Efficient Execution of Multiple Query Workloads in Data Analysis Applications*

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

† Dept. of Computer Science
University of Maryland
College Park, MD 20742

+ Dept. of Biomedical Informatics
The Ohio State University
Columbus, OH, 43210

{hcma,kurc,als,saltz}@cs.umd.edu

ABSTRACT

Applications that analyze, mine, and visualize large datasets are considered an important class of applications in many areas of science, engineering, and business. Queries commonly executed in data analysis applications often involve user-defined processing of data and application-specific data structures. If data analysis is employed in a collaborative environment, the data server should execute multiple such queries simultaneously to minimize the response time to clients. In this paper we present the design of a runtime system for executing multiple query workloads on a shared-memory machine. We describe experimental results using an application for browsing digitized microscopy images.

1. INTRODUCTION

The efficient storage, management, and manipulation of large datasets is important in many fields of science, engineering and business. Simulations and experimental measurements are the main sources of data in engineering and scientific applications. In environmental simulations, for example, the output of a simulation is one or more datasets, each consisting of a large number of numerical values. A scientist often needs to access, explore and analyze the datasets to gain insight into the problem at hand and to draw meaningful and useful conclusions from the raw data. Large volumes of data are also being gathered from business transactions by commercial institutions. The data has to be queried, mined and manipulated to discover interesting and important patterns so that an institution can make better business decisions.

Data analysis applications often access a subset of all the data available in a dataset. The data of interest is then processed to transform it into a new data product. That is, a query into the dataset includes a reference to a user-defined function for processing the data. In some cases data analysis is employed in a collaborative environment, where multiple clients access the same datasets and perform similar processing on the data. For instance, in medical training, a large group of students may want to simultaneously explore the same set of digitized microscopy slides, or visualize the same MRI and CT datasets. In that case, the data server needs to execute multiple queries simultaneously to minimize latencies to the clients. In a multi-client environment, there may be a large number of overlapping regions of interest, and common processing requirements (e.g., the same magnification level for microscopy images, or use of the same transfer functions to convert scalar values into color values) among the clients. Better utilization of system resources can be achieved if commonalities across multiple queries are exploited. For instance, intermediate data structures computed by a query can be used by another query, if the two queries have overlapping regions of interest and the same processing requirements.

Providing support for efficient execution of multi-query workloads, which includes applying common optimization techniques such as detection of common subexpressions in query plans, use of materialized views (or intermediate results), and data prefetching and caching, in data analysis applications, is a challenging issue for several reasons: (1) In many cases, datasets consist of application-specific complex data structures. For example, the datasets generated by scientific simulations or experimental measurements are often multi-dimensional and multi-resolution. That is, each data item in a dataset is associated with coordinates in a multi-dimensional space. Data dimensions can be spatial coordinates, time, or simulation conditions such as temperature and velocity. In some cases, the mesh modeling the physical domain can be unstructured and multi-resolution; therefore the data items can be irregularly and sparsely distributed in the underlying space. (2) Oftentimes, a user-defined data structure, which we call an *accumulator*, is used

*This research was supported by the National Science Foundation under Grants #ACI-9619020 (UC Subcontract #10152408) and #ACI-9982087, the Office of Naval Research under Grant #N6600197C8534, 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).

SC2001 Nov 2001, Denver ©2001 ACM 1-58113-293-X/01/0011 $5.00.

to hold intermediate results during processing. For example, a z-buffer can be used as an intermediate data structure to hold color and distance values in isosurface rendering of volume datasets. In data mining, histograms and count tables can be used to hold the distribution of attribute values for building decision trees [20]. (3) User-defined processing of data is an integral part of a query. User-defined functions may be simple operations such as minimum, sum and average, or complex functions such as visualization or data mining operations.

In earlier work we developed runtime support and examined strategies for efficient execution of data analysis applications on distributed-memory parallel machines with a disk farm [7, 14] and in distributed, heterogeneous environments [4]. Our previous work has focused on the efficient execution of a single query or a group of unrelated queries. In this work we design a framework to examine efficient strategies and runtime support for the execution of multiple query workloads in data analysis applications. We target data mining applications and applications that analyze, explore, and visualize large multi-dimensional, multi-resolution scientific datasets. Our goal is to develop (1) optimizations for scheduling system resources and operations, (2) data caching, (3) maintaining and using intermediate results, and to implement support for distributed and multi-threaded execution, when a multiple query workload is presented.

In this paper we describe the design of a runtime system for shared-memory multiprocessors. This system aims to improve the execution of multiple queries by (1) maintaining intermediate data structures generated by queries, (2) caching input data in memory, and (3) providing support for multi-threaded execution (i.e., each query is executed as a thread). We describe initial experimental results using an application, called the Virtual Microscope [2], for browsing digitized microscopy images.

2. SYSTEM ARCHITECTURE

Figure 1 illustrates the architecture of the framework, which consists of several service components, implemented as a C++ class library, and a runtime system. Although all the services can be customized and extended by an application developer through C++ class inheritance, we anticipate that most application developers will develop query objects to integrate application-specific operations on data into the runtime system. The runtime system is designed to support multi-threaded execution on shared-memory multiprocessor machines.

2.1 Query Server

The query server interacts with clients for receiving queries and returning query results. An application developer has to implement one or more query objects so that application-specific sub-setting and processing of data can be carried out as an integral part of query execution. When a query object is integrated into the system, it is assigned a unique *query type id*. Hence, a client query includes (1) a query type id, and (2) user-defined parameters to the corresponding query object implemented in the system. The user-defined parameters include dataset ids for the input datasets, query meta-information (e.g., predicate, query window), which describes the data of interest, and index ids for the indices to use for finding the data items that intersect the query.

