Performance and Overhead of Semantic Cache Management

BJÖRN ÞÓR JÓNSSON, MARÍA ARINBJARNAR, and BJARNSTEINN ÞÓRSSON
Reykjavík University
MICHAEL J. FRANKLIN
University of California, Berkeley
and
DIVESH SRIVASTAVA
AT&T Labs–Research

The emergence of query-based online data services and e-commerce applications has prompted much recent research on data caching. This article studies *semantic caching*, a caching architecture for such applications, that caches the results of selection queries. The primary contribution of this article is to revisit the performance and overhead of semantic caching using a modern database server and modern hardware. Initially, the performance study focuses on simple workloads and demonstrates several benefits of semantic caching, including low overhead, insensitivity to the physical layout of the database, reduced network traffic, and the ability to answer some queries without contacting the server. With moderately complex workloads, careful coding of remainder queries is required to maintain efficient query processing at the server. Using very complex workloads, we demonstrate that semantic caching works well in a range of applications, especially in network-constrained environments.

Categories and Subject Descriptors: H.2.4 [**Database Management**]: Systems—*Relational databases, query processing*

General Terms: Algorithms, Performance

Additional Key Words and Phrases: Semantic caching, data caching, online data services, e-commerce applications, wireless networks

1. INTRODUCTION

In the last decade, the world saw a paradigm shift in computing from tightly coupled local-area networks to loosely coupled wide-area networks such as the Internet. The demand for access to data on the Internet has been growing very rapidly as the number of users is constantly growing, and users are using

Author's address: B. Þ. Jónsson, Reykjavík University, Ofanleiti 2, IS-103 Reykjavík, Iceland; email: bjorn@ru.is.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or direct commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 1515 Broadway, New York, NY 10036 USA, fax: +1 (212) 869-0481, or permissions@acm.org.
© 2006 ACM 1533-5399/06/0800-0302 $5.00

the Internet more. Most of the early Web sites were static, and proxy caching was used to improve the response time of Web browsers. Recently, however, dynamic online data services (such as news servers and bibliography servers) and e-commerce applications (such as online stores) have become more and more visible. It is very likely that in the future much of the useful data on the Web will be available through such services.

Due to the sheer scale of the Internet, there are significant problems associated with accessing data from remote servers. Application servers are viewed as a key tool in distributing the workload of online data services and e-commerce applications, and data caching is viewed as a key technique for efficient application servers [Mohan 2002]. The increased use of wireless networks and applications (e.g., see Franklin and Zdonik [1997], and Schwartz [2002]) also implies opportunities for client caching. The characteristics of the environment and workloads, however, make most previous caching architectures unsuitable.

1.1 Caching Architectures

Caching research has typically centered around improving the scalability of new data-intensive environments and applications, and the trade-offs of the techniques studied have largely been determined by the characteristics of these applications. Early work on data caching, for example, focused on object-oriented database systems supporting applications such as CAD/CAM ([DeWitt et al. 1990; Carey et al. 1991; Franklin 1996]). These systems had very tight coupling between the clients and the server which allowed for sharing of individual tuples or whole disk pages (or a mixture of the two). The techniques employed in these studies have been termed *physical caching* techniques.

Currently, online services and e-commerce applications are typically based on simple predicates processed by server-side scripts; in the future, they may be based on more powerful query languages. These query-based services typically expose very little information about the physical layout of the database, and the clients have no control over their internals or interfaces; even application servers have only the information represented in the queries [Luo et al. 2002]. With respect to the caching architectures, the services must therefore be treated as autonomous legacy systems even though they may reside in state-of-the-art commercial database systems. In this environment, physical caching techniques are simply no longer applicable as they assume tight coupling between the client and the server. Additionally, with poor clustering and absence of indices, these architectures do not cope well with wireless environments [Dar et al. 1996].

Data warehousing and view management technologies have also been generating much interest, both in the academic community (e.g., see Stonebraker [1975]; Gupta and Mumick [1995]; Levy et al. [1995]; and Roussopoulos [1998]) and in the computer industry. Since data warehousing is essentially view caching, and views are essentially queries, a branch of caching studies has focused on caching the results of queries. Such techniques are loosely called *logical caching* techniques.

The query workloads of online services and e-commerce applications are very different, however, from those of data warehousing. Joins are rare [Luo

et al. 2002], and selection predicates are typically fine-grained. Much recent work has been done on table-granularity caching (e.g., see Luo et al. [2002], and Candan et al. [2001]) and caching of statically-defined table fragments [The TimesTen Team 2002]. These architectures are not dynamic in their handling of workloads, and will thus be successful only if the static definition of what to cache is correct. Traditional view caching techniques are therefore not appropriate either.

1.2 Semantic Caching

In Dar et al. [1996], we proposed a *semantic caching* architecture, which is a variant of logical caching that dynamically caches results of selection queries to single relations, and can be applied at application servers, clients, proxies, and even at database servers. The term semantic caching derives from the fact that the semantics of the SQL queries are used to organize the contents of the cache and to make inferences about the availability or lack of query results in the cache.

This work has since spawned a line of research that focuses on caching query results at a fine granularity (e.g., see Amiri et al. [2003] and Altinel et al. [2003]; a DBLP title search with the keywords "semantic cach" yields well over twenty references), and the relationship of semantic caching to Internet and data warehousing environments has become increasingly apparent.

The initial simulation results of Dar et al. [1996] demonstrated the effectiveness of the caching strategy, when used with an idealized server that supported semantic caching very well. In Jónsson [1999], however, we studied the performance of the semantic caching client against a then current commercial server (Informix Universal Server). The results showed that the server did not perform well and, in fact, that it obviated many of the benefits of the semantic caching approach.

In the time that has passed since then, database technology and query optimization have been continuously improving. Furthermore, hardware performance has improved tremendously as well, in particular the CPU performance and memory capacity. Therefore, this article revisits the performance and overhead of semantic caching, using a modern database server and modern hardware.

1.3 Contributions of the Article

This article makes two primary contributions. The first contribution is to define the semantic caching architecture with a general, formal model that characterizes the architecture. The second contribution of this article is to present an extensive study of the performance of a prototype implementation of semantic caching.

The performance is studied under both a network-constrained environment and a higher-bandwidth local-area environment. As updates are typically infrequent and localized to a few tables in online data services and e-commerce applications [Luo et al. 2002], we have focused on query workloads in this work. In order to test the capabilities of semantic caching, we study workloads of varying complexity.

Initially, the performance study focuses on relatively simple selection workloads and demonstrates several intrinsic benefits of semantic caching, including low overhead, insensitivity to the physical layout of the database, reduced network traffic, and the ability to answer some queries without contacting the server.

With moderately complex workloads, careful coding of remainder queries is required to maintain efficient query processing at the server. We propose and study two different approaches to remainder query processing.

Then the study turns to very complex selection workloads. The results show that semantic caching utilizes the network very efficiently across a wide range of workloads. The query response time is also excellent for a range of applications, although that range is much more restricted due to cache management overhead and inefficient server processing of requests for data missing from the cache.

The remainder of the article is organized as follows. Section 2 surveys the related work in the area. Section 3 defines the general semantic caching architecture and briefly describes the prototype implementation. Section 4 contains the detailed performance study. Finally, Section 5 presents our conclusions.

2. RELATED WORK

In this section, we briefly review the related work on database caching (for more detailed surveys, see Jónsson [1999], and Arinbjarnar et al. [2006]). The early work focused on physical caching techniques—object caching (e.g., see DeWitt et al. [1990]), page caching (e.g., see Franklin [1996]), and combinations of the two (e.g., see Kemper and Kossmann [1994]). As demonstrated in Jónsson [1999], however, these techniques are not applicable to the problem of application server caching.

Another line of work focused on materialized views and view caching (refer to Roussopoulos [1998] for a survey of view caching work). In Delis and Roussopoulos [1992, 1998], an enhanced client-server architecture based on view caching techniques was proposed and studied. In this architecture, each client stored query results retrieved from the server as base tuples along with index structures. The queries addressed by the architecture were project-select-join queries. Reuse of cached query results was based on subsumption and thus partial matches were not utilized. The enhanced client-server architecture was geared towards join-based workloads, and performed well under such workloads. Workloads with selection predicates that tend to overlap in arbitrary ways, however, would cause problems for the enhanced client-server architecture. Another architecture based on view caching techniques is the A*Cache architecture [Keller and Basu 1996]. In Deshpande et al. [1998] a hybrid approach that has features from both physical and view-caching approaches is proposed, namely, a chunking architecture for the processing of aggregations in an OLAP environment.

Much recent work has been done on table granularity, middle-tier caching, typically in the context of e-commerce applications (e.g., see Luo et al. [2002] and Candan et al. [2001]). These architectures are logical in the sense that the

tables are typically obtained via queries or replication, and hence the architectures do not necessarily copy the structure of the tables. In Luo et al. [2002], it is shown that there is likely a clear separation between update-intensive tables and read-intensive tables in e-commerce applications, and that access is heavily skewed towards the read-intensive. Needless to say, when access is further skewed within the relations, table-level caching is wasteful. In The TimesTen Team [2002], caching of statically-defined table fragments is described. Such architectures have better control than table caching architectures but may suffer from the problem of incorrectly defined fragments.

In Amiri et al. [2003], an architecture called DBProxy is proposed that is quite similar to semantic caching. They propose to implement the cache as a JDBC driver that can be plugged under the application server. Selections are handled similar to semantic caching except that the description of cached queries may overlap, in contrast to our nonoverlapping semantic regions. Joins are handled as if they were a relation, effectively simulating a materialized view. Updates are handled via an update stream from the server. The work does not extensively study the trade-offs of complex workloads as is done in this article.

