COMMIT;
```

```
  COMMIT
```

Но какой будет результат? Вторая транзакция «щелкает переключателем», и результат зависит от значения, от которого она будет отталкиваться.

```
=> SELECT * FROM lights ORDER BY id;
```

```
 id | lamp  | state
----+-------+-------
  1 | red   | on
  2 | green | on
(2 rows)
```

С одной стороны, команда во второй транзакции не должна видеть изменений, сделанных после начала ее выполнения. Но с другой — она не должна потерять изменения, зафиксированные другими транзакциями. Поэтому после снятия блокировки она перечитывает строку, которую пытается обновить.

В итоге, первая транзакция выключает свет, а вторая снова включает его.

Такой результат интуитивно кажется правильным, но достигается он за счет того, что транзакция может увидеть несогласованные данные: часть — на один момент времени, часть — на другой.

Но если изменение выполняется не в одной команде SQL, то обновление будет потеряно. Повторим тот же пример.

```
=> BEGIN;
```

```
BEGIN
```

```
=> UPDATE lights SET state = 'off' WHERE lamp = 'red';
```

```
UPDATE 1
```

```
  => BEGIN;
```

```
  BEGIN
```

Сначала читаем значение и запоминаем его на клиенте:

```
  => SELECT state AS old_state FROM lights WHERE lamp = 'red' \gset
```

```
=> \echo :old_state
```

```
on
```

А затем обновляем на сервере:

```
=> UPDATE lights
SET state = CASE WHEN :'old_state' = 'on' THEN 'off' ELSE 'on' END
WHERE lamp = 'red';
```

Вторая транзакция ждет завершения первой.

```
=> COMMIT;
```

```
COMMIT
```

```
UPDATE 1
```

```
=> COMMIT;
```

```
COMMIT
```

Какой результат будет на этот раз?

```
=> SELECT * FROM lights ORDER BY id;
```

```
 id | lamp  | state
----+-------+-------
  1 | red   | off
  2 | green | on
(2 rows)
```

В этом случае вторая транзакция перезаписала свои изменения «поверх» изменений первой транзакции. На уровне Read Committed сервер не может это предотвратить, поскольку команда UPDATE фактически содержит предопределенную константу.

Repeatable Read

Должен предотвращать аномалии

- потерянные обновления
- грязное чтение
- неповторяющееся чтение:
  повторное чтение строки вернет другое значение,
  если оно было изменено и зафиксировано другой транзакцией

Реализация в PostgreSQL

- также не допускает фантомное чтение:
  повторный запрос по одному и тому же условию вернет иную выборку,
  если другая транзакция добавила и зафиксировала новые строки,
  удовлетворяющие этому условию
- для предотвращения аномалий транзакции могут обрываться

Согласно стандарту, в дополнение к потерянным обновлениям и грязным чтениям уровень изоляции Repeatable Read должен предотвращать аномалию, которую мы видели, когда рассматривали уровень Read Committed:

- **Неповторяющееся чтение** (non-repeatable read). После того, как транзакция T1 прочитала строку, транзакция T2 изменила или удалила эту строку и зафиксировала изменения (COMMIT). При повторном чтении этой же строки транзакция T1 видит, что строка изменена или удалена.

Реализация PostgreSQL более строгая, чем того требует стандарт, и предотвращает также аномалию

- **Фантомное чтение** (phantom read). Транзакция T1 прочитала набор строк по некоторому условию. Затем транзакция T2 добавила строки, также удовлетворяющие этому условию. Если транзакция T1 повторит запрос, она получит другую выборку строк.

На самом деле существуют и другие аномалии (не отраженные в стандарте), которые допускаются в PostgreSQL на уровне Repeatable Read.

От уровня Read Committed этот уровень изоляции отличается и тем, что на нем транзакция может быть оборвана, чтобы не допустить аномалию (такую транзакцию следует повторить). На уровне Read Committed этого не происходит никогда — если стоит выбор между корректностью и эффективностью, предпочтение всегда отдается эффективности.

Repeatable Read и неповторяющееся чтение

Убедимся в отсутствии аномалии неповторяющегося чтения.

