A Scalable Cooperative Semantic Caching (CoopSC) Approach to Improve Range Queries

Andrei Vancea*, Laurent d'Orazio†, Burkhard Stiller*
* Department of Informatics (IFI), University of Zürich, Zürich, Switzerland
† Blaise Pascal University - LIMOS, France
Email: {vancea, stiller}@ifi.uzh.ch, laurent.dorazio@isima.fr

***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 approach, 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. The approach has been validated and experiments show that CoopSC improves the performance of range queries.**

***Index Terms*—Cooperation in P2P, Semantic Caching, Data Base, P2P Application, Implementation.**

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 [5]. Semantic caching [8] 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 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 [16]. 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.

Peer-to-peer (P2P) networks have been applied successfully for enhancing beyond the traditional client-server communication, thus, they are applicable to the distribution problem outlined. E.g., the CoopNet [15], uses a cooperative network caching architecture for solving Web flash crowd scalability problems. These results show that a cooperative P2P-based caching approach significantly increase the performance of client-server architectures under heavy load.

Like for most existing database cache architectures [8], the major aim of CoopSC is the enhancement of the performance of read-intensive query workloads. Such types of workloads are frequently used in many type of applications, including decision-support systems. Select-project queries, where the predicate is a n-dimensional range condition, are commonly used when queries dimensional data (e.g., geographic information). Thus, again, the real-life case is considered with a high priority. Furthermore, with the emergence of cloud computing infrastructures, using a cooperative database caching approach can have economic advantages, because cloud providers usually bill data transferred between cloud environment and the outside world. Thus, the minimization of amount of data sent by database server can achieve such a cost reduction.

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.

The following examples do illustrate suitable applications 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 C_1

COLLABORATECOM 2011, October 15-18, Orlando, United States
Copyright © 2012 ICST
DOI 10.4108/icst.collaboratecom.2011.247093

asks for the events the happened in the area between (20, 20) and (40, 40) (Q_1: 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 C_2 asks for the earthquakes that happened in the area between (30, 30) and (50, 50) (Q_2: 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, Q_2 will be split in a remote probe, which will be sent to C_1, 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 tuples from the server (select * from earthquakes where $39 < lat$ and $lat < 50$ and $30 < long$ and $long < 50$ or $30 < lat$ and $lat < 40$ and $39 < long$ and $long < 50$).

Using the CoopSC approach within a cloud-computing infrastructure presents economic advantages because most cloud providers (e.g., Amazon EC2, Rackspace) bill data transferred between cloud environment and outside world. Two scenarios are considered (Figure 1): a) an operational database is running within a cloud environment while clients are running outside. b) several nodes run inside a cloud environment in order to performed specific tasks which use data that originate from a database which is running outside the cloud. In both scenarios, using the CoopSC approach reduces the amount of data sent by database server and thus, reduces the amount of money that has to be paid for data transfer.

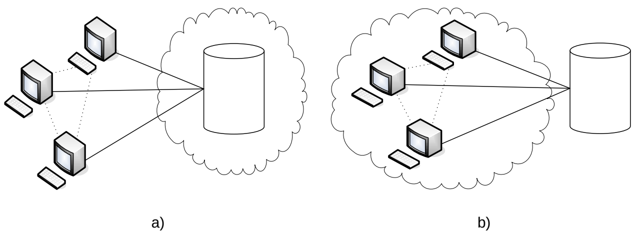


Fig. 1. Cloud Computing Scenarios

Therefore, the Cooperative Semantic Caching (CoopSC) approach extends the general semantic caching mechanism by using a P2P approach in order to enable 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. Solutions must be provided for allowing clients to efficiently discover and use cache entries stored by other peers. Handing update statements is another important issue that CoopSC must tackle.

This paper is organized as follows: While Section II discusses related work, Section III outlines the design of CoopSC approach. The evaluation of the approach is presented in Section IV. Section V discusses some issues related to administration of CoopSC deployments and types of workloads that benefit from using it. Finally, some concluding remarks are given in Section VI.

II. Related Work

Client side caching is a commonly used technique for reducing the response time of database queries [5]. Classic client-side caching approaches include page and tuple caching. When using page caching, clients cache pages of fixed size. Queries are processed on client side down to the level of page access. If a particular page is not found in local page, a request is sent to database server and the missing page is transferred. The page caching system is implemented using mechanisms which are similar with the one used in the implementation of page-based database buffer managers. When tuple caching is used, clients cache individual tuples (or objects). This approach offers maximum flexibility, but it can suffer from performance problems caused by sending a large number of small messages.

The semantic caching approach which, was introduced in [8] as the basic concept, caches results of old queries and allows these results to be used for answering new queries. This paper describes semantic caching concepts and compares the approach with page and tuple caching. The cache is organized into disjoint semantic regions. Each semantic region contains a set of tuples and a constraint formula, which describes the common property of the tuples. Simulations were performed for single and double attribute selection queries. These simulations show that semantic caching outperforms both tuple and page caching. [12] runs an extensive performance study of a semantic caching prototype implementation for range queries up to four attributes. Experiments were performed using the Wisconsin benchmark [4] data set, show that semantic caching decreases both the response time and the amount of data sent by database server for one and two-dimensions selection queries, while for queries with higher dimensions the decrease is only significant in regards to the amount of data sent by server. However, the classic semantic caching approach - as referred to in [8] and [12] - does not handle update queries.