In Altinel et al. [2003], another architecture for application server caching is presented, called DBCache, which partially builds on and extends the semantic caching architecture. They define cache tables, which are either static or dynamic, and cache constraints that help in determining the usefulness of data in the cache. The cache manager implementation smartly uses a full-fledged DB2 database system to manage the cache and uses the federated query processing capabilities of the server to faciliate processing of queries at the client. The architecture supports queries with an equality predicate on at least one domain complete column, where domain completeness indicates that if one record with a particular value is in the cache, then all records with that particular value are in the cache. Extending the architecture to handle range queries is not considered. Joins are handled via referential cache constraints. If all the required data is in the cache, the query is performed at the client; otherwise the query is submitted to the server and the results are cached at the client.

Other works that extend logical client caching ideas include Ren and Dunham [1999, 2000] where projections are also considered and Godfrey and Gryz [1999] where project-select-join queries are considered whose aim is to support caching of query results for general project-select-join queries, while still supporting reuse of partial results whenever possible. Semantic caching ideas have been applied to various applications, such as mobile clients [Dar et al. 1996; Ren and Dunham 1999, 2000; Zheng et al. 2004] Web caching [Chidlovskii et al. 1999; Chidlovskii and Borghoff 2000; Lee and Chu 2001], XML databases [Chen et al. 2002; Hristidis and Petropoulos 2002], and data grids [Wan et al. 2004].

3. THE SEMANTIC CACHING ARCHITECTURE

In this section, we first present a formal model of the semantic caching architecture. We then briefly present the prototype implementation used in this

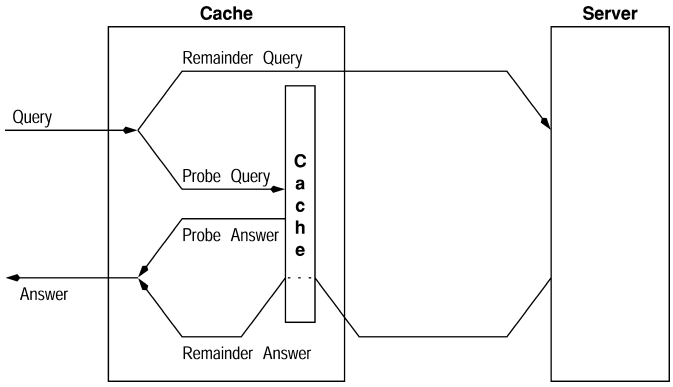


Fig. 1. Query processing in the semantic cache.

article. Note that, while the semantic caching model can handle a wide range of selection queries, the prototype implementation caches results of range queries. Finally, we discuss how to extend the prototype to handle general query processing.

3.1 Model of the Semantic Cache

3.1.1 *Cache Contents.* The semantic cache stores results of queries that are submitted to the server. The queries supported in-cache by the semantic cache are selections on single relations. In this model, the concept of *constraint formulas* refers to the predicates of these selections. Any constraint domain that is closed under conjunction, disjunction, and negation can be used.[1] The following definition describes the queries supported by the cache and provides the basis for the formal model of the semantic cache.

Definition 3.1. Consider an n-ary relation R, with attributes $A_1, \ldots, A_n$. A *query* Q on relation R is a constraint formula whose free variables are from $A_1, \ldots, A_n$.

The portion of each relation present in the cache is described by a constraint formula. This formula represents a view on the relation; the view changes dynamically based on the queries posed to the cache and the data that is evicted from the cache to make room for new query results. The entire contents of the cache are described by a set of such constraint formulas, one for each database relation.

3.1.2 *Server Communication and Reuse of Partial Matches.* Semantic caching reuses partial matches in the cache. As shown in Figure 1, each query presented to the cache is split into two disjoint portions: one that can be completely answered using the tuples present in the cache, and another that requires tuples to be shipped from the server. The notions of a probe query and a remainder query correspond to these two portions of the query.

[1]As discussed later, however, the constraint formulas supported in our prototype are range predicates over relation attributes connected by $\wedge$ and $\vee$ (assumed to be in disjunctive normal form).

Definition 3.2. Consider a query Q on a relation R. If V denotes the constraint formula describing the set of tuples of R present in the cache, then the *probe query* of Q into V, denoted by $\mathcal{P}(Q, V)$, is the constraint formula $Q \wedge V$.

Definition 3.3. Consider a query Q on a relation R. If V denotes the constraint formula describing the set of tuples of R present in the cache, then the *remainder query* of Q with respect to V, denoted by $\mathcal{R}(Q, V)$, is the constraint formula $Q \wedge (\neg V)$.

It should be clear that $\mathcal{P}(Q, V) \wedge \mathcal{R}(Q, V) \equiv \mathit{false}$, indicating that the probe and remainder queries do not overlap, and that $\mathcal{P}(Q, V) \vee \mathcal{R}(Q, V) \equiv Q$, indicating that the union of the answers to the two queries forms the answer to the original query.

The answer to the probe query is already in the cache and can be returned to the user (or the query engine) immediately. The answer to the remainder query needs to be fetched from the server and brought into the cache before it can be returned. The constraint formula for the remainder query is used to construct a query that is submitted to the server. Following is a concrete example that illustrates the definitions presented so far in this section.

Example 3.4. Consider a query to find employees whose salary exceeds \$50,000 and who are at most 30 years old. This query can be described by the constraint formula:

$$Q = (\text{Salary} > 50{,}000 \wedge \text{Age} \leq 30)$$

on the relation *Employees* (Name, Salary, Age). Assume that the cache contains all employees whose salary is less than \$100,000 as well as all employees who are between 25 and 28 years of age. This view can be described by the constraint formula:

$$V = (\text{Salary} < 100{,}000) \vee (\text{Age} \geq 25 \wedge \text{Age} \leq 28).$$

The probe query into the cache, or the intersection of the query with the cache contents, is described by the constraint formula:

$$\begin{aligned}\mathcal{P}(Q, V) = \ & (\text{Salary} > 50{,}000 \wedge \text{Salary} < 100{,}000 \wedge \text{Age} \leq 30) \vee \\ & (\text{Salary} > 50{,}000 \wedge \text{Age} \geq 25 \wedge \text{Age} \leq 28).\end{aligned}$$

This constraint formula encapsulates all the tuples contained in the cache that are answers to the query. The remainder query, on the other hand, is described by the constraint formula:

$$\begin{aligned}\mathcal{R}(Q, V) = \ & (\text{Salary} \geq 100{,}000 \wedge \text{Age} < 25) \vee \\ & (\text{Salary} \geq 100{,}000 \wedge \text{Age} > 28 \wedge \text{Age} \leq 30).\end{aligned}$$

This constraint formula describes the tuples that need to be fetched from the server.

When the constraint formulas are arithmetic constraints over the attributes $A_1, \ldots, A_n$, they have a natural visualization in the n-dimensional semantic space $\mathcal{D}_1 \times \mathcal{D}_2 \times \cdots \times \mathcal{D}_n$, where $\mathcal{D}_i$ is the domain of attribute A_i. Figure 2 depicts

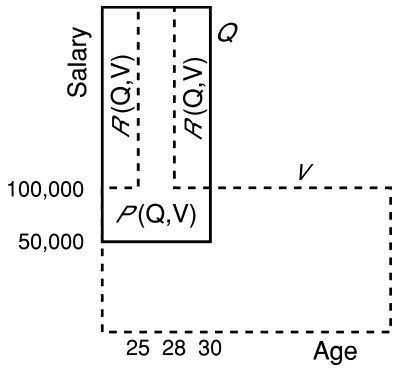


Fig. 2. Query, probe query, and remainder query for Example 3.4.

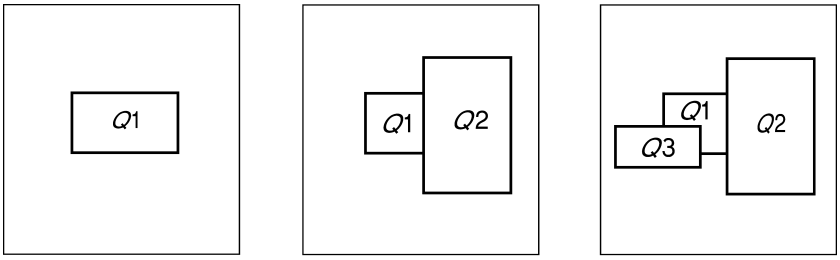
the projection of the query Q, cache contents V, probe query $\mathcal{P}(Q, V)$, and remainder query $\mathcal{R}(Q, V)$ onto the Salary and Age attributes of the employee relation.

3.1.3 *Data Grouping and Cache Overlap.* The semantic cache manages the cache contents as a collection of *semantic regions* that group together related tuples. These semantic regions are disjoint and each tuple in the cache is associated with exactly one semantic region. The semantic regions can be modified dynamically based on the queries that are posed to the cache or other requirements of the application.

Definition 3.5. Consider an n-ary relation R. A *semantic region* S on relation R is a triple (Q_S, T_S, v_S), where Q_S is a query, T_S is the set of tuples in the answer to Q_S, and v_S is a replacement value. If V denotes the constraint formula describing the set of tuples of R present in the cache and $S_1, \ldots, S_m$ is the set of semantic regions on relation R, then $S_1, \ldots, S_m$ must form a *partition* of V, that is, $Q_{S_i} \wedge Q_{S_j} \equiv$ false for all $i \neq j$, and $\bigvee_{i=1,\ldots,m} Q_{S_i} \equiv V$.

