Constraint Checking with Partial Information [*]

Extended Abstract

Appears in *Proceedings of the Thirteenth Symposium on Principles of Database Systems, 1994*

Ashish Gupta **Yehoshua Sagiv**[†] **Jeffrey D. Ullman** **Jennifer Widom**
Dept. of Computer Science, Stanford University, Stanford CA 94305.

Abstract

Constraints are a valuable tool for managing information across multiple databases, as well as for general purposes of assuring data integrity. However, efficient implementation of constraint checking is difficult. In this paper we explore techniques for assuring constraint satisfaction without performing a complete evaluation of the constraints. We consider methods that use only constraint definitions, methods that use constraints and updates, and methods that use constraints, updates, and "local" data.

1 Introduction and Motivation

Efficient constraint checking is an important problem in both traditional, centralized databases and loosely coupled, distributed databases. Because constraints can be as complex as queries, yet in principle could be violated whenever the database changes, it is essential that constraint checks occur only when absolutely necessary. Thus, there has been considerable effort devoted to determining efficiently when a given update can affect the validity of one or more constraints.

Looking Only at Constraints and Updates

Much can be discovered looking only at the constraints themselves or only at the constraint and update. Often, tests that confirm an update does not cause a violation of a constraint can be obtained from known techniques for query containment. Only if the this test is inconclusive will we make a second test that looks at the data as well as the update.

Using Local Data

When query containment by itself fails, there are often reasons to look at tests that examine only a subset of the database. Especially, the database may be divided into "local" and "remote" data with respect to the site of the update. Accessing remote data may be expensive or impossible, so we wish to conduct a *local test* using only the constraints, the update, and the local data. Only if this test is inconclusive do we need to make a second test that looks at the remote data.

Tests Using the Query Language

Especially important is the question of whether the tests involved can be expressed in the query language of the database system. If we can express tests as queries, then we have hope of being able to use the structure of the database, e.g., indexes, to make the tests far more efficient than their theoretical upper bounds. If tests cannot be expressed in the same language used for queries and constraints, then tests are unlikely to be of adequate efficiency to be used in practice.

Major Results

While we consider a variety of constraint classes, the following are the most significant advances.

1. When constraints are conjunctive queries (hereafter abbreviated CQ) without arithmetic comparisons, we can construct their best local test in time that is exponential in the size of the query, but independent of the data. Moreover, the test itself can be expressed in relational algebra, so it is likely to be within the query language of any database system.

2. For constraints that are CQ's with arithmetic, we offer an alternative to the containment test

[*]Work supported by NSF grants IRI–91–16646 and IRI–92–23405, ARO grant DAAL03–91–G–0177, USAF grant F33615–93–1–1339, the US-Israel BSF grant 92–00360, and a grant of Mitsubishi Electric Corp.

[†]Permanent address: Dept. of CS, Hebrew Univ., Jerusalem, Israel.

of Klug. We reduce the containment to a logical expression about arithmetic, whose verification is fast in the (usual) case of constraints that involve few repetitions of the same predicate.

3. For some interesting subsets of CQ's with arithmetic, we give best local tests that are expressible in relational algebra or in recursive datalog.

2 Basic Definitions and Concepts

In this section we set the framework for the results.

Constraints

A *constraint* is a query whose result is a 0-ary predicate that we call **panic**. If the query produces $\emptyset$ on a given database D, then D is said to *satisfy* the constraint, or the constraint is said to *hold* for D.

Languages for Expressing Constraints

The language in which constraints are expressed affects both the question of whether tests for satisfaction can be written in the system's query language and the complexity of such tests. We shall model query languages through logic, and the principal classes of constraint/query languages of interest to us are:

1. Conjunctive queries (Chandra and Merlin [1977]).
2. Unions of CQ's (Sagiv and Yannakakis [1981]). These are equivalent to nonrecursive datalog programs.
3. Conjunctive queries with arithmetic comparisons (Klug [1988]).
4. Conjunctive queries with negated subgoals (Levy and Sagiv [1993]).
5. Other combinations of (2) through (4), that is, CQ's or unions of CQ's, with or without arithmetic comparisons and with or without negated subgoals.
6. Recursive datalog, including cases with or without arithmetic comparisons and with or without negated subgoals.

There are actually 12 combinations of features, organized as suggested in Fig. 2.1.

Example 2.1: The following constraint is a CQ constraint.

```
panic :- emp(E,sales) &
         emp(E,accounting)
```

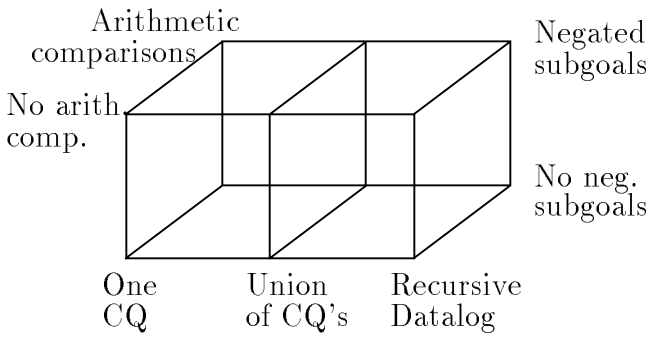

Fig. 2.1. Classes of logical languages.

