# CubeCache: Efficient and Scalable Processing of OLAP Aggregation Queries in a Peer-to-Peer Network

Sangeetha Seshadri[†], Brian F.Cooper[†], Ling Liu[†]

[†]College of Computing
Georgia Institute of Technology
{sangeeta, cooperb, lingliu}@cc.gatech.edu

## Abstract

Peer to Peer (P2P) data sharing systems are emerging as a promising infrastructure for collaborative data sharing among multiple geographically distributed data centers within a large enterprise. This paper presents CubeCache, a peer-to-peer system for efficiently serving OLAP queries and data cube aggregations in a distributed data warehouse system. CubeCache combines multiple client caches into a single query processing and caching system. Compared to existing peer-to-peer systems the CubeCache solution has a number of unique features. First, we add a query processing layer to perform in-network data aggregation over peer caches. Second, we introduce the concept of *Query-Trails*: a cache listing recent data requestors. Query-Trails make it easier to find caches that are likely to have data needed for a query. Third, we design a benefit measure that incorporates the 'rarity' of a chunk into the notion of benefit, allowing controlled replication of chunks in a system plagued by frequent node departures or failures. We report the results of analysis and an experimental study using simulations and an implemented prototype that shows the CubeCache solution reduces the server load, improves query throughput and reduces query latency for OLAP tasks.

## 1. Introduction

Organizations frequently expand organically, adding geographically distributed branches and acquiring subsidiaries. As a result, a single centralized data warehouse may be too expensive or difficult to construct. Instead, the enterprise may temporarily or permanently decide to utilize a number of smaller, remotely located data warehouses. Decision support tasks then require answering queries by aggregating data over several of these smaller data-warehouses. In this scenario, since a single enterprise-wide warehouse may not exist, the only options are to perform the aggregations at the client-side or in the middle-tier such as the web-proxy. In particular, a large number of users located close to each other, at say the enterprise's corporate headquarters, may be issuing OLAP queries to the remote data centres. Answering these queries can require vast quantities of static and possibly overlapping warehouse data to be transported from a remote location.

Client-side caching [30, 10] is a technique that has been widely deployed to minimize the costs incurred during distributed OLAP operations. Usually, each node caches data independently. However, if instead these nodes were to collaborate and share cached data, we could build a system that was much more efficient than an individual client cache and far more scalable and fault tolerant than a large central cache. The primary problems faced in such a distributed cache are the following:

- *Data Location* - Searching and locating data in different caches with minimal network and processing overhead.
- *Cost-Aware Caching* - Designing a caching policy that minimizes both computational and network costs of a miss.
- *Scalability & Fault Tolerance* - Making the system scalable and tolerant to faults like the failure of nodes hosting a cache

In this paper, we propose CubeCache – a peer-to-peer OLAP caching system that combines multiple clients into a distributed data caching and processing engine. CubeCache minimizes the cost of processing queries and performs distributed aggregation computations, with little or no additional investment in terms of hardware and administration. The peer-to-peer nature of CubeCache introduces a high degree of parallelism that enhances the scalability and cost-effectiveness of our system.

CubeCache builds on the fundamental distributed caching techniques developed in PeerOLAP [3] by adding a layer to perform in-network query processing. In particular, CubeCache can perform aggregation queries in parallel at different peer caches. Moreover, we present techniques for efficient distributed location of cached data for use in aggregations, in order to minimize accesses to back-end data sources. We introduce a new technique – *Query Trails,* for efficient data location in the CubeCache network. Query-Trails improve object location by allowing peers to 'reach' up-to *twice* the depth of flooding with negligible additional costs.

Given that a CubeCache system is built out of a network of cooperating client caches, query processing in the system is vulnerable to client failures. In order to preserve the efficiency of query processing despite failures, we introduce a *weighted-benefit* caching policy designed to enhance fault tolerance. Our policy assigns higher weight to "rare" data, so that widely replicated data is ejected from peer caches before less replicated data. This improves the likelihood that every data item is cached multiple times, minimizing the impact of client failures. Experiments show that the performance of the CubeCache system degrades gracefully with node failures unlike a central cache.

We describe cache-based techniques for scalable processing of queries from multiple data centers. CubeCache peers use the chunk-based caching scheme [5, 4] to cache the results of queries and share their cache with the rest of the network. Unlike with central caching of data cube chunks, our system is scalable, has no central point of failure and performs well even in with different user access patterns.

### 1.1 Related Work and Background

Peer-to-peer systems have applications ranging from data sharing such as in Gnutella [7] and Freenet [8] to massive parallel computations in projects like SETI [9]. Although Data Warehousing [2, 1] and P2P systems have individually received a lot of attention, not much work has been done on *using* P2P systems in Data Warehousing. PeerOLAP[3] was the first and to our knowledge, the only system reported so far, that uses a P2P network to perform distributed caching of OLAP queries. Our system enriches the PeerOLAP system by a number of important new features. First, we provide support for aggregations in the distributed cache. We perform in-network aggregations, reducing the bandwidth consumption, distributing the workload and allowing our system to be useful even in situations, where pre-aggregated data is not available in the warehouse. Next, we improve the efficiency of data location through the use of query trails. And finally, we present a benefit metric that incorporates the impact of 'node failures', there by taking into consideration a very common feature of P2P systems into consideration.

Distributed OLAP systems often have to deal with semantic issues, such as reconciling schemas from different data sources. Our focus in this paper is on performance issues. The semantic issues arising from cross-enterprise queries in decision support systems have been investigated in [6] and their techniques can be applied to handle tasks such as data-modelling and integration.

Conceptually, cooperative Web-Cache systems which cache Internet objects among themselves [15, 23, 24, 25, 26, 27] are very similar to our system. However, query caching is quite different from object caching since apart from the data-location problem that is common to both, query caching has the additional problem of composing new results from partial results. Caching in web-proxies and the middle tier are discussed in [15, 16, 17].

Many P2P systems have been proposed for data-management, including P2P databases such as PIER[28]. While these systems implement the database in a peer-to-peer system, our approach uses a P2P system to cache and process OLAP queries at the client-end. P2P systems have also been used as storage solutions [8] in systems such as Piazza [29] that focuses on the dynamic placement problem.

Also related is Data Warehousing and OLAP query processing. Data Warehouses often store multidimensional data that can be viewed as a *data-cube* [1] and allow answering of OLAP aggregation queries. Several caching-based approaches have been implemented to accelerate the processing of OLAP queries [5, 10, 11, 12, 18]. Many approaches including pre-computation of aggregates and caching in the middle tier [10, 11, 12, 13, 15], have been suggested and implemented to achieve better response.

Our approach uses the chunk-based [5, 4] caching. Chunks [18] divide the data-cube into uniform semantic regions. The fine granularity of chunks allows better reuse and also saves the amount of cache space utilized since overlapping regions need to be stored only once.

In this paper we present CubeCache – a scalable and efficient P2P system for caching and processing OLAP queries. In the next section we outline our major contributions and provide the organization of the paper.

