CS 325 - Reading Packet: "Database design, part 1"

Sources:

* Kroenke, "Database Processing: Fundamentals, Design, and Implementation", 7th edition, Prentice Hall, 1999.
* Rob and Coronel, "Database Systems: Design, Implementation, and Management", 3rd Edition, International Thomson Publishing, 1997.
* Connolly and Begg, "Database Systems: A Practical Approach to Design Implementation and Management", 3rd Edition, Addison-Wesley.
* Korth and Silberschatz, "Database System Concepts"
* Sunderraman, "Oracle 9i Programming: A Primer", Addison-Wesley.
* Ullman, "Principles of Database Systems", 2nd Edition, Computer Science Press.

Basics of Database Design

Assume that your ERD is finished, your database model is complete. *Now*, and only now, are you ready to convert that database model into a database design/schema, converting that ERD+lists-of-attributes+business-rules into the database design's/schema's table-structures+relationships+domains+business-rules. This conversion from a database model to a database design/schema is called database design. (Yes, it is both process and result...!)

Note that the database model does not go away as a result of this process -- it remains, as documentation of the scenario. In fact, it is VERY common to uncover problems with your model during the database design process -- you should correct the model if you find any such problems! (And that is what I will expect you to do during the course project -- I will expect that the latest model and the latest design always correspond with one another.) And if still-later steps, such as population, writing example reports, etc., uncover more issues, then both the model and the design should be updated accordingly.

Please **NOTE** that correcting the model does NOT mean turning it into the design...! It means, for example, fixing incorrect cardinalities if they are revealed.

How shall we depict our resulting database design/schema? In lecture (and informally), I usually start out by writing it in **relation-structure form**, supplemented by either arrows or SQL-style foreign key clauses to indicate the relationships. (This is what I'll do in the examples here, and what you'll do for homework/test problems involving the database design process.) This form does not include domain information about the attributes, however, so to conveniently include that, you will depict your project's final database design/schema in the form of SQL create-table statements. (See how this form includes table structure, relationship information, and at least some physical domain information for the attributes?) This is not the only possibility, but it is quite convenient, as once you are done you have a lovely basis for a SQL script creating those tables. And of course we are already maintaining a set of

business rules (which will also likely be added to during the design process, "documenting" either the client's decisions you find are needed along the way or at least documenting the decisions you are making in various regards...!)

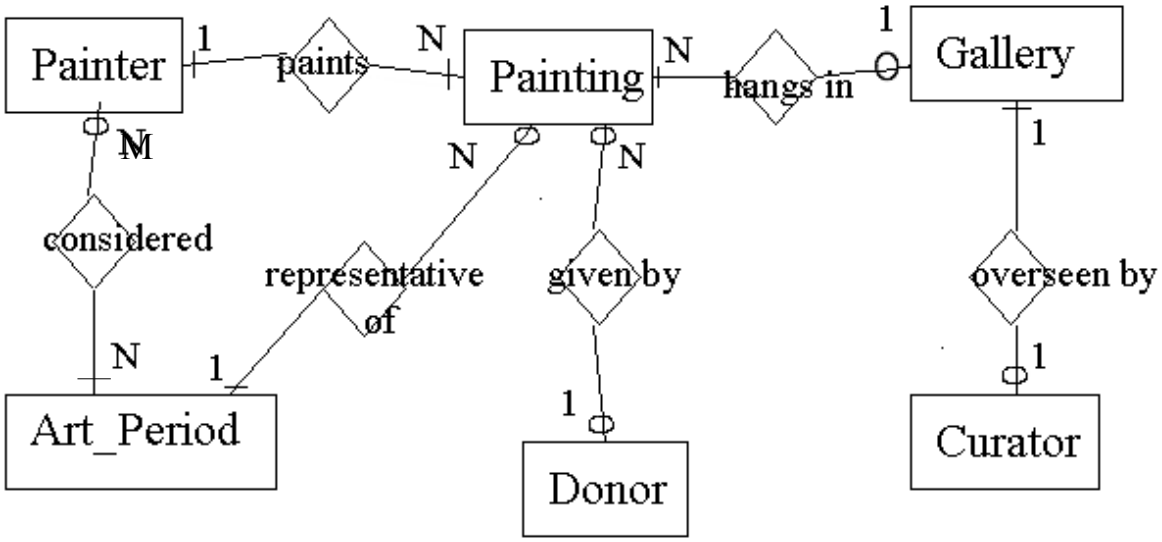
Note that the resulting database design/schema is **DBMS-independent**, except, perhaps, for how the domains are indicated. This relational design is not tied to any particular DBMS; it can then be implemented using any relational DBMS. (Although the "bells and whistles" available in different relational DBMS's vary greatly, basic table creation is very similar.) Even our choice of depicting the design/schema as SQL create-table statements is almost DBMS-independent -- it would probably be quite reasonable to adapt those statements for use by just about any relational DBMS. (Indeed, I have heard of at least one student successfully implementing his/her database course project database design on a different DBMS after the semester was over.) The main change would likely be to change the types for the columns (their physical domains).

