In: *Proc. of the Fifth Int'l Workshop on the Deductive Approach to Information Systems and Databases*, Costa Brava, Spain, Sep. 1994

Validating Constraints with Partial Information: Research Overview*

Jennifer Widom (invited speaker)
Ashish Gupta
Yehoshua Sagiv†
Jeffrey D. Ullman

Department of Computer Science
Stanford University
Stanford, CA 94305-2140
widom@cs.stanford.edu

Abstract

We are interested in the problem of validating the consistency of integrity constraints when data is modified. In particular, we consider how constraints can be checked with only "partial information". Partial information may include: (1) the constraint specifications only, (2) the constraint specifications and the modified data, or (3) the constraint specifications, the modified data, and portions of the existing data. Methods for constraint checking with partial information can be much more efficient than traditional constraint checking methods (e.g. because work is done at compile time, or because less data is accessed). Partial information methods also enable constraint checking in scenarios where traditional constraint checking methods fail (e.g. in distributed environments where not all data is accessible). We explain how existing methods and results for query containment and for independence can be applied to problems (1) and (2) above, and we give an overview of our research into problem (3).

1 Introduction

The description of a database usually includes a *schema*, which describes how data in the database is structured, and a set of *integrity constraints*, which describe those states of the data that are considered semantically valid. Normally, an integrity constraint is specified either as a logical formula or as a query. In the case of a logical formula C_F, the database *violates* the constraint if C_F evaluates to false, while the database is *valid* with respect to the constraint (or the database *satisfies* the constraint) if C_F evaluates to true. In the case of a query C_Q, the database violates the constraint if C_Q produces a non-empty result, while the database is valid with respect to the constraint if C_Q produces an empty result. For first-order logic and first-order query languages, these formulations usually are equivalent.

*Research supported by NSF grants IRI-91-16646 and IRI-92-23405, by ARO grant DAAL03-91-G-0177, by ARPA contract F33615-93-1-1339, by the Reid and Polly Anderson Faculty Scholar Fund, and by equipment grants from Digital Equipment Corporation and IBM Corporation.

†Permanent address: Department of Computer Science, Hebrew University, Jerusalem, Israel.

Consider a database and an integrity constraint C (specified as a formula or a query), and suppose the database is modified. The brute-force method of checking constraint C for a modification is to first perform the modification, then evaluate the constraint formula or query on the new database. Obviously this can be quite inefficient, and it requires access to the constraint specification, the modification, and all of the existing data relevant to the constraint. We are exploring methods for constraint checking that require only some of this information. Our methods can be much more efficient than the brute-force method, and since only partial information is required, our methods can be used in scenarios where the brute-force method cannot be used. Note that all of our methods hinge on the following simple but useful assumption:

> **Initial Consistency Assumption**: Before the database is modified, all integrity constraints are valid.

1.1 Partial Information

We consider three classes of problems, delineated by how much information is used to validate a constraint. For each problem, the goal is to obtain a test that uses only the partial information to determine, after the database is modified, whether the constraint is still valid. Here are the three classes of problems we address:

1. The partial information includes only the specifications of all integrity constraints. This problem corresponds to implication of constraint formulas or queries, since the only way we can validate a constraint C without looking at the modification or any of the data is if there are other constraints $C_1, \ldots, C_n$, known to be valid, such that whenever $C_1, \ldots, C_n$ are valid then so is C.

2. The partial information includes only the constraint specifications and the data modification. This problem corresponds to the *query independent of update* problem, which has been studied in [BT88, Elk90, LS93].

3. The partial information includes the constraint specifications, the data modification, and a portion of the data relevant to the constraint. This is the problem we study in depth. It arises most naturally in distributed environments where constraints span multiple databases—in such environments it is preferable to avoid accessing remote data to validate a constraint [GT93, GW93]. Note, however, that the problem is applicable in any environment where it may be expensive or impossible to access portions of the data.

1.2 Properties of Tests

For the problems outlined above, the goal is to obtain a constraint *test*—a method that uses the partial information to validate a constraint. Note that unless all information is used (the constraint, the modified data, and all of the existing data relevant to the constraint) the test will be conservative: If the test succeeds, then it is guaranteed that the database satisfies the constraint. However, if the test fails it may be necessary to make a further test using more information to determine whether the constraint is valid.

