

6 Constraints and Updating

6.1 Introduction

As in the theory book, this chapter deals with database constraints, not to be confused with type constraints (not supported in SQL) and SQL's so-called domain constraints discussed in Chapter 2.

In Chapter 1, Example 1.3, you saw a simple example of a database constraint declaration expressed in SQL, repeated here as Example 6.1 (though now referencing `IS_ENROLLED_ON` rather than `ENROLMENT`).

Example 6.1: Declaring an integrity constraint.

```
CREATE ASSERTION MAX_ENROLMENTS
    CHECK ( ( SELECT COUNT(*) FROM IS_ENROLLED_ON ) <= 20000 ) ;
```

`CREATE ASSERTION` is SQL's counterpart of **Tutorial D's** `CONSTRAINT`, but it is an optional conformance feature that first appeared in SQL:1992 and very few SQL implementations (at the time of writing in 2012) support it.

Without `CREATE ASSERTION`, one might attempt to implement the required constraint along the lines of Example 6.1a.

Example 6.1a: Alternative formulation for `MAX_ENROLMENTS`

```
ALTER TABLE IS_ENROLLED_ON
ADD CONSTRAINT MAX_ENROLMENTS
    CHECK ( ( SELECT COUNT(*) FROM IS_ENROLLED_ON ) <= 20000 ) ;
```

Explanation 6.1a

- **ALTER TABLE IS_ENROLLED_ON** announces that an alteration to the definition of base table `IS_ENROLLED_ON` is being specified.
- **ADD CONSTRAINT MAX_ENROLMENTS** states that the alteration in question is the addition of something SQL calls a *table constraint*, and its name is `MAX_ENROLMENTS`. A table constraint is a condition that is required to be satisfied by every row appearing in the base table for which that constraint is defined. Thus, in general, it is an open expression of the kind that can appear as the condition of a `WHERE` clause. In this example the condition is in fact a closed expression—it contains no reference to a column of `IS_ENROLLED_ON`—and thus if it is satisfied by one row of that table, then it is satisfied by all of them.
- The last line is exactly as written in Example 6.1, but there’s a subtle difference in meaning. Because a table constraint is one that must be satisfied by every row of the applicable table, it is always satisfied when the applicable table is empty—there is no row for which the constraint fails. In the case at hand, this is not a problem, because obviously, when `IS_ENROLLED_ON` is empty, then its cardinality—zero—does not exceed 20,000. However, suppose a constraint was required to the effect that `IS_ENROLLED_ON` must never be empty. That could be achieved by changing `<= 20000` to `> 0` in Example 6.1, but the same change to Example 6.1a would be ineffectual: when `IS_ENROLLED_ON` is empty, it contains no row that fails to satisfy the constraint. That is why SQL is incomplete with respect to database integrity when support for `CREATE ASSERTION` is absent.

Now, I wrote, “one might attempt to implement the required constraint” this way, suggesting that it might not be such a good idea after all. Even though it does have the desired effect in the example at hand, the fact that a table constraint is one that is required to be satisfied by each row in the relevant table means that the DBMS is very likely to evaluate it for each row that is added to or updated in the table, whereas of course it needs to be evaluated just once per update operation that affects `IS_ENROLLED_ON`. This is why **Tutorial D** has no counterpart of SQL’s table constraints. They provide a useful shorthand for certain special cases (like the `NOT NULL` constraint on a column, or a check for a column having nonnegative values only, for example) but they give rise to traps when used inappropriately. A favourite example is `CHECK (EXISTS (SELECT * FROM t))`, as a table constraint on table `t`. It is satisfied even when `t` is empty!

Unfortunately (or fortunately?), Example 6.1a is in any case somewhat hypothetical, because the condition contains a table expression—a subquery. The appearance of a subquery in a table constraint remains an optional conformance feature and to this day many implementations fail to support it—and of those that do support it, at least one that does so fails to enforce such constraints at all times.

One of my reviewers (Erwin Smout) showed me a possible workaround for use when the “no subqueries in table constraints” restriction is in force. Example 6.1b applies this “hack” to Example 6.1a.

Example 6.1b: Workaround for when subqueries not permitted in CHECK constraints

```
CREATE FUNCTION NO_MORE_THAN_20000_ENROLMENTS ( )
    RETURNS BOOLEAN ;
    RETURN ( SELECT COUNT(*) FROM IS_ENROLLED_ON ) <= 20000 ;

ALTER TABLE IS_ENROLLED_ON
ADD CONSTRAINT MAX_ENROLMENTS
CHECK ( NO_MORE_THAN_20000_ENROLMENTS( ) ) ;
```

Caveat lector: You are strongly advised to complete Exercise 2 in this chapter’s Exercises section before you commit to using this workaround in earnest.

“Not Enforced” Table Constraints

A constraint that is not enforced is not really a constraint within the meaning of the act, but SQL does have such a concept and it needs to be mentioned here. With the exception of UNIQUE and PRIMARY KEY specifications (see Section 6.4, the subsection headed **Keys**), a table constraint can be declared as either ENFORCED (the default option) or NOT ENFORCED. The “enforcement characteristic” of such a constraint can be changed by means of an ALTER TABLE statement. For example, the table constraint MAX_ENROLMENTS becomes not enforced by execution of the statement ALTER TABLE ALTER CONSTRAINT MAX_ENROLMENTS NOT ENFORCED. The immediate effect is the same (ignoring effects on the catalog) as if ALTER TABLE DROP CONSTRAINT MAX_ENROLMENTS had been given. However, it’s easier to reinstate the constraint using ALTER TABLE ALTER CONSTRAINT MAX_ENROLMENTS ENFORCED than to repeat the whole of Example 6.1a.

NOT ENFORCED cannot be specified for domain constraints or assertions, so use of CREATE ASSERTION should really be the preferred method of declaring a constraint, but unfortunately that option is not widely available.

Historical Note and Comments

[NOT] ENFORCED first arrived in SQL:2007. It remains an optional conformance feature. At first sight it seems to be rather a strange feature, but, assuming there is a genuine requirement for it, one could observe that the effect of switching a table constraint between ENFORCED and NOT ENFORCED can be obtained less conveniently by using ALTER TABLE/DROP CONSTRAINT and ALTER TABLE/ADD CONSTRAINT.

One possible motivation for this feature lies in performance considerations. The feasibility of addressing some integrity requirements decreases with database size and frequency of updates. Rather than leave the database exposed permanently to the possibility of becoming inconsistent, one could declare a “not enforced” constraint and switch it to being enforced at a convenient time. Of course, if the constraint is then found to be violated, then some ad hoc intervention will be needed to address the problem. This approach to maintaining integrity is clearly far from perfect but is perhaps better than nothing in circumstances that offer no practical alternative.

All of that having been said, why the feature is not available with key constraints and constraints declared by `CREATE ASSERTION` is a mystery to this writer.

It might seem that feature is restricted to constraints that have been explicitly named. However, the SQL standard specifies that an implementation-dependent constraint name is given by default. In that case, the unique name assigned by the system will show up in the catalog and could then be used in an `ALTER TABLE/ALTER CONSTRAINT` or `ALTER TABLE/DROP CONSTRAINT` statement.

