Database Design Notes

Jeff Song

Mar. 2020

Table of Contents

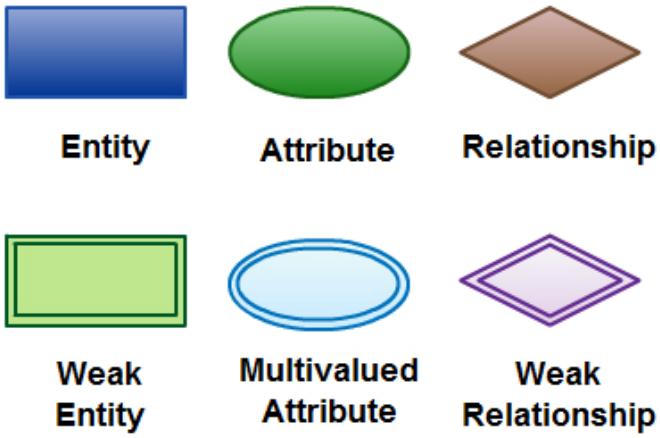
1. Conceptual Design - ER Modeling

ER modelling is based on two concepts:

- Entities, defined as tables that hold specific information (data)
- Relationships, defined as the associations or interactions between entities

1.1. Notations of ERD

1.1.1. Elements in Chen Diagram

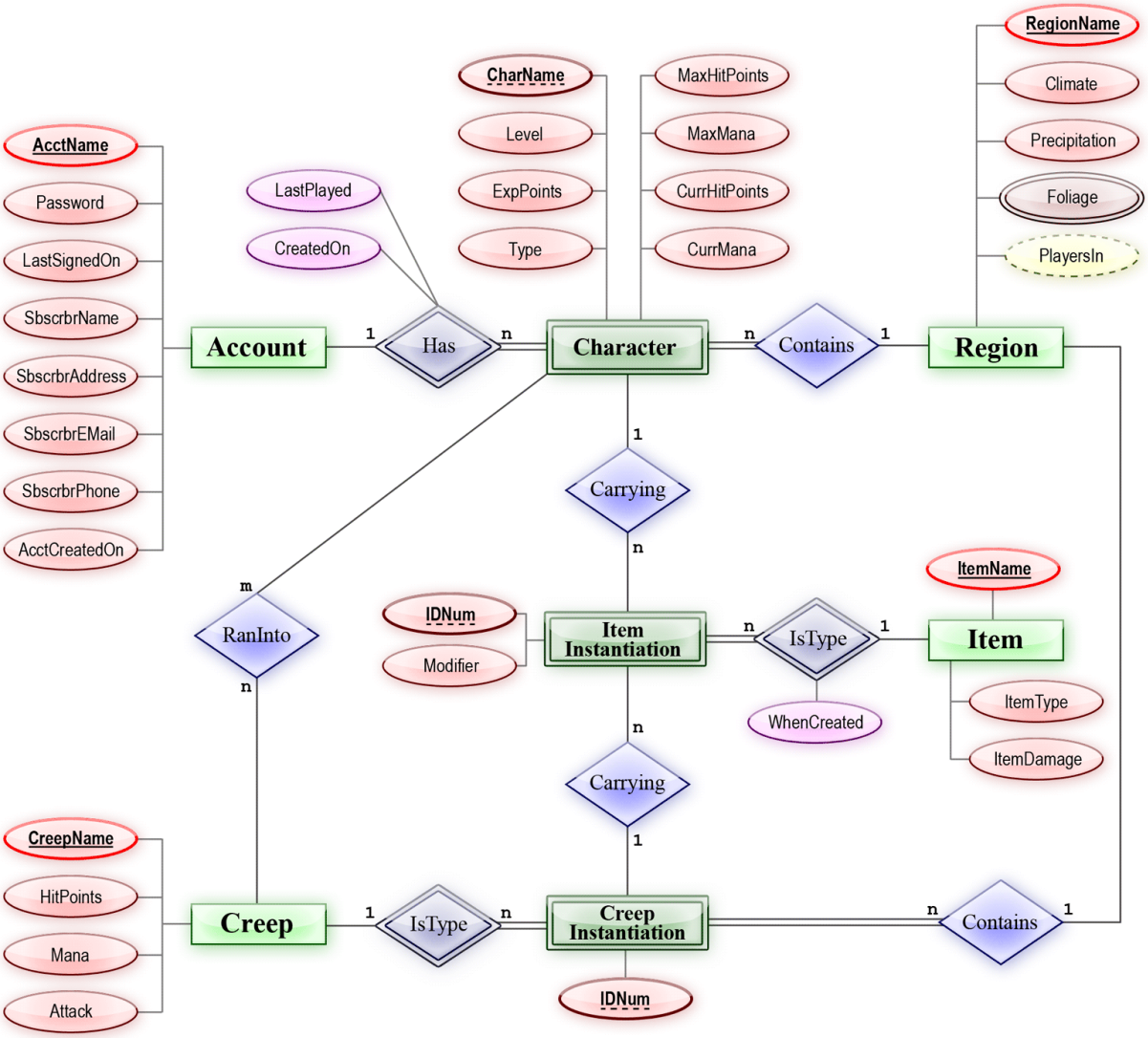


Elements in ER diagrams

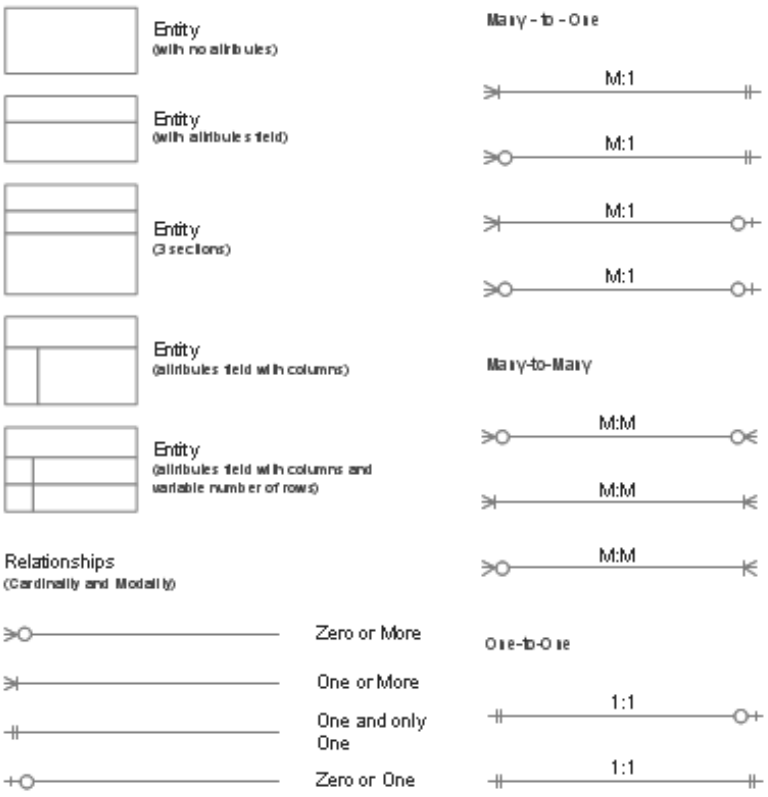
1.1.2. Relationship notation in Chen Diagram

