Cooperative Database Caching within Cloud Environments

Andrei Vancea, Guilherme Sperb Machado, Laurent D'orazio, Burkhard Stiller

Cooperative Database Caching within Cloud Environments

Andrei Vancea[1], Guilherme Sperb Machado[1], Laurent d'Orazio[2], and Burkhard Stiller[1]

[1] Department of Informatics (IFI), University of Zürich, Zürich, Switzerland, {vancea, machado, stiller}@ifi.uzh.ch
[2] Blaise Pascal University - LIMOS, France, laurent.dorazio@univ-bpclermont.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 architecture, which extends the classic semantic caching approach by allowing clients to share their local caches in a cooperative matter. 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. Since most cloud providers charge in a "pay-per-use" matter the amount of transferred data between the cloud environment and the outside world, using such a cooperative caching approach within cloud environmnents presents additional economical advantages. This paper studies possible use-cases of CoopSC within real-world cloud environment and outlines both the technical and economical gains.

1 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 [9]. Semantic caching [12] 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 [19]. 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 [18], uses a cooperative network caching architecture for solving Web flash crow scalability problems. These results show that a cooperative P2P-based caching approach significantly increase the performance of client-server architectures under heavy load.

Nowadays, cloud computing environments have become an important technology which delivers computing resources (CPU, bandwidth, storage) as a service. The end-users do not need to have information about the physical location of these resources, which can be scaled up or down, in an elastic matter, depending on the real-time demand. End-users are charged based on the usage of these resources. Thus, optimizing the use of cloud-based computing resource has now also an important economic importance. The CoopSC architecture reduces the amount of data transferred between database servers and clients. Many cloud providers (e.g., Amazon EC2 [2], Rackspace [5]) bill data transferred between cloud environment and outside world. Therefore, Using the CoopSC approach within a cloud-computing infrastructure presents also economic advantages. This paper studies CoopSC within real-life cloud environment, determines possible usage scenarios, and outlines both its performance-wise and economic advantages.

Like for most existing database cache architectures [12], the major aim of CoopSC [23] 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.

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. CoopSC is suited for applications in which database servers store a large amount of data which needs to be transferred and processed by clients which are running in a distributed environments.

The CoopSC approach was also successfully used in context of network management and traffic analyzing architectures. The NMCoopSC (Network Management CoopSC) architecture [22] optimizes the access to NetFlow records, which are collected by routers and stored in relation database systems. Different analyzers access these records in order to perform task such as intrusion detection of traffic accounting.

2 Related Work

One of the key purposes of caching mechanisms is to reduce the volume of transferred data. Less transferred data can be translated in less costs to maintain an application. Therefore, in the scope of this work is important to analyze which providers or technologies enable an optimum gain to deploy CoopSC. Cloud Computing can be considered as a technology that enables solutions as CoopSC with an optimized spending, since the "pay-per-use" concept [6] is embedded in the Cloud Providers business model. Just paying for the data that is actually transferred between nodes (i.e., server and clients) have clear advantages over the traditional fashion, when customers used to pay a monthly fixed amount independently of how much traffic was spent. However, Cloud Providers may present different charging schemes, mainly related to Infrastructure-as-a-Service (IaaS) [6] products. It is important to analyze each of the charging schemes before deploying any solution into the Cloud.

In the past, Amazon EC2 [2] did not charge for data transfers of two virtual machine instances in the same availability zone. The availability zone is a region that customer defines to host its instances in the moment of the Amazon's instance creation. Nowadays, Amazon charges in and out data transfers independently of the instance's zone [1], meaning that any inbound or outbound traffic will be charged on each virtual node. Rackspace [5] offers a product called Cloud Servers which is very similar to Amazon EC2. Among some technical and feature differences between them, the charging scheme differs related to data transfer between two virtual instances: if the customer does not transfer data using the network interface which has a public IP assigned, such transfer is not charged at all. GoGrid [4] can be considered as a midpoint taking into consideration on how Amazon and Rackspace charges for data transfer. GoGrid does not charge per any inbound traffic into deployed instances, however it charges per any outbound traffic. The examples illustrated above may have advantages and/or disadvantages depending on how the traffic is generated considering the application employed.

