MySQL

ПРЕДСТАВЛЕНИЯ, СОБЫТИЯ И ПОДГОТОВЛЕННЫЕ ЗАПРОСЫ

СОДЕРЖАНИЕ

ПРЕДСТАВЛЕНИЯ

ВВЕДЕНИЕ

- Мы уже умеем пользоваться функциями и процедурами, которые позволяют реализовать очень сложную логику и в качестве результата возвращать либо одно значение (у функций), либо набор значений или даже несколько наборов значений (у процедур). Однако нам не всегда надо реализовывать именно логику - очень часто нам надо написать очень сложный, но все же один запрос, который может часто повторяться в рамках нашего приложения. Для этого в MySQL существуют т.н. представления - объекты базы данных, которые по своей сути являются сохраненными запросами для выборки данных (при помощи оператора **SELECT**) из какой-либо таблицы или нескольких таблиц. Само по себе представление не хранит данные, оно является посредником при выборке этих данных.

- Можно выделить три основных преимущества, которые мы можем получить при помощи представлений:

  - ➢ Упрощение работы с базой данных. Если у нас в приложении есть один сложный запрос, который мы повторяем многократно, то разумнее и легче оформить его в качестве представления, а не переписывать в нескольких местах один и тот же огромный SQL код.

  - ➢ Усиливает безопасность вашего приложения. В определенной таблице могут храниться очень чувствительные данные, открытие которых может нарушать закон, банковскую тайну и т.д. - с помощью представления можно позволить выбрать только ограниченные данные.

  - ➢ Позволяет работать тем приложением, которые устарели, но по какой-то причине не могут быть обновлены. Если старая большая таблица была разделена на несколько меньших, то мы может сделать представление, которое будет представлять выборку из этих меньших таблиц, к которой устаревшее приложение сможет обращаться как к единой таблице.

- Кроме вышеупомянутых преимуществ, которые связаны с выборкой, представления можно использовать для вставки,

обновления и удаления только тех данных, которые разрешены (перечисленные операции будут осуществляться не напрямую по отношению к таблице и при посредничестве представления). Хотя в данном случае должны выполняться некоторые условия, которые мы рассмотрим позднее.

- При использование представлений есть ряд ограничений, которые мы должны строго соблюдать:

  - ➢ Таблицы, к которым производится запрос в рамках представления, должны реально существовать и не быть временными (**TEMPORARY**).

  - ➢ Для представлений нельзя создавать триггеры (**TRIGGER**).

  - ➢ Для запросов внутри представлений нельзя использовать переменные - как пользовательские, так и системные.

  - ➢ Длина псевдонимов для столбцов, которые будет возвращать представление, не должны превышать 64 символа.

- Далее мы перейдем непосредственно к созданию и использованию представлений, но сначала представим, что мы разрабатываем банковское приложение, внутри которого три таблицы - люди (persons), счета (accounts) и платежи (payments). Создадим же эти таблицы и заполним данными:

```
-- создание таблиц
CREATE TABLE persons (
    id INT UNSIGNED NOT NULL AUTO_INCREMENT PRIMARY KEY,
    name VARCHAR(255) NOT NULL,
    citizenship CHAR(2) NOT NULL,
    personal_code VARCHAR(255) NOT NULL,
    phone VARCHAR(255) NOT NULL
);

CREATE TABLE accounts (
    id INT UNSIGNED NOT NULL AUTO_INCREMENT PRIMARY KEY,
    person_id INT UNSIGNED NOT NULL,
    status VARCHAR(255) NOT NULL,
    total DECIMAL(20,2),
    currency CHAR(3) NOT NULL,
    CONSTRAINT person_fk FOREIGN KEY(person_id)
REFERENCES persons(id)
);
```

```
CREATE TABLE payments (
    id INT UNSIGNED NOT NULL AUTO_INCREMENT PRIMARY KEY,
    account_id INT UNSIGNED NOT NULL,
    payment_sum DECIMAL(20,2),
    CONSTRAINT account_fk FOREIGN KEY(account_id) REFERENCES
accounts(id)
);

-- заполнение таблиц данными
INSERT INTO persons VALUES
(1, 'John Smith', 'US', '110789-10345', '+1 23456789'),
(2, 'Andrew Norton', 'FR', '040765-85392', '+33 87654321'),
(3, 'Mary Sue', 'DE', '170199-23467', '+49 43215678');

INSERT INTO accounts VALUES
(1, 1, 'VIP', 100000, 'USD'),
(2, 2, 'VIP', 999000, 'EUR'),
(3, 2, 'REGULAR', 3500, 'USD'),
(4, 3, 'REGULAR', 50, 'GBP'),
(5, 3, 'REGULAR', 300, 'EUR');

INSERT INTO payments VALUES
(1, 1, 1000),
(2, 2, 3000),
(3, 3, 50),
(4, 4, 10),
(5, 5, 100);
```

Таблица persons

id	name	citizenship	personal_code	phone
1	John Smith	US	110789-10345	+1 23456789
2	Andrew Norton	FR	040765-85392	+33 87654321
3	Mary Sue	DE	170199-23467	+49 43215678

Таблица accounts

id	person_id	status	total	currency
1	1	VIP	100000.00	USD
2	2	VIP	999000.00	EUR
3	2	REGULAR	3500.00	USD
4	3	REGULAR	50.00	GBP
5	3	REGULAR	300.00	EUR

Таблица payments

id	account_id	payment_sum
1	1	1000.00
2	2	3000.00
3	3	50.00
4	4	10.00
5	5	100.00

СОЗДАНИЕ И ИСПОЛЬЗОВАНИЕ ПРЕДСТАВЛЕНИЙ

- Создание представления начинается с конструкции **CREATE VIEW**. Однако после ключевого слова **CREATE** можно вместить целых три дополнительных необязательных конструкции:

  ➢ **OR REPLACE** - есть представление с таким названием уже существует, оно будет перезаписано.

