Moving Geo Databases to Smart Phones – An Approach for Offline Location-based Applications

Jörg Roth

Department of Computer Science
Univ. of Applied Sciences Nuremberg
Kesslerplatz 12, 90489 Nuremberg, Germany
Joerg.Roth@Ohm-Hochschule.de

Abstract: Mobile location-based services usually access large geo databases via a wireless network. Sometimes it is useful to store a certain amount of geo objects on the mobile device. In this paper we present an approach to store geo data in an embedded SQL database on smart phones. Besides the storage of geometries in SQL tables we have to provide an efficient geometric indexing mechanism. We used the *Extended Split Index* that successfully was used for large server geo databases. Due to the special characteristics of embedded smart phone databases we have to introduce an extension, the *In-Memory Index*. Our paper concludes with performance considerations.

1 Introduction

Applications that take into account a mobile user's current position, so-called *location-based* applications, are getting more and more popular. There exist powerful platforms such as Google Maps that support a developer to present maps and query for nearby objects. Currently, the typical way to access geo data is to use a central service. As geo objects produce a huge amount of data (e.g. 10 mio. records for Germany), this data cannot be easily stored on mobile devices. There exist scenarios, however, where it is suitable to store at least a subset of geo data on the mobile device and work offline. A wireless communication link could be too slow or too cost-intensive, or even not available. We could think about trekking applications that should work in weakly connected natural environments, services for researchers at archaeological sites or applications that support emergency services in disaster areas.

Developers of corresponding applications should rely on established storage mechanisms to save development costs. In this paper we present an approach to store and retrieve geo data with the help of an embedded SQL database widely available on current smart phones. As a main challenge, geometric indexing and geometric queries are supported.

2 Related Work

In this paper we only consider vector data, i.e. an object is represented by its polygonal geometry and further properties such as name and classification. This data type is used if we want to identify objects that reside at a specific location or that are inside a certain given area. We can also use vector data to draw a map. The main contribution of a spatial database is a fast processing of geometric queries with the help of a spatial index. Common spatial indexes are based on trees such as Quadtrees [FB74] or variations of R-Trees [Gu84]. They first approximate the shape of an object by a bounding box that is inserted into a tree structure. A query uses the tree to identify a (hopefully small) set of candidates that have to undergo exact geometric checks. Tree structures have a major drawback: multiple accesses are required to run through the tree to find appropriated geometries. An approach that checks a geometric property per object is more suitable for slow smart phone databases.

Typical spatial databases are *Oracle Spatial* [Mu09], *PostgreSQL/PostGIS* [Ne10] or *MySQL Spatial Extensions* [Sun11]. The interface to the application developer is SQL, but to formulate spatial queries different SQL extensions are used. The Open Geospatial Consortium (OGC) proposed some standards to access spatial extensions. The so-called *Simple Features* provide a framework for geometries that include geometry types such as the *Line String* or *Multi Polygon* [He05, He06].

Existing spatial databases use different SQL expressions to create geometric data or formulate geometric queries. As a main disadvantage, these constructions cannot be transferred to smart phones. Even though most smart phones platforms support embedded SQL databases, usually *SQLite* [Kr10], spatial extensions are currently not available as built-in components.

As an exception, *SQL Anywhere* provides an additional component that is able to access spatial data on smart phones [Syb07]. It also comes along with its SQL extension to define geometry data types and queries. The *UltraLite* version runs one the mobile platforms Windows Mobile, iPhone and Blackberry. The major focus of SQL Anywhere is to run enterprise applications that synchronize their data with mobile devices. It thus is not suitable for small and inexpensive Apps, that turn out to be the major basis for future location-based applications.

3 The Extended Split Index

The main goal of our approach is to use a standard SQL database to store geo objects and to execute geometric queries. This is an important prerequisite to use this approach on smart phones as they locally only support a non-spatial standard database. Figure 1 shows the dataflow to execute a database query.

