J. Cent. South Univ. (2014) 21: 190−198
DOI: 10.1007/s11771-014-1930-7

Springer

An adaptive range-query optimization technique with distributed replicas

Sayar Ahmet[1], Pierce Marlon[2], Fox C. Geoffrey[2, 3, 4]

1. Department of Computer Engineering, Kocaeli University, Kocaeli 41380, Turkey;
2. Department of Computer Science, Indiana University, Bloomington 47404, USA;
3. Community Grids Lab, Indiana University, Bloomington 47404, USA;
4. School of Informatics and Computing, Indiana University, Bloomington 47404, USA

Abstract: Replication is an approach often used to speed up the execution of queries submitted to a large dataset. A compile-time/run-time approach is presented for minimizing the response time of 2-dimensional range when a distributed replica of a dataset exists. The aim is to partition the query payload (and its range) into subsets and distribute those to the replica nodes in a way that minimizes a client's response time. However, since query size and distribution characteristics of data (data dense/sparse regions) in varying ranges are not known a priori, performing efficient load balancing and parallel processing over the unpredictable workload is difficult. A technique based on the creation and manipulation of dynamic spatial indexes for query payload estimation in distributed queries was proposed. The effectiveness of this technique was demonstrated on queries for analysis of archived earthquake-generated seismic data records.

Key words: distributed systems; load balancing; range query; query optimization

1 Introduction

Analysis of data is an important step in understanding and solving a scientific problem. Analysis involves extracting the data of interest from all available data sources and can occur at several stages along a data processing pipeline ranging from raw data to advanced data products. However, in many areas of science and engineering, the ability to analyze information is increasingly hindered by dataset sizes. The vast amount of data in scientific datasets makes efficient access to the data and management of potentially heterogeneous system resources for data processing a difficult task. Thus, subsetting and aggregation are widespread techniques used in large-scale scientific data intensive applications. First, subsets are calculated and distributed to the replica nodes and then the results are aggregated to create the response to the main query set. Such a technique is possible in cases where the distributed replica of a dataset exists. This work argues that a common compile/run-time programming support can be developed for applications in which large datasets are queried in distributed computing environments [1].

Distributed parallel processing is the most promising approach for designing scalable and high performance computers that are suitable for tackling distributed data processing problems. This approach provides significant advantages over the shared memory ones in terms of cost and scalability. However, systems based on this approach are much more complicated to program than shared memory machines. One major reason for this is the lack of a single global address space. As a result, programmers, compilers, or run-time environments are responsible for distributing code and data over processors as well as for managing communications among tasks. The distribution is significantly important for the efficiency of the parallel programs in a distributed memory machine. For a good data distribution, evenly distributing the workload over processors should be considered so that parallelism is maximized and communication is minimized. Yet, due to the stringent characteristics and dynamic nature of data, performing efficient load balancing and parallel processing over the unpredictable workload is not easy, the problem is illustrated in Fig. 1. The work is cut into independent smaller pieces, no matter they are of the same size ranges, or sizes are highly variable. Creating subsets for the uniformly distributed datasets such as raster images is straightforward. In contrast, creating subsets for non-uniformly distributed datasets is cumbersome; for example, vector data are defined with the (x, y) coordinates on a 2-dimensional plane (Fig. 1). In such a case, subsetting most probably results in data

Received date: 2012−11−08; **Accepted date:** 2013−07−07
Corresponding author: Sayar Ahmet, Associate Professor, PhD; Tel: +90−262−3033583; E-mail; ahmet.sayar@kocaeli.edu.tr

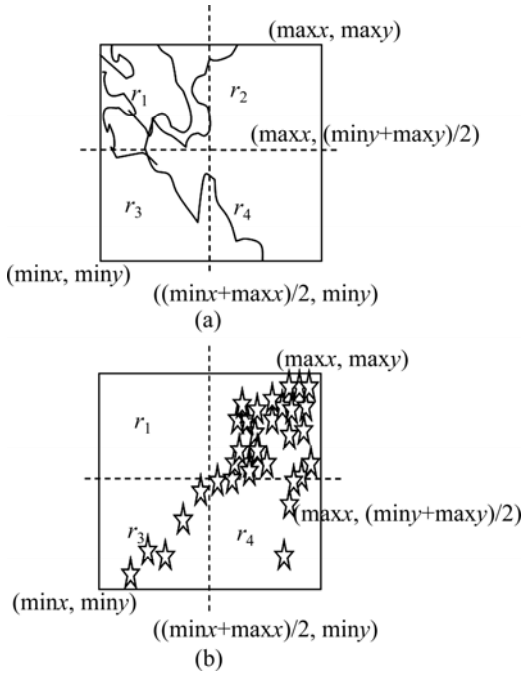

Fig. 1 Problem illustration with straightforward partitioning: (a) Linestring/polygon data; (b) Point data

and execution skews.

