SpatiaLite test coverage on Java ?
YES !!!

More or less one year ago (on 2011 January 28) I performed some basic testing about using SpatiaLite on Java via the Xerial JDBC connector supporting SQLite.
If you are anyway interested on this old topic, you can read my original report from here:
http://www.gaia-gis.it/gaia-sins/spatialite-cookbook/html/java.html

So I thought that coming back yet another time on Java was an interesting and useful experience.
The most recent SpatiaLite's version supports a comprehensive test coverage, based on the top of ***gcov*** and ***lcov*** (both tools belong to the GNU C/C++ build toolchain).
So standing things, porting the full test coverage on a different language / compiler initially seemed to be really difficult, and porting on Java a rather impossible task.

But after a most careful evaluation, I soon discovered that a partial solution was possible, and was quite easy to be implemented.
Happily enough, Brad Hards, the initial developer of the SpatiaLite test coverage, wisely introduced a nice generic test case container (***check_sql_stmt.c***) fully based on execution of SQL statements.
Now we currently have about 2,000 SQL test cases available, and covering any possible SQL function supported by SpatiaLite.
Porting ***check_sql_stmt.c*** on Java is surely possible, and isn't too much difficult: but in order to better understand, let us first examine the detailed structure of a single SQL test case.

asgeojson7.testcase

```
asgeojson - bad args (3 arg, out of range option)
:memory: #use in-memory database
SELECT asgeojson(GeomFromText("Point(1 2)", 4326), 4, 22);
1 # rows (not including the header row)
1 # columns
asgeojson(GeomFromText("Point(1 2)", 4326), 4, 22)
{"type":"Point","coordinates":[1,2]}:0
```

- each single SQL test case is fully defined by a distinct text file [**.testcase** suffix]
- all test cases are stored within the **sql_test_cases** directory
- a # marks a comment: i.e. any further text found to the right of the leftmost # will be ignored
- the first line in the **.testcase** file is expected to contain the test name (useful to identify each individual test case)
- the second line contains the name of the DB-file to be connected in order to run the test case
  - as usual, the **:memory:** conventional path simply intends a memory-based DB
  - otherwise the corresponding DB-file is expected to be found into the **sql_test_cases** directory
  - if the DB-name ends with **_RO**, then a *read-only connection* to the DB-files is assumed to be established.
- the third line represents the SQL statement to be executed
- the fourth line contains the number of rows expected to be found in the returned resultset
- the fifth line contains the number of columns expected to be found in the returned resultset
- all the remaining lines have a variable layout, depending on the declared number of rows and columns.

- the next *n-columns* lines are expected to represent the column names
- any remaining line [*n-rows* * *n-columns*] is expected to represent a result value
- any result value can declare a *precision* (useful to truncate floating point values, so to shield the test against any platform-depending rounding effect): in this case a ***:precision*** suffix will be appended. As a special case a **:0** precision suffix simply means "*never truncate*", and is required in order to mask GML and KML results (obviously containing lots of '**:**')

Implementing all this in Java isn't too much difficult: here is the basic workflow:

1. reading the **sql_test_cases** directory, so to identify any **.testcase** file
2. reading and parsing each **.testcase** file
3. establishing a DB connection as required by the test case
4. executing the test case SQL statement
5. fetching the resultset returned by the previous step
   A) checking the rows and columns count
   B) checking the column names
   C) checking the result values

You can find all this already implemented in the **SpatialiteTests.java**, **TestContainer.java** and **RunTest.java** sources. Reading and studying the source code may be an interesting opportunity to better understand the problem, but isn't the main goal of this report.
We'll focus our attention instead on more interesting general order facets, such as installation, environment setting and deployment.

Please note well: the general architecture adopted by the Java JDBC connector is very close to the similar approach adopted by .NET C#, Python, PHP and many others language bindings. So the following considerations could be easily applied to many others environments/platforms/languages.

