Курс базы данных и SQL. Лекция 6

Терминология

Транзакция — это набор последовательных операций с базой данных, соединенных в одну логическую единицу.
Изоляция — это свойство транзакции, которое позволяет скрывать изменения, внесенные одной операцией транзакции при возникновении явления race condition
Процедура - это подпрограмма (например, подпрограмма) на обычном языке сценариев, хранящаяся в базе данных.
ACID - это набор из четырех требований к транзакционной системе, обеспечивающих максимально надежную и предсказуемую работу.

Доброго времени суток, уважаемые студенты!

Понятие транзакции, свойства ACID.

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. Для начала создадим таблицу **accounts**:

```
DROP TABLE IF EXISTS accounts;
CREATE TABLE accounts (
  id INT PRIMARY KEY AUTO_INCREMENT,
  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;
-- Чтобы изменения вступили в
-- силу, мы должны выполнить команду COMMIT
COMMIT;
-- скрипт выполнять полностью: начиная от первой и до самой последней строчки
```

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

```
SELECT * FROM accounts;
```

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

```
START TRANSACTION;
  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;
ROLLBACK; -- Откат до исходного состояния
```

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

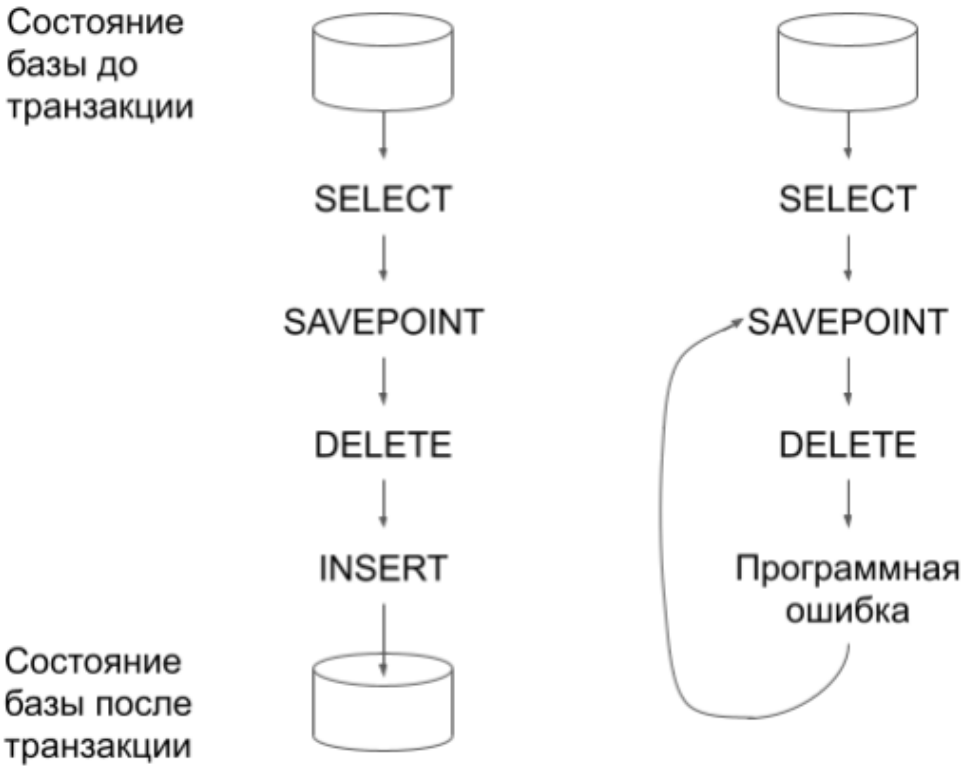
- CREATE INDEX
- DROP INDEX
- CREATE TABLE
- DROP TABLE
- TRUNCATE TABLE
- ALTER TABLE
- RENAME TABLE
- CREATE DATABASE
- DROP DATABASE
- ALTER DATABASE

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

В случае сбоя в транзакции откат можно делать до некой точки сохранения - SAVEPOINT.