```
=> BEGIN ISOLATION LEVEL REPEATABLE READ;
```

```
BEGIN
```

```
=> SELECT * FROM lights WHERE lamp = 'red';
```

```
 id | lamp | state
----+------+-------
  1 | red  | off
(1 row)
```

```
=> UPDATE lights SET state = 'on' WHERE lamp = 'red' RETURNING *;
```

```
 id | lamp | state
----+------+-------
  1 | red  | on
(1 row)

UPDATE 1
```

Какое значение получит первая транзакция?

```
=> SELECT * FROM lights WHERE lamp = 'red';
```

```
 id | lamp | state
----+------+-------
  1 | red  | off
(1 row)
```

Повторное чтение измененной строки возвращает первоначальное значение.

```
=> COMMIT;
```

```
COMMIT
```

Repeatable Read и фантомное чтение

Фантомные записи также не должны быть видны. Проверим это.

```
=> BEGIN ISOLATION LEVEL REPEATABLE READ;
```

```
BEGIN
```

```
=> SELECT * FROM lights WHERE state = 'off';
```

```
 id | lamp | state
----+------+-------
(0 rows)
```

```
=> INSERT INTO lights(lamp,state) VALUES ('yellow', 'off')
    RETURNING *;
```

```
 id |  lamp  | state
----+--------+-------
  3 | yellow | off
(1 row)

INSERT 0 1
```

```
=> SELECT * FROM lights WHERE state = 'off';
```

```
 id | lamp | state
----+------+-------
(0 rows)
```

Действительно, транзакция не видит добавленной записи, удовлетворяющей первоначальному условию. Уровень изоляции Repeatable Read в PostgreSQL более строгий, чем того требует стандарт SQL.

```
=> COMMIT;
```

```
COMMIT
```

Уберем желтую лампочку.

```
=> DELETE FROM lights WHERE lamp = 'yellow';
```

```
DELETE 1
```

Repeatable Read и потерянные изменения

Необходимость все время видеть ровно те же данные, что и в самом начале, не позволяет перечитывать измененную строку в случае обновления.

Воспроизведем тот же пример, который мы видели на уровне изоляции Read Committed. Сейчас все лампочки включены.

```
=> BEGIN ISOLATION LEVEL REPEATABLE READ;
```

```
BEGIN
```

```
=> UPDATE lights SET state = 'off' WHERE lamp = 'red';
```

```
UPDATE 1
```

```
=> BEGIN ISOLATION LEVEL REPEATABLE READ;
```

```
BEGIN
```

```
=> SELECT state AS old_state FROM lights WHERE lamp = 'red' \gset
```

```
=> \echo :old_state
```

```
on
```

```
=> UPDATE lights
SET state = CASE WHEN :'old_state' = 'on' THEN 'off' ELSE 'on' END
WHERE lamp = 'red';
```

Вторая транзакция ждет завершения первой.

```
=> COMMIT;
```

```
COMMIT
```

```
ERROR:  could not serialize access due to concurrent update
```

Во второй транзакции получаем ошибку. Строка была изменена; обновить неактуальную версию невозможно (это будет потерянным изменением, что недопустимо), а увидеть актуальную версию тоже невозможно (это нарушило бы изоляцию).

```
=> ROLLBACK;
```

```
ROLLBACK
```

Таким образом, потерянное обновление на уровне Repeatable Read не допускается.

Repeatable Read и другие аномалии

Можно ли быть уверенным в том, что следующая команда включит все лампочки?

```
UPDATE lights SET state = 'on' WHERE state = 'off';
```

Можно ли быть уверенным в том, что следующая команда выключит все лампочки?