The predicate caching approach, presented in [13], caches locally results of old queries together with predicates that describe cache's content. A subsequent query may be answered from local cache if it can be determine that its results are totally contained in the local cache. The approach supports select-project-join queries. Because storing and processing exact cache predicate descriptions might be computational expensive, the predicate caching solutions keeps only a conservative approximative cache description, which guarantees that data thought to be in the cache is present in it. Furthermore, compared with semantic caching [8], predicate caching approach stores cache entries that do not necessary have to be disjoint. On one hand, duplicate data could be stored in different cache entries, which might negatively influence the performance of the caching system. On the other hand, making sure the all cache entries are disjoint might be timewise expensive, especially when more complex select-project-join queries types are supported. Predicate caching also supports update statements. Modifications are initially performed locally and only sent to database server when new local queries can not be computed locally and are sent to servers. When conflicts occur, the approach provides mechanism for canceling transactions.

XCache [6] 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 [6].

Furthermore, these three approaches described in [8], [13] and [6] 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 [7] caches old results of database queries in order to answer new queries and to allow for the cached entries to be shared between clients. Wigan supports only queries that can be expressed as conjunctions of single attribute range conditions. 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. Another drawback of this approach is that is uses a centralized tracker in order to determine, which cached entries 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. This can be avoided in a fully decentralized approach. Furthermore, Wigan does not handle update queries, too.

[14] describes a cooperative caching architecture for answering XPath queries with no predicates. 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 this 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.

The Dual Cache approach [10] is a caching service built on top of the Gedeon data management system [9]. The system performs 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 system allows new types of cache resolution to be added. In order to overcome the scalability issues of flooding, client are divided into communities. Thus, only clients that are in the same community can cooperate. Dual Cache handles non-range predicates 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 I illustrates the key differences between the cooperative semantic caching approaches investigated as related work as well as outlining already for comparing dimensions the new CoopSC approach, which will be developed within this project.

TABLE I
COOPERATIVE SEMANTIC CACHING APPROACHES

Approach	Data Model	Query Types	Hit Types	Resolution	Update
Wigan [7]	Relational	Simple range selections	Strict	Centralized tracker	No
XPath Index-Cache [14]	XML	XPath (no predicates)	Strict	Distributed Index	No
Dual Cache [10]	Gedeon	Non-range queries	Strict	Flooding	No
CoopSC [20]	Relational	Range select-project queries	Strict, Partial	Distributed Index	Yes

Other research projects aim at the provisioning of cooperative caching facilities in Web environments. For example, [15] presents CoopNet, a cooperative network architecture, where clients cooperate in order to improve the overall network performance. It is described how CoopNet is used for solving Web flash crowd scalability problems. In this approach, clients that have already downloaded Web content, start serving the content to other clients, relieving the server of this task. The redirection of requests from the server to other clients is handled by a centralized component running at the server side. Thus, this approach does not integrate the distribution aspect.

Squirrel [11] is a decentralized, P2P Web caching system. It enables Web browsers to share their local caches in a scalable matter. All Web clients are a part of a P2P overlay based on the Pastry [17] system. Each URL (Uniform Resource Locator) is associated with a node from the P2P overlay, which is called the home node. This association is done by applying a hash function on the URL and choosing the node with the closest ID to the hash value. Two approaches are implemented: Home-Store and Directory. When the Home-Store approach is used, Squirrel stores objects both at client caches and at its home node. In the Directory approach, home nodes only store the IDs of other existing nodes the have the relevant content.

Thus, existing cooperative semantic caching systems lack the support of complex query types. There are no approaches in place, which 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 will be applied to both classic and cooperative semantic caching approaches. Compared with the classic materialized views solutions, query rewriting and handling update statements is the context of cooperative semantic caching presents many aditional scalability challenges which the CoopSC project solve. The CoopSC projects solves these challenges in a distributed environment as mentioned above, while the CoopSC's very basic idea has been published in [20].

III. Design of The CoopSC Approach

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 *probe*, *remote probe*, and *remainder* sub-queries using a *query rewriting* process. 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 tuples from the server.

In order to execute the query rewriting, 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.

CoopSC handles the execution of n-dimensional select-project queries. Similarly with the approach presented in [8], the local cache is organized into disjoint *semantic regions*. A semantic region is defined as a set of tuples and a constrained formula which determines the common property of the tuples. Clients interrogating a specific database server form the P2P overlay network, which is used for indexing the semantic regions.

Semantic regions are stored, in-memory, by the Cache Manager [20]. The Cache Manager also implements a replacement policy which is orthogonal to this proposal. Each client manages its local cache entries independently. As a result, popular entries will be automatically replicated to multiple clients and thus the scalability of this approach is increased.

Figure 2 describes the main structures which are used for representing semantic regions and queries. The $SemanticRegion$ structure contains an unique identifier, the name of the table, the set of fields, the predicate which describes it, and the set of tuples which determines the content of region. A *query* is defined by the name of table, the set of fields and the predicate.

Based of structures presented in Figure 2, is what follows the concepts of intersection and subsumption between queries and regions will be clearly defined. These concepts will be used in the later parts of this section.

```
Structures
  struct $SemanticRegion$
      $id$ : int
      $table$ : string
      $fields$ : FieldsSet
      $pred$ : Predicate
      $ntuples$ : NTuples
  struct $Query$
      $table$ : string
      $fields$ : FieldsSet
      $pred$ : Predicate
```

Fig. 2. CoopSC Structures

Definition 1: Let r be a semantic region and q a query. r and q are said to *vertically intersect* if $r.table = q.table$ and $r.fields \cap q.fields \neq \emptyset$.

