Leveraging Cloud Computing in Geodatabase Management

Ariel Cary*, Yaacov Yesha†, Malek Adjouadi‡ and Naphtali Rishe*
*School of Computing and Information Sciences
Florida International University, Miami, FL 33199 USA
Email: {acary001,rishen}@cis.fiu.edu
†Computer Science and Electrical Engineering
University of Maryland Baltimore County, Baltimore, MD 21250 USA
Email: yayesha@umbc.edu
‡College of Engineering and Computing
Florida International University, Miami, FL 33174 USA
Email: adjouadi@fiu.edu

***Abstract*—In this work, we leverage Cloud computing technologies in scaling out data management in geographical databases. In particular, we tackle the issue of data indexing in parallel. First, spatial data is partitioned and indexed in a Hadoop MapReduce cluster. Two main partitioning strategies are evaluated: a) A linear-complexity method based on Z-order values, and b) An iterative algorithm based on X-means clustering. The advantages and drawbacks of each method are weighted in with relation to query performance. Second, interactive queries are processed from a local site using the index data structures built in the Cloud. We perform an experimental study on a real dataset of 110 million spatial objects representing property parcels in the United States. Our results support Cloud computing as an effective technology to cope up with huge datasets and, in particular, MapReduce parallel programming model in easing parallel processing implementations.**

***Keywords*-GIS; Cloud Computing; MapReduce; Parallel Indexing;**

I. Introduction

Geodatabases represent the core of Geographical Information Systems (GIS). Geographical databases store both spatial and non-spatial attributes, and make extensive use of indexing data structures to provide fast spatial query processing. Particularly, R-tree indexes [1] are widely implemented on spatial attributes, and inverted files [5] on non-spatial attributes for free text searches. Data indexing may take a substantial amount of time depending on the size of the database. Users must wait until indexes are built before submitting queries. Waiting time may become a critical issue if frequent index refreshes are required.

There are currently many geographic databases of interest of moderate to large sizes, e.g. nationwide property parcels in the United States and international yellow pages contain dozens of millions of records. On the Internet, geographical information is constantly increasing by means of modern geotagging-enabled applications that allow users to associate geographic metadata to Web resources, e.g. Flickr[1] photo sharing application reports more than 102 million geotagged items (visited in May 2010). Single-process spatial indexing of such large-scale datasets may take excessive amount of time or it may be infeasible. Practical GIS applications must be able to cope up with current large-scale datasets.

Cloud computing, the idea of using computer cycles of hosts connected to a network, is a viable alternative to improve the scalability and high availability of (Internet) applications. In particular, Google's MapReduce [6] parallel programming model and its open-source clone Hadoop [13] have attracted the interest of both academic and industrial environments in implementing scalable and fault-tolerant data-intensive applications [14] [15]. Further, construction of R-tree indexes and inverted files are amenable for parallelization in MapReduce [8] [7].

In this paper, we leverage Cloud computing in combination with the MapReduce programming model to tackle data indexing of spatial and non-spatial attributes on large-scale geographical datasets. Three main ideas are discussed in this work. First, the general architecture of a geodatabase management system in the Cloud is presented. Second, two partitioning strategies for parallel index construction are discussed guided by our spatial query requirements. Third, since data partitioning strategies may influence index efficiency, e.g. objects likely to be accessed together may be located in different partitions, we experimentally evaluate query response times under both partitioning schemes. As a specific case, we target at processing k nearest neighbor (k-NN) spatial queries with Boolean constraints on non-spatial attributes [4]. Our techniques were implemented in the context of the TerraFly system – A 40-TB database of aerial and satellite imagery, and Web-based GIS[2].

The rest of the paper is organized as follows. Section II discusses Cloud computing related work. In Section III, we present the general architecture of our database and query systems. Parallel construction of spatial indexes is discussed

[1]http://www.flickr.com/map

[2]http://terrafly.fiu.edu

in Section IV. Experimental study is performed in Section V by indexing a dataset with 110 millions of spatial objects and evaluating query response times. Finally, Section VI summarizes our work.

