ECE356: Database Systems Winter 2018

Lecture 6 — Data Definition

Jeff Zarnett

We have thus far not yet learned about how to formally create relations. It is now time to fix that. The SQL data definition language (DDL) looks a lot like the query language that we have used thus far, but it allows us to define the structure of the data.

Specifically, our data definition language allows us to define [**?**]:

- The schema of each relation.
- The type of each attribute.
- Integrity constraints.
- Indices on relations.
- Security/authorization information for a relation.
- Physical storage structure.

Attributes have types, and the SQL standard includes the following built in types [**?**, **?**]:

- **char**(n): A fixed length character string with a user-specified length of n. This one should never be used because it pads strings and it means comparisons are a pain and comparing a char attribute with another kind of string is an issue.
- **varchar**(n): A variable length character string with user-specified length n.
- **int** (or **integer**): an integer (exact size and maximum value is system dependent).
- **smallint**: a smaller integer. A fun-sized integer, so to speak.
- **numeric**(p, d): a fixed-point number with user-specified prevision; p is the the number of digits and d of those are to the right of the decimal point.
- **real**: floating point with a machine-dependent precision.
- **double precision**: double-precision floating point with machine-dependent precision.
- **float**(n): a floating point number with precision of at least n.
- **boolean**: A boolean value (what did you expect?)
- **date**, **time**, **datetime**: self explanatory, right?

Schema Definition - Create Table

If we wish to define a SQL relation, the syntax for this is to create a relation (table) is called (unsurprisingly), `create table`. The syntax for this command requires a name as well as a listing of the attributes (fields) and their types (definitions). It is also customary to include at least one constraint, the primary key. As before we put a semicolon at the end of the statement to designate the end of the statement.

```
CREATE TABLE r
 (A1 D1, A2 D2, ... An Dn,
 integrity-constraint-1,
 ...
  integrity-constraint-k);
```

A more concrete example:

```
CREATE TABLE student
 (id varchar(8),
  userid varchar(8) NOT NULL,
  firstname varchar(64),
  lastname varchar(64),
  birthday date,
  department_id int,
  PRIMARY KEY( id )
  FOREIGN KEY( department_id ) REFERENCES department( id )
 );
```

On some attributes an additional qualifier `not null` was added, and this means that a value of `null` is forbidden from being assigned to that attribute. This is, in a way, a form of integrity constraint. Like all integrity constraints, it means an insert or update statement that tries to set such an attribute to an impermissible value will be rejected.

The primary-key definition in this case specifies that `id` is the primary key for the relation which imposes requirements that the attribute cannot be null and must also be unique. In this situation we have defined the primary key as exactly one attribute, but a relation's primary key may be formed by the combination of several attributes.

The last sort of constraint that is shown in the example is the foreign key constraint that mentions the department ID. It means that for each student, the department id value must either be null or the value must match to the id value of a department tuple. If the types of the attributes don't match (e.g., department is defined with varchar for its id rather than integer) then adding this foreign key constraint will fail.

We did not do this in the above case, but we can put a name to foreign keys or other constraints. The name must be unique in the database schema,

Another way that adding a foreign key might fail is if the target relation does not exist. This means we would have to create relations in some order that means the foreign keys are all satisfied at the time of the creation. That might not be realistic, though, based on the desired schema. Fortunately, we can add them in later.

Altering tables

In addition to adding in some integrity constraints we can change the table definition, or remove constraints.

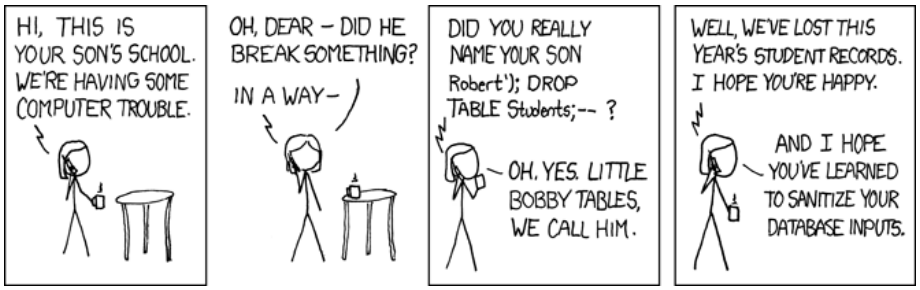
Truncate Table

If we wish to remove all tuples from a relation without affecting the structure at all, the command for that is to truncate the table: `TRUNCATE TABLE students;` would remove from the database all tuples of the students relation but would leave its definition unchanged. Truncating the table may fail if it would violate some constraints (breaking referential integrity).

There are some scenarios where this is a desirable operation. One possible scenario is if you have some application where login session information is stored in the databAse and you want to invalidate and remove all sessions you can truncate the table.

Drop Table

If we wish to remove a relation from the schema, the syntax for this is `DROP TABLE students`. This deletes the table and all of its content and then the content is permanently lost. And again, the drop operation may fail if the table to be deleted is referenced in some external constraints.



Obligatory XKCD (`https://xkcd.com/327/`).