Optimizing SQL access to really complex polygons

[was: *multipolygon intersection query optimization help*]

A really interesting question was posted few days ago on the SpatiaLite User Group mailing list; here you can read the full thread:
http://groups.google.com/group/spatialite-users/browse_thread/thread/855e37b503e73e0a

This question offers us a really good pretext to examine many general order questions about SpatiaLite and Spatial SQL.
So we'll conduct a thorough analysis of the proposed test case, then attempting to identify an appropriate alternative approach to the original problem allowing to exploit any possible speed optimization.

The original problem (*quickly resumed*)

- querying a MULTIPOLYGON-type table (aka layer) so to identify the actual feature (if any) intercepting an arbitrary POINT (XY)
- this table/layer represents World Borders: each feature (aka row) corresponds to a single Country and contains a Geometry representation (of the MULTIPOLYGON type, SRID=4326: i.e. geographic coordinates are used, expressed ad longitude/latitude).
- the original dataset is the Shapefile offered for public download at the following URL: http://thematicmapping.org/downloads/world_borders.php
- please note: we can classify the test dataset as a very lightweight one: there are simply 246 features (Countries), and the total size is just a few MB. Not at all a *huge* dataset.

original SQL snippet:

```
SELECT NAME
FROM world_borders
WHERE Intersects(geom, GeomFromText('POINT(-135.477995 -83.628368)'))
  AND PK_UID IN (
    SELECT pkid
    FROM idx_world_borders_geom
    WHERE xmin < MbrMinX( GeomFromText('POINT(-135.477995 -83.628368)'))
      AND ymin < MbrMinY( GeomFromText('POINT(-135.477995 -83.628368)'))
      AND xmax > MbrMaxX( GeomFromText('POINT(-135.477995 -83.628368)'))
      AND ymax > MbrMaxY( GeomFromText('POINT(-135.477995 -83.628368)'))
);
```

commented SQL:

- a sub-query is (*correctly*) used in order to access the Spatial Index
- this will quickly identify any feature intercepted by the POINT: anyway, this first step will simply apply a quick (but coarse) filtering, being based on MBRs [aka BBOX]
- so the final step is the one using Intersects() in order to accurately evaluate if the POINT actually intercepts the MULTIPOLYGON roughly identified by its MBR

User's Complaint:

The above SQL query successfully works: anyway measured timings seem to be strongly unsatisfactory (confirmed: SpatiaLite can do much more than this).

Post Mortem

The original approach adopted by the user initially posting this question (and the many suggestions advanced by any further user contributing to the discussion) followed two main strategies:

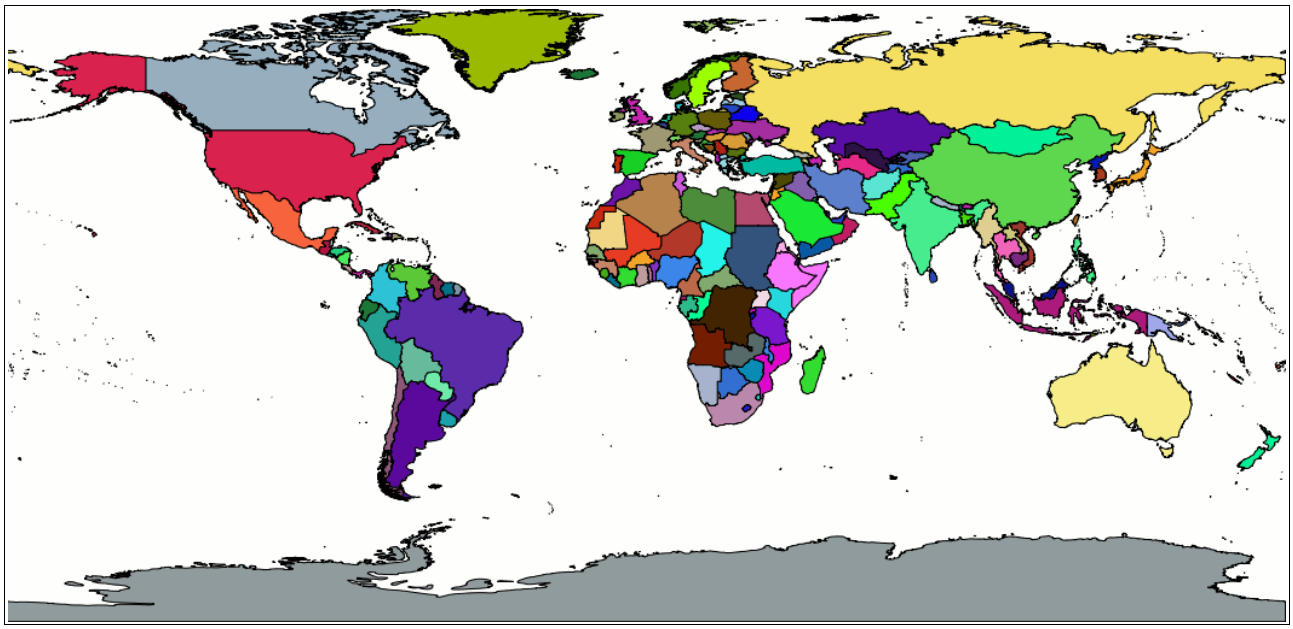
- loading the whole DB-file in-memory, so to speed up any I/O access
- attempting to optimize the SQL code on way or the other.