It states that no employee can be in both the Sales and Accounting departments. Here, we assume *emp* is a predicate representing the traditional Employee-Department relation. We follow the common Prolog convention that names beginning with a lower-case letter are constants (including predicate names), and names beginning with a capital are variables. □

Example 2.2: The following constraint says that every employee with a salary under 100 must be assigned to a department.

```
panic :- emp(E,D,S) &
         not dept(D) &
         S < 100
```

This constraint query is not a CQ, because it has a negated subgoal. It also has a subgoal, $S < 100$, whose predicate is an arithmetic comparison, rather than an abstract symbol. □

Example 2.3: The following constraint query says that every employee must have a salary in the allowed range for the employee's department.

```
panic :- emp(E,D,S) &
         salRange(D,Low,High) &
         S < Low

panic :- emp(E,D,S) &
         salRange(D,Low,High) &
         S > High
```

Now, the constraint no longer resembles a variety of CQ. However, it is in the class of nonrecursive datalog with arithmetic comparison predicates permitted. That class is the same as finite unions of CQ's, again with arithmetic comparisons permitted. □

Example 2.4: The following constraint says that no employee can be his or her own boss.

```
panic :- boss(E,E)
boss(E,M) :- emp(E,D,S) & manager(D,M)
boss(E,F) :- boss(E,G) & boss(G,F)
```

This constraint query is written in recursive datalog. □

Limits on Available Information

We shall consider three different kinds of problems, depending on how much information we are willing to look at. Our goal is to obtain a test using only the allowed information that assures a constraint is not violated.

- The test may assume that no constraints were violated before the update.
- Success of the test must imply that the constraint is satisfied. However, if the test fails, it may or may not be the case that the constraint is satisfied; we need to make a different test involving more information to find out.

Here are the three levels of information that we are interested in:

1. The least information is the constraints alone. This problem corresponds to implication of constraint queries, since the only way we could be sure a constraint C is satisfied without looking at any update or any data is if there were other constraints $C_1, \ldots, C_n$, known to be satisfied, such that whenever C implies **panic**, so does at least one of $C_1, \ldots, C_n$.
2. An intermediate kind of problem is when we are allowed to see both the constraints and the update. This problem has been called *query independent of update* in Elkan [1990], Tompa and Blakeley [1988], Levy and Sagiv [1993]. That is, given a constraint C, known to be satisfied before an update, can we be sure that after a certain update C will continue to hold? More generally, we may also know that certain constraints other than C also hold before the update, and we may use that information.
3. The most general problem we shall consider is one in which there are some "local" predicates and some "remote" predicates. We wish to tell whether a constraint C is satisfied after an update, given that it, and perhaps some other constraints, were satisfied before. To make the decision, we are allowed to look not only at C and the update, but also at the data in the relations corresponding to the local predicates. This problem arises in distributed constraint maintenance, where we would like to avoid accessing remote data, which in some scenarios is expensive or impossible to access (Gupta and Widom [1992], Gupta and Ullman [1992]).

Correct and Complete Tests

Tests are algorithms that look at the given constraints, the update, and the permitted subset of the data, and respond either:

1. "Yes," the constraint in question continues to hold, on the assumption that this and other given constraints held previously, or
2. "I don't know" whether the constraint continues to hold.

There is a third possible outcome: "no, the constraint definitely becomes violated." However, for the common classes of constraints that we consider, this outcome is not possible unless the constraint involves only local data.

We expect that each test is *correct*, in the sense that whenever it says "yes," the constraint does hold. Another important property of tests is that they be *complete*, meaning that whenever the test says "I don't know," there is some state of the information not accessed by the test for which the constraint ceases to hold after the update. In the case where we are using the constraint, the update, and the local data, we refer to a complete test as a *complete local test*.

Applications

The theory developed here has a number of related uses.

1. As we describe it, the problem is to manage a set of constraints $C_1, \ldots, C_n$ on a database D. As D changes, we need to know which if any of the constraints are violated. Generally, we can assume that all constraints hold prior to the most recent change.
2. A related problem concerns *active databases*, where we have a collection of *rules* of the form "if C holds, then perform action A." We can see such a rule as a constraint **panic** `:-` C with the action A performed in response to deriving **panic**. We are especially interested in the case where the modifications to the database are the result of the actions of the rules. Unlike (1), we cannot assume that all "constraints" (the conditions in the rules) hold prior to an action, because of the way active rules are normally detected and fired (Ceri and Widom [1990, 1991]).

3. Another problem of similar type is view maintenance. We are given an expression defining a view V of a database D, and we want to know whether and how updates to D can affect the value of V. This problem has been studied by, e.g., Tompa and Blakeley [1988], Blakeley, Coburn, and Larson [1989], and Ceri and Widom [1991].

3 Constraint Subsumption

When we are allowed to look only at the constraints themselves, our only opportunity to take advantage of the information is through subsumption of one constraint by one or more other constraints. If C is a constraint query, and $\mathbf{C} = \{C_1, \ldots, C_n\}$ is a set of constraint queries, we say $\mathbf{C}$ *subsumes* C if whenever C is violated, some C_i in $\mathbf{C}$ is also violated. In that case, there is no need to check C.

Since constraint queries only produce $\{\mathbf{panic}\}$ or $\emptyset$ as a result, subsumption is a special case of containment of programs. Recall that one program P contains another, Q, if on any database the result of P is a superset of the result of Q. We write $Q \subseteq P$ in that case. Then the following is obvious:

Theorem 3.1: Constraint set $\mathbf{C} = \{C_1, \ldots, C_n\}$ subsumes constraint C if and only if, viewed as programs, $C \subseteq C_1 \cup \cdots \cup C_n$. $\Box$

There are many known results about program containment that apply directly to constraint subsumption. For example, if the constraints are CQ's (Chandra and Merlin [1977]) or unions of CQ's (Sagiv and Yannakakis [1981]; these are equivalent to nonrecursive datalog programs), the problem is "only" NP-complete. Since constraints tend to be short, the exponential complexity of the problem may not present a bar to solution in general.

