SQL: Data Definition Language

csc343, Introduction to Databases
Diane Horton
Winter 2016
Types
Table attributes have types

• When creating a table, you must define the type of each attribute.
• Analogous to declaring a variable’s type in a program. Eg, “int num;” in Java or C.
• Some programming languages don’t require type declarations. Eg, Python.
• Pros and cons?
• Why are type declarations required in SQL?
Built-in types

- **CHAR(n)**: fixed-length string of n characters. Padded with blanks if necessary.
- **VARCHAR(n)**: variable-length string of up to n characters.
- **TEXT**: variable-length, unlimited. Not in the SQL standard, but psql and others support it.
- **INT = INTEGER**
- **FLOAT = REAL**
- **BOOLEAN**
- **DATE; TIME; TIMESTAM** (date plus time)
Values for these types

• **Strings**: ‘Shakespeare’s Sonnets’
  Must surround with single quotes.

• **INT**: 37

• **FLOAT**: 1.49, 37.96e2

• **BOOLEAN**: TRUE, FALSE

• **DATE**: ‘2011-09-22’

• **TIME**: ‘15:00:02’, ‘15:00:02.5’

• **TIMESTAMP**: 'Jan-12-2011 10:25'
And much more

- These are all defined in the SQL standard.
- There is much more, e.g.,
  - specifying the precision of numeric types
  - other formats for data values
  - more types
- For what psql supports, see chapter 8 of the documentation.
User-defined types

• Defined in terms of a built-in type.
• You make it more specific by defining constraints (and perhaps a default value).
• Example:

```sql
create domain Grade as int
    default null
check (value>=0 and value <=100);
create domain Campus as varchar(4)
    default 'StG'
check (value in ('StG','UTM','UTSC'));
```
Semantics of type constraints

• Constraints on a type are checked every time a value is assigned to an attribute of that type.
• You can use these to create a powerful type system.
Semantics of default values

• The default value for a type is used when no value has been specified.
• Useful! You can run a query and insert the resulting tuples into a relation -- even if the query does not give values for all attributes.
• Table attributes can also have default values.
• The difference:
  • attribute default: for that one attribute in that one table
  • type default: for every attribute defined to be of that type
Keys and Foreign Keys
Key constraints

• Declaring that a set of one or more attributes are the **PRIMARY KEY** for a relation means:
  • they form a key (unique and no subset is)
  • their values will never be null (you don’t need to separately declare that)
• Big hint to the DBMS: optimize for searches by this set of attributes!
• Every table must have 0 or 1 primary key.
  • A table can have no primary key, but in practise, every table should have one. Why?
  • You cannot declare more than one primary key.
Declaring primary keys

• For a single-attribute key, can be part of the attribute definition.

```sql
create table Blah (  
    ID integer primary key,  
    name varchar(25));
```

• Or can be at the end of the table definition. (This is the only way for multi-attribute keys.) The brackets are required.

```sql
create table Blah (  
    ID integer,  
    name varchar(25),  
    primary key (ID));
```
Uniqueness constraints

• Declaring that a set of one or more attributes is **UNIQUE** for a relation means:
  • they form a key (unique and no subset is)
  • their values *can* be null; if they mustn’t, you need to separately declare that

• You can declare more than one set of attributes to be **UNIQUE**.
Declaring UNIQUE

• If only one attribute is involved, can be part of the attribute definition.
  ```sql
  create table Blah (  
    ID integer unique,  
    name varchar(25));
  ```

• Or can be at the end of the table definition. (This is the only way if multiple attributes are involved.) The brackets are required.
  ```sql
  create table Blah (  
    ID integer,  
    name varchar(25),  
    unique (ID));
  ```
We saw earlier how nulls affect “unique”

• For uniqueness constraints, no two nulls are considered equal.

• E.g., consider:
  ```sql
  create table Testunique ( 
      first varchar(25), 
      last varchar(25), 
      unique(first, last)
  )
  ```

• This would prevent two insertions of
  ('Diane', 'Horton')

• But it would allow two insertions of
  (null, 'Schoeler')

This can’t occur with a primary key. Why not?
Foreign key constraints

• Eg in table Took:
  `foreign key (sID) references Student`

• Means that attribute sID in this table is a foreign key that reference the primary key of table Student.
  • Every value for sID in this table must actually occur in the Student table.

• Requirements:
  • Must be declared either primary key or unique in the “home” table.
Declaring foreign keys

• Again, declare with the attribute (only possible if just a single attribute is involved) or as a separate table element.

• Can reference attribute(s) that are not the primary key as long as they are unique; just name them.

```sql
create table People (  
    SIN integer primary key,  
    name text,  
    OHIP text unique  
);
create table Volunteers (  
    email text primary key,  
    OHIPnum text references People(OHIP));
```
Enforcing foreign-key constraints

• Suppose there is a foreign-key constraint from relation R to relation S.

