A Framework for Multiple Query Optimization on Multidimensional Data Analysis Applications

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
Ohio State University
Columbus, OH, 43210
{kurc.1,saltz.3}@osu.edu

Abstract

Efficient storage and analysis of large volumes of data are important issues in many fields of science, engineering, and business. In many cases, data analysis is employed in a collaborative environment where multiple clients access the same datasets and perform similar processing on the data. In this work, we present a multi-query optimization framework that permits the identification and the utilization of data and computation reuse opportunities in the presence of user-defined operators and aggregations for speeding up the execution of individual queries and query batches. We also present a methodology for functionally decomposing complex queries in terms of primitives so that multiple reuse sites are exposed to the query optimizer, to increase the amount of computation that can be reused. Furthermore, we show how query scheduling techniques, coupled with intelligent cache replacement policies, can further improve collective query processing. Finally, we give experimental results highlighting the performance improvements obtained by our methods using real scientific data analysis applications.

1 Introduction

"The purpose of computing is insight, not numbers" [27]. Therefore, the ultimate goal in the process of analyzing data, especially large datasets, is to gain insight into new phenomena that underlie these massive data collections in order to fulfill the role of computing as a tool for better understanding scientific and business realities.

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 in scientific studies. In environmental simulations, for

example, the output of a simulation is one or more datasets (in which each tuple or group of tuples are associated with spatial/temporal coordinates), each consisting of a large number of numerical values. A scientist often needs to access, explore, analyze, and visualize these datasets to gain insight into the problem at hand and to draw meaningful and useful conclusions about the validity and correctness of a theoretic model that underlies the simulation.

The number of applications that require analytical methods is steadily growing due to a confluence of factors. The first factor is simply the availability of adequate storage and computational resources. More important though is the inherent perception that these applications will shed new light upon business and scientific phenomena. In this context, *data analysis applications* can be defined as ones that access a subset of all the data available – the *hot spots* – which are the data points and regions of highest interest. The data of interest is usually processed and transformed into a data product. Data products are often generated by computing an aggregation over some of the dimensions of the dataset, ranging from simple associative arithmetic and logical operators to more complicated application-specific functions.

In many 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 the emerging field of medical imaging [10], a possible scenario is a large group of students wanting to simultaneously explore the same set of digitized microscopy slides or visualize the same Magnetic Resonance Imaging (MRI) and Computerized Tomography (CT) results. In these situations, the data server needs to execute multiple queries simultaneously to minimize latencies to the clients. In this 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.

This work investigates the problem of optimizing multiple data analysis queries for computation-intensive and/or data-intensive applications. Many different aspects of the multiple query optimization problem have been investigated in other contexts, particularly in relational databases. However, the scale of the datasets, the application-specific nature of data structures and the computation of user-defined aggregates require developing new optimization techniques to ensure good system performance, especially under heavy query workloads.

The need to develop new techniques and extend the methodologies proposed by other re-

searchers comes from the fact that data analysis applications have many aspects that are particular to them. Namely, current techniques suffer from limitations in handling either unstructured operators or operators that are not known *a priori*. In fact, most of the techniques assume a well-defined set of operators – relational database operators – and also assume specific algorithms implementing the operators. Such a restricted set of operators allows for defining efficient data structures for representing data and computation reuse scenarios. However, in more general data analysis applications there is not a pre-defined set of operators and algorithms that can be leveraged to construct an efficient query optimizer. Moreover, because of the exploratory nature of these applications, new operators and algorithms can be dynamically added to a data analysis application. Therefore it is necessary to provide ways to describe operators and the aggregates generated by such operators so that a generic query optimizer can detect optimization opportunities. The second important limitation of previously investigated techniques lies in the fact that data analysis application are often very data and computation-intensive. Thus, scalability and distributed processing issues must be accounted for from the outset so that good performance can be obtained by the optimization engine. In this context, there are four major contributions from this paper:

1. We propose a model for detecting application-specific reusable aggregates and a mechanism for automatically reusing data and computation through an *active semantic cache*. The model consists of a data transformation framework with a set of customizable operators that exposes information about how a particular (intermediate or final) data product was generated and how it might relate to other data products (i.e., how it can be transformed in case of partial reuse opportunities).

2. We show how the generic operator types from our model can be used to derive better query scheduling and cache replacements policies to further improve query optimizer performance.

3. We explore a model for functionally decomposing complex computations that can be used to described sophisticated queries in terms of simpler primitives. Query decomposition makes more reuse sites available to the query optimizer.

4. Using real scientific data analysis applications, we experimentally analyze the impact of each of our optimization techniques by employing a full-fledged server implemented as part of this

work. Our results show that 1) the active semantic cache model is very effective in reducing query response time under multi-query loads, 2) the cache replacement and query scheduling policies tailored to use the active semantic cache can further improve the performance of the server, and 3) functional decomposition is an effective approach for exposing data and functional reuse opportunities in data intensive applications.

The rest of this paper is organized as follows. Section 2 discusses how our work relates to previous research. Section 3 describes the data processing model we target and the optimization framework we employ. Section 4 describes the active semantic cache infrastructure, a new cache replacement policy, and multiple *informed* query scheduling policies. Section 5 describes how and why complex queries can be broken into simpler primitives. Section 6 describes two of our case study applications and how they are implemented using our data analysis framework Section 7 evaluates all of our optimization techniques and provides evidence showing how each particular technique improves performance. And, finally, Section 8 summarizes this work and presents a discussion of lessons learned.

2 Related Work

Multiple query optimization can be defined as a set of techniques aimed at minimizing the total cost of processing a set of queries by creating an optimized access plan for the entire set of queries [42]. Optimizations for the execution of multiple simultaneous queries have been extensively investigated in the context of relational databases [19, 24, 30, 41, 44]. In general, 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. One critical aspect in optimizing multi-query batches lies in identifying data and computation reuse opportunities. Indeed, it has been shown that exploiting reuse is a powerful mechanism for improving the performance of computational systems in general [11]. For relational database systems, it has been shown that identifying common subexpressions [19, 24, 42] and applying view materialization strategies [23, 34] can yield sizable decreases in query execution time when processing

multiple query batches. However, when applications do not conform to the relational database model and where the developer can extend the database by adding application-specific processing capabilities and operators, the optimization techniques developed for relational databases cannot be applied directly.

One way of leveraging results computed in the past in order to speed up the evaluation of queries under execution is through the use of a semantic cache. Semantic caching is essentially a mechanism for storing data products and their corresponding semantic descriptions and has been extensively studied by other researchers [25, 38].

The amount of reuse that the query optimizer can exploit depends on how many reuse sites are discovered. We therefore suggest, in Section 5, an approach for segmenting complex queries into primitives that is similar to the component-based application development paradigm. Several research projects have investigated the design and implementation of component-based frameworks for application development and deployment [2, 13, 16, 18, 37, 39]. In these frameworks, the application processing structure is decomposed into a set of interacting computational components. Most efforts on component-based frameworks have focused on improving the performance of one or more independent queries by effectively decomposing the application structure and efficiently scheduling application components. Our work differs in that we also examine the application of functional decomposition for increasing data and computation reuse opportunities.

Another area related to this work is query scheduling. In a more general setting, the query optimization and scheduling problem has been extensively investigated in past surveys [22]. It has been suggested that carefully scheduling queries plays an important role in terms of better using computational and I/O resources and also in terms of the amount of data reuse. Gupta et al. [26] 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 (non-relational datasets). Sinha and Chase [43] present some heuristics to minimize the flow time of queries and to exploit inter-query locality for large distributed systems. The work of Mehta et al. [32] is one of the first to tackle the problem of scheduling queries in a parallel database by considering batches of queries, as is done in this work, as opposed to scheduling one query at a time.

3 Optimization Model

In this work, we refer to an instance of a data analysis operation as a *query* that processes input data via user-defined operations, generates intermediate aggregates, and produces an output dataset. Frequently, the query result is a data product or an aggregate constructed by applying computational operations to transform the raw data into an artifact that can be used to identify the underlying phenomena, frequently through visualization techniques. An interesting fact is that many of these applications share a common processing model that we will describe shortly. Understanding this processing model is important because we construct our optimization framework around it, essentially by identifying where intermediate and final aggregates are generated and by providing a means for reusing them in subsequent query evaluations.

3.1 Generic Data Processing Model

Queries for many data analysis applications [1, 15, 21, 31] can be described according to a common processing structure [20]. In fact, the algorithm shown in Figure 1 describes abstractly the steps for processing a generic data analysis query. We refer to these queries as **range-aggregation queries** (RAG) since they typically have both spatial and temporal predicates that are defined as ranges, i.e., a multi-dimensional bounding box in the underlying multidimensional attribute space of the dataset (the *query domain*). Only data elements whose associated coordinates fall within the multidimensional box are retrieved and processed according to an application-specific and usually user-defined method.

In the processing loop, *Datasets* are the data and associated meta-data information available to the system for processing a query. The datasets can be classified as *input*, *output*, or *temporary*. **Input** datasets correspond to the data to be processed. **Output** datasets are the result of applying an operation to the input dataset. **Temporary** datasets (temporaries) are created during query processing to store intermediate results. Often a user-defined data structure is used to describe and store a temporary dataset. Temporary and output datasets are tagged with the operation employed to compute them and also with the query meta-data information. Temporaries are also referred to as *aggregates*, and we use the two terms interchangeably.

