Site-Autonomous Distributed Semantic Caching

Norvald H. Ryeng, Jon Olav Hauglid, Kjetil Nørvåg
Department of Computer and Information Science
Norwegian University of Science and Technology, Trondheim, Norway
{ryeng,joh,noervaag}@idi.ntnu.no

ABSTRACT

Semantic caching augments cached data with a semantic description of the data. These semantic descriptions can be used to improve execution time for similar queries by retrieving some data from cache and issuing a remainder query for the rest. This is an improvement over traditional page caching, since caches are no longer limited to only base tables but are extended to contain intermediate results. In large-scale distributed database systems, using a central server with complete knowledge of the system will be a serious bottleneck and single point of failure. In this paper, we propose a distributed semantic caching method where sites make autonomous caching decisions based on locally available information, thereby reducing the need for centralized control. We implement the method in the DASCOSA-DB distributed database system prototype and use this implementation to do experiments that show the applicability and efficiency of our approach. Our evaluation shows that execution times for queries with similar subqueries are significantly reduced and that overhead caused by cache management is marginal.

Categories and Subject Descriptors

H.2.4 [**Database Management**]: Systems—*Query processing*

General Terms

Algorithms, Experimentation, Performance

Keywords

Distributed querying, semantic caching

1. INTRODUCTION

Large, distributed systems often use site autonomy as a way to reduce communication costs, allowing sites to make their own decisions and rely more on locally available information and less on information that must be fetched from their neighbors. In addition, if we can allow some of the housekeeping information to be slightly outdated without affecting the query results, further decoupling of sites is possible.

SAC'11 March 21–25, 2011, TaiChung, Taiwan.

Caching is one aspect of a query processing system that lends itself to autonomous decisions. Each site can cache the data it needs to speed up its own processing, without coordinating with other sites first. In distributed database systems, there is a choice of either shipping data to the sites where queries are processed or shipping queries to the sites where data are stored. Caching is possible and useful in both types of systems, but the nature of these systems provide different caching opportunities and call for different caching solutions. In this paper, we adapt the idea of semantic caching, taken from data shipping systems, and present a new method for semantic caching for a large query shipping system.

The main idea of semantic caching is to tag the cached data items with semantic information, typically predicates used in select queries. By looking at the tags, subsequent queries can identify cached items that can replace parts of the query. The data that is not in cache is fetched by a remainder query, and together the remainder query and the cached query provides the answer to the original query.

One of the challenges that are introduced when semantic caching is moved into a system of autonomous sites is that no single site has full knowledge of the query workload. When queries can enter the system from any site, and each site processes only a small part of each query before the result is shipped off to the next site, no single site has the complete picture of the query workload. This limits the metrics available for caching algorithms, but as we demonstrate, it is still possible to make globally sound caching decisions.

By building semantic caches of intermediate results on the sites where these results are produced, subsequent similar queries can benefit from retrieving some of their data from cache and issuing remainder queries to perform the rest of the operations. Our method builds a globally accessible, distributed cache based on autonomous sites. Whereas in traditional semantic caching each site has its own local cache that is not shared with other sites, our method gives sites access also to the cache entries elsewhere in the network.

With semantic caching comes the possibility of making caching decisions based on more than access statistics. Richer caching algorithms can be defined that inspect the semantics and decide to cache data that is not the most frequently used, but that will give a higher performance gain when used. Even if the join of two tables is a less frequent subquery than the tables themselves, more time may be saved if the result of the join is cached than if the tables are cached. By making sites autonomous, we also open up for the possibility of using different caching algorithms on different sites.

The contributions of this paper is as follows: We present a new method for semantic caching of intermediate results in a distributed database system, using autonomy to increase scalability. We demonstrate how this caching method reduces query execution time with

Symbol	Description
S	Site
T	Table T
T_i	Fragment i of table T
n	Algebra node
N	Algebra tree
N_n	Subtree rooted at n
c	Cache entry
C	List of cache entries
$\mathcal{C}$	One site's cache
ts	Timestamp
$query(c)$	Query representation of cache entry

Table 1: Symbols.

