Semantic Data Caching and Replacement

Shaul Dar[*]
Data Technologies Ltd.
dar@dtl.co.il

Michael J. Franklin[†]
University of Maryland
franklin@cs.umd.edu

Björn T. Jónsson[†]
University of Maryland
bthj@cs.umd.edu

Divesh Srivastava
AT&T Research
divesh@research.att.com

Michael Tan[*]
University of Maryland
mdtanx@cs.umd.edu

Abstract

We propose a semantic model for client-side caching and replacement in a client-server database system and compare this approach to page caching and tuple caching strategies. Our caching model is based on, and derives its advantages from, three key ideas. First, the client maintains a semantic description of the data in its cache,which allows for a compact specification, as a *remainder query*, of the tuples needed to answer a query that are not available in the cache. Second, usage information for replacement policies is maintained in an adaptive fashion for *semantic regions*, which are associated with collections of tuples. This avoids the high overheads of tuple caching and, unlike page caching, is insensitive to bad clustering. Third, maintaining a semantic description of cached data enables the use of sophisticated value functions that incorporate semantic notions of locality, not just LRU or MRU, for cache replacement. We validate these ideas with a detailed performance study that includes traditional workloads as well as a workload motivated by a mobile navigation application.

1 Introduction

1.1 Data-shipping Architectures

A key to achieving high performance and scalability in client-server database systems is to effectively utilize the computational and storage resources of the client machines. For this reason, many such systems are based on *data-shipping*. In a data-shipping architecture, query processing is performed largely at the clients, and copies of data are brought on-demand from servers to be processed at the clients. In order to minimize latency and the need for future interaction with the server, most data-shipping systems use the local client memory and/or disk to *cache* the data that they have received from the server for possible later reuse.

Data-shipping architectures were popularized by the early generations of Object-Oriented Database Management Systems (OODBMS). These systems were aimed, in large part, at providing very efficient support for *navigational* access to data (i.e., pointer chasing), as found in object-oriented programming languages. Data-shipping is well suited to navigational access, as it brings data close to the application, allowing for very lightweight interaction between the application and the database system.

When caching is incorporated into a data-shipping architecture, servers are used primarily to service cache misses, and thus, client-server interaction is typically *fault-driven*. That is, clients request *specific* data items from the server when such items cannot be located in the local cache. The relationship between the client and server in this case is similar to that between a database buffer manager and a disk manager in a centralized database system. Not surprisingly, the techniques used to manage client caches in existing data-shipping systems are closely related to those developed for database buffer management in traditional systems. That is, a client cache is managed as a pool of individual items, typically pages or tuples. An individual item can be located in the cache by performing a lookup using its identifier, or by scanning the contents of the cache.

As with traditional buffer managers, one of the key responsibilities of a client cache manager is to determine which data items should be retained in the cache, given limited cache space. Such decisions are made using a cache *replacement policy*; each of the items is assigned a value and when space must be made available in the cache, the item or items with the least value are chosen as replacement victims. The value function for cache items is typically based on access history, such as a Least Recently Used (LRU) or a Most Recently Used (MRU) policy.

1.2 Incorporating Associative Access

In recent years, it has become apparent that large classes of applications are not well-served by purely navigational access to data. Such applications require *associative* access to data, e.g., as provided by relational query languages.

[*] The work of Shaul Dar and Michael Tan was performed when they were at AT&T Bell Laboratories, Murray Hill, NJ, USA.

[†] Supported in part by NSF Grant IRI-94-09575, an IBM SUR award, and a grant from Bellcore.

Proceedings of the 22nd VLDB Conference
Mumbai(Bombay), India, 1996

Associative access imposes different demands on a cache manager than navigational access. For example, using associative access, data items are not specified directly, but are selected and grouped dynamically based on their data values. Because of the differences between navigational and associative access, many client-server systems that focus on associative access forego the data-shipping architecture in favor of a query-shipping approach, where requests are sent from clients to servers using a higher-level query specification. The traditional query-shipping approach, however, as supported by most commercial relational database systems, does not support client caching. Thus, query-shipping architectures are less able to exploit client resources for performance or scalability enhancement.

In this paper, we propose a *semantic* model for data caching and replacement. Semantic caching is a technique that integrates support for associative access into an architecture based on data-shipping. Thus, semantic caching provides the ability to exploit client resources, while also exploiting the semantic knowledge of data that arises through the use of associative query specifications. In this approach, servers can process simple predicates (i.e., constraint formulas) on the database, sending back to the client those tuples that satisfy the predicate. The results of these predicates can then be cached at the client. A novel aspect of this approach, however, is that rather than managing the cache on the basis of individual items we exploit the semantic information that is implicit in the query predicates in order to more effectively manage the client cache.

1.3 Semantic Caching

Our semantic caching model is based on, and derives its advantages from, three key ideas.

First, the client maintains a semantic description of the data in its cache, instead of maintaining a list of physical pages or tuple identifiers. Query processing makes use of the semantic descriptions to determine what data are locally available in the cache, and what data are needed from the server. The data needed from the server are compactly specified as a *remainder query*. Remainder queries provide reduced communication requirements and additional parallelism compared to faulting-based approaches.

Second, the information used by the cache replacement policy is maintained in an adaptive fashion for *semantic regions*, which are associated with sets of tuples. These sets are defined and adjusted dynamically based on the queries that are posed at the client. The use of semantic regions avoids the high storage overheads of the tuple caching approach of maintaining replacement information on a per-tuple basis and, unlike the page caching approach, is also insensitive to bad clustering of tuples on pages.

Third, maintaining a semantic description of the data in the cache encourages the use of sophisticated value functions, in determining replacement information. Value functions that incorporate semantic notions of locality can be devised for traditional query-based applications as well as for emerging applications such as mobile databases.

We validate the advantages of semantic caching with a detailed performance study that is focused initially on traditional workloads, and is then extended to workloads motivated by a mobile navigation application.

2 Architectures for Cache Management

In order to evaluate the performance impact of semantic caching, we compare it to two traditional cache management architectures: page caching and tuple caching. In this section, we first outline the primary dimensions for comparing the three architectures in the context of associative query processing. We then describe the approaches in light of these dimensions. We focus on the particular instantiations of the architectures that are studied in this paper, rather than on an analysis of all possible design choices. More detailed discussions of the traditional architectures can be found in, among other places, [DFMV90, KK94, Fra96].

2.1 Overview of the Architectures

In this paper, we assume a client-server architecture in which client machines have significant processing and storage resources, and are capable of executing queries. We focus on systems with a single server, but all of the approaches studied here can be easily extended to a multiple server or even a peer-to-peer architecture, such as SHORE [C+94]. The database is stored on disk at the server, and is organized in terms of pages. Pages are physical units — they are fixed length. The database contains index as well as data pages. We assume that tuples are fixed-length and that pages contain multiple tuples. Pages also contain header information that enables the free space within a page to be managed independently of space on any other page.

In this study, there are three main factors that impact the relative performance of the architectures: (1) data granularity, (2) remainder queries vs. faulting, and (3) cache replacement policy. We address these factors briefly below.

