Data Sufficiency for Queries on Cache

Oded Shmueli
HP Israel Science Center*

and

Kurt Shoens
HPL Software Technology Laboratory†

Keywords: Databases, Client-Server, Cached Data, Containment, Data Sufficiency

Abstract

In distributed computing environments, replication of data provides improved availability, isolation between workloads with different characteristics, and improved performance through local access to data. The “real data” is server resident and by “local data” we refer to cached client data. We examine which data should be cached on behalf of a cached query. The minimum requirement for cached data for a query $Q$ is that it enables answering $Q$ locally.

We consider the following:

- Definitions of what data is cached for a cached query.
- Deciding whether cached data can be used to solve a “new” query.
- Deciding whether a “new” query to be cached is already effectively cached due to caching of other queries.
- A simple class of caching rules.
1 Introduction

In distributed database environments, replication provides improved availability, isolation between on-line and decision support workloads, and local access to data. With the rising popularity of the Client/Server architecture, there has been much interest in client/server caching [KB96]. The main motivation is the increased processing power of client machines. This allows the clients to locally perform queries efficiently. Still, clients are limited in the amount of information they can store locally. So, data is mostly maintained on the server’s side. This basic scenario raises many problems of implementation.

There have been a number of suggestions as to how this caching should be done. Keller and Basu [KB96] suggest a predicate based scheme for caching data at the client site. Wilkinson and Neimat have considered the problem in the context of object caching [WN90]. Carey et al. [CFLS91, CFZ94, FCL93] consider page level caching using centralized indexing.

The distributed system model we consider resembles the Sybase Replication Server [Moi96]. We assume that a primary data server holds an “official up-to-date copy” of the data for which it is the primary site. The system employs a replica distribution engine that supplies clients with primary data copies called replicas. So, when a primary data copy is updated, this update, and perhaps other related data, is sent to clients that hold local copies (or query defined derived copies) of the primary data that was updated. The exact nature of this replication mechanism, called CacheServer, is discussed in a separate paper [SNS95].

The client is responsible for caching queries. By “caching queries” we mean caching copies of primary data tuples that are judged “relevant” in answering the query. In fact, some of these “queries” need not even be user queries but may be DBA-defined “slices” of primary data judged useful for the client’s working patterns.

Caching resembles view maintenance (materialization), however query answers are not cached, rather raw material for answering queries is cached. So, no explicit views are maintained at the client (although they might as another, orthogonal, optimization). Since only “ordinary” tuples exist at the client, the client may use the same programs it would have used on the server. The client may use a standard query processor which uses standard query optimization techniques. Thus, both applications and systems remain standard, the complexity is relegated to the CacheServer. (At an abstract level, one may think of the CacheServer as maintaining a complex view, namely the cache itself.)

Saving raw material increases the likelihood that cached data be relevant for many queries. Furthermore, there is a tradeoff between storing “precisely” the relevant tuples or a superset thereof. The more precise the cached data, the less likely it will be useful in the future. Of course, having a precise cache reduces its size with the obvious performance and storage advantages. These, and similar observations appear in [KB96].

In this paper we examine the following:

- Definitions of what data is cached for a “cached query”. We introduce the idea of “caching modes” which are methods for deciding upon “caching rules”.

- Deciding whether cached data can be used to solve a “new” query posed at the client by utilizing locally cached data only. We show that employing “caching modes” makes an easier decision as to whether a query can at all be answered based solely on cache.
• Deciding whether a “new” query to be cached at the client is already effectively cached due to caching of other queries; in such a case there is no need to take any actions in order to cache the new query.

• Consider a sub-class of caching rules whose bodies have a single EDB atom and inequalities.

• Classify the complexity of the query sufficiency problem (also for the above sub-class).

These have direct bearings on the protocol to be followed by the client and the replication server (item 1), on the way the client handles a locally posed query (item 2), on the actions taken in order to cache a new query (item 3), and on the practicality of various caching schemes (items 4 and 5). The second item is related to the problem of answering a query using views, see [LMSS95] for a recent paper on this topic.

The paper is organized as follows. Section 2 presents terminology. In section 3 we introduce “caching modes”. Section 4 examines the problem of whether the cache has sufficient data for evaluating a “new” query. In section 5 we present the advantages of the “caching modes” idea. Section 6 considers the problem: given a “new” query to cache, is it “effectively cached”. In section 7 we examine a simple sub-class of caching rules. Section 8 examines the complexity of the query sufficiency problem. We conclude in section 9.

