

# MySQL

Добавление, обновление и удаление данных

Добавление данных (INSERT)

Добавление данных является одним из самых простых методов взаимодействия с СУБД MySQL. В отличие от получения данных, вставка может быть описана гораздо меньшим количеством ключевых слов и конструкций (если кратко, то надо помнить только 2 оператора - **INSERT INTO** и **REPLACE INTO**, а также ключевое слово **VALUES**). Но и в данном случае есть много “подводных камней” и особенностей, знание которых с одной стороны позволит избежать потенциальных проблем, а с другой - создавать более эффективные и элегантные решения (приложения, сайты и т.д.) на базе MySQL.

## Подготовка к добавлению данных (создание таблиц)

Чтобы начать работать с добавлением данных создадим две таблицы - простые гости (**guests**) и клиенты (**clients**). Код для создания обеих таблиц приведен ниже:

```
CREATE TABLE guests (  
    id BIGINT UNSIGNED NOT NULL AUTO_INCREMENT PRIMARY KEY,  
    name VARCHAR(255) NOT NULL,  
    visit_date DATE,  
    total_paid DECIMAL(20,2) DEFAULT 0  
);  
  
CREATE TABLE clients (  
    id BIGINT UNSIGNED NOT NULL AUTO_INCREMENT PRIMARY KEY,  
    name VARCHAR(255) NOT NULL,  
    verified BOOLEAN NOT NULL,  
    created_at DATE  
);
```

# Простейший способ добавления данных

Основным оператором для вставки является **INSERT INTO**. Именно с него начинается запрос, затем следует название таблицы, после этого в скобках через запятую перечисляются столбцы (можно перечислять не все, а лишь необходимые), для которых будут вставляться значения. Потом идет ключевое слово **VALUES**, а уже после него в скобках через запятую перечисляются значений столбцов (должны идти в том же порядке, в котором в этом же запросе столбцы были перечислены после названия таблицы). Приведем пример - вставим гостя в таблицу **guests**:

```
INSERT INTO guests (  
    id, name, visit_date, total_paid  
) VALUES (  
    1, 'John Smith', '2023-01-01', 0  
);
```

Имя и дата визита указаны в кавычках. Кавычки обязательны для текстовых типов данных (**VARCHAR**, **CHAR**, **TEXT**), типов даты и времени (**DATE** и **DATETIME**), а также для типа **JSON**. Притом неважно, используем ли мы одинарные или двойные кавычки - MySQL понимает оба варианта.

<u>id</u>	name	visit_date	total_paid
1	John Smith	2023-01-01	0.00

# Экранирование данных

Что делать, если запрос на добавление данных сам по себе содержит кавычки? Если ничего не делать, то при вставке произойдет ошибка. Однако есть очень простой способ этого избежать - в текстовых значениях надо добавить обратный слэш \ перед каждой кавычкой (т.е. произвести экранирование) - тогда запрос пройдет без нарушений и данные будут вставлены нормально (при этом сами экранирующие символы будут отброшены и сохранены не будут):

```
INSERT INTO guests (  
    id, name, visit_date, total_paid  
) VALUES (  
    2, 'Martin O\'Brien', '2023-01-02', 0  
);
```

<u>id</u>	name	visit_date	total_paid
1	John Smith	2023-01-01	0.00
2	Martin O'Brien	2023-01-02	0.00

## Добавление данных без перечисления столбцов

Два предыдущий раза мы использовали при запросу именно тот порядок столбцов, который определен при создании базы данных. Важно знать что в этом случае (когда мы хотим вставить все столбцы именно в этом порядке), можно использовать упрощенную версию **INSERT INTO**, в которой вообще нет перечисления столбцов:

```
INSERT INTO guests VALUES (  
    3, 'Mary Sue', '2023-01-03', 20  
);
```

<u>id</u>	name	visit_date	total_paid
1	John Smith	2023-01-01	0.00
2	Martin O'Brien	2023-01-02	0.00
3	Mary Sue	2023-01-03	20.00

## Добавление данных с перечислением столбцов в произвольном порядке

Приведем пример, когда столбцы для добавления указаны в произвольном порядке. Хотя и в этом случае мы должны четко понимать, что значение для столбцов быть именно в том порядке в котором указаны столбцы в запросе (не может столбец **id** быть на втором месте, а значение для него на четвертом - в этом случае оно обязано быть перечислено вторым):

```
INSERT INTO guests (  
    visit_date, id, total_paid, name  
) VALUES (  
    '2023-01-04', 4, 35.99, 'Joe Maldini'  
);
```

<u>id</u>	name	visit_date	total_paid
1	John Smith	2023-01-01	0.00
2	Martin O'Brien	2023-01-02	0.00
3	Mary Sue	2023-01-03	20.00
4	Joe Maldini	2023-01-04	35.99



## Добавление данных без упоминания некоторых столбцов

Иногда нам не надо перечислять все столбцы и все значения для них. Например, для таблицы **guests** значение **id** может генерироваться само, т.к. при создании этого столбца мы добавили атрибут **AUTO\_INCREMENT**, столбец **visit\_date** вообще не является обязательным - если не предоставить для него значение, то в базу данных сохранится **NULL**. В свою очередь **total\_paid** имеет значение по умолчанию 0 - оно и попадет в таблицу, если мы не укажем значение явно. Приведем пример всего вышесказанного (т.е. вставим в таблицу только значение столбца **name**) и убедимся, что запрос выполнится успешно:

```
INSERT INTO guests (name) VALUES ('Rolf Meier');
```

<u>id</u>	name	visit_date	total_paid
1	John Smith	2023-01-01	0.00
2	Martin O'Brien	2023-01-02	0.00
3	Mary Sue	2023-01-03	20.00
4	Joe Maldini	2023-01-04	35.99
5	Rolf Meier	NULL	0.00

## Получение идентификатора последней вставленной строки

Можно заметить, что в предыдущем запросе, как и ожидалось, значение столбца `id` было сгенерировано автоматически (т.к. это столбец с автоинкрементом). В таких случаях (при автоматическом создании значения через автоинкремент, если прописывать значение вручную, то этот функционал недоступен) у нас появляется возможность узнать, какое именно последнее значение было сгенерировано. Для этого используется специальная функция **`LAST_INSERT_ID()`**. Если мы запросим у этой функции данные сразу последнего запроса (когда был вставлен Rolf Meier), то получим в ответ пятерку (5):

```
SELECT LAST_INSERT_ID();
```

<code>LAST_INSERT_ID()</code>
5

Если запросов с автоматической вставкой автоинкремента в контексте данного подключения к базе данных не было, MySQL будет всегда возвращать нуль (0).

