MET CS 669 Database Design and Implementation for Business
Term Project Iteration 3

Contents

Iteration Introduction

In Iteration 2, you modeled your database design with structural database rules and a high-level ERD. In this iteration, you enhance your high-level design by adding a new type of relationship – specialization-generalization. You also start on a detailed design by developing an initial *DBMS Physical* ERD.

To help you keep a bearing on where you have been and where you are headed, let's again look at an outline of what you created in prior iterations, and what you will be creating in this and future iterations.

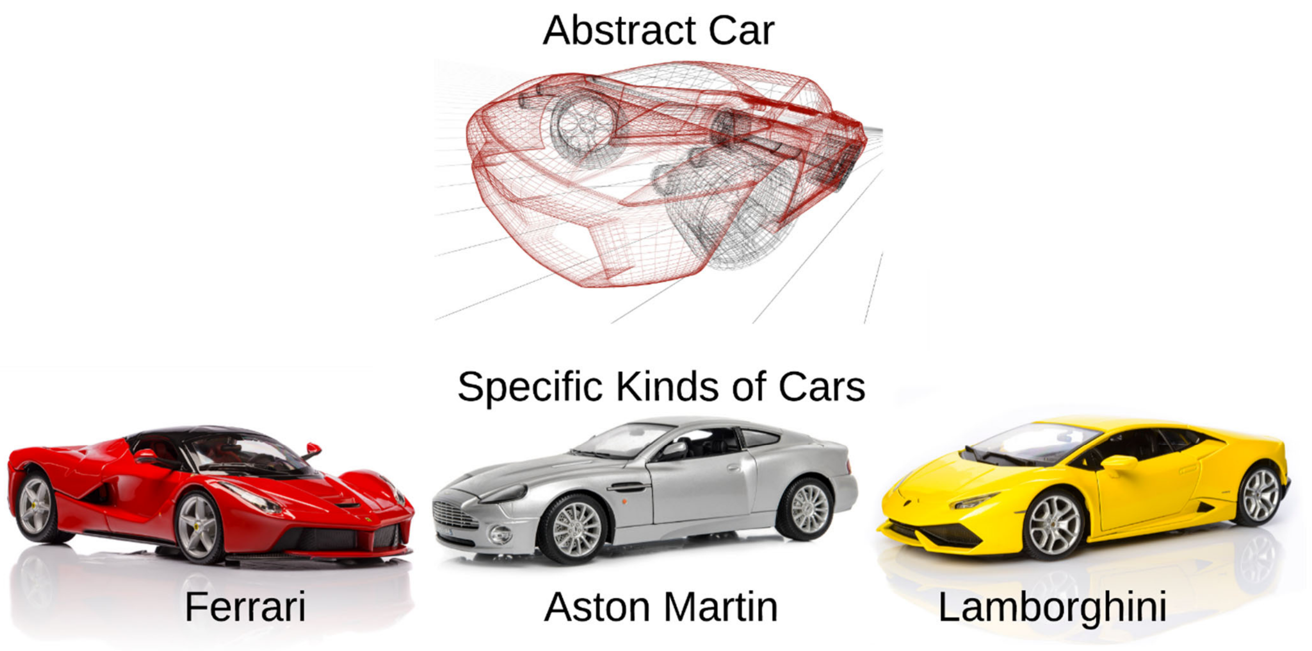
Prior Iterations	Iteration 1	*Project Direction Overview* – You provide an overview that describes who the database will be for, what kind of data it will contain, how you envision it will be used, and most importantly, why you are interested in it. *Use Cases and Fields* – You provide use cases that enumerate steps of how the database will be typically used, also identify significant database fields needed to support the use case. *Summary and Reflection* – You concisely summarize your project and the work you have completed thus far, and additionally record your questions, concerns, and observations, so that you and your facilitator or instructor are aware of them and can communicate about them.
	Iteration 2	*Structural Database Rules* – You define structural database rules which formally specify the entities, relationships, and constraints for your database design. *Conceptual Entity Relationship Diagram (ERD)* – You create an initial ERD, the universally accepted method of modeling and visualizing database designs, to visualize the entities and relationships defined by the structural database rules.
Current Iteration	Iteration 3	*Conceptual Extended Entity Relationship Diagram (EERD)* – You add specialization-generalization into your conceptual ERD and structural database rules. *Initial DBMS Physical ERD* – You create an initial DBMS physical ERD, which is tied to a specific relational database vendor and version, with SQL-based constraints and datatypes.
Future Iterations	Iteration 4	*Full DBMS Physical ERD* – You define the attributes for your database design and add them to your DBMS Physical ERD. *Normalization* – You normalize your DBMS physical ERD to reduce or eliminate data redundancy. *Database Structure* – You create your tables, sequences, and constraints in SQL.
	Iteration 5	*Reusable, Transaction-Oriented Store Procedures* – You create and execute reusable stored procedures that complete the steps of transactions necessary to add data to your database. *Questions and Queries* – You define questions useful to the organization or application that will use your database, then write queries to address the questions. *Index Placement and Creation* – To speed up performance, you identify columns needing indexes for your database, then create them in SQL.
	Iteration 6	*History Table* – You create a history table to track changes to values, and develop a trigger to maintain it. *Data Visualizations* – You tell effective data stories with data visualizations.

Before you proceed to add more sections to your design document, first revise the sections already completed in prior iterations, if necessary. If there are any issues with your database design, it is critical you first correct them, before mapping the design into a DBMS physical ERD.

Specialization-Generalization Relationships

While most database designs contain far fewer specialization-generalization relationships than associations, the relationship itself is nevertheless critical when it is needed. The kind of relationship you learned about previously is association, where one entity is in some way associated with another entity. A specialization-generalization relationship allows some entities to specialize another more abstract entity, where the specialized entities are said to be a more specific kind of the abstract entity. This kind of relationship is quite different from an associative relationship.

Let's take a look an example. The more abstract entity could be Car and the specific kinds of cars could be a Ferrari, an Aston Martin, and a Lamborghini. This is illustrated in the following image.



The abstract Car has properties that apply to all cars, such as the fact that all cars have tires, windows, body panels, and engines. Each specific kind of car has additional properties that apply only that car, such as unique shapes that apply only to Ferraris, unique color shades for the Lamborghini, or special engine configurations that apply only to Aston Martins. Each of the specific kinds of cars – Ferraris, Aston Martins, and Lamborghinis – have everything that the abstract Car entity has, such as tires, windows, body panels, and engines. But not all cars have the unique shapes, colors shades, or engine configurations that these specific kinds of cars have. Thus, the specialization-generalization relationship gets its name from the fact that some entities *specialize* another *generalized* entity.

Informally, we say that the specialization-generalization relationship is an "is a" relationship because the specialized entities *are a* special kind of the generalized entity. From our example, we say that a Ferrari *is a* car, an Aston Martin *is a* car, and a Lamborghini *is a* car. This kind of relationship is very different from association, which is informally a “has a” relationship. A Ferrari does not *have* a car; a Ferrari *is* a car. For an associative example, in the structural database rule fragment “A person may own many cars”, we could substitute “has a”, for example, “A person may have many cars”, and the sentence still makes sense. We would never say, “A person is a car” because that does not make any sense. We see that there are two very different kinds of relationships – associative and specialization-generalization – and each means the entities are related in different ways.

Now that you understand the concept, we need to flush out database specific terms. The abstract entity is termed the *supertype*, and the specialized entities are termed the *subtypes*. Using our prior example, Car is the supertype, and Ferrari, Aston Martin, and Lamborghini are the subtypes. If you’re familiar with the object-oriented (OO) model or languages, you might be thinking that we should use the term “parent” for the abstract entity, and “child” for the specialized entities since that’s how they are termed in the OO model. Because database modeling and object-oriented modeling were developed independently, they ended up with different terms. Database modeling already uses the terms “parent” and “child” for associative relationships, so we need different terms for specialization-generalization relationships. Knowing the precise terminology will help you keep the concepts straight.

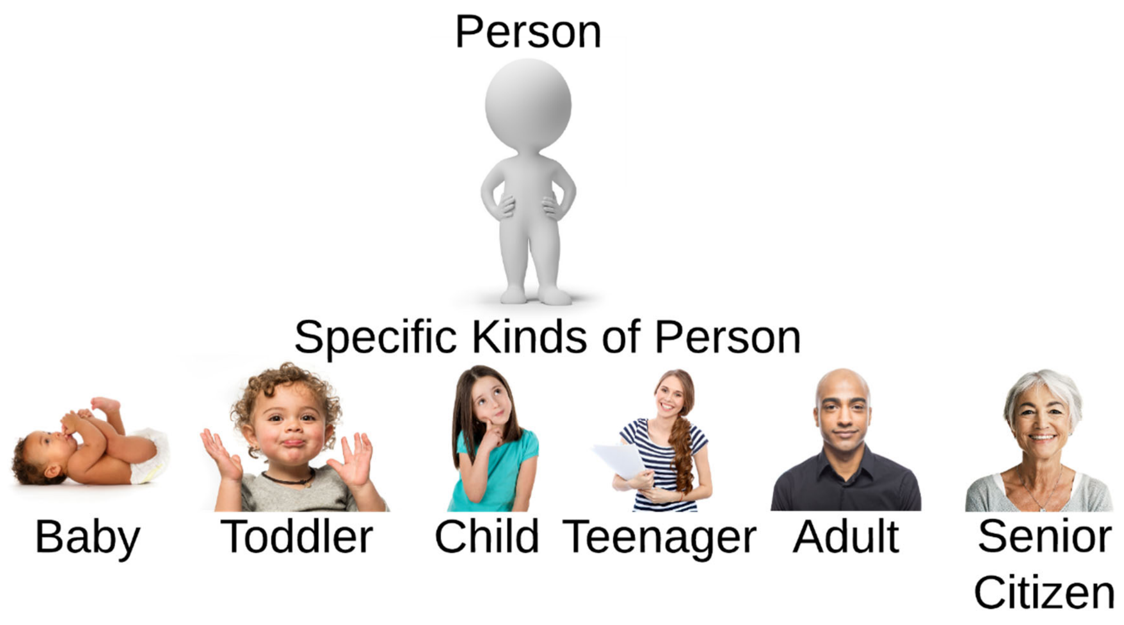
Specialization-Generalization Constraints

The specialization-generalization relationship has two constraints – completeness and disjointness. You’re already familiar with the associative constraints – participation and plurality – but keep in mind that the completeness and disjointness constraints are different then these. The completeness constraint, as its name suggests, is an indication of whether the subtypes provided in the relationship is complete (i.e. exhaustive). If the list of subtypes is complete, we say that the relationship is *totally complete*. If the list of subtypes is incomplete, we say that the relationship is *partially complete*.

Completeness Constraint

Using our prior example, we can ask whether Ferraris, Aston Martins, and Lamborghinis are a complete list of cars. The obvious answer is no. There are many other kinds of cars, so we say that relationship is partially complete. When a relationship is partially complete, a data item is permitted to be described as the supertype alone. For example, if our data item is a particular Honda Accord with VIN 1GKCS18RXJ8586391, we can claim that “Honda Accord with VIN 1GKCS18RXJ8586391 is a car”. We cannot correctly claim that “Honda Accord with VIN 1GKCS18RXJ8586391 is a Ferrari” or “Honda Accord with VIN 1GKCS18RXJ8586391 is an Aston Martin” or “Honda Accord with VIN 1GKCS18RXJ8586391 is a Lamborghini”. Data items that are described as the supertype alone are allowed only when the relationship is partially complete.

When a relationship is totally complete, a data item must be described as the supertype and at least one subtype. Let’s take a look at a totally complete example, “Each person is an infant, toddler, child, teenager, adult, or senior citizen”, which is illustrated in the following image.



Person is the supertype, and Infant, Toddler, Child, Teenager, Adult, and Senior Citizen are the subtypes. The list of subtypes is exhaustive, that is, a person must be a baby, toddler, child, teenager, adult, or senior citizen. There is no other kind of person. A data item must be described as a Person and as one of the listed subtypes. For example, if we have a particular data item representing Michelle Floyd who is a teenager, we can correctly claim, “Michelle Floyd is a teenager, and a teenager is a person.” While it is also true that “Michelle Floyd is a person” is accurate, it is more correct to indicate the specific kind of person she is, a teenager. And every other data item will be a baby, toddler, child, teenager, adult, or senior citizen. There will exist no data item that is not one of these listed subtypes.

Disjointness Constraint

A disjointness constraint indicates whether one data item can be described as multiple subtypes simultaneously. If each data item can be described as only one subtype, we say the relationship is *disjoint*. If each data item can be described as multiple subtypes simultaneously, the relationship is *overlapping*. In both of the examples we have seen thus far – Car and Person – the relationship is disjoint. A car is either a Ferrari, Aston Martin, or Lamborghini; one car cannot be both a Ferrari and a Lamborghini. Likewise, a person is either a baby, toddler, child, teenager, adult, or senior citizen; one person cannot be both a baby and an adult.

An example of an overlapping relationship is, “A sports fan can be a baseball fan, basketball fan, soccer fan, several of these, or none of these.” This is illustrated in the following image.

Sports Fan
Specific Kinds of Sports Fans
Baseball Fan
Basketball Fan
Soccer Fan

Sports Fan is the supertype, and Baseball Fan, Basketball Fan, and Soccer Fan are the subtypes. It is quite plausible that the same person can be a fan of multiple sports. For example, perhaps Bobbi Quinstone is a fan of both baseball and basketball, so we could say "Bobbi Quinstone is a baseball fan and a basketball fan". Perhaps Loretta Fiora is a fan of all three sports, so we could say "Loretta Fiora is a baseball fan, a basketball fan, and a soccer fan". This relationship is overlapping because one data item can be described as multiple subtypes.

If a relationship is overlapping, it does not mean that every data item *must* be described as multiple subtypes. It means that every data item *is permitted* to be described as multiple subtypes. If the relationship is both partially complete and overlapping, some data items could be described as the supertype only, while others are described as the supertype and several subtypes. If the relationship is totally complete and overlapping, some data items may be described as a single subtype while others are described as multiple subtypes. It is important to keep the two constraints distinct in your mind, as they work together but constrain two different aspects of the relationship.

Specialization-Generalization Structural Database Rules

Structural database rules are phrased differently for specialization-generalization relationships. Rather than describing a peer-to-peer associative relationship, we are describing a hierarchical relationship. Let's illustrate by using some of the same examples mentioned previously in this document. For the first example of Ferraris, Aston Martins, and Lamborghinis, we can phrase the structural database rule as follows.

A car is a Ferrari, Aston Martin, Lamborghini, or none of these.

That's it! There is no need to describe various perspectives as we do with associative structural database rules. We start with "A car…" to indicate that Car is the supertype. We then use "A car *is a"* to indicate that there are subtypes that are specific kinds of Car. We then name the subtypes, “A car is a *Ferrari, Aston Martin, Lamborghini*”. Thus far with “A car is a Ferrari, Aston Martin, Lamborghini…”, we’ve named the supertype, subtypes, and indicated that there is a specialization-generalization relationship.

