CoopSC: A Cooperative Database Caching Architecture

Vancea, A; Stiller, B
CoopSC: A Cooperative Database Caching Architecture

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 and the amount of data sent by database server, because the server only answers those parts of queries that are not available in the cooperative cache.
CoopSC: A Cooperative Database Caching Architecture

Andrei Vancea, Burkhard Stiller
Communication Systems Group CSG, Department of Informatics IFI, University of Zürich
Binzmühlestrasse 14, CH—8050 Zürich, Switzerland
Email: [vancea|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 manner. 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 and the amount of data sent by database server, because the server only answers those parts of queries that are not available in the cooperative cache.

Keywords — Cooperation in P2P, Semantic Caching, Data Base, P2P Application, Implementation, Prototype

I. INTRODUCTION

A way of achieving scalability in database management systems is to effectively utilize resources (storage, CPU) of client machines. 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 that is available in a local cache (probe query) and a query that retrieves missing n-tuples from the database server (remainder query). This approach is especially suited for low-bandwidth environments or when the database server is under heavy load. Semantic caching was successfully applied for optimizing the execution of queries on mobile clients or over loosely-coupled wide-area networks [10]. Semantic caching requires more resources on clients. Storage is needed for storing cache entries. Clients’ CPU usage will also increase, because they, locally, execute the probe sub-query.

In most applications, database servers are queried by multiple clients. When using the classic semantic caching approach, clients store and manage their own local caches independently. If the number of clients is high, the amount of data sent by database server and queries response times can rapidly increase even when caching is used. The performance can be further improved by allowing clients to share their entries in a cooperative way. Another limitation of existing semantic caching solutions is that they do not handle update queries. Modification performed in the database are not propagated to cache entries stored by clients.

Therefore, the Cooperative Semantic Caching (CoopSC) approach extends the general semantic caching mechanism by enabling clients to share their local semantic caches in a cooperative manner. 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. In order to execute the query rewriting, the cache entries of all clients will be indexed in a distributed data structure built on top of a Peer-to-peer (P2P) overlay that is formed by all clients which are interrogating a particular database server. Additionally, CoopSC designs a suitable and efficient mechanism for handling update queries. When the content of the database is changed, modifications are reflected in the cooperative cache. Furthermore, CoopSC supports select-project queries, where the query predicate is a n-dimensional range condition, which are commonly used in many database applications.

CoopSC decreases the response time of database queries, 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 can be significantly reduced. Thus, this approach is suited in the following two scenarios: (a) Database server and clients are located in a higher-bandwidth local-area environment. Clients execute a large number of queries in parallel. In this scenario, server’s processing resources (CPU, disk access) are the bottle-neck of the system. Using cooperative caching decreases queries response time because it reduces servers’ resources usage; (b) Database server and clients are located across the Internet in a network-constrained environment (e.g., the infrastructure of multi-national corporation). Clients execute queries that return a large amount of data (n-tuples). In this scenario, the cooperative caching approach is beneficial because it reduces the amount of data sent by database servers.

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 II discusses related work, Section IV outlines the architecture of CoopSC. Section V describes the implementation of a prototype system, based on the architecture outlined in Section IV. Some initial evaluation results are presented in Section VI. Finally, some concluding remarks and an overview of the future work are given in Section VII.

II. RELATED WORK

The semantic caching approach was introduced in [5] as the basic concept. [5] 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 for range queries up to four attributes. However, the classic semantic caching approach — as referred to in [3] and [6] — does not handle update queries.

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. However, update queries are also not handled in [3].

Furthermore, these two approaches described in [5] and [3] do not allow clients to share their caches in a cooperative way. Thus, only local cache entries can be used for answering queries. 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. Wigan supports only queries that can be express as conjunctions of single attribute range conditions. Thus, the query “select * from earthquakes where lat < 10 and long < 20” is supported, while “select * from earthquakes where lat < 10 or long < 20” is not. A cached query Q₁ can be used for answering a query Q₂ only, if Q₂ is strictly subsumed by Q₁. In real world applications, the number of cases, in which this happens, is limited. Another drawback of this approach is that it uses a centralized tracker in order to determine, which cached query can be used when answering a new query. A centralized approach will show in certain cases scalability and reliability problems, since the tracker represents a single point of failure [8]. This can be avoided in a fully decentralized approach. Furthermore, Wigan does not handle update queries, too.

[7] 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 approach works with the XML data model and supports simple XPath queries that have no selection predicates. XPath queries assume a hierarchical XML structure and return a sub-tree of the structure. When answering a query, the XPath approach searches for a cache entry that strictly subsumes the given query. Thus in consequence, partial hits are not supported. Another problem with this approach is that is does not handle update queries as well.