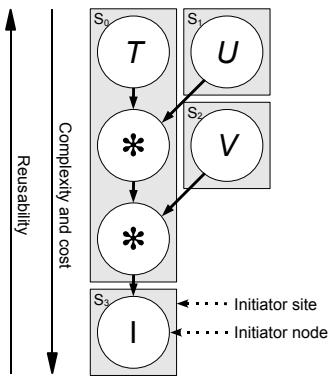

Figure 1: Example query from unmodified DASCOSA-DB.

almost no overhead. We also demonstrate that the more advanced cache replacement policies that are possible with a semantic cache give considerable improvements over traditional LRU. Experimental evaluation of the costs and benefits of our semantic caching method is done by implementing it in the DASCOSA-DB [9] distributed database prototype.

The rest of this paper is organized as follows. We start with a review of related work in Section 2. Section 3 describes the system setting. Our caching method is described in detail in Section 4. Section 5 describes our experiments and results, and we conclude the paper and outline future work in Section 6.

2. RELATED WORK

Semantic caching [7] and predicate-based caching [13] augment cached data with a semantic description of the data. The benefits of semantic caching include low overhead and reduced network traffic [11, 18]. Cache tables [1, 3, 16] are somewhat similar to semantic caching, but only caches tables, not intermediate or final results of queries. Semantic caching has also been applied to deductive databases [4] and web querying systems [6, 15]. Common to all these systems are that they are built for a single query entry point to the system.

There are several approaches to filling caches. Cache investment [14] aims to optimize for the future by deliberately executing suboptimal queries to generate cache entries that have a higher hit rate. Caching of time-consuming operations has also been studied for single values that are duplicated in a result [10]. Identification of candidates for caching and insertion as cache nodes at different levels in the algebra tree [8] is often done during query planning.

Cache entries may exist with similar, but not exactly the same, data to what has been requested. Such cache entries may be transformed to match the request [2]. This increases the cache hit rate for workloads with many similar queries. View materialization [5, 17] is a kind of explicit caching requiring manual intervention.

Our approach is similar to that of PeerOLAP [12] in that sites operate under a large degree of autonomy and make local caching decisions. However, while PeerOLAP uses broadcast messages to locate caches, our approach retrieves information on existing cache entries from a distributed catalog service. This way we avoid flooding the network on each query.

3. PRELIMINARIES

In this section we describe the system and query model used in the rest of this paper. The symbols used in the paper are found in Table 1. We have implemented our caching method in the DASCOSA-DB distributed database system prototype. Our caching method is general and does not put many limitations on the underlying system, but some details of the implementation are dependent on details of the underlying system.

3.1 System Model

The system consists of a number of sites, each site S_i being a single computer node or a parallel system acting as a single entity seen from other sites. All sites can store relational tables, and these may be horizontally fragmented. Fragment i of table T is denoted T_i.

The sites in the system are autonomous, and the only requirement is that they have a common protocol for execution of queries and metadata management. This means that some sites may choose not to cache, and those that do cache may choose different cache replacement policies. This makes it possible to include sites under different administrative domains, allowing interoperability while at the same time allowing each administrative domain to remain autonomous.

Metadata management, including information on where data is stored, is performed through a common catalog service. This catalog service is itself assumed to be fault tolerant. It can be realized in a number of ways. For example, in DASCOSA-DB, the catalog service is realized by a distributed hash table where all sites participate. The organization of the catalog is not important to our caching method, but our implementation relies on some implementation details of the catalog service.

3.2 Query Model

We assume queries are written in some language that can be transformed into relational algebra operators, for example SQL. These algebra operators constitute an algebra tree for the query, and each subtree of the query tree is a subquery. Throughout this paper we will use the terms subtree and subquery interchangeably.

Queries may arrive from any site of the system. The site that introduces a query to the system, called the initiator site for that query, becomes the coordinator for that query. When a query is entered at one site, this site becomes the initiator site for that query. The initiator site decomposes the query into an algebra tree, e.g., as the one shown in Figure 1. The example query accesses the three tables T, U and V located at sites, S_0, S_1 and S_2, respectively. Query processing is distributed between these three sites and the initiator site for the query, S_3. When the query planner has assigned each algebra node N_i to a site, S_i, the query is shipped to these sites using Algorithm 1.

When query processing starts, each node of the algebra tree pro-

Algorithm 1 Stepwise transmission of algebra tree.