2 Terminology

All queries we consider are conjunctive, that is rules of the form $q(X) \gets q_1(X_1), \ldots, q_m(X_m)$ where $q$ and the $q_i$’s are predicate symbols, $X$ and the $X_i$’s are vectors of constants and variables, and $q_i(X_i)$ is called an atom [Ull89]. The head of the query is $q(X)$ and its body is $q_1(X_1), \ldots, q_m(X_m)$. Some atoms may have built-in predicate symbols, and may also be written using infix notation, e.g., $X_1 < 17$. Such atoms are called built-in atoms.

A program is a finite collection of rules. One predicate is singled out as the target predicate of the program. The relation computed for the target predicate is the result of the program’s computation. The conjunctive queries (or rules) that make the program, may refer in their bodies to predicate names that are head predicates in rules of the programs. Predicates are thus partitioned into those that only appear in bodies of queries, called EDB predicates, and those that appear in heads of rules, called IDB predicates. If predicate $p$ appears in the head of a rule in program $P$ and predicate $q$ in the rule’s body, then $p$ depends on $q$. We consider only non-recursive programs in which there is no cycle in the “depends-on” relation.

A program is evaluated by evaluating its rules one at a time. A rule $r$ is evaluated only once all rules whose head predicates appear in $r$’s body have already been evaluated. The rule evaluation consists of assigning values to the rule’s variables, verifying that all body atoms are satisfied with this assignment, and deriving an IDB atom (fact) for the rule’s head atom based on the variable assignment [Ull89].

Conjunctive queries are a mathematical formalism that can be used to express SQL queries of the form:

---

1We assume that rules are safe, i.e. all head variables appear in the body [Ull89].
Select distinct $A$ ...
From $R_1, \ldots, R_m$
Where $C_1 \text{ AND } \ldots \text{ AND } C_n$

The above query can be expressed as the conjunctive query:

$$p(A, \ldots) \leftarrow c_1, \ldots, c_n, d_1, \ldots, d_k$$

Each atom $c_i$ is of the form $R_i(\ldots)$ where the arguments of $c_i$ reflect the variable equalities implied by the where-clause of the SQL query as well as equality to constants. Each $d_i$ is an inequality relating variables and/or constants.

For example,

Select distinct $A, S.D$
From $R, S$
Where $R.A = S.B \text{ AND } S.C > 6 \text{ AND } R.A > 3$

is expressed as the conjunctive query:

$$p(A, D) \leftarrow R(A, X), S(B, C, D), A = B, C > 6, A > 3$$

or equivalently:

$$p(A, D) \leftarrow R(A, X), S(A, C, D), C > 6, A > 3.$$

A query is *sufficiently cached* at a client if the client is always guaranteed to locally store a subset $S$ of the tuples, of the server resident database, such that the query may be answered correctly by applying it, *unchanged*, to that subset $S$. This definition does not specify how the members of $S$ are decided upon and how $S$ is maintained over time (as tuples are inserted, modified, or deleted). Maintenance issues are addressed in [SNS95].

Given a query $Q$ and a set of views (queries), the *rewriting* (resp., *complete rewriting*) problem is whether one can pose $Q$ by using some views (resp., only the views, as opposed to the base relations that are referenced in $Q$) [LMSS95]. The work considers conjunctive queries with built-in predicates. For the (complete) rewriting problem to be applicable, one needs view definitions and their associated tuples. As we shall see in the next section, our description of which data is cached will be in terms of views, each described by a union of conjunctive queries. However, we maintain no association between a cached tuple and which, if any, cache description query it corresponds to.

