Exploiting Functional Decomposition for Efficient Parallel Processing of Multiple Data Analysis Queries*

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

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

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

Abstract

Reuse is a powerful method for improving system performance. In this paper, we examine functional decomposition for improving data reuse, and therefore overall query execution performance, in the context of data analysis applications. Additionally, we look at the performance effects of using various projection primitives that make it possible to transform intermediate results generated during the execution of a previous query so that they can be reused by a new query. A satellite data analysis application is used to experimentally show the performance benefits achieved using the techniques presented in the paper.

1 Introduction

Exploiting reuse is a powerful mechanism for improving the performance of computational systems in general [10]. For relational database systems, it has been shown that identifying common subexpressions [16, 19, 36] and applying view materialization strategies [18, 22, 29, 40] can yield sizable decreases in query execution time when processing multiple query batches. When applications that do not conform to the relational database model are targeted, especially in applications where the developer can extend the database by adding application-specific processing capabilities and operators, the multiple query optimization techniques developed for relational databases cannot be applied directly. In this work, we refer to an instance of a data analysis operation as a *query* that processes input data via user-defined operations, generates intermediate results, and, produces an output dataset. Our goal is to propose a mechanism for exposing reuse sites, and also to illustrate methods for employing these reuse opportunities to minimize the cost of processing a set of data analysis queries.

In this paper we investigate inter-query and intra-query performance improvements by breaking the query processing into a chain of *primitive* operations. Two observations are central to our approach. The first one is that in many data analysis applications, a portion (or several portions) of the dataset domain is usually regarded as a *hot spot*, meaning a region (or regions) where most of the interesting phenomena are located. The second observation is that more sophisticated data analysis operations can frequently be defined in terms of simpler primitive operations via what we call *functional decomposition*. Additionally, subsets of these primitive operations can be shared by many higher level operations, and, by extension, different query types. We define functional primitives as the user-defined smallest data processing operations in a particular query execution schema that produce a temporary aggregate as a result.

A software engineer implementing the code for a higher level operation can implement it as a single

*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 Grants #B500288 and #B517095 (UC Subcontract #10184497).

monolithic block, or implement the required primitive operations separately, and compose them to build a more complex operation. From the standpoint of executing a single query, it is usually better to have the data processing executed as a single unit, since execution of smaller operations will require the use of temporary accumulators and extra overhead costs, such as increased memory usage and buffer copies. On the other hand, temporary results can potentially be used by other queries of the same type, and perhaps also by queries of other types, either directly or through *ad hoc* data transformations [6]. We argue that using primitive operations will often lead to improved system performance, since it exposes many potential optimization sites to the query planner. In the following sections, we elaborate on an approach for functionally decomposing data analysis queries to improve data and computation reuse, as well as increasing exploitation of intra-query parallelism. Our approach relies on a framework for the bottom-up construction of query processing chains based on primitive operations. We perform a quantitative case study on a satellite data processing application to evaluate the performance improvements that can be obtained through the functional decomposition of queries for a real data analysis application.

2 Related Work

The bulk of work on multiple query optimization has been done for relational databases [16, 18, 19, 22, 26, 29, 36, 40]. Systems that support data analysis applications subjected to multiple query workloads have not been extensively studied. In this work, we approach the problem of identifying common subexpressions (or temporaries in our terminology) from an algorithmic perspective, in a similar way to that of Kang et al. [26]. Kang et al. restrict their domain to relational operators. In contrast to their work, we cannot decompose queries into high-level, well-defined, and pre-existent primitives that can then be converted into low-level programs (the algorithms). This is because the data analysis applications targeted in this work do not in general lend themselves to being described as a finite set of common primitives. The processing structure of these applications must be defined via extensible application-specific operations. Therefore, we take a bottom-up approach in which the high-level operators responsible for the query processing chain are described in terms of low-level primitives implemented by the application developer. The query processing system uses this information to infer points of reuse. In earlier work [5, 6, 7, 8, 9], we investigated frameworks, query scheduling, and cache replacement issues for data analysis applications. This paper differs from our previous work in that we look at the effect on data reuse of functional decomposition.

There are a number of research projects that focus on component-based models for developing applications in a distributed environment [1, 3, 13, 15, 24, 31, 32, 33, 38]. In these models, the application processing structure is decomposed into a set of interacting computation components. The earlier work on component-based frameworks has focused on improving the performance of a single or a set of independent queries by effectively decomposing the application structure and efficiently scheduling application components. Our work differs from this previous work in that we examine the application of functional decomposition for increasing data reuse opportunities.

3 Case Study Application: Kronos

Remote sensing has become a very powerful tool for geographical, meteorological, and environmental studies. Advanced sensors attached to satellites orbiting the earth collect information, from which a dynamic view of the surface of the planet can be extracted [39, 41]. 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. Kronos [41] is a software system that provides on-demand access to raw data and user-specified data product generation [17, 37]. It has been built on the premise that typical queries dealing with remotely sensed data have a common processing model.

Kronos targets datasets composed of remotely sensed AVHRR GAC level 1B (Advanced Very High Resolution Radiometer Global Area Coverage) orbit data [30]. 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, quality indicators are stored with the raw data.

Several types of queries can be posed to the Kronos system. Queries can be as simple as visualizing the remotely sensed data for a given region using a given cartographic projection [25], or as complex as statistically comparing a *composite* data product across two different time stamps [35]. A typical query specifies a three-dimensional bounding box that covers a region of the surface of the earth over a period of time. A composite image of the selected area is generated from input IFOV records whose coordinates fall into the bounding box. Generating a composite image requires projecting the area onto a two-dimensional grid that represents the final two-dimensional image, and selecting the "best" sensor value that maps to each grid point. A scientist can choose the projection and composition functions that are most suitable for the study. A correction algorithm for the IFOV records may also be part of the query processing, since many different techniques can be employed to eliminate inconsistencies in the raw data due to instrument drift, atmospheric distortion, and topographic effects [20].

4 Generic Data Processing Model

Kronos is an example of a data analysis application. Although it is designed and implemented for a specific type of remotely sensed data, it employs a processing structure that is common in many data analysis applications. This processing structure can be described abstractly, as shown in Figure 1.

(* Datasets *)
$I \leftarrow$ Input
$O \leftarrow$ Output
$A \leftarrow$ Accumulator (Temporary Aggregate)
$M_i \leftarrow$ Query meta-data information
1. $[S_I] \leftarrow Select(I, M_i)$
   (* Initialization *)
2. **foreach** a_e **in** A **do**
3. $\quad a_e \leftarrow Initialize()$
   (* Processing *)
4. **foreach** i_e **in** S_I **do**
5. $\quad$ **read** i_e
6. $\quad S_A \leftarrow Map(i_e)$
7. $\quad A \leftarrow Operation(A, S_A)$
   (* Finalization *)
8. **foreach** a_e **in** A **do**
9. $\quad o_e \leftarrow Output(a_e)$

Figure 1. The query processing loop.

In the figure, *Datasets* are the data and associated meta-data information available to the system for processing a query. The datasets can be classified as *input*, *output*, or *temporary*. **Input** datasets correspond to the data to be processed. **Output** datasets are the result of applying an operation to the input dataset. **Temporary** datasets (temporaries) are created during processing to maintain intermediate results. Often a user-defined data structure, referred to as *accumulator*, is used to describe a temporary dataset. Temporary and output datasets are tagged with the operation employed to produce them and also with the query meta-data information. Temporaries are also referred to as *aggregates*, and we will mostly use the term aggregate in this paper.

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

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

5 Aggregation Operations, Data Reuse, and Functional Decomposition

For data analysis applications, the user-defined *Operation* in Figure 1 often is an aggregation operation. An aggregation operation takes a collection of input tuples $t_1, t_2, \ldots, t_n$ fitting some selection criteria and computes a tuple in the output dataset. For applications like Kronos that deal with spatio-temporal range queries, the selection criteria is usually *temporal* – all tuples that fall in a given time period are aggregated – and/or *spatial/geographically-oriented* – all tuples that fall in a given spatial region (often described via an n-dimensional rectangle) are aggregated.

