THE EXTENDED SPLIT INDEX TO EFFICIENTLY STORE AND RETRIEVE SPATIAL DATA WITH STANDARD DATABASES

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

ABSTRACT

Geometric and geographic data have special demands on the database query mechanism. To store and retrieve huge amounts of geo data, special spatial databases thus offer geometric column types and spatial indexes. Even though spatial databases are getting more and more available, spatial operations are not standardized, thus applications cannot easily switch between different databases. Moreover, spatial databases are not available for all platforms. Mobile device platforms, e.g., usually only support standard databases without any spatial extensions. Our approach is thus based on relational standard databases and we introduce a *spatial add-on* that translates geometric queries to standard SQL. It provides a new spatial index, the Extended Split Index, which is optimized for the add-on. It especially avoids any index reorganization, makes use of one-dimensional non-spatial indexes available in SQL databases and heavily reduces the number of candidates that have to undergo further geometric checks. We demonstrate the strength of our approach with a performance evaluation based on more than 200 000 geo objects.

KEYWORDS

Spatial database, spatial index, geo data

1. INTRODUCTION

Spatial databases are used to store huge amounts of geographic or geometric data. Besides typical tasks of databases they support geometric data types and operations. Geometric objects such as polygons can in principle be stored in traditional table columns, but then typical queries (e.g. 'find polygons that enclose a specific point' or 'find polygons that overlap with a given other polygon') would be very difficult to execute. In particular, standard index columns fail to optimize geometric queries. As a solution, spatial databases provide special spatial indexes that are optimized to speed up geometric queries.

Spatial operations are usually integrated *inside* a database management system. To deal with geometric objects both as atomic attributes (i.e. column values) as well as structured objects, geometries are modeled deeply inside the database engine. In some cases however, the operation of spatial databases is not possible. E.g., small computers or mobile end-user devices only support relational standard databases without any spatial data types. A further issue related to spatial databases is currently the lack of standardized interfaces: whereas standard SQL queries are standardized, every spatial database has its own syntax to model geometries and to formulate spatial queries. Moreover, the set of geometric operations is different between the spatial databases. Thus, an application developer has to decide very early which spatial database to use and cannot easily switch later.

Our approach goes into another direction: we provide a geospatial add-on based on a non-spatial relational database. The add-on translates geometric queries to standard SQL. As the add-on does not have any internal access to index structures, excessive index reorganizations are inappropriate. Thus we provide a new spatial index called the *Extended Split Index* that is optimized for our intended usage scenario.

2. RELATED WORK

Typical spatial databases are Oracle Spatial (Murray, 2008), PostgreSQL/PostGIS (Neufeld, 2009) or MySQL Spatial Extensions (Sun, 2009). 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 classes such as the LineString or MultiPolygon (Herring, 2005). A further document maps geometric classes to SQL expressions (Herring, 2006). A similar approach is ISO SQL/MM Spatial that also is based on the OGC Simple Features (Stolze, 2003).

Existing spatial databases, however, are different how they offer geometric properties; especially the interfaces to formulate geometric attributes differ. E.g., PostGIS uses the function **`AddGeometryColumn`** to create a geometric attribute; values are defined by **`GeometryFromText()`**. In MySQL Spatial Extensions we have the column type **`GEOMETRY`** that is defined by **`CREATE TABLE`**; values are defined by **`GeomFromText()`**. Oracle Spatial uses the column type **`SDO_GEOMETRY`** that is assigned by **`SDO_GEOMETRY(…SDO_ORDINATE_ARRAY())`**. Further differences also affect the interface, thus it is difficult the change the underlying spatial database after an application or service once has been developed.

Besides the storage of geometries, the main contribution of a spatial database is the fast processing of geometric queries with the help of a spatial index. Common spatial indexes are based on trees such as Quadtrees (Finkel and Bentley, 1974) or variations of R-Trees (Guttman, 1984). They first approximate the shape of an object by a bounding box that is inserted into a tree structure. A query goes down through the tree until an appropriate tree node is found. The corresponding bounding box then can be used to identify a (hopefully small) set of candidates that have to undergo further geometric checks.