## Добавление нескольких строк одновременно (теория)

До этого мы рассматривали добавление данных в контексте вставки только одной строки. Однако MySQL позволяет вставлять несколько строк одновременно - для этого надо просто перечислить скобки со значениями столько раз, сколько нам необходимо строк, разделив эти скобки запятыми. Количество строк теоретически неограниченно, но если MySQL серверу не хватит памяти, запрос на добавление данных закончится неудачей.

На количество разрешенной памяти влияет файл **my.ini** и его настройка **max\_allowed\_packet**. Эта настройка устанавливает то количество разрешенное памяти, которое может занимает каждый конкретный запрос. Она может установлена как в разделе клиента (**[mysql]** - т.е. максимальный размер отправляемого запроса), так и в разделе сервера (**[mysqld]** - максимальный размер принимаемого запроса). По умолчанию клиенту выделено 16MB, а серверу - 64MB.

## Добавление нескольких строк одновременно (пример)

Приведем пример добавления нескольких строк за один раз - вставим данные в таблицу **clients**:

```
INSERT INTO clients (  
  name, verified, created_at  
) VALUES  
(  
  'Jack Hawkins', 0, '2023-02-01',  
  'Sheila Elliott', 1, '2023-02-02',  
  'Rosa Fernandez', 1, '2023-02-03');
```

<u>id</u>	name	verified	created_at
1	Jack Hawkins	0	2023-02-01
2	Sheila Elliott	1	2023-02-02
3	Rosa Fernandez	1	2023-02-03

Обратим внимание, что значения столбца **id** были сгенерированы автоматически с учетом того, что в таблицу были вставлены несколько строк - каждая строка получила свой уникальный **id**, и не было попытки вставить всем одинаковое значение.

## Заполнение данными из другой таблицы (теория)

Существует еще один способ вставки нескольких строк одновременно. Сам по себе он не гарантирует, что данные вообще будут вставлены, но сейчас мы поймем, что вероятность этого весьма высока. Дело в том, что в этом случае в качестве источника данных выступает не данные, предоставляемые пользователей, а данные, получаемые из другой таблицы. Мы делаем запрос в некую таблицу при помощи **SELECT**, выбирая необходимый набор столбцов, и сразу же после этого вставляем их в таблицу при помощи **INSERT INTO**.

## Заполнение данными из другой таблицы (пример)

Преобразуем всех гостей (таблица **guests**) в клиентов (таблица **clients**) в рамках одного запроса:

```
INSERT INTO clients (  
    name, verified, created_at  
) SELECT name, IF(total_paid > 0, 1, 0), '2023-02-10' FROM guests;
```

Названия столбцов в **SELECT** не имеют значения, важен их порядок - он должен по смыслу совпадать с тем порядком столбцов, который определен в **INSERT**. Кроме того, не все данные могут выбираться из таблицы напрямую. Например, значение для `verified` мы устанавливаем на лету для каждой строки (проверенными станут только те гости, которые потратили хоть какое-то количество денег), а значение **created\_at** мы просто прописали вручную - '2023-02-10':

<u>id</u>	name	verified	created_at
1	Jack Hawkins	0	2023-02-01
2	Sheila Elliott	1	2023-02-02
3	Rosa Fernandez	1	2023-02-03
4	John Smith	0	2023-02-10
5	Martin O'Brien	0	2023-02-10
6	Mary Sue	1	2023-02-10
7	Joe Maldini	1	2023-02-10
8	Rolf Meier	0	2023-02-10

## Недостаток INSERT ... SELECT (инициализация)

**INSERT ... SELECT** - довольно мощный инструмент, но данная конструкция имеет один недостаток. Чтобы его увидеть, внесем в таблицу еще одну строку:

```
INSERT INTO clients (name, verified, created_at) VALUES ('Ava Harding', 0, '2023-04-01');
```

<u>id</u>	name	verified	created_at
1	Jack Hawkins	0	2023-02-01
2	Sheila Elliott	1	2023-02-02
3	Rosa Fernandez	1	2023-02-03
4	John Smith	0	2023-02-10
5	Martin O'Brien	0	2023-02-10
6	Mary Sue	1	2023-02-10
7	Joe Maldini	1	2023-02-10
8	Rolf Meier	0	2023-02-10
11	Ava Harding	0	2023-04-01

## Недостаток INSERT ... SELECT (объяснение)

Очевидно, что произошла странность - значение автоинкремента возросло скачкообразно (оно должно было равняться 9, а стало 11). Это не вина последней вставки, главный виновник - **INSERT ... SELECT**. Дело в том, при использовании такой конструкции не гарантируется, что значение автоинкремента будет соразмерно вставленным данным, но гарантируется, что это значение будет уникальным. Это можно предотвратить, если поставить в **my.ini** настройку **innodb\_autoinc\_lock\_mode=0**, которая блокирует таблицу во время вставки и вставляет строки строго одну за другой. Но лучше этого не делать, т.к. блокировка может снизить скорость работы с этой таблицей.



