libspatialite v.2.4.0-RC5b Experimental

Spatial Index UPDATE

Just some considerations about R*Tree and Spatial Index; here is a short summary:

- How to corrupt an R*Tree (*oh, yes ... sometimes it happens ...*)
- Checking and Recovering broken R*Trees
- the brand new SpatialIndex module (Virtual Table)

How to corrupt an R*Tree Spatial Index

A short recapitulation to understand better:

- any SQLite R*Tree simply is a distinct table [*actually: a Virtual Table*]
- SQLite on its own is completely unaware of correspondences relating an R*Tree and the corresponding **`table.geometry`**
- SpatiaLite implements several **triggers** in order to ensure that the R*Tree would constantly be fully synchronized with the corresponding **`table.geometry`**
- A relational **`JOIN`** between the R*Tree and the corresponding **`table.geometry`** is ensured by correspondent **`ROWID`** values.
- Each single row stored within any SQLite's table is uniquely identified by a **`ROWID`** value.
- If the table has a **`PRIMARY KEY`**, then the **`ROWID`** value is immutably related to the **`PRIMARY KEY`** value(s).
- But when the table has no PRIMARY KEY, then the **`ROWID`** simply is the relative row number.

```
CREATE TABLE test (name TEXT NOT NULL);
SELECT AddGeometryColumn('test', 'geom', 4326, 'POINT', 'XY') ;
SELECT CreateSpatialIndex('test', 'geom');
INSERT INTO test (name, geom) VALUES ('a', MakePoint(1, 1, 4326));
INSERT INTO test (name, geom) VALUES ('b', MakePoint(2, 2, 4326));
INSERT INTO test (name, geom) VALUES ('c', MakePoint(3, 3, 4326));
INSERT INTO test (name, geom) VALUES ('d', MakePoint(4, 4, 4326));
INSERT INTO test (name, geom) VALUES ('e', MakePoint(5, 5, 4326));
SELECT ROWID, name, ST_AsText(geom) FROM test;
```

ROWID	name	ST_AsText(geom)
1	a	POINT(1 1)
2	b	POINT(2 2)
3	c	POINT(3 3)
4	d	POINT(4 5)
5	e	POINT(5 5)

Absolutely nothing strange in all this: we have simply created a new table, then inserting just few rows.
Please note well: this table *has no **`PRIMARY KEY`*** defined.

```
SELECT pkid, xmin, xmax, ymin, ymax FROM idx_test_geom;
```

pkid	xmin	xmax	ymin	ymax
1	1.000000	1.000000	1.000000	1.000000
2	2.000000	2.000000	2.000000	2.000000
3	3.000000	3.000000	3.000000	3.000000
4	4.000000	4.000000	4.000000	4.000000
5	5.000000	5.000000	5.000000	5.000000

```
SELECT ROWID, name, ST_AsText(geom)
FROM test
WHERE ROWID IN (
    SELECT pkid
    FROM idx_test_geom
    WHERE pkid MATCH RTreeIntersects(2, 2, 3, 3));
```

ROWID	name	ST_AsText(geom)
2	b	POINT(2 2)
3	c	POINT(3 3)

Test #1: we'll simply query the R*Tree then performing a trivial query using the R*Tree.
Not at all surprisingly, anything runs as expected.

```
DELETE FROM test WHERE name IN ('a', 'd', 'e');
VACUUM;
SELECT ROWID, name, ST_AsText(geom)
FROM test
WHERE ROWID IN (
    SELECT pkid
    FROM idx_test_geom
    WHERE pkid MATCH RTreeIntersects(2, 2, 3, 3));
```

ROWID	name	ST_AsText(geom)
2	c	POINT(3 3)

Test #2: we'll just DELETE some rows; then we'll VACUUM the DB in order to reclaim any unused storage space.
And finally we'll perform the same identical query using the R*Tree Spatial Index: but this time we'll get a ***wrong result set***. A row is obviously missing.
Why ? really simple to explain … because the R*Tree is now severely ***corrupted***.

```
SELECT ROWID, name, ST_AsText(geom) FROM test;
```

