

An Amateur's Introduction to Integrity Constraints and Integrity Checking in SQL

Andreas Behrend, Rainer Manthey and Birgit Pieper

University of Bonn, Institute of Computer Science III,
Römerstr. 164, D-53117 Bonn, Germany
idb@cs.uni-bonn.de

Abstract. In this paper we survey the various forms of integrity constraints provided in SQL. A critical discussion of the implications of SQL's approach to integrity follows, particularly identifying misconceptions to be found in some textbooks. The relationship between views and constraints in SQL is investigated and contrasted with the state-of-the-art in integrity checking for deductive databases. As recursive views have been introduced into SQL recently, integrity maintenance over recursive views will become an issue for future DBMSs supporting the new standard in full.

1 Introduction

Integrity constraints are a classical feature of data models, and their automated checking and enforcement is by now firmly established as a key functionality of a DBMS. Textbooks on databases distinguish static and dynamic constraints, the former being regarded as most important. Their nature as invariants against state modifications is well understood by now. However, paragraphs about integrity tend to be short in most DB books, often limited to an outline of the most popular types of constraints such as key and referential constraints. Integrity checking methods are usually not presented at all. The SQL standard offers an elaborate system of various forms of static constraints, providing schema designers with a powerful tool for expressing a wide range of state invariants.

Commercial relational DBMS products supporting SQL, however (such as, e.g., Oracle [Ora99] or DB2 [IBM99]) do not support the more advanced forms of constraints. Even today, more than 8 years after the SQL'92 standard has been issued, none of these commercial systems supports assertions, the most general form of constraint in SQL! Scientific literature, i.e. research papers, devoted to integrity on the other hand provide a wealth of promising results applicable to very general and powerful forms of integrity constraints.

Our group has been conducting research on integrity since many years, mainly in the context of Datalog rather than SQL, e.g. [BDM88,BMM92,Man94]. Recently, we took a closer look at SQL constraints, motivated by preparations for a new SQL course and by work on a forthcoming dissertation. Up till then, our knowledge about integrity in SQL did not go significantly beyond the level of treatment as provided by database textbooks. A closer look

at syntax and semantics of SQL constraints as presented in popular SQL books [Date97,MS93,GW99,GP99] revealed quite a few contradictory or at least puzzling statements, in particular about those features like multi-table check constraints and assertions presently not supported by major database vendors. It turned out that obtaining reliable and complete information about integrity in SQL is surprisingly difficult.

By now, we believe to know much more about the "true" nature of SQL's integrity concept, mainly based on a careful analysis of the new SQL3 standard document [ANSI99] and resulting from numerous discussions about the implications of our findings. In this paper, we summarize what we believe to be useful results of this enterprise. In particular, we focus on

- the differences between table check constraints and assertions,
- the relationship between views and constraints, and
- the crucial question of efficiency of integrity checking.

Issues covered intensively by the existing literature (such as referential integrity, e.g.) will be treated by passing only. Furthermore, we simplify matters by excluding duplicates and nulls, following well-founded recommendations by Chris Date in his classical book on SQL [Date97]. For space reasons we do not touch the issue of triggers and their relationship to integrity either, even though the trigger concept is a significant addition of SQL3 and clearly matters as far as integrity is concerned.

The title of our paper paraphrases the titles of two famous database papers written many years ago on recursive query processing [BR86] and normalization theory [BBG78]. Each of them surveys a specific topic not well-covered in generally accessible literature at that time, and each presents results with a certain degree of critical distance to established opinions and views. We are certainly not aiming at a formal treatment of the issue as in the normalization paper, but try to follow the spirit of the paper on recursion. We do so, because a formal treatment of any issue related to SQL is a space consuming exercise beyond the scope of a conference paper. Nevertheless we would like to point out already here that the SQL standard is pretty formal and exact as far as the basic semantic decisions about constraints is concerned.

After a short summary of the main features of the SQL constraint system a more thorough discussion of selected aspects will follow. The relationship between views and constraints is investigated in section 4, efficient techniques for incremental integrity checking are surveyed in section 5. A summary of positive and - in our opinion - negative aspects of design decisions in SQL concludes the paper.

2 Integrity in SQL: A Mini-Tutorial

In this section we will briefly summarize the way how SQL deals with constraints and integrity. A thorough coverage of syntactical details is beyond the scope of this paper, readers are referred to, e.g., chapter 14 in Date's book. We will follow

the standard's terminology as far as possible, but will have to deviate from or extend it where notions are missing or conventions appear inconsequent to us.

2.1 Syntax: SQL Constraints and Assertions

Rather than offering a single, homogeneous constraint concept, SQL provides four main syntactic categories of static constraints:

- assertions,
- table constraints,
- column constraints, and
- domain constraints.

In order to avoid confusion, we will use the term "integrity constraint" (short: IC) whenever we refer to both, SQL constraints and SQL assertions. Each IC has to have a unique constraint name in SQL (generated by the DBMS if missing in a constraint definition, mandatory however for assertions).

Assertions are the most general form of IC in SQL. They correspond most directly to the general perception of static ICs as invariants. Each assertion is introduced in a separate assertion definition, containing - apart from the constraint name - a single search condition prefixed by the keyword CHECK. As an example consider:

```
CREATE ASSERTION ac1
CHECK (NOT EXISTS (SELECT * FROM t1
                   WHERE a11 NOT IN
                   (SELECT a21 FROM t2)));
```

Here, ac1 is the name of the assertion, t1 and t2 are names of tables restricted by the assertion, and a11 and a21 are column names. Search conditions are arbitrary Boolean SQL expressions: In principle every expression which can appear in the WHERE part of a select clause is admissible as a search condition in an assertion. Assertions are intended to be used specifically for expressing multi-table ICs. However, single-table ICs are acceptable according to the standard either.

