ABSTRACT

Title of Dissertation: DISTRIBUTED MULTIDIMENSIONAL INDEXING FOR SCIENTIFIC DATA ANALYSIS APPLICATIONS

Beomseok Nam, Doctor of Philosophy, 2007

Dissertation directed by: Professor Alan Sussman
Department of Computer Science

Scientific data analysis applications require large scale computing power to effectively service client queries and also require large storage repositories for datasets that are generated continually from sensors and simulations. These scientific datasets are growing in size every day, and are becoming truly enormous. The goal of this dissertation is to provide efficient multidimensional indexing techniques that aid in navigating distributed scientific datasets. In this dissertation, we show significant improvements in accessing distributed large scientific datasets.

The first approach we took to improve access to subsets of large multidimensional scientific datasets, was data chunking. The contents of scientific data files typically are a collection of multidimensional arrays, along with the corresponding metadata. Data chunking groups data elements into small chunks of a fixed, but data-specific, size to

take advantage of spatio-temporal locality since it is not efficient to index individual data elements of large scientific datasets.

The second approach was the design of an efficient multidimensional index for scientific datasets. This work investigates how existing multidimensional indexing structures perform on chunked scientific datasets, and compares their performance with that of our own indexing structure, SH-trees. Since R-trees were proposed, various multidimensional indexing structures have been proposed. However, there are a relatively small number of studies focused on improving the performance of indexing geographically distributed datasets, especially across heterogeneous machines. As a third approach, in an attempt to accelerate indexing performance for distributed datasets, we proposed several distributed multidimensional indexing schemes: replicated centralized indexing, hierarchical two level indexing, and decentralized two level indexing.

Our thorough experimental results show that great performance improvements are gained from distribution of multidimensional index. However, the design choices for distributed indexing, such as replication, partitioning, and decentralization, must be carefully considered since they may decrease the overall performance in certain situations. Therefore, this work provides performance guidelines to aid in selecting the best distributed multidimensional indexing scheme for various systems and applications. Finally, we describe how a distributed multidimensional indexing scheme can be used by a distributed multiple query optimization middleware as a case-study application to generate better query plans by leveraging information about the contents of remote caches.

DISTRIBUTED MULTIDIMENSIONAL INDEXING FOR SCIENTIFIC DATA ANALYSIS APPLICATIONS

by

Beomseok Nam

Dissertation submitted to the Faculty of the Graduate School of the
University of Maryland, College Park in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
2007

Advisory Committee:

Professor Alan Sussman, Chairman/Advisor
Professor William D. Dorland
Professor Jeffrey K. Hollingsworth
Professor Joseph F. JáJá
Professor David M. Mount
Dr. Henrique C. M. Andrade, IBM T.J. Watson

©Copyright by

Beomseok Nam

2007

DEDICATION

To my parents – Jasoon Goo and Kyung-Won Nam for their love and support.

ACKNOWLEDGEMENTS

Acknowledgements

My gratitude to those who have helped me complete this dissertation cannot be adequately expressed here. Please accept my apologies if you find yourself unjustly missing of find your contribution inadequately credited. I am truly grateful to all those who supported me one way or another.

I have been very fortunate to work with my thesis advisor, Dr. Alan Sussman, whose suggestions led me throughout this thesis. He has encouraged me in all the time of research and has guided me on the right track with his penetrating insight to all the problems I had. I am indebted to my former colleague, Henrique Andrade, who assisted me with small and big things and always had his doors and ears open for whatever I wanted.

Several faculty provided me with their guidance about my proposal and feedback about this work, including Dr. Jeff Hollingsworth, Dr. Joseph Ja'Ja', Dr. David Mount, and Dr. William Dorland. I am very grateful to Dr. Kern Koh, my M.S. degree advisor at Seoul National University,

who introduced me how to do research and encouraged me to come to the United States for advanced education. Hyokyung Bahn, who studied with me during the M.S. courses, also incited me to decide studying in the united states.

My close friends in Korea, who have made the life in graduate school bearable by chatting through email, messenger, cyworld, or whatever, deserve all the credit for making me keep working. Also, I am grateful to my friends in College Park and KGCS members for spending their spare time with me. Without them I couldn't have completed such a long journey of graduate school. Especially I was fortunate to have friends, Jae-Yong Lee and Youngmin Kim, who struggled together not to starve in College Park everyday.

Finally, I would like to give my special thanks to my parents, family, and Su Ryoun for their love.

April 17, 2007

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

Introduction

This dissertation investigates the problem of indexing distributed large scientific datasets for computation–intensive and/or data–intensive applications. Increasingly powerful computers have made it possible for computational scientists and engineers to model physical phenomena in great detail. Scientific applications are not only computationally demanding, but also they generate and process very large datasets (terabytes to petabytes today) that are geographically distributed due to the limited storage capacity of a single site, or to either organizational or technical issues so that datasets are stored where they are produced.

As more storage capacity has become required to store large datasets, recent Data Grid research has focused on developing more scalable distributed storage systems [12, 52]. Large scale distributed storage systems require a data discovery mechanism to locate a specific data item, based on centralized directory services, such as MCAT (meta-data catalog) for the Storage Resource Broker [12].

Many scientific datasets are made up of collections of multidimensional data items (i.e. having space and time attributes). In order to accelerate direct access to subsets of a dataset within a multidimensional range (a so-called *multidimensional range query*), numerous multidimensional indexing structures have been designed, starting with the

seminal work on R-trees [40]. In this dissertation, we focus on multidimensional range queries for distributed scientific datasets. Many scientific applications that process range queries can employ multidimensional indexes to improve overall performance, otherwise they have to scan the entire distributed dataset to find the subset of data that falls within the given range of values for each dimension in the query. Multidimensional range query is one of the most common type of retrieval patterns on multidimensional datasets.

In the past few decades, an enormous amount of research has been done to create efficient multidimensional indexing data structures including R-trees [40], R*-trees [13], Hybrid-trees [22], and so on. While most of them focus on low dimensional geographic information systems (GIS) or high dimensional multimedia retrieval systems, our target applications are scientific data analysis applications, which have different characteristics from images or multimedia data.

First of all, in some scientific applications such as NASA's EOSDIS, satellites continue to send a collection of data at the rate of 3-5 MB/sec. In order to index the data ingested at such fast rate, we must consider the performance of indexing structures in terms of not only searching but also updates. In the database community, most of recent research focus on the performance of index search, thus they sacrifice the update performance in order to accelerate the search performance. Since the index creation time is critical in some scientific applications, we need multidimensional indexing structures that are fast both for searching and updating. However, it is not enough to design an efficient indexing structure for insertion, since the number of data elements to be inserted into the index with a unique geographic location and time value for a single day is approximately 12 million in EOSDIS dataset. This means that we need to store 138 data elements into the index per second. It is very difficult for any existing multidimensional

indexing structure to index data at such fast rate. Therefore, we need an alternative way of indexing such huge datasets.

Second, scientific datasets are being generated continually and they are not likely to be deleted or changed afterward, therefore the size of datasets becomes truly enormous, which makes it difficult to keep the datasets in a single disk storage. Because of the the demand for large storage capacity, the datasets are often stored in distributed parallel storage systems. However, in practice, there is still a limitation in disk storage capacity of cluster machines. Therefore, while recent datasets are stored on disk storage, old datasets are moved to removable storage devices such as tape drives. Another reason why datasets are distributed across multiple machines is that geographically distributed sensor devices attached to satellites, aircraft, telescopes, etc, will store the datasets on their local machines, and large datasets generated by simulations will be stored near where they are produced. For both organizational and technical reasons, it is often not desirable to move or replicate huge datasets onto remote machines.

In order to index such large distributed datasets, distributing the index would maximize parallelism and range query performance instead of having a single centralized index. In database community, some research was done to design distributed indexing structures such as Master R-trees [50] and Master Client R-trees [88]. However, they assume that they can control the placement of the datasets for load balancing, thus datasets are declustered using a space filling curve. However due to the nature of scientific datasets, it is not always feasible to move around huge datasets as I pointed out. Thus, it is hard to guarantee load balancing for scientific datasets using declustering algorithms. In this dissertation, I will show that distributed indexing techniques significantly improves access to distributed scientific datasets, even when datasets are not evenly declustered to get the performance benefit of maximizing parallelism,

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 [67]. However, not much is known about the overhead incurred by an index server. Distributing the index across multiple servers would alleviate the overhead from an overloaded index server and make the index scheme more scalable.

In order to distribute the index, I present and compare three different approaches: replication, hierarchy, and decentralization [67, 68]. One way of distributing the index is replicating the whole index file onto multiple servers, which will make the index search faster. However, replication has some overhead for maintaining the consistency among replicas. Another way of distributing the index is making the index hierarchical and storing small portions of the index onto multiple servers. Hierarchical indexing allows us to distribute the index without considering consistency issues. However, still we need a centralized index server at the top level, which can be a performance bottleneck as in a centralized index. As an alternative way of distributing the index, I propose a fully decentralized index, called *DiST*, which operates in a fully distributed environment with no centralized control as in peer-to-peer techniques. Compared to the replicated index and two-level index, the benefit of a decentralized index is that there is no potential resource bottleneck. In this work, I compare these three distributed indexing schemes and describe how distributed indexing techniques can be used to improve the performance of query processing for the scientific data analysis applications.

1.1 Motivating Applications

The efficient information retrieval of large datasets is important in many fields of science, engineering, and business. Scientists and engineers as well as business managers often need to access, explore, and analyze datasets to gain insight into the problem and to draw meaningful conclusions about the huge and fuzzy information. Indexing is the fundamental method to solve the problem, employed by a large number of applications in various domains. A few examples of motivating applications and generated datasets that would benefit from indexing techniques that I investigated in this work are as follows:

Remotely Sensed Satellite Dataset Processing Applications

The Advanced Very High Resolution Radiometer (AVHRR) is a broad-band, four or five channel scanner, sensing in the visible, near-infrared, and thermal infrared portions of the electromagnetic spectrum. [72] This sensor is carried on NOAA's satellites (from NOAA-6 to NOAA-14) beginning with TIROS-N in 1978. As a satellite moves along a ground track on the earth, AVHRR sensor devices scan across the satellite's ground track, and gather datasets to form an instantaneous field of view (IFOV) at regular intervals. Each scan line has a time value, and each array element in the scan line has latitude and longitude values, hence it comprises a 3 dimensional dataset (Latitude, Longitude, and Time). The size of these datasets is truly enormous, thus we need efficient scalable systems to manage this kind of datasets.

Geographically Distributed Datasets

The datasets generated at multiple scientific laboratories such as tens of thousands of weather stations around the world are used to model and predict weather patterns or

storms, hurricanes, etc. The collection of such huge amounts of data would need more scalable and sophisticated indexing services than just a single index file. Also, numerous scientific simulation applications such as oil reservoir simulation [53] generate spatio-temporal datasets that describe physical scenarios. These simulations can lead to unmanageably large volumes of output data, stored on distributed data storage servers at multiple institutions.

Geographic Information Systems

Large number of Geographic Information Systems (GIS) datasets are managed by multiple private and public agencies, such as the U.S. Geological Survey (USGS), Maryland Department of Transportation (MDOT), etc. While many of the current GIS applications exploit a centralized data repository, some of recently developed GIS systems such as Geotechnical Information Management and Exchange (GIME) [112] targets a multitude of remote datasets under different administrative control.

Computer Vision Applications

Modern computer vision systems employs multiple cameras shooting the same scene from various perspectives. Using the images obtained from the cameras, computer vision applications perform virtual view rendering, complex shape and movement analysis, multi-person tracking, and so on [20, 21]. Basically these multi-perspective computer vision systems with more cameras (more views) can deliver more information about scenes, and reconstruct 3 dimensional features with more accuracy. However, a large number of cameras can produce huge volumes of image or video data unmanageable in a single storage device.

Distributed Query Processing Framework

Scientific datasets can be stored and processed on a cluster of parallel machines with ADR (Active Data Repository) [24, 54] or across a distributed set of machines (the Grid) with DataCutter [17, 18, 16]. DataCutter is a middleware infrastructure that enables processing of scientific datasets stored in archival storage systems across a wide-area network. DataCutter provides support for subsetting of datasets through multidimensional range queries, and application specific aggregation on scientific datasets stored in an archival storage system. DataCutter provides a core set of services, on top of which application developers can implement more application-specific services. One of the goals of DataCutter is to provide common support for subsetting large datasets through multidimensional indexing.

Multiple query optimization has been extensively studied in various contexts including relational databases and data analysis applications. [31, 37, 91, 107, 62, 35] The objective is to exploit subexpression commonality of multiple queries across a set of concurrently executing queries and reduce execution time by reusing cached outputs. MQO developed by Andrade et al. [9, 10] is a multiple query optimization framework for scientific data analysis applications. MQO stores query results in distributed buffer caches in order to reuse them for the incoming queries. In current version of MQO, there is no way of knowing the contents of distributed caches, thus whenever a new query arrives MQO performs linear scanning of distributed caches, which is obviously inefficient. Distributed indexing techniques can be applied to the cache look-up service of MQO in order to accelerate the performance. For the distributed cache look-up service, we must consider the update performance of index as in satellite data analysis applications since whenever a new query is received, we need not only to search the cache but also to update the content of the cache with the new query results.

1.2 Thesis and Contributions

In this dissertation, I support the following thesis: *distributed multidimensional indexing can greatly improve access to distributed large scientific datasets.* To support this thesis, I develop, apply, and evaluate a set of techniques for efficiently access subsets of scientific datasets of particular interest.

More specifically, this dissertation makes the following contributions not discussed in previous indexing research such as:

1. **An approach to increase the efficiency of index via data chunking**

   Due to very large size of scientific datasets, indexing individual data elements of scientific datasets seems to be almost infeasible or at best inefficient because the size of multidimensional indexing structures would be very large and expensive to query and manipulate. Due to the way of storing datasets from sensor devices, most scientific datasets have spatio-temporal locality, whereby data elements can be grouped and a single bounding box for each chunk is stored in order to reduce the index size for better performance. By grouping data elements into chunks, we can get a relatively tight bounding box for the spatio-temporal coordinates. There has not been prior research about exploiting spatial locality to improve the performance of indexing large scientific datasets.

2. **A design of an efficient indexing structure for chunked datasets**

   Data chunking requires an indexing structure that can index rectangular bounding box of each chunk. This dissertation discusses the problem of indexing rectangular data objects (chunked datasets) and presents an efficient multidimensional indexing structure for chunked datasets, which support fast insertion/deletion as well as fast search. Fast index update is as important as fast search in some sci-

entific data analysis applications. To the best of my knowledge, none of prior indexing research was concerned about the index update performance. Most of existing indexing structures sacrifice index update performance for search performance. But I developed an indexing structure that has no trade-offs between update and search.

3. **A set of techniques for distributing index for distributed datasets**

   This work provided three distributed multidimensional indexing schemes for highly scalable systems. First, I discuss the problem of centralized indexing and present its potential scalability problem and how replication improves the performance. Second, I present a hierarchical two level indexing which partitions a whole index into small local indexes and distributes clients' queries over multiple local index servers. Third, I introduce a decentralized two level indexing that eliminates any central resource bottleneck for highly scalable but static systems.

4. **Analyzing the design choices that affect the performance of distributed indexing**

   This work explores and 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 application. Also, I show how a distributed query processing framework can employ a distributed multidimensional indexing scheme based on the guideline that I present, and its performance benefits in terms of query response time.

1.3 Thesis Organization

The rest of this dissertation is organized as follows. In Chapter 2, I summarize the main research areas related to our work and discusses many relevant previous works. Chapter 3 elaborates on representative data analysis applications that I used to evaluate the performance of proposed indexes. In Chapter 4, I discuss data chunking, which helps generating more efficient indexes by reducing the number of objects in the index. Chapter 5 compares several multidimensional indexing structures with chunked datasets and proposes a new multidimensional indexing structure, called Spatial Hybrid trees (SH-trees). In Chapter 6, I explore a few different indexing techniques for distributed multidimensional datasets, including *DiST*, a fully decentralized indexing scheme. In Chapter 7, I show how a distributed query processing framework takes an advantage of a distributed indexing scheme as a case study. Finally Chapter 8 presents conclusions, summarizes the work, and points out possible directions for future work.

Chapter 2

Related Work

2.1 Indexing Scientifi c Datasets

Scientific instruments and simulations are creating data volumes that almost double each year, and new computing methods that analyze and organize such huge datasets have become necessary, but data analysis tools have not kept pace with them. Jim Gray et al [39] pointed out "Scientists need a way (1) to use intelligent indices and data organizations to subset the search, (2) to use parallel processing and data access to search huge datasets within seconds, and (3) to have powerful analysis tools that they can apply to the subset of data being analyzed". Many scientific applications use their own proprietary index and formats such as raw binary, ASCII, etc. Although there has been little research to devise generic index for scientific datasets, some efforts have been made to develop self-describing scientific data formats such as FITS [30], netCDF [70], HDF4 [33], and HDF5 [82]. Since we integrated indexing functionality into self-describing scientific data formats [65] as a seminal work, which we will discuss in Chapter 4, Gosink et al [38] accommodated bitmap index into HDF5 library and showed that they have improved the search performance compared to our R-tree based indexing structures. However, the limitation of the bitmap index is that it doesn't

support indexing distributed datasets.

2.2 Parallel and Distributed Multidimensional Index

Since the R-tree was introduced in 1984 for indexing multidimensional objects, little work was done on parallelizing R-tree operations until Kamel and Faloutsos [46] proposed the first parallel R-trees (Multiplexed R-trees) in 1992, for a machine with a single CPU and multiple disks. They investigated various declustering methods that decide how to distribute the leaves of an R-tree across multiple disks. Since that paper several parallel multidimensional indexing structures have been studied to extend Multiplexed R-trees, such as Master R-trees [50] and Master Client R-trees [88].

The Master R-tree was designed for a shared nothing environment (i.e. distributed memory parallel machine) by Koudas et al. [50]. 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. A Master Client R-tree, proposed by Schnitzer et al. [88], 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. Therefore, the master index does not have to keep the pointers to the data objects in its master index. Instead, it contains the server address where its local index must be searched again in order to get pointers to the data objects. The authors claim that the overhead of the master server can be reduced by maintaining a smaller master index file.

Liebeherr et al. [57] evaluated the performance benefits of partitioned B+-tree indexes for a single relation in distributed relational databases. The single relation is par-

titioned across all servers and the ranges for each fragment do not overlap each other. They introduced a partitioned global index (PGI) scheme and compared it with the classical partitioned index scheme (PI), in which each server indexes its own data but does not know about the global status, therefore a broadcast message must be sent to all the servers for each request. In PGI, each server has a master index, hence can forward range queries to the servers that have the requested data. However, since PGI has several serious problems in terms of index update, that work assumed that there will be no index update.

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* [58], which generalizes Linear Hashing to distributed systems, and distributed random trees (DRT) [51]. 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 Chapter 6.

Decentralized indexing has been a hot topic since P2P overlay networks had emerged. Recently Mondal et al. [64] proposed P2PR-trees, a variant of R-trees that targets P2P networks. They showed in a simulation study that P2PR-trees show better scalability than two-level Master Client R-trees, since they do not suffer from the central server bottleneck. However, the P2PR-tree is not fully decentralized, because it requires a large number of dedicated servers that maintain part of a static partitioning of the index. Thus, the P2PR-tree is similar to a replicated version of the Master-Client R-tree [67, 88], and has several other problems related to its static partitioning strategy, which may make it

perform poorly for data non-uniformly distributed in the multidimensional space.

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 [98] and CAN [80] 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 [36], 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 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 [29]. 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.

Ganesan et al. [34] proposed MURK (Multidimensional Rectangulation with KD-trees), which uses KD-trees to break up the data space into rectangles. MURK is similar to the space partitioning of CAN [80], except that MURK tries to split the load

across the servers equally using KD-trees instead of partitioning the data space equally. MURK supports multidimensional range queries in structured peer-to-peer systems, but its query routing algorithm is based on that of CAN; visit neighboring servers recursively until reaching the destination server, because there is no tree structured index, although MURK uses a space partitioning strategy as do KD-trees.

SkipNet and SkipIndex are distributed indexing overlay networks that use the Skip Graph data structure [42, 108]. While SkipNet supports one dimensional range queries, SkipIndex supports multidimensional queries in P2P networks [108]. SkipIndex is a distributed indexing overlay network that uses the Skip Graph data structure [108]. 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.

These P2P DHT based approaches are similar to our decentralized indexing scheme (DiST), however DiST targets stable dynamic systems. Hence DiST stores significant amounts of information in a server about neighboring servers in the 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). As we will show in Chapter 6, the greedy collection of peer information limits scalability but improves 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. 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 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.

2.3 Distributed Query Processing

Optimizations for the execution of multiple queries have been extensively investigated in the context of relational databases [45, 47, 85, 94], deductive databases [23], and agent applications [75]. Other researchers have designed and deployed run-time support and examined strategies for efficient execution of queries in data analysis applications on distributed-memory parallel machines with a disk farm [24, 54] and in distributed, heterogeneous environments [3, 18]. Our work is different from these works in that we optimize multiple query executions by taking advantage of distributed index. Although none of these prior efforts is directly related to distributed indexing, we highlight some of these works related to multiple query optimization framework that we integrated distributed multidimensional indexing into.

For distributed query processing, Rodríguez-Martínez and Roussopoulos [84] proposed database middleware (MOCHA) designed to interconnect distributed data sources. The system handles *data reduction* operators by *code-shipping*, which moves the code required to process the query to the location where the data resides and *data inflation* operators by *data-shipping*, which moves the input data to the client. In many cases, however, data-shipping is not an option due to the size of datasets. For these situations, several highly distributed applications have employed proxy front-ends to great benefit. Beynon et. al. [16] proposed a proxy-based infrastructure for handling data intensive applications, which was shown to reduce the utilization of wide-area network connections, reduce query response time, and improve system scalability. On the other hand, Beynon's approach as well as other proxy-based approaches, including earlier implementations of web proxies [103], rely on a single locally available cache. This approach is inherently less scalable than relying on a collection of cache structures available at multiple backend servers, assuming one can efficiently use them.

In order to effectively leverage multiple backend servers for query processing, methods for load balancing must be considered [109]. In other words, the savings resulting from reusing a cached result has to be weighed against the service time and extra load imposed on the server where the cached result is located. One study in this area was conducted by Mondal et al., where workload is shifted from heavily loaded servers to lightly loaded servers in shared nothing environments [63]. The master server in their scheme controls all other second level servers and maintains information about the second level index. The second level index servers periodically send messages concerning their load status to the master server so that the master server can migrate data from heavily loaded servers. In a wide area network, dynamic data migration may not be useful, especially when the size of data is very large. In the scientific applications we target, we cannot migrate the data chunks, which are typically small subarrays of large multidimensional arrays. In general, solutions to this problem are influenced by application and workload characteristics, and substantially more experimental characterization is required.

2.4 Indexing Services in the Computational Grid

OGSA-DAI (Open Grid Service Architecture-Database Access and Integration) is Grid middleware that allows data resources, such as relational or XML databases, to be accessed via web services [6]. OGSA-DQP (Open Grid Services Architecture-Distributed Query Processor) [6] is an extension of OGSA-DAI that provides a service-based distributed query processor. OGSA-DQP allows queries to be evaluated over multiple distributed relational databases, where the queries may include calls to web services. However, most of scientific datasets are not stored in relational databases for performance reasons and data size. Instead, several self-describing scientific data formats have been

developed for scientific datasets, such as HDF4, HDF5, and netCDF [70, 71, 82]. Thus, our work is different from OGSA-DQP in that our target applications are data intensive scientific data analysis applications, but we believe OGSA-DQP can adopt our indexing schemes into its framework so that it can support scientific datasets.

Federated MCAT (metadata catalog), also known as SRB zones, is a distributed resource discovery service for Storage Resource Broker. A single MCAT service has been known as a serious performance bottleneck. SRB zones improve MCAT performance in wide area network, by allowing various configurations in distributed environment, including replicated catalog. Although MCAT service supports dataset name, data location, and access control list, it does not support multidimensional range query into the contents of datasets, which is our main concern in this work.

Recently Zhang et al. compared the scalability performance of some MIS (Monitoring and Information Systems) systems, MDS2, R-GMA, and Hawkeye [110]. MDS is a Globus Toolkit monitoring and discovery service, R-GMA is a European DataGrid Relational Grid Monitoring Architecture, and Hawkeye is a monitoring and management system that uses Condor. MIS systems have commonality with our distributed indexing schemes in that they enable resource discovery. But none of them support multidimensional range queries as in OGSAI-DAI and MCAT.

Chapter 3

Case Study Applications

In most of our work, we deployed real software tools and employed real applications and datasets in order to make a comprehensive experimental performance characterization. I chose representative applications from different domains, namely, satellite data processing and computer vision.

3.1 Satellite Data Processing: Kronos

”Remote Sensing is the technology that is now the principal modus operandi (tool) by which (as targets or objects of surveillance) the Earth’s surface and atmosphere, the planets, and the entire Universe are being observed, measured, and interpreted from such vantage points as the terrestrial surface, earth-orbit, and outer space.” [96].

Among many remote satellite sensors, AVHRR (Advanced Very High Resolution Radiometer) has been widely used for a long time since the first 4-channel radiometer was carried on TIROS-N in 1978. It has been improved since then and now the latest version with 6 channels was carried on NOAA-18 launched in 2005 [73, 100]. The satellites orbiting the Earth gather remotely sensed AVHRR GAC (Global Area Coverage) level 1B datasets [69], stored as a set of arrays. Satellite datasets include geo-location

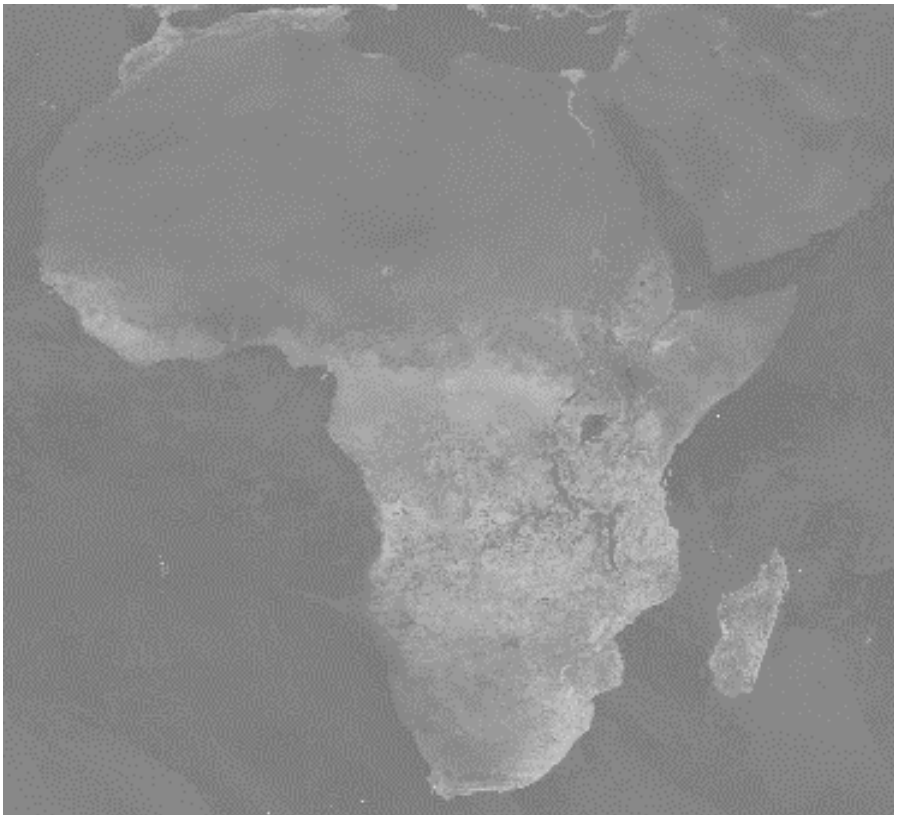


Figure 3.1: A Kronos data product. A 7-day (January 1-7, 1992) composite using Maximum NDVI (normalized difference vegetation index) as the compositing criteria and Rayleigh/Ozone as the atmospheric correction method. (Courtesy of H. Andrade)

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, as well as 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 [111] 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 very high rate, with the volume of data for a single day about 1GB. The dataset used for our experiments in this thesis was collected over one month (January 1992), and has a total size of more than 30GB. An AVHRR GAC dataset consists of a set of Instantaneous Field of View (IFOV) records, angular cones of visibility of the sensor, which corresponds to the surface on the Earth [72]. Each sensor reading is associated with a position (longitude and latitude) and the time the reading was recorded. Additionally, resolution information is stored with the raw data.

3.2 Volumetric Reconstruction

The *multi-perspective vision studio* is a volumetric reconstruction application used for *multi-perspective* imaging. In an environment where multiple cameras are used for simultaneously shooting scenes from various perspectives, more views can deliver more information about the scene and potentially allow recovery of interesting 3-dimensional features with high accuracy and minimal intrusion into the scene [20, 21]. The cameras shoot a scene over a period of time (a sequence of *frames*) from multiple perspectives and post-processing algorithms are used to develop volumetric representations. Multiple video streams generate very large amounts of image data that can become unmanageable unless there is an efficient way to store, catalog and process them.

The Keck Lab at the University of Maryland [20] is a multi-perspective imaging laboratory, consisting of 64 digital gray-scale cameras that synchronously capture video streams at frame rates up to 85 fps(frames per second). The resolution of the images is 648x484 gray scale with a depth of 8 bits per pixel. The size of one minute uncompressed multi-perspective video is approximately 100GB. From the captured images, a

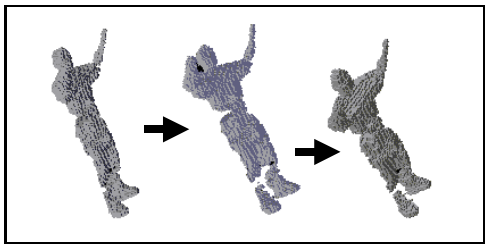


Figure 3.2: View of a volume from one perspective over 3 frames. (Courtesy of H. Andrade)

single frame 3D volume is reconstructed as an occupancy map encoded with an octree representation [86].

Users interact with the application by submitting queries. A query computes a set of volumetric representations of objects that fall inside a 3-dimensional box – one per frame – using a subset of the available cameras. The query result is a reconstruction of the foreground objects lying within the multidimensional query region (a pre-processing step removes background objects from the stored images, producing *silhouettes*). The reconstructed volume for a frame, i.e., the query result, is represented by an octree, which is computed to a requested depth d. Deeper octrees represent the resulting volume at higher resolutions.

A Volumetric Reconstruction query request specifies a dataset name, 3D region of interest within the dataset, a frame range with a constant stride, a depth (number of edges from the root to the leaf nodes) of the octree to represent the volume, which specifies the resolution of the reconstruction, and a set of cameras to use. The 3D OpenGL client renders the 3D volume of a frame at any time step, and a user can browse the sequence of 3D volumes, as shown in Figure 3.2. The client allows volume to be viewed from any point of view via mouse dragging.

3.3 Synthetic Query Workload Generator

In order to create workloads of range queries, we employed a variation of the *Customer Behavior Model Graph (CBMG)* technique to make query workloads realistic. CBMG has also been utilized by researchers analyzing performance of e-business applications and website capacity planning [61]. A CBMG can be characterized by an $n \times n$ matrix of transition probabilities between the n states, $P = [p_{i,j}]$, where each state 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. Examples of state transitions for Kronos include re-scaling a data product and panning in either direction through a dimension (e.g., in space or time).

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.

Chapter 4

Data Chunking

4.1 Overview

Data chunking logically partitions a dataset into coarse-grained blocks to reduce disk access time when accessing large amounts of data in a file. Most self describing scientific data formats store data as multidimensional arrays, to ease access from within scientific programs. Scientific applications access multidimensional arrays with various access patterns. Some applications read sub-arrays in row major order, or in column major order. Others read sub-arrays specified as regular sections [43]. Scientific data format libraries support reading sub-arrays with various access patterns, but most of them do not show good I/O performance along every dimension. Only a few libraries, which support data chunking, achieve similar performance for any kind of access pattern. For datasets consisting of data arrays, each data chunk can be viewed as a contiguous sub-array within the dataset. The order of data accesses into a multidimensional array critically affects the I/O performance. To achieve maximum I/O performance by minimizing disk seek operations, each chunk should be a single contiguous sequence in the file. We use the term *physical chunk* to refer to a sub-array that is a physically contiguous single sequence within a file on disk. Depending on the data access pattern,

physical chunking can provide much higher I/O performance than other data organizations [99]. A *logical chunk*, on the other hand, is a conceptual partitioning of a dataset on disk. A multidimensional dataset can be partitioned into logical chunks whether it is a single contiguous array or a physically chunked array. When a dataset is stored as a single array on disk, disk seek operations are required to access each row of a logical chunk. On the other hand, when a dataset is partitioned and ordered as physical chunks, the layout of the physical chunking can also be viewed as the logical chunking. However, logical chunking does not necessarily have to use the same partition as physical chunking, (i.e. a logical chunk in a physically chunked dataset can contain several physical chunks, and could even be a subset of a physical chunk). Logical chunking by itself does not improve I/O performance, but is necessary to create an index into the data.

