# Cooperative Caching for Grid Based Data Warehouses

Frank Dehne[1], Michael Lawrence[2], and Andrew Rau-Chaplin[3]

[1]Carleton University
School of Computer Science
Ottawa, ON, Canada
frank@dehne.net

[2]University of British Columbia
Department of Computer Science
Vancouver, BC, Canada
mklawren@cs.ubc.ca

[3]Dalhousie University
Faculty of Computer Science
Halifax, NS Canada
arc@cs.dal.ca

## Abstract

*In this paper we propose a grid-based OLAP application which distributes query computation across an enterprise grid. Our application follows a two-tiered process for answering queries based on sharing cached OLAP data between the users at the local grid site, and using grid scheduling approaches to execute the remaining parts of a query amongst a distributed set of OLAP servers. A new technique for extraction and aggregation of shared cached OLAP data is proposed, along with an efficient, aggregate-aware cache controller. An experimental evaluation of the proposed query processing and cooperative caching methods shows a significant reduction in query times compared to previously proposed methods.*

## 1 Introduction

The operation of modern distributed organizations, be they commercial, scientific, or health related, generate massive quantities of data. Decision makers increasingly construct large scale data warehouses and utilize On-Line Analytical Processing (OLAP) tools to glean from this rich data resource nuggets of information which can be used to better run their enterprises. A typical approach to OLAP based data warehouses is to construct a single centralized data repository by copying all of the raw data from the sites where it is generated to a central location, where it is integrated, and then to route all queries to that central location. As the amount of data and number of sites and users grow, this approach suffers from significant scalability problems.

More recently, distributed enterprises are adopting *grid computing* as a means of tackling computing problems requiring large amounts of computational power or reliable access to large amounts of data. There has been growing interest [7, 8, 15, 18, 19] in distributed data warehouses in the context of grid based computing resources.

In this paper we build on the grid OLAP model presented in [15] and propose new cooperative caching algorithms for grid based data warehouses. Our approach is to forgo the construction of a centralized data warehouse in favour of maintaining distributed data sources across a grid. In this context, queries must be routed to the appropriate data resources. Note that, unlike transaction processing queries, OLAP queries tend to involve large amounts of data aggregation and typically return large results. Fortunately, these results can often be used to help compute the answers to future queries as users roll-up their analysis.

Our approach is to take advantage of the hierarchal structure of a typical enterprise grid, blending new and sophisticated caching techniques and data grid scheduling to efficiently execute queries in a distributed fashion. Our aggregate-aware cache control algorithms take advantage of the hierarchal grid organization and the collection of local user's caches in order to reduce the amount of data retrieved from remote sites (see Figure 1).

This paper is the first to propose a *cooperative* caching strategy to speed up OLAP queries in the grid. While cooperative caching schemes exist (e.g. for Web Services [17] and P2P systems [11]), ours is the first that provides the ability to combine and aggregate cached data for future related OLAP queries. We believe that cooperative caching for OLAP amongst the users at a grid site is beneficial due to the likelihood that those users are interested in analyzing similar perspectives. We propose an efficient localized cache admittance scheme which uses a decay and refresh mechanism for controlling admission to and eviction from the cache, and a fast, aggregate-aware goodness metric for incoming OLAP view fragments.

We have prototyped the key components of our grid-based OLAP system in order to evaluate the effectiveness of the cache extraction and admission algorithms in comparison with recent OLAP caching proposals in the literature. Our experiments show that a significant reduction in query cost is achieved by sharing and aggregating locally cached

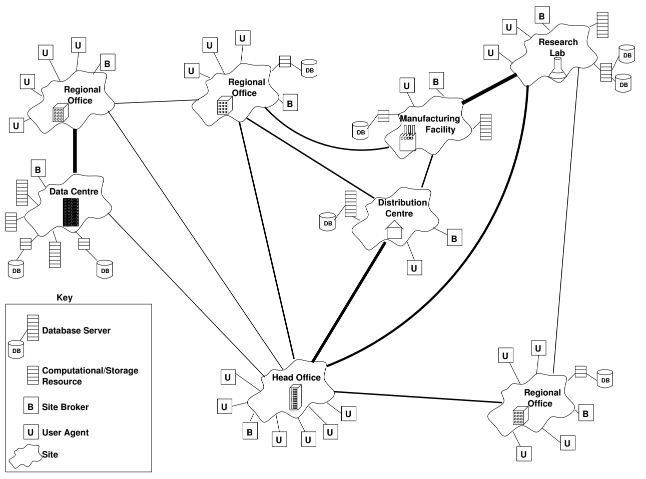

**Figure 1. An example OLAP enabled grid with the entities at each site and the connections between sites shown.**

data amongst users, and that our cache extraction method significantly outperforms previously proposed alternatives.

