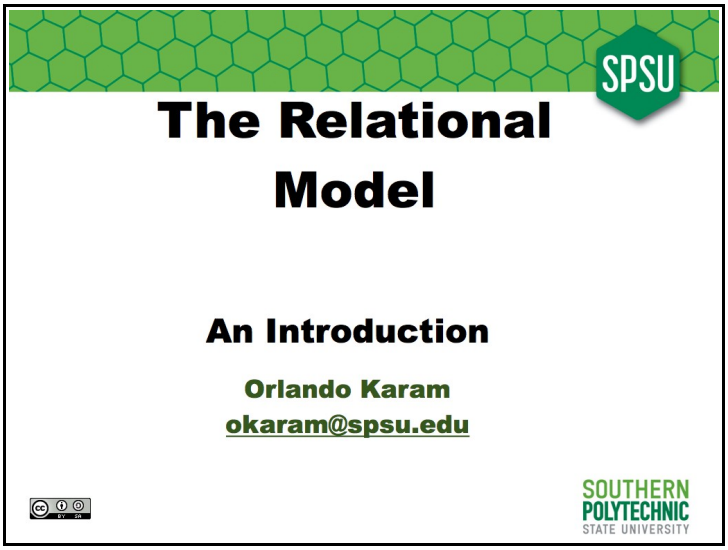
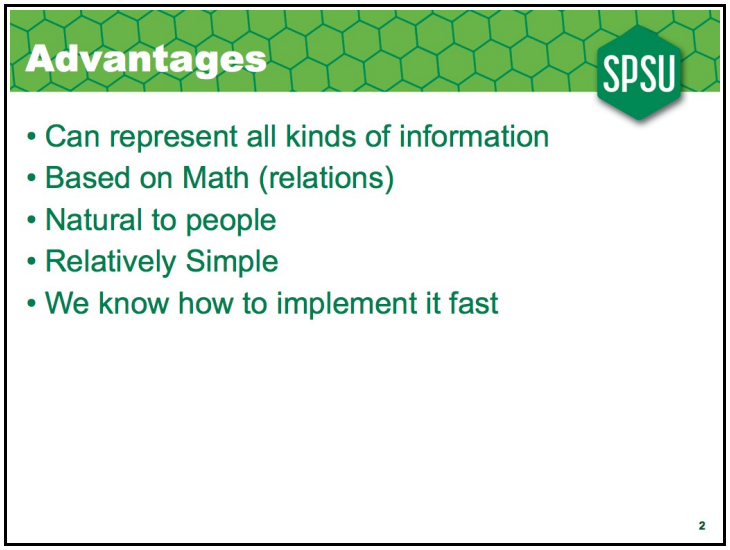


The relational model is by far the most used model for databases; although it is of a slightly lower level than the ER model, it is almost always used to implement databases.

The model is so successful because it is very flexible, and able to represent information of any kind; it also has a solid mathematical foundation, being based on the concept of a relation, which is a special kind of set.

The relational model also feel natural for most people, and it is relatively simple; also, starting in the 80s, we learned how to implement if very fast, at least fast enough for most applications.

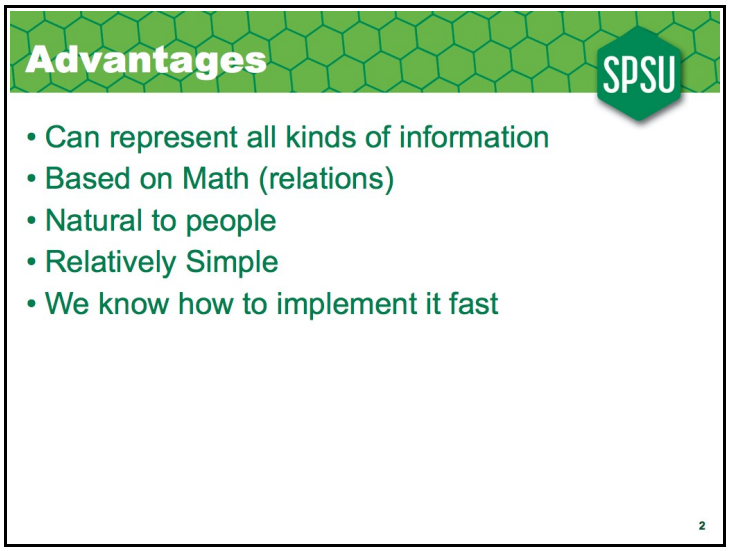


A data model entails the data structures used, the operations we can do on those structures, and the integrity constraints imposed on data.

