On space management in a dynamic edge data cache

Khalil Amiri, Renu Tewari, Sanghyun Park, and Sriram Padmanabhan
IBM Thomas J. Watson Research Center
{*amirik,tewarir,sanghyun,srp*}@*us.ibm.com*

Abstract

Emerging web applications are increasingly serving dynamic content generated by querying back-end database servers. The serving of dynamic content can be scaled by offloading applications and caching data at edge servers. In recent work, we proposed a persistent and self-managing edge-of-network data cache that is dynamically populated based on the application query stream and stored locally in a persistent database. In this paper, we discuss the challenges of cache maintenance in such a dynamic environment, focusing on replacement issues in the presence of update-based consistency protocols. We describe how our experience with a real prototype, built using a JDBC driver and DB2, highlights the limitations of traditional approaches. Furthermore, we propose a cache replacement mechanism and policy that address these challenges.

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 [11, 9]. Consistency and cache space management issues, however, can easily limit 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) [1]) acts as an application server proxy by offloading application components (e.g., JSPs, servlets, EJBeans) to the edge [7]. Database accesses by these edge application components, however, are still performed across the wide area network. To accelerate edge applications by eliminating wide-area network transfers, we have recently proposed and implemented DBProxy, a database cache that dynamically and adaptively stores data at the edge [2]. Since the edge server is limited in resources of space and processing power, we rely on efficient cache replacement policies and mechanisms to keep only the most beneficial data in the local database cache. Cache replacement for physical page-based caches, static files, and read-only semantic caches has been thoroughly studied in the literature. Yet, it introduces new challenges in the context of a persistent edge cache containing a large number of changing and overlapping "materialized views" of previous query results. This is because locally stored data can be shared by multiple cached "views", and can be updated by a cache consistency protocol.

In this paper, we review the design of the dynamic edge data cache in Section 2, discuss the limitations of traditional approaches to cache replacement and offer alternative solutions in Section 3, briefly review related work in Section 4, and summarize the paper in Section 5.

2 Background

We assume in this discussion that application components (e.g., servlets) are running on the edge server (e.g., using the IBM WebSphere Edge Server [7]). 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

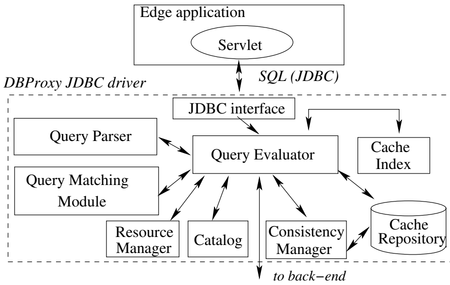

Figure 1: DBProxy key components. The query evaluator intercepts queries and parses them. The cache index is invoked to identify previously cached queries that operated on the same table(s) and column(s). A query matching module establishes whether the new query's results are contained in the union of the data retrieved by previously cached queries. A local database is used to store the cached data.

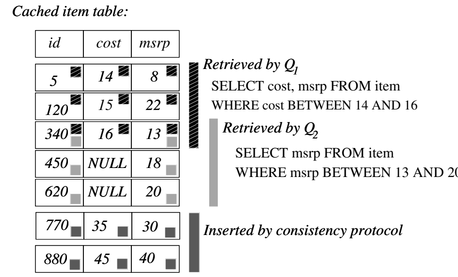

Figure 2: Local storage: The local *item* table after the queries Q_1 and Q_2 are inserted in the cache. The first three rows are fetched by Q_1 and the middle three are fetched by Q_2. Since Q_2 did not fetch the *cost* column, NULL values are inserted. The bottom two rows were not added to the table as a part of query result insertion, but by the update propagation protocol which reflects UDIs performed on the origin table.

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.

2.1 DBProxy overview

We designed and implemented an edge data cache, called DBProxy, as a JDBC driver which transparently intercepts the SQL calls issued by application components executed on the edge and determines if they can be satisfied from the local cache (shown in Figure 1). DBProxy is designed to be deployed on edge servers which could number in the tens to hundreds. Consequently, it needs to be self-managing to limit the administrative overheads of a large scale deployment. Furthermore, each edge server could serve a different population (i.e., may observe a different access pattern) and could contain different resource constraints, making manual optimizations impractical. To make DBProxy as self-managing as possible, while leveraging the performance capabilities of mature database management systems, we chose to design DBProxy to be: (i) persistent, so that results are cached across instantiations and crashes of the edge server; (ii) DBMS-based, utilizing a stand-alone database for storage to allow for the efficient execution of complex local queries; (iii) space-efficient, storing query results in common tables to avoid redundancy whenever possible; (iv) dynamically populated, populating the cache based on the application query stream without the need for pre-defined administrator views; and (v) dynamically pruned, adjusting the set of cached queries based on available space and relative benefits of cached queries.