2.1.1 Data Granularity

In any system that uses data-shipping, the granularity of data management is a key performance concern. As described in [CFZ94, Fra96], the granularity decisions that must be made include: (1) client-server transfer, (2) consistency maintenance, and (3) cache management. In this study (in contrast to [DFMV90]), all architectures ship data in page-sized units. Also, we examine the architectures in the context of read-only queries. Thus, the main impact of granularity in this study is on cache management. Tuple caching is based on individual tuples, page caching uses statically defined groups of tuples (i.e., pages) and semantic caching uses dynamically defined groups of tuples.

Given that tuples are fixed-length, the main differences between these three approaches to granularity are in the relative space overhead they incur for cache management (buffer control blocks, hash table entries, etc.), and in the flexibility of grouping tuples. Tuple caching incurs overhead that is proportional to the number of tuples that can be cached. In contrast, both page and semantic caching reduce overhead by aggregating information about groups of tuples. In terms of grouping tuples, semantic caching provides complete flexibility, allowing the grouping to be adjusted to the needs of the current queries. In contrast, the static grouping used by page caching is tied to a particular clustering of tuples that is determined *a priori*, independent of the current query access patterns.

2.1.2 Remainder Queries vs. Faulting

Another important way in which the architectures differ is in the way they request missing data from the server. Page caching is faulting-based. It attempts to access all pages from the local cache, and sends a request to the server for a specific page when a cache miss occurs. Tuple caching is similar to page caching in this regard, but takes care to combine requests for missing tuples so that they can be transferred from the server in page-sized groups. As described in Section 2.3, when there is no index available at the client, then the query predicate and some additional information are sent to the server to avoid having to retrieve an entire relation. This is an extension to tuple caching that we implemented in order to make a fairer comparison with semantic caching. Semantic caching describes the exact set of tuples that it requires from the server using a query called the *remainder query*. Sending queries to the server rather than faulting items in can provide several performance benefits, such as parallelism between the client and the server, and communications savings due to the compact representation of the request for missing items. An additional benefit of the approach is that in cases where all needed data is present at the client, a null remainder query is generated, meaning that contact with the server is not necessary.

2.1.3 Cache Replacement Policy

A final issue that impacts the performance of the alternative architectures is the cache replacement policy. A cache replacement policy dictates how victims for replacement are chosen when additional space is required in the cache. Such policies apply a value function to each of the cached items, and choose as victims, those items with the lowest values. In traditional systems, value functions typically are based on *temporal locality* and/or *spatial locality*. Temporal locality is the property that items that have been referenced recently are likely to be referenced again in the near future; the LRU policy is based on the assumption of temporal locality. Spatial locality is the property that if an item has been referenced, other items that are physically close to it are also likely to be referenced; page caching tries to exploit spatial locality under the assumption that clustering of tuples to pages is effective. As demonstrated in Section 3, semantic caching enables the use of a dynamically defined version of spatial locality, that we refer to as *semantic locality*. Semantic locality differs from spatial locality in that it is not dependent on the static clustering of tuples to pages; rather it dynamically adapts to the pattern of query accesses.

2.2 Page Caching Architecture

In page caching architectures (also referred to as *page-server* systems [DFMV90, CFZ94]), the unit of transfer between servers and clients is a page. Queries are posed at clients, and processed locally down to the level of requests for individual pages. If a requested page is not present in the local cache, a request for the page is sent to the server. In response to such a request, the server will obtain the page from disk (if necessary) and send the page back to the client. On the client side, page caching is supported through a mechanism that is nearly identical to that of a traditional page-based database buffer manager. A client can perform partial scans on indexed attributes by first accessing the index (faulting in any missing index pages) and then accessing qualifying data pages. If no index is present then a page caching approach will scan an entire relation, again faulting in any missing pages. As with a buffer manager, a page cache is managed using simple replacement strategies based on the usage of the data items, such as LRU or MRU.

2.3 Tuple Caching Architecture

Tuple caching is in many ways analogous to page caching, the primary difference being that with tuple caching, the client cache is maintained in terms of individual tuples (or objects) rather than entire pages. Caching at the granularity of a single item allows maximal flexibility in the tuning of cache contents to the access locality properties of applications [DFMV90]. As described in [DFMV90], however, the faulting in of individual tuples (assuming that tuples are substantially smaller than pages) can lead to performance problems due to the expense of sending large numbers of small messages. In order to mitigate this problem, a tuple caching system must group client requests for multiple tuples into a single message and must also group the tuples to be sent from servers to clients into blocks.

Scans of indexed attributes can be answered in a manner similar to page caching. For scans of non-indexed attributes however, there are two options. One option is for the client to first perform the scan locally, and then send a list of all qualifying tuples that it has in its cache, along with the scan constraint to the server. The server can then process the scan, sending back to the client only those qualifying tuples that are not in the client's cache. An alternative is for the client to simply ignore its cache contents when performing a scan on a non-indexed attribute. In this case, the scan constraint is sent to the server, and all qualifying tuples are returned; duplicate tuples can be discarded at the client.

Finally, the tuple cache, like a page cache, is managed using an access-based replacement policy such as LRU. Unlike the page cache, however, there is no notion of spatial locality for tuples, so only temporal locality is exploited.

2.4 Semantic Caching Architecture

Semantic caching manages the client cache as a collection of *semantic regions*; that is, access information is managed, and cache replacement is performed, at the unit of semantic regions. Semantic regions, like pages, provide a means for the cache manager to aggregate information about multiple tuples. Unlike pages, however, the size and shape (in the semantic space) of regions can change dynamically.

Each semantic region has a constraint formula describing its contents, a count of tuples that satisfy the constraint, a pointer to a linked list of the actual tuples in the cache, and additional information that is used by the replacement policy to rank the regions. The formula that describes a region specifies the region's location in the semantic space. Unlike the replacement value functions used by the page and tuple caching architectures, the value functions used by semantic caching may take information about the semantic locality of regions into account.

When a query is posed at a client, it is split into two disjoint pieces: (1) a *probe* query, which retrieves the portion of the result available in the local cache, and (2) a *remainder query*, which retrieves any missing tuples in the answer from the server. If the remainder query is not null (i.e., the query covers parts of the semantic space that are not cached) then the remainder query is sent to the server and processed there. Similar to tuple caching, the result of the remainder query is packed into pages and sent to the client. Unlike tuple caching, however, the mechanism for obtaining tuples from the server is independent of the presence of indexes.

3 Model of Semantic Caching

3.1 Basic Terminology

Semantic caching exploits the semantic information present in associative query specifications to organize and manage the client cache. In this study, we consider selection queries on single relations, where the selection condition is an arbitrary constraint formula (that is, a disjunction of conjunctions of built-in predicates); dealing with more complex queries within the framework of semantic caching is an important direction of future research. In semantic caching, the portion of a single relation present in the client cache is also described by a constraint formula; the entire contents of the client cache are described by a set of such constraint formulas, one for each database relation.

