REYKJAVÍK UNIVERSITY
HÁSKÓLINN Í REYKJAVÍK

Performance of Semantic Caching Revisited

María Arinbjarnar, Bjarnsteinn Þórsson, Björn Þór Jónsson

RUTR-CS06002 — September 2006

Reykjavík University - Department of Computer Science

Technical Report

ISSN 1670-5777

Performance of Semantic Caching Revisited

María Arinbjarnar*, Bjarnsteinn Þórsson*, Björn Þór Jónsson*

Technical Report RUTR-CS06002, September 2006

Abstract: A caching architecture for database clients called *semantic caching* was proposed in 1996 and evaluated extensively against a then-current relational database server in 1998. While semantic caching was shown to perform well for a range of workloads, the relational server was not well equipped to handle complex remainder queries. Since then, hardware has become increasingly faster with considerable increases in memory size and caching capabilities. Additionally, there have also been significant performance improvements in relational database systems, in particular query optimization. In this report we first review some recent related work. We then evaluate the semantic caching architecture against modern hardware and software, and propose and evaluate two new approaches to query execution at the relational server. Our conclusion is that despite the hardware and software performance improvements which have reduced query processing time very significantly, complex query workloads still present significant difficulties for the semantic caching architecture.

Keywords: Semantic caching; Performance evaluation.

(Útdráttur: næsta síða)

* Reykjavík University, Kringlan 1, IS-103 Reykjavík, Iceland. {maria01|bjarnsteinn|bjorn}@ru.is.

HÁSKÓLINN Í REYKJAVÍK
REYKJAVÍK UNIVERSITY

Afköst fyrirspurnaháðs skyndiminnis endurmetin

María Arinbjarnar, Bjarnsteinn Þórsson, Björn Þór Jónsson

Tækniskýrsla RUTR-CS06002, September 2006

Útdráttur: Fyrirspurnaháð skyndiminni er arkitektúr fyrir skyndiminni gagnasafnsbiðlara, sem fyrst var sett fram 1996. Afköst þess voru mæld ítarlega árið 1998, og var þá notaður nýlegur gagnasafnsmiðlari. Þótt afköst fyrirspurnaháðs skyndiminnis væru góð fyrir mörg notkunartilfelli, var gagnasafnsmiðlarinn ekki vel búinn til þess að ráða við flóknar fyrirspurnir. Síðan þá hafa afköst vélbúnaðar sífellt aukist, ásamt því að minni hefur orðið stærra og skyndiminnisgeta því öflugri. Að auki hafa afköst gagnasafnsmiðlara einnig aukist, og sér í laga hefur fyrirspurnabesturum farið mjög fram. Í þessari skýrslu er fyrst fjallað um nýlegar rannsóknir sem tengjast fyrirspurnaháðu skyndiminni. Svo eru afköst þess metin á móti nútímalegum vél- og hugbúnaði, og tvær nýjar aðferðir settar fram við úrvinnslu fyrirspurna á gagnasafnsmiðlurum. Niðurstaða okkar er sú að þrátt fyrir að framfarir á sviði vél- og hugbúnaðar hafi minnkað keyrslutíma fyrirspurna umtalsvert, ráði fyrirspurnaháð skyndiminni ekki enn vel við flóknar fyrirspurnir.

Lykilorð: Fyrirspurnaháð skyndiminni; Mat á afköstum.

(Abstract: previous page)

Contents

1 Introduction

Semantic Caching is a caching schema proposed in 1996 by Dar, Franklin, Jónsson, Srivastava and Tan [DFJ+96]. Semantic caching uses the semantics of queries to intelligently determine whether they can be satisfied from the client cache, either in whole or partially, and constructs a remainder query to retrieve from the database server any part of the user query that cannot be satisfied from the cache. The cache is managed by dividing the contents into semantic regions determined by the data retrieved and the user queries issued.

Jónsson evaluated semantic caching extensively against a then-current relational database server in 1998 [Jón99]. While semantic caching was shown to perform well for a range of workloads, the relational server was not well equipped to handle complex remainder queries. Since then, hardware has become increasingly faster with considerable increases in memory size and caching capabilities. Additionally, there have also been significant performance improvements in relational database systems, in particular query optimization.

This report documents an undergraduate independent study performed at the Reykjavík University Database Lab. The objective of the study was to update the performance evaluation of the semantic caching architecture against modern hardware and software to see how it performs relative to previous experiments. Furthermore, we proposed and evaluated two new approaches, based on standard SQL constructs, to remainder query execution at the relational server.

The performance evaluation consisted of three parts. In the first part, we studied single-attribute workloads in order to understand issues of clustering and indexing. In the second part, we studied workloads with selections over two attributes, in order to understand the performance of our proposed remainder query execution strategies. In the third and final part, we studied the performance of semantic caching with multi-attribute selections, to understand the effects of complex workloads on query response time and cache processing. Overall, our conclusion is that despite the hardware and software performance improvements, which have reduced query processing time very significantly, complex query workloads still present significant difficulties for the semantic caching architecture.

The remainder of this report is organized as follows. Section 2 reviews some recent related work. Section 3 describes our experimental environment. Sections 4 through 6 describe the performance experiments and their results.[1] Finally, Section 7 summarizes the performance results and gives our conclusions.

2 Related Work

The last decade has seen a great increase in diverse kinds of database driven web services. This increase calls for efficient, secure and accurate data handling to and from the database.

