MapReduce Based Location Selection Algorithm for Utility Maximization with Capacity Constraints

Yu Sun, Jianzhong Qi, Rui Zhang
Department of Computing and Information Systems
University of Melbourne, Australia
yusun.aldrich@gmail.com
jianzhong.qi@unimelb.edu.au
rui@csse.unimelb.edu.au

Yueguo Chen, Xiaoyong Du
Key Laboratory of Data Engineering and Knowledge Engineering
(Renmin University of China), MOE, China
chenyueguo@ruc.edu.cn
duyong@ruc.edu.cn

A Technical Report

April 2013

Abstract

Given a set of facility objects and a set of client objects, where each client is served by her nearest facility and each facility is constrained by a service capacity, we study how to find all the locations on which if a new facility with a given capacity is established, the number of served clients is maximized (in other words, the utility of the facilities is maximized). This problem is intrinsically difficult. An existing algorithm with an exponential complexity is not scalable and cannot handle this problem on large data sets. Therefore, we propose to solve the problem through parallel computing, in particular using MapReduce. We propose an arc-based method to divide the search space into disjoint partitions. For load balancing, we propose a dynamic strategy to assign partitions to reduce tasks so that the estimated load difference is within a threshold. We conduct extensive experiments using both real and synthetic data sets of large sizes. The results demonstrate the efficiency and scalability of the algorithm.

Chapter 1

Introduction

Location selection is a classic problem in operational research and has wide applications in decision support systems. For example, a urban planner may need to decide where to build a public car park or a hospital, a company executive may need to decide where to open a new branch office. In recent years, with the widespread use of Global Positioning Systems (GPS) and smartphones, location based social networks have become popular and location selection has found a new area of application.

In this report, we study a new location selection problem that has traditional as well as modern applications. Fig. 1.1(a) illustrates the problem. Let $c_1, c_2, ..., c_{13}$ be a set of office buildings and f_1, f_2, f_3 be a set of car parks. People work in the office buildings want to park their cars in their respective nearest vacant car parks. Since the number of parking lots in a car park is limited, some people may have to park faraway. We study where to build a new car park, so that after the new car park is built, the largest number of people can park in their nearest car park.

Similarly, let us assume a location based social network scenario. Fig. 1.1(a) denotes a board game group where $c_1, c_2, ..., c_{13}$ are group members while f_1, f_2, f_3 are the activity centers provided by the group organizers. The group members want to play games in their nearest activity centers, but an activity center has a capacity and cannot hold all group members. We study where to set up a new activity center, so that the largest number of group members can play board games in their nearest activity centers. On social networks there are many interest groups. Thus, this problem may be asked frequently and it needs an efficient solution.

The above motivating problems are modeled as the problem of ***l**ocation **s**election for **u**tility **m**aximization (LSUM)*: given a set of points C as the clients and a set of points F as the facilities, where each client c is served by her nearest facility and each facility f is constrained by a service capacity $v(f)$, the LSUM problem finds all the locations in the space on which if a new facility with a given capacity is established, the number of served clients by all facilities is maximized (in other words, the utility of the facilities is maximized). Here, every client c is associated with a weight, denoted by $w(c)$, which can be thought of as the number of clients that reside at the same site.

In the example shown in Fig. 1.1(b), let the weight of each client be 1, and the capabilities of f_1, f_2, f_3 be $4, 3, 3$, respectively. Then the current total service capacity,

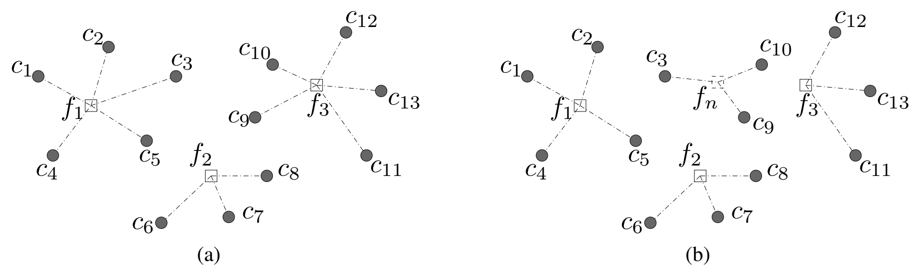


Figure 1.1: Problem examples

10, is less than the number of clients (a weighted sum), 13, and not all clients can be served by their respective nearest facilities. If a new facility with a capacity of 3 is set up on f_n . Then it will be the nearest facility of c_3, c_9 and c_{10}. Now the total capacity becomes 15 and every client gets served by her nearest facility. As a result, f_n is a problem answer.

In a recent study[23], the LSUM problem is formalized as the LSUM query and a branch-and-bound based algorithm is proposed to process the query. In the average case, the algorithm has a time complexity of $O(2^{\theta}|C|)$, where θ increases with $\frac{|C|}{|F|}$. In real applications, $|C|$ is usually very large and $|F|$ is relatively small. In this case, θ can get very large, which leads to prohibitively long query processing time. To overcome the inefficiency, we propose to leverage the power of parallel processing, in particular, the MapReduce framework, to achieve higher query processing efficiency.

There are two main challenges in applying the MapReduce framework for our problem: (i) How to disjointly divide the search space so that no area needs to be searched for more than once? (ii) How to assign balanced loads among different reduce tasks? In this report, we address these challenges and propose an efficient MapReduce based algorithm for the LUSM query.

The remainder of the report is organized as follows. Chapter 2 reviews related work. Chapter 3 describes preliminaries for the LUSM query. Chapter 4 gives an overview of our MapReduce based algorithm. Chapter 5 discusses how to partition the search space and Chapter 6 discusses load balancing. Experimental results are reported in Chapter 7 and the report concludes in Chapter 8.

Chapter 2

Related Work

2.1 Location Optimization

Location selection belongs to the category of location optimization problem, which is a classic problem in operational research. Various models [8, 9, 14, 15] have been proposed to solve location optimization problems of different settings. Some [14, 15] have taken the capacity constraints into consideration. However, in general, these models focus more on the demand and/or cost of setting up the facilities rather than maximizing the facility utility. Thus, we will not discuss these models further. Interested readers are refered to some recent reviews [5, 16, 21].

In the database community, studies in location optimization problems are mostly based on the ***b**ichromatic **r**everse **n**earest **n**eighbor (BRNN) query* introduced by Korn and Muthukrishnan [11]. Like many others [17, 18, 30, 33], the BRNN query is a variant of the *nearest neighbor (NN) query*. Given two object sets C and F and a query object from F, the BRNN query returns objects in C who perceive the query object as their nearest neighbor in F. The BRNN set of an object is also called the *influence set* of the object. Based on the influence set, Xia et al. [28], Wong et al. [26, 29], Zhou et al. [35] and Huang et al. [6, 7] have studied how to find the maximum or top-t most influential spatial sites. In addition, Zhan et al. [31] and Zheng et al. [34] considered the uncertainty in the problem. Zhang et al. [32] and Qi et al. [19] investigated the min-dist problem, which minimizes the average distance between the clients and the facilities. These studies did not consider capacity constraints and their algorithms do not apply.

Wong et al. [27] studied the spatial matching problem with capacity constraints. The study tries to assign each client to her nearest facility whose capacity has not exhausted. Due to the capacity constraints of the facilities, a client may be assigned to a facility very far away. U et al. [25] studied the problem further and proposed algorithms that assigned each client to a facility with a capacity constraint while the sum of the distance between each client and its assigned facility is minimized. Sun et al. [22] studied finding the top-k locations from a candidate set that maximize the total number of clients served by the locations. In another work [23], they proposed the problem studied in this report and a centralized algorithm for the problem. As we will use the centralized algorithm as our baseline in the experiments, we will detail it in Section 3.2.

2.2 MapReduce for Computation Intensive Problems

Since proposed, MapReduce has gained much popularity in studies to achieve efficiency and scalability. For example, Lu et al. [13] investigated processing k nearest neighbor joins using MapReduce. Tao et al. [24] studied the minimal MapReduce algorithms that minimize the storage cost, CPU and I/O cost as well as communication cost simultaneously and proposed minimal algorithms for database problems like ranking and spatial skyline queries.