4.2 Scientific Data Format: HDF

In order to help in navigating through large scientific datasets, many *self-describing scientific data file formats* have been developed such as Planetary Data System (PDS) [2], Network Common Data Format (NetCDF) [82], and Hierarchical Data Format (HDF4 and HDF5) [70, 33]. Self-describing data formats contain structural metadata that is used by a corresponding runtime library to navigate through the file to improve I/O performance, by allowing for direct access (once the metadata is read) to particular datasets within a file, or to parts of the dataset. Files in these self-describing formats may also contain application-specific metadata, which provides semantic information about the contents of the file [33]. The contents of scientific data files typically are a collection of multidimensional arrays, which we will refer to as datasets, along with the corresponding metadata.

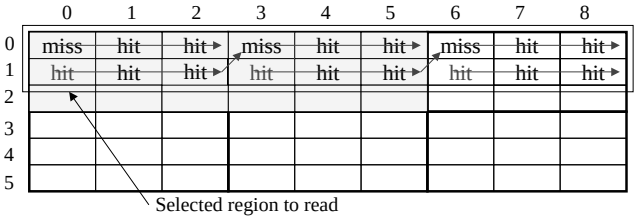
Hierarchical Data Format (HDF) is a self-describing scientific data file format and runtime library developed at the National Center for Supercomputing Applications (NCSA) to store and serve heterogeneous scientific data. A file stored in HDF contains supporting metadata that describes the contents of the file in detail, including information for each multidimensional array stored, such as the file offset, array size and the data type of array elements. HDF also allows application-specific metadata to be stored. Thus, the metadata within a file make HDF an essentially machine independent format. The most recent version of HDF is HDF5. Although HDF5 was designed to overcome some deficiencies of the older HDF4, HDF5 has a totally different internal representation of data objects from previous HDF versions.

4.2.1 Data chunking in HDF

To improve I/O performance, HDF supports two different storage layouts. The default storage layout is a contiguous layout, in which the elements of a multidimensional array are stored in either row-major order or column-major order. The second choice is a chunked layout, in which data is stored as physical chunks, small coarse-grained blocks of the sub-array, with each chunk stored in row-major or column-major order.

In HDF5, a chunked layout has several advantages over a contiguous layout. In particular, a chunked storage layout allows extending the size of a stored multidimensional array in any dimension, not just the slowest varying array dimension (outermost in row-major order, innermost in column-major order). In addition, disk space for a chunk does not have to be allocated on disk until data is written into that chunk, which can decrease disk storage requirements. HDF4, on the other hand, provides only some of the advantages of a chunked layout. In HDF4, extending the size of an array dimension is allowed only for the slowest varying dimension, but not for any other dimensions.

(a) H5Dread. A chunked layout can cause unnecessary cache misses.



(b) H5Xread. The H5Xread function reads data elements in chunk order to minimize cache misses.

Figure 4.1: *The ordering problem for H5Dread with a chunked layout*

In accessing a subset of a large dataset, data chunking reduces expensive disk seek times and improves overall I/O performance by taking advantage of spatial locality in any array dimension [99]. On the other hand, the contiguous storage layout can exploit spatial locality only in the dimension that varies fastest in storage order.

However, a chunked layout does not always provide better performance than a contiguous layout. One case in which data chunking may hurt I/O performance occurs when the size of a chunk is very large and the region selected to read is smaller than the size of a chunk, causing unnecessary data to be read from disk, since disk I/O is always done in units of complete chunks.

4.2.2 Potential problems with HDF data chunking

Both the HDF4 and HDF5 libraries cache data in a *data chunk cache* to improve I/O performance. However, the functions that read datasets in both libraries are designed as if the size of the data chunk cache is infinite, potentially causing significant performance problems. Because the read functions in the HDF libraries read arrays in row major (or column major) order, whether the array has a chunked layout or contiguous layout, that ordering does not match the ordering of data with a chunked storage layout, potentially leading to many data chunk cache misses.

Suppose we want to read two rows of a dataset stored with a chunked layout. The standard HDF library read function, H5Dread, reads the data in row major order, as shown in Figure 4.1(a). When the first row of the array is read, all array elements in the chunks that contain the first row are cached in the data chunk cache, along with the rest of the chunks. When the next row is read, the library searches in the cache, but will not find the chunk needed, because the default chunk cache size is 1MB. If the total size of the chunks that contain one row of a dataset is greater than 1MB, the data chunk cache will not be able to hold all the chunks and will evict the chunks in the cache using an LRU replacement policy. Therefore when the library reads the second row of array elements, the first element in the second row will not be found in the cache, as shown in Figure 4.1(a). So the chunk containing that array element must be read from disk again, and the same problem will occur for all other data chunks both in that row and in subsequent rows. The HDF library developers have recognized this problem, and warn of severe performance penalties in the HDF User's Guide [70]. Their solution to the problem is to add a function to the HDF5 API that increases the size of the data chunk cache, placing the burden of selecting the appropriate data chunk cache size on the application developer. We now propose another solution.

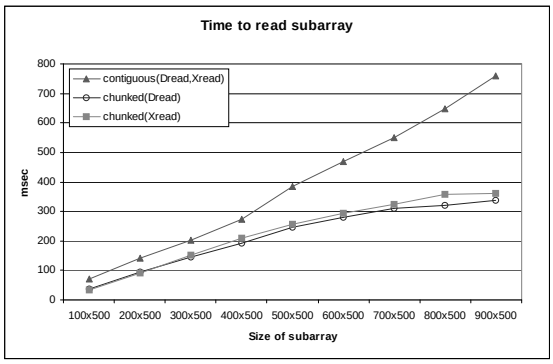
4.2.3 H5Xread

We have added new functionality to the HDF5 library, in the form of a function called H5Xread with the same interface as H5Dread, to read multidimensional array datasets from disk in the same order they are stored with a chunked storage layout. Such a strategy avoids unnecessary cache misses and reading the same chunk from disk multiple times. After chunks are retrieved from disk, they are reorganized in memory to produce the desired contiguous array layout. For arrays stored with a contiguous layout, H5Xread reads the data from disk in the same order as H5Dread. Figure 4.1 shows the difference in data accesses between the H5Xread and H5Dread functions. The array read function in the HDF4 library has the same performance problem as H5Dread, and the same functionality as in H5Xread can be implemented for that library.

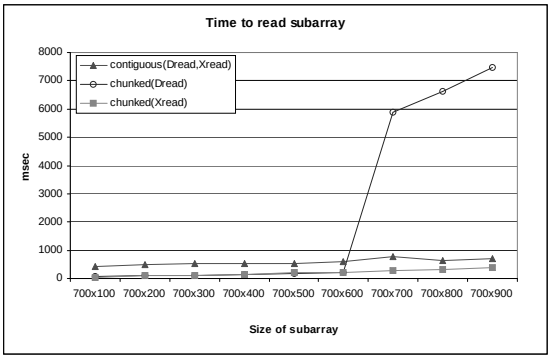
Performance evaluation of H5Xread

We now present the results of a performance evaluation of the standard HDF5 dataset read function, H5Dread, with our H5Xread function, for chunked storage layouts. In the experiment, we partitioned a two-dimensional 64MB AVHRR HDF5 dataset, containing an array of 4000x1000 elements, each of which is 16 bytes. The array was partitioned into 160 KB logical chunks, each of which contains 100x100 elements. For the chunked layout, we made the physical chunk size the same as the logical chunk size. The experiments were run on a SunBlade 100 workstation with a 500MHz Sparcv9 processor, 256MB memory, and a 7200RPM IDE disk with a seek time of 9ms.

Figure 4.2 shows the time to read two different shaped subarrays from the dataset. We measured the wall clock time, varying the number of rows read in in Figure 4.2(a), and varying the number of columns read in Figure 4.2(b). Figure 4.2(a) shows that the chunked storage layout provides better I/O performance than the contiguous layout



(a) The selected region has a fixed number of columns, and the number of rows increases.



(b) The selected region has a fixed number of rows, and the number of columns increases.

Figure 4.2: *Time to read selected regions of the dataset*

in most cases. The performance gap between the chunked layout and the contiguous layout increases as the number of rows increases. This is because as the size of a column grows, even more disk seek operations are needed for the contiguous layout than for the chunked layout.

For these experiments, we used the HDF library default sized data chunk cache of 1MB, so the chunk cache holds six of the 160KB chunks. In Figure 4.2(b), when the number of columns in the selected subarray is less than or equal to 600, the H5Xread function shows similar performance to that of H5Dread for a chunked layout, but as the number of columns increases, so that the size of each row increases, the cache fills up before reading an entire row - in this experiment when the row size reaches 700 elements, at which point the H5Dread function suffers from many cache misses, while H5Xread continues to provide stable I/O performance. The performance difference can be a large factor, here up to a factor of 9, as is seen in the right side of the figure.

The H5Xread functionality also provides stable performance characteristics for higher dimensional datasets. We have evaluated performance for three-dimensional datasets, and the results are essentially the same as those for the two-dimensional experiments, meaning that the H5Xread function with a chunked layout provides better performance than H5Dread with either a chunked or a contiguous layout.

4.3 Indexing with Data Chunking

A large number of indexing techniques have been proposed to improve the performance of range queries and nearest neighbor queries for multidimensional datasets. Techniques for speeding up searches into high-dimensional datasets have been researched extensively [19, 26]. The most common multidimensional indexing structure, the R-tree, is

a height-balanced tree similar to the well-known B-tree [40]. When point data is inserted into a leaf node of an R-tree, the minimum bounding boxes of the internal nodes are enlarged to cover the child nodes, sometimes requiring that internal nodes be split to maintain the balance criteria. For a given multidimensional range query, a search into an R-tree traverses all nodes in the tree with minimum bounding boxes that overlap the range. The R*-tree is an optimized R-tree extension that minimizes overlap of nodes [13].

The goal of using a spatial index is to avoid searching all the elements in a multidimensional dataset to perform a spatial range query. If the dataset is partitioned into coarse-grained chunks, and the bounding box for each chunk (i.e. the minimum and maximum values for each dimension) is placed in an index structure, not all elements within the dataset must be searched, but only elements in the chunks with bounding boxes that overlap the query range. This effectively reduces the amount of data retrieved from disk, and should improve query response time. The performance comparison of various indexing structures for data chunking will be discussed shortly in Chapter 5.

4.3.1 GMIL: Generic Multidimensional Indexing Library

We have designed and implemented a generic indexing library for various multidimensional scientific data formats using an R*-tree. The R*-tree provides better performance and storage utilization than an R-tree, especially for high-dimensional data. Figure 4.3 shows the design of the indexing library. A new multidimensional scientific data format can utilize the services of the indexing library by implementing three functions that (1) create an index file, (2) search the index file for a range query, and (3) read a subset of the dataset using the information returned from searching the index. The generic indexing library provides an API for these functions. In order to read data files in a specific

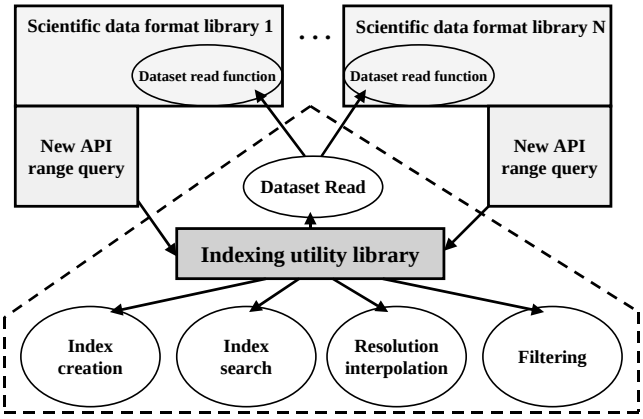


Figure 4.3: *Generic spatial indexing library*

scientific data format, the indexing library read function must call a read function from the particular scientific data format library. The name of the read function, and some additional information about various parameters, must be obtained from the scientific data format library.

The generic indexing library has an index creation module, an index searching module, a resolution interpolation module, and a filtering module. Multi-dimensional datasets, in particular ones with spatial and/or temporal dimensions, may contain data elements at different granularities. For example, multiple sensors on the same orbiting satellite may have different resolutions. Hence some sensor datasets may have arrays that are several times larger than the corresponding geographic datasets that allow for determining the spatio-temporal locations of the data elements. The generic indexing library addresses this problem by providing an interpolation mechanism. The last function that the indexing utility library provides is data filtering. Because data is stored as chunks, a range query can return all the chunks that overlap the given range query. However, not all data elements in those chunks will overlap the query range, so the library

supports data filtering to return only those data elements that fall within the query range. If the application can accept extra elements (i.e. perform its own filtering), the library can also return the unfiltered chunks.

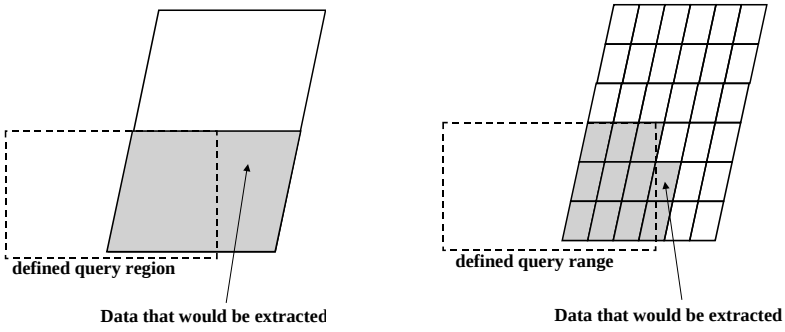
4.3.2 Case study: HDF-EOS vs GMIL

NASA's Earth Observing System Data and Information System (EOSDIS) is a system that acquires, stores, and distributes sensor data acquired from orbiting satellites. HDF was selected as the standard data format by the EOSDIS project, and a metadata schema was specified to store Earth Observing System (EOS) data [56]. In addition, a library was implemented on top of the HDF library, called HDF-EOS, to extend the capabilities of the HDF library to allow for the construction of special data structures, called *grids*, *swaths*, and *points* [56]. We focus on swaths, because that is the way most HDF-EOS data is stored [56, 81]. A grid structure is produced using a projection operation via a given mathematical transformation between the rows and columns of an array and the latitude/longitude information stored with the EOS data, and is used to store the results of such projection operations. The latitude/longitude information for each array element can be computed based on the array offset using map projections such as Mercator or Goode. A point structure is a table that contains data records taken at irregular time intervals and across scattered geographic locations. A swath structure is based on the way a typical satellite sensor acquires data, whereby an instrument takes a series of scans perpendicular to the ground track of the satellite as it moves along that ground track.

The HDF-EOS library has versions both for HDF4 and HDF5, called HDF-EOS4 and HDF-EOS5. Despite HDF4 and HDF5 being quite different data formats, the HDF-EOS4 and HDF-EOS5 libraries have essentially the same basic features for the HDF-

EOS data structures. Both versions of the HDF-EOS library allow a user to specify a range query, by specifying the data to retrieve as a box in latitude and longitude. Once a query region is defined, by the *defboxregion()* function, the user reads the data from that query region with an *extractregion*() function.

In an HDF-EOS swath structure, the latitude, longitude, and temporal information for the dataset is stored as separate arrays from the sensor value arrays. To retrieve the geographic information for a data element in a sensor value array, the elements in the geographic datasets that have the same offsets as the sensor element must be retrieved. The HDF-EOS library does not support spatial indexing structures. To read the sensor values that fall within a query range, the *defboxregion* function must scan every geographic dataset to obtain the location(s) of the region within the file, because the geographic information for the EOS datasets is not evenly distributed through the spatial domain (i.e. it has *spatial irregularity*) [95]. Once a region is defined with the *defboxregion* function, the corresponding *extractregion* function can be called to read the desired sensor data from the file. It is an expensive operation to scan all elements in a geographic dataset, so HDF-EOS provides several approximation options. First, an application can retrieve the set of scanlines that have any single element that overlaps the query range. In this *any-point* mode, all geographic data must still be searched. Second, if the mid-point of a scanline overlaps the query range, that scanline can be read in *mid-point* mode. In this mode, the *defboxregion* function reads only one column of the geographic dataset (the one for the middle element in the scanline). Finally, if both end points of a scanline overlap the query range, the entire scanline will be read in *end-point* mode. Mid-point and end-point selection are much faster than any-point selection, but there is a tradeoff between response time and accuracy in retrieving the desired data.



(a) EOS range query (b) Range query with data chunking

Figure 4.4: *Data read for a range query*

For our indexing library, creating the index requires reading all the geographic information for a swath to obtain minimum and maximum location (latitude/longitude) values for each logical/physical chunk. It is less expensive to build an index for a chunked storage layout than for a contiguous storage layout that uses logical chunks for the index, because reading each logical chunk from disk may require multiple disk seeks.

For reading subsets of a dataset using the indexing library, all elements in chunks that intersect the query range are read, while the HDF-EOS library returns all elements in any scanline that overlaps the query range. Therefore the number of elements read by the two libraries may be different. The range query functions return the query result in the form of a one dimensional array of data elements, but with EOS data each element in the array is associated with two-dimensional geographic coordinate information (latitude and longitude). However, some of the returned elements may not be in the query range, but the application cannot determine which elements should be discarded without the geographic coordinate information. Therefore the range query function in the indexing library returns the geographic information corresponding to each sensor value. Using this geographic information, applications can filter out sensor values that do not overlap

the query range. If the chunk size is large, the R*-tree search may end up reading more unnecessary data than the HDF-EOS *extractregion* function, but it is much more likely that the HDF-EOS function will read more unnecessary data.

Performance evaluation of GMIL

We evaluate performance for reading HDF-EOS data via range queries. We have implemented versions of the HDF-EOS4 and HDF-EOS5 range query APIs that call the indexing library. The test datasets range in size from 16MB with 30 chunks, to 128MB with 800 chunks. In our experiments we used H5Xread, to read data from the file, instead of the HDF library H5Dread, since it provides better performance.

In the experiments, we have measured both the time to create an R*-tree index file for various numbers of chunks, and the time to perform range queries using the index. For range query performance, we have measured the time to read a subarray for three different shapes of the selected region within a two-dimensional array. The first query selects a region that spans many columns, but relatively few rows. For this kind of query, the HDF-EOS defboxregion function reads the data that exactly matches the query range in any-point or mid-point mode. However, our indexing library read function may read extra elements that are not in query range, but are in chunks that overlap the query range. The second query selects a mostly square region from the 2D array. For this case, the HDF-EOS functions will usually read a much larger number of elements than our indexing library function. The third query selects a region that spans many rows, but relatively few columns. For the second and third queries, the HDF-EOS library reads many more elements than our indexing library does, since HDF-EOS reads all the elements in any scanline (row) that overlaps the query region,

All the results presented measure elapsed wall clock time. The size of the test dataset

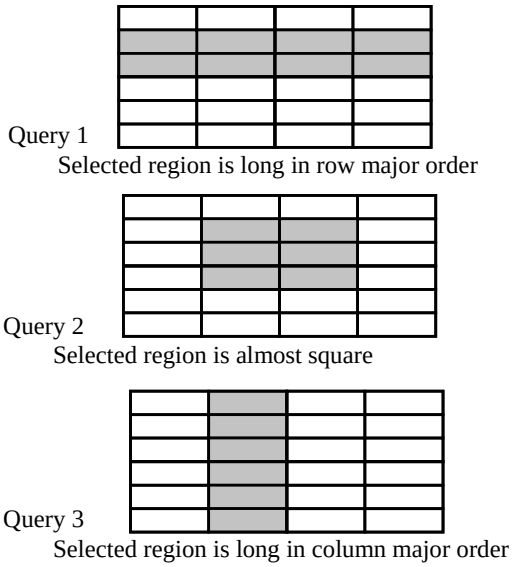


Figure 4.5: *Three types of range query*

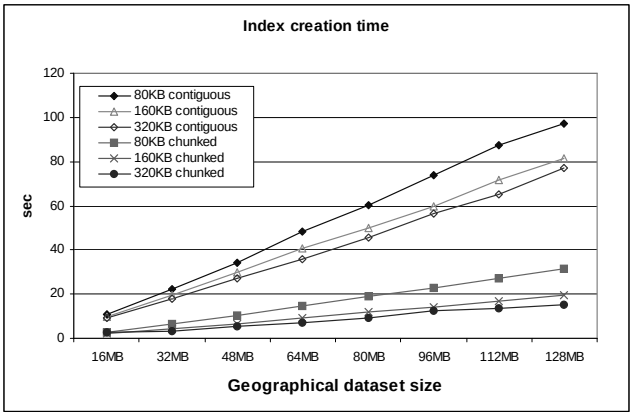
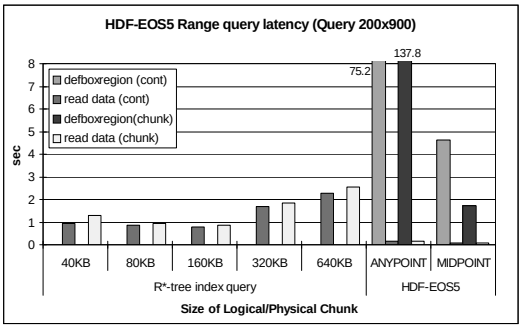


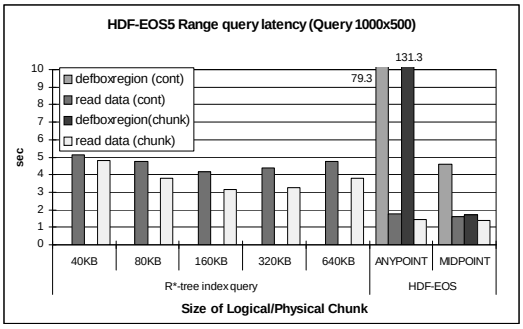
Figure 4.6: *Time to create an index file, varying the chunk size and the data layout*

for measuring range query performance is 4000x1000 elements, with each logical or physical chunk containing 100x100 elements of type double, for a total of 80KB per chunk. For measuring R*-tree index creation time, we created logical and physical chunk sizes of 0.8KB, 80KB, 160KB, and 320KB. Because the HDF-EOS4 library does not support data chunking, we measured performance only with a contiguous storage layout, and partitioned the arrays into logical chunks for indexing. The number of array elements requested for the first query is 200x900, for the second query 1000x500, and for the third query 2000x200. We ran the experiments on the same SunBlade 100 used for the data chunking experiments.

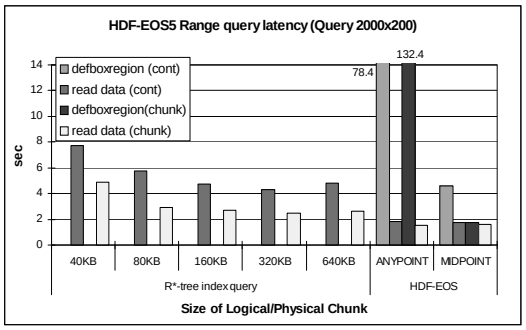
Figure 4.6 shows the time to create the R*-tree index file for various dataset and chunk sizes. The figure shows that the time to create the index depends linearly on the number of chunks, which is determined by the chunk size for a fixed size dataset. The question to answer then, is what is the best chunk size? There is a tradeoff between index creation time and disk access time for range queries. When the chunk size is small, the number of chunks is large and it takes a long time to create the index, as seen in Figure 4.6. On the other hand, a small chunk size will causes range queries to read less extra data. The most important decision criterion is that the index will be used for all searches, but once an index file is created it will not be changed unless the dataset is updated. Although the index is not likely to change often, the time to create the index file should not be ignored. For example, when the number of chunks becomes very large, for example 50,000, it takes several hours to create the index file on the experimental machine. Most of time to create the index file is spent building the R*-tree, performing operations to maintain the desired tree properties. Also, as shown in Figure 4.6, it is faster to create the index for a chunked layout compared to a contiguous layout, because reading the geographic dataset is more expensive for the logical chunks



(a) 200x900 query



(b) 1000x500 query



(c) 2000x200 query

Figure 4.7: *Time for range queries with HDF-EOS5*

in the contiguous layout.

For the experimental dataset, and for 800 chunks, the R*-tree library[1] created a 73KB index file, while the size of the dataset is 128MB. Also, an HDF file can contain several swath structures, each with its own latitude, longitude and time information, and a swath can contain several multidimensional datasets with sensor values. An index is therefore needed for each swath, not for every dataset. Therefore, the index file does not require a significant amount of disk storage compared to the size of the dataset it is

[1]We employ the HnRStar library, version 1.0 [48]

indexing.

Because the performance results were very similar for both the HDF-EOS5 and HDF-EOS4 libraries, we only show results for the HDF-EOS5 library. Figure 4.7 shows the time to read a subset of the dataset for the three queries, using both the indexing library range query function and the HDF-EOS5 standard range query functions. The time for the indexing library includes both searching into the R*-tree and reading the geographic and sensor value data from disk. As we described earlier, the HDF-EOS library has two separate functions to perform a range query, so there are two bars in the graph for each data layout (contiguous and chunked).

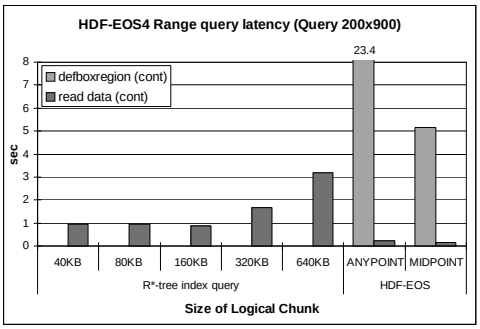
For a single query, the extractregion functions in HDF-EOS5 and HDF-EOS4 read only a subset of the sensor value dataset. But the corresponding defboxregion functions read every element in the geographic datasets to determine the file location information for the requested region in *any-point* mode, and read either one or two columns of the geographic data in *mid-point* mode or *end-point* mode, respectively. For the three queries in the experiments, the HDF-EOS defboxregion function returns an empty region in *end-point* mode, so no results are shown.

The indexing library range query function reads the R*-tree index file (if the index has not already been read into memory), and the chunks of the sensor value and geographic dataset returned by the R*-tree search. The geographic data can be used to filter the sensor data that is returned, but does not lie in the query range. As seen in Figure 4.7, the time to perform the extractregion operation in the HDF library is less than the indexing library query time in most cases, but that is because the extractregion function only reads data from the sensor value dataset, and does not read the geographic information. The location information to determine which sensor values to read is computed by the defboxregion function, and when we look at the time to execute that function, we see

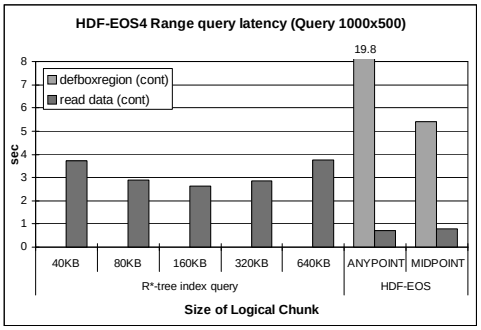
that using the index library to perform the range query provides enormous performance benefits. Comparing the time to read the data in HDF-EOS5 *any-point* mode to that of the indexing library, for all queries the indexing library time was less than 7% of the defboxregion time. If the HDF library is used to select a region in *mid-point* mode, the performance is about the same or somewhat worse than that of the indexing library, but the indexing library should return a better approximation to the data that actually falls within the query range. Also, the indexing library is guaranteed to return all data elements that fall within the query region, but the HDF-EOS library in mid-point mode will not return a scanline with a mid-point element that does not fall within the query range.

As Figure 4.7(a) shows, the performance of the indexing library for reading a region with many columns and relatively few rows decreases as the chunk size grows, because the indexing library range query function reads more unneeded data from disk. For this kind of query, the contiguous layout performs best because it does not cause many disk seeks, so gives about the same or even slightly better I/O performance than a chunked layout. The defboxregion function reads the geographic dataset one scanline (row) at a time, and that is very inefficient, since it will cause many disk seek operations to read each scanline. Therefore defining a region takes much longer with a chunked layout than a contiguous layout for this type of range query. For this type of query, the number of extracted elements is the same for both the indexing library range query function the and the HDF-EOS query function.

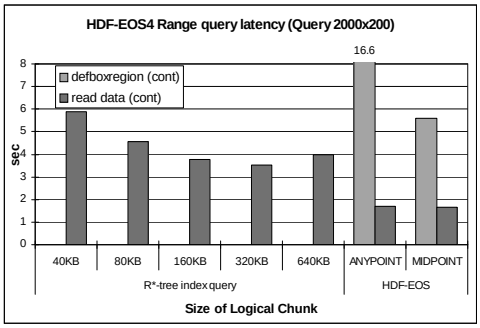
We see from Figure 4.7(b) that for the second query that covers a mostly square region, the performance of the HDF-EOS extractregion function is worse than for the first query with many columns and few rows, because extractregion reads the entire scanline for every one that overlaps the query range, not just the elements in the query range.



(a) Query 1. Many columns, few rows (200x900)



(b) Query 2. Mostly square region (1000x500)



(c) Query 3. Many rows, few columns (2000x200)

Figure 4.8: *Time for range query with HDF-EOS4*

For the third query with many rows and few columns, we see from Figure 4.7(c) that as the chunk size grows the time to read data for the indexing library decreases, because of fewer disk seek operations. In the best case, even though the indexing library function must also read the geographic dataset, which is done by the defboxregion function in the HDF-EOS library, the indexing library function takes about the same time as extractregion for a chunked layout. This is because extractregion reads a large amount of unneeded data, as was the case for the second query. For the third query, the amount of unneeded data read by extractregion is even larger than for the second query.

Even though the amount of unneeded data read by the indexing library is usually less than for the HDF-EOS library, it is still necessary to filter the unneeded data. When the size of the chunks is small, filtering is not expensive, but the R*-tree search time will be long because of the large number of leaf nodes in the tree. However, R*-tree search time is very small compared to the time to read the datasets from disk.

In our experiments, as the chunk size grows larger, performance decreases because the indexing library reads extra data that is outside of the query range. And if the chunk size is too small, performance also decreases because of additional disk seeks. However, overall the indexing library shows much higher performance than HDF-EOS *any-point* mode, and better performance than *mid-point* mode for many queries, despite the indexing library performing the filtering needed to remove unnecessary data using geographic information, which is not provided by the HDF-EOS library.

4.4 Summary

We have shown that I/O performance can be improved with the use of both multidimensional indexing structures and data chunking, for navigating through multi-dimensional

self-describing scientific datasets. Indexing individual data elements of large scientific datasets makes the size of index even larger than input dataset, thus it would lead to poor performance. Due to the way of storing datasets from sensor devices, most scientific datasets have spatio-temporal locality, whereby we can group data elements and store a single bounding box for each chunk in order to reduce the index size for better performance.

Our generic indexing tool targets scientific data formats such as netCDF, HDF, and SILO, which contain structural metadata. Data stored in such self-describing formats may be easily accessed across heterogeneous platforms using the runtime library API for each format. Data stored in these formats contain application-specific semantic information about the contents of the file, so that no other information is necessary to interpret the data. Experimental results, on NASA Earth observing satellite datasets, have shown that the generic scientific indexing library greatly improves the performance of range queries, as compared to using the format-specific runtime libraries.

Chapter 5

Indexing Structures for Scientific Datasets

In this chapter, we discuss a few widely used multidimensional indexing tree structures, concentrating on issues related to performance for indexing chunked scientific datasets. As we have shown in Chapter 4, many scientific libraries perform a brute force range query operation inefficiently, but multidimensional indexing structures allow performing range queries efficiently.

In the past couple of decades, extensive research has been carried out on multidimensional indexing structures, to enable efficient range queries and nearest neighbor searches. However, most of the recent studies have focused on high-dimensional feature-based similarity searches into a relatively small number of point data items.

Many scientific instruments, ranging from sensors on Earth orbiting satellites to light microscopes, can produce hundred of gigabytes of spatio-temporal daily, consisting of billions of individual data elements. Storing each data element in a huge scientific dataset into a multidimensional indexing tree is impractical, because the size of the index could be even larger than the raw dataset, and the performance of queries would be poor due to the size of the index. The data chunking optimization described in Chapter 4 solves this performance problem.

In this chapter, we focus on the problem of indexing rectangular objects (multidi-

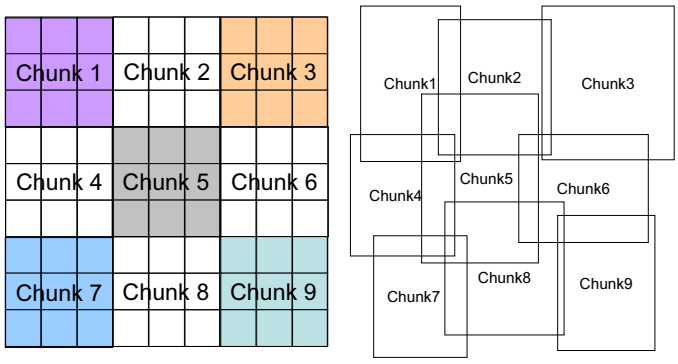


Figure 5.1: *Dataset with nine chunks and corresponding bounding boxes in problem space*

mensional bounding boxes), introducing the Spatial Hybrid tree (SH-tree), an extension of the Hybrid-tree [22], and perform a comparative study of SH-trees against other indexing techniques for multidimensional rectangular datasets.

