

Entity-Relationship Model : Basic concepts, Design Issues, Mapping Constraints, Keys, Entity-Relationship Diagram, Weak Entity Sets, Extended E-R features.

Relational Database Design using ER Model

- Data is the main raw material of any system (computerized or manual). So how an organization should **collect**, **update**, and **store data** is an important step in designing any system.
- **Data modeling** is a set of tools and techniques to **understand** and **analyze** this step properly. It is often the first step in database design and object-oriented programming.

The outcome of data modeling is a **data model** which **describing**

- the data **objects/entities** – **names**, **types** and **constraints** that should hold for each data element in the entities) and
- their **relationships**.

A complete model is a mechanism to answer any query. Traditionally, data models have been built during the **analysis and design phases** of a project to ensure that the requirements for a new application are fully understood.

The **entity–relationship (E-R) model** is a high-level data model. It is based on collection of basic objects exist in real world, called entities, and of relationships among these objects.

- It starts **with a set of requirements**. Analyst intuitively identifying objects of importance about which a system is to store data (**entities**) from the requirement and then identifying the attributes which describe **each of the entities and the relationships** between the entities.
- Requires **detailed understanding of the system**.

In data modeling phase of any system design we generally create **Entity Relation Diagram**. It is a graphical representation for understanding and organizing the data independent of the actual database implementation. The ER model defines the **conceptual view** of a database.

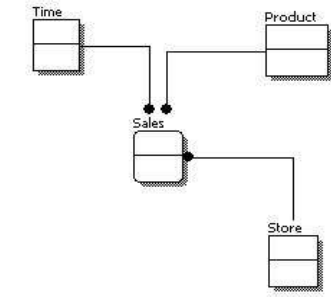
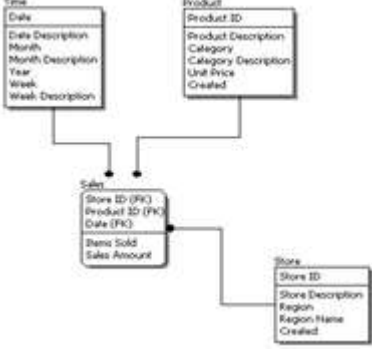
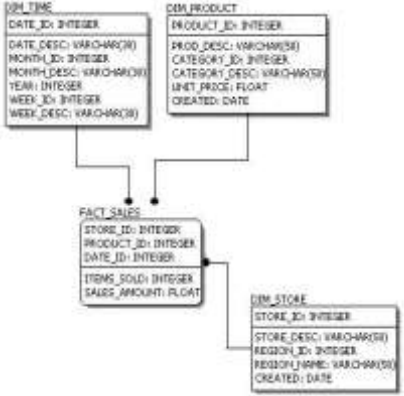
The **ER Model** was Proposed by **Peter Chen** (1976) and widely used to designing a database.

There is no standard for representing data objects in ER diagrams. Each modelling methodology uses its own notation and various notations are in place. Some of the widely used notations are :

- Information Engineering notations
- Chen notation
- Bachman notation
- Martin notation

As data model creation is a **complex** process, it is generally divided into **multiple steps**. During analysis phase we generally create **conceptual** and **logical** data model. During the design phase we generally create the **physical** data model through **successive refinement** of the model created in analysis phase and then create the actual database based on this model.

All the steps are briefly described below. Sometimes some of the steps may be merged.

<p>Conceptual Data Model preparation</p> <p>Model is not invalidated, if a different DBMS is used later on.</p>	<p>Identifies highest-level relationships between the different entities (business entities).</p> <ul style="list-style-type: none"> • Done in requirement analysis step • Only identify entities and relationship • No attribute/ primary key specified • Generally ER Model is used • Independent of specific DBMS features. 	
<p>Logical Data Model preparation</p>	<p>Describes the data in as much detail as possible, without regard to how they will be physical implemented in the database.</p> <ul style="list-style-type: none"> • It is refinement of conceptual model • Attribute, primary key, foreign key specified • Normalization occurs at this level. • Further enhancement of earlier ER model • The model is not invalidated, if a different DBMS is used later on. 	
<p>Physical Data Model preparation</p>	<p>Describes all table structures, including column name, data type, constraints, primary key, foreign key, and relationships between tables.</p> <p>At this we also design indexes, table distribution, buffer size, etc., to maximize performance of the final system, considering expected work load.</p> <p>We also address security issues and enforce appropriate access control.</p> <ul style="list-style-type: none"> • It is specific to RDBMS used. • ER diagram may be used. • DBA use it for creating actual RDB 	

The table below compares the different features of above models :

Feature	Conceptual	Logical	Physical
Entity Names	√	√	
Entity Relationships	√	√	
Attributes		√	
Primary Keys		√	√

We can see that the complexity increases from conceptual to logical to physical.


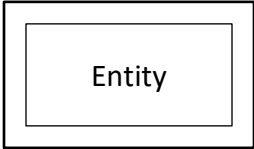

Foreign Keys		√	√
Table Names			√
Column Names			√
Column Data Types			√
Constraints			√

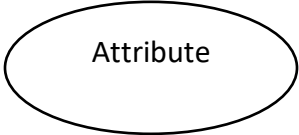
The **Database Design process** can be carried out in the following sequence:

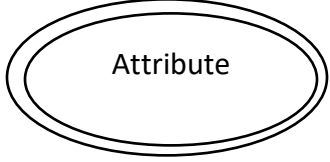

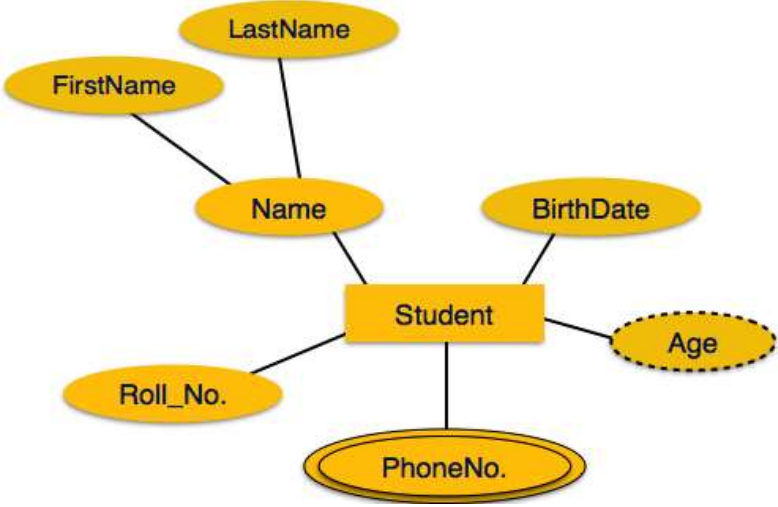
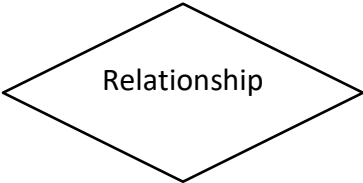
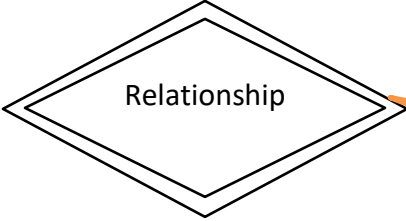
- Modelling the data using an **Entity-Relationship Diagram**
- Mapping the Entity-Relationship Diagram to a **relational database design**
- **Normalizing** the **relational database design** to a non-redundant **relational database design**.

Some Useful Terms

We need to be familiar with the following terms to **draw ER diagram**. The terms and notations used for the respective terms are discussed in following table.

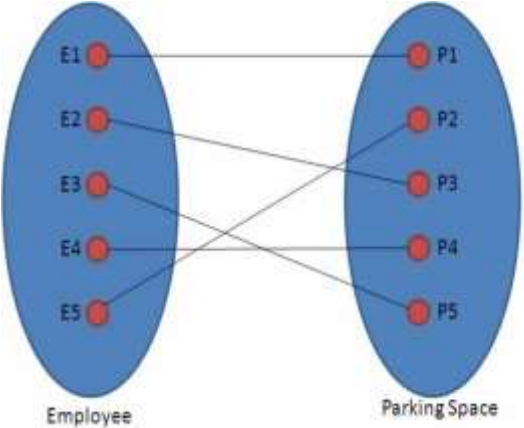
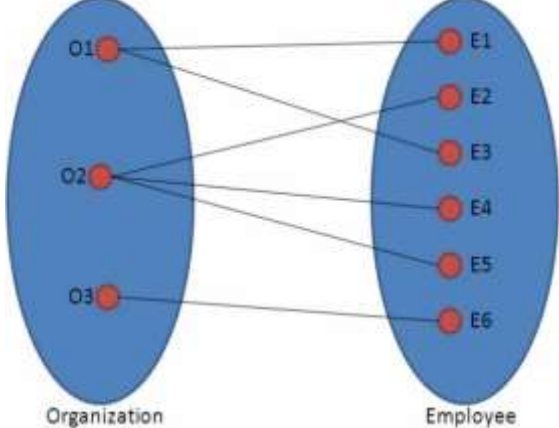
<p>Entity</p> 	<ul style="list-style-type: none"> • An entity can be a real-world object (living/nonliving), that can be easily identifiable. For example, in a school database, students, teachers, classes, and courses offered can be considered as entities. All these entities have some attributes or properties that give them their identity. • We will collect data about these identified entities of the system. It is also known as entity type. • An entity set is a collection of similar types of entities. For example, a Students set may contain all the students of a school; likewise a Teachers set may contain all the teachers of a school from all faculties.
<p>Weak entity</p>  	<p>A weak entity is an entity that depends on the existence of another entity i.e. that cannot be identified by its own attributes. It uses a foreign key combined with its attributed to form the primary key. An entity like order item is a good example for this. The order item will be meaningless without an order so it depends on the existence of order.</p> <p>In a parent/child relationship, a parent is considered as a strong entity and the child is a weak entity.</p> <p>Let us consider another scenario, where we want to store the information of employees and their dependents. The every employee may have zero to n number of dependents. Every dependent has an Depid and DepName.</p>

