Towards Intelligent Semantic Caching for Web Sources *

Dongwon Lee and Wesley W. Chu ({dongwon,wwc}@cs.ucla.edu)
Dept. of Computer Science, University of California, Los Angeles, CA 90095, USA

Abstract. An intelligent semantic caching scheme suitable for web sources is presented. Since web sources typically have weaker querying capabilities than conventional databases, existing semantic caching schemes cannot be directly applied. Our proposal takes care of the difference between the query capabilities of an end user system and web sources. In addition, an analysis on the match types between a user's input query and cached queries is presented. Based on this analysis, we present an algorithm that finds the best matched query under different circumstances. Furthermore, a method to use semantic knowledge, acquired from the data, to avoid unnecessary access to web sources by transforming the cache miss to the cache hit is presented. To verify the effectiveness of the proposed semantic caching scheme, we first show how to generate synthetic queries exhibiting different levels of semantic localities. Then, using the test sets, we show that the proposed query matching technique is an efficient and effective way for semantic caching in web databases.

Keywords: Semantic Caching, Web Database, Query Matching, Semantic Locality

1. Introduction

Web databases allow users to pose queries to distributed and heterogeneous web sources. Such systems usually consist of three components (Adali et al., 1996; García-Molina et al., 1995): 1) mediators to provide a distributed, heterogeneous data integration, 2) wrappers to provide a local translation and extraction, and 3) web sources containing raw data to be queried and extracted. In the *virtual* approach (Florescu et al., 1998), the queries are posed to a uniform interface and submitted to multiple sources at runtime. Such querying can be very costly due to run-time costs. An effective way to reduce costs in such an environment is to cache the results of prior queries and to reuse them (Alonso et al., 1990; Franklin et al., 1993).

Let us first consider a motivating example for semantic caching.

Example 1: Two queries, $Q1$ and $Q2$, are asked against a relation `emp(name,age,title,gender)` using Datalog notation of Zaniolo et al. (1997) and saved in the cache as follows:

```
Q1: male(name) <- emp(name,_,_,'m').
Q2: 50s_mngr(name) <-
```

* This research is supported in part by DARPA contract No. N66001-97-C-8601 and SBIR F30602-99-C-0106.

```
emp(name,age,'manager',_), 50<=age<60.
```

When a new query $Q3$ asking "male manager's name in his fifties" is given, it can be evaluated against either the `emp` relation or the stored queries $Q1$ and $Q2$ in the cache as follows:

```
E1: 50s_male_manager(name) <-
        emp(name,age,'manager','m'), 50<=age<60.
E2: 50s_male_manager(name) <- male(name), 50s_mngr(name).
```

Both evaluations yield identical results. However, when the `emp` relation is stored remotely or temporarily unavailable due to network partition, using the evaluation $E2$ against the stored queries is much more efficient. □

Semantic caching (Dar et al., 1996; Keller and Basu, 1996; Larson and Yang, 1985; Ren and Dunham, 1998; Sellis, 1988) exploits the semantic locality of the queries by caching a set of semantically associated results, instead of tuples or pages which are used in conventional caching. Semantic caching can be particularly effective in improving performance when a series of semantically associated queries are asked if the results may likely overlap or contain one another. Applications, such as the cooperative database system (Chu et al., 1994) and geographical information system, are the examples.

So far most semantic caching schemes in client-server architectures are based on the assumption that all participating components are full-fledged database systems. If a client sends a query $\mathcal{A}$ but its cache contains answers for $\mathcal{A} \wedge \mathcal{B}$, then the client has to send a modified query $\mathcal{A} \wedge \neg\mathcal{B}$ to the server to retrieve the remaining answers. In web databases, however, web sources such as plain web pages or form-based IR systems have very limited querying capabilities and cannot easily support such complicated (e.g., negation) queries.

Our proposed semantic caching scheme is based upon the following three key ideas:

1. Since querying capabilities of web sources are weaker than those of queries from end users, query translation and capability mapping are necessary in semantic caching.

2. With an efficient method to locate the best matched query from the set of candidates, semantic caching for web sources can significantly improve system performance.

3. Semantic knowledge can be used to transform a *cache miss* in a conventional caching to a *cache hit*.

The rest of the paper is organized as follows. In Section 2, we introduce background information and related work for semantic caching. In Section 3, we describe our proposed intelligent semantic caching in detail. In Section 4, query matching technique is presented. Then, experimental results follow in Section 5. Finally, concluding remarks are discussed in Section 6.

2. Background

2.1. Preliminaries

Our caching scheme is implemented in a web database test-bed called CoWeb (Cooperative Web Database) at UCLA. The architecture consists of a network of mediator and wrapper components (Adali et al., 1996; García-Molina et al., 1995). The focus of the system is to use knowledge for providing cooperative capabilities such as conceptual and approximate web query answering, knowledge-based semantic caching, and web triggering with fuzzy threshold conditions. The input query is expressed in the SQL[1] language based on the mediator schema. The mediator decomposes the input SQL into sub-queries for the wrappers by converting the `WHERE` clause into disjunctive normal form, $\mathcal{DNF}$ (the logical OR of the logical AND clauses), and dis-joining conjunctive predicates. CoWeb handles *selection* and *join* predicates with any of the following operators $\{>, \geq, <, \leq, =\}$.

Our semantic caching approach is closely related to the *query satisfiability* and *query containment* problems (Guo et al., 1996; Ullman, 1988). Given a database $\mathcal{D}$ and query $\mathcal{Q}$, applying $\mathcal{Q}$ on $\mathcal{D}$ is denoted as $\mathcal{Q}(\mathcal{D})$. Then, $\langle \mathcal{Q}(\mathcal{D}) \rangle$, or $\langle \mathcal{Q} \rangle$ for short, is the n-ary relation obtained by evaluating the query $\mathcal{Q}$ on $\mathcal{D}$. Given two n-ary queries, $\mathcal{Q}_1$ and $\mathcal{Q}_2$, if $\langle \mathcal{Q}_1(\mathcal{D}) \rangle \subset \langle \mathcal{Q}_2(\mathcal{D}) \rangle$ for any database $\mathcal{D}$, then the query $\mathcal{Q}_1$ is *contained* in the query $\mathcal{Q}_2$, that is $\mathcal{Q}_1 \subseteq \mathcal{Q}_2$. If two queries *contain* each other, they are *equivalent*, that is $\mathcal{Q}_1 \equiv \mathcal{Q}_2$.

The solutions to both query satisfiability and containment problems vary depending on the exact form of the predicate. If a conjunctive query has only selection predicates with the five operators $\{>, \geq, <, \leq, =\}$, the query satisfiability problem can be solved in $O(|\mathcal{Q}|)$ time for the query $\mathcal{Q}$ (Guo et al., 1996). On the other hand, the conjunctive query containment problem is shown NP-complete (Chandra and Merlin, 1977) although in the common case where no predicate appears

[1] Current CoWeb implementation supports only SPJ (Select-Project-Join) type SQL.

more than twice, there appears to be a linear-time algorithm (Saraiya, 1991; Ullman, 1997).

2.2. Related Work

Past research areas related to semantic caching include conventional caching (Alonso et al., 1990; Franklin et al., 1993), query satisfiability and containment problems (Guo et al., 1996; Ullman, 1988), view materialization (Levy et al., 1995; Larson and Yang, 1985), query folding (Qian, 1996), and semantic query optimization (Chu et al., 1994). Recently, semantic caching in a client-server or multi-database architecture has received attention (Ashish et al., 1998; Dar et al., 1996; Godfrey and Gryz, 1999; Keller and Basu, 1996; Ren and Dunham, 1998; Chidlovskii and Borghoff, 2000). Deciding whether a query is answerable or not is closely related to the problem of finding complete rewritings of a query using views (Levy et al., 1995; Qian, 1996). The main difference is that semantic caching techniques evaluate the given query against the semantic views, while query rewriting techniques rewrite a given query based on the views (Cluet et al., 1999). Further, our proposed technique is also more suitable for web databases where the querying capability of the sources is not compatible with that of the clients. In such settings, it is generally impossible to get a snapshot of the given views to materialize them since query interfaces simply do not allow it.

