GeekBrains

Базы данных. Интерактивный курс

Урок 6

Транзакции, переменные, представления

Транзакции

Транзакцией называется атомарная группа запросов SQL, т. е. запросы, которые рассматриваются как единое целое. Если база данных может выполнить всю группу запросов, она делает это, но если любой из них не может быть выполнен в результате сбоя или по какой-то другой причине, не будет выполнен ни один запрос группы. Все или ничего.

Операции с денежными средствами — классический пример, показывающий, почему необходимы транзакции. Если при оплате покупки происходит перевод от клиента электронному магазину, то счет клиента должен уменьшиться на эту сумму, а счет электронного магазина — увеличиться на нее же.

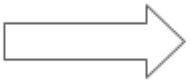
Пусть у нас есть таблица **account** со счетами пользователей. В этой же таблице есть счет интернет-магазина. Он отличается тем, что внешний ключ **user_id** у него принимает значение **NULL**. Для осуществления покупки нам необходимо переместить 2000 рублей со счета клиента на счет магазина.

1. Убедиться, что остаток на счете клиента больше 2000 рублей.

2. Вычесть 2000 рублей со счета клиента.

3. Добавить 2000 к счету интернет-магазина.

id	user_id	total
1	4	5000
2	3	0
3	2	200
4	NULL	25000



id	user_id	total
1	4	3000
2	9	0
3	2	200
4	4	27000

Вся операция должна быть организована как транзакция, чтобы в случае неудачи на любом из этих трех этапов все выполненные ранее шаги были отменены.

Давайте смоделируем ситуацию в консоли. Для начала создадим таблицу **accounts**:

```
DROP TABLE IF EXISTS accounts;
CREATE TABLE accounts (
  id SERIAL PRIMARY KEY,
  user_id INT,
  total DECIMAL (11,2) COMMENT 'Счет',
  created_at DATETIME DEFAULT CURRENT_TIMESTAMP,
  updated_at DATETIME DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP
) COMMENT = 'Счета пользователей и интернет магазина';

INSERT INTO accounts (user_id, total) VALUES
  (4, 5000.00),
  (3, 0.00),
  (2, 200.00),
  (NULL, 25000.00);
```

Начинаем транзакцию командой **START TRANSACTION**:

```
START TRANSACTION;
```

Далее выполняем команды, входящие в транзакцию:

```
SELECT total FROM accounts WHERE user_id = 4;
```

Убеждаемся, что на счету пользователя достаточно средств:

```
UPDATE accounts SET total = total - 2000 WHERE user_id = 4;
```

Снимаем средства со счета пользователя:

```
UPDATE accounts SET total = total + 2000 WHERE user_id IS NULL;
```

Перемещаем их на счет интернет-магазина. Давайте посмотрим состояние таблицы **accounts**:

```
SELECT * FROM accounts;
```

Изменения в рамках транзакций не сохранены в таблицах. Давайте переключимся в другую консоль и посмотрим состояние таблицы **accounts** глазами другого пользователя:

```
SELECT * FROM accounts;
```

Как видим, другие пользователи видят исходное состояние таблицы. Чтобы изменения вступили в силу, мы должны выполнить команду **COMMIT**:

```
COMMIT;
```

Если команда проходит без ошибок, изменения фиксируются базой данных и другие пользователи тоже начинают их видеть.

```
SELECT * FROM accounts;
```

Если в момент, когда мы выполняли транзакцию, какая-то другая транзакция уже изменила счет пользователя, то команда COMMIT завершится ошибкой и все изменения в рамках текущей транзакции будут аннулированы. Мы можем и самостоятельно отменять транзакции.

Например, давайте спишем еще 2000 рублей со счета пользователя:

```
START TRANSACTION;
```

Начинаем транзакцию:

```
SELECT total FROM accounts WHERE user_id = 4;
UPDATE accounts SET total = total - 2000 WHERE user_id = 4;
UPDATE accounts SET total = total + 2000 WHERE user_id IS NULL;
```

И тут мы выясняем, что не можем завершить транзакцию, например, пользователь ее отменяет или происходит еще что-то. Чтобы ее отметить мы можем воспользоваться командой:

```
ROLLBACK;
```

Давайте посмотрим состояние таблицы **accounts**:

```
SELECT * FROM accounts;
```

Можем видеть, что изменения не отразились на состоянии таблицы.