The remainder of this paper is organized as follows: Section 2 describes OLAP and gives an overview of our grid based OLAP application, also discussing related work. Section 3 reviews the architecture of the OLAP Enabled grid. Sections 4 and 5 describe the details of the query processing algorithm and caching strategies, respectively. An experimental evaluation is described in Section 6.

## 2 Background and Related Work

A typical data warehouse stores its information according to a star schema having a fact table with $d$ feature attributes (dimensions), and some number of measure attributes. For example, the fact table shown in Table 1 consists of 3 feature attributes (PRODUCT, CUSTOMER, and TIME) and a single measure attribute (AMT). In addition to the fact table, there are dimension tables which give further details about the dimensions. These details often define a hierarchy on the values of a dimension. For example, products in the bicycle store data warehouse of Table 1 may be grouped into categories such as *bicycles*, *parts* and *tools*.

A common type of query in OLAP data warehousing is the range-aggregate query, performed using the `SELECT` and `GROUP BY` clauses in the Standard Query Language (SQL). Typically the user selects a subset of the feature attributes from either the fact or dimension tables or both, and at least one measure attribute from the fact table with some aggregate function applied to it. The selected dimensions are used for grouping the results, and aggregation of the measure attribute(s) is applied over all records having identical values for the selected dimensions.

| PRODUCT | CUSTOMER | TIME | AMT |
|---|---|---|---|
| Trek 4900 | E. Rushton | 0418154212 | $650 |
| Kona Dew | O. Baltzer | 0418165122 | $900 |
| Tire Levers | O. Baltzer | 0418165122 | $2 |
| 26" Tube | G. Hickey | 0419092104 | $5 |
| Yellow Jersey | E. Rushton | 0419100515 | $85 |

**Table 1. The `SALES` table of a bicycle store data warehouse.**

For example, a user of the bicycle store data warehouse shown in Tables 1 may be interested to see how much each customer spent on each purchase. Such information can be obtained by an OLAP query selecting `CUSTOMER`, `TIME`, and `SUM(AMT)`, grouping the results by `CUSTOMER` and `TIME`, represented with the following SQL expression:

**SELECT** `CUSTOMER, TIME, SUM(AMT)`
**FROM** `SALES`
**GROUP BY** `CUSTOMER, TIME`

Aggregate queries in OLAP are categorized by the dimensions they choose in the `GROUP BY` clause, and the aggregated table resulting from such queries are called *views*. In the case that a query contains selection ranges on one or more of the dimensions, its results represent a view *fragment*. If a data warehouse has $d$ dimensions, and the number of elements in dimension $i$'s hierarchy is $H_i$ (where non-hierarchal dimensions $D$ have the size 2 hierarchy $D \rightarrow$ "all"), then the total number of views is $\prod_{i=1}^{d}(H_i)$ .

Harinarayan et. al. introduced the *data cube lattice* in [10], expressing the relationship between views as a partial order. Each view is a node, and there is a path from a view $v$ to a view $w$ in the lattice if queries on $w$ can be answered also using $v$. This occurs when $w$ groups on a subset of $v$'s dimensions, each at the same or lower levels of their respective hierarchies. More precisely, a view $v$ can be represented as a tuple of $d$ values $(v_1, v_2 \ldots, v_d)$, where $v_i$ is the level of dimension $i$'s hierarchy that $v$ groups on. The partial order amongst views as defined by the lattice is $v \preceq w$ iff $v_i \preceq_i w_i$, where $\preceq_i$ is the partial order defined by dimension $i$'s hierarchy. The complete data cube lattice for the bicycle store data warehouse is shown in Figure 2. A fragment of a view $v$ (resulting from a query on $v$) can be aggregated to produce fragments on descendants of $v$ so long as it contains the entire range of values for those dimensions which are aggregated out.

There has been recent related work on grid based OLAP applications [2, 18, 19], OLAP aware caching [13], and distributed caching in P2P and Web Services settings [11, 17]. In [2], a grid application for performing data mining and OLAP tasks on heterogeneous health care data sources is described. The focus here is primarily on the application

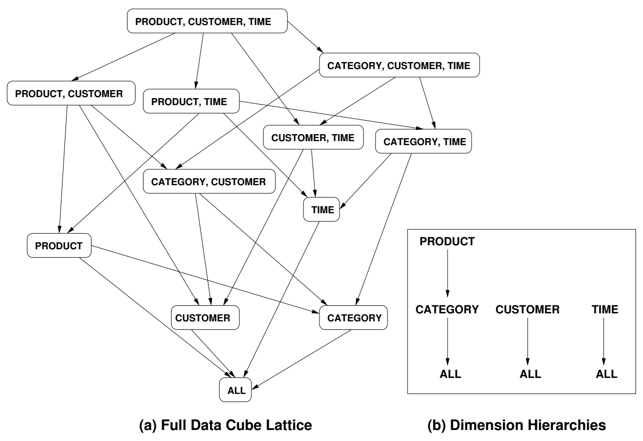

**Figure 2. The full data cube lattice and dimension hierarchies for the bicycle store data warehouse shown in Table 1.**

and data integration issues, rather than the efficiency of the OLAP processing. In [18, 19] the focus is on the challenging problem of building an OLAP datacube in a grid environment. Although query processing is briefly addressed, the proposed approach is quite simplistic. It does not make use of cached results which, we believe is the key to efficiency in the grid OLAP setting, because of the high network latency and relatively low bandwidth between widely geographically dispersed grid entities.

