# MySQL

БЛОКИРОВКИ И ТРАНЗАКЦИИ

# СОДЕРЖАНИЕ

# БЛОКИРОВКИ

## ВВЕДЕНИЕ

- В предыдущих разделах мы имели дело с небольшими таблицами, количество строк в которых практически всегда было меньше двух десятков. Кроме того, мы не делали запросы слишком часто и не использовали разные сессии для отправки этих запросов. Но надо понимать, что в реальной жизни существуют таблицы, в которых хранятся миллионы строк, и в рамках этих таблиц миллионы пользователей каждую миллисекунду производят множество операций. Можно догадаться, что рано или поздно такая база данных столкнется с проблемой одновременного изменения и чтения одних и тех же данных со стороны разных пользователей - то есть, с проблемой т.н. конкурентного доступа к данным.

- Представим, что мы разработали гигантскую бухгалтерскую систему, которая обслуживает компании по всему миру. В определенный момент бухгалтеры из нескольких стран начинают одновременно сдавать отчеты, которые будут характеризовать общую прибыль (или убыток) некой компании. Общий баланс считается следующим образом - из результирующей таблицы берется общий баланс на данный момент, к нему прибавляется сумма конкретного бухгалтера и обновленный баланс вставляется обратно (т.е. обновляется). Однако допустим, что параллельно эти же действия будет делать второй бухгалтер, который успеет начал раньше первого и закончил раньше первого (но закончил после того, как первый начал). Если так произойдет и мы оставим таблицы базы данных в их обычном виде, то первый бухгалтер просто перезапишет данные второго, и в результате мы получим в балансе неправильные данные.

- Как же решить проблему конкурентного доступа данных? Есть несколько решений, но сначала мы рассмотрим классический вариант - применение блокировок. Основной их смысл состоит в том, что некая сессия захватывает доступ к определенным видам взаимодействия с необходимыми ей таблицами и производит свои операции, пока остальные сессии просто ждут. После того, как сессия закончила свои операции, она явно снимает блокировку доступа, что позволяет другим сессиям наконец получить доступ к ранее заблокированным данным.

- В MySQL существуют два типа блокировок по тем операциям, которые разрешены в контексте этих операций:

  - Первый тип называется разделяемой блокировкой (**SHARED LOCK**) или же блокировкой на чтение. **SHARED LOCK** означает, что абсолютно все сессии (как та, которая захватила эту блокировку, так и те сессии, которые ждут окончания блокировки) могут читать данные из блокируемых таблиц или столбцов, но также ни одна сессия не может в этих таблицах и столбцах данные менять (удалять, обновлять, вставлять) - этого не может делать даже сессия - хозяйка блокировки.

  - Второй тип называется исключительной блокировкой (**EXCLUSIVE LOCK**) или же блокировкой на запись. Эта блокировка разрешает сессии - хозяйке как читать, так и менять те данные, которые блокируются. В свою очередь другие сессии не могут ни читать, ни менять заблокированные данные - они просто будут ожидать снятия блокировки.

- Блокировки на чтение и блокировки на запись имеют разный приоритет. Это означает, то после того, как появилась возможность захватить неделимый доступ к неким данным, и на это претендуют блокировки с разными типами, MySQL не будет производить сложные вычисления и определять, какая из них должны быть первой. Сначала будут последовательно исполнены те сессии, которые просили блокировку на запись (**EXCLUSIVE LOCK**), а только потом настанет очередь блокировок на чтение (**SHARED LOCK**). Это поведение можно изменить - если запустить MySQL сервер с параметром ***--low-priority-updates***, то первый приоритет получать блокировки на чтение.

- Кроме типизации по разрешенным операциям, блокировки в MySQL можно типизировать по тем сущностям, к которым блокировки применяются - таблицам, всей базе данных, некому текстовому ключу (т.н. пользовательская блокировка), индексам, следующего ключа и т.д. Мы не будем рассматривать их все, а попробуем разобраться только с первыми тремя, т.к. именно они в основном применяются на практике. Однако сначала создадим две таблицы и заполним их данными, чтобы объяснения были как можно более наглядными. Таблицы будут эмулировать следующую ситуацию - у нас есть банк, в котором есть счета (таблица accounts), на которых лежат деньги пользователей

(для простоты будем считать, что все счета в этом банке в одной и той же валюте - евро). Средства могут переводиться со счета на счет, но этот перевод должен фиксироваться с помощью платежей (таблица payments), которые содержат информацию о том, из какого и в какой счет были перенаправлены деньги и сколько этих денег было переведено.

```
CREATE TABLE accounts (
    id INT UNSIGNED NOT NULL AUTO_INCREMENT PRIMARY KEY,
    name VARCHAR(255) NOT NULL,
    total DECIMAL(30,2) NOT NULL
);

CREATE TABLE payments (
    id INT UNSIGNED NOT NULL AUTO_INCREMENT PRIMARY KEY,
    from_account_id INT UNSIGNED NOT NULL,
    to_account_id INT UNSIGNED NOT NULL,
    payment_sum DECIMAL(30,2) NOT NULL,
    FOREIGN KEY(from_account_id) REFERENCES accounts(id),
    FOREIGN KEY(to_account_id) REFERENCES accounts(id)
);

-- заполним данными только таблицу accounts
INSERT INTO accounts
VALUES
(1, 'John Smith', 10000),
(2, 'Mary Sue', 20000),
(3, 'Michael Adams', 30000);
```

Таблица accounts:

| id | name | total |
|---|---|---|
| 1 | John Smith | 10000 |
| 2 | Mary Sue | 20000 |
| 3 | Michael Adams | 30000 |

## БЛОКИРОВКА ТАБЛИЦ

- Блокировка таблиц осуществляется при помощи ключевой конструкции **LOCK TABLES**, после которой следуют названия таблицы или нескольких таблиц через запятую - притом после названия каждой таблицы следует указать тип блокировки - на чтение (**READ**) или на запись (**WRITE**).