Like in earlier parallel processing techniques such P2P computing [20] and Grid computing [1], many efforts have been made in MapReduce for load balancing on skewness input and complex, non-linear algorithms. For example, Kolb et al. [10] designed methods to handle skewed data for entity resolution. Gufler et al. [3] addressed the load balancing problem in processing MapReduce jobs with complex reducer tasks. They proposed two load balancing approaches that can evenly distribute the workloads on the reducers based on a cost model. They [4] achieved further performance gains by improving the cost estimation through gathering statistics from the mappers. Kwon et al. [12] presented a system that automatically mitigates skewness for user defined MapReduce programs by redistributing the unprocessed input data of the task with the largest expected remaining processing time.

Chapter 3

Preliminaries

In this chapter, we first present some basic concepts and provide a formal definition of the studied problem, and then briefly describe an existing centralized solution to the problem and the MapReduce framework.

3.1 Problem Definition

We consider data objects (as points) in a 2-dimensional Euclidean space S. Given two data objects p and q, $|p, q|$ denotes the Euclidean distance between p and q in S.

Definition 1 (bichromatic reverse nearest neighbor (BRNN) query) *Given two data sets F and C and a query object $f \in F$, the BRNN query returns a subset of C, denoted by $B(f)$, such that $\forall c \in B(f)$ and $f_i \in F$, $|c, f| \leq |c, f_i|$*

The returned subset is call the *influence set* or *BRNN set* of the query object, representing the set of client objects that share service from facility f. For example, in Fig. 3.1(a), the 4 black dots represent the client objects, the 2 small rectangles represent the facility objects. The BRNN set of f_1 and f_2 is $\{c_1, c_2\}$ and $\{c_3, c_4\}$, repectively. Therefore, c_1 and c_2 (resp. c_3 and c_4) are served by f_1 (resp. f_2).

We use the ***n**earest **f**acility **c**ircle* (NFC) [11] to help identify the BRNNs.

Definition 2 (nearest facility circle) *Given a client object c, the nearest facility circle of c, denoted as $n(c)$, is a circle that centers at c and has a radius of $|c, f_c|$, where f_c is the nearest facility of c.*

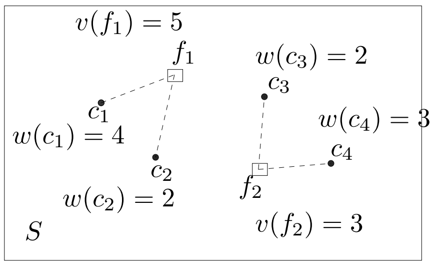
In Fig. 3.1(b), the circles represent the NFC of the client objects in Fig. 3.1(a). We denote the set of all points inside (outside) $n(c)$ by $\bar{c}$ ($\hat{c}$).

Definition 3 (consistent region) *A consistent region R is an arbitrary set of locations (points) that for any client $c \in C$, either all points in R are in $\bar{c}$ or in $\hat{c}$, i.e., $\forall c \in C$, $p_i, p_j \in R$, $p_i \neq p_j$, either $p_i, p_j \in \bar{c}$ or $p_i, p_j \in \hat{c}$*

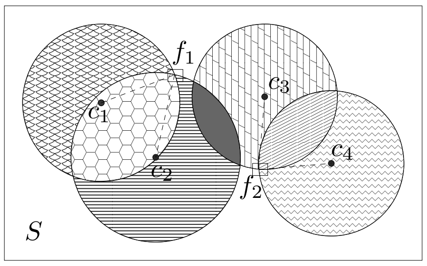
It can be proved [23] that each location p in a consistent region R has the same BRNN set, i.e., $\forall p_i, p_j \in R$, $B(p_i) = B(p_j)$, thus when searching the optimal locations, the points in R can be considered as a whole and we call this set the BRNN set of R, denoted by $B(R)$. However, there can be infinite number of consistent regions since every single point can form such a region. We combine all consistent regions that have the same BRNN set to form the *maximal region*.

Definition 4 (maximal region) *A consistent region R is a maximal region if $\forall R' \cap R \neq \emptyset$, which implies $B(R') = B(R)$, then $R' \subseteq R$*

The maximal regions in Fig. 3.1(b) are those parts represented by the different shadows. In the following, we call a maximal region simply as a region.



(a)



(b)

Figure 3.1: Definitions illustration: $w(c_1) = 4$, $w(c_2) = 2$, $w(c_3) = 2$, $w(c_4) = 3$, $v(f_1) = 5$, $v(f_2) = 3$, $v(f_n) = 4$

The served weight of a facility f is computed as $\min\{v(f), \sum_{c \in B(f)} w(c)\}$. The total weight of the served clients, denoted by $ser(C, F)$, is computed as the sum of the served weights of all facilities, i.e., $\sum_{f \in F} min\{v(f), \sum_{c \in B(f)} w(c)\}$. In the above example, if the weight of c_1, c_2, c_3 and c_4 is 4, 2, 2 and 3, respectively, and the capacity of f_1 and f_2 is 5 and 3, respectively. Then the served weight of f_1 and f_2 is $5 = min\{5, 4+2\}$ and $3 = min\{3, 2+3\}$, respectively, and hence $ser(C, F) = 5+3 = 8$

Definition 5 ***Location Selection for Utility Maximization (LSUM)***
*Given the capacity $v(f_n)$ of a new facility f_n, the **location selection query for utility maximization** returns a set of locations M, which contains all points in the data space S such that if f_n is set up on any of these locations, the number of newly served clients, i.e., the utility of f_n denoted by $u(M)$ is maximized. Here, $u(M) = ser(C, F \cup \{f_n\}) - ser(C, F)$.*

For example in Fig. 3.1(b), if a new facility f_n with capacity 4 is set up in the dark-gray region, the BRNN set will be $\{c_2, c_3\}$. Thus, c_2 and c_3 will be served by f_n and c_1 (resp. c_4) is served f_1 (resp. f_2). The weight of newly served clients (or the utility of f_n) is $3 = min\{5, 4\} + min\{3, 3\} + min\{4, 2+2\} - 8$. The other shadowed regions have different BRNN sets, which may lead to different utilities. To find optimal locations, we only need search all such regions.

3.2 Existing Centralized Algorithm

An existing centralized algorithm [23] processes the query as follows. Since the new facility should serve at least one client object, the optimal locations must locate in the NFC. Given the capacity of the new facility, the algorithm scans regions within each NFC and calculates the weight of newly served clients. All regions within an NFC are enumerated by the combinations of inside and outside its intersected NFCs. An enumerated region may not exist. The algorithm checks and discards such a region. The algorithm also prunes unpromising regions with the best results it has found so far. For the sake of brevity, we omit the details of checking the existence of enumerated regions and the pruning rules.

We illustrate the algorithm succinctly with the example in Fig. 3.1. To answer the query, the algorithm only needs to search the different shadowed regions. However, a challenge is to obtain all such shadowed regions. The algorithm adopts the approach of enumerating. To enumerate all such regions, the algorithm scans one NFC after another. First it enumerates regions within $n(c_1)$, since $n(c_1)$ only intersects with $n(c_2)$, it obtains regions $\bar{c_1}\bar{c_2}$ and $\bar{c_1}\hat{c_2}$ through enumerating inside and outside $n(c_2)$, the two regions all exist and the utility is 1 and -1, respectively. Here -1 means less clients are served (because f_n shares too many weights from f_1 and there is a pruning rule utilizing this property). Next, the algorithm enumerates within $n(c_2)$, $n(c_2)$ intersects with $n(c_1)$ and $n(c_3)$, to avoid repeated search, it only enumerates the NFC that bears a bigger id and smaller id NFC is treated as outside without enumeration. Thus the obtained regions are $\hat{c_1}\bar{c_2}\bar{c_3}$ and $\hat{c_1}\bar{c_2}\hat{c_3}$ by enumerating inside and outside $n(c_3)$, again the two regions exist and the utility is 1 and 3, respectively. Next NFC is $n(c_3)$, however, the algorithm estimates that the upper utility of all regions inside $n(c_3)$ (outside $n(c_1)$ and $n(c_2)$) is 2, which is smaller than the current best result 3, thus $n(c_3)$ is skipped. The same skipping happens to $n(c_4)$. No other NFC to scan, the search procedure ends. And the dark-gray region is returned as the query result.

3.3 MapReduce Framework

MapReduce [2] is a programming framework exploiting the parallelism among a cluster of computing nodes, which is widely used in both scientific and commercial applications for its simplicity, flexibility, fault tolerance and scalability. Hadoop[1] is one of the most popular implementations of MapReduce framework. The execution of MapReduce algorithms in hadoop is known as *job* (or round). Each job consists of three phases *map*, *shuffle* and *reduce*. To execute a job, the number of *map tasks* and *reduce tasks* is pre-defined. Each processor in the cluster can execute one task at a time. Data is represented as *key-value* pairs between phases.