[8] shows how Amazon S3 [2] can be used as a storage system for general database applications. In order to reduce the economic cost, the solution caches data on client side and thus, limiting the interactions with the cloud provider. This caching approach only supports strict matches and uses a simple TTL (time-to-live) mechanism for invalidating cache entries.

Client side caching is a commonly used technique for reducing the response time of database queries [9]. 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 [12] 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. However, the classic semantic caching approach does not handle update queries.

Furthermore, the approaches described in [12] does not allow clients to share their caches in a cooperative way. Thus, only local cache entries can be used for answering queries.

[17] 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 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 [14] is a caching service built on top of the Gedeon data management system [13]. 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 and supports only strict hits between query entries. Update queries are also not handled.

Therefore, 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 additional 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 [23]. Using CoopSC within cloud environments offers new economic advantages due to the charging schemes used by most cloud providers.

3 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.

Assuming CoopSC is used in the context of geographical information system (GIS), which stores data about earthquakes, the following example illustrates a possible usage scenario: client C_1 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 $long < 50$).

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.

3.1 Query Rewriting

The query rewriting process 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.

As mentioned, the result of query rewriting process is a *query plan tree* 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 the following types of nodes for executing union and join operations, selection 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*).

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.

The *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 with results of interrogations sent to *distributed index*. These results can refers semantic regions stored by other clients.

3.2 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 $lat < 20$ 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 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.

The distributed index is based on the P2P index described in [21], which adapts the classic MX-CIF quad trees [20] in order to be stored on top of a P2P overlay. CoopSC tailors and implements this approach for efficiently supporting distributed query rewriting.

3.3 Updates

When the content of the database is changed, modifications must be reflected in the cooperative cache. Handling updating 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 which might determine inconsistent results.

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*, 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.

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 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.

4 Scenarios and 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. Evaluation was performed using existing comercial cloud infranstructure providers (Rackspace and Amazon EC2). The economical advantages are outlined using the charging scheme of the two cloud providers valid in November 2010.

4.1 Scenarios

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: a) 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; b) an operational database is running within a cloud environment while clients are running outside. In both scenarios, using the CoopSC approach reduces amount of data sent by database server and thus reduces amount of money that has to be paid for data transfer.

The first scenario (Fig. 1a) corresponds to non-operational use cases in which cloud environment is used for executing specific tasks using data that originates from outside the cloud. For example, cloud solutions could be used for performing CPU intensive simulations with data that is stored in a database located in client's own infrastructure. Multiple cloud nodes are used for decreasing computation times. Clients cache and share input data in order to reduce the communication between cloud environment and outside.

The second scenario (Fig. 1b) expresses operational use cases in which cloud solutions are used as alternative to constructing and maintaining a operational data center.A corporation could use a cloud infrastructure for keeping corporate data which is accessed by clients located in geographically distributed working centers. A cooperative caching solution reduces amount of data sent by database server and, thus, decreases the monetary cost for data transfer.

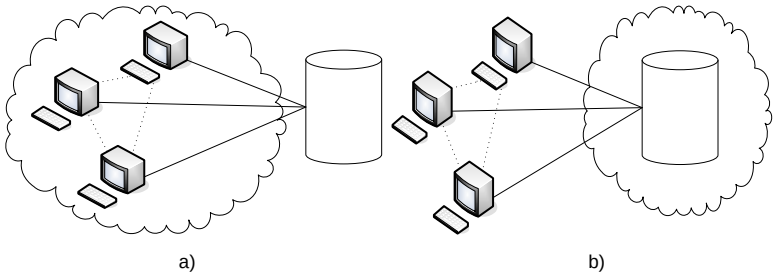


Fig. 1. Cloud Computing Scenarios

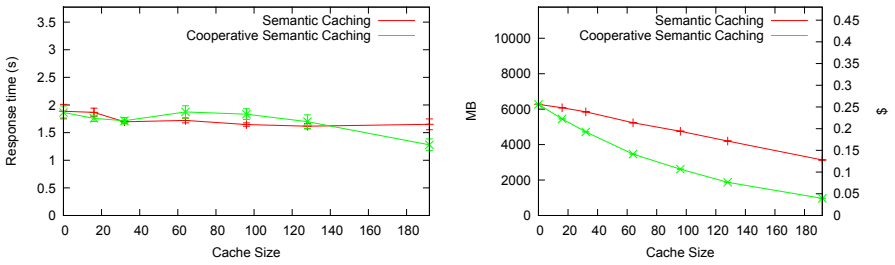
4.2 Evaluation

The evaluation was done using the Wisconsin benchmark [7] relation of 10 million tuples, where each 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 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, total amount of data sent by database server is also measured. Based on the charging schemes, the amount of money paid for data transferred are also calculated

