An Adaptive Peer-to-Peer Network for Distributed Caching of OLAP Results

Panos Kalnis[†] Wee Siong Ng[§] Beng Chin Ooi[§] Dimitris Papadias[†] Kian-Lee Tan[§]

[†]Department of Computer Science
Hong Kong University of Science and Technology
Clear Water Bay, Hong Kong
{kalnis,dimitris}@cs.ust.hk

[§]Department of Computer Science
National University of Singapore
{ngws,ooibc,tankl}@comp.nus.edu.sg

Abstract

Peer-to-Peer (P2P) systems are becoming increasingly popular as they enable users to exchange digital information by participating in complex networks. Such systems are inexpensive, easy to use, highly scalable and do not require central administration. Despite their advantages, however, limited work has been done on employing database systems on top of P2P networks.

Here we propose the PeerOLAP architecture for supporting On-Line Analytical Processing queries. A large number of low-end clients, each containing a cache with the most useful results, are connected through an arbitrary P2P network. If a query cannot be answered locally (i.e. by using the cache contents of the computer where it is issued), it is propagated through the network until a peer that has cached the answer is found. An answer may also be constructed by partial results from many peers. Thus PeerOLAP acts as a large distributed cache, which amplifies the benefits of traditional client-side caching. The system is fully distributed and can reconfigure itself on-the-fly in order to decrease the query cost for the observed workload. This paper describes the core components of PeerOLAP and presents our results both from simulation and a prototype installation running on geographically remote peers.

1. Introduction

Effective decision-making is vital in a global competitive environment where business intelligence systems are becoming an essential part of virtually every organization. The core of such systems is a data warehouse, which stores historical and consolidated data from the transactional databases, supporting complicated ad-hoc queries that reveal interesting information. The so-called On-Line Analytical Processing (OLAP) queries typically involve large amounts of data and their processing should be efficient enough to allow interactive usage of the system.

Distributed database technology is extensively used in data warehouses to access the operational databases, extract, clean and integrate the data. [6] discuss the particular issues of the data warehouse environment for distributed and parallel computation. The warehouse itself can also be implemented as a distributed database. If a central warehouse exists, standard replication methods can be used to transfer data to departmental data marts [20]. For decentralized implementations, where each department builds and independent data mart, [10] employ a global schema in a middleware to allow transparent access to all data. [1] have also proposed an architecture for executing OLAP queries over a decentralized schema. Their middleware component follows an economical approach, similar to the *Mariposa* system [25].

These systems assume that the users belong to the organization that owns the data warehouse and have access to the proprietary infrastructure. The query requirements are well defined and the problems are related to data placement, materialized view selection and query optimization, given a static network of servers.

Here we investigate a different problem: a large number of ad-hoc and geographically spanned users, access sporadically a number of separate warehouses and possibly correlate information from all of them. Imagine for instance many individual investors from all around the world, who trade stocks in the New York Stock Exchange. In contrast to professional stockbrokers, these users are unlikely to have any proprietary tool to access the stock market's warehouse, and most probably connect with a simple applet through their web browser. The primary problem in this case is not the processing of the queries in the warehouses, but rather the efficient usage of the available bandwidth, since the size of results from OLAP queries may greatly vary from a few tuples to many megabytes of data. This can be especially true if the user is not satisfied with highly aggregated information, but needs access to detailed data in order, for example, to correlate the NY prices with the ones from the major European markets.

Intuitively, the problem is similar to accessing web pages from remote web servers. Caching in web proxy-servers has been used extensively in practice to deal with the latency caused by slow network connections and accelerate the retrieval of the same URL from users at the same geographic area. [16] employs active caching techniques [2], to cache OLAP data together with web pages in web proxy-servers using a three-tier architecture. [12] employs similar ideas to implement a multi-tier caching system for OLAP queries on

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for pro£t or commercial advantage and that copies bear this notice and the full citation on the £rst page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior speci£c permission and/or a fee.
ACM SIGMOD '2002 June 4-6, Madison, Wisconsin, USA
Copyright 2002 ACM 1-58113-497-5/02/06 ...$5.00.

a dedicated infrastructure.

In this work we follow a different approach focusing on the client side. Continuing the previous example, assume that users from Hong Kong pose queries to the NYSE warehouse and some results are cached at their local computer [4, 5, 15, 21] hoping that subsequent queries can reuse this data. However, the size of each client's cache is relatively small compared to the size of the warehouse, while the network cost of transferring large amounts of data from overseas is high. On the other hand, it is possible that some other user in Hong Kong, who has fetched part of the required data recently, can be accessed through a much faster network connection. By sharing their cache contents all clients can benefit, because the available space for caching is larger and the amortized network cost is lower.

In the following sections we will describe PeerOLAP which is a distributed caching system for OLAP queries based on a Peer-to-Peer[1] (P2P) network. The contributions of this work include: (i) the proposal of the PeerOLAP architecture, (ii) the employment of three cache control policies that impose different levels of cooperation among the peers, and (iii) the development of adaptive techniques that dynamically reconfigure the network structure in order to minimize the query cost.

PeerOLAP is complementary to distributed data warehouses, which deal with the efficient execution of OLAP queries, since we focus on the effective utilization of client resources. The same relationship exists with middleware approaches like [16] and [12]. Also the traditional client-side caching in client-server systems is a special case of our system, where the client caches do not cooperate.

We focus on OLAP for several reasons: (i) OLAP data have a regular structure which allows easy decomposition and reuse of previous results, (ii) the size of the results is typically large and justifies the overhead of searching neighbor peers, (iii) updates in data warehouses are infrequent compared to transactional databases, therefore the cached data are valid for a long time and (iv) queries exhibit temporal and geographical locality; for instance many Hong Kong users are likely to request from NYSE similar data (e.g., related to the Hang Seng index).

The rest of the paper is organized as follows: in Section 2 we include some essential background and review the related work. Section 3 presents an overview of the PeerOLAP architecture. Section 4 describes the modules of an autonomous peer, the query optimization algorithm, the caching policy and the adaptive behavior of the system, while in Section 5 we present the results from the experimental study. Finally, Section 6 concludes the paper with a discussion about our future work.

2. Background

Conceptually, data warehouses deal with multidimensional views of data. Under this model, there is a set of measures that are the objects of analysis, such as *sales*. The measures depend on a set of dimensions (i.e. business perspectives); as

[1]The term 'P2P' has been used in the database literature to identify systems where each node may act both as a server and a client assuming static configuration [14]. Such systems are generalizations of the traditional client-server model and standard distributed techniques can be applied. Here, 'P2P' refers to dynamic systems with ad-hoc participation, such as Napster and Gnutella

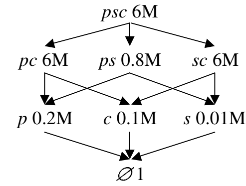

Figure 1: A data cube lattice. The dimensions are *Product*, *Supplier* and *Customer*

an example consider *product*, *customer* and *supplier*. Thus, a measure is a value in the multidimensional space which is defined by the dimensions. Each dimension is described by a domain of attributes (e.g., product ids). The set of attributes may be related via a hierarchy of relationships, a common example of which is the temporal hierarchy (day, month, year).

There are $O(2^d)$ possible group-by queries for a data warehouse with d dimensional attributes, which compose the data cube. A detailed group-by query can be used to answer more abstract aggregations. [9] introduce 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 (Figure 1).

A common technique to accelerate OLAP is to pre-calculate some aggregations and store them as materialized views, provided that some statistical properties of the expected workload are known in advance. [9, 24, 23] describe greedy algorithms for the view selection problem. These methods follow a static approach, where the views are selected once when the warehouse is set up. A dynamic approach is inspired by semantic data caching [3, 13]: instead of caching a list of physical pages or tuple identifiers, the results of previous queries together with their semantic description are stored.

