Efficient Execution of Multi-Query Data Analysis Batches Using Compiler Optimization Strategies*

Henrique Andrade[†,‡], Suresh Aryangat[†], Tahsin Kurc[‡], Joel Saltz[‡], Alan Sussman[†]

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

[‡] Dept. of Biomedical Informatics
The Ohio State University
Columbus, OH, 43210
{kurc.1,saltz.3}@osu.edu

Abstract

This work investigates the leverage that can be obtained from compiler optimization techniques for efficient execution of multi-query workloads in data analysis applications. Our approach is to address multi-query optimization at the algorithmic level by transforming a declarative specification of scientific data analysis queries into a high-level imperative program that can be made more efficient by applying compiler optimization techniques. These techniques – including loop fusion, common subexpression elimination and dead code elimination – are employed to allow data and computation reuse across queries. We describe a preliminary experimental analysis on a real remote sensing application that is used to analyze very large quantities of satellite data. The results show our techniques achieve sizable reduction in the amount of computation and I/O necessary for executing query batches and in average executing times for the individual queries in a given batch.

1 Introduction

Multi-query optimization has been investigated by several researchers, mostly in the realm of relational databases [8, 11, 20, 28, 29, 33]. We have devised a database architecture that allows efficiently handling multi-query workloads in data analysis applications where user-defined operations are also part of the query plan [2, 5]. The architecture builds on a data and computation reuse model that can be used to systematically expose reuse sites in the query plan when application-specific aggregation methods are employed. This model relies on an *active semantic cache*, in which semantic information is attached to prior computed aggregates that are cached by the system. This permits the query optimizer to retrieve the matching aggregates based on the metadata description of a new query. The cache is active in that it allows application-specific transformations to be performed on the cached aggregates so that they can be reused to speed up the evaluation of the query at hand. The reuse model and active semantic caching have been shown to effectively decrease the average turnaround time for a query, as well as to increase the database system throughput [2, 3, 5].

As we have described, our original system relies on sophisticated caching for optimizing multiple queries. On the other hand, a well-accepted definition of the multi-query optimization problem is as follows: "to minimize the total cost of processing a series of queries by creating an optimized access plan for the entire query sequence" [29]. This definition implies that a batch of queries is available to the query optimizer for evaluation at a given point in time. Our earlier approach does not require this *synchronization* point, because it leverages data and computation reuse for queries submitted to the system over an extended period of time. For a batch of queries, on the other hand, a global query plan that accommodates all the queries can be more profitable than creating individual query plans and scheduling queries based on those plans, especially if information at the algorithmic level for each of the query plans is exposed. A similar observation was the motivation for a study done by Kang et al. [20] for relational operators.

The need to handle query batches arises in many situations. In a data server concurrently accessed by many clients, there can be multiple queries awaiting execution. Sophisticated sense-and-respond systems [7] also require handling query batches. In such systems, queries are frequently persistent, meaning that the set of queries sent for processing does not vary much, i.e., at given points in time

*This research was supported by the National Science Foundation under Grants #EIA-0121161, #EIA-0121177, #ACI-9619020 (UC Subcontract #10152408), #ACI-0130437, #ACI-0203846, and #ACI-9982087, and Lawrence Livermore National Laboratory under Grant #B517095 and #B500288.

(e.g., every 10 minutes, daily, or weekly) a set of queries are submitted for processing. Although the queries are persistent, the data is transient, changing over time. A typical example is the daily execution of a set of queries for detecting the probability of wildfire occurring in Southern California. In this context, a sense-and-respond system could issue multiple queries in batch mode to analyze the current (or close to current) set of remotely sensed data at regular intervals and trigger a response by a fire brigade, which could, in turn, start preventive and controlled fires to avoid a catastrophic event. In such a scenario, a pre-optimized batch of queries can result in better resource allocation and scheduling decisions by employing a single comprehensive query plan. Indeed, this could yield significant decreases in average query execution time.

Many researchers have worked on database support for scientific databases. SEQUOIA 2000 [31] is one of the pioneering projects in that arena. Commercial projects [12] have also identified requirements that are specific to scientific applications. Optimizing query processing for scientific applications using compiler optimization techniques has also attracted attention of several researchers including those in our own group. Ferreira et. al. [14, 15, 16, 17] have done extensive studies on using compiler and runtime analysis to speed up processing for scientific queries. In particular, they have investigated compiler optimization issues related to single queries with spatio-temporal predicates, which are similar to the ones we target [15].

In this work, we investigate the application of compiler optimization strategies to execute a *batch* of queries for scientific data analysis applications as opposed to a single query. Our approach is a multi-step process consisting of the following tasks: 1) Converting a declarative data analysis query into an imperative description; 2) Assigning the set of imperative descriptions for the queries in the batch to the query planner; 3) Employing traditional compiler optimization strategies such as common subexpression elimination, dead code elimination, and loop fusion, so the planner generates a single, global and more efficient query plan.

The rest of this paper is organized as follows. Section 2 elaborates on our approach, describing abstractly the kind of scientific applications we target, as well as the general processing format of their queries. It also describes a real remote sensing application as a motivating scenario. Section 3 describes how our optimization strategy works and is integrated into our existing database framework. Section 4 presents experimental evidence showing the benefits obtained from employing the optimizations. And, finally, Section 5 presents a summary and concluding remarks, as well as descriptions of a few extensions that we intend to tackle in the near future.

$\mathcal{R} \leftarrow Select(I, O, M_i)$
foreach(r $\in$ $\mathcal{R}$) {
  $O[\mathcal{S}_L(r)] \quad = \quad \mathcal{F}(O[\mathcal{S}_L(r)], I_1[\mathcal{S}_{R1}(r)], \ldots, I_n[\mathcal{S}_{Rn}(r)])$
}

Figure 1. General Data Reduction Loop.

2 Query Optimization Using Compiler Optimization Techniques

In this section, we describe the class of data analysis queries targeted in this work, and present an overview of the optimization phases for a batch of queries.

2.1 Data Analysis Queries

Queries in many data analysis applications [1, 6, 10, 21] can be defined as range-aggregation queries (RAGs) [9]. The datasets for range-aggregation queries can be classified as *input*, *output*, or *temporary*. **Input** (*I*) datasets correspond to the data to be processed. **Output** (*O*) datasets are the final results from applying one or more operations to the input datasets. **Temporary** (*T*) datasets (temporaries) are created during query processing to store intermediate results. A user-defined data structure is usually employed to describe and store a temporary dataset. Temporary and output datasets are tagged with the operations 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.