• When must the DBMS ensure that:
  • the referenced attributes are PRIMARY KEY or UNIQUE?
  • the values actually exist?

• What could cause a violation?

• You get to define what the DBMS should do.

• This is called specifying a “reaction policy.”
Other Constraints and Assertions
“check” constraints

• We’ve seen a check clause on a user-defined domain:
  
  ```sql
  create domain Grade as smallint
  default null
  check (value>=0 and value <=100);
  ```

• You can also define a check constraint
  
  • on an attribute
  • on the tuples of a relation
  • across relations
Attribute-based “check” constraints

• Defined with a single attribute and constrain its value (in every tuple).
• Can only refer to that attribute.
• Can include a subquery.
• Example:
  ```sql
  create table Student (  
    sID integer,  
    program varchar(5) check  
      (program in (select post from P)),  
    firstName varchar(15) not null, ...);
  ```
• Condition can be anything that could go in a WHERE clause.
When they are checked

• Only when a tuple is inserted into that relation, or its value for that attribute is updated.

• If a change somewhere else violates the constraint, the DBMS will not notice. E.g.,
  • If a student’s program changes to something not in table P, we get an error.
  • But if table P drops a program that some student has, there is no error.
“not null” constraints

• You can declare that an attribute of a table is NOT NULL.

create table Course(
    cNum integer,
    name varchar(40) not null,
    dept Department,
    wr boolean,
    primary key (cNum, dept));

• In practise, many attributes should be not null.

• This is a very specific kind of attribute-based constraint.
Tuple-based “check” constraints

• Defined as a separate element of the table schema, so can refer to any attributes of the table.
• Again, condition can be anything that could go in a WHERE clause, and can include a subquery.
• Example:

```sql
create table Student (  
    sID integer,  
    age integer, year integer,  
    college varchar(4),  
    check (year = age - 18),  
    check college in  
      (select name from Colleges));
```
When they are checked

- Only when a tuple is inserted into that relation, or updated.
- Again, if a change somewhere else violates the constraint, the DBMS will not notice.
How nulls affect “check” constraints

• A check constraint only fails if it evaluates to false.
• It is not picky like a WHERE condition.
• E.g.: check (age > 0)

<table>
<thead>
<tr>
<th>age</th>
<th>Value of condition</th>
<th>CHECK outcome</th>
<th>WHERE outcome</th>
</tr>
</thead>
<tbody>
<tr>
<td>19</td>
<td>TRUE</td>
<td>pass</td>
<td>pass</td>
</tr>
<tr>
<td>-5</td>
<td>FALSE</td>
<td>fail</td>
<td>fail</td>
</tr>
<tr>
<td>NULL</td>
<td>unknown</td>
<td>pass</td>
<td>fail</td>
</tr>
</tbody>
</table>
Example

• Suppose you created this table:
  ```sql
  create table Frequencies(
    word varchar(10),
    num integer,
    check (num > 5));
  ```

• It would allow you to insert (`'hello'`, `null`) since `null` passes the constraint `check (age > 0)`

• If you need to prevent that, use a “not null” constraint.
  ```sql
  create table Frequencies(
    word varchar(10),
    num integer not null,
    check (num > 5));
  ```
Naming your constraints

- If you name your constraint, you will get more helpful error messages.
- This can be done with any of the types of constraint we’ve seen.
- Add `constraint «name»` before the `check («condition»)`
Examples

cREATE DOMAIN Grade AS SMALLINT
    DEFAULT NULL
    CONSTRAINT gradeInRange
        CHECK (value >= 0 AND value <= 100);

CREATE DOMAIN Campus AS VARCHAR(4)
    NOT NULL
    CONSTRAINT validCampus
        CHECK (value IN ('StG', 'UTM', 'UTSC'));

CREATE TABLE Offering(
    ...;
    CONSTRAINT validCourseReference
        FOREIGN KEY (cNum, dept) REFERENCES Course);
• Order of constraints doesn’t matter, and doesn’t dictate the order in which they’re checked.
Assertions

- Check constraints apply to an attribute or table. They can’t express constraints across tables, e.g.,
  - Every loan has at least one customer, who has an account with at least $1,000.
  - For each branch, the sum of all loan amounts < the sum of all account balances.
- Assertions are schema elements at the top level, so can express cross-table constraints:
  create assertion (<name>) check (<predicate>);
Powerful but costly

• SQL has a fairly powerful syntax for expressing the predicates, including quantification.

• Assertions are costly because
  • They have to be checked upon every database update (although a DBMS may be able to limit this).
  • Each check can be expensive.

• Testing and maintenance are also difficult.