Our scheme does not preclude using view rewriting techniques. If a query $Q$ is sufficiently cached (independently of view maintenance), and one maintains views based on the cached data, and rewrites (completely or partially) a query $Q$ based on these views, using view rewriting techniques, then one is guaranteed of a correct result for $Q$ from this rewriting.
3 Caching Modes and Queries Expressing Them

Consider a query $Q$ of the form ($BIP$ are the built-in atoms):

$$q(X) \leftarrow q_1(X_1), \ldots, q_m(X_m), BIP$$

The goal is that evaluating $Q$, as is, on the cache would result in the same answer as evaluating $Q$ on the database. One way of ensuring this is that for each EDB atom $q_i(X_i)$ occurrence in the body we introduce a rule of the form $qq_i(X_i) \leftarrow Body'$, where $Body'$ includes the $q_i(X_i)$ occurrence and some subset of the rest of the body atoms. Predicate $qq_i$ defines the tuples cached for predicate $q_i$ due to this occurrence. Query $Q$ is sufficiently cached because any tuple $t$ for $q_i$ that participates in a satisfaction of the body of $Q$, will also participate in satisfying a subset $Body'$ of that body and hence will be cached. Therefore, when $Q$ is evaluated on the cache this tuple will be available.

There is a tradeoff between the cardinalities of bodies’ subsets $Body'$ used and the ability to easily decide whether a tuple of some $q_i$ should be cached, the more precise the decision, the more work it entails. In addition, deciding how to cache each query individually seems to induce a high overhead. Therefore, we introduce the concept of caching mode which is a prescription for constructing $Body'$.

A caching mode choice is applied to each body atom occurrence, or pair of occurrences (for Sat-join). The choice is a function of the work involved in data caching decisions versus the expected benefit. There is a spectrum of possibilities for caching modes and we list the ones that seem most reasonable to employ. The cached data may be described as a collection $W$ of conjunctive queries.

1. **All-body (everything).** All tuples for all relations names that are mentioned in the body of the query are cached.
   For each $q_i(X_i)$ add a new query to $W$, $qq_i(X_i) \leftarrow q_i(X_i)$.

2. **Sat-body (potential satisfaction.)** Consider a tuple $t$ for relation $q_i$ in the body of the query. Form a new query by matching $q_i(t)$ and $q_i(X_i)$. If this new query is satisfiable, then $t$ is cached. Consider the cached query

$$q(X) \leftarrow a(X, Y), b(Y, Z), Z > X, X > 10$$

and the tuple $a(5, 8)$. After the matching the query becomes

$$q(5) \leftarrow a(5, 8), b(5, Z), Z > 5, 5 > 10$$

This new query is not satisfiable and hence the tuple $t$ will not be cached.
   For each $q_i(X_i)$ add a new query to $W$, $qq_i(X_i) \leftarrow q_i(X_i), BIP$.

3. **Sat-tuple (actual satisfaction.)** Consider a tuple $t$ for relation $q_i$. $t$ is cached if $t$ can currently participate in some satisfying assignment to the body of the query. That is, if there are tuples $t_1, \ldots, t_n$ in the database so that
A variation on the above definitions is considering existential variables which are variables that appear only in a single query body atom and do not appear in the query’s head. Such a variable signifies a purely existential constraint. Hence if there are tuples that have the same values for non-existential variables, then there is no point in caching both tuples (and, in general, if there is a set of such tuples, only one need be cached). This simple observation may be incorporated to each of the above caching modes, resulting in four new definitions, in which only one representative out of a set of tuples, that agree on all the non-existent variables, is cached: All-body-e, Sat-body-e, Sat-tuple-e, and Sat-join-e. Existential variables in the heads of rules describing the cached data are specially marked as existential.

4 Is a Query Sufficiently Cached

Suppose the cache content is described by a set, say $W$, of conjunctive queries (not necessarily formed following our caching modes). Assume that the tuples that are cached for relation $q_i$ are described by rules with head predicate $qq_i$. W.l.o.g., all rules for predicate $qq_i$ have only variables in their head argument positions, have the same sequence of head variables (i.e., they are rectified, [UL89])\(^2\), and mention $q_i$ in their bodies. Let $Q$ be a query $q(X) \leftarrow q_1(X_1), \ldots, q_n(X_n), BIP$. W.l.o.g., all $X_i$’s contain only variables. We have the following straightforward observation.

**Proposition 1:** If $Q$ is sufficiently cached then $Q$ is equivalent to a program $P$ with the following rules:

- $q(X) \leftarrow qq_1(X_1), \ldots, qq_n(X_n), BIP$. This rule defines the target relation for the query.
- For each predicate $q_j$ the rules defining $qq_j$.

