Processing All *k*-Nearest Neighbor Queries in Hadoop

Takuya Yokoyama[1], Yoshiharu Ishikawa[2,1,3], and Yu Suzuki[2]

[1] Graduate School of Information Science, Nagoya University, Japan
[2] Information Technology Center, Nagoya University, Japan
[3] National Institute of Informatics, Japan
{yokoyama,suzuki}@db.itc.nagoya-u.ac.jp, y-ishikawa@nagoya-u.jp

Abstract. A k-nearest neighbor (kNN) query, which retrieves nearest k points from a database is one of the fundamental query types in spatial databases. An *all k-nearest neighbor query* (AkNN query), a variation of a kNN query, determines the k-nearest neighbors for each point in the dataset in a query process. In this paper, we propose a method for processing AkNN queries in *Hadoop*. We decompose the given space into cells and execute a query using the MapReduce framework in a distributed and parallel manner. Using the distribution statistics of the target data points, our method can process given queries efficiently.

1 Introduction

An *all k-nearest neighbor query* (an *AkNN query* for short) is a variation of a k-nearest neighbor query and determines the k-nearest neighbors for each point in the given dataset in one query process. It is a useful operation for batch-based processing of a large distributed point dataset. Consider, for example, a location-based service which recommends each user his or her nearby users, who may the candidates of new friends. Given that users' locations are maintained by the underlying database, we can generate such recommendation lists by issuing an AkNN query (e.g., $k = 5$) on the database. In a centralized database environment, we can use the existing AkNN algorithms [3, 5, 11].

Although efficient algorithms for AkNN queries are available for centralized databases, we need to consider to support distributed environments where the target data is managed in multiple servers in a distributed way. User location information of a location-based service in the above example may be distributed in many servers. In such a case, we need to consider to use cloud computing technologies for efficiently executing queries. Especially, *MapReduce*, which is a fundamental framework for processing large-scaled data in distributed and parallel environments, is a promising method for enabling scalable data processing. In our work, we focus on the use of Apache *Hadoop* [6] since it is quite popular software for MapReduce-based data processing.

In this paper, we propose a method for efficiently processing AkNN queries in Hadoop. The basic idea is to decompose the target space into smaller cells.

At the first phase, we scan the entire dataset and get the summary of the point distribution. According to the information, we determine an appropriate cell decomposition. In the following phases, we determine k-NN objects for each data points by considering the maximal range in which possible k-NN objects are located.

2 Related Work

2.1 MapReduce and Hadoop

In this work, we assume the use of the distributed and parallel computing framework *Hadoop* [6, 10]. Here we briefly describe *MapReduce* [4], which it the foundation of Hadoop data processing. The data processing in Hadoop is based on input data partitioning; the partitioned data is executed by a number of tasks executed in many distributed nodes. There exist two major task categories called *Map* and *Reduce*. Given input data, a *Map* function processes the data and outputs key-value pairs. Based on the Shuffle process, key-value pairs are grouped and then each group is sent to the corresponding Reduce task. A user can define own Map and Reduce functions depending on the purpose and they are applied to the input data. The input and output formats of these functions are simplified as key-value pairs. Using this generic interface, the user can focus on his own problem and does not have to care how the program is executed over the distributed nodes.

An AkNN query is regarded as a kind of a *self-join* query. Join processing in the MapReduce framework has been studied intensively recent years [1, 7], but generally speaking, MapReduce only supports equi-joins; development of query processing methods for non-equi joins is one of the interesting topics on the MapReduce technology [9].

2.2 All k Nearest Neighbor Queries

An *all k-nearest neighbor query* (an AkNN query for short) is a query to determine the k-nearest neighbors for each data point in the given dataset. An example of a location-based service is described in Section 1, but it is also useful in other applications. For example, [2] uses an AkNN query for the preprocessing of the succeeding data mining process.

For AkNN queries, there are proposals that use R-trees and space-filling curves [3, 5, 11], but they are limited for the use in a centralized environment. For processing AkNN queries in Hadoop in an efficient manner, we need to develop a query processing method that effectively uses the MapReduce framework.

3 Distributed and Parallel Processing Based on Cell Decomposition

3.1 Basic Idea

We now describe the basics of our AkNN query processing method. We consider two dimensional points with x and y axes. Basically, we decompose the target space into $2^n \times 2^n$ small cells. The constant n determines the granularity of the decomposition. Since the k-nearest neighbor points for a data point is usually located in the nearby area of the point, we can expect that most of the k-NN objects are found in the nearby cells. Therefore, a simple idea is to classify data points into the corresponding cells and we compute candidate k-NN points for each point. The process can be parallelized easily and is suited to the MapReduce framework.