Table constraints appear to be less general than assertions as they are intended to restrict the rows in one particular table only. As such they are always "attached to" (or: "associated with") a particular table by including them into the CREATE TABLE statement defining that table. The respective table definition in addition provides a context for interpreting shorthand notations which may be used in a table constraint, but not in an assertion. There are three categories of table constraints:

- table unique constraints (UNIQUE, PRIMARY KEY),
- table referential constraints (FOREIGN KEY), and
- table CHECK constraints.

Table unique and table referential constraints are specialized syntactic forms for expressing key and foreign key properties: A "table check constraint" (TCC for short) is a generic constraint able to impose arbitrary restrictions on rows of the respective table. As an example consider:

```

CREATE TABLE t2 (... ,
    CONSTRAINT tc21 PRIMARY KEY a21,
    CONSTRAINT tc22 FOREIGN KEY a22 REFERENCES t3(a31),
    CONSTRAINT tc23 CHECK (a21 > 0 AND NOT a21 = a22));

```

A **PRIMARY KEY** constraint is a stronger form of **UNIQUE** constraint preventing the respective column(s) from containing **NULLs** (which we exclude in this paper anyway). Moreover, there may be one primary key constraint per table only. **FOREIGN KEY** constraints may in addition introduce certain "referential actions" to be automatically triggered by the DBMS on violations of such constraints. We do not address this issue here, as it is orthogonal to the integrity checking problem and closely related to triggers.

As for assertions, any SQL search condition is acceptable in a TCC according to the standard, even though conditions at least referencing the table defined are clearly intended here. The following check constraint conforms to the standard, but not to the intention of the concept:

```

CREATE TABLE t3 (... ,
    CONSTRAINT tc31 CHECK (EXISTS(SELECT * FROM t2)));

```

Column constraints are analogously divided into column unique constraints, column referential constraints, and column check constraints. They provide shorthand notations for table constraints applying to values of one particular column only, and are directly embedded into the respective column definition. Each column constraint may be equivalently expressed by a TCC (see Date for details). **Domain constraints** can be used to "factor out" conditions to be imposed on every column ranging over the respective domain. Without the domain constraints feature, such conditions would have to be expressed as column constraints in every such column definition. Thus, each domain constraint is a shorthand for one or more column constraints. We omit domain and column constraints in the following as they do not present any specific problems as compared to TCCs.

2.2 Semantics: Reducing Constraints to Assertions

The original motivation for distinguishing different constraint categories was the desire to offer various levels of granularity for expressing ICs, depending on the syntactic unit subject to the restriction expressed by the IC. On each such level, certain ICs expressible on a higher level as well may be expressed more concisely due to convenient shorthand notations. The semantics of each of these categories is explicitly defined in the standard by means of a systematic rewriting into a logically equivalent form in a category of higher granularity:

- Domain constraints are translated into column constraints,
- column constraints are translated into table constraints, and
- table constraints are finally translated into assertions.

As an example illustrating this systematic way of reducing low granularity constraints ultimately to assertions consider the following definition of a domain d2 referenced (among others) in table t4:

```
CREATE DOMAIN d2 AS INTEGER
      CONSTRAINT dc21 CHECK (VALUE BETWEEN 1 AND 10);
```

```
CREATE TABLE t4 (a41 d2, ...);
```

After folding dc21 as a column check constraint into the table definition, and expanding the resulting column constraint into a TCC on t4, this TCC can be finally extracted from the context of the definition of t4 and turned into an independent assertion:

```
CREATE ASSERTION ac2
      CHECK (NOT EXISTS
            (SELECT * FROM t4
             WHERE NOT (a41 BETWEEN 1 AND 10)));
```

The semantics of this assertion defines the semantics of all its shorthand versions. A similar rewriting convention applies to unique and referential constraints. Each of them can be systematically transformed into an equivalent TCC. As an example consider:

```
CREATE TABLE t5 (... ,
      CONSTRAINT tc51 UNIQUE (a51,a52));
```

This can be rewritten (assuming duplicate-free tables) into:

```
CREATE TABLE t5 (... ,
      CONSTRAINT tc51' CHECK (NOT EXISTS
            (SELECT * FROM t5 AS X WHERE
             EXISTS (SELECT * FROM t5 AS Y
                     WHERE X <> Y AND
                           X.a51 = Y.a51 AND
                           X.a52 = Y.a52))));
```

Such a transformation exercise impressively motivates the usefulness of an abbreviated notation for unique constraints!

It is important to understand this well-designed hierarchy of stepwise refinement underlying the SQL constraint system. Ultimately, assertions alone are sufficient for expressing any kind of static constraint. However, we will keep TCCs as the only other category as they can be viewed as a kind of prototype form of a shorthand IC. Moreover, various quite intricate problems arise from the duality of TCCs and assertions (to be discussed later).

2.3 Basics of Integrity Checking in SQL

Now we turn to the way ICs have to be checked during an SQL transaction. Two questions have to be answered: at which point in time does checking take place,

and what is actually checked at this moment. The standard [ANSI99] is very clear about this issue (4.17.1 Checking of constraints): There are two constraint modes, **immediate** and **deferred**. An immediate constraint or assertion is effectively checked at the end of (or: immediately after) each SQL-statement, deferred ICs at the end of the current transaction. If no explicit constraint mode has been specified (as in all our examples up till now), immediate checking is implicitly assumed as a default mode.