At site S_i, after receiving N_i:

$n_i \leftarrow root(N_i)$
for all $n_c \in children(n_i)$ **do**
  $N_c \leftarrow subtree(n_c)$
  $S_c \leftarrow getAssignedSite(n_c)$
  $Send(N_c, S_c)$
end for

duces an intermediate result that is shipped to the parent (or *downstream*) node. Our distributed semantic caching method caches these intermediate results, such as the result of $T * U$, and reuses them in subsequent queries. This can save significant amounts of processing. The cumulative cost and complexity of the intermediate results increases towards the root, but reusability decreases. Caching the result of nodes close to the root of the tree means we save more work when we get a cache hit, but cache hits are more frequent for intermediate results closer to the leaves.

4. DISTRIBUTED SEMANTIC CACHING

In order to implement semantic caching, we modify the localization, dissemination and processing steps of DASCOSA-DB, and add a fourth: cache registration. These modifications and extensions are described in the following sections.

4.1 Query Localization

After query decomposition, the query is represented as a tree N of algebra operator nodes. This is given as input to the query localization step.

The initiator site has to do catalog lookups for all tables referenced by N. This is done by requesting from the catalog service a list of table fragments T_i and the sites S_{T_i} on which they are located. In DASCOSA-DB, the request for information about one table is handled by one site of the distributed catalog service. That site knows of all fragments of that table. We extend the information stored at the catalog site to also include an index of some, but not necessarily all (see Section 4.4), cache entries of intermediate results involving that table.

We also extend the catalog lookup request by piggybacking a representation of N onto the request messages, as shown in Figure 2. The catalog service site responds with its normal result of table fragments and their locations and adds an additional list $C = \{c_1, c_2, c_3, \ldots\}$ of cache entries it knows of. The extended reply message is shown in Figure 3. Each entry $c = \langle N_c, S_c, ts_c \rangle$ describes the cached query N_c, the site S_c that stores the cache entry and the timestamp ts_c of the entry. All entries in C are relevant cache entries, by which we mean entries that can be used to answer the query.

After receiving all lookup replies, we have accumulated information on all relevant cache entries for N. The optimizer may decide to rewrite the query to use some cache entries that do not exactly match a subtree of the current plan. Due to space constraints, we refer to [7] and [11] for more details on how queries are transformed to take advantage of semantic caches.

After caches have been found and the query plan has been adapted, the next step is to assign each node n of N to a site. Leaf nodes are table accesses and are assigned to the sites that store the corresponding table fragments. Normally, DASCOSA-DB assigns an operator node to the same site as one of its operands. We extend this algorithm to exploit cached data. By looking at the subquery N_n rooted at n and the list of cache entries, if $\exists\, c : N_c = N_n$, n is assigned to S_c. If there are more than one cache entry for N_n, the localizer chooses one. Nodes that are not found in cache are assigned to sites using the normal query localization algorithm.

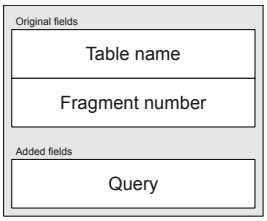

Figure 2: Extended lookup message.

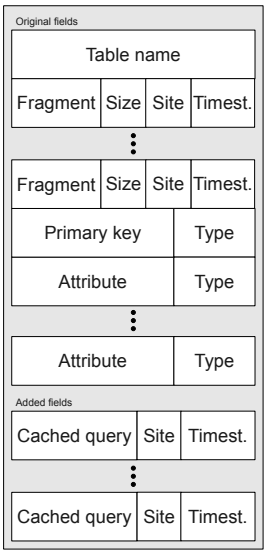

Figure 3: Extended lookup reply message.

The initiator site does not actually decide whether to use a cache entry. It only assigns algebra nodes to sites where the catalog service says there are matching cache entries. Sites are autonomous in caching decisions, so a site that was intended by the initiator to deliver data from cache may have replaced the cache entry when the query arrives, making it necessary to process the query in full.

When all nodes of the query have been assigned to a site, the initiator site starts shipping out the query to the rest of the sites participating in resolving it.

4.2 Query Dissemination

The query N is shipped stepwise to the participating sites, using a modified version of Algorithm 1. The initiator site assigns $root(N)$ to itself, and then for all $n \in children(root(N))$ sends out N_n to S_n, which again send out the subtrees of the nodes they receive, etc. This continues until the leaf nodes, i.e., table access nodes, are received by the sites that store the corresponding tables. The timestamps ts_{T_i} of all table fragments referenced by nodes in a subtree are piggybacked onto that subtree as it is sent out.

