Multiple Range Query Optimization with Distributed Cache Indexing *

Beomseok Nam†, Henrique Andrade‡, Alan Sussman†

† Dept. of Computer Science
University of Maryland
College Park, MD 20742
{bsnam,als}@cs.umd.edu

‡ IBM T. J. Watson Research Center
19 Skyline Drive
Hawthorne, NY 10598
hcma@us.ibm.com

Abstract

MQO is a distributed multiple query processing middleware that can use resources available on the Grid to optimize query processing for data analysis and visualization applications. It does so by introducing one or more proxies that act as front-ends to a collection of backend servers. The basic idea behind this architecture is *active semantic caching*, whereby queries can leverage available cached results in the proxy either directly or through transformations. While this approach has been shown to speed up query evaluation under multi-client workloads, the caching infrastructure in the backend servers is not used well for query processing. Because this collective caching infrastructure scales with the number of servers, it is an important asset. In this paper, we describe a distributed multidimensional indexing scheme that enables the proxy to directly consider the cache contents available at the backend servers for query planning and scheduling. This approach is shown to produce better query plans and faster query response times as we experimentally demonstrate.

1 Introduction

Multiple query optimization has been extensively studied in various contexts including relational databases and data analysis applications [Andrade et al. 2002; Dar et al. 1996; Godfrey and Gryz 1999; Sellis and Ghosh 1990]. The objective is to exploit processing commonality across a set of concurrently executing queries and reduce execution time by reusing previously computed results. Although finding a globally optimal query plan was shown to be an NP-complete problem [Sellis and Ghosh 1990], heuristics can generate good plans. We have developed middleware aimed at aiding and optimizing the development of applications that process multi-query workloads [Andrade et al. 2004]. This middleware 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 [Andrade et al. 2002] 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), our 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.

Another 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 [Andrade et al. 2004]. 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

*This research was supported by the National Science Foundation under Grants #EIA-0121161 and #CNS-0540216, and NASA under Grant #NNG06GE75G. SC2006 November 2006, Tampa, Florida, USA 0-7695-2700-0/06 $20.00

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.

There has been extensive research on indexing data structures in the past, starting with the seminal work on R-trees [Guttman 1984]. On the other hand, relatively little effort has been devoted to designing *distributed* indexing schemes. Recently we have studied several distributed multidimensional indexing schemes, including replicated indices, hierarchical indices, and decentralized indices [Nam and Sussman 2005; Nam and Sussman 2006]. For relatively stable configurations (few index updates), the simplest way to distribute the index is to replicate it onto multiple servers. For dynamically changing index contents, a better method consists of partitioning the index and storing the pieces on multiple servers in a hierarchical fashion, as we will describe in Section 4. Moreover, in order to make the indexing more scalable, maintenance of the top-level index for hierarchical indexing can be decentralized [Nam and Sussman 2006]. In this paper 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 this issue in the context of a computationally expensive computer vision application.

We believe that the work presented in this paper makes several contributions and extends the state-of-the-art in the area of distributed query processing. The most important contribution is the integration of distributed query planning with distributed indexing, which results in improvements both in query processing time as well as increased system throughput. The rest of the paper is organized as follows. In Section 2 we discuss other research related to distributed indexing and multiple query optimization. In Section 3 we describe the architecture of the MQO middleware. We discuss how distributed indexing was integrated into the query evaluation process in Section 4. In Section 5 we describe experimental results for several query scheduling approaches, including index-based ones, and examine the costs and savings of indexing measuring both query execution and waiting time, as well as batch execution time. Finally, in Section 6, we make concluding remarks and explore possible extensions to this work.

2 Related Work

To the best of our knowledge, this paper is the first attempt to combine distributed multidimensional indexing with distributed query processing. The problem of integrating distributed query planning with distributed indexing and caching intersects many research areas. From distributed query processing to earlier efforts on managing distributed indices, substantial research has been done. In this section, we highlight some of the work we deem most relevant to our own effort.