The application can perform standard queries in a traditional way using the SQL interface. Only if an application is interested in geometries, our components are involved:

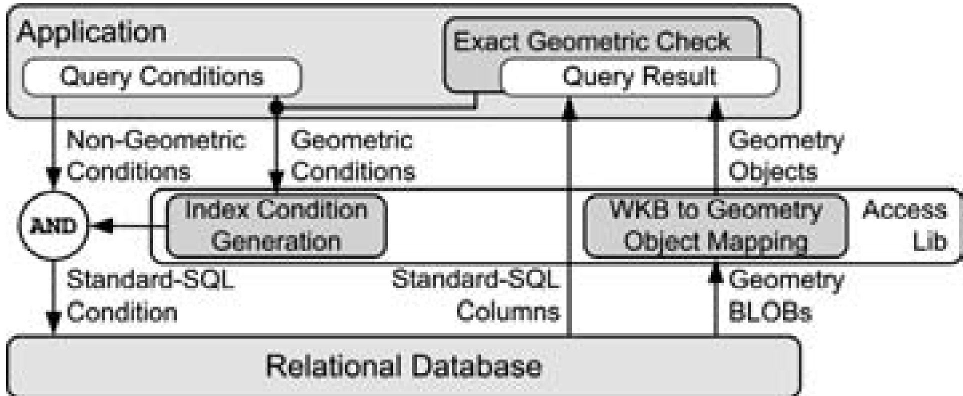

Figure 1: Data flow for geometric queries

- The *Index Condition Generation* transforms a geometric condition such as "inside a given polygon" into an SQL index condition according to our *Extended Split Index* approach (see section 3.1).

- The *Well-known Binary to Geometry Object Mapping* reads geometries from the database. As standard SQL does not support geometric values, they are stored as Binary Large Object (BLOP). The established format *Well-Known Binary* (WKB) is used [He05]. This format can easily be transformed into geometry objects in the application's objects space.

- The *Exact Geometric Check* is required as the SQL index condition produces a superset of results. Only rectangle conditions ("inside a given rectangle") can fully be processed inside the SQL query as complex polygonal conditions would require the database to fully interpret the set of geometries.

The components inside the *Access Lib* are very small (19 kB binary) and linked to the application. Currently, we support applications developed in Java, but other programming languages are conceivable. For exact geometric checks, we use the *Java Topology Suite* (*JTS*) [Aq03] that fully supports the OGC Simple Features.

3.1 Mapping of Geometries to 1-dimensional Indexes

Our spatial index, called *the Extended Split Index* [Ro09] has the following properties:

- An index value is stored in the geo object's data row and only depends on its geometry. This is an important difference to spatial indexes that base on trees, where a certain index value is influenced by multiple geo objects.

- Every geometric query is executed by a *single* standard SQL query.

- The spatial index is mapped to a standard SQL column index (i.e. a non-spatial index). We heavily rely on the efficiency of Integer indexes inside the database to speed up geometric queries.

- The set of candidates retrieved by the spatial index is sufficiently small.

Our index covers two dimensional finite areas ($x_0 \ldots x_{max}$, $y_0 \ldots y_{max}$). Three dimensions are conceivable, but not discussed in this paper. To, e.g., store world-wide geographic data, we would use latitude/longitude coordinates. Figure 2 illustrates the concept.

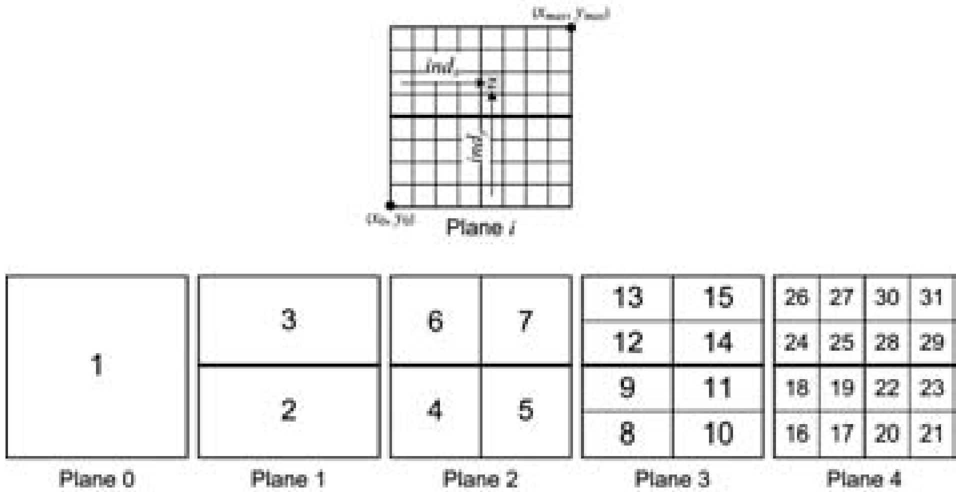

Figure 2: Mapping of numbers to tiles

Planes with numbers $0 \ldots p$-1 cover the area with different resolutions. Planes are divided into *tiles* using either horizontal or vertical separators. The basic idea is to uniquely map integer numbers z to tiles of the covered area. This mapping has some similarities to the z-curve [TH81], but where the z-curve maps a single geometry to a single tile, we map a single geometry to all intersecting tiles of different planes.