Map Each map task takes the input (for each line, function *map* is called once) and generates a list of key-value pairs $\langle k, v\rangle$. Usually key k is numeric, value v represents information to be processed.

[1] http://hadoop.apache.org/

Shuffle During the shuffle phase, the key-value list is distributed to reduce tasks under the restriction that pairs of same key being delivered to the same task. In hadoop, the function *getPartition* determines to which reduce task should key k be assigned.

Reduce In the reduce phase, each reduce task is allocated a processor to execute. If the number of reduce tasks exceeds the number of processors, after a task is completed, the next task is dispatched to the released processor with a framework-specific scheduling mechanism. Within a reduce task, function *reduce* is called to process the values of a same key. The function is called several times until all types of keys dispatched to the task are dealt with.

Chapter 4

Overview of MapReduce Based Algorithm for LSUM Query

To take advantage of parallelism, we divide the data space into partitions and search the query result in each partition synchronously. We then merge the local results and identify the final query result.

The whole process is achieved by 3 MapReduce jobs:

(i) **Search space partitioning**, where the search space is divided into partitions,

(ii) **Local optimal region searching**, where each partition is searched to find local results, and

(iii) **Local result merging**, where the local results are merged to get the final result.

4.1 Search Space Partitioning

The first job reads in all the initial input, answers the BRNN queries and draws the NFCs to set up the search space. Then it divides the search space into a number of partitions to be processed in parallel. To achieve load balancing, it also estimates the workload of each partition and makes the workload as even as possible. The strategy to partition the search space and to achieve load balancing will be discussed in Chapter 5 and 6, respectively. The map phase outputs each line of the initial input with the same key k, so that the key-value pairs $\langle k, line \rangle$ are processed by one reduce task. In the reduce phase, the output of partitions in the form of key-value pair is $\langle t_i, m_p \rangle$ $(1 \leq t_i \leq s)$, where the key t_i indicates by which reduce task should a partition p be processed and s denotes the number of reduce tasks of the next job, the value m_p represents message to rebuild the partition p. The massage contains the relevant facility and client objects from the initial input and results to the BRNN (and intersection) query. The facility and client objects can be output without modifications. The output of each query result is a list of object IDs. Usually, the first job is within the computation capability of one node in the cluster.

The reason for the first job to use one reduce task to take in the whole initial input and distribute the spatial query results to the next job is to reduce the cost of index building. Since the LSUM problem is computation intensive, the cost of distributing data in the cluster is relatively small. However, the cost of building a index and querying on the index cannot be neglected, especially for some disk-based index like R-tree. And our problem needs to leverage the spatial index, e.g. k-d tree, R-tree etc., to efficiently deal with some basic spatial queries such as BRNN queries and intersection queries. It is not plausible to force all reduce tasks of the next job to build their own index over the whole space but only issue queries corresponding to a small area. It is beyond the scope of this report to determine how to separate the initial input such that for each part one task can build a partial index that gives the same result as the entire index. Considering the storage volume and network bandwidth of nowadays clusters and the relatively small storage space occupied by the query results, such results can be separated and distributed along with other data. In this way, the index only need to be built once and all queries are conducted once.

4.2 Local Optimal Region Searching

The second job receives the intermediate data output by the space partitioning job, gathers the partition information and searches optimal regions in the recovered partitions. The map phase of this job writes the input key-value pairs $\langle t_i, m_p \rangle$ $(1 \leq t_i \leq s)$ directly to output (Mapper class *IdentityMapper* can be used in hadoop). To control workload distribution, each reduce task is designed to process *only one type of key*, i.e., reduce task t_i only processes key t_i (overriding the function $getPartition$ can accomplish this in hadoop). In reduce phase, reduce task t_i is dispatched to an available processor to execute. Reduce task t_i processes the key-value pair $\langle t_i, list[p] \rangle$. Incorporating all value fields of key t_i, $list[p]$ carries the information of a list of partitions. Each partition p can be recovered from relevant messages m_p. The core method to search the regions within each partition is same to the centralized algorithm [23]. After searching, each reduce task outputs the local result $\langle u, R \rangle$. Key u is the utility, and the value is the information representing the corresponding region R. This task incurs a big computation cost, so it shall be conducted in parallel among all available processors in the cluster.

4.3 Local Result Merging

The local result output by each reduce task of the last job in just partially optimal. Finally, the third job collects the result $\langle u, R \rangle$ output by each reduce task, picks out entries with the maximum utility and outputs the final query result. This job is necessary especially when there are a large amount of partial results. With rare exceptions, the collected results can fit into the disk of a centralized server and the final result can fit into the main memory. Therefore this job can be completed by one reduce task and no extra selecting efforts are needed.

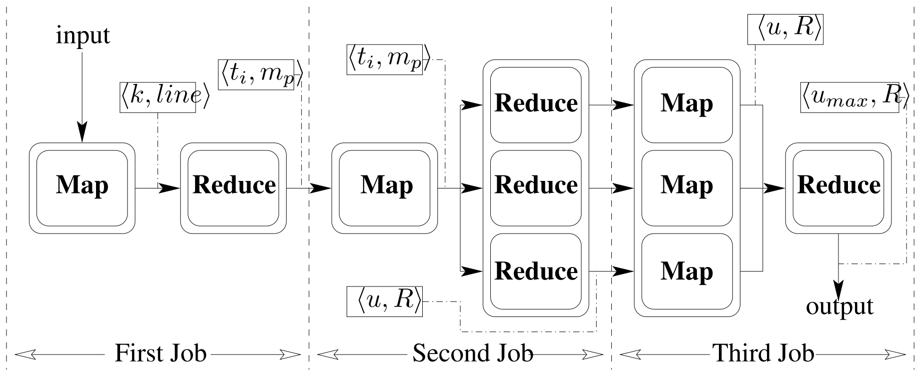


Figure 4.1: Overview of the MapReduce based algorithm for LSUM query

The flowchart in Fig. 4.1 illustrates the jobs designing and data flow for the MapReduce algorithm.

Chapter 5

Search Space Partitioning

In this chapter, we focus on how to partition the search space to enable parallel processing. We first describe a straightforward method of grid partitioning, then we propose an improved partitioning strategy called the arc-based partitioning.

5.1 Grid Partitioning

A straightforward way to divide the search space is using a regular grid, where the regions intersecting each cell form a partition. A a major drawback of the strategy is that a region can intersect multiple cells and hence belongs to multiple partitions. For example, in Fig. 5.1, the search space is divided into four cells. The gray region inter-

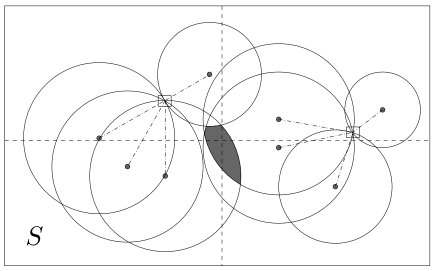


Figure 5.1: Grid partitioning

sects four cells, hence it will be assigned to 4 partitions. The above drawback results in repeated searching in the later stages, which is a waste of the limited computation resources. As the grid granularity gets finer, the problem gets worse because a region can intersect more smaller cells. To avoid this drawback, we propose a partitioning strategy based on the NFC arcs as follows.

5.2 Arc-based Partitioning

To address the drawback of grid partitioning, we propose to use arcs from the NFCs to divide the space and form the *arc-based partitions*. Since regions are already bounded by arcs, arc-based partitions can separate the space disjointly. For instance, in Fig. 5.2, the above space is divided into four partitions, and the partitions do not share a common region.

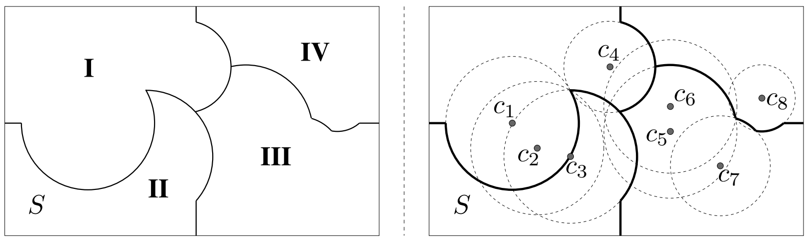


Figure 5.2: Arc-based partitioning

The method to achieve such partitions from the gird is as follows. For a cell g in the grid, if g contains the center of a NFC $n(c_i)$, with the regions locating inside $n(c_i)$ but outside $n(c_j)$ $(1 \leq j < i)$ being assigned to g, the bounded-arcs of such regions (maybe with the boundaries of g) form an arc-based partition. If g does not contain the center of any NFC, it is an empty partition. An empty partition does not need to search. Assigning increasing id c_i $(1 \leq i \leq 8)$ to the NFC in Fig. 5.2, then applying the above method, we are able to get the four arc-based partitions as shown.