Semantic caching and the corresponding indexing techniques which require that the cached results be exactly matched with the input query are presented in Sellis (1988). In our approach, the cached results do not have to be exactly matched with the input query in order to compute answers. Chen and Roussopoulos (1994) approaches semantic caching from the query planning and optimization point of view. Dar et al. (1996) maintains cache space by coalescing or splitting the semantic regions, while we maintain cache space by reference counters to allow overlapping in the semantic regions. Further, we provide techniques to find the best matched query under different circumstances via extended and knowledge-based matching. In Keller and Basu (1996), *predicate descriptions* derived from previous queries are used to match an input query with the emphasis on updates in the client-server environment. In Chidlovskii and Borghoff (2000), a semantic caching scheme for conjunctive keyword-based web queries is introduced. Here, to quickly process a comparison of an input query against the semantic views, binary signature method is used. Issues such as probe vs. remainder query and region coalescing that were originally dealt in Dar et al. (1996) are further explored with real life experiments.

In Ashish et al. (1998), selectively chosen sub-queries are stored in the cache and are treated as information sources in the domain model. To minimize the expensive cost for containment checking, the number of semantic regions is reduced by merging them whenever possible. Ren and Dunham (1998) defines a semantic caching formally and addresses query processing techniques derived from Larson and Yang (1985). A comprehensive formal framework for semantic caching is introduced in Godfrey and Gryz (1999) illustrating issues, such as when answers are in the cache, when answers in the cache can be recovered, etc. Adali et al. (1996) discusses semantic caching in the mediator environment with knowledge called *invariants*. Although the invariants are powerful tools, due to their support of arbitrary user-defined functions as conditions, they are mainly used for substituting a domain call. On the contrary, we propose a simpler and easier way (i.e., $\mathcal{FIND}$) to express a fragment containment relationship on relations that can be acquired (semi)-automatically.

3. Semantic Caching Technique

3.1. Semantic Caching Model

A semantic cache is essentially a hash table where an entry consists of a *(key, value)* pair. The key is the semantic description based on the previous queries. The value is a set of answers that satisfy the key. The semantic description made of a prior query is denoted as *semantic view*, $\mathcal{V}$. Then, an entry in the semantic cache is denoted as $(\mathcal{V}, \langle\mathcal{V}\rangle)$ using the notation $\langle\mathcal{V}\rangle$ in Section 2.

To represent a submitted SQL query in a cache, we need: 1) relation names, 2) attributes used in the `WHERE` clause, 3) projected attributes, and 4) conditions in the `WHERE` clause (Larson and Yang, 1985; Ren and Dunham, 1998). For semantic caching in CoWeb, we use only "conditions in the `WHERE` clause" for the following reasons. In our settings, since there is one wrapper covering one web source, and thus 1-to-1 mapping between the wrapper and the web source, the relation names are not needed. In addition, the majority of the web sources has a fixed output page format from which the wrapper (i.e., extractor) extracts the specified data. That is, whether or not the input SQL query wants to project some attributes, the output web page that the wrapper receives always contains a set of pre-defined attribute values. Since retrieval cost is the dominating factor in web databases, CoWeb chooses to store all attribute values contained in the output web page in the cache. Thus the attributes used in conditions and the projected attributes do not need to be stored.

Furthermore, by storing all attribute values in the cache, CoWeb can completely avoid the *un-recoverability problem*, which can occur when query results cannot be recovered from the cache even if they are found in the cache, due to the lack of certain logical information (Godfrey and Gryz, 1999). As a result, queries stored in the semantic cache of the CoWeb have the form "`SELECT * FROM web_source WHERE condition`", where the `WHERE` condition is a conjunctive predicate. Hereafter, user queries are represented by the conditions in the `WHERE` clause.

3.2. Query Naturalization

Different web sources use different ontology. Due to security or performance concerns (Florescu et al., 1998), web sources often provide different query processing capabilities. Therefore, wrappers need to perform the following pre-processing of an input query before submitting it to the web source:

1. Translation: To provide a 1-to-1 mapping between the wrapper and the web source, the wrapper needs to schematically *translate* the input query.

2. Generalization & **Filtration**: If there is no 1-to-1 mapping between the wrapper and the web source, the wrapper can *generalize* the input query to return more results than requested and filter out the extra data. For instance, a predicate `(name='tom')` can be generalized into the predicate `(name LIKE '%tom%')` with an additional filter `(name='tom')`.

3. Specialization: When there is no 1-to-1 mapping between the wrapper and the web source, the wrapper can *specialize* the input query with multiple sub-queries and then merge the results. For instance, a predicate `(1998<year<2001)` can be specialized to a disjunctive predicate `(year=1999` $\vee$ `year=2000)` provided that `year` is an integer type.

The original query from the mediator is called ***input query***. The generated query after pre-processing the input query is called ***native query***, as it is supported by the web source in a native manner (Chang et al., 1996). Such pre-processing is called *query naturalization*. The query used to filter out irrelevant data from the native query results is called *filter query* (Chang et al., 1996). When the translation is not applicable due to the lack of 1-to-1 mapping, CoWeb applies generalization or specialization based on the knowledge regarding the querying capability of the web source. This information is pre-determined by a domain expert such as wrapper developer. CoWeb carries a *capability*

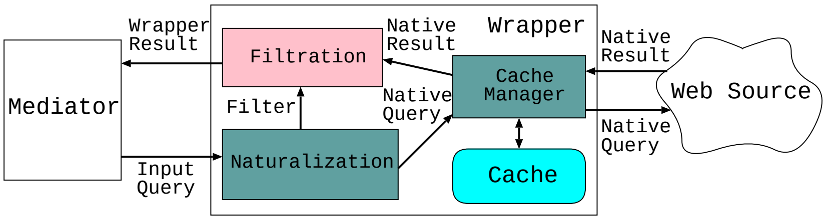


Figure 1. The control flow among the mediator, wrapper, and web source. An input query from the mediator is naturalized in the wrapper and converted to a native query. A filter query can be generated if needed. The cache manager then checks the native query against the semantic views stored in the cache to find a match. If a match is found but no filter query was generated for the query, results are retrieved from the cache and returned to the mediator. If there was a filter query generated, then the results need to be filtered to remove the extra data. If no match is found, the native query is submitted to the web source. After obtaining native results from the web source, the wrapper performs post-processing and returns the final results to the mediator. Finally the proper form of the native query (e.g., disjunctive predicates are broken into conjunctive ones) is saved in the cache for future use.

description vector (CDV), a 5-tuple vector, to describe the querying capability of the web source. For each attribute of the web source, the associated 5-tuple vector carries: 1) `in`: describes whether the web source must be given binding for this attribute or not. It can have two values – `man` for mandatory, `opt` for optional. 2) `out`: describes whether this attribute will be shown in the results or not. It has the same values as `in`. 3) `op`: contains operators being supported by the attribute. 4) `any`: contains a string value to be used as a wild card. 5) `domain`: represents the complete domain values of the attribute. Currently three types – `list, interval`, and `set` – are supported.

The expressive power of the CDV is less than that of the Vassalos and Papakonstantinou (1997), but equivalent to that of the Levy et al. (1996). Unlike Levy et al. (1996) where query capabilities are described for a whole web source, each attribute in CoWeb carries its own description. Figure 1 illustrates the control flow among the mediator, wrapper and web source in detail.

Example 2: Imagine a web source that supports queries on the relation `employee(name,age,title)` with only "=" operator. Then, an input query $\mathcal{Q}$:`(20`$\leq$`age`$\leq$`22` $\wedge$ `title='manager')` needs to be naturalized (i.e., specialized) into a native query $\mathcal{V}$:`((age=20` $\wedge$ `title='manager')` $\vee$ `(age=21` $\wedge$ `title='manager')` $\vee$ `(age=22` $\wedge$ `title='manager'))`. Further, since semantic views use only conjunctive predicates, the native query $\mathcal{V}$ is partitioned into three conjunctive parts, $\mathcal{V}_1$:`(age=20` $\wedge$ `title='manager')`, $\mathcal{V}_2$:`(age=21` $\wedge$ `title='manager')`, and $\mathcal{V}_3$:`(age=22`

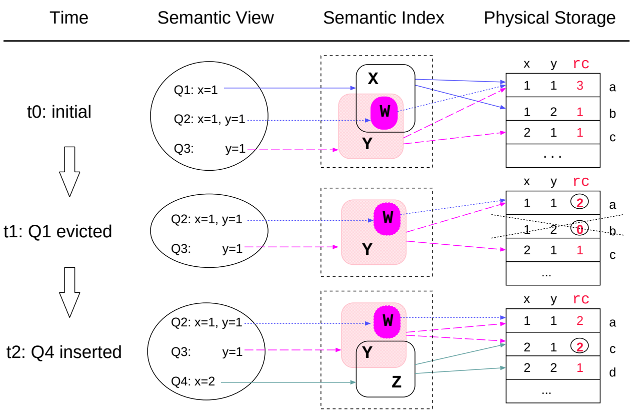


Figure 2. A cache replacement example. When Q_1 is evicted at time $t1$, the corresponding reference counters are decremented. The tuple `b` is deleted since its reference counter is 0, but the tuple `a` and `c` remain in the physical storage. When Q_4 is inserted at time $t2$, new tuple tuple `d` is inserted to the physical storage and the reference counter of the tuple `c` is increased.

