Conjunctive Point Predicate-based Semantic Caching for Wrappers in Web Database*

Dongwon Lee and Wesley W. Chu

Department of Computer Science
University of California, Los Angeles
{dongwon,wwc}@cs.ucla.edu

Abstract

A semantic caching scheme suitable for web database environments is proposed. In our scheme, tasks for query translation/capability mapping (named as *query naturalization*) between wrappers and web sources and tasks for semantic caching are seamlessly integrated, resulting in easier query optimization. A semantic cache consists of three components: 1) *semantic view*, a description of the contents in the cache using sub-expressions of the previous queries, 2) *semantic index*, an index for the tuple IDs that satisfy the semantic view, and 3) *physical storage*, a storage containing the tuples (or objects) that are shared by all semantic views in the cache. Types of matching between the native query and cache query are discussed. Algorithms for finding the optimal match of the input query in semantic cache and for cache replacement are presented. The proposed techniques are being implemented in a cooperative web database (CoWeb) prototype at UCLA.

1 Introduction

Two techniques for implementing web database systems (hereafter WebDB) are the *warehousing* and the *virtual* approaches [8]. In the *warehousing* approach, data from multiple sources are prefetched into a local repository and queries are applied to the repository, making query response fast and reliable with the risk of obsolete data. In the *virtual* approach, queries are posed to a uniform interface, which decomposes and applies queries to multiple sources at run time. Querying can be costly due to run-time costs or can not be answered if the server is unavailable. However, this approach always provides up-to-date data.

Given the vast amount of web sources and their autonomous nature, an effective way to reduce costs in using the *virtual* approach is to cache the results of the prior queries and reuse them. When a series of semantically associated queries are asked, the results may likely overlap or contain one another. Caching can be effective in improving the performance for such cases, which often occur in applications like the cooperative database system [4], associative query answering system [5], and geographical information system.

Although caching techniques in distributed systems have been extensively investigated in literature (e.g., [1, 7]), they are not directly applicable to WebDB because of the WebDB constraints. For instance, due to the autonomous nature of the web source, server-side caching is not feasible. Also, since a unique key for a tuple (or object) is not always provided, pointer caching or page caching is not applicable.

In this paper, we shall propose a novel caching scheme suitable for the *virtual* approach. Our scheme sends data to the client (wrapper) side and stores them locally. The cache uses the sub-expressions of the previous queries as semantic keys and avoids unnecessary accesses to the web sources at run time if possible[1].

2 Background

CoWeb (Cooperative Web Database) is a WebDB being developed at UCLA. The architecture consists of the mediator and wrapper components shown in Figure 1. The focus of the system is to use knowledge for providing cooperative capabilities such as conceptual and approximate web query answering, semantic caching, and web triggering with fuzzy threshold conditions.

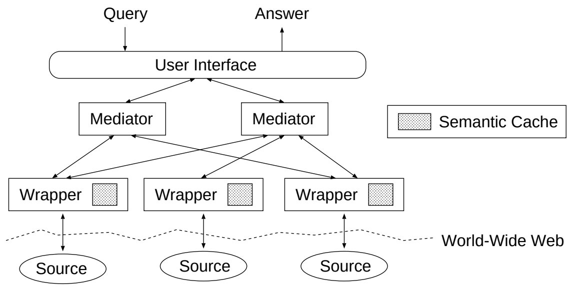

Figure 1: CoWeb Architecture.

The sources are web-based Information Retrieval (IR) systems that use form-based templates with *vector space model*. The majority of the web sources (called *hidden web* in [8]) are accessible through such templates, which typically

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

[1] Issues regarding memory/disk-based caching and consistency maintainence are not covered in this paper.

In Proc. ACM-CIKM Workshop on Web Information and Data Management (WIDM'98), Washington DC, USA, November 6, 1998

have the form of "field operator value", where the operator is either equality or approximation. Note that some of the range or set operators can be transformed into equality or approximation operators. For instance, *(year < 1998)* can be interpreted as *(year_less_than = 1998)*. The predicate with equality or approximation operators is denoted as a *point predicate*. Then, the wrapper in CoWeb submits only queries consisting of the conjunctions of *point predicates* to the web source.