A RAG query typically has both spatial and temporal predicates, namely a multi-dimensional bounding box in the underlying multi-dimensional attribute space of the dataset. Only data elements whose associated coordinates fall within the multidimensional box must be retrieved and processed. The selected data elements are mapped to the corresponding output dataset elements. The mapping operation is an application-specific function that often involves finding a collection of data items using a specific spatial relationship (such as intersection), possibly after applying a geometric transformation. An input element can map to multiple output elements. Similarly, multiple input elements can map to the same output element. An application-specific aggregation operation (e.g., sum over selected elements) is applied to the input data elements that map to the same output element.

Borrowing from a formalism proposed by Ferreira [14], a range-aggregation query can be specified in the general loop format shown in Figure 1. A *Select* function identifies the subdomain that intersects the query metadata M_i for a query q_i. The subdomain can be defined in the input attribute space or in the output space. For the sake of

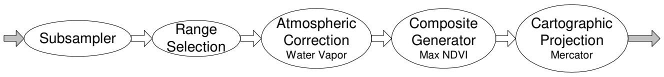

Figure 2. A typical Kronos query specified as a sequence of low-level primitive functions. This query produces a *data product* transformed using a cartographic projection method.

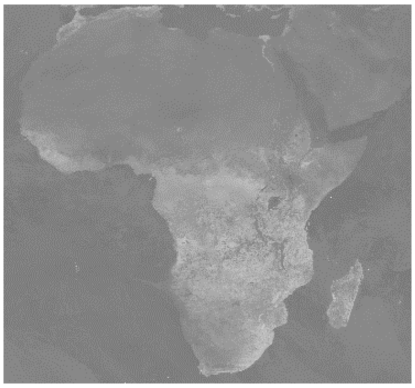

Figure 3. A Kronos data product. A 7-day (January 1-7, 1992) composite using Maximum NDVI (normalized difference vegetation index) as the compositing criteria and Rayleigh/Ozone as the atmospheric correction method.

discussion, we can view the input and output datasets as being composed of collections of objects. An object can be a single data element or a data chunk containing multiple data elements. The objects whose elements are updated in the loop are referred to as *left hand side*, or LHS, objects. The objects whose elements are only read in the loop are considered *right hand side*, or RHS, objects.

During query processing, the subdomain denoted by $\mathcal{R}$ in the *foreach* loop is traversed. Each point r in $\mathcal{R}$ and the corresponding *subscript functions* $\mathcal{S}_L(r), \mathcal{S}_{R1}(r), \ldots, \mathcal{S}_{Rn}(r)$ are used to access the input and output data elements for the loop. In the figure, we assume that there are n RHS collections of objects, denoted by $I_1, \ldots, I_n$, contributing the values of a LHS object. It is not required that all n RHS collections be different, since different subscript functions can be used to access the same collection.

In iteration r of the loop, the value of an output element $O[\mathcal{S}_L(r)]$ is updated using the application-specific function $\mathcal{F}$. The function $\mathcal{F}$ uses one or more of the values $I_1[\mathcal{S}_{R1}(r)], \ldots, I_n[\mathcal{S}_{Rn}(r)]$, and may also use other scalar values that are inputs to the function, to compute an aggregate result value. The aggregation operations typically implement *generalized reductions* [18], which must be commutative and associative operations. A commutative and associative aggregation operation produces the same output value irrespective of the order in which the input 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 final output can be generated by combining the temporary datasets.

2.2 Case Study Application – Kronos

Before we present our approach and system support for multi-query optimization for query batches, we will briefly describe the Kronos application used as a case study in this paper.

Remote sensing has become a very powerful tool for geographical, meteorological, and environmental studies [19]. Usually systems processing remotely sensed data provide on-demand access to raw data and user-specified data product generation [10]. Kronos [19] 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 [24]. The raw data is continuously collected by multiple satellites and the volume of data for a single day is about 1GB. 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 primitives that form a processing chain on the sensor data (see Figure 2 which can produce a data product of the form shown in Figure 3):

1. **Range Selection (Retrieval)** retrieves the relevant IFOVs from the raw AVHRR data.

2. **Atmospheric Correction (Correction)** applies an atmospheric correction algorithm and modifies the relevant part of the selected input data tuples.

3. **Composite Generator (Composite)** aggregates many IFOVs for the same spatial region and multiple temporal coordinates.

4. **Subsampler** converts the input data to a user-specified spatial resolution. In our system, subsampler was not actually implemented as a primitive. Rather, since its

behavior only alters the discretization level of the output grid (e.g., one pixel per each 4 Km^2), the information is included into the query bounding box.

5. **Cartographic Projection (Projection)** applies a mapping function that converts a uniform 2-dimensional (spherical) grid into a particular cartographic projection.

All the primitives (with the exception of Range Selection) may employ different algorithms (i.e., multiple atmospheric correction methods) that are specified as a parameter to the actual primitive (e.g., Correction(T0,Rayleigh/Ozone), where Rayleigh/Ozone is an existing algorithm and T0 is the aggregate used as input). In fact, Kronos implements 3 algorithms for atmospheric correction, 3 different composite generator algorithms, and more than 60 different cartographic projections.

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 [19], or as complex as statistically comparing a *composite* data product across two different time periods [27]. As a result of that, queries can be as simple as executing a single primitive or as complex as using all the available primitives, as shown in Figure 2.

2.3 An Approach to Solving the Multi-Query Optimization Problem

The objective of multi-query optimization is to take a batch of queries, expressed by a set of declarative query definitions (e.g., using the SQL extensions of PostgreSQL [26]), and generate a set of optimized data parallel reduction loops that represent the global query plan for the queries in the batch. Optimization is a multi-phase process, in which declarative query definitions are first converted into imperative loops that conform to the canonical data reduction loop of Figure 1, and then those loops are optimized using various compiler techniques.