(I'm not sure we mentioned it during our discussion of database models, but note that our ER models are also DBMS-independent! The model is the same regardless of the relational DBMS on which one plans to implement the eventual database.)

Some texts distinguish between a logical design and a physical design -- sometimes the logical design refers to the model, sometimes it refers to an intermediate stage such as a design depicted in relation-structure form (perhaps augmented by a **data dictionary** to provide early domain information for the relations' attributes). The physical design, then, is more DBMS-dependent. For our project, you are sort of creating a combined logical/physical design, in a sense. It works well for this project.

Converting a Model into a Relational Database Design

So: consider the following database model, expressed as an ER diagram (it is also available as a separate link along with this "lecture", in cause you'd like to have it open in a separate window for easier reference while walking through the examples weaving through the discussion of the design process):



Painter

PTR_LNAME
Ptr_fname
Ptr_dob
Ptr_dod

Art_Period

PD_NAME
Pd_begin
Pd_end

Painting

PTG_NUM
Ptg_title
Ptg_cost
Ptg_acquis_date

Gallery

GAL_NUM,
Gal_street_addr
Gal_name

Donor

DON_NUM
Don_lname
Don_fname
Don_street_addr
Don_city
Don_state
Don_zip
Don_email (MV)

Curator

CUR_LNAME
Cur_email
Cur_phone (MV)
Cur_certification (MV)

We have stated several times now that an **entity class** is NOT equivalent to a table/relation -- rather, each entity class will correspond to one **or more** relations in a relational database design/schema. Indeed, we'll find, in practice, that most database designs/schemas have more relations than their corresponding ER mode has entity classes; this is NORMAL, and even expected! (And it will be true in your projects as well, given the minimum project requirements, selected to help ensure that you get some practice with the most important modeling and design features.)

STEP 1, Part 1 - obtaining an initial set of "base" tables

But, we did say that each entity class corresponds to at least one relation -- a good first step in converting an ER model to a database design/schema is to define what I call a "base" relation for each entity class in the model (understanding that additional relations will later be added during the later steps of the design process). The name of the new relation could certainly be based on the name of the entity class, especially if it seems suitable for a relation name -- for the above model, then, we could get the following "base" tables (starting to write each in relation-structure form):

```
Painter(
Painting(
Donor(
Curator(
Art_period(
Gallery(
```

STEP 1, Part 2 - determining primary keys for the "base" tables

For each "base" table, look at the identifying attribute(s) noted, if any. Is it/Are they unique? Is it/Are they a suitable candidate key, a suitable candidate for the primary key of that "base" table? THIS is when you'll either decide to make such identifying attributes the primary keys for their "base" tables, or if you will add a more suitable additional attribute to serve as primary key instead.

NOTE that this is NOT an error in your model if you decide to add a more suitable attribute to serve as a primary key -- it is simply an implementation choice. No modification to your model is necessary. The situation is likewise if you rename attributes to meet naming conventions such as replacing dashes or blanks with underscores, rewriting the names to make them more readable, etc. (However, if you realized that you had left out some important entity class characteristic during this process -- the curator's first name, or something -- then you should add that to a revised model.)

Note that numeric codes, or fixed-length strings in which the characters happen to be digits, do make for very convenient primary keys -- varying-length strings of alphabetical characters are just too prone to matching issues. Also, people and organizations have a tendency to change names over time -- department names change, people's names change, company names change, and so on. We know that queries are very dependent on primary and foreign keys. I don't want you to feel that you must always put in a number or code for a table's primary key, but it does often happen. Do **not** take this so far that you always add a new numeric key for every table -- we'll see shortly some peculiar varieties of tables for which multiple-attribute primary keys are exactly what we want, and should be allowed to stay that way. But also don't be surprised if you find yourself adding such primary key attributes to some of your "base" tables at this step.

For our example model, then:

* entity class Painter's identifying attribute, PTR_LNAME, would definitely not be a suitable candidate for "base" table Painter's primary key -- last names are not even unique! We'll add a **PTR_NUM** attribute to the Painter "base" table, to serve as its primary key, and add Ptr_lname as a "plain" attribute:

```
Painter(PTR_NUM, Ptr_lname,
```

* entity class Painting's identifying attribute, PTG_NUM, sounds like it could be a quite-suitable primary key for "base" table Painting. Once you verify (with the client or using the business rules) that this PTG_NUM is indeed unique, we can add it to the Painting "base" table, to serve as its primary key:

```
Painting(PTG_NUM,
```

* entity class Donor's identifying attribute, DON_NUM, also sounds like it could be a quite-suitable primary key for "base" table Donor, once it is confirmed that it is indeed unique:

```
Donor(DON_NUM,
```

* entity class Curator's identifying attribute, CUR_LNAME, is another attribute unsuitable for serving as a primary key, as it cannot be guaranteed to be unique, and so we'll add a **CUR_NUM** attribute to the Curator "base" table, to serve as its primary key, and add Cur_lname as a "plain" attribute:

```
Curator(CUR_NUM, Cur_lname,
```

* entity class Art_period's identifying attribute, PD_NAME, is very likely unique, but as a descriptive name that could be hard to match, easy to mis-type, and possibly even prone to change based on "fashions" of discourse, we'll introduce a PD_CODE attribute to serve as primary key instead, and keep Pd_name as a "plain" attribute:

```
Art_period(PD_CODE, Pd_name,
```

* entity class Gallery's identifying attribute, GAL_NUM, also sounds like it could be a quite-suitable primary key for "base" table Gallery, once it is confirmed that it is indeed unique:

```
Gallery(GAL_NUM,
```

STEP 1, Part 3 - adding the remaining attributes from the ER model to the design/schema thus-far

Now, consider each non-identifying attribute in each entity class' attribute list in turn. Is it single-valued? If so, add it to the "base" table for that entity class. (If your model's attribute lists include whether attributes are mandatory or not, then at this point you could also record if this attribute should be declared as **not null** -- a mandatory attribute, one that must have a value, should be declared to be **not null**.)

How about multi-valued attributes? We know that a relation cannot have multi-valued attributes -- we were reminded of this last week, in our discussion of 1st Normal Form (1NF) -- and so you should not be surprised that **each** multi-valued attribute will lead to a **separate** new table in the design/schema:

For each multi-valued attribute:

* create a new table with an appropriate name (often based on some combination of the entity class' name and the multi-valued attribute's name, although to reduce confusion you should avoid having an attribute with the same name as a table)

* make the primary key of the entity class' "base" table an attribute in this new table, and make it a

foreign key referencing the entity class' "base" table

* add the multi-valued attribute to this new table (NOT to the entity class' "base" table), and

* make the primary key of this new table consist of BOTH of these attributes (or all of them, if the "base" table happens to have a multi-attribute primary key)

You'll recognize this as exceedingly similar to the process for normalizing a set of relations into 1NF, except that since the tables don't exist yet, we don't have to remove the multi-valued attribute from any table -- we are adding it to an appropriate table to begin with! (Make sure that this is clear to you: the multiple-valued attribute is **not** an attribute in that entity class's "base" table! Instead, it is an attribute in this newly-added table.) And this approach is very flexible and very query-able.

(Remember, too, that none of these new tables resulting from multi-valued attributes result in any changes to the model -- they are simply part of the design/schema corresponding to that model. Likewise, no foreign key you add as part of the design process gets added to the model, either -- they are simply part of implementing the relationships depicted in the model.)

Walking through this process for our example model and example relations-so-far:

* all of entity class Painter's remaining attributes are single-valued, and so we will simply add them to the Painter "base" table:

```
Painter(PTR_NUM, Ptr_lname, Ptr_fname, Ptr_dob, Ptr_dod,
```

* likewise, all of entity class Painting's remaining attributes are also single-valued, so all are added to Painting's "base" table:

```
Painting(PTG_NUM, Ptg_title, Ptg_cost, Ptg_acquis_date,
```

* entity class Donor's remaining attributes include one multi-valued attribute, so we'll ad all but that one to Donor's "base" table:

```
Donor(DON_NUM, Don_lname, Don_fname, Don_street_addr, Don_city,
      Don_state, Don_zip,
```

...and then we'll create a new table for the multi-valued attribute Don_email:

```
Donor_email(
```

...with the primary key of Donor's "base" table and the multi-valued attribute added to the new table, both making up its primary key, and with the primary key of Donor's "base" table defined as a foreign key referencing the Donor's "base" table:

```
Donor_email(DON_NUM, DON_EMAIL)
   foreign key (don_num) references Donor
```

* entity class Curator's remaining attributes include one single-valued attribute, Cur_email (which is not a typo -- in this scenario, the client happens to only want to keep a single preferred email address for each curator, even though they also want multiple phone numbers per curator to be possible. Remember, your model is based on the *user's* view of their world!) So, we'll add Cur_email to the Curator "base" table:

```
Curator(CUR_NUM, Cur_lname, Cur_email,
```

  Curator's remaining attributes also include two multi-valued attributes, Cur_phone and Cur_certification. Each will result in the addition of a new table, each with the Curator's "base" table's primary key as a foreign key, each containing that particular multi-valued attribute, and each having both attributes as its primary key:

```
Curator_phone(CUR_NUM, CUR_PHONE)
   foreign key (cur_num) references Curator

Curator_certif(CUR_NUM, CUR_CERTIFICTN)
   foreign key (cur_num) references Curator
```

* all of entity class Art_period's remaining attributes are single-valued, and so we will simply add them to the Art_period "base" table:

```
Art_period(PD_CODE, Pd_name, Pd_begin, Pd_end,
```

* all of entity class Gallery's remaining attributes are single-valued, and so we will simply add them to the Gallery "base" table:

```
Gallery(GAL_NUM, Gal_street_addr, Gal_name,
```

STEP 2 - Handling relationships

Now that all of the ER model's attributes have been added to the design/schema thus far, it is time to handle the relationships. Every relationship class -- every relationship line -- in the ER model will require something to be done to the developing design/schema, so that the tables are related appropriately, reflecting the ER model. Since there are three kinds of relationships -- 1:1, 1:N, and N:M -- then once you know how each kind should be handled, you will be able to handle each of the relationship lines appropriately.

STEP 2, Part 1 - Handling 1:N relationships

1:N relationships are typically the most common kind of relationships in a model. Each of them is handled in the same way: the primary key of the "base" table of the entity class on the "1" side of the relationship is added to the "base" table of the entity class on the "N" side of the relationship as a foreign key.

Do you see how this relates the "base" tables as the model implies? Each instance of the entity on the

"N" side is related to at most one instance of the entity on the "1" side -- so, the foreign key in the "N" side's base table will have a value, or be null. This is reasonable and workable. (And if the relationship is mandatory, that foreign key attribute can be declared to be **not null**, so it must be set to a value.) If you tried it the other way -- since each instance of the entity on the "1" side might be related to multiple instances of the entity on the "N" side -- you'd be in trouble, since, in a relation, a foreign key on the "1" side cannot be multiple-valued any more than any other attribute can be.

Often, the entity class on the "1" side is called the parent in this relationship, and the entity class on the "N" side is called the child in this relationship. This lets us describe what is done to implement a 1:N relationship in the database design/schema more easily: the primary key of the parent's "base" table is added to the child's "base" table as a foreign key.

