Multiple Query Scheduling for Distributed Semantic Caches

Beomseok Nam [a] Minho Shin [b] Henrique Andrade [c] Alan Sussman [d]

[a] *Oracle, 100 Oracle Parkway, Redwood Shores, CA 94065*

[b] *ISTS, Dartmouth College, Hanover, NH 03755*

[c] *IBM T. J. Watson Research Center, 19 Skyline Drive, Hawthorne, NY 10532*

[d] *Dept. of Computer Science, University of Maryland, College Park, MD 20742*

Keyword

Multiple query optimization; Distributed query scheduling; Data intensive computing

Abstract

In distributed query processing systems, load balancing plays an important role in maximizing system throughput. When queries can leverage cached intermediate results, improving the cache hit ratio becomes as important as load balancing in query scheduling, especially when dealing with computationally expensive queries. The scheduling policies must be designed to take into consideration the dynamic contents of the distributed caching infrastructure. In this paper, we propose and discuss several distributed query scheduling policies that directly consider the available cache contents by employing distributed multi-dimensional indexing structures and an exponential moving average approach to predicting cache contents. These approaches are shown to produce better query plans and faster query response times than traditional scheduling policies that do not predict dynamic contents in distributed caches. We experimentally demonstrate the utility of the scheduling policies using MQO, which is a distributed, Grid-enabled, multiple query processing middleware system we developed to optimize query processing for data analysis and visualization applications.

1 Introduction

In distributed query processing systems, cache hit ratio is as important as load balancing, especially for data and computation intensive applications. In this paper, we propose distributed query scheduling policies that consider the available cache contents by employing distributed multidimensional indexing structures or by employing a statistical prediction method to estimate remote cache contents.

Considerable research has been aimed at minimizing the cost of processing a collection of database queries. This research spans various contexts, including relational databases and database middleware designed to support data analysis applications [11, 12, 21, 27, 30]. In spite of those efforts, the fundamental planning problem has been shown to be NP-complete [27]. The optimization problem in a distributed environment is even more difficult, as there are additional difficult engineering aspects associated with workload distribution and planning. Nevertheless, many researchers have focused on minimizing query processing time through data and computation reuse using heuristics or probabilistically efficient techniques.

Exploiting sub-expression commonality across multiple queries has been shown to be useful. Such reuse reduces execution time by exploiting cached results [1, 2]. Over the last few years, we have developed a distributed query processing framework named MQO, which was specifically designed to support scientific data analysis applications. Such scientific data analysis applications process multidimensional spatial-temporal range queries. Query results are tagged with multidimensional coordinate metadata and stored into so-called *active semantic caches*. MQO utilizes a semantic cache in its distributed application servers, and can reuse query results and intermediate query data products to compute all or parts of subsequent queries. While the large capacity of a distributed cache infrastructure can substantially decrease query processing time, performing scheduling and planning while preserving cache locality is a hard problem. For example, in a distributed setting, attempting to obtain global cache snapshot information for query planning is not

practical, since cache replacement operations occur dynamically and frequently.

Heterogeneous Grid computing environments provide an ideal setting for data intensive applications that need extensive computational resources, since additional resources can be tapped in incrementally. In a Grid environment, it is not uncommon to partition datasets onto parallel storage systems and replicate them, which both improves scalability and avoids single point of failure problems. In order to harness distributed and replicated datasets, our query processing framework implements a simple directory service used to store and maintain information about the location of datasets, and also keeps track of the available query processing capabilities of the servers that store the datasets. When more than one server is able to process a query, the information maintained by the directory service can be used to optimize query processing.

In this work, we extended the simple directory service originally available in MQO to ensure that query planning can benefit from large increases in the size of the distributed semantic cache, which is made possible by pooling multiple Grid-enabled servers. In principle, increasing the size of the distributed cache can substantially decrease the amount of time required to process a query, provided that we can (1) efficiently index the cache contents and (2) subsequently make use of that information in query planning and scheduling.

The main contribution of this work is the design of distributed query scheduling policies that produce better query plans and faster query response time than traditional scheduling policies that do not consider remote cache contents. We achieve those results by, on average, simultaneously increasing the cache hit ratio and balancing the workload across distributed cache servers. We survey index-based scheduling policies [23], and propose a new scheduling policy that employs a statistical method, *exponential moving average*, to achieve load balance and predict cache contents.

The rest of the paper is organized as follows. In Section 2 we discuss other re-

search efforts related to distributed indexing and multiple query optimization. In Section 3 we describe the architecture of the MQO middleware. In Section 4 we discuss how distributed indexing was integrated into the query evaluation process. In Section 5 we present a new scheduling policy named DEMA. In Section 6 we present an experimental evaluation, where we examine the performance impact of different scheduling policies, measuring both query execution and waiting time, as well as batch execution time. Finally, in Section 7, we make concluding remarks and describe possible extensions to this work.

2 Related Work

Considerable work has been done on the problem of query optimization in application domains ranging from relational databases [8, 16, 25], to decision-support systems [13, 32], to data intensive analytical applications [3].

Mehta and DeWitt [20] considered how to plan multiple queries using variables such as current CPU utilization, memory usage, and I/O load. Their goal was to determine the degree of intra-operator parallelism in parallel database system, to minimize the total execution time of declustered join methods. Garofalakis and Ioannidis [10] presented a resource usage model to handle multiple query scheduling on hierarchical parallel systems. Sinha and Chase [28] looked at how inter-query locality can be exploited for large distributed systems. Their work centered on heuristics to minimize the flow time of queries where the query scheduler could re-arrange the query execution order to maximize the reuse of cached data. Our research group [1, 3] has investigated methods for splitting a query into parallel sub-queries in order to increase scheduling flexibility and maximize query locality in Grid environments.

There have also been numerous efforts to develop distributed query processing middleware systems for data-intensive scientific applications. Rodríguez-Martínez and

Roussopoulos [24] described database middleware (MOCHA) designed to interconnect distributed data sources. The system moves the code required to process the query to the location where the data resides. When neither data-shipping nor code shipping are viable options, distributed applications have employed proxy front-ends to distribute the processing for a query. Beynon et. al. [6, 29] proposed a proxy-based infrastructure for handling data intensive applications. Such approaches are inherently less scalable than relying on a collection of distributed cache servers available at multiple back-ends.

To seamlessly integrate multiple back-end servers as a single query server, it is generally helpful to efficiently index the data (cached or otherwise) that each server has access to. The R-tree was one of the first multidimensional object indexing data structures to be developed [14]. Kamel and Faloutsos [15] extended that work, by proposing parallel R-trees (Multiplexed R-trees). One of the limitations of that approach was that it targeted only a single CPU with multiple disks. That limitation was overcome by Master R-trees [19] and Master Client R-trees [26], both designed for shared nothing environments (i.e., distributed memory parallel machines). Both approaches assume 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 back-end servers for query processing, methods for load balancing must be considered, as we previously demonstrated through simulation [31]. More specifically, the savings resulting from reusing a cached result have 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. [22], where workload was shifted from heavily loaded servers to lightly loaded servers in shared nothing environments. Their approach for load balancing is different from ours in that our work investigates query scheduling policies instead of moving input datasets between servers to improve load balance.

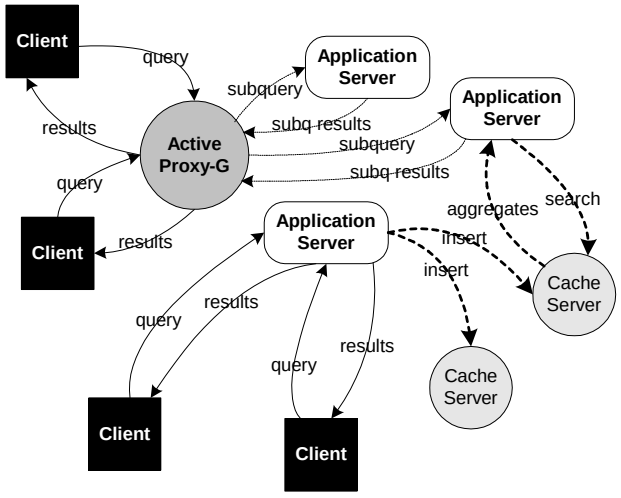


Fig. 1. *Overview of the MQO middleware*

Our previous work [23] describes a distributed multidimensional indexing scheme that enables the proxy to directly consider the cache contents available at the back-end servers for query planning and scheduling. That approach is shown to produce better query plans and faster query response times than traditional scheduling policies. In Section 4 we describe these index-based scheduling policies and compare them with the statistical estimation-based policy developed for this work.