For distributed query processing, Rodríguez-Martínez and Roussopoulos [Rodríguez-Martínez and Roussopoulos 2000] 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. When data-shipping is not an option due to the size of the datasets, distributed applications have employed proxy front-ends. Beynon et. al. [Beynon et al. 2002] 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 [Wessels and Claffy 1998], 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 seamlessly integrate multiple backend servers as a single query server, it is necessary to efficiently index the data (cached or otherwise) that each of them has access to. The R-tree was one of the first multidimensional object indexing data structures to be developed [Guttman 1984]. Kamel and Faloutsos [Kamel and Faloutsos 1992] extended that work, by proposing parallel R-trees (Multiplexed R-trees). One of the limitations of that approach was that it targets a single CPU with multiple disks. That limitation was overcome by Master R-trees [Koudas et al. 1996] and Master Client R-trees [Schnitzer and Leutenegger 1999], both designed for shared nothing environments (i.e., distributed memory parallel machines). Both approaches assume that datasets are declustered using a space filling curve and relative stability of the indexed datasets. Both assumptions may not hold true in scenarios where distributed dynamic caches are indexed and updated frequently.

In order to effectively leverage multiple backend servers for query processing, methods for load balancing must be considered as we previously demonstrated through simulation [Zhang et al. 2005]. 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 [Mondal et al. 2001]. Their approach for load balancing is different from ours in that our work in-

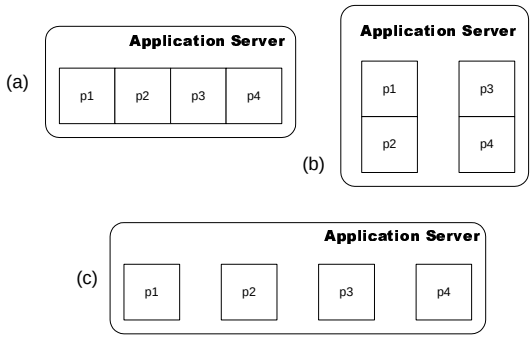


Figure 1: *Application Servers with different parallel configurations. (a) shared memory, (b) distributed shared memory, or (c) distributed memory*

vestigates query assignment policies instead of moving input datasets around to improve load balance.

3 Improving the Multiple Query Processing Middleware

Over the last few years we have designed and built middleware to support large scale data analysis applications [Andrade et al. 2004; Beynon et al. 2001; Beynon et al. 2002]. The Multi-Query Optimization [Andrade et al. 2004] middleware (MQO) is one of the results of this effort. MQO provides an environment based on C++ abstract operators that are customized when new applications are first developed and implemented or when existing applications are ported. 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 discussion, 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 provide the context for the integration of distributed cache indexing capabilities into the middleware.

MQO's Grid configuration consists of a proxy service (one or more APGs), an application query processing service (one or more application backend servers), and a data caching service (one or more cache servers) as shown in Figure 2. Note that application and cache servers can run in the same address space as a single component. 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 1 graphically depicts these different configurations. APG enables the backend application servers to be distributed and connected in any hierarchy forming a computational Grid as shown in Figure 2.

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 [Andrade et al. 2002]: (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

