Time and Space Optimization for Processing Groups of Multi-Dimensional Scientific Queries[*]

Suresh Aryangat, Henrique Andrade, Alan Sussman

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

Abstract

Data analysis applications in areas as diverse as remote sensing and telepathology require operating on and processing very large datasets. For such applications to execute efficiently, careful attention must be paid to the storage, retrieval, and manipulation of the datasets. This paper addresses the optimizations performed by a high performance database system that processes groups of data analysis requests for these applications, which we call queries. The system performs end-to-end processing of the requests, formulated as PostgreSQL declarative queries. The queries are converted into imperative descriptions, multiple imperative descriptions are merged into a single execution plan, the plan is optimized to decrease execution time via common compiler optimization techniques, and, finally, the plan is optimized to decrease memory consumption. The last two steps effectively reduce both the time and space to execute query groups, as shown in the experimental results.

1 Introduction

Many applications are emerging that process very large multi-dimensional datasets. One example of such an application is Kronos, which is used by earth scientists to process satellite images of the Earth. Another example is the Virtual Microscope, which provides realistic digital emulation of a high power light microscope. Many similar data analysis applications display a common processing structure [2]. We have developed a scientific database system that exploits this common processing structure and performs various optimizations geared towards reducing both the time to process a single data analysis request (a *query*) and to improve database throughput. Queries from these applications are not only expensive to compute, consuming large amounts of I/O and computational resources, but also have very high memory utilization requirements and often require executing user-defined operations that are not easily implemented in commercial relational databases. In previous work, we have leveraged data and computation reuse for queries individually submitted to the system over an extended period of time. However, for a set of queries considered as a single *group*, a global query plan that accommodates all the queries can often be much more efficient than executing each query separately, assuming that we can devise methods for efficiently removing redundancies across queries, while minimizing the use of memory resources.

The system accepts a declarative data analysis query, specified in PostgreSQL, for a group of queries, and converts it to an imperative form. The resulting program contains one or more loops over multidimensional ranges of some subset of the dataset attributes (e.g., latitude and longitude for remote sensing). Once in imperative form, the system performs query plan transformations based on algorithms commonly used by compilers to optimize the intermediate or low-level representations of program source code. After performing these *execution time reducing* transformations, each original query is converted into a sequence of loops iterating on subsets of the original attribute range(s). Because they came from a declarative query, the loops have the property that changing their execution order does not change the correctness of the results. The system is therefore allowed to reorder the execution of the loops to also optimize *memory usage*. Reducing memory utilization is important because it affects the performance of the system (e.g., by avoiding paging), and may also affect the performance of other applications sharing the same processor and physical memory. In this paper, we describe the time and space optimization techniques we have designed

[*]This research was supported by the National Science Foundation under Grants #EIA-0121161, #ACI-9619020 (UC Subcontract #10152408), and #ACI-9982087, Lawrence Livermore National Laboratory under Grant #B517095, and NASA under Grants #NAG5-11994 and #NAG5-12652.

and implemented for groups of scientific queries, and show experimental results that quantify the benefits of the optimizations.

2 Related Work

Many researchers have been working on support for scientific databases. SEQUOIA 2000 [25] is one of the pioneering projects in that arena. Commercial projects [10] have also identified database requirements that are specific to scientific applications. Optimizing the execution of query groups in relational databases has also attracted interest [18]. Optimizing query processing for scientific applications using database technology has been the goal of several researchers. Ferreira et al. [11, 12, 13, 14] have done extensive research on using compiler and runtime analysis to speed up the processing of *individual* scientific, data-intensive queries. In particular, they have investigated compiler optimization issues related to queries with spatio-temporal predicates, similar to the ones we target [12].

Multi-query optimization has been investigated by several researchers, mainly in relational databases [6, 9, 23, 24, 26]. Similarly, in our previous work, we have devised optimization techniques that allow efficient handling of multi-query workloads when user-defined operations are also part of the query plan [2, 4]. For optimizing a group 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 individual query plans is exposed. A similar observation was the motivation for a study done by Kang et al. [17] for relational operators, and also motivated our earlier work on scientific data analysis queries [1].

3 Database Architectural Overview

The database architecture we have been developing enables efficient handling of multi-query workloads where **user-defined** operations are also part of the query plan [2, 3]. The architecture builds on a data and computation reuse model that can be employed to systematically expose reuse sites in the query plan when these user-defined operations are part of the queries.