For the special case of OLAP, [21] developed a semantic cache manager called *Watchman*. The 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 of re-evaluating a result and its size.

Dynamat [15] is another OLAP cache manager, which stores *fragments* instead of arbitrarily shaped query results. Fragments are aggregate query results in a finer granularity than views since they may include equality selections on some dimensions. They may be further aggregated to answer more general queries, but the data from multiple fragments cannot be combined. The caching policy is similar to Watchman.

[5] performs a regular decomposition of the multidimensional space into chunks [28], which constitute the smallest piece of cached information (Figure 2). When a query is asked, the system computes the set of chunks required to answer it, and splits it into two subsets based on whether they are cached or not. To answer the query, the system will request the missing chunks from the warehouse. Only chunks at the same aggregation level as the query are considered, i.e., no aggregation is performed on the cached results; this option, however, is exploited in an extension of their work [4]. The admission and replacement algorithms are similar to Watchman.

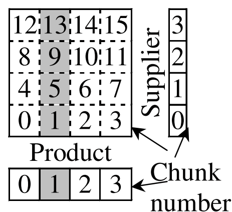

Figure 2: The ps, p and s views decomposed regularly into chunks. Chunk p_1 can be computed from $ps_{\{1,5,9,13\}}$

In PeerOLAP results are also cached in the form of chunks. Except from the positive effect due to the finer granularity of data, caching chunks has two more advantages: (i) the uniformity of the semantic regions, which allows chunks from different results to be easily combined in order to construct answers for new queries and (ii) good space utilization, since there is no overlapping of cached results.

All the systems above, adopt traditional client-server architectures with client-side or server-side caching. There is no cooperation among caches and the network factor is not considered. Alternatively, [16] proposes a middleware approach where caching of OLAP results is performed in web proxy-servers together with web pages. Since a web proxy-server cannot cache dynamic data, they employ active caching techniques [2] and cache an applet together with the data, which is responsible for deciding whether subsequent queries may use the cached results. However, the caching policy of the web proxy-server must be altered and the OLAP data must be a substantial part of the total traffic in order to achieve any practical results.

[12] extends the previous idea by employing a dedicated infrastructure called *OLAP Cache-Servers* (OCS). OCSs are similar to web proxy-servers but are optimized for caching OLAP data and have computational capabilities allowing local computation of summarized results from detailed data that were previously cached. OCSs may form an arbitrary network and can cooperate. The granularity of caching however is very coarse, since they store entire views. PeerOLAP is different in several aspects: (i) there is no special mid-tier infrastructure for caching OLAP results, but rather, each client chooses to participate in a network with other clients in order to implement a large distributed cache, (ii) PeerOLAP network is dynamic, compared to the static connections among OCSs, (iii) the granularity of caching is finer, allowing a query to be answered by partial results from many peers, where in OCS a node either can answer the entire result or cannot answer it at all and (iv) PeerOLAP can cache data from many warehouses simultaneously.

The requirements of our system match perfectly with the characteristics of the P2P technology. Most of the existing P2P systems, like Napster [17] and Gnutella [7] provide content sharing. Others, like ICQ [11] allow users to exchange personal messages, while systems like Seti-at-home [22] enable the sharing of computational power. An evaluation of P2P systems can be found in [26]. These systems, however, lack database characteristics mostly in the areas of semantics, data transformation and data relationships. Therefore, they are limited to transferring content at the object level and cannot support the execution of complex queries across multiple sources or use intermediate results in order to answer consecutive queries.

Piazza [8] is the first system to deal with database management issues in P2P systems. Each peer can have any of the following four roles: *data origin* which provides the original content, *storage provider* which stores materialized views, *query evaluator* which uses its CPU resources to evaluate a query and *query initiator* which poses new queries to the system. Piazza deals primarily with the data placement problem, i.e. the selection of strategic places to store data in order to improve query performance. Although this is also an issue in distributed databases, there are fundamental differences since P2P systems do not have a centralized schema. In addition, the membership of a peer in the system is ad-hoc and dynamic, therefore it is very difficult to predict or reason about the location and quality of the system's resources. In Piazza, the data placement problem is solved by separating logically the system into smaller *spheres of cooperation* and advertising the set of materialized views to all the nodes of a sphere.

In our case, there are similar issues, like the mechanism that should be used in order to inform others about the contents of a peer's cache. However there are also many differences: First, the finer granularity of data poses a heavy overhead to advertising protocols. Consequently, there is no public catalogue and the optimizer of PeerOLAP needs to propagate each query to the network, while Piazza can use the advertised information. Also, since Piazza is a data placement system, it can pre-fetch beneficial data to the proper nodes, while PeerOLAP is a caching system so it can only reuse data that has been previously requested. Another major difference is that in Piazza there are predefined spheres of cooperation, while PeerOLAP adapts dynamically its behavior and the network structure in order to adjust to the current workload. Finally, since PeerOLAP focuses on OLAP queries, we can employ several optimizations, which are not applicable for multi-purpose DBMSs.

In the following section we will describe in detail the architecture of the PeerOLAP network. PeerOLAP builds on and extends BestPeer [18] for OLAP applications. Briefly, BestPeer is a generic P2P system designed to serve as a platform to develop P2P applications easily and efficently. It has the following features: (1) It employs mobile agents; (2) It shares data at a finer granularity as well as computational power; (3) It can dynamically reconfigure the BestPeer network so that a node is always directly connected to peers that provide the best service; (4) It employs a set of *location independent global name lookup* (LIGLO) servers to uniquely recognize nodes whose IP addresses may change as a result of frequent disconnection and reconnection.

3. The PeerOLAP Network

The PeerOLAP network is a set of peers that access data warehouses and pose OLAP queries. Each peer P_i has a local cache and implements a mechanism for publishing its cache contents and its computational capabilities. Other peers can connect to P_i and request a result. P_i may either answer the query (or part of it) locally, if it has the required data, or propagate the query to its neighbors. In either case, all results return directly to the peer that initiated the query. The goal of PeerOLAP is to act as a combined virtual cache, where all the components offer resources aiming at achieving lower query cost.

Figure 3 depicts a typical PeerOLAP network consisting of 7 peers and two data warehouses. There is an arbitrary set

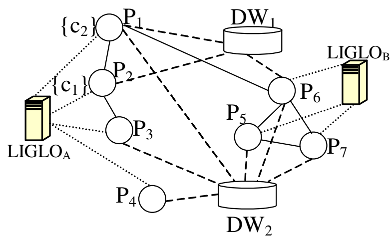

Figure 3: A typical PeerOLAP network

of connections among peers denoted by solid lines, and each peer also connects directly to one or multiple warehouses simultaneously. Assume that P_2 issues a query q referring to chunks c_1, c_2 and c_3. If c_1 is already at the local cache, P_2 will send a request for c_2 and c_3 to its neighbors P_1 and P_3. P_1 contains c_2, therefore it computes an estimation for the cost of retrieving and transferring this result back to P_2, and at the same time it forwards the request to P_6. Note that both c_2 and c_3 are requested[2] since P_6 may be able to provide c_2 with lower cost than P_1.

In order to avoid flooding the network with messages, a maximum number of hops is assigned to each message. Assuming that this number is 2, the query will not be propagated to the neighbors of P_6. On the other hand, P_3 will not forward the message although there is still one hop allowed, since a peer can direct to the warehouse only its local queries in order to avoid overloading the server with the same query. There is also a mechanism for breaking message loops: each peer keeps a queue of the recent messages and rejects the ones that had been processed before.

