A Semantic Caching Scheme for Wrappers in Web Databases

Dongwon Lee
Department of Computer Science
University of California at Los Angeles
dongwon@cs.ucla.edu

Wesley W. Chu
Department of Computer Science
University of California at Los Angeles
wwc@cs.ucla.edu

Last Revised: February 20, 1999

Abstract

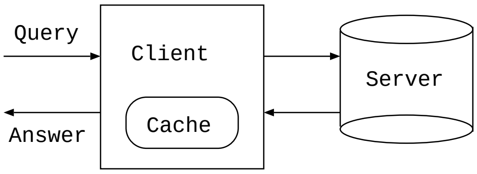
We present a new semantic caching scheme suitable for wrappers in web databases. Since the web sources in web databases have typically weaker querying capabilities than conventional databases, it is not trivial to apply existing semantic caching schemes directly. We provide a seamlessly integrated query translation and capability mapping between the wrappers and web sources in the semantic caching to cope with such difficulties and describe several related issues. In addition, an analysis on the match types between the user's input query and queries stored in the cache is presented. We show how to use semantic knowledge acquired from the data to avoid unnecessary access to web sources by transforming the cache miss to the cache hit. Further, a polynomial time algorithm based on the extended and knowledge-based matching is proposed to find the best matched query in the cache. Finally, experimental results are presented to illustrate the effectiveness of our proposed semantic caching scheme.

UCLA-CS-TR-990004

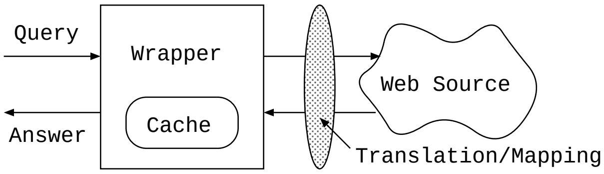
1 Introduction

Web databases allow users to pose queries to and retrieve answers from distributed, heterogeneous sources. Such systems usually consist of three components [ACPS96, GMHI+95]; 1) mediators to provide a distributed, heterogeneous data integration, 2) wrappers to provide a local translation and extraction, and 3) web sources which contain raw data to be queried and extracted. In the *virtual* approach [FLM98], 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 the prior queries and to reuse them ([ABGM90, FCL93]). *Semantic caching* (e.g., [LY85, Sel88, KB96, DFJS96, RD98]) exploits the semantic locality of the queries by caching a set of semantically associated results, instead of a tuple or page which is used in conventional caching. Most semantic caching schemes in client-server architectures are based on the assumption that all participating components are full-fledged database systems. Thus, when a user asks $\mathcal{A} \wedge \mathcal{B}$, but the cache contains $\mathcal{A} \wedge \mathcal{B} \wedge \mathcal{C}$, then the system sends a more complicated query $(\mathcal{A} \wedge \mathcal{B}) - \mathcal{C}$ to retrieve a remaining portion of the answers. However, in web databases, web sources such as plain web pages or form-based IR systems have very limited querying capabilities and can not easily support such complicated queries. Figure 1 illustrates the difference in semantic caching between client-server and web database architectures.



(a) Client-server architecture



(b) Web database architecture

Figure 1: Semantic caching in two architectures.

Our proposed semantic caching scheme is based upon the following three key ideas. First, since the querying capability of web sources is weaker than the user's queries, query translation and capability mapping need to be considered in the semantic caching. Second, when the number of queries stored in the cache increases, methods need to be developed to locate the best matched query from the set of candidates. This is because the time to find the best matched query in the cache is CPU-bounded, while the time to retrieve data from the web source is I/O-bounded. Third, a cache miss in a conventional caching scheme may be transformed into a cache hit via semantic knowledge.

This paper is organized as follows. In section 2, we introduce some background. In section 3, we describe our proposed semantic caching model and other related issues. In section 4, query matching between an input query and queries stored in the cache is presented. Then, experimental results follow in section 5. Finally, related works and conclusions are discussed in sections 6 and 7, respectively.

2 Background

Our caching scheme is implemented in a testbed web database called CoWeb (Cooperative Web Database) at UCLA. The architecture consists of mediator and wrapper components [GMHI$^+$95, ACPS96]. 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 [LC98], and web triggering with fuzzy threshold conditions [CYM99]. CoWeb uses the *global-as-view* approach where for each relation $\mathcal{R}$ in the mediated schema, we write a query over the web source relations specifying how to obtain $\mathcal{R}$'s tuples from the web source [FLM98]. 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 disjoining conjunctive predicates. CoWeb handles *selection* and *join* predicates with any of following operators $\{>, \geq, <, \leq, =\}$.

Our semantic caching approach is closely related to the *query satisfiability* and *query containment* problems [Ull89, GSW96]. Given a database $\mathcal{D}$ and query $\mathcal{Q}$, let applying $\mathcal{Q}$ on $\mathcal{D}$ be 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 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 $\mathcal{Q}$ and the query containment problem can be solved in $O(|\mathcal{Q}_1|^2+|\mathcal{Q}_2|)$ for the $\mathcal{Q}_1 \subseteq \mathcal{Q}_2$ [GSW96]. However, when the operator $\neq$ is added, it can be shown that the problem becomes NP-complete [CM77].

3 A Semantic Caching Scheme

3.1 Semantic Caching Model

A semantic cache is a hash table with a key and value pair entries. The key is the semantic description based on the previous queries. The value is a set of answers that satisfies the key. We denote the semantic description made of prior query by *semantic view*, $\mathcal{V}$. Then a semantic cache with n such entries {E} can be described as:

$$\text{semantic cache} = \{E_i | E_i = (\mathcal{V}_i, \langle\mathcal{V}_i\rangle), 1 \leq i \leq n\}$$

To represent a SQL query, we need: 1) relation names, 2) attributes used in the WHERE clause, 3) projected attributes, and 4) conditions in the WHERE clause [LY85, RD98]. For semantic caching in CoWeb, we use only "conditions in the WHERE clause" for the following reasons. Since there is a 1-to-1

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

mapping between the wrapper and the web source in CoWeb, the relation name is not needed. In addition, the majority of the web sources have a fixed output xml/html 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.

