Contents

CMPE2400 – Databases – Notes V

Database Design

Database design is a tricky topic. In order to effectively design, normalize, and implement a database, many details must be known, and the process benefits from experience. Obviously you can't get experience with design until you've gone through the process at least once, so that is what you will do. You will work through the process of analysis, design, and implementation of a database, with guidance.

The database backend that you will create will be based on a repair system, and here are some of the details about the system:

Units that need repair are put through a diagnostic. The diagnostic is performed by a technician. The diagnostic produces zero or more repair orders, each detailing a specific repair requirement. Each repair is carried out by one or more technicians. Every action taken must be correctly dated. Several reports will need to be run that will detail what work was done, the technicians that did the work, and on what units. Additional details on the reports will be revealed later.

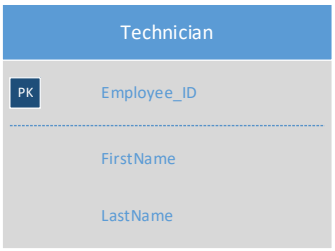
The first step is some kind of analysis of the problem you are attempting to solve. This is normally accomplished with something like an entity relationship model. An entity relationship model is a high-level, abstract description of information and processes, and consists of entities and the relationships between them. Complete E-R modeling and diagramming is somewhat beyond the scope of this course, but simplified E-R modeling will help you resolve the details of the design.

Identify all of the entities in the system. An entity is something that exists physically or logically, and can be considered a 'thing'. An entity can be thought of as a noun, like an employee, a desk, or anything that can be uniquely identified. In the case of the above description, the entities will be things like the units to be repaired, the technicians, the repair orders, and others.

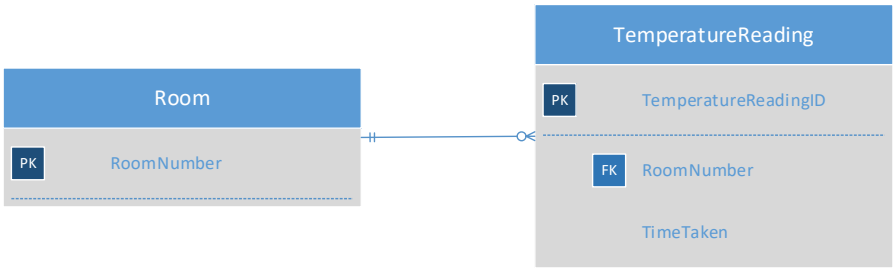
Once all of the entities in the system have been identified, all of the attributes of each entity need to be identified. Entities are eventually going to become tables in the database, and the attributes will become the columns. For an entity like the technician, this would include a name, employee ID, and others. Ensure that all attributes are broken down to the atomic level. This means that things like names are stored as first and last as separate attributes.

Next, you must identify primary keys for each entity. Remember that primary keys must be unique. If a natural primary key exists, like an employee ID or a social insurance number, use it. You may need to create a primary key from other attributes, or use other mechanisms (more on this later) to generate a primary key.

Once all of the entities, their attributes, and primary keys have been identified, you may now start drawing a simplified E-R diagram. Primary keys are underlined in E-R diagrams (unless there is another obvious indication of the primary key). A technician, for example, might look something like this:



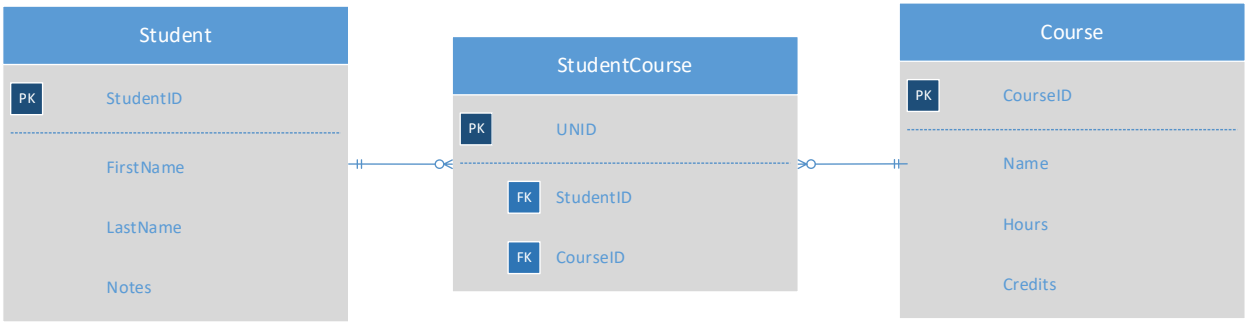
At this point, you need to determine what relationships each entity has with other entities, and their cardinality. For example, consider the case where you have identified your entities as rooms and temperature readings. A room can have zero or more temperature readings, but a temperature reading applies to exactly one room.



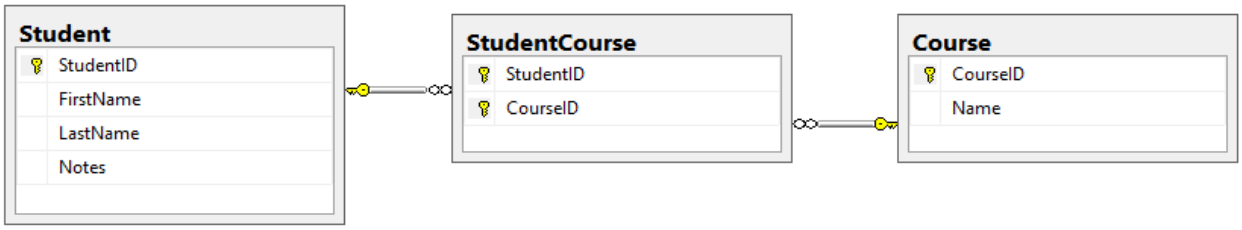
The above *Crow's foot* notation shows this as 'A room may have zero, one, or many temperature readings, but a temperature reading applies to exactly one room'.