OLAP data is unique from a caching perspective in that the results of some queries (fragments) can be used to compute some or all of the results of queries on different views of the lattice. Kotidis and Roussopoulos [13] take advantage of this by designing a cache which can further aggregate cached fragments for producing a query result. However, their approach is all or none, in that either the entire query result is obtained by aggregating a single cached fragment, or not at all. They do not consider aggregating and combining multiple cached fragments to answer a query, and they do not consider fragmenting a query and answering part of it from cache and part from the backend. In [6], this is relaxed by partitioning each view into discrete chunks, and caching and querying on a chunk based level. However, this requires special indexes and functionality at the back end.

Distributed caching approaches have been examined in P2P and Web Services settings [11, 17]. In [11], processing of OLAP queries in peer-to-peer networks is considered. They use the chunk based caching scheme of [6], and peers propagate requests for chunks amongst each other and data warehouses. In [17] from CCGrid'06, in-memory caching of web objects in large scale data centers is considered. Each node maintains an index of the other's caches, which they use to cooperatively answer requests.

## 3 The OLAP Enabled Grid

### 3.1 Entities

Our application, the OLAP Enabled Grid, is designed based on the observation that the structure of an enterprise grid is typically hierarchal: there are a number of sites in the organization, each having a number of computational entities. Each site is a location where the enterprise has operations, and it is the case that transmission within a site is much faster than transmission between sites (e.g. LAN vs. WAN transmission). The entities at a site are attached computers (sequential or parallel) which are able to participate in the OLAP process, for example a user or a database server. An illustration showing the various entities present in an example enterprise is given in Figure 1.

As can be seen in the figure, each entity has a role according to the functionality it offers in the OLAP Enabled Grid. There are a total of four different roles:

1. *Database Server* - A machine which manages an operational database in the enterprise. The data maintained by different Database Servers is independent and follows a common warehouse schema.
2. *Computational/Storage Resource* - A machine which offers storage space and processing power to the grid.
3. *Site Broker* - Responsible for the organization and coordination of resources within that site.
4. *User Agent* - The workstation of a user who is interested in performing OLAP operations on the data managed by the Database Servers. Each User Agent has an amount of cache space on disk for storing the results of previously answered queries.

### 3.2 System Architecture

An overview of the logical components of the proposed system from the perspective of a user is shown in Figure 3. The corresponding layers of the Open Grid Services Architecture presented in [9] are shown as well. In this section we give a brief introduction to the role of each component in the system. The details of query processing are described in the sections which follow.

The user interacts with a Front End which displays query results and translates requests into OLAP queries which are answered by the Query Service. The Query Service uses the Distributed Cache Index Service: a global index implemented on the Site Broker of all cached fragments on the local site. Having such an index allows as much of the query as possible to be answered by local data. We could have also followed the approach of [17] and put a cache index on each User Agent. However, given the possibility of a large number of User Agents and a high degree of

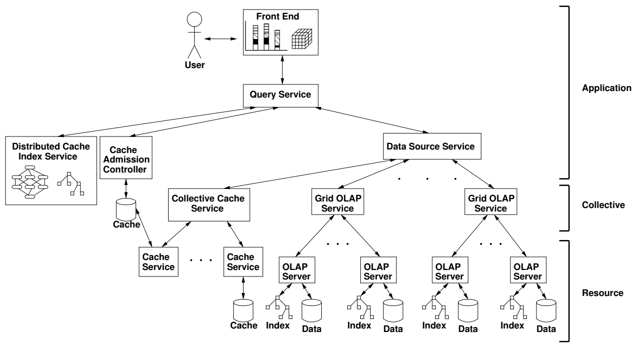

**Figure 3. Components of the proposed grid-based OLAP solution, and the corresponding Open Grid Services Architecture layers as presented in [9].**

query fragmentation, this would likely result in a very large number of messages between User Agents to keep the cache indexes up to date. We could have forgone an index as in the P2P system of [11], but this does not guarantee that the maximum amount of local data will be used.

The Query Service uses the Data Source Service to obtain both cached fragments as well as query results from the backend, which in turn uses a Collective Cache Service for the cached fragments and various Grid OLAP Services for obtaining the parts of the query which are not cached. There is one Grid OLAP Service corresponding to the data of each Database Server in the grid. However, due to the presence of other computational and storage resources on a site, there may be multiple underlying OLAP servers which can answer an OLAP query on that data. In our implementation, the OLAP Servers store the data in a normalized multidimensional format using R-Trees as indices as described in [3–5]. For a particular Database Server, the Site Broker at its site implements the corresponding Grid OLAP Service, whose job is to choose which of the OLAP Servers will answer a given query. Which Grid OLAP Services a particular sub-query goes to depends on the selection ranges of that sub-query. The data is partitioned across Grid OLAP Services horizontally by dimension values, and a sub-query is sent to each Grid OLAP Service whose range of values intersect with that of the sub-query.