Dual Cache [9] is an caching service built on top of the Gedeon data management system [12]. 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 new types of cache resolution can be added. In order to overcome the scalability issues of flooding, clients are divided into communities. Thus, only clients that are in the same community can cooperate. Dual Cache handles non-range selections only (e.g.: lat = 20 and long = 50) and supports only strict hits between query entries. Update queries are also not handled.

Therefore and in summary, Table 1 illustrates the key differences between the cooperative semantic caching approaches investigated as well as outlining already for comparing dimensions the new CoopSC system.

**Conclusion:** Existing cooperative caching systems lack the support for complex query types. There are no approaches that handle generic n-dimensional range selections. Another limitation of existing solutions is the way in which cache entries are used for answering a new query: existing approaches only look for an entry that strictly subsumes the query. Thus, combining multiple entries in order to answer a given query is not supported. Furthermore, most approaches do not provide a scalable way of finding, which entries are suitable for answering new queries. Another challenge being faced with is the design of an efficient mechanism for handling update queries that shall be applied to both classic and cooperative semantic caching approaches. The CoopSC projects aims at solving these challenges mentioned above, while the CoopSC’s very basic idea has been published in [13].

### Table 1. Cooperative Caching Approaches

<table>
<thead>
<tr>
<th>Approach</th>
<th>Data Model</th>
<th>Query Types</th>
<th>Cache Hit Types</th>
<th>Resolution Method</th>
<th>Updates</th>
</tr>
</thead>
<tbody>
<tr>
<td>Wigan [4]</td>
<td>Relational</td>
<td>Simple range selections</td>
<td>Strict</td>
<td>Centralized tracker</td>
<td>No</td>
</tr>
<tr>
<td>XPath Index-Cache [7]</td>
<td>XML</td>
<td>XPath (no predicates)</td>
<td>Strict</td>
<td>Distributed index</td>
<td>No</td>
</tr>
<tr>
<td>Dual Cache[9]</td>
<td>Gedeon</td>
<td>Non-range queries</td>
<td>Strict and partial</td>
<td>Distributed index</td>
<td>No</td>
</tr>
<tr>
<td>CoopSC [13]</td>
<td>Relational</td>
<td>n-Dimensional range select-project queries</td>
<td>Strict and partial</td>
<td>Distributed index</td>
<td>Yes</td>
</tr>
</tbody>
</table>

### III. SAMPLE APPLICATION FOR COOPSC

The following example illustrates a suitable application of the approach. In the context of an international seismological research project, consider a central database that stores data about earthquakes. For each earthquake, the database keeps the location (latitude, longitude), the time of the event, the magnitude and other relevant information. The database is accessed by clients which are located in different research centers across the world. The research centers need data about the events that happened in a particular area, in specified interval of time and of a certain magnitude. This type of interrogations can be easily expressed as a n-dimensional range query. Because such a database is usually very large, the database server has to send a large amount of data. A cooperative caching approach can significantly reduce this amount of data.

Consider the following example: client C₁ asks for the events that happened in the area between (20, 20) and (40, 40) (Q₁: select * from earthquakes where 20 < lat and lat < 40 and 20 < long and long < 40). The server returns the result set, and the client stores it in the local cache. Client C₂ asks for the earthquakes that happened in the area between (30, 30) and (50, 50) (Q₂: select * from earthquakes where 30 < lat and lat < 50 and 30 < long and long < 50). As it can be clearly seen, the two areas overlap. Thus, Q₂ will be split in a remote probe, which will be sent to C₁, that returns the events that happened between (30, 30) and (40, 40) (select * from earthquakes where 30 < lat and lat < 40 and 30 < long and long < 40) and a remainder that returns the missing n-tuples from the server (select * from earthquakes where 30 < lat and lat < 50 and 30 < long and long < 50 or 30 < lat and lat < 40 and 39 < long and long < 50).

### IV. COOPSC ARCHITECTURE

The new CoopSC approach developed handles the execution of n-dimensional range select-project 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 determines 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.

A. Components

The main components of the CoopSC architecture, utilizing the P2P overlay functionality, are illustrated in Figure 1.

The Query Rewriter determines the probe, which is the part of the remote probes, and remainder sub-queries. In the first step, the query is available in the local cache. This is accomplished by making 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 part of the query that 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.

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 C.

The Database Executor (cf. Figure 1) execute queries on database server and returns result sets, while the Peer Executor 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.

B. Interactions

The interactions between the CoopSC components are also illustrated in Figure 1. The Query Executor 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 Executor (4) or to Database Executor (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).

