# A Self-managing Data Cache for Edge-Of-Network Web Applications

Khalil Amiri
IBM T. J. Watson Research Center,
Hawthorne, NY
amirik@us.ibm.com

Sanghyun Park*
Pohang University of Science and Technology,
Pohang, Korea
sanghyun@postech.ac.kr

Renu Tewari
IBM Almaden Research Center,
San Jose, CA
tewarir@us.ibm.com

## ABSTRACT

*Database caching at proxy servers enables dynamic content to be generated at the edge of the network, thereby improving the scalability and response time of web applications. The scale of deployment of edge servers coupled with the rising costs of their administration demand that such caching middleware be adaptive and self-managing. To achieve this, a cache must be dynamically populated and pruned based on the application query stream and access pattern. In this paper, we describe such a cache which maintains a large number of materialized views of previous query results. Cached "views" share physical storage to avoid redundancy, and are usually added and evicted dynamically to adapt to the current workload and to available resources. These two properties of large scale (large number of cached views) and overlapping storage introduce several challenges to query matching and storage management which are not addressed by traditional approaches. In this paper, we describe an edge data cache architecture with a flexible query matching algorithm and a novel storage management policy which work well in such an environment. We perform an evaluation of a prototype of such an architecture using the TPC-W benchmark and find that it reduces query response times by up to 75%, while reducing network and server load.*

## Categories and Subject Descriptors

H.2.4. [**Database Management**]: Systems—*Distributed databases, query processing*; H.3.5 [**Information storage and Retrieval**]: Online Information Services

## General Terms

Algorithms, Management, Performance

*The author performed this work while at the IBM T. J. Watson Research Center


*CIKM'02,* November 4–9, 2002, McLean, Virginia, USA.


## Keywords

E-commerce, Dynamic content, Semantic caching

## 1. INTRODUCTION

As the WWW evolves into an enabling tool for information gathering, business, shopping and advertising, the data accessed via the Web is becoming more dynamic, often generated on-the-fly in response to a user request or customer profile. Examples of such dynamically generated data include personalized web pages, targeted advertisements and online e-commerce interactions. Dynamic page generation can either use a scripting language (e.g., JSP) to dynamically assemble pages from fragments based on the information in the request (e.g., cookies in the HTTP header), or require a back-end database access to generate the data (e.g., a form-based query at a shopping site).

In order to enhance the scalability of dynamic content serving, enterprise software and content distribution networks (CDNs) solutions, e.g., Akamai's EdgeSuite [2] and IBM's Websphere Edge Server [12] are offloading application components— such as servlets, Java Server Pages, Enterprise Beans, and page assembly—that usually execute at the back-end, to edge servers. Edge servers, in this context, include client-side proxies at the edge of the network, server-side reverse proxies at the edge of the enterprise, or more recently the CDN nodes [2]. These edge servers are considered as an extension of the "trusted" back-end server environment since they are either within the same administrative domain as the back-end, or within a CDN contracting with the content provider. Edge execution not only increases the scalability of the back-end, it reduces the client response latency and avoids over-provisioning of back-end resources as the edge resources can be shared across sites.

In such an architecture, the edge server acts as an application server proxy by handling some dynamic requests locally and forwarding others that cannot be serviced to the back-end. Application components executing at the edge access the back-end database through a standardized interface (e.g., JDBC). The combination of dynamic data and edge-of-network application execution introduces a new challenge. If data is still fetched from the remote back-end, then the increase in scalability due to distributed execution is marginal. Consequently, caching—the proven technique for improving scalability and performance in large-scale distributed systems—needs to evolve to support structured data caching at the edge.

In this paper, we design a practical architecture for a self-managing data cache at the edge server to further accelerate Web applications. We propose and implement a solution, called DBProxy, which intercepts JDBC SQL requests to check if they can be satisfied locally. DBProxy's local data cache is a stand-alone database engine that maintains partial but semantically consistent materialized views of previous query results. The underlying goal in the DBProxy design is to achieve the seamless deployment of a self-managing proxy cache along with the power of a database. DBProxy, therefore, does not require any change to the application server or the back-end database. The work reported in this paper builds upon the rich literature on materialized views [6, 16, 5, 28, 21, 11, 1], specifically focusing on the management of a large number of changing and overlapping views. In particular, the approach we report in this paper contains several novel aspects:

- DBProxy decides *dynamically* what "views" are worth caching for a given workload without any administrator directive; new views are added on the fly and others removed to save space and improve execution time.
- To handle a large number of query results (views), a scalable expression matching engine is used to decide whether a query can be answered from the union of cached views.
- To avoid significant storage redundancy overhead when views overlap, DBProxy employs a novel *common schema table storage policy* such that results from the same base table are stored in a common local table as far as possible.
- The storage and consistency maintenance policies of DBProxy introduce new challenges to cache replacement because cached data is shared across several cached views complicating the task of view eviction.

We report on an evaluation of DBProxy using the TPC-W benchmark which shows that it can speed-up query response time by a factor of 1.15 to 4, depending on server load and workload characteristics.

The rest of this paper is organized as follows. Section 2 reviews background information and previous work in this area. Section 3 describes the design and implementation details of DBProxy, focusing on its novel aspects. Section 4 reports on an evaluation of the performance of DBProxy and Section 5 summarizes our conclusions.

## 2. BACKGROUND

Current large e-commerce sites are organized according to a 3-tier architecture—a front-end web server, a middle-tier application server and a back-end database. While offloading application server components to edge servers can improve scalability, the real performance benefit remains limited by the back-end database access. Moreover, application offloading does not eliminate wide-area network accesses required between the edge server and the database server. For further improvement in access latency and scalability data needs to be cached at the edge servers.