A query can be split into two disjoint portions: one that can be completely answered using the tuples present in the client cache, and another that requires tuples to be shipped from the server. In semantic caching, the notions of a probe query and a remainder query correspond to these two portions of the query. More formally, given a query on relation R with constraint formula Q, if V denotes the constraint formula describing the set of tuples of R present in the client cache, then the *probe query*, denoted by $\mathcal{P}(Q, V)$, can be defined by the constraint formula $Q \wedge V$ on R. Further, the *remainder query*, denoted by $\mathcal{R}(Q, V)$, can be defined by the constraint formula $Q \wedge (\neg V)$ on R.

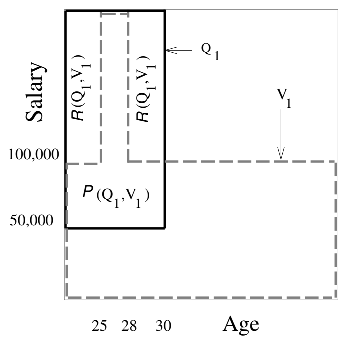


Figure 1: Semantic Spaces

For example, consider a query to find all employees whose salary exceeds $50,000$, and who are at most 30 years old. This query can be described by the constraint formula $Q_1 = (Salary > 50,000 \wedge Age \leq 30)$ on the relation $employee(Name, Salary, Age)$. Assume that the client cache contains all employees whose salary is less than $100,000$ as well as all employees who are between 25 and 28 years old. This can be described by the formula $V_1 = (Salary < 100,000 \vee (Age \geq 25 \wedge Age \leq 28))$.

The probe query $\mathcal{P}(Q_1, V_1)$ into the client cache is described by the constraint formula $((Salary > 50,000 \wedge Salary < 100,000 \wedge Age \leq 30) \vee (Salary > 50,000 \wedge Age \geq 25 \wedge Age \leq 28))$. This constraint describes those tuples in the cache that are answers to the query. The remainder query $\mathcal{R}(Q_1, V_1)$ is described by the constraint formula $((Salary \geq 100,000 \wedge Age < 25) \vee (Salary \geq 100,000 \wedge Age > 28 \wedge Age \leq 30))$. This constraint describes those tuples that need to be fetched from the server.

When the constraint formulas are arithmetic constraints over attributes $A_1, \ldots, A_n$, they have a natural visualization as sub-spaces of 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 1 depicts the projection onto the $Salary$ and Age attributes of the semantic spaces associated with the $employee$ relation, query Q_1, cache contents V_1, the probe query $\mathcal{P}(Q_1, V_1)$ and the remainder query $\mathcal{R}(Q_1, V_1)$.

3.2 Semantic Regions

Client cache size is limited, and existing tuples in the cache may need to be discarded to accommodate the tuples required to answer subsequent queries. Semantic caching manages the client cache as a collection of *semantic regions* that group together semantically related tuples; *each tuple in the client cache is associated with exactly one semantic region*. These semantic regions are defined dynamically based on the queries that are posed at the client.

Each semantic region has a constraint formula that describes the tuples grouped together within the region, and has a single replacement value (used to make cache replacement decisions) associated with it; all tuples within a semantic region have the replacement value of that region.

When a query intersects a semantic region in the cache, that region gets split into two smaller disjoint semantic regions, one of which 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 a 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 $2n + 1$ regions; of these regions $n + 1$ are part of the query. The question then arises whether or not to coalesce some or all of these regions into one or more larger regions.

A straightforward approach is to always coalesce two regions that have the same cache replacement value, resulting in only one region corresponding to the query. With small (relative to cache size) queries, this strategy can lead 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 a large region can empty a significant portion of the cache, resulting in poor cache utilization.

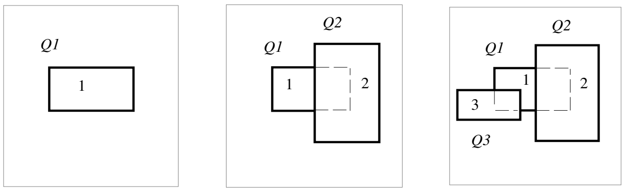
Another option is to never coalesce. For small queries that tend to intersect, this can lead to excessive overhead, but for larger queries, it alleviates the granularity problem.

In our approach, therefore, we use an adaptive heuristic. Regions with the same cache replacement value may be coalesced if either one of them is smaller than 1% of the cache size. As shown in Section 5.1, this heuristic strikes a good balance between the two extremes.

3.3 Replacement Issues

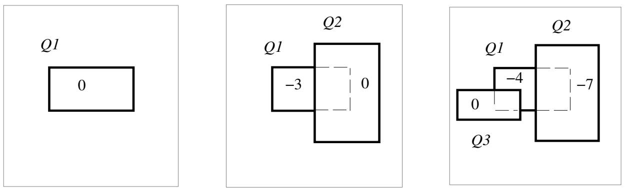
When there is insufficient space in the cache, the semantic region with the lowest value and all tuples within that region are discarded from the cache. Semantic regions are, thus, the unit of cache replacement. The value functions used by semantic caching can be based on temporal locality (e.g., LRU, MRU), or on semantic locality of regions. Below, we describe two caching/replacement policies, one where the replacement value is based on recency of usage, and another where it is based on a distance function.

Maintaining replacement values based on *recency of usage* allows for the implementation of replacement policies such as LRU or MRU. Conceptually, tuple caching and page caching associate a replacement value with each tuple or page, corresponding to the latest time the item in the cache was accessed. Maintaining replacement values based on recency of usage in the semantic caching approach associates such a value with each semantic region, based on the sequence of queries issued at the client. Figure 2 illustrates the semantic regions and their associated replacement values, based on recency of usage, for a sequence of three range queries on a single binary relation. The solid lines show the semantic regions created when full coalescing is performed, the dotted lines depict the additional semantic regions that would result if no coalescing were performed.



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

Figure 2: Semantic Regions: Recency of Usage



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

Figure 3: Semantic Regions: Manhattan Distance

The constraint formula $Q1$ corresponding to the first query is the only semantic region (with value 1) after $Q1$ is issued (see Figure 2(a)). The second query $Q2$ overlaps with the semantic region with value 1, and the constraint formula $Q2$ is the semantic region with value 2. Since semantic regions have to be mutually disjoint, the semantic region with value 1 "shrinks", after $Q2$ is issued, to the portion that is disjoint with $Q2$ (see Figure 2(b)). Similar shrinking occurs when the third query is issued; note that the semantic region with value 1 is no longer convex, and its constraint formula is not conjunctive. In fact, semantic regions may not be connected in the semantic space.

An alternative to using recency information for determining replacement values is to use *semantic distance*. Figure 3 shows the result of using *Manhattan distance* in the previous example. In this case, each semantic region is assigned a replacement value that is the negative of the *Manhattan distance* between the "center of gravity" of that region and the "center of gravity" of the *most recent* query. With this distance function, semantic regions that are "close" to the most recent query have a small negative value, irrespective of when they were created, and are hence less likely to be discarded when free space is required.

