Lab

Topic: Defining a data structure in SQLite.

Purpose: Create and delete a table. Database attachment. Data types. Column and table restrictions. Foreign keys FOREIGN KEY. Changing tables and columns.

Progress

Creating and deleting a table. Database attachment

The CREATE TABLE command is used to create tables. The general formal syntax of the CREATE TABLE command:

```
1   CREATE TABLE table_name
2   (column_name1 data_type column_attributes1,
3   column_name2 data_type column_attributes2,
4   ..........................................
5   column_nameN data_type column_attributesN,
6   table_attributes
7   )
```

The CREATE TABLE command is followed by the name of the table. The name of the table acts as its identifier in the database, so it must be unique. Also, it must not start with "sqlite_" because table names starting with "sqlite_" are reserved for internal use.

Then, after the table name, the column names, their data types, and their attributes are listed in parentheses. At the end, you can define attributes for the entire table. Column attributes and table attributes are optional.

Let's create the simplest table. Before executing the CREATE TABLE command, regardless of what we use - the sqlite3 console client, the DB Browser for SQLite graphical client or some other client, we first open the database where we want to create a table.

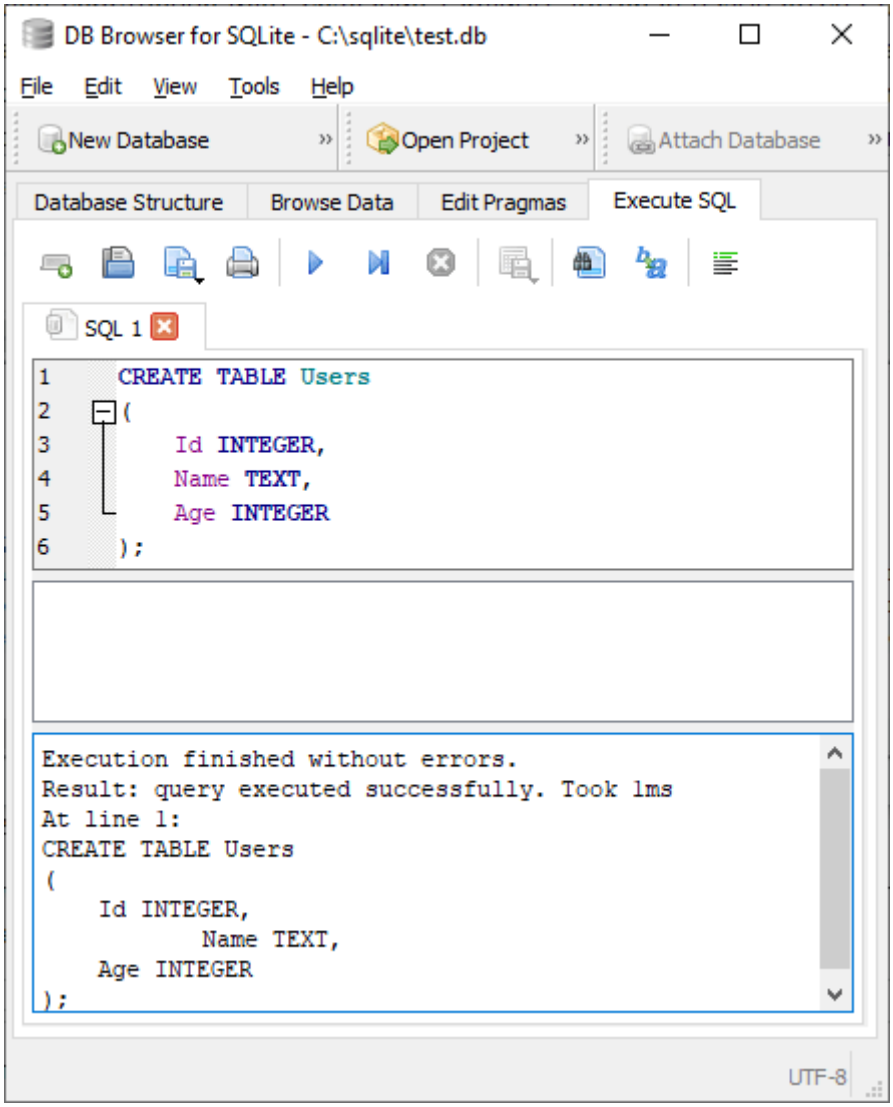
To create a table, execute the following script:

```
1   CREATE TABLE Users
2   (
3   Id INTEGER,
```

```
4    Name TEXT,
5    Age INTEGER
6    );
```

