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

# STORING LOCATION-BASED SERVICES' DATA IN KEY-VALUE STORE

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

Abstract. This paper proposes some solutions and discusses some issues related to the mapping of relational data and queries to a Key-Value store. The problem is stated for Location-based services (LBS), which deal with millions of users and thus millions of database records. The mapping process of a relational table consists in a transformation that assigns a part of the tables's row to the `Key` part and another part to the `Value`. We discuss two approaches to the mapping: generic and a LBS specific. The generic approach is a universal solution that can be applied to any relational to Key-Value mapping process.

## 1. Introduction

The data management research community is confronted with building consistent, available, and scalable data management systems capable of serving petabytes of data for millions of users especially for web application. Distributed database systems [13], [9] were the first generic solution that dealt with data not bounded to the limits of a single machine. These systems are not used frequently in industry because of their high complexity and due to the crippling effect on performance caused by partial failures and synchronization overhead.

Recent years different classes of scalable data management systems have been appeared such Google′s Bigtable [5], Amazon′s Dynamo [8] and PNUTS [6] from Yahoo! and others. All of these systems deal with petabytes of data, serve millions of requests with high availability requirements and run on cluster computing architectures. With the growing popularity of the cloud computing paradigm, many applications are moving to the cloud.

Received by the editors: October 25, 2011.

2010 *Mathematics Subject Classification.* 68P15, 68P20.

1998 *CR Categories and Descriptors.* H.3.2 [**Information Storage and Retrieval**]: Information Storage – *File organization*.

*Key words and phrases.* Key-Value store, relational model, data design, query mapping.

Nowadays mobile network operators create differentiation through the delivery of highly personalized services. One of the most powerful ways to personalize mobile services is based on location. These systems have millions of users. Location-based services (LBSs) are IT services for providing information that has been created, selected, or filtered taking into consideration the current locations of the users or mobile objects. They can also appear in conjunction with conventional services like telephony. The attractiveness of LBSs results from the fact that their participants do not have to enter location information manually, but that they are automatically pinpointed and tracked.

The simplest type of LBSs provide the mobile user with nearby *points of interest*s such as hospitals, parking places, 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. Location-based services in mobile network store usually the required data in distributed database systems. The response time of these systems for a large number a users is not adequate. In a current work [7] we study the performance of *location based applications*. In the previous approach [7] the problem of finding events, the events' show times in the current week for a user was studied. The search for show times was made around the user's position. The data has been stored in a relational database in normalized tables and the response time was not acceptable.

[1] analyzes the design choices that allowed modern scalable data management systems to achieve orders of magnitude higher levels of scalability compared to traditional databases. The authors give some possible design solutions for data management in the cloud.

Database research recognizes today [4] that different data models and database technologies should be used in different application domains. As an example, many companies in the Web industry have abandoned traditional relational DBMSs for so-called "No-SQL" data stores. This paper refers to *key-value* stores. Our goal is to analyze the pros and cons of transferring huge amounts of data of a location-base application from a distributed relational DBMS to a *key-value* store in the cloud in order to exploit the performance of the latter and, in the same time, to provide good response times for queries. The goal of this paper is to lay some design principles for the data management systems serving the next generation of applications in the cloud.

The structure of the paper is as follows. After this introductory section, the following section introduces the technical details of a *No-SQL* store. The

original part of the paper follows. The next two sections discuss in some detail the options regarding data design and storing, and relational query mapping to *key-value* stores. Last section draws some conclusions and sets future research directions.

## 2. No-SQL store

The data model for "No-SQL" store it is not a unique one. It can be build by basic data elements called "documents", "objects", or "records". According to [4] its essence is not the lack of SQL, but the presence of the following features:

- a little or no pre-defined schema; objects, records, or documents can have any number of attributes of any type;
- a simple query interface, and not a SQL processor;
- high scalability over dozens or hundreds of nodes, with the price of giving up 100% ACID semantics;
- eventual consistency - guaranteed consistency only within a single object, record, or document;
- high availability, necessary to make scalability across many machines useful.