3.4 An Operational Model

We now describe an operational model of semantic caching. In this model the client processes a stream of queries $Q_1, \ldots, Q_m$ on relation R. Let V_{i-1} denote the cache contents for relation R, and $\mathcal{S}_{i-1}$ denote the set of semantic regions of relation R, when query Q_i is issued. V_0 is the constraint formula $false$, and $\mathcal{S}_0$ is empty. Processing query Q_i, involves the following steps:

1. Compute the probe query $\mathcal{P}(Q_i, V_{i-1})$ and the remainder query $\mathcal{R}(Q_i, V_{i-1})$ from Q_i and V_{i-1}. Partly answer query Q_i from the set of tuples that satisfy $\mathcal{P}(Q_i, V_{i-1})$.
2. Repartition $\mathcal{S}_{i-1}$ into $\mathcal{S}'_i$ and update the replacement values associated with the semantic regions in $\mathcal{S}'_i$ based on $\mathcal{P}(Q_i, V_{i-1})$, $\mathcal{R}(Q_i, V_{i-1})$, and the caching/replacement policy used.
3. Fetch the tuples of R that satisfy the constraint formula $\mathcal{R}(Q_i, V_{i-1})$ from the server.
4. If the cache does not have enough free space, discard semantic regions $S_1, \ldots, S_k$ with low values among the set of semantic regions $\mathcal{S}'_i$, and discard tuples in the cache that satisfy the constraint formulas $S_1, \ldots, S_k$ until enough space is free.
5. Answer the rest of query Q_i by taking the set of tuples that satisfy $\mathcal{R}(Q_i, V_{i-1})$.
6. Compute V_i by taking the disjunction of V_{i-1} and $\mathcal{R}(Q_i, V_{i-1})$, and then taking the difference with respect to $S_1, \ldots, S_k$; Determine the semantic regions $\mathcal{S}_i$ in the cache and update their replacement values based on $\mathcal{S}'_i$, $\mathcal{R}(Q_i, V_{i-1})$, the discarded semantic regions $S_1, \ldots, S_k$, and the caching/replacement policy.

4 Simulation Environment

4.1 Resources and Model Parameters

Our simulator is an extension of the one used in [FJK96], written in C++ using CSIM. It models a heterogeneous, peer-to-peer database system such as SHORE [C+94], and provides a detailed model of query processing costs in such a system. For this study, the simulator was configured to model a system with a single client and a single server.

Table 1 shows the main parameters of the model. Every site has a CPU whose speed is specified by the *Mips* parameter, *NumDisks* disks, and a main-memory buffer pool. At the client, the size of the buffer pool is *ClientCache*.[1] The details of buffer management overhead for the different client caching strategies are described in Section 4.2.

The CPU is modeled as a FIFO queue. The client has an optional disk-resident cache, which also uses the parameter *ClientCache*; the memory cache is not used in this case. The disk cache is used for queries on non-indexed attributes, and the whole disk cache is scanned in sequence when answering such queries. Disks are modeled using a detailed characterization adapted from the ZetaSim model [Bro92]. The disk model includes an elevator scheduling policy, a controller cache, and read-ahead prefetching. There are many parameters to the disk model (not shown) including: rotational speed, seek factor, settle time, track and cylinder sizes, controller cache size, etc. In addition to the time spent waiting for and accessing the disk, a CPU overhead of *DiskInst* instructions is charged for every disk I/O request.

Mips	50	CPU speed of a site (10^6 inst/sec)
NumDisks	1	number of disks on a site
ClientCache	250	cache size at the client (Kb)
DiskInst	5000	inst. to read a page from disk
PageSize	4096	size of one data page (bytes)
NetBw	8	network bandwidth (Mbit/sec)
MsgInst	20000	inst. to send/receive a message
PerSizeMI	12000	inst. to send/receive a page
Display	0	inst. to display a tuple
Compare	2	inst. to apply a predicate
Move	1	inst. to copy 4 bytes

Table 1: Model Parameters and Default Settings

The database, the server buffer pool, and the client's disk cache are organized in pages of size *PageSize*. Pages are the unit of disk I/O and data transfer between sites. The network is modeled as a FIFO queue with a specified bandwidth (*NetBw*); the details of a particular technology (e.g, Ethernet, ATM) are not modeled. The cost of sending a message involves the time-on-the-wire (based on the size of the message), a fixed CPU cost per message (*MsgInst*), and a size-dependent CPU cost (*PerSizeMI*).

When scanning a relation at the server, there is a dedicated process which attempts to keep the scan one page ahead of the consumer at the client. This leads to overlap between disk reads and network messages, which is most apparent when the result size is small relative to the amount of data scanned. In the extreme case, network communication can be done completely parallel to the disk reads. This overlap does not arise when data is *faulted* in to the client, as there is no dedicated process at the server in this case.

In addition to the CPU costs for systems functions such as messages and I/Os, there are also costs associated with the functions performed by query operators. The costs that are modeled are those of displaying, comparing, and moving tuples in memory.

4.2 Buffer Management at the Client

In order to maintain fairness to the different caching architectures, the *ClientCache* parameter includes both the space needed for buffer management overhead, and the space available for storing data. Since we do not consider updates in this study, we do not model the overhead needed to facilitate updates. We also do not model the CPU cost of cache management at the client.

To estimate the overhead of page buffer management, we

[1] As each page is referenced only once per query, and server buffers are cleared between queries, the buffer size at the server does not matter.

RelSize	10000	Size of database relation (tuples)
TupleSize	200	Size of each tuple (bytes)
QuerySize	1–10%	% of relation selected by each query
Skew	90%	% of queries within a hot region
HotSpot	10%	Size of the hot region (% of relation)

Table 2: Workload Parameters and Default Settings

used the Buffer Control Block of [GR93]. After removing all attributes pertaining to updates and concurrency control, we were left with 28 bytes per page. To model the storage cost of indexes, we assume that the primary index takes up negligible space, as also the upper levels of the secondary index. The leaf level of the secondary index, however, has 8 bytes per tuple. This adds up to 188 bytes of overhead for a page of 20 tuples. In a cache of size 250Kb, we can then fit $\frac{256000}{4096+188} \approx 60$ pages.

For tuple shipping the same data structure can be used for cache management, with two exceptions. Tuple size needs to be kept, and tuple identifiers are typically larger than page identifiers. However, since we used fixed size tuples, and do not have a specific implementation of tuple identifiers, we chose to use 28 bytes per tuple. With the 8 bytes for indexes, that adds up to 36 bytes per tuple. In a cache of size 250Kb, we can then fit $\frac{256000}{200+36} \approx 1085$ tuples.

For semantic caching, the buffer management information is kept on a semantic region basis. The replacement information needed is similar to page and tuple caching; however, the page identifier, the frame index and the hash overflow pointer are not needed. Instead, we need additional pointers to the list of factors in the constraint formula describing the region, and to the list of tuples in the region. This is a total of 24 bytes. For each factor in the constraint formula we need the endpoints of the range of each attribute (8 bytes per attribute), and a pointer to the next factor (4 bytes). For each tuple we need a pointer to the next tuple (4 bytes). Note, that we do not need to model a storage overhead for indexes at the client, as the semantic cache uses semantic information to organize the data. Since the overhead is variable, our implementation simply makes sure that the size of the overhead data structures and the actual data is never more than the size of the cache.

