THE KEYS TO GOOD KEYS

COMP 1140: Database and SQL | Brendan Shea, PhD (Brendan.Shea@rctc.edu)

In this lesson, we'll be answering the following questions:

1. What makes for a good primary key? What are the different types of primary keys?
2. How can you choose an appropriate primary key for a given entity?
3. How do you model 1:1 relationships? What about "time variant" data?
4. What is a **fan trap,** and how can you avoid them?

PICKING PRIMARY KEYS: SOME RULES OF THUMB

In previous lessons, we've said that a *primary key* is an attribute (or set of attributes) that serve to fully determine all of the other attributes of the entity. That is, if you know the primary key, you can get ALL of the other information we know about the entity. More technically, we said that a primary key is a "minimal superkey", in that it (1) does not contain any "unnecessary" attributes other than those it *needs* and (2) uniquely determines all of the other attributes in the relation.

Part of the beauty of relational DBMS software is they will simply NOT ALLOW you to create tables the lack primary keys, or to insert multiple entities that have the same primary key. Yay! That being said, not all "legal" choices of primary keys are good ones. With this it mind, here are a few rules of thumb that must be followed:

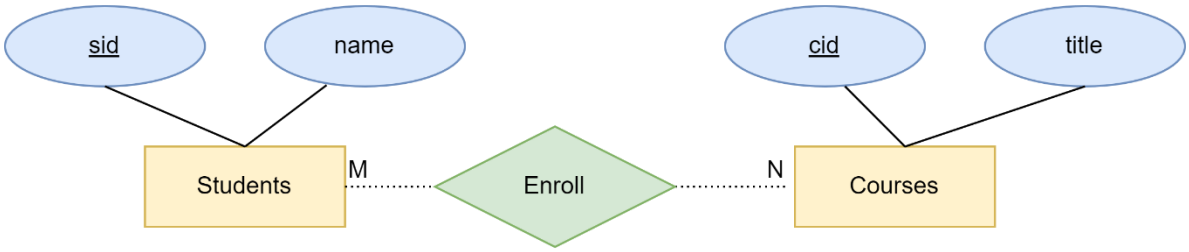
1. **Be Something Unique.** Ok, so this one isn't really a "rule of thumb", but a requirement of the relational model. Primary keys are not allowed to be NULL (they must be "something") and you can't have two entities with the same primary key ("unique").
2. **Live Naturally (When You Can).** If the entity type that you are modeling has a **"natural key"** that is used to identify it in the real world, this can SOMETIMES be used as a primary key. For example, if you are creating a database for a company where employees have already been assigned employee id numbers, these could also make great primary keys.
    - You should use natural keys as primary keys only if doing so doesn't break any of the *other* rules laid out here. If there isn't a natural key (or if the natural key won't work), you'll need to create an artificial **surrogate key** to use as a primary key. For example, we might *create* an attribute "id_number" even if no such thing exists right now (and even if it has no meaning/use outside of the database itself).
3. **You Mean Nothing to Me, and This Will Never Change.** While it may seem odd, we generally do NOT want primary keys that have any "meaning" (or **"semantic content"**) in the real world. So, for example, things such as email addresses, full legal names, date of birth, etc. would generally NOT be good primary keys, even if we were somehow assured they were unique. The reason is because data that has meaning might *change*. You might change your email address or name or even (more rarely) discover you were born on a different day than you originally thought.
4. **Numbers (and GUIDs) are the Rule.** Where possible, its generally good practice to use NUMBERS (and more specifically, integers) as primary keys, as these can be assigned "serially" (in numerical order: 0, 1, 2, …) by the RDBMS and can be easily indexed. In distributed NoSQL/JSON databases, you'll sometimes have to use **Globally Unique IDs (GUIDs)** instead, which are collections of characters and numbers. Both of these options are much, much better than using things such as long blocks of text, which waste space and, more importantly, slow down the operation of the database.
5. **Keep it "Simple" Stupid.** In general, its best to use as *few attributes as possible* for the primary key. Ideally, we would like to use a **simple key,** made up of exactly ONE attribute. For example, using the single attribute "id_number" as a primary key should be preferred to a combination of 2+ attributes.
6. **Security: It's Shouldn't Be a Big Deal if You Know.** Finally, you should NOT choose something as primary key that is potentially sensitive, as primary keys are easily accessible to many users of the database. For example, using a social security number is generally a bad idea.