Usually, a single spatial query leads to multiple tree node queries. Moreover, modifications or insertions of geometries may result in excessive tree reorganizations. This is not suitable for an add-on that has to consider the index as a usual table column without any internal access and which has to retrieve and modify index values through the SQL interface. In this paper we thus suggest a new spatial index that only changes the index value of the respective data row within a single update.

3. THE ADD-ON APPROACH

To support geometric queries on standard SQL databases we developed a small add-on (80 kB binary) that is linked to the application or service. Currently, we support applications developed in Java, but other programming languages are conceivable. Besides the formulation of queries, the add-on provides an automatic object mapping, thus the application developer can easily access geometries by objects inside the application's object space. We use the Java Topology Suite JTS (Aquino, 2003) as geometric engine – it fully supports the OGC Simple Features.

3.1 The Extended Split Index

Our new spatial index, called the *Extended Split Index* has the following properties:

- Only index values of modified geometries are changed for every conceivable modification.
- Every geometric query is executed by a single standard SQL query.
- The spatial index is mapped to a single standard SQL index column (i.e. a non-spatial index).
- 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.

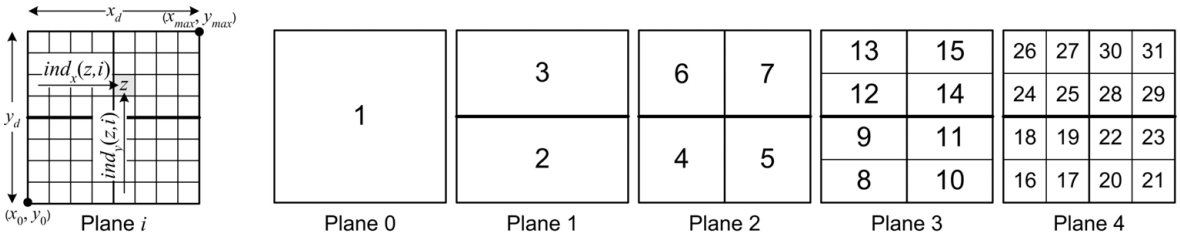

Fig. 1. Mapping of numbers to tiles

Fig. 1 illustrates the concept: indexed *planes* cover the geometric area with different resolutions. Planes are divided into *tiles* using either horizontal or vertical separator lines. The number of tiles per plane is always a power of 2.

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 (Tropf and Herzog, 1981), 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 i has an index $\lfloor \log_2(z) \rfloor$ (here, $\lfloor\rfloor$ denotes *rounding off*); the lower left tile of this plane has the index number $z_0=2^i$. To get the tile coordinates ($ind_x(z, i)$, $ind_y(z, i)$) 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 $interleave_1(n)=(n_m \ldots n_3 n_1)_2$ i.e. it returns even or odd bits of a number. The coordinates then are

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

where the tiles have a size

$$rx_d(i) = \frac{x_d}{2^{\lfloor i/2 \rfloor}},\ ry_d(i) = \frac{y_d}{2^{\lfloor (i+1)/2 \rfloor}} \tag{2}$$

The rectangle related to a tile is

$$\begin{gathered}\left(x_0 + ind_x(z,i) \cdot rx_d(i), y_0 + ind_y(z,i) \cdot ry_d(i)\right) \\ \left(x_0 + \left(ind_x(z,i)+1\right) \cdot ry_d(i), y_0 + \left(ind_y(z,i)+1\right) \cdot ry_d(i)\right)\end{gathered} \tag{3}$$

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

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

On the other hand, smaller tiles inside a given tile z have the numbers

$$\{2 \cdot z, 2 \cdot z + 1\} \cup \{4 \cdot z, \ldots 4 \cdot z + 3\} \cup \{8 \cdot z, \ldots 8 \cdot z + 7\} \cup \ldots \tag{5}$$

These sets are extremely simple to compute and it can be expressed by one-dimensional interval conditions *inside* a single SQL query.