The function $Select$ identifies the set of data elements in a dataset that intersect the query

```
// Datasets
I ← Input
O ← Output
T ← Temporary Aggregate
M_i ← Query meta-data information
1. S_I ← Select(I, M_i)
   // Initialization
2. foreach t_e in T do
3.    t_e ← Initialize()
   // Processing
4. foreach i_e in S_I do
5.    read i_e
6.    S_T ← Map(i_e)
7.    T ← Operation(T, S_T)
   // Finalization
8. foreach t_e in T do
9.    o_e ← Output(t_e)
```

Figure 1: Abstract query processing code.

meta-data M_i for a query q_i. The data elements retrieved from the storage system are mapped to the corresponding temporary dataset items (step 6), and an application-specific operation (e.g., sum over selected tuples in a relational database or an image processing operation) is applied on the input data elements (step 7). To complete the processing, the intermediate results in T are post-processed to generate final output values. *Map* is an application-specific function that finds the set of output data items to which the input data elements contribute. An input element may map to a set of output elements, for example, due to irregular data distributions. *Operation* describes an application-specific data transformation function that, given the input data I, produces a data product from I. An instance of an operation uses the meta-data information to generate output data from the relevant parts (domain) of I.

A *query type* is the definition of a processing chain in which a collection of input datasets $I_1 \ldots I_n$ (collectively called I), the necessary meta-data information M that describes the data of interest, a temporary dataset T, and an output dataset O are specified. The meta-data information M defines the domain (e.g., relational predicate, or bounding box for multi-dimensional range-aggregation queries) and the functions *Operation* and *Map* to be applied to the input data I to

generate the output O.

3.2 Data Transformation Model

When multiple queries are processed in the context of a data analysis application, we expect that there will be inter- and intra-query commonalities because queries tend to concentrate on regions of higher interest in the dataset. Examples are the visualization of an interesting feature by multiple users inspecting an MRI data product and the execution of slightly different hydrodynamic simulation models over the same spatio-temporal grid.

In this description, query q_j refers to a query whose results (referred to as aggregate $\mathcal{J}$) are already computed and available for reuse, and query q_i refers to a query that has been scheduled for execution and whose results (referred to as aggregate $\mathcal{I}$) must be computed. To simplify the description of the model, we assume that each query generates a single output aggregate, but that simplification is not a limitation of the model.

We describe a mechanism for exploiting commonalities in the form of a data and computation reuse model. The model assumes that the two queries q_i and q_j, described respectively by metadata M_i and M_j, can each potentially share input elements (i_e, step 5 in the query processing loop depicted in Figure 1), temporary aggregate elements (t_e, step 7), and output elements (o_e, step 9). When multiple query types must be processed, cached data elements (i_e, t_e, and o_e) may not be able to be directly reused. However, if the framework provides the ability to perform a data transformation operation – converting one aggregate into another – data elements potentially can be reused. We refer to this type of data caching as *active semantic caching*.

The data transformation model is the cornerstone of our active semantic caching framework. First, it is responsible for the identification of reuse possibilities when a new query is to be executed. The query semantic information (i.e., metadata information) is used to find the matching entries in the data cache. Second, by transforming an aggregate cached in the system to another aggregate, data reuse opportunities increase. In our framework, information about multiple possible overlaps and multiple scheduling possibilities can be used to drive better cache management policies, as will be shown in Section 4.1, and also to process a query batch to better use available system resources, as will be shown in Section 4.2.

Conceptually, the following set of abstract operators describes the information the framework uses to identify and exploit redundancy elimination opportunities at runtime:

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

$$\mathbf{overlap_{project}}(M_i, M_j) = k, \text{where } 0 \leq k \leq 1 \tag{2}$$

$$\mathbf{project}(M_i, M_j, \mathcal{J}) = \mathcal{K}_{M_k}, \text{where } M_k \subseteq M_i \text{ and } \mathcal{K} \subseteq \mathcal{I} \tag{3}$$

Equation 1 describes the $compare$ function that returns *true* or *false* depending on whether aggregate $\mathcal{I}$ described by meta-data M_i is the same as $\mathcal{J}$ described by M_j. When the application of this function returns *true*, the query planner has identified a common subexpression elimination opportunity. Hence, in a query plan where $\mathcal{I}$ is supposed to be computed, the query executor can replace uses of $\mathcal{I}$ with a reference to aggregate $\mathcal{J}$.

In many situations, aggregates $\mathcal{I}$ and $\mathcal{J}$ partially overlap, which means that $compare$ is false, but partial reuse is still possible. Equation 2 describes the $overlap$ function that returns a value (overlap index) between 0 and 1 that represents the amount of overlap between aggregates $\mathcal{I}$ and $\mathcal{J}$. This function is computed by inspecting the domain of a cached aggregate (described by M_j) and the domain of the query under processing (M_i) in a two-step process. First, the Map function is applied to the domain of the cached aggregate and the amount of multidimensional overlap with the query domain is computed. Second, another factor of the overlap is computed with respect to a set of data transformation functions (called $project$ functions), which are responsible for identifying each *relevant* element from the cached aggregate and *converting* it (or a collection of them) into an output data element for the aggregate being computed. More precisely, the $project$ function, shown in Equation 3, takes one data aggregate $\mathcal{J}$ whose meta-data is M_j and *projects* it over M_i by extracting and transforming the parts of $\mathcal{J}$ that are relevant to $\mathcal{I}$, either computing it in its entirety or generating $\mathcal{K}$, which is a partial computation of $\mathcal{I}$. Each projection function manipulates the input aggregate in different ways to convert it into the aggregate required by the query being computed. Although any *ad hoc* data manipulation can be supported, we have identified a collection of pre-defined projection primitive classes that will be described in detail in Section 3.3.

Upon identifying a reusable aggregate, the query executor must complete the computation of

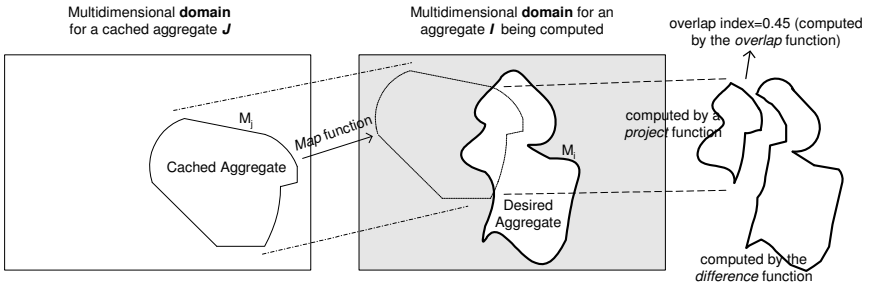


Figure 2: Using the *Map*, *Overlap*, *Project*, and *Difference* operators to extract a reusable part from a cached aggregate.

the desired aggregate by evaluating the portions that need to be computed from input data or from other aggregates. The following equation defines an operator that computes the set of subquery descriptors for the incomplete regions:

$$\mathbf{difference}(M_i, M_j) = C \tag{4}$$

where C is the set of subquery descriptors defined to satisfy the following conditions:

$$(\bigcup_{M_k \in C} M_k) \cup M_j \equiv M_i \tag{5}$$

$$\forall \text{ meta-data pairs } M_x, M_y \text{ where } M_x \in C, M_y \in C, \text{ and } M_x \neq M_y, M_x \cap M_y = \emptyset \tag{6}$$

After the relevant parts of $\mathcal{J}$ are obtained, Equation 4 can be employed to instantiate the subqueries for computing the aggregates that are necessary to finalize the processing for generating $\mathcal{I}$. The conditions stated in Equations 5 and 6 establish that these descriptors are the *necessary* and *minimal* ones required to complete the processing for $\mathcal{I}$ once $\mathcal{J}$ has been reused. The diagram in Figure 2 shows all the operators in action.

In order to make our description less abstract, we present an example of how this set of functions can be employed for query optimization. Given an image visualization application, assume there is a query that is supposed to return an image within a given bounding box, with a specific magnification level, and processed according to a particular image processing algorithm. Because of the amount of data to be processed, the input data and the output image may be tiled for parallel processing. Each temporary aggregate tile t_e and each output tile o_e can be described by a 3-tuple,

[bounding box, zoom factor, image processing algorithm], which constitutes the tile meta-data M. When a query must be processed and the algorithm in Figure 1 is executed, the operations in steps 5, 7, and 9 are not immediately performed. Instead, using the customized set of operators for the application at hand, a cache lookup is performed, trying first to find a complete match (applying Equation 1), and, if that fails, a partial match is attempted (applying Equation 2), in which case a projection primitive may be employed (Equation 3). Finally, subqueries describing the incomplete areas of the query region are generated using Equation 4, which are then automatically submitted for processing.

3.3 Reuse Primitives for Aggregation Operations

Conventional data caching approaches require a complete and perfect match (i.e., cache hit or miss) between the output to be computed and a previously computed aggregate. Equation 3 introduced the notion of *projection primitives* that enable *transforming* an aggregate generated by a query so that it can be reused to completely or partially satisfy a new query. We call the mechanism that allows such reuse operations *active semantic caching*. This mechanism gives the query engine a better chance to exploit reuse than does conventional caching.