[1] The results presented in Sections 4 through 6 have appeared in [JAÞ+06].

Fortunately hardware has become cheaper and hence client processing power is growing fast and getting more robust. Most often the client is only using a small portion of its hardware capabilities, which opens up the possibility of caching data in the client and running many of the necessary algorithms on the client instead of the database server, thus freeing up the database server and the network for other traffic.

The present architecture usually consists of at least the three following layers (e.g., see [LKM+02]):

Database layer: The database server is typically a powerful, multiprocessor machine with huge memory and disk space.

Application layer: The application layer is where the application logic resides. This layer can sometimes be divided into sub-layers to isolate individual activities, such as network dispatching, caching or locking mechanisms. The application layer usually resides on a separate machine(s) from the database server, with a fast network connection between the database server and application server.

Presentation layer: The presentation layer runs entirely on the client; typically (but not necessarily) on a commercial Internet browser.

There are two main opportunities for caching: application layer caching and client caching. Caching at the application layer is convenient because it is relatively simple to implement and maintain as it is close to the application logic. Application layer caching is also very fast and can take full advantage of the actual database structure. It effectively relieves the excess throughput of the database server [LKM+02, Tim02].

Client caching is also a promising caching technique; it effectively moves the overhead of performing calculations and storing the actual cached data to the user hardware. This saves both hardware costs for the web service company in question and provides some added bonuses for the user. The data is cached at the client computer and the algorithm to decide whether the client needs to contact the application layer or the database server for data handling also runs on the client. This effectively compensates for slow or shaky network connections; the user can live without an established connection to the application layer for some periods of time without necessarily needing to stop working. This, of course, is very useful in today's mobile computers, which tend to have an unstable connection.

According to Amiri et al. [APTP03], the primary concerns in this environment are:

Database independence: As applications may access various back-end databases, which can be running on different database management systems from different vendors, it is necessary to have caching techniques independent of database management systems.

Self-management: The cache needs to be able to adapt dynamically to workloads and available resources to eliminate the need for costly administration.

Fast query matching: With increased cache sizes, it is important that the cache be able to efficiently determine whether a query can be satisfied locally or not.

Efficient space management: Necessary to effectively benefit from the caching potential.

Consistency: So that effective scaling can be achieved without compromising correctness of the application.

There is an obvious need for data consistency. In many e-commerce applications data integrity can be of paramount importance and thus we need a robust locking mechanism to ensure consistency without affecting performance and scalability too much. Such a locking mechanism needs to use either invalidation or propagation. With propagation, the database server can transmit changes fast and efficiently to individual clients, but it runs the risk of using too much throughput when traffic is high. This is not true for the invalidation process, since information is only re-fetched when it is re-used. Carey et al. [CFLS91] propose to gain the pros of both propagation and invalidation by constructing a heuristic dynamic locking mechanism called optimistic-dynamic two-phased locking mechanism. On the whole, the proposed mechanism is more efficient than either the invalidation or propagation locking mechanisms.

2.1 Data Warehousing

Data warehousing has been somewhat different from other areas of caching due to the infrequency of data updating, although updates are becoming more common. The queries are also frequently asking for averages, sums, counts, etc. from large quantities of data, which calls for costly calculations that return small sets of data. The data warehouse environment also calls for short response times. This is the perfect setting for intelligent caching that can take place at more than one level.

Scheuermann et al. (in [SSV96]) propose WATCHMAN (WArehouse inTelligent CacHe MANager). WATCHMAN evaluates rate of reference, size for each retrieved set and execution cost of the associative query, by employing a special profit metric that aims to reduce query response time by minimizing execution cost. WATCHMAN uses also a complementary cache admission algorithm, as not all data retrieved needs to be stored in the cache, especially if it would push out other query results that are frequently accessed and require costly calculations. The WATCHMAN cache replacement algorithm shows cost savings by a factor of three. But data warehousing is not descriptive for other caching environments, since it is a very closed and controlled environment, while other caching applications do not have that luxury.

2.2 *e*-Commerce

In e-commerce there are very different concerns from that of data warehousing. E-commerce applications run on one or more large database servers that are accessed through application servers. The data stored is dynamically changing both from input from users and also from the actual companies running the applications in question. This calls for robust and intelligent caching techniques.

An interesting caching technique that is frequently used in e-commerce is the caching of tables, which may be implemented as replication of data. Views are also an interesting way to cache data; a fairly obvious quality of using views is that it keeps the relational context of the data. This is also the main complication of using views; view caching makes it necessary to maintain and update the data in a relational context and raises questions on when and how this is best accomplished. Many existing solutions use production rules to do view maintenance, with deductive algorithms in database systems.

An alternative solution is the ADMS system proposed in [DR92], which maintains simple materialized views, called View-Caches, in a multi-database environment. ADMS uses materialized views in query optimization and addresses questions of caching, buffering, access paths, etc. As Pottinger and Levy [PL00] discuss the problem becomes even more profound when adding the complication of keeping the cache dynamic and flexible, as that requires the views to be database system independent; the views need to have a technique for handling relational data that is independent of individual database system versions. This maintenance issue touches on other closely related issues such as data integration, query optimization, and the maintenance of physical data independence [PL00].