However, we may not be able to determine k-NN points at one step; we need to perform an additional step for such a case. Data points in other nearby cells may belong to the k-nearest neighbors. To illustrate this problem, consider Fig. 1, where we are processing an AkNN query for $k = 2$. We can find 2-NN points for `A` by only investigating the inside of cell `1` since the circle centered at `A` and tightly covers 2-NN objects (we call such a circle a *boundary circle*) does not overlap the boundary of cell `0`. In contrast, the boundary circle for `B` overlaps with cells `1`, `2`, and `3`. In this case, there is a possibility that we can find 2-NN objects in the three cells. Therefore, an additional investigation is necessary.

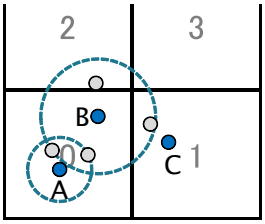


Fig. 1. Cell-based k-NN processing ($k = 2$)

The idea is simple but there is a problem; we may not be able to draw the boundary circle for a point. Consider point `C` in Fig. 1. For this point, there is only one (less than k) point in cell `1`. Thus, we cannot draw the boundary circle. We solve the problem in the following subsection.

3.2 Merging Cells Using Data Distribution Information

We solve the problem described above by prohibiting the situation that there are not enough points in each cell. The idea is very simple. We first check the number of points within each cell. If we find a cell with less number of points, we merge the cell with the neighboring cells to assure that the number of points

in the merged cell is greater than or equal to k. After that, we can draw the boundary circle.

The outline of the idea is illustrated in Fig. 2, where 4×4 decomposition is performed. At the first step, we count the number of points in each cell. Then, we merge the cells with less number of objects with the neighboring cells. In our method, we employ the hierarchical space decomposition used in quadtrees [8]. When we perform merging, we merge four neighboring cells which correspond to the same node at the parent level.

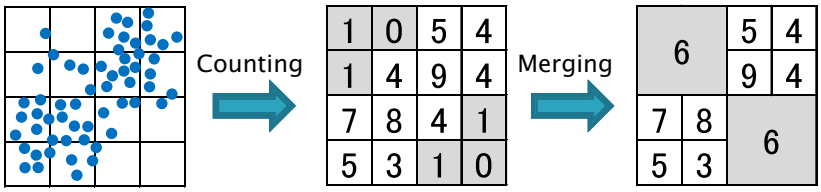


Fig. 2. Cell merging using distribution information

The problem of this approach is that we need to perform an additional counting phase before the nearest neighbor computation. However, it can simplify the following steps. The distribution information is useful in other ways. If we can know there is no points in a cell beforehand, we do not need to consider the cell in the following processes. As shown in the experiments, the cost of the counting is relatively small compared to the total cost.

4 Details of Query Processing

We describe the detail of the query processing method based on the idea shown in the previous section. It consists of four steps. We assume that the input dataset is a set of records, which has the format `<id, x, y>`. The parameters n and k are specified initially by the user.

4.1 MapReduce1: Getting Distribution Information and Cell Merging

In this step, we decompose the entire space into $2^n \times 2^n$ cells and count the number of points that fall in each cell. The counting process can be directly implemented as a MapReduce process:

- **Map**: It receives the input with the format shown above and computes the cell number for each data point based on its coordinates. The format of the output record is a key-value pair `<cell_id, 1>`.
- **Reduce**: It receives a set of records which has the same cell ids and sum the value parts of the records. The output is with the format `<cell_id, sum>`.

Note that the Shuffle process classifies the output records of the Map function and records with the same ids are sent to a corresponding node.

Although this is the basic processing method, we can further improve the performance. In Hadoop, we can use an optional *Combiner* in the MapReduce process. A Combiner is used when we aggregate the intermediate data within a Map task before sending the output of the Map function to the following Shuffle process. It can reduce the size of the intermediate dataset. If we use this option, MapReduce1 is modified as follows:

- **Map**: Same as above.
- **Combiner**: It receives a set of records with the same cell ids and sums the value parts of the records. The output is with the format `<cell_id, sum>`.
- **Reduce**: Same as above.

