Relational Database Design

- Consequences of poor database design
- The process of database normalization
- Functional dependencies
- Lossless joins and dependency preservation
- Dr. E.F.Codd Rules
- Difference between DBMS and RDBMS
- 1st Normal Form, 2nd Normal Form, 3rd Normal Form, Boyce-Codd Normal Form
- Examples of normalization
- E-R modeling (different types of entities, attributes, relationships and their representation in the E-R diagram)
- Case studies: Library (Book issue and return), Bank (Opening saving bank account)

Database:

A database can be defined as a collection of coherent, meaningful data. The phrase collection of coherent data needs to have a point of reference to be understood.

DBMS:

The system that would help in managing data in such a database, called Data Base Management System. DBMS is a system that allows inserting, updating, deleting and processing of data.

➢ DBMS V/S RDBMS

DBMS	RDBMS
❖ In DBMS relationship between two tables or file are maintained programmatically.	❖ In RDBMS relationship between two tables or files can be specified at the time of table creation.
❖ DBMS does not support client/server architecture.	❖ Most of the RDBMS support client/server architecture.
❖ DBMS does not support distributed database.	❖ Most of the RDBMS support distributed database.
❖ In DBMS there is no security of data.	❖ IN RDBMS there are multiple levels of security ✓ Logging in O/S level ✓ Command level ✓ Object level
❖ Each table is given an extension in DBMS	❖ Many tables are grouped in one database in RDBMS
❖ DBMS may satisfy less than 7 to 8 rules of Dr.E.F.Codd	❖ RDBMS may satisfy more than 7 to 8 rules of Dr.E.F.Codd
❖ Naming Conventions:	❖ Naming Conventions:
1.Field	1.Column, Attributes
2.Record	2.Row, Tuple, Entity
3. File	3.Table, Relation, Entity Class

A Consequence of Bad Design

Consider the following relation scheme pertaining to the information about a student maintained by a university:

STDINF (Name, Course, Phone_No, Major, Prof, Grade**)**

Figure 6.1 shows some tuples of a relation on the relation scheme **STDINF** ***(Name,*** *Course, Phone_No, Major, Prof, and Grade).* The functional dependencies among its attributes are shown in Figure 6.2. The key of the relation is *Name Course* and the relation has, in addition, the following functional dependencies *{Name --> Phone_No, Name --> Major, Name Course --> Grade, Course --> Prof}.*

Here the attribute *Phone_No,* which is not in any key of the relation scheme **STDINF**, is not functionally dependent on the whole key but only on part of the key, namely, the attribute *Name*. Similarly, the attributes *Major and Prof*, which are not in any key of the relation scheme **STDINF** either, are fully functionally dependent on the attributes *Name and Course*, respectively. Thus the determinants of these functional dependencies are again not the entire key but only part of the key of the relation. Only the attribute Grade is fully functionally dependent on the key **Name Course**.

Student data represented in relation **STDINF**

Name	Course	Phone_No	Major	Prof	Grade
Jones	353	237-4539	Comp Sci	Smith	A
Nick	329	427-7390	Chemistry	Turner	B
Jones	328	237-4539	Comp Sci	Clark	B
Martin	456	388-5183	Physics	James	A
Dulles	293	371-6259	Decision	Cook	C
Duke	491	823-7293	Mathemati	Lamb	B
Duke	356	823-7293	Mathemati	Bond	in prog
Jones	492	237-4539	Comp Sci	Cross	in prog
Baxter	379	839-0827	English	Broes	C

The relation scheme **STDINF** can lead to several undesirable problems:

- **Redundancy**: The aim of the database system is to reduce redundancy, meaning that information is to be stored only once. Storing information several times leads to the waste of storage space and an increase in the total size of the data stored.

  For Example: In the relation of Figure 6.1, the Major and Phone_No of a student-are stored several times in the database: once for each course that is or was taken by a student.

- **Update Anomalies**: Multiple copies of the same fact may lead to update anomalies or inconsistencies when an update is made and only some of the multiple copies are updated.

  For Example, a change in the Phone_No of Jones must be made, for consistency, in all tuples pertaining to the student Jones. If one of the three tuples of Figure 6.2 is not changed to reflect the new Phone_No of Jones, there will be an inconsistency in the data.

- **Insertion Anomalies**: If this is the only relation in the database showing the association between a faculty member and the course he or she teaches, the fact that a given professor is teaching a given course cannot be entered in the database unless a student is registered in the course. Also, if another relation also establishes a relationship between a course and a professor who teaches that course (for example, the SCHEDULE relation of Figure A), the information stored in these relations has to be consistent.

- **Deletion Anomalies**: If the only student registered in a given course discontinues the course, the information as to which professor is offering the course will be lost if this is the only relation in the database showing the association between a faculty member and the course she or he teaches.