Anyway, none of them apparently seem to produce any speed benefit at all.

After all, a sound technical explanation exist accounting for all this.

- why the ***in-memory*** approach fails in this case
  - the DB-file is really small: just about ten MB.
  - any operating system is surely able to access a so small-sized file in a fast and efficient way: we can lazy rely on plain, basic I/O optimization supported by the underling file system
  - the in-memory approach can really offer first class optimization: but only for really huge-sized DB-files. i.e. when standard I/O buffering strategies usually adopted by the underling file system are not adequate to support fast random access, due to frequent buffer stalls.
  - but in our test case, the in-memory approach surely is an overkill; and cannot produce any speed benefit.
- why SQL optimization is apparently irrelevant in this case
  - I've performed some initial testing on my own, so to attempt to introduce some useful SQL optimization: but I quickly discovered that following such an approach timings changed in a very limited range: I measured just a few percent points difference between badly written and strongly optimized SQL queries.
  - conclusion: there is an obvious ***bottleneck*** (this bottleneck accounts for more than 90% of consumed processing time; and this fully explains why any possible SQL optimization is apparently irrelevant)
  - this bottleneck is related to input **data**.

Identifying the bottleneck



Just a quick *visual* glance at the dataset: there are few really huge and complex Countries (e.g. Russia, Canada, USA, China, Australia, Brazil ...).
These Countries aren't simply ***wide*** (i.e. covering a big surface): they are impressively ***complex***.
Let us examine e.g. Canada in full detail: this MULTIPOLYGON has about 500 elementary POLYGONs; some of them have more than 100 vertices (and one exceeds 5,000 vertices).
Not at all surprisingly, computing ST_Intersects() on behalf of Canada is a really heavy (and lengthy, time-consuming) operation.

