DBProxy: A dynamic data cache for 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

Sriram Padmanabhan
IBM T.J. Watson Research Center
Hawthorne, NY
srp@us.ibm.com

Abstract

The majority of web pages served today are generated dynamically, usually by an application server querying a back-end database. To enhance the scalability of dynamic content serving in large sites, application servers are offloaded to front-end nodes, called edge servers. The improvement from such application offloading is marginal, however, if data is still fetched from the origin database system. To further improve scalability and cut response times, data must be effectively cached on such edge servers. The scale of deployment of edge servers and the rising costs of their administration demand that such caches be self-managing and adaptive. In this paper, we describe DBProxy, an edge-of-network semantic data cache for web applications. DBProxy is designed to adapt to changes in the workload in a transparent and graceful fashion by caching a large number of overlapping and dynamically changing "materialized views". New "views" are added automatically while others may be discarded to save space. In this paper, we discuss the challenges of designing and implementing such a dynamic edge data cache, and describe our proposed solutions.

1 Introduction

Data accessed via the Web is increasingly dynamic, generated on-the-fly in response to a user request or customer profile. Examples of such dynamic data include personalized web pages, targeted advertisements or online e-commerce interactions. Dynamic data is served using a 3-tiered architecture consisting of a web server, an application server and a database; data is stored in the database and is accessed on-demand by the application server components and formatted and delivered to the client by the web server. To improve scalability and performance, caching at edge servers has been widely deployed on the web for static HTML pages. For dynamic content, which requires database accesses, caches are typically by-passed by marking the content uncacheable. Recent work has targeted extending the static caching concept by storing the result of a dynamic web request as HTML fragments or other formats indexed by the exact URL string or HTTP request header. Consistency, limited reuse of cached data, and cache space management, however, can easily gate the scalability of these schemes.

In more recent architectures, the edge server (which collectively refers to client-side proxies, server-side reverse proxies at the edge of the enterprise, or caches within a content distribution network (CDN) [2]) acts as an application server proxy by offloading application components (e.g., servlets, Java Server Pages, Enterprise Beans) to the edge [14]. 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 becomes 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 scal-

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

ability and performance in large-scale distributed systems—needs to evolve to support structured data caching at the edge.

In this paper, we describe a practical architecture for a self-managing data cache at the edge server to further accelerate Web applications. Our prototype, called DBProxy, 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. To avoid the significant overlap among cached query results and storage redundancy, DBProxy employs a *common-schema table storage policy* that stores results in a common local table as far as possible. The common table adds new challenges to cache replacement as the underlying data is shared across results. The cache replacement mechanisms address these challenges and adjust to operate under various space constraints using a cost-benefit based replacement policy.

DBProxy decides dynamically what views are worth caching for a given workload; new views are added on the fly and others removed to save space and improve execution time. Such dynamic changes add new challenges to the consistency maintenance. Furthermore, such a dynamic semantic cache can contain hundreds to thousands of queries. To decide if a new query is a cache hit, it has to be tested for containment against all plausible previously cached queries. Sophisticated query containment algorithms are required that can scale to such large cache sizes. In this paper, we discuss the various challenges to dynamic edge data caching, describe how we address them, and report on a summary evaluation of our proposed approach.

Our discussion of DBProxy in this paper is organized as follows. Section 2 discusses the challenges of adaptive and dynamic data caching. Section 3 describes the design and implementation details of DBProxy. Section 4 reports on a summary evaluation of its performance. Section 5 reviews related work and Section 6 summarizes our conclusions.

2 Challenges of dynamic data caching

While offloading application server components to the edge server can improve scalability, the real performance benefit remains limited by the back-end database access. For further improvement in access latency and scalability data needs to be cached at the edge servers. There are several approaches to data caching on the edge, mainly including full replication and materialized views.

Replication relies on an administrator to define what must be cached. The local tables simply mirror the corresponding back-end base tables. Full database replication is often undesirable because it incurs a large space overhead, and edge server resources are usually limited. Query response caches, on the other hand, 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.

