Sqlite3

Sqlite3 Overview

What is SQLite3?

SQLite3 is a lightweight, serverless, self-contained database engine that is widely used for small to medium-sized applications, and exist from the start in a lot of languages.

Key Features of SQLite3

- **Serverless**: No need to install or manage a separate database server.
- **Lightweight**: Simple and fast, with minimal resource consumption.
- **Self-contained**: All database data is stored in a single file.
- **Cross-platform**: Compatible with most operating systems.
- **Widely Used**: Common in mobile applications (e.g., Android and iOS) and embedded systems.

When to Use SQLite3

- Applications with moderate amounts of data.
- Projects requiring minimal setup or server management.
- Rapid prototyping and testing.
- Local databases for desktop or mobile applications.

Introducing DB Browser for SQLite

What is DB Browser for SQLite?

DB Browser for SQLite is a graphical user interface (GUI) tool that allows users to interact with SQLite databases without writing SQL queries.

Key Features

- **Database Management**: Create, open, and save SQLite database files.
- **Query Execution**: Run SQL queries to interact with the database.
- **Data Viewing and Editing**: View and edit table contents directly.
- **Export/Import**: Import and export data in CSV and other formats.
- **Visual Design**: Create and modify tables, indexes, and relationships visually.

Setting Up DB Browser

1. **Download and Install**:
    - Visit DB Browser for SQLite.
    - Download the appropriate version for your operating system.
    - Install the software.
2. **Basic Navigation**:
    - **Open Database**: Open an existing SQLite database or create a new one.
    - **Browse Data**: View and edit table content.
    - **Execute SQL**: Write and execute SQL queries.
    - **Design Tables**: Use the visual interface to create or modify tables.

Sqlite3 commands:

Data Types in SQLite3

SQLite uses a **dynamic typing system** that is flexible but behaves differently compared to strict typing in databases like MySQL or PostgreSQL.

SQLite columns can store any type of data, but they are **type-affinity aware**, meaning each column has a preferred data type.

Column Affinity	**Accepted Types**
TEXT	Stores values as text, even if you insert numbers.
NUMERIC	Tries to store values as numbers but can store text if needed.
INTEGER	Stores values as integers.
REAL	Stores values as floating-point numbers.
BLOB	Stores data exactly as it is without type conversion.
NONE	No type preference; stores data in the format it is provided.

Creating table:

```
CREATE TABLE IF NOT EXISTS students (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    name TEXT NOT NULL,
    age INTEGER,
    grade TEXT
);
```

Inserting data:

```
INSERT INTO students (name, age, grade) VALUES ("Dor", 12, "A")
```

Or multiple:

```
INSERT INTO students (name, age, grade) VALUES ('Moshe', 42, 'C'), ('Dor',
12, 'A')(......);
```

Select data:

```
SELECT * FROM users
```

Update data:

```
UPDATE users SET age = ? WHERE username = ?
```

Delete data:

```
DELETE FROM users
```

Using js:

Connecting:

```
const sqlite3 = require("sqlite3");

const db = new sqlite3.Database(":memory:");
const db = new sqlite3.Database("./db/plainSqlite.db");
```

Create a table:

```
const createTable = () => {
  const createTableSQL = `
    CREATE TABLE IF NOT EXISTS users (
      id INTEGER PRIMARY KEY,
      username TEXT NOT NULL,
      birthday TEXT,
      age INTEGER
    );
  `;
  db2.run(createTableSQL, (err) => {
    if (err) {
      console.error("Error creating table:", err.message);
    } else {
      console.log("Table 'users' created successfully.");
    }
  });
};
```

Insert data:

```
const insertUser = (username, birthday, age) => {
const insertSQL = `INSERT INTO users (username, birthday, age)
VALUES (?, ?, ?)`;
db2.run(insertSQL, [username, birthday, age], function (err) {
    if (err) {
      return console.error(err.message);
    }
    console.log(`User added with ID: ${this.lastID}`);
  });
};
```

Select all users:

```
const selectUsers = () => {
  const selectSQL = `SELECT * FROM users`;
  db2.all(selectSQL, [], (err, rows) => {
    if (err) {
      throw err;
    }
    console.log("All Users:");
    rows.forEach((row) => {
      console.log(row);
    });
  });
};
```

Update a user's age by username:

```
const updateUserAge = (username, newAge) => {
  const updateSQL = `UPDATE users SET age = ? WHERE username = ?`;
  db2.run(updateSQL, [newAge, username], function (err) {
    if (err) {
      return console.error(err.message);
    }
    console.log(`Row(s) updated: ${this.changes}`);
  });
};
```

Delete a user by username:

```
const deleteUser = (username) => {
  const deleteSQL = `DELETE FROM users WHERE username = ?`;
  db2.run(deleteSQL, username, function (err) {
    if (err) {
      return console.error(err.message);
    }
    console.log(`User deleted: ${this.changes}`);
  });
};
```

Summary of sqlite3 with js:

- db.run(sql, params, callback)
  - Executes an SQL query that does not return data (`INSERT`, `UPDATE`, `DELETE`).
- db.get(sql, params, callback)
  - Retrieves a single row from the database.
- db.all(sql, params, callback)
  - Retrieves all rows that match the query.
- db.each(sql, params, callback, completeCallback)
  - Iterates over each row in the result set and calls a callback for each row.
- db.prepare(sql)
  - Prepares an SQL statement for execution, useful for executing the same query multiple times with different parameters.

```
const stmt = db.prepare("INSERT INTO students (name, age, grade)
VALUES (?, ?, ?)");
stmt.run("Moshe", 42, "C");
stmt.run("Dor", 12, "A");
stmt.finalize();
```

- db.close(callback)
  - Closes the database connection.