  If another relation in the database also establishes the relationship between a course and a professor who teaches that course, the deletion of the last tuple in STDINF for a given course will not cause the information about the course's teacher to be lost.

Functional dependencies

Functional dependencies are the outcome (result) of the interrelationship among attributes of an entity represented by a relation or due to the relationship between entities that is also represented by a relation.

A functional dependency is a relationship between two attributes. Typically between the PK and other non-key attributes within the table. Thus, if R represents a relation and the set X of attributes represents the key attribute of R, then for any other set of attribute Y of R, the functional dependency represent as

X ————> Y

The left-hand side of the FD is called the determinant, and the right-hand side is the dependent.

If R represents a many-to-one relationship between two entities, say from E1 to E2, and if X contains attributes that form a key of El and Y contains attributes that contain d key of E2, again the FD X —> Y will hold.

But if R represents a one-to-one relationship between entity E1 and E2, the FD Y —› X will hold in addition to The FD X —> Y.

Examples:

SID ————-> Name, Address, Birthdate

SID determines names and address and birthdays. Given SID, we can determine any of the other attributes within the table.

Sid, Course ———> DateCompleted

SiD and Course determine date completed. This must also work for a composite PK.

ISBN ———> Title

ISBN determines title.

Decomposition

All attributes of an original schema (R) must appear in the decomposition (R_1, R_2):

$$R = R_1 \cup R_2$$

Decompose the relation schema *Lending-schema* into:

Branch-schema = (branch-name, branch-city,assets)

Loan-info-schema = (customer-name, loan-number, branch-name, amount)

Let R be a relation schema

A set of relation schemas { R1, R2,…, Rn } is a decomposition of R if

R = R1 U R2 U …..U Rn each Ri is a subset of R (for i = 1,2…,n)

Goal of Decomposition :

- Eliminate redundancy by decomposing a relation into several relations in a higher normal form.
- It is important to check that a decomposition does not lead to bad design

Problem with Decomposition:

- Given instances of the decomposed relations, we may not be able to reconstruct the corresponding instance of the original relation – information loss

Example:

R

Model Name	Price	Category
a11	100	Canon
s20	200	Nikon
a70	150	Canon

R1

Model Name	Category
a11	Canon
s20	Nikon
a70	Canon

R2

Price	Category
100	Canon
200	Nikon
150	Canon

R1 U R2

Model Name	Price	Category
a11	100	Canon
a11	150	Canon
s20	200	Nikon
a70	100	Canon
a70	150	Canon

R

Model Name	Price	Category
a11	100	Canon
s20	200	Nikon
a70	150	Canon

Lossy decomposition:

- In previous example, additional tuples are obtained along with original tuples
- Although there are more tuples, this leads to less information
- Due to the loss of information, decomposition for previous example is called lossy decomposition or lossy-join decomposition

Lossless Join Decomposition

In a database, we sometimes decompose tables into sub-tables, as you learned, in order to avoid repetition of information in the tables (so to avoid redundancy). However, when we decompose a

table in sub-tables, we don't want to lose any data. In other words, when we join the sub-tables later on, we would want to recover our initial table.

A decomposition {R1, R2,…, Rn} of a relation R is called a lossless decomposition for R if the natural join of R1, R2,…, Rn produces exactly the relation R.

A decomposition is lossless if we can recover:

R(A, B, C)

Decompose

R1(A, B) R2(A, C)

Recover

R'(A, B, C)

Thus, R' = R

Lossless Decomposition Property

R : relation

F : set of functional dependencies on R

X,Y : decomposition of R

Decomposition is lossles if :

X ∩ Y → X, that is: all attributes common to both X and Y functionally determine ALL the attributes in X OR

X ∩ Y → Y, that is: all attributes common to both X and Y functionally determine ALL the attributes in Y

In other words, if X ∩ Y forms a superkey of either X or Y, the decomposition of R is a lossless decomposition

Show that decomposition is Lossless Decomposition

- Since branch-name branch-city assets, the augmentation rule for FD implies that:
  - branch-name branch-name branch-city assets
- Since Branch-schema ∩ Loan-info-schema = {branch-name}
  - Thus, this decomposition is Lossless decomposition

Dependency Preservation

Another desirable property in database design is dependency preservation. So the need for dependency preservation is as follows:

- We would like to check easily that updates to the database do not result in illegal relations being created.
- It would be nice if our design allowed us to check updates without having to compute natural joins.
- To know whether joins must be computed, we need to determine what functional dependencies may be tested by checking each relation individually.