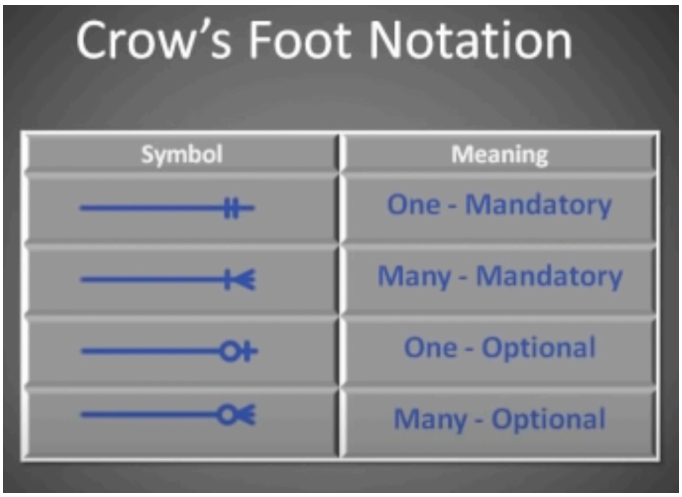
Cardinality in ER diagrams using UML notation

1.1.3. Example of Chen Diagram



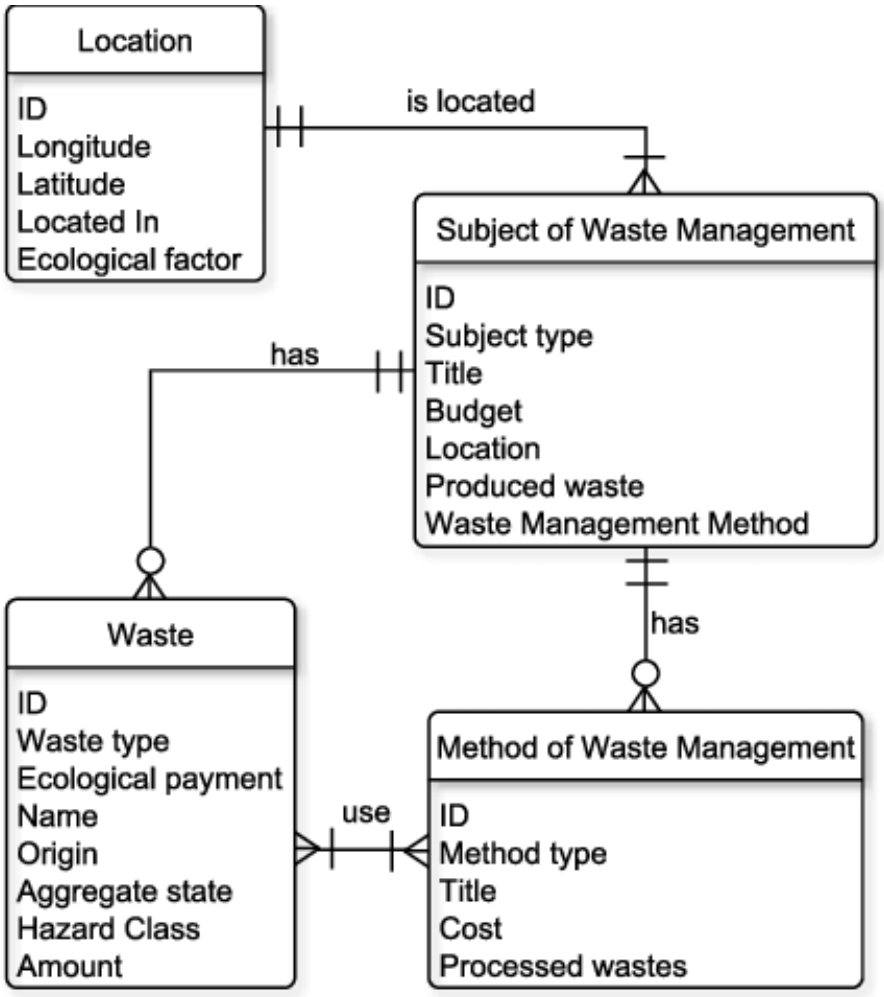
1.1.4. Crow's Foot Notations





Crow's foot notation

1.1.5. Example of Crow's Foot Notations



1.2. Kinds of Entities and the Way to Identify the Entities

1.2.1. Independent entities

Independent entities, also referred to as kernels, are the backbone of the database. They are what other tables are based on.

Kernels have the following characteristics:

- They are the building blocks of a database.
- The primary key may be simple or composite.
- The primary key is not a foreign key.
- They do not depend on another entity for their existence.

1.2.2. Dependent entities

Dependent entities, also referred to as derived entities, depend on other tables for their meaning.

These entities have the following characteristics:

- Dependent entities are used to connect two kernels together.
- They are said to be existence dependent on two or more tables.
- Many to many relationships become associative tables with at least two foreign keys.
- They may contain other attributes.
- The foreign key identifies each associated table.
- There are three options for the primary key:
    1. Use a composite of foreign keys of associated tables if unique
    2. Use a composite of foreign keys and a qualifying column

3. Create a new simple primary key

1.2.3. Characteristic entities (complementary description of the major entity)

Characteristic entities provide more information about another table.

These entities have the following characteristics:

- They represent multivalued attributes.
- They describe other entities.
- They typically have a one-to-many relationship.
- The foreign key is used to further identify the characterized table.
- Options for primary key are as follows:
    1. Use a composite of foreign key plus a qualifying column
    2. Create a new simple primary key.

1.3. Attributes

Each entity is described by a set of attributes (e.g., Employee = (Name, Address, Birthdate (Age), Salary).
Each attribute has a name, and is associated with an entity and a domain of legal values.
However, the information about attribute domain is not presented on the ERD.
In the entity relationship diagram, each attribute is represented by an oval with a name inside.

There are a few types of attributes you need to be familiar with. Some of these are to be left as is, but some need to be
adjusted to facilitate representation in the relational model. This first section will discuss the types of attributes. Later
on we will discuss fixing the attributes to fit correctly into the relational model.

1.3.1. Simple attributes

Simple attributes are those drawn from the atomic value domains; they are also called single-valued attributes. In the COMPANY database, an example of this would be: Name = {John} ; Age = {23}

1.3.2. Composite attributes

Composite attributes are those that consist of a hierarchy of attributes. Using our database example, and shown in Figure 8.3, Address may consist of Number, Street and Suburb. So this would be written as → Address = {59 + 'Meek Street' + 'Kingsford'}

1.3.3. Multivalued attributes

Multivalued attributes are attributes that have a set of values for each entity. An example of a multivalued attribute from the COMPANY database, as seen in Figure 8.4, are the degrees of an employee: BSc, MIT, PhD.

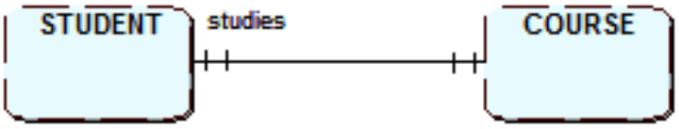
1.3.4. Derived attributes

Derived attributes are attributes that contain values calculated from other attributes. An example of this can be seen in Figure 8.5. Age can be derived from the attribute Birthdate. In this situation, Birthdate is called a stored attribute, which is physically saved to the database.

