Exploiting templates to scale consistency maintenance in edge database caches

Khalil Amiri
IBM Canada
Toronto, ON
kamiri@ca.ibm.com

Sara Sprenkle[*]
Duke University
Durham, NC
sprenkle@cs.duke.edu

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

Semantic database caching is a self-managing approach to dynamic materialization of "semantic" slices of back-end databases on servers at the edge of the network. It can be used to enhance the performance of distributed Web servers, information integration applications, and Web applications offloaded to edge servers. Such semantic caches often rely on update propagation protocols to maintain consistency with the back-end database system. However, the scalability of such update propagation protocols continues to be a major challenge. In this paper, we focus on the scalability of update propagation from back-end databases to the edge server caches. In particular, we propose a publish-subscribe like scheme for aggregating cache subscriptions at the back-end site to enhance the scalability of the filtering step required to route updates to the target caches. Our proposal exploits the template-rich nature of Web applications and promises significantly better scalability. In this paper, we describe our approach, discuss the tradeoffs that arise in its implementation, and estimate its scalability compared to naive update propagation schemes.

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

1 Introduction

The performance and scalability of Web applications continues to be a critical requirement for content providers. Traditionally, static caching of HTML pages on edge servers has been used to help meet this requirement. However, with a growing fraction of the content becoming dynamic and requiring access to the back-end database, static caching is by-passed as all the dynamically generated pages are marked uncachable by the server.

Dynamic data is typically served using a 3-tiered web serving 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. In more recent architectures, the edge server (which includes client-side proxies, server-side reverse proxies, or caches within a content distribution network(CDN) [2]) acts as an application server proxy by offloading application components (e.g., JSPs, servlets, EJBeans) to the edge [12, 7]. Database accesses by these edge application components, however, are still retrieved from the back-end server over 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 structured data at the edge [4]. The cache in this scenario is a persistent edge cache containing a large number of changing and overlapping "materialized views" stored in common tables.

The scalability of such a scheme depends on how efficiently we can maintain consistency with the back-end database without undue processing at the back-end or network bandwidth overheads. Consistency in semantic caches has been traditionally achieved through two approaches. The first is timeout-based. Data in the cache is expired after a specified timeout period, regardless of whether it has been updated at the back-end or not. The second approach is update-based consistency which relies on propagating the relevant changed data to the caches where it is locally applied.

Timeout-based consistency suffers from the disadvantage of increased latency and network message overheads as the cache needs to validate the data with the back-end or fetch any modified data. Update-based consistency propagates all the new data to the cache when any change happens, but suffers from serious scalability limitations both as the number of caches grow, and as the size of an individual cache, in terms of the number of cached views, increases. The scalability problem arises from the back-end overhead of "figuring out", when a row changes, which of the target caches it must be forwarded to.

One approach that was used initially in DBProxy was to propagate all changes at the back-end and apply them to the local cache. This increases the network bandwidth overhead and increases the size of the local cache. In this paper, we propose *template-based filtering* that efficiently aggregates cache subscriptions at the back-end by exploiting the similarity of "views" in edge query caches. Similar aggregations have been proposed for event matching (against client subscriptions) in publish subscribe systems.

We discuss the specifics of our approach, describe how we handle dynamic changes in cache subscription, and highlight the challenges and trade-offs that arise in this space.

The rest of the paper is organized as follows. We review the design of our prototype dynamic edge data cache in Section 2, and describe its consistency maintenance framework in Section 3. We review naive filtering schemes in Section 4 and describe our approach in Section 5. We discuss related work in Section 6 and summarize the paper in Section 7.

