Noname manuscript No.
(will be inserted by the editor)

Analyzing Design Choices for Distributed Multidimensional Indexing

Beomseok Nam · Alan Sussman

This is a preprint version of the manuscript published in Journal of Supercomputing, Volume 59, Issue 3, pp 1552-1576, Springer, 2012

Abstract Scientific datasets are often stored on distributed archival storage systems, because geographically distributed sensor devices store the datasets in their local machines and also because the size of scientific datasets demands large amount of disk space. Multidimensional indexing techniques have been shown to greatly improve range query performance into large scientific datasets. In this paper, we discuss several ways of distributing a multidimensional index in order to speed up access to large distributed scientific datasets. This paper compares the designs, challenges, and problems for distributed multidimensional indexing schemes, and provides a comprehensive performance study of distributed indexing to provide guidelines to choose a distributed multidimensional index for a specific data analysis application.

Keywords M
ultidimensional indexing, Distributed indexing, Decentralized indexing, Data intensive computing.

1 Introduction

The Grid is an ideal environment for running applications that require extensive computational and storage resources, such as data-intensive scientific data

B. Nam
School of Electrical and Computer Engineering
Ulsan National Institute of Science and Technology
Ulsan, 689-798, Korea
E-mail: bsnam@unist.ac.kr

A. Sussman
Department of Computer Science
University of Maryland
College Park, MD 20742, USA
E-mail: als@cs.umd.edu

analysis applications. The Grid allows additional resources to be employed incrementally as need arises. As more storage capacity has been needed to store large scientific datasets, recent *data grid* research has focused on developing more scalable distributed storage systems [2, 5, 10, 16].

Scientific instruments can produce hundred of gigabytes of spatio-temporal data daily, consisting of billions of individual data elements. As distributed storage system capacity grows, efficient data discovery mechanisms to locate a specific data item become important. Many scientific datasets are made up of collections of multidimensional arrays, and multidimensional range queries represent an important class of data access patterns in scientific data analysis computing. Multidimensional indexing structures, such as R-trees or its variants, allow for direct access to subsets of a dataset in order to improve data access performance [3, 11]. However, we believe that a single indexing structure is not efficient enough to handle geographically distributed large scientific datasets. For instance, a user may issue a query to retrieve a subset of some specific datasets that are 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, a spatial indexing scheme is needed that is both more scalable and more robust than a centralized index.

Many widely used data discovery mechanisms are based on centralized directory services, such as MCAT (metadata catalog) for the Storage Resource Broker [2, 24]. However, a centralized directory service has several potential problems including server scalability, single point of failure, and single authority administration. A straightforward and widely used method to achieve scalability and avoid single points of failure is *replication* [1,6,17,31,32]. There has been extensive research on data replication, however relatively little effort has been devoted to data discovery mechanisms that are common in the relational database community, such as indexing.

In a distributed environment, although the size of the indexing structure is much smaller than that of the input datasets, an index can become a performance bottleneck since the index tends to be accessed much more frequently than the input data [20]. In the relational database community, some work has done to design distributed indexing schemes, but that work assumes that the placement of the datasets can be controlled for load balancing, and also mainly focuses on declustering datasets using space filling curves [13, 27, 28]. Due to the nature of scientific datasets, it can be very difficult to move around huge datasets, instead our work focuses on distributing the index itself rather than reorganizing the datasets.

Index replication in distributed environments helps improve search performance by spreading workload and also by locating the index closer (in network terms) to clients, but may make updating the index expensive due to consistency requirements across replicas. Instead of replication, we propose a form of hierarchical indexing, which distributes parts of the index onto multiple data servers. Both replication and hierarchical indexing reduce the overhead of a single centralized index. However, a central server is still needed for both indexing schemes, which can be a potential performance bottleneck. As an

alternative way of distributing the index, we have proposed a fully decentralized two level index, called *DiST*, which works in a peer-to-peer fashion [21]. Compared to a replicated index and a simple two-level index, the main benefit of a decentralized index is that there is less potential for a resource bottleneck.

In this paper, we explore and compare the design, challenges, and problems of the distributed multidimensional indexing schemes we have developed over the last several years. We also provide performance models that support experimental results and guidelines for choosing a distributed multidimensional index strategy for data intensive scientific data analysis applications.

The rest of the paper is organized as follows. In Sections 2, 3, and 4 we briefly describe the algorithms for the three different distributed multidimensional indexing schemes, and provide experimental results in Section 5. In Section 6 we describe a performance model and discuss various factors that should be considered when applying multidimensional indexing schemes to data intensive applications. In Section 7 we discuss other research related to distributed indexing, and we conclude in Section 8.

2 Centralized Indexing

In the centralized indexing scheme, which is commonly used in many database applications, a single index server stores all the index tree nodes, as shown in Figure 1. Since data items are distributed across multiple data servers, leaf level nodes in a centralized index contain server names, data file names, and offset information. All range queries must be forwarded to the central index server, and the central server searches its index and returns pointers to the data to the requesting client. After receiving the pointers, clients can request data objects from the specified data servers after parsing the information returned from the index server. Alternately, the central index server can multicast data read requests to the appropriate servers, which may increase the overhead on the central index server. For index insert operations, for when a client stores new data in one or more data servers, the data servers forward index update messages to the centralized index server to complete the operations.

The centralized indexing scheme is easy to implement, but has several drawbacks. First, in a wide area network, network latency may cause significant performance degradation. Second, a centralized index server is a potential resource bottleneck, particularly as the number of data servers and clients scales up to large configurations. Third, centralized indexing has a single point of failure. Even if data servers are accessible, there is no way to search into datasets when the centralized index server is not accessible. A straightforward way of solving these problems is to replicate the centralized index onto multiple servers. Although replication of data objects has been extensively studied in various fields, replication of the index has only started to receive attention recently [20]. The MCAT service in the Storage Resource Broker (SRB), developed at the San Diego Supercomputing Center, is one example of a system that can replicate metadata, such as an index [25]. However, replication

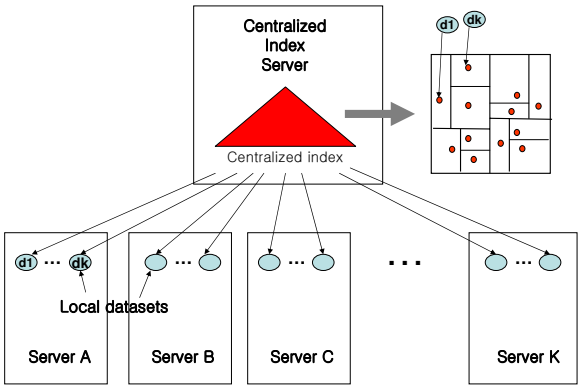


Fig. 1 Centralized Indexing - A single index server stores all the index tree nodes. Since data items are distributed across multiple data servers, leaf level nodes in a centralized index have server names, file names, and offset information. For all range queries, the central server searches its index and returns pointers to the data to the requesting client.

makes index update operations very expensive [20], hence we have proposed alternative indexing schemes that will be discussed in Sections 3 and 4.

3 Hierarchical Two Level Indexing

Because of the algorithms and data structures used for multidimensional indexing, updating or searching an index file in parallel is a good way to distribute the load on a centralized server. There are two ways to parallelize index operations. One method is to replicate the index, while the other is to partition the index and distribute the parts to multiple servers. Partitioning not only spreads client requests across multiple index servers, but also decreases the amount of the work to be done by each server for an index request (search or insert), because each server has a smaller index to operate on.

