Relations

Matt Warner

Contents

Chapter 1

Normalization

1 Anomolies

If our database is a single relation with schema ***SP*** (SuppName, SuppAddr, Item, Price).

With out instance data:

SuppName	SuppAddr	Item	Price
John	10 Main	Apple	$2.00
John	10 Main	Orange	$2.50
Jane	20 State	Grape	$1.25
Jane	20 State	Apple	$2.25
Frank	30 Elm	Apple	$6.00

There are some common things that we might want to do that would cause issues. We refer to these as **anomalies**, and there are split into three categories.

Insertion Anomaly

Let's say we want to add a new vendor, "Sally", and store her address, "40 Pine", but she is not selling anything yet. Can this be inserted into the relation SP?

Sally	40 Pine	???	???

The answer to this question is **NO**. The *primary key* is (SuppName, Item), but we only have SuppName. The *entity integrity constraint* is violated if we try to insert the data as a tuple in this relation. It cannot fit. We call this an *insertion anomaly.*

Deletion Anomaly

This time, let's say that Frank no longer sells Mango. We want to take that out of the database so nobody can order a mango that is not available. Our new tuple would look like this:

Frank	30 Elm	???	???

Can this tuple remain in the relation with the Mango information removed?

No, it cannot. The *primary key* is (SuppName, Item), and the Item is going away. The *entity integrity constraint* is violated if we remove the data from the tuple in this relation. We can either keep the whole tuple, advertising fake mango, or delete the whole tuple and lose the information on Frank, which doesn't exist in any other tuples. We call this a *deletion anomaly.*

Update Anomaly

Next, let's say that John is moving to a different address. We would want to change it once for every item John is selling.

John	**10 Main**	Apple	$2.00
John	**10 Main**	Apple	$2.00

This isn't a big deal with only two items, but as John's list of supplied items grows, so does the amount of database work that needs to be done every time he moves. If any of the SuppAddr values for John don't agree, then it may not be clear which is the right address for John. This is an *update anomaly.*“

In summary, we have:

- Insertion anomalies
  - When a piece of data cannot be inserted because it violates some *constraint* of the relation.
  - Usually is the *entity integrity constraint* being violated, but not always.
- Deletion anomalies
  - When deleting some piece of data, a *deletion anomaly* is when more data is lost than intended.
  - Usually this is caused when the data removed is part of the *primary key*, which would cause a violation of the entity integrity constraint.
- Update anomalies
  - When updating a single value requires changes to multiple tuples, this is an *update anomaly*.
    - This is caused by unnecessary redundancies in the data.
    - These cause inefficiency, and potential inconsistencies.

2 Decomposition

Here, we represent the original data in two relations, rather than the one.

SP(SuppName, Item, Price)

SuppName	Item	Price
John	Apple	$2.00
John	Orange	$2.50
Jane	Grape	$ 1.25
Jane	Apple	$2.25
Frank	Mango	$6.00

SP(SuppName, SuppAddr)

SuppName	SuppAddr
John	10 Main
Jane	20 State
Frank	30 Elm

Now, we can make changes to insert Sally and his address without needing to care whether or not he is selling anything. We can update johns address in one spot instead of multiple, and we can delete Jane from the top table when she is no longer selling anything.

3 Keys

Keys are one of the basic requirements of a relational database model. It is widely used to identify the tuples uniquely in the table. We also use keys to set up relations amongst various columns and tables of a relational database.

Types of Keys

- Super key
- Candidate Key
- Primary key
- Foreign key

Super key

A super key is an attribute or set of attributes whose values can uniquely identify any tuple.

Every relation has at least one - the set of all attributes in the relation (since duplicate tuples are considered to be the same tuple)

There can potentially be many available, some more useful that others.

Candidate Key

This is a minimal super key. It is the minimal set of attributes that can uniquely identify a tuple. For example, `Student_ID` in `Student` relation.