An interesting algorithm is the MiniCon algorithm proposed in [PL00], which attempts to solve this problem. It begins in a similar way to a bucket algorithm: when MiniCon finds sub-goals in a view that correspond to sub-goals in the query, it starts to consider join predicates and variables to be able to map the sub-goals of the view to that of the query. The MiniCon algorithm scales well and effectively shows that views can be used for caching large web applications.

Another very efficient caching method for the application layer is the TimesTen caching technique; it caches table fragments in the application layer. The table fragments are described with extended SQL syntax, forming tables in the application layer that are handled by a relational database handler [Tim02].

Altinel et al. [ABK+03] proposed a system called DBcache, which defines new database objects called Cache Tables and uses DB2's distributed query processing for application layer caching. An optimizer can then decide on whether to execute the query at the local database or at the back end server. The optimizer can also choose to execute parts of the query locally and the rest at the backend server. The system is able to support static, prearranged data caching, and can also rewrite input queries into an appropriate form that can process dynamic subsets at runtime. The system constructs special query plans that have three parts: probe query, local query, and remote query. The probe query executes first and determines whether to use remote and/or local queries. The dynamic caching adds only limited overhead in the cache database and it is clear that the benefits of better response time, higher scalability and availability on the Internet are worth the effort [ABK+03].

DBProxy is another caching technique, proposed by Amiri et al. [APTP03], which uses materialized views on edge servers. DBProxy is implemented using the IBM Websphere Edge Server, and uses a query evaluator to determine an access hit or miss by invoking a query matching module. The query matching module uses the query constraints and other clauses as its arguments, it decides whether to satisfy the query locally. If the query can

not be satisfied locally then the query matching module rewrites the query for the back-end server. DBProxy uses a resource manager to keep score of hit rates, response times, and such, and a consistency manager to maintain cache consistency.

Candan et al. [CLL+01] discuss how to respond to dynamically changing web pages. They propose an intelligent invalidation technique based on the database content, which enables caching of dynamically generated web pages.

2.3 Mobility and Location Dependency

Mobile clients typically have much smaller memory than a PC and thus what to cache becomes more important than in other applications. Caching is even more useful for mobile clients than in stationary computers because wireless links are relatively unreliable and limited which makes it a necessity to have previously stored, relative or necessary information in memory, for the given application to run efficiently and correctly. Additionally the computer is traveling between locations and is thus going in and out of specific service areas and will need to continuously update specific information. Ren and Dunham [RD00] discuss the benefits of semantic caching over page caching in location dependent mobile clients with emphasis on these special concerns of mobility.

When dealing with location dependent mobile computers it is good to use a replacement algorithm like FAR (Furthest Away Replacement), discussed in [DFJ+96, MdCCC04, RD99, RD00, ZLL04], that uses a direction factor to determine what to replace next. Thus the algorithm determines in what direction the mobile unit is heading, e.g. by using GPS information, and replaces first whatever is farthest away and behind the client.

In semantic caching, semantic regions are used to determine which data will be discarded from the cache next. Semantic caching can use LRU or semantic replacement policies [DFJ+96]. Ren and Dunham [RD99] propose an addition to this replacement scheme. Instead of storing every new query result in the cache they only store query results that are semantically related to the formerly cached data. This successfully avoids the storing of any cold queries in the cache, an important thing in mobility due to the generally small cache size.

Another interesting investigation into nearest neighbourhood searches in mobility is done by Zheng et al. [ZLL04], where an index is used by the server, based on a Voronoi Diagram, to support nearest neighbourhood querying of stationary service by mobile users, and semantic caching is used to enhance the access efficiency of such service. A Voronoi Diagram records information about the closest regions corresponding to a set of geometric points. Voronoi Diagram has a high maintenance and construction cost, especially for high dimensions, but location based services have a two-dimensional search space and are infrequently updated so this is not an issue here.

Manica et al. [MdCCC04] discuss in some detail the complications associated with semantic caching in mobile units and propose a cache replacement policy ASCR (Adaptive Semantic Cache Replacement) that uses previous movements of the client to calculate possible future movements.

2.4 Other Applications

There are some other caching applications that can put semantic caching to good use. Stuckenschmidt [Stu04] has an interesting discussion on using semantic caching as a high level optimization technique for RDF (Resource Description Framework) querying to supplement existing work on lower level techniques. The similarity of semantic caching to RDF queries is utilized to determine the costs of modifying the results of a previous query into the result for the actual query.

Luo and Xue [LX04] propose a template based proxy to handle user defined functions, so that the proxy can answer previously cached data in collaboration with the original website using partial semantic caching as the main caching schema, and show that in practice this is an efficient caching schema for websites that use a great amount of user defined functions.

3 Experimental Environment

The objective of the study reported here was to update the performance evaluation of the semantic caching architecture against modern hardware and software to see how it performs relative to previous experiments. In addition to the hardware and software improvements, however, database sizes have also increased significantly in this period and hence the databases and workloads have been scaled up by an order of magnitude. In this section we describe the experimental environment used in this report.

3.1 System Configuration

In the following, the semantic caching client prototype is compared against a non-caching client, which also uses ODBC to connect to the database server. Clients are run on a 1.5 GHz Intel Pentium PC with 512 MB of memory and an 18.6 GB disk, running Windows XP Professional Workstation SP2. The server is run on a 1.6 GHz Intel Pentium PC with 512 MB of memory and an 18.6 GB 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 2.5 GB and stored in a regular file. The server is used as it was installed; i.e., no performance tuning has been done. Both machines are used exclusively for the experiments. The network connecting the two machines is a 100 Mbit isolated Ethernet.