In dealing with scientific data, the need to handle query groups arises in many situations. In a data server concurrently accessed by many clients, there can be multiple queries awaiting execution or clients submitting multiple queries simultaneously. In such a scenario, an optimized plan for executing the group of queries can result in better resource allocation and scheduling decisions. The type of queries we address are denoted as range-aggregation queries (RAGs) [7]. 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. Borrowing from a formalism proposed by Ferreira [11], a RAG query can be specified in the general loop format shown in Figure 1. A *Select* function identifies the sub-domain that intersects the query metadata (bounding box) M_i for a query q_i. The sub-domain can be defined in the input attribute space (dataset(s) to be processed) or the output space (data product to be generated). For simplicity, we view the input and output datasets as being composed of collections of multidimensional 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 called *right hand side*, or RHS, objects.

```
$\mathcal{R} \leftarrow Select(I, O, M_i)$
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 Processing Loop.

During query processing, the domain 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 Figure 1, we assume that there are n RHS collections of objects, denoted by $I_1, \ldots, I_n$, contributing to the computation for a LHS object. All n RHS collections do not have to 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 and user-defined 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* [15], 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 results can be computed for each subset and a new intermediate result or the final output can be generated by combining them.

We have employed PostgreSQL [22] as the declarative query language used by clients to formulate queries. PostgreSQL has language constructs for creating new data types

(CREATE TYPE) and new data processing routines, called user-defined operations (CREATE FUNCTION). The only part of the standard PostgreSQL implementation that we employ is its parser, since all other data processing services are handled within our database engine.

3.1 Kronos Queries

Kronos is an example of a remote sensing application [16], and provides tools for geographical, meteorological, and environmental studies. Advanced sensors attached to earth-orbiting satellites collect measurements, from which a dynamic view of the planet's surface can be produced [16]. The raw data gathered by satellite sensors can be post-processed to carry out studies ranging from monitoring land cover dynamics to estimating biomass and crop yield. Systems processing remotely sensed data often wish to provide on-demand access to raw data and user-specified data product generation [8]. Kronos processes datasets composed of remotely sensed AVHRR GAC level 1B (Advanced Very High Resolution Radiometer – Global Area Coverage) orbit data [21]. The volume of data accumulated per day is about 1 GB. An AVHRR GAC dataset consists of a set of Instantaneous Field of View (IFOV) records organized into 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. Data quality indicators are also stored with the raw data. Kronos has been ported to our database system, and Figure 2 shows three sample PostgreSQL Kronos queries (in the black box) that specify multiple **user-defined** operations. In the figure, the spatio-temporal bounding box and spatial-temporal resolution are described by a pair of three-dimensional coordinates (latitude, longitude, time) in the input dataset domain. The user-defined operations are Retrieval (for accessing the raw datasets), Correction (for cleaning up the raw sensor measurements to remove atmospheric effects), Composite (for aggregating the data), and Project (for cartographically projecting out the data product).

4 Time Optimization Techniques

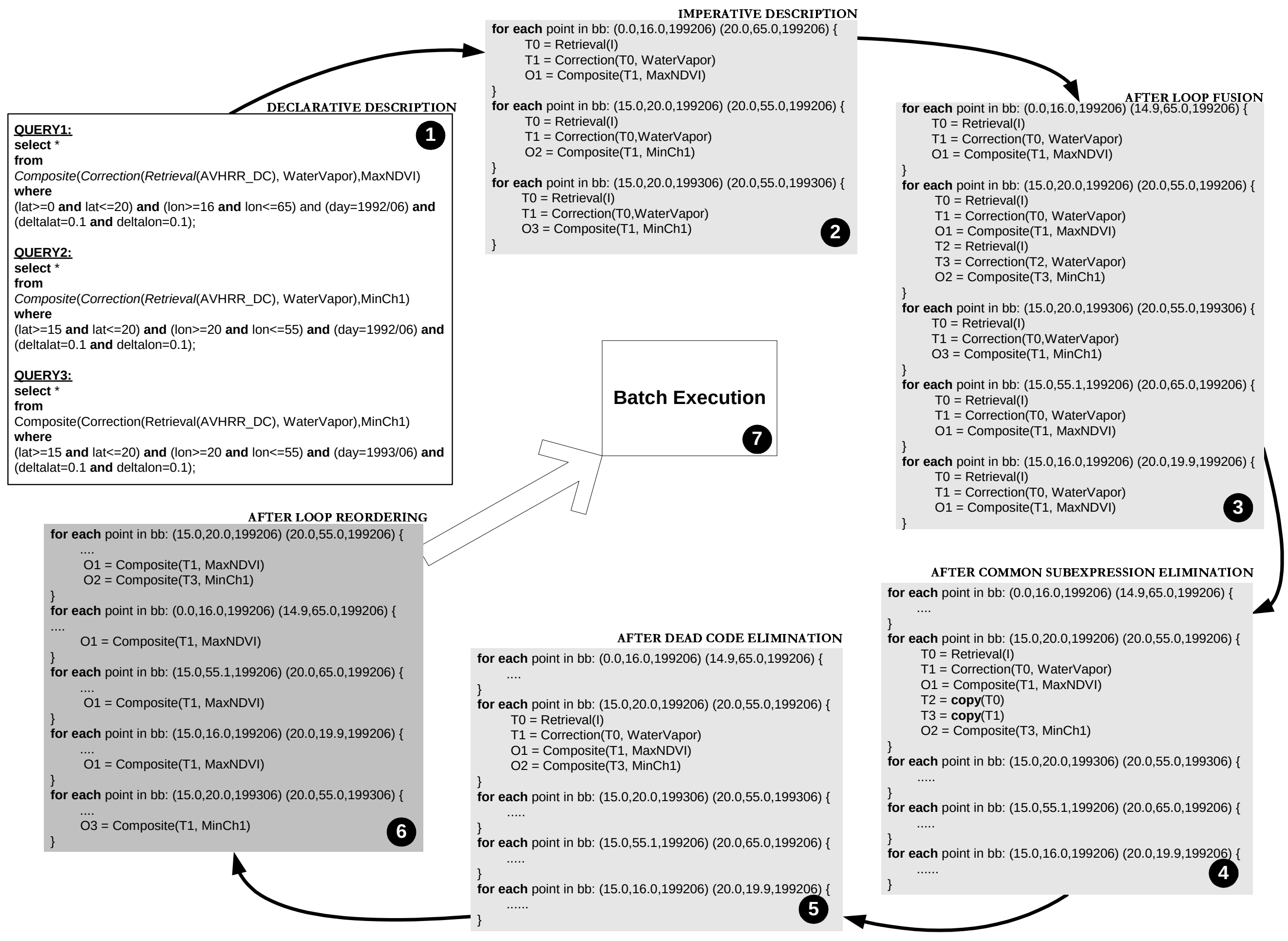
Time optimization techniques consist of methods to reduce the time required to execute a query group submitted to the database by removing redundant I/O operations and/or redundant computations. An overview of our approach is depicted in steps 2 to 5 in Figure 2.

After transforming the declarative queries into multiple *foreach* loops, the first stage of the optimization employs the bounding boxes for each query in the group to perform **loop fusion/splitting** operations – merging and fusing the bodies of loops representing queries that iterate at least partially over the same domain $\mathcal{R}$ (e.g., loops for the first two queries in step 2 of Figure 2 partially overlap. In step 3, we see a loop for the overlapping region and three loops for the non-overlapping regions). The goal of this phase is to expose opportunities for subsequent common subexpression elimination and dead code elimination, as it merely reorganizes the loops. Two distinct tasks are performed when a new loop is incrementally integrated into the current query group plan. First, the bounding box 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 the new loop (i.e, the new query does not overlap with any of the existing loops, or the iteration domain for the new query is either partially/totally subsumed by or partially/totally subsumes that of one of the loops already in the query plan). Additional loops may be generated and the process is recursively repeated for them.

In the second stage, after the loops for all the queries in the group are added to the query plan, redundancies in the loop bodies can be removed, employing straightforward optimizations – common subexpression elimination and dead code elimination. 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. The other occurrences become simple (and cheaper) **copy** operations as seen in step 4 of Figure 2. Finally, the removal of redundant expressions often causes the creation of useless code – assignments that generate *dead variables* that are no longer needed to compute the output results of a loop (e.g., T2 and T3 are removed during step 5 of Figure 2).

Although similar to standard compiler optimization algorithms, all of the algorithms were implemented to handle an intermediate code representation we devised to represent the query plan. We emphasize that we are not compiling C or C++ code, but rather the query plan representation.

Experimental analysis of techniques for reducing group execution time show that substantial decreases are indeed observed (see Section 6). However, a side effect of these techniques is that loops that are part of a single query may not be sequentially executed. For example, in step 5 of Figure 2 the first, second, fourth and fifth loops are part of processing Query 1. During the execution of the third loop, Query 1 buffers are allocated but not being used as Query 3 is being computed. Unless loops are reordered, the implication is that either more memory (in comparison to a non-optimized plan) will be utilized for keeping the query buffers available or more bookkeeping will be required to keep track of the buffers for partially completed queries as they are swapped to and from disk. The issue is further complicated because when a query group is large, there are



