SQL support for integrity constraints

Hi! We are going to discuss SQL support for integrity constraints.

Types of integrity constraints

(1) Non-null
(2) Key
(3) Referential integrity
(4) Attribute-based
(5) Tuple-based
(6) General assertions

Non-Null Constraints

Restricts attributes to not allow NULL values

```
Ex: CREATE TABLE Student (ID integer NOT NULL,
                          name char(30) NOT NULL,
                          address char(100),
                          GPA float NOT NULL,
                          SAT integer)
```

Ex: ID is key for Student => no two tuples in Student can have the same values for their ID attribute

There are two kinds of keys in SQL, Key Constraints

(1) PRIMARY KEY: at most one per relation, automatically non-null, automatically indexed (in Oracle)

(2) UNIQUE: any number per relation, automatically indexed

There are two ways to define keys in SQL:

(a) With key attribute

(b) Separate within table definition

Ex: CREATE TABLE Student (ID integer PRIMARY KEY,
name char(30),
address char(100),
GPA float,
SAT integer,
UNIQUE (name,address))

Referential Integrity

Referenced attribute must be PRIMARY KEY
(e.g., Student.ID, Campus.location)

Referencing attribute called FOREIGN KEY
(e.g., Apply.ID, Apply.location)

There are two ways to define referential integrity in SQL:

(1) With referencing attribute
(2) Separate within referencing relation

Ex: CREATE TABLE Apply(ID integer REFERENCES Student(ID),
location char(25),

```
date char(10),
major char(10),
decision char,
FOREIGN KEY (location) REFERENCES Campus(location))
```

Can omit referenced attribute name if it's the same as referencing attribute:

```
ID integer REFERENCES Student, ...
```

Can have multi-attribute referential integrity. Can have referential integrity within a single relation.

```
Ex: Dorm(first-name,
    last-name,
    room-number,
    phone-number,
    roommate-first-name,
    roommate-last-name,
    PRIMARY KEY (first-name,last-name),
    FOREIGN KEY (roommate-first-name,roommate-last-name)
      REFERENCES Dorm(first-name,last-name))
```

A foreign key is a group of attributes that is a primary key is some other table. Think of the foreign key values as pointers to tuples in another relation. It is important to maintain the consistency of information in the foreign key field and the primary key in the other relation. In general the foreign key should not have values that are absent from the primary key, a tuple with such values is called a dangling tuple (akin to a dangling pointer in programming languages).

Referential integrity is the property that this consistency has been maintained in the database.

A foreign key in the *enrolledIn* table: Assume that the following tables exist.

- *student*(*name*, *address*) - *name* is the primary key
- *enrolledIn*(*name*, *code*) - *name*, *code* is the primary key
- *subject*(*code*, *lecturer*) - *code* is the primary key

The *enrolledIn* table records which students are enrolled in which subjects. Assume that the tuple (*joe*, *cp2001*) indicates that the student *joe* is enrolled in the subject *cp2001*. This tuple only models reality if in fact *joe* is a student. If *joe* is not a student then we have a problem, we have a record of a student enrolled in some subject but no record that that student actually exists! We assert this reasonable conclusion by stating that for the *enrolledIn* table, *name* is a foreign key into the *student* table. It turns out that *code* is also a foreign key, into the *subject* table. Students must be enrolled in subjects that actually exist.

If we think about the E/R diagram from which these relations were determined, the *enrolledIn* relation is a relationship type that connects the *student* and *subject* entity types. Relationship types are the most common source of foreign keys constraints.

In SQL, a foreign key can be defined by using the following syntax in the column specification portion of a CREATE TABLE statement.

<*column name*> <*column type*> ... **REFERENCES** <*table name*> ...,

Let's look at an example.

Creating the *enrolledIn* table:

```
CREATE TABLE enrolledIn (
        name VARCHAR(20) REFERENCES student,
        code CHAR(6) REFERENCES subject
        );
```

Alternatively, if more than one column is involved in a single foreign key, the foreign key constraint can be added after all the column specifications.

CREATE TABLE <*table name*> (
    <*column specification*> ...
    REFERENCES(<*list of columns*>) <*table name*>, ...
);

Let's look at an example.

