Implementing OLAP Query Fragment Aggregation and Recombination for the OLAP Enabled Grid

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

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

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

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

Abstract

In this paper we propose a new query processing method for the OLAP Enabled Grid, which blends sophisticated cache extraction techniques and data grid scheduling to efficiently satisfy OLAP queries in a distributed fashion. The heart of our approach is our query Fragment Aggregation and Recombination (FAR) strategy that partitions OLAP queries into subqueries which can be effectively answered by retrieving and aggregating *multiple fragments of cached data from nearby grid sources, or as a last resort, more remote backend data warehouses. We have implemented and experimentally evaluated our query processing method and found that our strategy reduces query time between 50% and 60% for practical user cache sizes and network parameters.*

1 Introduction

Many enterprises are generating massive amounts of data on a day-to-day basis measuring various facets of their operations. In an On-Line Analytical Processing (OLAP) environment, users pose queries on this data whose answers are used to drive the decision making process. They are interested in identifying and analyzing the trends and anomalies hidden within the data. Typically, OLAP queries make heavy use of aggregation, and may take a long time to compute due to the large amounts of data which must be scanned.

It is natural for the data of an enterprise to be stored at the location where it is generated. Many enterprises operate in a highly distributed fashion, and hence the locations holding the actual data may be distributed at distant sites from one another. For example consider the international manufacturing and sales enterprise shown in Figure 1, with its operational sites distributed at different locations all over the world. Each site records data in which the members of the enterprise wish to identify trends and anomalies using OLAP techniques. One of the regional offices is exposed showing that it contains one database server, six user agents with local caches and a site broker on its local area network. Individuals in this enterprise may use OLAP techniques to gain information about the quantity of sales of an item during a particular time period, organized by geographical region, or perhaps about the rate and types of manufacturing defects produced over time.

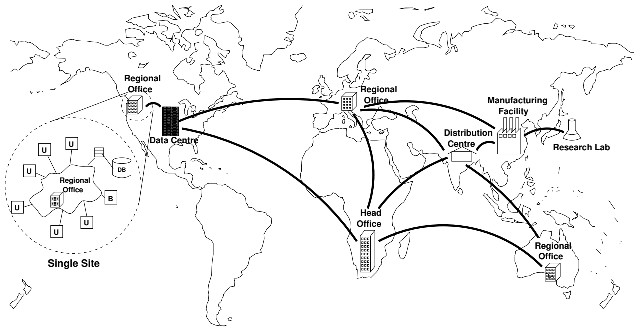


Figure 1. A distributed enterprise.

The standard approach to OLAP for such an enterprise would be to construct a single, centralized data warehouse at the data centre, by extracting and integrating all of the enterprise's data in that location. This data warehouse would then be the sole site which computes analytical queries for the entire enterprise. Although simple, this approach suffers from significant scalability problems in terms of the number of users, sites and amount of data that can be effectively managed.

1-4244-0910-1/07/$20.00 ©2007 IEEE.

Recently there has been growing interest in the design of grid based OLAP applications [1, 12, 13]. In [1], a grid application for performing data mining and OLAP tasks on heterogeneous health care data was described. The focus here was primarily on the application and data integration issues, rather than efficiency issues. In [13] the focus was on the challenging problem of constructing OLAP datacubes in a grid environment. Although query processing was briefly addressed the proposed approach is quite simplistic. It did not make use of cached results which 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. In [12] the authors sketched a model for the OLAP Enabled Grid. The idea was to take advantage of the hierarchal structure of a typical enterprise grid and use collaborative caching to reduce the need to collect data from distant sites.