Example 1: Let:

r_1 = (10, wisconsin, {unique1, unique2, two}, $unique1 < 10$,{...})

r_2 = (11, wisconsin, {unique2, four}, $unique1 < 10$, {...})

two semantic regions and

q = (wisconsin, {unique1, two}, $unique1 < 40$)

a query.

r_1 and q *vertically intersect*, while r_2 and q do not because their fields do not intersect.

Definition 2: Let r be a semantic region and q a query. r and q are said to *horizontally intersect* if $r.table = q.table$ and the predicate $q.pred \wedge r.pred$ is satisfiable.

Example 2: Let:

r_1 = (10, wisconsin, {unique1, unique2}, $unique1 < 10$, {...})

r_2 = (11, wisconsin, {unique2, unique2}, $unique1 > 100$, {...})

two semantic regions and

q = (wisconsin, {unique1, two}, $unique1 < 40$)

a query.

r_1 and q *horizontally intersect*, while r_2 and q do not because the predicate $(unique1 > 100) \wedge (unique1 < 40)$ is not satisfiable.

Definition 3: Let r be a semantic region and q a query. r and q are said to *intersect* if they *vertically intersect* and *horizontally intersect*.

If a semantic region R intersects a given query Q, it can be used for partially answering Q. Query will be split in a sub-query that can be answered using R and sub-queries for which R does not provide relevant tuples.

Definition 4: Let r_1 and r_2 be two semantic regions. r_1 and r_2 are said to be *disjoint* if $r_1.table \neq r_2.table$ or $r_1.fields \cap r_2.fields = \emptyset$ or the predicate $r_1.pred \wedge r_2.pred$ is not satisfiable.

Example 3: Let:

r_1 = (10, wisconsin, {unique1, unique2, two, four}, $unique1 < 10$, {...})

r_2 = (11, wisconsin, {unique1, unique2, two}, $unique1 > 100$, {...}

r_3 = (12, wisconsin, {unique2, four}, $unique1 > 0$, {...}

three semantic regions.

r_1 and r_2 are disjoint, while r_1 and r_3 are not because $(unique1 < 10) \wedge (unique1 > 0)$ is satisfiable.

Definition 5: Let r be a semantic region and q a query. It is said that r *vertically subsumes* q if $r.table = q.table$ and $r.fields \supseteq q.fields$.

Example 4: Let:

r_1 = (10, wisconsin, {unique1, unique2, two, four}, $unique1 < 10$, $\{\dots\}$)

r_2 = (11, wisconsin, {unique1, unique2, two}, $unique1 < 10$, $\{\dots\}$

two semantic regions and

q = (wisconsin, {unique1, four}, $unique1 < 40$)

a query.

r_1 *vertically subsumes* q, while r_2 does not because $r_2.fields \not\supseteq q.fields$.

Definition 6: Let r be a semantic region and q a query. It is said that r *horizontally subsumes* q if $r.table = q.table$ and $q.pred \Rightarrow r.pred$.

Example 5: Let:

r_1 = (10, wisconsin, {unique1, unique2}, $unique1 < 100$, $\{\dots\}$)

r_2 = (11, wisconsin, {unique2, unique2}, $unique1 < 50$, $\{\dots\}$)

two semantic regions and

q = (wisconsin, {unique1, two}, $unique1 < 70$)

a query.

r_1 *horizontally subsumes* q, while r_2 does not because $(unique1 < 70) \nRightarrow (unique1 < 50)$.

Definition 7: Let r be a semantic region and q a query. It is said that r *subsumes* q if r *vertically subsumes* q and r *horizontally subsumes* q.

If region R subsumes query Q, Q can be answered completely using the content of R. Thus, subsumption is a much stronger condition than intersection.

A. Query Rewriting

The query rewriting process (illustrated in Figure 3) determines parts of a given query that can be answered using local cache (*probe*), caches of other clients (*remote probe*) or database server (*remainder*) and the way in which they are combined in order to return the final query result. This process is executed by a component, running on client side, called *Query Rewriter*. The result of query rewriting process is a *query plan tree*, which describes how query is to be executed. Initially, the query rewriting checks entries stored in local cache (*Local Rewriting*). Afterwards, the distributed index is interrogated in order to determine remote cache entries which can be used for answering given query (*Distributed Rewriting*).

This section will, first, describe the structure of *query plan trees*. Afterwards, the local and distributed rewriting process will be presented.

1) Query Plan Tree: As mentioned, the result of query rewriting process is a *query plan tree* (exemplified in Figures 5 and 6). Its leafs refer semantic regions (stored locally or remotely) or sub-query which are to be executed by database server.

A *query plan tree* contains types of nodes for executing union and join operations, selecting tuples from local cache entries (*SelectProject*), returning the content of specified region (*Region*), executing given query on server(*Remainder*) and returning result of a query plan tree executed on a different CoopSC client (*Remote*).