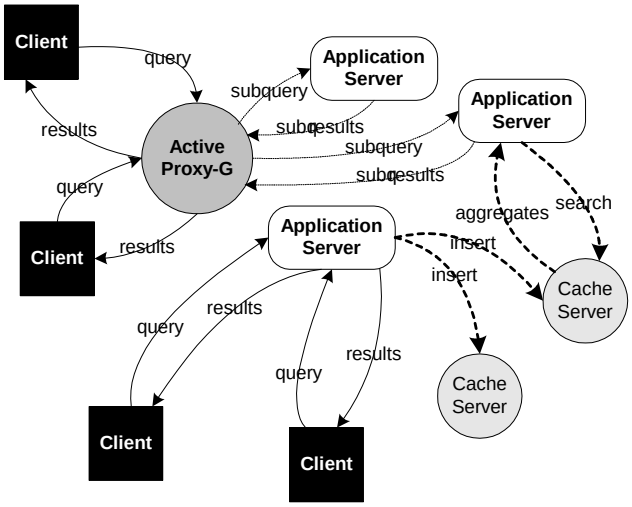


Figure 2: *Overview of the MQO middleware*

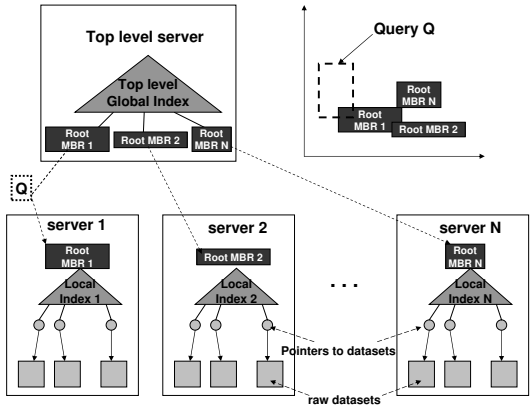


Figure 3: *A two-level hierarchical index*

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 [Kim et al. 2005]. 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.

4 Distributed Indexing

A multidimensional index enables update and search operations to be performed in parallel, thus providing the means for distributing the load across multiple servers. There are several ways to implement a distributed semantic cache index, depending on workload characteristics. The best choice depends on the type of index operations to be performed frequently (e.g., lookups, inserts, deletes) as well as the nature of the expected cache contents. In previous work [Nam and Sussman 2005; Nam and Sussman 2006], we have studied three types of distributed indexing schemes: index replication, hierarchical indexing, and decentralized indexing. Each of them addresses 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.

The hierarchical indexing scheme partitions the index and distributes it across multiple servers. This is accomplished by creating a two-level hierarchy, where a *global* index stores only the minimum bounding rectangles (MBRs[1]) representing the root node of each *local* index.

Data analysis queries typically have at least two components forming the query predicate: one that specifies the kind of processing necessary in order to generate the desired data product (e.g., sampling, aggregation, filtering, geometric transformation, etc.), and another that specifies the spatial domain, usually in the form of an MBR, i.e., a minimum bounding rectangle representing latitude/longitude ranges, 3-D spatial coordinates, etc. Thus, not only is the final data product associated with multidimensional coordinates, but so are the intermediate aggregates computed as a query is processed. All of these aggregates are automatically cached and indexed by the middleware.

Figure 3 depicts a sample internal organization of a hierarchical index. To search the index, the multidimensional predicate of a query is presented to the *global index* in order to determine which local index (or indices) may contain objects

[1]An MBR is a multidimensional hypercube that encompasses all the multidimensional data objects (points or hyper-rectangles) stored in the subtrees rooted at any given node in the tree-based index.

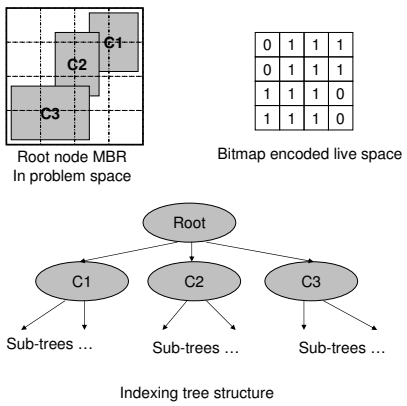


Figure 4: *Bitmap live space encoding*

relevant to the query. Each data server has its own index for the data available locally (*local index*). The comparison between the query's MBR and those MBRs defining the local indices results in the list of candidate backend servers.

