Load-balancing Remote Spatial Join Queries in a Spatial GRID

Anirban Mondal Masaru Kitsuregawa

Institute of Industrial Science
University of Tokyo, Japan
{anirban,kitsure}@tkl.iis.u-tokyo.ac.jp

Abstract. The explosive growth of spatial data worldwide coupled with the emergence of GRID computing provides a strong motivation for designing a spatial GRID which allows transparent access to geographically distributed data. While different types of queries may be issued from any node in such a spatial GRID for retrieving the data stored at other (remote) nodes in the GRID, this paper specifically addresses spatial join queries. Incidentally, skewed user access patterns may cause a disproportionately large number of spatial join queries to be directed to a few 'hot' nodes, thereby resulting in severe load imbalance and consequently increased user response times. This paper focusses on load-balanced spatial join processing in a spatial GRID.

1 Introduction

The explosive growth of spatial data worldwide coupled with the prevalence of spatial applications has made efficient management of geographically distributed spatial data a necessity. Spatial applications often arise in town planning, cartography, resource management, GIS (Geographic Information Systems), CAD (Computer-Aided Design) and computer vision. Incidentally, the emergence of GRID computing [4], which is associated with the massive integration and virtualization of geographically distributed computing resources, provides a strong motivation for designing a spatial GRID [11] which allows transparent access to geographically distributed data. While different types of queries (e.g., spatial select queries[1], nearest neighbour queries, similarity search queries and spatial join queries) may be issued from any node in the GRID for retrieving the data stored at other (remote) nodes of the GRID, this paper specifically addresses spatial joins on remote data since such queries constitute a typically expensive as well as popular class of query in spatial databases. Incidentally, a spatial join query retrieves from two spatial relations all the tuple pairs satisfying a given spatial predicate.

Now let us understand the importance of optimizing remote spatial joins with the help of an example. This year's Olympic Games is expected to attract tens of

[1] Our previous work [11] studied load-balancing of spatial select queries in a spatial GRID.

thousands of visitors to Athens and many of these visitors would possibly wish to find a hotel near a bus station for the purpose of convenient transportation. Such visitors may issue the following query from their respective home countries which may be quite far away from Athens: *Find all the hotels in Athens which are near to any bus station.* Assuming there are two relations *'Hotels'* (containing details, such as location, rental charges, of all the hotels in Athens) and *'Bus Stations'* (containing information concerning all the bus stations in Athens), this translates to a remote spatial join operation. Interestingly, the above scenario is also *equally* applicable to any major international event that attracts people from various countries. Notably, the recent trend of increased globalization has significantly increased the importance as well as the performance demands of such global applications. Unfortunately, the current state-of-the-art does *not* allow a user to perform this kind of operation efficiently.

Skews in initial data distributions, skewed user access patterns and changing popularities of data regions may cause a disproportionately large number of spatial join queries to be directed to a few 'hot' nodes, thereby resulting in severe load imbalance and consequently increased user response times. From our example, given the huge number of queries from potential visitors to Athens, the nodes containing the data of Athens would quickly become overloaded, thereby necessitating a load-balancing mechanism for processing remote spatial joins efficiently. Several factors such as wide-area communication overheads, node heterogeneity and lack of centralization make the problem of load-balanced spatial join processing in GRIDs significantly more complex than that of load-balanced spatial join processing in traditional distributed environments such as clusters. However, we believe that the time has come to deal head-on with this problem. The main contributions of our proposal are as follows.

- We present a dynamic data placement strategy involving *online* data replication in GRIDs, the objective being to bring the data closer to the node from which it is frequently queried.
- We propose a novel load-balancing strategy for speeding up spatial joins in GRID environments.

Our performance evaluation demonstrates the effectiveness of our proposed approach in reducing the response times of spatial joins in GRIDs. To our knowledge, this work is one of the earliest attempts at addressing the load-balancing of remote spatial joins via *online* data replication in GRID environments. The remainder of this paper is organized as follows. Section 2 discusses related work, while Section 3 presents the system overview. Issues concerning load-balancing in spatial GRIDs are presented in Section 4. The proposed strategy for load-balancing spatial joins in GRIDs is discussed in Section 5, while Section 6 reports our performance evaluation. Finally, we conclude in Section 7.

2 Related Work

Important ongoing GRID computing projects such as the European DataGrid [2], the Grid Physics Network (GriPhyN) [13] and the Earth Systems Grid (ESG)

[6] aim at efficient distributed handling of huge amounts of data (in terabyte or petabyte range).