Queries stored in the semantic cache at the wrapper of the CoWeb has the form "`select * from web_source where condition`". By storing all attribute values in the cache, CoWeb completely avoids the *unrecoverability problem* which can be occurred when query results can not be recovered from the cache even if they are found due to the lack of some logical information [GG98]. Hereafter, we only use the conditions in the WHERE clause to represent the user's query.

3.2 Query Naturalization

Different web sources use different ontology. Due to security or performance concerns [FLM98], web sources provide different query processing capabilities. Therefore, the wrapper needs to pre-process the input query before submitting it to the web source.

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

`Generalization & Filtration:` When 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 later. For instance, a predicate ($5 < age < 10$) can be generalized into the predicate ($3 \leq age < 10$) with an additional filter ($age > 5$).

`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 ($1 < x < 4$) can be specialized by a disjunctive predicate ($x = 2 \vee x = 3$) provided that x is an integer type and its value is incremented by 1.

The original query from the mediator is called the *input query*. The generated query after pre-processing the input query is called the *native query*, since it is supported by the web source in a native manner [CGMP96]. Such pre-processing is called the *query naturalization*. The query used to filter out irrelevant data from the native query results is called the *filter query* [CGMP96]. 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. For further information on such schemes to represent querying capabilities, refer to [VP97], for instance.

3.3 Format of Semantic View

The format of the semantic views change when an input query is naturalized to a native query. Therefore, the question arises whether to use the input query format before the naturalization or the native query format after the naturalization as the semantic view.

`Input query as semantic views:` It is easier to find a match since the incoming query has the same format as the semantic view. However, this approach can not handle the generalization process well. When an input query A is generalized into a native query $A \vee B$, the native query retrieves more answers than requested from the web source. But since the semantic view uses the input query format A, it loses the additional data retrieved by B.

`Native query as semantic views:` The query needs to be naturalized first before it is compared to the semantic view since the incoming query has a different format from the semantic view. However, unlike input query case, the generalization can be easily handled.

We choose the second approach for trading space for speed; we store additional data retrieved from the web source by using the native query format as the semantic view when the generalization process occurs. Note that while the input query and filter query are always a conjunction of predicates, the native query can be in $\mathcal{DNF}$. If such case occurs, we partition the native query into conjunctive parts and save them separately.

Example 1: Suppose since a web source A does not support range operators for attribute x, an input query $\mathcal{Q}$: $1 \leq x \leq 3 \wedge y = 1$ needs to be naturalized (i.e., specialized) into a native query $\mathcal{V}$: $(x = 1 \wedge y = 1) \vee (x = 2 \wedge y = 1) \vee (x = 3 \wedge y = 1)$. Further, since semantic views use only conjunctive predicates, the native query $\mathcal{V}$ is partitioned into three conjunctive parts, $\mathcal{V}_1$: $x = 1 \wedge y = 1$, $\mathcal{V}_2$: $x = 2 \wedge y = 1$, and $\mathcal{V}_3$: $x = 3 \wedge y = 1$. 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 saved as semantic views in the cache. ■

3.4 The Control Flow in the Wrapper with Semantic Cache

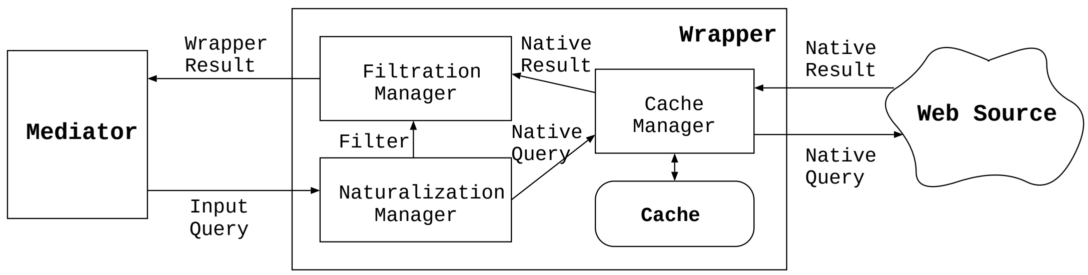


Figure 2: The control flow in the wrapper with semantic cache.

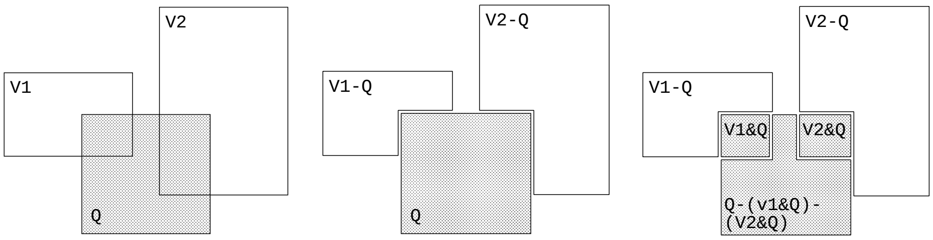
The control flow among the mediator, wrapper, and web source is illustrated in Figure 2. An input query from the mediator is naturalized in the wrapper and converted to a native query. A filter query can be generated. The cache manager then checks the native query against the semantic views stored in 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 through to remove 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 (i.e., disjunctive predicates are broken into conjunctive ones) is saved in the cache for future use.

3.5 Semantic View Overlapping

A semantic view creates a spatial object[2] in a n-dimensional hyperspace. Excessive overlapping of the semantic views may waste the cache space for duplicate answers. When overlapping is not allowed, different approaches can be adopted to cope with the newly inserted semantic view overlapping with the existing ones. Figure 3 illustrates three different approaches to the semantic view overlapping issue. Figure 3.a shows the approach which allows overlapping among semantic views. Although the overlapped portion among semantic views are redundantly stored in the cache, this is a straightforward scheme. In the physical layer of the cache, redundant answers can be stored as pointers to avoid duplicate copies (e.g., [KB96, LC98]). In general, the insertion of a semantic view that overlaps n semantic views in the cache can result in the formation of $n+1$ semantic views. Figure 3.b and Figure 3.c show approaches which do not allow overlapping among semantic views. In Figure 3.b, the overlapped portions are coalesced to the new semantic view and the remaining semantic views are modified appropriately. For instance, the semantic view $\mathcal{V}_1$ was modified to $\mathcal{V}_1 - \mathcal{Q}$ to take out the overlapped portion. The insertion of a semantic view that overlaps with n semantic views in the cache causes the formation of $n+1$ semantic views. In Figure 3.c, the overlapped portions become separate semantic views. The insertion of a semantic view overlaps n semantic views in the cache results in the formation of $2n+1$ semantic views. An adaptive approach between Figure 3.b and Figure 3.c is also possible; semantic views are coalesced if either one of them is smaller than 1% of the cache size [DFJS96].