**Proof of Proposition 1:** If $Q$ is sufficiently cached. This means that $Q$ can always be solved correctly on the locally stored subset of the database tuples. These tuples are the union of the relations defined by the $qq_j$ defining rules. \(\square\)

\(^2\)Rectification might necessitate introducing more inequalities into bodies of rules in $W$. 

$$q_1(X_1), \ldots, q_{i-1}(X_{i-1}), q_i(X_i), q_{i+1}(X_{i+1}), \ldots q_n(X_n), BIP$$

are mutually satisfied by matching $q_j(X_j)$ with $t_j, j = 1, \ldots, n$. For each $q_i(X_i)$ add a new query to $W$, $qq_i(X_i) \leftarrow q_1(X_1), \ldots, q_m(X_m), BIP$.

4. *Sat-join (binary join satisfaction,)* Here we choose a pair of EDB body atoms, $q_i(X_i)$ and $q_j(X_j)$, such that $X_i \cap X_j$ is non-empty. Cache a tuple for $q_i$ (resp., $q_j$) only if it has a “matching” tuple in $q_j$ (resp., $q_i$), agreeing on values for the common attributes, and satisfying $BIP$.

For a chosen pair $q_i(X_i), q_j(X_j)$ such that $X_i \cap X_j$ is non-empty, add two new queries to $W$: $qq_i(X_i) \leftarrow q_i(X_j), BIP$ and $qq_j(X_j) \leftarrow q_i(X_i), BIP$. 


The converse of Proposition 1 does not hold. Consider the following example. Let query $Q$ be $q(X) \leftarrow q_1(X,Y), q_2(Y,Z)$. Let the cache content consist only of tuples for relation $q_1$ as described by the rule $qq_1(X,Y) \leftarrow q_1(X,Y), q_2(Y,Z)$. Now, $Q$ is equivalent to a conjunctive query that only mentions the $qq_i$'s, namely to $q(X) \leftarrow q_1(X,Y)$. However, $Q$ is not sufficiently cached. If we evaluate $Q$ on the cache, the result is an empty set of tuples since there are no $q_2$ tuples in the cache at all.

Consider query $Q$ as above. We can test whether $Q$ is sufficiently cached as follows. We rewrite $Q$ as $Q'$: $q(X) \leftarrow qq_1(X_1), \ldots, qq_n(X_n), BIP$. $Q'$ describes the computation of $Q$ relative to the cached data. Since all rules in $W$ are rectified, we can replace each $qq_i(X_i)$ in the body of $Q'$ with the disjunction of the bodies of the rules of $W$ defining $qq_i$, with appropriate renaming of variables. We can now transform the body into disjunctive normal form, and finally derive a conjunctive query out of each disjunct. The result is a set $CONJ$ of conjunctive queries whose union is equivalent to $Q'$.

We now test whether $Q$ is equivalent to the union of queries in $CONJ$. Observe that each rule for $qq_i$ mentions $q_i$ in its body. Therefore, all queries in $CONJ$ have the general form

$$q(X) \leftarrow QBody, a_1, \ldots, a_m$$

where $QBody$ is the body of $Q$, and $a_i$ is the body atoms (except for one $q_i$ occurrence) of some rule defining $qq_i$. Therefore, $Q$ obviously contains the union of the queries in $CONJ$.

We therefore need test whether $Q$ is contained in this union. Klug [Klu88] has shown that a query $Q$ is contained in the union of the queries in a set $Y$ if and only if for each “representative” database $D_i$, there is a query $Q1$ in $Y$ such that $Q$'s result on $D_i$ via valuation $\theta_i$ is contained in $Q1$'s result on $D_i$. Here, the $D_i$'s are taken out of a finite set of representative databases. This set of databases is based on the constants in $Q$, the constants in $Y$'s queries and the possible orderings, on the variables of $Q$ and these constants, that respect the inequalities of $Q$. A $D_i$ is obtained from $Q$ via a valuation $\theta_i$. So, we do have an expensive (exponential) procedure of testing query sufficiency.

Consider caching modes All-body-e, Sat-body-e, Sat-tuple-e and Sat-join-e. Recall that existential variables are specially marked in rule heads. In forming $CONJ$, when replacing $q_i(X_j)$ with a disjunction of bodies of rules from $W$ defining $qq_j$, an existential variable in the head of a rule defining $qq_j$ is replaced with a unique new variable for each usage.