Creating the *enrolledIn* table:

```
CREATE TABLE enrolledIn (
    name        VARCHAR(20),
    code        CHAR(6),
    REFERENCES(name) student,
    REFERENCES(code) subject
    );
```

Policies for maintaining integrity

The foreign key constraint establishes a relationship that must be maintained between two relations, basically, the foreign key information must be a subset of the primary key information. Below we outline the *default* SQL policy for maintaining referential integrity.

- Allow inserts into primary key table.
- Reject updates of primary key values (potentially must update foreign key values as well).
- Reject deletion of primary key values (potentially must delete foreign key values as well).
- Reject inserts into foreign key table if inserted foreign key is not already a primary key.
- Reject updates in foreign key table if the updated value is not already a primary key.

- Allow deletion from foreign key table.

Updating the student mini-world: By default SQL supports the following.

- Inserts into the *student* table are allowed.
- Updates of the *name* attribute in the *student* table are rejected since it could result in a dangling tuple in the *enrolledIn* table.
- Deleting a tuple from the *student* table is rejected since it could result in a dangling tuple in the *enrolledIn* table.
- Inserts into the *enrolledIn* table are permitted only if *name* already exists in the *student* table (and *code* in the *subjects* table).
- Updates of the *name* attribute in the *enrolledIn* table are permitted only if the updated *name* already exists in the *student* table.
- Deleting a tuple from the *enrolledIn* table is permitted.

While this is the default (and most conservative) policy, individual vendors may well support other, more permissive, policies.

Constraints

In SQL we can put constraints on data at many different levels.

Constraints on attribute values

These constraints are given in a CREATE TABLE statement as part of the column specification. We can define a column to be NOT NULL specifying that no null values are permitted in this column. The constraint is checked whenever a insert or update is made to the constrained column in a table.

A general condition can be added to the values in a column using the following syntax after the type of a column.

CHECK <*condition*>

The condition is checked whenever the column is updated. If the condition is satisfied, the update is permitted, but if the check fails, the update is rejected (hopefully with an apropos error message!).

A column for logins: Suppose that the *login* column in a student table should conform to JCU style logins (e.g., sci-jjj or jc222222). In the CREATE TABLE statement, the *login* column can be defined as follows.

login CHAR(10) CHECK (login LIKE '___-%' OR login LIKE 'jc%')

Recall that LIKE is a string matching operator. The '_' character will match a single possible ASCII character while '%' will match zero or more ASCII characters. The condition is checked whenever a new login is inserted or a login is updated.

Another way to check attribute values is to establish a DOMAIN and then use that DOMAIN as the type of a column.

Representing gender: Below we create a DOMAIN for representing gender.

CREATE DOMAIN genderDomain CHAR(1) CHECK (value IN ('F','M'));

This DOMAIN only has two possible values 'F' and 'M'. No other value can be used in a column of genderDomain type. In the CREATE TABLE statement the created DOMAIN can be used like any other type in the column specification.

```
CREATE TABLE student (
    ...
    gender genderDomain,
    ...
    );
```

A user will only be able to insert or update the gender to some value in the genderDomain.

Constraints on tuples

Constraints can also be placed on entire tuples. These constraints also appear in the CREATE TABLE statements; after all the column specifications a CHECK can be added.

CREATE TABLE <*table name*> (

...

CHECK <*condition on tuples*>, ...

);

Too many male science students: Let's assume that the powers that be have decided that there are way too many male science students and so only female science students will be allowed. If we assume that all science students have a login that starts with 'sci-' then we can enforce this constraint using the following syntax.

```
CREATE TABLE student (
    ...
    gender genderDomain,
    login CHAR(10) CHECK (login LIKE '___-%' OR login LIKE 'jc%'),
    ...
    CHECK (login LIKE 'sci-%' AND gender = 'F'),
);
```

Constraints on tuples in more than one tables

Constraints can also be added to tuples in several tables. Since these constraints involve more than one table they are not part of the CREATE TABLE statement, rather they are declared separately as follows.

CREATE ASSERTION <*assertion name*> **CHECK** <*constraint*>;

The constraint that is checked usually involves a SELECT.