II. Related Work

Cloud Computing Computer Clouds are typically optimized for one of the two major workloads: analytical and OLTP. Hadoop MapReduce is designed for analytical or batch workloads of data-intensive applications. Hadoop stores data on a distributed file system (HDFS), inspired in Google's GFS [11], optimized for massive sequential *I/O*. In Hadoop, input data is streamed into mappers and reducers and output is sequentially written back to HDFS. On the other hand, OLTP or serving Clouds are oriented to interactive applications that usually require a few random *I/O* accesses to quickly respond users' requests. Amazon Web Services (AWS)[3] and Google App Engine platform are examples of serving Clouds. Usually, analytical-oriented Clouds are used to pre-compute some form of intermediate results, e.g. pre-aggregating data or data indexing, which are later used by serving Clouds for online request processing, e.g. Web searches or online shopping. In the present work, our main use of Cloud computing technology is in constructing spatial indexes, which has batch processing characteristics. Thus, MapReduce-based Cloud computing is a natural choice for the problem we tackle.

R-tree Indexes R-trees [1] are extensively used in spatial databases. Many variants exist that optimize various aspects of the data structure for improving retrieval performance of a *single* R-tree [2] [3]. In our own work [8], we tackled the scalability issue of R-tree index constructions in MapReduce by concurrently indexing individual database partitions. In this paper, we employ another data partitioning strategy based on X-means clustering [9], and empirically study query performance on a real database under different partitioning schemes. Existing R-tree optimization techniques can be leveraged during the construction of individual R-tree indexes.

Machine Learning There has been research in implementing scalable platforms for machine learning algorithms. Chu et al. [12] used the MapReduce model to implement a large variety of machine learning algorithms on multi-core computers, including among others K-means, logic regresion, naïve Bayes, and SVM. In the open source community, the Mahout[4] project aims at building scalable MapReduce-based machine learning libraries. Currently Mahout supports four task categories: Recommendation mining, clustering, classification, and frequent itemset mining.

[3]http://aws.amazon.com

[4]http://mahout.apache.org

III. Spatial Data Processing in the Cloud

Geodatabases store spatial attributes, e.g. points representing object locations, or polygons that approximate shapes of objects. Furthermore, a rich set of non-spatial attributes accompany spatial objects. For example, in a geodatabase of property parcels, home owner names and physical street addresses may be stored in text attributes in addition to property polygons. In the rest of the paper, we assume a spatial database $D = \{o_1, o_2, ..., o_n\}$ where every $o \in D$ has two types of attributes: spatial and textual.

Geographical data is periodically received or updated in geodatabases. Fresh data has to be first indexed before users can start issuing queries. Most geodatabases employ R-trees for spatial indexing, and inverted files for indexing text attributes. The indexing phase may take a substantial amount of time, depending mainly on two factors: a) The size of the database (e.g. several R-tree insert operations or lengthy pre-processing steps in bulk constructions may be required), and b) The size of the lexicon (e.g. typically, a significant amount of time is spent sorting terms to compose posting lists in an inverted file).

Cloud computing represents a viable alternative to scale out data-intensive processing to lower indexing times. Figure 1 shows the general architecture of our spatial data management system in the Cloud, and query processing subsystem. First, a spatial dataset D is uploaded to HDFS in the Cloud for parallel processing via Hadoop MapReduce jobs – a compound of map and $reduce$ functions. Data indexing is performed by three types of jobs executed in sequence:

1) Data Partitioning – These jobs prepare the data for parallel processing by partitioning D according to spatial proximity. Data partitions are non-overlapping subsets d_j such that $D = \left\{\bigcup_{j=1..R} d_j\right\}$, where R is a parameter that defines the number of partitions. Section IV discusses two main partitioning strategies in MapReduce and a method to estimate R.
2) R-tree Index – Job mappers take as input the data partitions $\{d_j\}$ built in the previous phase and pass them along to reducers. R-tree constructions are executed in parallel by R reducers, one for each partition d_j. Each reducer outputs an R-tree r_j built on its input d_j.
3) Inverted File – Hadoop jobs process d_j subsets and build their inverted files. The output is a set of inverted files i_j, one for every input d_j.