```
UPDATE lights SET state = 'off' WHERE state = 'on';
```

Если одна транзакция включает лампочки, а другая — выключает, в каком состоянии останется таблица после одновременного выполнения двух этих транзакций?

Проверим. Начальное состояние:

```
=> SELECT * FROM lights ORDER BY id;
```

```
 id | lamp  | state
----+-------+-------
  1 | red   | off
  2 | green | on
(2 rows)
```

```
=> BEGIN ISOLATION LEVEL REPEATABLE READ;
```

```
BEGIN
```

```
=> UPDATE lights SET state = 'on' WHERE state = 'off';
```

```
UPDATE 1
```

Первая транзакция включила красную лампочку, и теперь все огни горят.

```
=> SELECT * FROM lights ORDER BY id;
```

```
 id | lamp  | state
----+-------+-------
  1 | red   | on
  2 | green | on
(2 rows)
```

```
=> BEGIN ISOLATION LEVEL REPEATABLE READ;
```

```
BEGIN
```

Вторая транзакция не видит этих, еще не зафиксированных, изменений и считает, что красный свет выключен. Поэтому она выключает зеленый и для нее все огни погашены.

```
=> UPDATE lights SET state = 'off' WHERE state = 'on';
```

```
UPDATE 1
```

```
=> SELECT * FROM lights ORDER BY id;
```

```
 id | lamp  | state
----+-------+-------
  1 | red   | off
  2 | green | off
(2 rows)
```

Теперь обе транзакции фиксируют свои изменения...

```
=> COMMIT;
```

```
COMMIT
```

```
=> COMMIT;
```

```
COMMIT
```

И оказывается, что...

```
=> SELECT * FROM lights ORDER BY id;
```

```
 id | lamp  | state
----+-------+-------
  1 | red   | on
  2 | green | off
(2 rows)
```

...одна лампочка включена, а другая — выключена.

Это пример аномалии конкурентного доступа — несогласованная запись (write skew), — которая возможна, даже если нет грязного, неповторяющегося и фантомного чтений.

Serializable

Должен предотвращать любые аномалии

- результат конкурентного выполнения совпадает с результатом выполнения транзакций в какой-либо последовательности

Реализация в PostgreSQL

- для предотвращения аномалий транзакции могут обрываться
- для правильной работы уровень Serializable должен использоваться всеми транзакциями
- уровень не работает на репликах

Полное отсутствие каких бы то ни было аномалий достигается, если команды, выполняемые в конкурентно работающих транзакциях, приводят к такому же результату, какой получился бы в случае последовательного — одна завершилась, следующая началась — выполнения этих транзакций (при каком-нибудь порядке выполнения).

В PostgreSQL этот уровень реализован и работает, если все транзакции в системе используют уровень Serializable.

Следует учитывать, что реализация имеет определенные ограничения:

- до версии 12 запросы не распараллеливались;
- запросы не могут выполняться на репликах;
- транзакции могут обрываться чаще, чем это действительно необходимо (таким образом, страдает эффективность, поскольку оборванные транзакции приходится выполнять повторно).

Serializable

На этом уровне транзакции могут положиться на то, что на их работу никто не повлияет.

```
=> BEGIN ISOLATION LEVEL SERIALIZABLE;
```

```
BEGIN
```

```
=> UPDATE lights SET state = 'on' WHERE state = 'off';
```

```
UPDATE 1
```

Первая транзакция включила зеленую лампочку, все огни горят.

```
=> SELECT * FROM lights ORDER BY id;
```

```
 id | lamp  | state
----+-------+-------
  1 | red   | on
  2 | green | on
(2 rows)
```

Вторая транзакция не видит зафиксированных изменений: для нее красный свет еще горит, и она выключает его.

```
=> BEGIN ISOLATION LEVEL SERIALIZABLE;
```

```
BEGIN
```

```
=> UPDATE lights SET state = 'off' WHERE state = 'on';
```

```
UPDATE 1
```

```
=> SELECT * FROM lights ORDER BY id;
```

```
 id | lamp  | state
----+-------+-------
  1 | red   | off
  2 | green | off
(2 rows)
```

Теперь обе транзакции фиксируют свои изменения...