Since the MBR for a local index is just approximate information about the data stored in a backend server, it is possible that the search on the global index will return, in addition to servers that have aggregates satisfying the query, one or more backend servers that do not have any data objects that overlap the query range. This may occur for several reasons, for example, the space within an MBR may not be densely populated or objects may have been removed.

Integrating distributed 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 (Spatial Hybrid trees) for the local index. SH-trees provide the same functionality as R-trees, but have better insert and delete performance, without sacrificing search performance [Nam and Sussman 2004; Nam and Sussman 2006].

Each proxy has its own semantic cache, that is, a proxy has both a local index as well as the global index for the MBRs of the backend application servers. When the proxy receives queries from clients, it searches its local index first in order to locate suitable objects in its own semantic cache. Assuming the query cannot be fully computed by the proxy, the proxy generates subqueries for query regions that are not fully computed. These subqueries are expressed in terms of a query predicate that also specifies the spatial domain as an MBR. The MBR for each of the subqueries is then used to search the global index to locate an application server with the greatest amount of MBR overlap or for maximizing some other optimization heuristic, as will be discussed in detail later.

To update the hierarchical index when a new data object is created (for example, as a result of a query generating new *cacheable* intermediate results), the object's MBR is compared against the current MBR for the local index root node. If the new aggregate's MBR is outside the current local index MBR, the local index MBR must be enlarged to include the new data object. Whenever the MBR of a local index is enlarged (or shrunk, because an object is no longer indexed), the MBR update must be forwarded to the global index. When the global index server receives the update notification, it replaces the old MBR related to the local index requesting the update. Because in many cases insert/delete operations may not change the local index MBR, many such operations are done locally without updates being sent to the global 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) as shown in Figure 4. In the example, the root node of the index tree has three descendant tree nodes, whose MBRs are depicted as gray rectangles. The upper left corner and lower right corner of the root node MBR represent dead space. Thus if the global index receives a query that falls in that area, it will forward the query to this local server only to find out that the actual dataset stored in this local server does not overlap the query.

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. 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 4. 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 – the exponential growth of hypervolume as a function of dimension [Bellman 1961].

4.1 Multiple Query Scheduling Policies

The distributed index address 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 5.

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, in this paper, 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.

[2]As will be seen in Section 5, 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.

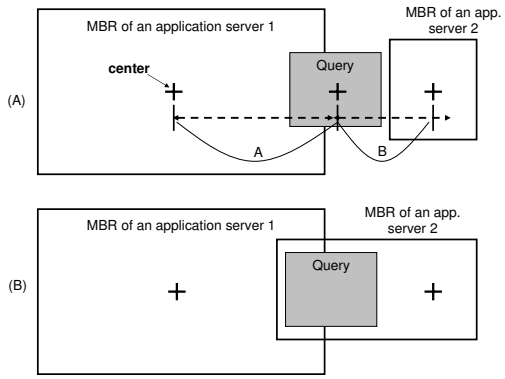


Figure 5: *Minimum distance policy*

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 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 5 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.

5 Experiments

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.

5.1 A Volumetric Reconstruction Application

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 [Borovikov et al. 2003].

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.

5.2 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.

