STUDIA UNIV. BABEŞ–BOLYAI, INFORMATICA, Volume **LVI**, Number 4, 2011

LOCATION BASED APPLICATION PERFORMANCE STUDY IN MYSQL

ADRIAN SERGIU DĂRĂBANT, VIORICA VARGA, LEON ŢÂMBULEA, AND BAZIL PÂRV

Abstract. This paper presents a short study and an optimization process based on data organization in mySQL databases. Spatial data is largely used nowadays not only on web but also in mobile and desktop applications. We show that organizing spatial data as spatial structures, as it comes naturally is not always the best approach when dealing with query response times. Although many database engines have strong spatial features, characteristics of the data and applications might fit better sometimes with relational modeling or even, in extreme cases, with deviated relational modeling.

1. Introduction

Location-based services (LBSs) are IT services for providing information that has been created taking into consideration the current locations of the users or mobile objects. They can also appear in conjunction with conventional services like telephony. LBSs' participants do not have to enter location information manually, they are automatically pinpointed and tracked.

The simplest type of LBSs are enquiry and information services, which provide the mobile user with nearby *points of interest* such as restaurants, movies, or filling stations. The user is automatically located by the mobile network. He must specify the points of interest, for example, whether he would like to receive a list of all nearby theaters or concerts, and the desired maximum distance between his current position and the points of interest. The request is then passed to a service provider, which assembles a list of appropriate points of interest and returns it to the user.

When dealing with LBSs it is first important to be clear about the meaning of the term *location.* Basically, the term location is associated with a certain

Received by the editors: October 20, 2011.

2010 *Mathematics Subject Classification.* 68P15.

1998 *CR Categories and Descriptors.* H.2.8 [**Database Management**]: Database Applications – *Spatial databases and GIS*.

Key words and phrases. relational database, query optimization, location based services.

place in the real world. These kinds of locations belong to the class of *physical locations*. There are also virtual locations (see [2]). There are different kind of physical locations, we are interested in spatial locations. A spatial location represents a single point in the Euclidean space. Another, more intuitive term for spatial location is position. A convenient way to express spatial location is to use an ellipsoidal coordinate system that models the Earths surface as an ellipsoid. A reference ellipsoid describes the shape of the Earth by fixing an equatorial and a polar radius. The origin of an ellipsoidal coordinate system is given by two reference planes, both arranged orthogonal to each other and crossing the geocenter. The horizontal plane corresponds to the equatorial plane. The vertical reference plane includes the Earths rotation axis and hence intersects North and South Poles.

A position at the surface of the Earth is then represented by the angles between the reference planes and the line passing from the geocenter to the position (see Figure 1). The latitude ϕ is defined as the angle between the equatorial plane and the line, that is, it represents the NorthSouth direction measured from the Equator, while the longitude λ describes the angle between the vertical plane and the line, that is, it reflects the EastWest direction of a position with regard to the vertical reference plane

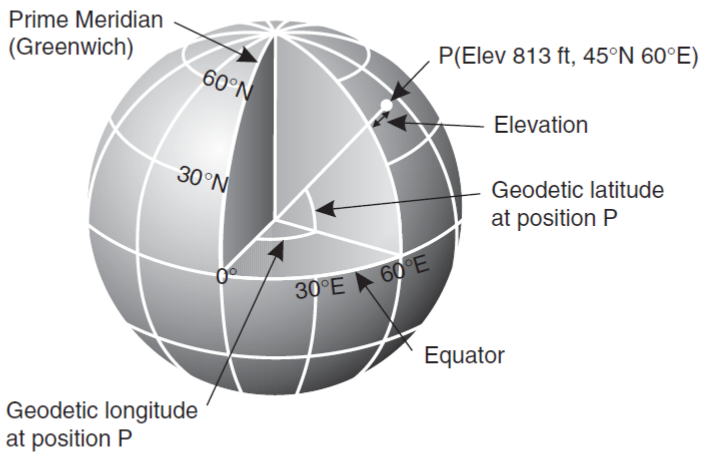
Prime Meridian (Greenwich)
P(Elev 813 ft, 45°N 60°E)
60°N
Elevation
30°N
Geodetic latitude at position P
0°
30°E
60°E
Equator
Geodetic longitude at position P