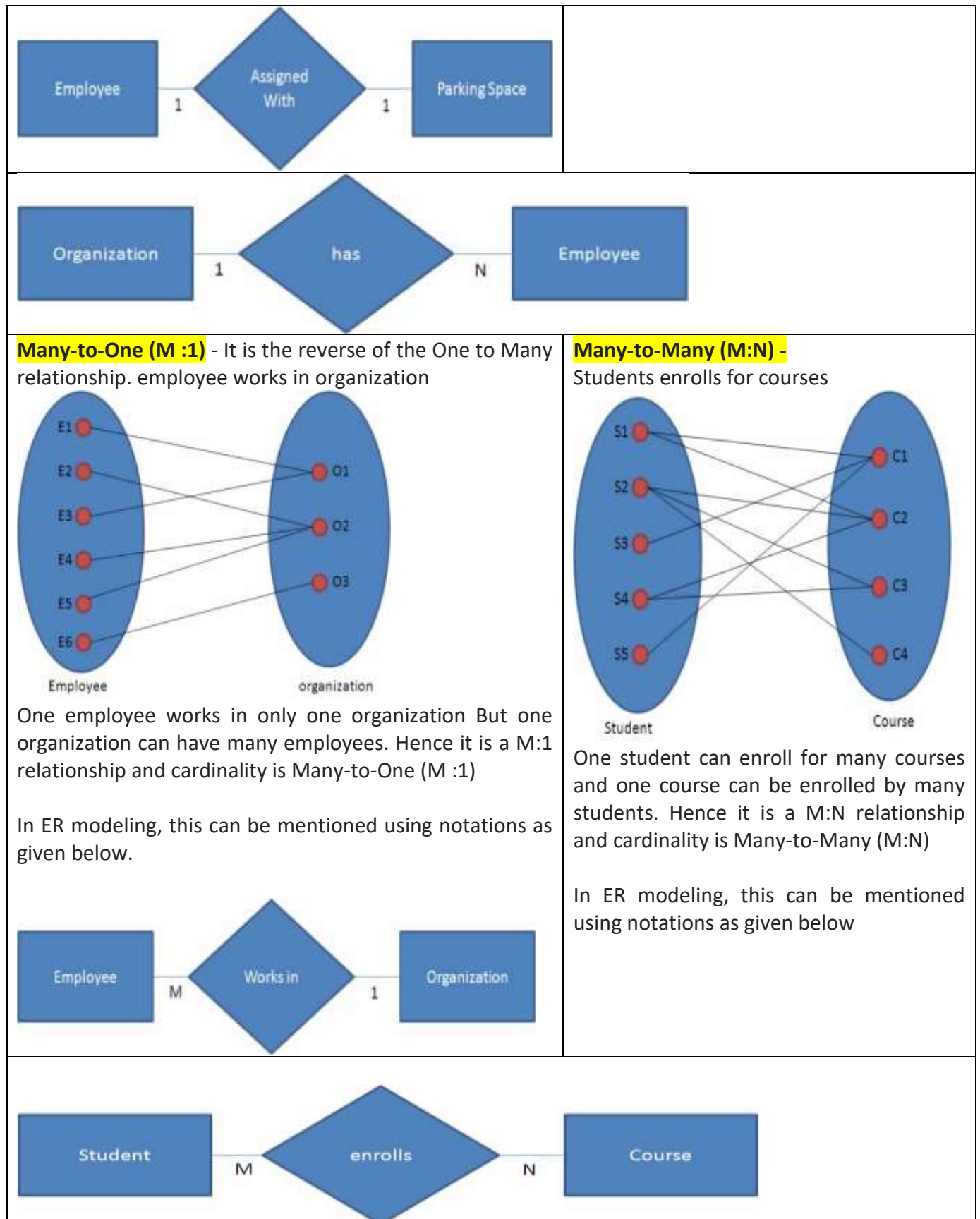
<table><tr><th>EmpId</th><th>EmpName</th></tr><tr><td>123</td><td>Susil Pal</td></tr><tr><td>456</td><td>Bupen Hazarika</td></tr><tr><td>890</td><td>Papu Jadav</td></tr></table> <table><tr><th>EmpId</th><th>DepId</th><th>DepName</th></tr><tr><td>123</td><td>1</td><td>Rahat</td></tr><tr><td>123</td><td>2</td><td>Chahat</td></tr><tr><td>456</td><td>1</td><td>Raju</td></tr><tr><td>456</td><td>2</td><td>Ruhi</td></tr><tr><td>456</td><td>3</td><td>Raja</td></tr></table>	EmpId	EmpName	123	Susil Pal	456	Bupen Hazarika	890	Papu Jadav	EmpId	DepId	DepName	123	1	Rahat	123	2	Chahat	456	1	Raju	456	2	Ruhi	456	3	Raja	<p>Now let us consider the data in these two table as shown in figure. Employee 123 has two dependent, Employee 456 has three dependents and Employee 890 has no dependent.</p> <p>Now, in case of Dependent entity Depid cannot act as primary key because it is not unique. No dependent can exist without the employees. In Employee table EmpId is primary key and in dependent table EmpId + DepId is primary key. As DepId in addition to EmpId is used to identify the dependent the column DepId is known as discriminator.</p>
EmpId	EmpName																										
123	Susil Pal																										
456	Bupen Hazarika																										
890	Papu Jadav																										
EmpId	DepId	DepName																									
123	1	Rahat																									
123	2	Chahat																									
456	1	Raju																									
456	2	Ruhi																									
456	3	Raja																									
<h3>Attributes</h3> <div></div> <p>Different notations are used to represent different attributes as shown below.</p>	<p>Properties/characteristics which describe entities or relationship are called attributes.</p> <ul style="list-style-type: none">The set of possible values that an attribute can take is called the domain of the attribute. For example, the attribute day may take any value from the set {Monday, Tuesday ... Friday}.All entities in a given entity set have same attributes. <p>Simple attribute : If an attribute cannot be divided into simpler components. Example : employee_id of an employee.</p> <p>Composite attribute : If an attribute can be split into components, it is called a composite attribute. Example : Name of the employee which can be split into First_name, Middle_name, and Last_name.</p> <p>Single valued Attributes : Can take only a single value for each entity instance, it is a single valued attribute. Example : age of a student.</p>																										
<h3>Key attribute</h3>	<ul style="list-style-type: none">The attribute (or combination of attributes) which is unique for every entity instance is called key attribute. For each entity set we choose a key. There could be more than one candidate key, if so, we designate one of them as the primary key.<ul style="list-style-type: none">Super Key – A set of attributes (one or more) that collectively identifies an entity in an entity set.Candidate Key – A minimal super key is called a candidate key. An entity set may have more than one candidate key.Primary Key – A primary key is one of the candidate keys chosen by the database designer to uniquely identify the entity set.																										

<p>Multi-valued Attributes</p> 	<p>An attribute can take more than one value for each entity instance.</p> <p>Example : telephone number/ email id of an employee, a particular employee may have multiple telephone numbers or email id. So in actual table design it will may to more than one column or design separate table.</p>
<p>Derived Attribute</p> 	<p>An attribute which can be calculated or derived based on other attributes is a derived attribute. Example : age of employee which can be calculated from date of birth and current date.</p> <p>Example with different types of attributes:</p> 
<p>Relationships</p>  	<ul style="list-style-type: none"> • Association between entities are called relationship and is represented by diamond-shaped box. • Name of the relationship is written inside the diamond-box. All the entities (rectangles) participating in a relationship, are connected to it by a line. <p>Example : An employee works for an organization. Here "works for" is a relation between the entities employee and organization.</p> <p>However in ER Modeling, To connect a weak Entity with others, you should use a weak relationship notation as shown.</p> <div data-bbox="685 1566 1133 1638" style="border: 1px solid orange; padding: 5px; margin: 10px auto; width: fit-content;"> <p>To be used for weak entity</p> </div> <p>Degree of a Relationship</p> <p>Degree of a relationship is the number of entity types involved. The n-ary relationship is the general form for degree n. Special cases are unary, binary, and ternary , where the degree is 1, 2, and 3, respectively.</p> <p>Example for unary relationship : An employee is a manager of</p>

	<p>another employee (only one relation employee involved)</p> <p>Example for binary relationship : An employee works-for department.</p> <p>Example for ternary relationship : customer purchase item from a shop keeper (three relations involved)</p>
Cardinality of a Relationship	<p>Relationship cardinalities specify how many of each entity type is allowed. Four possible connectivity are as follows:</p> <ol style="list-style-type: none"> 1. One to one (1:1) relationship 2. One to many (1:N) relationship 3. Many to one (M:1) relationship 4. Many to many (M:N) relationship <p>The minimum and maximum values of this connectivity is called the cardinality of the relationship</p>

Example for Cardinality

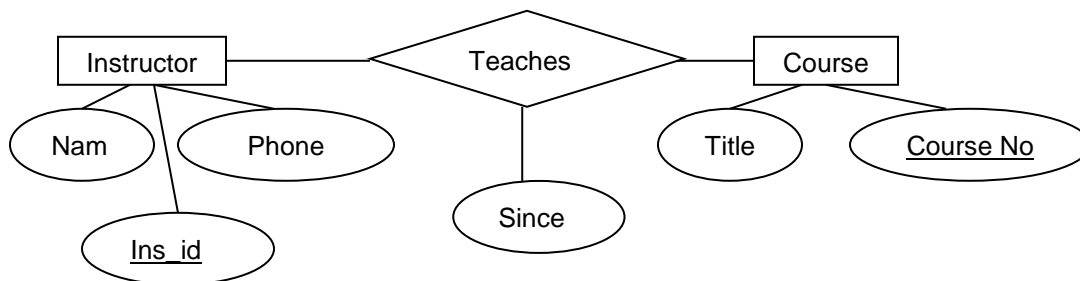
<p>One-to-One (1:1) - Employee is assigned with a parking space.</p>  <p>One employee is assigned with only one parking space and one parking space is assigned to only one employee. Hence it is a 1:1 relationship and cardinality is One-To-One (1:1)</p> <p>In ER modeling, this can be mentioned using notations as given below</p>	<p>One-to-Many (1:N) - Organization has employees</p>  <p>One organization can have many employees, but one employee works in only one organization. Hence it is a 1:N relationship and cardinality is One-To-Many (1:N)</p> <p>In ER modeling, this can be mentioned using notations as given below</p>
--	---



Relationship with descriptive attributes

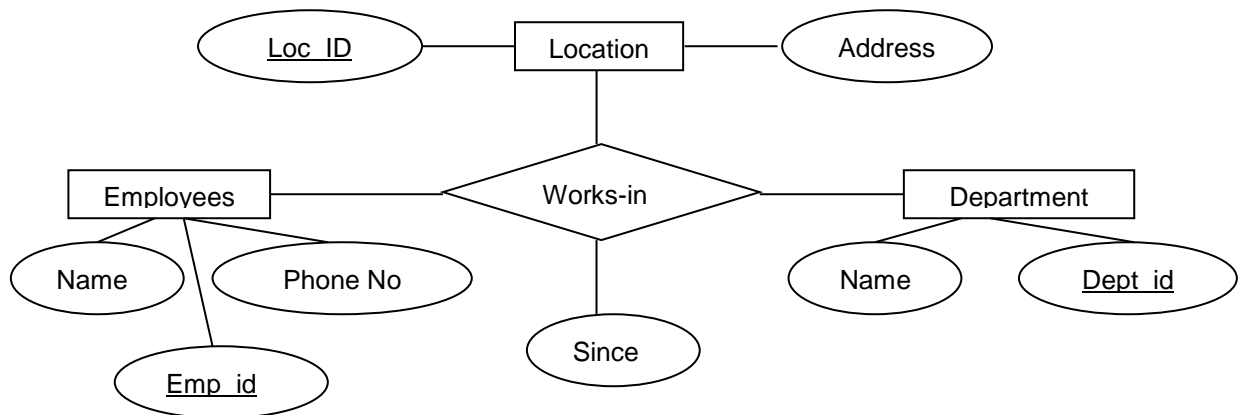
A relationship can also have **descriptive attributes**.

In the following ER diagram : Instructors teach courses. Instructors have a name, phone number and Instructor ID (primary key). Courses are numbered and have titles. The relationship Teaches must be uniquely identified by the participating entities (Ins_id + Course no), without referencing the descriptive attribute. Thus for a given instructor-course pair, we cannot have more than one associated **since** value.