```
QUERY1:
select *
from
Composite(Correction(Retrieval(AVHRR_DC), WaterVapor),MaxNDVI)
where
(lat>=0 and lat<=20) and (lon>=16 and lon<=65) and (day=1992/06) and
(deltalat=0.1 and deltalon=0.1);

QUERY2:
select *
from
Composite(Correction(Retrieval(AVHRR_DC), WaterVapor),MinCh1)
where
(lat>=15 and lat<=20) and (lon>=20 and lon<=55) and (day=1992/06) and
(deltalat=0.1 and deltalon=0.1);

QUERY3:
select *
from
Composite(Correction(Retrieval(AVHRR_DC), WaterVapor),MinCh1)
where
(lat>=15 and lat<=20) and (lon>=20 and lon<=55) and (day=1993/06) and
(deltalat=0.1 and deltalon=0.1);
```

```
for each point in bb: (0.0,16.0,199206) (20.0,65.0,199206) {
        T0 = Retrieval(I)
        T1 = Correction(T0, WaterVapor)
        O1 = Composite(T1, MaxNDVI)
}
for each point in bb: (15.0,20.0,199206) (20.0,55.0,199206) {
        T0 = Retrieval(I)
        T1 = Correction(T0,WaterVapor)
        O2 = Composite(T1, MinCh1)
}
for each point in bb: (15.0,20.0,199306) (20.0,55.0,199306) {
        T0 = Retrieval(I)
        T1 = Correction(T0,WaterVapor)
        O3 = Composite(T1, MinCh1)
}
```

```
for each point in bb: (0.0,16.0,199206) (14.9,65.0,199206) {
        T0 = Retrieval(I)
        T1 = Correction(T0, WaterVapor)
        O1 = Composite(T1, MaxNDVI)
}
for each point in bb: (15.0,20.0,199206) (20.0,55.0,199206) {
        T0 = Retrieval(I)
        T1 = Correction(T0, WaterVapor)
        O1 = Composite(T1, MaxNDVI)
        T2 = Retrieval(I)
        T3 = Correction(T2, WaterVapor)
        O2 = Composite(T3, MinCh1)
}
for each point in bb: (15.0,20.0,199306) (20.0,55.0,199306) {
        T0 = Retrieval(I)
        T1 = Correction(T0,WaterVapor)
        O3 = Composite(T1, MinCh1)
}
for each point in bb: (15.0,55.1,199206) (20.0,65.0,199206) {
        T0 = Retrieval(I)
        T1 = Correction(T0, WaterVapor)
        O1 = Composite(T1, MaxNDVI)
}
for each point in bb: (15.0,16.0,199206) (20.0,19.9,199206) {
        T0 = Retrieval(I)
        T1 = Correction(T0, WaterVapor)
        O1 = Composite(T1, MaxNDVI)
}
```

```
for each point in bb: (0.0,16.0,199206) (14.9,65.0,199206) {
        ....
}
for each point in bb: (15.0,20.0,199206) (20.0,55.0,199206) {
        T0 = Retrieval(I)
        T1 = Correction(T0, WaterVapor)
        O1 = Composite(T1, MaxNDVI)
        T2 = copy(T0)
        T3 = copy(T1)
        O2 = Composite(T3, MinCh1)
}
for each point in bb: (15.0,20.0,199306) (20.0,55.0,199306) {
        .....
}
for each point in bb: (15.0,55.1,199206) (20.0,65.0,199206) {
        .....
}
for each point in bb: (15.0,16.0,199206) (20.0,19.9,199206) {
        ......
}
```

```
for each point in bb: (0.0,16.0,199206) (14.9,65.0,199206) {
        ....
}
for each point in bb: (15.0,20.0,199206) (20.0,55.0,199206) {
        T0 = Retrieval(I)
        T1 = Correction(T0, WaterVapor)
        O1 = Composite(T1, MaxNDVI)
        O2 = Composite(T3, MinCh1)
}
for each point in bb: (15.0,20.0,199306) (20.0,55.0,199306) {
        .....
}
for each point in bb: (15.0,55.1,199206) (20.0,65.0,199206) {
        .....
}
for each point in bb: (15.0,16.0,199206) (20.0,19.9,199206) {
        ......
}
```