Consider Kronos queries as examples. For our study, queries are defined as a 3-tuple: [spatio-temporal bounding box and spatio-temporal resolution, correction method, compositing method]. The spatio-temporal bounding box (in the WHERE clause) specifies the spatial and temporal coordinates for the data of interest. The spatio-temporal resolution (or output discretization level) 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 (in the FROM clause) specifies 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, finally, the compositing method (in the FROM clause) defines the aggregation level and function to be employed to *coalesce* multiple input grid points into a single output grid point. Two sample Kronos queries specified in PostgreSQL are illustrated in Figure 4. Query 1, for instance, selects the raw AVHRR data from a data collection named AVHRR_DC for the spatio-temporal boundaries stated in the WHERE clause (within the boundaries for latitude, longitude, and day). The data is subsampled in such a way that each output pixel represents 4 KM^2 of data (with the discretization levels defined by *deltalat*, *deltalon* and *deltaday*). Pixels are also corrected for atmospheric distortions using the *WaterVapor* method and composited to find the maximum value of Normalized Difference Vegetation Index (*MaxNDVI*).

Figure 4 presents an overview of the optimization process. The goal is to detect commonalities between Query 1 and 2 in terms of common spatio-temporal domains and the primitives they require. In order to achieve this goal, the first step in the optimization process is to parse and convert these queries into imperative loops conforming with the loop in Figure 1. That loop presents the high-level description of the same queries, with the spatio-temporal boundaries translated into input data points (via index lookup operations). In data analysis applications, the raw data is typically declustered across multiple disks as varying-size chunks, to permit efficient I/O scheduling to fetch the data necessary to process user queries. Therefore, loops can iterate on points, blocks, or chunks depending on how the raw data is stored, declustered, and indexed.

We should note that we have omitted the calls to the subscript mapping functions in order to simplify the presentation. These functions enable both finding an input data element in the input dataset and determining where it is placed in the output dataset (or temporary dataset). In some cases, mapping from an absolute set of multidimensional coordinates (given in the WHERE clause of the query) into a relative set of coordinates (the locations of the data elements) may take a considerable amount of time. Thus, minimizing the number of calls to the mapping operations can also improve performance.

As seen in Figure 4, once the loops have been generated, the following steps are carried out to transform them into a global query plan:

1. The imperative descriptions are concatenated into a single ***workload program***.

2. The domains for each of the *foreach* loops are inspected for multidimensional overlaps (i.e., multidimensional bounding box intersections are computed). Loops with domains that overlap are fused by moving the individual loop bodies into one or more combined loops. Loops corresponding to the non-overlapping

Figure 4. An overview of the entire optimization process for two queries. *MaxNDVI* **and** *MinCh1* **are different compositing methods and** *Water Vapor* **designates an atmospheric correction algorithm. All temporaries have local scope with respect to the loop. The discretization values are not shown as part of the loop iteration domains for a more clear presentation.**

domain regions are also created. An intermediate *program* is generated with two parts: combined loops for the overlapping areas and individual loops for the non-overlapping areas.

3. For each combined loop, common subexpression elimination and dead code elimination techniques are employed. That is, redundant RHS function calls are eliminated, redundant subscript function calls are deleted, and duplicated retrieval of input data elements is eliminated.

In Section 3.3 we will discuss in greater details how each of these steps are performed and how the optimization process is integrated into our database middleware.

3 System Support

In this section, we describe the runtime system that supports the multi-query optimization phases presented in Section 2. The runtime system is built on a database engine we have specifically developed for efficiently executing multi-query loads from scientific data analysis applications in parallel and distributed environments [2, 3]. The compiler approach described in this work has been implemented as a front-end to the Query Server component of the database engine.

The Query Server is responsible for receiving queries from the clients, generating a query plan, and dispatching them for execution. It invokes the Query Planner every time a new query is received for processing, and continually computes the best query plan for the queries in the waiting queue which essentially form a query batch. When resources (e.g., processors) become available, the optimized set of loops for the query batch is dispatched for execution. The system support and process for computing the best plan is described in the next sections.

3.1 The Declarative Query Front-End

A declarative language is one that can express what the desired result of a query should be, without explaining exactly how the desired result is to be computed. Previous researchers have already postulated and verified the strengths of using declarative languages from the perspective of end-users, essentially because the process of accessing the data and generating the data product does not need to be specified, as is the case with procedural or imperative languages [35]. SQL-86 (also called SQL-1) or SQL-92 (also called SQL-2) are traditional declarative languages for database applications. However, the very nature of scientific data analysis applications requires handling datasets and generating data products that are application-specific and user-defined, which makes it cumbersome and, in many cases, impossible to pose a query using these languages.

Stonebraker and Brown [32] describe a simple, four-category taxonomy for classifying DBMS (Database Management System) applications. A key distinction made by the taxonomy is concerned with the complexity of the data, meaning both the raw data and the processed data product that is generated as a result of a query. Specifically, they point out the intrinsic semantic shortcomings of declarative languages traditionally associated with RDBMSs (Relational Database Management Systems). Fortuitously, however, SQL-3 and object-relational DBMSs (ORDBMSs) address the basic impediments to dealing with application-specific data processing [13].

Given the limitations of SQL-2, we have employed PostgreSQL as the declarative language of choice for our system. PostgreSQL has language constructs for creating new data types (CREATE TYPE) and new data processing routines, called user-defined functions (CREATE FUNCTION) [26].

The only relevant part of PostgreSQL to our database is its parser, since the other data processing services all are handled within our existing database engine. Therefore, for the prototype used to gather the experimental results in this paper, we extracted the PostgreSQL parsing code and integrated it into our system through an API that can handle the language constructs that are used to describe a query (SELECT statements), catalog management operations (CREATE TABLE and DROP TABLE), user-defined types and functions (CREATE FUNCTION, CREATE TYPE, etc), and data loading operations (INSERT INTO).

3.2 The Imperative Query Back-End

The appeal of employing a declarative language as the query interface for end-users has extensive implications in the design of the database backend. Indeed, Ullman [35] states that "the use of declarative languages implies extensive optimization by the system if an efficient implementation of declaratively-expressed wishes is to be found". There are two issues embedded in that statement – *translation* and *optimization*. The first issue requires the conversion of the declarative query into an imperative description. The second issue lies in how to optimize the imperative description, with the availability of potentially multiple equivalent operators provided by the runtime system (e.g., multiple ways to compute the maximum value for an attribute for a set of tuples) given the data characteristics (e.g, the dataset is sorted on that particular attribute).

In this paper, the complexity of the translation phase is somewhat constrained for two main reasons. First, the spatio-temporal nature of the applications we support makes the canonical **for** loop in Figure 1 a solution to virtually any query posed to the system, since one needs to visit all the data points pertaining to the query bounding box. This fact implies that there is a one-to-one mapping between a declarative query and an imperative query (note that this is not true for traditional relational queries, in which there may be multiple instances of an operator like *join*, for example). Second, the canonical **for** loop, despite being an imperative description, is at a high enough level of abstraction that it is a unique description of the computation to be performed, but still lends itself to different execution strategies.