3.2 Database and Workloads

The database used in the experiments in this section is based on the Wisconsin benchmark [DeW93]. It contains a single relation of 10 million tuples, called *Wisc10M*.[2] Each

[2] 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 6 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.

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

Table 1: Workload Parameters and Settings

tuple holds 208 bytes of data, for a total of over 2 GB 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 with a B^+-tree index, but is completely unclustered; *Unique3* is both unindexed and unclustered.[3] The candidate keys have values from 0 to 9,999,999.

Table 1 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 above. A fixed portion of the queries (*Skew*) has the center-point 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 3.4.

3.3 Metrics

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 data-point 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 non-caching client, however, the performance of each query is independent of the previous queries and the metrics converge much faster than with caching. For the non-caching 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, i.e., they were neither displayed to a screen nor written to a file.

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

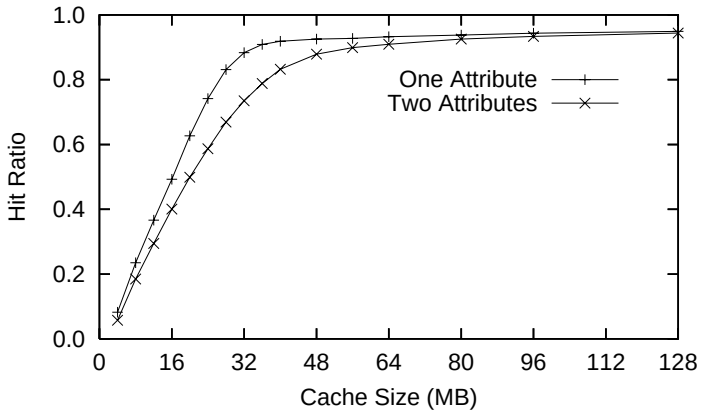

Figure 1: Cache hit ratio as a function of cache size

Note that in all of the experiments in this report, 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.

3.4 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 1 shows the hit ratio of the cache, as the cache size is varied from 4 MB to 128 MB. 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 30 MB. For queries over two attributes, however, the hot spot requires much more memory, and is not contained until around 60 MB. This is because the query workload is created such that the *center point* 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: 32 MB, 64 MB and 96 MB.

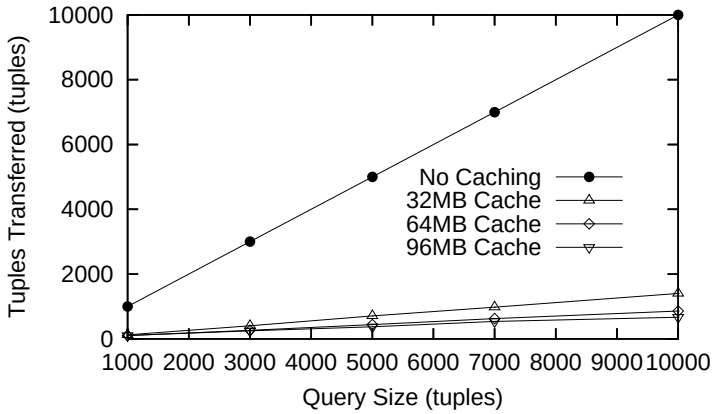

Figure 2: Tuples transferred for clustered attribute

4 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.

4.1 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 2 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 non-caching, reducing network traffic by 85–93%.

Figure 3, 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 3 shows that semantic caching provides significant savings over non-caching, ranging from 81% to 92%. 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.[4] 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, as cache processing (not shown) always requires less than 15 milliseconds for the semantic cache. In

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

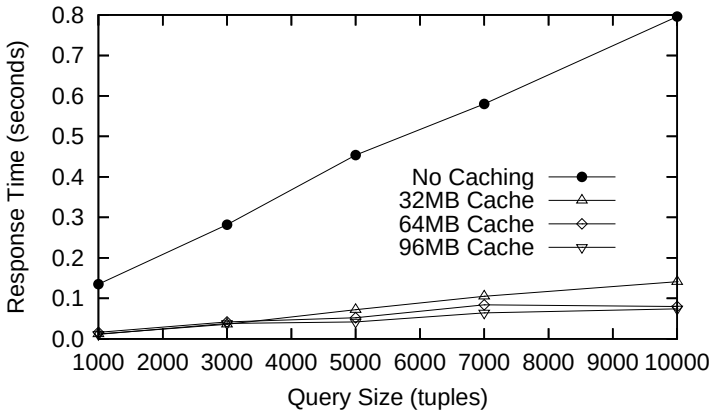

Figure 3: Response time for clustered attribute

some cases, however, the client is able to completely answer the queries from cache and avoid contact with the server resulting in extremely efficient query processing. The percentage of queries answered completely from 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%.

4.2 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 4 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.

