**UNIT – 1**

**Database**

Name itself indicates what it is. Database is a place/container where all the data is stored. But what is data? In a database, even a smallest piece of information becomes data. For example, Student is a data, course is a data, and Color is a data, height, weight, food everything is data. In short, all the living and non-living objects in this world is a data.

**Why we need data?**

We need data so that we can perform various actions on them. Say, we do not have any database and we want to record what is the height and weight of a baby in a year. What we do is note it in a piece of paper every month. At the end of the certain period or year, we would check up if he/she is growing correctly. If some entry is wrong or irrelevant, we correct it or strike it off. Same is done using database. We would be storing all these information in the database. If we want to check the growth, we would be pulling the information from the database, if we need to change any information, we can update/delete them. But all the data will be at one place - Database.

**What type of data is stored in database?**

In a database, we would be grouping only related data together and storing them under one group name called table. This helps in identifying which data is stored where and under what name. It reduces the time to search for a particular data in a whole database. For example, Student, Teacher, Class, Subject, Employee, Department etc form individual tables.

**And for whom these datas are stored?**

We store only related data - related to one particular requirement / application. For example, Student database - it will have all the information of students ranging from his ID, Name, Date of birth, class, to grade, prizes who are studying in a particular College.

**How do we determine which data is relevant to be put in a particular database?**

It all depends on what database we are developing, and what is the exact requirement/purpose of it. Say, we need to create College database. What could college database contain? First thing is that we need to store college information like its name, address. Next comes courses offered in that college, Staffs and their details, students and their details. But do we store all these information under one table - College? Will database be quick in getting the data or updating? Certainly Not! It would become a chaos if everything is stored in a single table. Hence they introduce certain rules to manage the database - relational database management system (RDBMS). RDBMS is a program that guides us how to create and maintain a database. It tells us how to divide related information into different tables and inter-relate them so that we can select/insert/update/delete all the related data easily and efficiently.

**DEFINITION OF DBMS**

DBMS is software which is used to manage the collection of interrelated data.

**Database System Applications**

* DBMS contains information about a particular enterprise
  + Collection of interrelated data
  + Set of programs to access the data
  + An environment that is both *convenient* and *efficient* to use
* Database Applications:
  + Banking: all transactions
  + Airlines: reservations, schedules
  + Universities: registration, grades
  + Sales: customers, products, purchases
  + Online retailers: order tracking, customized recommendations
  + Manufacturing: production, inventory, orders, supply chain

Human resources: employee records, salaries, tax deductions

**File systems Vs DBMS:**

The typical file processing system is supported by the operating systems. Files are created and manipulated by writing programs so the permanent records are stored in various files. Before the

advent of DBMS, organizations typically stored the information using such systems.

Ex: Using COBOL we can maintain several files (collection of records) to access those files we

have to go through the application programs which have written for creating files, updating file,

inserting the records

The problems in file processing system are

File processing system is good when there is only limited number of files and data in are very less. As the data and files in the system grow, handling them becomes difficult.

Data Mapping and Access: - Although all the related informations are grouped and stored in different files, there is no mapping between any two files. i.e.; any two dependent files are not linked. Even though Student files and Student\_Report files are related, they are two different files and they are not linked by any means. Hence if we need to display student details along with his report, we cannot directly pick from those two files. We have to write a lengthy program to search Student file first, get all details, then go Student\_Report file and search for his report.

When there is very huge amount of data, it is always a time consuming task to search for particular information from the file system. It is always an inefficient method to search for the data.

**Data Redundancy**: - There are no methods to validate the insertion of duplicate data in file system. Any user can enter any data. File system does not validate for the kind of data being entered nor does it validate for previous existence of the same data in the same file. Duplicate data in the system is not appreciated as it is a waste of space, and always lead to confusion and mishandling of data. When there are duplicate data in the file, and if we need to update or delete the record, we might end up in updating/deleting one of the record, leaving the other record in the file. Again the file system does not validate this process. Hence the purpose of storing the data is lost.

Though the file name says Student file, there is a chance of entering staff information or his report information in the file. File system allows any information to be entered into any file. It does not isolate the data being entered from the group it belongs to.

**Data Dependence**: - In the files, data are stored in specific format, say tab, comma or semicolon. If the format of any of the file is changed, then the program for processing this file needs to be changed. But there would be many programs dependent on this file. We need to know in advance all the programs which are using this file and change in the entire place. Missing to change in any one place will fail whole application. Similarly, changes in storage structure, or accessing the data, affect all the places where this file is being used. We have to change it entire programs. That is smallest change in the file affect all the programs and need changes in all them.

**Data inconsistency**: - Imagine Student and Student\_Report files have student’s address in it, and there was a change request for one particular student’s address. The program searched only Student file for the address and it updated it correctly. There is another program which prints the student’s report and mails it to the address mentioned in the Student\_Report file. What happens to the report of a student whose address is being changed? There is a mismatch in the actual address and his report is sent to his old address. This mismatch in different copies of same data is called data inconsistency. This has occurred here, because there is no proper listing of files which has same copies of data.