When multiple queries are submitted to the system, the intermediate and final results (i.e. temporary and output datasets) from an aggregation operation carried out for a query can be cached for reuse by other queries. The desirability of data reuse requires defining the concepts of *usefulness* and *granularity* of an aggregate. The usefulness of an aggregate measures how many queries can reuse the cached result. The more queries can benefit from an aggregate, the more useful an aggregate is. In addition, a fine-grain aggregate likely has much higher reuse potential than a coarse-grain one. For instance, for weather data, maximum yearly precipitation can be computed from collections of the maximum precipitation for 12 months, or 52 weeks, or 365 days. Daily aggregates can be reused for computing weekly and monthly aggregates, whereas a monthly aggregate can only be used for a yearly aggregate. On the other hand, despite its higher potential for reuse, a finer-grain aggregate ordinarily requires more space for storage. For example, a vegetation index for a 100 Km^2 area can be obtained either from ten 10 Km^2 cells or one hundred 1 Km^2 cells. The amount of *effort* put into generating an aggregate varies with the complexity of the operation and also with the number of tuples that are aggregated. Essentially what we are measuring is the amount of computational time and I/O bandwidth used to compute the aggregate. The *utility* metric of an aggregate is the normalized effort (time to compute) with respect to the amount of space required to cache the aggregate (size). This metric is associated with the granularity of the aggregate in the sense that coarser-grain aggregations will have higher utility compared to finer-grain aggregations that occupy the same amount of storage space. A more complete description of these caching issues can be found in [5].

To optimize the execution of multiple queries, a system with limited resources should cache the aggregates with the highest utility. Moreover, the system should implement mechanisms to maximize the usefulness of aggregates both while they are being computed and after they have been cached. In this section, we present two mechanisms for this purpose. *Projection primitives* target intermediate and final results that are already in the cache, while the goal of *functional decomposition* is to divide the execution of a complex aggregation operation into sub-operations so that intermediate results generated by sub-operations can be cached to achieve greater data reuse.

5.1 Projection Primitives for Aggregation Operations

Conventional data caching approaches require a complete and perfect match (i.e. cache hit or miss) between the output to be computed and a previously computed aggregate. We introduce the notion of projection primitives that make it possible to *transform* an aggregate generated for a query so that it can be used to completely or partially satisfy a new query. We call the use of such projection operations *active caching*. With this mechanism, the system has a better chance to exploit reuse than with conventional caching.

Based on our experience with Kronos and other applications [2, 11, 14, 27], we have identified four kinds of projection primitives based on the type of reuse they can leverage: dimensional overlap, composable reduction operations, invertible functions, and inductive functions.

Dimensional (Spatio-temporal) Overlap Primitives: In applications dealing with range queries, commonly one of the clauses in the query meta-data information gives the spatial and/or temporal coordinates of the region over which the query will perform a computation. A spatio-temporal (dimensional) projection

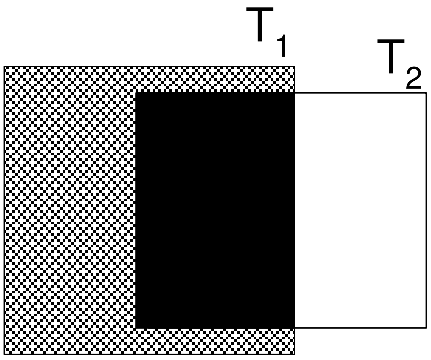


Figure 2. Spatio-temporal projection. The projection primitive retrieves the black rectangle from T_1 in order to partially compute T_2.

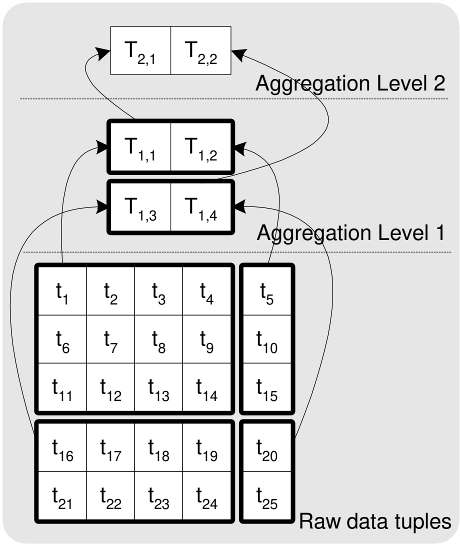


Figure 3. Composable reduction operation. The temporary produced at aggregation level 2 can be computed from temporaries at aggregation level 1, instead of aggregating the raw data tuples.

primitive essentially performs a geometric translation and/or rotation on the cached overlapped data, in addition to clipping the n-dimensional region to conform to the predicate for the new query. Figure 2 illustrates this situation for 2-dimensional data. The new query is completely answered if the cached aggregate completely subsumes the one being computed. Otherwise, the query is partially answered, in which case the query regions that cannot be computed from the cached aggregate must be calculated from the input data or from other cached aggregates.

Composable Reduction Operations Primitives: Aggregation operations that implement *generalized reductions* [23] are commutative and associative. A commutative and associative aggregation operation produces the same output value regardless of the order the input tuples are processed. That is, the set of input data elements can be divided into subsets, temporary datasets can be computed for each subset, and a new intermediate result or the output can be generated by combining the temporary datasets. Essentially, a composable reduction operation primitive takes one or more temporaries with a finer-level aggregation[1] and transforms them into a coarser-level aggregation, coalescing multiple data points into a single new data point (see Figure 3). However, in order for this primitive to generate correct result, a *congruence relationship* should exist between the cached aggregates and the one to be computed.

We define a congruence relationship as follows. Suppose for a set of tuples $t_1, t_2, \ldots, t_n$, a collection of aggregates $T_{l,1}, T_{l,2}, \ldots, T_{l,m}$ is generated using a reduction operation f such that $T_{l,j} = f(t_o, \ldots, t_p)$ and $S_{l,j} = \{t_o, \ldots, t_p\}$. In this case, $T_{x,i}$ can be defined by a projection primitive g over the set of $T_{l,j}$ and x is congruent to l, iff

$$S_{x,i} = \cup S_{l,j} \quad and \quad \cap S_{l,j} = \emptyset$$

Here, x is an aggregation level that is coarser than l. For example, $T_{3,1} = sum(t_1, t_2, t_3, t_4, t_5, t_6)$ is equivalent to $sum(T_{2,1}, T_{2,2})$, if $T_{2,1} = sum(t_1, t_2, t_3, t_4)$ and $T_{2,2} = sum(t_5, t_6)$. On the other hand, if $T_{2,1} = sum(t_1, t_2)$ and $T_{2,2} = sum(t_3, t_4)$, the original equivalence does not hold, because t_5 and t_6 map to neither $T_{2,1}$ nor $T_{2,2}$, meaning that that they are not congruent. In this case, a congruence relationship can be established by computing $T_{2,3} = sum(t_5, t_6)$ using

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

the input data. Similarly, if $T_{2,1} = sum(t_1, t_2, t_3, t_4)$ and $T_{2,2} = sum(t_3, t_5, t_6)$, $T_{3,1}$ cannot be computed from $T_{2,1}$ and $T_{2,2}$ since the tuple t_3 is in both $T_{2,1}$ and $T_{2,2}$. As an example from satellite data processing, the maximum temperature per week cannot be computed from two-day aggregations, since a day from the second week would be included in the fourth two-day aggregate.

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

An inductive aggregation primitive *coarsens* or *refines* an aggregate. In the coarsening process, an already computed later-inductive-step aggregate may be used to compute a prior-inductive-step by removing the contributions of the intermediate inductive steps. An example of inductive operations is performing 3-dimensional volume construction from a set of 2-dimensional images [5, 14]. In order to produce a 3-dimensional volume, an octree data structure is constructed from the 2-dimensional images. An octree of depth $n+1$ can be used to build an octree of depth n without retrieving and processing the input images. Similarly, a lower resolution image in a digitized microscopy application can be produced from a higher resolution image by subsampling [6]. A projection primitive in these cases benefits from the fact that building $f(n)$ from $f(n+1)$ and g is cheaper than generating it from the input dataset. The refining process may require access to the input dataset, but the new aggregate can be computed faster if a previously computed aggregate from a prior inductive step is used, as seen in Figure 4.

Invertible Aggregation Primitives: Some aggregations are computed by applying functions to a single input tuple or to a collection of input tuples. Some of these functions may be algebraically invertible. For example, in satellite data processing, atmospheric correction [20] is often employed to account for atmospheric effects on remotely sensed data, and is performed by applying a function, for example, of the form $f(x) = c_1 + \frac{c_2}{c_3 \times (1 - c_4 \times x)}$. In that case, the value of x can be recovered by computing f^{-1}. For invertible functions, reuse requires extra computation to extract the input value. Therefore, using this primitive requires trading off the cost of retrieving the raw input data versus applying the invertible function to compute the input tuple from a cached aggregate.