After MapReduce1, we perform cell merging. For this process, the scalability is not required since we just consider statistics values so that we perform the computation in a node. This procedure receives an input with the format shown in the middle of Fig. 2 and merges low-number cells with the neighboring cells based on the hierarchical quadtree structure in a bottom-up manner. The cell merging procedure outputs the mapping information how each cell id corresponds to a new cell id in the merged cell decomposition. This information is read in the following MapReduce processes as additional data.

4.2 MapReduce2: First Stage of All$k$NN Computation

In this step, we collect input records for each cell and then compute candidate kNN points for each point in the cell region. The Map and Reduce functions are summarized as follows:

- **Map**: It receives the original data points and computes the corresponding cell id, and then output records with the format `<cell_id, id, coord>`, where `id` is the point id and `coord` is the coordinates of the point.
- **Reduce**: It receives records corresponding to a cell; the records has the format `<id, coord>`. The Reduce function calculates the distance for each combination of two points in the cell and computes the k-NN points for each point in the cell. The output records have the format `<id, cood, cell_id, kNN_list>`, where `id` is used as the key and `kNN_list` is the list of the kNN points for point `id` and has the format $[\langle o_1, d_1\rangle, \ldots, \langle o_k, d_k\rangle]$, where o_i is the i-th NN point and d_i is its distance.

Note that the Shuffle operation collects records with the same cell ids into one node.

4.3 MapReduce3: Updating k-NN Points

In this step, we use the idea described in Section 3 that uses the notion of a boundary circle. If it is necessary, we perform an additional process and updates k-NN points for the points. The MapReduce process is outlined as follows:

- **Map**: It receives the result of MapReduce2. For each point, we first compute the bounding circle. There are two cases for the following process:
  1. *The bounding circle does not overlap with other cells*: In this case, we do not need to perform a further process and the k-NN points are fixed. Thus, we output key-value pairs with the format `<cell_id, id, coord, kNN_list, true>`, where `cell_id` is the key and the last item `true` is a flag to denote the process is finished. In the following Reduce operation, we do not perform any further processing for the records in which the flags are set; it only converts the record format.
  2. *The bounding circle overlaps with other cells*: Since there is a possibility that neighboring cells may contain k-NN points, we need an additional process for checking. For preparing the additional phase, we output key-value pairs with the form `<cell_id', id, coord, kNN_list, false>`, where the key `cell_id'` is the cell id of one of the overlapped cells and the last item `false` represents that the process is not finished. If multiple cells overlap with the bounding circle, we output multiple corresponding key-value pairs. In the example of Fig. 1, we generate three pairs `<1, B, ...>`, `<2, B, ...>`, and `<3, B, ...>` for `B`.

  In this Map process, the procedure also outputs the second type of records with the format `<cell_id, id, coord>`, which are used for the following k-NN points computation and the key is `cell_id`.
- **Reduce**: The Shuffle operation sends records with the same cell ids to the corresponding node, and the records are treated as the input of this Reduce function. Since different types of records exist in the input, it first classifies the records and then update k-NN points for the points that require additional checks. We need to perform additional distance computation between some limited number of record pairs. The output is a set of records with the format `<id, coord, cell_id, kNN_list>`, where `id` is the key.

4.4 MapReduce4: Integrating k-NN Lists

We need this step because we may have multiple updates of k-NN points for a point. We should merge these k-NN lists and finally construct the result k-NN list. The MapReduce process is summarized as follows:

- **Map**: It receives the result of the former step and then outputs the k-NN list. The format of output records is `<id, kNN_list>`, where `id` is the key.
- **Reduce**: It receives the records with the same keys (the format is described above) and then it creates the integrated list with the format `<id, kNN_list>`.

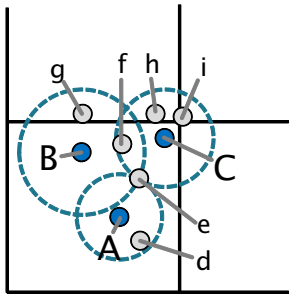
If we needed to investigate multiple cells for a point, the Shuffle process groups the multiple outputs and then sends them to the Reduce function.

Based on the above four steps, we can finally determine the k-NN points for each point and we can quit the process.