• So assertions must be used with great care.
Triggers

• Assertions are powerful, but costly.
• Check constraints are less costly, but less powerful.
• Triggers are a compromise between these extremes:
  • They are cross-table constraints, as powerful as assertions.
  • But you control the cost by having control over when they are applied.
The basic idea

• You specify three things.
  • Event: Some type of database action, e.g., after delete on Courses or before update of grade on Took
  • Condition: A boolean-valued expression, e.g., when grade > 95
  • Action: Any SQL statements, e.g., insert into Winners values (sID)
Using SQL “schemas”
Schema: a kind of namespace

• “psql csc343h-dianeh” connects you to a database called csc343h-dianeh.
  (Substitute your cdf userid of course.)
• Everything defined (tables, types, etc.) goes into one big pot.
• Schemas let you create different namespaces.
• Useful for logical organization, and for avoiding name clashes.
Creating a schema

• You already have a schema called “public”.
• You can also create your own. Example:
  ```sql
  create schema University;
  ```
• To refer to things inside a particular schema, you can use dot notation:
  ```sql
  create table University.Student (...);
  select * from University.Student;
  ```
When you don’t use dot notation

• If you refer to a name without specifying what schema it is within:
  • Any new names you define go in the schema called “public”
  • E.g., if you create a table called frindle, you actually are defining public.frindle.
  • When referring to a name, there is a search path that finds it.
The search path

• To see it the search path:
  
  show search_path;

• You can set the search path yourself. Example:
  
  set search_path to University, public;

• The default search path is: “$user”, public
  
  • schema “$user” is not created for you, but if you create it, it’s at the front of the search path.
  
  • schema public is created for you.
Removing a schema

• Easy:
  
  drop schema University cascade;

• “cascade” means everything inside it is dropped too.

• To avoid getting an error message if the schema does not exist, add “if exists”.
Usage pattern

• You can use this at the top of every DDL file:

```
drop schema if exists University cascade;
create schema University;
set search_path to University;
```

• Helpful during development, when you may want to change the schema, or test queries under different conditions.
Workflow

• One effective way to work:
  • Create a DDL file with the schema.
  • Create a file with inserts to put content in the database.
  • In the postgresQL shell, import these.
  • Run queries directly in the shell or by importing queries written in files.
Reaction Policies
Example

• Suppose R = Took and S = Student.
• What sorts of action must simply be rejected?
• But a deletion or update with an sID that occurs in Took could be allowed ...
Possible policies

- **cascade**: propagate the change to the referring table
- **set null**: set the referring attribute(s) to null
- There are other options we won’t cover. Many DBMSs don’t support all of them.
- If you say nothing, the default is to forbid the change in the referred-to table.
Reaction policy example

• In the University schema, what should happen in these situations:
  • csc343 changes number to be 543
  • student 99132 is deleted
  • student 99132’s grade in csc148 is raised to 85.
  • csc148 is deleted
Note the asymmetry

• Suppose table R refers to table S.
• You can define “fixes” that propagate changes backwards from S to R.
• (You define them in table R because it is the table that will be affected.)
• You cannot define fixes that propagate forward from R to S.
Syntax for specifying a reaction policy

• Add your reaction policy where you specify the foreign key constraint.

• Example:
  ```
  create table Took ( 
      ... 
      foreign key (sID) references Student 
        on delete cascade 
      ... 
  );
  ```
What you can react to

• Your reaction policy can specify what to do either
  • **on delete**, i.e., when a deletion creates a dangling reference,
  • **on update**, i.e., when an update creates a dangling reference,
  • or both. Just put them one after the other.
Example:
**on delete restrict on update cascade**
What your reaction can be

• Your policy can specify one of these reactions (there are others):
  • restrict: Don’t allow the deletion/update.
  • cascade: Make the same deletion/update in the referring tuple.
  • set null: Set the corresponding value in the referring tuple to null.
Semantics of Deletion

- What if deleting one tuple violates a foreign key constraint, but deleting others does not?
Semantics of Deletion

• What if deleting one tuple affects the outcome for a tuple encountered later?
• To prevent such interactions, deletion proceeds in two stages:
  • Mark all tuples for which the WHERE condition is satisfied.
  • Go back and delete the marked tuples.
DDL Wrap-up
Updating the schema itself

• **Alter:** alter a domain or table
  ```sql
  alter table Course
  add column numSections integer;
  alter table Course
  drop column breadth;
  ```

• **Drop:** remove a domain, table, or whole schema
  ```sql
  drop table course;
  ```

• **How is that different from this?**
  ```sql
  delete from course;
  ```

• **If you drop a table that is referenced by another table, you must specify “cascade”**

• **This removes all referring rows.**
There’s more to DDL

• For example, you can also define:
  • indices: for making search faster (we’ll discuss these later).
  • privileges: who can do what with what parts of the database

• See csc443.