Conjunctive Point Predicate-based Semantic Caching for Web Databases (Extended Abstract)

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

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

Last Revised: September 17, 1998

Abstract

A novel 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, providing greater levels of query optimization opportunities. 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 storage*", a placeholder for data satisfying the semantic view, and (3) "*physical storage*", a storage containing tuples (or objects) that are shared by all semantic storages in the cache. Because of the inherent characteristics of IR systems, issues similar to the classical query containment problem arise. Possible match types and detailed algorithms for comparing the input query with stored semantic views are discussed. We consider these issues in the context of a prototype web database system being developed at UCLA.

UCLA-CS-TR-980030

1 Introduction

Typical web database systems (hereafter WebDB), which make querying viable on distributed, heterogeneous sources accessible through the web, have mainly three components [Ull97]; (1) mediator, which does a distributed, heterogeneous data integration, (2) wrapper, which does a local translation and data extraction, and (3) web source, which contains raw data to be queried and extracted.

Two techniques for implementing WebDB are the *warehousing* and the *virtual* approaches [FLM98]. 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. Hence, this approach is more appropriate for data which is not recurrently updated or whose size is relatively small to bring in. In the *virtual* approach, queries are posed to an uniform interface, which decomposes and applies queries to multiple sources at run time. Due to accompanying run-time costs, querying could be costly or even unavailable. However this approach always provides up-to-date data. Hence, this approach is more suitable for real-time data (e.g., stock) or data whose size is too big to bring in regularly[1].

Given the vast amount of web sources and their autonomous nature, we believe that developing techniques to ask queries at run time using the *virtual* approach is very important. One of the most effective ways to reduce costs in using the *virtual* approach is to cache the results of the prior queries and reuse them. The effectiveness of the caching is even increased for certain applications, where a series of semantically associated queries are asked in a row, and the results very likely overlap or contain one another. The cooperative database system [CYC+96], the associative query answering system [CZ97], and geographical information system are such examples.

Although caching techniques in distributed systems have been investigated in great detail in literature (e.g., see [ABGM90, FCL93]), they have received little attention in the WebDB context. Moreover, WebDB has more constraints than conventional client-server systems. For instance, due to the lack of the cooperation from the server, server-side caching is not pertinent. Also, since a unique ID for a tuple (or object) is not always guaranteed to be provided, tuple-ID (a.k.a. pointer-caching) or page caching is not applicable.

In this paper, we shall propose a novel caching scheme suitable for WebDB in the *virtual* approach. Our scheme is based on the *client-side*, *data-shipping*, *semantic* approach. Whether the cache is based on main-memory, disk, or both, it is orthogonal to the issues being discussed in this paper and we assume that hereafter cache is only based on the main-memory for illustration purposes. Also the cache coherence maintainence issue is not covered in this paper.

[1] e.g., Microsoft TerraServer at http://terraserver.microsoft.com/

2 Preliminaries

In this section, we explain a few basic concepts and examples to be used throughout the paper. In what follows, we use a relation `Child(group,name,gender,age,height,weight)`, which consists of three groups, {infant, toddler, pre-school}, with their corresponding age limits, 12, 24, 36 months, respectively.

[Definition 2.1] Given a database D, a *Positive Selection Predicate (P^+)* is the predicate with the form of $X\ \Theta\ Y$ in infix notation, where X is an attribute name, Y is an attribute name or a constant value and Θ is any of following SQL operators $\{>, \geq, <, \leq, =,$ *LIKE, IN, BETWEEN*$\}$. A *Negative Selection Predicate (P^-)* is created negating the above Θ by appending an $\neg$ operator. Note that some of the P^+ and P^- operators are identical (e.g., $\neg \leq$ is same as $>$). When the context is clear, we use a *Selection Predicate (P)* to denote both P^+ and P^- for short.

A *Conjunctive Predicate (CONJ)* is the set of Ps connected by only logical AND operators and a *Disjunctive Predicate (DISJ)* is the set of Ps connected by only logical OR operators. Likewise, a *Conjunctive Normal form (CNF)* is the logical AND of the *DISJ*s and a *Disjunctive Normal Form (DNF)* is the logical OR of the *CONJ*s. Formally, with $P_j \in \{P^+, P^-\}$,

$$CONJ = \prod_{j=1}^{n} P_j, \tag{1}$$

$$DISJ = \sum_{j=1}^{n} P_j, \tag{2}$$

$$CNF = \prod_{i=1}^{m}(\sum_{j=1}^{n} P_j) = \prod_{i=1}^{m}(DISJ), \tag{3}$$

$$DNF = \sum_{i=1}^{m}(\prod_{j=1}^{n} P_j) = \prod_{i=1}^{m}(CONJ) \tag{4}$$

□

We will use the DNF form as our basis for query processing because of its simplicity and well-studied semantics. Converting regular boolean predicates into the DNF with unfolded negations while preserving same semantics is a well-known algorithm in the literature [McC86, CGMP96].

[Example 2.1] Given a query *(height < 15 ∨ height > 20) ∧ ¬ (name = 'sylvie')* in CNF, an equivalent DNF and other sub-components are shown in Table 1. □

A central idea behind our semantic caching as well as query capability reconciliation is closely related to the so-called *query containment* problem. It is formally defined as follows.

Type	Predicate
CNF	*(height < 15 ∨ height > 20) ∧ ¬ (name = 'sylvie')*
DNF	*(height < 15 ∧ name ≠ 'sylvie') ∨ (height > 20 ∧ name ≠ 'sylvie')*
CONJ	1: *(height < 10 ∧ name ≠ 'sylvie')*, 2: *(height > 20 ∧ name ≠ 'sylvie')*
P^+	1: *height < 10*, 2: *height > 20*
P^-	*name ≠ 'sylvie'*

Table 1: Predicate types and examples.