This strategy removes the drawback, but it is very likely to lead to unbalanced load. The number of regions in a partition that holds more small id NFCs is usually larger than a partition that holds less. As shown in the Fig. 5.2, the northeast partition contains only 3 regions since it holds two NFCs c_6 and c_8 that has relatively bigger id. How to achieve load balancing is discussed in the next chapter.

Chapter 6

Load Balancing

In this chapter, we discuss how to balance the load of the second job (local optimal region searching), which is the most time-consuming job. For simplicity, we assume the nodes in the cluster are equipped with the same hardware, and hence with the same computation capability. We focus on achieving a balanced load among the reduce tasks of the second job.

6.1 Partition Cost Estimation

As discussed above, the arc-based partitioning is associated with the NFC. So to estimate the cost of searching local optimal regions in a partition, we can use the *maximum* number of enumerations associated with the NFCs in the partition. With the arc-based partitioning strategy, the finest granularity is one partition containing only one NFC. If we adopt the finest granularity, then the maximum number of enumerations associated with the NFC contained by a partition can be used as the estimated cost of the partition. Particularly, if the contained NFC intersects with n NFCs that have a bigger id, then the cost of this partition is 2^n because there are at most 2^n combinations (inside and outside the n NFCs) to enumerate.

Admittedly, this is an estimated cost only considering the worst case. The first disturbing factor is in reality there can be much less real regions on average. For example, in Fig. 6.1, the left side sketches the contours of the partition, the right side displays the corresponding NFC and contained regions. In the right side, NFC $n(c_2)$ intersects with four NFCs, three NFCs $n(c_3)$, $n(c_4)$ and $n(c_5)$ having a bigger id, so the estimated cost for the associated partition is $2^3 = 8$. However, since $n(c_3)$ and $n(c_4)$ does not intersect, regions in both $n(c_3)$ and $n(c_4)$ ($\bar{c}_3\bar{c}_4\bar{c}_5$ and $\bar{c}_3\bar{c}_4\hat{c}_5$) will not be enumerated, for region $\bar{c}_3\bar{c}_4$ does not exist and there is no need to do subsequent enumeration on $n(c_5)$. So the real cost of the partition will be 6, not estimated 8. The second disturbing factor is the pruning techniques in the searching algorithm. As the query needs to find the optimal regions, during the search, the algorithm maintains the best result found so far and uses it to prune unpromising search branches. Recall the example illustrating the searching algorithm in Section 3.2, $n(c_3)$ and $n(c_4)$ are pruned

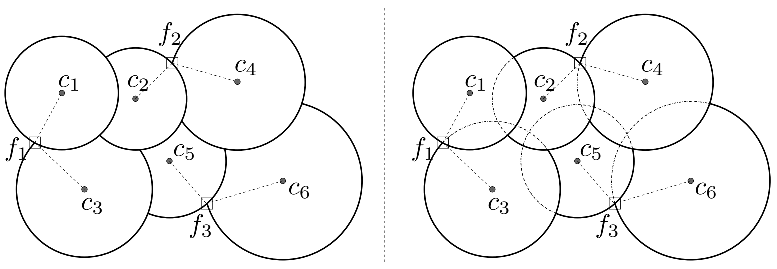


Figure 6.1: Partition cost estimation

without searching.

Both the above factors have an uncertain impact on the estimated cost. Nevertheless, such estimation can still work as an indicator for the load assignment. During the real search procedure, a partition with much bigger estimated cost rarely occupies less computation resources than one with a much smaller cost, for they are both affected by the above factors. Therefore, we utilize such estimation to balance the load among reduce tasks.

6.2 Partition Assignment

Recall that there are $|C|$ client objects and we set the number of reduce tasks for the second job to s. The $|C|$ clients leads to $|C|$ NFCs. Adopting the finest granularity, we have $|C|$ partitions with each one carrying an estimated cost.

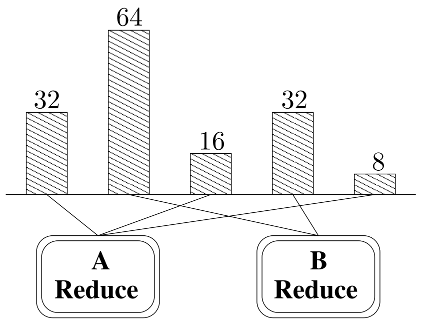
Since $s \ll |C|$, each reduce task would process a bundle of partitions. Regardless of the disturbing factors, the load of a reduce task is the sum of the cost of the assigned partitions. In this chapter, we focus on strategies of assigning partitions to reduce tasks to achieve load balancing.

6.2.1 Round-robin Assignment

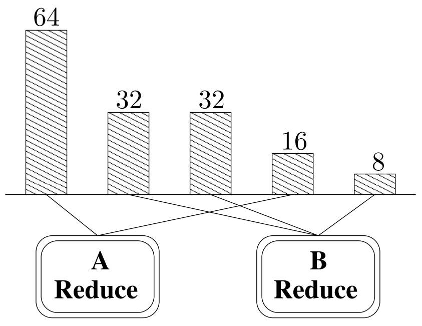
A straightforward way to assign these partitions is the round-robin strategy. Specifically, we arrange the s reduce tasks in a row and the ith partition is assigned to the $\underline{i \bmod s}$th reduce task. For example, in Fig. 6.2(a), we have five partitions whose cost is 32, 64, 16, 32 and 8, respectively, and two reduce tasks A and B. With the round-robin strategy, the first, third and fifth partition goes to task A and the second and fourth goes to task B, which results in the estimated load of A and B is 56 and 96, respectively.

At first sight, this strategy rarely leads to a balanced assignment. However, due to the disturbing factors, the estimated load is merely an indicator, not the exact running time. Though the estimated load seems unbalanced, in reality, especially when there are a large number of partitions, the real workload may be balanced.

The round-robin strategy can work as an indicator to the effectiveness of our cost estimation method and a baseline comparing to other strategies.



(a) Round-robin strategy



(b) Least-load strategy

Figure 6.2: Basic assignment strategies

6.2.2 Least-load Assignment

In this strategy, we adopt the greedy assignment approach proposed by Gufler et al. [3] and we call it *least-load assignment*. The strategy picks the most *expensive* partition not yet allocated, and assigns it to the reduce task that has the least total load. The above steps repeat until all partitions have been assigned. For illustration, let us use the above example again with least-load assignment strategy in Fig. 6.2(b).

In this figure, all the partitions have been decreasingly sorted. At first, 64 is allocated to A, then 32 to B. Next, the second 32 is also assigned to B, since B at present has less load than A. Then 16 is allocated to A and 8 to B according to the load priority as well. After assignment, A and B has a total estimated load of 80 and 72, respectively. The following Algorithm 1 gives the details of the least-load assignment strategy.

Algorithm 1: LeastloadAssignment

```
Input: $Q_p$                                       /*partition queue*/
       $PQ_r$                                      /*reduce task priority queue*/
Output: $PQ_r$                  /*reduce task queue with assigned partitions*/
1 while $Q_p$ is not empty do
2     $p \leftarrow head(Q_p)$                      /*remove the head of $Q_p$ to $p$*/
3     $r \leftarrow head(PQ_r)$
4     $r.addPartition(p)$                       /*assign partition $p$ to reduce task $r$*/
5     $r.load \leftarrow r.load + p.cost$
6     $PQ_r.insert(r)$                                         /*reinsert $r$*/
7 return $PQ_r$
```

However, when very few largest partitions *dominate* all the other partitions, the strategy will fail to work. Here dominating means the sum of the cost of the other partitions is still smaller than the cost of a *huge* partition. In such a situation, the reduce tasks processing the few largest partitions become the bottleneck and slow down the speed of the query processing. For example, when the partition costs are as Fig. 6.3 shows, the dominating situation would occur. If we have two reduce tasks to process I or three tasks to process II, the assigned load distribution will be 1024, 120 for I and 1024, 512, 56 for II, respectively, which is very skew and the advantage of parallelism will not be fully utilized. To effectively deal with all possible cost distribution, we propose the next dynamic assignment strategy.