Each semantic region thus has a set of tuples, a constraint formula that describes the tuples grouped together within the region, and a single replacement value. For example, the data in Figure 2 corresponding to $\mathcal{R}(Q, V)$ could be one semantic region, and the data corresponding to $\mathcal{P}(Q, V)$ could be another. When there is insufficient space in the cache, the semantic region with the lowest value and all tuples within that region are discarded from the cache. Semantic regions are, thus, the unit of cache replacement.

3.1.4 *Forming Semantic Regions.* Recall that the definition of semantic regions only specifies that the regions need to be disjoint, but has no other requirements about how regions are formed. How the semantic regions are formed, however, determines the granularity of the regions in the cache and thus has a significant effect on the performance of the semantic cache. This section describes the process of forming regions based on the queries that are posed to the cache.



(a) Regions after $Q1$ (b) Regions after $Q2$ (c) Regions after $Q3$

Fig. 3. Semantic region formation: full coalescing.

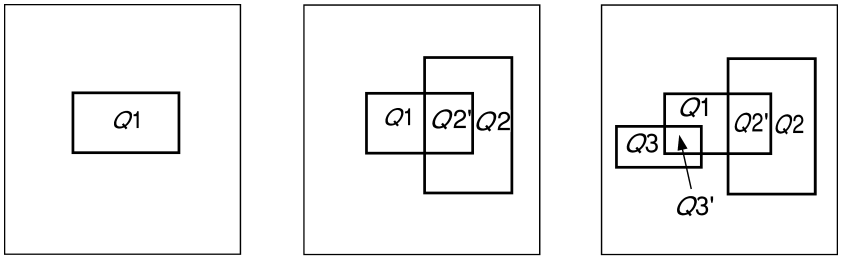
When a query intersects with a semantic region in the cache, that region is split into two smaller disjoint semantic regions. One of those regions is the intersection of the semantic region and the query, and the other is the difference of the semantic region with respect to the query. Data brought into the cache as the result of the remainder query also forms a new semantic region. Thus, the execution of a query that overlaps n semantic regions in the cache can result in the formation of $2n + 1$ such regions; of these regions, $n + 1$ form the answer to the query. The problem of region formation is then whether or not to coalesce some or all of these $n + 1$ regions into one or more larger regions.

The straightforward approach is *full coalescing*, that is, to always coalesce all the regions that were used by the most recent query, resulting in only one region corresponding to the query. The following example illustrates this approach.

Example 3.6. Figure 3 illustrates the semantic regions formed for a sequence of three range queries on a single binary relation using full coalescing. The constraint formula $Q1$ corresponding to the first query forms the only semantic region after $Q1$ is issued (Figure 3(a)). The second query $Q2$ forms a new semantic region. Since this new semantic region overlaps with the semantic region $Q1$ and semantic regions must be mutually disjoint, the semantic region $Q1$ shrinks to the portion that is disjoint with $Q2$ (Figure 3(b)). Similar shrinking occurs when the third query is issued (Figure 3(c)). Note that the semantic region $Q1$ is no longer convex and its constraint formula is no longer conjunctive. In fact, semantic regions may even be disconnected in the semantic space.

With small queries (relative to cache size), full coalescing leads to good performance. When the answer to each query takes up a large fraction of the cache, however, this strategy can result in semantic regions that are excessively large. The replacement of such a large region can empty a significant portion of the cache, resulting in poor cache utilization.

Another option is *no coalescing*, or to never coalesce any regions. Figure 4 demonstrates the semantic regions formed using no coalescing, for the same stream of queries as in Figure 3. For small queries that tend to intersect, this can lead to excessive overhead but, for larger queries, it alleviates the granularity problem. In the experiments reported in this article, full coalescing is used since using no coalescing resulted in many more regions and more processing overhead.



(a) Regions after $Q1$ (b) Regions after $Q2$ (c) Regions after $Q3$

Fig. 4. Semantic region formation: no coalescing.

3.2 Prototype Implementation

We have implemented a semantic caching prototype for application server or client caching that caches results of range queries based on the semantic caching model. The prototype consists of about 24K lines of well-documented C++ code (the semantic cache itself is fewer than 10K lines). The architecture of the prototype is illustrated in Figure 5(a), which shows the main components as well as the flow of control information (solid lines) and data (dashed lines).

The prototype connects to relational servers via the ODBC standard. When the cache connects to the server, it first reads the schema information, such as the names of the relations and the names and types of their attributes; this information is used to set up the data structures of the cache. To subsequently retrieve results from the server, the cache forms SQL queries that are submitted to the server. At the server, each query is first optimized and then run according to the best plan.

The prototype has a rudimentary query processor that handles range selection queries over single relations. The User Interface reads pregenerated query specifications from text files and passes them on to the Client Controller. The Client Controller coordinates with the Semantic Cache to determine the probe query and the remainder query by scanning through all the regions for the given relation. Then a query tree such as the one shown in Figure 5(b) is constructed. Note that all the operators in the figure run at the prototype.

The operators use the iterator model of query processing [Graefe 1994]. The RegionScan operator scans all the regions that form the probe, while the RemoteScan operator obtains the data in the remainder query from the server via the ODBC interface and inserts it into the cache before passing the data up the query tree. The Union operator concatenates the two input streams. Finally, the Display operator displays query results to the user.[2]

A package was implemented that manipulates constraint formulas in disjunctive normal form (DNF). It deals with predicates that are closed ranges of integers.[3] In the prototype, each constraint formula is a list of nonoverlapping

[2]The operator tree in Figure 5 is the most general tree possible. If the probe query is empty, both the RegionScan operator and the Union operator are omitted. Likewise, for empty remainder queries, the RemoteScan and Union operators are omitted.

[3]Extending the package to handle open predicates, such as $Emp.Age > 50$, is straightforward.

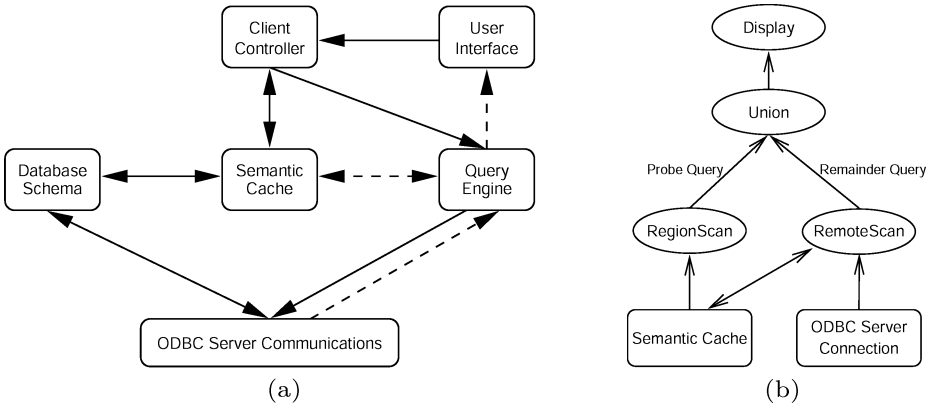


Fig. 5. Semantic caching prototype (a) architecture and (b) query tree.

n-dimensional rectangles, termed *factors*. The operations required by the semantic caching model (Intersection, Difference, etc.) have been implemented as well as operations required by semantic value functions (Center, Distance, etc.). In the prototype, the remainder query is processed after the probe query to avoid complications in loading data into the cache. Since the execution of the probe query is generally very efficient, the effects on the results are minimal.

3.3 General Query Processing

The processing of basic selections in the cache has been discussed previously. It is likely, however, that semantic caching must be used to answer queries that require more processing than the constraint manipulation that the cache can manage. The general approach to processing such queries is simple: push all the selections that the cache can handle to the bottom of the query plan and evaluate the remaining operators on the results of those selections. Pushing selections down is a standard query optimization technique in relational databases. This section describes how the semantic cache might be used to answer more complex queries in the future.

3.3.1 *Handling Updates.* In the general case, updates can only be handled with some cooperation from the server. The implementation options are the same for semantic caching as for other cache architectures. Time-To-Live information can be submitted with the remainder query results and maintained with semantic regions. Alternatively, the server may notify clients of changes in which case the clients can either invalidate the corresponding cache contents or retrieve the changes from the server. The server can maintain (partial) information about the contents of the clients' cache or simply broadcast all changes. Updates have been studied for many other caching architectures (e.g., see Franklin [1996], Delis and Roussopoulos [1998], and Amiri et al. [2003]); studying update performance for semantic caching is an interesting avenue for future work.

3.3.2 *Join Queries.* As semantic caching deals with selections over single relations, it is not particularly well suited for processing join queries with no additional selection predicates. Consider, for example, a join query with no selections on either relation.

```
SELECT *
FROM Employees E, Dependents D
WHERE E.ssn = D.emp_ssn
```

In order to process this query, both relations must be present in the cache.[4] If additional selections are made on either relation, however, these selections can be pushed down to the cache, resulting in performance improvements. If the selection is on the join attribute, then that selection can be applied to both relations, resulting in further savings.

3.3.3 *Other Queries.* There are some unconventional ways in which the semantic regions might be used to facilitate the query processing, in particular when semantic query optimization can be applied. Following are two examples of this.

Ordering Query Results. Consider, for example, the following query that selects all employees over 50 years old and orders the result based on their age.

```
SELECT *
FROM Employees E
WHERE E.Age > 50
ORDER BY E.Age DESC
```

Assume there are some semantic regions in the cache that contribute tuples to the answer. If none of the contributing regions intersect on the *Employees.Age* attribute, then it is sufficient to first order the regions, and then sort the tuples of each region independently. Many small sorting operations require less memory and time than one large operation. Even when some regions intersect along the *Employees.Age* attribute, partial orderings may be considered that reduce the sorting work.