**Data Isolation**: - Imagine we have to generate a single report of student, who is studying in particular class, his study report, his library book details, and hostel information. All these informations are stored in different files. How do we get all these details in one report? We have to write a program. But before writing the program, the programmer should find out which all files have the information needed, what is the format of each file, how to search data in each file etc. Once all these analysis is done, he writes a program. If there is 2-3 files involved, programming would be bit simple. Imagine if there is lot many files involved in it? It would be require lot of effort from the programmer. Since all the datas are isolated from each other in different files, programming becomes difficult.

**Security**: - Each file can be password protected. But what if have to give access to only few records in the file? For example, user has to be given access to view only their bank account information in the file. This is very difficult in the file system.

**Integrity**: - If we need to check for certain insertion criteria while entering the data into file it is not possible directly. We can do it writing programs. Say, if we have to restrict the students above age 18, then it is by means of program alone. There is no direct checking facility in the file system. Hence these kinds of integrity checks are not easy in file system.

**Atomicity**: - If there is any failure to insert, update or delete in the file system, there is no mechanism to switch back to the previous state. Imagine marks for one particular subject needs to be entered into the Report file and then total needs to be calculated. But after entering the new marks, file is closed without saving. That means, whole of the required transaction is not performed. Only the totaling of marks has been done, but addition of marks not being done. The total mark calculated is wrong in this case. Atomicity refers to completion of whole transaction or not completing it at all. Partial completion of any transaction leads to incorrect data in the system. File system does not guarantee the atomicity. It may be possible with complex programs, but introduce for each of transaction costs money.

**Concurrent Access**: - Accessing the same data from the same file is called concurrent access. In the file system, concurrent access leads to incorrect data. For example, a student wants to borrow a book from the library. He searches for the book in the library file and sees that only one copy is available. At the same time another student also, wants to borrow same book and checks that one copy available. First student opt for borrow and gets the book. But it is still not updated to zero copy in the file and the second student also opt for borrow! But there are no books available. This is the problem of concurrent access in the file system.

Describing data : Levels of Abstraction

• Database Schema: The description of a database. Includes descriptions of the database structure and the constraints that should hold on the database.

• Schema Diagram: A diagrammatic display of (some aspects of) a database schema.

The data in a DBMS is described at three levels of

Defines DBMS schemas at three levels:

o Internal schema at the internal level to describe physical storage structures and access paths. Typically uses a physical data model.( how a record (e.g., customer) is stored-Physical level)

o Conceptual schema at the conceptual level to describe the structure and constraints for the whole database for a community of users. Uses a conceptual or an implementation data model.( describes data stored in database, and the relationships among the data – Logical level)

o External schemas at the external level to describe the various user views. Usually uses the same data model as Data Base Management Systems the conceptual level.(describes data as seen by a user/application – View Level)

o Schema – describes contents of the database

o e.g., what information about a set of customers and accounts and the relationship between them)

o **Physical schema**: how data is stored at physical level (how)

o **Logical schema**: data contained at the logical level (what)

o **Database Instance**: The actual data stored in a database at a particular moment in time. Also called database state (or occurrence).

**Data Independence** :

When a schema at a lower level is changed, only the mappings between this schema and higher-level schemas need to be

changed in a DBMS that fully supports data independence. The higher-level schemas themselves are unchanged. Hence, the application programs need not be changed since they refer to the external schemas

 **Physical Data Independence**  the ability to modify the physical schema without changing the logical schema

o Applications depend on the logical schema

o In general, the interfaces between the various levels and components should be well defined so that changes in some parts do not seriously influence others.

 **Logical Data Independence**  the ability to modify conceptual schema without changing the external Schema orapplication programs

**DATAMODELS**

A Database model defines the logical design and structure of a database and defines how data will be stored, accessed and updated in a database management system. While the **Relational Model** is the most widely used database model, there are other models too:

* Hierarchical Model
* Network Model
* Entity-relationship Model
* Relational Model

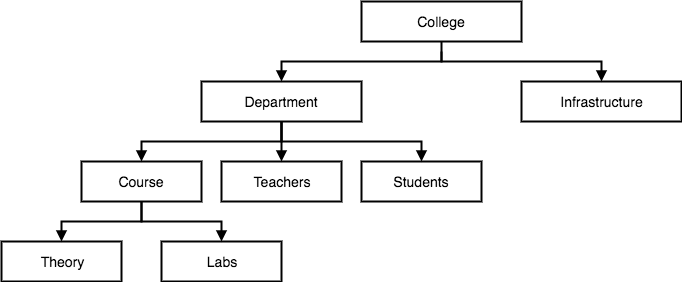
## Hierarchical Model

This database model organises data into a tree-like-structure, with a single root, to which all the other data is linked. The heirarchy starts from the **Root** data, and expands like a tree, adding child nodes to the parent nodes.

In this model, a child node will only have a single parent node.

This model efficiently describes many real-world relationships like index of a book, recipes etc.

In hierarchical model, data is organised into tree-like structure with one one-to-many relationship between two different types of data, for example, one department can have many courses, many professors and of-course many students.