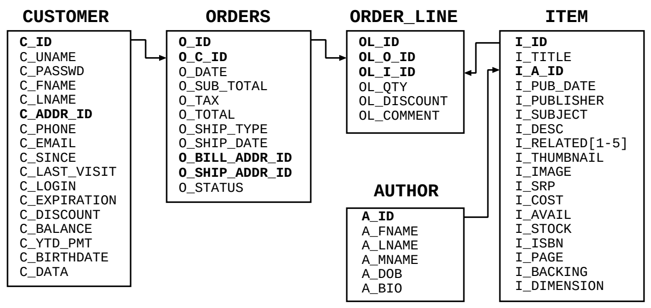

Figure 1. Simplifi ed database schema of an on-line bookstore used by the TPC-W benchmark.

The materialized view approach for data caching [6, 13], uses a separate table for each view, and relies on an administrator for view definition [30]. 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 or recent news, the views cached may have to be adapted by the administrator.

2.1 Edge data caching requirements

The scale of deployment of edge servers and the rising costs of their administration demand that such caches be self-managing and adaptive. More specifically, we distinguish the following key design requirements for a dynamic data cache:

- Database independence: Edge applications can access multiple back-end databases, which can have different schemas and possibly reside in DBMSs from different vendors. As a result, the edge data cache should ideally not assume any particular DBMS or database schema. Furthermore, it should allow data cached from multiple back-ends to share the space resources available at the edge.

- Self-management: The cache should adapt dynamically to the workload and the available resources.

We would like the cache to contain a large number of materialized views, matching the application access patterns. Views are added on-the-fly in response to queries from the edge application that miss in the cache, although the actual added view may be a generalization of an application's query. This eliminates the need for an administrator to decide or optimize the particular materialized view for each edge cache.

- Fast query matching: A dynamic cache can contain a large number of cached queries, potentially hundreds or thousands. At the same time, it must be able to quickly detect if a new query is a cache hit. Therefore, it should be able to perform efficient query matching, i.e., determine if a new query is answerable by the union of cached views quickly even as the cache size grows.

- Efficient space management: Given that a dynamic data cache will contain a large number of cached queries or views, the cache needs to reduce the overhead of redundant storage by storing the large number of overlapping views in common-tables rather than each one separately. Moreover, space has to be optimized by evicting views that add low benefit while ensuring that any shared data is not removed.

- Consistency: Consistency has to be maintained efficiently in the presence of a large and varying number of views. The amount of work done to maintain the consistency of edge caches should scale well with the number of caches and their size.

2.2 E-commerce example

Consider for example an e-commerce site such as an on-line bookstore. Figure 1 shows a typical simplified schema for such a database. The example is derived from the schema used by the TPC-W benchmark which is used in our evaluation experiments. This on-line site allows keyword searches based on author names, titles, bestseller lists, related books purchased etc. The benchmark, like many real-world e-commerce applications, contains a variety of query types. Consider the searching the item table based on cost or suggested retail price:

Q_A: SELECT	i_avail, i_cost FROM $item$	WHERE $i\_cost < 5$
Q_B: SELECT	i_avail, i_cost FROM $item$	WHERE $i\_cost > 25$
...	...	...
Q_N: SELECT	i_srp, i_cost FROM $item$	WHERE i_srp BETWEEN 30 AND 65

Such queries should be cached and used to answer new queries whose constraints are contained in the above ranges. To avoid redundancy, the cache needs to share storage across the overlapping vertical and horizontal sections of the item table retrieved by these queries. Moreover, upon receiving a new query, the cache needs to verify if this new query is contained in the union of data sets retrieved by tens to hundreds of queries like the one above. This query matching operation must be efficient and scale well with the number of cached queries. Furthermore, in the event of a change to the base item table at the origin, the amount of work required to update the cached "views" should scale well with the number of queries cached.