Considering their data model and functionality, "No-SQL" data stores can be split into three groups: *key-value stores*, *document stores* and *extensible record stores*. We will use *key-value* stores, which have a distributed index for object storage. The stored objects are not interpreted by the system; they are stored and handled back to the client application as BLOBs. Basic functionality of *key-value* stores usually include object replication, partitioning the data over many machines, and a sort of object persistence;

A *key-value* store record has two parts: `Key` and `Value` part.

Different client API's were elaborated for different systems, see for example [2], [3]. We present the interface of [2] which includes the following operations:

- `STORE`: stores (*key, value*) in the file;
- `ADD`: adds (*key, value*) to the file iff the lookup for *key* fails;
- `REPLACE`: replaces (*key*, $value_1$) with (*key*, $value_2$) based on (*key*, $value_2$);
- `GET`: retrieves either (*key, value*) or a set of ($key_i$, $value_i$) pairs based on *key* or $key_i, i = 1 \dots k$.
- `DELETE`: deletes (*key, value*) from the file based on *key*

Taking into account these facts, the process of mapping a relational database to a *key-value* store is split into two sub-processes: migrating the data and transforming the queries.

## 3. Data design and storing in key-value store

The structure of one record in a key-value file being simple, the `Key` part and the `Value` part have to be determined. In order to know the structure of the record in a key-value file, we have to store its metadata. It can be represented using well-known formats like XML or JSON. The structure of the *key-value* record for LBS applications after the mapping can be described using (a) a generic style or (b) a custom structure for spatial data. The first solution is called generic because it can be applied to any relational database.

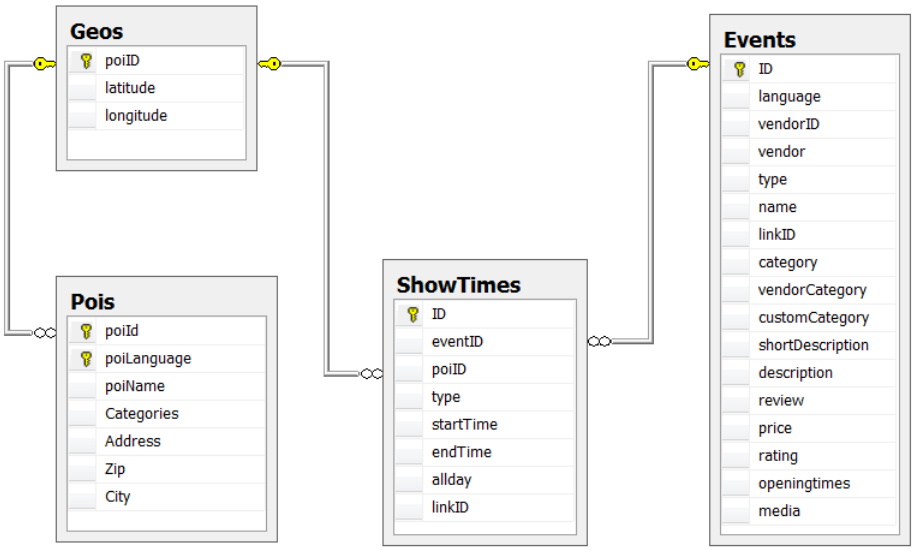


Figure 1. Simplified database scheme

3.1. **Generic style.** In this article the running example is about finding events, the events' show times in the current week for a user in a LBS application. The location of the user is known, the search for show times is made in the user's bounding box, which is a rectangle. In the first approach the structure of the data is presented in a relational database in normalized tables: `Geos`, `Pois`, `Events` and `ShowTimes`. Every point of interest has a different name for each 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. The simplified structure of the database is presented in Figure 1.