  ➢ **ALGORITHM** - может быть равен **TEMPTABLE** (при запросе MySQL будет всегда создавать временную таблицу), MERGE (запрос к представлению будет объединен с запросом внутри представления), **UNDEFINED** (значение по умолчанию - MySQL сама решит, что ей делать с запросом в каждый конкретный момент времени). **TEMPTABLE** необходим, если представление участвует в агрегатном запросе на группировку - если там будет неправильное количество столбцов, возникнет ошибка.

  ➢ **DEFINER** - можем явно указать создателя представления.

  ➢ **SQL SECURITY** - определяет, для кого будут проверяться права на запрос внутри представления - для создателя или для того, кто данное представление вызывает.

  После названия представления можно указать псевдонимы для выбираемых столбцов, но обычно это делается в самом запросе (после **SELECT**). Уже после запроса можно указать конструкцию **WITH CHECK OPTION** - используется в том случае, если мы захотим производить вставки или обновления данных при помощи представления (мы рассмотрим эту особенность подробнее в соответствующем подразделе):

```
CREATE
    [OR REPLACE]
    [ALGORITHM = {UNDEFINED | MERGE | TEMPTABLE}]
    [DEFINER = user]
    [SQL SECURITY { DEFINER | INVOKER }]
    VIEW название_представления [(список_названий_столбцов)]
    AS запрос_на_выборку
[WITH CHECK OPTION];
```

- Пришло время написать наше первое представление. Применим наши знания, чтобы получить все суммы платежей (таблица payments) с соответствующими валютами (таблица accounts) и именами владельцев счетов, с которых были совершены платежи (т.е. мы делаем запрос сразу в три таблицы):

```
CREATE VIEW get_payments_info
AS
SELECT per.name, acc.currency, pay.payment_sum
FROM persons per
JOIN accounts acc ON (per.id = acc.person_id)
JOIN payments pay ON (acc.id = pay.account_id);
```

- Теперь воспользуемся нашим новым представлением - запросы в него выглядят как запросы в таблицу, только вместо название таблицы - название представления:

```
SELECT * FROM get_payments_info;
```

name	currency	payment_sum
John Smith	USD	1000.00
Andrew Norton	EUR	3000.00
Andrew Norton	USD	50.00
Mary Sue	GBP	10.00
Mary Sue	EUR	100.00

- Из представление необязательно получать все данные, можно выбрать значения только тех столбцов, которые нам необходимы:

```
SELECT currency, payment_sum FROM get_payments_info;
```

currency	payment_sum
USD	1000.00
EUR	3000.00
USD	50.00
GBP	10.00
EUR	100.00

- Кроме того, данные, возвращаемые из представления, могут фильтроваться с помощью конструкции WHERE, например, получим платежи только в евро (EUR):

```
SELECT currency, payment_sum FROM get_payments_info
WHERE currency = 'EUR';
```

currency	payment_sum
EUR	3000.00
EUR	100.00

- Наконец, представления и их данные могут участвовать в других запросах. Например, мы хотим получить всю информацию о заказах, но дополнительно хотим получить гражданство и персональный код каждого плательщика:

```
SELECT p.citizenship, p.personal_code, gpi.*
FROM persons p
JOIN get_payments_info gpi ON p.name = gpi.name;
```

citizenship	personal_code	name	currency	payment_sum
US	110789-10345	John Smith	USD	1000.00
FR	040765-85392	Andrew Norton	EUR	3000.00
FR	040765-85392	Andrew Norton	USD	50.00
DE	170199-23467	Mary Sue	GBP	10.00
DE	170199-23467	Mary Sue	EUR	100.00

- Надо заметить, что внутри представления также можно вставлять фильтрацию и использовать агрегатные функции. Чтобы это продемонстрировать, сначала создадим представление с фильтрацией, которое получает персональные данные только для тех людей, которые имеют элитные счета (со статусом, равным VIP):

```
CREATE VIEW get_vip_persons_data
AS
SELECT name, phone, personal_code
FROM persons WHERE id IN (
    SELECT person_id FROM accounts
    WHERE status = 'VIP'
);
```

Получим данные элитных клиентов из нашего нового представления:

```
SELECT name, phone, personal_code FROM get_vip_persons_data;
```

name	phone	personal_code
John Smith	+1 23456789	110789-10345
Andrew Norton	+33 87654321	040765-85392

- В конце этой подглавы создадим представление с агрегатными функциями и группировкой - получим среднюю сумму платежа (таблица payments) для каждой валюты (можно получить ее из таблицы accounts):

```
CREATE VIEW get_avg_payments_sum
AS
SELECT acc.currency, AVG(pay.payment_sum)
FROM accounts acc
JOIN payments pay ON (acc.id = pay.account_id)
GROUP BY acc.currency;
```

Теперь задействуем нашу процедуру для выборки средних значений платежа:

```
SELECT * FROM get_avg_payments_sum;
```

currency	AVG(pay.payment_sum)
USD	525.000000
EUR	1550.000000
GBP	10.000000

ИСПОЛЬЗОВАНИЕ ПРЕДСТАВЛЕНИЙ ДЛЯ МОДИФИКАЦИИ ДАННЫХ

- Как уже упоминалось во введении, представления можно использовать не только для просмотра данных, которые находятся в неких таблицах, но и для обновления, добавления и удаления этих данных. Но не каждое представление позволяет такие модификации - для этого должен выполняться ряд условий:

  - ➢ Можно взаимодействовать только с теми столбцами, которые явно возвращает представление. Это значит, что можно обновлять, вставлять и фильтровать столбцы не всей таблицы или всех таблиц, которые используются в представлении, а только тот ограниченный набор, который мы видим при выборке данных из представления.