Figure 1. Latitude and longitude on Earth

Spatial location or position information represents an appropriate means for exactly pinpointing an object on Earth. One of the positioning methods used in LBSs is the Global Positioning System (GPS), for example, delivering 48°06′37″N 16°34′11″E we know that the target person currently resides at the airport in Wien, Austria. A well known application is to determine the distance between the LBS user and a selected point of interest and to derive the expected traveling time from that. A simple approach would deliver only the geodesic (straight)line distance between both positions, but it would be more convenient for the user to get the shortest route distance in a road or public transportation network, preferably in combination with the shortest route displayed on a map and additional navigation assistance.

Spatial databases [3] and Geographic Information Systems (GISs) are the essential key technologies which deal with the mapping between spatial and descriptive location information as well with maintaining and deriving relationships between locations in general. Spatial database management systems support queries that involve the space characteristics of the underlying data. For example, a spatial database may contain polygons that represent building footprints from a satellite image, or the representation of lakes, rivers and other natural objects. It is important to be able to query the database by using predicates related to the spatial and geometric characteristics of the objects. To handle such queries, a spatial database system is enhanced by special tools. These tools include new data types, sophisticated indexing mechanisms and algorithms for efficient query processing that differ from their counterparts in a conservative alphanumeric database.

The aim of this paper is the study of query optimization in a mySQL database containing a large dataset with information about points of interest, events and their geographical coordinates. Data models explored in this paper are: normalized relational data, spatial data model and relational data model with materialized views. For query optimization we use B+ tree indexes and stored procedures in case of normalized relational data model, R-trees for the spatial model. The next section is about the used data models and the query performance for these models. The last section presents the experimental results in the proposed models.

2. mySQL relational versus spatial organization performance

In this paper we study and give solutions on optimizing data access to a large mySQL database about points of interest, events and their geographical coordinates. Being a POI database all queries that are running against the database are invariably filtering on the position coordinates. For most of them time is also involved as an interval. The first solution that comes to mind is to model data using the spatial features of the mySQL server in order to let the system perform the computations on the geographical coordinates. The test case database we try to optimize has between 5 and 10 GB of data with millions of non unique records containing geographical coordinates and the

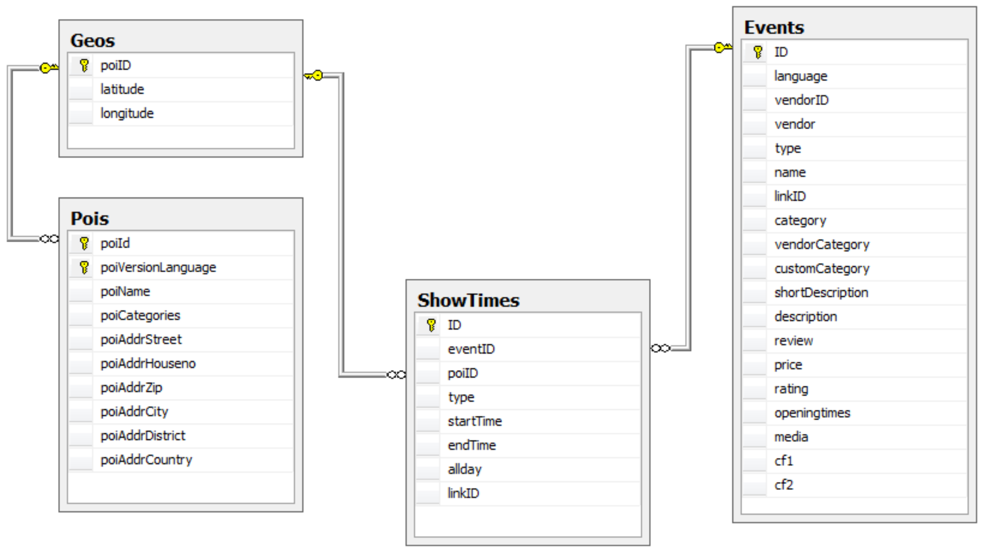

Figure 2. Simplified database scheme

main goal is to find a data organization/structure that has the best response times for queries.