- После того, как таблица была заблокирована, в контексте сессии - хозяйки блокировки можно производить необходимые операции, но следует помнить одно важное условие - внутри блокировки можно работать только с теми таблицами, на которые эта блокировка была наложена. В противном случае запрос будет отменен, а нам вернется ошибка.

- По завершении работы сессия - хозяйка блокировки должна снять блокировку при помощи конструкции **UNLOCK TABLES**. Также блокировка будет снята в том случае, если произойдет разрыв сессии. Что случится, если блокировка будет осуществляться слишком долго? Теоретически другие сессии должны ждать столько секунд, сколько прописано в параметре @@SESSION.lock_wait_timeout. Проверить его значение можно следующим образом:

```
SELECT @@SESSION.lock_wait_timeout;
```

| @@SESSION.lock_wait_timeout |
| --- |
| 50 |

Если нас не устраивает указанное в параметре количество секунд, мы может его изменить - поставим 100 секунд (если по прошествии этих секунд блокировка не будет получена, то будет возвращена ошибка Lock wait timeout exceeded):

```
SET @@SESSION.lock_wait_timeout = 100;
```

- Еще один важный момент - блокировка касается не только сессий, но и запросов, которые осуществляют события (events) - события также будут ждать окончание блокировки и только потом осуществят свои операции с данными.

- Теперь перейдем к практике - предположим, что два счета (владельцы John Smith и Michaels Adams) одновременно отправляют третьему счету (владелец Mary Sue) денежные переводы на 2000 и 3000 евро. Чтобы обезопасить передачу данных, осуществим каждый из двух переводов с помощью блокировки таблиц - (первым блокировку получит John Smith): Перевод от счета, которым владеет John Smith (id = 1) на счет, которым владеет Mary Sue (id = 2):

1 сессия:

```
LOCK TABLES accounts WRITE, payments WRITE;
-- вычитаем сумму у отправителя
UPDATE accounts SET total = total - 2000 WHERE id = 1;
-- записываем информацию о платеже
INSERT INTO payments (from_account_id, to_account_id,
payment_sum) VALUES (1, 2, 2000);
-- прибавляем сумму получателю
UPDATE accounts SET total = total + 2000 WHERE id = 2;
```

Перевод от счета Michaels Adams (id = 3) на счет, которым владеет Mary Sue (id = 2). Этот перевод начался одновременно с переводом, который делал John Smith, но он будет ждать, т.к. John Smith первым получил блокировку.

2 сессия:

```
LOCK TABLES accounts WRITE, payments WRITE;
-- вычитаем сумму у отправителя
UPDATE accounts SET total = total - 3000 WHERE id = 3;
-- записываем информацию о платеже
INSERT INTO payments (from_account_id, to_account_id,
payment_sum) VALUES (3, 2, 3000);
-- прибавляем сумму получателю
UPDATE accounts SET total = total + 3000 WHERE id = 2;
```

Освободим блокировки и дадим выполнится запросу 2 сессии, затем убедимся, что новые данные были успешно занесены в таблицы accounts и payments.

1 сессия:

```
UNLOCK TABLES;
```

2 сессия:

```
UNLOCK TABLES;
```

Таблица accounts:

| id | name | total |
|---|---|---|
| 1 | John Smith | 8000 |
| 2 | Mary Sue | 25000 |
| 3 | Michael Adams | 27000 |

Таблица payments:

| id | from_account_id | to_account_id | payment_sum |
|---|---|---|---|
| 1 | 1 | 2 | 2000.00 |
| 2 | 3 | 2 | 3000.00 |

- Теперь же применим блокировку на чтение (**READ**). Блокировать будем таблицу accounts, которую будет просматривать (читать) Michael Adams (он первым получит блокировку), одновременно свой счет будет просматривать John Smith и одновременно менять свое имя Mary Sue (на Mary Jane).

  Получение данных владельцем счета Michael Adams с применением блокировки на чтение:

  1 сессия:

```
LOCK TABLES accounts READ;
SELECT name, total FROM accounts WHERE id = 3;
```

  Получение данных владельцем счета John Smith с применением блокировки на чтение (выполнится сразу, т.к. блокировка на чтение позволяет читать из заблокированных данных):

  2 сессия:

```
LOCK TABLES accounts READ;
SELECT name, total FROM accounts WHERE id = 1;
```

  Обновление данных владельцем счета Mary Sue с применением блокировки на запись (будет ждать, пока чтение закончит Michael Adams и John Smith):

  3 сессия:

```
LOCK TABLES accounts WRITE;
UPDATE accounts SET name = 'Mary Jane' WHERE id = 2;
```

  Освободим все блокировки, тем самым позволив выполниться обновлению, а затем посмотрим изменения в таблице accounts:

  1 сессия:

```
UNLOCK TABLES;
```

2 сессия:

```
UNLOCK TABLES;
```

3 сессия:

```
UNLOCK TABLES;
```

| id | name | total |
|---|---|---|
| 1 | John Smith | 8000 |
| 2 | Mary Jane | 25000 |
| 3 | Michael Adams | 27000 |

- Кстати, если в предыдущем пункте поменять запросы от имени John Smith и Mary Sue местами, то John Smith не сможет прочитать данные сразу, т.к. в очереди он будет после Mary Sue, которая осуществляет блокировку на запись (как мы уже говорили, эта блокировка блокирует как запись, так и чтение для других сессий). Есть еще один случай - мы оставляем первый запрос без изменения, потом попытаемся изменить имя на Mary Jane, но без взятия блокировки, а после этого тоже без блокировки запросим имя John Smith. В этом случае John Smith получит информацию о себе без ожидания, но запрос на изменения все равно будет ждать окончания первой блокировки.