## 5 Advantages of Caching Modes

In section 4 we considered the problem: given a query $Q$ and a set $W$ of cache description rules, is $Q$ sufficiently cached? This question basically asks whether a particular rewriting of $Q$, replacing $q$ atoms with $qq$ atoms, is equivalent to $Q$. Another interesting question is whether one can rewrite $Q$ by replacing only some of the $q$ atoms, and eliminating the rest, thereby obtaining a query $Q'$ equivalent to $Q$. This basically asks for a complete rewriting of $Q$ by only using some of the $qq_i$'s. Unfortunately, one has to guess a subset of the $q_i$’s whose replacement with $qq_i$’s, and eliminating the rest, will result in an equivalent query. A more difficult question is whether $Q$ can be rewritten into queries $Q_1, \ldots, Q_n$, that only use the $qq_i$’s, whose union equals to $Q$. One advantage of the caching modes approach is that to answer this question there is only one particular rewriting to check, uniformly replacing each
q_i with qq_i. While we may “miss” some esoteric, more efficient, rewritings, our searching for one is easier.

Another advantage of using caching modes is their simplicity. They present a simple decision of caching on a per body atom basis. A significant benefit of using the particular caching modes we suggest is implied by the following two properties of the caching rules set W, when constructed according to the caching modes:

1. (in body) If a tuple is cached then it participates in a satisfying assignment of a body of a rule in W. This is because if qq_i(X_i) is a head atom for a rule in W then q_i(X_i) must appear in the body of that rule.

2. (symmetry) If caching a tuple for relation q_i follows from a satisfying body assignment to a rule in W in which a tuple in q_i participates, then this q_i tuple is cached as well. This is because of the way W rules are constructed: if a q_i atom appears in a rule body and a qq_i atom in the head of a W rule, then there is another rule in W with the same body with the qq_i atom in the head.

Let Q be the conjunctive query q(X) ← q_1(X_1), ..., q_n(X_n), BIP. We characterize when a query is sufficiently cached given that cached data is described by rules, following our caching modes scheme. Let W be the collection of conjunctive queries describing the cached data.

Theorem 1: Q is sufficiently cached iff Q is equivalent to a program consisting of the rules of W and a collection of conjunctive queries, say Q_1, ..., Q_m, each with head predicate q which is the target predicate, and the Q_i’s only mention, except for built-in predicates, the q_{q_i}’s.

Proof of Theorem 1: (only if) Follows from Proposition 1. (if) Suppose that Q is equivalent to a union of conjunctive queries Q_1, ..., Q_m over the q_{q_i}’s. Consider the full database. If we eliminate from the database the set of tuples T that do not participate in satisfying the body of any q_{q_i} rule, this will not affect what is computed by the q_{q_i}’s on the database. Hence it will not affect what is computed on the database by each of Q_1, ..., Q_m which are defined in terms of the q_{q_i}’s. Since Q is equivalent to the union of the Q_1, ..., Q_m, this will not affect what Q computes on the database. We next show that the eliminated tuples are the non-cached tuples.

Let us focus on relevant relations, i.e. those that are mentioned in some rule in W. Suppose a relevant relation tuple is not cached. Then it does not participate in any satisfying body assignment; otherwise by property 2, ‘symmetry’, of W, it would be cached. Thus it is eliminated (i.e. in T). Suppose a relevant relation tuple is eliminated (i.e. in T). This means it does not participate in satisfying the body of any rule in W. So, by property 1, ‘in body’, of W, it is not cached. We conclude that a relevant relation tuple is eliminated iff it is not cached.