Thus, in each experiment, three measurements are made: query response time (a), amount of data sent by database server and amount of money paid for data transfer (b).

Scenario A In this experiment, database is located in a Zurich, while clients are running in nodes provided by Rackspace [5] cloud infrastructure. The size of clients' caches are varied from 0 to 192 MB. The experiment uses 5 clients. 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 200,000. Each query returns 10,000 tuples. Key results of this experiment are presented in Fig. 2. Due to the instability of the resource provided to virtual machines by Rackspace response time measurements (Fig. 2a) show a high degree of instability and thus, the performance-wise benefits of CoopSC are not evident. Fig. 2b shows the amount of data sent by database server during experiments. Taking Rackspace's charging scheme into consideration, the amount of money that has to be paid for data transferred is computed. Thus, the economic-wise benefits of using a cooperative caching solution in this scenario are shown.

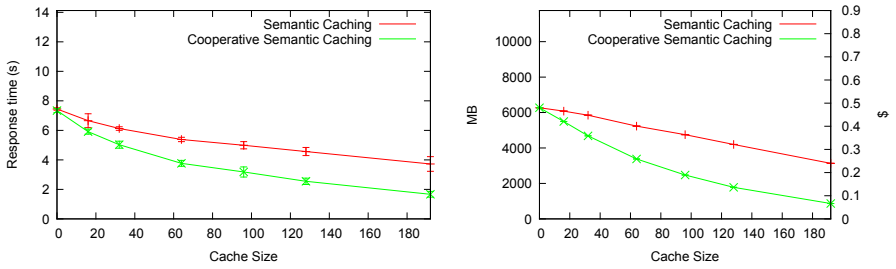


(a) Response Time (b) Data Transferred

Fig. 2. Scenario A: Cache Size

Scenario B In this scenario the database server runs in a large Amazon EC2 [2] instance while clients run in nodes located in the EmanicsLab testing environment. Two experiments were performed: the first experiment measures how cache size influences the performance of system, while the second experiment varies the update rate. The cost of data transfer is computed using Amazon EC2's pricing scheme. It is assume that the total monthly amount of data transfer is between 1 GB and 10 TB.

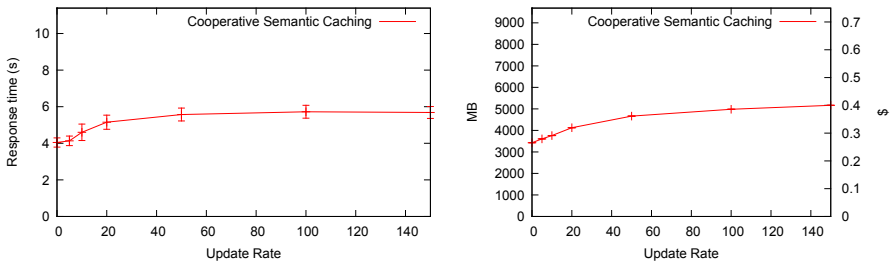
The workload of the first experiment is generated similarly with Scenario A. Key results of this experiment are presented in Fig. 3. Analyzing response time



(a) Response Time (b) Data Transferred

Fig. 3. Scenario B: Cache Size

(Fig. 3a), 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. In the semantic caching approach, the amount of data sent by database server is reduced, because database server only sends parts of queries which are missing from local cache. The cooperative approach further decreases this amount of data because clients can also transfer tuples from caches of other peers. Reducing the amount of data also determine the reduction in the cost of data transfer which is cleary visible in Fig.3b).