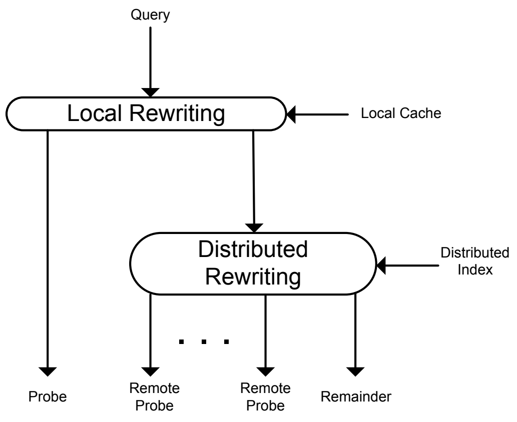


Fig. 3. Query Rewriting

2) Local Rewriting: The *Local Rewriting* process scans local cache and determines which semantic regions can be used for answering a given query. The result of local rewriting is an initial *query plan tree* which only contains references to local cache or database server (exemplified in Figure 5).

Each region is compared with given query. Figure 4 illustrates the possible relations between given query and current semantic region:

a) region and query do not intersect; current region is not used for answering query.
b) region subsumes query; query can be completely answered using current region.
c) region vertically subsumes query and horizontally intersects it; query is split into two sub-queries; one can be answered using current region, while for the other the local rewriting will continue using the following regions.
d) region horizontally subsumes query and vertically intersects it; query is split into two sub-queries; one can be answered using current region, while for the other the local rewriting will continue using the following regions.
e) region both horizontally and vertically intersects query; query is split into three sub-queries; one can be answered using current region, while for the other two the local rewriting will continue using the following regions.

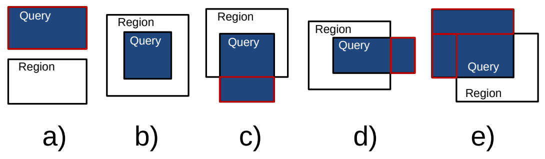
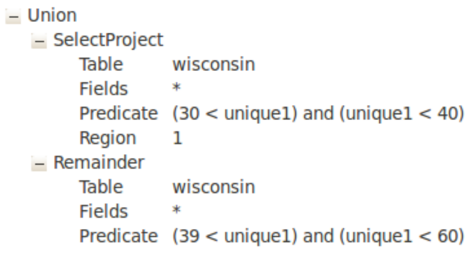


Fig. 4. Region/Query Overlapping

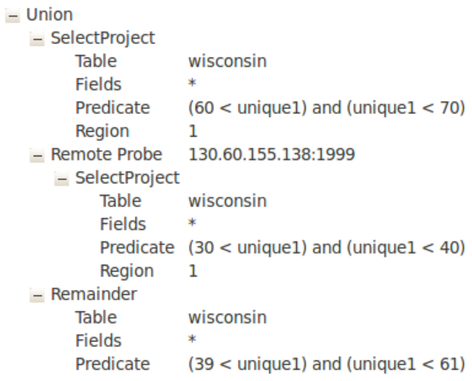
3) Distributed Rewriting: As illustrated in Figure 3, *distributed rewriting* uses the distributed index in order to determine which remote semantic regions can be used for answering given query. The *query plan tree*, generated during *local rewriting* is modifying by replacing $Remainder$ nodes



```
- Union
  - SelectProject
      Table       wisconsin
      Fields      *
      Predicate   (30 < unique1) and (unique1 < 40)
      Region      1
  - Remainder
      Table       wisconsin
      Fields      *
      Predicate   (39 < unique1) and (unique1 < 60)
```

Fig. 5. Local Query Plan Tree

with results of interrogations sent to *distributed index*. These results can refers semantic regions stored by other clients. Figure 6 illustrates a possible result of distributed rewriting.



```
- Union
  - SelectProject
      Table          wisconsin
      Fields         *
      Predicate      (60 < unique1) and (unique1 < 70)
      Region         1
  - Remote Probe     130.60.155.138:1999
    - SelectProject
        Table        wisconsin
        Fields       *
        Predicate    (30 < unique1) and (unique1 < 40)
        Region       1
  - Remainder
      Table          wisconsin
      Fields         *
      Predicate      (39 < unique1) and (unique1 < 61)
```

Fig. 6. Distributed Query Plan Tree

B. 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 tuples and a predicate. Under the given assumptions, the predicate is a double attribute selection (Example: $10 < lat$ and $lat < 20$ and $20 < long$ and $long < 30$). Queries are also double attribute selections (Example: select * from earthquakes where $10 < lat$ and $lat < 20$ and $20 < long$ 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 represented with the coordinates of their top-left and bottom-right corners. This representation will be used for both semantic regions and queries.

The distributed index must be able to index semantic regions. Removing regions from index shall also be supported. Furthermore, given a query Q, the distributed index must return a *query plan tree* that contains references to semantic regions stored in different CoopSC clients and minimizes the part of query which is answered by database server.

1) P2P-based Distributed Index: The distributed index is based on the P2P index described in [19], which adapts the classic MX-CIF quad trees [18] in order to be stored on top of a P2P overlay. CoopSC tailors and implements this approach for efficiently supporting distributed query rewriting.

The two-dimensional square-based area is, recursively, divided into four equal-sized square blocks until a given *fundamental maximum level*, f_{max} is reached. Each square is associated with a node from the P2P overlay. The association between squares and peers is done by applying a hash function on coordinates of squares' center points and selecting, for each square, the peer that has the closest ID to the hash value. Each semantic region is indexed in the square of *minimum size* (thus, maximum level) that contains given region.

Figure 7 exemplifies the quad tree space division and the way in which four give semantic regions are indexed. Regions R_1 and R_4 are indexed in the root node, since they are not contain in any child square. R_2 is totally contained in the top-right level 2 sub-square, thus is indexed in the corresponding level 2 node. R_3 is indexed in a level 3 node.