4.3 Workload Specification

We use a benchmark consisting of simple selections. The size of the result *QuerySize* is varied in the experiments, but is always smaller than the cache. A fixed portion of the queries (*Skew*) has the semantic centerpoint within a hot region of size *HotSpot*.[2] The remaining queries are uniformly distributed over the cold area.

As shown in Table 2, we use a single relation with 10,000 tuples of 200 bytes each. We have intentionally kept the database small and have sized the cache proportionally, in order to make the running of a large number of experiments feasible. As with all caching studies, what determines the performance is the relative sizes of the cache, databases, and access regions, rather than their absolute sizes.[3] The relation has three candidate keys, which we adopted from the Wisconsin benchmark: *Unique2* is indexed and perfectly clustered; *Unique1* is indexed but completely unclustered; *Unique3* is both unindexed and unclustered.

[2] Since the only requirement for a hot query is that the centerpoint be within the hot spot, a sizable fraction of the query may lie outside the hot spot. The semantic area adjacent to the hot spot will therefore also have a significant number of hits.

5 Experiments and Results

In this section we examine the performance of the three caching architectures using a workload consisting of selection queries on a Wisconsin-style database using various indexed and non-indexed attributes. As shown in Table 2, the access pattern is skewed so that 90% of the queries have a centerpoint that lies within the hot region consisting of the middle 10% of the relation. In all the experiments in this section, the client cache is set to 250Kb, which is sufficient to store the entire hot region, including overhead, for all three approaches.

The primary metric used is response time. Where necessary, other metrics such as cache hit rates, message volumes, etc. are used. The numbers were obtained by averaging the results of three runs of queries. Each run consisted of 50 queries to warm up the cache followed by 500 query executions during which the measurements were taken. The results presented here are a small, but representative set of the experiments we have run. In particular we ran numerous sensitivity experiments varying cache size, hot region size, tuple size, skew, etc.

5.1 Indexed Selections

We first study the performance of the three caching architectures when performing single- and double-attribute selections on indexed attributes. Figure 4 shows the response time for the three caching architectures when the selection is performed on the *Unique2* attribute, which has a clustered index. The x-axis of the figure shows the query result size expressed as a percentage of the relation size. In this case, it can be seen that all three architectures provide similar performance across the range of query sizes. As the query size is increased (while the cache size is held constant), the response time for all of the architectures worsens due to lower client cache hit rates. Tuple caching has the worst performance in this experiment and page and semantic caching perform roughly equally. Tuple caching's worse performance in this case is due to its relatively high space overhead. As described in Section 4.2, tuple caching incurs an overhead of 36 bytes per every 200 byte tuple in the indexed case. In contrast, page caching incurs an overhead

[3] We also conducted experiments where the database, cache, and the queries, were all scaled up by a factor of 10. The results (in terms of relative performance) in this case were nearly identical.

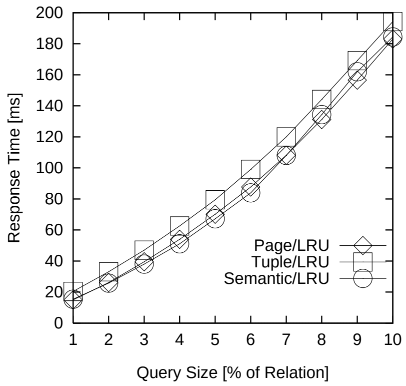


Figure 4: Resp. Time, *Unique2*
Mem. Cache, Varying Query Size

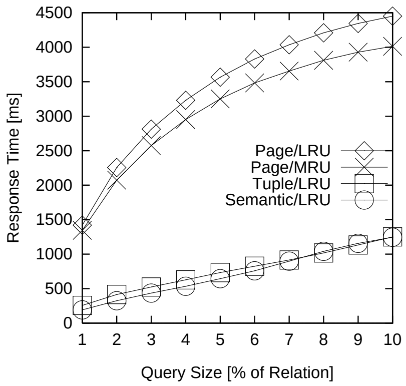


Figure 5: Resp. Time, *Unique1*
Mem. Cache, Varying Query Size

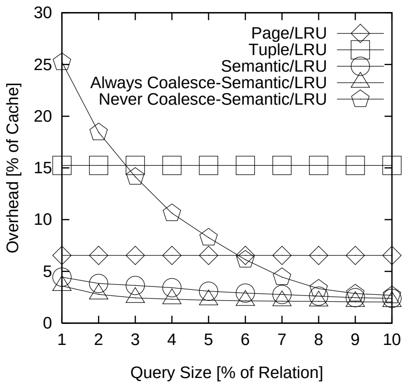


Figure 6: Overhead, *Unique1/Unique2*
Mem. Cache, Varying Query Size

of less than 10 bytes per tuple, and because *Unique2* is a clustered attribute, nearly all of the tuples in an accessed page satisfy the query. Thus, page caching has approximately 10% more data in the cache than tuple caching here. Semantic caching has even lower space overhead than page caching in this experiment; however, this slight advantage is mitigated by an equally slight degradation in cache utilization as the query size increases. With larger regions, the replacement granularity of semantic caching increases. Replacing large regions temporarily opens up large holes in the cache, which is detrimental to overall cache utilization.

Figure 5 shows the response times for the architectures when the selection is on *Unique1*, the non-clustered indexed attribute. In this figure, the performance of page caching is shown for two different cache value functions: LRU and MRU. In this experiment, the page caching approach performs far worse than both the tuple and semantic caching approaches. Page caching's poor performance here is to be expected; since *Unique1* is unclustered, the hot region of the relation is not able to fit entirely in the cache. MRU helps page caching slightly in this case, because the non-clustered index scan processes the pages of the relation sequentially. Of course, random clustering is the worst case for page caching, which is based on the assumption of spatial locality. Nevertheless, comparing this graph with the previous one demonstrates the sensitivity of page caching to clustering. Also the two experiments demonstrate that the space overhead of semantic caching is the same or better than page caching, but that unlike page caching, a semantic cache is not susceptible to poor static clustering.

The first two experiments examined single-attribute queries. We also studied queries that are multi-attribute selections on the combination of *Unique1* and *Unique2*. The results in this case (not shown) are similar to those of the non-clustered selection of the previous experiment: page caching suffers due to poor clustering; tuple and semantic caching provide similar, and much better performance. The important aspect of this experiment, however, can be seen in Figure 6, which shows the total space overhead (as a percent of the cache size) incurred by page and tuple caching and three variants of semantic caching.

The storage overhead for tuple caching and page caching is proportional to the number of items that fit in the cache, so it is independent of the query size. Page caching has an overhead of 6.5% (including the cost of unused space on the pages) while the overhead of tuple caching is 15.2% for all query sizes in Figure 6. Despite its advantage in overhead, however, page caching still performs much worse than tuple caching in this experiment because of the lack of clustering with respect to the *Unique1* attribute.