I joined MITAS because
I wanted **real responsibility**

The Graduate Programme
for Engineers and Geoscientists
www.discovermitas.com

Month 16
I was a construction
supervisor in
the North Sea
advising and
helping foremen
solve problems

Real work
International opportunities
Three work placements







6.2 A Closer Look at Constraints and Consistency

Effects of NULL

Section 6.2 in the theory book starts with

A constraint is defined by a truth-valued expression, such as a comparison. A database constraint is defined by a truth-valued expression that references the database. To be precise, the expression defines a *condition* that must be *satisfied* by the database at all times.

and goes on to justify the use of the term *satisfied*. Unfortunately, SQL has two definitions of this term. A row satisfies a condition in a WHERE clause only when the condition evaluates to TRUE, but it satisfies a table constraint when the condition evaluates to either TRUE or UNKNOWN. Thus, `SELECT * FROM T WHERE c` might yield an empty table even when *c* is the condition specified in some table constraint for T and T itself is far from empty.

When Are Constraints Checked?

Under the model described in the theory book, constraints are conceptually checked at all *statement boundaries* (and only at statement boundaries). By default the same is true of SQL. However, SQL does not support the “multiple assignment” concept, described in the theory book, for database updates. For that reason it has to include an alternative method of addressing the problems that multiple assignment addresses. SQL does so by allowing the checking of specified constraints to be temporarily deferred and reinstated later—but never across a *transaction boundary*. As a result, it is possible for the database to appear to be inconsistent, but only to the user whose as yet uncommitted transaction has given rise to that state of affairs.

As a consequence of deferred constraint checking, SQL code that depends on consistency with declared constraints is obviously exposed to that assumption of consistency being false when the code is executed while checking is deferred. For example, the table expression `SELECT Name FROM IS_CALLED WHERE StudentId = 'S1'` might be expected never to result in a table containing more than one row, thanks to the key constraint applying to IS_CALLED; thus it might be used in a scalar subquery. However, if the checking of that key constraint is temporarily deferred and two or more rows with StudentId equal to 'S1' temporarily appear in that table, then the scalar subquery will give rise to a run-time exception. Fortunately, SQL does allow a constraint to be declared as `NOT DEFERRABLE`, and that is the default option.

Historical Note

Deferred constraint checking first arrived in SQL:1992. It remains an optional conformance feature.

6.3 Expressing Constraint Conditions

Use of Table Expressions

With the exception of key constraints, the examples in the theory book all explicitly reference at least one relvar and thus involve invocations of relational operators or aggregate operators. Assuming support for `CREATE ASSERTION`, we can always derive SQL counterparts of these examples using table expressions and truth-valued operators, but when that assumption does not hold we need to look for alternative solutions using table constraints. In most cases these will entail the use of subqueries and even that technique is prohibited by many implementations. In some cases special syntactic constructs are available, as we shall see, but there are several for which no SQL solution is available unless the implementation supports `CREATE ASSERTION` or subqueries in table constraints.

Now, the reason usually given for lack of support for subqueries in constraints is that in general such expressions can require the DBMS to examine the entire content of possibly very large tables. If database updates are expected to occur frequently—and are perhaps required to occur very frequently indeed—then declaration of such constraints would give rise to an intolerable slowing down of the updating process. Of course this is an extremely valid concern and we have to admit that integrity might occasionally have to be compromised for performance reasons, but consider the user with a small database that is subject to comparatively infrequent updating but nevertheless has strong integrity requirements. Might not such a user feel unfairly treated by a system that prohibits the declaration of required constraints? Defenders of the status quo respond to this argument by holding that language constructs that can give rise to disappointment for performance reasons, to such an extent as to militate against their use in common practical situations, should be banned. But sometimes users resort to implementing constraints, as best they can, in application code when they wish to enforce a constraint that is not supported by the DBMS but nevertheless does not adversely impair performance. The DBMS could almost certainly enforce such constraints much more efficiently *and much more reliably*. We can also point to various other SQL constructs that might be subject to similar concerns but are supported nonetheless. For example, if tables T1, T2, and T3 each contain 100,000 rows, then `SELECT * FROM T1, T2, T3`, when evaluated, delivers a table containing a quadrillion rows.

Procedural Constraint Enforcement (Triggers)

SQL has an alternative method of addressing database integrity, involving event-driven procedural code. The special procedures that can be used for this purpose are called *triggers* and the events that activate them are specified update operations. For example, suppose it is required for every row in `IS_CALLED` to have a matching row on `StudentId` in `IS_ENROLLED_ON`, enforcing a business rule to the effect that every registered student must be enrolled on at least one course. Then a triggered procedure might be activated every time `INSERT` is used to add a row to `IS_CALLED`, checking to see if a matching row exists in `IS_ENROLLED_ON` and raising an exception if there isn't one. But that wouldn't be sufficient to address the requirement. Further triggers would be needed, activated by `UPDATE` statements on `IS_CALLED` and `IS_ENROLLED_ON` that cause changes to `StudentId` values in either of those tables, and by `DELETE` statements on `IS_ENROLLED_ON`. As this simple example demonstrates, use of triggered procedures for constraint enforcement can be complicated and error-prone. As one practitioner told me, "It quickly gets *so* complicated that it's almost impossible for a human *not* to make errors..., and even when you're not facing a 'complicated' case, the work to be done is tedious and boring". The subject is beyond the scope of this book but is dealt with at length and in meticulous detail by the authors of reference [13].

ie business school

#1 EUROPEAN BUSINESS SCHOOL
FINANCIAL TIMES 2013

#gobeyond

MASTER IN MANAGEMENT

Because achieving your dreams is your greatest challenge. IE Business School's Master in Management taught in English, Spanish or bilingually, trains young high performance professionals at the beginning of their career through an innovative and stimulating program that will help them reach their full potential.

- Choose your area of specialization.
- Customize your master through the different options offered.
- Global Immersion Weeks in locations such as London, Silicon Valley or Shanghai.

Because you change, we change with you.

www.ie.edu/master-management | mim.admissions@ie.edu |

Download free eBooks at bookboon.com



Click on the ad to read more

Use of COUNT and NOT EXISTS

The theory book describes and discusses various general methods of expressing constraints, eventually noting that support for “=” with relation operands is sufficient for completeness. It also notes that every constraint can be expressed as an invocation of IS_EMPTY, where IS_EMPTY(r) is equivalent to $r\{\} = \text{TABLE_DUM}$. First, though, it gives Example 6.2, showing how to use COUNT to test a relation for emptiness. Example 6.2 here is a direct translation of that one into SQL.

Example 6.2: Testing for absence of counterexamples.

```
CREATE ASSERTION Must_be_enrolled_to_take_exam
  CHECK ( ( SELECT COUNT(*)
            FROM   EXAM_MARK
            WHERE  ( Student_Id, CourseId ) NOT IN
                  ( SELECT Student_Id, CourseId
                    FROM   IS_ENROLLED_ON ) )
          = 0 ) ;
```

Of course, counting all the rows is rather excessive when it is sufficient just to see if the table contains anything at all. Examples 6.3 and 6.3a illustrate the use of NOT EXISTS, SQL’s counterpart of Tutorial D’s IS_EMPTY operator. Example 6.3a shows how to express the constraint as a table constraint in case CREATE ASSERTION is not available but the system does support subqueries in table constraints. Notice how a table constraint avoids the need for the double negation that usually arises with tests for emptiness—instead of checking for the non-existence of a row that fails to satisfy a given condition, we give the inverse condition that every row must satisfy. For the sake of variety, Example 6.3 uses an invocation of EXCEPT in place of Example 6.2’s use of NOT IN.