ROWID	name	ST_AsText(geom)
1	b	POINT(2 2)
2	c	POINT(3 3)

```
SELECT pkid, xmin, xmax, ymin, ymax FROM idx_test_geom;
```

pkid	xmin	xmax	ymin	ymax
2	2.000000	2.000000	2.000000	2.000000
3	3.000000	3.000000	3.000000	3.000000

Post Mortem: performing a **VACUUM** compacts any unused space: and consequently **ROWID**s for the **test** table have been reassigned.
But the corresponding R*Tree is still exactly the same as above: **it's a real catastrophe** !!!
Relational correspondences between the main table rows and the R*Tree aren't any longer valid.

Please note well: all this happens simply because the **test** table has no **PRIMARY KEY** defined (a not so common condition).

Please note well (2): there is absolutely no way to prevent such catastrophe: SpatiaLite fully relies upon **triggers** to ensure consistency between the main table and the corresponding R*Tree. But trigger are (quite obviously) disabled while performing a **VACUUM** operation.
Anyway, if the table correctly has a declared **PRIMARY KEY** performing a VACUUM is an absolutely safe and risk-free operation.
This issue simply affects any table without a **PRIMARY KEY**; and in this case too corruption arises only when a **VACUUM** is performed after executing some **DELETE**.

You are now warned.

1. defining any table without any supporting PRIMARY KEY is strongly discouraged
2. and can lead to severe Spatial Index inconsistencies

Checking and Recovering broken R*Trees

```
SELECT CheckSpatialIndex('test', 'geom');
> 0
SELECT RecoverSpatialIndex('test', 'geom');
> 1
SELECT CheckSpatialIndex();
> 1
SELECT ROWID, name, ST_AsText(geom)
FROM test
WHERE ROWID IN (
    SELECT pkid
    FROM idx_test_geom
    WHERE pkid MATCH RTreeIntersects(2, 2, 3, 3));
```

ROWID	name	ST_AsText(geom)
1	b	POINT(2 2)
2	c	POINT(3 3)

The **ChekSpatialIndex()** function will check if the required Spatial Index is valid and fully consistent.
And the **RecoverSpatialIndex()** function will attempt to recover the required Spatial Index into a valid and fully consistent state.

Syntax:

```
SELECT CheckSpatialIndex('test', 'geom');
SELECT CheckSpatialIndex();

SELECT RecoverSpatialIndex('test', 'geom');
SELECT RecoverSpatialIndex('test', 'geom', 1);
SELECT RecoverSpatialIndex();
SELECT RecoverSpatialIndex(1);
```

The `ChekSpatialIndex()` function comes in two flavors:

- you can specify both a **table** and a **geometry-column**: in this case only the corresponding R*Tree (if actually existing) will be checked.
- otherwise you can invoke this function with no arguments: in this case any R*Tree (as defined into **geomety_columns**) will be checked.

The `RecoverSpatialIndex()` function supports more options:

- you can specify both a **table** and a **geometry-column**: in this case only the corresponding R*Tree (if actually existing) will checked first; and only if it is found to be in an inconsistent state will then be actually recovered.
- same as above, but appending a further **TRUE** boolean value: in this case the R*Tree will be unconditionally recovered.
- otherwise you can invoke this function with no arguments: in this case any R*Tree (as defined into **geomety_columns**) will be checked first, and eventually recovered if required.
- and finally you can invoke this function passing a single **TRUE** boolean value; in this case any R*Tree (as defined into **geomety_columns**) will be unconditionally recovered.