R-tree indexes and inverted files may be built simultaneously. For the specific hybrid spatio-text index structure we use in our experiments, inverted files include references to R-tree nodes [4]. Thus, we built them in sequence. After data is indexed in the Cloud, indexes are downloaded to our local site for query processing. A spatial query processor logically consolidates the individually built R-trees by grouping their root nodes under a single node (or a few of them if node

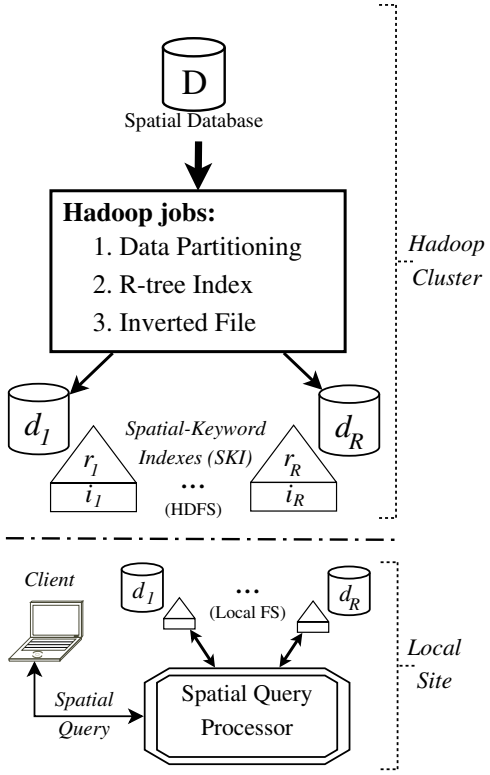

Figure 1. Spatial data indexing in Hadoop and local query processing.

capacity is exceeded) that constitutes the root of a big R-tree index on dataset D. The big R-tree and inverted files are combined into a hybrid logical structure for efficient spatial query processing.

The query processor uses the query resolution algorithm described in [4] with one main difference. At the beginning, all individual root nodes of the big R-tree are added into the priority queue regardless of whether they satisfy the Boolean text predicates or not. Evaluating node satisfiability may be expensive and not necessary; for example, only a few sub-trees (those nearby query location) may be visited during query processing. Thereafter, the algorithm continues as if it would be accessing a single index: best-first traversal on R-tree nodes (may retrieve R-tree nodes of any sub-tree) plus pruning of non-satisfying node entries w.r.t. Boolean text constraints.

R-tree index constructions in MapReduce was studied previously in [8]. Similarly, there are well known inverted file solutions in MapReduce [7]. In the next section, we focus on data partitioning strategies.

IV. Spatial Data Partitioning in MapReduce

Data partitioning is a key issue that has to be addressed in parallel processing approaches. The choice of a partitioning schemes is usually application dependant and driven by how the data is intended to be retrieved. We consider two main data partitioning strategies motivated by typical geographical database spatial query requirements, such as nearest neighbor and range spatial queries.

Algorithm 1 MapReduce X-means clustering algorithm

```
1: procedure MAP(id, Object o)
2:     d ← ∞
3:     p ← ∅
4:     for all c ∈ Centroids do                ▷ Compute nearest
5:         if dist(o, c) < d then              ▷ centroid to o
6:             d ← dist(o, c)
7:             p ← c
8:         end if
9:     end for
10:    EMIT(p, o)          ▷ Assign o to its nearest centroid
11: end procedure

12: procedure REDUCE(c, list[o_1, o_2, ...])
13:    c_p ← Re-compute cluster centroid on list[o_1, o_2, ...]
14:    if size[list[o_1, o_2, ...]] > MIN_SIZE then
15:        (c_1, c_2) ← Run 2-means on list[o_1, o_2, ...]
16:        BIC_1 ← BIC score of clustering {c_p}
17:        BIC_2 ← BIC score of clustering {c_1, c_2}
18:        if BIC_2 > BIC_1 then
19:            EMIT({c_1, c_2})          ▷ Child clusters outlive
20:        else
21:            EMIT({c_p})               ▷ Parent cluster outlives
22:        end if
23:    else          ▷ Parent cluster is too small for splitting
24:        EMIT({c_p})
25:    end if
26: end procedure
```