```
for each point in bb: (15.0,20.0,199206) (20.0,55.0,199206) {
        ....
        O1 = Composite(T1, MaxNDVI)
        O2 = Composite(T3, MinCh1)
}
for each point in bb: (0.0,16.0,199206) (14.9,65.0,199206) {
....
        O1 = Composite(T1, MaxNDVI)
}
for each point in bb: (15.0,55.1,199206) (20.0,65.0,199206) {
        ....
        O1 = Composite(T1, MaxNDVI)
}
for each point in bb: (15.0,16.0,199206) (20.0,19.9,199206) {
        ....
        O1 = Composite(T1, MaxNDVI)
}
for each point in bb: (15.0,20.0,199306) (20.0,55.0,199306) {
        ....
        O3 = Composite(T1, MinCh1)
}
```

Figure 2. Simplified overview of the optimization process for three sample Kronos queries. From 1 to 2, PostgreSQL queries are transformed from a declarative representation to an imperative one; 3 to 5 are execution time decreasing steps; 6 is a memory usage decreasing step; and 7 is the group execution. *MaxNDVI* **and** *MinCh1* **are different compositing methods and** *Water Vapor* **designates an atmospheric correction algorithm.**

many possible loop orderings (and loop interdependencies), so obtaining the *best* ordering is a computationally expensive problem. In the next section, we show how we can minimize this deleterious side effect to obtain the better execution order shown in step 6 of Figure 2.

5 Space Optimization Techniques

Space optimization consists of methods employed to decrease memory utilization, by avoiding keeping in memory query buffers that are not being manipulated by a given loop. The optimization essentially consists of decreasing the *live ranges* [20] of query buffers by reordering loops in the query plan[1] as seen in step 6 of Figure 2.

The loops resulting from applying the method described in Section 4 can be executed in any order to produce correct results, because the user-defined operations are generalized reductions. Each of the possible orderings can be evaluated with respect to two metrics, *maximum memory usage* (MMU) and *average memory usage* (AMU). MMU captures spikes in memory utilization, while AMU captures the amount of memory being utilized *continuously* during the group processing. The order of execution of the loops affects the allocation and deallocation of query buffers and, hence, MMU and AMU. The loops require the use of one or

[1]We refer to the ordering generated by the time optimization techniques described in Section 4 as the **default** order.

Loop	*Used Variables*	*Bitmap*
1	$L_1 = \{v_1, v_2\}$	1 1 0
2	$L_2 = \{v_1, v_2\}$	1 1 0
3	$L_3 = \{v_3\}$	0 0 1
4	$L_4 = \{v_1, v_2\}$	1 1 0

Table 1. Mapping loops in a query plan to bitmaps.

more query buffers to hold intermediate aggregates and the final results for the queries in the group. The query planner uses both *temporary* (the T variables in Figure 2) and *output* (the O variables in Figure 2) query buffers. Output query buffers are used to store the final results of queries to be sent back to clients. Temporary query buffers are used to store intermediate results. Buffer space for a query is allocated for a given loop if it has not already been allocated by a previous loop (i.e., it has not become live yet). Thus, the first loop that is executed must allocate memory for all of the queries that it partially or completely computes. Buffer space for a query is deallocated after the last loop computing results for that query completes and the data product can be returned to the client.

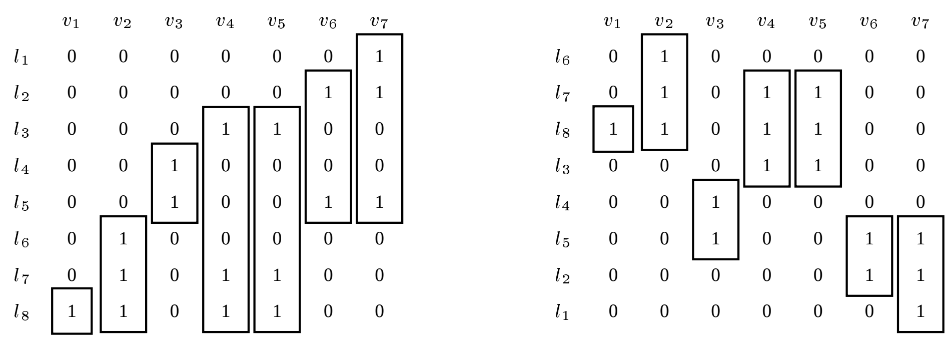
Finding the *best* loop ordering is hard because of interdependencies between loops and queries. Rather than showing a formal definition of the problem (which can be found in [5]), we illustrate it with an example. If there are three queries q_1, q_2, and q_3 and a four loop query plan (i.e., loops $l_1 \ldots l_4$), suppose that the query plan generated after the time optimizations are performed is the following: loop l_1 computes results for q_1 only, l_2 for q_2 only, l_3 for q_3 only, and l_4 computes partial results for all queries. If l_1 is executed first, the live range for q_1 buffers will end when l_4 finishes. If l_4 is executed immediately after l_1, the length of the live range for q_1 buffers is minimized, but the duration of the live ranges for q_2 and q_3 buffers increases. If q_1 buffers are much larger than q_2 and q_3 buffers, or if l_1 can be completed much faster than l_2 or l_3, that reordering is superior to other orderings. This brief discussion highlights that the size of query buffers (or variables), the size of the multi-dimensional loop iteration space (or bounding box), and the cost of executing the loop body (i.e., the cost of executing each of the loop body operations) are all factors in obtaining the best loop ordering.