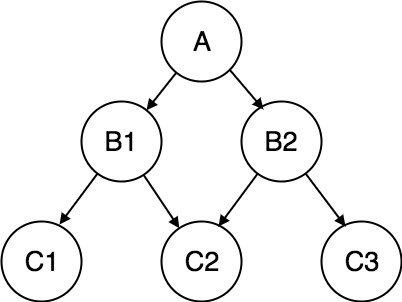


## Network Model

This is an extension of the Hierarchical model. In this model data is organised more like a graph, and are allowed to have more than one parent node.

In this database model data is more related as more relationships are established in this database model. Also, as the data is more related, hence accessing the data is also easier and fast. This database model was used to map many-to-many data relationships.

This was the most widely used database model, before Relational Model was introduced.



## Entity-relationship Model

In this database model, relationships are created by dividing object of interest into entity and its characteristics into attributes.

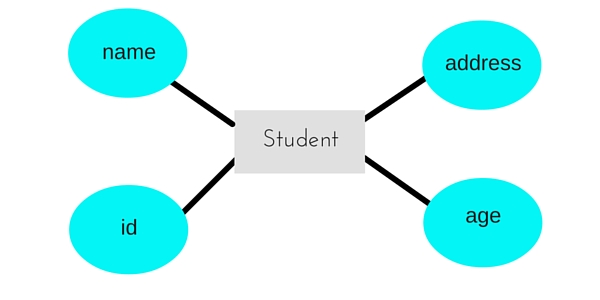
Different entities are related using relationships.

E-R Models are defined to represent the relationships into pictorial form to make it easier for different stakeholders to understand.

This model is good to design a database, which can then be turned into tables in relational model(explained below).

Let's take an example, If we have to design a School Database, then **Student** will be an **entity** with **attributes** name, age, address etc. As **Address** is generally complex, it can be another **entity** with **attributes** street name, pincode, city etc, and there will be a relationship between them.

Relationships can also be of different types. To learn about [E-R Diagrams](https://www.studytonight.com/dbms/er-diagram.php) in details, click on the link.



## Relational Model

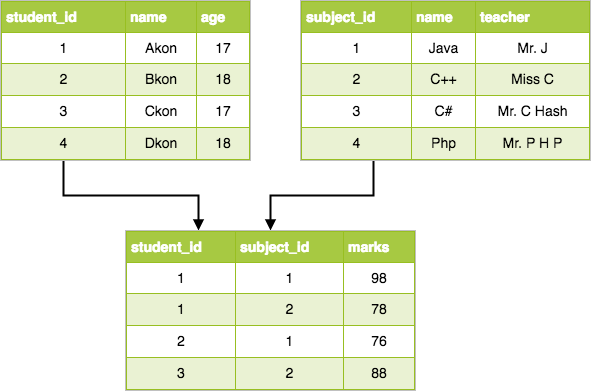
In this model, data is organised in two-dimensional **tables** and the relationship is maintained by storing a common field.

This model was introduced by E.F Codd in 1970, and since then it has been the most widely used database model, infact, we can say the only database model used around the world.

The basic structure of data in the relational model is tables. All the information related to a particular type is stored in rows of that table.

Hence, tables are also known as **relations** in relational model.

In the coming tutorials we will learn how to design tables, normalize them to reduce data redundancy and how to use Structured Query language to access data from tables.



## Types of Database languages

We need a method to create all the logical objects like tables, views, procedures and packages in the database and we need some interface between the user and the database, so that we can access the data stored in it. We also need a standardized method to organize these tables and views in the database.DBMS is software that defines different operations to be carried out in the database. It varies from creating a database, tables, index, constraints to manipulating the data in the database like inserting, deleting, updating, retrieving, sorting etc. In order to perform all these operations, DBMS defines two forms of database languages.

# ****Data Definition Language - DDL****

DDLs are used to define the metadata of the database. i.e.; using this, we create schema, tables, constraints, indexes in the database.  DDLs are also used to modify Schema, tables index etc. Basically, using DDL statements, we create skeleton of the database. It helps to store the metadata information like number of schemas and tables, their names, columns in each table, indexes, constraints etc in the database. Some of the DDL commands are

## Create

Create is used to create schema, tables, index, and constraints in the database. The basic syntax to create table is as follows.

CREATE TABLE tablename (Column1 DATATYPE, Column2 DATATYPE, … ColumnN DATATYPE);

CREATE TABLE STUDENT (STUDENT\_ID CHAR (10), STUDENT\_NAME CHAR (10));

This DDL statement creates STUDENT table with his ID and Name.

An index on the table is created as follows. These created indexes are not visible to the users, but they internally help to run the query quickly.

CREATE INDEX index\_name on Table\_name (column\_name)

CREATE INDEX std\_name ON STUDENT (STUDENT\_FIRST\_NAME, STUDENT\_LAST\_NAME);

Here std\_name is the index, which is created on First name and last name in STUDENT table.

Usually constraints are created along with table creation. Constraints are defined on the columns of the table. They define the characteristic of the column. There are different types of Constraints present.

* 1. **NOT NULL** – This constraint forces the column to have non-null value. We cannot enter/update any NULL value into such columns. It must have valid value all the time. For example, each student in STUDENT table should have class specified. No student can exist without class. Hence class column in the STUDENT table can be made NOT NULL.

