Semantic Caching via Query Matching for Web Sources*

Dongwon Lee **Wesley W. Chu**

Department of Computer Science
University of California, Los Angeles
Los Angeles, CA 90095, USA

Email: {dongwon,wwc}@cs.ucla.edu

Abstract

A semantic caching scheme suitable for wrappers wrapping web sources is presented. Since the web sources have typically weaker querying capabilities than conventional databases, existing semantic caching schemes cannot be applied directly. A seamlessly integrated query translation and capability mapping between the wrappers and web sources in semantic caching is described. In addition, an analysis on the match types between the user's input query and cached queries is presented. Semantic knowledge acquired from the data can be used to avoid unnecessary access to the web sources by transforming the cache miss to the cache hit. A polynomial time algorithm based on the proposed query matching technique is presented to find the best matched query in the cache. Experimental results reveal the effectiveness of the proposed semantic caching scheme.

1 Introduction

Web databases allow users to pose queries to distributed and heterogeneous web sources. Such systems usually consist of three components [2, 14]: 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 [11], the queries are posed to a uniform interface and submitted to multiple sources at run-time. 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 [1, 10].

Semantic caching (e.g., [9, 16, 20, 21, 22]) 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. The semantic caching can be particularly effective in improving performance when a series of semantically associated queries are asked. Thus, the results may likely overlap or contain one another. Applications such as the cooperative database system [8] and geographical information system are the examples. Semantic caching can be provided for both mediators and wrappers to improve the query response time:

1. Wrapper-level caching: The wrapper has a 1-to-1 mapping with the web source. Since the web source typically has a local view on a "single" relation, queries involve only selection and projection. However, the querying capability between the wrapper and web source needs to be considered.

2. Mediator-level caching: The view at mediator consists of multiple views from wrappers. Thus, complex join operations among multiple views are common at this level.

To a greater or lesser extent, most previous works have focused on mediator-level caching. We argue that wrapper-level caching itself can be useful in a variety of applications. For instance, consider the following example.

Example 1: Let us consider a technical report archive query interface. The query interface is an IR system with form-based fields such as author, title, abstract, organization and identifier, which allow only selection and projection operations. Now suppose the following query is cached locally.

```
SELECT  *
FROM    TechnicalReport
WHERE   title LIKE '%web%'
AND     organization IN {UCB, UCI, UCLA, UCSD}
```

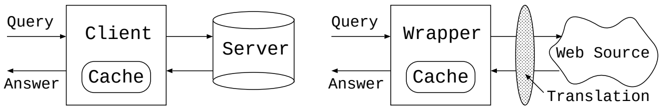
Then, the subsequent query $\mathcal{Q}_1$: "find all TRs whose titles contain 'web' and are from the UCLA archive" can be answered from the cache in its entirety. However, another query $\mathcal{Q}_2$: "find all TRs whose titles contain 'web' and are written by John Smith" cannot be answered from the cache. Furthermore, if we know that Professor 'John Smith' is only with the UCB, then $\mathcal{Q}_2$ can be answered from the local cache. ■

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 gets 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 (i.e., negation) queries.

*This research is supported in part by DARPA contract No. N66001-97-C-8601.

In Proc. 8th ACM Int'l Conf. on Information and Knowledge Management (CIKM), Kansas City, Missouri, USA, November, 1999

Figure 1 illustrates the difference in semantic caching between client-server and web database architectures.



(a) Client caching (b) Wrapper-level caching

Figure 1: Semantic caching in two architectures.

Our proposed semantic caching scheme is based upon the following three key ideas. 1) With an efficient method to locate the best matched query from the set of candidates, semantic caching at wrappers can significantly improve the system performance. 2) Since the querying capability of web sources is weaker than the user's queries, query translation and capability mapping are necessary in semantic caching. 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 for semantic caching. In Section 3, we describe our proposed wrapper-level semantic caching. In Section 4, query matching algorithms are presented. Then, experimental results follow in Section 5. Finally, related works and conclusions are discussed in Section 6 and 7, respectively.