Clearly we want our tests to be *correct*, i.e. if the test succeeds then the constraint is valid. It also is desirable for tests to be *complete*, but completeness here must be with respect to the partial information used. A test is complete in this sense if, whenever it determines that a constraint may be violated (i.e. the test fails), then there is some configuration of the unused information for which the constraint indeed is violated.

Another desirable property of tests is that they be *query-based*. This property is applicable only when the partial information includes portions of the database, i.e. the third class of problems in Section 1.1. A test is query-based if it can be performed by executing one or more queries over the available data, where the queries are expressed in a language such as SQL or Datalog. The obvious advantage of query-based tests is that they can rely on an existing query processor. The query processor can automatically find efficient algorithms for execution, and it can exploit available access structures such as indices. A test method that isn't query-based might execute an algorithm directly on the data, or it might create a (very large) Datalog program with one ground fact for each of the known tuples.

1.3 Outline of Paper

In Section 2 we introduce our constraint specification language—constraints are specified as logical queries in a Datalog-like notation. Using this language, we consider the three classes of constraint validation problems outlined in Section 1.1. In Section 3 we consider validating constraints using only the constraint specifications. In Section 4 we consider validating constraints using the constraint specifications and the modified data. In Section 5 we consider validating constraints using the constraint specifications, the modified data, and a portion of the existing data. We do not give complete research results in each of these sections, but rather we provide examples and point out the relevant issues; detailed results appear in [GSUW94, Gup94, GW93]. In Section 6 we conclude and discuss future directions of this research.

2 Constraint Queries

We consider relational databases where relations are modeled as predicates and queries are expressed as logical rules that derive a *result* predicate, as in, e.g., *Datalog* [Ull89]. A *constraint* is expressed as a query whose result is a special 0-ary predicate that we call `panic`. If the query produces $\emptyset$ on a given database, then the constraint is valid for that database. If the query produces `panic` then the constraint is violated. The difficulty of validating constraints naturally depends on the language that we use to express constraint queries. Examples of interesting languages for expressing constraint queries are:

- conjunctive queries [CM77]
- nonrecursive Datalog, or (equivalently) unions of conjunctive queries [SY80]
- conjunctive queries with arithmetic comparisons [Klu88]
- nonrecursive Datalog with negation [Ull89]
- recursive Datalog, possibly with arithmetic comparisons and/or negation [Ull89]

Several examples are now given. We use a generic corporate relational database whose schema should be self-explanatory. The following query specifies a constraint that no employee can be in both the "sales" and the "accounting" departments. Note that this constraint is expressed using a conjunctive query.

$\texttt{panic} \text{ :- } \texttt{emp}(E, \text{"sales"}, S)\ \&\ \texttt{emp}(E, \text{"accounting"}, S')$

The following constraint is violated whenever an employee with salary less than 100 is not in a department. This query uses an arithmetic comparison and negation.

$\texttt{panic} \text{ :- } \texttt{emp}(E, D, S)\ \&\ \neg\texttt{dept}(D)\ \&\ S < 100$

The following constraint uses an intensional relation (`boss`) and recursion; it ensures that no employee is his or her own boss.

$\texttt{panic} \text{ :- } \texttt{boss}(E, E)$
$\texttt{boss}(M, E) \text{ :- } \texttt{emp}(E, D, S)\ \&\ \texttt{manager}(D, M)$
$\texttt{boss}(B, E) \text{ :- } \texttt{boss}(B, B')\ \&\ \texttt{boss}(B', E)$

The following two constraints together specify that every employee's salary is within a range specified for his or her department.

C_1: $\texttt{panic} \text{ :- } \texttt{emp}(E, D, S)\ \&\ \texttt{sal-range}(D, \mathit{Low}, \mathit{High})\ \&\ S < \mathit{Low}$
C_2: $\texttt{panic} \text{ :- } \texttt{emp}(E, D, S)\ \&\ \texttt{sal-range}(D, \mathit{Low}, \mathit{High})\ \&\ S > \mathit{High}$

We assume that every constraint query includes exactly one rule with `panic` in the head. Any constraint with multiple `panic` rules can equivalently be expressed as multiple constraints, as illustrated in the salary range example above.