### 1.2 Contributions and Overview

Our main contributions include:

- A scalable technique to perform distributed in-place aggregation computations in client-side caches.
- The concept of '**Query-Trails**' which improves data location in the CubeCache peer-to-peer network.
- The **Weighted-Benefit** caching policy that takes into consideration the 'rarity' of a chunk while computing its cost for cache replacement decisions. This allows the system to be fault tolerant, minimizing the performance degradation due to node failures.

We present in Section 2, the motivation behind the CubeCache system and provide some real-world situations where the system can be deployed. An overview and running example of the CubeCache system is described in Section 3. Each of the components of the system and the techniques used to perform query processing and aggregation are discussed in detail in Section 4. Section 5 presents the details of the experiments conducted and an analysis of the results. We finally conclude in Section 6, providing our plans for future work and some insights gained while implementing the system.

## 2. Motivation and Applications

In this section we describe several scenarios in which our system can prove to be useful. In each case, the absence of a single, centralized warehouse means that data must be aggregated in a network of clients, usually workstations connected by a high-speed LAN. These clients may only be

able to access base data at distributed data sources over a slow Internet connection.

### 2.1 Single Enterprise Single Schema

Warehouses consolidate vast quantities of enterprise data and decision support systems run complex OLAP queries on this data. However, building a warehouse, consolidating and maintaining data, is both a time-consuming and costly process and the returns on investment are uncertain. Moreover, many small enterprises do not utilize all the features of these systems, and may want to implement a low-cost system that allows them to aggregate data from across smaller data-marts. Consider a retail chain with hundreds of stores all over the country. Each store records its daily transactions in a transactional database. Rather than build an enterprise data warehouse at a single go, the organization might prefer doing so in phases, or first implement a prototype. Using data marts, which are miniature data-warehouses, can make the process of implementing an enterprise-wide business intelligence system an incremental one there by spreading costs. For example, in the scenario described above, the retail chain may decide to implement its decision support system for one region at a time using small data-marts that conform to a single global schema. Users sitting at a remote location may issue queries like 'What is the total sales of a product over all the regions for the month of January?' to the system which involves gathering data from each region's warehouse and performing aggregations over it.

### 2.2 Single Enterprise Multiple Schema

This scenario occurs frequently when an enterprise grows by acquisitions where the organization may now have to deal with multiple warehouses implemented according to multiple schemas. Although the primary concern in these scenarios is semantic in nature, there is still a need to answer OLAP queries where pre-aggregated results may not be available and vast amounts of data will have to be transported from more than one location in order to answer queries.

### 2.3 Multiple Enterprise Multiple Schema

Corporations may collaborate with each other in order to increase their sales. Consider a restaurant chain with outlets all over the country collaborating with a popular soft-drink manufacturer and selling the drink in their outlets. With both organizations providing each other restricted views of sales data an integrated decision making system in the absence of a central warehouse is required, since neither organization may be willing to invest and maintain a warehouse for such data.

## 3. System Overview and Example

The CubeCache system consists of a set of peers (such as user workstations) that access remote data-sources and issue OLAP queries. There is no inherent structure to the network, and peers can join or leave at any time. Queries could vary from simple queries - targeted at a single data source and involving no aggregation to complex ones - queries that require aggregation of results across single/multiple data-sources. Peers that retrieve results may cache them for future use.

Let us now see an example of how the CubeCache system works. Consider an enterprise that has sales data for four regions – North, South, East and West, stored in four different data-marts maintained at the respective regional head offices. Users sitting in the corporate head office at a different location need to access this data. The user PCs are connected in a peer-to-peer network as shown in Figure 1. The solid lines indicate connections between peers and the dotted lines are connections between end-systems in this network and a remote warehouse. Note that all peers may not be connected to the warehouses.

Now, assume that the user at peer P1 issues a query for '*Total Sales*' which requires data to be aggregated over all regions. P1 searches the CubeCache network for the results of *'Total Sales'*. Although '*Total Sales'* results may not be found at any peer, partial results may be cached by several peers. For example, P2 has results for *South* and *West* regions and P10, for *North* and *East* regions. Peer P1 can examine the query and determine that these cached partial results, called *chunks* [4, 5], and are sufficient to answer the query. Peer P1 can ask P2 and P10 to compute total sales for the regions (chunks) they do have, or request the chunks directly and compute the aggregates itself. Then, P1 can compute '*Total Sales*' over all regions by combining these partial aggregates. Asking the peers to transfer just the aggregates, rather than the whole chunks, has 2 advantages – (1) it reduces bandwidth usage since, instead of entire chunks, only aggregated results are being transported and (2) P2 and P10 perform aggregation computation in parallel, making processing more efficient. Peer P1 may then cache the results of '*Total Sales'* for use by other peers in the future.

Implementing this system efficiently requires dealing with several challenges. First, if no peer has the full or partial results, then the querying peer must go directly to the back-end data mart, potentially incurring a long latency. Thus, we would like to improve the likelihood that chunks are cached somewhere in the peer-to-peer network. To address this issue, we introduce a cache replacement policy based on the *benefit* of a chunk, where benefit is determined by the relative cost of storing the chunk versus accessing the back-end. This caching policy is described in Section 4.4.

Second, there must be an efficient way to determine if the chunk is available in the network. We could keep a directory of chunks, but this directory would be hard to maintain as we expect caches to add or remove chunks frequently. We could contact every peer and ask if they have the required chunks, but this "broadcast" approach is too expensive in a large system. Instead, we combine