In hierarchical two level indexing as shown in Fig. 2, 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 local indexes, each of which is only big enough to span all the data in the local server. When a range query is submitted to the server owning the global index, the server compares the range with the MBBs of the local servers 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. However, the global index gives approximate information about local indexes in order to avoid broadcasting queries to all data servers.

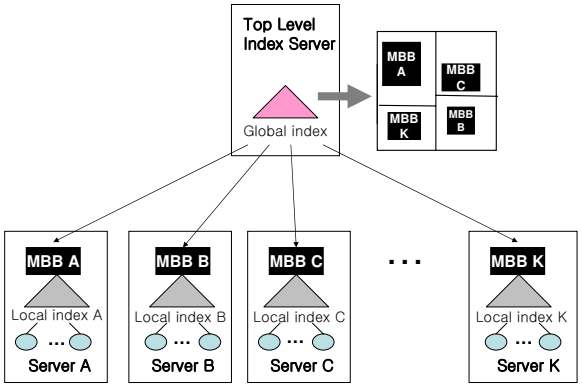


Fig. 2 Two Level Hierarchical Indexing - Each data server has its own local index for local data. For searching the index, a global index is used to determine which local index(es) must be accessed. The local index(es) contain the information about the actual data stored in the data servers.

The size of the top level global index depends on the number of local indexes which is correspondent to the number of data servers, not on the total number of data objects being indexed. Therefore, the size of the global index is much smaller than for the centralized index, so searching the global index is faster than searching the centralized index. The data servers are responsible for searching their own local indexes, reading the parts of the datasets pointed to by the index, and returning the data to the requesting client.

When a sensor device or a simulation stores data into a local server, the data will first be inserted into the local index for that server in the hierarchical two level indexing scheme. When a data object is inserted that is outside the current MBB of the local index, that MBB must be extended to include the new data object. When the low level MBB changes, an update notification is forwarded to the top level global index server. When the global index server receives the update notification, it searches its index, deletes the old MBB and inserts the new one. Most index updates are performed in the local data servers for two level indexing, while all index updates are performed in the central server for centralized indexing.

4 Decentralized Two Level Indexing

Although hierarchical two level indexing parallelizes indexing operations, improving index performance, it is still vulnerable to the single point of failure problem. The global index in hierarchical two level indexing can be replicated to solve the problem, but replication needs a mechanism for creating, locating, and deleting replicas. In much work on replication, replicas are considered read-only copies of data objects, which do not change or do so very infrequently. This assumption does not apply universally, and especially not for an

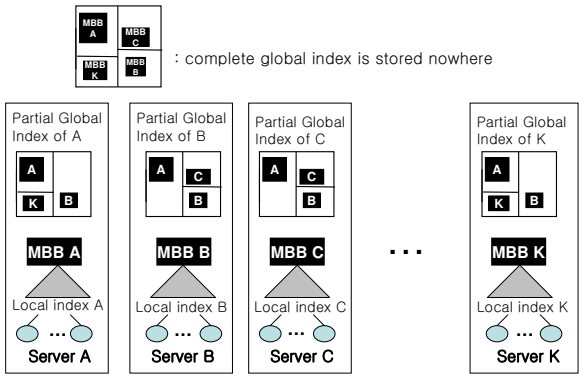


Fig. 3 Decentralized DiST Indexing - DiST is a decentralized version of the two level indexing scheme. The global index is distributed across all the servers.

index, because the index will be modified whenever data is stored, replicated, or deleted.

In this section, we present a fully decentralized two level indexing scheme, which is an extension of hierarchical two level indexing that employs lazy replication mechanism for robustness and efficiency, and peer-to-peer style query routing for correctness.

For a fully decentralized index, several challenges must be considered [7,8]:

- **Distribution**: The index needs to be partitioned across a large number of servers, in order to avoid potential bottlenecks and ensure load balance.
- **Dynamism**: Servers 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**: The search for a data object succeeds if this data 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 of servers involved in a query should be at most in logarithmic order, since broadcasting is not efficient.

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

4.1 Distribution: Decentralized Global Index

DiST is a decentralized version of the two level indexing scheme that we described in Section 3. 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

Algorithm 1
Server Join Algorithm

procedure
$ServerJoin(RootMBB, NewServer)$
1: $OwnerID := GetOwner(RootMBB)$
2: **if** $OwnerID ==$ me **then**
3: $Insert(GlobalIndex, RootMBB, NewServer)$
4: $GlobalIndexCopyForward(NewServer, GlobalIndex)$
5: **else**
6: $JoinRequestForward(OwnerID, RootMBB, NewServer)$
7: **end if**
end procedure

Figure 3. The DiST global index partitions the complete multi-dimensional attribute space (i.e. it is a *space partitioning* spatial index), as is done for KD-trees [4], and each leaf node in the tree corresponds to an MBB of a local index as for the basic two level index.

For the DiST global index, we convert a K dimensional rectangle $[L_1, U_1] \times [L_2, U_2] \times ...[L_K, U_K]$ for an MBB of a local index, where L_k/U_k is the lower/upper bound for dimension k, into a $2{\cdot}K$ dimensional point data $(L_1, U_1, L_2, U_2, ..., L_K, U_K)$ in order to use KD-tree style static space partitioning. The transformation of a rectangle for a range query in K dimensions must be handled differently, because the rectangle must be converted correctly into a rectangle in $2 \cdot K$ dimensions for searching. The converted range query becomes an unbounded rectangle, i.e. the range query $[qL_1, qU_1] \times ...[qL_K, qU_K]$, where qL_k/qU_k is the lower/upper bound of the query for dimension k, is converted into searching for points in the space $[-\infty, qU_1] \times [qL_1, \infty] \times ...[-\infty, qU_K] \times [qL_K, \infty]$. As far as we know there is no decentralized indexing scheme that can store rectangular data without using the point transformation method. Even grid-based DHT routing methods, such as in CAN [26], do not store rectangular data. Data partitioning methods such as R-trees [11] are not suitable in a decentralized environment because their dynamic partitioning strategy can create consistency problems between the global indexes in different nodes. We discuss this issue further in Section 4.2.

When a server joins the system in DiST, 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 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 (i.e., a rectangle in 2D becomes a 4D point) [12, 21]. The insertion algorithm has the previous owner of the partition divide its current space into two parts, and assigns one of the new partitions to the new server. However, the previous owner does not need to forward that split update to all other

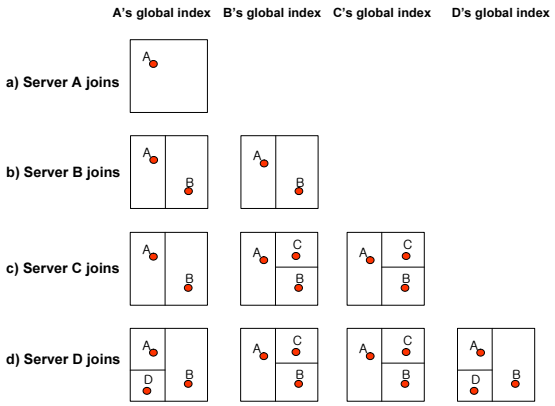


Fig. 4 Server Join in DiST

servers in the system. The reason is that the query routing algorithm can deal with stale index information.