If no cache entries exist, the query shipping behaves exactly as in the unmodified Algorithm 1, but sites that have cached previous results must check their caches to see if the query matches any entries.

When a site receives a query, it checks the table fragment timestamps $ts^N_{T_i}$ from the query against entries $c \in \mathcal{C}$ in the local cache

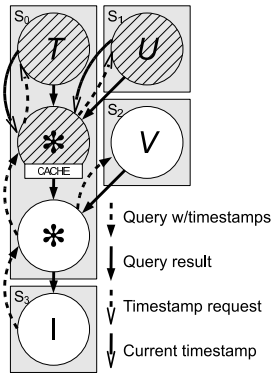

Figure 4: Query dissemination with table fragment timestamps.

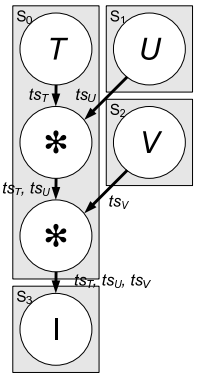

Figure 5: Result and timestamp propagation.

to see if any of its cache entries should be invalidated. If $\exists\, T_i, c : ts^N_{T_i} > ts^c_{T_i}$, c is outdated and should be removed. This is done for all cache entries that involve these table fragments, not only those relevant to the current query.

After removing outdated cache entries, the site checks if $\exists\, c \in \mathcal{C} : query(c) = N_n$. If such a cache entry is found, the site has to request current timestamps $ts^*_{T_i}$ of all the table fragments that are referenced by N_n. This is done to guarantee that the cache entry is up-to-date. The locations of the fragments are found by looking at the received query, which contains table scan operators assigned to the corresponding sites.

The cache entry timestamp consists of a set of table fragment timestamps $ts^c_{T_i}$ such that $\forall\, T_i : ts^N_{T_i} \leq ts^c_{T_i} \leq ts^*_{T_i}$. If $\forall\, T_i : ts^c_{T_i} = ts^*_{T_i}$, the site holds back the whole subtree rooted at the cached algebra node and stops query dissemination of that branch. The algebra node is replaced by a special node that delivers the result from cache. If $\exists\, T_i : ts^*_{T_i} > ts^c_{T_i}$, the cache entry is outdated and is removed before query dissemination continues as if the cache entry had never existed, assigning the root node to be processed locally and sending the subtrees rooted at the children of this node to the sites to which they have been assigned by the initiator site.

Figure 4 shows how the query is distributed to the participating sites. In the example, the query with timestamps is sent out upstream from the initiator site, split at each algebra node. A cache entry for $T * U$ is found on site S_0 and a special request is made to the sites storing relevant table fragments (in this example, the tables consist of only one fragment each) to retrieve the current timestamps. Table T is stored on site S_0, so the timestamp request is processed locally. Site S_1 receives the request from site S_0 and replies with the current timestamp of table U.

Our caching method supports variations of this timestamp policy. Different isolation levels may allow caches that are older than the current table fragment timestamps, leading to more cache hits. Timestamp policies should be the same on all sites, since the system as a whole cannot guarantee stronger isolation levels than the weakest timestamp policy allows.

Dissemination stops when all branches of the query tree have been terminated by a leaf node delivered to the site to which it has been assigned or by a deliver-from-cache operator. As soon as a leaf node is delivered, or a deliver-from-cache node created, the node enters the processing step and starts producing data.

4.3 Query Processing and Caching

In the dissemination step, it was discovered whether any relevant cache entries existed and were up-to-date, in which case dissemination of that subtree stopped and the root of the subtree was replaced by a special node serving data from cache. Cache entries are locked in cache as long as they serve an ongoing query.

The special deliver-from-cache operators simply deliver the cached result, including the cached timestamps, which by now are guaranteed to be up-to-date. Since the timestamps of the table fragments needed to produce the cached result are stored with the cache entry, they can easily be retrieved and sent with the result, providing timestamps for caching of downstream nodes.

If data is not served from cache, there might be an opportunity for caching. To allow downstream caches, table fragment timestamps are propagated along with the results of operators, as shown in Figure 5. At each interior node n in the query tree, the result of the operation is tagged with a combined timestamp $ts_n = \cup_{n' \in children(n)} ts_{n'}$ of the timestamps of all operands.