In the relational data model, the only data structure is the relation or table, which is a set of tuples or records.

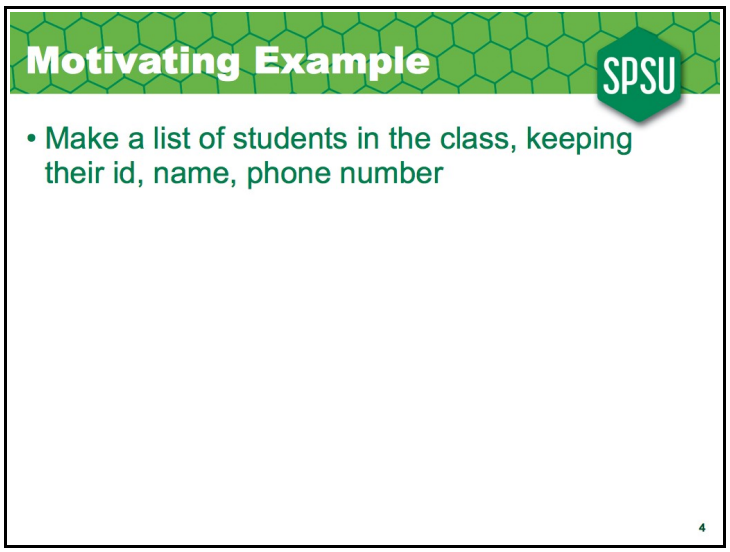
The relational data model supports many operations on these tables, corresponding to the basic CRUD operations; you can insert tuples into a table, update or delete some rows, and read (or select) certain rows; a more complete set of operations is offered by the relational algebra, and SQL.

The model supports a wide range of integrity constraints, allowing us to represent all kinds of data.



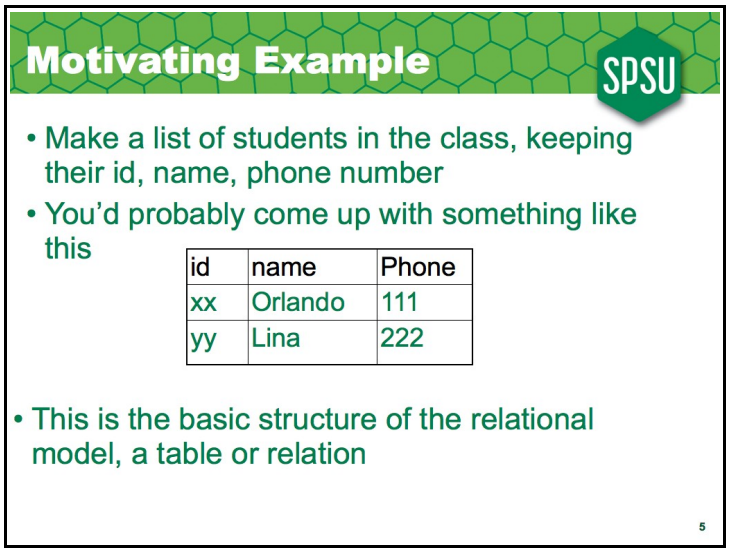
To show how natural is the relational model, lets go to an example; imagine you're asked to create a list of the students in the class, keeping their id, name and phone number.

Go ahead, and create the structure for such a list, and add some sample data.



Chances are, you came up with something like this; this is the basic structure of the relational model, and it is called a *table*, or a *relation*. We call each of the rows a tuple (or also a row). The columns are called attributes or fields.