Issues concerning spatial databases with GIS applications can be found in [15], while a comprehensive survey on spatial indexes has been presented in [5]. Parallel spatial join processing has been extensively researched in the traditional domain. The proposals in [7] and [1] discuss synchronous traversal of generalization trees and R*-trees respectively. The work in [1] investigates parallel *load-balanced* spatial join processing using R*-trees on a shared-virtual-memory architecture. The PBSM (Partition Based Spatial-Merge) algorithm [12] first partitions the inputs into smaller chunks and uses a computational geometry based plane-sweeping technique to obtain a set of candidate pairs and then the tuples corresponding to the candidate set are fetched from disk to determine whether the join condition is actually satisfied. The work in [10] proposes a parallel non-blocking spatial join algorithm which uses duplicate avoidance and addresses main memory issues. Several dynamic [16] load-balancing techniques for clusters have also been proposed specifically for *clusters*. Notably, neither the the existing spatial join techniques nor the existing load-balancing strategies consider GRID-related issues such as heterogeneity and wide-area communication overheads essentially because these issues do *not* arise in traditional environments.

Incidentally, our proposal amounts to some form of caching of the results. Recent works on caching include [9, 14]. The work in [9] analyzes the effects of different design choices involving cache structure, cache capacity, and timeouts for caching previously discovered routes in demand routing protocols for wireless ad hoc networks. In [14], a semantic caching scheme has been proposed for accessing location-dependent data in mobile environments.

3 System Overview

We envisage the spatial GRID as comprising several clusters, where each cluster comprises nodes that belong to the same Local Area Network (LAN) [11]. This facilitates the separation of concerns between intra-cluster and inter-cluster load-balancing issues. Given that intra-cluster issues have been extensively researched, this paper specifically focusses on inter-cluster issues. For each cluster, the most reliable and best administered node is selected as the cluster leader, any ties being resolved arbitrarily. A cluster leader's job is to coordinate the activities (e.g., load-balancing, searching) of the nodes in its cluster. We define distance between two clusters as the communication time τ between the cluster leaders and if the value of τ for two clusters is less than a pre-defined threshold, the clusters are regarded as *neighbours*. A cluster C_i is considered to be *relevant* to a query Q_i if C_i contains at least a non-empty subset of the answers to Q. Given that the number of queries waiting in node N_i's job queue is W_i and taking the heterogeneity in node processing capacities into account, we define L_{N_i}, the load of N_i, as follows:

$$L_{N_i} = W_i \times (CPU_{N_i} \div CPU_{Total}) \tag{1}$$

where CPU_{N_i} denotes the CPU power of N_i and CPU_{Total} stands for the total CPU power of the cluster in which N_i is located. The *load of a cluster* is calculated as $\sum L_{N_i}$ i.e, the sum of the loads of its individual members. Given that the loads of two clusters C_i and C_j are L_{C_i} and L_{C_j} respectively and assuming without loss of generality that $L_{C_i} > L_{C_j}$, the normalized load difference Δ between C_i and C_j is computed as follows:

$$\Delta = ((L_{C_i} \times TC_i) - (L_{C_j} \times TC_j))/(TC_i + TC_j) \qquad (2)$$

where TC_i is the sum of the CPU power of all the members of C_i. Similarly, TC_j is the sum of the CPU power of all the members of C_j.

Spatial indexing mechanisms may vary across clusters. Hence, we propose a generalized indexing scheme which is built on top of the existing index at a cluster node. The indexing scheme in each cluster comprises three index structures, namely IID (Index structure for Internal Data), IED (Index structure for External Data) and IRD (Index structure for Replicated Data). Note that we distinguish between a cluster's *internal data* (the data which is originally stored at a cluster) and its replicated data because it may *not* be possible to integrate the replicated data smoothly into the existing index structure (for the internal data) of the cluster as the replicated data may be far apart in space from the cluster's internal data. Such separation of concerns between internal data and replicated data also makes it easier to periodically delete infrequently accessed replicas for optimizing disk space usage. IID is a *generalized* two-tier indexing mechanism, the first-tier of which resides at the cluster leader C_i and is essentially a list, each entry of which is of the form (*region*,*node_id*), where *region* represents a specific region and *node_id* stands for the node in the cluster at which the region is located. At the second tier, every node has its own independent index structure for the data allocated to it. For example, in case of R-trees [8], rectangular-shaped regions would be stored in the first-tier at C_i, while the second-tier would comprise R-trees at the individual nodes. IED and IRD are hierarchical tree-based index structures, which reside at the cluster leader. In our example, IED/IRD would be R-tree-like structures except that their leaf nodes would contain cluster IDs of neighbouring clusters' data regions instead of pointers to objects in the database. Updates to IED/IRD are periodically exchanged between neighbouring cluster leaders preferably via piggybacking onto other messages. Notably, in this paper, we have used the R-tree as an example, but our proposed technique may also be applicable to other spatial indexing structures albeit with certain modifications. Hence, in the near future, we also plan to investigate the use of other spatial indexing structures for performing spatial joins in GRIDs.