3 Using Constraints Only

The first problem we consider is constraint validation when the only information used is the constraint specifications themselves, not even the modified data. In the case of a database with a single constraint, this problem is inapplicable: we cannot check if a modification violates a constraint without at least examining the modification. However, in a database with multiple constraints, the validity of one constraint C_1 may imply the validity of another constraint C_2 (equivalently, if C_2 is violated then so is C_1). In this case, C_2 need not be checked when it is known that C_1 is valid. Since we are assuming an environment in which all constraints must be valid after all modifications, if C_1 implies (or *subsumes*) C_2, then C_2 can be ignored altogether. In the more general case, suppose we have a database with a set of constraints $\{C_1, \ldots, C_n\}$. An additional constraint C is subsumed by $\{C_1, \ldots, C_n\}$ if, whenever C is violated then so is at least one $C_i \in \{C_1, \ldots, C_n\}$. Since subsumption of C by $\{C_1, \ldots, C_n\}$ is independent of data or data modifications, the test can be made once, at constraint-definition time.

The constraint subsumption problem corresponds directly to *query containment* [Ull89]. Recall that a query Q_1 is contained in a query Q_2, denoted $Q_1 \subseteq Q_2$, iff, for any database D, the result of Q_1 in D is contained in the result of Q_2 in D. Hence, for the two-constraint case described above, if $C_2 \subseteq C_1$ then we need never validate C_2 given the validity of C_1. In the more general case, if $C \subseteq C_1 \cup \cdots \cup C_n$ then we need never validate C given the validity of $C_1, \ldots, C_n$. Note that since constraint queries produce only $\emptyset$ or `panic`, constraint subsumption is actually a special case of query containment, however for some classes of constraint queries the problems can be shown as equivalent; for details see [GSUW94].

Our constraint subsumption test is complete as defined in Section 1.2 if the containment test that we use determines $C \subseteq C_1 \cup \cdots \cup C_n$ whenever the containment holds (i.e. the containment test is not conservative). To see this, suppose the containment test accurately determines $C \subseteq C_1 \cup \cdots \cup C_n$, and it decides that $C \subseteq C_1 \cup \cdots \cup C_n$ does not hold. Then there is some database for which $C_1 \cup \cdots \cup C_n = \emptyset$ but $C =$ `panic`. Consequently, there is some state of the unused information for which the constraint is violated, and hence the test is complete as defined in Section 1.2. Query containment algorithms generally give us a complete method for checking constraint subsumption [Ull89].

4 Using Constraints and Modified Data

The second problem we consider is constraint validation when the information used is the constraint specifications and the modifications to the data. This problem is useful for checking a constraint without incurring the time or cost associated with accessing the database, or when the database is inaccessible. Methods for constraint checking using the modified data are applied at run time (as opposed to the compile time methods described in the previous section), and they serve as an optimization: either we determine that the modification cannot violate a constraint, or we determine that the modification may violate the constraint but we need to examine the database to be sure (recall the discussion in Section 1.2).

We begin by considering only one constraint C; we generalize to a set of constraints below. Let D be an arbitrary database, let $C(D)$ denote the result of evaluating constraint query C on database D, so either $C(D) = \emptyset$ or $C(D) =$ `panic`. Now suppose we perform a modification m that takes the database from D to $D(m)$. We assume that C is valid before the modification, i.e.

$C(D) = \emptyset$, and we want to verify that C is valid after the modification, i.e. $C(D(m)) = \emptyset$. Hence, we want to verify that $C(D) = \emptyset$ implies $C(D(m)) = \emptyset$, and we want to do so by examining only the modification m, and not the databases D or $D(m)$. This problem is a special case of the problem of testing whether a query is independent of a modification. Given a query Q and a modification m, Q and m are *independent* if there is no database such that the result of Q is different before and after the modification. The independence problem has been studied in, e.g., [Elk90, LS93].

