Proxy-Server Architectures for OLAP

Panos Kalnis
Department of Computer Science
Hong Kong University of Science and Technology
Clear Water Bay, Hong Kong
http://www.cs.ust.hk/~kalnis

Dimitris Papadias
Department of Computer Science
Hong Kong University of Science and Technology
Clear Water Bay, Hong Kong
http://www.cs.ust.hk/~dimitris

ABSTRACT

Data warehouses have been successfully employed for assisting decision making by offering a global view of the enterprise data and providing mechanisms for On-Line Analytical processing. Traditionally, data warehouses are utilized within the limits of an enterprise or organization. The growth of Internet and WWW however, has created new opportunities for data sharing among ad-hoc, geographically spanned and possibly mobile users. Since it is impractical for each enterprise to set up a worldwide infrastructure, currently such applications are handled by the central warehouse. This often yields poor performance, due to overloading of the central server and low transfer rate of the network.

In this paper we propose an architecture for OLAP cache servers (OCS). An OCS is the equivalent of a proxy-server for web documents, but it is designed to accommodate data from warehouses and support OLAP operations. We allow numerous OCSs to be connected via an arbitrary network, and present a centralized, a semi-centralized and an autonomous control policy. We experimentally evaluate these policies and compare the performance gain against the existing systems where caching is performed only at the client side. Our architecture offers increased autonomy at remote clients, substantial network traffic savings, better scalability, lower response time and is complementary both to existing OLAP cache systems and distributed OLAP approaches.

1. INTRODUCTION

A data warehouse is a collection of historical summarized information, which aims at improving decision-making. The process of accessing and manipulating these data is referred as On-Line Analytical Processing (OLAP) [CCS93]. OLAP operations typically deal with aggregations, which involve huge amounts of data, as opposed to accessing only a few tuples at a time in an On-Line Transaction Processing (OLTP) environment. Moreover, OLAP operations may require data, which are distributed in numerous geographically spanned databases. Therefore, although it is possible to execute such queries against operational databases, in practice a separate database is used to implement the data warehouse.

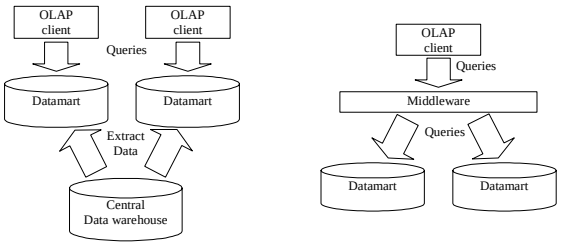
ACM SIGMOD 2001, May 21-24, Santa Barbara, California, USA.

Usually, each department in an enterprise is interested only in a fragment of the total data. If a central data warehouse has been built, the data can be partitioned and extracted in departmental data warehouses, called data marts (Figure 1a). This approach ensures consistency across the enterprise. Nevertheless, it may be too hard to create a global model for the central warehouse. In this case, each department builds its own data mart (decentralized approach), and a middleware layer is employed to provide a global view of the data (Figure 1b).

The implementation of a data warehouse, naturally involves many distributed database issues. The process of accessing the operational databases, extracting, cleaning and integrating the data in the warehouse, is a distributed database application. In [GLWZ99] the authors discuss the particular issues of the data warehouse environment for distributed and parallel computation. Some of the challenges include the maintenance of the warehouse data consistency given a set of autonomous sources, the process of resuming failed warehouse loads without undoing the incomplete load, and the parallelization of view maintenance tasks.

Furthermore, accessing the data warehouse is also a direct application of distributed database technology. For the centralized case [OV99] standard replication methods can be used, although some distributed protocols, like two-phase commit, may be inapplicable. For the decentralized case Hull et. al. [HZ96] employ a global schema over the data. Albrecht et. al. [AGL98, AL98] have also proposed an architecture for distributed OLAP queries, over a decentralized schema. Their middleware component follows an economical approach (similar to the *mariposa* system [SAL+96]) to assign fragments of an OLAP query to the most beneficial data mart, while each data mart dynamically trades for data fragments that will potentially maximize its market share.

All of these approaches assume a static organizational structure with well-defined needs for data access and OLAP operations. However, the popularity of Internet and WWW nowadays, has created new opportunities to access information, which are not restricted to the narrow limits of a single enterprise. Imagine for instance a stock-market data warehouse. Professional stockbrokers



(a) Centralized **(b)** Decentralized

Figure 1: Two approaches for building a data warehouse

naturally have access to it by one of the methods described above. However, the globalization of economy enables many individuals around the world to trade stock at the same market and they also need access to the data. It is clearly impractical to predict some access patterns and set up the infrastructure dedicated to this particular application. The need to perform OLAP operations over large amounts of data by geographically spanned ad-hoc users is not limited to the financial sector. Meteorological and environmental databases or other scientific information sources also have similar requirements. Moreover, decision makers from large multinational enterprises need to manipulate global company information regardless of their location and mobility.

In this paper we propose an architecture for distributed *OLAP Cache-Servers* (OCS). OCSs are similar to proxy-servers, which are widely used in the WWW environment. An OCS, however, does not store static pages, but it dynamically caches results from OLAP queries and also has computational capabilities. In our architecture, OCSs are geographically spanned and connected through an arbitrary network. Clients for OLAP data, do not access the data warehouse directly, but address their query to an OCS, which in turn, either answers the query from its local cache, or redirects it to one of its neighbors. Our architecture provides increased autonomy at remote clients, substantial network traffic savings and better scalability. From the user's point of view, the improvement is translated to lower response time. Our contributions include:

- The proposal of an architecture for distributed OLAP cache servers.
- The experimental evaluation of a centralized, a semi-centralized and an autonomous control policy.
- The development of an algorithm for efficient on-line incremental updating of the cached fragments.

The rest of the paper is organized as follows: In section 2 we review the related work on dynamic caching of OLAP queries. Section 3 presents our architecture, while in section 4 we discuss the cost model and the alternative caching policies. In section 5 we present our update method. The simulation studies are described in section 6 and section 7 includes our summary remarks and concludes the paper.

2. BACKGROUND AND RELATED WORK

For the rest of the paper we will assume that the multi-dimensional data are mapped on a relational database using a star schema [Kim96]. Let D_1, D_2, …, D_n be the dimensions (i.e. business perspectives) of the database, such as *Product*, *Customer* and *Time*. Let M be the measure of interest; *Sales* for example. Each D_i table stores details about the dimension, while M is stored in a fact table F. A tuple in F contains the measure plus pointers to the dimension tables.

There are $O(2^n)$ possible group-by queries for a data warehouse with n dimensional attributes. A detailed group-by query can be used to answer more abstract aggregations. Harinarayan et. al. [HRU96] introduced the search lattice L, which is a directed graph whose nodes represent group-by queries and edges express the interdependencies among group-bys. There is a path from node u_i to node u_j, if u_i can be used to answer u_j (see Figure 2).

A common technique to accelerate OLAP is to precalculate some aggregations and store them as materialized views, provided that some statistical properties of the expected workload are known in advance. The aim is to select a set of views for materialization, such that the query cost is minimized, while meeting the space and/or maintenance cost constraints which are provided by the administrator. [HRU96, Gupt97, GM99] describe greedy algorithms for the view selection problem. In [GHRU97] an extension of these algorithms is proposed, to select both views and indices on them. [BPT97] employ a method which identifies the relevant views of a lattice, for a given workload. [SDN98] use a simple and fast algorithm for selecting views in lattices with special properties.

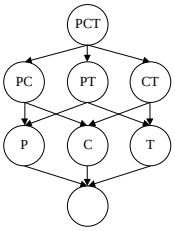


Figure 2: A data-cube lattice. The dimensions are *Product*, *Customer* and *Time*

All these methods follow a *static* approach where the views are selected once when the warehouse is set up. Periodically, the administrator monitors the workload of the system and recalibrates it by selecting a new set of views, if the query pattern has changed significantly. The recalibration process is expensive and pays off only if it is performed after a long period of time. An alternative solution is the dynamic caching of the OLAP queries' results. Due to the interdependencies among OLAP queries, keeping semantic information about the cache contents leads to better utilization of the cache than page caching. Some early work on semantic caching for general SQL queries appears in [DFJ+96, KB96]. For the special case of OLAP, [SSV96] developed a cache manager called *watchman*. Their system stores in the cache the results of the query, together with the query string. Subsequent queries can be answered by the cached data if there is an exact match on their query strings. The authors present admission and replacement algorithms that consider the cost to re-evaluate a result and also the size of the result and are, therefore, suitable for OLAP data.