CoWeb uses the *Global As View* approach [8] which specifies mediator schema in accordance with the web source schema. The input query is expressed in the SQL language based on the mediator schema. The mediator decomposes the input SQL into sub-queries for the wrappers by converting the WHERE clause into the disjunctive normal form (DNF, the logical OR of the logical AND clauses) and disjoining conjunctive clauses. Using the SQL syntax, point predicates in the conjunctive clauses use only {=, LIKE} operators. Each conjunctive clause is executed by the wrapper as a separate thread and their results are merged together at the end. For a detailed discussion, refer to [12].

3 Query Naturalization

Web sources use different terminologies and also have different query processing capabilities due to security or performance concerns, etc. [8]. Therefore, the wrapper needs to preprocess the input query before submitting it to the web source.

`Translation` : to provide one-to-one mapping between the wrapper and web source, the wrapper needs to schematically *translate* the input query. We assume that such mechanisms are available (e.g., [10]).

`Augmentation & Filtration` : when there is no one-to-one mapping, the wrapper can *augment* the input query to return more results than requested and filter out the extra data. For instance, if we use the relation `Child(group,name,gender,age)`, then a predicate *(name = 'sylvie')* can be augmented to the *(name LIKE 'sylvie')* with the additional filter *(name = 'sylvie')*.

`Simulation` : when there is no one-to-one mapping, the wrapper can *simulate* the input query with multiple queries and then merge the results. For instance, a *range* predicate $(1 < x < 4)$ can be simulated by a disjunctive predicate $(x = 2 \vee x = 3)$ provided that x is an integer type. In CoWeb, all incoming *range* and *negation* queries are converted into *point* and *positive* queries.

The original query from the mediator is called an *Input Query (IQ)*. The generated query after preprocessing the input query is called a *Native Query (NQ)*. Such a preprocessing task is called *Query Naturalization*. The query used to filter out the irrelevant data from the native query results is called a *Filter Query (FQ)* [3]. While IQ can have range and negative predicates since it is not a naturalized query, NQ can have only point and positive predicates since it is a naturalized query.

4 Semantic Caching

The semantic cache in CoWeb has mainly three components: 1) *semantic view*, a description of the contents in the cache using sub-expressions of the previous queries, 2) *semantic index*, an index for the tuple IDs that satisfy the semantic view, and 3) *physical storage*, a storage containing the tuples (or objects) that are shared by all semantic views in the cache. The *semantic view* in the cache is also called *Cache Query (CQ)* as opposed to IQ, NQ and FQ. Given a database D and query Q, applying query Q on the relation D is denoted as $Q(D)$ and the relation obtained by evaluating the query Q on D is denoted as $\langle Q(D) \rangle$, or $\langle Q \rangle$ for short. Accordingly, an entry in the cache shall be denoted as (V, $\langle V \rangle$), where V is a semantic view and $\langle V \rangle$ is a physical storage containing tuples satisfying the V.

4.1 Query types for caching: input query vs. native query

There are two choices for caching the data: IQ (input query) vs. NQ (native query). The main difference is that the former contains *range* and *negative* operators, whereas the latter contains only *point* and *positive* operators and yields smaller granualities. If the IQ is augmented with an NQ and FQ pair during query naturalization, then $\langle NQ \rangle \supset \langle IQ \rangle$ since $\langle IQ \rangle \equiv \langle NQ \rangle \wedge \langle FQ \rangle$. Thus if the cache stores only the (IQ, $\langle IQ \rangle$) pair, it loses augmented data contained in $\langle NQ \rangle \wedge \neg \langle IQ \rangle$. Since query augmentation and filtration occurs frequently, it is preferable to retain the entire set by using (NQ, $\langle NQ \rangle$) rather than to retain parts of the set by choosing (IQ, $\langle IQ \rangle$). If an NQ has a disjunction of conjunctive clauses, then each conjunctive clause should be decomposed and stored separately. For instance, after *IQ: name = 'sylvie'* $\wedge$ *age IN (11, 23, 35)* is naturalized to *NQ: (name LIKE 'sylvie'* $\wedge$ *age = 11)* $\vee$ *(name LIKE 'sylvie'* $\wedge$ *age = 23)* $\vee$ *(name LIKE 'sylvie'* $\wedge$ *age = 35)*, the following three entries, instead of the original IQ or generated NQ, are decomposed from the NQ and added to the cache: 1) *name LIKE 'sylvie'* $\wedge$ *age = 11*, 2) *name LIKE 'sylvie'* $\wedge$ *age = 23*, and 3) *name LIKE 'sylvie'* $\wedge$ *age = 35*.

