SQL оператор создания баз данных.

Создание и удаление таблицы. Прикрепление базы данных

Для создания таблиц используется команда **CREATE TABLE**. Общий формальный синтаксис команды **CREATE TABLE**:

```
1    CREATE TABLE название_таблицы
2    (название_столбца1 тип_данных атрибуты_столбца1,
3    название_столбца2 тип_данных атрибуты_столбца2,
4     .....................................................
5     название_столбцаN тип_данных атрибуты_столбцаN,
6     атрибуты_уровня_таблицы
7    )
```

После команды **CREATE TABLE** указывается название таблицы. Имя таблицы выполняет роль ее идентификатора в базе данных, поэтому оно должно быть уникальным. Кроме того, оно не должно начинаться на "sqlite_", поскольку названия таблиц, которые начинаются на "sqlite_", зарезервированы для внутреннего пользования.

Затем после названия таблицы в скобках перечисляются названия столбцов, их типы данных и атрибуты. В самом конце можно определить атрибуты для всей таблицы. Атрибуты столбцов, а также атрибуты таблицы указывать необязательно.

Создадим простейшую таблицу. Перед выполнением команды **CREATE TABLE** вне зависимости, что мы используем - консольный клиент sqlite3, графический клиент DB Browser for SQLite или какой-то другой клиент, вначале откроем базу данных, где мы хотим создать таблицу.

Для создания таблицы выполним следующий скрипт:

```
1    CREATE TABLE Users
2    (
3        Id INTEGER,
4        Name TEXT,
5        Age INTEGER
6    );
```

В данном случае таблица называется "Users". В ней определено три столбца: Id, Age, Name. Первые два столбца представляют идентификатор пользователя и его возраст и имеют тип **INTEGER**, то есть будут хранить числовые значения. Столбец "Name" представляет имя пользователя и имеет тип **TEXT**, то есть представляет строку. В данном случае для каждого столбца определены имя и тип данных, при этом атрибуты столбцов и таблицы в целом отсутствуют.

И в результате выполнения этой команды будет создана таблица Users с тремя столбцами.

Создание таблицы при ее отсутствии

Если мы повторно выполним выше определенную sql-команду для создания таблицы Users, то мы столкнемся с ошибкой - ведь мы уже создали таблицу с таким названием. Но могут быть ситуации, когда мы можем точно не знать или быть не уверены, есть ли в базе данных такая таблица (например, когда мы пишем приложение на каком-нибудь языке программирования и используем базу данных, которая не нами создана). И чтобы избежать ошибки, с помощью выражения **IF NOT EXISTS** мы можем задать создание таблицы, если она не существует:

```
1    CREATE TABLE IF NOT EXISTS Users
2    (
3        Id INTEGER,
4        Name TEXT,
5        Age INTEGER
6    );
```

Если таблицы Users нет, она будет создана. Если она есть, то никаких действий не будет производиться, и ошибки не возникнет.

Прикрепление базы данных

Также мы можем прикрепить базу данных и затем в ней уже создать базу данных.

Для прикрепления базы данных применяется команда **ATTACH DATABASE**:

```
1    ATTACH DATABASE 'C:\sqlite\test.db' AS test;
```

После команды **ATTACH DATABASE** указывается путь к файлу базы данных (в данном случае это путь "C:\sqlite\test.db"). Затем после оператора **AS** идет псевдоним, на который будет проецироваться база данных. То есть в коде для обращения к базе данных "C:\sqlite\test.db" будет применяться имя "text". При обращении к таблице из этой базы данных, сначала указывается псевдоним базы данных и через точку название таблицы:

```
1    псевдоним_бд.таблица
```

Например, создадим таблицу в прикрепленной базе данных:

```
1    ATTACH DATABASE 'C:\sqlite\test.db' AS test;
2    CREATE TABLE test.users
3    (
4        id INTEGER,
5        name TEXT,
6        age INTEGER
7    );
```

Для создания таблицы users в бд test.db название таблицы предваряется псевдонимом: test.users.

И после открытия базы данных test.db в ней можно будеть увидеть таблицу users.