$\wedge$ `title='manager')`. Thus, three entries ($\mathcal{V}_1$, $\langle\mathcal{V}_1\rangle$), ($\mathcal{V}_2$, $\langle\mathcal{V}_2\rangle$), and ($\mathcal{V}_3$, $\langle\mathcal{V}_3\rangle$) are inserted as semantic views into the cache. □

3.3. Semantic View Overlapping

A semantic view creates a spatial object[2] in an n-dimensional hyperspace, which creates overlapping. For instance, two queries ($10 \leq$ `age` $\leq 20 \wedge 30k \leq$ `sal` $\leq 40k$) and ($15 \leq$ `age` $\leq 25 \wedge 35k \leq$ `sal` $\leq 45k$) create an overlapping ($15 \leq$ `age` $\leq 20 \wedge 35k \leq$ `sal` $\leq 40k$). Since excessive overlapping of the semantic views may waste the cache space for duplicate answers, the overlapped portions can be coalesced to the new semantic views and the remaining semantic views are modified appropriately or can be completely separate semantic views. For details, refer to Lee and Chu (1999). In CoWeb, unlike these approaches, the overlapping of the semantic views is allowed to retain the original form of the semantic views. By using a *reference counter* to keep track of the references of the answer tuples in implementing the cache, the problem of storing redundant answers in the cache is avoided (Keller and Basu, 1996).

3.4. Cache Replacement Policy

According to pre-determined evaluation functions (e.g., LRU, semantic distance), the corresponding replacement values (e.g., access order, dis-

[2] This is called a *semantic region* in Dar et al. (1996) and a *semantic segment* in Ren and Dunham (1998).

Table I. Query match types and their properties. $\mathcal{V}$ is a semantic view and $\mathcal{Q}$ is a user query.

Match Types	Properties	Answers from	
		Cache	Web source
Exact match	$\mathcal{V} \equiv \mathcal{Q}$	$\langle \mathcal{V} \rangle$	$\emptyset$
Containing match	$\mathcal{V} \not\subseteq \mathcal{Q} \wedge \mathcal{Q} \subseteq \mathcal{V}$	$\langle \mathcal{Q}(\langle \mathcal{V} \rangle) \rangle$	$\emptyset$
Contained match	$\mathcal{V} \subseteq \mathcal{Q} \wedge \mathcal{Q} \not\subseteq \mathcal{V}$	$\langle \mathcal{V} \rangle$	$\langle \mathcal{Q} \wedge \neg V \rangle$
Overlapping match	$\mathcal{V} \not\subseteq \mathcal{Q} \wedge \mathcal{Q} \not\subseteq \mathcal{V}$	$\langle \mathcal{Q}(\langle \mathcal{V} \rangle) \rangle$	$\langle \mathcal{Q} \wedge \neg \mathcal{V} \rangle$
Disjoint match	$\mathcal{Q} \wedge \mathcal{V}$ is *unsatisfiable*	$\emptyset$	$\langle \mathcal{Q} \rangle$

tance value) are computed and added to the semantic view. Individual tuples stored in the physical storage contain a reference counter to keep track of the number of references. After the semantic view for replacement has been decided, all tuples belonging to the semantic view are found via the semantic index and their reference counters are decremented by 1. The tuples with counter value 0 are removed from the physical storage. The corresponding semantic view and semantic index are then removed from the cache entries. An example is illustrated in Figure 3.4. Note that the objects in the semantic index can be overlapped, but not in the physical storage. Also, there is no coalease among overlapping or containing semantic indices.

3.5. Match Types

When a query is compared to a semantic view, there can be five different match types. Consider a semantic view $\mathcal{V}$ in the cache and a user query $\mathcal{Q}$. When $\mathcal{V}$ is *equivalent* to $\mathcal{Q}$, $\mathcal{V}$ is an **exact match** of $\mathcal{Q}$. When $\mathcal{V}$ *contains* $\mathcal{Q}$, $\mathcal{V}$ is a **containing match** of $\mathcal{Q}$. In contrast, when $\mathcal{V}$ is *contained* in $\mathcal{Q}$, $\mathcal{V}$ is a **contained match** of $\mathcal{Q}$. When $\mathcal{V}$ does not contain, but intersects with $\mathcal{Q}$, $\mathcal{V}$ is an **overlapping match** of $\mathcal{Q}$. Finally, when there is no intersection between $\mathcal{Q}$ and $\mathcal{V}$, $\mathcal{V}$ is a **disjoint match** of $\mathcal{Q}$. The exact match and containing match are **complete matches** since all answers are in the cache, while the overlapping and contained match are **partial matches** since some answers need to be retrieved from the web sources. The detailed properties of each match type are shown in Table I. Note that for the contained and overlapping matches, computing answers requires the union of the partial answers from the cache and from the web source.

The `MatchType(`$\mathcal{Q}$`,`$\mathcal{V}$`)` algorithm then can be derived from Table I in a straightforward manner. Using algorithms developed for solving

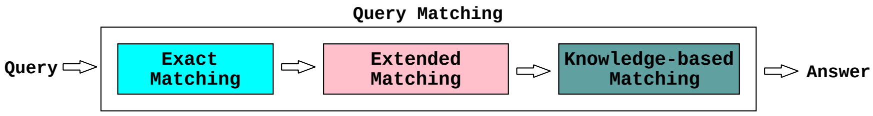


Figure 3. The flow in the query matching technique.

the satisfiability and containment problems (Saraiya, 1991; Guo et al., 1996; Ullman, 1997), the `MatchType` algorithm can be implemented in $O(|\mathcal{Q}| + |\mathcal{V}|)$ complexity (limited containment case when no predicate appears more than twice).

4. Query Matching Technique

Let us now discuss the process of finding the best matched query from the semantic views, called **query matching**, which consists of three steps: *exact*, *extended*, and *knowledge-based matching*, as depicted in Figure 3.

4.1. Exact Matching

Traditional caching considers only *exact matches* between input queries and semantic views. If there is a semantic view that is identical to the input query, then it is a cache hit. Otherwise, it is a cache miss.

4.2. Extended Matching

Extended matching extends the exact matching for those cases where an input query is not exactly matched with a semantic view. Other than the exact matching, the containing match is the next best case since it only contains some extra answers. Then, between the contained and overlapping matches, the contained match is slightly better. This is because the contained match does not contain extra answers in the cache, although both have only partial answers (see Table I). Note that for an input query, there can be many containing or contained matches. In the following subsections, we present how to find the best match among the different candidates in a cache.

4.2.1. *The* `BestContainingMatch` *&* `BestContainedMatch` *Algorithms*

Intuitively, we want to find the most specific semantic view which would incur the least overhead cost to answer the user's query (i.e., the smallest superset of the input query). Without loss of generality,

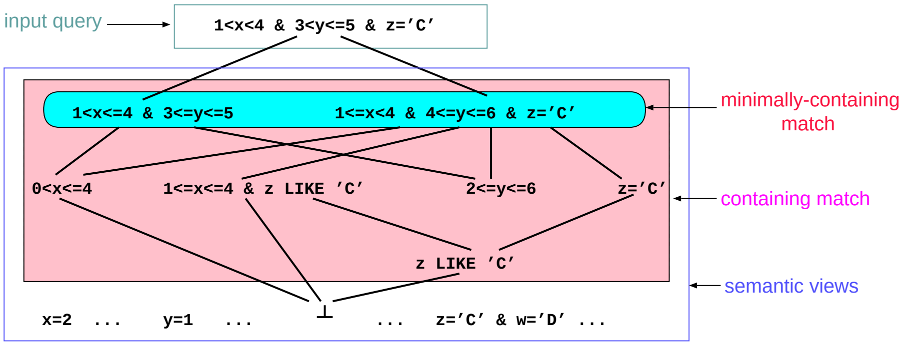


Figure 4. Example query containment lattice. Given the input query, there are seven containing matches. Among them, two are the minimally-containing matches. That is, these two semantic views are the smallest superset of the given input query.

we discuss the case of the containing match only (the contained match case can be defined similarly). We first define the query containment lattice.