COMPOSITE KEYS

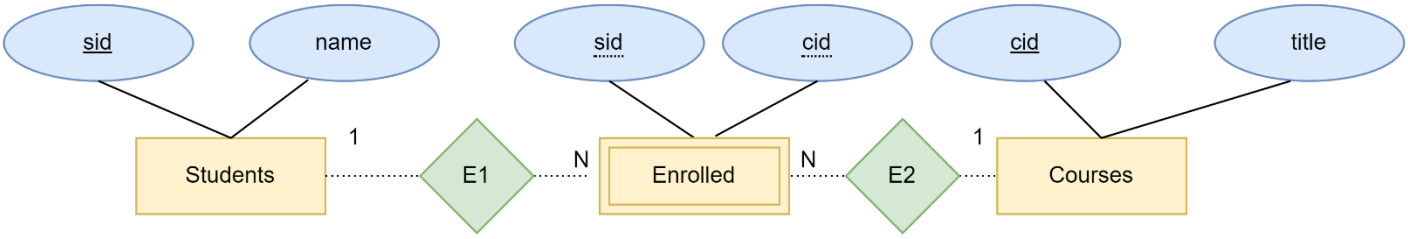
According to the rules laid out in the previous section, we should generally prefer simple keys. However, in some cases, it can make good sense to use **composite keys,** which are made up of two or more attributes. In particular, composite keys should be often be used for the following sorts of entities:

Case 1: "Bridge" or "Linking" Entities. As we've previously discussed, the relational model can't "natively" many-to-many (M:N) relationships, such as those between Student and Course (where many students may register in a particular course AND a given student may register in many courses). In this case, we create a linking entity Enrolled, with a composite primary of key made up of the two *foreign*

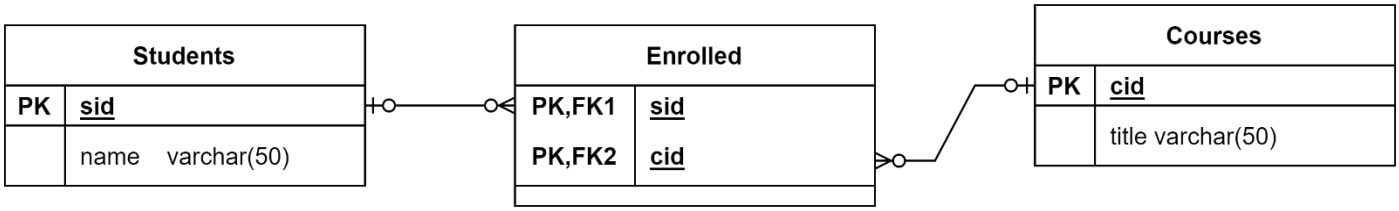
keys that correspond to the primary keys of the entities on each "side." So, for example, consider the following E-R diagram, which contains an optional M:N relationship:



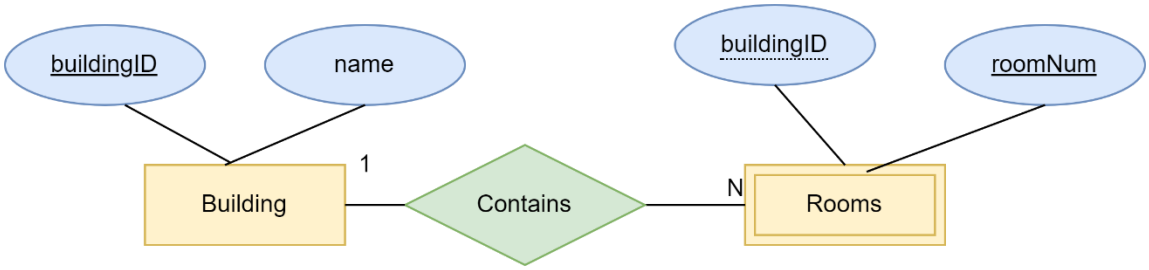
In order to make this relational-model friendly, we'll need to "break up" this M:N relationship into two distinct 1:M relationships. This, in turn, requires that we model the relationship "Enroll" as an entity. In particular, it will be a weak entity, whose existence depends on Students and Courses. Here E1 and E2 are simply generic names for the relationships that Students and Courses have with this "new" entity:



Notice that Enrolled here has a **composite key** that is built out of two foreign keys. Here's a modified crow's foot showing the full **relational schema:**



Case 2: Any weak entity. This is actually a more general version of case 1. As you'll recall a weak entity is (by definition) an entity whose existence depends on some other entity (to which it has a "strong" relationship) AND it derives part/all of its primary key from this entity (or entities). In the previous example, Enrolled provides an example of this. So, for example, in the following diagram, the primary key for Room (a weak entity) is defined, in part, by the foreign key from the Building entity. Notice that roomNum *would not be an adequate primary key on its own.* After all, there may be rooms with the same number in different buildings!