4.5 Illustrative Example

We show an example of MapReduce1 to 3 steps that finds AkNN points. Figure 3 shows the distribution of points and we focus on points `A`, `B`, and `C`. The three points are the representative example patterns:

- `A` (the bounding circle does not overlap with other cells): k-NN points are determined at MapReduce2 step.
- `B` (the bounding circle overlaps with one cell): We can determine k-NN points at MapReduce3 step by investigating additional cell `2`.
- `C` (the bounding circle overlaps with multiple cells): We investigate additional cells `1`, `2`, and `3` at MapReduce3 step. Then we integrate their results at MapReduce4 step and determine the k-NN points.



id	coord
A	(60, 40)
B	(40, 80)
C	(90, 90)
d	(75, 30)
e	(75, 65)
f	(65, 85)
g	(40, 105)
h	(80, 105)
i	(105, 105)

Fig. 3. k-NN example ($k = 2$)

Figure 4 illustrates the execution steps of the entire MapReduce steps. We can see that the k-NN lists for points `A`, `B`, and `C` are incrementally updated and finally fixed.

5 Experiments

We have implemented the method described in Section 4. In this section, we evaluate the performance of the MapReduce program running in a Hadoop environment.

5.1 Datasets and Experimental Environment

The experiments are performed using two synthetic datasets: the datasets `1M` and `10M` consist of 1,000,000 and 10,000,000 points in the target space, respectively. Their file sizes are 34MB and 350MB[4].

[4] We have evaluated the performance using a real map dataset. However, since the number of entries is small (no. of points = 53,145), we found that the overhead dominates the total cost and there is no merit to use Hadoop. Therefore, we used the synthetic dataset here to illustrate the scalability of the method.

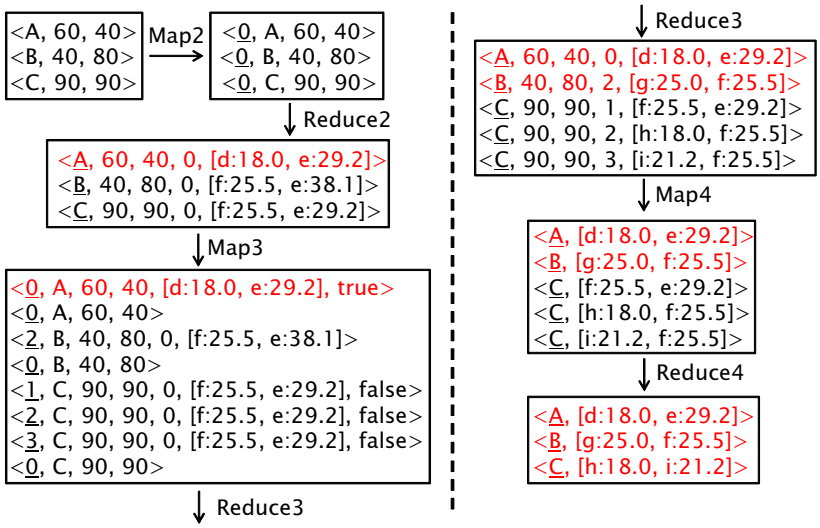


Fig. 4. Processing AkNN query on example dataset

We use three nodes of Linux 3.0.0-14-server (Ubuntu 11.10) with Intel Xeon CPU (E5620 @ 2.40GHz). Since each CPU has 4×2 cores, we have 24 cores in total. The system has 500GB storage and the servers are connected by 1G bit Ethernet. We run Hadoop version 0.20.203.0 in the system. The number of replicas is set to 1 since we do not care failures in this experiment. The number of max number of Map tasks and Reduce tasks are set to 8.

5.2 Experiment 1: Changing Number of Reduce Tasks

To evaluate parallel processing behavior, we run queries by changing the number of Reduce tasks as 1, 2, 4, 8, 12, 16, and 24. The granularity parameter is set to $n = 8$ and the parameter k is set to 5. The results are shown in Figs. 5 and 6. Total execution time consists of "Cell Merging" and MapReduce2 to 4, where "Cell Merging" means that the time required until the cell merging step; it includes the cell merging process and its preceding MapReduce1 step.

Figure 5 shows the result for dataset `1M`. In this case, the execution time takes the minimum when the number of Reduce tasks is 8 and the cost increases for the large number of Reduce tasks. The reason is that the size of the input is small for this setting and the overhead of parallelization has more impacts on the performance. In this case, we can determine the final k-NN points for 16% of the data points at MapReduce2 step. It means that we need to perform additional steps for 84% of the data points in MapReduce3 and 4 phase.

