Databases - Data Integrity

Gordon Royle

School of Mathematics & Statistics
University of Western Australia

Data Integrity

For most applications it is imperative that the database remain in a logically consistent state, so there are a variety of mechanisms to help preserve *data integrity*.

Next week, we will discuss the *transactions* mechanism, which is a very low-level mechanism robust to genuinely unpredictable events, such as system crashes and the actions of other users.

This helps maintain data integrity in several ways:

- Transactions protect against system crashes half-way through a sequence of statements reflecting a single logical operation
- Transactions permit multiple users to simultaneously use the database without needing to be aware of each other

Referential Integrity

Referential integrity means that whenever a value in one table *refers to* a tuple in another table, then the system should ensure that there is always a matching tuple in the second table.

Consider the following schema:

```
Student (id:integer, name:string)
Unit (code:string, name:string)
Enrolled (sid:integer, ucode:integer)
```

Here, `Enrolled.sid` *refers to* the `id` column in the `Student` table, while `Enrolled.ucode` *refers to* the `code` column in the `Unit` table.

(Notice that the names of the columns in `Enrolled` do not have to match the columns that are being referred to.)

Current contents

```
mysql> select * from Student;
+------+----------+
| id   | name     |
+------+----------+
|  123 | Jane     |
|  456 | Ebenezer |
|  789 | Martin   |
+------+----------+
mysql> select * from Unit;
+----------+---------------------+
| ucode    | name                |
+----------+---------------------+
| CITS1402 | Databases           |
| CITS2211 | Discrete Structures |
+----------+---------------------+
2 rows in set (0.00 sec)
```

Legitimate enrolments

Any tuple in `Enrolled` should should then connect a *legitimate student* with a *legitimate unit*.

Of the four commands

```
INSERT INTO Enrolled VALUES(123, "CITS1402");
INSERT INTO Enrolled VALUES(124, "CITS1402");
INSERT INTO Enrolled VALUES(123, "CITS1412");
INSERT INTO Enrolled VALUES(124, "CITS1412");
```

only the first should succeed.

The tuple `(124, "CITS1402")` refers to a student with id `124`, but there is no such student — this is called a *dangling pointer*.

Key constraints

The table containing the pointers (references) is called the *child table*, while the tables that the pointers point to (the references refer to) are called the *parent tables*.

(So in this case, `Enrolled` is the child table, and `Student`, `Unit` are the parent tables.)

When the referred-to column (or columns) are a key for the parent table, then SQL provides a mechanism for ensuring that the references between the tables are always in a consistent state — this is the *key constraint* mechanism.

Prepare the parent tables

The column (or columns) in each of parent tables must be a key for that table.

```
CREATE TABLE Student (
id INT PRIMARY KEY,
name VARCHAR(64));

CREATE TABLE Unit (
code INT PRIMARY KEY,
name VARCHAR(64));
```

This guarantees that each student has a unique `id` and that each unit has a unique `code` — clearly these sort of constraints are essential if the database is to make any sense at all.

Prepare the child table

```
CREATE TABLE Enrolled (
  sid INT,
  ucode VARCHAR(8),
  FOREIGN KEY (sid)
    REFERENCES Student(id),
  FOREIGN KEY (ucode)
    REFERENCES Unit(code));
```

This informs the DBMS that every value of `Enrolled.sid` should match (exactly) one of the values of `Student.id`.

After this declaration the DBMS will take over the management of these relationships and ensure that they remain consistent.

Enforcing key constraints

```
mysql> INSERT INTO Enrolled VALUES(123, "CITS1402");
Query OK, 1 row affected (0.01 sec)

mysql> INSERT INTO Enrolled VALUES(124, "CITS1402");
ERROR 1452 (23000): Cannot add or update a child row:
   a foreign key constraint fails
   (`test`.`enrolled`, CONSTRAINT `enrolled_ibfk_1`
   FOREIGN KEY (`sid`) REFERENCES `Student` (`id`))
```

As desired, any attempt to enrol a non-existent student will be prevented. (Immediately an entire set of “fat finger” data entry errors are eliminated.)

Updates

The key constraint means that data must be inserted first into the parent tables (i.e. students and units first) and only then into the child table (the enrolments).

What about updating and deleting rows?

```
mysql> UPDATE Student SET id = 1234 where id = 123;
ERROR 1451 (23000): Cannot delete or update a parent row:
  a foreign key constraint fails
  (`test`.`enrolled`, CONSTRAINT `enrolled_ibfk_1`
  FOREIGN KEY (`sid`) REFERENCES `Student` (`id`))
```

The key field for a parent row cannot be changed while some value in a child table is pointing to it, which of course would normally be exactly the desired behaviour.

Alternative options

What are some other behaviours that make sense when updating or deleting rows in these tables?

- *Forbidding* the operation
  This is the default, and operations that would cause a referential integrity violation are not executed.
- *Cascading* the operation
  This means to "fix up" the referential integrity violation by doing the same thing to the row in the child table as has happened to the matching row in the parent table.
- Using `NULL` to allow the operation to occur, but leaving the database in a state that is neither fully consistent nor fully inconsistent.