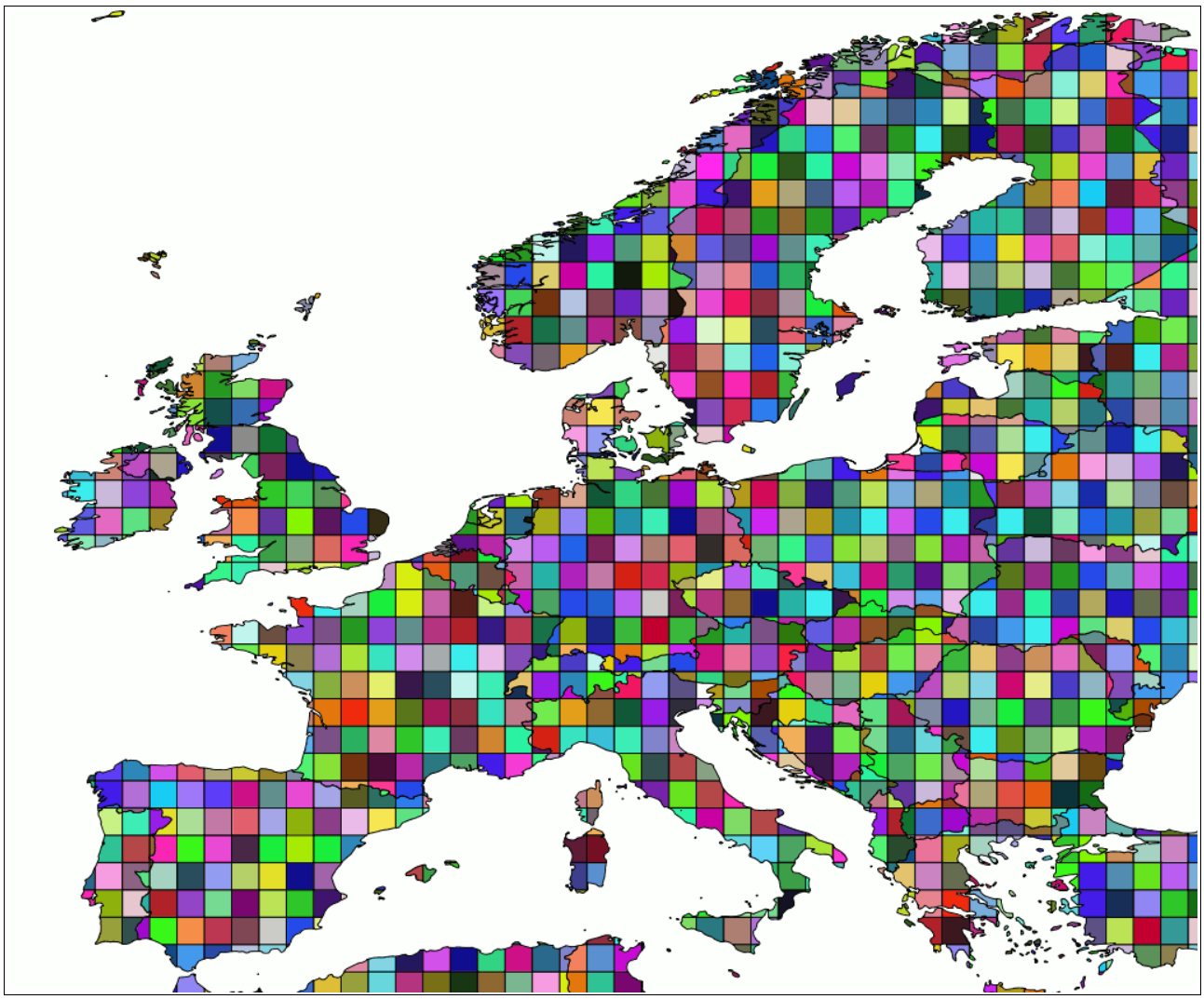
As a rule-of-thumb, we can assume that computing ST_Intersects() will require a time proportional to the actual number of vertices used to represent the polygon's boundary.
Resolving for a triangle (or square) will be a simple and quick affair; but resolving an highly complex polygon [> 1,000 vertices] will require for sure an impressive amount of floating point trigonometric functions to be calculated.
Modern CPUs (FPUs) are surely fast and powerful: but not at all so fast and powerful to perform lots and lots of sophisticated calculations in a negligible time.

Circumventing the bottleneck

All right, now we are fully aware: we have identified the bottleneck accounting for any undesirable slowness and poor performance.
We have to imagine an appropriate method allowing to ***simplify*** our Countries (MULTIPOLYGONs); we absolutely have to reduce someway the number of vertices required to represent elementary polygons.
And we obviously want that such simplification doesn't negatively affects in any way the original accuracy and precision of the WorldBoundary dataset.

So we cannot absolutely apply ST_Simplify():

- this will break the topological consistency of boundaries (*really bad* …)
- this will reduce precision (*awful* ..)



But nothing forbids us to rearrange the original dataset so to split each Country into many (=*simpler*) polygons, still fully preserving unaffected precision and accuracy.
The simplest way I can imagine to achieve such a result is to compute the intersections between each Country and a square grid (each cell of the grid representing 1 square degree).