- Проверим утверждение, что при блокировке на чтение нельзя изменять данные в таблице. Попробуем обнулить сумму в первом платеже, но в качестве блокировки таблицы укажем **READ**:

```
LOCK TABLES payments READ;
UPDATE payments SET payment_sum = 0 WHERE id = 1;
UNLOCK TABLES;
```

В ответ мы получим ошибку, но не совсем ту, которую ожидали:

```
ERROR 1100 (HY000): Table 'accounts' was not locked with LOCK TABLES
```

Какое отношение имеет таблица accounts к нашему запросу, если мы работаем только с таблицей payments? Дело в том, что payments включает в себя два столбца *from_account_id* и

*to_account_id*, которые как раз имеют ограничение внешнего ключа по отношению к столбцу id таблицы accounts. При блокировке какой-либо таблицы все остальные таблицы, связанные с изначальной таблицей ограничениями внешних ключей, также неявно проверяются на все виды взаимодействия. Таким образом MySQL проверяет не только целевую таблицу, а и все связанные с ней через внешние ключи таблицы - даже если изменения в блокированной таблице не могут привести к изменениям в связанной таблице. Если бы внешних ключей не было, мы бы получили ошибку:

```
ERROR 1099 (HY000): Table 'payments' was locked with a READ lock and can't be updated.
```

## ГЛОБАЛЬНАЯ БЛОКИРОВКА

- Если нам необходимо заблокировать все таблицы во всех базах данных, то мы должны выполнить следующую команду:

```
FLUSH TABLES WITH READ LOCK;
```

- Эта команда запретит менять данные во всех таблицах, но тем не менее позволит читать из них (это можно понять из ключевого слова **READ** внутри команды).

- Данная блокировка полезна в том случае, если мы хотим сделать глобальную резервную копию, но не хотим, чтобы в процессе копирования в таблицах хоть что-нибудь менялось. Мы выполняем глобальную блокировку, затем просто копируем интересующие нас файлы, а затем быстро снимаем блокировку.

- Снять глобальную блокировку можно командой **UNLOCK TABLES**.

## ПОЛЬЗОВАТЕЛЬСКАЯ БЛОКИРОВКА

- Табличные и глобальные блокировки определены на уровне самой базы данных - мы уже могли увидеть, каким образом осуществляются проверки всех таблиц, как блокировки выстраиваются в очередь согласно определенному порядку и т.д. Однако в MySQL есть возможность определить свою личную - пользовательскую блокировку. Она не будет блокировать базы данных, таблицы или что-то другое - все можно будет свободно

читать и модифицировать. Единственное, что будет блокироваться - другие попытки использовать блокировку с таким же текстовым ключом.

- Для создания пользовательской блокировки необходимо сразу же после команды **SELECT** написать функцию **GET_LOCK**, которая принимает два параметра - текстовый ключ, и то количество секунд, которое следует ждать получение этой блокировки:

```
SELECT GET_LOCK('text-key', 10);
```

Если одновременно не существует пользовательских блокировок с таким же текстовым ключом (другие ключи не будут нас блокировать) либо они будут сняты быстрее, чем через 10 секунд, то данная конструкция вернет единицу:

| GET_LOCK('text-key', 10) |
| --- |
| 1 |

Если с нашим запросом на получение блокировки уже будет существовать блокировка с таким же ключом и по прошествии 10 секунд она не будет снята, то в ответ мы получим ноль:

| GET_LOCK('text-key', 10) |
| --- |
| 0 |

- Снять пользовательскую блокировку можно при помощи функции **RELEASE_LOCK**, которая в качестве параметра принимает уже упомянутый текстовый ключ. Также не надо забывать ставить перед функцией команду **SELECT**. Если все пройдет успешно (т.е. до этого мы действительно брали эту блокировку), то данная конструкция вернет нам единицу.

```
SELECT RELEASE_LOCK('text-key');
```

- Далее попытаемся разобраться с практическим применением пользовательской группировки. Например, у нас есть приложение, с помощью которого мы можем изменить данные клиента в таблице accounts. Мы хотим быть уверены, что в тот момент, когда мы производим изменения, никто другой не может

менять этого же самого клиента. Мы не говорим о том, что надо заблокировать таблицу accounts, мы хотим, чтобы нам был предоставлен исключительный доступ к модификации одной записи, связанной со счетом, который мы собрались модифицировать. Для решения этой проблемы, мы договаривается с командой разработки, что все запросы на модификацию таблицы accounts должны начинаться с пользовательской группировки, ключом которой должен быть в следующем формате: **НАЗВАНИЕ_ТАБЛИЦЫ-ИДЕНТИФИКАТОР_МОДИФИЦИРУЕМОГО_ПОЛЬЗОВАТЕЛЯ**
Сразу скажем - никто не запрещает нам менять таблицу accounts с помощью блокировки с другим ключом или вообще без блокировки - это все происходить только на уровне договоренности и не более того.

- Приведем конкретный пример, который объяснялся в предыдущем примере - померяем имя у Mary Jane (с id = 2) обратно на Mary Sue с помощью пользовательской блокировки:

```
-- применим CONCAT, чтобы склеить 'accounts-' и 2
SET @accounts_update_key = CONCAT('accounts-', 2);
-- предположим, что попытка блокировки сразу вернула единицу
SELECT GET_LOCK(@accounts_update_key, 30);
UPDATE accounts SET name = 'Mary Sue' WHERE id = 2;
-- снятие блокировки
SELECT RELEASE_LOCK(@accounts_update_key);
```

До того времени, как мы не снимем блокировку, любая попытка получить блокировку с таким же ключом accounts-2, будет возвращать ноль, что означает, что работа со строкой, содержащий идентификатор 2, еще не закончена. Ниже приведено состояние таблицы accounts после произведенных изменений:

| **id** | **name** | **total** |
|---|---|---|
| 1 | John Smith | 8000 |
| 2 | Mary Sue | 25000 |
| 3 | Michael Adams | 27000 |

