University of Zurich
Department of Informatics

Andrei Vancea
Burkhard Stiller

A Cooperative Semantic Caching Architecture for Answering Selection Queries

TECHNICAL REPORT – No. 2009.07

September 2009

University of Zurich
Department of Informatics (IFI)
Binzmühlestrasse 14, CH-8050 Zürich, Switzerland

A. Vancea, B. Stiller:
Technical Report No. 2009.07, September 2009
Communication Systems Group (CSG)
Department of Informatics (IFI)
University of Zurich
Binzmühlestrasse 14, CH-8050 Zurich, Switzerland
URL: http://www.csg.uzh.ch

A Cooperative Semantic Caching Architecture for Answering Selection Queries

Andrei Vancea

Communication Systems Group (CSG),
Department of Informatics (IFI), University of Zurich
Binzmühlestrasse 14
CH-8050 Zürich, Switzerland

vancea@ifi.uzh.ch

Burkhard Stiller

Communication Systems Group (CSG),
Department of Informatics (IFI), University of Zurich
Binzmühlestrasse 14
CH-8050 Zürich, Switzerland

stiller@ifi.uzh.ch

ABSTRACT

Semantic caching is a technique used for optimizing the evaluation of database queries by caching results of old queries and using them when answering new queries. CoopSC is a cooperative database caching architecture, which extends the classic semantic caching approach by allowing clients to share their local caches in a cooperative matter. Cache entries of all clients are indexed in a distributed data structure constructed on top of a Peer-to-Peer (P2P) overlay network. This distributed index is used for determining those cache entries that can be used for answering a specific query. Thus, this approach decreases the response time of database queries, because the server only answers those parts of queries that are not available in the cooperative cache.

Categories and Subject Descriptors

H.2 [Database Management]: Systems; H.3.4 [Information Storage and Retrieval]: Systems and Software—*distributed systems, performance evaluation*

General Terms

Performance, Measurement.

Keywords

Cache, peer-to-peer system, semantic cache

1. INTRODUCTION

Client side caching is a commonly used technique for reducing the response time of database queries [2]. Semantic caching [5] is a database caching approach in which results of old queries are cached and used for answering new queries. A new query will be split in a part that retrieves the portion of the result, which is available in a local cache (*probe query*) and a query that retrieves missing n-tuples for the database server (*remainder query*). This approach is especially suited for low-bandwidth environments or when database server is under heavy load. Semantic caching was successfully applied for optimizing the execution of queries on mobile clients or over lousy-coupled wide-area networks.

Peer-to-Peer (P2P) networks have been applied successfully for improving the performance of cache systems [11]. In consequence, this paper proposes using a P2P approach in order to enhance the semantic caching technique. The outcome of these development steps is CoopSC, an approach for Cooperative Semantic Caching. CoopSC [13] extends the general semantic caching mechanism by enabling clients to share their local semantic caches in a cooperative matter. When executing a query, the content of both the local semantic cache and entries stored in caches of other clients can be used. A new query will be split into a probe, *remote probes*, and a remainder query. The probe retrieves the part of the answer which is available in the local cache. Remote probes retrieve those parts of the query, which are available in caches of other clients. The remainder retrieves the missing n-tuples from the server.

Consider the following example: the client C_1 asks for all the persons older then 15 (Q_1 : *select * from persons where age > 15*). The server returns the result set, and client stores it in the local cache. The client C_2 asks for all persons between 5 and 10 (Q_2 : *select * from persons where 5 < age and age < 10*). A subsequent query, executed by C_1, which asks for all the persons younger then 20 (Q_3 : *select * from persons where age < 20)* will be split in a probe, a remote probe and a remainder. The probe query retrieves, from the local cache, all persons between 15 and 20. The remote probe, which will be sent to C_2 returns the persons between 5 and 10. Then remainder retrieves the missing n-tuples from the server (*select * from persons where 0 < age and age <= 5 or 10 <= age and age <= 15*). These sub-queries are executed in parallel and after their executions their results are integrated into a final result. This approach increases the throughput of database servers, because servers only handle the portions of queries that can not be answered using the cooperative cache. Also, the amount of data sent by database servers is significantly reduced.