Test platform #1:
Java is intrinsically cross-platform, but both SQLite and SpatiaLite are C native libraries; accordingly to this testing more than a single platform seems to be a wise option.
The first platform I tested is configured as follows:

- Intel Core i5 model 670
- Windows 7 Professional 64 bit (SP1)
- Java JDK 7 update 3 for Windows [x86_64, 64 bit]
- Java JRE 7 update 3 for Windows [x86_64, 64 bit]

Test platform #2:

- Windows XP Professional 32 bit (SP3) [Virtual PC]
- Java JDK 7 update 3 for Windows [x86, 32 bit]
- Java JRE 7 update 3 for Windows [x86, 32 bit]

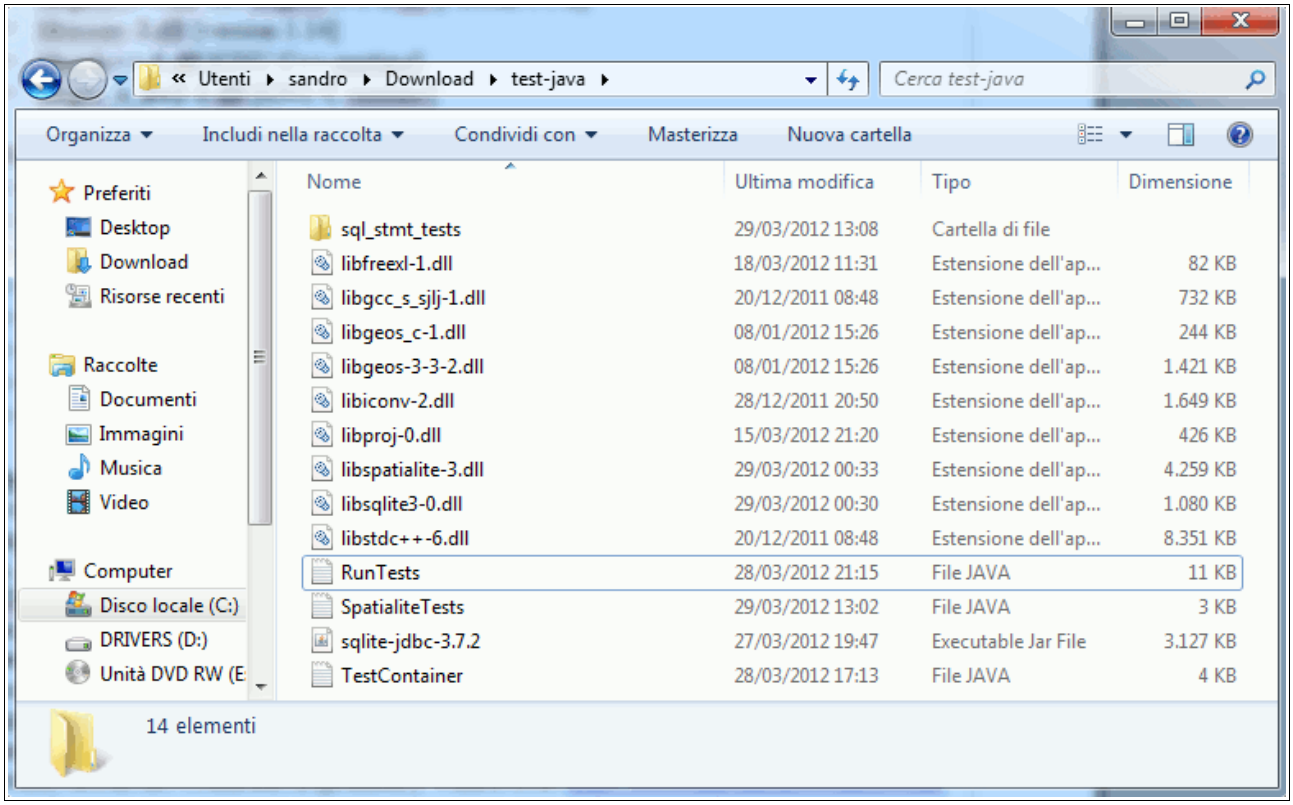
The Xerial JDBC connector:
Required in order to connect SQLite DB-files on Java. I simply used the most recent version available from: http://www.xerial.org/maven/repository/artifact/org/xerial/sqlite-jdbc/3.7.2/
Please note: this one is nowadays obsolete, because it's based on SQLite 3.7.2: the current SQLite version is 3.7.11, and the SQLite developers declared any version prior to 3.7.6.3 as *deprecated*.
Bad news doesn't stop here: the Xerial JDBC project seems to apparently be quite dead. The most recent update I noticed is dated May 2011; not at all comfortable … serious doubts surely arise about the future evolution of this project.

