Creating and dropping tables

In this part of the SQLite tutorial, we will cover the data definition language (DDL) of the SQLite database. The *DDL* consists of SQL statements that define the database schema. The *schema* is the database structure described in a formal language. In relational databases, the schema defines the tables, views, indexes, relationships, or triggers.

The SQLite supports the following DDL statements:

- CREATE
- DROP

In SQLite, the `CREATE` statement is used to create tables, indexes, views, and triggers. The `DROP` statement removes tables, indexes, views, or triggers.

Creating tables

The `CREATE` statement is used to create tables. It is also used to create indexes, views, and triggers.

To create a table, we give a name to a table and to its columns. Each column can have one of these data types:

- NULL — The value is a NULL value
- INTEGER — a signed integer
- REAL — a floating point value
- TEXT — a text string
- BLOB — a blob of data

```
sqlite> CREATE TABLE Testing(Id INTEGER);
sqlite> .schema Testing
CREATE TABLE Testing(Id INTEGER);
```

We create a simple `Testing` table with the `CREATE TABLE` statement. The `.schema` command shows the formal definition of the table.

```
sqlite> CREATE TABLE Testing(Id INTEGER);
Error: table Testing already exists
```

If we try to create a table that already exists, we get an error. Therefore the `CREATE TABLE` statement has an optional `IF NOT EXISTS` clause. With this clause nothing is done and we receive no error.

```
sqlite> CREATE TABLE IF NOT EXISTS Testing(Id INTEGER);
```

We get no error message for trying to create a table that already exists.

Dropping tables

The `DROP` statement is used to delete a table from a database.

```
sqlite> .tables
```

```
Cars     Friends  Testing
sqlite> DROP TABLE Testing;
sqlite> .tables
Cars     Friends
```

We show the available tables with the `.tables` command. The `DROP TABLE` statement removes the `Testing` table from the database.

```
sqlite> DROP TABLE Testing;
Error: no such table: Testing
```

Trying to drop a table that does not exist leads to an error. With the `IF EXISTS` clause we can avoid this error.

```
sqlite> DROP TABLE IF EXISTS Testing;
```

This statement will drop the `Testing` table only if it exists.

Constraints in SQLite

In this part of the tutorial, we will work with constraints. *Constraints* are placed on columns. They limit the data that can be inserted into tables. In SQLite, we have the following constraints:

- NOT NULL
- UNIQUE
- PRIMARY KEY
- FOREIGN KEY
- CHECK
- DEFAULT

NOT NULL constraint

A column with a `NOT NULL` constraint cannot have `NULL` values.

```
sqlite> CREATE TABLE People(Id INTEGER, LastName TEXT NOT NULL,
   ...> FirstName TEXT NOT NULL, City TEXT);
```

We create two columns with `NOT NULL` constraints.

```
sqlite> INSERT INTO People VALUES(1, 'Hanks', 'Robert', 'New York');
sqlite> INSERT INTO People VALUES(2, NULL, 'Marianne', 'Chicago');
Error: People.LastName may not be NULL
```

The first `INSERT` statement succeeds, while the second fails. The error says that the `LastName` column may not be `NULL`.

Primary key constraint

The `PRIMARY KEY` constraint uniquely identifies each record in a database table. There can be more `UNIQUE` columns, but only one primary key in a table. Primary keys are important when designing database tables. Primary keys are unique IDs. We use them to refer to table rows. Primary keys become foreign keys in other tables when creating relations among tables. Due to to a 'longstanding coding oversight', primary keys can be `NULL` in SQLite. This is not the case with other databases.

```
sqlite> DROP TABLE Brands;
sqlite> CREATE TABLE Brands(Id INTEGER PRIMARY KEY, BrandName TEXT);
```

The `Id` column of the `Brands` table becomes a `PRIMARY KEY`.

```
sqlite> INSERT INTO Brands(BrandName) VALUES('Coca Cola');
sqlite> INSERT INTO Brands(BrandName) VALUES('Pepsi');
sqlite> INSERT INTO Brands(BrandName) VALUES('Sun');
sqlite> INSERT INTO Brands(BrandName) VALUES('Oracle');
sqlite> SELECT * FROM Brands;
Id          BrandName
----------  ----------
1           Coca Cola
2           Pepsi
3           Sun
4           Oracle
```

In SQLite if a column is `INTEGER` and `PRIMARY KEY`, it is also auto-incremented.