Based on a number of applications [1, 12, 15, 31], we have identified four classes of projection primitives. They are based on the type of reuse they can leverage: dimensional overlap, composable reduction operations, invertible functions, and inductive functions.

3.3.1 Dimensional (Spatio-temporal) Overlap Primitives

In applications that deal with range queries, one common part of the query meta-data describes the spatial and/or temporal coordinates of the region over which the query will perform a computation. A dimensional projection primitive performs a geometric translation and/or rotation on the cached overlapped data, in addition to clipping the *n*-dimensional region to conform to the predicate for the new query. The new aggregate can be directly obtained if the cached aggregate completely subsumes the one being computed. Otherwise, the new aggregate can be partially generated, in which case the region that cannot be computed from the cached aggregate must be calculated from input data or from other cached aggregates using the difference operator (Equation 4). Indeed,

the difference operator returns C as the minimal set of multidimensional regions obtained when M_j, the cached aggregate dimensional coordinates, is subtracted from M_i, the coordinates of the aggregate to be computed.

3.3.2 Composable Reduction Operations Primitives

Aggregation operations that implement *generalized reductions* [28] are commutative and associative. A commutative and associative aggregation operation produces the same output value irrespective of the order in which the input data elements are processed. That is, the set of input data elements can be divided into subsets. Temporary datasets can be computed for each subset and a new intermediate result or the output can be generated by combining the temporary datasets. Interestingly, a composable reduction operation primitive takes one or more temporary aggregates with a finer-level aggregation[1] and transforms them into a coarser-level aggregation, coalescing multiple data points into a single new data point. However, in order for this primitive to generate a correct result, a *congruence relationship* must exist between the cached aggregate and the one to be computed.

We define a congruence relationship as follows. Suppose for a set of data elements $t_1, t_2, \ldots, t_n$, a collection of aggregates $T_{l,1}, T_{l,2}, \ldots, T_{l,m}$ is generated using a reduction operation f such that $T_{l,j} = f(t_o, \ldots, t_p)$ and let $S_{l,j} = \{t_o, \ldots, t_p\}$, where f is a generalized reduction operation, l designates the aggregation level, j designates a particular aggregate, and $S_{l,j}$ designates the set of data elementss used for computing $T_{l,j}$. Then $T_{x,i}$ – the aggregate to be computed, $\mathcal{I}$ – can be defined by a projection primitive over $T_{l,j}$ – the cached aggregate $\mathcal{J}$ – iff x is congruent to l, which means that the following condition must hold:

$$S_{l,j} \subset S_{x,i} \tag{7}$$

Here, x is an aggregation level that is coarser than l. The difference operator is then defined as follows:

[1]The aggregation level is associated with the amount of input data that needs to be aggregated. In image visualization, it can be the amount of zooming applied to an image. In satellite data processing, it can be the amount of daily data to be used for calculating a given data product.

$$C = \bigcup_{k \neq j \wedge S_{l,k} \subset S_{x,i}} S_{l,k} \tag{8}$$

3.3.3 Inductive Aggregation Primitives

Several aggregation operations can be described inductively, i.e., $f(n+1) = f(n)\ op\ g$, where g is a generic function, and *op* is an aggregation operation. In some cases, the function can be written as $f(n) = f(n+1)\ op^{-1}\ g$, meaning that $f(n)$ can also be computed from $f(n+1)$ by employing the inverse operation of *op*. Here, n designates the inductive step or, in some cases, how much precision one desires for a computation.

An inductive aggregation primitive can use a cached aggregate from a later inductive step to compute an aggregate for a prior inductive step by removing the contributions of the intermediate inductive steps, computed using the set C generated by the difference operator. Consider computing aggregate $\mathcal{I}$ defined by $f(n)$. Aggregate $\mathcal{J}$ is cached and defined as $f(n+1)$. $\mathcal{I}$ can be computed from $\mathcal{J}$ by computing $f(n+1)\ op^{-1}\ g$. Conversely, if $\mathcal{I}$ is cached we compute $f(n)\ op\ g$ to obtain $\mathcal{J}$. In this case the set C computed by the **difference** operator determines the additional computation and data required to compute $\mathcal{J}$.

An example of inductive operations is performing 3-dimensional volume construction from a set of 2-dimensional images [15]. In order to produce a 3-dimensional volume, an octree data structure is constructed from a set of 2-dimensional images. An octree of depth $n+1$ can be used to build an octree of depth n without retrieving and processing the input images. The primitive in this case benefits from the fact that building $f(n)$ from $f(n+1)$ and g is cheaper than generating it from the input dataset.

3.3.4 Invertible Aggregation Primitives

Many aggregations are computed by applying functions to a single input data element or to a collection of input data elements. Some of these functions may be invertible, more specifically they can be algebraically or procedurally reversed (given the result, the input can be obtained). Let us assume that aggregate $\mathcal{J}$ was computed by applying function f to a collection of data elements T. In order to compute aggregate $\mathcal{I}$ by applying function g, using the same data elements in

T, we must recover the original data elements by using f^{-1} to compute the original set T and, subsequently, apply g. For example, in satellite data processing, atmospheric correction is often employed to account for atmospheric effects on remotely sensed data and is performed by applying a function, for example, of the form $f(x) = c_1 + \frac{c_2}{c_3 \times (1 - c_4 \times x)}$, where x is the uncorrected value for a sensor measurement and c_1, c_2, c_3, and c_4 are physical constants [29]. In this case, the uncorrected value of x can be recovered by algebraically computing f^{-1}.

Usually, for invertible functions, leveraging reuse requires extra computation to re-generate the input data elements. Therefore, using this primitive requires trading off the cost of retrieving the input data against the cost of applying the invertible function to compute the input data element from a cached aggregate.

3.3.5 Composition of Multiple Projection Primitives

Oftentimes, combinations of projection primitives can be used to transform a cached aggregate for reuse by another query. For instance, a dimensional projection primitive can be followed by a primitive for either composable reduction operations or inductive aggregation functions to produce the desired aggregate. The query plan will determine how cached aggregates are going to be manipulated according to a set of transformations in order to obtain the data product defined by a query as will be seen in Section 5.

3.4 Overlap Functions

Related to each projection primitive is the issue of computing the amount of overlap between a cached aggregate and the aggregate sought by a query plan, with respect to a particular projection primitive (Equation 2). There is a one-to-one correspondence between each projection primitive and an overlap function. In general, the overlap function returns an index between 0 (no overlap) and 1 (full overlap). When a query plan is computed, the overlap function is used to rank cached aggregates in terms of how much they can help in computing the desired aggregate.

For dimensional projections, the overlap function computes a normalized value that measures how much of the desired aggregate can be computed from the cached one. This is accomplished by calculating the geometric overlap using two bounding boxes in the multi-dimensional space of

the dataset – one for the cached aggregate and the second describing the aggregate to be computed.

For composable reduction projections, the overlap function returns the *congruence level*, which is defined as the percentage of overlap between the aggregation level of a cached aggregate and the aggregation level to be computed based on meta-data associated with both aggregates (e.g., an aggregate for days 1 and 2 has a 0.5 overlap with an aggregate for days 1, 2, 3 and 4 from the same year).

For inductive functions, the overlap function computes an index in terms of *inductive distance*. The distance is normalized based on the inductive step being searched for (e.g., an image with a resolution of 2 Km^2 per pixel has an overlap of 0.5 with the same image at 4 Km^2 per pixel resolution). Finally, for invertible functions, the overlap function returns whether or not the function inverse can be computed (either 0 or 1).

4 Active Semantic Caching

The data reuse framework described in Section 3 employs caching and supports transformations on cached aggregates. An advantage of this scheme, in contrast to the way multi-query optimization is performed in relational database systems, is that a particular query execution plan may be optimized using results generated earlier. That is, some of the strategies previously explored optimize queries only within a single query batch, which presumes that a group of queries can be formed and simultaneously processed with a single, optimized query plan.

To demonstrate our optimization framework, we have built an active semantic cache system component. It is primarily responsible for providing dynamic storage space for intermediate aggregates generated as partial or final results for a query. It is said to be *active* because aggregates may be automatically transformed, and *semantic* because meta-data is usually stored with the aggregates to permit transformations to be applied. In our implementation, the system manages a fixed amount of in-core memory and secondary storage for holding cached aggregates.

A query can take advantage of the semantic cache during query planning. In particular, two operations are supported by the system – memory allocation and aggregate lookup. When a query must allocate space for an aggregate, the size of the aggregate and the corresponding meta-data are passed as parameters to an *alloc* method that handles all the bookkeeping in the resource

allocation phase [3]. This design ensures that all dynamically allocated memory is accounted for by the semantic cache to avoid paging activity. On the other hand, lookup operations are used to determine whether an aggregate to be generated can be computed entirely or partially from the aggregates stored in the cache. The lookup operation is defined in terms of both the **compare** and the **overlap** operators described in Section 3 in order to compute the amount of overlap between the sought aggregate and the cached ones, returning a pointer to the best match. A detailed description of the system architecture and a discussion of how applications are supposed to customize the abstract version of the active semantic cache operators can be found in [5, 6, 8].

