Efficient Execution of Multiple Query Workloads in Data Analysis Applications[*] (Extended Abstract)

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

† Institute for Advanced Computer Studies
and
Dept. of Computer Science
University of Maryland
College Park, MD 20742

+ Dept. of Pathology
Johns Hopkins Medical Institutions
Baltimore, MD 21287

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

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. The accuracy and value of the information provided by decision support systems and data-mining applications largely depend on the amount of data, so many businesses are inclined to gather and store as much data as possible.

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. Technological advances are providing cost-effective solutions for storing large datasets; it is possible to build a 1-Terabyte storage system using commodity disks and PCs at about $10,000. However, software support is needed to efficiently store large datasets, access the data of interest, and process 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

[*]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).

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.

Optimizations for the execution of multiple queries have been extensively investigated in the context of relational databases [7, 8, 10, 11, 12, 14]. Common optimization techniques include detection of common subexpressions in query plans, use of materialized views (or intermediate results), and data prefetching and caching. Once common subexpressions 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.

In this work we are designing 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 optimizations for scheduling system resources and operations, for data caching, for maintaining and using intermediate data results, and support for distributed and multi-threaded execution, when a multiple query workload is presented.

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, techniques and runtime support for relational databases are difficult to directly apply to multiple query workloads in those applications 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, called *accumulator*, is used 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 [13]. (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 operations and data mining operations.

In earlier work we developed runtime support and examined strategies for efficient execution of queries in data analysis applications on distributed-memory parallel machines with a disk farm [6, 9] and in distributed, heterogeneous environments [3, 5]. Our previous work has focused on the efficient execution of a single query. 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. 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.

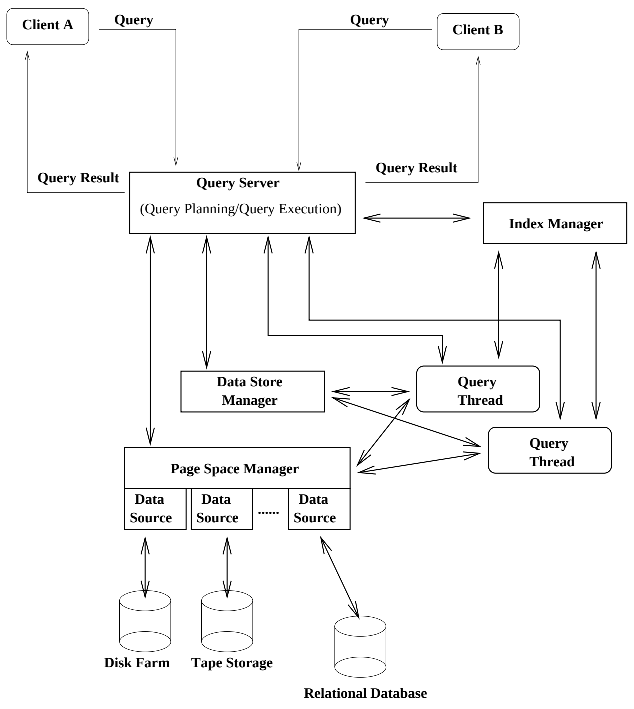


Figure 1: The system architecture.

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. The runtime system supports multi-threaded execution on a shared-memory multiprocessor.

2.1 Query Server

The query server 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 (e.g., 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 intersect the 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 type id*. The implementation of a new query object is done through C++ class inheritance and virtual methods. A *Query* base class is provided by the system for this purpose. A query object 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 and provides a virtual method, called `overlap`. A *QueryMI* base class 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. Both the query meta-data object and the accumulator meta-data object are implemented by the application developer by sub-classing the base classes provided by the system. Note that the final result for a query can also be viewed as an intermediate result for another query. Thus, the meta-data information for the output is also implemented using an accumulator meta-data object. 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 the processing of data retrieved from the dataset in the `execute` method. 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) is the only policy 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]. 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 a data source. 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. Currently, we have a data source implementation for the Unix file system.

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.

We should note that 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). More complex cache models are required to achieve good performance in these cases. In particular, cache blocks can be non-rectangular and variable sized. More complex index structures must be developed to efficiently find blocks in the cache. In addition, updates to the indices, when blocks are inserted or deleted from the cache, should be done efficiently to reduce cache lookup costs. In addition, 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.