In this work, a compile-time/run-time approach is presented for minimizing the response time of 2-dimensional range queries when the distributed replica of a dataset exists. The proposed framework assumes that replica nodes are created in advance and the service-level binding information is stored in a file. The work presented is summarized as developing an efficient query planning strategy to determine the best combination of replica nodes and their corresponding payloads in order to optimize query response time in distributed environments. Enhancements are at the application level, not at network level. The technique is based on creating, updating and manipulating dynamic spatial indexes for query payload estimation in distributed queries. This technique takes the data dense/sparse regions into consideration (Fig. 1). A data structure called workload estimation table (WET) developed at compile-time and used during run-time to divide the workload for a range query into subranges carrying equal sizes of query payload. WET is routinely synchronized with the remote replicated datasets, occupies very little space, and provides accurate query-size estimates over a broad range of spatial queries. The effectiveness of the proposed technique is demonstrated on an earthquake simulation derived from Turkey's archived seismic data records.

2 Related work

Most replication studies address issues such as data availability (e.g., in node and network failures) [2−3] and maximizing I/O performance by duplicating data and serving from different nodes (creating a content distribution network). GANESAN et al [4] proposed a partitioning technique on large-scale parallel databases in peer-to-peer (P2P) systems. As tuples are inserted or deleted, the partitions are adjusted, and data are moved to achieve storage balance across the participating nodes. In this approach, each participant server needs to be well structured and informed about the whole P2P overlay network. In contrast, our approach treats all the participant servers as black boxes and only uses their interfaces for data access. WENG et al [5] defined a partial replication framework to speed up the execution of range queries. In partial replication, a portion of dataset, which is likely to be accessed more frequently, is extracted, re-organized, and re-distributed across the storage system. Again, in this approach, the participant nodes need to be informed about the whole system and must have extra services enabling them to communicate. BEYNON et al [6] introduced a technique to support execution of subsetting and aggregation operations for querying large datasets in parallel. The authors focused on optimizing ordinary striped access to uniformly distributed datasets, such as images, maps, and dense multi-dimensional arrays. However, their technique might not give the expected performance gains when using non-uniformly distributed datasets such as Earth related vector datasets (census data, earthquake seismic data, etc.).

Indexing is a well-known and widely used approach to the optimization of query execution time for non-uniformly distributed datasets [7]. Indexing keeps metadata for datasets and enables efficient searching and extraction for a given query. The Seti project proposed an index structure for efficient range queries on non-uniformly distributed spatial trajectory datasets [8]. To create an index, an R-tree structure was used. A related work was the TrajStore project [9]. The researchers concentrated on trajectory datasets took the dense and sparse regions into consideration and proposed a 2-level index structure. They proved that the results were much better than R-tree based indexes for 2-dimensional range queries. Both projects focused on creating index enabled range queries on non-uniformly distributed datasets, but they used local datasets. However, when the datasets are distributed and provided by third party services, range query optimization becomes more challenging. Handling distributed datasets in a performance efficient way requires addressing the issues related to replications [5], parallel queries [6], load-balancing (I/O parallelism) and declustering. Some of these issues have overlapping interests. In particular, declustering [10] is an approach for the creation of an efficient query framework using

indexlike structures. In declustering, datasets are distributed across several storages in a well-structured manner and information about the distribution is kept in an index file. Query efficiency depends on how well datasets are distributed across the storage nodes and how well the index structures are compatible with the data characteristics and query attributes.

This work solved the query approximation problem to partition and shared the workload across the replicated data nodes. A load-balancing (I/O parallelism) approach was introduced for multi-dimensional distributed range queries in replicated environments. The datasets were not local or centralized. Therefore, the query size and distribution characteristics of data of varying ranges were not known a priori, and the data servers were treated as black boxes. To optimize the load balancing and efficiency of parallel queries, the data dense/sparse regions were taken into consideration and approximated before partitioning the original query [11]. Having identified roughly equivalent partitions (in terms of number of objects contained) offline, the resulting index was used to split range queries into subqueries (corresponding each to a spatial partition), which were then distributed to the various servers in a round-robin fashion. The proposed index table is called WET, which is similar to a balanced *k*d-tree (short for *k*-dimensional tree) index structure [12]. WET is also a special case of binary space partitioning trees. It is not only an index but also provides ready-to-use subranges of almost equal sizes of query payloads. The terms of subranges and partitions are used interchangeably throughout the work.

3 Architecture: Adaptive range query optimization with distributed replicas

The proposed architecture is composed of compile-time and run-time approaches for efficient parallel execution of 2-dimensional range queries when the distributed replicas of a dataset exist. During compile-time, the system collects information about the dataset and prepares an indexlike table to efficiently partition the whole dataset into subranges (smaller partitions). The query payloads that correspond to the ranges in the table are expected to be equal in size. At run-time, the system makes parallel queries in accordance with the subranges calculated at compile-time and then aggregates the results to create the answer for the main query. The architecture aims at eliminating data and execution skew for efficient I/O parallelization. The proposed architecture promises some advantages over naive (straightforward) partitioning such as smaller response time, especially in systems with a high degree of parallelism, and efficient usage of replica nodes through load balancing.

3.1 Problem formulation

Figure 1 illustrates the problem regarding efficient partitioning of ranges for non-uniformly distributed datasets. Although this is a general class of problem, for illustration purposes, geospatial domain is selected. Figure 1(a) shows the city boundary data in the form of linestrings or polygons, and Fig. 1(b) shows the seismic data in the form of points. Those are spatial datasets and defined/queried with 2-dimensional, (x, y) Cartesian coordinates. The set of (x, y) coordinate values is accessed with range queries. Ranges are called minimum bounding rectangles (MBR) or bounding boxes. Both are the same and formulated as R=[(min$x$, min$y$), (max$x$, max$y$)]. (min$x$, min$y$) refers to the lower left, and (max$x$, max$y$) refers to the upper right corners. min$x$, min$y$, max$x$ and max$y$ are assumed to be integers or rational numbers.