To determine the order of the loops such that either the MMU or AMU metrics are minimized, every possible loop ordering has to be considered – a "brute force" approach. The order computed by this method always yields the optimal ordering with respect to minimizing MMU or AMU, but requires generating $n!$ orderings, where n is the number of loops in the query plan. A branch-and-bound (**bnb**) algorithm can significantly reduce the amount of time required to compute the optimal order. For some loop sets, it may also be possible to divide the original set of loops into two or more subsets that are independent with respect to the variables they compute, by extracting the connected components from a graph whose vertices represent loops and edges represent variables. In our bnb implementation the input loop set is preprocessed with a connected components analysis and each loop subset resulting from this analysis is submitted separately to the bnb algorithm. Despite such improvements, the running time of this algorithm is still exponential in the number of loops in the worst case (and in most of the cases that we tested experimentally). For this reason, we do not describe the details of these approaches more completely here, but refer the interested reader to [5]. In conclusion, heuristics are clearly necessary for larger sets (i.e., $n \geq 10$) of loops.

5.1 Variable Grouping Heuristics

The variable grouping family of heuristics considers each variable (query buffer) in turn and attempts to contiguously group loops using that variable. In order to achieve this goal, a bitmap representation for each loop in the query plan is employed. For example, suppose the query plan employs four *foreach* loops, which in turn, use three variables. As seen in Table 1, the corresponding bitmap representation requires three bits, one per variable in the query plan. Each loop is represented by its own bitmap, i.e., if the variable is used, its corresponding bit position is set to 1, otherwise the bit position is set to 0. For now, we assume that the first variable v_1 in the query plan corresponds to the leftmost bitmap position, v_2 to the second leftmost position, and so on. The intuition is that once the loops *using* a variable v are grouped together, this variable live range will be decreased as seen for the set of loops in the more elaborate example depicted in Figure 3.

Once the input loop set is converted to a collection of bitmaps, we must first determine the order to process the bit positions (i.e., sort the live ranges by their cost) and then order the bitmaps to produce a loop execution order that minimizes either MMU or AMU. For example, based on the optimized query plan depicted by Figure 3(b), suppose that it is decided that the v_7 related loops will be processed first, i.e., in the order l_5, l_2, and l_1. Obtaining a loop order with the shortest live ranges is straightforward. However, obtaining the order in which to process the bit positions, and hence the variables, of the bitmaps representing the loops is a complex problem. In addition to the exponential number of possibilities, a good solution also depends on several criteria: the number of iterations performed by each loop, the cost of executing the statements inside each loop, the amount of memory associated with each query buffer (vari-



	v_1	v_2	v_3	v_4	v_5	v_6	v_7
l_1	0	0	0	0	0	0	1
l_2	0	0	0	0	0	1	1
l_3	0	0	0	1	1	0	0
l_4	0	0	1	0	0	0	0
l_5	0	0	1	0	0	1	1
l_6	0	1	0	0	0	0	0
l_7	0	1	0	1	1	0	0
l_8	1	1	0	1	1	0	0

(a) Bitmaps for the *original* query plan – default loop order. Boxes denote the live ranges for each variable

	v_1	v_2	v_3	v_4	v_5	v_6	v_7
l_6	0	1	0	0	0	0	0
l_7	0	1	0	1	1	0	0
l_8	1	1	0	1	1	0	0
l_3	0	0	0	1	1	0	0
l_4	0	0	1	0	0	0	0
l_5	0	0	1	0	0	1	1
l_2	0	0	0	0	0	1	1
l_1	0	0	0	0	0	0	1

(b) Bitmaps for the query plan after being optimized for memory utilization

Figure 3. Decreasing query buffers live ranges

able), how many and which variables are processed by a given loop, etc. Indeed, this order is extremely important in determining whether the heuristic produces a memory efficient group execution query plan. Note that fixing the order for processing a variable may affect the order that will be imposed on the processing of the other variables. These multiple criteria give rise to several possible heuristics.

The first ordering method, called the "unused memory potential" method (**vghu**), is based on the observation that the cost to be minimized is the amount of unused memory allocated during the execution of any individual loop. Consequently, the ordering to be chosen is based on the amount of unused memory that a variable can contribute, termed its "unused memory potential". It is defined as the product of the variable's size and the sum of the running times of all loops that do not require that variable to be allocated.

The second method is to prioritize variables based solely on their size (**vghs**). The reasoning behind this method is that the unused memory cost function depends directly on the size of the variables used by the loops and, therefore, grouping loops together according to the size of the shared variable may reduce the amount of unused memory in the resulting loop execution order.

A different approach consists of applying a randomized method, in which a pre-defined number of random orders are evaluated by computing their MMU and/or AMU. The **best** of these orders is chosen for the actual plan. We refer to this variation as **vghr**.

The fourth method is called "deterministic reordering" (**vghd**), which iteratively reorders loops based on *decreasing* values for the unused memory cost function computed for each loop bitmap. After loops have been thus reordered, the heuristic is applied, yielding a new loop execution order. The unused memory cost function for each bit vector is then recomputed, resulting in a new bit position order. This process repeats until AMU or MMU does not decrease for several iterations.

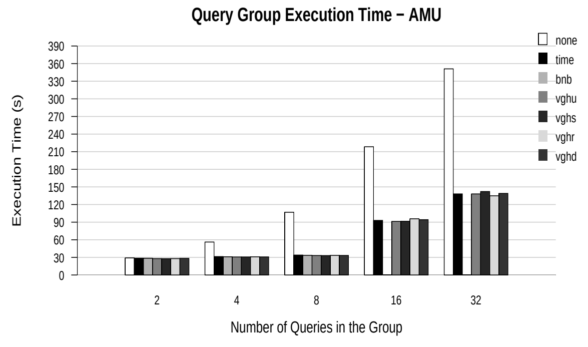