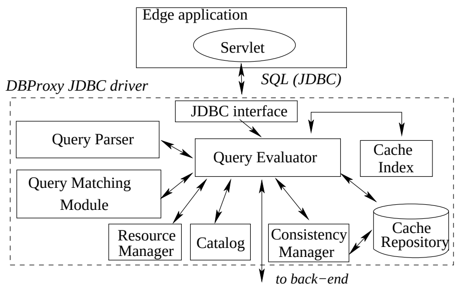

Figure 1. DBProxy key components. The query evaluator intercepts queries and parses them. The cache index is invoked to identify previously cached queries that operated on the same table(s) and columns. A query matching module establishes whether the new query's results are contained in the union of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

2 Semantic caching over the Web

2.1 DBProxy overview

We designed and implemented an edge data cache, called DBProxy [4], as a JDBC driver which transparently intercepts the SQL calls issued by application components (e.g., servlets) executed on the edge and determines if they can be satisfied from the local cache (shown in Figure 1). 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 eliminate redundancy using common tables and to allow for the efficient execution of complex local queries; (iii) dynamically populated, populating the cache based on the application query stream without the need for pre-defined administrator views; and (iv) dynamically pruned, adjusting the set of cached queries based on available space and relative benefits of cached queries.

2.2 Common Local 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 *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 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 it 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 an example 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 (i_id) and the two columns requested by the query (i_cost and i_srp). All queries are rewritten to retrieve the primary key so that identical rows in the cached table are identified. Later, and 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 ($i\_id = 450, 620$) and existing rows ($i\_id = 340$) are updated. Note also that since Q_2 did not select the i_cost column, a NULL value is inserted for that column.

Cached queries in DBProxy are organized according to a multi-level index of schemas, tables and clauses for efficient matching.

3 Consistency Management

Read-only queries received by DBProxy are satisfied from the local cache, whenever possible. Updates are always passed to the back-end database directly, bypassing the cache. The effects of updates trickle back to the edge cache, through a push or a pull approach.

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. Specifically we assume that three consistency properties hold. First, the cache guarantees *lag consistency* with respect to the origin. This means 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. Second, DBProxy exhibits *monotonic state transitions*, i.e, it does not allow an application to see a relatively current database state, then see a previous state corresponding to a point earlier in time. Finally, DBProxy supports *immediate visibility of local updates* where the result of a later read by an application will show the effect of the local update it had committed earlier.

To achieve these properties, DBProxy relies on protocols that force the cache to pull updates from the server on certain events, and also may decide to bypass the cache in others. The details of the consistency protocols and criteria are not the focus of this paper, but are more fully described in [3]. Suffice it to say, here, that regardless of the particular criteria, changed data must be efficiently propagated from the back-end database to the edge caches. In this section, we discuss two alternative approaches to update propagation, and describe a server-side filtering architecture which can be used to propagate changes in back-end tables only

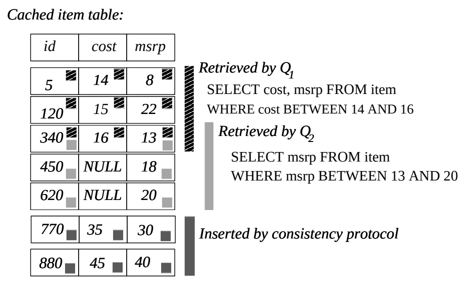

Figure 2. Local storage. The local *item* table entries after the queries Q_1 and Q_2 are inserted in the cache. The first three rows are fetched by Q_1 and the last three are fetched by Q_2. Since Q_2 did not fetch the *i_cost* column, NULL values are inserted. The bottom two rows were not added to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

to the caches that require them.