Another alternative is proposed by [DRSN98]. They decompose regularly the multidimensional space into *chunks* [ZDN97], and when a new result arrives, they cache the corresponding chunks. When a new query is asked, the system computes the set of chunks that are required to answer it and splits it into two subsets: the set of chunks that are present in the cache and these that are not present. To answer the query, the system will retrieve the first set of chunks from the cache and will send a query for the missing chunks to the backend. Only chunks at the same aggregation level as the query can be used, i.e. no aggregation is done on the cached results. The admission and replacement algorithms are similar to *watchman*.

Dynamat [KR99] is another OLAP cache manager which considers both the size and maintenance cost constraints. *Dynamat* stores "fragments", which are aggregate query results in a finer granularity than views (they may include selection on single values on some dimensions). During the on-line phase, when a query arrives, the system checks whether there is a related materialized fragment in the cache (not necessarily at the same

level of aggregation as the query). In this case, the fragment is used to answer the query, otherwise the base tables are accessed and the result is stored as a new fragment in the cache. If the cache gets full (space constraint) a fragment is chosen for replacement or the new results are not materialized depending on a benefit metric. During the update window, *dynamat* must discard a number of fragments so that the remaining ones can be updated within the available time.

A subject relevant to our architecture is *active caching* [CZB98] in web proxies. This approach allows the proxy server to construct pages dynamically, by caching the data together with an applet to manipulate them. Most of the work in this field deals with the integration of the applet mechanism to the proxies. To the best of our knowledge, no work has been published on active caching for OLAP data.

The systems presented above were focused on a three-tier architecture, where the caching takes place in the middle tier. Our architecture is more general, by allowing multiple levels of caching and cooperation among the cache servers and also by considering the network factor. It the next section we describe our approach in detail.

3. SYSTEM ARCHITECTURE

The architecture of our system is similar to the proxy-server architecture, which is widely used to cache web pages. There are three layers: The data warehouse layer, the *OLAP Cache Server* (OCS) layer and the client layer. In the general case, there are many distinct warehouses, corresponding to different enterprises. The data warehouses are connected to a number of OCSs which are interconnected through an arbitrary network. The clients do not access the warehouses directly; instead, they send the query to an OCS. Each OCS contains a cache where previous results are stored, and has computational capabilities, so it can use the cached results for further aggregations. The intuition behind our architecture is that most of the time, some OCS in the network will be able to satisfy the requests from the clients, saving network and computation cost incurred by directing the request to the data warehouse.

Based on the degree of control distribution, we describe three variations of our architecture:

- Centralized: A central site has full knowledge about the status of all OCSs. This site is responsible for constructing the execution plan for every query and deciding if and where will the results be cached.
- Semi-centralized: Same as before, but the central server only constructs the execution plans for the queries, while the OCSs control autonomously their cache.
- Autonomous: There is no central control site. All decisions are taken locally at the OCSs.

Intuitively, the centralized control leads to lower total execution cost and better utilization of the OCSs, due to the global knowledge of the system's status. This approach is used in many distributed DBMSs and in our case it is ideal for intranet installations, where the entire infrastructure belongs to the same organization, but traditional fragmentation methods cannot be used for some reasons; for example when the query patterns are dynamic and unpredictable. The autonomous control, on the other hand, is suitable for large scale unstructured environments (e.g. Internet), by providing better scalability and availability of the system. The semi-centralized system, finally, is a trade-off between the other approaches.

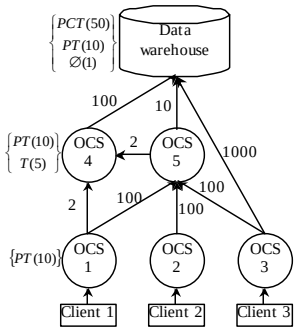


Figure 3: A network of OCSs. There is a directed edge from a site *i* to a site *j* if *i* can direct a query to *j*. The numbers on the edges are the inverse of the transfer rates of the network connections

As an example, consider the configuration of Figure 3. There are three clients, a network of five OCSs and a data warehouse. The cache contents are presented next to the OCSs, and the numbers in parenthesis represent the size of the views. The edges correspond to network connections; there is a directed edge from a site *i* to a site *j*, if *i* can direct a query to *j*. The numbers next to the edges denote the network cost. For simplicity, we assume that each client is connected to only one OCS and the network cost between a client and its corresponding OCS is zero. This convention allows us to assume that all the control algorithms and cache policies are built inside the OCSs. In a real system, however, a client can implement the same algorithms as the OCSs, connect to several OCSs and even include a local cache.

Assume that client 1 asks the following SQL query:

```
SELECT P_id, T_id, SUM(Sales)
FROM data
GROUP BY P_id, T_id
```

Each query is identified by its grouping attributes, so we can refer to this SQL statement as *q*="*PT*". The query is directed to $OCS^{[1]}$. Since *PT* is in the local cache, all three control approaches decide that the cached result should be used. The total cost is 10 and no further action is required. Now, let the same client ask a query *q*="∅", i.e. the sum of all sales. The query is directed again to $OCS^{[1]}$ but this time the exact answer is not in the cache. In the centralized approach, the query is redirected to the central control site which checks its global information and decides that the query should be answered by $OCS^{[4]}$ with total cost 7 including the network cost (i.e. it reads *T* and transmits *sum(T)*). Therefore, it redirects the query to $OCS^{[4]}$ together with information about the network path that the result must follow back to the client (i.e. from $OCS^{[4]}$ to $OCS^{[1]}$ and then to the client). It also includes directions about which OCSs in the return path should cache the new result, and which results they must evict if the cache is full.

In the semi-centralized approach, the same scenario is repeated, but the central site does not have any control on the contents of the OCSs caches. When the result is transferred from the OCS that answers the query (i.e. $OCS^{[4]}$) to the client, all the OCSs in the path (i.e. $OCS^{[4]}$, $OCS^{[1]}$) decide individually on their admission and replacement policies.

In the autonomous case, $OCS^{[1]}$ checks the query and estimates

that it can calculate ∅ from *PT* with cost 10, but it also asks its first degree neighbors. $OCS^{[5]}$ cannot calculate the answer, but $OCS^{[4]}$ can calculate it from *T* with cost 5 and transfer it back with cost 2, therefore the total cost is 7. So the query is redirected to $OCS^{[4]}$. However, since $OCS^{[4]}$ does not have the exact answer, it asks the data warehouse, which has already materialized ∅, but due to a slower network, the total cost is 1+100 = 101. So $OCS^{[4]}$ calculates locally the result and sends it to $OCS^{[1]}$. At the same time $OCS^{[4]}$ decides whether it should cache the new result at its local cache, based on a *goodness* metric. $OCS^{[1]}$ then returns the result to the client and decides whether to store the result in its own cache.

When the data in the warehouse are updated, the cached results in the OCSs are not invalidated. Instead, the warehouse calculates the necessary deltas and when a result is required, the corresponding OCS decides whether to ask for the updated view, or to request the corresponding delta and incrementally update the cached data. Actually, the deltas themselves are objects just like the query results, and are cached in the OCSs using the same mechanism.

An OLAP cache server, is different from a common proxy server for web pages in three basic ways:

- An OCS has computational capabilities and can derive dynamically the results following different execution plans, as opposed to accessing static pages, which are identical to the ones that were requested.
- The cache admission and replacement policies are optimized for OLAP operations.
- An OCS can update its contents incrementally, instead of just invalidating the outdated cached contents.

