CS211 Lecture: Persistence; Introduction to Relational Databases

last revised October 30, 2006

Objectives:

1. To understand the need for persistent objects
2. To recognize alternative approaches to providing persistence:
   a. Explicit save/restore (perhaps using serialization)
   b. The above combined with logging of transactions
   c. A database systems
3. To understand the fundamentals of the relational model
4. To understand the concept of "key" (superkey, candidate, primary, foreign)

Materials:

1. Projectable version of simple database + second version of Advises table
2. Projectable version of schema diagram for simple database
3. Projectable version of schema diagram for lab database

I. Introduction

A. Thus far, almost everything we have done has involved objects that reside in main memory (RAM) on some computer. This means, of course that those objects "live" only while the program is running, and cease to exist when the program is terminated, either via normal exit or as a result of a system crash, power failure, etc.

   1. This is a consequence of the fact that the CPU can only directly manipulate information that is stored in main memory. Information stored elsewhere (e.g. on disk) must be brought into main memory before it can be manipulated.

   2. Note that access times for current main memory technologies is on the order of 60-70 ns. Access time for data on disk is on the order of 10 ms. Since 1 ms is 1 million ns, this is over a 100,000 to 1 ratio!

B. Obviously, for many applications this is not sufficient. We need some way to make certain objects PERSISTENT - to preserve them between runs of the program.

   EXAMPLE:

   In the registration database example we have used in several labs, there is no persistence mechanism - all courses start out empty when we first run the program, and enroll/drop/grade operations are lost when the program exits. Though we've used the program to illustrate many interesting concepts, as it stands right now it's actually useless!

EXAMPLE:

Which objects in our Video Store system need to be persistent?

ASK

C. Because this is so important, it turns out there are a number of ways of meeting this need.

1. The approach taken by many familiar applications, utilizing a File menu with New, Open, and Save options

   a) This approach is well-supported in Java.

      (1) If an object is an instance of a serializable class (one that declares that it implements the marker interface Serializable), it can be written to a file opened as an ObjectOutputStream using the stream's writeObject method, and the object can be read back from the same file using the readObject method of ObjectInputStream.

      (2) You used this approach in the last project in CS112.

      (3) Your Video store project also uses this approach - except that your "Open" and "Save" operations are done automatically at program startup/shutdown.

   b) However, this approach has very serious limitations.

      ASK

      (1) Data is saved only when the user explicitly uses the Save menu option, or at system shutdown, or - in some cases - automatically at regular intervals by an auto-save facility. If the program crashes or the power is lost, all work done since the last Save is lost.

      This may be acceptable for applications like a word processor, but is not actually unacceptable for recording transactions in a bank or a video store. (The way your semester project handles this would not be the way a real system would be built.)

      (2) It the stored database is large, then an “Open” or “Save” operation can take a great deal of time.

2. A second approach is to combine the Open/Save approach with some sort of LOGGING of transactions (either to paper or to some nonvolatile storage medium). If the system crashes, the log is used to redo transactions done since the crash.

However, this still doesn't address the problem of the time needed to load or store a large file, and redoing a log can be a problem in its own right.

3. A third approach is to make use of objects that reside on disk, with some or all of this information replicated in main memory where the program can use it, and that arrange to update the copy on disk whenever the copy in main memory is changed.

   This is the approach we will pursue in this series of lectures. Persistence in this sense is typically achieved by making use of some sort of on-disk database management system.

   a) Accessor operations on such an object actually get the requested information from the on-disk database - (though once it is gotten, a copy may be kept in main memory to avoid repeated trips to disk. (Remember the 100,000 : 1 access time ratio!)

   b) Mutator operations on such an object actually update the on-disk database.

II. Introduction to Relational Database Systems

A. At the outset, we should consider the question of what kind of database system we should use to support an object-oriented program.

1. There are database systems known as object-oriented databases, whose structure is object oriented. The entities stored in such databases are objects

2. However, most commercial databases use one of several non OO models, of which the most prevalent today is the RELATIONAL model. There are a whole host of good reasons for using a relational database to provide persistence for an OO program, even though the two models are different:

   a) Wide-scale availability of relational databases.

   b) Standardization - the relational model and the language most commonly used to access it are standardized.

   c) Legacy data - many organizations have large quantities of data already stored in relational databases, which it would be nice to be able to access from OO programs.

d) Ad-hoc queries: the relational model allows many operations on a database to be done interactively from a terminal session, without needing to write a specialized program. This facilitates extracting information from such a database as needed, without having to anticipate all possible queries and write software for them.

e) Solid mathematical underpinnings: the relational model is grounded in the mathematics of sets and relations, and thus has a sound theoretical basis which we can use to reason about the behavior of relational databases.

