Chapter 8
Integrity

*Integrity* refers to the accuracy and correctness of data in the database. For example, we might require that supplier number be of the form `Snnnn`, and unique; status values be in the range 1 to 100, etc.. The latter requirement can be represented as follows:

```
CONSTRAINT SC3 IS_EMPTY
{S WHERE STATUS<1 OR STATUS>100}
```

The above constraint will be registered in the system catalog under the name SC3.

In SQL, it may be represented as follows:

```
check (status >=1 and status <=100)
```
When checking this requirement, the involved boolean expression typically take the form \( \text{IS\_EMPTY}(\ldots) \), thus checking if the corresponding relation is empty. If the resulted table is NOT empty, this constraint has been violated.

Below is its calculus analog:

\[
\text{CONSTRAINT SC3} \\
\quad \text{FORALL SX (SX.STATUS} \geq 1 \text{ AND SX.STATUS} \leq 100); \\
\]

Note the \text{FORALL} quantifier, which is pretty typical in specifying constraints.

It is pretty easy to remove a constraint, e.g.,

\[
\text{DROP CONSTRAINT SC3; } \\
\]

This is allowed in SQL.
Constraint classification

There are four groups of integrity constraints:

1. A type constraint specifies the legal values for a given type.

2. An attribute constraint specifies the legal values for a given attribute.

3. A table constraint specifies the legal values for a given table.

4. A database constraint specifies the legal values for a given database.
Type constraints

Essentially, a type constraint enumerates the legal values of that type. E.g.,

```
TYPE WEIGHT POSSREP(RATIONAL)
   CONSTRAINT THE_WEIGHT>0.0;
```

Since any expression that can yield a value of type WEIGHT has to refer to the selector, the constraints are always immediately checked during the execution of some selector invocation of the involved type. Thus, if a system is supported by appropriate type constraints, it cannot have a bad value.

Another example,

```
Create type S# Char(5);
```
Attribute constraints

An attribute constraint declares that a specified attribute is of a specified type, and is always checked immediately. E.g.,

VAR S BASE RELATION{
  S#     S#
    Check (S# like 'S[0-9]'),
  SNAME NAME,
  STATUS INTEGER,
  CITY   CHAR}...;

Thus, the values of those attributes are constrained. Any attempt to enter a value into the database that is of a wrong type will be rejected.
Another example

In SQL, we can specify five attribute constraints, *not null*, *check*, *unique*, *primary key*, and *references*. For example,

```
Create table titles (  
title_id char(6) not null  
  primary key check (title_id  
    like '[A-Z][0-9][0-9]'),  
title varchar2(80) not null  
  unique,  
type char(12)  
  default 'unclassified' null  
  check (type in ('business', 'trad_cook',  
    'psychology', 'unclassified')),  
pub_id char(4) null  
  foreign key references publishers (pub_id),  
pubdate datatime null  
  default current date);
```

**Question:** What is the difference between *primary key* and *unique*?
Table constraints

A table constraints is put on an individual rel-var (table). For example,

CONSTRAINT SC5 IS_EMPTY
  (S WHERE CITY='London' AND STATUS NOT = 20);

CONSTRAINT PC4 IS_EMPTY
  (P WHERE COLOR=COLOR('Red') AND
   CITY NOT='London');

The commonly used table constraints are for
the primary key, and foreign key, when they are
involved with more than one attributes. For example,

CREATE TABLE SP (  
  S# CHAR(5),     
  P# CHAR(6),     
  QTY NUMERIC(9), 
  PRIMARY KEY (S#, P#), 
  FOREIGN KEY (S#) REFERENCES S, 
  FOREIGN KEY (P#) REFERENCES P);
Database constraints

This is a constraint that interrelates two or more distinct tables. E.g.,

CONSTRAINT DBC1 IS_EMPTY
  ((S JOIN SP) WHERE STATUS<20 AND QTY>QTY(500));

CONSTRAINT DBC2 SP{S#}<=S{S#};

**Question:** How to implement the above constraint?

**Answer:** Use a foreign key.

Database constraints cannot be checked immediately, but rather, they have to be deferred to end-of-transaction, i.e., to COMMIT time.

**Question:** Can we populate the SP table before we populate both the S and the P tables? Why?

**Homework:** Exercises 9.1, and 9.3 (a–i).
Golden rule

Every table should be associated with a predicate, states the properties any relation associated with this table should stick to. For example, the one for S could be this: the supplier with the specified supplier numbers (S#) has the specified name (SNAME) and the specified status value (STATUS), and is located in the specified city (CITY); moreover, no two suppliers have the same supplier number at the same time.

