Efficient and Complete Tests for Database Integrity Constraint Checking*

Ashish Gupta
Yehoshua Sagiv†
Jeffrey D. Ullman
Jennifer Widom

Department of Computer Science
Stanford University
Stanford, CA 94305-2140
{agupta,sagiv,ullman,widom}@cs.stanford.edu

1 Introduction

An important feature of modern database management systems is the automatic checking of *integrity constraints.* An integrity constraint is a predicate or query such that if the predicate holds on a state of the data, or equivalently if the query produces an empty answer, then the database is considered *valid.* When an integrity constraint is *violated*, i.e., when the predicate does not hold or the query produces a non-empty answer, then the update creating the undesirable database state must be rejected or some other compensating action must be taken.

We are interested in efficient methods for checking integrity constraints (hereafter called constraints) as a database is updated. Here, general efficiency is measured both in the amount of data that needs to be accessed in order to check a constraint, and in whether the check can be performed by submitting a query to the database system (rather than running an algorithm directly on the data). In terms of complexity, we are not interested in methods that are exponential in the size of the data or in the number of constraints, but we are willing to accept methods that are exponential in the size of the constraints themselves since, in databases, constraints tend to be short.

Suppose that we have a constraint C, and a database update occurs. We need to ensure that C still holds after the update. Assume that we have available at least the update itself and the definition of C. In addition to this information, there are three levels in the amount of data we might use to check the constraint: *none*, *some*, or *all.* Using none of the data corresponds to the *query independent of update* problem, which has been studied in its generality in [BT88, Elk90, LS93] and with respect to constraints by us in [GSUW93]. Using all of the data amounts to efficient evaluation of predicates or queries over the database [BC79, GMS93, HD92, Nic82, QW91, UO92]. We study the case where *some* of the data is used to check the constraint. This scenario arises whenever certain data involved in the constraint is very expensive or impossible to access, such as in distributed database systems or collaborative design [TH93, GT93]. Hereafter we refer to the portion of the data used to check a constraint as *accessible* data, and we refer to the portion of the data involved in a constraint but not used to check the constraint as *inaccessible* data.

*Research sponsored by NSF grants IRI–91–16646 and IRI–92–23405, by ARO grant DAAL03–91–G–0177, by ARPA contract F33615–93–1–1339, and by a grant of Mitsubishi Electric Corp.

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

Note that, unless all of the relevant data is accessible, our constraint checking methods will be conservative. That is, by looking at only some of the data, we may be able to determine that the constraint still holds, or we may determine that it is necessary to look at all of the data to check the constraint. A check is *correct* if, whenever it determines that a constraint still holds, then indeed the constraint holds. We also want our checks to be *complete*, but completeness is with respect to the accessible data. A check is complete in this sense if, whenever we determine that a constraint may not hold, there is some configuration of the inaccessible data for which the constraint indeed does not hold.

In the remainder of this short paper we outline the languages we have been considering for database constraints and we solidify the notion of using *some* of the data to check a constraint. We then give several examples that illustrate when and how our constraint checking methods apply. (Due to space limitations, complete technical results are not included.) The examples serve to bring out a number of problems we have not yet solved, which are enumerated at the end of the paper.

2 Problem Definition

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* [Ull88]. Examples are given below. 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 D, then the constraint holds for D. If the query produces **panic** then the constraint is violated. The difficulty of checking constraints depends on the language that we use to express constraint queries. Examples of interesting languages for expressing constraint queries are:

1. Conjunctive queries [CM77].
2. Nonrecursive Datalog, or unions of conjunctive queries [SY80].
3. Conjunctive queries with arithmetic comparisons [Klu88].
4. Datalog with negation [Ull88].
5. Recursive Datalog, possibly with arithmetic comparisons and/or negation and/or arithmetic operators [Ull88].

