Scheduling Multiple Data Visualization Query Workloads on a Shared Memory Machine*

Henrique Andrade†, Tahsin Kurc‡, Alan Sussman†, Joel Saltz‡

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

‡ Dept. of Biomedical Informatics
The Ohio State University
Columbus, OH, 43210
{kurc-1,saltz-1}@medctr.osu.edu

Abstract

Query scheduling plays an important role when systems are faced with limited resources and high workloads. It becomes even more relevant for servers applying multiple query optimization techniques to batches of queries, in which portions of datasets as well as intermediate results are maintained in memory to speed up query evaluation. In this work, we present a dynamic query scheduling model based on a priority queue implementation using a directed graph and a strategy for ranking queries. We examine the relative performance of four ranking strategies and compare them against a first-in first-out (FIFO) scheduling strategy. We describe experimental results on a shared-memory machine using two different versions of an application, called the Virtual Microscope, for browsing digitized microscopy images.

1 Introduction

Multi-query optimization is becoming increasingly important in many application domains, such as relational databases [9, 14, 15, 20, 21, 22, 23], deductive databases [7], decision support systems [27], and data intensive analytical applications (or data analysis applications) [3]. In a collaborative environment, multiple clients may submit queries to the data server. There may be a large number of overlapping regions of interest and common processing requirements among the clients. Several optimizations can be applied to speed up query execution when multi-query workloads are presented to the server. Query scheduling is another common technique that can be applied in combination with other optimizations such as use of materialized views (or intermediate results), and data prefetching and caching. The importance of scheduling lies in the fact that by properly ordering the queries in a parallel environment it is possible to better exploit the database server and extract larger benefits from its parallel capabilities [16]. Query scheduling techniques have been extensively used to speed up the execution of queries in the face of limited resources [11]. Nevertheless, the study of the problem of scheduling multiple query workloads, specifically for highly data intensive visualization queries, and more generally for data analysis applications, is lacking.

The nature of data analysis applications makes query optimization a challenging problem: Datasets consist of application-specific complex data structures. Oftentimes, a user-defined data structure is used to hold intermediate results during processing. In addition, queries in the visualization applications we target, as well

*This research was supported by the National Science Foundation under Grants #ACI-9619020 (UC Subcontract #10152408) and #ACI-9982087, Lawrence Livermore National Laboratory under Grant #B500288 (UC Subcontract #10184497), and the Department of Defense, Advanced Research Projects Agency, USAF, AFMC through Science Applications International Corporation under Grant #F30602-00-C-0009 (SAIC Subcontract #4400025559).

as queries in data analysis applications in general, tend to be expensive in terms of the required processing power and a I/O bandwidth, because user-defined processing of data is an integral part of a query. For these reasons, it is not unusual for them to require a long time to supply results. To improve response times and overall system performance, query optimization and query scheduling are of paramount importance to better leverage the software infrastructure and the underlying hardware resources.

In this paper, we develop a model for scheduling multiple queries. The multi-query scheduling model is based on an implementation of priority queue and dynamic in that we assume queries are submitted to the server dynamically. The priority queue is implemented using a directed graph to represent the commonalities and dependencies among queries in the system and a ranking strategy to sort the queries for execution. We describe four different ranking strategies. We perform an experimental evaluation of the strategies using two versions of a microscopy visualization application that were deployed in a generic, multithreaded, multiple query workload aware runtime system [3]. Each of these versions of the application has different CPU and I/O requirements, creating different application scenarios. We experimentally show how a dynamic query scheduling model that takes into consideration cached results and data transformation opportunities improves system performance by lowering the overall query execution time for a large number of clients submitting multiple queries.

The scheduling model we propose in this work is somewhat dependent on a multiple query aware middleware we have designed and implemented to support data analysis applications [3]. Thus, we present an architectural overview of our testbed in the next section. In Section 3 we describe the visualization application and its two implementations. Section 4 discusses the dynamic query scheduling model and the four scheduling policies we have developed. In Section 5, we present and discuss the experimental results we have obtained. Related research is compared to our work in Section 6. Finally, conclusions and future extensions to our current work are given in Section 7.

2 System Architecture

We have implemented the Virtual Microscope application [2] using a generic multiple query aware middleware for data analysis applications. This middleware infrastructure, which consists of a multithreaded query server engine, was previously described in [3], but we provide a description of the middleware here in order to help with the presentation of the scheduling model.

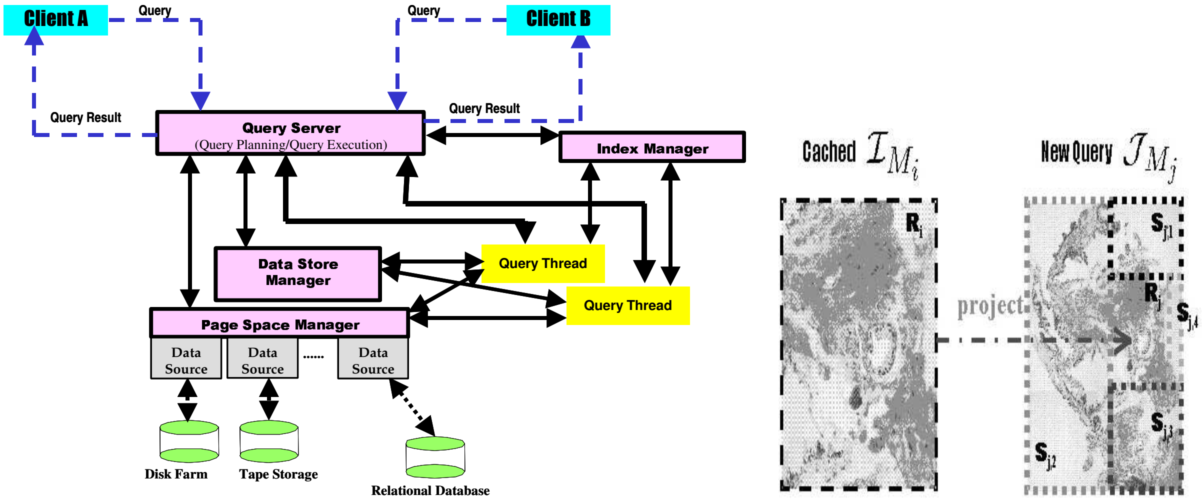
Figure 1 (a) illustrates the architecture of our framework, which consists of several service components, implemented as a C++ class library and a runtime system. The current runtime system supports multithreaded execution on a shared-memory multiprocessor machine, and has been tested on both Linux and Solaris machines.

2.1 Query Server

The query server is implemented as a fixed-size thread pool (typically the number of threads is the number of processors available in the SMP) that interacts with clients for receiving queries and returning query results. A client query includes (1) a query type id, and (2) user-defined parameters to the query object implemented in the system. The user-defined parameters include a *dataset id* for the input dataset, *query meta-information*[1] (e.g., SQL predicate), *query window*, which describes the data of interest, and an *index id* for the index to be used for finding the data items that are part of a given query.

An application developer can implement one or more query objects for application-specific subsetting and processing of datasets. When a query object is integrated into the system, it is assigned a unique *query*

[1]The query predicate meta-information describes which part of the dataset is relevant to satisfy a query, and is domain dependent, e.g., it can be an n-dimensional bounding box in visualization applications or a boolean expression in relational database queries.



(a) (b)

Figure 1: (a) The system architecture. (b) The query answering mechanism. Once a new query q_j with predicate M_j is submitted, the system tries to find a complete or a partial blob that can be used to compute q_j. Once it is found (blob $\mathcal{I}$ region R_i, in our example), a data transformation is applied with the user-defined `project` method to compute region R_j. Sub-queries – $S_{j,1}$, $S_{j,2}$, $S_{j,3}$, and $S_{j,4}$ – are generated to complete the query processing and produce the answer blob $\mathcal{J}$.

type id. The implementation of a new query object is done through C++ class inheritance and the implementation of virtual methods. A *Query* base class is provided for this purpose. A query object is associated with (1) an `execute` method, (2) a query meta-information object `qmi`, which stores query information, and (3) an accumulator object `qbuf`, which encapsulates user-defined data structures for storing intermediate results. A *QueryMI* base class is provided for implementing new accumulator meta-data objects. It provides the stubs for several methods that an application writer is required to customize. Both the query meta-data object and the accumulator meta-data object are implemented by the application developer by deriving a subclass from the base classes provided by the system.

