DiST: Fully Decentralized Indexing for Querying Distributed Multidimensional Datasets*

Beomseok Nam and Alan Sussman
UMIACS and Dept. of Computer Science
University of Maryland
College Park, MD 20742
{bsnam, als}@cs.umd.edu

Abstract

Grid computing and Peer-to-peer (P2P) systems are emerging as new paradigms for managing large scale distributed resources across wide area networks. While Grid computing focuses on managing heterogeneous resources and relies on centralized managers for resource and data discovery, P2P systems target scalable, decentralized methods for publishing and searching for data. In large distributed systems, a centralized resource manager is a potential performance bottleneck and decentralization can help avoid this bottleneck, as is done in P2P systems. However, the query functionality provided by most existing P2P systems is very rudimentary, and is not directly applicable to Grid resource management. In this paper, we propose a fully decentralized multidimensional indexing structure, called DiST*, that operates in a fully distributed environment with no centralized control. In DiST, each data server only acquires information about data on other servers from executing and routing queries. We describe the DiST algorithms for maintaining the decentralized network of data servers, including adding and deleting servers, the query routing algorithm, and failure recovery algorithms. We also evaluate the performance of the decentralized scheme against a more structured hierarchical indexing scheme that we have previously shown to perform well in distributed Grid environments.*

1 Introduction

Grid computing is emerging as a new paradigm for managing large scale distributed resources in wide area networked computing environments [4]. A Grid is a distributed computing environment that may cross multiple administrative domains. There are many issues that must be addressed to provide a complete Grid computing infrastructure. One issue is how to locate distributed resources efficiently. A centralized resource discovery indexing service, such as the MCAT (metadata catalog) in the Storage Resource Broker [2], can be a performance bottleneck in large scale systems [2, 16].

In this paper, we propose a fully decentralized multidimensional indexing structure, called *DiST* (Distributed Search Tree), that operates in a distributed environment with no centralized control. We have designed a decentralized indexing structure to solve the scalability and reliability problems of a centralized indexing method. Decentralization is a main goal of Peer-to-Peer (P2P) overlay networks, which are attracting a lot of attention in the distributed systems community. In P2P systems, research is being conducted on efficiently searching for data objects in a dynamic environment. One of the reasons why P2P systems have been successful is that they support distributed data repositories robustly. That property of P2P systems has much in common with the goals of Grid computing, in particular data grids, but Grid computing is not limited to supporting data sharing. In the near future we expect that the Grid will need more efficient data discovery mechanisms, preferably not based on centralized indexing schemes as in P2P systems. Recently various P2P overlay networks have been proposed including Chord and CAN [3, 5, 20]. However, supporting *range* queries in P2P systems is still an open problem, especially for multidimensional data.

Multidimensional range queries are an important class of problems in Grid computing as well. For instance, consider a set of large-scale distributed machines, located all over the world. In such a system, users may want to issue a request to find machines with a given set of constraints, such as a machine with at least 1GB of main memory and a network delay to a particular host of less than 1 second. Another example of range queries in Grid application are to datasets with an underlying multidimensional attribute space. For those requests, a user may issue a query to retrieve a subset of some specific

*This research was supported by the National Science Foundation under Grant #EIA-0121161, and NASA under Grant #NAG-512652.

1-4244-0054-6/06/$20.00 ©2006 IEEE

datasets that may be distributed across multiple sites (e.g., NASA satellite sensor datasets over a range of latitude, longitude, and time). In order to handle such range queries efficiently in a Grid environment, a spatial indexing scheme is needed that is both more scalable and more robust than a centralized indexing scheme. Replication can be used to reduce access latency, improve data locality, and increase robustness and scalability. In previous work [16], we have evaluated the performance of two different distributed multidimensional indexing schemes that employ replication and hierarchy in different combinations. One method has a single centralized index that can be replicated as needed, while the other employs a two-level hierarchy with a local index on every server and a centralized global index (that can also be replicated) that aggregates the information in the local indexes. Both schemes reduce the overhead on the indexing servers by distributing the workload for searching the index. However, replication strategies cause high overhead for updating an index (i.e. adding or deleting data objects), especially when strong consistency across replicas is needed. In addition, replication requires a sophisticated management mechanism for creating, deleting, and locating replicas. Since the number of replicas has a significant effect on the performance of both index look-up and update, for the best performance the number and placement of replicas should be determined dynamically from workload characteristics.

In this paper, we focus on a fully decentralized indexing scheme that does not require such an adaptive management strategy for scientific applications that navigate through large distributed multidimensional datasets. A fully decentralized indexing structure promises to scale to thousands of servers without a central bottleneck. Every server in the DiST decentralized indexing scheme contains a local index for data stored on that server, as in the two-level hierarchical scheme discussed earlier, but also contains partial global index information. DiST does not require a dedicated server to store and search the global index, as does the two-level hierarchical scheme. Based on incomplete global information, any server in the DiST system can forward a query to other servers as needed to satisfy a query. The DiST query routing algorithm guarantees that queries will be delivered to the correct destination servers, potentially routing through intermediate servers, unless the destination servers have left the system or failed. In addition to evaluating the basic performance of the DiST algorithms, we also describe how to reduce the number of network hops needed for query routing to improve performance, and also how to recover from server failures or departures to make DiST more robust.