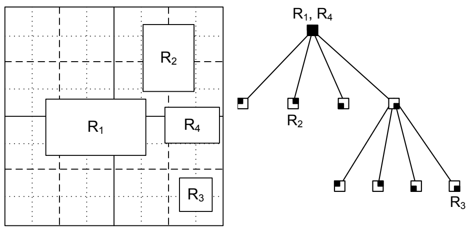


Fig. 7. MX-CIF Quad Tree Example

As described in [19], for performance and reliability reasons, the root node is not stored in the P2P overlay. Only nodes at a level higher or equal to a given *fundamental minimum level*, f_{min} are considered. Thus, when implemented distributively on top of a P2P overlay MX-CIF quad trees are transformed into *forests*. It is assumed that f_{min} is chosen in such a way that every semantic regions is contained in a square associated with a node of level f_{min}.

Figure 8 and 9 present the pseudo-codes for adding and removing semantic region to/from distributed index. As mentioned, each region is added to the quad node of maximum level that contains it. Methods $sendAddRegion$ and $sendRemoveRegion$ route request through the P2P overlay until the node associated with the specified *Quad* is reached.

Figure 10 contains the pseudo-code for the algorithm that handles rewriting requests within distributed index. An initial query rewriting, similar with local rewriting, is performed using indexed semantic regions. Afterwards, the rewriting process continues with children quad nodes.

The distributed index can be adapted to n-dimensional selections by dividing the n-dimensional space into 2^n equal size quads. For single attribute selections, quads are reduced to intervals.

C. Updates

When the content of the database is changed, modifications must be reflected in the cooperative cache. Handling updating

```
ADD-REGION($QN : QuadNode, R : Region$)
    for $child : QN.children()$
        do if $child.contains(R)$
            then $sendAddRegion(child, R)$
                return
    $QN.regions.add(R)$
```

Fig. 8. Distributed Index: Add Region

```
REMOVE-REGION($QN : QuadNode, R : Region$)
    for $child : QN.children()$
        do if $!child.empty() \wedge child.contains(R)$
            then $sendRemoveRegion(child, R)$
                return
    $QN.regions.remove(R)$
```

Fig. 9. Distributed Index: Remove Region

efficiently presents the following challenging issues: a) not all modifications are generated directly by clients; database server can have active components which perform changes as result of different events; b) the update mechanism must avoid combining region that pertain to different database snapshots.

The following example illustrates a scenario when combining regions that originate from different snapshots causes inconsistencies: client A executes Q_1: "select * from persons where $20 < age$ and $age < 40$" and caches its result in region R_1. Afterwards, client B updates the age of a person from 25 to 50. Finally, A execute Q_2: "select * from persons where $20 < age$ and $age < 60$" which is split in a probe which returns region R_1 and a remainder which executes "select * from persons where $39 < age$ and $age < 60$" on server-side and returns result. The updated person will be present both in region R_1 and also in the remainder, because the modification was performed after the execution of Q_1 and thus, the final result will be inconsistent.

CoopSC handles updates 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 quad space division as the distributed index which was presented in the previous section. For each quad from a given *fundamental update level*, f_{update} ($f_{min} \leq f_{update} \leq f_{max}$), database server stores a virtual timestamp which is initialized with 0. These timestamps are incremented when modification are performed to tuples pertaining to particular quads. Semantic regions are augmented with virtual timestamps of quads they intersect at the moment of retrieval from database. Referring to the example from figure 7 and assuming that $t_{update} = 2$, R_1 will store four timestamps values, R_2 one timestamp, R_3 also one, and R_4 two timestamps.

Before rewriting a new query, client asks database server for the virtual timestamps of the quads that intersect given query. The rewriting process will not use entries for which

```
REWRITE($QN : QuadNode, Q : Query$)
    $n \leftarrow rewrite(Q, QN.regions)$
    for $r : n.remainders()$
        do $result \leftarrow \emptyset$
            for $c : QN.children()$
                do if $c.intersects(r.query)$
                    then
                        $m \leftarrow sendRewrite(c, r.query)$
                        $results.add(m)$
            $r \leftarrow Union(results)$
    return $n$
```

Fig. 10. Distributed Index: Rewrite

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. These timestamps are also used during distributed rewriting in order to only consider up-to-date remote semantic regions and to discard old ones.

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 quad causes invalidation of all entries which intersect that quad. Increasing the fundamental update level can reduce the number of valid entries which are discarded, but, in the same it also enlarges the number of virtual timestamps stored in distributed index and regions.

Figure 11 contains the pseudo-code of the update mechanism within the general query execution algorithm. Initially, quads of level f_{update} which intersect given query are determined. The timestamps of these squares are then returned from database server. The query rewriting process is then executed and a query plan tree returned. The query plan tree is executed and a result returned. Afterwards, the timestamps values of query's squares are determined again and compare with the original values. If differences are noticed, the result that uses the cache is discarded, because at least a modification occurred during query rewriting or query plan execution and system can not guarantee that result contains only tuples from a single database snapshot.

```
EXECUTE($Q : Query$)
    $quads \leftarrow query.getIntersectedSquares(f_{update})$
    $before \leftarrow database.getTimestamps(quads)$
    $plan \leftarrow rewrite(Q, before)$
    $result \leftarrow plan.execute()$
    $after \leftarrow database.getTimestamps(quads)$
    if $before \neq after$
        then return $database.execute(Q)$
        else return $result$;
```