Oftentimes combinations of projection primitives can be used to transform a cached aggregate for reuse by another query. For instance, a dimensional projection primitive can be followed by a primitive for either composable reduction operations or inductive aggregation functions to produce a desired aggregate.

5.2 Overlap Functions

Related to the projection of an aggregate is detection of the *overlap* between a cached aggregate and an aggregate required for a given query, with respect to a projection primitive. There is a one-to-one correspondence between each projection primitive and an overlap function. In general, the overlap function returns an index between 0 (no overlap) and 1 (full overlap). For dimensional projections, the overlap function computes a normalized value that measures how much of the desired aggregate can be computed from that projection. This is accomplished by calculating the geometric overlap using two bounding boxes – one for the cached aggregate and one describing the aggregate to be computed. For composable reduction projections, the overlap function returns the *congruence level*, which is defined as the percentage of overlap between the aggregation level of a cached aggregate and the aggregation level to be computed (e.g., an aggregate for days 1 and 2 has a 0.5 overlap with an aggregate for days 1, 2, 3 and 4 from the same year). For inductive functions, the overlap function computes an index in terms of *inductive distance*. The distance is normalized based on the inductive step being searched for (e.g., an image with a resolution of 2 Km^2 per pixel has an overlap of 0.5 with the same image at 4 Km^2 per pixel resolution). Finally, for invertible functions, the overlap function returns whether or not the function inverse can be computed (either 0 or 1).

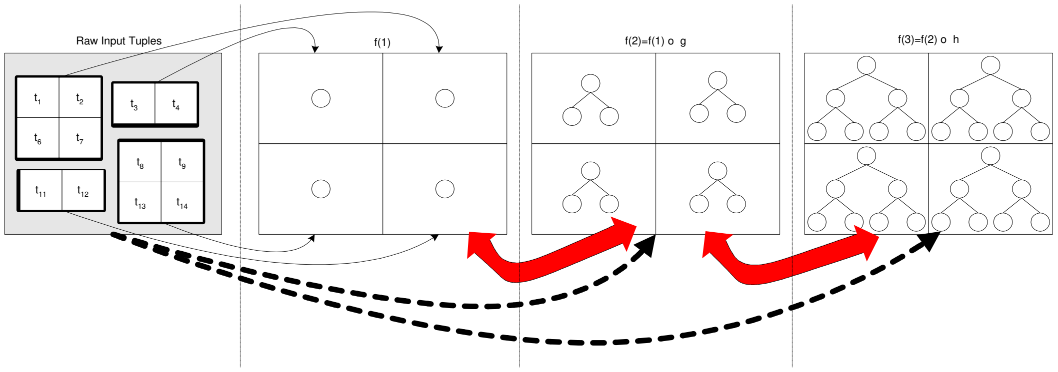


Figure 4. Inductive aggregation primitives. f(1), the base case, is computed from the raw input tuples. f(2) is computed from f(1), and f(3) from f(2). In some cases, the raw input tuples are still necessary, but aggregates from prior inductive steps are used to speed up the generation of later inductive steps.

5.3 Functional Decomposition

In many applications, step 7 in Figure 1 involves relatively *complex* operations, which can be implemented from a set of *primitive* operations. We refer to an operation as primitive if it is an application-specific minimal and indivisible part of data processing. An example is the processing of satellite data, in which sensor data is first range-selected and subsampled, correction algorithms are applied on the data, an aggregation operation is performed, and, finally, a projection is carried out to yield the final query output [25].

A complex function can be defined as a composition of several primitive operations: $O \stackrel{M}{\leftarrow} f_1 \circ f_2 \ldots \circ f_n(I)$, where $f_1, f_2, \ldots f_n$ are the primitive operations and M is the domain as defined by the query meta-data. Such an implementation results in a single aggregate for data reuse, namely the output of the composite function. However, for each of the primitive functions, different algorithms can be chosen for a query. For example, data correction is one step in query execution in Kronos, and various researchers prefer different ways of performing correction on raw sensor data [34, 41]. So a fully composite implementation effectively reduces the reusability of an aggregate. In this work, we examine the implementation of complex operations as a sequence of primitive operations: $T_1 \stackrel{M}{\leftarrow} f_3(I)$, $T_2 \stackrel{M}{\leftarrow} f_2(T_1)$, and $O \stackrel{M}{\leftarrow} f_1(T_2)$, where T_1 and T_2 are additional aggregates. When a complex operation is decomposed into a sequence of primitive operations, a query goes through a processing chain in which aggregates generated by a primitive in the chain are used by the next primitive. In essence, aggregates are materialized along the processing chain, as opposed to data elements being consumed, as in an *iterator*-based pipelined processing chain [21]. The decomposition approach has high potential to increase data reuse opportunities, but requires more space for caching aggregates than would a pipelined organization.

6 Functional Decomposition and Data Reuse in Kronos

We now describe a decomposition of the processing structure of Kronos into primitive functions and show how a data transformation model can be constructed. Ideally, the workbench for an environmental scientist exploring remotely sensed data will contain many high-level operations that may share several processing steps. Our task is to decompose the high-level operations into primitives and expose the commonalities that exist in the processing chain of the collection of queries supported by the workbench.

6.1 Functional Decomposition for Kronos

We can classify queries supported by Kronos into two main types based on the amount of complexity required for processing them. 1) **Low complexity:** a 3D

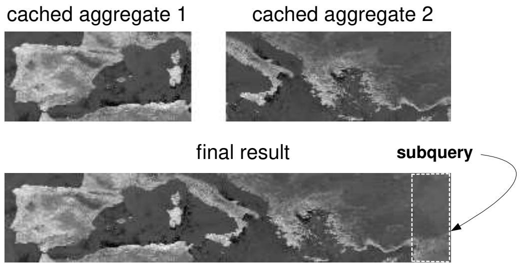


Figure 5. Dimensional overlap – two cached aggregates and an automatically generated subquery are used to generate the query result.

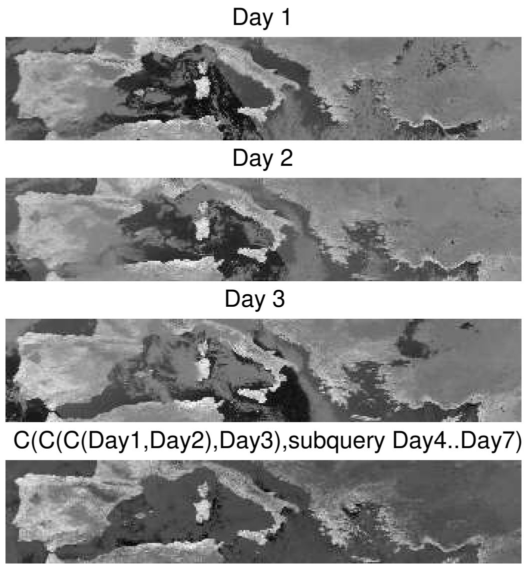


Figure 6. Composable overlap – three cached aggregates and an automatically generated subquery are composed to generate the query result.

box specifying the region and time of interest and possibly an atmospheric correction algorithm and output resolution. The raw data is retrieved and displayed. 2) **High complexity:** a 3D box specifying the region, time, compositing function, atmospheric correction algorithm, cartographic projection, and resolution. The cartographic projection is applied to data that has been aggregated, calibrated, and corrected. These two types of queries can be expressed as a combination of the following data processing primitives.

Range Selection: Given a uniform 2-dimensional grid and temporal coordinates, retrieves the relevant IFOVs from raw AVHRR data (R). The output of this function is the selected raw data (S).

Atmospheric Correction: Applies an atmospheric correction algorithm and modifies the relevant part of the selected raw data tuples (S). This function is *annotated* with a correction algorithm, with choices including Rayleigh/Ozone, Water Vapor, or Stratospheric Aerosol. The output from the atmospheric correction function is calibrated raw data (C).

Composite Generator: Generate a *data product* (D) from the calibrated raw data (C). The product generation consists of aggregating many IFOVs for the same spatial region and multiple temporal coordinates, according to a particular aggregation criteria. The function is *annotated* with the aggregation criteria, which can be either maximum NDVI (normalized difference vegetation index), or maximum sensor Channel 1 value, or maximum sensor Channel 2 value.

