CPS221 Lecture: Relational Database Creation and Design

last revised November 12, 2014

Objectives:

1. To introduce the anomalies that result from redundant storage of data
2. To review the notion of primary and foreign keys
3. To introduce the notion of lossless decomposition
4. To introduce the notion of functional dependency
5. To introduce integrity constraints
6. To introduce the SQL create table statement

Materials:

1. Projectable of schema diagram for SQL use lab database
2. Projectable of code for creating database used in SQL Use lab
3. Ability to demonstrate the above using db2
4. Projectable of mysql data types

I. Basic Principles of Relational Database Design

A. The topic of relational database design is a complex one, and one we consider in detail in the DBMS course. For now, we look at a few simple principles.

B. One principle is that each relation should have a subset of its attributes which, together, form a PRIMARY KEY for the relation.

1. It is helpful, then, to specify the primary key of each relation as part of the design process.

2. Of course, we need the primary key of an entity in order to create the tables for any relationships in which it participates, since the primary keys of the entities become columns in the table representing the relationship.

3. Good DBMS software will be capable of enforcing a PRIMARY KEY CONSTRAINT - i.e. a primary key can be declared when a table is created, and the DBMS will signal an error if an attempt is made to insert or modify a row in such a way as to create two rows with the same primary key value(s).

C. Another principle is to develop the database scheme in such a way as o avoid storage of redundant information.

1. Often, this will involve decomposing a relation scheme into two or more smaller schemes.

   EXAMPLE:

   We might be inclined to represent information about student registrations by a scheme like this:

```
enrolled(department, course_number, section,
             days, time, room, title,
             student_id, last_name, first_name,
             faculty_id, professor_name)
```

2. However, a scheme like this exhibits several serious problems, all arising from REDUNDANCY:

   a) The course's id, days, time, room, and title are stored once for each student enrolled - potentially dozens of copies.

   b) The student's id, last and first names are stored once for each course the student is enrolled in.

   c) The professor's id and name is stored once for each student enrolled in each course he/she teaches

3. Redundancy is a problem in its own right, since it wastes storage, and increases the time needed to back up or transmit the information. Moreover, redundancy gives rise to some additional problems beyond wasted space and time:

a) The UPDATE ANOMALY.

Suppose the room a course meets in is changed. Every Enrolled row in the database must now be updated - one for each student enrolled.

(1) This entails a fair amount of clerical work.

(2) If some rows are updated while others are not, the database will give conflicting answers to the question "where does ____ meet?"

b) An even worse problem is the DELETION ANOMALY.

(1) Suppose that the last student enrolled is dropped from the course. All information about the course in the database is now lost! (One might argue that this is not a problem, since courses with zero enrollment make no sense. However, this could happen early in the registration process - e.g. if a senior is mistakenly registered for a freshman course, and this is corrected before freshmen register. In any case, the decision to delete a course should be made by an appropriate administrator, not by the software!

(2) Likewise if a student is dropped from all his/her courses, information about the student is lost. This may not be what is intended.

c) There is a related problem called the INSERTION ANOMALY:

(1) We cannot even store information in the database about a course before some student enrolls - unless we want to create a "dummy" student.

(2) Likewise, we cannot store information about a student until the student is enrolled in at least one course.

(3) Can you think of another example?

ASK

We cannot store information about a faculty member who is not teaching any courses - e.g. a faculty member on sabbatical.

4. A better scheme - though still not a perfect one - would be to break this scheme up into several tables:

```
enrolled(department, course_number, section,
student_id)
course(department, course_number, section, days,
time,
              room, title, faculty_id)
student(student_id, last_name, first_name)
professor(faculty_id, professor_name)
```

The process of breaking a large single scheme into two or more smaller schemes is called DECOMPOSITION.

(Actually, this example should be decomposed further, for reasons that we won't be able to talk about here.)