4.1 Cache Replacement

Our multiple query optimization framework is application independent. The data server that implements the optimization framework was conceived with the design decision that multiple applications with multiple query types can be supported simultaneously. This decision takes root in the same principle that governs how Database Management Systems (DBMSs) are constructed, i.e., multiple applications making use of common infrastructure. Under this assumption, it is reasonable to assume that the aggregates computed by queries may have different construction costs. Hence, the use of simple cache replacement policies (such as least recently used) can potentially degrade overall system performance, since a misguided eviction can discard a very expensive aggregate when others that are cheaper to compute could have been discarded instead. Many cache replacement policies have been proposed in the past, and our optimization model supports several of them (e.g., Least Recently Used, Least Frequently Used, Size). However, none of these policies benefits directly from information available in the data transformation model. Therefore we have adopted a replacement policy used in Web caching – Least Relative Value (LRV) [17]. This policy replaces the aggregate that has the least *value*, which can be computed in several different ways. Ideally, *value* should be a relative measure of how expensive it is to generate a specific aggregate. To capture this metric, we extend the data transformation model with two new functions:

$$\mathbf{outputsize}(M_i) = c_1, \tag{9}$$

where c_1 is the query-specific estimated size in bytes of the result aggregate for query q_i over

the domain defined by its meta-data M_i, and

$$\mathbf{inputsize}(M_i) = c_2, \tag{10}$$

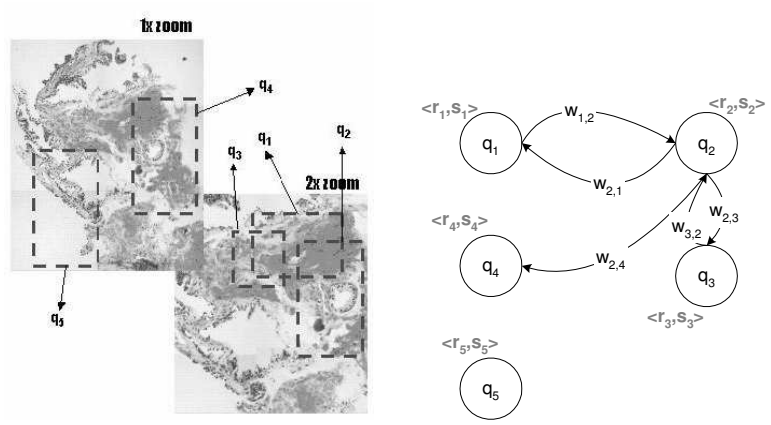
where c_2 is the estimated size in bytes of the amount of input data to be processed by query q_i over the domain defined by its meta-data M_i.

Using these two operators, we have devised two variants for calculating the value of an aggregate. The first variant uses the ratio $\frac{inputsize(M_i)}{outputsize(M_i)}$. We refer to this method as LRVA. The second variant is the ratio $\frac{q_{ttc}(M_i)}{outputsize(M_i)}$, We refer to this method as LRVB. The first variant, LRVA, is most suitable for I/O-intensive queries, for which most of the execution time is spent retrieving input data from disk. The second variant, LRVB, captures the complete cost of processing a query (q_{ttc}), including I/O, compute time, and time spent on bookkeeping tasks. We also employ *aging* as a means to alleviate the problem of retaining expensive aggregates indefinitely [4]. An exponential decay function is used to decrease the value of an aggregate over time.

4.2 Query Scheduling

In general, cache replacement policies attempt to discard the most *irrelevant* aggregate when an eviction decision is made. Complementary to this decision is scheduling a query to achieve maximum reuse based on what the cache currently holds and the queries currently being executed. Our implementation of the active semantic cache based server middleware can employ several different parallel organizations [5, 6, 8] (e.g., multithreading, cluster of workstations, and computational Grid), however each of them allows the execution of multiple queries simultaneously. The level of concurrency is limited by the number of processors that are available to the system. Since each query is executed by a processor, when all of the processors are busy, a new query received by the system is put into a *waiting* queue. A good query scheduling policy should attempt to order the execution of waiting queries so that they benefit from greater cache locality without starving other waiting queries. In a sense, a good cache replacement policy complements the query scheduling policy by maintaining in cache a *working set* of data aggregates manipulated by the current query workload.

In this section, we show how the data transformation model described in Section 3.2 can be used



(a) (b)

Figure 3: (a) Executing multiple queries over two parts of a Virtual Microscope slide (see Section 6.1) with different magnification levels. (b) The corresponding scheduling graph.

to drive better query scheduling decisions. Our approach to the multiple query scheduling problem is based on the use of a priority queue. Multiple ranking strategies are used to impose an execution order based on different inter-query dependency criteria. Each of them uses semantic information obtained from the data transformation model to assign weights for the waiting queries. 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 full or partial overlap, as seen in Figure 3. Each vertex represents a query that is either waiting to be computed, is currently being computed, or was recently computed and has its results cached. A directed edge $e_{i,j}$ in the graph connecting q_i to q_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 in question is not symmetric. 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 satisfy q_j using a projection primitive. Therefore, the scheduling process is tightly coupled with the optimization model, because the data transformation model is needed to analyze how two queries are related. In the framework, the weight $w_{i,j}$ associated with any two edges i and j is computed as follows:

$$w_{i,j} = \mathbf{overlap}(M_i, M_j) \times \mathbf{outputsize}(M_i), \tag{11}$$

The weight is essentially a measure of the number of bytes that can possibly be reused if q_i is executed first and then q_j is executed, so that the query plan q_j reuses the aggregates generated by q_i. Associated with each node $q_i \in V$ 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 four 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.

The scheduling model is dynamic in the sense that new queries can be inserted as they are received from clients. Upon receiving a new query, several events are triggered, causing topological and/or re-ranking modifications in the scheduling graph [9]. Once a query q_i is scheduled for execution, s_i is updated to `EXECUTING` and, once it finishes, its state is updated to `CACHED`, meaning that its results are available in the active semantic cache for reuse. If memory must be reclaimed for a cache replacement operation, the state of the query q_i will be set to `SWAPPED_OUT`, meaning that its results are no longer available for reuse (note that this implies communication between the query scheduler and the active semantic cache). 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 topological transformation triggers a re-computation of the ranks of the nodes that were previously neighbors with q_i. The up-to-date state of all queries is therefore always available to the query server. The ranking of nodes can be viewed as a topological sort operation. We briefly examine several ranking strategies; a detailed description of the strategies can be found in [9].

1. **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 waiting queries depend on a specific query, computed as $overlap \times outputsize$. The intuition behind this policy is that it quantifies how much other queries will benefit if query q_i is executed first.

2. **Farthest First (FF):** The nodes are ranked based on a measure of how much a query depends upon another query. When q_i depends on q_j, and q_i gets scheduled for execution while q_j

is executing, the computation of q_i will stall until q_j finishes because its results can be used to generate a portion of the result for q_i. Therefore, farthest first ranking gives a measure of how likely a query is to block because it depends on a result that is either being computed or is still waiting to be computed.

3. **Closest First (CF):** The queries are ranked using a measure of how many of the nodes a query depends upon have already executed (and whose results are possibly `CACHED`) or are currently under execution. The intuition behind this policy is that scheduling queries that are "close" has the potential to improve locality, making caching more beneficial.

4. **Closest and Non-Blocking First (CNBF):** In this strategy, nodes are ranked by a measure of how many of the nodes that a query depends upon have been or are being executed[2]. The intuition is the same as that for CF; however, CNBF attempts to prevent interlock situations.

As baseline cases in our experimental setup, we also investigated two traditional scheduling policies – First in First out (FIFO), in which queries are served in the order they arrive, and Shortest-Job-First (SJF) in which queries are ordered by their estimated execution times. The shorter the execution time for a query is, the higher its rank will be. The size in bytes of the processed input data, $inputsize(M_i)$, for query q_i is used as an estimate of the relative execution time of the query. The FIFO strategy targets fairness; queries are scheduled in the order they arrive. On the other hand, the goal of SJF is to reduce average waiting time by scheduling queries with shorter execution times earlier. The goal of MUF is to schedule earlier the queries that are the most beneficial for other waiting queries. The objective of FF is to avoid scheduling queries that are mutually dependent and have one of them wait for the other to complete. CF and CNBF are similar, but CF aims to achieve higher locality to make better use of cached (or soon to be cached) aggregates, while CNBF tries to improve locality without paying the price of having to wait for dependencies to be satisfied.

[2]We deduct the weight for the nodes being executed because we want to decrease the probability of having very little overlap due to the deadlock avoidance algorithm implemented in the query server (described in [3]).

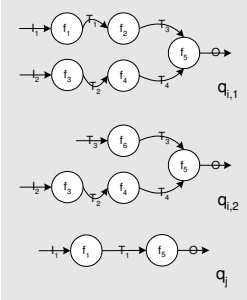


Figure 4: Two functionally decomposed queries q_i and q_j. q_i has two execution plans $q_{i,1}$ and $q_{i,2}$. f_6, in query $q_{i,2}$, is a projection primitive, taking an aggregate of type T_3 and generating a projected aggregate of the same type. In $q_{i,1}$, f_1 and f_3, the leaf nodes, use raw data as their input. Each T_i represents a potential reuse site.

5 Functional Decomposition

In many applications, the processing required to evaluate a query involves relatively complex operations that can be implemented from a set of *primitive* operations. We refer to an operation as primitive if it is an application-specific minimal and indivisible part of data processing. An example is the processing of satellite data, in which sensor data is first range-selected and subsampled, correction algorithms are applied to the data, an aggregation operation is performed, and, finally, a cartographic projection is carried out to yield the final query output, called a data product [29].