[Definition 2.2] Given a database D and query Q, applying Q on D is denoted by $Q(D)$. $\langle Q(D)\rangle$, or $\langle Q\rangle$ for short, is the n-ary relation obtained by evaluating the query Q on D. Given two n-ary queries, Q_1 and Q_2, if $\langle Q_1(D)\rangle \subset \langle Q_2(D)\rangle$ for an arbitrary relation D, then the query Q_1 is *contained* in the query Q_2 and denoted by $Q_1 \subseteq Q_2$. If two queries *contain* each other, they are *equivalent* and denoted by $Q_1 \equiv Q_2$. □

[Example 2.2] For two queries, Q_1: *(height = 20 ∧ weight ≥ 25)* and Q_2: *(height = 20 ∧ weight ≥ 30)*, $Q_1 \supseteq Q_2$ holds because $\langle Q_1(Child)\rangle \supset \langle Q_2(Child)\rangle$. □

[Definition 2.3] Given two *selection predicates*, P_1: *(attr* Θ_1 *val)* and P_2: *(attr* Θ_2 *val)*, which only differ in their operators, if $P_1 \subseteq P_2$, then Θ_1 is *contained* in Θ_2 and denoted by $\Theta_1 \subseteq \Theta_2$. □

[Example 2.3] For two operators, = and *LIKE*, = is *contained* in *LIKE* because *(attr = val)* ⊆ *(attr LIKE val)* for an arbitrary *attr* and *val*. □

3 An Overview of IVY

IVY (Integrated View sYstem) is a WebDB system being developed at UCLA. It has the characteristic architecture of the mediator and wrapper components found in other WebDB systems (e.g., [GMHI+95, KLSS95]), but focuses on narrower application domains. The sources are predominantly Information Retrieval (IR) systems that are accessible through web interfaces. Since the majority of data-intensive web sources (called *hidden web* [FLM98]), are accessible through form-based IR systems on the web, we believe it is a tolerable constraint.

This limitation leads to an important ramification on the design of IVY; since most IR systems support only interfaces based on the vector-space model [Sal89] for simple queries using limited keyword searching[2], web sources support only selection predicates of the form "field operator value", where operator is {=, LIKE}. Note that *range* or *set predicates* are excluded, because the *range* operator (e.g., *year < 1998*) can be treated as a *point* operator (e.g., *year_less_than = 1998*). We shall

[2]Another extreme of the spectrum is using an interface for specifying complex selection criteria in boolean logic (e.g., "intel AND 'Pentium Chip' AND NOT domain:intel.com" for AltaVista search engine). Such case is considered in author's another paper [LSV98].

denote such a selection predicate as a ***point predicate*** to differentiate it from the *range predicate*.

IVY embraces the *Global As View (GAV)* approach [FLM98], explicating mediator schema with respect to the web source schema. The input query is expressed in the SQL[3] language over the mediator schema. From the SQL statement, the mediator spawns sub-queries for wrappers by converting the WHERE clause into DNF and disjoining each CONJ clause. Each CONJ is executed by the wrapper as a separate thread and their results are amalgamated together at the end. In IVY, both `project` (π) and `join` ($\bowtie$) operations are operated at the mediator level while `select` (σ) conditions are pushed down to the wrapper level.

While the mediator-level caching atop the wrapper-level is viable too, here we focus only on the wrapper-level caching. We consider the hybrid caching in both the mediator and wrapper level as one of the important future research directions. Due to the space limitation, hereafter, we only address the wrapper portion of the system.

Upon getting a request in CONJ from the mediator, the wrapper asks the Data Source Manager (DSM) to fetch the requested data. DSM probes in the order of (1) cache, (2) local source, and (3) remote source. The local and remote source components in DSM are placeholders pointing to external sources (denoted by dotted lines in Figure 1). In case of the cache hit, DMS fetches data from the cache. Else if the web source data is brought into a local repository (data warehouse), then DSM fetches data from the local source. If all fails, DSM needs to fetch data from the web source. The purpose of the local source in IVY is twofold; (1) it is a repository for the *data warehousing* approach, containing data directly loaded or subscribed from the web source, and (2) it is an additional buffer containing data overflowed from the cache. The overall architecture of the wrapper is depicted in Figure 1.

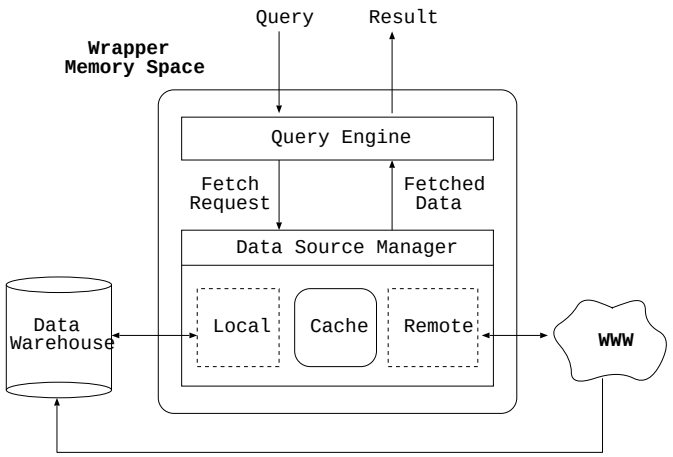

Figure 1: IVY wrapper architecture.

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

4 Query Naturalization

Upon receiving an input query, ***IQ***,[4] from the mediator, the wrapper needs to pre-process it before submitting it to the web source. This is because web sources typically use different terminologies and have different query processing capabilities due to security or performance concern, etc. [FLM98]. Such pre-processing between the wrapper and web sources include:

`Translation` : to provide one-to-one mapping between two parties, the wrapper needs to schematically *translate* the input query. Since we are not addressing schema-level translation, we assume that such mechanisms for performing this task are available. (e.g., [GMHI+95, CGM98]).

`Augmentation` & `Filtration` : when there is no one-to-one mapping, but the web source has a *stronger* (returning more results than requested) query processing capability, the wrapper needs to *augment* the input query. To filter out the extra data, the wrapper also provides filtering information. For instance, a predicate *(name = 'sylvie')* can be *augmented* to the *stronger (name LIKE 'sylvie')* with the additional filter *(name = 'sylvie')*.

`Simulation` : when there is no one-to-one mapping, but the web source has a *weaker* (returning less results than requested) query processing capability, the wrapper needs to *simulate* the input query. 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 IVY, all incoming *range* or *negation* queries are converted into *point* or *positive* queries.

[Definition 4.1] The pre-processing tasks of `translation, augmentation` & `filtration`, and `simulation` are collectively referred to as a *Query Naturalization*. An ***Input Query (IQ)*** is the input request in CONJ from the mediator to the wrapper. A newly generated query after the *query naturalization* is referred to as a ***Native Query (NQ)***, while additional filtering information is referred to as a ***Filter Query (FQ)***[5]. Their relationship is shown in Figure 2. □

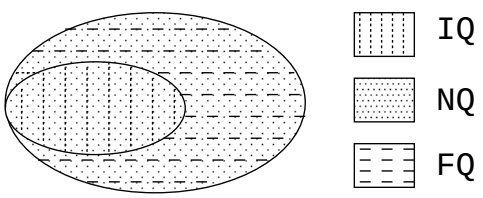

Figure 2: Ven diagram representation of the relationship among IQ, NQ, and FQ. Without a *query naturalization*, NQ would be same as IQ with an empty FQ.