## Добавление данных с обновлением - INSERT INTO ... ON DUPLICATE KEY UPDATE (теория)

Иногда мы хотим вставить данные, которые должны частично обновить уже существующие строки в таблице, а частично создать новые строки. Допустим у нас в административной панели есть список пользователей и есть возможность добавлять новых. Приходят данные о пользователях все сразу - следовательно часть мы должны обновить, а часть - создать как новых. MySQL позволяет это сделать при помощи конструкции **INSERT INTO ... ON DUPLICATE KEY UPDATE ...** .

Данная конструкция работает следующим образом - если в данных нет столбцов (или столбцы есть, но они имеют значение **NULL**), которые являются ключом (первичным или уникальным), то произойдет вставка данных. Если же значения таких столбцов есть - для этой строки будет выполняться условие, которое указано после ключевого слова **UPDATE**.

## Добавление данных с обновлением - INSERT INTO ... ON DUPLICATE KEY UPDATE (пример)

Приведем пример запроса с добавлением и обновлением в контексте таблицы **clients**. Попытаемся обновить имя и верифицированность для клиента с `id = 4`, а также добавить нового клиента:

```
INSERT INTO clients (id, name, verified, created_at) VALUES  
(4, 'Homer Simpson', 1, '2023-05-01'),  
(NULL, 'Josh Dolton', 0, '2023-05-01') AS new  
ON DUPLICATE KEY UPDATE name = new.name, verified = new.verified;
```

<u>id</u>	name	verified	created_at
1	Jack Hawkins	0	2023-02-01
2	Sheila Elliott	1	2023-02-02
3	Rosa Fernandez	1	2023-02-03
4	Homer Simpson	1	2023-02-10
5	Martin O'Brien	0	2023-02-10
6	Mary Sue	1	2023-02-10
7	Joe Maldini	1	2023-02-10
8	Rolf Meier	0	2023-02-10
11	Ava Harding	0	2023-04-01
12	Josh Dolton	0	2023-05-01

## Добавление данных с заменой - REPLACE INTO (теория)

У конструкции **INSERT INTO ... ON DUPLICATE KEY UPDATE ...** есть альтернатива - это директива **REPLACE INTO**. Какое же между ними различие? Если **INSERT ... UPDATE** вставляет строку или обновляет уже существующую, то **REPLACE** с одной стороны тоже вставляет строку, но с другой - при попытке вставить данные с уже существующим ключом (первичным или уникальным) сначала удалит существующую строку, а потом вставит те данные, которые пришли в запросе. Это значит, что при использовании **REPLACE** не будет доступно обновление отдельных столбцов - мы заменяем всю строку.

## Добавление данных с заменой - REPLACE INTO (пример)

Приведем пример с конструкцией **REPLACE INTO**:

```
REPLACE INTO clients (id, name, verified, created_at) VALUES  
(4, 'John Smith', 1, '2023-06-01'),  
(NULL, 'Emily Carter', 0, '2023-06-01');
```

<u>id</u>	name	verified	created_at
1	Jack Hawkins	0	2023-02-01
2	Sheila Elliott	1	2023-02-02
3	Rosa Fernandez	1	2023-02-03
4	John Smith	1	2023-06-01
5	Martin O'Brien	0	2023-02-10
6	Mary Sue	1	2023-02-10
7	Joe Maldini	1	2023-02-10
8	Rolf Meier	0	2023-02-10
11	Ava Harding	0	2023-04-01
12	Josh Dolton	0	2023-05-01
14	Emily Carter	0	2023-06-01

## Скачкообразное увеличение значения автоинкремента

Посмотрев на результат, мы увидим, что автоинкрементное значение id совершило скачок на 1 позицию (ожидалось, что оно будет равно 13, а оно равняется 14). Это связано с “проделками” **ON DUPLICATE KEY UPDATE** из предыдущего шага при работе со строкой с id = 4. Эта конструкция попыталась сначала вставить данные, что увеличило значение автоинкремента на единицу, но ей это не удалось, и она просто обновила строку. **REPLACE INTO** ни в чем не виновата.

## Обработка ошибок при множественном добавлении (проблема)

Что случится, если мы вставляем много строк, но некоторые из них имеют неправильные значения - вставляются ли правильные строки или запрос будет отменен полностью? Если не применять специальных конструкций, то произойдет второй вариант. Например, вот такой запрос не пройдет и данные первой (правильной) строки не будут вставлены в таблицу:

```
INSERT INTO clients(id, name, verified, created_at) VALUES  
(NULL, 'Joseph Parker', 1, '2023-05-02'),  
(1, 'Some Name', 1, '2023-05-02'),  
(NULL, 'Elen Walters', 1, 'wrong-date');
```

Заметим, что предыдущий запрос все же окажет некоторое влияние на таблицу. Была попытка (хотя и неудачная) вставить 3 строки - значит значение автоинкремента (которое после манипуляций с **REPLACE INTO** равнялось 16 - ведь тогда было вставлено 2 строки после того, как автоинкремент равнялся 14) сместилось на 3, и если мы вставим новую строку, то она получит id = 19 (если мы не укажем id явно).

