Scalable template-based query containment checking for web semantic caches

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

Semantic caches, originally proposed for client-server database systems, are being recently deployed to accelerate the serving of dynamic web content by transparently caching data on edge servers. Such caches require fast query containment tests to determine if a new query is contained in the results of cached queries. Query containment checking algorithms have been studied in the context of query optimization and materialized view selection, but their scalability remains a serious limitation. We argue that application queries are usually instantiations of a smaller number of base templates and show how this can be exploited to scale up containment checking. Our contributions include (i) algorithms to detect similarity between query predicates; (ii) efficient algorithms for proving containment among similar query predicates; (iii) a technique to dynamically aggregate similar queries in the cache to support efficient search; and (iv) integration of these schemes into a two-level containment checker. We describe our approach, report on its implementation in a dynamic web data cache, and show that it can reduce query containment cost by an order of magnitude for web workloads.

1 Introduction

Dynamic caching of structured data across distributed servers such as application servers and edge servers is a growing need of several web applications [3]. Besides e-commerce, dynamic data caching is also applicable to data integration systems, client-server databases [5, 12], and other emerging applications such as peer-to-peer data stores [7]. Techniques for data caching range from full or partial-table replication to exact-string-match based query response caches. Semantic caches, where the cached data is described by the set of query predicates, offer a *dynamic* and low administrative overhead solution to data caching. The concept of a semantic cache (also called predicate cache) had been proposed for client-server database systems [5, 12]. Recently, it has received renewed interest in the context of web-based dynamic data caching [15, 1, 2]. Several recent proposals have also suggested applying semantic caching to XML stores [4, 10].

In a dynamic semantic cache the data is populated on-the-fly (e.g., on a query miss) based on the application's query stream. The contents of the cache are described by the set of queries whose results have been inserted. A new query is considered to hit in the cache if it is logically contained in the previously cached queries. This containment check requires column (attribute) coverage, i.e., the necessary columns have been fetched by a previous query (or a union of queries), and that the new query's predicate is more *restrictive* than the predicates of the cached data. Sophisticated query containment algorithms have been proposed and studied in the literature [17, 13]. However, scalability of these techniques to a large set of cached queries and predicate terms is an issue. Our performance experiments with a dynamic data cache for web applications have shown that the cost of containment checking can become significant [2, 1]. The problem of query containment and equivalence has also been studied in the more generic problem setting of query rewriting and materialized view selection [14, 19, 16, 9]. However, these techniques are not directly applicable to the dynamic caching problem considered in this paper.

The scalability challenge of query containment check-

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

ing arises from the use of general algorithms designed for complex arbitrary predicates. Additionally, the checking becomes expensive as it needs to linearly scan through all the candidate cached queries. We observe that most applications (e.g., web-based forms or Java programs using prepared statements) issue *template-based* queries, whose selection predicates share the same structure across queries and differ only in a few numeric or string constants. In this paper we propose techniques to exploit these template-based queries in order to improve the performance and scalability of containment checking by dynamically aggregating similar queries in the cache, devising specialized algorithms and indexes to search them, and testing containment by checking a new query against the *aggregation of similar previously cached queries*.

Luo and Naughton proposed a form based proxy caching approach [15]. In their scheme, templates were specifically designed for conjunctive keyword based predicate terms such as *Oracle AND Java*. They also require that the web application designer must explicitly provide the template definitions to the proxy cache. In our work, templates can consist of arbitrary simple and complex predicates on several attributes based on numeric or string parameters, e.g., *A=5 AND B = 'JAVA' OR C='ORACLE'*. Also, we propose mechanisms to merge and aggregate a large number of similar query predicates in the cache efficiently. Furthermore, we propose schemes to infer templates from prepared statements or SQL strings by intercepting the JDBC calls at the proxy-cache layer.

Assuming M cached queries, n conjuncts in a query, and m terms in a conjunct, the worst-case complexity of a traditional query containment algorithm is $O(M \times m^2 \times n^2)$. Our template-based approach significantly reduces this complexity to $O(MAX\{(K \times m \times n),\ (n \times lg[\frac{M}{K}])\})$, where K is the number of templates. In practice, K is usually much smaller than M and likely to result in significant improvement in query containment performance. Our experiments validate this intuition by showing a 60% improvement in the response times of queries (both hits and misses).

1.1 Contributions

In this paper we focus on algorithms and data structures for efficient query matching which include:

Template-based query containment algorithms: While containment checking can be an expensive operation across arbitrary predicates, very efficient specialized algorithms can be devised to check containment for queries with similar selection predicates. We confirm this intuition with two theorems and show how they can be applied to the query containment problem for similar queries.

MAP data structure for dynamic query aggregation: We propose to aggregate values ranges of similar queries inserted in the cache dynamically using data structures, which we call *Merged Aggregate Predicates* (MAPs). The query containment problem is converted into a search of one or a limited number of MAPs. We discuss how MAPs are maintained as queries are inserted and removed from the cache and how they can be indexed for a quick search.

Template detection: When a query is received, it may not be declared by the application as an instantiation of an explicit parameterized template. We propose algorithms for detecting similarity between a newly received query and cached queries. We describe how an index of MAPs can be used to speed up this similarity detection step.

Integration in a dynamic edge data cache: We implement these concepts in a dynamic edge database cache for web applications, called DBProxy. We evaluate the performance of our query containment techniques in DBProxy and compare them to traditional approaches using two representative web e-commerce benchmarks. We show that our scheme can reduce the cost of containment checking by more than an order of magnitude compared to a general containment checking approach.

The remainder of this paper is organized as follows. We discuss preliminaries, including traditional query matching algorithms and application query programming styles in Section 2. We present an overview of our overall approach in Section 3. In Section 4, we describe the merged aggregate predicate (MAP) data structures and their usage and maintenance aspects. We discuss our approach to identifying templates using similarity detection between queries in Section 5. We evaluate the proposed algorithms in Section 6, discuss related work in Section 7 and summarize the paper in Section 8.

2 Preliminaries

We evaluated our template-based query matching techniques by implementing them in DBProxy, a dynamic edge data cache for web applications [1]. DBProxy maintains a large number of semantically consistent materialized views of previous query results in a local database. The content of the edge cache is described by a cache index containing the list of queries, their search predicates and the remaining clauses. When a new query is received, the list of queries that operated on the same table, table list or join condition is retrieved. A query containment checker 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.

Next, we describe a general query containment checking algorithm described in the literature, and employed in the initial implementation of the DBProxy edge cache. Furthermore, we report on its overhead and scalability under web workloads.