P_2 does not have any knowledge about the number of the peers that will respond. Therefore, it waits until all the requested chunks are found or a timer expires. Missing chunks are requested from the data warehouse. Note that although the warehouse can provide any chunk, it is the last option due to the high network cost.

After searching has terminated, P_2 decides which chunks to keep in the local cache. Each chunk is assigned a benefit value. If an incoming chunk c has a higher benefit than some cached results, these results are evicted and c is stored, else c is rejected. For chunks that were sent directly from the warehouse (meaning that they were not found in the neighborhood of P_2), we explore the option of caching them in some neighbor of P_2 (i.e. P_1 or P_3).

Since peers can enter and leave the network dynamically, a mechanism is necessary to provide the newcomer with an initial set of neighbors. Here we employ LIGLO servers to maintain a list of the online peers, together with details about the warehouses that they access, their physical location, speed of network connection, etc. A newcomer peer P contacts a LIGLO server and gets a set of potential neighbors. Then P decides independently the set of peers that it will try to connect to. Except from LIGLO servers, the PeerOLAP network is fully distributed without any centralized administration point. Furthermore, LIGLO servers are not involved in the query processing and can be completely eliminated if the set of initial neighbors can be otherwise determined; for instance, peers on the same segment of a LAN may connect to each other.

[2]A variation, where only the not-yet-found chunks are requested, is discussed in the next section.

The set of initial neighbors is by no means optimal, since their cached results may be irrelevant to P. Furthermore, connections may be dropped as some peers leave the network. Therefore, each peer implements a mechanism which constantly evaluates the current neighbors and drops or adds peers to the neighbor list, in order to achieve lower query cost. Intuitively, peers with similar query patterns should be neighbors. In such case, if a result is not found in the initial peer, there is a high probability that one of the direct neighbors will contain it. Due to the limited availability of resources, each peer cannot have more than k neighbors, where k is a parameter of the system. Even if there were unlimited resources on a peer, it is not appropriate to have too many neighbors, since the network will be overloaded with messages, most of them being negative responses.

Here we make an effort to optimize the set of neighbors of each peer, by formulating the problem as a second level of caching. The size of the second level "cache" is the number of available network resources, while the "cached" objects are the connections to neighbors. Each neighbor is assigned a benefit and may be dropped if a more beneficial neighbor is found. Continuing our previous example, assume that during the last 10 queries from P_2, 5 chunks where found in P_1, 8 in P_6 and none in P_3. It is obviously beneficial for P_2 to have P_6 as a direct neighbor, in order to avoid the overhead of reaching it through P_1. Therefore, the connection to P_3 is dropped and is substituted with a (virtual) connection to P_6.

In the next section we describe in detail the components of our architecture, and present the query processing and caching algorithms. We also discuss the algorithm for network reconfiguration.

4. Peer Architecture

The PeerOLAP network consists of numerous low-end workstations which connect to data warehouses, pose OLAP queries and process the results. Every peer maintains a local cache and implements a P2P protocol for connecting with other caches. The application layer is separated from the cache control unit; therefore the cache is not aware about the semantics of the data. Both the creation of the execution plan and the caching policy are fully decentralized.

Figure 4 depicts the architecture of an autonomous peer. There are two basic layers: *application* and *cache layer*. The application layer implements the user interface, the query optimizer and the query execution engine. It has knowledge about the schema of the warehouse and the semantics of the data. In our implementation, the application layer is built as a Java agent. When the user connects to a data warehouse (e.g., by accessing its web site), the warehouse server sends to the peer a mobile agent which implements all the logic of the application layer. The agent then connects to the cache layer, which is already running on the peer, and all the data requests are directed through the cache layer. More than one agents are allowed to run simultaneously at the same peer if the user wants to connect to multiple warehouses. In our implementation, the logic of all agents is the same although every warehouse supplies its own schema. PeerOLAP provides an environment where different mobile agents can reside and perform their tasks. The versatility to adapt to different requirements for query optimization and execution renders the system highly extensible and powerful. Note that the application layer does not have to be

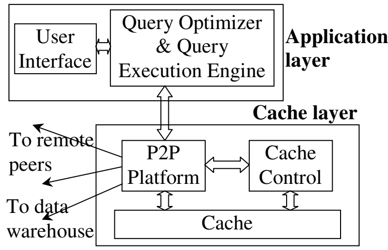

Figure 4: Architecture of a peer

implemented as an agent. Assuming that the user routinely connects to some data warehouses, the client software can be permanently installed on the local peer.

The cache layer consists of three modules:

1. The local cache, which is organized as a chunk file [5].
2. The cache control module, which implements the admission and replacement policy of the cache.
3. The P2P platform, which implements the low level communication (among the peers, and between the peer and the warehouse), the data transfer and the remote agent support. Also, in collaboration with the cache control module, it is responsible for the network reconfiguration.

Apart from simplifying the development process, there is another advantage from separating the peer in two layers: by distinguishing the cache from the semantics of the data, the cache can store simultaneously data from multiple warehouses. From the cache's point of view, each piece of data is a chunk, which is identified by a unique ID. It is the application layer's responsibility to ask for the correct set of chunks and advise the cache about the benefit of storing a specific chunk. Therefore, each peer can support simultaneous access to multiple warehouses by allowing many agents to run together. Also, a peer can store chunks that do not belong to its local warehouses, but are beneficial to some neighbors. In an extreme case, a peer may have only its cache layer running without executing any local application.

4.1 Cost Model

Let c be a chunk and $size(c)$ its size in tuples. $S(c, P)$ denotes the cost of computing c in node P. If P is a peer of the cache network and we do not allow any aggregation on cached results, then $S(c, P) = a \cdot size(c)$, where a is constant. On the other hand, if cached results can be further aggregated, $S(c, P)$ is the total cost of reading the required set of more detailed chunks and performing the necessary computations. The network cost N for transferring c from node Q to node P is:

$$N(c, Q \to P) = \frac{Cn(P \to Q)}{k} + \frac{size(c)}{Tr(Q \to P)} \qquad (1)$$

where $Cn(P \to Q)$ is the cost of establishing a connection between the two nodes, k is the number of chunks that will be transferred together in a batch operation, and $Tr(Q \to P)$ is the transfer rate between Q and P. If there is already an established connection between the two nodes, $Cn()$ is zero.

When c is asked at peer P, the peer decides the location Q from where it will request the data. Therefore, the total cost T of answering c at P by using data from Q is:

$$T(c, Q \to P) = S(c, Q) + N(c, Q \to P) \qquad (2)$$

Obviously, if the chunk exists locally (i.e. $P \equiv Q$), $N()$ is zero.

4.2 Query Processing

A query q has the form:

```
SELECT <grouping predicates> AGR(measure)
FROM data
WHERE <selection predicates>
GROUP BY <grouping predicates>
```

Let σ and γ be the set of selection and grouping predicates, respectively. View v is the representative view of q, if the set of dimensions of v is $\sigma \cup \gamma$. For example, a query that asks for the sum of sales of a set of products for each customer, corresponds to the *pc* view of Figure 1. A node in the PeerOLAP network can compute the result of such queries by first accessing the set C of required chunks at the same level of granularity as the representative view and then performing the necessary selections and aggregations on them. Here we focus on the problem of locating, accessing and caching the chunks of C, therefore we consider queries involving selections on the grouping predicates only (i.e. $\sigma \subseteq \gamma$). Furthermore, the predicates of σ are such that the results match with the boundaries of entire chunks. More general queries can be computed by post-processing the chunks of C.

We assume that the warehouses are read-only meaning that the clients cannot issue update statements. If the contents of the warehouse change, it must broadcast the relevant invalidation messages. Alternatively, it can set the expiration time in each chunks it computes.