The OCS architecture is complementary to the distributed database approaches since the later deal with the efficient calculation of queries in a network of servers, while we deal with the efficient delivery of OLAP results to the geographically spanned users. Our approach is also complementary to the previous caching techniques for OLAP, since they deal with a single cache, while we focus on the cooperation of many cache servers over a network. In the sequel, we will describe in details the components of our architecture.

3.1 The OLAP Cache Server

The OLAP Cache Server (OCS) is the basic entity of the architecture. It consists of a view pool where the cached data is stored, a query optimizer which constructs the execution plan, and an admission control unit which implements the admission and replacement policies of the cache.

An important design aspect is the granularity at which the data is cached. Given a fact table and its corresponding dimensions, the data cube operator [GBLP96] computes the whole data cube, i.e. the set of all possible aggregations. The data cube can be viewed conceptually as a relation *DC*, whose attributes are the dimensions and the measures. The tuples that contain aggregated data on dimension D_i, are distinguished by the special value *ALL* in the attribute D_i. Any horizontal fragmentation of *DC* is a valid candidate for caching. An arbitrary fragmentation however, complicates both the cache admission control and the creation of optimized execution plans since a huge number of possible combinations of fragments need to be considered.

In our prototype we assumed for simplicity that the queries involve whole views, so we chose to fragment the data cube at the granularity of views. The same approach is used in [HRU96, GHRU97, SDN98]. However, the OCS architecture is independent from the granularity of fragmentation, therefore any of the other methods proposed in the literature [KR99, DRSN98] can be employed, if the expected workload is different. Actually, since each OCS is an independent entity, it can use its own method of fragmentation. For instance, an OCS that is close to the data warehouse can cache whole views, since it is expected to serve in parallel many queries for many non-related users. Therefore, there is a high probability that large portions of the views are utilized. On the other hand, OCSs which are closer to the clients, are expected intuitively to access some hot spots of data, so it is better to employ a finer granularity of fragmentation.

The cached data are stored on a disk. The physical organization can follow either the ROLAP or the MOLAP approach. In our prototype, we follow the former approach, i.e. we consider the cached data to be stored as tables in a relational DBMS. Recall that we assumed queries that involve whole views, so we require sequential scans of the stored views and we don't need any indexing. However, depending on the workload, such an implementation can be inefficient. For such cases, we can use a MOLAP approach, like multidimensional arrays stored in chunk files [DRSN98] or Cubetrees [RKR97]. Again, each OCS may use its own physical organization, as long as it conforms to a common interface.

3.2 Query optimizer

The query optimizer constructs an optimized execution plan for each query, considering both the calculation and the network cost factors. We propose three policies for the query optimizer: autonomous, centralized and semi-centralized. They can be described as follows:

3.2.1 Autonomous policy for the query optimizer

This policy assumes that all the decisions about query execution are taken locally in each OCS and there is no central coordinating site. Consider a query *q* that reaches $OCS^{[i]}$ either from a client or a neighbor $OCS^{[j]}$. $OCS^{[i]}$ checks whether it can answer *q* locally. There are three possibilities:

(i) $OCS^{[i]}$ has the exact answer in its cache. In this case it reads the answer and sends it to the site that asked the query. Let *M* be the set of cached views in $OCS^{[i]}$, $v_q \in M$ be the view which is the exact answer of *q* and $size(v_q)$ be the size of *v* in tuples. By assuming a linear cost model [HRU96] the computational cost of answering *q* in the presence of *M* equals to the cost of reading the answer from the cache, i.e.

$$c\_cost(q, M) = size(v_q)$$

(ii) $OCS^{[i]}$ cannot answer *q*. This means that the cached set *M* does not contain either v_q or any ancestor of v_q. The query must be redirected to another site $OCS^{[j]}$ which can be selected by several heuristics. In our implementation we guide the selection of $OCS^{[j]}$ using the following method (Figure 4): $OCS^{[i]}$ sends a request for *q* to all of its first degree neighbors. The neighbors reply whether they can answer *q* and if they can, they also return the estimated cost. Let $cost(q, M_j)$ be the cost to answer *q* at $OCS^{[j]}$. Then:

$$total\_cost(q, OCS^{[j]}) =$$
$$= c\_cost(q, M_j) + network\_cost(OCS^{[j]} \rightarrow OCS^{[i]}, q) =$$
$$= c\_cost(q, M_j) + size(q) / TR(OCS^{[j]} \rightarrow OCS^{[i]})$$

```
1   path := ∅
2   PROCEDURE select_neighbor(OCS[i], q) {
3       path := path ∪ OCS[i]
4       min_cost := ∞
5       FOR every OCS[j] ∈ Neighbors(OCS[i]){
6           c_cost := Ask OCS[j] what is the computation cost
7           total_cost := c_cost + cost to transfer the result
8           IF total_cost < min_cost THEN {
9               best_neighbor := OCS[j]
10              min_cost := total_cost(OCS[j], q)
11          } // END IF
12      } // END FOR
13      IF min_cost < ∞ THEN {
14          path := path ∪ best_neighbor
15          return path
16      } ELSE {
17          choose random neighbor OCS[j]
18          select_neighbor(OCS[j], q)
19      } // END IF
20  } // END select_neighbor
```

Figure 4: Algorithm for selecting the neighbor to redirect the query

where $TR(OCS^{[j]} \rightarrow OCS^{[i]})$ is the transfer rate of the network for the connection between $OCS^{[j]}$ and $OCS^{[i]}$. The $OCS^{[j]}$ with the minimum total cost is selected and the query is redirected to it. If none of the neighbors can answer *q*, $OCS^{[i]}$ selects one randomly and redirects the query to it. Note that this algorithm is a heuristic and cannot guarantee optimal execution plans. It is possible that an OCS which is not a first degree neighbor of $OCS^{[i]}$ can answer *q* with a lower cost. However, if we allow each OCS to ask recursively all the nodes in the network in order to achieve the optimal cost, it will increase the network load and the query latency due to the message traffic, while the savings will not be substantial. Actually, the difference between the cost of the execution plan produced by the heuristic and the optimal cost, decreases as the network cost increases, as we show in the experimental section. This is exactly the case where the presence of a cache server is important.

(iii) $OCS^{[i]}$ does not have the exact answer v_q of *q*, but it contains in the cache an ancestor *u* of v_q which can be used to derive the answer for *q*. The naive approach is to read *u* from the local cache, perform the necessary aggregations and return the result to the user. However, this is not always a good execution plan. Consider the case of Figure 5, where the client asks a query which is *GROUP BY T*. Since the client knows only $OCS^{[1]}$ it sends the query to it. $OCS^{[1]}$ does not have *T* in its cache, but it can compute it from *CT* with $total\_cost = cost(T,\{CT\}) = 100$. However, if $OCS^{[1]}$ redirects the query to $OCS^{[2]}$ which has *T* in the cache, the query can be answered with $total\_cost = cost(T, \{T\}) + network\_cost(OCS^{[2]} \rightarrow OCS^{[1]}, T) = 10 + 2 \cdot 10 = 30$. In order to construct a good execution plan, we employ the same idea as in case (b); $OCS^{[i]}$ polls only its first degree neighbors and decides whether it will answer the query locally or it will redirect it, based on the *total_cost* metric. Note that in contrast to case (b), if none of the neighbors can answer *q*, the search does not continue and *q* is executed locally. In the experimental section we show that the performance of this heuristic depends on the tightness of the network, i.e. how many network connections exist among the OCSs. Note that in a real system, the network cost minimization is usually more important, because the transfer rate in Internet environments may change dynamically and drastically. In order to further reduce the network at the expense of calculation cost, we implemented a heuristic similar to the one proposed by [BPT97]: if the total cost of answering a query locally is less than α times worse than the total cost of using the best neighbor, the result is calculated locally. In our experiments we set α to 10%.

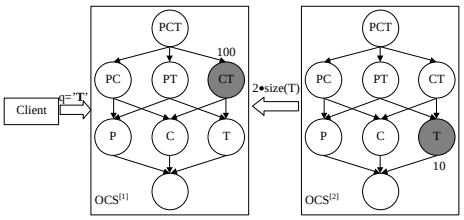


Figure 5: $OCS^{[2]}$ answers q="T" cheaper than $OCS^{[1]}$

The autonomous policy is scalable, easy to implement, and provides high availability to the system, since any OCS can dynamically enter the network or go off-line without affecting the correct operation of the other sites. Therefore, it is suitable for large unstructured and dynamically changing environments, like the Internet. This policy however, does not guarantee a good execution plan. Bellow, we present a centralized policy which always produces the optimal plan.