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;
SELECT * FROM accounts;
```

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.

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.

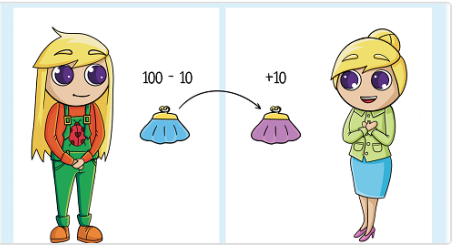
Сохраняемость гарантирует, что изменения, внесенные в ходе транзакции, будучи зафиксированными, становятся постоянными. Это означает, что изменения должны быть записаны так, чтобы данные не могли быть потеряны в случае сбоя системы. Транзакции ACID гарантируют, что интернет-магазин не потеряет ваши деньги.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

Эффекты с картинками:

Требования ACID на простом языке

Мне нравятся книги из серии Head First O`Reilly — они рассказывают просто о сложном. И я стараюсь делать также. Когда речь идёт о базах данных, могут всплыть магические слова

https://habr.com/ru/post/555920/



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.

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 accounts;
```

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;
```

Переменные также могут объявляться при помощи оператора **SET. Команда SET, в отличие от оператора SELECT, не возвращает результирующую таблицу**:

```
SET @last = NOW() - INTERVAL 7 DAY; -- от текущей даты отнять 7 дней
SELECT CURDATE(), @last;
```

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.

В MySQL временная таблица — это особый тип таблицы, который позволяет вам сохранить временный набор результатов, который вы можете повторно использовать несколько раз в
одном сеансе. Временная таблица очень удобна, когда невозможно запрашивать данные, требующие одного SELECT оператора с JOIN предложениями.

Временная таблица создается с помощью CREATE TEMPORARY TABLE. Обратите внимание, что ключевое слово TEMPORARY добавлено между ключевыми словами CREATE и TABLE.

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

DESCRIBE temp; -- Показ всех столлбцов в таблице temp
```

Хранимые процедуры и функции

Хранимые процедуры и функции позволяют сохранить последовательность SQL-операторов и
вызывать их по имени функции или процедуры:
- CREATE PROCEDURE procedure_name
- CREATE FUNCTION function_name

Разница между процедурой и функцией заключается в том, что функции возвращают значение и их можно встраивать в SQL-запросы, в то время как хранимые процедуры вызываются явно.

Процедуры.

Синтаксис:

```
DELIMITER {custom delimiter}
CREATE PROCEDURE {proc_name}([optional parameters])
BEGIN
    // procedure body...
    // procedure body...
END
{custom delimiter}
```

- `delimiter` - это команда, которая необходима для изменения разделителя SQL-инструкций с `;` на `//` во время определения процедуры. Это позволяет разделитель `;` использовать в теле процедуры для передачи на сервер.
- `proc_name` - уникальное имя хранимой процедуры, длиной не более 64 символа. Имена процедур не чувствительны к регистру, поэтому в одной схеме не может быть двух событий с именами `procname` и `ProcName`;
- в процедуру можно передать параметры (`optional_parametrs`)

IN - Параметр может ссылаться на процедуру. Значение параметра не может быть перезаписано процедурой.

OUT - Параметр не может ссылаться на процедуру, но значение параметра может быть перезаписано процедурой.

IN OUT - Параметр может ссылаться на процедуру, и значение параметра может быть перезаписано процедурой.
Для создания хранимой процедуры предназначен оператор CREATE PROCEDURE, после которого указывается имя процедуры. Давайте создадим процедуру, которая выводит текущую версию MySQL-сервера:

```
DELIMITER //

CREATE PROCEDURE my_version ()
BEGIN
  SELECT VERSION();
END //

-- CALL proc_name;
CALL my_version (); -- Вызов процедуры
```

После команды CREATE PROCEDURE указывается имя процедуры и круглые скобки, в которых
обычно указывают входящие и исходящие параметры. Мы рассмотрим их чуть позже в рамках текущего урока. Между ключевыми словами BEGIN и END размещаются SQL-команды, которые выполняются всякий раз при вызове хранимой процедуры.