Moving averages are a statistical method to predict trends by smoothing out short-term fluctuations [9]. Moving averages are commonly used to predict stock prices and financial data. However, to the best of our knowledge, this paper is the first work extending moving averages to distributed query processing, predicting distributed remote cache contents, and achieving load balance in a distributed system.

3 Improving the Multiple Query Processing Middleware

Muti-Query Optimization (MQO) is a middleware system designed to simplify the development and speed up the execution of large-scale data analysis applications. MQO's Grid-enabled configuration employs a proxy service comprising one or more Active Proxy-G (APG) instances, an application query processing service

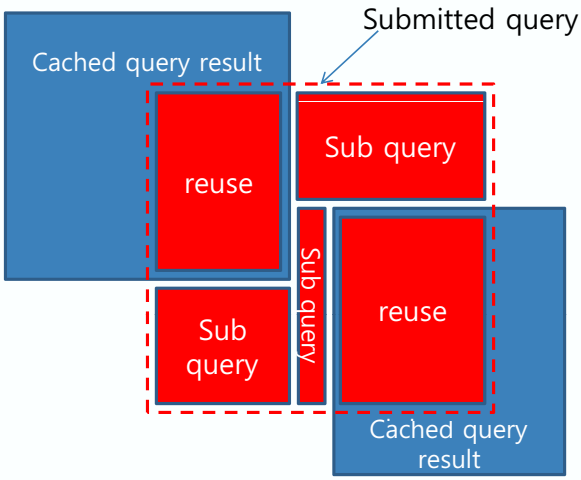


Fig. 2. *Sub-queries generated from reusing cached aggregates.*

comprising one or more back-end application servers, and a data caching service comprising one or more cache servers. This architecture is shown in Figure 1. The APG works as a front-end to the distributed multiple back-end application servers that execute application specific operators. These operators implemented by scientific applications process multidimensional range queries, which access the subset of data that falls within the range of values for each dimension. When a range query is received by the proxy, it may be able to process the query directly using its local cache contents. If the intermediate results in the cache alone cannot be used to fully compute the results for a query, the proxy server generates sub-queries for the unresolved portions as in Figure 2 and repeats the same process for the sub-queries, recursively. If no processing can be done by the proxy, the query is forwarded to back-end application servers, which then use their local cache or directly access the raw datasets to compute the results.

When a client submits a query through the proxy, the proxy's main task is to locate a suitable back-end application server to process it. The proxy employs a directory service (which we refer to as the Light Directory Service, or LDS), where information such as the location of datasets as well as *recent* performance metrics are stored. Dataset locations constrain the set of back-end application servers that can be used for servicing a query (i.e., in the current prototype a query can only be processed by a back-end application 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 back-end application 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 [2]: (1) round-robin, where a replica is selected based solely on where the last query was serviced, and (2) load-based policies where the *least busy* back-end application server with a suitable replica is selected. Note that clients can also directly submit queries to back-end application servers, if they know where the datasets are located. This capability may increase the potential for load imbalance. That is, imperfect information at the APG as well as additional load from servers directly submitting queries to back-end application servers compound the scheduling problem.

With the original query scheduling policies employed by MQO, the proxy service could only leverage previously computed results that were part of queries it had seen (i.e., queries that had 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 [2], it does not permit the utilization of *intermediate* data products that are automatically cached as a query is processed, because these data products are only available at the back-end application servers. Furthermore, the proxy cache can only grow in size up to the available storage in the node hosting the proxy. For these reasons and in order to generate better query plans that take into consideration the contents of remote semantic caches, an efficient distributed index is needed.

The semantic caches available at the back-end 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 to the point, it is very difficult to keep track of the up-to-date contents of remote semantic caches in large distributed sys-

tems. However, strong cache consistency is not 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 inexpensive as measured by I/O and CPU processing costs, simply Distributing the load across back-end application 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 [17, 18]. For these applications, increased 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 do that via distributed indexing.

4 Query Scheduling Policies with Distributed Indexing

In many scientific and large-scale visualization application domains, a data analysis query typically has two components that make up its predicate: one that specifies the processing necessary to generate the desired data product (specified as a directed graph containing nodes that denote the operators to apply on the flow of data, e.g., sampling, aggregation, filtering, geometric transformation, etc.), and another that specifies the spatial domain, usually in the form of a minimum bounding rectangle (MBR) that represents 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 data products computed as a query is processed. All of these aggregates are automatically cached and indexed by the middleware.

The multidimensional coordinates for the datasets can be indexed using multidimensional indexing structures, such as R-trees [14]. 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

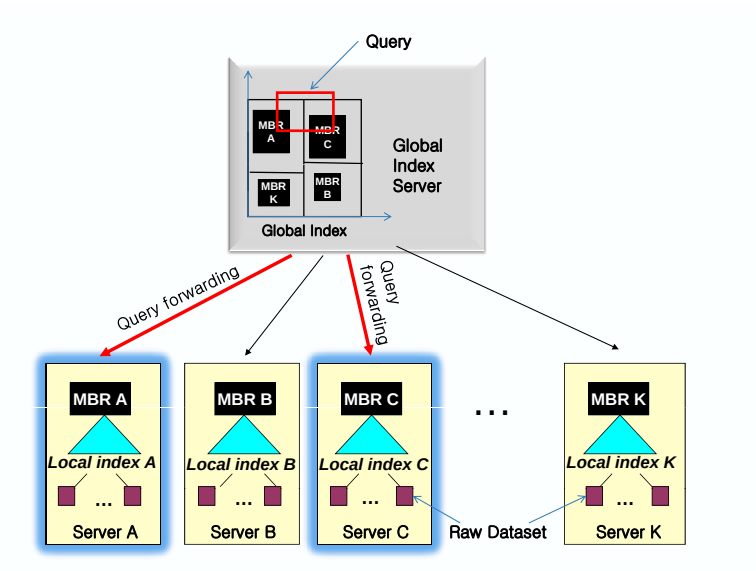


Fig. 3. *A Distributed index*

implement a distributed multidimensional 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. Since the index tracks the contents of an application server's semantic cache, as new objects are inserted or removed (as a result of a cache eviction), the index must be frequently and quickly updated. In most cases the overhead of query update is negligible compared to the benefits of cache hits in data intensive applications.

Multidimensional indexing can be integrated into MQO's distributed caching infrastructure to aid in locating candidate servers for executing queries or sub-queries on behalf of the proxy service, to make better query planning and scheduling decisions [23]. Each data object stored in the back-end servers' semantic cache is tagged with spatial coordinates, in addition to other attributes.

Figure 3 depicts an example internal organization for a distributed 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 relevant to the query. Each data server has its local index for managing the data products

that are generated locally as a byproduct of computing the results of queries. The comparison between the query's MBR and those MBRs defining the local indices results in the list of candidate back-end servers for the query. Hence, integrating distributed indexing with the MQO middleware contains two main tasks: (1) extending the back-end application server with a local index that tracks the contents of its semantic cache; and (2) extending the proxy service so that it can host the global index.

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 back-end 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 sub-queries for query regions that are not fully computed. These sub-queries are expressed in terms of a query predicate that also specifies the query spatial domain as an MBR. The MBR for each of the sub-queries is then used to search the global index to locate an application server with the greatest amount of MBR overlap or for maximizing other metrics.