3.2.2 Centralized and semi-centralized policies for the query optimizer

In the distributed databases literature, it is common to have centralized control [OV99] by a site *S* that keeps information about the contents of all sites as well as the network costs. When an OCS receives a query it sends it to *S* which decides about the optimal execution plan and redirects the query to the optimal OCS. Together with the query, S also sends routing information, i.e. the complete path that the result must follow from the OCS back to the client. Essentially, every time there is a change at an OCS status (i.e. the contents of its cache, or the network connections) the central site must be notified. Although this method can lead to optimal execution plans, it is not suitable for large installations mainly because of network overloading due to the messages, low availability and low scalability of the system. It is, however, an excellent policy for small-scale systems, like enterprise intranets.

The query optimizer of the semi-centralized policy is exactly the same as the one for the centralized case. As we discuss at the next section, the two policies only differ on their caching strategies.

4. CACHING POLICY

We mentioned before that when a client submits a query, it follows a path through OCSs until some cost criteria are met in an OCS which eventually answers the query. The results follow the same path back to the client (recall that this happens for the autonomous policy; for the other two policies, the routing of the result is decided by the central site). All the OCSs which were involved in this process (including the one that answered the query) have the chance to cache the results. Note that the same results can be cached in more than one OCS. The caching policy is different for each one of the three control policies we presented before. Bellow, we describe the replacement and admission algorithms for each policy.

4.1 Caching algorithm for the autonomous policy

When the cache is full and new results need to be cached, some of the previous cached views must be evicted. Traditional page replacement algorithms such as LRU and LFU are not suitable in the OLAP environment since they don't consider the cost differences to calculate results at different levels of aggregation, and the varying sizes of the results. A more suitable alternative is presented in [SSV96]. Let *f(v)* be the frequency of queries that are answered exactly by view *v*. Let *size(v)* be the size of *v* and *cost(v)* be the cost to calculate *v*. The *goodness* of *v* is defined as:

$$goodness(v) = \frac{f(v) \cdot \cos t(v)}{size(v)}$$

The view with the lowest *goodness* is the first to be evicted. The intuition is that when it is very costly to recompute a view, this view should stay in the cache. If the cost of two views is the same, then the larger one should be evicted, since it frees more space in the cache. [KR99] employ the same idea and name this policy *Smallest Penalty First* (SPF).

The drawback of SPF is that it does not consider the interdependencies among the views in the data cube. A new result may or may not be beneficial, depending on the views that are already cached. The *benefit* metric was used in [HRU96] for the view selection problem. Let L be the data cube lattice and $v \in L$. Let $M \subseteq L$ be the set of views which are already in the cache and $V \subseteq M$ be the set of cached views that can answer v. The cost to calculate v from $u \in V$ is: $cost(v, u) = size(u)$. The cost to calculate v in the presence of M is defined as $\cos t(v, M) = \min_{u \in V} (\cos t(v, u))$. Then, the total cost of calculating all views in L, is:

$$\cos t(L) = \sum_{v \in L} \cos t(v, M)$$

Let u be a view in L. The benefit of storing u in the cache is the cost difference between *cost(L)* when u is not cached and *cost(L)* after u is cached, scaled by the frequency of the queries that are affected:

$$benefit(u) = \sum_{v \in L} f(v) \cdot [\cos t(v, M) - \cos t(v, M \cup \{u\})]$$

Then the *goodness* of u is the *benefit* normalized by the size:

$$goodness(u) = \frac{benfit(u)}{size(u)}$$

We call this policy *Lowest Benefit First* (LBF). The same metric has been used in [DRSN98] but since they do not allow the cached results to be used in computing new aggregations, their policy is identical to SPF.

A tricky part in the above formulas, is the computation of *cost(v)*. In a centralized environment, it is always possible to have precise cost estimations. In the presence of multiple OCSs however, unless v is already in the local cache, it is difficult to calculate the cost accurately because, as was mentioned in the previous section, v or an ancestor of it may be cached in many sites in the network and can be accessed with various costs. A solution would be to follow the same method as the query optimizer, i.e. to poll the first degree neighbors of $OCS^{[i]}$ and propagate the polling appropriately if none of the neighbors contains a superset of v. Although this is a viable alternative for the query optimizer, we cannot apply it to calculate the benefits because the number of control messages will grow dramatically. Essentially, any admittance or eviction will have to propagate through the network in order to update all OCSs. To solve this problem, we used the following heuristic: For every view $v \in L$, if v can be answered by a view $u \in M$ where M is the set of locally cached views, then $cost(v) = cost(v, M)$. Else we assume that v cannot be answered by any other OCS and has to be redirected to the warehouse. Therefore,

$$cost(v) = cost(v, WH) + network\_cost(warehouse \rightarrow OCS^{[i]}, v)$$

where *WH* is the set of materialized views in the warehouse. Since we expect that the set *WH* of materialized views in the warehouse will not change frequently (typically the warehouses pre-materialize statically a set of views), it is safe to assume that all OCSs can request and keep information about *WH*. Therefore, they can compute *cost(v, WH)* without contacting the warehouse.

The complexity of computing the benefit of a result in one OCS is $O(|L|)$, because the addition of a new view in the cache can potentially change the cost for every view in the data cube lattice. Nevertheless, this is the worst case. In order to compute the benefit more efficiently, we maintain the following information for each view v in the lattice: (i) the $cost(v,M)$ of answering a query that corresponds to this view, and (ii) the ancestor view $A(v,M)$ used to answer queries that correspond to v. Initially the selection set is empty and $A(v,M)$ = NULL, $cost(v,M)$ = |DW| for each v, denoting that queries that correspond to v are answered directly from the warehouse. When v_m is selected for materialization, information about the other views is updated by a procedure (see Figure 6) that visits the descendants of v_m, updating information to those that may benefit from it. This recursive function initially takes as parameters the triple ($\{v_m\}$, v_m, $\Delta C = 0$). If a v is not benefited by v_m, then the descendants of v need not be checked. Otherwise, the benefit of using v_m may propagate to the lower lattice level. In order to avoid testing a node twice, we mark the set of nodes already visited because of the materialization of v_m. The benefit (cost reduction) of adding v_m is accumulated in ΔC.

Let $M = \{v_1, v_2, ..., v_n\}$. Assume that the cache is full and we need to free space of size s in order to accommodate a new result. We can compute the goodness of all $v_i \in$ M in time $O(n \cdot |L|)$ and order them in ascending order. We select the first k of them such that their total size is enough to free space s, and we mark them as possible victims. Observe however, that if a view v_i is evicted from the cache, the goodness values of the rest and consequently the order of eviction, may change. We address this problem by using a greedy algorithm, which selects at each iteration one victim and recalculates the goodness values for all the remaining views. The idea is similar to the one used in [HRU96]. The drawback is that the time complexity of the algorithm grows to $O(n^2 \cdot |L|)$, but by employing an algorithm similar to *update_addition* the overhead in the average case is acceptable.