As in the Collective Cache Service, the Data Source Service immediately forwards results of sub-queries back to the Query Service as soon as they are received. This is to allow the Cache Admission Controller (described in Section 5) to make caching decisions on each fragment while the Query Service is waiting for the remaining fragments, rather than trying to do them all at once. When all of the results of the sub-queries have been obtained by the Query Service, the overall query result is constructed and returned to the Front End for display to the user.

## 4 Query Processing Algorithms

This section describes the basic steps taken in order to execute a user's OLAP query on the grid. Referring to Figure 3, this involves the description of the Query Service, Distributed Cache Index Service, Collective Cache Service, Data Source Service and Grid OLAP Service. Based on our sharing of local caches, a two-tiered process for answering queries is proposed. The first tier uses the caches on the local site in a collaborative manner to answer as much of the query as possible, and the second tier requests the missing fragments from the OLAP servers.

The first tier involves using the Distributed Cache Index Service on the local Site Broker to find all of the locally cached fragments which can be used to answer parts of the query. The Query Service uses knowledge of these fragments to determine a set of sub-queries requesting the remaining fragments. This information is called a *fragmentation plan*. A fragmentation plan is constructed as follows: given a query $q$ on a view $v$, it first finds all cached fragments on $v$ which intersect with the selection range of that query. It then calculates a set of sub-queries representing the parts of $q$ which do not intersect with the retrieved fragments, and checks for cached fragments on each of $v$'s parents in the data cube lattice which can be aggregated to answer parts of these sub-queries. The search then proceeds recursively from their parents. A full description of our query processing and cache extraction algorithms is in [14].

Once constructed, the fragmentation plan is sent back to the Query Service on the User Agent, which asks the Data Source Service to retrieve the data outlined in the fragmentation plan. The Data Source Service sends the cache requests from the fragmentation plan to the Collective Cache Service, and is able to identify which Grid OLAP Services to contact for each sub-query.

In the second tier, a Grid OLAP Service of a Site Broker receives a query. It polls each OLAP Server, requesting estimates of how quickly they can answer the given query. Once the OLAP Servers have responded, the query is sent to the OLAP Server with the smallest estimated response time. The results are sent back to the Data Source Service of the User Agent which requested them. As in [20, 21], the query is scheduled on the OLAP Server which estimates it can answer it and return the results to the user the quickest.

## 5 Cooperative Caching Algorithms

In this section the operation of the Cache Admission Controller implemented on each User Agent is described. Given the potentially large amount of work that goes into computing fragments to answer a query, the key to an efficient grid based OLAP system is an effective caching scheme. The challenge in our case is how to assign a value representing the "goodness" of caching a particular fragment, since fragments which can be computed locally are presumably not as valuable as fragments computed on remote sites. The caching process is for the User Agent to compute the goodness of each fragment and make a local caching decision upon receiveing it, notifying the Distributed Cache Index Service on the Site Broker of changes to its cache contents. The disadvantage to this is that there will be many small fragments in the cache, possibly increasing the complexity of the Distributed Cache Index Service and the number of subqueries it returns. However, the advantage is that since there may be vast differences in the cost of obtaining the various subfragments and in their benefit to queries on the local site, better use of the cache space can be made by only caching the most valuable subfragments thereby also reducing the overlap of fragments in the cache.

When cached data is aggregated, accurate caching in OLAP is a difficult problem. There are many ways in which the goodness value of a fragment can be assigned, ranging in the tradeoff they provide between accuracy and speed. For example, fast and rough measures of computing goodness of a fragment could be the cost to retrieve it [6]. A slower and more exact measure is the Benefit Per Unit Space (BPUS) goodness used in [12,16], whose complexity is quadratic in the number of views in the data cube lattice. There is also the added disadvantage in our scenario that it requires the User Agent to have knowledge about all of the fragments cached by other User Agents on its site.

Our goal was to devise a goodness measure in combination with a caching strategy which is efficient yet still takes into account the benefit of further aggregating a cached fragment. The proposed caching strategy maintains a priority queue of fragments in increasing order of goodness. Each time the cache is accessed, the goodness of all items in the cache is decreased (either by subtracting or dividing by a fixed amount), except for the one which is accessed, which has its goodness reset back to the original value and is repositioned in the priority queue. A fragment $f$ is admitted to the cache if space can be made for it by evicting a set of fragments whose total goodness is less than that of $f$. Cache admission is described in Algorithm 1.