The `execute` method implements the user-defined processing of data. In the current design of the `execute` method, the application developer is expected to implement (1) index lookup operations, (2) operations for the initialization of intermediate data structures, and (3) the processing of data retrieved from the dataset. When a query is received from a client, the query server instantiates the corresponding query object and spawns a *Query Thread* (Figure 1(a)) to *execute* the query. Specifically, when the system receives predicate meta-information M (stored in `qmi`) for a query, query processing begins. First, the runtime system searches for cached partial results that can be reused to either completely or partially answer a query. This lookup operation uses an `overlap` operator to test for potential matches. The `overlap` method interface is defined in the *QueryMI* class, and is implemented by the application developer so that the user-defined accumulator meta-data object associated with the query object can be compared with another accumulator meta-data object for the same query type. Note that the result for one query can also be viewed as an intermediate result for another query. Thus, the meta-data information for an output object is also represented using an accumulator meta-data object. A `project` method is then called so that the cached result can be *projected*, potentially performing a transformation on the cached data, to generate a portion of the output for the current query. The `project` method interface is also defined in the *QueryMI* class and has to be implemented by the application developer. Finally, if the current query is only partially answered by cached results, sub-queries are created to compute the results for the portions of the query that have not been computed from cached results. The sub-queries are processed just like any other query in the system, thereby allowing more intermediate results to be reused. It is possible that a query may be completely answered by multiple intermediate results generated from the results cached from previously executed queries.

The transformation of intermediate data structures and values via the `project` method is the cornerstone of our system architecture. First, it is responsible for the identification of reuse possibilities when a new query is being executed. Second, information about multiple possible overlaps and multiple scheduling possibilities can be used to process a query batch in a way that better utilizes available system resources. Since the `project` method is application specific, for new queries to be added to the system the application developer must write methods by subclassing the infrastructure core classes. Formally, the following equations describes the *data transformation model* our system uses in order to explore common subexpression elimination and partial reuse opportunities:

$$cmp(M_i, M_j) = \textit{true} \text{ or } \textit{false} \tag{1}$$

$$overlap_{project}(M_i, M_j) = k, 0 <= k <= 1 \tag{2}$$

$$\mathcal{I}_{M_i} \overset{project(M_i, M_j, \mathcal{I})}{\rightarrow} \mathcal{J}_{M_j} \tag{3}$$

Equation 1 describes the comparison function cmp that returns *true* or *false* depending on whether intermediate data result $\mathcal{I}$ described by its predicate M_i is the same as $\mathcal{J}$ described by M_j. When cmp returns true, the system has identified a common subexpression elimination opportunity, because query q_j can be completely answered by returning $\mathcal{I}$.

In some situations, queries q_i and q_j, albeit different, have some overlap, which means that cmp is false, but partial reuse is still possible. Equation 2 describes the $overlap$ function that returns a value between 0 and 1 that represents the amount of overlap between intermediate data results $\mathcal{I}$ and $\mathcal{J}$. This overlap is computed with respect to a data transformation function $project$ that needs to be provided by the application developer. The $project$ function, given in Equation 3, takes one intermediate data result $\mathcal{I}$ whose predicate is M_i and *projects* it over M_j (i.e. performs a data transformation) as seen in Figure 1 (b). One additional function also needs to be implemented:

$$qoutsize(M_i) = z, \text{where } z \text{ is the size in bytes of the results for query } q_i \text{ over predicate } M_i \tag{4}$$

Function $qoutsize$ returns the query output size in terms of the number of bytes needed to store the query results. Currently, this function is only used by the query scheduler. For queries whose predicates only allow the query server to know how much space is going to be needed by actually executing the query, it is not feasible to provide the exact amount of space needed to hold the query results beforehand. In these situations, an estimate of that amount should be returned.

The query server interacts with two system components that deal with the I/O subsystem and the collection and management of intermediate data blobs: the *Page Space Manager* (PS) and the *Data Store Manager* (DS). This interaction is to inquire about available memory to better schedule the queries and to send hints for data and page replacement policies.

2.2 Data Store Manager

The data store manager (DS) is responsible for providing dynamic storage space for intermediate data structures generated as partial or final results for a query. Intermediate data structures present one type of commonality among multiple queries, and the middleware can exploit these commonalities to perform operations analogous to compiler optimizations such as common subexpression elimination and partial redundancy elimination [19]. That is, given an intermediate result $\mathcal{I}$ computed by a query q_i, the output $\mathcal{J}$ for a query q_j may be computed from $\mathcal{I}$. One trivial case is when more than one query is able to share the exact same content held by some data structure used by another query. The most important feature of the data store is that it records semantic information about intermediate data structures. This allows the use of intermediate results to answer queries later submitted to the system. A query thread interacts with the data store using a *DataStore* object, which provides functions similar to the C language function *malloc*. When a query wants to allocate space in the data store for an intermediate data structure, the size (in bytes) of the data structure and the corresponding accumulator meta-data object are passed as parameters to the *malloc* method of the data store object. The data store manager allocates the buffer space, internally records the pointer to the buffer space and the associated meta-data object containing a semantic description, and returns the allocated buffer to the caller.

The data store manager also provides a method called `lookup`. This method can be used by the query server to check if a query can be answered entirely or partially using the intermediate results stored in the data store as seen in Figure 1(b). Since the data store manager maintains user-defined data structures and can apply projection operations on those data structure, the projection method should be customized for each intermediate data structure, especially if some data transformation is required. If no data transformation is required, by default the projection method is the identity function.

A hash table is used to access accumulator meta-data objects in the data store manager. The hash table is accessed using the dataset id of the input dataset. Each entry of the hash table is a linked list of intermediate data structures allocated for and computed by previous queries. The `lookup` method calls the `overlap` method for each accumulator meta-data object in the corresponding linked list, and returns a reference to the object that has the largest overlap with the query.

2.3 Page Space Manager

The page space manager (PS) controls the allocation and management of buffer space available for input data in terms of fixed-size pages. All interactions with data sources are done through the page space manager. Queries access the page space manager through a *Scan* object. This object is instantiated with a data source object and a list of pages (which can be generated as a result of index lookup operations) to be retrieved from the data source. The pages retrieved from a data source are cached in memory. The manager implements replacement policies that are specified by the query server. The current implementation of the page space manager uses a hash table for searching pages in the memory cache.

The page space manager also keeps track of I/O requests received from multiple queries so that overlapping I/O requests are reordered and merged, and duplicate requests are eliminated, to minimize I/O overhead. For example, if the system receives a query into a dataset that is already being scanned for another query, the traversal of the dataset for the second query can be *piggybacked* onto the first query in order to avoid traversing the same dataset twice.

2.4 Data Sources

A data source can be any entity used for storing datasets. In the current implementation, data is accessed in fixed-size pages by the Page Space manager. That is, a dataset is assumed to have been partitioned into fixed-size pages and stored in a data source. Therefore, the data source abstraction presents a page-based storage medium to the runtime system, whereas the actual storage can be a file stored on a local disk or a remote database accessed over a wide-area network. When data is retrieved in chunks (pages) instead of as individual data items, I/O overhead (e.g., seek time) can be reduced, resulting in higher application level I/O bandwidth [1]. Using fixed-size pages allows more efficient management of memory space. A base class, called *DataSource*, is provided by the runtime system so that an application developer can implement support for multiple physical devices and data storage abstractions, be it a DVD juke-box or relational tables residing in a DBMS. The base class has virtual methods with semantics similar to the Unix file system (i.e., open, read, write, and close methods). This provides a well-defined interface between the runtime system and the storage medium. An application developer can implement a DataSource object with application- or hardware-specific optimizations. For instance, if there are multiple disks attached to a host machine, a declustering algorithm can be implemented in a DataSource object so that the data pages stored through that object are distributed across multiple disks. As a result, high I/O bandwidth can be achieved when data is retrieved to evaluate a query. We have implemented two data source subclasses, one for the Unix file system and a second to overcome the 2GB file size limitation in the Linux ext2 file system.