When a cluster leader C_i receives a query Q, it first checks its IID and IRD to ascertain whether any of its cluster nodes is relevant to Q. If C_i finds that *none* of its nodes is relevant to Q, it checks its IED and sends Q to its neighbouring cluster leaders which are relevant to Q. In case *none* of its neighbouring cluster leaders contain the answers to Q, C_i broadcasts Q to *all* of them. This process continues till either the answers to Q are retrieved or Q is timed-out. Assume

the existence of n clusters in the GRID, $C_1, C_2, C_3, ...C_n$. Now suppose cluster C_i issues a spatial join query Q_i for the data in cluster C_j. A straightforward solution would be for C_j to process Q_i and return the results to C_i, but this solution would *not* be efficient in scenarios where many spatial join queries, which attempt to retrieve data from the same regions in C_j, are issued from C_i to C_j. Our primary focus is to reduce the response times of such spatial join queries on remote data via replication.

4 Issues concerning Load-balancing of Spatial Joins via Replication in GRIDs

This section addresses important issues which need to be addressed when supporting load-balancing of spatial joins via replication in GRIDs.

Hotspot detection

The heat of data regions should be defined with respect to clusters which issue queries for these regions. For example, if cluster C_S issues a large number of queries pertaining to data region D_i of cluster C_T, but cluster C_O issues no queries for D_i, D_i will be considered a 'hot' region only w.r.t. C_S (but *not* w.r.t. C_O). Understandably, many different clusters accessing D_i infrequently would make D_i a 'hot' region in the conventional sense, but replicating D_i at any of these clusters may *not* necessarily be useful for reducing response times and may indeed be counter-productive owing to replication-related overheads. For hotspot detection purposes, every node within a cluster maintains its own access statistics comprising a list $HotList$, each entry of which is of the form ($data$,ptr_{access}). Here $data$ represents a specific data region and ptr_{access} is a pointer to a three-dimensional array of the form (cluster_id,num,$avgtime$), where cluster_id stands for the ID of the particular cluster which accessed $data$, num indicates the number of times that cluster accessed $data$ and $avgtime$ represents the average processing time it took for a node to perform spatial join on $data$. The value of $avgtime$ is used in ascertaining the benefit of replicating a specific data region as we shall see in Section 5. This information is periodically sent by every node to its cluster leader, thereby enabling the cluster leader to determine the 'popularity' of different data regions with respect to different clusters.

For reflecting *current* hotspots accurately in dynamic GRID environments, we propose that $HotList$ should be initialized periodically i.e., the information in $HotList$ should be deleted periodically and then $HotList$ should be populated with fresh access information. Additionally, whenever an overloaded source cluster node N_i has completed offloading some part of its load to a node at another cluster, N_i refreshes its own $HotList$ by deleting those entries in $HotList$ which triggered the replication so that $HotList$ reflects hotspots concerning which action has *not* yet been taken. Moreover, we maintain access statistics information at the granularity of the respective leaf node levels of the index structures at the nodes. Throughout this paper, we shall use the term **data region** to indicate

the spatial region corresponding to the Minimum Bounding Rectangle (MBR) of the data stored at a leaf node of the IID at a particular node. An interesting question which arises here is: *Given that several different kinds of queries can be issued to a real system, how do we know whether the leaf node accesses are being made specifically for a spatial join query?* Incidentally, at the leaf node level, it is *not* feasible to determine the type of query for which the leaf node is being accessed, but this information can be found at the query engine level.

What to replicate?

Once a 'popular' region D_i w.r.t. a specific cluster C_i has been detected, our strategy is to replicate the *results of the spatial join operation on* D_i at C_i. Note that replicating the results (as opposed to replicating the data itself) can significantly benefit those subsequent spatial join queries whose spatial select windows have considerable overlap with D_i since C_i will *not* need to do any processing at all for a significant part of these queries. Additionally, replicating the results can reduce communication cost significantly if join selectivity of D_i is low. Even in case of high join selectivity of D_i, intuitively we can understand that the communication cost of replicating the result tuples can never exceed that of replicating the data itself. Our strategy assumes that the datasets are relatively static. Note that we use 'data replication' and 'replication of result tuples' interchangeably throughout this paper to imply replication of result tuples.