Therefore, this paper solves the following problem: given a database server and a number of clients that execute queries in parallel and cache the results of old queries, design an architecture that allows the cache entries to be shared between clients.

This paper is organized as follows: While Section 2 discusses related work, Section 3 outlines the architecture of CoopSC. Section 4 describes the implementation of a prototype system, based on the architecture outlined in Section 3. Some initial evaluation results are presented in Section 5. Finally, some concluding remarks and an overview of the future work are given in Section 7.

2. RELATED WORK

The semantic caching approach was introduced originally in [5]. This paper describes semantic caching concepts and compares the approach with page and tuples caching. The cache is organized into disjoint *semantic regions*. Each semantic region contains a set of n-tuples and a constraint formula, which describes the common property of the n-tuples. Experiments were performed for single and double attribute selection queries. [6] runs an extensive performance study of a

semantic caching prototype implementation. It shows that the performance of semantic cache systems degrades rapidly when increasing the number of dimensions of the selection predicate.

XCache [3] determines a semantic caching architecture developed for XML (eXtended Markup Language) queries. The system implements algorithms for checking the query containment for XQueries and algorithms that perform query rewriting.

The first two approaches described ([5], [3]) do not allow clients to share their caches in a cooperative way. Thus, only local cache entries can be used for answering queries. In contrast, CoopSC extends the approach described in [5] by allowing clients to cooperate in order to increase the efficiency of caching system.

The Wigan system [4] caches old results of database queries in order to answer new queries and to allow for the entries cached to be shared between clients. In Wigan, a cached query Q_1 can be used for answering a query Q_2 only, if Q_2 is strictly subsumed by Q_1. In real world applications, the number of cases, in which this happens, is limited. In the same way CoopSC does also support cases, in which there is only an overlap between Q_1 and Q_2. A difference between Wigan and CoopSC is the way, in which the system finds an entry in a remote cache that can be used for answering a query. Wigan uses a centralized approach, where a tracker knows the content of the caches of all clients, while CoopSC uses a completely distributed approach. A centralized approach may show in certains cases scalability and reliability problems (the tracker represents a single point of failure) [9], which can be avoided in a fully decentralized approach.

[8] describes a cooperative caching architecture for answering XPath queries with no predicates. Two methods of organizing the distributed cache are proposed: (a) *IndexCache*: each peer caches the results of its own queries; and (b) *DataCache*: each peer is assigned a particular part of the cache data space. The IndexCache approach has similarities with the CoopSC approach. In both approaches results of old queries are indexed in a distributed data structure. The main differences are related to the type of queries supported and the way, in which cached entries are used for answering new queries. While [8] works with the XML data model and support XPath queries, CoopSC is built on top of a relational database and supports selection queries. XPath queries assumes a hierarchical XML structure and returns a sub-tree of the structure, while the selection query return n-tuples which for which a particular predicate is true.When answering a query, the XPath approach searches for a cache entry that strictly subsumes the given query. Thus, partial hits are not supported. In contrast, CoopSC supports partial hits by splitting queries into probes, remote probes, and remainders.

ACS (Adaptive Caching Service) [10] is an adaptive caching service built on top of the Gedeon data management system [12]. It provides a flexible programming interface that allows for new types of cache approaches to be implemented. The system does a separation between query and object caches. It also allows cache entries of clients to be shared in a cooperative matter. The cooperation is done using a flooding approach, but the ACS allows new types of cache resolution to be added. In order to overcome the scalability issues of flooding, the clients are divided into communities. Thus, only clients that are in the same community can cooperate. CoopSC uses a distributed index in order to determine which remote