Bellow, we will discuss two query processing policies, an eager and a lazy one, which differ on the amount of effort they put on constructing the execution plan.

4.2.1 *Eager Query Processing (EQP)*

Assume that a user issues a query q at peer P. The EQP policy answers q by performing the following steps:

1. The query is decomposed into chunks at the same granularity as the representative view. Let C_{all} be the set of required chunks.

2. P first checks its own cache. Let C_{local} be the set of chunks that are present and C_{miss} be the remaining chunks.

3. P sends a message to its neighbors $Q_1, \ldots, Q_k$ asking for the C_{miss} set. If Q_i has a subset of C_{miss}, then it estimates the cost $T(c_i, Q \to P)$ for each of the chunks and sends these estimations to P. If a peer does not have any of the required chunks, it does not respond. In any case, Q_i propagates the request for the entire C_{miss} set to its own neighbors recursively, until the maximum allowed number of hops is reached.

4. P keeps receiving responses for a period t, after which it assumes that no more results are expected.

5. Let C_{peer} be the subset of C_{miss} that was found in the PeerOLAP network. P constructs the execution plan for C_{peer} in a greedy manner: A chunk c_i is randomly selected from C_{peer} and is assigned to Q_i, where Q_i is the peer that can provide c_i with the lowest cost. Next, a chunk c_j is selected from the remaining chunks in C_{peer}. Let Q_j be the peer that provides c_j with the minimum cost. If Q_i

also contains this chunk, the algorithm checks whether the total cost $T(\{c_i, c_j\}, Q_i)$ of acquiring both chunks from Q_i is smaller than $T(c_i, Q_i) + T(c_j, Q_j)$ in which case it assigns c_j also to Q_i. The process continues for the rest of the chunks in C_{peer}. Observe that acquiring multiple chunks simultaneously from the same peer may be cheaper, because the cost of sending messages and initializing the network connections is shared.

6. P initializes direct connections to the peers defined by the execution plan and requests the corresponding chunks. The peers send back the chunks that have not been evicted in the meantime. Let $C_{evicted}$ be the set of evicted chunks.

7. The set C_{DW} of chunks still missing is: $C_{DW} = C_{miss} - (C_{peer} - C_{evicted})$. P gets these chunks directly from the warehouse.

8. P composes the answer and returns it to the user. The new chunks are sent to the cache control module and any necessary reconfiguration of the network is performed.

Only chunks at the same aggregation level as the query are considered. By exploiting the possibility of computing missing chunks by further aggregating the cached results, a more efficient execution plan may be constructed. However, in such case the number of ways to compute a chunk grows exponentially to the number of dimensional attributes and the construction of the execution plan becomes a difficult optimization problem which is outside the scope of this paper. Nevertheless, the cost model is general enough to deal with aggregations if they are performed within the scope of a single peer.

4.2.2 *Lazy Query Processing (LQP)*

The previous policy attempts to expand the search space as much as possible in order to locate the maximum number of chunks. The drawback, however, is that the system is overloaded with messages, many of which are redundant, either because some of the accessed peers are irrelevant to the query, or because multiple peers contain the same chunk, but their cost difference does not justify the high message overhead. Here we present a second policy, called *Lazy Query Processing* which tries to reduce the number of visited peers.

LQP is similar to EQP except from step 3; P sends the request to all of its neighbors $Q_1, \ldots, Q_k$, but each neighbor will propagate the request only to its most beneficial neighbor. In addition, if Q_i can answer some of the chunks, it removes them from the propagated message. As a result, if the entire query can be answered by Q_i, the message is not propagated. The process is repeated until the maximum allowed number of hops h_{max} is reached. If each peer has k neighbors the number of messages are $O(k \cdot h_{max})$ while for EQP this number becomes $O(k^{h_{max}})$.

We already mentioned that the new chunks are forwarded to the cache control module, which decides whether it is beneficial to store some of them locally. The next paragraph explains this issue; the notion of a peer's benefit will be clarified in Section 4.4, where we discuss the adaptive behavior of the system. The LQP policy fits well in this concept since intuitively we wish to form small sub-networks with similar query patterns.

4.3 Caching Policy

In order to define the cache control policy, a benefit metric $B()$ is assigned to every chunk c at a peer P. Naïve LRU or LFU schemes are inapplicable for OLAP queries because the cost of computing chunks varies greatly at different levels of aggregation. [21] defines a metric, which is a function of the cost to compute a result normalized by its size and frequency. The same metric is used in [15]. [5] proposes a caching algorithm, called ClockBenefit, which is a generalization of LRU. The benefit of a chunk is measured by the fraction of the base table that it represents. Therefore, if there are n chunks in a view v, the benefit of each chunk is $\frac{|D|}{n}$, where $|D|$ is the size of the base table. Since the number of chunks at higher levels of aggregation is small, they have a higher benefit. The benefit is thus proportional to the cost of computing a chunk. The exact cost is not important in their case, since the back-end always computes each chunk from the base tables and also the cache does not perform any aggregation.

Here we define the benefit $B()$ of a chunk c in a peer P as:

$$B(c, P) = \frac{T(c, Q \to P) + a \cdot H(P \to Q)}{size(c)} \quad (3)$$

where $H(P \to Q)$ is the number of hops from peer P to Q and a is a constant representing the overhead of sending one message. Intuitively, a high value of $H()$ denotes that it is difficult to locate a result, therefore it is more beneficial to keep it locally. Notice that the cost of locating a result is proportional to the number of hops rather than the number of peers visited, since a peer sends each request to all its neighbors in parallel. The benefit value is normalized by dividing the total cost of obtaining a chunk by its size.

Recall that $T()$ is the total cost of computing and transferring a result; its inclusion in the benefit denotes that results which are expensive to obtain, should be stored locally. Although our caching algorithm is similar to ClockBenefit, the $\frac{|D|}{n}$ metric is not suitable, since we allow pre-aggregation at the data warehouse. Therefore, the computation cost of a chunk depends on the set of materialized views.

PeerOLAP allows replication of a chunk in many peers. Replication should be performed only if it is absolutely necessary, because it consumes space that could be used for other chunks. The above mentioned benefit function facilitates the replication of objects in a controlled manner. Let c be a highly aggregated chunk that is asked for the first time and is computed from the warehouse. Since both the computation and network cost are expected to be high but its size small, the benefit will be high. Assume that P caches c and Q requests c from P. Since the cost of retrieving and transferring c is now lower, the probability that Q caches the same result also decreases. If Q needs c in the future it can find it in P and its available cache space can be used for more beneficial chunks.

4.3.1 *Admission and Replacement Algorithm*

It should be obvious from the previous example that an incoming chunk is not cached by default, but only if it is beneficial enough for the peer. The admission and replacement algorithm, called *Least Benefit First* (LBF) is presented below. LBF is an LRU-like algorithm, which considers the benefits of the objects. It assigns a weight $W()$ to every cached chunk, which initially is equal to the chunk's benefit. $W()$ is decreased each time a new chunk is considered for admission, and is restored to its original value whenever the chunk is accessed again. When a new chunk c_{query} arrives, LBF sorts the cached chunks in ascending weight order and

marks as potential victims the first ones which, if evicted, will release enough space for c_{query}. In order to avoid accessing the entire cache index each time a new object arrives, we employ CLOCK [5]. Observe that the sorting step of line 8 requires at most $O(\log |CIndex|)$ time, where $|CIndex|$ is the number of objects in the cache, because the objects are previously sorted and in every step the position of only one object may change. The new chunk is stored only if its benefit is greater than the combined weight of the victims.