A complex function can often be defined as a composition of several primitive operations: $O \overset{M}{\leftarrow} f_1 \circ f_2 \ldots \circ f_n(I)$ where $f_1, f_2, \ldots f_n$ are the primitive operations and M is the domain as defined by the query meta-data. The implementation of a complex function as a monolithic computation results in a single aggregate becoming available for later data reuse, namely the output of the complex function. However, for each primitive function f_i, typically different algorithms can be chosen by a particular user query. Therefore, a monolithic implementation effectively reduces the likelihood of identifying reusable aggregates, even though some intermediate result could have been employed by other queries.

Based on this observation, we suggest the implementation of complex operations as a sequence of primitive operations: $T_1 \overset{M}{\leftarrow} f_n(I)$, $T_2 \overset{M}{\leftarrow} f_{n-1}(T_1)$, $\ldots$, $T_{n-1} \overset{M}{\leftarrow} f_2(T_{n-2})$ and $O \overset{M}{\leftarrow} f_1(T_{n-1})$, where T_1, T_2, $\ldots$, and T_{n-1} are intermediate temporary aggregates. When a complex operation is decomposed into a sequence of primitive operations, the query plan produces a processing chain in which aggregates generated by a primitive in the chain are used as input to the next primitive. In

essence, aggregates are materialized along the processing chain, in contrast to data elements being consumed as in a relational database *iterator*-based pipelined processing chain.

In general, the decomposition approach has high potential to increase data reuse opportunities (since the transformation model discussed in Section 3.2 can be employed by each primitive), at the expense of requiring more space for caching intermediate aggregates that would otherwise be discarded or not even generated, and more bookkeeping.

The primary step in decomposing a complex query consists of identifying the functional primitives that make up the query processing chain. In our approach, we expect the application developer to present to the system a functional description of the primitive operations. This assumption is also made by other frameworks that require the functional decomposition of a complex computation [2, 13, 36, 37].

The *execution chain* of a query type q_i is represented by a directed acyclic graph $G_i(V, E)$, referred to as a *query execution chain graph*. A vertex represents a function primitive and an edge corresponds to a data dependency between the two primitives sharing the edge. In the context of a query graph, a projection primitive can also be viewed as a function primitive. Hence it can be represented by a vertex in the graph. A vertex is referred to as a *sink*, or *output*, vertex if it is the one that generates the output data product for the query. A *source*, or *input*, vertex is a vertex that processes raw input data elements selected by the query. In a topological sort of the query graph, the sink vertex is at the top level (i.e., level zero), whereas source vertices form the bottom level of the query graph. An intermediate vertex at level l has the dual role of consuming the temporary dataset generated by the primitive(s) immediately before (at level $l + 1$) and generating the temporary dataset for the primitive(s) immediately after (at level $l - 1$). An example of query decomposition is shown in Figure 4. In the example, query q_i has two alternative query plans. The second query plan $q_{i,2}$ uses a projection primitive to decrease execution time by reusing a cached aggregate.

For a given query type q_i, the query graph, $G_i(V, E)$ is traversed in a breadth-first, top-down fashion, starting from the sink vertex. The algorithm displayed in Figure 5 is executed at each level of the graph. First, the semantic cache is searched for aggregates that overlap the primitive meta-data (step 1). If there is complete overlap (step 2), the output is computed from the cached aggregate by applying the appropriate projection primitive(s) (step 3). If cached aggregates can

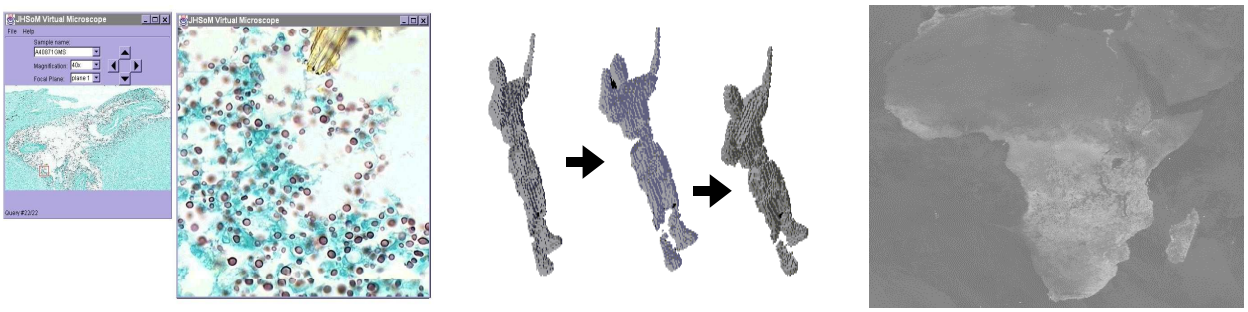
```
INPUT: ec the query graph and the level currently being executed
OUTPUT: executes a primitive by recursively computing the processing chain
1:  stat ← LOOKUP(ovlps) // locates cached aggregates that fully or partially overlap with this primitive. The compare and overlap operators are used to
    compare this primitive's meta-data with the meta-data for the cached aggregates and find the best match
2:  if stat=FULL_OVERLAP then // invokes the appropriate project operator to transform the cached aggregate into what this primitive needs
3:     PROJECT(ovlps) // uses the project operator
4:     return
5:  else
6:     if stat=PARTIAL_OVERLAP then // creates and runs subprimitives by using the difference operator to compute the non-overlapped areas
7:        GENERATEANDRUNSUBPRIMITIVES(ovlps,ec,level) // recursively calls the run method for each subprimitive. This causes an
          implicit call to the run method, so that other cached aggregates can be employed
8:        PROJECT(ovlps) // uses the project operator
9:        return
10:    else // no reuse has been detected, runs primitive and compute results from input data
11:       // a SOURCE_LEVEL primitive processes raw input data
          // a non-SOURCE_LEVEL primitive at level l processes input data generated by a primitive at level l+1
12:       if level ≠ SOURCE_LEVEL then // locates the input dataset for this primitive
13:          stat ←LOOKUP(input_dataset)
14:          if stat ≠ FULL_OVERLAP then // the input dataset is not completely cached, hence executes the primitive at level+1 that generates the
             input dataset
15:             RUN(ec,level+1) // runs the primitive and stores results in input_dataset
          // execute the primitive – calls the primitive-specific processing method for processing the input dataset
16:       EXECUTE()
```

Algorithm 1: PRIMITIVE::RUN(EXECUTIONCHAIN EC, LEVEL) – executes a query processing chain

only partially be used to compute the primitive under evaluation, subprimitives are recursively scheduled for execution to compute the incomplete regions (step 7). On the other hand, if no overlap was detected, the primitive must be executed from scratch by computing its result from input data. At this phase (step 12), there are two possibilities. If the primitive is a source vertex, its processing only requires access to the raw input data accessed from a data source. Otherwise, the primitive is an intermediate vertex. In that case, the algorithm attempts to locate the required input data in the cache (step 13). If the input dataset is not located, the primitive at the next level needs to be recursively executed to generate it (step 15). In either case, the input dataset will be available for processing at step 16. The `execute` method is application-specific and performs the computation necessary to generate the output dataset for a specific primitive from the input dataset. Each output dataset is cached, which may cause cache evictions according to one of the policies described in Section 4.

6 Case Study Applications

In order to showcase our techniques, we have chosen representative applications from three different domains: telepathology, computer vision, and satellite data processing. We describe two



(a) (b) (c)

Figure 5: (a) The Virtual Microscope client. (b) The result of a volumetric reconstruction query (3 frames). (c) A Kronos data product. A 7-day (January 1-7, 1992) composite using Maximum NDVI (normalized difference vegetation index) as the compositing criteria and Rayleigh/Ozone as the atmospheric correction method.

of these applications in more details. For a description of the Volumetric Reconstruction (VR) application, we refer the reader to [4, 15]. All of these applications have been ported to our system in terms of writing functionally decomposed primitives and customizing the optimization model operators (from Section 3) via C++ inheritance from the abstract classes provided by the support library that accompanies the database runtime system.

6.1 The Virtual Microscope

The Virtual Microscope (VM) application [1] implements a realistic digital emulation of a high power light microscope. Figure 5 displays the VM client Graphical User Interface. VM not only emulates 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, such as allowing multiple users to view different parts of the same slide simultaneously.

The raw input data for VM is captured by digitally scanning collections of microscope slides. Each digitized slide is stored on disk at the highest magnification level and can contain multiple focal planes. The size of one slide with a single focal plane can be up to several gigabytes, uncompressed. A VM query is described by a 3-tuple of the form [image id, bounding box, magnification algorithm]. During query processing, the data that intersect the query region, which is a two-dimensional rectangle within the input image, is retrieved from secondary storage and clipped

Figure 6: A high complexity Kronos query specified as a sequence of low-level functions. This query produces a *data product* transformed by a cartographic projection method.

to the query window. Then it is processed to compute the output image at the desired magnification. Two magnification algorithms – subsampling and pixel averaging – are available as primitives to process the high resolution clipped images to produce lower resolution images. Each of these results in a different version of VM. The first function employs a simple subsampling operation and the second 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 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 since it must aggregate several input pixels to compute an output pixel.