There are two commonly used approaches to data caching at the edge server. One requires an administrator to configure what views or tables are to be explicitly prefetched. The other dynamically caches query results but stores them in unstructured forms. DBProxy manages to combine both the dynamic caching of query responses and the compactness of common table storage.

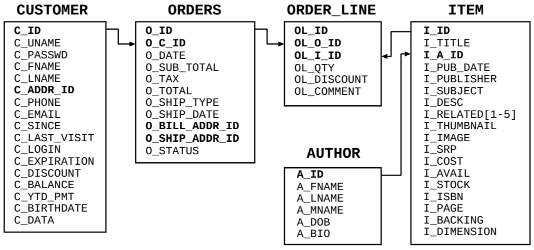


**Figure 1: Simplified database schema of an on-line bookstore used by the TPC-W benchmark.**

### 2.1 Previous work

There has been a wealth of literature on web content caching, database replication and caching, and materialized views. Caching of static Web data has been extensively studied [27, 10, 26, 25] and deployed commercially [2]. A direct extension of static caching techniques to handle the caching of database-generated dynamic content, called passive or *exact-match* caching, simply stores the results of a web query in an unstructured format (e.g., HTML or XML) and indexes it by the request message (e.g., the exact URL string or exact header tags) [18]. Such exact-match schemes have multiple deficiencies. First, their applicability is limited to satisfying requests for previously specified input parameters. Second, there is usually a mismatch in the formats of the data stored at the cache (usually as HTML or XML) and the database server (base tables) which complicates the task of consistency maintenance. Finally, space is not managed efficiently due to the redundant storage of multiple copies of the same data as part of different query results.

The limitations of dynamic content caching solutions and the importance of database scalability over the web have prompted much industry interest in data caching and distribution. Database caching schemes, using full or partial table replication, have been proposed [20, 24, 17]. These are heavy-weight schemes that require significant administrative and maintenance costs and incur a large space overhead. Full database replication is often undesirable because edge servers usually have limited resources, are often widely deployed in large numbers, and can share the cache space among a number of back-end databases.

Materialized views offer another approach to data caching [5, 11]. They use a separate table to storage each view and rely on an administrator for view definition [28]. Materialized views are an effective technique for data caching [5, 11]. Their effectiveness in caching data on the edge hinges, however, on the "proper view" being defined at each edge cache. In some environments, where each edge server is dedicated to a different community or geographic region, determining the proper view to cache can require careful monitoring of the workload, involve complex trade-offs, and presume knowledge of the current resource availability at each edge server. When resources and workloads exhibit changes over time, for example, in response to advertisement campaigns

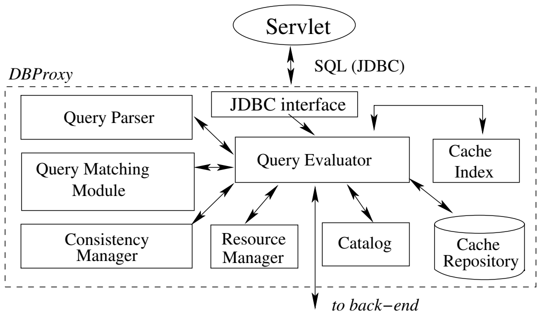


**Figure 2: DBProxy key components.**

or recent news, the views cached may have to be adapted by the administrator. In a dynamic and self-managing environment, it is undesirable to require an administrator to specify and adapt view definitions as the workload changes. Moreover, the large number of views that are usually maintained by the edge cache requires different storage and query matching strategies.

Exact-match based query response caches eliminate the need for administrator control by dynamically caching data, but store data in separate tables for exact matching, thereby, resulting in excess storage redundancy. Semantic caches have been proposed in client-server database systems, to automatically populate client caches without the need for an administrator. The contents of the client cache are described logically by a set of "semantic regions", expressed as query predicates over base tables as opposed to a list of the physical pages [8]. The semantic cache proposal, however, is limited to read-only databases and its query matching and replacement algorithms were only evaluated in a simulation. While web applications are dominated by reads and can tolerate weaker consistency semantics, updates still have to be supported. A simplified form of semantic caching targeting web workloads and using queries expressed through HTML forms has been recently proposed [18]. The caching of query results has also been proposed for specific applications, such as high-volume major event websites [9, 14]. The set of queries in such sites is known a priori and the results of such queries are updated and pushed by the origin server whenever the base data changes.

# 3. ARCHITECTURE AND ALGORITHMS

The architecture of DBProxy assumes that the application components are running on the edge server (e.g., using the IBM Websphere Edge Server [12]). The edge server receives HTTP client requests and processes them locally; passing requests for dynamic content to application components which in turn access the database through a JDBC driver. The JDBC driver manages remote connections from the edge server to the back-end database server, and simplifies application data access by buffering result sets, and allowing scrolling and updates to be performed on them. DBProxy is implemented as a JDBC driver which is loaded by edge applications. It therefore transparently intercepts application SQL calls and determines if they can be satisfied from the local cache.

As shown in Figure 2, the cache functionality is contained in several components. The *query evaluator* is the core module in DBProxy and contains the caching logic. It determines whether an access is a hit or a miss by invoking a *query matching module* which takes the query constraint and its other clauses as arguments. The query evaluator also decides whether the results returned by the back-end on a miss should be inserted in the cache. It rewrites the queries that miss in the cache before passing them to the back-end to prefetch data and improve cache performance. The *resource manager* maintains statistics about hit rates and response times, and adapts cache contents and configuration parameters accordingly. The *consistency manager* is responsible for maintaining cache consistency. The remainder of this section focuses on the query matching engine and the storage scheme (cache repository) employed in DBProxy.