Чтобы воспользоваться только что созданной хранимой процедурой, используем команду CALL, после которой указываем имя вызываемой процедуры:

```
CALL my_version ();
```

Мы получили текущую версию MySQL-сервера. Чтобы получить список хранимых процедур, можно воспользоваться командой:

```
SHOW PROCEDURE STATUS; -- Все процедуры

SHOW PROCEDURE STATUS LIKE 'my_version%'; -- Конкретная процедура
-- При использовании ключевого слова
-- LIKE можно вывести информацию только о тех процедурах,
-- имена которых удовлетворяют шаблону
```

Для удаления хранимых процедур и функций предназначены операторы DROP PROCEDURE и DROP FUNCTION. Давайте удалим процедуру my_version:

```
DROP PROCEDURE my_version;
```

Синтаксис команды допускает использование ключевого слова IF EXISTS:

```
DROP PROCEDURE IF EXISTS my_version;
```

В этом случае, если хранимой процедуры уже не существует, команда завершается без сообщения об ошибке.

Функции

Синтаксис:

```
DELIMITER {custom delimiter}
CREATE FUNCTION function_name [ (parameter datatype [, parameter datatype]) ]

RETURNS return_datatype

BEGIN

   declaration_section

   executable_section

END;
```

Пример:

```
CREATE FUNCTION get_version ()
RETURNS TEXT DETERMINISTIC
BEGIN
  RETURN VERSION();
END -- Процедура не до конца написана
```

Функция создается командой CREATE FUNCTION, после которой идет имя функции. Хранимая
функция встраивается в SQL-запросы, как обычная mysql-функция. Она должна возвращать значение. Ключевое слово RETURNS указывает возвращаемый тип, например TEXT мы можем заменить на VARCHAR(255). Ключевое слово DETERMINISTIC (дэтеминистик) сообщает, что результат функции детерминирован, т.е., при каждом вызове будет возвращаться одно и то же значение, и если его закешировать в рамках запроса, ничего страшного не произойдет. Если значения, которые возвращает функция, каждый раз различны, то перед DETERMINISTIC (дэтеминистик) следует добавить отрицание NOT. Далее следует тело функции, которое размещается между ключевыми словами BEGIN и END. Внутри тела обязательно должно присутствовать ключевое слово RETURN, которое возвращает результат вычисления. В данном случае мы просто возвращаем результат вызова mysql-функции VERSION().

Для вызова хранимой функции не требуется специальной команды, как в случае хранимых процедур. Порядок их вызова совпадает с порядком вызова встроенных функций MySQL:

```
SELECT get_version();
```

Основная трудность, которая возникает при работе с хранимыми процедурами и функциями,
заключается в том, что символ точки с запятой (;) используется в теле запроса для разделения
SQL-команд. Создание хранимой процедуры или функции — это тоже команда, которая тоже должна завершаться точкой с запятой. В результате возникает конфликт.

Чтобы его избежать, во всех клиентах предусмотрена возможность переназначать признак окончания запроса, в консольном клиенте mysql это осуществляется при помощи команды DELIMITER.

Давайте снова назначим разделителем два слеша:

```
DELIMITER //
```

Теперь мы можем воспользоваться новым разделителем:

```
-- Рабочий вариант
DROP FUNCTION IF EXISTS get_version;
DELIMITER //

CREATE FUNCTION get_version ()
RETURNS TEXT DETERMINISTIC
BEGIN
  RETURN VERSION();
END//

SELECT get_version();
```

Параметры процедур и функций

Хранимые процедуры и функции могут использовать параметры. Параметры могут передавать значения внутрь функции и извлекать результаты вычисления. Для этого каждый из параметров снабжается одним из атрибутов: IN, OUT или INOUT.
Параметр param_name предваряет одно из ключевых слов IN, OUT, INOUT, которые позволяют задать направление передачи данных:
● IN — данные передаются строго внутрь хранимой процедуры, но если параметру с данным
модификатором внутри функции присваивается новое значение, по выходу из нее оно не
сохраняется и параметр принимает значение, которое он имел до вызова процедуры.

● OUT — данные передаются строго из хранимой процедуры. Даже если параметр имеет
какое-то начальное значение, внутри хранимой процедуры оно не принимается во внимание.

С другой стороны, если параметр изменяется внутри процедуры, после ее вызова он имеет
значение, присвоенное ему внутри процедуры.
● INOUT — значение этого параметра как принимается во внимание внутри процедуры, так и
сохраняет свое значение по выходу из нее.
Атрибуты IN, OUT и INOUT доступны лишь для хранимой процедуры, в хранимой функции все
параметры всегда имеют атрибут IN.