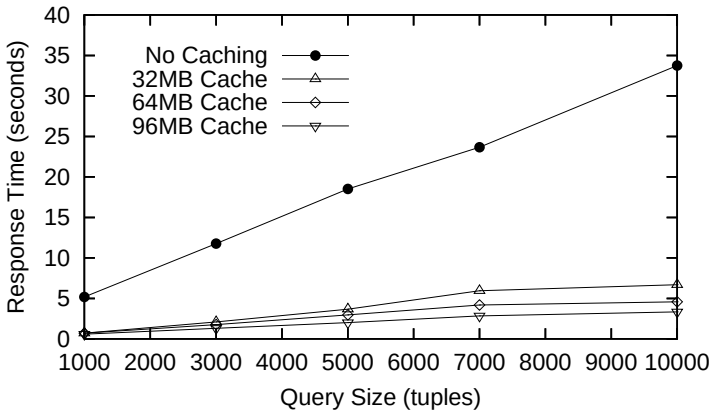

Figure 4: Response time for unclustered attribute

4.3 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, as 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 cache.

Figure 5 shows the fraction of queries that generate a remainder query to the server (all non-caching queries are considered remainder queries in this figure). Overall, Figure 5 shows that with a 32 MB cache, 72–81% of all queries may be satisfied entirely from cache, while with the larger cache sizes, 87–91% may be answered from cache. As explained above, response time is improved proportionately.

Figure 5 shows a very interesting effect, however, as the performance is slightly improved with larger queries for larger cache sizes, while for the smaller 32 MB cache, performance is noticably worse with larger queries. The explanation for the worse performance of the 32 MB 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 lower hit ratio and higher ratio of remainder queries. For the larger cache sizes, cold data evicts other cold data and performance is improved as expected.

5 Double Attribute Experiments

In the experiments above, we saw that semantic caching performs very well for single attribute queries. In many cases, however, more than one attribute is included in the query

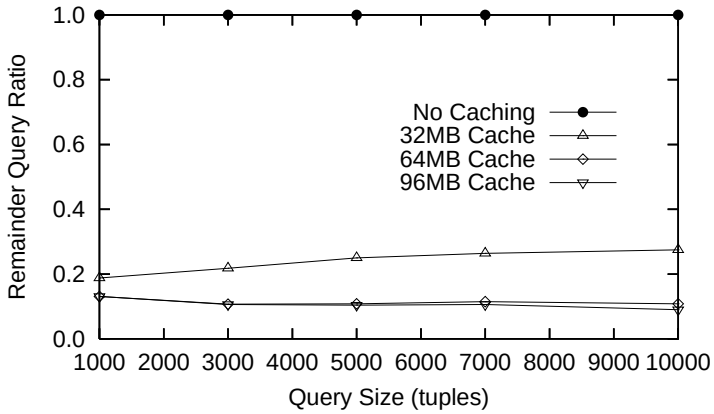

Figure 5: Fraction of queries resulting in a remainder query for unindexed attribute

constraints. The workloads of this sub-section therefore considers 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 two-fold. First, as mentioned above, the "warm spot" around the hot spot extends further with two attribute workloads than with a single attribute, requiring a larger cache to achieve 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.

