Servicing Mixed Data Intensive Query Workloads

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

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

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

Abstract

When data analysis applications are employed in a multi-client environment, a data server must service multiple simultaneous queries, each of which may employ complex user-defined data structures and operations on the data. It is then necessary to harness inter- and intra-query commonalities and system resources to improve the performance of the data server. We have developed a framework and customizable middleware to enable reuse of intermediate and final results among queries, through an in-memory semantic cache and user-defined transformation functions. Since resources such as processing power and memory space are limited on the machine hosting the server, effective scheduling of incoming queries and efficient cache replacement policies are challenging issues that must be addressed. We have addressed the scheduling problem in earlier work, and in this paper we describe and evaluate several cache replacement policies. We present experimental evaluation of the policies on a shared-memory parallel system using two applications from different domains.

1 Introduction

The availability of low-cost storage systems, built from a cluster of PCs with a disk farm, allows many institutions to create data repositories and make them available for collaborative use. As a result, efficient handling of multiple query workloads is an important optimization in many application domains [2, 21, 38].

The query optimization and scheduling problem has been extensively investigated in the relational database community [15]. Multiple query optimization techniques for relational databases traditionally rely on caching common subexpressions [25, 30, 33, 37]. Nevertheless, deploying these techniques in a broader context, specifically for data analysis applications, remains a challenging problem. Several database and middleware frameworks that target this class of applications have been developed [6, 7, 8, 13, 20]. Although these frameworks provide efficient and scalable common runtime support for a wide range of applications, they do not attempt to leverage inter- and intra-query commonalities when executing multiple query workloads.

When data analysis techniques are employed in a collaborative environment, queries from multiple clients are likely to have overlapping regions of interest and similar processing requirements (i. e. the same operations on data). Several optimizations can be applied to improve system response time. These optimizations include reuse of intermediate and final results, data prefetching, and data caching. Since resources, such as CPU power and memory space, are limited on the machine hosting the data server, two issues that should be addressed to optimize use of available resources are (1) effectively scheduling incoming queries and (2) efficient cache replacement policies. We have previously developed an object-oriented framework to support efficient utilization of common subexpressions and partial results [2]. The underlying runtime system implements an in-memory semantic cache to maintain user-defined data structures for intermediate results. In earlier work [4] we addressed the query scheduling problem, and in this paper we evaluate cache replacement policies. We describe the implementation of two applications using the object-oriented framework. These applications come from different domains and exhibit different data access and processing characteristics. Using the two applications as case studies, we investigate a number of cache replacement policies, when a data server has to concurrently service data intensive query workloads from multiple applications. We present experimental results for these applications on a shared-memory parallel system.

2 Related Work

Gupta et. al. [19] point out that in most prior solutions to the multiple query optimization problem, cache space constraints and the choice of the best set of intermediate results to maintain in a cache have been ignored. Their work presents an approach for ordering the query workload so that each query benefits the most from cached results. In our work, we formulate this issue as a cache replacement problem. The cache replacement problem has been extensively investigated in the domain of buffering techniques for operating systems, and as a way to improve the performance of web servers. However, in general, researchers have used the least recently used (LRU) algorithm as the replacement policy of choice [28]. Robinson and Devarakonda [31] note that LRU relies on temporal locality to provide good performance. We argue that, for effectively handling mixed query workloads for data analysis applications, temporal locality is not the only important factor in optimization, because different cached intermediate results (*aggregates*) may have different construction costs and evicting the wrong aggregate (e. g, one that is very expensive to compute) can degrade performance significantly.

Dar et. al. [17] explore data caching and cache replacement issues for client-side caching in a client-server relational database system that stores semantic descriptions of cached data items, similar to what our data analysis middleware does (as can be seen in Section 4). The authors state that the semantic description is the building block for more sophisticated replacement policies that are not based solely on temporal and spatial locality. They discuss a policy based on *semantic distances*, in which cached results that are "closer" to recent queries are less likely to be evicted. That work does not consider any sort of data retrieval and computation costs (i.e. the cost to re-compute the aggregate) as a parameter. On the other hand, in the high performance web server and web proxy community, this is a concern that has permeated the research - specifically taking into consideration the cost of retrieving an object that was evicted from the cache. Cao and Irani [11] present a caching algorithm that incorporates locality as well as cost and size as parameters for eviction. Arlitt et. al. [5, 18] expand on this work by conducting a performance evaluation of web proxy replacement policies and suggest policies that are geared towards keeping more popular and smaller objects in cache.

Although we have formulated our issue as a cache replacement policy, the fact that intermediate results that are cached in our framework may have different construction costs (including both I/O and computation) and different ratios of construction cost to the amount of cache space needed for storage makes our scenario quite different from previous work. Our contribution in this paper is to explore cache replacement policies that make better use of information available in terms of the various costs associated with cached data structures, so the query answering system can be more effective in reducing query execution time for potentially expensive data analysis queries.

3 Motivating Applications

There is a rapidly growing set of applications that query and analyze scientific datasets. Large scientific datasets arise in many fields. Examples include datasets from long running simulations of time-dependent phenomena that periodically generate snapshots of their state [12, 16, 24, 35], archives of raw and processed remote sensing data [23, 26], archives of medical images [1, 36], and gene and protein databases [27, 29]. One example of a data analysis applications is water contamination studies. Environmental scientists study the water quality of bays and estuaries using long running hydrodynamics and chemical transport simulations [16]. The fluid flow data generated by hydrodynamics simulations can be used by multiple chemical transport simulators, which model reactions and transport of contaminants to carry out various contamination and recovery scenarios [22]. Another example is satellite data processing, in which earth scientists study environmental conditions by processing remotely sensed data continuously acquired from satellite-based sensors. A typical analysis processes satellite data for several days to a year and generates one or more composite images of the area under study [14]. We now briefly describe the two applications used as case studies for this paper.