Note that this is still a heuristic. The optimal solution is to select this combination of views that frees enough space and the combined benefit is minimal, but this is equivalent to the optimal view selection problem, which has exponential complexity.

```
1   function update_addition(ViewSet V, View v_m, int ΔC) {
2       nextV = ∅; subV = ∅;
3       FOR each v ∈ V { // for each potential affected node at this lattice level
4           IF (cost(v,M) > r_m) THEN { //  if the cost of answering queries using v_m is smaller than the current cost
5               A(v,M) = v_m; // update v's ancestor used to answer queries
6               ΔC += f_v·(cost(v, M) - r_m); //accumulate benefit of materializing v_m
7               cost(v,M) = r_m; // update v's cost
8               add all children of v to nextV; //they should be checked at the next level
9           } ELSE add all children of v to subV; //they should not be checked at the next level
10      } // END FOR
11      nextV = nextV - subV; // remove from consideration children of non-benefited views
12      IF nextV ≠ ∅ THEN update_addition(nextV, v_m, ΔC); //update the affected nodes at the lower lattice level
13  } // END update_addition
```

Figure 6: update_addition function

For each query q the system updates the frequency $f(v_q)$ of the corresponding view. When a new result v_R is available, the system checks whether v_R is already in the cache. If not, it checks if there is enough space in the cache to store v_R. If v_R cannot fit in the available space, a set of possible *victims* is selected as described above. The combined goodness *CG* of the *victims* set is:

$$CG(victims) = \frac{\sum_{v \in L} f(v) \cdot [cost(v, M - victims) - cost(v, M)]}{\sum_{u \in victims} size(u)}$$

Observe that the *combined goodness* of the victims set can be less than the sum of the *goodness* for all victims. The new result is admitted in the cache only if $CG(victims) < goodness(v_R)$. In this case, all the views in the *victims* set are evicted from the cache, and an algorithm similar to the *update_addition* is called to update the accumulated costs in the affected views of *L*.

4.2 Caching algorithm for the centralized and semi-centralized policies

In the centralized policy, all the decisions about caching are taken at the central site. Recall that the central site generates the path that the results will follow back to the client. For each OCS in the path, the central site uses a method like the one described in the previous policy, to compute the goodness of each cached result. The problem is that since the global information about the OCSs is used, the above algorithms will be too expensive due to the potential huge problem size. Therefore, we don't use the LBF policy, but rather we employ the SPF policy where the computational cost of the *goodness* is smaller and the algorithm scales better for larger systems. As shown in the experimental evaluation section, the performance difference between LBF and SPF is not significant, so the application of SPF is an acceptable trade-off.

The semi-centralized policy attempts to combine the benefits of the previous two policies. As we already mentioned, it uses the same query optimizer as the centralized policy. However the cache admission and replacement algorithms are the same as the autonomous policy. We show in our experiments that the semi-centralized policy is only marginally worse than the centralized one.

5. UPDATES

Updates in the operational databases are not reflected in the data warehouse in real-time; instead they are deferred and applied in large batches, during regular update windows. While the data warehouse is being updated, it is not available for user queries. During the update phase, both the fact tables and any existing materialized views are updated.

Our architecture, however, does not have an explicit update phase. Instead, during the update window of the warehouse, the OCSs continue their on-line operation using the data in their local cache. If a query reaches the warehouse, it is rejected; else the old data are used. After the warehouse goes back on-line with the new data, the results that are already cached in the OCSs are marked as OUT_OF_DATE. This can be achieved either by a message which is broadcasted by the warehouse, or by setting a VALID_UNTIL tag when the results arrive at the cache.

The data warehouse uses the changes of the base tables' data, to compute a set of summary-delta tables [MQM97]. The deltas can be computed in a pre-processing stage, before the beginning of the update window, and are stored in the warehouse. Since it may be too expensive to compute and store all the possible deltas for the data cube, only the deltas for the views that are materialized at the warehouse are considered. The deltas are then treated as normal data at the OCSs. In particular, an OCS can request delta tables from its neighbors and may cache them. It can also compute new deltas from their ancestors since the summary deltas can also be considered as views in the lattice [MQM97].

Recall that the outdated data are not evicted from the cache. Assume that a query q comes, and $OCS^{[i]}$ decides to answer it by using view v from its local cache, but v is outdated. $OCS^{[i]}$ must now decide whether it is going to request for the new version v_{new} of v, or for the corresponding delta δv. Both v_{new} and δv are treated by the system as normal queries, so we use the previous cost models. In the case of δv, however, we must also add the cost of updating v with the new values. If v and δv are sorted according to the same key, the updating of v can be calculated by merging v with δv, so the cost is:

$$total\_cost(q) = total\_cost(\delta v) + size(\delta v) + size(v)$$

Observe that both v_{new} and δv can be calculated potentially by more general views in the local cache, therefore $OCS^{[i]}$ first checks its local cache and if the data cannot be found there, then the query is propagated to the neighbours, as explained above. The updated view v_{new} is not cached by default, since its size may increase due to new insertions and cannot fit in the available cache. Instead, it is forwarded to the admission control unit and is treated as a normal result. The same happens with the δv; however

all deltas are evicted from the cache after one update period, i.e. after the warehouse is loaded with the new results. Since the deltas of the previous periods are evicted, we also have to evict all the views that were outdated but didn't have the chance to be updated during the last period.

6. EXPERIMENTAL EVALUATION

In order to test our architecture, we developed a prototype that implements the algorithms described above, except that there is no underlying DBMS. Therefore, the execution and network costs are estimations based on the previous cost model. In our experiments we employed datasets from the TPC-H benchmark [TPC], and the APB benchmark [APB]. We used a subset of the TPC-H database schema consisting of 14 attributes while the fact table contained 6M tuples. For the APB dataset, we used the full schema for the dimensions (Figure 7), but only one measure. The size of the fact table was 1.3M tuples. The sizes of the nodes in the lattice were calculated with the analytical algorithm of [SDNR96]. All experiments were run on an UltraSparc2 workstation (200MHz) with 256MB of main memory.

In our experiments we varied the cache size (C_{max}) from 1% to 5% of the size of the full data cube (i.e. the case where all nodes in the lattice are materialized). The cache size was the same for all OCSs. For the TPC-H dataset, 1% of the data cube is around 10M tuples while for the APB dataset it is almost 0.58M tuples. A set of views were pre-materialized in the data warehouse using the *GreedySelect* algorithm [HRU96]. We allowed 10% of the full data cube for pre-materialization and we provided *GreedySelect* with accurate statistics about the expected workload. We assumed that the set of the pre-materialized views satisfied the maintenance-time constraint of the warehouse.

The *Detailed Cost Saving Ratio* (DCSR) [KR99] was employed to measure the results. DCSR is defined as:

$$DSCR = \frac{\sum_i w\cos t_i - \sum_i \cos t_i}{\sum_i w\cos t_i}$$

where, in our case, $cost_i$ is the total cost of answering query q_i (i.e. it includes both the calculation and the network cost) and $wcost_i$ is the total cost of answering q_i without the presence of any caching (i.e. the clients send all the requests to the warehouse).

In Figure 8a we show the network configuration we used for our experiments. There are three OCSs which are connected with the warehouse. The numbers on the edges represent the ratio of the cost to transfer one tuple through the network, over the cost to bring it from a local disk to the memory. Assuming that the transfer rate of a disk is about 1.5MB/sec [AAD+96], the links

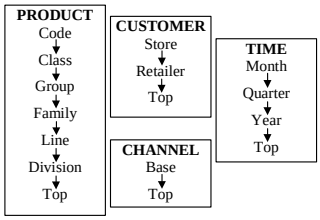


Figure 7: The dimensions of the APB dataset

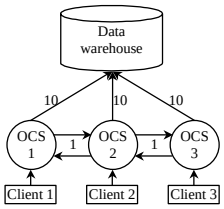
from the OCS to the warehouse equal to 1.2Mbps (i.e. a T1 connection). There are also links among the OCSs which are faster (i.e. about 12Mbps). Figure 8b depicts the same network but we assume a traditional client server architecture where each client has its own cache. Obviously the local caches do not cooperate, therefore we omit the links among them. In Figure 8c we have a client-server architecture but there is no cache in the network. We used this configuration to calculate the worst-case cost $wcost_i$.

Our query set contained a set of 1500 queries for each client. We selected the queries randomly, by using a zipf distribution, in a way that queries at the lower levels of the lattice occur at a higher probability than more detailed queries at the higher levels of the lattice. Each client box in Figure 8, represents a set of clients which are connected to the same OCS. In order to simulate this, we selected the queries at the same level following a uniform distribution without introducing any hot spots in the query set.

