CS 325 - DB Reading Packet 4: "Entity-relationship modeling, part 1"

NOTE: you are required to follow **course standards** for ERDs, regardless of the different ERD notations used in different software and textbooks.

Sources:

- Kroenke, "Database Processing: Fundamentals, Design, and Implementation", 7th edition, Chapter 1, Prentice Hall, 1999.
- Connolly and Begg, "Database Systems: A Practical Approach to Design Implementation and Management", 3rd Edition, Addison-Wesley.
- Korth and Silberschatz, "Database System Concepts"
- Rob and Coronel, "Database Systems: Design, Implementation, and Management", 3rd Edition, International Thomson Publishing, 1997.
- Ricardo, "Databases Illuminated", Jones and Bartlett.
- Sunderraman, "Oracle 9i Programming: A Primer", Addison-Wesley.
- Ullman, "Principles of Database Systems", 2nd Edition, Computer Science Press.
- Sharon Tuttle, “Database Design”, Lecture readings, Cal Poly Humboldt.

Introduction to data modeling and Entity-Relationship modeling

We are now entering our discussion of **data modeling** -- "the process of creating a **representation** of the **users' view** of the data" (Kroenke). Kroenke gives the opinion that "it is the **most** important task in the development of effective database applications" -- and, indeed, it is quite true that a poor underlying data model can make both developers and users less than satisfied, resulting in database applications that are "difficult to use, incomplete, and very frustrating".

When you took your first programming course, it was (hopefully!) stressed that you should determine the algorithm, the desired program logic, before actually starting to write code, whether in C++, or Java, or whatever programming language is being used to implement the program. This logic can be expressed in a number of reasonable ways, including pseudocode or flow-charts. Somewhat analogously, one uses **modeling** to come up with a logical database model before actually implementing a database, indeed as **part** of the process of determining how to reasonably implement that database.

Make sure that this is clear, because it is a major theme in this course, and if you take this one idea away from this course, you will be a better database designer:

You should create a logical database model BEFORE YOU EVEN START THINKING ABOUT TABLES AT ALL.

Jumping right into creating tables is a mistake. When you jump into the table-design phase too early, it is like just starting to type in C++ code before you have really decided what you are doing. You will likely make table design decisions that you will either have to locate and correct later, or that will be hard to deal with and hard for users and applications to work with.

Aside: expanding on the concept of a database design/database schema

A **database design**, or **database schema**, is the design, the foundation upon which the database and its applications are built (Kroenke). In our "mathematical" discussion of relations in the reading packet introducing the relational model, we gave the somewhat "mathematically-oriented" definition of a **relational schema**:

relation structures plus constraints on allowable data values

More often, I'll use this slightly-more pragmatic definition of a database schema/database design, from Kroenke. This definition of a database design/database schema includes:

- its **tables**,
- **relationships**,
- **domains**, and
- **business rules**

You should know what a table/relation is at this point; and, you should know that a column's/attribute's domain is the set of values that column can contain. Relationships in this context essentially mean foreign keys, those primary keys from one table that are defined as foreign keys in another table so that a row in one table can be related to a row in another table based on a foreign key column's value. And, finally, business rules are day-to-day operational rules or assumptions within a scenario or setting or "world" -- they are further constraints, if you will, on what is acceptable for a given database based on that scenario or setting or "world". For example, if your setting is a sports club, and a business rule for that club is that members can only check out up to 10 pieces of equipment at a time, then that is saying it wouldn't be good for the database to permit a check-out of an 11th piece of equipment for some member.

(And, if you think about it, these two definitions of a database schema/database design don't really conflict -- a relation structure does give a table's structure, and the relationships, domains, and business rules can reasonably be viewed as constraints on allowable data values.)