Our work here focuses on the query fragmentation aspect, giving new details and algorithms which are experimentally evaluated. We introduce a new query processing scheme for OLAP in a grid that carefully blends sophisticated cache extraction techniques and data grid scheduling to efficiently satisfy queries in a distributed fashion. The heart of our approach is the query Fragment Aggregation and Recombination (FAR) strategy that partitions OLAP queries into subqueries which can be effectively answered by retrieving *and aggregating* fragments of cached data from nearby sources, or as a last resort, more remote backend data warehouses. Tier 1 of our query processing algorithm makes efficient use of many cooperating user caches on the same site as the query initiator. Those subqueries that can not be resolved locally are then passed on to Tier 2 processing which schedules them over remote servers on distant sites, based on data grid scheduling mechanisms.

In order to evaluate the FAR query processing method in the grid context we have implemented the principle components of our grid application, namely the cache index search of the Site Brokers, as well as the cache admission controller and subsystem responsible for fetching cached fragments on the users. We have benchmarked this implementation with a focus on evaluating the benefit of our query processing approach that aggregates and recombines bits of cached OLAP data in order to answer as much of a query locally as possible. Our experiments show that our FAR approach results in a significant reduction in query cost as opposed to directly sending queries to the backend. For caches of from 50 to 250MB in size we observe a reduction in query time of 50% and 60% and a corresponding reduction in non-local traffic.

2 Background

A typical data warehouse stores its information according to a star schema having a central fact table with d feature attributes (dimensions), and some number of measure attributes. For example, a simple fact table might consist of 3 feature attributes (A,B,C) and a single measure attribute (SALES). In addition to the fact table, there are often dimension tables which give further details about the dimensions. These details often define a hierarchy on the values of a dimension. For example, *time* forms a hierarchy on *week*, *month*, and *year*.

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). In such queries, the user requests aggregated measure values for sets of records which are grouped by their values for a particularly interesting subset of the feature dimensions. Our techniques in this paper apply to the commonly used non-holistic set of aggregation functions, where the correct answer can be formed by an aggregation of aggregates.

Aggregate queries in OLAP are categorized by the dimensions they choose to group by, and the aggregated table resulting from such queries are called *views*. For example, starting from the fact table (A,B,C) if we group results by A and B, aggregating each unique (A,B) pair over all values of C, we get a new view, namely (A,B). 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 possible views is $\prod_{i=1}^{d}(H_i)$.

Harinarayan et. al. introduced the *data cube lattice* in [7], expressing the relationship between views as a partial order (directed acyclic graph). 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. For data warehouses with dimension hierarchies the data cube takes the form of a general partial order, however without dimension hierarchies it is a hypercube. 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 a fact table having feature dimensions (A,B,C) is shown in Figure 2.

To represent queries and fragments, we follow the approach of [9] and associate with each v_i an interval representing the range of values selected by a query or contained by a fragment. Hence for a view $v = (v_1, v_2, \ldots v_d)$, a query on v is of the form $q = ((q_1, v_1), (q_2, v_2), \ldots (q_d, v_d))$, where q_i is a [min,max] interval representing the selection range on attribute v_i. 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. Figure 2 also shows view fragments having 3, 2, and 1 dimension and the relationship between those view fragments.

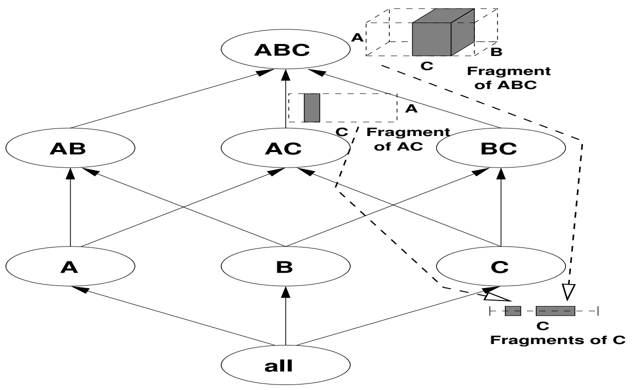


Figure 2. The data cube lattice and 3, 2, and 1 dimensional view fragments.

3 The OLAP Enabled Grid

Query processing in our application, the OLAP Enabled Grid [10, 12], proceeds 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's workstation or a database server. The details of one of the sites (a regional office) of the example enterprise from the introduction are exposed in Figure 1, revealing the various entities attached to a site.

