MySQL

ОГРАНИЧЕНИЯ, НОРМАЛЬНЫЕ ФОРМЫ И СВЯЗИ

СОДЕРЖАНИЕ

ОГРАНИЧЕНИЯ

ВВЕДЕНИЕ

- В разделах о создании и модификации структурах таблиц мы встречались с некоторыми ограничениями, которые можно накладывать на столбцы и их значения. Самым ярким представителем таких ограничений является атрибут **NOT NULL**. Как мы помним, столбец, имеющий данный атрибут, должен обязательно получить значение при вставке строки в таблицу, которой этот столбец принадлежит. Также некоторым подвидом ограничения можно считать атрибут **UNSIGNED**, который разрешает только положительные числа для своего столбца. Однако в MySQL существует ряд специальных явных ограничений, которые могут иметь гораздо более сложную и специфичную логику. Эти ограничения мы и рассмотрим в данной главе.

CHECK

- **CHECK** позволяет применить условие (его мы определяем сами, но оно должно включать проверки на **>, >=, <, <=, =, <>, !=**) для проверки значения, которое мы пытаемся вставить в столбец. Если проверка не будет пройдена, то данные не будут вставлены в таблицу, а MySQL вернет ошибку.

- Приведем пример такого ограничения. Допустим, нам необходимо создать в базе данных таблицу разработчиков, которые будут работать над проектом. У нас сложный проект, и мы договорились, что будем принимать только опытных людей - с опытом работы от 5 лет. Чтобы у рекрутеров не возникало даже и мысли вписывать в таблицу новичков, при создании таблицы поставим на поле опыт (experience) ограничение **CHECK**:

```
-- в поле experience могут быть вставлены числа только >= 5
CREATE TABLE developers (
    firstname VARCHAR(255) NOT NULL,
    lastname VARCHAR(255) NOT NULL,
    position VARCHAR(255) NOT NULL,
    age INT UNSIGNED NOT NULL,
    experience INT UNSIGNED NOT NULL CHECK(experience >= 5)
);
```

- В рамках одного **CHECK** ограничения можно вставлять не одно условие, а сразу несколько. Например, предположим, что для

нашего проекта нам в качестве разработчиков нужны только программисты и дизайнеры. Мы можем применить тип **ENUM**, но **CHECK** нам дает альтернативу для решения нашей проблемы:

```
/*
    в столбце position могут быть вставлены только строки
    'programmer' или 'designer'.
*/
CREATE TABLE developers (
    firstname VARCHAR(255) NOT NULL,
    lastname VARCHAR(255) NOT NULL,
    position VARCHAR(255) NOT NULL CHECK(position =
'administrator' OR position = 'designer'),
    age INT UNSIGNED NOT NULL,
    experience INT UNSIGNED NOT NULL CHECK(experience >= 5)
);
```

- Ограничение **CHECK** можно вставлять не только рядом с тем столбцом, где оно используется, можно вставлять его так же, как вставляется столбец:

```
CREATE TABLE developers (
    firstname VARCHAR(255) NOT NULL,
    lastname VARCHAR(255) NOT NULL,
    position VARCHAR(255) NOT NULL,
    age INT UNSIGNED NOT NULL,
    experience INT UNSIGNED NOT NULL,
    CHECK(position = 'administrator' OR position =
'designer'),
    CHECK(experience >= 5)
);
```

- Более того, все ограничения **CHECK** можно указать в рамках одной конструкции (хотя так делать не стоит, т.к. потом будет сложнее удалять конкретные ограничения):

```
CREATE TABLE developers (
    firstname VARCHAR(255) NOT NULL,
    lastname VARCHAR(255) NOT NULL,
    position VARCHAR(255) NOT NULL,
    age INT UNSIGNED NOT NULL,
    experience INT UNSIGNED NOT NULL,
    CHECK((position = 'administrator' OR position =
'designer') AND (experience >= 5))
);
```

- Важно знать следующий факт - любому ограничению при его создании назначается имя. Если мы не ставим имя сами, то MySQL присваивает это имя автоматически. Однако лучше всего это делать явно, т.к. СУБД присваивает не слишком читаемые названия:

```
-- явное присваивание названия ограничениям
CREATE TABLE developers (
    firstname VARCHAR(255) NOT NULL,
    lastname VARCHAR(255) NOT NULL,
    position VARCHAR(255) NOT NULL,
    age INT UNSIGNED NOT NULL,
    experience INT UNSIGNED NOT NULL,
    CONSTRAINT position_limit CHECK(position =
'administrator' OR position = 'designer'),
    CONSTRAINT experience_limit CHECK(experience >= 5)
);
```

- Само собой, ограничения **CHECK** можно создавать не только в момент создания таблицы, но и после того, как таблица уже существует. Как и ранее, можно создавать **CHECK** и с названием, которое назначит сама MySQL:

```
-- создание ограничения с явным назначением имени
ALTER TABLE developers ADD CONSTRAINT age_limit CHECK(age >=
18);

-- создание ограничения, которому название даст сама MySQL
ALTER TABLE developers ADD CHECK(age >= 18);
```