Example: Science enrollments are down and so the word has gone out that no science student can be enrolled in fewer than five subjects!

```
CREATE ASSERTION RaiseScienceRevenue CHECK
    5 > ALL
        (SELECT COUNT(code)
```

```
                    FROM EnrolledIn, Student
                    GROUP BY code
                    WHERE login LIKE 'sci-%');
```

Later after there are no more science students the administrators decide this is a stupid idea and choose to remove the constraint.

```
DROP ASSERTION RaiseScienceRevenue;
```

General Assertions

Constraints on entire relation or entire database. In SQL, stand-alone statement:

```
CREATE ASSERTION  CHECK ()
```

Ex: Average GPA is > 3.0 and average SAT is > 1200

```
CREATE ASSERTION HighVals CHECK(
  3.0 < (SELECT avg(GPA) FROM Student) AND
  1200 < (SELECT avg(SAT) FROM Student))
```

Ex: A student with GPA < 3.0 can only apply to campuses with rank > 4.

```
CREATE ASSERTION RestrictApps CHECK(
 NOT EXISTS (SELECT * FROM Student, Apply, Campus
        WHERE Student.ID = Apply.ID
        AND Apply.location = Campus.location
        AND Student.GPA < 3.0 AND Campus.rank <= 4))
```

Assertions checked for each modification that could potentially violate them.

Triggers

A trigger has three parts.

1. event - The trigger waits for a certain event to occur. Events are things like an insertion into a relation.
2. condition - Once an event is detected the trigger checks a condition (usually somehow connected to the event - e.g., after an insertion check to see if the salary attribute is non-negative). If the condition is false then the trigger goes back to sleep, waiting for the next relevant event. If the condition is true then an *action* (see below) is executed. In effect the trigger is waiting for events that satisfy a certain condition.
3. action - An action is a piece of code that is executed. Usually this will involve either undoing a transaction or updating some related information.

Shoe salaries cannot be lowered: Suppose that we wanted to ensure that people in the shoe department can *never* have their salary *lowered*. We would like to set up a trigger that waits for update events to the employee relation, and if those updates lower the salary of a shoe employee, then the update should be undone. In SQL3 we could set up the trigger using the following statement.

```
CREATE TRIGGER ShoeSalaryTrigger
AFTER UPDATE OF salary ON employee
REFERENCING
        OLD AS OldTuple
        NEW AS NewTuple
WHEN (OldTuple.salary > NewTuple.salary)
UPDATE employee
SET salary = OldTuple.salary FOR EACH ROW
```

In this statement the *event specification* is in the second line, the condition is the sixth line, and the action is the seventh and eighth lines. The third, fourth, and fifth lines establish tuple variables, OldTuple and NewTuple, that are the original and updated tuple in the employee table. The trigger is activated just *after* rather than just *before* the update, as specified in line 2. Finally, the ninth line specifies that each row (tuple) in the

employee table must be checked.

The example above demonstrates that triggers are useful in situations where an arbitrary action must be performed to maintain the consistency and quality of data in a database. But each trigger imposes an overhead on database processing, not only to wake up triggers and check conditions, but to perform the associated action. So triggers should be used judiciously.

Recursion

The proposed SQL3 standard extends SQL in an interesting direction. A well-known limitations of SQL is that it does not have recursion, or a "looping" mechanism. For example in SQL (or with relations) we can represent a graph by maintaining a table of edges.

from	**to**
a	b
b	c
c	d

The *Edge* table

Within this graph a simple SQL query can be used to compute nodes that are connected by paths of length two by joining the *Edge* table to itself.

```
SELECT A.from, B.to
FROM Edge as A, Edge as B
WHERE A.to = B.from;
```

The result of this query is given below.

from	**to**
a	a
b	d

The result of *Edge* ⋈*Edge*

For this graph there is also a path of length three from *a* to *d* via *b* and *c*, so to get the transitive closure we would have to do a further join with *Edge*. In general since we can't know in advance how many joins need to be done, we need to have some kind of loop where we keep joining *Edge* to the result of the previous round to derive new connections in the graph. SQL3 has such a recursive construct.

WITH RECURSIVE .... TODO

Declaring and Enforcing Constraints

Two times at which constraints may be declared:

1. Declared with original schema.
   Constraints must hold after bulk loading.

2. Declared later.
   Constraints must hold on current database.

After declaration, if a SQL statement causes a constraint to become violated then (in most cases) the statement is aborted and a run-time error is generated.