  - ➢ Представление не должно возвращать значения, полученные при помощи таких агрегатных функций как **SUM**, **MIN**, **MAX**, **COUNT**, **AVG** и т.д., а также значения, полученные при помощи оконных функций.

  - ➢ Не должно возвращаться значений, которые получены с помощью выборок после оператора **SELECT**, например, таких:

```
SELECT
    column_name,
    (SELECT other_column FROM other_table LIMIT 1)
FROM some_table;
```

  - ➢ Внутри представления не должны использоваться следующие команды - **DISTINCT**, **GROUP BY**, **HAVING**, **UNION** или **UNION ALL**.

  - ➢ Внутри представление не должно быть запросов в другие представление, которые не разрешены для модификации.

- ➢ Если в представлении используется соединение таблиц, то для разрешения модификации это может быть только внутреннее объединение (**INNER JOIN** или просто **JOIN**).

- ➢ Чтобы выполнить вставку данных, представление не должно возвращать значение одного и того же столбца (с разными псевдонимами) несколько раз.

- ➢ Нельзя явно указывать **ALGORITHM = TEMPTABLE** при создании представления.

- Сначала рассмотрим самые примитивные случаи модификации данных и создадим три представления, каждая из которые получает абсолютно все данные из какой-либо одной доступной нам таблицы (persons, accounts, payments):

```
-- все данные из таблицы persons
CREATE VIEW get_all_persons_info
AS
SELECT id, name, citizenship, personal_code, phone
FROM persons;

-- все данные из таблицы accounts
CREATE VIEW get_all_accounts_info
AS
SELECT id, person_id, status, total, currency
FROM accounts;

-- все данные из таблицы payments
CREATE VIEW get_all_payments_info
AS
SELECT id, account_id, payment_sum
FROM payments;
```

- Сперва произведем вставку (**INSERT**) в представление *get_all_persons_info*, которое отвечает за таблицу persons:

```
INSERT INTO get_all_persons_info(
    id, name, citizenship, personal_code, phone
) VALUES (
    4, 'Michael Brown', 'US', '230595-99999', '+1 33445566'
);
```

Если мы заглянем в таблицу persons, то сможем убедиться, что туда был успешно добавлен человек с именем Michael Brown:

id	name	citizenship	personal_code	phone
1	John Smith	US	110789-10345	+1 23456789
2	Andrew Norton	FR	040765-85392	+33 87654321
3	Mary Sue	DE	170199-23467	+49 43215678
4	Michael Brown	US	230595-99999	+1 33445566

- Далее произведем обновление (**UPDATE**) данных представления *get_all_accounts_info* (таблица accounts) - добавим 1000 единиц счету, который имеет person_id = 2 (Andrew Norton):

```
UPDATE get_all_accounts_info
SET total = total + 1000
WHERE person_id = 2;
```

id	person_id	status	total	currency
1	1	VIP	100000.00	USD
2	2	VIP	1000000.00	EUR
3	2	REGULAR	4500.00	USD
4	3	REGULAR	50.00	GBP
5	3	REGULAR	300.00	EUR

- А теперь произведем удаление (**DELETE**) из представления *get_all_payments_info* (отвечает за таблицу payments) - самый маленький платеж с id = 4:

```
DELETE FROM get_all_payments_info WHERE id = 4;
```

id	account_id	payment_sum
1	1	1000.00
2	2	3000.00
3	3	50.00
5	5	100.00

- После рассмотрения тривиальных случаев перейдем к более продвинутым вариантам модификации. Вспомним, что у нас есть представление *get_payments_info*, которое делает запрос одновременно в три таблицы (при помощи **JOIN**) и возвращает по одному столбцу из каждой из этих таблиц (persons - name, accounts - currency, payments - payment_sum). Попытаемся же обновить поле *name* таблицы *persons* при помощи представления - переименуем Mary Sue в Mary Jane.

```
UPDATE get_payments_info
SET name = 'Mary Jane'
WHERE name = 'Mary Sue';
```

id	name	citizenship	personal_code	phone
1	John Smith	US	110789-10345	+1 23456789
2	Andrew Norton	FR	040765-85392	+33 87654321
3	Mary Jane	DE	170199-23467	+49 43215678
4	Michael Brown	US	230595-99999	+1 33445566

- Теперь мы попытаемся переименовать человека обратно, но уже при помощи представления *get_vip_persons_data*, которое в числе прочих возвращает столбец *name*.

```
UPDATE get_vip_persons_data
SET name = 'Mary Sue'
WHERE name = 'Mary Jane';
```