2 Background

Our caching scheme is implemented in a web database testbed called CoWeb (Cooperative Web Database) at UCLA. The architecture consists of a network of mediator and wrapper components [2, 14]. 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 [17], 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 disjoining 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 [15, 24]. 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 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 $\mathcal{Q}_1 \subseteq \mathcal{Q}_2$ [15]. When the operator $\neq$ is added, however, it is shown that the problem becomes NP-complete [6].

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

3 Wrapper-Level Semantic Caching

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 prior query is denoted as *semantic view*, $\mathcal{V}$. 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.

Queries stored in the cache at the wrapper of the CoWeb have the form "`select * from web_source where condition`". By storing all attribute values in the cache, CoWeb completely avoids the *unrecoverability problem* which can occur when query results cannot be recovered from the cache even if they are found due to the lack of certain logical information [12]. For the rationale storing all the attributes instead of only the projected ones, the interested readers should refer to [18]. 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 [11], web sources often provide different query processing capabilities. Therefore, wrapper needs to perform the following pre-processing of the 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 by a disjunctive predicate `(year=1999` $\vee$ `year=2000)` provided that `year` is an integer type.

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*, as it is supported by the web source in a native manner [4]. 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* [4]. 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, the interested readers should refer to [25].

Example 2: Suppose a web source supports queries on `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') ∨ (age=22 ∧ 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 ∧ title='manager'), $\mathcal{V}_2$:(age=21 ∧ title='manager'), and $\mathcal{V}_3$:(age=22 ∧ 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 The Control Flow in the Wrapper with Semantic Cache

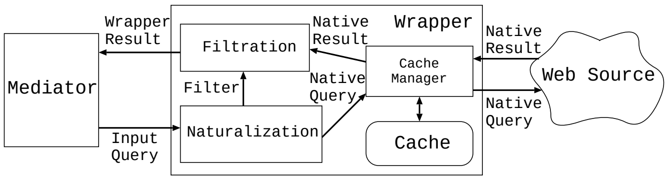


Figure 2: The control flow in the wrapper.

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 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 (i.e., disjunctive predicates are broken into conjunctive ones) is saved in the cache for future use.

3.4 Semantic View Overlapping

A semantic view creates a spatial object[2] in an n-dimensional hyper-space, which creates overlapping. For instance, two queries (10≤age≤20 ∧ 30k≤sal≤40k) and (15≤age≤25 ∧ 35k≤sal≤45k) create an overlapping (15≤age≤20 ∧ 35k ≤sal≤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 [17, 18]. 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 [16, 17].

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 match** since all answers are in the cache, while the overlapping and contained match are **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 matches, computing answers requires the union of the partial answers from the cache and from the web source.

[2] This is called a *semantic region* in [9] and a *semantic segment* in [21].

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*.

4.1 Exact Matching

Traditional caching considers only exact match between an input query and semantic view. 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

In this section, we shall introduce *extended matching* which extends the exact matching for those cases when 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 1).

4.2.1 The `MatchType` Algorithm

First, given a semantic view $\mathcal{V}$ and input query $\mathcal{Q}$, the shown `MatchType` algorithm returns the proper match type using the properties in Table 1. Based on the algorithms developed for solving the satisfiability and containment problem [15, 24], the `MatchType` algorithm can be implemented with $O(|\mathcal{Q}|^2 + |\mathcal{V}|^2)$ complexity.