3.3 The Multi-Query Planner

The multi-query planner is the system module that receives an imperative query description from the Query Server and iteratively generates an optimized query plan for the queries received, until the system is ready to process the next query batch.

The loop body of a query may consist of multiple *function primitives* registered in the database catalog. In this work, a function primitive is an application-specific, user-defined, minimal, and indivisible part of the data processing [5]. A primitive consists of a function call that can take multiple parameters, with the restriction that one of them is the input data to be processed and the return value is the processed output value. An important assumption is that the function has no side effects. The function primitives in a query loop form a chain of operations transforming the input data elements into the output data elements. A primitive at level l of a processing chain in the loop body has the dual role of consuming the *temporary dataset* generated by the primitive immediately before (at level $l-1$) and generating the temporary dataset for the primitive immediately after (at level $l+1$).

Figure 4 shows two sample Kronos queries that contain multiple function primitives. In the figure, the spatio-temporal bounding box is described by a pair of 3-dimensional coordinates in the input dataset domain. *Retrieval*, *Correction*, and *Composite* are the user-defined primitives. *I* designates the portion of the input domain (i.e., the raw data) being processed in the current iteration of the *foreach* loop and *T0* and *T1* designate the results of the computation performed by the *Retrieval* and *Correction* primitive calls. *O1* and *O2* designate the output for Query 1 and Query 2, respectively.

Optimization for a query in a query batch occurs in a two-phase process in which the query is first integrated into the current plan and, then, redundancies are eliminated. The integration of a query into the current plan is a recursive process, defined by the spatio-temporal boundaries of the query, describing the loop iteration domain. The details of this process are explained in the next sections.

3.3.1 Loop Fusion

The first stage of the optimization mainly employs the bounding boxes for the new query, as well as the bounding boxes for the set of already optimized loops in the query plan, as shown in Algorithm 3 in Appendix A. The optimization essentially consists of *loop fusion* operations – merging and fusing the bodies of loops representing queries that iterate at least partially over the same domain $\mathcal{R}$. The intuition behind this optimization goes beyond the traditional reasons for performing loop fusion, namely reducing the cost of the loops by combining overheads and exposing more instructions for parallel execution. The main goal of this phase is to expose opportunities for subsequent common subexpression elimination and dead code elimination, substantially reducing the loop strength.

Algorithm 3 performs two distinct tasks when a new loop $newl$ is to be integrated into the current query batch plan. First, the query domain for the new loop is compared against the iteration domains for all the loops already in the query plan. The loop with the largest amount of multidimensional overlap is selected to incorporate the statements from the body of $newl$. The second task is to modify the current plan appropriately, based on three possible scenarios: 1) The new query represented by $newl$ does not overlap with any of the existing loops, so $newl$ is added to the plan as is; 2) The iteration domain for the new loop $newl$ is exactly equal to that of a loop already in the query plan (loop $bestl$). In this case, the body of loop $bestl$ is *merged* with $newl$, i.e., the two loop bodies are combined so that the loop body statements for $newl$ are appended to the existing statements for $bestl$ (the **merge** method in Algorithm 3); 3) The iteration domain for $newl$ is either subsumed by that of $bestl$, or subsumes that of $bestl$, or yet there is a partial overlap between the two iteration domains. This case requires computing several new loops to replace the original $bestl$. The first new loop iterates only on the common, overlapping domain of $newl$ and $bestl$, which is computed by the **commonArea** function in Algorithm 3. Next, $newl$ is merged with $bestl$ (now with updated boundaries) and the resulting loop is added to the query plan (i.e., $bestl$ is replaced by $updatedl$). Second, loops covering the rest of the domain originally covered by $bestl$ are added to the current plan L. The function **complementTiles** computes the appropriate bounding boxes for each of these loops. Finally, the additional loops representing the rest of the domain for $newl$ are computed using the function **additionalTiles**, and the new loops become *candidates* to be added to the updated query plan. They are considered candidates because those loops may also overlap to other loops already in L. The *addNewLoop* function described by Algorithm 3 is recursively invoked for each of those loops. This last step guarantees that there will be no iteration space overlap across the loops in the final query batch plan.

3.3.2 Redundancy Elimination

After the loops for all the queries in the batch are added to the query plan L, redundancies in the loop bodies can be removed employing straightforward peephole optimizations – common subexpression elimination and dead code elimination. In our case, common subexpression elimination consists of identifying computations and data retrieval operations that are performed multiple times in the loop body, eliminating all but the first occurrence [23]. Our method is a variation of the traditional algorithm employed for this type of optimization and consists of keeping track of the *available expressions*, i.e., those that have been computed so far in the loop body. In our case, each of the statements are of the following form:

$$T_i[\mathcal{S}_L(r)] = \mathcal{F}_j(T_i[\mathcal{S}_L(r)], T_j[\mathcal{S}_{R_1}(r)], \ldots, T_k[\mathcal{S}_{R_n}(r)], p_1, \ldots, p_k)$$

where T_i is either a temporary aggregate or a query output aggregate, T_j,. . .,T_k are aggregates computed earlier in the processing chain, and $p_1, \ldots, p_k$ are other primitive-specific parameters. In such a scenario, each statement creates a new available expression (i.e., represented by the right hand side of the assignment), which can be accessed through a reference to the temporary aggregate on the left hand side of the assignment. Algorithm 2 in Appendix A performs detection of new available expressions and substitution of a call to a primitive by a *copy* from the temporary aggregate containing the redundant expression.

The equivalence of the results generated by two statements is determined by the **find** function, which inspects the *call site* for the primitive function invocations. This operation essentially means establishing that in addition to using the same (or equivalent) input data, the parameters for the primitive are also the same or equivalent. Because the primitive invocation is replaced by a copy operation, primitive functions are required to not have any side effects.

The removal of redundant expressions usually causes the creation of useless code – assignments that generate *dead variables* – that is no longer needed to compute the output results of a loop. By definition, a variable is dead if it is not *used* by any of the statements from the location in the loop body where it is *defined* until the last statement in the loop body being inspected [23]. We extended this definition to also accommodate situations in which a statement conforms to the form $T_i \leftarrow \textbf{copy}(T_j)$, where T_i and T_j are both temporaries. In this case, all the uses of T_i can be simply replaced by T_j.