D. Decomposition must be done with care, lest information be lost.

EXAMPLE:

Suppose, in avoiding to store redundant information, we had come up with this decomposition (same as above, except for no enrolled scheme, and no faculty_id attribute in course.)

```
course(department, course_number, section, days, time,
      room, title)
student(student_id, last_name, first_name)
professor(faculty_id, professor_name)
```

1. It appears that we haven't lost any information - all the data that was stored in the original single scheme is still present in some scheme. Indeed, each value is stored in exactly one table.

2. However, we call such a decomposition a LOSSY-JOIN DECOMPOSITION, because we have actually lost some information.

   What information have we lost?

   ASK

   a) We have lost the information about what students are enrolled in what courses.

   b) We have lost the information about which faculty member teaches which course.

   c) In contrast, our original decomposition was LOSSLESS-JOIN. If we did the following natural join (where |X| stands for natural join):

      ```
      enrolled |X| course |X| student |X| professor
      ```

      we would get back the undecomposed table we started with.

      (If we tried to do a similar set of natural joins on our lossy-join decomposition, we would end up with every student enrolled in every course, taught by every professor!)

3. The "acid test" of any decomposition performed to address redundancy is that it must be LOSSLESS-JOIN.

E. Recall that the overall design for a database can be represented by a SCHEMA DIAGRAM.

PROJECT: Schema diagram for database used in lab

1. Note: one box per table - specifying table name

2. Each of the columns in the table is listed in its box.

3. Each box is divided into "compartments" for primary key and other attributes - except for an "all key" table.

   The primary key compartment is shaded - but this is not done for "all key" tables.

4. The arrows represent foreign keys.

5. Primary keys and foreign keys actually arise from the underlying characteristics of the reality being modeled - which we represent by a notion known as a *functional dependency*.

II. **Functional Dependencies**

A. Definition: for some relation-scheme R, we say that a set of attributes B (B a subset of R) is functionally dependent on a set of attributes A (A a subset of R) if, for any legal relation on R, if there are two tuples t1 and t2 such that t1[A] = t2[A], then it must be that t1[B] = t2[B].

(This can be stated alternately as follows: there can be no two tuples t1 and t2 such that t1[A] = t2[A] but t1[B] != t2[B].)

B. We denote such a functional dependency as follows:

A → B (Read: A determines B)

Example: For our Enrolled database, the following FD's certainly hold:

department, course_number → title
department, course_number, section → days
department, course_number, section → time
department, course_number, section → room
student_id → last_name
student_id → first_name
faculty_id → professor_name

C. One interesting question is the relationship between department, course_number, and section, on the one hand, and professor on the other hand.

1. Since courses can be team taught, a simple FD would be incorrect - e.g.

   NOT: department, course_number, section → faculty_id

2. However, there is a relationship between sections of a course and faculty teaching the section. The relationship is a more complicated one called a MULTIVALUED DEPENDENCY, which we won't discuss in this course (though we do in the DBMS course.)

3. Note that functional dependencies are defined in terms of the UNDERLYING REALITY that the database models - not some particular set of values in the database.

   For example, it happens that, for the students in many courses
   last_name → first_name
   (and sometimes first_name → last_name!)

   However, this is not inherent in the underlying reality, so we would not include it as an FD in designing a database representation for students in a course.

D. From the FD's, we can determine the candidate keys, and choose primary keys, for the scheme.

1. Formally, we say that some set of attributes K is a SUPERKEY for some relation scheme R if

   $$K \rightarrow R$$

   (a superkey determines <u>all</u> the attributes of a relation)

2. We say that K is a CANDIDATE KEY if it has no proper subsets that are superkeys.

   a) *EXAMPLE:* For the scheme

      student(student_id, last_name, first_name)

      R - the set of all attributes - is { student_id, last_name, first_name }

      { student_id, last_name } is a superkey, because

      student_id, last_name → student_id, last_name, first_name

      but { student_id, last_name } is not a candidate key, because student_id all by itself is a superkey

      student_id is a superkey because

      student_id → student_id, last_name, first_name

      student_id is also a candidate key, because it obviously has no proper subsets that are superkeys.

   b) *EXAMPLE:* For the same scheme, if we insisted that no two students could have the same full name, we would have

      last_name, first_name → student_id, last_name, first_name

      In this case, last_name and first_name would be both a superkey and a candidate key. (In general, though, this is not a good idea!)