Now the proposed structure of the *key-value* record is presented. The metadata describes for each table of a database: the key part in `primaryKey` tag, the structure of the record in `Attribute` tags, index files in `IndexFile` and `IndexAttributes` tags, and the relationships in `foreignKey` tags.

```
<Databases>
  <DataBase dataBaseName="SpatialPOIs">
    <Tables>
      <Table tableName="geo" fileName="geo.kv" rowLength="114">
        <Structure>
          <Attribute attributeName="poiID" type="char" length="64" isnull="0"/>
          <Attribute attributeName="latitude" type="double" isnull="0"/>
          <Attribute attributeName="latitude" type="double" isnull="0"/>
        </Structure>
        <primaryKey>
          <pkAttribute>poiID</pkAttribute>
        </primaryKey>
        <IndexFiles>
          <IndexFile indexName="geoLat.ind" keyLength="25" isUnique="0" indexType="BTree">
            <IndexAttributes>
              <IAttribute>latitude</IAttribute>
            </IndexAttributes>
          </IndexFile>
          <IndexFile indexName="geoLong.ind" keyLength="25" isUnique="0" indexType="BTree">
            <IndexAttributes>
              <IAttribute>longitude</IAttribute>
            </IndexAttributes>
          </IndexFile>
        </IndexFiles>
      </Table>
      <Table tableName="poi" fileName="poi.kv" rowLength="626"></Table>
    </Tables>
  </DataBase>
</Databases>
```

FIGURE 2. The structure of the data

**Example 1.** For the running example the structure of `geo` *key-value* file is on Figure 2. The primary key and index files of the table can be seen too. In Figure 3 the metadata for `poi` *key-value* file is presented with the relationship (`foreignKey` tag) to `geo` *key-value* file.

Transforming a table `T` from the relational DB to the *key-value* store yields a set *key-value* files (usually more than one). The data from one table is stored in a *key-value* file, which we will name `master` file, where the `Key` part is the primary key of the table (`primaryKey` tag) and the `Value` part

```
- <DataBase dataBaseName="SpatialPOIs">
  - <Tables>
    + <Table tableName="geo" fileName="geo.kv" rowLength="114"></Table>
    - <Table tableName="poi" fileName="poi.kv" rowLength="676">
      - <Structure>
          <Attribute attributeName="poiID" type="char" length="64" isnull="0"/>
          <Attribute attributeName="poiLanguage" type="int" isnull="0"/>
          <Attribute attributeName="poiName" type="varchar" length="100" isnull="0"/>
          <Attribute attributeName="Categories" type="varchar" length="255" isnull="0"/>
          <Attribute attributeName="Address" type="varchar" length="100" isnull="0"/>
          <Attribute attributeName="Zip" type="varchar" length="25" isnull="0"/>
          <Attribute attributeName="City" type="varchar" length="100" isnull="0"/>
        </Structure>
      - <primaryKey>
          <pkAttribute>poiID</pkAttribute>
          <pkAttribute>language</pkAttribute>
        </primaryKey>
      - <foreignKeys>
        - <foreignKey>
            <fkAttribute>poiID</fkAttribute>
          - <references>
              <refTable>geo</refTable>
              <refAttribute>poiID</refAttribute>
            </references>
          </foreignKey>
        </foreignKeys>
      - <IndexFiles>
        - <IndexFile indexName="poiPoID.ind" keyLength="64" isUnique="1" indexType="BTree">
          - <IndexAttributes>
              <IAttribute>poiID</IAttribute>
            </IndexAttributes>
          </IndexFile>
        - <IndexFile indexName="poiName.ind" keyLength="100" isUnique="1" indexType="BTree">
          - <IndexAttributes>
              <IAttribute>poiName</IAttribute>
            </IndexAttributes>
          </IndexFile>
        </IndexFiles>
      </Table>
```

FIGURE 3. The structure of the data with relationships

is the concatenation of non-primary key attributes, so every attribute from `Structure` tag except the primary key of the table.

One table may have one or more unique index files. For every unique index file a *key-value* file has to be created: the `Key` part is the search key of the

index file, namely the concatenation of `IAttribute` values. The `Value` part of the unique index file is the corresponding `Key` part of the master file, which is the primary key of the `master` file. The search process on a unique index key requires two `GET(Key)` methods, the first on the unique index and the second on the master file, using as a key the value obtained in the first `GET(Key)` method. Where the underneath implementation allows access to the address of the `Value` part of the master file one can alternatively store this address as the `Value` part in the unique index. This approach avoids an additional data access when the index is used.

For every non-unique index file a new *key-value* file is created. More solutions can be given. If the *key-value* store accepts duplicate key values, then the mapping is immediate. Otherwise one can use solutions based on inverted indexes or *making search key unique* as proposed in [11] pp. 356-358.