Для некоторых операторов нельзя выполнить откат при помощи оператора **ROLLBACK**. К их числу относят следующие команды:

- **CREATE INDEX**
- **DROP INDEX**
- **CREATE TABLE**

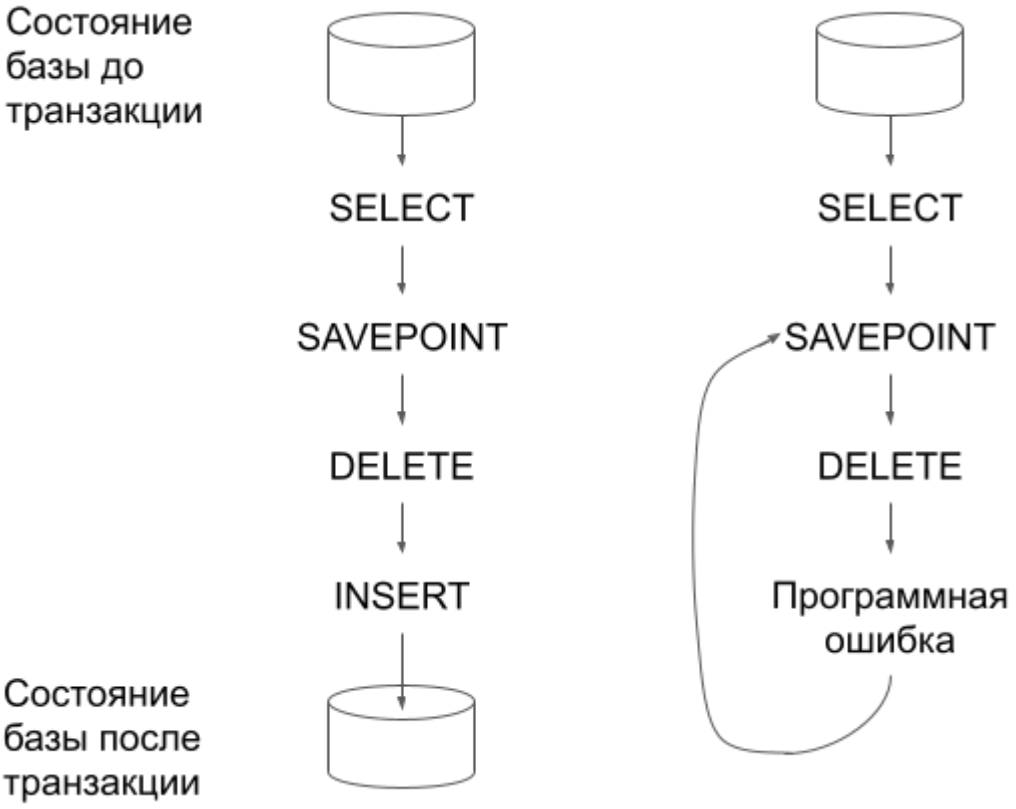
- **DROP TABLE**
- **TRUNCATE TABLE**
- **ALTER TABLE**
- **RENAME TABLE**
- **CREATE DATABASE**
- **DROP DATABASE**
- **ALTER DATABASE**

Не помещайте их в транзакции с другими операторами.

Кроме того, существует ряд операторов, которые неявно завершают транзакцию, как если бы был вызван оператор **COMMIT**:

- **ALTER TABLE**
- **BEGIN**
- **CREATE INDEX**
- **CREATE TABLE**
- **CREATE DATABASE**
- **DROP DATABASE**
- **DROP INDEX**
- **DROP TABLE**
- **DROP DATABASE**
- **LOAD MASTER DATA**
- **LOCK TABLES**
- **RENAME**
- **SET AUTOCOMMIT=1**
- **START TRANSACTION**
- **TRUNCATE TABLE**

Транзакции не могут быть вложенными, потому что любой оператор, начинающий транзакцию, приводит к завершению предыдущей.



Точка сохранения представляет собой место в последовательности событий транзакции, которое может выступать в качестве промежуточной точки восстановления. Откат текущей транзакции может быть выполнен не к началу транзакции, а к точке сохранения.

Для работы с точками сохранения предназначены два оператора:

- **SAVEPOINT**
- **ROLLBACK TO SAVEPOINT**

```
START TRANSACTION;
SELECT total FROM accounts WHERE user_id = 4;
SAVEPOINT accounts_4;
UPDATE accounts SET total = total - 2000 WHERE user_id = 4;
```

Допустим мы хотим отменить транзакцию и вернуться в точку сохранения. В этом случае мы можем воспользоваться оператором **ROLLBACK TO SAVEPOINT**:

```
ROLLBACK TO SAVEPOINT accounts_4;
```

Допускается создание нескольких точек сохранения. Если текущая транзакция имеет точку сохранения с таким же именем, старая точка удаляется и устанавливается новая. Все точки сохранения транзакций удаляются, если выполняется оператор **COMMIT** или **ROLLBACK** без указания имени точки сохранения.