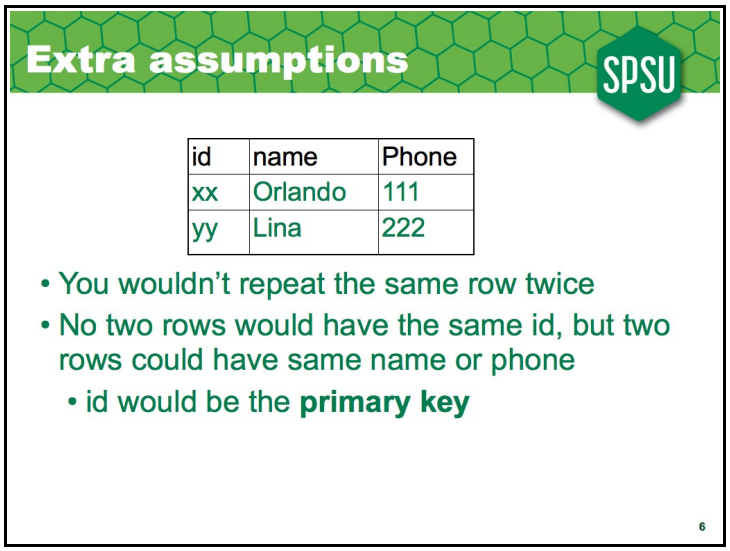
Notice that the first row gives the name of the attributes, and that each row is of the same kind, it has the same kind of information.



There is also a few more assumptions, that is good to make explicit.

First, you wouldn't repeat the exact same row twice, right ? That wouldn't make sense; when we formalize the model, we will say that a table is a *set* of rows or tuples, and so it doesn't allow duplicates.

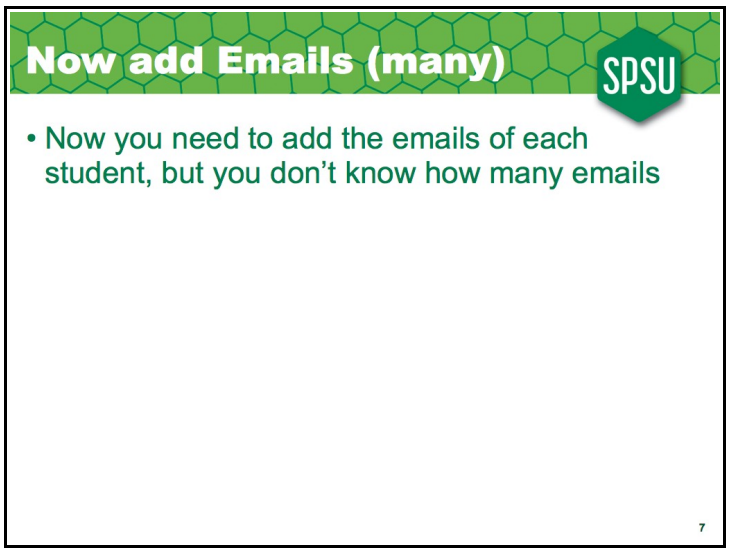
Also, you wouldn't expect two students to have the same id, right ? But there could be two students with the same name, or phone number; eventually, we will make the fact that no two students can have the same id a constraint, so that the database will reject wrong data. We would say that id is the primary key of our table.



In the relational model, each table is required to have a primary key.

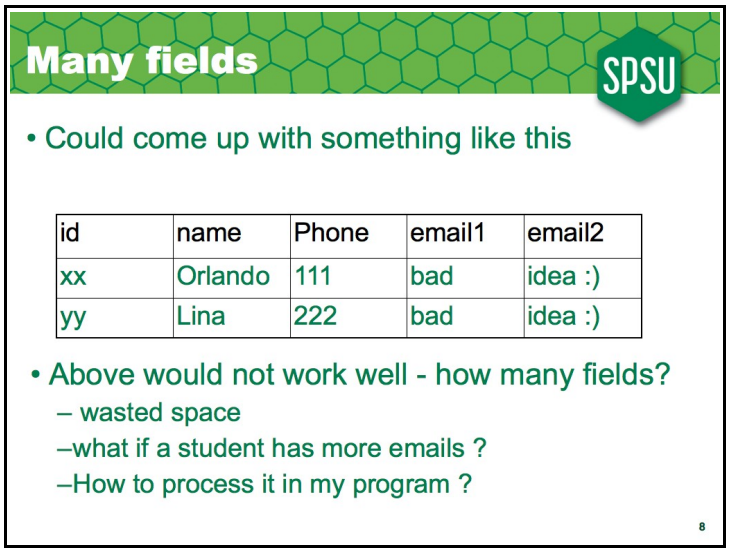
Now, let's assume that you're asked to also keep track of student's emails, but a student can have more than one email, and you're asked to keep *all* emails, regardless of how many a student has; moreover, you don't know how many emails could a student have.