Запрос выполнится без ошибок. Однако если мы проверим таблицу *persons*, то обнаружим, что данные не изменились. Дело в том, что представление *get_vip_persons_data* внутри себя имеет фильтрацию по статусу - возвращает только людей со статусом VIP. Mary Jane не имеет такого статуса, поэтому она не может быть обновлена. Если же мы попробуем поменять того человека, у которого такой статус есть (например, Andrew Norton), то обновление пройдет успешно:

```
UPDATE get_vip_persons_data
SET name = 'Larry Norton'
WHERE name = 'Andrew Norton';
```

id	name	citizenship	personal_code	phone
1	John Smith	US	110789-10345	+1 23456789
2	Larry Norton	FR	040765-85392	+33 87654321
3	Mary Jane	DE	170199-23467	+49 43215678
4	Michael Brown	US	230595-99999	+1 33445566

- В следующем примере попытаемся разобрать тот случай, когда мы пытаемся обновить значение столбца, который не упомянут при выборке данных представления. Например, представление из прошлого примера *get_vip_persons_data* возвращает значения из столбцов name, personal_code и phone в рамках таблицы persons. Попытаемся обновить столбец citizenship для персоны с name = John Smith (у него есть счета со статусом VIP, поэтому он поддерживается этим представлением).

```
UPDATE get_vip_persons_data
SET citizenship = 'FR'
WHERE name = 'John Smith';
```

Если мы выполним этот запрос, то мы не только не увидим обновленных данных в таблице, но и получив в ответ ошибку *ERROR 1054 (42S22): Unknown column 'citizenship' in 'field list'*. Таким образом, запомним, что при модификации представлений можно обращаться только к тем столбцам, которые это представление явно возвращает.

- Напоследок попытаемся обновить данные в представлении *get_avg_payments_sum*, которая возвращает валюты и среднее значение платежей для этих валют. Попытаемся поменять все счета в долларах (USD) на счета в фунтах (GBP):

```
UPDATE get_avg_payments_sum
SET currency = 'GBP'
WHERE currency = 'USD';
```

В ответ мы получим закономерную ошибку *ERROR 1288 (HY000): The target table get_avg_payments_sum of the UPDATE is not updatable*. Это означает, что данное представление вообще нельзя модифицировать. Дело в том, что там применяется агрегатная функция **AVG**. Даже несмотря на то, что обновляемый

столбец не используется совместно с этой функцией, использование **AVG** автоматически делает все представление необновляемым.

ИСПОЛЬЗОВАНИЕ КОНСТРУКЦИИ WITH CHECK OPTION

- В предыдущей главе при модификациях представлений мы часто сталкивались с ограничениями, которые не давали нам осуществить задуманное. Однако это были чисто технические ограничение (отсутствие необходимых столбцов, применение агрегатных функций и т.д.). Есть случаи, когда нам необходимо запретить модификацию на логическом уровне. Например, мы хотим, чтобы с помощью представления было бы нельзя вставлять в таблицу persons людей, которые являются гражданами определенной страны. Теми средствами, которыми мы обладаем на данный момент, сделать это достаточно сложно. Однако MySQL предлагает дополнительную конструкцию **WITH CHECK OPTION**, которая ставится в конце объявления представления и которая помогает накладывать логические ограничения на модификацию представлений.

- Конструкция **WITH CHECK OPTION** актуальна, если внутри представление используется фильтрация при помощи **WHERE**. Если при модификации (удаление, добавление или обновление) новые данные могут привести к несоответствию с фильтрацией, то модификация будет отклонена и мы получим в ответ ошибку.

- На основе нашей таблицы persons создадим представление, которое соответствует тому, о чем мы говорили в самом начале - не позволяет вставлять в себя граждан, например, Дании (DK). Также дополним наше новое представление конструкцией **WITH CHECK OPTION**.

```
CREATE VIEW get_all_non_danish_persons
AS
SELECT id, name, citizenship, personal_code, phone
FROM persons
WHERE citizenship != 'DK'
WITH CHECK OPTION;
```

- Теперь при помощи новой процедуры попробуем занести в таблицу persons человека с датским гражданством и посмотри, что из этого выйдет.

```
INSERT INTO get_all_non_danish_persons (
    id, name, citizenship, personal_code, phone
) VALUES (
    5, 'Mads Larsson', 'DK', '22031997-14722', '+45 92452629'
);
```

Ожидаемо данные не будут вставлены, а мы в ответ получим следующую ошибку - *ERROR 1369 (HY000): CHECK OPTION failed 'views.get_all_non_danish_persons'*.

- Далее попытаемся изменить уже существующую строку внутри таблицы persons с присвоением датского гражданства человеку с id = 1 (John Smith).

```
UPDATE get_all_non_danish_persons
SET citizenship = 'DK'
WHERE id = 1;
```

Мы столкнемся с той же самой ошибкой, с которой уже встречались при попытке вставить данные: *ERROR 1369 (HY000): CHECK OPTION failed 'views.get_all_non_danish_persons'*

- В конце этой подглавы упомянем, что конструкция **WITH CHECK OPTION** может быть дополнена ключевым словом **LOCAL** или **CASCADED** - **WITH LOCAL CHECK OPTION** или **WITH CHECK CASCADED OPTION**. **LOCAL** означает, что проверка будет происходить только для данного представления, а **CASCADED** - для все используемых внутренних представлений, если таковые имеются.