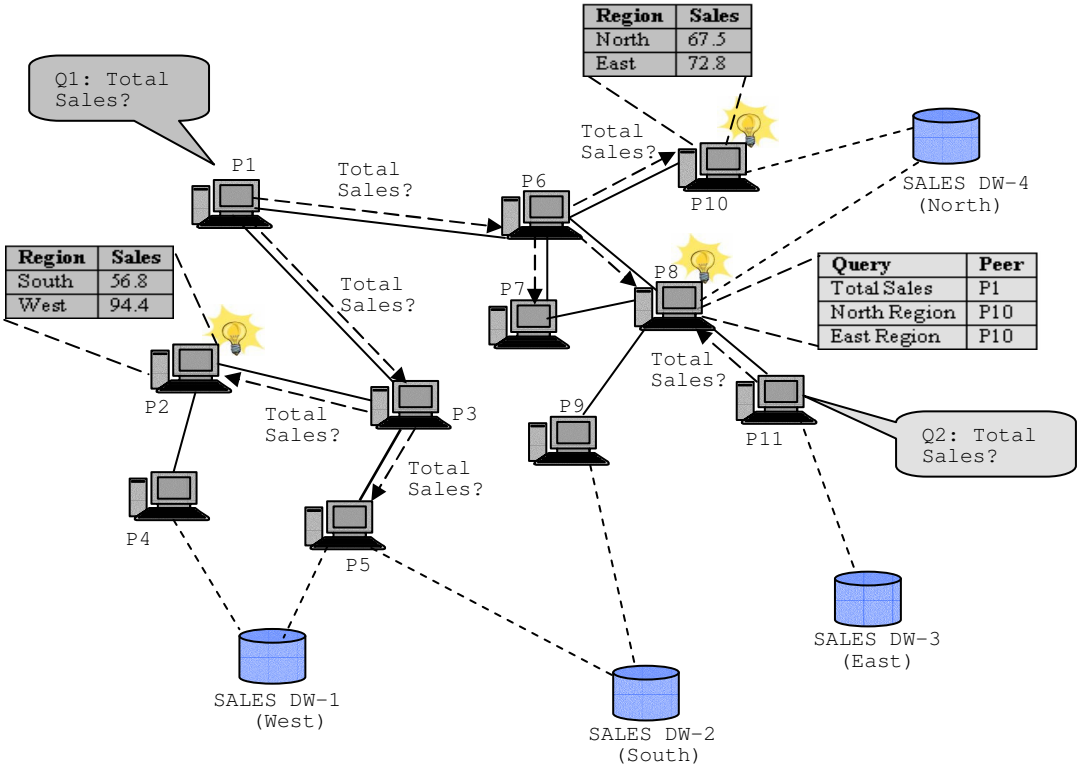


Figure 1. The CubeCache System

limited broadcast with a special lookup mechanism, called *query-trails*. Query trails are described in Section 4.2.

## 4. System Components

The architecture of a CubeCache peer is shown in Figure 2. Each peer accepts SQL queries from a local client, parses the queries and retrieves results. It caches results in a local cache and shares this cache with other nodes in the network. A peer also processes search requests from other peers in the network and returns responses. The **Request/Response Interface** receives and begins processing requests from the local client and other peers. First, the query is passed to the SQL parser, which parses the query and returns a list of chunks that are required to answer the query. The selection predicates can either be on the 'group-by' or 'non group-by' attributes. Since selections on non group-by attributes are already factored in before the aggregations are performed, the selection predicates on non-group by attributes of a cached chunk and an incoming query should match exactly for the cached chunk to be 'useful' in answering the query [5].

Next, the list of chunks are passed to the **Search and Forwarding logic (SFL)** module, which invokes the Cache Control module to search the local caches (the Chunk Cache and the Query-Trail Cache) for the chunk. If the chunk is not found, depending upon the forwarding policy (described in Section 4.3) the SFL module returns a list of peers to which the request should be forwarded. After the query results have been computed, the Request/Response interface constructs and returns a response message containing the results.

The **Cache Control** maintains the *Chunk Cache*, the *Query-Trail Cache* and the *Connection Cache*. The Chunk and Query-Trail caches are described in the next two sections. We use the Connection-Cache technique described in [3] to re-organize the peers into beneficial neighbourhoods. Each peer records in its connection-cache the performance (in terms of number of cache hits) of other known peers and periodically reviews its neighbour list. The peer then adds or drops neighbour links in order to maintain an optimal set of neighbors who provide more useful results.

The **Connection Repository** maintains the connection details for the neighbours in the network and warehouses. At regular intervals, the **Topology Control** module reorganizes the set of neighbours, adding new links and dropping existing ones depending upon the performance of peers as recorded by the connection-cache [3].

Apart from these components, we assume that every peer either maintains locally or has knowledge of schema details like the levels, dimensions and hierarchies. Since this information is quite small, it could either be maintained locally or at a single small local database to which a fast connection is available.

We also assume that every peer returns responses directly to the requesting node since anonymity is not a concern in our system. Also, since updates to the

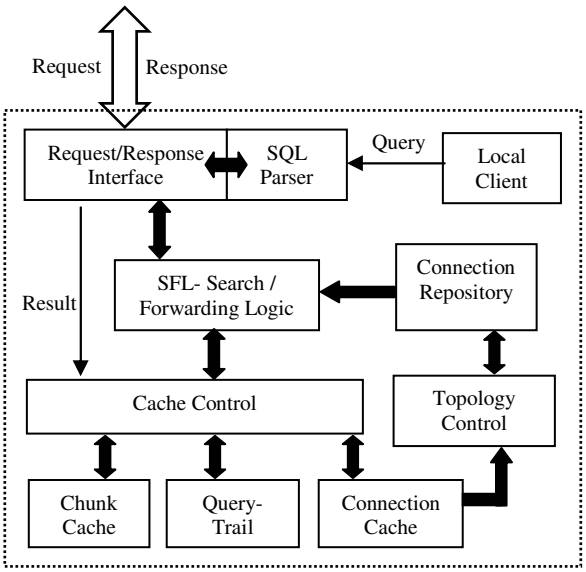

Figure 2. CubeCache Peer Architecture

warehouse data are infrequent, we assume that when an update occurs or when incremental loads are performed, the warehouse will broadcast a message to the system, instructing the peers to clear their caches.

### 4.1 Chunk Cache

Each peer caches query results in the form of chunks [4,5] and shares the cache with other peers in the network. Chunks divide the data cube into uniform semantic regions and allow finer grain caching. Chunks allow better utilization of the cache space, since overlapping regions of the result sets of different queries need to be stored only once.

The chunk cache stores actual chunks along with the following information: a chunk id, dimension vector and benefit value. The **chunk id** is used to identify the chunk uniquely. This chunk id can be made unique across multiple data cubes by concatenating the namespace with the chunk id. The **dimension vector** can be viewed as a bit map representation of a chunk's dimension. The **benefit value** is used by the caching policy to determine admission and replacement and represents the cost of computing this chunk. We describe the computation of this benefit value in Section 4.5.1

When a peer receives a query, it determines the needed chunks. Some of these chunks may be available locally. Chunks that are not locally available must be retrieved from other peers or from the data mart back-end.

### 4.2 Query Trail Cache

If a peer does not have the right chunks, it must contact other peers to ask for that chunk. We could broadcast the request to all of the other peers. However, broadcast is expensive, and thus it is desirable to proactively seek out peers that are likely to have the right chunks.

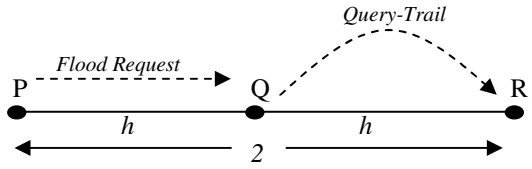

Figure 3. Reach of a peer P

We make the following observation: if a peer has recently requested its neighbours for a particular chunk, it is very likely to have that chunk in its cache. In order to leverage this information to improve our search, we introduce the concept of a Q*uery Trail Cache* (QT-Cache), which caches recent requests received by the node. Each peer uses its query-trail cache to find peers that have recently requested for a chunk and forwards the current request to that node. Query-Trail requests are sent directly to the node and are not forwarded further.