There are many ways to express these elements; for the course project, I will specify how these need to be expressed, but many means are possible. (After all, we have already discussed three different ways to depict tables!) Many means of expressing designs/schemas are relational-DBMS-independent - with this information, you could then actually physically implement the database so designed using any reasonable relational DBMS! (Really, this is not unlike how a good algorithm -- or program design? -- can be programmed using any of a variety of reasonable programming languages.)

The Entity-Relationship Model

There is more than one way to model a database, and more than one way to depict that model -- we will be studying one of the classic means, using the Entity-Relationship model. As we have already mentioned, the **relational model** was first developed by **E.F. Codd**, at IBM, in **1970**, based on a branch of mathematics, relational algebra. The **Entity-Relationship Model** was introduced by **Peter Chen** in a **1976** paper. It is commonly abbreviated as the **E-R model**, and diagrams based on this model are often called **ERD**s (Entity-Relationship diagrams). Once you have an ERD depicting an E-R model you are satisfied with for a scenario (and, more importantly, that the client is satisfied with), it is then quite reasonable to convert that ERD into a database schema/design, as we will see.

ERDs and the E-R model are widely used, but oddly enough there is no one standard notation for ERDs. Some parts of the notation are fairly standard, and other parts vary quite significantly. The course text gives two variants of such notations; other texts I have seen (including Rob and Coronel, Kroenke, Connoly and Begg, and Ricardo) have different variations from these. However, the basic ideas, the basic underlying model, is the same -- what differs is just how you graphically depict that information. Note that, to keep things clearer for everyone in the course, there will be a course standard ERD notation, and you are expected to use this notation for all E-R models you create for this course.

The entity-relationship data model is based on a perception of a real world which consists of a set of basic objects called **entities**, and **relationships** among these objects. (Korth and Silberschatz) These can be expressed by an E-R diagram (ERD) that expresses the overall logical structure of these entities and relationships.

Key elements of the E-R model are **entities**, **relationships**, **attributes**, and **identifiers**. (Kroenke)

Entities

An **entity** is an object that exists and is distinguishable from other objects. (Korth and Silberschatz) It is something that can be identified in the users' environment-of-use; it is something they may want to track or record. For example, John Harris with Social Security Number 111-11-1111 is an entity in an IRS scenario; savings account 114 at the Downtown Small Town Bank is an entity. The registration of a student Jones in a section 42333 of a course Math 123 is an entity in a school. So, note that entities may be concrete, like a person or a book, or abstract, like a holiday or a registration, or something in-between.

An **entity class** is a **set** of entities of the same type (Korth and Silberschatz), although some texts turn this on its ear and call Korth and Silberschatz' entity class an entity, and call Korth and Silberscatz' entity an entity instance! As long as you can distinguish between the broader set of instances and an individual instance, you should be fine. (That is, be able to distinguish between the broader set of savings accounts and the individual instance savings account 114.) It is the broader set that you depict in an ERD.

Entity classes -- the broader set -- are virtually always depicted in ERDs as labeled rectangles (usually in landscape orientation).

Figure 1 - Savings-Account entity class as it would appear in an ERD

Attributes

Attributes, or **properties** as they are sometimes also called, describe entity characteristics that are **important** in the user's world/environment. Really, we could say that an entity is represented by a set of attributes/properties. Some possible attributes of a `CUSTOMER` entity class might be `Last-name`, `Social-security-number`, `Street`, `City`, `Zip-code`, and `Phone-number`**.** If this `CUSTOMER` was an entity class for, say, a car dealership, would you be likely to include such attributes as `Hair-color` or `Fingernail-length`? No, because those characteristics are not significant to the users' world view. (`Hair-color` might be, however, for a `CUSTOMER` entity class within a make-up salon scenario.) Some possible attributes of a `Savings-Account` entity class might include `Account-number` and `Balance`.