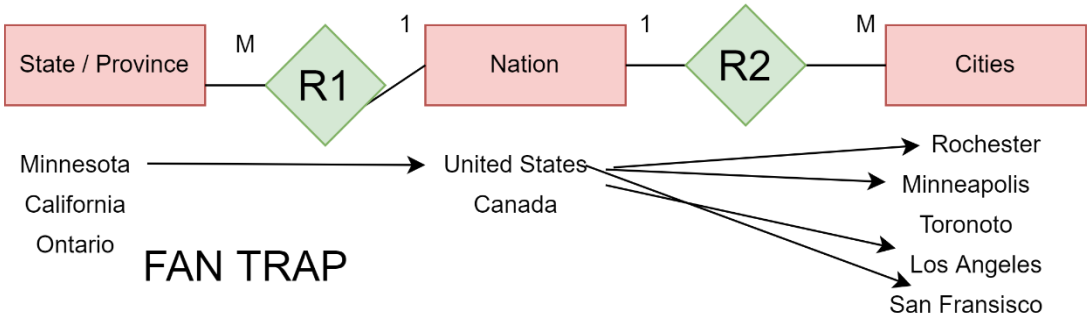
DESIGN PATTERNS: DOS AND DON'TS

Data modeling is somewhere between an art and a science, so its impossible to specify rules that apply to each and every case. That being said, there are some common "patterns" in real-world data that occur pretty frequently. Learning to recognize these patterns, and their impact on modeling, can save you quite a bit of work:

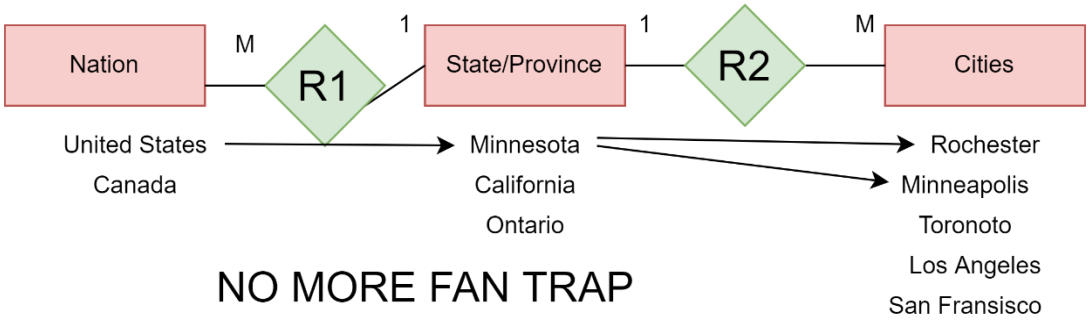
(Reminder) Many-to-many relationships need to be broken up. There are many, many examples of many-to-many relationships in the real world. However, the relational model can't handle these. So, if A and B are in M:N relationship, we "model" this by creating a new "bridge" entity E to capture the relationship between A and B, and to which A and B both bear a 1:M relationship. In general, the primary keys for E will just be a combination of the keys for A and B (see Enrolled, above).

The "Middle Term" and the Fan Trap. It's important to remember that many perfectly "consistent" data models (which satisfy the "rules" of the relational model) can fail to capture those aspects of the real world that they are intended to be "about." These might be called **semantic errors** (in that they concern the "meaning" of the data) as opposed to **syntactic errors** (the sort that produce error messages from the DBMS) One way that this can occur is a so-called **"fan trap",** which occurs when has entities that are "linked" together (such as Students, Enrolled, and Courses), but one chooses the "middle" term incorrectly, and thus fails to capture something about the real-world relationship. This can cause seemingly reasonable queries to "fan out" and produce numerous, nonsensical results:

So, for example, suppose that we model the relationship between Cities, States, and Nations in terms of the business rules that "states are in nations AND cities are in nations." When we try to use relational algebra (or SQL) to ask questions like "Tell me which cities are in Minnesota" we INCORRECTLY are told that this includes every city in the US!