In contrast to page and tuple caching, the space overhead of semantic caching is dependent on both the query size and the coalescing strategy. The three lines shown for semantic caching in Figure 6 show the overhead for three different approaches to coalescing regions. The highest space overhead is observed when coalescing is turned off ("Never Coalesce"). Recall that a query that touches n regions can result in the creation of up to $n + 1$ new regions. If these new regions are not coalesced, the overhead incurred can be significant. As can be seen in the figure, the overhead is significantly worse for smaller queries than for larger ones. For 1% queries, there are 55 regions and nearly 275 factors. In contrast, when coalescing is performed aggressively ("Always Coalesce") overhead is decreased substantially (e.g., by 85% for the smallest query). As stated previously, however, aggressive coalescing can also negatively affect cache utilization by increasing the granularity of cache replacement. In this experiment, aggressive coalescing has as much as 10% lower cache utilization compared to never coalescing. Finally, the regular "semantic" line, shows the effectiveness of the default coalescing heuristic described in Section 3.2. In this case, the overhead is only slightly higher than that of always coalescing, while the cache utilization (not shown) is nearly the same as that of never coalescing. Thus, these results demonstrate that the simple coalescing heuristic used by semantic caching is highly effective.

Finally, it should also be noted that the space overhead of semantic caching is impacted by the dimensionality of the semantic space. In this case, since the semantic space is two-dimensional, semantic caching incurs somewhat higher

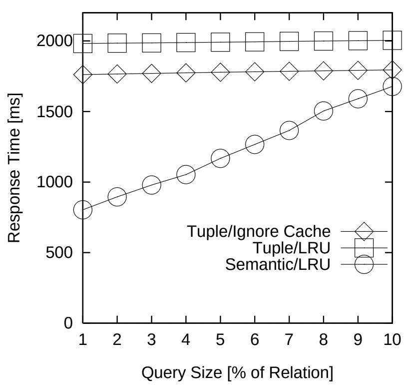


Figure 7: Resp. Time, *Unique3*
Disk Cache, Varying Query Size

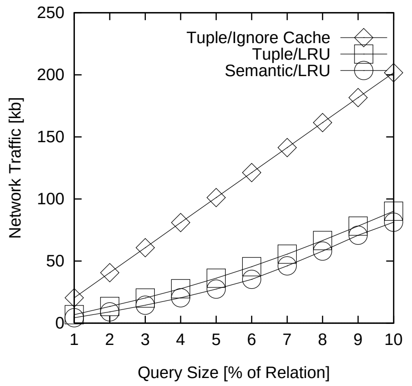


Figure 8: Network Volume, *Unique3*
Disk Cache, Varying Query Size

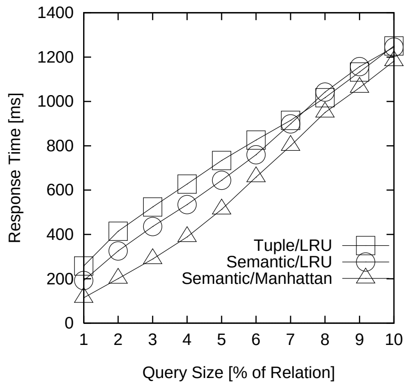


Figure 9: Resp. Time, *Unique1*
Mem. Cache, Varying Query Size

overhead due an increase in the number of semantic regions and the complexity of the constraint formulas that describe them. For small queries, the overhead of the never coalesce case is over four times higher than in a single-attribute semantic space. The default coalescing heuristic, however, does not suffer from this overhead explosion: its overhead even for the smallest queries is only about one third higher than in the single attribute case.

5.2 NonIndexed Selections

As described in Section 2, the availability (or lack) of indexes at clients dictates the manner in which the page and tuple caching architectures process queries. In this section we examine the performance of the tuple caching and semantic caching architectures when performing selections on an *unindexed* attribute (*Unique3*).[4] For tuple caching, we explore two approaches to processing selections on unindexed attributes. One approach exploits the client cache by first applying the selection predicate to all of the cached tuples of the given relation and sending the list of qualifying tuples, along with the selection predicate to the server. The server then applies the predicate to the entire relation (recall that there is no index) and sends any qualifying tuples that are missing from the cache. The second approach simply ignores the cache and sends the predicate to the server. In this case all qualifying tuples are sent to the client.[5]

Figure 7 shows the response time of semantic caching and the two tuple-based architectures when the client uses its *local disk* as a cache, rather than its memory. We use a disk cache here, in order to demonstrate a fundamental advantage of semantic caching over tuple (or page) caching; namely, that the use of *remainder queries* for requesting missing tuples from the server enables the client and the server to process their (disjoint) portions of the query *in parallel*. In contrast, for a client to exploit a tuple cache in this case, it must scan the local cache prior to initiating the scan at the server. The result of the sequential processing in this experiment is that tuple caching has worse response time even than a tuple-based approach that completely ignores the cache. The main reason for this non-intuitive behavior is that because the selection is applied to a non-indexed attribute, any data request sent to the server results in a full scan of the relation (from disk) at the server. The cost of this scan dominates all other activities in this case, and since the server is able to overlap communication with I/O, the communication costs do not factor into the total response time. Thus, in this experiment, tuple caching performs extra work prior to contacting the server, but sees no benefit in response time resulting from this work. Such a benefit, however, is evident in Figure 8 which shows the number of bytes sent across the network per query. In this case, the use of the client cache results in a significant reduction in message volume. In a network constrained environment (e.g., a wireless mobile network), such communication savings may be the dominant factor. Finally, it should be noted that when a memory cache is used rather than a disk cache, the performance of tuple caching is roughly equal to that of the "tuple ignore" policy in this experiment.

Turning to the performance of semantic caching in Figure 7, it can be seen that semantic caching provides significant performance benefits for small queries. This result is unexpected, because as described above, any data request sent to the server incurs a full relation scan, resulting in performance similar to that of "tuple ignore". This result illustrates another fundamental advantage of semantic caching, namely that by maintaining *semantic* information about cache contents, a semantic caching system can identify cases when *it can answer a query without contacting the server*. In this experiment, over 60% of the small (1%) queries are answered completely from the client's cache, thus avoiding the disk scan at the server.[6] In contrast, tuple caching, which also often had an entire answer in cache, was still required to perform a disk scan at the server, only

[4] Page caching performs significantly worse than the others here due to the lack of clustering, and is therefore not shown.

[5] Note that these approaches assume that the server has the ability to process selection predicates, as is also required for semantic caching.

[6] When the query size is so large that no queries are answered completely in cache, then the performance of semantic caching becomes equal to that of "tuple ignore" in this experiment.

to find that no extra tuples were needed. Finally, it should be noted that in environments where communication channels are scarce, such as cellular networks, the ability to operate independently of the server can result in significant monetary savings in addition to performance gains.

5.3 Semantic Value Function

The previous experiments brought out several intrinsic benefits of maintaining cache contents using semantic information, including low space overhead, insensitivity to page clustering, client-server parallelism, and the ability to answer some queries without contacting the server. In this section we demonstrate another advantage of semantic caching: the ability to incorporate *semantic locality* in cache replacement value functions. As an example we use the Manhattan distance described in Section 3.3.