The mode of an IC may be changed from immediate to deferred during a transaction unless the respective IC has been declared non-deferrable. Changing from deferred to immediate is possible as well. If this happens, the respective IC is immediately checked over the state reached when the change took place. Throughout, whenever a constraint or assertion is not satisfied when checked, an exception is raised. *Not being satisfied* is defined in [ANSI99] quite clearly:

”An assertion is satisfied if and only if the specified <search condition> is not false.” (4.17.4)

For TCCs, the standard is a bit less precise, when stating

”A table check constraint is satisfied if and only if the specified <search condition> is not false for any row of a table.” (4.17.2)

Saying ”not false for any row of a table” does not state that the table called ”a table” actually is (or even has to be) the table to which the respective table constraint is attached. In a later part the ambiguity is resolved, however:

”.. let SC be the <search condition> .. and let T be the table name ..” of a table check constraint; ”the table constraint is not satisfied if and only if EXISTS (SELECT * FROM T WHERE NOT SC) is true.” (11.6)

The universal statement ”not false for any row”, however, clearly conforms with the transformation semantics stated in 11.6, as a universal quantifier has to be expressed in SQL by NOT EXISTS (SELECT ... WHERE NOT ...).

3 A Closer Look at Constraints and Assertions in SQL

So far we have been summarizing what is to be found in textbooks and in the standard about syntax and semantics of ICs. Everything appears to be pretty clear until you begin to ask more detailed questions and to consider less obvious examples. In this section, we discuss selected aspects more in-depth and challenge the relevant sources for more elaborate issues.

3.1 Explicit and Implicit Quantification

An assertion is essentially a Boolean expression. Even though the SQL standard does not state any restrictions on the particular kind of expression to be used as an assertion, there certainly are such restrictions. Whereas, e.g.,

```
CHECK (NOT EXISTS (SELECT * FROM t6 WHERE a61 < 0))
```

is a perfect candidate assertion, the following is not (even though permitted by the standard):

```
CHECK (a > 0)
```

Each row variable explicitly or implicitly occurring in an assertion has to be "bound" by some explicit or implicit quantifier. In the first condition, the table name t6 serves as a row variable bound by the quantifier in front of the select clause. The second condition is a shorthand for $X.a > 0$, where X is an implicit variable ranging over some "anonymous" table for which a column called 'a' has been declared. If embedding this comparison into an assertion declaration, there would be no such table nor any quantifier binding the variable. Quite obvious in a strikingly trivial example like this, but less obvious in one of the many SQL formulations where hidden quantifiers have to be made explicit in order to decide whether each variable occurring is indeed bound. An example of such quantifier appears in the following inclusion dependency:

```
CHECK (NOT EXISTS (SELECT * FROM t1 WHERE
                   (a11 NOT IN (SELECT a21 FROM t2))))
```

The innermost select clause introduces a variable t2 which is implicitly quantified due to its embedding into an IN condition. An equivalent formulation avoiding the IN reveals this quantifier:

```
CHECK (NOT EXISTS (SELECT * FROM t1 WHERE
                   NOT EXISTS (SELECT * FROM t2
                               WHERE a11 = a21))))
```

The condition `CHECK a > 0` may, however, be used as a TCC, because any `CREATE TABLE` statement is associated with an implicit variable ranging over rows of the respective table. It serves as reference for every column name not otherwise bound. Moreover, implicit universal quantification is assumed by the standard for this implicit variable associated with each table definition (as shown in the previous section). In the context of

```
CREATE TABLE t (a INTEGER, ..., CHECK (a > 0));
```

the semantics of the TCC can be made explicit as follows:

```
CREATE TABLE t (... ,
                 CHECK (NOT EXISTS (SELECT * FROM t
                                     WHERE NOT (a > 0))));
```

It is only because of these two assumptions - implicit variable and implicit universal quantifier - that the "trivial comparison" condition not acceptable as an assertion works as a TCC. Both, the standard as well as the book by Date introduce the rewriting semantics for TCCs explicitly, however, the wellformedness requirements arising from the subtle interaction between implicit and explicit variables and quantifiers are mentioned only by Date in a footnote on p. 207 (without elaborating on such requirements any further).

3.2 Check Constraints versus Assertions

Due to this convention of automatic expansion of check constraints into a fully quantified form, TCCs can always be reformulated as assertions (see Date, p. 204). But is the converse true as well? Date claims that this is always possible (p. 203), Celko [Cel00] claims the opposite:

”An assertion can do things that a CHECK() clause attached to a table cannot do, because it is outside of the tables, involved.” (p. 139)

Discovering the ”true” relationship between TCCs and assertions indeed turned out to be one of the toughest job during our investigations. There are two issues determining the answer to this question: The treatment of multi-table TCCs, and the effect of empty tables on IC semantics. Let us turn to multi-table check constraints first. Groff/Weinberg [GW99] claim that such constraints are not allowed at all:

”An assertion is a database constraint that restricts the contents of the database as a whole ... But unlike a check constraint, the search condition in an assertion can restrict the contents of multiple tables and the data relationships among them.” (p. 382)

Date [Date97] clearly states the opposite:

”Note carefully, however, that *associated with a specific base table* does not mean that such a constraint cannot refer to other base tables; Rather it means simply that the constraint cannot exist if the associated base table does not exist ...” (p. 204)

Note here, that assertions referencing a non-existing table are not well-formed either! Melton [MS93] agrees with Date, but recommends not to make use of multi-table TCCs:

”It is possible (given some awkwardness) to express restrictions involving multiple tables with regular table CHECK constraints, but this is not really recommended. Instead, you should use assertions for that purpose.” (p. 211)

The standard, as the ultimate reference, does not contain anything about this issue. There is no exclusion of the feature and no recommendation about when to use a TCC or an assertion. Any search condition is admitted as a TCC - that’s it! So Date is right, Celko is wrong, and Melton probably gives the best advice on how to deal with this situation.