3. For this reason, we'll devote the rest of this series of lectures to
   a) Learning about the relational database model
   b) Learning how to access relational databases from Java programs.

4. This is a big topic. (We offer an entire course on DBMS's, and that just scratches the surface.) We will look at only a small subset of it.

B. We should note, at the outset, that the "OO world" and the "relational world" are two distinct worlds.

1. They have distinct histories
   a) OO: Comes out of the world of discrete simulation; much subsequent work arose motivated by the development of GUI's in the PC world - OO is the natural paradigm for designing a GUI.
   b) Relational Database systems: comes out of the world of business data processing; much of the work has been done in the mainframe world. Database systems historically have had a strong batch processing flavor.

2. They grew up in different parts of the world.
   a) OO originated in Norway, and is very strong in Europe
   b) Early work on relational databases was done by IBM and at Berkeley.

3. In a couple of places, they have very distinct ways of handling certain key issues - which we will refer to as we get to them.

4. Nonetheless, they also have some striking similarities, in some cases having discovered fairly similar solutions to certain problems.

5. In recent years, the relational database model has been evolving toward the OO model, with facilities being added that correspond to facilities present in OO (though still handled in a quite different way). For this series of lectures, however, we will deal with the "traditional" relational model.

6. For now, then, we will leave the OO world and enter the relational database world, with occasional glances back at OO.

C. While an OO model can represent everything about a system, a relational model focusses on just the entities comprising the system.

1. We can represent the static structure of an OO system by a class diagram, in which we include:
   a) Classes (which give rise to sets of objects). Each object, in turn, has identity, state, and behavior. A class may be an entity class, boundary class, or controller class. (Though sometimes we only deal with the first in class diagrams)
   b) Relationship between individual objects.
   c) Relationships between classes (generalization, realization, dependency).

2. The relational data model uses more limited mechanisms to represent:
   a) Entity sets - i .e. sets whose elements are (real or abstract) <u>things.</u> Entities have identity and state, but the basic relational model has no mechanism for representing behavior. Thus, a relational database typically does not deal with controller or boundary classes (for which behavior is the main thing).
   b) Relationships between entities.
   c) There is no mechanism (in the basic relational model) for representing relationships between sets of entities.

D. In a relational database, information is represented by <u>relations</u> (colloquially known as tables).

1. Some relations correspond to sets of entities. Each row in the table corresponds to one entity, and each column to one attribute.

   Example: The following relations (tables) might be used to represent students and faculty

   PROJECT

Student

studentID	lastName	firstName
1111111	Aardvark	Anthony
2222222	Cat	Charlene
3333333	Dog	Donna
4444444	Fox	Frederick
5555555	Gopher	Gertrude
6666666	Zebra	Zelda

Faculty

facultyID	lastName	firstName	livesIn
1	Bjork	Russell	Beverly
2	Brinton	Stephen	Hamilton
3	Crisman	Karl	Lynn
4	Levy	Irvin	Hamilton
5	Senning	Jonathan	Hamilton
6	Stout	Richard	Ipswich
7	Veatch	Michael	Danvers

2. A relation (table) can also be used to represent a relationship - that is, an association between entities.

   For example, the following table might represent the relationship "advises" between Faculty and Students

Advises

studentID	facultyID
1111111	1
2222222	2
3333333	1
3333333	5
...	

   Each row represents the existence of a relationship (association) between one student entity and one faculty entity - i.e. the first row indicates that Bjork advises Anthony Aardvark.

3. Note an important difference between OO systems and relational systems.

   a) In an OO system, entities and relationships are represented in totally different ways - the one by objects belonging to some class, the other by either a reference or a collection of references. In a relational system, entities are relationships are represented in exactly the same way - by relations (tables).

   b) But if the association itself has attributes, then in an OO system, it is objectified as a relationship class - i.e a very different representation is used. In a relational system, there is no change to way it is represented - e.g. if the "Advises" relationship included a "lengthOfTiime" attribute, it could be represented by a table like the following (which simply adds a column for the attribute):

Advises

studentID	facultyID	lengthOfTime
1111111	1	4
2222222	2	1
3333333	1	3
3333333	5	1
...		

