# CS211 Lecture: Database Design

last revised November 21, 2006

*Objectives:*

1. To introduce the anomalies that result from redundant storage of data
2. To introduce the notion of functional dependencies
3. To introduce the basic rules for BCNF normalization

*Materials:*

1. Handout: progressive normalization of a registration scheme to BCNF

## 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, which we will make more formal later.

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)
```

*HANDOUT*

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, as we shall see - 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

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. 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. Many DBMS's (though not MySQL, unfortunately), allow foreign keys to be declared when a table is created. The DBMS will then enforce the rule that no row can be inserted or modified in such a way as to have foreign key values that do not appear in some row of the table being referenced.

   e.g. if we made student_id a foreign key in Enrolled, referencing the Student table, then it would be impossible to insert a row in Enrolled containing a student_id that does not appear in Student.

F. Nulls

1. One interesting question that arises in database design is how are we to handle a situation where we don't have values available for all the attributes of an entity. We have already seen that relational DBMS's provide a special value called NULL that can be stored in such a column.

2. In designing a database, it will sometimes be necessary to specify that certain columns CANNOT ever contain a NULL value. This will necessarily be true of any column that is part of the primary key, and may be true of other columns as well. Most DBMS's allow the designer to specify that a given column cannot be NULL.

3. One wants to think rather carefully about whether one wants to allow a column to contain a NULL value.

   a) Allowing NULL is certainly appropriate if the column represents information one might not have or which might not be meaningful for every row in the table.

      Example: A medical records database might want to allow `age` to be NULL because this might not be known for a patient treated in the emergency room,

      Example: A medical records database might want `insurance` to be NULL to allow for uninsured patients.

   b) NULL values are problematic if allowing them is a result of poor design.

      Example: In our original scheme for EnrolledIn (the one with all attributes in one table), if we wanted to store information about a student who is not enrolled in any courses we could only do so by recording the student as "enrolled" in a NULL course. But this is a consequence of bad design, and ceases to be necessary when we decompose the design properly.

c) Sometimes the question is an open one, with good arguments possible for both answers.

Example: Suppose you wanted to store your Video Store information in a database. There are two ways of representing rental information:

(1) You could have a separate `rented` table with attributes `customerID`, `copyID` and `dateDue`.

(2) You could include `rentedToCustomerID` and `dateDue` attributes in the `Copy` table, because a given copy can only be rented to one person at a time. (These would have to be NULL if the item was on the shelf.)

(3) Advantages/disadvantages of the approaches?

ASK

(a) A separate table is cleaner, and has the advantage that recording a rental or return is a matter of inserting/deleting a row, rather than modifying one.

(b) Including rental information in the `Copy` table makes some inquiries more efficient, since it avoids the need for joining two tables.

## 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:

$A \rightarrow B$

(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 → R

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

3. *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.

4. *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!)

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!

## III. Normal Forms

A. Relational database theorists have developed a number of normal forms which can be used to develop appropriate designs. These are covered in detail in a DBMS course. For now, we'll just look at them briefly.

1. The tests are applied separately to the design of each entity set.

2. If any design fails a test, it is typically NORMALIZED by decomposing it into two or more entity sets which share a common key.

B. First Normal Form (1NF):

1. A relation scheme R is in 1NF iff, for each tuple t in R, each attribute of t is atomic - i.e. it has a SINGLE, NON-COMPOSITE VALUE.

2. This rules out:

a) Repeating groups

b) Composite columns in which we can access individual components - e.g. dates that can be either treated as unit or can have month, day and year components accessed separately.

3. Example:

Recall the problem that arises because of team teaching (the FD department, course_number, section → faculty_id does NOT hold).

We might try to solve this by storing several values in the faculty_id column - e.g. (using names instead of faculty id's, since I don't know the latter!)

