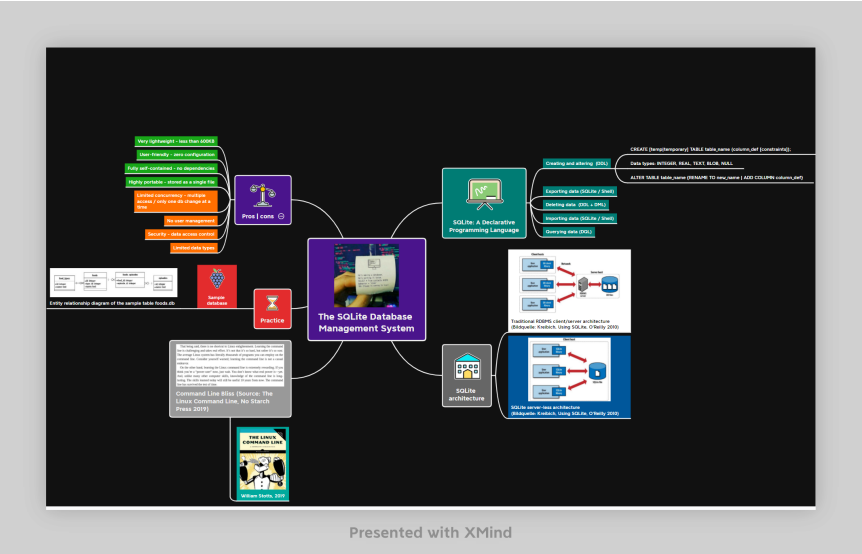
README

Open tinyurl.com/sqlite-codealong-org to code along.

- SQLite Overview / Pros and Cons
- Explore the SQLite shell
- Creating a database (DDL)
- Creating a table (DDL)
- Inserting values into a table (DML)
- Querying a table (DQL)
- Formatting stdout (DCL)
- Changing table content (DML)
- Change table structure (DDL)
- Viewing the table index (DQL)
- Viewing the table schema (DQL)
- Viewing table views (DQL)
- Viewing the master table (DQL)
- Exporting data (DML)
- Deleting a table (DDL)
- Deleting a database (DDL)
- Importing data (DML)
- Writing CSV files (DML)
- Interfacing with the shell (DCL)

SQLite overview



Single-file approach (instead of client/server)

SQLite architecture vs. RDBMS client/server architecture:

1. No management system between database files and user apps
2. Instead just one file directly connected to the user app

Declarative programming language

SQLite is a declarative programming language. Its properties:

1. DDL: creating and altering tables

   ```
   CREATE [temp] TABLE table_name (col_def [constraints]);
   ALTER TABLE table_name (RENAME TO new_name|ADD COLUMN col_def);
   ```