For our intended usage, we also require the inverse mapping. For a coordinate (p_x, p_y) of ($x_0 \ldots x_{max}$, $y_0 \ldots y_{max}$) we get the tile index z on plane i as follows:

$$z(p_x, p_y, i) = 2^i + \left\lfloor \frac{(p_x - x_0)}{rx_d(i)} \right\rfloor + \left\lfloor \frac{(p_y - y_0)}{ry_d(i)} \right\rfloor \cdot 2^{\lfloor i/2 \rfloor} \tag{6}$$

For a given geometry, we want to compute the smallest tile that fully encloses this geometry. If we formulate a query, we compute embedded and surrounded tiles for the query geometry (equations 4, 5) and retrieve the set of candidates with the help of a one-dimensional index of z values.

Because a point can be embedded into arbitrarily small tiles, we have to specify a maximum plane number i_{max} that depends on the supported integer type. The relation between i_{max} and the maximum tile index z_{max} is

$$i_{\max} = \lfloor \log_2 (z_{\max} + 1) \rfloor - 1 \tag{7}$$

For the data type **INT8**, we, e.g., get i_{max}=62 and 2147483648 tiles in x-direction. Covering the Earth's surface, the maximum tile size then is 1.86 cm. This is a sufficient resolution for usual location-based applications.

For non-point geometries we map the lower-left and upper-right corners (p_{x0}, p_{y0}, $p_{x\max}$, $p_{y\max}$) of the bounding box to their tile indexes on plane i_{max}. If these index numbers are different (which is the probable case), we compute the largest common tile index z_{rect} as follows

$$\begin{aligned} & z_{rect}(p_{x0}, p_{y0}, p_{x\max}, p_{y\max}) = \\ & \max\left(z_{ij} \middle| z_{ij} = z(p_{x0}, p_{y0}, i) = z(p_{x\max}, p_{y\max}, j), i, j \in \{0, \ldots i_{\max}\}\right) \end{aligned} \tag{8}$$

Even though this formula looks complex, we can compute z_{rect} with the help of simple binary shifts.

After these preparations we now can apply our spatial index as follows:

- The add-on automatically creates an integer column for every geometry column. It is marked as index column inside the standard database.
- For geometries stored in the database, z values are computed according to equations 6 and 8 and stored inside the index column.
- The geometry itself is serialized to a column that is able to store binary values (usually of type **BLOB**).
- If we change a geometry, the z value has to be adapted. If we remove a geometry, the z value is removed as well, as it is stored in the same data row. No further computations or data changes are required.
- For queries that contain geometrical conditions, the candidate set is retrieved according to equations 4 and 5. For this, a query SQL string is extended by further conditions (see below). This extension is automatically performed by the add-on.
- As the set of candidates is a superset of real hits, the candidates undergo further geometric checks outside the database. These checks are carried out automatically by the add-on.

To give an example: We want to find all cities in Germany with less than 100 000 inhabitants. The non-geometric part of the query is

```
SELECT * FROM CITIES WHERE INHABITANTS<100000
```