Удаление таблиц

Для удаления таблицы применяется команда **DROP TABLE**, после которой указывается название удаляемой таблицы. Например, удалим таблицу users:

```
DROP TABLE users;
```

По аналогии с созданием таблицы, если мы попытаемся удалить таблицу, которая не существует, то мы столкнемся с ошибкой. В этом случае опять же с помощью операторов **IF EXISTS** проверять наличие таблицы перед удалением:

```
DROP TABLE IF EXISTS users;
```

При определении столбцов таблицы для них необходимо указать тип данных. Каждый столбец имеет определенный тип данных. Для хранения данных в в SQLite применяются следующие типы:

- **NULL**: указывает фактически на отсутствие значения
- **INTEGER**: представляет целое число, которое может быть положительным и отрицательным и в зависимости от своего значения может занимать 1, 2, 3, 4, 6 или 8 байт
- **REAL**: представляет число с плавающей точкой, занимает 8 байт в памяти
- **TEXT**: строка текста в одинарных кавычках, которая сохраняется в кодировке базы данных (UTF-8, UTF-16BE или UTF-16LE)
- **BLOB**: бинарные данные

Стоит отметить, что SQLite оперирует концепцией **классов хранения** или **storage class**. И по сути все эти пять типов называются классами хранения. Концепция классов хранения несколько шире, чем тип данных. Например, класс **INTEGER** по сути объединяет 6 различных целочисленных типов данных разной длины. Однако это больше относится к внутренней работе SQLite. И внешне, например, на уровне определения таблицы и работы с данными мы будем работать с типом **INTEGER**, а не со всеми реальными типами, которые скрываются за этим названием. Поэтому фактически классы хранения ассоциируются с типа данных. И выше представленные пять классов хранения также обычно называют типами данных и данные можно применять при определении столбцов:

```
CREATE TABLE users
(
    id INTEGER,
    name TEXT,
    age INTEGER,
    weight REAL,
    image BLOB
);
```

Кроме того, мы можем применять идентификатор **NUMERIC**. Этот идентификатор не представляет отдельного типа данных. А фактически представляет столбец, который может хранить данные всех пяти выше перечисленных типов (в терминологии SQLite NUMERIC еще называется **type affinity**). Например:

```
1    CREATE TABLE users
2    (
3        id INTEGER,
4        name TEXT,
5        age NUMERIC
6    );
```

При определении столбцов и таблиц для них можно задать ограничения. Ограничения позволяют настроить поведение столбцов и таблиц. Ограничения столбцов указываются после типа столбца:

```
1    имя_столбца тип_столбца ограничения_столбца
```

Ограничения таблицы указываются после определения всех столбцов.

Рассмотрим, какие ограничения столбцов мы можем использовать.

PRIMARY KEY

Атрибут **PRIMARY KEY** задает первичный ключ таблицы. Первичный ключ уникально идентифицирует строку в таблице. Например:

```
1    CREATE TABLE users
2    (
3        id INTEGER PRIMARY KEY,
4        name TEXT,
5        age INTEGER
6    );
```

Здесь столбец id выступает в качестве первичного ключа, он будет уникально идентифицировать строку и его значение должно быть уникальным. То есть у нас не может быть таблице users более одной строки, где в столбце id было бы одно и то же значение.

Установка первичного ключа на уровне таблицы:

```
1    CREATE TABLE users
2    (
3        id INTEGER,
```

```
4       name TEXT,
5       age INTEGER,
6       PRIMARY KEY(id)
7   );
```

Первичный ключ может быть составным. Такой ключ использовать сразу несколько столбцов, чтобы уникально идентифицировать строку в таблице. Например:

```
1   CREATE TABLE users
2   (
3       id INTEGER,
4       name TEXT,
5       age INTEGER,
6       PRIMARY KEY(id, name)
7   );
```

В данном случае в качестве первичного ключа выступает связка столбцов id и name. То есть в таблице users не может быть двух строк, где для обоих из этих полей одновременно были бы одни и те же значения.