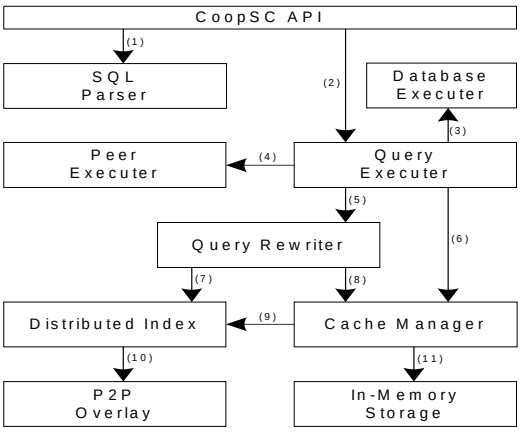

Figure 1. CoopSC Architecture

cache entries can be used for answering a query. ACS handles point queries, while CoopSC supports range selection queries. Also, ACS only supports strict hits between query entries, while CoopSC also handles partial hits.

In summary, Table 1 illustrates the key differences between the semantic caching approaches investigated.

Table 1. Cooperative Semantic Caching Approaches

Approach	Data Model	Query Types	Cache Hit Types	Resolution Method
Wigan [4]	Relational	Range selections	Strict	Centralized tracker
XPath IndexCache [8]	XML	XPath (no predicates)	Strict	Distributed index
ACS [10]	Gedeon	Point queries	Strict	Flooding
CoopSC [13]	Relational	Range selections	Strict, Partial	Distributed index

3. ARCHITECTURE

The new CoopSC approach developed handles the execution of range selection queries. Its architecture must allow clients to store results of old queries and use them for answering new queries. A mechanism must also be provided to allow cache entries to be shared between clients.

Similarly with the approach presented in [5], the local cache is organized into disjoint semantic regions. A *semantic region* is defined as a set of n-tuples and a constrained formula which determine the common property of the n-tuples. Clients interrogating a specific database server form the P2P overlay network, which is used for indexing the semantic regions.

3.1 Components

The main components of the CoopSC architecture are illustrated in Figure 1

The *Query Executer* handles the execution of queries. It first splits the query into probe, remote probes and remainder sub-queries. Further, it executes each sub-query, in parallel, by accessing the local cache entries or by sending it for execution to either a different client to database server. After an

execution of sub-queries, the *Query Executer* integrates their result sets and returns the final result of the query. It also handles the execution of queries, which originated from other clients.

The *Cache Manager* (cf. Figure 1) stores semantic regions and implements the LRU (Least Recently Used) replacement policy. LRU is motivated by the concept of temporal locality: a recently used entry has a good chance to be used the near future.. The distributed index must be synchronized with the content of the clients' caches, thus when a semantic region is added or removed from the cache it. updates the distributed index

The *Query Rewriter* (cf. Figure 1) splits a query into probes, remote probes, and remainder sub-queries. In the first step, the Query Rewriter determines the probe, which is the part of the query that is available in the local cache. This is accomplished by scanning local semantic regions and checking, if they overlap with the query. This first step also returns the *local remainder*, which represents the portion of the query, which is not available in the local cache. In order to determine *remote probes* and the *remainder*, the distributed index is interrogated with the local remainder. The query rewriting process in illustrated in Figure 2.

All semantic regions are indexed in a *Distributed Index* (illustrated in Figure 1. The purpose of the distributed index is to make the query rewriting process efficient. For each query given as input the distributed index component returns a list of semantic regions that semantically overlap the query and minimize the portion of query that must be executed by the database server. The distributed index is described in Section 3.3.

The *Database Executer* (cf. Figure 1) execute queries on database server and returns result sets, while the *Peer Executer* executes queries that return n-tuples from the semantic caches of other CoopSC clients.

The *CoopSC API* (Application Programming Interface) is a programming interface that allows writing applications that use the CoopSC architectures

3.2 Interactions

The interactions between the CoopSC components are also illustrated in Figure 1.