PROJECT

III. The "Key" Concept

A. The example we just looked at illustrates a crucial notion in a database systems - the concept of a key.

   1. This concept is why we chose to use the studentID and facultyID to represent the fact that Bjork advises Anthony Aardvark, rather than the two names.

   2. We now explore this concept in detail.

B. In the relational model, a relation (table) is a set, in the mathematical sense - that is, each of its members must be distinct. This implies that the members of a relation must be DISTINGUISHABLE - there must be some difference among them whereby we can tell them apart.

   Normally, we expect that the value(s) of a single attribute or a group of attributes will suffice to distinguish one member of a relation from all others.

EXAMPLE: In the relation Student, one of the attributes is studentID. We ensure (when students are admitted) that each student has a studentID distinct from all others - though it might be that two students happen to have the same name.

C. Superkey - We call any set of attribute values which suffices to distinguish one member of an entity set from all others a superkey for that entity set.

1. In general, there can be many superkeys for a given entity set.

   EXAMPLE: For the relation Faculty, the facultyID is a superkey. But if we insist that no two faculty can have the same name, then so is lastName + firstName.

   EXAMPLE: For the entity set Faculty, livesIn is <u>not</u> a superkey.

   EXAMPLE: In the case of Student, studentID is a superkey. However, in general, we would not expect a student's name to be a superkey for the entity set - we can easily have two students with the same name.

2. Indeed, if a given set of attributes is a superkey, then any superset of those attributes is also a superkey.

   EXAMPLE: Since studentID is a superkey for Students, so is studentID + lastName.

3. Sometimes a superkey needs to be composite - i.e. to consist of more than one attribute

   EXAMPLE: In a library, books are identified by call number. But a library might have more than one copy of some book. In this case, each copy is given a unique copy number (copy 1, copy 2 ...) In this case, the attributes callNumber + copyNumber together are a superkey, though neither is by itself - we can't have books that have both the same callNumber and the same copyNumber (though we could have two copies of any one callNumber, or "copy 2" of two different books).

4. Note that - in almost every case - the complete set of attributes for an entity is a superkey for the entity set. All of the examples we work with in this course will have this property.

   In many cases, this is uninteresting, though we will encounter entity sets for which the full set of attributes is the only possible superkey.

5. Finally, note that the notion of superkey is determined by the logic of the database scheme, not by a particular instance.

EXAMPLE: Suppose that, at a certain point in time, a college had no two students with the same name. That would not make lastName + firstName a superkey for Student, because there is no inherent reason why two students couldn't have the same lastName + firstName.

D. Candidate key - a candidate key is a superkey which has no proper subsets that are also superkeys - i.e. it is, in some sense minimal.

EXAMPLE: For our college example, for Faculty, both facultyID and lastName + firstName are candidate keys (assuming we disallow having two faculty with the same name). However, the combination facultyID + livesIn, though a superkey, is not a candidate key.

E. Primary key - the primary key of an entity set is a particular candidate key chosen by the database designer as the basis for uniquely identifying entities in the entity set.

1. Candidate keys are called that because they are candidates for being chosen as the primary key.
2. If a given entity set has only one candidate key, then the choice of primary key is obvious. But if there are multiple candidate keys, the database designer must choose one to be the primary key.

   EXAMPLE: For Faculty, if we disallowed having two faculty with the same name, we could choose either facultyID or lastName + firstName as the primary key. (In this case, we would almost certainly choose facultyID, because it's simpler and wouldn't "break" if we did hire two faculty with the same name.)

3. Note that it's often the case that, in establishing the attributes for an entity, we include some "ID" attribute that is a natural choice for primary key - e.g. studentID, facultyID.
4. As a matter of good design, we always identify a primary key for each relation (table) we create in a relational database. Often, the primary key attribute(s) is/are underlined when describing a relation scheme.

   EXAMPLE: The attributes of Faculty could be represented as facultyID, lastName, firstName, livesIn.