Analysis of Microscopy Data: The Virtual Microscope (VM) [1] is an application designed to support interactive viewing and processing of digitized images of tissue specimens. VM provides a realistic digital emulation of a high power light microscope. The raw data for such a system can be captured by digitally scanning collections of full microscope slides under high power. At the basic level, VM can emulate the usual behavior of a physical microscope including continuously moving the stage and changing magnification and focus. In order to achieve high I/O bandwidth during data retrieval, each focal plane in a slide is regularly partitioned into data chunks, each of which is a rectangular subregion of the 2D image. Each pixel in a chunk is associated with a coordinate (in x- and y-dimensions) in the entire image. During query processing, the chunks that intersect the query region are retrieved from disk. Each retrieved chunk is first clipped to the query window. Each clipped chunk is either subsampled or a local averaging operation is applied to achieve the magnification level (zooming factor) specified by the query. The resulting image blocks are directly sent to the client. The client assembles and displays the image blocks to form the query output.

Volumetric Reconstruction in Multi-perspective Vision: The availability of commodity hardware and recent advances in vision-based interfaces, virtual reality systems,

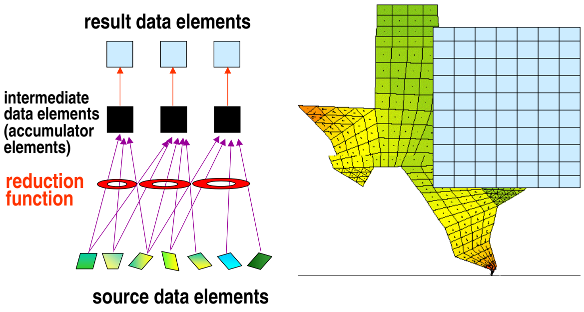

Figure 1. Typical query processing for a data analysis application: raw data is retrieved from storage, a reduction operation is applied, which generates intermediate data elements. The intermediate results are combined to generate the final query result.

and more specialized interests in 3D tracking and 3D shape analysis have given rise to multi-perspective vision systems, i. e. systems with multiple cameras usually spread throughout the perimeter of a room [10, 34]. The cameras shoot a scene from multiple perspectives and post-processing algorithms are used to develop volumetric representations that can be used for various purposes, including 1) to allow an application to render the volume from an arbitrary viewpoint at any point in time, 2) to enable users to analyze the 3D shape by requesting region-varying resolution in the reconstruction, 3) to create highly accurate three dimensional models of the shape and reflectance properties of three dimensional objects, and 4) to obtain combined shape and action models of complex non-rigid objects. The data volume associated with a single multi-perspective image stream can be substantial. An example is the Keck Lab at the University of Maryland [10]. The laboratory consists of 64 cameras that synchronously capture video streams at a rate of up to 85 frames a second; one minute of such multi-perspective video requires approximately 100GB of storage. The reconstructed volume for a single frame is represented as an occupancy map encoded with an octree representation [32]. A database query into a multi-perspective image dataset specifies a 3D region in the volume, a frame range, the cameras to use, and a resolution for the reconstruction. The query result is a multi-resolution reconstruction of the foreground object region lying within the query region. This information can then be used for 3D shape analysis and 3D tracking.

4 A Framework for Data Reuse in Processing Multiple Queries

The example applications presented in Section 3 seemingly differ greatly in terms of their input datasets and resulting data products. However, processing of queries for these applications shares some common characteristics. Figure 1 depicts the query processing structure in these data analysis applications. Pseudo-code representing the structure is shown in Figure 2. In the figure, 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 first phase of query processing (lines 2 and 3) allocates and initializes an accumulator, which is a user-defined (or application-specific) data structure to maintain intermediate partial results. The reduction phase[1] consists of retrieving the relevant data items specified by M_i (line 5), mapping these items to the corresponding output items (line 6), and executing application specific aggregation operations on all the input items that map to the same output item (lines 7 and 8). Oftentimes, aggregation operations are commutative and associative. That is, the output values do not depend on the order input elements are aggregated. To complete the processing, the intermediate results in the accumulator are post-processed to generate final output values (lines 9 and 10).

In an environment where multiple clients submit queries to a data server, many instances of inter- and intra-query commonalities will appear (e. g., visualization of an interesting feature by many users). That is, two queries q_i and q_j, described by query predicate meta-information[2] M_i and M_j, respectively, may share input elements i_e (line 5), accumulator elements a_e (line 8), and output elements o_e (line 10). The framework described in this paper handles reuse of input items i_e by implementing a buffer management layer that caches input data elements, much in the same way as traditional database management systems. More interesting, though, is reusing a_e and o_e, after they are computed during query processing. These entities sometimes cannot be directly reused because they may not exactly match a later request, but may be reused if some data transformation can be applied (i.e. convert a cached aggregate into the one that is required). Our prior results show that, because of the large volumes of data processed by the targeted class of applications, reusing results from previous queries via transformations often leads to much faster query execution than computing the entire output from the input data. Therefore, a data transformation model is the cornerstone of the multiple

[1] This phase is called the *reduction phase* because the output dataset is usually (but not necessarily) much smaller than the input dataset.

[2] Query meta-information describes which part of the dataset is required to satisfy a query, and is domain dependent (e. g. a 3-dimensional bounding box in a visualization application or a boolean expression for a relational database query).

$I \leftarrow$ Input Dataset
$O \leftarrow$ Output
$A \leftarrow$ Accumulator
1. $[S_I] \leftarrow Intersect(I, M_i)$
(* Initialization *)
2. **foreach** a_e **in** A **do**
3. $a_e \leftarrow Initialize()$
(* Reduction *)
4. **foreach** i_e **in** S_I **do**
5. **read** i_e
6. $S_A \leftarrow Map(i_e)$
7. **foreach** a_e **in** S_A **do**
8. $a_e \leftarrow Aggregate(i_e, a_e)$
(* Finalization *)
9. **foreach** a_e **in** A **do**
10. $o_e \leftarrow Output(a_e)$

Figure 2. The query processing algorithm. Our framework has provisions for optimizing steps 5–8 and 9-10 for scenarios where multiple query workloads coming from one or more users are directed to the system.

query optimization framework. The following equations describe the abstract operators the system uses in order to explore common subexpression elimination and partial redundancy opportunities:

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

$$overlap_{project}(M_i, M_j) = k, 0 <= k <= 1 \quad (2)$$

$$\mathcal{I}_{M_i} \overset{project(M_i, M_j, \mathcal{I})}{\rightarrow} \mathcal{J}_{M_j} \quad (3)$$

Equation 1 describes the $compare$ function that returns *true* or *false* depending on whether intermediate data result $\mathcal{I}$ described by its predicate M_i is the same as $\mathcal{J}$ as described by M_j. When the application of this function returns true, the system has identified a common subexpression elimination opportunity, because query q_j can be completely answered by returning $\mathcal{I}$.