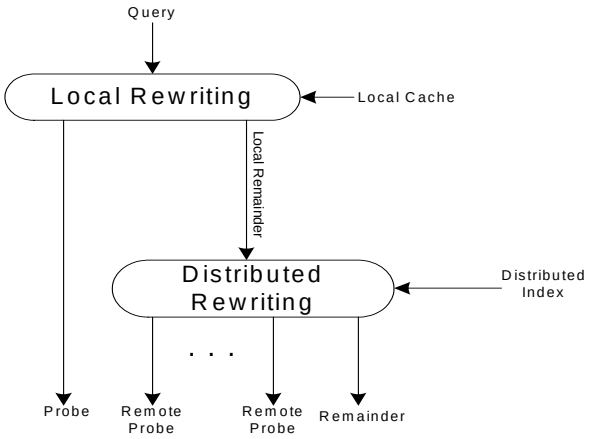

Figure 2. Query Rewriting

The Query Executer first the split the queries into probes, remote probes and remainder using the Query Rewriter (5). The sub-queries are executed by either accessing the local cache (6) or sending them to Peer Executer (4) or to Database Executer (3).

The Query Rewriter first accesses the local cache (8) in order to calculate the probe and local remainder. Remote probes and the remainder are calculated by interrogating the distributed index (8).

The CoopSC API provides an interface that allows the execution of database queries. The queries are first parsed (1) and then sent to Query Executor (2).

3.3 Distributed Index

This section describes the distributed structure that is used for indexing semantic regions. Only single attribute selections are considered, but, afterwards, the way in which this approach can be generalized for multi-attribute selections is presented.

As mentioned in the beginning of the section, semantic regions are defined by a set of n-tuples and a predicate. Under the given assumptions, the predicate is a single attribute selection. (Example : *10 <= age and age <= 30 or 40 <= age and age <= 50*). Queries are also single attribute selections (Example : *select * from persons where 20 <= age and age <= 45*). Single attribute selection predicates can be represented as sets of non-overlapping intervals (Example: {[10, 30], [40, 50]}). This representation will be used for both semantic regions and queries. Semantic regions must also contain information about the clients that store them and local identifiers, for differentiating regions stored by the same client. In order to simplify the notation, the name of selection attribute is discarded. Thus, a semantic region is represented as a triple that contains the address of the client, the local identifier and the set of intervals (Example: R = (192.168.100.40, 1, {[10, 30], [40, 50]}) while a query is represent as a set of intervals (Example: Q={[20, 45]}).

3.3.1 Problem Description

Given a query Q, the distributed index must return a list of semantic regions that overlap the query and minimize the part of the query that is not covered. The result is named the *rewriting* of query Q.

Example:

Consider the following four semantic regions:

- R_1=(192.168.0.100, 1, {[10, 30], [40, 60]}),
- R_2=(192.169.0.202, 2, {[100, 120]}),
- R_3=(192.168.20.23, 4, {[20, 50], [180, 190]}),
- R_4=(192.168.0.100, 2, {[80, 90]}),

and a query Q={[0, 50]}. The semantic regions R_1 and R_3 minimize the portion of query Q not covered ([0, 9]).

3.3.2 Solution

DST (Distributed Segment Tree) [14] is a distributed data structure that supports the execution of range and cover queries in a structured P2P environment. DST was adapted in order to support the type of queries described in Section 3.3.1.

Figure 3 illustrates the distributed segment tree for the interval [0, 7]. Each node of the tree is stored in a member of the P2P overlay. The association between nodes and peers is done by applying a hash function on intervals and selecting, for each interval, the peer that has the closest ID to the hash value.

These intervals from the tree are available in the form of [x * 2^y , (x + 1)* 2^y - 1], which are called *dyadic intervals* [6]. Any interval of size R can be expanded by a union of no more then *2*log R* dyadic intervals [6]. Example: the interval [2, 6] can be expanded as [2, 3], [4, 5], [6, 6] (cf. Figure 3).

Semantic regions are indexed in the following way: intervals associated with regions are expanded into dyadic intervals. For each dyadic interval, the DST stores the client's address and the local identifier. Example: when indexing the region R=(192.168.0.1, 4, {[2, 6]}), nodes [2, 3], [4, 5] and [6, 6] will store the entry (192.168.0.1, 4).