The rest of the paper is organized as follows. In Section 2 we discuss other research related to distributed indexing. In Section 3 we introduce our fully decentralized multidimensional indexing scheme, and show experimental results in Section 4. In Section 5 we conclude and discuss future work.

2 Related Work

Since Kamel and Faloutsos proposed the first parallel R-trees (Multiplexed R-trees) [9], for a machine with a single CPU and multiple disks, several parallel multidimensional indexing structures, such as Master R-trees [10] and Master Client R-trees [18], have been developed to extend Multiplexed R-trees. The Master R-tree was designed for a shared nothing environment (i.e. a distributed memory parallel machine, or cluster) [10]. A single server maintains all the internal nodes of the R-tree except the leaf level data nodes, which are declustered across the other servers. The Master Client R-tree is a two-level distributed R-tree that has a single master index on a master server and local client indexes on the other servers. The Master Client R-tree is similar to the Master R-tree in the sense that it declusters leaf level nodes across data servers. However each data server creates its own local index using the leaf level nodes that are assigned to it.

Master R-trees and Master Client R-trees require at least one dedicated server to maintain global status information about the distributed index, which is a potential bottleneck. To avoid centralized accesses, several fully decentralized indexing structures have been proposed, and are collectively called SDDS (Scalable Distributed Data Structures). These include LH* [13], which generalizes Linear Hashing to distributed systems, and distributed random trees (DRT) [11]. Our decentralized indexing scheme is similar to DRT in that we are using KD-trees as the basic indexing data structure and that each server maintains some part of the overall global KD-tree, which we will describe in detail later.

In fully distributed systems (i.e., pure P2P systems) peers are directly addressed, typically via a hashing scheme, to return the data objects they contain. The Chord [19] and CAN [17] systems implement distributed hash tables to provide efficient lookup of a given key value. These systems assign a unique key to each data object (i.e., a file) and forward queries to specific servers based on a hash function. Although these systems guarantee locating a data object within a bounded number of network hops, they require tight control over data placement and the topology of the overlay network that they create. In a Grid environment, arbitrary data placement is not always feasible due to both organizational and technical issues (e.g., the size of the datasets). In broadcast-based P2P systems (also called unstructured P2P systems) such as Gnutella [6], message flooding is employed to forward queries, since each peer does not have data placement information. Message flooding does

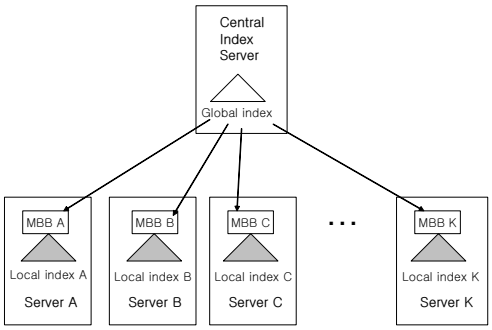


Figure 1. *Centralized Two Level Indexing*

not guarantee accurate query results, thus is not feasible for typical requests in a Grid environment that require accurate query results.

The recently developed P-tree, a fully decentralized B+-tree, enables one dimensional range queries in a pure P2P network [3]. The P-tree assumes that a peer stores only a single data object, thus in order to store more than one data object each peer needs to be mapped to by multiple *virtual peers*. The routing algorithm in a P-tree is based on virtual peers, thus a peer may be accessed multiple times while routing to virtual peers. For this reason, the P-tree is not suitable for a system that stores many data objects. Also a one dimensional range query is not adequate for scientific data analysis applications that access and process multidimensional data.

SkipIndex is a distributed indexing overlay network that uses the Skip Graph data structure [20]. Since Skip Graph only works for one dimensional data, SkipIndex first builds a KD-tree for multidimensional data, then builds the 1D Skip Graph on the leaves of the KD-tree. Although DiST and SkipIndex use the same KD-tree style partitioning strategy, the major difference between SkipIndex and DiST is their scalability. Since SkipIndex targets large scale P2P overlay networks, it limits the number of remote peers that can be directly accessed by a given peer, as for most P2P overlay networks. However, this may cause high traffic for peers close to the root in the KD-tree hierarchy. If the number of peers directly accessible by a single peer is limited, the number of routing hops generally increases. Since our main concern is designing a decentralized indexing scheme rather than a scalable P2P overlay network, we do not limit the number of peers directly accessible to a single peer to give the best range query performance.

3 Fully Decentralized Indexing

Scientific instruments can produce hundred of gigabytes of spatio-temporal data daily, consisting of billions of individual data elements. Storing each data element into a multidimensional indexing data structure is impractical, because the size of the index would become very large, and the performance of queries could be poor due to the size of the index. Instead, we can build a bounding box in space and time for a part of the dataset with data elements having nearby spatio-temporal coordinates (a *chunk*). That allows storing only the bounding boxes into the index to reduce its size and make index searches faster [14].

When datasets are distributed across multiple servers, it is also important to distribute the index itself to get the benefits of parallelism. Figure 1 shows a two level hierarchical indexing scheme, as an example of a distributed index. In two level hierarchical indexing, each data server has an index for data stored on that server (a *local index*). To search the index, a *global index* is used to determine which local index(es) must be accessed. The global index stores the Minimum Bounding Boxes (MBBs) of the root nodes of the local indexes, each of which is only big enough to span all the bounding boxes of the data chunks in a local server. When a range query is submitted to the server owning the global index, the server compares the range with those MBBs and returns the list of servers that have overlapping MBBs with the given range. Since the global index does not contain any information about the actual data stored in the servers, it is possible for the global index server to return local servers for a query when, in fact, those local servers do not have any data that overlaps the query range. More details about the two level indexing scheme can be found in [16].