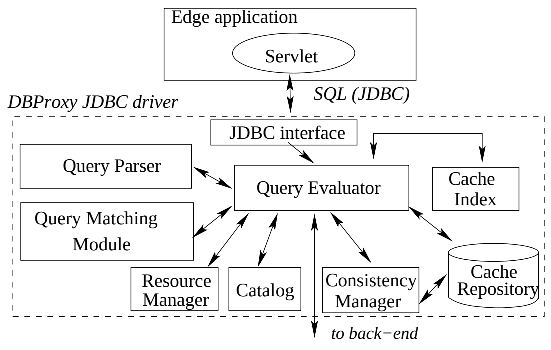

Figure 2. DBProxy key components.

3 DBProxy Architecture

The architecture of DBProxy assumes that the application components are running on the edge server (e.g., using the IBM Websphere Edge Server [14]). 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.

3.1 Design Overview

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*

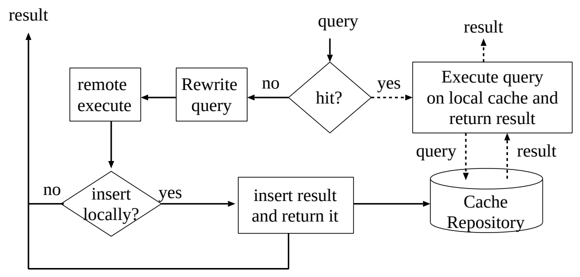

Figure 3. Cache hit and miss processing. The dashed arrows represent the hit path and the solid arrows represent the miss path. Query misses do not always insert data in the local cache.

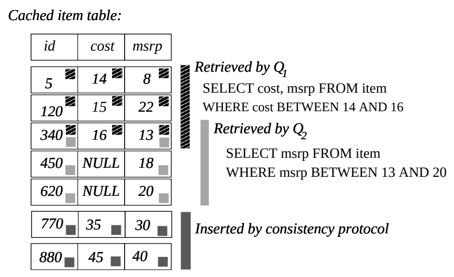

Figure 4. Local storage. The local *item* table entries after the queries Q_1 and Q_2 are inserted in the cache. The first three rows are fetched by Q_1 and the middle three are fetched by Q_2. Since Q_2 did not fetch the *i_cost* column, NULL values are inserted.

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 processing paths of a hit and a miss are shown in Figure 3.

DBProxy employs novel techniques for data storage, query matching, consistency maintenance, and cache replacement to achieve the desired efficiency. We overview these various aspects in the remainder of this section.

3.2 Cache Repository

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. 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 4 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 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. Observe from Figure 4 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, DBProxy's containment checker ensures that the cache does not return incorrect (e.g., false NULLs) or incomplete results.