For some combinations of a language above and an amount of information used (*none*, *some*, or *all*), the constraint checking problem can be reduced to other problems that have been studied in the literature; see [GSUW93] for a discussion. For instance, conjunctive query containment results can be used to check constraints when only updates and constraint definitions are used [LS93].

In this paper we focus our discussion on constraints expressed as conjunctive queries with arithmetic comparisons, we suppose the only accessible relation is the updated relation, and we consider updates that are insertions of a single tuple. The general form of a conjunctive query constraint is:

$$\textbf{panic} :\!- \; l \;\&\; r_1 \;\&\; \ldots \;\&\; r_n \;\&\; c_1 \;\&\; \ldots \;\&\; c_k.$$

Here, l is the predicate for which the corresponding relation L is accessible, the relation R_i for each of the r_i's is inaccessible, and each c_i is an arithmetic comparison involving one of $<, \leq, >, \geq, =$.[1]

[1]The use of L and R refers to the fact that, in distributed databases, the "Local" data is accessible and the "Remote" data is inaccessible.

Let tuple t be inserted into relation L and assume constraint C holds before the insertion. We want to use L, C, and t to infer that C is not violated after the insertion. We derive a condition that relation L needs to satisfy in order for t not to violate C. We refer to this condition as the *test* condition. If the test condition is satisfiable, then relations $R_1, \ldots, R_n$ do not need to be accessed. The test condition is obtained by reducing the problem outlined above to the problem of checking if a conjunctive query is contained in a union of conjunctive queries; details are in [GSUW93].

3 Examples

EXAMPLE 1. Consider an employee-department relational database with two relations:

$\mathtt{emp}(E,\ D,\ S)$ % *employee number E in department D has salary S*
$\mathtt{dept}(D,\ MS)$ % *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 a conjunctive query C such that if C produces **panic** then the constraint is violated:

C: **panic** :– $\mathtt{emp}(E, D, S)$ & $\mathtt{dept}(D, MS)$ & $S \geq MS$.

Let relation `EMP` for predicate `emp` be accessible and relation `DEPT` be inaccessible. Suppose tuple $\mathtt{emp}(e, d1, 50)$ is inserted into relation `EMP`. Constraint C will be violated if department $d1$ has a manager whose salary is ≤ 50. However, suppose department $d1$ already has an employee whose salary is 100. Since constraint C is not violated before the insertion, we can infer that no manager in $d1$ earns as little as 100, and therefore $\mathtt{emp}(e, d1, 50)$ does not violate constraint C.

The above inference procedure can be formalized by specifying a test condition on the relation `EMP` and the inserted tuple, such that if `EMP` satisfies the test condition, then the inserted tuple does not violate the constraint. For constraint C, the test condition is the following Datalog query that derives `insertion_ok` if and only if the inserted tuple does not violate C, independent of the value of relation `DEPT`.

`insertion_ok` :– $\mathtt{inserted}(E, D, S)$ & $\mathtt{emp}(X, D, Y)$ & $Y \geq S$.

Relation `INSERTED` contains only the inserted tuple and `EMP` does not contain the inserted tuple. This test is complete with respect to the accessible data, as defined in Section 1. □

Note, the test condition in Example 1 is a single Datalog rule and was derived without considering the actual value of the inserted tuple. We now give two examples that illustrate the complexity that simple arithmetic comparison operators $<, >, \leq, \geq$ introduce. Example 2 shows that the complete test could be a recursive Datalog program. The constraint in Example 3 also has a complete test in the form of a recursive Datalog program, but illustrates the computational complexity of evaluating the test.

EXAMPLE 2. We shall refer to this example as *forbidden intervals.*

C: **panic** :– $\mathtt{l}(X, Y)$ & $\mathtt{r}(Z)$ & $X \leq Z \leq Y$.

Each pair in the accessible relation `L` can be thought of as the ends of an interval that no Z in the inaccessible relation `R` may occupy.