- Наконец, ограничение **CHECK** можно удалить - это всегда делается при помощи имени нашего ограничения:

```
-- 1-й способ
ALTER TABLE developers DROP CHECK age_limit;

-- 2-й способ
ALTER TABLE developers DROP CONSTRAINT age_limit;
```

UNIQUE

- Еще один интересный подвид ограничения предоставляется при помощи атрибута **UNIQUE** - как можно понять из названия, он следит за тем, чтобы в столбце таблицы не было повторяющихся

значений - только уникальные. При попытке вставить значение, которое уже существует, MySQL будет возвращать ошибку и ничего вставлять не будет. Важный момент - мы можем создать столько столбцов с ограничением **UNIQUE**, сколько захотим.

- Представим, что мы проектируем таблицу граждан некого государства - очевидно, что могут быть одинаковые имена и одинаковые фамилии, но одинаковый персональный код быть не может. Поэтому столбец персонального кода является идеальным кандидатом для ограничения **UNIQUE** (в этом случае MySQL сама присвоит название ограничению):

```
CREATE TABLE citizens(
    firstname VARCHAR(255) NOT NULL,
    lastname VARCHAR(255) NOT NULL,
    personal_code VARCHAR(255) NOT NULL UNIQUE
);
```

- Если же мы хотим явно указать название ограничению, то мы должны применить следующий синтаксис:

```
-- 1-й способ
CREATE TABLE citizens(
    firstname VARCHAR(255) NOT NULL,
    lastname VARCHAR(255) NOT NULL,
    personal_code VARCHAR(255) NOT NULL,
    CONSTRAINT personal_code_unq UNIQUE(personal_code)
);

-- 2-й способ
CREATE TABLE citizens(
    firstname VARCHAR(255) NOT NULL,
    lastname VARCHAR(255) NOT NULL,
    personal_code VARCHAR(255) NOT NULL,
    UNIQUE KEY personal_code_unq (personal_code)
);
```

- **UNIQUE** имеет одну особенность, которую надо всегда помнить - если данное ограничение было присвоено столбцу, который разрешает вставлять в себя **NULL**, то в этот столбец значение **NULL** можно вставлять больше одного раза - т.е. в данном контексте **UNIQUE** не будет применять свою проверку. Дело в том, что **NULL** трактуется как “отсутствие значение” и оно просто игнорируется. Далее приведем пример таблицы, в которой может произойти такая ситуация:

```
CREATE TABLE citizens(
    firstname VARCHAR(255) NOT NULL,
    lastname VARCHAR(255) NOT NULL,
    personal_code VARCHAR(255) NULL UNIQUE
);
```

- Еще одной отличительной особенностью **UNIQUE** является то, что он может быть применен не только для одного столбца, а сразу для нескольких столбцов. Это значит, что комбинация значений этих столбцов должна быть уникальна, но значение какого-либо столбца из одной комбинации вполне может совпадать со значением такого же столбца из другой комбинации.

- Приведем пример **UNIQUE** для нескольких столбцов сразу. Допустим, что у нас есть таблица **staff**, в рамках которой один и тот же человек может занимать несколько должностей. Это значит, что в данной таблице могут быть несколько одинаковых работников и несколько одинаковых должностей, но не должно быть комбинаций, которые включают в себя одного и того же работника и должность одновременно. Далее пример:

```
CREATE TABLE staff(
    employee VARCHAR(255) NOT NULL,
    position VARCHAR(255) NOT NULL,
    CONSTRAINT emp_pos_unq UNIQUE(employee, position)
);
```

Эта таблица может быть заполнена следующим образом:

employee	position
John Smith	programmer
John Smith	designer
Mary Evans	programmer

Однако вот так заполнена эта таблица быть не может:

employee	position
John Smith	programmer
John Smith	programmer
Mary Evans	programmer

- Ситуация со столбцами, в которых разрешено значение **NULL**, касается и **UNIQUE** для нескольких столбцов. Создадим таблицу для такого случая:

```
CREATE TABLE staff(
    employee VARCHAR(255) NULL,
    position VARCHAR(255) NULL,
    CONSTRAINT emp_pos_unq UNIQUE(employee, position)
);
```

В этом случае мы можем заполнить ее так:

employee	position
John Smith	NULL
John Smith	NULL
NULL	programmer
NULL	programmer

- Как и случае с **CHECK**, создать ограничение **UNIQUE** можно и после создания таблицы.