The purpose of this strategy, particularly with the decaying/refreshing of goodness values over time is that it adapts to the changing query demands of the users (e.g., view $v$ is queried less, while view $w$ begins to be queried more) but also accounts for the later caching of descendants of a fragment. For example if a fragment $f$ is frequently aggregated in cache for the purpose of computing fragments on descendant views, its goodness will frequently be refreshed to the original value and $f$ will have a low chance of being

**Algorithm 1** Cache Admission

```
Input: Fragment $f$ with goodness $g(f)$ and storage size
    $s(f)$. Priority queue $Q$ of cached fragments of total
    size $s(Q)$. Maximum cache size $S_{max}$.
Output: Set of fragments $F$ which have been evicted from
    the cache.
 1: $F \leftarrow \emptyset$
 2: $g(F) \leftarrow 0$
 3: $s(F) \leftarrow 0$
 4: while $S_{max} - s(Q) + s(F) < s(f)$ do
 5:     $F \leftarrow F \cup \{f_{min}\}$
 6:     $f_{min} \leftarrow Q.dequeue()$
 7:     $g(F) \leftarrow g(F) + g(f_{min})$
 8:     $s(F) \leftarrow s(F) + s(f_{min})$
 9: end while
10: if $g(f) < g(F)$ then
11:     $F \leftarrow \{f\}$
12: else
13:     $Q.enqueue(f)$
14: end if
```

evicted from cache. However if some fragments which are descendants of $f$ become cached and used instead, then $f$ will be accessed less often, its goodness decaying until it is eventually evicted.

The goodness of a fragment should reflect the savings in query cost it provides at all levels of aggregation. This depends on both the quantity of savings and the relative frequency with which it is expected to occur. The quantity of savings in query cost that $f$ provides is the relative difference between the cost it took to get $f$ and the cost of answering future queries on $f$. Hence we define

$$savings(f) = cost(f) - query\_cost(f) \qquad (1)$$

Since the User Agent which requested $f$ is the one making the caching decision, it can accurately compute $cost(f)$ by recording the time taken to retrieve it. A User Agent will likely store the records in $f$ on a contiguous space on disk without any specialized index, so $query\_cost(f)$ is modeled as the time it takes to scan $f$ from disk (the size of $f$ in bytes divided by the disk bandwidth of the user storing it).

The benefit of aggregating $f$ to other views needs to be taken into account. For example a small fragment of one of the lower level views provides a large savings in query cost, but to only a very small proportion of all possible queries. By contrast, a large fragment of one of the higher level views provides a smaller savings in query cost, but can be used to compute a much larger proportion of the possible queries. We call the proportion of the data cube lattice that a fragment $f$ covers in the feature dimension space at all levels of aggregation the *volume* of $f$. For example if a fragment $f$ covers half of the multidimensional area of a view $v$, but cannot be aggregated to produce fragments on any of $v$'s descendants, then $f$'s volume is $1/(2 \times num\_views)$. To describe how the volume of a fragment is computed, first consider the simple case of a data cube lattice with no dimension hierarchies. We use the same notation for fragments as in [13], which is the same for that of a view (Section 2), except associated with each dimension is an interval $I_i$ specifying the range of values that the fragment contains for dimension $i$. A fragment $f = ((I_1, h_1), (I_2, h_2), \ldots (I_d, h_d))$ can be aggregated into a fragment on a view $v = (a_1, a_2, \ldots a_d)$ if, for each $i$ such that $h_i = D_i$ and $a_i =$ all, we have $I_i = (min(D_i), max(D_i))$. Hence, if we let $GD(f)$ be the set of all $i$ such that $h_i = D_i$ and $I_i = (min(D_i), max(D_i))$ (GD = "Global Dimensions"), then $f$ can be aggregated into fragments on a total of $2^{|GD(f)|}$ views. On each of these views, the fraction of multidimensional space that $f$ covers is given by the product of the proportion of each dimension's range selected by $f$, i.e.

$$\prod_{i:h_i \neq \text{all}} \frac{max(I_i) - min(I_i)}{max(D_i) - min(D_i)}$$

Hence, for the case with no dimension hierarchies, we have

$$volume(f) = \frac{2^{|GD(f)|}}{2^d} \prod_{i:h_i \neq \text{all}} \frac{max(I_i) - min(I_i)}{max(D_i) - min(D_i)} \tag{2}$$

of the total space of the data cube covered by a fragment $f$ at all levels of aggregation.

When there are dimension hierarchies the volume calculation is slightly less straightforward, since it is not necessary to have $I_i = (min(D_i), max(D_i))$ to aggregate along a hierarchal dimension $D_i$. For example, all products do not need to be selected to produce the aggregate for a particular subset of categories. Furthermore, the actual proportion of a dimension's range which is selected by a fragment is slightly different at each level of the dimension's hierarchy. For example, if there are four equally sized product categories, then selecting products covering half of the first and last categories and all of the second and third categories covers 75% of the entire product dimension. However moving down the hierarchy to the category level only allows aggregation of two categories, covering only 50% of its range. The complete volume calculation for the case with dimension hierarchies requires enumerating all of the views to which $f$ can be aggregated, and examining the corresponding selection range on these levels. This increases the complexity of the volume calculation from linear in the no dimension hierarchies case, to exponential when there are hierarchies.

