MySQL

Блокировки и транзакции

Блокировки

Введение

В реальной жизни существуют базы данных и 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 - т.е. с проблемой конкурентного доступа к данным.

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 (соединение) захватывает доступ к определенным видам взаимодействия с необходимыми ей таблицами и производит свои операции, пока остальные сессии (соединения) 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.

Типы блокировок

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.

Особенности блокировок

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 будут получать блокировки на чтение.

Подготовка к работе

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 (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

Длительность ожидания блокировки

Что случится, если блокировка будет осуществляться слишком долго? Теоретически другие сессии должны ждать столько секунд, сколько прописано в их 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;
```

Если же мы хотим поменять время ожидания не для конкретной сессии, а глобально, то нам следует модифицировать параметр **@@GLOBAL.lock_wait_timeout**

Работа с блокировками (теория)

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.

Работа с блокировками (блокировка на запись)

Два счета (владельцы John Smith и Michael Adams) одновременно отправляют третьему счету (владелец Mary Sue) денежные переводы на 2000 и 3000 евро. Осуществим каждый из двух переводов с помощью блокировки таблиц:

1 сессия (John Smith - > Mary Sue):

```
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;
```

```
UNLOCK TABLES;
```

2 сессия (Michael Adams -> Mary Sue) - будет ждать, когда закончится блокировка (**UNLOCK TABLES;**) первой сессии:

```
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;
```

```
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

Работа с блокировками (блокировка на чтение)

Будем блокировать таблицу accounts, которую будет просматривать (читать) Michael Adams (он первым получит блокировку), одновременно свой счет будет просматривать John Smith и менять свое имя Mary Sue (на Mary Jane).

1 сессия - получение данных владельцем счета Michael Adams с применением блокировки на чтение:

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

```
UNLOCK TABLES;
```

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

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

```
UNLOCK TABLES;
```

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

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

```
UNLOCK TABLES;
```

Таблица **accounts**:

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

Каскадность блокировки (через внешние ключи)

Если мы попытаемся обновить таблицу, но перед этим объявим блокировку на чтение - произойдет ошибка. Однако надо знать, что блокировке какой-либо таблицы все остальные таблицы, связанные с изначальной таблицей ограничениями внешних ключей, также неявно проверяются на все виды взаимодействия. Это становится очевидным, когда мы заблокируем на чтение таблицу **payments**, попытаемся ее обновить, но вместо того, чтобы получить ошибку в контексте **payments**, мы получим ошибку в контексте таблицы **accounts** (они как раз связаны между собой ограничением первичного ключа):

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

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

Глобальная блокировка

Если нам необходимо заблокировать 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). Снять глобальную блокировку можно стандартной командой **UNLOCK TABLES**.

Пользовательская блокировка (теория)

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 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 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');
```

Пользовательская блокировка (практика)

Приведем конкретный пример - 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

Блокировки могут помочь 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 работу базы, выделяем приоритетное соединение, а остальные бесцельно простаивают. Пользовательская блокировка позволяет ограничится блокировкой только одной строки, но она не дает никаких гарантий и работает только в том случае, если все пользователи соблюдают договоренности. Кроме того, ни одна из 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 возобновления работы окажутся в половинчатом 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 данные можно вернуть обратно только в рамках таблиц на движке InnoDB.

Характеристики транзакций

Транзакции 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)**: определят, 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. 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.

Синтаксис создания транзакций

```
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];
```

Описание атрибутов, которые инициализируют транзакцию

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 (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.

Описание атрибутов, которые завершают транзакцию

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 (теория)

Перед тем, как мы перейдем непосредственно к применению транзакций, 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 СУБД ставит **COMMIT** и фиксирует изменения. За это поведение отвечает настройка **@@autocommit**, значение которой можно получить следующим образом:

```
SELECT @@autocommit;
```

@@autocommit
1

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

```
SET autocommit = 0;
```

Если мы хотим переопределить поведение для всех сессий и для всех последующих подключений, то делаем так:

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

Автоматический коммит (практика)

Теперь попытаемся вставить в таблицу accounts новую запись при выключенном автокоммите:

```
-- 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;
```

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

Применение транзакций на практике (обновление одной и той же таблицы) (1)

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.

Применение транзакций на практике (обновление одной и той же таблицы) (2)

Теперь зафиксируем данные для обеих сессий и убедимся, что таблицы находятся в нормальном состоянии:

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

```
COMMIT;
```

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

```
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

Применение транзакций на практике (обновление одних и тех же данных) (1)

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 c id = 3 -> John Smith c 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 c id = 5 -> John Smith c 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.

