Introduction to SQLite

Pierre Blavy

Introduction to SQLite

A simple Database system

Pierre BLAVY [1]

[1]INRA, 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

person		
person_id	first_name	Last_name
1	Janine	Tutor
2	Thérèse	Ponsable
3	Paul	Auchon

cd		
cd_id	title	artist
1	La religion du flip	Stupeflip
2	Here be dragons	Kilimanjaro darkjazz ensemble
3	SM	Metallica

Relations

person_own_cd	
person_id	CD_id
1	1
1	2
2	3

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 begining)
- 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

Who owns what			
First_name	Last_name	Title	Artist
Janine	Tutor	La religion du flip	Stupeflip
Janine	Tutor	Here be dragons	Kilimanjaro darkjazz ensemble
Thérèse	Ponsable	SM	Metallica

Problems

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

→ Add a line ?

Who owns what			
First_name	Last_name	Title	Artist
Paul	Auchon		

Spreadsheets are bad for storing relations

Data as spreadsheet

Who owns what			
First_name	Last_name	Title	Artist
Janine	Tutor	La religion du flip	Stupeflip
Janine	Tutor	Here be dragons	Kilimanjaro darkjazz ensemble
Thérèse	Ponsable	SM	Metallica
Paul	Auchon		

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

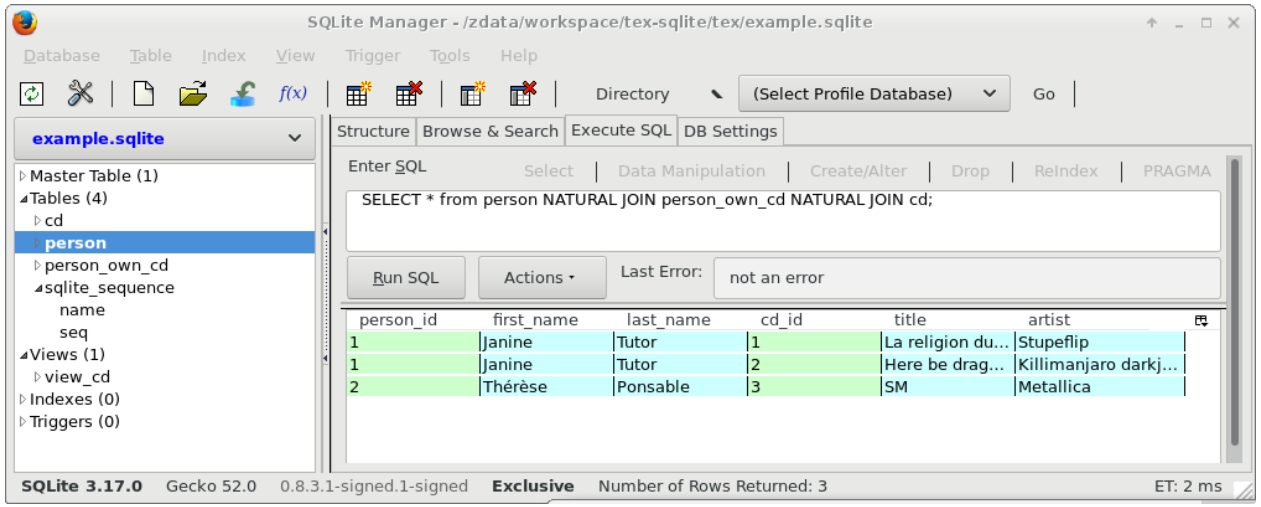
Hard to evolve

- 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/`
Mac, Windows, Linux	sqlitebrowser	`http://sqlitebrowser.org/`
Linux	sqlite3, SQLite studio	Ask your package manager
Libraries		
R	package "RSQLite"	`http://cpc.cx/iZj`
C	libsqlite	`http://cpc.cx/iZh`
C++	sql_wrapper	`http://cpc.cx/iZi`
Matlab	native	`http://cpc.cx/iZg`
Python	sqlite3	`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 $\neq$ chronological)
- NO : daylight saving time, localtime (confusion)
- YES: UTC as yyyy-mm-dd hh:mm:ss
- YES: seconds since documented epoch (ex unix timestamp)

SQL : create data table

doc : http://cpc.cx/iZm

Example

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

Guidelines

- ID's as : `xxx_id integer primary key autoincrment`
  - `primary key` : indexed, unique, not null
  - `autoincrment` : new ids are automatically generated
- Others fileds : `xxx datatype` (doc : `http://cpc.cx/iZl`)

datatype	description
integer	integer number like 1, dates as seconds since epoch
float	floating point number like 3.5
varchar	character string, dates as yyyy-mm-dd

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

```
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

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

```
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
                          -- doc : http://cpc.cx/j0k
HAVING  condition         -- condition on groupped lines
                          -- doc : http://cpc.cx/j0U
ORDER BY col1, col2,...   -- sort results
                          -- doc : http://cpc.cx/j0l
;
```

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
```

person		
person_id	first_name	last_name
1	Janine	Tutor
2	Thérèse	Ponsable
3	Paul	Auchon

X

person_own_cd	
person_id	cd_id
1	1
1	2
2	3

=

person			person_own_cd	
person_id	first_name	last_name	person_id	cd_id
1	Janine	Tutor	1	1
1	Janine	Tutor	1	2
1	Janine	Tutor	2	3
2	Thérèse	Ponsable	1	1
2	Thérèse	Ponsable	1	2
2	Thérèse	Ponsable	2	3
3	Paul	Auchon	1	1
3	Paul	Auchon	1	2
3	Paul	Auchon	2	3

Join tables

Join = Cartesian product + condition

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

person			person_own_cd		
person_id	first_name	last_name	person_id	cd_id	
1	Janine	Tutor	**1**	1	Janine owns the CD 1 (La religion du flip)
1	Janine	Tutor	**1**	2	Janine owns the CD 2 (Here be dragons)
1	Janine	Tutor	2	3	
2	Thérèse	Ponsable	1	1	
2	Thérèse	Ponsable	1	2	
2	Thérèse	Ponsable	**2**	3	Thérèse owns the CD 3 (SM)
3	Paul	Auchon	1	1	
3	Paul	Auchon	1	2	
3	Paul	Auchon	2	3	

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 : store dynamic queries

```
CREATE VIEW view_cd AS
  SELECT * FROM person
  NATURAL JOIN person_own_cd
  NATURAL JOIN cd;
```

view_cd					
person_id	first_name	last_name	cd_id	title	artist
1	Janine	Tutor	1	La religion du flip	Stupéflip
1	Janine	Tutor	1	Here be dragons	Killimanjaro darkjazz...
2	Thérèse	Ponsable	3	SM	Metallica

- 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

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

Thérèse gives SM to Janine

```
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

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

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

Bind

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

Get

```
#data from sqlite
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

RSqlite : Transaction

Transactions

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

if(everything_OK){
  dbCommit(db); #actualy do the changes
}else{
  dbRollback(db); #don't change anything
}
```

- Transaction makes database changes atomic
- Groping 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 concurency)
- Need some learning, but not so much.
- Designing complex databases is still hard