Following the methods in [LS93], we use constraint C and modification m to construct a constraint query C' such that C' is valid before the modification (i.e. $C'(D) = \emptyset$) iff C is valid after the modification (i.e. $C(D(m)) = \emptyset$). Once such a constraint C' is constructed, it is sufficient to test the containment $C' \subseteq C$. To see that the containment test is sufficient, assume $C' \subseteq C$ and $C(D) = \emptyset$. Then, by $C' \subseteq C$ we know $C'(D) = \emptyset$, and consequently $C(D(m)) = \emptyset$. Suppose now that we have a set of constraints $\{C_1, \ldots C_n\}$ and we want to validate one of the constraints C in the set. The same procedure as the two-constraint case is followed, except the containment test becomes $C' \subseteq C_1 \cup \cdots \cup C_n$.

We do not describe the construction of constraint query C' here. C' effectively incorporates modification m into constraint query C, so that evaluating C' on D is equivalent to evaluating C on $D(m)$. Details of the construction can be found in [LS93]; discussion and examples of its application to constraints can be found in [LS93, GSUW94, Gup94]. One interesting point to note is that constraint query C' may require a language that is more expressive than the language used for constraints $\{C_1, \ldots C_n\}$. (Recall the discussion of constraint query languages in Section 2.)

Similar to Section 3, our constraint test here is complete as defined in Section 1.2 if the containment test it relies on determines $C' \subseteq C_1 \cup \cdots \cup C_n$ whenever the containment holds. The argument is analogous to the one given in Section 3.

5 Using Constraints, Modified Data, and Partial Existing Data

The problem in which we are most interested is validating constraints when we use the constraint specifications, the modifications, and a portion of the data relevant to the constraint. This problem was motivated initially by a distributed collaborative design scenario [GT93], but it is applicable in any scenario where some data relevant to a constraint is more difficult or expensive to access than other data. We refer to the portion of the data used to check a constraint as the *accessible* data, and we refer to the portion of the data involved in a constraint but not used to check the constraint as the *inaccessible* data. As in the previous section, our methods for this scenario serve as an optimization: either we determine that the modification cannot violate a constraint, or we determine that the modification may violate the constraint but we need to examine more of the relevant data to be sure.

We focus our discussion here on constraints expressed as conjunctive queries with arithmetic comparisons, we suppose the only accessible relation is the modified relation, and we consider modifications that are insertions of a single tuple. This simplified scenario serves well to illustrate our methods and results; generalizations can be found in [GSUW94, Gup94, GW93].

The form of the constraint queries we consider is:

$$C\text{: } \texttt{panic} \text{ :- } l \ \& \ r_1 \ \& \ \cdots \ \& \ r_n \ \& \ c_1 \ \& \ \cdots \ \& \ c_k$$

Here, l is the accessible (or "local") relation, $r_1, \ldots, r_n$ are the inaccessible (or "remote") relations, and each c_i is an arithmetic comparison involving one of $<$, $\leq$, $>$, $\geq$, or $=$. We also require:

- Variables in the c_i's must also appear in l or in one of the r_i's.
- No variable may appear twice among l and the r_i's. (This is not a restriction over Datalog—multiple variable occurrences can be handled by using distinct variables and equating them in a c_i.)
- There are no constants in l or in the r_i's. (Again, this is not a restriction—variables are used and then equated to constants in the c_i's.)

Let tuple t be inserted into relation l and assume constraint C holds before the insertion. We want to use C, t, and l to infer that C is still valid after the insertion. Using C and t, we derive a test condition on relation l such that if the test is satisfied then the constraint is valid. We have developed a number of methods for deriving tests, reported in [GSUW94, Gup94, GW93]. Rather than giving all details and proofs of the methods here, we instead provide a number of examples that serve to illustrate the tests and the relevant issues.

Example 5.1 Consider again our corporate database, where a tuple $\texttt{emp}(e, d, s)$ says that employee number e in department d has salary s, and a tuple $\texttt{dept}(d, ms)$ says that some manager in department d has salary ms. Let the constraint assert that every employee earns less than every manager in the same department. This constraint is expressed as:

$$C : \texttt{panic} \text{ :- } \texttt{emp}(E,\ D,\ S) \ \& \ \texttt{dept}(D,\ MS) \ \& \ S \geq MS$$

Let relation `emp` be accessible and relation `dept` be inaccessible. Suppose tuple $\texttt{emp}(e, d, 50)$ is inserted. Constraint C will be violated if department d has a manager whose salary is less than or equal to 50. However, suppose department d already has an employee whose salary is 100. Since constraint C is not violated before the insertion, we can infer from relation `emp` alone that no manager in d earns less than or equal to 100. Therefore, inserted tuple $\texttt{emp}(e, d, 50)$ cannot violate constraint C.