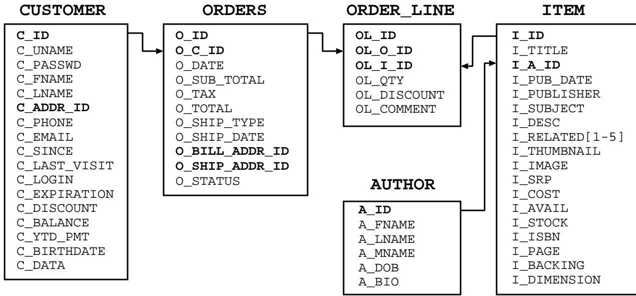

Figure 1. Simplified database schema of an on-line bookstore used by TPC-W. Customers can issue search queries and order books.

	Breakdown of HIT cost (msec)	
Database Size	**Containment Checking**	**Local Execution**
TPCW-10K	57.88	16.37
TPCW-100K	77.32	19.60

Table 1. Containment checking overhead. The table shows the breakdown of hit cost in a query cache supporting dynamic data caching in a web edge server. The numbers are reported from an execution of the TPC-W benchmark against a 10K and 100K item database. The workload consisted of 8 emulated browsers, running for an hour on a 400 Mhz Pentium II with 128 MB of memory.

2.1 General query containment checking

Consider a cached query, Q_1, and a newly received query, Q_2. Formally, the results of query Q_2 are contained in the results of query Q_1 if three conditions [13] are satisfied:

Attribute coverage: This requires the columns in the select list of Q_2 to be a subset of the select list of Q_1.

Tuple coverage: This means that for all instances d of the database, a tuple that belongs to the result set of Q_2 must also belong to the result set of Q_1. Alternatively, this means that for all possible values of a tuple t, the WHERE predicate of Q_2 implies that of Q_1. Denoting by P_1 and P_2 the WHERE predicates of Q_1 and Q_2 respectively, the tuple coverage requirement can be expressed as:

$$\forall(t) \quad : \quad P_2(t) \;\Rightarrow\; P_1(t)$$

This is true if and only if the expression, $P_2 \wedge \neg P_1$, is unsatisfiable.

Selectability: This requires that the new query can be evaluated on the local cache. That is, any columns mentioned in the WHERE or any additional clauses of Q_2 must be in the select list of Q_1.

Tuple coverage checking algorithm. By far, the most expensive condition to check is tuple coverage. DBroxy uses a tuple coverage checking algorithm based on the ones reported in [17, 13], which we extend to handle predicates over columns with continuous ranges (not just integers). The algorithm in general proceeds by converting the product expression $P_2 \wedge \neg P_1$ into an AND-OR normal form, expressing it as the OR of one or more conjuncts:

$$\begin{aligned} P_2 \wedge \neg P_1 &= C_1 \vee C_2 \ldots \vee C_n \quad \text{where:} \\ C_i &= t_1 \wedge t_2 \ldots \wedge t_m \end{aligned}$$

The leaf predicates (t_i's) are atomic predicates comparing columns, or a column and a constant (string, numeric, or set value), such as "*col op constant*" or "*col op col*", where $op = \{<, >, =, IN, LIKE, \ldots\}$. Note that the product expression $P_2 \wedge \neg P_1$ is unsatisfiable only if all the conjuncts are unsatisfiable. The algorithm tests each conjunct for satisfiability, one at at time, terminating if any one is found satisfiable. If all are unsatisfiable, then $P_2 \wedge \neg P_1$ is also unsatisfiable and containment is ensured. The main component of the algorithm is the one verifying the satisfiability of conjunct expressions of atomic predicates (the C_i's).

The algorithm proceeds in two steps. First, it adjusts the valid ranges for the columns based on predicates that compare a column with a constant value. Then, it processes atomic predicates that compare columns with each other (e.g. "$cost \leq msrp$") to verify whether given the current valid ranges for each column and the comparison constraint, there is a plausible set of values for all the columns mentioned in C_i. If any column has an empty set of plausible values, the conjunct is judged unsatisfiable. The algorithm has $O(n^2)$ worst-case complexity where n is the number of columns that appear in the conjunct C_i and is described in more detail in [13].

2.2 Scalability of general containment checking

Note that the cost of the containment checking grows with the number of conjuncts in the product expression ($P_2 \wedge \neg P_1$). Furthermore, the total cost of cache containment checking grows with the number of cached queries, since a new query must be checked against all the plausible candidate queries in the cache. Consider an application that submits the following two successive query predicates, changing the bounds on the *msrp* and *num_reviews* column:

$$\begin{aligned} P_1 &= (cost < 15 \wedge msrp < 8) \vee (num_reviews > 5) \vee (avail > 20) \\ P_2 &= (cost < 15 \wedge msrp < 4) \vee (num_reviews > 8) \vee (avail > 20) \end{aligned}$$

The general containment checking algorithm produces the

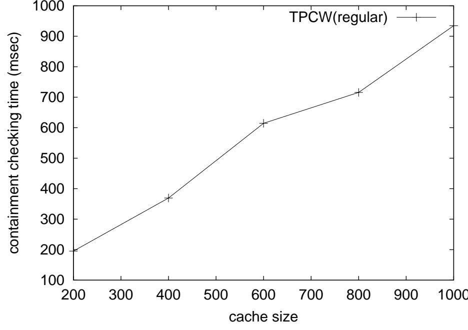

Figure 2. The cost of containment checking as the number of cached queries increases under the TPC-W benchmark. Cost is measured as the time taken on a Pentium II 400 Mhz machine by a Java implementation of the general containment algorithm to test for containment of a new query in the cache (in case of a miss).

following product expression for $P_2 \wedge \neg P_1$:

$$\begin{aligned}
&(cost < 15 \wedge msrp < 4 \wedge cost \geq 15 \wedge num_reviews \leq 5 \wedge avail \leq 20) \\
\vee\ &(cost < 15 \wedge msrp < 4 \wedge msrp \geq 8 \wedge num_reviews \leq 5 \wedge avail \leq 20) \\
\vee\ &(num_reviews > 8 \wedge cost \geq 15 \wedge num_reviews \leq 5 \wedge avail \leq 20) \\
\vee\ &(num_reviews > 8 \wedge msrp \geq 8 \wedge num_reviews \leq 5 \wedge avail \leq 20) \\
\vee\ &(avail > 20 \wedge cost \geq 15 \wedge num_reviews \leq 5 \wedge avail \leq 20) \\
\vee\ &(avail > 20 \wedge msrp \geq 8 \wedge num_reviews \leq 5 \wedge avail \leq 20)
\end{aligned}$$

After which it considers each conjunct (each line in the above expression) in turn, and computes the valid range for each column (*cost*, *msrp*, and *avail*) to discover that in each case the valid range is empty, declaring each conjunct unsatisfiable and, thereby, the entire expression to be unsatisfiable. In general, some constraints may compare columns against each other. This requires transitive updates of the valid ranges for a dependent column based on its related columns.