In [8] we explored dividing a spatial dataset in MapReduce based on Z-order values of objects' geographical coordinates. Z-order values map d-dimensional coordinates to a single-dimensional value, which can be conveniently sorted. This technique has two main advantages. First, it has linear complexity on the mappers' input; partition boundaries are computed once for all mappers in $O(m \log m)$ time, where m is a small percentage of n [8]. Second, it allows to generate almost equally-sized partitions. Its main drawbacks are that spatial locality is not always well preserved, e.g. objects nearby may be placed in different partitions, and the number of partitions is a parameter in the algorithm.

As an alternate method, we use X-means iterative clustering algorithm [9] to divide the space in clusters of spatial objects. X-means extends the popular K-means clustering technique with good approximates for the parameter K – the number of clusters. The algorithm uses Bayesian Information Criterion (BIC) scores to rank clusterings found by varying the number of clusters over a predetermined range. The main assumption in using BIC scores is that the data

follows a Gaussian distribution. We use X-means to find a clustering of D and consider every cluster found as an individual partition.

Algorithm 1 show the pseudo-code of the map and $reduce$ routines in the MapReduce X-means clustering algorithm. Mappers compute distance of objects $o \in D$ to every centroid $c \in Centroids$ (lines 4–9) and emit as intermediate output the pair (p, o) in line 10, where p is the closest centroid to object o; in the experiments, we used as distance function $dist(o, c)$ the great-circle distance with an estimation of the Earth radius. The set of $Centroids$ is stored in the cluster's globally accessible distributed file system (HDFS). Reducers receive a list of objects belonging to a given cluster c, and its centroid is re-computed (line 13). If the cluster size is sufficiently large, it is considered for splitting in two smaller clusters in lines 15–22; otherwise, the re-computed cluster centroid is emitted as final output (line 24). BIC scores of parent (BIC_1) and child (BIC_2) clusterings are compared to make a decision about splitting the parent cluster or not in line 18. Finally, centroids of the clustering with higher BIC score are emitted for the next iteration. The map and $reduce$ routines are iterated until the number of clusters in the global clustering exceeds the maximum allowed. The clustering with the highest global BIC score is chosen as the final clustering.

Note that the $reduce$ method assumes the entire cluster set can be loaded into memory. For clusters with very large number of objects, this may not be true. In the latter case, reducers can compute only BIC_1 and write objects back to HFDS to execute 2-means (and compute BIC_2) in a subsequent MapReduce job. In our experiments, we set the minimum number of clusters to a large enough value so objects of any cluster can fit in main memory.

The advantage of the clustering method is that an adequate number of partitions (i.e. the number of reducers R) is automatically estimated. On the other hand, partition sizes may vary considerably, and iterating on the clustering algorithm might be computationally expensive. To alleviate the first problem, we impose a minimum cluster size before a cluster is considered for splitting to avoid generating very small partitions. On the second issue, our experimental results show that the additional clustering overhead, which is incurred only once, pays off by lowering query response times.

V. Experimental Study

The experimentation was divided in two parts. In the first part, two spatial-keyword indexes on a real database were built in a Hadoop cluster (CluE). Spatial partitioning variants discussed in Section IV were used in subdividing the database for parallel indexing: Z-order based partitioning (ZO) and X-means clustering (XM). In the second part, query response times were measured using ZO and XM indexes running on a local machine.