# ТРАНЗАКЦИИ

## ВВЕДЕНИЕ

- Все виды блокировок безусловно могут помочь нам достигнуть нашу цель - решить проблему конкурентного доступа к данным. Однако это достается нам очень дорогой ценой. В случае с глобальной блокировкой или блокировкой таблиц мы по сути останавливаем корректную работу базы данных, выделяем приоритетное соединение (сессию), а остальные бесцельно простаивают. Пользовательская блокировка действительно позволяет ограничится блокировкой только одной строки, но, как мы уже видели, она не дает никаких особых гарантий и работает только в том случае, если все пользователи базы данных соблюдают договоренности. Кроме того, ни одна из вышеперечисленных блокировок не решает проблему неполного изменения данных. Например, если нам надо сделать 4 запроса, но после 2 запросов база данных на некоторое время перестала работать, то данные после восстановления работы базы данных окажутся в неком половинчатом, неконсистентном состоянии (например, деньги спишутся с одного счета, но не попадут на другой).

- Для решения проблем, перечисленных в предыдущем пункте в MySQL существует механизм транзакций. **Транзакция** — процесс, группа операций, которая может быть выполнена либо полностью успешно, соблюдая целостность данных и независимо от параллельно идущих других транзакций, либо не выполнена вообще и тогда она не должна произвести никакого эффекта.

- Сразу скажем, что транзакции работают только для таблиц на движке InnoDB. При работе с другими движками (например, MyISAM) можно применять конструкции, которые связаны с транзакциями, но это не даст никаких реальных результатов. Более того, это может привести к катастрофическим результатам, если какие-то данные были необратимо изменены или удалены. Внутри транзакции их можно вернуть обратно только в рамках таблиц на движке InnoDB.

- Транзакции в базах данных характеризуются 4 свойствами, которые широко известны под аббревиатурой **ACID** по первым буквам названий каждого из четырех свойств на английском языке (Atomicity, Consistency, Isolation, Durability):

- **Atomicity (атомарность):** определяет, что транзакция является наименьшим, неделимым блоком шагов алгоритма. Говоря другими словами, любые части (подоперации) транзакции либо выполняются все одновременно, либо не выполняется ни одной такой части. Поскольку в реальности всё же возникает некоторая последовательность выполнения команд внутри транзакции, вводится понятие «отката» (rollback), при котором результаты всех до сих пор произведённых действий возвращаются в исходное состояние.

- **Consistency (согласованность):** по окончанию транзакция оставляет данные в непротиворечивом состоянии. Скажем, если поле в базе данных описано как имеющее только уникальные значения строк, то ни при каком исходе транзакции дубликатов никакой строки появится не может.

- **Isolation (изоляция):** конкурирующие, параллельно текущие во времени транзакции не могут пересекаться на одних и тех же ресурсах. Для обеспечения изоляции вводятся, к примеру, специальные замки на измененных ресурсах, запрещающие другим транзакциям эти ресурсы менять до окончания поменявшей транзакции.

- **Durability (долговечность):** независимо от проблем на нижних уровнях (к примеру, обесточивание системы или сбои в оборудовании) изменения, сделанные успешно завершенной транзакцией, останутся сохраненными после возвращения системы в работу.

- Из всего вышеперечисленного можно понять, что транзакции дают нам то, что давали блокировки, только не в контексте всей базы данных или всей таблицы, а только для конкретных данных (строк таблицы). Причем это все происходит не на уровне простой договоренности, а с помощью конкретного механизма, который просто не позволит сделать что-то неправильно. Помимо блокирования данных мы также могли заметить, что замечательный механизм транзакций следит за целостностью данных - т.е. данные нельзя занести наполовину. Они либо вставляются все, либо не будет вставлено вообще ничего.

- Далее рассмотрим синтаксис создания транзакций. Чуть ниже представлена общая схема, а под ней расписан каждый шаг:

```
START TRANSACTION | BEGIN [WORK]
    [характеристики_транзакции];

характеристики_транзакции: {
    WITH CONSISTENT SNAPSHOT
  | READ WRITE
  | READ ONLY
}

-- сами запросы и прочая необходимая логика

COMMIT [WORK] [AND [NO] CHAIN] [[NO] RELEASE];
ROLLBACK [WORK] [AND [NO] CHAIN] [[NO] RELEASE];
```

Все начинается с ключевой конструкции **START TRANSACTION** или с конструкции **BEGIN WORK** (притом слово **WORK** не является обязательным). Эти команды практически идентичны, но после **BEGIN WORK** нельзя ставить какую-либо из необязательных характеристик, которые перечислены ниже:

- **WITH CONSISTENT SNAPSHOT** - состояние базы данных для транзакции будет фиксироваться не на момент первой выборки, а сразу (т.е. мы будем видеть и читать только то состояние таблиц, которые были в момент начала транзакции - даже если будут какие изменения со стороны других сессий). Работает только с уровнем изоляции **REPEATABLE READ** (об уровнях изоляции мы поговорим немного позже).

- **READ WRITE** - во время транзакции будет можно как делать выборки из таблиц, так и модифицировать данные в этих таблицах (по сути, это поведение по умолчанию).

- **READ ONLY** - во время транзакции из таблиц можно будет только читать, любые виды модификаций будут запрещены, попытка их сделать вызовет ошибку:

```
ERROR 1792 (25006): Cannot execute statement in a READ ONLY transaction.
```

Завершается транзакция либо ключевым словом **COMMIT**, либо ключевым словом **ROLLBACK**. При использовании **COMMIT** все изменения, которые мы совершили во время транзакции, будут зафиксированы в реальных таблицах. Если же мы применим **ROLLBACK**, то произойдет откат всех изменений, произведенных

во время транзакции до момента, который был сразу же до начала этой транзакции (перед **START TRANSACTION**). После **COMMIT** и **ROLLBACK** можно (но это не является обязательным условием) поставить две следующие команды:

- **AND CHAIN** - сразу по окончании транзакции начинается новая транзакция, которая будет иметь ту же характеристику (**READ WRITE** или **READ ONLY**), которая имела и предыдущая транзакция.