Figure 9 shows the response time for selection queries on the non-clustered, indexed attribute *Unique1*. As can be seen in the figure, the Manhattan distance provides better performance for all query result sizes in this experiment. The Manhattan distance is more effective than LRU at keeping the hot region in memory, resulting in a better cache hit rate. The reason that LRU loses in this workload is that there are a significant number of queries (10%) that land in the cold region of the relation. Such cold data is not likely to be accessed in the near future, but it stays in the cache until it ages out of the LRU chain. In contrast, using the Manhattan distance function, such a cold range would loose its value when the next "hot range" query is submitted.

6 Mobile Navigation Application

In the previous section, we showed that semantic locality can improve performance even in a randomized workload. In this section, we further examine the benefits of semantic locality by exploring a workload that has more semantic content than the selection-based workloads studied so far. The workload models mobile clients accessing remotely-stored map data through a low-bandwidth wireless communication network (see, e.g., [D+96]). Each tuple in the database represents a road segment in the map, and each page is a collection of such tuples. The application must update the map data displayed to the user at regular intervals, depending on the user's current location, direction and speed of motion.

6.1 Workload Specification

The database is one relation, two of whose attributes take values between 0 and 8191. This pair of attributes forms a dense key of the relation; there is a tuple for every possible pair of values. These two attributes can be viewed as the X and Y co-ordinates in a 2-dimensional space. The relation is clustered using the Z-ordering [Jag90] on these two attributes. Each tuple is 200 bytes long.

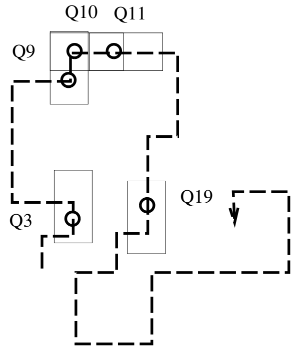


Figure 10: Random Query Path

We use a benchmark of simple selections of tuples, which is characteristic of map data accesses in a navigation application. Each query is in the form of a rectangle of size 8×16, oriented along one of the two axes in the semantic space of the two spatial attributes of the relation; thus, each query answer has 128 tuples. The location and orientation of the query rectangle depends on the user's current location and direction of motion. A query path corresponds to navigating through the 2-dimensional space in a Manhattan fashion. Figure 10 gives an example of such a query path.

We simulated a variety of query profiles: random, squares, and Manhattan "lollipops". The *random* profile has a fixed probability of moving in one of the four directions. In each step, moving left, right or backward is by 4 units, moving forward is by 8 units; the difference essentially models different speeds of motion. The *square* profile involves the query path repeatedly traversing a fixed size square in the 2-dimension space. The *Manhattan lollipop* profile is a square balanced on top of a "stick". Each query path goes up the stick, traverses around the square multiple times, goes down the stick, and then repeats the cycle.

6.2 Semantic Value Function

Consider the query path in Figure 10. Using a replacement policy like LRU is not very appropriate for such query profiles. Assume that when $Q19$ is issued, some map data must be discarded from the client cache. If an LRU policy is used, the map data associated with $Q3$ is likely to be discarded, since it has not been accessed for a long time. A semantic caching policy can recognize the semantic proximity of $Q3$ and $Q19$, and discard the data associated with $Q9, Q10, Q11$ in preference to the data associated with $Q3$, resulting in better cache utilization. We now describe a semantic value function, the *directional Manhattan* distance function, that maintains a single number with each semantic region based on its Manhattan distance from the user's current location and direction of motion.

Assume that the user's direction of motion is the positive X axis (for other directions of motion, the distance function is defined similarly), and let p_a, p_l, p_r and p_b denote the weights that model the relative importance of retaining in the cache semantic regions that are ahead of, to the left of, to the right of, and behind the current region. Let (x_u, y_u) be the user's current location, and (x, y) be the center of a semantic region S in the cache. The replacement information associated with S is computed as $-(d_{par} + d_{perp})$, where the values d_{par} (parallel distance) and d_{perp} (perpendicular distance) are defined as follows:

$$\begin{aligned} d_{par} &= \text{if } x > x_u \text{ then } (1 - p_a) * (x - x_u) \\ & \quad \text{else } (1 - p_b) * (x_u - x) \\ d_{perp} &= \text{if } y > y_u \text{ then } (1 - p_l) * (y - y_u) \\ & \quad \text{else } (1 - p_r) * (y_u - y) \end{aligned}$$

6.3 Performance Results

We present a performance comparison of LRU, MRU and the directional Manhattan distance function for semantic caching for various query profiles. The metric used is average response time to answer queries over a sequence of 500 queries. We also studied the LRU and MRU value functions for tuple caching; since they always do slightly worse than their semantic counterparts, we do not discuss them further.

A key characteristic of the query profiles we study is the possibility of *loops* in a query path, i.e., the user can visit or be close to a previously visited location. When the query path is random and the loops are small, LRU is expected to perform well since recent data will be retained in the cache. When the query path is regular and the loops are larger, MRU is expected to perform well, since older data (guaranteed to be touched again) will be retained in the cache. We demonstrate that, in contrast to LRU and MRU, a value function based on semantic distance, performs *robustly*, across a wide range of loop sizes.

We study random query paths, for four different choices of probability values. The directional Manhattan distance function is the winner, though LRU is a close second. An interesting point to note is that the directional Manhattan distance function performs substantially better than MRU when the query path is totally random ($.25/.25/.25/.25$). When the query path approaches a straight line ($.80/.10/.10/.00$), all approaches perform comparably – there is not much scope for improvement in this case.[7] Our results are summarized in table 3.

Each step for the square and the Manhattan lollipop profiles is 8 units long. The square sizes studied were 32×32 and 160×160. This query profile – predictable and cyclic – is ideal for MRU, which is the clear winner. The query results for the 32×32 square are just slightly larger than the cache size. A semantic distance function can be expected to be useful in this case, and the directional Manhattan distance function considerably outperforms LRU. The query results for the 160×160 square are approximately five times larger than the cache size. LRU and the directional Manhattan distance function essentially keep the same data in the cache, and hence they perform similarly.

[7] In the absence of loops, i.e., when data is touched at most once, caching is not useful, and no value function will perform well.

Size/Path	Dir. Manhattan	LRU	MRU
Random			
.25/.25/.25/.25	1.00 (29.4 ms)	1.06	2.24
.33/.33/.33/.00	1.00 (42.5 ms)	1.05	1.52
.50/.20/.20/.10	1.00 (44.6 ms)	1.03	1.38
.80/.10/.10/.00	1.00 (56.1 ms)	1.01	1.04
Square			
32×32	2.29	9.57	1.00 (7.23 ms)
160×160	1.22	1.22	1.00 (51.9 ms)
Manhattan Lollipop			
160/32×32/1	1.86	2.02	1.00 (47.1 ms)
160/32×32/5	1.00 (62.6 ms)	1.22	1.11
160/32×32/10	1.00 (49.2 ms)	1.38	1.60
160/32×32/50	1.00 (34.9 ms)	1.69	2.54

Table 3: Mobile Query Paths

For the Manhattan lollipop query path, the square size is 32×32, and the stick length is 160; we considered different values for the number of times the square is traversed in each cycle: $1, 5, 10$ and 50 (in this case the query path does not complete a full cycle). When the square is traversed once in each cycle, the path is very regular and MRU outperforms the other approaches. When the square is traversed a large number of times in each cycle, the regularity breaks down and MRU begins to lose. The break-even point between MRU and the directional Manhattan distance function is 4 rounds, and the break-even point between MRU and LRU is between 6 and 7 rounds. The directional Manhattan distance function is always better than LRU, and hence is the clear winner when the square is traversed many times.