3. The primary key for any scheme is simply the candidate key that is chosen to serve this purpose. (If there is only one candidate key, of course, there is no choice - but in some cases there may be multiple candidate keys to choose from.)

a) *Example:* For the example professor scheme we used at the start of the lecture, `professor(faculty_id, professor_name)`,
the only FD is

`faculty_id` → `faculty_id, professor_name`

so `faculty_id` is necessarily our choice for primary key

b) *Example:* There actually are two relevant FDs that hold for the course scheme

```
department, course_number, section →
department, course_number, section,
 days, time, room, title, faculty_id
```

but also

```
days, time, room →
department, course_number, section, days, time,
 room, title, faculty_id
```

so in this case we could choose either `department, course_number, section` or `days, time, room` as the primary key (though obviously the first would be a lot easier to use; moreover, it wouldn't be subject to last minute changes, so it would almost certainly be the one we choose.

E. It important to bear in mind that functional dependencies are properties of the underlying reality - not a particular set of data.

Example: For the people in this class, the dependency

last_name → first_name

appears to hold. But this is not a fundamental property of the reality; it is an accident of a particular set of data. Thus, we would not include this as an FD when designing a relational database to represent enrollment in a class!

F. An important property of FD's is that, if we decompose a scheme R into two schemes R1 and R2, the join is lossless if and only if at least one of $R1 \cap R2 \rightarrow R1$ or $R1 \cap R2 \rightarrow R2$ holds - i.e. the intersection of the two schemes must be a candidate key for at least one of them if the join is to be lossless

Example: Given the scheme (W, X, Y, Z) with dependencies

$W \rightarrow X, Y$ and $Y \rightarrow Z$

1. The decomposition (W, X, Y) and (Y, Z) is lossless because the intersection of the two schemes is Y, which is a key of the second scheme.

2. The decomposition (W, X) and (W, Y, Z) is lossless because the intersection of the two schemes is W, which is actually a key of both

3. The decomposition (W, X, Z) and (X, Y, Z) is not lossless because the intersection of the two schemes is X, Z which is not a key of either

III. **Integrity Constraints**

A. One of the advantages of using a DBMS rather than accessing files directly is that it is possible, when setting up a database, to specify various integrity constraints on the data that are enforced for all accesses to the data.

1. We consider this topic much more extensively in CPS352, but for now we will look at some high points.

2. When appropriate constraints are incorporated into the design of a database, they are enforced for <u>every</u> program that uses that database. Moreover, if a constraint is changed, the change will be enforced for all future accesses without in any way needing to modify programs

accessing the database. (This is one advantage that results from using a DBMS rather than accessing data files directly.)

3. For our demonstrations, we will be using db2 instead of the mysql system you used in lab. At the present time, mysql does not implement all of the kinds of checking that are built into "industrial strength" DBMS's. (However, we used it in lab because it's a lot easier to work with!)

   (Setup to run db2
   • use bash
   • be sure command db2 start database manager has been given
   • use db2 -t to get use of semicolons as terminators
   • connect using connect to cps221 user bjork)

B. We have already looked at one kind of constraint, called ENTITY INTEGRITY: Each table should have a primary key whose value serves to uniquely identify rows in the table. This follows from the fact that a relational table models a SET, and therefore each row in it must be unique among all the elements of that set.

1. This can be incorporated into the declaration for the table by means of a PRIMARY KEY constraint.

   PROJECT: code for creating database used in SQL use lab.

   A primary key constraint can be specified in one of two ways

   a) If the primary key consists of a single column, it can be specified as a column constraint.

      Example: id column in student

   b) If the primary key is composite, it must be expressed as a table constraint

      Example: department, course_number in course_offered

c) Note, by the way, that in both cases the relevant column(s) needs a not null constraint.