Algorithm 1 LeastBenefitFirst(c_{query})

1: /* c_{query} is the query chunk */
2: **if** c_{query} is already in the cache **then**
3: W(c_{query}) := B(c_{query}, P) /* reset W(c_{query}) to its initial value */
4: **else**
5: Let c_{CLOCK} be the chunk corresponding to the CLOCK position
6: W(c_{CLOCK}) := W(c_{CLOCK}) - B(c_{query}, P)
7: Advance CLOCK position
8: $CIndex$:= List of all cached chunks sorted in ascending W(c_i) order
9: $victims := \emptyset$
10: $next := 0$
11: **while** $FreeCacheSpace + \sum_{v_i \in victims} size(v_i) < size(c_{query})$ **do**
12: $victims := victims \bigcup CIndex_{next}$
13: $next++$
14: **end while**
15: $W_{victims} := \sum_{v_i \in victims} W(v_i)$ /* the total weight of all victims */
16: **if** $W_{victims} \leq B(c_{query})$ **then**
17: Evict *victims* from cache
18: Insert c_{query}
19: W(c_{query}) := B(c_{query}, P)
20: **end if**
21: **end if**

LBF resembles the GD [27] algorithm, which is used for caching web pages. GD, however, will always cache a new object even if it needs to evict more beneficial ones. In our case such behavior is contradictory with the controlled replication scheme that we aim at achieving.

The LBF algorithm controls the local cache of each peer. Next we will present three policies, which describe the behavior of the entire system and enforce progressively higher degree of collaboration.

4.3.2 *Isolated Caching Policy (ICP)*

The rational behind the *Isolated Caching Policy* is that a peer P is completely autonomous and will attempt to benefit from the other peers in a greedy manner. P publishes its cache contents and employs the algorithms that were described before, but it does not count the hits on its cache by the other peers. Therefore, if a neighbor Q requests a chunk c, which is in the cache of P, P will provide c but it will not update its weight back to the original value (line 3 of LBF). If c is not important for P it will eventually be evicted even if it is beneficial for the neighbor peers.

Although ICP disregards collaboration, it suits the philosophy of P2P systems. Recall that the peers do not necessarily belong to the same organization. Instead they may belong to autonomous users who would like to have complete control on the resources they provide.

4.3.3 *Hit Aware Caching Policy (HACP)*

In contrast to ICP, the *Hit Aware Caching Policy* considers the hits from other peers in an effort to ensure that the caches cooperate with the aim of minimizing the total query cost. In order to comprehend this consider again the benefit function of LBF: If P finds a chunk c in a peer Q, then $B(c, P)$ is lower than if c were answered by the warehouse; therefore, the probability that P caches c decreases. Intuitively, LBF implements a passive way of collaboration, based on an optimistic approach, since it assumes that c will still be in Q when it needs it again. In order for this to happen, HACP increases the benefit of c in Q, whenever c is used by another peer.

4.3.4 *Voluntary Caching*

The *Voluntary Caching Policy* attempts to exploit underutilized resources that may exist in some peers and at the same time avoid wasting any result that has been obtained from the warehouse. Assume two peers, P and Q where P exhibits a heavy workload and its cache is full with high-benefit chunks, while Q poses a few queries and its cache is under-utilized with low-benefit chunks. P asks for chunk c, which is found only at the warehouse. Although c has a substantial benefit, assume that P cannot admit it because the benefit of the potential victims is higher. Instead of discarding it, the voluntary caching policy will ask whether any neighbor of P can cache the result. If such a neighbor, say Q, exists, c will be forwarded to it. In case that multiple neighbors volunteer to cache c, P selects the one with the highest $B(c, Q) - B(victims, Q)$ value. Naturally, P has to pay the cost of transferring c to Q which is added to the total query cost. The intuition is that this cost will be amortized by subsequent requests for c. Note that due to the transferring cost, the benefit of caching c at Q becomes:

$$B(c, Q) = \frac{T(c, DW \rightarrow P) - T(c, P \rightarrow Q)}{size(c)} \tag{4}$$

where DW is the warehouse, P the requesting peer and Q the caching peer.

The voluntary caching policy may work either in conjunction with ICP (v-ICP) or with HACP (v-HACP).

4.4 Network Reorganization

The previous techniques attempt to minimize the total query cost by constructing efficient execution plans and caching query results. In this section we try to optimize the network structure, by creating virtual neighborhoods of peers with similar query patterns. The goal is to assign a set of neighbors to each peer P, so that there is a high probability for P to obtain missing chunks directly from them without having to search a large part of the network. These neighbors are the only ones that P can visit directly.

Ideally a peer should be able to communicate with all others by direct connections, in order to have complete knowledge about the contents of all caches. This is impractical for two reasons: (i) network connections consume resources at the peer and (ii) the entire network would be flooded with messages. Nevertheless, as we show experimentally good results can be obtained even with a limited set of *beneficial* neighbors. Note that the initial neighbors that a peer con-

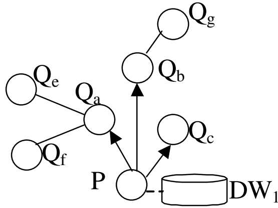

Figure 5: An example network structure

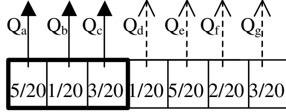

Figure 6: The LFU connection cache at peer P (numbers represent hit ratios)

nects to when entering the network, are nothing more than starting points and they are by no means optimal. Additionally, even if a good set of neighbors is known at the connection time, the query patterns may change or some of the neighbors may leave the network.

Motivated by the above, we formulate the problem as a special case of caching. Each peer has a number of available network resources, which are the equivalent of cache cells, and the objects that are cached are the direct connections to other peers. Each connection is assigned a benefit value and the most beneficial connections are selected to be the peer's neighbors.

Similar to the LBF policy, we follow an optimistic approach assuming that if a peer was contacted once it can be found again latter. From this assumption, and given that the cached objects cannot be further aggregated, it is clear that a hit to any peer is of equal benefit, regardless of the chunk that was retrieved. Recall that in any case the results are send back to the peer that initiated the query via a direct connection, which opens for this transfer. Therefore, in Figure 5 if Q_f provided a very beneficial object to P while Q_e provided a less beneficial one, each connection is charged with one hit. For these reasons, we use a simple LFU policy for caching network connections.

Since the number of allowed network connections nc_{max} is expected to be small, we can maintain accurate statistics for more than nc_{max} connections. For instance, in Figure 6 $nc_{max} = 3$ and the neighbors of P are $Q_{a,b,c}$ but we maintain a cache of 7 connections. The set of neighbors is not altered every time there is a change in the LFU cache in order to avoid frequent re-configurations of the network. Rather, the system waits until k requests are served (where k is a system parameter) and then selects as neighbors the nc_{max} more beneficial connections. In the previous example, if it is already time for reorganization, Q_b will be evicted and it will be replaced by Q_e.

Notice that the network connections considered here are virtual and differ from the physical network connections. Consequently the "neighbor" relation is asymmetric: if P has Q as neighbor, the opposite is not necessary true. In case that some relations are symmetric, the two directions can share the same physical connection, thus saving the cost of initializing new connections. Here we do not consider the minimization of this cost.

Table 2: The schema of the APB dataset. The values represent the size of the domain in each dimension at the corresponding level of hierarchy

	Product	Customer	Channel	Time
L_0	1	1	1	1
L_1	4	99	9	2
L_2	15	900	-	8
L_3	75	-	-	24
L_4	300	-	-	-
L_5	605	-	-	-
L_6	9000	-	-	-

Table 3: The schema of the SYNTH dataset

	D_1	D_2	D_3	D_4
L_0	1	1	1	1
L_1	25	25	5	10
L_2	50	50	25	50
L_3	100	-	50	-

5. Experimental Evaluation

We used two implementations to evaluate the performance of PeerOLAP. The first one is an actual prototype consisting of a data warehouse server in Hong Kong, and 10 peers in Singapore. The prototype was used to test the fundamental aspects of the architecture and to derive real-life parameters that were subsequently used by a simulator to evaluate the behavior of PeerOLAP in various situations. Table 1 illustrates this set of parameters, which will be used in this section.