CoWeb uses the approach shown in Figure 3.a for following reasons. 1) The original form of the semantic views are retained while in other approaches the existing semantic views that overlap with the new one need to be modified. (e.g., in Figure 3.b, $\mathcal{V}_1$ changes to $\mathcal{V}_1 - \mathcal{Q}$). Often this modification involves a set difference (negation) operator which is difficult to support by the web source, which has limited querying capabilities. In addition, as the semantic views get more complicated, it becomes more expensive to find a match, since the modified semantic view may no longer be a conjunctive form. 2) Since the semantic views retain the original form, they do not lose the user's querying pattern. If the semantic view of the frequently asked pattern $\mathcal{V}_1$ is modified to $\mathcal{V}_1 - \mathcal{Q}$ as in Figure 3.c, then a subsequent query $\mathcal{V}_1$ can not be answered efficiently. 3) By using a *reference counter* to keep track of the references of the answer tuples in implementing the cache, CoWeb can avoid the problem of the storing redundant

[2] This is called a *semantic region* in [DFJS96] and a *semantic segment* in [RD98].



3.a overlapping 3.b non-overlapping, full-coaleasing 3.c non-overlapping, no-coaleasing

Figure 3: Different approaches to semantic view overlapping issue. $\mathcal{V}_1$ and $\mathcal{V}_2$ are two semantic views in the cache and $\mathcal{Q}$ is a new semantic view inserted into the cache.

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

Table 1: Query match types and their properties. $\mathcal{V}$ is a semantic view and $\mathcal{Q}$ is a user query.

answers in the cache [KB96, LC98].

3.6 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 called the *full match* since all answers are in the cache while the overlapping and contained match are called the *partial match* since some answers need to be retrieved from the web sources.

The detailed properties of each match type are shown in Table 1. Note that for the contained and overlapping match, computing answers requires the union of the partial answers from the cache and from the web source. When a query $\mathcal{Q}$ is compared to a set of semantic views, there may be several containing, contained, or overlapping matches. Then, the best one is the one which yields the least overhead cost to answer the given query. Our approach is to construct a lattice among many candidates and find the best one. Suppose $\mathcal{Q}$ is a query and $\mathcal{U}_Q$ corresponds to the set of all the *containing* or *contained matches* of $\mathcal{Q}$ 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 $\mathcal{U}_Q \cup \{\top, \perp\}$. For the 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 the contained match case, the `lub` of the lattice is the special symbol $\top$ and the `glb` of the lattice is the query $\mathcal{Q}$ itself.

Once we have constructed the lattice over candidate matches, we need a concept of minimality and maximality to find the best one. A containing match of $\mathcal{Q}$, say $\mathcal{A}$, is called the ***minimally-containing match*** of $\mathcal{Q}$ if and only if there is no other containing match of $\mathcal{Q}$, say $\mathcal{B}$, such that $\mathcal{A} \subseteq \mathcal{B} \subseteq \mathcal{Q}$ in $\mathcal{Q}$'s query containment lattice. Symmetrically, a contained match of $\mathcal{Q}$, say $\mathcal{A}'$, is called the ***maximally-contained match*** of $\mathcal{Q}$ if and only if there is no other contained match of $\mathcal{Q}$, say $\mathcal{B}'$, such that $\mathcal{A}' \supseteq \mathcal{B}' \supseteq \mathcal{Q}$ in $\mathcal{Q}$'s query containment lattice.

Example 2: A query containment lattice for $\mathcal{Q}$: $(1 < x < 4 \wedge 3 < y \leq 5 \wedge z = 'C')$ and their containing matches are illustrated in Figure 4. ■

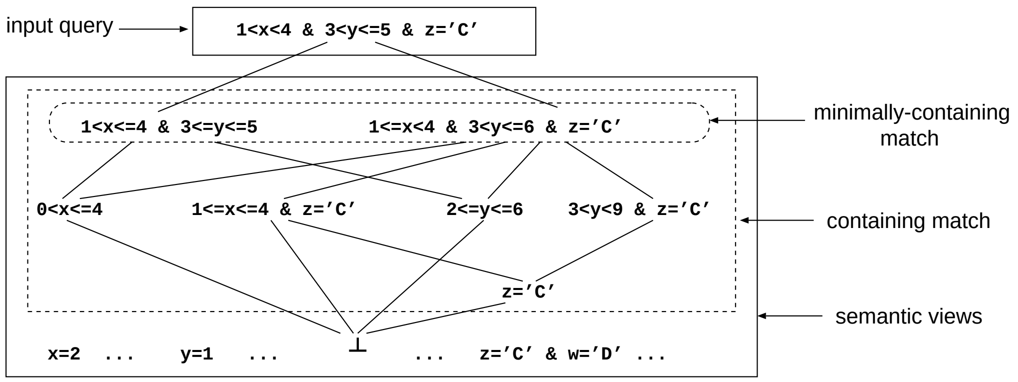


Figure 4: A *query containment lattice* example.

4 Query Matching between an Input Query and Semantic Views

4.1 Extended Matching

In matching an input query with a semantic view, the best case is the exact match which is equivalent to the conventional cache hit. The next best case is the containing match since it only contains some extra answers. After that, the contained match case is considered to be slightly better than the overlapping match case[3]. Although both have to retrieve partial answers from the cache as well as the web source, the contained match does not contain extra answers in the cache (see Table 1).

4.1.1 The `MatchType` Algorithm

Given a semantic view $\mathcal{V}$ and input query $\mathcal{Q}$, the `MatchType` algorithm returns the proper match type using the properties in Table 1. Using algorithms developed for solving the satisfiability and containment

[3]The comparison is based on the case when the amount of the contained answers and the overlapped answers is same.

problem in the literature [Ull89, GSW96], the computation complexity of the `MatchType` algorithm is $O(|\mathcal{Q}|^2 + |\mathcal{V}|^2)$.