In some situations, queries q_i and q_j have some overlap, which means that $compare$ is false, but partial reuse is still possible. Equation 2 describes the $overlap$ function that returns a value between 0 and 1 that represents the amount of overlap between intermediate data result $\mathcal{I}$ and $\mathcal{J}$. This overlap is computed with respect to some data transformation function $project$ that needs to be provided by the application developer. The $project$ function, seen in Equation 3, takes one intermediate data result $\mathcal{I}$ whose predicate is M_i and *projects* it (performs a transformation) to produce data product $\mathcal{J}$ with predicate M_j.

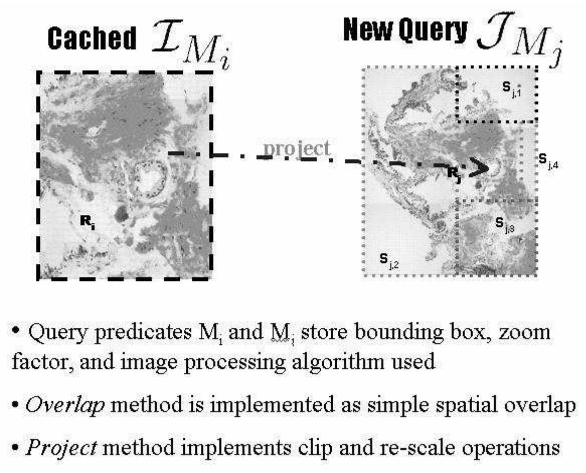

Figure 3. Once a new query q_j with meta-information M_j is submitted, the system tries to find a complete or partial match in cache that can be used to compute q_j. If a match is found (region R_i, in our example), a data transformation is applied via the user-defined `project` method to compute region R_j. Subqueries – $S_{j,1}$, $S_{j,2}$, $S_{j,3}$, and $S_{j,4}$ – are generated to complete the query processing and produce the complete result $\mathcal{J}$.

5 Multiple Query Processing Middleware

The architecture of the middleware consists of several service components, implemented as a C++ class library and a runtime system. The runtime system supports multithreaded execution on a cluster of shared-memory multiprocessor machines. In a cluster environment, each machine hosts a complete instance of the system, with all the service components available. Raw datasets are declustered across all the nodes to maximize parallelism and I/O bandwidth. Queries submitted to the system are broadcasted to all nodes, and each node computes the parts of the results that correspond to the local input data and cached results. At the end, a global merge phase is executed to combine results from each node. A complete description of the middleware can be found in [2, 3]. We briefly describe some of components of the middleware framework that are relevant to detection of common subexpressions and partial data reuse opportunities.

Query Server: The query server interacts with clients for receiving queries and returning query results, and is implemented as a fixed-size thread pool (typically the number of threads is set to the number of processors available on an SMP node). A query scheduler [4] is employed to dynamically order client requests for assignment to available threads. A client request contains a *query type id* and user-defined parameters to a query object that the application developer implemented. The user-defined parameters include a *dataset id* for the input dataset, *query meta-information*, and an *index id* for the index to be used for finding the data items that are requested by the query.

An application developer can implement one or more query objects that are responsible for application-specific subsetting and processing of datasets. The implementation of a new query object is done through C++ class inheritance and the implementation of virtual methods. A query object is associated with (1) an `execute` method, (2) a query meta-information object, and (3) an accumulator object, which encapsulates user-defined data structures for storing intermediate results. The `execute` method implements the user-defined processing of data. In the current design, this method is expected to carry out index lookup operations, the initialization of intermediate data structures, and the processing of data retrieved from the dataset. Both the query and accumulator meta-data objects are implemented by the application developer by deriving from a base class provided by the system.

When a query is received, the query server instantiates the corresponding query object and assigns a *Query Thread* to execute the query. The query thread searches for results cached in memory that can be reused to either completely or partially answer a query. The lookup operation employs the user-defined `overlap` operator to test for potential matches among those cached results. The user-defined accumulator meta-data object associated with the query object is compared with the accumulator meta-data objects of the cached results for the same query type. The user-defined `project` method is then called so that the cached result can be *projected*, potentially performing a transformation on the cached data, to generate a portion of the output for the current query. Finally, if the current query is only partially answered by the cached results, sub-queries are created to compute the results for the portions of the query that have not been computed from cached results.

Data Store Manager: The data store manager is responsible for providing dynamic storage space for intermediate data structures generated as intermediate or final results for a query. The most important feature of the data store is that it records semantic information about intermediate data structures (i. e. a semantic cache [17]). This allows the use of the intermediate results to answer queries later submitted to the system. A query thread interacts with the data store via functions similar to the C language function *malloc*. When a query allocates space in the data store for an intermediate data structure, the size (in bytes) of the data structure and the corresponding accumulator meta-data object are passed as parameters to the space allocator . The data store manager allocates the buffer space, internally records the pointer to the buffer space and the associated meta-data object, and returns the allocated buffer to the caller. The data store manager also provides the `lookup` method, to identify partial results that can be used to satisfy a new request. Since the data store manager maintains user-defined data structures and can apply projection operations on those data structure, user-defined projection methods may be provided for each type of intermediate data structure. If no data transformation is required, by default the projection method is the identity function.