The implementation of a new query object is done through

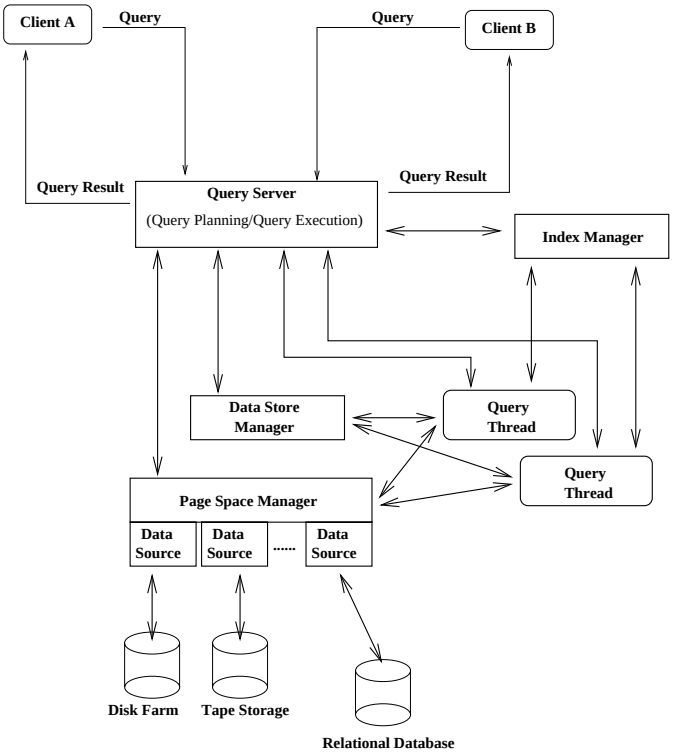


Figure 1: The system architecture.

C++ class inheritance and virtual methods. Two base classes, namely *Query* and *QueryMI*, are provided by the system for this purpose. A query object, which is implemented as a sub-class of the Query base class, is associated with (1) an `execute` function, (2) a query meta-data object, which stores query information, and (3) an accumulator meta-data object, which encapsulates user-defined data structures for storing intermediate results. The `execute` method implements the user-defined processing of data. In the current implementation, the application developer needs to implement index lookup operations, operations for the initialization of intermediate data structures, and operations for the processing of data retrieved from the dataset in the `execute` method. The QueryMI base class, containing a virtual method called `overlap`, is provided for implementing new accumulator meta-data objects via sub-classing the base class. The `overlap` method 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. The final result for a query can also be viewed as an intermediate result for another query. Therefore, the meta-data information for the output is also represented using an accumulator meta-data object.

When a query is received from a client, the query server instantiates the corresponding query object and spawns a *Query Thread* (Figure 1) to execute the query. The query server interacts with the *Page Space Manager* (PS) and the *Data Store Manager* (DS) to inquire about available memory to better schedule the queries and to send hints for data and page replacement policies. The least-recently used (LRU) policy is the only one currently implemented in the data

store manager.

2.2 Data Sources

A data source can be any entity used for storing datasets. In the current implementation, data is accessed in fixed-size pages. 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 currently stored in the 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 overheads (e.g., seek time) can be reduced, resulting in higher application level I/O bandwidth [1]. Also, using fixed-size pages allows more efficient management of memory space. Currently, we have two data source implementations; one for the Unix file system, and a second one that circumvents the 2GB file size limitation in the *ext2* file system (a commonly used file system for Linux) by breaking up big files into multiple smaller, up to 2GB files.

A base class, called *DataSource*, is provided by the system so that an application developer can implement new data sources. 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 incorporate application- or hardware-specific optimizations into a DataSource object. 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.

2.3 Page Space Manager

The page space manager (PS) controls the allocation and management of buffer space available for input data. 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) 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 overheads. 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 *piggy-backed* onto the first query in order to avoid traversing the same dataset twice.

The design of a cache model is closely related to the application and datasets handled by the cache. In some cases, data elements in a dataset can be distributed irregularly (e.g., grid points in unstructured, multi-resolution grids). In particular, cache blocks can be non-rectangular and variable sized. More complex cache models and index structures are required to achieve good performance in those cases. Updates to the indices, when blocks are inserted or deleted from the cache, should be done efficiently to reduce cache lookup costs. Although general cache replacement policies such as LRU can provide good performance, better cache utilization can be achieved through a cache replacement policy that takes into account the request patterns of an application. Application-specific prefetching can be incorporated into data caching. For example, visualization of time varying data may request data elements in consecutive time steps. In that case, while one set of time steps is visualized, data for the following steps can be prefetched. We are planning to add prefetching and implement several application-specific replacement policies in the near future.

2.4 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. That is, given an intermediate result X computed by a query $Q1$, the output Y for a query $Q2$ may be computed from X. One trivial case is when more than one query is able to share the exact same data structure. The most important feature of the data store is that it allows a user-defined query to store semantic information about intermediate data structures. This property enables the use of intermediate results to evaluate queries. A query thread interacts with the data store using a *DataStore* object, which provides functions similar to the C language *malloc*. When the query wants to allocate space from 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, records a pointer to the buffer space and meta-data object, and returns the allocated buffer to the caller.

The data store manager implements a method, called `lookup`. This method can be used by a query or the query planner to check if a query can be answered entirely or partially using the intermediate results stored in the data store. Since the data store manager maintains user-defined data structures, the `lookup` method calls the `overlap` method implemented by the application developer (see Section 2.1) for the corresponding intermediate data structures. The `overlap` method takes an accumulator object and returns a real number between 0 and 1, called the *overlap index*. The number represents the percent overlap between the current query and the intermediate result in the data store. If the overlap index is 1, it means the query overlaps entirely with the intermediate result, so the query can be answered by using the intermediate result. If the overlap index is less than 1, the `execute` function of the query is expected to generate sub-queries to retrieve the subsets of the dataset required to evaluate the portions of the query (which cannot be computed using the intermediate results), and to pass the sub-queries to the query server.

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

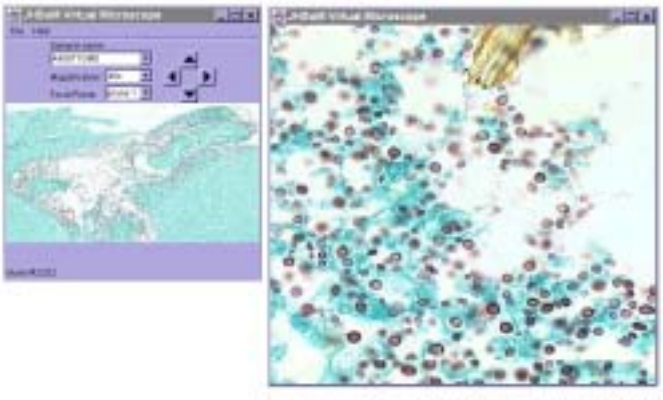


Figure 2: The Virtual Microscope client.

returns a reference to the object that has the largest overlap with the query.