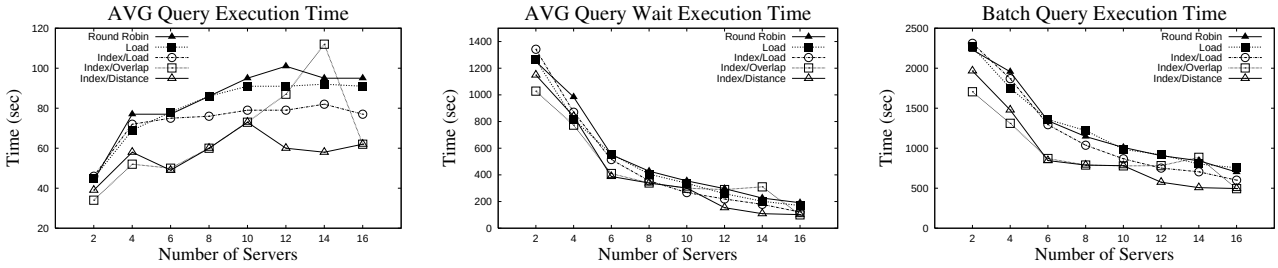
The dataset we used is a multi-perspective sequence of 2600 frames generated by 13 synchronized color cameras, each producing 640×480 pixel images at 30 Hz [Borovikov et al. 2003]. The test dataset is partitioned into 32 silhouette image files (each file is 329 MB in size totaling about 10 GB). In order 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 8 query batch files that have 100 queries each, with various query inter-arrival times, 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 reconstruct the entire available volume) and the depth of an octree was 6, except for the experiments shown in Figure 8. Queries also used data from all the available cameras for reconstruction.

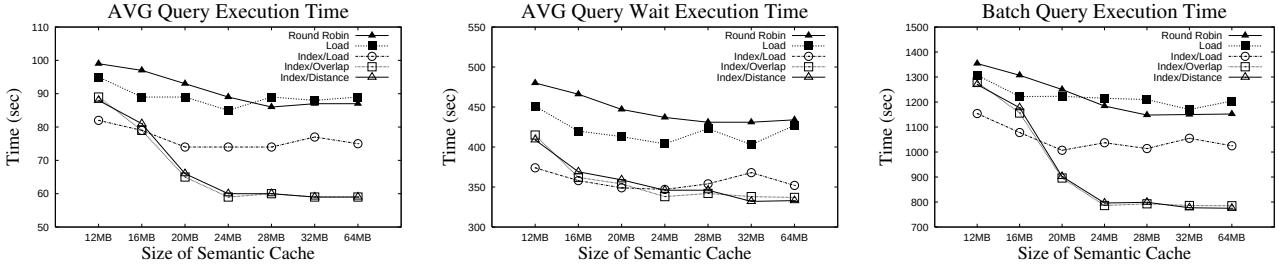
To measure performance, we considered the following metrics: *Query Wait and Execution Time* (QWET), *Query Execution Time* (QET), and *Total Batch Query Time* (TotalBQT). QWET is the amount of time from the moment a query is submitted to the system until it completes. That is, QWET includes the delay (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.



(a) Query Execution Time (b) Query Wait + Execution Time (c) Total Batch Query Time

Figure 6: *The Effect of Number of Servers*



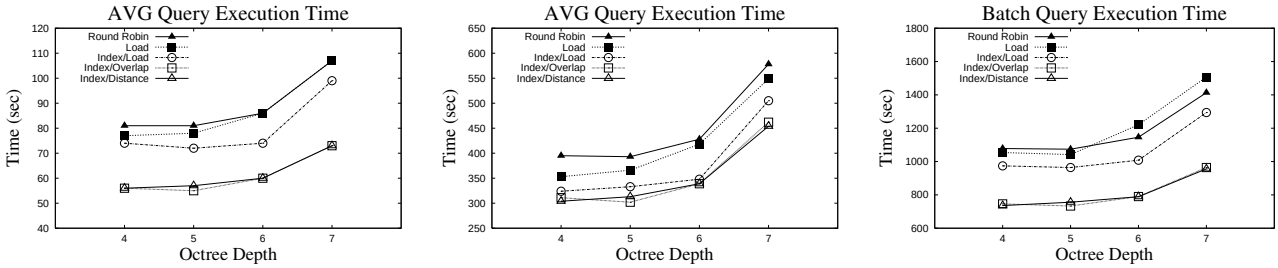
(a) Query Execution Time (b) Query Wait and Execution Time (c) Batch Batch Query Time