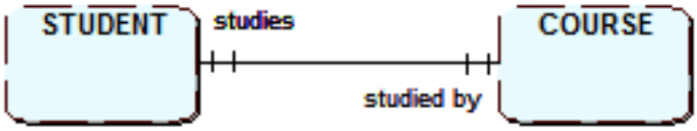
1.4. Entity Relationships

Relationships are the glue that holds the tables together. They are used to connect related information between entities.

If there is a relationship between an occurrence of one entity type and an occurrence of another entity type, then it is shown on the entity relationship diagram as a line linking the two entity symbols. The relationship between the two entities should be labelled by using a suitable verb. For example the relationship "STUDENT Studies a COURSE" would be represented as follows:



As it is important to consider a relationship from **both directions** you should also label the relationship from COURSE to STUDENT as follows:



The relationship labels should be positioned as above, near to the relevant entities to aid readability.

1.4.1. Relationship Cardinality

Once you have established a relationship between two entity types it is important to consider how many occurrences of one entity could be related to the other entity. This is referred to as **"cardinality"**.

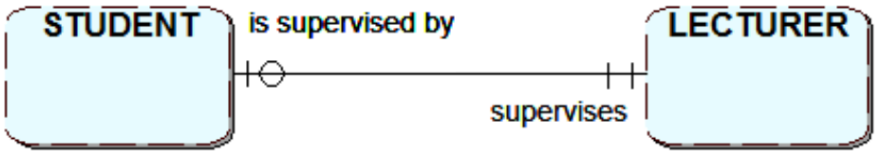
There are three types of relationship cardinality:

One to One abbreviated as **1:1**
One to many abbreviated as **1:M**
Many to Many abbreviated as **M:M** or **M:N**

1.4.2. Relationship Optionality

When describing an entity relationship you need to record the fact on the ERD that in some cases an occurrence of an entity type may not always be present, in which case the relationship is said to be **optional**. Using the previous cardinality example, the model states that a lecturer supervises a student. However, what if some lecturers do not act as supervisors to students? In this situation an occurrence of LECTURER will not always be related to an occurrence of STUDENT so it will be an optional relationship. However, if you consider the relationship from the STUDENT perspective it is still present as all students must have a supervising LECTURER.

To denote that a relationship can be optional a small circle is included on the relationship line at the end that is optional. The following shows the optional 1:1 relationship between STUDENT and LECTURER:



A one to one (1:1) relationship with optionality.

The circle might be viewed as the letter O for optional but it is best considered as a zero.

1.5.

2. Logical Database Design

- Convert ERD into a logical database design
- Identify Relations, Primary Keys and Foreign Keys
- Provide the logical design for relationships of different cardinalities

An ERD can be converted into a logical design suitable for a relational database by defining a set of relations, some of which have been derived directly from the ERD entities, others coming from the relationships. The relations include all the keys which will be required to link the tables in the database when it is created.

Summary of the rules for deriving the relations and their keys from the relationships:

1:1, the FK can go in either relation or combine to form one relation
1:1 with optionality at one end, the PK from the mandatory end is stored as FK at optional end
1:1 with optionality at both ends, create a new relation including a FK from either end used as PK
1:M, the PK from the one end goes in the many end as a FK
1:M optionality at the one end, create a new relation with FK from many end it becomes the PK
M:M, create a new relation with a compound PK made up of the FKs from both relations

Key things to bear in mind are:
- tables should not have repeating attributes
- a foreign key should never hold a null value.

2.1. Relations

A **Relations** is a logical construct that is similar to a table. i.e. CUSTOMER (**CustomerID**, FirstName, LastName, …) is a Relation with the **CustomerID** as the Primary Key.

2.2. Keys

- A Key is the entity identifier, uniquely identifies an entity concurrence.
- A Primary Key is 'chosen' for a given relation / table
- A Candidate Key is a possible Primary Key
- A Compound Key is a key consisting of two or more attributes.
- A Foreign Key is used to establish the relationships between entities.

An important constraint on an entity is the key.
The key is an attribute or a group of attributes whose values can be used to uniquely identify an individual entity in an entity set.

2.2.1. Candidate key

A candidate key is a simple or composite key that is unique and minimal.
It is unique because no two rows in a table may have the same value at any time.
It is minimal because every column is necessary in order to attain uniqueness.

2.2.2. Composite key

A composite key is composed of two or more attributes, but it must be minimal.

2.2.3. Primary key

The primary key is a candidate key that is selected by the database designer to be used as an identifying mechanism for the whole entity set. It must uniquely identify tuples in a table and not be null. The primary key is indicated in the ER model by underlining the attribute.

- A candidate key is selected by the designer to uniquely identify tuples in a table. It must not be null.
- A key is chosen by the database designer to be used as an identifying mechanism for the whole entity set. This is referred to as the primary key. This key is indicated by underlining the attribute in the ER model.

2.2.4. Secondary key

A secondary key is an attribute used strictly for retrieval purposes (can be composite), for example: Phone and Last Name.

2.2.5. Alternate key

Alternate keys are all candidate keys not chosen as the primary key

2.2.6. Foreign key

A foreign key (FK) is an attribute in a table that references the primary key in another table OR it can be null.

Both foreign and primary keys must be of the same data type.

3. Integrity rules and constrains

3.1. Domain Integrity

Domain restricts the values of attributes in the relation and is a constraint of the relational model.
However, there are real-world semantics for data that cannot be specified if used only with domain constraints.

3.2. Entity integrity

To ensure entity integrity, it is required that every table have a primary key.
Neither the PK nor any part of it can contain null values. This is because null values for the primary key mean we cannot identify some rows. For example, in the EMPLOYEE table, Phone cannot be a primary key since some people may not have a telephone.

3.3. Referential integrity

Referential integrity requires that **a foreign key must have a matching primary key** or it must be null. This constraint is specified between two tables (parent and child); it maintains the correspondence between rows in these tables. It means the reference from a row in one table to another table must be valid.

Examples of referential integrity constraint in the Customer/Order database of the Company:

- Customer(**CustID,** CustName)
- Order(OrderID, **CustID**, OrderDate)

To ensure that there are no orphan records, we need to enforce referential integrity. An orphan record is one whose foreign key FK value is not found in the corresponding entity – the entity where the PK is located. Recall that a typical join is between a PK and FK. The referential integrity constraint states that the customer ID (CustID) in the Order table (child table) must match a valid CustID in the Customer table (parent table).

Most relational databases have declarative referential integrity. In other words, when the tables are created the referential integrity constraints are set up.

Here is another example from a Course/Class database:

- Course(**CrsCode**, DeptCode, Description)
- Class(**CrsCode, Section**, ClassTime)

The referential integrity constraint states that CrsCode in the Class table must match a valid CrsCode in the Course table. In this situation, it's not enough that the CrsCode and Section in the Class table make up the PK, we must also enforce referential integrity.
When setting up referential integrity it is important that the PK and FK have the same data types and come from the same domain, otherwise the relational database management system (RDBMS) will not allow the join.

3.4. Foreign key rules (additional to the Referential Integrity)

Additional foreign key rules may be added when setting referential integrity, such as what to do with the child rows (in the Orders table) when the record with the PK, part of the parent (Customer), is deleted or changed (updated). For example, the Edit Relationships window in MS Access (see Figure 9.1) shows two additional options for FK rules: **Cascade Update** and **Cascade Delete**. If these are not selected, the system will prevent the deletion or update of PK values in the parent table (Customer table) if a child record exists. The child record is any record with a matching PK. In some databases, an additional option exists when selecting the Delete option called Set to Null. In this is chosen, the PK row is deleted, but the FK in the child table is set to NULL. Though this creates an orphan row, it is acceptable.