The last part of the sentence is saved for the completeness and disjointness constraints. Notice the last phrase in “A car is a Ferrari, Aston Martin, Lamborghini, *or none of these”* is indicating that the relationship is partially complete, that is, the list of subtypes is not exhaustive. There are other kinds of cars other than those identified. The lack of another phrase indicates that the relationship is disjoint, that is, there is nothing indicating that a single car can be more than one of the given subtypes simultaneously. *If* we had stated, “A car is a Ferrari, Aston Martin, Lamborghini, *several of these*, or none of these”, we would have been indicating that the relationship is overlapping. But we did not use the phrase “several of these”, so it is disjoint.

Just as with the associative structural database rules, we can phrase them in many different ways, but the entities, relationship, and constraints must be entirely unambiguous. The precise wording may vary. For example, we could use the phrase “some of these” instead of “several of these”, or “not any of these” instead of “none of these”. Although one might envision many different ways of reordering a structural database rule for the specialization-generalization relationship, I recommend sticking to the ordering illustrated in the first example:

A supertype is a subtype1*,* subtype2*, …* subtypeN*,* several of these, or none of these.

The reason is, this format unambiguously states the entities, relationships, and constraints. If you use this format, you and your facilitator or instructor can focus on the design rather than exactly on how things are worded. Of course, you would only use “several of these” to indicate an overlapping relationship, only use “none of these” to indicate a partially complete relationship, and remove these phrases if they don’t apply to the particular relationship.

The aforementioned person and sports fan examples already have their structural database rules listed out. For the person example, the rule is:

Each person is an infant, toddler, child, teenager, adult, or senior citizen.

Person is identified as the supertype, Infant, Toddler, Child, Teenager, Adult, and Senior Citizen are identified as the subtypes, and the lack of any other phrases indicates the relationship is totally complete and disjoint. Every person must be one of those subtypes, and can only be one.

For the sports fan example, the rule is:

A sports fan can be a baseball fan, basketball fan, soccer fan, several of these, or none of these.

Sports fan is identified as the supertype, Baseball fan, Basketball fan, and Soccer fan are identified as the subtypes. The phrase “several of these” indicates the relationship is overlapping, that is, that a Sports fan could be a fan of several of these sports at the same time. The phrase “none of these” indicates that the relationship is partially complete, so there can be kinds of sports fans other than those listed.

Specialization-Generalization Structural Database Rule Derivation

Let’s take a look at the process of deriving specialization-generalization structural database rules for TrackMyBuys.

Adding Specialization-Generalization to TrackMyBuys

First, I look again at my existing use cases. After some careful thought, the first one peeks my interest for this purpose.

Account Signup/Installation Use Case

1. The person visits TrackMyBuys’ website or app store and installs the application.
2. The application asks them to create an account when its first run.
3. The user enters their information and the account is created in the database.
4. The application asks them to install browser plugins so that their purchases can be automatically tracked when they make them.

Something that would be useful for TrackMyBuys is to have different kinds of accounts. Perhaps I will offer a free account that has some limitations on the functionality, and a paid account which offers all of the features. I modify the use case as follows.

Account Signup/Installation Use Case (New)

1. The person visits TrackMyBuys’ website or app store and installs the application.
2. The application asks them to create either a free or paid account when its first run.
3. The user selects the type of account and enters their information and the account is created in the database.
4. The application asks them to install browser plugins so that their purchases can be automatically tracked when they make them.

Notice that #2 now mentions free and paid accounts. I derive a fourth structural database rule to support the change to the use case as follows.

4. An account is a free account or a paid account.

This specialization-generalization rule turned out to be quite short, but does capture the intent. My database only has two kinds of accounts – free and paid – and that is the complete list. The relationship is totally complete. The account must be either free or paid, so the relationship is disjoint. I did not put any of the verbiage such as “several of these” or “none of these” since the rule is totally complete and disjoint.

As I was thinking about specialization-generalization, it also became plain to me is that I will want treat different kinds of purchases differently (very broadly). I will likely have different data to store for online purchases than for face-to-face purchases. My original purchase use case is as follows.

Automatic Purchase Tracking Use Case

1. The person visits an online retailer and makes a purchase.
2. The TrackMyBuys browser plugin detects that the purchase is made, and records the relevant information in the database such as the purchase date, price, product, store, etc.

To support the different kinds of purchases, I modify it as follows.

Automatic Purchase Tracking Use Case (New)

1. The person visits an online retailer and makes a purchase.
2. The TrackMyBuys browser plugin detects that the purchase is made, and records the relevant information in the database such as whether the purchase is online or face-to-face, the purchase date, price, product, store, etc.

Notice that #2 now mentions the online and face-to-face purchase types.

I now derive a 5th structural database rule from this change to the use case.

5. A purchase is an online purchase, a face-to-face purchase, both, or none of these.

This relationship allows my database to have both types, but I am not entirely confident these are the only types available. So, I made this relationship partially complete to give users the flexibility to purchase in ways other than these two listed. That is why I used the phrase “or none of these”. Also, I know that many stores have both a face-to-face and online presence, for example Walmart, Target, Barnes & Noble, just to name a few. So, I made the relationship overlapping in case a person makes purchase from the same store (such as Walmart) both online and face-to-face. This is why I used the phrase “both”.

Now I have five structural database rules, including my original three associative, plus the two I just created.

1. Each purchase is associated with an account; each account may be associated with many purchases.
2. Each purchase is made from a store; each store has one to many purchases.
3. Each purchase has one or more products; each product is associated with one to many purchases.
4. An account is a free account or a paid account.
5. A purchase is an online purchase, a face-to-face purchase, both, or none of these.

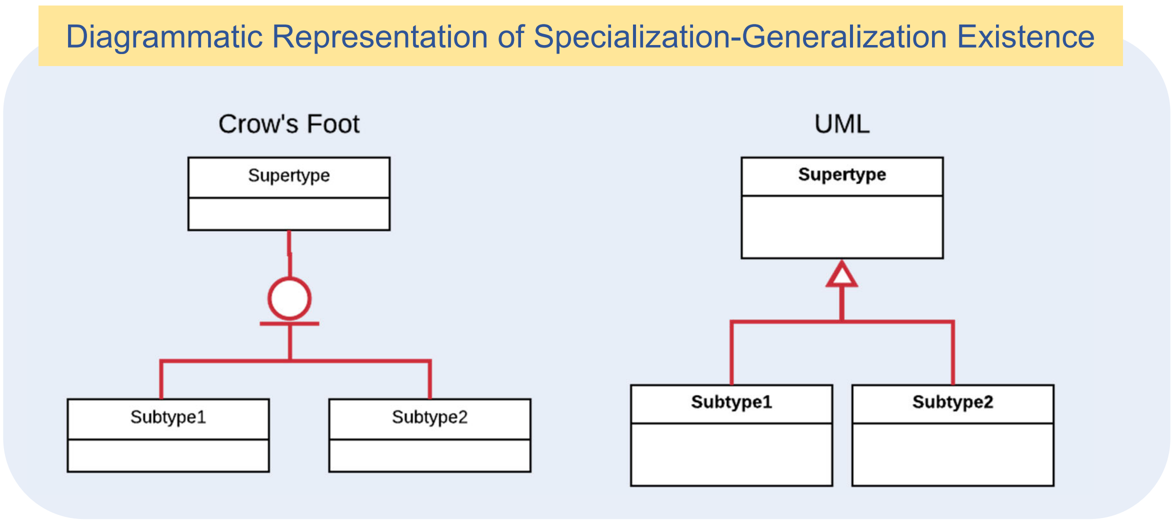
Adding Specialization-Generalization to Your Structural Database Rules

For example, my original use cases from Iteration 1 did not support any specialization-generalization relationships, so I modify two use cases to support two of them. If your original use cases do not support specialization-generalization, you have the option to either add new use cases, or modify existing use cases. Note that at least 10 entities should be identified in your structural database rules in this iteration – the 8 or more you had from the last iteration, and at least two subtypes from this iteration.

Diagramming Specialization-Generalization Relationships

There are different notations for diagramming specialization-generalization relationships for both Crow’s Foot and UML, when compared to associative relationships. UML natively supports specialization-generalization. Crow’s Foot style does not support specialization-generalization relationships, and so the notations we use are categorized as “extensions to Crow’s Foot”. For this reason, if an entity-relationship diagram contains at least one specialization-generalization relationship, the diagram can be termed an extended entity-relationship diagram (EERD). Some still describe such a diagram as an ERD rather than EERD, so the use of the term EERD is not universal.

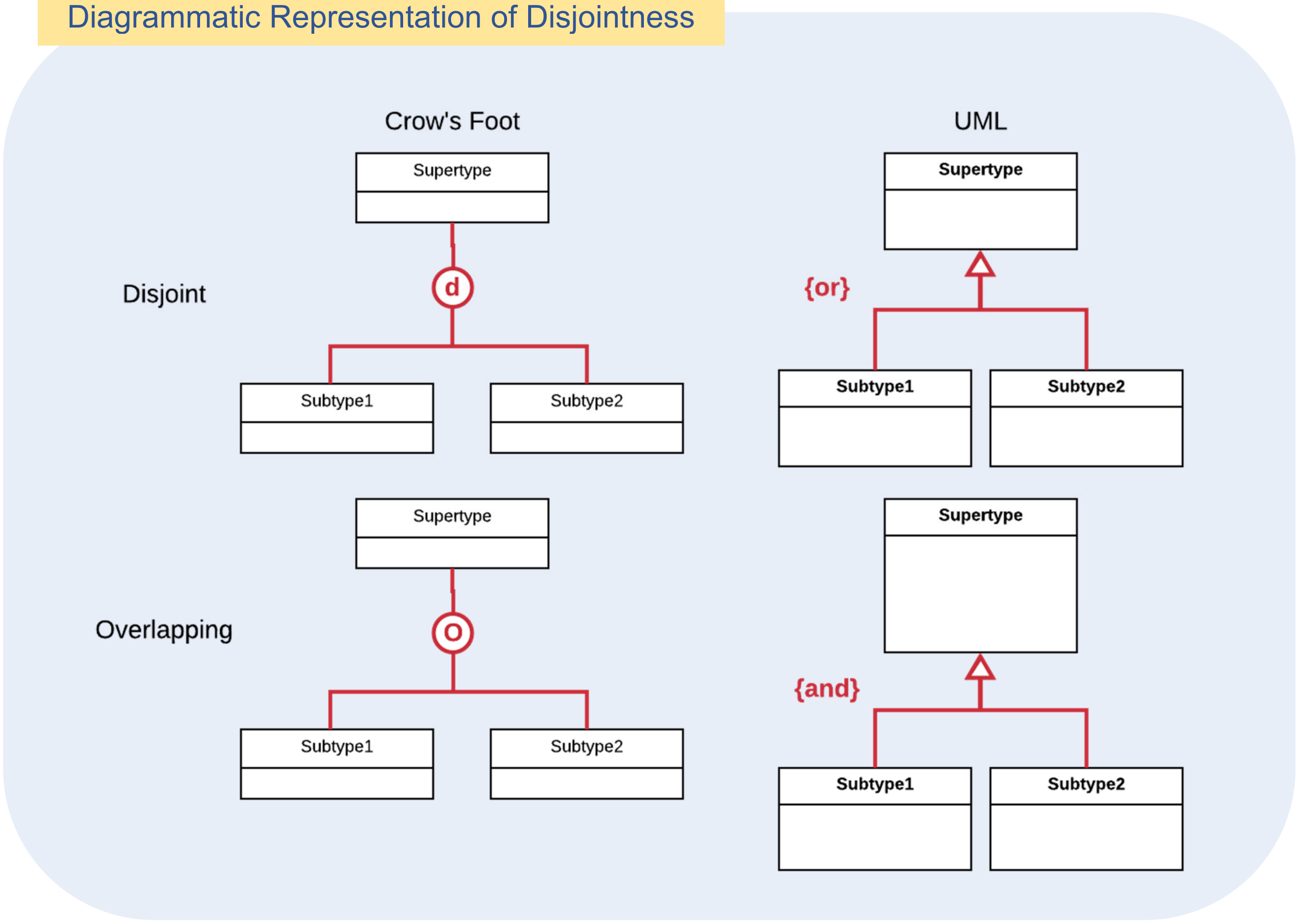
The existence of the specialization-generalization relationship is diagrammed in the following figure.


Diagrammatic Representation of Specialization-Generalization Existence
Crow's Foot
Supertype
Subtype1
Subtype2
UML
Supertype
Subtype1
Subtype2

Notice that extensions to Crow’s Foot uses a circle and a bar in the middle of the relationship line, and UML uses a triangle just beneath the supertype. There is no need to give a relationship name to specialization-generalization relationships because it’s always the same, “is”.

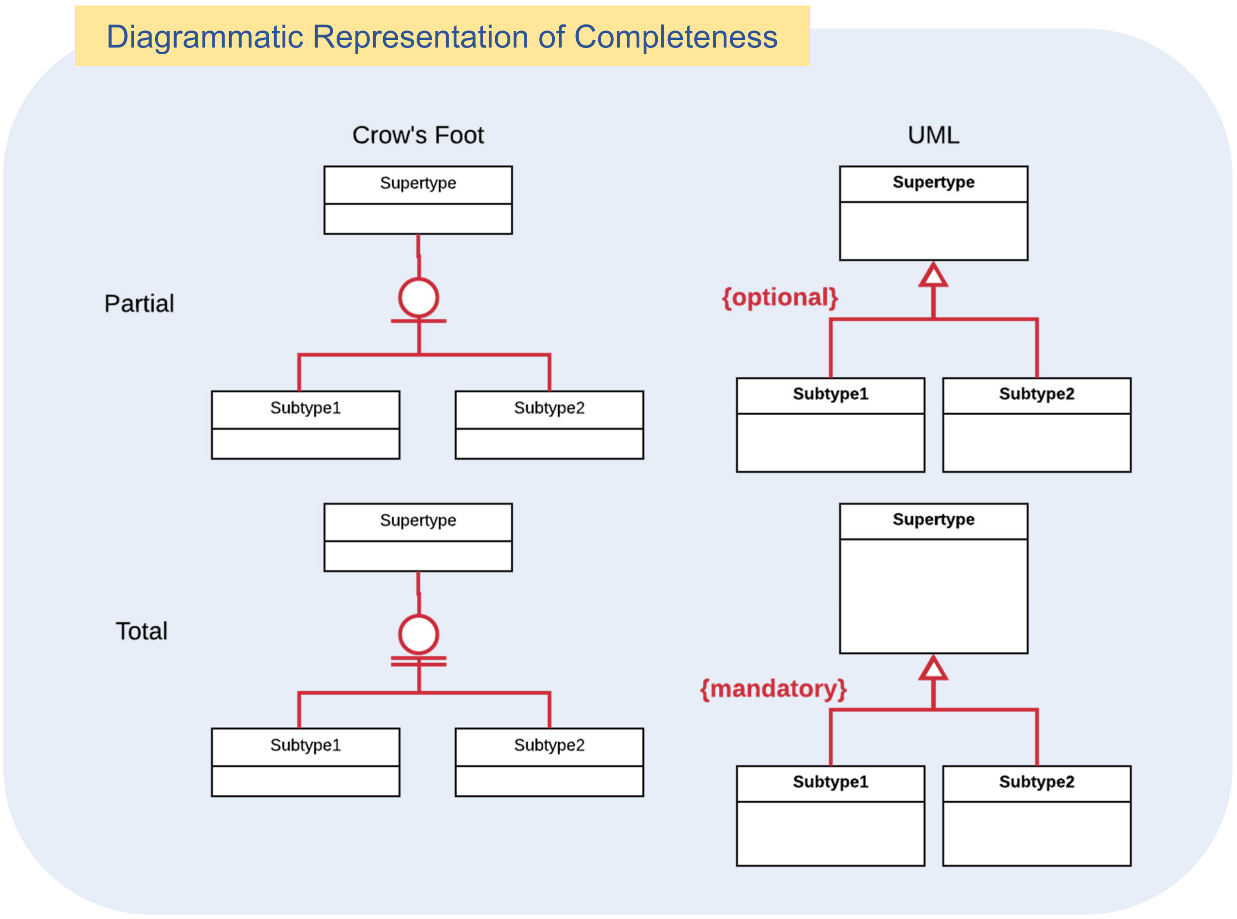
The disjointness constraint is diagrammed in the following figure.