CREATE TABLE STUDENT (STUDENT\_ID NUMBER (10) NOT NULL

STUDENT\_NAME VARCHAR2 (50) NOT NULL,

AGE NUMBER);

* 1. **UNIQUE** – This constraint ensures, the column will have unique value for each row. The column value will not repeat for any other rows in the table.

Passport number of individual person is unique. Hence passport column in the PERSON table is made UNIQUE. It avoids duplicate entry of passport number to other persons.

CREATE TABLE STUDENT (STUDENT\_ID NUMBER (10) NOT NULL UNIQUE

STUDENT\_NAME VARCHAR2 (50) NOT NULL,

AGE NUMBER);

OR

CREATE TABLE STUDENT (STUDENT\_ID NUMBER (10) NOT NULL,

STUDENT\_NAME VARCHAR2 (50) NOT NULL,

AGE NUMBER

CONSTRAINT uc\_StdID UNIQUE (STUDENT\_ID));

* 1. **PRIMARY KEY** – This constraint is another type of UNIQUE constraint. This constraint forces the column to have unique value and using which, we can uniquely determine each row.

As we have seen in STUDENT example, STUDENT\_ID is the primary key in STUDENT tables.

CREATE TABLE STUDENT (STUDENT\_ID NUMBER (10) NOT NULL PRIMARY KEY,

STUDENT\_NAME VARCHAR2 (50) NOT NULL,

AGE NUMBER);

OR

CREATE TABLE STUDENT (STUDENT\_ID NUMBER (10) NOT NULL,

STUDENT\_NAME VARCHAR2 (50) NOT NULL,

AGE NUMBER

CONSTRAINT pk\_StdID PRIMARY KEY (STUDENT\_ID));

* 1. **FOREIGN KEY** – This constraint helps to map two or more tables in the database. It enforces parent-child relationship in the DB. Foreign key in the child table is the column which is a primary key in the parent table.

For example, each employee works for some department. Hence to map employee and department tables, we have to have DEPARMENT\_ID of DEPARTMENT table in EMPLOYEE table too.  DEPARTMENT\_ID is the primary key in DEPARTMENT table (Parent table) and is foreign key in EMPLOYEE table (Child table).

CREATE TABLE EMPLOYEE (EMPLOYEE\_ID VARCHAR2 (10) PRIMARY KEY,

EMP\_NAME VARCHAR2 (50),

DOB DATE,

……

DEPT\_ID NUMBER

CONSTRAINT fk\_DeptId FOREIGN KEY (DEPT\_ID) REFERENCES DEPARTMENT (DEPARTMENT\_ID));

* 1. **CHECK** – This constraint is used to check for specific conditions on the column. For example, if age has to be entered between 25 and 32, we can use CHECK Constraint. This will not allow to enter the age32.

CREATE TABLE STUDENT (STUDENT\_ID NUMBER (10) NOT NULL,

STUDENT\_NAME VARCHAR2 (50) NOT NULL,

AGE NUMBER CHECK (AGE >= 25 **and** AGE<= 32));

* 1. **DEFAULT** – This constraint specifies the default value to be entered when no value is entered to it. Suppose whenever we enter an entry in the STUDENT table, apart from Student details we also have to store the date when it is being entered. This entry would always be SYSDATE. Instead of entering it each time when we do an entry, if we set the default value of this column as SYSDATE, this column will be always inserted with SYSDATE. If we need to override this value with any other date, then we have to explicitly insert the new date.

CREATE TABLE STUDENT (STUDENT\_ID NUMBER (10) NOT NULL,

STUDENT\_NAME VARCHAR2 (50) NOT NULL,

AGE NUMBER,

…..

CREATED\_DATE DATE DEFAULT SYSDATE);

## Alter

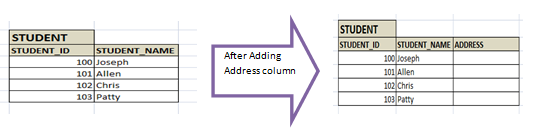
Suppose we have created a STUDENT table with his ID and Name. Later we realize that this table should have his address and Age too. What do we do at this stage? We will add the column to the existing table by the use of ALTER command.

**To add a new column:** ALTER TABLE table\_name ADD column\_name datatype;  
**To delete a column:** ALTER TABLE table\_name DROP COLUMN column\_name;  
**To modify a column:** ALTER TABLE table\_name MODIFY column\_name datatype;  
**To rename table:** ALTER TABLE table\_name RENAME TO new\_table\_name;  
**To rename the column:** ALTER TABLE table\_name RENAME COLUMN old\_Column\_name to new\_Column\_name;

Suppose we want to add Address column to the STUDENT table.

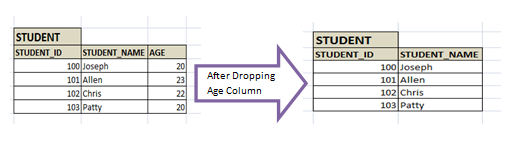
ALTER TABLE STUDENT ADD Address varchar2 (100);

Once we add new columns to the existing table, the column value for the existing data would be NULL. If we need value in them, either we have to set some default value or we need to explicitly update each column with proper value.