2.5 Index Manager

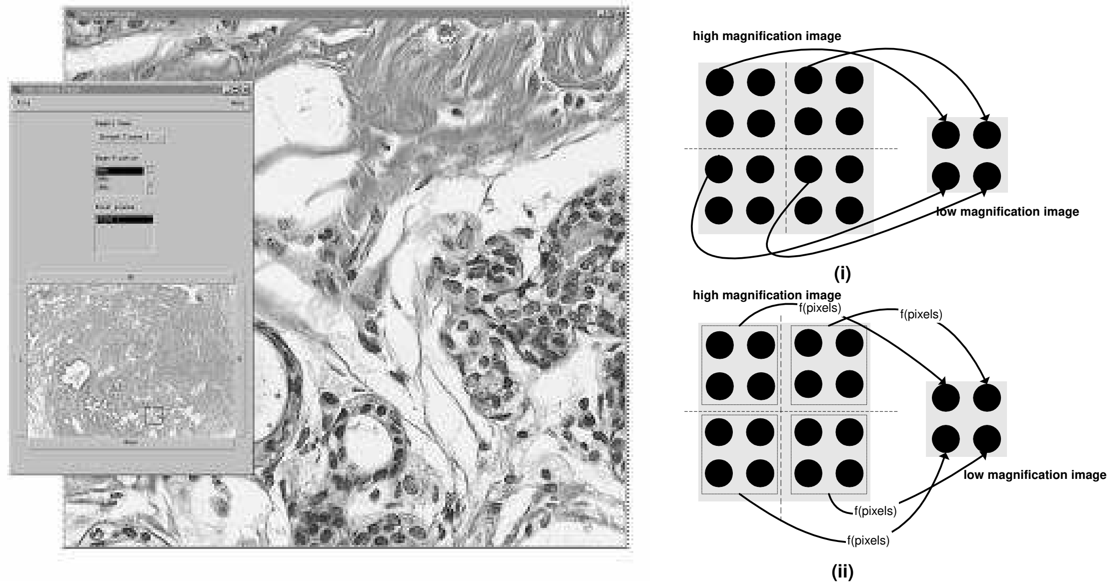
The index manager provides indexing support for the datasets. A query thread interacts with the index manager to access indexing data structures and search for data that intersect with the query. In our current prototype, index lookups produce the list of pages from a data source that are necessary to answer a query.

The integration of spatial indexing mechanisms such as R-Trees [13] and R*-Trees [4], or any other specialized indexing mechanism, is achieved by derivation from base classes defined in our core infrastructure.

3 Analysis of Microscopy Data with The Virtual Microscope

Before we describe the multi-query scheduling model, we present the implementation of a microscopy visualization application using our middleware. The Virtual Microscope (VM) application [2] implements a realistic digital emulation of a high power light microscope. Figure 2(a) displays the VM client GUI. VM can not only emulate the behavior of a physical microscope, including continuously moving the stage and changing magnification, but also provides functionality that is impossible to achieve with a physical microscope. One example of additional functionality is in a teaching environment, where an entire class of students can access and individually manipulate the same slide at the same time, searching for a particular feature in the slide. In that case, the data server may have to process multiple queries simultaneously. Thus it becomes important to efficiently cache intermediate and final results in memory and schedule client queries so as to better use the cached results and better explore data transformation opportunities.

The raw input data for VM can be captured by digitally scanning collections of full microscope slides. Each digitized slide is stored on disk at the highest magnification level and can contain multiple focal planes. The size of a slide with a single focal plane can be up to several gigabytes, uncompressed. In order to achieve high I/O bandwidth during data retrieval, each focal plane in a slide is regularly partitioned into data chunks, each of which is a rectangular subregion of the 2D image. Each pixel in a chunk is associated with a coordinate (in x- and y-dimensions) in the entire image. As a result, each data chunk is associated with a minimum bounding rectangle (MBR), which encompasses coordinates of all the pixels in the chunk. An index is created using the MBR of each chunk. Since the image is regularly partitioned into rectangular regions, a simple lookup table consisting of a 2-dimensional array serves as an index. During query processing, the chunks that intersect the query region, which is a two-dimensional rectangle within the input image, are retrieved from disk. Each retrieved chunk is first clipped to the query window. Each clipped chunk is then processed to compute the output image at the desired magnification. We have implemented two functions to process the high resolution clipped chunks to produce lower resolution images, each of which results in a different version of VM. The first function employs a simple subsampling operation, and the second one implements an averaging operation over a window. For a magnification level of N given in a query, the subsampling function returns every N^{th} pixel from the region of the input image that intersects the query window, in both



(a) (b)

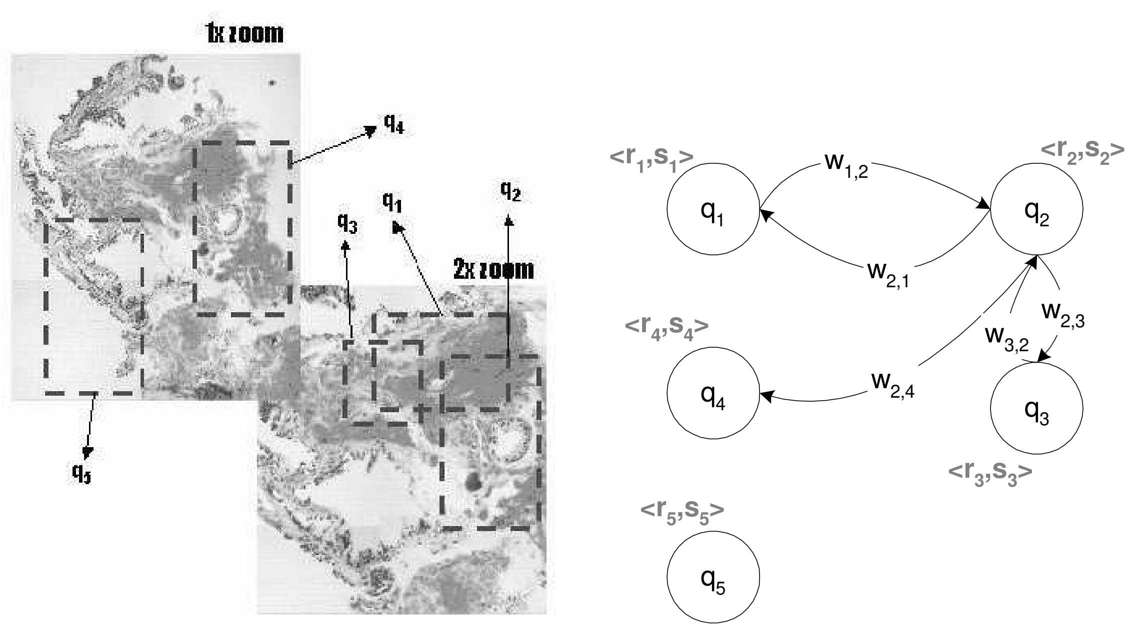
Figure 2: (a) The Virtual Microscope client. (b) Operations to compute images at lower magnification levels from a higher resolution image. Figure (b-i) illustrates the subsampling operation. Figure (b-ii) shows the pixel averaging operation, in which the function $f(pixels)$ computes the mean for a given number of pixels from an image at higher resolution.

dimensions. The averaging function, on the other hand, computes the value of an output pixel by averaging the values of $N \times N$ pixels in the input image. The *averaging* function can be viewed as an imaging processing algorithm in the sense that it has to aggregate several input pixels in order to compute an output pixel. Algorithms such as image enhancement and automatic feature extraction would have similar relative computing and I/O requirements, and thus we argue that the experimental results obtained in this work provide insight into how other similar image processing algorithms would perform on our system. Figure 2(b) illustrates the application of the two functions. The resulting image blocks are directly sent to the client. The client assembles and displays the image blocks to form the query output.