5.1 Spatial Indexing Structures for Scientifi c Datasets

Multidimensional indexing trees can be classified into two categories: *space partitioning methods* and *data partitioning methods*. Data partitioning methods such as R-trees split an overflow node by grouping its child nodes into two sub-sets. However space partitioning methods such as KDB-tree split an overflow node by partitioning its data space into two sub-spaces. This classification is also based on the data structures of internal tree nodes. In space partitioning methods, the internal tree node is represented by a binary KD-tree (i.e., split dimensions and split positions.) Each leaf node of the binary KD-tree points to a child tree node. However, in data partitioning methods, the internal tree node is represented by a list of bounding boxes of child nodes. Therefore, the number of fan-outs in data partitioning methods is dimension dependent, while it is dimension independent in space partitioning methods.

5.1.1 Space Partitioning Methods

KDB-tree: Robinson has developed a balanced B-tree version of the binary KD-tree [14], the KDB-tree [83]. Unfortunately, minimum node utilization is not guaranteed for KDB-trees because of the *downward cascading split* problem. A KDB-tree does not allow overlapping partitions, as does the standard KD-tree, but when a tree node must be split it is not always possible to find disjoint partitions in a KDB-tree. In such cases, some sub-partitions must be split at the same split value as for the parent node, even if the sub-partitions do not meet the minimum storage utilization requirement for a node. The split can propagate all the way down to the leaf nodes, which can make range query performance poor.

Spatial KD-tree: A Spatial KD-tree (SKD-tree) [74] is another variant of the binary KD-tree designed for non-point spatial objects. An SKD-tree allows sub-partitions to overlap, by having two split positions in one split dimension. Each split position represents the boundary of the lower or upper sub-region, respectively. However, the SKD-tree is a memory-based, not a disk based data structure, which means that it is a binary tree that does not consider disk page size unlike B-tree. Hence it is not suitable for very large databases.

Object duplication methods: Matsuyama's KD-tree [60] is another variant of the binary KD-tree for non-point spatial data. In Matsuyama's KD-tree, an extensive object duplication strategy is used, hence objects can be stored in multiple leaf nodes. The R+-tree [93] is a disk based indexing method that uses the object duplication strategy. However, object duplication methods may create infinite recursive loops when inserting rectangles into the tree, if there is at least one non-point region, denoted as a *hot spot*, that falls completely inside all the child partitions of a node, as shown in Figure 5.2. In such a case, no matter what split dimension or split position is selected, either or both of

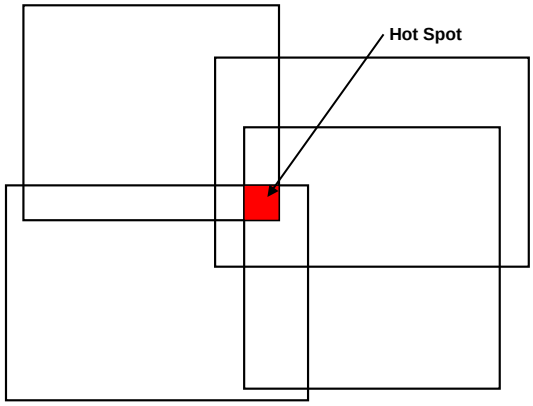


Figure 5.2: *Disjoint partitioning is not possible due to a hot spot*

the two resulting nodes will overflow again because the hot spot will belong to at least one of the resulting sub-partitions, so the resulting nodes must duplicate all the child bounding boxes that cover the hot spot. For this reason, disjoint partitioning methods based on the object duplication strategy are not feasible for non-point data.

Hybrid-tree: To solve the downward cascading split problem for KDB-trees, several variants have been proposed, such as hB-trees [59] and Hybrid-trees [22]. The Hybrid-tree solves the downward cascading split problem by allowing overlap of the two sub-regions after a node is split, as in data partitioning methods [22]. While the internal nodes for the data partitioning methods are lists of bounding boxes and pointers to child nodes, each internal node for the disk based space partitioning methods is a binary KD-tree, with each leaf of the KD-tree containing the sub-partition of a child internal node in the top-level tree and a pointer to the child node in the top-level tree. An internal node in a Hybrid-tree is also a binary KD-tree, whose nodes contain both a splitting dimension and two splitting positions in that dimension. By having two splitting positions instead of one, the Hybrid-tree allows overlapping regions when a downward cascading split is unavoidable. However, the Hybrid-tree allows overlap only in non-leaf nodes, and the overlapping region is created or extended only when a node overflows during

object insertion, so must be split. Therefore, non-point spatial objects cannot be indexed in a Hybrid-tree.

5.1.2 Data Partitioning Methods

R-tree and R*-tree: Instead of duplicating objects, spatial objects can be indexed by allowing overlapping regions, as in R-tree based index structures [40]. Although R-trees can be used for non-point data, a large amount of overlap between internal nodes in R-trees leads to search performance problems. To reduce overlapping regions for R-trees, Beckmann et al. proposed an optimized version of R-trees, called R*-trees [13]. The R*-tree insertion algorithm reinserts elements from a node that overflows, instead of splitting the node. This forced reinsertion feature of R*-trees improves search performance, but insertion can become very expensive.

X-tree: Berchtold et al. developed another variant of the R-tree, called an X-tree [15], which avoids highly overlapping bounding boxes via the use of *supernodes*. A supernode is a tree node that spans multiple pages on disk, thus has a larger capacity than a normal node. When a node must be split and a large amount of overlap between sub-partitions is unavoidable, the X-tree algorithm increases the capacity of the node instead of splitting it. If there would be a large amount of overlap between two nodes after a split, the probability that both nodes would be accessed by a search operation is high. Hence, sequential access to supernodes should be faster than random access to two separate nodes. However, supernodes have the overhead of additional disk management costs at index creation time. Therefore, before the X-tree insertion algorithm creates a supernode, it tries to find an overlap-free split based on past split history. For more details on supernodes, see [15]. However, split history is not useful for non-point spatial objects, because an overlap-free split is not always possible for non-point data. Even if

an overlap-free split can be found, in most cases it will not be acceptable since it will not meet minimum node utilization requirements.

5.2 Spatial-Hybrid Tree

Although the most important performance evaluation criteria often is searching the index, we can not ignore index creation performance since scientific applications can generate datasets very quickly. Most of the existing variants of R-trees sacrifice creation/insertion performance in order to make searches faster, which we want to avoid. Thus, we propose a new multidimensional indexing structure that performs fast index searches without a high cost for index creation. Also our new index structure needs to support rectangular bounding boxes, created from data chunking.

In this section we introduce the Spatial Hybrid-tree (SH-tree), a new multidimensional indexing structure that supports efficient range queries on non-point data objects, in both low and high dimensional spaces [66]. The SH-tree combines the properties of the SKD-tree and the Hybrid tree, both of which are based on space partitioning methods, and allows overlapping sub-regions by having two split positions in one split dimension. The SKD-tree allows overlapping sub-regions only when a mutually disjoint partition is not possible because of the volumes of the data objects, whereas the Hybrid-tree allows overlapping sub-regions when a downward cascading split is unavoidable [22]. In other words, the Hybrid-tree creates a new overlapping region when a node that overflows must be split, while the SKD-tree adjusts overlapping regions so that one region will fully contain a new object that is to be inserted. The SH-tree employs the node splitting algorithm of the Hybrid-tree and the insertion algorithm of the SKD-tree.

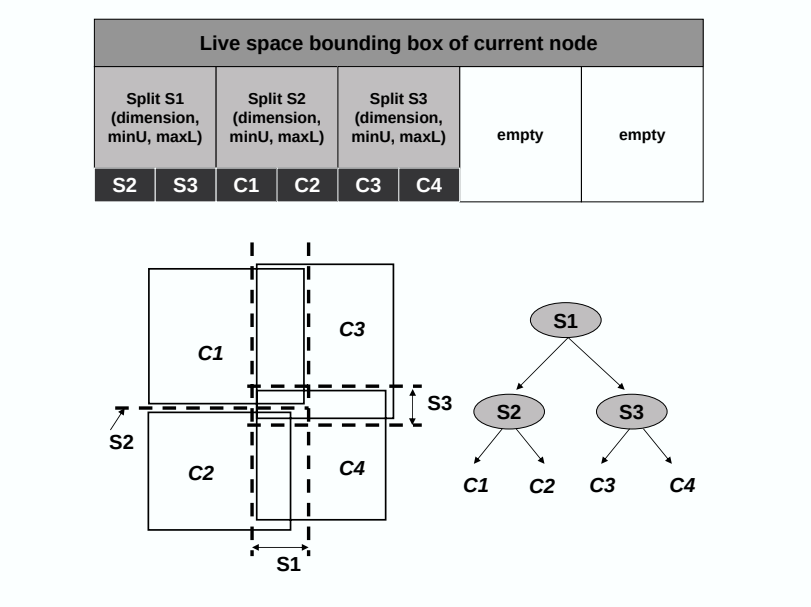


Figure 5.3: *KD-tree representation of an internal node of an SH-tree*

5.2.1 Insertion

Figure 5.3 depicts an internal node of an SH-tree. An internal node for a balanced space partitioning method such as a KDB-tree [83] is represented as a binary KD-tree, not a list of bounding rectangles as for R-trees. In SH-trees, one split dimension and two split positions are required for each child node in order to allow overlapping regions between child nodes. One split position represents the minimum boundary of the upper (right) region ($minU$) and the other the maximum boundary of the lower (left) region ($maxL$) in the split dimension.

When a new data object is inserted into a node in the SH-tree, the insertion algorithm compares the MBR of the object with the split information in the root node of the internal KD-tree. If the object is completely inside one of two sub partitions in the root level, the algorithm repeats the same comparison in the next lower level in the KD-tree of the node until the object reaches a leaf node, which points to a child node in the SH-tree.

However if the object does not fit completely inside either of the two sub-partitions, either $minU$ or $maxL$ for the node must be adjusted to include the object. Which one is adjusted is determined based on which sub-partition causes less enlargement of the region, to minimize the size of the overlapping region ($maxL - minU$). Figure 5.4(a) shows an example. This internal node has four child nodes, with each of their sub-regions represented by the bold outlined rectangles. When a new data object that does not fit completely inside any of the four children is inserted into the node, the algorithm must compare the object with the split positions of each level in the KD-tree of the node, and adjust the positions accordingly.

Algorithm 1 shows one way to extend the sub-regions, which is similar to how it is done in the SKD-tree insertion algorithm [74]. Suppose we are inserting an object o whose boundary is ($o.Low(splitDim)$, $o.High(splitDim)$) in the split dimension (splitDim). If $o.Low(splitDim)$ is less than $minU$ and $o.High\ (splitDim)$ is greater than $maxL$, as seen in the root level node of Figure 5.4(a), either $minU$ or $maxL$ must be updated to minimize the increase in the overlapping region.

However this algorithm has a potential performance problem, since when a split position is changed it not only has an effect on the boundaries of the child node that contains the inserted object, but may also increase the boundaries of the other child sub-regions. If the split information to be updated is in the leaf level of the KD-trees, it only increases the region of the one child that will contain the object. However, if a split position in a higher level of the KD-tree is shifted, it increases size of the region of the more than one child node. We refer to this problem as the *cascading overlap problem*. While a basic property of KD-trees causes the cascading overlap problem for non-point data, which is that split positions are shared among child sub-regions, the benefit of sharing split positions is to allow the node fan-out to be independent of the number of

Algorithm 1

SH-tree MBR Insertion algorithm

procedure

$KDTreeInsert(Object\ o, KDTreeNode\ kdNode)$

1: **if** $kdNode$ is a leaf node **then**
2: return $kdNode.childSHTreeNode$
3: **if** object is inside the left sub-region **then**
4: $KDTreeInsert(o, kdNnode.left)$
5: **else if** object is inside the right sub-region **then**
6: $KDTreeInsert(o, kdNode.right)$
7: **else if** object is not inside left nor right sub-region **then**
8: **if** left sub-region requires less enlargement **then**
9: $kdNode.maxL := o.High(kdNode.splitDim)$
10: return $KDTreeInsert(o, kdNode.left)$
11: **else if** right sub-region requires less enlargement **then**
12: $kdNode.minU := o.Low(kdNode.splitDim)$
13: return $KDTreeInsert(o, kdNode.right)$

end procedure

procedure

$SHTreeInsert(Object\ o, SHTreeNode\ shNode)$

1: **if** $shNode$ is a leaf node **then**
2: **if** $shNode$ is full **then**
3: return $SplitNode()$
4: **else**
5: store *o* in an empty slot
6: return NULL
7: $chosenChildNode := KDTreeInsert(o, kdTreeRoot)$
8: $status := SHTreeInsert(o, chosenChildNode)$
9: **if** $status == SPLIT$ **then**
10: // store split information into kd-tree of $shNode$
11: **if** $AddChild(status.splitInfo, status.newNode) == OVERFLOW$ **then**
12: return $SplitNode()$

end procedure

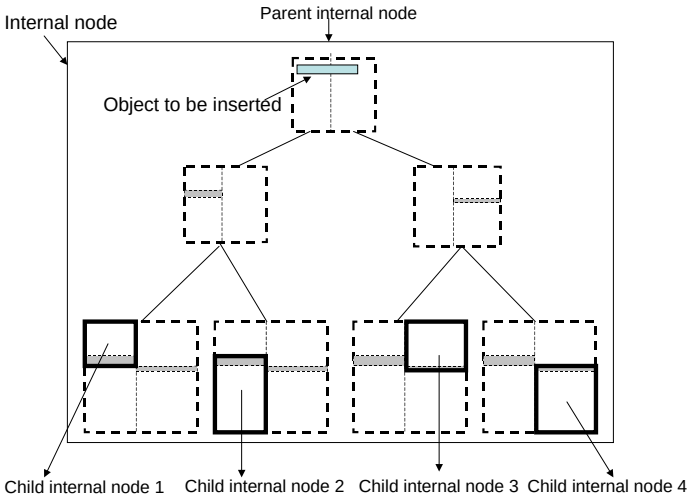
dimensions of the bounding boxes.

One of the benefits of the KD-tree internal node representation is reduced insertion

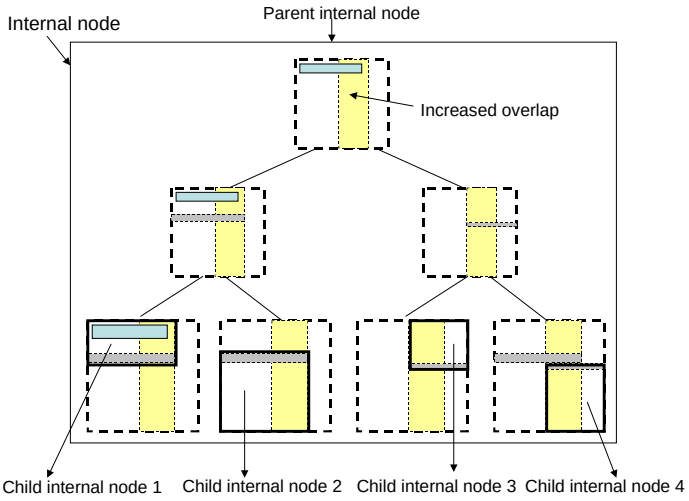
algorithm complexity. R-tree based indexing structures use a list of bounding rectangles in an internal node. Therefore, in order to determine which child node should be assigned a newly inserted object, the R-tree insertion algorithm must compare the query with the MBRs of all child nodes, which requires $\#dimensions \times n$ comparisons of real numbers, where n is the node capacity (number of children). On the other hand, the SH-tree insertion algorithm performs only $\log n$ comparisons when the internal KD-tree is balanced, but n comparisons in the worst case (when the tree is highly skewed), which is still faster than the R-tree insertion algorithm.

5.2.2 Node Splitting

The $minU$ and $maxL$ values, and the split dimension, are locally optimized to reduce the overlap when a node that overflows must be split. The goal of the node split algorithm for SH-trees is to minimize the distance between the two split positions $(maxL - minU)$ for better search performance. For an N-dimensional dataset, only one of the dimensions is used as a split dimension. For each dimension, the bounding boxes of the child sub-regions of the node to be split are sorted twice, based on their lower and upper boundaries in the split dimension. The sub-region with the lowest upper bound and the sub-region with the highest lower bound are selected and put into the lower and upper resulting regions respectively, until the minimum required node utilization is reached. When the minimum required node utilization for both regions is reached, it must be determined which region will increase in size the least if each remaining sub-region is inserted into that region. In this way, all the children are placed into the two resulting regions to achieve minimal overlap in the split dimension. This process is performed for each dimension, and the dimension that causes the smallest overlapping region is chosen to be split. After $minU$ and $maxL$ values and the split dimension are



(a) Overlapping region must be adjusted when a new data object to be inserted is not fully covered by any sub-region. (Shaded regions represent the overlaps.)



(b) Cascading overlap problem - enlarging the sub-region of child internal node 1 enlarges the sub-region of child node 2.

Figure 5.4: *Dynamic adjustment of overlapping sub-regions in an internal node of an SH-tree*

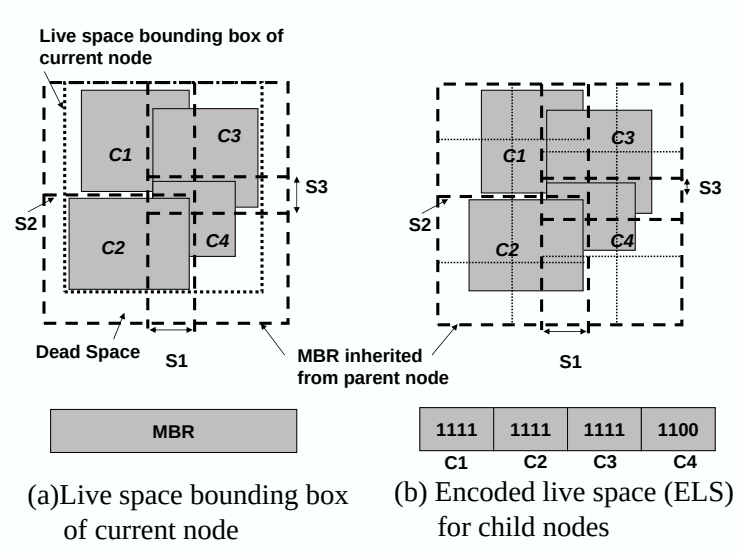


Figure 5.5: *Dead space elimination: live space bounding box vs. live space encoding*

chosen, the split information is stored in the parent node of the node to be split. The complexity of the split algorithm of SH-trees is proportional to the cost of the sorting algorithm, $O(\#dimensions \cdot n \log n)$, where n is the node capacity (maximum number of child nodes). The goal of the R-tree node split algorithm is to minimize the volumes of the resulting MBRs. While an exhaustive algorithm generates all possible splits, that is too expensive in general, so most R-tree implementations employ one of two heuristics. Quadratic split selects the next child entry to assign to one of the two new nodes by selecting the child node that requires the minimum expansion of a current new node MBR, and linear split simply chooses the next child node in the node list to place into one of the two new nodes. The SH-tree node split algorithm has lower complexity than the quadratic split policy used for R-trees ($O(n^2)$).

5.2.3 Object Deletion: Live Space Bounding Box

In both SKD-trees and Hybrid-trees, deletion is a problem because of the overlapping regions between nodes. When a data object that caused the creation of an overlapping region is deleted from the tree, and if the overlapping region is not necessary for other data objects in a node, the overlapping region should be removed in order to make index search faster. However, no such mechanism exists for either SKD-trees or Hybrid-trees. In SKD-trees, the overlapping regions only grows, and in Hybrid-trees the overlapping regions do not change once they are created, which is possible because hybrid trees do not support non-point data. This unnecessary overlap problem is mainly because splitting positions are shared by multiple child nodes. The shared split positions generate approximate (not tight) bounding boxes for child nodes, and there is no way of knowing the precise occupied regions within child nodes unless all sub-trees are searched. In R-trees, condensing bounding boxes is not a problem, because the bounding box information in an internal node is the precise information for all its sub-trees.

In order to solve this problem, SH-trees store the minimum bounding box information in the node itself instead of in the parent, as shown in Figure 5.3. With this additional bounding box information, which we refer to as a *live space bounding box*, SH-trees can avoid searching all sub-trees in order to condense overlapping regions. Instead, the deletion algorithm needs to access a small number of child nodes to determine the actual overlap. The live space bounding box also solves the *dead space problem* of space partitioning methods (i.e. the regions in the MBR of an internal node where no actual data objects are located.).

There have been some previous efforts to solve the dead space problem, such as the *ELS* (Encoding Live Space) data structure used for Hybrid-trees. ELS divides the MBR of a child node into a regular grid and encodes an occupancy map using a small

number of bits, as shown in Figure 5.5. ELS helps improve search performance, but it is not sufficient to condense overlapping regions. ELS gives an approximate hint for the bounding boxes of the child nodes. Besides, ELS is beneficial only with static datasets. If any object is inserted or deleted, the occupancy map must be reconstructed from scratch. Contrary to ELS, the algorithm using the live space bounding box must access the child nodes to get precise bounding box information so that it can condense the overlapping region appropriately. When precise bounding boxes for child nodes are known, it is simple to remove unnecessary overlap. First start from the leaf node whose minimum bounding box was condensed from deleting the object. The algorithm proceeds to the parent node and compares the condensed MBR with the split information in the parent node. In order to check whether the split position can be shifted to reduce the overlap, the algorithm must visit the child nodes of the parent that caused the overlap of the split, in order to get live space bounding boxes for those nodes. After accessing the live space bounding boxes for the children, if the live space bounding box of the parent node can be condensed, then this process is performed recursively up the tree until the root node is reached. Reading the live space bounding boxes of child nodes could be an expensive overhead for disk-based indexing structures since live space bounding boxes reside in child nodes, but it does not cause more overhead than just referencing another pointer in main memory indexes.

Although both the ELS and live space bounding box data structures improve range query performance, they make the number of fan-outs (number of child nodes) for a tree node dependent on the number of dimensions of the data because the space for the encodings depends on the number of dimensions. Higher fan-out is better because it makes the tree height smaller, which makes the paths through the tree for search and insertion shorter. The number of fan-outs for R-trees, Hybrid trees with ELS, and SH-

trees with the live space bounding box are as follows: [1]

1. R-trees:

$$\frac{PageSize}{\#dimensions \cdot (S(lowerBound) + S(upperBound)) + S(ChildPointer)}$$

2. Hybrid trees:

$$\frac{PageSize}{S(ELS) + S(minU) + S(maxL) + S(splitDim) + S(ChildPointer)}$$

3. SH-trees:

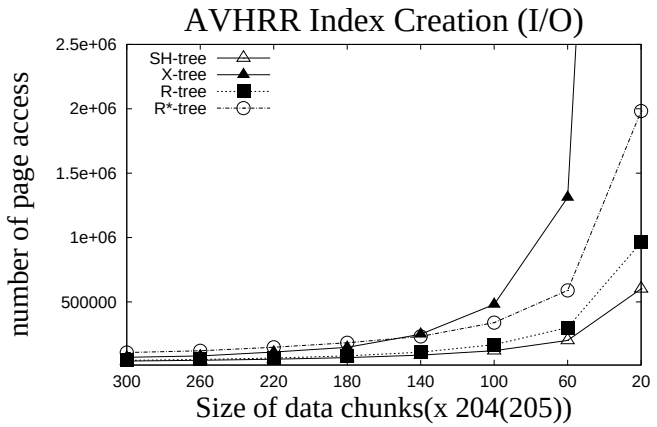
$$\frac{PageSize - \#dimensions \cdot (S(lowerBound) + S(upperBound))}{S(minU) + S(maxL) + S(splitDim) + S(ChildPointer)}$$

For R-trees, the node fan-out is inversely proportional to the number of dimensions, and similarly for Hybrid trees, because the amount of space for ELS encoding is proportional to the number of dimensions. For SH-trees with the live space bounding box, the number of dimensions only decreases the numerator in the formula, so the number of fan-outs for SH-trees decreases linearly with the number of dimensions. Hence, for high dimensional data SH-trees have a larger number of child nodes for a given node compared to R-tree based structures.

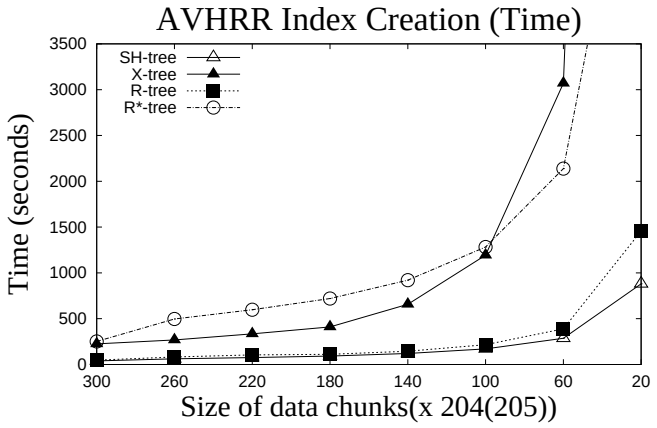
5.3 Experiments

We measured the performance of both index creation and search using the SH-tree, R-tree, R*-tree, and X-tree algorithms. The experiments were run on a SunBlade 100 workstation with a 500MHz Sparcv9 processor, 256MB memory, and a 7200RPM IDE disk with a seek time of 9ms.

[1] $S()$ in the formulas denotes the number of bytes needed to represent the value



(a) Disk Page Accesses

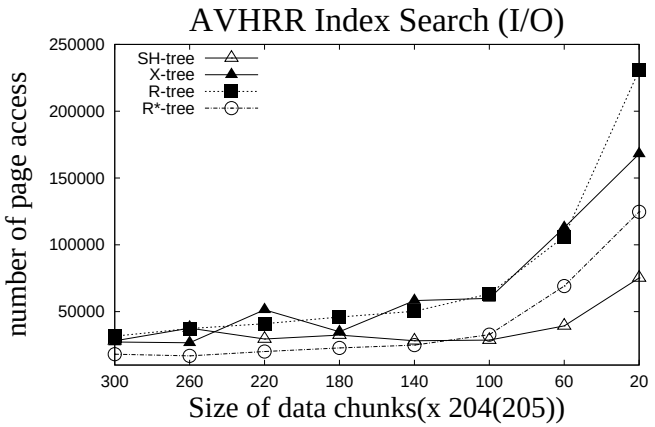


(b) Response Time

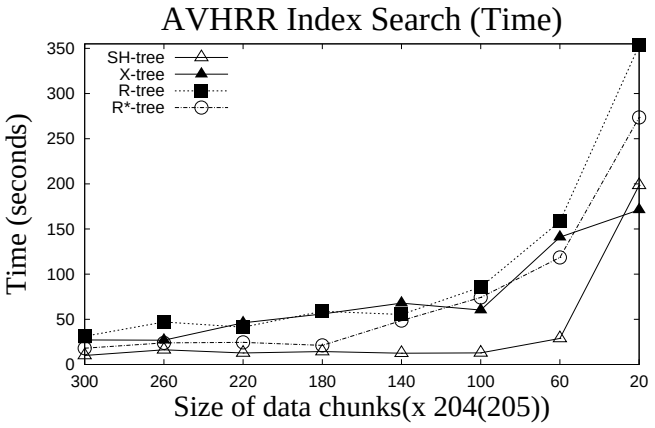
Figure 5.6: *Index Creation for AVHRR Dataset*

5.3.1 AVHRR Dataset

The AVHRR dataset described in Chapter 3 is used to evaluate low-dimension (3D) multidimensional indexing trees. Because the AVHRR sensor swings across the ground track, the sensor values and meta values are stored as two dimensional arrays. We partitioned those arrays into equal sized rectangular chunks. The length of a ground



(a) Disk Page Accesses



(b) Response Time

Figure 5.7: *Index Search for AVHRR Dataset*

track can grow indefinitely, but the length of the cross track is fixed at 409 values. Hence we divided the cross track into 2 unequal partitions - 204 and 205, as was done for the same data in a previous study several years ago [25], and evaluated the performance of the indexing trees with various sized data chunks along the ground track.

We evaluate both the insertion and search times for the SH-tree, X-tree, R*-tree, and R-tree algorithms. For the experiments, we modified the R-tree and R*-tree implemen-

tations from the database group at the University of California, Riverside and the X-tree implementation from Dr. Kriegel's group at Universität München [1]. We modified those codes for fair comparison, so that the X-tree implementation writes dirty pages to disk whenever they are updated, as do all the other tree implementations. For the R-tree algorithm we chose a linear cost split policy, which is as fast as the SH-tree algorithm for creating the index. For common experimental parameters such as minimum node utilization, we chose the same values for all algorithms, and for the tree specific parameters, we used the default values in the various implementations; for example a 0.2 threshold overlap value for topological split in the X-tree algorithm. We turned off OS disk file caching, via the Solaris *directio* system call, which makes the execution times of the algorithms correlate more closely with the number of disk page accesses.

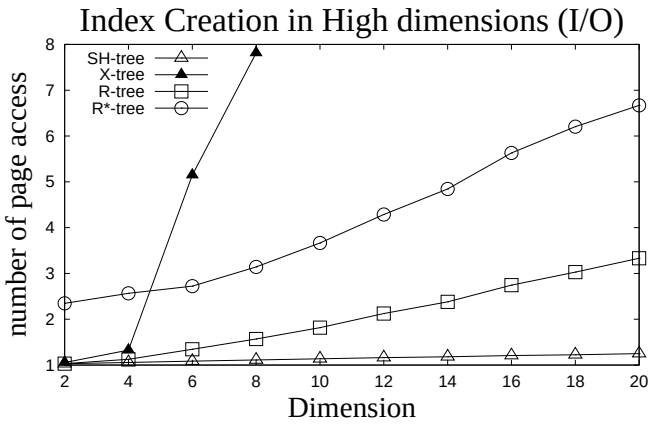
Figure 5.6 shows the time and the number of page writes for inserting bounding boxes, with various data chunk sizes. As the chunk size decreases, the number of leaf nodes in the indexing trees increases, since we are partitioning a fixed size dataset. When the data chunk size is 300x204 (or 300x205), we insert about 40,000 data chunks into the indexing trees, but when the chunk size is 20x204, there are about 560,000 data chunks. Measuring the number of page accesses, the SH-tree algorithm writes the fewest number of pages for inserting all the rectangles, in all cases. For a small chunk size (20x204), the X-tree algorithm writes approximately 20 times as many disk pages as the SH-tree, and 6 times as many disk pages as the R*-tree and R-tree algorithms in the worst case, because of the large size of its supernodes. When a bounding box (or sub-regions) of an internal node must be updated, all other trees access only one page, but several pages must be written for a supernode of the X-tree, although those pages are adjacent on disk. Because of these multiple page accesses, the X-tree index creation algorithm has even worse performance than the notoriously expensive R*-tree

algorithm. The timing results presented in Figure 5.6(b) show the elapsed wall clock time for inserting the bounding boxes of all data chunks into the index.

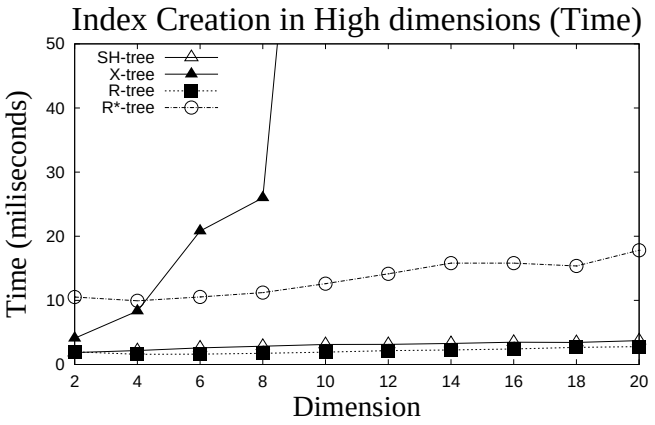
The SH-tree algorithm is fast not only for building the tree, but also for range queries. We generated 2000 range queries using the AVHRR query workload generator. These queries are generated from the workload model, using 16 geographic places of interest (hotspots) at a randomly selected time. The average size of a query in latitude and longitude is approximately 18 degrees and the maximum time span of a query is 10 days. When the data chunks are small, the SH-tree algorithm accesses only 1/4 as many disk pages as the R-tree algorithm and half that of the R*-tree algorithm, as shown in Figure 5.7. However as the data chunk size grows, the SH-tree algorithm tends to generate large overlapping regions, due to the cascading overlap problem. Although the R*-tree algorithm outperforms the SH-tree algorithm for large data chunk sizes, the SH-tree algorithm shows better or almost equal performance compared to the R-tree and X-tree algorithms. An interesting result is that the X-tree algorithm does not perform well for the non-point AVHRR dataset. In the worst case, the X-tree algorithm reads 2.7 times more disk pages than the R*-tree algorithm. For the small number of data chunks, the X-tree algorithm shows similar performance to the R-tree algorithm in disk page accesses. We noted in Section 5.1.2 that the split history used by the X-tree algorithm does not produce better trees for non-point data objects.