Considering a dataset as a subset of 2-dimensional rational space, Q^2, there are unlimited numbers of rectangle (i.e., ranges) that can be defined over the data space R. Let's say data space R is partitioned into non-overlapping rectangular regions $\{r_1, r_2, r_3, \cdots, r_n\}$ where $r_i = [(x_i^1, y_i^1), (x_i^2, y_i^2)]$. The number n changes depending on the average sizes of r_i set. As the average sizes of r_i increase, n decreases.

If the area of input rectangles (r_i) is denoted by $A(r_i)$, then, $A(R) = \sum_{k=1}^{n} A(r_i)$.

If the query payloads of the input rectangles (r_i) are denoted by $L(r_i)$, then, $L(R) = \sum_{k=1}^{n} L(r_i)$.

The ordinary binary partitioning shown in Fig. 1 gives the best results for maximizing the performance in parallel queries if the datasets were uniformly distributed. To uniformly distribute datasets, query payloads can be inferred from their range sizes. In other words, sizes of partitions (r_i in the figure) are proportional to their corresponding payloads (number of stars representing point data). However, optimal partitioning of such data shown in Fig. 1 is difficult to achieve because polygons, line-strings, points, etc., are neither distributed uniformly nor of similar sizes. Optimal partitioning is a process in which computational load on each partition ($L(r_i)$) becomes roughly the same. $L(r_i)$ represents the number of points.

Load is mainly a cost measure to determine the sizes of the partitions. The goal is to find out the most efficient number of partitions and the size for a given data space. The aim is to cut the data space R into smaller pieces (r_i) with approximately equal loads ($L(r_i)$) and to enable the efficient load balancing for the parallel queries. The solution approach is based on creating and using WET, which is actually a 2-dimensional tree.

The research problem can be illustrated on a sample scenario. We have two replicated data servers serving replicated data whose population in 2-dimensional space is illustrated in Fig. 1(b). When doing a naive partitioning (as illustrated in Fig. 1(b)) and assigning the partitions to the replica servers in round-robin fashion, the first replica server gets partitions r_1 and r_3, and the second replica server gets partitions r_2 and r_4. As a result, the first replica server serves 5 points and the second replica server serves 27 points. This is not a fair sharing of workload. By contrast, optimal partitioning is expected to result in a workload sharing where each replica server gets an equal number of points to serve, i.e., 16 points. Even for such a small dataset, performance gain would be almost two folds.

3.2 Compile-time (indexing)

At compile-time, the system prepares an indexlike structure called WET. This section elaborates on WET and its functionalities in sharing the main query payload among the replica nodes.

WET is a representation of the data distribution characteristics in the form of a list of small ranges whose query sizes are relatively close to each other. WET is an index table representing related data in a remote database. Due to the dynamic nature of data, WET is created once and synchronized/refined at time intervals to reflect the changes in a remote database. Time intervals are defined by application developers. The interval change depends on the data dynamicity. WET carries only the minimum bounding rectangles, not the actual data. Our aim is to partition the data space into rectangular regions in such a way that each pair of rectangular regions overlaps only at the boundaries and each rectangular region's corresponding query sizes are as equal as possible. Our algorithm continues to partition the rectangular regions recursively until the query sizes for subregions are less than or equal to threshold query payload size (t). In addition, the size differences between the partitions, i.e., subregions (fluctuation), are controlled by the error parameter (E_r). E_r is a tunable parameter that can be set to any value such that $0< E_r <1$. As E_r gets closer to 0, partitions' query sizes get closer to each other and WET provides better information for I/O parallelization. On the other hand, the time to create WET increases. At each iteration, a region (R) is partitioned into two subregions (R_1 and R_2). Eventually, at the end of the recursive iterations, every partition R_i has a query payload less than or equal to t, and the ratio of query payloads of any R_i and R_j becomes close to 1 depending on the pre-defined E_r value. The formula of the recursive partitioning algorithm is given in Eq. (1).

The recursive algorithm to create/refine WET is

$$\mathrm{PT}(R, t, E_r)=\mathrm{PT}(R_1, t, E_r)+\mathrm{PT}(R_2, t, E_r) \quad (1)$$

where PT is the main routine creating/refining WET, R is the overall range covering all the data in the database, t is the threshold query payload size (allowable maximum query size for a partition), which is a predefined value, E_r is the maximum allowable query size difference between subregions obtained from binary cut, which is a predefined value as

$$E_r = \frac{|L(R_1)-L(R_2)|}{\max[L(R_1), L(R_2)]} \quad (2)$$

PT(R, t, E_r) is a routine to recursively partition the region R into subregions whose corresponding query sizes are less than t, as shown in Fig. 2. PT is based on binary partitioning but makes a balanced partition at each iteration as improvement. To make balanced partitions, PT routine calls the PTInBalance (Fig. 3) subroutine. PTInBalance finds the most efficient cutting point (MP) for a range R in which the ratio of the difference in query payload sizes to the maximum of the query payload sizes is less than E_r. PTInBalance (Fig. 3) is a way of determining the center of gravity for range R in terms of query payload. The query payload for each range is obtained by getData(R_i) routine. It makes a remote call to the database and gets the actual data. The time to finish partitioning of R into R_1 and R_2 depends on the error parameter E_r. As E_r decreases, the time for algorithm to finish increases. The algorithm produces a WET in which there is no subregion whose query size is larger than any other subregion's query size more than (1+E_r) times.