Suppose relation `L` has the tuples $(3, 6)$ and $(5, 10)$. The tuples of relation `R` that violate the constraint given tuple $\mathtt{l}(3, 6)$ lie in the interval $[3, 6]$ and similarly, the tuples of relation `R` that violate the constraint given tuple $\mathtt{l}(5, 10)$ lie in the interval $[5, 10]$. If the constraint is not violated then we can infer that the tuples of the inaccessible relation lie outside the *forbidden intervals* $[3, 6]$ and $[5, 10]$ and therefore outside the combined forbidden interval $[3, 10]$.

Let tuple `l`(a, b) be inserted into relation `L`. If $a \geq 3$ and $b \leq 10$, then the forbidden interval for `l`(a, b) is contained in the union of the forbidden intervals of one or more existing tuples, and relation `R` need not be accessed in order to infer that constraint C is not violated. Note, the complete test may need to access multiple existing tuples in order to make the above inference. An incomplete, but sufficient, test would be to check that the forbidden interval for some single existing tuple contains the forbidden interval for the inserted tuple. That corresponds to using a single tuple in the accessible relation as opposed to using an arbitrary number of tuples, and was the approach taken in our initial work [GW93]. The sufficient test is linear in the number of tuples in `L` whereas the complete test could be exponential, if implemented naively. With some preprocessing, the complete test can also be evaluated in linear time [GSUW93].

The complete test for this example is the following recursive Datalog program that derives `insertion_ok` if and only if the inserted tuple does not violate C, assuming that C was not violated before the insertion.

`insertion_ok` :– `inserted`(A, B) & `forbidden_int`(C, D) & $A \geq C$ & $B \leq D$.
`forbidden_int`(C, D) :– `l`(C, D).
`forbidden_int`(C, D) :– `forbidden_int`(C, X) & `forbidden_int`(Y, D) & $X \geq Y$.

□

EXAMPLE 3. Consider a constraint C that involves two variables in the inaccessible relation:

C: **panic** :– `l`(U, V, W, Z) & `r`(X, Y) & $U \leq X \leq V$ & $W \leq Y \leq Z$.

Intuitively, the above constraint is the *forbidden interval* constraint in two dimensions. A tuple in relation `R` defines a point in a two dimensional space and a tuple in relation `L` defines a rectangular region in this 2-D space. Constraint C requires that all the points defined by the inaccessible relation `R` lie outside every rectangular region defined by the accessible tuples. Therefore, an inserted tuple `l`(a, b, c, d), does not violate C if the rectangle defined by `l`(a, b, c, d) is contained in the union of the rectangles defined by the existing tuples in `L`. The test for determining when a rectangle is contained in a set of other rectangles can still be represented as a recursive Datalog program. However, building the program is not as straightforward as in Example 2. In addition, the complexity of the test is high even with preprocessing. Without preprocessing the test is exponential in the number of tuples in `L`. □

4 Discussion

In [GSUW93] we identify some subclasses of conjunctive query constraints with arithmetic comparisons for which the complete test is a (recursive) Datalog program. We also identify some subclasses where the complete test does not need to consider multiple tuples from the accessible relation, but can consider tuples one at a time (i.e., there is no need to consider combinations of tuples, as in Example 2). The test condition for conjunctive query constraints, including the subclasses, is NP-complete. However, in at least some cases, the exponential behavior is only in the size of the constraint specifications, which we believe will be relatively small. In other cases the tests may be exponential in the size of the database or the number of constraints in the system. In such cases sufficient tests, instead of complete tests, may be preferable.

For conjunctive query constraints that use function symbols like $+, -$ (instead of only arithmetic comparisons), the complete test is an implication condition where both sides of the implication use disjunction and the function symbols. Even though the implication condition can be derived in time exponential in the size of the constraint, evaluating the implication may be undecidable or have very high complexity. However, for some subclasses the ideas outlined in this paper can be extended to derive sufficient decidable tests.