Suppose we want to Drop Age column from STUDENT table.

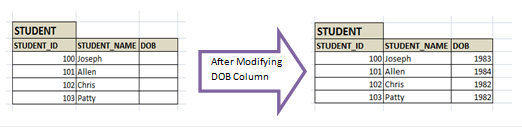
ALTER TABLE STUDENT DROP COLUMN AGE;



Once column is dropped, the entire information in the column is lost.

Suppose we want to modify the DOB column to have only year in it.

ALTER TABLE STUDENT MODIFY DOB NUMBER (4);



Note that, in order to change the one datatype to another datatype, that particular column should not have any value. If we are changing the length of the column, then we can do this with data in the column. If the length of the column is reduced, then the value in the column will be trimmed to adjust with new length. Suppose name column is modified from Varchar2 (20) to Varchar2 (10) and one of the name was ‘Albert Einstein’. After column modification, the name would be automatically trimmed to ‘Albert Ein’

Alter command is even used to modify the indexes and constraints of the table.

ALTER INDEX index\_name RENAME TO new\_index\_name;

**DROP: -** DROP statement is used to remove the table or index from the database. It can even be used to remove the database.  Once the DROP statement is executed, the object will not available for use.

DROP TABLE table\_name;

DROP INDEX index\_name;

DROP DATABASE database\_name;

DROP TABLE STUDENT;

## Truncate

Truncate statement is used to remove the content of the table, but keeps the structure of the table. This simply removes all the records from the table. No partial removal of data is possible here. It also removes all the spaces allocated for the data.

TRUNCATE TABLE table\_name;

TRUNCATE TABLE STUDENT;

## Comment

We have created some tables and columns in the database. After some days, when developer visits back these tables and columns, wonders why he has created it. He has forgotten, what was the purpose of creating those columns in the table? So what could be done in this case? We need some method to store the description of the tables and columns. Some columns would take only few values like 0 for vegetarian, 1 for eggetarian and 2 for non-vegetarian. But seeing 0, 1 or 2, we will not know what it is.  If we store those values somewhere, it would be helpful in future to enter the values. For this purpose, we use a command – COMMENT.

As the name says, it allows adding comments to tables, views and columns in the database. These comments are stored in the data dictionary of the database. This helps the developer to understand what the column is.

**Comment of Table/view:** COMMENT ON TABLE table\_name IS 'text';

**Comment of Column:** COMMENT ON COLUMN table.column IS 'text'

For Example:

COMMENT ON TABLE STUDENT IS ‘This table has all the details of Students’;

COMMENT ON COLUMN STUDENT.DOB IS 'Date of Birth of the Student'

If we need to drop the comment, then we just have to give the comment without any text.

COMMENT ON TABLE STUDENT IS ‘’;

COMMENT ON COLUMN STUDENT.DOB IS ‘’;

# ****Data Manipulation Language - DML****

When we have to insert records into table or get specific record from the table, or need to change some record, or delete some record or perform any other actions on records in the database, we need to have some media to perform it. DML helps to handle user requests. It helps to insert, delete, update, and retrieve the data from the database. Let us see some of them.

## Select

Select command helps to pull the records from the tables or views in the database. It either pulls the entire data from the table/view or pulls specific records based on the condition. It can even retrieve the data from one or more tables/view in the database.

The basic SELECT command is

SELECT \* FROM table\_name; -- retrieves all the rows **and** columns **from** table table\_name **and** displays it **in** tabular form.

SELECT COLUMN1, COLUMN2, COLUMN3 **from** table\_name; -- retrieves only 3 columns **from** table table\_name

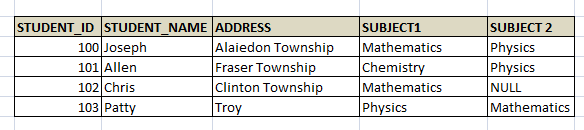
SELECT t1.COLUMN1, t2.COLUMN1

FROM table\_name1 t1, table\_name2 t2

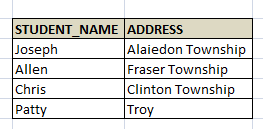
WHERE t1.COLUMN2 = t2.COLUMN2; -- Combines 2 tables **and** retrieves specific columns **from** both the tables.

Some examples of SELECT:

SELECT \* FROM STUDENT; -- All the columns are retrieved **and** displayed **in** below format



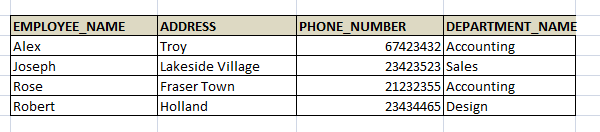
SELECT STUDENT\_NAME, ADDRESS FROM STUDENT; -- Retrieves only name **and** address **from** STUDENT table



SELECT e.EMPLOYEE\_NAME, e.ADDRESS, e.PHONE\_NUMBER, d.DEPARTMENT\_NAME

FROM EMPLOYEE e, DEPARTMENT d

WHERE E.DEPARTMENET\_ID = D.DEPARTMENT\_ID; -- Displays employee name, address, phone number **and** the department name **for** which he works, **by** joining EMPLOYEE **and** DEPARTMENT tables.