Each entity has a role according to the functionality it offers for query processing in the OLAP Enabled Grid. We categorize entities into four different roles based on their participation.

1. *Database Server* - A machine which manages an operational database in the enterprise. The data maintained by different Database Servers is independent.
2. *Compute/Storage Server* - 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 performing OLAP operations on the data managed by the Database Servers. Each User Agent maintains a cache for storing results of previously answered queries.

In the OLAP Enabled Grid, a Compute/Storage Server on a site is used to implement the functionality of an *OLAP Server* on the data maintained by one or more of that site's Database Servers. In order to provide high scalability, we have multiple redundant OLAP Servers per Database Server (depending on the availability of Compute/Storage Servers on the site). We refer to the collection of redundant OLAP Servers for a particular Database Server's data as a *Grid OLAP Service* for that data. All of the Grid OLAP Services use a common data warehouse schema, and the data of the enterprise is partitioned horizontally (by dimension value) across the Grid OLAP Services. For a detailed overview of the system components and their corresponding layers of the Open Grid Services Architecture [6], see [10]. In the following section we describe how the Site Broker, User Agents, and OLAP Servers participate together in order to process queries in the OLAP Enabled Grid.

4 Query Processing Algorithms

This section describes our two-tiered query processing strategy and the algorithms for partitioning a query and answering subqueries from cached data. Tier 1 of the query processing uses the User Agents' caches on the local site in a cooperative manner to answer as much of the query as possible, due to the fact that transmission within a site is much quicker than between sites. Tier 2 sends subqueries for the missing fragments to other sites which may contain data for these fragments (we use the bounding box approach as in [12] to determine this), where they are answered using a grid scheduling approach. The entire process is outlined in Algorithm 1. Tier 1 is performed in Steps 1 to 9 of the algorithm, while Tier 2 is performed in Steps 10 to 18. Note that many of the steps in the algorithm are asynchronous due to the fact that they are performed on different entities.

4.1 Tier 1: Local Cache Extraction

The first tier in the query answering process consists of extracting cached query results relevant to the incoming query. In order to identify fragments on views higher up in the lattice which can be aggregated, the Site Broker implements an aggregate aware index on the collection of

Algorithm 1 Two-Tiered Query Answering Overview

```
1:  User Agent $U$ sends query $q$ to its local Site Broker $B$.
2:  $B$ performs the FAR cache search to find a set of cached
    fragments $F$ for $q$.
3:  $B$ formulates subqueries for the remaining parts of $q$.
4:  $B$ sends the fragmentation plan back to $U$.
5:  for all cached fragments $f$ in $F$ on a User Agent $U'$ do
6:      Send a request for $f$ to $U'$.
7:      $U'$ performs any necessary aggregation and returns $f$.
8:      $U$ checks if $f$ should replace other fragments in its cache.
9:  end for
10: for all sub-queries $q'$ in the fragmentation plan do
11:     for all Grid OLAP Services $G$ to which $q'$ should be sent do
12:         $U$ sends $q'$ to the broker $B'$ at $G$'s site.
13:         for all OLAP Servers $S$ for $G$ do
14:             $B'$ checks to see how quickly $S$ can answer $q'$.
15:         end for
16:         $B'$ sends $q'$ to the OLAP Server $S_{min}$ which can
            answer it the quickest.
17:         $S_{min}$ answers $q'$ and sends the result back to $U$.
18:     end for
19: end for
20: for all Fragments $f$ received from remote OLAP Servers do
21:     $U$ checks if $f$ should replace other fragments in its cache.
22: end for
23: $U$ notifies $B$ of any cache updates.
24: $U$ combines the retrieved fragments into the overall
    query result.
```

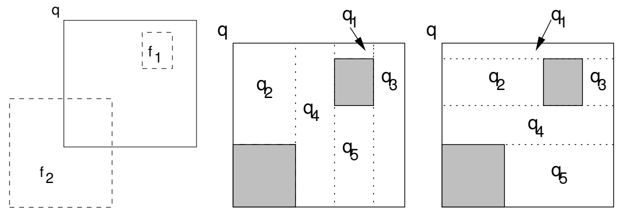
locally cached data. The index consists of the data cube lattice structure, with an R-Tree for each view indexing cached fragments of that view.