We approximate the true volume for the case of dimension hierarchies by using the selection range on a hierarchal dimension to approximate the corresponding selection range on all levels of the hierarchy. Hence, as in the non-hierarchal case, we multiply the number of possible aggregates of $f$ by the product over all non global dimensions of the fraction of the range selected of that dimension, approximating the proportion of each of the views covered. To enumerate the number of possible views which the fragment can be further aggregated on, we count the number of levels $l(h_i)$ below the selected level $h_i$ of a dimension $D_i$, including $h_i$ itself. For non-hierarchal dimensions $D_i$, where either $h_i = D_i$ or $h_i =$ all, we have $l(D_i) = 2$. The number of possible aggregates of $f$ is then

$$\prod_{i \in GD(f)} l(h_i) \prod_{i \notin GD(f)} (l(h_i) - 1)$$

and the total volume of $f$ is then

$$volume(f) = \frac{\prod_{i \in GD(f)} l(h_i) \prod_{i \notin GD(f)} l(h_i) - 1}{num\_views} \times \prod_{i:h_i \neq \text{all}} \frac{max(I_i) - min(I_i)}{max(D_i) - min(D_i)} \tag{3}$$

The final goodness measure is

$$goodness(f) = \frac{volume(f) \times savings(f)}{size(f)} \tag{4}$$

Which gives a measure of the savings in query cost a fragment provides, weighted by the volume representing the probability with which these savings are expected to be applied, scaled to the storage space of the fragment.

## 6 Experimental Results

In this section we describe a prototype of key components of the OLAP Enabled Grid and a careful evaluation of their performance with respect to caching. In the remainder, we refer to our cache extraction strategy as Fragment Aggregation and Recombination (FAR), since it both aggregates cached fragments as well as attempts to recombine multiple cached fragments for answering a query. FAR is compared in the context of the grid to the caching approach of Kotidis and Roussopoulos [13], which we will refer to as Fragment Aggregation (FA). Their approach, like ours, will search up the lattice looking for fragments which can be aggregated to produce a query result, however it differs in that a query must be answered by exactly one such cached fragment and will not be broken further into sub-queries. Whereas Kotidis et al. suggest this does not pay, our results show that it can indeed be of substantial benefit to the users on a grid site under realistic circumstances.

We have fully implemented the Query Service, Distributed Cache Index Service, Cache Admission Controller, Collective Cache Service, and Data Source Service. The

Cache Services, Grid OLAP Services (both local and remote) and OLAP Servers are simulated. The result is a working implementation of Tier 1 query processing and caching on a single site, with the other sites and data being simulated by single OLAP Server entities. Our implementation is a parallel program written in the Python scripting language, with communication between the entities (Site Broker, Users and OLAP Servers) being achieved with MPI.

## 6.1 Experimental Setup

For all our tests, the lattice used has 5 feature dimensions and a single measure dimension *sales*. One of the dimensions has a 5-level non-linear time hierarchy, while the other two dimensions have 2 and 3 level linear hierarchies respectively. The total number of rows in the fact table is 10 million, resulting in a lattice with 288 views totaling 35 GB in storage size.

We use two different types of query distributions in the experiments. The *uniform* distribution spreads queries uniformly amongst views and their selection ranges amongst dimension values. This is a difficult query load for caching as there is no relationship between queries whatsoever. The *hot region* distribution used in [11] represents a more realistic scenario where a subset of the aggregates are of particularly high interest to the users. In the hot region distribution a large percentage of the queries are distributed amongst a small set of views (the "hot region"). We also distribute the selection ranges according to a hot region.

Each User Agent is configured with a specified cache size, a disk bandwidth, a query stream, and optionally a list of fragments initially filling its cache. Each OLAP Server is configured with a disk bandwidth, a network bandwidth to the local site, a fragment of the fact table which specifies the partition of the overall data maintained by that OLAP Server, and a list of materialized views at that OLAP Server.

Most of our tests use the Detailed Cost Savings Ratio (DCSR) measure [13]

$$DCSR = \frac{\sum_q (time_{nocache}(q) - time_{cache}(q))}{\sum_q time_{nocache}(q)}$$

which measures the reduction in overall query time achieved by the cache as a percentage of query time without a cache. In order to achieve this we also implemented a version of the system with no caching components.