3.1 Update propagation approaches

To maintain consistency, DBProxy relies on a data propagator, which captures all UDIs (updates, deletes, inserts) to the tables at the origin and forwards them to the edge caches either in their entirety or after filtering.
Zero-filtering propagation. This approach, which we initially implemented in DBProxy, propagates all changes to the edge caches, regardless of whether or not they match cached predicates. This places no filtering load on the server. Data changes are propagated to the edges tagged by their transaction identifiers and applied to the edge cache in transaction commit order. Since cached data is maintained as partially populated copies of back-end tables, changes committed to the base tables at the origin can be applied "as is" to the cached versions, without the need to re-execute the queries. Future queries that will execute over the cache will retrieve from these newly propagated changes any matching tuples. This solution presumes slowly changing data with few updates which is typical of some web environments.
Server-side filtering. The problem with propagation based consistency with zero filtering is that all changes to the back-end are propagated regardless of whether they are required by the cache or not. This not only wastes network bandwidth but also increases the size of the local cache database. Periodic garbage collection is required to clean the unaccessed tuples. An alternative approach is to filter the tuples that change in the back-end database and forward to each cache only the "relevant" ones. This filtering step can place, however, high load on the back-end site as the number of caches and the number of views per cache increase. The rest of the paper will describe filtering in more detail, and suggest an approach to make it more scalable.

4 Basic filtering

Notation. Before we discuss the details of filtering, we first define a few notational conventions to simplify the rest of the discussion. For each query in the cache we have a subscription at the back-end. We express a subscription S as a 3-tuple $S = (T, A, P)$, where T is the table name(s) accessed by the query. In case of a join, T may contain two or more tables. A is the set of attributes or columns projected by the query, and P is the search predicate. P is assumed to be in AND-OR normal form. When expressed in its AND-OR normal form, we denote by C_i the i^{th} *conjunct* in that expression. Each conjunct contains several *predicate terms* (e.g., cost < 15) ANDed together. These predicate terms are atomic conditions, such as equality or inequality predicates over columns.

Let N be the number of edge caches. For cache i, the set of subscriptions are denoted by S_i, and the number of subscriptions are therefore equal to $|S_i|$. The j^{th} subscription of cache i is, therefore, $S_{ij} = (T_{ij}, A_{ij}, P_{ij})$.
Overview of Filtering. In this scheme, caches "subscribe" to a number of "logical" update streams. In particular, each cached view or query corresponds to one subscription. A filtering server at the back-end site manages subscriptions for the edge caches. Subscriptions are dynamic, that is they change as new views are added or evicted.

The filtering server, therefore, has to test each newly changed tuple (or row in a table) against all the $\sum_{i=1..N} |S_i|$ subscriptions. This results in a linear search overhead as the number of caches N increases, and as the number of subscriptions $|S_i|$ per cache increase.

Precisely, for each tuple t_k let us assume that t_k^{old} is the old value and t_k^{new} is the value after a change (i.e., a UDI). In case of a newly inserted tuple t_k^{old} is NULL, similarly, when a tuple is deleted t_k^{new} is NULL. Assuming there is a single cached query denoted by S_{ij} with predicate P_{ij}, then the filtering algorithm decides to route tuple t_k to the target cache if either of the following two conditions hold:

- $t_k^{new} \in P_{ij}$
- $t_k^{old} \in P_{ij}$ AND $t_k^{new} \notin P_{ij}$

In the first case the tuple is inserted or updated in the cache. In the second case the tuple is either deleted or updated. The pseudo-code of the filtering algorithm is shown below.

```
filter (TUPLE Told, TUPLE Tnew, CACHE i)
begin
    for every SUBSCRIPTION Sij from CACHE i {
       for every CONJUNCT C in Sij {
         if( C.matches(Tnew) )
            return TUPLE_MATCH;
            else if ( C.matches(Told) )
               return TUPLE_MATCH;
         }
      }
      return TUPLE_NO_MATCH;
end
```

The filtering procedure above is invoked once for each cache, for each tuple updated by a transaction in the back-end server. For each subscription $S_{ij} = C_1 \vee C_2 \vee ... \vee C_m$, the tuple is tested against each of the C_i's, and is forwarded to the edge cache if it matches any of them. Note that each of the C_i's is expressed as a bunch of atomic attribute tests (of the form $attr$ $\{=,<,...\}$ val) ANDed together.