Page Space Manager: The page space manager controls the allocation and management of buffer space available for input data in terms of fixed-size pages. All interactions with data sources are done through the page space manager. The pages retrieved from a Data Source are cached in memory. The page space manager also keeps track of I/O requests received from multiple queries so that overlapping I/O requests are reordered and merged, and duplicate requests are eliminated, to minimize I/O overhead.

6 Implementation of the Two Example Applications

6.1 The Virtual Microscope

A Virtual Microscope (VM) query describes a 2-dimensional region in a slide, and the output is a potentially lower resolution image generated by applying a user-defined aggregation operation on high-resolution image chunks (see Section 3). We have implemented two functions to process high resolution input chunks to produce lower resolution images in VM [4]. Each function results in a different version of VM with very different computational requirements, but similar I/O patterns. The first function employs a simple subsampling operation, and the second implements an averaging operation over a window. For a magnification level of N given in a query, the subsampling function returns every N^{th} pixel from the region of the input image that intersects the query window, in both dimensions. The averaging function, on the other hand, computes the value of an output pixel by averaging the values of a group of $N \times N$ pixels in the input image. We have added a query object to the runtime system for each of the processing functions. The accumulator for these functions is a 2-dimensional pixel array, each entry of which stores values for a pixel in the lower resolution output image. Each accumulator element a_e and each output element o_e can be described by the 3-tuple (bounding box, zoom factor, image processing algorithm) which constitutes

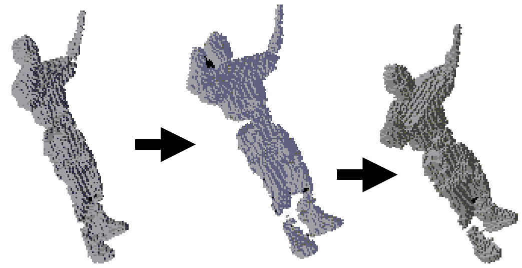

Figure 4. View of a volume from one perspective over 3 frames, computed by a volumetric reconstruction query into the system.

the query meta-information M (see Section 4). Input elements are cached in memory by the page space manager. Accumulator and output elements are cached in the data store and tagged with the appropriate meta-information. When a query enters the system, the algorithm in Figure 2 is executed. However, the operations at lines 5, 6, 8, and 10 are not immediately performed, instead a cache search into the data store is first performed, to find either a complete match (applying Equation 1), or if that fails a partial match (applying Equation 2). If a partial match is found, a projection function must be applied (Equation 3).

Several types of data reuse may occur for queries in the VM application. A new query with a query window that overlaps the bounding box of a previously computed result can reuse the result directly, after clipping it to the new query boundaries (assuming the zoom factors of both queries are the same). Similarly, a lower resolution image for a new query can be computed from a higher resolution image generated for a previous query, if the queries cover the same region. In order to detect such reuse opportunities, an `overlap` function was implemented to intersect two regions and return an overlap index, which is computed as

$$overlap\ index = \frac{I_A}{O_A} \times \frac{I_S}{O_S} \qquad (4)$$

In this equation, I_A is the area of intersection between the intermediate result in the data store and the query region, O_A is the area of the query region, I_S is the zoom factor used for generating the intermediate result, and O_S is the zoom factor specified by the current query. O_S should be a multiple of I_S so that the query can use the intermediate result. Otherwise, the value of the overlap index is 0.

6.2 Volumetric Reconstruction

A Volumetric Reconstruction query q_i is described by a query meta-information 5-tuple M_i:

1. a dataset name D_i,
2. a 3-dimensional box B_i: $[x_l, y_l, z_l, x_h, y_h, z_h]$,
3. a set of frames F_i: $[f_{start}, f_{end}, step]$,
4. the depth (number of edges from root the leaf nodes) of the octree, which specified the resolution of the reconstruction: d_i, and
5. a set of cameras C_i: $[c_1, c_2, \ldots, c_n]$.

Semantically, a query builds a set of volumetric representations of objects that fall inside the 3-dimensional box – one per frame – using a subset of the set of cameras for a given dataset (Figure 4). For each frame, the volumetric representation of an object is constructed using the set of images from each of the cameras in C_i. The reconstructed volume is represented by an octree, which is computed to depth d_i. Deeper octrees represent a higher resolution for the output 3-dimensional object representation.

Each individual image taken by a camera is stored on disk as a data chunk. A 3-dimensional volume for a single time step is constructed by aggregating the contributions of each image in the same frame for all the cameras in C_i into the output octree. The aggregation operations are commutative and associative. Thus, the images can be retrieved in any order and the octree is built incrementally by adding the contribution of each retrieved image. Note that it is also possible to create the final octree by having each one or a subset of cameras build a separate octree and then combine the partial octrees into a single output octree [10]. The final output is sent to the client for further analysis (e.g., visualization, object tracking).

In a multiple client environment, overlap and potential reuse opportunities among queries (submitted by one or more clients) and from previous queries executed by the system may be detected. One example of a reuse opportunity is the generation of a lower resolution octree from a higher resolution octree that was computed for an earlier query. In order to detect such possible overlaps, we implemented the following customizations for the *compare* (see Algorithm 1) and *overlap* functions (see Algorithm 2):

Algorithm 1 *bool* compare(M_i,M_j)

```
if D_i ≠ D_j then
    return false
if B_i ≠ B_j then
    return false
if F_i ≠ F_j then
    return false
if C_i ≠ C_j then
    return false
return true
```

Algorithm 2 *float* overlap(M_i,M_j)

if $D_i \neq D_j$ **then**
    return 0;
$v_{ovlp} \leftarrow \frac{CommonVolume(B_i,B_j)}{Volume(B_j)}$
$f_{ovlp} \leftarrow \frac{|F_i \cap F_j|}{|F_j|}$
if $C_i \supset C_j$ **then**
    $c_{ovlp} \leftarrow \frac{|C_i|}{|C_j|}$
else
    $c_{ovlp} \leftarrow 0$
$d_{ovlp} \leftarrow 1 - 0.1 \times (d_i - d_j)$
return $v_{ovlp} \times f_{ovlp} \times c_{ovlp} \times d_{ovlp}$

Our implementation of the volume reconstruction algorithm employs an earlier implementation [10] as a black-box, and that implementation returns an octree for each frame in a sequence of frames. Therefore, the data store maintains the octrees for each frame requested by a query along with its associated meta-information. The transformation of these cached results into results for incoming queries requires the utilization of *project* functions that transform the aggregate appropriately. Algorithm 2 hints at several projection operations: (1) for the *query box* – multiple volumes can be composed to form a new volume, or a larger volume can be cropped to produce a smaller one; (2) for entire *frames* – use the cached frames as necessary; (3) for *cameras* – if the new query requires more cameras than were used for a cached octree, generate a new octree from the images for the new cameras, and merge the two octrees; (4) *depth* – use a deeper octree to generate a shallower one. One or more combinations of these functions may be automatically applied using one or more cached results. The middleware generates the subqueries necessary to complete the processing of the original query accordingly.