- **RELEASE** - после окончания транзакции сессия будет немедленно перезагружена. Это значит, что пропадут все объявленные переменные и подготовленные запросы.

## АВТОМАТИЧЕСКИЙ КОММИТ ПО УМОЛЧАНИЮ (INNODB)

- Перед тем, как мы перейдем к применению транзакций на практике, обратим внимание внимание на тот факт, что в MySQL по умолчанию каждый отдельный запрос в таблицу InnoDB является транзакционным. Это значит, что после каждого обновления в InnoDB СУБД ставит COMMIT и фиксирует изменения. За это поведение отвечает настройка **@@autocommit**:

```
SELECT @@autocommit;
```

| @@autocommit |
| --- |
| 1 |

- Если мы хотим переопределить это поведение конкретно для нашей сессии, то следует выполнить нижележащую команду:

```
SET autocommit = 0;
```

- Кстати, если мы хотим переопределить поведение для всех сессий и для всех последующих подключений, то следует сделать так:

```
SET GLOBAL autocommit = 0;
```

Теперь попытаемся вставить в таблицу accounts новую запись:

```
-- выключим автокоммит для нашей сессии
SET autocommit = 0;

INSERT INTO accounts (name, total) VALUES ('Chong Li',
35000);
```

Если мы просто закончим сессию или прервем соединение, а затем вернемся вновь, то обнаружим, что счета с именем Chong Li у нас не будет, т.к. мы явно не зафиксировали наши изменения. Если мы все же хотим занести счет Chong Li в таблицу, то надо произвести следующие операции:

```
-- опять выключим автокоммит для нашей сессии
SET autocommit = 0;

INSERT INTO accounts (name, total) VALUES ('Chong Li',
35000);

-- явно фиксируем изменения
COMMIT;
```

В этом случае наша таблица обновится и мы увидим новую запись и во всех последующих подключениях (заметим, что Chong Li получил id = 5, а не 4 - дело в том, что автоинкремент не подвержен влиянию откатов транзакций - если уже был увеличен хоть в каком-то виде, то сам он уже не откатится - такова его особенность):

| id | name | total |
|---|---|---|
| 1 | John Smith | 8000 |
| 2 | Mary Sue | 25000 |
| 3 | Michael Adams | 27000 |
| 5 | Chong Li | 35000 |

- Вернем в первоначальное состояние значение автокомита - чтобы дальнейшие операции с СУБД не делали нам сюрпризов:

```
SET GLOBAL autocommit = 0;
```

## ПРИМЕНЕНИЕ ТРАНЗАКЦИЙ НА ПРАКТИКЕ

- Пришла пора поработать с транзакциями напрямую. Но перед тем, как непосредственно начать делать запросы, отметим, что все примеры, которые представлены в этой подглаве, действительно будут работать при любом уровне изоляции транзакции. Однако могут быть другие случаи, которые мы подробно рассмотрим в подглаве об этих уровнях изоляции. А пока применим наши новые знания, чтобы помочь перечислить деньги со счета John Smith (5000 евро) на счет Mary Sue и одновременно со счета Chong Li (3000 евро) на счет Michael Adams. Не будем коммитить каждую транзакцию сразу, чтобы посмотреть на то, как они будут взаимодействовать.

  1 сессия (John Smith c id = 1 -> Mary Sue c id = 2):

```
START TRANSACTION;

-- вычитаем сумму у отправителя
UPDATE accounts SET total = total - 5000 WHERE id = 1;
-- записываем информацию о платеже
INSERT INTO payments (
    from_account_id, to_account_id, payment_sum
) VALUES (
    1, 2, 5000
);

-- прибавляем сумму получателю
UPDATE accounts SET total = total + 5000 WHERE id = 2;
```

  2 сессия (Chong Li c id = 5 -> Michael Adams c id = 3):

```
START TRANSACTION;

-- вычитаем сумму у отправителя
UPDATE accounts SET total = total - 3000 WHERE id = 5;
-- записываем информацию о платеже
INSERT INTO payments (
    from_account_id, to_account_id, payment_sum
) VALUES (
    5, 3, 3000
);

-- прибавляем сумму получателю
UPDATE accounts SET total = total + 3000 WHERE id = 3;
```

  В отличие от использования блокировки таблицы не будут заблокированы - действительно, запись и обновление

происходили в одних и тех же таблицах, однако модификации не затрагивали одни и те же строки - поэтому вторая транзакция не ждала другую. Теперь зафиксируем данные для обеих сессий и убедимся, что таблицы находятся в нормальном состоянии:

1 сессия:

```
COMMIT;
```

2 сессия:

```
COMMIT;
```

Все должно выполниться успешно, и наши таблицы должны выглядеть следующим образом:

Таблица accounts:

| id | name | total |
|---|---|---|
| 1 | John Smith | 3000 |
| 2 | Mary Sue | 30000 |
| 3 | Michael Adams | 30000 |
| 5 | Chong Li | 32000 |

Таблица payments:

| id | from_account_id | to_account_id | payment_sum |
|---|---|---|---|
| 1 | 1 | 2 | 2000.00 |
| 2 | 3 | 2 | 3000.00 |
| 3 | 1 | 2 | 5000.00 |
| 4 | 5 | 3 | 3000.00 |

- Далее попытаемся проанализировать тот случай, если в транзакции мы пытаемся как-то изменить одни и те же данные. Для этого будем перечислять деньги на счет одному и тому же человеку - Michael Adams и Chong Li попытаются одновременно отправить John Smith по 5000 евро.