Пока мы не используем оператор **START TRANSACTION**. В MySQL каждый запрос рассматривается как транзакция. Обычно говорят, что MySQL работает в режиме автозавершения транзакций. Мы можем отключить такой режим при помощи команды:

```
SET AUTOCOMMIT=0;
```

В этом случае любая последовательность команд будет рассматриваться как транзакция:

```
SELECT total FROM accounts WHERE user_id = 4;
UPDATE accounts SET total = total - 2000 WHERE user_id = 4;
UPDATE accounts SET total = total + 2000 WHERE user_id IS NULL;
SELECT * FROM accounts;
```

Чтобы сохранить изменения, нужно выполнить команду **COMMIT** или отменить команды при помощи **ROLLBACK**. Например, давайте отменим транзакцию:

```
ROLLBACK;
SELECT * FROM accounts;
```

Чтобы включить режим автоматического завершения транзакций, необходимо выполнить оператор **SET AUTOCOMMIT=1**.

Транзакций недостаточно, если система не удовлетворяет принципу ACID. Аббревиатура ACID расшифровывается как атомарность, согласованность, изолированность и сохраняемость).

- Atomicy — атомарность.
- Consistency — согласованность.
- Isolation — изолированность.
- Durability — сохраняемость.

Атомарность подразумевает, что транзакция должна функционировать как единая неделимая единица. Вся транзакция была либо выполняется, либо отменяется. Когда транзакции атомарны, не существует такого понятия, как частично выполненная транзакция.

При выполнении принципа согласованности база данных должна всегда переходить из одного непротиворечивого состояния в другое непротиворечивое состояние. В нашем примере согласованность гарантирует, что сбой между двумя UPDATE-командами не приведет к исчезновению 2000 рублей со счета пользователя. Транзакция просто не будет зафиксирована, и ни одно из изменений в этой транзакции не будет отражено в базе данных.

Изолированность подразумевает, что результаты транзакции обычно невидимы другим транзакциям, пока она не закончена. Это гарантирует, что, если в нашем примере во время транзакции будет выполнен запрос на извлечение средств пользователя, такой запрос по-прежнему будет видеть 2000 рублей на его счету.

Сохраняемость гарантирует, что изменения, внесенные в ходе транзакции, будучи зафиксированными, становятся постоянными. Это означает, что изменения должны быть записаны так, чтобы данные не могли быть потеряны в случае сбоя системы. Транзакции ACID гарантируют, что интернет-магазин не потеряет ваши деньги. Это очень сложно или даже невозможно сделать с помощью логики приложения.

Уровни изоляции

Транзакции требуют довольно много ресурсов и замедляют выполнения запросов. Поэтому часто приходится идти на компромиссы и дополнительную настройку транзакций.

Изолированность — более сложное понятие, чем кажется на первый взгляд. Стандарт SQL определяет четыре уровня изоляции с конкретными правилами, устанавливающими, какие изменения видны внутри и вне транзакции, а какие нет:

- **READ UNCOMMITTED**
- **READ COMMITTED**
- **REPEATABLE READ**
- **SERIALIZABLE**

Более низкие уровни изоляции обычно допускают большую степень совместного доступа и вызывают меньше накладных расходов. На первом уровне изоляции, **READ UNCOMMITTED**, транзакции могут видеть результаты незафиксированных транзакций.

На практике **READ UNCOMMITTED** используется редко, поскольку его производительность не намного выше, чем у других. На этом уровне вы видите промежуточные результаты чужих транзакций, т.е. осуществляете грязное чтение.

Уровень **READ COMMITTED** подразумевает, что транзакция увидит только те изменения, которые были уже зафиксированы другими транзакциями к моменту ее начала. Произведенные ею изменения останутся невидимыми для других транзакций, пока она не будет зафиксирована.

На этом уровне возможен феномен невоспроизводимого чтения. Это означает, что вы можете выполнить одну и ту же команду дважды и получить различный результат.

Уровень изоляции **REPEATABLE READ** решает проблемы, которые возникают на уровне **READ UNCOMMITTED**. Он гарантирует, что любые строки, которые считываются в контексте транзакции, будут выглядеть такими же при последовательных операциях чтения в пределах одной и той же транзакции, однако теоретически на этом уровне возможен феномен фантомного чтения (phantom reads).

Он возникает в случае, если вы выбираете некоторый диапазон строк, затем другая транзакция вставляет новую строку в этот диапазон, после чего вы выбираете тот же диапазон снова. В результате вы увидите новую фантомную строку. Уровень изоляции **REPEATABLE READ** установлен по умолчанию.

Самый высокий уровень изоляции, **SERIALIZABLE**, решает проблему фантомного чтения, заставляя транзакции выполняться в таком порядке, чтобы исключить возможность конфликта. Уровень **SERIALIZABLE** блокирует каждую строку, которую транзакция читает.