Diagrammatic Representation of Disjointness



Notice that extensions to Crow's Foot uses a "d" in the circle to indicate the relationship is disjoint, and an "O" to indicate the relationship is overlapping. UML uses the word "or" within braces to indicate the relationship is disjoint (you can read this as "either/or"), and "and" to indicate the relationship is overlapping.

The completeness constraint is diagrammed in the following figure.

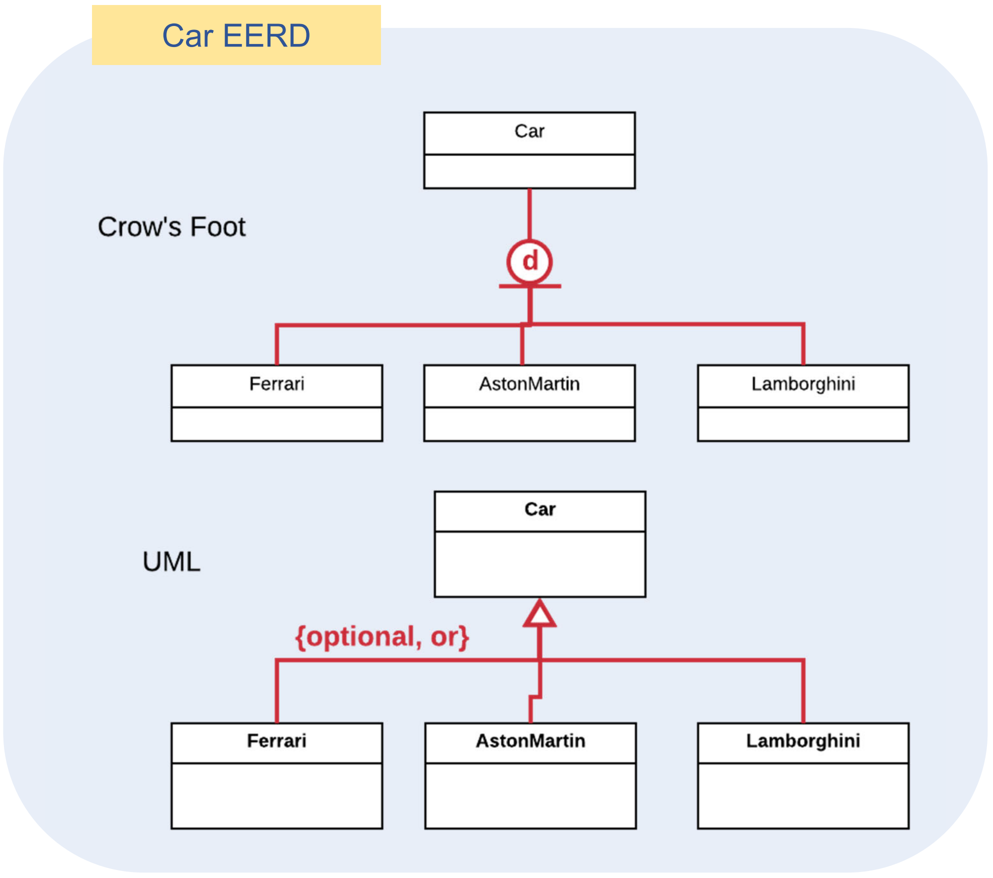


Notice that extensions to Crow's Foot uses a single bar to indicate partial completeness, and two bars for total completeness. UML uses the word "optional" in braces to indicate partial completeness, and "mandatory" to indicate total completeness. The UML words may seem confusing until you understand the perspective. It's stating effectively that the relationship is either optional or mandatory to the supertype. If the relationship is mandatory to the supertype, then the relationship is totally complete, because every supertype must have a corresponding subtype (the list of subtypes is exhaustive if so). If the relationship is optional to the supertype, then there can exist a supertype data item that has no corresponding subtype data item. Do not let the UML words confuse you however; they are just an alternate way to indicate completeness, but do not change the meaning of completeness.

Example Specialization-Generalization Diagrams

So that you can see more complete EERDs, let's take a look at creating diagrams for the example structural database rules used previously in this document.

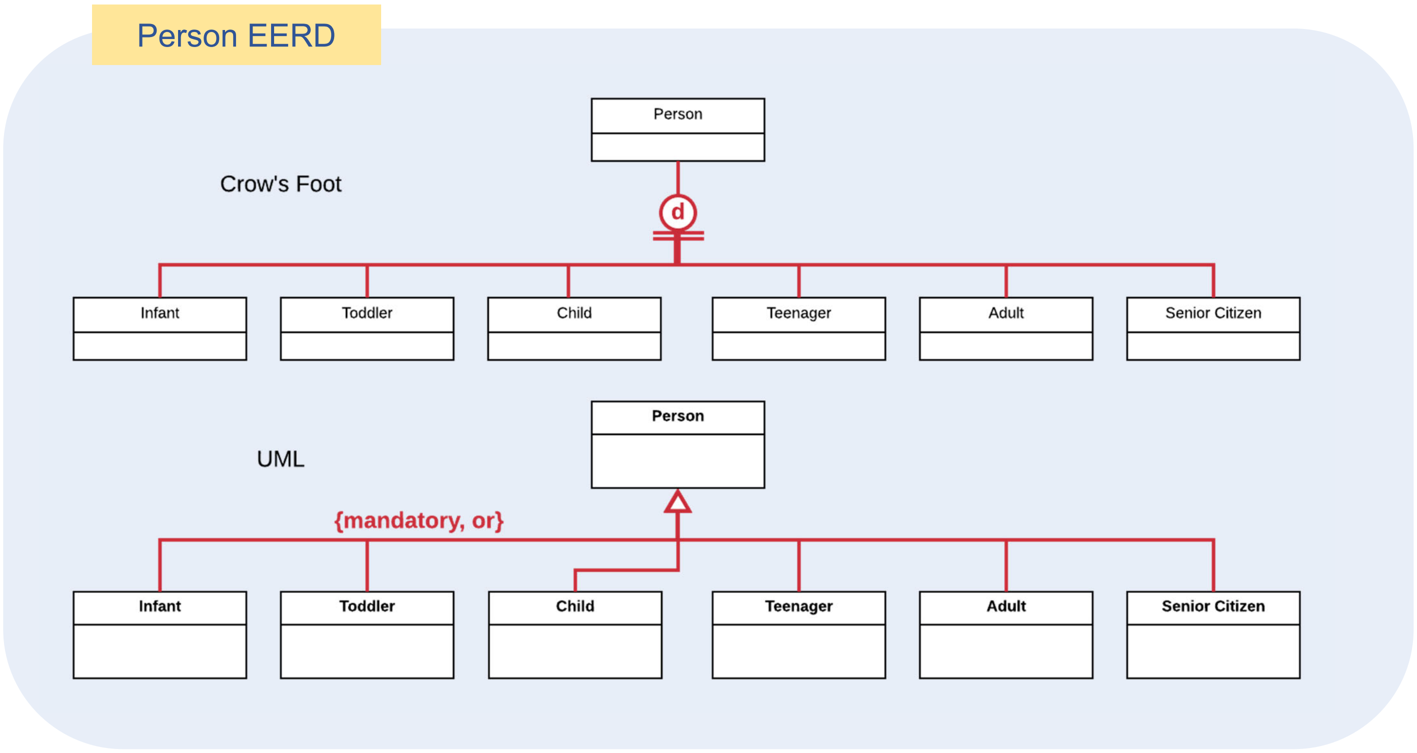
We diagram “A car is a Ferrari, Aston Martin, Lamborghini, or none of these” as follows.



Notice that for the Crow’s Foot EERD, we use the “d” to indicate that the relationship is disjoint, because a car can only be one of these, but not multiple of these simultaneously. We use a single bar to indicate that the relationship is partially complete, since there are other kinds of cars other than those listed. For UML, we use the word “or” to indicate the relationship is disjoint, and the word “optional” to indicate the relationship is partially complete.

You will also notice that we removed the space in “Aston Martin”. The reason is, we need these names to be legal database identifiers since we are going to put them into a database. Removing the space and keeping the capital to make “AstonMartin” is known as CamelCase, a common technique in programming. In CamelCase, the first letter of each word is capitalized, and there are no spaces. You will also see an alternative strategy for databases, which is to replace the spaces with underscores. “Aston Martin” would become “Aston_martin” using that strategy. Both are popular strategies and you can use either one. Some databases uppercase all table and column names in result sets, and generally if you’re using such a database, you will get better readability if you use the underscore strategy, because it’s easer to read “ASTON_MARTIN” than “ASTONMARTIN”, for example.

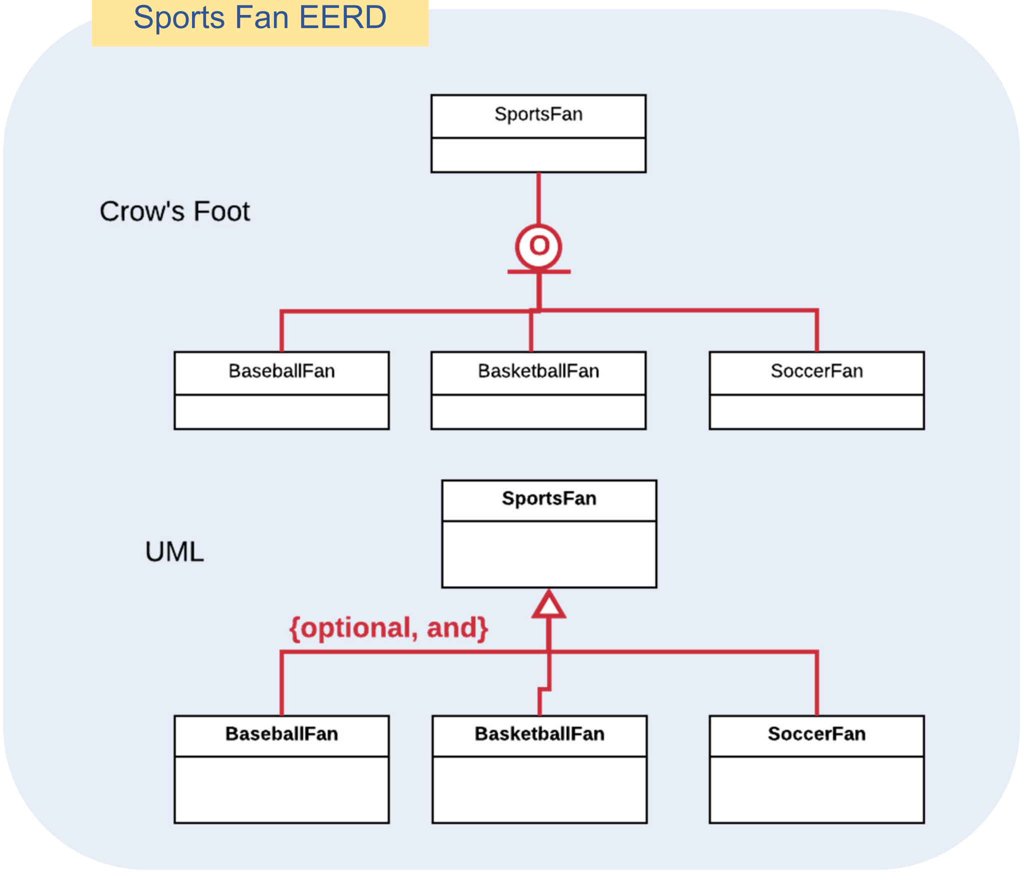
We diagram “Each person is an infant, toddler, child, teenager, adult, or senior citizen” as follows.



Using Crow's Foot, we use a "d" to indicate the relationship is disjoint, since a person cannot be more than one of these at the same time. We use double bars to indicate the relationship is totally complete, since the list is exhaustive; a person must fit into one of these categories. Using UML, we use the word "or" to indicate the relationship is disjoint, and "mandatory" to indicate the relationship is totally complete.

We diagram "A sports fan can be a baseball fan, basketball fan, soccer fan, several of these, or none of these" as follows.

Sports Fan EERD



Using Crow's Foot, we use the "O" to indicate that the relationship is overlapping, since the same sports fan can be fans of multiple sports. We use the single bar to indicate the relationship is partially complete, since there are other kinds of sports people are fans of other than those listed. Using UML, we use the word "and" to indicate the relationship is overlapping, and the word "optional" to indicate the relationship is partially complete.