In order to identify a set of locally cached fragments to answer subparts of the query, our index lookup (Step 2 of Algorithm 1) consists of a variation of a breadth first search up the lattice. It begins at the view over which the original query was posed, and identifies fragments at this view which overlap in part with the query. The parts of the query which can be answered from these overlapping fragments are subtracted, and the search proceeds recursively up the lattice with the set of subqueries for which no overlapping cached fragments have yet been found. We do not formulate subqueries for fragments at higher levels in the lattice than the original query, since these fragments are less aggregated, containing more data and hence taking longer to transfer from the remote servers. Our cache search is similar to an aggregate aware caching proposal by Deshpande and Naughton [5], except theirs is based on a discrete partitioning of views and query results into equal sized chunks, where as ours uses rectangle geometry to identify fragments and partition a query.

When the search terminates at the view containing all of the dimensions in the fact table, it has identified a set of cached fragments which are stored on User Agents on the local site, and a set of subqueries which must be answered using the backend OLAP Servers in Tier 2. We refer to this as the *fragmentation plan*, which is returned to the User Agent from the Site Broker following the cache index lookup.

In order to implement the FAR cache search as described above, we need an algorithm which computes a set of subqueries Q given a query q and a set of intersecting view fragments F. Geometrically, this is the difference between q and the union of all fragments in F. An example is shown in Figure 3. Figure 3(a) shows a query q on a two dimensional view and two intersecting fragments f_1 and f_2, and subfigures (b) and (c) show possible solutions. Algorithm 2 gives our iterative algorithm over the dimensions of the query which subtracts a single fragment from a query and gives the set of subqueries. For each dimension, there are 3 cases (Steps 4, 16, and 21) which determine the range on that dimension of the query which will be used in further iterations (q^I), and the queries which are added to the cumulative solution (q^L, q^R). The solution (b) of Figure 3 is found if the horizontal dimension is visited first in the loop of Step 3 of the Algorithm 2, and the solution (c) of the figure if the vertical dimension is visited first. By convention we use half-open intervals to avoid overlap of the endpoints in the algorithm. An algorithm which computes the desired set Q given the set F is given in Algorithm 3. This algorithm also determines which fragments in F are actually necessary, since there may be fragments in F which are completely contained in the union of other fragments in F. The overall FAR cache search algorithm is based on breadth-first search and is given as Algorithm 4. An extensive complexity analysis is withheld here due to space limitations, but we believe that the size of the fragmentation plan returned by a Site Broker on Step 4 of Algorithm 1 is proportional to the product of the number of dimensions and the number of cached fragments which intersect with the query.

In the following section we describe Tier 2 query processing, which occurs in Steps 10 to 18 of Algorithm 1. Of key importance is our scheduling mechanism over the redundant OLAP Servers of a Grid OLAP Service, which occurs in Steps 13 to 17 of the algorithm. Following this, an empirical evaluation aimed at validating the performance of our local cache extraction method (FAR) under practical conditions is found in Section 5.



(a) Input problem (b) Solution 1 (c) Solution 2

Figure 3. Subquery formulation.

