Introduction to SQLite
Pierre Blavy

To cite this version:
Pierre Blavy. Introduction to SQLite. DEUG. Introduction to SQLite (Introduction to SQLite), 2017.
hal-02787575

HAL Id: hal-02787575
https://hal.inrae.fr/hal-02787575
Submitted on 5 Jun 2020

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L’archive ouverte pluridisciplinaire HAL, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d’enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Distributed under a Creative Commons Attribution - ShareAlike 4.0 International License
Introduction to SQLite
A simple Database system

Pierre BLAVY ¹

¹INRA, MoSAR

2017-04-10
Introduction: what’s this talk about

You will learn

- Why you should (not) use a database system
- SQLite
  - Database system
  - Local
  - Single user
- How to do your first database

You will NOT learn

- How to design complex databases
- Advanced SQL
What's a relational database?

**Data tables**

<table>
<thead>
<tr>
<th>person</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>person_id</td>
<td>first_name</td>
</tr>
<tr>
<td>1</td>
<td>Janine</td>
</tr>
<tr>
<td>2</td>
<td>Thérèse</td>
</tr>
<tr>
<td>3</td>
<td>Paul</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>cd</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>cd_id</td>
<td>title</td>
</tr>
<tr>
<td>1</td>
<td>La religion du flip</td>
</tr>
<tr>
<td>2</td>
<td>Here be dragons</td>
</tr>
<tr>
<td>3</td>
<td>SM</td>
</tr>
</tbody>
</table>

**Relations**

<table>
<thead>
<tr>
<th>person_own_cd</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>person_id</td>
<td>CD_id</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>

Janine owns La religion du flip
Janine owns Here be dragons
Thérèse owns SM
Databases v.s. Spreadsheet

**Spreadsheet is good if**
- Easy to use (at the beginning)
- Few tables
- Not too much data
- **No relations** (see latter)

**Spreadsheet are bad if**
- Relations
- A lot of data
- Evolution
- Interoperability
Introduction

I’ll not use a spreadsheet. What do I do?

When should I use SQLite?

- Relations between objects (any DB)
- Local and single user (SQLite)
- Exchange data between programs (SQLite)

When should I NOT use SQLite?

- A single simple small table, and no relations (use Spreadsheet)
- Multi user system (use MySQL, PostgreSQL, ...)
- Constantly evolving data structure (use NO-SQL?)
- Data that cannot be decomposed into tables and relations
Spreadsheets are bad for storing relations

**Data as Spreadsheet**

<table>
<thead>
<tr>
<th>Who owns what</th>
</tr>
</thead>
<tbody>
<tr>
<td>First_name</td>
</tr>
<tr>
<td>----------------</td>
</tr>
<tr>
<td>Janine</td>
</tr>
<tr>
<td>Janine</td>
</tr>
<tr>
<td>Thérèse</td>
</tr>
</tbody>
</table>

**Problems**

- Redundency (Janine Tutor)
- Missing data (Paul Auchon?)

→ Add a line?

<table>
<thead>
<tr>
<th>Who owns what</th>
</tr>
</thead>
<tbody>
<tr>
<td>First_name</td>
</tr>
<tr>
<td>----------------</td>
</tr>
<tr>
<td>Paul</td>
</tr>
</tbody>
</table>
Spreadsheets are bad for storing relations

Data as spreadsheet

<table>
<thead>
<tr>
<th>First_name</th>
<th>Last_name</th>
<th>Title</th>
<th>Artist</th>
</tr>
</thead>
<tbody>
<tr>
<td>Janine</td>
<td>Tutor</td>
<td>La religion du flip</td>
<td>Stupeflip</td>
</tr>
<tr>
<td>Janine</td>
<td>Tutor</td>
<td>Here be dragons</td>
<td>Kilimanjaro darkjazz ensemble</td>
</tr>
<tr>
<td>Thérèse</td>
<td>Ponsable</td>
<td>SM</td>
<td>Metallica</td>
</tr>
<tr>
<td>Paul</td>
<td>Auchon</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Hard to update

- Thérèse gives SM to Janine (don’t loose Therese)
- Rename Janine as Bob
  - Full scan of data
  - What about the “Janine and the Mixtape” band

Hard to evolve

- What if you add books, and who owns which book?
- How to do complex queries :
  - Which CD have people who read Lovecraft?
SQLite is available nearly everywhere

Perfect format for mixing programs

**Softwares**

Internet SQLite online [https://sqliteonline.com/](https://sqliteonline.com/)


Linux sqlite3, SQLite studio Ask your package manager

**Libraries**

R package "RSQLite" [http://cpc.cx/iZj](http://cpc.cx/iZj)

C libssqlite [http://cpc.cx/iZh](http://cpc.cx/iZh)

C++ sql_wrapper [http://cpc.cx/iZi](http://cpc.cx/iZi)

Matlab native [http://cpc.cx/iZg](http://cpc.cx/iZg)