Consider again the example from Figure 1. Assume that peer P1 caches the results of the 'Total Sales' query. A short while later, say, peer P11 issues a similar query for the total sales. P11 forwards this query to peer P8. By maintaining a list of previous requests in its Query-Trail cache, P8 is able to immediately re-direct P11's 'total sales' query to P1. Note that in the absence of this information, P11's query would never have reached peer P1 since P1 is beyond 2 hops from P11.

In this section, we explain the formal model of a *Query-Trail Cache*. We then describe two forwarding techniques based on query trails – *Maximum Effort Query Forwarding (MEQF)* and *Least Effort Query Forwarding (LEQF)*. But first, we discuss the advantage of using Query-Trails that makes it an attractive technique to us.

#### *4.2.1 Advantages of Query Trails – Improving Reach*

We limit the scope of broadcast by specifying a hopcount, *h*: each request for a chunk is forwarded to all peers within *h* hops of the requesting peer. However, by maintaining query trails, even with a hop count of *h*, we are able to achieve a 'reach' of up to 2*h*, *without flooding the network* [Figure 3]. This is because every peer within *h* hops of the requesting peer has seen requests from other peers up to an additional h hops away. Q*uery-trails* allow us to make intelligent forwarding decisions thereby increasing the chances of locating the desired chunk. Query–trails are generic and can be applied to any peer-to-peer system that supports an object location and where the query stream exhibits temporal locality.

#### *4.2.2 Terms and Definitions*

A query result is characterized by a set of chunks. Thus, for a query q, $Result\ (q) = \{c_1, c_2, \ldots, c_n\}$.

**Query Trail**: A *query trail* (*qt*) is a tuple of the form $\langle chunk\_id, requestor \rangle$ where,

*chunk_id* is the identifier of a data item, the request for which arrives at the node, and

*requestor* is a node belonging to the system that had originally requested for the chunk. (Note: The node identifiers may be IP addresses of the nodes or unique ids assigned to the nodes upon joining the system.)

**Query Trail Cache (QT-Cache)**: A Query Trail Cache (QT- Cache) of node P is a set of *qt* entries stored at node P that may be refreshed based on the caching policy. In CubeCache we use a simple LRU policy for query-trail cache admission and replacement.

$$QT-Cache(P) = \{qt_1, qt_2, \ldots, qt_n\}$$

where, each $qt_i$ is a *query trail* entry.

For a chunk $c_i$, QT(P, $c_i$) is the set of all nodes that have requested for the chunk $c_i$ at node P and is defined as

$$QT(P,c_i) = \{ n_i \mid (c_i, n_i) \in QT\text{-}Cache(P)\}$$

Thus

$$QT(P,q) = \bigcup_{c_i \in Result(q)} QT(P, c_i)$$

For instance, if nodes A, B, C, D and E have requested at node S for chunks $\{c_1, c_2\}$, $\{c_1, c_2\}$, $\{c_1, c_5\}$, $\{c_2, c_5\}$ and $\{c_5\}$ respectively, node S has the following query trail entries: $< c_1$, {A,B,C}>, $< c_2$, {A,B,D}> and $< c_5$, {C, D, E}>. Now, if a request for query $q_1$ arrives at S, where

$Result(q_1) = \{c_1, c_2, c_3\}$,

then,

$QT(P, c_1) = \{A, B, C\}$,
$QT(P, c_2) = \{A, B, D\}$ *and*
$QT(P,q1) = \{A, B, C, D\}$

### 4.3 Chunk Searching Process

Our chunk searching process combines limited broadcast with search over query trails. Each peer that receives a request for chunks must determine whether to forward that query over neighbour links (broadcast), query-trails, or both. The simplest case, *flooding only*, is to forward the query just to neighbour links. In this section we discuss two other policies that use query-trails – *Maximum Effort Query Forwarding (MEQF)* and *Least Effort Query Forwarding (LEQF).* Our query forwarding policies are complementary to the Eager, Lazy and Optimal query processing policies described in [3] and can be used in conjunction with any of the policies.

#### *4.3.1 Maximum Effort Query Forwarding (MEQF):*

In this policy, we forward messages to both neighbors *and* query trail entries. Since messages are forwarded to more peers, and intelligently too, the possibility of a hit is very high. However, since the number of search messages are higher, *maximum effort query forwarding* is best suited for the systems where the cost of searching is much less compared to the cost of obtaining data from the warehouse. Assume that a user issues a query q at peer P. The MEQF policy works as follows:

i. P checks its local cache for chunks required to answer q. Let $C_{miss}$ be the set of chunks that are not available locally.
ii. P forwards a request for the $C_{miss}$ chunks to
    a. All of the peers in neighbors(P)
    b. All of the peers in QT(P, $C_{miss}$); that is, the query trail peers that have previously asked for any chunk in $C_{miss}$
iii. Each of P's neighbors checks their local cache for the chunks in $C_{miss}$, and sends a message to P listing the chunks it has. Let $C_{miss}$' be the set of chunks in $C_{miss}$ that the neighbour does not have. Each neighbor repeats the forwarding process, sending a request for $C_{miss}$' to each of its own neighbors as well as any query trail peers that might have chunks in $C_{miss}$'.
iv. This process repeats for *h* hops, where *h* is specified by peer P. The *h* value is stored in the request message, and decremented at each hop.
v. Any peer that receives a request for chunks via a query-trail checks its local cache and returns any results to P. However, query-trail peers do not forward the request further.
vi. Note that each node that receives a request for chunks also updates its QT-Cache to add P and the associated query.
vii. P keeps receiving responses till a timer t expires, after which it assumes that no more results are expected. P then calculates the optimal peer to retrieve each chunk from, contacts that peer and retrieves the chunk from it. In case the chunk is not found at any peer in the network, P will retrieve the chunk from the warehouse.

#### *4.3.2 Least Effort Query Forwarding (LEQF)*

This policy is very similar to the MEQF except that, at each node Q, if a query-trail entry is found, then the request is sent only to that node and is not flooded to all neighbors. However, if no such query-trail is found, then the query is forwarded to all neighbors. This approach tries to minimize the number of search messages in the system and resorts to flooding only if query-trail that aids intelligent forwarding are not found. In systems where the cost of searching is high, the *LEQF* policy tries to minimize search messages. Our experiments show that this policy reduces search messages by nearly 10% while providing better performance than naïve flooding.

#### *4.3.3 Other forwarding approaches*

The MEQF policy forwards search messages to maximum number of peers and LEQF to the least number of peers. Other query forwarding policies are possible. A variant of the above two policies could be one in which we forward messages to the 'top-k' most promising nodes (neighbours or query trail peers), where *k* could be a tuneable parameter. This would be a good policy in a network where the degree of each node is very high and hence flooding results in exponential increases in the number of messages at every step. Yang and Garcia-Molina [31] demonstrate that such a directed forwarding

policy can improve performance in very large and densely-connected networks. However, further study is required to see if this policy would be equally effective for a smaller network of OLAP caches. In ongoing work, we are examining these and other policies.

