Answering Queries by Semantic Caches

Parke Godfrey
godfrey@cs.umd.edu
Department of Computer Science
University of Maryland
College Park, MD, USA

Jarek Gryz
jarek@cs.yorku.ca
Department of Computer Science
York University
Toronto, Canada

Abstract

There has been growing interest in *semantic query caches* to aid in query evaluation. Semantic caches are simply the results of previously asked queries, or selected relational information chosen by an evaluation strategy, that have been cached locally. For complex environments such as distributed, heterogeneous databases and data warehousing, the use of semantic caches promises to help optimize query evaluation, increase turnaround for users, and reduce network load and other resource usage. In this paper, we present a general logical framework for semantic caches. We consider the use of all relational operations across the caches for answering queries, and we consider the various possibilities to answer a query (and to partially answer a query) by cache. Specifically, we address when answers are in cache, when answers in cache can be recovered, and the notions of semantic overlap, semantic independence, and semantic query remainder.

While there has been much work relevant to the use of semantic caches, no one has addressed in conjunction the issues pertinent to the effective use of semantic caches to evaluate queries. This has been due in some cases to overly simplified assumptions (for truly effective cache use), and in other cases to the lack of a formal framework. We attempt to establish some of that framework here. Within that framework, we are able to illustrate the issues involved in using semantic caches for query evaluation. We show various applications for semantic caches, relate the work and other areas of study that are relevant, and establish an agenda of what needs to be accomplished to make semantic query caches a viable technology.

1 Introduction

There has been growing interest in *semantic query caches* to aid in query evaluation. Semantic caches are simply the results of previously asked queries cached locally, or selected relational information chosen by a strategy to be cached locally. In complex information environments such as mediation over distributed, heterogeneous databases and data warehousing, the use of semantic caches promises to help optimize query evaluation, increase turnaround for users, and reduce network load and other resource usage.

The concept of caching, of course, is a basic one in computer science. Integrated circuits for central processing units now have built-in high speed memory caches to reduce fetches to main memory. Operating systems employ essentially complex cache strategies to decide which virtual pages to keep in main memory, to reduce fetches to secondary memory (disk). Equivalently, relational database management systems employ buffer management strategies to reduce I/O to disk, thus reusing previously fetched data pages.

In distributed database environments, another layer of caching is possible: to cache information between servers and clients. Two caching approaches, *page caching* and *tuple caching* in which memory pages or tuples are cached, respectively, have been studied in this context [5, 11]. Semantic query caching offers a third approach. In [5], it is shown that semantic caching may generally outperform the page and tuple caching approaches. This is due to *semantic locality*: subsequent queries often are related conceptually with previous queries, so ultimately will be pulling data from the same logical sources. Thus, semantic caches will often contain some of the answers of the current query. In addition, in heterogeneous, distributed environments, it is unclear how page or tuple caching might be adapted. Semantic caching, however, can be well applied in these environments.

In this paper, we present a general logical foundation for semantic query caching. We explore in what ways queries can be answered by semantic caches, and, within this framework, we elucidate numerous issues to be addressed in implementing semantic query caching. We extend the paradigm of semantic caching to consider the use of the caches in composite to answer queries; thus we allow any relational operations across the caches (which are stored locally as relational tables). We call any relational expression across the caches a *cache expression*.

While the notion of a semantic cache itself is quite simple (it is an answer set stored as a relational table, labeled by the query that resulted in the answer set), the use of semantic caches to answer queries can be complex. This is because it requires reasoning over the query and cache formulas to determine how they are related semantically. Current database systems have no such facility to *reason* about the queries that they receive.[1] In order to use semantic query caching, specific tools to reason over cache and query expressions will be needed to determine when the caches can be used to answer, or to partially answer, the query.

Specifically, we address the topics of

1. deciding when answers are in cache,
2. extracting answers from cache,
3. semantic overlap and semantic independence, and
4. semantic remainder.

It is interesting that there are cases when it is possible to determine that the answers of the query are in cache, but there is not enough information locally that the answers can be recovered from cache. (Hence, there is a difference between topics 1 and 2.) Under topics 1 and 2, we consider when a query is contained in the caches, and, conversely, when a cache expression is contained by the query. Under topic 3, we generalize to the case when the query and a cache expression "overlap" somehow, but there is not containment in either direction. Lastly, under topic 4, we consider how *remainder queries* might be found that represent the "rest" of the query that could not be answered by cache. This topic is challenging and not yet well defined. We attempt to provide some insights.

In Section 2, we provide an overview of semantic caches and discuss various of its possible applications. In Section 3, we provide a logical formalism (based on Datalog and the logic model [32]) for semantic query caching, and address each of the topics enumerated above in turn. In Section 4, we discuss related work and topics, and then topics to be addressed. There is much work that is relevant to semantic query caching, and we are only able to provide a brief summary of the work. We relate what issues in semantic caching have been addressed, and which remain open. We conclude in Section 5.

2 Semantic Query Caching

2.1 Overview

We start with an informal description of the possible relationships between a query's answer set and the tuples stored explicitly in, or computable from, the caches. The boxes in Figure 1 abstractly represent relational tables. The clear boxes represent the answer tuples of a cached query, and the shaded boxes represent the answer tuples of a query. The boxes represent relational tables with "rows" (the horizontal) representing

[1] They do *evaluate* queries well, and this, of course, involves certain types of reasoning over the queries. However, by *reasoning* here, we mean the ability to compare queries, and to employ information about what the queries "mean" in support of applications.

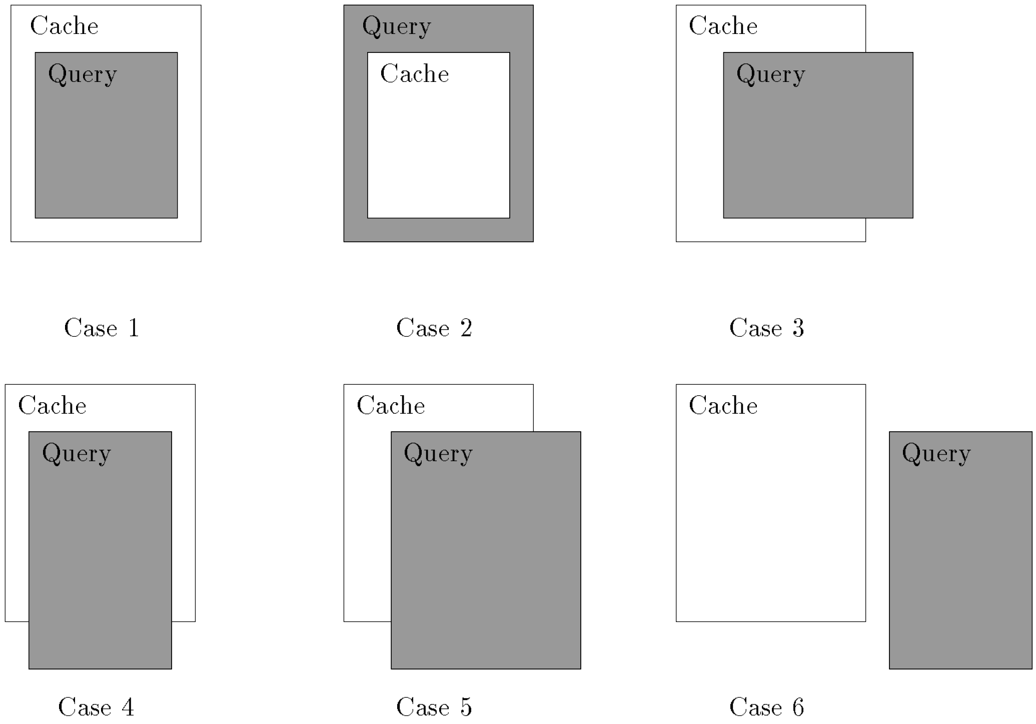

Figure 1: Possible relations between cached and current queries.