For a tile number z the corresponding plane has a number $i = \log_2(floor_2(z))$ where $floor_2(z)$ denotes the largest power of 2 not greater than z; the lower left tile of this plane has the number $z_0 = floor_2(z)$. To map locations we need an auxiliary function *interleave*: for a number n with binary representation $(n_m n_{m-1} \ldots n_3 n_2 n_1 n_0)_2$ we get $interleave_0(n) = (n_{m-1} \ldots n_2 n_0)_2$ and $interleave_1(n) = (n_m \ldots n_3 n_1)_2$. A tile location is

$$ind_x(z) = interleave_{i \bmod 2}\left(z - floor_2(z)\right),$$
$$ind_y(z) = interleave_{(i+1) \bmod 2}\left(z - floor_2(z)\right) \quad (1)$$

As point-like objects have no size, their tile numbers always reside on plane p-1. To map a coordinate x, y on the tile number z on a plane i we simply have to perform a mapping to ind_x, ind_y and reverse formula (1). For a non-point geometry we first create the bounding rectangle of the geometry and map two opposite corners of the bounding rectangle to individual z-values z_0, z_1. The tile number z of the geometry is the *longest common binary prefix* of z_0 and z_1.

An important advantage: for a given tile index z we get all bigger tiles that enclose this tile by the simple set

$$\{\lfloor z/2 \rfloor, \lfloor z/4 \rfloor, \lfloor z/8 \rfloor, ...1\} \tag{2}$$

where $\lfloor\rfloor$ denotes *rounding off*. On the other hand, smaller tiles inside a given tile z have the numbers

$$\{2 \cdot z, 2 \cdot z + 1\} \cup \{4 \cdot z, ...4 \cdot z + 3\} \cup \{8 \cdot z, ...8 \cdot z + 7\} \cup ... \tag{3}$$

These sets are simple to compute and can be expressed by one-dimensional interval conditions *inside* a single SQL query. We want, e.g., to find all cities in Bavaria with less than 100 000 inhabitants. Let's assume the tile number of the border of Bavaria is z=55 and we have 8 planes (i.e. tile numbers 1…255). Let **IND** be the name of the index column that holds the z values. The resulting SQL query then is

```
SELECT * FROM CITIES WHERE INHABITANTS<100000 AND
    (IND=55 OR IND BETWEEN 110 AND 111 OR IND BETWEEN 220 AND 223)
```

The first line contains the non-geometric condition; the second line the conditions according to formula (3). The second line is automatically created by the component *Index Condition Generation*. To speed up the query processing, **IND** is an index column inside the database.

Depending on the geometric query, the index expression is different. The three most important queries are:

- *WITHIN*: retrieve all geo objects that are completely inside a given geometry. For this query, z and the numbers of embedded tiles according to formula (3) are used.

- *INTERSECTS*: retrieve all geo objects that intersect with a given geometry. Here, we use z and tile numbers according to formulas (2) *and* (3).

- *CONTAINS*: retrieve all geo objects that fully enclose the given geometry. Only z and tiles according to formula (2) are used.

Again note that the index expression provides a superset of hits. Exact conditions, e.g. if a result is completely inside a given polygon, have to be checked later.

3.2 Shifted Index and Performance Considerations

Until now we still have a problem: even small geometries have a small tile number, if they intersect separators with low plane indexes. E.g., all objects intersecting the equator have z=1, even very small objects. Such objects lead to a constant amount of candidates for all queries as they always have to undergo the exact geometric check.

As a solution we use two *shifted* indexes, each of it computes its own z number. One index uses the given coordinates, where the second index adds a fix coordinate offset. The specific offset ensures that not both indexes simultaneously generate low z numbers. For queries, we create index conditions for both indexes, combined by **`AND`**, thus the higher z number has the largest impact to reduce the candidate set.

To find an appropriate offset is not a trivial task. In [Ro09] we made an in-depth analysis to find out optimal values. The best results generate offsets of 1/6 and 1/12 of the area size. For real data, the offset of 1/6 increases the average maximum plane number by 2 and effectively avoids the problem of low tile numbers.