На этом уровне может возникать множество задержек и конфликтов при блокировках. На практике данный уровень изоляции применяется достаточно редко. Изменить уровень изоляции можно при помощи команды **SET TRANSACTION**:

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

Внутренняя реализация транзакций

Давайте в двух разных сессиях начнем транзакции:

```
START TRANSACTION;
UPDATE accounts SET total = total - 2000 WHERE user_id = 4;
UPDATE accounts SET total = total + 2000 WHERE user_id IS NULL;
```

```
START TRANSACTION;
UPDATE accounts SET total = total - 1000 WHERE user_id = 4;
UPDATE accounts SET total = total + 1000 WHERE user_id IS NULL;
```

Как видите, вторая транзакция не выполняет запрос, так как одна транзакция ожидает завершения другой. Только после того как мы введем **COMMIT** или **ROLLBACK**,транзакция продолжит работу:

```
COMMIT;
```

Возможна ситуация взаимоблокировки, когда две или более транзакции запрашивают блокировку одних и тех же ресурсов. В результате образуется циклическая зависимость: одна транзакция будет до бесконечности ожидать окончания другой, и конфликт не разрешится до тех пор, пока не произойдет какое-то событие, которое снимет взаимную блокировку.

Для разрешения этой проблемы в системах баз данных реализованы различные формы обнаружения взаимоблокировок и таймаутов.

Журнал транзакций

В базах данных транзакции редко записываются сразу в таблицу. Вместо этого они помещаются в журнал транзакций. Когда происходит какое-либо изменение, оно изменяет не таблицу на жестком диске, а находящуюся в памяти копию данных. Это происходит очень быстро.

Затем подсистема хранения записывает сведения об изменениях в журнал транзакции, который хранится на диске и потому долговечен. Это также относительно быстрая операция, поскольку событие добавляется в конец журнала вместо операций ввода-вывода в разных местах таблицы.

Впоследствии фоновый процесс обновит таблицу на диске. Таким образом, большинство СУБД, использующих журнал транзакций, сохраняют изменения на диске дважды.

Если происходит сбой после того, как внесена соответствующая запись в журнал транзакции, но до того, как обновлены сами данные, подсистема хранения может восстановить изменения после перезапуска сервера.

Запросить параметры журнала транзакций можно при помощи следующего запроса:

```
SHOW VARIABLES LIKE 'innodb_log_%';
```

Из полученного отчета можно увидеть, что журнал транзакций имеет два файла, размером 50 Мб. По умолчанию файлы журнала транзакций располагаются в каталоге данных.

Получить путь к каталогу данных в вашей системе можно при помощи следующего запроса:

```
SHOW VARIABLES LIKE 'datadir';
```

Давайте выйдем из диалогового режима MySQL в консоль командной строки и перейдем в каталог данных:

```
cd /usr/local/var/mysql/
```

```
ls -la
```

Среди файлов мы можем обнаружить два: **ib_logfile0** и i**b_logfile1** — это и есть файлы журнала транзакций, все они сначала помещаются сюда. По умолчанию мы используем движок InnoDB, поэтому фактически в настоящий момент мы исследуем его устройство и устройство его реализации транзакций.

InnoDB хранит таблицы всех баз данных в едином табличном пространстве в файле **ibdata1**. Физически единое табличное пространство может располагаться в нескольких файлах. Более того, мы можем выделить отдельное табличное пространство под каждую из таблиц.

Транзакции помещаются в файлы **ib_logfile0** и **ib_logfile1** и потом перегоняются в файлы единого табличного пространства: если сервер MySQL останавливается штатно, все транзакции из журнала сохраняются в таблицу. Если происходит сбой и сервер останавливается, например из-за отсутствия питания, перед стартом MySQL проверяет журнал транзакций и перегоняет в единое табличное пространство все транзакции которые не были сохранены в таблицах. Таким образом, потерять сохраненные транзакции невозможно.

Увеличивая размер журнала транзакций, можно укоротить вставку записей при штатной работе MySQL. Однако, чем больше журнал транзакций, тем медленнее будет стартовать сервер MySQL; лучше всего оставить это значение по умолчанию.

Управление режимом сохранения транзакций

За режим управления сохранения транзакций отвечает переменная **innodb_flush_log_at_trx_commit**, которая может принимать следующие значения:

- 0 — сохранение журнала раз в секунду,
- 1 — сохранение после каждой транзакции,
- 2 — сохранение журнала раз в секунду и после каждой транзакции.