Применение транзакций на практике (обновление одних и тех же данных) (2)

Ожидание второй сессии будет до тех пор, пока первая сессии не сделает **COMMIT** или **ROLLBACK** (в нашем случае **COMMIT**). Сделаем же это:

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

```
COMMIT;
```

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

```
-- 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

Применение транзакций на практике (возвращение удаленных данных)

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;
```

Применение транзакций на практике (смешанный случай) (1)

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.

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;
```

Применение транзакций на практике (смешанный случай) (2)

Произойдет следующая 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 представленный ниже вид:

1 сессия:

```
COMMIT;
```

Таблица **accounts:**

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

Ограничения транзакций

После всех примеров может показаться, что транзакции могут все. Это не так - они 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	CREATE TABLE	DROP VIEW	LOAD DATA
ALTER FUNCTION	CREATE TABLESPACE	INSTALL PLUGIN	ANALYZE TABLE
ALTER PROCEDURE	CREATE TRIGGER	RENAME TABLE	CACHE INDEX
ALTER SERVER	CREATE VIEW	TRUNCATE TABLE	CHECK TABLE
ALTER TABLE	DROP DATABASE	UNINSTALL PLUGIN	FLUSH
ALTER TABLESPACE	DROP EVENT	ALTER USER	LOAD INDEX INTO CACHE
ALTER VIEW	DROP FUNCTION	CREATE USER	OPTIMIZE TABLE
CREATE DATABASE	DROP INDEX	DROP USER	REPAIR TABLE
CREATE EVENT	DROP PROCEDURE	GRANT	RESET
CREATE FUNCTION	DROP ROLE	RENAME USER	START REPLICA
CREATE INDEX	DROP SERVER	REVOKE	STOP REPLICA
CREATE PROCEDURE	DROP SPATIAL REFERENCE SYSTEM	SET PASSWORD	RESET REPLICA
CREATE ROLE	DROP TABLE	BEGIN	CHANGE REPLICATION SOURCE TO
CREATE SERVER	DROP TABLESPACE	LOCK TABLES	CHANGE MASTER TO
CREATE SPATIAL REFERENCE SYSTEM	DROP TRIGGER	START TRANSACTION	

Временные ограничения транзакций

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 или та транзакция в которой она произойдет, не будут выполнены и мы должны будем начать их сначала.

Если мы хотим увеличить или уменьшить время ожидания транзакции, то нам необходимо модифицировать переменную **@@SESSION.innodb_lock_wait_timeout** (или **@@GLOBAL.innodb_lock_wait_timeout** если нам надо изменить этот параметр глобально).

Уровни изоляции транзакций

Введение

В рамках транзакций существуют т.н. уровнях изоляции, которые могут оказать влияние на то, как будет вести себя транзакция в той или иной ситуации, т.е. 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.

Уровни 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 таблице, 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.

Получение уровня изоляции транзакций

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

Установка уровня изоляции транзакций

```
-- 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.

Пример работы транзакций с уровнем изоляции READ UNCOMMITTED (1)

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

Пример работы транзакций с уровнем изоляции READ UNCOMMITTED (2)

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.

Пример работы транзакций с уровнем изоляции READ COMMITTED (1)

В одной транзакции опять будем менять таблицу **accounts**, а во второй наблюдать ее изменения, но уже с уровнем изоляции **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

Пример работы транзакций с уровнем изоляции READ COMMITTED (2)

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

Пример работы транзакций с уровнем изоляции REPEATABLE READ (1.1)

В очередной раз в одной транзакции будем менять таблицу **accounts**, а во второй наблюдать ее изменения, но уже с уровнем изоляции **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

Пример работы транзакций с уровнем изоляции REPEATABLE READ (1.2)

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.

Вид таблицы **accounts** из 2 транзакции после того, как в 1 транзакции был сделан **COMMIT**:

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

Настоящий вид таблицы **accounts**:

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

Пример работы транзакций с уровнем изоляции REPEATABLE READ (2.1)

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.

Пример работы транзакций с уровнем изоляции REPEATABLE READ (2.2)

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

Пример работы транзакций с уровнем изоляции SERIALIZABLE (1)

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.

Пример работы транзакций с уровнем изоляции SERIALIZABLE (2)

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