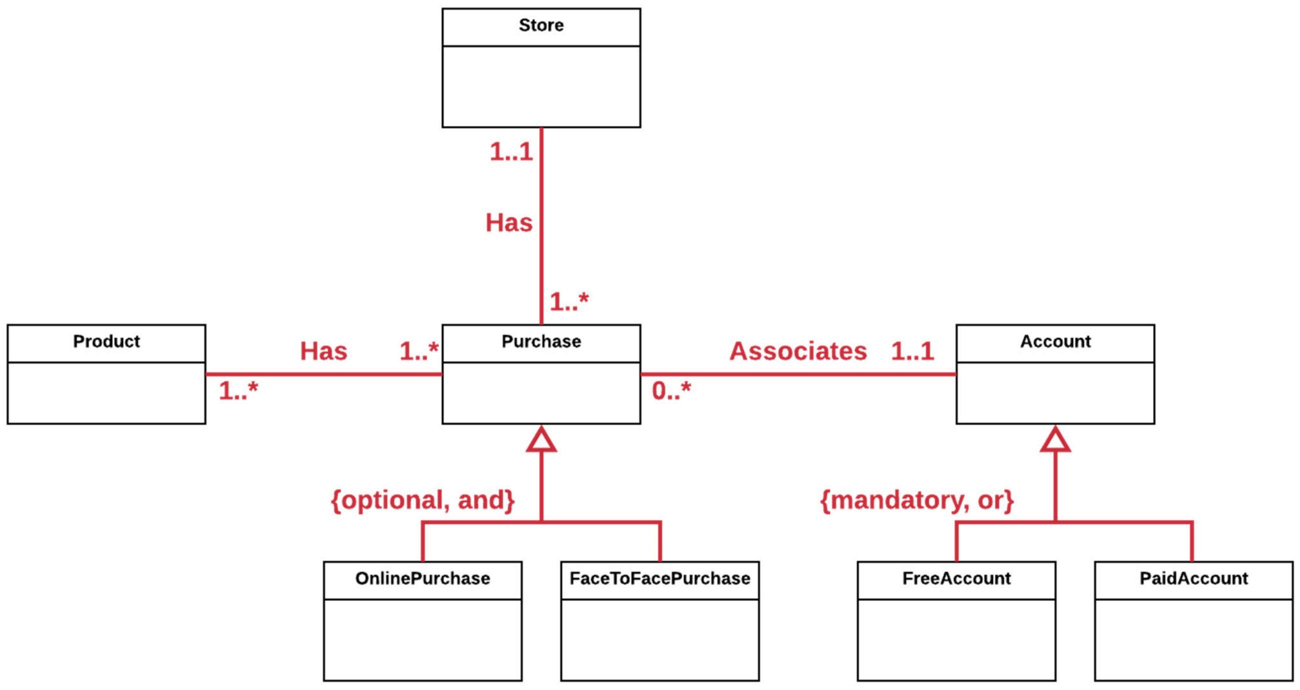
Now look at how I add in the specialization-generalization rules into the TrackMyBuys diagram.

TrackMyBuys ERD with Specialization-Generalization

Here are the associative structural database rules I came up, relisted below.

1. An account is a free account or a paid account.
2. A purchase is an online purchase, a face-to-face purchase, both, or none of these.

I then add these on diagrammatically to the initial ERD.



You see the two new entities under Account – FreeAccount and PaidAccount – and that the relationship is totally complete by use of the "mandatory" word, and disjoint by use of the "or" word. You also see the two new entities under Purchase – OnlinePurchase and FaceToFacePurchase – and that the relationship is partially complete by use of the "optional" word, and overlapping by use of the "and" word. These additions capture the two new structural database rules and use specialization-generalization.

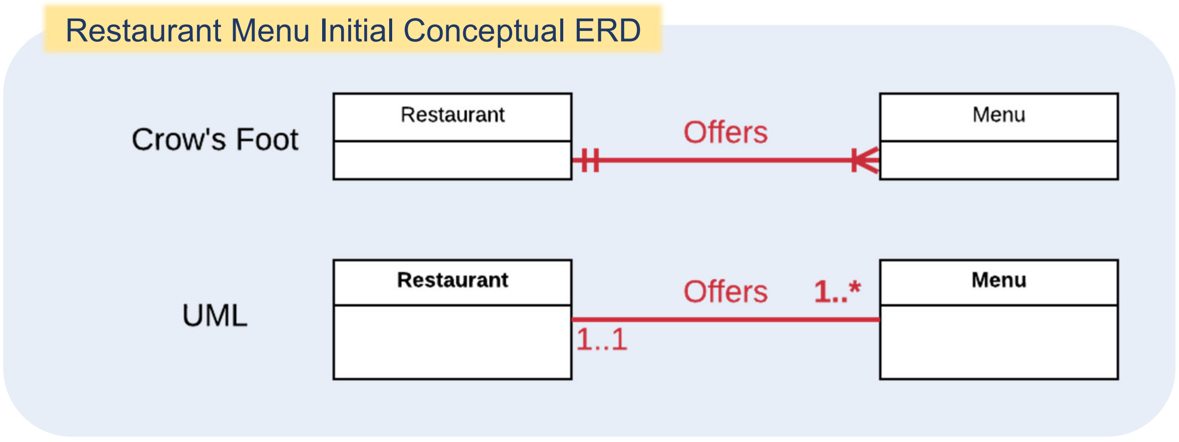
Representing Specialization-Generalization in Your ERD

Enhance, if necessary, any existing entities and relationships. Add in specialization-generalization into your conceptual ERD. Note that at least 10 entities are required in your conceptual ERD in this iteration – the 8 or more you had from the last iteration, and at least two subtypes from this iteration.

Levels of ERDs

There are three significant levels of ERDs – conceptual, logical, and database physical – and each serves a different purpose. A conceptual ERD represents the entities and relationships we perceive in our minds. There are broadly two classes of conceptual ERDs – initial and mature. When an initial conceptual ERD is created, we are only attempting to diagram the entities and relationships. We do not know what the attributes are, nor are they necessary, at this stage. This is what you created in Iteration 2.

As a reminder, a conceptual ERD can look like the following example, "Each restaurant offers one or more menus; each menu is offered at a restaurant."



Notice that only the entities, relationships, and relationship constraints are illustrated in the diagram.

SQL-relational constraints, attribute datatypes, synthetic keys, and other items related to the relational model are excluded. When a conceptual ERD matures through several iterations of design, we may add attributes, though in this course we only ask for initial conceptual ERDs.

We say that a conceptual ERD depends upon the problem domain only, because it is only tied to the rules of the organization we are modeling. A conceptual ERD is not tied to a relational database, which is why SQL-based constraints such as primary and foreign key constraints are not present, and why SQL-specific datatypes such as varchar or decimal are not present. Peter Chen introduced the base concepts for ERDs in his "The Entity Relationship Model – Toward a Unified View of Data" article in the later 1970s. He intentionally defined the highest-level model as abstracted from the specific implementation model (such as relational), so that the higher-level model lines up with the entities and relationships in our minds rather than the structural requirements of implementation. He envisioned that the higher-level model would be mapped to a lower-level model specific to the implementation model, such as the relational model, in a different level. This highest-level model has evolved as the conceptual ERD we have today.

A *logical* ERD depends upon the problem domain *and* is tied to the relational model, so we can think of a logical ERD as a visual representation of a relational schema. In logical ERDs, SQL-relational constraints including primary and foreign key constraints are included. Synthetic keys, if used, are also included. Each many-to-many relationship must be mapped to two one-to-many relationships (this will be described in more detail later in this document). Although logical ERDs are tied to the relational model, they are not tied to any particular database vendor and so are considered database agnostic. While attributes and cross-database datatypes are included, database-specific datatypes are not included.

A *DBMS physical* ERD depends upon the relational model *and* is tied to a specific database vendor and version, such as a particular version of Postgres, SQL Server, or Oracle. Of course, SQL-based constraints and synthetic keys are included as they are in logical ERDs. Database-specific datatypes are included in this type of ERD.

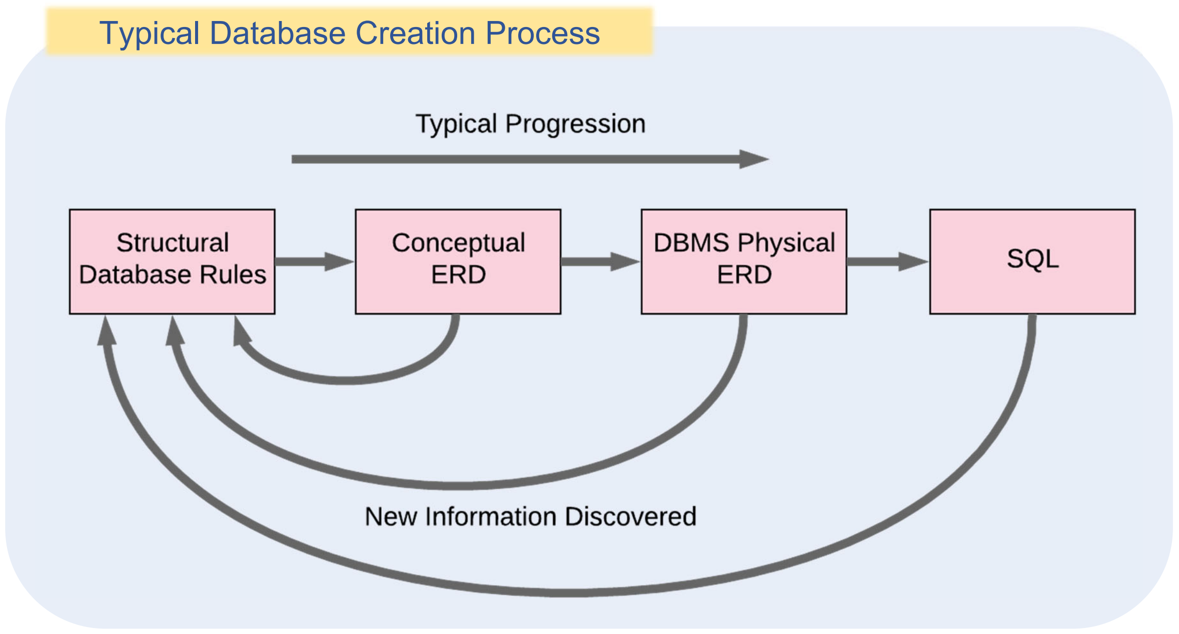
Some organizations create and maintain all three levels, and some organizations create only the conceptual and DBMS physical ERDs. The conceptual ERD serves two related purposes. First, it allows you to design your entities and relationship without simultaneously considering specific database limitations and structure. Second, it allows you to communicate your database design to less technical people, such as stakeholders and managers, who may not understand the database specific terminology, but do understand entities and relationships at a higher-level. That communication is critical because you need other people to confirm your design and make suggestions for improvement. The DBMS physical ERD provides a means for you to directly implement your database in SQL, since it contains the tables, keys, attributes, datatypes specific to the DBMS, and SQL-relational constraints.

Design Process

The process of creating a database usually works in a specific sequence. The structural database rules are defined and analyzed first. Next, the conceptual ERD is used to formally visualize and represent the structural database rules. Next, the DBMS physical ERD is created from the conceptual ERD following almost mechanical steps (which will be described later in this document). The database is then implemented in SQL directly following the definitions from the DBMS physical ERD. If a logical ERD is involved, it sits between the conceptual and DBMS physical ERDs.

This process usually happens iteratively. New information may be discovered at any step, and when discovered, we modify the structural database rules accordingly and adjust the ERDs and SQL as needed. Ideally, we make most the adjustments before we implement any SQL. It's very important that we always keep the structural database rules, the ERDs, and the SQL in-sync with each other. It would be confusing to all involved if the conceptual ERD does not convey the same as the structural database rules, if the DBMS physical ERD conveys something different than the conceptual ERD, and the SQL does not line up with the DBMS physical ERD. Keeping them in-sync is one key to avoiding many problems.

The typical database creation process is illustrated in the figure below.


Typical Database Creation Process
Typical Progression
Structural Database Rules
Conceptual ERD
DBMS Physical ERD
SQL
New Information Discovered

In this course, you will be creating and maintaining a conceptual ERD and a DBMS physical ERD. You created your conceptual ERD in Iteration 2. In this iteration, you will revise your conceptual ERD as needed, map your conceptual ERD to an initial DBMS physical ERD, add attributes to the ERD, and normalize it. Hang in there, because you are rapidly paving the way for actual implementation in SQL!

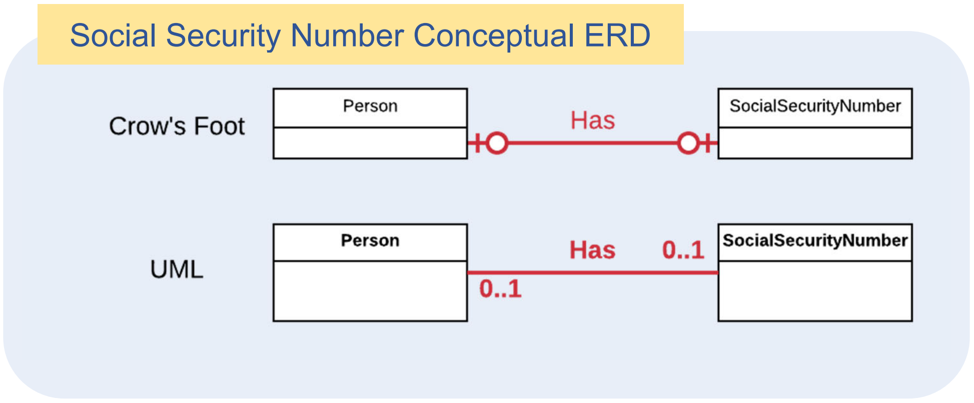
Mapping Conceptual ERDs to DBMS Physical ERDs

Given that we create the conceptual ERD before creating the DBMS physical ERD, the initial process is of mapping one to another rather than creating from scratch. The initial mapping contains the entities, relationships, and primary and foreign keys; the attributes are added later. Creating the initial mapping is actually quite mechanical in nature, and depends entirely upon the relationship classification existing between each two related entities.

Not surprisingly, we map associative relationships, informally known as "has a" relationships, differently than we map specialization-generalization relationships, informally known as "is a" relationships. First, let's look in some detail into the associative mapping as it is more complex.

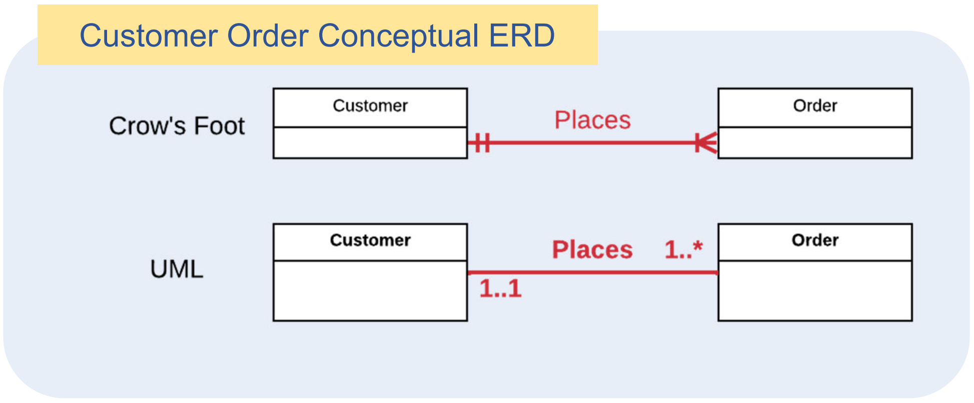
Associative Relationship Classification

A *relationship classification* is identified as the plurality on both sides of the relationship. As you recall, each relationship has a participation constraint and plurality constraint from the perspective of both entities involved. Relationship classifications reference only the plurality of both perspectives, resulting in three relationship classifications – one-to-one, one-to-many, and many-to-many. One-to-one means that the relationship is singular from both perspectives. One-to-many means that the relationship is singular from one perspective and plural from another. Many-to-many means that the relationship is plural from both perspectives. A relationship classification does not indicate direction. That is, if a relationship is one-to-many for example, we do not say that the "leftmost" entity is singular and the "rightmost" entity is plural. We don't know which entity has which plurality, but do know that one is singular and one is plural. Shorthand is often used to identify a particular classification for a relationship, and this shorthand is 1:1 for one-to-one, 1:M for one-to-many, and M:N for many-to-many. Relationship classifications summarize the properties of a relationship by leaving out some details.
Let's look at examples of each relationship classification, starting with 1:1. The structural database rule "Each person may have a social security number; each social security number may be assigned to a person", can be diagrammed conceptually as in the following image.