Давайте запросим ее состояние:

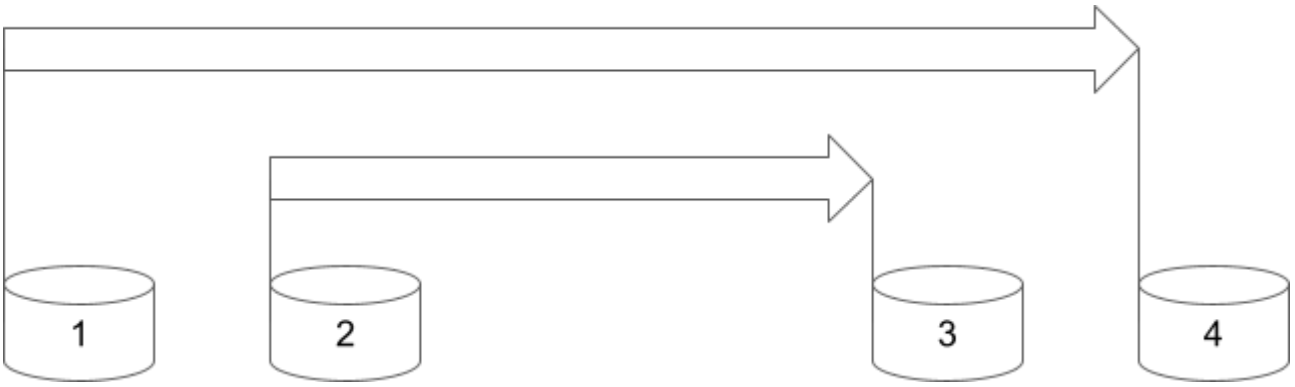
```
SHOW VARIABLES LIKE 'innodb_flush_log_at_trx_commit';
```

Установить новое значение:

```
SET GLOBAL innodb_flush_log_at_trx_commit = 0;
```

Механизм повышения степени конкурентности MVCC

Механизм транзакций в MySQL использует не просто механизм блокировки строк, но и механизм повышения степени конкурентности под названием MVCC (multiversion concurrency control — многоверсионное управление конкурентным доступом).



Этот механизм характерен не только для MySQL, но и вообще для реляционных СУБД: его также использует PostgreSQL и Oracle.

Когда мы упоминали уровни изоляции, мы отмечали, что возможны случаи, когда в ходе работы транзакции могут происходить фантомные или невоспроизводимые чтения. Если же мы будем жестко блокировать записи, как это происходит на уровне **SERIALIZABLE**, нам не избежать блокировок даже на операцию чтения.

Механизм MVCC позволяет во многих случаях вообще отказаться от блокировки и способен значительно снизить накладные расходы. Идея механизма MVCC состоит в создании мгновенных снимков состояния. Каждая транзакция читает данные из согласованного снимка состояния, то есть видит данные, которые были зафиксированы в базе на момент начала транзакции.

Даже если данные затем были изменены, каждая транзакция видит только старые данные по состоянию на конкретный момент времени. Выполняющая операцию записи транзакция может блокировать выполнение другой, записывающей в тот же объект. Однако операции чтения не требуют блокировок. Чтение никогда не блокирует запись, а запись — чтение. Благодаря этому база данных способна выполнять длительные запросы на чтение, продолжая в то же время обработку операций записи без какой-либо конкуренции блокировок между ними.

Многоверсионное управление конкурентным доступом MVCC работает только на уровнях изоляции **REPEATABLE READ** и **READ COMMITTED**. Уровень **READ UNCOMMITTED** несовместим с MVCC, поскольку запросы не считывают версию строки, соответствующую их версии транзакции. Они читают самую последнюю версию, несмотря ни на что. Уровень **SERIALIZABLE** несовместим с MVCC, поскольку операции чтения блокируют каждую возвращаемую строку.

Переменные

Часто результаты запроса необходимо использовать в последующих запросах. Для этого полученные данные следует сохранить во временных структурах. Эту задачу решают переменные SQL.

```
SELECT @total := COUNT(*) FROM products;
```

Объявление переменной начинается с символа @, за которым следует имя переменной. Значения переменным присваиваются посредством оператора **SELECT** с использованием оператора присваивания **:=**. В следующих запросах мы получаем возможность обращаться к переменной:

```
SELECT @total;
```

Переменная будет доступна только в текущей сессии.

```
SELECT @total;
```

Если мы откроем новую mysql-консоль и попробуем обратиться к переменной **@total**, мы получим неопределенное значение **NULL**. Переменные можно использовать не только в SELECT-списке, но и в WHERE-условии. Давайте извлечем товарную позицию с самой высокой ценой:

```
SELECT @price := MAX(price) FROM products;
```

Для этого воспользуемся функцией **MAX()** и сохраним максимальную цену товара в переменной **@pirce**.

```
SELECT * FROM products WHERE price = @price;
```

После этого мы можем использовать переменную в следующем запросе. Если в качестве значения переменной передается имя столбца, содержащего множество значений, то переменная получит последнее значение:

```
SELECT @id := id FROM products;
SELECT @id;
```

Имена переменных нечувствительны к регистру:

```
SELECT @id := 5, @ID := 3;
SELECT @id, @ID;
```

Несмотря на то, что может сложиться впечатление, что мы присваиваем значения двум разным переменным, мы работаем с одной и той же, и она получает значение 3. Переменные также могут объявляться при помощи оператора **SET**:

```
SET @last = NOW() - INTERVAL 7 DAY;
SELECT CURDATE(), @last;
```

Команда **SET**, в отличие от оператора **SELECT**, не возвращает результирующую таблицу. Переменные можно использовать для нумерации записей в таблице.