1 сессия (Michael Adams с id = 3 -> John Smith с id = 1):

```
START TRANSACTION;
```

```
-- вычитаем сумму у отправителя
UPDATE accounts SET total = total - 5000 WHERE id = 3;
-- записываем информацию о платеже
INSERT INTO payments (
    from_account_id, to_account_id, payment_sum
) VALUES (
    3, 1, 5000
);

-- прибавляем сумму получателю
UPDATE accounts SET total = total + 5000 WHERE id = 1;
```

2 сессия (Chong Li с id = 5 -> John Smith с id = 1):

```
START TRANSACTION;
-- вычитаем сумму у отправителя
UPDATE accounts SET total = total - 5000 WHERE id = 5;
-- записываем информацию о платеже
INSERT INTO payments (
    from_account_id, to_account_id, payment_sum
) VALUES (
    5, 1, 5000
);
-- прибавляем сумму получателю
UPDATE accounts SET total = total + 5000 WHERE id = 1;
```

Вот теперь вторая сессия будет ждать первую чтобы предотвратить взаимную перезапись, т.к. предпринята попытка обновить одни и те же данные - строку с информацией о счета владельца John Smith. Ожидание второй сессии будет до тех пор, пока первая сессии не сделает **COMMIT** или **ROLLBACK** (в нашем случае **COMMIT**). Сделаем же это:

1 сессия:

```
COMMIT;
```

2 сессия:

```
-- здесь 2 сессии уже будет позволено продолжить
COMMIT;
```

Таблица accounts:

| id | name | total |
|---|---|---|
| 1 | John Smith | 13000 |

| 2 | Mary Sue | 30000 |
|---|---|---|
| 3 | Michael Adams | 25000 |
| 5 | Chong Li | 27000 |

Таблица payments:

| **id** | **from_account_id** | **to_account_id** | **payment_sum** |
|---|---|---|---|
| 1 | 1 | 2 | 2000.00 |
| 2 | 3 | 2 | 3000.00 |
| 3 | 1 | 2 | 5000.00 |
| 4 | 5 | 3 | 3000.00 |
| 5 | 3 | 1 | 5000.00 |
| 6 | 5 | 1 | 5000.00 |

- Теперь попробуем применить транзакции, чтобы вернуть назад некоторые данные, которые (казалось бы) были потеряны безвозвратно. Предположим, что внутри транзакции мы по ошибке удалили все строки из таблицы *payments*. В настоящих банках подобные данные считаются информацией строгой отчетности, которую могут запросить государственные службы и прочие важные инстанции. Потеря таких данных может даже привести к краху банка. Однако транзакции могут спасти нас с помощью ключевого слова **ROLLBACK**:

```
START TRANSACTION;

-- удаляем по ошибке все платежи
DELETE FROM payments;

-- пытаемся получить данные из таблицы payments
SELECT * FROM payments;
```

  В этот момент мы увидим пустую таблицу payments, что может привести к очень большим проблемам. Однако мы можем воспользоваться **ROLLBACK**, чтобы вернуть все так, как было до начала транзакции:

```
ROLLBACK;
```

- В продолжение рассмотрим ситуацию, которую не покрывали пользовательские блокировки, и которую табличные блокировки делали слишком грубо и глобально. Мы возьмем случай, когда в транзакции изменяется какая-либо строка в таблице accounts, и в то же самое время другая сессия пытается изменить эту же самую строку без использования транзакции. Как мы должны помнить, табличная блокировка будет блокировать всю таблицу, а пользовательская блокировка в таком случае не сможет заблокировать вообще ничего. Предположим, что два менеджера пытаются одновременно изменить баланс пользователя Mary Sue (id = 2) - один в рамках транзакции пытается добавить 4 тысячи евро, а второй без транзакции хочет отнять семь тысяч. При этом открылась еще одна сессия без транзакции, в которой баланс правится для совсем другого пользователя - John Smith (id - 1) - ему в рамках таблицы accounts добавляется 10000 евро. Запустим эти запросы, но, как и в предыдущем случае, не будем сразу сразу коммитить транзакцию, чтобы убедиться в правильности наших расчетов:

  1 сессия (добавление 4000 для Mary Sue в транзакции):

```
START TRANSACTION;

UPDATE accounts SET total = total + 4000 WHERE id = 2;
```

  2 сессия (снятие 7000 для Mary Sue без транзакции):

```
UPDATE accounts SET total = total - 7000 WHERE id = 2;
```

  3 сессия (прибавление 10000 для John Smith без транзакции):

```
UPDATE accounts SET total = total + 10000 WHERE id = 1;
```

  Произойдет удивительная вещь - транзакционная 1 сессии успешно прибавит 4000 для Mary Sue, однако второй запрос (вне транзакции), который должен отнять у Mary Sue 7000 евро, замрет в ожидании. В свою очередь третий запрос (тоже вне транзакции) обновит данные для John Smith, который также находится в той же таблице. Если мы выполним комит первой транзакции, то в конце концов и второй запрос выполнится успешно, что приведёт таблицу accounts в следующий вид:

  1 сессия:

```
COMMIT;
```

| **id** | **name** | **total** |
|---|---|---|
| 1 | John Smith | 23000 |
| 2 | Mary Sue | 27000 |
| 3 | Michael Adams | 25000 |
| 5 | Chong Li | 27000 |

Какие мы должны сделать два главных вывода из всего увиденного? Во-первых, мы должны понять, что когда происходит транзакция, то она следит за целостностью данных не только в рамках других транзакций, но и за попытками изменений данных в простых запросах - что выгодно отличается от пользовательских блокировок. Если какой-то запрос пытается менять строку с тем же самым ключом в таблице, которую меняет транзакция, то запрос будет приостановлен (или же на это время будет приостановлена транзакция, если запрос был выполнен первым). Во-вторых, транзакция не препятствует обновлению тех строк в таблице, которые она не меняет сама - как мы могли видеть, противоположную логику имели табличные блокировки, которые блокировали все строки в таблице без разбора. Более того, транзакции не запрещают чтение из таблиц, которые обновляются - даже именно те строки, которые обновляются в данный момент.