6.2 Satellite Data Processing – Kronos

Remote sensing has become a very powerful tool for geographical, meteorological, and environmental studies [29]. Usually systems processing remotely sensed data provide on-demand access to raw data and user-specified data product generation [21]. Kronos [29] is an example of such a class of applications. It targets datasets composed of remotely sensed AVHRR GAC level 1B (Advanced Very High Resolution Radiometer – Global Area Coverage) orbit data [35]. The raw data is continuously collected by multiple satellites and the volume of data for a single day is about 1 GB. An AVHRR GAC dataset consists of a set of Instantaneous Field of View (IFOV) records organized according to the scan lines of each satellite orbit. Each IFOV record contains the reflectance values for 5 spectral range channels. Each sensor reading is associated with a position (longitude and latitude) and the time the reading was recorded. Additionally, data quality indicators are stored with the raw data.

The processing structure of Kronos can be divided into several basic functions that form a processing chain on the sensor data (see Figure 6): **Range Selection** retrieves the relevant IFOVs

from the raw AVHRR data. **Atmospheric Correction** applies an atmospheric correction algorithm and modifies the relevant part of the selected input data tuples. **Composite Generator** aggregates many IFOVs for the same spatial region and multiple temporal coordinates. **Subsampler** converts the input data to a user-specified spatial resolution. **Cartographic Projection** applies a mapping function that converts a uniform 2-dimensional (spherical) grid into a particular cartographic projection. All these primitives may employ different algorithms (e.g., multiple atmospheric correction methods, multiple cartographic projection methods, etc) that are specified by the query predicate.

Several types of queries can be posed to a Kronos-like system. Queries can be as simple as visualizing the remotely sensed data for a given region using a particular cartographic projection [29], or as complex as statistically comparing a *composite* data product across two different time periods [40]. As a result, queries can be as simple as executing a single primitive or as complex as using all the available primitives, as shown in Figure 6. The functional decomposition of queries permits the utilization of several **project** operators as stated in Section 5. Therefore, when cached aggregates are available, the optimization framework can exploit different kinds of reuse for a single query.

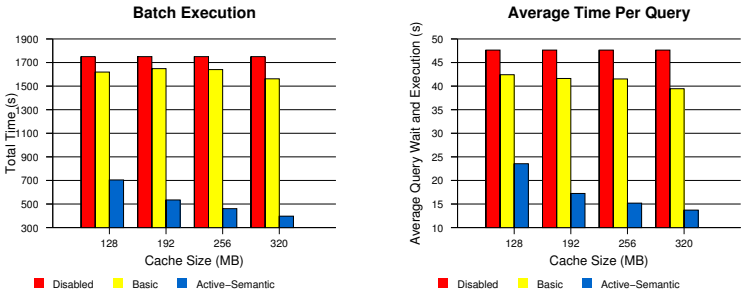
For our study, queries are defined as a 4-tuple: [spatio-temporal bounding box, spatio-temporal resolution, correction method, compositing method]. The spatio-temporal bounding box specifies the spatial and temporal coordinates for the data of interest. The spatio-temporal resolution describes the amount of data to be aggregated per output point (i.e., each output pixel is composed from x input points, so that an output pixel corresponds to an area of, for example, 8 Km^2). The correction method and the compositing method specify, respectively, the atmospheric correction algorithm to be applied to the raw data to approximate the values for each IFOV to the *ideal* corrected values (by trying to eliminate spurious effects caused by, for example, water vapor in the atmosphere), and the aggregation level and function to be employed to *coalesce* multiple input grid points into a single output grid point.

For the query types supported by Kronos, the dimensional (spatio-temporal) overlap projection primitive can be employed on the output of the Range Selection function to eliminate the data elements that fall outside the bounding box of a query being evaluated by reusing an overlapping cached aggregate. The dimensional overlap primitive can similarly be used on the output of other

basic Kronos functions. The invertible aggregation primitive can be used on the output of the Atmospheric Correction function. The original uncorrected sensor data can be obtained from a cached aggregate by inverting the correction algorithm used to generate that aggregate. Both the composable reduction operation primitive and the inductive aggregation primitive can be employed for the Composite Generator function. For composable reductions, the aggregation level across the two aggregates (the one cached and the one under computation) must be congruent. In that case, for example, we may want to compute a data product for a week's worth of data. Available cached aggregates for days 1, 2, and 3 can be merged using the compositing method to generate a temporary intermediate aggregate. A subquery for days 4 to 7 would be automatically issued (using the difference operator), and that result would be composed with the temporary aggregate, producing the final result for days 1 to 7. For inductive aggregations, an aggregate can be used as an initial partial result for computing a new aggregate in which additional data must be processed using the raw input data. The inductive aggregation primitive can be used for the Subsampler function to compute a lower resolution output from a higher resolution aggregate, as well as a higher resolution aggregate from a lower resolution aggregate by employing additional input data.

7 Experimental Results

Employing the optimization model and utilizing the data transformation framework can significantly improve the query execution performance. The improvements are associated with the amount of reuse that can be detected and leveraged, as well as with how effectively the active semantic cache keeps the aggregates with the highest potential for reuse in cache and how effectively the query scheduler selects a query for execution in order to benefit from the current state of the system. Ultimately, the amount of data locality in the workload determines how effective the optimization framework will be. Our performance experiments attempt to capture these effects. In our discussion, we will typically show the improvements in terms of three metrics. **Query Wait Time** (QW) is the amount of time elapsed between when a query is received by the system and when it is scheduled for execution. **Query Execution Time** (QE) is the amount of time from the moment it is scheduled for execution until it is completely evaluated. And **Query Wait and Execution Time** (QWE), which is simply the result of $QW + QE$.



(a) (b)

Figure 7: Caching strategies. *Basic* allows reuse of aggregates that are a complete match. *Active-Semantic* caching allows reuse of aggregates when partial overlap occurs and data transformation is necessary. (a) Batch execution time. (b) Average execution time per query.

7.1 Improving Performance via Data Reuse

As we have discussed, the main feature of our optimization framework is the reuse of cached aggregates by applying projection primitives. Such reuse is profitable when performing the transformation is less expensive than recomputing the aggregate from input data. The results in Figure 7 compare the performance of *active semantic caching* with both *no* caching and a *basic* semantic cache implementation, where basic semantic caching only uses a cached aggregate if it *exactly* matches the requested aggregate.

We assembled an experimental setup using a heterogeneous distributed configuration of the database system consisting of clients, application database servers, and an instance of a proxy that functions as the entry point to the system [6] (this is one example of many parallel configurations that the database system can be configured to use [3]). We have instantiated 5 database servers on 5 nodes including: an 8-processor 550MHz Pentium III Linux SMP machine hosting 8 datasets (two for Volumetric Reconstruction and six for the Virtual Microscope) and multiple single-processor 650 MHz Pentium III machines hosting copies of the same six VM datasets. The SMP machine had a single database server serving both VM and VR queries, with up to 4 queries under simultaneous execution. The uniprocessor machines were able to answer a single query at any given time. A 24-processor UltraSparc III SMP Solaris machine hosted the proxy entrypoint to the collective database system.

We employed 12 clients. Four of them generated 16 queries each for volume reconstruction,

and the other eight each generated 32 pixel averaging VM queries. The VM clients used a workload model that emulates the behavior of real users as described in [14] and the VR clients used a synthetic workload model fully described in [6]. Both workload models define how queries by a single client are going to be generated, and rely on locality meaning that there are several *interesting regions* within a dataset and the client is going to generate a request for those regions with higher probabilities than for the uninteresting areas. Noise is also added to the spatial boundaries of a query to ensure that no two queries are exactly the same and queries from different clients are not related. We chose to use the workload generator 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 parameters (both the number of clients and the number of queries) in a controlled way. Each VM dataset is a 10000×10000 3-byte per pixel image, totaling around 1.8 GB for the six images. Each VR dataset is a 5200 frame collection (13 cameras, 400 frames), totaling approximately 780 MB for the two collections. In all of our experiments, the emulated client applications were executed simultaneously on a cluster of PCs connected to the SMP machine via 100 Mbit Ethernet. Each client submitted queries independently from the other clients, but waited for the completion of one of its query before submitting another one.

Figure 7 shows that basic caching improves batch execution time by around 10% compared to no caching at all, and that increasing the cache size for basic caching does not significantly improve performance. On the other hand, active semantic caching decreases batch execution time from 56% up to 75% compared to basic caching, and more caching space implies lower query computation time. Similar results are also observed when measuring the average query execution time (Figure 7(b)).

7.2 Improving Performance via Functional Decomposition

Functionally decomposing the query types can potentially increase overall system performance. We highlight one of several results obtained by optimizing the execution of multiple Kronos query workloads. This result shows how functional decomposition decreases average query execution time. More extensive experimental results, including scalability behavior, the impact of different projection primitives, and the impact of increasing cache size are reported in [7]. Extensive experi-

mental results for cache replacement policies can be found in [4]. All the experiments were run on a 24-processor SunFire 6800 machine with 24 GB of main memory running Solaris 2.8. A dataset containing one month (January 1992) of AVHRR data was used, totaling about 30 GB.