Пример для создания ограничения с явным присвоением имени:

```
-- создание таблицы
CREATE TABLE staff(
    employee VARCHAR(255) NOT NULL,
    position VARCHAR(255) NOT NULL
);

-- создание ограничения для таблицы (явное указание имени)
ALTER TABLE staff ADD CONSTRAINT emp_pos_unq UNIQUE(employee,
position);
```

Пример для создания ограничения без явного присвоения имени:

```
-- создание таблицы
CREATE TABLE citizens(
    firstname VARCHAR(255) NOT NULL,
    lastname VARCHAR(255) NOT NULL,
    personal_code VARCHAR(255) NOT NULL
);

-- создание ограничения для таблицы (без указания имени)
ALTER TABLE citizens ADD UNIQUE(personal_code);
```

- Удаляется ограничение следующим образом:

```
-- 1-й способ
ALTER TABLE staff DROP KEY emp_pos_unq;

-- 2-й способ
ALTER TABLE staff DROP CONSTRAINT emp_pos_unq;
```

FOREIGN KEY

- Еще один способ ограничить значения столбца связан уже не с одной таблицей, а может касаться двух или более таблиц. Это так называемое ограничение внешнего ключа (**FOREIGN KEY**)- при его применении в столбец какой-либо таблице (или нескольких таблицах) можно вставить только то значение, какое уже вставлено в определенный столбец какой-то другой конкретной таблицы. Важный момент - на столбец таблицы с изначальными значениями обязательно должен быть поставлен так называемый индекс (**INDEX**). Подробнее об индексам мы поговорим в дальнейших разделах, но пока запомним, что индекс способствует ускорению поиска по таблице в контексте столбца, которому установлен этот индекс. Еще полезно запомнить, что ограничение **UNIQUE** и ограничение **PRIMARY KEY** (его мы рассмотрим в следующем подразделе) получают индекс автоматически при создании.

- Столбец таблицы с изначальными значениями и столбец таблицы, которая ссылается на изначальную таблицу, должны иметь одинаковый тип - в противном случае MySQL не позволит создать ограничение внешнего ключа.

- Приведем пример создания данного ограничения. Сначала создадим таблицу clients с изначальными значениями:

```
CREATE TABLE clients(
    id INT UNSIGNED NOT NULL,
    firstname VARCHAR(255) NOT NULL,
    lastname VARCHAR(255) NOT NULL,
    INDEX(id)
);
```

Затем создадим таблицу orders со столбцом client_id, который ссылается на столбец id первоначальной таблицы clients (таким образом, в столбце client_id таблицы orders могут хранится

только те значения, которые уже записаны в столбце id таблицы clients):

```
CREATE TABLE orders(
    client_id INT UNSIGNED NOT NULL,
    reference VARCHAR(255) NOT NULL UNIQUE,
    price NUMERIC(20,2),
    FOREIGN KEY(client_id) REFERENCES clients(id)
);
```

Предположим, что в таблице **clients** хранятся следующие записи:

id	firstname	lastname
1	John	Smith
2	Mary	Evans
3	Joanna	Brooks

При таких значениях в первой таблице во второй таблице в столбце **client_id** могут хранится только 1, 2, 3 - при попытке вставить в **client_id** другие значения MySQL вернет ошибку и ничего вставлять не будет. Далее пример таблицы **orders** с правильными значениями:

client_id	reference	price
1	23924384745	99.99
2	48499494432	115.75
3	93843746626	20.17
2	66884848848	65.00

- Как и во всех предыдущих случая, ограничение **FOREIGN KEY** имеет одно исключение, связанное со значением **NULL**. Если столбцу таблицы, зависящему от столбца первоначальной таблицы, разрешить вставлять значение **NULL**, то в зависимый столбец можно будет вставлять значение, которое отлично от значений первоначального столбца - можно понять, что это значение **NULL**. Чтобы продемонстрировать наши утверждения, обновим столбец **client_id** таблицы orders - разрешим вставлять вышеупомянутый **NULL**:

```
ALTER TABLE orders MODIFY COLUMN client_id INT UNSIGNED NULL;
```

Теперь в таблице **orders** могут хранится следующие значения:

client_id	reference	price
1	23924384745	99.99
NULL	34563394432	46.22
NULL	16702746626	21.15

- Ограничение **FOREIGN KEY** предоставляет еще одну удобную возможность - он позволяет определить поведение значений в том случае, если значение в изначальной таблице было удалено или обновлено. Всего есть 3 варианта:

  1. Запретить удаление и модификацию значения первоначального столбца - это поведение по умолчанию, но можно явно его прописать при помощи слова **RESTRICT**. Данный пример запретит менять и обновлять столбец id в таблице clients:

```
/*
    поведение по умолчанию (неявно запрещает удалять и
    обновлять значение id в clients)
*/
CREATE TABLE orders(
    client_id INT UNSIGNED NOT NULL,
    reference VARCHAR(255) NOT NULL UNIQUE,
    price NUMERIC(20,2),
    FOREIGN KEY(client_id) REFERENCES clients(id)
);

-- явный запрет удалять и обновлять значение id в clients
CREATE TABLE orders(
    client_id INT UNSIGNED NOT NULL,
    reference VARCHAR(255) NOT NULL UNIQUE,
    price NUMERIC(20,2),
    FOREIGN KEY(client_id) REFERENCES clients(id) ON DELETE
RESTRICT ON UPDATE RESTRICT
);
```