General syntax of SELECT is:

SELECT column\_list FROM table-name

[WHERE Clause]

[GROUP BY clause]

[HAVING clause]

[ORDER BY clause];

Where

**WHERE Clause** - here we can specify the filter conditions to the query. We can add any number of conditions.

**GROUP BY** - We can combine specific categories of columns together and show the results. For example, there are multiple employees working in different department. Using group by option we can display how many employees are working in each department.

SELECT d.DEPATMENT\_NAME, COUNT (e.DEPATMENT\_ID) total\_emp\_count

FROM EMPLOYEE e, DEPARTMENT d

WHERE e.DEPARTMENT\_ID = d.DEPARTMENT\_ID – **this** condition pulls the matching employees

GROUP BY e.DEPARTMENT\_ID;

In the above query instead of count (e.DEPARTMENT\_ID), we can give count (1). Both are same.

Some of the Group functions are

**Count** - it counts the total number of records in the table/s after applying ‘where’ clause. If where clause is not specified, it gives the total number of records in the table. If Group by clause is applied, it filters the records based on where clause (if any), then groups the records based on the columns in group by clause and gives the total count of records in each grouped category.

**SUM** – It totals the value in each numeric column. We can use this function to find the total marks of a student, total salary of an employee in a specific period etc.

**AVG** – It gives the average value of a column, provided column has numeric value. E.g.: Average age of students present in particular class.

**MAX** – It gives the maximum value in a column. For example, highest scorer in the class can be retrieved by MAX function.

**MIN**– It gives the minimum value in a column. For example, lowest paid employee in a department can be obtained by MIN.

These group functions can be used with group by clause or without it.

**Having Clause** - Using this clause we can add conditions to the grouped categories and filter the records. For examples, if we want to display the details of the department which has more than 100 employees.

SELECT d.DEPATMENT\_NAME, COUNT (e.DEPATMENT\_ID) total\_emp\_count

FROM EMPLOYEE e, DEPARTMENT d

WHERE e.DEPARTMENT\_ID = d.DEPARTMENT\_ID – **this** condition pulls the matching employees

GROUP BY e.DEPARTMENT\_ID

HAVING COUNT(e.DEPATMENT\_ID)>100; -- filters more than 100 employees present **in** specific department. We cannot give **this** condition **in** the **where** clause **as** **this** **is** the result of Group **by** clause.

**ORDER BY** - This clause helps to sort the records that are retrieved. By default, it displays the records in ascending order of primary key. If we need to sort it based on different columns, then we need to specify it in ORDER BY clause. If we need to order by descending order, then DESC keyword has to be added after the column list.

SELECT \* FROM EMPLOYEE

ORDER BY EMPLOYEE\_NAME DESC; -- Displays all records **in** **descending** order of employee name

We can combine two or more columns into one column by using || in select statement.

SELECT EMP\_FIRST\_NAME || ‘  ‘ || EMP\_LAST\_NAME  **as** emp\_name

FROM EMPLOYEE WHERE EMP\_ID = 1001; -- Here first name **and** **last** name of employee **with** id 1001 to show it **as** emp\_name

## Insert

Insert statement is used to insert new records into the table. The general syntax for insert is as follows:

INSERT INTO TABLE\_NAME

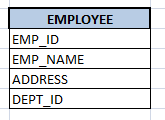
(col1, col2, col3,...colN) VALUES (value1, value2, value3,...valueN);

It inserts value to each column from 1 to N. When we insert the data in this way, we need to make sure datatypes of each column matches with the value we are inserting. Else, data will not be inserted or will insert wrong values. Also, if there is any foreign key constraint on the table, then we have to make sure foreign key value already exists in the parent table.

INSERT INTO EMPLOYEE (EMP\_ID, EMP\_NAME, ADDRESS, DEPT\_ID) VALUES

(10001, ‘Joseph’, ‘Troy’, 11101); -- Here dept\_id 11101 **is** a foreign key **and** it has be inserted **into** department table before inserting **into** the employee table.

We can insert the values into the table without specifying the columns, provided we enter the value in the order the table structure is.  Imagine table structure for Employee is as follows and then we can insert the record without specifying column names as below:



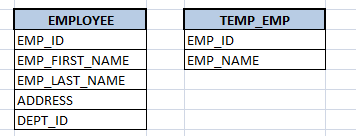
INSERT INTO EMPLOYEE VALUES (10001, ‘Joseph’, ‘Troy’, 11101);

If we are specifying the column list then we need not insert it in the order of table structure. It inserts the values in the order the column is listed in the INSERT statement.

INSERT INTO EMPLOYEE (EMP\_ID, DEPT\_ID, ADDRESS, EMP\_NAME) VALUES

(10001, 11101,’Troy’, ‘Joseph’);

We can even copy some of the column values from the existing table. Suppose we have to copy emp\_id, emp\_name from Employee table to some temporary table. Then we can write as follows: (Assuming here that datatype in both the tables are same and emp\_name has enough buffers to store both first name and last name combined).