Example 6.3: Use of NOT EXISTS

```
CREATE ASSERTION Must_be_enrolled_to_take_exam_alternative1
  CHECK (
    NOT EXISTS ( SELECT  StudentId, CourseId
                 FROM    EXAM_MARK
                 EXCEPT
                 SELECT  StudentId, CourseId
                 FROM    IS_ENROLLED_ON ) ) ;
```


Example 6.3a: Alternative formulation as a table constraint

```
ALTER TABLE EXAM_MARK
ADD CONSTRAINT Must_be_enrolled_to_take_exam_alternative2
CHECK ( EXISTS ( SELECT StudentId, CourseId
                  FROM IS_ENROLLED_ON
                  WHERE StudentId = EXAM_MARK.StudentId
                        AND CourseId = EXAM_MARK.CourseId )
      ) ;
```

In Example 6.3a, note the use of the table name, EXAM_MARK, as a range variable to qualify references to columns of that table. As always, the condition given as the operand of CHECK is one that would be legal as a WHERE condition following a FROM clause specifying just the table to which the constraint applies (viz., FROM EXAM_MARK in the case at hand).

Now, if the SQL implementation doesn't allow subqueries to appear in table constraints and doesn't support CREATE ASSERTION, then none of the formulations in Examples 6.2, 6.3, and 6.3a will be available. Happily, this particular constraint can be expressed as a foreign key constraint, as we shall see later in Section 6.4, the subsection headed **Foreign Keys**.

Example 6.4 is a translation into SQL of the corresponding example in the theory book, which is included there merely to show that for any scalar comparison there is an alternative formulation using IS_EMPTY.

Example 6.4: MAX_ENROLMENTS expressed using an invocation of NOT EXISTS

```
CREATE ASSERTION MAX_ENROLMENTS_alternative1
CHECK (NOT EXISTS (SELECT *
                  FROM (VALUES (SELECT COUNT(*)
                                FROM IS_ENROLLED_ON)) AS V(N)
                  WHERE V.N > 20000 ) ) ;
```

Explanation 6.4

- **VALUES (SELECT COUNT(*) FROM IS_ENROLLED_ON)** denotes a table with just one column, unnamed, in whose single row the value of that column is the number of rows in the current value of IS_ENROLLED_ON. The SELECT expression is parenthesized to make it into a scalar subquery and given as the argument to an invocation of VALUES, which makes the number denoted by that scalar subquery into a one-row, one-column table. (Actually, it might be safer to place an extra pair of parentheses around the SELECT expression here. Although **VALUES 1** and **VALUES (1)** are equivalent, it might not be clear as to which role the single parentheses are taking: do they denote a scalar subquery, as I have assumed, or are they the optional ones surrounding a single table expression? If the latter, we would expect a syntax error.)
- **AS V(N)** defines the range variable V to refer to what in this case is just the single row of that table, and also assigns the name N to its only column.
- **WHERE V.N > 20000** operates on that one-row, one-column table to yield a table of heading (N INTEGER) that is empty if and only if the single row in that one-row, one-column table fails to satisfy the condition $N > 20000$. Thus, the result is empty only when the number of enrolments is in fact no greater than the maximum allowed.



"I studied English for 16 years but...
...I finally learned to speak it in just six lessons"

Jane, Chinese architect

ENGLISH OUT THERE

Click to hear me talking before and after my unique course download



Use of Table Comparisons

Table comparisons are described in Chapter 5, Section 5.9, where it is noted that although table expressions cannot be compared, we have `TABLE (t)` to convert a table expression t into a value expression of type `ROW (r) MULTISSET`, where r is the row type of t . However, the only operator in SQL for comparing two multisets is “=”, so SQL has no direct counterparts of the theory book’s Examples 6.5, 6.6, and 6.7, which use “ \subseteq ”, and nor does this chapter. Those examples are shown merely to demonstrate that every constraint that can be expressed as an invocation of `IS_EMPTY` can be formulated alternatively as an invocation of “ \subseteq ”. If SQL were to have a counterpart of that operator, it would presumably have to be an “is submultiset of” operator, where $m1$ is a submultiset of $m2$ if and only if each element of $m1$ appears at least as many times in $m2$ as it does in $m1$. But SQL doesn’t have such an operator.

Effects of NULL

Here’s an important distinction between expressions denoting tables and expressions denoting multisets of rows: a table expression cannot evaluate to `NULL`, whereas a multiset expression can. Moreover, although a row expression can evaluate to `NULL`—for instance, `CAST (NULL AS ROW (X INTEGER, Y INTEGER))` is a legal expression—`NULL` cannot appear as an element of the body of a table. Every element of the body of a table is indeed a row. However, if a table has a column whose declared type is a row type, then `NULL` might appear in place of a value for that column in some row of that table.

It follows from the foregoing discussion that although a multiset expression in general can evaluate to `NULL`, an invocation of `TABLE` will never do so (recall that, counterintuitively, `TABLE` operates on a table and returns a multiset of rows). Nor can `NULL` ever appear as an element of a multiset resulting from an invocation of `TABLE`. However, the comparison `TABLE (t1) = TABLE (t2)`, where $t1$ and $t2$ are equal in cardinality, evaluates to `UNKNOWN` whenever `NULL` appears at any level of nesting within either of $t1$ or $t2$.

Use of Truth-Valued Aggregate Operators

Example 6.8 in the theory book is an awkward one using double negation, offered as motivation for the neater way of expressing such constraints subsequently shown in Example 6.9. Example 6.8 reads as demanding that no exam mark shall not be in the required range, whereas Example 6.9 reads, more naturally, as requiring that every mark shall be in that range. Here, the SQL translations illustrate SQL’s `BETWEEN` and `NOT BETWEEN` shorthands for in-range tests.

Example 6.8: Restricting exam marks to between 0 and 100

```
CREATE ASSERTION Marks_between_0_and_100
CHECK ( NOT EXISTS ( SELECT *
                     FROM EXAM_MARK
                     WHERE Mark NOT BETWEEN 0 AND 100 ) ) ;
```

As mentioned in Chapter 5, SQL has EVERY as its counterpart of **Tutorial D**'s aggregate operator AND.

Example 6.9: Restricting exam marks to between 0 and 100 using EVERY

```
CREATE ASSERTION Marks_between_0_and_100_using EVERY
CHECK ( ( SELECT EVERY ( Mark BETWEEN 0 AND 100 )
        FROM EXAM_MARK ) ) ;
```

x BETWEEN y AND z is equivalent to $x \geq y$ AND $x \leq z$.

It follows from Example 6.9 that if the SQL standard's CREATE ASSERTION and type BOOLEAN are both supported, then use of EVERY provides an alternative method of testing a table for being empty. If tx is a table expression, then we have the scalar subquery (SELECT EVERY(FALSE) FROM (tx) AS T). When the result of tx contains a row, that row clearly fails to satisfy the condition FALSE and so the result of the scalar subquery is FALSE; otherwise the table is empty and the result is UNKNOWN, in which case the constraint is deemed to be satisfied, as previously explained. The reason why the result is UNKNOWN instead of what it should correctly be, viz. TRUE, is explained in **Effects of NULL**.