StudentID	RollNo	Name	MobileNo	EmailID
A1	1	Matt	9120	a@gmail.com
A2	2	John	8732	b@gmail.com
A3	3	Luke	8344	c@gmail.com

In this table: `StudentID` $\rightarrow$ `RollNo, Name, MobileNo, EmailID`. Therefore, it is a candidate key.

Primary Key

The **primary key** for a relation is chosen by the database designer from among the relation's candidate keys. It becomes the "official" key that is used to reference tuples within the relation. There can be only one.

Once a primary key is chosen, each of the attributes in the relation will be either **prime** or **non-prime** with respect to the relation.

- A **prime** attribute is one of the attributes that can be found in any of the candidate keys.
- A **non-prime** attribute is one of the attributes *not found* in any of the candidate keys.

Once a primary key is chosen for it, the **schema** of a relation is written with the primary key's attributes underlined:

$$\textbf{Relation\_Name}(\underline{A_1}, A_2, A_3, \ldots, A_n)$$

Foreign Keys

A **foreign key** is a tool used to link relations within a database. Since every relation has a primary key that uniquely identifies each tuple, the values of those key attributes can be used from another relation to reference individual tuples.

The relation whose primary key is being used is the **home relation**.

4 Domain

The **domain** of an *attribute* is the set of all possible values it may hold.
The **domain** of a *set of attributes* is the set of all possible combinations of values for the attributes in the set.

5 Order Independence

In relations, the order things appear doesn't matter. There are ways to force them to sort later when we're working with SQL, but the relation itself has no order for either rows or attributes.

It doesn't matter what order the attributes appear in, if two relational schemas have the same name, the same attributes, and the same primary key, then they are equivalent.
So, all of these are equivalent:

$$R(\underline{A}, B, C, D)$$

$$R(D, C, B, \underline{A})$$

$$R(\underline{A}, D, B, C)$$

Tuples are stored unordered. If you need to have them appear in some order later, you will be able to sort based on the values inside of them using SQL.

6 Constraints

6.1 Entity contegrity constraint

The entity integrity constraint applies to all relations. It states that no tuple may exist within a relation that has null value for any of attributes that make up the primary key.

This is a consequence of the primary key being a candidate key, which is minimal and cannot do its job with less data.

6.2 Referential Integrity Constraint

The referential integrity constraint applies to all foreign keys. It constrains the values of foreign keys in relations to values that actually exist as primary keys for tuples within the home relation.

If the foreign key is otherwise allowed to be NULL, then that is also an acceptable value.

7 Functional Dependencies

A *functional dependency* is a statement about which attributes can be inferred from other attributes. If we take X and Y as *sets* of attributes, we can write:

$$X \rightarrow Y$$

If, whenever unique values for **all** of the attributes in X are known, unique values for **each** of the attributes of Y are guaranteed to be possible to look up or to infer using those values. This is read either as:

$$X \text{ functionally determines } Y, \textit{ or}$$

$$Y \text{ is functionally dependent upon } X$$

Defintion

For a functional dependency to exist between two attributes, x and y, that is:

$$x \rightarrow y$$

Then the following must be true:

$$\begin{array}{ll} \text{if} & t_1.x = t_2.x \\ \text{then} & t_1.y = t_2.y \end{array}$$

They are statements about the operational data. Later on, we will see how to read them off of ER diagrams, though they may come from elsewhere as well.

Real-life Examples

$$ZID \rightarrow \text{StudentFirstName, StudentLastName, Birthday}$$

If I identify a student using their ZID, that student has *one* first name, last name, and birthday.

$$\text{StudentFirstName} \not\rightarrow \text{ZID}$$

The first name is not enough to determine a single ZID, as there are multiple students with the same first name.

$$\text{ZID, CourseID, Semester} \rightarrow \text{Grade}$$

If i know which student, which course, and which semester, I can find a single grade.

Keep in mind that *functional dependencies* are constraints present within the operational data your database models. They don't necessarily describe how things work in the real world, but they do have to accurately describe any data you will store in your database.

Additionally, *functional dependencies* **must** hold for all possible data values. Attempts to add data that does not obey the functional dependencies will result in anomalies.

