CS 145 Introduction to Databases

Stanford University, Fall 2015

Referential Integrity in SQLite

Declaring Referential Integrity (Foreign Key) Constraints

Foreign key constraints are used to check referential integrity between tables in a database. Consider, for example, the following two tables:

```
create table Residence (
  name VARCHAR PRIMARY KEY,
  capacity INT
);
create table Student (
  id INT PRIMARY KEY,
  firstName VARCHAR,
  lastName VARCHAR,
  residence VARCHAR
);
```

We can enforce the constraint that a Student's residence actually exists by making `Student.residence` a foreign key that refers to `Residence.name`. SQLite lets you specify this relationship in several different ways:

```
create table Residence (
  name VARCHAR PRIMARY KEY,
  capacity INT
);
create table Student (
  id INT PRIMARY KEY,
  firstName VARCHAR,
  lastName VARCHAR,
  residence VARCHAR,
  FOREIGN KEY(residence) REFERENCES Residence(name)
);
```

or

```
create table Residence (
  name VARCHAR PRIMARY KEY,
  capacity INT
);
create table Student (
  id INT PRIMARY KEY,
  firstName VARCHAR,
  lastName VARCHAR,
  residence VARCHAR REFERENCES Residence(name)
);
```

or

```
create table Residence (
  name VARCHAR PRIMARY KEY,
```

```
  capacity INT
);
create table Student (
  id INT PRIMARY KEY,
  firstName VARCHAR,
  lastName VARCHAR,
  residence VARCHAR REFERENCES Residence    -- Implicitly references the
      primary key of the Residence table.
);
```

All three forms are valid syntax for specifying the same constraint.

Constraint Enforcement

There are a number of important things about how referential integrity and foreign keys are handled in SQLite:

- The attribute(s) referenced by a foreign key constraint (i.e. `Residence.name` in the example above) must be declared `UNIQUE` or as the `PRIMARY KEY` within their table, but this requirement is checked at run-time, not when constraints are declared. For example, if `Residence.name` had not been declared as the `PRIMARY KEY` of its table (or as `UNIQUE`), the `FOREIGN KEY` declarations above would still be permitted, but inserting into the `Student` table would always yield an error.
- **Foreign key constraints are not checked by default.** If you want SQLite to enforce foreign key constraints specified on your tables, you must enable them with the command:

  ```
  PRAGMA foreign_keys = ON;
  ```

  once per database session (i.e. once per invocation of `/usr/class/cs145/bin/sqlite`). Even if you have previously enabled foreign key constraint checking while using a particular database, new sessions with that database will not check foreign key constraints unless you issue this `PRAGMA` command. If you forget to do so, foreign key constraints will no longer be enforced for that session, and any insertions that normally would violate a foreign key constraint will execute without any warning or error message!
- **Bulk-loading into a SQLite database while checking referential integrity can be very, very slow** – we do not recommend it. It is faster to bulk-load your data with referential integrity checking turned off instead. You can then run SQL queries over your tables to verify that these constraints hold. After that, constraint-checking cand (and should!) be turned on.
- Referential integrity checking can be *deferred*. This means that the constraint is not checked until the current transaction ends, or, if there is no active transaction, when the current statement ends. This can be useful when adding tuples to multiple tables – so that you don't need to worry about the order of inserts, or, in the case of referential integrity, between two tables in both directions.

  A foreign key constraint can be made deferrable with the keywords `DEFERRABLE INITIALLY DEFERRED`:

  ```
  create table Residence (
    name VARCHAR PRIMARY KEY,
    capacity INT
  );
  create table Student (
    id INT PRIMARY KEY,
    firstName VARCHAR,
    lastName VARCHAR,
    residence VARCHAR REFERENCES Residence DEFERRABLE INITIALLY DEFERRED
  );
  ```

  By doing so, we can write:

  ```
  BEGIN;
  ```

```
insert into Student values (123, 'Ben', 'Savage', 'Gavilan');
insert into Residence values ('Gavilan', 50);
COMMIT;
```

and no error will be raised after the insertion into `Student`.

- Special action can be taken when the referenced tuple is updated or deleted. Let's say our `Residence` and `Student` tables contain the tuples inserted above:

```
select * from Student;
id      firstName  lastName    residence
------  ---------  ---------   ---------
123     Ben        Savage      Gavilan

select * from Residence;
name     capacity
-------  --------
Gavilan  50
```

  By default, updating or removing the tuple in `Residence` is not permitted, since it would leave the tuple in `Student` *dangling*, and the database would therefore be in an inconsistent state. SQLite supports `ON UPDATE` and `ON DELETE` actions that will automatically keep the database in a consistent state upon modification:

  - `RESTRICT`: This is the default behavior – it prohibits a change to the `Residence` tuple, as long as there are `Student` tuples that depend on it.
  - `CASCADE`: Changes to the `Residence` tuple will be propagated to all `Student` tuples that depend on it.
  - `SET NULL`: A change to the `Residence` tuple will set the referencing value in `Student.residence` to `null`.

  `ON UPDATE` and `ON DELETE` actions can be specified along with a foreign key constraint declaration, as in these examples:

```
residence VARCHAR REFERENCES Residence ON UPDATE RESTRICT
residence VARCHAR REFERENCES Residence ON DELETE CASCADE
residence VARCHAR REFERENCES Residence ON UPDATE SET NULL ON DELETE RESTRICT
```