5.1 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.[5] The lack of this feature is understandable, 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))
```

[5] 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.

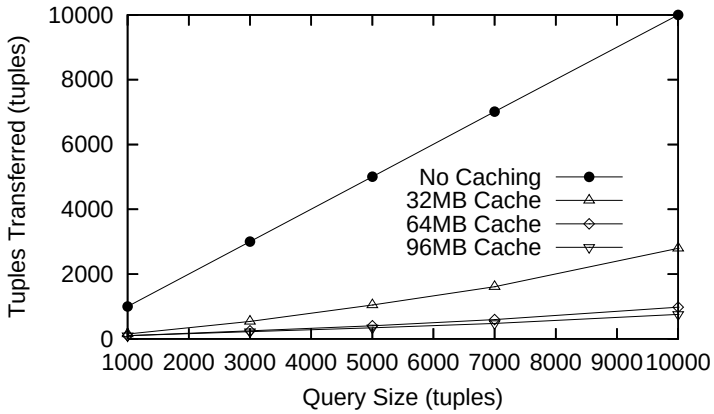

Figure 6: Tuples transferred for two attributes

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 may 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.

5.2 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 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 6 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 32 MB 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.

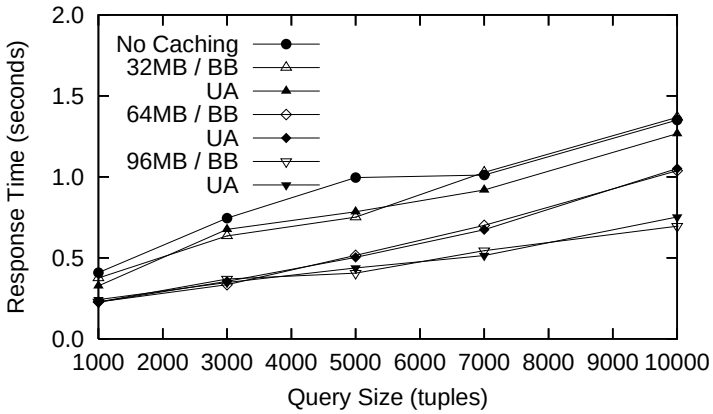

Figure 7: Response time for clustered and unclustered attributes

Turning to the query response time, Figure 7 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.[6]

Figure 7 furthermore shows that the time savings are much smaller than the 80–90% savings seen with single attribute queries. For the small cache, in particular, no savings are seen at all. Even with the largest cache, only 50–60% savings are seen. 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 non-caching client. The cold queries, on the other hand, take much more time; since these queries must be evaluated both by the non-caching and semantic caching clients, they weigh heavily in the average.

5.3 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 8 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 ap-

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

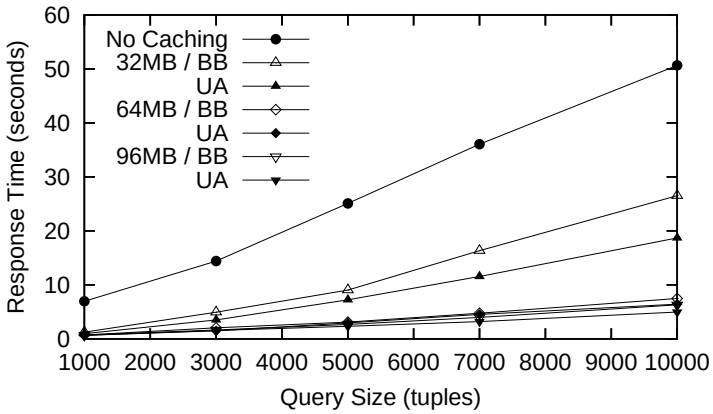

Figure 8: Response time for two unclustered attributes

proach 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.

5.4 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 above, the UA approach may 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.

6 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.

Jónsson [Jón99] 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. Jónsson [Jón99] 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.

6.1 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 [DeW93] 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 wrap-around) by $i * 10\%$.

Each query returns 10,000 tuples and queries are multidimensional hyper-cubes with uniform side-lengths. The number of attributes (d) is varied from 1 to 5 with the attributes being 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 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 and 5 correspond to $p = 1$. Figure 9 shows four possible two-attribute queries when $p = \frac{r}{2}$.

As before, a hot-cold distribution is used with the hot queries (90%) being drawn from the center of the relation and the cold queries (10%) from the rest of the relation. Assigning

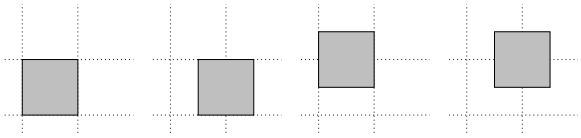

Figure 9: Four possible two-attribute queries with $p = \frac{r}{2}$

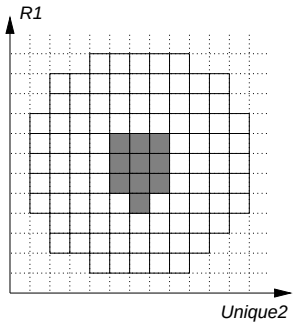

Figure 10: 10 hot hyper-cubes with two attributes

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 hyper-cubes of the same size as the queries (10,000 tuples). The "center-point" of the relation was found and the 10 hyper-cubes closest to it (using euclidean distance) were assigned as the hot-spot. For example, Figure 10 shows the hot (shaded) hyper-cubes for two attributes. When queries were generated, hot queries were required to have the center-point in one of the hot hyper-cubes. Additionally, the overlap of queries was determined using the "partial overlap restriction" described above.

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

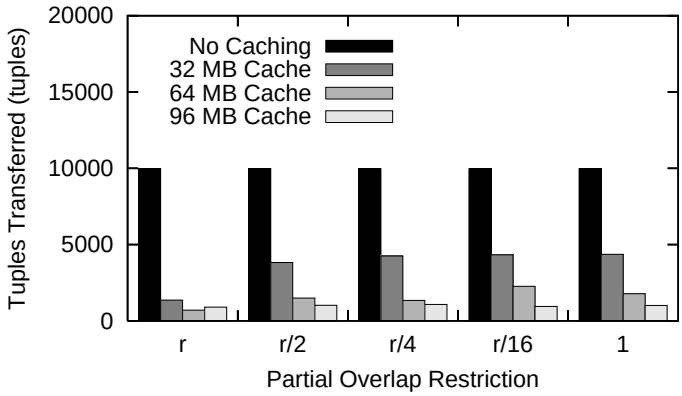

Figure 11: Tuples transferred for three attributes

6.2 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 11 shows the savings in tuples transferred for semantic caching compared to the non-caching 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 32 MB 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 non-caching architecture.

Figure 12, 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 non-caching. For restricted overlap, the performance is also good for the larger cache sizes, while for significant overlap ($p \geq \frac{r}{16}$) the benefits of semantic caching disappear for the most part.