```
=> COMMIT;
```

```
COMMIT
```

```
=> COMMIT;
```

```
ERROR:  could not serialize access due to read/write dependencies among transactions
DETAIL:  Reason code: Canceled on identification as a pivot, during commit attempt.
HINT:  The transaction might succeed if retried.
```

И вторая транзакция получает ошибку.

Действительно, при последовательном выполнении транзакций допустимо два результата:

- если сначала выполняется первая транзакция, а потом вторая, то все лампочки будут погашены;
- если сначала выполняется вторая транзакция, а потом первая, то все лампочки будут включены.

«Промежуточный» вариант получить невозможно, поэтому выполнение одной из транзакций завершается ошибкой.

Важный момент: чтобы уровень Serializable работал корректно, на этом уровне должны работать все транзакции. Если смешивать транзакции разных уровней, фактически Serializable будет вести себя, как Repeatable Read.

Итоги: стандарт SQL

	потерянные обновления	грязное чтение	неповторя-ющееся чтение	фантомное чтение	другие аномалии
Read Uncommitted	—	да	да	да	да
Read Committed	—	—	да	да	да
Repeatable Read	—	—	—	да	да
Serializable	—	—	—	—	—

Подытожим информацию.

В стандарте SQL определены четыре уровня изоляции транзакций: Read Uncommitted, Read Committed, Repeatable Read, Serializable (по умолчанию).

Таблица показывает соответствие между уровнями изоляции и теми аномалиями, которые они могут допускать.

Отметим, что для обеспечения уровня изоляции Serializable недостаточно исключить три аномалии, определенные в стандарте.

Итоги: PostgreSQL

	потерянные обновления	грязное чтение	неповторя-ющееся чтение	фантомное чтение	другие аномалии
Read Uncommitted					
Read Committed	*	—	да	да	да
Repeatable Read	—	—	—	—	да
Serializable	—	—	—	—	—

* предотвращается не всегда

Реальные СУБД, и PostgreSQL в том числе, не реализуют буквально требования стандарта (которые были сформулированы в те времена, когда вопросы изоляции еще не были изучены теоретиками). Названия уровней изоляции сохраняются, но реализация их строже, чем это требует стандарт.

В PostgreSQL используется *изоляция на основе снимков данных* (Snapshot Isolation) в сочетании с *многоверсионностью*. Это определяет следующие свойства:

- Грязное чтение не допускается вообще, причем без ущерба для производительности.
- Уровень Read Committed используется по умолчанию и применяется в большинстве систем. Зачастую он требует ручной установки блокировок, чтобы обеспечить корректность.
- Уровень Repeatable Read не допускает не только неповторяющееся чтение, но и фантомное чтение (хотя и не обеспечивает полную изоляцию). Этот уровень удобен для отчетов, поскольку позволяет нескольким запросам видеть согласованные данные, а только читающие транзакции никогда не обрываются.
- Уровень Serializable также полностью реализован, но имеет определенные ограничения.

https://postgrespro.ru/docs/postgresql/13/transaction-iso

Практика

1. Проверьте, что на уровне изоляции Read Committed не предотвращается аномалия фантомного чтения.
2. Убедитесь, что команда DROP TABLE транзакционна.
3. Начните транзакцию с уровнем изоляции Repeatable Read (и пока не выполняйте в ней никаких команд). В другом сеансе удалите строку и зафиксируйте изменения. Видна ли строка в открытой транзакции?
   Что изменится, если в начале транзакции выполнить запрос, но не обращаться в нем ни к одной таблице?
4. Начните транзакцию Repeatable Read и выполните какой-нибудь запрос. В другом сеансе создайте таблицу. Видно ли в первой транзакции описание таблицы в системном каталоге? Можно ли в ней прочитать строки таблицы?

1. Действуйте так же, как было показано в демонстрации.

1. Фантомное чтение на уровне Read Committed

```
=> CREATE DATABASE mvcc_isolation;
```

```
CREATE DATABASE
```

```
=> \c mvcc_isolation
```

```
You are now connected to database "mvcc_isolation" as user "student".
```

Пустая таблица:

```
=> CREATE TABLE foo(id integer);
```

```
CREATE TABLE
```

Во втором сеансе начнем транзакцию Read Committed и выполним запрос.