Fig. 11. Update Handling

IV. Evaluation

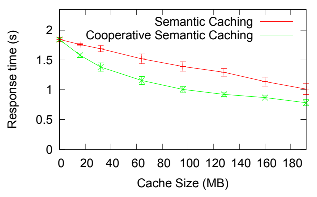
The CoopSC approach was implemented and evaluated using a PostgreSQL database server and a number of clients that execute, in parallel, single and double indexed attribute selection queries. Updates statements were also evaluated. The EmanicsLab research testing network was used for this evaluation. Clients are running in 10 nodes located across Europe, while the database server runs on a more powerful machine located in Zurich. During the evaluation, three scenarios were used: cooperative semantic caching approach, classic semantic caching and no caching approach.

The evaluation was done using the Wisconsin benchmark [4] relation of 10 million tuples, where each tuple contains 208 bytes of data. Each query is a range selection on either the *unique1* attribute (Example: select * from wisconsin where 4813305 < unique1 and unique1 < 4823306) or on *unique1* and *unique2* (Example: select * from wisconsin where 4813305 < unique1 and unique1 < 4823306 and 23000 < unique1 and unique1 < 33000). 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 warm-up queries until cache is filled. The response time, for each client, is calculated by averaging the response time of 10 testing sessions of 50 queries each. The error bar is calculated using these 10 values. For each scenario, the total amount of data sent by database server is also measured. Furthermore, for the cooperative caching scenario, in each session, numbers of tuples that originated from the local cache, remote caches and the database server are determined. *Local* and *peers* hit rates are computed for the cooperative caching scenario. The local hit rate is defined as the percent of tuples from result sets that originated from the local cache, while peers hit rate refers to tuples that were returned from caches of the other clients.

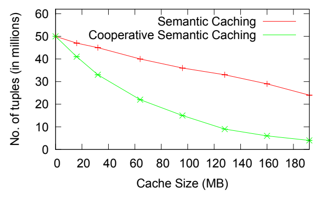
Thus, in each experiment, four measurements are made: the first two compare the cooperative semantic caching approach with classic semantic caching and no caching scenario, in the relation to (a) query response time and (b) amount of data sent by database server. The other two measurements refer only to the cooperative caching approach and determine (c) tuples' origin (d) and hit rates.

For single attribute selections experiments, the distributed index was configured with the following parameters $f_{min} = 10, f_{max} = 18, f_{update} = 15$. For the double attributes workload these paramenters were $f_{min} = 15, f_{max} = 20, f_{update} = 18$.

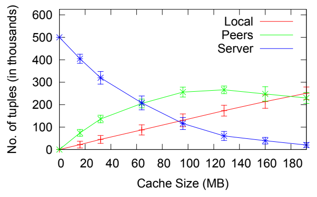
The first experiment measures how the variation of the size of clients' caches influences the performance of the two caching approaches for single-attribute selections. The size of clients' caches are varied from 0 to 192 MB. This experiment uses 10 clients. These workloads have standard deviations of 500,000. 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 300,000. Key results of this experiment are presented in Figure 12. Analyzing the response time (Figure 12a), for small cache sizes, the difference between the two approaches is reduced, because hit rates are small in both scenarios and the database server has to handle executions of most queries. While the cache sizes increase, the benefits of the cooperative caching approach become more visible. 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. In the semantic caching approach, the number of tuples sent by the database server is reduced, because the database server only sends parts of queries which are missing from the local cache. The cooperative approach further decreases the number of tuples sent because clients can also transfer tuples from caches of other peers. Graph 12c illustrates the origin of tuples for the cooperative caching approach. For small cache size, most tuples are returned from the database server. As cache size increases, both the number of tuples returned from the local cache and caches of other clients increase. For larger cache size, number of tuples returned from caches of other peers decreases because most queries can be answered using entries from the local cache and thus, cooperation is reduced. Hit rates have a similar behavior (Figure 12d). For small cache size both the local and peers hit rate are reduced. Increasing cache size causes the increase of both hit rates. The peers hit rate reaches a maximum, and then starts to decrease because the need of cooperation decreases.