Definition 3 (Query Containment Lattice)
Suppose $\mathcal{Q}$ is a query and the set $\mathcal{U}_Q$ corresponds to the set of all the *containing/contained matches* of the $\mathcal{Q}$ found in the cache. Then, the *query containment lattice* is defined to be a partially ordered set $\langle \mathcal{Q}, \subseteq \rangle$ where the ordering $\subseteq$ forms a lattice over the set $\mathcal{U}_Q \cup \{\perp\}$. For a containing match case, the greatest lower bound (glb) of the lattice is the special symbol $\perp$ and the least upper bound (lub) of the lattice is the query $\mathcal{Q}$ itself. For contained match case, the least upper bound (lub) of the lattice is the special symbol $\perp$ and the greatest lower bound (glb) of the lattice is the query $\mathcal{Q}$ itself. ■

Definition 4 (Minimality and Maximality)
A containing match of the $\mathcal{Q}$, $\mathcal{A}$, is called ***minimally-containing match*** of the $\mathcal{Q}$ and denoted by $\mathcal{M}(\mathcal{C}; \mathcal{Q})_{min}$ if and only if there is no other containing match of the $\mathcal{Q}$, $\mathcal{B}$, such that $\mathcal{A} \subseteq \mathcal{B} \subseteq \mathcal{Q}$ in $\mathcal{Q}$'s query containment lattice. Symmetrically, a contained match of the $\mathcal{Q}$, $\mathcal{A}$, is called ***maximally-contained match*** of the $\mathcal{Q}$ and denoted by $\mathcal{M}(\neg\mathcal{C}; \mathcal{Q})_{max}$ if and only if there is no other contained match of the $\mathcal{Q}$, $\mathcal{B}$, such that $\mathcal{A} \supseteq \mathcal{B} \supseteq \mathcal{Q}$ in $\mathcal{Q}$'s query containment lattice. ■

An example of the query containment lattice is shown in Figure 4. Note that for a given query $\mathcal{Q}$, there can be several minimally-containing matches found in the cache as illustrated in Figure 4. In such cases, the *best* minimally-containing match can be selected based on such heuristics as the number of answers associated with the semantic view, the

Input: $\{\mathcal{V}_1, ..., \mathcal{V}_k\}$; **Output**: Best $\leftarrow \mathcal{V}_i \in \{\mathcal{V}_1, ..., \mathcal{V}_k\}$;

Best $\leftarrow \emptyset$, Bucket$_{containing} \leftarrow \{\mathcal{V}_1, ..., \mathcal{V}_k\}$;
for $\mathcal{V}_i \leftarrow \mathcal{V}_1$ to $\mathcal{V}_k$ **do**
  for $\mathcal{V}_j \leftarrow \mathcal{V}_1$ to $\mathcal{V}_k$; $i \neq j$ **do**
    if `MatchType`($\mathcal{V}_i$, $\mathcal{V}_j$) = `containing_match` **then**
      Bucket$_{containing} \leftarrow$ Bucket$_{containing} - \mathcal{V}_j$;
for $\mathcal{V}_i \in$ Bucket$_{containing}$ **do**
  Best $\leftarrow$ pick one heuristically from Bucket$_{containing}$;
return Best;

Figure 5. The `BestContainingMatch` algorithm.

number of predicate literals in the query, etc. The `BestContainingMatch` algorithm is shown in Figure 5. It first finds the minimally-containing matches using the containment lattice and if there are several minimally-containing matches, then pick one heuristically.

With $|\mathcal{V}_{max}|$ being the length of the longest containing match and k being the number of containing matches, the running time of the `BestContainingMatch` algorithm becomes $O(k^2|\mathcal{V}_{max}|)$ without any indexing on the semantic views. Observe that the `BestContainingMatch` algorithm is only justified when finding the best containing match is better than selecting an arbitrary containing match followed by filtering. This occurs often in web databases with a large number of heterogeneous web sources or in multi-media databases with expensive operations for image processing. The `BestContainedMatch` algorithm is similar to the case of the `BestContainingMatch` algorithm.

4.2.2. *The* `BestOverlappingMatch` *Algorithm*

For the overlapping matches, we cannot construct the query containment lattice. Thus, in choosing the best overlapping match, we use a simple heuristic: choose the overlapping match which overlaps *most* with the given query. There are many ways to determine the meaning of overlapping. One technique is to compute the overlapped region between two queries in n-dimensional spaces or compare the number of associated answers and select the one with maximum answers.

4.3. Knowledge-based Matching

According to our experiments in Section 5, partial matches (i.e., overlapping and contained matches) constitute about 40% of all match types for the given test sets (see Table IV). Interestingly, a partial match can be a complete match in certain cases. For instance, for the `employee` relation, a semantic view $\mathcal{V}$:`(gender='m')` is the overlapping match of a query $\mathcal{Q}$:`(name='john')`. If we know that `john` is in fact a male employee, then $\mathcal{V}$ is a containing match of $\mathcal{Q}$ since $\mathcal{Q} \subseteq \mathcal{V}$. Since complete matches (i.e., exact and containing matches) eliminate the need to access the web source, transforming a partial match into a complete match can improve the performance significantly.

Obtaining semantic knowledge from the web source and maintaining it properly are important issues. In general, such knowledge can be obtained by human experts from the application domain. In addition, database constraints, such as inclusion dependencies, can be used. Knowledge discovery and data mining techniques are useful in obtaining such knowledge (semi-)automatically. For instance, the association rules tell if the antecedent in the rule is satisfied, then the consequent of the rule is likely to be satisfied with certain confidence and support.

How to manage the obtained knowledge under addition, deletion, or implications is also an important issue. Since the focus of this paper is to show how to utilize such knowledge for semantic caching, the knowledge acquisition and management issues are beyond the scope of this paper. We assume that the semantic knowledge that we are in need of was already acquired and was available to the cache manager. We use a generic notation derived from Chu et al. (1994) to denote the containment relationship between two fragments of relations.

Definition 5 (Fragment Inclusion Dependency)
A *fragment inclusion dependency* ($\mathcal{FIND}$) says that values in columns of one fragment must also appear as values in columns of other fragment. Formally, $\pi_{<A_1,\ldots,A_k>}(\sigma_{\mathcal{P}}(\mathcal{R})) \diamond \pi_{<B_1,\ldots,B_k>}(\sigma_{\mathcal{Q}}(\mathcal{S}))$, where $\diamond \in \{\equiv, \subseteq\}$, $\mathcal{P}$ and $\mathcal{Q}$ are valid `SELECT` conditions, $\mathcal{R}$ and $\mathcal{S}$ are valid relations, and A_i and B_i are attributes compatible each other. Often LHS or RHS is used to denote the left or right hand side of the $\mathcal{FIND}$. A set of $\mathcal{FIND}$ is denoted by Δ and assumed to be closed under its consequences (i.e., $\Delta = \Delta^*$). ■

Definition 6 (Query $\triangle$-Containment)
Given two n-ary queries, $\mathcal{Q}_1$ and $\mathcal{Q}_2$, if $\langle \mathcal{Q}_1(\mathcal{D}) \rangle \subset \langle \mathcal{Q}_{\in}(\mathcal{D}) \rangle$ for an arbitrary relation $\mathcal{D}$ obeying the fragment inclusion dependencies, then the query $\mathcal{Q}_1$ is $\triangle$-*contained* in the query $\mathcal{Q}_2$ and denoted by $\mathcal{Q}_1 \subseteq_\triangle \mathcal{Q}_2$. If two queries $\triangle$-*contain* each other, they are $\triangle$-*equivalent* and denoted by $\mathcal{Q}_1 \equiv_\triangle \mathcal{Q}_2$ (Gryz, 1998). ■

Now using $\mathcal{FIND}$ framework, we can easily denote the various semantic knowledges. For instance, let's consider the classical inclusion dependency. Inclusion dependency is a formal statement of the form $\mathcal{R}[\mathcal{X}] \subseteq \mathcal{S}[\mathcal{Y}]$, where $\mathcal{R}$ and $\mathcal{S}$ are relation names, $\mathcal{X}$ is an ordered list of attributes of $\mathcal{R}$, $\mathcal{Y}$ is an ordered list of attributes of $\mathcal{S}$ of the same length as $\mathcal{X}$ (Johnson and Klug, 1984). For instance, the inclusion dependency "every manager is also an employee" can be denoted as $\pi_{<manager\_id>}(\sigma_*(\texttt{manager\_table})) \subseteq \pi_{<employee\_id>}(\sigma_*(\texttt{employee\_table}))$ in $\mathcal{FIND}$, where σ_* means selecting every tuples in the relation. In addition, $\mathcal{FIND}$ can easily capture association rules found via data mining techniques.