Many texts initially represent attributes within ERDs by writing their names within ovals, and connecting those ovals to the entity class they are for by a line. The optional course text follows this fine tradition. However, you can also imagine -- if it isn't immediately obvious when you look at such diagrams -- that this gets unwieldy quite quickly! More importantly, it starts to get in the way of reading the ERD in general. So, more often in practice, attributes are listed separately, at the bottom of the page or on attached page(s), in a list for each entity class. That's what we'll do for this class (although we'll add a bit more notation to these attribute lists, as will be described).

Conceptually, attributes may be **simple** or **composite**, or **single-valued** or **multi-valued**. What is meant by simple or composite? Simple implies not further subdivisable or atomic, composite implies there are parts to be subdivided into. Consider a `Salary` attribute, generally a single, indivisible value, versus a `Name` attribute, often subdividable into first name, middle name, last name. Some DBMSs do have ways to handle composite attributes quite elegantly, but some do not. To facilitate detailed queries in the future, it is wise to change composite attributes into a series of simple attributes. (That is, have attributes `Last-name` and `First-name` rather than the attribute `Name`.)

What about single-valued or multi-valued? This means, **for a given entity instance**, can it only have a single `Last-name`, or might it have multiple `Last-name`s? If it can only have one, then `Last-name` is considered a single-valued attribute; if it can have more (consider an attribute `Phone-number`), that attribute is considered a multi-valued attribute. It is important to the later design that multi-valued attributes, if any, are identified at the modeling stage -- but identifying them as multi-valued is all that is needed (or desired) at this stage. Remember, we are **not** thinking about tables yet!

In the oval notation for attributes, multi-valued attributes are sometimes indicated by connecting them to their entity class rectangle with a double-line; however, since we are avoiding the oval notation, we'll instead note any multi-valued attributes by writing `(mv)` after them in the entity class's attribute lists.

Important note: in general, do **not** attempt to deal with multi-valued attributes by having numbered single-valued attributes -- that is, avoid such attributes as `Phone-number1`, `Phone-number2`!! It is better to just indicate that `Phone-number` is a multi-valued attribute. However, sometimes, after further reflection or consultation with a client, you discover that what the client really wants is just a single preferred-contact phone number, or a cell phone and a home phone, distinguished accordingly. If so, that's how the attributes should be set up (`Preferred-phone`, or `Cell-phone` and `Home-phone`, for example). But multi-valued attributes are simply a part of many scenarios, and the important thing in the modeling stage is to simply note them as such. In an ERD, attributes will be assumed to be single-valued unless specifically noted as multi-valued.

Identifiers

When thinking about entity classes, each entity instance is, well, a separate instance of that entity. It has an identity distinct from the other entity instances of that class. How does someone in that scenario tell different entity instances apart? What attribute or attributes does someone in that scenario use to distinguish different entity instances from one another? At the model level, one might designate one or more attributes as **identifiers**, identifying attributes, that those in the scenario use to identify or distinguish entity instances.

Do we mean primary key? **No**, because these are not tables yet! (We'll eventually see that each entity ends up being represented as one **or more** tables in the eventual database design/schema.) I think this is more to better characterize the entity classes at the modeling stage, and perhaps to later suggest possible reasonable candidate keys -- **but** at the modeling stage, these don't have to be as robust as the eventual tables' primary keys will be designed to be. These may be a single attribute -- `Account-number` -- or composite, made up of several attributes -- `{First-name, Last-name, Middle-Initial}`. As you can see by that second example, these may not even necessarily be unique yet, at this modeling stage -- we often identify people by their names, even though we know they aren't unique. It is more important, while modeling, to represent how those within a scenario tell instances of entity classes apart. We'll think about (and devise) good primary keys for the eventual tables later, when we'll be in a better position to do so.

So, how will we indicate identifiers, identifying attributes, in our ERDs? We'll underline them or write them in all-uppercase within our attribute lists for each entity. (In the oval notation, their names are often underlined or written in all-caps within their respective ovals as well!)