In order to determine the rewriting of a query, its intervals are expanded into dyadic intervals. The rewriting is calculated separately for each dyadic interval, in the following way: if the node associated with the dyadic interval contains entries, the first entry is returned. Example: interval [4, 5] has entries, thus, the region (192.168.0.1, 4) is returned. Otherwise, ancestor nodes (*e.g.*, parent, grandparent) are checked. If any of ancestor contains entries, the first one is returned. Iterating through ancestors is done in logarithmic time. Example: interval [2, 2] does not have any entries, but its parent has. Thus, region (192.168.0.1, 4) is returned. If no such ancestors are found, the rewriting must be calculated by checking descendant nodes. Example: the rewriting of interval [0, 3] is the region (192.168.0.1, 4), which is only present in node [2, 3]. In order to avoid accessing all descendants nodes, which is executed in linear time, for each node, the rewriting that contains only descendant intervals is precalculated, when semantic regions are added to the index. Thus, for each dyadic interval, the distributed index stores the set of semantic regions that contain the given interval and the rewriting that contains only descendants nodes.

Figure 5 illustrates the content of the distributed index after the regions (192.168.0.1, 4, {[2, 6]}) and (192.168.0.1, 5, {[1, 2]}) were added. *R* is set the regions that contains the given interval while *L* is the rewriting that contains only descendant nodes.

For each node, the rewriting *L* is calculated in the following way: if the left child contains regions, the first regions is added; otherwise left child's *L* rewriting is added. The same operation is executed for the right child. The algorithm is presented in Figure 4. When adding or removing semantic

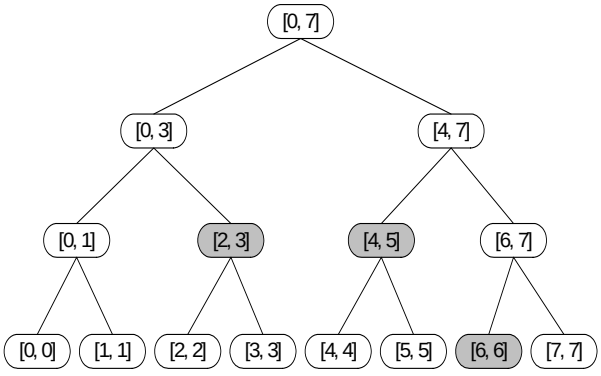

Figure 3. Distributed Segment Tree

```
procedure calculateL(node)
begin
   node.L = {}
   if (isLeaf(node)) return
   if (not node.left.R.empty())
      node.L = node.left.R[1]
   else
      node.L = node.left.L
   if (not node.right.R.empty())
      node.L = node.L U node.right.R[1]
   else
      node.L = node.L U node.right.L
end
```

Figure 4. Descendant Rewriting Pseudo-code

regions in the index, the L rewriting is recalculated for all ancestors nodes. This operation is performed in logarithmic time.l

DST can be generalized to work with multiple dimensions [14]. For *d* dimensions, each node has 2^d children. [14] describes how a multi-dimensional interval can be expanded into dyadic multi-dimensional intervals. The rest of the algorithm for calculating the rewriting is independent of the number of dimensions.

4. IMPLEMENTATION

A prototype C++ implementation of the CoopSC architecture was developed. The implementation works with the PostgreSQL database and uses the Chimera P2P overlay [15]. PostgreSQL was chosen because it is free, full-feature, and a very mature database system. Chimera is a light-weight and efficient P2P overlay developed in C++.