We employed the dataset from the APB benchmark [19] in addition to a synthetic dataset (SYNTH) which was also used by [5] (see Tables 2 and 3). The total space of the entire cube was around 3.5G tuples for APB and 69M tuples for SYNTH. The total space was divided in chunks in a way that the chunk dimension range at any level was kept proportional to the number of distinct values at that level. The size of the largest chunk was 1M tuples.

The *Detailed Cost Saving Ratio* (DCSR) [15] was employed to measure the results. DCSR is defined as:

$$DCSR = \frac{\sum_i wcost(q_i) - \sum_i cost(q_i)}{\sum_i wcost(q_i)} \quad (5)$$

where $wcost(q_i)$ is the total cost of answering the query q_i in the worst case, and $cost(q_i)$ is the cost achieved by the system. For the worst-case scenario, we assumed that the peers do not have any cache[3], so all the queries must be answered by the warehouse (Figure 7d). Note that these costs include both $T(c, Q \rightarrow P)$ (i.e. the cost of calculating and transmitting a chunk) plus the overhead of the messages.

The tested configurations consisted of one data warehouse at a remote location (i.e., the transfer rate of the connection was TR_R) and a set of 1 to 100 local peers. The speed of all local connections was set to TR_L.

5.1 PeerOLAP vs. Client-Side-Cache Architecture

In the first set of experiments we compared PeerOLAP against a traditional client-server architecture with client-side-caching (C-S) (Figure 7b). First we considered the best

[3]Theoretically, the worst-case cost can be higher, due to messages. This is not significant for our results, since we are interested on the relative performance of different policies.

Table 1: Parameters derived from the prototype

Parameter	Value	Comments
TR_R	3.68891 KB/sec	Average transfer rate between remote peers (WAN)
TR_L	594.9347 KB/sec	Average transfer rate between local peers (LAN)
TR_D	4675.945 KB/sec	Average transfer rate from the disk
AMT_R	1.2975 sec/mes	Average time per message between remote peers (WAN)
AMT_L	0.3765 sec/mes	Average time per message between local peers (LAN)
ICT_R	3.68 sec/con	Average time to initiate a remote connection (WAN)
ICT_L	0.36 sec/con	Average time to initiate a local connection (LAN)

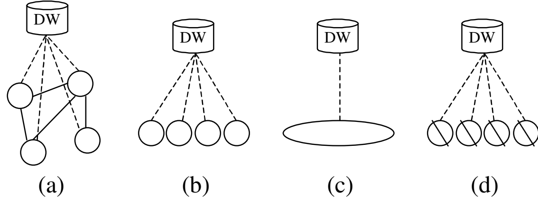

Figure 7: Configurations with one data warehouse. Dashed lines represent remote, and solid lines local connections, respectively: (a) PeerOLAP, (b) Client-Side-Cache, (c) One large cache, (d) Clients without cache

case for PeerOLAP, where all peers are connected to each other (i.e., clique network). We used 10 peers and we varied the cache size of each one from 0.001% to 10% of the total data cube size. The query set consisted of 20K queries following the 80-20 rule (i.e., 80% of the queries access a hot region representing 20% of the entire data cube). Each peer initiates the same number of queries. For fairness of comparison with C-S, PeerOLAP used its most naïve configuration: the optimizer employs the lazy policy (LQP) and the cache policy is ICP.

The results are shown in Figures 8 and 9. At the same figures we draw the results of a hypothetical 1-peer system (Figure 7c) having cache size equal to the sum of the caches of all peers. This configuration, called CentralCache represents the optimal case of the system. It is clear that in a clique configuration PeerOLAP achieves near-optimal performance. The cost difference from CentralCache is due to the replication of some objects, which is difficult to be completely avoided, and the cost of the messages. PeerOLAP easily outperforms C-S as expected. The results from both APB and SYNTH dataset are similar, although the absolute values differ. Since the trends were the same for all our experiments, in the following we only present the results from SYNTH.

Next we tested a more realistic configuration: each peer was connected to 4 others only, and the maximum hops allowed for searching was set to 3. The cache size of each peer was set to 1% of the total cube size, while all policies remained the same as before. The number of peers varied from 10 to 100. The query set was generated as follows: The peers were divided to groups of 10. For each group we provided a separate query set following a 90-10 distribution, while there was no intersection among the hot regions of different groups. Again 20K queries were generated and each peer initiated the same number of queries. The results are presented in Figure 10.

The performance of C-S is almost constant since, irre-

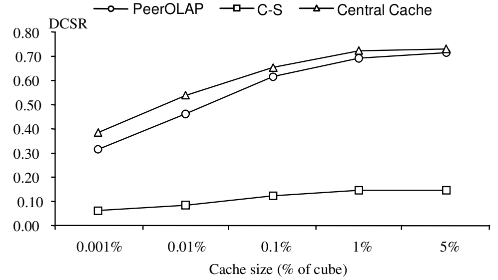

Figure 8: PeerOLAP vs. Client-Side-Cache system: (APB dataset)

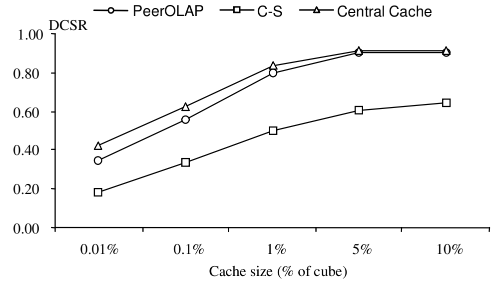

Figure 9: PeerOLAP vs. Client-Side-Cache system: (SYNTH dataset)

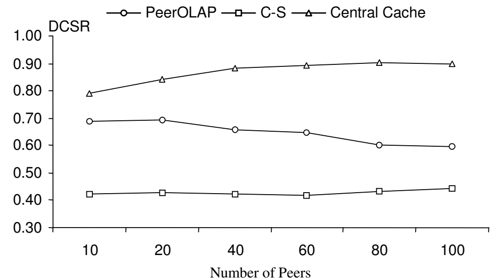

Figure 10: Groups of ten peers each, accessing the same hot region (4 neighbors per peer, 3 hops allowed)

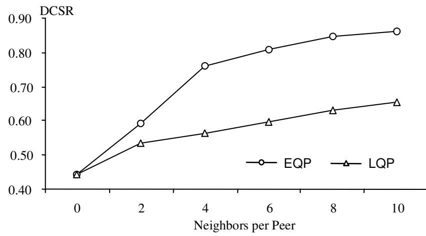

Figure 11: Query optimization for a network of 100 peers and 3 hops

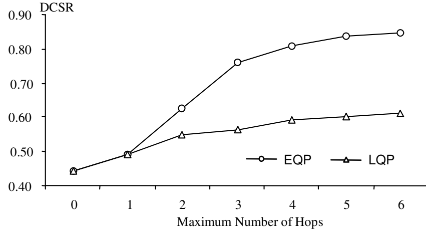

Figure 12: Query optimization for a network of 100 peers and 4 neighbors per peer

spectively of the number of peers, the size of an individual cache remains the same. CentralCache also improves when size of the total cache increases, as expected. The behavior of PeerOLAP is more complicated: for 10 peers, there is only one group, and the system attempts to exploit the contents of neighbor caches, as before. However, its performance now is not very close to the optimal, because the number of neighbors and the number of hops are limited. Although in the best case each peer can reach 12 others (i.e. number of neighbors times number of hops), the structure of the network may contain loops so the actual number of peers that are explored is lower. Due to the limited knowledge of the contents of other caches, the performance drops. This is more obvious when the number of peers increases. More peers with irrelevant data are inserted therefore it is more difficult for a peer to find others with similar workloads. Nevertheless, even when there are 100 peers in the network, PeerOLAP is still considerably better than C-S, partially because it can locate peers at the same group, and also because it takes advantage on similarities on the "cold" part of the workload.