A decomposition D = {R1, R2, ..., Rn} of R is dependency-preserving with respect to F if the union of the projections of F on each Ri in D is equivalent to F; that is if $(F1 \cup F2 \cup \ldots \cup Fn)+ = F+$

Note: It is not necessary that all dependencies from the relation R appear in some relation Ri. It is sufficient that the union of the dependencies on all the relations Ri be equivalent to the dependencies on R.

Property of Dependency-Preservation

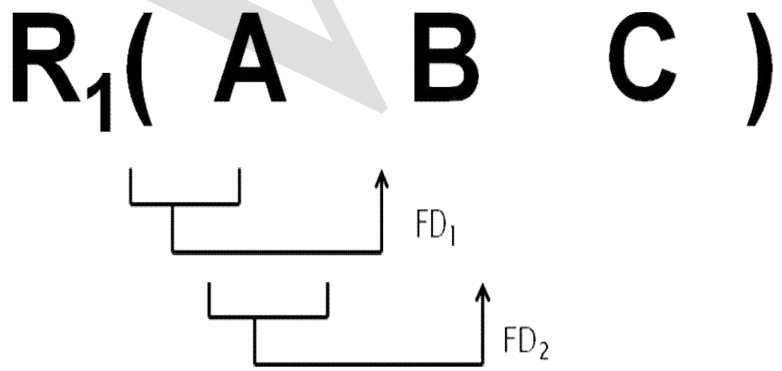
If decomposition is not dependency-preserving, therefore, that dependency is lost in the decomposition

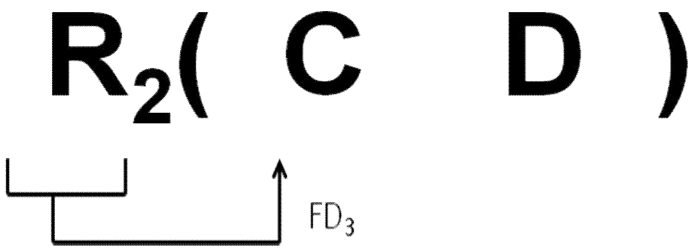
Example

R (A B C D)

- FD1: A ➔ B
- FD2: B ➔ C
- FD3: C ➔ D

Decomposition:
R1 (A B C) R2 (C D)





Normalization

Normalization is a process that helps analysts or database designers to design table structures for an application.

Goal: The focus of normalization is to attempt to reduce redundant table data to the very minimum.

Normalization is a technique that:

- Decomposes data into two-dimensional tables
- Eliminates any relationships in which table data does fully depend upon the primary key of a record
- Eliminates any relationship that contains transitive dependencies

Through the normalization process, the collection of data in a single table is replaced, by the same data being distributed over multiple tables with a specific relationship being setup between the tables.

When the process of normalization is applied to table data and this data is spread across several associated (i.e. a specific relationship has been established) tables, it takes a query much longer to run and retrieve user data from the set of tables.

Hence, often in a commercial application after 100% normalization is carried out across the master tables often the table structures are de-normalized deliberately to make SQL queries run faster. This means that in commercial applications there is often a tradeoff between redundant table data and the speed of query execution.

Need of Normalization OR Why normalization is carried out?

Normalization is carried out for the following reasons:

1. To structure the data between tables so that data maintenance is simplified
2. To allow data retrieval at optimal speed
3. To simplify data maintenance through updates, inserts and deletes
4. To reduce the need to restructure tables as new application requirements arise

5. To improve the quality of design for an application by rationalization of table data

First Normal Form:

When a table is decomposed into two-dimensional tables with all repeating groups of data eliminated, the table data is said to be in its first normal form.

The repetitive portion of data belonging to the record is termed as repeating groups.

To understand the application of normalization to table data the following table structure will be taken as an example:

Field	Key	Type
Project number	- -	
Project name	- -	
Employee number	- -	1-n
Employee name	- -	1-n
Rate category	- -	1-n
Hourly rate	- -	1-n

In above table, 1-n indicates that there are many occurrences of this field - it is a repeating group.

Data held in the above table structure:

Project number	Project name	Employee	Employee name	Rate categor	Hourly rate
P001	Using MySQL On Linux	E001	Sharanam Shah	A	7000
P001	Using MySL On Linux	E002	Vaishali Shah	B	6500
P001	Using MySQL On Linux	E006	Hansel Colaco	C	4500
P002	Using Star Office On Linux	E001	Sharanam Shah	A	7000
P002	Using Star Office On Linux	E007	Chhaya Bankar	B	4000

In the above data there are a few problems:

- The Project Name in the second record is misspelled. This can be solved by removing duplicates. Do this using normalization
- Data is repeated and thus occupies more space

A table is in 1st normal form if:

- There are no repeating groups

- All the key attributes are defined
- All attributes are dependent on a primary key

So far there are no keys, and there are repeating groups. So remove the repeating groups, and define the primary key.