The join algorithm of DiST is shown in Algorithm 1. When a server, call it S, receives a join request from a new server, it searches its own global index ($GetOwner()$) to figure out whether the MBB of the new server is inside the region owned by S. If the new server's MBB is not in S's region, the join request will be forwarded to another server, whose region potentially includes the new server's MBB ($JoinRequestForward()$). Otherwise, the server inserts the MBB into server S's global index ($Insert()$), which results in splitting its region. After insertion, the updated global index is copied to the new server ($GlobalIndexCopyForward()$).

Figure 4 shows an example of a server join. Whenever a new server B joins the system, the server sends a join request to any existing server, A in this example. Since the bounding box of the new server falls inside the region owned by A, A splits the multi-dimensional space it owns and the new server becomes the owner of one of the new partitions. If server C then 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. Finally server D joins, whose bounding box falls inside server A's region. Server A splits its space and forwards a copy of the global index to server D. Note that the global indexes in this example join sequence end up not being consistent.

4.2 Correctness: DiST Query Routing

For searching the index, the DiST query routing algorithm guarantees that any range query will eventually be forwarded to the actual destination owner

Algorithm 2
Range Query Routing Algorithm

procedure
RangeQuery(*QueryMBB*, *QueryID*, *QueryHistory*, *Sender*)
1: **if** QueryID is already processed **then**
2: *QueryResultForward*(*Sender*, *NULL*)
3: **else**
4: *OwnerIDList* := *GetOwnerList*(*QueryMBB*)
5: *QueryHistory*+ = *OwnerIDList*
6: **for all** *OwnerID* in *OwnerIDList* **do**
7: **if** *OwnerID* == me **then**
8: *Result* += *LocalSearch*(*QueryMBB*)
9: *local*$_s$*earch* := TRUE
10: **else if** *OwnerID* is not in *QueryHistory* **then**
11: *QueryRequestForward*(*OwnerID*, *QueryMBB*, *QueryID*, *QueryHistory*, *me*)
12: **end if**
13: **end for**
14: **if** *local*$_s$*earch* **then**
15: *ReturnQueryResult*(*QueryHistory*, *Result*)
16: **end if**
17: **end if**
end procedure

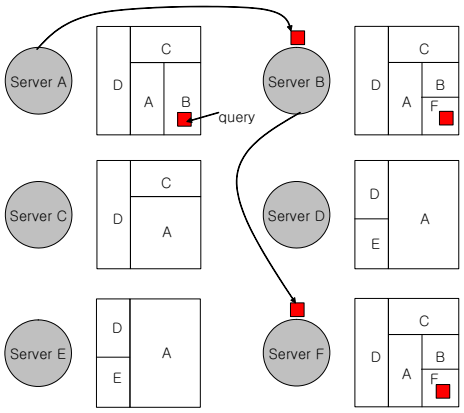


Fig. 5 Query Routing in DiST

server that has the requested data, although the query can be submitted to any server, and none of the servers in the system may have 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, namely KD-trees. The updated global information is propagated in a lazy manner as we will describe later.

Algorithm 2 shows the range query algorithm of DiST. When a server, call it S, receives a range query, it checks whether the query was previously processed. If not, it searches its own global index (*GetOwnerList*()) to get the

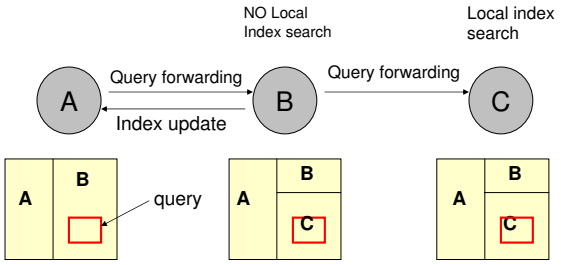


Fig. 6 Lazy index update is triggered when unnecessary forwarding is detected.

list of servers that overlap the given query range. For each server in the list, the algorithm checks whether the server is in the query forwarding history. If a server is not in the history, the algorithm forwards the query to the server. If server S is in the list, S searches its local index ($LocalSearch()$). If the local index is searched, the result must be returned to the client with the query forwarding history, so that client does not have to wait indefinitely for results ($ReturnQueryResult()$).

Figure 5 is an example that shows how DiST query routing works. 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.

Maintaining only a partial global index at each server may result in more network messages and longer routing path for search queries compared to fully propagating global index updates. In the example shown in Figure 6, server A must forward a query to server B, since A does not have partition information for server C. If server A has partition information for server C, 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 Figure 6.

4.3 Dynamism: DiST Recovery

DiST employs the recovery mechanism used for CAN (Content-Addressable Networks) distributed hash tables (DHTs), so that each server in DiST has to maintain a neighbor server list [26]. In DiST, remote servers may send queries directly to the failed server. After detecting a server failure, the remote

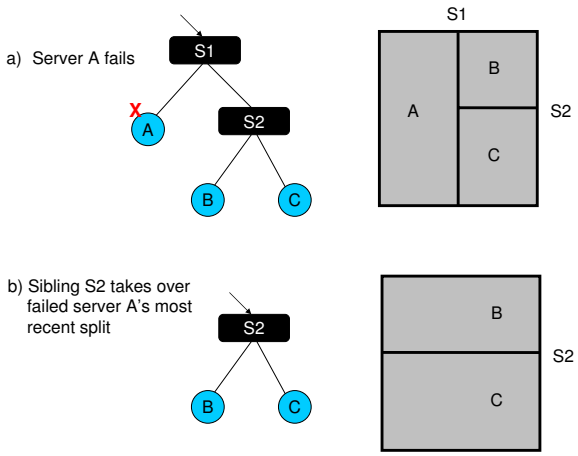


Fig. 7 When a server fails, sibling server(s) in space partitioning tree must take over the failed server's region.

server simply deletes the failed server from its global index. When a node is deleted from the KD-tree index, a sibling node or sibling subtree takes over the partition for the deleted node, as shown in Figure 7. In the example, server *A*'s most recent split S1 must be deleted from the KD-tree, and its sibling (*split S2*) will take the place of S1.

In DiST, this means that the sibling server(s) becomes a replacement server(s) and takes over the partition for the failed server. When a server joins the system and inserts its MBB into the global index, the server that ends up splitting its partition for the new server must forward a global index update message to its neighbors, in order to maintain correct neighbor information. If a failed server 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 overall correctness.

The DiST recovery algorithm has a problem that is common to all decentralized indexing schemes, namely that it cannot recover from multiple simultaneous server failures. In order to make DiST more fault tolerant, the algorithms can be modified to allow multiple servers to own a single partition. If all the servers that share the same partition fail at the same time, the partition can be deleted as described above. If all the servers in more than two adjacent partitions fail simultaneously, that can still cause an unexpected network partition, which is not recoverable in *any* currently available large scale decentralized systems.

4.4 Efficiency: Lazy Index Updates

When the global index is not a balanced KD-tree, the partial global indexes may create a long message chain for a range query search. 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. To improve performance, 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. The intended effect is to replicate the global index across all the data servers in a lazy manner. Lazy 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. In many applications, range queries are much more frequent than update requests, thus lazy index updates will make the partial global indexes become complete and consistent quickly.

Query forwarding history is required to eliminate duplicate lazy index update messages as well as duplicate query processing. However, even with keeping track of the query forwarding history, a server can receive the same query multiple times due to the query routing properties of DiST. Therefore we need to assign a unique query ID to each query. Using the query ID, servers can detect and not forward duplicate queries.