One way of decentralizing the index is to replicate the global index across all the servers. However, full replication requires broadcast messages when the index needs to be updated. For a relatively small number of servers, full replication can be a good choice, especially when the frequency of index search operations is much greater than the frequency of index updates. However, as the number of servers increases, index updates become a serious performance problem. In order to make a distributed index scalable, several challenges must be considered [3, 5]:

- **Distribution**: The index needs to be partitioned across a large number of servers, in order to avoid potential bottlenecks and ensure load balance.
- **Dynamism**: Nodes may join, leave or fail at any time. Therefore we need an efficient recovery mechanism to ensure correctness. And the number of servers involved for any index update should be minimized.
- **Correctness**: A search for data object o succeeds if o is stored in a server that has not failed. If some servers in the query routing path fail, a recovery mechanism should find another routing path to the destination server.
- **Efficiency**: The number of network hops for a query should be at most logarithmic. Also, the number

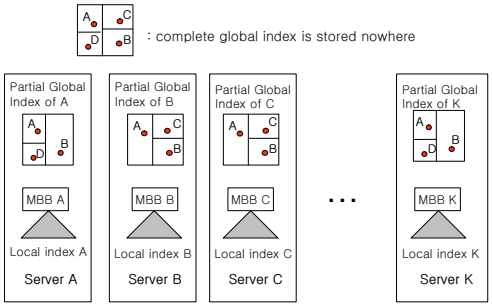


Figure 2. *Decentralized DiST Indexing*

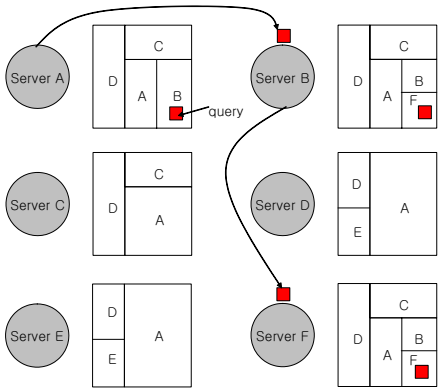


Figure 3. *Query Routing in DiST*

of servers involved for a query should be at most logarithmic, since broadcasting is not efficient.

We describe how our decentralized indexing scheme, *DiST*, satisfies these properties.

Distribution: DiST is a decentralized version of the two level indexing scheme. Each server has a local index for the data stored on that server, and the global index is distributed across all the servers, as shown in Figure 2. DiST obviously satisfies the *distribution* property. The global index of DiST partitions the complete multi-dimensional attribute space (i.e. it is a *space partitioning* spatial index), as is done for KD-trees, and each leaf node in the tree corresponds to an MBB of a local index. *Data partitioning* spatial indexing schemes, such as R-trees [7] and its variants, are not suitable for a fully decentralized environment, because their partitioning strategies are very dynamic and allow overlapping partitions between sibling nodes in the tree. Overlapping partitions and frequent updates may result in a large number of partition update messages for a decentralized index.

When a server joins the system, it becomes an owner of a specific partition in the multi-dimensional space, which corresponds to a leaf node in a KD-tree. The partition is determined by the KD-tree insertion algorithm, which assigns ownership of partitions to servers. Each server that joins the system already has its own local index, and the MBBs of the local indexes are stored in the decentralized, partitioned global index. When a new server joins the system and inserts the MBB of its local index into the global index, that MBB will fall inside an area in the multi-dimensional space that is owned by an existing server. The MBB will map into exactly one partition, owned by one existing server, since we convert the MBB into a single high dimensional point for insertion into the tree. More details on that conversion will be provided shortly. The insertion algorithm has the previous owner divide its current space into two parts, and assigns one of the newly split partitions to the new server. However, the previous owner does not need to forward that split update to all other servers in the system. The reason is that the query routing algorithm we describe later, in Section 3.1, can deal with stale index information.

3.1 Correctness: Range Query Routing

The DiST query routing algorithm guarantees *correctness*, because any range query will eventually be forwarded to the actual destination owner server that has the requested data, although the query can be submitted to any server, and none of the servers in the system has a complete and up-to-date global index. Therefore we allow inconsistent global information across servers, so long as we can guarantee correct search results. So whenever a server joins the system, only one other server must update its global index to ensure correct query results. Minimizing information propagation is one reason why we chose a static space partitioning method, KD-trees, as the global index spatial data structure instead of a data partitioning method.

Figure 3 shows how DiST guarantees correct range query results. When a query is submitted to server A, the server searches its global index and forwards the query to server B, since the global index of server A indicates that the query range falls inside the region owned by server B. However that region turns out to have been split previously, when another server, F, joined the system. Although server A does not have complete, up-to-date, global index partitioning information, the query can still be forwarded to the right server (server F in the example), since server B can forward the query to server F. In this way, the query can be delivered to the right server(s) with a small number of network messages.