2.2 Common store

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 *shared* 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 at the origin server. 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, as discussed below. When a query is worth caching, a local result table is created (if one does not already exist) 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. For example, Figure 2 shows a local table cached at the edge. The local *item* table is created just before inserting the three rows retrieved by query Q_1 with the primary key column (*id*) and the two columns requested by the query (*cost* and *msrp*). All queries are rewritten to retrieve the primary key so that identical rows in the cached table are identified. Later, to insert the three rows retrieved by Q_2, the table is *altered* if necessary to add any new columns not already created. Next, new rows fetched by Q_2 are inserted ($id = 450, 620$) and existing rows ($id = 340$) are updated. Note also that since Q_2 did not select the *cost* column, a NULL value is inserted for that column. The query matching module ensures that the queries executed against the cache do not return any of the "fake" NULL values in local tables. To handle a large and varying set of cached views, the query

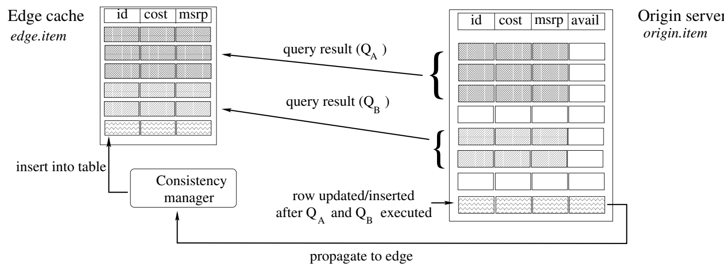

Figure 3: Update propagation in DBProxy. The figure shows a table in the origin database server (*item*), and its partially populated copy in the edge cache. Query results over the table are inserted in the cache. When a row is inserted, updated, or deleted in the origin, the change is reflected in the cache without evaluating whether the row matches any of the cached predicates.

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 of schemas, tables and clauses for this purpose.

2.3 Update propagation

Update transactions in DBProxy are routed to the back-end database without applying them first to the edge cache, and are therefore guaranteed transactional semantics. Read-only queries in DBProxy are satisfied from the cache if the data is locally available. Data consistency is ensured by subscribing to a stream of updates propagated by the back-end server. Specifically, the cache guarantees δ-consistency, that is, the view exported by the cache to a query corresponds to a consistent past database state that is within δ time units. Other important properties like *view monotonicity* and *immediate update visibility* are also guaranteed and the protocols that ensure them are described in [2]. It suffices to note here that the edge data cache can be updated along two paths as shown in Figure 3: the first is through the insertion of a new query result upon a query miss, and the second is through the stream of refresh messages propagated by the origin server. Refresh messages contain updates, deletes and inserts which are applied to the edge server's partially populated cached copies of the origin tables, without first checking if the tuples propagated to the cache satisfy the predicate of any cached query. We found that such a check can induce significant overhead due to the potentially large number of cached queries and the complexity of query predicates. This is illustrated in Figure 2, where the two bottom rows in the table are inserted after being propagated from the origin because they were written by an update transaction. No check is made as to whether these rows match the predicates of queries Q_1 or Q_2. Excess rows inserted in the table must later be "garbage collected" by the cache replacement process.

3 Cache replacement

To limit space overhead and optimize the usage of usually limited edge resources, the cache space has to be managed such that unused data gets evicted 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. 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.

3.1 Replacement policy

The function of the cache replacement policy is to determine the set of queries to replace from the cache. Cache replacement policies have been extensively studied in different areas, from virtual memory and file buffer caching to, more recently, web caching. The policy we use is a combination of previous approaches and is most suited for edge data caching where the query processing costs and sizes are different. Given a space constraint, our policy tries to maximize the benefit of storing the query results locally for the cost of the space used, similar to the traditional knapsack problem of optimizing the cost-benefit. Determining the benefit of a query depends on multiple factors, namely: i) recency of access (that is the factor used in the LRU policy); ii) frequency of access (that is used in the LFU policy and is useful for skewed access patterns); iii) miss cost versus hit cost (especially since the query execution costs are high and variable); and iv) the frequency of updates (since an update adds to the overhead of caching). We use an estimated access frequency measure that balances the recency of access (using the last access time) and the frequency of access. Combining the above set of factors, the benefit of maintaining an object in the cache is proportional to the estimated access frequency and the differential miss processing cost, and is inversely proportional to the update frequency. The update frequency term is maintained at the level of tables. The benefit of query is then offset by its space overhead.