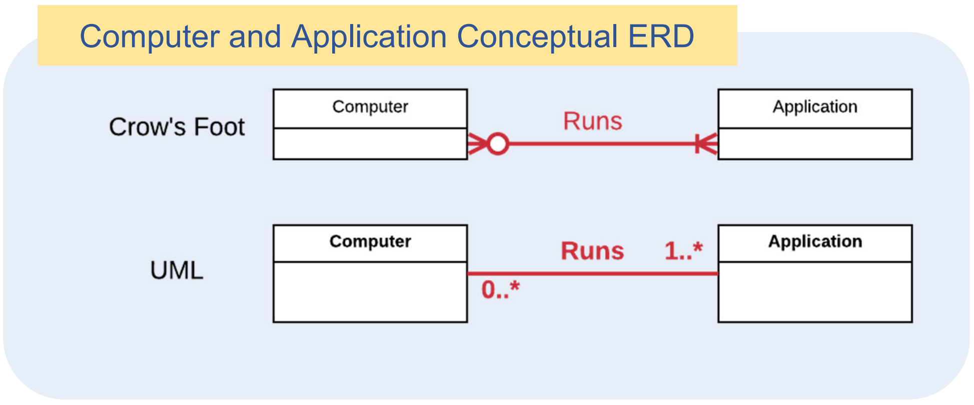
Notice that the relationship is optional and singular to both Person and SocialSecurityNumber. A person may or may not have a social security number (depending upon whether they are U.S. citizens), but they can have at most one. Likewise, any particular social security number may or may not be assigned to a person, but if it is assigned, it is assigned to only one person. This kind of relationship has a classification of 1:1 (one-to-one) since the relationship is singular from both perspectives.

Next, let's look at a 1:M relationship. The structural database rule "An order is placed by a customer; each customer places one or more orders" is illustrated in the following diagram.



Notice that from the perspective of the Customer, each customer may place many orders, but from the perspective of the order, it is always placed by one customer. The plurality from both perspectives makes the relationship classified as a 1:M (one-to-many).

Now we can look at a M:N relationship with the structural database rule “A computer runs many applications; each application may run on many computers”. The relationship is illustrated in the following diagram.



From the perspective of Computer, the relationship is mandatory and plural. From the perspective of Application, the relationship is optional and plural. The relationship, being plural from both perspectives, is thus M:N (many-to-many).

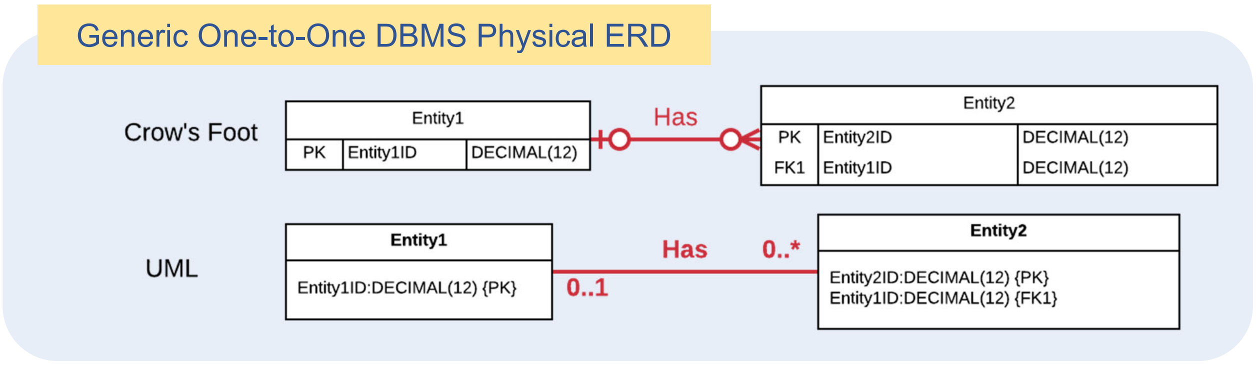
From these three examples, you see the relationship classification includes plurality, but excludes the participation constraint.

Mapping Associative Relationships to DBMS Physical ERDs

Now let’s look at mapping conceptual relationships to DBMS physical ERDs in more detail. This mapping is mechanical, once you understand the concepts. Since foreign keys are used to enforce relationships in the relational model, the process of mapping relationships mostly deals with which foreign keys to create and where to place them. Recall that one table references another through a foreign key in the relational model. *The relationship classification of a relationship determines the number and placement of the foreign key(s). The same classification always maps to the same number and placement of foreign keys.* The entity and relationship names may differ for every relationship, but there are only three ways of placing foreign keys. This is why this mapping is mostly mechanical.

Mapping One-to-One Relationships

When two entities are related with a 1:1 relationship, the DBMS physical ERD will have a foreign key in either one of the entities to enforce the relationship. There is no right or wrong entity, so you pick one that makes the most sense for your design. Generically, the mapping of any 1:1 relationship is as illustrated in the following diagram.



Generic One-to-One DBMS Physical ERD
Crow's Foot
Entity1
PK Entity1ID DECIMAL(12)
Has
Entity2
PK Entity2ID DECIMAL(12)
FK1 Entity1ID DECIMAL(12)
UML
Entity1
Entity1ID:DECIMAL(12) {PK}
Has
0..1
0..*
Entity2
Entity2ID:DECIMAL(12) {PK}
Entity1ID:DECIMAL(12) {FK1}

There are several choices I made in the diagram above worth explaining here. For illustrative purposes, I've used generic names "Entity1" and "Entity2" to represent the entities, and the generic word "Has" for the relationship. I am using the best practice of creating a synthetic primary key for each entity that has the name of the entity followed by "ID". Entity1 has a primary key named Entity1ID, and Entity2 has Entity2ID. I arbitrarily selected the participation constraints of optional from both perspectives, though obviously, not all 1:1 relationships have optional participation constraints.

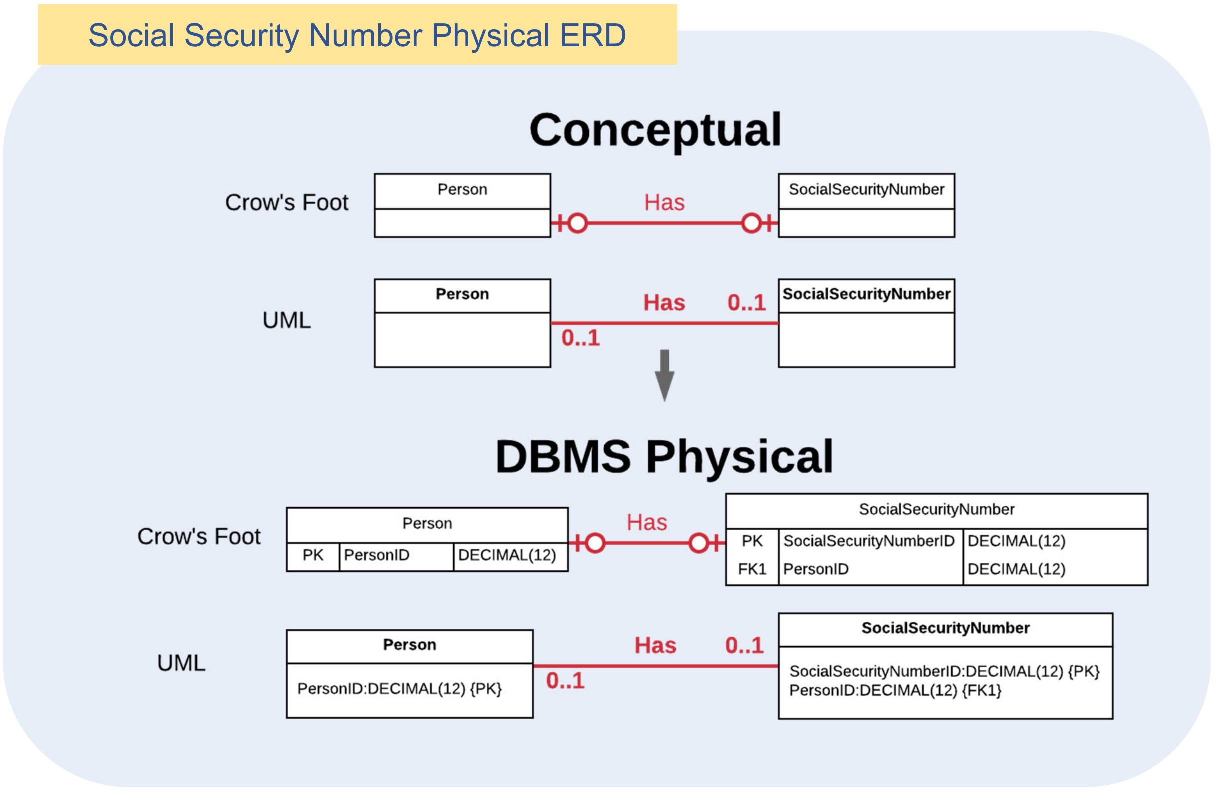
You undoubtedly noticed that the initial DBMS physical ERD contains primary and foreign attributes and their datatypes; these deserve further explanation, especially because attributes have not been present in the initial conceptual ERDs you have worked with thus far. Each attribute has a name which is how it is identified, and a datatype which indicates which kinds of values can be stored in the attribute. In a DBMS physical ERD, we use the precise datatypes that will be used in the SQL for that database. I've chosen to use DECIMAL because that is supported by virtually any modern relational database. I've chosen DECIMAL*(12)* because it supports integers up to 12 digits, and that will support a large amount of rows when the database is implemented. The precise datatypes and limits you use may vary depending upon your organization's best practices. Nevertheless, you want to choose a datatype that supports the kind of data it stores, and limits that are not excessively high, but are not too low. DECIMAL(12) is a reasonable choice for tables that may contain millions of rows, or more, over time.

Of particular importance to the initial mapping process is the placement of the foreign keys. Keys are not present in a conceptual ERD, but are present in the DBMS physical ERD. This is one of the most significant differences between the two types of ERDs. You will also notice that Entity2 references Entity1 with the foreign key Entity1ID. As mentioned previously, with a 1:1 relationship, either entity could contain the foreign key. I selected Entity2, but could have also selected Entity1. A theoretically pure implementation would have both entities contain foreign keys referencing each other; however, such cyclical references are impractical. If Entity1 references Entity2, and Entity2 references Entity1, then it's very difficult to insert rows into either table once they are implemented in SQL. Instead, we use the best practice of using one foreign key in either one of the entities to reference the other entity for 1:1 relationships.

How does that the participation constraint affect the design? The participation constraint affects whether or not certain foreign keys are nullable. In the example above, the relationship is optional from both perspectives. Since Entity2 has a foreign key to Entity1, and the relationship is optional to Entity2, the foreign key is made nullable. If the relationship was mandatory to Entity2, we would require that the foreign key have a value through a NOT NULL constraint. The participation constraint does not affect foreign key placement, only the nullability.

How do the entity and relationship names affect the design? The names only affect the primary and foreign key names. If an entity is named "Pizza", then it's primary key would be named PizzaID following the best practice of creating synthetic keys. Any entity that references that Pizza entity would use a foreign key named PizzaID. The names do not otherwise affect the structure of the design. The names of the entities could be "Person" or "Computer" or "Customer" or anything else, the relationship name could be something other than "Has", and it would not change where the foreign keys are placed or most other parts of the design.

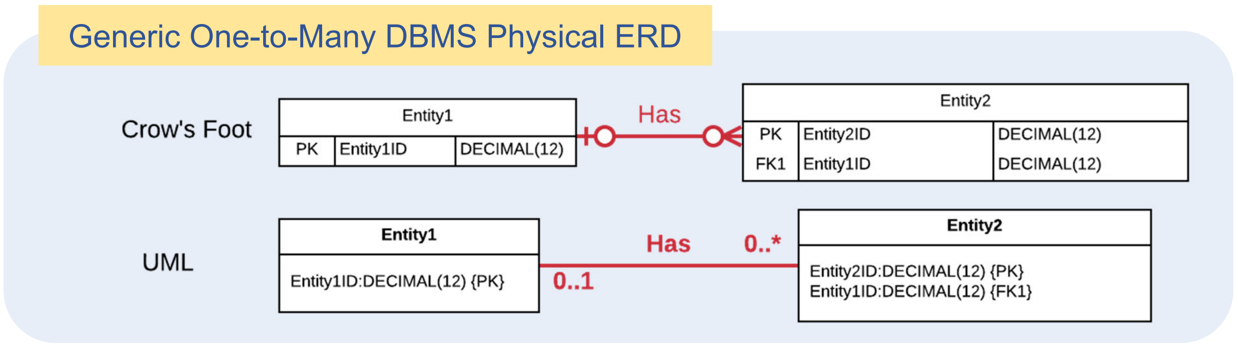
Now that you've seen the abstract methodology for mapping 1:1 relationships into a DBMS physical ERD, let's take a look at a concrete example. We'll map same social security number conceptual ERD created earlier, and this mapping is illustrated in the following figure.



Let's review how this mapping works. Person has a primary key named "PersonID", and SocialSecurityNumber has a primary key named "SocialSecurityNumberID". More important, SocialSecurityNumber has a foreign key to Person. We already identified the relationship, "Each person may have a social security number; each social security number may be assigned to a person" as 1:1; therefore, we follow the 1:1 mapping methodology of placing the foreign key in either entity. I selected the SocialSecurityNumber entity, which is why it contains the PersonID foreign key. Although not visible in the diagram, we would leave PersonID nullable since the relationship is optional (any particular social security number may or may not be assigned to a person). That's all there is too it for 1:1 mappings. We've initially mapped this relationship!

Mapping One-to-Many Relationships

When two entities are related with a 1:M relationship, the DBMS physical ERD will have a foreign key in the entity that is related at most one of the other entity. That is, the entity that is singular has the foreign key. This is illustrated abstractly below.