4.2 Matching native query with cache query

To obtain answers from the cache, we need to find semantic views which are "exactly" the same as or a "superset" of the given query. Although semantic views are described by conjunctive point predicates for a single relation, because of the missing attributes in a query, *query containment (subsumption) problems* arise. For example, a predicate $(x = 1 \wedge y = 2)$ is contained in $(x = 1)$ since the attribute y is missing in the latter predicate.

To simplify query matching, we normalize all queries to have the same dimension and order as the attributes in the relation. All predicates in the NQ (native query) and CQ (cache query) are: 1) sorted in some pre-determined order and 2) padded with a special *don't care* predicate, $\bigotimes$, for the missing attributes. For instance, based on the attribute order in the `Child` relation, a predicate *(name LIKE 'sylvie'* $\wedge$ *gender = 'girl')* is translated to *(*$\bigotimes$ $\wedge$ *name LIKE 'sylvie'* $\wedge$ *gender = 'girl'* $\wedge$ $\bigotimes$*)* in the query matching algorithm.

Definition: Given two queries, Q_1 and Q_2, if $\langle Q_1 \rangle \subset \langle Q_2 \rangle$, then the query Q_1 is *contained* in the query Q_2 and denoted as $Q_1 \subseteq Q_2$. If two queries *contain* each other, they are *equivalent* and denoted as $Q_1 \equiv Q_2$. When a CQ is *equivalent* to an NQ, the CQ is an *exact match* of the NQ, $\mathcal{M}(\mathcal{E}; NQ)$. When a CQ *contains* an NQ, the CQ is a *containing match* of the NQ, $\mathcal{M}(\mathcal{C}; NQ)$. In contrast, when a CQ is *contained* in an NQ, the CQ is a *contained match* of the NQ, $\mathcal{M}(\neg\mathcal{C}; NQ)$. When a CQ does not contain, but intersects with an NQ, the CQ is an *overlapping match* of

Match Types	Properties
$\mathcal{M}(\mathcal{E};NQ)$	$CQ \equiv NQ \Longrightarrow CQ = \mathcal{M}(\mathcal{E};NQ)$, $answer = \langle CQ \rangle$
$\mathcal{M}(\mathcal{C};NQ)$	$CQ \supseteq NQ \Longrightarrow CQ = \mathcal{M}(\mathcal{C};NQ)$, $answer = \langle NQ(\langle CQ \rangle) \rangle$
$\mathcal{M}(\neg\mathcal{C};NQ)$	$CQ \subseteq NQ \Longrightarrow CQ = \mathcal{M}(\neg\mathcal{C};NQ)$, $answer = \langle CQ \rangle \cup \langle NQ \wedge \neg CQ \rangle$,
$\mathcal{M}(\mathcal{O};NQ)$	$CQ \not\subseteq NQ$, $CQ \not\supseteq NQ$, $\langle CQ \rangle \cap \langle NQ \rangle \neq \emptyset$ $\Longrightarrow CQ = \mathcal{M}(\mathcal{O};NQ)$, $answer = \langle NQ(\langle CQ \rangle) \rangle \cup \langle NQ \wedge \neg CQ \rangle$,
$\mathcal{M}(\neg\mathcal{D};NQ)$	$NQ \wedge CQ = \emptyset \Longrightarrow CQ = \mathcal{M}(\mathcal{D};NQ)$, $answer = \emptyset$
$\mathcal{M}(\mathcal{C};NQ)_{min}$	$m \in \{\mathcal{M}(\mathcal{C};NQ)\}$, $\forall_{i \neq j}\{m_i \subseteq m_j, m_i \not\supseteq m_j\}$ $\Longrightarrow m_i = \mathcal{M}(\mathcal{C};NQ)_{min}$, $answer = \langle NQ(\langle m_i \rangle) \rangle$
$\mathcal{M}(\mathcal{C};NQ)_{opt}$	$m' \in \{\mathcal{M}(\mathcal{C};NQ)_{min}\}$, $\forall_{i \neq j}\{cost(NQ \wedge m'_i) < cost(NQ \wedge m'_j)\} \Longrightarrow$ $m'_i = \mathcal{M}(\mathcal{C};NQ)_{opt}$, $answer = \langle NQ(\langle m'_i \rangle) \rangle$
$\mathcal{M}(\mathcal{O};NQ)_{opt}$	$m'' \in \{\mathcal{M}(\mathcal{O};NQ)\}$, $\forall_{i \neq j}\{cost(NQ \wedge m''_i) < cost(NQ \wedge m''_j)\}$ $\Longrightarrow m''_i = \mathcal{M}(\mathcal{O};NQ)_{opt}$, $answer = \langle NQ(\langle m''_i \rangle) \rangle \cup \langle NQ \wedge \neg m''_i \rangle$