Input: $\mathcal{Q}$, $\mathcal{V}$ **Output**: Type;

Type ← *null*;
if $\mathcal{Q} \wedge \mathcal{V}$ *is unsatisfiable* **then** Type ← `disjoint_match`;
else if $\mathcal{Q} \equiv \mathcal{V}$ **then** Type ← `exact_match`;
else if $\mathcal{Q} \subseteq \mathcal{V} \wedge \mathcal{Q} \not\supseteq \mathcal{V}$ **then** Type ← `containing_match`;
else if $\mathcal{Q} \not\subseteq \mathcal{V} \wedge \mathcal{Q} \supseteq \mathcal{V}$ **then** Type ← `contained_match`;
else Type ← `overlapping_match`;
return Type;

Algorithm 1: The `MatchType`

4.2.2 The `BestContainingMatch` and `BestContainedMatch` Algorithms

Given an input query and many containing match candidates, the `BestContainingMatch` algorithm finds a containing match that incurs minimal effort to filter out the extra answers. Using *query containment lattice* concept [18], the

	Match Types	Properties	Answers from	
			Cache	Web source
Complete match	Exact match	$\mathcal{V} \equiv \mathcal{Q}$	$\langle\mathcal{V}\rangle$	$\emptyset$
	Containing match	$\mathcal{V} \not\subseteq \mathcal{Q} \wedge \mathcal{Q} \subseteq \mathcal{V}$	$\langle\mathcal{Q}(\langle\mathcal{V}\rangle)\rangle$	$\emptyset$
Partial match	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*	$\emptyset$	$\emptyset$

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

algorithm first finds all minimally-containing matches (the ones that do not contain other containing matches). The main idea here is to find the smallest superset of the input query. For detail discussion, refer to [18]. The algorithm then 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 such heuristics as the number of answers associated with the semantic view, the number of predicate literals in the query, etc.

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;

Algorithm 2: The `BestContainingMatch`

Let $|\mathcal{V}_{max}|$ be the length of the longest containing match and k be the number of the containing matches. Then, the computation takes $O(k^2|\mathcal{V}_{max}|^2)$ 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 large number of heterogeneous web sources or in multimedia databases with expensive operations for image processing. The `BestContainedMatch` algorithm is similar to the case of the `BestContainingMatch` algorithm.

4.2.3 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, partial matches (i.e., overlapping and contained matches) constitute about 40% of all match types for the given test sets (see Table 3). Interestingly, a partial match can be a complete match in certain cases. For instance, for the `employee(name,gender,addr)` 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.

4.3.1 Semantic Knowledge Acquisition

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. 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 [8].

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.

4.3.2 Semantic Knowledge Notation

We use a generic notation derived from [8] 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 $\mathcal{FIND}$ has a form $\sigma_{\mathcal{P}} \diamond \sigma_{\mathcal{Q}}$ where σ is a select operation, $\mathcal{P}$ and $\mathcal{Q}$ are conjunctive WHERE conditions and $\diamond \in \{\equiv, \subseteq\}$. 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., $\Delta = \Delta^*$).

4.3.3 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 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 $\mathcal{Q}$.

Sketch of Proof: Due to limited space, we only show the proof for the second case of the overlapping match transformation here. For the overlapping match, from Table 1, 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. ■

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 3: 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 $\{LHS \subseteq RHS\} \in \Delta$. ■

4.3.4 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 implemented by modifying the `MatchType` algorithm 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 $O(|\mathcal{Q}|^2 + |\mathcal{V}|^2 + |LHS|^2 + |RHS|^2)$. Let $|\mathcal{L}_{max}|$ and $|\mathcal{R}_{max}|$ denote the length of the longest LHS and RHS in Δ and let $|\Delta|$ denote the number of $\mathcal{FIND}$ in Δ, then the 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 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 `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 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 $O(k|\Delta|(|\mathcal{Q}|^2 + |\mathcal{V}_{max}|^2 + |\mathcal{L}_{max}|^2 + |\mathcal{R}_{max}|^2))$ time. Assuming that in general $|\mathcal{V}_{max}|$ is longer than others, the complexity can be simplified to $O(k|\Delta||\mathcal{V}_{max}|^2)$. In addition, the `BestContainingMatch` and `BestContainedMatch` takes at most $O(k^2|\mathcal{V}_{max}|^2)$. Therefore, the total computational complexity of the `BestMatch` algorithm is $O(k|\Delta||\mathcal{V}_{max}|^2) + O(k^2|\mathcal{V}_{max}|^2)$.