Subsampler: Converts the input data to a user-specified spatial resolution. The subsampling operation can be performed at different stages in the query processing chain. In Kronos, a discrete grid is computed based on the pixel resolution, and only pixels falling within a fixed distance of grid intersections are processed. Therefore, input to the subsampler primitive can be the raw data (R), the selected raw data (S), or a data product (D). The output is the subsampled raw data (S_S) or data product (D_S).

Cartographic Projection: Applies a mapping function that converts a uniform 2-dimensional grid (on the sphere) into a particular cartographic projection. Like the composite generator function, this function also is annotated with an algorithm, which can be selected from an extensive list that includes Universal Transverse Mercator, Polyconic, Gnomonic, and others. Input to this function can be selected raw data (S), a data product (D), or subsampled data (S_S or D_S). The output is the projected input (P_S, P_D, P_{S_S}, or P_{D_S}).

Figure 8 shows the chain of processing for a query of low complexity type, and the processing chain for a high complexity query is displayed in Figure 9.

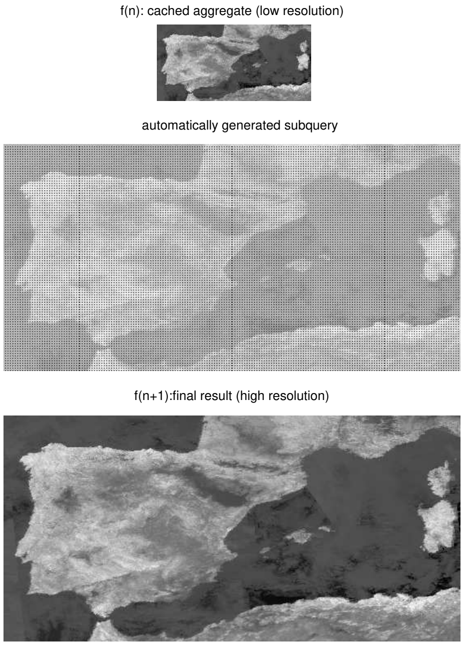


Figure 7. Inductive aggregation – a high resolution image is generated from a low resolution image and a subquery with the missing pixels. The black dots in the subquery image are the ones that are directly reused from the low resolution image.

6.2 Projection Primitives

The functional decomposition of Kronos queries, as described in the previous section, permits the deployment of various projection primitives that can leverage reuse opportunities during query execution.

The dimensional (spatio-temporal) overlap projection primitive can be employed on the output of the Range Selection function to eliminate the data elements that fall outside the bounding box of a new query. The dimensional overlap primitive can also be used on the output of other functions. When a partial overlap is detected, the spatio-temporal range attribute can be repartitioned to automatically dispatch subqueries that compute the remaining parts of the new aggregate as seen in Figure 5.

The invertible aggregation primitive can be used on the output of the Atmospheric Correction function. The original non-corrected information can be obtained by inverting the correction algorithm.

Figure 8. A low complexity Kronos query specified as a sequence of low-level function primitives. This query produces a *corrected* **and** *subsampled* **version of the raw data for a given set of geographical and temporal coordinates.**

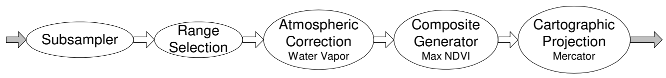


Figure 9. A high complexity Kronos query specified as a sequence of low-level function primitives. This query produces a *data product* **transformed by a cartographic projection method.**

Both the composable reduction operation primitive and the inductive aggregation primitive can be employed for the Composite Generator function. For composable reductions, the aggregation level across the two aggregates must be congruent. An example of that can be seen in Figure 6. For inductive aggregations, an aggregate can be used as an initial partial result for computing a new temporary, with additional temporal data aggregated from the raw input data. Figure 7 shows an example of that in Kronos.

The inductive aggregation primitive can be used for the Subsampler function, to compute a lower resolution output from a higher resolution aggregate, as well as a higher resolution aggregate from a lower resolution aggregate by employing additional input data.

7 System Support

The system support is implemented as an extension to the multiple query optimization (MQO) middleware we have been developing. The middleware provides a C++ class library for application developers to implement queries with user-defined processing operations. The runtime system consists of several services and employs a multithreaded execution environment in order to simultaneously execute multiple queries on a cluster of shared-memory multiprocessor machines. A description in greater detail of the middleware infrastructure can be found in [6, 7]. In this section, we present the operations supported by the caching service and describe how a functionally decomposed query is executed.

7.1 Implementing a Query

The primary step in implementing a query consists of identifying the functional primitives that make up the query processing structure. In this work, we expect the application developer to present to the system a functional description of the query type to be registered. This assumption is also made by other frameworks that require the functional decomposition of a complex computation [3, 12, 31, 32].

The *execution chain* of a query type q_i is represented by a directed acyclic graph $G_i(V, E)$, referred to as a *query graph*. A vertex represents a function primitive and an edge corresponds to a data dependency between the two primitives sharing the edge. An edge is marked with a *cacheable* flag. If this flag is set, the output of the function primitive at the tail of the edge is cached by the system. In the context of a query graph, a projection primitive can be viewed as a function primitive, hence can be represented by a vertex in the graph. A vertex is referred to as a *sink*, or *output*, vertex if it is the one that generates the output data product for the query. A *source*, or *input*, vertex is the vertex that processes the input data elements selected by the query. In a topological sort of the query graph, the sink vertex is at the top level, whereas source vertices form the bottom level of the query graph. An example of query decomposition is shown in Figure 10. In the example, query q_i has two alternative implementations. The second implementation $q_{i,2}$ uses a projection primitive, trying to decrease execution time by reusing an aggregate.

The middleware framework provides a C++ base class from which user-defined function primitives can be derived. The base class has a virtual `execute` method that the application developer implements to define a primitive operation. This function takes the data to be processed and its meta-data information, and produces a data product and meta-data information associated with the data product. The base class

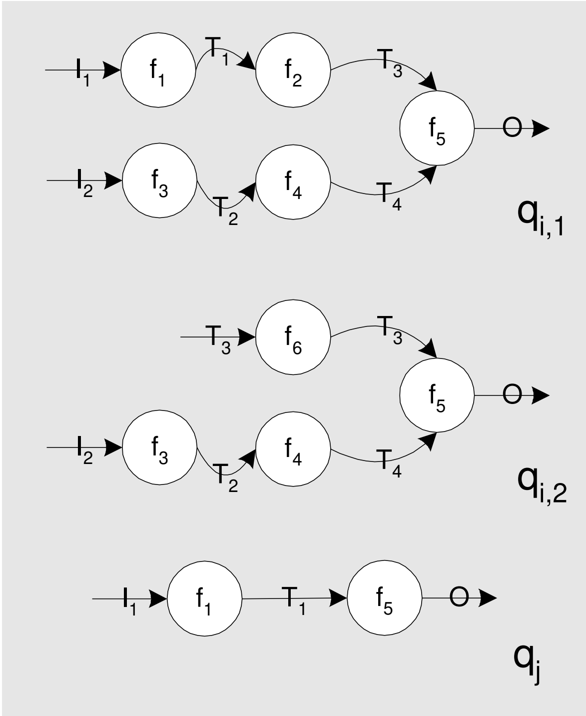


Figure 10. Two functionally decomposed queries q_i and q_j. q_i has two execution strategies $q_{i,1}$ and $q_{i,2}$. f_6, in query $q_{i,2}$, is a projection primitive, taking an aggregate of type T_3 and generating a projected aggregate of the same type. In $q_{i,1}$, f_1 and f_3, the leaf nodes, use raw data as their input.

also provides two additional virtual methods, `overlap` and `project`, which must be implemented by the application developer. The project method implements the projection primitives that can be performed on cached aggregates to produce input for the corresponding function primitive. The overlap method must return the amount of overlap between a cached aggregate and a new query, along with the type of the projection primitive(s) to be applied.

7.2 Caching Infrastructure

The central system component to exploit reuse opportunities is the caching infrastructure, referred to as the *Data Store*. The Data Store employs a two-tier architecture that uses both main memory and secondary storage. The in-core cache implements on-demand lookup for reusable aggregates, while the persistent cache in secondary storage makes it possible to maintain cached aggregates across different invocations of the server [4, 7]. In this paper, we focus on in-core caching of aggregates.