5 Experiments: Distributed Indexing

5.1 Experimental Environment

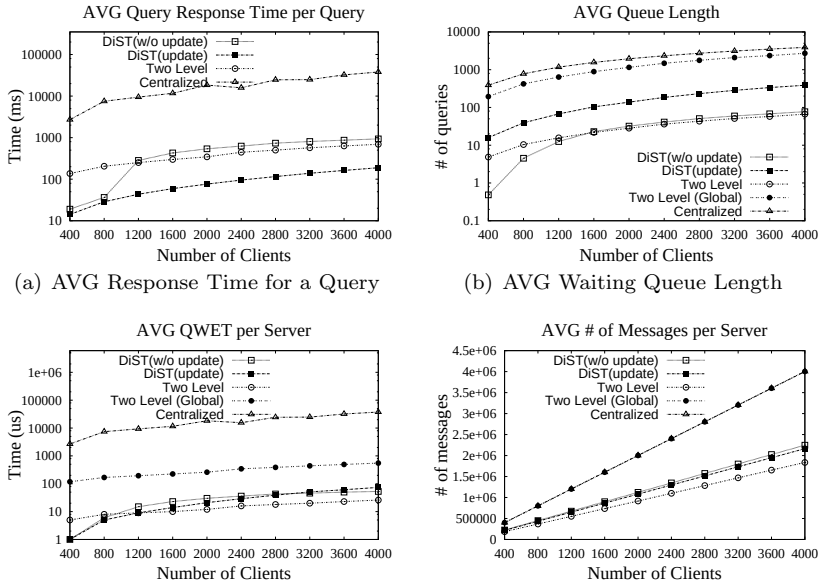
We have measured the performance of the three different indexing schemes on 41 Linux cluster machines. Each of the 41 servers has two Intel Xeon 2.66GHz processors and 2GB of memory, and the servers are connected by a Myrinet network with a nominal maximum transfer rate of 1 gigabit/sec per node. Intercommunication between index servers is done via TCP sockets.

We used a three dimensional remote satellite dataset, from the AVHRR (Advanced Very High Resolution Radiometer) sensor [23, 30]. The satellites orbiting the Earth gather remotely sensed AVHRR GAC (Global Area Coverage) level 1B datasets [22], stored as a set of arrays. Satellite datasets include geo-location fields, time fields, and some additional metadata. As the satellite moves along a ground track over the Earth, it records longitude, latitude, and time values, in addition to the sensor values. Because the sensor swings across the ground track, the sensor values and meta values are stored as two dimensional arrays of structures that contain sensor values and metadata. The raw data collected by satellite sensors can be post-processed to generate satellite images. Kronos [34] is an example of such a class of applications. Kronos allows scientists to carry out Earth system modeling or analysis through a Java interface, using AVHRR GAC data. NOAA's satellites continue to send raw

sensor data at a high rate, with the volume of data for a single day about 1GB. The dataset used for our experiments was collected over one month (January 1992), and has a total size of more than 30GB.

As the satellite moves along a ground track over the earth, its sensor swings across the ground track, and the sensor values and meta values are stored as two dimensional arrays. We partitioned those arrays into 120,000 equal-sized rectangular chunks (each 250 KB), built three dimensional bounding boxes (latitude, longitude, and time) for each chunk, and assigned 3,000 chunks (750 MB) to each of 40 data servers. We used an extra server as the dedicated index server (41 servers total) for the centralized index and for the global index in the hierarchical two level indexing scheme. The clients are distributed evenly across the 40 server machines and each of them submits 1,000 sequential queries, waiting for one query to complete before issuing the next query. Thus the total number of concurrent queries is almost the same as the number of concurrent clients.

In order to create range query workloads, we employed a variation of the *Customer Behavior Model Graph (CBMG)* technique to create realistic query workloads [19]. A CBMG can be characterized by an $n \times n$ matrix of transition probabilities between n states, $P = [p_{i,j}]$, where each state typically represents a stage in an e-business transaction. Similarly, a sequence of data visualization queries in a data analysis application can be seen as moving through different states. In our query model, the first query in a batch specifies a multidimensional point and a set of ranges for each dimension (a geographical region and a set of temporal coordinates, i.e. a continuous period of days). The subsequent queries in the batch are generated based on the following operations: a *new point of interest*, *spatial movement*, *temporal movement*, *resolution increase or decrease*. We have previously selected hot points of interest where an initial query will be centered (e.g., the Amazon rain forest for a hypothetical deforestation-related query). These points are defined in terms of spatio-temporal coordinates. In this way, subsequent queries after the first one in the batch may either remain around that point (moving around its neighborhood) or move on to a different point altogether. These transitions are controlled by the CBMG probability matrix. For the experiments in this paper, we set up the workload modeled by the CBMG to have a 40 % probability of picking a new point of interest for a subsequent query, 20 % probability of a spatial movement operation, 20 % probability of selecting a new resolution, and a 20 % of a temporal movement (selecting a different time period). We also measured the performance of the indexing schemes on other workloads with other values for the CBMG probabilities that are representative of common user query patterns for AVHRR data, and the performance of the distributed indexing schemes are similar to the results shown in this section.



(a) AVG Response Time for a Query

(b) AVG Waiting Queue Length

(c) AVG Query Wait Execution Time for a Server

(d) Communication Cost per Server

Fig. 8 Search performance varying the number of clients

5.2 Experimental Results

Figure 8 shows the search performance of the three indexing schemes for an increasing number of clients. Note that the graphs are log scale. The query response time shown in Figure 8(a) is the amount of time from the moment a query is submitted to the system until it completes. The average query processing time for the two level indexing scheme is approximately 5% that of the centralized indexing scheme when there are 400 concurrent clients, and only 2% of the time when there are 4000 concurrent clients. The communication costs for both indexing schemes are almost the same as shown in Figure 8(d). But the performance gap between the two indexing schemes comes from the difference in size of the centralized index (proportional to the number of MBBs in the index) vs. the global index for the two level scheme (proportional to the number of local data servers).

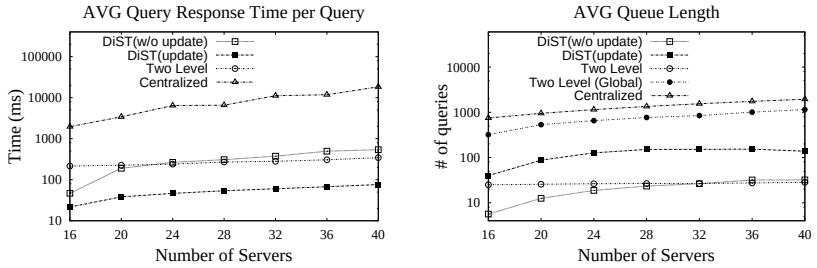
The comparison between hierarchical two level indexing and DiST is more interesting. Under light workloads, DiST (without lazy updates) performs searches faster than the two level indexing scheme does. However, when the system is heavily loaded, the query response time of DiST rapidly increases and becomes longer than for two level indexing. One of the reasons is that DiST has longer routing paths than two level indexing. When the system is not heavily loaded, the long routing path does not hurt overall query response time significantly because the servers are connected via a very fast network

and because the queries do not share query routing paths due to the decentralized nature of the DiST search protocol. However, as more queries are received than the servers can process immediately, queries are enqueued for processing, thus the long routing paths for DiST and queuing delays in the servers become a critical performance factor that increases query response time.