4.3.1. *Transforming Partial Matches to Complete Matches*

Our goal is to transform as many partial matches (i.e., overlapping and contained matches) to complete matches (i.e., exact and containing matches) as possible with the given $\mathcal{FIND}$ set Δ. The overlapping match can be transformed into four other match types, while the contained match can only be transformed into the exact match.

1. Overlapping Match: Given a query $\mathcal{Q}$, its *overlapping match* $\mathcal{V}$ and a dependency set Δ,

- If $\{LHS \equiv RHS\} \in \Delta, \mathcal{Q} \equiv LHS, \mathcal{V} \equiv RHS$, then $\mathcal{V}$ is the *exact match* of the $\mathcal{Q}$.
- If $\{LHS \subseteq RHS\} \in \Delta, \mathcal{Q} \subseteq LHS, RHS \subseteq \mathcal{V}$, then $\mathcal{V}$ is the *containing match* of the $\mathcal{Q}$.
- If $\{LHS \subseteq RHS\} \in \Delta, \mathcal{V} \subseteq LHS, RHS \subseteq \mathcal{Q}$, then $\mathcal{V}$ is the *contained match* of the $\mathcal{Q}$.
- If $\{LHS \subseteq RHS\} \in \Delta, \mathcal{Q} \subseteq LHS, \mathcal{V} \wedge RHS$ is *unsatisfiable*, or $\{LHS \subseteq RHS\} \in \Delta, \mathcal{V} \subseteq RHS, \mathcal{Q} \wedge LHS$ is *unsatisfiable*, then $\mathcal{V}$ is the *disjoint match* of the $\mathcal{Q}$.

Proof: Here, we only show the proof for the second case of the overlapping match transformation. Others follow similarly as well. For the overlapping match, from Table I, we have $\mathcal{Q} \not\subseteq \mathcal{V} \wedge \mathcal{V} \not\subseteq \mathcal{Q}$. If the condition part is satisfied, then we have $\mathcal{Q} \subseteq LHS \subseteq RHS \subseteq \mathcal{V}$, thus $\mathcal{Q} \subseteq \mathcal{V}$ since $\subseteq$ is a transitive operator. This overwrites the original property $\mathcal{Q} \not\subseteq \mathcal{V}$. As a result, we end up with a property $\mathcal{Q} \subseteq \mathcal{V} \wedge \mathcal{V} \not\subseteq \mathcal{Q}$, which is the property of the containing match. (q.e.d)

2. Contained Match: Given a query $\mathcal{Q}$, its *contained match* $\mathcal{V}$ and a Δ, if $\{LHS \equiv RHS\} \in \Delta, \mathcal{Q} \subseteq LHS, \mathcal{V} \subseteq RHS$, then $\mathcal{V}$ is the *exact match* of $\mathcal{Q}$.

Example 7: Suppose we have a query $\mathcal{Q}$:`(salary=100k)` and a semantic view $\mathcal{V}$:`(title='manager')`. Given a Δ: $\{\sigma_{80k \leq salary \leq 120k} \subseteq \sigma_{title='manager' \wedge age \geq 40}\}$, $\mathcal{V}$ becomes a containing match of $\mathcal{Q}$ since $\mathcal{Q} \subseteq LHS, RHS \subseteq \mathcal{V}$ and $\{LHS \subseteq RHS\} \in \Delta$. □

4.3.2. *The* Δ-`MatchType` *Algorithm*

Let us first define an augmented `MatchType` algorithm in the presence of the dependency set Δ. The Δ-`MatchType` algorithm can be implemented by modifying the `MatchType` algorithm in Section 3.5 by adding additional input, Δ, and changing all $\equiv$ to $\equiv_\Delta$ and $\subseteq$ to $\subseteq_\Delta$. The computational complexity of $\mathcal{Q} \equiv_{\Delta'} \mathcal{V}$ where Δ' contains the single $\mathcal{FIND}$ = LHS $\diamond$ RHS is then $O(|\mathcal{Q}| + |\mathcal{V}| + |LHS| + |RHS|)$. Let $|\mathcal{L}_{max}|$ and $|\mathcal{R}_{max}|$ denote the length of the longest LHS and RHS in Δ and let $|\Delta|$ denote the number of $\mathcal{FIND}$ in Δ, then the total computational complexity of the Δ-`MatchType` algorithm is $O(|\Delta|(|\mathcal{Q}|+|\mathcal{V}|+|\mathcal{L}_{max}|+|\mathcal{R}_{max}|))$ in the worst case when all semantic views in the cache are either overlapping or contained matches. Since the gain from transforming partial matches to complete matches is I/O-bounded and the typical length of the conjunctive query is relatively short, it is a good performance trade-off to pay overhead cost for the CPU-bounded Δ-`MatchType` algorithm in many applications.

4.4. The `QueryMatching` Algorithm: Putting It All Together

The `QueryMatching` algorithm shown in Figure 6 finds the best semantic view in the cache for a given input query in the order of the exact match, containing match, contained match and overlapping match. If all semantic views turn out to be disjoint matches, it returns a null answer. It takes into account not only exact containment relationship but also extended and knowledge-based containment relationships. Let $|\mathcal{V}_{max}|$ denote the length of the longest semantic views. Then the `for` loop takes at most $O(k|\Delta|(|\mathcal{Q}| + |\mathcal{V}_{max}| + |\mathcal{L}_{max}| + |\mathcal{R}_{max}|))$ time. Assuming that in general $|\mathcal{V}_{max}|$ is longer than others, the complexity can be simplified to $O(k|\Delta||\mathcal{V}_{max}|)$. In addition, the `BestContainingMatch` and `BestContainedMatch` takes at most $O(k^2|\mathcal{V}_{max}|)$. Therefore, the total computational complexity of the `QueryMatching` algorithm is $O(k|\Delta||\mathcal{V}_{max}| + k^2|\mathcal{V}_{max}|)$.

5. Performance Evaluation via Experiments

The experiments were performed on a Sun Ultra 2 machine with 256 MB RAM. Each test run was scheduled as a cron job and executed between midnight and 6am to minimize the effect of the load at the web site. The test-bed, CoWeb, was implemented in Java using jdk1.1.7.

Input: $\mathcal{Q}$, $\mathcal{U}_V \leftarrow \{\mathcal{V}_1, ..., \mathcal{V}_k\}$, Δ; **Output**: Best $\leftarrow \mathcal{V}_i \in \mathcal{U}_V$;

Best $\leftarrow \emptyset$; B_{cnting}, B_{cnted}, $\mathrm{B}_{ovlp} \leftarrow \emptyset$;
for $\mathcal{V}_i \leftarrow \mathcal{V}_1$ to $\mathcal{V}_k$ **do**
  switch Δ-`MatchType`($\mathcal{Q}$, $\mathcal{V}_i$, Δ) **do**
  case `exact_match:` **return** $\mathcal{V}_i$;
  case `containing_match:` $\mathrm{B}_{cnting} \leftarrow \mathrm{B}_{cnting} + \mathcal{V}_i$;
  case `contained_match:` $\mathrm{B}_{cnted} \leftarrow \mathrm{B}_{cnted} + \mathcal{V}_i$;
  case `overlapping_match:` $\mathrm{B}_{ovlp} \leftarrow \mathrm{B}_{ovlp} + \mathcal{V}_i$;
  otherwise: skip;
if $B_{cnting} \neq \emptyset$ **then** Best $\leftarrow$ `BestContainingMatch`(B_{cnting});
else if $B_{cnted} \neq \emptyset$ **then** Best $\leftarrow$ `BestContainedMatch`(B_{cnted});
else if $B_{ovlp} \neq \emptyset$ **then** Best $\leftarrow$ `BestOverlappingMatch`(B_{ovlp});
return Best;

Figure 6. The `QueryMatching` algorithm.

