Database Integrity Checking

Hendrik Decker
Universidad Politécnica de Valencia, Spain

Davide Martinenghi
Free University of Bozen/Bolzano, Italy

INTRODUCTION

Integrity constraints (or simply "constraints") are formal representations of invariant conditions for the semantic correctness of database records. Constraints can be expressed in declarative languages such as datalog, predicate logic, or SQL. This article highlights the historical background of integrity constraints and the essential features of their simplified incremental evaluation. It concludes with an outlook on future trends.

BACKGROUND

Integrity has always been an important issue for database design and control, as attested by many early publications (e.g., Bernstein & Blaustein, 1982; Bernstein, Blaustein, & Clarke, 1980; Codd, 1970, 1979; Eswaran & Chamberlin, 1975; Fraser, 1969; Hammer & McLeod, 1975; Hammer & Sarin, 1978; Nicolas, 1978, 1982; Wilkes, 1972); later ones are too numerous to mention. Expressing database semantics as invariant properties persisting across updates had first been proposed by Minsky (1974). Florentin (1974) suggested expressing integrity constraints as predicate logic statements. Stonebraker (1975) proposed formulating and checking integrity constraints declaratively as SQL-like queries.

Functional dependencies (Armstrong, 1974; Codd, 1970) are a fundamental kind of constraints to guide database design. Referential integrity has been part of the 1989 SQL ANSI and ISO standards (McJones, 1997). The SQL2 standard (1992) introduced the CHECK and ASSERTION constructs (i.e., table-bound and table-independent SQL query conditions) as the most general means to express integrity constraints declaratively (Date & Darwen, 1997). Since the 1990s, uniqueness constraints, foreign keys, and complex queries involving EXISTS and NOT became common features in commercial databases. Thus, arbitrarily general and complex integrity constraints can now be expressed and evaluated in most relational databases. However, most of them offer efficient support only for the following three simple kinds of declarative constraints:

- **Domain Constraints:** Restrictions on the permissible range of attribute values of tuples in table columns, including scalar SQL data types and subsets thereof, as well as options for default and null values.
- **Uniqueness Constraints:** As enforced by the UNIQUE construct on single columns, and UNIQUE INDEX and PRIMARY KEY on any combination of one or several columns in a table, preventing multiple occurrences of values or combinations thereof.
- **Foreign Key Constraints:** For establishing a relationship between the tuples of two tables, requiring identical column values. For instance, a foreign key on column emp of relation works_in requires that the emp value of each tuple of works_in must occur in the emp_id column of table employee, and that the referenced column (emp_id in the example) has been declared as primary key.

For more general constraints, SQL manuals usually recommend using procedural triggers or stored procedures instead of declarative constructs. This is because such constraints may involve nested quantifications over huge extents of several tables. Thus, their evaluation can easily become prohibitively costly. However, declarativity does not need to be sacrificed for efficiency, as shown by many methods of simplified integrity checking as cited in this survey. They are all based on the seminal paper (Nicolas, 1982).

SIMPLIFIED INCREMENTAL INTEGRITY CHECKING

A common idea of all integrity checking methods is that not all constraints need to be evaluated, but at most those that are possibly affected by the incremental change caused by database updates or transactions. Anticipating updates by patterns, most incremental integrity checking methods allow for simplifications of constraints to be generated already at schema compilation time. Such compiled simplifications are parametric conditions to be instantiated, possibly further optimized, and evaluated upon given update requests. For generating them, only the database schema, the integrity constraints, and the update patterns are needed as input.

Their evaluation, however, may involve access to the stored data at update time. Methods that generate compiled simplifications are described, for example, by Christiansen and Martinenghi (2006), Decker (1987), and Leuschel and De Schreye (1998). For unanticipated ad-hoc updates, the generation of simplifications takes place at update time. Optimizations for efficient evaluation of simplified constraints are addressed, for example, by Sheu & Lee (1987).