Table 1: Query match types and their properties.

the NQ, $\mathcal{M}(\mathcal{O};NQ)$. Finally, when there is no intersection between NQ and CQ, the CQ is a *disjoint match* of the NQ, $\mathcal{M}(\mathcal{D};NQ)$. Furthermore, a *minimally-containing match* of the NQ, $\mathcal{M}(\mathcal{C};NQ)_{min}$, is the $\mathcal{M}(\mathcal{C};NQ)$ which does not contain any other $\mathcal{M}(\mathcal{C};NQ)$. An *optimally-containing match* of the NQ , $\mathcal{M}(\mathcal{C};NQ)_{opt}$, is the $\mathcal{M}(\mathcal{C};NQ)_{min}$ that yields "optimal" cost[2]. Finally, an *optimally-overlapping match* of the NQ, $\mathcal{M}(\mathcal{O};NQ)_{opt}$, is the $\mathcal{M}(\mathcal{O};NQ)$ that yields "optimal" cost. Detailed properties of query match types are shown in Table 1. □

Example: Given an NQ: *(x = 1 ∧ y = 2 ∧ z = 3)* and a cache with 3 entries, CQ_1: *(x = 1 ∧ y = 2)*, CQ_2: *(x = 1 ∧ z = 3)*, CQ_3: *(x = 1)*, CQ_1, CQ_2, and CQ_3 are $\mathcal{M}(\mathcal{C};NQ)$. Because CQ_3 contains both CQ_1 and CQ_2, only CQ_1 and CQ_2 are $\mathcal{M}(\mathcal{C};NQ)_{min}$, If the $cost(CQ_1 \wedge NQ) < cost(CQ_2 \wedge NQ)$, then $\mathcal{M}(\mathcal{C};NQ)_{opt} = CQ_1$. □

4.2.1 Properties of exact and disjoint match

Given an NQ and a CQ, if the CQ is $\mathcal{M}(\mathcal{E};NQ)$, then it has 1) the same operators, and 2) the same values for all sub-predicates. If the CQ is $\mathcal{M}(\mathcal{D};NQ)$, then it has 1) the same operators for all sub-predicates, and 2) different values in one or more of the sub-predicates. Finding $\mathcal{M}(\mathcal{E};NQ)$ in the cache is the best cache hit case. For the $\mathcal{M}(\mathcal{D};NQ)$ case, accessing data from the web source is required.

4.2.2 Properties of containing match

There are two cases when a predicate P_1 contains another predicate P_2: 1) the operator in P_1 contains the operator in P_2 while the value remains the same (e.g., *(name LIKE 'sylvie') ⊇ (name = 'sylvie')*) , or 2) P_1 contains *don't care* predicate (e.g., $\bigotimes \supseteq$ *(name LIKE 'sylvie')*). For two conjunctive predicates, CP_1 and CP_2, if each sub-predicate of CP_1 contains each correspondent of CP_2, then CP_1 contains CP_2. Given n entries in the cache, the maximum computation complexity to find all the $\mathcal{M}(\mathcal{C};NQ)$s is $O(n)$ since each entry needs to be probed at least once. When there are several $\mathcal{M}(\mathcal{C};NQ)$s, we need to find the $\mathcal{M}(\mathcal{C};NQ)$ that yields minimal cost, which is the $\mathcal{M}(\mathcal{C};NQ)_{opt}$. The computation complexity of finding all $\mathcal{M}(\mathcal{C};NQ)_{min}$s in n $\mathcal{M}(\mathcal{C};NQ)$s is $O(n^2)$ and of finding the unique $\mathcal{M}(\mathcal{C};NQ)_{opt}$ in n $\mathcal{M}(\mathcal{C};NQ)_{min}$s is $O(n)$.