The implementation of VM using the runtime system described in this paper is done by sub-classing two of the base classes provided by the middleware, *Query* and *QueryMI*. We have added two query classes to the runtime system – one for each of the processing functions. The output image generated by a query is also available as an intermediate result and is stored in the data store manager so that it can be used by other queries. The magnification level, the processing function, and the bounding box of the output image in the entire dataset are stored as meta-data. An `overlap` function was implemented to intersect two regions and return an overlap index, which is computed as

$$overlap\ index = \frac{I_A}{O_A} \times \frac{I_S}{O_S} \qquad (5)$$

In this equation, I_A is the area of intersection between the intermediate result in the data store and the query region, O_A is the area of the query region, I_S is the zooming factor used for generating the intermediate result, and O_S is the zooming factor specified by the current query. O_S should be a multiple of I_S so that the query can use the intermediate result. Otherwise, the value of the overlap index is 0.



(a) (b)

Figure 3: (a) Sample queries over two parts of a sample dataset with different magnification levels. (b) The correspondent scheduling graph. Node q_1 is associated with 2-tuple $< r_1, s_1 >$, where r_1 is the node rank, and s_1 is its state. Edge $e_{1,2}$ connects q_1 to q_2 and it means that q_1 can be used to answer q_2 since there is spatial overlap between q_1 and q_2's MBR in the sample query diagram. The edge weight $w_{1,2}$ shows how much reuse will be achieved in terms of bytes ($overlap(q_1, q_2) \times qoutsize(q_1)$) if q_2 is computed after q_1's results are available and are reused. If a query does not have any edges to or from the other nodes, that means that the query does not share any commonalities with other queries, either because its predicate does not overlap with other query predicates or because it operates on another dataset. See, for example, q_5.

4 Dynamic Query Scheduling Model

The query scheduling model presented in this paper assumes the data server is dynamically receiving queries. This dynamic nature of requests creates a need to evaluate the current state of the server system before a new query is slated for execution.

Our approach to the multiple query scheduling problem is based on the use of a priority queue. The priority queue is implemented as a directed graph, $G(V, E)$, where V denotes the set of vertices and E is the set of edges, along with a set of strategies for ranking queries waiting for execution. The graph, referred to as the *query scheduling graph*, describes the dependencies and reuse possibilities among queries because of potentially full or partial overlap as is seen in Figure 3. Each vertex represents a query that is waiting to be computed, is being computed, or was recently computed and its results are cached. A directed edge in the graph connecting q_i to q_j, $e_{i,j}$, means that the results of q_j can be computed based on the results of q_i. In some situations a transformation may exist in only one direction due to the nature of the data transformation, i.e., the data transformation function is not invertible. In that case, the two nodes are connected in only one direction (e.g., $e_{2,4}$ in Figure 3 (b)). Associated with each $e_{i,j}$ is a weight $w_{i,j}$, which gives a quantitative measure of how much of q_i can be used to answer q_j through some transformation function[2]. The weight is computed as the value of $overlap(q_i, q_j) \times qoutsize(q_i)$, which is a measure of the number of bytes that can

[2]Currently, the system stores the weight for a single type of transformation function, namely the best one available.

be reused.

Associated with each node $q_i \in V$ there is a 2-tuple $< r_i, s_i >$, where r_i and s_i are the current rank and the state of q_i, respectively. A query can be in one of the following states `WAITING, EXECUTING, CACHED`, or `SWAPPED_OUT`. The rank is used to decide when a query in the `WAITING` state is going to be executed, so that when a *dequeue* operation is invoked on the priority queue the query node with the highest rank is scheduled for execution.

This scheduling model is dynamic in the sense that new queries can be inserted as they are received from clients. A new query triggers (1) the instantiation of a new node q_j in the query scheduling graph, (2) the addition of edges to all nodes q_k that have some overlap with q_j by adding edges $e_{k,j}$ and $e_{j,k}$ (we refer to these nodes as "neighbor nodes"), and (3) the computation of r_j. Also, all the q_k nodes have their r_k updated, because the new query will potentially affect those nodes in terms of reuse opportunities. The state of a new node is set to `WAITING`. Once a query q_i has been scheduled, its s_i is updated to `EXECUTING`, and once it has finished, its state is updated to `CACHED`, meaning that its results are available for reuse. If the system needs to reclaim memory, the state of the query q_i may be switched to `SWAPPED_OUT`, meaning that its results are no longer available for reuse. At that point the scheduler removes the node q_i and all edges whose source or destination is q_i from the query scheduling graph. This morphological transformation triggers a re-computation of the ranks of the nodes which were previously neighbors with q_i. Therefore the up-to-date state of the system is reflected to the query server. The ranking of nodes can be viewed as a topological sort operation. Updates to the query scheduling graph and topological sort are done in an incremental fashion to avoid performance degradation because of the cost of running the scheduling algorithm.

A dynamic scheduling model, as opposed to a static one, is preferable for the class of applications we target because it is not known *a priori* when the queries will arrive to the system, which parts of the datasets will be required, and what kinds of computations will be required. There is no beforehand knowledge the system could use to base static decisions upon. Hence, the dynamic scheduling model is likely to result in better performance than scheduling algorithms that perform updates only when new queries arrive at the system [24].

The primary objective of our multi-query scheduling model is to minimize the response time of the each query submitted to the system. We define the *response time* of a query as the sum of the waiting time in the queue and the query execution time. A secondary objective is to decrease the overall execution time for all the queries as a batch (i.e. system throughput). Associated with both goals is the objective of making better use of cached data and the data transformation model, to prevent reissuing I/O operations or recomputing results from input data by effectively ranking and sorting queries for execution. In this work, we have developed and examined five different ranking strategies:

1. **First in First out (FIFO).** Queries are served in the order they arrive at the system.

2. **Most Useful First (MUF).** The nodes in the `WAITING` state in the query scheduling graph are ranked according to a measure of how much other nodes depend on a given node, in terms of $overlap \times qoutsize$. The intuition behind this policy is that it quantifies how many queries are going to benefit if we run query q_i first. Formally the rank for each node in `WAITING` state is computed as follows:

$$r_i = \sum_{\forall k | \exists e_{i,k} \wedge s_k = waiting} w_{i,k}$$

3. **Farthest First (FF).** The nodes are ranked based on a measure of how much a node depends upon another node in terms of $overlap \times qoutsize$. When q_i depends on q_j, and if q_i gets scheduled for execution while q_j is executing, the computation of q_i will stall until q_j finishes and its results can be used to generate a portion of the result for q_i. Although this behavior is correct and efficient in the

sense that I/O is not duplicated, it wastes CPU resources because other queries that do not overlap either q_i or q_j cannot be executed because of the limited number of threads in the query server thread pool. Therefore, farthest first rank gives an idea of how probable a query is to block because it depends on a result that is either being computed or is still waiting to be computed. The rank is computed as follows:

$$r_i = - \sum_{\forall k | \exists e_{k,i} \wedge (s_k = waiting \vee s_k = executing)} w_{k,i}$$

The higher the rank for a node is, the greater its chance of being executed next. Therefore, the negative sign in the equation above makes nodes with higher dependences have smaller ranks.

4. **Closest First (CF).** The nodes are ranked using a measure of how many of the nodes a query depends upon have already executed (and whose results are therefore CACHED) or are currently being executed. The rank r_i is computed as:

$$r_i = \sum_{\forall j | \exists e_{j,i} \wedge s_j = cached)} w_{j,i} + \sum_{\forall k | \exists e_{k,i} \wedge s_k = executing)} \xi w_{k,i}, \text{ where } 0 < \xi < 1$$

ξ is a factor that can be hand-tuned in a way to give more or less weight to dependencies that rely on intermediate results still being computed. The fact that a query q_k is still *executing* may cause q_i to block until q_k has finished, in which case q_i sits idle wasting CPU resources while it waits for q_k to finish. The intuition behind this policy is that scheduling queries that are "close" has the potential to improve locality, making caching more beneficial.