So, Q’s computation on the database state from which the tuples of T are eliminated is the same as Q’s computation on the cache. Hence, Q is sufficiently cached. □

---

3This is true even if Q’s body mentions an EDB relation q_i that is not mentioned in any rule of W. Q is not satisfiable in this case.
Whereas caching modes may introduce some extra cached data as compared to a more intricate way of defining caching rules, they substantially simplify the decision as to whether a new query can be at all answered based on the cached data. This is because, by the Theorem, our caching modes ensure that there is a rewriting of $Q$ just in case there is one particular rewriting, that of replacing each $q_i$ by $qq_i$. Formally:

**Corollary:** Suppose caching modes are used to form caching rules. If $Q$ is equivalent to a union of conjunctive queries $Q_1, \ldots, Q_m$, that only mention the $qq_i$'s, then $Q$ is equivalent to a single conjunctive query $Q'$ that is obtained from $Q$ by replacing each $q_i$ by $qq_i$. □

## 6 Is A New Query Effectively Cached

Suppose that queries $Q_1, \ldots, Q_m$ are cached in various modes. Given a query $Q$ we would like to check if $Q$ is effectively cached in mode $M_i$ for body atom $q_i$ (one of All-body, Sat-body, Sat-tuple or Sat-join); by this we mean that all the tuples that would have been cached for $Q$, had we cached for body atom $q_i$ in mode $M_i$, are always guaranteed to be cached due to the caching of $Q_1, \ldots, Q_m$. Observe that if $Q$ is effectively cached in modes $M_i$, then $Q$ is also sufficiently cached. Thus effective caching of $Q$ in some modes $M_i$ implies that $Q$ is sufficiently cached, although the converse is not always true.

A procedure to answer this question is as follows:

- Consider, for all cached queries $Q_j$, the $qq_i$ type rules for each predicate $q_i$ mentioned in the body of query $Q_j$. The form of the $qq_i$ rules is according to the mode in which $q_i$ in the body of $Q_j$ is cached (existential modes are possible). Let $W$ be the set of resulting rules which describe the cache.

- Construct $qq_i$ type rules for each predicate $q_i$ mentioned in the body of $Q$. The construction of these new $qq_i$ rules is according to the mode $M_i$ for which $q_i$ in the body of $Q$ is checked for caching (again, existential modes are possible). Let $T$ be the set of resulting rules$^4$.

- Check, for each predicate $qq_i$ mentioned in $T$ that the union of rules for $qq_i$ defined in $T$ is contained (as a query) in the union of rules for $qq_i$ in $W$. This holds iff each rule in $T$, viewed as a query, is contained in the union of rules for $qq_i$ in $W$. If the answer is YES (resp., NO) then $Q$ is effectively (resp., not effectively) cached in modes $M_i$ by virtue of caching $Q_1, \ldots, Q_m$ in their respective modes.

## 7 A Simple Sub-class of Caching Rules

Let us fix a database schema $\bar{D}$. Consider a query $Q$ given by intervals, e.g.

$$q(A, B, D) \leftrightarrow R(A, B, D), (6 \leq A \leq 8), (8 \leq B \leq 99), (8 \leq D \leq 77)$$

$^4$We observe that for each $qq_i$ there may be more than one rule in $T$, since there may be a number of distinct occurrences of $q_i$ in the body of the rule defining $Q$ and each of these occurrences gives rise to a rule in $T$. 

8
and queries $Q_1, \ldots, Q_n$ of the same form, i.e. $R(A, B, D)$ conjoined with a collection of interval inequalities on the columns of $R$. Queries of this type specify a subset of the tuples for $R$ that reside in a generalized “box” in $k$ dimensions, where $k$ is the number of columns in $R$. Hence we call these queries box queries. The box queries are reminiscent of the left and right semiinterval queries of [Klu88] although they are less general in that a single relation is allowed in the query’s body and are more general in that intervals are considered.

We consider the problem of whether the fact that all the queries $Q_i$ are sufficiently cached implies that $Q$ is sufficiently cached. We first treat the case where all inequalities involve $\leq$ or $\geq$. W.l.o.g., assume that each column of $R$ is bound to an interval in all queries, this requirement can always be met by detecting the largest and smallest boundaries for each column $c$, say $\text{MAX}_c$ and $\text{MIN}_c$, and adding $(\text{MIN}_c - 1) \leq X_c \leq (\text{MAX}_c + 1)$ to each query not mentioning column $c$ ($X_c$ is the variable used for column $c$).

The problem is solved as follows:

1. For column $c$ of $R$, let $C_c = C_c[1] < \cdots < C_c[n(c)]$ be all the constants mentioned as boundaries of an interval for column $c$ in either $Q$ or some $Q_i$.

2. Let $Q \text{MIN}_c$ (resp., $Q \text{MAX}_c$) be the left (resp., right) interval bound on column $c$ in $Q$. Let $\text{min}_c$ (resp., $\text{max}_c$) be the index of $Q \text{MIN}_c$ (resp., $Q \text{MAX}_c$) in $C_c$.