## 3.1 Scalable query matching

To handle a large and varying set of cached views, the query matching engine of DBProxy must be highly optimized to ensure a fast response time for hits. Cached queries are organized according to a multi-level index based on the schema, table list, and columns accessed. Furthermore, the queries are organized into different classes based on their structure. Query mixes found in practice not only include simple queries over a single table and join-queries, but also Top-N queries, aggregation operators and group-by clauses, and complex predicates including UDFs and subqueries. A query matching algorithm, optimized for each query class, is invoked to determine if a query is a hit or a miss. The corresponding hit and miss processing, that includes query rewriting, is also optimized for each query class.

### *3.1.1 Query matching algorithm*

When a new query $Q_n$ comes in, the query matching engine retrieves the list of cached queries which operated on the same schema and on the same table(s). The list of retrieved queries is searched to see if one passes the attribute and tuple coverage tests, described below. If there exists a cached query $Q_c$ which passes these two tests with respect to $Q_n$, then $Q_n$ is declared to be a hit, otherwise it is declared a miss.

### *3.1.2 Attribute coverage test*

Attribute coverage means that the columns referred to by the various clauses of a newly received query must be a subset of the columns fetched by the cached query. This test is implemented as follows. For each cached query $Q_c$, we associate a set of column names mentioned in any clause of $Q_c$ (e.g., the SELECT, WHERE clause). Given a cached query $Q_c$ and a new query $Q_n$, we check that the column list of $Q_n$ is contained in the column list of $Q_c$. Because the number of cached queries can be quite large, an efficient algorithm is needed to quickly check what column lists *include* a new query's column list.

Let $m$ be the maximum number of columns in a table $T$. We associate with a query $Q_c$ to table $T$ an $m$-bit signature, denoted by $bitsig(Q_c)$, which represents the columns referred to by the various clauses of $Q_c$. The $i$-th bit in the signature is set if the $i$-th column of $T$ appears in one of the query's clauses. This assumes a canonical order among the columns of table $T$. For example, one ordering criterion is the column definition order in the initial '`create table`' statement. Suppose a table has four columns $A, B, C$, and $D$. Consider a query that selects columns $B$ and $D$, the bit-string signature associated with the query would be 0101.

Using the bit string signature, the problem of verifying that the column list mentioned in a query $Q_n$ is included in the column list mentioned in a query $Q_c$ can be reduced to a bit-string-based test:

$$bitsig(Q_n) \ \&\& \ bitsig(Q_c) \ = \ bitsig(Q_n)$$

The operator *&&* denotes the bitwise AND operator between the two bit-strings of the same length. Therefore, the problem of attribute coverage is reduced to a linear scan of cached queries, where the bit-string signature of $Q_n$ is bitwise ANDed with that the of the cached query to determine the set of queries that subsume the columns fetched by $Q_n$. Since query signatures transform the problem of attribute coverage into a string search problem, a more scalable approach is to organize the bit-strings of cached queries into a *trie* [23] for quick search by an incoming new signature.

### 3.1.3 Tuple coverage test

When a new query is first received by the cache, it is parsed into its constituent clauses. In particular, the query's WHERE clause is stored as a boolean expression that represents the constraint predicate. The leaf elements of the expression are simple predicates of the form "col *op* value" or "col *op* col". These predicates can be connected by AND, OR, or NOT nodes to make arbitrarily complex expressions. Once the query is parsed, the cache index is accessed to retrieve the set of queries that operated on the same tables and retrieved a super-set of the columns required by the new query. A query matcher is invoked to verify whether the result set of this new query is contained in the union of the results of the previously cached candidate queries.

**Tuple coverage checking algorithm.** The result of query $Q_B$ is contained in that of query $Q_A$ if for all possible values of a tuple $t$ in the database, the WHERE predicate of $Q_B$ logically implies that of $Q_A$. That is:

$$\forall(t) \quad Q_B.wherep(t) \Rightarrow Q_A.wherep(t)$$

This is equivalent to the statement that the product expression $Q_B.wherep \ \ AND \ \ (NOT(Q_A.wherep))$ is *unsatisfiable*. Checking for the containment of one query in the union of multiple queries simply requires considering the logical OR of the WHERE predicates of cached queries for the unsatisfiability verification.

The containment checking algorithm's complexity is polynomial (of degree 2) in the number of *columns* that appear in the expression. It is based on the algorithm described in [22, 15] which decides satisfiability for expressions with integer-based attributes. We extend that algorithm to handle floating point and string types. Floating point types introduce complexities because the plausible intervals for an attribute can be open. Our algorithm is capable of checking containment for clauses that include disjunctive and conjunctive predicates including a combination of numeric range, string range and set membership predicates.

**Top-N Queries.** Top-N queries fetch a specified number of rows from the beginning of a result set which is usually sorted according to an order-by clause, as in these two queries:

| | | | |
|---|---|---|---|
| $Q_3$: | SELECT | $i\_id$ FROM $item$ | WHERE $i\_cost < 15$ |
| | ORDER BY | $i\_cost$ | |
| | FETCH FIRST | 20 ROWS ONLY | |
| $Q_4$: | SELECT | $i\_id$ FROM $item$ | WHERE $i\_cost < 5$ |
| | ORDER BY | $i\_cost$ | |
| | FETCH FIRST | 20 ROWS ONLY | |

The query containment and evaluation algorithm for Top-N queries differs from that of simple (full-result) queries. Top-N queries are converted to simple queries by dropping the "Fetch First" clause. Containment checking is performed on these modified queries to verify if the predicates are contained in the union of cached queries which have the same order-by clause. Note that although the base query $Q_4$ is subsumed by the results of query $Q_3$ when the "Fetch First" clause is ignored, containment is not guaranteed when the Fetch clauses are considered. The second step in processing containment checking for a Top-N query is to ensure that enough rows exist in the local cache to answer the new query. DBProxy attempts local execution on the cached data. If enough rows are retrieved, then the query is considered a cache hit, otherwise it is considered a cache miss. We call this type of a cache miss, a "partial hit", since only a part of the result is found in the local cache although the containment check was satisfied.