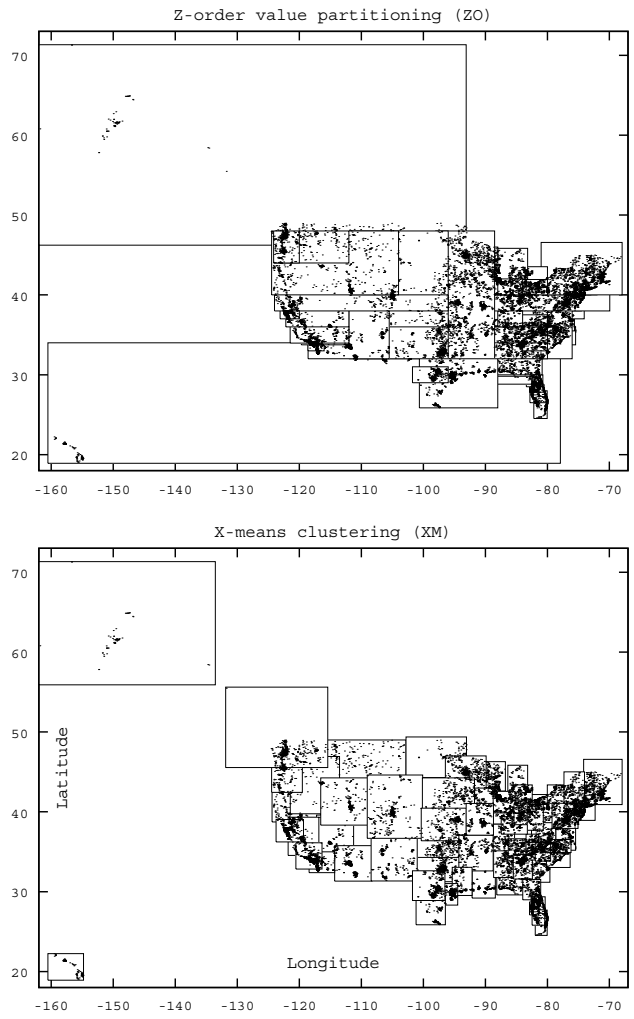

Figure 2. Data partitioning of United States property parcels based on Z-order values (top) and X-means clustering (bottom) with 62 partitions.

Spatial Database We used a spatial database of United States property parcels (USP). The dataset has a point representing the parcel location in geographical coordinates, and 17 other non-spatial attributes, including among others parcel id, owner's name, street address, and parcel type. The dataset has 110 million objects and around 12.8 million distinct textual terms.

Hadoop Cluster (CluE) The CluE cluster has around 400 commodity nodes (mostly homogeneous 8GB RAM and two 400GB disks) running Hadoop framework version 0.20 [10]. The Hadoop cluster is shared with other research institutions.

A. Spatial-Keyword Index Construction

We ran the MapReduce X-means clustering (Algorithm 1) with 118 *mappers* and initial random cluster centroids. The number of clusters (*reducers*) were varied between 30 and 100. We set the minimum cluster size as 1 million whenever

a local cluster was considered for splitting in X-means. We observed that the number of clusters consistently became stable around 60 after a few iterations. Clusterings with more clusters just marginally improved BIC scores. We selected a clustering with 62 cluster as the best clustering.

The number of clusters found by X-means was used as parameter for the Z-order partitioning scheme. Figure 2 visualizes the clustering found by X-means (with 62 clusters) after 7 iterations and the partitions determined by Z-order values; points are sampled object locations and partitions are represented by their minimum bounding rectangles (MBR). Although MBR overlapping is inevitable, we can observe in Figure 2 (bottom) that XM better approximates the distribution of the data and reduces overlapping. ZO incurrs in a lot more MBR overlapping, especially in denser areas like the eastern cost. It is a known result that excessive MBR overlapping hinders retrieval performance [2] [3]. In the next section, we empirically measure the effect of additional MBR overlapping incurred by ZO partitioning scheme.

Spatial-keyword indexes were built individually for ZO and XM partitions, and downloaded to our local site. Since the Hadoop cluster we used is shared with other researchers, it is hard to measure index construction times with accuracy. Elapsed times vary according to the cluster activity. Nonetheless, we observed that index construction times with ZO partitioning strategy were completed in about 40 minutes, while indexing with XM partitioning fluctuated between 70 and 100 minutes. The dominant time factor in XM was the clustering phase (around 50%).