УДАЛЕНИЕ ПРЕДСТАВЛЕНИЙ

- Для удаления используется команда **DROP VIEW** - она может быть дополнена конструкцией **IF EXISTS**, которая обезопасит нас от ошибки в том случае, если представление не существует.

```
-- удаление существующего представления без проверки
DROP VIEW get_all_non_danish_persons ;

-- удаление существующего представления с проверкой
DROP VIEW IF EXISTS get_all_payments_info;

-- удаление несуществующего представления с проверкой
DROP VIEW IF EXISTS undefined_view;
```

СОБЫТИЯ

ВВЕДЕНИЕ

- Очень часто в вычислительных системах необходимо делать по расписанию определенные действия - делать резервную копию, сохранять некие обобщающие данные за прошедший период или удалять устаревшую информацию. На операционной системе Windows за это отвечает планировщик задач (Task Manager), который позволяет запланировать запуск той или иной программы. В операционных системах на ядре Linux для этих же целей используется планировщик cron. Схожий инструментарий предоставляет и MySQL - внутри этой СУБД доступен инструмент под названием события (**EVENT**), который помогает назначить какие-либо действия на определенное время, либо исполняет эти действия периодически.

- Перед тем, как создавать события, надо убедиться в том, что в нашей СУБД включен планировщик событий - механизм, который следит, чтобы события отслеживались и запускались. Проверить его состояние можно следующим образом:

```
SELECT @@global.event_scheduler;
```

Если вернется значение ON - планировщик включен, если OFF - соответственно планировщик выключен.

@@global.event_scheduler
ON

Если мы хотим включить или выключить планировщик, то следует применять следующие способы:

```
-- выключение планировщика
SET GLOBAL event_scheduler = OFF;

-- включение планировщика
SET GLOBAL event_scheduler = ON;
```

- Создание события происходит при помощи ключевой конструкции **CREATE EVENT**, после которой можно разместить конструкцию IF **NOT EXISTS**, а уже потом будет следовать название события.

```
CREATE
    [DEFINER = пользователь]
    EVENT
    [IF NOT EXISTS]
    название_события
    ON SCHEDULE расписание
    [ON COMPLETION [NOT] PRESERVE]
    [ENABLE | DISABLE]
    [COMMENT 'строка']
    DO тело_события;

расписание: {
    AT временная_метка [+ INTERVAL интервал] ...
  | EVERY интервал
    [STARTS временная_метка [+ INTERVAL интервал] ...]
    [ENDS временная_метка [+ INTERVAL интервал] ...]
}

интервал:
    количество {YEAR | QUARTER | MONTH | DAY | HOUR | MINUTE |
                WEEK | SECOND | YEAR_MONTH | DAY_HOUR | DAY_MINUTE |
                DAY_SECOND | HOUR_MINUTE | HOUR_SECOND | MINUTE_SECOND}
```

Попытаемся разъяснить малопонятные моменты этой схемы:

- По умолчанию событие должно запуститься (или начать ждать того времени, когда оно сможет запуститься согласно расписанию) сразу же после создания. Это можно изменить при помощи ключевого слова **DISABLE**. Если же мы хотим явно указать поведение по умолчанию, то следует прописать ключевое слово **ENABLE**.

- Временная метка означает т.н. UNIX время - количество секунд с начала 1 января 1970 года во временной зоне UTC. Если мы хотим получит временную метку на данный момент, то можем подставить ключевое слово **CURRENT_TIMESTAMP**.

- В расписании нам доступны две опции - при использовании ключевого слова **AT** событие выполнится один раз, а при использовании ключевого слова **EVERY** событие будет исполнятся постоянно с определенным интервалом, но мы может назначить этому событию начальное время (**STARTS**) и конечное время (**ENDS**).

- По умолчанию событие, которое закончило свое выполнение, автоматически удаляется. Мы можем изменить это поведение проставив директиву **ON COMPLETION PRESERVE**. В этом случае событие не будет удалено, но выполняться больше не будет.

- Интервал всегда создается следующим образом - сначала идет ключевое слово **INTERVAL** (в самом начале после **AT** или **EVERY** слово **INTERVAL** писать не надо), затем количество временных единиц, а в конце тип этих самых единиц, например, **INTERVAL 1 DAY**, **INTERVAL 5 MINUTE**, **INTERVAL '6:15' HOUR_MINUTE**.

- Как уже было сказано в объяснении к схеме, событие может быть активированным или неактивированным. Если событие активировано, а мы хотим его остановить, необходимо выполнить следующую команду (подразумевается, что вместо event_name надо подставить название нужного нам события):

```
ALTER EVENT event_name DISABLE;
```

Команда, если мы хотим возобновить работу события:

```
ALTER EVENT event_name ENABLE;
```

- В дальнейшем мы поработаем с событиями с практической стороны, но сначала создадим 3 таблицы, которые мы будем использовать для примеров - sessions, transactions и accounting (данными их заполнять не будем, т.к. это понадобится сделать не сейчас, а в нужный момент).

```
CREATE TABLE sessions (
    id INT UNSIGNED NOT NULL AUTO_INCREMENT PRIMARY KEY,
    name VARCHAR(255) NOT NULL,
    role VARCHAR(255) NOT NULL,
    logged_in DATETIME NOT NULL
);

CREATE TABLE transactions (
    id INT UNSIGNED NOT NULL AUTO_INCREMENT PRIMARY KEY,
    total_sum DECIMAL(20,2) NOT NULL,
    paid_at DATETIME NOT NULL
);

CREATE TABLE accounting (
    id INT UNSIGNED NOT NULL AUTO_INCREMENT PRIMARY KEY,
    all_tansactions_sum DECIMAL(20,2) NOT NULL,
    calculated_at DATETIME NOT NULL
);
```

СОЗДАНИЕ И ЗАПУСК СОБЫТИЙ

- Создадим наше первое событие, которое заглядывает в таблицу *transactions*, получает сумму всех транзакций, дата оплаты которых стоит позже самой поздней даты в таблице *accounting*, а затем записывает эту общую сумму в таблицу *accounting* и ставит дату и время этой записи. По сути, это архивирование суммы всех оплат за каждый определенной период - может быть полезно для бухгалтерии и финансового контроля. Теоретически таким периодом должен быть месяц или хотя бы день, но мы в целях экономии времени установим интервалом 1 минуту.