We employ the standard dead code elimination algorithm, which requires marking all instructions that compute essential values. Indeed, our algorithm computes the def-use chain (connections between a *definition* of a variable and all its *uses*) for all the temporaries in the loop body, as shown in Algorithm 1 in Appendix A.

Algorithm 1 makes two passes over the statements that are part of a loop in the query plan. The first pass detects the statements that define a temporary and the ones that use it, updating the du data structure. A second pass over the statements looks for statements that define a temporary value, checking du for whether they are utilized, and removes the unneeded statements.

Both the common subexpression elimination and the dead code elimination algorithms must be invoked multiple times, until the query plan remains stable, meaning that all redundancies and unneeded statements are eliminated. At this point the Query Planner can submit the plan for sequential or parallel execution.

Although considerably similar to standard compiler optimization algorithms, all of the algorithms were implemented in the Query Planner to handle an intermediate code representation we devised to portray the query plan. It should be clear that we are not compiling C or C++ code, but rather the query plan representation. Indeed, the runtime system implements a virtual machine that can take either the unoptimized query plan or the final optimized plan and execute it, leveraging any possibly parallel infrastructure available for that purpose.

4 Experimental Evaluation

The evaluation of the techniques presented in this paper was carried out using the Kronos application (see Section 2.2). It was necessary to re-implement the Kronos primitives to conform to the interfaces of our database system. However, employing a real application ensures a more realistic scenario for obtaining experimental results. On the other hand, we had to employ synthetic workloads to perform a parameter sweep of the optimization space. We utilized a statistical workload model based on how real users tend to interact with the Kronos system.

4.1 A Query Workload Model

We designed several experiments to illustrate the impact of the compiler optimizations on the overall batch processing performance, using AVHRR datasets and a mix of synthetic workloads. All the experiments were run on a 24-processor SunFire 6800 machine with 24GB of main memory running Solaris 2.8. We used a single processor of this machine to execute queries, as our main goal in this paper is to evaluate the impact of the various compiler optimization techniques on the performance of query batches. The leverage from the multi-processor nature of the environment will be investigated in a future work to further decrease query batch execution time.

A dataset containing one month (January 1992) of AVHRR data was used, totaling about 30GB. In order to create the queries that are part of a batch, we employed a variation of the Customer Behavior Model Graph (CBMG) technique. CBMG is utilized, for example, by researchers analyzing performance aspects of e-business applications and website capacity planning [22]. 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.

In our model, the first query in a batch 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). The subsequent queries in the batch are generated based on the following operations: another *new point of interest*, *spatial movement*, *temporal movement*, *resolution increase* or *decrease*, applying a different *correction algorithm*, or applying a different *compositing operator*. In our experiments, we used the probabilities shown in Table 1 to generate multiple queries for a batch with different workload profiles. For each workload profile, we created batches of 2, 4, 8, 16, 24, and 32 queries. A 2-query batch requires processing around 50 MB of input data and a 32-query batch requires around 800 MB, given that there is no redundancy in the queries forming the batch and also that no optimization is performed. There are 16 available points of interest; for example, Southern California, the Chesapeake Bay, the Amazon Forest, etc. This way, depending on the work-

Transition	*Workload 1*	*Workload 2*	*Workload 3*	*Workload 4*
New Point-of-Interest	5%	5%	65%	65%
Spatial Movement	10%	50%	5%	35%
New Resolution	15%	15%	5%	0%
Temporal Movement	5%	5%	5%	0%
New Correction	25%	5%	5%	0%
New Compositing	25%	5%	5%	0%
New Compositing Level	15%	15%	10%	0%

Table 1. Transition probabilities. A sequence of queries in a query batch is generated as follows: the first query always uses *New Point-of-Interest* and subsequent queries are generated based on modifications to the query attributes based on the transition selected using the probabilities in the table.

load profile, subsequent queries after the first one in the batch may either linger around that point (moving around its neighborhood and generating new data products with other types of atmospheric correction and compositing algorithms) or move on to a different point. These transitions are controlled according to the transition probabilities in Table 1. More details about the workload model can be found in [4].

For the results shown in this paper each query returns a data product for a 256×256 pixel window. We have also compiled results for larger queries – 512×512 data products. The results using those queries are completely consistent with the ones we show here. In fact, in absolute terms the improvements are even larger. However, we had to restrict ourselves to smaller batches of up to 16 queries due to the memory footprint exceeding 2 GB (the amount of addressable memory using 32-bit addresses employed when utilizing gcc 2.95.3 in Solaris).

4.2 Experimental Study

We studied the impact of the proposed optimizations varying the following quantities:

- The number of queries in a batch (from a 2-query batch up to a 32-query batch).
- The optimizations that are turned on (none, only common subexpression elimination and loop fusion – CSE-LF; or common subexpression elimination, dead code elimination, and loop fusion – CSE-DCE-LF).
- The workload profile for a batch. Workload 1 represents a profile with high probability of reuse across the queries. In this workload profile, the probability that a query will choose a new point-of-interest and the amount of spatial movement around the point-of-interest are very low (5% and 10%, respectively). This results in high overlap in regions of interest across queries. Moreover, the probabilities of choosing new correction, compositing, and resolution values are low. Workload 4, on the other hand, denotes a profile with the lowest probability of data and computation reuse. All the queries from this workload choose the same resolution, correction, and compositing values, but the amount of spatial movement and the probability of choosing a new point-of-interest for a query is quite high. The other profiles – 2 and 3 – are in between the two extremes as far as the likelihood of data and computation reuse.

Our study measured five different performance metrics: batch execution time, number of statements executed (loop body statements), average query turnaround time[1], average query response time[2], and plan generation time (i.e., the amount of time from when the parser calls the query planner until the time the plan is fully computed).

4.2.1 Batch Execution Time

The amount of time required for processing a batch of queries is our most important metric, since that is the main goal of the optimizations we employ. Figure 5 shows the reduction in execution time for different batches and workload profiles, comparing against executing the batch *without any optimizations*. The results show that reduction in the range of 20% to 70% in execution time is achieved. Greater reductions are observed for larger batches using the workload profile 1, which shows high locality of interest. In this profile, there is a very low chance of selecting a new point of interest or performing spatial movement (which implies high spatial and temporal locality as seen in Table 1). Therefore, once some data is retrieved and computed over, most queries will reuse at least the input data, even if they require different atmospheric correction and compositing

[1] Query turnaround time is the time from when a query is submitted until when it is completed [30, 34].