The SpatiaLite's own DLL (*and required dependecies*):
I used the most recent development snapshot available, this corresponding to 3.1.0-RC2
All the required DLLs were built using MinGW (both 32 bit and 64 bit editions).
These are binary/native dynamic libraries, so strictly platform dependent; here is the full list for both 32 bit and 64 bit archs:

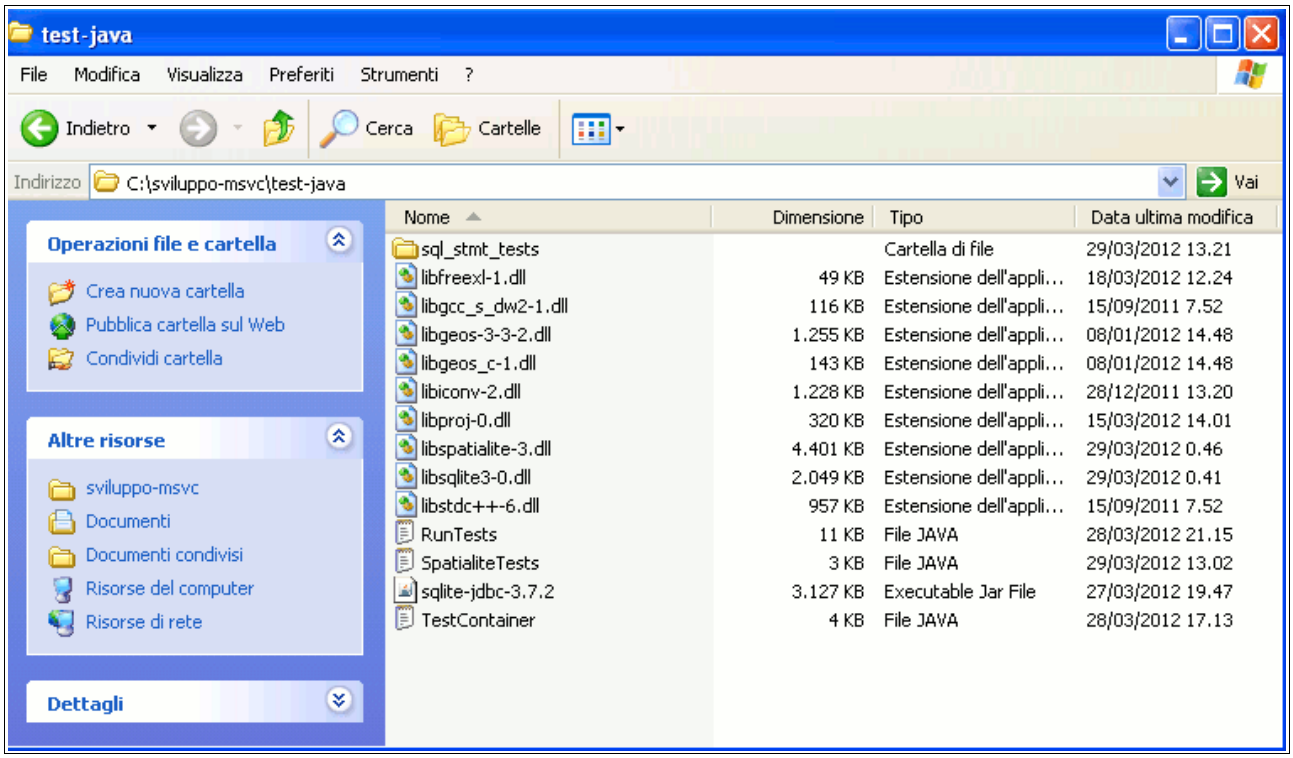
- Win32
  - **libspatialite-3.dll** [version 3.1.0-RC]
  - **libsqlite3-0.dll** [version 3.7.11]
  - **libfreexl-1.dll** [version 1.0.0b]
  - **libproj-0.dll** [version 4.8.0]
  - **libgeos_c-1.dll** and **libgeos-3-3-2.dll** [version 3.3.2]
  - **libiconv-2.dll** [version 1.14]
  - **libstdc++-6.dll** [GNU C++ runtime]
  - **libgcc_s_dw2-1.dll** [GNU C runtime]