Example of Ternary Relationship

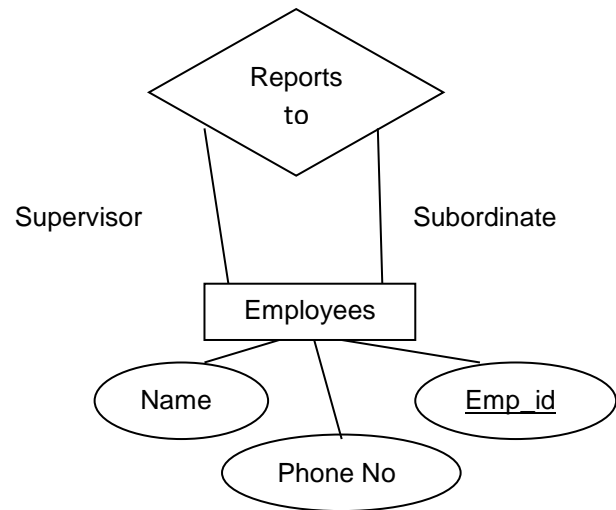
Suppose that each department has offices in several locations and we want to record the locations at which each employee works. This is an example of **ternary** relationship because we must record an association between an employee, a department, and a location.



Example of Role Indicator

Here two entities from the same entity set are involved in the relationship, since employees report to other employees. However they will play different role, which is reflected in the **role indicator** supervisor and subordinate.

In relationship set of the relation Report_to, the attribute Emp_id of both the entities will be involved by changing the attribute name suitably (generally role indicator is concatenated e.g. Supervisor_Emp_id and Subordinate_Emp_id)



Constraints

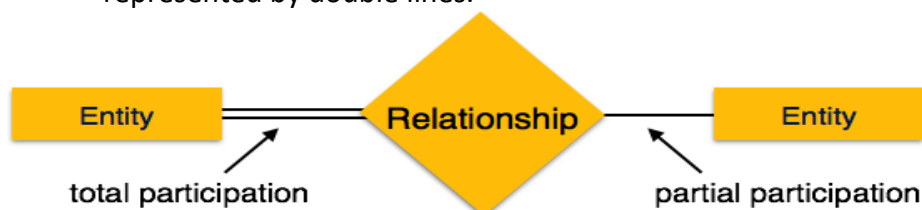
We may define **certain constraints** to which the contents of a database must conform. These constraints may be of different types :

1. **Mapping Cardinality constraint** : **Mapping cardinalities, or cardinality ratios**, express the number of entities to which another entity can be associated via a relationship set. Here we shall concentrate on only binary relationship sets. For a binary relationship set R between entity sets A and B, the mapping cardinality must be one of the following:
One to one , **One to many** , **Many to one** , **Many to many**. (Examples shown above)

The appropriate mapping cardinality for a particular relationship set obviously depends on the real-world situation that the relationship set is modeling.

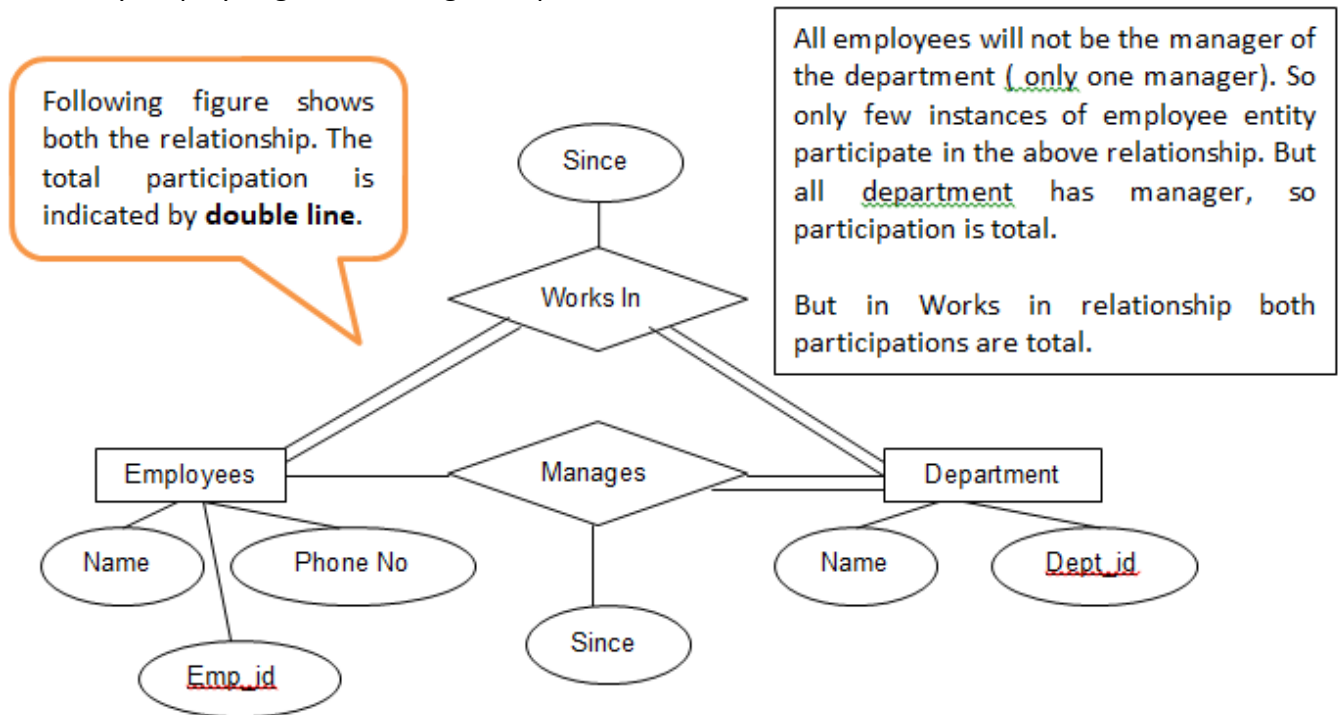
2. **Participation Constraints** : It is of two type.

- **Total Participation** – Each entity is involved in the relationship. Total participation is represented by double lines.



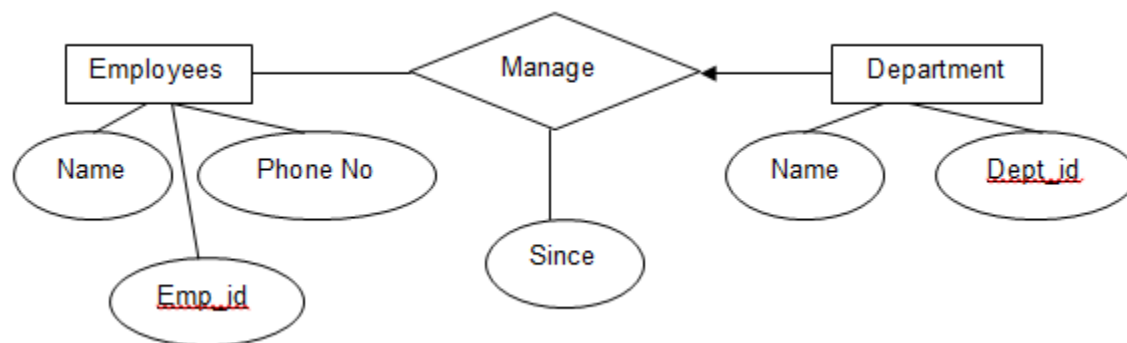
- **Partial participation** – Not all entities are involved in the relationship. Partial participation is represented by single lines.

Say every department is required to have a manager. In this case all the department entities in department set must appear in the **manages** relationship. This is an example of **participation constraint** and the participation of department entity set is said to be **total**. A participation that is not total is said to be **partial**. Example : participation of Employees entity set in manages is partial, since not every employee gets to manage a department.



3. Key Constraints

A key allows us to identify a set of attributes that suffice to distinguish entities from each other. Keys also help uniquely identify relationships, and thus distinguish relationships from each other. We have already discussed different types of key attributes.



<p>Employees manages departments such that</p> <ol style="list-style-type: none">Each department has at most one manager – It is an example of key constraint. It is indicated by arrow. It indicates that we can uniquely determine the managers, if we know the department id from the relationship set.Single employee is allowed to manage more than one departments. <p>A relationship like this is called one-to-many relationship.</p>	<table><thead><tr><th><u>Dept Id</u></th><th><u>Emp Id</u></th><th><u>Since</u></th></tr></thead><tbody><tr><td>001</td><td>501</td><td>3/4/09</td></tr><tr><td>002</td><td>504</td><td>9/5/08</td></tr><tr><td>003</td><td>501</td><td>10/2/10</td></tr></tbody></table> <p>If we impose the restriction – one employee is allowed to manage only one department (adding arrow from employee to manages) then it will be a one-to-one relationship.</p>	<u>Dept Id</u>	<u>Emp Id</u>	<u>Since</u>	001	501	3/4/09	002	504	9/5/08	003	501	10/2/10						
<u>Dept Id</u>	<u>Emp Id</u>	<u>Since</u>																	
001	501	3/4/09																	
002	504	9/5/08																	
003	501	10/2/10																	
<p>In contrast, in the Works In relationship set, an employee is allowed to work in several departments and a department is allowed to have several employees – it is an example of many-to-many relationship.</p>	<table><thead><tr><th><u>Dept Id</u></th><th><u>Emp Id</u></th><th><u>Since</u></th></tr></thead><tbody><tr><td>001</td><td>501</td><td>3/4/09</td></tr><tr><td>002</td><td>504</td><td>9/5/08</td></tr><tr><td>002</td><td>501</td><td>10/2/10</td></tr><tr><td>004</td><td>501</td><td>10/2/08</td></tr><tr><td>005</td><td>501</td><td>8/8/09</td></tr></tbody></table>	<u>Dept Id</u>	<u>Emp Id</u>	<u>Since</u>	001	501	3/4/09	002	504	9/5/08	002	501	10/2/10	004	501	10/2/08	005	501	8/8/09
<u>Dept Id</u>	<u>Emp Id</u>	<u>Since</u>																	
001	501	3/4/09																	
002	504	9/5/08																	
002	501	10/2/10																	
004	501	10/2/08																	
005	501	8/8/09																	

Design Issues

The notions of an entity set and a relationship set (set for defining relationship between entities) are not precise, and it is possible to define a set of entities and the relationships among them in a number of different ways. So some issues known as **design issues** may arise while designing an E-R. We are discussing some basic issues and relative solutions concerning them.

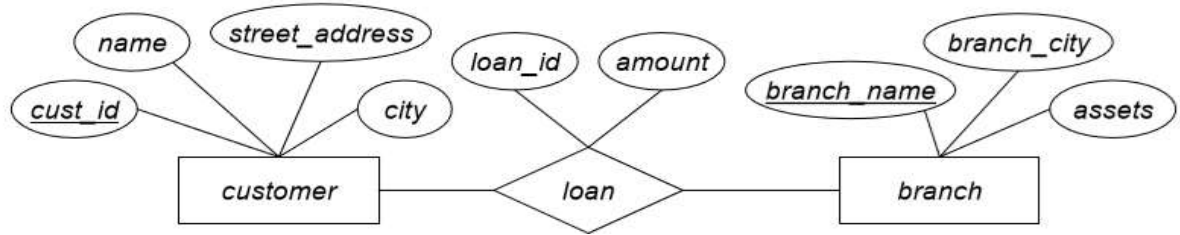
1. **Use of Entity Sets versus Attributes** : Sometimes designer needs to take a decision whether a data item will be consider as **attribute** or a **separate entity**.