tuples and "columns" (the vertical) attributes. Assume initially that the only operations performed on the queries were selects and projects. Then the current query may overlap with a cached query in a number of possible ways. The query may be computable entirely from a cache (case 1) or only partially (cases 2 through 5). Case 3 represents what we call a *vertical* partition of a query: only some of the attributes-of-interest for the query (a projection of the query) are available in cache. If the cache contains the key columns of the relation in the query, the missing columns could be imported from the server, to be joined with the cache locally. Case 4 represents the situation when *some* of the answers to the query are available from cache; we refer to this case as a *horizontal* partition of a query. Case 5 represents a mixed partition of a query.

The scenario above can be generalized so that both caches and queries may involve joins. Then the white boxes in Figure 1 may represent the result of an arbitrary join of *several* caches stored locally. Even in the simplest case (1)—when all columns of interest of the query are in cache—it may be impossible to compute the answer set to the query without more information from the servers. For example, if a cached query $\mathcal{C}$ resulted from the join $R_1 \bowtie ... \bowtie R_n$ and the current query $\mathcal{Q}$ is the join $R_1 \bowtie ... \bowtie R_n \bowtie R_{n+1}$, we would at least need to compute the values from a join column of the relation R_{n+1} to evaluate $\mathcal{Q}$ by cache. Cases 2 through 5 introduce yet another complication: the need to *modify* the query so that only the part of the query that cannot be evaluated by cache is evaluated subsequently.[2] Whether this should be done depends primarily on one's objective: answer set pipelining generally benefits from heavier use of caches than is the case for optimizing the overall query response time.

Of course, one should not be mislead by Figure 1. We are not really interested in how the *answer sets* of queries and caches *actually* overlap (that is, the actual tuples they share in common). Instead, we are interested in (the characterization of) any subsets of answers that the query and cache *must* share. This will be the case if the query and the cache are somehow semantically related. If the query and cache are semantically unrelated, then we will not be able to use the cache to answer any of the query. For instance, consider the two relations $employee(X)$ and $stock\_holder(X)$. Assume these are base relations (not views).

[2] Such a query is called a *remainder query* in [10] or a *trimmed query* in [20].

Then the contents of the *employee* table in no way affects the *stock_holder*table, nor vice versa.[3] The two tables may still share values by happenstance (for instance, *employee*(*john*) and *stock_holder*(*john*)). The only way to determine how a query and cache actually overlap, of course, is to actually evaluate the query.

Two view relations may be defined in part over the same base relations. Thus, if the query employs one of the view relations and the cache the other, then we may be able to determine semantically that they must share answers. Furthermore, we might be able to determine a query expression that is evaluable against the cache that retrieves these answers. This is the topic of this paper.

By expressing queries and caches in a logical formalism, we are able to employ analytical tools developed for logic databases to decide when a cache (or combination thereof) answers, or partially answers, the query [33]. The basic inference needed is containment determination for extensional, conjunctive queries (called *conjunctive query containment* in [32]): we say that the query $\mathcal{F}$ is contained in the query $\mathcal{G}$, if all answers to $\mathcal{F}$—say, a cache—are also answers to $\mathcal{G}$—say, the query. Thus, the containment test alone is sufficient for the simplest case, when a single cache, which is extensional (that is, one that does not refer to views), partially answers the query. SQC with joins, or with queries and caches over views, requires more sophisticated inferencing.

2.2 Applications

Semantic query caching (SQC) can help to address a number of other issues that arise in mediated, distributed environments. We contend that SQC is critical for optimization in heterogeneous, multi-database environments.

- *Query optimization.*
  - *Improvement in overall query response time (traditional optimization).* Since part, or all, of query processing can be done by the client via caches, the workload at the database servers is reduced. If the answer set of a query is large, computing part of it at the client also provides savings in network communication. In addition, as some of the query is evaluated at the client (locally) and the rest is evaluated at the server (or servers), this may be done in parallel, reducing the overall time for evaluation substantially.
  - *Saving money.* In environments where there are monetary charges for information, such as in electronic commerce, caching techniques can be used to optimize over these monetary costs (instead of just for computational cost).
  - *Optimization of queries with few answers.* If the cardinality of the query's answer set can be determined in advance (for instance, that there is only *one* answer) and the number of answers to the query in cache is equivalent to the known cardinality, then the cached answer set can be determined to be complete, without any further work necessary.
  - *Optimization of queries in batch (multiple query optimization).* If a user or application requests the union of answers of a collection of queries, and if the queries are evaluated sequentially, then any part of a subsequent query that can be answered *by cache*—that is, those answers can be determined to have be obtained by previous queries—need not be re-evaluated. Only the parts of subsequent queries that are *semantically independent* of the previous queries need be evaluated.
- *Data Security.* We can limit the shuttling of sensitive data across the network by storing it at the client as caches. Such data does not have to consist of complete tables; it can be defined as parts of tables in the same way views are defined.
- *Fault tolerance.* Some databases may not be accessible at a given time. If a query can be partially computed from caches, at least *some* of the answers can be returned to the user.

[3] In truth, even though these are base relations, there may exist integrity constraint relationships between them. In such cases, one might be able to determine that *employee* semantically affects *stock_holder*. This is beyond the scope of this paper.

- *Approximate answering.* Sometimes a good approximation of aggregate values such as *average* can be obtained from caches. If it can be determined that a cache contains a representative sample of the tuples over which the aggregate function is to be computed, then it can be evaluated just over cache.
- *Better user interaction.*
  - *Answer set pipelining.* A subset of the answers that are computable at the client by cache can be returned to a user *promptly*, while remaining answers are being evaluated.
  - *Indirect answering.* The information that the query is contained in cache may sometimes be all that the user requires. This happens, for example, when in a sequence of queries it can be determined that the next query does not add any new tuples to those previously retrieved.
  - *Limiting the size of the answer set.* In some applications, a user may not be interested in retrieving all answers, but may be satisfied with just some (that is, with just a subset of a complete answer set). It might also be the case that the user might want to terminate the query evaluation if he or she finds that the answer set is larger than expected. In both cases, query processing can be sometimes terminated after retrieving just the answers from cache.