Допустим, есть таблица **tbl1** с единственным столбцом **value**, без первичного ключа:

```
SELECT * FROM tbl1;
```

И пусть требуется при выводе содержимого таблицы **tbl** пронумеровать строки. Мы можем завести переменную **@start**:

```
SET @start := 0;
```

И увеличивать значение на единицу по мере вывода записей:

```
SELECT @start := @start + 1 AS id, value FROM tbl1;
```

Помимо пользовательских переменных, сервер MySQL поддерживает большое количество системных переменных, с помощью которых можно осуществлять его тонкую настройку и масштабирование.

С некоторыми из переменных мы уже имели дело в предыдущих роликах. Получить их полный список можно при помощи оператора **SHOW VARIABLES**:

```
SHOW VARIABLES;
```

Оператор SHOW поддерживает ключевое слово LIKE, которое ведет себя так же, как в SELECT-запросах. При помощи этого ключевого слова можно отфильтровать выборку

```
SHOW VARIABLES LIKE 'read_buffer_size';
```

Сервер MySQL поддерживает два типа системных переменных:

- глобальные, которые влияют на сервер,
- и сеансовые, которые влияют на текущее соединение клиента с сервером.

При старте сервера происходит инициализация глобальных переменных значениями по умолчанию. Оператор SET позволяет изменять значения глобальных переменных уже после старта:

```
SET GLOBAL read_buffer_size = 2097152;
```

Для этого перед именем системной переменной устанавливается ключевое слово **GLOBAL**. Вместо ключевого слова **GLOBAL** можно перед названием переменной указывать два символа алеф:

```
SET @@global.read_buffer_size = 2097152;
```

Помимо глобальных переменных, сервер MySQL поддерживает набор сеансовых переменных для каждого соединения клиента с сервером. При установке соединения с сервером сеансовые переменные получают значения, заданные для глобальных переменных. Однако для тех сеансовых переменных, которые являются динамическими, клиент при помощи оператора **SET** может выставлять новые значения:

```
SET SESSION read_buffer_size = 2097152;
```

Для этого перед именем системной переменной устанавливается ключевое слово SESSION. Вместо ключевого слова GLOBAL можно перед названием переменной указывать два символа алеф, после которых указывает префикс session.

```
SET @@session.read_buffer_size = 2097152;
```

Чтобы установить локальной переменной значение глобальной, достаточно присвоить локальной переменной ключевое слово `DEFAULT`.

```
SET read_buffer_size = DEFAULT;
```

Временная таблица

Временная таблица автоматически удаляется по завершении соединения с сервером, а ее имя действительно только в течение данного соединения. Это означает, что два разных клиента могут использовать временные таблицы с одинаковыми именами без конфликта друг с другом или с существующей таблицей с тем же именем.

Создание временных таблиц осуществляется при помощи необязательного ключевого слова **TEMPORARY**:

```
CREATE TEMPORARY TABLE temp (id INT, name VARCHAR(255));
SHOW TABLES;
DESCRIBE temp;
```

Временные таблицы хранятся в файле **ibtmp1** в каталоге данных.

Динамические запросы

Динамическими называют запросы, которые, подобно пользовательским переменным, могут быть сохранены под конкретным именем и вызваны позже в течении сессии. Для объявления динамического запроса используется команда **PREPARE**:

```
PREPARE ver FROM 'SELECT VERSION()';
```

Выполняется такой динамический запрос при помощи команды EXECUTE

```
EXECUTE ver;
```

Динамические запросы имеют время жизни только в течение текущего сеанса. Т.е., после того, как соединение с сервером закрыто, динамический запрос перестаёт существовать. Если необходимо, чтобы сохранённый ранее запрос существовал более длительное время, необходимо прибегнуть к представлениям, которые мы рассмотрим на следующем уроке.