3. For $x_1 = \text{min}_1$ to $\text{max}_1 - 1$ do
   
   Consider the “small box” whose $r$’th dimension is bound between $C_c[x_r]$ and $C_c[x_r + 1]$. This “small box” is a piece of the space defined by $Q$’s “box”. Check that this “small box” is contained in the “box” defined by some $Q_i$. This is verified by checking that for each dimension $1 \leq r \leq k$, $C_c[x_r]$ (resp., $C_c[x_r + 1]$) is $\geq$ (resp., $\leq$) than the left (resp., right) interval boundary of the $Q_i$ query considered.

   If no such $Q_i$ is found the sufficiency test fails, exit.

4. The sufficiency test succeeds.

It can be easily verified that the containment check above takes $O(n^{(k+1)})$ where $n$ is the input length (the input contains the $Q_i$’s and $Q$). Since $k$ is a constant this gives us a polynomial algorithm. Interestingly, if instead of exiting upon detecting insufficiency, the “violating” “small box” is merely recorded, then upon exiting the embedded loops we have identified a collection of “missing small boxes”. This collection may be used to form one or more focussed “remainder queries” [DFJST96] to obtain the missing data. We defer the topic of forming such queries for future research.

If strict inequalities are used, i.e. $<$ or $>$, we do the following. With each constant $w$ we associate an infinitely small value $m_w$, in particular $m_w$ is smaller than the absolute value of the difference between any mentioned constants. We then translate $X < w$ into $X \leq w - m_w$ and $X > w$ into $X \geq w + m_w$. The rest of the procedure remains the same.

In fact, solving this problem allows us to also solve the following problems:

- The query sufficiency problem when only the Sat-body caching mode is used, with inequalities that only consider columns (e.g. $\text{COL}1 < \text{COL}2$) deleted from caching rules, and the checked query has the form of $Q$. 
• Same as above, but the checked query \( Q' \) is arbitrary. We construct hypothetical caching rules for \( Q' \), employing only the Sat-body mode (again with inequalities that only mention columns deleted) and check each such rule individually, “playing” the “role” of \( Q \), against the \( Q_i \)’s.

When we do not have a fixed database schema, the problem becomes coNP-Complete [GJ79]. An instance of the problem consists of \( Q, Q_1, \ldots, Q_n \). The problem is whether the union of the “boxes” defined by the \( Q_i \)’s contains the “box” of \( Q \). The complement of this problem, call it non-cover, is whether there is a point in the “box” of \( Q \) that is not covered by the “boxes” of the \( Q_i \)’s. This problem is in NP. To show completeness we reduce 3SAT [GJ79] to non-cover. Think about the interval \([0, 1]\) as encoding \( x_i = \text{true} \) and \([1, 2]\) as encoding \( x_i = \text{false} \). With each clause \( \{x_1, x_2, x_3\} \) we associate a query \( Q_i = g_1 \land g_2 \land g_3 \). If \( x_i \) is positive \( g_i = (1 \leq X_i \leq 2) \); If \( x_i \) is negative \( g_i = (0 \leq X_i \leq 1) \). The query \( Q \) is just the whole space, between 0 and 2, in each dimension. The “box” of \( Q_i \) defines a region where the clause is not satisfied. So, if the whole space is covered, this means that for each boolean variable assignment, at least one clause is false. So \( Q \) is covered just in case the original 3SAT formula is not satisfiable.

8 Complexity of the Query Sufficiency Problem

R. van der Meyden has shown that the containment decision problem when queries contain inequalities, i.e. is it the case that for all databases, the result of one query is contained in the result of another query, is \( \Pi_2^P \)-complete\(^5\) [vdM92]. Klug has previously shown that the containment problem is in \( \Pi_2^P \) [Klu88]. R. van der Meyden’s proof treated the sub-class of queries in which the query’s summary (i.e. head) has no attributes (i.e. head variables), such a query may retrieve either an empty tuple or no tuples whatsoever\(^6\). We use this fact in Theorem 2 (below) to show that the problem of testing sufficiency is \( \Pi_2^P \)-complete.