Figure 6 shows the case for dataset `10M`. In this case, the execution time decreases as the number of Reduce tasks increases. Since the density of points in the target space increases 10 times larger than that of dataset `1M`, we need to

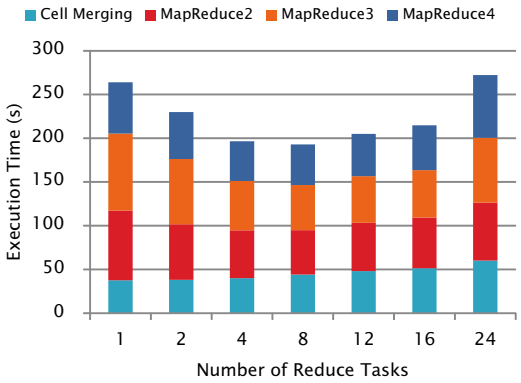


Fig. 5. Execution time for different no. of Reduce tasks (`1M`, $n = 8$ and $k = 5$)

perform additional steps only for 35.6% of the data points in the MapReduce3 phase. This is because we can determine k-NN points by checking only one cell in MapReduce2 step.

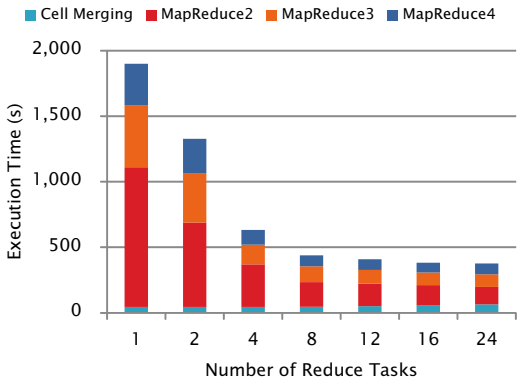


Fig. 6. Execution time for different no. of Reduce tasks (`10M`, $n = 8$ and $k = 5$)

As we can observe the figures, the execution time for "Cell Merging" step for dataset `10M` slightly increases as the increase of the number of Reduce tasks. The reason is that the use of Combiner in MapReduce1 step can reduce the amount of data size to be given to the Shuffle operation and the amount of inputs for Reduce tasks. Since the data processing cost is low, the increase of the number of Reduce tasks becomes the overhead.

5.3 Experiment 2: Changing k Values

In this experiment, we investigate how the performance changes for different k values. The granularity parameter is set to $n = 8$ and the number of Reduce

tasks is 24. The results shown in Fig. 7 for dataset `10M`. We omit the case for dataset `1M` since the overall trend is same as the case of `10M`.

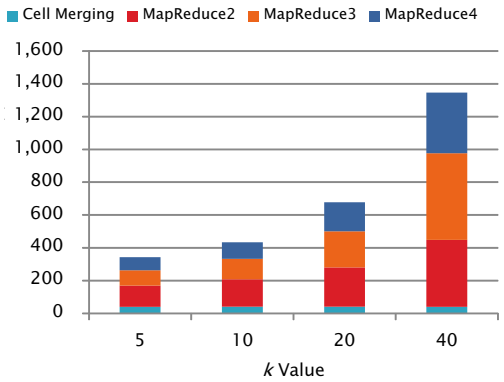


Fig. 7. Execution time for different k values (`10M`, $n = 8$)

As the figure illustrates, the processing time increases as the k value increases. Especially, the increases of the cost for MapReduce3 and MapReduce4 are large. The reason is that a large k value results in a large size of each intermediate record, and it results in the increase of data processing time. In addition to that, since the radius of a boundary circle becomes large, we need to investigate more data points in MapReduce3 and MapReduce4 steps.

Table 1 shows the detail of the processes. As shown in the table, the increase of k results in the increase of the average size of intermediate records, which contain k-NN lists. Since the k value directly influences the list length, the overhead becomes larger. The table also shows the ratio of points which require additional steps (MapReduce3 and 4). As shown in the table, we need more steps for large k values and it influences the total cost.

Table 1. Statistics of experiments for different k values

	$k = 5$	$k = 10$	$k = 20$	$k = 40$
Average size of a record (byte)	128	243	473	933
Ratio of points which need additional steps	35.6%	48.7%	64.5&	81.9%

5.4 Summary of Experiments

Based on the experiments, we have observed that the proposed AkNN query processing method can reduce the processing time by parallel processing, especially

for a large dataset (Experiment 1). In addition, it is observed that the increase of the k value results in the total processing cost (Experiment 2).