Say employee has two attributes **employee-name** and **telephone-number**. But telephone can be considered as an entity in its own right with attributes **telephone-number** and **location** (the office where the telephone is located). So we may redefine the employee entity set by treating a telephone as an **entity**. It permits employees to have **several telephone numbers** (including zero) associated with them.

Thus, treating telephone as an entity is more general than treating it as an attribute and is appropriate when the generality may be useful.

2. **Use of Entity Sets versus Relationship Sets** : It is not always clear whether an object is best expressed by an **entity set** or a **relationship set**.

A bank loan can be modeled as an entity. Alternatively it can be used as a relationship between **customers** and **branches**, with loan-number and amount as descriptive attributes. Each loan is represented by a relationship between a customer and a branch as shown below.



It has no issue if every loan is held by exactly one customer and is associated with exactly one branch. However it cannot represent conveniently a situation in which several customers hold a loan jointly. To handle such a situation, we must define a separate relationship for each holder of the joint loan. Each such relationship must, of course, have the same value for the descriptive attributes loan-number and amount. Two problems arise

1. the data are stored multiple times, wasting storage space, and
2. updates potentially leave the data in an inconsistent state

But if we consider loan as a separate entity set we can resolve above issues.

Another example : There is a course-registration record for each course that each student takes. Then, we have an entity set to represent the course-registration record. Let us call that entity set registration. Each registration entity is related to exactly one student and to exactly one course, so we have two relationship sets, one to relate course registration records to students and one to relate course-registration records to course.

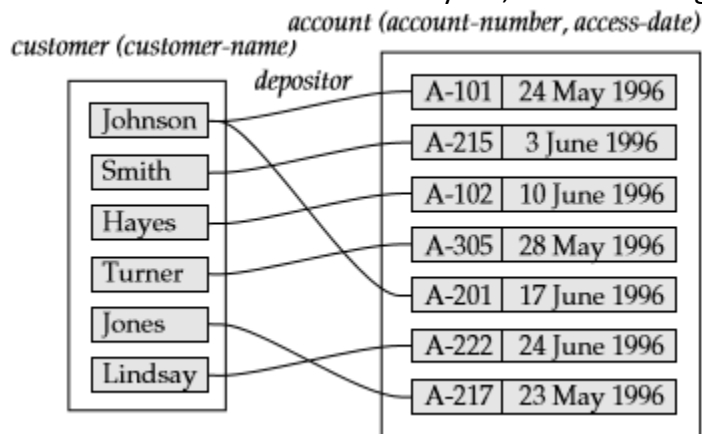
- Simple guideline for choosing entities or relationship
 - An entity is a thing
 - A relationship is an action involving entities.
- In general, evaluate schema against various scenarios
 - Watch out of for unnecessary redundancy or potential for consistency issues

3. **Binary versus n-ary Relationship Sets** : Relationships in databases are often binary. Some relationships that appear to be non-binary could actually be better represented by several binary relationships. For instance, one could create a ternary relationship parent, relating a child to his/her mother and father. However, such a relationship could also be represented by two binary relationships, mother and father separately. Using the two relationships mother and father allows us record a child's mother, even if we are not aware of the father's identity

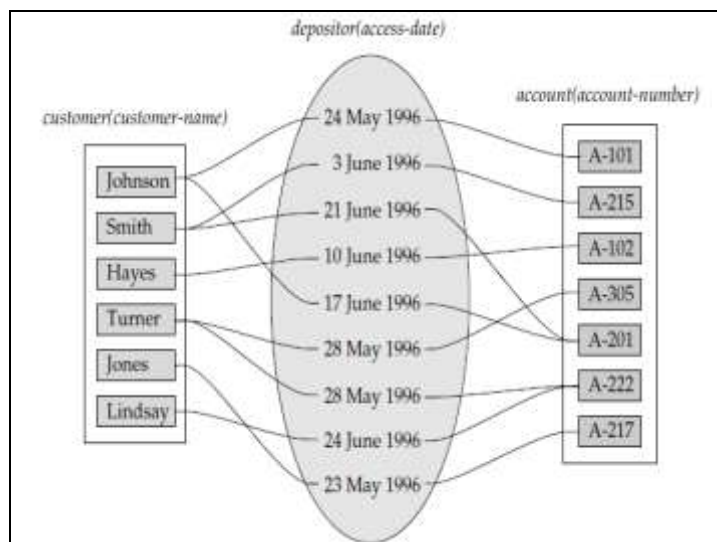
Conceptually, we can restrict the E-R model to include only binary relationship sets. However, this restriction is not always desirable.

4. **Placement of Relationship Attributes** : The cardinality ratio of a relationship can affect the placement of relationship attributes. Thus, attributes of one-to-one or one-to-many relationship sets can be associated with one of the participating entity sets, rather than with the relationship set. For instance, let us specify that customer is a one-to-many relationship set such that one

customer may have several accounts, but each account is held by only one customer. In this case, the attribute access-date, which specifies when the customer last accessed that account, could be associated with the account entity set, as shown in figure below.



- Attributes of a 1-M relationship can be repositioned to only the entity set on the “many” side of the relationship.
- For 1-1 relationship, the relationship attribute can be associated with either one of the participating entities.



The choice of attribute placement is more clear-cut for M-M relationship sets.

Let us specify the perhaps more realistic case that depositor is a many-to-many relationship set expressing that a customer may have one or more accounts, and that an account can be held by one or more customers as shown in figure.

In this case, access date must be an attribute of the depositor relationship set, rather than either one of the participating entities.

Extended E-R Features

Although the basic E-R concepts can model most database features, some aspects of a database may be more aptly expressed by certain extensions to the basic E-R model. We discuss the extended E-R features of specialization, generalization, higher- and lower-level entity sets, attribute inheritance, and aggregation.

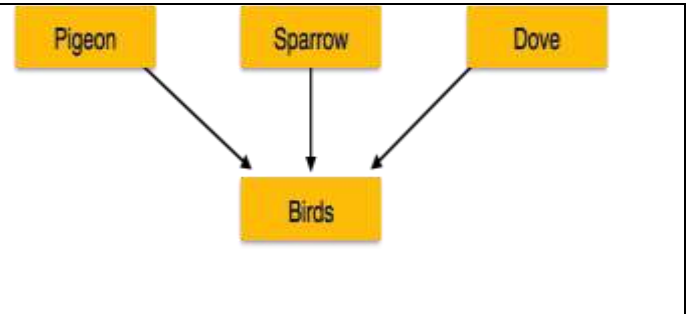
• Generalization & Specialization

The ER Model has the power of expressing database entities in a conceptual hierarchical manner. Going up in this structure is called **generalization**, where entities are clubbed together to represent a more generalized view. For example, a particular student named Mira can be generalized along with all the students. The entity shall be a student, and further, the student is a person. The reverse is

called **specialization** where a person is a student, and that student is Mira.

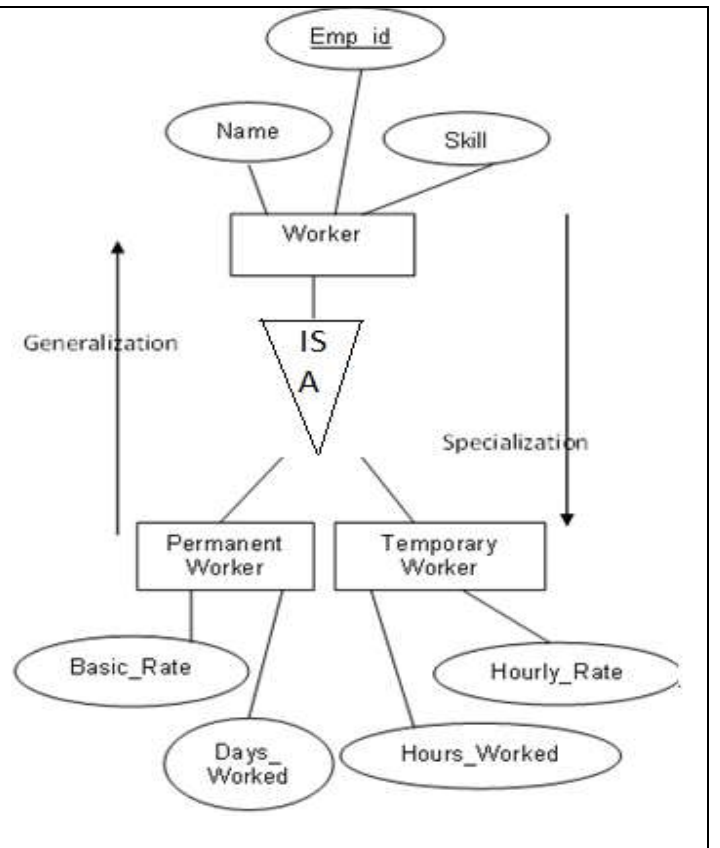
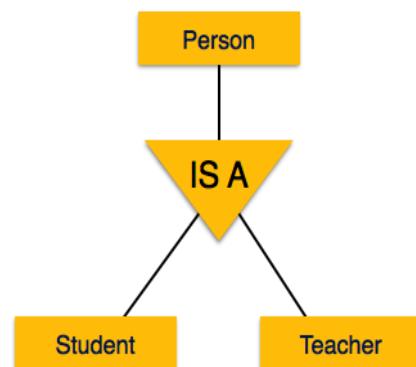
Generalization

In generalization, a number of entities are brought together into one generalized entity based on their **similar characteristics**. For example, pigeon, house sparrow, crow and dove can all be generalized as Birds.



Specialization

Specialization is the opposite of generalization. In specialization, a group of entities is divided into sub-groups based on their characteristics. In a school database, persons can be specialized as teacher, student, or a staff, based on what role they play in school as entities.



- **Attribute Inheritance**

A crucial property of the higher- and lower-level entities created by specialization and generalization is attribute inheritance. The attributes of the higher-level entity sets are said to be inherited by the lower-level entity sets. For example, student and teacher inherit the attributes of person. Thus, student is described by its name, age, and gender attributes, and additionally a roll no attribute.

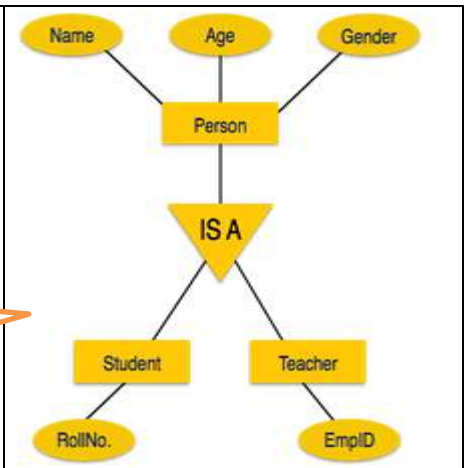
A lower-level entity set (or subclass) also **inherits participation in the relationship sets** in which its higher-level entity (or superclass) participates. For example the officer, teller, and secretary entity sets

can participate in the works-for relationship set, since the superclass employee participates in the works-for relationship.