## Обработка ошибок при множественном добавлении (решение проблемы)

А что, если мы хотим, чтобы в таких случаях были вставлены правильные строки, а неправильные проигнорированы? Решение этой проблемы есть - MySQL предоставляет конструкцию **INSERT IGNORE ...**, которая вставит все возможные строки и даже попытается исправить ошибки в неправильных строках и все равно их вставить:

```
INSERT IGNORE INTO clients(id, name, verified, created_at) VALUES  
(NULL, 'Joseph Parker', 1, '2023-05-02'),  
(1, 'Some Name', 1, '2023-05-02'),  
(NULL, 'Elen Walters', 1, 'wrong-date');
```

Первая строка была вставлена успешно, вторая отброшена, т.к. она нарушает ограничение первичного ключа, третья строка также была вставлена, но неправильное значение даты 'wrong-date' было автоматически трансформировано в '0000-00-00'

## Финальный вид таблицы clients после всех добавлений

<u>id</u>	name	verified	created_at
1	Jack Hawkins	0	2023-02-01
2	Sheila Elliott	1	2023-02-02
3	Rosa Fernandez	1	2023-02-03
4	John Smith	1	2023-06-01
5	Martin O'Brien	0	2023-02-10
6	Mary Sue	1	2023-02-10
7	Joe Maldini	1	2023-02-10
8	Rolf Meier	0	2023-02-10
11	Ava Harding	0	2023-04-01
12	Josh Dolton	0	2023-05-01
14	Emily Carter	0	2023-06-01
19	Joseph Parker	1	2023-05-02
20	Elen Waters	1	0000-00-00



Обновление данных (UPDATE)

# Основы обновления

Обновление осуществляется с помощью ключевого слова **UPDATE**. После этого слова необходимо указать название таблицы, затем ключевое слово **SET**, а уже после этого через запятую перечислить названия обновляемых столбцов и их значения в формате столбец1 = значение, столбец2 = значение и т.д. Запомним, что обновлять значения всех столбцов каждой строки не надо, можно обновить только некоторые столбцы. Посмотрим обновление на примере и обновим таблицу **clients** из предыдущей главы - поставим клиентам верифицированный статус и дату создания, равную '2023-07-01':

id	name	verified	created_at
1	Jack Hawkins	1	2023-07-01
2	Sheila Elliott	1	2023-07-01
3	Rosa Fernandez	1	2023-07-01
4	John Smith	1	2023-07-01
5	Martin O'Brien	1	2023-07-01
6	Mary Sue	1	2023-07-01
7	Joe Maldini	1	2023-07-01
8	Rolf Meier	1	2023-07-01
11	Ava Harding	1	2023-07-01
12	Josh Dolton	1	2023-07-01
14	Emily Carter	1	2023-07-01
19	Joseph Parker	1	2023-07-01
20	Elen Waters	1	2023-07-01

```
UPDATE clients SET verified = 1, created_at = '2023-07-01';
```

## Обновление данных только для конкретных строк (а не для всей таблицы)

С помощью прошлого запроса мы обновили всю таблицу, но эта ситуация происходит довольно редко. Гораздо чаще нам надо обновить какие либо конкретные строки. В данном случае применяется уже хорошо знакомый нам оператор **WHERE**, который ограничивает те строки которые будут обновлены. Попробуем обновить только те строки (поставим им другую дату создания - **created\_at**), у которых столбце **id** имеет значение 4, 5 или 6 или столбец **name** которых равен Josh Dolton:

id	name	verified	created_at
1	Jack Hawkins	1	2023-07-01
2	Sheila Elliott	1	2023-07-01
3	Rosa Fernandez	1	2023-07-01
4	John Smith	1	2023-01-01
5	Martin O'Brien	1	2023-01-01
6	Mary Sue	1	2023-01-01
7	Joe Maldini	1	2023-07-01
8	Rolf Meier	1	2023-07-01
11	Ava Harding	1	2023-07-01
12	Josh Dolton	1	2023-01-01
14	Emily Carter	1	2023-07-01
19	Joseph Parker	1	2023-07-01
20	Elen Waters	1	2023-07-01

```
UPDATE clients SET created_at = '2023-01-01' WHERE id IN (4,5,6) OR name = 'Josh Dolton';
```

## Дополнение значений при обновлении

Иногда при обновлении нам надо не полностью переписать значение, а дополнить его. Например, в нашей таблице **guests** есть столбец **total\_paid**, который у гостя с именем Mary Sue равен 20. Стало известно, что этот гость заплатил еще 10 евро. Чтобы обновить значение этого столбца, нам не надо получать его из таблицы, вручную прибавлять к нему 10, а уже затем вставлять обновленное значение в таблицу - это все можно сделать при помощи следующего единого запроса:

```
UPDATE guests SET total_paid = total_paid + 10 WHERE name = 'Mary Sue';
```

<u>id</u>	name	visit_date	total_paid
1	John Smith	2023-01-01	0.00
2	Martin O'Brien	2023-01-02	0.00
3	Mary Sue	2023-01-03	30.00
4	Joe Maldini	2023-01-04	35.99
5	Rolf Meier	NULL	0.00

## Применение CASE ... WHEN ... THEN ... ELSE (базовое применение)

Если в предыдущем случае мы обновляли значение только для одной конкретной строки, то в этот раз попробуем обновить несколько конкретных строк и указать для каждой из них свое значение. Для этого применим уже знакомую нам конструкцию **CASE ... WHEN ... THEN**:

```
UPDATE guests SET total_paid = CASE  
  WHEN name = 'John Smith' THEN 15.99  
  WHEN name = 'Martin O'Brien' THEN 22.35  
  WHEN name = 'Rolf Meier' THEN 85.49 END;
```

Строки с соответствующими именами действительно обновились, но мы с удивлением обнаружим, что обновилась вообще вся таблица, т.к. у всех строк, которые не попали в конструкцию фильтрации, столбец **total\_paid** получил значение **NULL**. В связи с этим запомним, что **CASE ... WHEN ... THEN** без дополнительных условий обновляет всю таблицу (там где условия не подходят, генерируется значение **NULL**) и в будущем будем использовать эту конструкцию только в совокупности со словом **ELSE**.