5. **Closest and Non-Blocking First (CNBF).** Nodes are ranked by a measure of how many of the nodes that a query depends upon have been or are being executed. We deduct the weight for the nodes being executed, because we would like to decrease the chances of having little overlap due to the system deadlock avoidance algorithm[3], and minimize waiting time because of the query having to block until the availability of a needed data blob that is still being computed. On the other hand, it also includes the nodes that have already been executed and are going to be useful to improve locality. The rank is computed as follows:

$$r_i = \sum_{\forall k | \exists e_{k,i} \wedge s_k = cached)} w_{k,i} - \sum_{\forall j | \exists e_{j,i} \wedge s_j = executing)} w_{j,i}$$

The intuition is the same as that for CF, but CNBF attempts to prevent interlock situations.

The scheduling algorithms are inherently different in that each of them focuses on a different goal. FIFO targets fairness; queries are scheduled in the order they arrive. The goal of MUF is to schedule queries earlier that are going to be the most beneficial for other waiting queries. The objective of FF is to avoid scheduling

[3]An example of a possible deadlock is the following situation: query q_i will reuse results from query q_j, queries $q_{i,1}$ and $q_{i,2}$ are issued to cover the region not overlapped by q_j. While those queries are being issued, another query q_k arrives in the system and detects overlap with $q_{i,1}$ (which is not finished yet – so q_k has to wait for it to finish), and spawns query $q_{k,1}$ to cover the remaining region. At this point, query $q_{i,2}$ is scheduled and detects ana overlap with $q_{k,1}$ (which is also not complete). The result is that q_i is waiting for q_k to finish and vice-versa, a textbook deadlock situation. The deadlock avoidance algorithm maintains a graph whose nodes are the query thread ids, and vertices are *waiting on* relationships. Every time an incoming query tries to find possible overlaps, it checks if adding another vertex to the graph will create a cycle, and therefore, a deadlock situation. If that is true, the reuse is ignored.

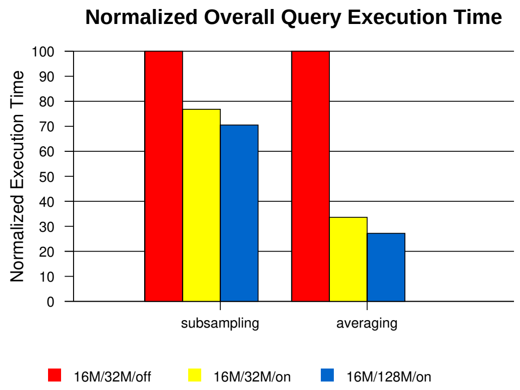


Figure 4: The relative performance improvements for the *subsampling* and *averaging* implementations of the Virtual Microscope queries using FIFO scheduling. The time for each one is normalized to the 16M/32M/off configuration. In the $xx/yy/on|off$ notation, the first number is the amount of memory allocated to the Page Space Manager, the second number is the memory available to the Data Store Manager, and on — off denotes whether the data transformation model and reuse of cached results are enabled or not.

queries that are mutually dependent and have one of them wait for the other to complete, and hence not productively use system resources. CF and CNBF are similar, but CF aims to achieve higher locality to make better use of cached (or soon to be cached) results, while CNBF tries to improve locality without paying the price of having to wait for dependencies to be satisfied.

5 Experimental Results

We have performed an experimental evaluation of the scheduling strategies using the VM application on an 8-processor SMP machine, running version 2.4.3 of the Linux kernel. Each processor is a 550MHz Pentium III and the machine has 4GB of main memory. For the experiments, we have employed three datasets, each of which is an image of size 30000x30000 3-byte pixels, requiring a total of 7.5GB storage space. Each dataset was partitioned into 64KB pages, each representing a square region in the entire image, and stored on the local disk attached to the SMP machine. We have emulated 16 concurrent clients. Each client generated a workload of 16 queries, producing 1024x1024 RGB images (3MB in size) at various magnification levels. Output images were maintained in the Data Storage manager as intermediate results for possible reuse by new queries. Out of the 16 clients, 8 clients issued queries to the first dataset, 6 clients submitted queries to the second dataset, and 2 clients issued queries to the third dataset.

We have used the driver program described in [5] to emulate the behavior of multiple simultaneous clients. The implementation of the driver is based on a workload model that was statistically generated from traces collected from experienced VM users. Interesting regions in a slide are modeled as points, and provided as an input file to the driver program. When a user pans *near* an interesting region, there is a high probability a request will be generated. The driver adds noise to requests to avoid multiple clients asking for the same region. In addition, the driver avoids having all the clients scan the slide in the same manner. The slide is swept through in either an up-down fashion or a left-right fashion as observed from real users. We have chosen to use the driver for two reasons. First, extensive real user traces are very difficult to acquire. Second, the emulator allowed us to create different scenarios and vary the workload behavior (both the number of clients and the number of queries) in a controlled way. In all of the experiments, the emulated clients were

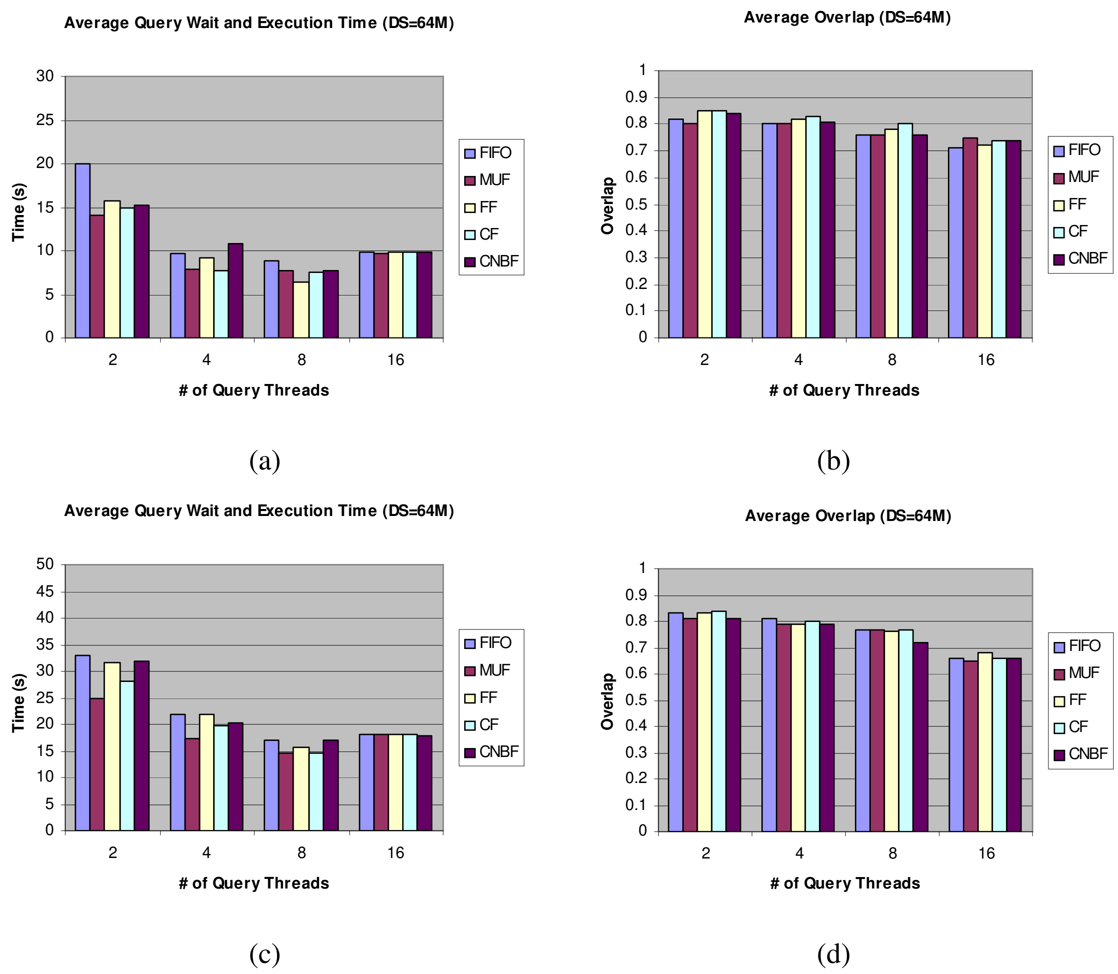
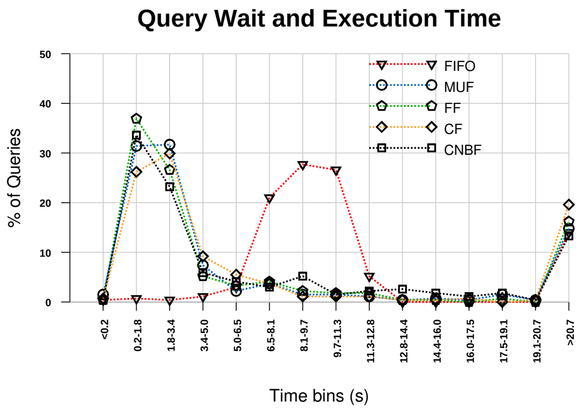