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 Q_A through Q_N (of Section 2.2) 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_srp DOUBLE,                  PRIMARY KEY (i_id) )
```

Join queries. The results of a join query are stored in a single table, that is described by the table list in the join query as well as the join constraint. When the join queries operate on the same table list and have the same join condition, the results of joins are inserted in the same table. The name of the local table for the query is stored in the cache index entry.

3.3 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 in DBProxy are organized according to a multi-level index. The first level of the index is the database schema. The second level is the table or list of tables accessed by the query. Finally, the third level of the index contains the columns named in any of the query's clauses. Hash tables are used to allow quick search of each level.

A new query received by DBProxy 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.

Baseline matching algorithm. The result of query Q_B is contained in that of query Q_A if for all possible values of the items in the database, the WHERE predicate of the former logically implies that of the latter. That is: $Q_B.wherep \Rightarrow Q_A.wherep$. This is equivalent to the statement that $Q_B.wherep\ AND\ (NOT(Q_A.wherep))$ is *unsatisfiable*. Taking the particular queries Q_A and Q_B mentioned in Section 2.2 as an example, the query matching module will test the following expression for unsatisfiability: $i\_cost < 5\ AND\ NOT\ (i\_cost > 25)$. Since this expression is *satisfiable*, "$i\_cost = 4$" being one particular solution, then Q_B is *not contained* in Q_A. 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 unsatisfiability verification.

The baseline algorithm is based on the algorithm described in [24, 17] 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.

Template-based query matching. While the above baseline algorithm can handle complex and general predicates, it can induce significant overhead when the cache contains a large number of queries [4]. We observe, however, that most applications issue template-based queries whose selection predicates share the same structure and vary only in a few numeric or string constants. DBProxy exploits the template-based nature of application query streams to reduce containment checking overhead by an order of magnitude. Speficically, similar queries (that are instantiations of the same template) are aggregated in the cache, and their predicates merged together, using specialized data structures, which we call merged aggregate predicates, or MAPs. When a new query is received, it is first checked for containment in the MAP corresponding to the aggregation of previously cached and similar queries. MAPs are augmented with indexes to allow for fast search. Containment checking is thus converted into an index-based data search, achieving significant speedups [4].

3.4 Consistency

DBProxy ensures data consistency by subscribing to a stream of updates propagated by the origin server. Traditional materialized view approaches update cached views by re-executing the view definition against the change ("delta") in the base data. DBProxy requires a more efficient mechanism, however, because of the potentially large number of queries that it caches.

Since cached data is maintained as partially populated copies of back-end tables, changes committed to the base tables at the origin can be simply propagated "as is" to the cached versions, without the need to re-execute the queries. Updates, deletes and inserts (UDIs) to base tables are propagated and applied to the partially populated counterparts on the edge. Future queries that will execute over the cache will retrieve from these newly propagated changes any matching tuples. This solution presumes slowly changing data, typical of most web environments, and trades off potentially unnecessary data propagation for lowering the processing overhead of determining how the cached views should be updated. However, when a table is undergoing a heavy update workload, DBProxy can disable the local copy for a specified period.

Read-only queries issued by edge applications are satisfied from the cache whenever possible. Update transactions are always forwarded to the back-end database for execution, without first applying them to the local cache. Because DBProxy is designed for large scale deployment, its consistency protocol must be as loosely coupled from the origin as possible. Consequently, the onus of ensuring cache consistency should fall as much as possible on the edge caches—where resources scale more naturally. The back-end server should only be responsible for periodic update propagation, a task that can be offloaded to a separate process or machine.

3.4.1 Consistency guarantees

While many web applications can tolerate slightly stale data in the edge cache, they are nevertheless interested in reasonable consistency guarantees. For example, applications usually require that they observe the effects of their own updates on an immediate subsequent query. Since a query following an update can hit locally in a stale cache, updates not yet reflected in the cache would seem to have been lost, resulting in strange application behavior. In this section, we describe three specific consistency properties which are individually or collectively desirable for distributed web applications.