Давайте сразу установим в качестве разделителя два слеша, для этого используем команду
DELIMITER. Создадим простейшую процедуру get_x, которая принимает единственный параметр value и устанавливает переменную.

```
DELIMITER //

DROP PROCEDURE IF EXISTS set_x//
CREATE PROCEDURE set_x (IN value INT)
BEGIN
  SET @x = value;
  SET value = value - 1000;

END//
SET @y = 10000//
CALL set_x(@y)//
SELECT @x, @y// x = 10000, y =  10000
```

спользование ключевого слова IN не обязательно — если ни один из атрибут не указан, СУБД MySQL считает, что параметр объявлен с атрибутом IN. В теле процедуры мы используем команду SET, чтобы создать пользовательскую переменную @x. Напоминаю, что наши собственные переменные создаются с использованием

символа @. При вызове хранимой процедуры мы можем передать в круглых скобках значение, которое будет использоваться вместо параметра внутри хранимой функции.

В отличие от пользовательской переменной @x, которая является глобальной и доступна как внутри хранимой процедуры set_x(), так и вне ее, параметры функции локальны и доступны для использования только внутри функции.

Хранимая процедура set_x() принимает единственный IN-параметр value, при помощи оператора SET его значение изменяется внутри функции. Однако после выполнения хранимой процедуры значение пользовательской переменной @y, переданной функции в качестве параметра, не изменяется. Если требуется, чтобы значение переменной менялось, необходимо объявить параметр процедуры с модификатором OUT.

При использовании модификатора OUT любые изменения параметра внутри процедуры отражаются на аргументе. Передача в качестве значения пользовательской переменной позволяет использовать результат процедуры для дальнейших вычислений. Однако передать значение внутрь функции при помощи OUT-параметра уже не получится.

```
DELIMITER //

DROP PROCEDURE IF EXISTS set_x//
CREATE PROCEDURE set_x (OUT value INT)
BEGIN
  SET @x = value;
  SET value = 1000;
END//
SET @y = 10000//
CALL set_x(@y)//
SELECT @x, @y// x = NULL, y = 1000
```

Чтобы через параметр можно было и передать значение внутрь процедуры, и получить значение, которое попадает в параметр в результате вычислений внутри процедуры, его следует объявить с атрибутом INOUT:

```
DELIMITER //

DROP PROCEDURE IF EXISTS set_x//
CREATE PROCEDURE set_x (INOUT value INT)
BEGIN
  SET @x = value;
  SET value = value - 1000;
END//
SET @y = 10000//
```

```
CALL set_x(@y)//
SELECT @x, @y// -- x = 10000, y = 9000
```

До этого момента локальные переменные в хранимой процедуре или функции объявлялись как входящие или исходящие параметры, однако это не всегда удобно. Часто требуется локальная переменная без необходимости передавать или возвращать с ее
помощью какие-либо значения.

Объявить такую переменную можно при помощи команды DECLARE. Один оператор DECLARE
позволяет объявить сразу несколько переменных одного типа, причем необязательное слово
DEFAULT позволяет назначить инициирующее значение. Те переменные, для которых не указывается ключевое слово DEFAULT, можно инициировать при помощи команды SET. Позже остановимся на этом подробнее. Команда DECLARE может появляться только внутри блока BEGIN...END, область видимости объявленной переменной также ограничена этим блоком.

```
DELIMITER //

DROP PROCEDURE IF EXISTS declare_var//
CREATE PROCEDURE declare_var ()
BEGIN
  DECLARE var TINYTEXT DEFAULT 'внешняя переменная';
  BEGIN
    DECLARE var TINYTEXT DEFAULT 'внутренняя переменная';
    SELECT var;
  END;
  SELECT var;
END//
CALL declare_var()//
```

Это означает, что в разных блоках BEGIN...END могут быть объявлены переменные с одинаковым именем, и действовать они будут только в рамках одного блока, не пересекаясь с переменными других.