C. Distributed Index

This section describes the distributed structure that is used for indexing semantic regions. Only double 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 double attribute selection (Example: 10 < lat and lat < 20 or 20 < long and long < 30). Queries are also double attribute selections (Example: select * from earthquakes where 10 < lat and lat < 20 or 20 < long and long < 30). Double attribute selection predicates can be represented as sets of non-overlapping axis-aligned rectangles (Example: {(10, 10, 20, 30), (40, 50, 80, 90)}). Rectangles are represent with the coordinates of their top-left and bottom-right corners. 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 names of selection attributes are discarded. Thus, a semantic region is represented as a triple that contains the address of the client, the local identifier and the set of rectangles (Example: R = (192.168.100.40, 1, {[10, 30, 40, 50], (100, 110, 124, 129)])[j]) while a query is represent as a set of rectangles (Example: Q = {{120, 110, 180, 190}})

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 distributed rewriting of query Q.

Example:

Consider the following four semantic regions:

- R1=(192.168.0.100, 1, {[0, 0, 75, 26], (76, 90, 110, 110)}),
- R2=(192.169.0.202, 2, {[10, 70, 35, 85]}),
- R3=(192.168.20.23, 4, {[21, 32, 58, 80]}),
- R4=(192.168.0.100, 2, {[21, 32, 58, 80]}),
and a query \( Q=\{(10, 15, 50, 45)\} \). The semantic regions \( R_1 \) and \( R_3 \) minimize the portion of query \( Q \) not covered.

2) Solution

The 2-dimensional space is divided into equal size rectangular boxes. Each box is specified by the system administrator. Each box is associated with a member of the P2P overlay. The association between boxes and peers is done by applying a hash function on boxes’ coordinates and selecting, for each box, the peer that has the closest ID to the hash value. Semantic regions are indexed in the nodes associated with the boxes they intersect. In order to determine the rewriting of a query, it is first determined the boxes that intersect the specify query. The rewriting is done independently for each box by scanning thought the regions within that box.

Figure 2 illustrates the distributed index for the examples presented at the beginning of the section. The size of boxes is (20, 30). Regions are indexed in the boxes they intersect. The query is illustrated as a blue rectangle. Its rewriting is performed by accessing the six boxes it intersects.

Special consideration must be taken when choosing the size of boxes. A smaller box size will increase the size of the distributed index, while larger boxes will increase the number of regions associated with each box which can negatively impact performance.

The distributed index can be adapted to n-dimensional selections by dividing the n-dimensional space into n-dimensional equal size boxes. For single attribute selections, boxes are reduced to intervals.

D. Update Queries

When the content of the database is changed, modifications must be reflected in the cooperative cache. This can be accomplished with a cooperation from the database server. An active database server component was developed in order to handle the execution of update, insert, and delete SQL statements using triggers. This component uses the same division into boxes as the distributed index which was presented in the previous section. For each such box, the database server stores a virtual timestamp which is initialized with 0. These timestamps are incremented when modification are performed to tuples pertaining to particular boxes. For example, when a new tuple with values (20, 10) for latitude and longitude is inserted, the virtual timestamps of the box situated upper-left corner is incremented (c.f. Figure 2).

Cache entries store the virtual timestamps of the boxes they intersect. For each such box, the database server stores a virtual timestamp which is initialized with 0. These timestamps are incremented when modification are performed to tuples pertaining to particular boxes. For example, when a new tuple with values (20, 10) for latitude and longitude is inserted, the virtual timestamps of the box situated upper-left corner is incremented (c.f. Figure 2).

Before rewriting a new query, client asks the database server for the virtual timestamps of the boxes that intersect the query. The rewriting process will not use entries for which some virtual timestamps are older than the ones returned by server. If such entries are found, they are also discarded in order to save storage space.

On one hand, an advantage of this approach is that queries results are always up-to-date. On the other hand, this solution can discard entries that are still valid. A modification performed on a single tuple, which pertain to a particular box causes the invalidation of all entries which intersect that box. Decreasing the size of boxes can reduce the number of valid entries which are discarded.

V. IMPLEMENTATION

A prototype C++ implementation of the CoopSC architecture was developed. The implementation works with the PostgreSQL database and uses the Chimera P2P overlay [14]. 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++.

The UML (Unified Modeling Language) diagram of the CoopSC implementation is illustrated in Figure 3.