Type of Transition	*Workload W1*	*Workload W2*
New Point-of-Interest	5%	65%
Spatial Movement	10%	35%
New Resolution	15%	0%
Temporal Movement	5%	0%
New Correction	25%	0%
New Compositing	25%	0%
New Compositing Level	15%	0%

Table 2. Transition probabilities – 2 workload profiles.

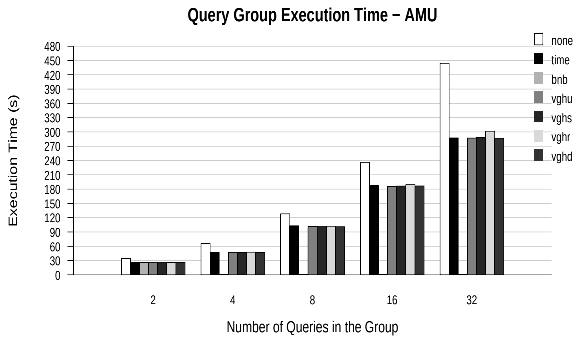
6 Performance Studies

We investigated the performance of the time and memory optimizing methods for actual Kronos queries (generated by a realistic workload model) and also for synthetic loops, which allowed us to explore a larger portion of the optimization space in a controlled fashion.

Our experiments were run on a 24-processor SunFire 6800 machine with 24 GB of main memory running Solaris 2.9. We used a single processor to execute the query groups. A dataset containing one month of AVHRR data was used, totaling about 30 GB. In order to create the queries that are part of a group, we employed a variation of the Customer Behavior Model Graph (CBMG) technique [19]. CBMGs are utilized, for example, to study e-business applications for website capacity planning. A CBMG is characterized by a set of n states, a set of transitions between states, and

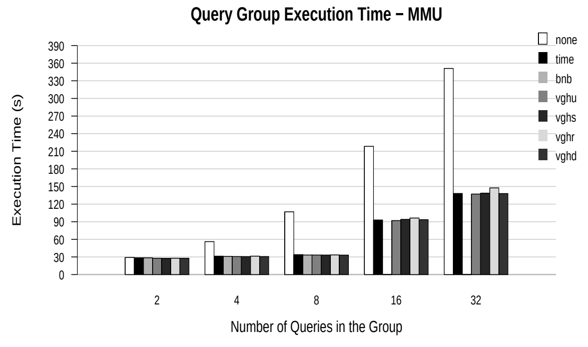


(a) Workload W1

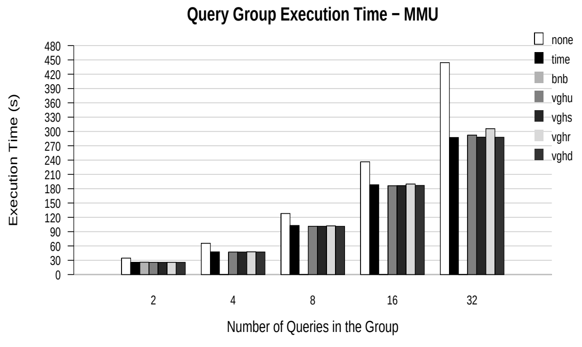


(b) Workload W2

Figure 4. Group execution time when optimizing average memory usage (AMU) for Kronos queries. *None* **indicates no optimizations performed and** *Time* **indicates time optimizations only.**



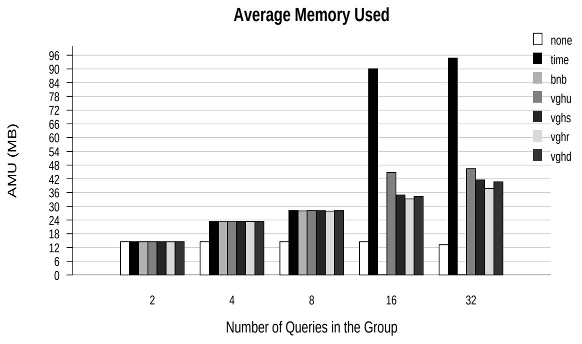
(a) Workload W1



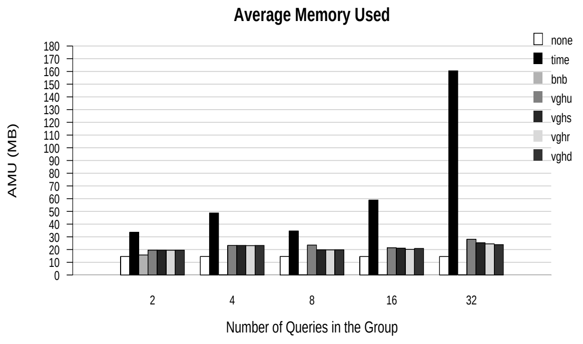
(b) Workload W2

Figure 5. Group execution time when optimizing maximum memory usage (MMU) for Kronos queries. *None* **indicates no optimizations performed and** *Time* **indicates time optimizations only.**

by an $n \times n$ matrix of transition probabilities between the n states. Kronos queries are defined as a 3-tuple: $[$ spatio-temporal bounding box and spatio-temporal resolution, correction method, compositing method $]$. The spatio-temporal bounding box specifies the spatial and temporal coordinates for the data of interest. In our model, the first query (i.e., the initial state) in a group 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). Subsequent queries (i.e., the other states) in the group are generated based on the following operations: a *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 2 to generate multiple queries for two groups with different workload profiles. For each workload profile, we created groups of 2, 4, 8, 16, 24, and 32 queries. An unoptimized 2-query group requires processing around 50 MB of input data and a 32-query group requires around 800 MB, assuming no redundancy in the queries. There are 16 available points of interest, including Southern California, the Chesapeake Bay,

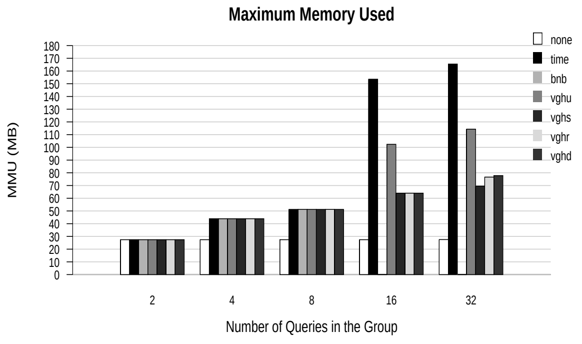


(a) Workload W1

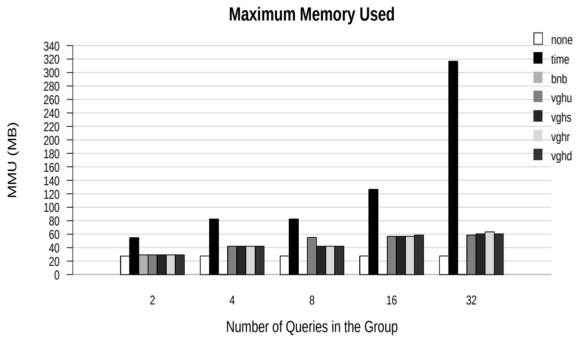


(b) Workload W2

Figure 6. Average memory usage (AMU) for Kronos queries. *None* **indicates no optimizations performed and** *Time* **indicates time optimizations only.**



(a) Workload W1



(b) Workload W2

Figure 7. Maximum memory usage (MMU) for Kronos queries. *None* **indicates no optimizations performed and** *Time* **indicates time optimizations only.**

the Amazon Forest, etc. The workload profile determines whether queries after the first one in the group either remain near that point (moving around its neighborhood and generating new data products, potentially using different atmospheric correction and compositing algorithms) or move to a different point. In the experiments, each query returns a data product for a 256×256 pixel window. We have also produced results for larger queries – 512×512 data products. The results from those queries are consistent with the ones we show here, with the performance improvements even larger in absolute terms. However, for the larger data products we had to restrict the experiments to smaller groups of up to 16 queries, because the memory footprint exceeded 2 GB (the amount of addressable memory using 32-bit addresses available when utilizing gcc 2.95.3 in Solaris).

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

1. The number of queries in a group, from a 2-query group to a 32-query group;
2. Execution time optimizations were either fully turned on or off;
3. The memory optimization that was used – **none**, **bnb**,

vghu, **vghs**, **vghr**, **vghd**;

4. The workload profile for a group.

Workload W1 represents a profile with high probability of reuse across the queries. In this workload profile, there is high overlap in regions of interest across queries. This is a consequence of assigning low probabilities to the New Point-of-Interest and Spatial Movement transitions, as seen in Table 2. Moreover, the probabilities of choosing new correction, compositing, and resolution values are low. Workload W2, on the other hand, describes a profile with the lowest probability of data and computation reuse. The number of loops in the query plans varied from 2 to 70 for Workload W1 and from 3 to 110 for Workload W2.