- Прежде чем создать события, учтем, что внутри события могут быть практически любые MySQL команды, которые, как известно, разделяются точкой с запятой. Поэтому, чтобы избежать ошибок, мы должны менять разделить (**DELIMITER**) перед созданием события и возвращать его обратно после создания.

```
DELIMITER |


CREATE EVENT IF NOT EXISTS save_accounting_info
ON SCHEDULE EVERY 1 MINUTE
DO
BEGIN
    DECLARE transactions_sum DECIMAL(20,2) DEFAULT 0;

    SELECT IFNULL(SUM(total_sum), 0)
    INTO transactions_sum
    FROM transactions WHERE paid_at > (
        SELECT MAX(calculated_at) FROM accounting
    );

    INSERT INTO accounting (
        all_tansactions_sum, calculated_at
    ) VALUES (
        transactions_sum , NOW()
    );

END |

DELIMITER ;
```

Мы не ставили время начала работы события, поэтому событие начнет отрабатывать сразу же. Если учесть, что наша таблица

transactions пуста, то в таблицу accounting каждую минуту будут попадать следующие данные:

id	all_tansactions_sum	calculated_at
1	0.00	2023-03-05 11:00:00
2	0.00	2023-03-05 11:01:00

- Теперь вставим в таблицу transactions три строки с оплаченными транзакциями и посмотрим, что теперь будет занесено в таблицу accounting теперь:

```
INSERT INTO transactions (
    total_sum, paid_at
) VALUES
(100, NOW()),
(200, NOW()),
(300, NOW());
```

Спустя несколько минут в таблице accounting появится общая сумма произведенных на тот момент транзакций, а затем в таблицу опять начнут попадать нули, т.к. новых оплат не происходит:

id	all_tansactions_sum	calculated_at
1	0.00	2023-03-05 11:00:00
2	0.00	2023-03-05 11:01:00
3	600.00	2023-03-05 11:02:00
4	0.00	2023-03-05 11:03:00

- Вставим в таблицу transactions другие данные, чтобы убедиться, что они затем вставляются в таблицу accounting, но не смешиваются с предыдущими данными. Затем несколько минут подождем и произведем выборку из таблицы accounting:

```
INSERT INTO transactions ( total_sum, paid_at)
VALUES
(400, NOW()),
(500, NOW());
```

Можно увидеть, что произошло все то же самое, что и в предыдущем случае - в таблицу были занесены данные за предыдущий период, а затем, когда новых транзакций нет, в таблицу опять начали вставляться нули:

id	all_tansactions_sum	calculated_at
1	0.00	2023-03-05 11:00:00
2	0.00	2023-03-05 11:01:00
3	600.00	2023-03-05 11:02:00
4	0.00	2023-03-05 11:03:00
5	900.00	2023-03-05 11:04:00
6	0.00	2023-03-05 11:05:00

- После завершения работы с событием *save_accounting_info*, выключим его, чтобы впустую не тратить вычислительные мощности и место на диске. В будущем мы сможем его запустить, но в данный момент работа этого события нам не требуется:

```
ALTER EVENT save_accountring_info DISABLE;
```

- Теперь рассмотрим событие, которое происходит не периодически, а только один раз - в конкретный момент времени (не **EVERY**, а **AT**). Но для начала заполним данными таблицу sessions - она потребуется для работы с нашим новым событием.

```
INSERT INTO sessions (
    name, role, logged_in
) VALUES
('John Smith', 'CLIENT', '2023-03-05 11:10:00'),
('Tom Morgan', 'ADMIN', '2023-03-05 11:12:00'),
('Anna Grant', 'CLIENT', '2023-03-05 11:14:00'),
('Paul Jones', 'ADMIN', '2023-03-05 11:16:00');
```

id	name	role	logged_in
1	John Smith	CLIENT	2023-03-05 11:10:00

2	Tom Morgan	ADMIN	2023-03-05 11:12:00
3	Anna Grant	CLIENT	2023-03-05 11:14:00
4	Paul Jones	ADMIN	2023-03-05 11:16:00

- Теперь представим, что для неких тестовых целей нам необходимо событие, которое удалит все сессии, но не прямо сейчас, а через 1 минуту после своего создания, притом после того, как оно отработало, оно не должно удаляться автоматически. Создадим же это события, но сначала сделаем для себя одно маленькое открытие. Дело в том, что если мы используем в событии только одну команду, то мы может обойтись без блока **BEGIN** … **END** и смены разделителей. Воспользуемся этим кратким синтаксисом, чтобы создать наше событие удаления:

```
CREATE EVENT IF NOT EXISTS delete_sessions
ON SCHEDULE AT CURRENT_TIMESTAMP + INTERVAL 1 MINUTE
ON COMPLETION PRESERVE
DO
DELETE FROM sessions;
```

Ровно через одну минуту все записи из таблицы sessions будут удалены, а событие *delete_sessions* будет приостановлено, но не удалено, т.к. мы использовали конструкцию **ON COMPLETE PRESERVE** при его создании.