<u>id</u>	name	visit_date	total_paid
1	John Smith	2023-01-01	15.99
2	Martin O'Brien	2023-01-02	22.35
3	Mary Sue	2023-01-03	NULL
4	Joe Maldini	2023-01-04	NULL
5	Rolf Meier	NULL	85.49

## Применение CASE ... WHEN ... THEN ... ELSE (правильное применение)

Восстановим неправильно обновленные значения таблицы при помощи той же самой конструкции **CASE ... WHEN ... THEN**, но в этом случае применим уже упомянутое выше ключевое слово **ELSE**:

```
UPDATE guests SET total_paid = CASE  
WHEN name = 'Mary Sue' THEN 30.00  
WHEN name = 'Joe Maldini' THEN 35.99  
ELSE total_paid END;
```

<u>id</u>	name	visit_date	total_paid
1	John Smith	2023-01-01	15.99
2	Martin O'Brien	2023-01-02	22.35
3	Mary Sue	2023-01-03	30.00
4	Joe Maldini	2023-01-04	35.99
5	Rolf Meier	NULL	85.49

## Применение CASE ... WHEN ... THEN ... ELSE (оптимизированное применение)

Теперь таблица была обновлена абсолютно правильно, но все же заметим, что запрос проверил абсолютно все строки таблицы, что может негативно сказаться на скорости выполнения запроса, если таблица будет очень большая. Для оптимизации в конце запроса можно указать значение **WHERE** и отфильтровать только те строки, значение которых должно обновиться. Результат запроса не изменится, но выполняться запрос в некоторых случаях будет быстрее:

```
UPDATE guests SET total_paid = CASE  
WHEN name = 'Mary Sue' THEN 30.00  
WHEN name = 'Joe Maldini' THEN 35.99  
ELSE total_paid END  
WHERE name IN('Mary Sue', 'Joe Maldini');
```

<u>id</u>	name	visit_date	total_paid
1	John Smith	2023-01-01	15.99
2	Martin O'Brien	2023-01-02	22.35
3	Mary Sue	2023-01-03	30.00
4	Joe Maldini	2023-01-04	35.99
5	Rolf Meier	NULL	85.49

# Применение CASE ... WHEN ... THEN ... ELSE (обновление нескольких столбцов)

**CASE ... WHEN ... THEN** также позволяет обновлять не только один столбец, но и несколько столбцов одновременно - для этого надо переписать запрос (не будем применять ключевое слово **ELSE**, т.к. **WHERE** оставит для обновления только те строки, которые нам нужны - других там не будет):

```
UPDATE guests SET
visit_date = CASE
    WHEN name = 'Martin O'Brien' THEN '2023-01-03'
    WHEN name = 'Rolf Meier' THEN '2023-01-03'
END,
total_paid = CASE
    WHEN name = 'Martin O'Brien' THEN 48.56
    WHEN name = 'Rolf Meier' THEN 78.33
END
WHERE name IN('Martin O'Brien', 'Rolf Meier');
```

<u>id</u>	name	visit_date	total_paid
1	John Smith	2023-01-01	15.99
2	Martin O'Brien	2023-01-03	48.56
3	Mary Sue	2023-01-03	30.00
4	Joe Maldini	2023-01-04	35.99
5	Rolf Meier	2023-01-03	78.33



# Обновление при помощи данных из других таблиц (SELECT)

Как и в случае с добавления данных, для обновления можно использовать не только прописанные вручную значения, но и данные из других таблиц. Например, обновим значение столбца `created_at` для таблицы `clients` на основе таблицы `guests` и ее столбца **`visit_date`** (для связки двух таблиц, будем использовать значения столбца **`name`** в **`clients`** и **`guests`**):

```
UPDATE clients c
SET c.created_at = IFNULL((SELECT g.visit_date FROM
guests g WHERE g.name = c.name), c.created_at);
```

Обратим внимание, что подзапрос на получение значения обернут в функцию **`IFNULL`**, которая вернет нам второй аргумент, если первый аргумент будет равен значению **`NULL`**. Не для всех людей в таблице **`clients`** есть соответствующие им строки с такими же именами в таблице **`guests`** - следовательно для них в подзапросе вернется **`NULL`**. Мы не хотим вставлять **`NULL`**, поэтому оставляем значение **`created_at`** из таблицы **`clients`**.

<u>id</u>	name	verified	created_at
1	Jack Hawkins	1	2023-07-01
2	Sheila Elliott	1	2023-07-01
3	Rosa Fernandez	1	2023-07-01
4	John Smith	1	2023-01-01
5	Martin O'Brien	1	2023-01-03
6	Mary Sue	1	2023-01-03
7	Joe Maldini	1	2023-01-04
8	Rolf Meier	1	2023-01-03
11	Ava Harding	1	2023-07-01
12	Josh Dolton	1	2023-01-01
14	Emily Carter	1	2023-07-01
19	Joseph Parker	1	2023-07-01
20	Elen Waters	1	2023-07-01

# Обновление при помощи данных из других таблиц (JOIN)

Интересно заметить, что предыдущий запрос можно переписать с помощью **JOIN**, который мы применяли в конструкции **SELECT** - такой способ является более предпочтительным, т.к. он позволяет обновить несколько столбцов и не использовать функцию **IFNULL**. Чтобы подтвердить вышесказанное, перепишем прошлый запрос с конструкцией **JOIN**:

```
UPDATE clients c JOIN guests g
ON c.name = g.name
SET c.created_at = g.visit_date;
```

<u>id</u>	name	verified	created_at
1	Jack Hawkins	1	2023-07-01
2	Sheila Elliott	1	2023-07-01
3	Rosa Fernandez	1	2023-07-01
4	John Smith	1	2023-01-01
5	Martin O'Brien	1	2023-01-03
6	Mary Sue	1	2023-01-03
7	Joe Maldini	1	2023-01-04
8	Rolf Meier	1	2023-01-03
11	Ava Harding	1	2023-07-01
12	Josh Dolton	1	2023-01-01
14	Emily Carter	1	2023-07-01
19	Joseph Parker	1	2023-07-01
20	Elen Waters	1	2023-07-01

## Обработка ошибок при множественном обновлении (проблема)

Далее зададимся примерно тем же самым вопросом, какой мы уже задавали в конце главы о добавлении данных. Мы можем обновить множество строк в рамках одного запроса - что же будет, если для каких-то строк (например в контексте конструкции **CASE ... WHEN ... THEN**) будут предоставлены правильные данные, а для других - данные с ошибками? Не будем ничего говорить заранее, просто попробуем осуществить такой ошибочный запрос на практике:

```
UPDATE clients SET created_at = CASE  
  WHEN id = 1 THEN '2023-08-01'  
  WHEN id = 2 THEN 'wrong-date'  
  ELSE created_at END;
```

Запрос завершится с ошибкой, и данные первой (правильной) строки обновлены не будут.