5.3.2 Synthetic Dataset

We present experimental results on synthetic datasets, looking at the effects of the dimensionality of the dataset on performance. We generated datasets of 200,000 uniformly distributed hypercubes in the unit hyper-rectangle, with the dimension of the datasets ranging from 2 to 20. As the number of dimensions of the dataset increases,



(a) Average Disk Page Accesses per Insertion

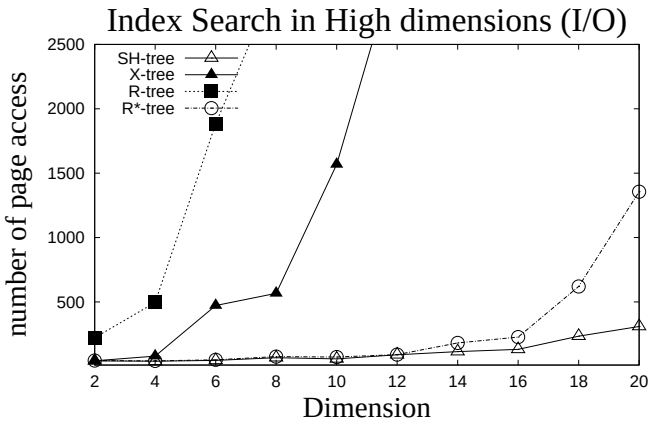


(b) Average Response Time per Insertion

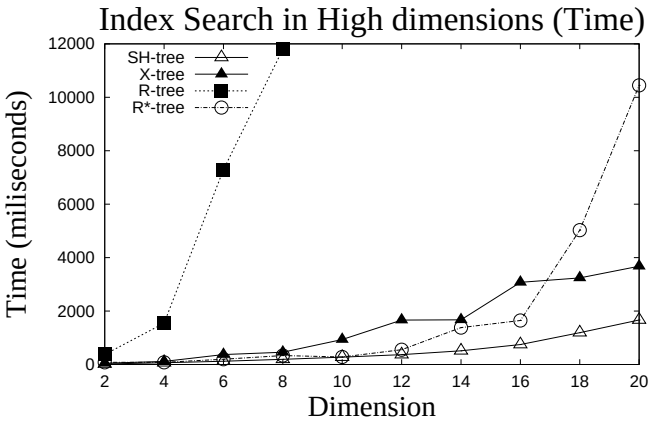
Figure 5.8: *Index Creation for Synthetic Datasets*

the indexing methods based on data partitioning, including the R-tree, R*-tree and X-tree algorithms, all suffer from reduced fan-out. However, the SH-tree data structure scales well to high dimensional datasets, because of its dimension independent fan-out. In three dimensions, the R-tree based trees have the same fan-out as the SH-tree for the same dataset.

Figure 5.8 shows the performance of index creation for the various algorithms, for



(a) Average Disk Page Accesses per Query



(b) Average Response Time per Query

Figure 5.9: *Index Search for Synthetic Datasets*

two to twenty dimensional datasets. Due to its node reinsertion strategy, the R*-tree algorithm takes much longer to create an index than either the R-tree or SH-tree algorithms. As was described for the AVHRR experiments, the X-tree algorithm suffers from the supernode problem, especially in high dimensions. In low dimensions, the SH-tree, R-tree, and X-tree algorithms access a similar number of disk pages, about 43% of that for the R*-tree algorithm. However as the number of dimensions increases, the X-tree

insertion algorithm becomes very expensive. In the worst case, the X-tree algorithm writes 667 times as many disk pages for the twenty dimension dataset as for the two dimension dataset, the R-tree and R*-tree algorithms access up to three times as many disk pages, but the SH-tree algorithm requires only 1% more disk accesses.

Comparing the algorithms for twenty dimensions, the number of disk writes for insertion with the SH-tree algorithm is only 19% that of the R*-tree algorithm, and only 0.2% that of the X-tree algorithm. Overall, the tree insertion algorithm for SH-trees appears to be very efficient. The time to create an index, as shown in Figure 5.8(b), is mostly proportional to the number of disk accesses.

For index searches in high dimensions, we generated and submitted 10,000 uniformly distributed hypercube queries. The performance of the SH-tree algorithm for index search is very good compared to the other algorithms, both for execution time and disk accesses, as seen in Figure 5.9. The SH-tree algorithms scale better than the other tree algorithms to high dimensions. The R-tree algorithm accesses more disk pages than the other algorithms across all numbers of dimensions, and the performance gap grows as the number of dimensions increases. Although the X-tree algorithm accesses fewer nodes than the R*-tree algorithm, it accesses 8 times more disk pages than the R*-tree algorithm and 36 times more pages than the SH-tree algorithm. In our experience, the root node of an X-tree tends to become a huge supernode in high dimensions. For example, with 20 dimensions and 200,000 objects, the size of root node was 635 pages. In that case, no matter how small a range query is submitted, at least 635 disk pages must be accessed unless the root node is kept in memory. However, the elapsed wall clock time for the X-tree algorithm is much better than expected compared to the number of disk operations it performs. This is a result of using supernodes; multiple adjacent disk blocks can be read with a single read system call avoiding expensive disk seek opera-

tions. Hence, the time to search the X-tree is less than for the R*-tree in 20 dimensions. The SH-tree algorithm accesses from 1.5% to 19% the number of disk pages as does the R-tree algorithm, from 22% to 94% that of the R*-tree algorithm, and from 2% to 98% that of the X-tree algorithm.

5.4 Summary

In this Chapter, we investigated how a few commonly used spatial indexing structures perform for multidimensional scientific datasets, and compare their features and performance with that of SH-trees, an extension of Hybrid trees, for indexing multidimensional rectangles.

We have shown that the SH-tree outperforms other spatial indexing techniques on both a real remote sensing dataset and for synthetic datasets, also showing that the SH-tree is more scalable to high dimensions than the other techniques. One of the important properties of SH-trees is that it has dimension independent number of fan-outs as in space partitioning methods but it supports rectangular data. This property makes tree height (search path) of SH-trees shorter than other data partitioning methods especially for high dimensions.

Another important property of SH-trees is that its insertion algorithm is simple and fast. While search performance of SH-trees comes from large number of fan-outs and short tree height, the low complexity of insertion and deletion algorithm makes the insertion/deletion performance of SH-trees efficient.

Scientific data analysis applications query into very large multidimensional datasets, which are growing in size every day, and are becoming truly enormous. For such a class of applications, SH-tree works fast both for searching and updating, and it also

supports indexing rectangular chunked datasets, which reduce the index size for even better performance.

Algorithm 2

SH-tree Node split algorithm

procedure

$SplitNode()$

1: **for** $i = 0$ to $Dimension$ **do**
2: $arrayLeft := AscendingSort(childNodes[].leftBoundary[i])$
3: $arrayRight := DescendingSort(childNodes[].rightBoundary[i])$
4: **while** minimum node utilization is not satisfi ed **do**
5: $leftPartition[] := arrayLeft[0]$
6: remove arrayLeft[0] from both arrayLeft and arrayRight
7: $rightPartition[] := arrayRight[0]$
8: remove arrayRight[0] from both arrayLeft and arrayRight
9: $maxL\_candidate := MAX(rightPartition)$
10: $minU\_candidate := MIN(leftPartition)$
11: **for all** $R \in arrayLeft[] \cup arrayRight[]$ **do**
12: **if** $(R.rightBoundary[i] - maxL\_candidate) < (minU\_candidate - R.leftBoundary[i])$ **then**
13: //R less extends left region
14: $leftPartition[] := R$
15: $maxL\_candidate := R.rightBoundary[i]$
16: **else if** $(R.rightBoundary[i] - maxL\_candidate) > (minU\_candidate - R.leftBoundary[i])$ **then**
17: //R less extends right region
18: $rightPartition[] := R$
19: $minU\_candidate := R.leftBoundary[i]$
20: **else**
21: add R into sub-partition that has less element
22: **if** $maxL\_candidate - minU\_candidate < maxL - minU$ **then**
23: $maxL := maxL\_candidate$
24: $minU := minU\_candidate$
25: $splitDim := i$
26: $leftChildren := leftPartition$
27: $rightChildren := rightPartition$
28: $this := ConstructSubtree(leftChildren)$
29: $newNode := ConstructSubtree(rightChildren)$
30: $status.splitInfo := (minU, maxL, splitDim)$
31: $status.newNode := newNode$
32: return $status$

end procedure

Algorithm 3

Deletion algorithm

procedure

$SHTreeDelete(Object\ o, SHTreeNode\ currNode)$

1: $childArray := KDTreeSearch(o)$
2: **for all** $c \in currNode.children$ **do**
3: //there is no more than one child c that contains o
4: $status := SHTreeDelete(o, c)$
5: **if** $status == NOTFOUND$ **then**
6: continue
7: **if** $status == UNDERUTILIZED$ **then**
8: $RemoveLeaf(c)$
9: **for all** $cc \in c.children$ **do**
10: $Reinsert(cc)$
11: **if** $currNode$ has fewer child nodes than minimum **then**
12: return UNDERUTILIZED
13: **if** $currNode == LEAF$ **then**
14: delete o
15: update live space bounding box
16: **else**
17: update live space bounding box using children's live space bounding box
18: return $FOUND$
19: return $NOTFOUND$

end procedure

Chapter 6

Distributed Indexing for Scientific Datasets

As more storage capacity has become required to store large scientific datasets, recent Data Grid research has focused on developing more scalable distributed storage systems [12, 52]. Large scale distributed storage systems require a data discovery mechanism to locate a specific data item. Many widely used data discovery mechanisms are based on centralized directory services, such as MCAT (metadata catalog) for the Storage Resource Broker [12]. 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*. There has been extensive research on data replication in the past, 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 [67]. 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. Compared to the replicated index and two-level index, the main benefit of a decentralized index is that there is less potential for a resource bottleneck.

In this chapter, we compare the strengths and weaknesses of these indexing schemes. Also, we have performed a scalability study of the indexing schemes via simulation, which is not possible to perform on a real distributed system because of resource constraints. Finally, we provide guidelines for choosing a distributed multidimensional index strategy for data intensive scientific data analysis applications.

6.1 Centralized Indexing

In the centralized indexing scheme, which is commonly used in many scientific data analysis applications, a single index server stores all the index tree nodes, as shown in Figure 6.1. 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 increases the overhead

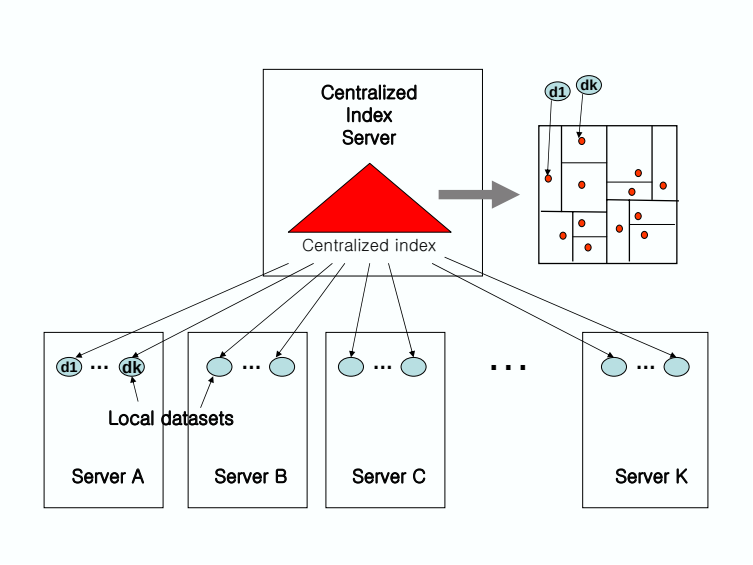


Figure 6.1: *Searching with Centralized Indexing*

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 [67]. 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 [78]. However, replication makes index update operations very expensive [67],

hence we have proposed alternative indexing schemes that will be discussed in Sections 6.2 and 6.6.

6.2 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, 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 bounding boxes of the data chunks 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.

The size of the top level global index depends on the number of local indexes, not on

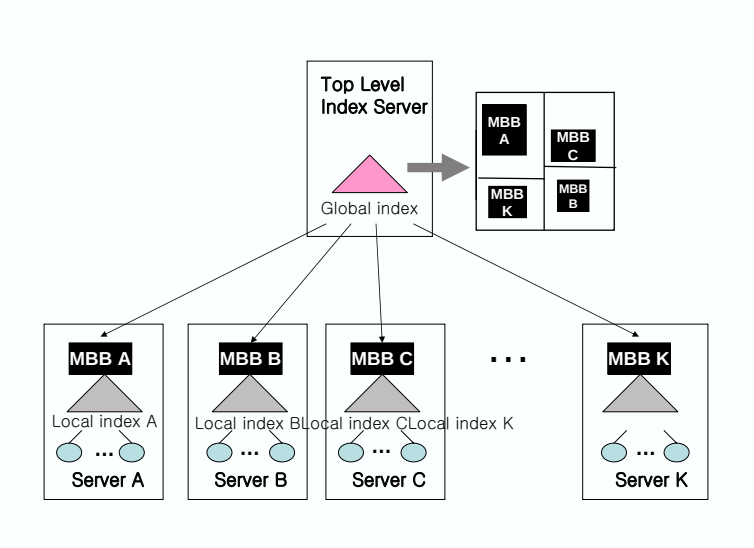


Figure 6.2: *Searching with Two Level Hierarchical Indexing*

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 root 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. Thus, 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.

6.3 Replication Management

Replication of persistent data objects in a wide area network not only reduces access latency, but also improves data locality and increases robustness and scalability. Replication has been shown to be useful for many purposes in distributed systems and in databases. However replication in distributed systems is done mainly for fault tolerance, while database research focuses on its performance implications [104, 101, 76, 41, 55, 5, 79, 105, 106, 27, 87].

For the centralized indexing scheme, the whole index can be replicated, but for the two-level indexing scheme only the global index should be replicated because there is no point to replicating a local index, since the local server will be accessed anyway to read the data from the server. If a local server fails, a replica of the local index for the failed server located on another server is useless, because it will not be possible to read the data since it is not available either. We do not consider replicating the data on a server in this work, since that is outside the scope of the dissertation.

In some of the literature, replicas are considered read-only copies of data objects, which do not change or do so infrequently. This assumption does not apply universally, and especially not for the index, because the index tends to be modified relatively frequently whenever data is stored, replicated, or deleted. As we will show in Section 6.5, it is desirable to create remote copies of indexes when read requests are predominant to reduce query response time, but the number of replicas must be limited to reduce update overhead to maintain consistency between index replicas. The appropriate number of index replicas is determined by several factors, including read/write statistics, network latency, response time, bandwidth, and index size.

When an index is replicated, a client must be able to find where the replicas are located and which one it should submit a query to. In addition, when a query is submitted

to a replicated index, the replicated index server has to determine whether it will handle the request or forward it to another replica for load balancing.

In order to ensure consistency across replicas, distributed locking protocols or atomic broadcasts have been extensively researched. Multidimensional indexing structures have non-deterministic internal structure. If we insert the same data objects in different orders, the resulting tree structures can be different. Nonetheless, any of those indexes will return the same result for any given query, as long as all the structures contain the same data objects in their leaf nodes (i.e., the same data objects have been inserted). We allow some inconsistency between replicas of an index in order to improve the performance of insertion operations, so that clients can insert data concurrently into different replicas, as if there are no *write-after-write* data dependencies across insertions. Among many consistency models, our replication protocol can be classified as an *eventual consistency model*, which requires replicas to converge to the same state after some amount of time. Although different index replicas may never converge to the same tree structures they will converge to the same state to return the same result for any query. There are some transient states when different index replicas may return different query results, but that only occurs in time periods before the replicas converge. However, because there is no global clock across a set of distributed servers and it is expensive to order requests (search and/or update) across different clients, it should not matter which replica has the most up-to-date information. Moreover, many scientific data processing applications do not require a strict consistency model, as do most commercial relational databases. For instance, the Globus toolkit MDS (Monitoring and Discovery System) service also has a weak consistency model [89]. In the consistency model of MDS, available information is recent, but not guaranteed to be absolutely up-to-date. This allows update costs to be reduced at the expense of having potentially slightly stale information. Note that if a

strong consistency model is required for a certain application, it is not difficult to deploy a global locking protocol, as is commonly used in distributed systems applications that require it. However that will increase the cost of index updates in proportion to the number of index replicas.

Although we do not employ global locking protocols, each server needs a local locking strategy to handle concurrent requests, since multiple client requests may arrive within a short time period to a server. We use a coarse-grained locking strategy in our implementation (locking the whole index for the duration of an insertion), but we could use a fine-grained locking algorithm that locks internal nodes of the multidimensional indexing structure in order to increase concurrency [97].

6.4 Performance Model

We present a simplified performance model for both the centralized and hierarchical indexing schemes, using the variables defined in Table 6.1.

The number of network messages for both indexing schemes is proportional to W_S (average number of data servers involved ini a query), which is dependent on query selectivity Q_S (the fraction of the dataset referenced by a query). We distinguish W_S from Q_S, because the dataset distribution may only depend on a one dimension out of several for the dataset, such as time (and not spatial dimensions).

When the workload is evenly distributed across replicas for load balancing, the average number of disk accesses to one of the index replicas in the centralized indexing scheme is

$$AvgDiskAccess_{central} \approx \lceil \frac{Q}{R} \rceil \cdot D_C. \tag{6.1}$$

Also, the average number of disk accesses to the top level global index for a single

server in the two-level hierarchical indexing scheme is $\lceil \frac{Q}{R} \rceil \cdot D_G$, while the average number of disk access to a local index in the two-level hierarchical indexing scheme is $Q \cdot \frac{W_S}{N} \cdot D_L$. Therefore the average number of disk accesses in two-level hierarchical indexing scheme is

$$AvgDiskAccess_{hierarchical} \approx \lceil \frac{Q}{R} \rceil \cdot D_G + Q \cdot \frac{W_S}{N} \cdot D_L \tag{6.2}$$

In Section 6.5 we show that D_G can be ignored, since D_C is always much larger 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 node utilization. If we assume that the 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

$$S_C \approx m + \frac{m}{k} + \frac{m}{k^2} + ... + \frac{m}{k^{\log_k m}} = m + \frac{m-1}{k-1} \tag{6.3}$$

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

$$S_L \approx \frac{m}{N} + \frac{\frac{m}{N} - 1}{k-1} = \frac{1}{N} \cdot (m + \frac{m-N}{k-1}) \tag{6.4}$$

and the size of the global index is

$$S_G \approx \frac{N}{k} + \frac{\frac{N}{k} - 1}{k-1} = \frac{N-1}{k-1}. \tag{6.5}$$

Since the number of data objects is usually much larger than the number of data servers ($m \gg N$), we may substitute S_L by $\frac{S_C}{N}$. The average number of disk accesses D_G, D_L, and D_C depends on the size of the index files ($(6.5)S_G, (6.4)S_L, (6.3)S_C$) and the query selectivity (Q_S). Thus we can substitute D_C in equation 6.1 by $S_C \cdot Q_S$. In equation 6.2 we can substitue D_L by $S_L \cdot Q_S \approx \frac{S_C}{N} \cdot Q_S$ and D_G by $S_G \cdot Q_S \approx \frac{N-1}{k-1} \cdot Q_S$.

We can then rewrite the average number of disk accesses to the centralized index as

$$AvgDiskAccess_{central} \approx \lceil \frac{Q}{R} \rceil \cdot D_C \approx \lceil \frac{Q}{R} \rceil \cdot S_C \cdot Q_S \tag{6.6}$$

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

$$AvgDiskAccess_{hierarchical} \approx \lceil \frac{Q}{R} \rceil \cdot D_G + Q \cdot \frac{W_S}{N} \cdot D_L \approx \lceil \frac{Q}{R} \rceil \cdot \frac{N-1}{k-1} + Q \cdot W_S \cdot S_C \cdot \frac{Q_S}{N^2} \tag{6.7}$$

From the formulas 6.6 and 6.7, we make the following hypotheses:

(1) As the number of replicas increases, the centralized index performs searches faster, but two-level indexing does not obtain much benefit from more replicas until the number of data servers (N) becomes very large.

(2) As the number of data servers (N) increases, two-level indexing performs searches faster, but the performance of centralized indexing does not change.

In Section 6.5, we show experimental results that support these two claims. Also, we measure the overhead for updating the index, which is an important factor in designing a distributed indexing scheme for datasets that change frequently over time, either because data is added or deleted, or data values are changed.

6.5 Performance Evaluation of the Distributed Indexing Schemes

6.5.1 Storage Resource Broker

We have employed the Storage Resource Broker (SRB) in the implementation of the distributed indexing schemes. The SRB is a client-server system developed at the San

Diego Supercomputer Center (SDSC) that provides a uniform interface for connecting to heterogeneous data resources, such as storage area networks (SANs), high performance multi-level storage systems (HPSS), Unix file systems, Oracle databases, etc., over a wide area network [12, 77]. The SRB provides a well-defined storage interface to heterogeneous storage resources by mapping from those interfaces to the underlying storage resource interfaces. Datasets managed by the SRB can be accessed through the MCAT (MetaData Catalog) service, which is a relational database designed to enable attribute-based querying and identification of data, via metadata attached to the datasets. However MCAT does not support multidimensional indexing operations.

We have implemented multidimensional spatio-temporal indexing modules on top of the basic SRB infrastructure, to support multidimensional range queries into datasets accessible by the SRB. We have chosen Spatial Hybrid trees (SH-trees) [66] for the indexing data structure, which we described in Chapter 5. Functions for building and searching SH-trees are implemented as SRB proxy functions. The proxy functions enable an SRB server to forward client requests to other SRB servers without any interaction with the clients [12]. Thus, clients do not need to know where the local indexes and datasets are located.

For performance reasons, we decided not to register index files in the MCAT, because the MCAT can be a serious performance bottleneck. If we register an index file in MCAT to make it easy to find, then whenever we open or close an index file the SRB server contacts the MCAT server to update the metadata for the file. Therefore we implemented a separate directory service to find a remote replicated index. Note that the local indexes for the two level hierarchical indexing scheme, which are not replicated, can be found directly through the global index.

6.5.2 Experimental Environment

We measured the performance of the centralized and two-level indexing schemes on two workstation clusters geographically distributed over a wide area network. The first is a Linux cluster at the University of Maryland, where each of 40 nodes has a Pentium III 650 MHz processor, and the nodes are connected by a 100Mb/sec switched Ethernet network. The second is a Linux cluster at Ohio State University, where each of 20 nodes has a Pentium III 933 MHz processor, also connected by switched 100 Mb/sec Ethernet. The two clusters are connected by the high bandwidth Internet2 wide area network.

We used 3 dimensional AVHRR satellite datasets described in Chapter 3 to evaluate the two indexing schemes. We partitioned the AVHRR datasets into equal sized rectangular chunks, built three dimensional bounding boxes (latitude, longitude, and time) for each of them, and stored the data server address, file name, and the array offset as a pointer to the chunk. The dataset was partitioned into 400,000 chunks. We assigned 10,000 chunks to each of 40 servers, 20 at Maryland and 20 at Ohio State. For our experiments we assigned chunks to servers in order of increasing time (i.e. the first 10,000 chunks in time to server 1, the next 10,000 to server 2, etc.) to achieve good temporal locality on each server. In order to create range queries, we employed the CBMG technique described in Chapter 3. For the experiments, we generated 100 3-dimensional queries (latitude, longitude, and time) per client (up to 80 clients; i.e. 8,000 queries). In the batch query workloads, there are 40 hot points of interest.

6.5.3 Experimental Results

Figure 6.3 shows the total elapsed wall clock time to insert 10,000 objects per client into the index. A client waits for one insertion to complete before performing the next insertion. We ran a single client on each server and increased the number of servers from

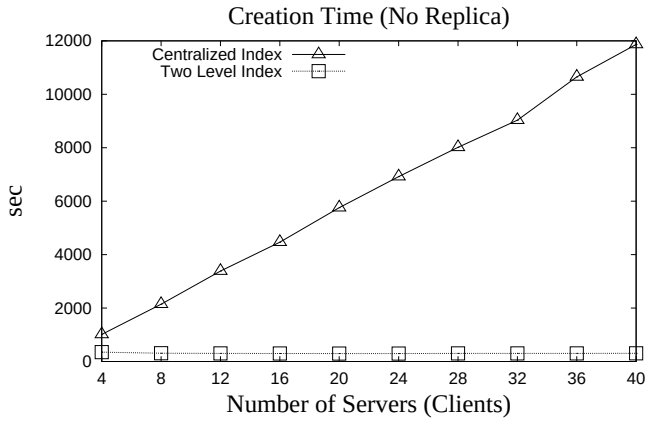


Figure 6.3: *Insertion Time without Replication.*

4 to 40. Thus, the total number of data objects inserted into the index also increased from 40,000 to 400,000. The entire index for the centralized scheme and the global index for the two-level scheme were located on a Maryland server. However, the average insertion time for the clients in Ohio is only 2% slower than for Maryland, and the search times also are about the same. Whether the index is on a local or a remote cluster does not affect the overall performance of index accesses greatly. This is partly because the clusters are connected by Internet2, which has much greater bandwidth than the local area network, and also because most of the time for the index operations is spent on disk I/O rather than in network delay, even for remote index accesses. For this reason, the graphs shown in this section do not distinguish whether the index servers are located in Ohio or Maryland.

As shown in Figure 6.3, the time to insert data into the centralized index increases rapidly as the number of clients increases, since only one insertion request at a time can be executed by the centralized server and the rest of the requests must wait in a queue. Meanwhile, the time to insert data into the two-level hierarchical index is almost independent of the number of concurrent clients. In the two-level hierarchical indexing

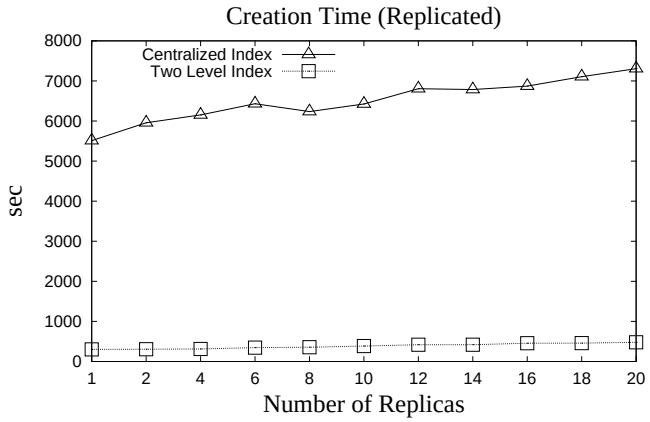


Figure 6.4: *Insertion Time with Replication.*

scheme, most of the insertion operations are done completely locally to the local indexes in the data servers, and involve the server containing the global index only when the index root node MBB on a local server changes, which is not too frequent.

Figure 6.4 shows the index creation time when the index is replicated. We fixed the number of clients at 20, and each client inserts locally 10,000 objects. (The number of data servers was also 20.) As the number of index replicas increases from 1 to 20, the insertion time for the centralized index increases by 32%, and the insertion time for the two-level hierarchical index increases by 58%, but from a much lower starting point. This is because network latency takes a larger proportion of the overall execution time for two level indexing compared to centralized indexing, since the size of the global index is much smaller. Inserting data into multiple index replicas is a non-blocking operation and is performed in parallel, therefore the cost for additional replicas is not very high. If we had employed a blocking insertion operation for strong consistency among replicas, the cost of having more replicas would likely have been quite substantial.

To evaluate the performance of index searches, we ran up to 80 clients with 40 data servers and each client submitted 100 queries. Figure 6.5 shows search performance

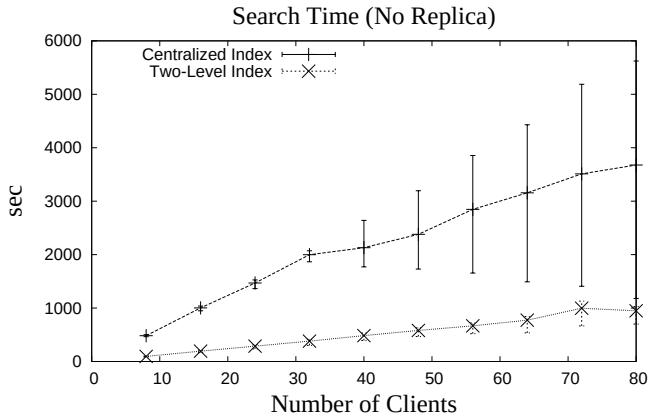


Figure 6.5: *Search Time without Replication.*

for both indexing schemes without replication. Error bars are shown in this graph to emphasize the standard deviation in execution time across clients, because the variance is quite large in contrast to the other experiments. Each query accessed data on 4 to 5 servers on average. The search time for the centralized indexing scheme in Figure 6.5 includes the time for the centralized index server to connect to the local servers that have the desired data for all 100 queries, even though we do not include the time for the servers to read and return the desired data (since the same data will be read for both indexing schemes). We also measured the time for the centralized index server to do its index lookup without connecting to the local servers, which was only 2-4% faster. That means up to 98% of the search time is spent on searching the index in the centralized server.

The two-level hierarchical indexing scheme is up to three times faster than the centralized index without replication. As the number of clients that submit queries increases, the performance gap between the two schemes increases. When the number of clients is over 40, significant resource contention begins to occur in the centralized index server, as shown by the large variance in execution time across queries for large

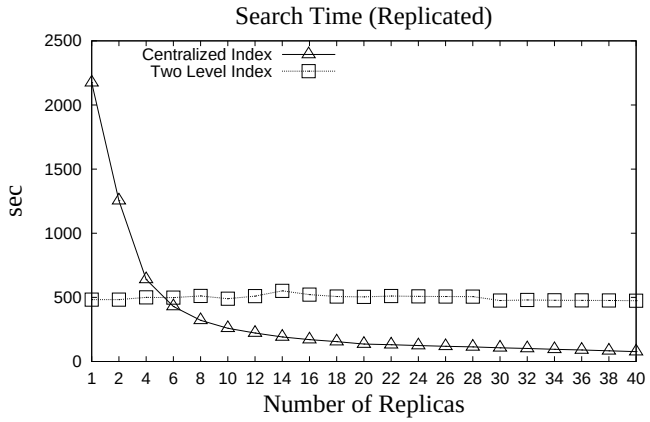


Figure 6.6: *Search Time with Replication.*

numbers of clients. The variance for the hierarchical scheme is very small.

Figure 6.6 shows search performance when the indexes are replicated onto multiple servers. The number of data servers was 40, and we ran a single client per node. We measured the search time varying the number of index replicas from 1 to 40. The submitted queries are forwarded to replicas in round-robin fashion in order to achieve load balance. As illustrated in the graph, the performance of the two-level hierarchical indexing scheme does not depend on the number of replicated indexes, because the server overhead in the global index server is very low even when there is only a single global index and 40 clients submit range queries to the same server. The file size of the global index is only 167KB for 40 data (and local index) servers, so searching such a small index does not cause much overhead. On the other hand, the performance of the centralized index improves significantly with more index replicas. When there are more than 4 replicas, searching the centralized index became even faster than the two-level hierarchical index. These experiments support the first claim we made in Section 6.4 - as the number of replicas increases, the centralized index performs searches faster, but two-level indexing does not obtain benefits from more replicas.

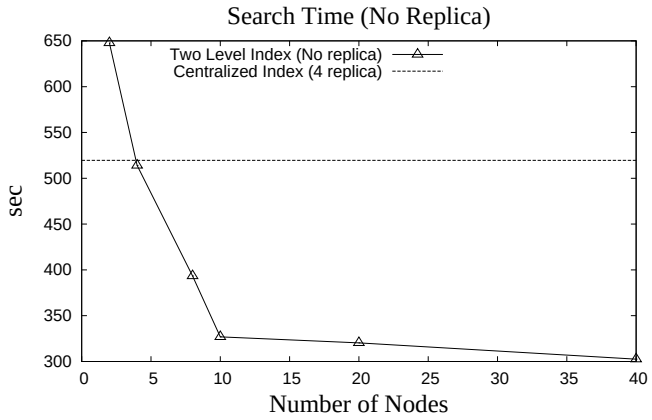


Figure 6.7: *The Effect of Number of Data Servers.*

When the centralized index is fully replicated, any query will access its global index in its local server. However some queries will be forwarded from other servers to the local server to do searches into the local index for the two-level scheme. Thus the number of accesses to the local indexes, which is a dominant factor in the response time for the two-level scheme, does not decrease when the global index is replicated.

Although our experiments showed that increasing the number of replicas of the global index for the two-level hierarchical indexing scheme did not increase performance, if the number of data servers is much larger than the 40 used in the experiments shown, and the number of clients that submit queries is also large, we suspect the global index may become a performance bottleneck.

For the experiments shown in Figure 6.7, we declustered 400,000 data objects across from 2 to 40 data servers (200,000 data objects per server when there are 2 data servers, and 10,000 local data objects per server when there are 40 data servers.) When there is a single data server, the performance of two-level indexing should be no different from that of the centralized index. While two-level indexing benefits more from parallelism as the number of data servers increases, the performance of centralized indexing

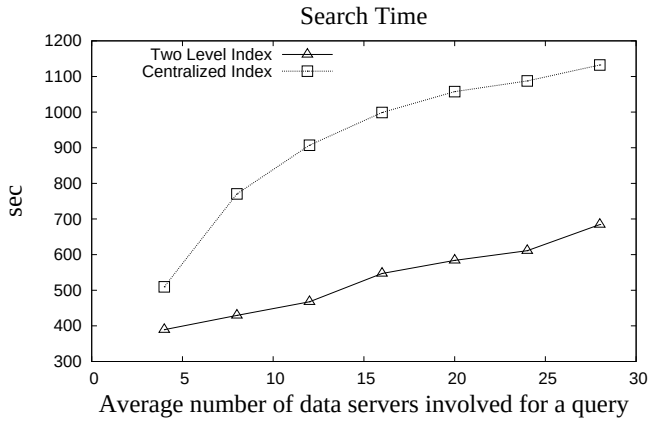


Figure 6.8: *The Effect of Query Selectivity (W_S).*

is independent of the number of data servers. Note that this experimental result support the second claim from the performance model described in Section 6.4. We have experimentally shown that both claims made from the models hold.