Top-n Queries. Another example is the following query, that requests the 5 highest paid employees over 50 (the syntax is borrowed from Carey and Kossmann [1997]).

```
SELECT *
FROM Employees E
WHERE E.Age > 50
ORDER BY E.Salary DESC
STOP AFTER 5
```

Obviously, if the cache contains a region (or regions) for *Employees.Age* > 50, then that region can be used to answer the query. But that is not the only

[4]Recall that the client or application server can always opt to ignore the cache and send the whole query to the server. The result of such a query, however, can currently not be cached in this architecture.

way this query can be answered. If a region with the description *Employees*. Salary $> x$ exists in the cache that has more than 5 tuples that satisfy *Employees*.Age >50, then the server need not be contacted as all the required results are in the cache.

4. PERFORMANCE EXPERIMENTS

The goal of this section is to study the performance of semantic caching against a modern database server on modern hardware. We first review previous results and explain why the traditional physical caching approaches are not feasible when connecting to a commercial relational server. Following a description of the experimental environment, the performance study is then divided into three main parts. In the first part, we study single attribute workloads in order to understand issues of clustering and indexing. In the second part, we study workloads with selections over two attributes in order to understand remainder query execution strategies. In the third and final part, we study the performance of semantic caching with multi-attribute selections, to understand the effects of complex workloads on query response time and cache processing. The section ends with a summary of the results.

4.1 Summary of Previous Results

In Dar et al. [1996], a simulation study was performed that compared semantic caching to the two most common client caching architectures, tuple caching and page caching. The tuple caching and page caching architectures were modeled after the corresponding architectures from DeWitt et al. [1990]. For each architecture, a server was simulated that served it in the most efficient manner.

As expected from the results of DeWitt et al. [1990], the primary trade-offs between the architectures during indexed access were those of data clustering vs. aggregation of overhead. Page caching uses the clustering of the database and is able to aggregate the overhead of keeping track of data stored in cache by storing whole disk pages. During unclustered access, however, page caching is very inefficient, and the cache overhead of storing useless data far outweighs the benefit of overhead aggregation. Tuple caching performs worse than page caching during clustered access due to its higher overhead of maintaining the cache description. Tuple caching, on the other hand, is able to recluster the data into the cache based on the access pattern, and thus is much less sensitive to clustering than page caching. Semantic caching has the best of both worlds: it is able to aggregate the cache overhead, while retaining the ability to recluster data into the cache based on the access pattern.

During unindexed access, however, the largest benefit of the semantic description is seen. In this case, neither page caching nor tuple caching are able to use the cache effectively because the cache description of these architectures can only describe what is in the cache, but not what is *not in the cache*. As a result, the page and tuple servers must perform an expensive scan operation for every query, regardless of whether any data is missing from the cache or not. The semantic cache description, on the other hand, allows the semantic cache to reason about the data that is missing from the cache and, in particular,

figure out when all the data needed for the answer is in the cache. This allows the client to avoid contacting the server in these cases, resulting in significant savings in certain workloads.

Additional experiments demonstrated the feasibility of using specialized semantic value functions that used properties of the data and queries to evaluate the usefulness of data in the cache and guide the cache replacement process.

As mentioned earlier, Dar et al. [1996] focused on comparing caching schemes that operate against a well-behaved server, that is, a server built specifically to serve each particular caching scheme. In Jónsson [1999], we studied the performance of the client prototype used in the current article against a then current commercial relational database server (Informix Universal Server). As the results showed, neither tuple caching nor page caching is feasible in such an environment, as both approaches require low-level information about data placement on disk which is not available outside the database server.

While semantic caching was shown in Jónsson [1999] to perform well for a range of workloads, the relational server was not well equipped to handle complex remainder queries. As we will see in the remainder of this section, the same observations hold today, despite faster hardware and significant improvements in database server performance.

4.2 Experimental Environment

4.2.1 *System Configuration.* In the following, the semantic caching client prototype is compared against a noncaching client which also uses ODBC to connect to the database server. Clients are run on a 1.5GHz Intel Pentium PC with 512MB of memory and an 18.6GB disk, running Windows XP Professional Workstation SP2. The server is run on a 1.6GHz Intel Pentium PC with 512MB of memory and an 18.6GB disk. The relational server used is Microsoft SQL Server 2000 Enterprise Edition, version 8.00.194, running on Windows Server 2000 Standard Edition. The database is assigned disk space of nearly 2.5GB and stored in a regular file. The server is used as it was installed; that is, no performance tuning has been done. Both machines are used exclusively for the experiments. The network connecting the two machines is a 100Mbit isolated Ethernet.

4.2.2 *Database and Workloads.* The database used in the experiments is based on the Wisconsin benchmark [DeWitt 1993]. It contains a single relation of 10 million tuples, called *Wisc10M*.[5] Each tuple holds 208 bytes of data, for a total of over 2GB of data. Three candidate keys from the relation are used in the experiments: *Unique2* is indexed (using a B^+-tree index) and perfectly clustered—the relation is ordered on this attribute; *Unique1* is also indexed

[5]In the experiments, only 10% of the relation is used. The reason for using this large relation is that some of the workloads used in Section 4.5 cannot be defined on a smaller relation. Although using such a large relation results in more expensive query evaluation in some cases, it does not favor semantic caching.

Table I. Workload Parameters and Settings

Parameter	Value	Description
QuerySize	1,000–10,000	Size of each query (tuples)
HotSpot	10,0000	Size of hot region (tuples)
Skew	90%	Percentage of queries to hot region
CacheSize	4–128 MB	Size of the cache

with a B^+-tree index, but is completely unclustered; *Unique3* is both unindexed and unclustered.[6] The candidate keys have values from 0 to 9,999,999.

Table I shows the key parameters of the workloads. A benchmark consisting of simple selection queries is used. The size of the result (*QuerySize*) ranges from 1,000 tuples to 10,000 tuples and is varied in the experiments by adjusting selectivities along one or more of the candidate keys listed previously. A fixed portion of the queries (*Skew*) has the centerpoint uniformly distributed within a hot region of size *HotSpot*. The remaining queries are uniformly distributed over the cold area which surrounds the hot spot. The cache size selection is discussed in detail in Section 4.2.4.

4.2.3 *Metrics.* As discussed in previous sections, the semantic caching architecture was motivated in part by network-constrained environments. In such environments, the most important metric is the number of tuples transferred across the network. Also of interest is the performance of semantic caching in high-bandwidth local-area networks, such as the network that the prototype and server are connected to. For that environment, the wall-clock response time of queries is the main metric. The prototype architectures maintain over 50 other metrics, such as cache hit rate, cache overhead, etc.; some of these metrics are discussed when they give further insights into the performance trade-offs.

Each datapoint was typically obtained by posing queries from the workload until the cache was full, and then averaging the metrics across 1,000 subsequent queries. For the noncaching client, however, the performance of each query is independent of the previous queries, and the metrics converge much faster than with caching. For the noncaching architecture, therefore, at least 100 queries were measured. These workloads were sufficient to obtain conclusive results within a reasonable timeframe. In all of the experiments, the results of individual queries were discarded, that is, they were neither displayed to a screen nor written to a file.

Note that in all of the experiments in this section, semantic caching maintains six attributes including all three candidate keys. For single and double attribute selections, this means that semantic caching maintains 5 and 4 attributes, respectively, that are not being used by the workloads. The cache overhead of semantic caching is therefore higher in these experiments than is strictly necessary.

[6]Note that contrary to intuition, *Unique2* is the clustered attribute and *Unique1* the unclustered one. This naming convention stems directly from the Wisconson benchmark.

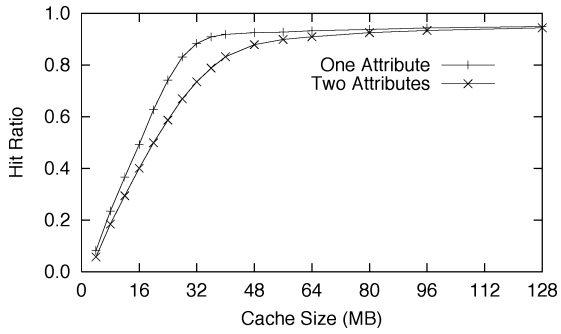


Fig. 6. Cache hit ratio as a function of cache size.

4.2.4 *Cache Size.* In order to determine suitable cache sizes for our experiments, we initially ran query workloads with 10,000-tuple queries over one and two attributes. Figure 6 shows the hit ratio of the cache as the cache size is varied from 4MB to 128MB. Overall the figure shows that for small cache sizes, the cache effectiveness is roughly proportional to the size of the cache. Once the cache becomes large enough to hold the hot spot, however, additional cache memory does not siginificantly improve the hit ratio. Note that the skew of the workloads is 90% so the hit ratio levels off after reaching 0.9.

For queries over one attribute, the hot spot fits in less than 30MB. For queries over two attributes, however, the hot spot requires much more memory and is not contained until around 60MB. This is because the query workload is created such that the centerpoint of the query is within the hot spot. Parts of the query, however, may extend beyond the hot spot—forming a warm spot of tuples which are also frequently accessed—in particular, with the large 10,000-tuple queries. In the remainder of this section, we have therefore chosen to experiment with three different cache sizes: 32MB, 64MB, and 96MB.

4.3 Single Attribute Experiments

In this first group of experiments, we study single-attribute workloads in order to better understand issues of clustering and indexing. We use three different attributes: a clustered, indexed attribute; an unclustered, indexed attribute; and an unindexed attribute.