We used the following schema available from USAir site[3]. Among 7 attributes, both `org` and `dst` are mandatory attributes, thus they should always be bounded in a query.

```
USAir(org, dst, airline, stp, aircraft, flt, meal)
```

5.1. Generating Synthetic Test Queries

Semantic caching inevidently behaves very sensitively according to the *semantic locality* (i.e., the similarity among queries) of the test queries. Because of difficulties to obtain real-life test queries from such web sources, synthetic test sets with different *semantic localities* were generated to evaluate our semantic caching scheme. Two factors to the query generator were manipulated using the distribution $\mathcal{D}$=$\{N_0{:}P_0$, $N_1{:}P_1$, ..., $N_7{:}P_7\}$ or $\mathcal{D}$=$\{org{:}P_0$, $dst{:}P_1$, ..., $meal{:}P_7\}$, where N_i is the number of attributes used in the `WHERE` condition and P_i is the percentage of the P-th item.

1. The number of the attributes used in the `WHERE` condition (NUM): A test query with a large number of attribute conditions (e.g., `age=20` $\wedge$ `40k<sal<50k` $\wedge$ `title='manager'`) is more specific than that of a small number of attribute conditions (e.g., `age=20`). Therefore, a test set with many such specific queries is likely to perform badly in semantic caching since there are not many exact or containing

[3] Flight schedule site at http://www.usair.com/. At the time of writing, we noticed that the web site has slightly changed its web interface and schema since then.

matches. Let us denote the number of attributes used in the `WHERE` condition as N_i (i.e., N_3 means that 3 attributes are used in the `WHERE` condition). For instance, the following input distribution $\mathcal{D}$=\{N_0:30%, N_1:20%, N_2:15%, N_3:3%, N_4:2%, N_5:6%, N_6:3%, N_7:1%\} can be read as "Generate more queries with short conditions than ones with long conditions. The probability distributions are 30%, 20%, 15%, 13%, 12%, 6%, 3%, 1%, respectively".

2. The name of the attributes used in the `WHERE` condition (NAME): A test set containing many queries asking about common attributes is semantically skewed and is likely to perform well with respect to semantic caching. Therefore, different semantic localities can be generated by manipulating the name of the attributes used in the `WHERE` condition. For instance, the following input distribution $\mathcal{D}$=\{org:14.3%, dst:14.3%, airline:14.3%, stp:14.3%, aircraft:14.3%, flt:14.3%, meal:14.3%\} can be read as "All 7 attributes are equally likely being used in test set". As an another example, the fact that flight number information is more frequently asked than meal information can be represented by assigning a higher percentage value to the `flt` attribute than the `meal` attribute.

5.2. Query Space Effect

Another important aspect in generating synthetic test queries is the *Query Space* that is the sum of all the possible test queries. For instance, for the input distribution for the NUM factor $\mathcal{D}$=\{N_0:0%, N_1:0%, N_2:$\frac{100}{6}$%, N_3:$\frac{100}{6}$%, N_4:$\frac{100}{6}$%, N_5:$\frac{100}{6}$%, N_6:$\frac{100}{6}$%, N_7:$\frac{100}{6}$%\}, the effects of applying this distribution to 100 query space and 1 million query space are different. That is, the occurrence of a partial or full match in the case with 100 query space is much higher than the occurrence of those in the case with 1 million query space. To take into account this effect, we need to adjust the percentage distribution.

Formally, given the n attribute list, $\{A_1, ..., A_m | A_{m+1}, ..., A_n\}$, among which $A_1, ..., A_m$ are mandatory attributes and the rest are optional attributes, and their domain value list, $D_1, ..., D_n$, respectively, all possible number of query combinations *(query space)*, T, satisfies:

$$\left(\prod_{j=0}^{m}|D_j|\right)\left(\sum_{k=0}^{n-m}|\alpha|^k\binom{n-m}{k}\right) \le T \le \left(\prod_{j=0}^{m}|D_j|\right)\left(\sum_{k=0}^{n-m}|\beta|^k\binom{n-m}{k}\right) \quad (1a)$$

where $\alpha = min(D_{m+1}, ..., D_n), \beta = max(D_{m+1}, ..., D_n)$, and $|D_j|$ is the cardinality of the values in domain D_j.

According to the calculation using Equation 1a, for instance, a total of 32,400 different SQL queries (i.e., query space) can be generated

Table II. Breakdown between the number of SELECT conditions and query space.

Number of attributes (NUM)	Query space size	Query space percentage
0	0	0 %
1	0	0 %
2	36	0.1 %
3	540	1.67 %
4	3168	9.8 %
5	9072	28.0 %
6	12672	39.2 %
7	6912	21.4 %
Total	32400	100%

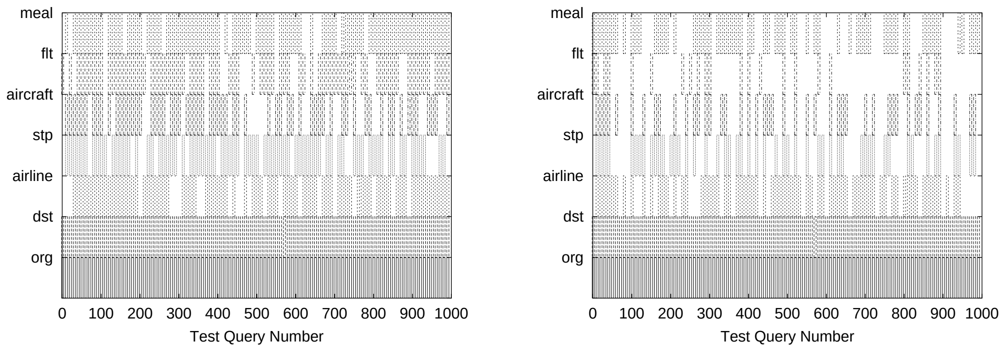
from the given USAir schema. The breakdown of query space is shown in Table II. Using the query space size and percentage, now we can adjust the percentage distribution for the input of the query generator. For instance, to make a uniform distribution in terms of the number of attributes, we give the following input distribution to the generator: $\mathcal{D}$=$\{N_0$:0%, N_1:0%, N_2:0.1%, N_3:1.67%, N_4:9.8%, N_5:28%, N_6:39.2%, N_7:21.4%$\}$, instead of $\mathcal{D}$=$\{N_0$:0%, N_1:0%, N_2:$\frac{100}{6}$%, N_3:$\frac{100}{6}$%, N_4:$\frac{100}{6}$%, N_5:$\frac{100}{6}$%, N_6:$\frac{100}{6}$%, N_7:$\frac{100}{6}$%$\}$.

5.3. Test Sets

The four test sets (`uni-uni`, `uni-sem`, `sem-uni`, and `sem-sem`) were generated by assigning different values to the two input parameters (NUM and NAME) after adjusting the query space effect. They are shown in Table III. `uni` and `sem` stand for *uniform* and *semantic* distribution, respectively. The total query space was set to 32,400. Each test set with 1,000 queries was randomly picked based on the two inputs. The `sem` values for the input NUM were set to mimic the Zipf distribution (Zipf, 1949), where it is shown that humans tend to ask short and simple questions more often than long and complex ones. The `sem` values for the input NAME were set arbitrarily, assuming that airline or stopover information would be more frequently asked than others. Figure 7 shows the different access patterns of the uniform and semantic distribution in terms of the chosen attribute names.

The following is an example of a typical test query generated.

```
SELECT org, dst, airline, stp, aircraft, flt, meal
```



a. Uniform distribution b. Semantic distribution

Figure 7. Attribute name access patterns. Shaded area means the attribute is being used in the test query. Since both `org` and `dst` attributes are mandatory, they are chosen always (thus completely shaded). Since the case b has more semantics, its access pattern is more skewed (i.e., `flt` attribute is seldom accessed in b while it is as frequently accessed as other attributes in a.) Also the case b shows `airline` attribute is more favored in the test query (the row is mostly shaded) than `aircraft` attribute.

Table III. Uniform and semantic distribution values used for generating the four test sets.