INSERT INTO TEMP\_EMP

SELECT e.EMP\_ID, e.EMP\_FIRST\_NAME || ‘ ‘|| e.EMP\_LAST\_NAME **as** emp\_name

FROM EMPLOYEE e;

## Update

Update statement is used to modify the data value in the table. General syntax for update is as below:

UPDATE table\_name

SET column\_name1 = value1,

column\_name2 = value2,

...

column\_nameN = valueN,

[WHERE condition]

Imagine, an employee has changed his address and it needs to be updated in the Employee table.

UPDATE EMPLOYEE

SET ADDRESS = ‘Clinton Township’

WHERE EMP\_ID = 10110;

If we do not specify ‘WHERE’ condition in the ‘UPDATE’ statement, then it will update the whole table with new address. Hence we have to specify which record/employee has to be updated with new address.

Suppose there is increment in the salary of all the employees by 10% and this has to be updated in the Employee table. How will we write update statement? Do we need to specify the WHERE clause? Can we specify the arithmetic calculation in the UPDATE statement?

UPDATE EMPLOYEE SET Salary = salary+ (salary\*0.1);

This is how we update the salary.

## Delete

Using Delete statement, we can delete the records in the entire table or specific record by specifying the condition.

DELETE FROM table\_name [WHERE condition];

Suppose we have to delete an employee with id 110 from Employee table. Then the delete statement would be

DELETE FROM EMPLOYEE WHERE EMP\_ID = 110;

If we do not specify the condition, then it would delete entire record from the Employee table. This statement is different from TRUNCATE in two ways:

* Using DELETE statement, we can delete few records by specifying the condition. If we do not specify the condition, it deletes entire records in table. Whereas TRUNCATE deletes all the records in the table.
* DELETE statement simply removes records from the table, whereas TRUNCATE statement frees the space occupied by the data. Hence TRUNCATE is more efficient than DELETE, when we have to empty the table.

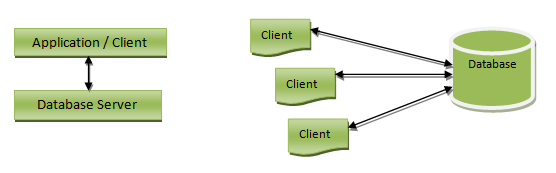
# ****Architecture of Database****

Database architecture can be 2-tier or 3 tier architecture based on how users are connected to the database to get their request done. They can either directly connect to the database or their request is received by intermediary layer, which synthesizes the request and then it sends to database.

## 2-tier Architecture

In 2-tier architecture, application program directly interacts with the database. There will not be any user interface or the user involved with database interaction. Imagine a front end application of School, where we need to display the reports of all the students who are opted for different subjects. In this case, the application will directly interact with the database and retreive all required data. Here no inputs from the user are required. This involves 2-tier architecture of the database.

Let us consider another example of two tier architecture. Consider a railway ticket reservation system. How does this work? Imagine a person is reserving the ticket from Delhi to Goa on particular day. At the same time another person in some other place of Delhi is also reserving the ticket to Goa on the same day for the same train. Now there is a requirement for two tickets, but for different persons. What will reservation system do? It takes the request from both of them, and queues the requests entered by each of them. Here the request entered to application layer and request is sent to database layer. Once the request is processed in database, the result is sent back to application layer for the user.



## Advantages of 2-tier Architecture

* Easy to understand as it directly communicates with the database.
* Requested data can be retrieved very quickly, when there is less number of users.
* Easy to modify – any changes required, directly requests can be sent to database
* Easy to maintain – When there are multiple requests, it will be handled in a queue and there will not be any chaos.

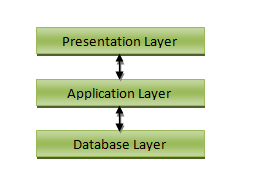
## Disadvantages of 2-tier architecture:

* It would be time consuming, when there is huge number of users. All the requests will be queued and handed one after another. Hence it will not respond to multiple users at the same time.
* This architecture would little cost effective.

## 3-tier Architecture

3-tier architecture is the most widely used database architecture. It can be viewed as below.

* **Presentation layer / User layer** is the layer where user uses the database. He does not have any knowledge about underlying database. He simply interacts with the database as though he has all data in front of him. You can imagine this layer as a registration form where you will be inputting your details.  Did you ever guessed, after pressing ‘submit’ button where the data goes? No right? You just know that your details are saved. This is the presentation layer where all the details from the user are taken, sent to the next layer for processing.
* **Application layer** is the underlying program which is responsible for saving the details that you have entered, and retrieving your details to show up in the page. This layer has all the business logics like validation, calculations and manipulations of data, and then sends the requests to database to get the actual data. If this layer sees that the request is invalid, it sends back the message to presentation layer.  It will not hit the database layer at all.
* **Data layer or Database layer** is the layer where actual database resides. In this layer, all the tables, their mappings and the actual data present. When you save you details from the front end, it will be inserted into the respective tables in the database layer, by using the programs in the application layer. When you want to view your details in the web browser, a request is sent to database layer by application layer. The database layer fires queries and gets the data. These data are then transferred to the browser (presentation layer) by the programs in the application layer.