Query selectivity is an important factor in evaluating multidimensional indexing structures. In the experiment shown in Figure 6.8, we varied query selectivity with a fixed number of data servers (40), and no index replication. Higher selectivity means a larger query range, hence more of the data is selected. The result shows that as query selectivity increases, the query response time for both indexing schemes increases as well. In centralized indexing, this is because higher query selectivity causes more paths to be searched in the index. Also, for two level hierarchical indexing, higher query selectivity causes more data servers to participate in a query. Although searching local indexes is performed in parallel, clients must wait for query results to be returned for each local index.

To summarize, our results indicate that no one scheme is always best. For index updates, two level hierarchical indexing is superior to replicated indexing since most updates can be executed locally. However, for searching centralized indexing performs

better once there are enough index replicas, and two level hierarchical indexing performs better as the number of data servers increases. Since the number of data servers is usually not a parameter that is arbitrarily set (only depending on where the data is located), we conclude that centralized indexing with replication can provide better search performance than hierarchical indexing. Note that when the centralized index is fully replicated on all servers, all index searches will be done locally, providing better performance than can be achieved with two level hierarchical indexing. However, this is only the case when the index is not updated frequently. Therefore, we argue that if datasets are frequently updated, two level indexing is a better choice than centralized indexing with replication, because of its low update overhead. However if the datasets are static, replication of the centralized index is a better choice rather than partitioning it as in the hierarchical scheme, trading space to store the replicated indexes for performance.

6.6 Decentralized Two Level Indexing

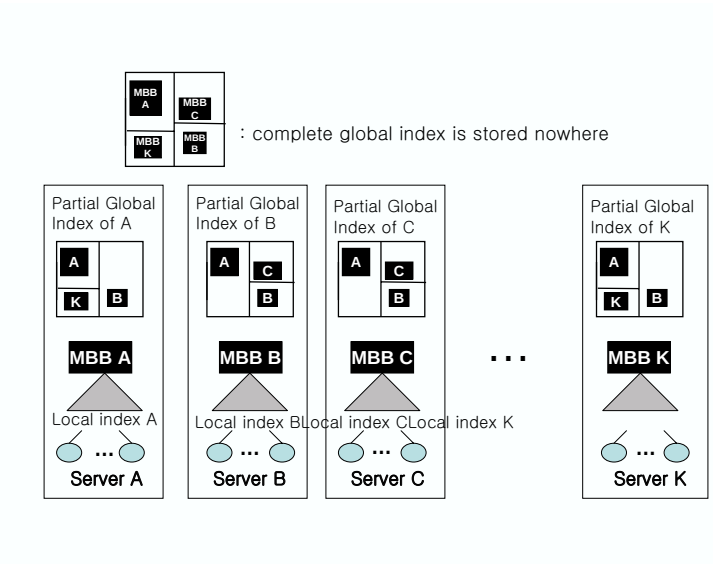


Figure 6.9: *Decentralized DiST Indexing*

DiST is a decentralized version of the two level indexing scheme that we described

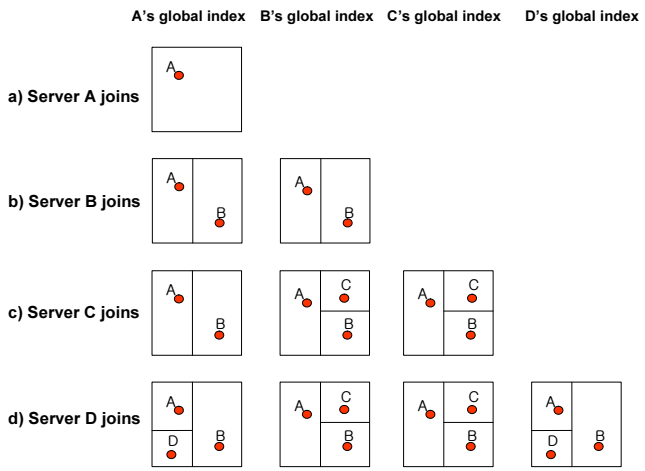


Figure 6.10: *Node Join in DiST*

in Section 6.2. 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 6.9. 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, and each leaf node in the tree corresponds to an MBB of a local index as for the two level index.

When a server joins the system, it becomes an owner of a specific partition in the multi-dimensional space. 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) [44, 68]. 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

Algorithm 4

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: $JoinRequestForward(OwnerID, RootBBX, NewNode)$

end procedure

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

Figure 6.10 shows an example of server join. 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. As shown in Figure 6.10, 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*.

For searching the index, the DiST query routing algorithm guarantees that 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

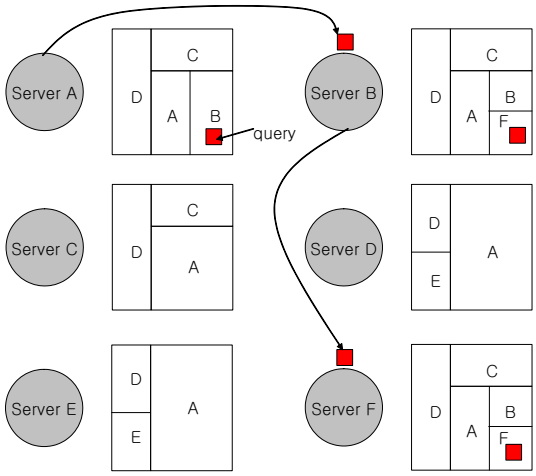


Figure 6.11: *Query Routing in DiST*

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, namely KD-trees. The updated global information is propagated in a lazy manner as we will describe later.

Figure 6.11 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.

Maintaining only a partial global index at each server may result in more network

Algorithm 5

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: $Result$ += $WaitQueryResults(QueryID, OwnerIDList)$
14: $QueryResultForward(Sender, Result)$

end procedure

messages and longer routing path for search queries compared to fully propagating global index updates, as shown in Figure 6.12. 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 Figure 6.12(b).

Lazy Index Updates

When the global index is not a balanced KD-tree, the partial global index may cause 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

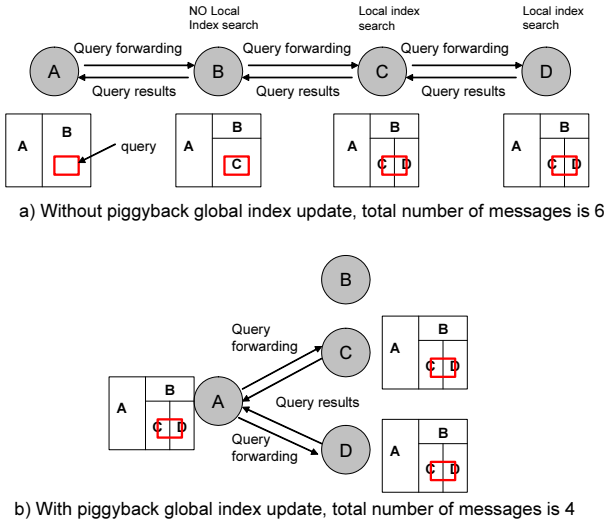


Figure 6.12: *A partial global index may cause additional messages for searches*

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 most 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.

In our first design and implementation of DiST [68], query results were collected and returned back up the query routing path, and lazy index update messages were attached to the query results. But we have determined that this method is not very efficient for index updates, which is reflected in search query response time. Thus in our new design and implementation, used for the experiments shown in Section 6.5.3, DiST returns index update messages immediately after servers detect stale partial global indexes in remote servers. Also, after searching its local index, a data server directly forwards the

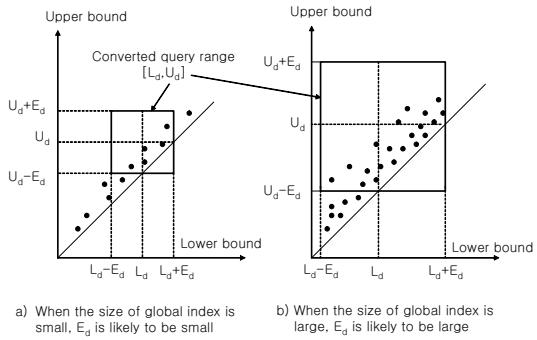


Figure 6.13: *Point Transformation*

query results to the server originating the query (the one the client submitted the query to), and that server collects the results and returns them to the client. The originating query server can determine whether it has received all the results from the data servers by keeping track of query forwarding history (i.e. information on what servers have already seen the query).

Query forwarding history is also required to eliminate duplicate query processing. 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, which should increase monotonically. Using the query ID, servers can detect and not forward duplicate queries.

The Transformation Effect Revisited

We convert a K dimensional rectangle $[L_1, U_1] \times [L_2, U_2] \times ...[L_K, U_K]$ for a data object, where L_k/U_k is the lower/upper bound for dimension k, into a $2 \cdot K$ dimensional point data $(L_1, U_1, L_2, U_2, ..., L_K, U_K)$. 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. Without optimiza-

tion, 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 $[-\infty, qU_1] \times [qL_1, \infty] \times ...[-\infty, qU_K] \times [qL_K, \infty]$. The transformation of a multidimensional rectangle into higher dimensional points causes poor range query performance, due to the unbounded query range. Henrich et. al. [44] propose a transformation technique to solve this problem. They described a split strategy suitable for skewed data distributions, and an improved transformation strategy for range queries. In this strategy, the root node in the index stores additional metadata - for each dimension, the longest edge over all rectangles stored in the tree (E_d). A range query can be then be transformed into a bounded rectangle in the higher dimension - $[qL_1 - E_1, qU_1] \times [qL_1, qU_1 + E_1] \times ...[qL_K - E_K, qU_K] \times [qL_K, qU_K + E_K]$. However, as the longest edge E_d in the tree gets longer, the search path in the tree also gets longer, which causes more message hops for range query searches. If the data rectangles are small, the converted point tends to be near the diagonal, with all coordinates having close to the same value, and if the sizes of all the rectangles are the same the converted point data will reside on a single line. When the rectangles are large, the converted point will be far from the diagonal, and E_d will be also large. If even a single large rectangle is stored in the tree, then it will affect range query performance by causing longer search paths. As far as we know there is no decentralized indexing scheme that can store rectangular data without using the point transformation method. In decentralized systems, grid-based DHT routing methods, such as in CAN [80], are known to be robust and do not have the routing bottleneck that KD-trees have near the root of the tree. However, with the extremely skewed data distribution resulting from the point transformation method, grid-based methods may suffer from unbalanced load.

When the transformation method is used with lazy index updates, the updates could make search performance even worse than when lazy updates are not used. With lazy index updates, the number of server bounding boxes stored in the global index increases and the largest edge will increase as well. Eventually all the servers will have the same largest edge, so the size of the resulting range query will be the same on all servers. In DiST, the size of a range query is more critical than in other space partitioning methods, because even if a range query overlaps a very small region in a server's partition, and the query does not overlap the high dimensional point for the server's local index MBB, DiST will return a hit so that the server must be accessed to process the range query. In Section 6.5, we show how lazy index updates affect range query performance on the point transformed data.

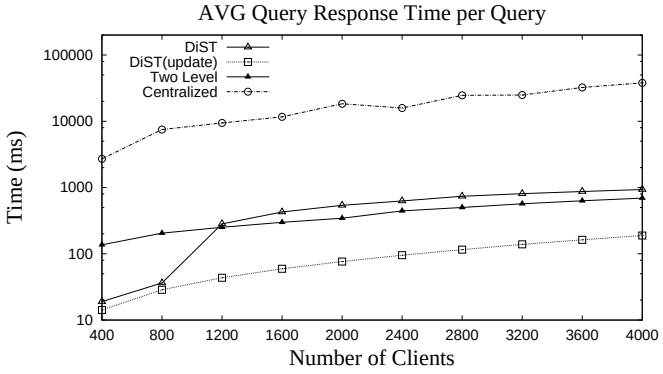
6.7 Experiments: Distributed Indexing

6.7.1 Experimental Environment

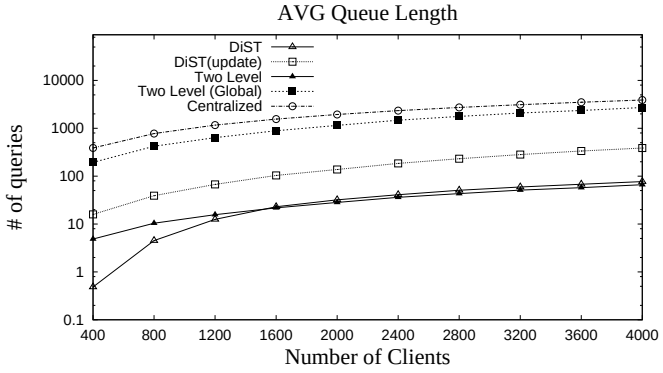
We have measured the performance of the three different indexing schemes on 41 Linux cluster machines. Each of the 41 servers has two Intel Xeon 2.66GHz processors and 2GB of memory, and the servers are connected by a Myrinet network with a nominal maximum of 1 gigabit/s data transfer rate per node.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

We used the same three dimensional satellite AVHRR dataset, which was partitioned into 120,000 chunks. We assigned 3,000 chunks to each of 40 data servers, and we used an extra server as dedicated index server (41 total) for the centralized indexing and global index in hierarchical two level indexing scheme. The clients are distributed evenly across the 40 server machines and each of them submits 1000 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.



(a) AVG Response Time for a Query

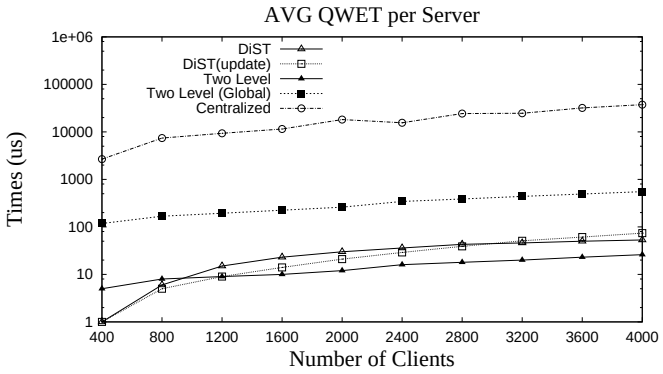


(b) AVG Waiting Queue Length

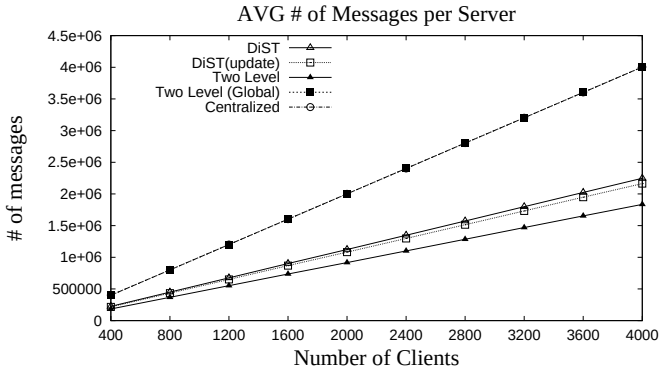
Figure 6.14: *Search performance varying the number of clients*

6.7.2 Experimental Results

Figure 6.14 and 6.15 show the search performance of three indexing schemes for different numbers of clients. Note that the graphs are log scale. The query response time shown in Figure 6.14(a) is the amount of time from the moment a query is submitted to



(a) AVG Query Wait Execution Time for a Server



(b) Communication Cost per Server

Figure 6.15: *Search performance varying the number of clients (cont'd)*

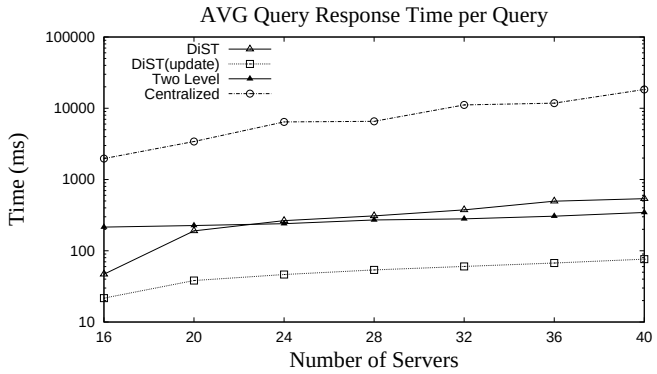
the system until it completes. The average query processing time for the two level indexing scheme is approximately 5% that of the centralized indexing scheme when there are 400 concurrent clients, and only 2% of the time when there are 4000 concurrent clients. The communication costs for both indexing schemes are almost the same as shown in Figure 6.15(b). 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).

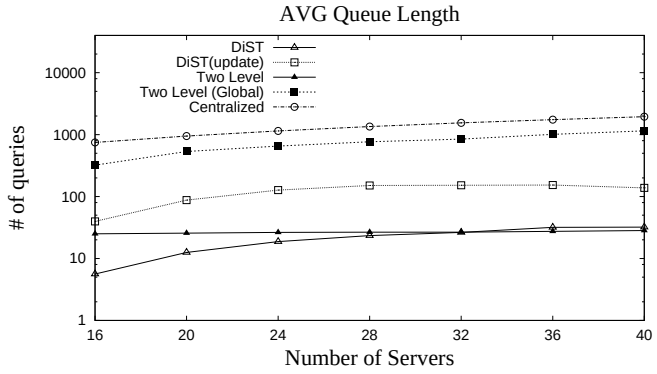
The comparison between hierarchical two level indexing and DiST is more interesting. Under light workloads, DiST(no update) performs searches faster than the two level indexing scheme does. However, when the system is heavily loaded, the query response time of DiST(no update) rapidly increases and becomes slower than two level indexing. One of the reasons is that DiST(no update) has longer routing paths than two level indexing. When the system is not heavily loaded, the long routing path doesn't 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 its decentralized nature. However, as more queries are received than the servers can process immediately, queries are enqueued for processing, thus the long routing paths for DiST(no update) and the queuing delay become a critical performance factor that increases query response time.

Figure 6.14(b) shows the average length of 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(no update) 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 in DiST(no update) 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 in DiST(no update) compared to two level indexing. On the other hand, lazy index update messages makes DiST(update) behave very differently. 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(no update). 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 its decentralized nature. Note that Figure 6.14(b) shows that the global index server for two level indexing has a high queuing delay.



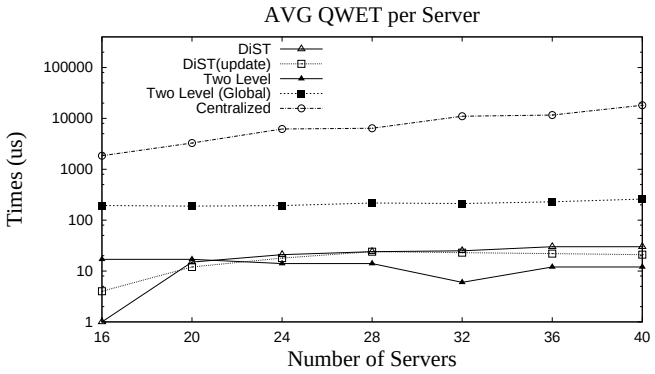
(a) AVG Response Time for a Query



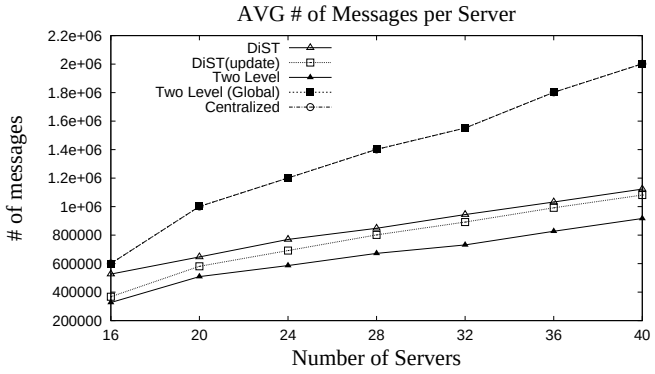
(b) AVG Waiting Queue Length

Figure 6.16: *Search performance varying the number of servers*

The query wait and execution time (QWET) shown in Figure 6.15(a) measures the



(a) AVG Query Wait Execution Time for a Server



(b) Communication Cost per Server

Figure 6.17: *Search performance varying the number of servers (cont'd)*

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(no update) has faster QWET than the global index server for two level indexing, but is slower when measuring query response time. Also DiST(update/no update) has a longer QWET than for the data servers for two level

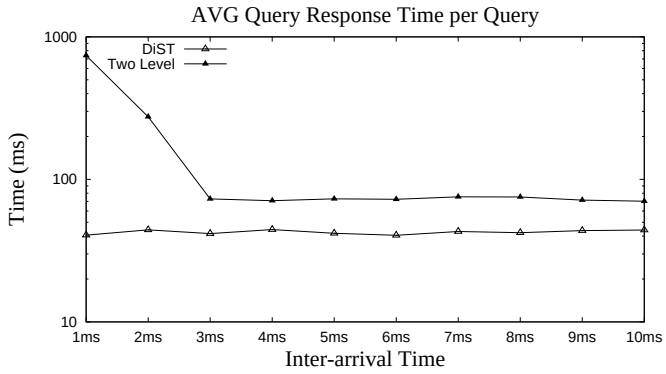
indexing, but has faster query response time than two level indexing. We measured QWET for the global index server and for the data servers in two level indexing separately because they have very different performance behaviors. These results clearly illustrate that the global index server in the two level indexing scheme is a serious performance bottleneck, similar to the problems seen for the centralized indexing server. In Figure 6.14(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 DiST(update).

Figure 6.15(b) shows the average number of network messages received by a server. The number of network messages for centralized indexing is almost the 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(no update), as shown in Figure 6.15(b).

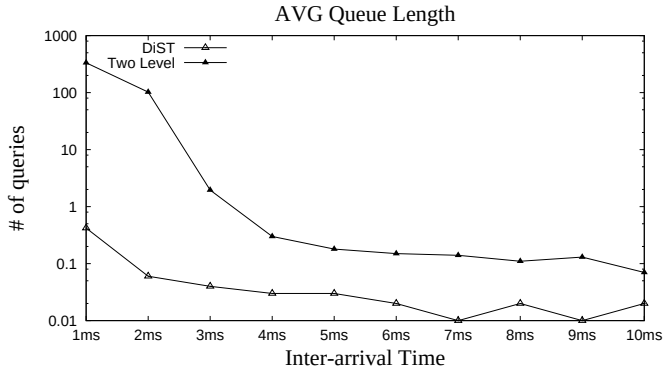
In order to evaluate the scalability of the indexing schemes, we measured index search performance varying the number of data servers from 16 to 40, as shown in Figure 6.16 and 6.17. Each data server receives queries from 50 local clients (i.e. 2000 total clients with 40 data servers), and each of the 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 6.16(a), DiST(no update) performs worse as we increase the number of data servers because the routing path for a query becomes longer with more data servers. Query response time for DiST(update) and two level indexing also grows, but not as much as for DiST(no update). Figure 6.16(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 global index server in the two level indexing scheme suffer more from resource contention than DiST.

If we had run thousands of data servers, the global index server for the two level scheme would have become a bottleneck as is the 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 6.18 shows simulated search performance for hierarchical two level indexing and 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 are measured from doing the lookup as part of the simulation, and is 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 6.18(b) shows that when the average query inter-arrival time is less than the time for searching a global index, more than 1 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 global index



(a) AVG Response Time for a Query

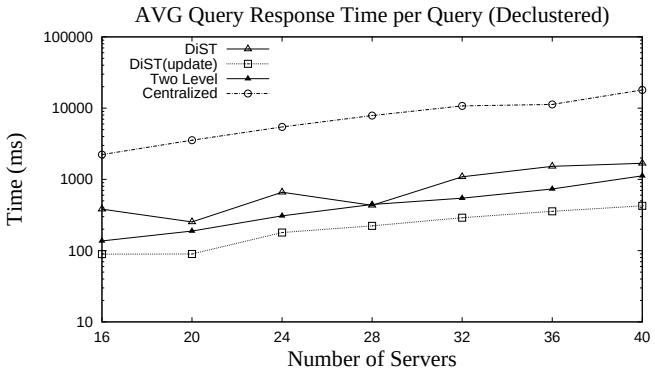


(b) AVG Waiting Queue Length

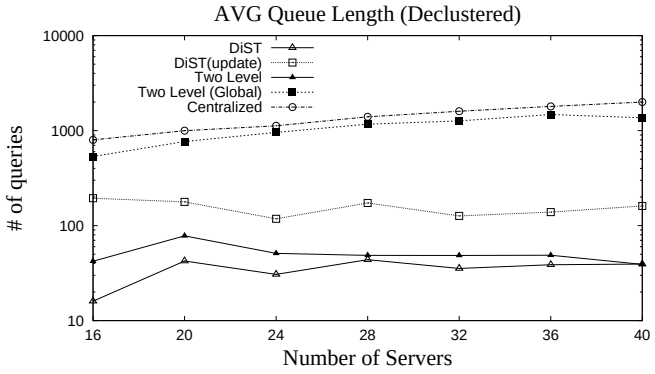
Figure 6.18: *Simulation Results*

server can also become a computation bottleneck.

In order to maximize parallelism for accessing spatio-temporal datasets, it is sometimes suggested that large datasets be declustered across distributed storage archives using space filling curves, such as Hilbert curves [50]. 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 to effectively broadcast all queries to all data servers. De-



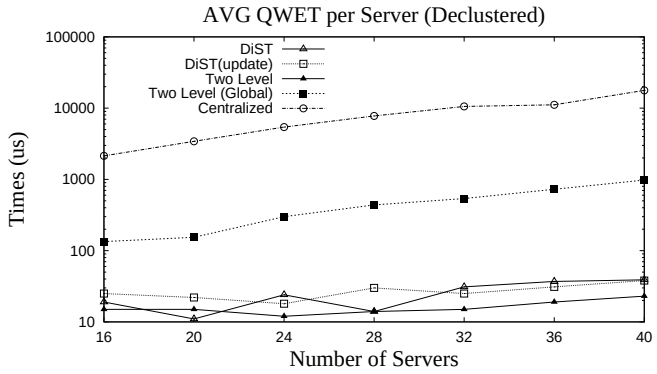
(a) AVG Response Time for a Query



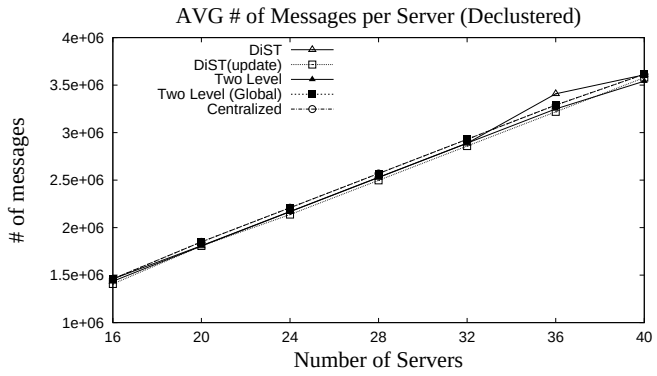
(b) AVG Waiting Queue Length

Figure 6.19: *Search performance with declustered datasets*

centralized 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 6.16 and 6.17, 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 6.19 and 6.20 show that the performance of DiST and two level



(a) AVG Query Wait Execution Time for a Server



(b) Communication Cost per Server

Figure 6.20: *Search performance with declustered datasets (cont'd)*

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 didn't include the time to read the actual datasets. Therefore, the performance degradation is purely from broadcasting query messages. As shown in Figure 6.20(b), 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.

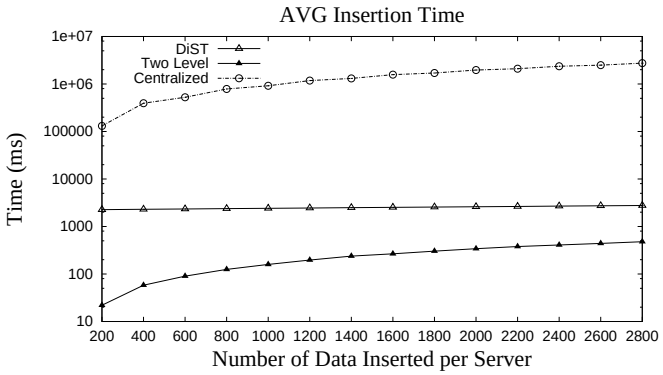


Figure 6.21: *Insertion performance*

In our last set of experiments, we measure insertion performance for the distributed indexing schemes. Figure 6.21 shows the elapsed time to insert data chunks into the index. The number of data servers used was 40 and each of data servers has one client that performs the data chunk insertion operations into the index. As we discussed earlier, most insertions are done only at the local indexes stored in the local data servers, for 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 is very 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.

6.8 Design Choices for Distributed Multidimensional Indexing

In this section we discuss some of the lessons we learned about distributed multidimensional indexing schemes, so that the appropriate indexing scheme can be chosen for a particular application. Each of the three different indexing schemes has strengths and weaknesses. In order to compare the different indexing schemes, it is important to determine application domains where assumptions about the datasets and indexing characteristics hold. First of all, "the scalability of the system" (i.e. the number of data servers) must be considered. Second, we need to perform "workload characterization" determining system properties such as expected query inter-arrival time. Third, "dataset characterization" must be performed, to understand dataset properties such as clustering and data distribution. Fourth, "index update frequency" is another important factor to consider, to understand whether the index will be static or dynamically updated, and if so, how frequently. We discuss these design factors in more detail. Table 6.2 shows the anticipated performance of the multidimensional indexing schemes for the different criteria.

Scalability of the System: The number of data servers affects not only the size of index but also the query routing path length. Distributing the index across a large number of data servers will decrease the size of the index but increase the query routing path length, hence the trade-off between the two effects determines the scalability of the distributed indexing scheme. Search performance for centralized indexing is not directly related to the number of data servers. However it is likely that more data servers

will have more data and a larger index in a central server, which makes centralized index search slow. Partitioning the index seems to always be the best choice for highly scalable systems, but the costs and benefits of decentralization should be carefully considered, since while decentralization does help with scalability in general, it could lead to long, slow message forwarding paths.

Workload Characterization: Accurate characterization of expected application workload can lead to the choice of the best distributed indexing scheme. Query inter-arrival time is the key aspect for selecting a distributed indexing scheme for various workload characteristics. We have shown that partitioning the index disperses query workload and decentralization eliminates any central bottleneck. In addition to query inter-arrival time, query selectivity (i.e. the size of query window) and the distribution of clients across the network are additional workload characteristics that may affect the performance of the distributed indexing schemes.

Dataset Characterization: The distribution of datasets across servers affects the query forwarding pattern for the distributed indexing schemes, as was shown in Section 6.5. In addition to dataset distribution, the size of datasets is another characteristic that can affect the relative performance of indexing schemes. A large dataset on a single server leads to a large local index, which will increase QWET 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.

Index Update Frequency: If clients add or delete datasets frequently, two level indexing is a better indexing scheme than centralized indexing because most insertion and deletion operations are done completely locally, accessing the global index server only when the MBB of a local index root node changes. While only one index update request

can be performed at any point in time by the centralized index server, the hierarchical or decentralized two level index performs index updates in parallel. In DiST, the update of the partial global index in other servers will not be performed immediately because of the decentralized nature of the update algorithm. The updated information will be propagated as other servers find out their partial global indexes are no longer valid during subsequent index searches. As we have shown in our experiments, the DiST insertion algorithm is very expensive compared to two level indexing. Also, lazy index update is not a cheap operation to perform frequently. And a lazy index update could propagate stale partial global index, thus could increase the number of data servers that need to be eventually updated with the correct information. For applications that require frequent index updates that propagate to the global index, the decentralized approach does not appear to be a good choice.

Table 6.1: Description of variables used to model distributed indexing

Variable	Description
N	# of data servers
Q	# of queries
R	# of replicas
D_C	average # of disk accesses for a single query with a centralized index
D_G	average # of disk accesses for a single query for the global part of the hierarchical index
D_L	average # of disk accesses for a single query for the local part of a hierarchical index
W_S	average # of data servers involved in a query ($0 \leq W_S \leq N$)
Q_S	query selectivity ($0 \leq Q_S \leq 1$)
S_C	size of centralized index
S_G	size of global index
S_L	size of local index

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 6.2: Design criteria of distributed multidimensional indexing

Chapter 7

Case Study: Multidimensional Indexing for Query Processing Middleware

Multiple query optimization has been extensively studied in various contexts, including relational databases and data analysis applications [31, 37, 91, 107, 62, 35]. The objective is to exploit sub-expression commonality across multiple queries to reduce execution time through computation and data reuse. Finding a globally optimal query plan has been shown to be an NP-complete problem [92], so creating a good multi-query execution plan can only be achieved using heuristics or probabilistic techniques. Nevertheless, multiple query optimization has been shown to be useful in several contexts.

Over the last few years, my research group has developed a distributed multiple query optimization middleware framework (MQO) for scientific data analysis applications [11]. A unique aspect of our middleware is the utilization of an active semantic cache, where intermediate aggregates used for computing a query are tagged and stored for future reuse. Applications ported to use the middleware can then leverage those cached results by either reusing them directly or by applying data transformations to them [11].