7 Related Work

Data-shipping systems have been studied primarily in the context of object-oriented database systems, and are discussed in detail in [Fra96]. The tradeoffs between page caching (called page servers) and tuple caching (called object servers) were initially studied in [DFMV90]. That work demonstrated the sensitivity of page caching to static clustering, and also the message overhead that results from sending tuples from the server one-at-a-time. In our implementation of tuple caching, we took care to group tuples into pages before transferring them from the server.

Alternative approaches to making page caching less sensitive to static clustering have been proposed [KK94, OTS94]. These schemes, known as *Dual Buffering* and *Hybrid Caching* respectively, keep a mixture of pages and objects in the cache based on heuristics. A page is kept whole in the cache if enough of its objects are referenced, otherwise individual objects are extracted and placed in a separate object cache. These approaches aim to balance the tradeoff between overhead and sensitivity to clustering. Semantic caching takes the different approach of using predicates to dynamically group tuples.

The caching of results based on projections (rather than selections) was studied in [CKSV86]. However, the work most closely related to ours is the predicate caching approach of Keller and Basu [KB96], which uses a collection of possibly overlapping constraint formulas, derived from queries, to describe client cache contents. Our work differs from [KB96] in three significant respects. First, in [KB96] there is no concept analogous to a *semantic region*. Recall that maintaining semantic regions allows, in particular, the use of sophisticated value functions incorporating semantic notions of locality. For discarding cached tuples, Keller and Basu use instead, a reference counting approach based on the number of predicates satisfied by the tuple. Second, the focus of [KB96] is largely on the effects of database updates. Third, [KB96] does not present any performance results to validate their heuristics.

Making use of the tuples in the cache can be viewed as a simple case of "using materialized views to answer queries". This topic has been the subject of considerable study in the literature (e.g., [YL87, CR94, CKPS95, LMSS95]). None of these studies, however, considered the issue of which views to cache/materialize given a limited sized cache, or the performance implications of view usability in a client-server architecture.

ADMS [CR94, R+95] caches the results of subquery expressions corresponding to join nodes in the evaluation tree of each user query. Subsequent queries are optimized by using previously cached views, so query matching plays an important role. Cache replacement is performed by tossing out *entire* views. Determining relevant data in the cache is considerably simpler in our approach, since only base-tuples of individual relations are cached.

8 Conclusions and Future Work

We proposed a semantic model for data caching and replacement that integrates support for associative queries into an architecture based on data-shipping. We identified and studied the main factors that impact the performance of semantic caching compared to traditional page caching and tuple caching in a query-intensive environment: unit of cache management, remainder queries vs. faulting, and cache replacement policy. Semantic caching maintains replacement information with semantic regions that can be dynamically adjusted to the needs of the current queries, uses remainder queries to reduce the communication between the client and server, and enables the use of semantic locality in the cache replacement policy.

We considered selection queries in our study, and are currently exploring the use of semantic caching for complex query workloads. Semantic caching discards entire regions from the cache, often resulting in poor cache utilization; we are investigating the use of region "shrinking" as a technique to alleviate this problem. In this study, we focused on query-intensive environments; exploring the impact of updates is necessary to make these techniques applicable to a larger class of applications. We studied the utility of conventional value functions (e.g., LRU and MRU), as well as of some semantic value functions (e.g., Manhattan distance and its directional variant) in traditional workloads as well as a mobile navigation workload. Our plans for future work include the further development of semantic value functions for this and other applications as well.

References

[Bro92] K. Brown. PRPL: A database workload specification language, v1.3. M.S. thesis, Univ. of WI, Madison, 1992.

[C+94] M. Carey, et al. Shoring up persistent applications. *Proc. ACM SIGMOD Conf.*, 1994.

[CFZ94] M. Carey, M. Franklin, M. Zaharioudakis, Fine-grained sharing in page server database systems, *Proc. ACM SIGMOD Conf.*, 1994.

[CKPS95] S. Chaudhuri, R. Krishnamurthy, S. Potamianos, K. Shim. Optimizing queries with materialized views. *Proc. of IEEE Conf. on Data Engineering*, 1995.

[CKSV86] G. P. Copeland, S. N. Khosafian, M. G. Smith, P. Valduriez. Buffering schemes for permanent data. *Proc. of IEEE Conf. on Data Engineering*, 1986.

[CR94] C. Chen, N. Roussopoulos. Implementation and performance evaluation of the ADMS query optimizer: Integrating query result caching and matching. *Proc. EDBT Conf.* 1994.

[DFMV90] D. DeWitt, P. Futtersack, D. Maier, F. Velez. A study of three alternative workstation-server architectures for object-oriented database systems, *Proc. VLDB Conf.*, 1990.

[D+96] S. Dar, et al. Columbus: Providing information and navigation services to mobile users. Submitted, 1996.

[Fra96] M. Franklin, *Client data caching: A foundation for high performance object database systems*, Kluwer, 1996.

[FJK96] M. Franklin, B. Jónsson, D. Kossmann. Performance tradeoffs for client-server query processing. *Proc. ACM SIGMOD Conf.*, 1996.

[GR93] J. Gray, A. Reuter. *Transaction processing: Concepts and techniques*. Morgan Kaufmann, 1993.

[Jag90] H.V. Jagadish. Linear clustering of objects with multiple attributes. *Proc. ACM SIGMOD Conf.*, 1990.

[KB96] A. Keller, J. Basu. A predicate-based caching scheme for client-server database architectures. *VLDB J*, 5(1), 1996.

[KK94] A. Kemper, D. Kossmann. Dual-buffering strategies in object bases. *Proc. VLDB Conf.*, 1994.

[LMSS95] A. Y. Levy, A. O. Mendelzon, Y. Sagiv, D. Srivastava. Answering queries using views. *Proc. PODS Conf.*, 1995.

[OTS94] J. O'Toole, L. Shrira. Hybrid caching for large scale object systems. *Proc. 6th Wkshp on Pers. Object Sys.*, 1994.

[R+95] N. Roussopoulos, et al. The ADMS project: Views "R" Us. *IEEE Data Engineering Bulletin*, June 1995.

[RK86] N. Roussopoulos, H. Kang. Principles and techniques in the design of ADMS+-. *IEEE Computer*, December, 1986.

[YL87] H. Z. Yang, P.-A. Larson. Query transformation for PSJ-queries. *Proc. VLDB Conf.*, 1987.