[4] We assume that the original query is *satisfiable* [Cha92] and do not try to check its validity.
[5] We adopt terms, NQ and FQ, from [CGMP96].

attribute	in	out	op	any	domain
`group`	opt	man	=, LIKE	null	set('infant', 'toddler', 'pre-school')
`name`	man	man	LIKE	null	null
`gender`	opt	man	=	Both	set('boy', 'girl')
`age`	opt	man	=	null	list(3, 36, 1)
`height`	opt	man	=	null	interval(1, ∞)
`weight`	opt	man	=	null	interval(1, ∞)

Table 2: $CDV(Child)$: web source should be given binding for the `name` attribute in accepting the input query, but not others. An attribute `gender` has a wildcard value as "Both". Hence, *(gender = 'girl')* can be, for instance, augmented to a *(gender = 'Both')*.

4.1 Query Capability Description

In order for the wrapper to conduct the *query naturalization*, it needs to know additional information with respect to a query processing capability of the target web source. IVY use a simple description vector for this purpose. Its expressive power is much less than that of the [VP97], but equivalent to that of the [LRO96]. Unlike [LRO96] where query capabilities are described for a whole web source, each attribute in IVY carries its own description. Formally,

[Definition 4.2] A *Capability Description Vector (CDV)* is a 5-tuple vector, `(in, out, op, any, domain)`, declaratively describing the query processing capability of the web source. We denote an attribute A's CDV as $CDV(A)$ and a relation R's CDV as $CDV(R)$.

- `in` describes whether the web source must be given binding for this attribute or not – `man` for mandatory, `opt` for optional.
- `out` describes whether this attribute will be shown in the results or not. It has same values as field `in`.
- `op` contains operators, from {=, LIKE}, being supported by the attribute.
- `any` contains a string value to be used as a wildcard.
- `domain` represents the complete domain values of the attribute. Three types – list, interval, and set – are supported. (`list(from, to, gap)`, `interval(from, to)`, and `set(`$\mathtt{val}_1$`, `$\mathtt{val}_2$`, ⋯, `$\mathtt{val}_n$`)`).

□

[Example 4.1] An example of CDV for the `Child` relation is shown in Table 2. Suppose a user wants to find "tall children named sylvie whose birthdays are near", which is in relational algebra $\Pi_*(\sigma_{name='sylvie' \wedge ageIN(11,23,35) \wedge height>30}(Child))$. Then, Table 3 shows each step through the query naturalization process. In the augmentation step, *(name = 'sylvie')* is translated to *(name LIKE 'sylvie')* since the operator = is not supported. Also, although the operator > is not supported for the attribute `height` and since it is an optional attribute, we can gain the same results by dropping the predicate *(height > 30)* completely while providing a right filter. Note that if the upperbound

Input Query (IQ)	*name = 'sylvie' ∧ age IN (11, 23, 35) ∧ height > 30*
After Augmentation	*name LIKE 'sylvie' ∧ age IN (11, 23, 35)*
Filter Query (FQ)	*name = 'sylvie' ∧ height > 30*
After Simulation	*name LIKE 'sylvie' ∧ (age = 11 ∨ age = 23 ∨ age = 35)*
Native Query (NQ)	*(name LIKE 'sylvie' ∧ age = 11) ∨ (name LIKE 'sylvie' ∧ age = 23) ∨ (name LIKE 'sylvie' ∧ age = 35)*

Table 3: A query naturalization example.

of the domain was not ∞, then we would have to compare costs involved in the *augmentation* and the *simulation* and choose the better one. □

5 Semantic Caching

In IVY, semantic caching has mainly three components: (1) "*semantic view*", a description of the contents of the cache using sub-expressions of the previous queries, (2) "*semantic storage*", a placeholder for data satisfying the semantic view, and (3) "*physical storage*", a storage containing tuples (or objects) and shared by all semantic storages in the cache. An entry in the cache shall henceforth be denoted by (V, $\langle V \rangle$), where V is a semantic view and $\langle V \rangle$ is a semantic storage associated with V, excluding physical storage. All such *semantic views* are collectively denoted by a ***Cache Query (CQ)***.

5.1 Caching Choices: IQ vs. NQ

There are two choices for caching the data: IQ (input query) vs. NQ (native query). The main difference is that the former uses *range, negative* queries, whereas the latter uses only *point, positive* (thus much smaller granualities) queries. If IQ is augmented into an NQ and FQ pair during query naturalization, then $\langle NQ \rangle \supset \langle IQ \rangle$ holds 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 happens frequently, we believe it is preferable to retain the whole set by choosing (NQ, $\langle NQ \rangle$) rather than to retain parts of the set by choosing (IQ, $\langle IQ \rangle$). Furthermore, IVY uses CONJ parts of the the NQ in DNF format as cache entries. For instance, after input query *name = 'sylvie' ∧ age IN (11, 23, 35)* is naturalized, three new entries broken from the NQ are added to the cache: (1) *name LIKE 'sylvie' ∧ age = 11*, (2) *name LIKE 'sylvie' ∧ age = 23*, and (3) *name LIKE 'sylvie' ∧ age = 35*, instead of the original IQ: *name = 'sylvie' ∧ age IN (11, 23, 35)*.

5.2 Control Flow

The integrated query naturalization and semantic caching process therefore consist of the four distinct states of the query. Its overall control flow is shown in Figure 3.

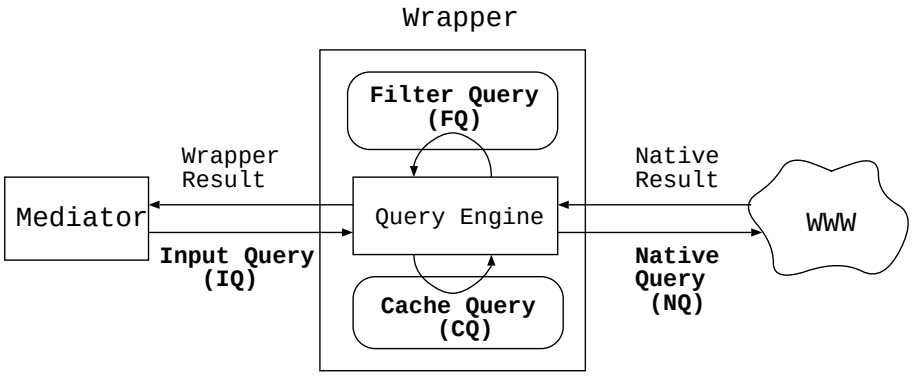

Figure 3: Control flow among four query types (IQ, CQ, NQ, and FQ).