To convert a table to its First Normal Form:

- The unnormalized data in the first table is the entire table
- A key that will uniquely identify each record should be assigned to the table. This key has to be unique because it should be capable of identifying any specific row from the table for extracting information for use. This key is called the table's primary key.

This following table is now in 1st normal form.

Field	**Key**
Project number	Primary Key
Project name	- -
Employee number	Primary Key
Employee name	- -
Rate category	- -
Hourly rate	- -

Second Normal Form

A table is said to be in its second normal form when each record in the table is in the first normal form and each column in the record is fully dependent on its primary key.

A table is in 2nd normal form if:

- It's in 1st normal form.
- It includes no partial dependencies (where an attribute is dependent on only a part of a primary key)

The steps to convert a table to its Second Normal Form:

- Find and remove fields that are related to the only part of the key
- Group the removed items in the another table
- Assign the new table with the key i.e. part of a whole composite key

To convert the table into the second normal form remove and place these fields in a separate table, with the key being that part of the original key they are dependent on.

This leads to the following 3 tables:

Table: EmpPro

Field	Key
Project number	Primary Key
Employee number	Primary Key

Table: Emp

Field	Key
Employee number	Primary Key
Employee name	- -
Rate category	- -
Hourly rate	- -

Table: Proj

Field	Key
Project number	Primary Key
Project name	--

Third Normal Form

Table data is said to be in third normal format when all transitive dependencies are removed from this data.

The table is in 3rd normal form if:

- It's in 2nd normal form
- It contains no transitive dependencies (where a non-key attribute is dependent on another non-key attribute).

A general case of transitive dependencies is as follows: A, B, C are three columns in table.

If C is related to B
If B is related to A
Then C is indirectly related to A

This is when transitive dependency exists.

To convert such data to its third normal form remove this transitive dependency by splitting each relation in two separate relations. This means that data in columns A, B, C must be placed in three separate tables, which are linked using a foreign, key.
Going through all the fields reveals the following:

- Employee table is the only one with more than one non-key attribute
- Employee name is not dependent on either Rate category or Hourly rate
- Hourly rate is dependent on Rate category

To convert the table into the third normal form remove and place these fields in a separate table, with the attribute it was dependent on as key, as follows:
This leads to the following 4 tables:

Table: EmpPro

Field	Key
Project number	Primary Key
Employee number	Primary Key

Table: Emp

Field	Key
Employee number	Primary Key
Employee name	- -
Rate category	- -

Table: Rate

Field	Key
Rate category	- -
Hourly rate	- -

Table: Proj

Field	Key
Project number	Primary Key
Project name	--

These tables are all now in their 3rd normal form, and ready to be implemented. There are other normal forms such as Boyce-Codd normal form, and 4th normal form, but these are very rarely used for business applications. In most cases, tables that are in their 3rd normal form are already conforming to these types of table formats anyway.

Boyce-Codd Normal Form

A relation is in BCNF, if and only if, every determinant is a candidate key.

The difference between 3NF and BCNF is that for a functional dependency A → B, 3NF allows this dependency in a relation if B is a primary-key attribute and A is not a candidate key,
Whereas BCNF insists that for this dependency to remain in a relation, A must be a candidate key.
Property of BCNF:-

- BCNF requires that all nontrivial dependencies be of the form $\alpha \rightarrow \beta$, where α is a super key

BCNF Decomposition

- Place the two candidate primary keys in separate entities
- Place each of the remaining data items in one of the resulting entities according to its dependency on the primary key

For Example:

The following table structure will be taken as an example:

Field	**Key**	**Type**
Project number	- -	
Project name	- -	
Employee number	- -	1-n
Employee name	- -	1-n
Rate category	- -	1-n
Hourly rate	- -	1-n

Table: Emp

Field	Key
Employee number	Primary Key
Project number	--
Employee name	- -
Rate category	- -

Table: Proj

Field	Key
Employee number	Primary Key
Project number	--
Project name	--

Table: EmpProj

Field	Key
Employee number	Primary Key
Project number	--

Dr E. F Codd rules

There are 12 Codd's Rules for RDBMS.