Input : $\mathcal{Q}, \mathcal{U}_V \leftarrow \{\mathcal{V}_1, ..., \mathcal{V}_k\}, \Delta$
Output: Best $\leftarrow \mathcal{V}_i \in \mathcal{U}_V$
Best $\leftarrow$ *null*;
Bucket$_{containing}$, Bucket$_{contained}$, Bucket$_{overlapping}$ $\leftarrow \emptyset$;
for $\mathcal{V}_i \leftarrow \mathcal{V}_1$ *to* $\mathcal{V}_k$ **do**
  switch Δ-`MatchType`$(\mathcal{Q}, \mathcal{V}_i, \Delta)$ **do**
    case `exact_match` **return** $\mathcal{V}_i$;
    case `containing_match`
      Bucket$_{containing}$ $\leftarrow$ Bucket$_{containing}$ + $\mathcal{V}_i$;
    case `contained_match`
      Bucket$_{contained}$ $\leftarrow$ Bucket$_{contained}$ + $\mathcal{V}_i$;
    case `overlapping_match`
      Bucket$_{overlapping}$ $\leftarrow$ Bucket$_{overlapping}$ + $\mathcal{V}_i$;
    otherwise skip;
if Bucket$_{containing}$ $\neq \emptyset$ **then**
  Best $\leftarrow$ `BestContainingMatch`(Bucket$_{containing}$);
else if Bucket$_{contained}$ $\neq \emptyset$ **then**
  Best $\leftarrow$ `BestContainedMatch`(Bucket$_{contained}$);
else if Bucket$_{overlapping}$ $\neq \emptyset$ **then**
  Best $\leftarrow$ `BestOverlappingMatch`(Bucket$_{overlapping}$);
return Best;

Algorithm 3: The `BestMatch`

5 Performance Evaluation via Experiments

The experiments were performed on a Sun Sparc 20 machine with 64 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 testbed, CoWeb, was implemented in Java using jdk1.1.7.

5.1 Experimental Setup

A wrapper was constructed to wrap the USAir[3] web site, which provided the following local view to the mediator:

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

Among 7 attributes, both `org` and `dst` were mandatory attributes, thus they should always be bounded in a query. Because of difficulties to obtain real-life test queries from such web source, synthetic test sets with different *semantic localities* were generated to evaluate our semantic caching scheme. Two inputs to the query generator were manipulated to generate different semantic localities in the test sets:

1. The number of the attributes used in the where condition (NUM): A test query with a large number of

[3]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.

attribute conditions is more specific than that of a small number of attribute conditions (e.g., a query (`age=20 ∧ 40k<sal<50k ∧ title='manager'`) is more specific than another query (`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 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). Then, the following input to the query generator $\{N_0$=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 to the query generator {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". For instance, the fact that flight number information is more frequently asked than meal information can be represented by assigning a higher probability value to the `flt` attribute than the `meal` attribute.

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) as shown in Table 2. `uni` and `sem` stand for *uniform* and *semantic* distribution, respectively. Note that `org` and `dst` are the two mandatory attributes. Since there are no queries with no or single attribute conditions, the probabilities for `org` and `dst` are not shown in Table 2 (i.e., they are 100%). The total number of the possible distinct queries that can be generated was set to about 32,400. Each test set with 1,000 queries was randomly picked based on the two inputs. The `sem` values for the input NUM are set to mimic the Zipf distribution [26], 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 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 suffices our purpose to test semantic caching. The following is an example of a typical test query generated.

```
SELECT *
FROM   USAir
WHERE  org='LAX' AND dst='JFK'
AND    87<=flt AND 1<=stp<=2 AND meal='S/S'
AND    aircraft='Boeing 757-200'
```

5.2 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 $1 \le i \le n$. 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 $0 \le c \le 1$ if $M_i = 0$[4]. 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.