Can you come up with a solution ? Try it …



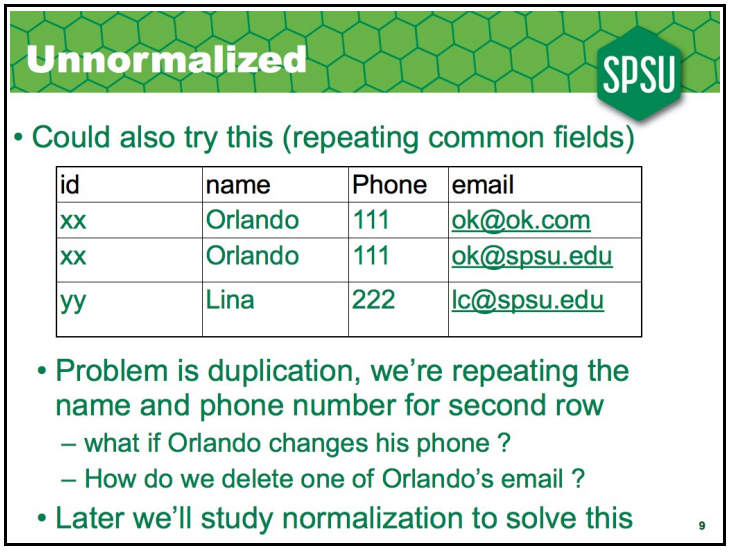
You could try something like this, adding several email fields, calling them email1, email2 et cetera; this is not a good solution for a variety of reasons; first, it potentially wastes space, since different students have a different number of emails; also, you'd need to modify your schema every time a student comes that has more emails than the ones you considered; modifying the schema is a more complicated operation that just adding data.

Finally, when you create database applications, each field is accessed separately, which means you'd need to write code to access each one of those email fields, in a way, this would be as, in your program, using variables email1, email2 et cetera instead of using an array.

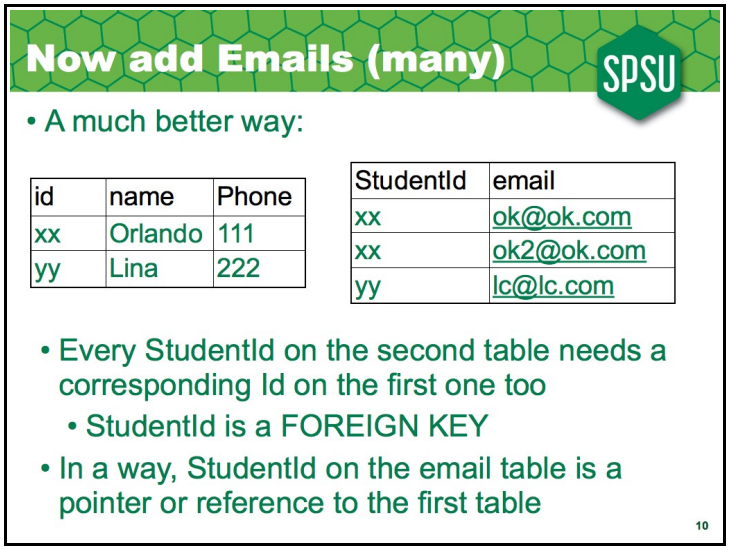


You may also come up with something like this, where we have only one email field, but we will have several rows on our table for each person, depending on the number of emails they have; this is problematic as well.

The main problem here is duplication of data; we're repeating the name and phone number of a person several times, depending on how many emails they have; this, besides wasting space introduces consistency problems; if we need to change a student's name, we may need to update several rows, and similarly for other changes to the table.