Figure 5: Running the subsampling ((a) and (b)) and the pixel averaging ((c) and (d)) versions of the Virtual Microscope with 64MB allocated to the Data Store Manager. (a) and (c) show the average query wait and execution time for different numbers of simultaneously running query threads for the 5 different policies. (b) and (d) show the average overlap seen by the queries for the different configurations.

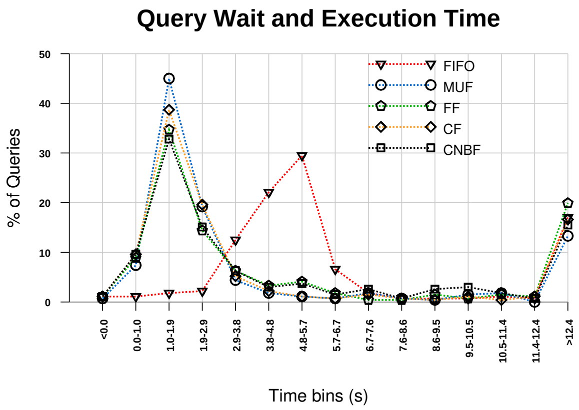
executed simultaneously on a cluster of PCs connected to the SMP machine via 100Mbit Ethernet. Each client submitted its queries independently from the other clients, but waited for the completion of a query before submitting another one.

We present experimental results for the two implementations of VM. These implementations exhibit different application characteristics, making it possible to examine how applications with different relative CPU and I/O requirements perform when multiple query optimization is applied. The first implementation – with the subsampling function – is I/O intensive. The CPU time to I/O time ratio is between 0.04 and 0.06 (e.g., for each 100 seconds, roughly between 4 and 6 seconds are spent on computation, and between 94 and 96 seconds are spent doing I/O). The second implementation – with the averaging function – is more balanced, with the CPU and I/O time ratio near 1:1. A query result using the first implementation is usually computed in approximately half the time for the same query using the second implementation.

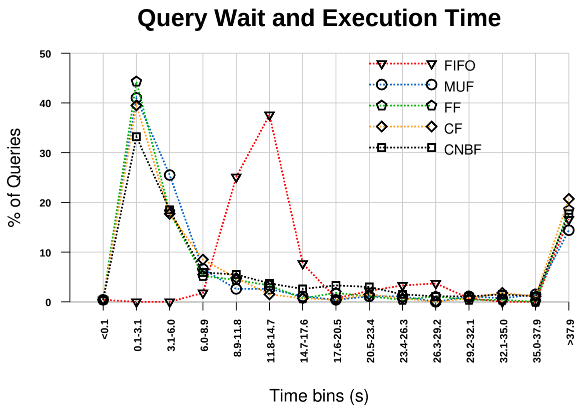
Figure 4 shows the performance benefits of using cached results and the data transformation model implemented in our infrastructure. In these experiments, the FIFO ranking strategy was employed and up to 4 query threads were allowed to concurrently execute in the system. The query execution time is reduced by



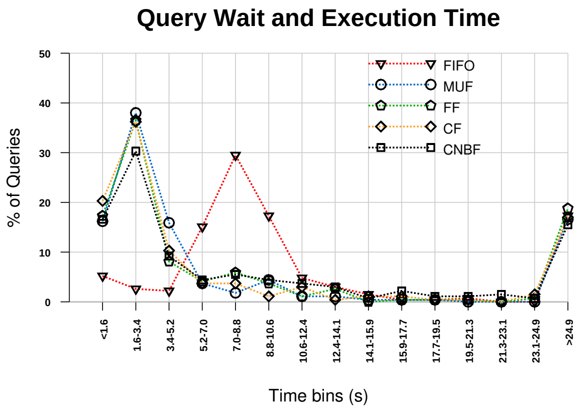
(a)



(b)



(c)



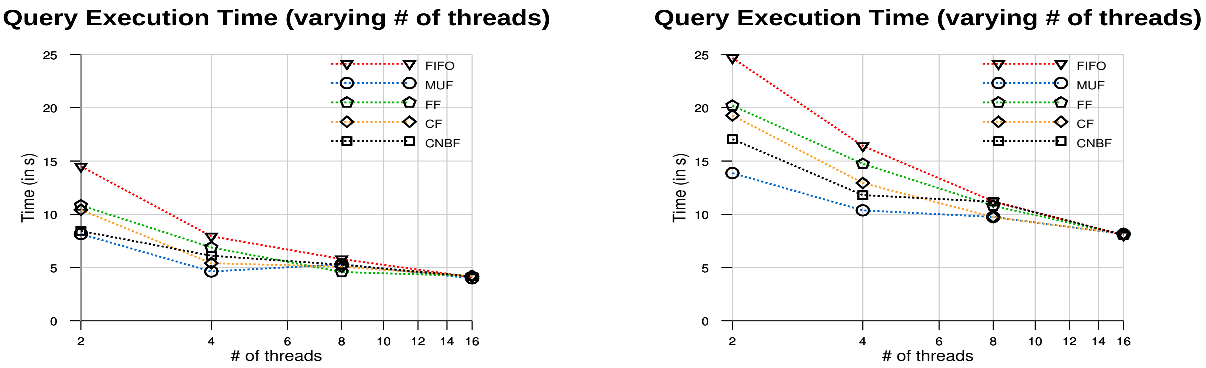
(d)

Figure 6: The density distributions of query response time for the subsampling version ((a) and (b)) and the pixel averaging version ((c) and (d)) with 64MB allocated to the Data Store Manager. The Y-axis is the percentage of queries that complete in a given time period, and the X-axis shows the buckets for time ranges. In (a) and (c) up to 2 simultaneous queries were allowed to execute simultaneously. In (b) and (d) up to 4 simultaneous queries run concurrently.

up to 30% for the version of VM with the subsampling function, and up to 73% for VM with the averaging function.

In the next set of experiments we examine the impact on performance of different ranking strategies (see Section 4)– for the CF strategy, ξ was fixed at 0.2. In the experiments, the sizes of memory allocated for the Page Space and Data Store managers were 16MB and 64MB, respectively. We varied the maximum number of query threads allowed to execute in the system from 2 to 16. Note that when up to 16 query threads are allowed in the system, more than one query may be assigned to a processor since there are only 8 processors on the SMP machine, and the synchronization costs due to additional threads may rise without adding any computational power.