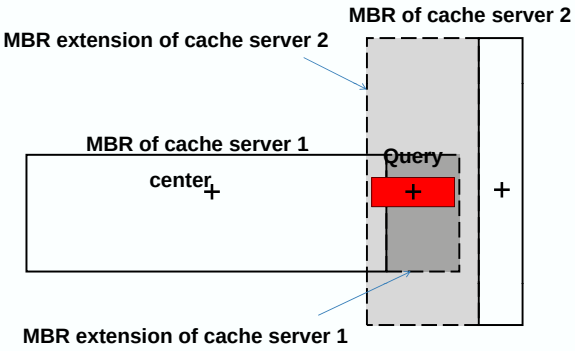
Since there are many options and combinations of existing data products where reuse might directly yield (or help computing) the required query result, a planning and scheduling approach can certainly benefit from such information. Specifically, an approach that considers the likelihood of cache hits against reducing the queuing delay from imbalances in assigning queries to back-end servers can provide substantial benefits. If we schedule queries based on cache-hit likelihood only, a server with popular aggregates can be swamped with a long queue of waiting queries. On the other hand, if we only take into account the queue size on each back-end server, many queries can experience long service times as they will be computed from scratch. Thus, query scheduling plays an important role in load balancing and, ultimately, in the overall response time and system throughput. In the rest of this section, we briefly discuss some of the traditional query scheduling policies, namely (*Round-robin* and *Load-based*), and summarize several index-based

scheduling policies that we investigated in prior work [23]. This discussion forms the baseline for a new query scheduling policy that makes use of *estimates* on the available cache contents, which we present in Section 5.

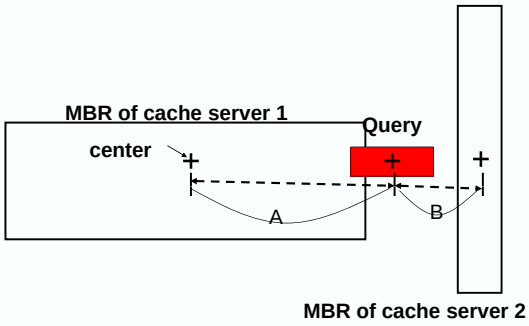
Round-robin: The round-robin scheduling policy assigns queries equitably to each application server as they arrive to the system. Round-robin is a simple, starvation-free scheduling policy that successfully balances server loads when all queries have similar costs. On the other hand, the round-robin policy clearly ignores state information for each back-end server, such as the current contents of its local cache or its current query load.

Load-based: The load-based scheduling policy assigns each query to the least busy server at the time the query is scheduled. The MQO middleware includes a workload monitoring service used for tracking the current load on each server that periodically collects server utilization parameters, such as internal thread pool usage, disk read rate, and the size of the query-wait queue. In our experimental evaluation (see Section 6), when employing this type of scheduling, we monitored the amount of time queries spent in the wait queue as a measure of the current server load.

Index/Load: The index/load combination scheduling policy assigns each query to the server with the smallest estimated processing time. The policy estimates the query processing time of each server based on a lookup into the global index as well as the current server load, i.e., the size of the query-wait queue. First, the policy does a lookup into the global index and decides whether the cache has reusable aggregates for the query. If there are reusable aggregates, the policy optimistically estimates the query processing time assuming cache hits for the query and cache misses for the rest of the queries in the queue. If no aggregates are reusable for the query, the policy pessimistically estimates the query processing time assuming that all queries in the queue will be processed from the scratch. When using this policy, if multiple back-end servers are found to have equal costs, one of them is chosen at random.



(a) Index/Overlap scheduling will select server 1 for the querv.



(b) Index/Distance scheduling will select server 2 since the query's center is closer to server 2.

Fig. 4. *Index/Overlap and Index/Distance policies*

Index/Overlap: The Index/Overlap scheduling policy assigns each query to the server that would require the smallest enlargement of the MBR for the aggregates cached at that server once that query is computed. Figure 4(a) shows an example of Index/Overlap scheduling. In the figure, the dashed rectangle represents the MBR extension for each server because of the new query. Since Server 1 would have the smaller enlargement triggered by executing the query, the policy assigns the query to it. This policy tries to keep the MBR of each back-end server as small as possible to achieve good clustering of queries with MBRs that are "close" in the multidimensional space. This minimum enlargement policy is similar to the node split policy of R-trees [14]. The decision whether to visit a child node depends on whether its

MBR overlaps the query region. Thus, R-trees are designed to try to minimize the total area of each MBR to reduce the number of node visits. For similar reason, the Index/Overlap policy tries to minimize MBR expansion to reduce *dead space* – (i.e., multidimensional regions in which no actual objects are located, but are covered by the MBR as a result of an enlargement operation made to accommodate a new object). This policy aims to minimize cache miss ratio by minimizing dead space across all servers. However, it may fail to balance the load across servers because a server with a larger MBR is more likely to be assigned queries. As for the Index/Load scheduling policy, ties are resolved by random selection. Unlike Load-based and Index/Load, however, the Index/Overlap policy does not need to monitor the load on the back-end servers.

Index/Distance: The Index/Distance scheduling policy assigns each query to the server whose MBR center is closest to the center of the query, using an Euclidean distance measure. Figure 4(b) shows an example. In the figure, the plus signs denote the geometric centers of the MBRs and the query. Since the center for Server 2 is closer to the query center than that of Server 1, the Index/Distance policy chooses Server 2 to process the query. Unlike the Index/Overlap policy, the index/Distance policy can balance the loads across the back-end servers as long as the geometric centers of the MBRs are evenly distributed across the back-end servers yield good load balancing, in general, this policy does imply good load balancing behavior.

5 The Distributed Exponential Moving Average Query Scheduling Policy

5.1 Exponential Moving Average

The *Distributed Exponential Moving Average (DEMA)* scheduling policy *asymptotically* estimates the center point of the cache contents using an exponential moving average (EMA) of the query center points.

EMA is a well-known statistical method to analyze historic data, which is most often used to predict stock prices and trading volumes [9]. In general, EMA-related methods compute a weighted average of all observed data by assigning exponentially more weight to recent data than earlier data. In the context of this paper, given that each server replaces old cache entries based on a least-recently used policy (LRU), assigning more weight to recent cached entries is a reasonable heuristic.

Alternatively, one can use a simple moving average (SMA) [9], which takes the average of the past N query center points. If the proxy is aware of the current number of cache entries in each back-end server, SMA can correctly represent the average of the current cache contents. SMA, however, cannot reflect the moving trend of arriving queries. Furthermore, keeping track of the last N query center points is also as costly as what is done in the index-based scheduling policies. In fact, SMA is similar to Index/Distance, but it uses the average of the center points of all cached results, while Index/Distance uses the center point of the minimum bounding box in the cache.

Let p_t be the center of the query at time $t > 0$ and EMA_t be the computed average at time t after adding p_t into the cache. Given the *smoothing factor* $\alpha \in (0, 1)$ and the previous average EMA_{t-1}, the next average EMA_t can be defined incrementally as

$$EMA_t = \alpha \cdot p_t + (1 - \alpha) \cdot EMA_{t-1} \tag{1}$$

which can be expanded as

$$EMA_t = \alpha p_t + \alpha(1-\alpha)p_{t-1} + \alpha(1-\alpha)^2 p_{t-2} + \alpha(1-\alpha)^3 p_{t-3} + \cdots \tag{2}$$

Although we call this value *EMA-value*, it can also be a multi-dimensional point if the query is in multi-dimension.

The definition of EMA in Equation 2 assumes an infinite number of past data, so

that the weight-sum amounts to one. In practice, however, the number of queries in the past is not infinite. Suppose N is the number of queries in the past. Then the current EMA value computed by Equation 1 becomes

$$EMA_t = \alpha p_t + \alpha(1-\alpha)p_{t-1} + \cdots + \alpha(1-\alpha)^{N-1}p_{t-N+1} \tag{3}$$

Therefore, the sum of weights is less than one by $(1-\alpha)^N$.

$$\begin{aligned}\text{Weight Error} &= \alpha + \alpha(1-\alpha) + \alpha(1-\alpha)^2 + \cdots + \alpha(1-\alpha)^{N-1} + \alpha(1-\alpha)^N + \cdots \\ &\quad -(\alpha + \alpha(1-\alpha) + \alpha(1-\alpha)^2 + \cdots + \alpha(1-\alpha)^{N-1}) \\ &= \alpha(1-\alpha)^N + \alpha(1-\alpha)^{N+1} + \alpha(1-\alpha)^{N+2} + \cdots \\ &= \alpha(1-\alpha)^N \cdot (1 + (1-\alpha) + (1-\alpha)^2 + \cdots + (1-\alpha)^{N-1} + \cdots) \\ &= (1-\alpha)^N \end{aligned} \tag{4}$$

This *weight-error* becomes smaller as the number of queries increases. In our experiments, N was sufficiently large (up to 700), so the weight error is negligible.