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 *piggybacked* onto the first query in order to avoid traversing the same dataset twice.

2.4 Data Store Manager

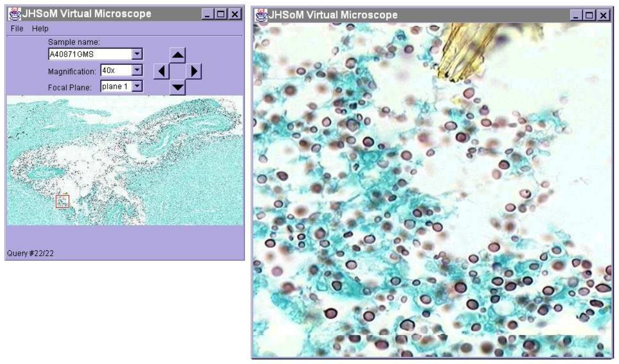
The data store manager (DS) is responsible for providing dynamic storage space for intermediate data structures generated as partial or final results for a query. Note that intermediate data structures present one type of commonality among multiple queries. That is, given an intermediate result X computed 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 records semantic information about intermediate data structures. This allows 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 a 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 also provides 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 operation of the `lookup` method should be customized for each intermediate data structure. This is achieved by implementing the `overlap` method provided by the accumulator meta-data object (see Section 2.1). 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 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.



Sample name: A40871GMS
Magnification: 40x
Focal Plane: plane 1
Query #22/22

Figure 2: The Virtual Microscope client.

3 Analysis of Microscopy Data: The Virtual Microscope

The Virtual Microscope (VM) [2] provides a realistic digital emulation of a high power light microscope. Figure 2 shows the client interface for VM. The raw data for such a system 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. At a basic level, the Virtual Microscope can emulate the use of a physical microscope, including continuously moving the stage and changing magnification.

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 subsampled to achieve the magnification level (subsampling 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 Virtual Microscope 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 implementation of the Virtual Microscope 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. A VM query retrieves a subregion of the image, specified by the query window, and subsamples the region to generate an image at the requested magnification. Therefore, 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

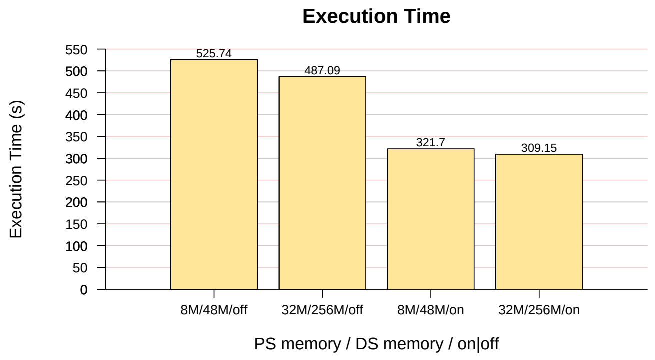


Figure 3: The total execution time (in seconds) of all the queries as the size of the memory used by the page space manager and the data store manager varies. `off` denotes the case in which no intermediate data structures are stored in the data store manager. `on` represents the case in which the data store manager maintains intermediate results. PS stands for the Page Space manager, and DS is the Data Store manager.

was implemented to intersect two regions and return an *overlap index*. The overlap index is computed as

$$overlap\ index = \frac{I_A}{O_A} \times \frac{I_S}{O_S} \tag{1}$$

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 subsampling factor used for generating the intermediate result, and O_S is the subsampling 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 subqueries 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

In this section we describe experimental performance results for the Virtual Microscope application implemented using the runtime infrastructure. The experiments were carried out 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. We have used the driver program described in [4] 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 real experienced users. Interesting regions are modeled as points in the slide, 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

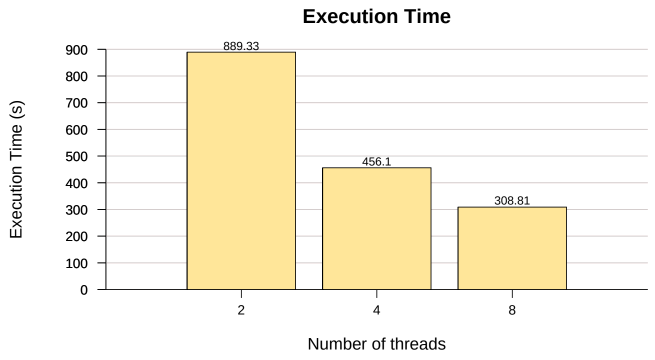


Figure 4: The total execution time (in seconds) as the maximum number of concurrent threads is varied.