If we extend CQ's to allow arithmetic comparisons as subgoals or we allow the subsuming constraints to be a recursive datalog program, then the problem is still solvable in exponential time. The former case is Π_2^p-complete (Klug [1988] and van der Mayden [1992]) and the latter is exponential-time-complete (Chandra, Lewis, and Makowsky [1981] and Sagiv [1988]).

If we allow the subsumed constraint to be a recursive datalog program, while the subsuming constraints are nonrecursive datalog, the problem remains decidable (Courcelle [1991]). The complexity of this problem was resolved by Chaudhuri and Vardi [1992], who showed it is complete for triply exponential time, with some less complex special cases. On the other hand, when both the subsuming and subsumed constraints are recursive datalog, the problem becomes undecidable (Shmueli [1987]).

Containment Versus Constraint Subsumption

Since constraint queries have a 0-ary goal predicate, one might wonder if the above cited results are too conservative; i.e., constraint subsumption is easier than query containment. It appears that for any common class of queries, that is not the case. Rather, constraint subsumption is just as hard as the corresponding query containment problem. For powerful query languages, where intermediate predicates are allowed, and several rules may be used (e.g., nonrecursive datalog), it is easy to reduce query containment to the corresponding constraint subsumption problem by adding rules, thus providing a lower bound on the complexity of constraint subsumption.

When constraints are single CQ's, it may not be clear that the problems are the same. In fact, the NP-completeness of containment for CQ's with 0-ary heads was proved explicitly in Chandra and Merlin [1977]. However, that result still leaves open the question for special classes of CQ's or generalizations of CQ's.

We can reduce CQ containment to constraint subsumption in a very robust way. If Q is a CQ of the form h :- B, we rename the predicate of the head h if it appears in the body B. We then "move" the head into the body, creating the CQ Q' that is

$$\mathbf{panic} \text{ :- } h \ \& \ B$$

If Q and R are two CQ's, it is easy to check that $Q \subseteq R$ if and only if $Q' \subseteq R'$. Thus, we can claim the following:

Theorem 3.2: For any class of CQ's that is closed under the addition of an additional subgoal that is of the ordinary type (uninterpreted predicate with arguments, not negated), the containment problem logspace-reduces to the corresponding constraint subsumption problem. $\Box$

4 Using the Update

There are similar observations that can be made when we are allowed to use a collection of constraints and an update to determine that another constraint is satisfied. There are two approaches that might be taken.

1. Convert each constraint C into another constraint C' that says "C is violated after this update." Then, we test whether C' is contained in the union of C and any other constraints that we assumed held before the update.

2. Find the complete test for whether the constraint C continues to hold after the update.

We shall explore (2) in the more general case of the existence of local data; see Sections 5 and 6. The first approach will be considered here.

Rewriting Constraints to Reflect Updates

Following techniques used in Levy and Sagiv [1993], we take a constraint C and an update, and we try to construct a new constraint C' that holds before the update if and only if C holds after the update. The test for whether C holds after the update, given that it and perhaps some other constraints $C_1, \ldots, C_n$ held before the update, is to see whether C' is contained in $C \cup C_1 \cup \cdots \cup C_n$.

When we construct C' from C, it may not be possible for C' to be in the same class as C, among the twelve classes indicated in Fig. 2.1. However, some of the classes are preserved. An example will illustrate the points and also indicate how constraints can be modified to account for updates.

Example 4.1: Suppose there are two constraints in an employee database:

```
C1: panic :- emp(E,D,S)  &  not dept(D)
C2: panic :- emp(E,D,S)  &  S > 100
```

C_1 is a referential integrity constraint; it says that every employee must be in a department that is mentioned in the *dept* relation. C_2 says that no employee may have a salary greater than 100.

Suppose there is an update in which *toy* is added to the set of departments. We can define a constraint that represents C_1 after the update as

```
dept1(D) :- dept(D)
dept1(toy)
panic :- emp(E,D,S) & not dept1(D)
```

Call this constraint C_3. Then in order to be sure that C_1 has not become violated by the update we need to check $C_3 \subseteq C_1 \cup C_2$. This happens to be the case, and in fact, C_2 is not needed in the containment. The methods of Levy and Sagiv [1993] suffice.

Note that C_3 is in the language of "union of CQ's with negation but no arithmetic comparisons," even though the constraint C_1 from which it was derived is in the narrower class of "CQ's with negation and no arithmetic comparisons." Another way to express C_3 is by the single rule

```
panic :- emp(E,D,S) &
         not dept(D) &
         D <> toy
```

Now, the constraint is in the language of "CQ's with both negation and arithmetic comparisons." □

However, we cannot do better than the two approaches suggested by Example 4.1. That is:

Theorem 4.1: Constraint C_3, stating that after insertion of *toy* into relation *dept* there is no employee in a department that does not appear in *dept*, cannot be expressed as a single CQ (over the predicates *emp* and *dept* denoting their values *before* insertion) without arithmetic comparisons, even if negation is allowed.

Proof: (Sketch) Let C be such a CQ expressing C_3. We claim that C cannot have an unnegated subgoal with predicate *dept*, or else it cannot produce **panic** whenever *dept* is empty. Similarly, C cannot have a subgoal *not dept*(d) where d is any constant such as *toy*, or we can easily construct an example where C fails to cause **panic** when it should. Thus, the only *dept* subgoals are of the form *not dept*(D) for some variable D.