4.1.2 The `BestContainingMatch` and `BestContainedMatch` Algorithms

Given an input query and many candidate containing matches, the `BestContainingMatch` algorithm finds a containing match that incurs minimal effort to filter out extra answers. The algorithm first finds all minimally-containing matches by eliminating containing matches which contain other containing matches. Then, the algorithm selects the best one from all the minimally-containing matches. Note that for a given query $\mathcal{Q}$, there can be several minimally-containing matches found in the cache. In such cases, the *best* minimally-containing match can be selected based on heuristics.

Input : $\{\mathcal{V}_1, ..., \mathcal{V}_N\}$
Output: Best = $\mathcal{V}_i \in \{\mathcal{V}_1, ..., \mathcal{V}_N\}$

Best = $\emptyset$, Bucket$(C)_{min}$ = $\{\mathcal{V}_1, ..., \mathcal{V}_N\}$;
for $\mathcal{V}_i = \mathcal{V}_1$ *to* $\mathcal{V}_N$ **do**
  for $\mathcal{V}_j = \mathcal{V}_1$ *to* $\mathcal{V}_N$; $i \neq j$ **do**
    if `MatchType`$(\mathcal{V}_i,\ \mathcal{V}_j)$ = `containing_match` **then** Bucket$(C)_{min}$ -= $\mathcal{V}_j$;

for $\mathcal{V}_i \in$ *Bucket*$(C)_{min}$ **do** Best = choose one from Bucket$(C)_{min}$;
return Best;

Algorithm 1: The `BestContainingMatch`

If $|\mathcal{V}_{max}|$ denotes the length of the longest containing match and there are N containing matches, then the computation takes $O(N^2|\mathcal{V}_{max}|^2)$ without any indexing on the semantic views. Observe that the `BestCotainingMatch` algorithm is only justified when finding the best containing match with minimal effort to filter out the extra answers is better than selecting an arbitrary containing match followed by filtering. The The `BestContainedMatch` algorithm is the symmetric case of the `BestContainingMatch` algorithm.

4.1.3 The `BestOverlappingMatch` Algorithm

For the overlapping matches, we can not 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. For instance, one can 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.

4.2 Knowledged-based Matching

According to our experiments, partial matches constitute about 40% of all match types for given test sets (see Table 3). Interestingly, a partial match can be a full match in certain cases. For instance, for the `employee(name,gender,addr)` relation, a semantic view $\mathcal{V}$: $gender = 'male'$ is the overlapping match of a query $\mathcal{Q}$: $name = 'john'$. However if we know in fact that john is a male employee (supposedly), then $\mathcal{V}$ is a containing match of $\mathcal{Q}$ since $\mathcal{Q} \subseteq \mathcal{V}$. Since full matches eliminate the need to access the web source, transforming a partial match into a full match can improve the performance significantly.

4.2.1 Semantic Knowledge Acquisition

Obtaining semantic knowledge from the web source and maintaining it properly is an important issue. In general, such knowledge can be obtained by a human expert from the application domain. In addition, database constraints such as inclusion dependencies can be used. Knowledge discovery and data mining techniques are useful to obtain such knowledge (semi-)automatically. Rule induction also provides a way to acquire such semantic knowledge. For instance, a rule (0201 $\leq$ sonar.class $\leq$ 0215 $\rightarrow$ sonar.type = 'SSN') implying that all sonar objects whose class values are between 0201 and 0215 must be 'SSN' type can be automatically acquired [CCH94].

4.2.2 Semantic Knowledge Notation

We use a generic notation derived from [CCH94] to denote the containment relationship between two fragments of relations. A *fragment inclusion dependency* ($\mathcal{FIND}$) assures that values in the columns of one fragment must also appear as values in the columns of other fragment. Formally, a FIND has a form $\sigma_{\mathcal{P}} \diamond \sigma_{\mathcal{Q}}$ where $\diamond \in \{\equiv, \subseteq\}$ and $\mathcal{P}$ and $\mathcal{Q}$ are conjunctive WHERE conditions. 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., $\Delta = \Delta^*$).

4.2.3 Transforming Partial Matches to Full Matches

Our goal is to transform as many partial matches to full matches as possible with the given dependency 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 if possible.

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

Example 3: Suppose we have a query $\mathcal{Q}$: (x = 1) and a semantic view $\mathcal{V}$: (y = 2). Given a Δ: $\{\sigma_{0 \leq x \leq 2} \subseteq \sigma_{y=2}\}$, $\mathcal{V}$ becomes a containing match of $\mathcal{Q}$ since $\mathcal{Q} \subseteq LHS, RHS \subseteq \mathcal{V}$, and $\{LHS \subseteq RHS\} \in \Delta$. ■

4.2.4 The Δ-`MatchType` Algorithm

Let us first define an augmented containment in the presence of the dependency set Δ. 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 an arbitrary relation $\mathcal{D}$ obeying the fragment inclusion dependency set Δ, then the query $\mathcal{Q}_1$ is Δ-*contained* in the query $\mathcal{Q}_2$ and denoted by $\mathcal{Q}_1 \subseteq_\Delta \mathcal{Q}_2$. If two queries Δ-*contain* each other, they are Δ-*equivalent* and denoted by $\mathcal{Q}_1 \equiv_\Delta \mathcal{Q}_2$.

Then, the Δ-`MatchType` algorithm can be easily implemented by modifying the `MatchType` algorithm; add additional input, Δ, and change all $\equiv$ to $\equiv_\Delta$ and $\subseteq$ to $\subseteq_\Delta$. The computational complexity of $\mathcal{Q} \equiv_{\Delta'} \mathcal{V}$ where Δ' contains single $\mathcal{FIND}$ = LHS $\diamond$ RHS is then $O(|\mathcal{Q}|^2 + |\mathcal{V}|^2 + |LHS|^2 + |RHS|^2)$. Let $|\mathcal{L}_{max}|$ and $|\mathcal{R}_{max}|$ denote the length of the longest LHS and RHS in Δ and let $|\Delta|$ denotes the number of $\mathcal{FIND}$ in Δ, then total computational complexity of the Δ-`MatchType` algorithm is $O(|\Delta|(|\mathcal{Q}|^2 + |\mathcal{V}|^2 + |\mathcal{L}_{max}|^2 + |\mathcal{R}_{max}|^2))$ 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 full matches is I/O-bounded and the typical length of the conjunctive query is relatively short, it is a good performance trade-off in many applications to pay overhead cost for the CPU-bounded Δ-`MatchType` algorithm.