same manner. The slide is swept through in either an up-down fashion or a left-right fashion as observed from real users. We emulated 16 concurrent clients executing on a cluster of PCs connected to the SMP machine via 100Mbit Ethernet. Each client generated a workload of 100 queries. The output for each query was a 1024x1024 RGB image. In the experiments we used a dataset of size 26000x26000 3-byte pixels. The image was partitioned into 64KB pages, each of which represents a square region in the entire image, and stored on the local disk attached to the SMP machine. 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.

In the first set of experiments, we look at the performance implications of using intermediate results for evaluating queries. In Figure 3, `off` denotes the case in which no intermediate data structures are stored in the data store manager. With `on`, the data store manager maintains intermediate data structures and queries can be evaluated using the intermediate results in the data store. In these experiments, up to 8 query threads are allowed to simultaneously execute at the server. As is seen from the figure, better performance is obtained 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.

The second set of experiments examines how query server performance changes as the maximum number of query threads that are allowed to execute concurrently is varied. The runtime system creates a thread pool that can be configured to set the maximum number of threads that can be simultaneously spawned. As is shown in Figure 4, 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.

Figure 5 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 6. The number of hits to the pages cached in the page space manager remains almost constant, since it depends on the amount of memory allotted to the page space manager. As the number of page requests decreases, fewer I/O requests to the storage system need to be generated, reducing the I/O overhead.

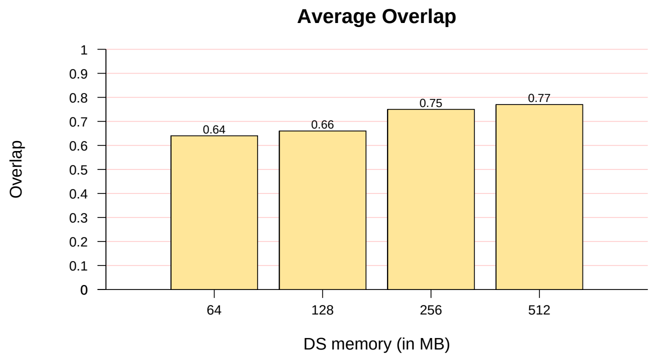


Figure 5: The average overlap index per query.

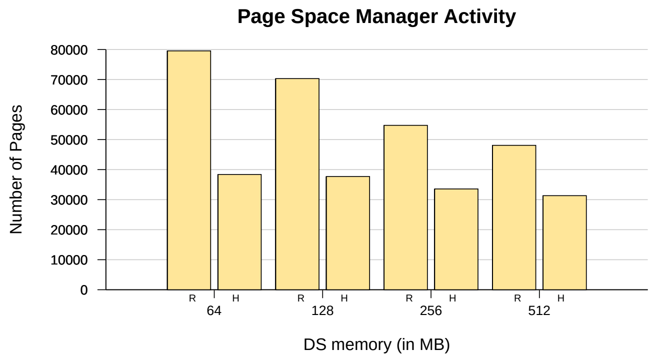


Figure 6: The page space manager activity as the amount of memory allocated for the data store manager is varied. R is the page requests to the page space manager, and H is the cache hits.

5 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 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.

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, T. Kurc, A. Sussman, and J. Saltz. Design of a framework for data-intensive wide-area applications. In *Proceedings of the 9th Heterogeneous Computing Workshop (HCW2000)*, pages 116–130. IEEE Computer Society Press, May 2000.

[4] 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.

[5] 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*. IEEE Computer Society Press, May 2001. To appear.

[6] C. Chang, R. Ferreira, A. Sussman, and J. Saltz. Infrastructure for building parallel database systems for multi-dimensional data. In *Proceedings of the Second Merged IPPS/SPDP Symposiums*. IEEE Computer Society Press, Apr. 1999.

[7] 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.

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

[9] 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.

[10] 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.

[11] 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.

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

[13] 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.

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