The Query Executor component is implemented using the classes \texttt{coopsc::QueryRewriter} and \texttt{coopsc::PeerServer}. The class \texttt{coopsc::QueryRewriter} implements the query rewriting algorithm, as described is Section III.
clients first ask for the virtual timestamps of boxes they intersect. The server and all clients are located in the same LAN (Local Area Network). The PostgreSQL database server is used with its default configuration. Upon interpreting the results below with respect to the amount of data sent by the database server, it must be noted that PostgreSQL uses SSL (Secure Sockets Layer) for communicating with clients, which also compresses data sent. Thus, a set of similar queries was 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 and the amount of data sent by the server were measured.

The evaluation was done using the Wisconsin benchmark [1] relation 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). Similarly with the evaluation of other cache architectures [5], [6], queries executed by each client have a semantic locality. For each client, the centerpoints of queries were randomly chosen to follow a normal distribution curve with a particular standard deviation. For each experiment, 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 the end, each client executes 50 queries for which the response time is not measured. 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 client’s caches are full before starting these measurements. The post-queries are executed in order to make sure that the database server load remains constant while measuring the response times.

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. For each scenario, the total amount of data sent by database server is also measured.

The first experiment determines how varying the number of clients influences the performance of the three scenarios. The number of clients are varied from 2 to 40. When 40 clients are used the response time for the no-caching scenario becomes unreasonable high (more than 10 seconds) and the evaluation is stopped. The size of clients’ caches is 64 MB. The workloads have standard deviations of 150,000. The means of the gaussian curves are distributed uniformly over the range of the unique1 attribute. The difference between the means of two consecutive clients is 1,000 to 20,000 tuples. The experiment uses 10 clients. The workloads have standard deviations of 250,000. The difference between the gaussian means of two consecutive clients is 300,000. Key results of this experiment are presented in Figure 4 and Figure 5. As it can be seen, both the response time and the amount of data sent by server when using the cooperative caching approach are 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 and the amount of data sent increase. 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 clients. This decreases the average response time and the amount of data sent. When using CoopSC, cache entries are shared between clients. This causes a further decrease of the average response time and of the amount of data sent, because the hit rate of the cache system increases.

The next experiment measures how the size of selections influences the performance. The size of selections is varied from 1,000 to 20,000 tuples. The experiment uses 10 clients. The workloads have standard deviations of 250,000. The difference between the gaussian means of two consecutive clients is 300,000. Key results of this experiment are presented in Figure 6 and Figure 7. As the size of selections increases, the response times and the amount of data sent by database server also increase. Similarly with the first experiment, the semantic caching approach is more efficient than the no-caching approach because database server only handles parts of queries which can not be answered using the cache. The cooperative caching solution outperforms the local caching approach due to the increase of hit rate of the cache system.
The last experiment measures how the variation of the size of clients’ caches influences the performance of the two caching approaches. The size of clients’ caches are varied from 0 to 256 MB. The experiment uses 10 clients. The workloads are generated in the same way as in the previous experiment. The results are illustrated in Figure 8 and Figure 9. For small cache sizes, the difference between the two approaches is reduced, because hit rates are small in both scenarios and database server has to handle executions of most queries. While the cache sizes increase, the benefits of the cooperative caching approach become more visible, both in respect to query response times and also to amount of data sent by database server. For large cache sizes, the difference becomes again reduced, because a large part of queries can be answered completely by accessing only the local cache and, thus, in many situation the cooperative cache is not needed.

![Figure 8: Cache size - response time](image)

![Figure 9: Cache size - transferred data](image)

VII. CONCLUSIONS AND FUTURE WORK

The CoopSC approach determines 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 manner. CoopSC supports n-dimensional range select-project queries. Update queries are also handled. The key components of the CoopSC architecture were described, major details outlined, and implemented. The proposed architecture was evaluated and compared with the classic semantic caching approach. These evaluation results show that CoopSC, especially by applying distributed principles and the P2P overlay techniques in particular, reduces the response time of range selection queries and the amount of data sent by database server.

Thus, the CoopSC approach shows that using a cooperative semantic caching approach can increase the performance of database systems by reducing queries’ response time and the amount of data sent by a database server. This makes the approach suited both in (a) a higher-bandwidth local-area environments where server’s processing resources (CPU, disk access) are the bottleneck of the system and (b) in network-constrained environments.

Further experiments will investigate how the semantic locality of these queries affects the performance of the CoopSC system. Evaluations for multi-dimensional selections and for update workloads are also planned for being performed in the distributed P2P overlay environment. Furthermore, the CoopSC update handling mechanism will be compared with other existing approaches.

REFERENCES

10. Q. Ren, M.H. Dunham: Using semantic caching to manage locationwe dependent data in mobile computing, Annual International Conference on Mobile Computing and Networking, Boston, Massachusetts, USA, August, 2000