In order to evaluate the benefits of partitioning an application into primitives and to quantitatively measure performance, we investigated the system behavior using a synthetic workload model. We employed a variation of the Customer Behavior Model Graph (CBMG), which is a technique utilized, for example, by researchers analyzing performance aspects of e-business applications and website capacity planning [33]. A CBMG can be characterized by a set of n states, a set of transitions between states, and by an $n \times n$ matrix, $P = [p_{i,j}]$, of transition probabilities between the n states. A typical query for Kronos may employ all the primitives seen in Section 6.2. A query specifies a geographical region, a set of temporal coordinates (a continuous period of days), a resolution level (both vertical and horizontal), a correction algorithm (from 3 possibilities), and a compositing operator (also from 3 different algorithms). Once a query is executed, the expected transitions from that query to the next one issued by the same client are given by one of the following operations: *spatial movement*, *temporal movement*, *resolution increase* or *decrease*, applying a different *correction algorithm*, or applying a different *compositing operator*. For our experiments, we used the transition matrix in Table 1 to build the CBMG. *Workload 1* was used for all the experiments and *workload 2* was used specifically to help analyze the performance impact of functional decomposition. For a given client, the initial state – empty – must use the *New Point-of-Interest* transition to get started. During startup, the client window dimensions (i.e., the size of the data product generated by a query) is also pre-selected and kept constant across the client session. From that point forward, each transition has its own fixed probability of being selected. The number of queries to be generated per client is fixed, and that number is also an input variable to the model. More details about the workload model can be found in [7].

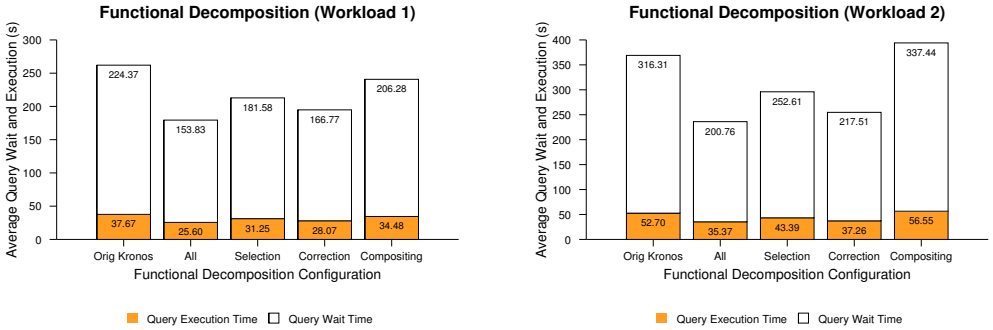
In order to evaluate functional decomposition, we have collected experimental results for a single configuration: 16 clients generating 4 queries each[3], submitting them to the version of Kronos running on our database server, configured with a fixed cache size of 1 GB and with the ability to service up to 2 queries at the same time[4]. Two different workloads were employed: workload

[3] A client application waits for a query to be answered before it submits its next query

[4] In its original form Kronos can only process one query at a time. To make it able to process two queries simulta-

Transition	*Workload 1*	*Workload 2*
New Point-of-Interest	5%	5%
Spatial Movement	40%	10%
New Resolution	15%	10%
Temporal Movement	5%	5%
New Correction	15%	30%
New Compositing	5%	30%
New Compositing Level	15%	10%

Table 1: Transition probabilities. A sequence of queries for one user is generated as follows: the first query always uses *New Point-of-Interest* and the next queries are generated based on the modifications of the query attributes based on the transition selected with the probabilities in the table.



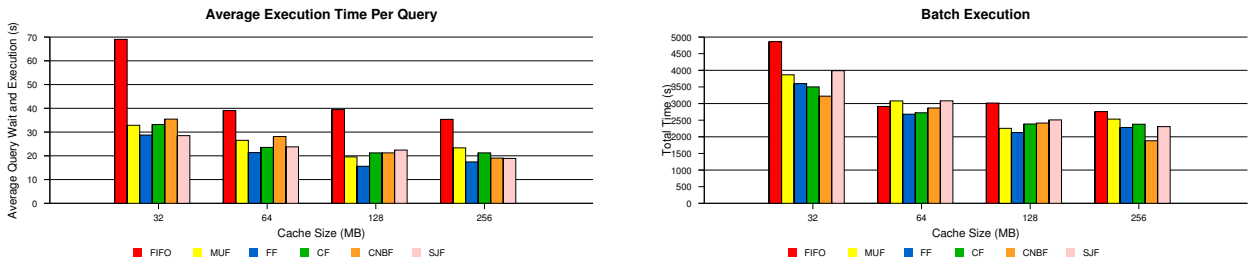
(a) (b)

Figure 8: The effects of functional decomposition for (a) Workload 1 and (b) Workload 2. Average time per query is shown for 1) the original Kronos code, 2) for caching at all levels, 3) caching only at range-selection, 4) caching only at correction, and 5) caching only at the last stage of the execution chain – compositing.

1 and workload 2, with transition probabilities shown in Table 1. Workload 1 has the bulk of its transitions driven by spatial movement, which means that a great deal of reuse across queries can occur at the compositing level (making caching at that level alone potentially beneficial). On the other hand, workload 2 has most of its transitions changing the compositing and atmospheric correction methods, which requires reaggregating the data at the compositing level (making caching at that level not especially beneficial).

Of particular interest in Figures 8 (a) and (b) is that caching at all levels improves performance much more than configurations in which only a single reuse point was exposed. For the two work-

neously, we also wrote a wrapper that can spawn multiple separate Kronos processes on demand to produce the desired multiprocessing level.



(a) (b)

Figure 9: (a) The 95%-trimmed mean for the query response time, when more memory is added to the cache. (b) The total execution time of a single batch of 256 queries, as the memory allocated to the cache is increased. For both figures, up to 4 queries are allowed execute simultaneously.

loads, caching at all levels achieves between 30% and 36% decrease in average query execution time compared to the original Kronos implementation. For workload 2, we see that caching only at the last primitive in the processing chain is not helpful because very little reuse can occur at that level.

7.3 Improving Performance through Query Scheduling Heuristics

In the previous experiments, queries were scheduled in FIFO order. Now we show evidence that scheduling decisions based on the data transformation model can improve performance. Since all of the ranking methods are heuristic, in order to quantify their relative effectiveness and to assess how they compare to the traditional FIFO and SJF scheduling, we devised an experimental setup employing Virtual Microscope queries using the subsampling and pixel averaging implementations.

The results were obtained on the 24-processor SMP, running Solaris 2.8, described in Section 7.2. In the experiments, we turned off the operating system file cache via the `directio` Solaris function, so that no operating system file caching occurs. We employed the same three VM datasets described in Section 7.1, and the same client setup and driver program.

In one set of experiments, we evaluated the effect of simply using active data caching (*versus* not using it) on the performance of the FIFO and SJF strategies. Neither strategy takes into account the state of the cache for scheduling queries. However, we observed that overall system performance improved by as much as 35% and 70% for FIFO and 40% and 70% for SJF for the

subsampling and averaging implementations of the Virtual Microscope, respectively. Performance also increased as more memory was allocated for caching.

The second set of experiments evaluated how each scheduling policy affects the database performance from the perspective of average QWE time and total execution time for a query batch. Figure 9(a) shows the 95%-trimmed mean[5] for the query response time achieved by different ranking strategies. As is seen from the figure, FIFO performs discernibly worse than the other ranking strategies, as expected. The MUF, FF, CF, and CNBF strategies perform slightly better than SJF in most cases. The SJF strategy aims to minimize average waiting time by executing shorter queries first, but may suffer from less reuse of cached results.

The other objective of employing multi-query optimization strategies is to decrease the overall execution time for all the queries in a batch (i.e., increase the system throughput) in contrast with lowering the execution time per individual query. In fact, a client may submit a batch of queries to the server. For example, if we want to create a movie from a case study using the Virtual Microscope, we may want to submit a set of queries, each of which corresponds to visualization of a specific part of the slide being studied. In that case, it is important to decrease overall execution time for the query batch. Figure 9 (b) shows the overall 95%-trimmed mean execution time of 256 queries as a single batch (although in this case, individual clients still submit one query at time). CF and CNBF result in better performance than do other strategies, especially when resources are scarce (i.e., a small cache). Our results show that when the objective is to minimize the total execution time of a query batch, taking into account locality is important for better performance, since the cache is better utilized. Additional query scheduling results including different workloads and experimental setup are available in [9].

8 Conclusions

We have investigated the problem of optimizing multi-query workloads for data analysis applications in the presence of user-defined, application-specific aggregations and operations. We presented a processing model common to many of these applications and identified optimization

[5] A 95%-trimmed mean is computed by discarding the lowest and highest 2.5% of the scores and taking the mean of the remaining scores.

strategies based on a data transformation model that makes use of active semantic caching. To provide a mechanism for identifying data and computation reuse opportunities, we devised an extensible optimization framework (with user-customizable operators) based on a data transformation model that allows a runtime system to infer how a data product can be transformed into another that is usable for other queries submitted for processing. Workloads based on real usage profiles, as well as synthetic workloads, were employed to thoroughly analyze the behavior of various aspects of the system and the optimization model.