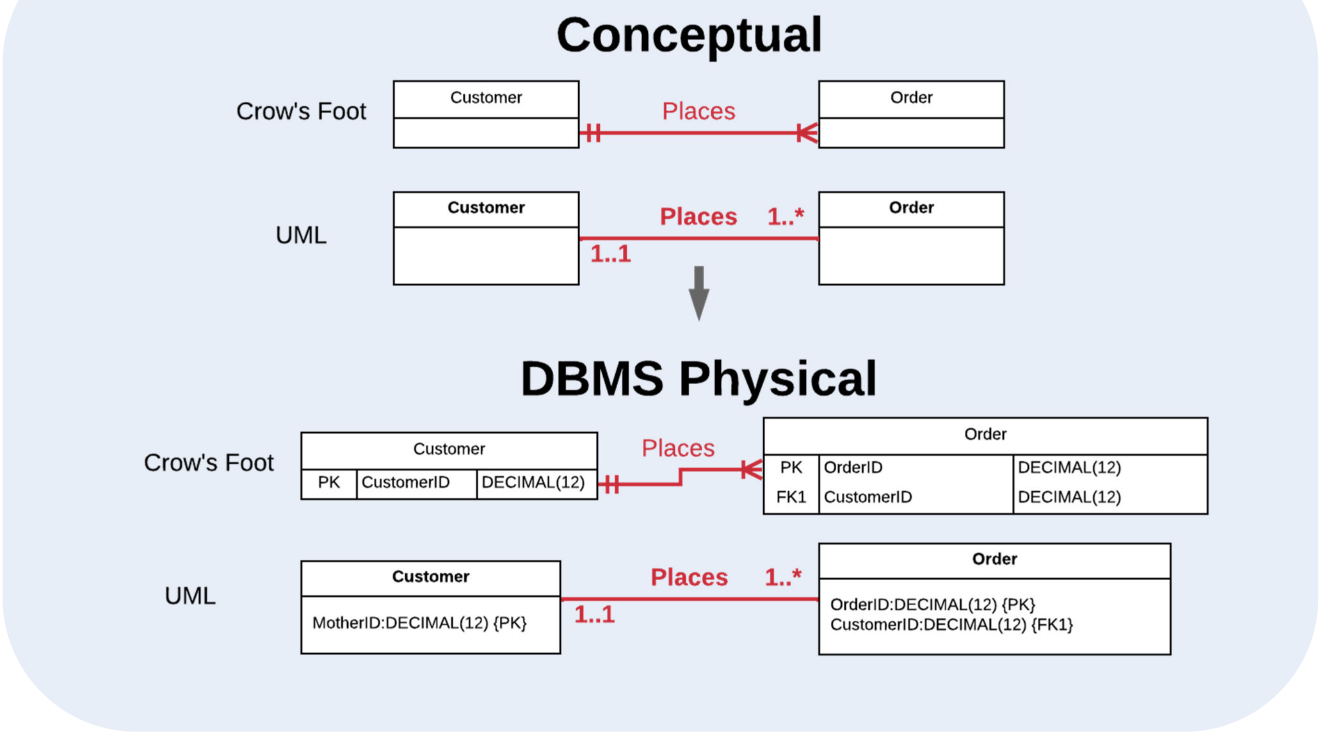

Generic One-to-Many DBMS Physical ERD
Crow's Foot
Entity1
PK Entity1ID DECIMAL(12)
Has
Entity2
PK Entity2ID DECIMAL(12)
FK1 Entity1ID DECIMAL(12)
UML
Entity1
Entity1ID:DECIMAL(12) {PK}
Has
0..1
0..*
Entity2
Entity2ID:DECIMAL(12) {PK}
Entity1ID:DECIMAL(12) {FK1}

There are several important things worth explaining in the example. Similar to the 1:1 example, I arbitrarily chose participation constraints of optional, and followed the best practice of using synthetic primary keys. Entity1 can be related to many of Entity2, making Entity1 plural, and Entity2 is related to at most one of Entity1, making Entity2 singular. The foreign key has been placed in Entity2, the singular entity. Unlike the 1:1 relationship, we cannot place the foreign key in either entity; we must place the foreign key in the singular entity for 1:M relationships.

Why does it matter where we place the foreign key? The answer lies in one seemingly innocuous property of the relational model: every field contains at most one value. When we apply that property to foreign keys, then every foreign key contains at most one value. A foreign key cannot contain references to many instances; it contains a reference to at most one instance. This is why we must place the foreign key in the singular entity. In our 1:M example, Entity2 can reference Entity1 with a foreign key, but Entity1 cannot reference Entity2, since such a key would require a foreign key to contain multiple values. *The fact that a foreign key can only reference one value is the driving principle behind how we map relationships to DBMS physical ERDs.*

Let's look at a concrete example of mapping a 1:M relationship. We'll map the customer order conceptual ERD we created previously, and this mapping is illustrated in the following figure.

Customer Order DBMS Physical ERD

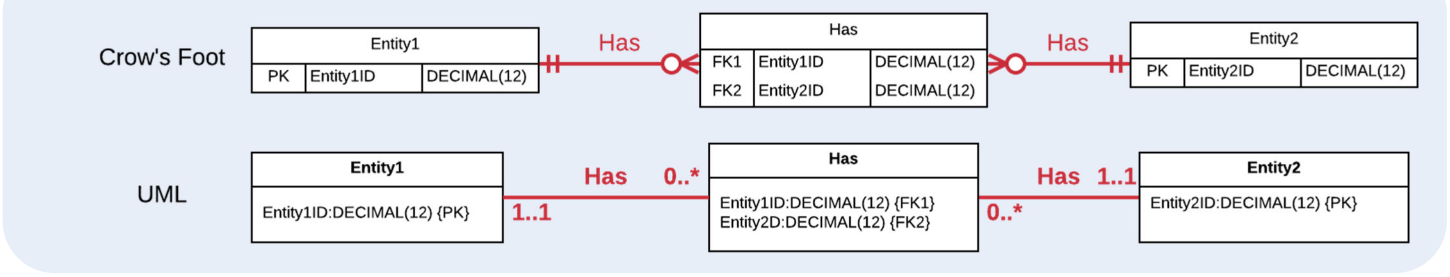


In this example, a customer can place many orders, but each order is placed by one customer. For this reason, we place the foreign key in Order (recall that the singular entity gets the foreign key). Order has the CustomerID foreign key. And of course, CustomerID is the primary key of Customer, and OrderID is the primary key of Order.

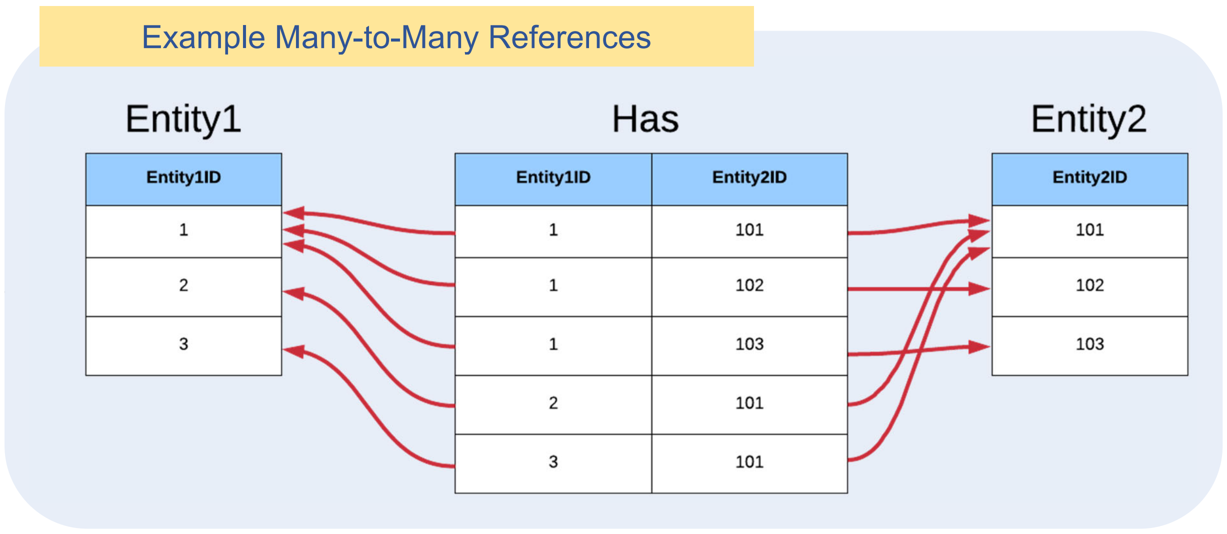
Mapping Many-to-Many Relationships

When two entities are related with a M:N relationship, the relationship itself must be reified into an entity. The new entity is termed a bridging or linking entity, because it exists just to bridge or link the other two entities together. Wait! What? Why do we need to create a new entity? The reason goes back to the driving principle that every foreign key can contain at most one value. If each instance of Entity1 can reference multiple instances, and each instance of Entity2 can reference many instances, then we cannot place the foreign key in either Entity1 or Entity2. Foreign keys cannot reference multiple instances. An abstract many-to-many relationship is illustrated below.

Generic Many-to-Many DBMS Physical ERD


Crow's Foot
Entity1
PK Entity1ID DECIMAL(12)
Has
Has
FK1 Entity1ID DECIMAL(12)
FK2 Entity2ID DECIMAL(12)
Has
Entity2
PK Entity2ID DECIMAL(12)
UML
Entity1
Entity1ID:DECIMAL(12) {PK}
Has
0..*
1..1
Has
Entity1ID:DECIMAL(12) {FK1}
Entity2D:DECIMAL(12) {FK2}
Has
1..1
0..*
Entity2
Entity2ID:DECIMAL(12) {PK}

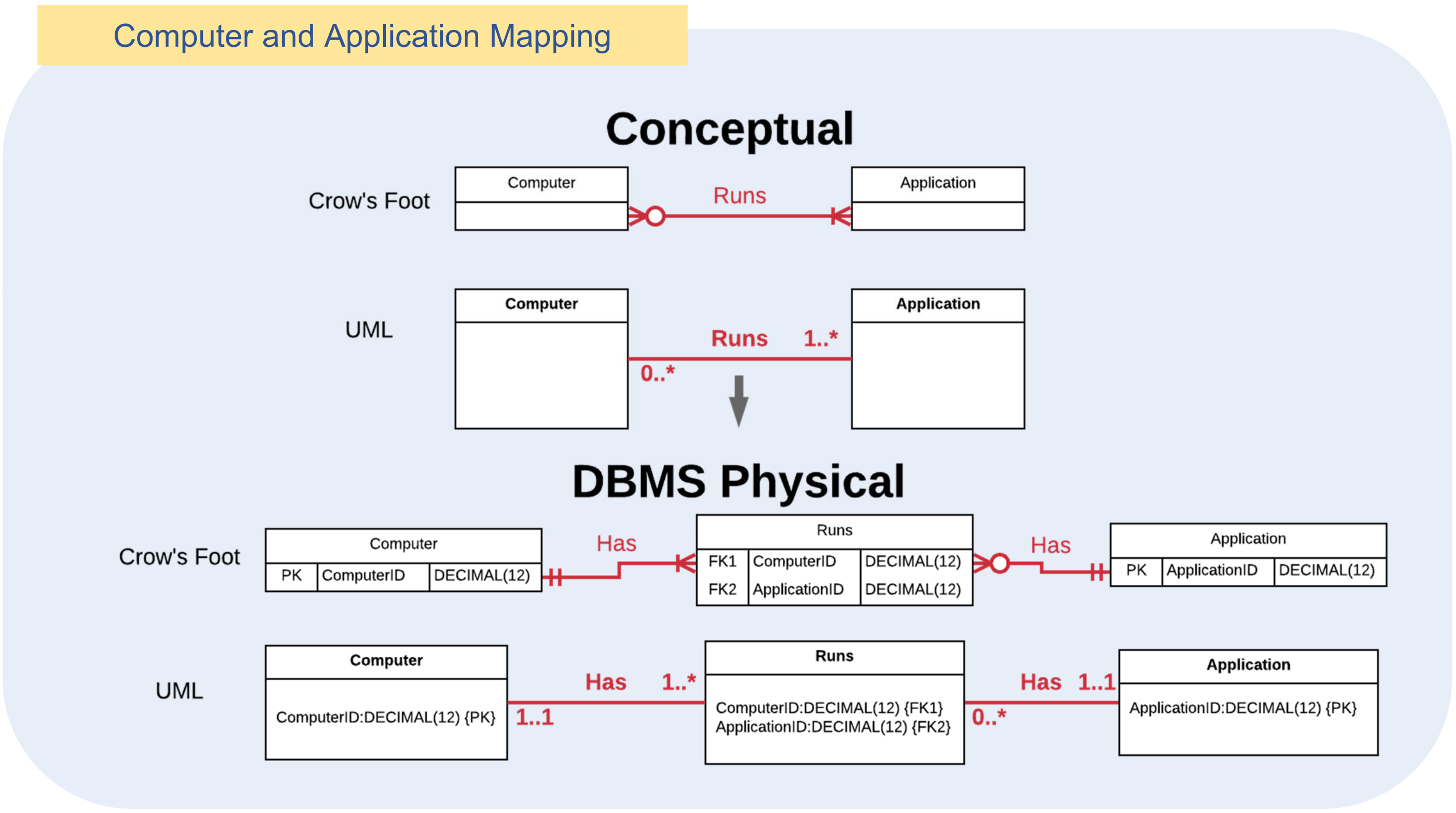
The most obvious difference between this M:N relationship diagram and the others is that there are three entities instead of two. The many-to-many “Has” relationship is reified into a “Has” entity, which has two foreign keys, one to reference each of the other entities. The Has entity is referred to as the bridging or linking entity. I could have named the bridging entity something different, for example, Entity1Entity2Link, but opted for “Has” here because that is the name of the relationship.
Since each foreign key can contain only one value, the Has entity is necessary to support the M:N relationship in the relational model. For example, imagine there is an instance 1 of Entity1 that is related to instances 101, 102, and 103 in Entity2, and that instance 101 in Entity2 is related to instances 1, 2, and 3 in Entity1. The foreign key references in the Has entity would then look like the following figure.



Each pair of references in the Has entity is the “glue” that relates the instances of Entity1 and Entity2 together. Understanding the way the reference pairs work in the bridging entity is essential for understanding why we reify the relationship into an entity. Review the previous example until you are sure you understand it.

Another important property of the M:N DBMS physical diagram is that the new bridging entity is related to the other two entities through two 1:M relationships. To state this another way, the M:N relationship at the conceptual level becomes an entity which is related to the other entities through 1:M relationships. Going back to the driving principle that every foreign key can contain at most one value, we now see that the relational model does not support the M:N relationship directly; the most the relational model supports is a 1:M relationship. This is why we must break a M:N relationship into two 1:M relationships.

For additional understanding, let’s take a look at a concrete example of mapping a M:N relationship, by mapping the previously created Computer and Application conceptual ERD. Recall that the structural database rule for the example is “A computer runs many applications; each application may run on many computers”. The mapping is diagrammed below.

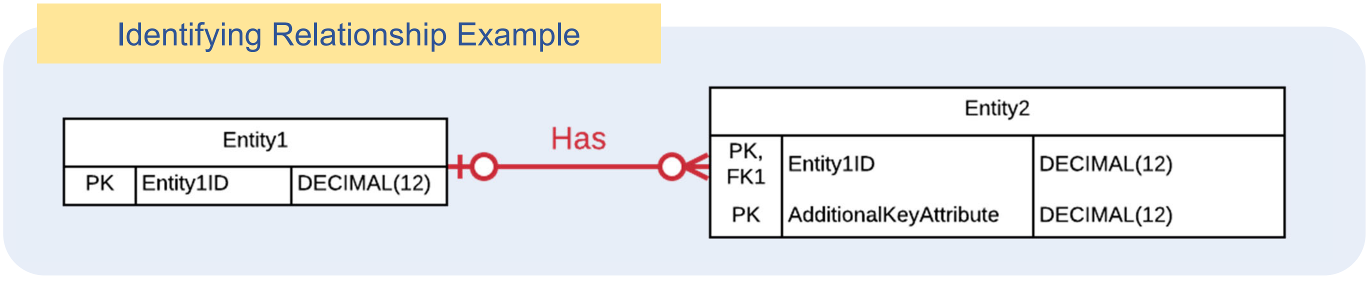


In the conceptual ERD, Computer and Application are related with a M:N relationship. In the DBMS physical ERD, the Runs relationship is reified into an entity which contains foreign keys to both Computer and Application. Runs is related to both Computer and Application with a 1:M relationship for each. As far as the participation constraint, conceptually an application may or may not be run on any computer (optional), but a computer must run at least one application (mandatory). This is reflected in the DBMS physical ERD in that Application has an optional relationship with Runs, but Computer has a mandatory relationship with Runs.