   Data types available: `REAL`, `TEXT`, `BLOB NULL`

2. Exporting data (SQLite <-> shell)
3. Deleting data (DDL for tables + DML for table content)

4. Importing data (SQLite <-> shell)

5. Querying data (DQL)

6. Console (`sqlite3`) control (DCL)

7. You can use graphical development tools but SQLite favors the command line. It has a small set of flags and options:

```
sqlite3 --help

Usage: sqlite3 [OPTIONS] FILENAME [SQL]
FILENAME is the name of an SQLite database. A new database is created
if the file does not previously exist.
OPTIONS include:
   -A ARGS...           run ".archive ARGS" and exit
   -append              append the database to the end of the file
   -ascii               set output mode to 'ascii'
   -bail                stop after hitting an error
   -batch               force batch I/O
   -box                 set output mode to 'box'
   -column              set output mode to 'column'
   -cmd COMMAND         run "COMMAND" before reading stdin
   -csv                 set output mode to 'csv'
   -deserialize         open the database using sqlite3_deserialize()
   -echo                print commands before execution
   -init FILENAME       read/process named file
   -[no]header          turn headers on or off
   -help                show this message
   -html                set output mode to HTML
   -interactive         force interactive I/O
   -json                set output mode to 'json'
   -line                set output mode to 'line'
   -list                set output mode to 'list'
   -lookaside SIZE N    use N entries of SZ bytes for lookaside memory
   -markdown            set output mode to 'markdown'
   -maxsize N           maximum size for a --deserialize database
   -memtrace            trace all memory allocations and deallocations
   -mmap N              default mmap size set to N
   -newline SEP         set output row separator. Default: '\n'
   -nofollow            refuse to open symbolic links to database files
```

```
-nonce STRING         set the safe-mode escape nonce
-nullvalue TEXT       set text string for NULL values. Default ''
-pagecache SIZE N     use N slots of SZ bytes each for page cache memory
-quote                set output mode to 'quote'
-readonly             open the database read-only
-safe                 enable safe-mode
-separator SEP        set output column separator. Default: '|'
-stats                print memory stats before each finalize
-table                set output mode to 'table'
-tabs                 set output mode to 'tabs'
-version              show SQLite version
-vfs NAME             use NAME as the default VFS
-zip                  open the file as a ZIP Archive
```

Pros and cons

Pros	Cons
Very lightweight $<$ 600KB	Limited concurrency (1 access)
user friendly - zero config	no user management
self-contained - no dependencies	security: no data access control
highly portable - stored in 1 file	limited data types

Explore the `sqlite3` shell/console/terminal (DQL)

- The learning of a programming language begins with being to get answers first on the syntax, and then on the logic of the program
- Get all help on the console from the OS shell: open an shell inside Emacs with `M-x eshell` and run this command:

  ```
  sqlite3 --help
  ```

- You can now open SQLite on a new (or existing) database while directly specifying that it displays output in tables with columns and header.
- Try that with the `foods.sqlite` database that you created last time.

  ```
  sqlite3 -header -column foods.sqlite
  ```

- On the console, check that the initialization worked with `.show`

- If you don't have a database, you still run the command and check the settings - the database will now be created.
- Get the help on the console for in-program control commands:

  ```
  .help
  ```

- How can I find out where I am? (Substitute `pwd` for `DIR` when you're on Windows)

  ```
  .shell pwd
  ```

Creating a database (DDL)

- Create a new file `sqlite.org` to code along. Create a headline "Creating a database DDL".
- Use `<sqlite` if you defined it in `org-structure-template-alist` to create a new codeblock. Check which database sqlite3 writes to, and if it has any tables in it:

  ```
  .database
  .tables
  ```

Creating a table with `CREATE TABLE` (DDL)

- The general structure of the command:

  ```
  CREATE [temp|temporary] TABLE table_name (col_def [constraints]);
  ```

- `temporary` tables can be useful for querying but they disappear when the session ends - they are transient, not permanent.
- Let's create a simple table `test` with an `INTEGER` field `id` that is a `PRIMARY KEY`, and a `TEXT` field called `value` - check that it was created:

  ```
  CREATE TABLE test (id INTEGER PRIMARY KEY, value TEXT);
  .tables
  ```

- There is a number of `constraints` to ensure data integrity.

SQL Table Constraints

When creating a table in SQL, various constraints can be specified to enforce data integrity and rules within the table. These constraints are:

PRIMARY KEY

- Ensures unique values across the table and cannot contain NULL values.
- Uniquely identifies each row in a table.

```
CREATE TABLE example (
  id INT PRIMARY KEY,
  name TEXT
);
```

- For a composite primary key:

```
CREATE TABLE example (
id1 INT,
id2 INT,
name TEXT,
PRIMARY KEY (id1, id2)
);
```

FOREIGN KEY

- Establishes a relationship between the key columns of two tables.
- Ensures that the value in the child table matches one of the values in the parent table's primary key or a unique key.

```
CREATE TABLE orders (
  order_id INT PRIMARY KEY,
  product_id INT,
  FOREIGN KEY (product_id) REFERENCES products(product_id)
);
```

UNIQUE

- Ensures that all values in a column are unique.
- Multiple rows can have NULL values unless the column is explicitly set to NOT NULL.

```
CREATE TABLE example (
  id INT PRIMARY KEY,
  email TEXT UNIQUE
);
```