Exploiting overlap between different spatial join queries

Interestingly, every spatial join query has an associated MBR associated with it either explicitly or implicitly. We shall designate this MBR as SPJMBR (Spatial Join's Minimum Bounding Rectangle). For example, the join query *"Find a hotel near a station in Athens within a 5 km radius of X, where X is a certain landmark in Athens"* explicitly specifies the SPJMBR associated with the join query, while the query *"Find a hotel near a bus station in Athens"* implicitly specifies that the SPJMBR for this query corresponds to the MBR of Athens. Intuitively, efficient exploitation of overlaps between different spatial join queries requires a mechanism for storing the replicas in a manner which enables quick identification of overlap between spatial join queries and existing replicas.

In our proposed system, whenever a replica of result tuples is stored at a cluster, the cluster leader also stores the SPJMBR corresponding to that replica. Identification of overlap between an SPJMBR and a spatial join query Q can be classified into 3 cases: (a) Q's MBR does *not* intersect with SPJMBR: This implies that there is *no* overlap between the query and the existing replica. (b) Q's MBR is fully contained within the SPJMBR: This means that all the results tuples requested by Q are already in the stored replica and only a spatial select query using Q's MBR as the spatial select condition should be run on the replicated data to obtain the answers to Q. We propose to run this spatial select condition on the replicated data at the cluster C_{Rep} where the replicated data exists (if C_{Rep} is *not* overloaded and has sufficient disk space) to save

communication costs. However, if C_{Rep} is overloaded and/or C_{Rep} has insufficient available disk space, the replica is sent to the cluster C_{Issue} which issued Q and C_{Issue} needs to run Q's MBR as the spatial select query on the replica to obtain the query results. (c) Q's MBR partially intersects with SPJMBR: The implication is that the results of Q already exist for the intersecting part between Q's MBR and SPJMBR, but for the non-intersecting parts, the results need to be computed. In this case, the tuples in the intersecting part are sent to the query issuing peer, while a spatial join operation needs to be run to get the result tuples in the non-intersecting parts between Q's MBR and SPJMBR. This spatial join operation involving the non-intersecting parts of Q's MBR and SPJMBR should be run at C_{Rep} if C_{Rep}'s load is low, otherwise it should be executed at C_{Issue}.

Whenever a spatial join query Q arrives at a cluster leader, the cluster leader traverses its list of SPJMBRs and identifies and exploits overlaps (if any) in the manner stated above. Additionally, in order to optimize disk space usage, each cluster leader keeps track of the replicas in its cluster nodes as well as the number of accesses made to each of the replicas during recent time intervals. Replicas whose access frequency during recent time intervals falls below a pre-defined threshold are deleted because the valuable disk space consumed by such unused replicas can be put to better use by storing 'hot' data, thereby improving system performance.

5 Load-balancing Strategy for Spatial Joins in GRIDs

In our proposed strategy, a cluster leader determines itself to be overloaded if its load exceeds the average loads of its neighbouring clusters by more than $y\%$. (The value of y is application-dependent and in our case, we assume $y = 15\%$.) When a cluster leader determines itself to be overloaded, it periodically checks the frequency with which its **data regions**[2] are being queried by other cluster leaders during the recent time intervals. Based on this information, the cluster leader C_i creates a set ψ comprising all cluster leaders which have issued more than η queries for *any* of its regions. (η is a threshold parameter which influences the sensitivity of load-balancing.) C_i sends a message to each member of set ψ informing its own disk space requirement (i.e., the amount of disk space required to store the replicated data) and requesting information concerning their load status, list of neighbours and whether their available disk space is sufficient to store the replicated data. After receiving the necessary status information of ψ's members, C_i evaluates their replies one-by-one. Members of ψ whose available disk space is too low to store the replicated data or whose normalized load difference with C_i (Δ) falls below a pre-specified threshold are deleted from ψ. For such members, C_i adds their list of neighbouring clusters to ψ. For these neighbouring clusters, clusters with low disk space or those with low normalized load difference with C_i are deleted from ψ. The remaining members of ψ are

[2] Recall that 'data region' refers to the spatial region corresponding to the MBR of the data stored at a leaf node of the IID at a particular node.

candidates for replication. For each member α of ψ, C_i traverses each hot data region H that has been queried by α and decides whether to replicate the spatial join result tuples associated with H on a case-by-case basis. Now let us see how C_i makes this decision.