The complexity of the matching function grows with the size or complexity of the subscription. If the average number of conjuncts per subscription is m and the number of terms per conjunct is t, and if the average cache size is S, then assuming that there are N caches, the total complexity can be expressed as $O(N \times S \times m \times t)$. To simplify the analysis, when estimating filtering cost, we will assume, for the rest of the discussion, that each subscription has a single conjunct with t terms, in which the filtering complexity can be summarized as $O(N \times S \times t)$.

5 Template-based filtering

In order to improve the scalability of the basic filtering approach we propose template-based filtering that exploits the similarity among the subscriptions corresponding to the cached queries. Such similarities exist because queries are generated by applications which often define parametrized query templates which are instantiated into actual queries at run-time by binding the variable parameters.

Templates. In Java-based Web applications, two query programming styles are observed: i) explicitly declared templates where queries are pre-declared as *prepared statements* and where parameters are set through explicit calls, or ii) undeclared templates where queries are composed by the *string concatenation* of a fixed part and a variable part. Such templates are often seen in the servlet programs running at the Web server which process the inputs from the front-end interface consisting of web-page forms. If the template is not explicitly declared, our caching driver has to infer implicitly that some submitted queries follow the same template.

When referring to a group of similar query predicates instantiated by the same template, we call the part of the predicate that changes across the group as the *variant part*, to distinguish it from the *fixed part*. For the rest of this discussion, we will focus on a single conjunct in the query's predicate expression. Consider the two predicate expressions:

$$P_1 = (cost < 15 \wedge msrp < 8 \wedge num\_reviews = 5)$$
$$P_2 = (cost < 15 \wedge msrp < 8 \wedge num\_reviews = 10)$$

A comparison of these two predicate expressions suggests that they are likely two instantiations of a template P_t of the form: $P_t = (cost < 15 \wedge msrp < 8 \wedge num\_reviews = ?)$ In this example, the predicates P_1 and P_2 contain a single conjunct $(cost < 15 \wedge msrp < 8 \wedge num\_reviews = ?)$. Within the conjunct, $(cost < 15 \wedge msrp < 8)$ is the fixed part and $(num\_reviews = ?)$ is the variant part. The atomic test $(cost < 15)$ is called a predicate term.

Templates in real applications. We looked at the number of templates used in two e-commerce benchmarks, TPC-W, which emulates an on-line bookstore and ECD-W, which is much more complex and emulates a customizable on-line shopping mall. The number of templates in these realistic full-fledged Java applications were 18 and 81, respectively. Furthermore, the percentage of queries generated by the top-5 templates where 79% an 68% respectively.

We detect templates on-line by comparing the structure of search predicates in a query stream. Our template detection algorithms are described in [5].

5.1 Template-based filtering: Single cache case

The main intuition behind our approach is that the fixed part of the predicate should be tested only once for all the query subscriptions generated from the same template. This is similar to the motivation behind event matching schemes in publish-subscribe systems. However, because of the prevalence of templates in Web applications, we suggest using specialized data structures, to aggregate the parameter instantiations of the variant part of the predicate. This allows for fast matching of a new tuple against the collection of the variant parts. We call these indexed aggregate data structures, Merged Aggregated Predicates (MAPs).

In the above example, assume that many subscriptions are received which are all generated by the same template. In that case, the MAP can be a hash table aggregating the variant integer terms that appear in the $num\_reviews = ?$ test. The filtering test can therefore be reduced to testing the *num_reviews* attribute of the tuple against existence in the hash table. The tests against the *cost* and *msrp* attributes would be performed only once. The complexity of testing a tuple against a hash-table MAP is constant. If we denote by K the number of templates, which is usually much smaller than the number of subscriptions, $K << S$, the total complexity is $O(K)$, which is independent of cache size, unlike the cost of naive filtering, $O(S \times t)$. Of course, not all MAPs are hash tables, and therefore the complexity of looking up a MAP is not always constant, but is at worst logarithmic in cache size, as we show below.