```
FA 112   TR 9:45 J237 'ARTS IN CONCERT'
                    (PELKEY, JONES, HERMAN)
```

However, this is not in 1NF, since the faculty attribute is not atomic

4. Any non-1NF scheme can be made 1NF by "flattening" it - e.g.

FA 112 TR 9:45 J237 'ARTS IN CONCERT' PELKEY
FA 112 TR 9:45 J237 'ARTS IN CONCERT' JONES
FA 112 TR 9:45 J237 'ARTS IN CONCERT' HERMAN

a) i.e. we create three distinct rows, one for each value

b) Of course, this creates new redundancy problems, addressed by the theory of multivalued dependencies.

c) Since we won't be discussing these here, we'll assume for the rest of our example that all courses have a SINGLE faculty member (either no team teaching, or include only one professor in the listing for the course)

5. 1NF is desirable for most applications, because it guarantees that each attribute in R is functionally dependent on the primary key, and simplifies queries.

6. However, there are some applications for which atomicity may be undesirable - e.g. keyword columns in bibliographic databases, or multimedia databases where a "column" may actually be a movie. Normalization theory for such situations is still being researched.

C. Second Normal Form (2NF):

1. A 1NF relation scheme R is in 2NF iff each non-key attribute of R is FULLY functionally dependent on each candidate key. By FULLY functionally dependent, we mean that it is functionally dependent on the whole candidate key, but not on any subset of it.

2. Example: Consider our original Enrollment scheme:

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

What is/are the candidate key(s)?

*ASK*

department, course_number, section, student_id is our CK

because: department and course_number together determine title; department, course_number, and section together determine days, time, room and faculty_id; student_id determines last_name and first_name, and faculty_id determines professor_name

However, we have several partial dependencies:

a) title depends only on department, course_number

b) days, time, room, and faculty_id depend only on department, course_number, and section

c) last_name and first_name depend only on student_id

3. Any non-2NF scheme can be made 2NF by a decomposition in which we factor out the attributes that are dependent on only a portion of a candidate key, together with the portion they depend on.

   Example: The dependencies listed above lead to the following 2NF decomposition

```
Course(department, course_number, title)
Section(department, course_number, section, days,
       time, room,
       faculty_id. professor_name)
Student(student_id, last_name, first_name)
Enrolled(department, course_number, section,
        student_id)
```

4. Observe that any 1NF relation scheme which does NOT have a COMPOSITE primary key is, of necessity, in 2NF.

5. 2NF is desirable because it avoids repetition of information that is dependent on part of the primary key, but not the whole key, and thus prevents various anomalies.

D. Third Normal Form (3NF) / Boyce-Codd Normal Form (BCNF)

1. In the history of normalization theory, there was developed a definition for what is called third normal form (3NF)

   a) It was later shown that this definition does not eliminate all undesirable redundancies (and hence anomalies.) As a result, a new, more stringent definition was proposed.

   b) However, there are some unusual cases where the new definition creates a problem that the initial definition did not have. For this reason, both definitions have been retained, and the new definition is called Boyce-Codd Normal Form (BCNF).

c) We will use the more stringent BCNF form (which is actually easier to test).

2. A normalized relation R is in BCNF iff, for every dependency of the form A → B that hold on R, A is a superkey. (There are no dependencies on attribute sets that are not superkeys)

3. Example: In the above decomposition, Section is not BCNF (or 3NF for that matter) because there is what is called a transitive dependency.

   a) The primary key of Section is department, course_number, section. Any superkey must include these attributes.

   b) But faculty_id by itself determines professor_name - i.e.

      faculty_id → professor_name holds

      This is called a transitive dependency because the primary key determines professor_name indirectly - i.e. through another attribute (faculty_id).

   c) Any non-BCNF scheme can be decomposed into BCNF schemes by factoring out the attribute(s) that are transitively-dependent on the primary key, and putting them into a new scheme along with the attribute(s) they depend on.

      Example: We decompose Section to

```
Section(department, course_number, section, days,
                time,room,faculty_id)
Professor(faculty_id, professor_name)
```

E. Beyond 3NF/BCNF there are further normal forms called 4NF, and 5NF that we won't discuss here, but do discuss in the DBMS course. We aim to ensure that every database design we produce is in the highest normal form (generally at least BCNF, but often higher).

   For now, we'll stop at BCNF - which can be summarized by the following rule:

   Every attribute depends on the key, the whole key, and nothing but the key