About VirtualSpatialIndex

The brand new ***VirtualSpatialIndex*** modyke is intended to simplify R*Tree Spatial Index usage in SQL queries. As you already know, in SQLite / SpatiaLite R*Tree Virtual Tables can be actually used an a very efficient Spatial Index: anyway an explicit **sub-query** is required in order to inquiry the corresponding Spatial Index.

```
SELECT lc1.lc_name AS "Tuscan Local Council",
  c1.county_name AS "Tuscan County",
  lc2.lc_name AS "Neighbour LC",
  c2.county_name AS County,
  r2.region_name AS Region
FROM local_councils AS lc1,
  local_councils AS lc2,
  counties AS c1,
  counties AS c2,
  regions AS r1,
  regions AS r2
WHERE c1.county_id = lc1.county_id
  AND c2.county_id = lc2.county_id
  AND r1.region_id = c1.region_id
  AND r2.region_id = c2.region_id
  AND r1.region_name LIKE 'toscana'
  AND r1.region_id <> r2.region_id
  AND ST_Touches(lc1.geometry, lc2.geometry)
  AND lc2.ROWID IN (
    SELECT pkid
      FROM idx_local_councils_geometry
      WHERE pkid MATCH RTreeIntersects(
        MbrMinX(lc1.geometry),
        MbrMinY(lc1.geometry),
        MbrMaxX(lc1.geometry),
        MbrMaxY(lc1.geometry)))
ORDER BY c1.county_name, lc1.lc_name;
```

[taken from: http://www.gaia-gis.it/spatialite-2.4.0-4/spatialite-cookbook/html/neighbours.html]

The above SQL queries exemplifies how an R*Tree Spatial Index was accessed following the ***classic*** way:

- specifying the R*Tree table was required: **`FROM idx_local_councils_geometry`**
- a *geo-callback* function was required in order to specify the MBR to be searched for: **`WHERE pkid MATCH RtreeIntersects(...)`**
- and in turn this required to explicitly set the MBR corners: **`MbrMinX()`** ...

Using ***VirtualSpatialIndex*** adds some *syntactic sugar* to all this:

```
SELECT lc1.lc_name AS "Tuscan Local Council",
  c1.county_name AS "Tuscan County",
  lc2.lc_name AS "Neighbour LC",
  c2.county_name AS County,
  r2.region_name AS Region
FROM local_councils AS lc1,
  local_councils AS lc2
JOIN counties AS c1
  ON (c1.county_id = lc1.county_id)
JOIN counties AS c2
  ON (c2.county_id = lc2.county_id)
JOIN regions AS r1
  ON (r1.region_id = c1.region_id)
JOIN regions AS r2
  ON (r2.region_id = c2.region_id)
WHERE r1.region_name LIKE 'toscana'
  AND r1.region_id <> r2.region_id
  AND ST_Touches(lc1.geometry, lc2.geometry)
  AND lc2.ROWID IN (
    SELECT ROWID
    FROM SpatialIndex
    WHERE f_table_name = 'local_councils'
      AND search_frame = lc1.geometry)
ORDER BY c1.county_name, lc1.lc_name;
```

This one is exactly the same query as above: but using ***VirtualSpatialIndex*** now we are allowed using a simpler (and clearer) syntax.

Some useful explanations:

- **`SpatialIndex`** is a **Virtual Table** implementing the ***VirtualSpatialIndex*** logic
- every new DB created by ***libspatialite-RC5b*** automatically includes the **`SpatialIndex`** table immediately after creation.
- anyway, you can explicitly create this table on any already existing DB simply typing:
  - **`CREATE VIRTUAL TABLE SpatialIndex USING VirtualSpatialIndex();`**

The **`SpatialIndex`** table contains the following columns:

- **`f_table_name`**, **`f_geometry_column`**:
  - exactly the same as in **`geometry_columns`**; they are used so to identify the required Geometry **`table.column`** and the corresponding R*Tree (if any).
- **`search_mbr`**:
  - corresponding to any arbitrary Geometry: this is used so to set the MBR to be searched within the R*Tree [***Intersects*** mode is assumed anyway].