Table 1 shows the proportion of time devoted to containment checking in case of a cache hit in our DBProxy edge cache. The table reports costs for the TPC-W benchmark for a 10K and 100K item database. As shown in Figure 2, the cost of containment checking (in case of a miss) for a query can grow to tens or hundreds of milliseconds even for a limited size cache (100-200 queries). Furthermore, it does not scale well as the number of cached queries increases. Note that in the above example, if we recognized that the only values that changed between P_1 and P_2 are the upper-bound of *msrp* and the lower bound of *num_reviews*, we would more easily deduce that $P_2 \Rightarrow P_1$.

3 Template-based query containment

Queries submitted by applications often follow a template style. For instance, in Java-based web applications, two query programming styles can be seen: 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.

Notation. Before we get into the details of our approach, we define a few notational conventions to simplify the rest of the discussion. We denote a generic *predicate expression* (interchangeably called a predicate) by P_i. When a predicate is expressed in its AND-OR normal form, we denote by C_i the ith *conjunct* in that expression. Each conjunct contains several *predicate terms* ANDed together, with the predicate terms denoted as t_k. These predicate terms are atomic conditions, such as equality or inequality predicates over columns. When multiple conjuncts appear in the expression, the terms are denoted by a double subscript, with the jth term in C_i denoted as t_{ij}. When referring to a group of similar query predicates such as the instantiations of a particular template, we refer to the part of the predicate that changes across the group as the *variant predicate*. Similarly, when considering a single conjunct in the predicate, we refer to the changing part as the *variant conjunct*. In the example shown below, P_1 and P_2 are the predicate expressions, while $(cost < 15 \quad \wedge \quad msrp < 8)$ is one of the conjuncts, and within it $(cost < 15)$ is a predicate term. Furthermore, within the conjunct $(cost < 15 \wedge msrp < \ ?)$, the variant conjunct consists of a single predicate term, namely $(msrp < \ ?)$.

Templates. Consider the two predicate expressions described in Section 2:

$$\begin{aligned}
P_1 &= (cost < 15 \wedge msrp < 8) \vee (num_reviews > 5) \vee (avail > 20) \\
P_2 &= (cost < 15 \wedge msrp < 4) \vee (num_reviews > 8) \vee (avail > 20)
\end{aligned}$$

A comparison of these two predicate expressions suggests that they are likely two instantiations of a template P_t of the form:

$$P_t = (cost < 15 \wedge msrp < \ ?) \vee (num_reviews > \ ?) \vee (avail > 20)$$

The main intuition behind our approach is that once the common terms are removed, the *variant predicate* is much smaller and faster to test for containment. In particular, the aggregated values of the variant predicates can be indexed as data for fast containment testing. In this section, we present two basic theorems that support the intuition that proving containment across the variant predicates is sufficient to ensure containment across the entire predicate expressions.

3.1 Variant Predicate: Basic theorems

First, we consider two conjuncts, which differ in some predicate terms while agreeing in the remaining predicate terms. The following theorem holds.

Theorem 1 *Consider two conjuncts, C_1 and C_2:*
$$\begin{cases} C_1 = t_1 \wedge t_2 \ldots \wedge t_k \wedge t_{k+1} \ldots \wedge t_n \\ C_2 = t'_1 \wedge t'_2 \ldots \wedge t'_k \wedge t_{k+1} \ldots \wedge t_n \end{cases}$$
Suppose that only the first k predicate terms of C_1 and C_2 differ. Let d denote an instance of the database on which the predicates are defined. The following result holds:

$$if\ \forall (t \in d) \quad \wedge_{i \in \{1,\ldots,k\}} t'_i \Rightarrow \wedge_{i \in \{1,\ldots,k\}} t_i \quad then \quad \forall (t \in d) \quad C_2(t) \Rightarrow C_1(t)$$

Recall that $\forall t: \quad C_2(t) \Rightarrow C_1(t)$, written also as $C_2 \Rightarrow C_1$, means that for any tuple t in the database d, if t matches C_2 then it also matches C_1. Or equivalently, the result set of C_2 is a subset of that of C_1. The theorem states that it is sufficient to show logical implication across the variant part of the conjunct ($t'_1 \wedge t'_2 \ldots \wedge t'_k \Rightarrow t_1 \wedge t_2 \ldots \wedge t_k$) to guarantee that $C_2 \Rightarrow C_1$, essentially ignoring the constant part across C_1 and C_2.

Theorem 2 *Consider two predicate expressions, P_1 and P_2, represented in their AND-OR normal form, consisting of n terms. Suppose that P_1 and P_2 differ in the first k conjuncts and agree in the remaining $n-k$ conjuncts:*
$$\begin{cases} P_1 = C_1 \vee C_2 \ldots \vee C_k \vee C_{k+1} \ldots \vee C_n \\ P_2 = C'_1 \vee C'_2 \ldots \vee C'_k \vee C_{k+1} \ldots \vee C_n \end{cases}$$
The following holds:

$$if\ C'_i \Rightarrow C_i \ \forall_{i \in \{1,\ldots,k\}} \ then\ P_2 \Rightarrow P_1.$$