The smoothing factor α in Equation 1 determines the degree of decay used to expunge older data. For example, α close to 1 drastically decreases the weight assigned to the past data and α close to 0 gradually decreases that weight. The value of α can be adjusted based on the size of the cache space. Given a cache size L, we want to assign an enough weight-sum to the latest L queries (otherwise it can cause false positives focusing too much on the old queries). At the same time, we want to assign a non-negligible weight to the oldest query in the cache (otherwise it can cause false negatives focusing too much on the recent queries). From our empirical evaluation, we learned that the cache hit ratio was maximized when α was close to $1/L$. Since the weight-sum of the latest L queries is $1-(1-\alpha)^L$ and the weight of the oldest query in the cache is $\alpha(1-\alpha)^{L-1}$, choosing $\alpha = 1/L$ means to assign a weight more than 0.63 ($1-(1-1/L)^L > 0.63$ when $L > 0$) to the queries in the cache, and keep the weight of the oldest query in the cache around $1/L(1-1/L)^{L-1}$ (e.g., 0.12 for $L = 3.5$ and 0.08 for $L = 4.7$ in our experiments).

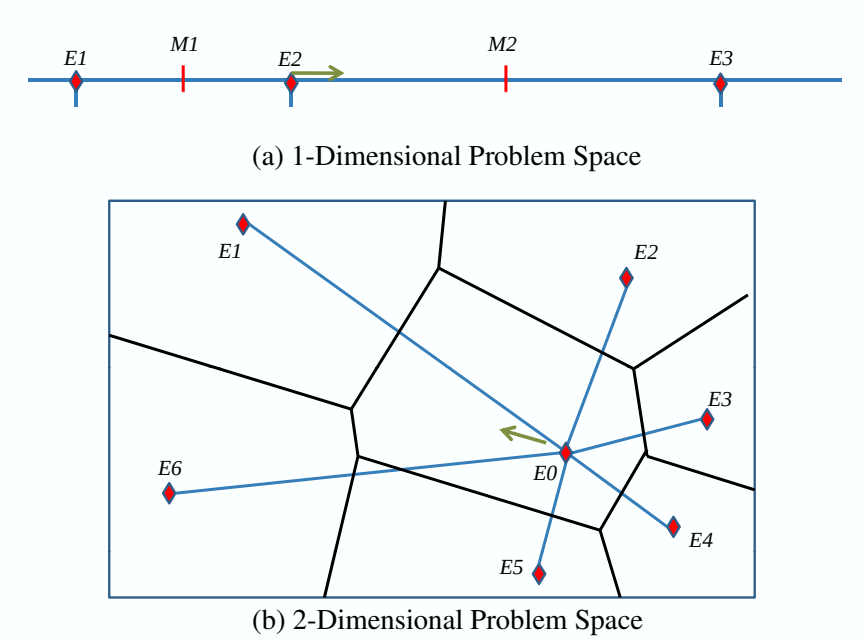
Algorithm 1.
DEMA Algorithm

INPUT: a client query Q
$MinDistance \leftarrow MaxNum$
for $s = 1$ to $NumberOfApplicationServers$ **do**
  if EMA[s] is not initialized **then**
    forward query Q to server s.
    $EMA[s] \leftarrow Q$;
    return;
  else
    $Distance \leftarrow EuclideanDistance(EMA[s], Q)$;
    if $Distance < MinDistance$ **then**
      $SelectedServer \leftarrow s$
      $MinDistance \leftarrow Distance$
    end if
  end if
end for
forward query Q to $SelectedServer$.
$EMA[SelectedServer] \leftarrow (\alpha \cdot Q) + (1 - \alpha) \cdot EMA[SelectedServer]$

5.2 Exponential Moving Average in Distributed Query Processing

The distributed EMA (DEMA) query scheduling policy employs EMA computations as its main mechanism. In this case, the proxy calculates the EMA point for each server and assigns each query to the server whose current EMA point is the closest to the center of the query, as shown in Algorithm 1. Suppose we have n servers and E_s represents the EMA point of the sth server. For each query with its center point at q, the proxy assigns the query to the server s^* such that the distance between E_{s^*} and q is the smallest among all servers. Then, the proxy updates the EMA value of the assigned server by $E_{s^*} = \alpha q + (1 - \alpha)E_{s^*}$. Figure 5(a) shows an example of a one dimensional problem space, where if the center of a query is between E_2 and M_2, the query is forwarded to server 2.

The performance of a query-scheduling policy depends on how well it can balance queries among application servers. Ideally, we want to assign new queries to the different back-end servers with the same probability. The probability of assigning a new query to a specific server depends on the size of the region (as a line in 1-



(a) 1-Dimensional Problem Space

(b) 2-Dimensional Problem Space

Fig. 5. *DEMA for load balancing*

dimension space or as an area in 2-dimension space) where the server is the closest among all the servers. In Figure 5(a), the region of server with EMA value of E_2 is the range between $M1$ and $M2$. For a 2-dimensional space, Figure 5(b) shows a Voronoi diagram for the EMA-values of the servers, and the central area is the region for the server of E_0.

DEMA balances the server loads by keeping the region size of each server as similar as possible. Let us assume that the center of the query is uniformly distributed in the problem space. In Figure 5(a), the query between M_1 and M_2 is assigned to the server with E_2. Moreover, a new query is more likely to land on the right segment $\overline{E_2M_2}$ than the left one $\overline{M_1E_2}$. Therefore, by Equation 1, E_2 is more likely to move to the right than the left, balancing itself within its region $\overline{M_1M_2}$. As all the servers try to balance the distance from their left neighbor (the distance being $2 \times \overline{M_1E_2}$) with the distance from their right one (the distance being $2 \times \overline{E_2M_2}$), the region for each server converges to the same length.

Similarly, DEMA also balances the server load in two dimensions, as shown in

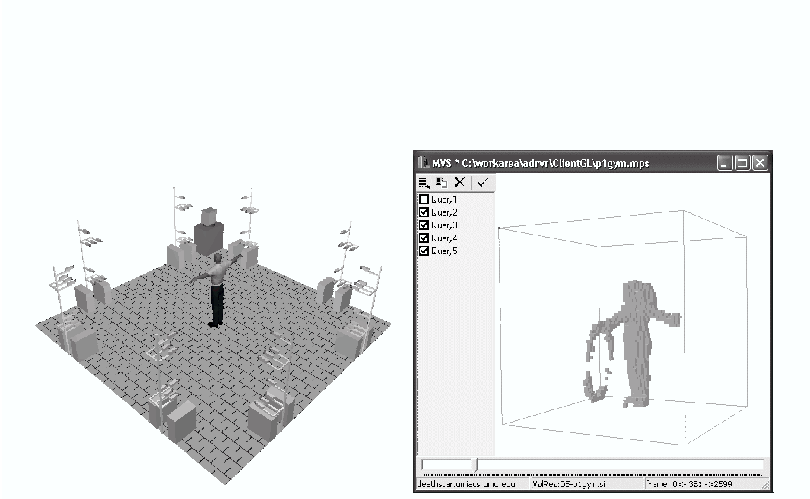
Figure 5(b). In the figure, E_0 tends to move toward the center of its area because the queries are more likely to land on the larger side of E_0 (i.e., the upper-left portion of the area) than the other side (i.e., the lower-right portion). As every server does the same balancing, the distance from each server to any of its boundaries converges to the same length, resulting in similar sized regions for each server. One can make a similar argument for higher dimensional problem spaces. In Section 6 we empirically show that DEMA balances the load best among all the distributed scheduling policies described in this paper.

Note that the DEMA policy can exert control over how responsive it is to changes in the workload or cache sizes by tweaking the α parameter. Also, the DEMA policy is simple in terms of state management, requiring no updates of back-end application server information, while index based scheduling policies require updating the MBR information for each of the back-end servers, which requires communication, although a server performs an update only if a query changes the MBR representing all of the cached data for that server.