The basic optimization model has been extended to incorporate query scheduling techniques and caching infrastructure improvements. We showed that scheduling is an effective technique to further improve performance. And we investigated the use of *informed* query scheduling techniques that take into consideration the relationships among queries that are waiting to be executed, queries under execution, and queries whose results have been cached. This information is made explicit by the same data transformation model employed for detecting reuse sites, and was shown to be useful to reorder the execution of waiting queries to better exploit the alternative execution paths made available by the optimization transformations.

We also discussed functional decomposition as a mechanism for exposing more sites for optimization to the data transformation model. We proposed a methodology for breaking up complex queries into atomic primitives, and showed how queries can be executed by employing a top-down, recursive greedy algorithm for identifying all reusable aggregates available in the cache.

Our optimization framework addresses some of the main shortcomings of previous multi-query optimization methods for handling application-specific aggregation operations. As a result, when real data analysis queries are executed using a prototype implementation of the database middleware, the performance optimizations enable large decreases in both single query and query batch execution times. Moreover, we have gone beyond the basic issue of handling application-specific operations and showed that our overall strategy of closely coupling the basic optimization framework with (1) more complex techniques for functionally decomposing complex queries, (2) query scheduling and (3) informed cache replacement policies, leads to large improvements in the overall performance of data analysis queries.

References

[1] 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 *AMIA98*. American Medical Informatics Association, November 1998.

[2] K. Amiri, D. Petrou, G. R. Ganger, and G. A. Gibson. Dynamic function placement for data-intensive cluster computing. In *Proceedings of the 2000 USENIX Symposium on Internet Technologies and Systems*, San Diego, CA, 2000.

[3] H. Andrade. *Multiple Query Optimization Support for Data Analysis Applications*. PhD thesis, Department of Computer Science, University of Maryland, December 2002.

[4] H. Andrade, T. Kurc, A. Sussman, E. Borovikov, and J. Saltz. On cache replacement policies for servicing mixed data intensive query workloads. In *Proceedings of the 2nd Workshop on Caching, Coherence, and Consistency, held in conjunction with the 16th ACM International Conference on Supercomputing*, New York, NY, June 2002.

[5] H. Andrade, T. Kurc, A. Sussman, and J. Saltz. Efficient execution of multiple workloads in data analysis applications. In *Proceedings of the 2001 ACM/IEEE Supercomputing Conference*, Denver, CO, November 2001.

[6] H. Andrade, T. Kurc, A. Sussman, and J. Saltz. Active Proxy-G: Optimizing the query execution process in the Grid. In *Proceedings of the 2002 ACM/IEEE Supercomputing Conference*, Baltimore, MD, November 2002.

[7] H. Andrade, T. Kurc, A. Sussman, and J. Saltz. Exploiting functional decomposition for efficient parallel processing of multiple data analysis queries. Technical Report CS-TR-4404 and UMIACS-TR-2002-84, University of Maryland, October 2002. A shorter version appears in the Proceedings of IPDPS 2003.

[8] H. Andrade, T. Kurc, A. Sussman, and J. Saltz. Multiple query optimization for data analysis applications on clusters of SMPs. In *Proceedings of the 2nd IEEE/ACM International Symposium on Cluster Computing and the Grid*, Berlin, Germany, May 2002.

[9] H. Andrade, T. Kurc, A. Sussman, and J. Saltz. Scheduling multiple data visualization query workloads on a shared memory machine. In *Proceedings of the 2002 IEEE International Parallel and Distributed Processing Symposium*, Fort Lauderdale, FL, April 2002.

[10] Association for Pathology Informatics. *http://www.pathologyinformatics.org*.

[11] Y. Arens and C. A. Knoblock. Intelligent caching: Selecting, representing, and reusing data in an information server. In *Proceedings of 1994 ACM International Conference on Information and Knowledge Management*, pages 433–438, 1994.

[12] M. D. Beynon, C. Chang, U. Catalyurek, T. Kurc, A. Sussman, H. Andrade, R. Ferreira, and J. Saltz. Processing large-scale multidimensional data in parallel and distributed environments. *Parallel Computing*, 28(5):827–859, May 2002. Special issue on Data Intensive Computing.

[13] M. D. Beynon, T. Kurc, U. Catalyurek, C. Chang, A. Sussman, and J. Saltz. Distributed processing of very large datasets with DataCutter. *Parallel Computing*, 27(11):1457–1478, October 2001.

[14] M. D. 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*, Rhodes, Greece, June 1999. ACM Press.

[15] E. Borovikov, A. Sussman, and L. Davis. A high performance multi-perspective vision studio. In *Proceedings of the 2003 International Conference on Supercomputing*, San Francisco, CA, June 2003. ACM Press.

[16] R. Bramley, K. Chiu, S. Diwan, D. Gannon, M. Govindaraju, N. Mukhi, B. Temko, and M. Yechuri. A component based services architecture for building distributed applications. In *Proceedings of the Ninth IEEE International Symposium on High Performance Distributed Computing (HPDC-9)*, Pittsburgh, PA, August 2000.

[17] P. Cao and S. Irani. Cost-aware WWW proxy caching algorithms. In *Proceedings of the 1997 USENIX Symposium on Internet Technologies and Systems*, pages 193–206, Monterey, CA, December 1997.

[18] Common Component Architecture Forum. *http://www.cca-forum.org*.

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

[20] C. Chang. *Parallel Aggregation on Multi-Dimensional Scientific Datasets*. PhD thesis, Department of Computer Science, University of Maryland, April 2001.

[21] C. Chang, B. Moon, A. Acharya, C. Shock, A. Sussman, and J. Saltz. Titan: a High-Performance Remote-Sensing Database. In *Proceedings of the 13th International Conference on Data Engineering*, 1997.

[22] S. 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.

[23] S. Chaudhuri, R. Krishnamurthy, S. Potamianos, and K. Shim. Optimizing queries with materialized views. In *Proceedings of the 11th International Conference on Data Engineering*, pages 190–200, Los Alamitos, CA, 1995. IEEE Computer Society Press.

[24] F.-C. F. Chen and M. H. Dunham. Common subexpression processing in multiple-query processing. *IEEE Transactions on Knowledge and Data Engineering*, 10(3):493–499, 1998.

[25] S. Dar, M. J. Franklin, Björn Jónsson, D. Srivastava, and M. Tan. Semantic data caching and replacement. In *Proceedings of the 22th VLDB Conference*, pages 330–341, Mumbai, India, September 1996.

[26] A. Gupta, S. Sudarshan, and S. Vishwanathan. Query scheduling in multi query optimization. In *International Database Engineering and Applications Symposium, IDEAS'01*, pages 11–19, Grenoble, France, 2001.

[27] R. Hamming. *Numerical Methods for Scientists and Engineers*. McGraw-Hill, New York, NY, 1962.

[28] High Performance Fortran Forum. High Performance Fortran – language specification – version 2.0. Technical report, Rice University, January 1997. Available at http://www.netlib.org/hpf.

[29] S. Kalluri, Z. Zhang, J. JáJá, D. Bader, N. E. Saleous, E. Vermote, and J. R. G. Townshend. A hierarchical data archiving and processing system to generate custom tailored products from AVHRR data. In *1999 IEEE International Geoscience and Remote Sensing Symposium*, pages 2374–2376, 1999.

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

[31] T. Kurc, U. Catalyurek, C. Chang, A. Sussman, and J. Saltz. Visualization of very large datasets with the Active Data Repository. *IEEE Computer Graphics and Applications*, 21(4):22–33, July/August 2001.

[32] M. Mehta, V. Soloviev, and D. J. DeWitt. Batch scheduling in parallel database systems. In *Proceedings of the 9th International Conference on Data Engineering*, Vienna, Austria, 1993.

[33] D. A. Menascé and V. A. F. Almeida. *Scaling for E-Business*. Prentice Hall PTR, 2000.

[34] H. Mistry, P. Roy, S. Sudarshan, and K. Ramanritham. Materialized view selection and maintenance using multi-query optimization. In *Proceedings of the 2001 ACM-SIGMOD Conference*, Santa Barbara, CA, 2001. ACM Press.

[35] National Oceanic and Atmospheric Administration. *NOAA Polar Orbiter User's Guide – November 1998 Revision*. compiled and edited by Katherine B. Kidwell. Available at http://www2.ncdc.noaa.gov/docs/podug/cover.htm.

[36] R. Oldfield and D. Kotz. Armada: A parallel file system for the computational grid. In *Proceedings of the 1st IEEE International Symposium on Cluster Computing and the Grid*, Brisbane, Australia, May 2001. IEEE Computer Society Press.

[37] B. Plale and K. Schwan. Dynamic querying of streaming data with the dQUOB system. *IEEE Transactions on Parallel and Distributed Systems*, 14(4):422–432, April 2003.

[38] Q. Ren, M. H. Dunham, and V. Kumar. Semantic caching and query processing. *IEEE Transactions on Knowledge and Data Engineering*, 15(1):192–210, 2003.

[39] M. Rodríguez-Martínez and N. Roussopoulos. MOCHA: A self-extensive database middleware system for distributed data sources. In *Proceedings of the 2000 ACM-SIGMOD Conference*, pages 213–224, Dallas, TX, 2000.

[40] D. P. Roy, L. Giglio, J. D. Kendall, and C. Justice. Multi-temporal active-fire based burn scar detection algorithm. *International Journal of Remote Sensing*, 20(5):1031–1038, 1999.

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

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

[43] A. Sinha and C. 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.

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