In the following we will try to apply a few optimizations on the different data organizations in order to improve query response time while still maintaining mySQL as database engine. Most of the queries variants are searching different kind of events (and show times) and places in an area around the user position and within a specified time interval. The location of the user is given. The search for show times is made in the bounding box of the user's position. In the first approach we store the data in a relational database in normalized tables: `Geos`, `Pois`, `Events` and `ShowTimes`. Every point of interest has different name in every different language, so we store the location of the point of interest in the `Geo` table once and the name, address, etc. in table `Pois`. One event can appear with its show time at different points of interest, so we store the event once in table `Events` and its show times in table `ShowTimes`, where the `poiID` represents the location of the event. We present only a few of tables' columns. The simplified structure of the database is presented in Figure 2.

The problematic queries against this database involve the join of tables `Geos`, `Pois`, `Events` and `ShowTimes`. In this paper we pick five of the slow queries for presentation. The queries select show times of different events from crowded areas, like Paris, Wien, New York, etc. The location is parameter for every query; we give the bounding box of the selected city. For example the latitude for Paris has to be between 48.6 and 48.93 and the longitude between 2.20 and 2.5. The start time and end time are parameters too; usually we select the show times next week after the current date. Every query returns the point of interest's location, name and address, the name of event and other information.

(1) The first query (QT1) selects the show times of *every* event category from one given vendor in the given location for the given time period.

```
SELECT e.ID, sh.poiID, e.language, e.type, e.vendor, e.vendorID,
   e.name, e.category, e.vendorCategory, p.poiVersionLanguage,
   GROUP_CONCAT(sh.startTime) AS START_TIMES, GROUP_CONCAT(sh.endTime),
   g.latitude, g.longitude, p.poiName, p.poiAddrCity,
  FROM (Events AS e INNER JOIN ShowTimes sh ON e.ID = sh.eventID)
    INNER JOIN Geos AS g ON sh.poiID = g.poiID
    INNER JOIN Pois AS p ON sh.poiID=p.poiId
  WHERE g.latitude BETWEEN ? AND ? AND g.longitude BETWEEN ? AND ?
    AND (sh.startTime >= ? OR sh.endTime >= ? OR
           (sh.startTime = ? AND sh.endTime IS NOT NULL))
    AND sh.startTime < ? AND sh.type = 0 AND e.vendor= ?
  GROUP BY sh.eventID, sh.poiID, e.language, p.poiVersionLanguage
  ORDER BY sh.startTime;
```

(2) The second query (QT2) searches the show times of *theater, movies, dance, etc* event categories from one given vendor in the given location for the given time period.

```
SELECT e.ID, sh.poiID, e.language, e.type, e.vendor, e.vendorID,
   e.name, e.category, e.vendorCategory, p.poiVersionLanguage,
   GROUP_CONCAT(sh.startTime) AS START_TIMES, GROUP_CONCAT(sh.endTime),
   g.latitude, g.longitude, p.poiName, p.poiAddrCity,
  FROM (Events AS e INNER JOIN ShowTimes sh ON e.ID = sh.eventID)
    INNER JOIN Geos AS g ON sh.poiID = g.poiID
    INNER JOIN Pois AS p ON sh.poiID=p.poiId
  WHERE g.latitude BETWEEN ? AND ? AND g.longitude BETWEEN ? AND ?
    AND (sh.startTime >= ? OR sh.endTime >= ? OR (sh.startTime= ? AND
            sh.endTime IS NOT NULL)) AND sh.startTime < ?
    AND e.type = ? AND e.vendor = ? AND  category like ?
  GROUP BY sh.eventID, sh.poiID, e.language, p.poiVersionLanguage
  ORDER BY sh.startTime;
```