In order to locate objects that can be reused for computing the result of a new query,

MQO's query planner inspects the semantic information for all of the cached objects, attempting to identify the objects that directly or through data transformations will help in computing the new query result. This approach was shown to work well for small caches with tens or hundreds of objects, as a naïve linear search for relevant reusable objects was acceptable. However, as the price of main memory steadily drops, it is not uncommon to find machines with many gigabytes of RAM, which allows MQO to leverage much larger semantic caches. In this scenario, starting with perhaps a few thousand cached objects to be considered during query planning, it becomes imperative to improve the cache look-up mechanism as a means to lower planning time.

Since many scientific datasets are multidimensional (i.e., with space and time object attributes), these datasets can be efficiently indexed using multidimensional spatial tree structures. Likewise, intermediate objects computed when these datasets are processed for queries also have spatio-temporal attributes so can be indexed using similar techniques, making them available for reuse when computing other queries.

MQO is able to efficiently use computational resources from SMP machines and clusters of distributed memory parallel machines. The middleware was also extended with a proxy service [10] that allows data analysis and visualization applications to be distributed onto a heterogeneous Grid computing environment. The Grid is an ideal environment for running applications that need extensive computational and storage resources, as additional resources can be employed incrementally as need arises. For example, as new large scientific datasets are generated as a result of simulations or acquisition of sensor readings or when the pool of users interested in the data increases, new storage and processing resources are required in order to keep up with the additional load. Moreover, because of the demand for storage capacity, bandwidth, and fault tolerance, datasets are often stored in distributed parallel storage systems. For these reasons,

in order to harness the processing power of multiple replicas for distributing the query workload (potentially from several co-existing applications), middleware proxy service implements a simple directory service – the Lightweight Directory Service (LDS). LDS stores and maintains information about the location of datasets, the availability of query processing capabilities, and near-real-time load information on the backend data servers. When input datasets are available on more than one backend server, the information maintained by LDS can be used to distribute the query processing.

While the availability of a distributed cached infrastructure can substantially decrease the amount of time required to process a query, good planning and scheduling becomes harder. That is, forwarding a query to backend servers with lower workloads may actually be detrimental to overall performance, since other busier servers may have cached aggregates that will considerably speed up processing. Striking a balance between reuse of cached aggregates and load balancing can be achieved if additional information is available. For example, if the proxy is also aware of the cache contents in each of the backend servers, it might be better to forward a query to the server that has portions of the query results in its semantic cache, even if it is busier than an alternative server.

In this chapter, we describe the design of a single indexing data structure that is simultaneously able to efficiently locate objects based on semantic attributes as well as based on the cache eviction metric. Also, we experimentally study this indexing approach, demonstrating that, under representative workloads, it yields sizable decreases in query planning time compared to a more traditional configuration that uses a sequential scan of the objects in the semantic cache. Finally, we describe how we integrated hierarchical distributed indexing in the multi-query optimization middleware in order to improve query planning and scheduling performance. We will also experimentally study

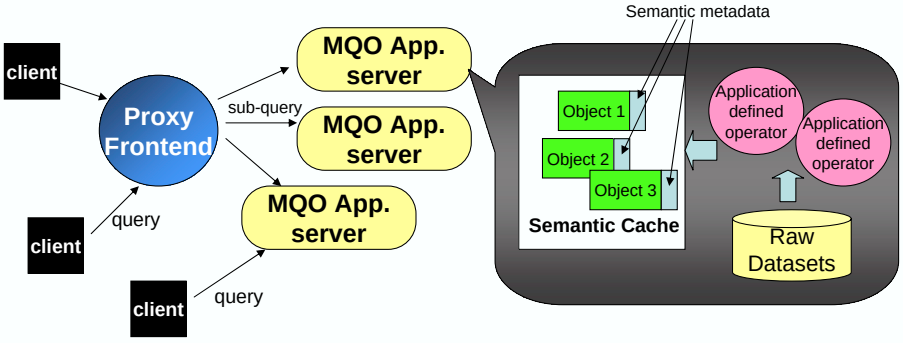


Figure 7.1: *A Grid-enabled MQO system configuration*

this issue in the context of a computationally expensive computer vision application. For relatively stable configurations (few index updates), the simplest way to distribute the index is to replicate it onto multiple servers. However, for dynamically changing index contents, we show that a better method consists of partitioning the index and storing the pieces on multiple servers in a hierarchical fashion.

7.1 Multiple Query Processing Middleware

MQO provides an environment based on C++ abstract operators that are customized when new applications are implemented, or when existing applications are ported. MQO supports several types of computational platforms, transparently employing platform-specific optimizations. From large SMP machines, to clusters of homogeneous nodes, to a distributed heterogeneous Grid environment, MQO is able to use the application-customized operators for efficient query planning and scheduling [11]. MQO offers three main features to improve query processing performance: automatic load balancing, parallel sub-query execution, and semantic caching.

In the rest of this section, we focus our description on MQO's semantic cache infrastructure. Figure 7.1 shows a simplified view of the overall MQO architecture, when configured for a Grid-enabled environment. In this configuration, a proxy server is the system interface with clients. The proxy is used to: (1) receive queries from clients, (2) compute a distributed query plan whereby subqueries may be created and dispatched to different backend MQO application servers for detailed planning and execution, (3) collect the subquery results, (4) assemble the final query results, and, finally, (4) return the results to the client.

When a query is submitted, the proxy instantiates a query object and spawns a query thread, which is responsible for planning, execution, and result assembly and delivery. Semantic caches are available both at the proxy (for final query results) as well as at all of the backend application servers, where intermediate objects resulting from query computation are stored. During planning, the proxy query planner searches locally for cached results to compute the query result. While conventional data caching requires a complete and perfect match for reuse, MQO's *active* semantic cache employs automatic data transformation operators, referred to as *projection* primitives, that enable transforming an existing cached object into a data product relevant for the new query. If the proxy cannot fully compute the query from a single cached object, it generates sub-queries for the incomplete query regions as depicted in Figure 7.2. The sub-queries are recursively processed through the same planning and execution process. When the proxy has exhausted the local reuse possibilities for a query (or sub-query), the query (or sub-queries) is shipped to back-end application servers for further processing.

Sub-queries may be processed by different application servers in parallel [11]. As previously stated, the application servers also have their own semantic caches, hence objects resulting from prior processing are stored in their caches along with semantic

information. Therefore the planning and execution process at the backend servers are performed in a similar fashion to what happens at the proxy.

Originally, MQO's query planner sequentially scanned all cached data objects to locate reusable candidates for evaluating a new query. While this simple approach was adequate for a small cache with few objects, it is limiting in at least two ways. First, it is inadequate for distributed query planning, i.e., when the proxy needs to dispatch subqueries for remote execution at different back-end servers. In this case, it is usually profitable to be able to route sub-queries to backend servers whose caches will aid in the processing. Otherwise, computing query results must resort to processing raw input data, incurring I/O and processing costs. Second, with large caches hosting a large number of objects, locating reusable objects becomes a disproportionately large part of the query planning time. This problem and the interaction between the cache indexing mechanism and the cache replacement policy are the subject of the rest of this chapter.

7.1.1 Semantic Cache Indexing Issues

One focus of cache indexing is on efficiently performing cache replacement operations. Depending on the replacement policy, a data structure with search time complexity O(1) is often available (such as an LRU list – see Section 7.1.2 for more detail), but most replacement policies can be implemented with a simple priority queue (heap), where operations take O($\log n$), where n is the number of objects in the cache. Similarly, locating a particular entry in the cache is usually not challenging. In other words, because cache objects are looked up based on a simple key, such as physical addresses or URL (in case of web caches), a simple hash table look-up suffices. In this case, look-up operations result in either a single hit or a miss.

While that is the case for *traditional* caching, when *semantic* caching is employed for

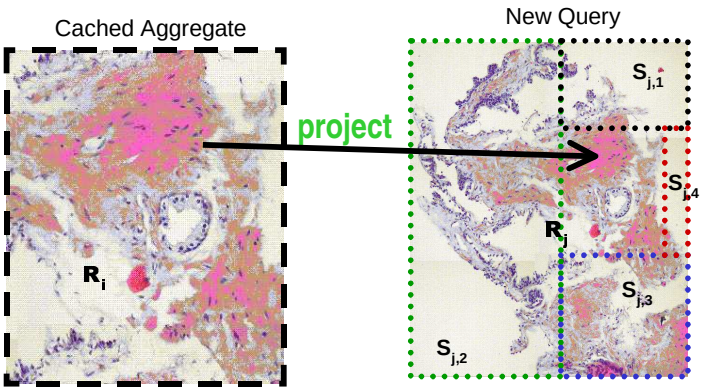


Figure 7.2: *Example for the query execution process in a Virtual Microscope application – an MQO-based application for analyzing digital microscopy collections. MQO reuses a cached object (R_i), performs a data transformation by automatically decreasing the image resolution, and spawns subqueries ($S_{j,1}$, $S_{j,2}$, $S_{j,3}$, and $S_{j,4}$) to generate R_j. (Courtesy of H. Andrade)*

multi-query optimization, the scenario is different. For example, look-up operations are based on multi-dimensional attributes (e.g., spatio-temporal ranges) and also can result in *partial hits*, i.e., an object may partially satisfy the look-up predicate (e.g., a partial spatial overlap or a visualization data product that can be transformed by re-scaling). Hence, the underlying indexing data structure needs to address these requirements.

We refer to the index for a main memory semantic cache as the *cache index*. Ideally, the cache index data structure should primarily make semantic cache look-up operations fast, while, secondarily, keeping the cost of index updates (insertion and deletion of objects) low, since those operations happen less frequently than lookups, particularly for large caches.

Currently, all applications using the MQO middleware employ range aggregation queries [7]. A typical query has spatio-temporal predicates that are defined as ranges,

i.e., a multi-dimensional bounding box in the underlying multidimensional attribute space of the dataset. Similarly, individual tuples in the dataset as well as aggregate objects generated during query processing have spatio-temporal attributes. Therefore, in designing our indexing data structure, we focus on multidimensional index as the basic support for indexing the hyper-rectangular objects stored in the semantic cache.

In the rest of this section, we describe our approach for indexing the contents of a semantic cache, and achieving the desired performance. Our discussion centers on indexing requirements for query planning as well as cache management operations. We demonstrate how a single data structure is able to efficiently address those needs by merging a cache object's semantic information and its utilization profile, which is used for driving replacement decisions. Finally, we discuss improvements to the deletion algorithm, as efficient cache replacement also depends on efficiently reorganizing the index as objects get evicted from the cache.

7.1.2 Cache Replacement Priority Queue

While a multidimensional index can accelerate semantic cache search operations, a separate data structure such as a priority queue is needed for cache replacement. In practice, a crucial part of designing a caching mechanism is the implementation of the replacement policy, to choose the object(s) to be evicted to make room for a new one if the cache is full. A *bad* replacement decision with respect to the *working set* [32] can be costly as in many cases recomputing an object may be very expensive in both computation and I/O, particularly for scientific applications[1]. For this reason, cache replacement

[1]For example, to compute a visualization data product for a digital microscopy application such as the Virtual Microscope [4] requires locating and processing high resolution images. A modern disk such as Seagate's Barracuda 7200.10 provides a 78 MB/s sustained transfer rate [90]. Using a single disk similar

policies have been studied extensively as one of the most effective ways to alleviate the widening gap in access and transfer times between in-core processor caches, main memory, and storage devices. In other words, the effectiveness of caching directly depends on the replacement policy evicting the object least likely to be used in the future and, therefore, avoiding the cost of having to locate, retrieve, and re-process input data.

A large number of replacement policies have been proposed in the context of computer architecture, operating systems, database systems, and, more recently, web proxies. Replacement policies differ in how they choose an object for eviction. This choice affects the decision of how to index the cache contents. For example, the Least Recently Used (LRU) policy, one of the most popular replacement policies, is often implemented by employing a linked list. In a nutshell, the object most recently reused is at the head of the list and the oldest at the tail. As an object is selected for reuse, it gets moved to the head of the list. And the time complexity of insertion and deletion from an LRU linked list is O(1). When a cached object is referenced, the update also can be performed in O(1) time, assuming the semantic tag for the object includes a pointer to its LRU linked list node. While these update operations are efficient, for query planning it is important to *quickly* (i.e., better than O(n), where n is the number of cached objects) locate the objects relevant for computing a new query. This is clearly not possible with a linked list structure, since searching a linked list takes O(n) time.

Aside from LRU, other cache replacement policies typically require more sophisticated data structures for locating an eviction candidate. An example is the Least Frequently Used (LFU) policy, which is often implemented using a priority queue or heap data structure. The heap ensures O(log n) insertion/deletion/search times. Similarly,

to this one, the transfer of high-resolution input data alone – such as a collection of 20,000 $\times$ 20,000 32-bit pixels – from storage to main memory takes approximately 156 seconds.

several other more sophisticated replacement policies for non-uniform cache objects (e.g., Least Relative Value [8]) can also be implemented using a heap.

Considering the cache indexing issues described in Section 7.1.2, our goal is to efficiently satisfy the planning and cache management indexing requirements of a semantic cache in an integrated fashion.

7.1.3 An Integrated Approach – Merging the Indices

Instead of relying on two different indexing mechanisms, one suitable for locating reusable cached objects and another for supporting eviction decisions, our basic idea is to extend existing multidimensional indexing structures to address both needs, accommodating replacement policies that can be implemented by a priority queue.

Multidimensional indexing trees can be modified to support a priority queue for cache replacement, as shown in Figure 7.3. Each leaf node in the indexing tree stores a cached object's semantic information in addition to its eviction metric for cache replacement purposes (e.g., the most recent access time for LRU). Using the object's eviction metric, we overlay an m-ary heap onto the indexing tree. Whenever a cached object is inserted or deleted, we need to *heapify* [28] the tree structure to restore the heap properties. If the multidimensional indexing structure is balanced, the complexity of insertion/deletion is logarithmic. When leaf nodes are accessed as a result of a look-up operation, the eviction metric can be updated, also triggering a heapify operation.

This approach can be used to modify any multidimensional indexing structure if it is a tree. For example, as will be seen in Section 7.2, we experimented with Spatial-Hybrid tree (SH-trees), that works efficiently for rectangular data as we discussed in Chapter 5. This combined cache index will benefit cache replacement policies such as LFU, where the underlying data structure is a heap, more than cache replacement policies such as

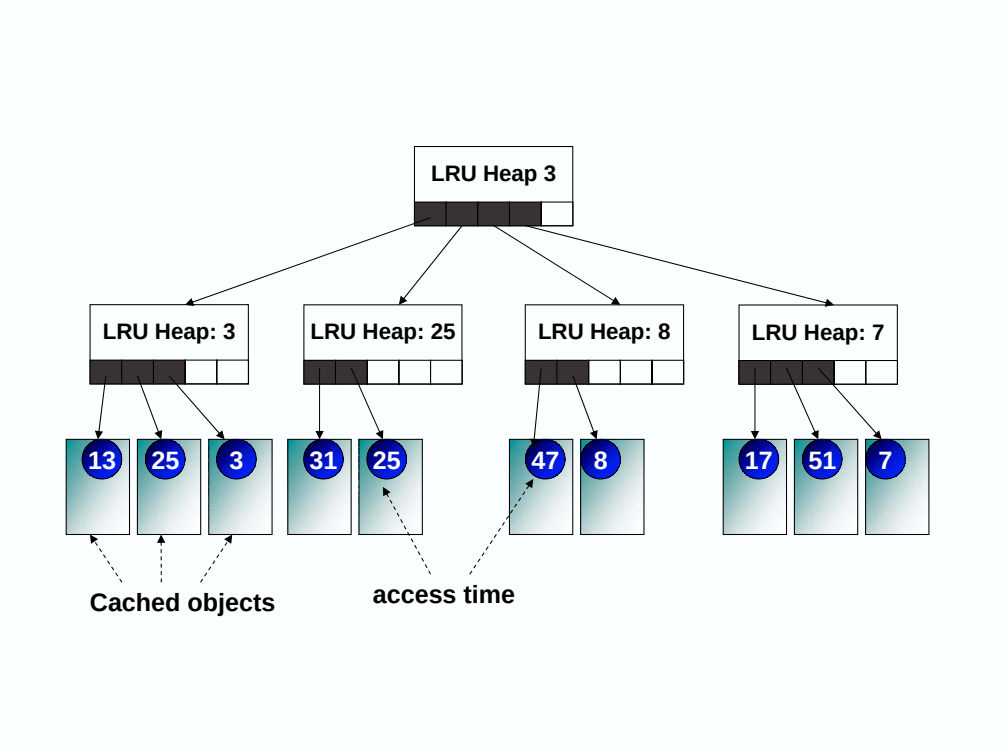


Figure 7.3: *Cache Index with Cache Replacement Priority Queue*

LRU that can be implemented as a linked list, because the overhead of insert/delete operations for a linked list is minimal.

7.1.4 Improving Cached Object Deletion

Most multidimensional indexing research has focused on search performance for multidimensional indexing structures, while index update performance has been neglected in favor of better search performance. For example, both R-trees and R*-trees employ an expensive reinsertion algorithm when a node overflows. This approach is used because reinsertion is known to improve the tree structure for subsequent searches [13].

However, reinsertion might be expensive in the context of semantic cache indexing, as it may be frequently triggered by cache evictions. Hence we designed an alternative reinsertion strategy. Instead of reinserting from the root node, the child nodes of the underutilized node are reinserted from the parent node of the underutilized node.

Algorithm 6

Deletion algorithm

procedure

$MergeSibling(Node\ underutilzed, Node\ parent, Queue\ q)$

1: **for all** $c \in underutilized.child$ **do**
2: $sibling := parent.FindMinEnlargementSibling(c)$
3: $status := sibling.AddSubTree(c)$
4: **if** $status == FULL$ **then**
5: $PushIntoReinsertQueue(q, c)$
6: break

end procedure

procedure

$CondenseTree(Node\ leaf)$

1: $N := leaf$
2: $q := empty$
3: **while** N is not empty **do**
4: $P := Parent(N)$
5: **if** N is underutilized **then**
6: remove N in P
7: $MergeSibling(N, P, q)$
8: **else**
9: adjust N's MBR in P
10: $N := P$
11: reinsert all entries in q

end procedure

This alternative reinsertion strategy is similar to merging the underutilized node with its sibling nodes.

Figure 7.4 shows an example of node merging as performed by the proposed deletion method (shown in Algorithm 6). In the standard R-tree deletion algorithm, line 7 of $CondenseTree$ is $PushIntoReinsertQueue(q, allentriesinN)$. But in our enhanced deletion algorithm, we call $MergeSibling(N, P, q)$ in order to avoid reinsertion. The $MergeSibling$ function picks a sibling node that requires minimum enlargement across

all sibling nodes to accommodate each entry in an underutilized node. But if the sibling has no empty slots, the entry is inserted into the reinsertion queue. This algorithm can greatly reduce the number of reinsertions. In the example, suppose C1 is to be deleted since it does not have enough child nodes. Although C1 is deleted, its dangling child nodes must be reinserted somewhere in the tree. Thus, we need to determine which sibling node will contain each dangling node, and the bounding boxes of the affected nodes must change accordingly. However this merging process is not as simple as it seems. If a sibling node is full, the merging process will make the sibling split, and the parent node may also split recursively. If the parent node splits, there is a problem: which parent should be used for the rest of the dangling child nodes? Our answer to this question is that we do not split the parent node. For each dangling node, if the chosen sibling node is full, the algorithm puts the dangling child node into a reinsertion queue. After all the dangling child nodes are merged into the sibling nodes or put into the reinsertion queue, the algorithm reinserts the child nodes in the reinsertion queue from the root node, as is done for R-trees. This algorithm reduces the number of expensive reinsertion operations and overall makes delete operations faster, as we show experimentally in Section 7.2.

7.2 Experiments: Cache Index

The primary objective of the experimental study is to measure semantic cache index performance when query planning-related look-ups are performed, while at the same time the cache is having objects added and removed as a result of cache replacement operations. We measured semantic cache look-up time with and without the cache index, to show that the overhead of updating the cache index is negligible, considering that it

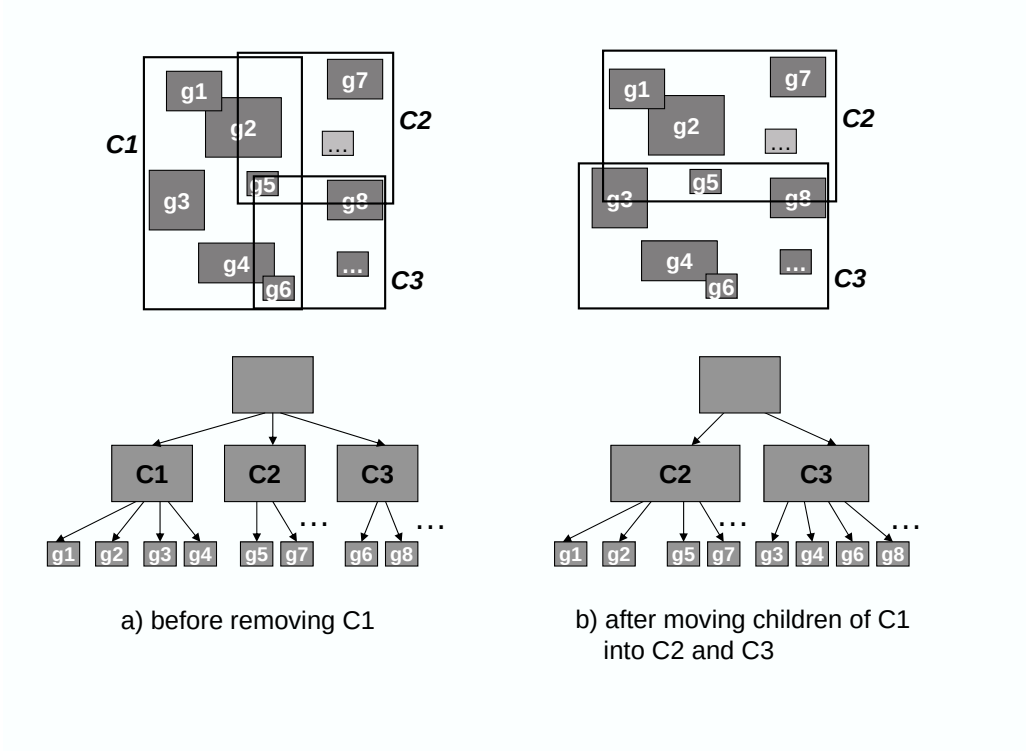


Figure 7.4: *Merging an underutilized node. Instead of reinserting dangling children (g1-g4) from root node, they are inserted directly into their parent's sibling nodes (C2 or C3).*

greatly speeds up cache searches.

7.2.1 Experimental Environment

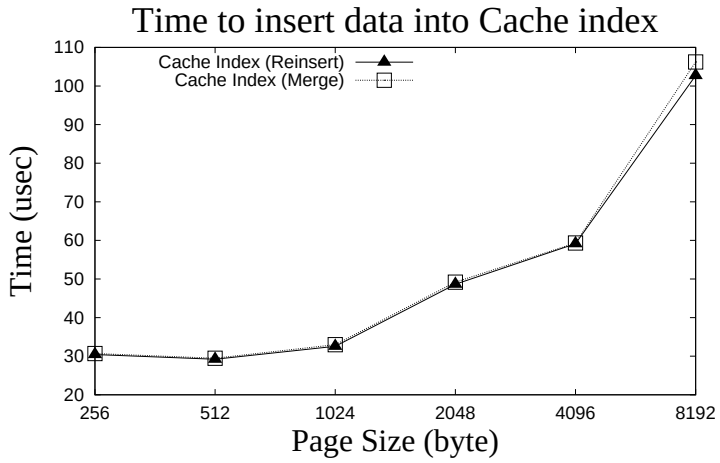
To experiment with the semantic cache indexing improvements, we assembled synthetic query workloads based on Kronos queries [111], a satellite data processing application that was previously ported to MQO.

As extensive user-generated traces from Kronos queries are not available, we employed a variation of the Customer Behavior Model Graph (CBMG) technique to scale up the limited user traces we have. This approached enabled us to generate a large number of queries whose aggregate behavior is similar to that of real users interacting with the system.

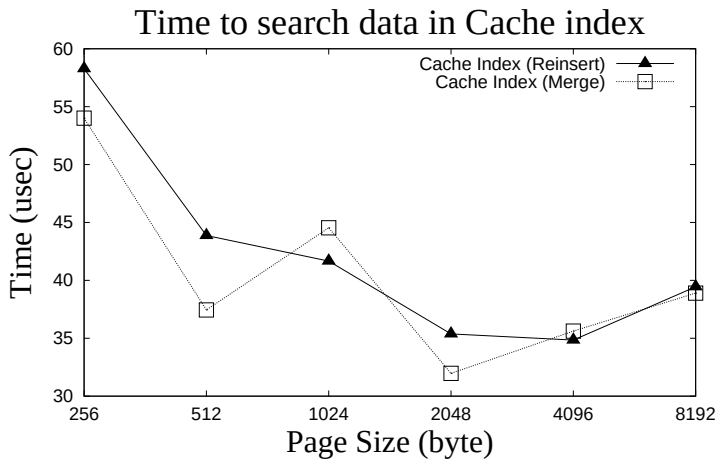
For the experiments, we generated 1,000,000 multidimensional queries (3D queries except for the experiments shown in Figure 7.9 and 7.10), based on different CBMG transition probabilities. As the results we obtained were similar in terms of trends and relative performance, we show results for the following configuration: 20% of queries select a *new point of interest*, 40% of the queries were generated by moving the query window, emulating *spatial movement*, and the remaining 40% of queries were generated by increasing or decreasing the query window size (where the visualization data product *resolution is increased or decreased*).

The central point of the study is understanding the improvements in query planning obtained through using the cache index. For this reason, executing the queries is not really necessary. Thus, instead of running the real MQO middleware, and to better isolate the planning phase, we modeled the semantic cache behavior of MQO and measured only the cache index performance, without including the time to read the raw datasets from disk and fully assemble the visualization data products, which can be extremely

time consuming.



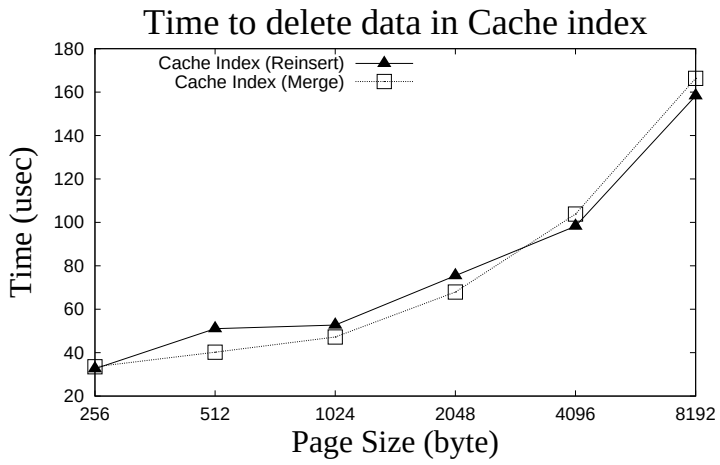
(a) Insertion Time



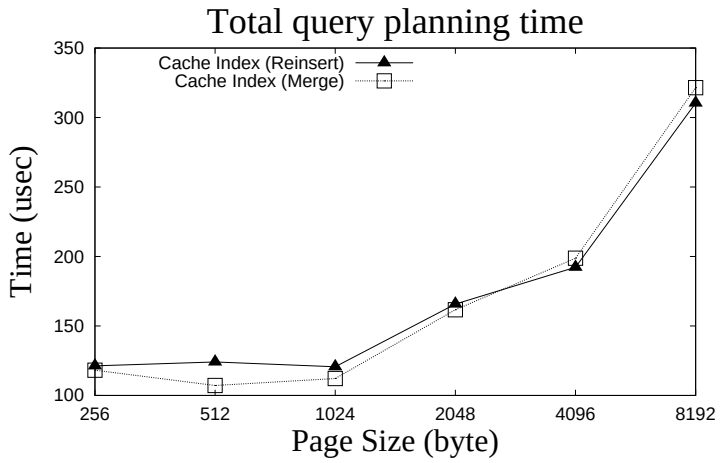
(b) Search Time

Figure 7.5: *Average Cache Index Access Time for a Query with Various Page Sizes*

We measured the performance of the cache index for insert, delete, and search operations using SH-trees described in Chapter 5. The insertions and deletions occur when cached objects are replaced for new queries. The measured times are average results over one million queries.



(a) Deletion Time



(b) Query Planning Time (search+insert+delete)

Figure 7.6: *Average Cache Index Access Time for a Query with Various Page Sizes (cont'd)*

The experiments were run on a Linux machine with a 2.4 GHz Intel Pentium 4 processor and 512 MB memory. We fixed the node utilization factor (the minimum number of child nodes of a valid non-root node divided by the node capacity) to 40% (a common value used in many R-tree implementations), and the page size to 1 KB, except

for the experiments shown in Figure 7.5 and 7.6 that vary the page size.

7.2.2 Experimental Results

We implemented three different cache index data structures. The first one contains two separate data structures, a multidimensional index and a priority queue (heap). This configuration is identified as "Cache Index (Separate)". The second implementation is a single combined multidimensional index as shown in Figure 7.3, denoted as "Cache Index (Reinsert)". The third implementation is also a combined multidimensional index, but its deletion algorithm employs the sibling merge optimization described in Section 7.1.4, denoted as "Cache Index (Merge)". Our baseline configuration is referred to as "SCAN", which employs a priority queue without a multidimensional index (i.e., query planning steps requiring access to the semantic cache are carried out as a sequential scan over the cache contents).

Note that the index tree node size does not have to be the same as the disk page size since the cache index resides in main memory. The index tree node size is an important performance factor since it determines node fan-outs. Thus we measured the performance of the cache index as a function of tree node size. Figure 7.5 and 7.6 illustrate how node fan-out for the cache index trees affects performance. We now describe the performance metrics we measured. Insertion time is the amount of time needed to insert a new query result into the cache index and priority queue. Search time is the amount of time needed to find objects in the cache index. Deletion time is the amount of time spent on cache replacement. Query planning time is the total amount of time spent on insertion, search, and cache replacement. For a single query, a single search operation is carried out to search for hits in the cache, and a single insert operation is performed to insert the query result. However, the number of cache delete operations

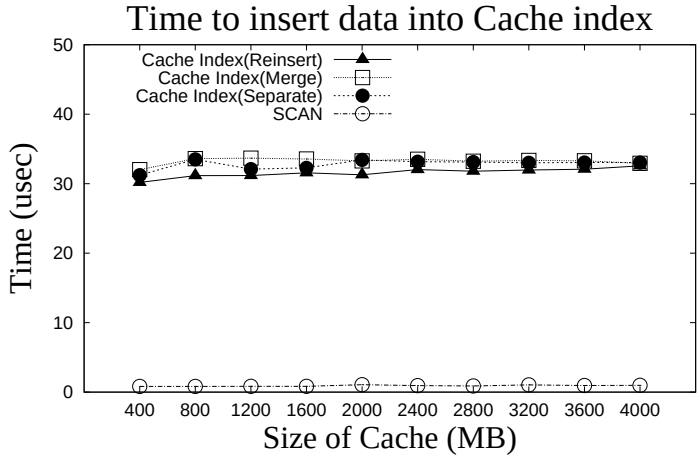
depends on the volume of the current query, depending on how many objects must be be replaced in the cache, which in turn depends on how big the objects are.

SH-trees can hold 38 child pointers for a page size of 1 KB. Unlike R-trees, SH-trees have a dimension-independent number of child pointers, as for KDB-trees. In contrast to search time, insertion and deletion time increases as node size increases. For an 8 KB node size, each node has 313 child pointers. With fewer child pointers, the insertion and deletion algorithms require less computation time to split and recalculate the bounding boxes for tree nodes. However, the longer search paths caused by smaller fan-out increase search time.

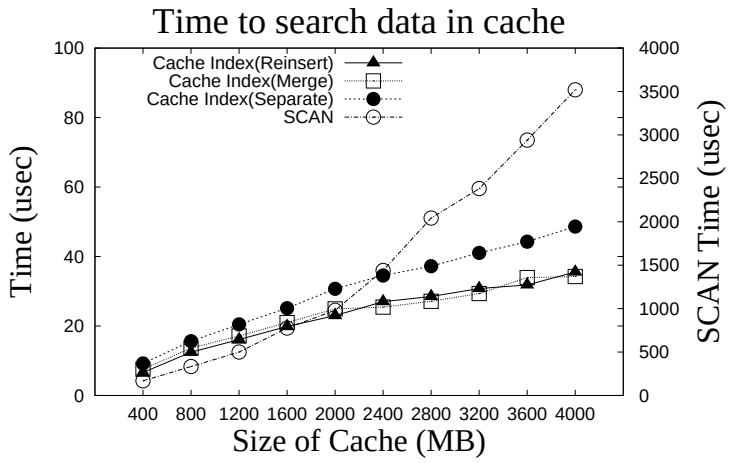
The node fan-out of all disk-based balanced tree structures must be larger than 3, so that we can split a node without violating the minimum node utilization constraint (40% for these experiments). Thus we could not run the experiments for node size smaller than 256 bytes. The total execution time slightly increases when the node size is smaller than 512 bytes. Figure 7.5 and 7.6 show that the cache index with sibling merge for deletes shows better performance than the delete with reinsert in most cases. Also, both cache index structures show good performance when the node size is 512 or 1024 bytes. Thus, for the rest of our experiments, we fixed the index node size to 1KB.