PTInBalance(R, E_r) does not take threshold data size as parameter because its task is only cutting the given region R into two subregions whose query sizes fluctuate with the error E_r. The algorithm interacts with the remote data server at each iteration and makes queries with newly calculated ranges. According to the results of the query sizes, it adapts the ranges and repeats the same thing with newly calculated queries. It keeps doing this until the query sizes for the partitions get close to each other based on predefined E_r. If E_r is defined as 0, it means both query sizes for the partitions are equal. In

```
PT(R, t, Er)
   [(R1, size1):(R2, size2)]=PTInBalance(R, Er)
   If ((size1≤t) or (size2≤t)) {*/sizes are almost the same/*
      /*Put the partitions into memory/disk as paris*/
      Store(<R1, size1>)
      Store(<R2, size2>)
      Return
   }
   Else
      PT(R1, t, Er); PT(R2, t, Er)   /**recursive calls/
}
```

Fig. 2 Recursive binary partitioning routine

that case, all the partitions are of equal size that is equal to the threshold data size t.

Let's make it clear how the algorithm works on the sample dataset given in Fig. 4(a). The coordinate values of the data lie in range R, R=((0,0), (1,1)). Figure 4(b) shows a sample WET created from the sample dataset in Fig. 4(a). Since the total query size in database (Load) is 32 MB, t is 5 MB, and E_r is 0.2, the minimum partition size is calculated as 4 MB. After running the algorithm given above, we get a WET which has 7 partitions as listed below:

((0.00, 0.00), (0.45, 0.56)), ((0.45, 0.00), (0.74, 0.56)), ((0.74, 0.00), (1.00, 0.64)), ((0.74, 0.64), (1.00, 0.81)), ((0.74, 0.81), (1.00, 1.00)), ((0.64, 0.56), (0.74, 0.75)) and ((0.00, 0.56), (0.64, 1.00)).

Figure 4 illustrates that the regions having higher data density are most probably cut into a higher number of subregions, this partition indicates unit query ranges for parallelization.

The complexity analysis of WET table creation is studied. WET table is similar to kd-tree structure. As in kd-trees, R is split into 2 equal-sized subregions. Let $T(R)$ be the time needed to build a WET table on range R; and $T(1)=O(1)$, and then $T(R)=2T(R)+x$. In the case of kd-trees, x represents the time to calculate the median of a set of n values, which is computed in $O(n)$ time. Overall kd-tree creation takes $O(n\log n)$ time according to the master theorem case-2. However, in the case of WET creation, since the data are not local, and their distribution is not known a priori, it is not possible to clearly define the time x that is required to divide R into almost equal-sized subregions. x is actually the time to finish PTInBalance subroutine for one partitioning. PTInBalance includes a remote subroutine called getData, and it makes computation complexity hard to solve. Since the remote data servers are taken as black boxes, their computation complexities can not be estimated. If PTInBalance takes linear time in terms of range R, then WET table creation takes $O(R\log R)$ time. If it takes quadratic time, WET table creation takes $O(R^2)$ time.

3.3 Run time

The output of the compile-time query planning is

```
PTInBalance(R, E_r) {                          /*Like finding out center of gravity*/
   current_E_r=1;
   l=maxx
   r=maxx
   While (current_E_r>E_r) {
      mp=(l+r)/2
      R_1=minx, miny, mp, maxy    /*R=R_1+R_2*/
      R_2=mp, miny, maxx, maxy
      data_1=getData(R_1)          ---> Remote data access to find out data
      data_2=getData(R_2)               size for corresponding range/partition
      If (size(data_1)>size(data_2));
         {r=mp}
      else
         {l=mp}
      current_E_r=(size(data_1)−size(data_2))/max[size(data_1), size(data_2)]
   }
   return [(R_1, size(data_1)):(R_2, size(data_2))]
}
```