  2. При удалении и обновлении значения первоначального столбца удалять всю строку из зависимой таблицы (при

удалении значения первоначального столбца) или обновлять значение зависимого столбца (при обновлении значения первоначального столбца). Осуществляется при помощи слова **CASCADE**:

```
CREATE TABLE orders(
    client_id INT UNSIGNED NOT NULL,
    reference VARCHAR(255) NOT NULL UNIQUE,
    price NUMERIC(20,2),
    FOREIGN KEY(client_id) REFERENCES clients(id) ON DELETE
CASCADE ON UPDATE CASCADE
);
```

3. Если в зависимом столбце разрешено значение **NULL**, то при удалении или обновлении значения первоначального столбца зависимому столбцу в качестве значения автоматически вставить **NULL**. Осуществляется при помощи конструкции **SET NULL**:

```
CREATE TABLE orders(
    client_id INT UNSIGNED NULL,
    reference VARCHAR(255) NOT NULL UNIQUE,
    price NUMERIC(20,2),
    FOREIGN KEY(client_id) REFERENCES clients(id) ON DELETE
SET NULL ON UPDATE SET NULL
);
```

- Необязательно указывать одинаковое поведение при удалении и обновлении. Можно сделать так, чтобы обновление первоначального столбца была запрещено (**RESTRICT**), а при удалении удалять всю строку (**CASCADE**):

```
CREATE TABLE orders(
    client_id INT UNSIGNED NOT NULL,
    reference VARCHAR(255) NOT NULL UNIQUE,
    price NUMERIC(20,2),
    FOREIGN KEY(client_id) REFERENCES clients(id) ON DELETE
CASCADE ON UPDATE RESTRICT
);
```

- Как мы помним, если мы не указываем явно название ограничение, то это за нас делает MySQL. Если же мы хотим

сами указать название, то это можно сделать следующим образом:

```
CREATE TABLE orders(
    client_id INT UNSIGNED NOT NULL,
    reference VARCHAR(255) NOT NULL UNIQUE,
    price NUMERIC(20,2),
    CONSTRAINT control_client_id FOREIGN KEY(client_id)
REFERENCES clients(id)
);
```

- Добавление внешнего ключа уже после создания таблицы происходит стандартно:

```
-- создание ограничения с явным назначением имени
ALTER TABLE orders ADD CONSTRAINT control_client_id  FOREIGN
KEY(client_id) REFERENCES clients(id);

-- создание ограничения, которому название даст сама MySQL
ALTER TABLE orders ADD FOREIGN KEY(client_id) REFERENCES
clients(id);
```

- Удалять внешний ключ можно двумя способами:

```
-- 1-й способ
ALTER TABLE orders DROP FOREIGN KEY control_client_id;

-- 2-й способ
ALTER TABLE orders DROP CONSTRAINT control_client_id;
```

PRIMARY KEY

- При создании таблиц и столбцов нам рано или поздно должна прийти в голову следующая мысль - “Каким образом мы будем осуществлять выборку данных из таблицы, если нам нужна только какая-то конкретная строка?”. Также логичной будет такая мысль - “Каким удобным способом мы можем передать куда-то информацию о записи в таблице, но не целиком, а только частично, т.к. нам не нужная обработка этой записи в данный момент, но может потребоваться в дальнейшем?”. Для решения этих проблем в таблицах принято использовать называемый первичный ключ (**PRIMARY KEY**) - столбец в таблице, который однозначно и навсегда идентифицирует каждую запись.

Важно знать, что первичный ключ может включать в себя несколько столбцов, но опять же - комбинация их значений должна однозначно идентифицировать каждую запись в таблице.

- Из всего сказанного можно сделать вывод, что ограничение **PRIMARY KEY** является братом-близнецов уже рассмотренного ограничения **UNIQUE**. Они действительно очень похожи - первичный ключ также должен содержать уникальное в контексте таблицы значение и также при создании автоматически получает индекс, но в отличие от **UNIQUE** он, во-первых, никогда не может быть равен **NULL** и, во-вторых, в контексте таблицы может только один первичный ключ.

- Далее создадим таблицу с первичным ключем - в данном случае выберем вариант с одним столбцом id (в результате id должен получать только уникальные значение - в противном случае MySQL не будет вставлять данные и вернёт ошибку):

```
CREATE TABLE products(
    id INT UNSIGNED PRIMARY KEY,
    name VARCHAR(255) NOT NULL,
    price DECIMAL(20,2) NOT NULL,
    description TEXT NOT NULL
);
```

- Как обычно, повторим создание такой же таблицы, но уже с явным указанием названия ограничения **PRIMARY KEY**:

```
CREATE TABLE products(
    id INT UNSIGNED,
    name VARCHAR(255) NOT NULL,
    price DECIMAL(20,2) NOT NULL,
    description TEXT NOT NULL,
    CONSTRAINT id_pk PRIMARY KEY(id)
);
```