The amount of time required to execute the queries in the group (including the optimization phases) is shown in Figures 4 and 5. The execution time for the time optimized groups without memory optimization (**time**) is significantly decreased (35% for W2 with 32 queries to 61% for W1 with 32 queries) from the unoptimized group (**none**), and the size of the decrease correlates with the amount of redundancy across the queries in the group. The execution time is not significantly affected by any of the heuristics, compared to the execution time of the time optimized loops. Nor is it significantly affected by whether we are optimizing to decrease MMU or AMU.

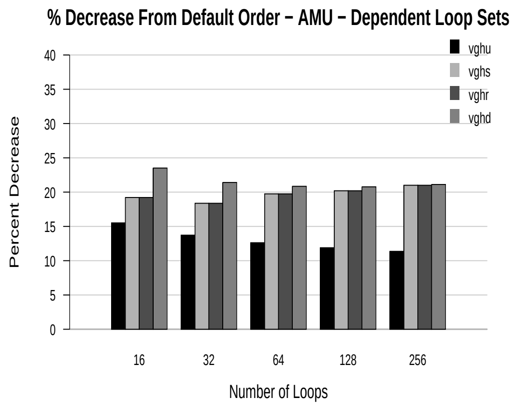
A comparison of average and maximum memory usage among the various memory optimization algorithms, as well as the unoptimized version, is presented in Figures 6 and 7, respectively. The **bnb** algorithm was used to determine the optimal loop ordering in groups only when the number of loops resulting from the time optimizations was 8 or fewer, due to its exponential running time. Thus the bar for **bnb** is omitted for some configurations. The first item to note is that when comparing **none** (i.e., all optimizations are off) vs. **time** (i.e., only time optimizations are on), the increase in the amount of memory utilized is, in most cases, substantial. For example, in Figure 6(b) there is an 11-fold increase in AMU for the 32-query group, as a side effect of having multiple queries active simultaneously, despite the decrease in execution time. That effect shows the importance of reordering the loops. Indeed, for that same configuration, after applying the heuristics AMU is only about twice that of the unoptimized and slower plan. In these experiments, there is no clear pattern in terms of which heuristic is the best, but **vghr** seems to be slightly better for decreasing AMU, while **vghs** seems to be better for decreasing MMU.

Experimental Results With Synthetic Loops

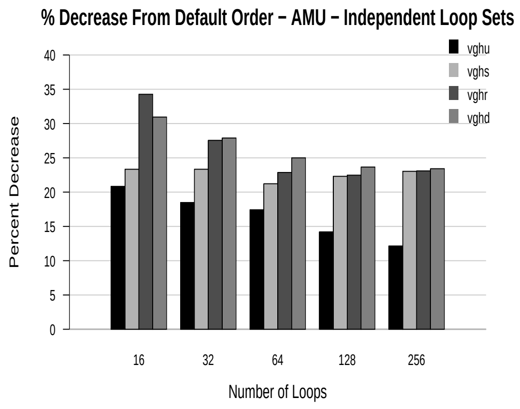
We studied the impact of the number of loops on the performance of each loop ordering heuristic, as well as the effect of loop inter-dependency, using synthetic generated loop sets. The synthetic loops were randomly generated by varying the number of loops (16 to 256), the total number of variables (either 20 – we refer to this configuration as **dependent loop sets**, or 40 – we refer to this configuration as **independent loop sets**), minimum and maximum number of variables per loop (6 to 10), minimum and maximum size of variables (10 to 100), and minimum and maximum running time for a loop (10 to 100). It should be noted that these loops do not correspond to actual queries, but represent possible outcomes from the time optimization phase.

For dependent loop sets, there is a 99% probability of loop interdependency, i.e., multiple loops partially computing a variable. The independent loop sets correspond to an 86% probability of loop interdependency. The performance results are shown in Figures 8 and 9. The optimal values for these metrics could not be obtained because the amount of time required to determine them via the bnb method was too long. For dependent loop sets, the heuristics are able to decrease AMU from that of the default loop order by up to 25% for 16 loops, but are less effective for larger numbers of loops. As was observed for lower numbers of loops, the best performer was the variable grouping heuristic using 100 randomized variable orders (**vghr**), which lowered AMU by at least 20% for all numbers of loops tested. The worst performance was observed for the variable grouping heuristic with the unused variable potential method (**vghu**) which still reduced average memory usage by 11%-15%. For independent loop sets, the heuristics performed significantly better than for dependent loop sets. The **vghr**, and **vghd** heuristics performed similarly, reducing AMU by up to 34%. Again, **vghu** yielded the worst performance but still reduced average memory usage by 12%-20%. The performance results for reduction of MMU show that the heuristics are not effective for large numbers of dependent loop sets. The reduction in maximum memory usage is closely correlated to the number of loops in the input loop set for both dependent and independent loop sets, and decreases markedly as the number of loops increases. For dependent loop sets, the heuristics are able to decrease MMU by 5%-15% for 16 loops. This percentage quickly goes to 0 for larger numbers of loops. For independent loop sets, the heuristics are able to decrease MMU by 14%-28%, but this percentage goes to near 0% for 256 loops.

The running times when optimizing the larger synthetic input loop sets for average and maximum memory usage are given in Figures 10 and 11. The running times for each heuristic appear to be linearly correlated with the number of loops in the input loop set. Optimizing independent loop sets required more time than the dependent loop sets because the number of variables for independent loop sets was twice that of the dependent loop sets. This result indicates that the heuristics are also dependent on the total number of variables used by all of the loops in the input loop set, in addition to the number of loops in this set. There do not appear

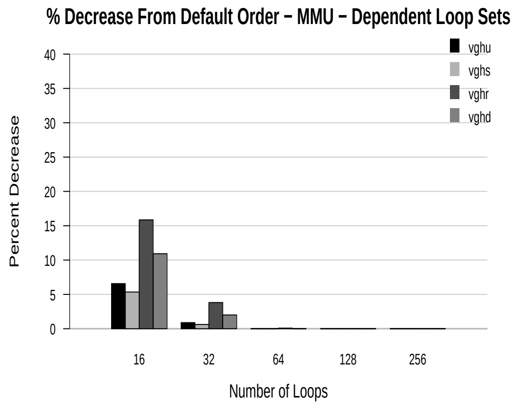


(a) Dependent Loop Sets

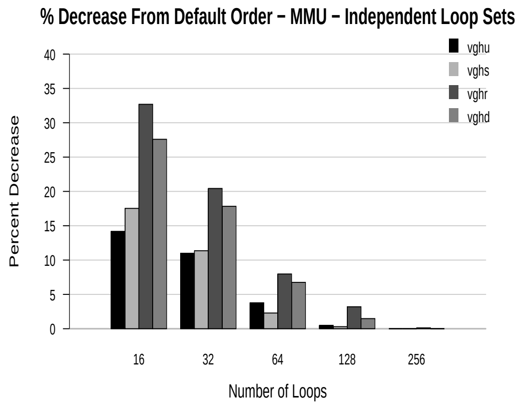


(b) Independent Loop Sets

Figure 8. Percent decrease in average memory usage (AMU) for heuristics applied to the default ordering of (a) dependent loop sets and (b) independent loop sets.