Effects of NULL

Let $aggop(x)$ be an invocation of some aggregate operator $aggop$ in SQL, where x is an expression (usually an open expression) to be evaluated against each row of the table t determined by the context in which the invocation appears. Then $aggop$ considers only those rows that satisfy the condition x IS NOT NULL. It follows that if $aggop$ is EVERY or SOME and x evaluates to TRUE or FALSE for at least one row of t , then the result is either TRUE or FALSE, never UNKNOWN. However, if x evaluates to UNKNOWN for every row of t (which is true in the particular case when t is empty), then SQL's other general rule kicks in, requiring the result to be NULL, which is equivalent to UNKNOWN when it appears in the place of a BOOLEAN value. That anomaly is to some extent compensated for, when EVERY is used in constraint declarations, by SQL's rule that a constraint is deemed to be satisfied when it evaluates to UNKNOWN. However, (SELECT SOME(TRUE) FROM (tx) AS T) is not reliable as an existence test because it evaluates to UNKNOWN if the result of tx is empty, when a constraint based on that condition would be deemed satisfied. That problem could be addressed by writing COALESCE((SELECT SOME(TRUE) FROM (tx) AS T), FALSE) or, equivalently, (SELECT SOME(TRUE) FROM (tx) AS T) IS TRUE (see Chapter 3, Section 3.5 **Deriving Predicates from Predicates**, Figure 3.1a in the subsection headed *Other monadics*).

6.4 Useful Shorthands for Expressing Some Constraints

Section 6.4 in the theory book describes three special classes of constraint and shorthands that have been proposed for them, not all of which have been adopted in **Tutorial D**. The three special classes are tuple constraints, key constraints, and foreign key constraints. SQL has counterparts of all three, as different kinds of table constraints.

CHECK Constraints

A CHECK constraint is a table constraint defined using the key word CHECK, as already illustrated in several examples in this chapter. In particular, a CHECK constraint can be used to express a constraint such as the one shown in Example 6.10, referred to in the theory book as a tuple constraint (so one might call it a row constraint in SQL). This is clearly the way most SQL users would prefer to express such a constraint—in fact, it is the only way when Examples 6.8 and 6.9 are unavailable for want of support for subqueries in constraints.

Example 6.10: Shorthand for a row constraint

```
ALTER TABLE EXAM_MARK
  ADD CONSTRAINT Mark_in_range
  CHECK ( Mark BETWEEN 0 AND 100 ) ;
```

Excellent Economics and Business programmes at:



university of
 groningen




“The perfect start
 of a successful,
 international career.”

CLICK HERE
 to discover why both socially
 and academically the University
 of Groningen is one of the best
 places for a student to be

www.rug.nl/feb/education



Actually, it's not *quite* the only way. The constraint can be included in the column definition for `Mark` in the `CREATE TABLE` statement for `EXAM_MARK`, as shown in Example 6.10a. A constraint declared as part of a column definition is called a *column constraint*.

Example 6.10a: Column constraints included in a column definition

```
Mark INTEGER NOT NULL CHECK ( Mark BETWEEN 0 AND 100 ),
```

Example 6.10a has two separately declared column constraints in the same column definition. Both are unnamed but could be named if desired. Constraints that are declared without names acquire implementation-dependent names that show up in the database catalog. As in **Tutorial D**, naming a constraint allows it to be dropped when no longer needed (SQL uses the same key word, `DROP`).

The first column constraint, `NOT NULL`, is short for `CHECK (Mark IS NOT NULL)`. One might conclude that `BETWEEN 0 AND 100` would be allowed as short for `CHECK (Mark BETWEEN 0 AND 100)`, but that would be a wrong conclusion. One might also wonder if `CHECK (Mark BETWEEN 0 AND 100)` could be included (perversely) in the definition of some column other than `Mark`. In fact it can. Moreover, a column constraint can reference other columns in the same table, in which case the choice as to which column definition to include it in becomes arbitrary and one might prefer to write it as a regular table constraint (at the expense of an extra key word and a comma).

Effect of `NULL`

Until SQL:1999, if a column was subject to a `NOT NULL` constraint, then every value v appearing in that column could be guaranteed to compare equal with itself and not equal to every value of its type other than itself. That guarantee does not hold with all the additional types that were added to SQL in SQL:1999. For example, if a column is defined on type `ROW (x INTEGER, y INTEGER)`, then a `NOT NULL` constraint will not prevent the value `ROW (CAST (NULL AS INTEGER), 42)` appearing in that column. A similar comment applies to user-defined structured types, where the value of a component of the structure being `NULL` does not confer “nullness”, so to speak, on the whole value (see Chapter 2, Section 2.10, **Types and Representations**, the subsection **Effect of `NULL`**).

Keys

We have already seen one way of declaring a key in SQL, in `CREATE TABLE` statements. For example, the one for `EXAM_MARK` in the introduction to Chapter 5 includes the table constraint `PRIMARY KEY (StudentId, CourseId)`. This is almost equivalent to **Tutorial D**'s `KEY { StudentId, CourseId }`, the exceptions being: (a) there is some significance to the order in which the column names are written, as explained in the following section on foreign keys, and (b), as the key words `PRIMARY KEY` suggest, no more than one primary key can be specified for the same base table.

A `PRIMARY KEY` specification carries an implicit `NOT NULL` constraint on each column of the specified key. When more than one key constraint is required, the key word `UNIQUE` must be used in place of `PRIMARY KEY` for all or all but one of them. A `UNIQUE` specification does *not* carry an implicit `NOT NULL` constraint on each column of the specified key (says the SQL standard, though I am aware of at least one SQL implementation where it does).

Whether declared using `PRIMARY KEY` or `UNIQUE`, at least one column must be specified. SQL has no direct counterpart of **Tutorial D**'s `KEY { }`.

When a key consists of just one column it may be expressed in shorthand as a column constraint. For example, in the `CREATE TABLE` statement for `COURSE`, the primary key could be specified by adding `PRIMARY KEY` to the column definition for `CourseId`.

SQL differs from **Tutorial D** in its support for keys in the following respects:

- SQL does not require at least one key for every base table. In **Tutorial D**, if no key is explicitly declared, then `KEY { ALL BUT }` is implicit.
- When no key is specified there is no prohibition on multiple appearances of the same row.
- SQL does not recognize the empty set as a key.
- SQL allows a key to be a proper superset of another key for the same base table. (This “feature” is sometimes used as a workaround for the fact that the columns of the foreign key are required to correspond to those of a declared key of the referenced table.)

Effects of `NULL`

When a `UNIQUE` specification u for base table t includes a column c that is not subject to a `NOT NULL` constraint, the appearance of several rows having `NULL` in place of a value for c and equal values for the other columns specified in u is permitted. It is only when each column of the specified “key” has a value that those column values may not appear in the same combination in more than one row of t .

WHEN/THEN Key Constraints

Temporal databases are beyond the scope of the theory book, but the problems that arise with them and proposed solutions to those problems are described in detail in reference [12], which presents its proposals as notional extensions to **Tutorial D**. One of these extensions is a special shorthand called a `WHEN/THEN` constraint, and SQL has a somewhat similar solution to the particular problem addressed by such constraints (though it falls far short of addressing all of the problems described in reference [12]).