Figure 5 shows the average query response time and the average overlap achieved by different ranking strategies. The query response time is the sum of waiting time in the queue and the execution time of the query and is shown as *Query Wait and Execution* time in the figures. As is seen from the figure, when other policies besides FIFO are employed, we see reductions up to 32% in average response time for the subsam-



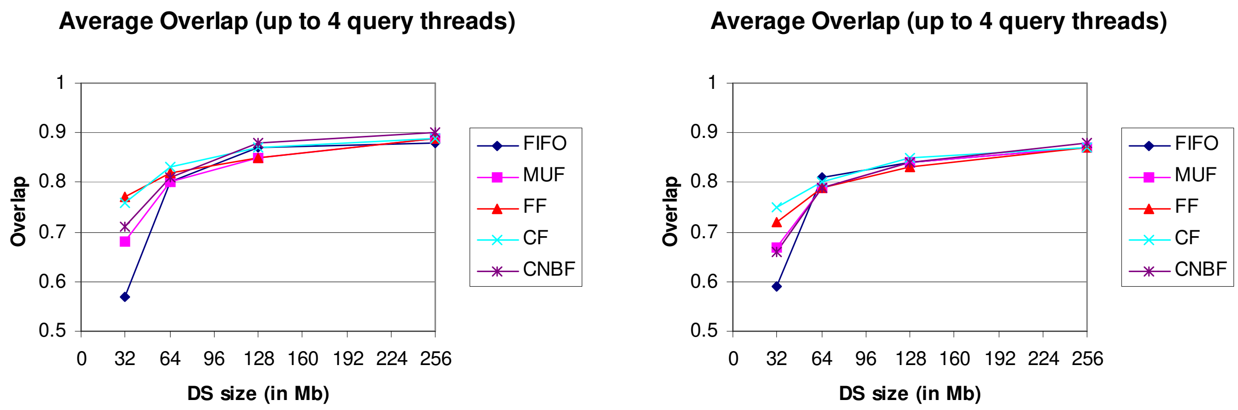
(a) (b)

Figure 7: The 90%-trimmed mean for the query execution time shows how each query scheduling policy scales when more processors (i. e. more queries) are activated at the same time. When 16 processors are available, no gain is observed for any scheduling policy. (a) The subsampling version. (b) The pixel averaging version of VM.

pling version of VM, and up to 30% with the averaging implementation of VM. One interesting result shown in the figure is that policies FF and especially CF induce the highest average overlap during query processing, but that does not get translated into lower execution times. In most of the experimental configurations, MUF is the policy with the lowest average individual query wait and execution time. This apparent contradiction occurs because much of the cost of retrieving more pages from the data source is hidden by the Page Space manager. In cases where I/O cost is high, CF and FF definitely perform better. CNBF shows its best performance when up to 16 simultaneous queries are allowed into the system, although it is only marginally better than the other policies. Its superiority in those cases is because CBNF tries to schedule non-interlocking queries. In a situation where blocking is highly probable due to the number of possible simultaneous queries, this provides the best performance.

The performance benefits of query scheduling in terms of *average* query response time are somewhat misleading, because a large number of queries will see an improvement much higher than the average. Figure 6 displays the distribution of query response times for the different query scheduling strategies. The response time for most of the queries scheduled using the MUF, FF, CF, and CNBF ranking strategies are to the left of the spike for the FIFO strategy, meaning that the bulk of the queries execute faster with other strategies than with FIFO. Figure 7 shows that when we plot the 90%-trimmed mean for the query execution time[4], for almost all configurations all four ranking strategies are better than FIFO. In particular, the MUF policy gives improvements up to 45% for the subsampling implementation and up to 44% for the averaging implementation of VM. The improvements are inversely proportional to the maximum number of queries running concurrently. However, scheduling almost always provides some performance benefits. All of the ranking strategies behave poorly when up to 16 query threads are enabled. That is because the system is not completely scalable, and for that many threads the I/O subsystem cannot keep up with the amount of requests it receives. The averaging implementation of VM displays better scalability with respect to the number of query threads because it is more balanced in terms of CPU and I/O usage, therefore adding more processing power benefits that implementation more.

[4]A trimmed mean is calculated by discarding a certain percentage of the lowest and the highest samples and then computing the mean of the remaining ones. For example, a 90%-trimmed mean is computed by discarding the lower and higher 5% of the scores and taking the mean of the remaining scores [17].



(a) (b)

Figure 8: Average overlap increases when more memory is added to the Data Store Manager. (a) The subsampling implementation. (b) The pixel averaging implementation.

Figure 8 shows the relative performance of the ranking strategies when the size of memory allocated to the Data Store manager is varied. In these experiments, we allocated 32MB, 64MB, 128MB, or 256MB for the Data Store manager. Since each query takes up to 3MB, with 32MB allocated for the Data Store manager, up to 10 queries have enough space to be computed at the same time, assuming that nothing is stored in cache. Therefore, the 32MB configuration displays memory contention because too little space is available for caching. As is seen from the figure, all the policies are positively affected by the addition of more memory to the Data Store Manager, as expected. More memory means that more partial results can be saved and leveraged to answer new incoming queries. This is reflected in two metrics we have collected: the average overlap and the query response time. The charts in Figure 8 also show an interesting result. Although in most of the configurations we have studied MUF is the policy that yields the best performance, CF is the policy yielding the highest overlap, especially for the subsampling version of VM. The more overlap a query sees, the less I/O it incurs. Hence, if I/O is expensive, CF will likely provide the lowest query wait and execution time. We have recently ported the query execution engine to the Solaris operating system. Figure 9 shows the 90%-trimmed mean for query execution time on a 24-processor Solaris SunFire SMP machine. In Solaris, the operating system file cache can be completely turned off, and no I/O amortization happens other than that caused by the Page Space Manager. In this case, scheduling the queries with the CF policy results in the best performance.

6 Related Work

The query optimization and scheduling problem has been extensively investigated in past surveys [8, 11]. The work of Mehta et al. [18] is one of the first that has tackled the problem of scheduling queries in a parallel database by considering batches of queries, as opposed to one query at a time. The authors propose algorithms to explore common operations being executed in the context of a batch. They target sharing data structures for joins and selects, and the goal of their scheduling algorithms is to find the global schedule for all queries that minimizes the total execution time of a given batch.

Traditionally, multiple query optimization techniques rely on caching common subexpressions [22]. Cache space is limited by nature, and it is very well possible that the space available will not be enough

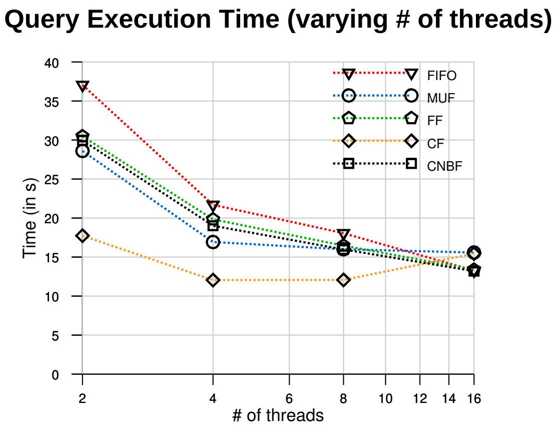


Figure 9: The 90%-trimmed mean for the query execution time on a 24-processor SunFire, showing results for each query scheduling policy with no operating system level file caching, for the subsampling implementation of VM.

to store all the common subexpressions detected. Carefully scheduling queries plays an important role, because one could craft the query execution order in a way to better exploit expressions that have been already cached. Gupta et al. [12] present an approach that tackles this problem in the context of decision support queries. Although our objectives have some similarities to theirs, we deal with a different domain of queries and a different database architecture. Sinha and Chase [24] present some heuristics to minimize the flow time of queries and to exploit interquery locality for large distributed systems. Their motivating application is the Nile application, which is a particle collision simulation. The objective of their heuristic is mainly to decide which parts of the dataset should be present in the local cache of each of the processing nodes of the distributed environment. That work is similar to ours in that they tackle data intensive analysis applications, as does our infrastructure. On the other hand, in that work only input data fetching reorder is employed as an optimization, as opposed to the data transformations that are allowed in our runtime system. Waas and Kersten [26] present scheduling techniques for multiple query workloads generated by web-based requests. Although in a different domain, this work has similarities to our work in two ways. First, it deals with datasets that are mostly read-only. Second, their scheduling algorithm, instead of trying to model system parameters, attempts to leverage data that is currently cached based on a metric that determines the distance between those data and what a waiting query needs. That approach is similar to what we model in our scheduling graph.