- **Rule 1: The Information Rule -** All data should be presented in table form
- **Rule 2: Guaranteed Access Rule -** All data should be accessible without ambiguity. This can be accomplished through a combination of the table name, primary key, and column name
- **Rule 3: Systematic Treatment of Null Values -** A field should be allowed to remain empty. This involves the support of a null value, which is distinct from an empty' string or a number with a value of zero. Of course, this can't apply to primary keys. In addition, most database implementations support the concept of a not-null field constraint that prevents null values in a specific table column
- **Rule 4: Dynamic On-Line Catalog based on the Relational Model -** A relational database must provide access to its structure through the same tools that are. used to access the data. This is usually accomplished by storing the structure definition within special system tables
- **Rule 5: Comprehensive Data Sublanguage Rule -** The database must support at least one clearly defined language that includes functionality for data definition, data manipulation, data integrity, and database transaction control. All commercial relational databases use forms of standard SQL (i.e. Structured Query Language) as their supported comprehensive language
- **Rule 6: View Updating Rule -** Data can be presented in different logical combinations called views. Each view should support the, same full range of data manipulation that has direct access to a table, available. In practice, providing update and delete access to logical views is difficult and is not fully supported by any current database
- **Rule 7: High-level Insert, Update, and Delete -** Data can be retrieved from a relational database in sets constructed of data from multiple rows and/or multiple tables. This rule states that insert, update, and delete operations should be supported for any retrievable set rather than just for a single row in a single table
- **Rule 8: Physical Data Independence -** The user is isolated from the physical method of storing and retrieving information from the database. Changes can be made to the underlying architecture (hardware, disk storage methods) without affecting how the user accesses it
- **Rule 9: Logical Data Independence -** How data is viewed should not be changed when the logical structure (table's structure) of the database changes. This rule is particularly difficult to satisfy. Most databases rely on strong ties between the data viewed and the actual structure of the underlying tables
- **Rule 10: Integrity Independence -** The database language (like SQL) should support constraints on user input that maintain database integrity. This rule is not fully implemented by most major vendors. At a minimum, all databases do preserve two constraints through

SQL. No component of a primary key can have a null value. If a foreign key is defined in one table, any value in it must exist as a primary key in another table.

- **Rule 11: Distribution Independence -** A user should be totally unaware of whether or not the database is distributed (whether parts of the database exist in multiple locations). A variety of reasons make this rule difficult to implement.
- **Rule 12: Non subversion Rule -** There should be no way to modify the database structure other than through the multiple row database language (like SQL). Most databases today support administrative tools that allow some direct manipulation of the data structure.

Entity-relationship modeling

Entity-Relationship (ER) model, a high-level data model that is useful in developing a conceptual design for a database. Creation of an ER diagram, which is one of the first steps in designing a database, helps the designer(s) to understand and to specify the desired components of the database and the relationships among those components. An ER model is a diagram containing entities or "items", relationships among them, and attributes of the entities and the relationships.

ER modeling helps you to analyze data requirements systematically to produce a well-designed database. So, it is considered a best practice to complete ER modeling before implementing your database.

Why use E-R Diagrams?

- Helps you to define terms related to entity relationship modeling
- Provide a preview of how all your tables should connect, what fields are going to be on each table
- Helps to describe entities, attributes, relationships
- ER diagrams are translatable into relational tables which allows you to build databases quickly
- ER diagrams can be used by database designers as a blueprint for implementing data in specific software applications
- The database designer gains a better understanding of the information to be contained in the database with the help of ERP diagram
- ERD is allowed you to communicate with the logical structure of the database to users

Components of the ER Diagram

- Entities
- Attributes

- Relationships

Entity

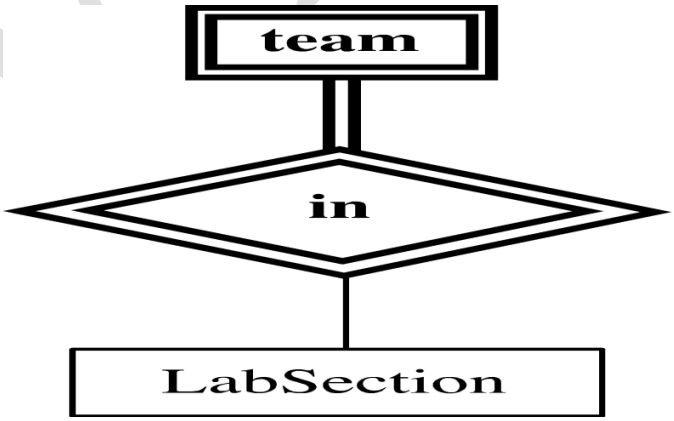
An **entity** is a real-world item or concept that exists on its own. In our example, a particular student (such as, "Emanuel Vagas"), team, lab section, or experiment is an entity. The set of all possible values for an entity, such as all possible students, is the **entity type**. In an ER model, we diagram an entity type as a rectangle containing the type name, such as *student.*

An entity type is **strong** if its existence does not depend on some other entity type. Otherwise, the entity type is **weak**. A weak entity is a type of entity which doesn't have its key attribute. It can be identified uniquely by considering the primary key of another entity. For that, weak entity sets need to have participation.