We use all the above features of ER-Model in order to create classes of objects in object-oriented programming.

Inheritance is an important feature of Generalization and Specialization. It allows lower-level entities to inherit the attributes of higher-level entities.

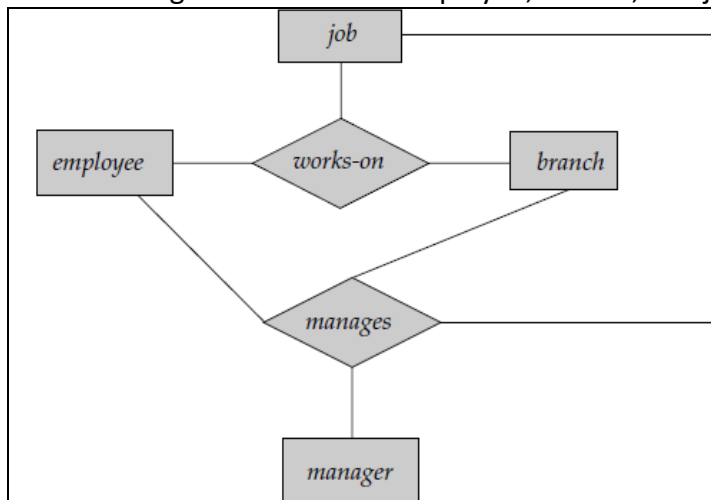
For example, the attributes of a Person class such as name, age, and gender can be inherited by lower-level entities such as Student or Teacher.



• Aggregation

One limitation of the E-R model is that it cannot express relationships among relationships.

To illustrate the need for such a construct, consider the following ternary relationship **works-on** shown in figure between an employee, branch, and job.



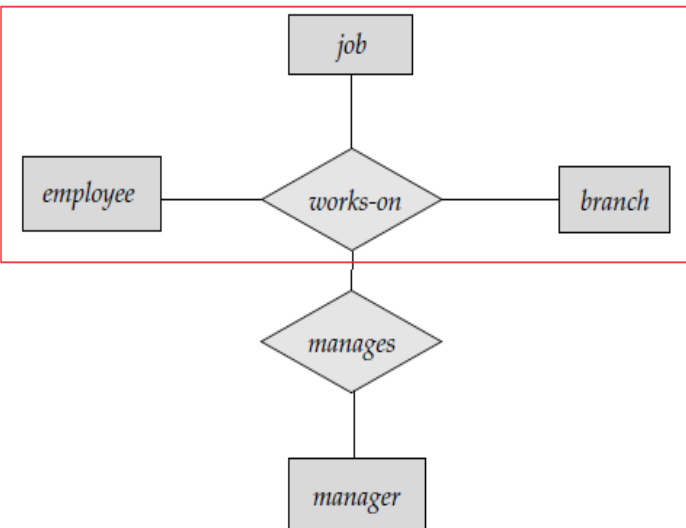
To represent this relationship we can create a **quaternary** relationship **manages** between employee, branch, job, and manager. (A quaternary relationship is required—a binary relationship between manager and employee would not permit us to represent which (branch, job) combinations of an employee are managed by which manager.) Using basic E-R modeling constructs, we obtain the E-R diagram as shown above.

Now, suppose we want to record **managers for tasks performed by an employee at a branch**; that is, we want to record managers for (employee, branch, job) combinations. Let us assume that there is an entity set **manager**.

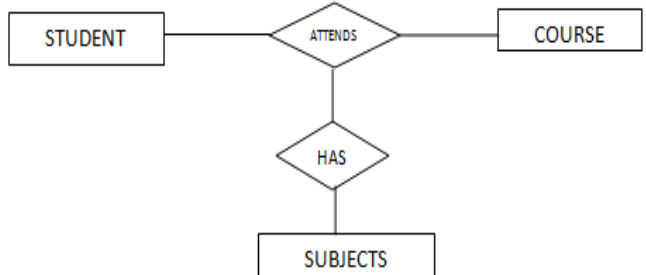
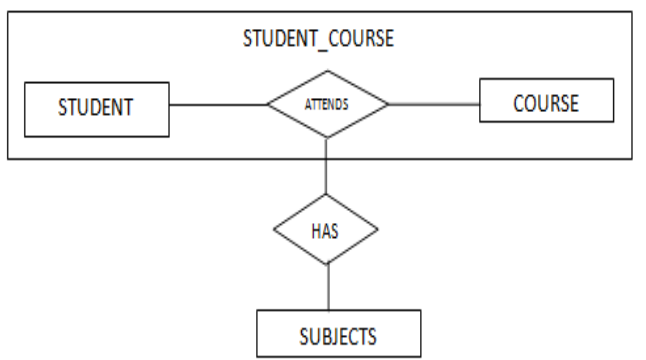
Relationship sets **works-on** and **manages** represent overlapping information. It appears that the relationship sets works-on and manages can be **combined** into one single relationship set. But we should not do this because

- Some employee, branch, job combinations many not have a manager.

So we cannot **discard the works-on** relationship. There is redundant information in the resultant figure, however, since every employee, branch, job combination in manages is also in works-on.

<p style="text-align: center;">E-R diagram with aggregation.</p> 	<p>We can eliminate this redundancy via aggregation.</p> <p>In this example, we regard the relationship set works-on (relating the entity sets employee, branch, and job) as a higher-level entity set called works-on.</p> <p>Such an entity set is treated in the same manner as is any other entity set. We can then create a binary relationship manages between works-on and manager to represent who manages what tasks.</p>
--	--

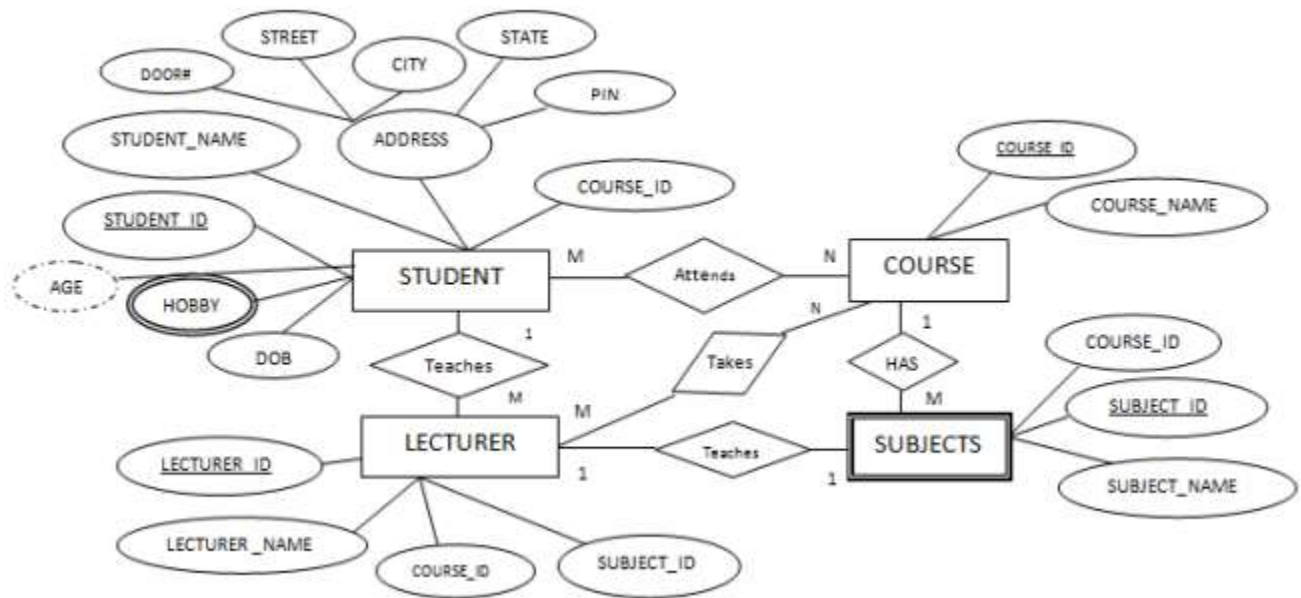
Another Example :

<p>Student attends the Course, and he has some subjects to study. At the same time, Course offers some subjects. Here a relation is defined on a relation. But ER diagram does not entertain such a relation. It supports mapping between entities, not between relations. So what can we do in this case?</p>	
<p>So, merge STUDENT and COURSE as one entity. This process of merging is called aggregation. It is completely different from generalization. In generalization, we merge entities of same domain into one entity. In this case we merge related entities into one entity.</p> <p>Here we have merged STUDENT and COURSE into one entity STUDENT_COURSE. This new entity forms the mapping with SUBJECTS. The new entity STUDENT_COURSE, in turn has two entities STUDENT and COURSE with 'Attends' relationship.</p>	

Convert ER Diagram into Tables

As the database grows, the ER diagram representation becomes more complex and crowded. Once designing ER diagram is complete, we need to put it into logical structure. But how it can be done? Let us discuss this in the last section.

There are various steps involved in converting it into tables and columns. Each type of entity, attribute and relationship in the diagram takes their own depiction here. Consider the ER diagram below and will see how it is converted into tables, columns and mappings.



The basic rule for converting the ER diagrams into tables is

- **Convert all the Entities in the diagram to tables :** All the entities represented in the rectangular box in the ER diagram become independent tables in the database. In the below diagram, **STUDENT, COURSE, LECTURER and SUBJECTS** forms individual tables.

- **All single valued attributes of an entity is converted to a column of the table**

In the STUDENT Entity, STUDENT_ID, STUDENT_NAME form the columns of STUDENT table. Similarly, LECTURER_ID, LECTURER_NAME form the columns of LECTURER table. And so on.

- **Key attribute in the ER diagram becomes the Primary key of the table.**

- **Declare the foreign key column, if applicable.**

In the diagram, attribute COURSE_ID in the STUDENT entity is from COURSE entity. Hence add COURSE_ID in the STUDENT table and assign it foreign key constraint. COURSE_ID and SUBJECT_ID

in LECTURER table forms the foreign key column. Hence by declaring the foreign key constraints, mapping between the tables are established.

- **Any multi-valued attributes are converted into new table.**

A **hobby** in the Student table is a multivalued attribute. Any student can have **any number of hobbies**. So we cannot represent multiple values in a single column of STUDENT table. We need to store it separately, so that we can store any number of hobbies, adding/ removing / deleting hobbies should not create any redundancy or anomalies in the system. Hence we create a separate table **STUD_HOBBY** with **STUDENT_ID** and **HOBBY** as its columns. We create a **composite** key using both the columns.