- Также приведем пример таблицы, которая имеет составной первичный ключ. В таблице есть два столбца (version и type), комбинация значений которых должна быть уникальна. Несмотря на то, что у этих двух столбцов не указана конструкция **NOT NULL**, **PRIMARY KEY** автоматически запрещает вставлять **NULL** в качестве значения любого из этих столбцов:

```
CREATE TABLE documents (
    version INT UNSIGNED,
    type VARCHAR(255),
    name VARCHAR(255),
    content TEXT NOT NULL,
    CONSTRAINT ver_type_pk PRIMARY KEY(version, type)
);
```

- И все же в контексте первичного ключа (кстати, это же относится и к ограничению **UNIQUE**) есть одна возможность явно не указывать значение при вставке записи в таблицу. Если столбец с ограничением **PRIMARY KEY** или **UNIQUE** имеет тип целого числа (**TINYINT**, **INT**, **BIGINT** и т.д.), то ему можно присвоить атрибут **AUTO_INCREMENT**. Данный автоинкремент автоматически генерирует для своего столбца последовательное численное значение в том случае, когда в его таблицу вставляется запись, и в этой записи явно не предоставляется значение для столбца, содержащего автоинкремент.

- В рамках одной таблицы разрешен только один столбец с автоинкрементом.

- По умолчанию автоинкремент имеет изначальное значение 1 - таким образом, при первой вставке в столбце будет вставлено число 1, при второй вставке - 2, при третьей - 3 и т.д.

- Если в столбец с автоинкрементом будет вручную вставлено значение, которое превышает значение последнего автоматически созданного числа, то автоинкремент возьмет это значение, увеличенное на единицу, в качестве новой точки отсчета для генерации следующего значения.

- Приведем пример автоинкремента - для этого возьмем уже рассмотренную таблицу products и ее столбец id:

```
CREATE TABLE products(
    id INT UNSIGNED PRIMARY KEY AUTO_INCREMENT,
    name VARCHAR(255) NOT NULL,
    price DECIMAL(20,2) NOT NULL,
    description TEXT NOT NULL
);
```

- Интересно заметить, что при создании таблицы можно поставить изначальное значение инкремента, чтобы он начинался не с единицы, а, например, со 100:

```
CREATE TABLE products(
    id INT UNSIGNED PRIMARY KEY AUTO_INCREMENT,
    name VARCHAR(255) NOT NULL,
    price DECIMAL(20,2) NOT NULL,
    description TEXT NOT NULL
) AUTO_INCREMENT = 100;
```

- Теперь опять вернемся к первичному ключу и рассмотрим то, как его добавлять уже после того, как таблица была уже создана:

```
-- создаем таблицу без первичного ключа
CREATE TABLE products(
    id INT UNSIGNED,
    name VARCHAR(255) NOT NULL,
    price DECIMAL(20,2) NOT NULL,
    description TEXT NOT NULL
);

-- 1-й способ создания ограничения для столбца id
ALTER TABLE products ADD PRIMARY KEY(id);

-- 2-й способ создания ограничения для столбца id (с именем)
ALTER TABLE products ADD CONSTRAINT id_pk PRIMARY KEY(id);
```

- Удалить первичный ключ можно следующим образом:

```
ALTER TABLE products DROP PRIMARY KEY;
```

НОРМАЛИЗАЦИЯ И НОРМАЛЬНЫЕ ФОРМЫ

ВВЕДЕНИЕ

- Умение создавать таблицы в базе данных - очень важная вещь в работе разработчика и аналитика, но не менее важно создавать эти таблицы правильно. В том случае, если при проектировании были допущены ошибки, в процессе работы с базой данных могут происходить различные аномалии. Например, при вставке могут понадобиться данные, которых еще не существуют, при удалении удалиться данные, которые следовало сохранять любой ценой, а при модификации одна половина целевых значений обновиться, а другая по ошибке останется неизменной.

- Рассмотрим примеры всех трех аномалий - начнем с аномалии вставки. Предположит, что у нас есть таблица преподавателей, которые читают людям лекции о компьютерных технологиях:

Name	Course	Room
John Smith	MySQL	21
Juan Lopez	Java	18
Ashley Johns	Python	15

  Представим случай, что мы наняли нового талантливого преподавателя, но обстоятельства складываются так, что мы не знаем пока, что именно он будет преподавать. По закону мы обязаны занести его в базу данных, но что же нам указать в столбцах Course и Room? Единственным чисто техническим решением будет вставить в эти столбцы специальное значение **NULL**, но надо признать, что вся эта ситуация свидетельствует о том, что при планировании что-то было сделано неправильно.