The DiST query routing algorithm guarantees correct search results, because the split information that a server has in its global index (from other servers joining the system) is *always* correct. If split positions are allowed to change dynamically, the change must be propagated to *all* the servers in the system. Otherwise the DiST query routing algorithm will not be able to find the correct destination for the query. Static space partitioning methods such as KD-trees satisfy this requirement, but KD-trees

Algorithm 1
Range Query Routing Algorithm

procedure
RangeQuery(*QueryBBX*, *QueryID*, *QueryHistory*, *Sender*)

```
1: OwnerID := GetOwner(RootBBX)
2: if QueryID is already processed then
3:    QueryResultForward(Sender, NULL)
4: else
5:    OwnerIDList := GetOwnerList(QueryBBX)
6:    QueryHistory+ = OwnerIDList
7:    for all OwnerID in OwnerIDList do
8:       if OwnerID == me then
9:          Result += LocalSearch(QueryBBX)
10:      else if OwnerID is not in QueryHistory then
11:         QueryRequestForward(OwnerID,
            QueryBBX, QueryID, QueryHistory, me)
12:         Forwarded := TRUE
13:      end if
14:   end for
15:   Result += WaitQueryResults(QueryID, OwnerIDList)
16:   QueryResultForward(Sender, Result)
17: end if
```

end procedure

Algorithm 2
Node Join Algorithm

procedure
NodeJoin(*RootBBX*, *NewNode*)

```
1: OwnerID := GetOwner(RootBBX)
2: if OwnerID == me then
3:    Insert(GlobalIndex, RootBBX, NewNode)
4:    GlobalIndexCopyForward(NewNode, GlobalIndex)
5: else
6:    JoinRequestForward(OwnerID, RootBBX, NewNode)
7: end if
```

end procedure

only work for point (not rectangular) data. If the data objects are not points (in our case, the MBB of the local indexes), then space partitioning methods that require disjoint sub-partitions are unsuitable. Therefore, we need to design a new static space partitioning method for rectangular data. Otherwise, we could convert a rectangular bounding box into a higher dimensional point. Henrich et al. studied a transformation method that converts rectangular data into high dimensional point data [8] (i.e., a rectangle in 2D becomes a 4D point). More details about the transformation method and its optimization techniques can be found in [8, 15].

3.2 Dynamism: Decentralized Node Join

The server join algorithm is described in Algorithm 2, and Figure 4 shows an example. Whenever a new server joins the system, the server sends a join request to any existing server, and the recipient of the join request, call it R, searches its global index. If the bounding box of the new server falls inside the region owned by R, R splits the multi-dimensional space it owns and the new server becomes the owner of one of the new partitions. Otherwise, R forwards the join request to the server that R's global index says owns the sub-partition containing the bounding box of the new server. The join request routing algorithm is the same as the query routing algorithm described in Algorithm 2. As shown in Figure 4, if server C sends a join request to server A, server A searches its global index and forwards the request to server B, since the bounding box for server C is inside the region the index says is owned by server B. Server B also searches its global index and determines that the root bounding box of server C falls inside the space it owns, so B splits its space and forwards a copy of its global index to server C.

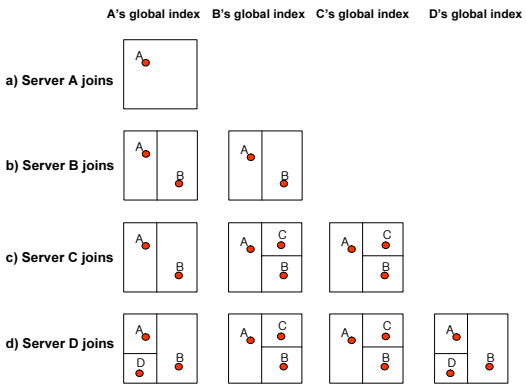


Figure 4. *Node Join in DiST*

Figure 4d shows the state of each global index after server D joins via contacting server A. Note that each global index may contain different global information. However, all servers always have split information that does not change, unless servers leave or fail. If a server leaves or fail, the global split information must be updated, and that will affect the *correctness* of the DiST algorithms. Hence we must handle servers leaving or failing in a robust way, to guarantee correct search results. Recovery from a server leaving or failing will be described in Section 3.3.

Since we convert the bounding box of the root of the local index into a higher dimensional point to insert it into the partitioned global index, the converted point can move within the high dimensional space when the root bounding box changes (i.e., an update to the local index on a server propagates to the root of the local tree). If the converted point does not cross the boundary of the partition that the point currently is in, no global partition update to other servers is necessary. In the case where the converted point does cross the boundary of its current partition, there are two steps required to update the global index. The first step is to delete the old partition from the global indexes in a few servers - the server itself and the neighbor servers. The second step is forwarding the join request to the server that owns the partition that the new point falls into, just as for a new server.

As we discussed earlier, one of the properties that decentralized indexing must satisfy is *dynamism*. When an