- При работе с транзакциями мы иногда можем столкнуться с уже знакомой нам по блокировкам ошибкой *Lock wait timeout exceeded*. Это может произойти в том случае, если одна транзакция или даже запрос долго ждут другую транзакцию, которая изменила необходимые им данные, но еще не выполнила команду **COMMIT** или **ROLLBACK**. Если это ошибка произойдет, то тот запрос не будет выполнен, и мы должны будем выполнить его вновь.

- После всех приведенных примеров может показаться, что транзакции могут все. Это не так - они действительно отлично контролируют данные, но бессильны перед изменением структуры таблиц, добавлением и удалением новых триггеров, функций, событий и представлений - короче говоря, перед всеми операциями **DDL (DATA DEFINITION LANGUAGE)**. Ниже перечислен список команд, при использовании которых транзакция автоматически выполнит **COMMIT**:

**ALTER EVENT**
**ALTER FUNCTION**
**ALTER PROCEDURE**
**ALTER SERVER**
**ALTER TABLE**
**ALTER TABLESPACE**
**ALTER VIEW**
**CREATE DATABASE**
**CREATE EVENT**
**CREATE FUNCTION**
**CREATE INDEX**
**CREATE PROCEDURE**
**CREATE ROLE**
**CREATE SERVER**
**CREATE SPATIAL REFERENCE SYSTEM**
**CREATE TABLE**
**CREATE TABLESPACE**
**CREATE TRIGGER**
**CREATE VIEW**
**DROP DATABASE**
**DROP EVENT**
**DROP FUNCTION**
**DROP INDEX**
**DROP PROCEDURE**
**DROP ROLE**
**DROP SERVER**
**DROP SPATIAL REFERENCE SYSTEM**
**DROP TABLE**
**DROP TABLESPACE**
**DROP TRIGGER**
**DROP VIEW**
**INSTALL PLUGIN**
**RENAME TABLE**
**TRUNCATE TABLE**
**UNINSTALL PLUGIN**
**ALTER USER**
**CREATE USER**
**DROP USER**
**GRANT**
**RENAME USER**
**REVOKE**
**SET PASSWORD**
**BEGIN**
**LOCK TABLES**
**START TRANSACTION**
**UNLOCK TABLES**
**LOAD DATA**
**ANALYZE TABLE**
**CACHE INDEX**
**CHECK TABLE**
**FLUSH**
**LOAD INDEX INTO CACHE**
**OPTIMIZE TABLE**
**REPAIR TABLE**
**RESET**
**START REPLICA**
**STOP REPLICA**
**RESET REPLICA**
**CHANGE REPLICATION SOURCE TO**
**CHANGE MASTER TO**

## УРОВНИ ИЗОЛЯЦИИ

- В предыдущих подглавах мы несколько раз упоминали о неких загадочных уровнях изоляции, которые могут оказать влияние на то, как будет вести себя транзакция в той или иной ситуации. Пришло время уделить этому внимание. Дело в том, что транзакции могут по разному взаимодействовать с данными, которые меняет другая транзакция. При определенном уровне изоляции одна транзакция может видеть незафиксированные обновления другой транзакции, которые потом будут отменены при помощи **ROLLBACK** (т.н. **грязное чтение**), либо получать разные уже зафиксированные обновления при одних и тех же

запросах в одни и те же таблицы (**неповторяемые чтения**), а также не просто видеть изменившиеся строки, а видеть совершенно новые строки, которых раньше не было при таком же запросе (**фантомы**). Кроме того, некоторые уровни транзакции немного по разному блокируют данные при модификациях. Познакомимся подробнее с каждым уровнем изоляции:

- **READ UNCOMMITTED** — чтение незафиксированных изменений своей транзакции и конкурирующих транзакций, возможны грязные чтения, неповторяемые чтения и фантомы. При обновлении данных блокирует только обновляемую запись, разрешает другим транзакциям вставку в обновляемую таблицу.

- **READ COMMITTED** — чтение всех изменений своей транзакции и зафиксированных изменений конкурирующих транзакций, грязные чтения невозможны, возможны неповторяемые чтения и фантомы. Особенности блокировок при обновлении такие же, как и у **READ UNCOMMITTED**.

- **REPEATABLE READ** — уровень по умолчанию для MySQL. Чтение всех изменений своей транзакции, любые изменения, внесенные конкурирующими транзакциями после начала своей недоступны, грязные и неповторяемые чтения невозможны, возможны фантомы. Кроме блокировки обновляемой записи может заблокировать создание новых записей, которые могут попасть под фильтрацию при обновлении. Например, при обновлении счетов, на которых более 25000 евро, запись в таблицу, содержащая более 25000 евро будет блокироваться и ждать, пока транзакция с обновлением не будет завершена.

- **SERIALIZABLE** — самый высокий уровень изоляции, который решает проблему фантомного чтения, заставляя транзакции выполняться в таком порядке, чтобы исключить возможность конфликта. Если при помощи этого уровня в транзакции строки были хотя бы просмотрены, то в других транзакциях и запросах они и их таблицы будут заблокированы для любых видов изменения.

- Для получения уровня изоляции есть следующие команды:

```
-- получение уровня изоляции для нынешней сессии
SELECT @@transaction_ISOLATION;
```

```
-- получение глобального уровня изоляции для всей базы данных
SELECT @@global.transaction_ISOLATION;
```

Результат будет примерно таким:

| @@global.transaction_ISOLATION |
| --- |
| REPEATABLE-READ |

- Устанавливается уровень изоляции схожим методом:

```
-- установка уровня изоляции для нынешней сессии
SET SESSION TRANSACTION ISOLATION LEVEL REPEATABLE READ;
```

```
-- установка глобального уровня изоляции для всей базы данных
SET GLOBAL TRANSACTION ISOLATION LEVEL REPEATABLE READ;
```

Важный момент - если мы поменяем уровень изоляции глобально, то уровень сессии на данный останемся неизменным. Для изменении в рамках сессии его надо менять явно, а не ждать, что он изменится сам.