So, for the example, this is how each 1:N relationship would be handled:

relationship: Painter - paints - Painting

Since Painter is the "1" side (the parent) of this relationship, and Painting is the "N" side (the child), then the Painter "base" table's primary key will be added to the Painting "base" table as a foreign key:

```
Painter(PTR_NUM, Ptr_lname, Ptr_fname, Ptr_dob, Ptr_dod,

Painting(PTG_NUM, Ptg_title, Ptg_cost, Ptg_acquis_date, ptr_num,
   foreign key (ptr_num) references Painter
```

relationship: Painting - representative of - Art_period

Since Art_period is the "1" side of this relationship (the parent), and Painting is the "N" side (the child), then the Art_period "base" tables's primary key will be added to the Painting "base" table as a foreign key:

```
Painting(PTG_NUM, Ptg_title, Ptg_cost, Ptg_acquis_date, ptr_num,
         pd_code,
   foreign key (ptr_num) references Painter,
   foreign key (pd_code) references Art_period

Art_period(PD_CODE, Pd_name, Pd_begin, Pd_end,
```

relationship: Painting - given by - Donor

Since Donor is the "1" side of this relationship (the parent), and Painting is the "N" side (the child), then the Donor "base" tables's primary key will be added to the Painting "base" table as a foreign key:

```
Painting(PTG_NUM, Ptg_title, Ptg_cost, Ptg_acquis_date, ptr_num,
         pd_code, don_num,
   foreign key (ptr_num) references Painter,
   foreign key (pd_code) references Art_period,
   foreign key (don_num) references Donor

Donor(DON_NUM, Don_lname, Don_fname, Don_street_addr, Don_city,
```

```
      Don_state, Don_zip,
```

relationship: Painting - hangs in - Gallery

Since Gallery is the "1" side of this relationship (the parent), and Painting is the "N" side (the child), then the Gallery "base" tables's primary key will be added to the Painting "base" table as a foreign key:

```
Painting(PTG_NUM, Ptg_title, Ptg_cost, Ptg_acquis_date, ptr_num,
         pd_code, don_num, gal_num,
   foreign key (ptr_num) references Painter,
   foreign key (pd_code) references Art_period,
   foreign key (don_num) references Donor,
   foreign key (gal_num) references Gallery

Gallery(GAL_NUM, Gal_street_addr, Gal_name,
```

(Note: you will not always end up adding so many foreign keys to the same table .. Painting is just very "central" in this particular model, and just happens to be the "N" side (child) of a number of 1:N relationships...! That doesn't mean it might not be the parent in some other 1:N relationship, or be involved in additional 1:1 or N:M relationships.)

STEP 2, Part 2 - Handling N:M relationships

N:M relationships are the next most-common relationships in a model, more frequent than 1:1 relationships, although usually less frequent than 1:N relationships. Each of these is handled in a particular way, also: each results in a so-called **intersection table** being created, which consists of the primary keys of the each of the "base" tables of the entity classes involved, both of these making up the primary key of the new table, and each defined to be a foreign key referencing its respective "base" table. Often the name of the intersection table is based on the relationship class name, or on the names of the two entity classes involved, or some combination of the two.

Do you see how this relates the "base" tables as the model implies? Each instance of the entity on one side of this relationship might be related to multiple instances of the entity on the other side -- trying to put a primary key of one into the other as a foreign key would be doomed to failure, as either choice would result in an "illegal" multiple-valued foreign key attribute. But the new so-called "intersection table" really is depicting the intersecting relationships of these two entity classes -- if an instance on one side is related to 4 instances on the other, then there will be four rows representing this in this intersection table. If an instance on the other side is related to 2 instances on the first side, then there will be two rows representing this in this intersection table. Since a value of either attribute might appear more than once, it is only the pair of these values that can be guaranteed to be unique in this intersection table, and thus both attributes must be part of the intersection table's primary key. And joins between the two entities' "base" tables and this intersection table allow for a wealth of possible queries regarding these multiply-related entity instances.

So, for the example, this is how each N:M relationship would be handled:

relationship: Painter - considered - Art_period

We would create an intersection table -- let's call it Painter_in_period.

```
Painter_in_period(
```

It gets the Painter "base" table's primary key as an attribute, it gets the Art_period "base" table's primary key as another attribute -- both make up the new table's primary key, and each is defined to be a foreign key back to its respective "base" table:

```
Painter_in_period(PTR_NUM, PD_CODE)
   foreign key (ptr_num) references Painter,
   foreign key (pd_code) references Art_period
```

STEP 2, Part 3 - Handling 1:1 relationships

1:1 relationships are the least frequent kind of relationships in a model, in my experience. They also often have a bit more flexibility in how they are converted into a design. After all, since an instance of the entity on one side is related to at most one instance of the entity on the other side, placing the primary key of the "base" table of the entity class on one side of the relationship into the "base" table of the entity class on the other side of the relationship as a foreign key should work fine -- whichever "direction" you choose!