Example MAPs. We describe through a few examples, the various possible MAPs. Consider the following template: $P_t = (cost <? \wedge msrp < 8 \wedge num\_reviews = 5)$

In this case, the subscriptions can be aggregated into a single value corresponding to the maximum upper bound parameter received for the cost attribute (e.g., for cost $<$ 10, cost $<$ 20, cost $<$ 50, we only store the value 50). Consider, alternatively, a slightly more complex expression: $P_t = (cost\ BETWEEN\ ?\ AND\ ? \wedge msrp < 8 \wedge num\_reviews = 5)$

Our approach in this case is to merge successive instantiations of a BETWEEN predicate into an interval set, where all overlapping intervals are merged together. For example, the intervals $cost\ BETWEEN\ 10\ AND\ 20$, $cost\ BETWEEN\ 15\ AND\ 25$ will be merged and represented as $[10, 25]$. Further, the intervals are sorted to enable efficient binary searching. Thus intervals [10, 20], [45, 60], [25, 35] will be sorted to be [10, 20], [25, 35], [45, 60]. If a new interval say [15, 30] is added that overlaps with the previous intervals they are merged together and the list becomes [10, 35], [45, 60]. More formally, the sorted interval array $SA = ([x_1, X_1], ..., [x_m, X_m])$ has the following non-overlapping property:

$$x_i \leq X_i < x_{i+1} \text{ for every } i = 1, ..., m-1.$$

A more complex MAP arises when the predicate contains parameters corresponding to tests that involve more than a single attribute. For example:

$$P_t = (cost <? \wedge msrp <? \wedge num\_reviews = 5)$$

In this the case, the aggregation of various instantiations of the template is not straightforward. Note that we can associate a MAP with each parameter (the *cost* and the *msrp* upper bounds). However, a tuple can "hit" in both MAPs, while not matching any subscription (because it matches the *cost* test corresponding to one subscription, and the *msrp* test corresponding to another).

More generally, consider the case where the variant part of the conjunct contains more than one variant predicate term. We denote the variable part of the conjunct, C_j, as $\hat{C_j}$. Note that the columns referred to in $\hat{C_j}$ could be different. For a conjunct with predicate terms conditioning over the same column, we can sometimes convert $\hat{C_j}$ to a single BETWEEN predicate (e.g., in case of a combination of $\leq$ and $\geq$). However, in the general case, $\hat{C_j}$ can have predicate terms that refer to several different columns, for example: $\hat{C_j} = (col_1 <\ ?) \wedge (col_2 =\ ?) \wedge (col_3\ BETWEEN\ ?\ and\ ?)$. The MAP associated with $\hat{C_j}$ is a list of regions in a k-dimensional space. In this particular example, the region is upper-bounded by a single value along the first dimension (col_1), corresponds to a single value along the second dimension (col_2), and is upper and lower-bounded along the third dimension (col_3). In this 3-dimensional space, the region is described by a rectangular plane bounded on three sides and extending to infinity along the fourth side.

A single instantiation of the variant conjunct, $\hat{C_j}$, can therefore be thought to correspond to a region, or rectangle in k-dimensional space. The dimensionality of the region space is upper-bounded by the number of columns that appear in $\hat{C_j}$. We associate a MAP with $\hat{C_j}$ which maintains the set of rectangles corresponding to the variant conjuncts of all cached queries. Overlapping or adjacent rectangles are merged if possible. In general, a MAP contains a set of overlapping and/or disjoint rectangles.

Testing if a tuple matches the aggregation of several subscriptions reduces to inspecting the corresponding MAP to verify whether the "point" corresponding to the tuple is contained in any of the k-dimensional rectangles aggregated in the MAP. To allow quick search of such composite MAPs, one approach is to organize the rectangles corresponding to cached queries using a *memory-resident multi-dimensional index* such as an R-tree [10]. The rectangles are organized hierarchically using their minimum bounding rectangles (MBR). Testing a tuple against the aggregated subscriptions translates into searching the multi-dimensional index to find out if there exists at least one rectangle at the leaf level which contains the tuple.