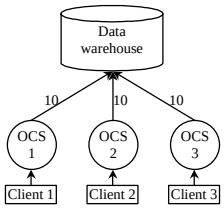
6.1 Comparison of the caching policies

In the first set of experiments, we compared LBF with SPF, LRU and LFU in terms of *DCSR*. In order to obtain comprehensive results we employed the network configuration of Figure 8b. The results are shown in Figure 9a. LRU and LFU performed slightly worse than SPF and are not presented in the diagrams. LBF performs better than SPF but the difference is significant only in some cases. Recall that LBF is computationally more expensive and does not scale well when the size of the lattice grows. Therefore, it is reasonable to use LBF for the semi-centralized and the autonomous cases, because the lattices in each OCS are comparatively small and the computation of the *benefit* values is distributed, so the overhead is affordable. On the other hand, for the centralized case LBF is too expensive, so we use SPF instead without compromising significantly the quality of the results.

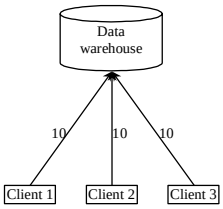
The *DCSR* measure alone can be misleading, since it is computed as the sum of the cost of calculating a result plus the cost of transferring it through the network. Therefore, a low *DCSR* value may be due to very low calculation cost but high network cost. In



(a) The full network configuration

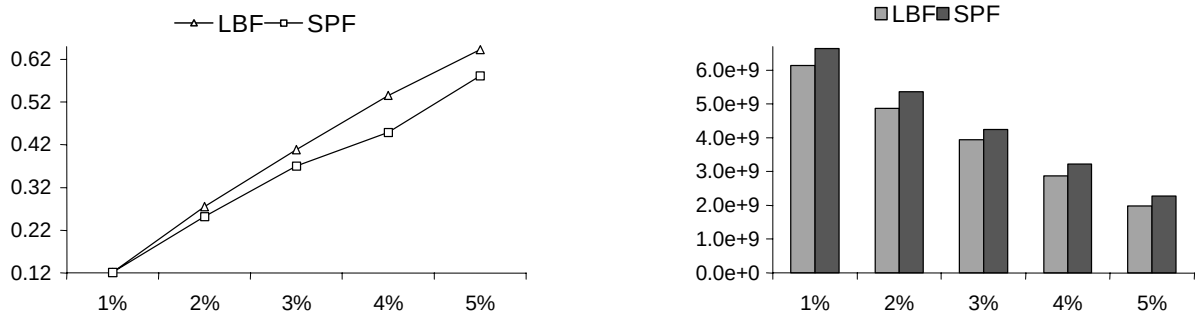


(b) The configuration that is used by a traditional client-side-cache system



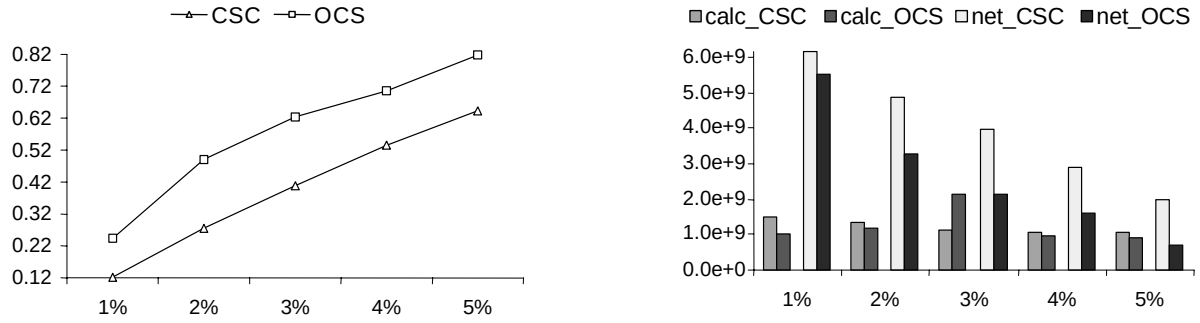
(c) The worst case configuration, where no caching is allowed

Figure 8: The tested network configuration



(a) DCSR vs C_{max} **(b)** Network cost vs C_{max}

Figure 9: The client-side-cache system with the APB dataset



(a) DCSR vs C_{max} **(b)** Calculation cost and network cost vs C_{max}

Figure 10: Comparison between the OCS architecture and the client-side-cache configuration

the Internet environment, however, where the quality of the network connection can vary widely, it is important to minimize the network cost. In Figure 9b we present the estimated network cost for the LBF and SPF policy. Again the LBF policy is better.

The ratio of the size of the complete data cube over the size of the fact table is around 22 for the APB dataset and almost 140 for the TPC-H, so the later dataset is sparser. Recall that the queries are biased towards the lower parts of the lattice where the size of the views is usually small. Due to the sparseness of TPC-H, even when the cache size corresponds to 2.5% of the data cube, there is enough space in the cache for many small results, which constitute the majority of the workload. Therefore the *DCSR* metric grows fast. The maximum value of *DSCR* (i.e. when the cache is infinite) is 88% for TPC-H, while for the APB dataset it is 89%. In the APB dataset, the transitions are smoother, due to the higher density. Due to lack of space we only present the results for the APB dataset. The trend of the results for TPC-H was similar, but they were obtained for lower values of $C_{max.}$

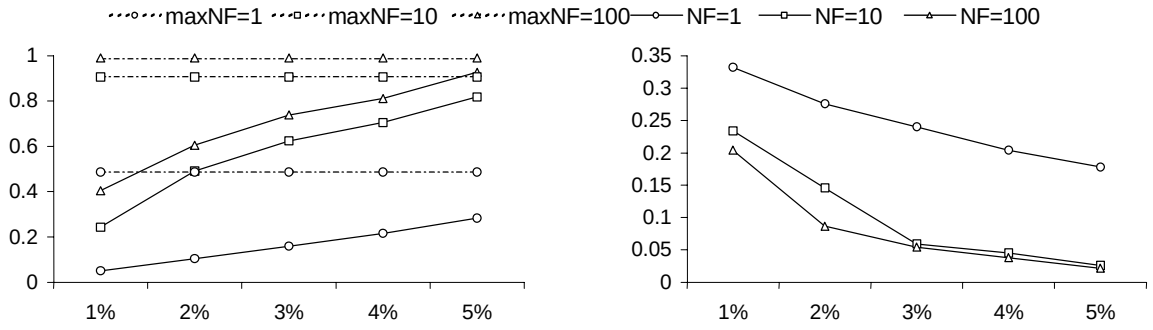
6.2 OCS vs. client-side-cache architecture

In the next experiment we compare the OCS architecture (Figure 8a) with the traditional client-side-cache (CSC) configuration (Figure 8b). The results are summarized in Figure 10a. We used the LBF admission policy for both configurations, and for fairness we used the autonomous control policy for the OCS architecture. The OCS architecture clearly outperforms CSC in terms of *DCSR*, by exploiting the contents of the near-by caches through the fast network connections. In Figure 10b we plot separately the calculation and the network cost. Observe that the OCS architecture may compromise the calculation cost (i.e. it may use a neighbor OCS that has cached a very detailed view) in order to achieve lower network cost. This behavior is desirable as explained above.

We stretch here that we don't argue against the client-server architecture with a client-side cache. On the contrary, the OCS architecture is complementary to CSC and they can easily coexist. Consider the case where each client has its own cache and there is also a network of OCSs where the client connects. Obviously the client can benefit both from its own cache and from the results that have been cached in the OCSs. Naturally, the performance gain of using the OCS infrastructure is larger when there is no client side cache, as in a web browser.

6.3 Effect of the network cost

In the next experiment we evaluate the effect of the network cost in our architecture. Again, we use the configuration of Figure 8a except that we vary the network parameters. Let *NF* be the network cost factor between the data warehouse and each OCS. *NF* takes values from 100 down to 1 resulting to a transfer rate from 120Kbps up to 12Mbps. The network cost factor among the OCSs remains constant and equal to 1. The results are summarized in Figure 11. As expected, *DCSR* is higher for slower networks. When the network is slow, if a result is found in a nearby OCS, the cost of fetching it is substantially lower than transferring it from the warehouse. Therefore the savings accumulate fast and result to high values for the *DCSR*. On the other hand, when the network is fast, it doesn't make a big difference where the result is coming from, so *DCSR* is lower. In Figure 11a we plotted *DCSR* versus C_{max} for *NF* = 1, 10 and 100 and we also plotted the maximum values of *DCSR* when the cache



(a) DCSR vs C_{max} for NF = 1, 10 and 100 **(b)** Hit ratio of the warehouse vs C_{max} for NF = 1, 10 and 100

Figure 11: The effect of the network cost

is infinite. An interesting observation is that for fast networks, *DCSR* approaches its maximum value at a slower rate than when the network speed is lower. In order to explain this, we measured the hit ratio of the warehouse (Figure 11b).