Algorithm 2 Compute Query Fragment Diff

Input: Query $q = ((q_1, v_1), (q_2, v_2), \ldots (q_d, v_d))$, Fragment $f = ((f_1, v_1), (f_2, v_2), \ldots (f_d, v_d))$ such that $f \cap q \neq \emptyset$

Output: Set Q of non-intersecting subqueries of q such that $q = \bigcup_{q' \in Q} q' \cup (f \cap q)$ and no query in Q intersects with f.

```
1:  $Q \leftarrow \emptyset$
2:  $q^I \leftarrow q$
3:  for all $i$ such that $v_i \neq$ all do
4:    if $q_i$ contains $f_i$ then
5:      if $min(q_i) < min(f_i)$ then
6:        $q^L \leftarrow q^I$
7:        $i$th range in $q^L \leftarrow (min(q_i), min(f_i))$
8:        $Q \leftarrow Q \cup \{q^L\}$
9:      end if
10:     if $max(q_i) > max(f_i)$ then
11:       $q^R \leftarrow q^I$
12:       $i$th range in $q^R \leftarrow (max(f_i), max(q_i))$
13:       $Q \leftarrow Q \cup \{q^R\}$
14:     end if
15:     Replace $i$th range in $q^I$ with $f_i$
16:   else if $max(q_i) > max(f_i)$ then
17:     $q^R \leftarrow q^I$
18:     min of $i$th range in $q^R \leftarrow max(f_i)$
19:     $Q \leftarrow Q \cup \{q^R\}$
20:     $i$th range in $q^I \leftarrow (min(q_i), max(f_i))$
21:   else if $min(q_i) < min(f_i)$ then
22:     $q^L \leftarrow q^I$
23:     max of $i$th range in $q^L \leftarrow min(f_i)$
24:     $Q \leftarrow Q \cup \{q^L\}$
25:     $i$th range in $q^I \leftarrow (min(f_i), max(q_i))$
26:   end if
27: end for
```

4.2 Tier 2: Query Scheduling

Once the User Agent has received a fragmentation plan for its query from the Site Broker, it begins issuing requests for subqueries to the Grid OLAP Services on the various sites. The Site Brokers at each site also implement the Grid OLAP Services, by choosing which of the redundant OLAP Servers for that Grid OLAP Service to send each particular incoming query to. Our approach is similar to previous grid

Algorithm 3 Compute Subqueries

Input: Query $q = ((q_1, v_1), (q_2, v_2), \ldots (q_d, v_d))$ Set of intersecting fragments F

Output: Set $F' \subseteq F$ such that $q \cap \bigcup_{f' \in F'} f' = q \cap \bigcup_{f \in F} f$. Set Q of non-intersecting subqueries of q such that $q = \bigcup_{q' \in Q} q' \cup \bigcup_{f' \in F'} (f' \cap q)$ and no query in Q intersects with a fragment in F'.

```
1:  $Q \leftarrow \{q\}$
2:  $F' \leftarrow \emptyset$
3:  for all $f \in F$ do
4:    $Q' \leftarrow \emptyset$
5:    for all $q' \in Q$ do
6:      if $f$ intersects $q'$ then
7:        $Q' \leftarrow Q' \cup$ Compute Query Fragment Diff$(q', f)$
8:        $F' \leftarrow F' \cup \{f \cap q'\}$
9:      end if
10:   end for
11:   $Q \leftarrow Q'$
12: end for
```

Algorithm 4 FAR Cache Search

Input: Query $q = ((I_1, v_1), (I_2, v_2), \ldots, (I_d, v_d))$

Output: Set of fragments F of q which are cached, and a set of sub-queries Q of q which need to be answered by the OLAP servers.

```
1:  $F \leftarrow$ all cached fragments intersecting $q$ as found by
    searching the R-Tree of view $(v_1, v_2, \ldots v_d)$
2:  $F, Q \leftarrow$ Compute Subqueries$(q, F)$
3:  $Queue \leftarrow$ Parents$((v_1, v_2, \ldots v_d))$
4:  while $Queue \neq \emptyset$ and $Q \neq \emptyset$ do
5:    $v \leftarrow Queue.dequeue()$
6:    for $q' \in Q$ do
7:      Re-write $q'$ over view $v$
8:      $F' \leftarrow$ all cached fragments intersecting $q'$ as
        found by searching the R-Tree of view $v$
9:      $F', Q' \leftarrow$ Compute Subqueries$(q', F')$
10:     if $Q' = \emptyset$ then
11:       $F \leftarrow F \cup F'$
12:       $Q \leftarrow Q \backslash \{q'\}$
13:     end if
14:     for $v'$ parent of $v$ do
15:       if $v' \notin Queue$ then
16:         $Queue.enqueue(v')$
17:       end if
18:     end for
19:   end for
20: end while
```