Python sqlite3 [http://cpc.cx/j0R](http://cpc.cx/j0R)
SQL tutorial
Database guidelines

Tables (doc: http://www.bkent.net/Doc/simple5.htm)

- One data table per real life object with one ID column
- One table per relation, relations should only reference existing ID’s

Be atomic

- NO : adress
- YES: name, street, number, zip_code, city, country

Be stable

- NO : age (require daily update)
- YES: birth date

Dates

- NO : date as dd-mm-yyyy (alphabetical ≠ chronological)
- NO : daylight saving time, localtime (confusion)
- YES: UTC as yyyy-mm-dd hh:mm:ss
- YES: seconds since documented epoch (ex unix timestamp)
Example

```sql
create table person(
    person_id    integer primary key autoincrement,
    first_name   varchar,
    last_name    varchar
);
```

Guidelines

- ID's as: `xxx_id` integer primary key autoincrement
  - primary key: indexed, unique, not null
  - autoincrement: new ids are automatically generated

- Others fileds: `xxx` datatype (doc: http://cpc.cx/iZl)

<table>
<thead>
<tr>
<th>datatype</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>integer</td>
<td>integer number like 1, dates as seconds since epoch</td>
</tr>
<tr>
<td>float</td>
<td>floating point number like 3.5</td>
</tr>
<tr>
<td>varchar</td>
<td>character string, dates as yyyy-mm-dd</td>
</tr>
</tbody>
</table>
SQL: create relation table

PRAGMA foreign_keys = ON;
create table person_own_cd(
    person_id integer not null,
    cd_id integer not null,
    foreign key person_id references person(person_id) on delete cascade,
    foreign key cd_id references cd(cd_id) on delete cascade
);

- Only columns that contains IDs
  - IDs references objects in data tables (here person and cd)
  - IDs names should match referenced names (see latter NATURAL JOIN)
  - IDs type must match referenced type (here integer)
  - IDs columns are most of the time not null

- Foreign keys (doc https://sqlite.org/foreignkeys.html)
  - Enable support once with PRAGMA foreign_keys = ON;
  - foreign key xxx references ttt(yyy): each value of column xxx of this table must exists in the column yyy of table ttt;
  - on delete cascade: deleting in referenced table, also delete here. Here, deleting a person or a CD deletes the related lines in person_own_cd
Import your data: from csv

Use your graphical interface
  - Check for Data or Import in menu

Commandline in sqlite3 doc: http://cpc.cx/j0X
create table xxx(...);
.mode csv
.separator ","
.import test.csv tablename
Import your data

**SQL : Insert**

```sql
insert into person(first_name,last_name) values
('Janine' , 'Tutor'),
('Thérèse', 'Ponsable'),
('Paul'   , 'Auchon');
```

- Only **NOT NULL** and primary key column have to be filled
- ID's will be automatically generated (**autoincrement**)  
- Use NULL For missing values
Import your data
Idem for the other tables

```sql
insert into cd(title,artist) values
('La religion du flip','Stupeflip'),
('Here be dragons' , 'Killimanjaro darkjazz ensemble' ),
('SM' ,'Metallica');
```

```sql
insert into person_own_cd(person_id,cd_id) values
(1,1),
(1,2),
(2,3);
```
Export data to csv

Use your graphical interface

- Check for Data or Export in menu

Use sqlite3 commandline doc: http://cpc.cx/j0Y

```
.headers on
.mode csv
.separator ","
.output test.csv
SELECT * FROM table;
```
SQL : SELECT (doc : http://cpc.cx/j0h)

SELECT col1, col2, ... -- columns
FROM table1, table2, ... -- cartesian product of tables
JOIN table3 on cond -- joints
WHERE condition1 -- condition
AND condition2 -- other conditions
AND condition3 -- ...

GROUP BY col1, col2,... -- group lines

HAVING condition -- condition on grouped lines

ORDER BY col1, col2,... -- sort results

;
SQL : SELECT examples

Select everything in person
SELECT * from person;

Select specific columns in person
SELECT first_name, last_name FROM person;

Select specific lines:
last name starts with a letter between B and Q
SELECT * from person
WHERE lower(last_name) >= "b"
AND lower(last_name) <= "q";

- lower is a function that convert a string to lowercase
- Others functions : https://sqlite.org/lang_corefunc.html
Join tables

Cartesian product:

SELECT * from person, person_own_cd

<table>
<thead>
<tr>
<th>person</th>
<th>person_own_cd</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>id</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>first_name</th>
<th>last_name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Janine</td>
<td>Tutor</td>
</tr>
<tr>
<td>Thérèse</td>
<td>Ponsable</td>
</tr>
<tr>
<td>Paul</td>
<td>Auchon</td>
</tr>
</tbody>
</table>

Pierre BLAVY (INRA)
Join tables

Join = Cartesian product + condition

SELECT * from person, person_own_cd
WHERE person.person_id = person_own_cd.person_id