Such a predicate should serve as the criterion for acceptability of updates in the table in question. For example, an attempt to insert a new supplier with the same supplier number as that of some existing supplier must be rejected.
Ideally, the system should know and understand the predicate for every table, so that it could deal correctly with all possible attempts to update the database. Practically, this is difficult to do since the system does not always know everything about the suppliers, although it does know a good approximation. More precisely, it knows all the constraints that apply to the supplies at a certain point.

Thus, we define the table predicate for a specific table to be the logic and of all table constraints that apply to that table. Then, by the *Golden rule*, we mean that

No update operation is allowed on any table if its predicate will be violated.
Transition constraints

Besides the state constraints, it is also necessary to have transition constraints, i.e., constraints on legal transitions from one correct state to another. For example, it is a “legal” transition to get married to a widowed, a divorced, but never married to a married. On the other hand, it is not “legal” to let a widowed to be divorced, to let a divorced to be widowed, etc.. W.r.t. the supplier database, we have the following:

CONTRAINT TRC1 IS_EMPTY
   (((S’{S#, STATUS} RENAME STATUS AS STATUS’)
    JOIN S {S#,STATUS}) WHERE STATUS’>STATUS);

Here, a primed table name, such as S’, refers to the corresponding table as it was prior to the update under consideration. Thus, the above means that no status can be decreased.
“Keys” are the keys

The relational model always stresses the concept of keys. We will ignore the possibility that a key value can be null for a moment.

Let $R$ be a table, by definition, the set of all attributes of $R$ has the uniqueness property (?). In practice, it is often true that some proper subset already possesses this property. This leads to the concept of a candidate key:

Let $K$ be a set of attributes of table $R$. Then $K$ is a candidate key for $R$ iff 1) Uniqueness: No legal value of $R$ ever contains two distinct tuples with the same value of $K$, and 2) irreducibility: no proper subset of $K$ has the uniqueness property.
It is clear that every table does have at least one candidate key (?). The second condition is crucial. Otherwise, the relevant constraint will not be correctly enforced.

For example, if we pick up \{S#, CITY\}, instead of just \{S#\}, as the candidate key. Then, instead of enforcing the global uniqueness of S#s, we will only get the weaker case that within each city, S# will be different (?).

More specifically, such tuples as \((S1, 'Bob', 20, 'Paris'), (S1, 'Smith', 10, 'London')\) would be allowed at the same time in S.

In this case, we see that the second condition of \{S#, CITY\} being a candidate key is violated.
A (simple) key can be declared as follows:

```
KEY {<attribute names commalist>}
```

For example,

```
CREATE TABLE S (  
    S#        CHAR(5),  
    SNAME    CHAR(20),  
    STATUS   NUMERIC(5),  
    CITY     CHAR(15},  
    PRIMARY KEY (S#));
```

A key can also be composite. For example,

```
CREATE TABLE SP (  
    S#        CHAR(5),  
    P#        CHAR(6),  
    QTY       NUMERIC(9),  
    PRIMARY KEY (S#, P#));
```
A candidate key definition is really just a shorthand for a certain table constraint. Candidate keys provide the basic tuple(row)-level addressing mechanism in the relational model, i.e., it provides the only system-guaranteed way of pinpointing some specific row.

For example, $S$ WHERE $S# = S#('S3$) is guaranteed to yield at most one tuple, while $S$ WHERE CITY='Paris' yields an unpredictable number of tuples.

A superset of a candidate key is a superkey, which is unique (?), but not necessarily irreducible.

**Question:** How to say *it* in SQL?

**Answer:** Use unique

**Homework:** Exercises 9.5(a–f), 9.6, and 9.8.
Primary keys and alternate keys

When there are several candidate keys, exactly one of them should be chosen as the primary key, and others are called alternate keys. Every base table always has exactly one primary key.

A foreign key is a set of attributes of one table $R_2$ whose values are required to match values of some candidate key of another table $R_1$.

For example, consider the attribute S# of table SP, since a given value for S# should not allowed to appear in table SP if it is not already a S# value for S.
More specifically,...