Example: Consider the two queries in the example of Section 2.2. In this case, the last conjunct consisting of the predicate term $(avail > 20)$ is the common part across expressions, while the first conjunct $(cost < 15 \wedge msrp <?)$ and the second conjunct consisting of one predicate term $num_reviews >?$ differ across instantiations. By Theorem 2, it suffices to show implication among the variant predicate sub-expressions to establish implication across the entire conjunct. That is, it suffices to show the following:

$$\begin{array}{lcl} (cost < 15 \wedge msrp < 4) & \Rightarrow & (cost < 15 \wedge msrp < 8) \\ (num_reviews > 8) & \Rightarrow & (num_reviews > 5) \end{array}$$

to conclude that $P_2 \Rightarrow P_1$. ■

3.2 Containment checking via variant predicates

Our approach requires that all predicate expressions are converted to their AND-OR normal forms. All NOT operators are removed from the predicates. This is achieved by applying DeMorgan's Law ($\neg(P_j \wedge P_k) = \neg P_j \vee \neg P_k$ and $\neg(P_j \vee P_k) = \neg P_j \wedge \neg P_k$). The NOT operators that propagate down to the leaf terms (t_i's) are removed after reversing the conditions inside the leaf predicate terms. For example, "$\neg(cost < 5)$" becomes "$cost \geq 5$", and "$\neg(cost = 10)$" becomes "$(cost < 10 \vee cost > 10)$". Finally, a predicate, P, is expressed in its AND-OR normal form as follows (where the terms t_i are predicate terms such as "$col\ \ op\ \ col$" or "$col\ \ op\ \ constant$", etc.):

$$\begin{array}{ll} P = & C_1 \vee C_2 \ldots \vee C_n \quad \text{where each conjunct is of the form:} \\ C_i = & t_1 \wedge t_2 \ldots \wedge t_m \end{array}$$

Eliminating common disjuncts: Suppose that P_t is a parametrized query predicate template. Consider two successive instantiations of P_t, namely P_1 and P_2 respectively. When expressed in their AND-OR normal form, P_1 and P_2 can be expressed as follows:

$$\begin{array}{ll} P_1 = & C_1 \vee C_2 \ldots \vee C_k \vee C_{k+1} \vee C_n \\ P_2 = & C'_1 \vee C'_2 \ldots \vee C'_k \vee C_{k+1} \vee C_n \end{array}$$

In the above expressions, the conjuncts C_{k+1} through C_n are common across instantiations, while the first k conjuncts may differ. By Theorem 2, the query containment checker can simply focus on the first k conjuncts. In particular, the goal of testing that P_2 is contained in P_1, is transformed into one of establishing implication among the first k conjuncts of P_1 and P_2. Precisely, it has to show that:

$$C'_i \quad \Rightarrow \quad C_i \quad \forall_{i \in \{1,\ldots,k\}}$$

Conjunctive expression containment: Since the $C'_i s$ are conjuncts, the problem of testing containment reduces to that of testing containment across similar conjuncts. A conjunct (C_i above) is an AND of one or more predicate terms t_{ij}, such as '$col\ \ op\ \ constant$'. For simplicity, we have grouped all the common predicate terms in the conjunct together and called it c and similarly we have grouped all the differing terms. Separating out the common and differing parts of C'_i and C_i we have:

$$\begin{array}{lcl} C'_i & = & t'_1 \wedge c \\ C_i & = & t_1 \wedge c \end{array}$$

Theorem 1 can be used to simplify this task by ignoring the common part (c) of the conjunct and focusing on the variant part, where it is sufficient to show that: $t'_1 \Rightarrow t_1$.

Conjunctive expression containment across multiple similar queries: So far we discussed pair-wise matching of two similar queries. We can further match against multiple cached queries belonging to the same template. Consider a new query with conjunct C_0. We are interested in testing its containment in m previous queries where each is an instantiation of the same conjunct, denoted by C_1

through C_m. Expanding each conjunct in the above expression into its different and common predicate terms respectively, $(C_i = t_i \wedge c)$, we reduce the containment test across the m queries to establishing the following sufficient condition:

$$t_0 \wedge c \quad \Rightarrow \quad (t_1 \vee t_2 ... \vee t_m) \wedge c$$

By Theorem 1, ignoring the common part, it is sufficient to show that:

$$t_0 \quad \Rightarrow \quad (t_1 \vee t_2 ... \vee t_m)$$

For example, let's consider a simple conjunct, one with a single term, such as $t_0 : msrp < 20$. The containment test of t_0 against 3 previously cached instantiations, say $t_1 : msrp < 30$, $t_2 : msrp < 45$, $t_3 : msrp < 25$, can be handled by maintaining the maximum value of all previously cached predicate terms and comparing against it, i.e., to match against $msrp < 45$. Essentially, our approach is to merge previous individual predicate term instantiations into a merged aggregate predicate (MAP) to simplify containment checking. In Section 4, we describe in detail how MAPs are created and stored for each type of variant conjunct.

3.3 Overall containment checking algorithm

When a new query Q_n is received, we check if it is an instantiation of an explicit query template (i.e., a JDBC *PreparedStatement*). If it is not an instance of a *PreparedStatement*, then we examine the query to see if it is similar to previously received queries. Specifically, a template-matcher compares the structure of the query to previous ones in an attempt to infer if it follows an undeclared (implicit) query template.

If the new query, Q_n, is found to be an instantiation of an explicit or implicit query template, a MAP-checker checks if Q_n is contained in the MAPs associated with the template. In case of a miss, and if Q_n is inserted in the cache, the MAPs aggregating the predicates of cached queries are updated to include the selection predicate of the new query. If template-based checking fails to prove containment, the general containment checker can be invoked to test for containment against cached queries that are not similar to the new query. Specifically, the general checker tests the new query for containment in the list of cached queries that operated on the same table(s), and which satisfy the attribute coverage and selectability conditions.

3.4 Complexity Analysis

To compare the general containment checking algorithm described in Section 2 with our template-based checking algorithm, consider a semantic cache containing M queries. To simplify the analysis, assume that the conversion of each query to its AND-OR normal form generates n conjuncts each consisting of m predicate terms.

General containment checking: In this case, containment checking consists of testing the satisfiability of the expression $(P_2 \wedge \neg P_1)$. Assuming that the negation of P_1 also contains n conjuncts each with m predicate terms, the AND with P_2 will factor in another n conjuncts making it a total of n^2 conjuncts each with $2 \cdot m$ terms that have to checked for satisfiability. The checking of a conjunct with $2 \cdot m$ terms in the worst case could be $O(m^2)$. Thus the total complexity for a satisfiability check among two queries becomes $O(n^2 \cdot m^2)$. Checking against all the M cached queries will result in an overall complexity of $O(M \cdot n^2 \cdot m^2)$.

Template-based containment checking: Assume in this case that the semantic cache consists of K templates, where $K \leq M$. When a new query is received, and if it is not generated by an explicitly declared template, it must be matched with one of the cached templates. For a query with n conjuncts with m terms, the complexity of trying to match a query with a single cached template is $O(m \cdot n)$. Given K cached templates, the complexity is $O(K \cdot m \cdot n)$. Once a template is identified, testing for containment proceeds one conjunct at a time. Each conjunct in the template is associated with a MAP, and therefore in the worst case, the number of MAPs to check against equals the number of conjuncts n. Testing for containment within a single MAP is $O(1)$ for simple inequalities. For complex MAPs, as we describe in Section 4, the matching will be $O(lg[\frac{M}{K}])$, where $\frac{M}{K}$ is the average number of queries that belong to the same template. The complexity of containment testing within all the MAPs associated with the template is $O(n \cdot lg[\frac{M}{K}])$. The overall complexity for template-based containment checking is, therefore, $MAX\{(K \cdot m \cdot n),\ (n \cdot lg[\frac{M}{K}])\}$, which is much lower than that of the general approach.