4.3 The `BestMatch` Algorithm: Putting It All Together

The `BestMatch` algorithm 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 naïve containment relationships but also knowledge-based containment relationships. If $|\mathcal{V}_{max}|$ denotes the length of the longest semantic views, then the `for` loop takes $O(N|\Delta|(|\mathcal{Q}|^2 + |\mathcal{V}_{max}|^2 + |\mathcal{L}_{max}|^2 + |\mathcal{R}_{max}|^2))$ time at most. Assuming that in general $|\mathcal{V}_{max}|$ is longer than others, we can rewrite the complexity as $O(N|\Delta||\mathcal{V}_{max}|^2)$. In addition, the `BestContainingMatch` and `BestContainedMatch` takes at most $O(N^2|\mathcal{V}_{max}|^2)$. Therefore, the total computational complexity of the `BestMatch` algorithm is $O(N|\Delta|(|\mathcal{Q}|^2 + |\mathcal{V}_{max}|^2 + |\mathcal{L}_{max}|^2 + |\mathcal{R}_{max}|^2)) + O(N^2|\mathcal{V}_{max}|^2) \approx O(N|\Delta||\mathcal{V}_{max}|^2) + O(N^2|\mathcal{V}_{max}|^2)$.

5 Experiments: Performance Evaluation

The experiments were performed on a Sun Sparc 20 machine. 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 testbed, CoWeb, was implemented in Java using jdk1.1.7.

Input : $\mathcal{Q}, \mathcal{U}_V = \{\mathcal{V}_1, ..., \mathcal{V}_N\}, \Delta$
Output: Best $= \mathcal{V}_k \in \mathcal{U}_V$

```
Best = null, Bucket(C) = Bucket(¬C) = Bucket(O) = ∅;
for V_i = V_1 to V_N do
    switch Δ-MatchType(Q, V_i, Δ) do
        case exact_match return V_i;
        case containing_match Bucket(C) += V_i;
        case contained_match Bucket(¬C) += V_i;
        case overlapping_match Bucket(O) += V_i; otherwise skip;
if Bucket(C) ≠ ∅ then Best = BestContainingMatch(Bucket(C));
else if Bucket(¬C) ≠ ∅ then Best = BestContainedMatch(Bucket(¬C));
else if Bucket(O) ≠ ∅ then Best = BestOverlappingMatch(Bucket(O));
return Best;
```

Algorithm 2: The `BestMatch`

5.1 Experimental Setup

We set up a wrapper which wraps up the USAir[4] web site and provides a local view:

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

to the mediator. Among 7 attributes, both `org` and `dst` are mandatory attributes, thus they should always be bounded in a query. To find out the effectiveness of our scheme, we generated test sets having different *semantic localities*. Test SQL queries in a given test set differ only in their WHERE clause while SELECT and FROM clauses are identical. We manipulated two factors to generate different semantic localities: 1) the number of attribute conditions and 2) the name of the attributes. If users ask short queries more frequently rather than long ones (i.e., fewer conditions), then we can manipulate the first factor to differentiate the generated test set. Also, if certain attributes in a schema are asked more frequently than others, we can manipulate the second factor. Using the above USAir schema having 7 attributes, for instance, the first input {0:30%, 1:20%, 2:15%, 3:13%, 4:12%, 5:6%, 6:3%, 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". For the second input, {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 "When selecting the attribute, all 7 attributes have same probability of being chosen".

We have generated four test sets, `uni-uni`, `uni-sem`, `sem-uni`, and `sem-sem`, by giving different values for the two inputs. The `uni` and `sem` stand for a *uniform* and *semantic* distribution, respectively. The total number of the possible distinct queries that our query generator can make was set to about 32,400 and 1,000 queries for each test set were randomly picked based on the two inputs. Actual values used

[4]Flight schedule site at http://www.usair.com. Experiments were performed during Oct. and Nov. period in 1998. At the time of writing, we noticed that the web site has slightly changed its web interface and schema since then.

Scheme	Number of the attributes in conditions						Name of the attributes in conditions				
	2	3	4	5	6	7	airline	stp	aircraft	flt	meal
uniform	16.7%	16.7%	16.7%	16.7%	16.7%	16.7%	20%	20%	20%	20%	20%
semantic	40%	25%	15%	10%	5%	5%	40%	25%	10%	5%	20%

Table 2: Uniform and semantic distribution values for two factors for generating four test sets.

in our experiments are shown in Table 2. The first semantic scheme values are set to mimic the Zipf distribution [Zip49], where it is shown that human tends to ask short and simple questions more often than long and complex ones. The second semantic scheme values are set arbitrarily assuming that airline or stopover information will be more frequently asked than others. As long as it is a semantically skewed test set, it suffice our purpose to test semantic caching. Following is a typical test query generated.

```
SELECT   *              FROM    USAir
WHERE    org = 'LAX'    AND     dst = 'JFK'
AND      stp <= 1       AND     airline = 'US Express'
AND      meal = 'S/S'   AND     aircraft = 'Boeing 757-200'
```

5.2 Performance Metrics

We use two metrics to evaluate the effectiveness of the caching scheme.

1. ART (Average Response Time): $\frac{total\ response\ time\ for\ N\ queries}{N}$. Note that if we want to eliminate the initial noise when an experiment first starts, then we can use ART of the last k queries (sliding window) in the query set.

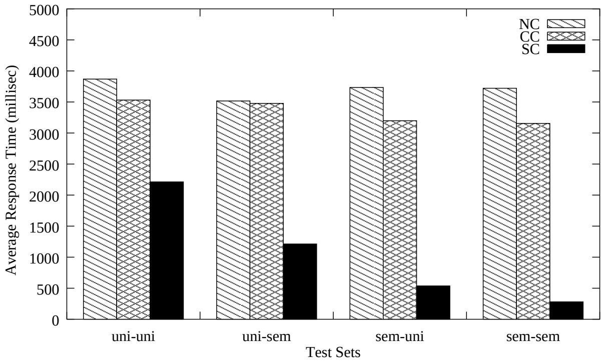
2. CCR (Cache Coverage Ratio): Since the traditional cache hit ratio metric misses the effect of the partial matching in the semantic caching, we use a *cache coverage ratio* instead. Given a query set consisting of N queries $q_1, ..., q_N$, let A_i be the number of answers found in the cache for the query q_i, and let B_i be the total number of answers for the query q_i for $1 \leq i \leq N$. Then $CCR = \frac{\sum_{i=1}^{N} QCR_i}{N}$, where 1) $QCR_i = \frac{A_i}{B_i}$ if $B_i > 0$ and 2) $QCR_i = c$ for $0 \leq c \leq 1$ if $B_i = 0$[5]. The QCR (query coverage ratio) of the exact match and containing match is 1 since all answers must come from the cache, while the QCR of the disjoint match is 0, since all answers must be retrieved from the web source.