Relationships

A **relationship** is an association among several entities. And, again, we have relationship classes, and relationships (or relationships, and relationship instances), but, again, as long as you can keep the overall set and the individual instance straight, you should be fine. Relationship classes are associations amongst entity classes, relationship instances are associations amongst entity instances, and people just tend to say "relationships" after a while and depend on context to indicate which is really meant, relationship classes or relationship instances.

If you are tempted to give a relationship class an attribute -- which is permitted in some variants of ERDs -- I usually suspect that there is a buried entity class trying to make itself known, often a significant association or transaction within that scenario. I think it is often clearer to determine what that association or transaction entity class is, and to model that instead.

We will give each relationship class a name; it is hard to choose descriptive names that work in "both" directions of the relationship, so don't worry if they are rather "unidirectional." I think "verb-based" names work best for relationship classes. Avoid using the same name for different relationship classes.

So, for some examples: we may define a relationship which associates `Customer` Harris with `Savings-Account` 114. That is, perhaps Harris is the owner of `Savings-Account` 114; we might call this an `owns` relationship. That relationship is an instance of the more general relationship class `owns`, a relationship between the `Customer` entity class and the `Savings-Account` entity class.

Formally, a relationship class is a mathematical relation on n >= 2 (possibly non-distinct) entity classes.

Relationship classes between entity classes are almost always indicated in ERDs by placing a line between the related entity classes. Often, this line has a diamond on it, and we'll follow this practice for CS 325 DL ERDs. You'll then write the name of the relationship class on or near that diamond (whichever is most convenient for you.

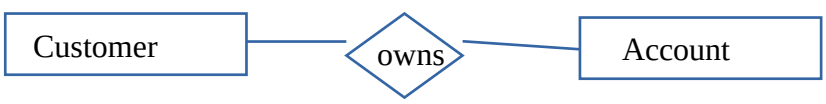

Figure 2 - ERD showing that a Customer entity class is related to an Account entity class via an owns relationship class

Figure 3 - ERD showing that an Instructor entity class is related to a Class-section entity class via a teaches relationship class

Degree of a relationship

The **degree** of a relationship is the number of entities involved in a relationship. As defined in the original E-R model, relationships may exist among many entities, although usually they only do so among 2. Such relationships -- with a degree of 2 -- are called **binary** relationships, and most of the relationship classes in a database system are binary. They tend to work best in practice within the relational model. So, relationships of a higher degree are usually re-cast as several binary relationships (possibly with the addition of association-entity classes).

Maximum Cardinalities

An E-R model may define certain constraints to which the contents of a database must conform. One important constraint is **maximum cardinalities** (sometimes also called "mapping cardinalities"), which express the number of instances of one entity class than can be associated via a relationship instance with instances of another entity class.

For a binary relationship class **R** between entity classes **A** and **B**, the maximum cardinality must be one of the following:

- **1:1** or **1-1** (read: "one-to-one")

For a 1:1 relationship class, an entity instance in entity class **A** is associated with **at most one** entity instance in entity class **B**, and an entity instance in entity class **B** is associated with **at most one** entity instance in entity class **A**.

This is actually the rarest maximum cardinality in practice, and you should be careful not to use it where it isn't really appropriate. However, assume you have a scenario with an `Employee` entity class and a `Company-car` entity class, and that it is a business rule in this scenario that every company car can be assigned to at most one employee (ever!), and that no employee can be associated with more than one company car (ever!). Then the maximum cardinality of this relationship class -- let's name it `assigned-car` -- is 1:1.

- **1:N** or **1-N** (read: "one-to-many")

For a 1:N relationship class, an entity instance in entity class **A** is associated with **possibly-many** entity instances in entity class **B**, and an entity instance in entity class **B**, however, can be associated with **at most one** entity instance in entity class **A**.

This is the most common maximum cardinality in practice. Assume a scenario of a university, in which an instructor can teach several class sections each semester, but each class section is only permitted to have a single instructor of record (official teacher, if you will). That is, an individual instructor can be related to several class sections, but each class section can be related to at most one instructor. Then the maximum cardinality of this relationship class -- let's name it `teaches` -- is 1:N.