Suppose a table has two columns representing a period of time throughout which the information conveyed by the other columns is recorded as having been the case. A salary history table for employees, with columns `From` and `To` for dates defining the applicable time periods, would be a good example. A constraint is needed to avoid the possibility of an employee being shown as having two different salaries on the same day, which could happen if two rows for the same employee have overlapping periods indicated by their `From` and `To` dates. The term “WHEN/THEN constraint” appeals to the notion of “unpacking” the table so that each row is replaced by one or more rows, one for each date contained in its from-to period: *when* the relation is unpacked, *then* the given key constraint (e.g., on employee number and date) is to hold.

Here’s a concrete example showing how SQL supports WHEN/THEN constraints.

```
CREATE TABLE SAL_HISTORY ( EmpNo CHAR(6),
                           Salary INTEGER NOT NULL,
                           From DATE
                           To DATE
                           PERIOD FOR During ( From, To ),
                           PRIMARY KEY ( EmpNo, During WITHOUT OVERLAPS )
                           ) ;
```

LIGS University

based in Hawaii, USA

is currently enrolling in the
Interactive Online **BBA, MBA, MSc,**
DBA and PhD programs:

- ▶ enroll **by October 31st, 2014** and
- ▶ **save up to 11%** on the tuition!
- ▶ pay in 10 installments / 2 years
- ▶ Interactive **Online** education
- ▶ visit www.ligsuniversity.com to find out more!

Note: LIGS University is not accredited by any nationally recognized accrediting agency listed by the US Secretary of Education. More info [here](#).





The `PERIOD FOR` specification states that the `From` and `To` values in each row denote a time interval (called a period because SQL uses the term “interval” for something else). The `From` values are treated as closed bounds, the `To` values as open bounds, so a given row in `SAL_HISTORY` indicates that an employee was paid a certain salary from the given `From` date up to *but not including* the given `To` date. The specification implies the column constraint `NOT NULL NOT DEFERRABLE ENFORCED` for each of columns `From` and `To`.

During `WITHOUT OVERLAPS`, which, if required, must appear as the last element of the key, specifies that if the same `EmpNo` value appears in two distinct rows of `SAL_HISTORY`, then the `From` and `To` values in those rows must denote `During` periods that do not overlap (have no date in common).

Historical Notes and Comments

Support for keys (and foreign keys) arrived in 1989, as part of an addendum to SQL:1987, the first international edition of the SQL standard.

`PERIOD FOR` and `WITHOUT OVERLAPS` arrived as an optional conformance feature in SQL:2011. Note that although the `WITHOUT OVERLAPS` specification in `SAL_HISTORY` prevents an employee from being recorded as having two or more different salaries on the same day, it does not enforce the “packed form” defined in reference [12]. In the given example, packed form would prevent the appearance of two or more rows with consecutive `From-To` periods showing the same salary for the same employee—a case of what reference [12] calls *circumlocution*. Clearly, two such rows can be replaced by a single row having the `From` value of the earlier period and the `To` value of the later one.

A question arises as to what happens if one of those implied `NOT NULL` constraints is dropped or altered to be `NOT ENFORCED` (the historical note in Section 6.1 shows how this might be done in accordance with the SQL standard). SQL:2011 is silent on that possibility.

Foreign Keys

(See the corresponding section in the theory book for the meaning of this term.)

Examples 6.2, 6.3, and 6.3a are alternative ways of formulating a constraint that enforces a business rule to the effect that every student who takes an exam must be enrolled on the applicable course. As it happens, that constraint can also be formulated as a foreign key, expressed as a table constraint for base table `EXAM_MARK`.

Example 6.3b: Alternative formulation for 6.3 as a foreign key constraint

```
ALTER TABLE EXAM_MARK
  ADD CONSTRAINT Must_be_enrolled_to_take_exam_alternative3
  FOREIGN KEY ( StudentId, CourseId )
  REFERENCES IS_ENROLLED_ON ;
```

The formulation in Example 6.3b is available only because the following conditions hold:

1. There is a one-to-one correspondence from the specified columns, in the specified order, to those of the primary key of `IS_ENROLLED_ON`. Corresponding columns do not have to have the same name but they must be of the same declared type. The table name for the *referenced table* (`IS_ENROLLED_ON` in the example) can be followed by a commalist of column names in parentheses, in which case that commalist—the *referenced columns*—must correspond exactly, in the correct order, to some key specified for the referenced table. The referenced columns must be explicitly specified when the applicable key is declared using `UNIQUE` rather than `PRIMARY KEY`, or when it is declared using `WITHOUT OVERLAPS`.
2. The referenced table and the referencing table (`EXAM_MARK` in the example) are both base tables.

A foreign key declaration in SQL can include a specification of a *compensatory action*, which defines an additional update to take place automatically when the constraint would otherwise be violated. For example, the specification `ON DELETE CASCADE`, when added to the foreign key declaration in Example 6.3b, states that when a row is deleted from the referenced table, `IS_ENROLLED_ON`, all matching rows in `EXAM_MARK` are to be deleted too. Similarly, `ON UPDATE CASCADE` specifies that when the `StudentId` or `CourseId` value of some row in `IS_ENROLLED_ON` is updated, the new value is propagated to all of the matching rows in `EXAM_MARK`. For another example, `ON DELETE SET DEFAULT` specifies that when a row in `IS_ENROLLED_ON` is deleted, the values for columns `StudentId` and `CourseId` in the matching rows of `EXAM_MARK` are replaced by the default values for those columns—in which case those default values must be sure to be matched by some row in the referenced table, of course.

A compensatory action, being a further update of some kind, might in turn result in violation of a foreign key constraint that might in turn have a compensatory action defined for it. The interactions between compensatory actions and triggered procedures are fully specified in the SQL standard but can be bewilderingly complicated.


When no compensatory action is required, SQL has two ways of dealing with foreign key constraint violations and allows the user to choose between the two. The two options are `NO ACTION` and `RESTRICT`. `ON DELETE NO ACTION` and `ON UPDATE NO ACTION` are self-explanatory: a constraint violation is to cause the delete or update to be rejected. But `RESTRICT`, rather than `NO ACTION` is the default option. `ON DELETE RESTRICT` and `ON UPDATE RESTRICT` can cause a delete or update to be rejected even before the overall effect of the statement has been evaluated and even when the overall effect would be accepted under `NO ACTION`. For example, suppose the table `T` is subject to the constraint

```
FOREIGN KEY (FK) REFERENCES T(K) ON UPDATE RESTRICT
```

and the following statement is executed:

```
UPDATE T SET K = K + 1 ;
```

If `T` has a row with 3 for column `K` and one or more rows with 3 for `FK`, the update is rejected even if `T` also has a row with 2 for `K` that will satisfy the foreign key constraint when 3 replaces 2. `ON UPDATE NO ACTION` had been specified instead, the update would be accepted because the overall effect would not cause a constraint violation—the constraint is properly checked at the statement boundary instead of being checked against some intermediate state that arises mid-execution.

.....Alcatel-Lucent 

www.alcatel-lucent.com/careers

What if you could build your future and create the future?

One generation's transformation is the next's status quo. In the near future, people may soon think it's strange that devices ever had to be "plugged in." To obtain that status, there needs to be "The Shift".



If the applicable key of the referenced table is defined using a period name and `WITHOUT OVERLAPS`, then the definition of the referencing table must include a period name pn defined on columns compatible with the corresponding ones of the referenced table, and the foreign key declaration must include `PERIOD pn`. The foreign key constraint is then considered to apply to the unpacked forms of the referencing and referenced tables, and the `ON DELETE/ON UPDATE` options are not supported—`RESTRICT` is implicit.