2.5 Index Manager

The index manager manages indices defined on the datasets. The query thread interacts with the index manager to access index data structures and search for data that intersect with the query.

3. AN EXAMPLE APPLICATION: ANALYSIS OF MICROSCOPY DATA WITH THE VIRTUAL MICROSCOPE

In this section, we describe the implementation of an application, called the Virtual Microscope (VM) [2], using our framework. The VM application implements a realistic digital emulation of a high power light microscope. Figure 2 shows the client interface for VM. At a basic level, the Virtual Microscope can emulate the use of a physical microscope, including continuously moving the stage and changing magnification. However, the VM application can provide functionality that a physical microscope can never achieve. One example of additional capability 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.

The raw input data for VM can be captured by digitally scanning collections of full microscope slides under high power. A slide 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 are retrieved from disk. Each retrieved chunk is first clipped to the query window. Afterwards, each clipped chunk is sub-sampled to achieve the magnification level (sub-sampling factor) given in the query. 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, `Query` and `QueryMI`, creating a total of four new objects, *VMQuery*, *MIVMQuery*, *VMQueryMI*, and *MIVMQueryMI*. The *VMQueryMI* and *MIVMQueryMI* objects store the meta-information associated with queries (e.g. dataset name, predicates, and MBRs) and intermediate data structures. The *MIVMQuery* object is used to query the server about datasets that are currently stored in the server. The VM queries are executed by the *VMQuery* object. 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 and the bounding box of the output image in the entire dataset is stored as meta-data. An `overlap` function was implemented to intersect two regions and return an overlap index, which is computed as

$$overlap\ index = \frac{I_A}{O_A} \times \frac{I_S}{O_S} \quad (1)$$

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 sub-sampling factor used for generating the intermediate result, and O_S is the sub-sampling factor specified by the current query. Note that 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. When a VM query is received by the server, a *VMQuery* object is instantiated and spawned as a query thread. The `execute` method of the *VMQuery* object first calls the `overlap` method to find the intermediate results in the data store manager that can be used in the evaluation of the query. A set of sub-queries is then generated, each of which corresponds to a subregion of the query window that cannot be satisfied by the intermediate results. Each sub-query is executed as a new VM query. If there are no intermediate results that overlap the query, the `execute` method performs an index lookup to find the pages that intersect the query and submits the list of pages to the page space manager. As pages are retrieved from the dataset, the execute method processes each retrieved page to generate a region of the output image. When all the pages have been processed, the output is sent back to the client and is also stored in the data store manager.

4. EXPERIMENTAL RESULTS

We describe experimental performance results for the Virtual Microscope 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 employed three datasets, each of which was 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 used the driver program described in [3] to emu-

late 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 real experienced 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, the real user traces were not available. 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 executed simultaneously on a cluster of PCs connected to the SMP machine via 100Mbit Ethernet. Each client submitted its queries independent from the other clients, but waited for the completion of a query before submitting another one.

4.1 High vs. Low Overlap among Queries

We first examine the performance of the runtime system under two scenarios. The first scenario depicts the case where the overlap among queries is high, i.e., the overlap method returns a large index value on average. For this scenario, we emulated 16 concurrent clients. Each client generated a workload of 16 queries to the same dataset as the other clients. The average value of the overlap index was 0.70 for the queries (with a reasonable resource allocation in terms of memory and number of threads). The second scenario corresponds to the case in which queries are mostly disjoint, i.e., a relatively small overlap index value is computed by the overlap method on average. For these experiments, we used 8 concurrent clients, each generating a workload of 16 queries. Out of the 8 clients, 4 clients issued queries to the first dataset, 3 clients submitted queries to the second dataset, and 1 client issued queries to the third dataset. This configuration resulted in an average overlap index of 0.59. The output for each query was a 2048x2048 RGB image for both scenarios.

The timing values in the figures are the total execution time, including time for sending results back to clients, for all the queries from all the clients (i.e., a total of 256 queries with 16 clients and 128 queries with 8 clients) at the server. Up to 4 query threads were allowed to concurrently execute at the server. In order to also look at the performance impact of the Linux file cache, we obtained measurements for both cold and warm file caches[1]. In the figures, `off` denotes the case in which no intermediate data structures are stored in the data store manager. In the `on` case, the data store manager maintains intermediate data structures and queries can be evaluated using the intermediate results in the data store.

Figure 3 displays the overall execution time when the overlap among queries is high, which means intermediate results can be used by a large percentage of queries. As is seen from the figure, better performance is attained when the data store manager maintains intermediate results. The total execution time decreases by about 30% with a cold OS file cache, and by about 40% with a warm file cache. The overall execution time is smaller with a warm file cache than with a cold file cache as expected, since the I/O overhead decreases when data is accessed from the file cache. However, our results show that even when the file cache is warm, performance can be improved further if intermediate results are maintained. We observe that as the size of the memory allocated for the page space manager and the data store manager increases, execution time decreases. The average size of the query output (the intermediate results to be cached) in these experiments is 12MB. Thus, when the amount of memory allocated for the DS is relatively small (e.g., when 64MB is allocated for the DS, only 5 intermediate results can be maintained), it is highly likely that the intermediate result to answer an incoming query will not be in the DS, unless the working set of the multiple query workload is very small.