If a containment check does not find any matching queries or the result set does not have enough rows, the query is sent to the origin for execution. However, before forwarding it, the original query $Q_{orig}$ is re-written into a modified query $Q_{mod}$, increasing the number of rows fetched. This optimization increases $N$ by a few multiples, if $N$ is small relative to the table size, otherwise the "Fetch First" clause is dropped altogether. The default expansion in our experiments was to double the value of $N$. This heuristic is aimed at reducing the event of later "partial hit" queries.

**Aggregation queries.** These are queries which have an aggregation operator (e.g., MAX, MIN, AVG, SUM) in their select list, such as $Q_5$ below:

SELECT MAX($i\_cost$) FROM $item$ WHERE $i\_a\_id = 10$

Aggregation queries can be cached as *exact-match* queries. However, to increase cache effectiveness, DBProxy attempts to convert such queries to regular queries so that they are more effective in fielding cache hits. Upon a cache miss, an aggregate query is sometimes rewritten and the aggregation operator in the select list is dropped. By caching the base row set, new aggregation operators using the same (or a more restrictive) query constraint can be satisfied from the cache. Of course, this is not always desirable since the base set may be too large or not worth caching. Estimates of the base result set are computed by the resource manager and used to decide if such an optimization is worth performing. Otherwise, the query is cached as an exact-match.

**Exact-match (complex) queries.** Some queries are too complex to check for containment because they contain complex clauses (e.g., a UDF or a subquery in the WHERE clause). These queries cannot be cached using the common-table approach. For such queries, we use a different storage and query-matching strategy. The results of such queries are stored in separate result tables, indexed by the query's exact SQL string. Note that such query matching is inexpensive, requiring minimal processing overhead.

**Query containment types.** In general, a query can be either (i) completely subsumed by a single previously cached view, (ii) subsumed by a union of more than one previously cached view, or (iii) partially contained in one or more cached views. The query matching module can discover if one of the first two cases occur, declaring it a cache hit. The third case is treated as a miss. While it is possible to perform a local query and send a remainder query to the back-end, we noticed that it complicates consistency maintenance and offers little actual benefit since execution is still dominated by the latency of remote execution.

### 3.1.4 Hit and Miss Processing

Upon a miss, queries are rewritten before they are sent to the origin site. Specifically, the following optimizations are performed. First, the *select list* is expanded to include the columns mentioned in the WHERE-clause, ORDER-BY-clause, and HAVING-clause. The goal of this expansion is to maximize the likelihood of being able to execute future queries that hit in the cache [1]. Second, the *primary key column is added to the select list*, as discussed previously. For instance, the following query:

```
SELECT    i_avail, i_cost  FROM item
WHERE     i_cost < 5
ORDER BY  i_pub_date
```

would be sent to the server as:

```
SELECT    i_id, i_pub_date, i_avail, i_cost  FROM item
WHERE     i_cost < 5
ORDER BY  i_pub_date
```

Note that the results returned by the back-end are not always inserted in the cache upon a miss. Statistics collected by the resource manager are used to decide which queries are worth caching.

## 3.2 Shared storage policy

Data in a DBProxy edge cache is stored persistently in a *local stand-alone database*. The contents of the edge cache are described by a cache index containing the list of queries. To achieve space efficiency, data is stored in *common-schema tables* whenever possible such that multiple query results share the same physical storage. Queries over the same base table are stored in a single, usually partially populated, cached copy of the base table. Join queries with the same "join condition" and over the same base table list are also stored in the same local table. This scheme not only achieves space efficiency but also simplifies the task of consistency maintenance because data in the cache and in the origin server is represented in the same format. In most cases, there is a one-to-one correspondence between a locally cached table and a base table in the origin server, although the local copy may be partially and sparsely populated. A local result table is created with as many columns as selected by the query. The column type and metadata information are retrieved from the back-end server and cached in a local catalog cache.

As an example, consider the 'item' table shown in Figure 1. It has 22 columns with the column *i_id* as a primary key. Figure 3 shows the shared local table after inserting the results of two queries $Q_1$ and $Q_2$ in the locally cached copy of the item table. This local table can be considered

[1]Note that although the result may be contained in the cache, a query may not be able to sub-select its results from the cache if the columns referenced in its clauses are not present.

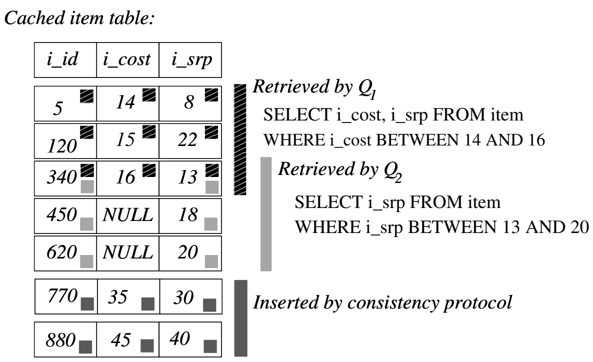


**Figure 3: Local storage. The local *item* table entries after the queries $Q_1$ and $Q_2$ are inserted in the cache. The first three rows are fetched by $Q_1$ and the middle three are fetched by $Q_2$. Since $Q_2$ did not fetch the *i_cost* column, NULL values are inserted.**

as a vertical and horizontal subset of the back end 'item' table. Both queries are rewritten if necessary to expand their select list to include the primary key, *i_id*, of the table so that locally cached rows can be uniquely identified. This avoids storing duplicate rows in the local tables when the result sets of queries overlap. The local 'item' table is created just before inserting the three rows retrieved by $Q_1$ with the primary key column and the two columns requested by the query (*i_cost* and *i_srp*). Since the table is assumed initially empty, all insertions complete successfully. To insert the three rows retrieved by $Q_2$, we first check if the table has to be *altered* to add any new columns (not true in this case). Next, an attempt is made to insert the three rows fetched by $Q_2$.