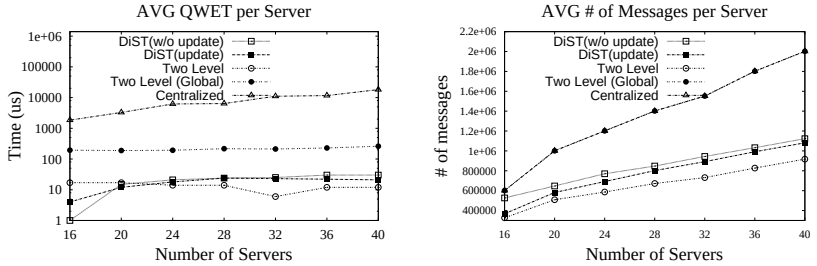
Figure 8(b) shows the average length of the waiting queues for query requests. Waiting queue length measures the number of waiting queries at a server when a query is enqueued, whereby we can measure the instantaneous server load. DiST has the shortest waiting queue length when the number of concurrent clients is 400. However as more clients submit queries, the number of waiting queries for DiST grows faster than that of two level indexing because the DiST implementation has slightly higher overhead for computing the query routing path than two level indexing. This causes more queries to be enqueued for DiST compared to two level indexing. On the other hand, using lazy index update messages makes DiST behave very differently (labeled as DiST(update) in the graphs). The size of a lazy index update message is much larger than that of a query message and it takes a significant amount of time to merge two partial global indexes relative to the time to process a range query, hence lazy index update messages makes enqueued queries wait longer than even DiST without lazy updates. In spite of the longer waiting queue, the reason why DiST(update) has the fastest query response time is that once the partial global index is updated, each server can directly forward client range queries to the correct data servers with a single routing hop, and there is no global index server bottleneck due to the decentralized nature of the protocols. Figure 8(b) also shows that the global index server for two level indexing has a high queuing delay.

The query wait and execution time (QWET) shown in Figure 8(c) measures the execution time and queuing delay from the moment a query is enqueued on a server until the server forwards it to other servers for further processing or returns to the client. QWET is different from query response time in many ways. First, the long QWET for DiST(update) does not mean slow query response time, due to that method's short routing path. On the other hand, DiST without lazy updates has a smaller QWET than the global index server for two level indexing, but is slower when measuring query response time. Also DiST both with and without lazy updates has a higher QWET than for the data servers for two level indexing, but has faster query response time than two level indexing. We measured QWET for the global index server and for the data servers in two level indexing separately because they have very different performance behaviors. These results clearly illustrate that the global index server in the two level indexing scheme is a serious performance bottleneck, similar to the problems seen for the centralized indexing server. In Figure 8(b), the length of the queue for the global index server is half to two thirds that of the centralized index server, but eight to thirteen *times* that of the data servers for DiST(update).

Figure 8(d) shows the average number of network messages received by a server. The number of network messages for centralized indexing is almost the



(a) AVG Response Time for a Query (b) AVG Waiting Queue Length



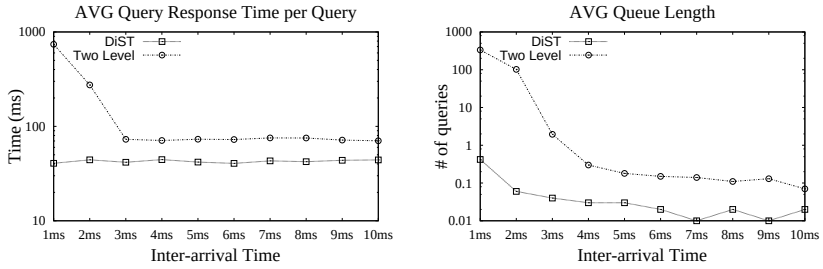
(c) AVG Query Wait Execution Time for a Server (d) Communication Cost per Server

Fig. 9 Search performance varying the number of servers

same as that for the global index server with two level indexing. Since DiST does not have any centralized server, the network messages are well distributed across the system, but the number of network messages is slightly higher overall than for the data servers in two level indexing because of routing overhead. In order to update the partial global indexes using lazy update messages, DiST requires additional network messages. However after the index is updated, the number of network messages are reduced because query routing paths become shorter. Thus DiST(update) eventually generates fewer network messages than DiST without lazy updates, as shown in Figure 8(d).

In order to evaluate the scalability of the indexing schemes, we measured index search performance varying the number of data servers from 16 to 40, as shown in Figure 9. Each data server receives queries from 50 local clients (i.e. 2000 total clients with 40 data servers), and each of the clients submits 1000 sequential queries. Hence more data servers means more concurrent queries as well. As we add more data servers, the size of the centralized index and the global index for two level indexing increases linearly. However, since the size of the global index for two level indexing is very small compared to the fully centralized index, the QWET for the global index server does not increase much for more data servers. In Figure 9(a), DiST without lazy updates performs worse as the number of data servers increases because the routing path for a query becomes longer with more data servers. Query response time for DiST(update) and for two level indexing also grows, but not as much as for

DiST without lazy updates. Figure 9(b) shows that the average wait queue length for all the indexing schemes, except for the data servers in the two level indexing scheme, increases as the number of data servers increases. However, the centralized index and the global index server for the two level indexing scheme suffer more from resource contention than does DiST.

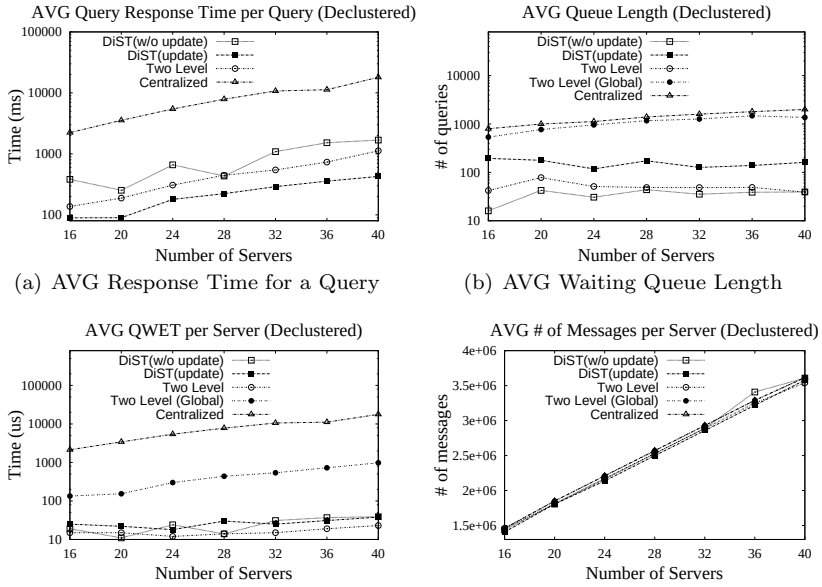


(a) AVG Response Time for a Query (b) AVG Waiting Queue Length

Fig. 10 Simulation Results

If we employ thousands of data servers, the global index server for the two level scheme would become a bottleneck, as would a centralized index. However, since we do not have access to that many servers, we implemented an event driven simulator to model the behavior of all the indexing schemes and observed that the global index server also becomes a bottleneck with thousands of data servers and performs much worse than DiST. Figure 10 shows simulated search performance for hierarchical two level indexing and for DiST. In this simulation, we distributed 120,000 multidimensional chunks across 1000 data servers. The average latency of a packet between two servers was fixed at 50ms, to simulate average wide area network latencies, and the times for searching the local index in the simulation were measured from doing the actual lookup as part of the simulation, and took less than 1ms in most cases. The time for searching the global index was approximately 2ms on the machine where we ran the simulation. Figure 10(b) shows that when the average query inter-arrival time is less than the time for searching a global index, more than one query is enqueued for hierarchical two level indexing, and the query response time for two level indexing becomes very long. In the experiments described previously, the reason why the global index server becomes a bottleneck is mainly due to network congestion, but the simulation results show that the global index server can also become a computation bottleneck.