The result of algebra nodes are candidates for caching at the site where they are processed, and the timestamps that comes with the result are used to timestamp cache entries that are created. The decision to cache the result or not is made when a node starts executing, and is based on the cache replacement policy.

In general, the cache replacement policy defines an ordering of cache entries. In block based caching, where cache entries are always of the same size, only the first entry to be replaced has to be identified. When caching intermediate results of differing sizes, multiple cache entries may have to be removed to fit one larger entry in cache. The ordering must also include queries, so that candidates for caching can be compared against any existing cache entry. In Algorithm 2 we define the general cache replacement algorithm using such an ordering. Given a successor relation $\succ_p$ defined by the cache replacement policy and the cache $\mathcal{C}$, we use the ordered set $\mathcal{C}_p = (\mathcal{C}, \succ_p)$, where $head(\mathcal{C}_p)$ is the least element of $\mathcal{C}_p$.

Each site decides autonomously which results to cache and may apply different cache replacement policies and have different cache sizes. Since cache entries are compared against queries that have not yet been processed, some values must be estimated, e.g., the size of the intermediate result of N_n. The successor relation $\succ_p$ that defines the ordering of cache entries may rely on a number of metrics, either measured or estimated.

4.3.1 Available Metrics

Each site has only a restricted view of the system, knowing only the algebra nodes passing through it and the subtrees rooted at these. It also knows the contents of its own cache and the usage statistics of that cache. We divide the available metrics into three

Algorithm 2 Decide to cache the result of N_n (true) or not (false).

At site S, when deciding whether or not to cache the result of N_n:

```
free ← free cache space
C_replaced ← ∅
C_remaining ← C_p
while free < size(N_n) do
  c ← head(C_remaining)
  if N_n ≻_p c then
    free ← free + size(c)
    C_replaced ← C_replaced ∪ {c}
    C_remaining ← C_remaining \ {c}
  else
    return false
  end if
end while
C_p ← C_remaining
return true
```

measurement categories: size, cost and query pattern.

The size of the result is necessary to decide if there is room for the result in the cache. However, before the query has been processed and the size can actually be measured, we must rely on estimates based on the available statistics. Table statistics that are available from the global catalog and included in N can be used to find the size of table fragments and estimate the size of the results of downstream operators.

The cost of resolving a query can be estimated knowing operand size and operator implementation details. The cost of reproducing a result from scratch is the cumulative cost of the algebra subtree, so the estimated cost of reproducing the result can be found by adding up the estimated cost of all nodes in the subtree.

A simpler cost estimate is the node's position in the algebra tree. The leaf nodes are table access nodes. Intermediate nodes have a higher cost, and the cost of reproducing the result increases towards the root. Instead of computing the cumulative cost, we can simply use the height of the subtree rooted at that node.

The simplest query pattern metric is to use least recently used (LRU) ordering of cache entries, allowing recently used cache entries to stay in the cache while less recently used entries are replaced.

4.3.2 Cache Replacement Policies

Based on the metrics defined above, we can implement several cache replacement policies.

LRU.

The simplest cache replacement policy is to always replace the least recently used cache entry. This policy is simple, but is not able to use the semantic information that the caching method makes available. Therefore, it is unable to separate between query results that require a lot of computational effort to reproduce and results that are easily recreated from scratch.

LRU + cost.

An improvement is to use the cost measure in addition to usage to decide which cache entry should be replaced next. The semantic information allows us to estimate the cost of each step in the query and to use cumulative cost to either favor queries of high or low complexity. The cost measure is given more weight than LRU, but as a cache entry ages also complex results may be replaced. We call these policies LC policies.

The LC^+ policy favors results of queries of high complexity, making sure that results that would take longer to reproduce are kept in cache longer than results of queries that have a lower computational complexity. The complex queries are not the most frequently reused, but more time is saved for each cache hit.

Our LC^- policy does the opposite. It adds a penalty to the complex queries. This means that the results of queries of lower complexity, which are expected to be more reusable and produce more cache hits, stays longer in cache. The choice between LC^+ and LC^- is a weighing of savings per cache hit versus number of hits.

LRU + height.

The cost estimate is a quite complex estimate to make. As described earlier, the height of the query tree may be a simple alternative to the full cost estimate. Like cost estimates, height can be used to favor either complex or simple queries, so we divide the LH policies into LH^+, which favors results of complex queries, and LH^-, which favors versatile results.