In this case, the table is called "Users". It defines three columns: Id, Age, Name. The first two columns are the user ID and age and are of type INTEGER , meaning they will store numeric values. The "Name" column represents the user's name and is of type TEXT , that is, it represents a string. In this case, a name and a data type are defined for each column, while column and table attributes are generally absent.

And as a result of executing this command, the Users table will be created with three columns.



Creating a table in its absence

If we re-execute a certain sql command above to create the Users table, we will encounter an error - we have already created a table with that name. But there may

be situations when we may not know for sure or not sure whether such a table exists in the database (for example, when we write an application in some programming language and use a database that we did not create). And to avoid the error, using the IF NOT EXISTS expression, we can set the table to be created if it does not exist:

```
CREATE TABLE IF NOT EXISTS Users
(
Id INTEGER,
Name TEXT,
Age INTEGER
);
```

If the Users table does not exist, it will be created. If it is, then no action will be taken and no error will occur.

Database attachment

We can also attach a database and then create a database in it.

The ATTACH DATABASE command is used to attach the database:

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

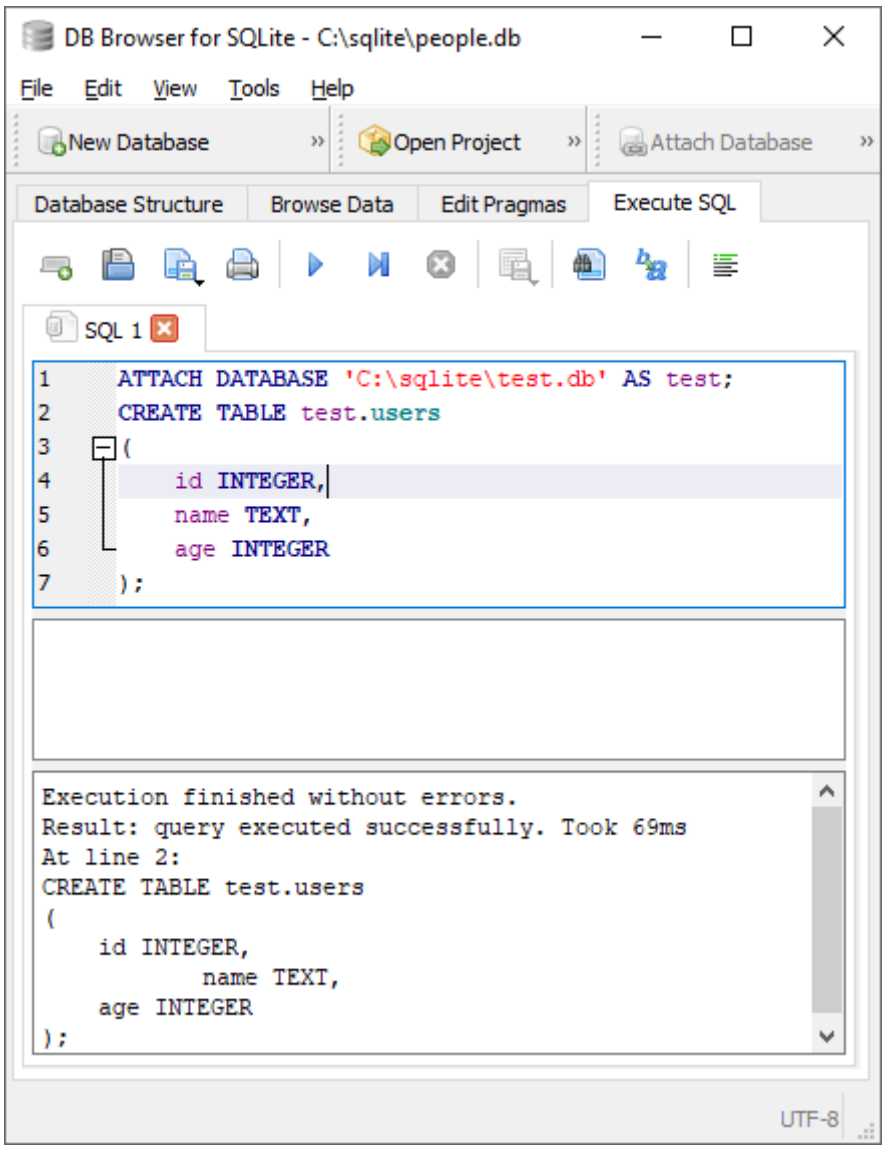
After the ATTACH DATABASE command, the path to the database file is indicated (in this case, it is the path "C:sqlite\test.db"). Then the AS statement is followed by the alias to which the database will be projected. That is, the name "text" will be used in the code to access the database "C:sqlite\test.db". When accessing a table from this database, the alias of the database is first indicated, followed by the name of the table through a dot:

```
db.table
```