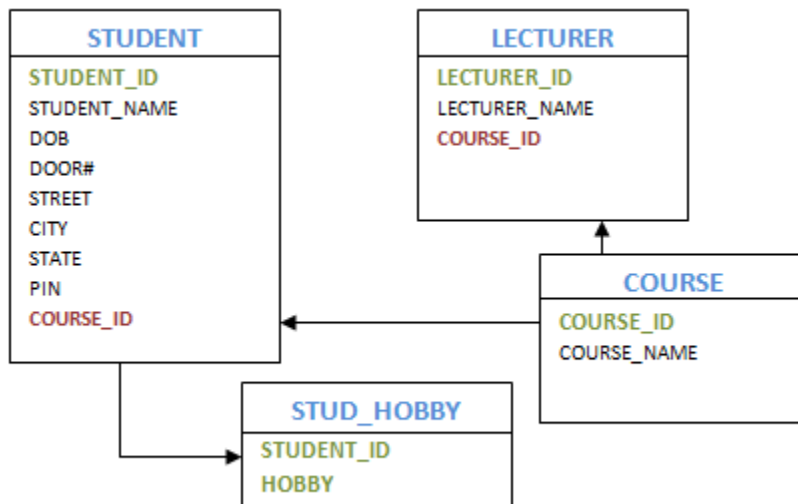
- **Any composite attributes are merged into same table as different columns.**

In the diagram above, Student Address is a composite attribute. It has Door#, Street, City, State and Pin. These attributes are merged into STUDENT table as individual columns.

- **One can ignore derived attribute, since it can be calculated at any time.**

In the STUDENT table, **Age** can be derived at any point of time by calculating the difference between DateOfBirth and current date. Hence we need not create a column for this attribute. It reduces the duplicity in the database.

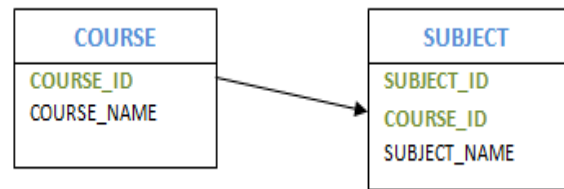
These are the very basic rules of converting ER diagram into tables and columns, and assigning the mapping between the tables. Table structure at this would be as below:



Let us see some of the special cases.

- **Converting Weak Entity**

Weak entity is also represented as table. All the attributes of the weak entity forms the column of the table. But the key attribute represented in the diagram cannot form the primary key of this table. We have to add a foreign key column, which would be the primary key column of its strong entity. This foreign key column along with its key attribute column forms the primary key of the table.



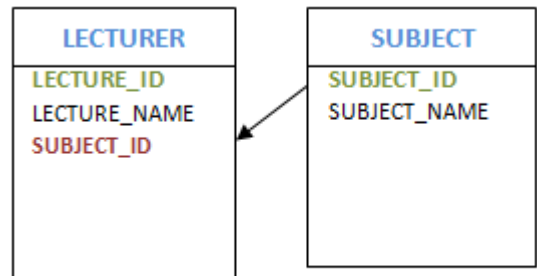
In our example above, SUBJECTS is the weak entity. Although SUBJECT_ID is represented as key attribute in the diagram, it cannot be considered as primary key. COURSE is the strong entity related to SUBJECT. Hence the primary key COURSE_ID of COURSE is added to SUBJECT table as foreign key. Now we can create a composite primary key out of COURSE_ID and SUBJECT_ID.

- **Representing 1:1 relationship**

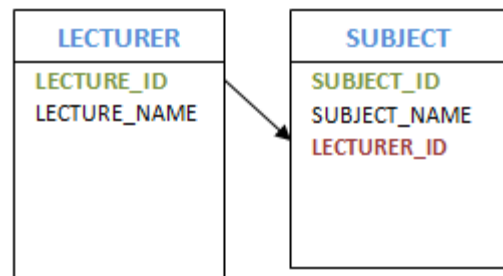
Imagine SUBJECT is not a weak entity, and we have LECTURER teaches SUBJECT relation. It is a 1:1 relation. i.e.; one lecturer teaches only one subject. We can represent this case in two ways

1. Create table for both LECTURER and SUBJECT. Add the primary key of LECTURER in SUBJECT table as foreign key. It implies the lecturer name for that particular subject.
2. Create table for both LECTURER and SUBJECT. Add the primary key of SUBJECT in LECTURER table as foreign key. It implies the subject taught by the lecturer.

In both the case, meaning is same. Foreign key column can be added in either of the table, depending on the developer's choice.

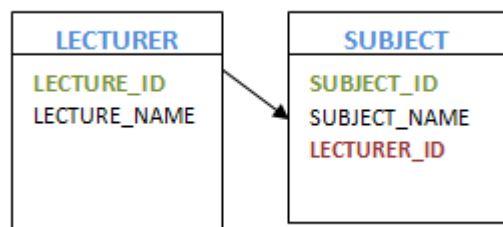


OR



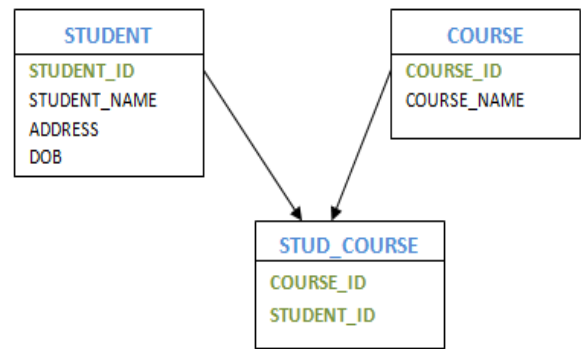
- **Representing 1:N relationship**

Consider SUBJECT and LECTURER relation, where each Lecturer teaches multiple subjects. This is a 1:N relation. In this case, primary key of LECTURER table is added to the SUBJECT table. i.e.; the primary key at 1 cardinality entity is added as foreign key to N cardinality entity.



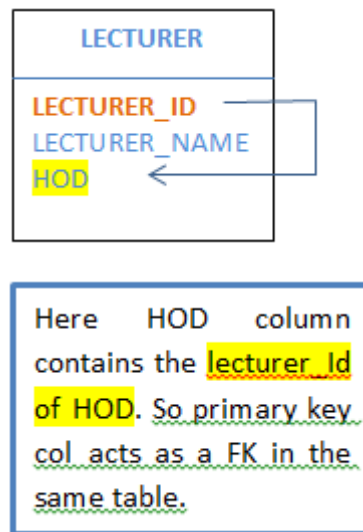
- **Representing M:N relationship**

Consider the example, multiple students enrolled for multiple courses, which is M:N relation. Create one more table for the relation 'Enrolment' and name it as **STUD_COURSE**. Add the primary keys of COURSE and STUDENT into it, which forms the composite primary key of the new table. We can add any additional columns, if present as attribute of the relation in ER diagram.



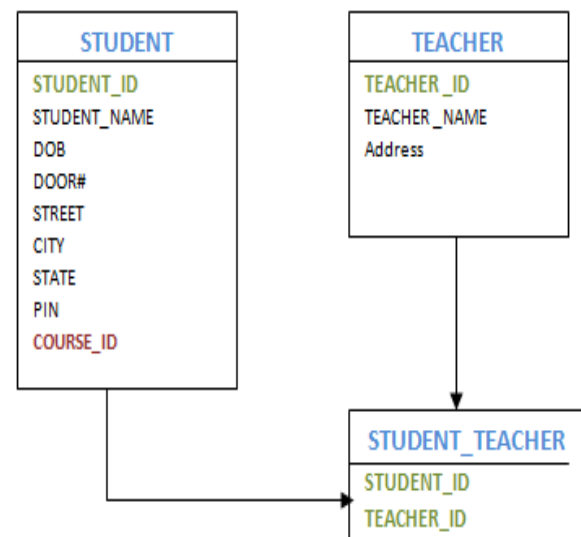
- **Self Referencing 1:N relation**

Consider the example of HOD and Lecturers. Here one of the Lecturers is a HOD of the department. i.e.; one HOD has multiple lecturers working with him. In this case, we create LECTURER table for the Lecturer entity. In order to represent **HOD**, we add one more column to LECTURER table which is **same column as primary key, but acts as a foreign key**. i.e.; **LECTURER_ID** is the primary key of LECTURER table. We add one more column HOD, which will have LECTURER_ID of the HOD. Hence LECTURER table will show HOD's Lecturer ID for each Lecturer. In this case, primary key column acts as a foreign key in the same table.



- **Self Referencing M:N relation**

Consider Student and Teacher example as 'Many students have Many Teachers teaching the subjects'. Here relation between **Student and Teacher is M:N**. In this case, create independent tables for student and teacher, and set their primary keys. Then we create a new table for the relationship 'have' as STUDENT_TEACHER, which will have student and teacher combination, and any other columns if applicable. Basically, student-teacher combination is the two primary key columns from respective tables, hence establishing the relationship between them. Both the primary keys from both tables act as a composite primary key in the new table. This reduces the storing of redundant data and consistency in the database.



Entity Relationship (ER) Modeling

Example - 1

We are going to design an Entity Relationship (ER) model for a college database. Say we have the following requirement statements.

1. A college contains many departments
2. Each department can offer any number of courses
3. Many instructors can work in a department
4. An instructor can work only in one department
5. For each department there is a Head
6. An instructor can be head of only one department
7. Each instructor can take any number of courses
8. A course can be taken by only one instructor
9. A student can enroll for any number of courses
10. Each course can have any number of students

Step 1 : Identify the Entities

From the statements given, the entities are **Department, Course, Instructor, Student**

Step 2 : Identify the relationships

- One department offers many courses. But one particular course can be offered by only one department. Hence the cardinality between department and course is **One to Many (1:N)**
- One department has multiple instructors. But instructor belongs to only one department. Hence the cardinality between department and instructor is **One to Many (1:N)**
- One department has only one head and one head can be the head of only one department. Hence the cardinality is **one to one. (1:1)**
- One course can be enrolled by many students and one student can enroll for many courses. Hence the cardinality between course and student is **Many to Many (M:N)**
- One course is taught by only one instructor. But one instructor teaches many courses. Hence the cardinality between course and instructor is **Many to One (N :1)**