Just few (*commented*) SQL snippets:

```
CREATE TABLE aux_borders (
  id INTEGER PRIMARY KEY AUTOINCREMENT,
  country_id INTEGER NOT NULL,
  CONSTRAINT fk_country FOREIGN KEY (country_id)
    REFERENCES world_borders (PK_UID));
```

a) we'll create a further auxiliary (helper) table named **aux_border**
b) this table has an explicitly defined relation (Foreign Key) allowing to JOIN the original **world_borders** table, so there isn't any need to duplicate data (*no redundancy, please*).

```
SELECT AddGeometryColumn('aux_borders', 'Geometry', 4326,
                         MULTIPOLYGON', 'XY');
SELECT CreateSpatialIndex('aux_borders', 'Geometry');
```

Then we'll add a MULTIPOLYGON geometry

```
INSERT INTO aux_borders (id, country_id, Geometry)
SELECT NULL, PK_UID,
   CastToMultiPolygon(ST_Intersection(Geometry,
     BuildMbr(10, 42, 11, 43, 4326))) AS geom
FROM world_borders
WHERE PK_UID IN (
   SELECT pkid FROM idx_world_borders_geometry
   WHERE xmin <= 11  AND ymin <= 43 AND
         xmax >= 10 AND ymax >= 42
) AND geom IS NOT NULL;
```

And finally we can populate this helper table using the above SQL statement:

- we'll use an INSERT … SELECT …
- ST_Intersection() will compute each *simple*-MULTIPOLYGON to be inserted into the helper table
- we'll use CastToMultiPolygon() for two good reasons:
  - to be absolutely sure that any sub-Country item will be of the MULTIPOLYGON type
  - and to suppress any *odd intersection* (POINT, LINESTRING … we must expect to get some result of this type, when the 1 square degree boundary exactly touches the Country border). but using CastToMultiPolygon() such *odd result* will become NULLs.
- we'll obviously use the Spatial Index so to speed up operations as much as possible
- and finally we'll set a `geom IS NOT NULL` clause so to suppress any useless row.

We'll obviously iterate the above SQL statement for each 1 square degree tile, ranging from latitude -90 to +90, and from longitude -180 to +180.

I'll anticipate the final result: at the end of the process, we'll have our WorldBorders represented by about **28,000** *smallest*, *simplest* fragments: compare this figure to the original 246 Countries !!!

And we'll have absolutely no information loss: the full original accuracy and precision are completely preserved.
That's not all: we can easily and efficiently relationally JOIN the **aux_borders** and the **world_borders** tables, because a Primary / Foreign Key explicit relation is defined.

the **writer.c** sample

as a didactic tool, you can usefully study the **writer.c** sample code implementing the above SQL based approach.
http://www.gaia-gis.it/spatialite-3.0.0-BETA1/world-borders-sample-code.zip

Please note: running the **writer** program will take about *a full hour*; a further confirmation that processing huge and complex polygons requires a lot of time (even if using powerful CPUs)

Final Test: reality check

All right, theory is good, but reality is by far better.

It's now time to check if this strategy (splitting the original dataset into many and many simplest fragments) does actually pays in pure speed terms.

I've simply rearranged the test code supplied by the original user who started this interesting thread. Now the (optimized) SQL query looks like:

```
SELECT w.NAME
FROM world_borders AS w, aux_borders AS a
WHERE a.country_id = w.PK_UID
    AND ST_Intersects(a.geometry, MakePoint(?, ?, 4326))
    AND a.id IN (
        SELECT pkid
        FROM idx_aux_borders_geometry
        WHERE xmin <= ?  AND ymin <= ?
          AND xmax >= ? AND ymax >= ?);
```

- we are using the *advanced* C API [prepared Statement]
- now we have a relational JOIN, so to get Geometry from the **aux_borders** helper table, but still retrieving any other data from the original **world_borders** table.
- please note: more or less, this is the same of the original query (the one running really slowly).
- but now we aren't any longer attempting to *filter* few highly complex polygons: this time we are filtering many simple polygons. And finally the SQL engine can rocket at its best.

Using purposely rearranged data [*aux_borders*], this query now runs about 500 times faster: really impressive !!!

Lesson to learn: each time you are attempting to get impressively high performances, simply focusing your attention on SQL isn't enough. Carefully examine the fine structure of your input data, so to identify any possible obnoxious bottleneck. Rearranging input data in a most convenient way may often be an absolutely relevant key point for success.

the **reader.c** sample

as a further didactic tool, you can usefully study the **reader.c** sample code testing the helper table actual effectiveness.
http://www.gaia-gis.it/spatialite-3.0.0-BETA1/world-borders-sample-code.zip