Relationship Strength

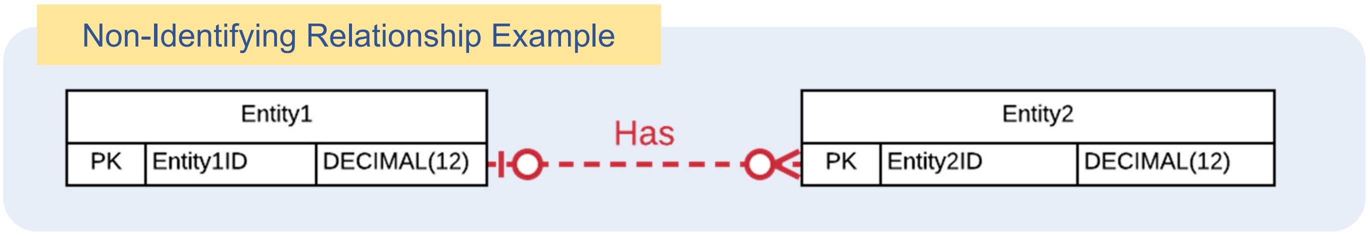
Although not something you are expected to explicitly model, relationship strength is a concept familiar to most anyone who understands or creates ERDs, so deserves a mention. Associative relationships are either identifying, also known as “strong”, or non-identifying, also known as “weak”. An identifying relationship is one where the entity’s primary key is composed of a foreign key to another entity, either in part or in full. That kind of relationship is identifying or “strong” because it is through that relationship that entity derives its identity (primary key). A non-identifying or “weak” relationship is any other kind of relationship, that is, any relationship that does not involve an entity’s primary key referencing another entity through a foreign key. Such a relationship is termed “non-identifying” because the entity’s identity (primary key) is not defined by the relationship.

In an ERD, identifying relationships can be designated with a solid relationship line, while non-identifying relationship can be identified with a dashed line. The use of dashed lines only applies when traditional notations such as Crow’s Foot are used; UML does not support altering the solidity of a relationship line based upon its strength. Below is an example of an identifying relationship with Crow’s Foot notation.



Notice that Entity2 has a composite primary key. One of the primary key's attributes is a foreign key to another entity, Entity1, and the second is some additional attribute that is not a foreign key. Since the primary key of Entity2 is composed of a foreign key to Entity1, the "Has" relationship between those two entities is an identifying (strong) relationship.

Below is an example of a non-identifying relationship with Crow's Foot notation.



Notice that in this example, Entity2's primary key stands alone. It is not composed of a foreign key to Entity1. Therefore, the relationship between Entity1 and Entity2 is considered non-identifying or weak, and the relationship line can be dashed to indicate that fact.

Although prevalent in textbooks*, the use of solid and dashed lines is relegated to academic exercises rather than being an important modeling technique* for many reasons. Following the best practice of using single-value synthetic keys for all tables means all relationships are non-identifying. No entity's primary key will be composed of a foreign key to another entity with such a practice. Further, mature designers, and even mature design tools, do not model weak and strong relationships with the dashed and solid notations. Virtually every line would be dashed making lessening its significance, and it's not considered to be a concept critical enough to warrant the effort. UML class diagrams do not support altered line solidity based upon relationship strength even though the object-oriented model supports many more kinds of relationships than the relational model. For these reasons, *you are not expected or required to use dashed lines for weak relationships for your project.* It is, however, beneficial to understand what it means if you view an ERD that does follow such conventions.

Summary of Mapping Associative Relationships

Whew! This is a lot to take in. There are many details involved with mapping associative relationships to conceptual ERDs, so let's review the main points before proceeding further.

✓ A DBMS physical ERD depends upon the relational model and is tied to a specific database vendor and version.

- ✓ DBMS physical ERDs contains SQL-based constraints, synthetic keys, and database-specific datatypes.

- ✓ DBMS physical ERDs are created by following mechanical steps which map the conceptual ERD to a DBMS physical ERD. The conceptual ERD is created first.

- ✓ A relationship classification summarizes the relationship by identifying the plurality from both perspectives.

- ✓ The relationship classification in the conceptual ERD determines the number and placement of the foreign key(s), and the number tables, in the DBMS physical ERD.

- ✓ When two entities are related with a 1:1 relationship, the DBMS physical ERD retains the related two entities, and a foreign key may be placed in either one of the entities.

- ✓ When two entities are related with a 1:M relationship, the DBMS physical ERD retains the related two entities, and a foreign key is placed in the entity that is related to at most one (that is, the singular entity).

- ✓ When two entities are related with a M:N relationship, the DBMS physical ERD retains the related two entities, and adds a third entity by reifying the relationship into an entity. The new entity contains foreign keys tol both of the other entities.

- ✓ When two entities are related with a M:N relationship, the DBMS physical ERD retains the related two entities, and adds a third entity by reifying the relationship into an entity. The new entity contains foreign keys to both of the other entities.

Mapping Your Associative Relationships

You have now learned enough to map your associative relationships to a DBMS physical ERD. The first step is to identify the relationship classifications for all of the associative relationships in your design. You can leave out the specialization-generalization relationships for now because we will deal with those in a separate task. Note that you can identify relationship classification either from your structural database rules, or from your conceptual ERD, since both identify the entities and relationships.

Once you've identified the relationship classifications, go ahead and create a DBMS physical ERD which contains your associative relationships.

Here is how I identify the relationship classifications and map the TrackMyBuys associative relationships to a DBMS physical ERD.

TrackMyBuys Relationship Classification and Associative Mapping

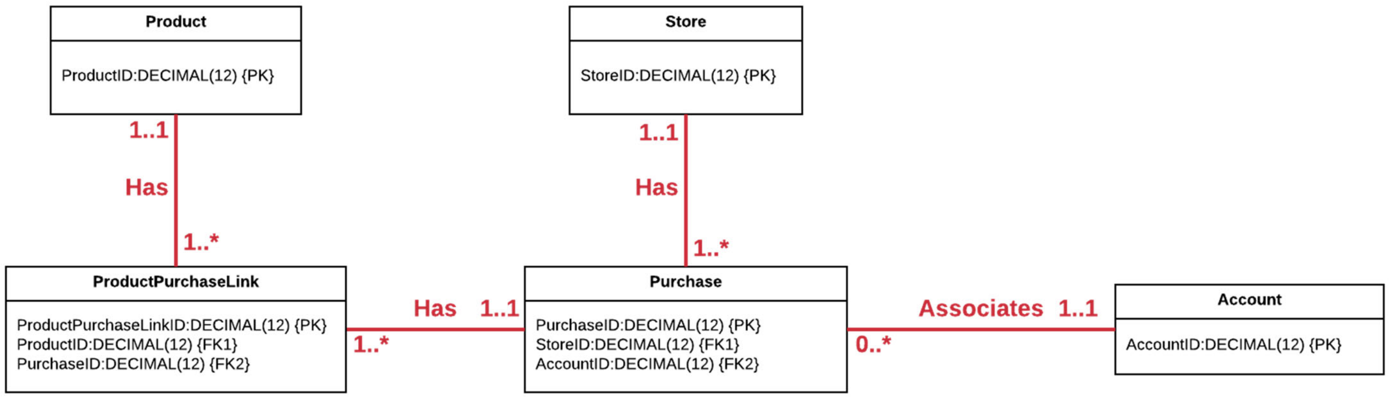
I opted to use the conceptual ERD to identify the relationships since I am a visual person. The associative relationships in my conceptual ERD are Store/Purchase, Product/Purchase, and Account/Purchase.

The Store/Purchase relationship is 1:M. Many purchases can happen in a store, but each purchase happens in exactly one store.

The Account/Purchase relationship is also 1:M. One account can be associated to many purchases, but each purchase is tied to one account.

The Product/Purchase relationships is M:N. Each product can be purchase many times, and each purchase can include many products.

The associative relationships in my conceptual ERD are Store/Purchase, Product/Purchase, and Account/Purchase, where Store/Purchase and Account/Purchase are 1:M relationships, and Product/Purchase is a M:N relationship. Here is the DBMS physical ERD I created from the conceptual ERD for these relationships.


Product
ProductID:DECIMAL(12) {PK}
Store
StoreID:DECIMAL(12) {PK}
1..1
Has
1..*
1..1
Has
1..*
ProductPurchaseLink
ProductPurchaseLinkID:DECIMAL(12) {PK}
ProductID:DECIMAL(12) {FK1}
PurchaseID:DECIMAL(12) {FK2}
Has
1..1
1..*
Purchase
PurchaseID:DECIMAL(12) {PK}
StoreID:DECIMAL(12) {FK1}
AccountID:DECIMAL(12) {FK2}
Associates
1..1
0..*
Account
AccountID:DECIMAL(12) {PK}

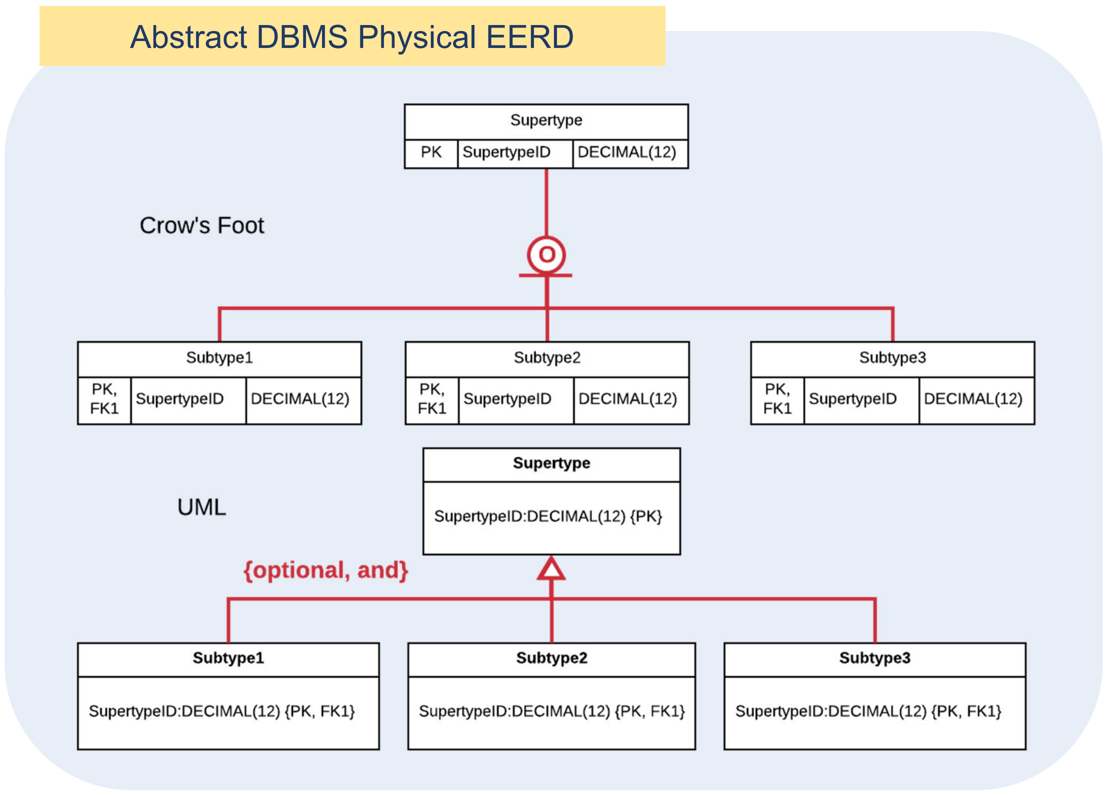
I followed the best practice of creating synthetic keys for all tables, and opted to make the primary keys of the DECIMAL(12) datatype to allow for a lot of records in my database. Since Store/Purchase and Account/Purchase are 1:M relationships, I retained the entities from the conceptual ERD, and placed foreign keys in Purchase since Purchase is the entity that has at most one store and at most one account.

Since Product/Purchase is an M:N relationship, it was necessary for me to create a bridging entity to support the relationship. I named that entity ProductPurchaseLink to give it a useful name in the database, because the original relationship name in the conceptual ERD is "Has", which isn't a particularly useful name. If I had some other useful relationship name in the conceptual ERD, I might have kept that name. The bridging entity has foreign keys to both Product and Purchase, resulting in two 1:M relationships between ProductPurchaseLink and Product and Purchase.

Mapping Specialization-Generalization Relationships

Specialization-generalization relationships are not surprisingly mapped differently than associative relationships. Thankfully, the mechanics of doing so are much simpler than its counterpart, because there is no need to map differently based upon relationship classification, and because the constraints on the relationship do not affect the design. There is only rule for mapping specialization-generalization relationships: the supertype contains a primary key, and all subtypes have the same primary key as the supertype through use of a foreign key constraint. That's it!

Let’s take a look at an abstract example first, diagrammed below.