4.4 Cache Registration

As soon as the last tuple is inserted into a cache entry, it is made available for use. However, it is not registered in the catalog until the site sends an update message to the catalog service.

The sites regularly update the catalog with information about local table fragments, and we extend these updates with information about cache entries. For each entry in the local cache, a site sends a representation of the algebra subtree and timestamps for each table fragment accessed by the subtree.

There are generally more than one table involved in a query. In DASCOSA-DB, the catalog of fragments of one table is stored on one catalog service site. To avoid having to register the cache entries on the catalog service sites for all tables involved in the query, we use a hashing function to select one site to which the query is sent. For each query, the hashing function is used to select the catalog site for one of the involved tables.

Since the cache entry for the result of an algebra tree is always registered at the catalog site storing information on one of the tables referenced by the algebra tree, our method guarantees that by sending the query with the lookup request for all tables, all cache entries are found. Due to the hashing function, the catalog of cache entries is distributed among catalog sites.

4.4.1 Cache Currency and Invalidation

The catalog stores a lower bound on the timestamps of all table fragments. When a catalog site receives an update message for a cache entry c, it can compare the timestamps $ts^c_{T_i} \in ts_c$ of the cache entry with the timestamps of table fragments $ts^K_{T_i}$ from its part K of the global catalog. If $\exists\, T_i : ts^c_{T_i} < ts^K_{T_i}$, table fragment T_i has been updated after the cache entry was created and this is registered in the catalog. The cache entry will not be registered, and the caching site is informed of the new timestamp.

Similarly, if $\exists\, T_i : ts^c_{T_i} > ts^K_{T_i}$, table fragment T_i has been updated since last catalog update, and $ts^K_{T_i}$ is updated. In this way, the cache entry updates are actually improving the catalog service freshness.

4.5 Transactional Support

Implementation of semantic caching does not change transactional support or locking policies in DASCOSA-DB. All locks for a query are acquired by the initiator site before query dissemination, and our semantic caching method does not change that behavior. The initiator site will acquire locks for all table fragments that are

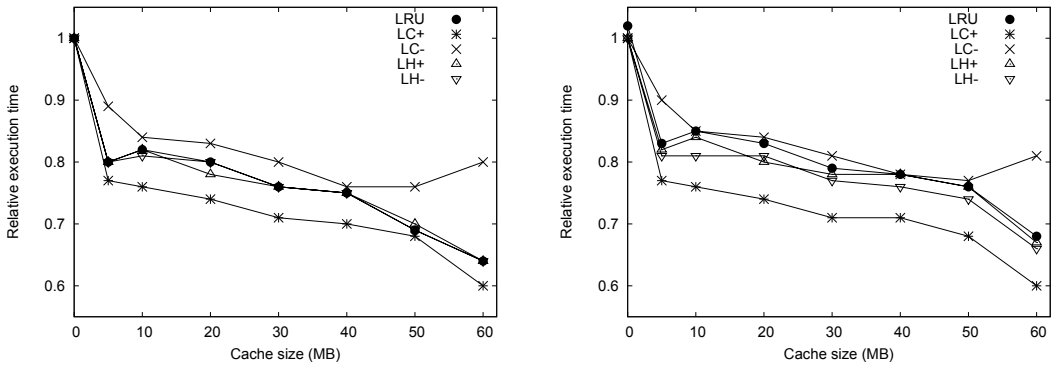

Figure 6: (a) Relative execution time of high-bandwidth system with uniform workload. (b) Relative execution time of low-bandwidth system with uniform workload.

needed, including those that form the basis for cache entries used by the query, thereby guaranteeing full transactional isolation.

5. EXPERIMENTAL EVALUATION

To evaluate the caching method, we have implemented it in the DASCOSA-DB distributed database system prototype. The caching method has been tried with the caching policies described in Section 4.3.2 and different query workloads. We used the DASCOSA-DB's existing cost functions for cost estimation.

The experiments were done on a TPC-H [19] dataset using scaling factor $SF = 0.1$, partitioned horizontally based on primary key and distributed over five sites, each running an instance of DASCOSA-DB. One more site with no table fragments was used for issuing queries. We generated query workloads consisting of 200 queries from the TPC-H benchmark queries, varying all substitution parameters of the benchmark. The substitution parameters were drawn either from a uniform distribution or from a skewed distribution where 80% of the values are drawn from 20% of the domain.