F. What about relations (tables) that represent relationships? Since relations are always sets, they, too, need a primary key.

1. The primary key of a relation that represents a relationship between entity sets is generally the union of the primary key attributes of all of the entity sets it relates.

EXAMPLE: If studentID is the primary key for Student, and facultyID is the primary key for Faculty, then studentID + facultyID is the primary key for Advises.

Note that there are multiple rows for studentID 3333333 and for facultyID 1, but there can be only one row for studentID 3333333 and facultyID 1.

2. The primary key of a relation (table) representing a relationship is thus typically composite - and may be the whole scheme (if the relationship has no attributes of its own).

3. An exception to this statement occurs in the case of a relationship that is 1 to anything (or 0..1 to anything). In this case, the primary key of the other entity is also, by itself, the primary key for the relationship, since no two rows could contain the same value.

   EXAMPLE: Suppose we represented information about a library by entities Borrower (primary key borrowerID) and Book (primary key callNumber + copyNumber), together with a CheckedOut relationship (with columns borrowerID, callNumber, copyNumber and dateDue). By the "primary key of a relationship is the union of the primary keys of the entities" rule, the primary key for CheckedOut should be borrowerID + callNumber + copyNumber. But since the relationship is 1 .. many from Borrowers to Books (a book can only be checked out to one person at a time), callNumber + copyNumber alone is a candidate key and therefore the primary key.

4. Note that, while we sometimes have a choice to make concerning the primary key for an entity set (if there are several candidate keys), we have no such choice to make for a relationship set - once the primary keys of the participating entities are specified, so is the primary key of the relationship set.

G. When a relation (table) includes the primary key of another relaton (table), the included key is called a FOREIGN KEY.

1. Foreign keys necessarily occur in relations (tables) representing relationships

   EXAMPLE: In the Advises table, borrowerID is a foreign key and facultyID is a foreign key.

2. Foreign keys can occur in other relations (tables) as well.

   For example, suppose we have a Departments table, with each department identified by a department code (e.g. CS). Suppose further

that we include a “firstMajor” attribute in our Student table. Of necessity, the value of this attribute must be a valid department code (you can’t major in ZZ at Gordon) - so we consider this attribute to be a foreign key.

H. It is with regard to keys that there is a fundamental distinction between the OO model and the relational model.

1. In the OO model, objects have identity, state, and behavior. In particular, we insist that two different objects can have distinct identity, even if their state happens to be the same.

   EXAMPLE: Suppose two different people have bank accounts that both happen to have a balance of $100. Those are still distinct bank accounts, even though their state is the same.

2. In the relational model, an entity set is a set. Therefore, two different entities are not allowed to have identical attributes - they must at least differ in their primary key.

3. This point of divergence is often moot, because in designing either an OO system or a relational database we often include an attribute in each entity precisely for the purpose of serving as a unique identifier - either a naturally occurring one (e.g. SSN) or one created for that purpose (e.g. a Gordon College student ID).

4. Actually, in an OO system each object does have an "attribute" that that is always unique. What is it?

   ASK

   The location in memory where the object is stored - represented by the reserved word this

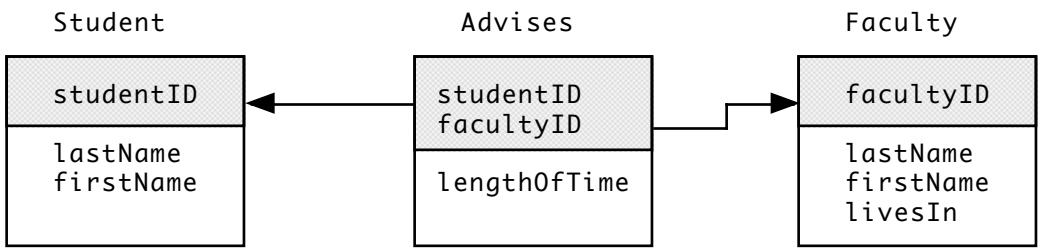
   (But we don't really regard this as an attribute or store it persistently.)

IV. Schema Diagrams

A. The structure of a relational database is called its schema.

B. A schema can be depicted graphically using a schema diagram. In some sense, a schema diagram plays a role similar to that of a class diagram for an OO system.