[2] Query response time is the time between when a query is submitted and the time the first results start being returned [30, 34].

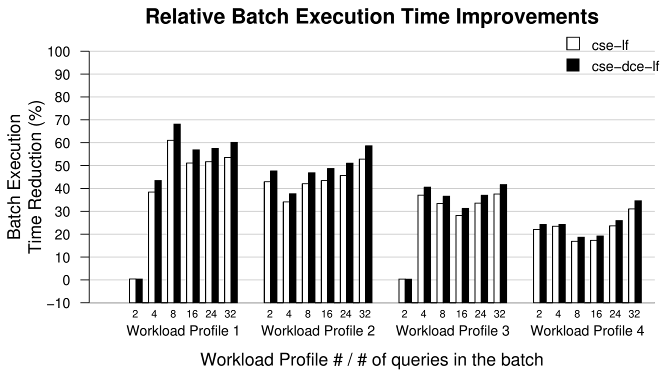

Figure 5. The reduction in batch executing time.

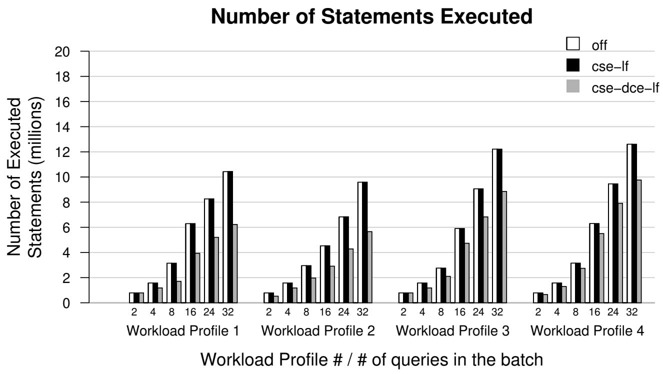

Figure 6. Number of loop body statements executed.

algorithms. Additionally, there are only 16 points of interest as we previously stated, which means that across the 32 queries at least some of the queries will be near the same point of interest, which again implies high locality. On the other hand, when a batch has only 2 queries, the chance of having spatio-temporal locality is small, so the optimizations have little effect. The 2-query batches for workload profiles 1 and 3 show this behavior (note that the y-axis in the chart starts at -10% improvement). In some experiments we observe that the percent reduction in execution time decreases when the number of queries in a batch is increased (e.g., going from a 4-query batch to an 8-query batch for Workload 3). We attribute this to the fact that queries in different batches are generated randomly and independently. Hence, although a workload is designed to have a certain level of locality, it is possible that different batches in the same workload may have different amounts of locality, due to the distribution of queries. The important observation is that the proposed approach takes advantage of locality when it is present. As with any optimization in general, and compiler optimizations in particular, the benefits of optimizations are realized in proportion to the probability of finding a chance to apply them! In our case, that presumes some spatio-temporal locality occurs. This seems to be the case for many data analysis applications, and for most remote sensing applications in particular, because of higher interest in some geographical areas during particular time periods (e.g., wild fire season in Southern California).

4.2.2 Number of Statements and Loop Strength

Figure 6 shows how each combination of optimizations affects the number of loop body statements executed. Loop fusion (LF) alone has no impact whatsoever for this metric. The same is true for common subexpression elimination (CSE), because neither optimization eliminates loop body statements. Interestingly, we observe that CSE alone is responsible for substantially decreasing batch execution time, as seen in Figure 5. The decrease is a function of loop strength reduction, due to the fact that many statements are changed from either an I/O intensive primitive (Retrieval) or a processing intensive primitive (either one of the algorithms used by the Correction and Composite primitives) into a much less expensive *copy* operation. On the other hand, when dead code elimination (DCE) is turned on in conjunction with the other optimizations, a sizable decrease occurs in the number of loop body statements executed, which accounts for a further decrease in the batch execution time as also seen in Figure 5. Indeed, one can see that the results in the two figures correlate, and for some configurations the optimizations do not provide any performance benefits (due to the lack of spatial or temporal locality).

4.2.3 Query Turnaround Time

Queries may be processed in a batch for two main reasons: 1) a client submits a batch, or 2) a batch is formed while the system is busy processing other queries, and interactive clients continue to send new queries that are stored in a waiting queue. In the second scenario, it is also important for a database system to decrease the average execution time per query so that interactive clients experience less delay between submitting a query and seeing its results. Although the optimizations are targeted at improving batch execution time, Figure 7 shows that they also have a positive impact on average query turnaround time. In these experiments, queries are added to the batch as long as the system is busy. The query batch is executed as soon as the system becomes available for processing it. As seen from the figure, for the workload profiles with higher locality (1 and 2), execution

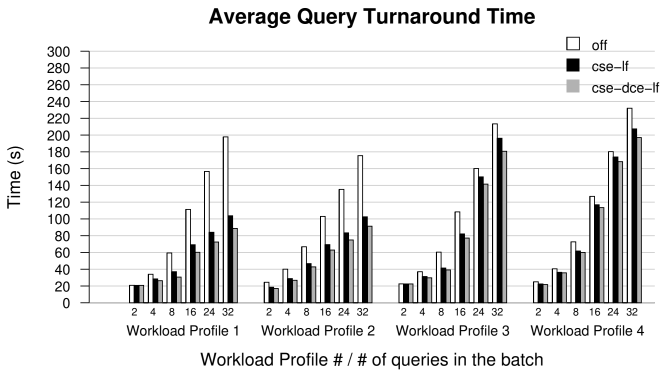

Figure 7. Average query turnaround time.

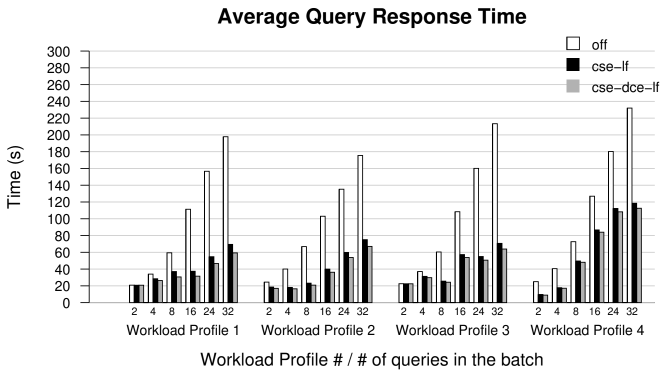

Figure 8. Average time to generate a partial result for a query.