### 4.4 Aggregate Computation Mechanism

In this section, we describe our in-network aggregate computation technique. This technique relies on the closure property of chunks – that is, higher level chunks can be composed by aggregating a fixed set of lower level chunks. When a peer requests for a chunk, although it might not be readily available, lower level chunks that will yield the required chunk upon aggregation may be available at distributed locations. We propose an *in-place*, *in-network* aggregate computation mechanism by which a peer computes a higher level chunk by intelligently combining partial aggregates available from other peers.

#### *4.4.1. In-Network Aggregate Computation Technique*

When a peer requires a particular chunk, it initiates a search in order to locate either the chunk or all the lower level chunks. If the required chunk is readily available in the network, the peer retrieves it. Otherwise, the peer explores the possibility of computing the chunk from the lower level chunks available at other peers. Since all the lower level chunks may not be available at a single peer, partial aggregates (computed from available cache-resident lower-level chunks) have to be retrieved from distributed locations, to compute the final aggregate. The primary challenge in this is to generate a cost-effective *query execution plan* that minimizes the cost of *in-network* aggregate computation.

In order to ensure the correctness, we must ensure that each lower level chunk appears *exactly* once in the final aggregate. Since this is an instance of the well known set-partitioning problem which is NP-hard [31], we employ a greedy heuristic to determine the nodes which should compute the partial aggregates. Computation of a higher level chunk can be done by aggregating lower level chunks along any one of several dimensions. Dimensions along which *all* the lower level chunks are available are selected as candidate dimensions as this requires no data to be fetched from the warehouse. Along a candidate dimension, the set of lower level chunks can be partitioned in more than one way. We define *C*, the set of lower level chunks as follows:

$$C = \{c_i^{lower} \mid \overset{n}{\underset{i=1}{Agg}}(c_i^{lower}) = N\}$$

where, *N* is the required aggregate result. We need to divide *C* into a set of partitions *P* such that the following criteria are satisfied:

$$P = \{PR_i \mid \bigcap_{i=1}^{k} PR_i = \varnothing \ \wedge \bigcup_{i=1}^{k} PR_i = C\}$$

For a partition set *P* to be a solution, the chunks in each partition should be available in their entirety at a *single* peer. In order to compute *P*, we first arrange nodes in the descending order of number of chunks along the selected dimension. We then greedily construct subsets using the following algorithm:

**Algorithm: Partitioning Algorithm**

```
Input: C                    //set of all lower level chunks.
       Result[1…n]          //chunks available at a peer i

Begin:
fetch[1…n] := null;
pendingList := C;
while (PendingList != null ) do
 assigned := 0;
 max := 0;
 for each i in 1..n loop
   if( | Result[i] ∩ pendingList| > max )
    assigned := i;
    Max := | Result[i] ∩ pendingList|;
   end if
 end loop
  fetch[assigned] := Result[i] ∩ pendingList ;
  pendingList := pendingList – fetch[assigned];
end while
End;
```

Once the partition set along each candidate dimension has been computed, we then choose the set with the *least cost* as the query execution plan. Next, we describe a model that helps define this notion of cost.

#### *4.4.2 Cost Model*

The peer has to retrieve chunks that it does not have cached, either because the chunk was never cached locally, or because it was cached but then ejected from the cache. Let *size(c)* be the size of a requested chunk *c*. Let *C* denote the set of all chunks and *P* denote the set of all peers. Let $C_i = \{c_{i1}, c_{i2} \ldots, c_{in}\}$ denote the set of all lower level chunks of $c_i$ such that all $c_{ij}$ are along the same dimension and

$$c_i = \overset{n}{\underset{j=1}{Agg}}(c_{ij}) \text{ , where } Agg \text{ denotes aggregation.}$$

$P(C_i)$ the *Power Set* of $C_i$, denotes the set of all subsets of $C_i$. We define a function A which denotes the cost of calculating a chunk *c* at a peer S as follows:

$$A: \ C \times P \longrightarrow \mathbb{R}^+$$

$$A(c_i, S) = \begin{Bmatrix} rep \times const \times size(c_i), \ if \ c_i \ exists \\ \sum_{k=1}^{n} A(c_{ik}, S), \ if \ \bigcup_{k=1}^{n} c_{ik} \in P(C_i) \end{Bmatrix}$$

The function A is defined as a constant multiple of the size of the chunk if it exists and the cost of aggregating an available subset of lower level chunks otherwise. For example, if 10 of 30 lower level chunks of chunk $c_i$ ,but not chunk $c_i$ itself, are available at a peer, and the peer receives

a request for $c_i$, the peer will return to the requesting peer the details of the 10 chunks. The requesting peer may then ask this peer to return the aggregate of a subset of these 10 chunks. The 'rep' term denotes the 'reputation' of the peer and takes into consideration the quality of results provided by a peer and the level of trust associated with it. If operating in an un-trusted environment, a node can associate low costs with trusted peers and high costs with untrusted peers. In our experiments we assume that we are operating in an environment where security is not an issue and hence *rep*=1. The 'const' term denotes the disk read rate.

The cost (in terms of time) incurred in transferring a chunk available at peer S to peer R can be defined as

$$N(c, R, S) = \text{constant} + \frac{\text{size(c)}}{\text{Bandwidth(R,S)}}$$

where 'constant' can be taken to be the cost of setting up a connection between the two nodes *R* and *S*.

The cost of sending request messages from *R* to *S* depends on the number of hops between the two nodes. We define the cost incurred by sending messages from *R* to *S* as *size(msg)/delay x M(R → S)*, where *M(R → S)* is the cost of sending the message from R to S over one or more hops. The function *M* is defined as:

$$M(R \rightarrow S) = \begin{cases} h_{\max} - h_{recd}, & \text{if } h_{recd} \geq 0 \\ -(h_{recd}), & \text{if } h_{recd} < 0 \end{cases}$$

where $h_{recd}$ is the hopcount of the system, *size(msg)* is a constant representing the size of a message, and $h_{max}$ indicates the maximum hop-count for the system. In our system, request messages received through a query-trail lookup are tagged with a negative hop-count in order to indicate that these messages should not be forwarded, so if $h_{recd}<0$, we calculate $M(R \rightarrow S)$ as $-h_{recd}$.

We define the *Total Cost* (TC) of chunk *c* obtained from a peer S by a peer *P*, *TC(c, S → R)* as

$$w_1 (delay \times (M(R \rightarrow S)) + w_2 (A(c, S)) + w_3 (N(c, S \rightarrow R))$$

where, the weights $w_1$, $w_2$ and $w_3$ are empirical parameters representing the relative costs of sending a chunk, sending a request message, and performing an aggregation. For example, if the cost of searching is high due to presence of high delays along the links between peer nodes, $w_1$ will be high. We can interpret the cost TC as the time required to obtain a chunk. Thus, in order to minimize response time, chunks with high costs need to be cached locally. When peer P issues a request for a particular chunk, it may receive responses from multiple peers. P then chooses to obtain the chunk from the peer which provides it with the lowest cost. The cost of answering a query would be a summation over the cost for each chunk required to answer the query.