(Be careful with these -- it matters very much which entity class is the 1, and which is the N, in such relationships!)

- **N:M** or **N-M** (read: "many-to-many")

For an N:M relationship class, an entity instance in entity class **A** is associated with **possibly-many** entity instances in entity class **B**, and an entity instance in entity class **B** is associated with **possibly-many** (and possibly a different number of) entity instances in entity class **A**.

This is a not-uncommon maximum cardinality in practice. Assume the scenario of a university, in which a student can have multiple majors, and a major can be majored in by more than one student. Then the maximum cardinality of this relationship class -- let's name it `majors-in` -- is M:N.

Now, how do you express these maximum cardinalities in an E-R diagram? This is a **big** point of variation, and can lead to definite issues of confusion if all those working on a project do not agree on a standard notation! For this course, you are required to indicate maximum cardinalities within an ERD as follows: we will put the 1 or M or N near the relationship line, near the entity class it goes with. This is straightforward for maximum cardinalities of 1:1 and M:N -- where, by the way, I do not care which entity gets the M and which gets the N -- but tricky for 1:N relationships. You put the 1 next to the entity class whose instance might be related to more than one of the other entity class' instances in that relationship -- and you put the N next to the entity class whose instance is only allowed to be related to at most one of the other entity class. That sounds strange, but it "reads" fairly well in the ERD, as I will demonstrate in the 1:N example below.

Consider the following examples:

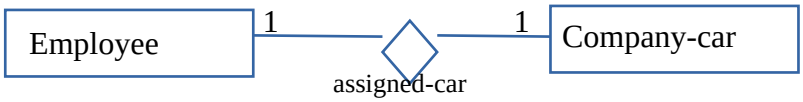

Figure 4 - ERD showing that relationship class assigned-car has a maximum cardinality of 1:1 between the entity classes of Employee and Company-car

By indicating the maximum cardinality by putting 1s near assigned-car's relationship line and near each entity class in Figure 4's ERD, we are saying that, in the assigned-car relationship, an employee instance may be related to at most one company-car instance, and that a company-car instance may be related to at most one employee instance. (An employee may have at most one company car assigned to them, and a company car may be assigned to at most one employee.)

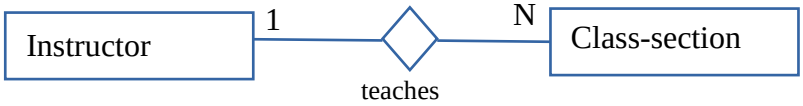

Figure 5 - ERD showing that relationship class teaches has a maximum cardinality of 1:N between the entity classes of Instructor and Class-section

By indicating the maximum cardinality by putting a 1 near the teaches relationship line near the Instructor entity class and putting an N near the teaches relationship line near the Class-section entity class in Figure 5's ERD, we are saying that, in the teaches relationship, an instructor instance may be related to more than one class-section instance, but that a class-section instance may be related to at most one instructor instance. (An instructor may teach many class sections, but a class section may have at most one instructor.)

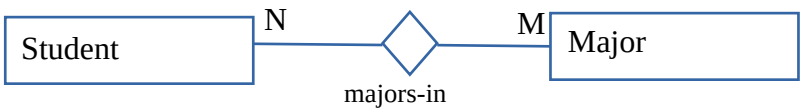

Figure 6 - ERD showing that relationship class majors-in has a maximum cardinality of N:M between the entity classes of Student and Major

By indicating the maximum cardinality by putting an N near the majors-in relationship line near the Student entity class and putting an M near the majors-in relationship line near the Major entity class in Figure 6's ERD, we are saying that, in the majors-in relationship, a student instance may be related to more than one major instance, and that a major instance may be related to more than one student instance. (A students may have more than one major, and a major may be majored-in by more than one student.)