Let's assume the tile number of the border of Germany is 55; we further define z_{max}=511, i.e. i_{max}=8. 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=27 OR IND=13 OR
         IND=6 OR IND=3 OR IND=1
         OR IND BETWEEN 110 AND 117
         OR IND BETWEEN 220 AND 227
         OR IND BETWEEN 440 AND 447)
```

Every SQL query is confined by i_{max}+1 index conditions. For this, we have to ensure that the maximum SQL string size is not exceeded. This is not a problem for typical SQL databases.

3.2 Further Reduction of Candidates

Until now we still have a problem: even small geometries may have a small tile index number, if they intersect separator lines with low plane indexes. E.g., all objects intersecting the equator have z=1. Such objects lead to a constant amount of candidates for all queries as tiles with z=1 are always inside the candidate set and have to undergo the exact geometric check (equation 4).

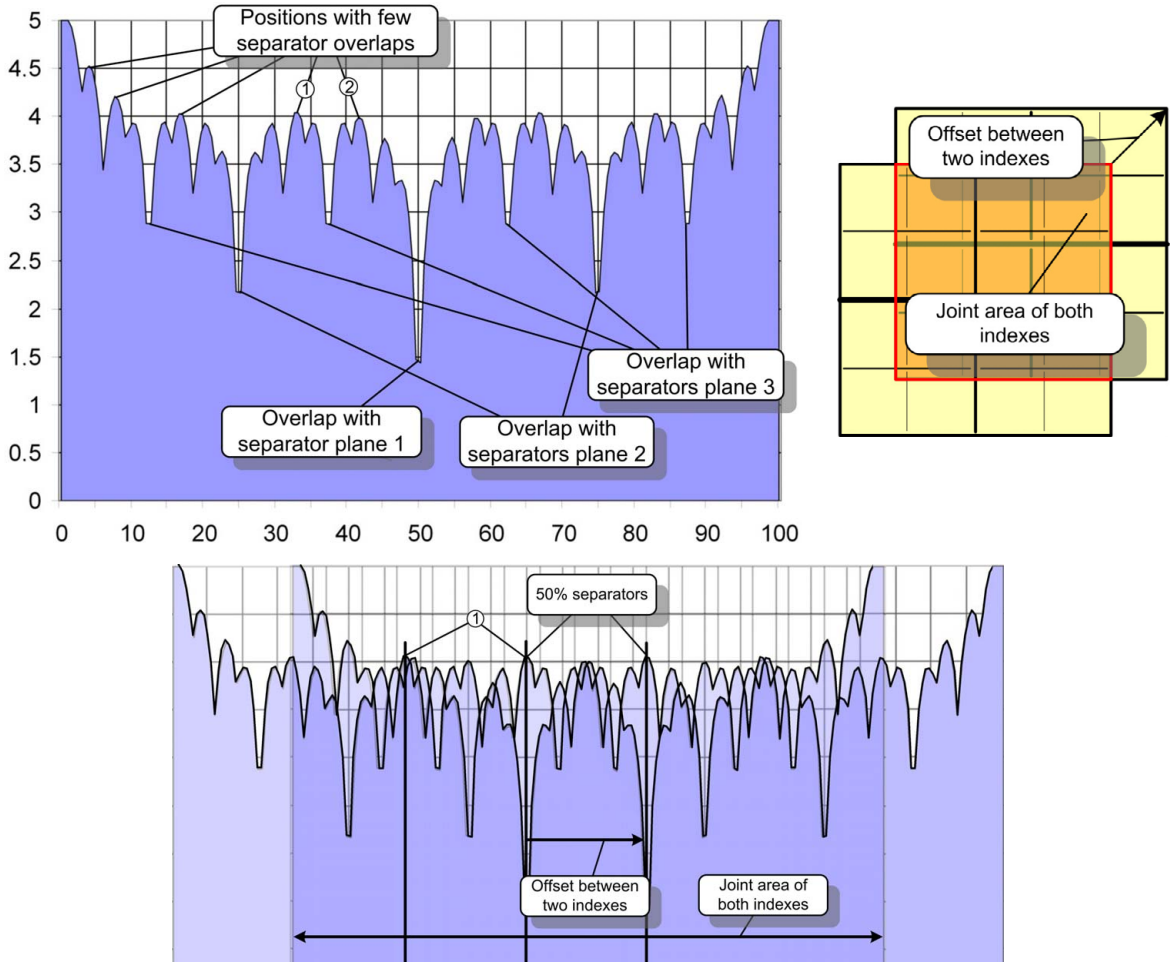

Fig. 2. Average plane number (top left), shifted index idea (top right and bottom)

To analyze this effect, we performed a simulation based on real geo data. Typical sizes of geo objects follow the power law, i.e. we have many small objects and only few large objects (Roth, 2005). Fig. 2 (top left) illustrates the results: the diagram maps a relative object's positions (i.e. its center) in one dimension (0-100%) to the average plane index (i_{max}=5). Not surprisingly, objects near the first separator (50%) have a low plane number on average. Separators of further planes (25%, 75%, 12.5%, 37.5%, 62.5% etc.) lead to further index degradations.

The solution to this problem is illustrated in fig. 2 (top right): we use two shifted indexes, each of it computes its own z number according to equations 6 and 8. 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 on the candidate set.

To find an appropriate offset is not a trivial task:

- If one index produces a low z number, the other index should produce a high one and vice versa.
- Objects that are larger than the offset may intersect with two shifted separators, thus the offset should be larger than the typical object size.