4 Merged Aggregate Predicates (MAPs)

In this section, we discuss how MAPs are used to perform containment tests and how they are maintained when queries are added and removed from the cache. The implementation of a MAP depends on the nature of the variant conjunct. The variant part of a conjunct can be a single predicate term or a bunch of terms. As discussed earlier, when the WHERE predicate is converted to an AND-OR normal form, there can be more than one conjunct with a variant part in it. Recall that a MAP is associated with the variant part of a *single conjunct*, therefore, a cached query predicate is associated with a list of MAPs, one for each conjunct which contains a variant part.

First, we discuss the case when the variant part of the conjunct consists of a single predicate term. In particular, we describe MAPs for each condition type occurring in such a predicate term: i) equality, ii) inequality, iii) BETWEEN,

Input : new query Q_n, template (storing previous instantiations)
Output : HIT or MISS

```
1  contained := MISS;
2  conj_map = template.conj_map[i];
3  for (int i=0; i < Q_n.num_param_conjuncts; i++) do
4  |  new_conj = Q_n.conjunct[i];
5  |  contained := conj_map[i].containCheck(new_conj);
6  |  if (contained != HIT) then
7  |  |_ return MISS;
   |_
   return HIT;
```

Algorithm 1: Template-based containment checking (mapContainCheck).

or iv) *other* conditions such as $\neq$, IN and NOT IN. Second, we discuss the more complex case when the variant part of the conjunct consists of several predicate terms. In this case, more complex composite MAPs are needed.

To illustrate the overall approach of MAP-based containment checking, consider the following query template in its original form as declared by the application:

$$P_t = (cost < 15 \vee msrp < 20) \wedge (num_reviews > ?)$$

When converted to its AND-OR normal form, the query selection predicate becomes:

$$P_t = (cost < 15 \wedge num_reviews > ?) \vee (msrp < 20 \wedge num_reviews > ?)$$

The template has two conjuncts, $C_1 = (cost < 15 \wedge num_reviews > ?)$, and $C_2 = (msrp < 20 \wedge num_reviews > ?)$. Both conjuncts have variant parts consisting of a single predicate term. In this case, two simple maps are associated with this template, `conj_map[1]` and `conj_map[2]`, corresponding to the two conjuncts, C_1 and C_2, respectively. As instantiations of this template are inserted in the cache, the two maps are updated accordingly based on the parameters specified in the predicate. Upon receiving a new query predicate, for example $(cost < 15 \vee msrp < 20) \wedge (num_reviews > 12)$, the containment test is reduced to: `conj_map[1].containCheck(12) AND conj_map[2].containCheck(12)` as shown in Algorithm 1.

An interesting dimension to aggregating predicates arises when queries are removed from the cache due to *cache replacement*. The unit of granularity of cache replacement in a query result or semantic cache varies from one system to another. One approach is to remove all the queries belonging to a given template together. Of course, this template-based replacement is the simplest approach to implement. In this case, it is not necessary to maintain any additional information in a MAP, since the entire MAP will be removed in a cache replacement decision. Another possible unit is an individual query. However, any query removal should ensure that the value(s) stored in the MAP are appropriately adjusted. Some of the predicates will require extra information to handle single query removals.

4.1 Equality predicates

In this case, we focus on a single conjunct that has a single variant predicate term, t, which consists of an equality test of type '$col = ?$'. Given m queries in the cache that follow this template, we denote by t_1 through t_m the instantiations of their respective predicate terms. We express the predicate terms, t_j (where $j \in \{1,...,m\}$), as $t_j : col = x_j$. We desire an efficient method to test for the containment of a newly received predicate term $t_0 : col = x_0$ in the union of cached predicates with terms, t_1 through t_m. To achieve this, we organize the values x_1 through x_m into a hash table. An inclusion test is sufficient for containment.

Replacement. Under individual query-based replacement, the cache replacement mechanism selects a query to evict from the cache. Suppose that this is the jth query which was associated with a value x_j. During eviction, all we need to do is remove the value x_j from the hash table. This is complicated, however, by the fact that x_j may have been referred to by another query that is inserted in the cache. One solution to this problem is to associate a `refcnt` with each value in the cache, which is incremented and decremented whenever a query is added or removed from the cache. Values with a zero `refcnt` in the hash table can be safely removed. Another solution is to allow the insertion of duplicate values in the hash table, one corresponding to each query. This approach differentiates between two values based on the identity of the query that inserted them.

4.2 Inequality predicates

Suppose t is a predicate term of the type '$col < ?$'. As before, the cached instantiations of the predicate terms are denoted by t_1 through t_m, and expressed as $t_j : col < x_j$. Upon receiving a new query predicate term $t_0 : col < x_0$, we want to quickly test its containment in t_1 through t_m. Note that all we need in this case is a single variable, $X = MAX_{j \in \{1,...,m\}}(t_j)$, which stores the maximum value of the x_j's. Similarly, if the term is of the form $col > ?$, then, all we need is a single variable, $X = MIN_{j \in \{1,...,m\}}(x_j)$. A similar approach can be used with the operators $\leq$ and $\geq$. However, in addition to the max/min variable, we need to recall whether the interval is open or closed. This approach applies to numeric and string constants, as long as the string constant does not contain regular expressions.

Replacement. Evicting a query with a value $x_j = X$ is not

straightforward, because the new value of the maximum (or minimum) X cannot be easily known. This is related to the general problem of efficiently computing an aggregate value over a data set which may be updated. One approach is to maintain a multi-set of all the values x_j of all cached predicates. A multi-set is a list augmented with a `refcnt` marking how many times each value occurs in the set (i.e., the number of queries which refer to that value). Upon an eviction of a query with $x_j = X$, the `refcnt` for the value x_j is decremented in the set and the aggregate value recomputed. For example, to test the containment of $t_0 : msrp < 20$ against 3 previously cached instantiations, say $t_1 : msrp < 30,\ t_2 : msrp < 45,\ t_3 : msrp < 25$, we maintain the maximum value of all previously cached predicate terms, i.e., 45, along with the values of all the cached predicate terms and their `refcnt`, i.e., $((30,1),(45,1),(25,1))$.