Furthermore, functional dependencies **can** be enforced during insertion if the database is set up properly.

7.1 Armstrong's Axioms

Armstrong's Axioms are a set of rules for operations that are permissible when manipulating *functional dependencies*.

Primary Rules:

Axiom of reflexivity:

$$\text{If } Y \subseteq X \text{ then } X \rightarrow Y$$

Axiom of augmentation:

$$\text{If } X \rightarrow Y, \text{ then } XZ \rightarrow YZ \text{ for any } Z$$

Axiom of transitivity:

$$\text{If } X \rightarrow Y \text{ and } Y \rightarrow Z, \text{ then } X \rightarrow Z$$

Secondary Rules

Decomposition:

$$\text{If } X \rightarrow YZ \text{ then } X \rightarrow Y \text{ and } X \rightarrow Z$$

Composition:

$$\text{If } X \rightarrow Y \text{ and } A \rightarrow B \text{ then } XA \rightarrow YB$$

***Union* (Notation):**

$$\text{If } X \rightarrow Y \text{ and } Y \rightarrow Z \text{ then } X \rightarrow YZ$$

***Pseudo-transitivity*:**

$$\text{If } X \rightarrow Y \text{ and } YZ \rightarrow W \text{ then } XZ \rightarrow W$$

***Self-determination*:**

$$I \rightarrow I \text{ for any } I$$

Example: Relation with FDs

Lets say we have the following relation:

EmpProj($\underline{\text{EmpID}}$,$\underline{\text{Project}}$, Supv, Dept, Case)

EmpID	Project	Supv	Dept	Case
e1	p1	s1	d1	c1
e2	p2	s2	d2	c2
e1	p3	s1	d1	c3
e3	p3	s1	d1	c3

Our functional dependencies are:

- EmpID, Project → Supv, Dept, Case
- EmpID → Supv, Dept
- Supv → Dept

As written, there are some anomalies present. We will use *normalization* to move toward a better design.

7.2 Revisiting Keys

When we talked about *keys*, we talked about how their purpose is to uniquely identify a tuple within a relation. Another way of stating this, now that we know about *functional dependencies*, is **the attributes of a superkey must functionally determine *all* of the attributes of the relation.**

Candidate keys and *primary keys* **are** *super keys*, so this is true of them as well, and they also satisfy additional requirements. As an example, say we have the relation ***R***($\underline{\text{a}}$,b,c,d,e,f)

$$a \rightarrow a, b, c, d, e, f$$

But, since it is always the case that $a \rightarrow a$ because of the self-determination axiom, we usually omit the left hand side from the righthand side. So we would usually write this instead:

$$a \rightarrow b, c, d, e, f$$

7.3 How to determine the functional dependencies

Lets say we have the following attributes, and want to identify all the functional dependencies:

- shipmentID
- shipmentDate
- origin
- destination
- shipID
- shipName
- CaptainID
- capatinName
- ItemID
- Description

- Weight
- quantity

In terms of functional dependencies we have:
shipmentID $\rightarrow$ shipmentDate, origin, Destination, shipID, ShipName, CaptainID, CaptainName
shipID $\rightarrow$ shipName, captainID, CaptainName
captainID $\rightarrow$ CaptainName
itemID $\rightarrow$ description, weight
itemID, ShipmentID $\rightarrow$ quantity

The first determinate is the shipmentID. If we know what the shipmentID is, then we also know its ShipmentDate, Origin, destination, shipID, ShipName, CaptainID and captainName. Regarding the captain, we are making an assumption that a ship has only one captain. We cannot include itemID, description, weight, or quantity because these all refer to the individual items, and a shipment ID cannot determine an item, because there can be many items.

We can ignore shipmentDate. Usually, you shouldn't think twice about skipping dates as the date alone cant really determine anything. The next attributes are origin and destination, we are also going to skip over these since they cant really determine anything either.