Borrowing from the greedy heuristic used in the knapsack solution for replacing queries with variable data-set sizes, we order the queries based on the ratio of $\frac{benefit}{space\_used}$. The objects with the smallest ratio are then marked for removal. While the parameters in the benefit computation can be estimated by maintaining various statistics, the space overhead of a query, as we discuss in Section 3.4, is more challenging to compute accurately because queries can have multiple overlapping tuples, i.e., the same row can "belong" to many cached queries.

3.2 Replacement mechanisms: Challenges

The replacement mechanism is the process by which the tuples belonging to queries marked for eviction are removed from the local database tables. This process is complicated by several factors including: the shared storage strategy, the containment checking overhead and the consistency policy used.

3.2.1 Shared store

Recall that the tuples brought in by different queries are stored in common tables as far as possible. In contrast to traditional replacement of files and memory pages, the underlying tuples can be shared across multiple queries in the cache. Consider the queries Q_1 and Q_2 of Figure 2, where the result set of query Q_1 contains rows with *id* $\{5{,}120{,}340\}$ and the result set of query Q_2 contains rows with *id* $\{340, 450, 620\}$. If query Q_2 is to be replaced while Q_1 remains in the cache, then only the rows with *id* $\{450{,}620\}$ can be removed. The space gained will be that of the two rows deleted and not the size of the entire query which was 3 rows. In general the replacement mechanism should support the following property.

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.*

Read-only lazy replacement: If we assume that the back-end database is read-only (i.e., there are no updates), a simple "counter" based mechanism that counts the number of references to a given tuple, can be used to evict queries that share common tuples in the local table. When a new tuple is inserted into the local table, a reference counter for that tuple is incremented. When a query is marked for deletion, the reference counter for a corresponding tuple is decremented. Eventually, a tuple is "lazily" deleted when its reference counter becomes zero, i.e., there are no queries in the cache that access that tuple. The assumption of a read-only database is obviously unrealistic in practice. Next, we relax this assumption and discuss the implications of updates for the replacement policy and mechanism.

3.2.2 Containment checking

When a new query is received by the cache, the query matching module (shown in Figure 1) verifies whether the new query's predicate is more restrictive than (i.e., is contained in) that of a cached query's predicate. Furthermore, the query matching module ensures that the cached query has retrieved all the columns required to evaluate the new query over the local cache. The query matching module can check if a new predicate is contained in the union of several cached predicates. In case of a miss, the query is cached and its predicate and other clauses are added to the cache index. A new query can potentially overlap with a large number of cached queries, but no attempt is made to compute or maintain information about this overlap. One approach to eliminate the problem of common tuples between queries is to partition the tuples into *non-overlapping sets* and index each set separately in the cache [3]. This approach, though theoretically possible, adds to the overhead of the containment checker, and we found that it was not practical when the number of queries was large. Splitting the queries increases the terms in the predicate clauses where for every pair of queries Q_1 and Q_2, their "intersection" set $Q_1 \cap Q_2$ is indexed as Q_1 *AND* Q_2. Since each query can intersect with multiple queries, the complexity of the clause in the number of terms, grows linearly with the number overlapping queries. The number of non-overlapping sets, on the other hand, grows exponentially with the number of queries. The containment checker is optimized to quickly find a matching query by using an index hierarchy starting with the table names and the column names used by the different clauses. This quickly narrows the number of queries to check for a full containment. For overlap checking, on the other hand, the set of possible queries is much larger. Apart from the performance issue, another problem with assuming non-overlapping query sets arises because of consistency maintenance when the back-end database is not read-only. Whenever tuples in the cache are updated, their membership in the non-overlapping sets may change.

3.2.3 Consistency management

The consistency manager, as described earlier, propagates *all* changes (UDIs) to the back-end tables that have been cached at the edge server. The update of a tuple in an edge cache may make it match a larger (or smaller) number of cached query predicates. It is possible that an updated tuple ceases to be useful, because it no longer matches any cached predicate. The replacement policy selects queries for removal, and the replacement mechanism must remove the underlying tuples that belong to these queries' results but do not belong to the results of any queries that are to remain in the cache. Moreover, the replacement mechanism should also evict any rows inserted by the consistency manager that do not match any query's predicate. For example, the rows in Figure 2 with *id* $\{770{,}880\}$, that were inserted by the consistency manager but which did not belong to either query's result set, should be evicted. In general, the following property should hold.

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 following observation affects the choice of the replacement mechanism and the containment checking strategy:

Observation 1 *The set of tuples forming the result of a cached a query can change dynamically due to possible UDI operations at the back-end that are propagated to the local database.*

This observation says that the membership of cached tuples in query results can change dynamically, which implies that maintaining explicit information about this membership or its overlap is not desirable. In particular, the lazy removal approach discussed above, which uses a reference counter will not have the correct counter value if the tuples are added and updated by the consistency manager. Creating non-overlapping sets becomes non-trivial when the consistency manager can add or update tuples. On a UDI, the tuple belonging to a query can change; since determining the reverse mapping, whether a tuple belongs to a query's result set, requires the re-evaluation of the query predicate over the new tuple, we cannot easily maintain accurate information about the membership of a tuple in one of the non-overlapping sets.