3 Evaluating Queries by Caches

3.1 Logical Notation

We employ the terminology of logic databases and Datalog [26, 32]. A database **DB** is defined as consisting of two parts: the extensional database, $\mathbf{E_{DB}}$, and the intensional database, $\mathbf{I_{DB}}$. The $\mathbf{E_{DB}}$ is the database's collection of *facts*. The $\mathbf{I_{DB}}$ is the database's collection of *rules* (relational views) and, perhaps, *integrity constraints*.[4] We assume that any given predicate is either defined via rules in the $\mathbf{I_{DB}}$ (and, hence, is called *intensional*) or is defined via facts in the $\mathbf{E_{DB}}$ (and, hence, is called *extensional*).

Rules are *Horn clauses*, which are logical sentences of the form:

$$\forall.\ a\langle \vec{x}\,\rangle \vee \neg b_1\langle \vec{x}_1\rangle \vee \ldots \vee \neg b_k\langle \vec{x}_k\rangle \tag{1}$$

in which $a\langle \vec{x}\,\rangle$ and each of $b_i\langle \vec{x}_i\rangle$'s are atomic formulas, and $\vec{x}$ and each $\vec{x}_i$ is shorthand notation for some list of variables and constants, say, $X_1, \ldots, X_n$. The '$\forall$.' is shorthand notation for that all free variables in the formula within its scope are to be universally quantified. The notation '$\exists$.' is likewise for existential quantification. In Datalog, a rule is written in further shorthand as an implication:

$$a\langle \vec{x}\,\rangle \ \leftarrow\ b_1\langle \vec{x}_1\rangle, \ldots, b_k\langle \vec{x}_k\rangle. \tag{2}$$

in which the universal quantification is understood. A *query clause* in Datalog is a clause as in (1), but with no positive atom (so (1) with $a\langle \vec{x}\,\rangle$ removed from the disjunction). It is written as:

$$\leftarrow\ q_1\langle \vec{z}_1\rangle, \ldots, q_k\langle \vec{z}_k\rangle. \tag{3}$$

(in which the $q_i\langle \vec{z}_i\rangle$'s in this case are atomic formulas). This notation is convenient for logic programming systems (such as Prolog) and deductive database systems that find answers to a query by means of refutation proofs [26, 32]. That is, given a query clause $\mathcal{C}$, if $\mathbf{DB} \cup \{\mathcal{C}\}$ is inconsistent, then the query represented by $\mathcal{C}$ has answers. The answers are the witness *groundings* of $\mathcal{C}$ that prove the inconsistency.

We shall find it more convenient to work with queries as conjunctive formulas, and not in "negated" form as with query clauses. We define a *query formula* to be an existentially quantified, conjunctive formula of the form:

[4] We do not consider integrity constraints in this paper.

$$\mathcal{Q}\colon\ \exists \vec{y}\,.\ q_1\langle \vec{z}_1 \rangle \wedge \ldots \wedge q_k\langle \vec{z}_k \rangle. \tag{4}$$

We shall often refer to the query formula simply as the query, when this is understood in context. We refer to free variables in $\mathcal{Q}$ (that is, the variables in $\mathcal{Q}$ but not in $\vec{y}$) as the *distinguished variables* of $\mathcal{Q}$, and the variables in $\vec{y}$ as the *existential variables* of $\mathcal{Q}$. (Note that the query formula is simply the negation of the corresponding query clause, plus an indication of which variables are to be considered distinguished and which are existential.)

We define an *unfolding* of a *query clause* as follows. A 1-*step unfolding* is simply the resolution resolvent of the query clause with a matching rule. This is the standard resolution step as in Prolog [32]. Say, without loss of generality, that for $q_1\langle \vec{z}_1 \rangle$ in (3) and $a\langle \vec{x} \rangle$ in (2), $q\langle \vec{z} \rangle\theta = a\langle \vec{x} \rangle\theta$ with most general unifier θ and that the variables of (2) ($\vec{x}$ and the $\vec{x}_i$'s) and of (3) (the $\vec{z}_i$'s) are appropriately *standardized apart* [32] (so a and q are actually the same predicate here). Then the 1-step unfolding is

$$\leftarrow b_1\langle \vec{x}_1 \rangle\theta,\ \ldots,\ b_k\langle \vec{x}_k \rangle\theta,\ q_2\langle \vec{z}_2 \rangle\theta,\ \ldots,\ q_n\langle \vec{z}_n \rangle\theta.$$

Let a k-*step unfolding* simply be a sequence of k 1-step unfoldings applied sequentially starting with the query clause and ending with the unfolding, for any finite k. We call any k-step unfolding simply an unfolding. Likewise, define the *unfolding* of a *query formula* $\mathcal{Q}$ as the corresponding query formula of any unfolding of the corresponding query clause of $\mathcal{Q}$ (preserving the distinguished variables of $\mathcal{Q}$, and adding any new variables that were introduced in the unfolding as existential variables).

We define an *abbreviated query formula* $\mathcal{Q}'$ of $\mathcal{Q}$ as in formula (4) (also to be called an *abbreviation* of $\mathcal{Q}$) as follows. For any $\vec{y}\,'$, such that $\vec{y}\,' \supset \vec{y}$ and $\vec{y}\,' \subseteq \bigcup_{i=1}^{k} \vec{x}_i$, the formula

$$\mathcal{Q}'\colon\ \exists \vec{y}\,'.\ q_1\langle \vec{x}_1 \rangle \wedge \ldots \wedge q_k\langle \vec{x}_k \rangle.$$

is called an *abbreviated* formula of $\mathcal{Q}$. Note that free variables of $\mathcal{Q}'$ (the distinguished variables) are a subset of the set of free variables of $\mathcal{Q}$. Thus, the answers of $\mathcal{Q}'$ are "sub-answers" of the answers of $\mathcal{Q}$, in that some "attributes" have been projected out.

In keeping with the logic model, we define an *answer* of $\mathcal{Q}$ with respect to database **DB** to be a ground substitution θ over the free variables of $\mathcal{Q}$, such that

$$\mathbf{DB} \models\ \mathcal{Q}\theta$$

The answer set of $\mathcal{Q}$ with respect to database **DB** is the set of all $\mathcal{Q}$'s answers. We shall denote the answer set as $[\![\mathcal{Q}]\!]_{\mathbf{DB}}$, or simply $[\![\mathcal{Q}]\!]$ when **DB** is understood. A relational *table* will be synonymous for us with an answer set.

A *semantic query cache* (or just *semantic cache* for short) is a pair of a query formula with its answer set, $\langle \mathcal{Q}, [\![\mathcal{Q}]\!] \rangle$. We presume that the query's answer set $[\![\mathcal{Q}]\!]$ has been stored locally as a relational table, and that the table has been labeled by the query formula $\mathcal{Q}$. We simply refer to the query formula that has been cached as a *cache formula* and the cached answer set as the *cache table*. Often, we shall use the term *cache* to refer to just the cache formula, when clear by context.