Our next determinate is shipID. This one is pretty obvious... a shipID determines shipName, captainID (again assuming there is one captain per ship), and CaptainName.

Next up is captainID, if we know the captainsID, we know the captains name, since each each Captain is linked to one captainID.

ItemID gives us its description and weight, and itemID + the shipmentID gives us the quantity of the item.

Our schema for this can be seen as such:
R(shipmentID, shipmentDate, origin, destination, shipID, shipName, CaptainID, CaptainName, ItemID, Description, Weight, Quantity)

At this point we should probally select our primary key. Since we select our primary key from our list of candidate keys, we need to first assess all our candidate keys. Recall that a candidate key is the minimum set of attributes necessary to uniquely identify a tuple.

The first attribute we should look at is obviously ShipmentID, since it can determine the most amount of attributes within the tuple. Since it does not determine all of our attributes, it by itself is not a candidate key, so we need another attribute to pair it with. We need an attribute or a set of attributes that can functional determine itemID, description, weight, and quantity. itemID determines description and weight and when it is paired with shipmentID, it also determines quantity. Therefore, our first candidate key is: $\{\underline{\text{ShipmentID}}\ \underline{\text{ItemID}}\}$.

Now, our prime attributes are `ShipmentID` and `ItemID` and our non-prime attributes are `ShipmentData, origin, destination, shipID, ShipName, CaptainID, CaptainName, Description, Weight` and `quantity`.

8 Normalization

8.1 First Normal Form $_1NF$

The requirement for a relation to be in $_1NF$ is that all of the values must be **atomic**.

What this usually looks like is a table with multiple values in a single cell. A non-$_1NF$ relation would not even technically count as a relation. This table has a cell that is non-atomic,

It looks X *would* have been the primary key, but it's not doing its job of uniquely determining Z, which is showing as a *repeating group* so X can't be a key.

X	Y	Z
x1	y1	z1
		z2
		z3
x2	y2	z4
x3	y2	z5

The notation for this "pseudo-relation". like the one above would be to use **inner parenthesis** on the repeating group, i.e.

R(<u>X</u>, Y,(Z))

This is not ${}_1NF$, and has functional dependencies:
$X \rightarrow Y$
$X, Z \rightarrow Z$ **but** $X \nrightarrow Z$

To move this pseudo-relation into an actual relation that doesn't violate ${}_1NF$, we need to choose a *real* primary key that meets the requirements. We do that using the FDs. In this case, (X,Z) works.

Changing the primary key yields: - ***R***(<u>X</u>, Y, <u>Z</u>)

X	Y	Z
x1	y1	z1
x1	y1	z2
x1	y1	z3
x2	y2	z4
x3	y2	z5

Now everything is atomic, and we are in ${}_1NF$. Notice that this did introduce a new update anomaly, but the other normal forms will take care of it. It is more important to get into ${}_1NF$ for now. As another example, consider the following unnormalized pseudo-relation:

R(<u>A</u>, B, C, (d_1, d_2, d_3), E, F)

A $\rightarrow$ B, C, E, F
A, $d_1 \rightarrow d_2, d_3$

Notice that (d_1, d_2, d_3) is a repeating group. A is not enough to form a primary key, it needs d_1 to be able to determine d_2 and d_3. So, the actual primary key in this case should be (A, d_1), making the ${}_1NF$ relation.

$\boldsymbol{R}_{1NF}$(<u>A</u>, B, C, d_1, d_2, d_3, E, F)

8.2 Second Normal Form ${}_2NF$

Second Normal Form (${}_2NF$) has to do with the concept of *full dependence.*

Given two sets of attributes, X and Y, we can say that Y is *fully dependent* on X, if (and only if)

- $X \rightarrow Y$
- No subset of X determines Y

A relation is in ${}_2NF$ if:

- It already meets the requirements of ${}_1NF$,
- All *non-prime* attributes of the relation are *fully dependent* upon the **entire** primary key.

What breaks ${}_2NF$ is when attributes are dependent upon only **part** of the primary key.