Ветвление

Оператор IF позволяет реализовать ветвление программы по условию. IF принимает значение либо TRUE (истину), либо FALSE (ложь). В MySQL TRUE и FALSE — константы для целочисленных значений 1 и 0. Если логическое выражение истинно, IF

выполняет SQL-выражения, которые размещаются в теле команды между ключевыми словами THEN и END IF.

```
DELIMITER //

DROP PROCEDURE IF EXISTS format_now//
CREATE PROCEDURE format_now (format CHAR(4))
BEGIN
  IF(format = 'date') THEN
    SELECT DATE_FORMAT(NOW(), "%d.%m.%Y") AS format_now;
  END IF;
  IF(format = 'time') THEN
    SELECT DATE_FORMAT(NOW(), "%H:%i:%s") AS format_now;
  END IF;
END//

CALL format_now('date')//
CALL format_now('time')//
```

Команда IF поддерживает ключевое слово ELSE. Давайте перепишем процедуру format_now с
использованием ELSE:

```
DELIMITER //

DROP PROCEDURE IF EXISTS format_now//
CREATE PROCEDURE format_now (format CHAR(4))
BEGIN
  IF(format = 'date') THEN
    SELECT DATE_FORMAT(NOW(), "%d.%m.%Y") AS format_now;
  ELSE
    SELECT DATE_FORMAT(NOW(), "%H:%i:%s") AS format_now;
  END IF;
END//


CALL format_now('date')//
CALL format_now('time')//
```

Если параметр format равен 'date', условие в IF является истинным, выполняется первый блок,
выводящий текущую дату. Если аргумент принимает любое другое условие, условие в IF является ложным и запрос выполняется в блоке ELSE.

Циклы

Циклы являются важнейшей конструкцией, без которой хранимые процедуры и функции не имели бы достаточно функциональности. MySQL предоставляет три цикла: **while**, **repeat** и **loop**. Их можно использовать в теле хранимой процедуры или функции, т.е., между ключевыми словами BEGIN и END.

```
DELIMITER //

CREATE PROCEDURE while_cycle ()
BEGIN
  DECLARE i INT DEFAULT 3;
  WHILE i > 0 DO
    SELECT NOW();
    SET i = i - 1;
  END WHILE;
END//
```

Здесь в процедуре while_cycle используется цикл WHILE для трехкратного вывода даты и времени. Цикл начинается с ключевого слова WHILE, после которого следует условие. Условие вычисляется на каждой итерации цикла: если оно возвращает истину (TRUE), очередная итерация выполняется, если при очередной проверке оно будет ложным (FALSE), цикл завершит работу. Чтобы не создать бесконечный цикл, условие подбирается таким образом, чтобы рано или поздно оно становилось ложным и цикл прекращал свою работу. Цикл while, в свою очередь, сам имеет тело, начало которого обозначается ключевым словом DO, а завершение — ключевым словом END WHILE. Все команды, которые располагаются между этими ключевыми словами, выполняются на каждой итерации цикла. Обратите внимание: перед циклом мы заводим переменную i, которой при помощи ключевого слова DEFAULT устанавливаем значение 3.
На каждой итерации мы уменьшаем значение i на единицу: пока i больше нуля, условие цикла остается истинным. Как только значение уменьшается до 0, условие возвращает FALSE и цикл завершает работу. Таким образом, текущая дата будет выведена только три раза. Давайте в этом убедимся.

```
CALL while_cycle()// --2023-03-19 15:37:53
```

Количество повторов не обязательно задавать внутри хранимой процедуры. Например, мы можем задать его в качестве входящего параметра.

```
DELIMITER //
DROP PROCEDURE IF EXISTS while_cycle;
```

```
CREATE PROCEDURE while_cycle (IN num INT)
BEGIN
  DECLARE i INT DEFAULT 0;
  IF (num > 0) THEN
    WHILE i < num DO
      SELECT NOW();
      SET i = i + 1;
    END WHILE;
  ELSE
    SELECT 'Ошибочное значение параметра';
  END IF;
END//

CALL while_cycle(2)//
```