## 6.2 DCSR vs Cache Size

The first set of tests aims to determine the cache search strategies' ability to make effective use of increasing cache space. The parameters for this experiment were as follows: Number of dimensions = 5 with 3 hierarchical, Lattice size = 35GB with 288 views, Duration of runs = 2 Hours, Number of User Agents = 10, Queries per User Agent = 10, Query distribution = Hot regions, User disk bandwidth = 20 MB/s, Average query result size = 3.34 MB, Cache size per User Agent = 50 to 500 MB, Number of OLAP Servers = 5, OLAP Server disk bandwidth = 80 MB/s, OLAP Server materialized views = 14 randomly chosen, OLAP Server network bandwidth = 1 local (900 kb/s), 4 remote (100 kb/s).

The DCSR of both FAR and FA as cache size per User Agent is increased, averaged over 5 independent runs, is shown in Figure 4. Each cache was initially warmed with random fragments drawn from the query distribution. The FAR strategy allows a significant query time reduction of 50% to 60% for caches between 50 and 250 MB in size. For larger cache sizes the benefits of the FAR approach begin to wane due to the increased cost of the recursive lattice cache search and number of separate requests which must be made for each query, although it is still more beneficial than the FA approach up to a cache size of 350 MB per user.

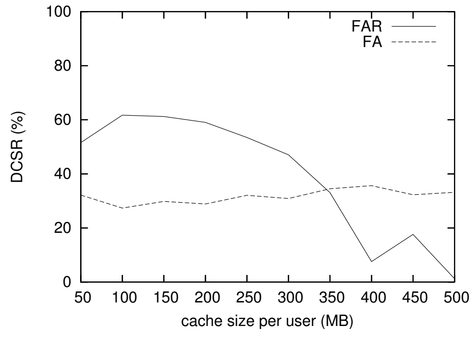

**Figure 4. Cost savings of FAR and FA as cache size per user is increased.**

## 6.3 Cache Warming

The previous tests were performed with the cache preloaded with a set of fragments drawn from the same distribution as the queries themselves. It is also important to examine the behaviour of a system starting with a cold cache, and how this changes over time as the cache warms up.

For the cache warming experiment, the same parameters are used as in the previous, except the cache sizes are fixed at 100 MB for each user, and the run lasts 8 hours during which time each user issues 40 queries. 10 independent runs are performed and the cost savings for each query is measured in the sequence.

The results are shown in Figure 5. For the FAR strategy, there is a general trend towards a higher cost savings for the later queries in the sequence, suggesting that they benefit from the cached results from earlier queries. The results are quite noisy for the reason that each point on the plot is computed from the results of a set of 100 queries, all of which are different from that of each other point on the plot. In contrast, the previous tests only varied the cache sizes while the queries remained the same. We observed a large sample variance between the times of individual queries, resulting in highly variable cost savings. The linear best fit function shown on the plot does yield the conclusion that for the FAR strategy there is a savings and that it does increase as the cache is filled, but we also observe that the quantity of savings appears to depend as much on the specifics of the query than on the fullness of the cache.

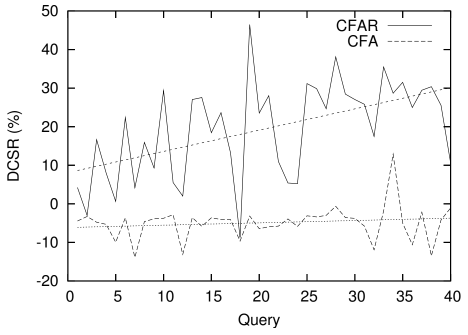

**Figure 5. The DCSR of the FAR and FA cache search over a sequence of 40 queries.**

## 6.4 Uniform Queries

The same set of tests as in Section 6.2 have been performed, this time using queries from the uniform random distribution. This is a much less favourable situation for caching due to the lack of relationship amongst the queries. The results, not shown here due to space limitations, are that a substantial cost savings can still be achieved by the FAR strategy. It also scales much better with increasing cache sizes in this case as compared to the hot region query distribution. For FAR, we see a DCSR which steadily increases from roughly 27.6% to 51.5% for 50MB and 500MB caches respectively, where as for FA this savings ranges from 22.6% to 24.6%. Further analysis shows that the improvement to FAR for larger cache sizes is due to the substantially smaller cache search time, illustrated in Figure 6.

## 7 Conclusions and Future Work

We have presented a cooperative caching scheme for the OLAP Enabled Grid in which the user caches distributed amongst the grid sites cooperate to increase the efficiency of OLAP query processing. We have proposed an efficient localized cache admittance scheme which uses a decay and refresh mechanism for controlling admission to and eviction from the cache, and a fast, aggregate-aware goodness metric for incoming fragments. We have experimentally evaluated our caching scheme comparing it against one which does not recombine multiple fragments to answer a query, and found our strategy to result in a much higher savings in query time for a reasonable total cache size indexed, even when the queries are uniformly distributed. Given that our prototype implementation performs well, the natural next step is to explore the implementation of the OLAP Enabled grid using a standard grid toolkit such as Globus, using the WS-DAI standard for grid database access as set by the Global Grid Forum [1].