5 Future Research Directions

Many interesting avenues remain unexplored in making constraint checking efficient following the framework we outlined above. We plan to:

- Consider more expressive constraint languages. Aggregate functions like MAX, SUM, AVG, *etc.* make the constraints more general. For instance, we might want a constraint requiring the average gradepoint of a graduating student to be at least 3.
- Use different amounts and type of information. For instance, constraint C_1 might be checked using constraints C_2 and C_3, possibly together with some functional dependency information. In distributed database systems, such algorithms can be used to increase the amount of constraint checking that can be done locally, without accessing remote data.
- Devise algorithms to efficiently perform local tests. As the examples in this paper illustrate, the test conditions often have high complexity. Techniques from constraint logic programming, operations research, and other areas provide ways of evaluating the tests efficiently. For instance, in Examples 2 and 3, algorithms from computational geometry are useful for efficient evaluation.
- For constraints where the complexity of local checking is inherently very high, it is useful to look for sufficient tests that are efficient to implement even though they may not be complete.

References

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

[BC79] Peter O. Buneman and Eric K. Clemons. *Efficiently Monitoring Relational Databases.* In *ACM Transactions on Database Systems*, Vol 4, No. 3, 1979, 368-382.

[CM77] Ashok K. Chandra and P.M. Merlin. Optimal Implementation on Conjunctive Queries in Relational Databases. In *9th ACM Symposium on Theory of Computing*, pages 77–90, ACM, 1977.

[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, 1990.

[GMS93] Ashish Gupta, Inderpal Singh Mumick, and V. S. Subrahmanian. Maintaining Views Incrementally. In *Proceedings of ACM SIGMOD 1993 International Conference on Management of Data*, pages 157–167.

[GSUW93] Ashish Gupta, Shuky Sagiv, Jeffrey D. Ullman, and Jennifer Widom. Constraint Checking with Partial Information. Submitted for publication, December 1993.

[GT93] Ashish Gupta and Sanjai Tiwari. Distributed Constraint Management for Collaborative Engineering Databases. In *Proceedings of the Second International Conference on Information and Knowledge Management (CIKM)*, Washington DC, November 1993.

[GW93] Ashish Gupta and Jennifer Widom. Local Checking of Global Integrity Constraints . In *Proceedings of ACM SIGMOD 1993 International Conference on Management of Data*, pages 49–59.

[HD92] John V. Harrison and Suzanne Dietrich. Maintenance of Materialized Views in a Deductive Database: An Update Propagation Approach. In *Workshop on Deductive Databases, JICLSP 1992*, pages 56–65, 1992.

[Klu88] A. Klug. On Conjunctive Queries Containing Inequalities. *Journal of the ACM*, 1(35):146–160, 1988.

[LS93] A.Y. 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.

[Nic82] J. M. Nicolas. Logic for Improving Integrity Checking in Relational Data Bases. *Acta Informatica*, 18(3):227–253, 1982.

[TH93] Sanjai Tiwari and H. C. Howard. Constraint Management on Distributed AEC Databases. In *Fifth International Conference on Computing in Civil and Building Engineering*, pages 1147–1154. ASCE, 1993.

[QW91] Xiaolei Qian and Gio Wiederhold. *Incremental Recomputation of Active Relational Expressions.* In *TKDE*, 1991.

[SY80] Yehoshua Sagiv and Mihalis Yannakakis. Equivalences Among Relational Expressions with the Union and Difference Operators. *Journal of the ACM*, 4(27):633–655, 1980.

[Ull88] J. D. Ullman. *Principles of Database and Knowledge-Base Systems*, Volumes 1 and 2. Computer Science Press, New York, 1989.

[UO92] Toni Urpi and Antoni Olive. A method for change computation in deductive databases. In *Proceedings of the Eighteenth International Conference on Very Large Databases (VLDB)*, pages 225–237, Vancouver, British Columbia, 1992.