Foreign Key constraints

are now supported by SQLite

Starting since v. **3.6.19** SQLite introduced fully support for Foreign Key constraints. And obviously, SpatiaLite too inherits such really interesting feature.

Here you can find the original SQLite doc page about Foreign Key constraints:
http://www.sqlite.org/foreignkeys.html

A quick and fast tutorial:

Step #1:

```
C:\>spatialite
SpatiaLite version ..: 2.4.0      Supported Extensions:
        - 'VirtualShape'          [direct Shapefile access]
        - 'VirtualText            [direct CSV/TXT access]
        - 'VirtualNetwork         [Dijkstra shortest path]
        - 'RTree'                 [Spatial Index - R*Tree]
        - 'MbrCache'              [Spatial Index - MBR cache]
        - 'VirtualFDO'            [FDO-OGR interoperability]
        - 'SpatiaLite'            [Spatial SQL - OGC]
PROJ.4 version ......: Rel. 4.7.1, 23 September 2009
GEOS version ........: 3.1.1-CAPI-1.6.0
SQLite version ......: 3.6.20
Enter ".help" for instructions
spatialite>
```

Launch the **spatialite** CLI front end: as you can notice it includes SQLite v. **3.6.20**, supporting the Foreign Key constraints. You can use the **spatialite-gui** tools as well, if you wish.

Step #2:

```
spatialite> PRAGMA foreign_keys;
1
```

By default any SQLite connection starts keeping the Foreign Key constraints disabled: this is to ensure full compatibility with older versions of SQLite.
In order to enable Foreign Key constraints you have to declare: `PRAGMA foreign_keys = 1;`
But SpatiaLite performs this task automatically: as you can see in the above step, Foreign Key constraints are enabled as soon as **spatialite** establishes a database connection.

Important notice:
This isn't true when using the SpatiaLite's C API. In this case the developer is fully responsible for activating (or not) the Foreign Key constraints.

Step #3:

```
spatialite> CREATE TABLE mother (
       ...> last_name TEXT NOT NULL,
       ...> first_name TEXT NOT NULL,
       ...> birth_date DATETIME NOT NULL,
       ...> CONSTRAINT pk_mother PRIMARY KEY
       ...> (last_name, first_name, birth_date));
```

Now we'll create a **mother** table.

- each *mother* is identified by her full name and birth date.
- we define a Primary Key spanning over three columns: so we are granted that only one row can be inserted presenting the same values combination. This is a so-called unique identifier.

Step #4:

```
spatialite> CREATE TABLE daughter (
       ...> last_name TEXT NOT NULL,
       ...> first_name TEXT NOT NULL,
       ...> birth_date DATETIME NOT NULL,
       ...> mother_last_name TEXT NOT NULL,
       ...> mother_first_name TEXT NOT NULL,
       ...> mother_birth_date DATETIME NOT NULL,
       ...> CONSTRAINT pk_daughter PRIMARY KEY
       ...> (last_name, first_name, birth_date),
       ...> CONSTRAINT fk_daughter FOREIGN KEY
       ...> (mother_last_name, mother_first_name, mother_birth_date)
       ...> REFERENCES mother (last_name, first_name, birth_date));
```

Now we'll create a **daughter** table:

- each daughter is identified by her full name and birthdate
- and we've defined a Primary Key to ensure uniqueness
- but now we've defined a Foreign Key as well:
  - so that each *daughter* row references a corresponding row into the *mother* table

This one is a so called one-to-many relationship: each one *mother* may have zero, one or many *daughters*, but each *daughter* has to have one and only one *mother*.

Step #5:

```
spatialite> INSERT INTO mother VALUES ('Smith', 'Jane', '1949-07-12');
spatialite> INSERT INTO mother VALUES ('Green', 'Mary', '1967-02-18');
spatialite> INSERT INTO mother VALUES ('White', 'Susan', '1978-06-12');
```

Now we'll INSERT some rows into the *mother* table.

```
spatialite> INSERT INTO mother VALUES ('Green', 'Mary', '1967-02-18');
SQL error: columns last_name, first_name, birth_date are not unique
```

Obviously we are not allowed to INSERT again this row, because such an action will violate the uniqueness constraint granted by the Primary Key we've defined.

Step #6:

```
spatialite> INSERT INTO daughter VALUES (
       ...> 'Ross', 'Stephanie', '1975-03-02',
       ...> 'Smith', 'Jane', '1949-07-12');
spatialite> INSERT INTO daughter VALUES (
       ...> 'Ross', 'Helen', '1978-09-21',
       ...> 'Smith', 'Jane', '1949-07-12');
```

Now we'll INSERT some rows into the *daughter* table.

```
spatialite> INSERT INTO daughter VALUES (
       ...> 'McCain', 'Mary', '1972-11-09',
       ...> 'Smith', 'Mary', '1955-04-05');
SQL error: foreign key constraint failed
```

Please note: this INSERT will fail, because we've not yet defined any *Mary Smith* into the *mother* table. This one is a Foreign Key constraint violation, and SQLite forbids this operation.

Step #7:

```
spatialite> DELETE FROM mother
       ...> WHERE last_name = 'White' AND first_name = 'Susan';
```

There is nothing wrong in this DELETE statement

```
spatialite> DELETE FROM mother
       ...> WHERE last_name = 'Smith' AND first_name = 'Jane';
SQL error: foreign key constraint failed
```

Please note: this DELETE will fail, because there are two rows into the *daughter* table referencing the *mother* row we are attempting to DELETE. This too is a Foreign Key constraint violation, and SQLite forbids this operation.

```
spatialite> BEGIN;
spatialite> DELETE FROM daughter
       ...> WHERE mother_last_name = 'Smith'
       ...> AND mother_first_name = 'Jane';
spatialite> DELETE FROM mother
       ...> WHERE last_name = 'Smith' AND first_name = 'Jane';
spatialite> COMMIT;
```

This works fine, because we are now deleting any dependent row form the *daughter* table before attempting to DELETE the required row from the *mother* table.