B. Query Processing

We evaluated response times of queries using indexes build in the Cloud with both partitioning techniques ZO and XM. Queries were processed according to the algorithm in [4] that combines R-trees and inverted files to prune the search space. We call ZO (XM) index to the index built with the ZO (XM) partitioning technique. Queries were run in an Intel Xeon E5520 machine with 16GB physical memory and two quad-core processors at 2.27GHz. Database and indexes were stored on a 6-disk RAID-5 array attached directly to the host. The main focus of this experiment is in measuring the number of R-tree nodes accessed during query processing, and the overall response time.

Query Workload Nearest neighbor (*NN*) queries with non-spatial constraints were generated as follows. A query location l was randomly chosen from the USP space. Non-spatial query constraints were composed by randomly selecting one, two and three terms from the USP lexicon. The special case of queries with no text constraints, i.e. conventional *NN* queries, was also considered. We retrieved the top-50 *NN* objects to l that match the conjunctive Boolean constraint on non-spatial attributes.

Figure 3 shows the minimum, maxium, and median measures over 100 top-50 *NN* queries with zero, one, two and three terms in their non-spatial constraints. We preferred median over average due to large dispersion of values, particularly in ZO index results. The experimental results confirmed what we expected. For regular *NN* queries (no constraints), the median of R-tree nodes accessed by ZO index is around 70% higher than XM index as can be seen in Figure 3 (top) due to increased MBR overlapping in ZO index. Response times follow similar behavior for those queries in Figure 3 (bottom). Queries with 1-term constraints still incurr in extra R-tree node retrievals for ZO index compared to XM index. Notably, queries with ZO index present a large gap between the minimum and maximum values on both R-tree nodes retrieved and response times. For 2-term and 3-term queries, R-tree nodes accessed are comparable between ZO and XM indexes, which indicates that matching objects are more scattered in the search space.

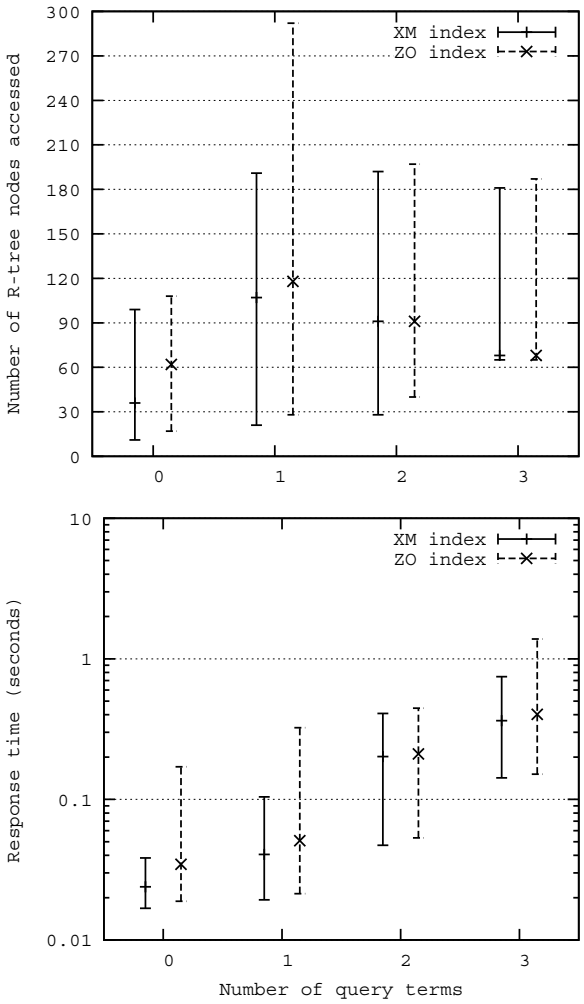

Figure 3. Performance metrics minimum, maximum, and median over 100 top-50 spatial queries with zero up to three query terms in the non-spatial constraint. Number of R-tree nodes accessed during query processing (top) and response times (bottom, *y*-axis is in logarithmic scale) are shown.