Summarizing the above, for each table `T` the following *key-value* files have to be produced:

- the data from `T` is stored in a `master` *key-value* file `T.key-value` with `Key = T.PrimaryKey` and `Value` = concatenation of non-primary key attributes of `T`;
- for every unique index file `T.I` of `T` a new `T.I.key-value` file is created with `Key` = concatenation of `IAttribute` values and `Value = T.Key` (master);
- for every non-unique index file `T.I` of `T` a new `T.I.key-value` file is created. Different solutions: inverted index or making search key unique or duplicate key values.

**Example 2.** For relational table `poi` the following *key-value* files have to be produced:

- the data from the table is stored in a `masterPOI` *key-value* file with `Key = (poiID+poiLanguage)` and `Value` = `poiName+Categories+Address+Zip+City`;
- one *key-value* file for the `poiName` unique index key with `Key = poiName` and `Value = (poiID+poiLanguage)` ;

3.1.1. *Inverted index for non-unique index.* A single *key-value* record for each search key is built, with `Key` = search key value and `Value` = list of tuple pointers of the records (primary keys from master), where the corresponding search key is in the value part.

**Example 3.** Latitude is a search key for `geo` table (see geoLat.ind) and let be 3 poiID: 23456XX, 45678YY, 67890ZZ for which the latitude is 48.3. Value

48.3 will be stored once in the Key part and Value part will store the list of poiID-s: 23456XX;45678YY;67890ZZ, see Figure 4.

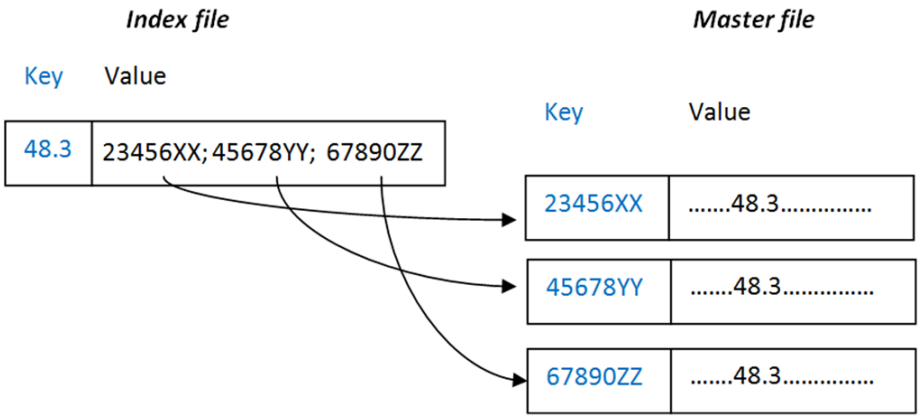


FIGURE 4. Inverted index example

3.1.2. *Make search key unique.* For each search key many *key-value* records are built, one for each record in the master with the same search key value. This is done by making search key unique, adding a record-identifier. In other words, `Key` = (search key + record identifier) and `Value` = NULL.

**Example 4.** For the example above, 3 different *key-value* records are built: (48.3#23456XX, NULL); (48.3#45678YY, NULL); (48.3#67890ZZ, NULL); See solution on Figure 5.

The "#" is used to separate the key part from unique identifier. The use of a separator can be avoided if constant (maximum) key-length is implemented. Shorter than maximum length keys are padded with spaces/blanks.

The option of making the search key unique, which is widely used for indexing relational databases, adds extra storage overhead for keys and extra custom code for insertion/deletion operations in the master file.

Inverted index option compared to *making the search key unique* adds low space overhead and no extra cost for queries, but may need extra code for handling long lists; also, deletion of a record in the master file may be expensive.

Compared to the relational model, finding a record in a *key-value* store using an index needs an additional read (`GET`) operation.