3.2 Determining when Answers are in Cache

We describe formally the conditions that need to be satisfied for the query $\mathcal{Q}$ to be answerable from the set of caches $\mathcal{C}_1, \ldots, \mathcal{C}_m$. We consider cases 1, 2, and 3 as depicted in Figure 1, which represent the relations between a query and caches. Let $\mathcal{Q}$ be a query with distinguished variables $\vec{x}$ and $\mathcal{E}$ be any select-project-join expression—to be called a *cache expression*—over any subset of the caches $\mathcal{C}_1, \ldots, \mathcal{C}_m$. Thus, a cache expression $\mathcal{E}$ can be expressed as an existentially quantified, conjunctive formula, just as query and cache formulas themselves:

$$\mathcal{E}\colon \exists \vec{y}.\ c_{i_1}\langle \vec{x}_{i_1}\rangle \wedge \ldots \wedge c_{i_k}\langle \vec{x}_{i_k}\rangle.$$

where $i_j \in \{1, ..., m\}$, $1 \leq j \leq k$, the variables and constants of the $\vec{x}_i$'s represent the appropriate selects and joins, and the variables of $\vec{x}_i$'s are appropriately named.

- *Containment: All answers to a query $\mathcal{Q}$ are in cache.* There is a finite collection of cache expressions $\mathcal{E}_1, \ldots, \mathcal{E}_n$ such that

$$\mathbf{I_{DB}} \models \forall.\ \mathcal{Q} \rightarrow (\mathcal{E}_1 \vee ... \vee \mathcal{E}_n) \tag{5}$$

- *Abbreviated containment: All answers of an abbreviation of $\mathcal{Q}$ are in cache.* There is $\mathcal{Q}'$, an abbreviation of $\mathcal{Q}$ and there is a finite collection of cache expressions $\mathcal{E}_1, \ldots, \mathcal{E}_n$ such that

$$\mathbf{I_{DB}} \models \forall.\ \mathcal{Q}' \rightarrow (\mathcal{E}_1 \vee ... \vee \mathcal{E}_n) \tag{6}$$

Note that $\mathbf{I_{DB}}$ is on the left hand side of the entailment operator in the above definitions. This means that inference over the rules in the $\mathbf{I_{DB}}$ is allowed. Thus, the right hand side is *not* at tautology, but holds only with respect to the $\mathbf{I_{DB}}$.

The following examples illustrate the two cases from above.

Example 1. Consider a database **DB**, with two tables: **Employee**[Name, SSN, Age] and **Benefits**[SSN, Provider].

1. Consider the following query $\mathcal{Q}$ which asks for names of employees with benefits:

$$\mathcal{Q}\colon\ q(N) \leftarrow \mathit{employee}(N,S,A),\ \mathit{benefits}(S,P).$$

and the caches $\mathcal{C}_1$ and $\mathcal{C}_2$ which store names of employees younger than 50 and older than 20 respectively:

$$\mathcal{C}_2\colon\ c_1(N) \leftarrow \mathit{employee}(N,S,A),\ A < 50.$$

$$\mathcal{C}_2\colon\ c_2(N) \leftarrow \mathit{employee}(N,S,A),\ A > 20.$$

Clearly, all answers to $\mathcal{Q}$ are contained in the union of answer sets for $\mathcal{C}_1$ and $\mathcal{C}_2$. Note, however, that without knowing the values of S in *benefits*(S,P) it would be impossible to *distinguish* the tuples that represent answers to $\mathcal{Q}$ from among all of the tuples in the union of $\mathcal{C}_1$ and $\mathcal{C}_2$.

2. Let the caches $\mathcal{C}_1$ and $\mathcal{C}_2$ be as defined above and the query $\mathcal{Q}$ now ask for names and SSN's of employees with benefits:

$$\mathcal{Q}\colon\ q(N,S) \leftarrow \mathit{employee}(N,S,A),\ \mathit{benefits}(S,P).$$

This query cannot be answered from any combination of $\mathcal{C}_1$ and $\mathcal{C}_2$. However, all sub-tuples projected for N (that is, the tuples with just the names of employees) are contained in the union of caches $\mathcal{C}_1$ and $\mathcal{C}_2$.

3.3 Finding the Answers in Cache

As illustrated in Example 1, the two tests describing the query-cache containment are not sufficient to guarantee that any answers to a query can actually be *retrieved* from cache. Thus, we state two other conditions that provide such a guarantee.

- *Answerability: Some answers to $\mathcal{Q}$ can be retrieved from cache.* There exists a cache expression $\mathcal{E}$ such that:

$$\mathbf{I_{DB}} \models \forall .\ \mathcal{E} \rightarrow \mathcal{Q} \tag{7}$$

- *Some answers of an abbreviation of $\mathcal{Q}$ can be retrieved from cache.* There is $\mathcal{Q}'$, an abbreviation of $\mathcal{Q}$, and a cache expression $\mathcal{E}$ for which:

$$\mathbf{I_{DB}} \models \forall .\ \mathcal{E} \rightarrow \mathcal{Q}' \tag{8}$$

The case when the query can be completely answered from cache is now easy to state. It is a simple combination of conditions (5) and (7): all answers to $\mathcal{Q}$ can be retrieved from cache if and only if there is a finite collection of cache expressions $\mathcal{E}_1, \ldots, \mathcal{E}_n$, such that

$$\mathbf{I_{DB}} \models \forall \vec{x}\ .\ \mathcal{Q} \rightarrow (\mathcal{E}_1 \vee ... \vee \mathcal{E}_n) \text{ and } \mathbf{I_{DB}} \models \forall \vec{x}\ .\ \mathcal{E}_i \rightarrow \mathcal{Q}, \text{ for each } i \in \{1, \ldots, n\}. \tag{9}$$

Condition (9) essentially establishes an equivalence between $\mathcal{Q}$ and the union of $\mathcal{E}_1, \ldots, \mathcal{E}_n$. We note that the only known general procedure for establishing equivalence between two queries is testing for containment in both directions [33]. The ultimate goal of SQC for many applications is to answer a query entirely from caches ((9)). Often, however, only one "half" of (9)—that is, either (5) or (7)—will be satisfied.

3.4 Semantic Overlap and Semantic Independence

We now consider cases 4, 5, and 6 from Figure 1, in which the query and the cache expression overlap somehow, but without being contained in one direction or another. There are two different ways in which they may overlap, but not be contained. First, it may be that the query $\mathcal{Q}$ itself is not contained by the cache expression $\mathcal{E}$, but an unfolding $\mathcal{U}$ of $\mathcal{Q}$ is. If $\mathcal{U}$ is answerable by $\mathcal{E}$, then $\mathcal{Q}$ is obviously partially answerable by $\mathcal{E}$. As we saw in the previous section in formula (4), it can be determined whether a collection of cache expressions in composite completely answer the query. Of course, it is possible that one can only partially answer $\mathcal{Q}$ with the cache expressions. This is possible when certain unfoldings of $\mathcal{Q}$ are not answerable by cache, even while the rest of $\mathcal{Q}$'s unfoldings may be completely answerable by cache.

It is also possible for a query $\mathcal{Q}$ and a cache expression $\mathcal{E}$, however, to *semantically overlap*, and yet *no* unfolding of $\mathcal{Q}$ is completely contained by $\mathcal{E}$. The sharing between $\mathcal{Q}$ and $\mathcal{E}$ may be finer grained. Consider the following example.