In order to demonstrate the functionality of the CoopSC architecture, a graphical user interface (GUI) was also implemented. The CoopSC GUI enables users to execute SQL (Structured Query Language) queries using the CoopSC cooperative cache. Both the result of the execution and the way in which the query was executed are displayed Figure 3 illustrates the CoopSC graphical user interface. The application windows is divided into three parts. The upper-left text box allows user to enter queries that are to be executed

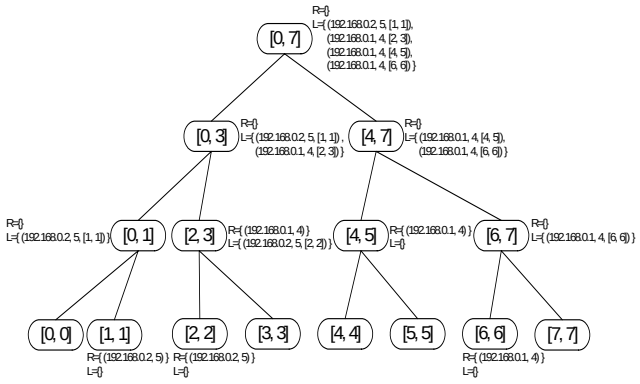

Figure 5. CoopSC Distributed Index

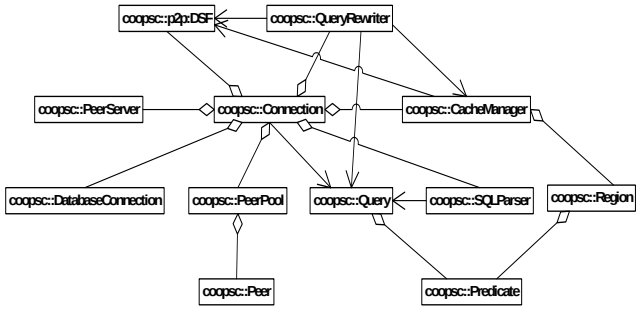

Figure 7. CoopSC UML Diagram

using CoopSC. The lower-left table displays results of query executions. The right side of the window contains a tree structure that shows how queries were split into probe, remote probes and remainders.

The UML (Unified Modeling Language) diagram of the CoopSC implementation is illustrated in Figure 7.

The Query Executer component is implemented using the classes *coopsc::Connection* and *coopsc::PeerServer*. The class *coopsc::Connection* handles the execution of local queries while *coopsc::PeerServer* handles the execution of queries originated from other clients.

The classes *coopsc::PeerPool* and *coopsc::Peer* implement the Peer Executer component. *coops::PeerPool* maintains a list of connections to other clients (instances of *coopsc::Peer* class). Maintaining such a list increases the performance of the system because a single connection can be used when answering multiple queries

The class *coopsc::CacheManager* contains a list semantic regions. Each semantic region is modeled as an instance of the class *coopsc::Region*.

The distributed index is implemented in the class *coopsc::p2p::DSF*. The distributed index accesses the Chimera P2P overlay when a new entry is added or removed from the index.

The class *coopsc::QueryRewriter* implements the query rewriting algorithm, as described is Section 3.

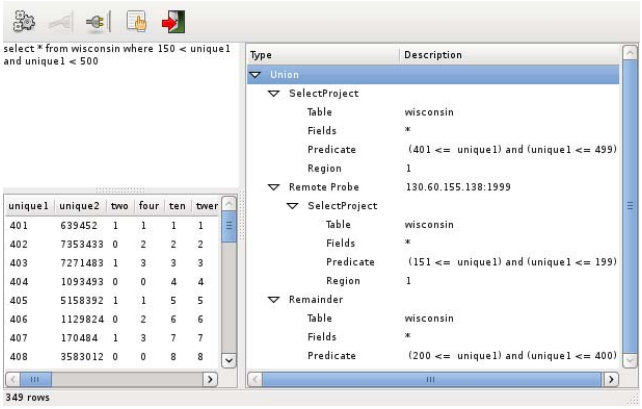

Figure 6. CoopSC GUI

5. EVALUATION

The CoopSC system was evaluated using a PostgreSQL database server and a number of clients that execute, in parallel, single attribute selection queries on a indexed unclustered attribute. Servers and clients are locate in the same LAN (Local Area Network), as shown in the evaluation scenario setup as illustrated in Figure 8. The set of same queries were executed under three different scenarios: (a) without using the cache; (b) using only the local semantic cache; and (c) using the cooperative semantic cache. In each scenario, the average response time was measured.

