SpatiaLite: 3D and compressed geometries

Addendum

Starting since version **2.4.0** SpatiaLite supports **3D** geometries as well.
The present addendum is intended to quickly explain the most relevant implementation details.

Geometries supported by SpatiaLite can be represented in one of the following spaces:

- **XY**: each point or vertex is identified by two coordinates [X,Y]. This corresponds to the classic **2D** model
- **XYZ**: each point or vertex is identified by three coordinates [X,Y,Z]. This corresponds to the classic **3D** model
- **XYM**: each point or vertex is identified by two coordinates and stores a **M**easure as well [X,Y,M]. This one is a **2D** model, because the Measure value has nothing to deal with geometry dimensions.
- **XYZM**: each point or vertex is identified by three coordinates and stores a **M**easure as well [X,Y,Z,M]. This one is a **3D** model

This approach is fully consistent with:

- ESRI Shapefiles
- OGC [Open Geospatial Consortium] specifications
- main GIS software implementations
- quite any spatially enabled DBMS

Strictly speaking, this is not a true 3D model, but really is a so called **2.5D** model.
This is because the Z and M dimensions can be actually stored into the DBMS, but they are ignored for any other practical purpose.
Don't expect functions such as **Intersects()**, **Difference()** or **Distance()** does really take care of the Z dimension: they simply compute their results using the [XY] values and ignoring at all any 3D consideration.

SpatiaLite now supports **compressed geometries** as well: this one is a SpatiaLite's own specific feature, allowing to save some disk space [actual compression widely depends on the actual data to be stored into the DB].
This may be a good new, specially for the many ones using SpatiaLite on *mobile devices* or in other crimped hardware configurations, where managing storage always is an hard fighting.

<u>Geometry classes:</u>

Accordingly to the above premise, the following Geometry Classes are now supported by SpatiaLite:

Class	**ID**
NONE	0
POINT	1
LINESTRING	2
POLYGON	3
MULTIPOINT	4
MULTILINESTRING	5
MULTIPOLYGON	6
GEOMETRYCOLLECTION	7
POINTZ	1001
LINESTRINGZ	1002
POLYGONZ	1003
MULTIPOINTZ	1004
MULTILINESTRINGZ	1005
MULTIPOLYGONZ	1006
GEOMETRYCOLLECTIONZ	1007
POINTM	2001
LINESTRINGM	2002
POLYGONM	2003
MULTIPOINTM	2004
MULTILINESTRINGM	2005
MULTIPOLYGONM	2006
GEOMETRYCOLLECTIONM	2007
POINTZM	3001
LINESTRINGZM	3002
POLYGONZM	3003
MULTIPOINTZM	3004
MULTILINESTRINGZM	3005
MULTIPOLYGONZM	3006
GEOMETRYCOLLECTIONZM	3007

WKT notations:

You can use WKT [***W****ell* ***K****nown* ***T****ext*] expressions to represent each geometry class, as shown in the following examples:

```
POINT(13.21 47.21)
POINTZ(13.21 47.21 0.21)
POINTM(13.21 47.21 1000.0)
POINTZM(13.21 47.21 0.21 1000.0)

LINESTRING(15.21 57.58, 15.81 57.12)
LINESTRINGZ(15.21 57.58 0.31, 15.81 57.12 0.33)
LINESTRINGM(15.21 57.58 1000.0, 15.81 57.12 1100.0)
LINESTRINGZM(15.21 57.58 0.31 1000.0, 15.81 57.12 0.33 1100.0)

MULTIPOINT(15.21 57.58, 15.81 57.12)
MULTIPOINTZ(15.21 57.58 0.31, 15.81 57.12 0.33)
MULTIPOINTM(15.21 57.58 1000.0, 15.81 57.12 1100.0)
MULTIPOINTZM(15.21 57.58 0.31 1000.0, 15.81 57.12 0.33 1100.0)

GEOMETRYCOLLECTION(
   POINT(13.21 47.21),
   LINESTRING(15.21 57.58, 15.81 57.12))
GEOMETRYCOLLECTIONZ(
   POINTZ(13.21 47.21 0.21),
   LINESTRINGZ(15.21 57.58 0.31, 15.81 57.12 0.33))
GEOMETRYCOLLECTIONM(
   POINTM(13.21 47.21 1000.0),
   LINESTRINGM(15.21 57.58 1000.0, 15.81 57.12 1100.0))
GEOMETRYCOLLECTIONZM(
   POINTZM(13.21 47.21 0.21 1000.0),
   LINESTRINGZM(15.21 57.58 0.31 1000.0, 15.81 57.12 0.33 1100.0))
```