- If in turn, the offset is too large, the joint area of both indexes is small.

Looking at fig. 2 (top left) we can identify some candidates for appropriate offsets. Positions (1) and (2) are e.g. local maxima and nearby the 50% separator, thus the shifted index may have its 50% separator on these positions. If the one index generates a low number for positions nearby 50%, the other index generates an index number near to the maximum (fig. 2 bottom).

We can compute the relative position (1) as follows:

$$\frac{1}{2}-\frac{1}{4}+\frac{1}{8}\ldots\frac{1}{2^{i_{\max}+1}}\approx\frac{1}{3}\ \text{ i.e. the offset is }\ \frac{1}{2}-\frac{1}{3}=\frac{1}{6} \tag{9}$$

For the relative position (2) we get

$$\frac{1}{2}-\frac{1}{4}+\frac{1}{8}+\frac{1}{16}-\frac{1}{32}\ldots\frac{1}{2^{i_{\max}+1}}\approx\frac{5}{12}\ \text{ i.e. the offset is }\ \frac{1}{2}-\frac{5}{12}=\frac{1}{12} \tag{10}$$

Fig. 3 shows average plane numbers of shifted indexes for different offset values. For offsets 1/6 and 1/12 we nearly got constant plane values of 4 (generally $i_{\max}$-1), thus these offsets are appropriate values. Fig. 3 (right) illustrates the problem, if the offset is too small: many objects intersect both corresponding shifted separators, thus the overall behavior tends to the behavior of a single index (fig. 2 top left).

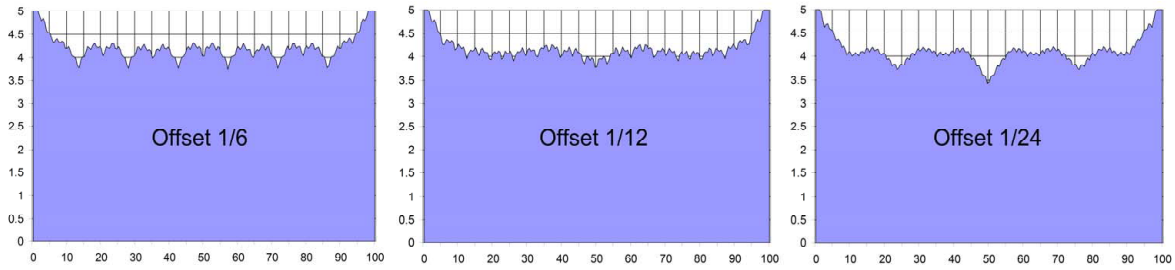

Fig. 3. Average plane number with shifted indexes

Using an offset of 1/6 the covered area is reduced by 31%; for 1/12 it is reduced by 16%. In order to still cover 100% of the original area, we have to decrease the resolution. Using the data type **`INT8`**, the remaining resolution is still sufficient.

3.3 Query Creation, Interoperability

The execution of the spatial index is fully hidden from the application developer. To allow the add-on to integrate the additional SQL conditions into a query, the developer has to use a special query API provided by the add-on. As an example: we again want to find all cities in Germany with less than 100 000 inhabitants. The application creates the query as follows:

```
select=dbman.getSelectStatement("CITIES");  // TABLE CITIES
condition=select.getWhere().createLT();     // CONDITION <
condition.addPart(select.                   // COL INHABITANTS
      getSelectTableReference().newColumnReference("INHABITANTS"));
condition.addPart(100000);                  // VALUE 100000
result=dbman.search(SearchType.WITHIN,GermanBorder,
      select);                              // INSIDE GERMANY
```

The additional spatial conditions are integrated with the help of the last call. Here, we assume that the variable **`GermanBorder`** contains the geometry of the German border.

Our add-on does not make any important demands on the underlying SQL database. Even though different SQL databases should in principle support the same standards, we identified some small differences, e.g.:

- The column type to store geometries usually is **`BLOB`**, but also **`ByteA`** (PostgreSQL), **`Binary`** (MS SQL Server) or **`LongVarBinary`** (HSQLSDB) are used. If a database does not support any binary type, we create a textual representation of a geometry stored in a **`VARCHAR`** column.
- The column for *z* values usually is **`INT8`**, but Oracle requires **`INTEGER`**.
- The connection strings (with login, password etc) differ between databases.

In order to provide a common interface to the add-on, we defined a wrapper that shields these small differences from the add-on. We implemented wrappers for PostgreSQL, MySQL, Oracle, Microsoft SQL-Server, SQLite and HSQLDB. Further wrappers can easily be implemented. The definition of a wrapper typically does not exceed 200 lines of code.

3.4 Performance Considerations

To evaluate the approach, we executed a number of performance measurements. They should justify the shifted index approach and demonstrate the performance for real scenarios.

To measure real systems means not only to measure the add-on but also the hardware, file system and database. In addition, the respective exact shapes of geo objects and query geometries highly affect the results. However, the following figures indicate a trend. We used a desktop computer with 2.49 GHz CPU and 3 GB RAM running Windows XP and a PostgreSQL database version 8.3. We imported an amount of 226 497 geo objects of the file 'Bavaria' from OpenStreetMap (OpenStreetMap, 2009). The add-on needed 2.4 ms per geo object to create an object entry in the database table including the index value.

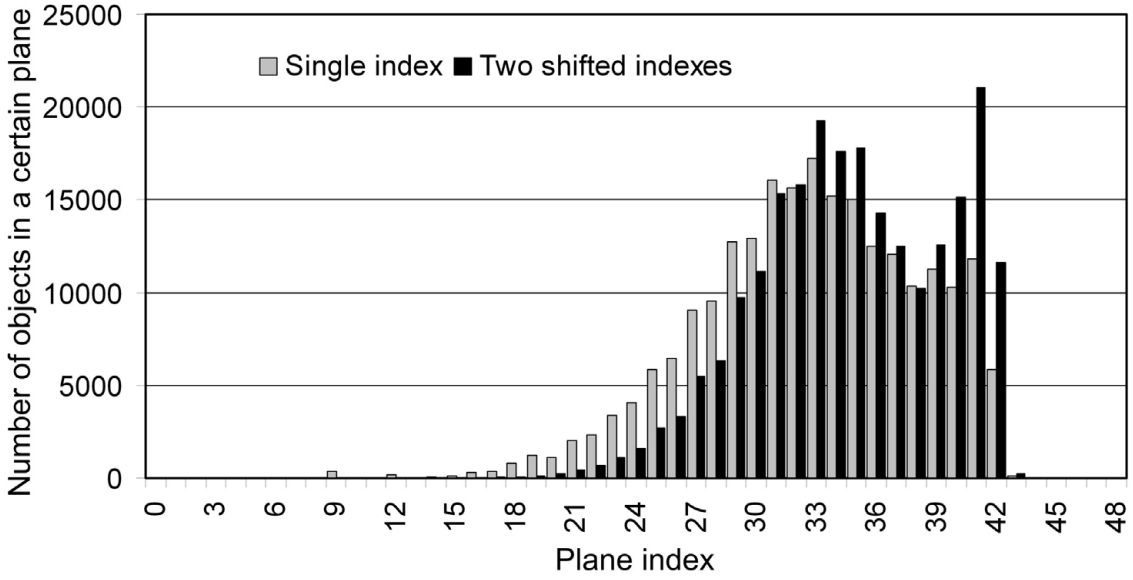

Fig. 4. Number of geo objects mapped to a certain plane

The first evaluation should identify the amount of objects that reside at a specific plane, both for a single index as well as for the shifted indexes (offset 1/6). For the latter we indicate the maximum plane number of both indexes. Fig. 4 shows the results. The average plane number is 32.8 (single index) and 34.7 (shifted indexes). This indicates a great benefit of the shifted indexes. In particular, the shifted indexes do not produce any low index numbers.

Further measurements indicate the actual runtimes (table 1). We execute purely geometric queries, i.e. the set of candidates was not reduced by non-geometric conditions. That is the common case, but would dilute our measurements. We measured the number of hits, the number of candidates, the time for database execution and the time for the geometric check by the geometry engine.