6.3 Experiment 7: Four or More Attributes

Figure 13 shows shows the savings in tuples transferred for semantic caching compared to the non-caching 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,

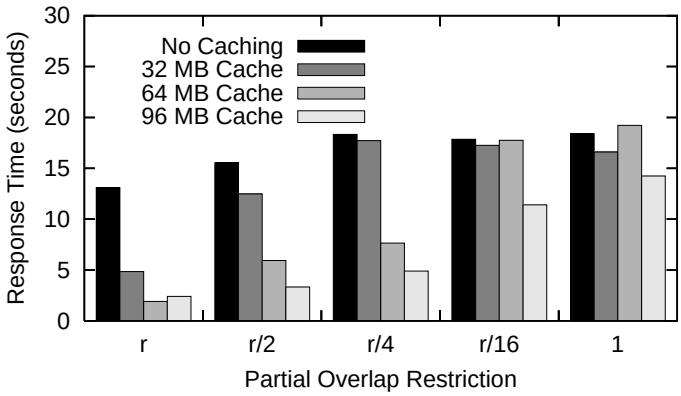

Figure 12: Response time for three attributes

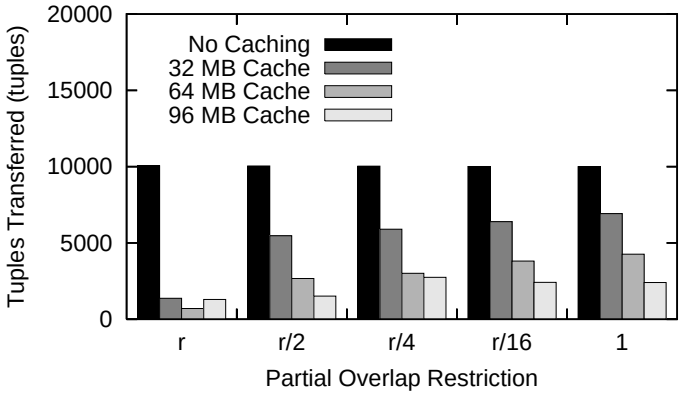

Figure 13: Tuples transferred for four attributes

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 14 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 non-caching architecture and with significant overlap it

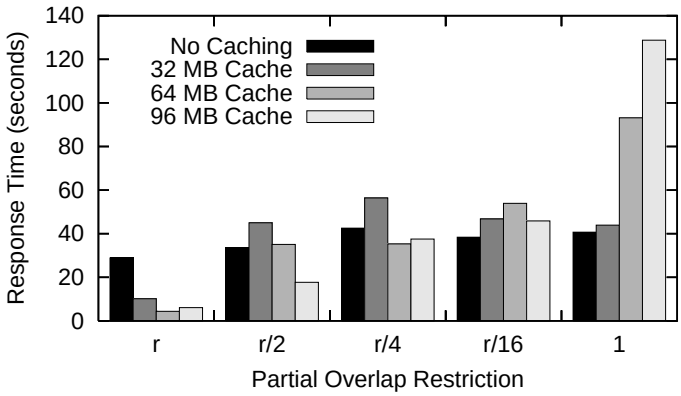

Figure 14: Response time for four attributes

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 96 MB cache. The primary reason for the query response time, however, is the complexity of the remainder queries that are sent to the server. Figure 15 shows the number of factors in the remainder query; checking as many as 75 factors results in significant CPU overhead at the server, which is the primary reason for the inefficient remainder query execution.

7 Summary and Conclusions

The current perfomance results may be 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 were 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.

When comparing the results presented in this report to those of [Jón99] we observe that, overall, the response time is generally improved by more than an order of magnitude, despite scaling up the database size and query workloads by an order of magnitude. This performance improvement is consistent with the performance improvements of CPU and

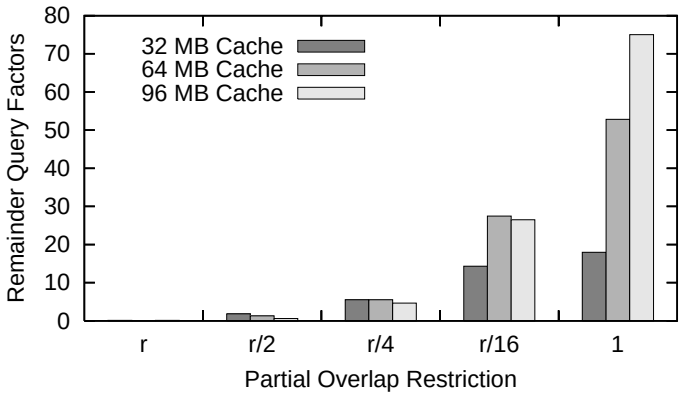

Figure 15: Remainder query factors for four attributes

memory. As disk performance has improved less, disk intensive operations such as unindexed scans show the least performance improvement. Even the modern database software has significant trouble with optimizing queries with many factors on the same set of attributes, which is likely due to the fact that such queries are infrequent in common applications. We are able to overcome this deficiency, by proposing two new remainder query execution strategies based on standard SQL constructs. Interestingly, however, the relative difference of the workloads has not changed dramatically since 1998, and complex query workloads still present significant difficulties for the semantic caching architecture.

In our view it is evident that semantic caching is a very good caching technique for workloads on one, two and even three attributes, but by the fourth attribute the hit rate drops and along with it the advantage of semantic caching. There are many applications that do not handle high-dimensional data to any extent and there the benefits of semantic caching are clear. There has been an increasing effort on research into semantic caching and related caching techniques in the last few years, especially with the increased emphasis on mobility. With mobile clients the benefits of semantic caching become especially clear, as these typically work with locally dependent low-dimensional data. We expect to see considerable development and research going into the area of intelligent caching techniques as the market demands continuously faster and more flexible clients and software applications.

References

[ABK+03] M. Altinel, C. Bornhövd, S. Krishnamurthy, C. Mohan, H. Pirahesh, and B. Reinwald. Cache tables: Paving the way for an adaptive database cache. In J. C. Freytag, P. C. Lockemann, S. Abiteboul, M. J. Carey, P. G. Selinger, and A. Heuer, editors, *Proceedings of the Conference on Very Large Data Bases (VLDB)*, Berlin, Germany, 2003. Morgan Kaufmann.

[APTP03] K. Amiri, S. Park, R. Tewari, and S. Padmanabhan. DBProxy: A dynamic data cache for web applications. In U. Dayal, K. Ramamritham, and T. M. Vijayaraman, editors, *Proceedings of IEEE Conference on Data Engineering*, Bangalore, India, 2003. IEEE Computer Society.