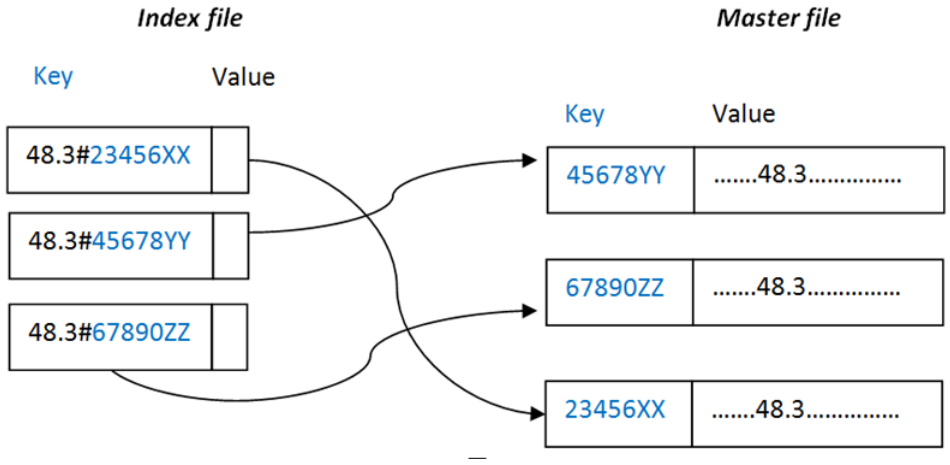


FIGURE 5. Making key unique example

3.2. **Custom solution for spatial data.** In this section we consider the design of the same environment in a *key-value* store. The divisions of the Earth in squares or rectangles called `Regions`, have to be stored in a `Regions` *key-value* file. Each region will have a unique key: the `RegionID`, so `Key` = `RegionID`. The `Value` part will have the following structure:

`UpperLeftLat,UpperLeftLong,LowerRightLat,LowerRightLong`

The poi-s, events, show times, can be stored in `Pois` *key-value* file. The aggregation of objects can be designed in different ways. One solution is a hierarchical style which can be described using regular expressions as follows:

```
geo(poi*, latitude, longitude);
poi(poiName,poiLanguage,Categories,Address,Zip,City,event*);
event(eventName, language, category, showtime*);
showtime(startTime, endTime);
```

`Pois.Key` = Region ID, `Pois.Value` = concatenation of `geo` composite objects, including poi-s, events, show times. By storing objects in this manner, information for one region is contained in a single object, so join operations are not necessary. The solution is usable, if such a `geo` object fits in memory. If it does not fit in memory, the region can be divided into more regions. One can use multiple division dimensions: regions, time, regions + time, etc.

Usually queries on spatial data involve not just a point (latitude, longitude), but a search area around the given point. This implies an additional radius. According to the location of the point in the region, multiple cases arise.

- the point and the search radius are completely contained in a single region
- the point and the search radius are contained in multiple regions

In a real system the search radius should be smaller than the region dimensions $r \leq l \leq L$, where $r$ is the search radius, $l$ and $L$ are the 2 dimensions of the region's bounding box. As a result a search area will not overlap more than four neighbourhooding regions.

## 4. Querying a key-value store

The mapping of queries needs to take into account the structure of the metadata, described in the previous section.

4.1. **Generic style.** In this case, there is a general solution: implement the `SELECT` statement over a *key-value* store (for more details see [15, 11]). This general solution is not discussed here.

The remaining part of this section discusses separately how to transform one-table queries and multiple-table queries, respectively. In the case of one-table queries, search by unique key is mapped straightforward, using `GET` on master and index files.

Search by non-unique key may use one of the following:

- *Use of inverted indexes.*
- *Make search key unique by adding a record-identifier.*

4.1.1. *Inverted index for non-unique index.* The search process is straightforward; the programmer has to handle the list of primary keys from *key-value* index files and retrieve every record from *key-value* master with the searched key, using `GET`.

4.1.2. *Make search key unique by adding a record-identifier.* The search is much difficult, because the search for one key value is transformed to a range query and range queries are not implemented yet in *key-value* stores [16, 12]. In our opinion, B+ Tree index is suitable for range queries, if the *key-value* store allows direct access to the tree.

**Example 5.** The search of all poiID-s for latitude 48.3 is transformed to the following range query:

```
minKey=48.3# + possible min value
maxKey=48.3# + possible max value
minKey <= Key <= maxKey
```

If the index file on Key is clustered, the minKey is found using the B+ tree and the data file is scanned sequentially from there until the record with $Key > maxKey$ is encountered.

In the case of multiple-table queries, each specific query is mapped in a custom way. However, some general guidelines can be given. Generally speaking, such a complex query is executed in a five-step process, as follows:

(1) apply the filter for each table, which reduces to a one-table query;
(2) establish a join order;
(3) compute the join in memory if possible; if not, apply hash join or other join types;
(4) compute `GROUP BY` operation of the query;
(5) compute `SORT` operation of the query.