3.5. Enterprise Constraints (Semantic Constraints)

Enterprise constraints – sometimes referred to as semantic constraints – are additional rules specified by users or database administrators and can be based on multiple tables.

Here are some examples.

- A class can have a maximum of 30 students.
- A teacher can teach a maximum of four classes per semester.
- An employee cannot take part in more than five projects.
- The salary of an employee cannot exceed the salary of the employee's manager.

3.6. Business Rules (used to identify the cardinality and connectivity)

Business rules are obtained from users when gathering requirements. The requirements-gathering process is very important, and its results should be verified by the user before the database design is built. If the business rules are incorrect, the design will be incorrect, and ultimately the application built will not function as expected by the users.

Some examples of business rules are:

- A teacher can teach many students.
- A class can have a maximum of 35 students.
- A course can be taught many times, but by only one instructor.
- Not all teachers teach classes.

3.7. Cardinality and connectivity

Business rules are used to determine cardinality and connectivity. Cardinality describes the relationship between two data tables by expressing the minimum and maximum number of entity occurrences associated with one occurrence of a related entity.

3.8. Convert ERD into Relations

1. Identify the Skeletal Relation with major attributes of the entity to represent each Strong Entity, assign **Primary Key** to unique the entity concurrences in the Relation. The **Primary Key** could be a single attribute, or a compound of many attributes.

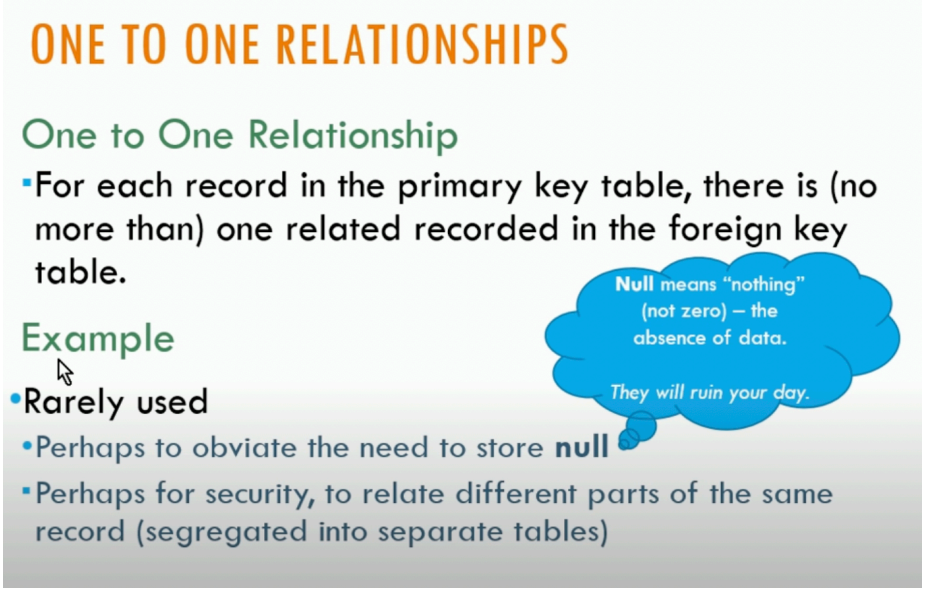
2. Identify the Relation for the weak entity, save the major attributes of corresponding entity in the Relation, copy the **Primary Key** from the Relation which representing the Strong Entity as the ***Foreign Key*** in the Relation representing the weak entity related to the strong entity. Assign the **Primary Key** for this Relation which representing weak entity. The **Primary Key** for the weak entities could be a simple attribute (most be a Surrogate Key), or a single ***Foreign Key***, or a compound key. If the Primary Key is a compound key, it could be ether a set of foreign keys refer to other entities or a combination of native attributes and foreign keys.

3. Resolving the relationships of different cardinality. Resolving exclusive relationships. Resolving Fan Trap and Chasm Trap modeling problems.

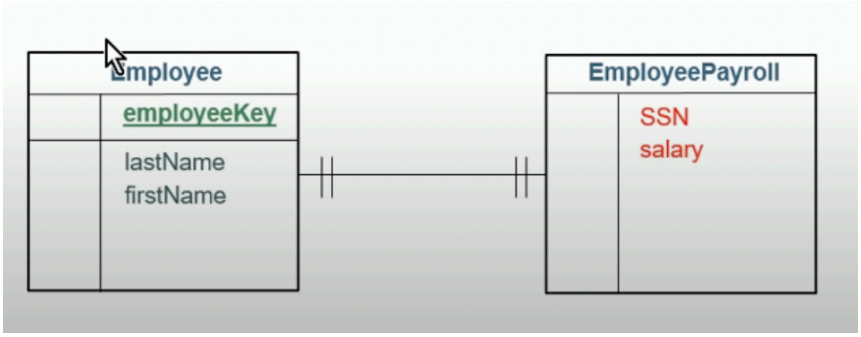
3.9. Convert Simple and Complex Relationships

3.9.1. 1: 1 with mandatory at each end relationship (-|-|----------|-|-)

A one to one (1:1) relationship is the relationship of one entity to only one other entity, and vice versa. It should be rare in any relational database design. In fact, it could indicate that two entities actually belong in the same table.

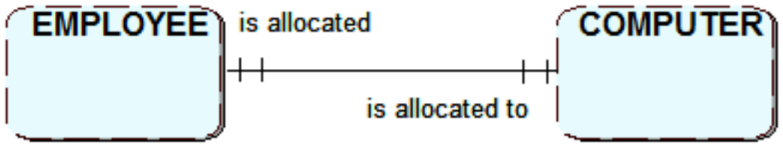


It is used to protect some information of the entity by create a separate table link to the primary table with Mandatory One relationship.



To create the Relations for 1:1 mandatory relationship, takes the set of attributes for one entity as the attributes of another entity, merge the set into the Relation represent another one.

Firstly, you may have a 1:1 relationship which is **mandatory at each end**, for example:



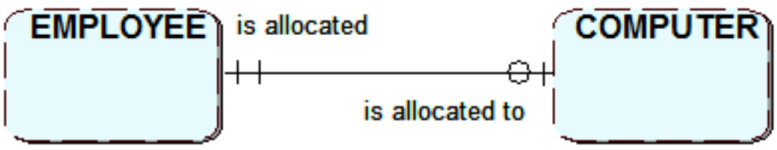
In this type of relationship there will only ever be one set of attributes from one entity type matched with one set of attributes from the other entity type, so both sets can be merged to form one database table. At the logical design stage the two entities are effectively merged into one relation with a single primary key. Either of the following skeleton relations could be used:

EMPLOYEE (**Employee_No,** Computer_ID)
or
COMPUTER (**Computer_ID,** Employee_No)

3.9.2. 1: 1 with optionality at one end relationship (-|-|----------0-|-)

1. Create a Relation for each of the entities.
2. Post the PK of the Relation representing the entity at the non-optional end to the Relation at the optional end as the FK.