5.3 Experimental Results

In Figure 3, 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 type 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 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.

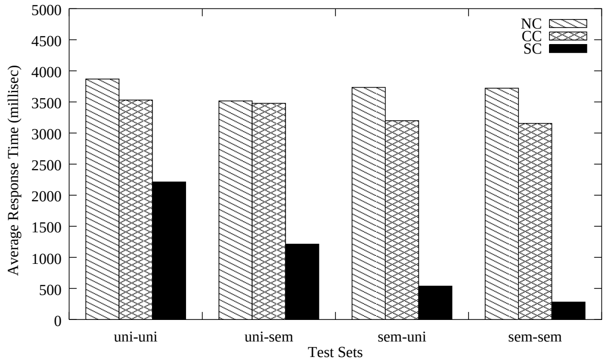


Figure 3: Performance comparison of the semantic caching with conventional caching.

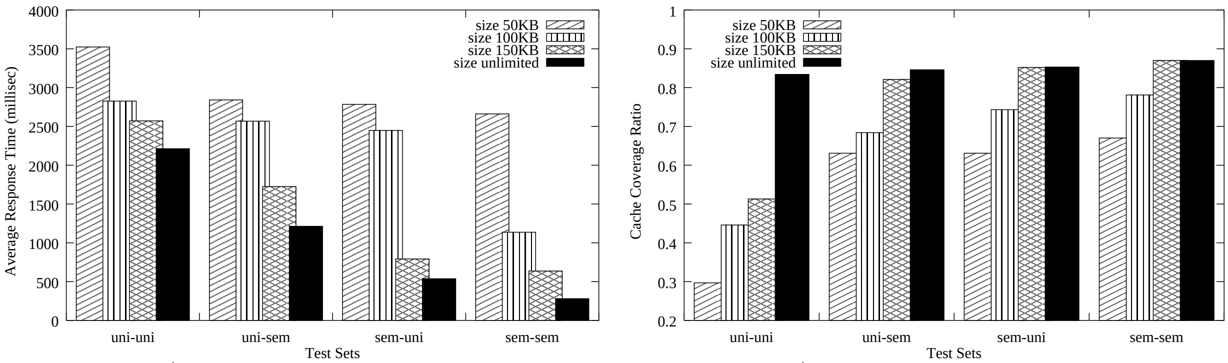
Next, we studied the behaviour of semantic caching with respect to cache size. We set the replacement algorithm as LRU and ran four test sets with cache sizes equal to 50KB, 100KB, 150KB, and unlimited. Because the number of answers returned from the USAir web site is on average small, the cache size was set to be small. Each test set contained 1,000 synthetic queries. Figure 4a and Figure 4b show the $\mathcal{T}$ and the $\mathcal{R}_c$ 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 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)

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

Scheme	Number of the attributes used (NUM)						Scheme	Name of the attributes used (NAME)				
	2	3	4	5	6	7		airline	stp	aircraft	flt	meal
`uni`	16.7%	16.7%	16.7%	16.7%	16.7%	16.7%	`uni`	20%	20%	20%	20%	20%
`sem`	40%	25%	15%	10%	5%	5%	`sem`	40%	25%	10%	5%	20%

Table 2: Uniform and semantic distribution values used for generating the four test sets.



4a) Average response time $\mathcal{T}$ 4b) Cache coverage ratio $\mathcal{R}_c$

Figure 4: Performance comparison of four test sets with selected cache sizes.