As mentioned earlier, note that, for Figure 6's ERD, for the relationship class `majors-in` between entity classes `Student` and `Major`, I don't care which entity class has the N and which has the M. What is important is that one is N and one is M.

The **appropriate** maximum cardinality for a particular relationship is obviously dependent on the users' model that is being modeled by the relationship class. Consider again the `teaches` relationship class between `Instructor` and `Class-section`. If the business rules for the scenario note that a class-section must be taught by at most one instructor, then the relationship is 1:N as shown. However, if the business rules indicate that some courses can be team-taught, then this relationship would instead be M:N (in a scenario with that business rule, a class-section can be taught by multiple instructors, and an instructor can teach multiple course-sections), and its maximum cardinalities would be changed accordingly.

Note: as you are developing a model, you will often come upon such questions -- how many instructors can a course-section have? Can a company-car be assigned to more than one employee? etc. In the real world, you would consult with your client when such questions come up, record the answer in the business rules for the scenario, and then reflect that answer in your model. (For your course project, you will pretend to consult with your client, and record their "answers" in your business rules.) At the very least, you should record such assumptions as assumed business rules, so you will have them available if you do get a chance to ask later (or if some concern or problem comes up!)

Maximum cardinalities have important implications for the eventual database design/schema that will result; it is important to note them in an ERD, and it is important that they reflect the users' view of their world!

Minimum Cardinality

Maximum cardinalities indicate what can be the case at **most** -- how many instructors **can** a course section be taught by, at most? How many departments **can** a student major in, at most? And these are usually discussed in broad terms: one or many, 1:1, 1:N, N:M.

They do not indicate whether a relationship **must** exist for every entity instance within an entity class. That's what is indicated by **minimum cardinality**, and usually minimum cardinalities are 1 (the relationship must exist on that end) or 0 (the relationship doesn't have to exist on one end). Does an instructor have to teach any class sections? Does a class section have to have an instructor? Does a student have to have a major? Does a major have to have at least one student majoring in that major?

Like maximum cardinalities, you need to consider both "ends".

And if there is much variation between how maximum cardinalities are indicated in ERDs, there is even **more** variation between how minimum cardinalities are indicated in ERDs! The course-required notation for minimum cardinalities will be as follows:

- if each entity instance of an entity class must participate in that relationship, place a 1 or a dash or a short line directly on the relationship line (perpendicular to the relationship line) near the other entity class' rectangle.
- if each entity instance of an entity class is not required to participate in that relationship, place a 0 or oval or obviously-roundish-shape directly on the relationship line near the other entity class' rectangle.

Again, this sounds strange, but it "reads" fairly well in the ERD, as I will demonstrate in the example below.

For example, if an individual instructor isn't required to necessarily teach any class sections, but every class section must be taught by exactly one instructor, then here is what the ERD would look like with minimum cardinalities added:

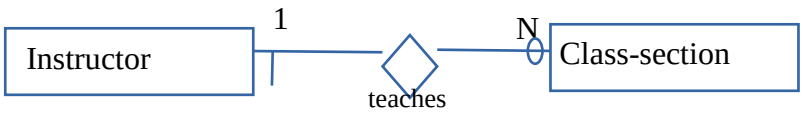

Figure 7 - ERD showing relationship class teaches' minimum cardinality between the entity classes of Instructor and Class-section

The oval on the teaches relationship line near the Class-section entity class means that each instructor is not necessarily required to be related to a class-section instance. The line on the teaches relationship line near the Instructor entity class means that a class-section instance must be related to an instructor. (An instructor does not necessarily have to teach any class-section, but a class section must have an instructor.)

Now consider the combination of minimum and maximum cardinalities in this example -- one could read the above as indicating that an Instructor instance may teach from zero to many class-sections, and a class-section instance must be taught by 1 and only 1 instructors (or, exactly one instructor).

These are the most fundamental basics of entity-relationship modeling; we'll continue with some useful additions to this in the next (non-SQL) reading packet.