Запросы можно параметризовать, используя для этого символ вопросительного знака. Давайте создадим динамический запрос, который извлекает товарные позиции одного из разделов интернет-магазина:

```
PREPARE prd FROM 'SELECT id, name, price FROM products WHERE catalog_id = ?';
```

Как видно, вместо значения внешнего ключа **catalog_id** используется знак вопроса. Давайте зададим идентификатор раздела при помощи переменной:

```
SET @catalog_id = 1;
```

При вызове запроса при помощи оператора **EXECUTE** передать значение параметра можно при помощи конструкции **USING**.

```
EXECUTE prd USING @catalog_id;
```

Динамический запрос может иметь более одного параметра, в этом случае они перечисляются после ключевого слова **USING** через запятую в том порядке, в котором они встречаются в динамическом запросе.

Динамические запросы имеют ряд ограничений: не допускается использование вложенных динамических запросов, а также нескольких сразу. Т.е., динамический запрос всегда представляет лишь один запрос. Параметр всегда передаёт строку, т.е., динамически задать имя таблицы или столбца не получится.

Удалить динамический запрос можно при помощи оператора **DROP PREPARE**:

```
DROP PREPARE prd;
```

Представления

Основными структурными единицами в реляционных базах данных являются таблицы. Однако язык запросов SQL предоставляет еще один способ организации данных — представления.

Представление — это запрос на выборку (**SELECT**), которому присваивается уникальное имя и который можно сохранять или удалять из базы данных как обычную хранимую процедуру.

Представления позволяют увидеть результаты сохраненного запроса таким образом, как будто это полноценная таблица базы данных. Представления позволяют более гибко управлять правами доступа к таблицам: можно запретить прямое обращение пользователей к таблицам, и разрешить доступ только к представлениям.

Представления позволяют также обеспечить обратную совместимость для программ, ориентирующихся на старую структуру базы данных — достаточно создать представления со структурой, соответствующей старым таблицам.

```
SELECT * FROM catalogs;
```

Давайте создадим представления таблицы **catalogs**, в котором записи будут поддерживаться в отсортированном состоянии.

```
CREATE VIEW cat AS SELECT * FROM catalogs ORDER BY name;
```

Для создания представления используется команда **CREATE VIEW**, после которой мы указываем имя представления **cat**. Затем после ключевого слова **AS** пишем запрос представления.

К представлению мы можем обращаться как к обычной таблице:

```
SELECT * FROM cat;
```

Мы получили список разделов интернет-магазина в отсортированном состоянии:

```
SHOW TABLES;
```

Представление рассматривается MySQL как полноценная таблица, оно появляется в списке таблиц команды **SHOW TABLES**.

В полученном представлении мы использовали *, поэтому оно принимает структуру таблицы **catalogs.** Однако при создании представления можно явно указать список столбцов и даже изменить их названия и порядок следования:

```
CREATE VIEW cat_reverse (catalog, catalog_id)
AS SELECT name, id FROM catalogs;
SELECT * FROM cat_reverse;
```

Мы поменяли название столбца **name** на **catalog**, а **id** — на **catalog_id**. При этом порядок следования столбцов меняется на обратный. При формировании списка столбцов представления задаются только имена столбцов, а тип данных, их размер и другие характеристики берутся из определения столбца исходной таблицы.

В качестве столбцов представления могут выступать вычисляемые столбцы. Создадим представление **namecat**, которое выводит столбцы **id** и **name** таблицы **catalogs**, и дополнительно столбец **total,** в который будет помещаться количество символов в имени каталога.

```
CREATE OR REPLACE VIEW namecat (id, name, total)
AS SELECT id, name, LENGTH(name) FROM catalogs;
```

Обратите внимание мы используем команду **CREATE OR REPLACE VIEW**, чтобы заменить уже существующее представление:

```
SELECT * FROM namecat ORDER BY total DESC;
```

Ключевое слово **ALGORITHM** определяет способ формирования конечного запроса с участием представления и может принимать три значения:

- **MERGE** — при использовании данного алгоритма запрос объединяется с представлением таким образом, что представление заменяет собой соответствующие части в запросе;
- **TEMPTABLE** — результирующая таблица представления помещается во временную, которая затем используется в конечном запросе;
- **UNDEFINED** — в данном случае СУБД MySQL самостоятельно пытается выбрать алгоритм, предпочитая использовать подход **MERGE** и прибегая к алгоритму **TEMPTABLE** (создание временной таблицы) только в случае необходимости, т. к. метод **MERGE** более эффективен.

Если ни одно из значений **ALGORITHM** не указано, по умолчанию назначается **UNDEFINED**.

```
CREATE ALGORITHM = TEMPTABLE VIEW cat2 AS SELECT * FROM catalogs;
```

В созданном представлении мы требуем от MySQL при каждом обращении к представлению создавать временную таблицу.

```
DESCRIBE products;
```

Следует отметить, что представления способны скрывать ряд столбцов за счёт того, что SELECT-запросы могут извлекать не все столбцы таблицы. Такие представления называются вертикальными представлениями.