4.2.3 Properties of contained and overlapping match

After exact, disjoint, and containing matches are found, the rest of the queries are either *contained* or *overlapping matches*[3]. Unlike others, whether or not two point predicates overlap cannot be determined by algebraic comparison; rather it requires the examination of the tuples belong to the semantic view. For instance, given two 2-dimensional queries Q_1: *(x = 1 ∧ ⨂)* and Q_2: *(⨂ ∧ y = 1)*, we need to intersect the corresponding answers of Q_1 and Q_2 to determine if they are overlapping. There are two methods to handle an *overlapping match*; method 1) based on the properties in Table 1, find the *optimally-overlapping match* and reuse its overlapped answers in the cache and submit modified query, $NQ \wedge \neg\ \mathcal{M}(\mathcal{O};NQ)_{opt}$[4] , to fetch only the missing answers from the web source, or method 2) ignore the overlapped answers in the cache and re-submit the original NQ to the web source. Method 1) has been used in [6], based on the assumption that fetching only missing answers yields a lower cost than fetching the entire query answers. This is not always true in WebDB since negation (¬) in front of the $\mathcal{M}(\mathcal{O};NQ)_{opt}$ is usually a very expensive operation for IR systems. Typically it becomes an "unsafe" operator due to its unbounded nature (e.g., x ≠ 10 is not computable in an infinite integer domain).

There are certain circumstances where an overlapping match can play an important role in query optimization, even without acquiring missing answers. Consider the following three cases: 1) if the user is able to specify the cardinality of the tolerable answer sets (e.g., CoBase [4]) and if the number of the tuples in an overlapping match meets the specifications, then there is no need to retrieve the missing data, 2) when a web search engine returns partial results to the user as soon as they become available, an overlapping match in the cache is helpful in giving the user an illusion of the fast query response, and 3) when an overlapping match is a "semantically" containing match, the missing data can be inferred from the domain knowledge. For instance, an NQ: *(group = 'toddler' ∧ name = 'sylvie')* is semantically contained in a CQ: *(group = 'toddler' ∧ gender = 'girl')* if one knows "sylvie is the girl's name". If the web source supports negation, then CoWeb uses the method 1, else uses the method 2.

4.2.4 Algorithm for finding the optimal match in the cache

Given an NQ and a semantic cache with cache queries, $\{CQ_1, ..., CQ_n\}$, the following algorithm, OPTMATCH, finds the optimal match in the cache. OPTMATCH iterates from CQ_1 to CQ_n in the cache to find the $\mathcal{M}(\mathcal{E};NQ)$ with the matching properties. The $\mathcal{M}(\mathcal{D};NQ)$ is discarded but the $\mathcal{M}(\mathcal{C};NQ)$ and $\mathcal{M}(\mathcal{O};NQ)$ are accumulated into the buckets, $Bucket(C)$ and $Bucket(O)$, respectively. OPTMATCH

[2] The cost may be based on the number of tuples in the *physical storage* of the *minimally-containing match*. For instance, the *minimally-containing match* might have the least number of tuples.

[3] Since a *contained match* is the special case of an *overlapping match*, without loss of generality, we only discuss the *overlapping match* due to limited space.

[4] This is called *remainder* query in [6].

stops and reuses the cached result from the physical storage as soon as $\mathcal{M}(\mathcal{E};NQ)$ is found. When no $\mathcal{M}(\mathcal{E};NQ)$ is found after all iterations and $Bucket(C)$ is not empty, OPTMATCH finds the $\mathcal{M}(\mathcal{C};NQ)_{opt}$ from the $\mathcal{M}(\mathcal{C};NQ)$s in $Bucket(C)$ using the matching properties. Otherwise, OPTMATCH finds $\mathcal{M}(\mathcal{O};NQ)_{opt}$ from the $\mathcal{M}(\mathcal{O};NQ)$s in $Bucket(O)$ using the matching properties. The computation complexity of the OPTMATCH is $O(n^2)$ where finding $\mathcal{M}(\mathcal{C};NQ)_{opt}$ constitutes the major cost.

4.3 Cache Replacement Policy

According to pre-determined evaluation functions (e.g., LRU, semantic distance), the corresponding replacement values (e.g., access order, distance value) are computed and added to the semantic view. Individual tuples stored in the physical storage contain a reference counter to keep track of the number of reference. After the semantic view for replacement has been decided, all tuples belong to the semantic view are found via the semantic index and their reference counters are decremented by 1. The tuples with counter value 0 are removed from the physical storage. The corresponding semantic view and semantic index are then removed from the cache entries. An example is illustrated in Figure 2.