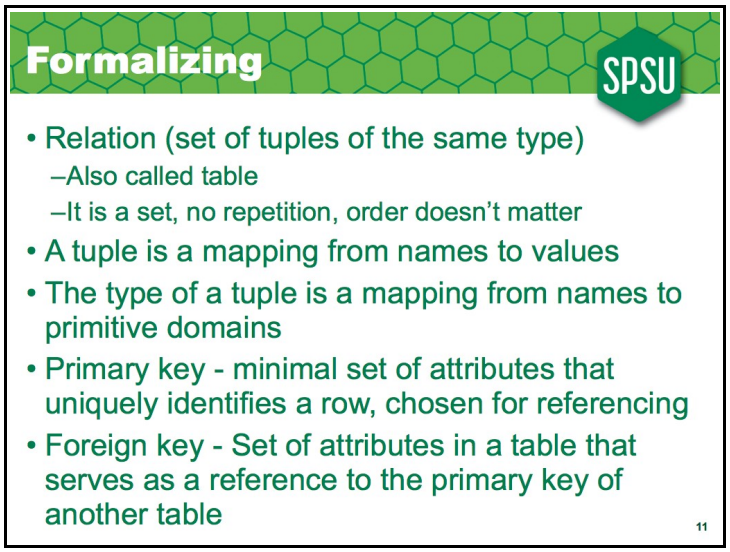
A better way involves dividing the information into two tables, one keeping the id, name and phone, and another table, keeping just the id of the student and the emails. Since we now have two tables, it is convenient to give them a name; we'll call the first table, the student table and the second one the emails table.



Notice that in the Student table, id is the primary key, it cannot be repeated; however, in the emails table, the studentId is not the primary key, since if a student has several emails there will be more than one row with the same studentId; without extra information, we cannot be sure two students don't share the same email, so we cannot use email as the primary key; the only choice is to use the combination of studentId and emails as the primary key for the emails table.

Notice that the StudentId field of the Emails table is strongly related with the id field on the student table; in fact, every value of StudentId needs to be present in the Student table for it to make sense; we say that StudentId is a foreign key, referencing the id field of the student table. The StudentId field is kind of a pointer or reference to the id field on the student table, allowing us to find the name and phone number of that student.

Now that we have an intuitive idea, lets formalize some of the definitions. A relation, or table is a set of tuples or rows, all of the same type. Notice that a relation is a set, so the same tuple cannot appear twice, and the order of tuples is irrelevant.



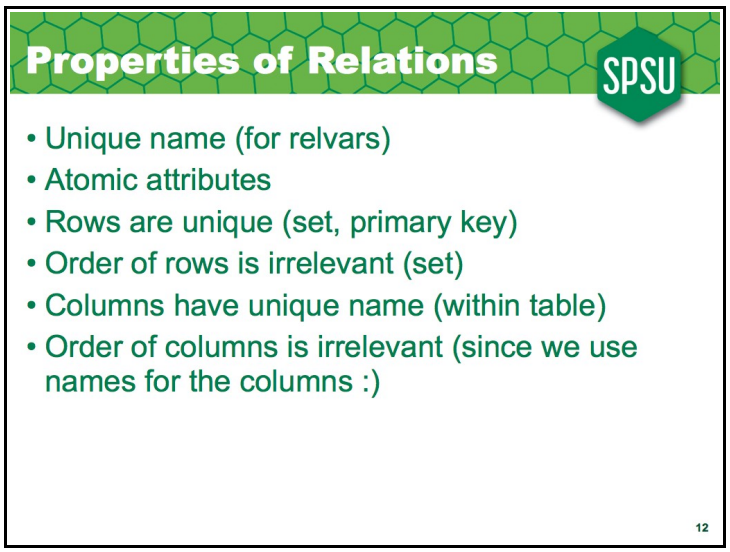
A tuple is a mapping, or a partial function, from names to values; for each name it gives you a value (or the fact that it has no value). You can also think of it as a set of pairs, each pair having a name and a value.

We also need to define what is a tuple type, which is a mapping from names to domains (a domain is a set of values); and a tuple is of that type if its values correspond to the domains on that type.

The primary key of a table is a set of attributes that uniquely identifies the rows in the table; that is, no two rows can have the same value for all of those attributes. Normally, the primary key is just one attribute, but it is defined as a set, so it can contain more than one attribute; the primary key will be the combination of several attributes.

Finally, a foreign key is a set of attributes in one table that serves as a reference to the primary key of another table; foreign keys allow us to get to the other fields from the other table; notice that the value of a foreign key field has to exist on the other table, the one it references.

Now let's formally specify the properties of relations in the relational model.