Compared to the single index, the shifted indexes significantly reduce the number of candidates. The time to check the candidate set is low compared to the database time. In summary, the Extended Split Index is an efficient approach with low costs for insertions. The query time is appropriate for real applications.

Table 1: Runtimes for different queries

Query Type	Query Geometry	Hits	Single Index			Shifted Indexes		
			Candidates	Database Time [ms]	Geometric Check [ms]	Candidates	Database Time [ms]	Geometric Check [ms]
CONTAINS	POINT(49.4574 11.0827)	2	1410	781	0.219	89	46	0.011
CONTAINS	POINT(49.8000 11.7000)	0	706	421	0.000	22	12	0.000
OVERLAPS	POLYGON(48.9574 11.0827…)	2	831	375	0.235	21	11	0.006
CONTAINS	POLYGON(49.4500 12.9000…)	0	590	313	0.000	71	30	0.000
WITHIN	POLYGON(49.4520 11.0927…)	33	3178	1109	0.047	272	104	0.006
WITHIN	POLYGON(49.4574 11.0827…)	345	3178	1188	0.078	2859	969	0.055

4. CONCLUSIONS AND FUTURE WORK

The presented spatial add-on allows the storage of huge amounts of geometric data and supports spatial queries. It only requires a standard SQL database and thus may be executed even on special platforms, e.g., mobile devices that are not able to run spatial databases. The add-on introduced the new Extended Split Index which only causes minimal costs for geometry updates and makes use of one-dimensional index columns supported by standard databases. Geometry queries are mapped to standard SQL queries.

In the current implementation, queries cannot express conditions on *derived* quantities of geometries such as the size of the surface area or shape diameter. In the future we want to integrate such quantities into queries. In principle, these quantities could be computed from geometries and stored in separate columns, but this would require additional column updates for each geometry change.

In the current implementation, we compare stored geometries with a given query geometry. As a further goal we want to introduce more complex query types known from traditional databases. The greatest challenge will be the *spatial join* that relates two geometric columns in a single query.

REFERENCES

Aquino J., 2003. JTS Topology Suite. Technical Specifications, Vivid Solutions

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

Guttman A., 1984. 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, 47-57

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

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

Murray C., 2008. Oracle Spatial Developer's Guide. Oracle, Aug. 2008

Neufeld K., 2009. PostGIS Manual

OpenStreetMap, 2009 excerpts for Europe, http://download.geofabrik.de/osm/

Roth J., 2005 A Decentralized Location Service Providing Semantic Locations. Informatik Report 323, University of Hagen, Jan. 2005

Stolze K., 2003. SQL/MM Spatial: The Standard to Manage Spatial Data in Relational Database Systems, BTW 2003, Leipzig, Feb 2003

Sun, 2009. MySQL 5.0 Reference Manual. Sun Microsystems

Tropf H., Herzog H., 1981. Multidimensional Range Search in Dynamically Balanced Trees. Applied Informatics, 2/1981, Vieweg, Germany, 71–77