5.3 Experimental Results

First, in Figure 5, we compared the performance difference of three cases: 1) no caching (NC), 2) conventional caching using exact match only (CC), and 3) semantic caching (SC). Both cache sizes were set to 200KB. Regardless of the type of the test set, NC shows no difference. CC shows only a little improvement over NC since the number of the exact matches was very small in all test sets. Note that if the small set of queries are very frequently asked, then CC will capture the semantic locality and may perform as good as SC. However, in our experiments, because of the randomness of test sets and large

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

number of containing matches, SC exhibits significant performance improvement over the others. The more semantics the test set has (thus more similar queries are found), the less time it takes to process.

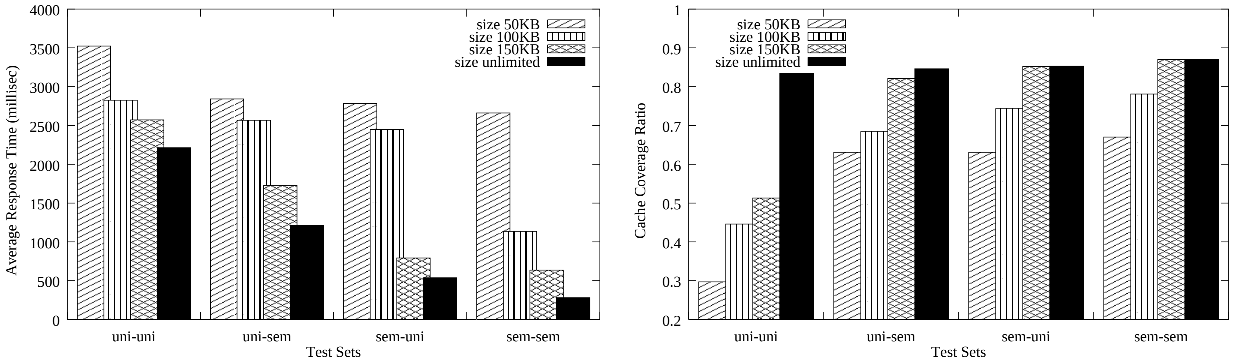


Average response time (millisec)			
	Test cases		
Test sets	NC	CC	SC
`uni-uni`	3,868	3,531	2,213
`uni-sem`	3,516	3,478	1,213
`sem-uni`	3,734	3,198	538
`sem-sem`	3,720	3,154	281

Figure 5: Performance comparison of three test cases.

Next, we tested how the semantic caching behaves differently with respect to the cache size. For this test, we set the replacement algorithm as LRU and run four test sets with varying cache sizes, 50KB, 100KB, 150KB, and unlimited, respectively. Because the USAir web site returns a small number of answers for an average query, the cache size was set very small If the average size of the answers to be stored in the cache becomes much bigger, then the cache size would be also increased to the MB level. Each test set contains 1,000 syntactic queries. Figure 6.a and Figure 6.b show the ART and the CCR of the semantic caching with varying size, respectively. Note that ART and CCR graph is symmetric. They show that as the cache size increases, the ART decreases and the CCR increases proportionally, since there are likely fewer cache replacements. The degree of the semantic locality in the test set plays important role; the more semantics the test set has, the better performance it has. The reason that there is little difference for the unlimited cache size in the CCR graph is due to no cache replacements. The same behavior occurs for the cache size 150KB for the `sem-uni` and `sem-sem` test sets for the CCR graph (Figure 6.a and Figure 6.b).

Next, we compared the performance difference between the LRU and MRU algorithm. The low-level memory management uses the strategy that uses a reference counter developed in [LC98]. Due to space limitation, we only show here `uni-uni` and `sem-sem` test set results. For this comparison, 10,000 syntactic queries are generated in each set and cache size is fixed to 150KB. Figure 7.a shows the ART of the two replacement algorithms. For both test sets, LRU outperforms MRU. Further, the ART gap between LRU and MRU increases as the semantic locality increases. This is because when there is a higher semantic locality, it is very likely that there is also a higher temporal locality. Figure 7.b shows the CCR of the two replacement algorithms. Like the ART case, LRU outperforms MRU. Note that the CCR graph of the LRU for `sem-sem` case slightly increases as the number of test queries increases while it stays fairly flat for `uni-uni` case. This is because when there is a higher degree of semantic locality in the test set such as `sem-sem` case and the replacement algorithm does not lose querying pattern (i.e., semantic locality) such

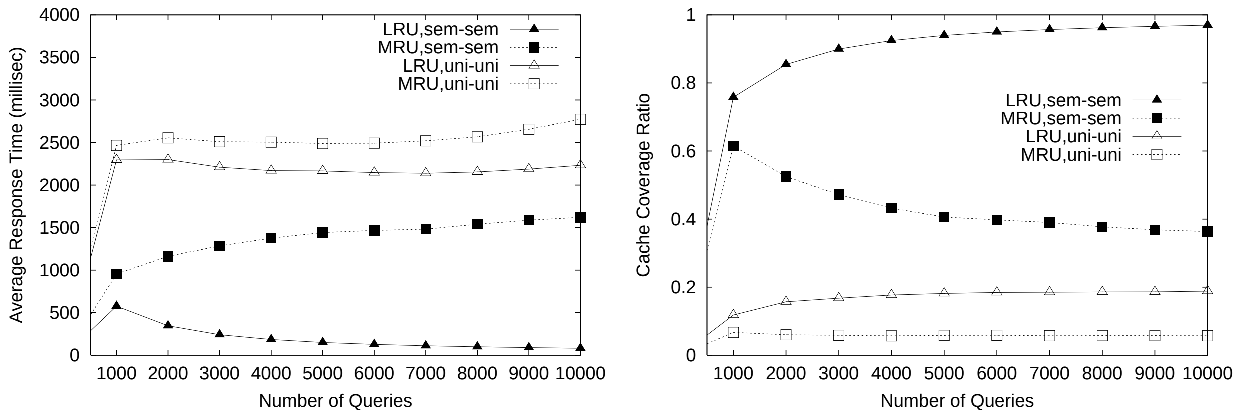


6.a Average response time

6.b Cache coverage ratio

Figure 6: Performance comparison for selected test sets with varying cache size.

as LRU, the number of the exact and containing matches are so high that most answers are found in the cache instead of the web source (Both cases constitute about 60% combined in Table 3). On the other hand, the CCR graph of the MRU for `sem-sem` case decreases as the number of test queries increases because MRU loses the querying pattern by swapping out the most recently used item from the cache.