In our method, we incorporated a preparation step to obtain the overall distribution of the points in the target space. As shown in the experimental results, this process, including the cell merging cost, is quite efficient compared to other processes of the algorithm. The observation is more clear especially when the dataset size is large. The preprocessing can simplify the algorithm because the strategy based on a boundary circle becomes simpler. Thus, the benefit of the first phase is larger than the cost of the process.

6 Discussion: AkNN Queries on Two Different Inputs

Since our proposed method is simple, we would be able to extend the algorithm for different and generalized cases. Especially, we can consider to calculate AkNN points for two different datasets. For example, given two datasets `A` and `B`, assume that we would like to calculate the k-NN points for each A point from `B` points. To cope with this change, our algorithm requires modifications; MapReduce2 and 3 should read two datasets `A` and `B` and compute the distances between both datasets. The cell merging method should be revised to consider the distribution information of two datasets.

To improve distance computation time of the approach, we can employ the idea used in the Reduce function of MapReduce3 in the proposed method. The Reduce function receives two types of datasets (update data of k-NN points and coordinate data for distance computation) and classifies them, and then performs distance computation. By applying similar classification for datasets `A` and `B`, we can perform k-NN computation for two different datasets.

Cell merging using distribution information of two datasets are slightly different from the original case. We judge whether we actually need to merge two cells based on two factors: 1) whether `A` points exist in the cell and 2) whether the number of `B` points in the cell is larger than or equal to k. Table 2 shows the decision table.

Table 2. Decision table when cell merging is necessary

Dataset A	Dataset B	Need Cell Merging?
Contains points	No. of points $\geq k$	No
Contains points	No. of points $< k$	Yes
No points	No. of points $\geq k$	No
No points	No. of points $< k$	No

In conclusion, we need to integrate the cell with neighboring cells only when there are `A` points in the cell and the number of `B` points fall in the cell is smaller than k. Note that we do not consider the cell with no `A` points because k-NN computation is not necessary for this case.

7 Conclusions

In this paper, we have proposed an AkNN query processing method in the MapReduce framework. By using cell decomposition, the method adapts the distributed and parallel query framework of MapReduce. Since k-NN points may be located outside of the target cell, we may need additional steps. We solved the problem by considering a boundary circle for the target point. In addition, to simplify the algorithm, we proposed to collect distribution statistics beforehand and the statistics is used for cell merging. In the experiments, we have investigated the behaviors of the algorithm for different parallelization parameters and different k values.

The future work includes how to estimate the appropriate number of cell decomposition granularity (n) by using statistics of the underlying data. In addition, the number of parallel processes is also an important tuning factor. As shown in the experiments, too many parallel processes may result in the increase of the total processing cost due to the overhead. If we can estimate these parameters accurately and adaptively, we would be able to achieve nearly optimal processing for any environments.

Acknowledgments

This research is partly supported by the Grant-in-Aid for Scientific Research (22300034) and DIAS (Data Integration & Analysis System) Program, Japan.

References

1. F. N. Afrati and J. D. Ullman. Optimizing joins in a map-reduce environment. In *Proc. EDBT*, pp. 99–110, 2010.
2. C. Böhm and F. Krebs. The k-nearest neighbour join: Turbo charging the KDD process. *KAIS*, 6(6):728–749, 2004.
3. Y. Chen and J. M. Patel. Efficient evaluation of all-nearest-neighbor queries. In *Proc. ICDE*, pp. 1056–1065, 2007.
4. J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *OSDI*, pp. 137–150, 2004.
5. T. Emrich, F. Graf, H.-P. Kriegel, M. Schubert, and M. Thoma. Optimizing all-nearest-neighbor queries with trigonometric pruning. In *Proc. SSDBM*, pp. 501–518, 2010.
6. The apache software foundation: Hadoop homepage. http://hadoop.apache.org/.
7. D. Jiang, A. K. H. Tung, and G. Chen. MAP-JOIN-REDUCE: Toward scalable and efficient data analysis on large clusters. *IEEE TKDE*, 23(9):1299–1311, 2011.
8. H. Samet. The quadtree and related hierarchical data structures. *ACM Computing Surveys*, 16(2):187–260, 1984.
9. R. Vernica, M. J. Carey, and C. Li. Efficient parallel set-similarity joins using MapReduce. In *Proc. SIGMOD*, pp. 495–506, 2010.
10. T. White. *Hadoop: The Definitive Guide*. O'Reilly, 2009.
11. J. Zhang, N. Mamoulis, D. Papadias, and Y. Tao. All-nearest-neighbors queries in spatial databases. In *Proc. SSDBM*, pp. 297–306, 2004.