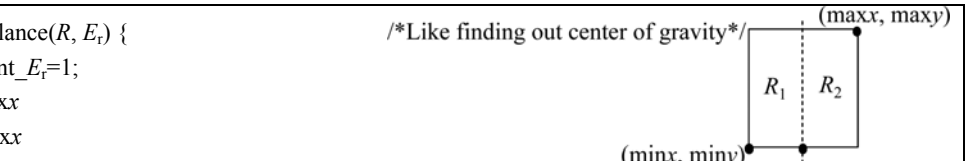

Fig. 3 Routine to find out best point to cut range R

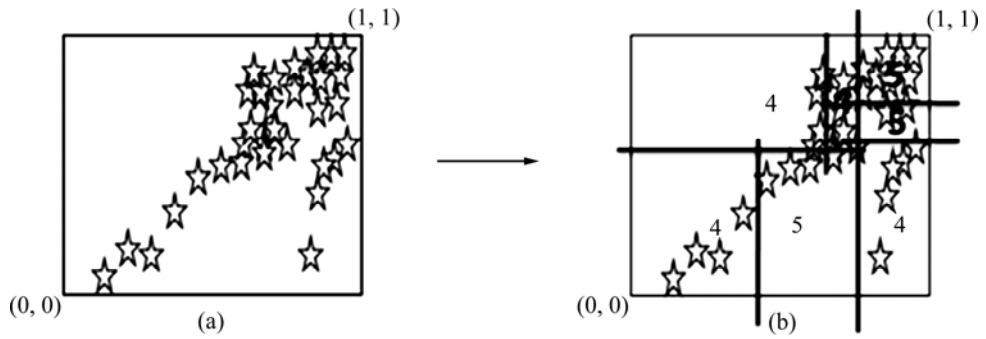

Fig. 4 Sample data space (a) and corresponding partitions (b) in WET

used at run-time to determine the best set of ranges to maximize I/O parallelism. At run-time, the system inspects the subranges and creates the parallel queries for subranges for a given query, hands them off to the replica nodes, and aggregates the results from subrange queries to create the response for the main query range. Information about the replica nodes is kept in a local file. This file is prepared at compile-time. The execution steps are elaborated as follows.

The client-side interaction is handled at run-time. As a first step, the main query range is positioned on WET and overlapping subranges in WET are calculated. The output of this work is the set of rectangles overlapping with the main query range. This is only a comparison of rectangles to see if they overlap in 2-dimensional space.

Let's illustrate this with a sample scenario, as illustrated in Fig. 5. The sample main query with range R is positioned on WET. R overlaps with p_5, p_6, p_7, p_8, p_9 and p_{10}. The set of ranges on which parallel queries are performed are calculated as p_5, p_6, p_7, p_8, r_1 and r_2. If the application does not treat partial and total overlapping ranges differently, then the selected ranges are calculated as p_5, p_6, p_7, p_8, p_9, and p_{10}, and r_1 and r_2 are the partial overlappings with p_9 and p_{10}, respectively. Application developers can choose partial or total overlapping depending on the data formats they use. For the point datasets, partial overlapping would give the best results but for polygon datasets both would give almost the same result.

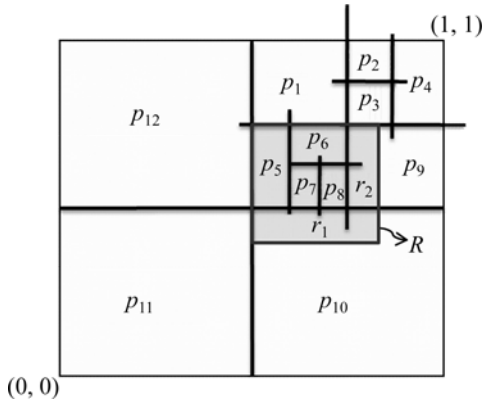

Fig. 5 Illustration of query decomposition with sample scenario

In the second step, the system creates/emulates the main query with the newly calculated subranges. In other words, parallel queries are created for the subranges. The subqueries inherit all the attributes from the main query except for the range attribute defining their rectangular shapes. As shown in Fig. 5, queries created for the partitions p_5, p_6, p_7, p_8, r_1 and r_2 have all of the same attributes except for their ranges. The structures of the queries change depending on the applications and remote servers. For example, remote servers might be common database servers or databases might be integrated to the system through middleware with standard service interfaces.

After creating subqueries for each partition, queries are assigned to the replicas through a separate thread of works in round-robin fashion [13]. The technique presented here ensures that each replica gets almost equal number of partitions. Moreover, since the subranges have almost equal query payloads, the replicated nodes have an almost equal workload. The subqueries are assigned to separate threads to capture the data from replica. The number of partitions each replica is calculated as

$$S = \left\lfloor \frac{N_{\text{partition}}}{N_{\text{replicas}}} \right\rfloor \tag{3}$$

$$N_{\text{rmg}} = N_{\text{partition}} - SN_{\text{replicas}} \tag{4}$$

where S is the share number of subqueries and N_{rmg} is the number of subqueries remaining ($0 \le N_{\text{rmg}} \le N_{\text{replicas}}-1$). If there are no subqueries remaining, then $N_{\text{rmg}}=0$ and every replica node is assigned a share number of subqueries. If N_{rmg} is different from 0, then the first N_{rmg} replicas are assigned S+1 subqueries each and the remaining replicas are assigned share subqueries each. This scenario of sharing is a natural result of the round-robin approach. After all threads are done with their work, the master thread aggregates the results, creates the final response to the main query, and sends the main query response to the client.

4 Results

2-dimensional range queries are used frequently in various applications such as spatial databases, geospatial information systems (GIS), computer vision, computer aided design (CAD) in engineering, and astronomy. In such applications, the data points are usually represented by 2-dimensional vectors corresponding to their locations. In addition, datasets are defined with both geometric and nongeometric attributes. In this work, the proposed technique in the (GIS) domain is examined and evaluated, which manages geographic datasets having geometric attributes such as points, lines, polylines, and polygons. These attributes are defined with point or point sets in some predefined standard formats (open geospatial consortium (OGC)) and accessed-queried with minimum bounding rectangles (MBR), also called bounding boxes.