Given that multi-table TCCs are possible: When to check such constraints? Every source on SQL constraints states that a TCC associated with a table T is to be checked whenever this particular table T is modified. Hardly anybody, however, states that TCCs might have to be checked on modifications of other tables than T, too. Omitting such a hint does not mean to make a literally false statement, but it means at least to raise a false impression. We assumed for

quite a while that "to be checked on every modification of T" is to be read as an if-and-only-if statement. Only after we came to the following statement by Melton [MS93], p. 206, our erroneous interpretation became apparent:

"You should be aware that any constraint that references data in more than one table is checked after a change to any of those tables."

Re-reading the standard after finding this citation revealed that there was no hint supporting the misconception of limiting TCC checking to modifications of the "owner" table either. Quite the opposite: In the section on constraint checking ([ANSI99], 4.7.1, p.21) the standard defines that - at least in principle - every immediate constraint has to be checked immediately after each statement, and every constraint in general has to be checked at transaction end. Proceeding this way would indicate a constraint violation if an embedded condition in a TCC referencing a foreign table would evaluate to false, independent of the question whether the statement executed manipulates the table to which the respective TCC is attached, or not.

Both observations together - multi-table TCCs are allowed and are to be checked on every change of a table involved - lead to the question about the difference between assertion and TCC. Apart from the possibility to make use of shorthands in a TCC there seems to be no difference left. So when to use a TCC, and when to use an assertion? Consider again the inclusion dependency already discussed earlier:

```
CHECK (NOT EXISTS (SELECT * FROM t1 WHERE
                  NOT EXISTS (SELECT * FROM t2
                              WHERE a11 = a21)))
```

This condition can be violated by either inserting a row into t1, or by deleting a row from t2, or by modifying a column in either table. The same IC can be expressed as a TCC associated with t1 by omitting the leading t1-quantifier:

```
CREATE TABLE t1 ( ...,
                 CHECK (EXISTS (SELECT * FROM t2 WHERE a11 = a21)));
```

The resulting TCC is semantically equivalent (due to the implicit quantifier) and syntactically shorter than the assertion. But why attach it as a TCC to t1 rather than (alternatively, of course) to t2?

Surprisingly, this doesn't work, as there is no reformulation of the full condition into an equivalent form containing a leading universal quantifier ranging over t2. The t2-quantifier cannot be moved to the front without losing equivalence. Thus, a shorthand notation cannot be used. The solution to attach the full condition, keeping both quantifiers in the order originally chosen, to t2 does not lead to an equivalent solution either. If attached to t2, the full inclusion dependency would become semantically equivalent to the following three-quantifier condition due to the implicit expansion governing the semantics of TCCs:

```
CHECK (NOT EXISTS (SELECT * FROM t2 WHERE NOT
                  (NOT EXISTS (SELECT * FROM t1 WHERE
                              NOT EXISTS (SELECT * FROM t2
                                           WHERE a11 = a21))))))
```

The outer t2-quantifier is not referenced in the inner part (representing the original dependency) at all, but nevertheless the extra quantifier leads to a change in semantics of the resulting new condition as compared with the original dependency. This is due to a bordercase, which is easily neglected!

Whenever t2 is empty in a particular state, the inclusion dependency is immediately violated, unless t1 is empty, too. However, a leading NOT EXISTS ranging over t2 will always be satisfied over states in which t2 happens to be empty, independent of the truth value of the WHERE part of the subsequent select clause: The select returns the empty set from an empty t2, thus the EXISTS is false, and the NOT EXISTS true. Date addresses the issue on p. 208, and calls this circumstance "counterintuitive". But it is only Melton who in addition discusses the consequence of this observation with respect to comparing assertions and TCCs. He states that assertions and TCCs are not equivalent in expressivity because of this very problem of empty tables.

But even this presumably clear statement has to be taken with great care. What Melton means is that a search condition SC introduced as an assertion can never be attached to any of the referenced tables as a TCC without at least modifying SC. Only if SC is of the form

```
NOT EXISTS (SELECT * FROM T WHERE NOT SC')
```

for some table T it is possible to transform it into a TCC. If so, it necessarily has to be attached as CHECK SC' to T (and nowhere else!). Any other type of assertion cannot be turned into a TCC at all without changing its meaning! This applies in particular to any purely existential IC: This kind of integrity constraint can only be expressed in assertion form. Nowhere in the literature we checked has this fundamental insight been stated clearly and with emphasis on the full consequences. It follows, that Date's claim that everything which can be expressed as assertions can be expressed as TCCs as well, is simply wrong.

3.3 Handling Constraints During Schema Design

Tables not containing any data do not occur that frequently during the lifetime of a database, but they do occur regularly just after schema design has been completed and population of the database is to commence. The standard clearly demands that every IC has to be checked (or at least: has to be satisfied) after the execution of any SQL-statement, including CREATE SCHEMA statements. Thus, every IC has to be satisfied immediately after installing the initial schema of any application. The state over which this very first check is performed necessarily has to consist of tables which are all empty. A "nasty" (and non-trivial) consequence of this obvious observation is that no existential IC may ever be

part of a CREATE SCHEMA statement, as any such IC would be violated in the initial state of the database.