Let D_t be the set of committed tuples at the origin database at time t. D_t is therefore the state of the database at time t and reflects the set of transactions that committed at the origin server before and up to time t. At any point in time, the DBProxy edge data cache contains a subset of the tuples in the origin database. Let C_t denote the set of tuples present in the DBProxy edge data cache at time t. We define the following three properties, which are guaranteed by DBProxy at increasingly higher performance costs:

P_1**: Lag (δ) consistency with respect to the origin.** The state of the cache C_t at time t is considered lag-consistent with the origin database, if there exists a time lag δ such that the state of the cache corresponds to the state of the database δ time units ago: $C_t = D_{t-\delta}$. We interpret this equality to mean that the values of the tuples in the local cache equal those at the back-end at an earlier time, although the cache may have less tuples.

P_2**: Monotonic state transitions.** This implies that the state of the database exported by DBProxy moves only forward in time. That is, if an edge application observes a database state D_{t_1} and later observes a state D_{t_2}, then $t_2 \geq t_1$.

P_3**: Immediate visibility of updates.** This requires 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).

DBProxy relies on a data propagator, which captures all UDIs to the tables at the origin and forwards them to the edge caches. Data changes are propagated to the edges tagged by their transaction identifiers and applied to the edge cache in transaction commit order. The stream of incoming UDIs, reflecting the effects of transactions committed at the origin site, is applied to the locally cached tables. The challenge, in ensuring lag consistency in DBProxy arises because local tables can be updated by propagation messages *as well as* by query result inserts on a miss. DBProxy incorporates algorithms that ensure that lag consistency is maintained upon result inserts. Separate protocols are used to provide the additional consistency properties if specified by the application. The details of the algorithms are fully described in a technical report [3].

3.5 Cache replacement

The consistency protocol requires that all inserts, updates and deletes performed at the origin site to any cached table be propagated to the edge cache. Consequently, the cache can contain data that does not belong to any cached query. Furthermore, data that is not effectively used must be evicted to limit space overhead and optimize the usage of usually limited edge resources. To manage limited space resources on the edge, DBProxy relies on a background garbage-collection process which evicts unused data from the cache safely, while preserving data consistency. Specifically, the goal of cache replacement is to maximize the benefit of the cache for a limited amount of available space. In contrast to traditional replacement of files and memory pages, the underlying rows can be shared across multiple queries, complicating the task of query eviction. Cache replacement in DBProxy ensures that the following two properties are maintained.

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 cache replacement component of DBProxy consists of a replacement policy, that determines what to replace, and a replacement mechanism, that determines how to remove the data such that the above properties are maintained. The function of the cache replacement policy, is to determine the set of queries to replace from the cache. Given a space constraint, it tries to maximize the benefit of storing the query results locally for the space used, similar to the traditional knapsack problem. Determining the benefit of a query depends on multiple factors, namely: recency of access, frequency of access, miss cost versus hit cost, and the frequency of updates. This benefit is offset by the space usage of the query.

Cache replacement is triggered when the space usage of the local database reaches a high watermark (HWM) value. The replacement then begins until space is sufficiently freed and reaches below a low watermark (LWM) value. The replacement is not triggered on demand on a query miss when there is no space to insert more rows in the local tables or to create a new table. Replacement is a background process and can be expected to proceed concurrently with query servicing and update operations.

A deeper discussion of alternative mechanisms for replacement, their properties and analysis can be found in [5]. We biefly describe here one of the mechanisms, group replacement, which is simple to implement and adds no overhead on hit, miss, or update propagation. Based on the replacement policy, a set of "victim" queries are marked for deletion. To ensure that the non-overlapping rows, which belong to only "victim" queries are properly marked, we carry out the following steps. First, an "*accessed*" flag [1] associated with all cached rows is set to *false*. Then, non-victim queries are executed in the background. This background execution can occur over an extended period to not affect the performance of foreground queries. Whenever a query is executed, the "*accessed*" flag of any selected row is set to *true*. At the end of this execution cycle, any row whose flag remains *false* can be safely deleted. To prevent rows that have been inserted or updated by a cache miss or by the consistency manager during the garbage collection cycle, from getting deleted, we set their "*accessed*" flag to *true*. The following two claims hold for the group replacement algorithm:

Claim 1 *Group replacement guarantees that: (i) no tuple is deleted, if it belongs to the results of a query which remains in the cache index, and (ii) if a tuple is never accessed it will be eventually removed from the cache.*

Claim 2 *DBProxy, with consistency and group replacement enabled, guarantees that local executions will return correct and entire results.*

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 5 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 5). 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.

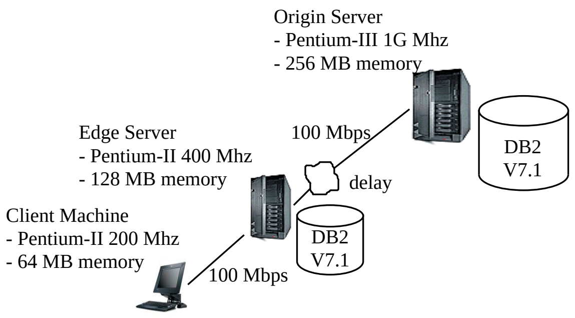

Figure 5. Evaluation environment. Three machines were used in the evaluation testbed based on the TPC-W benchmark. The client machine was used to run the workload generator, which applied a load equivalent to 8 emulated browsers. All machines ran Linux RedHat 7.1. Apache-Tomcat 3.2 was used as the app server.

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 [20]. 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

[1]This flag is implemented as a column defined during the creation of local tables.

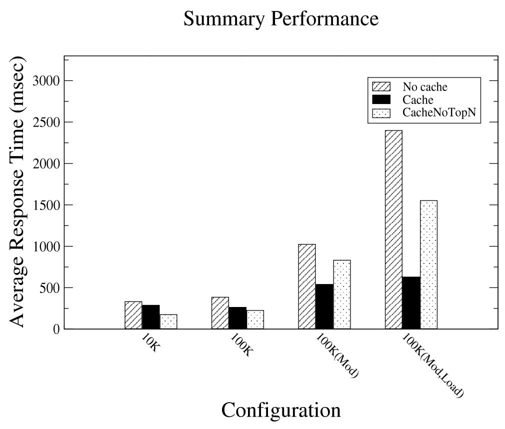

Figure 6. Summary of cache performance using the TPC-W benchmark. The four cases are: (i) Base TPC-W with a 10K item database, (ii) Base TPC-W with a 100K item database, (iii) Modified TPC-W with a 100K item database, (iv) Modified TPC-W with a 100K item database and a loaded back-end.

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.

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 6 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 6 (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 6) is discussed below.

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

	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.

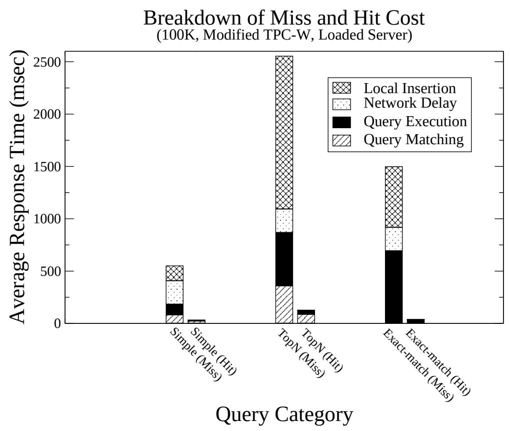

Figure 7. Breakdown of miss and hit cost under modified TPC-W with a loaded origin server.

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 6 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 7 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.

Effect of access skew. To investigate the effect of access skew on cache performance, which is more pronounced at the edge servers targeting a homogeneous user population, we modified the benchmark to exhibit a limited degree of skew in the search query distribution. Hot-sets are common in queries from real workloads, e.g., a skew based on the customers that are active or topics in the news recently. The standard benchmark selects the subject of a book that is queried by clients from a uniform distribution. We change this distribution to a Zipf distribution with parameter 0.9. Results showed that cache performance improves by 15% under skewed access while the no-cache performance improves by only 2%. Hit rates for the exact-match and Top-N query categories improve by about 3% each. This is expected since higher locality is likely to improve the performance of the caching configuration more than the non-caching configuration.

Template-based query containment checking. In the experiments reported in this section, the baseline query containment checking algorithm was used. To estimate the benefit of template-based containment checking, we performed the following two experiments. In the first, we execued the TPC-W benchmark against a cache employing a traditional containment checker, and in the second against a cache using template-based containment checking. The benchmark was executed for ten minutes to warm up the cache, then end-user response time was measured. We recorded an average response time improvement of 61% when template-based containment checking was used.

Overhead of replacement algorithm. To estimate the cost of our group replacement algorithm on the performance of the cache, we measured the average query response time for TPC-W — executing against the small database with 8 emulated browsers — with replacement and without replacement. In both cases, the same number of queries were actually used by the cache. In the without replacement case, victim queries were hidden from the cache index, but their data was not deleted from cached tables. For replacement, the background execution of the group replacement algorithm deleted the excess rows of victim queries. We found that the overhead of cleaning, in terms of slow-down of foreground queries, was low, around 10%, when 50% of the queries in the cache were evicted (i.e., half had to be executed by the group-replacement algorithm). Note that, in general, the execution overhead of group replacement grows with the number of queries that are to be maintained in the cache, and not with the number of queries that are to be evicted. Consequently, it is advantageous to start the group replacement algorithm, on reaching the LWM, when a large number of queries can be evicted such that a significant amount of space can be recovered.