This section presents the application of the proposed query optimization technique to distributed map rendering [14−15]. The system is a collection of OGC compatible GIS web services [16]. A map is composed of multiple data-layers. Layers are visual representations of

homogeneous geographic datasets provided by web map service (WMS) and web feature service (WFS) in the form of images and GML data, respectively. Geographic markup language (GML) data provided by WFS are converted to map layers at the WMS side. This is called vertical composition of map layers. Each layer in a map can be partitioned into addressable smaller partitions (bounding boxes) and each chunk can be handed off to a replica node. Replica nodes are WFS [17] or WMS [18]. This is called horizontal composition and it requires extra work on the side where the partitioning and composition work is done. The technique presented in Section 3 is for horizontal decomposition, which is based on query decomposition. Queries are 2-dimensional range queries defined in rectangular shapes and formulated as (min$x$, min$y$, max$x$ and max$y$). Figure 6 shows sample range query (x, y, x', y'), and its partitions (R_1, R_2, R_3, and R_4).

For the test case scenario, 2-layer map images are studied. The first layer (base-map) is LandSat imagery of Turkey, obtained from the NASA OneEarth project's OGC compatible WMS. The second layer is earthquake seismic data of Turkey recorded from 1992 to 2005. These feature datasets are kept in a database and served to the system through standard WFS web service interfaces. The WFS are replicated nodes for the test evaluations. Earthquake seismic data have some major attributes such as magnitude, location (x and y coordinates), date/time, and some other minor attributes. Queries to create maps from those datasets are done based on these attributes. Varying compositions of those parameter values change the queries' payloads.

The test setup shown in Fig. 6 illustrates two approaches, one is the straight-forward query approach and the other is the parallel query approach. Queries from clients are created by the interactive map tools originally developed for standard WMS services. Our implementation of WMS, which is called federator in Fig. 6, is enhanced with the proposed technique. The efficiency of our solution approach on such situations is demonstrated in a use case scenario (Fig. 6). This setup is a real world application developed for distributed earthquake analysis.

Performance is evaluated with earthquake seismic data kept in relational tables in MySQL database and integrated into the system through WFS. There are four computers with quad-cores each, and each computer has processors running at 2.33 GHz with RAM of 8 GB, and located in wide area network (WAN). There are four replicated WFS servers deployed on those computers. The active number of WFS at anytime during the test is gradually increased for the purpose of the performance evaluation (see Fig. 7).

In our experiments, we seek to answers to the following questions with the evaluation tests:

1) How do the number of replicas (WFS) and number of partitions (R_i) together affect the performance?

2) How is the number of partitions (for a specific query size) affected by the WET's threshold query size (t)?