4.3.1 *Experiment* 1*: Clustered, Indexed Attribute.* In this experiment, the selections are performed on the *Unique2* attribute which has a clustered index. This represents the most efficient access pattern at the server. Figure 7 shows the tuples transferred across the network for each configuration. The query result size is varied along the x-axis by changing the selectivity of the queries posed at the client. As the figure shows, semantic caching performs very well on this metric compared to noncaching, reducing network traffic by 85–93%.

Figure 8, on the other hand, shows the query response time when remainder queries are run against the relational database server across a local-area network. As was the case with network traffic, Figure 8 shows that semantic caching provides significant savings over noncaching, ranging from 81% to 92%.

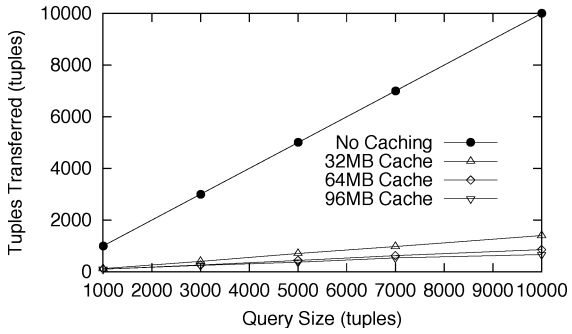


Fig. 7. Tuples transferred for clustered attribute.

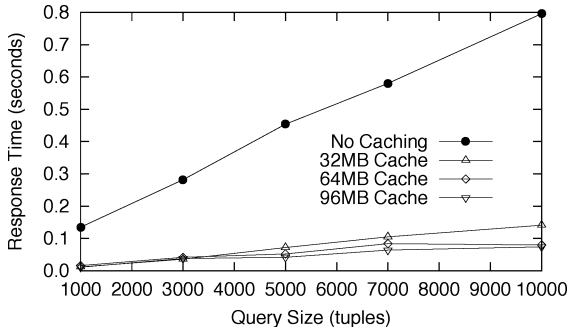


Fig. 8. Response time for clustered attribute.

The savings for response time are relatively smaller than for tuples transferred as there is a fixed cost of query optimization which is paid for each query sent to the server.[7] This cost is also relatively higher for the smaller queries, leading to smaller relative savings for small queries.

The bulk of the query response time is due to remainder query execution since cache processing (not shown) always requires less than 15 milliseconds for the semantic cache. In some cases, however, the client is able to completely answer the queries from the cache and avoid contact with the server, resulting in extremely efficient query processing. The percentage of queries answered completely from the cache (not shown) ranges from 72% to 81% for the smallest cache size and 87% to 91% for the largest cache size.

The cache overhead (not shown) consists of the data structures to keep track of regions and tuples and free space resulting from eviction of large regions from the cache. For all query sizes and cache sizes, the size of the data structures is about 3.3% of the cache size, while the free space is less than 2% in all cases, except for large queries running against the small cache, where the free space is less than 3%.

[7]The remainder queries were not coded to take advantage of plan caching which could reduce this cost.

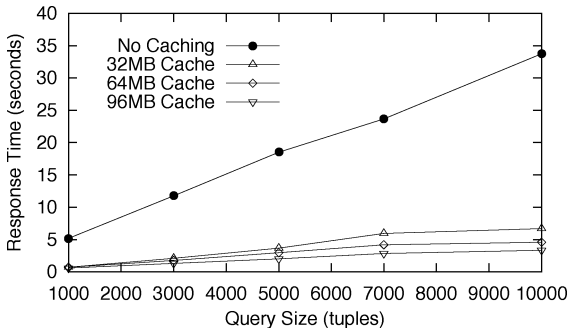


Fig. 9. Response time for unclustered attribute.

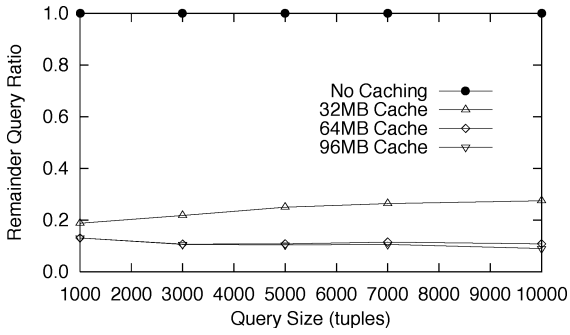


Fig. 10. Fraction of queries resulting in a remainder query for unindexed attribute.

4.3.2 *Experiment* 2: *Unclustered, Indexed Attribute.* This next experiment considers workloads where the selection is on *Unique1*, the unclustered, indexed attribute. The results for tuples transferred across the network are identical to the clustered case and are therefore not shown here. Figure 9 shows the query response times for the three architectures. As the figure shows, using semantic caching results in similar relative savings as before. Because the queries are now run against an unclustered index, however, the remainder query response time is much higher than for the clustered attribute by a factor of 40 or more. Cache processing time (not shown) is unaffected and remains below 15 milliseconds in all cases.

4.3.3 *Experiment* 3: *Unindexed Attribute.* This experiment considers selections on the unindexed attribute *Unique3*. The unindexed attribute is a special case in terms of query response time because the server *must* perform a sequential scan of the relation for each remainder query. Because each sequential scan costs a fixed amount of time (7–8 minutes with our large relation), the savings of semantic caching depend solely on how many queries are answered fully from the cache.

Figure 10 shows the fraction of queries that generate a remainder query to the server (all noncaching queries are considered remainder queries in this figure). Overall, Figure 10 shows that with a 32MB cache, 72–81% of all queries are satisfied entirely from cache, while with the larger cache sizes, 87–91% are

answered from the cache. As explained previously, response time is improved proportionately.

Figure 10 shows a very interesting effect, however, as the performance is slightly improved with larger queries for larger cache sizes, while for the smaller 32MB cache, performance is noticably worse with larger queries. The explanation for the worse performance of the 32MB cache is that, with such a small cache, a cold query will result in the eviction of hot spot data; with larger queries, more hot data must be replaced, resulting in a lower hit ratio and a higher ratio of remainder queries. For the larger cache sizes, cold data evicts other cold data and performance is improved as expected.

4.4 Double Attribute Experiments

In the experiments presented, we saw that semantic caching performs very well for single attribute queries in all cases. In many cases, however, more than one attribute is included in the query constraints. The workloads of this section, therefore, consider moderately complex workloads which use a combination of two attributes. The queries are square shaped with equal ranges along each attribute. The value of these experiments is twofold. First, as mentioned earlier, the warm spot around the hot spot extends further with two attribute workloads than with queries over a single attribute, requiring a larger cache to achieve a comparable hit ratio. More importantly, however, this workload exposes a flaw in the relational server, which is unable to efficiently process some of the remainder queries of semantic caching, often reverting to scanning large portions of an index or even the entire relation. We first explain why the server is unable to process remainder queries efficiently in some cases and propose two solutions to this problem based on augmenting the remainder queries with standard SQL constructs. Then we examine the performance of these remainder query approaches.

4.4.1 *Remainder Queries.* Consider the following actual remainder query over two attributes resulting from one of our experiments.

```
SELECT *
FROM Wisc10M
WHERE ((Unique2 > 1669497 AND Unique2 < 1694104) AND
       (Unique1 > 2037839 AND Unique1 < 2064950))
   OR ((Unique2 > 1698170 AND Unique2 < 1702035) AND
       (Unique1 > 1965742 AND Unique1 < 1971745))
   OR ((Unique2 > 1702034 AND Unique2 < 1707060) AND
       (Unique1 > 1965742 AND Unique1 < 1968517))
```

The optimizer should potentially be able to find a lower bound and an upper bound on each of the two attributes and use those bounds to limit the search. This feature is not implemented in SQL Server, however, resulting in a sequential scan of the entire relation.[8] The lack of this feature is understandable,

[8]Note that if all factors share a common upper or lower bound, then the optimizer is able to use that bound to limit query processing.

of course, since queries such as this remainder query are quite uncommon in typical workloads.

Our first remainder query approach compensates for this flaw by simply supplying a bounding box with the query, resulting in the following remainder query.

```
SELECT *
FROM Wisc10M
WHERE ((Unique2 > 1669497 AND Unique2 < 1694104) AND
       (Unique1 > 2037839 AND Unique1 < 2064950))
   OR ((Unique2 > 1698170 AND Unique2 < 1702035) AND
       (Unique1 > 1965742 AND Unique1 < 1971745))
   OR ((Unique2 > 1702034 AND Unique2 < 1707060) AND
       (Unique1 > 1965742 AND Unique1 < 1968517))

  AND ((Unique2 > 1669497 and Unique2 < 1707060) AND
       (Unique1 > 1965742 and Unique1 < 2064950))
```

The server is able to use the last constraint to guide the index usage, while the original constraints are used to filter out tuples that should not be returned.

Our second approach treats each individual factor with a separate SELECT statement. Since the factors are guaranteed not to overlap, the final result can then be assembled via UNION ALL statements resulting in the following query.

```
SELECT *
FROM Wisc10M
WHERE ((Unique2 > 1669497 AND Unique2 < 1694104) AND
       (Unique1 > 2037839 AND Unique1 < 2064950))
UNION ALL
SELECT *
FROM Wisc10M
WHERE ((Unique2 > 1698170 AND Unique2 < 1702035) AND
       (Unique1 > 1965742 AND Unique1 < 1971745))
UNION ALL
SELECT *
FROM Wisc10M
WHERE ((Unique2 > 1702034 AND Unique2 < 1707060) AND
       (Unique1 > 1965742 AND Unique1 < 1968517))
```

The benefit of the UNION ALL approach is that it may result in shorter index scans for factors that are not adjacent to each other. Furthermore, different indexes may be used for different queries which may lead to faster evaluation in some cases. The downside, on the other hand, is that factors which overlap significantly along one of the attributes can result in repeated scans of the same index portion which the bounding box approach is able to avoid. Both approaches, however, only return actual result tuples to the client.