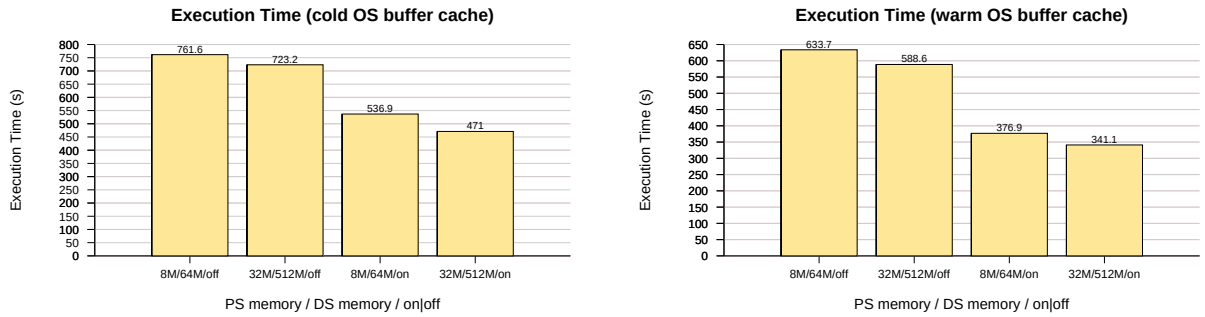
Figure 4 shows the total execution time for all the queries for the second scenario (i.e., 128 queries in total), when the value of the overlap index is relatively small. The results show that caching intermediate results in the data store manager introduces relatively little overhead, with a performance improvement of about 18% with a cold file cache and from 35% and 47% with a warm file cache achieved even when there is not much overlap among regions of interest for different queries. In this scenario, since the working set is larger, the Page Space Manager cannot hold a large percentage of all pages being used by the queries. However, the cost for accessing the pages from disk gets amortized better for the warm cache scenario, thus providing a larger performance benefit than for the cold cache.

4.2 Varying Number of Query Threads

In this set of experiments we look at the impact on performance of the maximum number of queries allowed to execute concurrently. The runtime system creates a thread pool that can be configured to set the maximum number of threads that can be simultaneously spawned. Since the runtime system provides larger performance improvements when there is a large number of overlapping regions of interest among queries, timings were measured with a value of overlap index of 0.70 (as in Figure 3 in Section 4.1). In the experiments, the sizes of the memory allocated for the data store manager and the page space manager were 512MB and 32MB, respectively. We present experimental results for both cold and warm Linux file caches.

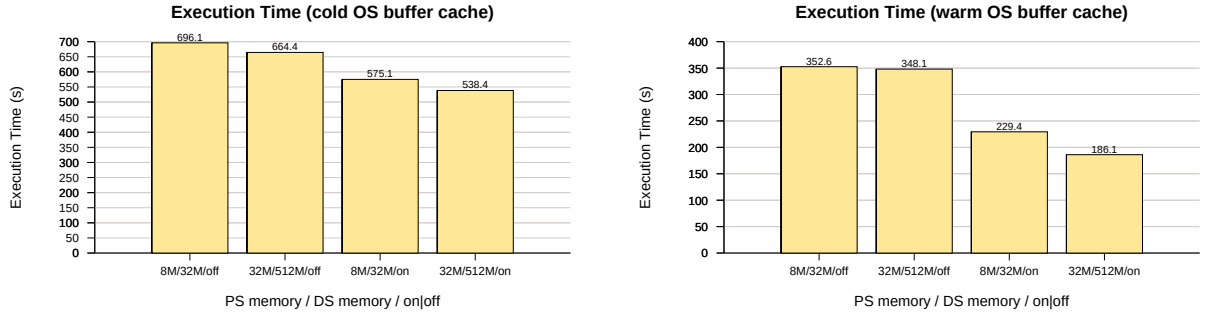
Figures 5(a) and (b) show the total query evaluation time for 256 queries and the breakdown of the evaluation time for a query, when the maximum number of query threads is varied from 2 to 16. As is seen from Figure 5(a), the query execution time decreases as the number of threads is increased. The execution time obtained for 16 threads is almost the same as 8 threads because the machine we used in the experiments has 8 processors. A similar trend in execution time across different number of threads is observed for both the cold and warm file caches. The decrease in execution time is not linear, because the SMP machine has only one disk and queries for the VM application are mostly I/O bound. As a result, as one can see from Figure 5(b), the I/O overhead dominates the execution time

[1]In order to clear the Linux file cache, we used a script that unmounted and re-mounted the partition, in which the datasets are stored. This strategy invalidates all the pages that may still be in the file cache, forcing I/O operations to access the disk when a page is later retrieved.



(a) (b)

Figure 3: The total execution time in seconds of all queries when the average overlap index value is 0.70. (a) The Linux file cache is erased before running the experiment, (b) the experiments start with a warm file cache. The size of the memory used by the Page Space manager (PS) and the Data Store manager (DS) is also varied. `off` denotes the case in which no intermediate data structures are stored in the data store manager. `on` represents the case in which the data store manager maintains intermediate results.



(a) (b)

Figure 4: The total execution time (in seconds) of all the queries as the size of the memory used by the page space manager and the data store manager varies. The average overlap index value is 0.59. (a) Cold file cache. (b) Warm file cache.

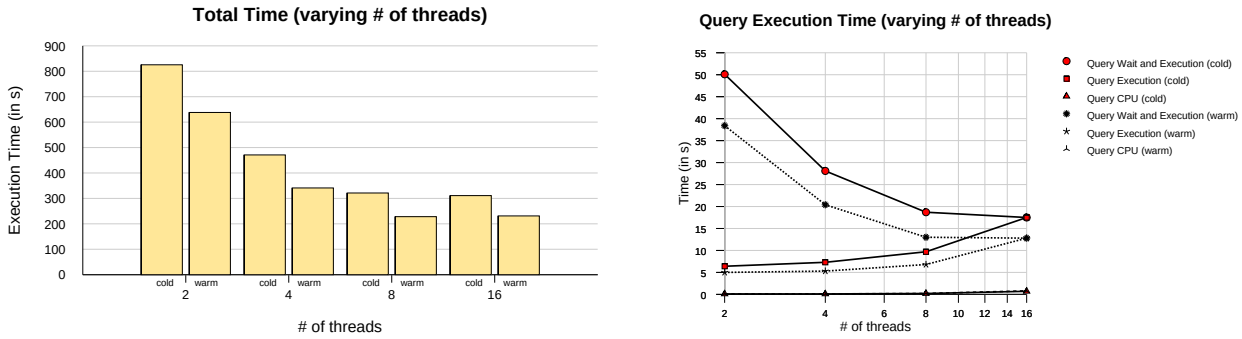
(denoted by *Query Execution* in the figure), and increases when more threads compete for the page space manager, hence for the disk. However, better performance is achieved as more threads are allowed to execute simultaneously, as a result of the decrease in the amount of time a query waits to be scheduled for execution. In the current implementation, queries are scheduled in FIFO order. A scheduling policy that takes into account the amount of overlap among queries and available resources can result in better performance.