Our geo data is based on *Open Street Map* [OSM11]. A complex import chain [Ro10] solves some problems of the original data that have certain geometric and topological shortcomings and only a weak classification.

Our database server (Windows Server 2008) with 10 mio. objects has a maximum query time of less than 200 ms even for large result sets. For random queries, 97% of the hits given by the SQL statement fulfill the geometry conditions, thus only 3% are false positive hits [Ro10] detected by the exact geometric check.

We now want to transfer this approach to smart phone databases.

4 The Extended Split Index on Smart Phones

To transfer the idea of the Extended Split Index to a smart phone platform seemed to be easy at first view. As target platform we selected Android. As the entire index mechanism is coded in Java, it could easily be transferred into an Android Java project. However, there were important differences:

- The database interface JDBC is not available on the Android platform. Instead, Android uses an own interface that is tailored to the SQLite database system.
- We cannot use JTS as geometry library and have to use a much simpler library that does not support the huge variety of geometric types and functions. Even though all required geometries can be stored and retrieved, we can only use bounding rectangles as query geometry. For most location-based applications this is sufficient.

From the server database we can select a geometric region (e.g. a polygonal area that encloses a city) and a list of object types (e.g. only roads, only points of interests). All matching objects are then exported to an embedded database file that can be stored on the Android's SD card. The database file has exactly the same table structure as the original server table; especially all geometries (points, line strings and polygons) are supported and stored as Well-known Binary.

4.1 A First Approach

To get an impression about the efficiency of the Extended Split Index on a real smart phone we conducted a number of performance tests on a Milestone (550 MHz ARM) with Android 2.1. We used three sizes of database exported from all geo objects of a larger town in Germany: 1. a district of the town (approx. 8000 objects), 2. the inner part of the town (approx. 25000 objects) and 3. the whole town including suburbs (80000 objects). The exported databases contain all geometric types (point, line string and polygon) and all available object types. Note that a typical application would limit the object types to the application-specific domain, thus the geo databases would be much smaller in reality.

We created 20000 random queries. The query geometry was a random rectangle with a size between 250m x 250m and 500m x 500m. The query selects all objects in the database that are contained in the query rectangle.

Typical database queries contain a preparation phase (e.g. to define the SELECT statement) and a phase when the application iterates through the result set with a database cursor. We measured both times. Table 1 shows the results.

Table 1: Average query time (in ms) for the first approach

Database	District in town (8 000 objects)	Inner part of the town (25 000 objects)	Whole town (83 000 objects)
Preparation time	12.1	11.7	34.5
Cursor time	177.7	1048.5	10779.9
Time total	189.8	1060.2	10814.4

This result was disappointing. Whereas for the small database, the total query time was acceptable, for the medium and larger database we got unacceptably long query times.

To explain this drawback, we have to consider the difference between the database systems. Whereas on the server, we have a traditional database (PostGres), on the Android device we only have an embedded database. This means: the entire database processing is done by the application that mainly accesses a single file. The main effects are: 1. very limited ability to cache indexing structures and 2. reading the results always means to iterate through a file. Even though the file system allows jumping to specific blocks of the file, additional parts of the file are transferred to the memory and utilize bandwidth.

4.2 Moving the Index into the Memory

The results above were discouraging. The Extended Split Index heavily bases on the efficiency of Integer index columns supported by the SQL database. As the embedded database has certain limitations regarding index columns, we got no significant speed up of geometric queries. However, we can save the idea if we move the indexing structure from the database into the application memory. We call this the *In-Memory Index* (*IMI*), presented in figure 3.