Example 2. Consider that the views *employee* and *taxed* are defined as:

$$\begin{array}{l} \mathit{employee}(X) \leftarrow \mathit{payroll}(X),\ \mathit{position}(X). \\ \mathit{taxed}(X) \leftarrow \mathit{payroll}(X),\ \mathit{national}(X). \end{array}$$

> Thus, an employee is someone on the payroll with an official position. The company sets aside taxes for anyone on the payroll who is a national. There may be people on the payroll who are not employees. For instance, retirees may be handled this way. Likewise, there may be people on the payroll who are not nationals. The company does not handle their taxes.
>
> Let *taxed*(X) be cached and *employee*(X) be the current query. Clearly, the query is not contained in the cache, nor vice versa. However, it is also clear they are semantically related, since they mutually rely on the same table *payroll*. Thus, some answers to the query are potentially in the cache (case 3 in Figure 1).

In essence, queries (and caches) overlap whenever they somehow mutually rely on some of the same sources. Let us show how we can logically capture when two query and cache formulas semantically overlap.

Queries $\mathcal{Q}$ and $\mathcal{E}$ *extensionally overlap iff* there exists a query formula $\mathcal{F}$ such that[5]

$$\models \forall.\, (\mathcal{Q} \rightarrow \mathcal{F}) \wedge (\mathcal{E} \rightarrow \mathcal{F}) \tag{10}$$

(call $\mathcal{F}$ an *overlap witness*) and, there exists a query formula $\mathcal{G}$ such that[6]

$$\models \forall.\, (\mathcal{G} \rightarrow \mathcal{Q}) \wedge (\mathcal{G} \rightarrow \mathcal{E}) \tag{11}$$

(call $\mathcal{G}$ an *overlap formula*). Neither (10) nor (11) alone are sufficient to guarantee that $\mathcal{Q}$ and $\mathcal{E}$ overlap. Condition (10) states that $\mathcal{Q}$ and $\mathcal{E}$ clearly share a common resource, $\mathcal{F}$. Thus, this indicates that they must share some sources, ultimately tables, in evaluation. However, $\mathcal{Q}$ and $\mathcal{E}$ may be queries on the same table, yet have incompatible select conditions, and, hence, cannot overlap. Condition (11) guarantees that there is an overlap formula $\mathcal{G}$, but does not guarantee that $\mathcal{Q}$ and $\mathcal{E}$ share resources. Indeed, in the degenerate case, $\mathcal{G}$ can be constructed as $\mathcal{Q} \wedge \mathcal{E}$. The conditions taken together, however, ensure that there is a meaningful overlap. In Example 2, $\mathit{payroll}(X) \wedge \mathit{position}(X)$ and $\mathit{payroll}(X) \wedge \mathit{national}(X)$ extensionally overlap. The overlap witness is $\mathit{payroll}(X)$, and the overlap formula is $\mathit{payroll}(X) \wedge \mathit{position}(X) \wedge \mathit{national}(X)$.

We can define a *most general overlap formula* as an overlap formula $\mathcal{G}$ such that there does not exist another overlap formula $\mathcal{G}'$ such that

$$\models \forall.\, (\mathcal{G} \rightarrow \mathcal{G}') \tag{12}$$

but

$$\not\models \forall.\, (\mathcal{G}' \rightarrow \mathcal{G}) \tag{13}$$

The answers of an overlap formula are answers both of the query and cache that overlap. Thus, if one can evaluate the overlap formula, one can partially answer the query. A most general overlap formula determines a maximal set of mutual answers.

For the intensional case, the definition of an overlap needs to be a little more complex. To test whether $\mathcal{Q}$ and $\mathcal{E}$ overlap, we need to examine whether any of their unfoldings overlap. Let $\mathcal{U}_{\mathcal{Q}}$ and $\mathcal{U}_{\mathcal{E}}$ be arbitrary unfoldings of $\mathcal{Q}$ and $\mathcal{E}$, respectively. Queries $\mathcal{Q}$ and $\mathcal{E}$ *intensionally overlap* with respect to $\mathbf{I_{DB}}$ *iff* any of their respective unfoldings $\mathcal{F}_{\mathcal{Q}}$ and $\mathcal{F}_{\mathcal{E}}$ overlap. This may be stated as follows. For any $\mathcal{U}_{\mathcal{Q}}$ and $\mathcal{U}_{\mathcal{E}}$ such that

$$\mathbf{I_{DB}} \models \forall.\, (\mathcal{U}_{\mathcal{Q}} \rightarrow \mathcal{Q}) \wedge (\mathcal{U}_{\mathcal{E}} \rightarrow \mathcal{E}) \tag{14}$$

and $\mathcal{U}_{\mathcal{Q}}$ and $\mathcal{U}_{\mathcal{E}}$ extensionally overlap, then $\mathcal{Q}$ and $\mathcal{E}$ intensionally overlap.

We call $\mathcal{G}$ a *horizontally-complete* overlap *iff* all free variables of $\mathcal{G}$ are also free variables of $\mathcal{Q}$ (this is case 2 of Figure 1). We call it an abbreviated overlap if there is an abbreviation $\mathcal{Q}'$ of $\mathcal{Q}$ and $\mathcal{Q}'$ overlaps with a cache expression $\mathcal{E}$ (this is case 4 of Figure 1). Abbreviated overlaps are only useful if we are willing to answer in part a query without all the attributes-of-interest [27]. This depends on the needs of the user and is a cooperative answering issue.

Call queries $\mathcal{Q}$ and $\mathcal{E}$ *semantically independent* (with respect to $\mathbf{I_{DB}}$) *iff* $\mathcal{Q}$ and $\mathcal{E}$ do not intensionally overlap (with respect to $\mathbf{I_{DB}}$) in any way. This is case 6 in Figure 1. Note that is necessary to have semantic overlap well defined before we can introduce the notion of semantic independence.

Determining overlap is a generalization of containment. If no cache expression can be found that is contained by the query, then we cannot partially answer the query locally. However, overlap expressions tell us what *almost* can be evaluated locally. Some of the tables and views in the overlap expression are apparently not available locally (else, we would have discovered a containment). If these would be inexpensive to import,

[5] Note that our definition of a query formula (4) does not allow disjunction nor negation, thus $\mathcal{F}$ cannot be a tautology and it cannot be a contradiction.

[6] Likewise for $\mathcal{G}$.

it might be worthwhile. In some cases, migrating, say, a small table might be sufficient to answer the query by cache, whereas evaluating the query at the server would be expensive. Thus, overlaps offer more choices to an evaluation strategy that employs semantic caches.

3.5 Semantic Remainder

If we can partially answer the query by cache, we still have the responsibility to evaluate the *rest* of the query's answers. Of course, the simplest action would be to evaluate the entire query anyway. For pipelining answers to the user, this is sufficient. The user gets the answers from cache quickly, while the query is being evaluated. However, this strategy does little to optimize the overall evaluation effort. It is also unacceptable when caches are kept for security reasons. On the other hand, computing the query that would return the remaining answers but *none* of the answers retrieved from caches may be too expensive as well. Consider the following example.