Figures 6(a) and (b) display page space manager activity and the amount of overlap seen by an incoming query. The number of page requests rises as the number of threads is increased; this is because each thread issues page requests independently. The figure also shows that more pages are served from the page space manager cache when more threads execute in the system. Since the average value of overlap index is large in the experiments, when there are more concurrent threads in the system the intersection of the working sets of the threads is more likely to be in the page space manager cache. Figure 6(b) shows that the amount of overlap between a new query and the current set of intermediate results stored in the data store manager is not sensitive to the number of query threads, as it remains almost the same when the number of concurrent threads is varied.

4.3 Varying Size of Data Store

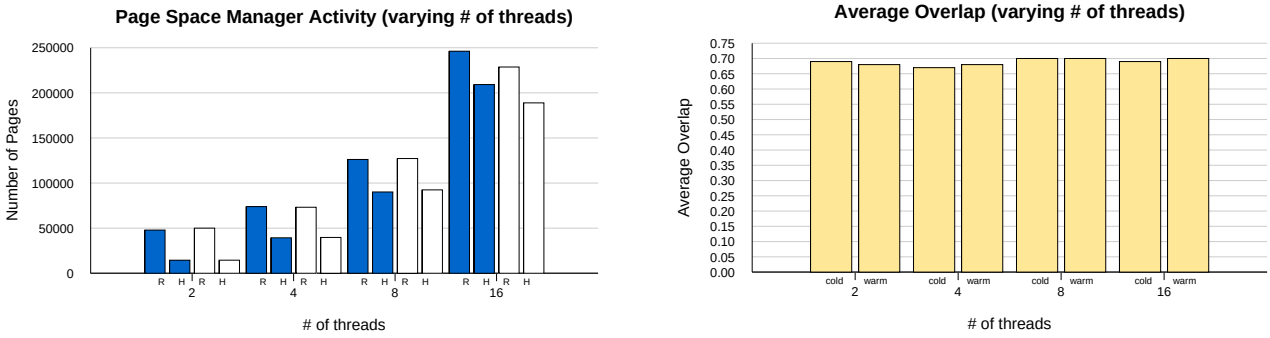
We now examine the performance implications of varying the size of the memory allocated for the data store manager. In this set of experiments, the maximum number of query threads was fixed at 4, and the size of the page space manager cache was 32MB.

Figures 7(a) and (b) show the total query evaluation time and the breakdown of the average evaluation time for a query into several components. As is seen from the figures, the total query evaluation time decreases as the size of the data store manager cache increases. The query execution time (denoted by Query Execution in Figure 7(b)) decreases by 12%, from 8.3 seconds for the 64MB data store cache to 7.3 seconds for the 512MB data store cache (with cold cache). The query waiting time also decreases by the slightly higher amount of 14%, since queries executing at the server finish faster. As more cache space is added to the data store, the average value of the overlap index increases, because more intermediate results can be maintained in memory (Figure 8(a)). As a result, the number of page requests



(a) (b)

Figure 5: (a) The total query evaluation time (in seconds), and (b) the breakdown of query evaluation time for cold (solid lines) and warm (dotted lines) file caches. *Query Wait and Execution* denotes the total time a query submitted to the server spends in waiting for execution plus executing at the server. *Query Execution* is the execution time of the query, which is the sum of computation time, I/O time, and communication time (to send the results back to the clients). *Query CPU* denotes the computation time of a query.



(a) (b)

Figure 6: (a) Page space manager activity. `R` is the number of page requests, and `H` is the number of hits to the pages in the page space manager. The dark and light colored bars show the results for the cold and warm file caches, respectively. (b) The average amount of overlap between the results in the data store manager and a new query submitted to the server.

to the page space manager decreases, resulting in less I/O overhead (Figure 8(b)).

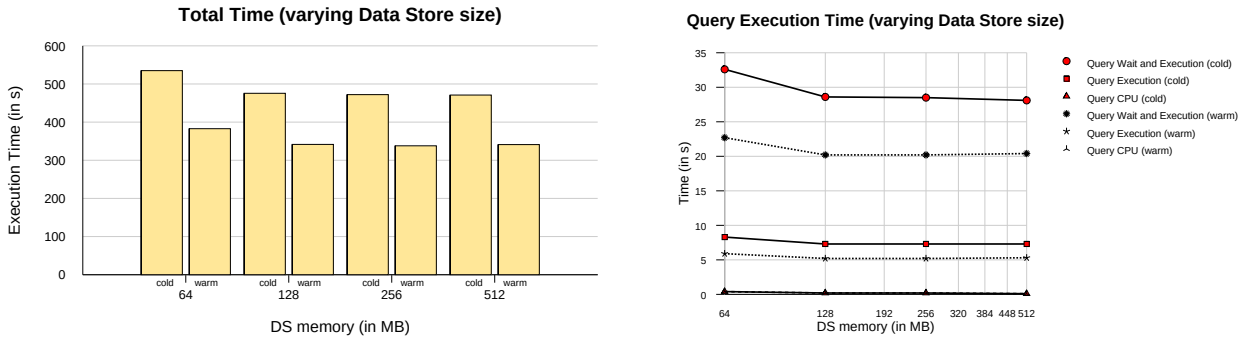
4.4 Varying Number of Queries

We also investigated the performance of the system when the number of queries is increased. In these experiments, we emulated 16 concurrent clients, each of which generated a workload of 100 queries to the same dataset. The output for each query was a 1024x1024 RGB image. The time values presented in the figures are the total execution time for all the queries from all the clients (i.e., a total of 1600 queries) at the server.

As is seen from Figure 9(a), performance improves when the data store manager maintains intermediate results, with the execution time of the 1600 queries decreasing by 38%. We also observe that as the size of the memory allocated for the page space manager and for the data store manager increases, the execution time decreases as expected. As is shown in Figure 9(b), when the size of the thread pool is increased from 2 to 4, the total query execution time drops by a factor of 1.95. The decrease in the execution time is less than linear when 4 more threads are added. This is because, as more threads are added, there is likely more contention to access the page space manager and the data store manager, as was also discussed in Section 4.2.