6 Experiments

The primary objective of the experimental study was to measure system throughput for the different scheduling policies described in Section 4 and Section 5. Performance improvements from planning and scheduling are typically highly dependent on the nature of applications, the system characteristics, and also the characteristics of the experimental workload. In order to shed light on the magnitude of improvements that can be expected from employing distributed indexing, we performed studies using a computationally intensive computer vision application, which we believe is a proxy for many of the visualization and data analysis techniques used by scientific applications.

The *multi-perspective vision studio* (or MPVS) is a volumetric reconstruction appli-



(a) Keck Lab Model (b) VR Client GUI

Fig. 6. *The Multi-Perspective Vision Studio application – (a) The set up for image capture lab (b) The application Graphical User Interface*

cation that processes very large amounts of image data from multiple video streams. In an environment where several cameras are used for simultaneously capturing scenes from various perspectives, as seen in Figure 6, more views can deliver additional information about the scene and potentially allow the recovery of interesting 3-D features with high accuracy and minimal intrusion into the scene [7].

Users interact with the MPVS application by submitting scene visualization 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, thereby displaying finer aspects and features found in the reconstructed scene.

6.1 Experimental Environment

We have measured the performance of MQO running the MPVS application on 16 back-end servers and 1 front-end proxy server. Back-end servers and the proxy were placed on different nodes of a Linux cluster. Each server has an Intel Xeon 2.66GHz processor with 2GB memory, and they are connected by Gigabit switched Ethernet. Each of the 16 back-end servers can independently compute a volumetric reconstruction query with access to complete replicas of the entire dataset.

The dataset we used is a multi-perspective sequence of 2600 frames generated by 13 synchronized color cameras, each producing 640 $\times$ 480 pixel images at 30 Hz [7]. 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 back-end 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 query batch files that have 700 queries each, with various query inter-arrival times, simulating multiple simultaneous users posing queries to the system, modeled as a Poisson process. The queries in a batch were constructed according to a synthetic workload model since our real user traces were not sufficient for carrying out all of the relevant experiments. The workload generator emulates a hypothetical situation in which users want to view a short, multi-second 3D instant replay of *hot* events that took place in the past, for example, in a live sports event broadcast. 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.

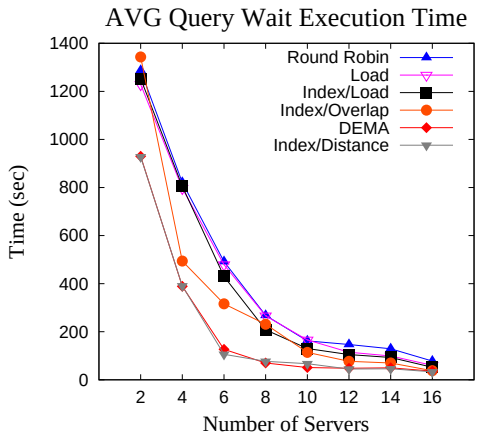
A query in a batch requests a set of 3D reconstructions associated with frames selected according to the following model. The center of the interval is drawn ran-

domly with a uniform distribution from the set of hot frames (100 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. With octree depth 6, the number of voxels to be computed and stored is 8^6 = 262144 voxels. The scene space in our dataset is 2m $\times$ 2m $\times$ 2m, and each voxel is a 3.125cm $\times$ 3.125cm $\times$ 3.125cm cube. This number gives an estimate of how much computation must be carried out per frame, as well as the amount of memory we require per reconstructed frame. Queries also used data from all the available cameras for reconstruction.

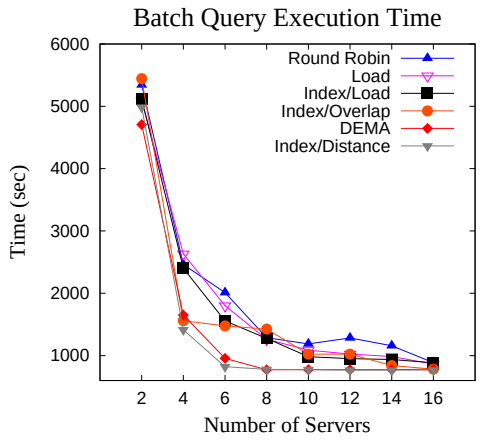
To measure performance, we considered the following metrics: *Query Wait and Execution Time* (QWET), *Query Execution Time* (QET), *Total Batch Query Time* (TotalBQT), *Cache Hit Ratio*, and *Load Balancing Factor*. QWET is the amount of time from the moment a query is submitted to the system until it 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 back-end server starts processing the query until completion as measured at the proxy. Hence QET completely depends on the local cache hit ratio, while QWET, to a greater degree, depends on load balancing across the back-end 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. We measured cache hit ratio by dividing the total number of image frames read from cache by the number of image frames requested by batch queries. In order to evaluate the load balancing behavior of the proposed scheduling policies, we computed the standard deviation of the number of queries assigned to each back-

end server.

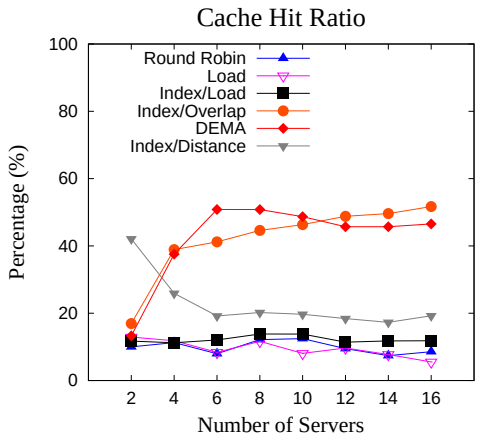
(a) AVG Query Execution Time



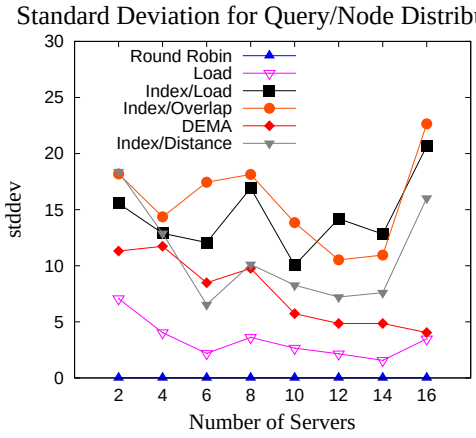
(b) AVG Query Wait + Execution Time



(c) AVG Total Batch Query Time



(d) Cache Hit Ratio



(e) Load Balance

Fig. 7. *The Effect of Number of Servers*

It should be noted that the MQO middleware has several control knobs [4]. 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 service, but kept it enabled on the back-end servers.

6.2 Performance

Figure 7 depicts system performance for the different query scheduling policies, varying the number of back-end servers. This experiment shows the average execution time of a query without and with the wait time included in Figures 7(a) and 7(b), respectively. In this experiment, we fixed the size of the semantic cache at 64 MB and used LRU as the cache replacement policy on all back-end servers. Each back-end server employed a single thread for processing the incoming queries. The front-end proxy runs as many threads as needed, allowing multiple queries to be *accepted* concurrently from clients. If the assigned back-end server is busy once a query is *dispatched* for execution, the query might have to wait to be serviced from the back-end server incoming queue.

While in general QWET (Figure 7(b)) and the total batch query execution time (Figure 7(c)) decrease as more back-end servers are added, when queries arrive at a faster rate than the back-end servers can process them, the queries might be waiting in queues for a considerable amount of time. That can be seen by contrasting the QET (Figure 7(a)) and QWET (Figure 7(b)) curves.

More interestingly, if the queries in the workload have a small number of hot spots, frequently used cache objects are dispersed through the multiple back-end server caches as the number of back-end servers increases, and the *per server* cache hit ratio is usually lower early in the processing sequence. On the other hand, additional back-end servers imply additional cache space. As a consequence, the average QET decreases as more queries can be directly computed from cached results, as seen in Figure 7(a). As expected, round-robin shows the worst performance in most cases.