Note that in the TPC-W application trace, we observe that less than 5% of the templates have more than one variable parameter in a conjunct. So, in practice, the need for R-trees and multi-dimensional MAPs should be limited.

Complexity analysis. Testing a tuple against a subscription is logarithmic in the number of aggregated subscriptions. Assume there are K templates in the cache, with each template having a single conjunct. Thus, there are S/K subscriptions from the cache associated with a given template, and the complexity is $O(K \times log(S/K))$ compared to the $O(S \times t)$ of the naive approach.

Subscription management. When queries are added to or removed from the cache, the subscriptions and the corresponding MAPs need to be updated in the filtering server. However, because MAPs represent aggregations of a set of values, updating them is not straightforward when a value in the aggregated set is removed.

A few alternatives can be distinguished here. One is to perform subscription deletion by entire template. A cache cannot unsubscribe by removing an individual query (i.e., a template instantiation). Instead, caches are allowed only to unsubscribe by dropping all the queries in a particular template.

The other alternative is to allow query-based unsubscriptions. For this, the MAPs must be modified to support individual query removals. For hash tables, removing a value can be implemented relatively easily. However, if the MAP is a min-max aggregation or a sorted interval list, updating the MAP after removing an element is not straightforward. The problem is recomputing a max over a set of values, when a random element is removed requires maintaining all the elements in the set. The same observation applies for the merged interval list. Therefore, under this scheme, we need to maintain all the individual elements that have been aggregated into the MAP. This set is not accessed during tuple tests, but is used to "recompute" the MAP when an element is removed. Note that MAPS can be lazily updated when unsubscriptions occur because sending more data to the cache than necessary increases load but does not compromise consistency.

5.2 Template-based filtering: Multiple caches

In this section, we consider the case where multiple caches subscribe to update streams at the back-end. One option is to use the single-cache template-based matching algorithm for each cache. This approach results in $O(N \times K \times log(S/K))$ filtering complexity for N caches. In this case, efficiency is gained only in aggregating queries from a single cache.

We investigate whether this simple algorithm can be improved if the templates are shared among caches. This arises in practice because often multiple "instances" of the same Web application can be executing on many edge servers. In this case, the subscriptions coming from the various caches will also be similar, having been generated by the same set of templates. Suppose that two caches have identical templates:

$T_1 : p_1 \wedge p_2 \wedge p_v$ and $T_2 : p_1 \wedge p_2 \wedge p_v$, where p_i are predicate terms.

When testing a tuple against these two template subscriptions, we need to test the tuple against $p_1 \wedge p_2$ for both caches. Then, we need to test the tuple against the aggregations of the variant parts. Suppose the variant part p_v is a simple equality query $p_v : attr == ?$. Suppose that cache 1 has cached *attr* values $7, 12, 15$ and cache 2 has cached *attr* values $12, 15, 16$. Under the simple scheme, the values for each cache would be stored in separate hash-tables. During the matching process, the filter server would consider each cache's MAP separately. The filter server would look up the value of `tuple.attr` in each cache's MAP hash-table. Alternatively, the values corresponding to the MAPs of various caches can be merged into a single unified MAP, which is capable of remembering which cache(s) each value is associated with. This can be achieved as follows. First, consider inequality tests. Without loss of generality, assume the variant part is of the form $attr <= ?$. While in the single cache case, the filter server maintains a single maximum value for all subscriptions that were generated by the same template, in the multi-cache case, the filter server maintains a sorted list of the maximum query parameter for each cache. The tuple matches queries for every cache whose maximum query parameter is greater than or equal to the value of the tuple's attribute. Merging MAPs of multiple caches will, however, require maintaining a more complicated data structure as query subscriptions are added and deleted.