Fixing Fan Traps. We can fix fan traps in one of two ways. We can either (1) recognize that they exist within our data model (after all, every model has gaps!), and carefully avoid asking the sorts of questions that cause problems or (2) fix our data model to more accurately capture the relationships in question (this is often the better choice). In the case above, this means recognizing we need to model the (better) business rules that "Cities are in states AND states are in nations.". The problem is solved!

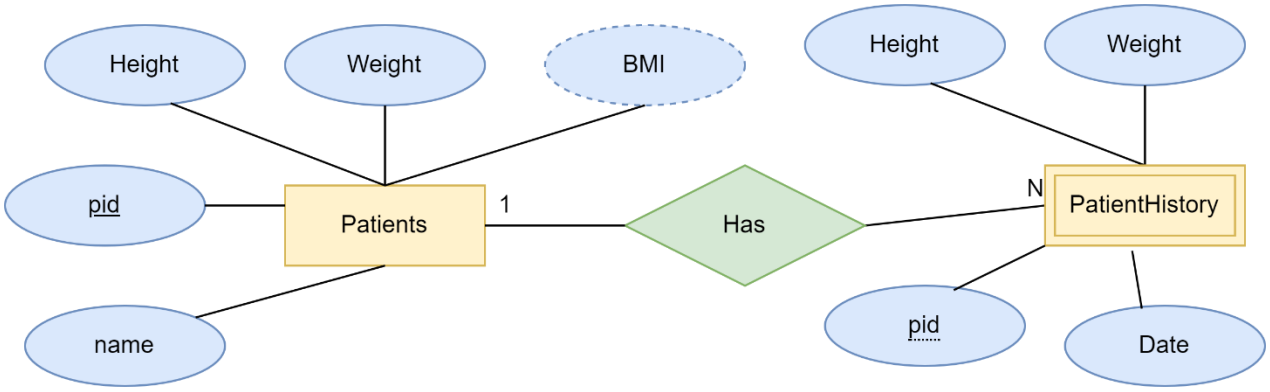


One-to-one relationships (sometimes) should be modeled as two distinct entities. As a rule of thumb, if entities A and B are in a one-to-one relationship (example, a citizen and their social security number), our model only needs to contains a table for ONE of these entities, while the other can be modeled as an attribute (Person=table; social security number = attribute). In some cases, however, we have good reasons for wanting to keep the entities "logically" separate. For example, a person employed at a medical facility might also be a patient at that same facility (that is, there is a one-to-one relationship: it is the exact same person!). However, there are lots of good reasons for keeping our Employees and Patients data separately. We can, of course, keep track of links between these (for example, with an optional foreign key in Patient linking to the primary key of Employee).

- **Other examples:** A subtype-supertype relationship (such as Student: Person) is, by definition, a 1:1 relationship (the same person is both a Student and a Person). In many cases, our relational model will end up storing data about the same person in two separate tables. Here, since a Student MUST be a Person (while a Person need not be a Student), we'd want to store a foreign key reference to the corresponding Person entity within the Student table (and not vice versa).

IT'S TIME(1), TIME(2), TIME(3)

Many real-world problems require that we keep track of how some data set changes over time. For example, in a medical database we want to know not just what a patient's weight and height and height are *right now,* but how this has changed over time. Such **time-variant data** can, in general, be modeled along the lines of a 1:M relationship between the entity and its own history. The weak "historical" entity will nearly always have a composite primary key made up of the primary key of the original entity plus the DATE. So, for example:



When it comes time to actually implement this in a relational database, we could simply collapse the Patients table into the PatientHistory table (since there's really no need to store the most recent height and weight *twice*). In many cases, this is what is done. In other cases, though, it can make sense to store "current" values separately, especially if these values require some sort of calculation (such as BMI), and they are going to be access freqeuently.

REVIEW QUESTIONS

1. In your own words, describe THREE different criteria for choosing a good primary key. Now, give an original example of a primary key choice that would violate each of these criteria.
2. Give an example of a many-to-many relationship between two entities from the real world (other than those we've discussed above). Now:
    a. Describe some attributes of each of the two entities, and say what you would use as the primary key, and why.
    b. Draw an ER diagram for this, using the M:N relationship.
    c. Draw a new ER diagram, with the M:N relationship replaced by two 1:M relationships with a "bridge" entity.
    d. Identify the primary keys for the bridge entity.
    e. Explain why/how your model does NOT commit a fan trap.
3. Give three examples of areas where you might want to keep time-variant data.