(3) The third query (QT3) selects the show times of a *parameter* given event category from the second vendor in the given location for the given time period.

```
SELECT e.ID, sh.poiID, e.language, e.type, e.vendor, e.vendorID,
   e.name, e.category, e.vendorCategory, e.cf1, e.description,
   GROUP_CONCAT(sh.startTime) AS START_TIMES, GROUP_CONCAT(sh.endTime),
   e.media, g.latitude, g.longitude, p.poiName,
   p.poiAddrCity, p.poiVersionLanguage
  FROM (Events AS e INNER JOIN ShowTimes sh ON e.ID = sh.eventID)
    INNER JOIN Geos AS g ON sh.poiID = g.poiID
    INNER JOIN Pois AS p ON sh.poiID = p.poiId
  WHERE g.latitude BETWEEN ? AND ? AND g.longitude BETWEEN ? AND ?
    AND (sh.startTime >= ? OR sh.endTime >= ? OR (sh.startTime= ? AND
            sh.endTime IS NOT NULL)) AND sh.startTime < ?
    AND e.type = ? AND e.vendor = ? AND e.customCategory LIKE ?
  GROUP BY sh.eventID, sh.poiID, e.language, p.poiVersionLanguage;
```

(4) The forth query (QT4) selects the show times of *cinema* point of interest category from the second vendor in the given location for the given time period.

```
SELECT e.ID, sh.poiID, e.language, e.linkID, e.vendorID, e.name,
   e.customCategory, e.rating, e.cf1, e.media,
   GROUP_CONCAT(sh.startTime) AS START_TIMES,
   g.latitude, g.longitude, p.poiName, p.poiVersionLanguage
 FROM (ShowTimes AS sh INNER JOIN Events AS e ON sh.linkID =e.linkID)
    INNER JOIN Geos AS g ON sh.poiID = g.poiID
    INNER JOIN Pois AS p ON sh.poiID = p.poiId
 WHERE p.poiCategories LIKE '%Cinema%'
   AND (sh.startTime >= ? OR sh.endTime >= ? OR (sh.startTime = ?
   AND sh.endTime IS NOT NULL)) AND sh.startTime < ?
   AND g.latitude BETWEEN ? AND ? AND g.longitude BETWEEN ? AND ?
   AND e.vendor=? AND sh.type = 1 AND e.cf2= ?
 GROUP BY e.id, sh.poiID, e.language, p.poiVersionLanguage;
```

(5) The fifth query (QT5) searches the show times of *concert* event category from one given vendor in the given location for the given time period.

```
SELECT e.ID, sh.poiID, e.language, e.name, e.category, e.vendorID,
  GROUP_CONCAT(sh.startTime) AS START_TIMES, GROUP_CONCAT(sh.endTime),
  g.latitude, g.longitude
 FROM (Events AS e INNER JOIN ShowTimes sh ON e.ID = sh.eventID)
    INNER JOIN Geos AS g ON sh.poiID = g.poiID
 WHERE g.latitude BETWEEN ? AND ? AND g.longitude BETWEEN ? AND ?
   AND (sh.startTime >= ? OR sh.endTime >= ? OR
      (sh.startTime = ? AND sh.endTime IS NOT NULL)) AND sh.startTime < ?
   AND (e.type = 1 OR (e.type = 0 AND e.category LIKE '%concerts%'))
   AND e.vendor = ?
 GROUP BY sh.eventID, sh.poiID, e.language
 ORDER BY sh.startTime
```

As we can see every query needs the join of the tables: `Geos`, `Pois`, `Events` and `ShowTimes`. We test these five queries as single SELECT statements against the MySQL database. The dates and location data were generated randomly from highly agglomerated cities around he world. The question marks are replaced with the actual query arguments.