Incorporating low level information (e.g., resource availability) into query scheduling models usually makes the computation of the next query to be executed more expensive due to the increased complexity of the scheduling model. Several researchers have targeted this problem using simulation models. Bouganim et al. [6] target query scheduling in a data integration system in order to cope with irregular data delivery rate from multiple sources, as well as to deal with varying computing and I/O resources. Although their focus is different than ours, it shows possible ways of incorporating low level requirements in terms of available memory in our scheduling algorithms. Garofalakis and Ioannidis [10] present a model to handle query scheduling on hierarchical parallel systems. This work is interesting in that it is one of the first that we are aware of that handles the complexities of modeling the multidimensional aspect of the multiple resources needed to compute relational queries.

7 Conclusions

We have developed a model for scheduling multiple queries in data analysis applications. Our results show that efficient scheduling of multiple queries is important to increase the performance of the server and decrease query response time. We have experimentally evaluated four different strategies against the FIFO policy using an application for visualizing digitized microscopy images. The experimental results obtained for two different versions of the application, which exhibit different computation to I/O ratios, show that all of the four strategies perform better than FIFO. Some of them are able to achieve substantial performance increases, specially in situations in which resources are scarce to the requirements of the queries.

We plan to extend this work on three fronts: (1) additional visualization applications (e.g., scientific visualization of 3-dimensional datasets) and other data analysis applications, (2) development of the capability for self-tuning, and (3) the incorporation of low level metrics into our query scheduling model. New visualization applications will allow more extensive evaluation of the middleware, both for performance improvements and for the suitability of the query scheduling policies to application-specific characteristics. We plan to look into extensions for self-tuning at the query server so that the server can change its scheduling decisions automatically based on the query workload and the availability of processing, I/O, and network bandwidth resources. Finally, adding low-level metrics to the scheduling model can enable the server to better control resource usage and avoid thrashing, as we have seen experimentally in situations when more queries than the number of processors available in the system are admitted for processing. This capability can help achieve better speedups when the underlying hardware resource availability scales at different rates for different resource, such as I/O, CPU, and communication bandwidth. Our experimental platforms, for example, allow for the addition of more CPU power, but the interprocessor communication network as well as the I/O subsystem have limited bandwidth. Hence, in this work, while the computational part of the queries scaled well, the same did not apply for the I/O part. As was seen in the performance results, admitting more queries for execution imposed can impose too high a load on available I/O resources, contributing to an overall degradation in system performance. Techniques analogous to *traffic shaping* [25], employed in communication networks to avoid worsening congestion problems, should be utilized for all resources.

To tackle these issues, we plan to work on more complex scheduling techniques as well as on the adaptation of our generic middleware to clusters of SMP nodes, in such a way that more resources can be applied to the applications in order to support larger workloads.

References

[1] Anurag Acharya, Mustafa Uysal, Robert Bennett, Assaf Mendelson, Michael D. Beynon, Jeff Hollingsworth, Joel Saltz, and Alan Sussman. Tuning the performance of I/O-intensive parallel applications. In *Proceedings of the Fourth ACM Workshop on I/O in Parallel and Distributed Systems*, May 1996.

[2] Asmara Afework, Michael D. Beynon, Fabian Bustamante, Angelo Demarzo, Renato Ferreira, Robert Miller, Mark Silberman, Joel Saltz, Alan Sussman, and Hubert Tsang. Digital dynamic telepathology - the Virtual Microscope. In *AMIA98*. American Medical Informatics Association, Nov 1998.

[3] Henrique Andrade, Tahsin Kurc, Alan Sussman, and Joel Saltz. Efficient execution of multiple workloads in data analysis applications. In *Proceedings of the 2001 ACM/IEEE SC'01 Conference (to appear)*, Denver, CO, November 2001.

[4] Norbery Beckmann, Hans-Peter Kriegel, Ralf Schneider, and Bernhard Seeger. The R^*-tree: An efficient and robust access method for points and rectangles. In *Proceedings of the 1990 ACM-SIGMOD Conference*, pages 322–331, May 1990.

[5] Michael D. Beynon, Alan Sussman, and Joel Saltz. Performance impact of proxies in data intensive client-server applications. In *Proceedings of the 1999 International Conference on Supercomputing*. ACM Press, June 1999.

[6] Luc Bouganim, Françoise Fabret, C. Mohan, and Patrick Valduriez. Dynamic query scheduling in data integration systems. In *Proceedings of the 16th International Conference on Data Engineering*, pages 425–434, San Diego, CA, 2000.

[7] Upen S. Chakravarthy and Jack Minker. Multiple query processing in deductive databases using query graphs. In *Proceedings of the 12th VLDB Conference*, pages 384–391, 1986.

[8] Surajit Chaudhuri. An overview of query optimization in relational systems. In *Proceedings of the ACM Symposium on Principles of Database Systems on Principles of Database Systems*, pages 34–43, Seattle, WA, 1998.

[9] Fa-Chung Fred Chen and Margaret H. Dunham. Common subexpression processing in multiple-query processing. *Transactions on Knowledge and Data Engineering*, 10(3):493–499, 1998.

[10] Minos N. Garofalakis and Yannis E. Ioannidis. Parallel query scheduling and optimization with time- and space-shared resources. In *Proceedings of the 23th VLDB Conference*, pages 296–305, Athens, Greece, 1997.

[11] Goetz Graefe. Query evaluation techniques for large databases. *ACM Computing Surveys*, 25(2), 1993.

[12] 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, Grenoble, France, 2001.

[13] Antonin Guttman. R-Trees: A dynamic index structure for spatial searching. In *Proceedings of the 1984 ACM-SIGMOD Conference*, pages 47–57, June 1984.

[14] Anant Jhingran. A performance study of query optimization algorithms on a database system supporting procedures. In *Proceedings of the 14th VLDB Conference*, pages 88–99, 1988.

[15] 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.

[16] Yu-Kwong Kwok and Ishfaq Ahmad. Static scheduling algorithms for allocating directed task graphs to multiprocessors. *ACM Computing Surveys*, 31(4):406–471, 1999.

[17] David M. Lane. *HyperStat Online: An Introductory Statistics Book and Online Tutorial for Help in Statistics Courses*. 1999. http://davidmlane.com/hyperstat/.

[18] Manish Mehta, Valery Soloviev, and David J. DeWitt. Batch scheduling in parallel database systems. In *Proceedings of the 9th International Conference on Data Engineering*, Vienna, Austria, 1993.

[19] Steve S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, San Francisco, CA, 1997.

[20] Arnon Rosenthal and Upen S. Chakravarthy. Anatomy of a modular multiple query optimizer. In *Proceedings of the 14th VLDB Conference*, pages 230–239, 1988.

[21] Prasan Roy, S. Seshadri, S. Sudarshan, and Siddhesh Bhobe. Efficient and extensible algorithms for multi query optimization. In *Proceedings of the 2000 ACM-SIGMOD Conference*, pages 249–260, 2000.

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

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

[24] 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*, pages 95–102, Bloomingdale, IL, 1996.

[25] Andrew S. Tanenbaum. *Computer Networks*. Prentice Hall PTR, Upper Saddle River, NJ, 1996.

[26] Florian Waas and Martin L. Kersten. Memory aware query scheduling in a database cluster. Technical Report INS-R0021, Centrum voor Wiskunde en Informatica, October 2000.

[27] Yihong Zhao, Prasad M. Deshpande, Jeffrey F. Naughton, and Amit Shukla. Simultaneous optimization and evaluation of multiple dimensional queries. In *Proceedings of the 1998 ACM-SIGMOD Conference*, pages 271–282, 1998.