Example 3. Let the query $\mathcal{Q}$ be:

$$\mathcal{Q}\text{: } q(N) \leftarrow \mathit{employee}(N,S,A).$$

and the cache $\mathcal{C}$ be:

$$\mathcal{C}\text{: } c(N) \leftarrow \mathit{employee}(N,S,A),\ \mathit{benefits}(S,P).$$

Clearly, $\mathcal{C}$ partially answers $\mathcal{Q}$. To save in network bandwidth, one could compute $\mathcal{Q} \wedge \textbf{not}\ \mathcal{C}$ at the server and only ship those results back.[7] This would require computing the join **Employee** $\bowtie$ **Benefits** at the server again.

Let us introduce the notation $\mathcal{Q}\backslash\mathcal{E}$ to represent the remainder query that results from $\mathcal{Q}$ when $\mathcal{E}$ has been removed.[8] We introduced this notation in [15], and call it a *discounted query* (the query $\mathcal{Q}$ discounted with respect to query $\mathcal{E}$). This concept is generally called a *remainder query* [10]. The discounted query should should satisfy the following conditions with respect to the query $\mathcal{Q}$ and the cache expression $\mathcal{E}$ that partially answers $\mathcal{Q}$.

- *Soundness.* All answers to $\mathcal{Q}\backslash\mathcal{E}$ should be correct; that is, for any $\mathcal{Q}$ and $\mathcal{E}$,

  $$[\![\mathcal{Q}\backslash\mathcal{E}]\!] \subseteq [\![\mathcal{Q}]\!]$$

  This condition should hold uniformly for all applications of semantic caching.

- *Completeness.* All answers to $\mathcal{Q}\backslash\mathcal{E}$, together with the answers retrieved from the cache, should provide the complete answer set of $\mathcal{Q}$; that is, for any $\mathcal{Q}$ and $\mathcal{E}$,

  $$[\![\mathcal{Q} - \mathcal{E}]\!] \subseteq [\![\mathcal{Q}\backslash\mathcal{E}]\!]$$

  As with soundness, this condition should hold for all applications.

- *Minimality.* $\mathcal{Q}\backslash\mathcal{E}$ and $\mathcal{E}$ should be semantically independent. If $\mathcal{Q}\backslash\mathcal{E}$ and $\mathcal{E}$ are *not* semantically independent, then some of the answers already retrieved from cache may be recomputed at the server. For some applications, such as caching secure data, semantic independence should be enforced at all costs. For other applications, (in particular, query optimization) cost effectiveness (discussed below) is most important, and we may be willing to recompute some answers to a query if this leads to more efficient use of resources. One way of enforcing semantic independence is simply to define $\mathcal{Q}\backslash\mathcal{E}$ as

[7] We have not considered negation in this paper. We introduce it here for discussion in this section. The query $\mathcal{Q} \wedge \textbf{not}\ \mathcal{C}$ should evaluate to all answers of $\mathcal{Q}$ *minus* those of $\mathcal{C}$, or $[\![\mathcal{Q}]\!] - [\![\mathcal{C}]\!]$, in which '$-$' is the standard relational *set minus* operator.

[8] We usurp the use of '$\backslash$' for discounting, so it does not mean the same as '$-$' here.

$\mathcal{Q} \wedge \textbf{not}\ \mathcal{E}$. This, however, may make $\mathcal{Q} \backslash \mathcal{E}$ more expensive to evaluate (as shown in Example 3) than $\mathcal{Q}$. Also, this does not capture what $\mathcal{Q} \backslash \mathcal{E}$ is intended to mean: query $\mathcal{Q}$ with all the overlaps with cache $\mathcal{E}$ "removed". (We discuss below a definition of $\mathcal{Q} \backslash \mathcal{E}$ that for some types of queries ensures semantic independence and cost effectiveness at the same time.)

- *Uniformity,* The following condition should hold:

$$[\![\mathcal{Q} \backslash \mathcal{E}]\!] - [\![\mathcal{E} \backslash \mathcal{Q}]\!] = [\![\mathcal{Q}]\!] - [\![\mathcal{E}]\!] \tag{15}$$

  If $\mathcal{A} \backslash \mathcal{B}$ is defined degenerately as $\mathcal{A}$ (thus, the discounting does nothing), then this trivially holds. If $\mathcal{A} \backslash \mathcal{B}$ is defined degenerately as $\mathcal{A} - \mathcal{B}$ then this again trivially holds. If $\mathcal{A} \backslash \mathcal{B}$ randomly evaluates to something between $[\![\mathcal{A}]\!]$ and $[\![\mathcal{A}]\!] - [\![\mathcal{B}]\!]$, this does not always hold. It would be possible for $\mathcal{Q} \backslash \mathcal{E}$ to evaluate to $[\![\mathcal{Q}]\!]$, but for $\mathcal{E} \backslash \mathcal{Q}$ to evaluate to $[\![\mathcal{E}]\!] - [\![\mathcal{Q}]\!]$, thus resulting in $[\![\mathcal{Q}]\!]$. However, if $\mathcal{A} \backslash \mathcal{B}$ is defined meaningfully, it should be possible to ensure uniformity.

- *Cost effectiveness*: evaluating $\mathcal{Q} \backslash \mathcal{E}$ and $\mathcal{E}$ should cost less than evaluating $\mathcal{Q}$. This condition can be stated differently depending on the application and computing environment. If the cost is measured by the processing time until all answers are retrieved and $\mathcal{Q} \backslash \mathcal{E}$ and $\mathcal{E}$ can be computed in parallel, then it is sufficient that $\mathcal{Q}$ costs more to evaluate that the more expensive of $\mathcal{Q} \backslash \mathcal{E}$ and $\mathcal{E}$. If the cost is measured by the amount of money paid for answers, then defining $\mathcal{Q} \backslash \mathcal{E}$ as $\mathcal{Q} \wedge \textbf{not}\ \mathcal{E}$ is always more cost effective than $\mathcal{Q}$ as long as $\mathcal{E}$ is nonempty.

We have been quite interested to define a semantics for $\mathcal{Q} \backslash \mathcal{E}$, and ways to evaluate $\mathcal{Q} \backslash \mathcal{E}$—that is, to produce $[\![\mathcal{Q} \backslash \mathcal{E}]\!]$—efficiently. In [15], we introduce a type of optimization we call *intensional query optimization* (IQO). The idea behind IQO is to "remove" certain unfoldings from a view query that, say, may be known to evaluate empty or which can be evaluated inexpensively locally. For IQO, we introduced and defined a weaker version of discounting: given query $\mathcal{Q}$ and a collection of some of its unfoldings $\mathcal{U}_1, \ldots, \mathcal{U}_k$, then $\mathcal{Q} \backslash \{\mathcal{U}_1, \ldots, \mathcal{U}_k\}$ denotes $\mathcal{Q}$ with those unfoldings "removed".[9]

We have explored various approaches to evaluate $\mathcal{Q} \backslash \{\mathcal{U}_1, \ldots, \mathcal{U}_k\}$. One method is to rewrite the query $\mathcal{Q}$ algebraically in such a way that the resulting query evaluates to $[\![\mathcal{Q} \backslash \{\mathcal{U}_1, \ldots, \mathcal{U}_k\}]\!]$. We explore the complexity issues of such rewrite techniques in [18]. Another approach is to develop a specialized evaluation strategy that can evaluate discounted queries ($\mathcal{Q} \backslash \{\mathcal{U}_1, \ldots, \mathcal{U}_k\}$) directly. In [17], we introduce such a method which we call *tuple tagging.* The method furthermore is an optimization as, in general, the discounted query is less expensive to evaluate than the query itself. (In [17], we show experimental evidence for this.)