HOWEVER - that is NOT saying to add foreign keys to BOTH "base" tables -- that is a recursive situation that is NOT desired here! Choose which "direction" you'd like, and only add a foreign key to ONE of the two "base" tables involved.

Does it really not matter which "direction" you choose, here? There is one situation when the choice is clear: when the 1:1 relationship is mandatory on one side (has a minimum cardinality of 1, indicated by a line on the relationship line near the entity rectangle) and optional on the other (has a minimum cardinality of 0, indicated by an oval on the relationship line near the entity rectangle). That is, an instance of the entity class on one side is required to be related to exactly one instance of the entity class on the other side, whereas an instance of the entity class on the other side may or may not be related to an instance of the entity on the first side. In this situation, it is preferable to add the foreign key to the side that must be related to an instance on the other side, which is the entity with the oval near its entity class. Confused? Fortunately, our example has such a 1:1 relationship, so we'll demonstrate this there.

But for cases where the 1:1 relationship is optional on both sides, or where the 1:1 relationship is mandatory on both sides, it will likely be OK whichever option you choose. Sometimes it is advised to try to decide which "direction" will work in favor of the most common queries one is likely to make, if you have enough information to be able to guess.

(Commonly, there is also a warning to double-check 1:1 relationships that are mandatory on both sides to make sure that both entity classes really are distinct entity classes, and that you are not trying to make an entity out of an attribute. For example, if:

* the relationship is mandatory in both directions,

* one of the entities does not participate in other relationships
* the identifying attributes of the two entities are the same

...then those are strong indications that these two entities may really be a single entity.

But if both entity classes have distinct attributes, particularly identifying attributes, of their own, or both participate in different relationships with other entity classes, either (or both) are good evidence that both are entity classes in their own right.)

So, for the example, this is how each 1:1 relationship would be handled:

relationship: Gallery - overseen by - Curator

Since this is a case where the relationship is mandatory on one side -- a Curator has to be related to at least one and at most one Gallery -- and optional on the other -- a Gallery does not have to be related to a Curator, but may be related to at most one Curator -- then the better choice would be to add the Gallery "base" table's primary key to the Curator "base" table as a foreign key:

```
Gallery(GAL_NUM, Gal_street_addr, Gal_name,

Curator(CUR_NUM, Cur_lname, Cur_email, gal_num
   foreign key (gal_num) references Gallery
```

Now, if one defines the gal_num attribute of the Curator "base" table to be **not null**, now every Curator will have to be related to a Gallery (will have to have a gal_num referencing the Gallery table). You cannot always get the DBMS to enforce mandatory relationships so easily, but when you can, it is a nice plus.

This, then, is the resulting database design/schema for this ER model:

```
Painter(PTR_NUM, Ptr_lname, Ptr_fname, Ptr_dob, Ptr_dod)

Art_period(PD_CODE, Pd_name, Pd_begin, Pd_end)

Painter_in_period(PTR_NUM, PD_CODE)
   foreign key (ptr_num) references Painter,
   foreign key (pd_code) references Art_period

Gallery(GAL_NUM, Gal_street_addr, Gal_name)

Curator(CUR_NUM, Cur_lname, Cur_email, gal_num)
   foreign key (gal_num) references Gallery

Curator_phone(CUR_NUM, CUR_PHONE)
   foreign key (cur_num) references Curator

Curator_certif(CUR_NUM, CUR_CERTIFICTN)
   foreign key (cur_num) references Curator

Donor(DON_NUM, Don_lname, Don_fname, Don_street_addr, Don_city, Don_state,
      Don_zip)
```

```
Donor_email(DON_NUM, DON_EMAIL)
   foreign key (don_num) references Donor

Painting(PTG_NUM, Ptg_title, Ptg_cost, Ptg_acquis_date, ptr_num, pd_code,
         don_num, gal_num)
   foreign key (ptr_num) references Painter,
   foreign key (pd_code) references Art_period,
   foreign key (don_num) references Donor,
   foreign key (gal_num) references Gallery
```

If you are comfortable with these steps, then you are ready to convert a large percentage of most ER models into good database designs/schemas. There are a few other relatively common model situations that might occur -- weak entities, for example, and supertype/subtype entities -- that we have not yet discussed. We will discuss these next week.