In order to maximize parallelism for accessing spatio-temporal datasets, it has been suggested that large datasets be declustered across distributed storage archives using space filling curves, such as Hilbert curves [14]. In such a case, we expect that two level indexing will have poor performance, because the MBB for the root of each local index would cover the entire spatio-temporal range of the whole dataset. That would cause the global index server



(a) AVG Response Time for a Query

(b) AVG Waiting Queue Length

(c) AVG Query Wait Execution Time for a Server

(d) Communication Cost per Server

Fig. 11 Search performance with declustered datasets

to effectively broadcast all queries to all data servers. Decentralized two level indexing (DiST) would have the same problem for declustered datasets, resulting in long routing paths for query forwarding. In order to determine how declustering affects index search performance, we ran the same experiments as shown in Figure 9, after declustering the datasets in a round robin fashion. Round robin had the same effect as space filling curve declustering, making all the MBBs for the local indexes have similar spatio-temporal attributes. The experimental results shown in Figure 11 show that the performance of DiST and two level indexing are not as good as for the clustered dataset experiments shown previously, as expected, but these schemes are still faster than centralized indexing. DiST(update) is the biggest victim of declustering, and its query response time became 4-7 times slower than for the clustered dataset experiments, while hierarchical two level indexing became 2-3 times slower. Note that we do not include the time to read the actual datasets. Therefore, the performance degradation is purely from broadcasting query messages. As shown in Figure 11(d), the number of network messages across all the different indexing schemes is about the same because most of the data servers will receive all the queries with declustering.

In our last set of experiments, we measure insertion performance for the distributed indexing schemes. Figure 12 shows the elapsed time to insert data chunks into the index. The number of data servers used was 40 and each data server has one client that performs the data chunk insertion operations into

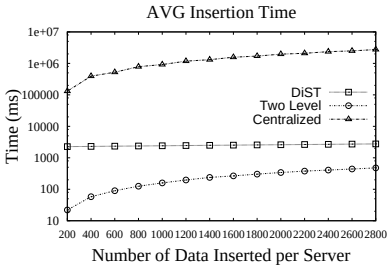


Fig. 12 Insertion performance

Variable	Description
N	# of data servers
$N_{decluster}$	average # of data servers involved in a query, determined by dataset distribution and query selectivity ($\leq N$)
Q	# of queries
Q_{SEL}	average query selectivity ($0 \leq Q_{SEL} \leq 1$)
H	average # of query routing hops
D_C	average # of disk accesses for a query with a centralized index
D_G	average # of disk accesses for a query for the global part of the two level index
D_L	average # of disk accesses for a query for the local part of a two level index
S_C	size of centralized index
S_G	size of global index
S_L	size of local index

Table 1 Description of variables used to model distributed indexing

the index. As we discussed earlier, most insertions are done only at the local indexes stored in the local data servers, for both two level indexing and DiST. Hence their insertion performance is greatly superior to that of centralized indexing. Although DiST insertion performance appears to be mostly independent of the number of data servers, insertion time increases slightly as for two level indexing, but is always much higher than for two level indexing. Note that the graph is log scale. The gap between DiST and hierarchical two level indexing comes from the overhead of the DiST join algorithm, which can be expensive. An update to the global index is performed by first deleting the old MBB from the partial global index at the local server, then inserting the new MBB into the partial global index (as for a new node join). The lazy update mechanism is then used to propagate the update. For centralized indexing, the high insertion cost comes from the high cost of the computation required to insert into a very large index. For applications that require frequent index updates that propagate to the global index, the decentralized approach does not appear to be a good choice.

6 Performance Model and Discussion

In order to analytically support the experimental results described in Section 5, we present a performance model for searching the index for each of the three distributed indexing schemes, using the variables defined in Table 1.

The number of network messages for centralized indexing and hierarchical two level indexing schemes is proportional to $N_{decluster}$ (the number of data servers involved in a query).

$N_{decluster}$ might not be proportional to query selectivity, Q_{SEL}, because the dataset distribution may only depend on one dimension out of several for the dataset, such as time (and not the spatial dimensions). When the datasets are evenly distributed using space filling curves, such as Hilbert curves [14], $N_{decluster}$ would be the number of data servers (N).

When there is a single centralized index, the average number of disk accesses is

$$DiskAccess_{central} \approx Q \cdot D_C. \tag{1}$$

Similarly the average number of disk access to the global index for a single server in the hierarchical two level indexing scheme is $Q \cdot D_G$, while the average number of disk accesses to a single partial global index in the decentralized indexing scheme is $H \cdot \lceil \frac{Q}{N} \rceil \cdot D_G$. Additionally the average number of disk accesses to a local index in both two level indexing and DiST is $Q \cdot \frac{N_{decluster}}{N} \cdot D_L$. Therefore the average number of disk access in the hierarchical two level indexing scheme is

$$\begin{aligned} &AvgDiskAccess_{hierarchical} \\ &\approx Q \cdot D_G + Q \cdot \frac{N_{decluster}}{N} \cdot D_L \end{aligned} \tag{2}$$

And the average number of disk access in decentralized two level indexing scheme is

$$\begin{aligned} &AvgDiskAccess_{decentralized} \\ &\approx H \cdot \lceil \frac{Q}{N} \rceil \cdot D_G + Q \cdot \frac{N_{decluster}}{N} \cdot D_L \end{aligned} \tag{3}$$

As we showed in Section 5, D_G can be ignored when there are a small number of data servers. Even when there are a large number of data servers, D_C is unlikely to be smaller than D_G.

The size of an index file depends on the number of data objects indexed (m), the fan-out degree for an internal node in the index tree (its number of children), and tree node utilization [11]. If we assume that the tree node utilization is 100%, the number of leaf nodes (m) in the index will be $(number\ of\ data\ objects)/k$, where k is the fan-out degree. Then the size of the centralized index (S_C) is

$$\begin{aligned} S_C &\approx m + \frac{m}{k} + \frac{m}{k^2} + ... + \frac{m}{k^{\log_k m}} \\ &= m + \frac{m-1}{k-1} \end{aligned} \tag{4}$$

When we distribute the data objects across N data servers, the size of a local index for the hierarchical two level indexing scheme S_L is:

$$S_L \approx \frac{m}{N} + \frac{\frac{m}{N} - 1}{k-1} = \frac{1}{N} \cdot (m + \frac{m-N}{k-1}) \tag{5}$$

and the size of the global index is

$$S_G \approx \frac{N}{k} + \frac{\frac{N}{k} - 1}{k-1} = \frac{N-1}{k-1}. \tag{6}$$

From formulas 5 and 6, we may substitute S_L by $\frac{S_C}{N}$ since the number of data objects is usually much larger than the number of data servers ($m \gg N$).