Note from Figure 3 that the row with $i\_id = 340$ has already been inserted by $Q_1$, and so the insert attempt would raise an exception. In this case, the insert is changed to an update statement. Also, since $Q_2$ did not select the *i_cost* column, a NULL value is inserted for that column. Sometimes, a query will not fetch columns that are defined in the locally cached table where its results will be inserted. In such a case, the values of the columns that are not fetched are set to NULL. The presence of "fake" NULL values in local tables raises the concern that such values may be exposed to application queries. However, as we show later, DBProxy does not return incorrect (e.g., false NULLs) or incomplete results, even in the presence of updates from the origin and background cache cleaning.

CLAIM 1. *No query will return an incorrect (or spurious) NULL value from the cache for read-only tables.*

Proof Sketch: (By contradiction) Assume a new query $Q_n$ is declared by the containment checker as a hit. Suppose that its local execution returns a NULL value (inserted locally as a "filler") for a column, say $C$ and a row say $R$, that has a *different* value in the back-end table. Since $Q_n$ is a cache hit, there must exist a cached query $Q_c$ which contains $Q_n$. Since row $R$ was retrieved by $Q_n$, it must also belong to $Q_c$ (by *tuple coverage*). Since column $C$ is in the select list of $Q_n$, it must also belong to the select list of $Q_c$ (by *attribute coverage*). It follows that that the value stored locally in column $C$ of row $R$ was fetched from the back-end when the results of $Q_c$ were inserted. Since the table is read-only, the value should be consistent with the back-end. □

In fact, the above claim holds also for non read-only tables where updates to the base table are propagated from the back-end to the cache. A proof of this stronger result requires a deeper discussion of the update propagation and consistency protocols, and can be found in the associated technical report [3].

**Table creation.** To reduce the number of times the schema of a locally cached table is altered, our approach consists of observing the application's query stream initially until some history is available to guide the initial definition of the local tables. The local table's schema definition is a tradeoff between the space overhead of using the entire back-end schema (where space for the columns is allocated but not used) and the overhead of schema alterations. If it is observed that a large fraction of the columns need to be cached then the schema used is the same as that of the back-end table. For example, after observing the list of queries (of Section 3.1.4) to the item table, the following local table is created:

CREATE TABLE *local.item*
(*i_id* INTEGER NOT NULL, *i_avail* DATE,
*i_cost* DOUBLE, *i_pub_date* DATE, PRIMARY KEY (*i_id*) )

### *3.2.1 Update propagation and view eviction*

Update transactions are always forwarded to the back-end database for execution, without first applying them to the local cache. This is often a practical requirement from site administrators who are concerned about durability. This choice is acceptable because update performance is not a major issue. Over 90% of e-commerce interactions are browsing (read-only) operations [7].

Read-only transactions are satisfied from the cache whenever possible. DBProxy ensures data consistency by subscribing to a stream of updates propagated by the origin server. When the cache is known to be inconsistent, queries are routed to the origin server by DBProxy transparently.

Traditional materialized view approaches update cached views by re-executing the view definition against the change ("delta") in the base data. Similarly, predicate-based caches match tuples propagated as a result of UDIs performed at the origin against all cached query predicates to determine if they should be inserted in the cache [13]. DBProxy inserts updates to cached tables without evaluating whether they match the cached predicates, thereby, avoiding the overhead of reverse matching each tuple with the set of cached queries. Updates, deletes and inserts (UDIs) to base tables are propagated and applied to the partially populated counterparts on the edge. This optimization is important because it reduces the time delay after which updates are visible in the local cache, thereby increasing its effectiveness. Because little processing occurs in the path of propagations to the cache, this scheme scales as the number of cached views increases.

To avoid unnecessary growth in the local database size, a background replacement process evicts excess tuples in the cache that do not match any cached predicates.

Claim 2. *DBProxy consistency management ensures that the local database undergoes monotonic state transitions.*

This implies that the state of the database exported by DBProxy moves only forward in time. The above guarantee is required to ensure that if the user sees data from the back-end database on a miss, then in future the user should not see stale data from the local database on a subsequent hit.

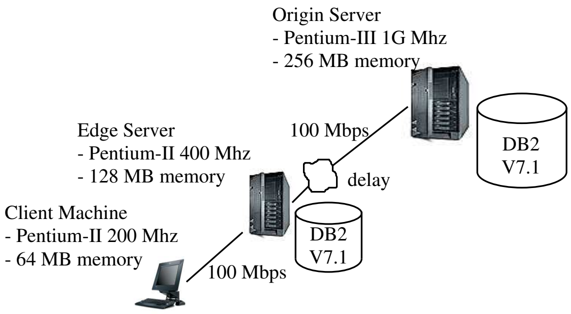


**Figure 4: Evaluation environment. Three machines were used in the evaluation testbed based on the TPC-W benchmark. The client machine was used to run the workload generator, which applied a load equivalent to 8 emulated browsers. All machines ran Linux RedHat 7.1. Apache-Tomcat 3.2 was used as the app server.**

Claim 3. *DBProxy consistency management ensures immediate visibility of updates.*

This claim implies that if an application commits an update and later issues a query, the query should observe the effects of the update (and all previous updates). The consistency guarantees provided by DBProxy and the associated proofs are discussed in an associated research report [3].