Now, consider the database with tuples

$emp(e, shoe, s)$
$emp(e, toy, s)$

and no other tuple for either *emp* or *dept*. C_3 produces **panic**, so C must also. Consider an instantiation of C's variables that satisfies the body of C. If any of the subgoals of the form *not dept*(D) instantiates to *not dept*$(shoe)$, we claim we can replace these by *not dept*(toy) and still produce **panic**. If D appears nowhere else, surely this change can be made. If D appears elsewhere, it can only be in another *not dept*(D) subgoal, which presents no problem, or in some subgoal of the form $emp(E, D, S)$. In the latter case, the instantiation of the subgoal must have been either

1. $emp(e, shoe, s)$, in which case we can legally instantiate it instead to $emp(e, toy, s)$, or

2. *not* $emp(a, b, c)$ for some constants a, b, and c, at least one of which (corresponding to variable D) is *shoe*. In this case, we can again replace *shoe* by *toy*, and the negated subgoal will continue to be true.

Now, consider the database with the same two tuples, $emp(e, shoe, s)$ and $emp(e, toy, s)$ for *emp*, but with additional tuple *dept*$(shoe)$. Since we established in the paragraph above that there is an instantiation of C that does not use *shoe* as an argument of a *dept* subgoal, the same instantiation produces **panic** on this database. However, C_3 does not produce **panic**. Thus, we contradict the assumption that

C was an equivalent of C_3. Therefore, there is no way to express C_3 as a single CQ without arithmetic comparisons. □

The first of the techniques in Example 4.1 generalizes; it shows that any language that allows us to add rules, even nonrecursive ones and rules without negation or arithmetic comparisons, allows us to express a constraint after an insertion in the same language. We thus claim

Theorem 4.2: The eight circled classes in Fig. 4.1 are preserved by insertions; that is, a constraint in the class after an insertion can be expressed in the same language. □

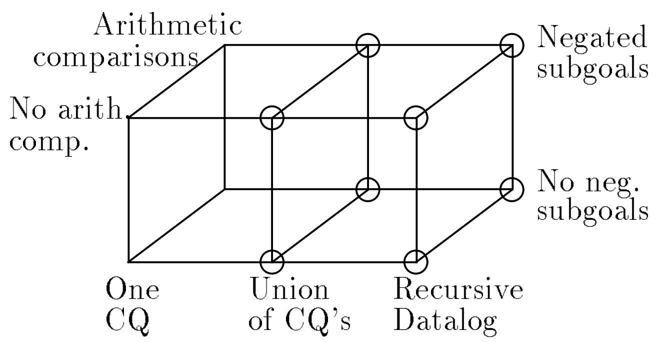

Fig. 4.1. Classes preserved under insertion.

Closure Under Deletion

When the update is a deletion, the situation is not much worse. Here is an example that illustrates the principal technique.

Example 4.2: Continuing with Example 4.1, suppose we delete the tuple $(jones, shoe, 50)$ from the emp relation. Then we need to construct a new predicate $emp1$ that reflects the deletion of this tuple. Here is one way to do so.

```
emp1(E,D,S) :- emp(E,D,S) & E<>jones
emp1(E,D,S) :- emp(E,D,S) & D<>shoe
emp1(E,D,S) :- emp(E,D,S) & S<>50
```

This predicate $emp1$ can substitute for emp in either C_1 or C_2 to create new constraints C_4 and C_5 that reflect the situation after this deletion. We then need to check $C_4 \subseteq C_1 \cup C_2$ and $C_5 \subseteq C_1 \cup C_2$. Note that in this construction, CQ's are brought into the class of nonrecursive datalog with arithmetic comparisons. □

There is a similar trick that uses negated subgoals instead of arithmetic comparisons. For instance, we could replace the subgoal $E <> jones$ in the first rule of Example 4.2 by *not* $isJones(E)$, where predicate $isJones$ is defined by

```
isJones(jones)
```

It does not appear to be possible to avoid using one of negation and arithmetic comparisons. Thus, we can only claim the following.

Theorem 4.3: The six classes circled in Fig. 4.2 can express constraints that result from a deletion. □

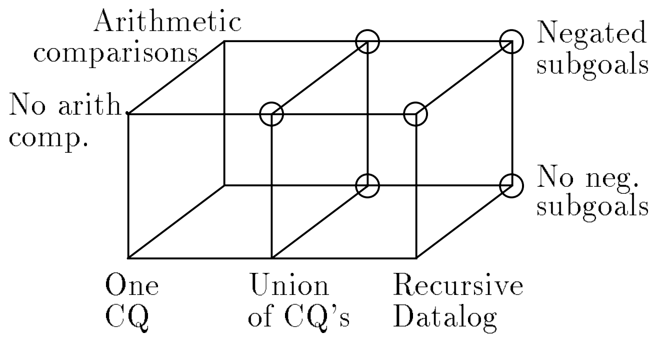

Fig. 4.2. Classes preserved under deletion.

5 Using Local Data

The main results of this paper concern the problem of checking constraints given an update and some predicates that are defined to be "local." We assume that for one reason or another (e.g., the expense of querying remote databases) we prefer to check the constraints using only the local data, and we would therefore like to derive the complete local test for our constraints. We show how to derive complete local tests for some important classes of constraints.

We focus on *conjunctive query constraints* (CQC's) of the following form

$$\textbf{panic} \text{ :- } l \ \& \ r_1 \ \&\ldots\& \ r_n \ \& \ c_1 \ \&\ldots\& \ c_k$$

Here, l is the one subgoal with a local predicate (although we can see l as a conjunction of local subgoals). Each of the r_i's is a subgoal with a remote predicate, and each of the c_i's is an arithmetic comparison. The following conditions are assumed throughout the balance of the paper:

- Variables in the c_i's must also appear in l or one of the r_j's.