Similarly, load-based scheduling also does not perform well. In both cases the poor performance is a consequence of neither policy considering the contents of the back-end server caches in scheduling queries for processing. On the other hand, as the local caches co-located with each back-end server get populated, the three index-based scheduling policies start to reap the benefits of increased cache hit rates even when more servers and cache space becomes available, as compared to the other policies (Figure 7(d)). Note that the DEMA scheduling policy shows significant performance benefits over the two non-index-based policies. Despite that, the DEMA scheduling policy fails to consistently outperform both the Index/Overlap and Index/Distance policies. In this case, however, the query batch used for the experiment comprises 200 queries with a 4 second average query inter-arrival time (exponentially distributed) and the DEMA weight factor α was 0.5. In such a configuration, when more than 10 back-end servers are available, most of the submitted queries are scheduled and executed immediately without waiting in the queue, as we can see that the absolute difference between QWET and QET is very small. Later in this section, we will show how the number of waiting queries and α affect overall performance.

With respect to load balancing, Figure 7(e) shows that the Index/Overlap policy has much higher standard deviation in the number of queries assigned to a back-end server, indicating inferior load balancing behavior compared to the Index/Distance and DEMA policies. This explains its higher average QWET compared to the Index/Distance policy, even though its cache hit rate is generally higher. As we discussed earlier, when the top-level MBR for a particular local index gets enlarged, the proxy becomes biased and chooses the back-end server with the largest overlapping MBR. Thus, a majority of queries are forwarded to a single back-end server, which results in that server having a longer wait queue, increasing QET and QWET, and also negatively impacting load balance. Unlike Index/Overlap, the other two index-based policies, Index/Distance and Index/Load, avoid such a load imbalance problem. Although Index/Load does not severely suffer from load im-

balance, 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 this problem creates large amount of dead space (i.e., the MBR contains large amounts of unfilled space) as compared to the Index/Distance and Index/Overlap policies, as those 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 described earlier. Surprisingly, however, the decrease in cache hit rate seems to indicate that the Index/Load policy loses its good query clustering properties, i.e., sending queries to back-end servers where there is a higher likelihood of a cache hit. Finally, and not surprisingly, the round-robin scheduling policy shows the lowest standard deviation. Despite its excellent load balancing characteristics, its low cache hit ratio accounts for its overall poor performance.

Next, we study the effect of increased load on the system. With 16 application servers and a larger query workload comprising 700 queries, we controlled the query inter-arrival time so that we can evaluate the effect of query scheduling as a result of the amount of concurrent load presented to the system. In this case we fixed each back-end servers' local cache size at 64 MB. These results are shown in Figure 8. A particularly illuminating result can be seen in Figure 8(d). As the mean query inter-arrival time increases, we can see that the cache hit ratio of Index/Distance policy fluctuates between 50% and 60%, while DEMA hit ratio stays constant at around 64%. The reason why DEMA shows constant hit ratio is because this policy depends on which queries have arrived before the current one, but not on how long ago they arrived. On the other hand, the Index/Distance policy makes scheduling decisions based solely on the cache index contents, not taking into account the queries that are already waiting in the queue. Hence, as more queries wait in the queue, it is more likely that Index/Distance will make poor server assignment decisions (as it cannot take into account other queries that might generate relevant results in the near future). This ultimately explains why DEMA outperforms Index/Distance in the heavier workload experiments (i.e., smaller query inter-arrival

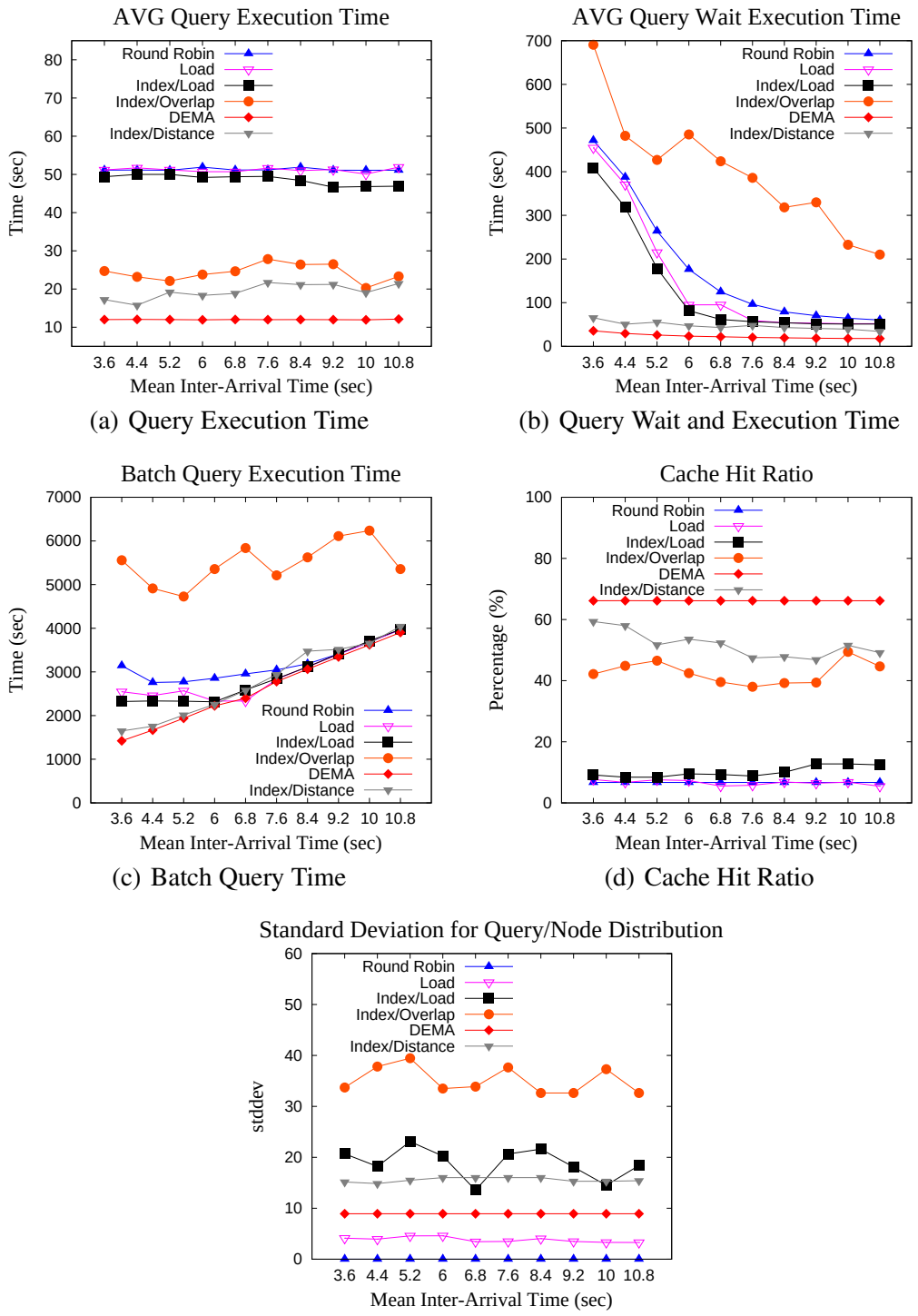
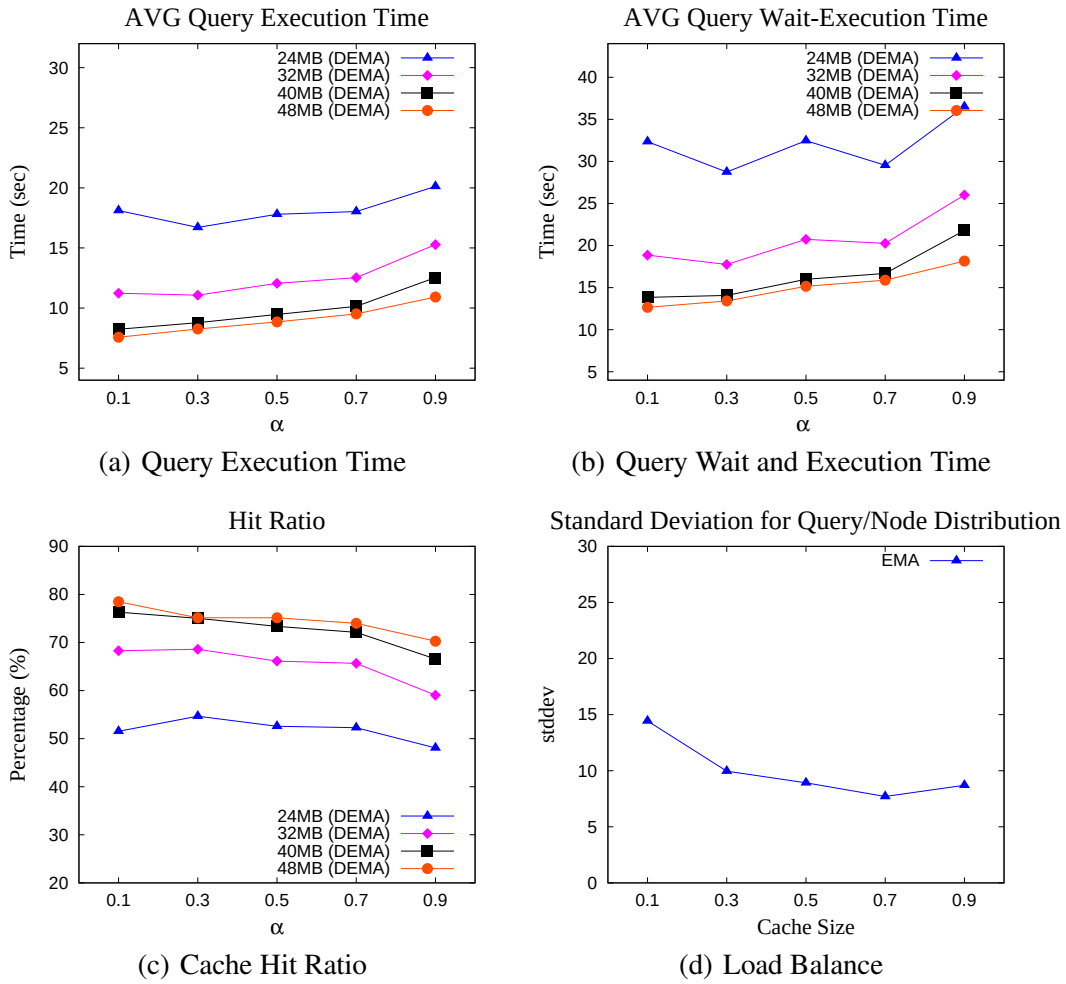