6.5 Updating Tables

Section 6.5 in the theory book is headed **Updating Relvars**. I could perhaps have used the heading **Updating Table Variables** here but such terminology is not used in SQL. Nevertheless, of course it is table variables—base tables or updatable views—that are updated, not tables *per se*. As the theory book does not cover the difficult and somewhat controversial topic of updating virtual relvars (as views are called in that book), this book likewise considers only base tables as targets. Also omitted, as in the theory book, is any discussion of SQL's comprehensive provisions for security and authorization, giving control over (among many other things) which users are authorized to do what kinds of updating to which tables.

The theory book introduces the topic of updating by describing the assignment operator, “:=” in **Tutorial D**. SQL uses a different syntax for assignment, using the key word `SET` and “=”. Thus, to add 1, so to speak, to the integer variable x , SQL has `SET x = x + 1`. However, the operator is not supported at all for tables, so SQL has no direct counterpart of the theory book's Example 6.11. It does, however, have counterparts of **Tutorial D**'s `INSERT`, `UPDATE`, and `DELETE` operators, which we can deal with here quite briefly by giving translations to SQL of the theory book's examples 6.12, 6.13, 6.14, and 6.16. (Example 6.15 is missing because that one uses “:=”.)

INSERT

Loosely speaking, `INSERT` takes the rows of a given *source* table and adds them to the specified *target* table, retaining all the existing rows in the target. Example 6.12 shows how `INSERT` can be used to add a single row to `IS_ENROLLED_ON`.

Example 6.12: Enrolling a student on a course using `INSERT`

```
INSERT INTO IS_ENROLLED_ON VALUES ('S3', 'C2') ;
```

Recall that `VALUES ('S3', 'C2')` denotes the table consisting of just the row `('S3', 'C2')`. If that row already exists in the target table, then the update has the effect of increasing the number of appearances of that row by one, unless some key is specified for that table (as is the case with `IS_ENROLLED_ON`), in which case the update fails.

Recall also that the columns of the result of a `VALUES` expression are effectively unnamed, so the column ordering has to be used to determine the correspondence between source and target columns. In Example 6.12, 'S3' becomes the `StudentId` value in the inserted row and 'C2' becomes the `CourseId` value, and that's because `StudentId` is defined to be the first column of `IS_ENROLLED_ON`, `CourseId` the second. However, the defined ordering can be explicitly overridden, as shown in Example 6.12a.

Example 6.12a: Overriding the defined column ordering

```
INSERT INTO IS_ENROLLED_ON (CourseId, StudentId)
VALUES ('C2', 'S3') ;
```

A `VALUES` expression is not restricted to tables of just one row. For example, the source table `VALUES ('S3', 'C2'), ('S4', 'C1')` would simultaneously enroll student S3 on course C2 and S4 on C1. In general, any table expression can be used as the source table for an `INSERT` invocation, just as any relational expression can be used for the same purpose in **Tutorial D**.

Example 6.13, like its counterpart in the theory book, illustrates the convenience of allowing any table expression to be the source for an `INSERT`. It assumes that all the exam scripts submitted by students have been marked and it has been decided to record marks of zero for students who failed to turn up for an exam they should have sat. (Remember that SQL's `EXCEPT` requires its operands to be of the same degree, unlike **Tutorial D**'s `NOT MATCHING`—hence the third element, 0, of the second operand in the example.)

Example 6.13: Awarding zero marks to students who failed to take the exam

```
INSERT INTO EXAM_MARK
SELECT StudentId, CourseId, 0
FROM IS_ENROLLED_ON
EXCEPT
SELECT StudentId, CourseId, 0
FROM EXAM_MARK ;
```

UPDATE

Loosely speaking, `UPDATE` changes some of the column values of some existing rows of its target table. Thus, although some rows disappear from the target and others arrive in it, so to speak, the cardinality of the table does not change. Suppose the exam board for course C2 decides that the exam has been marked too harshly and everybody's mark is to be increased by 5. Example 6.14 shows how.

Example 6.14: Adding 5 to all the marks for course C2

```
UPDATE EXAM_MARK SET Mark = Mark + 5
WHERE CourseId = 'C2' ;
```

The syntax is self-explanatory. The `WHERE` specification is optional and defaults to `WHERE TRUE`, meaning that the specified changes are to be applied to all existing rows in the target table. The expression `Mark = Mark + 5` is a *column assignment*. When several column assignments are needed they are separated by commas and the semantics of multiple assignment as described in the theory book apply: the right-hand sides are all evaluated before any column assignments are performed. The same column cannot be the target or more than one assignment.

SQL's `UPDATE` differs from **Tutorial D's** in the following interesting respect. Let relvar rv be assigned the relation `RELATION { TUPLE { X 1, Y 2}, TUPLE (X 2, Y 2) }`. Then `UPDATE rv WHERE X = 1 (X := 2)` causes rv to consist of just a single tuple, `TUPLE { X 2, Y 2 }`. The SQL counterpart, assuming no constraint violation would arise, causes the target table to contain two appearances of the row `ROW (2, 2)`.

Maastricht University *Leading in Learning!*

Join the best at the Maastricht University School of Business and Economics!

Top master's programmes

- 33rd place Financial Times worldwide ranking: MSc International Business
- 1st place: MSc International Business
- 1st place: MSc Financial Economics
- 2nd place: MSc Management of Learning
- 2nd place: MSc Economics
- 2nd place: MSc Econometrics and Operations Research
- 2nd place: MSc Global Supply Chain Management and Change

Sources: Keuzegids Master ranking 2013; Elsevier 'Beste Studies' ranking 2012; Financial Times Global Masters in Management ranking 2012

Maastricht University is the best specialist university in the Netherlands (Elsevier)

Visit us and find out why we are the best!
Master's Open Day: 22 February 2014

www.mastersopenday.nl



If the target table has a period name defined on two of its columns, then an UPDATE statement for that table can include a FOR PORTION OF clause, specifying a FROM and a TO value. In this case the cardinality of the target table *can* change. For example, if the SAL_HISTORY table contains the following rows:

```
( '123456', 55000, DATE('2011-09-01'), DATE('2012-08-01') ),
( '123456', 60000, DATE('2012-08-01'), DATE('9999-12-31') )
```

and it is discovered that employee 123456's salary was in fact increased to 60000 on July 1st, 2012, then Example 6.14a can be used to make the necessary correction. As a result, those two rows will be replaced by the following three:

```
( '123456', 55000, DATE('2011-09-01'), DATE('2012-07-01') ),
( '123456', 60000, DATE('2012-07-01'), DATE('2012-08-01') ),
( '123456', 60000, DATE('2012-08-01'), DATE('9999-12-31') )
```

Note that the second and third exhibit circumlocution: using more than one row to state what could equivalently be stated by a single row showing that employee 123456's salary is 60000 from July 1st, 2012 until SQL's rather pessimistic estimate of the end of time (this being what is sometimes used to indicate "indefinitely").

Example 6.14a: Updating a "portion" of the salary history table

```
UPDATE SAL_HISTORY
FOR PORTION OF During
          FROM DATE('2012-07-01') TO DATE('2012-08-01')
SET ( Salary = 60000 )
WHERE EmpNo = '123456' ;
```

DELETE

Loosely speaking, DELETE removes some existing rows from its target table. Suppose the university decides that course C3 is to be withdrawn. Example 6.16 shows how.