- No variable appears twice among l and the r_i's. Rather, multiple occurrences are handled by using distinct variables and equating them by arithmetic equality constraints.
- Constants do not appear among the ordinary subgoals. Again, the fix is easy. Just replace constants by new variables and equate those variables to the desired constant.
- The update is the insertion of a tuple into the relation for l.

Containment of CQC's

We first prove a result that applies to general CQ's with arithmetic comparisons, including those with nontrivial heads. If C is a CQ, let $A(C)$ be the set of subgoals of C that are arithmetic comparisons, and let $O(C)$ be the other, "ordinary" subgoals.

Theorem 5.1: Let C_1 and C_2 be CQC's. Then $C_1 \subseteq C_2$ if and only if the following holds. Let H be the set of all containment mappings (mappings from variables to variables that maps head to head and subgoals into subgoals; see Ullman [1989]) from $O(C_2)$ to $O(C_1)$. Then H is nonempty, and $A(C_1)$ logically implies $\vee_{h \text{ in } H}\ h\big(A(C_2)\big)$.

Proof: A proof sketch follows the example below. □

- Theorem 5.1 generalizes to the containment of C_1 in a union of CQC's in the obvious way. We must include containment mappings from any member of the union to C_1.
- Theorem 5.1 also holds for general CQ's with arithmetic, i.e., if the heads are not 0-ary.
- Theorem 5.1 is generalized to uniform containment of recursive programs in Levy and Sagiv [1993].

Example 5.1: There is in Ullman [1989] a theorem similar to Theorem 5.1 that gives a sufficient but not necessary condition for $C_1 \subseteq C_2$. Example 14.7 from Ullman [1989] gives a counterexample to the version of our Theorem 5.1 in which we are allowed to use only one of the containment mappings in H. Here is Example 14.7 restated as constraints for simplicity.

C_1': **panic** `:- r(U,V) & r(V,U)`
C_2: **panic** `:- r(U,V) & U <= V`

Intuitively, C_1' causes panic when the relation for r has a tuple and its reverse, while C_2 causes panic when r has a tuple with first component no greater than the second component. Clearly, whenever r has both (a,b) and (b,a), there will be in r a tuple with first component less than or equal to the second. Thus, we expect $C_1' \subseteq C_2$, but the test of Ullman [1989] does not indicate so.

Our first task is to rewrite C_1' so that the variables U and V appear only once in the ordinary subgoals. We introduce new variables T and V, and in the rewritten constraint, C_1, these are equated to U and V respectively. C_2 needs no rewriting, so the constraints are now:

C_1: **panic** `:- r(U,V) & r(S,T) &`
`U=T & V=S`

C_2: **panic** `:- r(U,V) & U <= V`

To apply Theorem 5.1, note that there are two containment mappings from the ordinary subgoals of C_2 to those of C_1:

1. $h(U) = U$; $h(V) = V$.
2. $g(U) = S$; $g(V) = T$.

Theorem 5.1 thus tells us we must verify

$$A(C_1) \ \Rightarrow\ h\big(A(C_2)\big)\ \vee\ g\big(A(C_2)\big)$$

That is,

$$U = T\ \wedge\ V = S\ \Rightarrow\ U \le V\ \vee\ S \le T$$

The above simplifies to $U \le V\ \vee\ V \le U$, which is true assuming that $\le$ is a total order. □

Proof: (Sketch of Theorem 5.1) First, observe that a containment mapping from $O(C_2)$ to $O(C_1)$ is defined by the way ordinary subgoals are mapped to ordinary subgoals, and any mapping is legal as long as it preserves predicates (assuming that a predicate has a unique number of arguments; if not, then rename predicates with different numbers of arguments). The reason is that we assume CQC's do not have multiple occurrences of variables. As a result, there will be at least one containment mapping, unless C_2 has a predicate not found in C_1.

In the latter case, it is easy to show that either

1. $A(C_1)$ is always false. Then, since
$$\vee_{h \text{ in } H}\ h\big(A(C_2)\big)$$
is false when H is empty, Theorem 5.1 holds.
2. $A(C_1)$ is not always false. Then we can find a database with an empty relation for those predicates in $O(C_2)$ but not in $O(C_1)$ that demonstrates C_1 is not contained in C_2. Since the test of Theorem 5.1 indicates no containment, the theorem holds for this case too.

Now, let us assume that the set of predicates for C_1 and C_2 are the same, and therefore H is not empty.

If: Suppose $A(C_1) \Rightarrow \vee_{h \text{ in } H}\ h\big(A(C_2)\big)$. Let g be some instantiation of variables of C_1 that makes the entire body true for some database D. Then since $g\big(A(C_1)\big)$ is true, there must be some h in H such that $g\big(h(A(C_2))\big)$ is true. Also, $g\big(h(O(C_2))\big)$ consists of tuples in database D, so $g \circ h$ is an instantiation of the variables of C_2 that makes C_2's entire body true with respect to D.

We have thus shown that for any database D, if C_1 yields **panic** then so does C_2; i.e., $C_1 \subseteq C_2$. Note that this argument easily generalizes to the case where the CQ's have nontrivial heads. It also generalizes to the case where C_2 is a union of CQ's.