For example, let's create a table in the attached database:

```
ATTACH DATABASE 'C:\sqlite\test.db' AS test
CREATE TABLE test.users
(
id INTEGER,
name TEXT,
age INTEGER
);
```

To create the users table in the test.db database, the table name is preceded by an alias: test.users.



And after opening the test.db database, you can see the users table in it.

Deleting tables

To delete a table, use the DROP TABLE command, followed by the name of the table to be deleted. For example, let's delete the users table:

```
1   DROP TABLE users;
```

By analogy with creating a table, if we try to delete a table that does not exist, we will encounter an error. In this case, again, using the IF EXISTS statements, check the presence of the table before deleting:

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

Data types

When defining table columns, they need to specify the data type. Each column has a specific data type. The following types are used to store data in SQLite:

- **NULL**: indicates virtually no value
- **INTEGER**: represents an integer that can be positive or negative and, depending on its value, can occupy 1, 2, 3, 4, 6, or 8 bytes
- **REAL**: represents a floating point number, occupies 8 bytes in memory
- **TEXT**: a single-quoted string of text stored in database encoding (UTF-8, UTF-16BE, or UTF-16LE)
- **BLOB**: binary data

It is worth noting that SQLite operates with the concept of storage classes. And in fact, all these five types are called storage classes. The concept of storage classes is somewhat broader than a data type. For example, the INTEGER class essentially combines 6 different integer data types of different lengths. However, this is more about the inner workings of SQLite. And externally, for example, at the level of defining a table and working with data, we will work with the INTEGER type, and not with all the real types hidden behind this name. Therefore, storage classes are actually associated with a data type. And the five storage classes presented above are also called data types and data can be used when defining columns:

```
1   CREATE TABLE users
2   (
3   id INTEGER,
4   name TEXT,
5   age INTEGER,
6   weight REAL,
7   image BLOB
8   );
```

Alternatively, we can use the NUMERIC identifier. This identifier does not provide a separate data type. And in fact, it represents a column that can store data of all five types listed above (in SQLite NUMERIC terminology, it is also called type affinity). Example:

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

Column and table restrictions

When defining columns and tables, you can set restrictions on them. Constraints allow you to customize the behavior of columns and tables. Column restrictions are indicated after the column type:

```
column_name column_type column_constraints
```

Table constraints are specified after all columns are defined.

Let's consider what column restrictions we can use.

PRIMARY KEY

The PRIMARY KEY attribute defines the primary key of the table. A primary key uniquely identifies a table row. Example:

```
CREATE TABLE users
(
id INTEGER PRIMARY key,
name TEXT,
age INTEGER
);
```

Here the id column acts as the primary key, it will uniquely identify the row and its value must be unique. That is, we cannot have a users table with more than one row, where the id column would have the same value.