- Далее несколько слов об аномалии редактирования. Предположим, что у нас есть все та же таблица преподавателей, но один из преподавателей - Ashley Johns - вышла замуж и сменила фамилию на Monroe. Кроме того, сделаем дополнительное предположение, что Ashley преподает не один курс, а несколько:

Name	Course	Room
John Smith	MySQL	21
Ashley ~~Johns~~ Monroe	Java	18
Ashley ~~Johns~~ Monroe	Python	15

Мы видим, что всесто того, чтобы изменить фамилию один раз, пришлось это сделать несколько раз. Т.е. Были осуществлены лишние действия, кроме того, мы можем сделать ошибку - обновить фамилию только один раз, а все остальные случаи оставить без изменения.

- Рассмотрим последнюю аномалию - аномалию удаления. Обратимся к нашей таблице с преподавателями в последний раз. Предположим, что Ashley решила уволиться и уехать с супругом в другую страну. В этом случае мы обязаны полностью удалить ее из таблицы.

Name	Course	Room
John Smith	MySQL	21
~~Ashley Monroe~~	~~Java~~	~~18~~
~~Ashley Monroe~~	~~Python~~	~~15~~

Удаляя Ashley, мы удалили одновременно и информацию о курсе, и информацию о месте его прохождения. Опять же, чисто технически возможно не удалять всю строку, а заменить имя преподавателя на **NULL**. Однако получится, что у нас в каждом поле может быть значение **NULL** - это может привести к тому, что рано или поздно у нас в таблице будет множество строк полностью без информации (**NULL**, **NULL**, **NULL**).

- Во всех предыдущих случаем самым главным врагом нормальной работы таблицы выступала избыточность данных. **Избыточность данных** - явление, когда одни и те же данные хранятся в базе в нескольких местах. Как уже было показано, с этим можно справляться на небольших проектах (подставлять **NULL** и т.д.), однако в таблицах с миллионами строк это может привести к чудовищным последствиям.

- Для борьбы с избыточностью уже упоминавшимся математиком Эдгаром Франком Коддом была предложена теория нормализации. **Нормализация** - постепенное преобразование базы данных путем декомпозиции отношений (таблиц), то есть разбиения одной таблицы на несколько. Существует несколько (6 основных и 2 дополнительных) этапов нормализации, которые позволяют устранить избыточность. Эти этапы называются **нормальными формами**:

  1. **Первая нормальная форма (1NF)**
  2. **Вторая нормальная форма (2NF)**
  3. **Третья нормальная форма (3NF)**
  4. Нормальная форма Бойса-Кодда (BCNF)
  5. Четвертая нормальная форма (4NF)
  6. Пятая нормальная форма (5NF)
  7. Доменно-ключевая нормальная форма (DKNF)
  8. Шестая нормальная форма (6NF)

- В рамках нашего курса мы рассмотрим только первые три формы, т.к. формы, начиная с формы Бойса-Кодда и четвертой формы, уже перестают приносить пользу при реальной разработке. Они крайне полезны с теоретической точки зрения, но чисто практически они могут создать другие проблемы, уже не связанные с избыточностью.[1]

[1] https://learn.microsoft.com/ru-ru/office/troubleshoot/access/database-normalization-description

ПЕРВАЯ НОРМАЛЬНАЯ ФОРМА

- Таблица находится в первой нормальной форме, если соблюдены два принципа:
  1. если ни одна из строк таблицы не содержит в любом своем столбце более одного значения (т.н. атомарность)
  2. в таблице не должно быть дублирующихся строк

- Рассмотрим таблицу персонала, где указаны имя, должность и номера телефонов сотрудников некой компании:

Name	**Position**	**PhoneNumber**
John Smith	Programmer	8274827347, 234878754481, 647363456784
Nicole Adams	Designer	982748273478
Harry Thomas	DB Administrator	28274827347
Anna Devine	Principal	827482734779
Anna Devine	Principal	827482734779

Данные внутри этой таблицы явно говорят нам о то, что она не находится в первой нормальной форме - программист John Smith имеет несколько телефонных номеров в одном столбце, а директор Anna Devine зачем-то повторяется два раза. Особую проблему представляет собой номер телефона - если нам надо будет его поменять, то мы должны зайти в столбец номера каждой строки, каким-либо образом преобразовать этот столбец в несколько номеров, проверить не соответсвует ли один из них тому, что нам надо менять, а только затем произвести модификацию. Надо заметить, что при большом количестве строк мы не всегда будем уверены, какой разделитель номеров был использован, что делает разделение практически невозможным. Поэтому, если изменяемым номером будет 8274827347, то применяя такую логику, мы можем случайно поменять все телефоны всем сотрудникам, чего мы конечно-же не хотим.

- Чтобы привести эту таблицу к первой нормальной форме, сначала необходимо удалить все дубликаты строк (можно догадаться, что необходимо убрать одну из двух строк с директором), а вот с программистом стоит поступить несколько сложнее - мы должны повторить его имя и должность столько

раз, сколько у него телефонов, каждый раз вписывая в соответствующий столбец разный номер телефона:

Name	**Position**	**PhoneNumber**
John Smith	Programmer	8274827347
John Smith	Programmer	234878754481
John Smith	Programmer	647363456784
Nicole Adams	Designer	982748273478
Harry Thomas	DB Administrator	28274827347
Anna Devine	Principal	82748273479

- Получившаяся таблица может выглядеть немного странно, кроме того, в ней по прежнему есть аномалии, но тем не менее она уже является полноценной реляционной таблицей, с которой можно работать стандартными средствами SQL.