7 Data Store Replacement Policies

Integral parts of the multiple query optimization framework are the query scheduling policy for incoming queries and the replacement policy employed in the semantic cache (i. e. the data store), which maintains intermediate (and final) results. Query scheduling and cache replacement policies are two complementary components. A good query scheduling policy attempts to order the execution of queries so that a query can benefit most from the cached data, without starving other waiting queries. In [4], we investigated a number of query scheduling policies using the LRU cache replacement policy. A good cache replacement policy complements the query scheduling policy by aiming to maintain in cache a *working set* of data items manipulated by the query workload. A replacement policy is essentially responsible for evicting a result stored in the data store when a new result must be cached and there is not enough space left for the new result.

7.1 Cache Replacement Policies

Least Recently Used (LRU): This policy replaces the intermediate result (aggregate) that has been requested least recently. The policy is based on the same principle as page replacement policies in operating systems. Every cached item is associated with a time stamp that stores the last time the item was accessed by a query, since the data server started execution. The item with the minimum time stamp is replaced when a new item must be stored in a full cache.

Size: Evicts the intermediate aggregate that occupies the largest space in the data store. This strategy attempts to maintain many aggregates with small memory footprints in cache, rather than a few results with large footprints. The premise of the Size strategy is that more queries are likely to benefit when a greater number of separate results, potentially generated for queries from different applications, are cached.

Least Frequently Used (LFU): This strategy evicts the intermediate aggregate which is accessed least frequently. It is based on the assumption that queries in a collaborative environment are likely to request the same or closely related regions of interest, with the same or similar processing requirements. Thus a cached result that has been reused by many queries is likely to be reused again. A reference count is associated with each cached data item. The count is incremented when the data item is reused in processing a query. The data item with the smallest count value is replaced with a new item when the cache is full.

Least Relative Value (LRV): This policy replaces the intermediate result that has the least *value*. The *value* metric can be computed in several different ways. Ideally, it should be a relative measure of how expensive it is to generate a given intermediate result. In this work, we have used two variants for calculating this metric. The first policy uses the ratio $\frac{q_{inputsize}}{aggr_{size}}$, where $q_{inputsize}$ is the number of bytes that have to be retrieved and processed from the raw input data to generate the intermediate aggregate, and $aggr_{size}$ is the amount of memory used by that aggregate. We refer to this method as LRVA. The second variant is the ratio $\frac{q_{ttc}}{aggr_{size}}$, where q_{ttc} is the time it takes to compute the intermediate result. We refer to this method as LRVB. The first variant is most suitable for I/O-bound queries, for which most of the execution time is spent retrieving the input data from disk. The second variant targets queries that are more compute-intensive, although q_{ttc} also accounts for I/O time.

7.2 Aging

A potential drawback of some of the cache replacement policies, in particular LFU, is that some cached items may be heavily used only during a limited time in the lifetime of the data server. As a consequence, those items will have a high reference count. In that case, those items may stay in the cache for a long time, even though they are no longer being used. *Aging* is a technique to alleviate this problem. It uses a decay function to decrease the reference count as the time passes. Several different implementations of this technique can be used. We have implemented an exponential *half life* factor to calculate the devaluation over time a cached aggregate observes. The factor is computed by the function $2^{-\frac{age}{T}}$, where age is the current age of the cached item (i. e. current time minus the last time the item was reused). T is the fixed and configurable half life, i. e., a constant that is set based on application and query characteristics as a reasonable period for half life decay. In T seconds, the metric for a given intermediate aggregate will decrease by half.

In this paper, we performed experiments with the following variations of the aging policies just described: ALFU (Aging LFU), ALRVA (LRVA with aging), and ALRVB (LRVB with aging).

8 Experimental Results

The goal of the experiments is to evaluate the cache replacement policies with different query workloads, using two query scheduling policies (First-In, First-Out (FIFO) and Shortest Job First (SJF)), and with several data store sizes. The experimental evaluation employed two implementations of the Virtual Microscope application – averaging and subsampling – and the Volumetric Reconstruction application on an 8-processor SMP machine, running version 2.4.3 of the Linux kernel. Each processor is a 550MHz Pentium III CPU and the machine has 4GB of main memory.

For the experiments, we employed three Virtual Microscope datasets, each of which is an image of size 30000x30000 3-byte pixels, requiring a total of 7.5GB storage space. Each dataset is partitioned into 64KB pages, each representing a square region in the entire image. The Volumetric Reconstruction dataset is composed of 400 frames synchronously collected from 13 cameras, each of which shot 400 images. Each image is a 320x240 1-byte black and white image. The dataset is approximately 381MB in size. All the datasets were stored on the local disk attached to the SMP machine.

For the first experiment shown in Figure 5, we have emulated 16 concurrent Virtual Microscope clients – 8 using the subsampling algorithm and 8 using the pixel averaging algorithm. Each client generated a workload of 32 queries, producing 1024x1024 RGB images (3MB in size) at various magnification levels. Output images were maintained in the data store as intermediate results for possible reuse by new queries. For each group of 8 clients, 4 clients issued queries to the first dataset, 3 clients submitted queries to the second dataset, and 1 client issued queries to the third dataset. Note that subsampled intermediate results cannot be used to generate averaged results and vice-versa.