An IQ from the mediator is naturalized by the query engine in the wrapper and converted into a NQ. Doing so, if needed, a FQ is also generated. Then, the query engine (DSM) checks the NQ with previously stored CQs in the semantic cache to find matches. If the match is found, cached results are returned to the mediator. Otherwise, the NQ is submitted to the web source. Upon getting native results from the web source, the wrapper post-processes them by applying the FQ and returns to the mediator. Finally the CONJs of the NQ are broken into pieces and inserted into the cache properly.

5.3 Matching

We want to find semantic views which are "exactly" the same as or a "superset" of the input query so that we can obtain answers from the cache. Although semantic views are described by *point predicates*, issues similar to the classical *query containment problem* arise because of the missing attributes in a query. For example, a predicate $(x = 1 \wedge y = 2)$ is contained in $(x = 1)$ since the attribute y is missing in the later predicate.

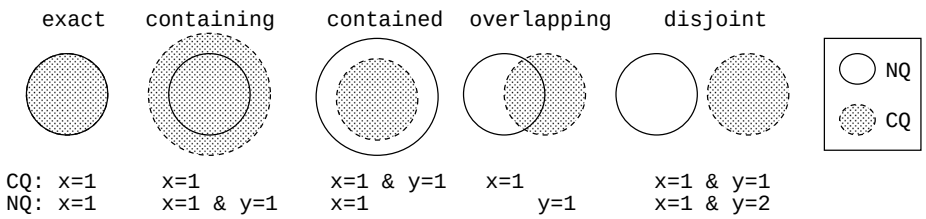

Figure 4: Ven diagram of the five match types and examples with 2-dimensional attributes space (x and y).

Type	Property
CQ^E	$NQ \equiv CQ^E$, *Answer* $\leftarrow \langle CQ^E \rangle$
CQ^C	$NQ \subseteq CQ^C$, *Answer* $\leftarrow NQ(\langle CQ^C \rangle)$
$CQ^{\neg C}$	$NQ \supseteq CQ^{\neg C}$, *Overlapped answer* $\leftarrow \langle CQ^{\neg C} \rangle$, *Missing answer* $\leftarrow \langle NQ \wedge (\neg CQ^{\neg C}) \rangle$, *Answer* $\leftarrow$ *Overlapped answer* $\vee$ *Missing answer*
CQ^O	$(NQ \not\subseteq CQ^O) \wedge (NQ \not\supseteq CQ^O) \wedge (NQ \wedge CQ^O \neq \emptyset)$, *Overlapped answer* $\leftarrow NQ(\langle CQ^O \rangle)$, *Missing answer* $\leftarrow \langle NQ \wedge (\neg CQ^O) \rangle$, *Answer* $\leftarrow$ *Overlapped answer* $\vee$ *Missing answer*
CQ^D	$NQ \wedge CQ^D = \emptyset$, *Answer* $\leftarrow \emptyset$
CQ^C_{min}	$CQ^{C'} \in \{CQ^C\}, CQ^{C''} \in \{CQ^C\}, CQ^{C'} \not\equiv CQ^{C''}$, $CQ^{C'} \subseteq CQ^{C''} \wedge CQ^{C'} \not\supseteq CQ^{C''} \Longrightarrow CQ^C_{min} = CQ^{C'}$, *Answer* $\leftarrow NQ(\langle CQ^C_{min} \rangle)$
CQ^C_{opt}	$CQ^{C'}_{min} \in \{CQ^C_{min}\}, CQ^C_{opt} = opt(cost(NQ \wedge CQ^{C'}_{min}))$, *Answer* $\leftarrow NQ(\langle CQ^C_{opt} \rangle)$
CQ^O_{opt}	$CQ^{O'} \in \{CQ^O\}, CQ^O_{opt} = opt(cost(NQ \wedge CQ^{O'}))$, *Overlapped answer* $\leftarrow NQ(\langle CQ^O_{opt} \rangle)$, *Missing answer* $\leftarrow \langle NQ \wedge (\neg CQ^O_{opt}) \rangle$, *Answer* $\leftarrow$ *Overlapped answer* $\vee$ *Missing answer*

Table 4: Types of varying query matches and their properties.

First we assume that 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, $\bigotimes$, when an attribute is missing in a predicate using the attribute order of that relation. For instance, a predicate *(height = 10 $\wedge$ name LIKE 'sylvie' $\wedge$ gender = 'girl')* can be preprocessed into *($\bigotimes$ $\wedge$ name LIKE 'sylvie' $\wedge$ gender = 'girl' $\wedge$ height = 10 $\wedge$ $\bigotimes$)*. These processings normalize all queries in the same dimensions as the number of the attributes. Thus, 5 different match types are possible between a NQ and a CQ (Figure 4). Formally,

[Definition 5.1] Given two n-ary queries, Q_1 and Q_2, if $\langle Q_1(D) \rangle \subset \langle Q_2(D) \rangle$, then the query Q_1 is *contained* in the query Q_2 and denoted by $Q_1 \subseteq Q_2$. If two queries *contain* each other, they are *equivalent* and denoted by $Q_1 \equiv Q_2$. When a CQ is *equivalent* to the NQ, it is an ***exact match***, CQ^E, of the NQ. When a CQ *contains* the NQ, it is a ***containing match***, CQ^C, of the NQ. In contrast, when a CQ is *contained* in the NQ, it is a ***contained match***, $CQ^{\neg C}$, of the NQ. When a CQ does not contain, but intersects with the NQ, it is an ***overlapping match***, CQ^O, of the NQ. Finally, when there is no intersection between NQ and CQ, the CQ is a ***disjoint match***, CQ^D, of the NQ. Furthermore, a ***minimally-containing match***, CQ^C_{min}, of the NQ is the CQ^C of the NQ, which does not contain any other CQ^C. An ***optimally-containing match***, CQ^C_{opt}, of the NQ is the CQ^C_{min} of the NQ, having "optimal" cost[6]. Finally, an ***optimally-overlapping match***,

[6]By default, the cost may be based on the number of tuples in the *physical storage* of the *minimally-containing*

	=	LIKE
=	$\equiv$	$\subseteq$
LIKE	$\supseteq$	$\equiv$

Table 5: Containment relationship between = and LIKE operators.