(a) Dependent Loop Sets



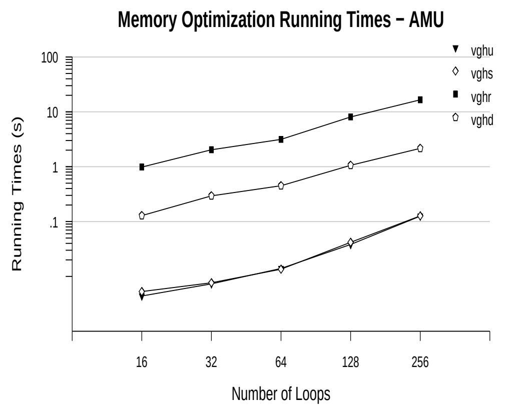
(b) Independent Loop Sets

Figure 9. Percent decrease in maximum memory usage (MMU) for heuristics applied to the default ordering of (a) dependent loop sets and (b) independent loop sets.

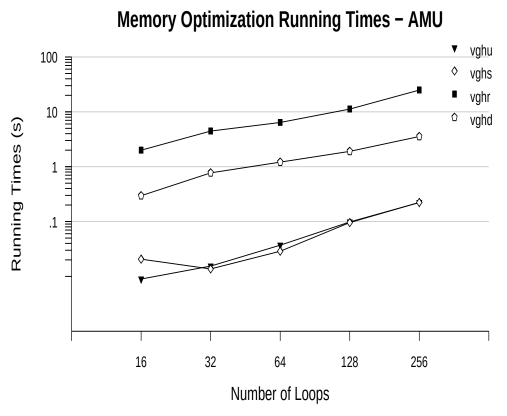
to be significant differences in running times when optimizing for average memory usage versus maximum memory usage. The heuristics with the fastest running times are the variable grouping heuristics using the unused memory potential and variable size methods (**vghu**, **vghs**).

7 Conclusions and Future Work

We have presented various methods that are used to optimize the execution of groups of range-aggregation queries, which are common in scientific data analysis applications. This work makes the following contributions: (1) A method to reduce the time to compute results for query groups. These optimizations are based on algorithms commonly

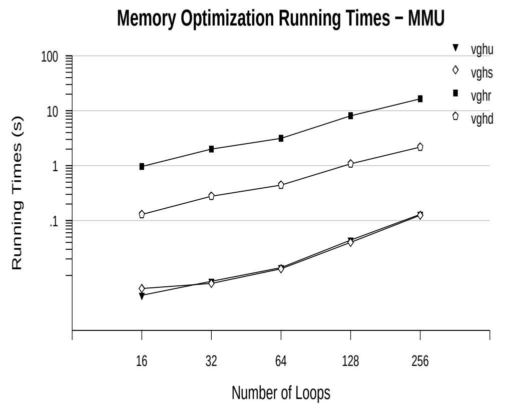


(a) Dependent Loop Sets

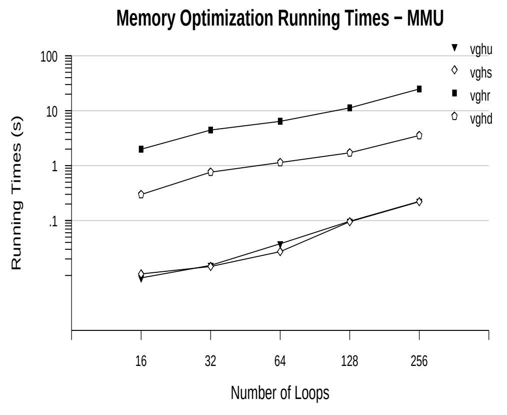


(b) Independent Loop Sets

Figure 10. Execution times in seconds when optimizing for average memory usage (AMU) for (a) dependent loop sets and (b) independent loop sets.



(a) Dependent Loop Sets



(b) Independent Loop Sets

Figure 11. Execution times in seconds when optimizing for maximum memory usage (MMU) for (a) dependent loop sets and (b) independent loop sets.

employed by compilers to reduce the execution time of compiled code; (2) Methods for optimizing the memory footprint for executing a group of queries. These optimizations were shown to improve the total query execution time without significantly impacting the execution time of the query.

This work opens up many opportunities for additional research. It may be possible to improve the variable grouping heuristic further by devising a method to determine the variable bit position order by utilizing the output of previous optimization attempts. In this way, the optimal loop ordering may be determined by iterating over some number of variable bit position re-orderings and optimization attempts. Additionally, a combination of systematic and heuristic procedures can be used to optimize loop ordering, for example by partitioning large query groups with interdependent loops and applying the branch-and-bound method on these smaller groups. Furthermore, optimizations should be designed to optimize memory hierarchy behavior. For example, iteration over datasets can be ordered to improve cache performance by increasing the spatial and temporal locality of data accesses. Executing loops in parallel to make use of parallel machines, both for individual loops and across independent loops, will result in much faster absolute execution times, but requires additional new optimization techniques.

References

[1] H. Andrade, S. Aryangat, T. Kurc, J. Saltz, and A. Sussman. Efficient execution of multi-query data analysis batches using compiler optimization strategies. In *Proceedings of the 16th International Workshop on Languages and Compilers for Parallel Computing (LCPC 2003)*, College Station, TX, October 2003.

[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 SC Conference (SC 2001)*, Denver, CO, November 2001.

[3] 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.

[4] 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 (IPDPS 2003)*, Nice, France, April 2003.

[5] S. Aryangat. Optimizing the execution of data analysis queries. Master's thesis, Department of Computer Science, University of Maryland, December 2003.

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

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

[8] 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 (ICDE 1997)*, 1997.

[9] 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.

[10] 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.

[11] 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.

[12] 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 (LCPC 2000)*, pages 339–354, Yorktown Heights, NY, August 2000.

[13] R. Ferreira, G. Agrawal, and J. Saltz. Compiling object-oriented data intensive applications. In *Proceedings of the 2000 International Conference on Supercomputing (ICS 2000)*, pages 11–21, Santa Fe, NM, May 2000.

[14] 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 (PACT 2001)*, pages 231–242, Barcelona, Spain, September 2001.

[15] 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.

[16] 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.

[17] 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.

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

[19] D. A. Menascé, V. A. F. Almeida, and L. W. Dowdy. *Capacity Planning and Performance Modeling*. Prentice Hall PTR, 2000.

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

[21] 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*.

[22] PostgreSQL 7.3.2 Developer's Guide. *http://www.postgresql.org*.

[23] P. Roy, S. Seshadri, S. Sudarshan, and S. Bhobe. Efficient and extensible algorithms for multi query optimization. In *Proceedings of the 2000 ACM International Conference on Management of Data (SIGMOD 2000)*, pages 249–260, 2000.

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

[25] M. Stonebraker. The SEQUOIA 2000 project. *Data Engineering*, 16(1):24–28, 1993.

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