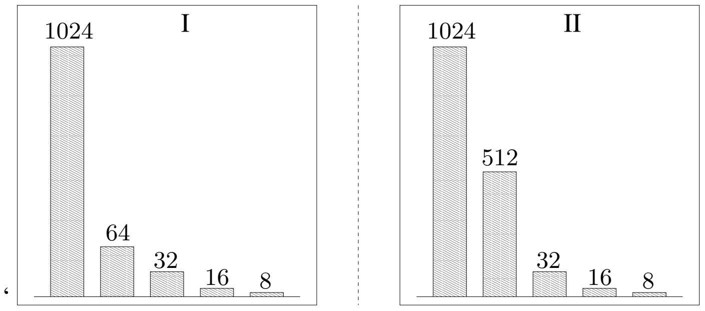


Figure 6.3: Dominating situations

6.2.3 Dynamic Assignment

The ideal strategy would assign every reduce task the same amount of load, regardless of the cost distribution. So in this strategy, we try to make the estimated load of every reduce task as close to the average as possible and name it *dynamic strategy*.

We first get the average of total cost, then subdivide the dominating partitions that are larger than the average (the subdividing result is also called partitions). After that, we assign the partitions with the least-load assignment strategy. If the estimated load of a reduce task is beyond the average over a threshold, the most expensive partition in the task is subdivided. The above procedure repeats until the load difference between each task and the average is within the threshold.

Subdividing a partition. If there are r regions in a partition, then the partition can be subdivided into at most r new partitions. However, without searching it is unable to measure the exact number of regions in a partition. The information we have is the partition's intersected NFC list. From the list, we can pick a bigger id NFC to divide the partition into two new partitions. The two new partitions contain regions inside and outside the NFC, which can be encoded as 1 and 0, respectively.

Generally, if we need to subdivide a partition into 2^t new partitions, we can exploit the first t bigger id elements of the intersected NFC list, and the 2^t new partitions are derived from pre-enumeration of the t NFCs.

For example, in Fig. 6.4 left side I, there is an NFC and the corresponding partition, the shadowed parts belong to other partitions, the three dashed arcs a_1, a_2 and a_3 are from the first three bigger id elements $n(c_{i_1})$, $n(c_{i_2})$ and $n(c_{i_3})$ in the list (arcs from other elements are omitted). We can subdivide the partition into $2^3 = 8$ new partitions by pre-enumerating inside and outside the three elements. The result is the eight new partitions encoded into $000, 001, 010, 011, 100, 101, 110, 111$, respectively. Here partition 000 means $\hat{c_{i_1}}\hat{c_{i_2}}\hat{c_{i_3}}$, 001 means $\hat{c_{i_1}}\hat{c_{i_2}}\bar{c_{i_3}}$, and so on.

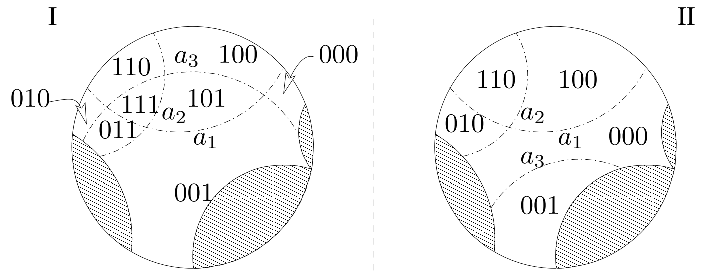


Figure 6.4: Subdividing a partition

The new partitions have equal estimated cost. If the cost of the original partition is 2^n, then each new partition has an estimated cost of 2^{n-t}.

Artificial new partitions. It happens that some new partitions may not exist but merely come from the enumeration. As Fig. 6.4 right side II shows, when a_3 does not intersect with a_1 or a_2, new partitions in both $n(c_{i_1})$ and $n(c_{i_3})$ or $n(c_{i_2})$ and $n(c_{i_3})$ $(011, 101, 111)$ are artificial and they do no actually exist. So when we subdivide a partition, all the new partitions will be checked. If a partition is not real, it will be discarded. A byproduct of checking the new partitions is refining the cost estimation for the original partition. Take the Fig. 6.4 right side II as an example, originally the estimated cost of the partition is 2^n. After subdividing and 3 artificial partitions being discarded, the total cost of the new partitions becomes $(2^3 - 3) \times 2^{(n-3)}$, which is smaller than 2^n and more precise. The more new partitions we produce, the more precise the estimation become, but also needs more additional subdividing and checking operations. A more precise estimation also changes the average we calculate before, which requires the average to be kept updating.

Strategy details. The dynamic assignment strategy works as follows. Given an input parameter ε, first it calculates the average cost ave of all partitions, subdivides the partitions whose cost is larger than $(1+\varepsilon)\times ave$ and discards the artificial partitions. Then it adopts the same method to assign these partitions as the least-load assignment strategy. After assigning all partitions and calculating the new average ave', if the load of a reduce task is larger than $(1 + \varepsilon) \times ave'$, then it subdivides the most expensive partition in the task, and repeats from the first step until the load of every reduce task is within the threshold.

When we subdivide a partition, the number of new partitions is determined by the number of reduce tasks s. We subdivide a partition into 2^t new partitions with $2^{t-1} < s \leq 2^t$. So that after dividing, the cost of a new partition which is 2^{n-t} will be no larger than ${}^{2^n}/_{s}$.

The dynamic assignment strategy is detailed in Algorithm 2.

Algorithm 2: DynamicAssignment

```
Input: $L_p$                                   /*List of partitions*/
       $PQ_r$                                  /*reduce task priority queue*/
       $\varepsilon$                           /*threshold $\varepsilon \geq 0$*/
Output: $PQ_r$                                 /*reduce task queue with assigned partitions*/
1  $even \leftarrow false$
2  $t \leftarrow$ first integer that makes $2^t \geq PQ_r.size$
3  while true do
4      $ave \leftarrow average(L_p)$                               /*get the average cost*/
5      for each $p \in L_p$ do
6          if $p.cost > (1 + \varepsilon) \times ave$ then
7              /*subdivide partition $p$ into $2^t$ subparts and return the existing ones*/
8              $subparts[2^t] \leftarrow subdivide(p, 2^t)$
9              $L_p.insertAll(subparts)$     /*insert all sub-parts into partition list*/
10     $LeastloadAssignment\langle L_p, PQ_r\rangle$        /*use same strategy as least-load
       assignment*/
11     $ave \leftarrow average(L_p)$                           /*get the new average cost*/
12     $even \leftarrow true$
13     for each $r \in PQ_r$ do
14         if $r.load > (1 + \varepsilon) \times ave$ then
15             $even \leftarrow false$                    /*load of reduce task $r$ is too large*/
16             $ep \leftarrow expPartition(r)$         /*return the most expensive partition*/
17             $subparts[2^t] \leftarrow subdivide(ep, 2^t)$
18             $L_p.insertAll(subparts)$
19         $L_p.insertAll(r.partitions)$       /*reinsert all partitions in $r$ to repeat*/
20     if $even$ then
21         break
22 return $PQ_r$
```

6.3 Determine the Number of Reduce tasks

In this section, we discuss how to determine the number of reduce tasks s in the second job. For each reduce task, the framework-specific scheduler allocates the task to an available processor to execute. And s should be equal to the number of available processors if the processors have the same computation capability and the real load of every task is the same, then. And it will be most efficient if all processors start processing the allocated task at once. However, in reality, due to various reasons (such as hardware dissimilarities, unbalanced load assignment et al.), some processors would end earlier and wait the others to complete. It is better to launch a second round of reduce tasks for them.

A reasonable approach is creating more reduce tasks than the number of processors. Suppose m processors are available, with $m < s$, the s tasks form a waiting list. Once a processor completes a task and becomes available, the scheduler allocates it the next task from the waiting list.

Due to the dominating situations, the above approach may have little effect on the round-robin and least-load assignment strategy. It may work well with the dynamic strategy for the reason that although the estimated load of each task is almost the same, the real running time may be different because of the disturbing factors or hardware difference.

Then how much should s be larger than m. If the partitions within a task is completely independent, then theoretically the larger s, the more balanced the load. However, in our problem, the pruning rules make the partitions correlated. A larger s means more reduce tasks and *smaller* partitions in each task. And smaller partitions has relatively weak pruning power. So a larger s may lead to more overall computation because of less pruned search branches. A larger s also means more communication cost and task setup cost. According to our experiments, setting s between $1.0 \times m$ and $4.0 \times m$ is a sound choice.