We can already define a limited notion of $\mathcal{Q} \backslash \mathcal{E}$ via $\mathcal{Q} \backslash \{\mathcal{U}_1, \ldots, \mathcal{U}_k\}$: find the collection of unfoldings of $\mathcal{Q}$ for which each is answerable by $\mathcal{E}$. However, we want to capture a stronger notion, and "remove" all overlaps with $\mathcal{E}$ instead. The semantics for $\mathcal{Q} \backslash \mathcal{E}$ is important to different applications. (It may be one version of $\mathcal{Q} \backslash \mathcal{E}$ is not sufficient.) For instance, the type and semantics of data security that one wishes to support could be provided by $\mathcal{Q} \backslash \mathcal{E}$, if it is defined correctly.

If we had an evaluation strategy that generally evaluates $\mathcal{Q} \backslash \mathcal{E}$ more inexpensively than $\mathcal{Q}$ itself—and we do already have such a strategy for $\mathcal{Q} \backslash \{\mathcal{U}_1, \ldots, \mathcal{U}_k\}$—then condition (15) above means that we would have a method to optimize set minus. Since set minus is an increasingly important relational operator which is used in analysis queries, for instance, in data warehousing environments, such an optimization technique might be quite worthwhile.

[9] $\mathcal{Q} \backslash \{\mathcal{U}_1, \ldots, \mathcal{U}_k\}$ might be called *syntactic discounting*, while $\mathcal{Q} \backslash \mathcal{E}$ might be called *semantic discounting*.

4 Related Work and Work to be Done

4.1 Previous and Current Work

Problems similar to the ones we discuss in this paper have been addressed in two different contexts: theoretically, as a query containment problem; and practically, as a query optimization problem. One query can be useful for answering another query when there is a semantic "overlap" between them (as discussed in Section 3.4). A special case of overlap is known as *query containment*; that is, when it can be shown that an answer set of one query is a subset of an answer set of another query. Containment between *extensional, conjunctive queries* (that is, queries that are logical conjunctions and do not involve intensional atoms) was first studied in [6], and the problem was shown to be NP-complete.[10] Several sub-classes of extensional, conjunctive queries have been identified to have polynomial-time algorithms [2, 3, 19]. Containment tests for extensional, conjunctive queries that permit negation have been presented in [23], and for those that involve arithmetic comparisons in [21]. Containment between intensional queries with respect to a Datalog program (or, equivalently, containment between Datalog programs) is computationally harder: the containment question between Datalog programs is generally undecidable [31]; and the question of whether a Datalog program is contained by a extensional, conjunctive query is doubly exponential [8].

An extension of the query containment problem is the problem of rewriting a given query by means of other queries. This is known as *query folding*. This problem has been considered in the context of heterogeneous database systems [25, 28] and query rewriting using materialized views [7, 24]. In each of these cases, however, only extensional, conjunctive queries have been considered.

Practical issues of discovering and exploiting query overlaps have been considered in the context of *multiple query optimization* [30]. Its goal is to optimize evaluation in batch of a set of queries, rather than the optimization of a single query. The developed techniques are geared towards finding and reusing common sub-expressions in the set of queries and are heuristics-based.

The idea of the caching of query results to optimize the processing of subsequent queries was first studied in [12] and [22]. In both cases, the developed techniques are restricted to a subset of extensional, conjunctive queries. (In particular, no self-joins are permitted.) The techniques do not, however, find queries that are contained by the original query; that is, queries which evaluate to a subset of the original query's answer set. In [9], the implementation of ADMS is described, which includes a query caching system based on the algorithms of [12]. Both [10] and [20] extend the paradigm of query caching to use caches to provide partial answers to the query. They both assume, however, that a semantic cache is only useful when some of the query's answers can be obtained from a single cache via project and select operations. Although this framework allows for an efficient implementation of semantic caching, it does not guarantee that all of the query's answers available from caches will, indeed, be found. Moreover, these semantic caching strategies have been designed explicitly for the purpose of query optimization, but other applications have not been considered. Query caching in heterogeneous environments has been investigated in [1]. This approach also does not consider joins over cached queries.

4.2 Future Work and an SQC Agenda

In [16], we proposed to extend the SQC paradigm by allowing for all relational operations to be performed over caches. Thus, caches can be considered in combination via joins to answer queries. In previous work, the focus has been on when a given cache table can be employed, perhaps with certain project and select operations, to answer partially the query. Although this restriction (considering caches singlely) allows for efficient implementation of SQC, it greatly restricts the opportunities to answer the query by the caches. Towards the goal of greater expressiveness with efficiency for better cache utility, we formalize a general

[10] In [6] and elsewhere, extensional, conjunctive queries are simply called conjunctive queries.

framework for semantic caching in first-order logic, and we consider tests needed to reason about queries and caches within the more general framework. If the query is answerable *entirely* from cache, no other queries need be evaluated. Otherwise, it may be useful to determine a *remainder query*, which finds the remaining answers of the query (those not found by cache) when evaluated. As discussed in Section 3.5, remainder queries may have many applications. Remainder queries have not received much attention yet, and previous work only considers them in limited contexts [10, 13, 15, 27].

It is important to note that for many applications, it may not be necessary to compute containments, overlaps, and discounting precisely. It is enough if we have methods that are guaranteed to be sound. They need not be complete (always return an answer when there is an answer) or precise (always return best answers). Many applications can use those tools opportunistically for gain. For instance, semantic query optimization can be applied if such tools are available. We do not need a complete and precise analysis of queries to have the benefit of semantic query optimization. Of course, other applications may need further or tighter guarantees. For instance, query security must assure that secured portions of a query are, indeed, "removed".

For successful SQC, many issues need to be resolved. All the standard issues that arise for any caching project must be addressed. First, efficient algorithms should be developed for particular applications of SQC. These must include not only generating a cache expressions that (partially) answer the query, but also choosing the best ones (if their answer sets overlap) as well as computing the remainder of the query to be evaluated at the server. Second, heuristics should be developed to decide when, for a given application, cache use would bring the desired benefits. Third, issues of cache maintenance should be resolved; these include cache replacement strategy, maintaining cache currency and merging semantically similar caches. There has been some consideration of SQC maintenance issues recently. In [4], cache maintenance is considered within the domain of WWW information sources. In [29], semantic caching is defined, and maintenance issues raised and addressed.