Theorem 2: The sufficiency problem is \( \Pi_2^P \)-complete.

Proof of Theorem 2: An instance of the containment problem consists of conjunctive queries \( Q \) and \( Q_1 \), the problem is to determine whether \( Q \) is contained in \( Q_1 \). This problem is \( \Pi_2^P \)-complete. We now show a reduction of the containment problem to the sufficiency problem. Consider a containment problem instance \( I \). W.L.o.g., \( Q \) and \( Q_1 \) both have head \( q() \), i.e. no attributes. We construct an instance of the sufficiency problem as follows.

Sufficiency is tested for the query \( Q \), which is

\[
q() \leftarrow q_1(X_1), \ldots, q_m(X_m), BIP.
\]

The cached queries are, for \( 1 \leq i \leq m \):

\[
h_i() \leftarrow q_i(X_i), Q_1(U_i)
\]

\(^5\)See [GJ79] for definitions.

\(^6\)Note that for positive existential queries the complexity of the containment problem for the sub-class is the same as that of the whole class (R. van der Meyden, personal communication).
$Q_1(U_i)$ is a variant of the body of query $Q_1$, $U_i$ is a sequence of unique new variables that do not appear anywhere else.

Suppose that the Sat-tuple caching mode is uniformly employed for all cached queries. Therefore, among the cached rules are, for $1 \leq i \leq m$:

$$qq_i(X_i) \leftarrow q_i(X_i), Q_1(U_i)$$

There are possibly additional rules for $qq_i$ due to the atoms in $Q_1(U_i)$. However, these do not introduce more relevant tuples for the computation (of $Q$ based on cached tuples) as compared to the tuples supplied by the rules we have listed. This is because all tuples, that could be supplied by these additional rules, that could possibly be matched with $q_i(X_i)$ in the body of $Q$, are supplied by the head $qq_i(X_i)$ once the body of the $qq_i$ rule that we have listed is satisfied.

Now, $Q$ is sufficiently cached, where the $qq_i$’s define the cached tuples, iff $Q$ is equivalent to $q() \leftarrow BIP, qq_1(X_1), \ldots, qq_m(X_m)$, which equals

$$q() \leftarrow BIP, q_1(X_1), \ldots, q_m(X_m), (Q_1(U_1), \ldots, Q_1(U_m)).$$

This query can be expressed by the following three rules program, with $q$ the target predicate:

1. $q() \leftarrow v(), h()$.

2. $v() \leftarrow BIP, q_1(X_1), \ldots, q_m(X_m)$.

3. $h() \leftarrow Q_1(U_1), \ldots, Q_1(U_m)$.

But, the second rule above actually expresses the original query $Q$. So, this program’s query is equivalent to $Q$ intersected with the query expressed by the third rule. But, the third rule can be written simply as:

$$h() \leftarrow Q_1(U_1)$$

namely expressing the original $Q_1$. So, $Q$ is sufficiently cached using the $qq_i$’s iff $Q$ is contained in $Q_1$. □

9 Conclusions

We have defined four basic modes of caching: All-body, Sat-body, Sat-tuple, and Sat-join. For each mode, the cached data can be characterized by a union of conjunctive queries. We have also treated existential variables and considered the effects of caching in existential modes.
We have explained how to test sufficiency for a query, that is whether the query can be answered by applying it, unchanged, to data cached for queries, where the cached tuples may be described as a union of conjunctive queries. Caching modes, in addition to simplifying the cache content definition, reduce the complexity of testing whether a query can be answered by using only cached data. When caching modes are used, a complete rewriting exists for a query iff the query is sufficiently cached.

The sufficiency problem is highly complex ($\Pi_2^p$-complete). The problem of determining effective caching, i.e. whether tuples that would have been cached on behalf of query $Q$ are already cached due to other cached queries, was also treated.

Intuitively, determining sufficiency leads to a containment problem of a conjunctive query in a union of conjunctive queries. Testing for effective caching leads to a series of problems, where each problem treats containment of a conjunctive query in a union of conjunctive queries. Due to the complex nature of these problems, one would like to formulate heuristics for solving them and identify subclasses of these problems that are provably easy. Some initial work along these lines appears in [Klu88]. In this paper we have considered the special case of “box” queries where sufficiency can be tested in polynomial time for a fixed database schema.

## 9 References