Let \( R_2 \) be a table. Then a *foreign key* in \( R_2 \) is a set of attributes of \( R_2 \), say \( FK \), such that

1. There exists a table \( R_1 \) (\( R_1 \) and \( R_2 \) not necessarily distinct) with a candidate key \( CK \), and,

2. each value of \( FK \) in the current value of \( R_2 \) is identical to the \( c \) value of \( CK \) in some tuple in the current value of \( R_1 \).

Again, we say that \( S\# \) of table \( SP \) is a foreign key, since \( S\# \) is a candidate key of table \( S \), and every value of \( S\# \) in \( SP \) is identical to some value of \( S\# \) in the table \( S \).
A couple of points

1. This inclusion requirement, as in condition 2, can be proper. For example, it could be the case that some supplier has not made a shipment, yet.

2. Each attribute of a given foreign key must have the same name and type as the corresponding component of the matching candidate key.

3. A foreign key value refers to a tuple containing the matching candidate key value. The constraint such a foreign key symbolizes is called a referential constraint, a database constraint.

4. A referential diagram such as $S \leftarrow SP \rightarrow P$ gives some explicit information. In general, there can be a reference path: or even a cycle: $R_n \rightarrow R_{n-1} \rightarrow \cdots \rightarrow R_1$. 
5. $R_1$ could be the same as $R_2$. For example,

```sql
VAR EMP BASE RELATION
  {EMP#, EMP#, ..., MGR_EMP#, EMP#, ...}
PRIMARY KEY {EMP#}
FOREIGN KEY {RENAME MGR_EMP# AS EMP#}
  REFERENCES EMP;
```

The above states that a manager must be an employee.

6. Based on this foreign key concept, the relational model includes the following referential integrity rule: The database must not contain any unmatched foreign key values, i.e., if $A$ refers to $B$, $B$ must exist.
Referential actions

Consider the following statement:

DELETE S WHERE S#=S#('S1');

Assume that the database also includes some shipments for supplier S1, and the application does not delete those shipment records as well, then, when checking the database referential constraint from shipments to suppliers, it will find a violation.

Thus an obvious compensating action would be that when deleting this supplier record, all the relevant shipments records will be automatically deleted.
This can be achieved by extending the foreign key definition as follows:

Crate table SP(
  ...
  FOREIGN KEY \{S#\} REFERENCES S
  ON DELETE CASCADE;

In the above, the CASCADE is called the referential action for the above DELETE rule. Another referential action can be RESTRICT, which will restrict the deletion to those records for which there are no matching shipments, or rejected otherwise.

If no referential action is specified, a default one is specified, i.e., NO ACTION. Then, a referential error might arise.
A couple of points

1. DELETE is not the only operation for which a referential action makes sense. For example, if we want to update a supplier record, for which there are several matching shipments. Again, the above referential actions can be used to ensure the referential integrity.

2. Besides those actions as mentioned, other actions can also be taken: Relevant actions can be written to some archive database; the shipments can be transferred to other suppliers, etc.. Such user-defined action can be invoked by, e.g., the form of CALL proc(...).

3. Let $R_1, R_2,$ and $R_3$ be related as follows: $R_3 \rightarrow R_2 \rightarrow R_1$. Then, a deletion of an $R_1$ tuple will cause deletion of tuple(s) of $R_2$, which will then cause deletion of $R_3$ tuples. If any of these deletion fails, the whole things fails, and the database will not be updated. Hence, database update is always atomic.
An SQL example

CREATE TABLE SP (  
  (S# S# NOT NULL, P# P# NOT NULL,  
   QTY QTY NOT NULL,  
   PRIMARY KEY (S#,P#),  
   FOREIGN KEY (S#) REFERENCES S  
ON DELETE CASCADE  
ON UPDATE CASCADE,  
   FOREIGN KEY (P#) REFERENCES S  
ON DELETE CASCADE  
ON UPDATE CASCADE,  
   CHECK (QTY>0 AND QTY<5001));
Assertions

General constraints are defined by means of
CREATE ASSERTION, with the following syntax:

CREATE ASSERTION <constraint name>
    CHECK (<conditional expression>);

For example,
1. Every supplier has status at least five.

CREATE ASSERTION IC13 CHECK
    ((SELECT MIN (S.STATUS) FROM S)>4);

2. Every part has a positive weight.

CREATE ASSERTION IC18 CHECK
    (NOT EXISTS (SELECT * FROM P
        WHERE NOT (P.WEIGHT>0.0)));

Unfortunately, SQL does not support this fa-
cility, yet.

Homework: Check out the remaining exam-