The evaluation was done using the Wisconsin benchmark relation [1] of 10 million n-tuples, where each n-tuple contains 208 bytes of data. Each query is a range selection on *unique1* attribute (Example: *select * from wisconsin where 4813305 < unique1 and unique1 < 4823306*), which returns 10,000 n-tuples. Similarly with the evaluation of other cache architectures [5], [6], queries executed by each clients have a semantic locality. For each client, the centerpoints of queries were randomly chosen to follow a normal distribution curve with a standard deviation of 180,000. Means of these curves are uniformly distributed over the range of the *unique1* attribute.

The size of clients' caches is 64 MB. The number of clients are varied from 2 to 45. When 45 clients are used the response time for the no-caching scenario becomes unreasonable high (more then 13 seconds) and the evaluation is stopped. Clients first execute 50 warm-up queries. The response time, for each client, is calculated by averaging the response time of following 500 queries. In order to determine the average response time of a particular scenario, the response times of all clients are averaged again. The warm-up queries were necessary in order to make sure that the clients' caches are full before starting these measurements. In order to improve the precision, a single measurement is made after the execution of 500 queries and the average response time is computed by dividing the result obtained to the number of queries.

Key results of this evaluation are presented in Figure 9. As it can be seen, the response time of the cooperative caching approach is lower compared to the scenario when no caching is used or when only local semantic caching is utilized. As the number of clients increases, the database server has to handle the execution of more queries in parallel, thus, the average response time increases. When running queries using the semantic caching approach, the server has to execute only parts of these queries that were not found in local caches of these

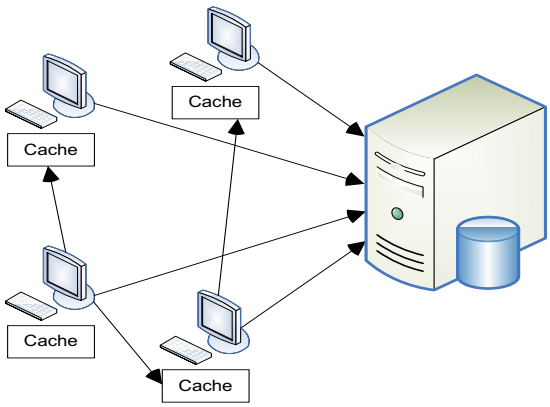

Figure 8. Evaluation Set-up

clients. This decreases the average response time. When using CoopSC, cache entries are shared between clients. This causes a further decrease of the average response time, because the hit rate of the cache system increases.

6. DEMOSTRATION DESCRIPTION

The demonstration to be shown consists of a running PostgreSQL database server and a number of CoopSC-GUI instances that execute various selection queries. Results of queries executions and the way in which the original queries were split into probes, remote probes, and remainders are displayed. Response times of queries are also shown and compared with the classic semantic caching approach. This clearly demonstrates the performance improvements of CoopSC architecture.

7. CONCLUSIONS AND FUTURE WORK

This paper presents CoopSC, a cooperative semantic caching architecture, that optimizes the execution of database queries by caching old query results in order to answer new queries, allowing clients to share their cache entries in a cooperative matter. The key components of the CoopSC architecture were described, details outlined, and implemented. The proposed architecture was evaluated and compared with the classic semantic caching approach. The evaluation results show that CoopSC reduces the response time of range selection queries when the database server is under heavy load.

Further experiments will investigate how the size of the selection, size of the cache and semantic locality of the queries affect the performance of the system. Evaluations for multi-dimensional selections are also planned.

8. REFERENCES

[1] D. Bitton, C. Turbyfill, A Retrospective on the Wisconsin Benchmark, Readings in Database Systems, pp 280-299, San Francisco, California, USA, 1988

[2] M. J. Carey, M. J. Franklin, M. Livny, E. J. Shekita, Data CachingTtradeoffs in Client-Server DBMS Architectures, ACM SIGMOD Record, Vol. 20, Issue 2, pp 357-366, May 1991.