5 Conclusions

In this paper, we presented a general logical framework for SQC. We specified conditions to determine when answers, or partial answers, to a query are *present* in cache, and whether they can be *retrieved* from cache. Our framework extends the previous work in this area in several ways.

1. Our criteria to check whether caches are useful in answering a query are complete in the sense that all answers that *can* be retrieved through any relational combination of cache expressions *can*, in fact, be discovered.
2. Our criteria work for intensional queries and caches (queries and caches over views). This makes our approach particularly pertinent for data warehousing and mediated environments.
3. We extend the notion of a partial answer to a query to account for the case when only a subset of requested attributes is returned to the user. Such answers have been shown useful in heterogeneous environments [27] when not all data sources are always available.
4. We introduce a new concept of *semantic overlap* between queries and caches. Previously, only containment between queries and caches has been considered. Semantic overlap allows for more possibilities to exploit caches for answering queries.
5. We introduce a much richer formalism for remainder queries, called *discounted queries*, and outline the issues involved in defining a formal semantics for them.

References

[1] S. Adali, S. Candan, Y. Papakonstantinou, and V. S. Subrahmanian. Query caching and optimization in distributed mediator systems. In *Proc. SIGMOD*, pages 137–148, Montreal, Canada, June 1996.

[2] A. Aho, Y. Sagiv, and J. Ullman. Efficient optimization of a class of relational expressions. *TODS*, 4(3):434–454, 1979.

[3] A. Aho, Y. Sagiv, and J. Ullman. Equivalence of relational expressions. *SIAM Journal of COmputing*, 8(2):218–246, 1979.

[4] N. Ashish, C. Knoblock, and C. Shahabi. Optimizing information agents by selectively materializing data. In *Proceedings of the Workshop on Artificial Intelligence and Information Integration*, pages 17–22, Madison, Wisconsin, July 1998. Held in conjunction with AAAI'98.

[5] M. Carey, M. Franklin, and M. Zaharioudakis. Fine-grained sharing in page server database system. In *Proceedings of Sigmod*, 1994.

[6] A. Chandra and P. Merlin. Optimal implementation of conjunctive queries in relational databases. In *Proc. Ninth ACM Symposium on the Theory of Computing*, pages 77–90, 1977.

[7] S. Chaudhuri, R. Krishnamurthy, S. Potamianos, and K. Shim. Optimizing queries with materialized views. In *Proceedings of the 11th ICDE*, pages 190–200, 1995.

[8] S. Chaudhuri and M. Vardi. On the equivalence of datalog programs. In *Proceedings of PODS*, pages 55–66, 1992.

[9] C. M. Chen and N. Roussopoulos. The implementation and performance evaluation of the ADMS query optimizer: Integrating query result caching and matching. In *Proc. of the* 4^{th} *EDBT Conference*, Cambridge, UK, 1994.

[10] S. Dar, M. Franklin, B. Jónsson, D. Srivastava, and M. Tan. Semantic data caching and replacement. In *Proceedings of VLDB*, 1996.

[11] D. DeWitt, P. Futtersack, D. Maier, and F. Velez. A study of three alternative workstation-server architectures for object-oriented database systems. In *Proceedings of VLDB*, 1990.

[12] S. Finkelstein. Common expression analysis in database application. In *Proceedings of SIGMOD*, pages 235–245, 1982.

[13] P. Godfrey and J. Gryz. A framework for intensional query optimization. In D. Boulanger, U. Geske, F. Giannotti, and D. Seipel, editors, *Proceedings of the Workshop on Deductive Databases and Logic Programming*, GMD-Studien Nr. 295, pages 57–68, Bonn, Germany, Sept. 1996. GMD-Forschungszentrum. Held in conjunction with IJCSLP'96.

[14] P. Godfrey and J. Gryz. Intensional query optimization. Technical Report CS-TR-3702, UMIACS-TR-96-72, Dept. of Computer Science, University of Maryland, College Park, MD 20742, Oct. 1996.

[15] P. Godfrey and J. Gryz. Overview of dynamic query evaluation in intensional query optimization. In *Proceedings of Fifth DOOD*, pages 425–426, Montreux, Switzerland, Dec. 1997. Longer version appears as [14].

[16] P. Godfrey and J. Gryz. Semantic query caching in heterogeneous databases. In *Proceedings KRDB at VLDB'97*, Athens, Greece, Aug. 1997.

[17] P. Godfrey and J. Gryz. A Strategy for Partial Evaluation of Views. Submitted, 1998.

[18] P. Godfrey and J. Gryz. View disassembly. Submitted, 1998.

[19] D. S. Johnson and A. Klug. Optimizing conjunctive queries that contain untyped variables. *SIAM Journal of Computing*, 12(4):616–640, 1983.

[20] A. M. Keller and J. Basu. A predicate-based caching scheme for client-server database architectures. *The VLDB Journal*, 5(2):35–47, Apr. 1996.

[21] A. Klug. On conjunctive queries containing inequalities. *Journal of the ACM*, 35(1):146–160, 1988.

[22] P.-A. Larson and H. Yang. Computing queries from derived relations. In *Proc. of 11th VLDB*, pages 259–269, 1985.

[23] A. Levy and Y. Sagiv. Queries independent of updates. In *Proc. of VLDB*, pages 171–181, 1993.

[24] A. Y. Levy, A. O. Mendelzon, Y. Sagiv, and D. Srivastava. Answering queries using views. In *Proc. PODS*, pages 95–104, 1995.

[25] A. Y. Levy, A. Rajaraman, and J. Ordille. Querying heterogeneous information sources using source descriptions. In *Proc. 22nd VLDB*, 1996.

[26] J. W. Lloyd. *Foundations of Logic Programming.* Symbolic Computation—Artificial Intelligence. Springer-Verlag, Berlin, second edition, 1987.

[27] H. Naacke, G. Gardarin, and A. Tomasic. Leveraging mediator cost models with heterogeneous data sources. In *Proceedings of the Fourteenth International Conference on Data Engineereing (ICDE'98)*, pages 351–360, Orlando, Florida, Feb. 1998.

[28] X. Qian. Query folding. In *Proceedings of the* 12^{th} *International Conference on Data Engineering*, pages 48–55, 1996.

[29] Q. Ren and M. H. Dunham. Semantic caching and query processing. Technical Report 98-CSE-04, Department of Computer Science and Engineering, Soutern Methodist University, Dallas, Texas, May 1998.

[30] T. Sellis and S. Ghosh. On the multiple-query optimization problem. *TKDE*, 2(2):262–266, June 1990.

[31] O. Shmueli. Decidability and expressiveness aspects of logic queries. In *Proc.* 6^{th} *ACM Symposium on Principles of Database Systems*, pages 237–249, 1987.

[32] J. D. Ullman. *Principles of Database and Knowledge-Base Systems, Volumes I & II.* Principles of Computer Science Series. Computer Science Press, Incorporated, Rockville, Maryland, 1988/1989.

[33] J. D. Ullman. Information integration using logical views. In *Proceedings of the Sixth International Conference on Database Theory (ICDT'97)*, Delphi, Greece, Jan. 1997.