We used the driver program described in [9] to emulate the behavior of a single client interacting with the data server, and generated 16 different client profiles. The implementation of the driver is based on a workload model that was statistically generated from traces collected from experienced VM users. Interesting regions in a slide are modeled as points, and provided as an input file to the driver program. When a user pans *near* an interesting region, there is a high probability a request will be generated. The driver adds noise to requests to avoid multiple clients asking for the same region. In addition, the driver avoids having all the clients scan the slide in the same manner. The slide is swept through in either an up-down fashion or a left-right fashion as observed from real users. We have chosen to use the driver for two reasons. First, extensive real user traces are very difficult to acquire. Second, the emulator allowed us to create different scenarios and vary the workload behavior (both the number of clients and the number of queries) in a controlled way. In all of the experiments, the emulated clients were executed simultaneously on a cluster of PCs connected to the SMP machine via 100Mbit Ethernet. Each client submitted its queries independently from the other clients, but waited for the completion of a query before submitting another one.

In the second experiment (whose results are in Figure 6), we used a workload composed of queries for the two implementations of the Virtual Microscope application and for the Volumetric Reconstruction application. We used a total of 16 clients (8 for subsampling, 4 for pixel averaging, and 6 for Volumetric Reconstruction). The 12 Virtual Microscope clients, each generating 16 requests, queried the same three datasets employed in the first experiment. Six clients accessed the first dataset, four the second one, and two the third dataset. The Virtual Microscope queries were generated using the same workload model as before. The 6 clients for the Volumetric Reconstruction application generated 8 queries each. Each client submitted queries constructed according to a synthetic workload model (since we do not have real user traces for the application at this time), in which "hot frames" were pre-selected, and the length of a "hot interval" was characterized by a mean and a standard deviation. A query (see Subsection 6.2) requests a set of volumes associated with frames selected with the following algorithm:

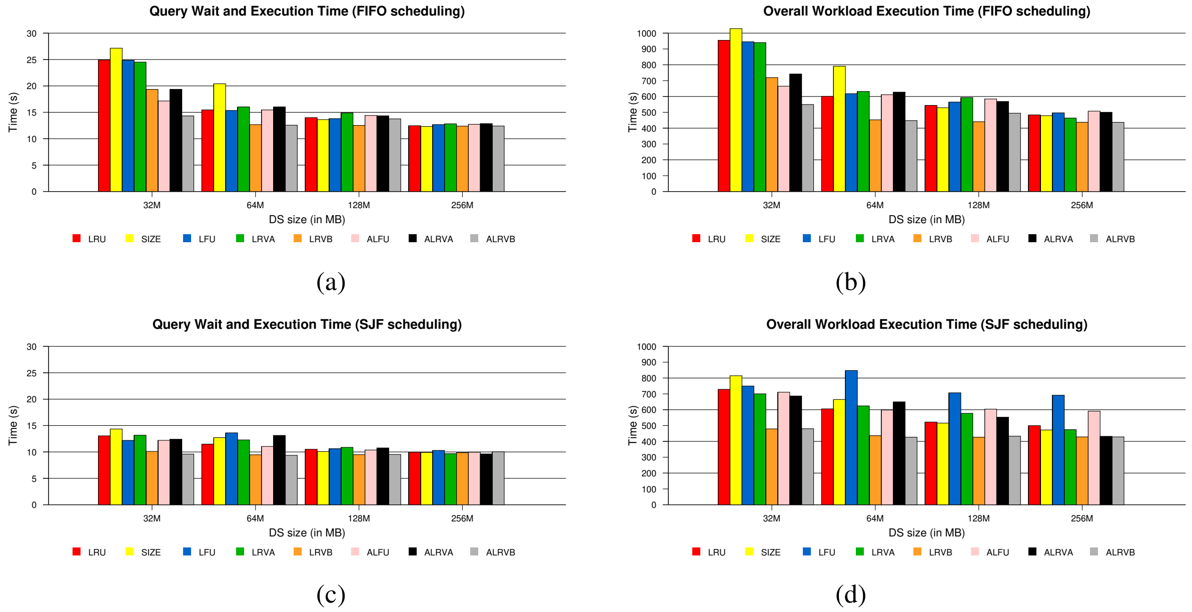

Figure 5. Results for Virtual Microscope queries only - subsampling and averaging implementations. The overall execution times for the complete workload are shown in (b) and (d), and the average execution times per query are shown in (a) and (c). (a) and (b) show times for queries scheduled using the FIFO policy and, (c) and (d) show times for queries scheduled using the SJF policy.