Fig. 8. *Workload Comparison*

times) shown in Figure 8. Interestingly, the Index/Overlap policy performs worst in this case primarily due to load imbalance. Since more queries are submitted than for the earlier experiments shown in Figure 7, the poor load-balancing quality of the

Index/Overlap policy causes longer average waits in the queue, decreasing overall system throughput.



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

(c) Cache Hit Ratio (d) Load Balance

Fig. 9. *Sensitivity to α values*

Finally, we look at the impact of calibrating the DEMA smoothing factor. We ran experiments with different α settings, employing the 700-query batch described earlier, with an exponentially distributed inter-arrival time with mean 7.6 seconds. For this experiment we employed 16 back-end servers and varied the amount of cache space available at each of them. The results are shown in Figure 9. Recall that with smaller α values the DEMA policy will give more weight to a large number of older entries in the cache. Therefore, when the cache capacity is inadequate to hold the working set and DEMA is configured with a small α value, scheduling decisions will be made from predictions based on cache contents that have been

evicted from the cache, possibly resulting in lower cache hit ratio. On the other extreme, a larger α setting also may result in poor scheduling decisions, since it will not accurately consider older cached results. In Figure 9 we see that setting α to 0.1 (a small value) results in a low cache hit ratio with a 24 MB cache. Similarly, higher value settings for α also result in worse hit ratios.

In the graphs shown in Figure 10, for smaller cache capacities, Index/Distance policy suffers from a higher rate of cache misses than DEMA. This is mainly because Index/Distance does not consider the queries in the waiting queue for scheduling purposes. In this situation, two factors contribute to inferior performance. First, when the cache is small more queries have to be computed from scratch, as is also the case for the other scheduling policies. However, because the decreased cache hit ratio makes queries wait longer in the queue, Index-based scheduling policies will evict results that could have been employed to speed up the computation of queries already in the waiting queue.

In these experiments, a single frame in the MPVS application requires 262 KB of storage. Inspecting query results revealed that the queries have high variance in their space requirements: the smallest query required 514 KB of cache space and the largest 20 MB. The variance is from the number of frames being different (2 - 80 frames). The average number of frames in the query workload was 26. Hence 24 MB of cache space can hold the intermediate data products for between 3 and 4 queries. Assuming that the cache has enough space to hold enough reconstructed frames to compute between 3 and 4 query results, then ideally we should choose the setting for α that gives most of weight to only 3 or 4 recent cache entries. From Equation 2, $\alpha = 0.7$ gives 99 % weight to the 4 oldest queries. However $\alpha = 0.7$ makes the weight of the oldest cache result almost negligible ($\alpha(1-\alpha)^{3.6} = 0.005$). On the other hand, $\alpha = 0.1$ gives more than 70 % weight to the query results that have already been evicted from cache. $\alpha = 0.3$ gives about 80 % weight to the most recent 4 queries, and the weight of the oldest cache result is about 0.08, which is not bad considering that $\alpha(1-\alpha)^{3.6}$ is at its maximum (0.087) when α is 0.22.

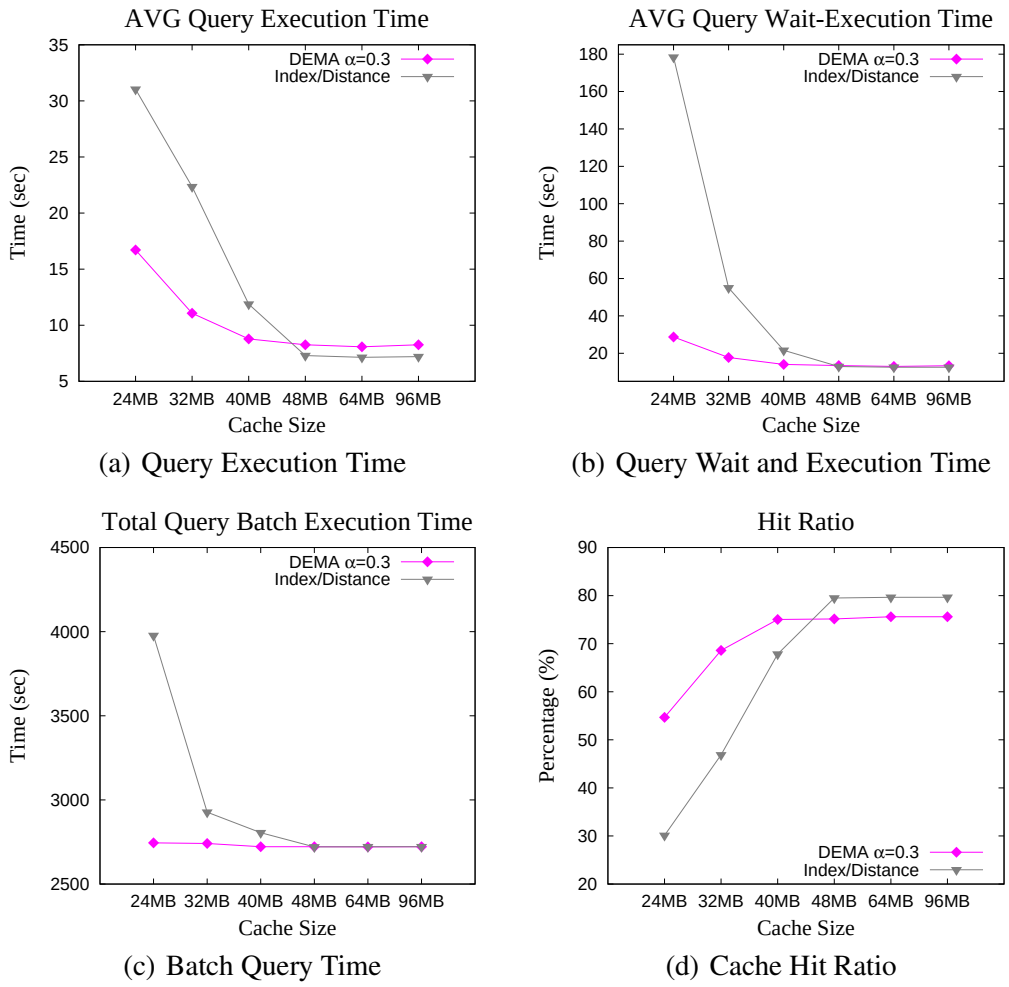


Fig. 10. *Effect of cache size*

Table 1
Comparison of Distributed Query scheduling policies

Scheduling	Status Collection	Cache Locality	Merit	Limitation
Round-Robin	No	No	Good load balancing in homogeneous environment	Heterogeneous systems
Load-Based	Yes	No	Good load balancing	Poor cache hit ratio
Index/Overlap	Yes	Computed	High cache hit ratio	Load imbalance
Index/Distance	Yes	Computed	High cache hit ratio, Good load balancing in homogeneous environment	Indexing overhead
Index/Load	Yes	Computed	Good load balancing	Limited cost model
DEMA	No	Predicted	Good for heavy loaded systems, No index required	Single front-end prediction