# Обработка ошибок при множественном обновлении (решение)

Наш предыдущий запрос окончился неудачно, но у нас есть возможность его выполнить при помощи конструкции **UPDATE IGNORE** - как и в случае добавления данных, все возможные строки будут обновлены, строки с ошибками - по возможности исправлены и тоже обновлены, а те строки, которые невозможно исправить - проигнорированы:

```
UPDATE IGNORE clients SET created_at = CASE  
  WHEN id = 1 THEN '2023-08-01'  
  WHEN id = 2 THEN 'wrong-date'  
  ELSE created_at END;
```

Ошибки не будет, первая строка обновится согласно предоставленному значению, а значение для второй строки будет преобразовано в '0000-00-00':

id	name	verified	created_at
1	Jack Hawkins	1	2023-08-01
2	Sheila Elliott	1	0000-00-00
3	Rosa Fernandez	1	2023-07-01
4	John Smith	1	2023-07-01
5	Martin O'Brien	1	2023-01-03
6	Mary Sue	1	2023-01-03
7	Joe Maldini	1	2023-01-04
8	Rolf Meier	1	2023-01-03
11	Ava Harding	1	2023-07-01
12	Josh Dolton	1	2023-01-01
14	Emily Carter	1	2023-07-01
19	Joseph Parker	1	2023-07-01

## Подготовка таблицы **clients** к дальнейшему использованию

Во-первых, не будем смиряться с ошибочным значением из предыдущего запроса и обновим проблемную строку правильно (например, значением 2023-08-01):

```
UPDATE clients SET created_at = '2023-08-01' WHERE id = 2;
```

Во-вторых, добавим в таблицу **clients** столбец **position**, который потребуется нам в дальнейшем:

```
ALTER TABLE clients ADD COLUMN position INT UNSIGNED;
```

## Обновление с подзапросом в обновляемую таблицу (проблема)

Представим, что в таблице **clients** нам надо присвоить новому столбцу **position** значение, которое соответствует его положению в контексте столбца **created\_at** (те клиенты, которые были созданы раньше - получают первые номера, те, которые позже - последние).

```
UPDATE clients a
SET a.position = (
    SELECT ROW_NUMBER() OVER(ORDER BY b.created_at)
    FROM clients b WHERE a.id = b.id
    ORDER BY CASE WHEN a.id = b.id THEN 1 ELSE 2 END LIMIT 1
);
```

Мы получим ошибку, смысл которой состоит в том, что нельзя напрямую обращаться к той таблице, которую мы обновляем.

# Обновление с подзапросом в обновляемую таблицу (решение проблемы) (1)

Мы должны знать, что это ограничение на запрос в обновляемую таблицу не техническое, а скорее логическое. Если нельзя обратиться к таблице, то можно обратиться к выборке из таблицы (т.е. получается “двойной” **SELECT**):

```
UPDATE clients a
SET a.position = ( SELECT c.number FROM (
    SELECT ROW_NUMBER()
      OVER(ORDER BY b.created_at) AS number
    FROM clients b
    ORDER BY CASE
      WHEN a.id = b.id THEN 1
      ELSE 2 END LIMIT 1
  ) c );
```

id	name	verified	created_at	position
1	Jack Hawkins	1	2023-08-01	12
2	Sheila Elliott	1	2023-08-01	13
3	Rosa Fernandez	1	2023-07-01	7
4	John Smith	1	2023-01-01	1
5	Martin O'Brien	1	2023-01-03	3
6	Mary Sue	1	2023-01-03	4
7	Joe Maldini	1	2023-01-04	6
8	Rolf Meier	1	2023-01-03	5
11	Ava Harding	1	2023-07-01	8
12	Josh Dolton	1	2023-01-01	2
14	Emily Carter	1	2023-07-01	9
19	Joseph Parker	1	2023-07-01	10
20	Elen Waters	1	2023-07-01	11

## Обновление с подзапросом в обновляемую таблицу (решение проблемы) (2)

На самом деле, мы немного слукавили в предыдущем пункте, т.к. главной задачей было показать, что нельзя напрямую обращаться к той таблице, которую мы сейчас меняем, но можно обращаться к выборке из нее. Есть гораздо более простой способ осуществить поставленную задачу - при помощи **JOIN**, который сам по себе создаст нужную нам выборку и предоставит такой же результат как и в предыдущем запросе:

```
UPDATE clients c
JOIN (
  SELECT
    id,
    ROW_NUMBER() OVER(ORDER BY created_at ASC) AS number
  FROM clients
) c2 ON c.id = c2.id
SET c.position = c2.number;
```



## Сортировка и ограничение по количеству при обновлении (проблема)

Как бы это удивительно не звучало, но обновление поддерживает сортировку (**ORDER BY**) и ограничение по количеству при обновлении (**LIMIT**). Можно сразу спросить - а зачем это вообще надо? Признаем, что это может понадобиться довольно редко и в основном необходимо, когда будут исправляться некие ошибки. Представим, что у нас в таблицу **clients** занесли нового пользователя, но дали ему такое же имя, какое существует в таблице (например, Rolf Meier):

```
INSERT INTO clients (name, verified, created_at, position)
VALUES ('Rolf Meier', 0, '2024-01-01', 14);
```

Позже мы узнали, что это было ошибочное действие и мы должны были дать имя Jonathan Evans. Нам надо исправить ошибку, но как это сделать, если мы уже забыли, какой был **id** у этой ошибочной строки, и все что мы помним, что ошибочная строка имела позднее время создания (**created\_at**). Если мы обновимся при помощи фильтрации по имени, то обновятся сразу две строки, т.к. имя у них одинаковое!