CQ^O_{opt}, of the NQ is the CQ^O of the NQ, having "optimal" cost. Detailed properties of each match type are shown in Table 4. □

[Example 5.1] Given a NQ: $(x = 1 \wedge y = 2 \wedge z = 3)$ and a cache with 3 entries, CQ_1: $(x = 1 \wedge y = 2)$, CQ_2: $(x = 1 \wedge z = 3)$, CQ_3: $(x = 1)$, all of CQ_1, CQ_2, and CQ_3 are the *containing* matches of the NQ, but only CQ_1 and CQ_2 are the *minimally-containing* matches, because CQ_3 contains both CQ_1 and CQ_2. If $cost(CQ_1 \wedge NQ) < cost(CQ_2 \wedge NQ)$, then the *optimally-containing match* would be only CQ_1. □

5.3.1 An *exact* and a *disjoint match*

Given an NQ and a CQ, if CQ is to be CQ^E, then it should have the (1) same operators, and (2) same values for all sub-predicates while to be CQ^D, it should have the (1) same operators for all sub-predicates, but (2) different values in one or more of the sub-predicates. Finding CQ^E in the cache is the ideal case of the cache hit and is tested first. In the case of CQ^D, accessing data from the web source is unavoidable. Formally,

[Theorem 5.1] Given two *positive selection predicate* sets with n elements, $\{P_1, ..., P_n\}$ and $\{Q_1, ..., Q_n\}$, if $P_1 \equiv Q_1$, $P_2 \equiv Q_2$, ..., $P_n \equiv Q_n$, and $CONJ_p = P_1 \wedge P_2 \wedge \cdots \wedge P_n$, $CONJ_q = Q_1 \wedge Q_2 \wedge \cdots \wedge Q_n$, then $CONJ_p \equiv CONJ_q$. □

The Theorem 5.1 shows that the *equivalence* property is preserved over the conjunction. Recall that the NQ and the CQ being compared are in CONJ format. Therefore, to find out the *exact* or *disjoint* match, we can compare each sub-predicate of the NQ and CQ successively as shown in Algorithm 1. Given n CQ entries in the cache and k sub-predicates in each CQ (i.e., there are k attributes in the relation on the web source), the worst-case running time to find all the *exact* or *disjoint* matches is $O(nk)$ since each entry in the cache needs to be probed at least once and the function *ExactOrDisjointMatch()* takes $O(k)$. Its correctness stems from the Theorem 5.1 trivially.

Algorithm 1 *ExactOrDisjointMatch(NQ, CQ)*;

INPUT: NQ, CQ
OUTPUT: $Flag \Leftarrow \{Exact, Disjoint, Else\}$;

```
1: P_1 ∧ ... ∧ P_k ⇐ NQ; Q_1 ∧ ... ∧ Q_k ⇐ CQ;
2: Flag ⇐ Exact;
3: for P_i = P_1 to P_k; Q_i = Q_1 to Q_k; do
4:    if P_i.op ≠ Q_i.op then
5:       Flag ⇐ Else; return Flag;
6:    end if
7:    if P_i.val ≠ Q_i.val then
8:       Flag ⇐ Disjoint;
9:    end if
10: end for
11: return Flag;
```

5.3.2 A *containing match*

There are two cases when a predicate P is *contained* in another predicate Q: (1) P's operator is *contained* in Q's operator while the value remains the same (e.g., *(name = 'sylvie')* $\subseteq$ *(name LIKE 'sylvie')*) , or (2) Q is *don't care* (e.g., *(name LIKE 'sylvie')* $\subseteq \otimes$). Furthermore, for two conjunctive predicates, one is *contained* in another as long as each sub-predicate respects the same containment relationship. A relationship among *point* operators that IVY uses is shown at Table 5. Formally,

[Theorem 5.2] Given two *positive selection predicate* sets with n elements, $\{P_1, ..., P_n\}$ and $\{Q_1, ..., Q_n\}$, if $P_1 \subseteq Q_1$, $P_2 \subseteq Q_2$, ..., $P_n \subseteq Q_n$, and $CONJ_p = P_1 \wedge P_2 \wedge \cdots \wedge P_n$, $CONJ_q = Q_1 \wedge Q_2 \wedge \cdots \wedge Q_n$, then $CONJ_p \subseteq CONJ_q$. $\square$

Based on the Theorem 5.2, one can find if a CQ is the CQ^C of the NQ by probing each sub-predicate step by step as shown in Algorithm 2. Given n CQ entries in the cache and k sub-predicates in each CQ, the worst-case running time to find all the CQ^Cs is $O(nk)$ since each entry in the cache needs to be probed at least once and the function *ContainingMatch()* takes $O(k)$.

5.3.3 A *minimally-containing* and an *optimally-containing match*

The *containing*, *minimally-containing*, and *optimally-containing match* can also be described from the lattice-theory point of views.

[Definition 5.2] Suppose $\mathcal{P}$ is a predicate and the set $\mathcal{U}_\mathcal{P}$ corresponds to the set of all possible ground terms for $\mathcal{P}$. Then, an *predicate containment lattice* is defined to be a partially ordered set

match. For instance, the *minimally-containing match* with less number of tuples may be considered to be "optimal" than the one with more number of the tuples.

Algorithm 2 *ContainingMatch(NQ, CQ)*;

INPUT: NQ, CQ
OUTPUT: $Flag \Leftarrow \{Containing, Else\}$;
1: $P_1 \wedge ... \wedge P_k \Leftarrow NQ$; $Q_1 \wedge ... \wedge Q_k \Leftarrow CQ$;
2: $Flag \Leftarrow Containing$;
3: **for** $P_i = P_1$ to P_k; $Q_i = Q_1$ to Q_k; **do**
4: **if** $P_i.op \not\subseteq Q_i.op$ & $Q_i \not\equiv \bigotimes$ **then**
5: $Flag \Leftarrow Else$; return $Flag$;
6: **end if**
7: **end for**
8: return $Flag$;

$\langle \mathcal{P}, \subseteq \rangle$ where the ordering $\subseteq$ forms a semi-lattice over the set $\mathcal{U}_{\mathcal{P}} \cup \{\bot\}$. The special symbol $\bot$ is the unique bottom element of the lattice. □

The greatest lower bound (glb), $\bot$, is same as the conjunction of don't care predicates ($\bigotimes \wedge ... \wedge \bigotimes$). Figure 5 shows an example *predicate containment lattice*. For instance, all predicates under the input query *(x=1 ∧ y=2 ∧ z='C')* at top of the lattice are the containing matches (e.g., *(x=1 ∧ y=2 ∧ z='C')* ⊆ *(z LIKE 'C')*). However, only predicates immediately under the input query are the *minimally-containing matches*.