Notice that the performance of PeerOLAP drops because we add peers with different workload. If more peers with similar workload are inserted, the performance typically increases, or remains the same at the worst case.

5.2 Evaluation of the Query Optimization Strategies

The next experiment evaluates the performance of the eager (EQP) and the lazy (LQP) query optimization strategies. We used a network of 100 peers, each equipped with cache space equal to 1% of data cube space. The caching policy was set to ICP and network reorganization was disabled. The query set consisted of 10 groups with 10 peers each, and every group was accessing a different hot region, as before. First we fixed the maximum number of hops to 3 and we varied the number of neighbors per peer. The results are shown in Figure 11.

Naturally, when there are 0 neighbors PeerOLAP is equivalent to C-S. When the number of neighbors increases, the knowledge of other peers' contents also improves leading to better performance. The performance gain is almost linear for LQP since the maximum number of peers it can search is also a linear function of the number of peers. EQP on the other hand, can explore up to $O(n^{hops})$ peers, where n is the number of neighbors. For example, when $n = 6$, EQP may potentially contact all the nodes (depending of course on the network structure) since $6^3 = 216$. Therefore the performance improves fast until it is almost equal to the optimal one. Similar results are shown in Figure 12, where we fix the number of neighbors to 4 and we vary the number of hops. Notice that when the number of hops is 1, the two policies are equivalent, since LQP always searches all its direct neighbors.

From these results, one might suggest that it is always preferable to follow the EQP strategy. However the performance metric we use here is based on the total execution cost and does not provide any information about the response time. EQP transmits a large amount of messages in the network. If all these messages need to be simultaneously processed, the response time will be affected considerably; such behavior contradicts the users' requirements.

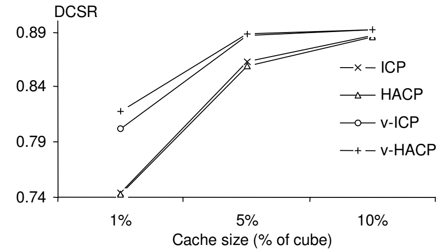

Figure 13: Comparison of the caching policies

5.3 Evaluation of the Caching Policies

In this set of experiments we evaluate the performance of the caching policies. We used a clique network consisting of 10 peers and we generated query sets consisting of 20K queries following a 90-10 distribution. In contrast with the previous experiments, here the number of queries each peer initiates is not the same for all peers. In each dataset Q_k, one of the peers receives k queries and the rest are divided equally to the remaining 9 peers. For instance, in the Q_{90} query set, 90% of the queries will be assigned to one peer, and the rest will receive 10/9=1.1% of the queries. In this way we want to simulate situations where some peers use heavily their resources, while others are underutilized.

Figure 13 compares the four combinations of caching policies for the Q_{90} query set and cache sizes varying from 1 to 10%. The experiments reveal that ICP and HACP have negligible performance differences. Moreover, there are cases where HACP performs slightly worse than ICP. This can be explained by the following example: assume that P fetches c_Q and Q fetches c_P from the warehouse and store them in their local cache with high benefit values. Then P and Q request c_P and c_Q respectively. c_P is not cached in P (neither c_Q in Q) since its benefit is low because it is already stored in the neighbor peer. At the same time, because of the HACP policy, c_P is forced to remain in Q (and c_Q in P). The result of this kind of "deadlock" is that both peers must pay the network cost of fetching the result from their neighbor,

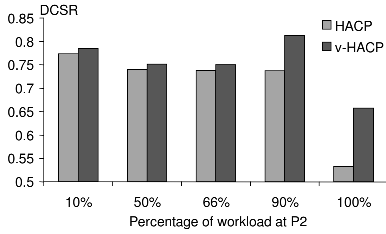

Figure 14: HACP vs. v-HACP for $Q_{10}, Q_{50}, \ldots, Q_{100}$ query sets

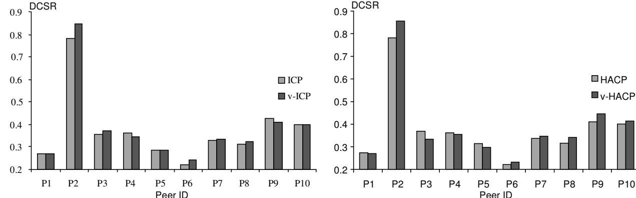

Figure 15: DCSR achieved by each individual peer for Q_{90}, and cache size 1%: (left) Isolated Caching Policy, (right) Hit Aware Caching Policy

in contrast with the ICP policy which would eventually enable each peer to cache the correct chunk. Assigning a lower weight to the remote accesses, compared with the local ones, only reduces but does not solve the problem.

Although HACP is not very beneficial itself, it combines well with the voluntary caching approach. In the Q_{90} dataset, there are 9 nodes, which are under-utilized. Voluntary caching allows some of the data from the heavy loaded peer to use the available resources of its neighbors. Therefore both v-ICP and v-HACP perform better than ICP and HACP. v-HACP is better than v-ICP because it allows the heavy-loaded peer to inform the others that the chunks it provided previously are still useful. Nevertheless, again the performance difference is not significant; the major performance gain comes from voluntary caching.

In Figure 14 we further investigate this issue: we compare HACP and v-HACP for workloads with different skew. We set the cache size to 1% and used query sets varying from Q_{10} to Q_{100} (i.e., all the queries are initiated by the same peer). v-HACP is better in all cases; however, when all peers are significantly loaded, the difference between the two policies is not large. Nevertheless, when some peers are under-utilized, v-HACP is clearly better. This is obvious in the extreme case, where all the queries are asked by the same peer, while the rest just share their caches. If voluntary caching is not used in this case, the caches of all 9 peers are always empty; this explains the substantial performance difference when v-HACP is employed. Note that the results among different query sets are not comparable.

Figure 15 presents the performance of each individual peer for the Q_{90} set where the cache size is set to 1%. Obviously P_2 is the peer which initiates the 90% of the queries. We have shown before that the overall performance of the system improves due to voluntary caching. Figure 15 reveals that in addition to the heavy-loaded peer, other peers may also benefit, but some may exhibit worse performance. This is true both for v-ICP and v-HACP, although the peers are affected in a different way.

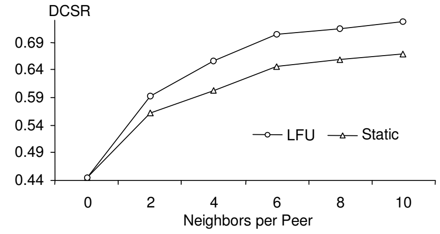

Figure 16: Effect of network reorganization

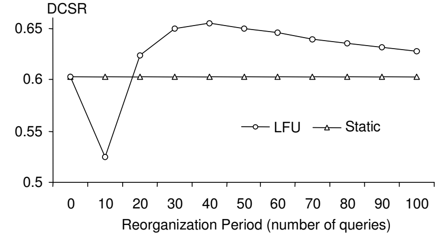

Figure 17: Frequency of network reorganization

5.4 Effect of Network Reorganization

In the last set of experiments we evaluated the adaptive behavior of PeerOLAP. We employed a network of 100 peers and we set the cache size of each to 1% of the data cube. The query optimization strategy was LQP and the caching policy was ICP. We used the same query set as in Section 5.2 (i.e. 10 groups with 10 peers each; every group accesses a different hot region). The maximum number of hops was set to 5. The period T_{reorg} that a peer reorganizes its neighbors was set to 40 (i.e., each time it has asked 40 queries).