Important notice:

Although existing OGC specifications tries to define conformance rules for both 2D and 3D geometries, you cannot expect a strong inter-operability between different implementations.
As a matter of fact this goal has been really achieved only in the case of 2D geometries [XY].
But when using 3D geometries [XYZ, XYM or XYZM] you cannot expect to safely transfer WKT or WKB notations between different DBMS, because there is an awful chaos of different (and mutually incompatible) formats or dialects.

Accordingly to this sad situation, using the old shapefile format seems to be the unique viable solution when transferring 3D data between different DBMS is absolutely required.

SQL functions:

Functions: **AsText(), AsBinary(), GeomFromText(), GeomFromWKB()** and alike will now support any Geometry class [following the above guide lines].

Function **GeometryType()** will now return the corresponding geometry class, i.e. 'LINESTRING', 'POINT ZM', 'MULTIPOLYGON M' …

Functions: **StartPoint()**, **PointN()**, **LinestringN(), GeometryN()** and alike always return a Geometry using the same dimension space as the original Geometry.

Individual coordinate values can now be inspected using the **X()**, **Y()**, **Z()** and **M()** functions. Please note:

- a call to the Z() function passing a POINT or POINTM geometry always return 0 (*zero*)
- and a call to the M() function passing a POINT or POINTZ geometry always return 0 (*zero*)

Explicit geometry class casting is allowed using the following functions:

- **CastToXY()** always return the corresponding Geometry in the [XY] space
- **CastToXYZ()** always return the corresponding Geometry in the [XYZ] space
- **CastToXYM()** always return the corresponding Geometry in the [XYM] space
- **CastToXYZM()** always return the corresponding Geometry in the [XYZM] space

Caveat: when reducing the dimensions [as in the case of calling **CastToXY()**], the unused dimensions [Z,M] will be simply omitted if present into the original Geometry.
And when augmenting the dimensions [as in the case of calling **CastToXYZM()**], any missing dimension [Z,M] will be represented as 0 (*zero*) into the returned Geometry.

Geometry type constraints:

The **AddGeometryColumn()** function is now defined as follows:

```
SELECT AddGeometryColumn('tbl', 'geom', 4326, 'POINT', 2);
SELECT AddGeometryColumn('tbl', 'geom', 4326, 'POINT', 'XY');
```

- both them requires the creation of a new POINT Geometry column in the [XY] space.

```
SELECT AddGeometryColumn('tbl', 'geom', 4326, 'POINT', 3);
SELECT AddGeometryColumn('tbl', 'geom', 4326, 'POINT', 'XYZ');
```

- both them requires the creation of a new POINT Geometry column in the [XYZ] space.

```
SELECT AddGeometryColumn('tbl', 'geom', 4326, 'POINT', 'XYM');
```

- requires the creation of a new POINT Geometry column in the [XYM] space.

```
SELECT AddGeometryColumn('tbl', 'geom', 4326, 'POINT', 'XYZM');
```

- requires the creation of a new POINT Geometry column in the [XYZM] space.

Please note: trying to INSERT or UPDATE a Geometry value not corresponding to the declared space dimensions for the corresponding column will now raise an exception.
This actually grants that each Geometry value stored into the same column has identical Geometry type and space dimensions.

Compressed Geometries:

SpatiaLite can handle the following compressed Geometry classes:

Class	ID
COMPRESSED_LINESTRING	1000002
COMPRESSED_POLYGON	1000003
COMPRESSED_LINESTRINGZ	1001002
COMPRESSED_POLYGONZ	1001003
COMPRESSED_LINESTRINGM	1002002
COMPRESSED_POLYGONM	1002003
COMPRESSED_LINESTRINGZM	1003002
COMPRESSED_POLYGONZM	1003003

Each compressed class is exactly equivalent to its corresponding uncompressed class: the only difference is in the binary internal representation used to store data.