CHECK

- Specifies a condition that must be true for all rows in the table.
- Used to enforce domain integrity by limiting the values that can be stored in a column.

```
CREATE TABLE example (
id INT PRIMARY KEY,
age INT CHECK (age >= 18)
);
```

NOT NULL

- Ensures that a column cannot have a NULL value.
- Enforces that a column must always have a data value.

```
CREATE TABLE example (
id INT PRIMARY KEY,
name TEXT NOT NULL
);
```

DEFAULT

- Assigns a default value to a column when no value is specified.
- If a row is inserted without a value for this column, the column will take the default value.

```
CREATE TABLE example (
id INT PRIMARY KEY,
name TEXT,
status TEXT DEFAULT 'active'
);
```

These constraints play a crucial role in maintaining data integrity, ensuring consistency, and enforcing database rules.

Inserting values into a table with `INSERT` (DML)

- Since we're returning to this section,
  1. delete the table that you already have in `test.db`
  2. `CREATE` a new table `test` that allows the `INSERT` of an `INTEGER =PRIMARY KEY` field `id` and a `TEXT` field `value`.
- Solution:

```
--CREATE TABLE test (id INT);
.tables
--This command handles an exception
DROP TABLE IF EXISTS test; -- delete table ‘test‘
.tables
CREATE TABLE test
(id INTEGER PRIMARY KEY,
value TEXT);
.schema
```

- Enter three records in your table:

```
INSERT INTO test (value) VALUES ('Ms. Jane Robinson');
INSERT INTO test (value) VALUES ('Mme. Carl Robinson');
INSERT INTO test (value) VALUES ('Mr. Edward Jones');
```

Querying a table (DQL)

- We can look at the entire table (show `header` and use `column` mode):

```
SELECT * FROM test;
```

- Notice that despite the code block header you can still change the options inside the code block:

```
.header off
SELECT * FROM test LIMIT 2;
```

- You notice that constraining `id` as `PRIMARY KEY` included `AUTO INCREMENT` - the value is automatically increased by one for each new row.

- If you used `AUTO INCREMENT`, you can get the value of the last non-empty row with an SQL function:

```
DROP TABLE IF EXISTS foo;
-- create a temporary table
CREATE TEMP TABLE foo (id INT AUTO INCREMENT, pray TEXT);
-- insert two records
INSERT INTO foo (pray) VALUES ("In Nomine Patri");
INSERT INTO foo (pray) VALUES ("Et Spiritui Sancto");
SELECT LAST_INSERT_ROWID();
```

Changing table content (DML)

- The second row contains a mistake: it should be 'Mr.' instead of 'Mme.' (which is French for 'Mrs.'):

```
SELECT * FROM test;
```

- To change this, we can use `UPDATE`, which uses a *row filter*:

```
UPDATE test
SET value = 'Mr. Carl Robinson'
WHERE id = 2;
SELECT  * FROM test;
```

- What do you think would happen if you'd forget the `WHERE` clause?

```
CREATE TEMP TABLE test1 AS
SELECT * FROM test; -- makes a copy
SELECT * FROM test1;
UPDATE test1
SET value = 'Mr. Karl Robinson'; -- missing the WHERE filter
SELECT * FROM test1;
```

Change table structure (DDL)

- SQLite is more limited to making schema changes than other SQL flavors,because of its architecture:
  1. You can rename tables
  2. You can add columns to an existing table
  3. You can drop existing columns from an existing table
  4. You can NOT delete, change or rename columns
- The DDL command is `ALTER TABLE`:

  ```
  ALTER TABLE tbl_name {REN|AME TO new_name | ADD COLUMN colDef}
  ```

- We rename the table `test` to `test_new`:

  ```
  .tables
  ALTER TABLE test RENAME TO test_new;
  .tables
  ALTER TABLE test_new RENAME TO test;
  .tables
  ```

- We can also add a column:

  ```
  ALTER TABLE test ADD COLUMN sex TEXT;
  SELECT * FROM test;
  ```