For the implementation of these operation see [15, 11].

4.2. **Custom solution for spatial queries.** If the division in regions is adequate, the search is restricted to locate the location point of the current user in a region (this will need an index file on longitude and/or latitude), then read from `Regions` *key-value* file by regionID, and the complex object will be in the corresponding `Value`.

## 5. Conclusions and further work

In this paper we investigated the opportunity of using *key-value* stores as a replacement for traditional relational DBMSs. Two issues were discussed: data design and query mapping, together with the additional costs incurred. Future work on this topic will include implementation and testing issues on concrete spatial data models.

## 6. 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] D. Agrawal, A. E. Abbadi, S. Antony, S. Das: Data Management Challenges in Cloud Computing Infrastructures, *Databases in Networked Information Systems, 6th International Workshop*, DNIS 2010, Aizu-Wakamatsu, Japan, March 29-31, pp. 1-10

[2] M. Berezecki, E. Frachtenberg, M. Paleczny, K. Steele: Many-core key-value store, in *Green Computing Conference and Workshops (IGCC), 2011 International*, Orlando, July, pp. 1-8

[3] C. Bunch, J. Kupferman, C. Krintz Active Cloud DB: A. Database-Agnostic HTTP API to Key-Value Datastores. In UCSB. CS Technical Report 2010-07

[4] R. Cattell: Relational Databases, Object Databases, Key-Value Stores, Document Stores, and Extensible Record Stores: A Comparison, http://www.odbms.org/download/Cattell.Dec10.pdf

[5] F. Chang, J. Dean, S. Ghemawat, W.C. Hsieh, D.A. Wallach, M. Burrows, T. Chandra, A. Fikes, R.E. Gruber: Bigtable: A Distributed Storage System for Structured Data. In: *OSDI*. 2006, pp. 205218

[6] B.F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H.A. Jacobsen, N. Puz, D. Weaver, R. Yerneni: PNUTS: Yahoo!s hosted data serving platform. *Proc. VLDB Endow. 1(2)*, 2008, pp. 12771288

[7] A.S. Dărăbant, V. Varga, L. Ţâmbulea, B. Pârv: Location Based Application Performance Study in mySQL, to be appeared in Studia Universitatis "Babes-Bolyai" Cluj-Napoca, Informatica.

[8] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, W. Vogels: Dynamo: amazons highly available key-value store. *SOSP*. 2007, pp. 205220

[9] B.G. Lindsay, L.M. Haas, C. Mohan, P.F. Wilms, R.A. Yost: Computation and communication in R*: a distributed database manager. *ACM Trans. Comput. Syst. 2(1)*, 1984, pp. 2438

[10] H. Garcia-Molina, J.D. Ullman, J. Widom *Database Systems: The Complete Book*, Prentice Hall, 2008.

[11] R. Ramakrishnan, J. Gehrke, *Database Management Systems*, 3rd Edition, WCB McGraw-Hill, 2003.

[12] S. Ramabhadran, S. Ratnasamy, J.M. Hellerstein, S. Shenker Prefix Hash Tree: An Indexing Data Structure over Distributed Hash Tables, Technical Report, 2004 http://berkeley.intel-research.net/sylvia/pht.pdf

[13] J.B. Rothnie Jr., P.A. Bernstein, S. Fox, N. Goodman, M. Hammer, T.A. Landers, C.L. Reeve, D.W. Shipman, E. Wong: Introduction to a System for Distributed Databases (SDD- 1). *ACM Trans. Database Syst. 5(1)*, 1980, pp. 117

[14] T. Shimizu, M. Yoshikawa Full-Text and Structural Indexing of XML Documents on B+-Tree, *IEICE TRANSACTIONS on Information and Systems*, Vol. E89-D(2006), No.1, pp. 237-247

[15] A. Silberschatz, H. Korth, S. Sudarshan *Database System Concepts*, McGraw-Hill, New York, 2006.

[16] Range queries in *key-value* systems, http://groups.google.com/group/project-voldemort/browse_thread/thread/cad4888b492d897f

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

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

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

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

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