Simplifications can be distinguished by the database state in which they are evaluated. *Post-test* methods must evaluate their simplifications in the *new*, updated state, for example, Decker and Celma (1994), Grant and Minker (1990), Lloyd, Sonenberg, and Topor (1987), Nicolas (1982), and Sadri and Kowalski (1988). *Pre-test* approaches, for example, Bry, Decker, and Manthey (1988), Christiansen and Martinenghi (2006), Hsu and Imielinski (1985), McCune and Henschen (1989), and Qian (1988), only access the *old* state before the update, that is, they need not execute the update prematurely, since undoing an updated state if integrity is violated is costly. In case of integrity violation, the eagerness of pre-tests to avoid rollbacks is a clear performance advantage over post-tests.

For convenience, a finite set of constraints imposed on a database D is called an *integrity theory* of D. For a database D and an integrity theory IC, let $D(IC)$ = *satisfied* denote that IC is satisfied in D, and $D(IC)$ = *violated* that it is violated. Further, for an update U, let D^U denote the updated database. Any simplification method M can be formalized as a function that takes as input a database, an integrity theory and an update, and outputs either *satisfied* or *violated*. Thus, soundness and completeness of M can be stated as follows:

Let D be any database, IC any integrity theory, and U any update. Suppose that $D(IC)$ = *satisfied*. Then, an integrity checking method M is *sound* if the following holds:

If $M(D, IC, U)$ = *satisfied* then $D^U(IC)$ = *satisfied*.

It is *complete* if the following holds:

If $D^U(IC)$ = *satisfied* then $M(D, IC, U)$ = *satisfied*.

This formalism is applicable to most integrity checking methods in the literature. Many of them are sound and complete for significant classes of relational and deductive databases, integrity theories, and updates. Some methods, however, are only shown to be sound, that is, they provide sufficient conditions that guarantee integrity satisfaction of D^U, for example, Gupta, Sagiv, Ullman, and Widom (1994); further checking is required if these conditions are not satisfied. The main advantage is that the evaluation of $M(D, IC, U)$ is typically much simpler than that of $D^U(IC)$.

Most integrity checking methods can be described by distinguishing three (possibly interleaved) phases, namely the *generation*, *optimization*, and *evaluation* of simplified tests. Next, these phases, numbered I, II, III, are illustrated by an example.

EXAMPLE

Consider a relational database with tables for workers and managers, defined as follows:

```
CREATE TABLE(worker(CHAR name, CHAR department))

CREATE TABLE(manager (CHAR name)).
```

Suppose there is an integrity constraint requiring that no worker is a manager, expressed as a denial by the following SQL condition, which forms the body of a related SQL assertion:

```
NOT EXISTS (SELECT * FROM worker, manager WHERE
worker.name = manager.name).
```

If the number of workers and managers is large, then checking whether this constraint is violated or not can be very costly. The number of facts to be retrieved and tested is in the order of the cardinality of the cross product of worker and manager, whenever the constraint is checked. Fortunately, however, the frequency and amount of accessing stored facts can be significantly reduced when going through phases I -III. Beforehand, a possible objection at this stage should be dealt with.

SQL programmers might feel compelled to point out that the previous constraint is probably much easier checked by some trigger such as the following one in MS SQL Server syntax:

```
CREATE TRIGGER no_worker_manager ON worker FOR
INSERT : IF EXISTS

(SELECT * FROM inserted, manager WHERE inserted.name
= manager.name) ROLLBACK.
```

Its evaluation would only need to access manager and a cached relation inserted containing the row to be inserted to worker, but not the stored part of worker. However, it is easily overlooked that the sample integrity constraint entails that somebody who is promoted to a manager must not be a worker, thus necessitating a second trigger for insertions into manager. In general, each occurrence of each atom occurring in a constraint requires a separate trigger, and it is by far not always as obvious as in the simple previous example how they should look. Apart from being error-prone, hand-