As we saw just before, such ICs can only be expressed in form of an assertion. Such assertions will have to be introduced by means of separate CREATE ASSERTION statements after all tables they reference existentially are no longer empty. This observation - not contained anywhere in literature either, as far as we know - has significant implications for the way how schema design has to be performed in SQL. Date observes similar problems in case of referential constraints introducing a "referential cycle", where two (or more) tables mutually reference each other in foreign key constraints. One of these referential constraints has to be added after constructing a consistent population of the database by means of an ALTER TABLE statement. Introducing a new form of statement for disabling and enabling integrity checking altogether (as offered, e.g., in Oracle) might help and avoid a cascade of follow-up assertion definitions.

Another issue arising during schema design is the proper choice of checking mode for both constraints and assertions. As mentioned above, the default mode is IMMEDIATE unless specified otherwise. This decision of the SQL designers was probably motivated by the intention to discover constraint violations as early as possible and to undo the effect of the violating statement only, rather than rolling back the entire transaction. The consequence of this decision to treat immediate and deferred ICs in a very different manner as far as their influence on transaction semantics is concerned clearly violates the well-established ACID paradigm. Atomicity is violated because the statements in an SQL transaction are either executed entirely, or partially (excluding those violating immediate ICs), or not at all (after violation of at least one deferred IC). Consistency, however, is retained as those statements violating immediate constraints are not executed. SQL users should at least be aware of this deviation from the ACID principle - books usually don't mention it.

An argument in favour of accepting this deviation is that partial rollback of individual statements can at least be traced by means of corresponding status parameters indicating which constraint violations occurred (and thus which parts of the transaction have not been executed), even though the transaction as a whole has been successfully committed. It would then be the responsibility of application programs to react appropriately to such irregular behaviour. This is certainly true from a pragmatic perspective, and things will certainly work well if applications are guaranteed to deal with this phenomenon in a reliable manner. However, atomicity as an principle of transaction handling remains violated, thus indicating the danger of this way of sharing responsibility between the DBMS and each individual application program. Similar, dangerous, arguments are often stated in favour of doing without integrity constraints and leaving consistency management to the application.

We mentioned referential cycles arising from mutual referential constraints between several tables earlier. But these are just special cases of a more general phenomenon of ICs depending on each other in such a way that they cannot be satisfied by individual modifications in separate transactions. Designers have to

be aware of every such "cluster" of ICs in order to correctly decide about the appropriate checking mode. Setting all ICs to IMMEDIATE will not work, as outlined above. Setting just one (or some) of the involved ICs to DEFERRED introduces restrictions on the order in which the tables involved have to be modified within a transaction: One ordering works, the other doesn't. Only choosing deferred mode for all constraints forming a "generalized referential cycle" prevents users from remembering which order of modification is required. Thus, the choice of default mode IMMEDIATE probably wasn't such a good idea at the end. Both problems related to the design of checking mode are not really fundamental, but SQL users ought to be aware of them, too.

4 Constraints and Views

View definitions in SQL3 have the following general structure:

```
CREATE [RECURSIVE] VIEW <view-name>(<list-of-column-names>)  
AS <query-expression>;
```

Due to the introduction of recursion - which is quite heavily restricted in SQL3 - the corresponding part of the standard (in particular 11.21) provides very heavy reading! Whether this is necessary or simply follows from the many restrictions imposed on recursion is another matter. In particular, recursion has to be linear (in a very elaborate sense). Only virtual views are provided; that is, no materialized views can be defined in SQL yet. In addition, it is not possible to define views without any columns, i.e., views of type *BOOL*. This is a consequence of the lack of Boolean queries in SQL throughout. We regret this omission because it means that a condition which may be introduced as an IC (and thus has to be true in every state) cannot be tested for satisfaction over some specific state (unless embedded into a select clause, the result of which is then "abused" as a substitute for a truth value).

4.1 Referencing Views in Integrity Constraints: Possible or Not?

Determining whether views may be referenced in ICs at all, provided another challenge when consulting the literature. If looking at the syntax of view definitions, a first obvious answer can be given: There is no counterpart to table constraints for views! No key or foreign key, not even a check constraint can be attached to a view. This came as a big surprise to us, as in our Datalog context things seemed to be quite different: An IC may reference any relation, a relation may be either a base relation or a derived relation - thus, ICs apply to views as well, due to the strict orthogonality of both features. The decision to exclude ICs attached to views, however, appears to be a conscious one in SQL, as triggers may not be attached to views either, but are restricted to base tables.

But what about references to views within assertions or within embedded selects in a TCC? In the very first sentence of section 4.17 (devoted to integrity constraints) the standard ([ANSI99]) states:

”Integrity constraints ... define the valid states of SQL data by constraining the values in the base tables.”(p. 48)

This seems to indicate that views cannot be constrained, even though the formulation is not ”by constraining base tables”. Several authors of textbooks remain equally vague by not excluding assertions on views explicitly, but by raising the impression that such restriction indeed exists by a very inconsequent use of the term *table*. The standard clearly defines the notion of a table as a generalization of various forms of ”SQL relations”, including base tables, temporary tables as well as viewed tables (a synonym for views). Thus, what applies to tables in general, ought to apply to views in particular.