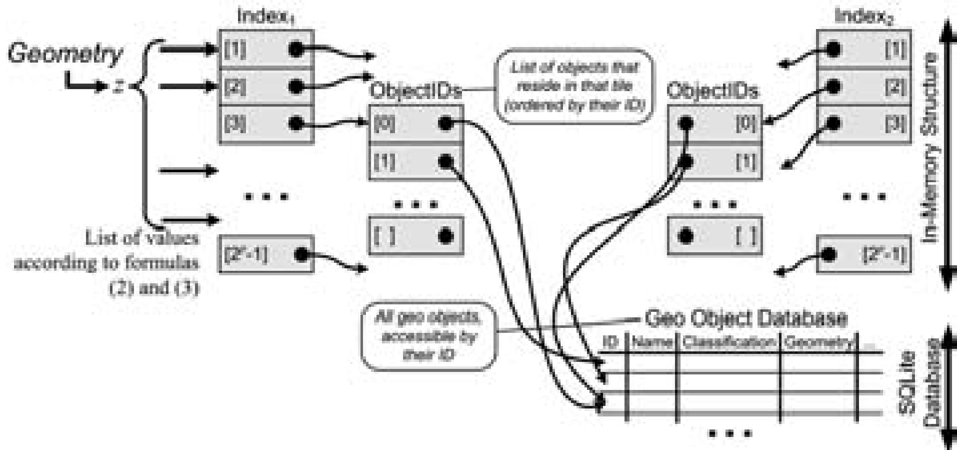

Figure 3: Structure of the In-Memory Index

The processing steps to execute a query are:

- We map the query geometry to tile numbers according to sections 3.1 and 3.2 (e.g. the number *z* and all numbers of embedded tiles, for index and shifted index).
- The list of tile numbers is used as array indexes of **`Index`**$_1$ and (shifted) **`Index`**$_2$. For each of the two we get multiple arrays **`ObjectIDs`** that are joined to two separate lists.
- We intersect these two lists. As a result we get a list of object IDs that can be used to query the database. This SQL query directly uses the primary object key (using **`SELECT`** … **`WHERE ID IN (ID`**$_1$**`,…, ID`**$_x$**`)`**)

One could argue that for the last step we again rely on an Integer index column. But in contrast to the statement in section 3.1, this statement directly accesses IDs without **`BETWEEN`**… and **`OR`**…. This significantly speeds up the query. The entire IMI can be precomputed during the database export on a host system and thus the creation does not affect the smart phone.

Some words about the runtime complexity to compute the list of IDs. Let z be the geometry's tile number that resides at plane i; we have 2^i tiles on a plane. Thus we have $floor_2(z)$ tiles on the plane of tile z. Let further denote n the total number of geo objects. We now assume a *WITHIN* query.

We can statistically show that for typical geo data, the number of objects are nearly equally distributed across the planes (*not* tiles). Thus the list **`ObjectIDs`** has an average size of

$$g(z) = \frac{n}{p \cdot floor_2(z)} \quad (4)$$

We have to additionally consider the objects inside embedded tiles. As the embedded tiles have approximately ½, ¼ etc. objects compared to tile z, we can estimate the total number of objects retrieved by one index by

$$g^*(z) = 2\frac{n}{p \cdot floor_2(z)} \quad (5)$$

The most time consuming part is to intersect the two object lists from shifted indexes. As the object IDs are only partly sorted, this requires $O((g^*_1+g^*_2)\cdot \log(g^*_1+g^*_2))$ steps.

Larger values of p lead to quicker computations, but also to larger arrays **`Index1`** and **`Index2`**. Experiments show that p=16 is an adequate value that balances speed and memory usage. The value of z only depends on the actual query geometry thus cannot be estimated beforehand. For a specific z and fix p the runtime complexity therefore is $O(n \cdot \log n)$.

If we assume 4 bytes for a pointer, the IMI requires $(2^p-1)\cdot 4$ bytes for references to the arrays **`ObjectIDs`**. As each geo object ID only occurs in a single array **`ObjectIDs`**, we have a total of n entries. As we have two shifted indexes, the IMI requires a total of $8\cdot(n+2^p-1)$ bytes. For, e.g., our town database using 16 planes this results in 1.1 Mbytes. The Android's typical heap size for Apps is 24 Mbytes, thus the required amount of memory for the IMI is acceptable.

4.3 Performance Measurements

We used the performance measurements above (section 4.1) to test our IMI approach. Table 2 shows the results.

Table 2: Average query time (in ms) using the In-Memory Index

Database	District in town (8 000 objects)	Inner part of the town (25 000 objects)	Whole town (83 000 objects)
Preparation time	227.5	304.4	580.3
Cursor time	92.9	90.0	238.8
Time total	320.4	394.4	819.0

The IMI significantly speeds up the cursor time but now requires a longer preparation time. For the small database the total time is even longer compared to the pure SQL query. But for larger databases, the IMI leads to acceptable average times.

In a second step we analyzed how the query time is affected by the result size. For query tests described above, 90% of the results have a size with less than 20 hits. Figure 4 shows, how the total time depends on the result size.