To fix ${}_2NF$ violations once we're in ${}_1NF$, *decomposition* is the solution.

Going back to our earily example: ***EmpProj***(EmpID, Project, Supv, Dept, Case)

EmpID	Project	Supv	Dept	Case
e1	p1	s1	d1	c1
e2	p2	s2	d2	c2
e1	p3	s1	d1	c3
e3	p3	s1	d1	c3

EmpID, Project → Supv, Dept, Case
EmpID → Supv, Dept
Supv → Dept

A quick glance confirms all values are atomic, so ${}_1NF$ is confirmed.

There is a ${}_2NF$ violation caused by (EmpID → Supv, Dept) because the primary key is (EmpID, Project), but only EmpID is on the LHS.

Observing the instance Data, you should easily see that the attributes of the RHS cause update anomalies in this table. We also can't insert a new employee with no project (insertion anomaly). These are symptoms of the ${}_2NF$ violation.

Decomposition Pattern

There is a pattern to follow for the decomposition.

Start with the original relation, and the FD that causes the violation.

EmpProj(EmpID, Project, Supv, Dept, Case) (Original relation)

EmpID → Supv, Dept (violates ${}_2NF$)

The attributes on the right-hand side of the functional dependency that violates ${}_2NF$ are removed from the original relation and placed into a newly created relation that has the FD's left-hand side as the primary key. A *foreign key* links the attributes from the left-hand side in the original table (the left-hand side is not removed) to the corresponding tuple in the new table, where it is the *primary key.*

- ***EmpProj***(EmpID, Project, Case)
- ***Employee***(EmpID, Supv, Dept)

EmpProj(EmpID, Project, Case)

EmpID	Project	Case
e1	p1	c1
e2	p2	c2
e1	p3	c3
e3	p3	c3

Employee(EmpID, Supv, Dept)

EmpID	Supv	Dept
e1	s1	d1
e2	s2	d2
e3	s1	d1

While the single relation we began with violated ${}_2NF$, this version with the two relations does not. There are still some anomailes remaining if you look closely, so we will look into ${}_3NF$.

8.3 Third Normal Form $_3NF$

To be in *Third Normal Form*, ($_3NF$), a relation must

- Already qualify to be in $_2NF$
- None of the non-prime attributes may be *transitvely dependent* upon the primary key.

By definition, all non-prime attributes are *functionally* dependent upon the primary key. What makes a *transitive dependency* is that there is also some non-prime attributes (which also depends on the key) that also functionally determines the attribute.

To quickly identify the *transitive dependencies* from the list of FDs, look on the left-hand side for attributes that are non-prime in the context of the current relation.

EmpProj(EmpID, Project, Case)
Employee(EmpID, Supv, Dept)

- Functional Dependencies
  - EmpID , Project$\rightarrow$ Supv, Dept, Case
  - EmpID $\rightarrow$ Supv, Dept
  - **Supv** $\rightarrow$ **Dept** (Transitive dependency)

In this case, the FD that causes our relations to violate $_3NF$ is (Supv $\rightarrow$ Dept), and the violation happens in the ***Employee*** relation. If you refer back to the instance data of that in the $_2NF$ solution, you can see that the violation can cause anomalies, so we want to fix it.

Just like in $_2NF$, we fix $_3NF$ by **decomposing** using the FD that causes the violation to occur.

Note:-

At no point do we change the FDs.

Following the same pattern we used for decomposing to get into $_2NF$, we start with the relation that has the violation, and the FD that causes the violation to occur.

- ***Employee***(EmpID, Supv, Dept)
- Supv $\rightarrow$ Dept

The attributes on the right-hand side of the FD are removed from the violating relation and placed into a newly created relation that has the FD's left-hand side as its primary key. A *foreign key* links the attribute from the left-hand side in the original table (the left-hand side is not removed) to the corresponding tuple in the new table, where it is the *primary key*.