Figure 7.7(a) shows the wall clock time to insert metadata for new data objects into the cache index. We increased the size of the cache from 400 MB to 4 GB and assume that the query result size for 1 query volume unit (latitude · longitude · time) is 1 KB. The query volume for this experiment varies from 12 to 818 and the average query volume is approximately 200. Hence the cache can store from approximately 2,000 (400MB/200KB) to 20,000 (4GB/200KB) cached objects. We approximated the size of cached objects to be the volume of the query range, because we assume that a larger range query generates larger query results. For example, the size of Kronos satellite

images for a certain region for two days would be twice as large as that for a single day. If many large query results are stored in the cache, the number of cached objects would be less than the case when many small query results are stored.



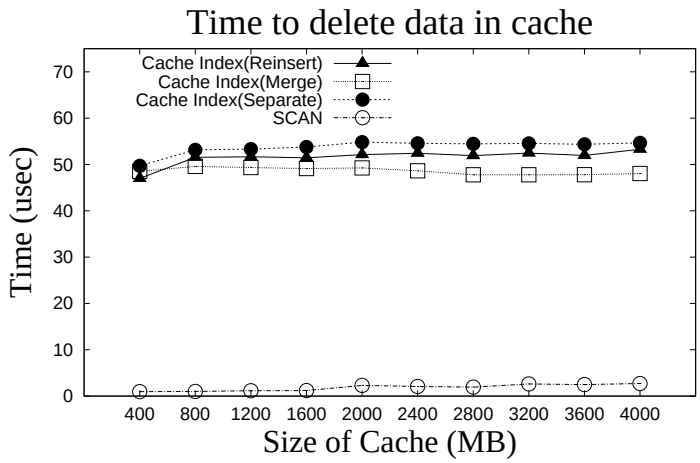
(a) Insertion Time



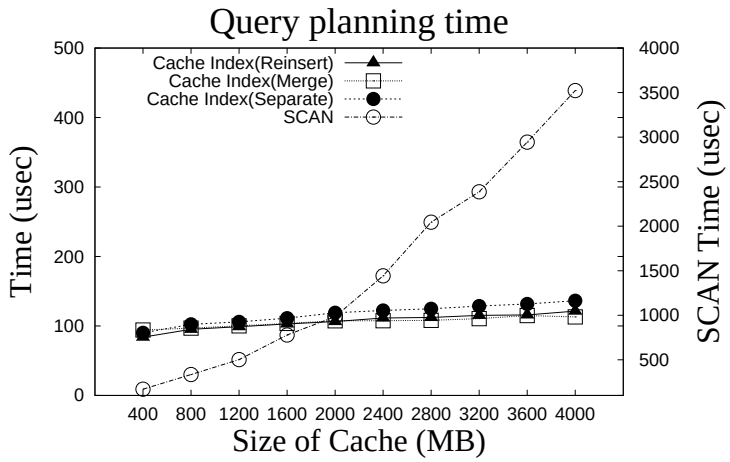
(b) Search Time

Figure 7.7: *Average Cache Index Access Time for a Query with Various Cache Sizes*

Cache Index (Merge) has the same performance as Cache Index (Reinsert) for insertions, since their insertion algorithms are the same. Cache Index (Separate) also shows



(a) Deletion Time



(b) Query Planning Time (search+insert+delete)

Figure 7.8: *Average Cache Index Access Time for a Query with Various Cache Sizes (cont'd)*

similar performance with the other cache indexing structures, which means that the extra insert operation into a separate heap does not cause much overhead. Although we increased the cache size, the insertion time for cache indices does not increase accordingly because the complexity of the insertion algorithm is logarithmic. However, the

insertion time for the Cache Index configurations is much higher than that of the SCAN configuration, which employs an LRU list.

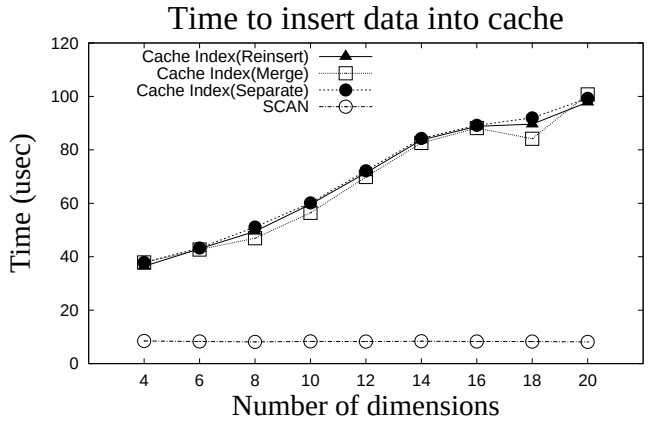
The deletion time for the SCAN configuration is also very low and constant, as expected. The Cache Index (Separate) and Cache Index(Reinsert) configuration also show similar deletion performance, which means that a separate priority queue does not have high overhead for deletion. However, the deletion performance of Cache Index (Merge) was up to 11% faster than that of Cache Index (Reinsert).

We expected that the sibling merge deletion algorithm would hurt search performance by increasing overlapping regions across child nodes, but the effect on search performance does not seem to be very significant from our experiments. The search time for Cache Index (Merge) was at worst only 8% slower than that of Cache Index (Reinsert), and sometimes Cache Index (Merge) was even faster. An interesting result in Figure 7.7(b) is that Cache Index (Separate) shows much slower search performance than the other cache indices, because of restructuring the heap from updating the time stamps after each successful search operation for a large number of cached objects. Note that the times for SCAN are on a different scale for Figures 7.7(b), 7.8(b), 7.9(b), and 7.10(b), and are shown on the right side of the graphs.

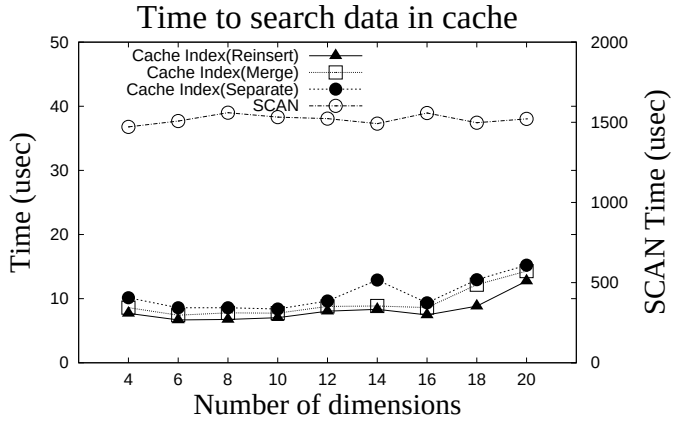
Although a separate heap does not increase insertion and deletion overhead much, we found that it causes some amount of overhead when searching. As we increase the cache size, the performance gap between linear scanning and the cache indexing methods grows. The search time for the linear scan increases as the cache size grows, because the large number of objects in the cache makes linear scanning expensive.

Instead of breaking down the performance into improvements in individual insert/delete/search operations, Figure 7.8(b) shows overall query planning time that includes all the insert, delete, and search time. The figure shows that performance improvements from the

cache index are quite substantial compared to linear scan. Sequential scan does not have much overhead for insertion or deletion, but scanning itself is expensive enough that when the cache can contain more than 2,000 data objects the SCAN configuration performs worse than the Cache Index configurations.



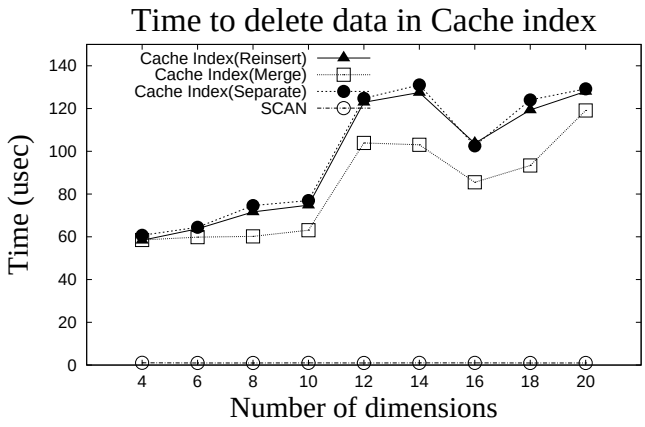
(a) Insertion Time



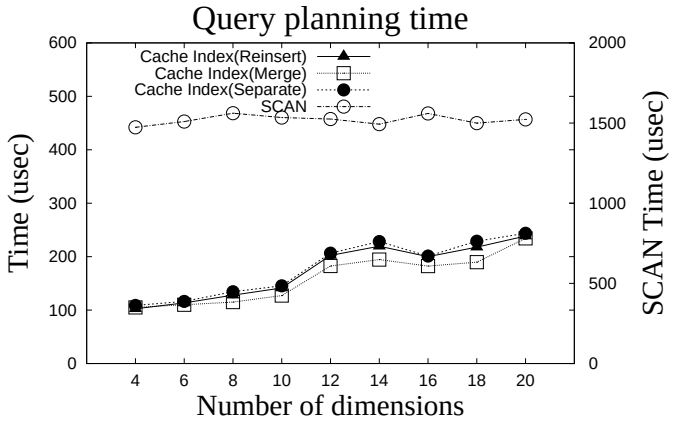
(b) Search Time

Figure 7.9: *Average Cache Index Access Time for a Query with Various Dimensions*

Weber et al. showed that any multidimensional index would perform worse than linear scanning for high dimensional data, due to the well known curse of dimensionality



(a) Deletion Time



(b) Query Planning Time (search+insert+delete)

Figure 7.10: *Average Cache Index Access Time for a Query with Various Dimensions (cont'd)*

problem, especially for nearest neighbor queries [102]. Range queries can also have this problem in high dimensions, but many scientific datasets have 4 or fewer dimensions (e.g., three-dimensional space and time). However, we still wanted to experiment with high-dimensional data, so we generated synthetic high dimensional query workloads (one million queries total) and ran the experiments with a fixed cache size that can store

approximately 4,000 objects, as shown in Figure 7.9 and 7.10. We ran the experiments with different cache sizes, but show only one of them since the results are similar. For this workload we used the same transition probabilities as for the previous experiments.

As the number of dimensions increases, the insert/delete/search time of the cache index methods also increases, but linear scan seems to be almost independent of dimensions. This is because if a data object does not overlap in the first dimension, the rest of the dimensions are skipped. Cache Index (Separate) shows the worst performance in most cases for search, and Cache Index (Merge) shows the best performance for delete. In terms of overall query planning time, Cache Index(Merge) shows the best performance and is about 6 times faster than linear scanning for 20 dimensions. However, we observed that the performance of the cache index degrades as the number of dimensions increases, while linear scan performance seems to be independent of the dimensionality of the data. Thus we suspect that with a large number of dimensions, a cache index would be of no use. But as we mentioned earlier, most scientific datasets are low dimensional, and a range query is a more common access pattern than nearest neighbor queries for many scientific data analysis applications.

7.3 MQO in Grid environment

MQO targets several types of computational platforms, transparently employing platform-specific optimizations. From large SMP machines, to clusters of homogeneous nodes, to a distributed heterogeneous Grid environment, MQO is able to use the application-customized operators for efficient query planning and scheduling. In the rest of this chapter, we focus on MQO's Grid configuration, which employs a proxy component referred to as the Active Proxy-G (or APG, for short). This discussion is necessary to

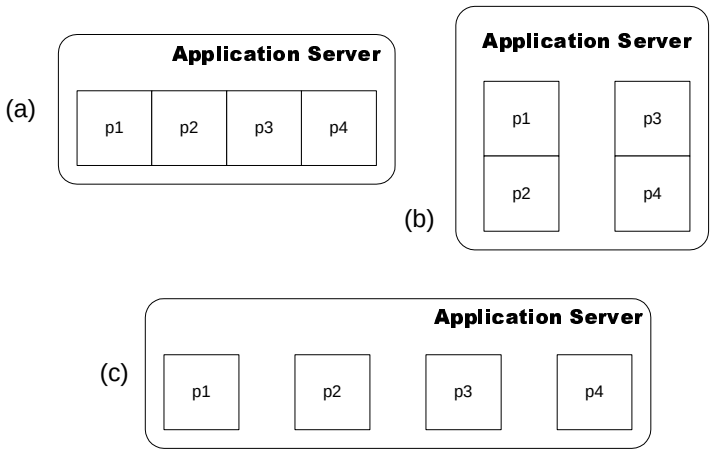


Figure 7.11: *Application Servers with different parallel configurations. (a) shared memory, (b) distributed shared memory, or (c) distributed memory*

provide the context for the integration of distributed cache indexing capabilities into the middleware.

The APG works as a front-end to the distributed multiple query optimization system. When a query is received by the proxy, it may be able to process the query directly using its local cache. If cached aggregates alone cannot be used to fully compute a query, the proxy server generates sub-queries for the unresolved portions and repeats the same process for the sub-queries, recursively. If no processing can be done by the proxy, the query is forwarded to backend application servers, which then use their local cache or directly access the raw datasets to compute the results. The backend application servers can run on cluster nodes, shared memory machines, or distributed shared memory machines with attached large-scale storage devices. Figure 7.11 graphically depicts these different configurations. APG enables the backend application servers to be distributed and connected in any hierarchy forming a computational Grid.

When a client submits a query through the proxy, the proxy's main task is to locate a suitable backend server to process it. The proxy employs a directory service (the Light

Directory Service – LDS), where information such as the location of datasets as well as workload performance metrics are stored. Dataset locations constrain the set of backend servers that can be used for servicing a query (i.e., in the current prototype a query can only be processed by a backend server that has direct access to the datasets referred to by the query). Performance metrics collected by the proxy can be used for partitioning and balancing the work when multiple backend servers are able to process a query. When replicas exist, the proxy has to select one of them based on a scheduling policy. The original MQO implementation could be configured to use two different policies [10]: (1) round-robin, where a replica is selected for processing a query based solely on where the last query was serviced, and (2) load-based policies where, by actively collecting metrics such as CPU and disk utilization, the *least busy* backend server with a suitable replica is selected. Note that clients can also directly submit queries to backend servers, if they know where the datasets are located, which further increases the potential for load imbalance. That is, imperfect information at the APG as well as additional load from servers directly submitting queries to backend servers compound the scheduling problem.

With the existing query scheduling policies, the proxy service could only leverage previously computed results that were part of queries it had seen (i.e., queries that have been submitted through the proxy interface). Moreover, the proxy cache contents are only related to the query *final* data product. While we have previously shown that this approach was indeed able to provide substantial decreases in query execution time, it does not permit the utilization of *intermediate* data products that are automatically cached as a query is processed because these are only available at the backend servers. Furthermore, the proxy cache can only grow in size up to the available memory in the node hosting the proxy. For these reasons and in order to generate better query plans

that can take into consideration the contents of remote semantic caches, an efficient distributed index is needed.

The semantic caches available at the backend application servers are independent and evict content as need arises according to their own cache replacement policies without any global coordination. In general, strong distributed cache consistency is expensive and inherently non-scalable. More directly, it is very hard to keep track of the up-to-date contents of remote semantic caches in distributed systems. On a more positive note, strong cache consistency is not really necessary for application correctness, as query results can always be computed directly from the raw datasets, albeit with a performance penalty. Therefore, it is possible to tolerate cache misses, which may occur when a query plan is assembled based on stale information. Typically, if recomputing a query from scratch is cheap as measured by I/O and CPU processing costs, simple distribution of the load across backend servers may perform reasonably well. However, many scientific and visualization applications are both data and compute intensive. It is often faster to reuse cached aggregates rather than to generate them from scratch [49]. For these applications, more reuse of cached aggregates and improved load balance will decrease average query execution time and maximize overall system throughput. As will be seen in the next section, we accomplished this through distributed indexing.

7.4 Distributed Indexing for Query Optimization

A distributed multidimensional index enables update and search operations to be performed in parallel, thus providing the means for distributing the load across multiple servers. In Chapter 6, we have studied three types of distributed indexing schemes: index replication, hierarchical indexing, and decentralized indexing. Each of them ad-

dresses different needs. Since cached objects stored in the middleware backend servers' semantic caches can potentially change very quickly due to workload characteristics and eviction requirements, the index replication approach is not suitable since it incurs significant overhead in propagating the index changes. Similarly, the decentralized indexing approach is not suitable either, because it does not perform well if the index is changing rapidly. Finally, hierarchical indexing has been shown to work well in a distributed environment even when updates are frequent.

Integrating hierarchical indexing with the MQO middleware consisted of extending the backend application server with a local index that tracks the contents of its semantic cache. The proxy was extended in order to host the global index. Since the system needs to be able to quickly insert, delete, and search the local index, we have employed SH-trees for the local index.

The low likelihood of global index updates comes at the expense of limited knowledge about objects available in the local indices. For example, global indices may have a large amount of *dead space* (i.e., multidimensional regions in which no actual objects are located, but are indexed as a result of an enlargement operation made to accommodate a new object).

The tradeoff between the amount of knowledge available at the global index versus the amount of communication can be controlled by creating additional hierarchy levels. With this change, the global index stores the MBRs of the second (or third) level nodes of the local indexes. Storing finer grained MBR information reduces the dead space and, as a consequence, also reduces the likelihood of cache misses. Alternatively, in order to mitigate this problem, we have devised a simpler technique that employs a *bitmap live space encoding* data structure, described in Chapter 5.2.3. The bitmap provides the global index with finer grain information for the root node MBR of the local indices

by partitioning the root node MBR into several subregions. If any next level tree node overlaps the partitioned subregion, it is marked with a 1, otherwise with a 0, as seen in Figure 5.5. The additional information can be used to eliminate some false cache hits. This approach is very economical for low dimensionality objects, as is common for many scientific datasets, which typically have fewer than 4 dimensions (e.g., space and time). For higher numbers of dimensions, the bitmap encoding suffers from the well known *curse of dimensionality* problem.

7.5 Multiple Query Scheduling Policies

The distributed index addresses the issue of locating candidates for executing queries or subqueries on behalf of the proxy. However, picking the best candidate for executing a query requires balancing the potential for reusing aggregates in the semantic cache of an application server versus the wait to be serviced by that server. In extreme cases, a server with popular aggregates may be swamped with additional load. Thus, query scheduling plays an important role in load balancing and, ultimately, in overall response time and system throughput.

In the rest of this section, we discuss 5 query scheduling policies we have implemented and experimented with, as will be shown in Section 7.6.

Round-Robin: Round-Robin scheduling is our baseline policy. It assigns a roughly equal number of queries to each application server. This technique is simple, well-understood, and generally performs well when queries and application servers are homogeneous. On the other hand, it does not take into consideration any state information, such as semantic cache contents and backend servers' individual loads.

Load-based: Load-based scheduling assigns a backend server to a query based on

the load observed in each of the backend servers. It does so by selecting the least busy backend server. This is done through MQO's Workload Monitor Service, which actively collects performance metrics from each of the application servers, by polling them periodically (the polling period is typically set to 15 seconds). Several individual metrics are collected, such as the server's internal thread pool utilization, disk read rate, and the size of the query wait queue. These metrics can be used to infer the server load. For simplicity, we employed only the size of the wait queue[2].

Index/Overlap: This policy makes scheduling decisions solely based on the result of a global index lookup operation. An exception exists for the initial n queries (n is the number of backend application servers) where round-robin is used for selecting the backend application server. When all the backend servers have received at least one query to process, each will have intermediate results in its cache and an MBR for its local cache index. Using these initial MBRs, subsequent queries are forwarded to the server that requires the minimum enlargement of its current local MBR (measured by the difference in volumes of the old and new MBRs). In other words, this policy tries to keep the MBR of each backend server as small as possible to achieve good clustering of queries with MBRs that are "close" in the multidimensional space.

Index/Distance: This policy makes scheduling decisions based on the result of a global index lookup similarly to the Index/Overlap policy. However, instead of looking for the backend server whose MBR has the greatest degree of overlap with a query, the proxy attempts to locate a server whose local index root MBR is the closest to the query's MBR (measured as the Euclidean distance between the geometric centers of the two

[2] As will be seen in Section 7.6, we used a volumetric reconstruction application to provide the workload for our experiments. The experimental queries are reasonably homogeneous in terms of the amount of processing and I/O necessary to compute their results, which makes the queue size a good indication of the system load.

MBRs). This policy also attempts to assign approximately the same number of queries to each server. It does so by trying to keep the MBRs of the backend servers roughly the same sizes. For example, in Figure 7.12 the query is forwarded to server 2, which results in enlarging its MBR. For a query whose center falls between server 1 and 2's MBR centers, the proxy may forward the query to either one of them with the same probability. The intuition behind this policy is that relying purely on the amount of overlap will bias the proxy towards backend servers whose root MBRs are geometrically large, because a large MBR is likely to have greater overlap with any given query. Using the distance method contributes to removing the bias, while still maintaining the clustering property expected from Index/Overlap.

Index/Load: This policy considers the results of the global index lookup in conjunction with the current load associated with each of the candidate backend servers. Based on the waiting queue size, the proxy estimates the wait time a new query will *probably* experience. For backend servers that the global index indicates do not have relevant reusable aggregates, the proxy makes a pessimistic assumption that no new reusable aggregates will be materialized and all of the waiting queries will be computed from scratch. Conversely, for servers that are reported as having reusable aggregates, the estimate optimistically assumes that the computation time will be amortized by directly reusing those cached objects. From this assessment, the proxy selects the backend server with the smallest time estimate to process the query.

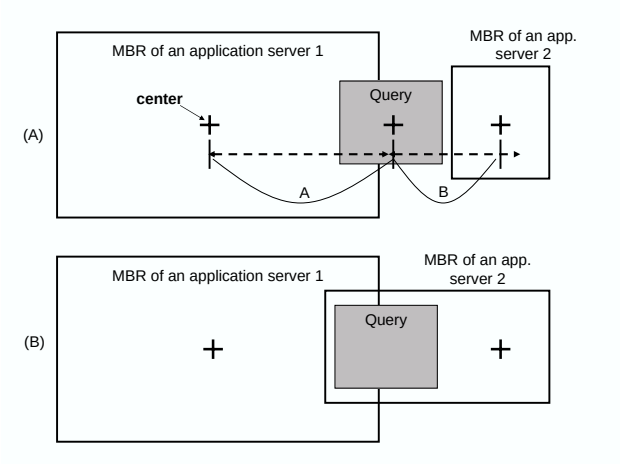


Figure 7.12: *Minimum distance policy*

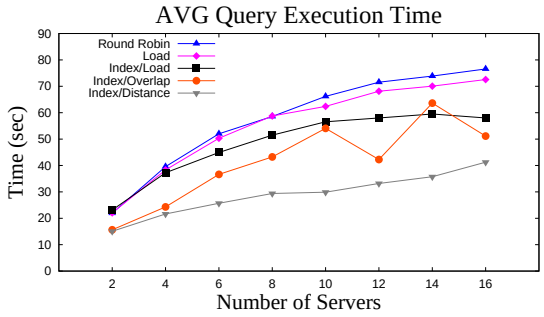
7.6 Experiments: Distributed Multiple Query Optimization

Improvements in planning and scheduling strategies are typically highly dependent on applications, system characteristics, and workloads. In order to shed light on the magnitude of improvements that can be expected by adopting distributed indexing, we performed experimental studies using a computationally intensive computer vision application, which can be seen as a representative example for many of the visualization techniques used by scientific applications.

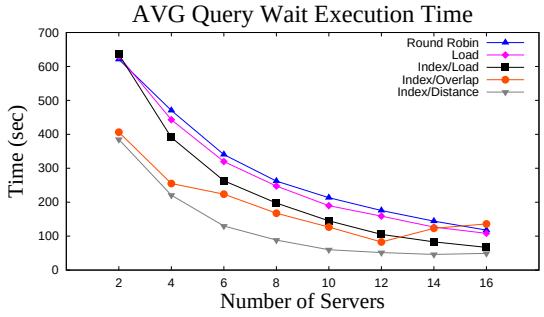
7.6.1 Experimental Environment

We employed an experimental configuration with 16 independent backend servers – i.e., full-fledged servers able to compute a volumetric reconstruction with access to replicas of the entire dataset – and a single proxy. Backend servers and the proxy were placed on different nodes of a Linux cluster. Each node is a Pentium III 650 MHz processor.

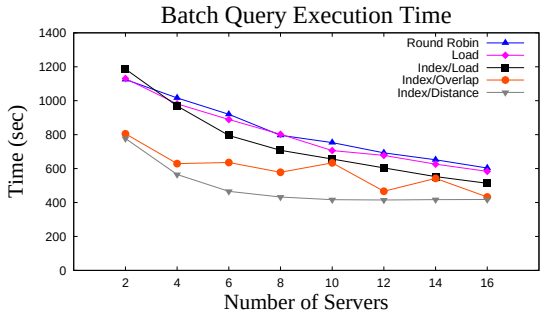
The nodes are connected by 100Mb/sec switched Ethernet.



(a) Query Execution Time (Average)



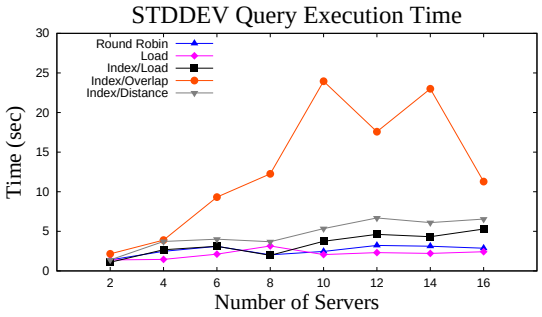
(b) Query Wait and Execution Time (Average)



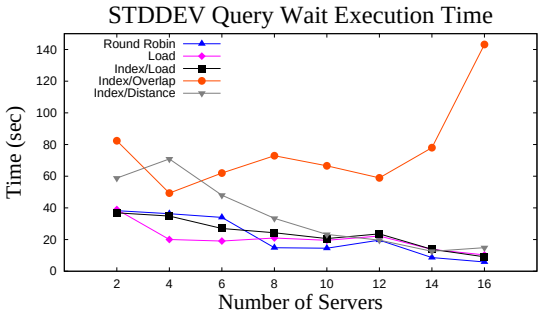
(c) Total Batch Query Time (Average)

Figure 7.13: *The Effect of Number of Servers*

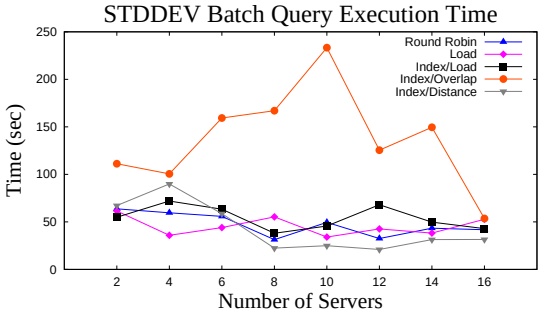
The dataset we used is a multi-perspective sequence of 2600 frames generated by 13



(a) Query Execution Time (Standard Deviation)



(b) Query Wait and Execution Time (Standard Deviation)



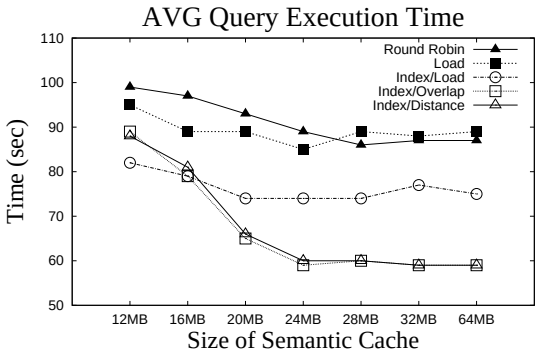
(c) Total Batch Query Time (Standard Deviation)

Figure 7.14: *The Effect of Number of Servers*

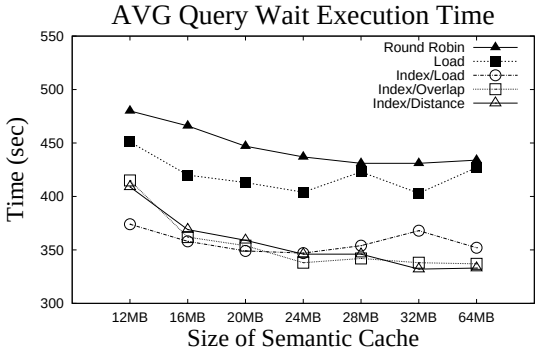
synchronized cameras, which we described in Chapter 3. The test dataset is partitioned into 32 silhouette image files (each file is 329 MB in size totaling about 10 GB). In order to evaluate the scheduling policies we replicated the datasets, thus each of the 16 backend servers stores the 10 GB dataset. Each of the 32 image files contains a collection of data chunks. A *chunk* of data is a single image whose attributes include a *camera index* and a *timestamp*.

We created 16 query batch files with the same query inter-arrival time for the experiments shown in Figure 7.13 and 8 query batch files with various query inter-arrival times for the experiments shown in Figure 7.17. Each batch file has 100 queries, simulating multiple simultaneous users posing queries to the system as a Poisson process. The queries in a batch were constructed according to a synthetic workload model since we do not have enough real user traces for the application. The workload generator emulates a hypothetical situation in which users want to view a short, multi-second 3D instant replay of *hot* events in, e.g., a basketball game. The workload generator takes as input parameters a set of "hot video frames" (e.g., slam dunks during the game) that mark the *interesting* scenes, and the length of a "hot interval" (i.e., the duration of the scene), characterized by a mean and a standard deviation.

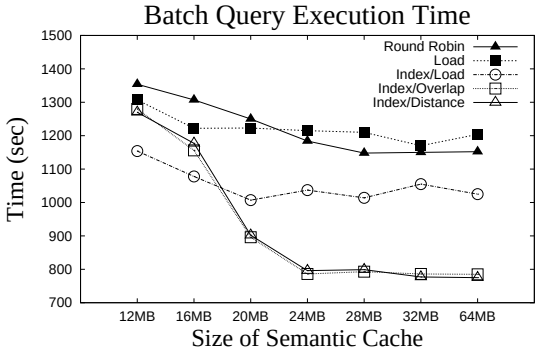
A query in a batch requests a set of reconstructions associated with frames selected according to the following model. The center of the interval is drawn randomly with a uniform distribution from the set of hot frames (10 hot frames were used). The length of the interval is selected from a normal distribution (each hot frame is associated with a mean video segment length, statistically varying from 34 to 62 frames). Between the first and last frame requested by a particular query, intermediate frames can be skipped, i.e., a query may process every frame, every 2nd frame, or every 4th frame. The skip factor is randomly selected. The 3-dimensional query box was also fixed (queries re-



(a) Query Execution Time



(b) Query Wait and Execution Time



(c) Total Batch Query Time

Figure 7.15: *The Effect of Semantic Cache Size*

construct the entire available volume) and the depth of an octree was 6, except for the experiments shown in Figure 7.16. Queries also used data from all the available cameras for reconstruction.

To measure performance, we considered the following metrics: *Query Wait and Execution Time* (QWET), *Query Execution Time* (QET), and *Total Batch Query Time* (TotalBQT). QWET is the amount of time from the moment a query is submitted to the system until it completes. That is, QWET includes the delay (due to the proxy being busy servicing other queries) plus the actual processing time. QET measures the elapsed time for a query to complete from the moment a backend server is selected until completion measured at the proxy. Hence QET depends on the local cache hit ratio, while QWET, to a greater degree, depends on load-balancing across the backend application servers. Finally, TotalBQT measures the total execution time for one query batch. From a user standpoint, lower QET and lower QWET implies faster query turnaround time. Lower TotalBQT implies higher query server throughput.

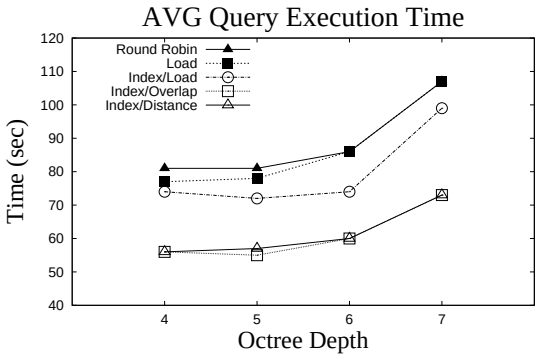
It should be noted that the MQO middleware has several control knobs. In order to focus on measuring the performance of the different scheduling policies without the influence of caching at the proxy, we disabled the semantic cache in the proxy and processed queries in FIFO order.

7.6.2 Experimental Results

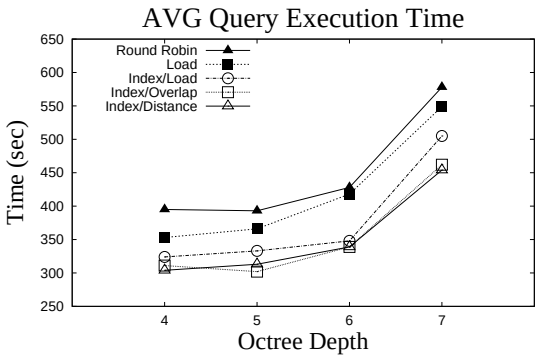
Figure 7.13 depicts system performance when we employed different query scheduling policies and varied the number of backend servers. Figure 7.13 shows the average execution time of 16 query batch files and Figure 7.14 shows the standard deviations across 16 query batches. For this experiment, we fixed the size of the semantic cache at 256MB and used LRU as the cache replacement policy on all backend servers. Each application

server employed a single thread for processing queries, since all the cluster nodes are uni-processors and would only marginally benefit from additional threads. However, for the front-end proxy, we varied the number of concurrent threads according to the number of application servers. For example, when 16 application servers are used, up to 16 threads are allowed in the proxy, which enables up to 16 queries to be simultaneously processed. Note that this does not imply that all 16 backend servers will be busy, i.e., multiple queries may be assigned to the same application server, depending on how good the scheduling policy is at load balancing.