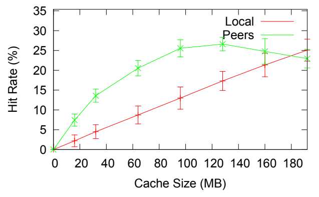
(a) Response Time



(b) Server Tuples



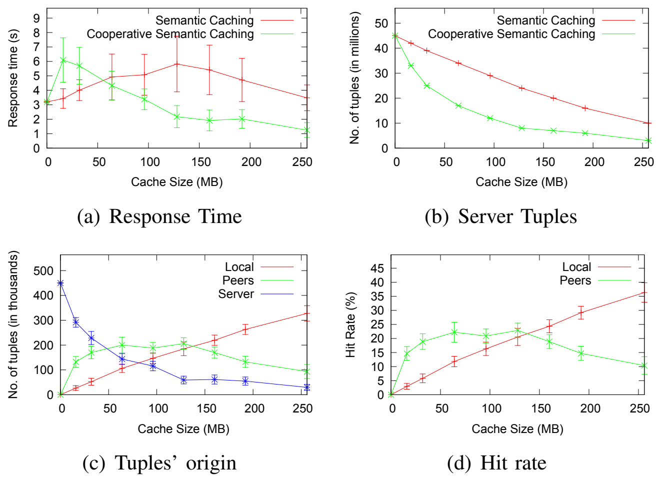
(c) Tuples' origin



(d) Hit rate

Fig. 12. Cache Size 1D

A similar experiment was executed for two dimensional selections. The workloads have standard deviations of 1,000,000 on both attributes. Queries are rectangles with sidelengths of 300,000 and return around 9,000 tuples. The results are illustrated in Figure 13. Except response time (Figure 13a), the other measurements are similar with the one from the single attribute selection. For two dimensional selections, the time-wise cost of query rewriting and accessing the distributed indexed is increased. For small cache size, due to the low peers hit rate, this cost overcomes the benefits of cooperation and thus, the cooperative caching approach performs worst than semantic caching. With the increase of cache size, the cooperative approach starts to outperform semantic caching because the hit rates increase which compensates for the cost of query rewriting.

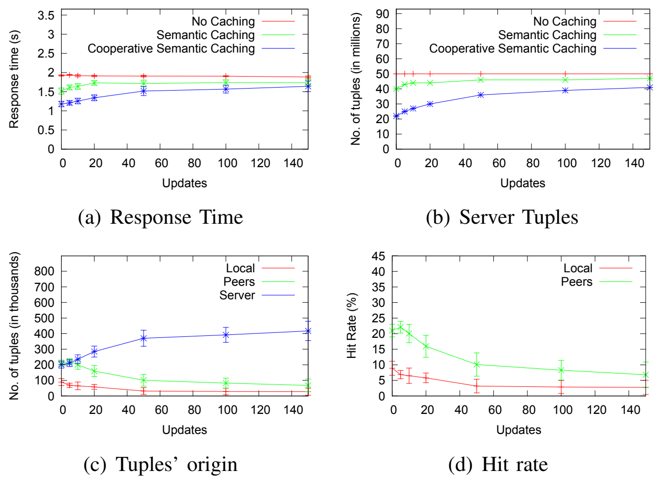


(a) Response Time (b) Server Tuples (c) Tuples' origin (d) Hit rate

Fig. 13. Cache Size 2D

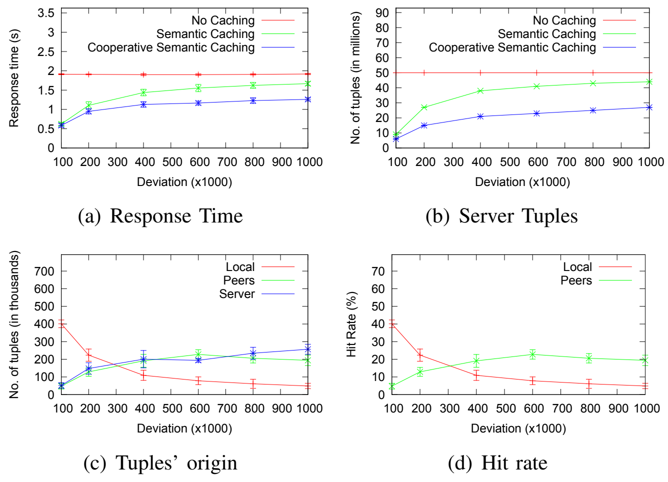
The third experiment investigates how update statements influence the performance of the two caching approaches. The size of clients' cache is 64 MB. The workload consists of a sequence of alternative selection and update sessions. Selection sessions are generated similarly with the first experiment. Update sessions contain a number of updates statements which modify a single tuple chosen randomly based on the normal distribution used for the selection sessions. The number of update statements per session is varied from 0 to 150. Figure 14 illustrates the results of this experiment. While the number of update statements per session increases, the performance of both caching systems starts to decrease because update statements invalidate an increasing number of cache entries. Thus, both the query response time (Figure 14a) and the number of tuples sent by database server (Figure 14b) increase. For the cooperative caching approach, increasing the number of update statements causes an increase of tuples that originate from the database server and decreases amount of data returned from either the local or remote caches (Figure 14c). Both the local and peers hit rate decrease with the increase of update statements due to the invalidation of cache entries (Figure 14d).

The last experiment measures how varying queries' locality influences the performance of both caching approaches. The size of clients' cache is 64 MB. The experiment uses 10 clients. The workloads' standard deviations are varied from 100,000 to 1,000,000. Figure 15 illustrates the results of this experiment. Increasing standard deviation of workloads decreases access locality and thus, the performance of both caching systems decreases (Figure 15a b). In the cooperative



(a) Response Time (b) Server Tuples (c) Tuples' origin (d) Hit rate

Fig. 14. Updates



(a) Response Time (b) Server Tuples (c) Tuples' origin (d) Hit rate

Fig. 15. Deviation

caching approach, increasing the standard deviation causes the number of tuples returned from the local cache to decrease, while the number of tuples returned from the server increases. The number of tuples returned from remote peers initially increases, because lowering access locality increases the need to cooperate. A further increase of standard deviation decreases also the amount of data returned from remote peers because with a general low access locality, relevant entries will not be found in other peers (Figure 15c). The local hit rate decreases with the increase of the standard deviation while peers hit rate initially increases and then decreases (Figure 15d).

V. Discussion

As presented in the Section III, the distributed index must be initially configured by specifying the f_{min} and f_{max} parameters. For this, it is assumed that system administrator has a priori knowledge about the type of workloads clients execute. These values must makes sure that the distributed tree structure has limited depth, in order to lower the communication costs, and, in the same time, it must determine a good distribution of indexed cache entries between nodes.