Relations in a database schema will need a name, that is unique, meaning different from all other relations in that schema. Notice that we will eventually see operations that act on relations to generate other relations; these new relations are usually only temporary, not part of the schema, and are not given a name. Date, one of the giants in relational databases calls the relations in a schema relvars, for relational variables, and the values relvals.

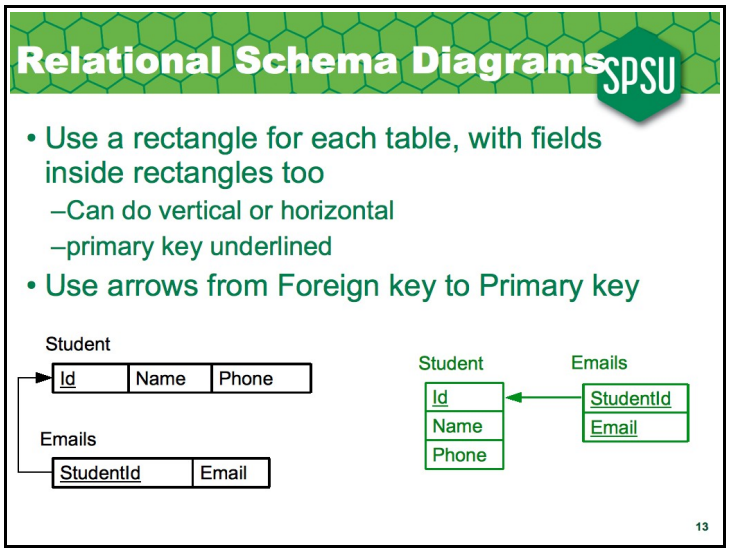
The next property is that the attributes in a table are atomic, they cannot be multivalued nor composite.

Since relations are sets, the rows in a table have to be unique; also, we have the primary key which makes a stronger uniqueness constraint. Again, since relations are set, the order of the elements of the set, the rows in the table, are not relevant. Notice that when we display the table, we will display it in a particular order, but we consider the information to be the same, even if we alter that ordering.

Since we need to distinguish among columns, we need each column within a table to have a different name, although columns of different tables might have the same names.

Since we use names to distinguish among the columns, their order is not relevant.

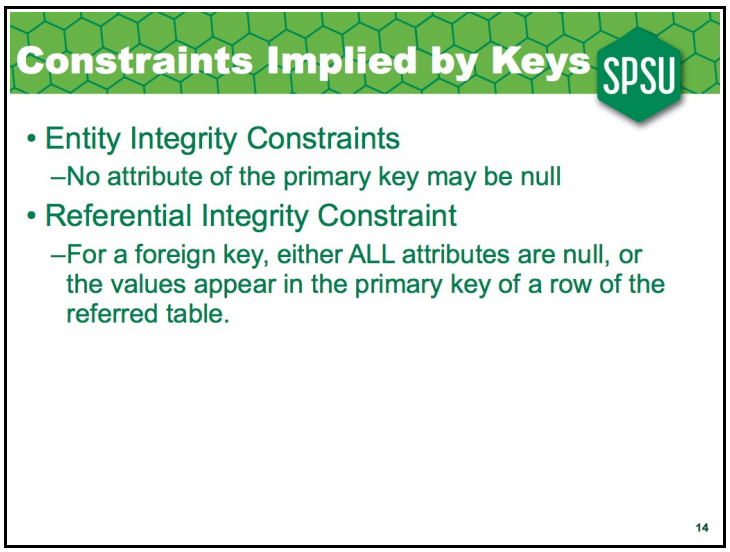
We will sometimes use diagrams to represent relational schemas (later we will learn SQL, and many people become so comfortable with SQL that they can read and assimilate SQL code so quickly that these diagrams are not useful anymore, except for complicated situations, where the diagrams can help.



When creating these relational schema diagrams, we represent each attribute with a rectangle, and write its name in the rectangle's center; we underline any attributes that belong to the primary key (again, there's only one primary key, but it is a set of attributes, so it may contain more than one attribute); We then join all the attributes into a bigger rectangle to represent the table, and we write the name of the table outside it. We use arrows to represent foreign key constraints; the arrow always goes from the foreign key to the primary key.

When we draw a table, we can put the attributes horizontally, making each table along rectangle, like the diagram on the left, or vertically, making tall rectangles like on the right; it is acceptable either way.