When the server is started up, a fixed amount of memory is allocated to the in-core cache. A query thread interacts with the Data Store using a *DataStore* object, which provides functionality similar to the C library *malloc* function. When a query must allocate space from the Data Store for an aggregate, the size (in bytes) of the aggregate and the corresponding meta-data information are passed as parameters to the *malloc* method of the DataStore object. This design ensures that all dynamically allocated memory is accounted for by the Data Store to avoid paging to disk. The Data Store also provides a `lookup` method. Lookup is used by the system to determine whether a query/primitive can be answered entirely or partially using the aggregates stored in the cache. A hash table is used to access meta-data information associated with each cached aggregate. The hash table is accessed using the dataset id of the input dataset. Each entry of the hash table is a linked list of aggregates allocated for and computed by other queries/primitives. The `lookup` method calls the `overlap` method for each query type in the corresponding linked list, and returns a reference to the object that has the greatest overlap with the query.

For efficient query planning and execution, there are four important mechanisms supported by the Data Store: tagging, pinning/unpinning, status and validity management. Since caching space is limited, a request for memory will trigger a replacement decision when the cache is full [5]. *Pinning* and *unpinning* are the operations that enable the system to ensure that a given aggregate will not be discarded or swapped out to secondary storage while the aggregate is being used in a computation. Pinned aggregates are not considered for replacement. The system also keeps track of the *status* of an aggregate, marking it as either still being computed or as completed. The status is needed because an aggregate being computed by a query may be identified as *useful* for another query being executed simultaneously. A related issue is the *validity* of an aggregate. During the computation of an aggregate an error may occur (e.g., because of invalid meta-data information or resource exhaustion). A validity flag is used to flag such situations so that queries that are waiting for the computation of an invalid aggregate are not blocked forever, nor attempt to use stale results.

7.3 Query Planning and Execution

The runtime system can execute multiple queries simultaneously. The level of concurrency is limited by the number of processors that are available to the system. Each query is executed as a separate thread on a single processor. If a query is described as a sequence of primitives, all the primitives are executed by the same processor. When all of the processors are busy, a new query received by the system is put into a *waiting* queue.

Query planning and execution is carried out in two steps. In the first step, a decision is made as to which query from the waiting queue is selected for execution. The second step determines the data reuse for the query and the set of subqueries that should be executed to generate the portions of the query result that cannot be generated from cached aggregates. In earlier work [9], we developed methods for scheduling queries in the waiting queue for execution. We describe here the second step to determine reuse for functionally decomposed queries.

For a given query, the query graph, $G_i(V, E)$, is traversed in a breadth-first, top-down fashion, starting from the sink vertex. Algorithm 1 is executed at each level of the graph. First, the data cache is searched for aggregates that overlap the query meta-data (step 1). If there is complete overlap (step 2), the output is computed from the cached aggregate by applying the appropriate projection primitive(s) (step 3). If cached aggregates can partially answer the query, function primitives at the next level of the graph are instantiated (step 14). The `instantiateAndRunPrimitives` function recursively executes Algorithm 1 at the next level. We note that input to a primitive can be the input dataset or the output of a primitive in the lower levels of the query graph. Therefore, when the `instantiateAndRunPrimitives` function returns, the input for the function primitives at the current level of the graph will be available.

As an example, consider $q_{i,1}$ in Figure 10. When a query of type $q_{i,1}$ is received, the runtime system starts the execution of the query at the top level (i.e., f_5) and searches the data cache for aggregates that have been created by f_5 and can be reused. If only partial or no overlap is found, f_5 is instantiated to compute the parts of the query that cannot be answered from the cache. At that point, Algorithm 1 is executed for f_5. The cache is searched for aggregates of type T_3 (created by f_2) and T_4 (created by f_4). If there are aggregates that overlap the query, appropriate projection primitives are executed to create part of the input for function f_5. Functions f_2 and f_4 are instantiated and Algorithm 1 is recursively executed for those nodes in the graph.

Algorithm 1 *bool* Primitive::run(QueryMI q, ExecutionChain ec, integer level)

Require: q the Query meta-data, ec the query graph, and the $level$ currently under execution

```
// find the cached aggregates that entirely or partially overlap with the query
stat ← lookup(ovlps)
if stat= FULL_OVERLAP then
    project(ovlps)
    return PR_OK
else
    // if partial overlap, perform projection
    if stat= PARTIAL_OVERLAP then
        project(ovlps)
    // check if we are at the last level of the graph
    if level ≠ LEAF_LEVEL then
        // generate and run primitives at (level+1) to compute the parts
        // of the query that cannot be answered by cached items
        instantiateAndRunPrimitives(q,ec,level+1)
    // execute the function primitive
    ret ← execute()
    return ret
```

8 Experimental Evaluation

In our experimental evaluation we use both representative single queries in the context of Kronos and batches of multiple queries generated from a synthetic workload. In Section 8.1 we present the synthetic workload model used to generate multiple queries. The experiments with single queries aim to quantify overheads and performance benefits of the system presented in this paper compared to the original Kronos system, which is tailored for single query evaluations.

We present five collections of experiments: (1) a performance study of the benefits of functional decomposition, (2) an evaluation of the performance improvements obtained by employing all combinations of projection primitives when a medium-sized collection of queries is evaluated, (3) an analysis of how each projection primitive behaves when overlaps of different types are observed, (4) an analysis of the overheads our system introduces when a single representative query is evaluated, and (5) an analysis of parallel execution in which multiple clients concurrently generate queries to be executed. Section 8.2 describes these experimental results. The experiments were run on a 24-processor SunFire 6800 Solaris 2.8 machine with 24GB of main memory. A dataset consisting of one month (January/1992) of AVHRR data was used, totaling about 30GB.

8.1 A Synthetic Workload Model

In order to evaluate the benefits of partitioning an application into primitives and to quantitatively measure the performance improvements, we investigate the system performance using a synthetic workload model. We employ a variation of the Customer Behavior Model Graph (CBMG), which is a technique utilized, for example, by researchers investigating performance aspects of e-business and website capacity planning [28]. 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. The static part of the CBMG model addresses the types of queries supported by the system, and does not depend on the way a particular client interacts with the system. The dynamic part of the model describes the possible ways of transitioning from one state to another.

A typical query in Kronos employs all the primitives, from getting the raw data in the range-selection primitive to getting the final data product from the Composite Generator. The query specifies a geographical region, a set of temporal coordinates (a continuous period of days), a resolution level (both vertical and horizontal), a correction algorithm (from 3 possibilities), and a compositing operator (also from 3 possi-

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

Table 1. Transition probabilities for two workloads.

bilities). Once a query is executed, the expected transitions from that query to the next one are given by one of the following operations: *spatial movement* (up, down, left, and right), *temporal movement*, *resolution increase* or *decrease*, applying a different *correction algorithm*, or applying a different *compositing operator*.

Among the dynamic aspects of a query is the geographical region and time it refers to. In our model, some spatio-temporal coordinates are deemed as hot spots. This assumption is grounded in the real way systems like Kronos are used, i.e., specific regions are more important than others for either political or scientific reasons. For example, a state department of agriculture is much more likely to run queries regarding its own state than others, or in crop yield prediction studies, the time immediately before the corn harvesting period is of special interest. Thus, one input to the workload model is a set of *hot* spatio-temporal coordinates where most queries are expected to occur. A fixed probability of selection is assigned to each hot spot. Another part of the model describing how the query server is used relates to the GUI application used to generate queries. The model assumes that queries will be displayed in windows with a set of fixed resolutions (e.g., 512×512 pixels) chosen on a per client basis. Each resolution has equal probability of being selected. For the correction algorithms and compositing operator, each algorithm is assumed to have an equal probability of being picked. The time period to composite over is chosen from four possibilities (1, 2, 4 and 7 days) with equal probability. Within the described framework, the transitions in the CBMG are modeled with several operations:

New Point-of-Interest: randomly selects another hot spot as its central coordinate and adds spatial noise (to avoid the central point being the exact the hot spot coordinate). Also randomly pick the correction algorithm, compositing operator, and length of compositing (in number of days). There are five hot spots: Southern California (January 3, 1992), the Amazon Forest (January 5, 1992), the Sahara Desert (January 7, 1992), the state of Maryland (January 20, 1992), and the Iberian Peninsula (January 15, 1992).

Spatial Movement: changes the spatial coordinates in such a way that a half-screen vertical or horizontal scroll is obtained. The direction of the movement is selected with uniform probability. All other query variables remain the same as in the previous state.

New Resolution: changes the spatial resolution, zooming in or out. There are five pre-defined resolutions: 16, 64, 144, 576, or 1296 Km^2 per pixel. The resolution is increased or decreased one level with equal probability.