- Теперь последовательно рассмотрим пример каждого уровня изоляции и начнем с **READ UNCOMMITTED**. В одной транзакции будем менять таблицу accounts, а во второй наблюдать ее изменения.

1 сессия:

```
START TRANSACTION;
UPDATE accounts SET total = 0;
```

2 сессия:

```
SET SESSION TRANSACTION ISOLATION LEVEL READ UNCOMMITTED;
START TRANSACTION;
SELECT * FROM accounts;
```

Мы убедимся, что 2 сессия увидит еще незафиксированные изменения 1 сессии - далее мы убедимся, что так умеет только этот уровень изоляции:

| id | name | total |
|---|---|---|
| 1 | John Smith | 0 |
| 2 | Mary Sue | 0 |
| 3 | Michael Adams | 0 |
| 5 | Chong Li | 0 |

1 сессия

```
ROLLBACK;
```

2 сессия

```
COMMIT;
```

После того, как мы выполнили **ROLLBACK** в первой сессии, создалась ситуация, когда вторая сессии могла использовать данные, которых как будто бы даже и не существовало.

- Далее попробуем поработать с **READ COMMITTED**:

1 сессия:

```
START TRANSACTION;
UPDATE accounts SET total = total - 1000;
```

2 сессия:

```
SET SESSION TRANSACTION ISOLATION LEVEL READ COMMITTED;
START TRANSACTION;
SELECT * FROM accounts;
```

Убеждаемся, что уровень **READ COMMITTED** не видит незафиксированные изменения:

| id | name | total |
|---|---|---|
| 1 | John Smith | 23000 |
| 2 | Mary Sue | 27000 |
| 3 | Michael Adams | 25000 |
| 5 | Chong Li | 27000 |

1 сессия:

```
COMMIT;
```

2 сессия:

```
SELECT * FROM accounts;
COMMIT;
```

А вот теперь, когда данные уже зафиксированы, **READ COMMITTED** уже начинает их видеть:

| id | name | total |
|---|---|---|
| 1 | John Smith | 22000 |
| 2 | Mary Sue | 26000 |
| 3 | Michael Adams | 24000 |
| 5 | Chong Li | 26000 |

- Пришло время проанализировать работу **REPEATABLE READ**:

1 сессия:

```
START TRANSACTION;
UPDATE accounts SET total = total + 500;
```

2 сессия:

```
SET SESSION TRANSACTION ISOLATION LEVEL REPEATABLE READ;
START TRANSACTION;
SELECT * FROM accounts;
```

**REPEATABLE READ**, как и **READ COMMITTED** не видит незафиксированные изменения:

| id | name | total |
|---|---|---|
| 1 | John Smith | 22000 |
| 2 | Mary Sue | 26000 |
| 3 | Michael Adams | 24000 |
| 5 | Chong Li | 26000 |

1 сессия:

```
COMMIT;
```

2 сессия:

```
SELECT * FROM accounts;
COMMIT;
```

Как бы это удивительно не звучало, мы могли заметить, что **REPEATABLE READ** не увидел изменений даже в том случае, если они уже даже были зафиксированы. Настоящий вид таблицы accounts:

| id | name | total |
|---|---|---|
| 1 | John Smith | 22500 |
| 2 | Mary Sue | 26500 |
| 3 | Michael Adams | 24500 |
| 5 | Chong Li | 26500 |

- Применим вариант блокирования вставки после обновления с фильтрацией в рамках **REPEATABLE READ**:

1 сессия:

```
START TRANSACTION;
UPDATE accounts SET total = total * 1.1 WHERE total > 25000;
```

2 сессия:

```
SET SESSION TRANSACTION ISOLATION LEVEL REPEATABLE READ;
START TRANSACTION;
INSERT INTO accounts (name, total) VALUES ('Jose Lopez',
30000);
```

Как и ожидалось, 2 сессия была заблокирована и ждет, когда закончится 1 сессия, ведь **REPEATABLE READ** не разрешает сразу добавлять те строки, которые могут попасть под фильтрацию, осуществляющуюся в другой транзакции.

1 сессия:

```
COMMIT;
```

2 сессия:

```
COMMIT;
```

После всех изменений таблица **accounts** будет выглядеть следующим образом:

| id | name | total |
|---|---|---|
| 1 | John Smith | 22500.00 |
| 2 | Mary Sue | 29150.00 |
| 3 | Michael Adams | 24500.00 |
| 5 | Chong Li | 29150.00 |
| 6 | Jose Lopez | 30000.00 |

- Последним рассмотрим уровень изоляции **SERIALIZABLE** - попытаемся в сериализованной транзакции просмотреть все строки таблицы accounts, а потом в другой транзакции попытаемся их модифицировать - сделать сумму средств на всех счетах равной нулю:

1 сессия:

```
SET SESSION TRANSACTION ISOLATION LEVEL SERIALIZABLE;
START TRANSACTION;
SELECT * FROM accounts;
```

2 сессия:

```
START TRANSACTION;
UPDATE accounts SET total = 0;
```

Мы можем убедиться, что только чтение строк таблицы accounts уже заблокировало их изменение, т.к. вторая сессия зависла и ожидает завершение первой сессии.

1 сессия:

```
COMMIT;
```

2 сессия:

```
COMMIT;
```

Вот теперь вторая сессия успешно завершится, и все счета будут обнулены:

| **id** | **name** | **total** |
|---|---|---|
| 1 | John Smith | 0.00 |
| 2 | Mary Sue | 0.00 |
| 3 | Michael Adams | 0.00 |
| 5 | Chong Li | 0.00 |
| 6 | Jose Lopez | 0.00 |