- ***Employee***(EmpID, Supv)
- ***SupvDept***(Supv, Dept) (new relation)

The right-hand side (Dept) that was a violation when it was in ***Employee*** because the left-hand side (Supv) was non-prime is no longer there to cause the problem. It is in the new relation where the left-hand side (Supv) is the primary key, and therefore we don't have a *transitive dependency*. These two relations no longer have the $_3NF$ violation.

Final results:

- ***EmpProj***(EmpID, Project, Case)
- ***Employee***(EmpID, Supv)
- ***SupvDept***(Supv, Dept)

EmpID	Project	Case
e1	p1	c1
e2	p2	c2
e1	p3	c3
e3	p3	c3

EmployeeProj

EmpID	Supv
e1	s1
e2	s2
e3	s1

Employee

Supv	Dept
s1	d1
s2	d2
s1	d1

SupvDept

Chapter 2

Converting ERD to Relational DB

The Conceptual Model is great for planning and documentation, but it needs to be converted to a logical model in order to actually be used. We will look at how we can use the relational data model to represent the schema that is represented by an ER diagram.

Our ER diagram is made up of *entities, relationships* and *attributes*.

The relational data model has *relations*, filled with *tuples*, which are made up of *attributes*

We will need to use these tools together to design a relational database in $_3NF$ that can hold the data from our design.

1 The steps

The process of converting an ER diagram to a relational database can be boiled down to a two step process.

1. Handle all of the **entities**
2. Handle all of the **relationships**

1.1 Step 1: Handle Entities

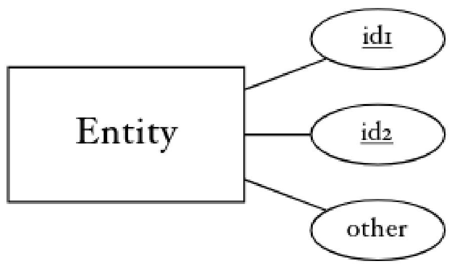
We will start with the **entities**, because they can stand on their own, unlike *relationships* or *attributes*.

In general, each *entity* will get its own *relation*. The *attributes* of the *entity* will become *attributes* in the schema of the relation created. There are some special cases to take into account, which will be handled from most independent to least, so:

- a. Strong (non-weak) entities that are *not* subtypes
- b. Strong (non-weak) entities that *are* subtypes
- c. Weak entities

Note that there is no reason to make a relation for a “date” entity or similar. The single value for the data is enough to determine it, and any other data associated with it is generally happening through a relationship anyway. Think about what data would go into such a table and how little use there would be for storing it separately.

1.1.1 Strong Entity, Not Subtype

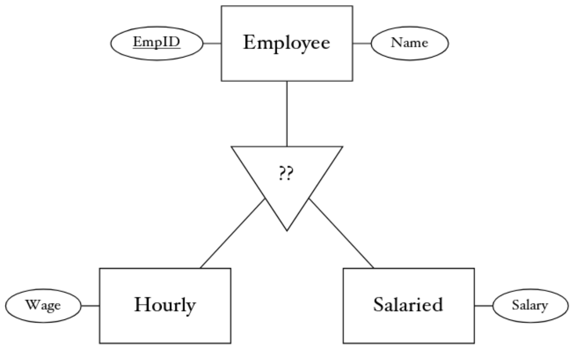


Example of a strong entity that is not a subtype.

- Make a new relation, whose name will be the same as the name of the entity
- The *primary key* of the relation will be all of the identifier attributes, taken together
- All attributes of the entity become attributes of the relation.

Schema of new relation from entity pictured above: ***Entity***(<u>idr</u>, <u>id2</u>, other)

1.1.2 Strong Entity, Subtype



Employee is a supertype (not subtype) so it gets handled in the previous step.

- ***Employee***(<u>EmpID</u>, Name)

Hourly and **Salaried** are each strong, but they are subtypes (each is a type of **Employee**), so they are handled here.

- This type of inheritance means that the subtypes *are* types of the supertype, so they are identified by **Employee's** EmpID.
- There are *two* methods for handling these subtypes.