Temporal Movement: moves to a point in the past or in the future (in terms of number of weeks). There are four equally probable options options: 1 or 2 weeks in the past, and 1 and 2 weeks in the future.

New Correction: changes the correction algorithm and leaves all other query variables the same. The new algorithm is picked with uniform probability from the three available.

New Compositing: changes the compositing operator and leaves all other query variables the same. The new operator is picked with uniform probability from the three available.

New Compositing Level: changes the compositing level (more or fewer days) and leaves all other query variables the same. The period of composition is increased/decreased in both directions, i.e., the initial and final day. The size of the increase/decrease is picked with uniform probability from 4 possible ones (1, 2, 4, or 7 days).

For a given client, the initial state – empty – must use the *New Point-of-Interest* transition to get started. During startup, the window size is also pre-selected and kept fixed across the client session. From that point forward, each transition has its own fixed probability of being selected. The number of queries to be generated per client is fixed, and that number is also an input variable to the model. For the experiments in the next section, we have used the transition matrix in Table 1. *Workload 1* was used for all the experiments and *workload 2* was used specifically to help analyze the performance impact of functional decomposition.

8.2 Experiments

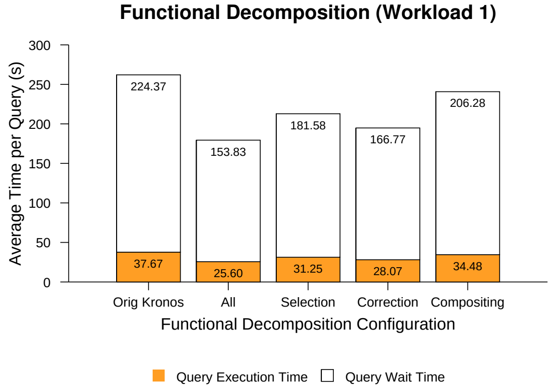
Evaluation of Functional Decomposition. Exposing reuse sites through functional decomposition increases the likelihood that a query will be partially or completely computed from cached aggregates. The benefits of this technique are dependent on the workload the system is processing. In order to evaluate functional decomposition, we have collected experimental results for a single configuration: 16 clients generating 4 queries each[2], submitting them to the server running with a fixed cache size of 1GB and with two query threads[3]. Two different workloads were employed: workload 1 and workload 2, with transition probabilities as presented in Table 1. Workload 1 has the bulk of its transitions driven by spatial movement, which means that a great deal of reuse across queries can occur at the compositing level (making caching at that level alone potentially beneficial). On the other hand, workload 2 has most of its transitions changing the compositing and atmospheric correction methods, which requires reaggregating the data at the compositing level (making caching at that level not especially beneficial). Of particular interest in Figures 11 (a) and (b) is that caching at all levels improves performance much more than configurations in which only a single caching point was exposed. For the two workloads, caching at all levels achieves about a 30% decrease in average query execution time compared to Kronos. For workload 2, we see that caching only at the last primitive in the processing chain is not helpful, because very little reuse can occur at that level.

Evaluation of Combinations of Projection Primitives. For this experiment, we employed a workload generated according to the synthetic workload model described in Section 8.1, consisting of 32 queries. In addition to testing all sixteen combinations of projection primitives, we also measured the performance of the original Kronos code. For a fair comparison, only a single query was executed at any given point in time for the active caching system (i.e. there was only a single query thread). The results in Figure 12 show that employing projection primitives is a main contributor to increasing overall system throughput. In the figure, DIM stands for Dimensional Overlap, INV for Invertible Aggregation, COMP for Composable Aggregation, and IND for Inductive Aggregation. We should note that DIM and IND can be employed for the output of all function primitives, whereas COMP is suitable for the output of Composite Generator and INV for that of Atmospheric Correction (Figure 9). In the experiments, we also have observed that the performance increases provided by employing each projection function in isolation are much less than those obtained by combining multiple projection functions. As is seen in Figure 12, combining functions enables greater reuse by making possible the use of other transformation functions.

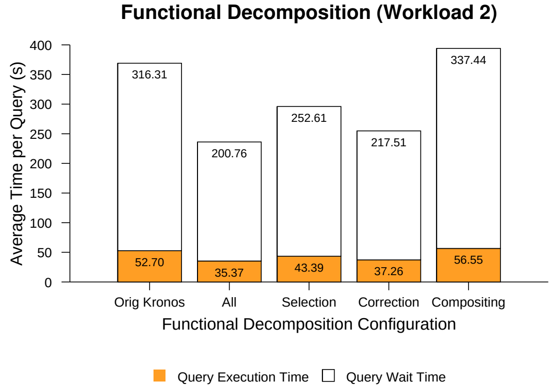
An interesting result from Figure 12 is that having all the optimizations turned on is not necessarily the best strategy, although it does produce an almost 50% drop in batch execution time compared to the original Kronos code. The configuration DIM-COMP-IND (all but Invertible Aggregation primitives are turned on) yields the overall best performance by a small margin. We attribute this result to the fact that Algorithm 1, which is responsible for detecting and reusing cached results, employs a top-down, greedy strategy that assumes that the higher an aggregate is in the execution chain (i.e. closer to a sink), the more *profitable* it is to have the aggregate employed for reuse. On the other hand, in Kronos the only situation in which the invertible aggregation function is employed is in the correction primitive. That primitive allows the runtime system to turn the *corrected* values back into the original input values, so that another correction method can be applied. In this scenario, if the input data is in the cache (from the *range-selection* primitive), the strategy in Algorithm 1 is definitely more expensive than simply using the input data and applying the new correction method directly. Hence, this experiment shows that a cost model for evaluating the query plan is needed in order to assign weights to each possible

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

[3]Since Kronos can only process one query at a time, to model Kronos processing two queries simultaneously we wrote a wrapper that spawns up to two separate Kronos processes on demand.



(a)



(b)

Figure 11. The effects of functional decomposition for (a) Workload 1 and (b) Workload 2. Average time per query is shown for 1) the original Kronos code, 2) for caching at all levels, 3) caching only at range-selection, 4) caching only at correction, and 5) caching only at the last stage of the execution chain – compositing.

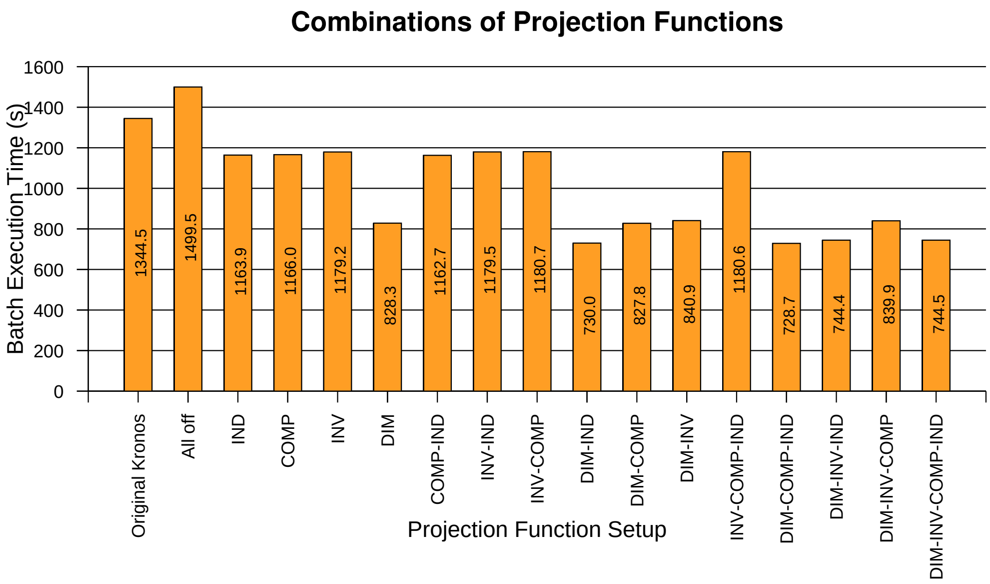


Figure 12. Combinations of projection functions for a batch of 32 queries. The horizontal axis shows the configuration of projection functions, stating the ones that are turned on. DIM stands for Dimensional Overlap, INV for Invertible Aggregation, COMP for Composable Aggregation, and IND for Inductive Aggregation.