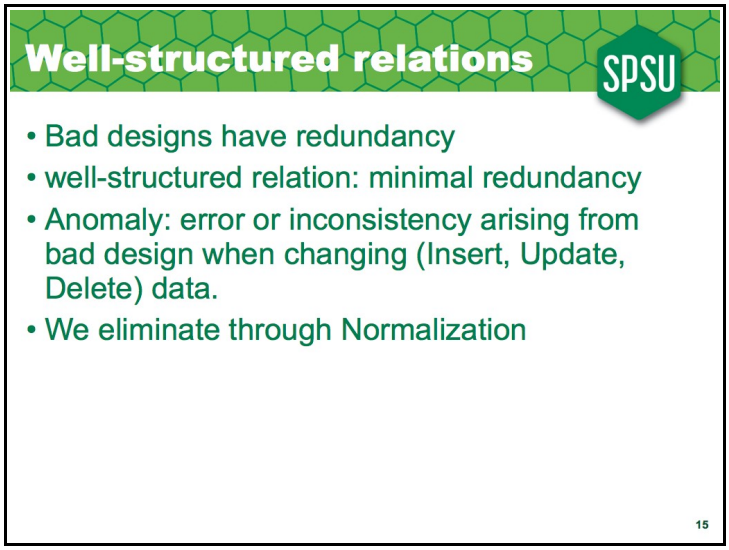
The primary and foreign keys of a relation come with implied constraints; first, there can be no row with null in any attribute of the primary key, since we will use the primary key to identify the row; we call this the Entity Integrity constraint; For foreign keys, either the whole foreign key is null (meaning we have no value for it) or the values for the foreign key have to appear as the primary key of one row of the referenced table. Notice that both the primary key of a table and the foreign keys are *sets* of attributes, although in these examples they've always been *singletons*, that is, sets containing only one attribute.



Depending on how we design our relations, we may introduce unwanted redundancy; we say that a relation is well-structured if it does have minimal redundancy.

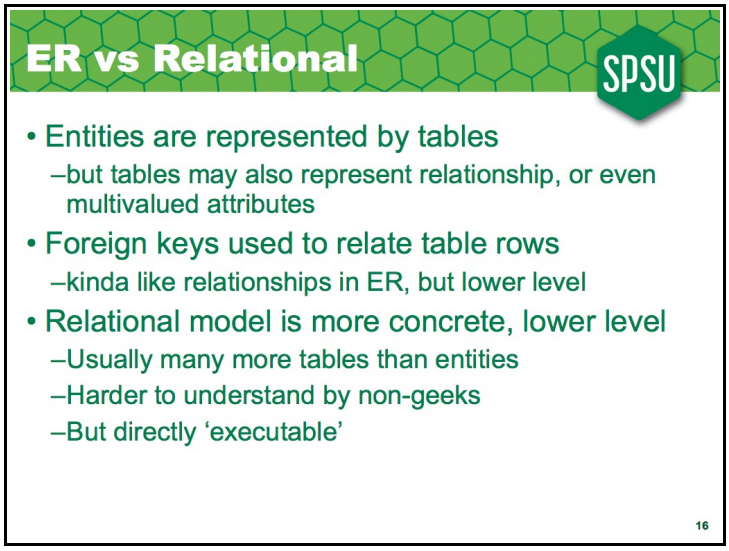
If a relation has redundancies (like some of the wrong answers to how to add multiple emails), it can introduce anomalies; that is, when we change the table (by inserting, updating or deleting rows) we either have to do convoluted operations or we introduce inconsistencies in our data.

This anomalies have been widely studied, and gave rise to several ways of normalizing our schema, so as to avoid this problems. We will study normalization later in this course.



Now, hopefully you understand the ER model now, and so comparing it with the relational model can help you understand it. Entities in the ER model will become tables, but tables may represent other things beside entities. In the ER model, we use relationships to associate entities; in the relational model, we use foreign keys to associate **tables**, but the tables don't always represent entities.

Finally, the relational model is less abstract, which usually means there are more tables than entities in the corresponding ER model, and that the ER model will be easier to



understand by 'normal' people; on the other hand, the relational model is directly implementable, which makes it easier for programmers.