7.a Average response time

7.b Cache coverage ratio

Figure 7: Performance comparison between LRU and MRU algorithms.

Table 3 shows the average percentages of the five match types for four test sets each with 1,000 queries. The fact that partial matches (contained and overlapping matches) constitute about 40% on average shows the potential usage of the knowledge-based matching technique. Figure 8.a shows an example of the knowledge-based matching using semantic knowledge, which is in this case a set of induced rules acquired by techniques developed in [CCH94]. Figure 8.b shows knowledge-based matching ratio $(\frac{\#\ knowledge\text{-}based\ matches}{\#\ partial\ matches})$ with varying semantic knowledge sizes. The size is represented in percentage against the number of the semantic views. Despite large number of the partial matches in `uni-uni` and `uni-sem` sets shown in Table 3, the ratios are almost identical for all test sets. This is mainly because

Test sets	Match types				
	Exact match	Containing match	Contained match	Overlapping match	Disjoint match
`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%

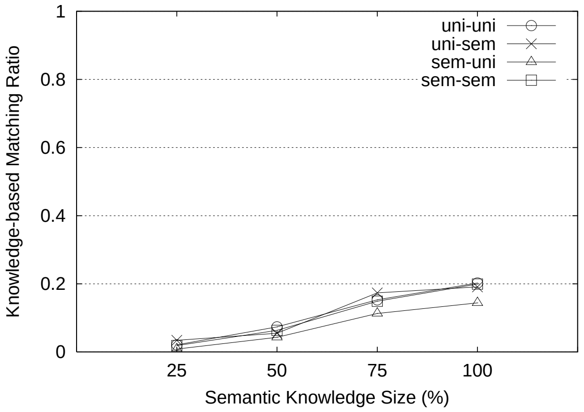
Table 3: Five match type breakdown in the semantic caching.

many of the partially matched semantic views in `uni-uni` and `uni-sem` sets have very long conditions, they fail to match the rules. Predictably, the degree of the knowledge-based matching is dependent on the size of the semantic knowledges.

$\mathcal{Q}$: $org = {'LAX'} \wedge dst = {'JFK'} \wedge 87 \leq flt \leq 88$
$\mathcal{V}$: $aircraft = {'Boeing757-200'}$

```
Rules:
1: (70 <= flt <= 79) => (aircraft = 'Boeing 747-200')
2: (80 <= flt <= 89) => (aircraft = 'Boeing 757-200')
3: ...
```

8.a User asks flight schedule from `LAX` to `JFK` with a range-specified flight number, but semantic cache has only overlapping match $\mathcal{V}$. However, according to the induced rule `2`, $\mathcal{V}$ is transformed into the containing match.



8.b Knowledge-based matching

Figure 8: Performance of the knowledge-based matching.

6 Related Works

Several areas related with the semantic caching have been studied extensively in the past: conventional caching (e.g., [ABGM90, FCL93]), query satisfiability and containment problems (e.g., [Ull89, GSW96]), view materialization (e.g., [LY85, LMSS95]), semantic query optimization (e.g., [CCH94, GGM96]), multiple query optimization, etc. Recently, semantic caching in client-server or multi-database architecture has received attention [KB96, DFJS96, GG98, RD98, AKS98]. Our scheme is, however, more suitable for web databases where the querying capability of the sources (i.e., web sites) is not compatible with the clients (i.e., wrappers).

[Sel88] discusses semantic caching and its indexing techniques, but it requires the cached results to be exactly matched with the input query. Our approach supports non-exact matches by using semantic knowledge. [CR94] approaches the semantic caching from the query planning and optimization point of view. [DFJS96] maintains cache space by coalescing or splitting the semantic regions while we maintains cache space by reference counters with the allowed overlapping of the semantic regions. Further, we present technique to find the best matched query under different circumstances via extended and knowledge-based matching. In [KB96], *predicate descriptions* derived from previous queries are used to match an input query with the emphasis on updates in the client-server environment. Reference counters were used to reclaim cache space without the usage of the concept analogous to the semantic region [DFJS96].

[ACPS96] discusses semantic caching in the mediator environment with knowledge called *invariants* although their focus is on cost model-based query optimization. Although invariants are more powerful than $\mathcal{FIND}$ due to their support of arbitrary user-defined function as conditions, they are mainly used for substituting a domain call. On the contrary, $\mathcal{FIND}$ is simpler yet easier to express a fragment containment relationship on relations. Also $\mathcal{FIND}$ can be acquired (semi)-automatically. In [AKS98], 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, they reduce the number of semantic regions by merging them whenever possible similar to [KB96]. [RD98] defines a semantic caching formally and addresses query processing techniques derived from [LY85]. They introduce additional query trimming techniques which use the *amending query* in addition to the *probe* and *remainder query* developed in [DFJS96]. [GG98] introduces a comprehensive formal framework in which they illustrate issues such as when answers are in the cache, when answers in the cache can be recovered, etc. Also, they identify applications for which semantic caching can be useful.

7 Conclusions & Future Works

Semantic caching techniques for wrappers in web databases are presented. Our scheme utilizes the query naturalization to cope with the schematic, semantic, and capability difference between the wrapper and web source. Further, we developed a polynomial time algorithm to find the best matched query from the cache using semantic knowledge. Even if the conventional caching scheme would yield a cache miss, our scheme could potentially locate a cache hit via semantic knowledge. Our algorithm guarantees to find the *best* matched query from many candidates based on the algebraic comparison of the queries and heuristics of the application. To prove the validity of our proposed scheme, we conducted a comprehensive set of experiments for different test queries with different degrees of semantic locality. Our experimental results confirm the effectiveness of our scheme for different cache sizes, cache replacement algorithms and semantic locality of test queries; the performance substantially improved as the cache size increased, as the cache replacement algorithm retained more querying pattern like in LRU, and as the degree of the semantic locality in the test queries increased. Finally, about 15 to 20 % performance improvement was confirmed with the usage of knowledge-based matching.