In most books the terms ’table’ and ’base table’ are, however, used synonymously in many places. Even the standard is inconsequent when speaking of ’table constraint’, e.g., because this notion suggests a constraint attached to any table, but means a constraint attached to a base table only. We are ”guilty” too, as we didn’t point out that we used table as a synonym for base table throughout section 2, before turning to views. None of the secondary sources we found mentioned the possibility of referencing views in an assertion or in a check constraint - with one notable exception: Date. However, you have to read as far as Appendix D, p. 443, to find the following statement:

”It is strange that integrity constraints can refer to views but not be stated as part of a view definition (contrast the situation with respect to base tables)”

Inspecting the standard text afterwards again revealed that all syntax rules admit views in search conditions, and that no further restriction on tables referencable in ICs can be found. Just below Date’s citation you find a striking example of a unique constraint on a view column which can be legally expressed by a single-table (or: single-view) assertion, but may not be expressed as part of the view definition. Thus, we conclude that at least assertions on views are indeed expressible in SQL. Admitting this feature (which makes sense also in view of an orthogonal design of the language), it is consequent to admit references to views in embedded select clauses in TCCs either. Once more, the standard is not explicit about this: There is no place where the feature is positively addressed, and no restriction excluding it from the language.

4.2 Checking Integrity over Views

Even though the standard seems to require full checking of every IC after every modification of a database, it is not hard to understand that such a definition should not be taken too literally. What is required in practice is to guarantee that an IC is evaluated (over the state after a modification) in all those cases where the satisfaction of the respective IC depends on run-time data. In many cases, there are sufficient conditions for concluding that a particular SQL statement may never lead to a violation of a particular IC, regardless of the actual data in the DB. The most obvious such restriction is the law that only such modifications

may possibly violate an IC which affect tables referenced in the IC. Though this seems to be trivial on first sight, it is justified only because every well-formed SQL IC is safe (or: range-restricted).

Which modifications do affect a view? There are two kinds of update which may cause the contents of a view to change. First, there are so-called "view updates", modifications directly expressed against a particular view. Such modifications are permitted only if the respective view is updatable, i.e., if the defining query expression permits a systematic reduction of the view update request to a modification of exactly one of the underlying base tables. Obviously, every update directly addressing a view referenced in some IC might violate that very IC. After translating the view update request "down" to one underlying base table, the resulting modifications of this table may affect other ICs, which will have to be checked in turn.

However, there is a second kind of modification to be considered when checking ICs referencing views. The base tables on which a view depends may be modified explicitly (and independent from any view update). Each such modification possibly causes rows to appear in or to disappear from the respective view. In [BDM88] we called such implicit changes of views *induced updates*. The easiest way of dealing with induced updates during integrity checking is to check every IC referencing a view based on some table T, whenever T has been modified. In the next section, we will see that even more efficient incremental techniques for handling induced updates are around.

Our main message in this subsection is to point out that both, view updates and base table updates causing induced updates on views have to be considered when checking ICs on views. For "Datalog users" (at least for academic ones) this insight is quite basic, for SQL users, most of them not even aware that the standard permits ICs on views, this may be a new result, even though not particularly surprising once understood. Needless to point out that nothing about induced updates can be found in any of our SQL sources or in the standard, even though the issue already arises for (non-recursive) SQL2 views.

For space reasons, we do not address the interesting, but intricate question of check options on views in this paper. If a view has been defined with the check option activated, view updates "contradicting" the view definition (in a particular sense, not easy to state precisely) are rejected. Thus, similarities with ICs can be observed. Readers are referred to Date, section 13.4, on this matter.

5 Principles of Efficient Incremental Integrity Checking

In the previous section we already pointed out that integrity checking performed strictly in accordance with the policy prescribed by the standard would be unnecessarily expensive, as a lot of redundant evaluation steps were required. This is because only those modifications may possibly cause a constraint violation which directly or indirectly affect a given IC. Informally, the term "*to affect an IC*" means that the respective modification may potentially change tables directly or indirectly controlled by the respective constraint.

Meanwhile, there has been a wide range of publications on efficient integrity checking in relational databases, proposing approaches to overcome these deficiencies. Two pioneering papers, by Nicolas [Nic82] - originally presented already in 1979 - and Bernstein/Blaustein [BBC82], introduced a technique nowadays known as **incremental integrity checking**. The key idea is to determine already at schema design time which ICs can be possibly affected by which update patterns, and to associate a particular specialization of the original IC with each pattern. At transaction time, each actually occurring modification is matched with the precompiled update patterns. If no match is possible, the respective modification is guaranteed not to violate any IC. If matching is successful, the corresponding specialized constraint is instantiated by the actual parameters of the modification and finally evaluated over the current database state.

In many cases, evaluating the specialized constraints is magnitudes more efficient than evaluating the original constraint, because the specialization technique tries to focus as much as possible on just those rows in the affected tables actually changed by the modification under consideration. Thus, the term *incremental checking* is well-motivated. However, there are cases where checking the full IC is unavoidable. This happens most frequently in presence of leading existential quantifiers. Additional optimization techniques, such as memoing rows satisfying such existential constraints (so-called "witnesses" of the existence condition), can be used to achieve incremental effects in these situations as well.