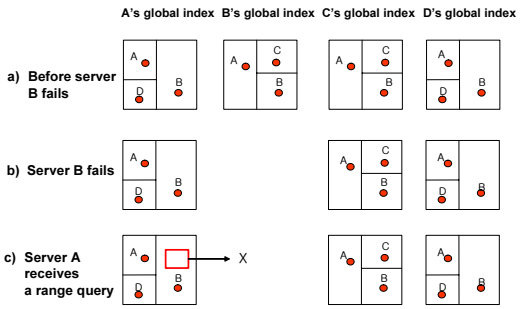


Figure 5. *Network Partition due to Node Failure*

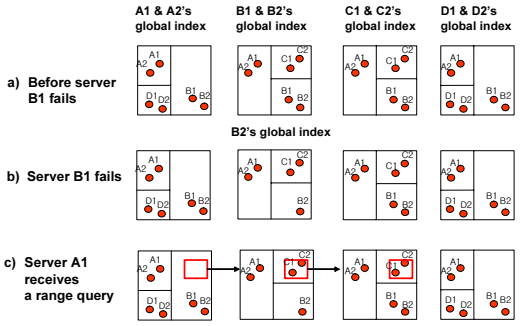


Figure 6. *Multiple servers in a single partition prevents network partition*

update of the global index is required, the number of servers involved in the update should be as few as possible. In DiST, the partition information for a server can be stored in any server in the system. This implies that many servers may need to receive global index update messages. However it is not efficient in a dynamic environment for each server to keep track of all the servers that must be updated when an update occurs. Thus, DiST updates the global index in a lazy manner. Some servers in DiST may have stale partition information due to the updates. But the routing algorithm of DiST will deliver all queries to the destination servers that are supposed to be forwarded to without affecting the correctness of index searches. In order to improve search performance, by making the global indexes converge to consistent states quickly, we describe a *piggyback* index update strategy in Section 3.4.

3.3 Dynamism: Decentralized Node Failure and Recovery

If a server leaves the system or fails, that can cause an unexpected network partition. Suppose there are four servers, each with its own global index, as shown in Figure 5. In the example, server B knows the partition owned by server C, but server A does not. If server B fails or leaves, a range query submitted to server A cannot be forwarded to server C, nor can correct query results be returned, because of the network partition.

The danger of this problem can be reduced by allowing multiple servers to own a single partition, as shown in Figure 6. If server *B1* fails and another server *A1* detects that *B1* is not responding, server *A1* can forward a query to another owner, in this example *B2*, for the same partition. Servers *B1* and *B2* will have different data and local indexes, but when a server forwards a query for the partition owned by *B1* and *B2*, it forwards to only one of them since both know the exact bounding boxes for other servers in the same partition.

If we restrict the number of servers for a partition, call that number K, to either 2 or 3, a server should only split its partition when a new server joins and more than 3 bounding boxes would end up in the same partition server. After the split, each partition will then have 2 server bounding boxes. If we want to increase the fault tolerance of the overall system, we can increase the number of servers that own a partition. Note that P2P systems based on distributed hash tables, such as CAN or Chord, cannot recover from multiple simultaneous neighbor failures [17, 19]. This problem also applies to the DiST recovery algorithm.

If all the servers in the same partition fail, we require an alternative recovery mechanism to the one just described. One (expensive) option is broadcasting or flooding recovery messages [6], which is inefficient.

Another recovery option is similar to the CAN mechanism in the sense that each server has to maintain a neighbor server list [17]. In DiST, remote servers may send queries directly to the failed server. After detecting a server failure, the remote server simply deletes the failed server from its global index. When a node is deleted from the KD-tree index, a sibling node or subtree takes over the partition for the deleted node. In DiST, this means that the sibling server(s) becomes a replacement server(s) and take over the partition for the failed server. When a server joins the systems and inserts its MBB into the global index, the server that ends up splitting its partition to give to the new server must forward a global index update message to its neighbors, in order to maintain correct neighbor information. If a failed node is detected during a global index search, the node is deleted as just described, and the search continues. This failure recovery algorithm works in a lazy manner, thus it does not cause high overhead and does not affect correctness. However the DiST recovery algorithm has the same problem as other decentralized indexing schemes, and cannot recover from multiple simultaneous server failures. If we combine the recovery algorithm with the partition sharing approach, DiST would be even more fault tolerant,

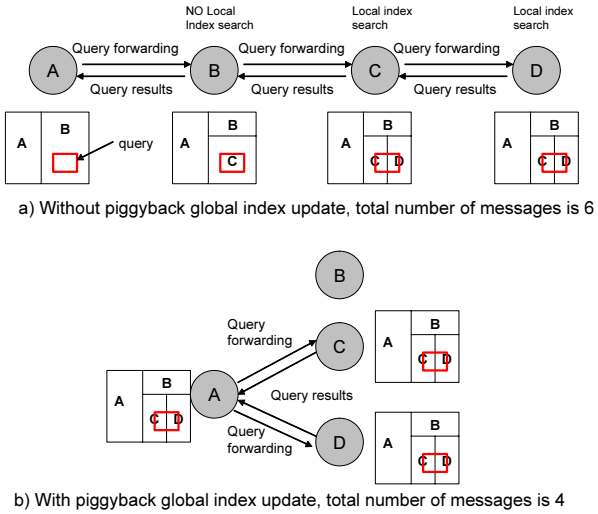


Figure 7. *A partial global index may cause additional messages for searches*

but recovery is still not always possible. We plan to investigate failure recovery algorithms more deeply in future work.

3.4 Efficiency: Piggyback Updates

Maintaining only a partial global index at each server may result in more network messages for search queries compared to fully propagating global index updates, as shown in Figure 7. In the example, server A must forward a query to server B, since A does not have partition information for servers C or D. If server A has partition information for servers C and D, the message from server A to server B is not needed, since the partition for server B does not overlap the given query range, as seen in the Figure 7(b).