Assuring consistency represents an important issue when designing the updates mechanism of the CoopSC approach, because query results are determined by combining cache entries stored by clients and tuples returned from database server. As described in Section III-C, if these components pertain to different database snapshots, query results might be inconsistent. In order to overcome this problem, CoopSC sends update statements directly for execution to database server and stores, in a separate table, virtual timestamps. Thus, before every query execution, clients request virtual timestamps that intersect given query from database. Because the number of returned timestamps is much smaller than size of query result set, performance costs of returning these values is minimal. Each tuple modification increments the corresponding virtual timestamp and so, a write-intensive workload can be negatively influence by the caching approach. Thus, CoopSC, similarly with other caching architecture, is suited for application in which queries return a large number of tuples and modification are infrequent. Another advantage of this approach is that it also works in scenarios when not all updates originate from clients that use CoopSC.

While Section IV presents only the evaluation for single and double-attributes selections, the CoopSC approach and its implementation support generic n-dimensional queries. The performance of higher dimensions queries for the classic semantic caching approach is described in [12]. Due to increased time-wise cost of query rewriting, the performance benefits are only significant in regards to amount of data sent by database server while query response time can even increase. The rewriting process of the CoopSC approach is more complex, because it accesses the distributed index, and thus, the expected benefits of CoopSC for multi-dimensional workloads are limited.

Since CoopSC uses relation model, it can be easily adapted to be used with SQL-based cloud providers, such as Microsoft SQL Azure [3], Amazon RDS [1] or Hive [2]. Investigating the performance of CoopSC on top of these providers remains an interesting future work direction.

VI. Conclusions

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 matter. CoopSC supports n-dimensional range select-project queries. Update queries are also handled. The design of the CoopSC approach was described and major details outlined. The proposed approach 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 for read-intensive workloads. The benefits for workloads with a significant number of updates statements are limited due to the increased invalidation of cache entries.

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. Further experiments will investigate how cloud based solutions can benefit from using cooperative semantic caching approaches.

Acknowledgement

This work was supported by the CoopSC project funded by the Swiss National Science Foundations, Contract No. 200021-134679/1.

References

[1] Amazon rds. http://aws.amazon.com/rds.

[2] The hive project. http://wiki.apache.org/hadoop/Hive.

[3] Microsoft sql azure. http://www.microsoft.com/en-us/sqlazure/default.aspx.

[4] Dina Bitton and Carolyn Turbyfill. A retrospective on the wisconsin benchmark. *Readings in database systems*, 1988.

[5] Michael J. Carey, Michael J. Franklin, Miron Livny, and Eugene J. Shekita. Data caching tradeoffs in client-server dbms architectures. *SIGMOD Record*, 20(2), 1991.

[6] Li Chen, Elke A. Rundensteiner, and Song Wang. Xcache: a semantic caching system for xml queries. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 2002.

[7] Nicholas Coleman, Rajesh Raman, Miron Livny, and Marvin Solomon. A peer-to-peer database server based on bittorrent. Technical Report 10891, School of Computing Science, Newcastle University, 2008.

[8] Shaul Dar, Michael J. Franklin, Björn Þór Jónsson, Divesh Srivastava, and Michael Tan. Semantic data caching and replacement. In *Proceedings of the VLDB*, 1996.

[9] Yves Denneulin, Cyril Labbé, Laurent d'Orazio, and Claudia Roncancio. Merging file systems and data bases to fit the grid. In *Data Management in Grid and Peer-to-Peer Systmes*, 2010.

[10] Laurent d'Orazio and Mamadou Kaba Traoré. Semantic caching for pervasive grids. In *Proceedings of the International Database Engineering and Applications Symposium (IDEAS)*, 2009.

[11] Sitaram Iyer, Antony Rowstron, and Peter Druschel. Squirrel: a decentralized peer-to-peer web cache. In *Proceedings of the Annual Symposium on Principles of Distributed Computing (PODC)*, 2002.

[12] Björn Þór Jónsson, María Arinbjarnar, Bjarnsteinn Þórsson, Michael J. Franklin, and Divesh Srivastava. Performance and overhead of semantic cache management. *ACM Transactions on Internet Technology*, 6(3), 2006.

[13] Arthur M. Keller and Julie Basu. A predicate-based caching scheme for client-server database architectures. *The VLDB Journal*, 5, 1996.

[14] Kostas Lillis and Evaggelia Pitoura. Cooperative xpath caching. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 2008.

[15] Venkata N. Padmanabhan and Kunwadee Sripanidkulchai. The case for cooperative networking. In *Revised Papers from the First International Workshop on Peer-to-Peer Systems (IPTPS)*, 2002.

[16] Qun Ren and Margaret H. Dunham. Using semantic caching to manage location dependent data in mobile computing. In *Proceedings of the MobiCom*, 2000.

[17] Antony I. T. Rowstron and Peter Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)*, 2001.

[18] Hanan Samet. The quadtree and related hierarchical data structures. *ACM Computing Surveys*, 16, 1984.

[19] Egemen Tanin, Aaron Harwood, and Hanan Samet. Using a distributed quadtree index in peer-to-peer networks. *The VLDB Journal*, 16, 2007.

[20] Andrei Vancea and Burkhard Stiller. Coopsc: A cooperative database caching architecture. In *Proceedings of the WETICE*, 2010.