We measured the execution time and cache hits of repeated executions of our query workload. These measurements are only meaningful on a relative scale, so execution time was measured relative to a baseline execution without caching. During this execution, caching code was completely disabled. Cache hits were measured relative to the number of queries in the workload.

5.1 Varying Network Bandwidth

In this experiment, the network bandwidth was varied to produce two different settings: a high-bandwidth setting with 100 Mbit/s links connecting the sites, and a low-bandwidth setting with 1 Mbit/s links. The substitution parameters of the queries were drawn randomly from the parameter domains, using a uniform distribution.

Figure 6(a) shows the average execution time relative to the execution time measured when caching was disabled, i.e., without any caching code running. The values shown for 0 MB cache thus show the overhead of the caching method, i.e., the extra cost associated with the caching method without any of its benefits. As is seen from the figure, the overhead is negligible.

Further, the graph shows that with the largest cache size, LRU saves 36% of the execution time, while LC^+ saves 40%. LH^+ and LH^- vary very little from LRU, while LC^-, favoring results of queries of low complexity, performs worst. This indicates that the savings gained from caching results of complex queries outweighs the possibilities for frequently used results of low complexity.

Comparing Figures 6(a) and 6(b), we see that the distance between LC^+ and LRU is greater in the low-bandwidth setting. This is in line with our expectations of LC^+, deciding to cache results of more complex queries, allowing for greater savings once they are used. When bandwidth is reduced, LC^+ stands out while the rest of the policies perform poorer.

5.2 Varying Parameter Distribution

In this experiment, query parameters were either drawn from a uniform distribution or from a skewed distribution where 80% of the values were drawn from 20% of the parameter domain. Network bandwidth was kept low (1 Mbit/s).

The skewed distribution was chosen to be a more realistic workload, where some values are more frequent than others. This would be the case in many real life systems. We believe the choice of 80/20 is a conservative one, which means slightly pessimistic results.

Figure 7(a) shows that LRU achieves a reduction in execution time of 55%, i.e., 11 percentage points more than with uniform parameter distribution, and LC^+ saves 56%. While LRU and LC^+ do not differ very much for large cache sizes, LC^+ achieves these savings also for much smaller cache sizes.

The comparison of cache hit rates in Figure 7(b) sheds some light on what is going on. LC^+ manages to keep the right entries in cache also for smaller cache sizes, while LRU needs large cache sizes to achieve this. We also note that a larger cache does not necessarily mean better hit rates, since cache entries are of different sizes. Large entries may displace multiple smaller entries, thus reducing the number of hits.

There is clearly an increased hit rate for the skewed workload, which is the reason for the improvement in execution time over the uniform workload.

6. CONCLUSION AND FUTURE WORK

In this paper, we have developed a new method for semantic caching in a distributed database system with autonomous sites, where caching policies and decisions can vary from site to site and workload statistics are sparse. By making sites autonomous, we allow the system to scale without excessive network traffic. We have shown how the result of subqueries can be cached and reused by subsequent, similar queries to speed up query processing.

We have implemented the semantic cache in the DASCOSA-DB

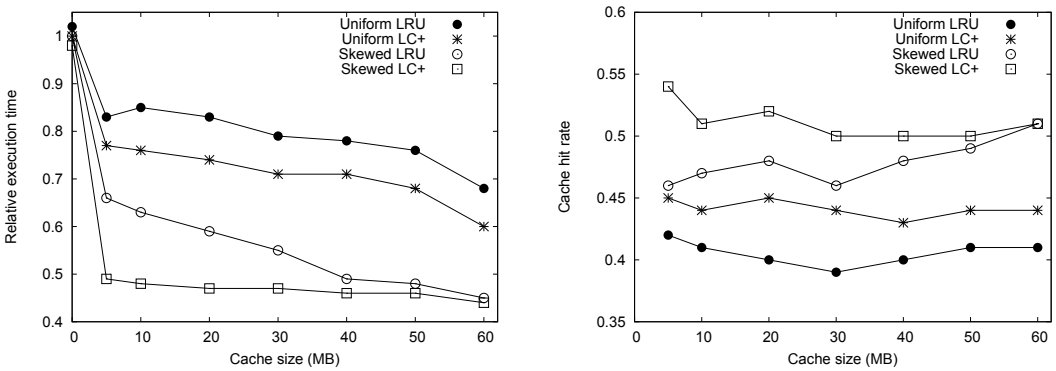

Figure 7: (a) Relative execution time of low-bandwidth system with both workloads. (b) Cache hit rates for low-bandwidth system with both workloads.

prototype as a proof of concept and platform for our experiments. The experiments have shown that we get considerable improvements in execution time when enabling semantic caching. The overhead of our caching method is also very low.