# Сортировка и ограничение по количеству при обновлении (решение проблемы)

Решение задачи из предыдущего пункта состоит в том, чтобы перед обновлением надо отфильтровать строки по имени (**WHERE**), затем отсортировать оставшиеся строки (**ORDER BY**) по времени создания от позднего к раннему (чтобы более поздняя строка оказалась первой), затем взять только первую строку (**LIMIT 1**) и обновить лишь ее. Продемонстрируем это на практике:

```
UPDATE clients c
SET name = 'Jonathan Evans'
WHERE name = 'Rolf Meier'
ORDER BY created_at DESC
LIMIT 1;
```

<u>id</u>	name	verified	created_at	position
1	Jack Hawkins	1	2023-08-01	12
2	Sheila Elliott	1	2023-08-01	13
3	Rosa Fernandez	1	2023-07-01	7
4	John Smith	1	2023-01-01	1
5	Martin O'Brien	1	2023-01-03	3
6	Mary Sue	1	2023-01-03	4
7	Joe Maldini	1	2023-01-04	6
8	Rolf Meier	1	2023-01-03	5
11	Ava Harding	1	2023-07-01	8
12	Josh Dolton	1	2023-01-01	2
14	Emily Carter	1	2023-07-01	9
19	Joseph Parker	1	2023-07-01	10
20	Elen Waters	1	2023-07-01	11
22	Jonathan Evans	0	2024-01-01	14

Удаление данных (DELETE)

## Основы удаления (1)

За удаление данных отвечает оператор **DELETE**, который всегда ставится в самом начале запроса. После него идет ключевое слово **FROM**, а уже после него название той таблицы, данные из которой должны быть удалены. Попробуем произвести удаление из таблицы **guests**:

```
DELETE FROM guests;
```

Если мы произведем удаление в таком виде, то из таблицы **guests** будут удалены абсолютно все строки - без возможности их восстановления. Но эффект присутствия строк в таблице все же может сохраниться. Если в таблице существовал столбец с автоинкрементом, и были вставлены столбцы, то максимальное значение автоинкремента сохранится. Например, у нас в таблице **guests** было пять записей и последняя запись имела **id**, равный пяти. Если же после удаления всех записей мы вставим новую строку, но она получит 6 в качестве значения столба этого столбца **id**:

```
INSERT INTO guests (  
  name, visit_date, total_paid  
) VALUES (  
  'John Smith', '2024-01-01', 1  
);
```

<u>id</u>	name	visit_date	total_paid
6	John Smith	2024-01-01	1.00

## Основы удаления (2)

Существует еще один способ удаления - использование конструкции **TRUNCATE**. Ее применение отличается от того, что мы видели в случае с **DELETE** - сначала пишется само слово **TRUNCATE**, потом ключевое слово **TABLE**, а уже затем название той таблицы, которую мы хотим очистить.

```
TRUNCATE guests;
```

**TRUNCATE** производит еще более жестокое удаление, чем **DELETE** - кроме уничтожения всех строк, она также стирает значение автоинкремента. Это значит, что первая вставленная после такой очистки строка получит **id**, равный 1:

```
INSERT INTO guests (name, visit_date, total_paid) VALUES ('Mary Sue',  
'2024-01-02', 22.58);
```

<u>id</u>	name	visit_date	total_paid
1	Mary Sue	2024-01-02	22.58

## Проблемы при удалении - внешние ключи (основы)

Иногда мы не сможем удалить данные из таблицы с помощью **DELETE** или **TRUNCATE**. Мы уже знаем, что в таблицах могут применяться внешние ключи (**FOREIGN KEY**). Вспомним, что если зависимая таблица имеет внешний ключ, в котором установлено ограничение **ON DELETE RESTRICT**, и в нее вставлено значение, которое уже вписано в таблицу-источник, то строку с этим значением из таблицы-источника удалять нельзя. Реализуем это наглядно - создадим две таблицы - авторы (**authors**) и книги (**books**), где столбец книг **author\_id** ссылается на столбец **id** авторов:

```
CREATE TABLE authors (  
    id INT UNSIGNED NOT NULL AUTO_INCREMENT PRIMARY KEY,  
    name VARCHAR(255)  
);  
  
CREATE TABLE books (  
    id INT UNSIGNED AUTO_INCREMENT NOT NULL PRIMARY KEY,  
    author_id INT UNSIGNED NOT NULL,  
    title VARCHAR(255) NOT NULL,  
    FOREIGN KEY(author_id) REFERENCES authors (id) ON DELETE RESTRICT ON UPDATE RESTRICT  
);
```

## Проблемы при удалении - внешние ключи (проблема)

Занесем в каждую таблицу (**authors** и **books**) по одной записи, а затем попытаемся удалить все из таблицы **authors** (сначала при помощи **DELETE**, а потом при помощи **TRUNCATE**):

```
INSERT INTO authors VALUES (1, 'Mark Twain');
```

```
INSERT INTO books VALUES (1, 1, 'Tom Sawyer');
```

```
DELETE FROM authors;
```

```
TRUNCATE TABLE authors;
```

Как для первого, так и для второго запроса, MySQL вернет нам ошибку - в первом случае (**DELETE**) будет сказано, что нарушаем ограничение внешнего ключа - т.е. есть строки, которые нельзя удалить, а во втором случае будет отмечено, что при наличии внешнего ключа команду **TRUNCATE** нельзя исполнять в любых контекстах.