4.3 BETWEEN predicates

Our approach is to merge successive instantiations of a BETWEEN predicate into an interval set, where all overlapping intervals are merged together. Further, the intervals are sorted to enable efficient binary searching. Consider a list of predicate terms, expressed as t_j : '$col\ BETWEEN\ x_j\ AND\ X_j$'. The interval set is described as $S_I = \{[x_1,X_1],[x_2,X_2],...,[x_m,X_m]\}$. Verifying the implication of $t_0 \Rightarrow (t_1 \vee ... \vee t_m)$ requires checking if the interval $[x_0,X_0]$ is a subset of the union of intervals in S_I.

To perform a fast test of interval containment, we maintain S_I as a *sorted interval array SA*. Whenever a new interval is added by a new instantiation of the template, the interval I_k is inserted in SA, merging it with any overlapping intervals. If there is no intervals in SA overlapping with I_k, it is added to SA as a new element. Suppose that there are t intervals in SA, $I_1 = [x_1,X_1],...,I_t = [x_t,X_t]$, which overlap with a new interval $I_k = [x_k,X_k]$. Then these $t+1$ intervals are merged into the single interval $[min(x_k,x_1,...,x_t), max(X_k,X_1,...,X_t)]$. Therefore, the sorted interval array $SA = ([x_1,X_1],...,[x_m,X_m])$ has the following non-overlapping property:

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

Given any new interval $I_0 = [x_0,X_0]$, we perform a binary search of SA to check if there is an interval in SA which subsumes I_0. The search proceeds by finding if there is an interval which contains x_0, the lower bound of I_0. If one exists, the upper bound of I_0 is checked for inclusion within that interval. There should be at most one such interval in SA. The complexity of this binary search is $O(\log_2 |SA|)$ where $|SA|$ is the number of intervals stored in SA.

Replacement. Evicting a query translates into removing an interval from the list of intervals in the MAP. For non-overlapping intervals it is straightforward. For merged intervals, we need to check if the interval can remain unchanged or whether it must be broken up into two intervals. This step requires considering the list of intervals in S_I which overlap with the interval being removed.

4.4 Other predicates

There are some types of predicate terms which were not mentioned in our discussion, such as $\neq$ (col <> val), IN ($col\ IN\ (c_1,c_2,...)$) or NOT IN predicates. Such predicate terms can be transformed during the computation of the AND-OR normal form into predicate terms which have been already discussed. For instance, a NOT-EQUAL predicate is logically equivalent to two comparison predicates ORed together, '$col < val\ OR\ col > val$'. IN predicates are logically equivalent to a set of equality predicates connected by the *OR* operator. *NOT IN* is similarly equivalent to a set of not-equal predicates connected by the AND operator. It is worth noting, however, that handling *IN* predicates explicitly can be more efficient than converting them into a list of equality predicates connected with the *OR* operator. The extension of our approach to handle IN predicates explicitly is relatively straightforward.

4.5 Conjunctions of atomic predicates

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 $\hat{C}_j$. $\hat{C}_j$ can have predicate terms that refer to different columns, for example: $\hat{C}_j = (col_1 <\ ?) \wedge (col_2 =\ ?) \wedge (col_3\ BETWEEN\ ?\ and\ ?)$. 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. Checking for containment of a new variant conjunct associated with a new query in the MAP translates into verifying whether the rectangle corresponding to the incoming query is contained in the union of 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 [8].

As a more detailed example, consider the following variant conjunct and its three instantiations:

C_t:	(A = ?)	and (B between ? and ?)	and C $\leq$?
C_1:	(A = 5)	and (B between 4 and 10)	and C $\leq$ 10
C_2:	(A = 10)	and (B between 6 and 8)	and C $\leq$ 13
C_3:	(A = 10)	and (B between 4 and 8)	and C $\leq$ 5

Each C_i generates a rectangle in 3-d space. For instance, C_1 corresponds to 5, $[4,10]$, $(-\infty,10]$. Note that the rectangles corresponding to C_2 and C_3 in the above example cannot be merged into a single one: C_{merged}: (A=10) and (B between 4 and 8) and C $\leq$ 13. If we merge these two regions, a new conjunct C_0: (A=10) and (B between 4 and 6) and C $\leq$ 10 would be judged as contained in the MAP; however, a tuple (A=10, B= 5, C=10) could be part of C_0 but is neither contained in C_2 nor C_3.

Replacement. To add a query, we add a rectangle and adjust the multi-dimensional index. When a query is evicted the index can be adjusted lazily or immediately after insertion.

5 Template Detection

As described in Section 3, template-based query containment checking is much more efficient compared to the general matching algorithm. Therefore, even if query templates are not declared explicitly by applications, it is beneficial to infer such templates by detecting similarity between queries. Template inference is often required even if the application declares templates explicitly (e.g., using the `preparedStatement` interface), because due to space and connection management, applications often deallocate a prepared statement soon after its use. In this section, we propose two algorithms to infer templates: (i) a simple string-based *constant-removal algorithm*, and (ii) a *canonical-form ordering algorithm* with enhanced inference capability.

Constant-removal: The constant-removal algorithm is based on the following observation: instantiations of the same query template should have similar WHERE predicates and identical remaining clauses. To declare two predicates as similar, we need to ensure that all their conjuncts (C_i) are similar. Two conjuncts are similar if they have the same number of predicate terms and all the corresponding predicate terms (t_i) are similar. Two predicate terms are similar if they contain the same column name, the same comparison operator, but possibly different constant values.

The above definition of 'similarity' implies that two *similar* queries become *identical* after removing all the constant values in the queries. If there is a cached query Q_i whose constant-less query string matches exactly that of the new query Q_n, the new query is inferred to be an instantiation of the template associated with Q_i. Note that we simply match the query string without any parsing, thereby, incurring low overhead. Otherwise, the inference algorithm fails and returns *FALSE*.