The cache hit rate is not the only factor influencing the performance. The cost of recomputing the cached data is also important. The savings made possible by caching the result of a complex query are sometimes higher than the savings from caching the results of less time-consuming queries with a higher hit rate. The information necessary to make such decisions is made available by semantic caching.

Our results indicate several ways to further improve query processing by semantic caching. The next step is to further enable the query optimizer to take advantage of cached intermediate results, including rewriting queries in otherwise suboptimal ways to increase cache hit numbers and rewrite queries to increase reusability of intermediate results. A semantic cache also allows for more advanced cache replacement policies, and further work should be done to find policies that care not only for the number of cache hits, but also the potential computational cost savings of a cache entry.

7. REFERENCES

[1] M. Altinel, C. Bornhövd, S. Krishnamurthy, C. Mohan, H. Pirahesh, and B. Reinwald. Cache tables: Paving the way for an adaptive database cache. In *Proceedings of VLDB*, 2003.

[2] H. Andrade, T. M. Kurç, A. Sussman, and J. H. Saltz. Active semantic caching to optimize multidimensional data analysis in parallel and distributed environments. *Parallel Computing*, 33(7-8):497–520, 2007.

[3] C. Bornhövd, M. Altinel, C. Mohan, H. Pirahesh, and B. Reinwald. Adaptive database caching with DBCache. *IEEE Data Engineering. Bulletin*, 27(2):11–18, 2004.

[4] U. S. Chakravarthy, J. Grant, and J. Minker. Logic-based approach to semantic query optimization. *ACM Trans. Database Syst.*, 15(2):162–207, 1990.

[5] S. Chaudhuri, R. Krishnamurthy, S. Potamianos, and K. Shim. Optimizing queries with materialized views. In *Proceedings of ICDE*, 1995.

[6] B. Chidlovskii, C. Roncancio, and M.-L. Schneider. Semantic cache mechanism for heterogeneous Web querying. *Computer Networks*, 31(11–16):1347–1360, 1999.

[7] S. Dar, M. J. Franklin, B. T. Jónsson, D. Srivastava, and M. Tan. Semantic data caching and replacement. In *Proceedings of VLDB*, 1996.

[8] L. M. Haas, D. Kossmann, and I. Ursu. Loading a cache with query results. In *Proceedings of VLDB*, 1999.

[9] J. O. Hauglid, K. Nørvåg, and N. H. Ryeng. Efficient and robust database support for data-intensive applications in dynamic environments. In *Proceedings of ICDE*, 2009.

[10] J. M. Hellerstein and J. F. Naughton. Query execution techniques for caching expensive methods. In *Proceedings of SIGMOD*, 1996.

[11] B. T. Jónsson, M. Arinbjarnar, B. Þórsson, M. J. Franklin, and D. Srivastava. Performance and overhead of semantic cache management. *ACM Transactions on Internet Technology*, 6(3):302–331, 2006.

[12] 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 *Proceedings of SIGMOD*, 2002.

[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, M. J. Franklin, G. Drasch, and W. Ag. Cache investment: integrating query optimization and distributed data placement. *ACM Transactions on Database Systems*, 25(4):517–558, 2000.

[15] D. Lee and W. W. Chu. Semantic caching via query matching for web sources. In *Proceedings of CIKM*, 1999.

[16] Q. Luo, S. Krishnamurthy, C. Mohan, H. Pirahesh, H. Woo, B. G. Lindsay, and J. F. Naughton. Middle-tier database caching for e-business. In *Proceedings of SIGMOD*, 2002.

[17] H. Mistry, P. Roy, S. Sudarshan, and K. Ramamritham. Materialized view selection and maintenance using multi-query optimization. *SIGMOD Record*, 30(2):307–318, 2001.

[18] Q. Ren and M. H. Dunham. Using semantic caching to manage location dependent data in mobile computing. In *Proceedings of MobiCom*, 2000.

[19] Transaction Processing Performance Council. TPC benchmark H (decision support) standard specification revision 2.11.0, 2010.