We recall the basic principles of incremental checking here, because almost all of the many papers on this issue have been written in the context of some other relational language, such as Quel, Datalog, or relational algebra. Surprisingly enough there is hardly anything on incremental checking in the context of SQL! Most database textbooks do not address the issue at all, or just scratch the surface of this subject. Good survey literature on integrity checking is missing, too. Within the tight space limits of this paper we can't afford to fill this gap either, but we would like at least to direct the interest of readers to this area of research. We believe that the wide-spread prejudice "Constraints simply don't pay off, because integrity checking is too expensive!" is mainly due to a lack in knowledge about dedicated optimization techniques introduced decades ago. Let us briefly illustrate the effect of specialization in the context of the ubiquitous inclusion dependency:

```
CHECK (NOT EXISTS (SELECT * FROM t1 WHERE
                  NOT EXISTS (SELECT * FROM t2 WHERE a11 = a21)))
```

Deletions from t1 as well as insertions into t2 may not affect this constraint. Technically, such irrelevant modifications can be detected by inspecting the polarity of each table within the constraint. Table t1 in the outermost select clause occurs negatively, due to the NOT EXISTS in front of the SELECT, t2 occurs positively due to the double negation. Modifications with the same polarity as the occurrence of the respective table name are irrelevant, those with opposite polarity potentially affect the IC. This rule applies to insertions and deletions, whereas every UPDATE statement affecting one of the columns occurring in the

constraint may affect the constraint (due to its dual role of replacing the old and introducing a new value for the respective column).

In order to be able to express specialized constraints in SQL, one may, e.g., introduce system-generated so-called transition tables (or: delta tables) containing one row for each inserted, deleted, or updated row of a specific table. Let $t1^+$ denote, e.g., the delta table representing insertions into $t1$. The specialized version of the inclusion dependency with respect to $t1$ -insertions is obtained by simply replacing $t1$ in the FROM part of the outermost SELECT by $t1^+$. Evaluating this specialized version of the original IC usually leads to a huge gain in evaluation efficiency, as in almost all cases the number of newly inserted rows will be dramatically smaller than the overall number of rows in the table. Be aware, specialization isn't always that easy!

The notion of a delta table is well-established in almost every paper on efficient integrity checking as well as in the context of incremental maintenance of materialized views by now. It is not difficult in principle to translate techniques such as those briefly sketched here to the SQL context, even though the syntactic richness of the language makes such an exercise a bit tedious. In their well-known paper [CW90], Ceri and Widom apply a similar technique in the context of "SQL-like" assertions, expressing specialized ICs in the form of Starburst triggers. Triggers are indeed a very convenient means of expressing the association between update patterns and specialized constraints (forming the condition part of such triggers). However, using SQL triggers as defined in the new standard for this purpose poses various problems of control of the overall checking process which are not easy to overcome. Till now, to the best of our knowledge, there is no proper proposal of incremental, trigger-based integrity checking covering the full SQL constraint spectrum and being based on SQL triggers. Neumann/Müller [NM97] present an implementation of SQL assertions via Oracle triggers, but they appear not to use incremental techniques.

In order to extend incremental integrity checking to ICs over views, efficient computation of induced updates is required. Once all induced updates are known, incremental techniques as outlined above can be used for determining their effect on ICs referencing these views. Since the middle of the 1980s various extensions of Nicolas' and Bernstein/Blaustein's approach have been published: e.g., [Dec86,LST86,BDM88,Oli91], just to mention a few. Fortunately, these methods, often called update propagation methods, fit well with incremental integrity checking, as they make use of delta tables and delta views for representing induced updates, too. For an early survey of this field refer to [BMM92]. If applied to recursive views, many of the well-known approaches to incremental propagation of updates run into trouble due to unstratifiability of the resulting delta views. An in-depth treatment of this rather recent discovery and a proposal for a remedy can be found in [Gri97]. Again, nearly all of the relevant contributions to integrity checking in presence of views have been formulated in a Datalog or relational algebra context. A transfer to the "SQL world" has not yet been delivered, but is crucial for efficiently implementing integrity checking in forthcoming systems supporting the SQL standard in full.

6 Integrity in SQL: The Good, the Bad, and the Ugly

What did we learn about integrity in SQL? Which are the pitfalls of the seemingly well-established concepts we identified? And which recommendations for more controlled use of potentially harmful features we derived from our observations?

ICs in SQL are essentially Boolean expressions of the language (apart from the popular shorthands we omitted here). The standard doesn't impose any limitations on the type of expression to use, thus promoting orthogonality, however, there are numerous implicit wellformedness conditions which have to be discovered by each SQL user anew. Particularly implicit quantification conventions can be a hard burden as far as semantics are concerned, convenient as they are in syntactic shorthands. Deciding between assertions and TCCs as an appropriate means for expressing multi-table ICs is a major issue as far as the very liberal standard is concerned. The wide range of syntactical alternatives combined with the intricate conventions for implicitly quantifying TCCs caused quite some trouble in SQL textbooks. The main "discoveries" we made - to be judged from our "amateur" status at the start of our investigation - are the following:

- Multi-table references are possible in *both*, assertions and TCCs.
- Both forms of ICs are checked whenever *any* of the referenced tables changes.
- Semantics of a condition used as TCC is different from that of the same condition used as an assertion, due to the implicit extension of TCCs and due to bordercases caused by empty tables.

We recommend - like Melton - to limit embedded references to foreign tables in TCCs to those cases where the resulting gain in syntactic clarity is really stunning (or to self-references as in functional dependencies, e.g.) - use assertions otherwise! Excluding any ICs not initially satisfied in an empty database from any DB schema looks weird to us - we recommend an approach which postpones automatic IC checking to the phase in the DB lifecycle immediately following the initial population of a DB. Favoring immediate checking as a default mode might be convenient for simple ICs, but will be dangerous for more elaborate constraint sets: Atomicity of transactions is violated by immediate checking, and "logical clusters" of ICs requiring deferred checking are hard to identify. The decision not to provide TCC-like syntax for views is awkward, as it violates orthogonality, too. Literature is very unclear about views being referenced in ICs - the standard admits this useful feature quite clearly. This means additional challenge to users, however, as they do not only have to master the "cliffs" of view updating but have to understand the concept of induced update either, apparently not yet perceived by the SQL community. Incremental integrity checking and update propagation techniques developed during the last twenty years offer powerful support for this most general form of IC, waiting to being adapted to the SQL context, which is certainly feasible, but by no means easy. The close relationship of update propagation to materialized view maintenance - another urgently required extension of SQL - might advance the interest in such techniques.