Canonical-form ordering: While the above algorithm is highly effective in practice, it has some limitations. For example, it does not detect similarity if the order of two commutative terms in a predicate expression is switched around.

```
Input  : new query Q_n, set of cached queries Q_c
Output : flag inferred
1 inferred := FALSE;
2 np := toCanonicalForm(Q_n.wherep);
3 for (each Q_t in Q_c such that (Q_n.table_list ==
  Q_t.table_list && Q_n.col_list ⊂ Q_t.col_list) ) do
4 |  tp := Q_t.canon_wherep;
5 |  if ( (Q_n.mint == Q_t.mint) && (Q_n.maxt == Q_t.maxt)
  |  ) then
6 |  |  if (equalPredicates(np, tp)== TRUE) then
7 |  |  |  Q_n.template := Q_t;
8 |  |  |  inferred := TRUE;
9 |  |  |  break;
  return inferred;
```

Algorithm 2: Canonical-form template inference algorithm canonInferTemplate.

The basic idea of this approach is to express the predicates in a canonical form before performing the comparison. This requires parsing the query predicate and converting it to an AND-OR form, then expressing it in a canonical order. Note that the steps of parsing and AND-OR normal form conversion are required by the MAP-based containment checker anyway, therefore, this approach places no extra overhead of its own. Once in AND-OR normal form, each conjunct is first transformed into a canonical representation, by arranging its predicate terms according to lexicographic order. Subsequently, the C_i's in the AND-OR normal form are in turn sorted lexicographically with respect to each other. For example, following is an example predicate before and after transformation to canonical form:

(SALARY $>$ 40000 $\wedge$ MANAGER = 'MIKE') $\vee$ (DEPT = 65)
(DEPT = 65) $\vee$ (MANAGER = 'MIKE' $\wedge$ SALARY $>$ 40000)

We also associate with each complex predicate two integers, the maximum number of terms in any conjunct (*maxt*) and the minimum number of terms in any conjunct (*mint*) of its AND-OR normal form to speed up matching.

Comparing two predicates in canonical form can be performed by comparing the resulting string representation of the predicates. String comparisons can be accelerated by computing signature codes over the strings corresponding to cached templates.

Template indexing. To reduce the number of templates actually compared to a new query, Algorithm 2 contains an additional filtering step. It first checks to verify whether both predicates (the new and the cached) agree in the size of the smallest and largest conjunct (*mint* and *maxt*) in the AND-OR normal form of the canonical-form predicate. Other attributes of a predicate can be used to do this filtering. An

index over the *mint* and *maxt* attributes is built to speed up the filtering of matching cached predicates.

When matching a new query against cached templates, we have so far required that they both have the same number of conjuncts, and that their conjuncts be similar. In fact, this requirement can be relaxed. For example, a query can have fewer conjuncts than the template it is matched with. It is only required that each conjunct in the new query's predicate matches a conjunct in the cached template.

6 Evaluation

We evaluate the performance of the proposed approach by comparing its achieved hit rate and containment checking overhead to an implementation of general containment checking. We first perform microbenchmarks that focus on measuring the performance of containment checking in isolation, then we compare end-user response time by integrating both implementations of containment checking in our DBProxy cache.

Evaluation environment. Both implementation were in Java and executed on a Pentium II 400 Mhz, with 128 MB of memory, running Linux RedHat 7.1. Two traces were collected from runs of two e-commerce benchmarks, namely TPC-W [20] and the Trade-2 WebSphere Commerce Suite (WCS) benchmark [11]. TPC-W is a transactional web benchmark, emulating user browsing and shopping patterns in an on-line bookstore. WCS is an integrated solution which is used by a large number of companies for managing on-line stores and applications. While TPC-W has a small number of templates accessing a small number of tables, WCS benchmark has many more templates targeting more than 500 tables and has many indices and triggers. Trade-2 is a benchmark that exercises the integrated WCS system. Table 2 describes the characteristics of the collected traces.

Traces. First, observe that the number of templates is much smaller than the number of queries which confirms the intuition that motivated our work on template-based query containment. Second, observe that the query distribution is highly skewed, with 70-80% of the queries mapping to the top 5 templates. This observation motivated our work on efficient MAP indexing because a few MAPs would contain a large number of queries.

Experimental methodology. In the first set of experiments, the goal was to quantify the reduced cost of containment checking along with the reduction of hit rate for different cache sizes. Initially the cache was populated to the desired size by running the trace. To capture the overhead, a query mix, containing five queries most frequently appearing in the trace, was executed, and the measurements were averaged over 50 samples. The same query mix was re-executed for each cache size data point, but each time changing the variant parameters in the query randomly to avoid submitting the exact same query. The queries in the cache only consist of the ones in the trace; those from the query mix are never inserted. For measuring the overhead, only the cache index was maintained without actually storing (or retrieving) the data in the local repository. The containment checker recorded a hit if it was a previously cached query, otherwise on a miss, the metadata (index and MAPs) was updated.

Trace	# of queries	# of templates	# of tables	% of queries from top-5 templates
TPC-W	11369	18	7	79%
Trade-2	6110	81	63	68%

Table 2. Characteristics of the traces used in the microbenchmarks. The traces contain query logs collected during the execution of the TPC-W and Trade-2 e-commerce benchmarks. For the TPC-W trace, the log was collected from a load of 4 emulated browsers accessing a 100K item store database. The Trade-2 log was collected from a similarly configured system with two users.

The experiments include the cost of template inference, as the application does not explicitly declare the templates. Each query is submitted as a separate constant query with no question marks. Our template inference engine classifies the queries using the algorithms discussed in Section 5.

Containment checking overhead. The template-based containment checker, as shown in Figures 3 and 4, is an order of magnitude faster than the general-purpose checker. Furthermore, the cost of template-based checking is relatively independent of the cache size (i.e., the number of queries in the cache) compared to the proportional increase of the general-purpose checker. Thus template-based matching is not only faster but also more scalable. Scalability is achieved partly due to the number of templates being small and being largely independent of the cache size. Secondly, the aggregation of queries into MAPs along with binary or hash-based searches speeds up lookup. One interesting observation is that the relative benefit of our approach (ratio of baseline to template-based) is higher with TPC-W than with the Trade-2 benchmark. As the trace statistics in Table 2 show, TPC-W has a smaller number of tables, and correspondingly larger number of queries per table. Under Trade-2 (WCS), the queries are distributed across a large number of tables, with fewer queries per table. Since the general containment checker must consider most of the queries over the same table it has a larger overhead in TPC-W than in the Trade-2 case.

Hit rate. Template-based checking has a slightly lower hit rate (5-10%) compared to the general purpose checker as shown in Figure 5 and 6. Since template-based matching

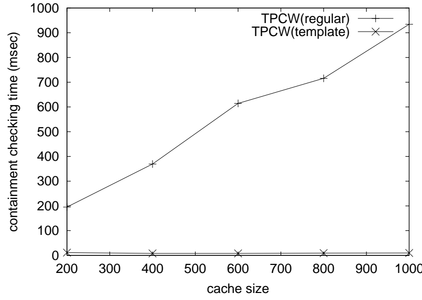

Figure 3. Cost of containment checking versus cache size for TPC-W.