In general, the average number of disk accesses per query D_G, D_L, and D_C depends on the size of the index files (S_G, S_L, S_C) and the query selectivity (Q_{SEL}). Thus we can substitute D_C in equation 1 by $S_C \cdot Q_{SEL}$. In equation 2 we can substitute D_L by $S_L \cdot Q_{SEL} \approx \frac{S_C}{N} \cdot Q_{SEL}$ and D_G by $S_G \cdot Q_{SEL} \approx \frac{N-1}{k-1} \cdot Q_{SEL}$. We can then rewrite the average number of disk accesses to the centralized index as

$$AvgDiskAccess_{central} \approx Q \cdot D_C \approx Q \cdot S_C \cdot Q_{SEL} \tag{7}$$

The average number of total disk accesses to both the global and local index in hierarchical two level indexing is then

$$AvgDiskAccess_{hierarchical} \tag{8}$$

$$\approx Q \cdot D_G + Q \cdot \frac{N_{decluster}}{N} \cdot D_L$$

$$\approx Q \cdot \frac{N-1}{k-1} + Q \cdot N_{decluster} \cdot S_C \cdot \frac{Q_{SEL}}{N^2} \tag{9}$$

and the average number of total disk accesses in decentralized two level indexing is

$$AvgDiskAccess_{decentralized} \tag{10}$$

$$\approx H \cdot \lceil \frac{Q}{N} \rceil \cdot D_G + Q \cdot \frac{N_{decluster}}{N} \cdot D_L \tag{11}$$

$$\approx H \cdot \lceil \frac{Q}{N} \rceil \cdot \frac{N-1}{k-1} + Q \cdot N_{decluster} \cdot S_C \cdot \frac{Q_{SEL}}{N^2}$$

Formulas 7, 9, and 11 support the experimental results discussed in Section 5 measuring I/O performance. With respect to the number of network messages required, hierarchical two level indexing and centralized indexing are the same, but messages are the major performance issue for decentralized indexing. The number of network messages for DiST not only depends on query selectivity, Q_{SEL}, the number of data servers (N), and the data distribution ($N_{decluster}$), but also is dependent on the order of server joins, which determines the partial global indexing tree balance ($log_2 N \leq$ partial global index height $\leq N$). However query routing path length will eventually converge to 1 with lazy index updates.

Design Criteria	Indexing Scheme			
	Centralized	Two Level	DiST(no update)	DiST(update)
Scalability	Worst	Good	Bad	Best
Heavy Workload	Worst	Good	Bad	Best
Clustered Datasets	Worst	Good	Bad	Best
Declustered Datasets	Worst	Good	Bad	Better
Frequent Update	Worst	Best	Bad	Bad

Table 2 Design criteria for distributed multidimensional indexing

6.1 Discussion

In order to compare the different indexing schemes, it is important to determine application domains where assumptions about the datasets and indexing characteristics hold, since each of the three different indexing schemes has strengths and weaknesses. We now discuss some of the lessons we have learned from the experimentals results and the performance model, so that the appropriate indexing scheme can be chosen for a particular application.

First of all, "index update frequency" is an important factor to consider, to understand whether the index will be static or dynamically updated, and if so, how frequently. Second, "the scalability of the system" (i.e. the number of data servers) must be considered. Third, we need to perform "workload characterization" to determine system properties such as expected query inter-arrival time. Fourth, "dataset characterization" must be performed, to understand dataset properties such as clustering and data distribution. Table 2 shows the anticipated relative performance of the multidimensional indexing schemes for the different criteria.

Index Update Frequency: The insertion performance model is similar to those shown above, except that query selectivity (Q_{SEL}) would be very small ($1/m$) and D_G would be 0 for most insertions in hierarchical two level indexing. Also hierarchical two level indexing does not have query routing overhead (H). Even though query routing path length is short, the DiST insertion algorithm is very expensive compared to hierarchical two level indexing as shown in the experiments. Hence the decentralized approach does not appear to be a good choice for applications that frequently update the index.

Scalability of the System: The number of data servers (N) affects not only the size of index but also the query routing path length (H). Distributing the index across a large number of data servers will decrease the size of the index in each server but increase the query routing path length, hence the trade-off between the two effects determines the overall scalability of the decentralized indexing scheme. Based on formula 9 and the simulation results shown in Figure 9, hierarchical two level indexing gets the most benefit from a large number of data servers. However lazy index update makes the average number of disk access for DiST similar to that of hierarchical indexing by reducing H to 1. Search performance for central indexing is not directly related to the number of data servers, but it is likely that more data servers will have

more data and a larger index in a central server will makes centralized index search slow. For large datasets, partitioning the index seems to always be a good choice for highly scalable systems.

Workload Characterization: In many cases, one of the most important issues for designing middleware systems is accurate characterization of the expected application workload, such as query inter-arrival patterns. Query inter-arrival time is also a key aspect for selecting a distributed indexing scheme. Under heavy workload, partitioning the index not only disperses query workload but also reduces the possibility of any central bottleneck. Query selectivity is one of the most important performance factors in multidimensional indexing performance study. But from our performance models query selectivity does not seem to be a critical performance factor that determines the best distributed indexing scheme. For example, $N_{decluster}$ depends more on data distribution rather than query selectivity. In addition to query inter-arrival time and query selectivity, the distribution of clients across the network is another workload characteristic that may affect the performance of the distributed indexing schemes. An assumption behind the performance model and our experiments is an even distribution of clients. Even for DiST with lazy index updates, if most of clients submit queries to a single server, DiST would be no better than hierarchical two level indexing.

Dataset Characterization:

The distribution of datasets across servers ($N_{decluster}$) affects the query forwarding pattern (H) for the decentralized indexing schemes, as was shown in Section 5. In the worst case H can be equal to N. However, $AvgDiskAccess_{central}$ is still greater than $AvgDiskAccess_{decentralized}$ when the size of the index is large enough:

$$AvgDiskAccess_{central} - AvgDiskAccess_{decentralized}$$
$$\approx Q \cdot (S_C \cdot Q_{SEL} \cdot (1 - \frac{N_{decluster}}{N^2}) - \frac{N-1}{k-1})$$

In addition to dataset distribution, the size of the datasets, not shown in the performance model, is another characteristic that can affect the relative performance of the indexing schemes. A large dataset on a single server leads to a large local index, which will increase QWET (query wait execution time) for that data server. As we have seen in the experiments for the declustered datasets, large QWET seems to be a more critical performance issue for DiST than for the other distributed indexing schemes, since high QWET also increases query forwarding delay.

7 Related Work

R-trees were one of the first multidimensional spatial indexing data structures to be developed [11]. Kamel and Faloutsos [13] extended that work, by proposing parallel R-trees (Multiplexed R-trees). One of the limitations of that approach was that it targets a single CPU and multiple disks. The limitation

was overcome by Master R-trees [14] and Master Client R-trees [28], both designed for distributed memory parallel environments. Both approaches assume that datasets are declustered using a space filling curve, and relative stability of the indexed datasets (i.e., few updates). Both assumptions may not hold true in many scientific data analysis applications.

Master R-trees and Master Client R-trees require at least one dedicated server to maintain global status information about the distributed index. To avoid centralized accesses, a few fully decentralized indexing structures have been proposed, and are collectively called SDDS (Scalable Distributed Data Structures). These include LH* [18], which generalizes Linear Hashing to distributed systems, and distributed random trees (DRT) [15]. 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.