Semantic caching at the mediator-level requires communication with multiple wrappers and creates horizontal and vertical partitions of input queries [GG98], resulting in more complicated cache matching.

Further research in this area is needed. Other cache issues which were not covered in this paper including selective caching, consistency maintainence, or indexing need to be studied further. For instance, due to the autonomous and passive nature of the web source, wrappers and their semantic caches are not aware of web source changes. An efficient way is needed to incorporate such changes of web sources into the cache in web databases.

References

[ABGM90] R. Alonso, D. Barbara, and H. García-Molina. "Data Caching Issues in an Information Retrieval System". *ACM Trans. on Database Systems (TODS)*, 15(3):359–384, September 1990.

[ACPS96] S. Adali, K. S. Candan, Y. Papakonstantinou, and V. S. Subrahmanian. "Query Caching and Optimization in Distributed Mediator Systems". In *ACM Proc. of the Int'l Conf. on Management of data (SIGMOD)*, pages 137–148, 1996.

[AKS98] N. Ashish, C. A. Knoblock, and C. Shahabi. "Intelligent Caching for Information Mediators: A KR Based Approach. In *Proc. of the 5th Knowledge Representation Meets Databases Workshop (KRDB)*, Seattle, May 1998.

[CCH94] W. W. Chu, Q. Chen, and A. Huang. "Query Answering via Cooperative Data Inference". *Journal of Intelligent Information Systems (JIIS)*, (3):57–87, feb 1994.

[CGMP96] C-C. K. Chang, H. García-Molina, and A. Paepcke. "Boolean Query Mapping Across Heterogeneous Information Sources". *IEEE Trans. on Knowledge and Data Engineering (TKDE)*, 8(4):515–521, August 1996.

[CM77] A. K. Chandra and P. M. Merlin. "Optimal Implementation of conjunctive Queries in Relational Databases". In *Proc. of the 9th ACM Symp. on the Theory of Computing*, pages 77–90, 1977.

[CR94] C. M. Chen and N. Roussopoulos. "The Implementation and Performance Evaluation of the ADMS Query Optimizer: Integrating Query Result Caching and Matching". In *ACM Proc. of the 4th Int'l Conf. on Extending Database Technology (EDBT)*, Cambridge, UK, 1994.

[CYM99] W. W. Chu, X. Yang, and W. Mao. "CoSent: A Cooperative Sentinel for Database Systems". Technical Report 990002, UCLA-CS-TR, 1999.

[DFJS96] S. Dar, M. J. Franklin, B. T. Jonsson, and D. Srivastava. "Semantic Data Caching and Replacement". In *Proc. of the 22nd Int'l Conf. on Very Large Data Bases (VLDB)*, pages 330–341, Mumbai (Bombay), India, 1996.

[FCL93] M. J. Franklin, M. J. Carey, and M. Livny. "Local Disk Caching for Client-Server Database Systems". In *Proc. of the 19th Int'l Conf. on Very Large Data Bases (VLDB)*, pages 641–654, Dublin, Ireland, 1993.

[FLM98] D. Florescu, A. Y. Levy, and A. Mendelzon. "Database Techniques for the World-Wide Web: A Survery". *ACM The SIGMOD Record*, 1998.

[GG98] P. Godfrey and J. Gryz. "Answering Queries by Semantic Caches". *Submitted for review*, 1998.

[GGM96] P. Godfrey, J. Gryz, and J. Minker. "Semantic Query Optimization for Bottom-Up Evaluation. In *Proc. of the 9th Int'l Symp. on Methodologies for Intelligent Systems (ISMIS)*, pages 561–571, 1996.

[GMHI+95] H. García-Molina, J. Hammer, K. Ireland, Y. Papakonstantinou, J. D. Ullman, and J. Widom. "Integrating and Accessing Heterogeneous Information Sources in TSIMMIS". In *AAAI Spring Symp. on Information Gathering*, 1995.

[GSW96] S. Guo, W. Sun, and M. A. Weiss. "Solving Satisfiability and Implication Problems in Database Systems". *ACM Trans. on Database Systems (TODS)*, 21(2):270–293, jun 1996.

[KB96] A. M. Keller and J. Basu. "A Predicate-based Caching Scheme for Client-Server Database Architectures". *The VLDB Journal*, 5(1):35–47, January 1996.

[LC98] D. Lee and W. W. Chu. "Conjunctive Point Predicate-based Semantic Caching for Wrappers in Web Database". In *ACM Int'l Workshop on Web Information and Data Management (WIDM'98)*, Washington DC, USA, November 1998.

[LMSS95] A. Y. Levy, A. O. Mendelzon, Y. Sagiv, and D. Srivastava. "Answering Queries Using Views". In *ACM SIGACT-SIGMOD-SIGART Symp. on Principles of Database Systems, (PODS)*, volume 14, pages 95–104, San Jose, California, 1995.

[LY85] P.-Å. Larson and H. Z. Yang. "Computing Queries from Derived Relations". In *Proc. of the 11th Int'l Conf. on Very Large Data Bases (VLDB)*, pages 259–269, Stockholm, Sweden, August 1985.

[RD98] Q. Ren and M. H. Dunham. "Semantic Caching and Query Processing". In *Southern Methodist University, Dept. of Computer Science and Engineering, Technical Report 98-CSE-04*, May 1998.

[Sel88] T. Sellis. "Intelligent Caching and Indexing Techniques For Relational Database Systems". *Information Systems (IS)*, 13(2):175–185, 1988.

[Ull89] J. D. Ullman. *"Principles of Database and Knowledge-Base Systems. Volume II: The New Technologies"*. Computer Science Press, 1989.

[VP97] V. Vassalos and Y. Papakonstantinou. "Describing and Using Query Capabilities of Heterogeneous Sources". In *Proc. of the 23rd Int'l Conf. on Very Large Data Bases (VLDB)*, pages 256–265, 1997.

[Zip49] G. K. Zipf. *"Human Behaviour and the Principle of Least Effot"*. Addison-Wesley, 1949.