Setting the primary key only at the table level:

```
CREATE TABLE users
(
id INTEGER,
name TEXT,
age INTEGER,
PRIMARY KEY(id)
);
```

A primary key can be complex. Such a key is to use several columns at once to uniquely identify a table row. Example:

```
CREATE TABLE users
(
id INTEGER,
name TEXT,
age INTEGER,
PRIMARY KEY(id, name)
```

```
7   );
```

In this case, the connection between the id and name columns acts as the primary key. That is, there cannot be two rows in the users table where both of these fields would have the same values at the same time.

AUTOINCREMENT

The AUTOINCREMENT constraint allows you to specify that the column value will be automatically incremented when a new row is added. This constraint works for columns that represent the INTEGER type with the PRIMARY KEY constraint:

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

In this case, the value of the id column of each newly added row will be incremented by one.

UNIQUE

The UNIQUE constraint specifies that the column can only store unique values.

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

In this case, the email column, which represents the user's phone, can only store unique values. And we won't be able to add two rows to the table that have the values for that column to match.

We can also define this constraint at the table level:

```
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   );
```

In this case, the uniqueness of the values is set for two columns at once - name and email.

NULL and NOT NULL

By default, any column, if it does not represent a primary key, can acquire the value NULL , that is, in fact, the absence of a formal value. But if we want to prohibit such behavior and establish that the column must have some value, then we should set the NOT NULL constraint for it:

```
1   CREATE TABLE users
2   (
3   id INTEGER PRIMARY key,
4   name TEXT NOT NULL,
5   age INTEGER
6   );
```

In this case, the name column cannot be NULL.

DEFAULT

The DEFAULT constraint specifies the default value for the column. If no value is provided for a column when adding data, the default value will be used.

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

Here, the age column defaults to 18.

CHECK

The CHECK constraint defines a limit on the range of values that can be stored in a column. To do this, after CHECK, a condition that must be met by a column or several columns is indicated in parentheses. For example, users' ages cannot be less than 0 or greater than 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   );
```

In addition to the age check, it also checks that the name column cannot have an empty string as a value (an empty string is not equivalent to NULL).

The keyword AND is used to connect conditions. Conditions can be set in the form of comparison operations greater than (>), less than (<), not equal to (! =).

CHECK can also be used at the table level:

```
1   CREATE TABLE users
2   (
3   id INTEGER PRIMARY key,
4   name TEXT NOT NULL,
5   age INTEGER NOT NULL,
6   CHECK ((age >0 AND age < 100) AND (name !=''))
7   );
```

The CONSTRAINT operator. Setting the name of the restrictions

You can set a name for the constraints using the CONSTRAINT statement. They are specified after the CONSTRAINT keyword before table-level constraints:

```
1    CREATE TABLE users
2    (
3    id INTEGER,
4    name TEXT NOT NULL,
5    email TEXT NOT NULL,
6    age INTEGER NOT NULL,
7    CONSTRAINT users_pk PRIMARY KEY(id),
8    CONSTRAINT user_email_uq UNIQUE(email),
9    CONSTRAINT user_age_chk CHECK(age >0 AND age < 100)
10   );
```

In the case of a constraint, PRIMARY KEY is called users_pk, UNIQUE is called user_phone_uq, and CHECK is called user_age_chk. The point of setting the names of the restrictions is that later through these names we will be able to manage the restrictions - remove or change them.

Foreign keys FOREIGN KEY

Foreign keys allow you to establish a relationship between tables. A foreign key is set to columns from a dependent, child table, and points to one of the columns from

the parent table. Typically, a foreign key points to a primary key from the associated parent table.

The general syntax for setting a table-level foreign key is:

```
[CONSTRAINT constraint_name]
FOREIGN KEY (column1, column2, ... columnN)
REFERENCES parent_table (parent_table_column1, parent_table_column2, ... parent_table_columnN)
[ON DELETE actions]
[ON UPDATE actions]
```

To create a foreign key constraint, FOREIGN KEY is followed by the table column that represents the foreign key. And after the REFERENCES keyword, the name of the related table is indicated, and then, in parentheses, the name of the related column to which the foreign key will point. The REFERENCES expression is followed by the ON DELETE and ON UPDATE expressions, which set the action when deleting and updating a row from the main table, respectively.

For example, let's define two tables and connect them using a foreign key:

```
CREATE TABLE companies
(
id INTEGER PRIMARY KEY AUTOINCREMENT,
name TEXT NOT NULL
);
CREATE TABLE users
(
id INTEGER PRIMARY KEY AUTOINCREMENT,
name TEXT NOT NULL,
age INTEGER,
company_id INTEGER,
FOREIGN KEY (company_id) REFERENCES companies (id)
);
```

In this case, tables of companies and users are defined. companies is the main one and represents the companies where the user can work. users is dependent and represents users. The users table is related to the companies table and its id column through the company_id column. That is, the company_id column is a foreign key that points to the id column from the companies table.

The CONSTRAINT statement can be used to set a name for a foreign key constraint:

```
CREATE TABLE users
```

```
2   (
3   id INTEGER PRIMARY KEY AUTOINCREMENT,
4   name TEXT NOT NULL,
5   age INTEGER,
6   company_id INTEGER,
7   CONSTRAINT users_companies_fk
8   FOREIGN KEY (company_id) REFERENCES companies (id)
9   );
```

ON DELETE and ON UPDATE

With the help of ON DELETE and ON UPDATE expressions, you can set actions that are performed, respectively, when deleting and changing a related row from the main table. The following options can be used as an action:

- **CASCADE**: Automatically deletes or changes rows from a dependent table when related rows in the parent table are deleted or changed.
- **SET NULL : when deleting or updating a related row from the main table, sets the value to NULL**for the foreign key column . (In this case, the foreign key column must support setting NULL)
- **RESTRICT**: rejects the deletion or modification of rows in the parent table if there are related rows in the dependent table.
- **NO ACTION**: same as RESTRICT.
- **SET DEFAULT**: when deleting a related row from the main table, sets the foreign key column to the default value specified using the DEFAULT attribute.

Cascading delete

Cascading delete allows when deleting a row from the main table to automatically delete all related rows from the dependent table. For this, the CASCADE option is used:

```
1   CREATE TABLE users
2   (
3   id INTEGER PRIMARY KEY AUTOINCREMENT,
4   name TEXT NOT NULL,
5   age INTEGER,
6   company_id INTEGER,
7   FOREIGN KEY (company_id) REFERENCES companies (id) ON DELETE CASCADE
```

```
8   );
```

The expression ON UPDATE CASCADE works in a similar way. When the primary key is changed, the value of the foreign key associated with it will automatically change. However, since primary keys change very rarely, and in principle it is not recommended to use columns with variable values as primary keys, the ON UPDATE expression is rarely used in practice.

Setting to NULL

If the SET NULL option is set for a foreign key, the foreign key column must be NULL:

```
1   CREATE TABLE users
2   (
3   id INTEGER PRIMARY KEY AUTOINCREMENT,
4   name TEXT NOT NULL,
5   age INTEGER,
6   company_id INTEGER,
7   FOREIGN KEY (company_id) REFERENCES companies (id) ON DELETE SET NULL
8   );
```

Changing tables and columns

If the table has already been created and needs to be changed, the ALTER TABLE command is used for this. It supports various options and capabilities. Let's consider only the main scenarios that we can face.

Renaming the table

To rename the table, the RENAME TO statement is used, after which the new table name is specified:

```
1   ALTER TABLE users
2   RENAME TO people;
```

Here the user table is renamed to "people".

Adding a new column

Let's add a new email column to the users table:

```
1   ALTER TABLE users
2   ADD COLUMN email TEXT NOT NULL;
```

In this case, the email column is of type TEXT and the NOT NULL constraint is defined for it.

Renaming a column

Rename the email column to login

```
1   ALTER TABLE users
2   RENAME COLUMN email TO login;
```

Delete a column

Let's delete the login column from the users table:

```
1   ALTER TABLE users
2   DROP COLUMN login;
```