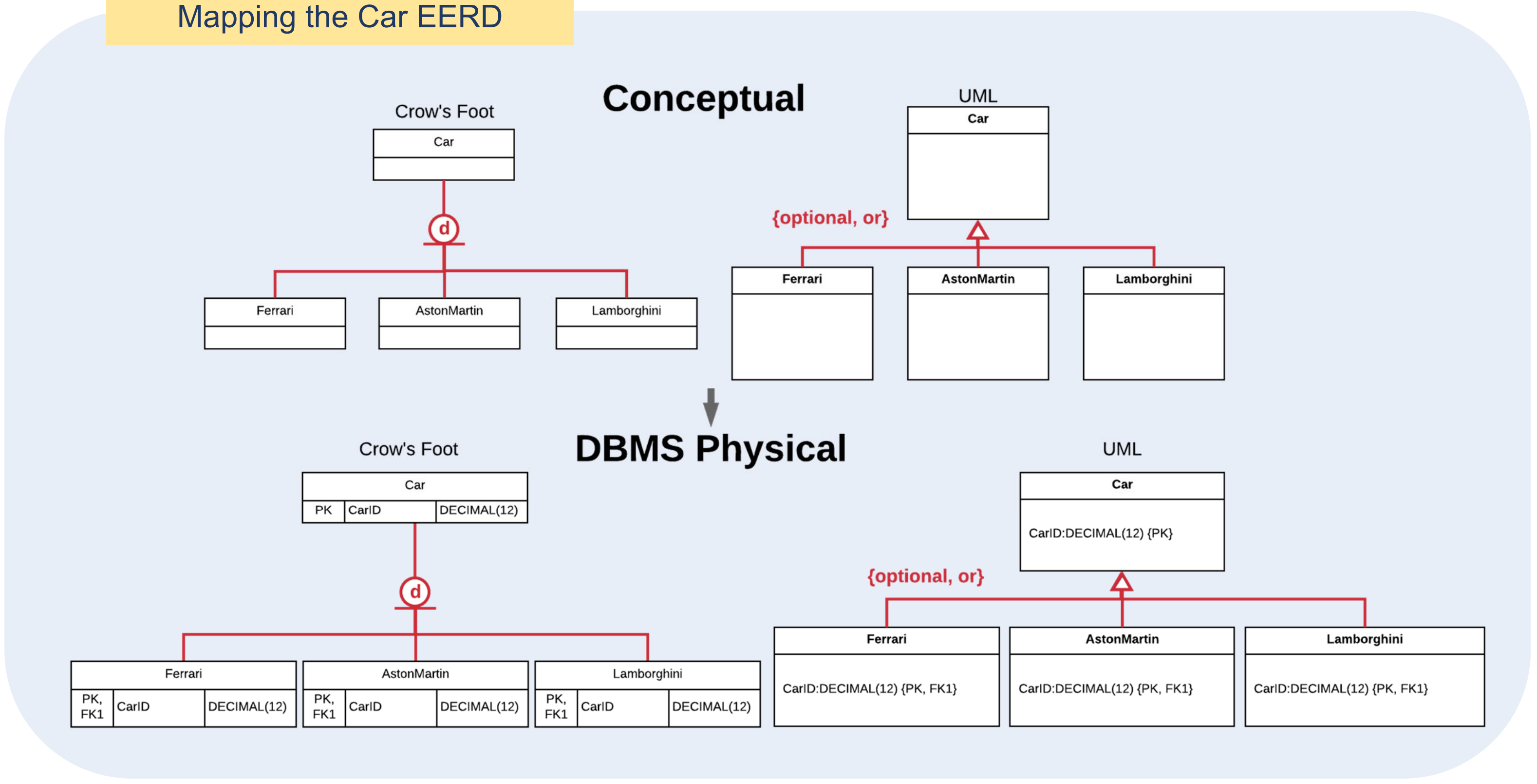

Abstract DBMS Physical EERD
Crow's Foot
Supertype
PK | SupertypeID | DECIMAL(12)
O
Subtype1
PK, FK1 | SupertypeID | DECIMAL(12)
Subtype2
PK, FK1 | SupertypeID | DECIMAL(12)
Subtype3
PK, FK1 | SupertypeID | DECIMAL(12)
UML
Supertype
SupertypeID:DECIMAL(12) {PK}
{optional, and}
Subtype1
SupertypeID:DECIMAL(12) {PK, FK1}
Subtype2
SupertypeID:DECIMAL(12) {PK, FK1}
Subtype3
SupertypeID:DECIMAL(12) {PK, FK1}

Most important, take a look at the primary keys of the entities in this example. The supertype has a primary key named “SupertypeID”, as does all of the subtypes. You will also notice that the primary keys of the subtypes are also foreign keys. Why is that? Because, the primary key of each subtype is a foreign key to the primary key of the supertype. Simply put, the identity of each subtype is the identity of the supertype because the specialization-generalization relationship is known as the “is a” relationship. If there is a Subtype instance with an associated Supertype instance, their identifies are one and the same. This is why the primary keys of the subtypes are foreign keys to the supertype.

There are a few other items of note in the example. We use the other best practices of using synthetic primary keys, and using a datatype that provides for enough records (in this case, DECIMAL(12)). I arbitrarily make the relationship partially complete and overlapping for this abstract example. However, *the completeness and disjointness constraints do not affect the mapping of specialization-generalization relationships.* The same design results from any combination of constraints on the relationship.

Let’s take a look at a concrete example by mapping the car example from Iteration 3 into a DBMS physical ERD. The structural database rule for the example is “A car is a Ferrari, Aston Martin, Lamborghini, or none of these.” The mapping looks as follows.

Mapping the Car EERD



The Car entity has a CarID primary key, and the subtypes all have a CarID primary key that is also a foreign key back to the Car entity. We do not need to create bridging entities or worry about making foreign keys null or not null, regardless of the disjointness and completeness constraints. The mapping is really that simple!

How are the disjointness and completeness constraints implemented? The simple answer is that both disjointness and completeness do not affect the database structure, and are not implemented as structure. The business logic for the application would maintain both constraints. For example, the application that uses this Car database would not allow a person to designate a car as both a Ferrari and a Lamborghini; the relationship is disjoint since one car cannot be both. And the database structure itself, such as the primary keys and foreign keys and number of entities, does not change whether a relationship is disjoint, overlapping, partially complete, or totally complete.

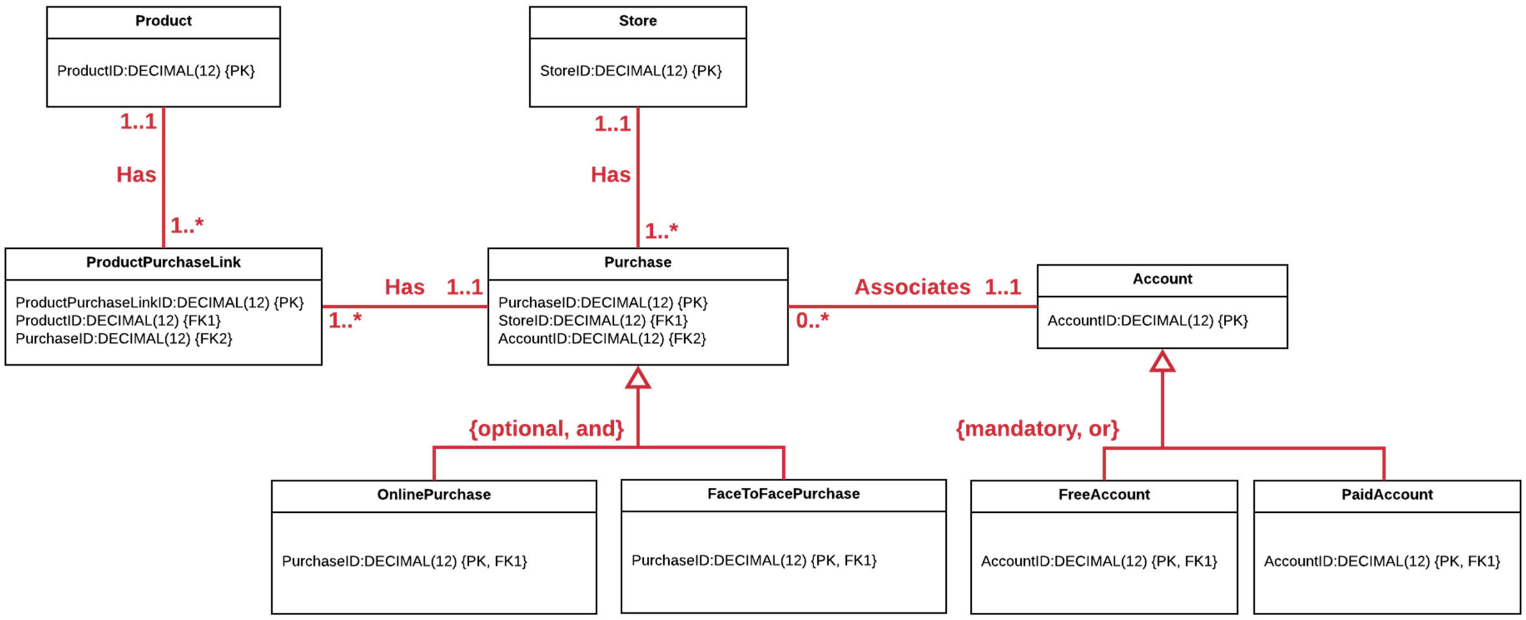
Mapping Your Specialization-Generalization Relationships

With your new understanding, you can now map your specialization-generalization relationships into your DBMS physical ERD. You already mapped your associative relationships, so you can just add in your specialization-generalization relationships into the existing diagram.

Here are my specialization-generalization mappings for TrackMyBuys.

TrackMyBuys Specialization-Generalization Mapping

I have two specialization-generalization relationships in my conceptual ERD, one for the Purchase entity and one for the Account entity. Here is my DBMS physical ERD with these relationships mapped into them.



Product
ProductID:DECIMAL(12) {PK}
Store
StoreID:DECIMAL(12) {PK}
1..1
Has
1..*
1..1
Has
1..*
ProductPurchaseLink
ProductPurchaseLinkID:DECIMAL(12) {PK}
ProductID:DECIMAL(12) {FK1}
PurchaseID:DECIMAL(12) {FK2}
Has 1..1
1..*
Purchase
PurchaseID:DECIMAL(12) {PK}
StoreID:DECIMAL(12) {FK1}
AccountID:DECIMAL(12) {FK2}
Associates 1..1
0..*
Account
AccountID:DECIMAL(12) {PK}
{optional, and}
{mandatory, or}
OnlinePurchase
PurchaseID:DECIMAL(12) {PK, FK1}
FaceToFacePurchase
PurchaseID:DECIMAL(12) {PK, FK1}
FreeAccount
AccountID:DECIMAL(12) {PK, FK1}
PaidAccount
AccountID:DECIMAL(12) {PK, FK1}

The additional entities under Purchase are OnlinePurchase and FaceToFacePurchase, each of which have a primary and foreign key of PurchaseID which reference the primary key of Purchase. The additional entities under Account are FreeAccount and PaidAccount, which have primary and foreign keys of AccountID which reference the primary key of Account. With these additional mappings, this DBMS physical now has all of the relationships in the conceptual ERD.

Summary and Reflection

You have made great progress this week. You now have an initial normalized DBMS physical ERD that has everything you need to start creating and using your database in SQL. Your database design is taking shape, and in just the next iteration you will be able to start implementing your structure in SQL. Update your project summary to reflect your new work.

Write down your questions, concerns, or observations, so that you and your facilitator or instructor are aware of them. I'm sure there is at least one thing worth writing down here about your progress, if not more!

Here is an updated summary as well as some observations I have about my progress on TrackMyBuys for this iteration.

TrackMyBuys Summary and Reflection

My database is for a mobile app named TrackMyBuys which records purchases made across all stores, making it the one stop for any purchase history. Typically, when a person purchases something, they can only see their purchase history with that same vendor, and TrackMyBuys seeks to provide a single interface for all purchases. The database must support a person entering, searching, and even analyzing their purchases across all stores.

The structural database rules and conceptual ERD for my database design contain the important entities of Store, Product, Purchase, and Account, as well as relationships between them. The design contains a hierarchy of Purchase/FaceToFacePurchase and Purchase/OnlinePurchase to reflect the two primary ways people purchase products. The design also contains a hierarchy of Account/PaidAccount and Account/FreeAccount to reflect the fact that people can signup for a free account or a paid account for TrackMyBuys. The initial DBMS physical ERD contains the same entities and relationships, and uses the best practice of synthetic keys.

I find it exciting that my database is taking shape with an initial DBMS Physical ERD. I can see how this will be created and implemented in SQL!

Items to Submit

In summary, for this iteration, you update your design document by revising sections from prior iterations. You add in specialization-generalization, then create an initial DBMS physical ERD that contains primary and foreign keys. Make sure to use the template provided with this iteration to ensure you are submitting all necessary items.

Evaluation

Your iteration will be reviewed by your facilitator or instructor with the criteria outlined in the table below. Note that the grading process:

- involves the grader assigning an appropriate letter grade to each criterion.
- uses the following letter-to-number grade mapping – A+=100,A=96,A-=92,B+=88,B=85,B-=82,C+=88,C=85,C-=82,D=67,F=0.
- provides an overall grade for the submission based upon the grade and weight assigned to each criterion.
- allows the grader to apply additional deductions or adjustments as appropriate for the submission.
- applies equally to every student in the course.

Criterion	**What is Measures**	**A**	**B**	**C**	**D**	**F**
Structure of Specialization-Generalization Rule(s) (5%)	This measures how well-formed and unambiguous the specialization-generalization structural database rules are. An excellent rule clearly defines the entities, relationships, and constraints, and identifies a supertype and at least two subtypes.	Entirely well-formed Entirely unambiguous	Mostly well-formed Mostly unambiguous	Partially well-formed Partially unambiguous	Insufficient form Mostly ambiguous	Rules missing *or* entirely malformed and ambiguous
Specialization-Generalization Sufficiency (10%)	This measures how essential the specialization-generalization entities and relationships are to your database. Being essential means that the supertype and subtype entities and the relationship support very important aspects of the database.	Entirely essential	Mostly essential	Secondary but useful	Mostly unnecessary	Specialization-generalization missing or entirely unnecessary
Accuracy of Specialization-Generalization Relationship Constraints (10%)	This measures how accurate the specialization-generalization completeness and disjointess constraints are given the needs of the database.	Entirely accurate	Mostly accurate	Somewhat accurate	Mostly inacdurate	Constraints missing or entirely inaccurate

Specialization-Generalization ERD Representation (10%)	This measures how well the supertype entities, subtype entities, and relationships reflect what is defined in the specialization-generalization structural database rules, and the correctness of the diagrammatic notation.	Entirely reflects rules Entirely correct diagrammatic notation	Mostly reflects rules Mostly correct diagrammatic notation	Somewhat reflects rules Somewhat correct diagrammatic notation	Marginally reflects rules Mostly incorrect diagrammatic notation	Specialization-generalization unrepresented *or* Does not reflect rules Incorrect diagrammatic notation
DBMS Physical Mapping Accuracy (40%)	This measures how correctly the DBMS physical ERD maps all entities, relationships, and relationship constraints. An excellent mapping is enforced properly with primary and foreign keys.	Entirely correct	Mostly correct	Somewhat correct	Mostly incorrect	DBMS physical ERD missing *or* entirely incorrect
Overall Presentation (10%)	This measures how well the choices for the structural database rules, entities, relationships, constraints, the ERD, and other design aspects are supported with explanations, in addition to the documentation organization and presentation.	Excellent support Well organized and presented	Good support Organized and presentable	Partial support Somewhat organized and presented	Mostly unsupported Mostly disorganized presentation	No explanations Entirely disorganized presentation
Prior Work Soundness (15%)	This measures how well any issues from prior iterations have been improved in order to provide a frame of reference for this iteration.	Completely improved *or* No improvement necessary	Mostly improved	Somewhat improved	Mostly not improved	No improvements
Preliminary Grade:	100.0	**Entities Deduction:** At least 10 required at the conceptual level At least two subtypes required 3 point deduction for each missing	0	**Lateness Deduction:** 5 points per day 4 days maximum Contact your facilitator for any exceptions	0	**Iteration Grade:**

Use the **Ask the Teaching Team Forum** if you have any questions regarding how to approach this iteration. Make sure to include your name in the filename and submit it in the *Assignments* section of the course.