Foreign key constraint

A `FOREIGN KEY` in one table points to a `PRIMARY KEY` in another table. It is a referential constraint between two tables. The foreign key identifies a column or a set of columns in one (referencing) table that refers to a column or set of columns in another (referenced) table.

The SQLite documentation calls the referenced table the parent table and the referencing table the child table. The parent key is the column or set of columns in the parent table that the foreign key constraint refers to. This is normally, but not always, the primary key of the parent table. The child key is the column or set of columns in the child table that are constrained by the foreign key constraint and which hold the `REFERENCES` clause.

We demonstrate this constraint using two tables: `Authors` and `Books`.

```
CREATE TABLE Authors(AuthorId INTEGER PRIMARY KEY, Name TEXT);
CREATE TABLE Books(BookId INTEGER PRIMARY KEY, Title TEXT, AuthorId INTEGER,
    FOREIGN KEY(AuthorId) REFERENCES Authors(AuthorId));
```

```
sqlite> DELETE FROM Authors WHERE AuthorId=1;
Error: foreign key constraint failed
```

Inserting, updating, and deleting in SQLite

In this part of the SQLite tutorial, we will insert, update and delete data from SQLite tables. We will use the `INSERT`, `DELETE`, and `UPDATE` statements. These statements are part of the SQL Data Manipulation Language, DML.

Inserting data

The `INSERT` statement is used to insert data into tables. We will create a new table in which to execute our examples.

```
sqlite> DROP TABLE IF EXISTS Cars;
sqlite> CREATE TABLE Cars(Id INTEGER PRIMARY KEY, Name TEXT,
   ...> Price INTEGER DEFAULT 'Not available');
```

We create a new table `Cars` with `Id`, `Name`, and `Price` columns.

```
sqlite> INSERT INTO Cars(Id, Name, Price) VALUES(1, 'Audi', 52642);
```

This is the classic `INSERT` statement. We have specified all column names after the table name and all values after the `VALUES` keyword. The first row is added into the table.

```
sqlite> INSERT INTO Cars(Name, Price) VALUES('Mercedes', 57127);
```

We add a new car into the `Cars` table. We have omitted the `Id` column. The `Id` column is defined as `INTEGER PRIMARY KEY`. Such columns are auto-incremented in SQLite. This means the SQLite library will add a new `Id` itself.

```
sqlite> SELECT * FROM Cars;
Id                Name                Price
---------------   ------------------  ----------
1                 Audi                52642
2                 Mercedes            57127
```

Here is what we have in the `Cars` table at the moment.

```
sqlite> INSERT INTO Cars VALUES(3, 'Skoda', 9000);
```

In this SQL statement, we did not specify any column names after the table name. In such a case, we have to supply all values.

Deleting data

The `DELETE` keyword is used to delete data from tables. First, we are going to delete one row from a table. We will use the `Cars2` table which we have created previously.

```
sqlite> DELETE FROM Cars2 WHERE Id=1;
```
We delete a row with `Id` 1.

```
sqlite> DELETE FROM Cars2;
```
This SQL statement deletes all data in the table.

Updating data

The `UPDATE` statement is used to modify a subset of the values stored in zero or more rows of a database table.

Say we wanted to change 'Skoda' to 'Skoda Octavia' in our `Cars` table. The following statement shows how to accomplish this:

```
sqlite> UPDATE Cars SET Name='Skoda Octavia' WHERE Id=3;
```

Quelle: http://zetcode.com/db/sqlite/