<table>
<thead>
<tr>
<th>person</th>
<th>person_own_cd</th>
</tr>
</thead>
<tbody>
<tr>
<td>person_id</td>
<td>person_id</td>
</tr>
<tr>
<td>first_name</td>
<td>cd_id</td>
</tr>
<tr>
<td>last_name</td>
<td>person_id</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th>person_id</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Janine</td>
<td>Tutor</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>Janine</td>
<td>Tutor</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>Janine</td>
<td>Tutor</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Thérèse</td>
<td>Ponsable</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Thérèse</td>
<td>Ponsable</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Thérèse</td>
<td>Ponsable</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Paul</td>
<td>Auchon</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Paul</td>
<td>Auchon</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Paul</td>
<td>Auchon</td>
<td>2</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Janine owns the CD 1 (La religion du flip)
Janine owns the CD 2 (Here be dragons)
Thérèse owns the CD 3 (SM)
Join tables

Cartesian product + condition
SELECT * from person, person_own_cd
WHERE person.person_id = person_own_cd.person_id;

JOIN ON
SELECT * from person JOIN person_own_cd
ON person.person_id = person_own_cd.person_id;

- Easy way to distinguish between joint condition (ON) and others conditions (WHERE)

NATURAL JOIN
SELECT * from person NATURAL JOIN person_own_cd;

- NATURAL JOIN automatically write a condition for columns of the same name here person.person_id = person_own_cd.person_id
CREATE VIEW view_cd AS
    SELECT * FROM person
    NATURAL JOIN person_own_cd
    NATURAL JOIN cd;

<table>
<thead>
<tr>
<th>view_cd</th>
</tr>
</thead>
<tbody>
<tr>
<td>person_id</td>
</tr>
<tr>
<td>first_name</td>
</tr>
<tr>
<td>last_name</td>
</tr>
<tr>
<td>cd_id</td>
</tr>
<tr>
<td>title</td>
</tr>
<tr>
<td>artist</td>
</tr>
<tr>
<td>----------</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>Janine</td>
</tr>
<tr>
<td>Tutor</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>La religion du flip</td>
</tr>
<tr>
<td>Stupéflip</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>Janine</td>
</tr>
<tr>
<td>Tutor</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>Here be dragons</td>
</tr>
<tr>
<td>Killimanjaro darkjazz...</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>Thérèse</td>
</tr>
<tr>
<td>Ponsable</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>SM</td>
</tr>
<tr>
<td>Metallica</td>
</tr>
</tbody>
</table>

- Views allow to store queries results as **dynamic tables**
- Views can be used as table, ex: SELECT * from view_cd;
- Views are always up to date
Update tables

**Rename Janine Tutor as Bob Tutor**

```sql
UPDATE person SET first_name = "Bob"
WHERE person_id = 1
```

**Thérèse gives SM to Janine**

```sql
UPDATE person_own_cd set person_id = 1 --1 is Janine
WHERE person_id = 2 --2 is Thérèse
AND cd_id = 3; --3 is SM of Metallica
```

- You **can** change data fields in data table
- You **can** change relations
- You **can not** change IDs in data tables
Delete SM of Metallica

DELETE FROM cd
WHERE cd_id = 3;

- Tip: try first with SELECT * instead of DELETE
- on delete cascade will drop related relations (here Thérèse owns SM)
RSqlite : connect/disconnect

Connect to a database
install.packages("RSQLite"); #once
library("RSQLite");
db = dbConnect(RSQLite::SQLite(),"a_file.sqlite");

Close connection
dbDisconnect(db);
RSqlite : Execute, bind, get

**Execute a query**

```r
dbExecute(db,"create table test(i integer)");
```

**Bind**

# data from R

# bind c(1,2), to the :AAA parameter, then execute
```r
dbExecute(db,"insert into test(i)values(:AAA)",
         list(AAA=c(1,2)));
```

**Get**

# data from sqlite

```r
d1 = dbGetQuery(db,"select * from test");

d2 = dbGetQuery(db,"select * from test where i > :i_min",
             list(i_min=1));
```

- No data expected : dbExecute. Returns: number of affected rows
- Data expected dbGetQuery. Returns: query data in a data.frame
- Use :XXX and list(XXX=something) to bind parameters
**Transactions**

dbBegin(db); #start a transaction
#... some queries

if(everything_OK){
    dbCommit(db); #actually do the changes
} else{
    dbRollback(db); #don’t change anything
}

- Transaction makes database changes atomic
- Grouping changes (like many inserts) in a transaction increase speed
- You must use transactions to keep DB consistency
Doc and conclusions

Learn by yourself
   SQL tutorial  https://www.w3schools.com/sql/
   SQLite doc   https://www.sqlite.org/lang.html

Conclusions
   + Simple, fast, local
   + Supported nearly everywhere
   + Public domain code, open-source libs
   + Basic operations are simple
     - Partial support of SQL
     - Single user (NO access rights, BAD concurrency)
   - Need some learning, but not so much.
   - Designing complex databases is still hard