- Win64
  - **libspatialite-3.dll** [version 3.1.0-RC]
  - **libsqlite3-0.dll** [version 3.7.11]
  - **libfreexl-1.dll** [version 1.0.0b]
  - **libproj-0.dll** [version 4.8.0]
  - **libgeos_c-1.dll** and **libgeos-3-3-2.dll** [version 3.3.2]
  - **libiconv-2.dll** [version 1.14]
  - **libstdc++-6.dll** [GNU C++ runtime]
  - **libgcc_s_sjlj-1.dll** [GNU C runtime]

Where to install the JDBC connector and the DLLs:
There is no easy and obvious answer: it's the well known *DLL hell* afflicting Windows.
After experiencing serious frustration and painful depression deriving from many unsuccessful attempts, I finally discovered that the easiest (may be, not at all elegant) solution is simply to copy all the DLLs and the JDBC **.jar** directly into the same directory (aka folder) where I had already copied my Java sources and the **sql_stmt_tests** directory.
Please note: when the Xerial JDBC connector fails to load SpatiaLite as a dynamic extension for one reason or the other, it simply reports some generic and meaningless error message.
Don't expect any help coming from these useless poor diagnostic. Rely instead on your own system skills, and perform some independent test, may be using the **sqlite3.exe** command line front-end.
I found really useful the wonderful Dependency Walker tool: http://www.dependencywalker.com/



here is shown the layout I've finally adopted for my Java test on Windows 7 64 bit



and this is the corresponding one I've adopted on Windows XP 32 bit

Compiling the Java classes:

```
>C:\"program files"\java\jdk1.7.0_03\bin\javac \
  -classpath ".;./sqlite-jdbc-3.7.2.jar" \
  RunTests.java TestContainer.java SpatialiteTests.java
```

not at all sophisticated … I've simply used the command line **javac** compiler

Running the test cases:

```
>C:\"program files"\java\jre7\bin\java \
  -classpath ".;./sqlite-jdbc-3.7.2.jar" SpatialiteTests

Test case: acos-text ... skipping Math SQL functions ...
Test case: acos(1) ... skipping Math SQL functions ...
Test case: acos(1.0) ... skipping Math SQL functions ...
Test case: acos(2) ... skipping Math SQL functions ...
Test case: acos(2.0) ... skipping Math SQL functions ...
Test case: asbinary
Test case: asbinary - POINTZM
Test case: asbinary - MULTILINESTRING
Test case: asbinary - MULTILINESTRINGZ
...
Test case: ST_WKBToSQL - bad blob
Test case: ST_WKBToSQL - float
Test case: ST_WKTToSQL
Test case: ST_WKTToSQL - blob
Test case: ST_WKTToSQL - bad WKT
Test case: yards to metres

tests found   : 2057
skipped tests : 93
failed tests  : 0

All tests succesfully completed
```

yet again, I've simply used the command line in order to run the test.
All right, now we are granted for sure that even on Java some 2,000 SQL test cases were successfully checked, without any failure at all.
Please note well: two different SQLite's versions were involved; one internally shipped by the JDBC connector (3.7.2), the other used to build SpatiaLite (3.7.11).
Anyway this configuration seems to be reasonably stable, robust and reliable: and I doesn't experienced any unpleasant crash or JVM failure.

Just a final clarification about ***skipped test*** ...
As I discovered on my own, the Xerial JDBC connector silently includes a *silver bullet*: all SQL Math functions [*Sin(), Cos(), Log(), Tan()* ..] are unconditionally added to the basic SQLite's own SQL core. Sadly enough, there is no way to override the Xerial own implementation.
And sadly too, their implementation is strongly different from the one supported by SpatiaLite (different signatures, different results): so the only reasonable solution is skipping Math test cases.