3) When the number of replicas (WFS) is kept the

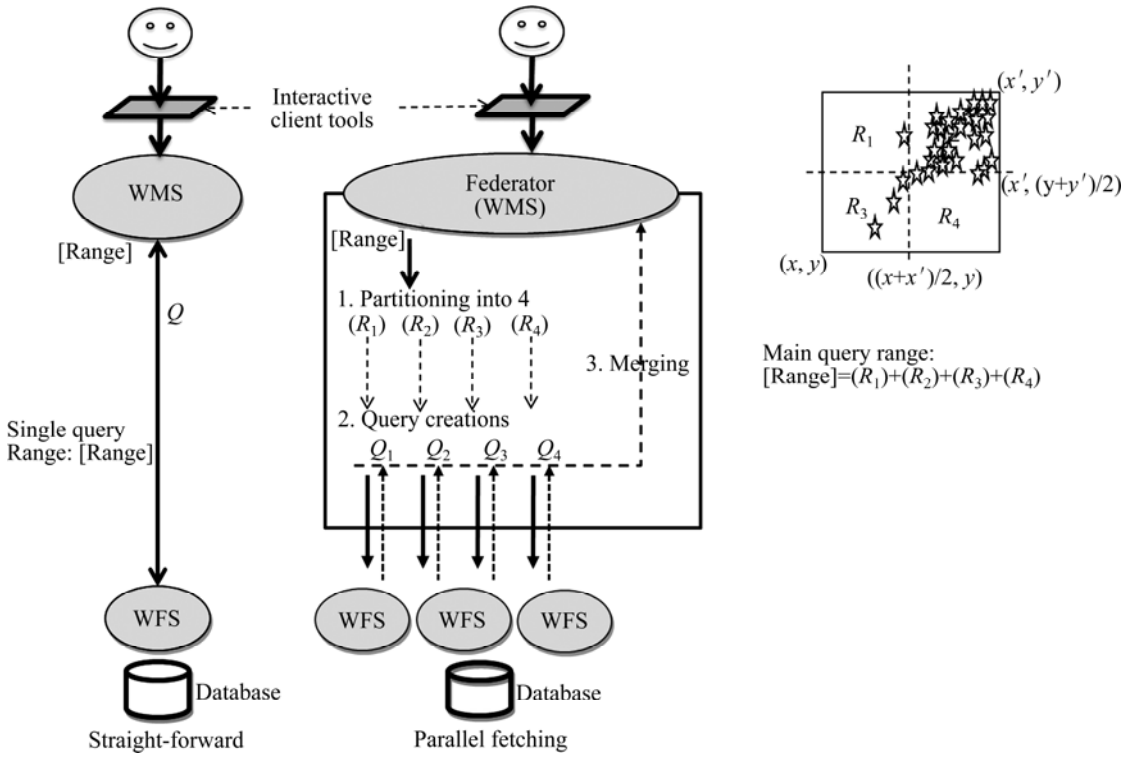

Fig. 6 Test setup

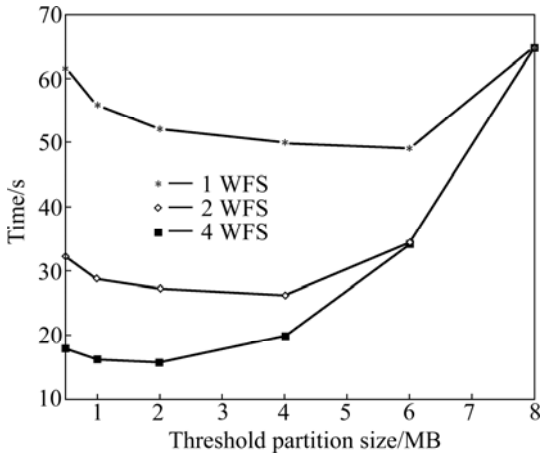

Fig. 7 Parallel query optimization performance

same, how does the partition-threshold size in WET affect the performance?

The values in Figs. 7 and 8 are obtained by running the tests on 10 randomly selected different regions (R). The areas of those regions (data space) are of varying sizes ($A_{\text{rea}}(R_i)$) but their corresponding query sizes are of 10 MB ($L_{\text{oad}}(R_i)$). As seen from Fig. 7, the system accesses the 10 MB of data in 65.06 s when the single query is used (see also the straightforward approach in Fig. 6). The average number of parallel queries is defined by the region's data distribution characteristics, the parameters used to build WET (threshold query size t and error E_{r}), and the actual main query size. WET built with different threshold query sizes might give different numbers of partitions for the same query ranges.

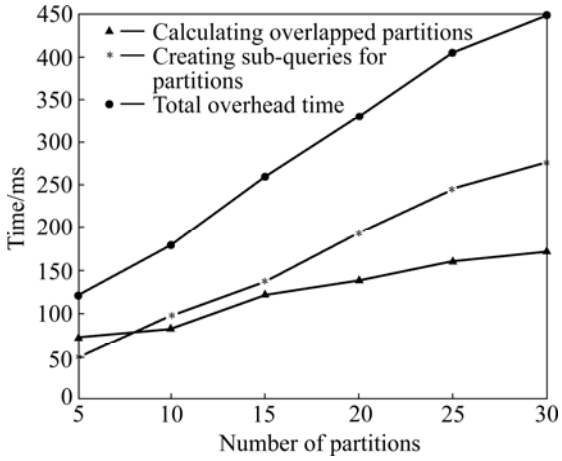

Fig. 8 Overhead times of parallelization

As illustrated in Fig. 7, in the case of having 4 replicas (WFS), the best performance is obtained by using 2 MB of threshold partition size. In such a case, the response time for 10 MB of query payload becomes 12.87 s. If the query is not parallelized, then for the single query of 10 MB payload the response time would be 65.06 s. As a result, the system is almost 5 times faster with the proposed technique using the WET index table for creating parallel queries.

The performance gain from the parallel querying increases as the partition number increases, as long as there are enough replicas. Above a specific threshold partition size, however, the ratio of the performance gain starts decreasing. This specific partition size changes depending on the other parameters such as number of replicas, error and query sizes. This pattern can be seen for each line representing a different number of replicas (Fig. 7). For example, in the case of using 4 WFS (circled-line), the threshold value for the partition size is about 2 MB. At this threshold value, the average number of partition is 8.5. The initial increase is due to improved load balance by reducing the effect of fluctuation in partitions' loads, and the decrease is due to the nonparallelizable overheads and limited number of replicas. In addition, success of parallel queries is based on how well we share the workload with replica nodes.

As E_{r} decreases, the balance in the workload share increases and gives better average query response times. It is explained through I/O parallelism, which is a natural result of the algorithm used to create WET (see Figs. 2 and 3). As the E_{r} value gets smaller, WET refinement takes a longer time. However, this is done at compile-time, and it does not affect the response time of the actual queries from the clients. As the number of replicas increases, the performance increases. As the threshold query size decreases, the fluctuation in query sizes between the partitions decreases and the degree of equal workload sharing increases.

As Fig. 7 illustrates, the response time does not scale linearly with the number of replicas. This is due to the decreasing threshold query size (t), which is represented by the x-axis. t is explained earlier in Eq. (1) and the following paragraph. The partition query sizes are around t and fluctuate with E_{r}=0.2. For queries of the same size (such as 10 MB as in Fig. 7), if t decreases, then both the number of partitions and the number of corresponding queries increase, and as a result, each replica gets more queries in parallel. An increased number of parallel queries on a WFS causes performance degradation because of the context switch time and related overheads. To eliminate this problem, we need to find a way to derive the cutoff parameter, the threshold partition size that leads to the best performance. This will be studied in the future.