### 4.5 Caching Policy

#### *4.5.1 Caching Policy- Weighted Benefit*

Our system uses a cache admission and replacement policy that is a variant of LRU-CLOCK and takes the 'Benefit' of a chunk into consideration. Unlike previous caching schemes that take a notion of "benefit" into account [5, 3, 18, 19], we consider the 'rarity' of a chunk besides its size and level of aggregation, Since nodes in our peer-to-peer cache might leave or fail at any time, we assign greater benefit to a chunk which is rare, than a chunk that is available at many nodes while considering it for caching.

We define the benefit of caching a chunk as follows: Assume that peer P issues a request for chunk *ch* and receives responses from *n* peers $P_1, P_2, \ldots, P_n$. The chunk is available at the back-end data source $P_w$ by default. Let the cost of retrieving the chunks from $P_1, P_2, \ldots, P_n, P_w$ be $C_1, C_2, \ldots C_n, C_w$ respectively, where $C_i \geq C_j$ *when* $i > j$. Also, we assume that $C_w \gg C_i \forall i \in [1, n]$. Each $C_i$ is calculated as *TC(ch, Pi → P)* [defined in section 4.4.1]. (We assume that the warehouse will never be down since our system is not designed to deal with queries in the case of warehouse failures.) The benefit of caching a chunk is the cost that will be incurred by not caching the chunk. If each peer may go down with a probability $p$, we define Weighted Benefit W*B(ch)* as follows:

$$WB(ch) = \frac{\sum_{j=1}^{n} p^{(j-1)}(1-p)C_j + p^n C_w}{size(ch)}$$

The formula calculates the benefit of a chunk depending upon its cost, size and degree of availability. When all n peers that possess the chunk are up, the cost of retrieving the chunk is equal to the cost of transferring the chunk from the peer offering the minimum cost. If this peer is down, then the cost incurred will equal the minimum cost offered by the remaining peers. If all the n peers that have the chunk cached are not available, then, the cost will be the cost of retrieving the chunk from the warehouse. (When the chunk is not available at any of the local peers, n=0 and $B(ch) = C_w/size(ch)$ ).

Our benefit metric takes into account the 'rareness' of a chunk and chunks that are available at many other known peers in the network are given lesser priority compared to chunks that are available only at say, one other peer. Note that chunks that are obtained from the backend are initially "rare" and thus are automatically replicated at other peers in the network. The probability *p* is a tuneable parameter. In systems that are expected to be very stable, where peers remain connected to the network for longer intervals of time this parameter can be set to a very small value, whereas in systems with greater churn, the value can be set high to allow greater replication of chunks. Since the peer will anyhow have to wait for all responses or a time-out to occur before calculating the minimum available cost, using our benefit measure does not involve the maintenance of any additional information. We briefly explain the admission and replacement algorithms next.

#### *4.5.2 Replacement and Admission Algorithm*

Each cached chunk is assigned a weight. Initially, this weight is equal to the calculated benefit. Each time there is a cache miss, the weight of each cached chunk is multiplied by a decay factor DF, where 0 < DF < 1. Whenever a chunk is accessed, its weight is restored to the calculated benefit. Thus chunks which are accessed frequently continue to have high weight, while a chunk that is only accessed once will soon have little weight. The value of DF determines the rate at which weights decay. For example, DF might equal 0.9.

Whenever a peer receives a chunk that is not in its cache, it must decide whether to cache that chunk. To do this, we compare the benefit of the new chunk to the current weights of the chunks in the cache. If the new chunk has a lower benefit than all of the weights of the cached chunks, the new chunk is discarded. Otherwise, the chunks with the lowest weight are ejected from the cache to make room for the new chunk. Note that because chunks may be of different sizes, it may be necessary to eject multiple chunks from the cache. Note also that the benefit of a chunk obtained from the warehouse will be very high, and that chunk will almost certainly be cached.

To minimize the number of chunks that are replaced as the result of a cache miss, we use the IVLR technique described in [20]. Traditional caching algorithms continue marking victims until there is enough space to insert the incoming item. However, removing the last item in this list might result in more than the required amount of space being released. IVLR arranges victim chunks in descending order of their sizes and composes a new set of victims by iteratively choosing the largest sized chunk from the existing victim-set till the required space is achieved. This results in a new victim-set that is a subset of the LRU victim-set.

## 5. Experiments

We conducted experiments to evaluate our techniques using extensive simulations and an actual implementation of our system. Our aim was to measure the performance of the system with various query-forwarding techniques, its fault tolerance – i.e. how performance degrades with node failures, and finally compare the response-time and through-put of the system with a central-caching scheme. We used simulations because they allowed us to examine several different system parameters and forwarding techniques for a variety of scenarios. We also examined query throughput and response time using an implemented CubeCache prototype. In summary, our results show:

- The MEQF and LEQF policies improve query processing performance by 20 and 10 percent (respectively) over naïve flooding.
- A CubeCache of 32 peers connected in a standard internet topology offers 70% savings with a cache size of just 1% of the data-cube per peer, where as a central cache offers only 30% savings.
- Our CubeCache system is affected minimally by node failures.
- The response time of our CubeCache system is 23% better than that of a central cache.
- The throughput of our CubeCache system is much higher than the central cache: for a system of 8 peers, offers almost 8 times the throughput of the central cache.

### 5.1 Experimental Setup

Our experiments were conducted using the TPC-D [22] benchmark with the Scale Factor (SF) set to 1. The size of the database was approximately 1GB.

We generated a query stream of 10,000 queries following the 80-20 rule, where 80% of the queries access 20% of the data cube. We used a commercial database as our backend, although, our caching scheme is independent of the database used. Since the chunk based caching scheme allows reuse of results only if the non-group by predicates match exactly, and since the TPC-D benchmark query does not provide us with enough variety to represent the access patterns of a large number of users, we generated our own query stream. Our query stream was a mixture of 30% proximity queries, 30% roll-up queries, 30% drill down queries and 10% random queries.

The prototype CubeCache system is implemented in Java and is used to compare the response-time and throughput of our system to the Central-Caching scheme. For the simulation experiments, we used 32 CubeCache peers connected in a realistic internet topology that was generated using GT-ITM [21], which is a tool to generate topologies.

#### *5.1.1 Metric*

We use the standard metric Detailed Cost Savings Ratio-(DCSR), used in [22, 3] to measure savings. The DCSR is defined as the ratio of savings to the worst case cost.

$$DCSR(Q) = \sum_{ch_i \in Q} \frac{Cost(ch_i, warehouse) - TC(ch_i)}{Cost(ch_i, warehouse)}$$

where $TC(ch_i)$ [described in Section 4.3.1] includes the cost of searching the CubeCache network and transferring the chunk $ch_i$. The cost of transfer for a chunk that has not been found in the CubeCache network will equal the cost of transfer from the warehouse. Thus, a higher DCSR value indicates better system performance.