C. The following is a schema diagram for the very simple database we have been using for examples:



PROJECT

1. Each box represents a relation (table). Each relation (table) has a name.

2. Each name inside the box represents an attribute (column in the table).

3. The box is divided by a horizontal line. The attributes above the line (in the shaded section of the box) constitute the primary key. (If there are two or more attributes, then the primary key is composite). Those below the line are not part of the primary key.

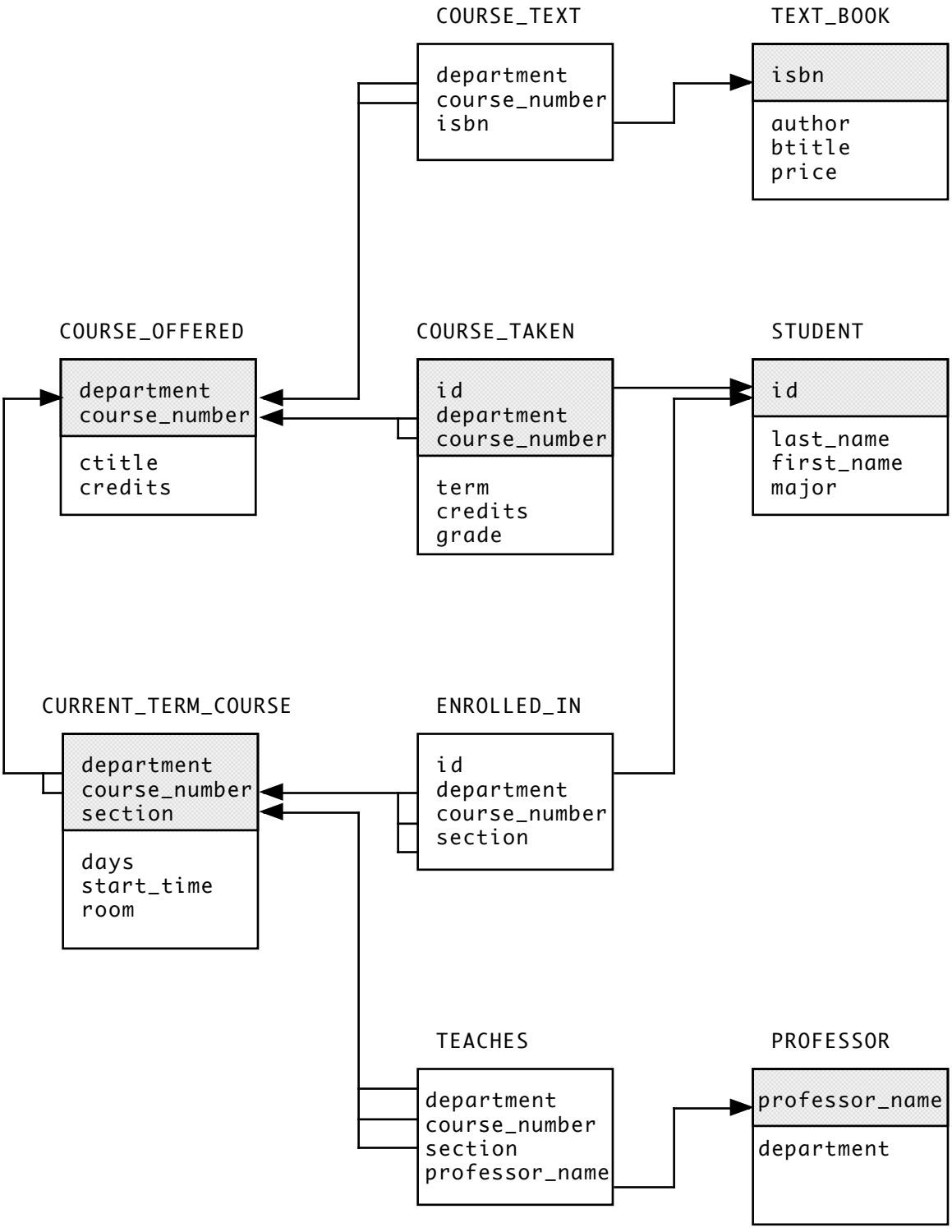
   If a relation is "all key" (i.e. every attribute is part of the primary key), the box is not divided at all - hence a box without a horizontal line represents an "all key" relation.