schedulers [14, 15], except using a cost modeling specific to range aggregate OLAP queries. The query is scheduled on the OLAP Server which estimates it can answer it and return the results to the user the quickest. This depends on both the CPU and network interface load on the various servers, their processing speed, disk bandwidth and load, as well as materialized indexes and views on the data as explained below.

We express the time to answer a query q on a particular OLAP Server S and return the results to the user as

$$t(q, S) = t_c(q, S) + t_n(q, S)$$

where t_c is the computation time and t_n is the network transfer time back to the user who requested it. When a Site Broker receives a query q for a particular Grid OLAP Service, it polls each OLAP Server S for that Grid OLAP Service, asking S to compute $t_c(q, S)$ and $t_n(q, S)$.

In the OLAP-Enabled Grid, we use R-Trees to index the data stored on each block of external memory, which is ordered on disk according to a multidimensional space-filling curve as in [2–4]. In those studies, we have observed that the time to answer a query is proportional to the amount of data which must be read from disk, which in turn depends on the selection ranges of the query and the materialized view which will be used to answer it. $t_c(q, S)$ can then be expressed as a function of the amount of data $d(q, S)$ to be read from disk and the available bandwidth and load on S's disk system, as in

$$t_c(q, S) = \frac{d(q, S)}{disk\_bandwidth(S)}$$

$d(q, S)$ itself depends on the materialized view of S which q will be answered on. In the OLAP-Enabled Grid, S maintains a reference $a(v)$ to the smallest materialized ancestor of each view v in the lattice, and upon receiving a query q, it translates q over $a(view(q))$. It can then use statistics about the selectivity of the dimensions in $a(view(q))$ to approximate the amount of data to be read to answer q on $a(view(q))$, and estimate the time this would take by checking the current load and available bandwidth of its disk system.

The estimation of $t_n(q, S)$ is relatively straightforward. If S is on a different site from the user, it depends on both the available bandwidth of the link from S to the gateway of the site and on the available bandwidth across the link to the user's site. Information of this nature can be obtained for example by using the Network Weather Service [17]. $t_n(q, S)$ is the estimated size of the query result divided by the minimum of these two values. If S is on the same site as the user, then $t_n(q, S)$ is the estimated size of the query result divided by the available bandwidth from S to the user. The available bandwidth depends on various factors including the load on S's network interface, and on the load and speed of the links.

5 Experimental Analysis

Our prototype implementation is a parallel program written in the Python scripting language, with communication between the entities (Site Broker, User Agents and OLAP Servers) being achieved with Message Passing Interface (MPI) bindings for Python. We use simulated data and data warehouse servers, which respond based on parameters for an estimated processing time and network bandwidth (both local and remote).

A diagram illustrating the software architecture of our implementation is shown in Figure 4. Each class is represented as a white rectangle, with classes which are programs being represented by a gray rectangle. Entities are connected to MPI in the figure, and files are shown as white rectangles with the corner folded in. All classes related to the manipulation of the data cube and its properties are grouped into the *Cube Manager*. The Cube Manager acts on a single data cube lattice and schema which is constructed by the XSDParser. The Cube Manager provides information about View instances and their organization into a Lattice instance. The relevant information about views for our implementation are their size, dimensions, ranges of those dimensions, and the mapping between ranges for dimensions organized into a hierarchy. Access to these properties and functions allows for manipulation of Query and Fragment instances over the lattice as required to perform the cache search, indexing, and aggregation/grouping of OLAP data. The number of records in each view is estimated using the technique of [16] which is based on dimension cardinalities and the number of records in the fact table.