Example 6.16: Withdrawing course C3, using DELETE

```
DELETE FROM COURSE WHERE CourseId = 'C3' ;
```

Every row that satisfies the given WHERE condition is deleted; rows that do not satisfy it remain in place.

As with UPDATE, a FOR PORTION OF clause can be specified if the target table has a defined period name, as illustrated in Example 6.16a.

Example 6.16a: Deleting a “portion” of the salary history table

```
DELETE SAL_HISTORY
FOR PORTION OF During
           FROM DATE('2012-01-01') TO DATE('2012-02-01')
WHERE EmpNo = '123456' ;
```

As a result, the row

```
('123456', 55000, DATE('2011-09-01'), DATE('2012-08-01'))
```

is replaced by the two rows

```
('123456', 55000, DATE('2011-09-01'), DATE('2012-01-01')),
('123456', 55000, DATE('2012-02-01'), DATE('2012-08-01'))
```

and the DELETE statement will have effected an increase in cardinality instead of the usual decrease.

MERGE and TRUNCATE

SQL has two more table update operators, MERGE and TRUNCATE.

MERGE, like INSERT, takes a source table *s* and uses it to update a target table *t*. Briefly, a MERGE statement specifies a matching condition to determine which rows of *s* have at least one matching row in *t* (under that specified matching condition). It then specifies an open-ended series of conditions to be applied to each row of *s* paired with actions to be applied on *t*. WHEN MATCHED AND *c1* THEN *x1* specifies that action *x1*, necessarily an UPDATE or DELETE, is to be applied on *t* for each matching row in *s* that satisfies the condition *c1*. WHEN NOT MATCHED AND *c2* THEN *x2* specifies that action *x2*, necessarily an INSERT, is to be applied on *t* for each non-matching row in *s* that satisfies the condition *c2*.

The curiously named TRUNCATE statement deletes all the rows from its specified target, bypassing any triggered actions, including compensatory actions, specified for that target. The target must be a base table.

Multiple Assignment

SQL supports multiple assignment to local variables and also applies multiple assignment semantics in SET clauses of UPDATE statements, but does not support multiple assignment in connection with updates on table targets. Thus, SQL has no counterpart to the theory book’s Example 6.17, simultaneously deleting from both COURSE and IS_ENROLLED_ON. If we assume that there must be at least one enrolment for each course, and that students can enroll only on existing courses, deferred constraint checking has to be used, as shown in Example 6.17 here.

Example 6.17: Withdrawing course C3 and deleting any enrolments on C3

Assume the definition of IS_ENROLLED_ON includes

```
CONSTRAINT Course_must_exist_for_enrolment
FOREIGN KEY ( CourseId ) REFERENCES COURSE ON DELETE NO ACTION
and the definition of COURSE includes
```

```
CONSTRAINT Enrolment_must_exist_for_course
CHECK ( CourseId IN ( SELECT CourseId FROM IS_ENROLLED_ON )
```

Then the desired effect can be achieved by this:

```
SET CONSTRAINTS Course_must_exist_for_enrolment DEFERRED ;
DELETE FROM COURSE WHERE CourseId = 'C3' ;
DELETE FROM IS_ENROLLED_ON WHERE CourseId = 'C3' ;
SET CONSTRAINTS Course_must_exist_for_enrolment IMMEDIATE ;
```

ON DELETE NO ACTION states that no compensatory action is to be used for enforcement of the foreign key constraint. Deferring the checking of that constraint allows the first DELETE statement to succeed in spite of the consequent existence of “orphan” rows in IS_ENROLLED_ON. Cancelling the deferment immediately after the second delete then causes the constraint to be checked. If the DELETE statements were the other way around, then we would have to defer Enrolment_must_exist_for_course instead.



> Apply now

REDEFINE YOUR FUTURE
**AXA GLOBAL GRADUATE
PROGRAM 2015**

redefining / standards 

agence.cdg © Photomistop



In Example 6.18, a straight translation of its counterpart in the theory book, the second statement, assigning to both X and Y, illustrates multiple assignment to local variables in SQL.

Example 6.18: A consequence of simultaneity

```
SET X = 1;
SET ( X, Y ) = ROW ( X + 1, X + 1 );
```

As in **Tutorial D**, the value 2 is assigned to both X and Y. The simultaneous assignment to X and Y can also be expressed using a `SELECT ... INTO` statement, as shown in Example 6.18a. An `INTO` clause can be used only in the first `SELECT` clause of such a statement and only when the resulting table contains no more than one row. (When a `SELECT` expression contains further `SELECT` expressions, the first `SELECT` clause is the one belonging to the outermost `SELECT` expression. The outermost `SELECT` expression cannot be combined with another by `UNION`, `EXCEPT`, or `INTERSECT`.)

Example 6.18a: Multiple assignment using `SELECT ... INTO`

```
SELECT * INTO X, Y
FROM VALUES ( X + 1, X + 1 ) AS T;
```

The key word `ROW` in Example 6.18 is optional. That being the case, you might ponder the distinctions among the statements listed in Example 6.18b. Which ones assign 1 to X and which assign `ROW (1)`? Is (b) legal if the declared type of X is `ROW (F1 INTEGER)`. Is (e) legal if the declared type of X is `INTEGER`?

Example 6.18b: Some puzzling syntactic variations

- a) `SET (X) = (1);`
- b) `SET X = 1;`
- c) `SET (X) = 1`
- d) `SET (X) = ((1));`
- e) `SET X = (1);`

Effects of NULL

If the row expression given as the source for a multiple assignment evaluates to `NULL`, then `NULL` is assigned to each target.

If a `SELECT ... INTO` statement results in an empty table, then the target variables are not updated and a completion condition is given to indicate that. This is not exactly an “effect of `NULL`”, of course, but I mention it here because of the contrast with a row subquery, which delivers a single row with `NULL` for each field when its table expression evaluates to an empty table. This observation might influence the choice between `SET` and `SELECT ... INTO`.

Historical Notes

`INSERT`, `UPDATE`, and `DELETE` have been in SQL from the beginning. `MERGE` was added in SQL:2003, `TRUNCATE` in SQL:2007.

Support for local variables and various programming language constructs is defined in Part 4 of the SQL standard, referred to as SQL/PSM. Part 4 first appeared in 1996, as an addendum to SQL:1992.

Support for multiple assignment to local variables was added in SQL:2003. `SELECT ... INTO` has been in SQL from the beginning, though until 1996 it was available only with “host” variables as targets, a host variable being one declared using some language other than SQL in a program written in that language. References to host variables are distinguished from references to SQL variables by prefixing them with colons (e.g., `:X`).

Transactions

For **Tutorial D**’s `BEGIN TRANSACTION`, `COMMIT`, and `ROLLBACK`, SQL has the same syntax except for `START` in place of `BEGIN`. However, `START TRANSACTION` is used only for outermost transactions and cannot be given when a transaction has been started and not completed. Inner transactions are started using a `SAVEPOINT` statement, giving a name—a savepoint name—that identifies the database state at the time of execution. If `SAVEPOINT SN1` has been given, for example, then `RELEASE SAVEPOINT SN1` has the same effect as a **Tutorial D** `COMMIT` for all updates performed since savepoint `SN1` was established—it merely relinquishes the possibility of cancelling just those updates and does not make their effects visible to other users. To cancel those updates `ROLLBACK TO SAVEPOINT SN1` is given, but then the savepoint name `SN1` remains in existence. In both cases, any further existing savepoints, established after `SN1`, are destroyed.