Only If: Now suppose that $A(C_1)$ does not imply $\vee_{h \text{ in } H}\ h\big(A(C_2)\big)$. Then there must be an instantiation g for the variables of C_1 such that $g\big(A(C_1)\big)$ is true, yet for no h in H is $g\big(h(A(C_2))\big)$ true.

Let D be the database consisting of exactly those tuples that are formed by applying g to the ordinary subgoals of C_1. Surely, C_1 produces **panic** on D. Suppose that C_2 produces **panic** on D. Then there has to be some instantiation f of the variables of C_2 that makes the ordinary subgoals of C_2 become tuples of D and such that $f\big(A(C_2)\big)$ is true. Because variables of C_1 and C_2 appear only once in ordinary subgoals, it is possible to write $f = g \circ h$, where h is a containment mapping from $O(C_2)$ to $O(C_1)$.

We know that $g\big(h((A(C_2))\big)$ is false. That is, $f\big(A(C_2)\big)$ is false, contradicting the assumption that f makes $A(C_2)$ true. We conclude that C_2 does not produce **panic** on database D. Since C_1 does, we have shown that when the condition of Theorem 5.1 does not hold, C_1 is not contained in C_2. □

Incidentally, one might question whether the conditions that no variable appear twice in ordinary subgoals and that constants not appear at all are essential in Theorem 5.1. The following examples show that they are.

Example 5.2: Consider the constraints

C_1: **panic** `:- p(X,X)`
C_2: **panic** `:- p(X,Y) & X=Y`

Since the two constraints are clearly equivalent, we know $C_1 \subseteq C_2$. However, since $A(C_1)$ is empty, the condition of Theorem 5.1 can easily be shown not to hold.

Similarly, consider the pair of constraints

C_1: **panic** `:- p(0,X)`
C_2: **panic** `:- p(Z,X) & Z=0`

Here, there are no duplications of variables in the ordinary subgoals of either constraint, but there is a constant in C_1. Again, the constraints are equivalent, but $C_1 \subseteq C_2$ does not follow from the statement of Theorem 5.1. □

Comparison With Klug's Approach

The reader should compare the approach of Theorem 5.1, where we need to consider all containment mappings, to the approach of Klug [1988], where all orders consistent with the arithmetic constraints are considered. Both approaches can be exponential in the worst case, and in fact the general problem is complete for Π_2^p (van der Mayden [1992]).

Klug's approach in the worst case requires an exponential number of tests, each of which could take exponential time. Ours creates one test that could in the worst case be exponential in the size of C_2, but the test for satisfaction of the implication is exponential only in the number of variables, that is, in the size of C_1.

However, for constraint checking, it is likely that the conjunctive queries involved will have few duplicate predicates, and these predicates and their multiplicities will be the same in both queries. (The material following shortly on reduction-based tests justifies this position.) Thus, there will tend to be few containment mappings in practice.

As we saw in Example 5.1, there may be logical simplifications that facilitate the checking of the implication. Thus, we expect our approach to perform better in real cases. In Section 6, we see applications where the approach embodied in Theorem 5.1 appears to be necessary to handle some special cases of CQC's.

Instantiating Local Predicates

In order to handle CQ's with local predicates and available local data, it is useful to develop a notation for instantiating the local predicate in all possible ways. If t is a tuple that could be in the relation for predicate l, and C is a CQC of the form described in the beginning of Section 5, then $\text{RED}(t, l, C)$, the *reduction* of C by t in l, is obtained by substituting the components of t for the corresponding variables in the arguments of l, and then eliminating l.

Example 5.3: We shall now introduce an important example, which we call *forbidden intervals.*

C: **panic** `:- l(X,Y) & r(Z) & X<=Z & Z<=Y`

That is, each pair in the local relation can be thought of as the ends of an interval which no Z in the remote relation may occupy.

Suppose that the relation for l has pairs $(3, 6)$ and $(5, 10)$, and we insert into l the tuple $(4, 8)$. Then:

$\text{RED}((3,6),l,C)$ = `r(Z) & 3<=Z & Z<=6.`
$\text{RED}((5,10),l,C)$ = `r(Z) & 5<=Z & Z<=10.`
$\text{RED}((4,8),l,C)$ = `r(Z) & 4<=Z & Z<=8.`

Note that

$$\text{RED}((4,8),l,C) \subseteq \text{RED}((3,6),l,C) \cup \text{RED}((5,10),l,C)$$

This observation tells us that when the stated insertion occurs, we need not fear that C is violated, if it was not previously violated. That is, the presence of the tuples $(3,6)$ and $(5,10)$ in l together assure us that the interval from 4 to 8 is clear of elements in the remote relation for r. $\Box$

- Example 5.3 points out that when CQC's have arithmetic comparisons, there may be a containment of one CQC C in the union of several CQC's, while C is not contained in any one CQC of the union. The same cannot happen if there are no arithmetic comparisons (Sagiv and Yannakakis [1981]). The need to consider containment of a CQC in several CQC's is the reason that the results of Gupta and Ullman [1992] or Gupta and Widom [1993] cannot be extended to allow arithmetic comparisons, and still get a complete test.

The following theorem formalizes the complete local test suggested by Example 5.3 for preservation of a single constraint.

Theorem 5.2: Let C be a CQC and let t be a tuple inserted into the local relation L for predicate l. Assume C holds before the update. Then the complete local test for guaranteeing that C holds after the update is whether $\text{RED}(t,l,C) \subseteq \cup_{s \text{ in } L}\ \text{RED}(s,l,C)$.

Proof: Straightforward argument building on the results of Nicolas [1982]. $\Box$

- Theorem 5.2 extends to the case where several constraints are assumed to hold prior to the update. We then add to the union on the right the reductions of the other constraints by all tuples in L.