The hit ratio *HR* for a site *i* is defined as:

$$HR(i) = \frac{number\ of\ queries\ answered\ by\ site\ i}{total\ number\ of\ queries}$$

The hit ratio by itself is not a reliable indicator of the system's performance. To illustrate this, consider the plots of *HR* for *NF* = 10 and *NF* = 100 and observe that they are almost equal for $C_{max} \geq$ 3%, while the respective *DCSR* values differ notably. The reason is that *HR* does not consider either the size or the cost to calculate a result. Nevertheless, we can see that the faster the network connection is, more queries are redirected to the warehouse. Recall that the warehouse has pre-materialized 10% of the data cube, therefore there is a high chance that it has the exact answer to the query. Since the network is fast, the total cost to calculate and transfer the query is lower than using an ancestor of the query that may be cached at an OCS. Consequently, the benefit due to the existence of OCSs is lower and *DCSR* grows at a slower rate.

6.4 Autonomous vs. centralized and semi-centralized policies

The previous results show that even for very small networks, the proposed architectures can achieve substantial improvement. The next set of experiments simulates real-life situations, where the number of nodes ranges from 50 to 250. The first experiment evaluates the performance of the centralized, the semi-centralized and the autonomous control policy. The configurations are chosen as follows: We first create a graph *G* containing *n* nodes, where each node corresponds to an OCS. We consider *G* to be undirected, i.e. if there is a connection from node *i* to node *j* there is also one from *j* to *i* with the same cost. Initially *G* has no edges. We then insert a new node *dw* which corresponds to the data warehouse, and we add an edge (i.e. network connection) from every node in *G* to *dw*, having cost equal to 100. Therefore each OCS has a direct connection to the warehouse through a slow link. Each OCS is also connected to a client (see Figure 12 for an example). Let *tightness* be:

$$tightness = \frac{number\ of\ edges\ in\ G}{maximum\ number\ of\ possible\ edges\ in\ G}$$

For each instance of *G*, we select a tightness value, and we add

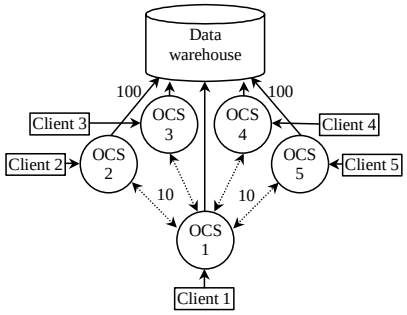
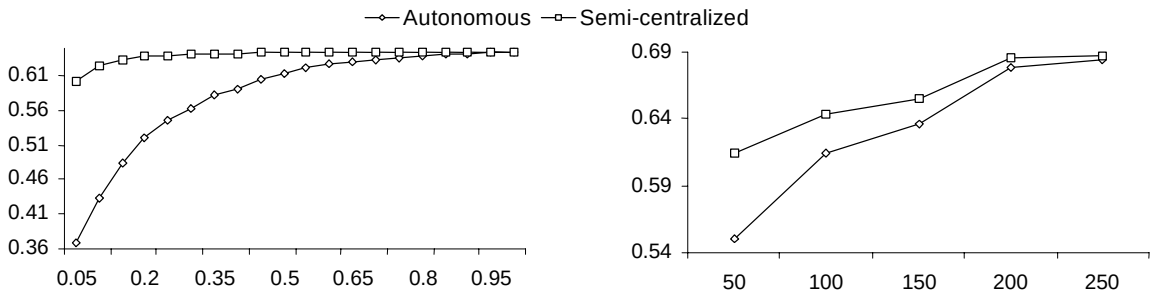


Figure 12: An example configuration.

randomly edges until this value is met. In our experiments we created graphs whose *tightness* varied from around 0.05 (star topology) to 1 (clique).

Our results are presented in Figure 13a and 13b. For the first diagram, we used a network containing 100 OCSs and we set C_{max} to 0.5%. The diagram presents the *DCSR* value for various values of *tightness* for the autonomous and the semi-centralized control policies. When the *tightness* is high, most of the OCSs are connected to each other with fast links, so there is a high chance for the autonomous policy to identify the optimal way to answer a query. When the *tightness* decreases, however, the probability to construct the optimal execution plan decreases. Consider a scenario where node *i* is connected to node *j* which is connected to *k*, but there is no direct connection from *i* to *k*. If *i* can answer a query at a lower cost than *j*, the autonomous policy will not check node *k*, which may be able to answer the query with the minimum total cost. The semi-centralized policy, on the other hand, can always identify the optimal execution plan; therefore its performance is better or in the worst case equal to the autonomous one. Observe that for small *tightness* the performance of the semi-centralized policy drops slightly. This happens because for some nodes the only edges that exist are the expensive ones towards the warehouse. Therefore, these nodes are isolated and cannot take advantage of the contents of the other OCSs.

We don't present at the diagram the centralized policy, because it was only marginally better than the semi-centralized one. Recall that the only difference between these policies is that the cache admission policy is centralized for the former approach and distributed for the other but in both cases only the OCSs that are in the optimal routing of the result, are considered. It turns out that even if an OCS takes a bad decision when admitting a result



(a) DCSR vs tightness for a network of 100 OCSs and C_{max} = 0.5%

(b) DCSR vs the number of OCSs. The tightness of the network is set to 50% and C_{max} = 0.5%

Figure 13: Comparison of the control policies

in the cache, this can be balanced by latter decisions and does not affect much the overall performance. Choosing the best execution plan, however, is more important, as the difference between the semi-centralized and the autonomous policy implies.

In Figure 13b, we keep the tightness constant and equal to 0.5 and we vary the number of OCSs in the network from 50 to 250. When the number of sites increases, so does the total cache size in the entire network; therefore there is a higher probability that a given query has already been cached in an OCS. Consequently, the performance of both policies increases. The difference between the two policies is larger when the number of OCSs is small, because the probability that the optimal cost can be achieved at a near-by site is lower[1]. This fact imposes a disadvantage to the autonomous policy. Observe, however, that when the number of sites increases, the performance of the two policies converges. This implies that we can overcome the scalability problems of the semi-centralized policy by replacing it with the autonomous policy when the system becomes substantially large, without compromising the performance significantly.

6.5 Updates

In the last set of experiments, we evaluate our update method. We compare it against invalidating all the affected cached data after an update at the warehouse contents. Recall that we required from our system to be functional even during the update period of the warehouse, therefore we did not consider any incremental update method that requires the system to go off-line. Our experiments run for the configuration of Figure 8a, with the autonomous control policy, and C_{max} was set to 3%. We assumed that the warehouse is updated once per day, and 2% of the fact table is affected (this equals to 26K tuples). We plotted the trace of *DCSR* for a two days period for the invalidate-all and the on-line incremental update methods respectively. The system starts at day 0. Since the cache is empty, at the beginning the value of *DCSR* is low, but after time passes, more results are cached and subsequent queries can be answered by the cached data. Therefore *DCSR* grows fast, until it reaches its highest value for the given cache setting (i.e. about 61% for the tested case) and stabilizes at that point.

At the beginning of day 1, the invalidate-all method, evicts from the cache all the views that were affected by the update.

[1] By definition, if the *tightness* is constant, the edges among OCSs grow quadratically when the number of nodes increases.