Chapter 7

Experimental Study

We evaluate the performance of the proposed algorithm on an in-house cluster. The cluster consists of 4 computing nodes. Two are equipped with Intel i7-3770 3.4GHz 4-core processors and the other two with Intel i7-2600 3.4GHz 4-core processors. One node has 32GB main memory, the other three each has 16GB of memory. Each node has one 3TB SATA hard disk, gigabit ethernet and is installed CentOS 6.0 operating system, Java 1.7.0 with a 64-bit server VM and Hadoop 2.0.0-cdh4.2.1. For the Hadoop environment, the replication factor is set to 2 and the size of virtual memory for each map and reduce task is set to 4GB.

We evaluate the following approaches in the experiments.

1. **Centralized** is the centralized algorithm introduced in Section 3. Particularly, when evaluating its performance, in the second job we allocate only one reduce task to process it.

2. **Round-robin** is the round-robin assignment strategy that assigns the partitions according to their positions in a list.

3. **Least-load** is the least-load assignment strategy that utilizes the load balancing assignment method proposed by Gufler et al.[3].

4. **Dynamic** is the dynamic assignment strategy we propose. It takes the dominating situation into consideration and controls the load difference among reduce tasks within a threshold, no matter how many reduce tasks are created.

Both real data sets and synthetic data sets are used in all experiments. Three real data sets NA, NE and US are retrieved from the Rtree Portal[1]. These data sets are preprocessed to remove duplicate points and some small parts. Table 7.1 lists the details of the real data sets. When using a real dataset, we uniformly sample from it to generate C and F. For the synthetic dataset SN, to simulate real-world scenarios, we generate C and F with the Zipfian distribution and the skew coefficient is set to 0.2 in a space domain of $10^4 \times 10^4$. We set the weight of client object to one and generate the capacity

[1]http://www.chorochronos.org/

of facility object using Gaussian distribution with mean and standard deviation set to $2 \times {}^{|C|}/_{|F|}$ and $0.4 \times {}^{|C|}/_{|F|}$, respectively. We leverage spatial index R-tree to answer the nearest neighbor query and the intersection query.

Table 7.1: Real Datasets

Dataset	Cardinality	Description
NA	24,360	locations in North America apart from Hawaii
NE	119,898	addresses of north east of U.S.
US	14,478	locations in U.S apart from Hawaii and Alaska

By default, all 16 processors are used, the number of reduce tasks of the second job is set equal to the the number of processors for the three assignment strategies, threshold ε is set to 10^{-6} for the dynamic assignment strategy.

7.1 Effect of ${}^{|C|}/_{|F|}$

The ratio ${}^{|C|}/_{|F|}$ has a significant effect on the searching algorithm. In this set of experiments, the value of ${}^{|C|}/_{|F|}$ is varied from 10 to 50. It is time-consuming when the ratio is large, so the experiments are conducted on small and medium data sets with $|C|$ no more than 3k. The results on four data sets are presented in Fig. 7.1. The left column illustrates the running time for different algorithms and different partition assignment strategies, the right column presents the corresponding variance of running time on the 16 reduce tasks for the three assignment strategies.

From the running time aspect, the centralized algorithm, referred as *baseline* henceforth, is slowest in most cases, especially when the ratio is large. The round-robin assignment strategies performs better than the baseline on majority points. The least-load strategy behaves similar to round-robin strategy, but fails to outperform the centralized algorithm on more points and in the case ${}^{|C|}/_{|F|} = 40$ in Fig. 7.1(g), it is beaten by the baseline by a wide margin. While the dynamic assignment strategy is considerably faster than the baseline in all settings except for a single case ${}^{|C|}/_{|F|} = 10$ on the US dataset, and it also consistently surpasses the other two strategies.

From the time variance aspect, the value reflects the real load difference among reduce tasks. The right column reinforces the superior of the dynamic assignment strategy to the other two strategies. The variance of the dynamic strategy is the smallest in all cases. In the case ${}^{|C|}/_{|F|} = 30$ in Fig. 7.1(e), though the running time of the dynamic strategy is the similar to the round-robin strategy, the corresponding variance in 7.1(f) is still smaller than the round-robin strategy. The reason behind is the other two strategies are unable to tackle the dominating situation.

Comparing the round-robin and least-load strategy in the right column, the least-load strategy is more sensitive to the dominating situation because when there is a dominating partition, the least-load strategy makes a reduce task only contain the dominating partition, in which case the pruning power may be further reduced.

7.2 Effect of Cardinality

The second set of experiments studies the effects of cardinality of input data sets on the centralized algorithm and MapReduce based algorithms with different assignment strategies. The ratio of $|C|/|F|$ is fixed to 25. For NA and US, we use them as medium data sets and vary the cardinality of $|C|$ from 3k to 20k and 1k to 13k, respectively. For NE and SN, they are used as large data sets and the cardinality of $|C|$ is varied from 20k to 100k.

The left column of Fig. 7.2 shows the performance on four data sets. The round-robin and least-load strategies behaves unsteady, half cases are better than baseline, half are worse. While the dynamic assignment strategy consistently uses less time than the baseline and the other two strategies in all settings. When the search space is separated, the pruning power is weakened on each reduce task, thus the entire searching computation increases. Due to dominating situations, The round-robin and least-load strategy fail to utilize the available computation resources to address the increasing. Therefore, in some cases, they even spend more time than the centralized algorithm.

The pattern of lines in the figures suggests the relationship between the running time and the cardinality is not monotonic. The inflection points are due to the different possible sampling distributions of facilities and clients, and the density of clients around facilities (reflected by $\theta \propto |C|/|F|$) mainly determines the searching cost.

From the time variance column on the right side of Fig. 7.2, the time variance of the least-load assignment strategy is largest on nearly all points, which suggests the real load distribution is quite unbalanced among reduce tasks. Round-robin strategy is marginally better than the least-load strategy. While the variance of dynamic assignment strategy is smaller than the other two strategies by a factor of an order of magnitude. On data sets NA and US, the time variance of dynamic assignment strategy increases with cardinality, while in NE and SN, it goes slightly opposite to the growing of cardinality, which suggests that a large cardinality may help the strategy to achieve a more balanced load.

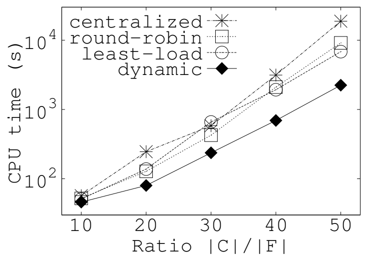
7.3 Fine Tuning of the Dynamic Assignment Strategy

In the next two sets of experiments, we fix the ratio $|C|/|F|$ as well as the cardinality and focus on studying the parameters that affect the dynamic assignment strategy.

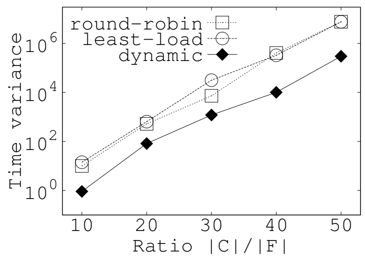
Fig. 7.3 illustrates the effect of the number of reduce tasks s using four data sets. We vary the number of reduce tasks from 1 to 256, which is sixteen times of the number of processors. The results are shown in Fig. 7.3. Fig. 7.3(a) shows that on every dataset, with the increasing of s, the running time decreases rapidly then moderately increases and all the lines reach the nadir around $s = 64$. When the reduce task number is 1, it is equal to the centralized algorithm which takes more time without parallel processing. When the number of reduce tasks is much larger than the number of available processors, the increased computation (plotted in 7.3(d)) prevails the parallelism. There are also additional cost of dispatching and starting more tasks. Hence at point $s = 256$, the running time slightly increase.

The time variance presented in Fig. 7.3(b) decreases monotonically with the growing of s on all data sets, which means creating more reduce tasks makes the processing time of each task tend to be uniform. The reason behind is the number of partitions in a reduce task is lessened, causing the effect of disturbing factors being weakened. An extreme case is each task containing only one region, then the processing time would be the same. However, Fig. 7.3(c) tells creating more reduce tasks means more partitioning cost, for a skewed dataset like SN, the cost increases dramatically. We add up all the processing time of the s reduce tasks and treat the sum as the indicator for the total computation of the MapReduce based algorithm, which is plotted in Fig. 7.3(d). Without exception, more reduce tasks leads to larger total computation on every dataset, which helps explain the line patten in Fig. 7.3(a).