The formal test condition that corresponds to this informal reasoning is the following Datalog query. The query derives `insertion-ok` if the test succeeds in determining that the constraint is not violated.

$$\texttt{insertion-ok} \text{ :- } \texttt{inserted}(E,\ D,\ S) \ \& \ \texttt{emp}(E',\ D,\ S') \ \& \ S' \geq S$$

Relation `inserted` contains only the inserted tuple, and relation `emp` does not contain the inserted tuple. This test is complete with respect to the accessible data, as defined in Section 1.2. To see this, suppose the test fails. Then there is no existing employee with a salary higher than S. Consequently, there may be a manager in department D with salary less than or equal to S, violating the constraint. □

Note that the test condition in Example 5.1 was derived without considering the actual value of the inserted tuple. Hence, the test can be derived at compile time, then instantiated at run time with the inserted tuple. Also, because the test is a simple Datalog query (equivalent to an SQL *select-project-join* query), it can be executed by almost any relational query processor. That is, the test is query-based as described in Section 1.2.

We now give two examples that illustrate the complexity that can arise in the test condition.

Example 5.2 Consider the following constraint query:

C: `panic` :- `interval`(X, Y) & `point`(Z) & $X \leq Z$ & $Z \leq Y$

The constraint states that each pair X, Y in relation `interval` defines the ends of an interval that no Z in relation `point` may occupy. Suppose that relation `interval` is accessible and relation `point` is inaccessible. Informally, using reasoning similar to Example 5.1, we come up with the following test condition:

`insertion-ok` :- `inserted`(X, Y) & `interval`(X', Y') & $X \geq X'$ & $Y \leq Y'$

This test condition says that an inserted interval will not violate the constraint if there already exists an interval completely containing the inserted interval.

Unfortunately, this simple and easy to evaluate test condition, although correct, is not complete. Suppose relation `interval` contains the tuples $(3, 6)$ and $(5, 10)$, and suppose tuple $(4, 8)$ is inserted. The test above fails since interval $(4, 8)$ is not contained in either interval $(3, 6)$ or $(5, 10)$. However, since the constraint tells us that no points occur in either interval $(3, 6)$ or $(5, 10)$, we should be able to infer that no points occur in the inserted interval $(4, 8)$.

The complete test for this example is the following recursive Datalog program, which uses an intensional relation `merge-int` to store all of the intervals obtained by merging overlapping intervals.

`insertion-ok` :- `inserted`(X, Y) & `merge-int`(X', Y') & $X \geq X'$ & $Y \leq Y'$
`merge-int`(X, Y) :- `interval`(X, Y)
`merge-int`(X, Y) :- `merge-int`(X, Z) & `merge-int`(Z', Y) & $Z \geq Z'$

Since recursion is required, the test may not be query-based, depending on the capabilities of the underlying database system. □

Example 5.3 Now consider the following constraint, similar to Example 5.2 but in two dimensions:

C: `panic :- rectangle`(U, V, W, Z) & `point`(X, Y) & $U \leq X$ & $X \leq V$ & $W \leq Y$ & $Y \leq Z$

A tuple in relation `point` defines a point in two dimensional space, and a tuple in relation `rectangle` defines a rectangular region in the same space. The constraint requires that all points defined by relation `point` lie outside every rectangular region defined by relation `rectangle`. Relation `rectangle` is accessible but relation `point` is inaccessible. An inserted tuple `rectangle`(a, b, c, d) does not violate C if the rectangle defined by the inserted tuple is contained in the union of the rectangles defined by the existing tuples in `rectangle`. The test for determining when a rectangle is contained in a set of other rectangles can still be represented as a recursive Datalog program, but building the program is not as straightforward as in Example 5.2. □

6 Conclusions and Future Work

Our methods for validating constraints with partial information promise to increase the efficiency of checking constraints when a database is modified; our methods also promise to make constraint checking feasible in environments where not all of the relevant information is readily accessible. We have applied our approach in the context of a Civil Engineering collaborative design project and have discovered that many common constraints arising in this domain can take advantage of our methods [GT93].