Figure 7: *The Effect of Semantic Cache Size*

5.3 Performance

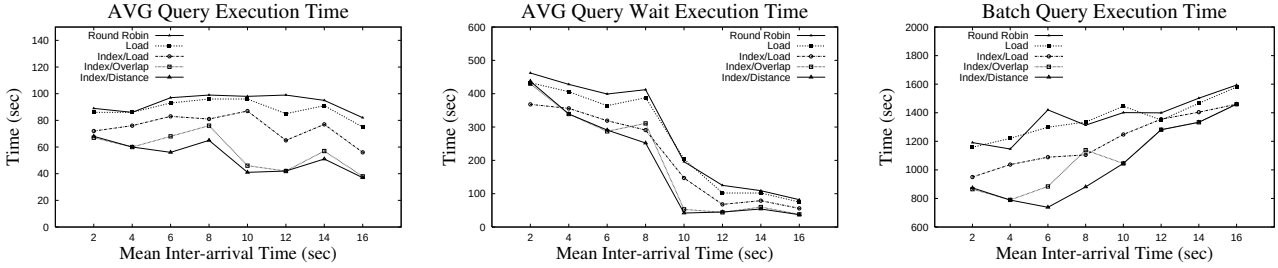
Figure 6 depicts system performance when we employed different query scheduling policies and varied the number of backend servers. For this experiment, we fixed the size of the semantic cache at 256MB and used LRU as the cache replacement policy on all backend servers. Each application server employed 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 6(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 6(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 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.



(a) Query Execution Time (b) Query Wait and Execution Time (c) Total Batch Query Time

Figure 8: *The Effect of Octree Depth*



(a) Query Execution Time (b) Query Wait and Execution Time (c) Total Batch Query Time

Figure 9: *Workload Comparison*

Figure 6(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/Overlap outperforms the other policies consistently, except when using only a small number of application servers (2 or 4). As seen in Figure 6(c), the total batch query time when using Index/Distance is around 60% to 88% of the time when round-robin is employed.

Figure 7 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 8 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 9 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 9, 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 9(a), as expected, we see that QET is not greatly affected by the inter-arrival time. In measuring query wait time (Figure 9(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 9(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.

To summarize, we have learned the following lessons from the experimental study. First, distributed indexing helps improve overall query processing performance, measured both by system throughput and by query response time. Second, load balancing is as important a factor in overall performance as cache hit rates for the distributed semantic caching infrastructure. Third, index-based scheduling that considers both load balancing and clustering properties (Index/Distance) tends to outperform less informed policies. Furthermore, it is more stable, rarely performing badly compared to the policies that use less information.

6 Conclusion

In this paper, we have described how a distributed multi-dimensional 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 visualization application show 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. To the best of our knowledge, this is the first work that shows that distributed multidimensional indexing helps improve query processing performance for a real distributed query processing system.

While we believe we have made progress in distributed query planning and scheduling and demonstrated this experimentally, ultimately, planning and scheduling in conjunction must be used in order to be able to keep a set of cached intermediate results that better represent the relevant working set in a distributed fashion. We intend to extend this work using different data analysis applications as well as different workload profiles in WAN-based heterogeneous environments. We postulate that under many circumstances data migration techniques and, possibly, pre-computation of frequently used cached aggregates by idle backend servers can help to further improve query processing performance. By disseminating the workset of reusable aggregates, wait times can be decreased as that will allow the proxy to use multiple backend servers for a higher percentage of queries, ultimately providing better workload distribution.

References

ANDRADE, H., KURC, T., SUSSMAN, A., AND SALTZ, J. 2002. Active Proxy-G: Optimizing the query execution process in the Grid. In *Proceedings of the ACM/IEEE SC2002 Conference*.

ANDRADE, H., KURC, T., SUSSMAN, A., AND SALTZ, J. 2004. Optimizing the execution of multiple data analysis queries on parallel and distributed environments. *IEEE Transactions on Parallel and Distributed Systems 15*, 6 (June), 520–532.