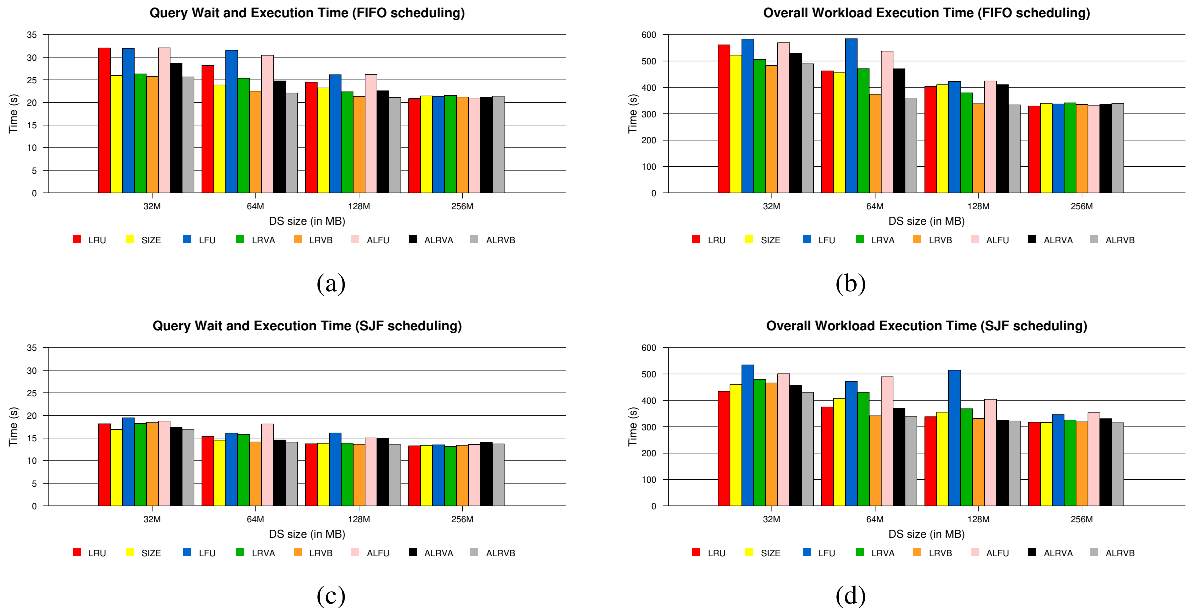

Figure 6. Results for queries from multiple applications (Virtual Microscope, averaging and subsampling implementations, and Volume Reconstruction queries). The overall execution times for the complete workload are shown in (b) and (d), and the average execution times per query are shown in (a) and (c). (a) and (b) show times for queries scheduled using the FIFO policy and, (c) and (d) show times for queries scheduled using the SJF policy.

the center of the interval is drawn randomly with a uniform distribution from the set of "hot frames", the length of the interval is selected from a normal distribution, and the $step$ value is selected randomly as either 1, 2, or 4. The depth and the 3-dimensional query box were fixed, as was the dataset, and we have used all the available cameras.

For all the experiments, we fixed the half life at 60 seconds for the replacement policies that use aging. We have allocated 32MB for the Page Space Manager, and have allowed a maximum of 4 queries to run simultaneously. The operating system file buffer cache was cleared at the beginning of each experimental run.

The primary difference between the first experiment and the second is the degree of overlap across the queries. The first experiment exhibits higher locality since only two applications are being run and more queries are generated (total of 512 queries, as opposed to 224 for the second). Hence the data store and the data transformation framework is more effective than in the second experiment. Nonetheless, in both experiments LRU is never the best replacement policy, neither in terms of average query wait and execution time, nor in terms of overall execution time for the complete workload. Performance improvements relative to LRU are as high as 40%, both for the average query wait and execution time and overall workload execution time, as seen in Figure 5(a) and (b). Even when SJF is used to schedule the queries, a 26% and 34% decrease is seen for average query wait and execution and overall execution time, respectively, as seen in Figures 5(c) and (d).

The relative benefits of the other replacement policies, in particular ALRVB, decrease as the size of the data store increases, and as locality decreases (i. e. in the second experiment), as Figures 5 and 6 show. The benefits of a more complex cache replacement strategy may even completely disappear, as Figure 6(d) shows when the workload workset is completely cached.

The performance results in Figure 5 show that when there is high locality, almost all cache replacement policies, with the exception of Size and LFU, outperform LRU for the FIFO scheduling policy. The same is not true for SJF scheduling, in which only the more "informed" policies (the LRV variations) outperform LRU.

In general, LRVB outperforms LRVA, and likewise for ALRVB and ALRVA. This is because their cost metric more precisely captures the *cost* associated with an aggregate when an eviction decision must be made. The time represented by q_{ttc} considers both the I/O and the computation cost for an aggregate, whereas $q_{inputsize}$ relies solely upon I/O costs. The same subsampling instance of the Virtual Microscope query takes 42.5 seconds to execute (99.6% of the time is I/O and 0.4% is computation) and require 3MB in storage, while averaging implementation takes 58.4 seconds (47.5% for I/O and 52.5% for computation) with the same storage requirement. One of our typical (for our workload) Volume Reconstruction queries takes 40.7 seconds (15.7% for I/O and 84.3% for computation) and 3.75MB of storage. With such disparate relative I/O costs and required storage size, q_{ttc} is definitely more accurate. The *aging* technique improves performance for all caching policies that employ it, for all the configurations we tested. The explanation, as we expected, is that it allows the data store to eventually evict aggregates that were once heavily reused but are not being reused any longer. We should note that more experiments should be performed to test other *half life* settings.

Overall the results show that the improvements achieved by using more sophisticated policies are significant, especially under severe space constraints (the 32MB data store size), confirming the results obtained by other researchers in other domains, as was discussed in Section 2.

9 Conclusions

The sheer volume of computation and I/O required by typical data analysis applications, as well as their reliance on non-standard aggregation operators, makes the task of providing multiple query optimization support a challenge. In previous work, we have described a generic middleware system that can be used for the implementation of such applications, allowing for the identification and utilization of common intermediate results. In this paper, we have demonstrated the importance of choosing cache replacement policies that consider the *relative value* of a given cached aggregate when eviction must be performed. Moreover, we have shown that *aging* has to be part of the solution to avoid having aggregates that were heavily reused at one point in time from taking up space in the cache when they are no longer part of the current working set.

We have proposed two metrics to be used when the lowest relative value cache replacement policies are employed. Although in all cases using the *time to compute* metric as the cost outperformed the estimated *query input size* metric, that metric is not without problems. The *time to compute* can be distorted by the very same characteristic that makes our system efficient in handling multiple query workloads – the data transformation functions leverage previously cached results to generate new ones. This fact causes the *time to compute* metric to under-value an aggregate, since it speeds up its computation, as opposed to truly representing the *time to compute* which would include the full I/O and computation costs. This is an issue still under investigation. We are analyzing methods for propagating the *time to compute* metric when a projection function is utilized.

Extensions to this work will come from several areas. The first is investigating the balance between query scheduling and cache replacement policies further by integrating additional scheduling policies, like the ones described in [4].

Self-tuning of the *aging* factor by inspecting the workload is another aspect worth studying. Finally, we would like to investigate the integration of a persistent cache into the framework, in which when an aggregate is selected for eviction it gets stored in a persistent medium (i.e. disk). This modification should make the system perform better overall, at the expense of more complexity in evaluating overlap and projection possibilities, as well as more complexity to manage a larger cache in persistent storage.