The background replacement process is also used to evict rows that belong to queries (views) marked for eviction. Recall that evicting a view from the cache is complicated by the fact that the underlying tuples are shared across views. Because these tuples may be updated after they are inserted, it is not simple to keep track of which tuples in the cache match which cached views. In particular, the replacement process must guarantee the following properties:

Property 1. *When evicting a victim query from the cache, the underlying tuples that belong to the query can be deleted only if no other query that remains in the cache accesses the same tuples.*

Property 2. *The tuples inserted by the consistency manager that do not belong to any of the results of the cached queries are eventually garbage collected.*

The background process executes the queries to be maintained in the cache, marking any accessed tuples, and then evicting all tuples that remain unmarked at the end of the execution cycle. The query executions are performed in the background over an extended period of time to minimize interference with foreground cache activity. A longer discussion of space management in DBProxy can be found in [4].

## 4. EXPERIMENTAL EVALUATION

In this section, we present an experimental study of the performance impact of data caching on edge servers. We implemented the caching functionality transparently by embedding it within a modified JDBC driver. Exceptions that occur during the local processing of a SQL statement are caught by the driver and the statement is forwarded to the back-end. The results returned from the local cache and from the remote server appear indistinguishable to the application.

**Evaluation methodology and environment.** Figure 4 represents a sketch of the evaluation environment. Three machines are used to emulate client browsers, an edge server and an origin server respectively. A machine more powerful than the edge server is used for the origin server (details in Figure 4). A fast Ethernet network (100 Mbps) was used to connect the three machines. To emulate wide-area network conditions between the edge and the back-end server, we artificially introduce a delay of 225 milliseconds for remote queries. This represents the application end-to-end latency to communicate over the wide area network, assuming typical wide area bandwidths and latencies. DB2 V7.1 was used in the back-end database server and as the local cache in the edge server.

We used the TPC-W e-commerce benchmark in our evaluation. In particular, we used a Java implementation of the benchmark from the University of Wisconsin [19]. The TPC-W benchmark emulates an online bookstore application, such as *amazon.com*. The browsing mix of the benchmark includes search queries about best-selling books within an author or subject category, queries about related items to a particular book, as well as queries about customer information and order status. However, the benchmark has no numeric or alphabetical range queries. The WHERE clause of most queries is based on atomic predicates employing the equality operator and connected with a conjunctive or a disjunctive boolean operator. We used this standard benchmark to evaluate our cache first. Then, we modified the benchmark slightly to reflect the type of *range queries* that are common in other applications, such as auction warehouses, real-estate or travel fare searches, and geographic area based queries. To minimize the change to the logic of the benchmark, we modified a single query category, the *get-related-items* query. In the standard implementation, items related to a particular book are stored in the table explicitly in a priori defined columns. Inquiring about related items to a particular book therefore simply requires inspecting these items, using a self-join of the item table. To model range queries, we change the get-related-items query to search for items that are within 5-20% of the cost of the main item. The query returns only the top ten such items (ordered by cost). The actual percentage varies within the 5-20% range and is selected using a uniform random distribution to model varying user input:

```
SELECT     i_id, i_thumbnail     FROM item
WHERE      i_cost                BETWEEN 284 AND 340
ORDER BY   i_cost FETCH FIRST    10 ROWS ONLY
```

The range in the query is calculated from the cost of the main item which is obtained by issuing a (simple) query. Throughout the graphs reported in this section, we refer to the browsing mix of the standard TPC-W benchmark as *TPC-W* and the modified benchmark as *Modified-TPC-W*, or *Mod* for short. We report baseline performance numbers for a 10,000 (10K) and 100,000 (100K) item database. Then, we investigate the effect of higher load on the origin server, skew in the access pattern, and dynamic changes in the workload on cache performance.

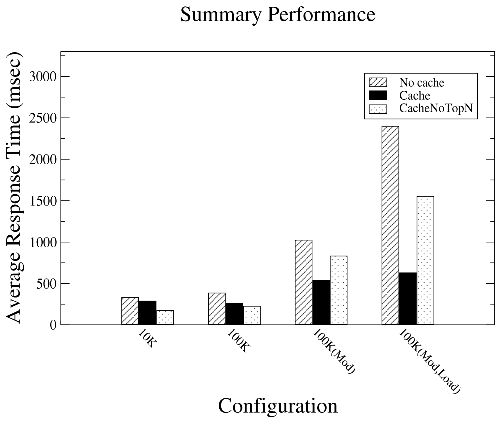


**Figure 5: Summary of cache performance using the TPC-W benchmark. The four cases are: (i) Base TPC-W with a 10K item database, (ii) Base TPC-W with a 100K item database, (iii) Modified TPC-W with a 100K item database, (iv) Modified TPC-W with a 100K item database and a loaded back-end.**

We focus on hit rate and response time as the key measures of performance. We measure query response time as observed by the application (servlets) executing at the edge. Throughout the experiments reported in this section, the cache replacement policy was not triggered. The TPC-W benchmark was executed for an hour (3600 seconds) with measurements reported for the last 1000 seconds or so. The reported measurements were collected after the system was warmed up with 4000 queries.

**Baseline performance impact.** Figure 5 graphs average response time across all queries for four configurations: (i) TPC-W with a 10K item database, (ii) TPC-W with a 100K item database, (iii) Modified-TPC-W with 100K item database, and (iv) Modified-TPC-W with 100K item database and a loaded origin server. There are three measurements for each configuration: (i) no cache (all queries to back-end), (ii) cache all, and (iii) cache no Top-N queries. Consider first the baseline TPC-W benchmark, or the two-leftmost configurations in the figure. The figure shows that response time improvement is significant, ranging from 13.5% for the 10K case to 31.7% for the 100K case. Note that the improvement is better for the larger database, which may seem at first counter-intuitive since the hit rate should be higher for smaller databases. We observed, however, that the local insertion time for the 10K database is higher than that for the 100K case due to the higher contention on the local database.