When the global index is not a balanced tree, the partial global index may cause a long message chain. This effect not only increases the number of messages, but also increases the size of the messages because intermediate servers must collect and merge query results to return them back up the chain . If the global index is completely skewed, the number of messages in the worst case is N, where N is the total number of servers. Although this is a rare case, it violates the desirable *efficiency* property from Section 3.

To improve performance, we have adopted the *lazy update* technique from distributed random tree (DRT) algorithms [11]. Two incomplete global indexes can be merged as they are traversed in either breadth or depth first order. With tree merging, as a server obtains a global index that is close to complete, it is likely that the number of network hops needed for any range query search operation will be close to 1. Even if a global index on a server is not complete, the number of network hops will be logarithmic in most cases, if the global index is well balanced. DiST without piggyback updates satisfies the *efficiency* property most of the time, and the piggyback update scheme further improves performance, as will be shown in the experiments in Section 4.

Unlike P2P overlay networks, DiST does not limit the number of servers directly accessible by a single server, and piggyback index updates can increase the number of directly accessible servers. As a server maintains more information about other servers, the scalability of the indexing scheme is compromised. In order to make DiST work in global scale systems, we must design a mechanism that controls the number of servers that are directly accessible by a single server. However, since our target applications are not pure P2P applications, but scientific data analysis applications, we are more concerned with range query performance than scalability to very large numbers of servers. In most applications, range queries are much more frequent than update requests, thus piggyback index updates will make the partial global indexes become complete and consistent quickly. As a result of piggyback updates, the number of network hops for any single query will converge to 1 as in centralized two level indexing, and the traffic load for servers close to the root in the KD-trees will be reduced.

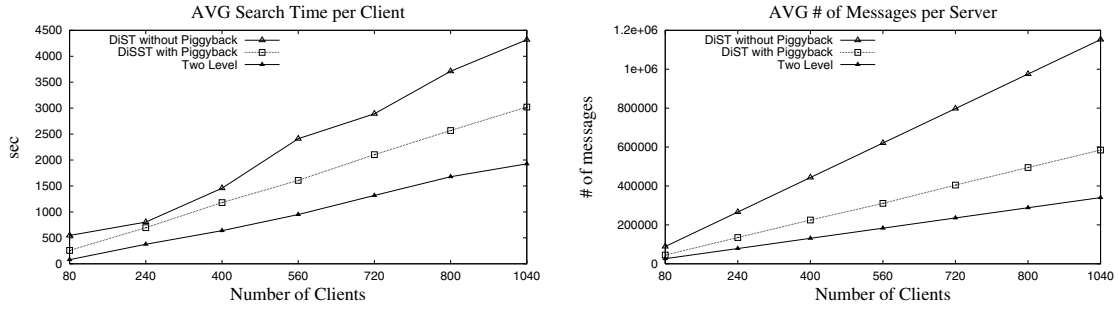
Piggyback updates are triggered when a server receives a query and detects that the query sender did not directly send the query to one or more servers that should receive the query. Query forwarding history is required to implement piggyback updates and to eliminate duplicate query processing. When a query is forwarded, each server has to attach information on what servers have already seen the query. Even with query forwarding history, a server can receive the same query multiple times due to query routing. Therefore we need to assign a unique query id to each query, which should increase monotonically. Using the query id, servers can detect duplicate queries.

4 Experiments

4.1 Experimental Environment

We have measured the performance of DiST on 41 server machines in a Linux cluster. Each of the 41 servers has a Pentium III 650 MHz processor and 768MB memory, and the servers are connected by channel-bonded Fast Ethernet (200Mb/sec per server). In order to emulate a larger configuration of more than 40 servers, for some of the experiments we also ran multiple index servers on a single machine. Intercommunication between index servers is done via LAM/MPI [12].

Three dimensional satellite image datasets were used to evaluate DiST. The satellites gather remotely sensed



(a) Response Time

(b) Communication Cost

Figure 8. *Search performance varying the number of clients*

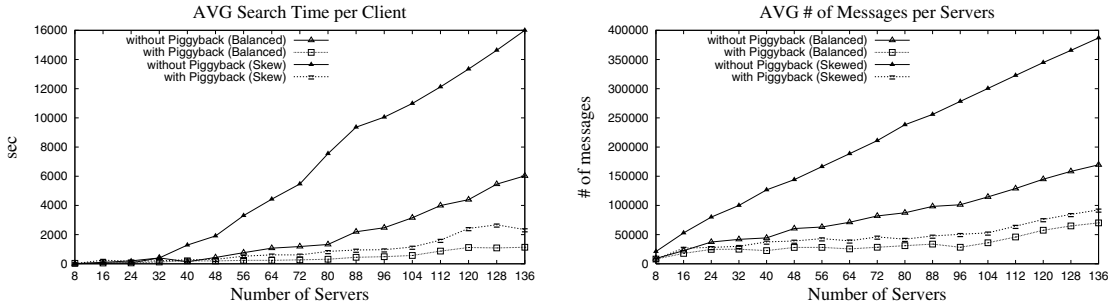
AVHRR (Advanced Very High Resolution Radiometer) GAC (Global Area Coverage) level 1B datasets, stored as a set of arrays. The datasets includes geo-location fields (latitude and longitude), time fields, and some additional metadata. As the satellite moves along a ground track over the earth, it records sensor and time values, with the geo-location metadata information then computed from satellite orbit models. Because the sensor swings across the ground track, the sensor and metadata values are stored as two dimensional arrays. We partitioned those 2D arrays into equal sized rectangular chunks, built three dimensional bounding boxes (latitude, longitude, and time) for each chunk, and stored the boxes into the local indexes in each server. For each chunk, the leaf node in the index contains the local server name, local file name, and the array offset within the file for the data for that chunk. We used SH-trees as the multidimensional data structure for local index on each server, since we have shown that SH-trees provide better performance than R-tree based indexing structures for chunked datasets [14].