Constructing Complete Local Tests for CQC's

Theorems 5.1 (extended to cover containment of a CQC in a union of CQC's) and 5.2 suggest an algorithm to test whether a CQC C is preserved by a local update. If t is the inserted tuple, and L is the local relation for predicate l, let H_s, for s in L, be the set of containment mappings from $\text{RED}(s,l,C)$ to $\text{RED}(t,l,C)$. For any tuple u, let C_u be $A\big(\text{RED}(u,l,C)\big)$, that is, the arithmetic comparisons in the reduction of C by u in l. Then we need to test whether C_t logically implies $\vee_{s \text{ in } L, h \text{ in } H_s}\ h(C_s)$.

Arithmetic Free CQC's

If there are no arithmetic comparisons in a CQC, the test of Theorem 5.2 can be simplified. (Furthermore, here we can allow constants and repeated variables to appear in the local and remote predicates.) There are two key differences when arithmetic is absent:

1. Whenever $\text{RED}(t,l,C)$ is contained in the union of reductions over all tuples in L, then $\text{RED}(t,l,C)$ is contained in one such reduction.
2. The containment of $\text{RED}(t,l,C)$ in $\text{RED}(s,l,C)$ depends not on the values of the components of the tuples t and s, but only on their patterns of equality and inequality.

Thus, while the test in general could be exponential in the size of the local relation L and also in the CQC, for the arithmetic-free case the test is exponential only in the size of the CQC.

Theorem 5.3: In time at most exponential in the size of an arithmetic-free CQC it is possible to construct an expression of relational algebra whose nonemptiness is the complete local test for preservation of the CQC after an insertion to the local relation.

Proof: (Sketch) The main ideas are as follows. Suppose t is the inserted tuple, and C is an arithmetic-free CQC with local relation L for predicate l. Let μ be a tuple of variables of the length appropriate for L. Then we need only ask if there is a containment mapping from $\text{RED}(\mu,l,C)$ to $\text{RED}(t,l,C)$. If there are one or more, then each containment mapping provides a set of constraints on the variables in μ. These can easily be translated into an algebraic expression on L. $\Box$

Example 5.4: The ideas should become clearer from the following example. Consider the arithmetic-free CQC

C_1: **panic** `:- l(X,Y,Y) & r(Y,Z,X)`

Let $\mu = (A,B,C)$. Then $\text{RED}(\mu,l,C_1)$ is `r(B,Z,A)`. Note that variables B and C were equated in μ by the substitution into the subgoal $l(X,Y,Y)$.

Now, consider inserted tuple

$$t = (a,b,c)$$

$\text{RED}(t,l,C_1)$ does not exist, because $b \neq c$ and there is thus no way to unify $l(t)$ with $l(X,Y,Y)$. Thus, there is no condition under which the insertion of t could invalidate C_1; i.e., the complete local test is "true".

Next consider the inserted tuple $s = (a,b,b)$. Now, RED(s, l, C_1) is `r(b,Z,a)`. There is one containment mapping from RED(μ, l, C_1) to RED(s, l, C_1), namely $h(A) = a$; $h(Z) = Z$; $h(B) = b$. This containment mapping says that the first two components of μ must be a and b, respectively. Moreover, the unification of μ with $l(X, Y, Y)$ required that the second and third components of μ be equal; i.e., $B = C$. Expressed in relational algebra on the relation L of which μ is an abstract tuple, this expression is $\sigma_{\#1=a \wedge \#2=b \wedge \#3=b}(L)$. That is, when we insert a tuple (a, b, b) into L, the complete local test is whether this tuple already exists in L, not a very useful test, but one that technically should be made. □

6 Complete Local Tests Expressible in Recursive Datalog

The problem of finding complete local tests for CQC's including arithmetic comparisons appears quite hard in general. However, there is a common case, called "independently constrained queries," where we are able to express the complete local test as a (recursive) datalog program with arithmetic.

Independently Constrained Queries

Call a variable in a CQC *remote* if it does not appear in a local subgoal. A CQC C is *independently constrained* (an ICQ) if every comparison, except an equality comparison, involves at most one remote variable.

Example 6.1: The forbidden intervals problem of Example 5.3 is typical:

C: **panic** `:- l(X,Y) & r(Z) & X<=Z & Z<=Y`

Only Z is a remote variable, and both comparisons involve Z and some variable that is not remote. Hence, C is an ICQ. In fact, every CQC with at most one remote variable is an ICQ.

Informally C is violated whenever there is a remote value Z that lies in some interval $[X, Y]$ defined by a pair (X, Y) in the relation L for l. If we insert a tuple (a, b) into L we can be sure C continues to hold if and only if the union of all the intervals (X, Y) in L covers $[a, b]$. Thus, inclusion of interval $[a, b]$ in the union of all the intervals of L is a complete local test.

- Unlike Theorem 5.3, this constraint C does not have a complete local test that is an expression of relational algebra. If such an expression existed, there would be a bound k, derivable from the expression, such that at most k different tuples of L are "looked at" in determining the presence of any one tuple in the result. However, we can then concoct an example where it takes $k+1$ tuples to "cover" the inserted tuple, proving that the hypothetical expression cannot be correct.

```
(1) interval(X,Y) :- l(X,Y)
(2) interval(X,Y) :- interval(X,W) &
                     interval(Z,Y) &
                     Z <= W
(3) ok(A,B) :- interval(X,Y) &
               X <= A & B <= Y
```

Fig. 6.1. Complete local test for forbidden intervals.