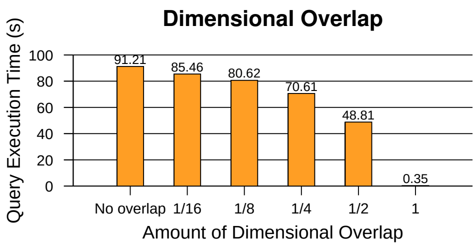


Figure 13. Improvements for Dimensional Overlap.

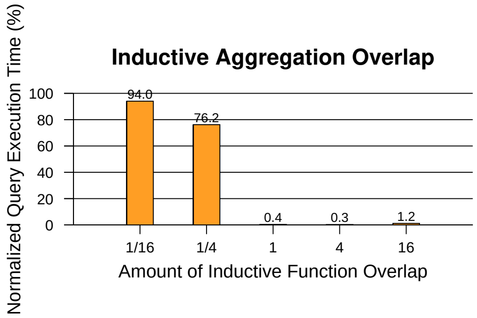


Figure 14. Improvements from Inductive Aggregation Overlap. The chart shows in percent how much time is needed to compute the query result assuming the availability of an aggregate whose resolution level is some multiple of the new query resolution.

optimization path. We also observe that dimensional overlap is responsible for the greatest improvement in batch execution time for the experimental workload. This is not a completely general claim, though, since in the synthetic workload model 40% of the transitions between queries were set to be spatio-temporal movements. Therefore, when a larger portion of the clients' requests require other correction or compositing methods, the other types of overlap will have a larger role in decreasing overall batch execution time.

Evaluation of Performance Improvement by Individual Projection Primitives. To understand how much each projection primitive contributes to decreasing query execution time, we evaluated the system using queries specifically tailored to benefit from data reuse opportunities provided by a single projection primitive. In each of the configurations, the baseline (100%) case is the execution of a given query completely from raw input data *versus* computing it with various amounts of overlap with a previously cached aggregate.

For spatio-temporal overlaps, we studied how query execution time is affected when the system identifies a cached aggregate that can be used to partially or fully answer the query. The test query produces a 7-day composite for the continental United States that requires retrieving 258.5 MB of input data. As is seen in Figure 13, we tried executing the query when the cache was empty and when aggregates that covered from one sixteenth up to half of the final query result were available in cache.

In order to see how effective detecting inductive aggregation overlaps is, we used a query with the same attributes as the query for the spatio-temporal overlaps experiment. However, we changed its resolution to 144 Km^2 per pixel. Each new query was answered by reusing the cached result generated by this query. The performance improvements are shown in Figure 14. Queries with resolution ratios of 16, 4, and 1, respectively, have an almost instantaneous response, since the cached composite only needs to be subsampled to yield the new query result. As the resolution ratio goes to 1/4 and 1/16, the improvement is less dramatic because a subquery needs to be generated and run to compute missing pixels, but some benefit from reuse is still seen.

For analyzing composable aggregation overlap, we employed a continental U.S. query, but one that generates an 8-day composite. We measured how long the query takes to execute, assuming the cache contains either a partial match that is a 1-day, 2-day, or 4-day composite, with the rest of the query answered from input data. The results can be seen in Figure 15.

For Kronos, the only primitive that allows for invertible aggregation reuse is correction. To evaluate its benefits, we show the performance improvements from inverting a result to produce the original input data (as opposed to retrieving the input data again), and then transforming the inverted values by applying a new correction method. Figure 16 shows the

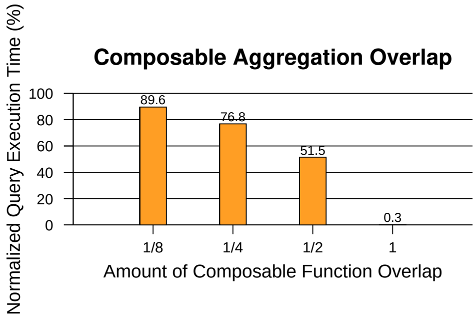


Figure 15. Improvements from Composable Reduction Overlap. The chart shows in percent how much time is needed to compute the query result assuming the availability of an aggregate with an aggregation level that evenly divides into the new query aggregation level.

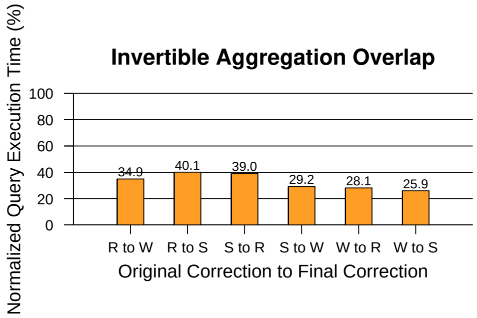


Figure 16. Improvements from Invertible Function Overlap. The chart shows in percent how much time is needed to compute the query result with a given correction method, given that the result is already available from another correction method. R, W, and S correspond, respectively, to Rayleigh/Ozone, Water Vapor, and Stratospheric Aerosol correction methods.

benefits obtained, given that a 7-day composite for the continental U.S. is available that used one correction method, and then a query requiring the same composite but using a different correction method is submitted. We observe improvements between 60% and 75%. This result confirms that the computation to apply a new correction method can be considerably less expensive than retrieving the input data from disk again.

Evaluation of System Overhead in Single-Query Scenarios. Using the active caching system to detect and explore reuse comes at a price. Single query evaluations may become more expensive to compute than in a monolithic system. Using three *typical* queries, we evaluate the extra costs imposed by our middleware as compared to the original Kronos code. The three queries are classified as *continental*, *regional*, and *local*, differing in the amount of data that needs to be retrieved, aggregated, and output. The continental query produces a 7-day aggregate using a particular correction/compositing combination for the continental U.S. with a 144 Km^2 per pixel resolution, the regional query generates the same aggregate with a 36 Km^2 per pixel resolution for the mid-eastern U.S. (a box encompassing (Kentucky, Tennessee, West Virginia, Maryland, Delaware, Virginia, North Carolina, South Carolina, and the District of Columbia), and the local query produces the same aggregate with a 9 Km^2 per pixel resolution for the region surrounding Chesapeake Bay. As Table 2 shows, the overhead introduced by the middleware (designated by MQO in the table) can be significant for small queries, however, for larger ones it is usually below 20%. This overhead is a by-product of buffer copies and bookkeeping tasks the middleware needs to perform to reuse aggregates for executing multiple query workloads.

Multithreading Improvements. Our middleware is designed to service multiple query workloads, because it optimizes query execution from active caching, and also because it supports simultaneous execution of multiple queries. In this experiment, we evaluate the system with respect to its multithreaded support. We employed 16 clients, each submitting an 8-query workload assembled using the previously described synthetic workload model. The cache space was fixed at 1GB. The results in Figure 17 show that the new system outperforms the original Kronos application by a factor of 5.6 when up to 16 queries are executed at the same time. In this configuration, the amount of commonality across the queries is much

Query	*Disk Read Volume* (MB)	*Original Kronos Execution Time* (s)	*MQO Execution Time* (s)
Continental (US)	258.5	81.99	91.64
Regional (Mid-eastern US)	89.7	22.19	26.35
Local (Chesapeake Bay)	74.4	3.85	5.40

Table 2. Single-query overhead.

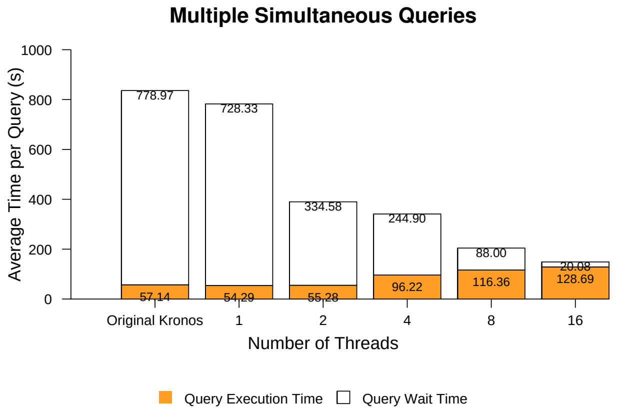


Figure 17. Multithreaded execution. Query execution time is measured from the time a query gets scheduled for execution until it completes. The wait time corresponds to the time a query spends in the waiting queue before being scheduled for execution.