- Delete the new column again with the `DROP COLUMN` clause, and then review the database structure:

  ```
  ALTER TABLE test DROP COLUMN sex;
  .schema
  ```

Viewing the table index (DDL)

- An index is a keyword index. Creating an index speeds up DQL commands and slows down DML commands.
- The following command creates an index for the only non-trivial column of the table `test`:

```
CREATE INDEX test_idx ON test (value);
```

- The console command `.indices` lists the defined indices, and the `.schema` command shows that the db architecture has changed:

```
.indices test
.schema
```

- We will prefer graphical (ERD) descriptions of the database architecture as soon as we have more than a handful of tables.

Viewing table views (DQL)

- Let's rekindle our knowledge of stored queries or views: for example a view that contains only the `value` column. And let's check the db size before the transaction:

```
.shell ls -l test.db
```

- Create the view:

```
CREATE VIEW value_view AS SELECT value FROM test;
.tables
.schema
```

- As you can see, `value_view` is listed by `.tables`. If you check the size, you'll see that it has not visibly changed:

```
.shell ls -l test.db
```

- You can delete views like tables with `DROP VIEW`.

** STARTING OVER FROM HERE **

- This command will not work in Emacs because you're trying to remove the database that you're currently writing to!

```
.shell rm ./test.db    /* on Windows, may have to use DEL */
```

- Unfortunately, you'll have to remove the old database `test.db` manually:
  1. Either by opening a shell (`M-x eshell`) and entering `rm test.db` or `DEL test.db` on Windoze.
  2. Or by opening Emacs' `Dired` with `C-x d` and removing the file with `d x` (cursor on the file).
- Run the following code block on `test.db` to have a working database with an index and a view in it so that you can continue from here.

```
/* starting over with table creation */
CREATE TABLE IF NOT EXISTS test (id INTEGER PRIMARY KEY, value TEXT);

/* insert records */
INSERT INTO test (value) VALUES ('Ms. Jane Robinson');
INSERT INTO test (value) VALUES ('Mme. Carl Robinson');
INSERT INTO test (value) VALUES ('Mr. Edward Jones');

/* correct record */
UPDATE test SET VALUE='Mr. Carl Robinson' WHERE id=2;

/* create table index for `value` column */
CREATE INDEX test_idx ON test (value);

/* create stored query view */
CREATE VIEW value_view AS SELECT value from test;

/* check tables and db schema */
.tables
.schema
```

- Check content of `test` table:

```
SELECT * FROM test;
```

Formatted printing (DQL)

- SQLite has its own `printf` function, which you know from C:

```
.header off
SELECT printf("The database contains %s", value) FROM test;
```

- On Windoze, you need single quotes.
- Sometimes, the SQLite version has unexpected effects.

```
.version
```

Viewing the master table (DQL)

- All the DDL commands are entered in a master table, which is named `sqlite_master` whose schema you can inspect:

```
.schema sqlite_master
```

- When you display the table's contents, you see the different layers of abstraction that we have generated so far:

```
SELECT type, name, tbl_name, sql
FROM sqlite_master;
```

Exporting data (DML)

- You've seen the `.dump` console command before. Without additional options, the whole db will be dumped.
- Let's look at the options using `.help`:

```
.help .dump
```

- Let's test this with the content of the `test` table - notice that the `--` is part of a flag, and not the SQL comment sign:
  1. redirect `.output` to `test_data.sql`
  2. `.dump --data-only`
  3. redirect `.output` to `stdout`
  4. view the file on the OS shell with `cat`

```
.output test_data.sql
.dump --data-only
.output stdout
.shell cat test_data.sql  /* on Windoze, try TYPE instead of ‘cat‘ */
```

- Compare this with the complete database dump: repeat all commands but without the `--data-only` option, and dump into `test_all.sql`:

```
.output test_all.sql
.dump
.output stdout
.shell cat test_all.sql /* On Windoze replace ‘cat‘ by TYPE */
```

- You can also use `.mode` to only extract query content. Check out the `.help` first:

```
.help mode
```