Meanwhile, there are a number of extensions and improvements yet to be made. For example:

- Consider more expressive languages for constraint queries. So far, we have been able to obtain complete tests for the general problem addressed in Section 5 (constraint validation with the constraint, the modified data, and a portion of the existing data) only when the constraint is expressed as a conjunctive query with inequalities. We have considered more expressive constraint languages, e.g. in [Gup94, GW93], but the tests we obtain are not always complete. We also are interested in going beyond logical constraint queries to include, e.g., aggregation.

- Use different amounts and type of information. Although in Sections 3 and 4 we considered sets of constraints, in Section 5 we used only a single constraint. We might use multiple constraints for the Section 5 scenario. We also might incorporate "built in" constraints into all of our methods, such as functional dependencies.

- Consider different amounts of accessible data. We have made the simplifying assumption that only one relation is accessible, and that the accessible relation is also the modified relation. We might consider multiple available relations and the case in which the accessible relations do not include the modified relation.

- Handle data modifications that are more general than a single insertion, e.g. deletions, updates, and sets of modifications.

Some of these extensions have been addressed in [Gup94], while others are still open. In addition to these extensions, we are interested in applying our results to related but somewhat different problems. Two possibilities are:

- *Active databases* [DHW94], where if we view an active rule's condition part as a constraint, then we might apply our methods to determine when a rule condition is satisfied without accessing all of the relevant information.

- *View maintenance* [CW91, GMS93], where our methods might be used to determine, using only a portion of the relevant information, whether a view needs to be updated when base relations are modified.

References

[BT88] J.A. Blakeley and F.W. Tompa. Maintaining materialized views without accessing base data. *Information Systems*, 13(4):393–406, 1988.

[CM77] A.K. Chandra and P.M. Merlin. Optimal implementation of conjunctive queries in relational databases. In *Proceedings of the Ninth ACM Symposium on Theory of Computing*, pages 77–90, 1977.

[CW91] S. Ceri and J. Widom. Deriving production rules for incremental view maintenance. In *Proceedings of the Seventeenth International Conference on Very Large Data Bases*, pages 577–589, Barcelona, Spain, September 1991.

[DHW94] U. Dayal, E.N. Hanson, and J. Widom. Active database systems. In W. Kim, editor, *Modern Database Systems: The Object Model, Interoperability, and Beyond.* ACM Press, New York, 1994.

[Elk90] C. Elkan. Independence of logic database queries and updates. In *Proceedings of the Ninth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 154–160, March 1990.

[GMS93] A. Gupta, I.S. Mumick, and V.S. Subrahmanian. Maintaining views incrementally. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, Washington, D.C., May 1993.

[GSUW94] A. Gupta, Y. Sagiv, J.D. Ullman, and J. Widom. Constraint checking with partial information. In *Proceedings of the Thirteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 45–55, May 1994.

[GT93] A. Gupta and S. Tiwari. Distributed constraint management for collaborative engineering databases. In *Proceedings of the Second International Conference on Information and Knowledge Management*, Washington, DC, November 1993.

[Gup94] A. Gupta. *Efficient Maintenance of Integrity Constraints and Views in Database Systems.* PhD thesis, Stanford University, September 1994.

[GW93] A. Gupta and J. Widom. Local verification of global integrity constraints in distributed databases. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 49–58, Washington, D.C., May 1993.

[Klu88] A. Klug. On conjunctive queries containing inequalities. *Journal of the ACM*, 35(1):146–160, 1988.

[LS93] A. Levy and Y. Sagiv. Queries independent of updates. In *Proceedings of the Nineteenth International Conference on Very Large Data Bases*, pages 171–181, Dublin, Ireland, August 1993.

[SY80] Y. Sagiv and M. Yannakakis. Equivalences among relational expressions with the union and difference operators. *Journal of the ACM*, 27(4):633–665, 1980.

[Ull89] J.D. Ullman. *Principles of Database and Knowledge-Base Systems, Volumes I and II.* Computer Science Press, Rockville, Maryland, 1989.