### 5.2 Effect of Query-Trails

In this experiment we study the effects of incorporating Query Trails in the CubeCache system. Each peer stored 20 recent requests in its query-trail cache. The number of requests to be maintained in the query-trail cache was chosen to maintain small trail-caches.

First we look at MEQF. Recall that MEQF forwards a request to all peers that requested for the same chunk recently (query-trail peers) as well as all neighbors. Figure 4 shows the cost savings as a function of the size of the

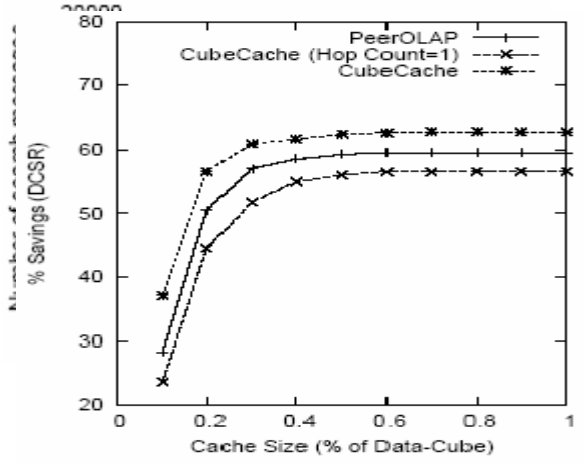


**Figure 4.** Maximum Effort Query Forwarding

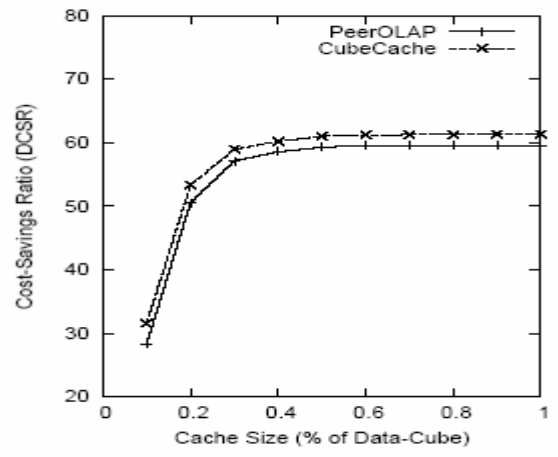


**Figure 5.** Least Effort Query Forwarding

cache at each peer. As the figure shows, MEQF provides 20% more savings on an average than flooding (i.e. PeerOLAP). This is because query-trails increase the probability of a hit by intelligently forwarding queries.

Next we looked at the LEQF policy. This policy forwards a request first based on the Query-Trail cache and then on all neighbour links if the query-trails are insufficient. As figure 5 shows, LEQF provides 10% more savings on an average than flooding. Since messages are forwarded intelligently, the probability of a hit is higher than in flooding. However, this policy contacts lesser number of peers than MEQF and hence offers lesser savings than MEQF.

We also examined the number of search messages required under each method. Figure 6 shows the results. As the figure shows, the MEQF method only requires 1% more search messages than flooding, even though it gave us a 20% improvement (i.e. Figure 4). Even though we are sending out messages on both neighbor links and query trails, each peer keeps only a small number of query trails, so the overhead is not large. LEQF actually requires 10% *fewer* messages than flooding, even though it produces 10% more cost savings (i.e. Figure 5). Fewer messages result because query trails result in hits 70% of the time, so it is often not necessary to flood at all.

### 5.3 Aggregations in CubeCache versus Central Cache.

In this section we compare the CubeCache system with in-network aggregation with a central caching scheme. (We did not compare with PeerOLAP in this case, since it does not support aggregations in the cache).

Figure 7 shows the cache size at which the central cache equals the performance of our CubeCache system. The central cache achieves equal performance only at cache sizes of beyond 15%. Note that the CubeCache system gives nearly 70% savings with a cache size of just 1% at each peer, whereas, the central cache provides only 30% savings. This is obviously due to the fact that each peer in the CubeCache system can access the local caches of peers reachable within a hop-count of 2. At very large cache sizes, our system's performance appears constant because most of the cache at each peer is unused. This is because each peer receives only a small fraction of the query stream where as all 10,000 queries are processed by the central cache.

### 5.4 CubeCache performance with node failures

Next, we examined the impact of node failures on our system. In this experiment we measure the hit-count (that is, the number of chunks successfully located in the cache) in the presence of node failures, compared to the PeerOLAP system. We simulated node failures to occur with a probability of 4% (a reasonable estimate for a P2P system with node departures). Figure 8 shows the comparison and CubeCache with its *weighted-benefit* caching policy clearly performs better. As the experiment progresses, and more nodes fail, we observe that the difference in performance is more pronounced, since CubeCache is more resilient to node failures due to its caching policy.

### 5.5 Experiments on the Actual Implementation

We conducted experiments to measure the response-time and throughput of the CubeCache system on a real implementation of the system. The experimental set-up consisted of eight CubeCache peers and two databases connected in a random network topology. We constructed the largest network possible given our installation. The experiment was conducted on Emulab[] using 10 nodes (Pentium4, 2800MHz, 512MB RAM running RedHat Linux 9.0). The inter-node delays between the peers set to ~2ms and that between a peer and a warehouse set to ~50ms. The peers are assumed to be part of an enterprise's LAN and the bandwidth available between peer nodes was set to ~100Mbps and that between peers and warehouses was set to ~28Kbps. Each database that functioned as a warehouse implemented the TPC-D dataset. Queries

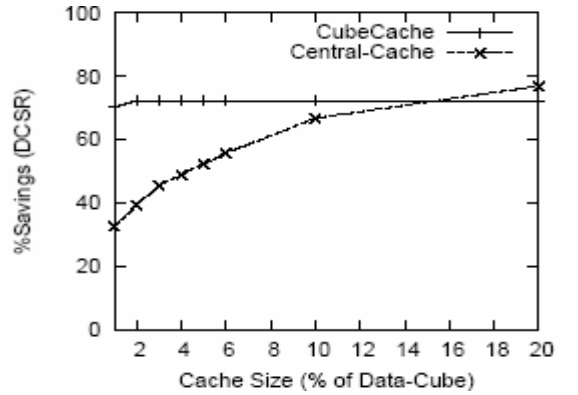


**Figure 7.** Comparison of CubeCache vs Central Cache

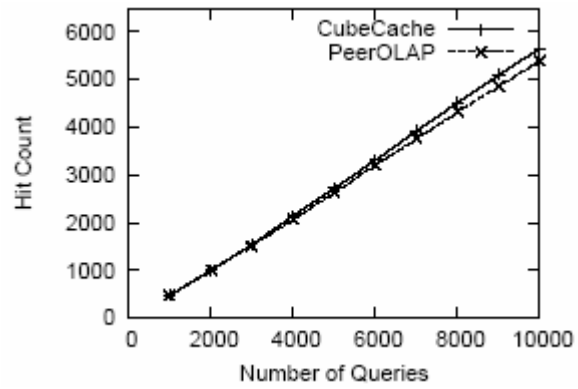


**Figure 8.** Effect of Weighted-Benefit Caching Policy

included those requiring data from a single warehouse to data to aggregation queries over data in both warehouses.