[3] L. Chen, E. A. Rundensteiner, S. Wang, XCache: A Semantic Caching System for XML Queries, ACM SIGMOD international conference on Management of data, Madison, Wisconsin, USA, pp 618, June 2002

[4] J. Colquhoun, P. Watson, A Peer-to-Peer Server based on BitTorrent, Technical Report No. 1089, School of Computing Science, Newcastle University, April 2008

[5] S. Dar, M. J. Franklin, B. Jonsson, D. Srivastava, M. Tan, Semantic Data Caching and Replacement, 22th International Conference on VLDB, Bombay, India, pp 330-341, September 1996

[6] A. C. Gilbert, Y. Kotidis, S. Muthukrishnan, M. Strauss, How to summarize the universe: Dynamic maintenance of quantiles, 28th International Conference on VLDB, Hong Kong, China. pp 454–465, August 2002.

[7] B. Jónsson, M. Arinbjarnar, B.Þórsson, M.J. Franklin, D. Srivastava, Performance and Overhead of Semantic Cache Management, ACM Transactions on Internet Technology, Vol. 6, Issue 3, pp 302-331, 2006

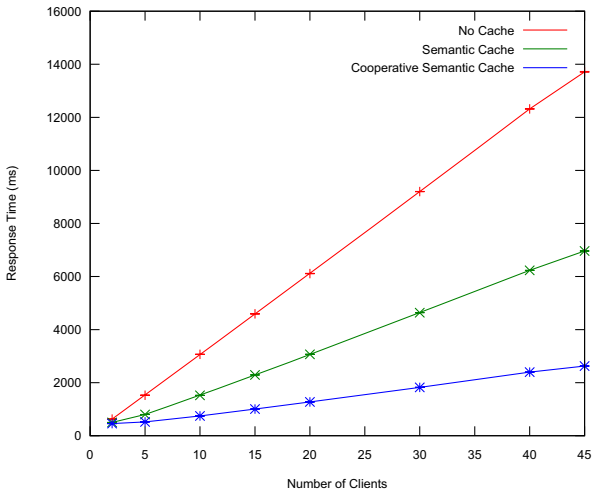

Figure 9. Evaluation Results

[8] K. Lillis, E. Pitoura, Cooperative XPath Caching, ACM SIGMOD International Conference on Management of Data pp 327-338, Vancouver, Canada, June, 2008

[9] K. Lua, J. Crowcroft, M. Pias, R. Sharma, S. Lim, A Survey and Comparison of Peer-to-Peer Overlay Network Schemes, IEEE Communications Surveys and Tutorials, Vol. 7, pp. 72-93, 2005

[10] L. d'Orazio, C. Roncancio, C. Labbé, F. Jouanot, Semantic Caching in Large-scale Querying Systems. Revista Colombiana De Computación (RCC), Vol. 9, No 1, pp 33-57, June 2008

[11] V. Padmanabhan, K. Sripanidkulchai, The Case for CooperativeNnetworking, International Peer-To-Peer Workshop, Cambridge, Massachsusetts, USA, pp 178-190, March 2002.

[12] O. Valentin, F. Jouanot, L. d'Orazio, Y. Denneulin, C. Roncancio, C. Labbé, C. Blanchet, P. Sens, C. Bonnard, Gedeon, un intergiciel pour grille de données, Conférence Française en Système d'Exploitation, Perpignan, France, October 2006

[13] A. Vancea, B. Stiller, Answering Queries Using Cooperative Semantic Caching, 3rd International Conference on Autonomous Infrastructure, Management and Security, University of Twente, The Netherlands, pp 203-206, July 2009

[14] G. Shen, C. Zheng, W. Pu, and S. Li, Distributed Segment Tree: A Unifed Architecture to Support Range Query and Cover Query, Technical Report No MSR-TR-2007-30, Microsoft Research, Redmond, WA, USA, March 2007

[15] B. Zhao, M. Allen, G. Swamynathan, K. Puttaswamy, L. Ganesh, Chimera, http://p2p.cs.ucsb.edu/chimera/html/home.html, September 2009