Итак, у нас выводится только две даты. Для досрочного выхода из цикла предназначен оператор LEAVE. Давайте ограничим цикл в процедуре NOWN только двумя итерациями, т.е., сколько бы выводов пользователь ни заказывал, максимальное количество, которое будет доступно — 2 .В тело цикла добавляется дополнительное if-условие, не допускающее достижение счетчика i значения 2. Как только условие срабатывает, выполняется команда LEAVE. Циклы можно вкладывать друг в друга, поэтому, чтобы команда LEAVE понимала, какой из циклов следует останавливать, ей всегда передается метка цикла, в данном случае cycle. Эту метку мы должны поместить перед ключевым словом WHILE и после ключевого слова END WHILE. Давайте запросим заведомо огромное значение, например 1000:

```
DELIMITER //
DROP PROCEDURE IF EXISTS while_cycle//
CREATE PROCEDURE while_cycle (IN num INT)
BEGIN
  DECLARE i INT DEFAULT 0;
  IF (num > 0) THEN
    cycle: WHILE i < num DO
    IF i >= 2 THEN LEAVE cycle;
    END IF;
    SELECT NOW();
    SET i = i + 1;
    END WHILE cycle;
  ELSE
    SELECT 'Ошибочное значение параметра';
  END IF;
END//
CALL while_cycle(1000)//
```

Как видим, выводятся только две даты, у нас сработал досрочный выход из цикла. Еще один оператор, выполняющий досрочное прекращение цикла — ITERATE. В

отличие от
оператора LEAVE, ITERATE не прекращает выполнение цикла, он лишь досрочно прекращает
текущую итерацию.
Давайте создадим хранимую процедуру, которая продемонстрирует ITERATE на практике:

```
DELIMITER //
DROP PROCEDURE IF EXISTS numbers_string//
CREATE PROCEDURE numbers_string (IN num INT)
BEGIN
  DECLARE i INT DEFAULT 0;
  DECLARE bin TINYTEXT DEFAULT '';
  IF (num > 0) THEN
    cycle : WHILE i < num DO
    SET i = i + 1;
    SET bin = CONCAT(bin, i);
    IF i > CEILING(num / 2) THEN ITERATE cycle;
 -- CEILING: возвращает наименьшее целое число
    END IF;
    SET bin = CONCAT(bin, i);
    END WHILE cycle;
    SELECT bin;
  ELSE
    SELECT 'Ошибочное значение параметра';
  END IF;
END//
CALL numbers_string(9)//
```

Внутри цикла счетчик i пробегает значения от 1 до 9, на каждой итерации значение счетчика
добавляется к строке bin. Если if-условие ложное, то значение добавляется два раза, если истинное, срабатывает оператор ITERATE и текущая итерация завершается досрочно. Именно поэтому в результатах мы видим удвоенные цифры до 5 и одиночные цифры после 5.

Оператор **REPEAT** похож на оператор WHILE.

```
DELIMITER //
DROP PROCEDURE IF EXISTS repeat_cycle//
CREATE PROCEDURE repeat_cycle ()
BEGIN
  DECLARE i INT DEFAULT 3;
  REPEAT
    SELECT NOW();
    SET i = i - 1;
  UNTIL i <= 0
```

```
  END REPEAT;
END//

CALL repeat_cycle()//
```

Однако условие для покидания цикла располагается не в начале тела цикла, а в конце. В результате тело цикла в любом случае выполняется хотя бы один раз. В конце цикла после ключевого слова UNTIL располагается условие; если оно истинно, работа цикла прекращается, если ложно, происходит еще одна итерация. Эта хранимая процедура должна выполняться в теле цикла три раза.

Цикл **LOOP**, в отличие от операторов WHILE и REPEAT, не имеет условий выхода. Поэтому он
должен обязательно иметь в составе оператор LEAVE.

```
DELIMITER //
DROP PROCEDURE IF EXISTS loop_cycle//
CREATE PROCEDURE loop_cycle ()
BEGIN
  DECLARE i INT DEFAULT 3;
  cycle: LOOP
    SELECT NOW();
    SET i = i - 1;
    IF i <= 0 THEN LEAVE cycle;
    END IF;
  END LOOP cycle;
END//

CALL loop_cycle()//
```

Так как мы используем оператор LEAVE, мы должны разместить перед ключевым словом LOOP и после END LOOP метку. Здесь она называется cycle.

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.

Книги:

- “Изучаем SQL”, книга Бейли Л.
- Алан Бьюли "Изучаем SQL" (2007)
- Энтони Молинаро "SQL. Сборник рецептов" (2009)

1. Виктор Гольцман “MySQL 5.0. Библиотека программиста”
2. “Изучаем SQL”, книга Бейли Л.
3. https://www.webmasterwiki.ru/MySQL
4. Поль Дюбуа “MySQL. Сборник рецептов”