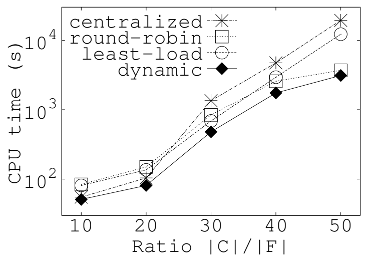
Then, we study the effect of the threshold ε on the strategy. We vary the threshold value from 10^{-3} to 10^{-7} and measure the same metrics as the number of reduce tasks. Fig. 7.4(a) shows similar line pattern to that of the number of reduce tasks but due to different reasons and we will explain it shortly. Fig. 7.4(b) illustrates the time variance of the 16 reduce tasks. The variance decreases along with the decreasing of ε, which is reasonable since the threshold ε controls the load difference among the reduce tasks. As expected, a smaller ε means more balanced load distribution. But in Fig. 7.4(a), all lines (except for SN) do not reach the nadir when ε is the smallest (10^{-7}). The reason lies in Fig. 7.4(c), $\varepsilon = 10^{-7}$ leads to much more partitioning time (except for SN) due to more checking and iteration. The cost increases significantly on dataset NA, NE and US when $\varepsilon = 10^{-7}$, which leads to lengthened running time. Fig. 7.4(d) illustrates that ε has a undetermined effect on the total computation. On dataset NE, SN and US, the total computation varies marginally with ε. While on dataset NA, it fluctuates roughly between 0.9ks and 2ks. With nothing else altered, the difference mainly comes from the variation of pruning power due to the changed distribution of partitions in each reduce task.



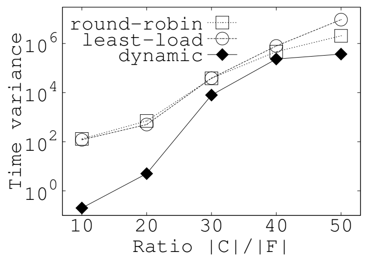
(a) NA



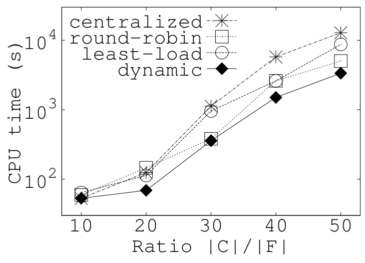
(b) NA



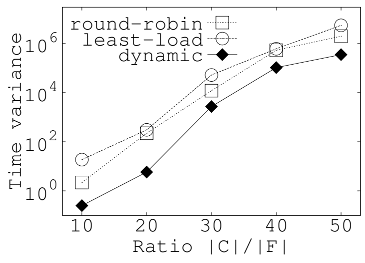
(c) NE



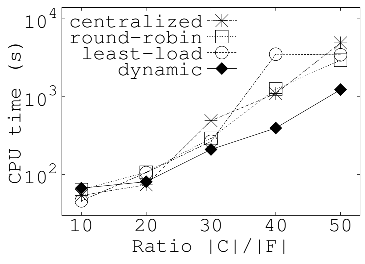
(d) NE



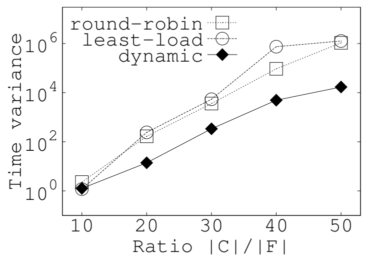
(e) SN



(f) SN



(g) US



(h) US

Figure 7.1: Effect of the ratio $^{|C|}/_{|F|}$ on the search algorithms and assignment strategies

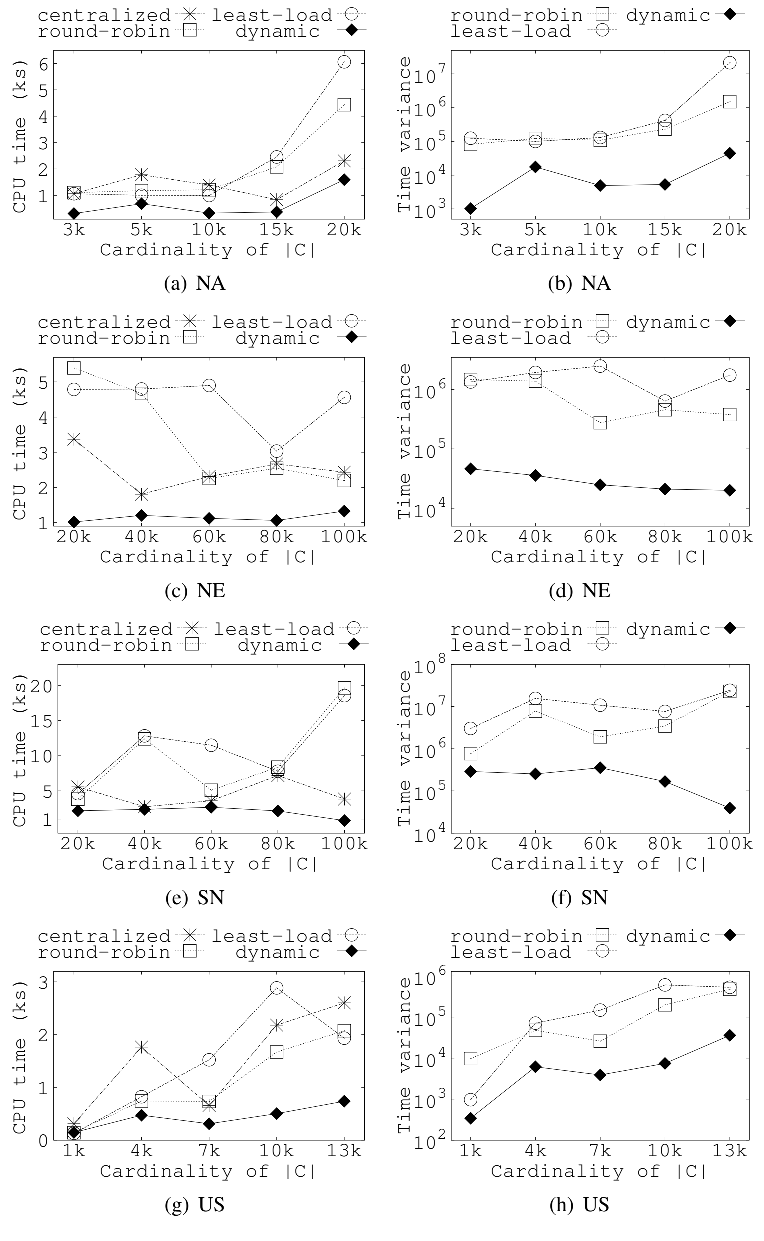
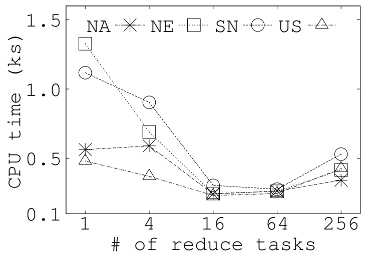
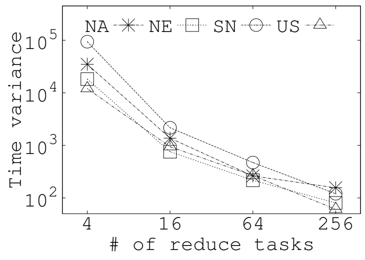


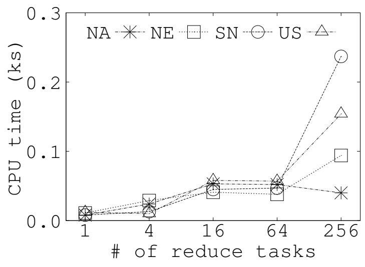
Figure 7.2: Effect of the cardinality on the search algorithms and assignment strategies