Давайте создадим такое представление для таблицы **products**:

```
CREATE OR REPLACE VIEW prod AS
SELECT id, name, price, catalog_id
FROM products
ORDER BY catalog_id, name;

SELECT * FROM prod;
```

Запросы к представлениям сами могут содержать условия **WHERE** и собственные сортировки. Правильное составление запроса к исходным таблицам — это забота MySQL.

```
SELECT * FROM prod ORDER BY name DESC;
```

Наряду с вертикальными используются горизонтальные представления, которые ограничивают доступ пользователей к строкам таблиц, делая видимыми только те строки, с которыми они работают.

Давайте создадим представление, которое извлекает из таблицы **products** только процессоры:

```
CREATE OR REPLACE VIEW processors AS
SELECT id, name, price, catalog_id
FROM products
WHERE catalog_id = 1;

SELECT * FROM processors;
```

В реальной практике могут встречаться смешанные представления, которые ограничивают таблицу и по горизонтали, и по вертикали. Термины «горизонтальное представление» и «вертикальное представление» являются условными и предназначены, чтобы лучше понять, как из исходной таблицы формируется представление.

Чтобы в представление можно было вставлять новые записи при помощи команды **INSERT** и обновлять существующие записи при помощи команды **UPDATE**, необходимо при создании представления использовать конструкцию **WITH CHECK OPTION**.

Во время вставки происходит проверка, чтобы вставляемые данные удовлетворяли WHERE-условию SELECT-запроса, лежащего в основе представления.

```
SELECT * FROM tbl1;
```

Воспользуемся таблицей **tbl1** для создания обновляемого представления:

```
CREATE VIEW v1 AS
SELECT * FROM tbl1 WHERE value < 'fst5'
WITH CHECK OPTION;

INSERT INTO v1 VALUES ('fst4');
```

Значение успешно вставилось.

```
INSERT INTO v1 VALUES ('fst5');
ERROR 1369 (HY000): CHECK OPTION failed 'shop.v1'
```

Однако при попытке вставить значение **fst5** срабатывает ограничение WHERE-условия.

Отредактировать представление можно при помощи команды **ALTER**:

```
ALTER VIEW v1 AS
SELECT * FROM tbl1 WHERE value > 'fst4'
WITH CHECK OPTION;
```

Кроме того, можно воспользоваться командой **CREATE OR REPLACE VIEW**:

```
CREATE OR REPLACE VIEW v1 AS
SELECT * FROM tbl1 WHERE value > 'fst4'
WITH CHECK OPTION;
```

Для удаления представлений предназначена команда **DROP VIEW**.

```
DROP VIEW cat, cat_reverse, namecat, prod, processors, v1;
```

При попытке удаления не существующего представления возникает ошибка:

```
DROP VIEW cat, cat_reverse, namecat, prod, processors, v1;
```

Как и многие команды, **DROP VIEW** поддерживает ключевое слово **IF EXISTS**, позволяющее игнорировать попытки удаления несуществующих представлений:

```
DROP VIEW IF EXISTS cat, cat_reverse, namecat, prod, processors, v1;
```

Используемые источники

1. https://dev.mysql.com/doc/refman/5.7/en/sql-syntax-transactions.html
2. https://dev.mysql.com/doc/refman/5.7/en/server-system-variables.html

3. https://dev.mysql.com/doc/refman/5.7/en/create-temporary-table.html

4. https://dev.mysql.com/doc/refman/5.7/en/sql-syntax-prepared-statements.html

5. https://dev.mysql.com/doc/refman/5.7/en/create-view.html

6. Линн Бейли. Head First. Изучаем SQL. — СПб.: Питер, 2012. — 592 с.

7. Грофф, Джеймс Р., Вайнберг, Пол Н., Оппель, Эндрю Дж. SQL: полное руководство, 3-е изд. : Пер. с англ. — М.: ООО "И.Д. Вильямс", 2015. — 960 с.

8. Дейт К. Дж. SQL и реляционная теория. Как грамотно писать код на SQL. — Пер. с англ. — СПб.: Символ-Плюс, 2010. — 480 с.

9. Кузнецов М.В., Симдянов И.В. MySQL на примерах. — СПб.: БХВ-Петербург, 2007. — 592с.

10. Кузнецов М.В., Симдянов И.В. MySQL 5. — СПб.: БХВ-Петербург, 2006. — 1024с.

11. Дейт К. Дж. Введение в системы баз данных, 8-е издание.: Пер. с англ. — М.: Издательский дом "Вильямс", 2005. — 1328 с.

12. Карвин Б. Программирование баз данных SQL. Типичные ошибки и их устранение. — Рид Групп, 2011. — 336 с.