In order to optimize the execution of the queries, first we study the execution plan of them, than we build additional index files for attributes from filter conditions or join conditions. The most widely used of several index structures in the relational approach is the B+ tree [6] that maintains efficiency despite insertion and deletion. It takes the form of a balanced tree in which every path from the root of the tree to a leaf of the tree is of the same length. In order to find a key in a B+ tree, the database engine has to read a number of blocks equal with the height of the tree. A leaf node stores the key together with the pointer (rid) to the data file. Using the rid of the record the whole record can be read with one I/O operation. An important property of the B+-tree index is its *fan-out* [4], which is the number of entries in a page. The height of the tree is proportional to $log_{fan-out}$(nr.data entries). In order to decrease the search time the fan-out has to be increased, so the height will decrease. We vary thus in our test the size of the block, in order to allow it to store more keys.

In the second approach we formulate the queries as stored procedures to benefit of already compiled execution plans. We try to force the execution of queries in different orders, processing the selections first and store the partial results in temporary tables.

In the third step we test the GIS functionality of mySQL. The `point` data type is introduced [5] for latitude and longitude, we have the next table:

```
CREATE TABLE ‘geoSpatial‘ (
  ‘poiID‘ varchar(64) NOT NULL,
  ‘location‘ point NOT NULL,
  PRIMARY KEY (‘poiID‘)
  )
```

Spatial data may be indexed too. R-tree (”R” from Region) is used for spatial data indexing ([1]). It involves organizing the minimum bounding rectangle (MBR) of the spatial objects in a tree structure. There are a few variations of R-tree indexing, MySQL uses R-trees with quadratic splitting [5], which is one of the standard methods of building an R-tree index. An R-tree index is similar to a B+-tree in many ways, and organizes the indexed nodes in a hierarchy where the nodes in the index represent the MBR of the objects in the node. The leaf nodes in the index contain references to the row that contain the data, just like a B+-tree index. An R-tree for the location column can be constructed with the following command:

```
CREATE SPATIAL INDEX sp_index
ON geoSpatial (location);
```

Query QT1 for point of interests near Wien using table `geoSpatial` is:

```
SET @g1 = GeomFromText(
’Polygon((48.1 16.2, 48.1 16.5, 48.3 16.5, 48.3 16.2, 48.1 16.2))’);
```

```
SELECT e.ID, sh.poiID, e.language, e.type, e.vendor, e.vendorID,
   e.name, e.category, e.vendorCategory, p.poiVersionLanguage,
   GROUP_CONCAT(sh.startTime) AS START_TIMES, GROUP_CONCAT(sh.endTime),
   X(g.location), Y(g.location), p.poiName, p.poiAddrCity,
  FROM (Events AS e INNER JOIN ShowTimes sh ON e.ID = sh.eventID)
    INNER JOIN geoSpatial AS g ON sh.poiID = g.poiID
    INNER JOIN Pois AS p ON sh.poiID=p.poiId
  WHERE MBRContains(@g1,g.location)
    AND (sh.startTime >= ? OR sh.endTime >= ? OR
           (sh.startTime = ? AND sh.endTime IS NOT NULL))
    AND sh.startTime < ? AND sh.type = 0 AND e.vendor= ?
  GROUP BY sh.eventID, sh.poiID, e.language, p.poiVersionLanguage
  ORDER BY sh.startTime;
```

In the fourth approach we use materialized views on the three main tables of the database and perform the same queries. We keep data fully organized in a relational manner. By default mySQL does not have support for materialized views but they can be fairly easy implemented with the help of the database engine primitives. There are also external implementations to mySQL (plug-ins) that export this functionality. One of these implementations: *flexviews*[7] provides even for incremental view update in order to optimize the insertions and table updates.

Finally there is a fifth approach where we try to load the entire database in a memory simulated disk, but with a proprietary specific file system in order to analyze the impact disk reading and seeking operations have to the overall processing.

3. Experiments and results

In the following paragraphs we show a comparative study of the five different scenarios proposed above. The approach with stored procedures exhibits exactly the same response time as executing the actual queries and is not represented in the following figures.