Finally, we would like to express our concern about the fact that both, TCCs with embedded subqueries as well as assertions are still not supported by leading commercial products, thus also excluding any ICs over views. The powerful, classical feature of DB integrity is severely "crippled" by this decision, which we believe not to be justified by efficiency considerations any more. Integrity checking techniques are far enough advanced by now in order to provide optimal support for these features.

References

- [ANSI99] ANSI/ISO/IEC 9075-2-1999: *Database Languages - SQL - Part 2: Foundation*. ANSI, 1999, New York/N.Y., USA, <http://www.cssinfo.com/ncitsgate.html>.
- [BR86] BANCILHON, F., RAMAKRISHNAN, R.: *An Amateur's Introduction to Recursive Query Processing Strategies*. Proc. SIGMOD 1986, Washington D.C., USA: 16-52
- [BBG78] BEERI, C., BERNSTEIN, P.A., GOODMAN, N.: *A Sophisticate's Introduction to Database Normalization Theory*. Proc. VLDB 1978, Berlin, Germany: 113-124
- [BBC82] BERNSTEIN, P.A., BLAUSTEIN, B.T.: *Fast Methods for Testing Quantified Relational Calculus Assertions*. Proc. SIGMOD 1982, Orlando/Flor., USA: 39-50
- [BDM88] BRY, F., DECKER, H., MANTHEY, R.: *A Uniform Approach to Constraint Satisfaction and Constraint Satisfiability in Deductive Databases*. Proc. EDBT 1988, Venice, Italy, LNCS 303: 488-505.
- [BMM92] BRY, F., MANTHEY, R. MARTENS, B.: *Integrity Verification in Knowledge Bases*. Proc. 2nd Russian Conf. on Logic Programming, 1991, St. Petersburg, Russia, LNCS 592 (1992): 114-139.
- [Cel00] CELKO, J.: *SQL for Smarties: Advanced SQL Programming*. Morgan Kaufmann, San Francisco/CA, USA, ISBN 1-55860-576-2, 2000.
- [CW90] CERI, S., WIDOM, J.: *Deriving Production Rules for Incremental Constraint Maintenance*. Proc. VLDB 1990, Brisbane, Australia: 566-577.
- [Date97] DATE, C.J., DARWEN, H.: *A Guide to The SQL Standard*. 4. Edition, Addison-Wesley, Reading/Mass., USA, ISBN 0-201-96426-0, 1997.
- [Dec86] DECKER, H.: *Integrity Enforcement on Deductive Databases*. Proc. 1st Int. Conf. on Expert Database Systems, Charleston, USA, 1986: 381-393.
- [GP99] GULUTZAN, P., PELZER, T.: *SQL-99 Complete, Really*. R&D Books Miller Freeman Inc., Lawrence/Kansas, USA, ISBN 0-87930-568-1, 1999.
- [Gri97] GRIEFAHN, U.: *Reactive Model Computation - A Uniform Approach to the Implementation of Deductive Databases*. Dissertation, University of Bonn, 1997, <http://www.cs.uni-bonn.de/~ulrike/Publications/publications.html>.
- [GW99] GROFF, J.R., WEINBERG, P.N.: *SQL: The Complete Reference*. Osborne/Mc Graw Hill, Berkeley/CA, USA, ISBN 0-07-211845-8, 1999.
- [IBM99] IBM: *SQL-Referenz DB2 V6*. IBM Corp.; New York; SC26-8416; 1999; <ftp://ftp.software.ibm.com/ps/products/db2/info/vr6/htm/db2s0/index.htm>
- [LST86] LLOYD, J.W., SONENBERG, E.A., TOPOR, R.W.: *Integrity Constraint Checking in Stratified Databases*. Technical Report 86/5, Department of Computer Science, University of Melbourne, Australia, 1986.

- [Man94] MANTHEY, R.: *Integrity and recursion: two key issues in deductive databases*. D. Karagiannis (ed.): "Information Systems and Artificial Intelligence: Integration Aspects", LNAI 474 (1994): 104-126
- [MS93] MELTON, J., SIMON, A.R.: *Understanding the NEW SQL: A Complete Guide*. Morgan Kaufmann Publishers, San Francisco/CA, USA, ISBN 1-55860-245-3, 1993.
- [NM97] NEUMANN, K., MÜLLER, R.: *Implementierung von Assertions durch Oracle7-Trigger*. Informatik-Berichte, Nr. 97-02, TU Braunschweig, Germany, 1997: 1-16.
- [Nic82] NICOLAS, J.-M.: *Logic for Improving Integrity Checking in Relational Data Bases*. Acta Informatica, Vol. 18 Nr. 3, Springer-Verlag, 1982: 227-253.
- [Oli91] OLIVÉ, A.: *Integrity Constraints Checking In Deductive Databases*. Proc. VLDB 1991, Barcelona, Spain: 513-523.
- [Ora99] LORENTZ, D., OERTEL, D., ET.AL.: *Oracle8i SQL Reference, Release 8.1.5*. Oracle Corporation, Redwood City, USA, Part No. A67779-01, 1999.