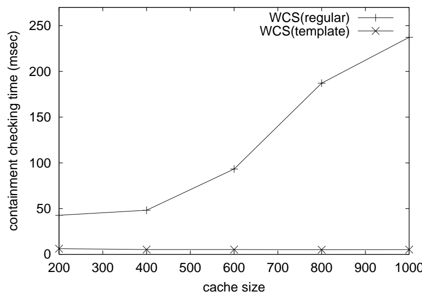

Figure 4. Cost of containment checking versus cache size for Trade-2 (WCS).

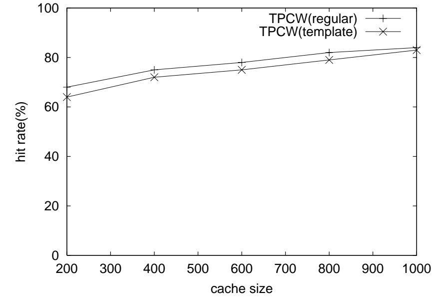

Figure 5. Hit rate versus cache size for TPC-W.

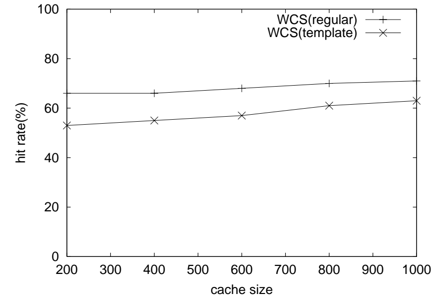

Figure 6. Hit rate versus cache size for Trade 2 (WCS).

does not check for containment across queries belonging to different templates it sometimes results in false misses, i.e., a query is declared a miss although it is contained in a cached query. This is due to the underlying premise of template-based checking that there is a much higher likelihood for a query to be contained in a cached query belonging to the same template. The general containment checker, on the other hand, checks against all candidate cached queries and has a higher hit rate. Template-based matching tries to balance the tradeoff of lower hit rate with faster containment checking, thereby reducing the total response time as shown below.

Application to edge data caching over the Web. In this experiment, we compare the total end-to-end response time. The client machine ran 8 emulated TPC-W browsers connected to an edge server (Pentium II, 400 Mhz, 128 MB RAM, Linux RedHat 7.1) using a fast Ethernet (100 Mbps) network. The servlet running at the edge server performs SQL queries to a back-end database server (Pentium III, 1 GHz, 256 MB RAM). The network connecting the edge server and the origin has an emulated latency of 225 milliseconds per average result set transfer. Our DBProxy dynamic data cache is deployed on the edge server, and transparently intercepts the JDBC-level queries being sent to the origin server. The cache is described briefly in Section 2 and more details can be found in [1]. In case of a hit, data is retrieved from the local cache, otherwise, a query is sent to the origin server and the data inserted in the cache. In one experiment we used a traditional containment checker, and in the other we used our template based approach. 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% from 863 msec., with general containment checking to 329 msec., with template-based checking.

7 Related work

Earlier work on database caching investigated predicate-based schemes and views to answer queries [18, 14, 5, 12, 6]. Recent papers have examined passive and active caching schemes for web applications and XML data [15, 4, 10]. Luo and Naughton described an active caching technique based on templates (forms) [15]. Our work considers a much larger set of predicate types for template matching that include numeric and string comparisons combined using conjuncts, disjuncts and negations. Our technique is also able to infer templates transparently by intercepting incoming JDBC SQL requests.

A wealth of previous work exists in the area of query containment and equivalence [17, 13]. 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. A good survey of work on answering queries using views can be found in [9]. The view selection and rewriting issues are more general than predicate expression containment. Most recently, Pottinger *et al.* have proposed more scalable approaches to the general view matching problem [16]. Our work differs in being focused on the query predicate expression containment issues and dealing with dynamically maintaining such templates in the presence of addition and deletions of queries in the cache.

8 Summary

Dynamic caching of application query results in a predicate cache promises to be an adaptive solution with low administrative overhead. In recent work, we have implemented DBProxy, a prototype of a caching system which maintains previous query results and answers new queries from the cache whenever it can prove the containment of the new query in the cached set. In this paper, we discussed the scalability challenges of query containment in large predicate caches and proposed algorithms for fast containment checking in such an environment. We exploit the template-based nature of queries generated by application programs to accelerate the containment test. We propose algorithms to infer templates by detecting similarity between query predicates and for proving containment among similar query predicates. We described the Merged Aggregate Predicate mechanism to dynamically aggregate the ranges of similar predicate terms for improving the efficiency of the template-matching containment algorithm. The template-matching and general containment checking algorithms can be combined in a two-level scheme for obtaining high performance as well as maximal hit rates. We conducted experiments using our DBProxy caching testbed against the TPC-W and the Trade-2 benchmarks. Our results show that the template-based scheme provides an order of magnitude improvement in query containment checking and significant improvement in overall query response times.

References

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

[2] K. Amiri, R. Tewari, S. Park, and S. Padmanabhan. On Space Management in a Dynamic Edge Data Cache. In *Proceedings of WebDB*, 2002.

[3] C. H. Bell (VP, Amazon.com). Customer Experience in a Merchandising Platform, June 2002. SIGMOD Conference keynote address.

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

[5] 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.

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

[7] S. Gribble, A. Halevy, Z. Ives, M. Rodrig, and D. Suciu. What can peer-to-peer do for databases, and vice versa? In *Fourth International Workshop on Web and Databases*, May 2001.

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

[9] A. Halevy. Answering Queries Using Views: A Survey. *VLDB Journal*, 16(3), 2001.

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

[11] IBM. WebSphere Commerce Suite. http://www-4.ibm.com/software/webservers/commerce/wcs51.html.

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

[13] 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.

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

[15] 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.

[16] R. Pottinger and A. Halevy. A Scalable Algorithm for Answering Queries Using Views. In *Proceedings of the International Conference on Very Large Databases (VLDB)*, 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.

[19] D. Srivastava, S. Dar, H. V. Jagadish, and A. Levy. Answering SQL Queries Using Materialized Views. In *VLDB Conference*, 1996.

[20] The Transaction Processing Performance Council. TPC-W benchmark. http://www.tpc.org/tpcw/default.asp.