3.3 Proposed mechanism: Group replacement

We propose a replacement mechanism that proceeds as a background process concurrently with query hit and miss processing. The idea is to perform pro-active cleaning such that the cache space usage never exceeds a maximum threshold. Thus replacement, in our architecture, is not triggered when there is no space to insert the miss results. While a thorough discussion of alternative mechanisms for replacement is the subject of ongoing work, we briefly describe here one particular and promising mechanism, called *group replacement*, which is simple to implement and adds no overhead on hit, miss, or update propagation. The basic idea of group replacement is to execute the queries that are to *remain* in the cache against all the locally cached rows, "marking" any rows that match any of the queries' predicates. A control column, used as "marked" flag, is created in each cached table. This flag is first reset at the beginning of the group replacement cycle, and is set whenever the row is accessed by the cached query. Once all cached queries are executed, any unmarked rows can be safely deleted. Group replacement is used in conjunction with a replacement policy to determine the set of "victim" queries to be deleted. Note that the overhead of group replacement is linear in the number of queries that remain in the cache.

3.4 Determining query size

The replacement mechanisms rely on the replacement policy to determine the query or set of queries to replace. As described earlier, one factor used by the replacement policy, when handling varying sized data-sets, is the size of the query. However, due to overlapping tuples between queries, determining the number of tuples that will get replaced is not straightforward. In particular, the following observation holds.

Observation 2 *The actual number of tuples of a query that can be garbage collected depends on the set of queries that* remain *in the cache.*

The above observation highlights the complexity of using the size as a factor to determine the set of queries to be replaced. It results in a circular dependency—the *replaceable* size depends on what remains in the cache and what remains in the cache depends on the ordering by the ratio of benefit to size. We assume, initially, that the total size of the query (as determined by the number of tuples hit in the last access) is a good estimator of the replaceable size of the query. Another approach is to evaluate each query and determine its count of "exclusive" (non-overlapping) tuples with respect to all the other queries currently in the cache. This is a high overhead operation, requiring a counter to be maintained for each tuple to represent the number of queries that access that tuple. Using this counter, a re-execution of the query can be used to determine the set of "exclusive" tuples that the query accesses, which becomes a measure of the replaceable size. These two approaches—*total size and exclusive size*— form two ends of the spectrum of heuristics used to determine the replaceable size of a query. An intermediate approach is to order the queries by benefit (without using size as a factor) and then re-executing each query in descending order of benefit while counting the new tuples accessed by a query to represent its replaceable size. We are evaluating this and other approaches in our ongoing work.

4 Related work

Client-server database systems have also addressed replacement issues in page-based and tuple-based client caches [5, 6]. Replacement in a cache indexed by semantic units such as views or query results introduces different challenges, however. Semantic caches have been proposed in client-server database systems [3] but that work addressed only read-only caching and used a different storage implementation. Predicate-based caches [8] addressed similar issues, but opted for a different approach to consistency maintenance. Tuples propagated as a result of UDIs performed at the origin are matched with all cached query predicates to determine if they should be inserted in the cache. A simplified form of semantic caching targeting web workloads and using queries expressed through HTML forms has been recently proposed [11], but this work did not address consistency or replacement. The caching of query results has also been proposed for specific applications, such as high-volume major event websites [4, 10]. 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.

5 Summary

Dynamic caching of data on edge servers based on the application query stream promises to be an adaptive caching solution with a low administrative overhead. We have implemented a prototype of a caching system which maintains previous query results in shared tables whenever possible. In this paper, we discuss the challenges of cache maintenance in such a dynamic environment, focusing on replacement issues in the presence of consistency guarantees. We describe a consistency protocol which propagates UDIs performed on the origin database over cached tables to the edge cache without first checking whether the new tuples match the cached query predicates. A "group replacement" mechanism operates in the background and removes all excess tuples from the cache. We discuss the challenges of devising a cache replacement policy for such a dynamic environment and propose insights into addressing these challenges.

References

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

[2] K. Amiri, S. Park, R. Tewari, and S. Padmanabhan. DBProxy: A self-managing edge-of-network data cache. IBM Research Report, 2002.

[3] 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.

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

[5] D. DeWitt, P. Futtersack, D. Maier, and F. Velez. A study of three alternative workstation-server architectures for object-oriented database systems. In *VLDB Conference*, pages 107–121, 1990.

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

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

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

[9] A. Labrinidis and N. Roussopoulos. WebView Materialization. In *SIGMOD Conference*, pages 367–378, 2000.

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

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