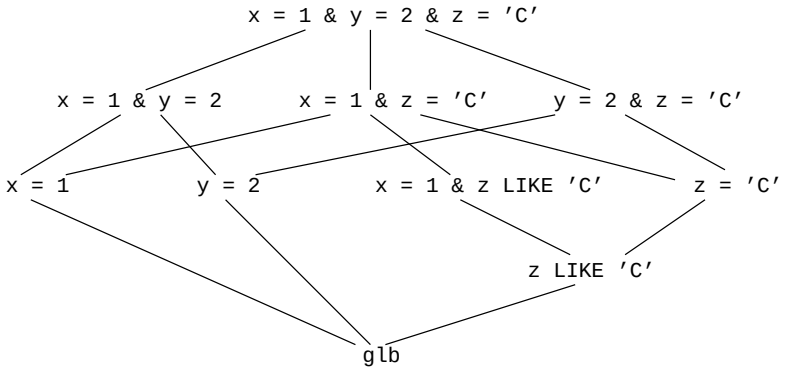

Figure 5: Example *predicate containment lattice*.

When there are several containing matches of the NQ, finding an optimally-containing match is important since it would incur only minimal effort in filtering out the extra data. The Algorithm 3 finds a unique CQ^C_{opt} of the NQ, provided CQ^C set, $\{CQ^C_1, ..., CQ^C_n\}$ by the Algorithm 2. Lines 2 – 9 find all CQ^C_{min}s by eliminating all CQ^Cs containing other CQ^C (thus *not* minimal). Then, lines 10 – 16 iterate through all CQ^C_{min}s and find a CQ^C with minimum cost (thus *optimal*).

In worst-case, if all the entries in the cache are the CQ^C, which is the case in the Algorithm 3, then the nested for loops in lines 2 – 9 costs $O(n^2k)$ running time since the *ContainingMatch()* in line 4 costs $O(k)$ when there are k sub-predicates in CQ and lines 5 – 7 cost only constant time.

Algorithm 3 *OptimallyContainingMatch(NQ, $\{CQ_1^C, ..., CQ_n^C\}$);*

INPUT: $NQ, \{CQ_1^C, ..., CQ_n^C\}$
OUTPUT: CQ_{opt}^C

```
1: BUCKET^C_min ⇐ {CQ^C_1, ..., CQ^C_n};
2: for CQ_i = CQ^C_1 to CQ^C_n do
3:    for CQ_j = CQ^C_1 to CQ^C_n; i ≠ j do
4:       Flag ⇐ ContainingMatch(CQ_i, CQ_j);
5:       if Flag = Containing then
6:          BUCKET^C_min -= CQ_j; {remove a redundant CQ^C}
7:       end if
8:    end for
9: end for{now BUCKET^C_min has only CQ^C_min}
10: CQ^C_opt ⇐ null; cost_min ⇐ ∞;
11: for all CQ_i such that CQ_i ∈ BUCKET^C_min do
12:    cost_i ⇐ NQ ∧ CQ_i;
13:    if cost_i < cost_min then
14:       CQ^C_opt ⇐ CQ_i; cost_min ⇐ cost_i;
15:    end if
16: end for
17: return CQ^C_opt;
```

Again in worst-case, if all the entries in the cache are the CQ_{min}^C (thus $BUCKET_{min}^C$ in line 11 has n elements in it), then lines 11 – 16 take $O(n)$ running time. Therefore, the worst-case running time of the Algorithm 3 is $O(n^2k) + O(n) = O(n^2k)$.

5.3.4 A *contained* and an *overlapping match*

Since a *contained match* is the special case of an *overlapping match*, without loss of generality, we will only discuss the *overlapping match*.

After *exact, disjoint*, and *containing matches* are found, the rest of the queries are either *contained* or *overlapping matches*. Unlike others, whether or not two *point* queries overlap cannot be determined by algebraic comparison; rather, it requires the examination of the tuples stored in the *semantic storage*. For instance, given two 2-dimensional queries Q_1*: (x = 1* $\wedge \otimes$*)* and Q_2*: (*$\otimes \wedge$ *y = 1)*, we need to intersect the corresponding tuples of Q_1 and Q_2 to determine if they are overlapping. To handle an *overlapping match*, we have two strategies;

`Strategy 1.` Based on the properties in Table 4, reuse overlapped answers in the cache and submit a modified NQ to fetch the missing answers from the web source. This strategy has been used in [DFJS96], based on the assumption that fetching missing answers using $NQ \wedge \neg CQ^O$ yields a lower cost than fetching the entire query answer. This is not always true in WebDB context since negation in front of CQ^O is usually a very expensive operation for IR systems.

Typically it becomes an "unsafe" operator due to its unbounded nature (e.g., x $\neq$ 10 is not computable in an infinite integer domain). Also, finding the *optimally-overlapping match* in the Section 5.3.5 becomes important since it would maximize the reuse of the stored data in the cache and minimize costs to fetch the missing data from the web source.

`Strategy 2.` Ignore the overlapped answers in the cache and re-submit the NQ to the web source. Although this is a simple and feasible approach, considering that the *overlapping match* is likely the most frequently-occurring match type in the semantic caching, this significantly decreases the effectiveness of the caching itself.

There are certain circumstances where an *overlapping match* can play an important role in query optimization, even without acquiring missing data [DFJS96, GG97].

- [WZ98] introduces the *early return*, whereby partial results are returned during the computation of user-defined aggregates. The user decides whether to continue the computation or not based on the partial results. The user is also able to specify the cardinality of the tolerable answer sets in CoSQL [CYC+96]. In such an environment, if the number of the tuples in an *overlapping match* meets the specifications, one doesn't need to retrieve missing data.

- Certain web search engines return partial results to the user as soon as they are available, while the remaining results are being computed. In such a user-oriented interactive querying system, fast query response time is crucial and an *overlapping match* in the cache is helpful to give the user an illusion of the fast query response.

- An *overlapping match* can be a "*semantically*" *containing match*. For instance, an NQ: *(group = 'toddler'* $\wedge$ *name = 'sylvie')* is *semantically contained* in a CQ: *(group = 'toddler'* $\wedge$ *gender = 'girl')* if one knows "sylvie is the girl's name".