While storing some uncompressed Geometry a double precision value [64 bits] is required to store each single coordinate value. Accordingly to this:

- storing a 1,000 vertices Linestring in the [XY] space will require 2,000 *doubles*, i.e. 128,000 bytes.
- storing the same 1,000 vertices Linestring in the [XYZ] space will require 3,000 *doubles*, i.e. 192,000 bytes.

Really huge Geometries are not at all uncommon in GIS data: finding many-thousand vertices Linestrings or Rings it's not at all exceptional while handling complex entities such as roads, rivers or administrative boundaries.

Applying some kind of Geometry compression we can sensibly reduce the required storage amount, accepting in the meanwhile some slight (may be, unnoticeable) precision loss.
The compressed storage implemented by SpatiaLite for Linestrings and Rings is as follows:

- the first vertex will be left untouched exactly as it is, i.e. it still uses double precision values
- and the last vertex will be left untouched as well, in order to fully preserve any required topology consistency.
- but any other intermediate vertex is now represented using single precision [32 bits] value. Due to the precision reduction we cannot any longer use the original coordinates: but we can easily compute the incremental differences existing with the immediately preceding vertex.

So, storing the above 1,000 vertices Linestring will now require (applying compression):

- 4 *doubles* and 1,996 *floats*, i.e. 64,128 bytes if using the [XY] space
- 6 *doubles* and 2,994 *floats*, i.e. 96,1922 bytes if using the [XYZ] space

As you can easily detect, applying compression to complex geometries will save about 50% of the required space: yes, we'll certainly introduce some slight approximation, but this may well be quite unnoticeable for most practical GIS purposes, such as map visualization.

Performance impact of geometry compression:
Don't be afraid too much ...

- yes, compressing and uncompressing geometries imposes some extra computational load
- but any modern CPU (even the slowest Celeron or Atom) has enough power to crunch lots of sums and subtractions in a quite instantaneous time
- on the other hand, disk access always is a painful bottle-neck, even on the most brilliant multi-core workstations.
- so, reducing by half the required I/O load surely represents a sensible benefit

After all, actual impact widely depends on several hardware-related factors (namely, CPU and disk speed).
But as a general rule, applying geometry compression doesn't have a negative impact on performance: and quite paradoxically, using a standard PC you can notice some global speed enhancement when applying geometries compression. This is easily explained: latest generation CPUs are really fast, and standard SATA disks are quite slow ...

Compression factors you can realistically achieve:
Don't expect any miracle ...

- POINT geometries aren't affected at all by compression
- simple Linestrings (or Rings) [i.e., ones containing very few vertices] doesn't benefit so much by applying compression
- any Spatial Index cannot benefit at all of geometry compression
- any other value contained into the DB [INTEGER, DOUBLE, TEXT] will not benefit at all by compressing geometries

Only the hugest Linestrings (or Rings) [i.e., ones containing many hundredths or thousands vertices] can really take full profit when applying geometry compression. So, after all, you cannot realistically expect to reach compression factors [for the whole DB] better than 25% or 30%: but this too can help under some very critical circumstances.

SQL functions supporting compression:

The **CompressGeometry()** function return a compressed geometry, and the **UncompressGeometry()** function return an ordinary, uncompressed geometry.
If the passed argument isn't a valid Geometry, then NULL will be returned.

Please note: trying to compress an already compressed geometry isn't an error, but you are simply wasting some CPU cycle to do nothing of useful: the same applies when trying to inflate an already uncompressed geometry.

In order to compress any geometry stored into a given DB, you can use the following SQL sample:

```
BEGIN;
UPDATE table_1 SET geom = CompressGeometry(geom);
UPDATE table_2 SET geom = CompressGeometry(geom);
...
UPDATE table_N SET geom = CompressGeometry(geom);
COMMIT;
VACUUM;
```