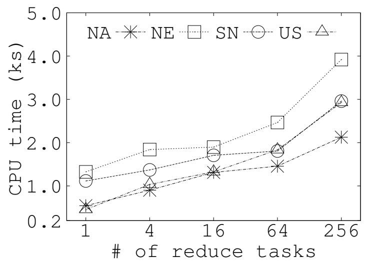
(a) running time



(b) variance

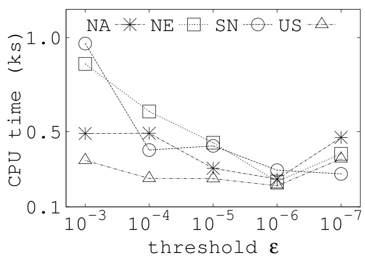


(c) partitioning time

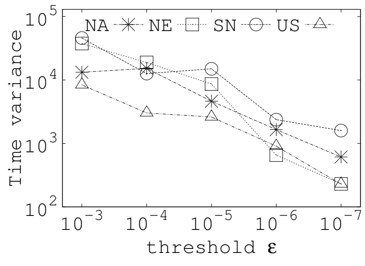


(d) total computation

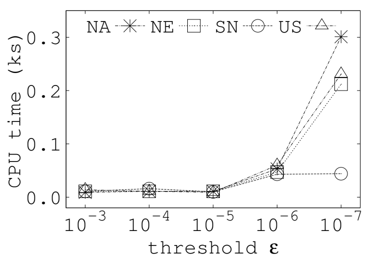
Figure 7.3: Effect of number of reduce tasks s on the dynamic assignment strategy



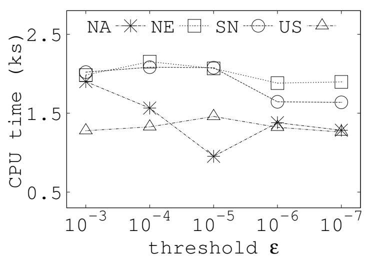
(a) running time



(b) variance



(c) partitioning time



(d) total computation

Figure 7.4: Effect of ε on the dynamic assignment strategy

Chapter 8

Conclusions

In this report, we studied a MapReduce based algorithm for the problem of location selection for utility maximization. By arc-based partitioning, we divide the search space into a number of disjoint partitions. To achieve load balancing, we propose the dynamic assignment strategy. The strategy effectively handles the dominating situation and controls the estimated load difference among reduce tasks within a threshold. Extensive experiments are performed using both real and synthetic data sets and the results demonstrate our proposed MapReduce based algorithm is efficient and scalable.

Bibliography

[1] Al-Khateeb, A., Rashid, N.A., Abdullah, R.: An enhanced meta-scheduling system for grid computing that considers the job type and priority. pp. 389–410 (2012)

[2] Dean, J., Ghemawat, S.: Mapreduce: Simplified data processing on large clusters. In: OSDI, pp. 137–150 (2004)

[3] Gufler, B., Augsten, N., Reiser, A., Kemper, A.: Handling data skew in mapreduce. In: The First International Conference on Cloud Computing and Services Science, pp. 574–583 (2011)

[4] Gufler, B., Augsten, N., Reiser, A., Kemper, A.: Load balancing in mapreduce based on scalable cardinality estimates. In: ICDE, pp. 522–533 (2012)

[5] Hale, T.S., Moberg, C.R.: Location science research: a review. Annals of Operations Research **123**(1-4), 21–35 (2003)

[6] Huang, J., Wen, Z., Pathan, M., Taylor, K., Xue, Y., Zhang, R.: Ranking locations for facility selection based on potential influences. In: The 37th Annual Conference on IEEE Industrial Electronics Society, pp. 2411–2416 (2011)

[7] Huang, J., Wen, Z., Qi, J., Zhang, R., Chen, J., He, Z.: Top-k most influential locations selection. In: CIKM, pp. 2377–2380 (2011)

[8] Kahraman, C., Ruan, D., Doan, I.: Fuzzy group decision-making for facility location selection. Information Sciences **157**, 135–153 (2003)

[9] Klose, A., Drexl, A.: Facility location models for distribution system design. European Journal of Operational Research **162**(1), 4–29 (2005)

[10] Kolb, L., Thor, A., Rahm, E.: Load balancing for mapreduce-based entity resolution. In: ICDE, pp. 618–629 (2012)

[11] Korn, F., Muthukrishnan, S.: Influence sets based on reverse nearest neighbor queries. In: SIGMOD, pp. 201–212 (2000)

[12] Kwon, Y., Balazinska, M., Howe, B., Rolia, J.: Skewtune: mitigating skew in mapreduce applications. In: SIGMOD, pp. 25–36 (2012)

[13] Lu, W., Shen, Y., Chen, S., Ooi, B.C.: Efficient processing of k nearest neighbor joins using mapreduce. Proc. VLDB Endow. **5**(10), 1016–1027 (2012)

[14] Melkote, S., Daskin, M.S.: Capacitated facility location/network design problems. European Journal of Operational Research **129**(3), 481–495 (2001)

[15] Melo, M., Nickel, S., Saldanha da Gama, F.: Dynamic multi-commodity capacitated facility location: a mathematical modeling framework for strategic supply chain planning. Computers and Operations Research **33**(1), 181–208 (2006)

[16] Melo, M.T., Nickel, S., Saldanha-Da-Gama, F.: Facility location and supply chain management–a review. European Journal of Operational Research **196**(2), 401–412 (2009)

[17] Nutanong, S., Tanin, E., Zhang, R.: Incremental evaluation of visible nearest neighbor queries. TKDE **22**(5), 665–681 (2010)

[18] Nutanong, S., Zhang, R., Tanin, E., Kulik, L.: Analysis and evaluation of v*-*k*nn: an efficient algorithm for moving *k*nn queries. VLDB J. **19**(3), 307–332 (2010)

[19] Qi, J., Zhang, R., Kulik, L., Lin, D., Xue, Y.: The min-dist location selection query. In: ICDE, pp. 366–377 (2012)

[20] Qiao, Y., von Bochmann, G.: Load balancing in peer-to-peer systems using a diffusive approach. pp. 649–678 (2012)

[21] Revelle, C.S., Eiselt, H.A., Daskin, M.S.: A bibliography for some fundamental problem categories in discrete location science. European Journal of Operational Research **184**(3), 817–848 (2008)

[22] Sun, Y., Huang, J., Chen, Y., Du, X., Zhang, R.: Top-k most incremental location selection with capacity constraint. In: WAIM, pp. 165–171 (2012)

[23] Sun, Y., Huang, J., Chen, Y., Zhang, R., Du, X.: Location selection for utility maximization with capacity constraints. In: CIKM, pp. 2154–2158 (2012)

[24] Tao, Y., Lin, W., Xiao, X.: Minimal mapreduce algorithms. In: SIGMOD (2013 to appear)

[25] U, L.H., Mouratidis, K., Yiu, M.L., Mamoulis, N.: Optimal matching between spatial datasets under capacity constraints. TODS **35**(2), 9:1–9:44 (2010)

[26] Wong, R.C.W., Özsu, M.T., Fu, A.W.C., Yu, P.S., Liu, L., Liu, Y.: Maximizing bichromatic reverse nearest neighbor for l p -norm in two- and three-dimensional spaces. VLDB J. **20**(6), 893–919 (2011)

[27] Wong, R.C.W., Tao, Y., Fu, A.W.C., Xiao, X.: On efficient spatial matching. In: VLDB, pp. 579–590 (2007)

[28] Xia, T., Zhang, D., Kanoulas, E., Du, Y.: On computing top-t most influential spatial sites. In: VLDB, pp. 946–957 (2005)

[29] Yan, D., Wong, R.C.W., Ng, W.: Efficient methods for finding influential locations with adaptive grids. In: CIKM, pp. 1475–1484 (2011)

[30] Yu, C., Zhang, R., Huang, Y., Xiong, H.: High-dimensional knn joins with incremental updates. GeoInformatica **14**(1), 55–82 (2010)

[31] Zhan, L., Zhang, Y., Zhang, W., Lin, X.: Finding top k most influential spatial facilities over uncertain objects. In: CIKM, pp. 922–931 (2012)

[32] Zhang, D., Du, Y., Xia, T., Tao, Y.: Progressive computation of the min-dist optimal-location query. In: VLDB, pp. 643–654 (2006)

[33] Zhang, R., Jagadish, H.V., Dai, B.T., Ramamohanarao, K.: Optimized algorithms for predictive range and knn queries on moving objects. Inf. Syst. **35**(8), 911–932 (2010)

[34] Zheng, K., Huang, Z., zhou, A., Zhou, X.: Discovering the most influential sites over uncertain data: A rank-based approach. TKDE **24**(12), 2156–2169 (2012)

[35] Zhou, Z., Wu, W., Li, X., Lee, M.L., Hsu, W.: Maxfirst for maxbrknn. In: ICDE, pp. 828–839 (2011)