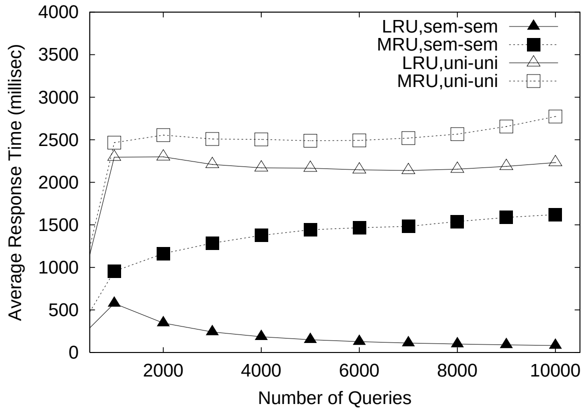
replacement algorithms. The low-level memory management used the strategy that incorporates a reference counter developed in [17]. Due to limited space, we only show `uni-uni` and `sem-sem` test set results. For this comparison, 10,000 synthetic queries were generated in each test set and cache size was fixed to be 150KB. Figure 5a 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 5b shows the $\mathcal{R}_c$ of the two replacement algorithms. Similar to the $\mathcal{T}$ case, LRU 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 `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 3) 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 3 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 6a shows an example of the knowledge-based matching using semantic knowledge. We used a set of induced rules acquired by techniques developed in [8] as semantic knowledge. Figure 6b 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 the semantic views. For instance, a size of 100% means that the number of the induced rules used as semantic knowledge equals to the number of the semantic views in the cache. Despite a large number of partial matches in the `uni-uni` and `uni-sem` sets shown in Table 3, 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 fail to match the rules. Predictably, the effectiveness of the knowledge-based matching depends on the size of the semantic knowledge.

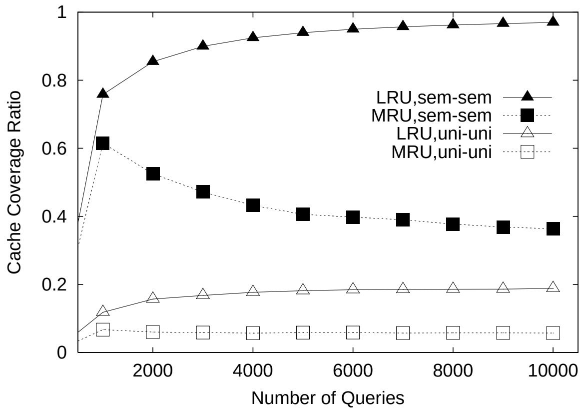
6 Related Works

Past research areas related to semantic caching includes conventional caching (e.g., [1, 10]), query satisfiability and containment problems (e.g., [15, 24]), view materialization (e.g., [19, 20]), query folding (e.g., [23]), and semantic query optimization (e.g., [8, 13]). Recently, semantic caching in client-server or multi-database architecture has received attention [3, 9, 12, 16, 21]. Deciding whether a query is answerable or not is closely related to the problem of finding complete rewritings of a query using views [19, 23]. 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 [5]. 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.

Semantic caching and the corresponding indexing techniques which require that the cached results be exactly matched with the input query are presented in [22]. In our approach, the cached results do not have to be exactly matched with the input query in order to compute answers. [7] approaches the semantic caching from the query planning and optimization point of view. [9] maintains cache space by coalescing or splitting the semantic regions while we maintain cache space by reference counters which allows overlapping in the



5a) Average response time $\mathcal{T}$



5b) Cache coverage ratio $\mathcal{R}_c$

Figure 5: Performance comparison of four test sets with LRU and MRU replacement algorithms.

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: Distribution of match types for four test sets.

semantic regions. Further, we provide techniques to find the best matched query under different circumstances via extended and knowledge-based matching. In [16], *predicate descriptions* derived from previous queries are used to match an input query with the emphasis on updates in the client-server environment.

In [3], 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. [21] defines a semantic caching formally and addresses query processing techniques derived from [20]. A comprehensive formal framework for semantic caching is introduced in [12] illustrating issues, such as when answers are in the cache, when answers in the cache can be recovered, etc. [2] discusses semantic caching in the mediator environment with knowledge called *invariants*. Although invariants are more powerful than $\mathcal{FIND}$ due to their support of arbitrary user-defined functions as conditions, they are mainly used for substituting a domain call. On the contrary, $\mathcal{FIND}$ is simpler and easier in expressing a fragment containment relationship on relations. $\mathcal{FIND}$ can also be acquired (semi)-automatically.