Step 3: Identify the key attributes

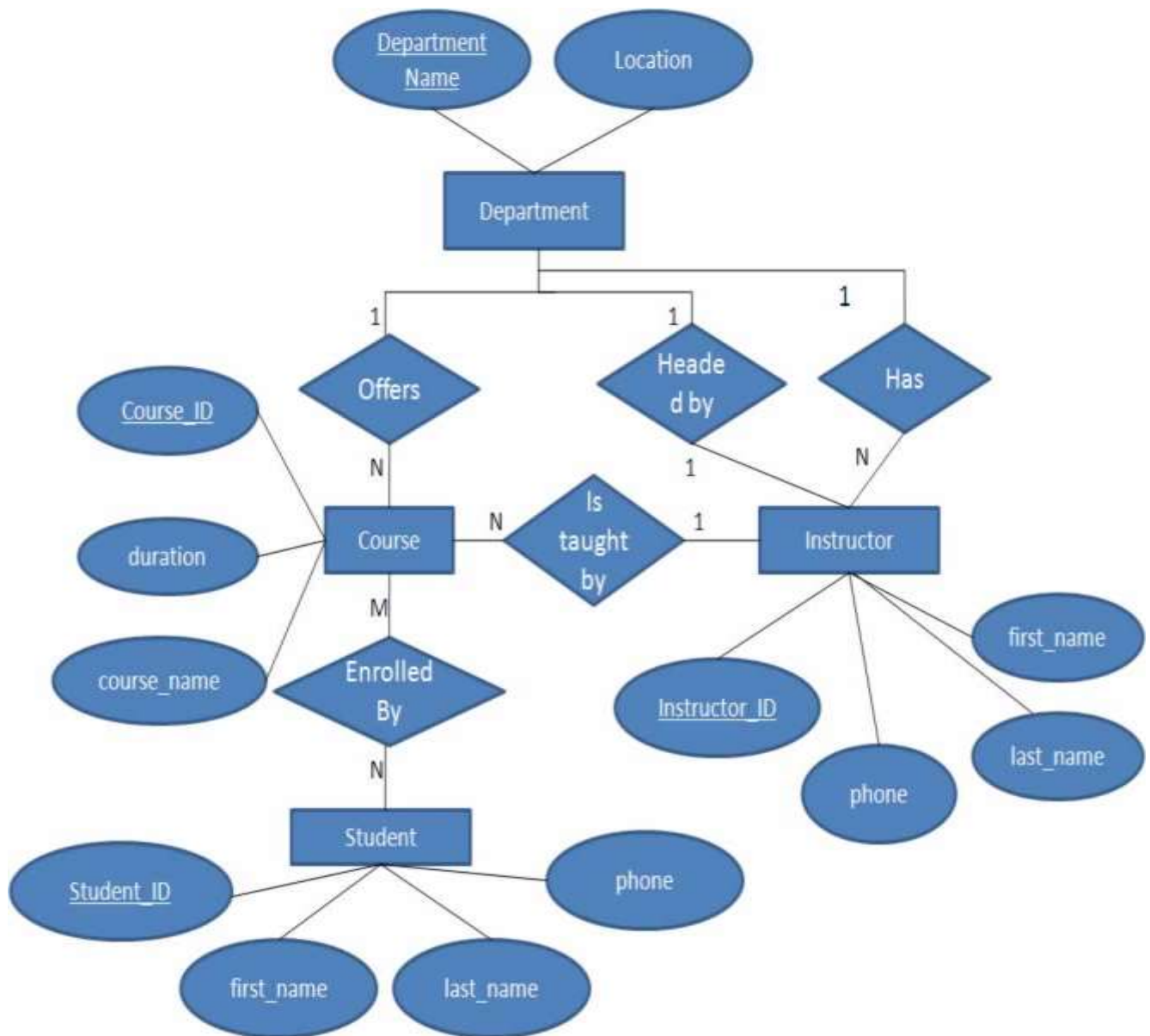
- "Department_Name" can identify a department uniquely. Hence Department_Name is the key attribute for the Entity "Department".
- Course_ID is the key attribute for "Course" Entity.
- Student_ID is the key attribute for "Student" Entity.
- Instructor_ID is the key attribute for "Instructor" Entity.

Step 4: Identify other relevant attributes

- For the department entity, other attributes are location
- For course entity, other attributes are course_name, duration
- For instructor entity, other attributes are first_name, last_name, phone
- For student entity, first_name, last_name, phone

Step 5: Draw complete ER diagram

By connecting all these details, we can now draw ER diagram as given below.



Example - 2

The music database is designed to store details of a music collection, including the albums in the collection, the artists who made them, the tracks on the albums, and when each track was last played.

The music database stores details of a personal music library, and could be used to manage your MP3, CD, or vinyl collection. Because this database is for a personal collection, it's relatively simple and stores only the relationships between **artists, albums, and tracks**. It ignores the requirements of many music genres, making it most useful for storing popular music and less useful for storing jazz or classical music.

List of requirements:

- The collection consists of albums.
- An album is made by exactly one artist.
- An artist makes one or more albums.
- An album contains one or more tracks
- Artists, albums, and tracks each have a name.
- Each track is on exactly one album.
- Each track has a time length, measured in seconds.
- When a track is played, the date and time the playback began (to the nearest second) should be recorded; this is used for reporting when a track was last played, as well as the number of times music by an artist, from an album, or a track has been played.

There's no requirement to capture composers, group members or sidemen, recording date or location, the source media, or any other details of artists, albums, or tracks.

Step 1 : Identify the Entities

From the statements given, the entities are **Artists, albums, Tracks and Played**.

Step 2 : Identify the relationships

- One artist can make many albums
- One album can contain many tracks
- One track can be played many times. Conversely, each play is associated with one track, a track is on one album, and an album is by one artist.

Step 3: Identify the key attributes

- The only strong entity in the database is Artist, which has an `artist_id` attribute that uniquely identifies it.
- Each Album entity is uniquely identified by its `album_id` combined with the `artist_id` of the corresponding Artist entity.

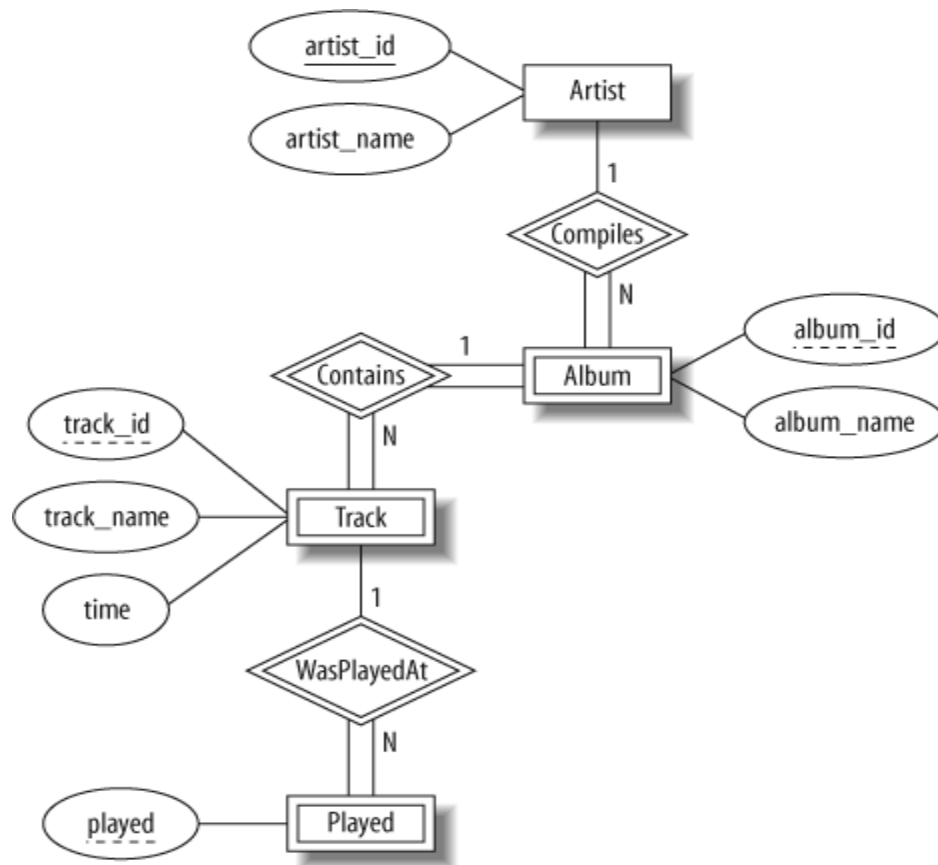
- A Track entity is similarly uniquely identified by its `track_id` combined with the related `album_id` and `artist_id` attributes.
- The Played entity is uniquely identified by a combination of its `played` time, and the related `track_id`, `album_id`, and `artist_id` attributes.

Step 4: Identify other relevant attributes

- The attributes are straightforward: artists, albums, and tracks have names, as well as identifiers to uniquely identify each entity. The track entity has a time attribute to store the duration, and the played entity has a timestamp to store when the track was played.

Step 5: Draw complete ER diagram

By connecting all these details, we can now draw ER diagram as given below.



What it doesn't do

- We've kept the music database simple because adding extra features doesn't help you learn anything new, it just makes the explanations longer. If you wanted to use the music database in practice, then you might consider adding the following features:

- Support for compilations or various-artists albums, where each track may be by a different artist and may then have its own associated album-like details such as a recording date and time. Under this model, the album would be a strong entity, with many-to-many relationships between artists and albums.
- Playlists, a user-controlled collection of tracks. For example, you might create a playlist of your favorite tracks from an artist.
- Track ratings, to record your opinion on how good a track is.
- Source details, such as when you bought an album, what media it came on, how much you paid, and so on.
- Album details, such as when and where it was recorded, the producer and label, the band members or sidemen who played on the album, and even its artwork.
- Smarter track management, such as modeling that allows the same track to appear on many albums.

Example - 3

The **university database** stores details about university students, courses, the semester a student took a particular course (and his mark and grade if he completed it), and what degree program each student is enrolled in. The database is a long way from one that'd be suitable for a large tertiary institution, but it does illustrate relationships that are interesting to query, and it's easy to relate to when you're learning SQL. We explain the requirements next and discuss their shortcomings at the end of this section.