МОДИФИКАЦИЯ СОБЫТИЙ

- Почему же мы не позволили удалиться предыдущему событию автоматически? Мы это сделали потом, что теперь следует продемонстрировать модификацию событий. У нас есть возможность изменить для события практически все (даже логику и расписание выполнения) при помощи уже знакомой конструкции **ALTER EVENT** (мы уже использовали эту конструкцию, чтобы выключить периодическое событие *save_accounting_info*). Представим, что мы хотим изменить *delete_sessions* таким образом, чтобы оно срабатывало 1 раз в минуту и удаляло все сессии, кроме тех, которые принадлежат посетителям со статусом ADMIN. Но сначала опять заполним таблицу sessions, чтобы потом увидеть, что наше событие действительно отрабатывает:

```
INSERT INTO sessions (
    name, role, logged_in
) VALUES
('Alfred Savage', 'CLIENT', '2023-03-05 11:30:00'),
('Ava Baker', 'ADMIN', '2023-03-05 11:32:00'),
('Harry Bishop', 'CLIENT', '2023-03-05 11:34:00'),
('Lara Gilbert', 'ADMIN', '2023-03-05 11:36:00');
```

id	name	role	logged_in
5	Alfred Savage	CLIENT	2023-03-05 11:30:00
6	Ava Baker	ADMIN	2023-03-05 11:32:00
7	Harry Bishop	CLIENT	2023-03-05 11:34:00
8	Lara Gilbert	ADMIN	2023-03-05 11:36:00

- Далее модифицируем наше событие *delete_sessions*:

```
ALTER EVENT delete_sessions
ON SCHEDULE EVERY 1 MINUTE
DO
DELETE FROM sessions WHERE role <> 'ADMIN';
```

Несмотря на то, что мы модифицировали событие таким образом, чтобы оно срабатывало сразу, оно не будет запущено автоматически. Дело в том, что событие было автоматически выключено, после того, как оно было использовано в прошлый раз. Поэтому в этот раз мы должны включить его вручную.

```
ALTER EVENT delete_sessions ENABLE;
```

Вот теперь событие начнет отрабатывать каждую минуту, удаляя все сессии, которые не принадлежат администраторам. Через минуту таблица sessions будет выглядеть следующим образом:

id	name	role	logged_in
6	Ava Baker	ADMIN	2023-03-05 11:32:00
8	Lara Gilbert	ADMIN	2023-03-05 11:36:00

УДАЛЕНИЕ СОБЫТИЙ

- Про удаление событий сразу отметим важную вещь - если даже мы удалим таблицы, которыми пользуется событие, событие все равно продолжит существовать и отрабатывать, хотя и с ошибкой (которую мы просто не будем видеть). Поэтому события всегда стоит удалять явно.

- Для явного удаления используется команда **DROP EVENT** - она может быть дополнена конструкцией **IF EXISTS**, которая обезопасит нас от ошибки в том случае, если событие не будет существовать.

```
-- удаление существующего события без проверки
DROP EVENT delete_sessions;

-- удаление существующего события с проверкой
DROP EVENT IF EXISTS save_accountring_info;

-- удаление несуществующего события с проверкой
DROP EVENT IF EXISTS undefined_event;
```

ПОДГОТОВЛЕННЫЕ ЗАПРОСЫ

ВВЕДЕНИЕ

- Одним из самых главных аспектов при разработке любого современного приложения является безопасность. Это относиться и к базам данных на базе MySQL. Может показаться, что это несколько преувеличенно - во всех предыдущих случаях мы могли подключиться к MySQL только если знали пароль и логин, а операции внутри баз данных были вполне безопасными и зависели только от нас (мы вручную подставляли все значения). Это действительно так, но MySQL чаще всего используется не напрямую, а через какие-либо библиотеки, пакеты или модули в языках программирования. С помощью этих модулей происходит подключение к MySQL и совершаются необходимые действия. Вот тут как раз и можно найти тонкое место в нашей безопасности. Если приложение получило данные от какой-либо внешней формы, которую заполнил злоумышленник на сайте, а затем эти данные без должной обработки были вставлены в запрос и отправлены в СУБД, то запрос может изменить свой первоначальный смысл и может произойти нечто, что не было предусмотрено создателями приложения!

- Рассмотрим простейший пример. У нас есть таблица admins, с помощью которой происходит аутентификация администратора на сайте (**В БАЗАХ ДАННЫХ НЕЛЬЗЯ ХРАНИТЬ ПАРОЛЬ В ОТКРЫТОМ ВИДЕ,** но для удобства демонстрации сделаем это).

```
CREATE TABLE admins (
    id INT UNSIGNED NOT NULL AUTO_INCREMENT PRIMARY KEY,
    login VARCHAR(255) NOT NULL,
    password VARCHAR(255) NOT NULL
);

INSERT INTO admins (login, password)
VALUES
('john.smith@gmail.com', 'abacaba'),
('alex.kova@gmail.com', '1a2b3c');
```

id	login	login
1	john.smith@gmail.com	abacaba
2	alex.kova@gmail.com	1a2b3c

Когда администратор хочет попасть в административную панель сайта, то он заполняет форму, где надо указать логин и пароль, затем форма отправляется на обработчик для проверки. Проверка происходит обычно следующим образом:

```
SELECT id
FROM admins
WHERE login = 'alex.kova@gmail.com' AND password = '1a2b3c'
LIMIT 1;
```