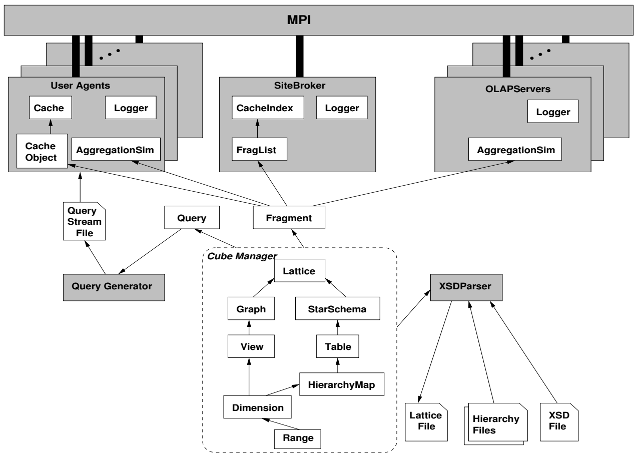


Figure 4. Software design.

The entities communicate asynchronously, the behaviour of each being determined by an event loop. For example, the event loop of the Site Broker waits until it has a message. If the message is a query, it performs the FAR search, produces the fragmentation plan and sends it back. The User Agents and OLAP Servers have AggregationSim objects, which simulate the aggregation of data from disk, producing a stream of data. Additionally, the User Agents use a Cache instance representing a collection of CacheObjects which are dynamically added to or removed from the Cache during the run, and can be requested by other users. Each of the entities has a logger which records time stamped events during the simulation, and our experimental data is gathered by analyzing these logs.

5.1 Experiments

For all our tests, we use a lattice with 5 feature dimensions and a single measure dimension *sales*. One of the dimensions (*time*) has a 5-level non-linear hierarchy, while another 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 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 [8, 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 on their values.

Each User Agent is configured with a specified cache size, a disk bandwidth, a query stream, and optionally a list of fragments with which to initially fill the 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.

To measure the benefit of our cache extraction strategy, we use the Detailed Cost Savings Ratio (DCSR) [9]

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

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.

The first set of tests aims to determine the cache search strategies' ability to make effective use of increasing cache space. We perform 5 independent runs using 5 different hot region distributions on a site with 10 User Agents. For each run, each User Agent initiates 10 queries over 2 hours, and has a set of fragments used to initially warm its cache which are generated from the same hot region distribution. The size of each User Agent's cache is varied from 50 MB to 500 MB, so that the Site Broker indexes 500 MB of fragments at the minimum and 5000 MB of fragments at the maximum, or between 1.4% and 14% of the size of the entire data cube lattice. We simulate a total of 5 Grid OLAP Services (1 local and 4 remote), each having a small set of randomly chosen materialized views and a disk bandwidth of 80 MB/s. The DCSR of FAR as a function of cache space per user is shown in Figure 5. Note that our FAR cache extraction 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 begins to wane due to the increased cost of the cache search and number of separate requests which must be made for each query. The nature of this degradation in performance is examined in the experiments that follow.

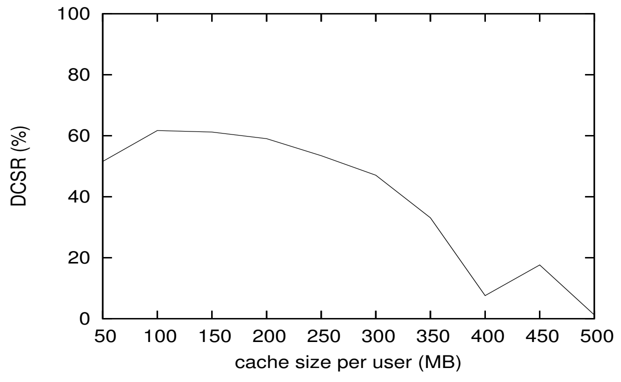


Figure 5. Cost savings of caching as cache size per user is increased.