According to **VirtualSpatialIndex** internal logic, **`f_table_name`**, **`f_geometry_column`** and **`search_mbr`** cannot be queried: if you'll attempt to do such a thing, you'll simply get back **`NULL`** values. You can query a ROWID value instead (corresponding to any matching ROWID withing the R*Tree)
Anyway you can set **`f_table_name`**, **`f_geometry_column`** and **`search_mbr`** values as required and appropriate into the **`WHERE`** clause.

```
SELECT ROWID
FROM SpatialIndex
WHERE f_table_name = 'local_councils'
  AND f_geometry_column = 'geometry'
  AND search_frame = lc1.geometry;
```

this one is a *fully qualified* ***VirtualSpatialIndex*** query; and is processed as follows:

- the *VirtualSpatialIndex* module will first check if an R*Tree Spatial Index is defined for **`local_councils.geometry`**
- if confirmed, then the corresponding R*Tree will be queried using the imposed **`search_frame`**
- and finally any matching **`ROWID`** will be returned into the ResultSet.

```
SELECT ROWID
FROM SpatialIndex
WHERE f_table_name = 'local_councils'
  AND search_frame = lc1.geometry;
```

but in many cases you can set a partial ***VirtualSpatialIndex*** query as well:

- usually each table simply has an unique Geometry column.
- if this assumption is actually satisfied, there is no need at all to specify an explicit value corresponding to **`f_geometry_column`** simply because the *VirtualSpatialIndex* module can easily identify which Geometry column corresponds to the table you've already specified.

Some further examples

```
SELECT lc1.lc_name AS "Local Council",
  c.county_name AS County,
  r.region_name AS Region
FROM local_councils AS lc1
JOIN counties AS c ON (
  c.county_id = lc1.county_id)
JOIN regions AS r ON (
  r.region_id = c.region_id)
LEFT JOIN local_councils AS lc2 ON (
  lc1.lc_id <> lc2.lc_id
  AND NOT ST_Disjoint(lc1.geometry, lc2.geometry)
  AND lc2.ROWID IN (
    SELECT ROWID
    FROM SpatialIndex
    WHERE f_table_name = 'local_councils'
      AND search_frame = lc1.geometry))
GROUP BY lc1.lc_id
HAVING Count(lc2.lc_id) = 0
ORDER BY lc1.lc_name;
```

[please see: http://www.gaia-gis.it/spatialite-2.4.0-4/spatialite-cookbook/html/islands.html]

```
SELECT pp.id AS PopulatedPlaceId,
  pp.name AS PopulatedPlaceName,
  lc.lc_id AS LocalCouncilId,
  lc.lc_name AS LocalCouncilName,
  c.county_name AS County,
  r.region_name AS Region
FROM populated_places AS pp
LEFT JOIN local_councils AS lc
  ON (ST_Contains(lc.geometry,
      Transform(pp.geometry, 23032))
    AND lc.lc_id IN (
      SELECT ROWID
      FROM SpatialIndex
      WHERE f_table_name = 'local_councils'
        AND search_frame = Transform(pp.geometry, 23032)))
LEFT JOIN counties AS c
  ON (c.county_id = lc.county_id)
LEFT JOIN regions AS r
  ON (r.region_id = c.region_id)
ORDER BY 6, 5, 4;
```

[please see: http://www.gaia-gis.it/spatialite-2.4.0-4/spatialite-cookbook/html/pop-places.html]

```
SELECT rw.name AS Railway,
  lc.lc_name AS LocalCouncil,
  c.county_name AS County,
  r.region_name AS Region
FROM railways AS rw
JOIN local_councils AS lc ON (
  ST_Intersects(rw.geometry, lc.geometry)
    AND lc.ROWID IN (
      SELECT ROWID
      FROM SpatialIndex
      WHERE f_table_name = 'local_councils'
        AND search_frame = rw.geometry))
JOIN counties AS c
  ON (c.county_id = lc.county_id)
JOIN regions AS r
  ON (r.region_id = c.region_id)
ORDER BY r.region_name,
  c.county_name,
  lc.lc_name;
```

[please see: http://www.gaia-gis.it/spatialite-2.4.0-4/spatialite-cookbook/html/railways-lc.html]