In structured peer-to-peer (P2P) systems, peers are directly addressed, typically via a hashing scheme, to return the data objects they contain. The Chord [29] and CAN [26] systems implement distributed hash tables (DHTs) 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. P-trees [7], MURK (Multidimensional Rectangulation with KD-trees) [8], and SkipIndex [33] support one dimensional or multidimensional range queries on P2P networks. These approaches are similar to our decentralized indexing scheme (DiST), however DiST targets stable and less dynamic systems. Hence DiST stores significant amounts of information in a server about neighboring servers in the P2P overlay network. In addition, DiST assumes that each data server has its own index for locally stored data, and the DiST algorithms work on top of those local indexes (i.e., a two level index). Another important difference between DiST and P2P networks is that DiST trades off some scalability, due to the greedy collection of peer information, with the benefit of improved search performance. Since most P2P DHTs target large scale P2P overlay networks, they limit the number of remote peers that can be directly accessed by a given peer [9, 26, 29]. If the number of peers directly accessible by a single peer is limited, the number of routing hops to a destination server generally increases. Since our main concern is range query performance rather than an arbitrarily scalable P2P overlay network, we do not limit the number of peers directly accessible by a single peer.

In broadcast-based P2P systems (also called unstructured P2P systems) such as Gnutella [9], message flooding is employed to forward queries, since each peer does not have data placement information. Message flooding does not guarantee accurate query results, thus is not feasible for data requests in scenarios that require accurate query results, as typically occur in scientific data analysis applications.

8 Conclusions

In this paper we have described and compared several distributed multidimensional indexing schemes. Our experimental study demonstrates that hierarchical two level indexing performs well in most situations, scaling well with the number of servers, with the size of the dataset, and with the workload offered by clients. However, the decentralized approach performs better than the other schemes under some conditions, such as when index update operations are not too frequent, so that we can efficiently update partial global indexes using lazy index update messages.

We have shown that partitioning the index across multiple machines is always a good choice to obtain the best performance, from experiments, simulations, and a performance model. However, as a method of partitioning a index, we need to continue to investigate decentralization techniques in order to solve consistency and update issues that can degrade overall performance.

References

1. B. Allcock, J. Bester, J. Bresnahan, A. Chervenak, I. Foster, C. Kesselman, S. Meder, V. Nefedova, D. Quesnel, and S. Tuecke, "Secure, efficient data transport and replica management for high-performance data-intensive computing," *Proceedings of IEEE Mass Storage Conference*, 2001.
2. C. Baru, R. Moore, A. Rajasekar, and M. Wan, "The SDSC Storage Resource Broker," *Proceedings of CASCON'98 Conference*, December 1998.
3. N. Beckmann, H. P. Kriegel, R. Schneider, and B. Seeger. "The R^*-tree: An efficient and robust access method for points and rectangles," *Proceedings of 1990 ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 322–331, May 1990.
4. J. L. Bentley. Multidimensional binary search trees used for associative searching. In *Communications of the ACM 18(9)*, 1975.
5. Biomedical informatics research network, *http:/www.nbirn.net*.
6. A. Chervenak, E. Deelman, I. Foster, L. Guy, and W. Hoschek. Giggle: A framework for constructing scalable replica location services. In *Supercomputing (SC 2001)*, 2001.
7. 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 (WebDB)*, 2004.
8. P. Ganesan, B. Yang, and H. Garcia-Molina. One torus to rule them all: Multidimensional queries in p2p systems. In *WebDB '04: Proceedings of the 7th International Workshop on the Web and Databases (WebDB)*, pages 19–24. LNCS, 2004.
9. Gnutella website. *http://www.gnutella.org*.
10. GriPhyn: Grid physic network. *http:/www.griphyn.org*.
11. A. Guttman. R-trees: A dynamic index structure for spatial searching. In *Proceedings of 1984 ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 47–57, 1984.
12. 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 (VLDB)*, pages 45–53, 1989.
13. I. Kamel and C. Faloutsos. Parallel R-trees. In *Proceedings of 1992 ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 195–204, 1992.
14. N. Koudas, C. Faloutsos, and I. Kamel. Declustering spatial databases on a multicomputer architecture. In *Proceedings of the 5th International Conference on Extending Databases Technology (EDBT)*, 1996.

15. B. Kroll and P. Widmayer. Distributing a search tree among a growing number of processors. In *Proceedings of 1994 ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 265–276, 1994.
16. J. Kubiatowicz, D. Bindel, Y. Chen, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, and B. Zhao. Oceanstore: An architecture for global-scale persistent storage. In *Proceedings of ACM ASPLOS*, 2000.
17. H. Lamehamedi, B. Szymanski, Z. Shentu, and E. Deelman. Data replication strategies in grid environments. In *the 5th International Conference on Algorithms and Architectures for Parallel Processing (ICA3PP)*, 2002.
18. W. Litwin, M. A. Neimat, and D. A. Schneider. *LH**: Linear hashing for distributed files. In *Proceedings of 1993 ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 327–336, 1993.
19. D. A. Menasce and V. A. F. Almeida. *Scaling for E-Business: Technologies, Models, Performance, and Capacity Planning*. Prentice Hall PTR, 2000.
20. B. Nam and A. Sussman. Spatial indexing of distributed multidimensional datasets. In *Proceedings of the 5th IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid)*, May 2005.
21. B. Nam and A. Sussman. DiST: Fully decentralized indexing for querying distributed multidimensional datasets. In *Proceedings of 20th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2006.
22. National Oceanic and Atmospheric Administration. *NOAA Polar Orbiter User's Guide – November 1998 Revision*. compiled and edited by Katherine B. Kidwell. *Available at http://www2.ncdc.noaa.gov/ docs/podug/cover.htm*.
23. NOAA satellite and information service. Advanced Very High Resolution Radiometer - AVHRR, 2005. *http://noaasis.noaa.gov/NOAASIS/ml/avhrr.html*.
24. A. Rajasekar, M. Wan, and R. Moore. MySRB & SRB – components of a data grid. In *Proceedings of the 11th IEEE International Symposium on High Performance Distributed Computing (HPDC)*, July 2002.
25. A. Rajasekar, M. Wan, R. Moore, and W. Schroeder. Data grid federation. In *Proceedins of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA)*, 2004.
26. 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.
27. H. Sagan. *Space-Filling Curves*. Springer-Verlag, 1994.
28. B. Schnitzer and S. T. Leutenegger. Master-Client R-Trees: A new parallel R-tree architecture. In *Proceedings of 11th International Conference on Scientific and Statistical Database Management (SSDBM)*, pages 68–77, 1999.
29. 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.
30. J. R. G. Townshend. Global data sets for land applications from the advanced very high resolution radiometer: an introduction. *International Journal of Remote Sensing*, 15:3319–3332, 1994.
31. S. Vazhkudai, S. Tuecke, and I. Foster. Replica selection in the globus data grid. In *Proceedings of the 1st IEEE International Symposium on Cluster Computing and the Grid (CCGrid)*, pages 106–113, 2001.
32. O. Wolfson, S. Jajodia, and Y. Huang. An adaptive data replication algorithm. *ACM Transactions on Database Systems*, 22(2):255–314, 1997.
33. 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.
34. Z. Zhang, J. JáJá, D. Bader, S. Kalluri, H. Song, N. E. Saleous, E. Vermote, and J. R. G. Townshend. Kronos: A Java-based software system for the processing and retrieval of large scale AVHRR data sets. Technical Report EECE-TR-99-006, University of New Mexico, November 1999.