In the RAM backed database comparison we show the worst response time in order to pinpoint the impact of disk operations. According to Figure 3 it seems that the disk operations do not have a great impact on the overall query processing time. However, we note here that the only RAM file system available that could store a 5 GB to 10 GB database was a *linux tmpfs* which is different than a fully fledged disk based file system as ext3. Tests performed on smaller databases that are loaded in file systems based ram disks do show a better response time. Creating a large file system based RAM disk was impossible with the operating system configuration we used to run our tests. Although the physical test machine has 24 GB of RAM that could be used, the kernel limitations of the underneath operating system did not allow creating

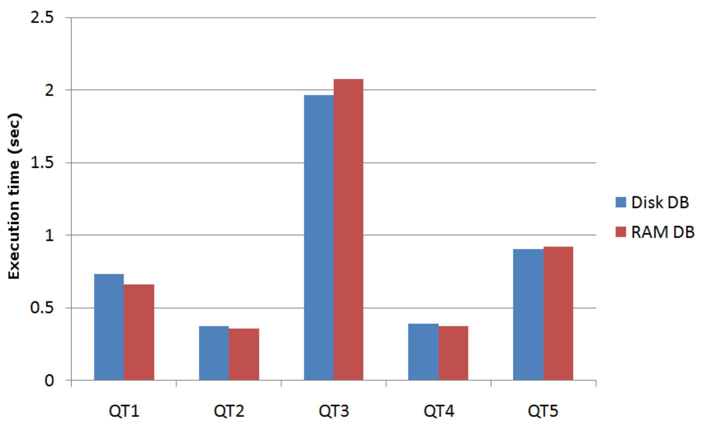

FIGURE 3. RAM loaded database versus on disk database.

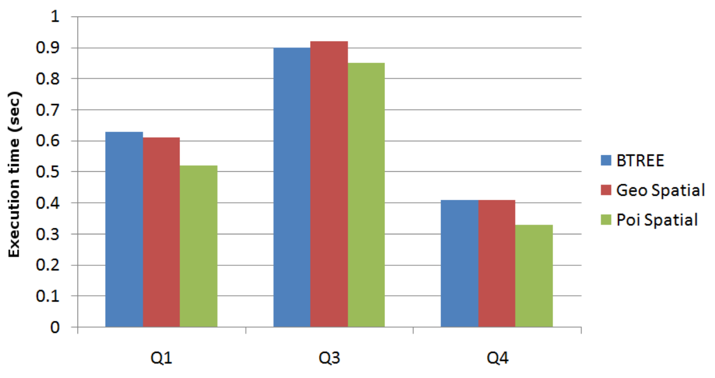

FIGURE 4. Spatial versus relational comparison

large usable file system based RAM disks. We did not focused our research on this variant because it is highly improbable for such a hardware dependent solution to be accepted in a practical production database system. We produced the RAM based experiments to merely show that the improvement of the response time is not that important to consider this approach as a potential solution. MySQL already caches data and can be instructed to keep indexes in system memory, fact that explains the insignificant improvements (where any are present) of the response time.

Figure 4 shows the average response times for the relational normalized approach and two spatial organizations with spatial data either in *Pois* or in

Geos tables. As observed the spatial approach improves only marginally the response time. Our initial approach was to model the entire dataset spatially. Searching in a well defined area often reduces to a few operations that could be easily implemented in pure SQL using the *between* and $<,>$ operators. When comparing the performance of the relational *between* operator, for range queries, with that of location aware primitives we found no major difference. Searching for points into rectangular or polygonal shapes is not faster than real numbers range checking using *between.* The entire approach using spatial features, as shown in figure 4, does not change the average response time in a significant way.