5 Related work

DBProxy leverages a wealth of previous work in the areas of web caching, content distribution networks, database caching, materialized views and query processing. Caching of static Web data has been extensively studied [29, 12, 28, 26] and deployed commercially [2]. More recent work has focused on caching dynamic data [10, 19, 9] represented in the cache as HTML or XML. While these approaches store data separately in unstructured forms, increase redundant storage and rely on invalidation-based consistency, DBProxy minimizes redundant storage and provides lag-consistency with update propagation. Semantic caches have been proposed in client-server database systems [8]. The semantic cache proposal, however, only handles read-only databases. DBProxy is similar, in concept, to the work on semantic caching but it supports consistency and differs in the implementation approach. A simplified form of semantic caching targeting web workloads and using queries expressed through HTML forms has been recently proposed [19]. The caching of query results has also been proposed for specific applications, such as high-volume major event websites [10, 16]. 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.

The importance of database scalability over the web has prompted much industry interest in data caching and distribution. Database caching schemes, using full or partial table replication, have been being proposed [22, 27, 21]. These are powerful schemes but require administration and maintenance costs and incur a large space overhead. Furthermore, most of them require application server modifications and explicitly bundle the data caching and application distribution logic. DBProxy, on the other hand, is self-managing and does not require any application or database modification as it is bundled as a JDBC driver.

Much previous work also exists in the area of query containment and equivalence. The algorithms used for containment checking within DBProxy are based on extensions of previous work in this area [24, 17]. Earlier work on database caching investigated predicate-based schemes and views to answer queries [11, 15, 18, 25]. Previous work in the area of materialized view routing (i.e., answering queries by rewriting using materialized views) also describes techniques for matching and containment [7, 18, 6, 30, 23, 13, 1]. DBProxy differs from the materialized view approach in relying only on the query stream to decide on cache population and replacement. DBProxy stores views in common tables and dynamically decides the views that are to be cached or replaced. Consistency management in most materialized view approaches does not scale to a large number of views, while the update propagations on the common-table used by DBProxy simplifies scalability.

6 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 includes a scalable template-based query containment checker, and maintains data consistency efficiently while guaranteeing several useful consistency properties. 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.

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, S. Park, R. Tewari, and S. Padmanabhan. Scalable template-based query containment checking for web semantic caches. In *ICDE Conference*, 2003.

[5] K. Amiri, R. Tewari, S. Park, and S. Padmanabhan. On space management in a dynamic edge data cache. In *WebDB Conference (Informal Proceedings)*, 2002.

[6] 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.

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

[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] A. Datta, K. Dutta, H. M. Thomas, D. E. VanderMeer, K. Ramamritham, and D. Fishman. A comparative study of alternative middle tier caching solutions to support dynamic web content acceleration. In *VLDB Conference*, 2001.

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

[11] P. Deshpande, K. Pamasamy, A. Shukla, and J. F. Naughton. Caching multi-dimensional queries using chunks. In *SIGMOD Conference*, pages 259–270, 1998.

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

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

[14] IBM Corporation. Websphere Edge Server. http://www-4.ibm.com/software/webservers/edgeserver/.

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

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

[17] 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.

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

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

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

[21] C. Mohan. DBCache (in VLDB 2001 Tutorial). http://www.almaden.ibm.com/u/mohan/Caching_VLDB2001.pdf.

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

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

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

[25] T. K. Sellis. Intelligent caching and indexing techniques for relational database systems. *Information Systems*, 13(2):175–185, 1988.

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

[27] Timesten. Front-Tier Data Cache. http://www.timesten.com/products/fronttier/ftwhitepaper.html.

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

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

[30] 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.