4.4.2 *Experiment* 4: *Clustered and Unclustered Attributes.* We first consider selections on clustered and unclustered attributes, both of which are indexed. Note that SQL Server is able to apply a very clever query processing

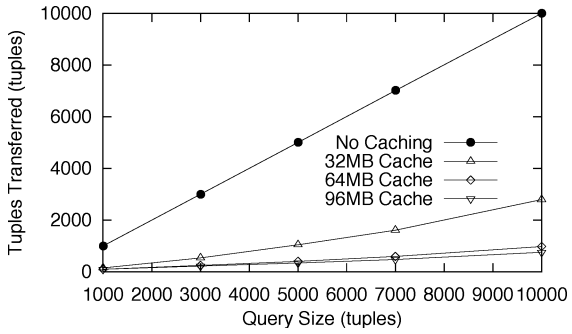


Fig. 11. Tuples transferred for two attributes.

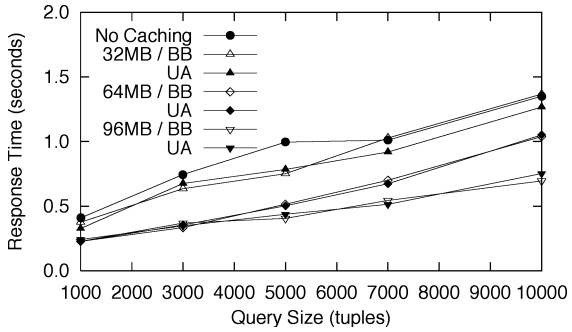


Fig. 12. Response time for clustered and unclustered attributes.

plan in this case because of the index architecture of the server. Each unclustered index stores not only the value of the indexed attribute, but also the value of the attribute used for the clustered index. This second value is typically used only to look up the tuples satisfying constraints on the unclustered attribute; with this workload, however, it is used directly to determine whether the tuples satisfy the constraints on the clustered attribute. Therefore, only tuples that actually satisfy the query are fetched.

Figure 11 shows the tuples transferred across the network. The figure shows that for the larger cache sizes, the semantic caching architecture performs well, saving 90–93% of the network traffic which is similar to the single attribute query workloads. For the small 32MB cache, however, the hot spot does not quite fit in the cache and the savings only range from 85% to 87%. The overhead of the cache (not shown) is similar to the single attribute case and does not affect the hit ratio.

Turning to the query response time, Figure 12 shows the response time of semantic caching with both the bounding box (BB) approach and the UNION ALL (UA) approach (recall that without using one of these approaches, semantic caching may perform much worse than no caching). We first observe that both approaches perform about the same in the figure.[9]

[9]The bumps with small cache and no cache appear to be due to random effects, the drawback of measuring a live system.

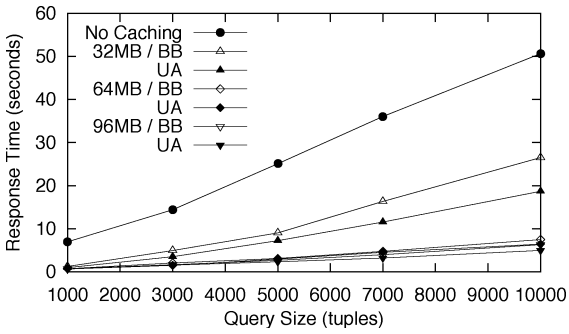


Fig. 13. Response time for two unclustered attributes.

Figure 12 further shows that the time savings are much smaller than the 80–90% savings seen with single attribute queries. For the small cache, in particular, there are no savings at all. Even with the largest cache, only 50–60% savings are shown. The reason for the smaller savings is that, since the server is also able to cache portions of the relation, the queries to the hot spot may be evaluated somewhat efficiently even with the noncaching client. The cold queries, on the other hand, take much more time; since these queries must be evaluated both by the noncaching and semantic caching clients, they weigh heavily in the average.

4.4.3 *Experiment* 5: *Two Unclustered Attributes.* We now consider selections on two unclustered attributes, both of which are indexed. In this case, SQL Server considers the option of scanning both indexes and retrieving only the intersection of the tuples satisfying the constraints on each attribute. This plan, however, is never used in our experiments; instead only one index is scanned and the tuples themselves are then checked against the other constraint.

Figure 13 shows the response time for this workload with both the bounding box (BB) approach and the UNION ALL (UA) approach. In this experiment, the UNION ALL approach performs slightly better, especially for the small cache. This is to be expected as index scans are cheap while fetching tuples is expensive, and the bounding box approach results in more tuples being fetched into memory.

The figure also shows much greater proportional savings than with the clustered and unclustered combination. The primary reason is that all queries are now very expensive, and therefore cold queries do not affect the average as much as before.

4.4.4 *Comparison of Remainder Query Approaches.* We conclude with a brief comparison of the bounding box (BB) and UNION ALL (UA) approaches. With single attribute selections and uniform query sizes, each remainder query has only one factor and both approaches result in the same remainder queries. We have observed that for two attribute queries, the UA approach performs slightly better, although the difference is quite small with large caches.

While not shown here, we also experimented with three attributes. In this case, the BB approach performed slightly better. The reason is that in our

workloads, with three or more attributes, the remainder query factors are more likely to overlap along one or more of the attributes. As mentioned previously, the UA approach can result in repeated scans of the same index portions in this case, leading to inferior performance.

In summary, both approaches have advantages in certain situations, but overall the difference is minimal. Since some systems may not implement the UNION ALL clause, we believe the BB approach to be more generally applicable.

4.5 Multi-Attribute Workloads

As the preceding experiments with two attributes have demonstrated, the relative performance gains of semantic caching are reduced with more attributes. This reduced performance is partly due to the query processing performance of the server which may struggle with some of the remainder queries, but is also partly due to the workload which allows arbitrary overlap between queries and may result in very complicated remainder queries. The goal of the following experiments is to analyze the effects of query overlap and the dimensionality of the queries on the performance of semantic caching.

In Jónsson [1999], we listed several concerns regarding the caching overhead at the client. Although improvements in CPU performance have extended the cache manipulation capabilities of the client, complex cache descriptions are still a concern. In Jónsson [1999], we also listed several concerns regarding the server processing of remainder queries. Although the bounding box approach to submitting remainder queries appears to lead to efficient execution, very complex remainder queries are still a concern. In order to study the effects of these cache and remainder query overheads, this section puts semantic caching through a stress test to determine its performance under increasingly difficult selection workloads.

4.5.1 *Experimental Environment.* The experiments reported in this section use the same system configuration as the experiments of the previous section. This discussion therefore focuses on the database and workloads used in this section.

As before, a Wisconsin benchmark relation [DeWitt 1993] of 10 million tuples is used where each tuple holds 208 bytes of data. In order to be able to use many similar attributes, 3 new random attributes, *R1* through *R3*, were added (the string-valued attribute *String4* was shortened to compensate). Each attribute *Ri* is defined based on *Unique1* by shifting it (with wraparound) by $i * 10\%$.

Each query returns 10,000 tuples and queries are multidimensional hypercubes with uniform sidelengths. The number of attributes (d) is varied from 1 to 5 with the attributes added in the following order: *Unique2* (i.e., the clustered attribute), *Unique1*, *R1*, *R2*, and *R3*. Since the query size is fixed, the queries must select a larger portion of each attribute as more attributes are added.

To model overlap of queries, a partial overlap restriction is defined based on the selectivity of the queries along each attribute. This value, called p, is also varied in each of the experiments. The possible values of p are r, $\frac{r}{2}$, $\frac{r}{4}$, $\frac{r}{16}$, and $\frac{r}{r} = 1$. More formally, $p = \frac{r}{c}$, where c is an integer, means the queries

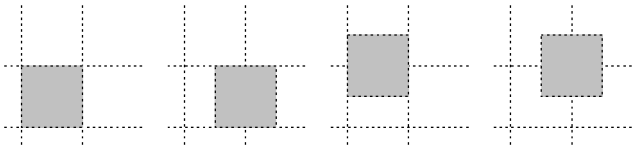


Fig. 14. Four possible two-attribute queries with $p = \frac{r}{2}$.

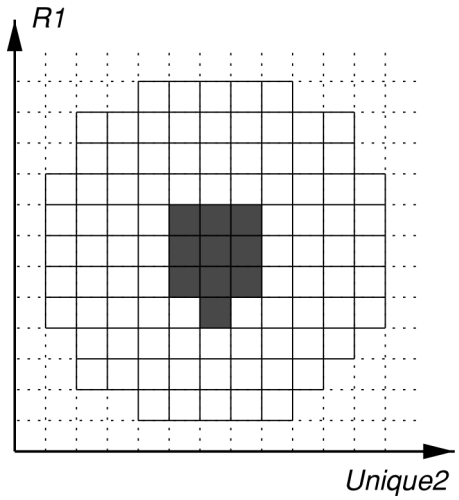


Fig. 15. Ten hot hypercubes with two attributes.

can overlap in any multiple of $\frac{r}{c}$ along each attribute. When $p = r$, there is no partial overlap; any two queries are either identical or they do not intersect. When $p = 1$, there are no restrictions; any query can intersect any other query in arbitrary ways. For example, the workloads of Sections 4.3 and 4.4 correspond to $p = 1$. Figure 14 shows four possible two-attribute queries when $p = \frac{r}{2}$.