The performance impact when the caching of Top-N queries is disabled is given by the rightmost bar graph in each of the configurations of Figure 5 (CacheNoTopN). Note that depending on the load and database size, caching these queries which happen also to be predominantly multi-table joins, may increase or decrease performance. DBProxy's resource manager collects response time statistics and can turn on and off the caching of different queries based on observed performance. The effect of server load (the rightmost configuration in Figure 5) is discussed below.

**Hit rate versus response time.** The remaining experimental results focus on the 100K database size. Table 1

| | Baseline TPC-W | | | Modified TPC-W | | |
|---|---|---|---|---|---|---|
| Query Category | Response time | Hit rate | Query frequency | Response time | Hit rate | Query frequency |
| Simple | 51 | 91 % | 23 % | 317 | 37 % | 47 % |
| Top-N | 935 | 68 % | 12 % | 852 | 66 % | 37 % |
| Exact-match | 211 | 76 % | 65 % | 458 | 54 % | 15 % |
| Total | 263 | 73 % | 100 % | 540 | 50 % | 100 % |
| No Cache | 385 | – | 100 % | 1024 | – | 100 % |

**Table 1: Cache performance under the baseline and modified TPC-W benchmarks. Database size was 100K items and 80K customers. The workload included 8 emulated browsers executing the browsing mix of TPC-W.**

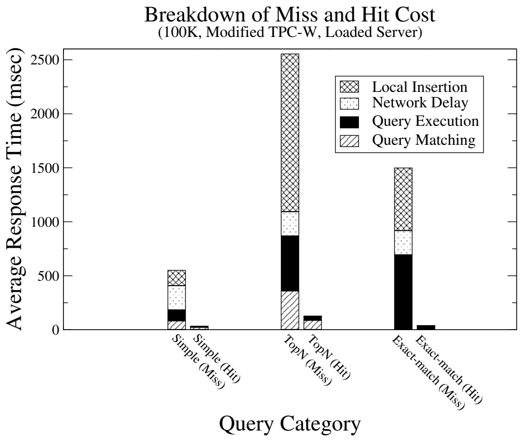


**Figure 6: Breakdown of miss and hit cost under modified TPC-W with a loaded origin server.**

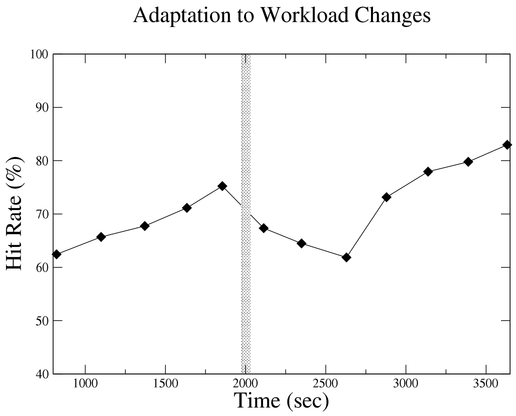


**Figure 7: Dynamic adaptation to workload changes. Cache hit rate recovers after a sudden change in the workload.**

provides a breakdown of response times and hit rates for each query category for the base TPC-W and Modified-TPC-W benchmarks running against a 100K item database. Hit rates vary by category, with the overall hit rate averaging 73% for the baseline benchmark and 50% for the modified benchmark. Observe that although the hit rates are significantly higher in the baseline TPC-W the response time improvement was higher under the Modified-TPC-W benchmark. This is because Modified-TPC-W introduces queries that have a low hit cost but a high miss cost. The query matching time for the modified TPC-W was also measured to be lower than that of the original TPC-W because the inclusion of the cost-based range query (with the query constraint involves a single column) reduced the average number of columns in a WHERE clause. The table also provides a comparison between the caching configuration and the non-caching configuration in terms of average response times across all query categories (the two bottom rows of the table).

**Effect of server load.** One advantage of caching is that it increases the scalability of the back-end database by serving a large part of the queries at the edge server. This reduces average response time when the back-end server is experiencing high load. Under high-load conditions, performance becomes even more critical because frustrated users are more likely to migrate to other sites. In order to quantify this benefit, we repeated the experiments using a Modified-TPC-W benchmark while placing an additional load on the origin server. The load was created by invoking another additional and equal number of emulated browsers directly against the back-end. This doubled the load on the origin for the non-caching configuration.

The two right-most configurations (100K Mod and 100K Mod,Load) in Figure 5 correspond to the Modified TPC-W benchmark running against a 100K item database with and without additional server load. The graph shows that increased server load resulted in more than doubling the average response time, increasing by a factor of 2.3 for the non-caching configuration. When caching at the edge server was enabled, the average response time increased by only 14%. Overall, the performance was improved by a factor of 4. Figure 6 shows the break-down of the miss and hit cost respectively for the loaded server case. Although not shown in the figure, we measured the remote execution time to double under load. Thus, and despite the cost of query matching in our unoptimized implementation, local execution was still very effective.

**Adaptation to workload changes.** One advantage of caching on-the-fly is that it dynamically determines the active working set, tracking any workload changes without an administrator input. To investigate the adaptation of our caching architecture to workload changes, we executed the following experiment. Using the modified TPC-W benchmark, we invoked a change in the workload in the middle of execution. For the first half of the execution, we used a cost-based get-related-items query. In the second half, we changed the get-related-items query to use the suggested retail price *i_srp* instead of *i_cost* in the get-related-items query. Figure 7 shows that the hit rate dips after the workload changes, but recovers gradually thereafter. The hit rate regains its former level within 17 minutes, which may appear long although that is mostly a reflection of the workload's limited activity rate (2-3 queries per second in this experiment from the 8 emulated browsers).