Examples of entities:

- Person: Employee, Student, Patient
- Place: Store, Building
- Object: Machine, product, and Car
- Event: Sale, Registration, Renewal
- Concept: Account, Course

In the physics laboratory ER model example, the entity type *student* is **strong** because its existence does not depend on some other entity type. However, the *team* entity type is **weak**. The existence of 'team'depends on the existence of *LabSection,* and we call the 'IN' identifying relationship. We draw double lines around the identifying relationship, the *team* entity type, and the line connecting the two to indicate the weak entity type.

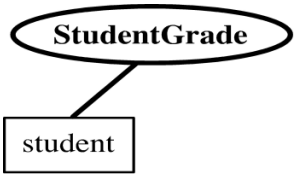


Difference between Strong and Weak Entity Set

Strong Entity Set	Weak Entity Set
Strong entity set always has a primary key.	It does not have enough attributes to build a primary key.
It is represented by a rectangle symbol.	It is represented by a double rectangle symbol.
It contains a Primary key represented by the underline symbol.	It contains a Partial Key which is represented by a dashed underline symbol.
The member of a strong entity set is called as dominant entity set.	The member of a weak entity set called as a subordinate entity set.
Primary Key is one of its attributes which helps to identify its member.	In a weak entity set, it is a combination of primary key and partial key of the strong entity set.
In the ER diagram the relationship between two strong entity set shown by using a diamond symbol.	The relationship between one strong and a weak entity set shown by using the double diamond symbol.
The connecting line of the strong entity set with the relationship is single.	The line connecting the weak entity set for identifying relationship is double.

Attribute

Each entity has **attributes**, or particular properties that describe the entity. For example, student Emanuel Vagas has properties of his own Student Identification number, name, and grade. A particular value of an attribute, such as 93 for the grade, is a **value** of the attribute. Most of the data in a database consists of values of attributes. The set of all possible values of an attribute, such as integers from 0 to 100 for a grade, is the **attribute domain**. In an ER model, an attribute name appears in an oval that has a line to the corresponding entity box.

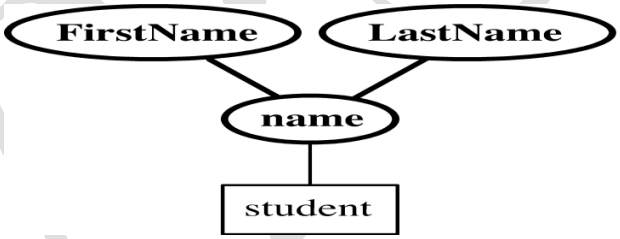


Sometimes the value of an attribute is unknown or missing, and sometimes a value is not applicable. In such cases, the attribute can have the special value of **null**. For example, until the professor grades a laboratory assignment, the team grade is missing or null. For a student who is auditing a course but participating as a team member, it is not applicable for that student to have an individual grade; the student's grade can have the value of null.

A **derived** attribute is one which can be derived from the values of other related attributes and entities. This type of attribute does not include in the physical database. However, their values are derived from other attributes present in the database. For example, age should not be stored directly. Instead, it should be derived from the DOB of that employee.

Multivalued attributes can have more than one value. For example, a student can have more than one mobile number, email address, etc.

An attribute can be simple or composite. A **simple attribute**, such as grade, is one component that is atomic. If we consider the name in two parts, last name and first name, then the name attribute is a composite. A **composite attribute**, such as "Emanuel Vagas", has multiple components, such as "Emanuel" and "Vagas"; and each component is atomic or composite. We illustrate this composite nature in the ER model by branching off the component attributes



Relationships

A **relationship type** is a set of associations among entities. For example, the *student* entity type is related to the *team* entity type because each student is a member of a team. In this case, a **relationship** or **relationship instance** is an ordered pair of a specific student and the student's particular physics team, such as (Emanuel Vagas, Phys201F2005A04), where Phys201F2005A04 is Emanuel's team number.

We use a diamond to illustrate the relationship type in an ER diagram. We arrange the diagram so that the relationship reads from left to right, "a student is a member of a team." Alternatively, we can arrange the components from top to bottom.

ER diagram notation for relationship type, *Member Of*

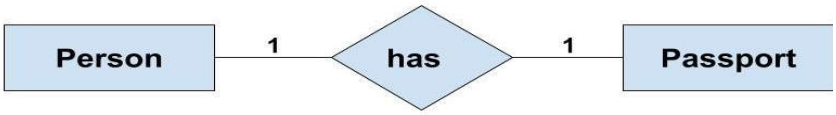
Types:

- One-to-One Relationship
- One-to-Many or Many-to-One Relationship
- Many-to-Many Relationship

One-to-one Relationship

Such a relationship exists when each record of one table is related to only one record of the other table.

For example, If there are two entities 'Person' (Id, Name, Age, Address) and 'Passport'(Passport_id, Passport_no). So, each person can have only one passport and each passport belongs to only one person.