The dataset used was collected over one month (January 1992), and has a total size of more than 30GB, with the volume of data for a single day about 1GB. The dataset was partitioned into 700,000 chunks. We assigned 5,000 chunks to each of 136 virtual servers. In order to create range queries, we modeled common query behaviors into the satellite datasets using a variation of the *Customer Behavior Model Graph (CBMG)* technique to match queries to a realistic workload, including hot spots in the data corresponding to areas of high interest [1].

4.2 Experimental Results

Figure 8 shows the search performance of the hierarchical two level index, with a centralized global index, vs. DiST for different numbers of clients. The total number of server machines that were used for DiST is 40 (the servers with the data), and we used an extra server as a dedicated global index server (41 total) for the two level indexing scheme. The clients are distributed evenly across the 40 server machines and each of them submits 2000 queries, waiting for one query to complete before issuing the next query. Since there is no front end in DiST, any client can connect to any server. For the two level indexing, a client can connect to any server as well, but all queries must be forwarded to the global index server. For the DiST measurements, each server joined the system in random order, so the global index is moderately balanced in this experiment. Figure 8(a) shows the average elapsed wall clock time per client for its 2000 queries, and Figure 8(b) shows the average number of network messages received at each server. In terms of both time and network messages, the search cost increases linearly as the number of clients increases from 80 to 1040. Figure 8 shows that two level indexing generates only about 1/4 the number of messages as DiST without the piggyback global index update method, and about half the number of messages as DiST with piggyback updates. This is because the maximum number of network messages for a single query for two level indexing is always 2 for the global index search and 2 for each local index search, which is optimal. The number of network messages eliminated by DiST piggyback updates is approximately half of the total number of network messages in our experiments. As more queries are submitted using DiST with the piggyback global index update, the number of network messages may decrease. However, the global index in DiST suffers from the dead space search problem, since we convert the bounding boxes into higher dimensional point data, but that is not a problem for the two level indexing that directly uses the bounding boxes.

The results from Figure 8(a) show that two level indexing is also faster than DiST. With 1040 clients, searching the two level index took about 2/3 the time to search with DiST and piggyback updates, on average, and less than half the time of DiST without piggyback updates,



(a) Response Time

(b) Communication Cost

Figure 9. *Search performance varying the number of servers*

on average. These times are consistent with the number of network messages shown in Figure 8(b). We doubt that it is possible to make DiST generate fewer network messages than does two level indexing, because of the nature of decentralized indexing, i.e. complete global information is not guaranteed to be available.

We also evaluated the scalability of DiST with respect to the number of servers. We increased the number of servers from 8 to 136. Since we ran this experiments on 40 machines, multiple servers ran in a single machine. For instance, with 120 servers we ran 3 servers on each machine. We compared two policies for DiST, with and without piggyback global index updates, and with two different global index structures, one a relatively balanced tree and the other an almost linear skewed tree structure. The global index in DiST is a binary KD-tree, which is not guaranteed to be balanced. We controlled the server join sequence in order to compare the search performance of balanced and skewed global indexes. When the global index for DiST is skewed, the number of network messages for a single range query is linear in the worst case, i.e. the total number of servers in the system. However with piggyback updates, the number of network messages decreases as more updates are performed, since each server will eventually obtain the entire global partitioning information, so can directly forward queries to the correct destination server(s).

Figure 9(b) shows that the number of messages increases as the number of servers increases. This is partly because the number of network messages increases with larger systems, and also because we ran more clients with more servers. When a new server joins, we also create a new client, which connects to the new server and submits 2000 range queries, as in the earlier experiments. The performance gap between the balanced global index and the skewed global index is quite substantial. A balanced global index generates only about 1/3 the number of network messages as with the skewed global index, without piggyback updates for the global index. With piggyback

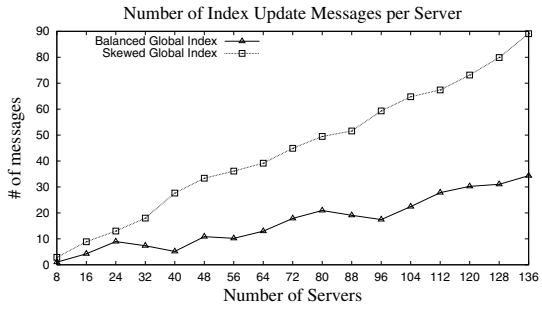


Figure 10. *Number of piggy-back global index update messages*

index updates, the message gap between the balanced global index and skewed global index decreases. However the balanced global index generates 3/4 the number of messages as for the skewed global index. If there are no server failures and no new servers join the system, eventually all the servers would have a complete global index. In such a case, no matter how much a global index is skewed and no matter what server receives a query, the number of messages for a query would be exactly 2.