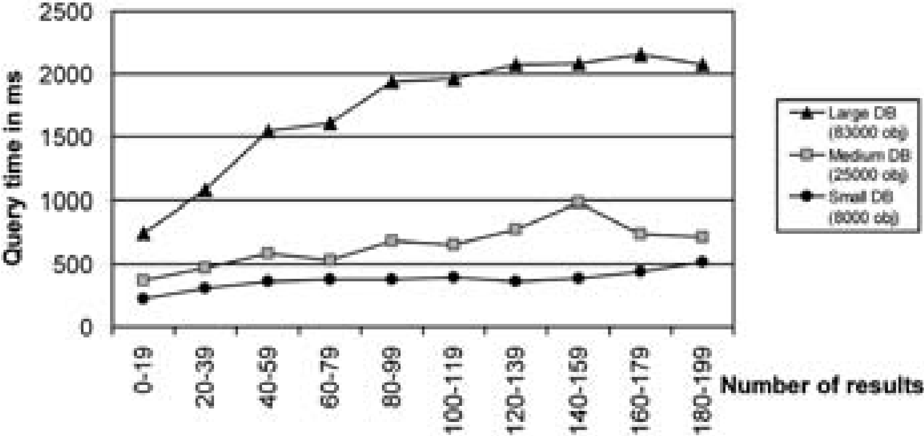

Figure 4: Query times (in ms) using the In-Memory Index

Not surprisingly, the query time increases for larger results. For the large database, the time goes up to approx. 2s that can be considered as critical for some applications. We still have to keep in mind that for a specific application the corresponding database does not contain all geo objects of, e.g. a town, but only a small subset. Thus the databases for certain applications are usually far smaller than in our tests.

5 Summary and Future Work

In this paper we presented an approach that allows a smart phone application to locally access geo data without the need to query across the network. This can be useful, if network costs are too high or a mobile network is not available. To speed up geometric queries we used the Extended Split Index approach that already successfully works in our huge server geo database.

If we transfer an approach to a smart phone we have to scale down our expectations. We showed that a naïve re-implementation of the Extended Split Index on the smart phone platform has important performance drawbacks due to a different database implementation. We thus presented an additional approach that uses an In-Memory Index. Even though the query execution time is far behind the server performance, for certain application scenarios it is acceptable. As a major benefit of our approach, an application developer can use established access mechanisms (especially SQL) to access and query data. This saves development costs.

Bibliography

[Aq03] Aquino, J.: JTS Topology Suite. Technical Specifications, Vivid Solutions, 2003.

[FB74] Finkel, R.; Bentley; J.L.: Quad Trees: A Data Structure for Retrieval on Composite Keys. Acta Informatica 4 (1): 1974; pp. 1–9.

[Gu84] Guttman, A.: R-Trees: A Dynamic Index Structure for Spatial Searching. Proc. of the 1984 ACM SIGMOD International Conference on Management of Data, Boston, Massachusetts, June 1984; pp. 47-57.

[He05] Herring, J. (ed.): OpenGIS® Implementation Specification for Geographic information - Simple feature access - Part 1: Common architecture, OGC, 2005.

[He06] Herring, J. (ed.): OpenGIS Implementation Specification for Geographic information - Simple feature access - Part 2: SQL option, OGC, 2006.

[Kr10] Kreibich, J. A.: Using SQLite O'Reilly, 2010.

[Mu09] Murray, C.: Oracle Spatial Developer's Guide. Oracle, June 2009.

[Ne10] Neufeld, K.: PostGIS Manual, 2010.

[OSM11] OpenStreetMap, excerpts for Europe, http://download.geofabrik.de/osm/, 2011.

[Ro09] Roth, J.: The Extended Split Index to Efficiently Store and Retrieve Spatial Data With Standard Databases, IADIS Intern. Conf. Applied Computing 2009, Rome, Nov. 19-21 2009; pp. 85-92.

[Ro10] Roth, J.: Übernahme von Geodatenbeständen aus Open Street Map und Bereitstellung einer effizienten Zugriffsmöglichkeit für ortsbezogene Dienste Praxis der Informationsverarbeitung und Kommunikation (PIK), Vol. 13, Heft 4, 2010; pp. 268-277.

[Sun11] Sun: MySQL 5.5 Reference Manual. Sun Microsystems, 2011.

[Syb07] Sybase iAnywhere: UltraLite Database Management Reference, March 2007.

[TH81] Tropf, H.; Herzog, H.: Multidimensional Range Search in Dynamically Balanced Trees. Applied Informatics, 2/1981, Vieweg, Germany, 1981; pp. 71–77.