According to figure 5, a materialized view structure highly improves the average response time. The main bottleneck in the query evaluations (also according to the query execution plans) proves to be the execution of join operations. This is usually a high cost operation, but the fact that MySQL only implements nested loops [5] as join strategy penalizes the average response times. The materialized view schema helps cutting by a factor of two some of the average response times. As the main purpose was to find a solution that would allow executing between 2-5 queries per second this is the only approach that seems to help. During the experiments we also played with the database parameters, but the materialized view approach was the one that kept the best performance. For databases with high and very high frequency update and insert operations it is probably not the best approach as one needs a very good implementation, with incremental updates, of the materialized view functionality. The materialized view reduces the response time by a factor of two for most of the applications. The problem we tried to solve implies a moderate to very low frequency update which proves to be the ideal candidate for our last approach.

4. Conclusions and further research

This paper shortly presents a location aware application and its data stored in a mySQL database under various organization (relational, spatial structures, relational with materialized views). Our initial database modeling approach was a fully normalized relational. The main drawback of this model is that it could not stand a high concurrency degree because of the large response times required to evaluate the user queries. We investigated alternative data organizations in order to allow for concurrency factors of 2-5 queries/sec. Given the fact that the database is mostly static we discovered that for this case the best scenario is not a spatial organization with spatial indexes but a relational model with materialized views. Of course this approach is best suited to mostly static databases but we used the study to discover the major problems of the

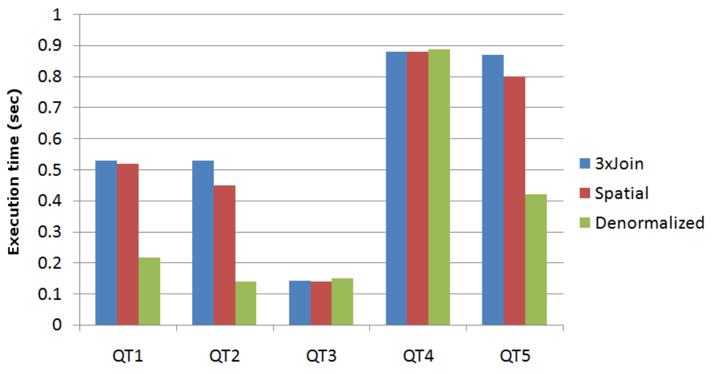

FIGURE 5. Relational, spatial and materialized view comparison.

mySQL engine in the context of spatial information and large databases in the context of simple spatial operations. We plan to adapt the current results in order to derive and implement a custom R-Tree mySQL indexing technique allowing loading large portions of the index in memory for fast geographical lookups, combined with a hash join execution strategy.

5. ACKNOWLEDGEMENT.

The author Viorica Varga has been fully supported by Romanian Ministry of Education in the frame of Research Grant CNCSIS PCCE-55/2008.

This work is also supported by Nokia Romania - Methods (Techniques) for Efficiently Searching in Spatial Data.

REFERENCES

[1] H. Garcia-Molina, J. D. Ullman, J. Widom: Database Systems: The Complete Book, Prentice Hall, 2009
[2] A. Küpper: Location-Based Services: Fundamentals and Operation. JOHN WILEY & SONS, LTD. (2005)
[3] Y. Manalopoulos; A. Papadopoulos; M. Gr. Vassilakopoulos: Spatial Databases: Technologies, Techniques and Trends, 2005.
[4] R. Ramakrishnan, J. Gehrke: Database Management Systems, Third Edition, WCB McGraw-Hill, 2003.
[5] Oracle Corporation: MySQL 5.1 Reference Manual, 2010.
[6] A. Silberschatz, H. F. Korth, S. Sudarshan: Database System Concepts, McGraw-Hill, Fifth Edition, (2005)
[7] * * *, FLEXVIEWS - Incrementally refreshable materialized views for MySQL, http://code.google.com/p/flexviews, consulted oct 2011.

Babes-Bolyai University, Faculty of Mathematics and Computer Science, 1 M. Kogălniceanu str., Cluj-Napoca 400084, Romania

E-mail address: dadi@cs.ubbcluj.ro

E-mail address: ivarga@cs.ubbcluj.ro

E-mail address: leon@cs.ubbcluj.ro

E-mail address: bparv@cs.ubbcluj.ro