The total cost C_H of replicating H from C_i at α consists of the cost $Extr_H$ of extracting H from IID at C_i, the communication cost Cm_H of transferring H and the bulkloading cost $Bulk_H$ of integrating H into the IRD of α. Hence, C_H is given by the following formula:

$$C_H = Extr_H + Cm_H + Bulk_H \tag{3}$$

Recall that every cluster leader maintains information concerning the average processing time and the accesses made to each data region from each of the clusters in the system. Let n_H denote the number of times H has been accessed by α and $avgtime_H$ represents the average processing time of H. Hence, the benefit B_H of replicating H at α can be estimated as follows:

$$B_H = (n_H \times avgtime_H) \tag{4}$$

From (3) and (4), we have the following formula:

$$Decide_H = (B_H - C_H \geq TH_{min}) \tag{5}$$

where TH_{min} is a pre-defined threshold parameter which is essentially application-dependent and on which the degree of load-balancing depends and $Decide_H$ is a boolean variable. Every member α of ψ for which $Decide_H$ returns 'TRUE' is put into a temporary list data structure which we shall designate as *'temp'*. The data structure of 'temp' is essentially a list structure where for each 'hot' data region H, the corresponding destination candidates (those members of ψ for which $Decide_H$ had returned 'TRUE') for H are stored in a linked list. Using the 'temp' data structure, the overloaded cluster leader uses a function **Select_dest_from_temp( )** which selects (as the destination cluster) the least loaded member in 'temp' (corresponding to each H) for each H. The load-balancing algorithm executed by an overloaded source cluster leader is depicted in Figure 1, while the load-balancing algorithm executed by a potential destination cluster leader is presented in Figure 2.

Observe that in contrast with existing works in traditional environments, our strategy does *not* use the value of normalized load difference when deciding upon the amount of data to replicate. This is because in our scenario, the increase in load at α owing to spatial join queries on H is negligible (even in case of spatial select conditions) as compared to the decrease in load for C_i especially since the join has already been computed. Moreover, note that replication is initiated from C_i to α whenever the normalized load difference between C_i and α exceeds a given threshold, irrespective of whether C_i is *really* overloaded or not. Even if C_i is *not* overloaded, we believe it is still reasonable to replicate at α since bringing the data closer to the cluster from where the data are being frequently queried implies a reduction in network overheads (as well as response times) for future spatial joins on the same data.

```
Algorithm LB_OverloadedSource( )
Create a set ψ comprising cluster leaders that issued more than η queries for any of its regions
if (ψ is an empty set) {
  exit
} else {
  for each element α in set ψ {
      Send message to α and asking α's disk space, load and neighbours' list
      Receive reply from α
      if  ( ( α's disk space is NOT sufficient ) OR ( Δ ≤ LOAD_THRESHOLD ) ) {
      /* Δ is the normalized load difference between itself and α */
          Delete α from set ψ and Add members of List_Neighbours to ψ
          for each member NG of List_Neighbours {
              Send message to NG asking NG's disk space availability and current load
              Receive reply from NG
              if ( ( NG's disk space is NOT sufficient ) OR ( Δ ≤ LOAD_THRESHOLD ) ) {
                Delete NG from ψ
              }
          }
      }
  }
  for each element α in ψ {
      for each data region H queried by α {
          if ( Decide_H = = TRUE ) {
            Put α into a temporary list designated as 'temp'
          }
      }
  }
  Select_dest_from_temp( )
}
end
```

Fig. 1. Load-balancing Algorithm executed by an overloaded source cluster leader

6 Performance Study

This section reports the performance evaluation of our proposed inter-cluster load-balancing technique via replication of result tuples of spatial join queries. Note that we consider performance issues associated *only* with inter-cluster load-balancing since a significant body of research work pertaining to efficient intra-cluster load-balancing algorithms already exists. Hence, for our experiments, we use a cluster size of 1. The machine used for the experiments had processing capacity of 1.7 GHz (Pentium-4), main memory of 768 Mbytes and disk space of 40GB. We ran the experiments under the Redhat Linux (version 7.3) operating system using LAM-MPI (version 7.00) for message-passing. In order to model inter-cluster communication in a wide area network environment, we assigned transfer rates for communication between cluster leaders randomly in the range of 0.8 Megabit/second to 1.2 Megabit/second. We used a maximum of 3 neighbouring cluster leaders corresponding to each cluster leader. The number

Algorithm LB_PotentialDestination( )
Receive message from overloaded source cluster leader SRC
/ The message contains disk space requirement of* SRC **/*
Send a Broadcast message to all the nodes in its cluster asking each node for its current load and disk space
Receive replies concerning current load and disk space of each node
Nodes with sufficient available disk space and load below a pre-defined threshold λ are put into a set $Candidate$
if ($Candidate$ is an empty set) {
  Send message to SRC stating that its disk space is insufficient and informing SRC about its list of neighbours
} else {
    Send message to SRC informing SRC about its sufficient disk space, its current load and its list of neighbours
}
Receive reply from SRC
if (SRC has selected it as the destination cluster) {
  Send a Broadcast message to the nodes in $Candidate$ for their current load status
  Receive the corresponding replies and select the least loaded node MIN from $Candidate$
  Send a message to SRC to replicate the data at MIN
}
end