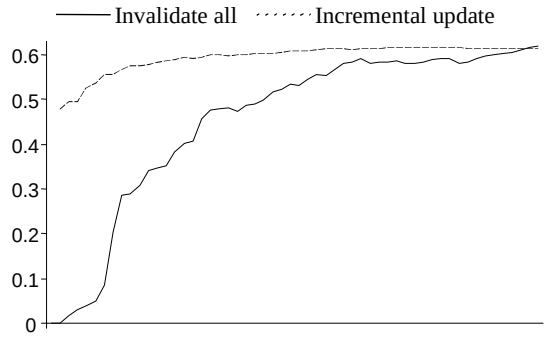


Figure 14: Trace of DCSR over time for the second day

Therefore, the cache needs again a training period, during which, many queries cannot be answered locally and *DCSR* drops. Naturally, after some time the cache contains enough data and *DCSR* starts growing again to its highest value. In Figure 14 we isolated the results for day 1, and we calculated *DCSR* starting from the beginning of day 1. When using the invalidate-all method, the system takes more time until it is trained. During this period many queries have to be answered by the warehouse at a higher cost. Although in both cases the system finally achieves the same performance, for the invalidate-all method, the early users do not benefit significantly from the caching. On the other hand, the on-line update method exhibits more balanced and stable behavior.

Note that if the number of queries per day is small and the invalidate-all method is used, there are not enough training data for the cache, so it cannot reach the maximum value; therefore the incremental update method is much better. On the other hand, if the number of queries is large, the percentage of the results that will be affected from the different update methods drops. Another remark is that the performance of the incremental update method depends on the percentage of the data cube that will be affected by the updates. Recall that the incremental update method attempts to minimize the cost by transferring only the deltas. If the size of the deltas is comparable to the size of the whole views, then the incremental updating doesn't pay off, since it doesn't only need to transfer a big delta, but also to merge it with the outdated cached result.

7. Conclusions and Future Work

In this paper we presented an architecture for caching dynamically results from OLAP queries in a network of cooperating OLAP

cache servers. We presented an admission policy based on the benefit of the cached result for all its descendants and we proposed three control policies for the OCSs, namely the centralized, the semi-centralized and the autonomous, which are suitable for different configurations of the network. We also proposed an on-line update mechanism where the deltas of the data are treated as normal cacheable results and the updated views are computed on the fly. We illustrate the efficiency of our methods by extensive experimentation, using the APB and TPC-H benchmarks.

Our architecture is beneficial for ad-hoc, geographically spanned and possibly mobile users, who sporadically need to access some data warehouses. Typical examples are non-professional investors from all over the world who access the warehouses of various stock markets through their web. Existing client-server systems that incorporate a client-side cache can also benefit by the OCS infrastructure, by using them as a middle layer. Note that this architecture is not intended to act as a distributed database system for warehousing, although it can cooperate with such systems.

Currently, our prototype does not support a DMBS. We are working towards connecting it with an underlying DMBS and we are also planning to support multiple unrelated data warehouses by the same OCS. This introduces many challenges like what metric should be used for the admission policy, since the data characteristics may vary significantly. Other open problems include the admission policy of dimension tables, the required metadata and also the design of more efficient update algorithms, possible based on prefetching.

ACKNOWLEDGMENTS

This work was supported by grants HKUST 6070/00E and HKUST 6090/99E from Hong Kong RGC.

REFERENCES

[AAD+96] Agarwal S., Agrawal R., Deshpande P.M., Gupta A., Naughton J.F., Ramakrishnan R., Sarawagi S., “On the Computation of Multidimensional Aggregates”, Proc. VLDB, 1996.

[AGL98] Albrecht J., Guenzel H., Lehner W., “An Architecture for Distributed OLAP”, Proc. Parallel and Distributed Processing Techniques and Applications, 1998.

[AL98] Albrecht J., Lehner W., “On-Line Analytical Processing in Distributed Data Warehouses”, Proc. International Database Engineering and Applications Symposium, 1998.

[APB] OLAP Council, “OLAP Council APB-1 OLAP Benchmark RII”, http://www.olapcouncil.org

[BPT97] Baralis E., Paraboschi S., Teniente E., “Materialized View Selection in a Multidimensional Database”, Proc. VLDB, 1997.

[CCS93] Codd E.F., Codd S.B., Salley C.T., “Providing OLAP to User-Analysts: An IT Mandate”, Technical report, 1993, http://www.arborsoft.com/essbase/wht_ppr/coddps.zip

[CZB98] Cao P., Zhang J., Beach P.B., “Active Cache: Caching Dynamic Contents on the Web”, Proc. Middleware ’98 Conference, 1988.

[DFJ+96] Dar S., Franklin M.J., Jonsson B.T., Srivastava D., Tan M., “Semantic Data Caching and Replacement”, Proc. VLDB, 1996.

[DRSN98] Deshpande P., Ramasamy K., Shukla A., Naughton J.F., “Caching Multidimensional Queries Using Chunks”, Proc. ACM-SIGMOD, 1998.

[GBLP96] Gray J., Bosworth A., Layman A., Pirahesh H., “Data Cube: a Relational Aggregation Operator Generalizing Group-by, Cross-tabs and Subtotals”, Proc. ICDE, 1996.

[GHRU97] Gupta H., Harinarayan V., Rajaraman A., Ullman J. D., “Index Selection for OLAP”, Proc. ICDE, 1997.

[GLWZ99] Garcia-Molina H., Labio W.J., Wiener J.L., Zhuge Y., “Distributed and Parallel Computing Issues in Data Warehousing”, Proc. ACM Principles of Distributed Computing Conference, 1999.

[GM99] Gupta H., Mumick I. S., “Selection of Views to Materialize Under a Maintenance-Time Constraint”, Proc. ICDT, 1999.

[Gupt97] Gupta H., “Selection of Views to Materialize in a Data Warehouse”, Proc. ICDT, 1997.

[HRU96] Harinarayan V., Rajaraman A., Ullman J. D., “Implementing Data Cubes Efficiently”, Proc. ACM-SIGMOD, 1996.

[HZ96] Hull R., Zhou G., “A Framework for Supporting Data Integration Using the Materialized and Virtual Approaches”, Proc. ACM-SIGMOD, 1996.

[KB96] Keller A.M., Basu J., “A Predicate-Based Caching Scheme for Client-Server Database Architectures”, VLDB Journal 5(1), 1996.

[Kim96] Kimball R., “The Data Warehouse Toolkit”, John Wiley, 1996.

[KR99] Kotidis Y., Roussopoulos N., “DynaMat: A Dynamic View Management System for Data Warehouses”, Proc. ACM-SIGMOD, 1999.

[MQM97] Mumick I.S., Quass D., Mumick B,S., “Maintenance of Data Cubes and Summary Tables in a Warehouse”, Proc. ACM-SIGMOD, 1997.

[OV99] Ozsu M.T., Valduriez P., “Principles of Distributed Database Systems”, Prentice Hall, 1999.

[RKR97] Roussopoulos N., Kotidis Y., Roussopoulos M. “Cubetree: Organization of and Bulk Updates on the Data Cube”, ACM-SIGMOD, 1997.

[SAL96] Stonebraker M., Aoki P.M., Litwin W., Pfeffer A., Sah A., Sidell J., Staelin C., Yu A., “Mariposa: A Wide-Area Distributed Database System”, VLDB Journal 5(1), 1996.

[SDN98] Shukla A., Deshpande P., Naughton J. F., “Materialized View Selection for Multidimensional Datasets”, Proc. VLDB, 1998.

[SDNR96] Shukla A., Deshpande P. M., Naughton J. F., Ramasamy K., "Storage Estimation for Multidimensional Aggregates in the Presence of Hierarchies", Proc. VLDB, 1996.

[SSV96] Scheuermann P., Shim J., Vingralek R., “WATCHMAN: A Data Warehouse Intelligent Cache Manager”, Proc. VLDB, 1996.

[TPC] Transaction Processing Performance Council, “TPC-H Benchmark Specification”, v. 1.2.1, http://www.tpc.org

[ZDN97] Zhao Y., Deshpande P., Naughton J.F., “An Array-Based Algorithm for Simultaneous Multidimensional Aggregates”, Proc. ACM-SIGMOD, 1997.