(a) Response Time (b) Data Transferred

Fig. 4. Scenario B: Updates

The second experiment investigates how update statements influence the performance of the cooperative caching approach. The size of clients' cache is 64 MB. The workload consists of a sequence of alternative selection and update sessions. Selection session 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 selection sessions. The number of update statements per session is varied from 0 to 150. Fig. 4 illustrates the results of this experiment. While the number of update

statements per session increases, the performance of the caching system starts to decrease because update statements invalidate an increasing number of cache entries. Thus, both query response time and number of tuples sent by database server increase. The cost of data transfer follows the same trend.

5 Summary and 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 within real-life cloud environments. These evaluation results show that CoopSC, especially by applying distributed principles and the P2P overlay techniques in particular, could reduce 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. The economic advantages of CoopSC in the context of cloud solutions were also outlined.

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. When used within cloud environments, CoopSC also reduces the amount of money that has to be paid for data transfer.

6 Acknowledgements

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

References

1. Amazon.com ec2: Ec2 frequently answered question. http://aws.amazon.com/ec2/faqs.
2. Amazon.com web services: Products and services. http://aws.amazon.com/product.
3. Global earthquake model. http://www.globalquakemodel.org.
4. Gogrid website: Gogrid cloud services. http://www.gogrid.com.
5. Rackspacecloud website: Rackspacecloud service. http://www.rackspacecloud.com.
6. Michael Armbrust, Armando Fox, Rean Griffith, Anthony D. Joseph, Randy H. Katz, Andrew Konwinski, Gunho Lee, David A. Patterson, Ariel Rabkin, and Matei Zaharia. Above the clouds: A berkeley view of cloud computing. Technical report, 2009.

7. Dina Bitton and Carolyn Turbyfill. A retrospective on the wisconsin benchmark. *Readings in database systems*, 1988.
8. Matthias Brantner, Daniela Florescu, David Graf, Donald Kossmann, and Tim Kraska. Building a database on s3. In *Proceedings of the 2008 ACM SIGMOD*, pages 251–264, New York, NY, USA, 2008. ACM.
9. 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.
10. Li Chen, Elke A. Rundensteiner, and Song Wang. Xcache: a semantic caching system for xml queries. In *Proceedings of the ACM SIGMOD*, 2002.
11. 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.
12. Shaul Dar, Michael J. Franklin, Björn Thór Jónsson, Divesh Srivastava, and Michael Tan. Semantic data caching and replacement. In *Proceedings of the International Conference on Very Large Databases (VLDB)*, 1996.
13. 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.
14. Laurent d'Orazio and Mamadou Kaba Traoré. Semantic caching for pervasive grids. In *Proceedings of the International Database Engineering and Applications Symposium (IDEAS)*, 2009.
15. Björn Thór Jónsson, María Arinbjarnar, Bjarnsteinn Thórsson, Michael J. Franklin, and Divesh Srivastava. Performance and overhead of semantic cache management. *ACM Transactions on Internet Technology*, 6(3), 2006.
16. Arthur M. Keller and Julie Basu. A predicate-based caching scheme for client-server database architectures. *The VLDB Journal*, 5, 1996.
17. Kostas Lillis and Evaggelia Pitoura. Cooperative xpath caching. In *Proceedings of the ACM SIGMOD*, 2008.
18. 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.
19. Qun Ren and Margaret H. Dunham. Using semantic caching to manage location dependent data in mobile computing. In *Proceedings of the Annual International Conference on Mobile Computing and Networking (MobiCom)*, 2000.
20. Hanan Samet. The quadtree and related hierarchical data structures. *ACM Computing Surveys*, 16, 1984.
21. Egemen Tanin, Aaron Harwood, and Hanan Samet. Using a distributed quadtree index in peer-to-peer networks. *The VLDB Journal*, 16, 2007.
22. Andrei Vancea, Laurent d'Orazio, and Burkhard Stiller. Optimization of flow record handling by applying a decentralized cooperative semantic caching approach. In *13th IEEE/IFIP Network Operations and Management Symposium (NOMS)*, Maui, Hawaii, USA.
23. Andrei Vancea and Burkhard Stiller. Coopsc: A cooperative database caching architecture. In *Proceedings of the IEEE WETICE*, 2010.