[CFLS91] M. J. Carey, M. J. Franklin, M. Livny, and E. J. Shekita. Data caching tradeoffs in client-server DBMS architectures. In J. Clifford and R. King, editors, *Proceedings of the ACM SIGMOD Conference on Management of Data*, Denver, CO, 1991. ACM.

[CLL+01] K. S. Candan, W.-S. Li, Q. Luo, W.-P. Hsiung, and D. Agrawal. Enabling dynamic content caching for database-driven web sites. In W. G. Aref, editor, *Proceedings of the ACM SIGMOD Conference on Management of Data*, Santa Barbara, CA, 2001. ACM.

[DeW93] D. J. DeWitt. The Wisconsin benchmark: Past, present, and future. In J. Gray, editor, *The Benchmark Handbook for Database and Transaction Processing Systems*. Morgan-Kaufmann Publishers, San Mateo, CA, 1993.

[DFJ+96] S. Dar, M. J. Franklin, B. Þ. Jónsson, D. Srivastava, and M. Tan. Semantic data caching and replacement. In T. M. Vijayaraman, A. P. Buchmann, C. Mohan, and N. L. Sarda, editors, *Proceedings of the Conference on Very Large Data Bases (VLDB)*, Bombay, India, 1996. Morgan Kaufmann.

[DR92] A. Delis and N. Roussopoulos. Performance and scalability of client-server database architectures. In L.-Y. Yuan, editor, *Proceedings of the Conference on Very Large Data Bases (VLDB)*, Vancouver, Canada, 1992. Morgan Kaufmann.

[JAÞ+06] B. Þ. Jónsson, M. Arinbjarnar, B. Þórsson, M. J. Franklin, and D. Srivastava. Performance and overhead of semantic cache management. *ACM Transactions on Internet Technology*, 6(3):302–331, 2006.

[Jón99] B. Þ. Jónsson. *Application-Oriented Buffering and Caching Techniques*. PhD thesis, University of Maryland, College Park, MD, 1999.

[LKM+02] Q. Luo, S. Krishnamurthy, C. Mohan, H. Pirahesh, H. Woo, B. G. Lindsay, and J. F. Naughton. Middle-tier database caching for e-business. In M. J. Franklin, B. Moon, and A. Ailamaki, editors, *Proceedings of the ACM SIGMOD Conference on Management of Data*, Madison, WI, 2002. ACM.

[LX04] Q. Luo and W. Xue. Template-based proxy caching for table-valued functions. In Y.-J. Lee, J. Li, K.-Y. Whang, and D. Lee, editors, *Proceedings of the International Conference on Database Systems for Advanced Applications (DASFAA)*, Jeju Island, Korea, 2004. Springer.

[MdCCC04] H. Manica, M. S. de Camargo, R. R. Ciferri, and C. D. A. Ciferri. A new model for location-dependent semantic cache based on pre-defined regions. In M. Solar, D. Fernández-Baca, and E. Cuadros-Vargas, editors, *30ma Conferencia Latinoamericana de Informática (CLEI2004)*, Arequipa, Peru, 2004.

[PL00] R. Pottinger and A. Y. Levy. A scalable algorithm for answering queries using views. In A. El Abbadi, M. L. Brodie, S. Chakravarthy, U. Dayal, N. Kamel, G. Schlageter, and K.-Y. Whang, editors, *Proceedings of the Conference on Very Large Data Bases (VLDB)*, Cairo, Egypt, 2000. Morgan Kaufmann.

[RD99] Q. Ren and M. H. Dunham. 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, 1999. ACM.

[RD00] Q. Ren and M. H. Dunham. Using semantic caching to manage location dependent data in mobile computing. In *Proceedings of the International Conference on Mobile Computing and Networking (MobiCom)*, Boston, MA, August 2000. ACM.

[SSV96] P. Scheuermann, J. Shim, and R. Vingralek. WATCHMAN: A data warehouse intelligent cache manager. In T. M. Vijayaraman, A. P. Buchmann, C. Mohan, and N. L. Sarda, editors, *Proceedings of the Conference on Very Large Data Bases (VLDB)*, Bombay, India, 1996. Morgan Kaufmann.

[Stu04] H. Stuckenschmidt. Similarity-based query caching. In H. Christiansen, M.-S. Hacid, T. Andreasen, and H. L. Larsen, editors, *Proceedings of the International Conference on Flexible Query Answering Systems (FQAS)*, Lyon, France, 2004. Springer.

[Tim02] TimesTen Team. Mid-tier caching: The TimesTen approach. In M. J. Franklin, B. Moon, and A. Ailamaki, editors, *Proceedings of the ACM SIGMOD Conference on Management of Data*, Madison, WI, 2002. ACM.

[ZLL04] B. Zheng, W.-C. Lee, and D. L. Lee. On semantic caching and query scheduling for mobile nearest-neighbor search. *Wireless Networks*, 10(6):653–664, 2004.

REYKJAVÍK UNIVERSITY

HÁSKÓLINN Í REYKJAVÍK

Department of Computer Science
Reykjavík University
Ofanleiti 2, IS-103 Reykjavík, Iceland
Tel: +354 599 6200
Fax: +354 599 6201
http://www.ru.is
ISSN 1670-5777