As a final note, despite DEMA's simplicity and surprisingly good performance, we should point out that it also has limitations that are not present for the indexing-based policies. For example, if there exists more than one proxy service server or if back-end application servers can receive queries directly from clients (which the MQO middleware does permit), the simple predictive model upon which DEMA relies for routing the queries for execution might be quite inaccurate, since the information the DEMA model relies on might be incomplete.

To summarize, we have learned the following lessons from the experimental study. First, distributed indexing can help improve overall query processing performance, measured both by system throughput metrics and by query response time. Second, load balancing is as important a factor in overall performance as is attempting to route queries to back-end servers aiming at improving reuse by increasing the cache hit likelihood. Third, an index-based scheduling policy that considers both load balancing and clustering properties (such as the Index/Distance policy) tends to outperform less informed policies. The Index/Distance policy is most stable, rarely performing badly compared to the policies that use less information. Finally, the DEMA policy, despite its simplicity, shows stable and competitive performance when compared to more informed indexing-based scheduling policies in situations where the system is heavily loaded and the cache size is not large enough to accommodate the working set. A summary of these findings is shown in Table 1.

7 Conclusion

In this paper we have described how a Grid-enabled multidimensional indexing scheme and an exponential moving average approach can be used by a distributed multiple query optimization middleware system to generate better query plans on behalf of the application it supports. We have devised scheduling policies that employ information about the contents of remote semantic caches as well as distributed system load information with two main goals in mind. First, queries should

be sent to servers that contain data products that are relevant to computing the results for these new queries, thus increasing the likelihood of reuse of pre-computed results. Second, queries should be dispatched to the back-end servers in a balanced way, so the distributed computing resources can be used with *equal* likelihood, reducing contention and response time and consequently and increasing the application's throughput.

We have also presented a novel statistical prediction-based query scheduling policy that shows comparable query response time and system throughput to index-based scheduling policies, while being simpler from a bookkeeping standpoint. Moreover, it outperforms the index-based scheduling policies when the system is heavily loaded and the cache size is not large enough to accommodate the working set.

While we have made progress in distributed query planning and scheduling, we believe that query scheduling and cache replacement policies must be designed to work in tandem such that the distributed local caches can try to closely capture the global and dynamic state of the relevant working set. We also postulate that, under many circumstances, data migration techniques and, possibly, the pre-computation of frequently used intermediate cached data products by *idle* back-end servers can help to further improve query processing performance. By distributing the working set of reusable cached results, wait times can be decreased because that will allow the proxy to use multiple back-end servers for a higher percentage of queries, ultimately providing better workload distribution. These are interesting ideas that we intend to investigate in the near future.

References

[1] Henrique Andrade, Tahsin Kurc, Alan Sussman, and Joel Saltz. Efficient execution of multiple query workloads in data analysis applications. In *Proceedings of the ACM/IEEE SC2001 Conference*, November 2001.

[2] Henrique Andrade, Tahsin Kurc, Alan Sussman, and Joel Saltz. Active Proxy-

G: Optimizing the query execution process in the Grid. In *Proceedings of the ACM/IEEE SC2002 Conference*, November 2002.

[3] Henrique Andrade, Tahsin Kurc, Alan Sussman, and Joel Saltz. Multiple query optimization for data analysis applicationson clusters of SMPs. In *Proceedings of the 2nd IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid)*. IEEE Computer Society Press, May 2002.

[4] Henrique Andrade, Tahsin Kurc, Alan Sussman, and Joel 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.

[5] Richard E. Bellman. *Adaptive Control Processes: A Guided Tour*. Princeton University Press, NJ, 1961.

[6] Michael Beynon, Chialin Chang, Umit Catalyurek, Tahsin Kurc, Alan Sussman, Henrique Andrade, Renato Ferreira, and Joel 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.

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

[8] Fa-Chung Fred Chen and Margaret H. Dunham. Common subexpression processing in multiple-query processing. *Transactions on Knowledge and Data Engineering*, 10(5):493–499, 199.

[9] Ya-lun Chou. section 17.9, Statistical Analysis. *Holt International*, 1975.

[10] Minos Garofalakis and Yannis Ioannidis. Multi-dimensional resource scheduling for parallel queries. In *Proceedings of 1996 ACM SIGMOD International Conference on Management of Data (SIGMOD)*, 1996.

[11] Bugra Gedik and Ling 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.

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

[13] Amit Gupta, S Sudarshan, and Sundar vishwanathan. Query scheduling in multi query optimization. In *International Database Engineering and Applications Symposium (IDEAS'01)*, pages 11–19, 2001.

[14] Antonin 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.

[15] Ibrahim Kamel and Christos Faloutsos. Parallel R-trees. In *Proceedings of 1992 ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 195–204, 1992.

[16] Myong H. Kang, Henry G. Dietz, and Bharat K. Bhargava. Multiple-query optimization at algorithm-level. *Data and Knowledge Engineering*, 14(1):57–

75, 1994.

[17] Jik-Soo Kim, Henrique Andrade, and Alan 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, April 2005.

[18] Jik-Soo Kim, Henrique Andrade, and Alan Sussman. Principles for designing data-/compute-intensive distributed applications and middleware systems for heterogeneous environments. *Journal of Parallel and Distributed Computing*, 67(7):755–771, July 2007.

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

[20] Manish Mehta and David J. DeWitt. Managing intra-operator parallelism in parallel database systems. In *Proceedings of the 21st International Conference on Very Large Data Bases (VLDB)*, pages 382–394, 1995.

[21] Mohamed F. Mokbel, Xiaopeng Xiong, Moustafa A. Hammad, and Walid 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.

[22] Anirban Mondal, Masaru Kitsuregawa, Beng Chin Ooi, and Kian Lee 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.

[23] Beomseok Nam, Henrique Andrade, and Alan Sussman. Multiple range query optimization with distributed cache indexing. In *Proceedings of the ACM/IEEE SC2006 Conference*, 2006.

[24] Manuel Rodríguez-Martínez and Nick 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.

[25] Prasan Roy, S Sehadri, S Sudarshan, and Siddhesh Bhobe. Efficient and extensible algorithms for multi-query optimization. In *Proceedings of 2000 ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 249–260, 2000.

[26] Bernd Schnitzer and Scott 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.

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

[28] Aman Sinha and Craig Chase. Prefetching and caching for query scheduling in a special class of distributed applications. In *Proceedings of the 1996*

International Conference on Parallel Processing (ICPP'96), pages 95–102, Bloomingdale, IL, 1996.

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

[30] Xiaopeng Xiong, Mohamed F. Mokbel, Walid G. Aref, Susanne E. Hambrusch, and Sunil Prabhakar. Scalable spatio-temporal continuous query processing for location-aware services. In *Proceedings of 16th International Conference on Scientific and Statistical Database Management (SSDBM)*, 2004.

[31] Kai Zhang, Henrique Andrade, Louiqa Raschid, and Alan 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.

[32] Yihong Zhao, Prasad M Deshpande, Jeffrey F Naughton, and Amit Shukla. Simultaneous optimization and evaluation of multiple dimensional queries. In *Proceedings of 1998 ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pages 271–282, 1998.