## Проблемы при удалении - внешние ключи (решение проблемы)

Несмотря на грозное предупреждение, мы все равно можем попытаться удалить данные из таблицы authors - для этого стоит использовать уже знакомое нам ключевое слово **IGNORE**. В этом случае ошибки не будет, но и данные из таблицы authors никуда не денутся.

```
DELETE IGNORE FROM authors;
```

Чтобы получить право удалить данные из таблицы с авторами, мы сначала должны удалить все ссылающиеся на них строки из таблицы книг - только тогда запрос **DELETE** будет выполнен успешно (**TRUNCATE** будет ругаться в любом случае):

```
DELETE FROM books;
```

```
DELETE FROM authors;
```



## Проблемы при удалении - внешние ключи (альтернативное решение проблемы)

Напомним, что такое ограничивающее внешнего ключа будет происходить только в том случае, если на него наложено ограничение **ON DELETE RESTRICT**. Если ограничение будет носить характер **ON DELETE CASCADE**, то запрос на удаление в любом случае пройдет успешно, но из таблицы `books` исчезнут все строки. А в том случае, если мы применим ограничение **ON DELETE SET NULL**, то в таблице `books` все значения столбцов `author_id` автоматически преобразуются в **NULL**.

# Применение фильтрации при удалении данных

Во всех предыдущих случаях мы удаляли (или пытались удалить) полностью все строки из таблицы. Но что если нам надо удалить не все строки, а только какие-то конкретные - например, конкретного клиента из таблицы **clients**? Решение элементарное - надо в очередной раз применить оператор **WHERE**, после которого можно написать необходимые нам условия. Для примера удалим из таблицы **clients** все строки, у которых время создания (**created\_at**) больше или равно 2023-07-01:

```
DELETE FROM clients WHERE created_at >= '2023-07-01';
```

<u>id</u>	name	verified	created_at	position
4	John Smith	1	2023-01-01	1
5	Martin O'Brien	1	2023-01-03	3
6	Mary Sue	1	2023-01-03	4
7	Joe Maldini	1	2023-01-04	6
8	Rolf Meier	1	2023-01-03	5
12	Josh Dolton	1	2023-01-01	2

## Удаление с подзапросом в очищаемую таблицу (проблема)

Как и в случае с обновлением, при удалении нельзя напрямую обращаться к той таблице, из которой мы сейчас удаляем. Например, просто для эксперимента попытаемся удалить из таблицы все строки, кроме тех, у кого **id** не является четным числом. Это можно сделать и без подзапроса, но для наглядности сделаем это именно так:

```
DELETE FROM clients  
WHERE id NOT IN (SELECT id FROM clients WHERE id % 2 = 0);
```

В ответ мы получим ошибку - во время удаления из таблицы нельзя зачитывать данные.

## Удаление с подзапросом в очищаемую таблицу (решение)

Решение в этом случае такое же, как и при обновлении - зачитать данные не из таблицы, а из выборки из таблицы. Далее приведен пример такого успешного запроса:

```
DELETE FROM clients
WHERE id NOT IN (
    SELECT a.id FROM (SELECT id FROM clients WHERE id % 2 = 0) a
);
```

<u>id</u>	name	verified	created_at	position
4	John Smith	1	2023-01-01	1
6	Mary Sue	1	2023-01-03	4
8	Rolf Meier	1	2023-01-03	5
12	Josh Dolton	1	2023-01-01	2

## Удаление с подзапросом во внешнюю таблицу

Фильтрации по значениям из какой-либо внешней таблицы не требуют никаких ухищрений и проходят абсолютно стандартно. Например, удалим все строки из таблицы **clients**, имена (**name**) которых отсутствуют в таблице **guests**:

```
DELETE FROM clients  
WHERE name NOT IN (SELECT name FROM guests);
```

<u>id</u>	name	verified	created_at	position
6	Mary Sue	1	2023-01-03	4

## Сортировка и ограничение по количеству при удалении (проблема)

Еще одна общая черта между осуществлением обновления и удаления состоит в том, что удаление поддерживает сортировку (**ORDER BY**) и ограничение по количеству (**LIMIT**). Воспроизведем случай со вставкой в таблицу **clients** строки, у которой будет имя (**name**), уже присутствующее в другой, уже ранее сохраненной строке. В данный момент у нас только одна строка с именем Mary Sue, поэтому и продублируем это имя, но дату поставим более позднюю - 2023-01-13:

```
INSERT INTO clients (name, verified, created_at, position) VALUES ('Mary Sue', 1, '2023-01-13', 5);
```

<u>id</u>	name	verified	created_at	position
6	Mary Sue	1	2023-01-03	4
23	Mary Sue	1	2023-01-13	5

Теперь по аналогии с обновлением осознаем, что совершили ошибку - такую строку вообще не надо было вставлять. Кроме того, мы уже не помним ничего об этой вставке кроме того, что дата вставки **created\_at** была позднее, чем у первой строки с таким именем.

## Сортировка и ограничение по количеству при удалении (решение проблемы)

Для решения проблемы из предыдущего шага нам надо получить все строки с таким именем (**WHERE name = 'Mary Sue'**), затем отсортировать (**ORDER BY created\_at**) по времени создания от позднего к раннему (чтобы более поздняя строка оказалась первой), затем взять только первую строку (**LIMIT 1**) и удалить только ее:

```
DELETE FROM clients
WHERE name = 'Mary Sue'
ORDER BY created_at DESC
LIMIT 1;
```

<u>id</u>	name	verified	created_at	position
6	Mary Sue	1	2023-01-03	4