Fig. 2. Load-balancing algorithm executed by each potential destination cluster leader

of clusters simulated in our experiments was 24. The interarrival time between queries arriving at a cluster was fixed at 10 milliseconds and the value of TH_{min} was set to 5 seconds. We have used two *real-life* datasets [3] for our experiments. The first dataset is the set of roads in Germany, while the second one is a dataset of railway lines in Germany. The first dataset comprises MBRs of 30,674 streets of Germany, while the second one consists of MBRs of 36,334 railroad lines in Germany. We had enlarged each of these datasets by translating and mapping the data for the purpose of our experiments. For our experiments, each of the clusters had more than 200000 rectangles for each of the relations. We used two R-trees at each cluster, one for each dataset. We assumed that one R-tree node fits in a disk page (page size = 4096 bytes). Hence, R-tree node capacity is the same as page size in our case. The height of each of the R-trees was 3 and the fan-out was 64. We generated queries for each cluster by using a spatial select (window query) condition in conjunction with the spatial join. Note that this is in consonance with real-world scenarios where spatial joins may be quite often accompanied by certain select conditions. The selectivity of each spatial join query was fixed at 40%. Assuming n queries for a particular cluster C_i, let us designate the queries as $Q_1, Q_2, ...Q_n$. We generated the n queries for C_i such that the queries had *at least* 75% overlap with each other. This overlap was generated by shifting the respective spatial select query windows in such a manner that each query had $x\%$ (where $x \geq 75\%$) overlap with the other queries.

For performing the spatial join operation at each cluster, we use an existing approach where the data from the smaller fragment is extracted and used to probe the index structure corresponding to the larger fragment. For the sake of convenience, we shall refer to our proposed technique as LBREP (Load-balancing via replication). Since *no* work on load-balanced processing of remote spatial joins in GRIDs exists, we shall compare the performance of LBREP with a technique which performs spatial join without load-balancing. We designate this reference technique as NOLB (No load-balancing). For all our experiments, we had run the system for an initial period of time to obtain access statistics information and once the system had reached a stable state (after the replication of result tuples have been performed), we noted down the results. We only present results associated with the stable state of the system. The replications that have

C_{Source}	$C_{Destination}$
1	24, 15
2	23,17
3	22, 14
4	21, 18
6	20, 15
8	19, 14
9	18, 10
11	17,15
12	16, 14
13	14, 10

(a) Table indicating replication

N	C_E	$C_Q(f)$
16	1	24(9),15(7)
12	2	23(6), 17(6)
12	3	22(6), 14(6)
4	4	21(3), 18(1)
4	6	20(3),15(1)
4	8	19(3), 14(1)
4	9	18(3), 10(1)
4	11	17(3),15(1)
4	12	16(3), 14(1)
4	13	14(3), 10(1)

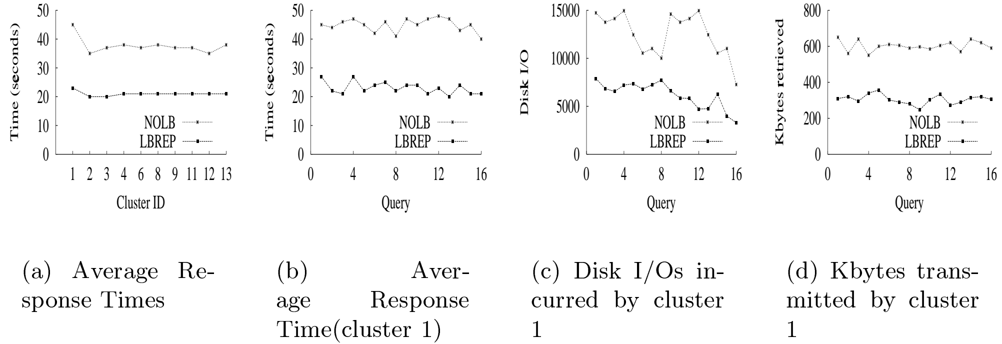
(b) $QD1$

N	C_E	$C_Q(f)$
16	1	15(10), 24(6)
12	2	17(4), 23(8)
12	3	14(3), 22(9)
4	4	18(3), 21(1)
4	6	15(3), 20(1)
4	8	14(3), 19(1)
4	9	10(3), 18(1)
4	11	15(3), 17(1)
4	12	14(3), 16(1)
4	13	10(3), 14(1)

