Principles of Advanced Database Integrity Checking

Hendrik Decker
Instituto Tecnológico de Informática, Spain

INTRODUCTION

Integrity constraints (hereafter, sometimes simply 'constraints') are formal representations of conditions for the semantic correctness of database records. In science, constraints are usually expressed in declarative knowledge representation languages such as datalog or predicate logic. In commercial databases, they are usually expressed by distinguished SQL statements.

BACKGROUND

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

Referential integrity, a special case of functional dependencies (Armstrong, 1974), has been included in the 1989 SQL ANSI and ISO standards (McJones, 1997). The SQL2 standard (1992) introduced the CHECK option and the ASSERTION construct as the most general means to express arbitrary integrity constraints declaratively in SQL (Date & Darwen, 1997). In the 1990s, uniqueness constraints, foreign keys, and complex queries involving EXISTS and NOT became common features in commercial database products. Thus, arbitrarily general integrity constraints could now be expressed and evaluated in most relational databases.

Integrity constraints may involve nested quantifications over huge extents of several tables. Thus, their evaluation can easily become prohibitively costly. Most SQL databases offer efficient support only for the following three simple kinds of declarative constraints:

- Domain constraints, i.e., restrictions on the permissible range of scalar attribute values of tuples in table columns, including 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 an identity relationship between columns of two tables. For instance, a foreign key on column emp of relation works_in may require that each emp value of works_in must occur in the emp_id column of table employee, where the referenced columns (here, emp_id) must be a primary key.

For more general constraints, SQL database manuals usually ask the designer to renounce declarative constructs and instead resort to procedural triggers and stored procedures. However, declarativity does not need to be sacrificed in order to obtain efficiency. One approach developed to that end, in the framework of predicate logic and datalog, was *soundcheck* (Decker, 1986). In the spirit of the latter, a translation of integrity constraints expressed in predicate logic into SQL is described in Decker (2003).

SIX PHASES OF SIMPLIFIED INTEGRITY CHECKING

Below, the *soundcheck* approach for simplifying the evaluation of integrity constraints is outlined as a succession of six phases. Except Step I, proposed in Decker (1987), this approach originates in Nicolas (1982). All or part of it is used in one way or another in most known methods for integrity checking. It can be easily implemented in SQL (Decker, 2003). In the next section of this article, Steps I-VI are illustrated with an example. The six phases are then discussed in general.

I Generate the difference between the old and the new state
II Skip idle updates
III Focus on relevant integrity constraints
IV Specialize relevant constraints
V Optimize specialized constraints
VI Evaluate optimized constraints

AN EXAMPLE OF SIMPLIFIED INTEGRITY CHECKING

For illustrating Steps I-VI, consider an update of a relational database with tables for workers and managers, defined as follows.

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

Now, suppose there is an integrity constraint requiring that no worker is a manager, expressed by the SQL condition:

```
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 by taking Steps I-VI. Before walking through them, a possible objection at this stage needs to be dealt with.

SQL programmers might feel compelled to point out that the constraint above is probably much easier checked by a trigger such as:

```
CREATE TRIGGER 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 also requires implicitly 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 example above how they should look like. Apart from being error-prone, hand-coded triggers may also bring about unpredictable effects of mutual interactions that are hard to control. Hence, hand-coding triggers, as recommended in many database manuals, hardly seem advisable.

Now, let INSERT INTO worker VALUES ('Fred', 'sales') be an update. Then, running Steps I through VI means the following:

I Generate difference between old and new state

The explicit update INSERT INTO worker VALUES ('Fred', 'sales') may have implicit update consequences on database views, the definition of which involves worker. The set of explicitly and implicitly updated facts constitutes the difference Δ between old and new database state. Each fact in Δ may violate integrity. Thus, Δ must be generated, and each fact in Δ needs to be run through Steps II-VI. For example, suppose a view containing all workers entitled to some benefit, for example, if they work in some distinguished department d, and a constraint C on that view. Then, C needs to be evaluated only if Fred's department is d; otherwise, no additional constraint needs to be checked.

II Skip idle updates

If Fred already has been a worker (e.g., in some other department) before the INSERT statement was launched, then it is not necessary to check again that he must not be a manager, since that constraint has already been satisfied before.

III Focus on relevant integrity constraints

Unless II applies, the constraint that no worker must be manager is clearly relevant for the given update and hence must be checked. Any integrity constraint that is not relevant for the insertion of rows into the worker table needs not be checked. For instance, a constraint requiring that each department must have some least number of workers is not relevant for insertions, but only for deletions in the worker table. Also, constraints that do not involve worker need not be checked.

IV Specialize relevant constraints

For the given INSERT statement, the WHERE clause of the SQL condition:

```
EXISTS (SELECT * FROM worker, manager WHERE
worker.name = manager.name)
```

can be specialized to a much less expensive form: