SQL

Data Definition Language (DDL)

CSC343, Introduction to Databases
Nosayba El-Sayed (based on slides from Diane Horton)
Fall 2015
Modifying a Database
Midterm next week

• Time: Oct 27 6-7pm  (Be there 10 min early)
• Location: Brennan Hall (BR200)

• Material:  everything up to Embedded SQL (so embedded sql is not included)

• Practice writing:
  • RA Queries
  • SQL Queries + Aggregates + Subqueries
  • Views

• Understand the rest of the material well, of course!
Database Modifications

• Queries return a relation.
• A modification command does not; it changes the database in some way.

• Three kinds of modifications:
  • **Insert** a tuple or tuples.
  • **Delete** a tuple or tuples.
  • **Update** the value(s) of an existing tuple or tuples.
Two ways to insert

- We’ve already seen two ways to insert tuples into an empty table:
  
  \[
  \text{INSERT INTO «relation» VALUES «list of tuples»;}
  \]
  \[
  \text{INSERT INTO «relation» («subquery») ;}
  \]

- These can also be used to add tuples to a non-empty table.
Naming attributes in INSERT

• Sometimes we want to insert tuples, but we don’t have values for all attributes.

• If we name the attributes we are providing values for, the system will use NULL or a default for the rest.

• Convenient!
Example

```sql
CREATE TABLE Invite (  
  name TEXT,  
  campus TEXT DEFAULT 'StG',  
  email TEXT,  
  age INT);

INSERT INTO Invite VALUES  
(  
'Mark','StG','m@m.com',18);

INSERT INTO Invite(name, email)  
(  
SELECT firstname, email  
FROM Student  
WHERE cgpa > 3.4 ) ;

Here, name and email get values from the query, campus gets the default value, and age gets NULL.
```
Deletion

• Delete tuples satisfying a condition:
  \[
  \text{DELETE FROM «relation» WHERE «condition»;}
  \]

• Delete all tuples:
  \[
  \text{DELETE FROM «relation»;}
  \]
Example 1: Delete Some Tuples

```
DELETE FROM Course
WHERE NOT EXISTS (
    SELECT *
    FROM Took JOIN Offering
    ON Took.oid = Offering.oid
    WHERE
        grade > 50 AND
        Offering.dept = Course.dept AND
        Offering.cnum = Course.cnum
);
```
Updates

• To change the value of certain attributes in certain tuples to given values:

```
UPDATE «relation»
SET «list of attribute assignments»
WHERE «condition on tuples»;
```
Example: update one tuple

- Updating one tuple:
  ```sql
  UPDATE Student
  SET campus = 'UTM'
  WHERE sid = 999999;
  ```

- Updating several tuples:
  ```sql
  UPDATE Took
  SET grade = 50
  WHERE grade >= 47 and grade < 50;
  ```
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; TIMESTAMP** (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).

Examples:

```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. (Think of the PK as the identity of a row..)
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**
  • 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:
  create table Testunique (  
    first varchar(25),  
    last varchar(25),  
    unique(first, last)  
  )

• This would prevent two insertions of  
  ('Stephen', 'Cook')

• But it would allow two insertions of  
  (null, 'Rackoff')  
  This can’t occur with a primary key. Why not?
Foreign key contraints

• Eg in table Took:
  
  \texttt{foreign key \{sID\} references Student}

• Means that attribute sID in this table is a foreign key that references the \textit{primary key} of table Student.
  
  • Every value for sID in this table must actually occur in the Student table.

• Requirements:
  
  • Must be declared either \textit{primary key} or \textit{unique} in the “home” table (i.e. table “Student” in the above example)
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.

• How/when can the DBMS ensure that:
  1. the referenced attributes are **PRIMARY KEY** or **UNIQUE**?
  2. the values actually exist?

• What could cause a violation?
  • Example: a row is deleted from **Course**; **Offering** is now referring to a course that doesn’t exist.

• You get to define what the DBMS should do.

• This is called specifying a “**reaction policy**.”
Reaction Policies

What’s the DBMS default?

Can we change it? How?
Example

• Suppose $T = \text{Took}$ and $S = \text{Student}$.

1. What sorts of action must simply be rejected?
   • Inserting in $T$ a student whose SID is not in $S$

2. How about a deletion or update of a Student whose SID occurs in Took?
Possible policies

- **cascade**: propagate the change to the referring table
  - E.g. if a *Student* leaves university, delete all their referrals in *Took!*

- **set null**: set the referring attribute(s) to null

- 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 
  ... 
  );
  ```
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 you can react to
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 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.
Example

- In the Guesses table, some students made more than one guess:
  - Who are these students?
  - Delete their guesses from the table!

```
csc343h-nosayba=> select * from guesses;
+-----------+--------+-----+
<table>
<thead>
<tr>
<th>number</th>
<th>name</th>
<th>guess</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Cole</td>
<td>365</td>
</tr>
<tr>
<td>2</td>
<td>Avery</td>
<td>500</td>
</tr>
<tr>
<td>3</td>
<td>Sam</td>
<td>502</td>
</tr>
<tr>
<td>4</td>
<td>Madeleine</td>
<td>390</td>
</tr>
<tr>
<td>5</td>
<td>Cole</td>
<td>450</td>
</tr>
<tr>
<td>6</td>
<td>Michael</td>
<td>1000</td>
</tr>
<tr>
<td>7</td>
<td>Mackenzie</td>
<td>700</td>
</tr>
<tr>
<td>8</td>
<td>Mackenzie</td>
<td>701</td>
</tr>
</tbody>
</table>
+-----------+--------+-----+
(8 rows)
```
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:
  ```
  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 \texttt{false}.
• It is not picky like a \texttt{WHERE} condition.

• \texttt{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 Tester(
      num integer,
      word varchar(10),
      check (num>5));
  ```

• It would allow you to insert `(null, ‘hello’)`

• If you need to prevent that, use a “not null” constraint.
  ```sql
  create table Tester(
      num integer not null,
      word varchar(10),
      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
  
  \texttt{constraint «name»}
  
before the

  \texttt{check («condition»)}
Examples

```sql
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);
```
Exercise

- Create table Student. sID is primary key. firstName, surName cannot be null. campus can only be StG, UTM, or UTSc.

```sql
CREATE TABLE Student (
    sID INTEGER,
    surName VARCHAR(25) NOT NULL,
    firstName VARCHAR(25) NOT NULL,
    campus VARCHAR(5)
check (campus in ('StG', 'UTM', 'UTSC')),
    email VARCHAR(30),
    cgpa FLOAT,
    PRIMARY KEY (sID)
);```
• 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:
  ```sql
  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, course)
Using SQL “schemas”
Schema: a kind of namespace

• “psql csc343h-mark” connects you to a database called csc343h-mark. (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:
  ```
  create schema University;
  ```
• To refer to things inside a particular schema, you can use dot notation:
  ```
  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:
  ```sql
  show search_path;
  ```

• You can set the search path yourself. Example:
  ```sql
  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:**
  
  ```sql
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.
Updating the schema itself

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

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

• How is that different from this?
  delete from course;

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

• This removes all referring rows.