Once the 'one-to-many' relationship is found and validated, include a foreign key in the entity (the many) you wish to join to the primary key entity (the one).

In the case of a 'many-to-many' relationship, a connecting table will be required. The sample database used in these notes is an example of this. One student can take many courses, and a course can be taken by many students:



In this case, you have the option of making the primary key in the 'StudentCourse' table a composite key, made from the two foreign keys. Since keys must be unique, it would become impossible to have the same student in the same course twice, if this is something you wanted to avoid.



Any ‘one-to-one’ relationships you discover should be collapsed into a single table.

Normalization

Normalization is a systematic process of optimizing a database. Note that the optimization is related to organization, reduction of data redundancy, flexibility, and data consistency. Normalization typically does not result in increased database performance, in fact, the opposite is typically observed.

Why normalize? Consider the data in a spreadsheet, which is often a good example of non-normalized data:

StudentID	Name	Address	Course
1000	Smith	123 Coven Street	CMPE1300
1001	Doe	#4 10132 44th Street	CMPE1500
1002	Smit	3458 106th Street	CMPE1000
1000	Smith	123 Coven Street	CMPE1000

There are several problems with storing data in this way:

- Update Anomaly – Changing the address for a student that is taking more than one course could yield different addresses for the same student.
- Insertion Anomaly – Adding a student that is not yet in any courses will require NULL to be inserted into the *Course* column.
- Deletion Anomaly – If is student is taking one course and drops it, the entire row could be deleted, losing the entirety of the student information along with it.

Normalization is described as a series of forms, and there are many. We will only consider normalization to the 3rd normal form. The normalization forms are:

1NF – First Normal Form

The first normal form requires:

- Break out non-atomic elements (some discretion is required)
- Creation of separate tables for each set of related data
- Identification of each set of related data with a primary key
- Elimination of repeating groups in individual tables

In short, to meet 1NF, each row must be unique, there must be a primary key (one or more columns), and no column may have more than one value. The following data, for example, is not in 1NF:

Employee Table		
Employee	Shoe Size	Trade
Mike	10	Plumbing, Roofing
Bob	9	Electrical
Joe	12	Plumbing

In 1NF form, the table would look like this:

Employee Table		
Employee	Shoe Size	Trade
Mike	10	Plumbing
Bob	9	Electrical
Joe	12	Plumbing
Mike	10	Roofing

In 1NF, there is additional data duplication, but each column contains only one value.

The first normal form typically requires each attribute to be atomic, meaning any attribute that can be decomposed to other attributes should be. There is a limit to how far you should take this, and typically it will mean that names, for example, will contain separate attributes for first and last.

The key for the above table could be *Employee + Trade*, as these together would uniquely identify each row.

In the case of a repeating group violation, extract the repeating elements into a new table. Also pull the original primary key as the foreign key for the new table. A new primary key is necessary for the new table, and as natural keys are preferred over surrogate (made-up, ie. identity) keys, the new primary key is usually a composite key comprised of the foreign key and a unique attribute from the repeating group.

2NF – Second Normal Form

The second normal form requires that a table be in 1NF, and every non-key attribute be dependent on the whole of every candidate key. Or, in other words, there must not be any partial dependency of any non-key column on any candidate keys.

In other words, 2NF will only have possible violations if it contains a composite primary key. If no composite keys exist, there are no 2NF violations. Otherwise, on tables with a composite key – all non-key attributes must depend on all the keys. If they don't, a violation exists. To correct this violation, pull out the attributes related to the partial key placing them into a new table using the partial key as the new primary key.

In the above table, *Shoe Size* is dependent on *Employee*, but not *Trade*. So, *Shoe Size* is not entirely dependent on the *Employee + Trade key*. This partial dependency makes the table fail 2NF.

To solve this problem, the *Trade* could be broken out to a separate table, and linked back to the *Employee* by name as a foreign key. This would result in the following two tables:

Employee Table	
Employee	Shoe Size
Mike	10
Bob	9
Joe	12

Trade Table	
Employee	Trade
Mike	Plumbing
Bob	Electrical
Joe	Plumbing
Mike	Roofing

In the employee table, all non-key columns (*Shoe Size*) are entirely dependent on the primary key, so this table is now 2NF.

The trade table is also 2NF, as there are no non-key columns. This table still requires *Employee + Trade* to serve as the primary key.

3NF – Third Normal Form

The third normal form requires that the table be 2NF compliant, and no *transitive functional dependencies* exist. A transitive dependency exists when one column is dependent on another column. Consider birthdate and age being represented in separate columns. Age is dependent on birthdate, and having both columns in the table means that updating one could lead to disagreement between the two. This is a good example to understand a transitive dependency, but not very practical, as age would likely be calculated, not entered.

Consider the case of columns that describe a city and postal code. The city would be dependent on the postal code, as a postal code will only appear in one city, or in other words, the postal code dictates the city. Again, updating one and not the other could result in values that disagree. To solve this, the postal code could be placed in a separate table as a primary key with the city as a column that fully depends on it. The postal code would then be a foreign key in the first table. Again, not very practical, as you would not put every city/postal code combination into a table.

A great 3NF example can be found at https://en.wikipedia.org/wiki/Third_normal_form

So, in summary:

- 1NF
  - Table has primary keys
  - Non-Atomic attributes are broken out
  - Repeating groups are extracted into new tables

- 2NF – No non-key dependencies
  - Only Composite Primary Key tables need be considered
  - Non-key attributes not dependent on entire key are extracted to new table

- 3NF – No transitive dependencies
  - Non-key attributes related to other non-key attributes are extracted to a new table