The second situation is a one-to-one relationship with **optionality at one end**, for example:

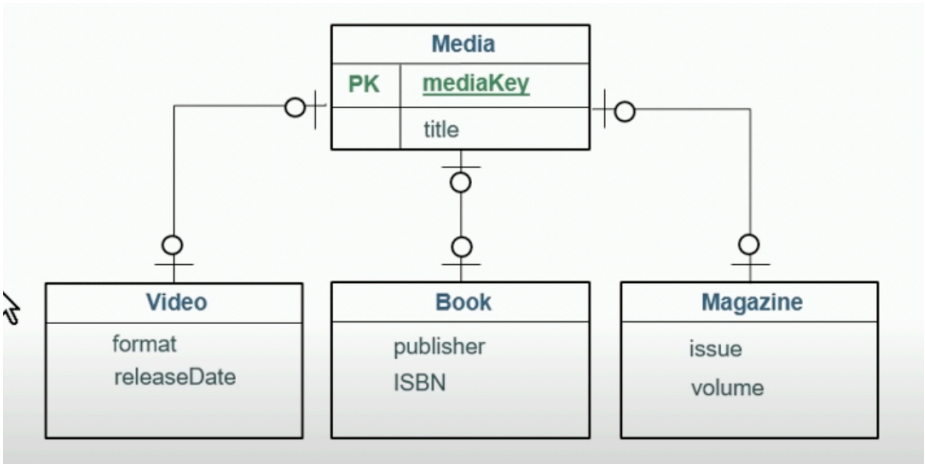


If an employee may not be allocated a computer then in this situation a relation must be created for each of the entities. The key of the entity at the non-optional end is posted into the relation for the optional entity. This approach avoids ending up with null value **Employee_No** attribute in the COMPUTER table

EMPLOYEE (**Employee_No**)
COMPUTER (**Computer_ID**, ***Employee No***)

3.9.3. 1: 1 with optionality at both ends relationship (-|-0----------0-|-)

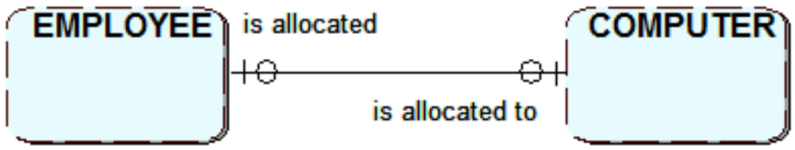
ONE on ONE relationship is often used to help breaking down a summarized entity which consist of different sub-entities have non-conformed attributes, to dedicated entities for each sub-entity.



1. Create a Relation for each of the entities.

2. Create a Link Relation to bridge the relationship between the two entities. The PK from both ends into this bridge Relation as FK, take ether of the FK as the PK for this intermediate Relation.

Finally, consider the third situation, which is a one-to-one relationship with **optionality at both ends**, for example where an employee may not be allocated a computer or where a computer may not be allocated to an employee:



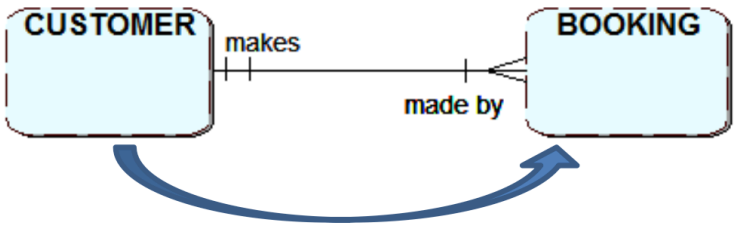
Three relations are needed, one for each entity and one to express the relationship between the employees and their computers:

EMPLOYEE (**Employee_No**)
COMPUTER (**Computer_ID**)
COMPUTER_ALLOCATION (***Employee_No***, ***Computer_ID***)

Note that **either** of the original identifiers could have been used as the primary key for the third relation. If a history of which computers have been allocated to an employee is needed, a further attribute would need to be included in the primary key e.g. date_allocated.

3.9.4. 1: M with mandatory relationship (-|-| --------|-<)

For example, earlier you modelled the one-to-many relationship between CUSTOMER and BOOKING on the conceptual level. To achieve a logical design for the tables in your database you will need to produce two relations: CUSTOMER and BOOKING. In order to take into account the link between these two relations you must copy the primary key of CUSTOMER (which is at the "one-end" of the relationship) into the relation BOOKING (at the "many-end" of the relationship). So the Customer_no. becomes a foreign key in the relation BOOKING. The following diagram emphasises this:



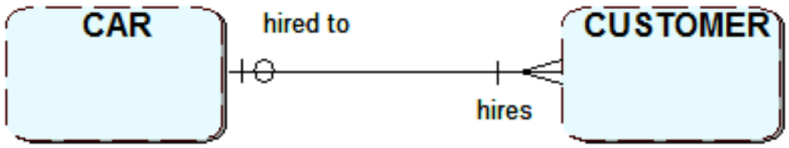
This ERD would result in the following two relataions:

CUSTOMER (**Customer no.**, customer name, customer_address, …)

BOOOKING (**Booking ref.**, Booking_date, ***Customer_no,*** **…**)

3.9.5. 1: M with optionality at the one end relationship (-|-0--------|-<-)

Consider the example used in Exercise 4, only now there is optionality at the one end of the relationship:



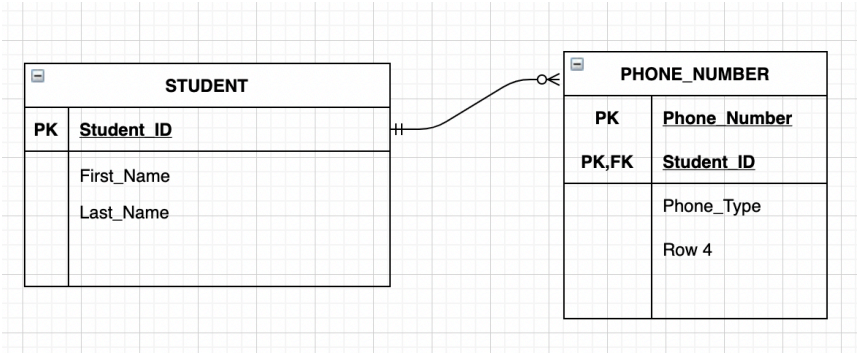
This would be interpreted as "a customer may not be allocated a car for hire immediately i.e. when they make a booking". In this situation, if you tried to assign the Car_reg. as the foreign key in CUSTOMER this would result in an empty value as there is no car reg. to enter! The value would be what is referred to as **null**; however it is **not acceptable** to allow a null value for any key. To avoid this situation, three relations are created in the logical design:

CAR (**Car_reg.**)
CUSTOMER (**Customer_no.**)
HIRED_CAR (***Customer_no.***, *Car_reg.*)

Note that the new relation contains two foreign keys ***Customer_no.*** and ***Car_reg.*** The ***Customer_no.*** was chosen as the primary key for the new relation as the customer is only allocated one hire car at a time. With this arrangement an entry is only made in the HIRED_CAR table when a car is actually allocated to a customer.