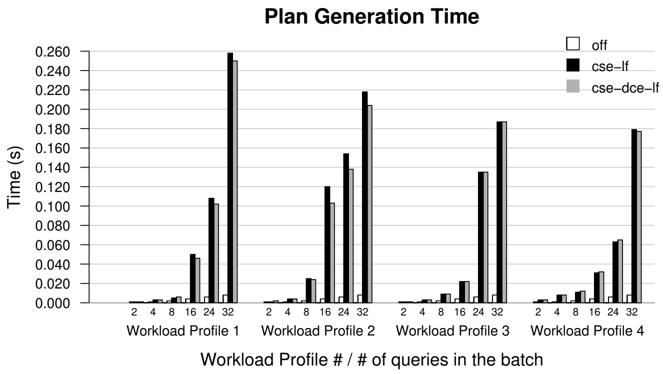

Figure 9. Time to generate a batch executing plan.

time decreases by up to 55%. Conversely, for batches with low locality there is no decrease in execution time, as expected.

4.2.4 First Response Time

Also related to the issue of satisfying interactive clients is the issue of showing *partial* progress during the computation of a query. From the perspective of human-computer interaction, it is usually preferable to have a system that provides an indication of progress during an expensive query [30]. The application of the loop fusion optimization creates multiple loops that generate partial results (e.g., portions of an output image dataset) for a single query. Once a loop is completely computed, the partial result (e.g., a sub-image) is ready for delivery to the client. Figure 8 shows that an added benefit of employing all the compiler optimizations is that the response time for generation of the first partial result also decreases considerably. The decrease can be as much as 70%, but is usually between 20% and 50%.

4.2.5 Plan Generation Time

The application of compiler optimization strategies introduces costs for computing the optimized query plan for the query batch. Figure 9 illustrates how much time is needed to obtain the execution plan for a query batch. There are two key observations here. The first observation is that the planning time depends on the number of exploitable optimization opportunities that exists in the batch (i.e., locality across queries). Hence, if there is no locality in the query batch, the time to generate the optimized plan (which is supposed to be the same as the unoptimized plan) is roughly equivalent to the time to compute the non-optimized plan. The second observation is that the time to compute a plan for batches that have heavily correlated queries increases exponentially (due to the fact that each spatio-temporal overlap detected produces several new loops which must be recursively inserted into the optimized plan). However, even though much more time is spent in computing the plan, executing the query batch is several orders of magnitude more expensive than computing the plan. As seen from Figures 5 and 9, query batch planning takes milliseconds, while query batch execution time can be hundreds of seconds depending on the complexity and size of the data products being computed. Finally, a somewhat surprising observation is the fact that adding dead code elimination to the mix of optimizations actually slightly decreases the time needed to compute the plan. The reason for this is that the loop merging operation and also subsequent common subexpression elimination operations become simpler, if useless statements are removed from the loop body. This extra improvement is doubly beneficial because the time to execute the batch decreases even further as seen in Figures 5, 7, and 8.

5 Conclusions

In this paper, we have described a framework for optimizing the execution of scientific data analysis query batches that employ well-understood compiler optimization strategies. The queries are described using a declarative representation – PostgreSQL – which in itself represents an improvement in how easily queries can be formulated by end users. This representation is transformed into an imperative representation using loops that iterate over a multidimensional spatio-temporal bounding box. That representation lends itself to various compiler optimizations techniques, such as loop fusion, common subexpression elimination, and dead code elimination. Our experimental results using a real application show that the optimization process is inexpensive and that when there is some locality across the queries in a batch, the benefits of the optimizations greatly outweigh the costs. The optimizations can provide sizable decreases in the amount of computation and I/O that are required for executing data and computation intensive queries, as our experimental evidence showed. Even more important is that this approach can very elegantly handle the issue of dealing with user-defined primitives that arise in scientific applications.

We have shown that the compiler optimization strategies can account for sizable decreases in the execution time of queries. Nevertheless, there are many aspects that have not yet been thoroughly investigated in the context of scientific databases. Two important issues we plan to address in the near future are batch scheduling for parallel execution and resource management. Use of loop fusion techniques not only reduces loop overheads, but also exposes more operations for parallel execution and local optimization. In fact, because of the nature of our target queries (i.e., queries involving primitives with no side effects and generalized reduction operations), each statement of the loop body can be carried out in parallel. This means that scheduling the loop iterations in a multithreaded environment or across a cluster of workstations can improve performance, assuming that synchronization and communication issues are appropriately handled. With respect to resource utilization, there are complex issues to be addressed, in particular with regard to memory utilization. When two or more queries are fused into the same loop, all the output buffers for the queries need to be allocated (at least partially) to hold the results produced by the loop iteration. Moreover, those buffers may need to be maintained in memory for a long time, since all the iterations required to complete a query may be spread across a large collection of loops that may be executed over a long time period (i.e., the first and last loop for a query may be widely separated in the batch plan). Indeed, the issue of when and how to schedule the execution of the multiple loops produced by the query planner, along with other resource management issues, are key aspects of the optimization process. Another approach worth exploring is partitioning query batches into sub-batches and then scheduling within a batch to minimize memory footprint, in a way similar to partitioning activities for web agents [25].

In our previous work [5], we have shown that employing an active semantic cache, in which requests for aggregates can be satisfied not only when there is an exact match, but also when a transformation primitive can automatically modify an aggregate to comply with the request, is an important mechanism for improving the performance of a database system dealing with multi-query batches. A natural extension of that is to allow common subexpression elimination to be active in those situations. In that case, a *transformable* equivalent expression is detected and the runtime system invokes the appropriate method to transparently perform the operation. If computing the transformation is cheaper than computing the aggregate from scratch, this approach will clearly outperform simpler common expression elimination operations. This is an extension we want to explore in a future prototype. Along the same lines, the active caching system and the batch optimizer can be integrated. For this scenario, the batch optimizer can also leverage the cache contents when performing common subexpression eliminations.

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] 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.

[3] 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.

[4] 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.

[5] H. Andrade, T. Kurc, A. Sussman, and J. Saltz. Exploiting functional decomposition for efficient parallel processing of multiple data analysis queries. In *Proceedings of the 2003 IEEE International Parallel and Distributed Processing Symposium*, Nice, France, April 2003.