IVY retrieves answers solely from the overlapping matches for special cases like the above three examples. Otherwise if the web source supports negation, then use the strategy (1), else use the strategy (2).

5.3.5 An *optimally-overlapping match*

Similar to the *optimally-containing match* in the Section 5.3.3, the *optimally-overlapping match* enables to maximize the reuse of the cached overlapping data and can be found using Algorithm 4 in $O(n)$ time.

Algorithm 4 *OptimallyOverlappingMatch(NQ, $\{CQ_1^O, ..., CQ_n^O\}$);*

INPUT: $NQ, \{CQ_1^O, ..., CQ_n^O\}$
OUTPUT: CQ_{opt}^O

```
1: CQ^O_opt ⇐ null; cost_max ⇐ −∞;
2: for CQ_i = CQ^O_1 to CQ^O_n do
3:    cost_i ⇐ NQ ∧ CQ_i;
4:    if cost_i > cost_max then
5:       CQ^O_opt ⇐ CQ_i; cost_max ⇐ cost_i;
6:    end if
7: end for
8: return CQ^O_opt;
```

5.3.6 Combined Algorithm

Algorithm 5 depicts the combined algorithm to compute the input query in the presence of the semantic caching in IVY. Symbolic functions in italic fonts are assumed to be implemented properly. For instance, *MatchType()* function returns a correct type among five different match types and can be easily implemented using the Algorithm 1 and 2.

The input of the algorithm consists of three items; an input query (IQ), a semantic cache (whose semantic views are $\{CQ_1, ..., CQ_n\}$), and a capability description vector (CDV) of the relation. In lines 7 – 18, algorithm iterates through all the entries in the cache and attempts to find the exact match. If there is the exact match, algorithm reuses the cached data. Otherwise, the containing and overlapping matches are accumulated into the coppresponding buckets, $BUCKET^C$ and $BUCKET^O$, respectively. After all the iterations, a containing match case is handled in lines 19 – 21 as described in the Section 5.3.2 and an overlapping match case is handled in lines 22 – 31 as described in the Section 5.3.4.

5.4 Replacement Policy

Semantic storage is the minimum unit for replacement like in [DFJS96]. According to pre-determined evaluation functions such as LRU, the replacement values are calculated and tagged to the entire semantic storage. Individual tuples stored in the physical storage contain a reference counter. After the semantic storage for replacement has been decided, all tuples in the semantic storage are examined and checked by the reference counter. The tuples with counter value 1 (that is pointed to by only one semantic storage) are safely removed from the cache. Otherwise the tuples remain in the cache and their corresponding counters are decremented. Then, the semantic view and semantic storage are removed from the cache entries. This example is depicted in Figure 6.

Algorithm 5 Compute $\langle IQ \rangle$

INPUT: IQ, $\{CQ_1, ..., CQ_n\}$, $CDV(R)$
OUTPUT: $\langle IQ \rangle$

1: $\langle IQ \rangle \Leftarrow \emptyset$; $Flag \Leftarrow null$;
2: $(NQ, FQ) \Leftarrow QueryNaturalization(IQ, CDV(R))$;
3: $NQ_1 \vee \cdots \vee NQ_m \Leftarrow NQ$;
4: $BUCKET^C \Leftarrow BUCKET^O \Leftarrow \emptyset$;
5: **for** $NQ_i = NQ_1$ to NQ_m **do**
6: $\langle NQ_i \rangle \Leftarrow \emptyset$;
7: **for** $CQ_j = CQ_1$ to CQ_n **do**
8: $Flag \Leftarrow MatchType(NQ_i, CQ_j)$;
9: **if** $Flag = Exact$ **then**
10: $\langle NQ_i \rangle \Leftarrow \langle CQ_j \rangle$; **exit** CQ_j loop;
11: **else if** $Flag = Disjoint$ **then**
12: discard CQ_j;
13: **else if** $Flag = Containing$ **then**
14: $BUCKET^C$ += CQ_j;
15: **else** $\{Flag = Overlapping\}$
16: $BUCKET^O$ += CQ_j;
17: **end if**
18: **end for**$\{CQ_j$ loop$\}$
19: **if** $BUCKET^C \neq \emptyset$ **then**
20: $CQ^C_{opt} \Leftarrow OptimallyContainingMatch(NQ_i, BUCKET^C)$;
21: $\langle NQ_i \rangle \Leftarrow NQ_i(\langle CQ^C_{opt} \rangle)$;
22: **else if** $BUCKET^O \neq \emptyset$ **then**
23: $CQ^O_{opt} \Leftarrow OptimallyOverlappingMatch(NQ_i, BUCKET^O)$;
24: **if** *SpecialCases* **then**
25: $\langle NQ_i \rangle \Leftarrow \langle CQ^C_{opt} \rangle$;
26: **else if** *StrategyOne* **then**
27: $\langle NQ_i \rangle \Leftarrow FetchFromSource(NQ_i \wedge \neg CQ^O_{opt})$; $InsertIntoCache(NQ_i, \langle NQ_i \rangle)$;
28: **else** $\{StrategyTwo\}$
29: $\langle NQ_i \rangle \Leftarrow FetchFromSource(NQ_i)$; $InsertIntoCache(NQ_i, \langle NQ_i \rangle)$;
30: ***end if***
31: **end if**
32: $UpdateCacheReplacementValues()$;
33: $\langle NQ \rangle$ += $\langle NQ_i \rangle$;
34: **end for**$\{NQ_i$ loop$\}$
35: $\langle IQ \rangle \Leftarrow FQ(\langle NQ \rangle)$; return $\langle IQ \rangle$;