AUTOINCREMENT

Ограничение **AUTOINCREMENT** позволяет указать, что значение столбца будет автоматически увеличиваться при добавлении новой строки. Данное ограничение работает для столбцов, которые представляют тип **INTEGER** с ограничением **PRIMARY KEY**:

```
1   DROP TABLE users;
2   CREATE TABLE users
3   (
4       id INTEGER PRIMARY KEY AUTOINCREMENT,
5       name TEXT,
6       age INTEGER
7   );
```

В данном случае значение столбца id каждой новой добавленной строки будет увеличиваться на единицу.

UNIQUE

Ограничение **UNIQUE** указывает, что столбец может хранить только уникальные значения.

```
1  CREATE TABLE users
2  (
3      id INTEGER PRIMARY KEY AUTOINCREMENT,
4      name TEXT,
5      age INTEGER,
6      email TEXT UNIQUE
7  );
```

В данном случае столбец email, который представляет телефон пользователя, может хранить только уникальные значения. И мы не сможем добавить в таблицу две строки, у которых значения для этого столбца будет совпадать.

Также мы можем определить это ограничение на уровне таблицы:

```
1  CREATE TABLE users
2  (
3      id INTEGER PRIMARY KEY AUTOINCREMENT,
4      name TEXT,
5      age INTEGER,
6      email TEXT,
7      UNIQUE (name, email)
8  );
```

В данном случае уникальность значений устанавлена сразу для двух столбцов - name и email.

NULL и NOT NULL

По умолчанию любой столбец, если он не представляет первичный ключ, может принимать значение **NULL**, то есть фактически отсутствие формального значения. Но если мы хотим запретить подобное поведение и установить, что столбец обязательно должен иметь какое-либо значение, то для него следует установить ограничение **NOT NULL**:

```
1  CREATE TABLE users
2  (
```

```
3       id INTEGER PRIMARY KEY,
4       name TEXT NOT NULL,
5       age INTEGER
6   );
```

В данном случае столбец name не допускает значение NULL.

DEFAULT

Ограничение **DEFAULT** определяет значение по умолчанию для столбца. Если при добавлении данных для столбца не будет предусмотрено значение, то для него будет использоваться значение по умолчанию.

```
1   CREATE TABLE users
2   (
3       id INTEGER PRIMARY KEY,
4       name TEXT,
5       age INTEGER DEFAULT 18
6   );
```

Здесь столбец age в качестве значения по умолчанию имеет число 18.

CHECK

Ограничение **CHECK** задает ограничение для диапазона значений, которые могут храниться в столбце. Для этого после CHECK указывается в скобках условие, которому должен соответствовать столбец или несколько столбцов. Например, возраст пользователей не может быть меньше 0 или больше 100:

```
1   CREATE TABLE users
2   (
3       id INTEGER PRIMARY KEY,
4       name TEXT NOT NULL CHECK(name !=''),
5       age INTEGER NOT NULL CHECK(age >0 AND age < 100)
6   );
```

Кроме проверки возраста здесь также проверяется, что столбец name не может иметь пустую строку в качестве значения (пустая строка не эквивалентна значению NULL).

Для соединения условий используется ключевое слово **AND**. Условия можно задать в виде операций сравнения больше (>), меньше (<), не равно (!=).

Также CHECK можно использовать на уровне таблицы:

```
CREATE TABLE users
(
    id INTEGER PRIMARY KEY,
    name TEXT NOT NULL,
    age INTEGER NOT NULL,
    CHECK ((age >0 AND age < 100) AND (name !=''))
);
```

Оператор CONSTRAINT. Установка имени ограничений

С помощью оператора **CONSTRAINT** можно задать имя для ограничений. Они указываются после ключевого слова CONSTRAINT перед ограничениями на уровне таблицы:

```
CREATE TABLE users
(
    id INTEGER,
    name TEXT NOT NULL,
    email TEXT NOT NULL,
    age INTEGER NOT NULL,
    CONSTRAINT users_pk PRIMARY KEY(id),
    CONSTRAINT user_email_uq UNIQUE(email),
    CONSTRAINT user_age_chk CHECK(age >0 AND age < 100)
);
```

В данном случае ограничение для PRIMARY KEY называется users_pk, для UNIQUE - user_phone_uq, а для CHECK - user_age_chk. Смысл установки имен ограничений заключается в том, что впоследствии через эти имена мы сможем управлять ограничениями - удалять или изменять их.