```
=> \c mvcc_isolation
```

```
You are now connected to database "mvcc_isolation" as user "student".
```

```
=> BEGIN;
```

```
BEGIN
```

```
=> SELECT * FROM foo;
```

```
 id
----
(0 rows)
```

Вставим новую строку в таблицу (изменения фиксируются).

```
=> INSERT INTO foo VALUES (1);
```

```
INSERT 0 1
```

```
=> SELECT * FROM foo;
```

```
 id
----
  1
(1 row)
```

Транзакция Read Committed видит добавленную строку.

Отличие от аномалии неповторяющегося чтения (при которой транзакция видит зафиксированные изменения в уже существовавших строках) состоит в том, что здесь видны строки, которых не было раньше.

```
=> COMMIT;
```

```
COMMIT
```

2. Транзакционность DDL

Откроем транзакцию, удалим таблицу foo, после чего транзакцию откатим.

```
=> BEGIN;
```

```
BEGIN
```

```
=> DROP TABLE foo;
```

```
DROP TABLE
```

```
=> ROLLBACK;
```

```
ROLLBACK
```

Таблица по-прежнему существует:

```
=> SELECT * FROM foo;
```

```
 id
----
  1
(1 row)
```

3. Момент, на который видны данные

Открываем транзакцию Repeatable Read:

```
=> BEGIN ISOLATION LEVEL REPEATABLE READ;
```

```
BEGIN
```

В другом сеансе удаляем строку (изменение фиксируется):

```
=> DELETE FROM foo;
```

```
DELETE 1
```

Проверяем содержимое таблицы в открытой транзакции:

```
=> SELECT * FROM foo;
```

```
 id
----
(0 rows)
```

Почему строка не видна? Дело в том, что в транзакции Repeatable Read (и Serializable) данные видны на момент начала первой команды, а не на момент выполнения оператора BEGIN.

Завершим транзакцию, вернем строчку и повторим эксперимент.

```
=> COMMIT;
```

```
COMMIT
```

```
=> INSERT INTO foo VALUES (1);
```

```
INSERT 0 1
```

Еще раз начнем транзакцию Repeatable Read и выполним в ней запрос, который не обращается к таблицам.

```
=> BEGIN ISOLATION LEVEL REPEATABLE READ;
```

```
BEGIN
```

```
=> SELECT 2*2;
```

```
 ?column?
----------
        4
(1 row)
```

В другом сеансе удалим строку.

```
=> DELETE FROM foo;
```

```
DELETE 1
```

На этот раз строка в первом сеансе видна:

```
=> SELECT * FROM foo;
```

```
 id
----
  1
(1 row)
```

Таким образом, даже команда, не обращающаяся к таблицам, определяет момент, на который видны данные.

```
=> COMMIT;
```

```
COMMIT
```

4. Изоляция и системный каталог

В первом сеансе начнем транзакцию Repeatable Read и выполним запрос.

```
=> BEGIN ISOLATION LEVEL REPEATABLE READ;
```

```
BEGIN
```

```
=> SELECT 1;
```

```
 ?column?
----------
        1
(1 row)
```

Во втором сеансе создадим таблицу .

```
  => CREATE TABLE bar AS SELECT now();
```

```
  SELECT 1
```

В первом сеансе пробуем обратиться к созданной таблице.

```
=> SELECT * FROM bar;
```

```
 now
-----
(0 rows)
```

Определение таблицы становится доступным сразу же независимо от уровня изоляции транзакции. То же касается и других изменений системного каталога: например, созданное ограничение целостности начинает действовать немедленно.

Но строки только что созданной таблицы подчиняются обычным правилам изоляции. Они будут видны только в новой транзакции.

```
=> COMMIT;
```

```
COMMIT
```

```
=> SELECT * FROM bar;
```

```
              now
-------------------------------
 2023-01-19 14:35:46.123586+03
(1 row)
```