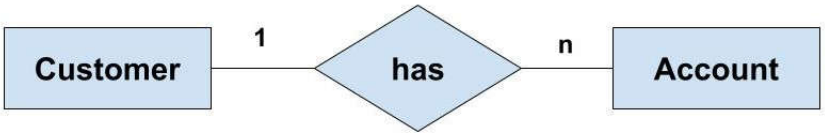


Such a relationship is not very common. However, such a relationship is used for security purposes. In the above example, we can easily store the passport id in the 'Person' table only. But, we make another table for the 'Passport' because Passport number may be sensitive data and it should be hidden from certain users. So, by making a separate table we provide extra security that only certain database users can see it.

One-to-Many or Many-to-One Relationship

Such a relationship exists when each record of one table can be related to one or more than one record of the other table. This relationship is the most common relationship found. A one-to-many relationship can also be said as a many-to-one relationship depending upon the way we view it.

For example, If there are two entity type 'Customer' and 'Account' then each 'Customer' can have more than one 'Account' but each 'Account' is held by only one 'Customer'. In this example, we can say that each Customer is associated with many Accounts. So, it is a one-to-many relationship. But, if we see it the other way i.e. many Account is associated with one Customer then we can say that it is a many-to-one relationship.



Many-to-Many Relationship

Such a relationship exists when each record of the first table can be related to one or more than one record of the second table and a single record of the second table can be related to one or more than one record of the first table. A many-to-many relationship can be seen as a two one-to-many relationship which is linked by a 'linking table' or 'associate table'. The linking table links two tables by having fields which are the primary key of the other two tables. We can understand this with the following example.

Example: If there are two entities type ‘Customer’ and ‘Product’ then each customer can buy more than one product and a product can be bought by many different customers.

Entity-relationship diagram

In order to begin constructing the basic model, the modeler must analyze the information gathered during the requirement analysis for the purpose of:

- classifying data objects as either entities or attributes,
- identifying and defining relationships between entities,
- naming and defining identified entities, attributes, and relationships,
- Documenting this information in the data document.
- Finally draw its ER diagram.

To accomplish these goals the modeler must analyze narratives from users, notes from meeting, policy and procedure documents, and, if lucky, design documents from the current information system.

Also some more points to be followed while drawing E-R diagram is as follows:-

- Every entity must be represented in the model
- Every entity must have at least one relationship
- Every entity must be a unique identifier.
- As far as possible, many to many relationship must be avoided.

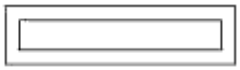
E-R diagrams constructs

In E-R diagrams, entity types are represented by squares. See the table below. Relationship types are shown in diamond shaped boxes attached to the participating entity types with straight lines. Attributes are shown in ovals, and each attribute is attached to its entity type or relationship type by a straight line. Multi valued attributes are shown in double ovals. Key attributes have their names underlined. Derived attributes are shown in dotted ovals.

Weak entity types are distinguished by being placed in double rectangles and by having their identifying relationship placed in double diamonds.

Attaching a 1, M, or N on each participating edge specifies cardinality ratio of each binary relationship type. The participation constraint is specified by a single line for partial participation and by double lines for total participation. The participation constraints specify whether the existence of an entity depends on its being related to another entity via the relationship type. If every entity of an entity set is related to some other entity set via a relationship type, then the participation of the first entity type is total. If only few member of an entity type is related to some entity type via a relationship type, the participation is partial.