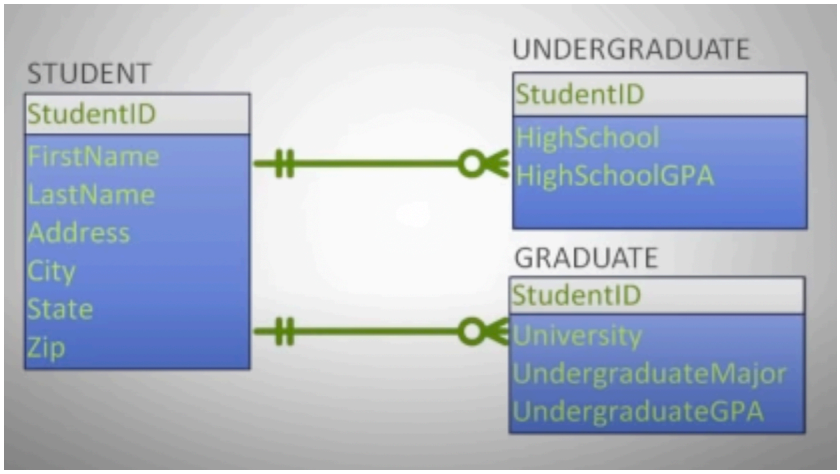
3.9.6. 1: M with optionality at many end relationship (-|-| --------0-<-)

A one to many (1:M) relationship should be the norm in any relational database design and is found in all relational database environments. For example, one department has many employees.



For student, it is optional to have one or more phone numbers.
But for each phone number, it is mandatory to have a student.

Create Sub-Entities, build One-Mandatory-to-Many-Optional relations with primary entity, to break down the major entity's attributes which does not fit for all instances into new tables. If we left them in the major entity table, there will be a lot empty fields for most of the instance's record.

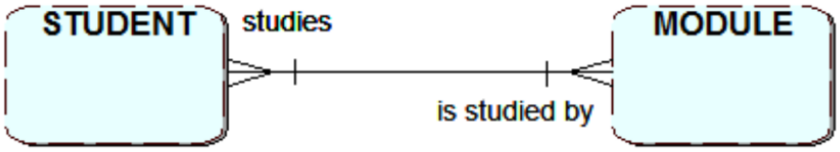


The Relations results are the same as the (1: M) with mandatory at each side. (-|-| --------|-<-)

3.9.7. 1: M with optionality at both end relationship (-|-0 --------0-<-)

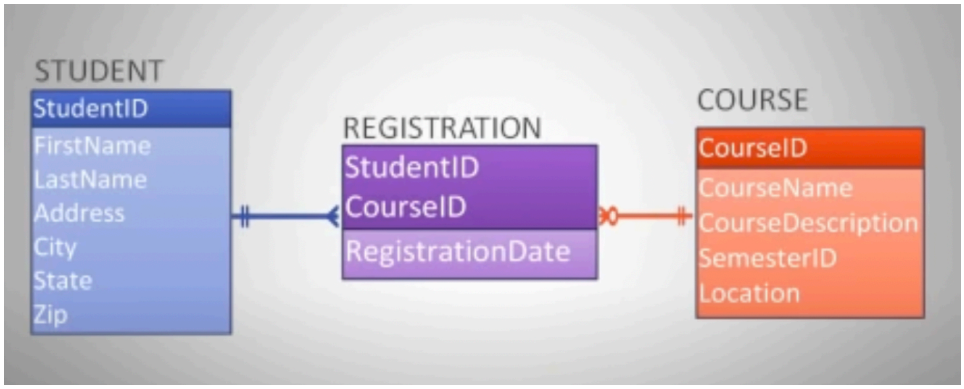
As same as the (M:N) with mandatory at each side. (-|-0--------|-<-)

3.9.8. (M: N) relationship. (->-| -------- |-<-)



STUDENT (**Student_id**)
MODULE (**Module_Id**)
STUDY (***Student_id, Module_id***)

Note that in the last relation here, which represents the M:N relationship, both foreign keys are needed together to form a compound primary key, which will uniquely identify occurrences that will exist in the resulting relationship table STUDY.



Example of mapping an M:N binary relationship type

• For each M:N binary relationship, identify two relations.

• A and B represent two entity types participating in R.

• Create a new relation S to represent R.

• S needs to contain the PKs of A and B. These together can be the PK in the S table OR these together with another

simple attribute in the new table R can be the PK.

• The combination of the primary keys (A and B) will make the primary key of S.

3.9.9. (M: N) with optionality at one end relationship. (->-0 -------- |-<-)

As same as the (M:N) with mandatory at each side. (->-| --------|-<-)

3.9.10. (M: N) with optionality at one end relationship. (->-0 -------- 0-<-)

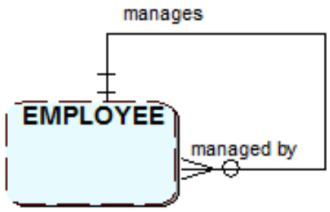
As same as the (M:N) with mandatory at each side. (->-| --------|-<-)

3.9.11. Recursive (1: M) with/out optionality at Many side relationship

A unary relationship, also called recursive, is one in which a relationship exists between occurrences of the same entity set. In this relationship, the primary and foreign keys are the same, but they represent two entities with different roles. See Figure 8.9 for an example. For some entities in a unary relationship, a separate column can be created that refers to the primary key of the same entity set.

Add an attribute named with which represents the other identity of the entity into the Relation, and makes this attribute ***Foreign Key*** refer to the **Primary Key** in the same Relation.

For a 1:M with or without optionality at the "many" end of the relationship, include a **foreign key** attribute to hold the primary key value of the one end record, i.e. the employee's manager's Employee_No. This would be read as "an employee manages zero, one or many employees and an employee is managed by one and only one employee".



EMPLOYEE (**Employee_No**, emp_name)

becomes

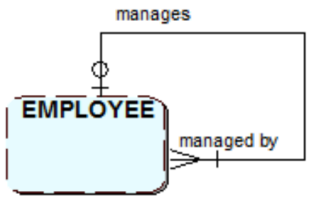
EMPLOYEE (**Employee_No**, emp_name, ***mgr_emp_no***)

3.9.12.Recursive (1: M) with optionality at one side relationship

It needs to be created one more Relation named by the new identity of the entity. Copy the name of the PK from the original Relation, as the name of the attribute in the new Relation, and set this attribute the FK refer to the original Relation's PK, then set this attribute the PK

of its Relation. Create another new attribute named by the additional identity of the entity, make this new attribute an FK refer to the PK of the original Relation as.

For a 1:M with optionality at the one end, a new relation is needed to avoid holding a null foreign key value. In this example not all employees have a manager, so a manager relation is needed which will hold the employee's Employee_No (Primary key) and their manager's emp_no (Foreign key), for each employee who has a manager.



EMPLOYEE (**Employee_No**, emp_name)

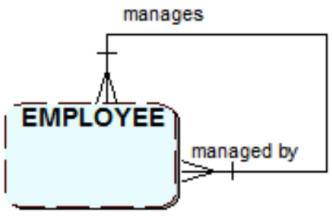
and

MANAGER (***Employee_No, mgr_emp_no***)

3.9.13.Recursive (M: M) without Mandatory at each side relationship

Like the previous situation, it needs to be created the new Relation with both attributes refer to the PK of original Relation (that means two FKs), but it needs to be set the PK with combining both attributes together.

Where there is a M:N relationship a new relation is used to hold the employee's Employer_No and their manager's emp_no. Both foreign keys are used to form a compound primary key



EMPLOYEE (**Employee_No**, emp_name,)

and

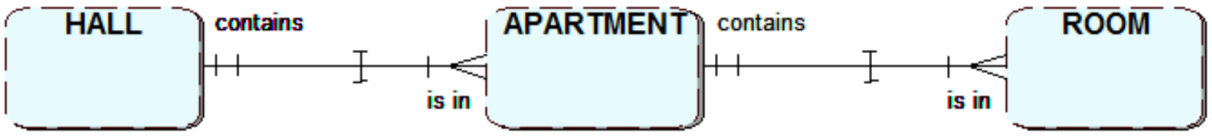
MANAGER (***Employee_No, mgr_emp_no***)

3.10. Mark the identifying relationship

An **identifying relationship** is a **relationship** between two entities in which an instance of a child entity is **identified** through its association with a parent entity, which means the child entity is dependent on the parent entity for its identity and cannot exist without it.