The recursive datalog program of Fig. 6.1 constructs maximal intervals from the tuples in L. That is, we combine overlapping intervals into one interval that includes them both, until we have the longest possible intervals constructed from the given intervals. Finally, given an inserted tuple (a, b), we need only determine whether $ok(a, b)$ is true. This condition is the complete local test. □

Theorem 6.1: For any ICQ we can construct a (recursive) datalog program with arithmetic to serve as a complete local test.

Proof: (Sketch) The proof generalizes Example 6.1. For any remote variable in an ICQ we can define intervals that the local data tells us are forbidden to that variable. The only major differences between Example 6.1 and the general case is that in general these intervals may be open to infinity or minus infinity, and they may be open or closed at either end. Thus, there may be as many as eight different predicates corresponding to *interval* in Fig. 6.1.

Begin by getting rid of $=$ and $\neq$ comparisons. We can remove $=$'s by equating variables and/or constants. We get rid of $X \neq Y$ by splitting the ICQ into two ICQ's, one with $X < Y$ and the other with $X > Y$. The resulting datalog programs can be defined by creating a new IDB predicate that represents the union of the goal predicates for the two programs.

Our datalog program begins with basis rules that initialize the forbidden intervals; rule (1) of Fig. 6.1 is an example. The intervals are constructed by imagining all nonremote variables to be constants. Define the maximum of the lower bounds on Z to be the low end of the interval ($-\infty$ if none) and the minimum of the upper bounds to be the high end of the interval (∞ if none). Since many different variables may be the lower or upper bound, depending on how the variables are ordered, we may need a different rule for every such order; each rule requires subgoals that check the presumed order is satisfied.

Then, we use recursive rules to group intervals together. Rule (2) of Fig. 6.1 gives the idea, but we must also consider the possibility of open or closed intervals and intervals open to $\pm\infty$. Finally, the complete local test is defined as in rule (3) of Fig. 6.1, again modified for the possibility of open intervals and infinite intervals. □

For a thorough treatment of constraint checking with partial information, including additional examples, results, and full proofs, refer to Gupta [1994].

Bibliography

Blakeley, J. A., N. Coburn, and P.-A. Larson [1989]. "Updating derived relations: detecting irrelevant and autonomously computable updates," *ACM Trans. on Database Systems* **14**:3, pp. 369–400.

Ceri, S. and J. Widom [1990]. "Deriving production rules for constraint maintainence," *Proc. International Conference on Very Large Data Bases*, pp. 566–577.

Ceri, S. and J. Widom [1991]. "Deriving production rules for incremental view maintainance," *Proc. International Conference on Very Large Data Bases*, pp. 577–589.

Chandra, A. K. and H. R. Lewis and J. A. Makowsky [1981]. "Embedded implicational dependencies and their inference problem," *Proc. Thirteenth Annual ACM Symposium on the Theory of Computing*, pp. 342–354.

Chandra, A. K. and P. M. Merlin [1977]. "Optimal implementation of conjunctive queries in relational databases," *Proc. Ninth Annual ACM Symposium on the Theory of Computing*, pp. 77–90.

Chaudhuri, S. and M. Y. Vardi [1992]. "On the equivalence of datalog programs," *Proc. Eleventh ACM Symposium on Principles of Database Systems*, pp. 55–66.

Courcelle, B. [1991]. "Recursive queries and context-free graph grammars," *Theor. CS* **78**, pp. 217–244.

Elkan, C. [1990]. "Independence of logic database queries and updates," *Proc. Ninth ACM Symposium on Principles of Database Systems*, pp. 154–160.

Gupta, A. [1994]. "Efficient Maintenance of Integrity Constraints and Views in Database Systems," Ph.D. thesis, Dept. of CS, Stanford University.

Gupta, A. and J. D. Ullman [1992]. "Generalizing conjunctive query containment for view maintenance and integrity constraint verification," *Joint Intl. Conf. on Logic Programming, Workshop on Deductive Databases.*

Gupta, A. and J. Widom [1993]. "Local verification of global integrity constraints in distributed databases," *ACM SIGMOD International Conf. on Management of Data*, pp. 49–58.

Klug, A. [1988]. "On conjunctive queries containing inequalities," *J. ACM* **35**:1, pp. 146–160.

Levy, A. Y. and Y. Sagiv [1993]. "Queries independent of update," *Proc. International Conference on Very Large Data Bases*, pp. 171–181.

Nicolas, J.-M. [1982]. "Logic for improving integrity checking in relational databases," *Acta Informatica* **18**:3, pp. 227–253.

Sagiv, Y. [1988]. "Optimizing datalog programs," in *Foundations of Deductive Databases and Logic Programming* (J. Minker, ed.), Morgan-Kaufmann, San Mateo.

Sagiv, Y. and M. Yannakakis [1981]. "Equivalence among relational expressions with the union and difference operators," *J. ACM* **27**:4, pp. 633–655.

Shmueli, O. [1987]. "Decidability and expressiveness aspects of logic queries," *Proc. Sixth ACM Symposium on Principles of Database Systems*, pp. 237–249.

Tompa, F. W. and J. A. Blakeley [1988]. "Maintaining materialized views without accessing base data," *Inform. Systems* **13**:4, pp. 393–406.

Ullman, J. D. [1989]. *Principles of Database and Knowledge-Base Systems, Vol. II: The New Technologies*, Computer Science Press, New York.

van der Meyden, R. [1992]. "The complexity of querying indefinite data about linearly ordered domains," *Proc. Eleventh ACM Symposium on Principles of Database Systems*, pp. 331–345.