Response times are mostly dominated by accesses to inverted files, and thus are proportional to the number of query terms. Overall, MBR overlapping incurred by ZO index negatively affects retrieval performance. Parallel computing in the Cloud helped scale out X-means clustering algorithm, which may be rather expensive for large-scale datasets, to achieve better query performance.

VI. Summary

In this work we presented a case of scaling out spatial data indexing, a critical data-intensive process, in geographical databases. We evaluated parallel index constructions on a 110-million sized database of property parcels in the United States. Two main data partitioning variants were explored, and the effects on spatial query performance assessed. Our work supports Cloud computing in scaling out computational and data intensive processing required by geodatabases.

As future work, we plan to consider other clustering algorithms (particularly, for clusters of arbitrary shapes) with the goal of further enhancing query response times.

Acknowledgment

This research was supported in part by NSF grants IIS-0837716, CNS-0821345, HRD-0833093, IIP-0829576, IIP-0931517, CNS-0837556, CNS-0426125. The authors thank "First American Spatial Solutions" (Austin, Texas 78758) for supplying the dataset used in the experiments.

References

[1] A. Guttman, *R-trees: a dynamic index structure for spatial searching*, In Proceedings of the 1984 ACM SIGMOD international Conference on Management of Data, 1984.

[2] N. Beckmann and B. Seeger, *A revised r*-tree in comparison with related index structures*, In Proceedings of the 35th SIGMOD international Conference on Management of Data, Providence, RI, 2009.

[3] I. Kamel and C. Faloutsos, *Hilbert R-tree: An improved R-tree using fractals*, In Proceedings of the International Conference on Very Large Databases, 500-509, 1994.

[4] A. Cary, O. Wolfson and N. Rishe, *Efficient and Scalable Method for Processing Top-k Spatial Boolean Queries*, In Proceedings of the 22nd international Conference on Scientific and Statistical Database Management, Heidelberg, Germany, June 2010, in press.

[5] J. Zobel and A. Moffat, *Inverted files for text search engines*, In ACM Comput. Surv. 38, 2, 2006.

[6] J. Dean and S. Ghemawat, *MapReduce: simplified data processing on large clusters*, In Proceedings of the 6th Conference on Symposium on Opearting Systems Design and Implementation, Volume 6, San Francisco, CA, 2004.

[7] J. Lin and C. Dyer, *Data-Intensive Text Processing with MapReduce*, Synthesis Lectures on Human Language Technologies, Vol. 3, 1, pp. 1-177, 2010.

[8] A. Cary, Z. Sun, V. Hristidis and N. Rishe, *Experiences on Processing Spatial Data with MapReduce*, In Proceedings of the 21st international Conference on Scientific and Statistical Database Management, New Orleans, LA, 2009.

[9] D. Pelleg and A.W. Moore, *X-means: Extending K-means with Efficient Estimation of the Number of Clusters*, In Proceedings of the Seventeenth international Conference on Machine Learning, 2000.

[10] National Science Foundation, Cluster Exploratory (CluE) Program. http://www.nsf.gov/pubs/2008/nsf08560/nsf08560.htm

[11] S. Ghemawat, H. Gobioff and S.T. Leung *The Google file system*, In SIGOPS Oper. Syst. Rev., 37, 5, pp. 29–43, 2003.

[12] C.-T. Chu, S.K. Kim, Y.A. Lin, Y. Yu, G. Bradski, A. Ng and K. Olukotun, *Map-Reduce for machine learning on multicore*, In Proceedings of Neural Information Processing Systems Conference (NIPS). Vancouver, Canada, 2006.

[13] Hadoop MapReduce. http://hadoop.apache.org/mapreduce

[14] Vertica Analytic Database. http://www.vertica.com/MapReduce

[15] Aster Data Systems. http://www.asterdata.com/resources/mapreduce.php