In order to better explore the apparent diminishing returns FAR exhibits for large caches, we have further broken down average query time into the three components queue time, search time and backend time. Queue time is the time that the query waits in the input queue of the Site Broker for its fragmentation plan to be produced. Search time is the time it takes for the Site Broker to perform the FAR cache extraction. Backend time is the length of the time period from when the User Agent receives the fragmentation plan from the Site Broker to when it has received the last subquery result. The average query time for the FAR approach broken down into these three components as cache size is increased is shown in Figure 6. For the 50 MB caches, the queueing time of queries is insignificant, as the Site Broker is able to keep up with the number of requests it receives. There is a large reduction in time at the backend from the 50 MB to the 100 MB caches, coupled with only a small increase in search time and a minor increase in queueing time (about 1/2 second on average), causing the overall query time to be lower. As the caches increase in size from 100 MB however, the decrease in backend time does not make up for the increase in cache search time and the resulting increase in queue time. There is a substantial increase in query time from 350 to 400 MB, where the cache search time makes a large jump of roughly 2.5 seconds causing the Site Broker to be taxed and consequently fail to service its queue in a timely manner. For the experiments using the uniform query distribution (not shown due to space limitations), the cache search time does not increase as drastically with cache size, enabling a much higher cost savings with larger caches. It is important to note that the increase in cache search time and the resulting increase in queue time for the hot region queries is largely an artifact of our implementation. Python, although extremely fast to implement in, is too slow to effectively implement the FAR cache search process. We estimate that the cache search time and corresponding queue time would be reduced by a factor of 50 to 100 in a well optimized C based implementation.

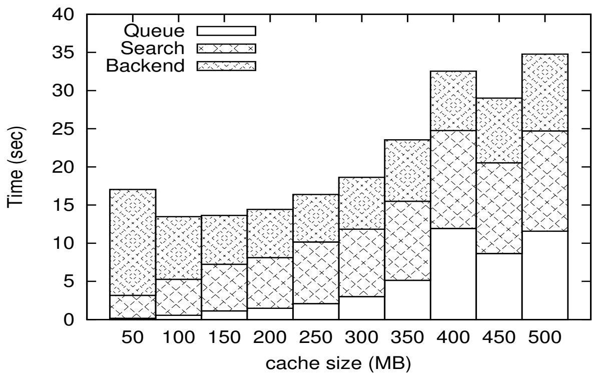


Figure 6. Average query time vs cache size, by queue time, search time and backend time. Hot region queries.

6 Conclusion

We have presented a new query processing scheme for the OLAP Enabled Grid that blends sophisticated cache extraction techniques and data grid scheduling to efficiently satisfy queries in a distributed fashion. The heart of our approach is our Fragment Aggregation and Recombination (FAR) method that partitions OLAP queries into subqueries which can be effectively answered by retrieving and aggregating fragments of cached data from nearby sources, or as a last resort, the backend data warehouses. We have implemented and experimentally evaluated our query processing method and found that our strategy reduces query time between 50% and 60% for practical sizes of users caches and realistic network parameters. Given that our prototype implementation performs well, the natural next step is to explore an optimized and scalable implementation using an efficient compiled language like C and a standard grid toolkit such as Globus.

References

[1] 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.

[2] 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.

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

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

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

[6] 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.

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

[8] 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.

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

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

[11] M. Lawrence. Multiobjective genetic algorithms for materialized view selection in olap data warehouses. In *Proc. GECCO'06*, 2006.

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

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

[14] 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.

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

[16] A. Shukla, P. Deshpande, J. F. Naughton, and K. Ramasamy. Storage estimation for multidimensional aggregates in the presence of hierarchies. In *The VLDB Journal*, pages 522–531, 1996.

[17] R. Wolski, N. T. Spring, and J. Hayes. The network weather service: a distributed resource performance forecasting service for metacomputing. *Future Gener. Comput. Syst.*, 15(5-6):757–768, 1999.