ВТОРАЯ НОРМАЛЬНАЯ ФОРМА

- Таблица находится во второй нормальной форме, если соблюдены три принципа:
  1. таблица уже находится в первой нормальной форме
  2. таблица имеет ключ (простой или составной), уникально идентифицирующий каждую строку таблицы
  3. все неключевые столбцы таблицы должны зависеть сразу от всех столбцов, входящих в ключ

- С ключем, уникально идентифицирующий каждую строку таблицы, мы уже сталкивались во время анализа ограничений - в MySQL с этой задачей отлично справляется **PRIMARY KEY**, который позволяет использовать как простые, так и составные ключи.

- Во-первых рассмотрим случай с простым ключем - таблицу учащихся университета с 3 столбцами - идентификатор (ID), имя (Name) и статус (Status). Публичным ключем сделаем столбец ID:

ID	**Name**	**State**
1	Jānis Bērziņš	Student
2	Иван Иванов	Student
3	John Smith	Free Listener

  Мы видим, что в таблице есть ключевой столбец ID, нет дублирующихся строк, также в таблице нет столбцов, хранящих более одного значения, кроме того, все неключевые столбцы (Name, State) зависят от ключевого столбца ID, т.е. зная значение всех столбцов ключа (в данном случае одного столбца), мы можем точно узнать имя и статус студента. Таким образом, эта таблица уже находится во второй нормальной форме - в этом контексте нам ничего делать не надо.

- Теперь рассмотрим ситуацию, в которой таблица имеет составной ключ. Предположим, что у нас есть таблица проектов, в который занесены названия проектов (project), ответственный за проект (curator), должность ответственного (position) и количество месяцев (months), выделенное для реализации проекта. В данном случае первичным ключем будут две колонки - название проекта и имя куратора.

project	**curator**	**position**	**months**
Site Development	Barbara Schmidt	System Analytic	2
Mobile Application Development	Michael Doe	Swift Programmer	1
CRM Development	Barbara Schmidt	System Analytic	24

Такая таблица уже не соответствует второй нормальной форме, т.к. неключевые столбцы не зависят полностью от всех ключевых столбцов сразу. Зная только название проекта можем ли мы сразу узнать должность куратора? Нет, не можем. Зато можем сразу узнать количество месяцев для реализации проекта, что является несоответствием. Есть проблемы и со вторым ключевым столбцом. Зная только имя куратора, можем ли мы сразу узнать количество месяцев реализации? Нет, не можем. Но зато сразу можем узнать должность, что опять является несоответствием.

- Чтобы такое несоответствие разрешить, произведем декомпозицию - т.е. разделим эту таблицу на три меньшие таблицы.

Таблица **employees**

id	**name**	**position**
1	Barbara Schmidt	System Analytic
2	Michael Doe	Swift Programmer

Таблица **projects**

id	**project**	**months**
1	Site Development	2
2	Mobile Application Development	1
3	CRM Development	24

Таблица **projects_curators**

project_id	**employee_id**
1	1
2	2
3	1

В результирующей таблице projects_curators теперь нет столбцов, которые не зависят от ключевых столбцов, более того, там вообще нет неключевых столбцов и все их комбинации уникальны - в результате нам ничего не остается, как признать эту таблицу находящейся во второй нормальной форме.

ТРЕТЬЯ НОРМАЛЬНАЯ ФОРМА

- Таблица находится в третьей нормальной форме, если соблюдены два принципа:
  1. таблица уже находится во второй нормальной форме
  2. все неключевые столбцы таблицы должны зависеть сразу от всех столбцов, входящих в ключ, но не должны зависеть от неключевых столбцов

- Можно понять, что третья нормальная форма является простым усилением второй формы - просто дополнительно запрещаются любые зависимости между вторичными столбцами (это несоответствие также называется **транзитивной зависимостью**). Для того, чтобы понять проблему, предположим, что у нас в очередной раз есть таблица работников некой компании - в таблице есть идентификатор - первичный ключ (id), имя работника (name), его должность (position), отдел (division) и описание отдела (division_description):

id	**name**	**position**	**division**	**division_description**
1	Barbara Schmidt	System Analytic	Research and Development	Analytics and Innovations
2	Joe D'Amato	Android Programmer	Mobile Development	Develops Mobile Apps
3	John Smith	Web Programmer	Web Development	Develops Sites

Таблица очевидно находится во второй нормальной форме, но сразу же бросается в глаза тот факт, что не все столбцы, связанные с подразделением, связаны с первичным ключом. Дело в том, что при каждом появлении какого-либо подразделения за ним неизменно будет следовать описание этого подразделения. Оба эти столбца не являются ключевыми - следовательно, мы нашли транзитивную зависимость, которая недопустима для третьей нормальной формы. Исправлять будем как обычно - разделением на таблицы меньшего размера.