References

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

[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'01 Conference*, Denver, CO, November 2001.

[3] 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 International Symposium on Cluster Computing and the Grid (to appear)*, Berlin, Germany, May 2002. Also available as University of Maryland Technical Report CS-TR-4300 and UMIACS-TR-2001-78.

[4] 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 International Parallel and Distributed Processing Symposium (to appear)*, Fort Lauderdale, FL, April 2002. Also available as University of Maryland Technical Report CS-TR-4290 and UMIACS-TR-2001-68.

[5] M. Arlitt, R. Friedrich, and T. Jin. Performance evaluation of web proxy cache replacement policies. In *Proceeding of Performance Tools'98*, Palma de Mallorca, Spain, September 1998.

[6] R. H. Arpaci-Dusseau, E. Anderson, N. Treuhaft, D. E. Culler, J. M. Hellerstein, D. Patterson, and K. Yelick. Cluster I/O with River: Making the fast case common. In *Proceedings of the Sixth Workshop on I/O and Parallel and Distributed Systems*, Atlanta, GA, 1999.

[7] P. Baumann, A. Dehmel, P. Furtado, R. Ritsch, and N. Widmann. The multidimensional database system RasDaMan. In *Proceedings of the 1998 ACM-SIGMOD Conference*, pages 575–577, Seattle, WA, 1998.

[8] M. D. Beynon, T. Kurc, A. Sussman, and J. Saltz. Design of a framework for data-intensive wide-area applications. In *Proceedings of the 9th Heterogeneous Computing Workshop*, Cancun, Mexico, May 2000.

[9] M. D. Beynon, A. Sussman, and J. Saltz. Performance impact of proxies in data intensive client-server applications. In *Proceedings of the 1999 International Conference on Supercomputing*. ACM Press, June 1999.

[10] 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.

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

[12] C. F. Cerco and T. Cole. User's guide to the CE-QUAL-ICM three-dimensional eutrophication model, release version 1.0. Technical Report EL-95-15, US Army Corps of Engineers Water Experiment Station, Vicksburg, MS, 1995.

[13] C. Chang, A. Acharya, A. Sussman, and J. Saltz. T2: A customizable parallel database for multi-dimensional data. *ACM SIGMOD Record*, 27(1):58–66, March 1998. Also available as University of Maryland Technical Report CS-TR-3867 and UMIACS-TR-98-04.

[14] C. Chang, A. Sussman, and J. Saltz. Scheduling in a high performance remote-sensing data server. In *Proceedings of the Seventh SIAM Conference on Parallel Processing for Scientific Computing*. SIAM, March 1997.

[15] S. Chaudhuri. An overview of query optimization in relational systems. In *Proceedings of the ACM Symposium on Principles of Database Systems on Principles of Database Systems*, pages 34–43, Seattle, WA, 1998.

[16] S. Chippada, C. N. Dawson, M. L. Martínez, and M. F. Wheeler. A Godunov-type finite volume method for the system of shallow water equations. *Computer Methods in Applied Mechanics and Engineering*, 1997. Also a University of Texas at Austin TICAM Report 96-57.

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

[18] J. Dilley and M. Arlitt. Improving proxy cache performance: Analysis of three replacement policies. 3(6):44–50, November/December.

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

[20] K. Hahn. Parallel query support for multidimensional data. In *VLDB'2001 Workshop - Supercomputing Databases*, Rome, Italy, 2001.

[21] 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.

[22] T. M. Kurc, A. Sussman, and J. Saltz. Coupling multiple simulations via a high performance customizable database system. In *Proceedings of the Ninth SIAM Conference on Parallel Processing for Scientific Computing*. SIAM, March 1999.

[23] Land Satellite Thematic Mapper (TM). *http://edcwww.cr.usgs.gov/nsdi/html/landsat_tm/ landsat_tm*.

[24] K.-L. Ma and Z. Zheng. 3D visualization of unsteady 2D airplane wake vortices. In *Proceedings of Visualization'94*, pages 124–31, Oct 1994.

[25] 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.

[26] NASA Goddard Distributed Active Archive Center (DAAC). Advanced Very High Resolution Radiometer Global Area Coverage (AVHRR GAC)

data. *http://daac.gsfc.nasa.gov/CAMPAIGN_DOCS/LAND_BIO/origins.html*.

[27] GenBank of The National Center for Biotechnology Information (NCBI). www.ncbi.nlm.nih.gov/Genbank/genbankstats.html.

[28] V. Pai, P. Druschel, and W. Zwaenepoel. IO-Lite: A unified I/O buffering and caching system. In *Proceedings of the 3rd Symposium on Operating Systems Design and Implementation*, pages 15–28, New Orleans, LA, February 1999.

[29] Protein data bank. `http://www.rcsb.org/pdb`.

[30] R. Pottinger and A. Levy. A scalable algorithm for answering queries using views. In *Proceedings of the 26th VLDB Conference*, Cairo, Egypt, 2000.

[31] J. Robinson and M. Devarakonda. Data cache management using frequency-based replacement. In *Proceedings of 1990 ACM Conference on Measurement and Modeling of Computer Systems*, pages 134–142, University of Colorado, Boulder, CO, 1990.

[32] H. Samet. *The Design and Analysis of Spatial Data Structures*. Addison-Wesley, 1990.

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

[34] R. Szeliski. Rapid octree construction from image sequences. *Computer Vision, Graphics, and Image Processing. Image Understanding*, 58(1):23–32, 1993.

[35] T. Tanaka. Configurations of the solar wind flow and magnetic field around the planets with no magnetic field: calculation by a new MHD. *Jounal of Geophysical Research*, 98(A10):17251–62, Oct 1993.

[36] The Visible Human Project. www.nlm.nih.gov/research/visible/visible_human.html.

[37] 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.

[38] Y. Zhao, P. M. Deshpande, J. F. Naughton, and A. Shukla. Simultaneous optimization and evaluation of multiple dimensional queries. In *Proceedings of the 1998 ACM-SIGMOD Conference*, pages 271–282, Seattle, WA, 1998.