- Write only the third row of `test` into another file, called `test_row.sql` using `.mode insert`:

```
.mode insert
.output test_row.sql
SELECT * FROM test WHERE id=3; -- 3rd row only
.output stdout
.shell cat test_row.sql  /* On Windoze, use TYPE instead of ‘cat‘ */
```

- You can now use the file `test_row.sql` to import data into a table called `'table'`:

```
CREATE TABLE IF NOT EXISTS 'table' (id INTEGER, value TEXT);
.tables
.read test_row.sql
SELECT * FROM 'table';
```

Deleting a table and a view (DDL)

- To delete a table and a view, use `DROP TABLE` and handle the exceptions:

```
DROP TABLE IF EXISTS 'table';
DROP VIEW IF EXISTS value_view;
.tables
```

- There is no `DROP DATABASE` command in SQLite (but there is in all other SQL flavors).

Deleting a database (DDL)

- There is no 'drop database' command in SQLite (unlike in other SQL flavors). Instead you just delete the database file:

```
.database
.tables
.shell ls -l test.db /* on Windoze, use DIR instead of `ls -l` */
.shell rm test.db /* On Windoze, use DEL instead of `rm` */
.shell ls -l test.db
.tables
```

Importing data (DML)

- You can import data to SQLite in two different ways:
  1. if the data are in an SQLite file (`.sql`) you can `.read` them.
  2. if the data are in a CSV file (`.csv`) you can `.import` them.
- Earlier, you saved the table to `test_all.sql`. Check out that this is indeed an SQL file from the console:

```
.shell head -n 5 test_all.sql  /* On Windoze try TYPE instead of `head -n 5` */
```

- If you don't have it for one reason or another, you can upload it from `tinyurl.com/test-sql`.
- Read the table into `test.db` using `.read`:

```
.tables
.read test_all.sql
.tables
SELECT * FROM test;
```

- If you try to re-import the data, the `UNIQUE` constraint will fail - do you know why?
- Upload the CSV test file from `tinyurl.com/test-all-csv` to your current working directory (where this file is).

```
test        value_view
```

- Check out which `.separator` SQLite is currently working with, and if it's not "," then change it to ",". Then `.import` the file to another table, `test_csv`:

```
.show
.sep ","
.show
/* 1st file = source, 2nd file = target */
.import test_csv.csv test_csv
.tables
SELECT * FROM test_csv;
```

Writing CSV files (DML)

- Using `.output`, you can also write CSV data. Write the data from the table `test` to a file `test_all.csv` using `.mode csv`

```
.mode csv
.output test_all.csv
SELECT * FROM test;  -- this will now be written to CSV
.output stdout
.shell cat test_all.csv
```

Interfacing with the shell (DCL)

- Almost all console commands can also be called from the command line interface (CLI). For example to `.dump` the entire database:

```
sqlite3 test.db .dump
```

- Notice that this last code block executes `bash` (the shell program), and not SQLite. If you don't have `bash` on your computer you can either install it via Cygwin or MSYS2, or you can run the `sqlite3` command on a separate CLI (Windows: CMD, MacOS: terminal).

- Or to `SELECT` the records:

  ```
  sqlite3 -header -column test.db "SELECT * FROM test"
  ```

- You didn't need a delimiter here, and you can add other commands, too:

  ```
  sqlite3 -header -column test.db "SELECT * FROM test" .schema
  ```

- The CLI in Linux supports redirection (the Windows shell does not):

  ```
  sqlite3 test2.db < test_all.sql   # redirect stdin to database
  sqlite3 -csv test2.db "SELECT * FROM test"  # output test as CSV
  ```

- You can also initialise a database with an `.sql` file:

  ```
  sqlite3 -init test_all.sql test3.db
  ```

- Now check that `test3.db` was created:

  ```
  ls -l test3.db test.db
  ```

- Whenever you invoke `sqlite3` on the CLI (not in a code block), the program will be started. If you don't want that, end with `.exit`:

  ```
  sqlite3 -init test.sql test4.db .exit
  ls -l test*.db
  ```