As before, a hot-cold distribution is used with the hot queries (90%) drawn from the center of the relation and the cold queries (10%) from the rest of the relation. Assigning a hot spot to the center of the relation is more difficult with many attributes than in the single-attribute case. To determine the hot spot, the relation was broken up into at least 100 adjacent whole hypercubes of the same size as the queries (10,000 tuples). The centerpoint of the relation was found and the 10 hypercubes closest to it (using euclidean distance) were assigned as the hotspot. For example, Figure 15 shows the hot (shaded) hypercubes for two attributes. When queries were generated, hot queries were required to have the centerpoint in one of the hot hypercubes. Additionally, the overlap of queries was determined using the partial overlap restriction described earlier.

Note that, as there are more variations and more attributes, the hot spot size grows, leading to a lower hit ratio. As a rule-of-thumb, the worst case ($p = 1$) can reference 2^{d-1} times the size of the intended hot spot because each query can be offset in each of the d attributes but, in at least one of those attributes, the query will fall in one of the other hot hypercubes. Of course, the access

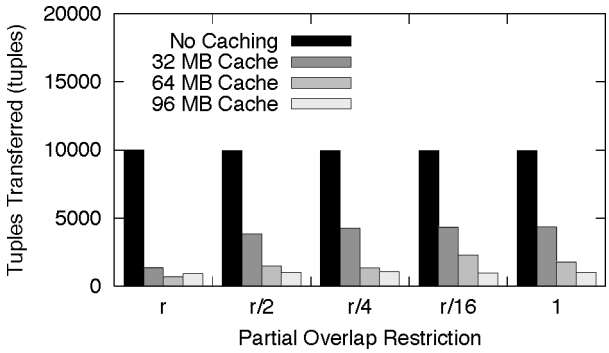


Fig. 16. Tuples transferred for three attributes.

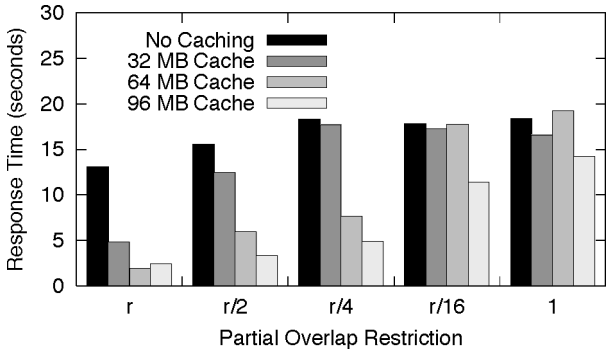


Fig. 17. Response time for three attributes.

frequency within the accessed area is not uniform. We have used cache sizes of 32MB, 64MB, and 96MB as before. Due to the long running times of some of the experiments, we have measured 100 queries for each setup.

4.5.2 *Experiment* 6: *Three or Fewer Attributes.* As was shown in the previous sections, semantic caching performs well in all respects for one and two attributes. In fact, the workloads of the previous experiments are equivalent to the case where $p = 1$ which is the most complex situation examined here.

Turning to the performance of semantic caching for queries with three attributes, Figure 16 shows the savings in tuples transferred for semantic caching compared to the noncaching architecture. The figure shows that when there is no partial overlap ($p = r$), semantic caching performs very well, saving 85–93% of the network traffic. With overlap, the effectiveness of the 32MB cache is significantly reduced, saving only 55–60% of the network traffic. The larger caches, however, still save about 80–90% of the network traffic on the noncaching architecture.

Figure 17, on the other hand, shows the query response time for the same workload. As before, the benefits with no partial overlap are significant, particularly for the larger cache sizes which save 80–85% of the query response time compared to noncaching. For restricted overlap, the performance is also good

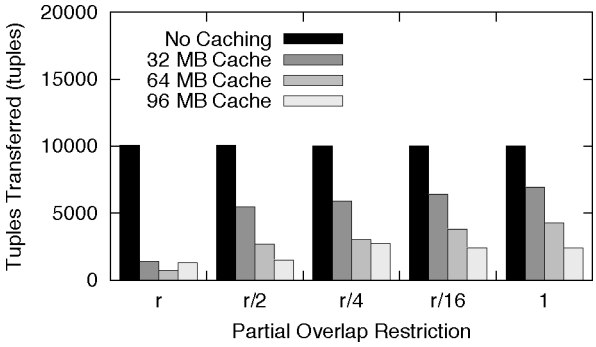


Fig. 18. Tuples transferred for four attributes.

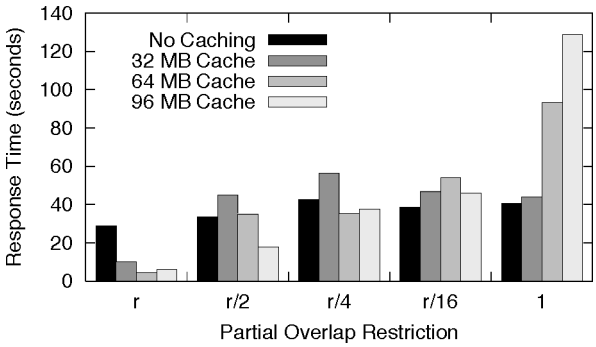


Fig. 19. Response time for four attributes.

for the larger cache sizes, while for significant overlap ($p \geq \frac{r}{16}$), the benefits of semantic caching disappear for the most part.

4.5.3 *Experiment* 7: *Four or More Attributes.* Figure 18 shows the savings in tuples transferred for semantic caching compared to the noncaching architecture when four attributes are used. The figure shows that, when there is no partial overlap ($p = r$), the performance is very good as before. With overlap, the largest cache is still able to maintain a decent hit ratio and save 70–85% of the network traffic. With five attributes (not shown), the same trend is seen.

Turning to the query response time, Figure 19 shows that with no overlap, the response time savings of semantic caching are 55–85%. With any partial overlap, however, semantic caching performs no better than the noncaching architecture, and with significant overlap, it even performs significantly worse than no caching, taking up to 3 times as long to execute queries in the extreme case. With five attributes (not shown), again the same trend is seen.

The excessive running times are partly due to cache management overhead at the client (not shown) which rises to 10 seconds for the case of the 96MB cache. The primary reason for the query response time, however, is the complexity of the remainder queries that are sent to the server. Figure 20 shows the number of factors in the remainder query; checking as many as 75 factors results in

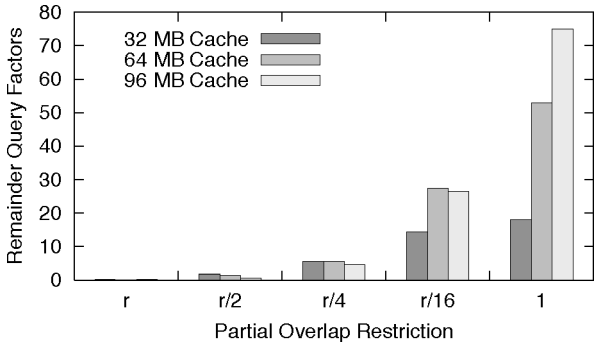


Fig. 20. Remainder query factors for four attributes.

significant CPU overhead at the server which is the primary reason for the inefficient remainder query execution.

4.6 Summary

The results are summarized as follows. With single attribute selection workloads, semantic caching performs very well in terms of network utilization and response time, regardless of the indexing and clustering status of the attribute. With queries over two attributes, remainder queries must be carefully coded to perform well, but, using a standard SQL rewrite, we are able to achieve good response time with semantic caching. With more complex workloads, the network utilization remains good with semantic caching, while the response time savings dissipate due to inefficient query execution of complex remainder queries. With very complex queries, in fact, using no caching is preferable.

5. CONCLUSIONS

In Dar et al. [1996], we proposed a semantic caching architecture which is a variant of logical caching that dynamically caches results of selection queries to single relations and can be applied at application servers, clients, proxies, and even at database servers. This work has since spawned a line of research that focuses on caching query results at a fine granularity (e.g., see Amiri et al. [2003], and Altinel et al. [2003]) and the relationship of semantic caching to Internet and data warehousing environments has become increasingly apparent.

The initial simulation results of Dar et al. [1996] demonstrated the effectiveness of the caching strategy, when used with an idealized server that supported semantic caching very well. In Jónsson [1999], however, we studied the performance of the semantic caching client against a then current commercial server (Informix Universal Server). The results showed that the server did not perform well and, in fact, that it obviated many of the benefits of the semantic caching approach.

In the time that has passed since then, database technology and query optimization have been continuously improving. Furthermore, hardware performance has improved tremendously as well, in particular the CPU performance and memory capacity. Therefore, this article has revisited the

performance and overhead of semantic caching using a modern database server and modern hardware.

Initially, the performance study focused on relatively simple workloads and uncovered some inherent strengths of the semantic caching architecture, including low overhead, insensitivity to clustering, reduced network traffic, and the ability to answer some queries without contacting the server.

With moderately complex workloads, careful coding of remainder queries was required to maintain efficient query processing at the server. We proposed and studied two different SQL-based approaches to remainder query processing and found an approach that submits a bounding-box constraint with the query to work well.

Then the study turned to very complex selection workloads. The results showed that semantic caching utilizes the network very efficiently across a wide range of workloads. The query response time was also excellent for a range of applications, although that range was much more restricted due to cache management overhead and inefficient server processing of requests for data missing from the cache.