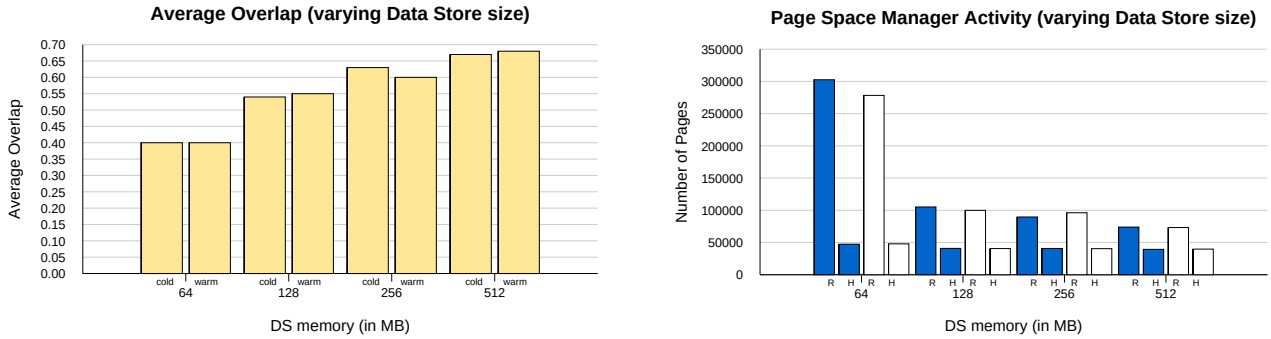
Figure 10(a) shows the average overlap index per query as the size of the data store memory is varied. As is seen from the figure, the overlap index increases as the size of the data store memory increases. Therefore, more intermediate results can be used to evaluate a query and more queries can be answered using intermediate results. A consequence of this is that queries will issue fewer page requests to the page space manager. This result is shown in Figure 10(b).

The performance improvement achieved in the experiments in this section is larger on average than that in the experiments in previous sections. When more queries are submitted to the server, the leverage that can be obtained by caching and sharing intermediate results among queries and by multi-threaded execution becomes even more discernible.



(a) (b)

Figure 7: (a) The total query evaluation time (in seconds), and (b) the breakdown of query evaluation time for cold (solid lines) and warm (dotted lines) file caches. *Query Wait and Execution* denotes the total time a query submitted to the server spends in waiting for execution plus executing at the server, *Query Execution* is the execution time of the query, which is the sum of computation time, disk I/O time, and communication time (to send the results back to the client). *Query CPU* denotes the computation time of a query.



(a) (b)

Figure 8: (a) The average amount of overlap between results in the data store manager and a query submitted to the server. (b) The page space manager activity. The amount of memory allocated for the data store manager is varied from 64MB to 512MB.

4.5 Effects of I/O and Communication Overheads on Scalability

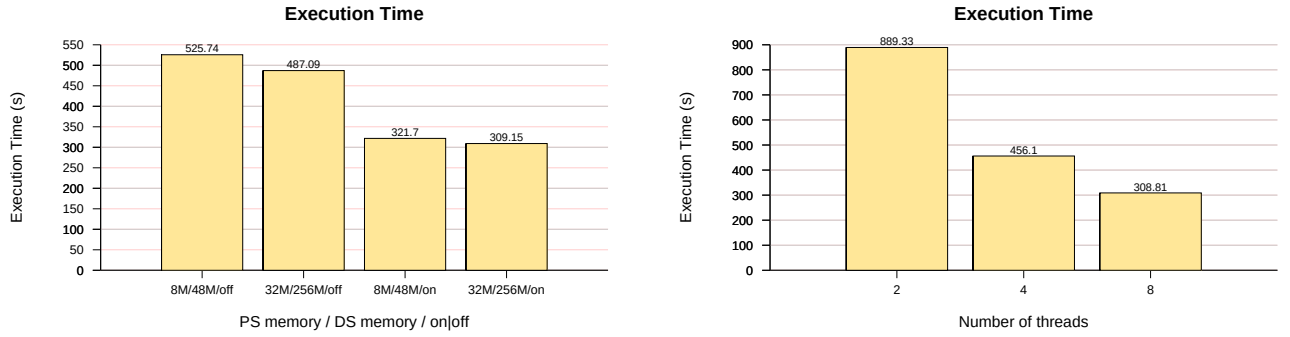
The performance results obtained in the experiments suffer from some non-scalability issues related to the network and storage subsystem. These issues account for the less than linear scalability seen in the experiments that depict the overall execution time.

First, although the datasets are partitioned and declustered across multiple files, all the files are stored on the same disk, since there is only one disk attached to the machine. When multiple queries are submitted to the server, we expect that there will be contention on the I/O subsystem when a thread retrieves, through the page space manager, the pages necessary to satisfy the query. As a result, the I/O cost is going to be higher, comparatively, dominating the cost of answering a query.

Second, our experimental setup is a complete client-server system. Queries are submitted and the results are shipped back to clients. In our prototype, the query result is usually 12MB in size (the size of the output may be less than 12MB if the query region happens to be near the edge of the input image). When there are 16 clients submitting queries at the same time, the server may face a demand for shipping up to 192 MB of data to the clients. There are two non-scalable components in the data path: the OS networking subsystem, and the physical interconnect, and both have bandwidth limitations. Linux is known to have a very scalable and fast networking subsystem, but there is still a large amount of data to move, and the Fast Ethernet interconnect we use also has a maximum of a few MB/s transfer rate. Therefore, the overall query execution time shows non-linear scaling when the number of query threads is increased, since communication cost is also included in the timings.