To manage the identifying Relationship among the Relations, always refer to the PKs of parental entities, and combine these FKs into the PK of child Relation.

The following shows an example for a student hall of residence where a number of halls exist, each have a number of apartments and within each apartment there are a number of student rooms. In this situation the apartment entity is dependent on the hall, as each apartment would be identified by a hall name and the apartment number together. Likewise, the room entity would also be dependent as its identifier would be made up of the primary key from the apartment entity and the room number. A large I symbol is shown on the relationship to indicate an identifying relationship as follows:



Identifying dependency relationships

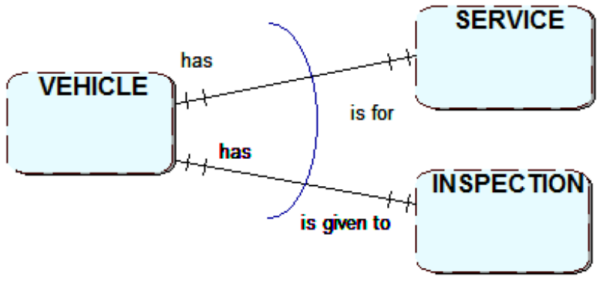
These are the relations showing the keys:

HALL (**Hall_name**)
APARTMENT (***Hall_name,*** **Apartment_no)**
ROOM (***Hall_name, Apartment_no,*** **Room_no**)

3.11. Resolving Exclusive Relationships

The following example is not perfect for resolving exclusive relationship, it needs the application takes action to check if there is the same vehicle registration number existed in both SERVICE and INSPECTION Relations. But from the data-store's perspective, it is right.

In some situations you may wish to show that a relationship is mutually exclusive. This is shown using an exclusive arc facing towards what would in effect be the optional entities. In this example a vehicle may receive a service or an inspection, but not both.



This results in the following relations, which avoids the null foreign key problem.

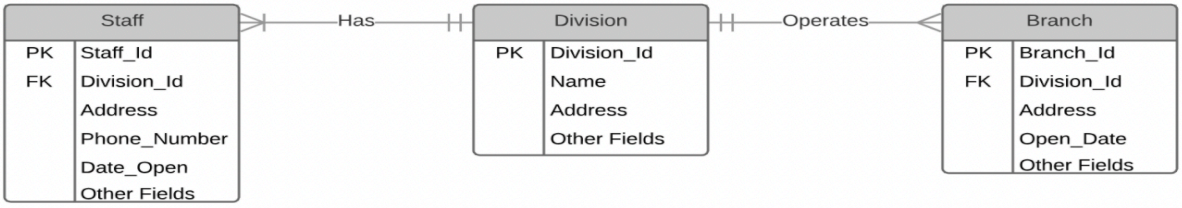
VEHICLE (**Registration_no**)
SERVICE (**Service_no**, *Registration_no*)
INSPECTION (**Inspection_no**, *Registration_no*)

3.12. Resolving Modeling Problem

3.12.1. Resolving Fan Trap

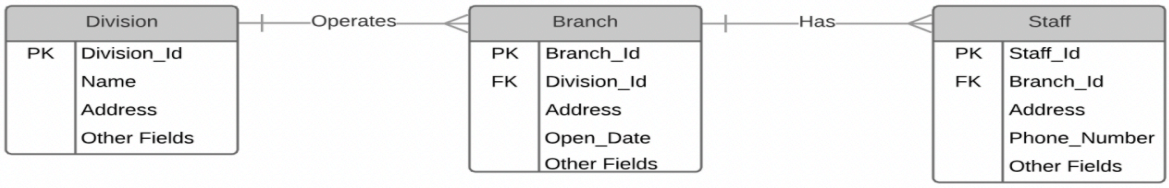
Fan trap occur in a situation when a model represents relationship between entity types however a path between certain entity occurrences is **ambiguous**.

Example: (Staff)-1:N-has-1:1-(Division)-1:1-operates-1:N-(Branch)



in this model it may be impossible to determine the branch a staff belongs to, in the situation when staff belong to division having more than 1 branches.

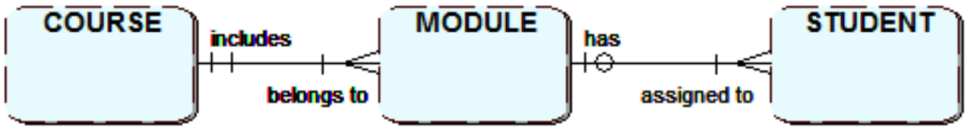
Restructuring the model resolves trap (Division)-1:1-operates-1:N-(Branch)-1:1-has-1:N-(Staff)



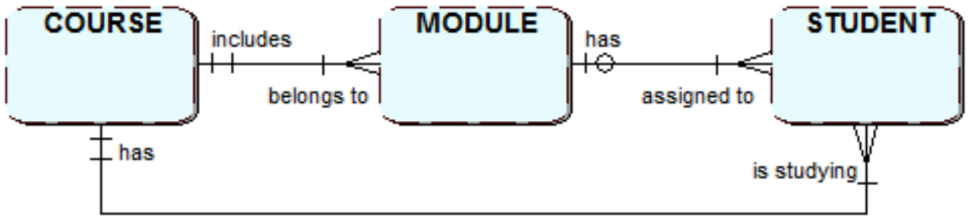
3.12.2. Resolving Chasm Trap

A chasm trap is created when relationships between entity types indicate a route linking them, but due to optionality it is not possible to make the required connection for all occurrences.

For example, suppose you want to identify which course a student is taking. The following model will not work if the student has not been assigned to a module. Although there is a link between course and module due to the Course_ID foreign key in the module relation, there would be no link between student and module if the student was not taking a module – there would be no foreign key Module_ID in student to provide a link to the module relation.



This problem can be resolved by adding a 1:M relationship linking course to student directly.



4. Normalization

At the logical design stage relations can be in one of four states, depending on whether certain conditions are satisfied.

These states are:

- **Un-normalised (UNF),**
- **1st Normal Form (1NF),**
- **2nd Normal Form (2NF),**
- **3rd Normal Form (3NF).**

Each state imposes further conditions on those required by the previous one.

The process of normalisation is used to produce a logical design which will lead to an efficient and effective database.

In particular, the aim is to avoid in the database:

- unnecessary wastage of storage space
- data redundancy (or data duplication – data (attributes) held in more than one place).

It is important to avoid data redundancy – storing the same data in more than one place has the potential to produce anomalies that can arise when data is inserted, amended or deleted. This is to be avoided in order not to compromise the integrity of the system. For example, if a customer's details were stored in many different places and they were not all updated at the same time this could cause inconsistencies which would be likely to result in out of date information being used.

If the ERD has been produced correctly, the resulting logical design will already be normalised to some extent. In this case, the normalisation process is used to check the definitions of the relations and, if necessary, refine them.

If the database is designed using a bottom-up approach where existing system documents, e.g. forms and reports have been considered, the initial logical design is unlikely to be normalised and applying the normalisation process is likely to cause changes to the original set of relations.