2. When a primary key is declared, DBMS will enforce the requirement that no two rows in the table will be allowed to have the same value(s) in the specified column(s). (Recall that we demonstrated this as part of the lecture introducing DBMS's) Let's look at another example

   Demo: `insert into student values('1111111', 'Zebra', 'Zelda', 'CPS');`

3. There is another form of entity integrity constraint - the unique constraint - that we won't discuss in this course.

C. REFERENTIAL INTEGRITY: A principle related to using lossless join decompositions to avoid redundancy is the explicit identification of FOREIGN KEYS.

1. In our lossless join decomposition, what made the decomposition work correctly is that the first scheme - enrolled - had foreign keys that referenced the course and student tables; and course had a foreign key that referenced the professor table.

2. Entity integrity constraints pertain to data within a single table, and can be enforced by looking at that table alone. Often, it is also the case that the logic of a system demands that an entry cannot logically occur in one table without a related entry occuring in another table:

   Example: A course section should not appear in current_term_course unless the corresponding course appears in course_offered, because certain important information about the course (its title and number of credits) appear only there.

3. This is, of course, the notion of a foreign key, which is what the arrows in a schema diagram represent.

   PROJECT SQL use lab database schema

4. The requirement that a matching row occur in another table for each value occurring in a certain column (or set of columns) in a particular table is called REFERENTIAL INTEGRITY. Referential integrity constraints are expressed in SQL by using a FOREIGN KEY constraint, which is specified by the use of the reserved words FOREIGN KEY and REFERENCES.

   PROJECT: code for creating database used in SQL use lab.
   Note foreign key constraint in course_offered

5. The DBMS will not allow a row to be inserted into the referencing table without the corresponding row existing in the referenced table.

   Demo `insert into current_term_course values('AAA', '101', ' ', 'MWF', '0800', 'KOS124');`

D. Finally, DATA INTEGRITY is concerned with ensuring the ACCURACY of the data.

1. In particular, the concern is with protecting the data from ACCIDENTAL inaccuracy, due to causes like:

   a) Data entry errors

   b) System crashes

   c) Anomalies due to concurrent and/or distributed processing

2. Of course, certain constraints are implicit in the data type for a column. For example, if a column is declared to be of a numeric type, it is not possible to insert an arbitrary character string into it.

   DEMO: insert a row into gradepoints with grade_points not numeric

```
insert into gradepoints values('W', 'x');
```

3. Because this sort of data typing is coarse-grained, relational DBMS's also support more fine grained specification of permitted values.

   PROJECT: code for creating database use in SQL use lab
   Note check constraint on gradepoints

   DEMO: `insert into course_taken values('1111111', 'BCM', '103', '2009FA', 4, 'Z');`

IV. Creating a Database in SQL

A. The first step in creating a database is to create the database itself, using the SQL `create database` statement. However, most DBMS's restrict this operation to database administrators, so we won't discuss it further. Instead, we will assume that the database itself has been created, and we are now ready to create the tables in it.

B. Most of the work of database creation is done using the SQL `create table` statement. Rather than discussing the syntax in detail, we'll use an example that illustrates the key features.

   We looked at the code for creating the database used in lab 11 - now we will look at it again to see some of the features of the SQL `create table` statement.

   PROJECT: Lab 11 database creation code

C. SQL `create table` statement consists of the phrase `create table` followed by the table name and a parenthesized list of column and constraint definitions.

   1. A column definition consists of the column name followed by the data type for the column.

a) There are quite a lot of possibilities for a data type. This is the set of data types available in mysql.

   PROJECT: mysql Data Types

   The most commonly used type is CHAR, which must specify a size (number of characters) for the column.

b) A column definition may also specify one or more constraints that the DBMS will enforce on values stored in the column, including the following:

   `(1)not null`

   `(2)primary key` (must also specify `not null`)

   `(3)unique`

   `(4)foreign key references` *table name*

   Walk through column definitions for student table

2. A constraint may also be specified as a table constraint. This must always be done when the constraint applies to multiple columns.

   Walk through column constraints for current_term_course table

3. Finally, check constraints that control the permissible values of a data item may also appear

   Walk through check constraint in course_taken table