There are many interesting directions for future work. Much work is already being done on replacement policies for particular applications such as mobile client caching and extending that work to other applications is a very interesting chellenge. For more traditional applications, such as application server caching for e-commerce, merging the ideas of Altinel et al. [2003] with the range-based techniques of our architecture should yield an architecture suitable for most such workloads. Finally, update handling, in particular in the context of relaxed consistency requirements, is a very important direction for future work.

ACKNOWLEDGMENTS

We owe thanks to the reviewers for their excellent suggestions. Also, we would like to thank the many students at Reykjavík University who have played with the semantic caching code and yielded insight into the capabilities of SQL Server.

REFERENCES

ALTINEL, M., BORNHÖVD, C., KRISHNAMURTHY, S., MOHAN, C., PIRAHESH, H., AND REINWALD, B. 2003. Cache tables: Paving the way for an adaptive database cache. In *Proceedings of the Conference on Very Large Data Bases (VLDB)*, Berlin, Germany. J. C. Freytag, P. C. Lockemann, S. Abiteboul, M. J. Carey, P. G. Selinger, and A. Heuer, Eds. Morgan Kaufmann.

AMIRI, K., PARK, S., TEWARI, R., AND PADMANABHAN, S. 2003. DBProxy: A dynamic data cache for web applications. In *Proceedings of the IEEE Conference on Data Engineering*. Bangalore, India. U. Dayal, K. Ramamritham, and T. M. Vijayaraman, Eds. IEEE Computer Society.

ARINBJARNAR, M., ÞÓRSSON, B., AND JÓNSSON, B. Þ. 2006. Performance of semantic caching revisited. Tech. rep. RUTR-CS06002, Reykjavík University.

CANDAN, K. S., LI, W.-S., LUO, Q., HSIUNG, W.-P., AND AGRAWAL, D. 2001. Enabling dynamic content caching for database-driven web sites. Santa Barbara, CA. In *Proceedings of the ACM SIGMOD Conference on Management of Data*, W. G. Aref, Ed. ACM.

CAREY, M. J., FRANKLIN, M. J., LIVNY, M., AND SHEKITA, E. J. 1991. Data caching trade-offs in client-server DBMS architectures. In *Proceedings of the ACM SIGMOD Conference on Management of Data*. Denver, CO. ACM.

CAREY, M. J. AND KOSSMANN, D. 1997. On saying "enough already!" in SQL. In *Proceedings of the ACM SIGMOD Conference on Management of Data*. Tucson, AZ, J. Peckham, Ed.

CHEN, L., RUNDENSTEINER, E. A., AND WANG, S. 2002. XCache: A semantic caching system for XML queries. In *Proceedings of the ACM SIGMOD Conference on Management of Data*. Madison, WI. M. J. Franklin, B. Moon, and A. Ailamaki, Eds. ACM.

CHIDLOVSKII, B. AND BORGHOFF, U. M. 2000. Semantic caching of web queries. *VLDB J. 9*, 1, 2–17.

CHIDLOVSKII, B., RONCANCIO, C., AND SCHNEIDER, M.-L. 1999. Semantic cache mechanism for heterogeneous web querying. *Comput. Netw. 31*, 11–16, 1347–1360.

DAR, S., FRANKLIN, M. J., JÓNSSON, B. Þ, SRIVASTAVA, D., AND TAN, M. 1996. Semantic data caching and replacement. In *Proceedings of the Conference on Very Large Data Bases (VLDB)*. Bombay, India. T. M. Vijayaraman, A. P. Buchmann, C. Mohan, and N. L. Sarda, Eds. Morgan Kaufmann.

DELIS, A. AND ROUSSOPOULOS, N. 1992. Performance and scalability of client-server database architectures. In *Proceedings of the Conference on Very Large Data Bases (VLDB)*. Vancouver, Canada. L.-Y. Yuan, Ed. Morgan Kaufmann.

DELIS, A. AND ROUSSOPOULOS, N. 1998. Techniques for update handling in the enhanced client-server DBMS. *IEEE Trans. Knowl. Data Engin. 10*, 3 (May/June), 458–476.

DESHPANDE, P. M., RAMASAMY, K., SHUKLA, A., AND NAUGHTON, J. F. 1998. Caching multidimensional queries using chunks. In *Proceedings of the ACM SIGMOD Conference on Management of Data*. Seattle, WA. L. M. Haas and A. Tiwary, Eds. ACM.

DEWITT, D. J. 1993. The Wisconsin benchmark: Past, present, and future. In *The Benchmark Handbook for Database and Transaction Processing Systems*, J. Gray, Ed. Morgan-Kaufmann Publishers, San Mateo, CA.

DEWITT, D. J., FUTTERSACK, P., MAIER, D., AND VELEZ, F. 1990. A study of three alternative workstation-server architectures for object oriented database systems. In *Proceedings of the Conference on Very Large Data Bases (VLDB)*. Brisbane, Australia. D. McLeod, R. Sacks-Davis, and H.-J. Schek, Eds. Morgan Kaufmann.

FRANKLIN, M. J. 1996. *Client Data Caching*. Kluwer Academic Press, Boston, MA.

FRANKLIN, M. J. AND ZDONIK, S. B. 1997. A framework for scalable dissemination-based systems. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems and Languages (OOPSLA)*. Atlanta, GA. ACM.

GODFREY, P. AND GRYZ, J. 1999. Answering queries by semantic caches. In *Proceedings of the 10th International Conference on Database and Expert Systems Applications (DEXA)*. Florence, Italy. T. J. M. Bench-Capon, G. Soda, and A. M. Tjoa, Eds. Springer.

GRAEFE, G. 1994. Volcano–an extensible and parallel query evaluation system. *IEEE Trans. Knowl. Data Engin. 6*, 1 (Feb.), 120–135.

GUPTA, A. AND MUMICK, I. S. 1995. Maintenance of materialized views: Problems, techniques and applications. *IEEE Data Engin. Bull. 18*, 2 (June), 3–18.

HRISTIDIS, V. AND PETROPOULOS, M. 2002. Semantic caching of XML databases. In *Proceedings of the 5th International Workshop on the Web and Databases (WebDB)*. Madison, WI. M. F. Fernandez and Y. Papakonstantinou, Eds. Informal Proceedings.

JÓNSSON, B. Þ. 1999. Application-oriented buffering and caching techniques. Ph.D. thesis, University of Maryland, College Park, MD.

KELLER, A. AND BASU, J. 1996. A predicate-based caching scheme for client-server database architectures. *VLDB J. 5*, 1, 35–47.

KEMPER, A. AND KOSSMANN, D. 1994. Dual-buffering strategies in object bases. In *Proceedings of the Conference on Very Large Data Bases (VLDB)*. Santiago de Chile, Chile. J. B. Bocca, M. Jarke, and C. Zaniolo, Eds. Morgan Kaufmann.

LEE, D. AND CHU, W. W. 2001. Towards intelligent semantic caching for Web sources. *J. Intell. Inform. Syst. 17*, 1, 23–45.

LEVY, A. Y., MENDELZON, A. O., SAGIV, Y., AND SRIVASTAVA, D. 1995. Answering queries using views. In *Proceedings of ACM SIGMOD/SIGACT Conference on Principles of Database Systems (PODS)*. San Jose, CA. ACM.

LUO, Q., KRISHNAMURTHY, S., MOHAN, C., PIRAHESH, H., WOO, H., LINDSAY, B. G., AND NAUGHTON, J. F. 2002. Middle-tier database caching for e-business. In *Proceedings of the ACM SIGMOD Conference on Management of Data*. Madison, WI. M. J. Franklin, B. Moon, and A. Ailamaki, Eds. ACM.

MOHAN, C. 2002. Application servers and associated technologies (tutorial). In *Proceedings of the ACM SIGMOD Conference on Management of Data*. Madison, WI. M. J. Franklin, B. Moon, and A. Ailamki, Eds. ACM.

REN, Q. AND DUNHAM, M. H. 1999. Using clustering for effective management of a semantic cache in mobile computing. In *Proceedings of the ACM International Workshop on Data Engineering for Wireless and Mobile Access (MobiDE)*. Seattle, WA. ACM.

REN, Q. AND DUNHAM, M. H. 2000. Using semantic caching to manage location dependent data in mobile computing. In *Proceedings of the 6th Annual International Conference on Mobile Computing and Networking (MobiCom)*. Boston, MA. ACM.

ROUSSOPOULOS, N. 1998. Materialized views and data warehouses. *ACM SIGMOD Record 27*, 1 (March), 21–26.

SCHWARTZ, E. June 10, 2002. IBM hints at taking wi-fi access national. InfoWorld.`infoworld.com`.

STONEBRAKER, M. 1975. Implementation of integrity constraints and views by query modification. In *Proceedings of the ACM SIGMOD Conference on Management of Data*. San Jose, CA. W. F. King, Ed. ACM.

THE TIMESTEN TEAM. 2002. Mid-tier caching: The TimesTen approach. In *Proceedings of the ACM SIGMOD Conference on Management of Data*. Madison, WI. M. J. Franklin, B. Moon, and A. Ailamaki, Eds. ACM.

WAN, H., HAO, X.-W., ZHANG, T., AND LI, L. 2004. Semantic caching services for data grids. In *Proceedings of the 3rd International Conference on Grid and Cooperative Computing (GCC 04)*. Wuhan, China. H. Jin, Y. Pan, N. Xiao, and J. Sun, Eds. Springer.

ZHENG, B., LEE, W.-C., AND LEE, D. L. 2004. On semantic caching and query scheduling for mobile nearest-neighbor search. *Wirel. Netw. 10*, 6, 653–664.

Received December 2002; revised May 2004; accepted May 2005