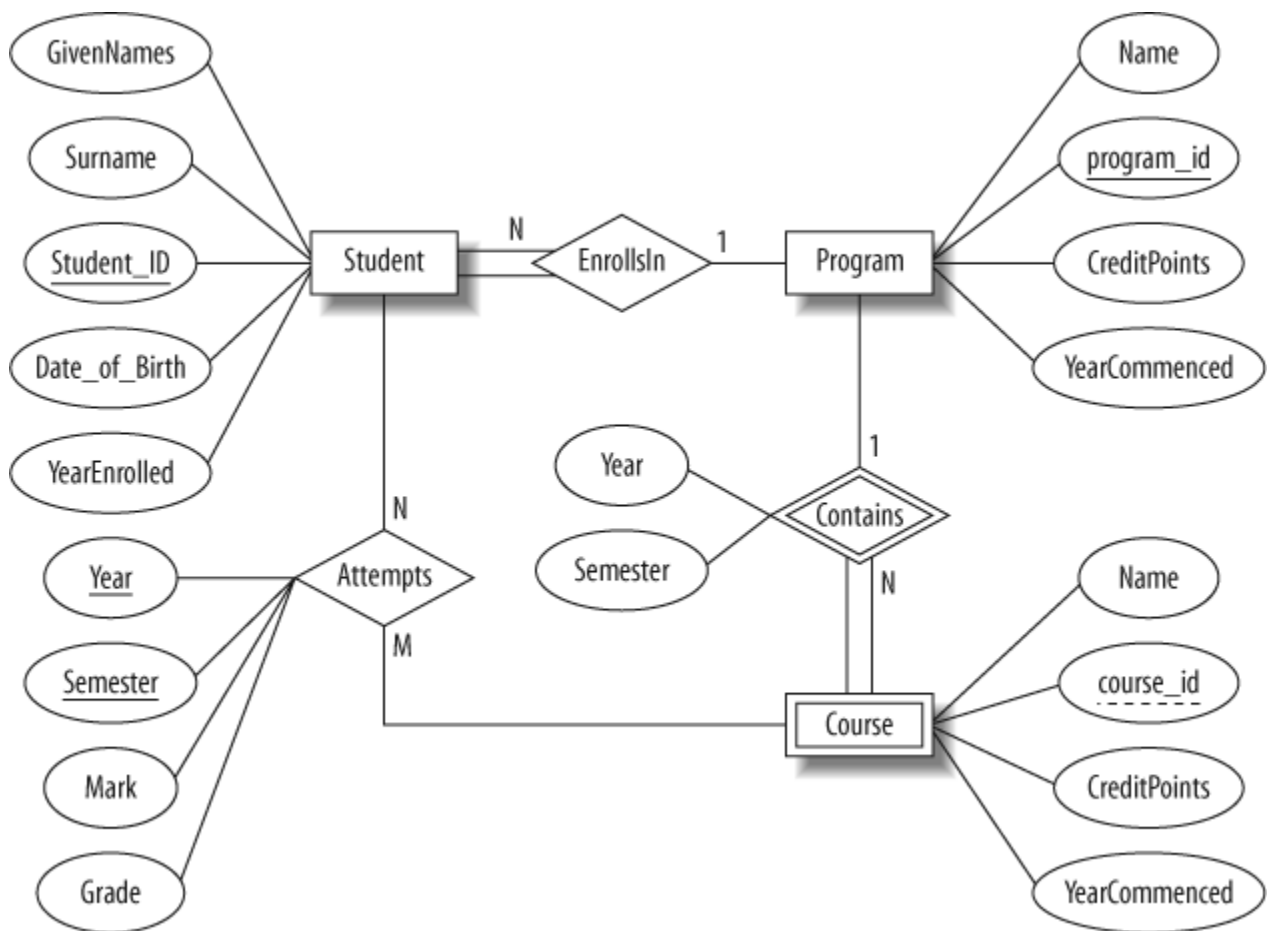
List of Requirements

- The university offers one or more programs.
- A program is made up of one or more courses.
- A student must enroll in a program.
- A student takes the courses that are part of her program.
- A program has a name, a program identifier, the total credit points required to graduate, and the year it commenced.
- A course has a name, a course identifier, a credit point value, and the year it commenced.
- Students have one or more given names, a surname, a student identifier, a date of birth, and the year they first enrolled. We can treat all given names as a single object—for example, "John Paul."
- When a student takes a course, the year and semester he attempted it are recorded. When he finishes the course, a grade (such as A or B) and a mark (such as 60 percent) are recorded.
- Each course in a program is sequenced into a year (for example, year 1) and a semester (for example, semester 1).

By analyzing above requirements following are our findings:

- Student is a strong entity, with an identifier, `student_id`, created to be the primary key used to distinguish between students (remember, we could have several students with the same name).

- Program is a strong entity, with the identifier program_id as the primary key used to distinguish between programs.
- Each student must be enrolled in a program, so the Student entity participates totally in the many-to-one EnrollsIn relationship with Program. A program can exist without having any enrolled students, so it participates partially in this relationship.
- A Course has meaning only in the context of a Program, so it's a weak entity, with course_id as a weak key. This means that a Course is uniquely identified using its course_id and the program_id of its owning program.
- As a weak entity, Course participates totally in the many-to-one identifying relationship with its owning Program. This relationship has Year and Semester attributes that identify its sequence position.
- Student and Course are related through the many-to-many Attempts relationships; a course can exist without a student, and a student can be enrolled without attempting any courses, so the participation is not total.
- When a student attempts a course, there are attributes to capture the Year and Semester, and the Mark and Grade.



What it doesn't do

Our database design is rather simple, but this is because the requirements are simple. For a real university, many more aspects would need to be captured by the database. For example, the requirements don't mention anything about campus, study mode, course prerequisites, lecturers, timetabling details, address history, financials, or assessment details. The database also doesn't allow a student to be in more than one degree program, nor does it allow a course to appear as part of different programs.

Example - 4

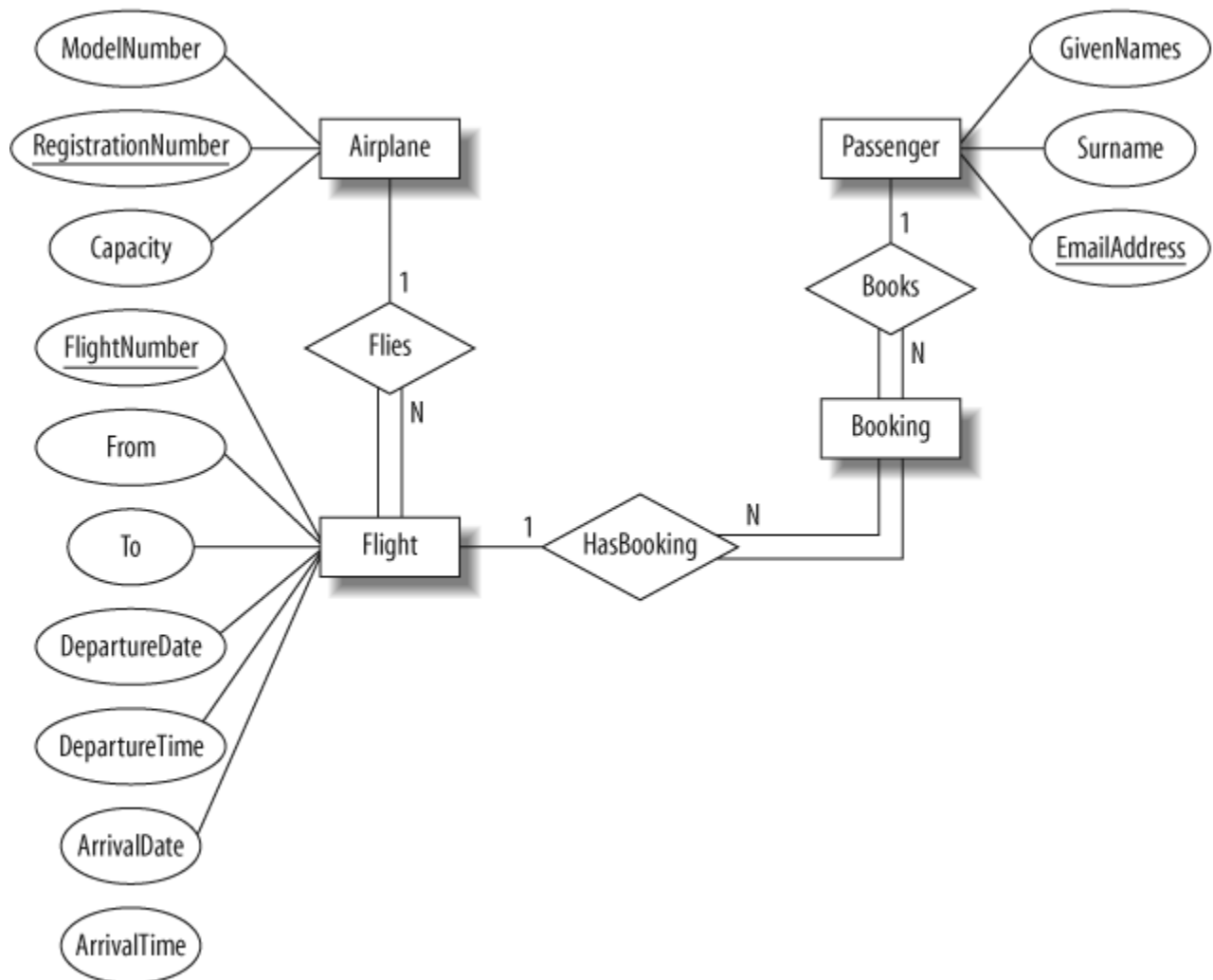
The flight database stores details about an airline's fleet, flights, and seat bookings. Again, it's a hugely simplified version of what a real airline would use, but the principles are the same.

List of Requirements:

- The airline has one or more airplanes.
- An airplane has a model number, a unique registration number, and the capacity to take one or more passengers.
- An airplane flight has a unique flight number, a departure airport, a destination airport, a departure date and time, and an arrival date and time.
- Each flight is carried out by a single airplane.
- A passenger has given names, a surname, and a unique email address.
- A passenger can book a seat on a flight.

After analyzing above requirement following are our findings:

- An Airplane is uniquely identified by its RegistrationNumber, so we use this as the primary key.
- A Flight is uniquely identified by its FlightNumber, so we use the flight number as the primary key. The departure and destination airports are captured in the From and To attributes, and we have separate attributes for the departure and arrival date and time.
- Because no two passengers will share an email address, we can use the EmailAddress as the primary key for the Passenger entity.
- An airplane can be involved in any number of flights, while each flight uses exactly one airplane, so the Flies relationship between the Airplane and Flight relationships has cardinality 1:N; because a flight cannot exist without an airplane, the Flight entity participates totally in this relationship.
- A passenger can book any number of flights, while a flight can be booked by any number of passengers. As discussed earlier in Intermediate Entities," we could specify an M:N Books relationship between the Passenger and Flight relationship, but considering the issue more carefully shows that there is a hidden entity here: the booking itself. We capture this by creating the intermediate entity Booking and 1:N relationships between it and the Passenger and Flight entities. Identifying such entities allows us to get a better picture of the requirements. Note that even if we didn't notice this hidden entity, it would come out as part of the ER-to-tables mapping process we'll describe next in Using the Entity Relationship Model."



What it doesn't do

- Again, this is a very simple flight database. There are no requirements to capture passenger details such as age, gender, or frequent-flyer number.
- We've treated the capacity of the airplane as an attribute of an individual airplane. If, instead, we assumed that the capacity is determined by the model number, we would have created a new **AirplaneModel** entity with the attributes ModelNumber and Capacity. The **Airplane** entity would then not have a Capacity attribute.
- We've mapped a different flight number to each flight between two destinations. Airlines typically use a flight number to identify a given flight path and schedule, and they specify the date of the flight independently of the flight number. For example, there is one IR655 flight on April 1, another on April 2, and so on. Different airplanes can operate on the same flight number over time; our model would need to be extended to support this.
- The system also assumes that each leg of a multihop flight has a different FlightNumber. This means that a flight from Dubai to Christchurch via Singapore and Melbourne would need a

different FlightNumber for the Dubai-Singapore, Singapore-Melbourne, and Melbourne-Christchurch legs.

- Our database also has limited ability to describe airports. In practice, each airport has a name, such as “Melbourne Regional Airport,” “Mehrabad,” or “Tullamarine.” The name can be used to differentiate between airports, but most passengers will just use the name of the town or city. This can lead to confusion, when, for example, a passenger could book a flight to Melbourne, Florida, USA, instead of Melbourne, Victoria, Australia. To avoid such problems, the International Air Transport Association (IATA) assigns a unique airport code to each airport; the airport code for Melbourne, Florida, USA is MLB, while the code for Melbourne, Victoria, Australia is MEL. If we were to model the airport as a separate entity, we could use the IATA-assigned airport code as the primary key. Incidentally, there’s an alternative set of airport codes assigned by the International Civil Aviation Organization (ICAO); under this code, Melbourne, Florida is KMLB, and Melbourne, Australia is YMML.

Example - 6

Suppose you are given the following requirements for developing a simple database for a league conducted by All India Football Association (IFAL) :

- The IFAL has many teams
- Each team has a name, a city, a coach, a captain, and a set of players
- Each player belongs to only one team
- Each player has a name, a position (such as left wing or goalie), a skill level, and a set of injury records
- A team captain is also a player
- A game is played between two teams (referred to as host_team and guest_team) and has a date (such as May 11th, 1999) and a score (such as 4 to 2).

Construct a clean and concise ER diagram for the IFAL database using the Chen notation. List your assumptions and clearly indicate weak entity, the cardinality mappings as well as any role indicators in your ER diagram.