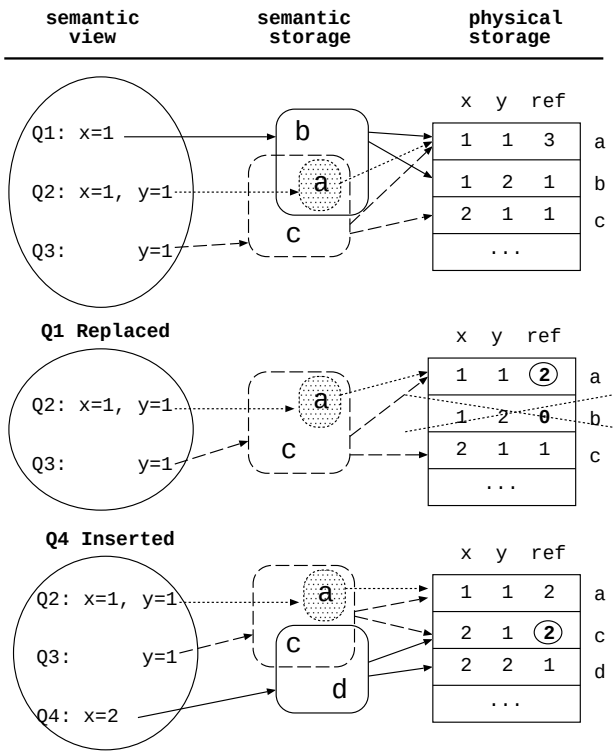

Figure 6: Cache replacement example: When Q_1 is replaced, tuple `b` is deleted, but tuple `a` and tuple `c` remain with the decremented reference counter. Upon Q_4's insertion, tuple `c` and tuple `d` are inserted to the semantic storage, but only tuple `d` is inserted to physical storage. Note that semantic storage can overlap, but not the physical storage. Also, there is no coalesce among overlapping or containing semantic storages.

6 Related Work

Unlike conventional caching schemes (e.g., [ABGM90, FCL93]), semantic caching has received relatively little attention. Among a few previous attempts, our work is more closely related to the [Sel88, AK94, KB96, DFJS96, GG97].

[Sel88] requires the cached results be exactly matched with the input query. [AK94] stores carefully chosen sub-queries in the cache and views them as information sources in the domain model, even including them in query planning. In [KB96], *predicate description* derived from previous queries is used to match an input query with the emphasis on updates in client-server environment. [DFJS96] introduces the notion of *semantic region* from which our *semantic view* and *semantic storage* concepts can be derived. [GG97] extends the previous works to a heterogeneous database environment, but lacks detail.

Our work differs from the prior works in the following aspects; (1) our method is suitable for WebDB, especially for information retrieval applications, (2) we combine semantic caching with *query naturalization*, which enables us to use *point predicates* to describe semantic caching in detail, and

(3) we provide detail analysis on various match types in semantic caching.

Finding proper match types for the input query in the semantic cache is analogous to rewriting input query via materialized view [LY85, GM95, Qia96], or the *query containment* problem [Ull88, Cha92, LMSS95]. The major focus of these works is to reduce the number of database relation literals in the rewritten query using "materialized" view relation literals. Therefore, caching was not their primary concern. Our work more focuses on the *point predicate* containment for a single relation.

7 Conclusion and Future Work

In summary, we have proposed a semantic caching scheme for WebDB, which uses query naturalization. We also discussed issues of finding optimal content matches in semantic caching.

The semantic caching scheme presented in this paper is being implemented and integrated to the fully functioning IVY prototype. We plan to conduct various performance experimentation with varying cost factors. Issues like cache coherence, cache merge/purge, fault-tolerance, etc. need to be investigated further.

Mediator-level semantic caching is more general than that of the wrapper-level, creating complex horizontal and vertical partition of a query as well as complicated cache matching [GG97]. We consider this as one of the important directions for future work.

8 Acknowledgment

Authors wish to thank H. Jean Oh at USC/ISI for her helpful comments regarding initial design.

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.

[AK94] Y. Arens and C. A. Knoblock. "Intelligent Caching: Selecting, Representing, and Reusing Data in an Information Server". In *ACM Proc. of the 3rd Int'l Conf. on Information and Knowledge Management (CIKM)*, 1994.

[CGM98] C-C. K. Chang and H. García-Molina. "Conjunctive Constraint Mapping for Data Translation". In *ACM Proc. of the 3rd Int'l Conf. on Digital Libraries (DL)*, pages 49–58, 1998.

[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.

[Cha92] E. P. F. Chan. "Containment and Minimization of Conjunctive Queries in OODB's". In *ACM SIGACT-SIGMOD Symp. on Principles of Database Systems (PODS)*, San Diego, California, June 1992.

[CYC+96] W. W. Chu, H. Yang, K. Chiang, M. Minock, G. Chow, and C. Larson. "CoBase: A Scalable and Extensible Cooperative Information System". *Journal of Intelligent Information Systems (JIIS)*, 6(11), 1996.

[CZ97] W. W. Chu and G. Zhang. "Associative Query Answering via Query Feature Similarity". In *Proc. of the Int'l Conf. on Intelligent Information Systems (IIS)*, The Bahamas, 1997.

[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.

[GG97] P. Godfrey and J. Gryz. "Semantic Query Caching for Heterogeneous Databases". In *Proc. of the 4th Knowledge Representation Meets Databases Workshop (KRDB)*, Athens, Greece, August 1997.

[GM95] A. Gupta and I. S. Mumick. "Maintenance of Materialized Views: Problems, Techniques, and Applications". In *IEEE Proc. of the 11th Int'l Conf. on Data Engineering (ICDE)*, June 1995.

[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.

[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.

[KLSS95] T. Kirk, A. Y. Levy, Y. Sagiv, and D. Srivastava. "The Information Manifold". In *AAAI Spring Symposium on Information Gathering*, 1995.

[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.

[LRO96] A. Y. Levy, A. Rajaraman, and J. Ordille. "Querying Heterogeneous Information Sources Using Source Descriptions". In *Proc. of the 22nd Int'l Conf. on Very Large Data Bases (VLDB)*, Mumbai (Bombay), India, 1996.

[LSV98] D. Lee, D. Srivastava, and D. Vista. "Generating Advanced Query Interfaces". In *Proc. of the 7th Int'l Conf. of World Wide Web (WWW)*, Australia, April 1998.

[LY85] P.-A. 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.

[McC86] E. J. McCluskey. *"Logic Design Principles"*. Prentice-Hall, 1986.

[Qia96] X. Qian. "Query Folding". In *IEEE Proc. of the 12th Int'l Conf. on Data Engineering (ICDE)*, pages 48–55, February 1996.

[Sal89] G. Salton. *"Automatic Text Processing"*. Addison-Wesley, 1989.

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

[Ull88] J. D. Ullman. *"Principles of Database and Knowledge-Base Systems. Volume II: The New Technologies"*. Computer Science Press, 1988.

[Ull97] J. D. Ullman. "Information Integration Using Logical Views". In *Proc. of the 22nd Int'l Conf. on Database Theory (ICDT)*, Delphi, Greece, 1997.

[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.

[WZ98] H. Wang and C. Zaniolo. "User-Defined Aggregates for Logical Data Languages". In *Proc. of the 6th Int'l Workshop on Deductive Databases and Logic Programming (DDLP)*, Manchester, UK, June 1998.