In Figure 16 we vary the number of neighbors per peer and compare our adaptive strategy, versus a static network. As the number of neighbors increases, the performance of the static system improves, because of the better knowledge about the contents of other peers. By rearranging the neighbors of a peer P, there are two possible benefits: (i) the cost of searching for chunks decreases because some distant beneficial relevant nodes are becoming direct neighbors and (ii) with high probability, the neighbors of a beneficial peer are also beneficial to P; therefore, larger groups are constructed incrementally.

In Figure 17 we set the number of neighbors per peer to 4 and we vary the reorganization period T_{reorg} from 0 to 100. When $T_{reorg} = 0$, the network is static. When T_{reorg} becomes 10, the performance drops significantly. This is due to the fact that there was not enough time to gather accurate statistics; the initial network structure happened to be quite beneficial and the new structure is worse. However, if we allow the system to collect more information, the resulting network structure will be better and the performance increases. Observe that for values of T_{reorg} greater than 40, DCSR drops again slowly. The reason now is different: reorganization is performed so infrequently that cannot follow the changes of the workload. In the extreme case, if T_{reorg} approaches infinite (practically if it is larger than the number of queries), the network becomes identical to static again. Fine-tuning T_{reorg} is outside the scope of this paper.

6. Conclusions

In this paper we have presented PeerOLAP, a distributed

caching system for OLAP results. In a typical client-server architecture, isolated remote clients access data warehouses and maintain previous results in their local caches. By sharing the contents of the individual caches, PeerOLAP constructs a large virtual cache which can benefit all peers. The system is fully distributed and highly scalable as there is no centralized administration point and no central catalogue. The network does not have any specific structure and participation of the peers is unpredictable.

As shown in the experimental evaluation, PeerOLAP achieves significant performance gains with respect to traditional systems. This is accomplished by (i) query optimization techniques that determine which chunks should be requested form the warehouse, and which should be retrieved from the peers (ii) caching policies that enable co-operation among caches and eliminate unnecessary replication of objects (iii) re-configuration mechanisms that create virtual neighbors of peers with similar access patterns.

Currently PeerOLAP considers only cached chunks at the same level of aggregation as the query. For our future work we will investigate the option of computing a result at the peer side by further aggregating the cached data. The complication in such case is that every chunk can be computed by exponentially (to the number of dimensions) many plans. [4] explore this problem for a single cache and propose a method, which maintains additional information with each chunk in order to achieve good amortized performance for the query optimizer. In the distributed case, however, this method is inapplicable since it requires complete knowledge of the cached contents.

Another possible extension is the development of more sophisticated algorithms for the network reconfiguration. Identifying the neighborhoods of peers with similar access patterns is essentially a clustering problem, which however is difficult to solve because: (i) there is no complete knowledge about the whole network at any site; thus, each peer should decide using only partial information, and (ii) the available information constantly changes as the caches get updated, or the peers enter/leave the network.

7. Acknowledgments

W.S. Ng, B.C. Ooi and K.L. Tan are partially supported A*STAR and NUS under grant R-252-000-015-112/303. P. Kalnis was supported by grants HKUST 6081/01E and HKUST 6070/00E from Hong Kong RGC.

References

[1] J. Albrecht and W. Lehner. On-line analytical processing in distributed data warehouses. In *IDEAS*, pages 78–85, 1998.

[2] P. Cao, J. Zhang, and P. B. Beach. Active cache: Caching dynamic contents on the web. In *Middleware Conference*, 1998.

[3] S. Dar, M. J. Franklin, B. T. Jonsson, D. Srivastava, and M. Tan. Semantic data caching and replacement. In *VLDB*, pages 330–341, 1996.

[4] P. Deshpande and J. F. Naughton. Aggregate aware caching for multi-dimensional queries. In *EDBT*, pages 167–182, 2000.

[5] P. Deshpande, K. Ramasamy, A. Shukla, and J. F. Naughton. Caching multidimensional queries using chunks. In *SIGMOD*, pages 259–270, 1998.

[6] H. Garcia-Molina, W. J. Labio, J. L. Wiener, and Y. Zhuge. Distributed and parallel computing issues in data warehousing. In *ACM Symposium on Principles of Distributed Computing*, 1998.

[7] Gnutella. http://gnutella.wego.com.

[8] S. Gribble, A. Halevy, Z. Ives, M. Rodrig, and D. Suciu. What can databases do for peer-to-peer? In *WebDB Workshop*, 2001.

[9] V. Harinarayan, A. Rajaraman, and J. D. Ullman. Implementing data cubes efficiently. In *SIGMOD*, pages 205–216, 1996.

[10] R. Hull and G. Zhou. A framework for supporting data integration using the materialized and virtual approaches. In *SIGMOD*, pages 481–492, 1996.

[11] Icq. http://www.icq.com.

[12] P. Kalnis and D. Papadias. Proxy-server architectures for olap. In *SIGMOD*, pages 367–378, 2001.

[13] A. M. Keller and J. Basu. A predicate-based caching scheme for client-server database architectures. *VLDB Journal*, 5(1):35–47, 1996.

[14] D. Kossmann. The state of the art in distributed query processing. *ACM Computing Surveys*, 32(4):422–469, 2000.

[15] Y. Kotidis and N. Roussopoulos. Dynamat: A dynamic view management system for data warehouses. In *SIGMOD*, pages 371–382, 1999.

[16] T. Loukopoulos, P. Kalnis, I. Ahmad, and D. Papadias. Active caching of on-line-analytical-processing queries in www proxies. In *International Conference On Parallel Processing*, pages 419–426, 2001.

[17] Napster. http://www.napster.com.

[18] W. S. Ng, B. C. Ooi, and K. L. Tan. Bestpeer: A self configurable peer-to-peer system (poster). In *ICDE*, 2002.

[19] Olap council apb-1 olap benchmark r-ii. http://www.olapcouncil.org.

[20] M. T. Ozsu and P. Valduriez. *Principles of Distributed Database Systems*. Prentice Hall, 1999.

[21] P. Scheuermann, J. Shim, and R. Vingralek. Watchman : A data warehouse intelligent cache manager. In *VLDB*, pages 51–62, 1996.

[22] Seti@home. http://setiathome.ssl.berkely.edu.

[23] A. Shukla, P. Deshpande, and J. F. Naughton. Materialized view selection for multidimensional datasets. In *VLDB*, pages 488–499, 1998.

[24] A. Shukla, P. Deshpande, and J. F. Naughton. Materialized view selection for multi-cube data models. In *EDBT*, pages 269–284, 2000.

[25] M. Stonebraker, P. M. Aoki, W. Litwin, A. Pfeffer, A. Sah, J. Sidell, C. Staelin, and A. Yu. Mariposa: A wide-area distributed database system. *VLDB Journal*, 5(1):48–63, 1996.

[26] B. Yang and H. Garcia-Molina. Comparing hybrid peer-to-peer systems. In *VLDB*, pages 561–570, 2001.

[27] N. Young. On-line caching as cache size varies. In *Symposium on Discrete Algorithms*, 1991.

[28] Y. Zhao, P. Deshpande, and J. F. Naughton. An array-based algorithm for simultaneous multidimensional aggregates. In *SIGMOD*, pages 159–170, 1997.