Figure 8 illustrates the overhead time from partitioning and parallel processing. There are three overhead time compared to straightforward single process work, calculating overlapped partitions, subquery creations, and aggregating the subquery results. The figure also shows the pattern of the changes in the overhead time, according to the changing partition

numbers, and their relative weights in total overhead. Because of the overhead time, if we do an unnecessary amount of partitioning, then there will not be a performance gain for less than a threshold-data size, but we see from the figure that it is less than some small amount that does not affect the overall performance considerably.

5 Conclusions

1) A compile/run-time approach is presented for execution of range queries on distributed nodes when replica optimization is employed. The technique is based on WET, which estimates the main query payload and partitions it into the most efficient numbers and sizes of subqueries.

2) The solution presented in this work treats the data servers as black boxes and does not touch their inner structure. Data servers are used only through their publicly available standard service interfaces.

3) The proposed approach is based on approximating the query size in advance. Algorithms for eliminating data skew is focused to achieve efficient I/O parallelization, although the proposed technique can be generalized to handle execution skew as well. The system runs the best for point data sets. However, in case of polygen data sets, the system might not give the expected performance gains.

References

[1] TANENBAUM A S, van STEEN M. Distributed systems: Principles and paradigms prentice hall upper saddle river [M]. 2nd ed. Upper Saddle river, USA: 2006: 704.

[2] CHEN C M, CHENG C T. Replication and retrieval strategies of multidimensional data on parallel disks [C]// CIKM ’03 Proceedings of the Twelfth International Conference on Information and Knowledge Management. New York, NY, USA: ACM Press, 2003: 32−39.

[3] CHERVENAK A, DEELMAN E, FOSTER I, GUY L, HOSCHEK W, IAMNITCHI A, KESSELMAN C, KUNSZT P, RIPEANU M, SCHWARTZKOPF B, STOCKINGER H, STOCKINGER K, TIERNEY B. Giggle: A framework for constructing scalable replica location supercomputing [C]// ACM/IEEE 2002 Conference. Baltimore, USA: IEEE Computer Society Press, 2002: 1−17.

[4] GANESAN P, BAWA M, GARCIA-MOLINA H. Online balancing of range-partitioned data with applications to peer-to-peer systemsvery large database (VLDB) [C]// Proceedings fo the 30th International Conference on Very Large Data Bases. Toronto, Canada: Morgan Kaufmam, 2004: 444−455.

[5] WENG L, CATALYUREK U, KURC T, AGRAWAL G, SALTZ J. Servicing range queries on multidimensional datasets with partial replicas IEEE international symposium on cluster computing and the grid [C]// CCGrid 2005. Cardiff, UK: IEEE, 2005: 726−733.

[6] BEYNON M, CHANG C, CATALYUREK U, KURC T, SUSSMAN A, ANDRADE H, FERREIRA R, SALTZ J. Processing large-scale multi-dimensional data in parallel and distributed environments [J]. Parallel Computing-Parallel Data-Intensive Algorithms and Applications, 2002, 28(5): 827−859.

[7] DEWITT D, GRAY J. Parallel database systems: The future of high performance database systems [J]. ACM Communications, 1992, 35(6): 85−98.

[8] CHAKKA V P, EVERSPAUGH A, PATEL J M. Indexing large trajectory data sets with SETI [C]// Conference on Innovative Data Systems Research (CIDR-2003). CA, USA: VLDB, 2003: 281−291.

[9] MAUROUX C P, WU E, MADDEN S. TrajStore: An adaptive storage system for very large trajectory data sets [C]// IEEE 26th International Conference on Data Engineering (ICDE). Long Beach, CA: IEEE Press, 2010: 109−120.

[10] JOEL B M, SALTZ J H. Scalability analysis of declustering methods for multidimensional range queries [J]. IEEE Transactions on Knowledge and Data Engineering, 1998, 10(2): 310−327.

[11] BENTLEY J L. Multidimensional binary search trees used for associative searching [J]. Communications of the ACM, 1975, 18(9): 509−517.

[12] FILHO Y S. Avarage case analysis of region search in balanced k-d trees [J]. Information Processing Letters, 1979, 8(5): 219−223.

[13] TANENBAUM A S. Modern operating systems [M]. New Jersey, USA: Pearson Prentice Hall, 2008.

[14] SAYAR A, PIERCE M, FOX G. Developing GIS visualization web services for geophysical applications [C]// Turkey: ISPRS Spatial Data Mining Workshop Ankara, 2005.

[15] SAYAR Ahmet, PIERCE Marlon, FOX Geoffrey-Charles. Grid technology for maximizing collaborative decision management and support: Advancing effective virtual organizations [M]. Bedfordshire, UK: IGI Global-Information Science Reference, 2009: 360−368.

[16] AYDIN G, SAYAR A, GADGIL H, AKTAS M S, FOX G C, KO S, BULUT H, PIERCE M E. Building and applying geographical information systems grids [J]. Concurrency and Computation: Practice and Experience, 2008, 20(14): 1653−1695.

[17] VRETANOS P A. Web Feature Service Implementation Specification [EB/OL]. 2002−11−02.

[18] BEAUJARDIERE J. OGC Web Map Service Interface [EB/OL]. Open GIS Consortium Inc. (OGC), 2006−03−15.

(Edited by FANG Jing-hua)