Figure 9 shows the throughput (in terms of Queries/min) of the CubeCache system and that of a central-cache. The cache size of the central-cache system was 10% of the data-cube, and each peer was assigned a cache size of 2% of the data cube. As the figure shows, the throughput of the CubeCache system is 8 times that of a central-cache. This is because each peer in the CubeCache system is able to process queries in parallel. On the other hand, in a central-cache system, all queries are processed at a single node. We also varied the number of clients and observed that increasing the number of clients increased throughput.

Figure 10 shows the response time of the CubeCache system with hop-count 1 and 2 and that of a central-cache. The response time in this case was measured as an average over ~3000 queries. As the graph shows, our CubeCache system with hop-count set to 2 offers nearly 23% better response time than the central cache. This is due to the fact that the central cache had to handle all 3000 requests and when requests arrived simultaneously, the processing workload on the central cache machine became very high, affecting performance. In the CubeCache system on the other hand, the queries were spread over 8 machines. We observed that the effective cache size ( that is, not counting duplicate chunks in different caches) was roughly the same as the central-cache. However, at a hop-count of 1 though, the CubeCache system performs slightly worse than the central cache, since the amount of cache reachable is lesser in this case. Overall we conclude that our CubeCache system offers comparable or better response-times and much better throughput than a central cache.

## 6. Conclusion and Future Work

In this paper we presented the CubeCache system – a P2P network that provides efficient and scalable processing of OLAP queries. Experiments performed on simulations and on a real implementation demonstrate that the CubeCache system performs better than existing flooding based P2P systems for OLAP query caching. Our system also outperforms a central cache in terms of both response-time and throughput. These improvements are a result of our primary contributions: an in-network data aggregation layer, the Query-Trails cache for efficient location of data chunks, and a smart caching algorithm to help preserve chunks despite failures.

We plan to further investigate data-location techniques, such as cache summary exchanging in order to improve data location in our system. Another direction of interest would be techniques to sync the query-trail cache with the data caching algorithm in order to minimize false positives in query trail hits. Finally, cooperative data placement techniques, in which objects evicted from one peer's cache may be accommodated at a neighboring peer's cache would make CubeCache cooperative in the true sense.

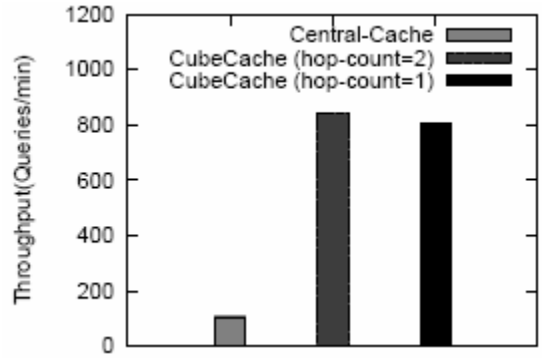


**Figure 9.** System Throughput

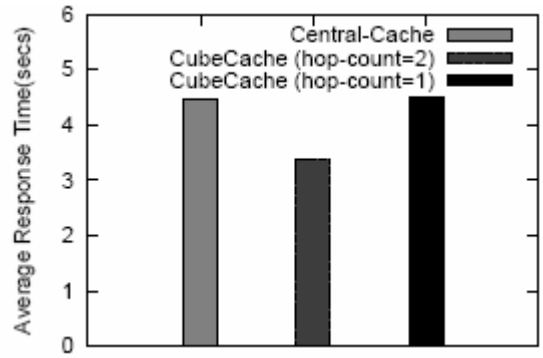


**Figure 10.** Response Time

## References


[1] J. Gray, S. Chaudhuri, A. Bosworth, A. Layman, D. Reichart, M. Venkatrao, F. Pellow and H. Pirahesh. Data cube: A relational aggregation operator generalizing group-by, cross-tab and subtotals. Data Mining and Knowledge Discovery, 1:29-54, 1997.

[2] S. Chaudhuri and U. Dayal. An overview of data warehousing and OLAP technology. SIGMOD Record, 26:65-74, 1997.

[3] P. Kalnis, W. S. Ng, B. C. Ooi, D. Papadias, and K. L. Tan. An adaptive peer-to-peer network for distributed caching of olap results. In *ACM SIGMOD*, pages 25–36, Madison ,Wisconsin, USA, 2002.

[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] Mauricio Minuto, Alejandro Vaisman. Aggregate Queries in Peer-to-Peer OLAP. DOLAP 2004, Washington DC.

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

[8] Open Source Community, "The Free Network Project – Rewiring the Internet", http://freenet.sourceforge.net

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

[10] P. Scheuermann, J. Shim, R. Vingralek, WATCHMAN: A Data Warehouse Intelligent Cache Manager, Proceedings of the VLDB, 1996.

[11] D.Srivastava, S.Dar, H.V.Jagadish and A.Levy. Answering Queries with Aggregation Using Views. VLDB, 1996

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

[13] R. Smith, C. Li, V. Castelli, and A. Jhingran. Dynamic Assembly of Views in Data Cubes. In PODS, pages 274--283, Seattle, Washington, June 1998.

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

[15] 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.

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

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

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

[19] Y. Zhao, P. Deshpande, and J. F. Naughton. An array-based algorithm for simultaneous multidimensional aggregates. In SIGMOD, pages 159{170, 1997P. Scheuermann, J. Shim, and R. Vingralek.

[20] Hosseini-Khayat.S "Replacement Algorithms for Object Caching", In ACM Symposium on Applied Computing, 1998

[21] E. Zegura, K. Calvert and S. Bhattacharjee. How to Model an Internetwork. Proceedings of IEEE Infocom '96, San Francisco, CA.

[22] Transaction Processing Performance Council, *TPC Benchmark D*, 1995

[23]SQURRIEL

[24] C. M. Bowman, P. B. Danzig, D. R. Hardy, U. Manber, and M. F. Schwartz. The Harvest information discovery and access system. Computer Networks and ISDN Systems, 28(1{2):119-125, Dec. 1995

[25] R. Tewari, M. Dahlin, H. Vin, and J. Kay. Beyond hierarchies: Design considerations for distributed caching on the internet. Technical Report CS98-04, Department of Computer Science, University of Texas at Austin, May 1998.

[26] J. Wang. A survey of web caching schemes for the internet. ACM Computer Communication Review, 29(5):36{46, Oct. 1999.

[27] D. Wessel. Squid internet object cache.http://squid.nlanr.net

[28] Ryan Huebsch, Joseph M. Hellerstein, Nick Lanham, Boon Thau Loo, Scott Shenker and Ion Stoica, Querying the Internet with PIER. VLDB 2003

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

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

[31] B.Yang and H.Garcia-Molina. Efficient Search in Peer-to-Peer Networks. In Proceedings of the 22nd IEEE International Conference on Distributed Computing Systems (ICDCS), July 2002