ENTITY TYPE

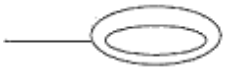


WEAK ENTITY TYPE

RELATIONSHIP TYPE

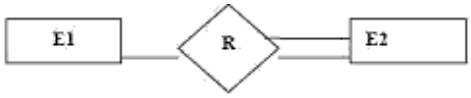
ATTRIBUTE

KEY ATTRIBUTE



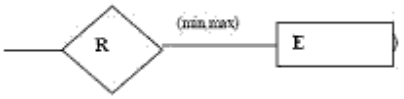
MULTIVALUED ATTRIBUTE

DERIVED ATTRIBUTE



TOTAL PARTICIPATION OF E2 IN R

Cardinality Ratio 1:N FOR E1:E2 IN R



Structural Constraint(Min,Max) On Participation Of E In R

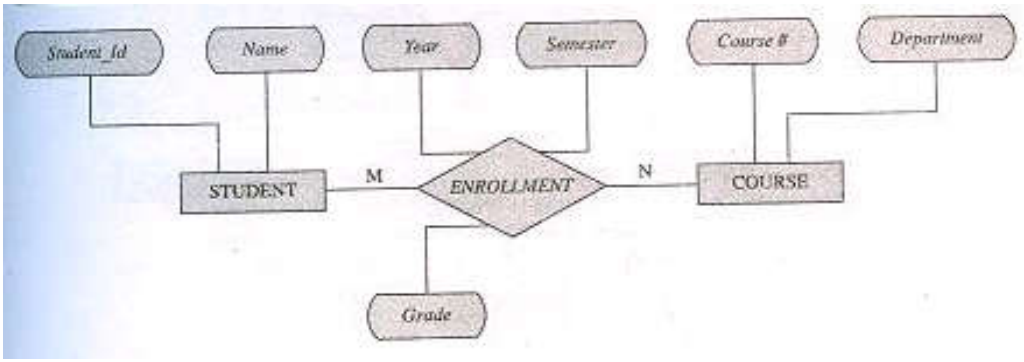
Converting an attribute association to a relationship

Shown in above figure In this case, the instances of the entity from the set DEPENDENTS are distinguishable only by their relationship with an instance of an entity from the entity set EMPLOYEE. The relationship set DEDUCTIONS is an example of an identifying relationship and the entity set DEPENDENTS is an example of a weak entity.

Instances of weak entity sets associated with the same instance of the strong entity must be distinguishable from each other by a subset of the attributes of the weak entity (the subset may be the entire weak entity). This subset of attributes is called the discriminator of the weak entity set.

The primary key of a weak entity set is thus formed by using the primary key of the strong entity set to which it is related, along with the discriminator of the weak entity.

A Binary relationship between different entity sets:



Fig(1) Binary Relationship Between two distinct entity sets

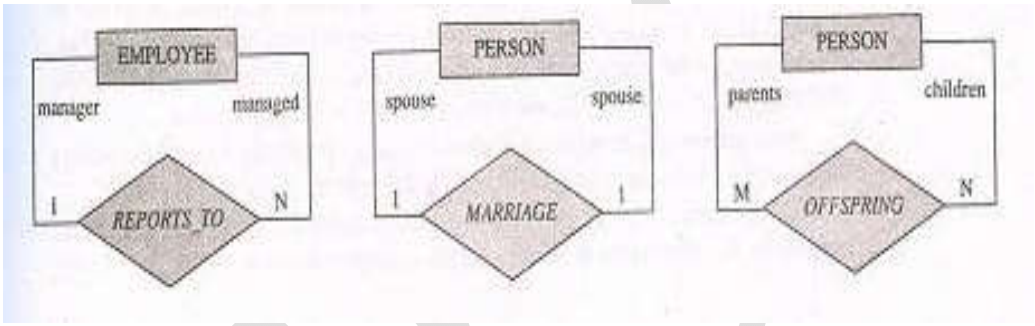


Fig (2) Relationship Between same entity sets

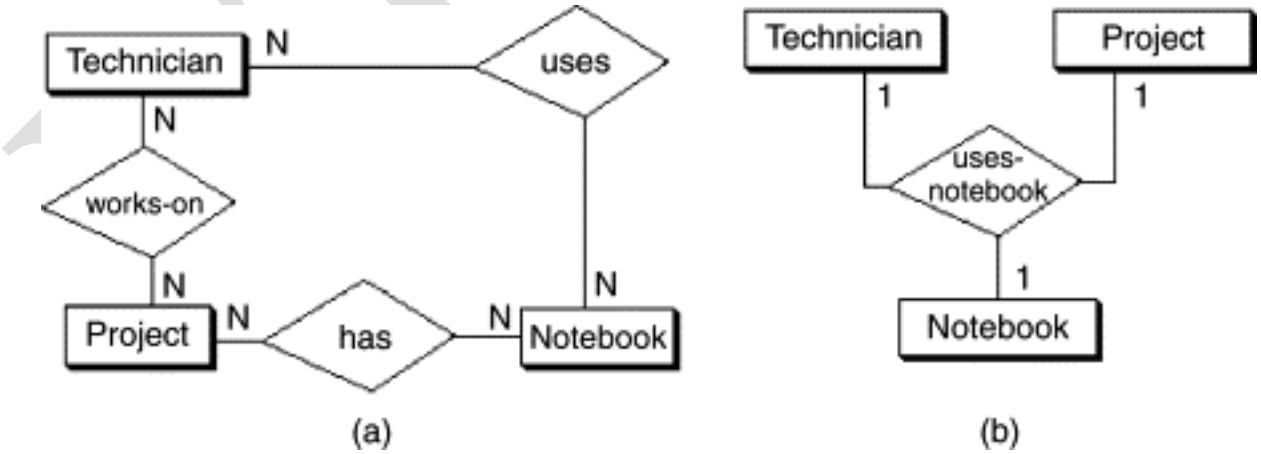


Fig (3) Ternary Relationship Between Three Entities Sets.

An ER diagram for a BANK database schema.