In the case of interval tests, we need to modify the MAP to maintain information about the cache(s) to which each interval (in the sorted interval list) corresponds.

Complexity analysis. Assuming caches share templates, then the complexity can be further reduced from $O(N \times K \times log(S/K))$ to $O(K \times log(N\dot{S}/K))$.

6 Related work

Earlier work on database caching investigated predicate-based schemes and views to answer queries [18, 15, 8, 13, 9]. Recent papers have examined passive and active caching schemes for web applications and XML data [16, 6, 11]. Luo and Naughton described an active caching technique based on templates (forms) [16]. Consistency protocols for predicate caches have been investigated in [13], but the focus was on limited-size client-server database systems, and not on scalable deployments over the Web.

Related to the problem of update filtering is query containment— the problem of deciding whether a query is contained in the union of a set of views or cached queries. A wealth of previous work exists in the area of query containment and equivalence [17, 14, 5]. 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.

Most related to our work is the event matching approach of publish-subscribe systems [1]. Gryphon uses a parallel search tree (PST) to aggregate client subscriptions described as conjunctive predicate expressions. The difference in our approach is that instead of using a generalized PST we use specialized indexed data structures (MAPs) to aggregate the variant parts of subscription expressions based on the particular operator that appears in the variant part. This exploits the abundance of templates in Web applications, and enables compact representations of the data structures used to aggregate subscriptions and accelerate filtering.

7 Conclusions

Semantic database caching is a self-managing approach to dynamic materialization of "semantic" slices of back-end databases on edge servers. It can be used to enhance the performance of distributed Web servers, information integration applications, Web service platforms, and Web applications offloaded to edge-of-network servers. The scalability of semantic caching, however, relies heavily on protocols that maintain consistency between the back-end database and the distributed caches. In this paper, we focus the problem of "scaling" the update propagation protocols in such environments. In particular, we propose *template-based filtering* that efficiently aggregates cache subscriptions at the back-end by exploiting the similarity of "views" in edge query caches. We discuss the specifics of our approach, describe how we handle dynamic changes in cache subscription, and highlight the challenges and trade-offs that arise in this space.

References

[1] M. K. Aguilera, R. E. Strom, D. C. Sturman, M. Astley, and T. D. Chandra. Matching events in a content-based subscription system. In *Symposium on Principles of Distributed Computing*, pages 53–61, 1999.

[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, April 2002.

[4] K. Amiri, S. Park, R. Tewari, and S. Padmanabhan. DBProxy: A Self-Managing Edge-of-Network Data Cache. In *19th IEEE International Conference on Data Engineering*, pages 821–831, March 2003.

[5] K. Amiri, S. Park, R. Tewari, and S. Padmanabhan. Scalable template-based query containment checking

for web semantic caches. In *IEEE International Conference on Data Engineering*, pages 493–504, March 2003.

[6] L. Chen and E. Rundensteiner. XCache: XQuery-based Caching System. In *Proceedings of the Fifth International Workshop on Web and Databases*, June 2002.

[7] Colin C. Haley. IBM, Akamai Boost 'Virtual Capacity' . http://www.internetnews.com/infra/article.php/2200501.

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

[10] A. Guttman. R-Trees: A Dynamic Index Structure for Spatial Searching. In *Proceedings of the ACM SIGMOD Conference on Management of Data*, June 1984.

[11] V. Hristidis and M. Petropoulos. Semantic Caching of XML Databases. In *Proceedings of the Fifth International Workshop on Web and Databases*, June 2002.

[12] IBM Corporation. Websphere Edge Server. http://www-4.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] 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.

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

[16] Q. Luo, J. F. Naughton, R. Krishnamurthy, P. Cao, and Y. Li. Active query caching for database web server. In *WebDB Conference (Informal Proceedings)*, pages 29–34, 2000.

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

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