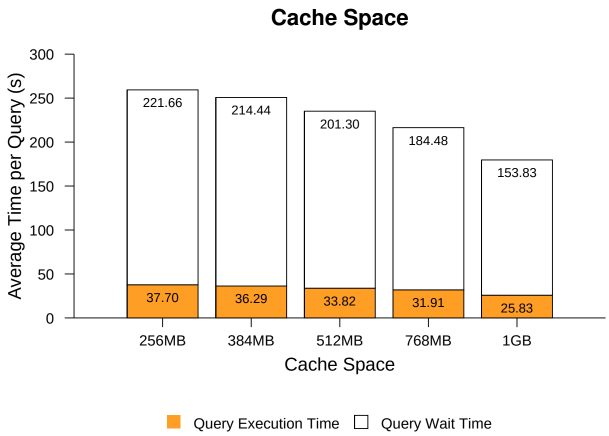


Figure 18. Increasing the space for storing cached aggregates improves the chances of detecting overlaps, lowering average query execution time.

lower than in the previous experiments since there are 16 clients acting independently. That is why for the 1-thread configuration the relative improvement is smaller than was shown in our first set of experiments (when only a single client was submitting queries). For the 4-thread configuration execution time increases sharply compared to the 1-thread case, showing that I/O has become a bottleneck since the application is data intensive, even though adequate computational power is available (the machine has 24 processors). The I/O bottleneck is the reason why a sublinear decrease in execution time is seen as more simultaneous queries are allowed to be executed.

Cache Space Improvements. The active caching system becomes more effective in exploiting reuse as the amount of space allocated for caching increases . In this last experiment, we show the system behavior as we increase the amount of cache space for a workload generated by 16 clients submitting 4 queries each. We kept the number of simultaneously evaluated queries fixed at 2, and varied the cache size from 256MB up to 1GB, as seen in Figure 18. A 30% drop in average query execution time is observed when the cache size is increased from 256MB up to 1GB.

9 Conclusions

Functional decomposition and automatic data reuse strategies are the central contributions of this work.

Functional decomposition enables data reuse, since decomposing the application improves the odds that a particular optimization can be employed. We have shown that these techniques can improve the performance of data analysis applications that perform range queries, when multiple query batches are executed. Our results also show the extra software infrastructure required by the runtime system to optimize multiple query workloads only lightly impacts system performance for single query evaluation. More importantly, the performance improvements obtained for multi-query batches largely outweigh those overheads.

An important result of this work is that, although multiple overlap/project strategies can be employed, the relative benefits for using each strategy individually is highly dependent on the types of commonalities (data access patterns and computations to be performed) present in the workload. Moreover, providing a cost model to analyze potential optimization paths is an area of future research that can ensure that the best set of strategies is employed when considering all the possible data transformations and a given set of available cached aggregates. We plan to explore this topic in more details in future work.

Although not the central point of this work, the Kronos application showed the characteristics of an I/O bound application, especially under heavier workloads. Our middleware system is able to employ clusters of machines, as well as more decentralized Grid configurations, which can be utilized to improve system throughput under such circumstances, by aggregating the I/O bandwidth available across multiple nodes. We plan to undertake an experimental investigation of these complex scenarios in the near future.

Acknowledgments. We would like to thank Frank Lindsay, Michael McGann, and Saurabh Channan from the Global Land Cover Facility at the Institute for Advanced Computer Studies for providing the Kronos code and the AVHRR datasets.

References

[1] M. Aeschlimann, P. Dinda, J. Lopez, B. Lowekamp, L. Kallivokas, and D. O'Hallaron. Preliminary report on the design of a framework for distributed visualization. In *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'99)*, pages 1833–1839, Las Vegas, NV, June 1999.

[2] A. Afework, M. D. Beynon, F. Bustamante, A. Demarzo, R. Ferreira, R. Miller, M. Silberman, J. Saltz, A. Sussman, and H. Tsang. Digital dynamic telepathology - the Virtual Microscope. In *AMIA98*. American Medical Informatics Association, November 1998. Also available as University of Maryland Technical Report CS-TR-3892 and UMIACS-TR-98-23.

[3] K. Amiri, D. Petrou, G. R. Ganger, and G. A. Gibson. Dynamic function placement for data-intensive cluster computing. In *the USENIX Annual Technical Conference*, San Diego, CA, June 2000.

[4] H. Andrade, T. Kurc, U. Catalyurek, A. Sussman, and J. Saltz. Persistent caching in a multiple query optimization framework. In *Proceedings of the 6th Workshop on Languages, Compilers, and Run-Time Systems for Scalable Computers*, Washington, DC, March 2002.

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

[6] 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.

[7] 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 (to appear)*, Baltimore, MD, November 2002. Also available as University of Maryland Technical Report CS-TR-4361 and UMIACS-TR-2002-41.

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

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

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

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

[12] M. D. Beynon, R. Ferreira, T. Kurc, A. Sussman, and J. Saltz. DataCutter: Middleware for filtering very large scientific datasets on archival storage systems. In *Proceedings of the Eighth Goddard Conference on Mass Storage Systems and Technologies/17th IEEE Symposium on Mass Storage Systems*, pages 119–133, College Park, MD, March 2000.

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

[14] E. Borovikov, A. Sussman, and L. Davis. An efficient system for multi-perspective imaging and volumetric shape analysis. In *Proceedings of the 2001 Workshop on Parallel and Distributed Computing in Imaging Processing, Video Processing, and Multimedia*, San Francisco, CA, 2001.

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

[16] 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.

[17] 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.

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

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

[20] H. Fallah-Adl, J. Jájá, S. Liang, J. Townshend, and Y. J. Kaufman. Fast algorithms for removing atmospheric effects from satellite images. *IEEE Computational Science & Engineering*, 3(2):66–77, Summer 1996.

[21] G. Graefe. Query evaluation techniques for large databases. *ACM Computing Surveys*, 25(2), 1993.

[22] J. Han, N. Stefanovic, and K. Koperski. Selective materialization: An efficient method for spatial data cube construction. In *Pacific-Asia Conference on Knowledge Discovery and Data Mining*, pages 144–158, 1998.

[23] High Performance Fortran Forum. High performance fortran – language specification – version 2.0. Technical report, January 1997. available at http://www.netlib.org/hpf.

[24] C. Isert and K. Schwan. ACDS: Adapting computational data streams for high performance. In *14th International Parallel & Distributed Processing Symposium (IPDPS 2000)*, pages 641–646, Cancun, Mexico, May 2000. IEEE Computer Society Press.

[25] 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.

[26] 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.

[27] T. Kurc, U. Catalyurek, C. Chang, A. Sussman, and J. Saltz. Visualization of large datasets with the Active Data Repository. *IEEE Computer Graphics and Applications*, 21(4):24–33, July/August 2001.

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

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

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

[31] R. Oldfield and D. Kotz. Armada: A parallel file system for computational. In *Proceedings of CCGrid2001: IEEE International Symposium on Cluster Computing and the Grid*, Brisbane, Australia, May 2001. IEEE Computer Society Press.

[32] B. Plale and K. Schwan. dQUOB: Managing large data flows using dynamic embedded queries. In *IEEE International High Performance Distributed Computing (HPDC)*, August 2000.

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

[34] D. P. Roy. Investigation of the maximum normalized difference vegetation index (NDVI) and the maximum surface temperature (T_s) AVHRR compositing procedures for the extraction of NDVI and T_s over forest. *International Journal of Remote Sensing*, 18(11):2383–2401, 1997.

[35] 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.

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

[37] C. T. Shock, C. Chang, B. Moon, A. Acharya, L. Davis, J. Saltz, and A. Sussman. The design and evaluation of a high-performance earth science database. *Parallel Computing*, 24(1):65–90, January 1998.

[38] M. Spencer, R. Ferreira, M. Beynon, T. Kurc, U. Catalyurek, A. Sussman, and J. Saltz. Executing multiple pipelined data analysis operations in the Grid. In *Proceedings of the 2002 ACM/IEEE SC02 Conference*. ACM Press, November 2002. To appear.

[39] P. M. Teillet, N. E. Saleous, M. C. Hansen, J. C. Eidenshink, C. Justice, and J. R. G. Townshend. An evaluation of the global 1-km AVHRR land dataset. *International Journal of Remote Sensing*, 21(10):1987–2021, 2000.

[40] J. Yang, K. Karlapalem, and Q. Li. Algorithms for materialized view design in data warehousing environment. In *Proceedings of the 23th VLDB Conference*, pages 136–145, 1997.

[41] Z. Zhang, J. JáJá, D. Bader, S. Kalluri, H. Song, N. E. Saleous, E. Vermote, and J. R. G. Townshend. Kronos: A java-based software system for the processing and retrieval of large scale AVHRR data sets. Technical Report EECE-TR-99-006, University of New Mexico, November 1999.