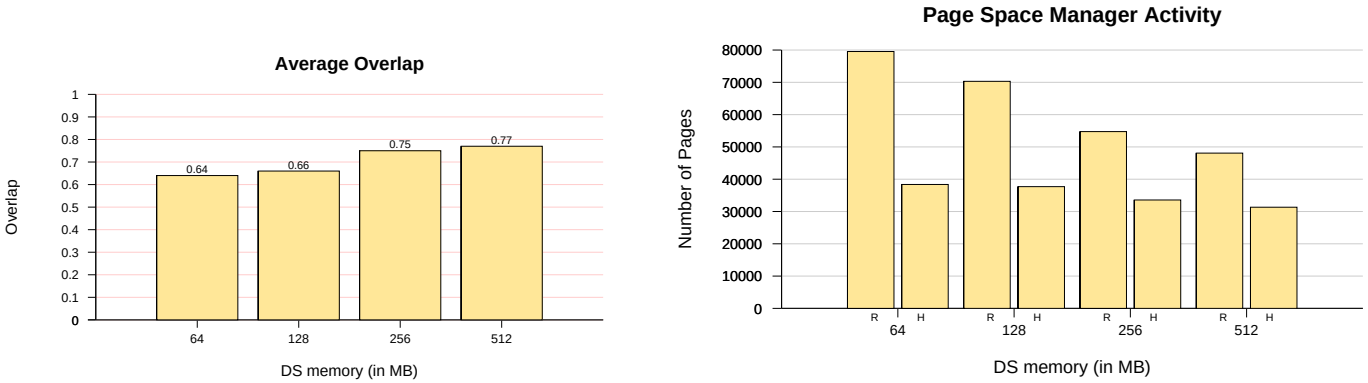
5. RELATED WORK

Optimizations for the execution of multiple queries have been extensively investigated in the context of relational databases [12, 13, 17, 18, 19, 21]. The proposed techniques include elimination of common sub-expressions, reordering of query plans, use of materialized views, and prefetching and caching of input data values. Once common subexpres-



(a) (b)

Figure 9: The total execution time (in seconds). (a) The size of the memory used by the page space manager and the data store manager varies. In the experiments, up to 8 query threads are allowed to simultaneously execute at the server. (b) The maximum number of concurrent threads is varied.



(a) (b)

Figure 10: (a) The average overlap index per query. (b)The page space manager activity as the amount of memory allocated for the data store manager is varied. `R` is the number of page requests to the page space manager, and `H` is the number of cache hits.

sions are detected for a batch of queries, individual query plans are merged or modified and queries are scheduled for execution so that the amount of data retrieved from the database is minimized and multiple executions of common paths in the query plans are avoided. The use of materialized views has also been investigated. Several views of subsets of database relations can be maintained so that queries can be evaluated using materialized views, to minimize the number of accesses to the database relations. The materialized views are the intermediate and final results computed during the execution of previous queries. Chakravarthy and Minker [5] present a mechanism that uses a hyper-graph representation of multiple queries. The hyper-graph representation enables modeling of a multi-query workload in a non-procedural way. The authors show how this representation can be used to identify common sub-expressions using heuristics. Sellis [19] discusses two possible alternatives for optimization: one alternative is based on local optimized sub-queries merged into a global access plan, and the other is based on the global optimization of multiple access plans for each of the sub-queries in order to obtain a globally optimal plan. The author describes two algorithmic approaches for these alternatives, Interleaved Execution and Heuristic Algorithm. These algorithms are quantitatively compared to Arbitrary Serial Execution, which is the non-optimized scenario. Shim et. al. [21] improve upon the Heuristic Algorithm. The authors show that the new proposed heuristic, albeit more expensive to compute, produces the best access plan much faster, i.e., expanding many fewer states. Tan and Lu [22] address an algorithmic approach for sharing intermediate results in pipelined query plans. They use a two-way approach, which first generates and then merges locally optimal plans. A sharing-graph is produced in the merge step. This graph is traversed in such a way that the next node to be selected from the graph benefits from the data already in memory. Various ideas previously explored for relational databases can be used for applications that analyze and visualize very large scientific datasets and for decision support applications. However, those techniques are difficult to directly apply to multiple query workloads in those applications because of the user-defined functions and complex data structures associated with queries.

Optimization for execution of queries and multi-query workloads also has been examined for data warehouses, parallel database systems, data mining applications, and data analysis in distributed environments. Dewhurst and Lavington [9] address query optimization by combining individual queries into larger queries for data mining applications. They assume a restricted query form, basically a relational select with a (ordered) count by group. They also define an operator, which, given two queries, produces the "simplest" query Q that includes (subsumes) both. Mehta et. al. [15] deal with the problem of scheduling multiple queries by partitioning them into batches. Their techniques identify common operations in batches of queries and schedule them in such a way that repetition is decreased. They consider sharing at a lower level, i.e., individual query operators (e.g., two selects with non-overlapping predicates). Yang et. al. [24] address the problem of which set of views should be materialized in order to improve query response time in a data warehouse, given a set of global queries and their access frequencies, and a set of base relations and their update frequencies. Chang et. al. [6, 7] present a runtime system and strategies for efficient query execution on distributed-memory parallel systems. The runtime system can execute queries concurrently on a distributed-memory parallel machine. However, each query is assigned its own workspace (e.g., memory for intermediate data structure). The runtime system switches between queries to issue I/O and communication operations, and handles the computation for a query when the corresponding I/O and communication operations complete. Beynon et. al. [4] address the use of group instances in a component-based framework for executing multiple query workloads in distributed environments. The application processing is represented as a group of interacting components, and queries are scheduled in a demand-driven fashion across multiple group instances. Our work differs from previous work in that we address efficient execution of multiple queries with user-defined functions in a framework that allows sharing of input data and query results among the queries.

Semantic caching methods to store query results have been investigated by Dar et. al. [8] and Godfrey and Gryz [10]. Those projects focus on caching at the client side so that the data movement between a client and a data server, hence the network load, is reduced. Nelson et. al. [16] present an architecture that is used for caching remote objects and that allows clients on a single machine to share a cache for accessing remote objects. Hellerstein and Naughton [11] and Voss [23] present methods for caching the results of user-defined functions in the context of object-relational and object-oriented databases. The goal of their work is to avoid duplicate computations of expensive methods by caching their results. Our work has similarities to these works in that data and query results are maintained in memory cache to speed up execution of queries. However, we have implemented the data store manager so that the result of a query can also be used by other queries to compute new results. Also, in this paper, the caching is implemented at the server side. This allows multiple queries from different clients to share and use query results. The data source abstraction provides a flexible mechanism such that by appropriate data source implementations, the runtime system can run at the server side or at the client side or as proxy between clients and the server.