[6] 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.
[7] Caltech. Sensing and responding – Mani Chandy's biologically inspired approach to crisis management. *ENGenious – Caltech Division of Engineering and Applied Sciences*, Winter 2003.
[8] 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.
[9] C. Chang. *Parallel Aggregation on Multi-Dimensional Scientific Datasets*. PhD thesis, Department of Computer Science, University of Maryland, April 2001.
[10] 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.
[11] 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.
[12] J. M. Cheng, N. M. Mattos, D. D. Chamberlin, and L. G. DeMichiel. Extending relational database technology for new applications. *IBM Systems Journal*, 33(2):264–279, 1994.
[13] R. Elmasri and S. B. Navathe. *Fundamentals of Database Systems*. Addison-Wesley, 2000.
[14] R. Ferreira. *Compiler Techniques for Data Parallel Applications Using Very Large Multi-Dimensional Datasets*. PhD thesis, Department of Computer Science, University of Maryland, September 2001.
[15] R. Ferreira, G. Agrawal, R. Jin, and J. Saltz. Compiling data intensive applications with spatial coordinates. In *Proceedings of the 13th International Workshop on Languages and Compilers for Parallel Computing*, pages 339–354, Yorktown Heights, NY, August 2000.
[16] R. Ferreira, G. Agrawal, and J. Saltz. Compiling object-oriented data intensive applications. In *Proceedings of the 2000 International Conference on Supercomputing*, pages 11–21, Santa Fe, NM, May 2000.
[17] R. Ferreira, J. Saltz, and G. Agrawal. Compiler and runtime analysis for efficient communication in data intensive applications. In *Proceedings of the 2001 IEEE International Conference on Parallel Architectures and Compilation Techniques*, pages 231–242, Barcelona, Spain, 2001.
[18] 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.
[19] 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.
[20] 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.
[21] 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.
[22] D. A. Menascé and V. A. F. Almeida. *Scaling for E-Business*. Prentice Hall PTR, 2000.
[23] S. S. Muchnick. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, San Francisco, CA, 1997.
[24] 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.
[25] F. Özcan and V. Subrahmanian. Partitioning activities for agents. In *Proceedings of the 2001 International Joint Conferences on Artificial Intelligence*, Seattle, WA, 2001.
[26] PostgreSQL 7.3.2 Developer's Guide. *http://www.postgresql.org*.
[27] 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.
[28] 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.
[29] T. K. Sellis. Multiple-query optimization. *ACM Transactions on Database Systems*, 13(1):23–52, 1988.
[30] B. Shneiderman. *Designing the User Interface – Strategies for Effective Human-Computer Interaction*. Addison Wesley, Reading, MA, 1998.
[31] M. Stonebraker. The SEQUOIA 2000 project. *Data Engineering*, 16(1):24–28, 1993.
[32] M. Stonebraker and P. Brown. *Object-Relational DBMSs – Tracking the Next Great Wave*. Morgan Kaufmann Publishers, San Francisco, CA, 1999.
[33] K. L. Tan and H. Lu. Workload scheduling for multiple query processing. *Information Processing Letters*, 55(5):251–257, 1995.
[34] A. S. Tanembaum. *Modern Operating Systems*. Prentice Hall, Upper Saddle River, NJ, 2001.
[35] J. D. Ullman. *Database and Knowledge-Base Systems*. Computer Science Press, 1988.

A Optimization Algorithms

This appendix presents the algorithms employed for detecting dead code elimination (Algorithm 1) and common subexpression elimination (Algorithm 2) opportunities and also shows how loop fusion is accomplished (Algorithm 3). These are high-level descriptions and some details are omitted for the sake of providing a better presentation.

```
function commonSubExpressionElimination()
INPUT: query plan (L)
OUTPUT: updated query plan (L')
  for each loop l in L do
    for each statement stmt in l do
      expr ← stmt.getExpression()
      tstmt ← stmt.getSourceStmtForTemp(expr)
      avEx ← availableExpressions.find(expr)
      if avEx then
        stmt.replaceWithResultOf(tstmt)
      else
        availableExpressions.add(expr)
end function
```

Algorithm 2: Replaces every redundant primitive call with a copy statement for all the loops in the query plan.

```
function deadCodeElimination()
INPUT: loop (l)
OUTPUT: updated loop (l')
  for each statement stmt in l do
    if stmt defines a temporary value t then
      temp ← du.find(t)
      if temp ≠ NULL then
        du.add(t)
      else
        if temp.isUsedBy() then
          temp.setIsDefinedBy(stmt)
        else
          deadstmt ← l.whereDefined(temp)
          l.removeStatement(deadstmt)
    if stmt uses a temporary value t then
      temp ← du.find(t)
      temp.setIsUsedBy(stmt)
  for each statement stmt in l do
    if stmt is an assignment of temporaries of the form
    t_j = t_i then
      temp ← du.find(t_j)
      sstmt ← stmt.getSourceStmtForTemp(t_i)
      for each statement tstmt using t_j do
        tstmt ← l.replaceWithResultOf(sstmt)
      l.removeStatement(stmt)
    else if stmt defines a temporary value t then
      temp ← du.find(t)
      if !temp.isUsedBy() then
        deadstmt ← l.whereDefined(temp)
        l.removeStatement(deadstmt)
end function
```

Algorithm 1: Removes all unneeded statements from a loop in the query plan.

```
function addNewLoop(Loop newl)
INPUT: new query loop (newl)
OUTPUT: L – set of fused query loops
  for each loop l in L do
    o ← computeOverlap(l.bb, newl.bb)
    if o < maxoverlap then
      bestl ← l
      maxoverlap ← o
  if maxoverlap = 0 then
    L.add(newl)
  else if maxoverlap = 1 then
    L.remove(bestl);
    updatedl ← newl.merge(bestl)
    L.add(updatedl);
  else
    commonbb ← commonArea(l, bestl)
    bestl.updateBoundaries(commonbb)
    updatedl ← newl.merge(bestl)
    L.remove(bestl)
    L.add(updatedl)
    C ← complementTiles(l, bestl)
    for each tile bounding box t in C do
      l ← new Loop(t, bestl)
      L.add(l)
    A ← additionalTiles(l, bestl)
    for each tile bounding box t in A do
      l ← new Loop(t, newl)
      addNewLoop(l)
end function
```

Algorithm 3: Generates the set of optimized loops for a query batch by recursively integrating a new loop ($newl$) into the existing collection of loops L.