Таблица **divisions**

id	**division**	**division_description**
1	Research and Development	Analytics and Innovations
2	Mobile Development	Develops Mobile Apps
3	Web Development	Develops Sites

Таблица **employees**

id	**name**	**position**	**division_id**
1	Barbara Schmidt	System Analytic	1
2	Joe D'Amato	Android Programmer	2
3	John Smith	Web Programmer	3

Теперь у нас нет транзитивных зависимостей в таблице с сотрудниками, т.к. ссылка на отдел представлена единым столбцом. Поэтому данная таблица находится в третьей нормальной форме.

СВЯЗИ

ВВЕДЕНИЕ

- Из прошлой главы мы узнали о том как бывает вредно хранить все данные в одной таблице, и какие проблемы можно получить, если не разделять большие таблицы на компактные таблицы без избыточности данных. Однако мы шли от противного - “как нельзя”, а теперь попробуем посмотреть на вопрос с другой стороны - “как можно”. Таким образом, мы рассмотрим типовые способы правильного разделения какой-либо сущности на таблицы. За эту область знаний в реляционной теории отвечают так называемые связи - способы взаимодействия между разными таблицами. Всего есть три стандартные связи - “один к одному”, “один ко многим” и “многие ко многим”.

СВЯЗЬ ОДИН К ОДНОМУ

- В рамках данной связи какая-либо конкретная строка в первой таблице не может иметь более одной связанной строки во второй таблице. Чаще всего это необходимо в том случае, если таблица имеет некоторый уровень нормализации (по крайней мере вторую нормальную форму), однако количество столбцов в ней является слишком большим. Хорошим примером является паспорта и люди (для удобства предположим, что это работники), живущие в стране, где запрещено иметь двойное гражданство. Теоретически всю паспортную информацию можно поместить в таблицу к работникам, но от этого она станет слишком громоздкой. Поэтому разделим нашу проблему на две таблицы - работники и паспорта. Паспорта будут связаны с работниками внешним ключем **employee_id** , который для усиления связи мы сделаем еще и уникальным (**UNIQUE**):

Таблица **employees**

id	**name**	**position**	**division**
1	Barbara Schmidt	System Analytic	Research and Development
2	Joe D’Amato	Android Programmer	Mobile Development
3	John Smith	Web Programmer	Web Development

Таблица **passports**

employee_id	code	nationality	age	sex
1	s8f8798sdf	German	35	woman
2	98dfg87dg8	Italian	22	man
3	gd5f6df6gf	American	30	man

СВЯЗЬ ОДИН КО МНОГИМ

- В связях такого рода одна строка в первой таблице может иметь много связанных строк во второй таблице. Но ни одна из этих совпадающих строк во второй таблице не должна иметь какую-либо другую связанную строку в первой таблице. Это одна из наиболее часто встречающихся связей. Привести пример крайне просто - это клиенты и их заказы. Один клиент может совершить сколько угодно заказов, но каждый конкретный заказ может принадлежать только одному клиенту. Реализуется при помощи простого внешнего ключа в таблице заказов **client_id**, который хранит единственное значение - идентификатор клиента.

Таблица **clients**

id	name	status
1	Alexander Kovalenko	Client
2	John Smith	Business Client

Таблица **orders**

id	client_id	reference	price
22	1	8df7d87fg8d7	99.99
23	2	df87gd8f7g88	55.55
24	1	d9f7g78fg7d8	22.22
25	2	98gh98fhg98f	11.11

СВЯЗЬ МНОГИЕ КО МНОГИМ

- В данном случае любая строка из первой таблицы может иметь сколь угодно связанных строк во второй таблице, и наоборот - любая строка из второй таблицы может иметь сколь угодно связанных строк в первой таблице. Как хороший пример можно упомянуть статьи и теги. Каждой статье можно поставить сколь угодно тегов, а любой тег может появиться у любой статьи. Однако перед нами должен возникнуть закономерный вопрос - а как организовать связь между ними, если в столбце, который мог бы указывать на связанную сущность, можно хранить только одно значение? Если хранить много идентификаторов сразу, то будет нарушен принцип атомарности. Решение очень простое - необходимо создать третью таблицу, где будут храниться комбинации идентификаторов статей и тегов в разных столбцах, которые будут иметь внешние ключи на статьи и теги соответственно:

Таблица **articles**

id	**title**	**content**
1	Basic Principles Every Programmer Must Know	…
2	Queries in MySQL	…
3	DDoS attacks and other incidents	…

Таблица **tags**

id	**tag**
1	development
2	databases
3	artificial intelligence

Таблица **articles_tags**

article_id	**tag_id**
1	1
1	2
2	1
2	2