Figure 10 shows the average number of piggyback global index update messages delivered to each server. When there are 136 servers in the system and the global index for those servers is skewed, 89 global index update messages are delivered to each server on average. Compared to the average total number of messages delivered (92,393) to each server, 89 additional messages is negligible, which means piggyback global index updates improve performance without much overhead. Since in the experiments each server has a single client and each client submits 2000 queries, the probability that a server receives a global index update message for any given query is about 4.5%. When the global index is balanced, the average number of global index updates per server is 34, thus the probability drops to 1.7%. If servers join or leave

the system frequently, the number of global index update messages would increase.

5 Conclusion and Future Work

In this paper we presented the design of a new decentralized multidimensional indexing scheme, DiST, that targets large distributed systems without centralized control. DiST accelerates multidimensional range query performance in distributed environments with no obvious performance bottlenecks, so that it shows comparable performance to a centralized two level indexing scheme, and provides improved scalability and reliability compared to centralized indexing.

There are still many interesting issues for decentralized indexing. We need to investigate more deeply the scalability of DiST and two level indexing, which we measured only with a relatively small number of data servers. In order to observe the scalability of DiST compared to two level indexing, we need to scale to at least hundreds of data servers, and we do not yet have access to that many server machines. Thus, we plan to do a simulation study in the near future. Also, we need to measure the performance effects of assigning multiple servers to the same partition, since the servers within the same partition must know each other's bounding boxes exactly, likely increasing the number of update messages required, but also decreasing the number of search messages due to the reduced search path length. Having multiple servers in the same partition has the same effect as reducing the number of servers in the system. Another direction of this work is to adapt other decentralized indexing schemes for P2P systems for the global index in a two level indexing scheme.

References

[1] H. Andrade, S. Aryangat, T. Kurc, J. Saltz, and A. Sussman. Efficient execution of multi-query data analysis batches using compiler optimization strategies. In *Proceedings of the 16th Workshop on Languages and Compilers for Parallel Computing*, pages 509–523, Oct. 2003.

[2] C. Baru, R. Moore, A. Rajasekar, and M. Wan. The SDSC Storage Resource Broker. In *Proceedings of CASCON'98 Conference*, Dec. 1998.

[3] A. Crainiceanu, P. Linga, J. Gehrke, and J. Shanmugasundaram. Querying peer-to-peer networks using P-trees. In *Proceedings of the 7th International Workshop on the Web and Databases*, 2004.

[4] K. Czajkowski, S. Fitzgerald, I. Foster, and C. Kesselman. Grid information services for distributed resource sharing. In *Proceedings of the 10th IEEE International Symposium on High Performance Distributed Computing (HPDC)*, Aug. 2001.

[5] P. Ganesan, B. Yang, and H. Garcia-Molina. One torus to rule them all: Multi-dimensional queries in p2p systems. In *WebDB '04: Proceedings of the 7th International Workshop on the Web and Databases*, pages 19–24. LNCS, 2004.

[6] Gnutella website. *http://www.gnutella.org*.

[7] A. Guttman. R-trees: A dynamic index structure for spatial searching. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD84)*, pages 47–57, 1984.

[8] A. Henrich, H.-W. Six, and P. Widmayer. The LSD tree: Spatial access to multidimensional point and non-point objects. In *Proceedings of the 15th International Conference on Very Large Data Bases (VLDB89)*, pages 45–53, 1989.

[9] I. Kamel and C. Faloutsos. Parallel R-trees. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD92)*, pages 195–204, 1992.

[10] N. Koudas, C. Faloutsos, and I. Kamel. Declustering spatial databases on a multi-computer architecture. In *Proceedings of the 5th International Conference on Extending Databases Technology (EDBT)*, 1996.

[11] B. Kroll and P. Widmayer. Distributing a search tree among a growing number of processors. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD94)*, pages 265–276, 1994.

[12] LAM/MPI website. *http://www.lam-mpi.org*.

[13] W. Litwin, M.-A. Neimat, and D. A. Schneider. *LH**: Linear hashing for distributed files. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD93)*, pages 327–336, 1993.

[14] B. Nam and A. Sussman. A comparative study of spatial indexing techniques for multidimensional scientific datasets. In *Proceedings of 16th International Conference on Scientific and Statistical Database Management (SSDBM)*, June 2004.

[15] B. Nam and A. Sussman. DiST: Fully decentralized indexing for querying distributed multidimensional datasets. Technical Report CS-TR-4720 and UMIACS-TR-2005-28, University of Maryland, 2005.

[16] B. Nam and A. Sussman. Spatial indexing of distributed multidimensional datasets. In *Proceedings of CCGrid2005: IEEE/ACM International Symposium on Cluster Computing and the Grid*, May 2005.

[17] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A scalable content addressable network. In *Proceedings of the 2001 ACM SIGCOMM Conference*, 2001.

[18] B. Schnitzer and S. T. Leutenegger. Master-Client R-Trees: A new parallel R-tree architecture. In *Proceedings of the 11th International Conference on Scientific and Statistical Database Management (SSDBM)*, pages 68–77, 1999.

[19] I. Stoica, R. Morris, D. Karger, F. Kaashoek, and H. Balakrishnan. Chord: A scalable Peer-To-Peer lookup service for internet applications. In *Proceedings of the 2001 ACM SIGCOMM Conference*, pages 149–160, 2001.

[20] C. Zhang, A. Krishnamurthy, and R. Y. Wang. SkipIndex: Towards a scalable peer-to-peer index service for high dimensional data. Technical Report TR-703-04, Princeton University, 2004.