6. CONCLUSIONS

We have presented a runtime infrastructure designed to provide efficient support for execution of multiple query workloads in data analysis applications on a shared-memory multiprocessor. The infrastructure provides a framework for application developers to implement application-specific processing on data and user-defined data structures for storing intermediate results. The system provides support for (1) maintaining and accessing user-defined intermediate data structures generated by queries, (2) caching of input data, and (3) multi-threaded execution. Our initial experiments using the Virtual Microscope application show promising results; significant performance improvements are achieved by using multiple threads for executing simultaneous queries and by making use of intermediate results.

We are currently examining query scheduling policies and query partitioning schemes. The FIFO query scheduling policy used in this work cannot make smarter decisions that would improve overall system throughput by issuing queries out-of-order. That can be beneficial when an incoming query can be (partially) answered with intermediate results in the Data Store before the results are possibly evicted. By reordering queries in a query batch, we can take advantage of the cached results that would otherwise be discarded. The query partitioning schemes will involve an architectural shift towards multi-tier hardware platforms (i.e., clusters of SMP machines), as opposed to the single SMP platform used in this paper. Such an architecture would allow the partitioning of the datasets and queries, and hence alleviate the non-scalability issues discussed in Section 4.5, by leveraging parallel I/O and parallel communication with clients.

We are also in the process of implementing a clustering algorithm for data mining. Our implementation decomposes the clustering algorithm into a set of operations (e.g., projection and selection of tuples that are to be clustered, distance matrix generation, and clustering computations). Our goal is to investigate the performance impact of organizing the processing structure of a data mining query into a pipeline of components. Such a decomposition may provide more opportunities for data reuse as each component will produce intermediate results. These intermediate results can be used by other queries, possibly increasing concurrency and resulting in pipelined processing of multiple queries.

7. REFERENCES

[1] A. Acharya, M. Uysal, R. Bennett, A. Mendelson, M. Beynon, J. K. Hollingsworth, J. Saltz, and A. Sussman. Tuning the performance of I/O intensive parallel applications. In *Proceedings of the Fourth Annual Workshop on I/O in Parallel and Distributed Systems (IOPADS)*. ACM Press, May 1996.

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

[3] M. Beynon, A. Sussman, and J. Saltz. Performance impact of proxies in data intensive client-server applications. In *Proceedings of the 1999 International Conference on Supercomputing*. ACM Press, June 1999.

[4] M. D. Beynon, T. Kurc, A. Sussman, and J. Saltz. Optimizing execution of component-based applications using group instances. In *Proceedings of CCGrid2001: IEEE International Symposium on Cluster Computing and*

the Grid, pages 56–63. IEEE Computer Society Press, May 2001.

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

[6] C. Chang, T. Kurc, A. Sussman, U. Catalyurek, and J. Saltz. A hypergraph-based workload partitioning strategy for parallel data aggregation. In *Proceedings of the Eleventh SIAM Conference on Parallel Processing for Scientific Computing*. SIAM, Mar. 2001.

[7] C. Chang, T. Kurc, A. Sussman, and J. Saltz. Optimizing retrieval and processing of multi-dimensional scientific datasets. In *Proceedings of the Third Merged IPPS/SPDP (14th International Parallel Processing Symposium & 11th Symposium on Parallel and Distributed Processing)*. IEEE Computer Society Press, May 2000. Also available as University of Maryland Technical Report CS-TR-4101 and UMIACS-TR-2000-03.

[8] S. Dar, M. J. Franklin, B. Jónsson, D. Srivastava, and M. Tan. Semantic data caching and replacement. In *Proceedings of 22nd International Conference on Very Large Data Bases*, Bombay, India, September 1996.

[9] N. Dewhurst and S. Lavington. Knowledge discovery from client-server databases. In *2nd International Conference on Principles of Knowledge Discovery in Databases*, 1998.

[10] P. Godfrey and J. Gryz. Answering queries by semantic caches. In *Proceedings of the 10th Database and Expert Systems Applications (DEXA'99)*, pages 485–498, Florence, Italy, August 1999.

[11] J. M. Hellerstein and J. F. Naughton. Query execution techniques for caching expensive methods. In *ACM SIGMOD International Conference on Management of Data*, ACM SIGMOD Record, Vol.25, No.2, pages 423–434, 1996.

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

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

[14] T. Kurc, C. Chang, R. Ferreira, A. Sussman, and J. Saltz. Querying very large multi-dimensional datasets in ADR. In *Proceedings of the 1999 ACM/IEEE SC99 Conference*. ACM Press, Nov. 1999.

[15] M. Mehta, V. Soloviev, and D. DeWitt. Batch scheduling in parallel database systems. In *Ninth International Conference on Data Engineering*, 1993.

[16] M. N. Nelson, G. Hamilton, and Y. A. Khalidi. A framework for caching in an object-oriented system. Technical Report SMLI TR-93-19, Sun Microsystem Laboratories, Inc., October 1993.

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

[18] P. Roy, S. Seshadri, S. Sudarshan, and S. Bhobe. Efficient and extensible algorithms for multi query optimization. In *SIGMOD Conference 2000*, pages 249–260, 2000.

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

[20] J. Shafer, R. Agrawal, and M. Mehta. SPRINT: A scalable parallel classifier for data mining. In *The 22nd VLDB Conference*, pages 544–555, Bombay, India, Sept 1996.

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

[22] K.-L. Tan and J. Lu. Workload scheduling for multiple query processing. *Information Processing Letters*, 55(5):251–257, 1995.

[23] W. E. Voss. *Caching Derived Data in Object-Oriented Databases, and An Intelligent System Design for Selecting Their Materialization Strategies*. UIUCDCS-R-98-2041, University of Illinois at Urbana-Champaign, Computer Science Department, May 1998.

[24] J. Yang, K. Karlapalem, and Q. Li. Algorithms for materialized view design in data warehousing environment. In *International Conference on Very Large Data Bases*, pages 136–145, 1997.