(c) $QD2$

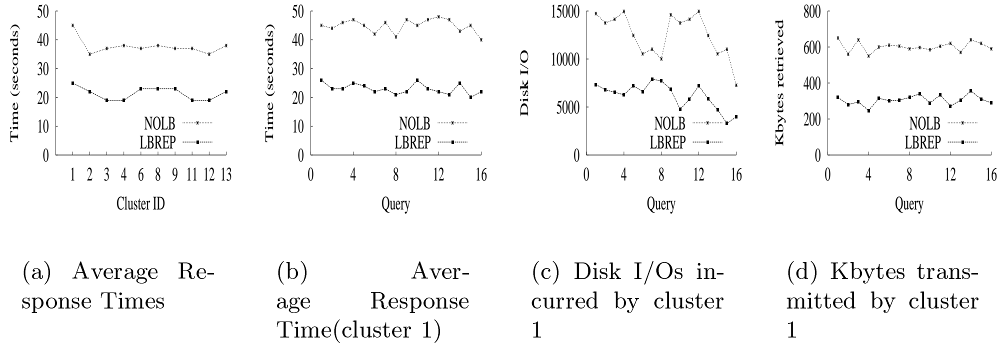
Fig. 3. Replication table and $QD1$ and $QD2$ for a 24-cluster GRID

already been performed (based on access statistics information) prior to the system reaching stable state are depicted in Figure 3(a). In Figure 3(a), C_{Source} represents the IDs of the source cluster whose data (spatial join result tuples) have been replicated, while $C_{Destination}$ stands for the IDs of the destination clusters where C_{Source}'s data has been replicated. For example, the first row of the table indicates that a portion of cluster 1's data has been replicated at clusters 24 and 15. Similarly, a part of cluster 2's data has been replicated at clusters 23 and 17 and so on. Note that the portions of cluster 1's data that have been replicated at clusters 24 and 15 need *not* necessarily be the same, even though overlap is possible between the replicated data of cluster 1 at cluster 24 and cluster 15. This is because the replication performed was based on previous access statistics, thereby implying that the replicated data at different clusters depends upon the queries that these clusters had issued during the past. Now we shall evaluate the relative performance of LBREP and NOLB by using different query distributions. Even though we had used several query distributions to test



(a) Average Response Times (b) Average Response Time(cluster 1) (c) Disk I/Os incurred by cluster 1 (d) Kbytes transmitted by cluster 1

Fig. 4. Results on $QD1$ for a 24-cluster GRID



(a) Average Response Times (b) Average Response Time(cluster 1) (c) Disk I/Os incurred by cluster 1 (d) Kbytes transmitted by cluster 1

Fig. 5. Results on $QD2$ for a 24-cluster GRID

the robustness of LBREP, in the interest of space, here we present only two such distributions. For the sake of convenience, we shall refer to these query distributions as $QD1$ and $QD2$ respectively. Figures 3b and 3c summarize $QD1$ and $QD2$. In Figure 3b, N denotes the number of queries, C_E indicates the ID of the cluster which processed the queries, C_Q represents the IDs of the clusters which issued those queries and f stands for the number of queries issued by a cluster. Note that the sequence of the queries arriving at each cluster is also specified by Figure 3b. For example, the first row of the table in Figure 3b indicates that 16 queries (let us designate them as $Q1$ to $Q16$) were processed by cluster 1. $Q1$ to $Q9$ were issued by cluster 24, $Q10$ to $Q16$ were issued by cluster 15. In contrast, the first row in Figure 3c indicates that $Q1$ to $Q10$ were issued by cluster 15, while cluster 24 issued $Q11$ to $Q16$. Owing to space constraints, we are *not* able to present the detailed results concerning *all* the queries in the system. Note that the selectivity of each spatial join query in case of both $QD1$ and $QD2$ was fixed at 40%. In all our experiments, cluster 1 is the most overloaded (hot) cluster

and also it was the last cluster in the GRID to complete processing. Hence, we shall examine details concerning the processing of queries that were directed to cluster 1.

Figures 4 and 5 depict the results corresponding to $QD1$ and $QD2$ respectively. Figure 4a indicates the average response times of *all* the queries directed to each cluster. The results demonstrate that LBREP is indeed able to decrease the average response times for each of the clusters significantly, especially decreasing the average response time of cluster 1 by upto 48%. The reduction in average response times occurs because of the reduction in disk I/O overhead at the query executing clusters as well as the reduction in communication overhead arising from transmission of result tuples to the clusters which issued the respective queries. To put things into perspective, we take a closer look at the processing of the 16 queries that were directed to cluster 1. Figure 4b depicts the individual response times of each of the 16 queries that were directed to cluster 1 for $QD1$, while Figure 4c shows the corresponding disk I/Os incurred for each query at cluster 1 for the same experiment. Figure 4d indicates the number of KBytes for each query that cluster 1 had to transmit to the cluster which had issued the query.