Method 1 - Big Table

The first method involves putting the attributes of the subtypes into the relation made for the supertype. So, the original relation:

- ***Employee***(<u>EmpID</u>,Name)

Would become something like:

- ***Employee***(<u>EmpID</u>, Name, Wage, Salary)

But it would beed to be modified to indicate which subtypes a given employee belongs to. This is handled differently depending on the IS-A's configuration

- For *disjoint subtypes*, where an instance of the supertype can only be one of the subtypes at a time, we can add an attribute, EmpType that has a value indicating which type this employee is:
  - ***Employee***(<u>EmpID</u>, Name, EmpType, Wage, Salary)

Note:-

For *generalization*, EmpType would not allow `NULL`. For *specialization*, it would be allowed.

- For *overlapping subtypes*, it is possible to be more than one at a time, so we need an individual true/false answer for each type:
  - ***Employee***(<u>EmpID</u>, Name, IsHourly, Wage, IsSalaried, Salary)

In this case, nothing about the schema would indicate *generalization* vs. *specialization.*

Below is an example of what some instance data for the design above could look like:

EmpID	Name	IsHourly	Wage	IsSalaried	Salary
1	Jim	false	NULL	true	$56,000
2	Bob	true	$20.00	false	NULL
3	Sally	false	NULL	true	$75,000
4	Jane	false	NULL	true	$65,800
5	Arvind	true	$5.00	true	$60,000

Note that, while it is probaly not common for someone to be both hourly and salaried, this design would support that.

Method 2

Method 2 involves creating a new relation for the subtype entity.

- The name of new relation would be the same as the name of the entity.
- The *primary key* of the new relation would be the same as the *primary key* for the supertype's relation.
- The *primary key* is also a *foreign key* to the existing table.
- An instance of the supertype entity will only have a tuple in the subtype relation if it is a member of that subtype, so we will not need any extra attribute like we did in method 1.
- The *foreign key* can be used to look up any of the attributes that are being inherited from the supertype.

The supertype table remains unchanged with method 2:

Employee($\underline{\text{EmpId}}$, Name)

The subtypes each get their own table.

- ***Hourly***($\underline{\text{EmpID}\dagger}$, Wage)
- ***Salaried***($\underline{\text{EmpID}\dagger}$, Salary)

The (†) will be used in these slides to indicate that the attribute is part of a *foreign key* (and, in this example, the whole thing).

In method 2, the supertype table is not changed, so the same data we used in the previous method would look like this:

Employee (supertype entity)

EmpID	Name
1	Jim
2	Bob
3	Sally
4	Jane
5	Arvind

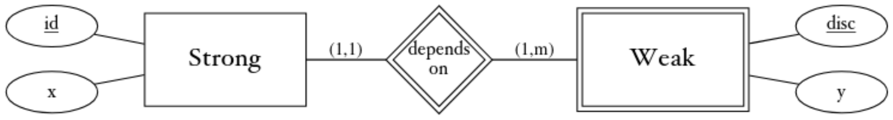
Hourly (subtype entity)

EmpID	Wage
2	$20.00
5	$5.00

Salaried (subtype entity)

EmpID	Salary
1	$56,000
3	$75,000
4	$65,800
5	$60,000

1.1.3 Weak Entity



Example with weak entity

The strong entity would already have a relation from section 1.1.1

- ***Strong***(id, x)

The weak entity gets its own relation. The *primary key* will be the concatenation of the weak entity's *discrimator* with the strong entity's identifier. The other attributes of the entity are brought in as non-prime attributes.

- ***Weak***(id†, disc,y)

The *identifier* portion is a foreign key to the **Strong** relation.

Note:-

A *discriminator* looks like an *identifier* (underlined attribute), but it is attached to a *weak entity.*

1.1.4 Entities: Functional Dependencies

The only *functional dependencies* introduced by the entites of an ER diagram are the ones introduced when the *identifiers* become *primary keys.* Remember that a primary key has to functioanlly determine all of the other attributes in a relation.

1.2 Step 2: Handle Relationships