Adding the desired behaviour

```
CREATE  TABLE Enrolled (
sid INT,
ucode VARCHAR(8),
FOREIGN KEY (sid)
  REFERENCES Student(id)
  ON UPDATE CASCADE
  ON DELETE CASCADE,
FOREIGN KEY (ucode)
REFERENCES Unit(code)
  ON UPDATE CASCADE
  ON DELETE CASCADE);
```

Cascading updates

```
mysql> SELECT * FROM Enrolled;
+------+----------+
| sid  | ucode    |
+------+----------+
|  123 | CITS1402 |
|  123 | CITS2211 |
|  456 | CITS1402 |
|  456 | CITS2211 |
|  789 | CITS1402 |
|  789 | CITS2211 |
+------+----------+
```

```
UPDATE Student SET id = 1000+id;
```

```
mysql> SELECT * FROM Enrolled;
+------+----------+
| sid  | ucode    |
+------+----------+
| 1123 | CITS1402 |
| 1123 | CITS2211 |
| 1456 | CITS1402 |
| 1456 | CITS2211 |
| 1789 | CITS1402 |
| 1789 | CITS2211 |
+------+----------+
```

Cascading deletions

```
mysql> SELECT * FROM Enrolled;
+------+----------+
| sid  | ucode    |
+------+----------+
| 1123 | CITS1402 |
| 1123 | CITS2211 |
| 1456 | CITS1402 |
| 1456 | CITS2211 |
| 1789 | CITS1402 |
| 1789 | CITS2211 |
+------+----------+
```

```
DELETE FROM Student WHERE id = 1123;
```

```
mysql> SELECT * from Enrolled;
+------+----------+
| sid  | ucode    |
+------+----------+
| 1456 | CITS1402 |
| 1456 | CITS2211 |
| 1789 | CITS1402 |
| 1789 | CITS2211 |
+------+----------+
```

The options

In MySQL the options for the phrase to follow `ON UPDATE` and `ON DELETE` include:

- `NO ACTION` or `RESTRICT`
  These both prevent the operation (and can be omitted)
- `CASCADE`
  As seen above
- `SET NULL`
  Used in phrases like `ON UPDATE SET NULL` which sets the entry in the child table to `NULL` if the parent table's row is changed.

A word of warning

Common databases vary a great deal in how closely they implement the SQL standard, especially the advanced features.

Many people have been caught out by the fact that MySQL, for example, *accepts* a number of standard SQL constructions, but doesn't actually *implement* them. For example

```
CREATE TABLE Enrolled (
  sid INT REFERENCES Student(id),
  ucode VARCHAR(8) REFERENCES Unit(code));
```

is legal SQL creating a foreign key relationship.

MySQL accepts the syntax, creates the table, but silently ignores the desired constraints!

Self-referential integrity

It is possible to have a single table that is simultaneously the parent (referred-to) and the child (referring) table. For example, suppose each student is assigned a *mentor* (a higher-level student) to look after them on first arriving at University.

```
CREATE TABLE Student (
  id INT PRIMARY KEY,
  name VARCHAR(64),
  mentor INT,
  FOREIGN KEY (mentor)
    REFERENCES Student(id)
      ON UPDATE CASCADE ON DELETE CASCADE);
```

Some experimenting

```
INSERT INTO Student VALUES (123, 'Amy', NULL);
INSERT INTO Student VALUES (345, 'Bill',NULL);
INSERT INTO Student VALUES (678, 'Charlie',NULL);
UPDATE Student SET mentor = 345 WHERE id = 678;
UPDATE Student SET mentor = 123 WHERE id = 345;
```

```
mysql> select * from Student;
+-----+---------+--------+
| id  | name    | mentor |
+-----+---------+--------+
| 123 | Amy     |   NULL |
| 345 | Bill    |    123 |
| 678 | Charlie |    345 |
+-----+---------+--------+
```

Some experimenting

What will happen if Amy graduates and we update the DB with

```
DELETE FROM Student WHERE id = 123;
```

Some experimenting

What will happen if Amy graduates and we update the DB with

```
DELETE FROM Student WHERE id = 123;
```

```
mysql> DELETE FROM Student WHERE id = 123;
Query OK, 1 row affected (0.00 sec)

mysql> select * from Student;
Empty set (0.00 sec)
```

The "cascade" did what we *asked* it to do, not what we *wanted* it to do (computers are like that).

Some experimenting

```
CREATE TABLE Student (
  id INT PRIMARY KEY,
  name VARCHAR(64),
  mentor INT,
  FOREIGN KEY (mentor)
    REFERENCES Student(id)
      ON UPDATE CASCADE ON DELETE SET NULL);

mysql> DELETE FROM Student WHERE id = 123;
mysql> select * from Student;
+-----+---------+--------+
| id  | name    | mentor |
+-----+---------+--------+
| 345 | Bill    |   NULL |
| 678 | Charlie |    345 |
+-----+---------+--------+
2 rows in set (0.00 sec)
```

Other forms of integrity

In addition to referential integrity, SQL provides a number of mechanisms that can be used to *validate data* to help catch either data entry errors, or statements that violate the "business logic" underlying the database.