## 5. CONCLUSIONS

Caching content and offloading application logic to the edge of the network are two promising techniques to improve performance and scalability. While static caching is well understood, much work is needed to enable the caching of dynamic content for Web applications. In this paper, we study the issues associated with a flexible dynamic data caching solution. We describe DBProxy, a stand-alone database engine that maintains partial but semantically consistent materialized views of previous query results. Our cache can be thought of as containing a large number of materialized views which are added on-the-fly in response to queries from the edge application that miss in the cache. It uses an efficient common-schema table storage policy which eliminates redundant data storage whenever possible. The cache replacement mechanisms address the challenges of shared data across views and adjust to operate under various space constraints using a cost-benefit based replacement policy. DBProxy maintains data consistency efficiently while guaranteeing lag consistency and supporting monotonic state transitions. We evaluated the DBProxy cache using the TPC-W benchmark, and found that it results in query response time speedups ranging from a factor of 1.15 to 4, depending on server load and workload characteristics.

## 6. ADDITIONAL AUTHORS

Additional authors: Sriram Padmanabhan (IBM T. J. Watson Research Center) email: `srp@us.ibm.com`

## 7. REFERENCES

[1] F. N. Afrati, C. Li, and J. D. Ullman. Generating efficient plans for queries using views. In *SIGMOD Conference*, pages 319–330, 2001.

[2] Akamai Technologies Inc. Akamai EdgeSuite. http://www.akamai.com/html/en/tc/core_tech.html.

[3] K. Amiri, S. Park, R. Tewari, and S. Padmanabhan. DBProxy: A Self-Managing Edge-of-Network Data Cache. Technical Report RC22419, IBM Research, 2002.

[4] K. Amiri, R. Tewari, S. Park, and S. Padmanabhan. On Space Management in a Dynamic Edge Data Cache. In *Fifth International Workshop on Web and Databases*, 2002.

[5] R. G. Bello, K. Dias, J. Feenan, J. Finnerty, W. D. Norcott, H. Sun, A. Witkowski, and M. Ziauddin. Materialized views in Oracle. In *VLDB Conference*, pages 659–664, 1998.

[6] S. Chaudhuri, S. Krishnamurthy, S. Potamianos, and K. Shim. Optimizing queries using materialized views. In *ICDE Conference*, pages 190–200, 1995.

[7] G. Copeland. Internal presentation. IBM Corporation.

[8] S. Dar, M. J. Franklin, B. T. Jónsson, D. Srivastava, and M. Tan. Semantic data caching and replacement. In *VLDB Conference*, pages 330–341, 1996.

[9] L. Degenaro, A. Iyengar, I. Lipkind, and I. Rouvellou. A middleware system which intelligently caches query results. In *Middleware Conference*, pages 24–44, 2000.

[10] L. Fan, P. Cao, J. Almeida, and A. Broder. Summary cache: A scalable wide-area web cache sharing protocol. In *SIGCOMM Conference*, 1998.

[11] J. Goldstein and P.-A. Larson. Optimizing queries using materialized views: A practical, scalable solution. In *SIGMOD Conference*, pages 331–342, 2001.

[12] IBM Corporation. WebSphere Edge Server. http:-//www.ibm.com/software/webservers/edgeserver/.

[13] A. M. Keller and J. Basu. A predicate-based caching scheme for client-server database architectures. *VLDB Journal*, 5(1):35–47, 1996.

[14] A. Labrinidis and N. Roussopoulos. Update propagation strategies for improving the quality of data on the web. In *VLDB Conference*, pages 391–400, 2001.

[15] P.-A. Larson and H. Z. Yang. Computing queries from derived relations: Theoretical foundations. Technical Report CS-87-35, Department of Computer Science, University of Waterloo, 1987.

[16] A. Levy, A. O. Mendelzon, Y. Sagiv, and D. Srivastava. Answering queries using views. In *PODS Conference*, pages 95–104, 1995.

[17] Q. Luo, S. Krishnamurthy, C. Mohan, H. Pirahesh, H. Woo, B. Lindsay, and J. F. Naughton. Middle-tier database caching for e-business. In *SIGMOD Conference*, 2002.

[18] Q. Luo and J. F. Naughton. Form-based proxy caching for database-backed web sites. In *VLDB Conference*, pages 191–200, 2001.

[19] M. H. Lipasti (University of Wisconsin). Java TPCW Implementation. http://www.ece.wisc.edu/ pharm/tpcw.shtml.

[20] Oracle Corporation. Oracle 9iAS Database Cache. http://www.oracle.com/ip/deploy/ias/docs/cachebwp.pdf.

[21] R. Pottinger and A. Levy. A scalable algorithm for answering queries using views. In *VLDB Conference*, pages 484–495, 2000.

[22] D. J. Rosenkrantz and H. B. Hunt. Processing conjunctive predicates and queries. In *VLDB Conference*, pages 64–72, 1980.

[23] G. A. Stephen. *String Searching Algorithms*. World Scientific Publishing, 1994.

[24] T. T. Team. Mid-tier caching: The timesten approach. In *SIGMOD Conference*, 2002.

[25] R. Tewari, M. Dahlin, H. Vin, and J. Kay. Beyond hierarchies: Design considerations for distributed caching on the internet. In *ICDCS Conference*, 1999.

[26] D. Wessels and K. Claffy. ICP and the Squid Web Cache. *IEEE Journal on Selected Areas in Communication*, 16(3):345–357, 1998.

[27] S. Williams, M. Abrams, C. Standridge, G. Abdulla, and E. Fox. Removal policies in network caches for world-wide web documents. In *SIGCOMM*, 1996.

[28] M. Zaharioudakis, R. Cochrane, G. Lapis, H. Pirahesh, and M. Urata. Answering complex SQL queries using automatic summary tables. In *SIGMOD Conference*, pages 105–116, 2000.