Observe that Figure 4b indicates that for *all* the queries directed to cluster 1, LBREP's performance is superior to that of NOLB in terms of response times. Such reductions occur because part of the results of the spatial join have already been replicated at clusters which issued these queries (clusters 24 and 15 in this case). The implication is that cluster 1 did *not* need to process a significant part of each of these queries, thereby resulting in reduction of disk I/O cost incurred by cluster 1. Moreover, since clusters 24 and 15 already had a part of the results associated with the queries that they issued, the number of result tuples that cluster 1 had to transmit to such clusters was also reduced, thereby reducing the communication overhead. Detailed investigation of the experimental results revealed that the reduction in disk I/O cost varied between 45% to 54%, while reduction in the total size of result tuples transmitted to the querying clusters varied between 46% to 52%. However, note that the price LBREP pays for improvements in response time is additional disk space usage since replication causes redundant usage of disk space. We believe that the overhead of additional disk space usage is justifiable because of the significant improvement in response times of spatial joins that LBREP provides.

The explanations for Figure 4 also hold good for the results in Figure 5. Observe that the performance of NOLB remains same in case of Figures 4 and 5 because in case of NOLB, no data has been replicated at the querying clusters, thereby implying that every query is completely processed at the query executing cluster and then the results are sent back to the querying clusters. We also find that the results in Figures 4 and 5 differ to some extent for LBREP. This is because the portions of cluster 1's data replicated at clusters 24 and 15 were *not* exactly the same, even though there was overlap between those portions.

7 Conclusion

Huge amounts of available spatial data worldwide and the prevalence of spatial applications, coupled with the emergence of GRID computing, provides a strong motivation for designing a spatial GRID. Skewed user access patterns may cause severe load imbalance in the system, thereby degrading system performance significantly. Our proposal has specifically focussed on speeding up remote spatial joins in this environment via a novel dynamic load-balancing strategy which deploys online replication. In the near future, we plan to address issues concerning dynamic data. Incidentally, for dynamic data, query results may change, thereby requiring updates to be propagated to the clusters containing the old replicated result tuples. Moreover, we shall investigate scalability issues concerning larger number of clusters. Additionally, we also plan to examine the use of other spatial index structures for performing spatial joins in GRIDs.

Acknowledgements: We wish to express our sincere thanks to the JSPS (Japanese Society for the Promotion of Science) for supporting this work.

References

1. T. Brinkhoff, H.P. Kriegel, and B. Seeger. Efficient processing of spatial joins using R-trees. *Proc. ACM SIGMOD*, pages 237–246, 1993.
2. European DataGRID. http://eu-datagrid.web.cern.ch/eu-datagrid/.
3. Datasets. http://dias.cti.gr/∼ytheod/research/datasets/spatial.html.
4. I. Foster and C. Kesselman. The GRID: Blueprint for a new computing infrastructure. *Morgan-Kaufmann*, 1999.
5. V. Gaede and O. Gunther. Multidimensional access methods. *ACM Computing Surveys*, 30(2):170–231, 1998.
6. Earth Systems GRID. http://www.earthsystemgrid.org/.
7. O. Gunther. Efficient computation of spatial joins. *Proc. ICDE*, pages 50–59, 1993.
8. A. Guttman. R-trees: A dynamic index structure for spatial searching. *Proc. ACM SIGMOD*, pages 47–57, 1984.
9. Y.C. Hu and D.B. Johnson. Caching strategies in on-demand routing protocols for wireless Ad Hoc networks. *Proc. MOBICOM*, pages 231–242, 2000.
10. G. Luo, J. F. Naughton, and C. Ellmann. A non-blocking parallel spatial join algorithm. *Proc. ICDE*, 2002.
11. A. Mondal, K. Goda, and M. Kitsuregawa. Effective load-balancing via migration and replication in spatial GRIDs. *Proc. DEXA*, 2003.
12. J. Patel and D. DeWitt. Partition based spatial-merge join. *Proc. ACM SIGMOD*, pages 259–270, 1996.
13. GriPhyN Project. http://www.griphyn.org/index.php.
14. Q. Ren and M.H. Dunham. Using semantic caching to manage location dependent data in mobile computing. *Proc. MOBICOM*, pages 210–221, 2000.
15. P. Rigaux, M. Scholl, and A. Voisard. Spatial databases with application to GIS. *Morgan Kaufmann ISBN 1-55860-588-6*, 2001.
16. P. Scheuermann, G. Weikum, and P. Zabback. Disk cooling in parallel disk systems. *IEEE Bulletin of the Technical Committee on Data Engineering*, 17(3):29–40, 1994.