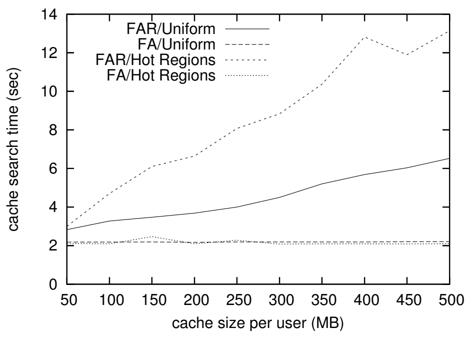

**Figure 6. Cache search time of FAR and FA for hot region and uniform queries as cache size per user is increased.**

## References

[1] M. Antonioletti, M. Atkinson, A. Krause, S. Laws, S. Malaika, N. W. Paton, D. Pearson, and G. Riccardi. Web services data access and integration - the core (WS-DAI) specification, version 1.0, June 2006.

[2] P. Brezany, A. M. Tjoa, M. Rusnak, J. Brezanyova, and I. Janciak. Knowledge grid support for treatment of traumatic brain injury victims. In *Proc. ICCSA'03*, 2003.

[3] Y. Chen, F. Dehne, T. Eavis, and A. Rau-Chaplin. Parallel ROLAP data cube construction on shared-nothing multiprocessors. *Distr. and Par. Databases*, 15:219–236, 2004.

[4] F. Dehne, T. Eavis, and A. Rau-Chaplin. Parallel multi-dimensional ROLAP indexing. In *Proc. CCGrid'03*, pages 86–93. IEEE, 2003.

[5] F. Dehne, T. Eavis, and A. Rau-Chaplin. The cgmCUBE project. *Distr. and Par. Databases*, 19(1):29–62, 2006.

[6] P. Deshpande and J. F. Naughton. Aggregate aware caching for multi-dimensional queries. In *Proc. EDBT'00*, 2000.

[7] W. Dubitzky, D. McCourt, M. Galushka, M. Romberg, and B. Schuller. Grid-enabled data warehousing for molecular engineering. *Par. Comp.*, 30(9-10):1019–1035, 2004.

[8] B. Fiser, U. Onan, I. Elsayed, P. Brezany, and A. M. Tjoa. On-line analytical processing on large databases managed

by computational grids. In *Proc. DEXA'04*, pages 556–560, Washington, DC, USA, 2004. IEEE Comp. Soc.

[9] I. Foster, C. Kesselman, and S. Tuecke. The anatomy of the Grid: Enabling scalable virtual organizations. *J. of High Performance Comp. Applications*, 15(3):200–222, 2001.

[10] V. Harinarayan, A. Rajaraman, and J. D. Ullman. Implementing data cubes efficiently. In *Proc. SIGMOD'96*, 1996.

[11] P. Kalnis, W. S. Ng, B. C. Ooi, D. Papadias, and K.-L. Tan. An adaptive peer-to-peer network for distributed caching of olap results. In *Proc. SIGMOD'02*, 2002.

[12] P. Kalnis and D. Papadias. Proxy-server architectures for olap. In *Proc. SIGMOD'01*, pages 367–378, New York, NY, USA, 2001. ACM.

[13] Y. Kotidis and N. Roussopoulos. A case for dynamic view management. *ACM Trans. Database Syst.*, 26(4):388–423, 2001.

[14] M. Lawrence. An Architecture and Caching Strategies for Grid-Enabled OLAP. Master's thesis, Dalhousie University, September 2006.

[15] M. Lawrence and A. Rau-Chaplin. The OLAP-enabled grid: Model and query processing algorithms. In *Proc. HPCS'06*, 2006.

[16] T. Loukopoulos, P. Kalnis, I. Ahmad, and D. Papadias. Active caching of on-line-analytical-processing queries in www proxies. In *Proc. ICPP'01*, pages 419–426. IEEE, 2001.

[17] S. Narravula, H.-W. Jin, K. Vaidyanathan, and D. K. Panda. Designing efficient cooperative caching schemes for multitier data-centers over rdma-enabled networks. In *Proc. CCGrid'06*, pages 401–408, Los Alamitos, CA, USA, 2006. IEEE.

[18] T. Niemi, M. Niinimaki, J. Nummenmaa, and P. Thanisch. Constructing an OLAP cube from distributed XML data. In *Proc. DOLAP'02*, 2002.

[19] T. Niemi, M. Niinimaki, J. Nummenmaa, and P. Thanisch. Applying grid technologies to XML based OLAP cube construction. In *Proc. DMDW'03*, 2003.

[20] S. Orlando, P. Palmerini, R. Perego, and F. Silvestri. Scheduling high performance data mining tasks on a data grid environment. In *Proc. Euro-Par'02*, 2002.

[21] S. Park and J. Kim. Chameleon: a resource scheduler in a data grid environment. In *Proc. CCGrid'03*. IEEE, May 2003.