Если по этому запросу вернулась строка, то это означает, что пользователь ввел правильные данные (которые существуют в базе данных) и его можно запускать в административную панель. Однако этот процесс опирается на то, что присланные данные были правильно обработаны, и все опасные последовательности были экранированы. В то же время очень часто разработчики забывают обработать данные, а потом к нам приходит следующий запрос:

```
SELECT id
FROM admins
WHERE login = '' OR 1 = 1; -- ' AND password = '123' LIMIT 1;
```

В результате мы будем запущены внутрь административной панели сайта как полноценный администратор. Как же это случилось? Получается, что пользователь прислал нам в качестве пароля *123* (не существует в базе данных), а в качестве логина - очень интересную конструкцию `' OR 1 = 1; --` , которая и поменяла смысл запроса (такие значения называют SQL инъекцией). Если изначально запрос задумывался как - “дайте мне администратора, у которого логин и пароль равняются присланным значениям”, то теперь это “дайте мне администратора, у которого имя равняется пустой строке или 1 равняется 1”. Один всегда равняется одному, поэтому запрос всегда будет отрабатывать успешно!

- Кончено, можно полностью переложить ответственность на разработчиков - пусть они всегда экранируют запросы и убирают все возможные опасности! Однако надо признать, что всего предусмотреть невозможно. Поэтому нужен механизм, который позволит в принципе убрать возможность ошибки. В MySQL такой механизм есть - он называется подготовленные запросы (**PREPARED STATEMENTS**).

- Подготовленный запрос - специальный шаблон в виде одного запроса (который обрамлен в кавычки), где вместо значений стоят знаки вопроса. Подготовленный запрос сначала создается, а потом вызывается (притом он может вызываться многократно) с необходимыми значениями. Важнейший момент - созданный подготовленный запрос существует лишь в то время, когда мы подключены к базе данных (т.е. аа время так называемой сессии). После того, как мы завершили свою работу и отключились от базы данных, все подготовленные запросы автоматически удаляются. Это значит, что подготовленные запросы надо создавать заново при каждом новом подключении.

- Кроме решения проблемы безопасности, подготовленные запросы имеют еще одну приятную особенность. Дело в том, что заранее подготовленный запрос лучше просчитывается интерпретатором MySQL, что может благоприятно сказаться на быстроте выполнения сложных и запутанных запросов.

- Создание подготовленных запросов происходит при помощи ключевой конструкции **PREPARE** … **FROM:**

```
PREPARE название_запроса FROM 'запрос';
```

Вызов подготовленного запроса осуществляется при помощи конструкции **EXECUTE** … **USING** … .

```
EXECUTE название_запроса USING значение, значение, … ;
```

СОЗДАНИЕ И ВЫПОЛНЕНИЕ ПОДГОТОВЛЕННЫХ ЗАПРОСОВ

- Попробуем создать наш первый подготовленный запрос и с помощью этого запроса решить проблему с аутентификацией администратора (обратим внимание, что в подготовленном запросе не надо ставить кавычки возле текстовых значений - достаточно просто поставить вопрос).

```
PREPARE admin_selection FROM
'SELECT id
FROM admins
WHERE login = ? AND password = ? LIMIT 1';
```

Теперь попробуем воспроизвести наш пример, который содержал SQL инъекцию в контексте нашего нового подготовленного запроса (воспроизведение будет с помощью переменных самого MySQL, которые должны передаваться в том порядке, в котором они перечислены в подготовленном запросе):

```
SET @login = "' OR 1 = 1; -- ";
SET @password = "123";

EXECUTE admin_selection USING @login, @password;
```

В качестве ответа мы получим пустую таблицу, что полностью соответствует ожиданиям. Если же мы подставим правильные значение, то получим идентификатор администратора:

```
SET @login = "john.smith@gmail.com";
SET @password = "abacaba";

EXECUTE admin_selection USING @login, @password;
```

id
1

- Подготовленные запросы можно использовать не только для получения данных, но и для вставки, обновления и удаления. Для примера попробуем реализовать запрос, который создает (т.е. вставляет) нового адмнинистратора:

```
PREPARE admin_creation FROM
'INSERT INTO admins (
    login, password
) VALUES (
    ?, ?
)';
```

А затем попытаемся создать при помощи этого запроса нашего нового администратора:

```
SET @new_login = "john.doe@gmail.com";
SET @new_password = "87654321";

EXECUTE admin_creation USING @new_login, @new_password;
```

После исполнения подготовленного запроса в таблице admins должна была появиться новая запись:

id	login	login
1	john.smith@gmail.com	abacaba
2	alex.kova@gmail.com	1a2b3c
3	john.doe@gmail.com	87654321

УДАЛЕНИЕ ПОДГОТОВЛЕННЫХ ЗАПРОСОВ

- Как уже говорилось, подготовленные запросы удаляются сами, после того, как будет закрыто соединение с базой данных. Однако если нам по какой-то причине необходимо это сделать явно и вручную, то в MySQL есть целый две конструкции, которые позволяют это сделать DEALLOCATE PREPARE и DROP PREPARE, после которых следует указывать название удаляемого подготовленного запроса:

```
-- 1-й способ удаления
DEALLOCATE PREPARE admin_selection;

-- 2-й способ удаления
DROP PREPARE admin_creation;
```