Scheme	Number of the attributes used (NUM)							
	0	1	2	3	4	5	6	7
`uni`	0%	0%	16.7%	16.7%	16.7%	16.7%	16.7%	16.7%
`sem`	0%	0%	40%	25%	15%	10%	5%	5%

Scheme	Name of the attributes used (NAME)						
	org	dst	airline	stp	aircraft	flt	meal
`uni`	100%	100%	20%	20%	20%	20%	20%
`sem`	100%	100%	40%	25%	10%	5%	20%

```
FROM   USAir
WHERE  org='LAX'   AND dst='DCA'
AND    6<=flt      AND 1<=stp<=2   AND meal='supper'
AND    aircraft='Boeing 757-200'
```

5.4. Performance Metrics

1. Average Response Time $\mathcal{T}$: $\mathcal{T}$ = (*total response time for n queries*) / n. To eliminate the initial noise when an experiment first starts, we can use $\mathcal{T}$ from the k queries of the sliding window instead of n queries in the query set.

2. Cache Coverage Ratio $\mathcal{R}_c$: Since the traditional cache "hit ratio" does not measure the effect of partial matching in semantic caching, we propose to use a *cache coverage ratio* as a performance metric. Given a query set consisting of n queries $q_1, ..., q_n$, let L_i be the number of answers found in the cache for the query q_i, and let M_i be the total number of answers for the query q_i for $1 \leq i \leq n$. Then $\mathcal{R}_c = \frac{\sum_{i=1}^{n} \mathcal{R}_{q_i}}{n}$, where 1) $\mathcal{R}_{q_i} = \frac{L_i}{M_i}$ if $M_i > 0$ and 2) $\mathcal{R}_{q_i} = c$ for $0 \leq c \leq 1$ if $M_i = 0$[4]. For instance, the query coverage ratio $\mathcal{R}_q$ of the exact match and containing match is 1 since all answers must come from the cache. Similarly, $\mathcal{R}_q$ of the disjoint match is 0 since all answers must be retrieved from the web source.

5.5. Experimental Results

In Figure 8, we compared the performance difference of three caching cases: 1) no caching (NC), 2) conventional caching using exact matching (CC), and 3) semantic caching using the *extended* matching (SC). Both cache sizes were set to 200KB. Regardless of the types of test set, NC shows no difference in performance. Since the number of exact matches was very small in all the test sets, CC shows only a little improvement in performance as compared to the NC case. Due to the randomness of the test sets and large number of containing matches in our experiments, SC exhibits a significantly better performance than CC. The more semantics the test set has (thus the more similar queries are found in the cache), the less time it takes to determine the answers.

Next, we studied the behavior of semantic caching with respect to cache size. We set the replacement algorithm as LRU and ran four test sets with cache sizes equal to 50KB, 100KB, 150KB, and unlimited. Because the number of answers returned from the USAir web site is, on average, small, the cache size was set to be small. Each test set contained 1,000 synthetic queries. Figure 9.a and Figure 9.b show the $\mathcal{T}$ and $\mathcal{R}_c$ for semantic caching with selected cache sizes. The graphs show that the $\mathcal{T}$ decreases and the $\mathcal{R}_c$ increases proportionally as cache size increases. This is due to the fact that there are fewer cache

[4] In our experiments, c was set to 0.5 for the overlapping and contained match when $M_i = 0$.

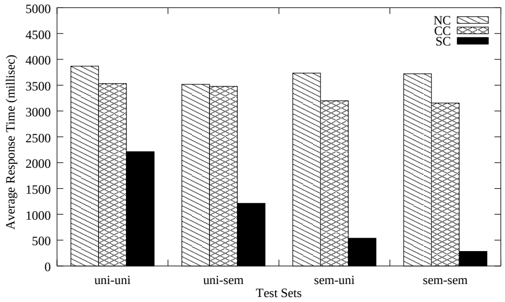
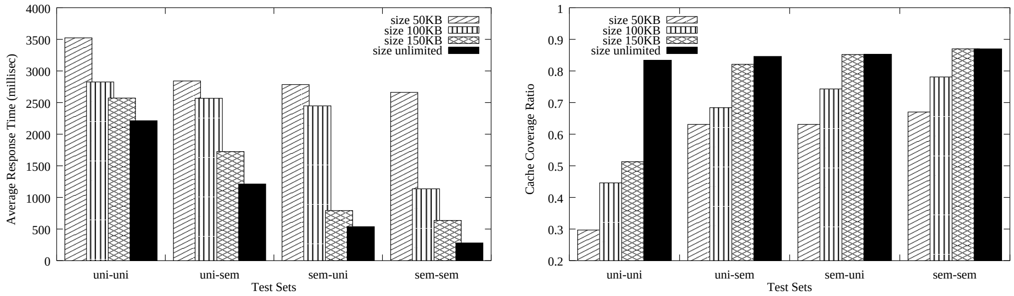


Figure 8. Performance comparison of the semantic caching with conventional caching.

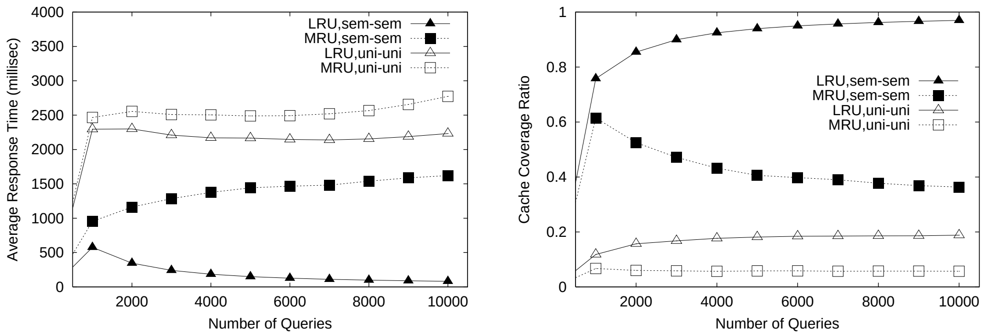


a. Average response time $\mathcal{T}$ b. Cache coverage ratio $\mathcal{R}_c$

Figure 9. Performance comparison of four test sets with selected cache sizes.

replacements. The degree of the semantic locality in the test set plays an important role. The more semantics the test set has, the better it performs. Due to no cache replacements, there is only a slight difference for the unlimited cache size in the $\mathcal{R}_c$ graph. The same behavior occurs in the $\mathcal{R}_c$ graph for the cache size with 150KB for the `sem-uni` and `sem-sem` test sets.

Next, we compared the performance difference between the LRU (least recently used) and MRU (most recently used) replacement algorithms. Due to limited space, we only show `uni-uni` and `sem-sem` test set results. For this comparison, 10,000 synthetic queries were generated in each test set and the cache size was fixed to be 150KB. Figure 10.a shows the $\mathcal{T}$ of the two replacement algorithms. For both test sets, LRU outperformed MRU. Further, the difference of the $\mathcal{T}$ between LRU and MRU increased as the semantic locality increased. This is because when there is a higher semantic locality, it is very likely that there is also a higher temporal locality. Figure 10.b shows the $\mathcal{R}_c$ of the two replacement algorithms. Similar to the $\mathcal{T}$ case, LRU



a. Average response time $\mathcal{T}$ b. Cache coverage ratio $\mathcal{R}_c$

Figure 10. Performance comparison of four test sets with LRU and MRU replacement algorithms.

Table IV. Distribution of match types for four test sets.

Test sets	Exact	Containing	Contained	Overlapping	Disjoint
`uni-uni`	0.4%	13.5%	7.8%	32.1%	46.2%
`uni-sem`	0.5%	27.7%	12.8%	36.8%	22.2%
`sem-uni`	5.1%	44.6%	12.0%	25.1%	13.2%
`sem-sem`	6.1%	52.0%	15.1%	18.0%	13.6%
Average	3.025%	34.35%	11.925%	28.0%	23.8%

outperformed MRU in the $\mathcal{R}_c$ case as well. Note that the `sem-sem` case in the $\mathcal{R}_c$ graph of the LRU slightly increased as the number of test queries increased while it stayed fairly flat in the `uni-uni` case. This is because when there is a higher degree of semantic locality in the test set such as in `sem-sem` case, the replacement algorithm does not lose its querying pattern (i.e., semantic locality). That is, the number of exact and containing matches is so high (i.e., 58.1% combined in Table IV) that most answers are found in the cache, as opposed to a web source. On the other hand, in the `sem-sem` case, the $\mathcal{R}_c$ graph of the MRU decreased as the number of test queries increased. This is true due to the fact that MRU loses its querying pattern by swapping the most recently used item from the cache.

Table IV shows the average percentages of the five match types based on 1,000 queries for four test sets. The fact that partial matches (contained and overlapping matches) constitute about 40% shows the potential usage of the knowledge-based matching technique.