Normalisation means transforming the relations by stages into:

a) First Normal Form – to ensure that an implemented table would have only a single value for each attribute (there are no repeating groups).
To achieve 1NF: Remove the repeating group completely into a new table and place a link (Foreign Key) in the NEW table. Remember to identify the compound Primary Key in the new table.

b) Second Normal Form – the non-key attributes depend on the whole (compound) key, not just part of it.
To achieve 2NF: Remove the attributes that are dependent on only one part of the key with that part key into a new table, leaving a link (Foreign Key) behind in the ORIGINAL table.

c) Third Normal Form – all the attributes are dependent on the key.
To achieve 3NF: Remove any attributes which are dependent on another non-key attribute and place them in a new table leaving a link (Foreign Key) behind in the ORIGINAL table.

Do not normalise all the attributes as one data set, normalise each logical data set separately.

To help you remember the 3 normal forms, the attributes in a relation should depend on:

1NF the key
2NF the whole key
3NF nothing but the key

When you have normalised all the individual data sets, you should check to see if any relations identified are not shown as entities on the original ERD. If they are not, it is advisable to add them to form a complete (composite) ERD showing all of the data requirements.

4.1. Un-normalized

An un-normalised table can be defined as having any of the following issues:

- there are repeating groups (of attribute names)
- the attribute values are not atomic (single)
- there are "embedded tables"

To start the process you have to identify whether the data is in un-normalised form. The following is a

4.2. First Normal Form

First Normal Form

A relation is in First Normal Form if all attributes are functionally dependent on the primary key. In other words, for each value of the primary key there is only one value for each attribute in the relation. Or, put simply, if you entered a Student_ID to search the database you would expect it to return just one set of student attributes.

To convert a table to First Normal Form, multiple values or repeating groups of attributes must be removed to form another relation. To do this the following steps are carried out:

1. Identify the primary key for the relation (you may have to invent one)
2. Identify the repeating group or groups
3. Remove the repeating group completely from the original relation and place it in a newly- created relation
4. Ensure the relations are linked by putting the primary key of the original relation into the new one as a foreign key
5. Define a primary key for the new relation. This usually consists of two attributes, one of which is the Foreign Key, but there could be more than two.

Un-normalised form	**First normal form**
Invoice_No	**Invoice_No**
Date	Date
Customer_No	Customer_No
Invoice_Name	Invoice_Name
Invoice_Address	Invoice_Address
(Item_ID	Subtotal
Description	Tax
Qty	Delivery
Price	Total
Amount)	
Subtotal	***Invoice_No***
Tax	**Item_ID**
Delivery	Description
Total	Qty
	Price
	Amount

4.3. Second Normal Form

Second Normal Form
A relation is in Second Normal Form if –

- It is already in 1NF and
- All non-key attributes are fully functionally dependent on the whole key.

Conversion from 1NF to 2NF

This conversion **only** applies to relations that have a **compound key**. i.e. any relation with a simple (single attribute) key is already in 2NF, if it is already in 1NF.

The process involves checking whether or not there are attributes that depend on only **one part** of the compound key. To do this, follow the steps below:

1. Identify attributes with partial key dependencies
2. Remove the attributes with partial key dependencies into a new relation
3. Make the part key they are dependent on the primary key for the new relation

Do not forget to ensure that the original relation retains its compound key.

Un-normalised form	First Normal Form	Second Normal Form
Invoice_No	**Invoice_No**	**Invoice_No**
Date	Date	Date
Customer_No	Customer_No	Customer_No
Invoice_Name	Invoice_Name	Invoice_Name
Invoice_Address	Invoice_Address	Invoice_Address
(Item_ID	Subtotal	Subtotal
Description	Tax	Tax
Qty	Delivery	Delivery
Price	Total	Total
Amount)		
Subtotal	***Invoice_No***	***Invoice_No***
Tax	**Item_ID**	***Item_ID***
Delivery	Description	Qty
Total	Qty	Amount
	Price	
	Amount	**Item_ID**
		Description
		Price

4.4. Third Normal Form

Third Normal Form

A relation is in Third Normal Form if -

- it is already in 2NF
- there are no functional dependencies between any pair of non-key attributes (i.e. there are no transitive dependencies)

If an attribute is functionally dependent on another attribute, that one is referred to as the **determinant**. For example:

Is Invoice No. a determinant of Invoice Name? Yes
Is Invoice Name a determinant of Invoice_No.? No

If an attribute is the determinant of a second attribute and that attribute is the determinant of a third attribute then the third attribute is **transitively** dependent on the first attribute.

Converting the data tables to the third normal form will further improve the logical design of the database.

Conversion to 3NF

This is a very similar process to the one for 2NF. To do this, follow the steps below:

1. Identify any attributes that are determined by another non-key attribute
2. Remove these attributes to a new relation
3. Set the non-key attribute to be the Primary key in the new relation
4. Convert the non-key attribute to a foreign key in the original relation.

The Third Normal Form is very similar to the Second Normal Form. However, instead of considering whether some attributes in the table are dependent on only part of a compound key, a check is made to see whether they are dependent on attributes which are **not** part of the key. In a very similar way to 2NF, you remove the attributes which depend on this (non-key) attribute into a new table.

Un-Normalised Form UNF	First Normal Form 1NF	Second Normal Form 2NF	Third Normal Form 3NF	Relation Name
Invoice_No	**Invoice_No**	**Invoice_No**	**Invoice_No**	**INVOICE**
Date	Date	Date	Date	
Customer_No	Customer_No	Customer_No	Customer_No	
Invoice_Name	Invoice_Name	Invoice_Name	***Invoice_Name***	
Invoice_Address	Invoice_Address	Invoice_Address	Subtotal	
(Item_ID	Subtotal	Subtotal	Tax	
Qty	Tax	Tax	Delivery	
Description	Delivery	Delivery	Total	
Price	Total	Total		
Amount)			**Invoice_Name**	**INVOICE ADDRESS**
Subtotal	***Invoice_No***	***Invoice_No***	Invoice_Address	
Tax	**Item_ID**	***Item_ID***		
Delivery	Qty	Qty	***Invoice_No***	**INVOICE ITEM**
Total	Description	Amount	***Item_ID***	
	Price		Qty	
	Amount	**Item_ID**	Amount	
		Description		
		Price	**Item_ID**	**ITEM**
			Description	
			Price	

4.5. Denormalization

Although you should always aim to implement a normalised set of tables where possible, there are occasions when you might consider denormalising, usually for performance reasons. In the following tables, if the Invoice Item's tax amount was a calculated value, the item's tax code would be used to search the Tax table for the appropriate row, and the corresponding tax % value would then be used to calculate the invoice item tax amount.

Item ID	Description	Quantity	Price	Total Price	Tax Code	Tax Amount
A101	Laptop	1	450	450	01	90

Invoice Item Table

Tax Code	Tax %
01	20

Tax Table

Denormalisation could be applied to speed up the calculation. The tax % amount could be stored within the Invoice Item table, thus saving a Tax table read, however the tax % value would now have to be maintained in both tables.

In this example there is another reason to consider denormalisation. Storing the actual tax % amount in the Invoice Item table would ensure that if the calculation had to be performed again in the future, when the tax rate in the Tax table may have changed, the result would still produce the same value which would have been calculated at the time the invoice item was initially created as it should.