7 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 pattern, and as the degree of the semantic locality increases in the test queries. Finally, 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 perfmrance of the wrappers wrapping web sources.

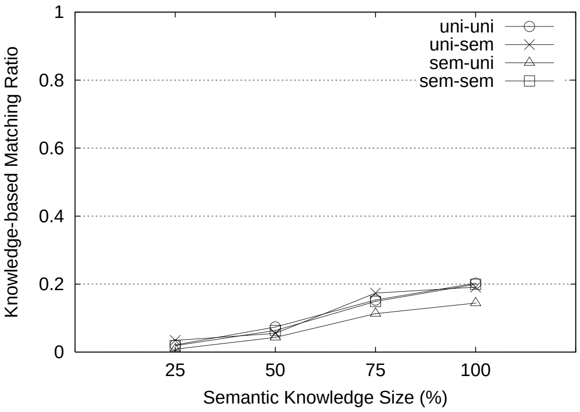
Semantic caching at the mediator-level requires communication with multiple wrappers and creates horizontal and vertical partitions as well as joins of input queries [12], which result in more complicated cache matching. Further research in this area is needed. Other cache issues that were not covered 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.

Acknowledgment: The authors wish to thank Parke Godfrey (U. Maryland), H. Jean Oh (USC/ISI), Frank Meng

```
Q: (org='LAX' ∧ dst='JFK' ∧ 87≤flt≤88)
V: (aircraft='Boeing 757-200')
```

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

6a) User asks flight schedule from LAX to JFK with a range-specified flight numbers. Semantic cache has only overlapping match $\mathcal{V}$. Based on the induced rule 2, $\mathcal{V}$ is transformed into a containing match.



6b) Knowledge-based matching ratio

Figure 6: Performance comparison of the knowledge-based matching.

(UCLA) and Cyrus Shahabi (USC) for their helpful comments during the writing of this paper.

References

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

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

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

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

[5] S. Cluet, O. Kapitskaia, D. Srivastava, "Using LDAP Directory Caches" Proc. *ACM PODS*, 1999.

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

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

[8] W. W. Chu, Q. Chen, A. Huang, "Query Answering via Cooperative Data Inference", JIIS, (3):57-87, 1994.

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

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

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

[12] P. Godfrey, J. Gryz, "Answering Queries by Semantic Caches", Proc. *DEXA*, 1999.

[13] P. Godfrey, J. Gryz, J. Minker, "Semantic Query Optimization for Bottom-Up Evaluation", Proc. *ISMIS*, 561-571, 1996

[14] H. García-Molina, J. Hammer, et el., "Integrating and Accessing Heterogeneous Information Sources in TSIMMIS", Proc. *AAAI Symp. on Information Gathering*, 1995.

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

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

[17] D. Lee, W. W. Chu, "Conjunctive Point Predicate-based Semantic Caching for Wrappers in Web Database", Proc. *Int'l Workshop on Web Information and Data Management (WIDM)*, 1998.

[18] D. Lee, W. W. Chu, "A Semantic Caching Scheme for Web Sources", UCLA-CS-TR-990004, 1999.

[19] A. Y. Levy, A. O. Mendelzon, Y. Sagiv, D. Srivastava, "Answering Queries Using Views", Proc. *ACM PODS*, 1995.

[20] P.-Å. Larson, H. Z. Yang, "Computing Queries from Derived Relations", Proc. *VLDB*, 1985.

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

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

[23] X. Qian, "Query Folding", Proc. *IEEE ICDE*, 1996.

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

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

[26] G. K. Zipf, "Human Behaviour and the Principle of Least Effort", Addison-Wesley, 1949.