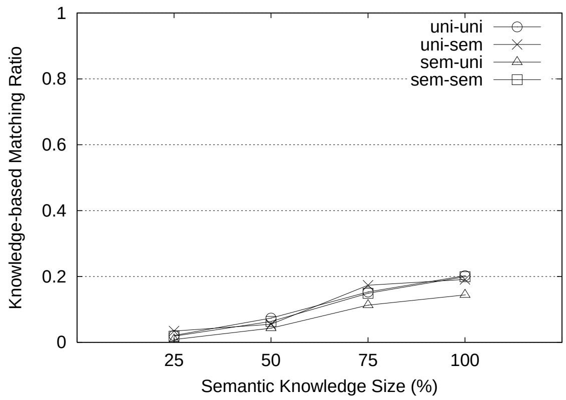


Figure 11. Performance comparison of the knowledge-based matching.

Figure 11 shows an example of the knowledge-based matching using semantic knowledge. We used a set of induced rules acquired by techniques developed in Chu et al. (1994) as semantic knowledge. Figure 11 shows knowledge-based matching ratios ($\frac{\#\ knowledge\text{-}based\ matches}{\#\ partial\ matches}$) with selected semantic knowledge sizes. The semantic knowledge size is represented as a percentage against the number of semantic views. For instance, a size of 100% means that the number of induced rules used as semantic knowledge equals the number of semantic views in the cache. Despite a large number of partial matches in the `uni-uni` and `uni-sem` sets shown in Table IV, it is interesting to observe that the knowledge-based matching ratios were almost identical for all test sets. This is due to the fact that many of the partially matched semantic views in the `uni-uni` and `uni-sem` sets have very long conditions and thus fail to match the rules. Predictably, the effectiveness of the knowledge-based matching depends on the size of the semantic knowledge.

6. Conclusions

Semantic caching via query matching techniques for web sources is presented. Our scheme utilizes the query naturalization to cope with the schematic, semantic, and querying capability differences between the wrapper and web source. Further, we developed a semantic knowledge-based algorithm to find the best matched query from the cache. Even if the conventional caching scheme yields a cache miss, our scheme can potentially derive a cache hit via semantic knowledge. Our algorithm is guaranteed to find the *best* matched query among many candidates, based on the algebraic comparison of the queries and semantic context of the applications. To prove the validity of our proposed scheme, a set of experiments with different test queries and with different degrees of semantic locality were performed. Experimental results confirm the effectiveness of our scheme for different cache sizes, cache replacement algorithms and semantic localities of test queries. The performance improves as the cache size increases, as the cache replacement algorithm retains more querying patterns, and as the degree of the semantic locality increases in the test queries. Finally, an additional 15 to 20 % improvement in performance can be obtained using knowledge-based matching. Therefore, our study reveals that our semantic caching technique can significantly improve the performance of semantic caching in web databases.

Semantic caching at the mediator-level requires communication with multiple wrappers and creates horizontal and vertical partitions as well as joining of input queries (Godfrey and Gryz, 1999), which result in more complicated cache matching. Further research in this area is needed. Other cache issues that were not covered in this paper, such as selective materializing, consistency maintainence and indexing, also need to be further investigated. For instance, due to the autonomous and passive nature of web sources, wrappers and their semantic caches are not aware of web source changes. More techniques need to be developed to incorporate such web source changes into the cache design in web databases.

References

Adali, S., K. S. Candan, Y. Papakonstantinou and V. S. Subrahmanian. Query Caching and Optimization in Distributed Mediator Systems. Proc. *ACM SIGMOD*, 1996.

Alonso, R., D. Barbara and H. García-Molina. Data Caching Issues in an Information Retrieval System. *ACM TODS*, 15(3):359-384, 1990.

Ashish, N., C. A. Knoblock and C. Shahabi. Intelligent Caching for Information Mediators: A KR Based Approach. Proc. *KRDB*, 1998.

Chidlovskii, B. and U. M. Borghoff. Semantic Caching of Web Queries. *The VLDB J.*, 9(2):2-17, 2000.

Chandra, A. K. and P. M. Merlin. Optimal Implementation of conjunctive Queries in Relational Databases. Proc. *ACM Symp. on the Theory of Computing*, 1977.

Chang, C-C. K., H. García-Molina and A. Paepcke. Boolean Query Mapping Across Heterogeneous Information Sources. *IEEE TKDE*, 8(4), 1996.

Chen C. M. and N. Roussopoulos. The Implementation and Performance Evaluation of the ADMS Query Optimizer: Integrating Query Result Caching and Matching. Proc. *EDBT*, 1994.

Cluet, S., O. Kapitskaia and D. Srivastava. Using LDAP Directory Caches. Proc. *ACM PODS*, 1999.

Chu, W. W., Q. Chen and A. Huang. Query Answering via Cooperative Data Inference. *JIIS*, (3):57-87, 1994.

Chu, W. W., H. Yang, K. Chiang, M. Minock, G. Chow and C. Larson. CoBase: A Scalable and Extensible Cooperative Information System. *JIIS*, 1996.

Dar, S., M. J. Franklin, B. T. Jonsson and D. Srivastava. Semantic Data Caching and Replacement. Proc. *VLDB*, 1996.

Franklin, M. J., M. J. Carey and M. Livny. Local Disk Caching for Client-Server Database Systems. Proc. *VLDB*, 1993.

Florescu, D., A. Y. Levy and A. Mendelzon. Database Techniques for the World-Wide Web: A Survery. *ACM SIGMOD Record*, 1998.

García-Molina, H., J. Hammer, K. Ireland, Y. Papakonstantinou, J. Ullman, and Jennifer Widom. Integrating and Accessing Heterogeneous Information Sources in TSIMMIS. Proc. *AAAI Symp. on Information Gathering*, 1995.

Godfrey, P. and J. Gryz. Semantic Query Caching for Heterogeneous Databases. Proc. *KRDB*, 1997.

Godfrey, P. and J. Gryz. Answering Queries by Semantic Caches. Proc. *DEXA*, 1999.

Gryz. J. Query Folding with Inclusion Dependencies. Proc. *IEEE ICDE*, 1998.

Guo, S., W. Sun and M. A. Weiss. Solving Satisfiability and Implication Problems in Database Systems. *ACM TODS*, 21(2):270-293, 1996.

Johnson, D. S. and A. Klug. Testing Containment of Conjunctive Queries under Functional and Inclusion Dependencies. *J. of Computer and System Sciences (JCSS)*, 1984.

Keller, A. M. and J. Basu. A Predicate-based Caching Scheme for Client-Server Database Architectures. *The VLDB J.*, 5(1), 1996.

Larson, P.-Å. and H. Z. Yang. Computing Queries from Derived Relations. Proc. *VLDB*, 1985.

Lee, D. and W. W. Chu. Semantic Caching via Query Matching for Web Sources. Proc. *ACM CIKM*, 1999.

Levy, A. Y., A. O. Mendelzon, Y. Sagiv and D. Srivastava. Answering Queries Using Views. Proc. *ACM PODS*, 1995.

Levy, A. Y., A. Rajaraman, J. Ordille. Querying Heterogeneous Information Sources Using Source Descriptions. Proc. *VLDB*, 1996.

Qian, X. Query Folding. Proc. *IEEE ICDE*, 1996.

Ren, Q. and M. H. Dunham. Semantic Caching and Query Processing. Southern Methodist University, *TR-98-CSE-04*, 1998.

Saraiya, Y. Subtree Elimination Algorithms in Deductive Databases. *Ph.D Thesis, Stanford U.*, 1991.

Sellis, T. Intelligent Caching and Indexing Techniques For Relational Database Systems. *IS*, 13(2), 1988.

Ullman, J. D. Principles of Database and Knowledge-Base Systems. Volume II: The New Technologies. *Computer Science Press*, 1988.

Ullman, J. D. Information Integration Using Logical Views. Proc. *ICDT*, 1997.

Vassalos, V. and Y. Papakonstantinou. Describing and Using Query Capabilities of Heterogeneous Sources. Proc. *VLDB*, 1997.

Zaniolo, C., S. Ceri, C. Faloutsos, R. R. Snodgrass and V. S. Subrahm. Advanced Database Systems. *Morgan Kaufmann Pub.*, 1997.

Zipf, G. K. Human Behaviour and the Principle of Least Effort. *Addison-Wesley*, 1949.