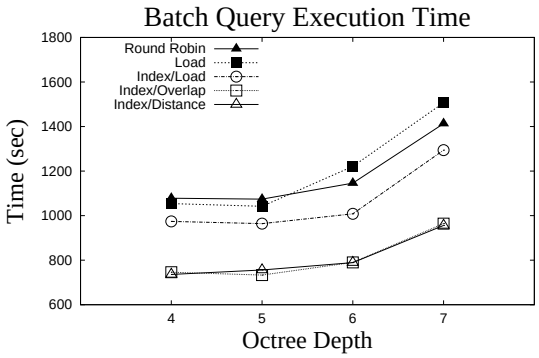
In general, as the number of application servers increases, frequently used cache objects are dispersed through the multiple backend server caches and the *per server* cache hit ratio drops. As a consequence, the average QET increases as more queries are computed from scratch without the benefit of caching as seen in Figure 7.13(a). Round-robin shows the worst performance in most cases. Load-based scheduling also does not show good performance, since neither policy considers the contents of the application server caches. As server caches get populated, the three index-based scheduling policies start to reap the benefits of increased cache hit rates, which causes decreased query execution time. An interesting result in Figure 7.13(a) is that the Index/Overlap policy does not show consistent performance due to load imbalance. As we discussed earlier, when the top-level MBR for a particular local index gets enlarged, the proxy becomes biased and chooses the backend server with the largest overlapping MBR. Thus, a majority of queries are forwarded to a single application server, which results in that server having a longer wait queue, increasing both QET and QWET. Note that QET includes the time waiting in the backend servers' queue, but not the time in the proxy's queue. Unlike Index/Overlap, the other two index-based policies – Index/Distance and Index/Load – manage to avoid such a load imbalance problem. Although Index/Load does not suffer



(a) Query Execution Time

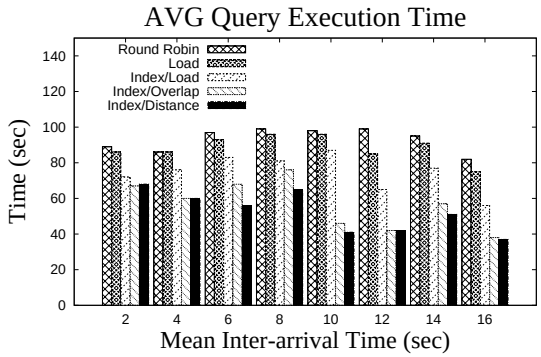


(b) Query Wait and Execution Time

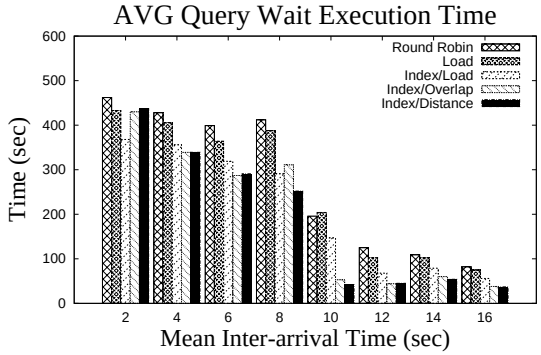


(c) Total Batch Query Time

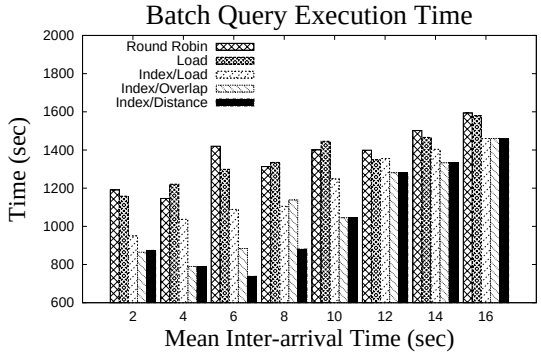
Figure 7.16: *The Effect of Octree Depth*



(a) Query Execution Time



(b) Query Wait and Execution Time



(c) Total Batch Query Time

Figure 7.17: *Workload Comparison*

from load imbalance, it tends to enlarge the local index MBRs leading to an increase in false hits, as the proxy does not take into consideration the clustering of cached aggregates. Occasionally, it creates large amount of dead space as opposed to Index/Distance and Index/Overlap, as those policies both favor not increasing the MBR. On the other hand, Index/Load benefits from bitmap encoding, which acts to mitigate the dead space problem as previously explained.

Figure 7.13(b) shows the query wait and execution time for the same experiment. As the number of servers increases, the average QWET seen by the proxy decreases as more queries can be executed concurrently. Note that while the QET improvements are not very large, hundreds of seconds are saved when measuring QWET and QBT due to more reuse. Similar to the QET result, Index/Distance outperforms the other policies consistently. As seen in Figure 7.13(c), the total batch query time when using Index/Distance is around 50% to 69% of the time when round-robin is employed. Figure 7.14(b) and 7.14(c) show that Index/Overlap has much higher standard deviation than the others. If query requests are fortunately distributed evenly, Index/Overlap achieves both high cache hit ratio and load balance. But if not, Index/Overlap performs even worse than round-robin due to load imbalance. The performance fluctuation is shown in Figure 7.13.

Figure 7.15 shows performance data for the scheduling policies as a function of the application servers' semantic cache sizes. For this experiment, we used 8 application servers and the proxy was configured with 8 threads. When the cache size is smaller than 24 MB, all policies suffer from a high rate of cache misses, since the cache cannot simultaneously accommodate many data products. In other words, the cache size is much smaller than the working set. In the experiments, the total size of the most frequently used cached aggregates was about 24 MB. Therefore, when the cache size

is smaller than that, queries may fail to find any cached aggregates at all. Because the round-robin and load-based policies are not targeted at maximizing reuse (although they may occasionally benefit from cache hits "by accident"), relatively speaking they are not severely impacted by small cache size (< 20 MB) nor do they particularly benefit from additional cache space. Since Index/Distance and Index/Overlap do not consider the size of the waiting queue, cache misses due to reduced cache size make the queries wait longer, which hurts overall system throughput. In such a case, Index/Load shows both the fastest query response time and the highest system throughput.

Figure 7.16 shows performance for the scheduling policies as the octree depth increases. The higher the depth, the more computationally expensive a query becomes due to the increased resolution of the volumetric reconstruction. Increased resolution translates into more space needed to compute and cache the results. Note that computational cost and memory requirement increase *exponentially* with octree depth. We ran 8 application servers, each with a 256 MB semantic cache. While we expected that the benefits from cache hits would have an exponential impact on the performance, because we kept the cache size fixed, we only observed a minor effect. Note that increased depth creates increased data product sizes, causing increased cache eviction activity and additional cache misses. In measuring system throughput, the performance gap between non-index based and index based policies increases slightly as the computation time increases. When the depth is 5, the total query batch time (QBT) with Index/Distance scheduling is 72% that of load-based scheduling, but it is 63% that of load-based scheduling when the depth is 7.

Finally, using the synthetic workload generator we described earlier, we created 8 different query workloads with different mean inter-arrival times to control the amount of concurrent load presented to the system. Note that the results for different workloads

depicted in Figure 7.17 are not directly comparable. Not only are the inter-arrival times different, but so are the the queries and the induced *workset* for caching. In other words, different queries have different cache hit rates, causing differences in processing time, which is unlikely to be a function of query inter-arrival time.

In this experiment, 8 application servers were used. As seen in Figure 7.17, the Index/Distance policy shows the best performance in most cases, with the other two index-based policies also outperforming the round-robin and load-based policies. In Figure 7.17(a), as expected, we see that QET is not greatly affected by the inter-arrival time. In measuring query wait time (Figure 7.17(b)), when the proxy server receives queries at a very high rate ($<$ 2 seconds on average between queries), Index/Load shows better performance than Index/Overlap and Index/Distance because of better load balancing. The query wait and execution time drops dramatically when the inter-arrival time is greater than 10 seconds, because the inter-arrival time becomes larger than the average query execution time (10 seconds $\times$ 8 servers = 80 $>$ QET). With large inter-arrival times, QWET has almost the same value as QET for a query, since almost no queries have to wait. In Figure 7.17(c), when the average inter-arrival time is greater than 12 seconds, we see that the total query batch time tends to stay around the same value, irrespective of the scheduling policy employed. This is because QBT only depends on the QET of the few last queries since the system is very lightly loaded.

7.7 Summary

To summarize, we have learned the following lessons from the experimental study. First, multidimensional indexing structures can help to efficiently find cached objects in a large semantic cache. Experimental results show that a cache index performs better than

linear scanning, and the performance benefits grow as the size of the semantic cache increases. Second, distributed indexing can help improve overall query processing performance, measured both by system throughput and by query response time. Third, load balancing is as important a factor in overall performance as cache hit rates for the distributed semantic caching infrastructure. Fourth, index-based scheduling that considers both load balancing and clustering properties (Index/Distance) tends to outperform less informed policies. Furthermore, Index/Distance policy is more stable, rarely performing badly compared to the policies that use less information.

Chapter 8

Conclusions and Future Work

In this Chapter, I conclude this dissertation by reviewing the thesis and its contributions and present some directions for future work.

8.1 Thesis and Contributions

In this dissertation, I supported the following thesis: *distributed multidimensional indexing can greatly improve access to distributed large scientific datasets.* Numerous emerging scientific data analysis applications need the support of a distributed multidimensional indexing service. The goal of this work was to investigate the problem of indexing large distributed scientific datasets for scientific data analysis applications, to provide efficient multidimensional indexing techniques that aid in navigating distributed scientific datasets, and to show significant improvements in accessing distributed large scientific datasets. The main contributions made by this dissertation include:

An approach to increase the efficiency of indexing via data chunking

In order to accelerate search and update performance of indexing, I defined a *logical data chunking* concept that groups data elements and store a bounding box for each

chunk instead of for each element. In scientific datasets, data elements that are nearby in a stored array (i.e. their indices are close) usually are also nearby in spatio-temporal coordinates, because the sensor or simulation data is stored in the same order it is acquired or produced. By grouping data elements into chunks, we can get a relatively tight bounding box for the spatio-temporal coordinates (meaning that the boxes for different chunks do not overlap much). Data chunking may cause data elements not within the requested query range to be retrieved, because if the bounding box of a data chunk overlaps the query range all the elements in the chunk must be accessed. There is therefore an overhead from data chunking - filtering out data elements not within the query range after they are read from disk. However, I have shown that the overhead of filtering the additional data elements is negligible

A design of an efficient indexing structure for chunked datasets

I have designed an efficient multidimensional indexing structure for rectangular data objects (chunked datasets), which supports fast insertion/deletion as well as fast search. Little emphasis has been laid upon the performance of multidimensional index inserts and deletes, as opposed to search performance. The *SH-tree* I developed is a disk based space partitioning indexing structure for rectangular data objects, which has dimension independent tree height and low insert/delete algorithmic complexity. I compared the performance of a few widely used multidimensional indexing structures with SH-trees, looking at insert, delete, and search operations, and showed that SH-trees overall perform better than the widely used indexing techniques.

A set of techniques for distributing index for distributed datasets

I developed and compared three approaches to distribute a multidimensional index - replicated centralized indexing, hierarchical two level indexing, and decentralized two level indexing. The experimental study demonstrated 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.

Analyzing the design choices that affect the performance of distributed indexing

This dissertation explored and compared the designs, challenges, and problems for distributed multidimensional indexing schemes, and also provided a comprehensive performance study of distributed indexing to provide guidelines to choose a distributed multidimensional index for a specific application.

As a case study application, I have described how hierarchical two level indexing scheme can be used by a distributed multiple query optimization middleware system to generate better query plans, leveraging information about the contents of remote semantic caches. Experimental results obtained using a computer vision application showed that employing this information for query scheduling results in both lower query response time and better system throughput than round-robin or load-based scheduling. I believe this is the first work that showed that distributed multidimensional indexing helps improve query processing performance for a real distributed query processing system.

8.2 Future Work

We foresee many possible extensions to the work presented in this dissertation. Although I showed significant improvements by employing the distributed indexing schemes, many improvements can still be made.

Performance study using more applications

I intend to extend this work using more various data analysis applications as well as different workload profiles in WAN-based heterogeneous environments. In this dissertation, I evaluated hierarchical two level indexing in the context of multiple query optimization framework, but there are other static data analysis applications where replicated centralized indexing or decentralized indexing is more appropriate than hierarchical two level indexing, as we discussed in Chapter 6.

Replica management of distributed index

I showed that it is sometimes desirable to create remote copies of indexes in wide area systems, since replication reduces access latency, improves data locality, and increases robustness and scalability. Since any system that has replicas needs a mechanism for creating, deleting, and locating replicas, we need to further investigate how many replicas should be either created or deleted, where to create new replicas, and which replica should be accessed to process a certain query. We have shown that more replicas accelerate search performance, but worsen update performance. The basic idea of the index replication mechanism should be that when read requests are dominating the system, more replicas must be created to reduce the query response time, and when write requests are dominant, some number of replicas must be deleted to reduce the update overhead for maintaining consistency between replicas. The decisions can be made

based on a few factors such as read/write statistics, network latency, response time, bandwidth, and index size.

Indexing service Grid-ification

It seems very likely that distributed multidimensional indexing techniques will be part of the Grid infrastructure. In such an environment, a set of interfaces and protocols have to be provided for a user to launch applications that will use the distributed indexing service. In fact, users should not be concerned with where datasets are located as long as they can access them through distributed indexing service provided by Grid. Supplying these interfaces for distributed indexing services is a potentially very useful extension of this work.

Grid resource matchmaking

Distributed multidimensional indexing service also can be used as a resource matchmaker in Grid. 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. 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. The distributed indexing schemes that I presented in this dissertation need to be investigated to see how to make them suitable for Grid matchmaking, otherwise a further research has to be done to extend this work.

BIBLIOGRAPHY

[1] R-tree Portal. *http://www.rtreeportal.org*.

[2] Planetary data system data preparation workbook. Technical Report JPL D-7669, Part I, Jet Propulsion Laboratory, California Institute of Technology, February 1995.

[3] M. Aeschlimann, P. Dinda, L. Kallivokas, J. L´opez, B. Lowekamp, and D. O'Hallaron. Preliminary report on the design of a framework for distributed visualization. In *Proceedings of the Parallel and Distributed Processing Techniques and Applications (PDPTA99)*, Las Vegas, NV, 1999.

[4] A. Afework, M. Beynon, F. Bustamante, A. Demarzo, R. Ferreira, R. Miller, M. Silberman, J. Saltz, A. Sussman, and H. Tsang. Digital dynamic telepathology - the Virtual Microscope. In *Proceedings of the 1998 AMIA Annual Fall Symposium*. American Medical Informatics Association, Nov. 1998.

[5] W. Allcock, J. Bester, J. Bresnahan, A. Chervenak, I. Foster, C. Kesselman, S. Meder, V. NEfedova, D. Quesnel, and S. Tuecke. Secure, effi cient data transport and replica management for high-performance data-intensive computing. In *IEEE Mass Storage Conference*, 2001.

[6] M. Alpdemir, A. Mukherjee, A. Gounaris, N. Paton, P. Watson, A. Fernandes, and D. Fitzgerald. Ogsa-dqp: A service for distributed querying on the grid. In *Proceedings of the Advances in Database Technology*, 2004.

[7] H. Andrade. *Multiple Query Optimization Support for Data Analysis Applications*. PhD thesis, Department of Computer Science, University of Maryland, Dec. 2002.

[8] H. Andrade, T. Kurc, A. Sussman, E. Borovikov, and J. Saltz. On cache replacement policies for servicing mixed data intensive query workloads. In *Proceedings of the 2nd Workshop on Caching, Coherence, and Consistency, held in conjunction with the 16th ACM International Conference on Supercomputing*, New York, NY, June 2002.

[9] H. Andrade, T. Kurc, A. Sussman, and J. Saltz. Efficient execution of multiple query workloads in data analysis applications. In *Proceedings of the ACM/IEEE SC2001 Conference*, Nov. 2001.

[10] H. Andrade, T. Kurc, A. Sussman, and J. Saltz. Active Proxy-G: Optimizing the query execution process in the Grid. In *Proceedings of the ACM/IEEE SC2002 Conference*, Nov. 2002.

[11] H. Andrade, T. Kurc, A. Sussman, and J. Saltz. Optimizing the execution of multiple data analysis queries on parallel and distributed environments. *IEEE Transactions on Parallel and Distributed Systems*, 15(6):520–532, June 2004.

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

[13] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger. The R^*-tree: An efficient and robust access method for points and rectangles. In *Proceedings of 1990 ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 322–331, May 1990.

[14] J. L. Bentley. Multidimensional binary search trees used for associative searching. In *Communications of the ACM 18(9)*, 1975.

[15] S. Berchtold, D. A. Keim, and H.-P. Kriegel. The X-tree: An index structure for high-dimensional data. In *Proceedings of the 22th International Conference on Very Large Data Bases (VLDB)*, pages 28–39, 1996.

[16] M. Beynon, C. Chang, U. Catalyurek, T. Kurc, A. Sussman, H. Andrade, R. Ferreira, and J. Saltz. Processing large-scale multidimensional data in parallel and distributed environments. *Parallel Computing*, 28(5):827–859, May 2002. Special issue on Data Intensive Computing.

[17] M. D. Beynon, R. Ferreira, T. Kurc, A. Sussman, and J. Saltz. DataCutter: Middleware for filtering very large scientific datasets on archival storage systems. In *Proceedings of the Eighth Goddard Conference on Mass Storage Systems and Technologies/17th IEEE Symposium on Mass Storage Systems*, pages 119–133. National Aeronautics and Space Administration, Mar. 2000. NASA/CP 2000-209888.

[18] M. D. Beynon, T. Kurc, U. Çataly¨urek, C. Chang, A. Sussman, and J. Saltz. Distributed processing of very large datasets with DataCutter. *Parallel Computing*, 27(11):1457–1478, Oct. 2001.

[19] C. B¨ohm, S. Berchtold, and D. Keim. Searching in high-dimensional spaces – index structures for improving the performance of multimedia databases. *ACM Computing Surveys*, 33(3):322–373, Sept. 2001.

[20] E. Borovikov, A. Sussman, and L. Davis. An efficient system for multi-perspective imaging and volumetric shape analysis. In *Proceedings of the 2001 Workshop on Parallel and Distributed Computing in Imaging Processing, Video Processing, and Multimedia*, San Francisco, CA, 2001.

[21] E. Borovikov, A. Sussman, and L. Davis. A high performance multi-perspective vision studio. In *Proceedings of the 17th ACM International Conference on Supercomputing (ICS)*, 2003.

[22] K. Chakrabarti and S. Mehrotra. The Hybrid tree: An index structure for high dimensional feature spaces. In *Proceedings of the 15th International Conference on Data Engineering (ICDE)*, pages 440–447, 1999.

[23] U. S. Chakravarthy and J. Minker. Multiple query processing in deductive databases using query graphs. In *Proceedings of the 12th International Conference on Very Large Data Bases (VLDB)*, pages 384–391, 1986.

[24] C. Chang, R. Ferreira, A. Sussman, and J. Saltz. Infrastructure for building parallel database systems for multi-dimensional data. In *Proceedings of the 13th International Parallel Processing Symposium*. IEEE Computer Society Press, Apr. 1999.

[25] C. Chang, B. Moon, A. Acharya, C. Shock, A. Sussman, and J. Saltz. Titan: A high performance remote-sensing database. In *Proceedings of the 13th International Conference on Data Engineering (ICDE)*, pages 375–384, 1997.

[26] E. Ch´avez, G. Navarro, R. Baeza-Yates, and J. Marroqu´ı n. Searching in metric spaces. *ACM Computing Surveys*, 33(3):273–321, Sept. 2001.

[27] 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.

[28] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. The MIT Press, 2nd edition, 2001.

[29] 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.

[30] E. W. G. D. C. Wells and R. H. Harten. FITS: A flexible image transport system. *Astronomy and Astrophysics Supplement Series*, 44:363–370, 1981.

[31] S. Dar, M. J. Franklin, B. T. Jonsson, D. Srivastava, and M. Tan. Semantic data caching and replacement. In *Proceedings of the 22th International Conference on Very Large Data Bases (VLDB)*, pages 330–341, 1996.

[32] P. J. Denning. The working set model for program behavior. *Communications of the ACM*, 11(5):323–333, May 1968.

[33] M. Folk. A White Paper: HDF as an Archive Format: Issues and Recommendations, January 1998. *http://hdf.ncsa. uiuc.edu/archive/hdfasarchivefmt.htm*.

[34] 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 (WebDB)*, pages 19–24. LNCS, 2004.

[35] B. Gedik and L. Liu. MobiEyes: Distributed processing of continuously moving queries on moving objects in a mobile system. In *Proceedings of the 9th International Conference on Extending Databases Technology (EDBT)*, 2004.

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

[37] P. Godfrey and J. Gryz. Answering queries by semantic caches. In *Proceedings of the 10th International Conference on Database and Expert Systems Applications (DEXA)*, pages 485–498, 1999.

[38] L. Gosink, J. Shalf, K. Stockinger, K. Wu, and W. Bethel. Hdf5-fastquery. In *Proceedings of 18th International Conference on Scientific and Statistical Database Management (SSDBM)*, 2006.

[39] J. Gray, D. T. Liu, M. Nieto-Santisteban, A. Szalay, D. Dewitt, and G. Heber. Scientific data management in the coming decade. *SIGMOD Record*, 34(4), Dec. 2005.

[40] 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.

[41] D. Hagimont and D. Louvegnies. Javanaise: Distributed shared objects for internet cooperative applications. In *IFTP International Conference on Distributed Systems, Platforms, and Open Distributed Processing Middleware*, 1998.

[42] N. J. Harvey, M. B. Jones, S. Saroiu, M. Theimer, and A. Wolman. SkipNet: A scalable overlay network with practical locality properties. In *Proceedings of the 4th USENIX Symposium on Internet Technologies and Systems (USITS '03)*, 2003.

[43] P. Havlak and K. Kennedy. An implementation of interprocedural bounded regular section analysis. *IEEE Transactions on Parallel and Distributed Systems*, 2(3):350–360, July 1991.

[44] 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.

[45] A. Jhingran. A performance study of query optimization algorithms on a database system supporting procedures. In *Proceedings of the 14th International Conference on Very Large Data Bases (VLDB)*, pages 88–99, 1988.

[46] 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.

[47] M. H. Kang, H. G. Dietz, and B. K. Bhargava. Multiple-query optimization at algorithm-level. *Data and Knowledge Engineering*, 14(1):57–75, 1994.

[48] N. Katayama. HnRStar tree library ver. 1.0. *http://research.nii.ac.jp/˜katayama/homepage/research/srtree*.

[49] J.-S. Kim, H. Andrade, and A. Sussman. Comparing the performance of high-level middleware systems in shared and distributed memory parallel environments. In *Proceedings of 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE Computer Society Press, Apr. 2005.

[50] 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.

[51] 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.

[52] 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.

[53] T. Kurc, U. Catalyurek, X. Zhang, J. Saltz, M. Peszynska, R. Martino, M. Wheeler, A. Sussman, C. Hansen, M. Sen, R. Seifoullaev, P. Stoffa, C. Torres-Verdin, and M. Parashar. A simulation and data analysis system for large scale, data-driven oil reservoir simulation studies. *Concurrency and Computation: Practice and Experience*, 2005. To appear.

[54] T. Kurc, C. Chang, R. Ferreira, A. Sussman, and J. Saltz. Querying very large multi-dimensional datasets in ADR. In *Proceedings of the ACM/IEEE SC1999 Conference*. ACM Press, Nov. 1999.

[55] 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.

[56] Larry Klein. An HDF-EOS and Data Formatting Primer , March 2001. *http://edhs1.gsfc.nasa.gov/waisdata/sdp/pdf/wp1750102.pdf*.

[57] J. Liebeherr, E. Omiecinski, and I. Akyildiz. The effect of index partitioning schemes on the performance of distributed query processing. *IEEE Transactions on Knowledge and Data Engineering*, 5(3), 1993.

[58] 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.

[59] D. Lomet and B. Saltzberg. The hB-tree: A multiattribute indexing method with good guaranteed performance. *ACM Transactions on Database Systems*, 15(4), 1990.

[60] T. Matsuyama, L. Hao, and M. Nagao. A file organization for geographic information systems based on the spatial proximity. In *Computer Vision, Graphics and Image Processing. Vol. 26, No.3*, pages 303–318, 1984.

[61] D. Menasce and V. A. F. Almeida. *Scaling for E-Business: Technologies, Models, Performance, and Capacity Planning*. Prentice Hall PTR, 2000.

[62] M. F. Mokbel, X. Xiong, M. A. Hammad, and W. G. Aref. Continuous query processing of spatio-temporal data streams in PLACE. In *Proceedings of the 2nd Workshop on Spatio-temporal Databases Management (STDBM)*, 2004.

[63] A. Mondal, M. Kitsuregawa, B. C. Ooi, and K. L. Tan. R-tree-based data migration and self-tuning strategies in shared-nothing spatial databases. In *ACM Proceedings of the 9th international symposium on Advances in Geographic Information Systems (GIS)*, pages 28–33, 2001.

[64] A. Mondal, Yilifu, and M. Kitsuregawa. P2PR-tree: An R-tree-based spatial index for peer-to-peer environments. In *Proceedings of the 1st international workshop on P2P*

Computing and Databases, 2004.

[65] B. Nam and A. Sussman. Improving access to multi-dimensional self-describing scientific datasets. In *Proceedings of the 3rd IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid)*, May 2003.

[66] 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.

[67] 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.

[68] 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.

[69] 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*.

[70] NCSA, University of Illinois. HDF User's Guide, version 4.1r5, November 2001. *ftp://ftp.ncsa.uiuc.edu/HDF/HDF /Documentation/HDF4.1r5/Users_Guide*.

[71] NCSA, University of Illinois. Introduction to HDF5, April 2001. *http://hdf.ncsa.uiuc.edu/HDF5/doc/H5.intro.html*.

[72] NOAA, National Climatic Data Center. NOAA KLM user's guide section 3.1, September 2000. *http://www2.ncdc.noaa.gov/docs/klm/html/c3/sec3-1.htm*.

[73] NOAA satellite and information service. Advanced Very High Resolution Radiometer - AVHRR, 2005. *http://noaasis.noaa.gov/NOAASIS/ml/avhrr.html*.

[74] B. C. Ooi, R. Sacks-Davis, and K. J. McDonell. Spatial k-d-tree: An indexing mechanism for spatial databases. In *IEEE COMPSAC Conference*, 1987.

[75] F. Özcan and V. Subrahmanian. Partitioning activities for agents. In *Proceedings of the 2001 International Joint Conferences on Artificial Intelligence*, Seattle, WA, 2001.

[76] G. Popek, B. Walker, J. Chow, D. Edwards, C. Kline, G. Rudisin, and G. Thiel. Locus: A network transparent high reliability distributed system. In *the 8th Symposium on Operating Systems Principle*, pages 169–177, 1981.

[77] 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.

[78] 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.

[79] K. Ranganathan and I. Foster. Design and evaluation of replication strategies for a high performance data grid. In *International Conference on Computing in High Energy and Nuclear Physics*, 2001.

[80] 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.

[81] Raytheon Systems Co., MD. HDF-EOS Users Guide for the ECS Project, Volume I: Overview and Example, November 2001. *http://hdfeos.gsfc.nasa.gov/hdfeos/Docs/HDFEOSv2. x_UG_vol1.pdf*.

[82] R. Rew, G. Davis, and S. Emmerson. NetCDF User's Guide for C, 1997. *http://www.unidata.ucar.edu/packages /netcdf/cguide.pdf*.

[83] J. T. Robinson. The K-D-B tree: A search structure for large multi-dimensional dynamic indexes. In *Proceedings of 1981 ACM SIGMOD International Conference on Management of Data (SIGMOD)*, 1981.

[84] M. Rodr´ı guez-Mart´ı nez and N. Roussopoulos. MOCHA: A self-extensible database middleware system for distributed data sources. In *Proceedings of 2000 ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 213–224. ACM Press, May 2000. ACM SIGMOD Record, Vol. 29, No. 2.

[85] A. Rosenthal and U. S. Chakravarthy. Anatomy of a modular multiple query optimizer. In *Proceedings of the 14th International Conference on Very Large Data Bases (VLDB)*, pages 230–239, 1988.

[86] H. Samet. *The Design and Analysis of Spatial Data Structures*. Addison-Wesley, 1990.

[87] M. Satyanarayanan, J. Kister, P. Kumar, M. Okasaki, E. Siegel, and D. Steere. Coda: A highly available file system for a distributed workstation environment. *IEEE Transactions on Computers*, 39(4):447–459, 1990.

[88] 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.

[89] J. Schopf, M. D'Arcy, N. Miller, L. Pearlman, I. Foster, and C. Kesselman. Monitoring and discovery in a web services framework: Functionality and performance of the globus toolkit's mds4. Technical Report ANL/MCS-P1248-0405, Argonne National Laboratory, Apr. 2005.

[90] Seagate. Barracuda 7200.100 – Data Sheet. http://www.seagate.com/docs/pdf/marketing/po_barracuda_7200_10.pdf.

[91] T. K. Sellis. Multiple-query optimization. *ACM Transactions on Database Systems*, 13(1):23–52, 1988.

[92] T. K. Sellis and S. Ghosh. On the multiple-query optimization problem. *IEEE Transactions on Knowledge and Data Engineering*, 2(2):262–266, 1990.

[93] T. K. Sellis, N. Roussopoulos, and C. Faloutsos. The R+-tree: A dynamic index for multi-dimensional objects. In *Proceedings of the 13th International Conference on Very Large Data Bases (VLDB)*, pages 507–518, 1987.

[94] K. Shim, T. K. Sellis, and D. Nau. Improvements on a heuristic algorithm for multiple query optimization. *Data and Knowledge Engineering*, 12:197–222, 1994.

[95] C. T. Shock, C. Chang, B. Moon, A. Acharya, L. Davis, J. Saltz, and A. Sussman. The design and evaluation of a high-performance earth science database. *Parallel Computing*, 24(1):65–90, Jan. 1998.

[96] N. M. Short. Remote Sensing Tutorial, 2006. *http://rst.gsfc.nasa.gov*.

[97] S. Song, Y. Kim, and J. Yoo. An enhanced concurrency control scheme for multidimensional index structure. *IEEE Transactions on Knowledge and Data Engineering*, 16(1):97–111, Jan. 2004.

[98] 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.

[99] M. S. Sunita Sarawagi. Effi cient organization of large multi-dimensional arrays. In *Proceedings of the Tenth International Conference on Data Engineering*, pages 328–336, February 1994.

[100] 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.

[101] 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.

[102] R. Weber, H.-J. Schek, and S. Blott. A quantitative analysis and performance study for similarity-search methods in high-dimensional spaces. In *Proceedings of the 24th International Conference on Very Large Data Bases (VLDB)*, 1998.

[103] D. Wessels and K. C. Claffy. ICP and the Squid web cache. *IEEE Journal on Selected Areas in Communications*, 16(3):345–357, Apr. 1998.

[104] M. Wiesmann, F. Pedone, A. Schiper, B. Kemme, and G. Alonso. Understanding replication in databases and distributed systems. In *20th International Conference on Distributed Computing Systems (ICDCS)*, Apr. 2000.

[105] O. Wolfson and S. Jajodia. Distributed algorithms for dynamic replication of data. In *PODS '92: Proceedings of the eleventh ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, pages 149–163, 1992.

[106] O. Wolfson, S. Jajodia, and Y. Huang. An adaptive data replication algorithm. *ACM Transactions on Database Systems*, 22(2):255–314, 1997.

[107] X. Xiong, M. F. Mokbel, W. G. Aref, S. E. Hambrusch, and S. Prabhakar. Scalable spatio-temporal continuous query processing for location-aware services. In *Proceedings of 16th International Conference on Scientifi c and Statistical Database Management (SSDBM)*, 2004.

[108] 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.

[109] K. Zhang, H. Andrade, L. Raschid, and A. Sussman. Query planning for the Grid: Adapting to dynamic resource availability. In *Proceedings of the 5th IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid)*, Cardiff, UK, May 2005.

[110] X. Zhang, J. L. Freschl, and J. M. Schopf. Scalability analysis of three monitoring and information systems: Mds2, r-gma and hawkeye. In *IEEE Transactions on Parallel and Distributed Systems*, June 2006.

[111] Z. Zhang, J. J´aJ´a, 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, Nov. 1999.

[112] R. Zimmermann, W.-S. Ku, and W.-C. Chu. Effi cient query routing in distributed spatial databases. In *ACM 12th International Symposium on Advances in Geographic Information Systems (GIS)*, 2004.