4. Each foreign key is connected by an arrow to the place where the key is defined, typically as (part of) the primary key of some other relation.

D. The following is a more complicated schema diagram - for the database we will be using in lab.

   Note: This schema assumes that "professor_name" is a superkey - which is, in fact, the way Gordon's current system works. (A professor_name may be something like DWEES-BOYD, I or DWEES-BOYD, M to distinguish Ian DeWeese-Body from Margie DeWeese-Boyd.

   PROJECT; GO OVER


COURSE_TEXT
department
course_number
isbn
TEXT_BOOK
isbn
author
btitle
price
COURSE_OFFERED
department
course_number
ctitle
credits
COURSE_TAKEN
id
department
course_number
term
credits
grade
STUDENT
id
last_name
first_name
major
CURRENT_TERM_COURSE
department
course_number
section
days
start_time
room
ENROLLED_IN
id
department
course_number
section
TEACHES
department
course_number
section
professor_name
PROFESSOR
professor_name
department

V. Some other Differences between OO and Relational Systems

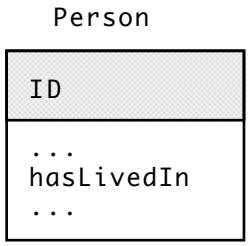
A. We have already seen some key differences between OO and relational systems.

1. The representation of both entities and relationships using the same basic approach: tables.
2. The "key" concept.
3. The related notion that identity is established by values of attributes.
4. We now consider a few more.

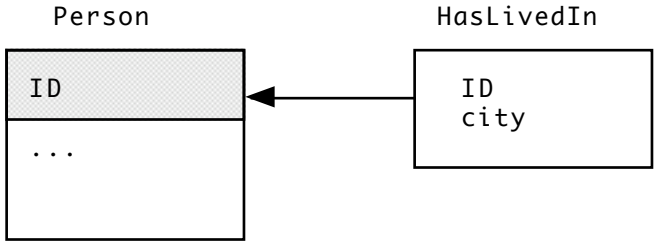
B. The relational model requires attributes to be simple, atomic, single values. This is not a requirement in some other database models, nor is it a requirement in OO, and represents one place where some work may need to be done when using a relational database with an OO system.

1. EXAMPLE: If an address is stored as street address, city, state, and zip, then a table cannot have an "address" column. Instead, it must have streetAddress, city, state, and zip columns. (Attributes cannot be composite).
2. EXAMPLE: if one of the pieces of information we store about a person is the cities in which they have lived, we cannot have a "cities" column in a table describing the person. Instead, we must use a separate table representing the relationship between the person and the various cities the person has lived in - e.g.

   <u>not:</u>



   but rather:



C. Sometimes, we will not know the value of a particular attribute for a particular entity, or it somehow does not apply in a particular case - in which case the value of that attribute is said to be NULL.

EXAMPLE:

Course grades are assigned at the end of a semester. Thus, while a course is in progress, there will be a row in the EnrolledIn table for each Student in the course, but the grade attribute in each row will be NULL.

1. Note that this appears similar to the concept of a null reference in a language such as Java. But it is not the same.

2. One property of NULL is that it is never treated as any value during a query - the principle being that, since NULL means we don't know a value, it should never participate in any query.

   EXAMPLE:

   Suppose we are calculating the GPA for a student. The NULL grades for courses the student is taking now don't count. (Notice that this gives quite a different result than what we would get if we took the grade to be 0!)

   (Contrast this with the null pointer exception you might get in Java if you tried to access an attribute whose value is null!)

3. If two different entities have the value NULL for some attribute, the two attributes are not considered equal - i.e. NULL never equals anything - even NULL.

D. Sometimes, a given attribute can be calculated from other information in the database - in which case, instead of storing it we may compute it upon demand. Such an attribute is called a DERIVED attribute.

EXAMPLE:

In a registration system, we may have tables Student, Course, and EnrolledIn. We may wish to include a "total enrolled" attribute for the Course table. This does not need to be an ordinary stored attribute; it can be a derived attribute that is calculated when needed by counting the number of EnrolledIn rows having that course's ID in them. (In a case like this, use of a derived attribute is not only a convenience, but also a good design practice - since it prevents the possibility of having inconsistent stored results if a row is inserted into/deleted from the EnrolledIn table but the enrollment count for the course is not updated.)