If an attempt is made to update the database when no transaction has been explicitly started, then a transaction is implicitly started. When no transaction has been started, a `SET TRANSACTION` statement can be given to specify various options to override the defaults that otherwise apply to the next transaction. The options can alternatively be specified in a `START TRANSACTION` statement. The options in effect will apply when a transaction is implicitly started or when it is started by a `START TRANSACTION` statement that does not override them.

One of the options for SET/START TRANSACTION is the so-called *isolation level*, which applies to the whole of the outermost transaction. The default isolation level is SERIALIZABLE, this being the only one that enforces all of the normally defined properties of transactions. The weakest level, READ UNCOMMITTED allows other concurrent users to see the effects of updates that have not yet been committed (and might never be, of course). Intermediate levels, READ COMMITTED and REPEATABLE READ, as well as UNCOMMITTED, allow a transaction to perceive changes to the database that have been effected by other, committed transactions (for example, by evaluating the same table expression more than once, without updating the database betweentimes, and getting different results).

Historical Notes

SET TRANSACTION appeared in SQL:1992. START TRANSACTION and SAVEPOINT were added in SQL:1999.



The image shows the BI Norwegian Business School logo, which is a central blue square with 'BI' in white, surrounded by a colorful, multi-colored starburst of lines. The lines are labeled with various business programs: Business, Strategic Marketing Management, International Business, Leadership & Organisational Psychology, Shipping Management, and Financial Economics. Below the logo is the text 'BI NORWEGIAN BUSINESS SCHOOL' and the EFMD EQUIS ACCREDITED logo.

Empowering People. Improving Business.

BI Norwegian Business School is one of Europe's largest business schools welcoming more than 20,000 students. Our programmes provide a stimulating and multi-cultural learning environment with an international outlook ultimately providing students with professional skills to meet the increasing needs of businesses.

BI offers four different two-year, full-time Master of Science (MSc) programmes that are taught entirely in English and have been designed to provide professional skills to meet the increasing need of businesses. The MSc programmes provide a stimulating and multi-cultural learning environment to give you the best platform to launch into your career.

- MSc in Business
- MSc in Financial Economics
- MSc in Strategic Marketing Management
- MSc in Leadership and Organisational Psychology

www.bi.edu/master



EXERCISES

1. SQL does not allow the empty set to be specified using `PRIMARY KEY` or `UNIQUE`. Write a table constraint that could be included in the definition of table `T` to simulate **Tutorial D's** `KEY { }`. Mention any optional conformance features of SQL that your solution uses.

2. Using some SQL implementation that is available to you, try out Example 6.1b on it, using a low number, say 1, in place of 20000. Is it accepted? If so, does it have the intended effect? If not, is it accepted when you place `BEGIN` and `END` around the `RETURN` statement? If it's still not accepted, try `RETURNS INTEGER` instead of `RETURNS BOOLEAN`, and move the comparison from the function body to the constraint. Now is it accepted? And if so, does it have the desired effect? Finally, try specifying a different base table, say `IS_CALLED`, in the `ALTER TABLE` statement (and drop the constraint from `IS_ENROLLED_ON`). When `IS_CALLED` is nonempty and `IS_ENROLLED_ON` is at its maximum cardinality, is `INSERT INTO IS_ENROLLED_ON ...` accepted? Or does the DBMS check the constraint only on updates to `IS_CALLED`?

3. Suppose the table definition for `COURSE` is extended to include a column `MaxExamMark`, whose value in each row is the maximum mark obtainable for that course's exam. `{StudentId, CourseId}` is a foreign key in `EXAM_MARK`, referencing `IS_ENROLLED_ON`. A constraint is needed to ensure that no student is awarded a mark greater than the relevant maximum.

- a) Write an SQL `ALTER TABLE` statement to address this requirement.
- b) Complete the following statement to make it equivalent to your solution for part (a):

```
CREATE ASSERTION ...
    CHECK ( SELECT EVERY( ... ) FROM EXAM_MARK ) ;
```

4. Now suppose that instead of there being a recorded maximum mark of each exam the maximum score for each question in each exam is recorded in the following relvar:

```
CREATE TABLE EXAM_QUESTION
( CourseId CID,
  Question# INTEGER,
  MaxMark INTEGER,
  PRIMARY KEY ( CourseId, Question# ) ;
```

For each course, the exam questions are supposed to be numbered sequentially, starting at 1.

- a) Write an SQL `CREATE ASSERTION` statement to address this requirement.
- b) Suppose the questions are subdivided into parts, a, b, c and so on, up to a maximum of six parts, and maximum marks are given for each part rather than for each question. Again, the parts for each question must be “numbered” sequentially, starting at a. Write an SQL `CREATE ASSERTION` statement to address *this* requirement.
- c) Devise shorthands, in the style of SQL, for expressing constraints of the kinds found in your solutions to a. and b.

5. Using the suppliers-and-parts database shown in Figure 4.13, define SQL integrity constraints to express the following requirements:

- a) Every shipment row must have a supplier number matching that of some supplier row.
- b) Every shipment row must have a part number matching that of some part row.
- c) All London suppliers must have status 20.
- d) No two suppliers can be located in the same city.
- e) At most one supplier can be located in Athens at any one time.
- f) There must exist at least one London supplier.
- g) The average supplier status must be at least 10.
- h) Every London supplier must be capable of supplying part P2.

6. For each example in Exercise 5, list the different kinds of update operation that, if permitted, would cause the constraint to be violated.

7. A database contains base tables T1 and T2. At all times at least one of these must be empty. The SQL implementation does not support `CREATE ASSERTION` but does allow subqueries to appear in table constraints. How can the stated requirement be implemented?

8. (Repeated from the body of the chapter.) Ponder the distinctions among the following examples.

- a) `SET (X) = (1) ;`
- b) `SET X = 1 ;`
- c) `SET (X) = 1`
- d) `SET (X) = ((1)) ;`
- e) `SET X = (1) ;`

Download free eBooks at bookboon.com

Which ones assign 1 to X and which assign ROW (1)? Is (b) legal if the declared type of X is ROW (F1 INTEGER). Is (e) legal if the declared type of X is INTEGER? You might like to try these out in some SQL implementation.

9. SQL has two ways of starting a transaction, `START TRANSACTION` for an outermost transaction and `SAVEPOINT` for inner ones. Describe any advantages and disadvantages you can think of for this scheme over one that uses the same method for all transactions.

10. SQL's `UNION`, `EXCEPT`, and `INTERSECT` operators are the only ones that have a `CORRESPONDING` option to specify that columns of two tables are to be paired by their names rather than their ordinal positions. List as many other operators and syntactic constructs in SQL that you can think of to which a `CORRESPONDING` option might usefully be added.

11. Consider the SQL implementation you are most familiar with. To what extent does it correctly support the standard features mentioned in this book? Is it relationally complete?

Need help with your dissertation?

Get in-depth feedback & advice from experts in your topic area. Find out what you can do to improve the quality of your dissertation!

Get Help Now



Go to www.helpmyassignment.co.uk for more info



Helpmyassignment