BELLMAN, R. E. 1961. *Adaptive Control Processes: A GuidedTour*. Princeton University Press, NJ.

BEYNON, M. D., KURC, T., ÇATALYÜREK, U., CHANG, C., SUSSMAN, A., AND SALTZ, J. 2001. Distributed processing of very large datasets with DataCutter. *Parallel Computing 27*, 11 (Oct.), 1457–1478.

BEYNON, M., CHANG, C., CATALYUREK, U., KURC, T., SUSSMAN, A., ANDRADE, H., FERREIRA, R., AND SALTZ, J. 2002. Processing large-scale multidimensional data in parallel and distributed environments. *Parallel Computing 28*, 5 (May), 827–859. Special issue on Data Intensive Computing.

BOROVIKOV, E., SUSSMAN, A., AND DAVIS, L. 2003. A high performance multi-perspective vision studio. In

Proceedings of the 17th ACM International Conference on Supercomputing (ICS).

DAR, S., FRANKLIN, M. J., JONSSON, B. T., SRIVASTAVA, D., AND TAN, M. 1996. Semantic data caching and replacement. In *Proceedings of the 22th International Conference on Very Large Data Bases (VLDB)*, 330–341.

GODFREY, P., AND GRYZ, J. 1999. Answering queries by semantic caches. In *Proceedings of the 10th International Conference on Database and Expert Systems Applications (DEXA)*, 485–498.

GUTTMAN, A. 1984. R-trees: A dynamic index structure for spatial searching. In *Proceedings of 1984 ACM SIGMOD International Conference on Management of Data (SIGMOD)*, 47–57.

KAMEL, I., AND FALOUTSOS, C. 1992. Parallel R-trees. In *Proceedings of 1992 ACM SIGMOD International Conference on Management of Data (SIGMOD)*, 195–204.

KIM, J.-S., ANDRADE, H., AND SUSSMAN, A. 2005. 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.

KOUDAS, N., FALOUTSOS, C., AND KAMEL, I. 1996. Declustering spatial databases on a multi-computer architecture. In *Proceedings of the 5th International Conference on Extending Databases Technology (EDBT)*.

MONDAL, A., KITSUREGAWA, M., OOI, B. C., AND TAN, K. L. 2001. 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)*, 28–33.

NAM, B., AND SUSSMAN, A. 2004. A comparative study of spatial indexing techniques for multidimensional scientific datasets. In *Proceedings of 16th International Conference on Scientific and Statistical Database Management (SSDBM)*.

NAM, B., AND SUSSMAN, A. 2005. Spatial indexing of distributed multidimensional datasets. In *Proceedings of the 5th IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid)*.

NAM, B., AND SUSSMAN, A. 2006. DiST: Fully decentralized indexing for querying distributed multidimensional datasets. In *Proceedings of 20th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*.

RODRÍGUEZ-MARTÍNEZ, M., AND ROUSSOPOULOS, N. 2000. MOCHA: A self-extensible database middleware system for distributed data sources. In *Proceedings of 2000 ACM SIGMOD International Conference on Management of Data (SIGMOD)*, ACM Press, 213–224. ACM SIGMOD Record, Vol. 29, No. 2.

SCHNITZER, B., AND LEUTENEGGER, S. T. 1999. Master-Client R-Trees: A new parallel R-tree architecture. In *Proceedings of 11th International Conference on Scientific and Statistical Database Management (SSDBM)*, 68–77.

SELLIS, T. K., AND GHOSH, S. 1990. On the multiple-query optimization problem. *IEEE Transactions on Knowledge and Data Engineering 2*, 2, 262–266.

WESSELS, D., AND CLAFFY, K. C. 1998. ICP and the Squid web cache. *IEEE Journal on Selected Areas in Communications 16*, 3 (Apr.), 345–357.

ZHANG, K., ANDRADE, H., RASCHID, L., AND SUSSMAN, A. 2005. 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)*.