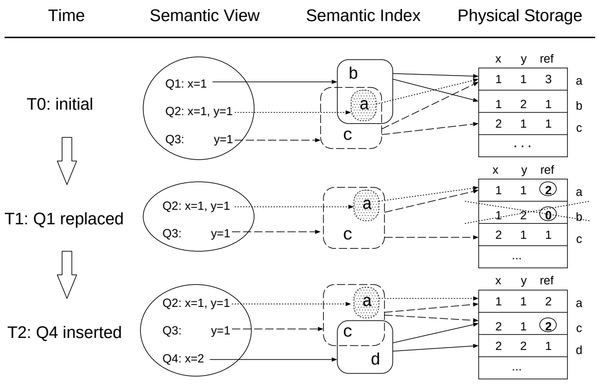

Figure 2: A cache replacement example. When Q_1 is replaced at time $T1$, the corresponding reference counters are decremented and tuple `b` is deleted, but tuple `a` and tuple `c` remain in the physical storage. When Q_4 is inserted at time $T2$, tuple `c` and tuple `d` are inserted to the semantic index, but only tuple `d` is inserted to the physical storage since tuple `c` already exists.

5 Related Work

In [2], selectively chosen sub-queries are stored in cache and treated as information sources in the domain model. The focus in [2] is on how to choose which sub-queries to cache while our focus is on how to find optimal match. [6] introduces the notion of *semantic region* from which *semantic view* and *semantic index* concepts can be derived. [6] maintains cache space efficiently by coaleasing or splitting the semantic regions while we maintains the physical storage efficiently by manipulating reference counters. In [11], *predicate description* derived from previous queries is used to match an input query with the emphasis on updates in the client-server environment. They also use reference counters to reclaim cache space. [9] extends the previous works (e.g., [2, 6, 11]) to a heterogeneous database environment and presents applications where semantic caching is useful.

Our work differs from the prior works in the following aspects: 1) our scheme is suitable for web-based IR applications, 2) we combine semantic caching with query naturalization, which enables us to use *point predicates* to describe specific semantic caching condition, and 3) we provide methods for finding optimal match in the semantic cache.

6 Conclusions & Future Work

Semantic caching techniques for wrappers in web database that utilize query naturalization are presented. We developed algorithms to match cache queries with an input query. Further, we have developed algorithms for cache replacement to maintain cache space efficiently.

We plan to continue investigating the indexing scheme and the *semantic* containment issue. Semantic caching at the mediator-level requires communication with multiple wrappers and creates horizontal and vertical partition of input queries [9]. This results in more complicated cache matching. Further research in that area is needed.

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] Y. Arens and C. A. Knoblock, "Intelligent Caching: Selecting, Representing, and Reusing Data in an Information Server", Proc. *ACM CIKM*, 1994.

[3] C-C. K. Chang, H. García-Molina and A. Paepcke, "Boolean Query Mapping Across Heterogeneous Information Sources", *IEEE TKDE*, 8(4):515-521, 1996.

[4] W. W. Chu, H. Yang, K. Chiang, M. Minock, G. Chow and C. Larson "CoBase: A Scalable and Extensible Cooperative Information System", *JIIS*, 6(11), 1996.

[5] W. W. Chu and G. Zhang, "Associative Query Answering via Query Feature Similarity", Proc. *IIS*, 1997.

[6] S. Dar, M. J. Franklin, B. T. Jonsson and D. Srivastava, "Semantic Data Caching and Replacement", Proc. *VLDB*, 330-341, 1996.

[7] M. J. Franklin, M. J. Carey and M. Livny, "Local Disk Caching for Client-Server Database Systems", Proc. *VLDB*, 641-654, 1993.

[8] D. Florescu, A. Y. Levy and A. Mendelzon, "Database Techniques for the World-Wide Web: A Survery", *ACM SIGMOD Record*, 27(3), 1998.

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

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

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

[12] D. Lee and W. W. Chu, "Conjunctive Point Predicate-based Semantic Caching for Wrappers in Web Database", *UCLA-CS-TR-980030*, 1998.