One way to accomplish this is to use `CHECK` constraints although, as we learned last week, MySQL will accept but silently ignore any `CHECK` statements.

Basic CHECK

Last week we considered

```
CREATE TABLE BankAccount (
  accountNumber INT,
  balance REAL CHECK (balance > -1000));
```

Using PostgreSQL as the database, we get

```
INSERT INTO BankAccount VALUES(123,-2000);
ERROR:  new row for relation "bankaccount"
  violates check constraint "bankaccount_balance_check"
  DETAIL:  Failing row contains (123, -2000)
```

Changing constraints

Each constraint has a name, in this case the automatically-supplied name `bankaccount_balance_check` so we can refer to the constraint if we need to remove it or replace it with a different one.

```
ALTER TABLE BankAccount
  DROP CONSTRAINT bankaccount_balance_check;
```

We can impose a more rigorous constraint:

```
ALTER TABLE BankAccount
  ADD CONSTRAINT new_balance_check
    CHECK (balance > 0);
```

Multi-column constraints

Check constraints can be declared over more than one column (thus enforcing some sort of relationship across the whole row) and they can use the normal logical operators.

```
CREATE TABLE BankAccount(
  id INT PRIMARY KEY,
  balance REAL,
  investment REAL,
  CHECK ((balance > 0) OR
         (balance > -1000 AND investment > 10000)));
```

Customers with at least $10,000 invested with the bank can be up to $999.99 dollars overdrawn, but all other customers must maintain a positive balance.

Multi-column constraints

The multi-column check works exactly as expected.

```
INSERT INTO BankAccount VALUES (1, -100, 5000);
ERROR:  new row for relation "bankaccount"
  violates check constraint "bankaccount_check"
DETAIL:  Failing row contains (1, -100, 5000).
```

Multiple `CHECK` constraints can be added to a table so any number of potential violations can be intercepted.

Subqueries

In principle (i.e. according to the SQL standard) `CHECK` constraints can also include *subqueries*. This permits a wide range of strong constraints to be enforced.

```
CREATE TABLE BankAccount (
  id INT PRIMARY KEY,
  customer_id INT,
  balance REAL,
  CHECK (balance > (-1)*SELECT SUM(I.balance)
                  FROM InvestmentAccount I
                  WHERE customer_id = I.customer_id));
```

Subqueries

Allowing subqueries in `CHECK` constraints also raises a large number of implementation issues – in particular, when the constraint should be checked?

Simple check constraints can basically only be violated when a tuple is *updated* or *inserted* — so the check is performed immediately before or after the update or insertion.

Constraints with subqueries can become invalid when *any of the tables involved* changes, via updates, insertions or deletions.

The performance penalty involved in repeatedly checking every row of a table for validity on every change to some distantly-related table means it is normally never implemented.

Adaptive checks

The `CHECK` mechanism is a blunt instrument — it simply generates an immediate error as soon as a constraint is violated, and leaves the user (or user's application) to remedy the problem.

Sometimes however, there are obvious actions that can be taken — for example, if an update results in a `BankAccount` balance dropping below the lower limit, then the shortfall might be automatically transferred from the investment account if enough money is available.

This cannot be achieved using `CHECK` constraints only, but as we have seen, the *trigger mechanism* can meet these requirements.

Other integrity mechanisms

There are a variety of other ways to constrain the values that can occur in a column

- `NOT NULL`
  Adding this after the type of an attribute (e.g. `taxFileNumber INT NOT NULL`) guarantees that this column will never contain any `NULL` entries.
- `UNIQUE`
  Adding this after the type of an attribute (e.g. `taxFileNumber INT UNIQUE`) guarantees that this column will never contain any duplicate entries.

A sample question - what is the output?

```
CREATE TABLE R (A INT PRIMARY KEY);
CREATE TABLE S (B INT PRIMARY KEY,
     FOREIGN KEY (B) REFERENCES R(A) ON UPDATE CASCADE);
CREATE TABLE T (C INT PRIMARY KEY,
     FOREIGN KEY (C) REFERENCES S(B) ON UPDATE CASCADE);

INSERT INTO R VALUES (1); INSERT INTO R VALUES (2);
INSERT INTO R VALUES (3); INSERT INTO R VALUES (4);
INSERT INTO R VALUES (5); INSERT INTO R VALUES (6);

INSERT INTO S VALUES (1); INSERT INTO S VALUES (2);
INSERT INTO S VALUES (4); INSERT INTO S VALUES (6);

INSERT INTO T VALUES (1); INSERT INTO T VALUES (2);
INSERT INTO T VALUES (6);

UPDATE R SET A = A + 10 WHERE A < 5;
SELECT SUM(C) FROM T;
```