## Advantages of 3-tier architecture:

* Easy to maintain and modify. Any changes requested will not affect any other data in the database. Application layer will do all the validations.
* Improved security. Since there is no direct access to the database, data security is increased. There is no fear of mishandling the data. Application layer filters out all the malicious actions.
* Good performance. Since this architecture cache the data once retrieved, there is no need to hit the database for each request. This reduces the time consumed for multiple requests and hence enables the system to respond at the same time.

## Disadvantages 3-tier Architecture

Disadvantages of 3-tier architecture are that it is little more complex and little more effort is required in terms of hitting the database.

***A Brief History of Database Management***

A Database Management System allows a person to organize, store, and retrieve data from a computer. It is a way of communicating with a computer’s “stored memory.” In the very early years of computers, “punch cards” were used for input, output, and data storage. Punch cards offered a fast way to enter data, and to retrieve it. Herman Hollerith is given credit for adapting the punch cards used for weaving looms to act as the memory for a mechanical tabulating machine, in 1890. Much later, databases came along.

Databases (or DBs) have played a very important part in the recent evolution of computers. The first computer programs were developed in the early 1950s, and focused almost completely on coding languages and algorithms. At the time, computers were basically giant calculators and data (names, phone numbers) was considered the leftovers of processing information. Computers were just starting to become commercially available, and when business people started using them for real-world purposes, this leftover data suddenly became important.

Enter the Database Management System (DBMS). A database, as a collection of information, can be organized so a Database Management System can access and pull specific information.  In 1960, Charles W. Bachman designed the Integrated Database System, the “first” DBMS. IBM, not wanting to be left out, created a database system of their own, known as IMS. Both database systems are described as the forerunners of [navigational databases](http://dbmswork.blogspot.in/2008/03/navigational-dbms.html).

By the mid-1960s, as computers developed speed and flexibility, and started becoming popular, many kinds of general use database systems became available. As a result, customers demanded a standard be developed, in turn leading to Bachman forming the Database Task Group. This group took responsibility for the design and standardization of a language called Common Business Oriented Language (COBOL). The Database Task Group presented this standard in 1971, which also came to be known as the “CODASYL approach.”

The CODASYL approach was a very complicated system and required substantial training. It depended on a “manual” navigation technique using a linked data set, which formed a large network. Searching for records could be accomplished by one of three techniques:

* Using the primary key (also known as the CALC key)
* Moving relationships (also called sets) to one record from another
* Scanning all records in sequential order

Eventually, the CODASYL approach lost its popularity as simpler, easier-to-work-with systems came on the market.

[Edgar Codd](https://www.theregister.co.uk/2013/08/19/ted_codd_90_relational_daddy/) worked for IBM in the development of hard disk systems, and he was not happy with the lack of a search engine in the CODASYL approach, and the IMS model. He wrote a series of papers, in 1970, outlining novel ways to construct databases. His ideas eventually evolved into a paper titled, [A Relational Model of Data for Large Shared Data Banks,](http://www.morganslibrary.net/files/codd-1970.pdf) which described new method for storing data and processing large databases. Records would not be stored in a free-form list of linked records, as in CODASYL navigational model, but instead used a “table with fixed-length records.”

IBM had invested heavily in the IMS model, and wasn’t terribly interested in Codd’s ideas. Fortunately, some people who didn’t work for IBM “were” interested. In 1973, Michael Stonebraker and Eugene Wong (both then at UC Berkeley) made the decision to research relational database systems. The project was called INGRES (Interactive Graphics and Retrieval System), and successfully demonstrated a relational model could be efficient and practical. INGRES worked with a query language known as QUEL, in turn, pressuring IBM to develop SQL in 1974, which was more advanced (SQL became ANSI and OSI standards in 1986 1nd 1987). SQL quickly replaced QUEL as the more functional query language.

[RDBM Systems](http://searchdatamanagement.techtarget.com/feature/A-look-at-the-leading-operational-database-management-systems) were an efficient way to store and process structured data. Then, processing speeds got faster, and “unstructured” data (art, photographs, music, etc.) became much more common place. Unstructured data is both non-relational and schema-less, and Relational Database Management Systems simply were not designed to handle this kind of data.

# Introduction to Database Design

### Major Steps in Database Design

1. **Requirements Analysis:** Talk to the potential users! Understand what data is to be stored, and what operations and requirements are desired.
2. **Conceptual Database Design:** Develop a high-level description of the data and constraints (we will use the ER data model)
3. **Logical Database Design:** Convert the conceptual model to a schema in the chosen data model of the DBMS. For a relational database, this means converting the conceptual to a relational schema (logical schema).
4. **Schema Refinement:** Look for potential problems in the original choice of schema and try to redesign.
5. **Physical Database Design:** Direct the DBMS into choice of underlying data layout (e.g., indexes and clustering) in hopes of optimizing the performance.
6. **Applications and Security Design:** How will the underlying database interact with surrounding applications.

### Entity-Relationship Data Model (ER)

 **entity**: An entity is a real-world object or concept which is distinguishable from other objects. It may be something tangible, such as a particular student or building. It may also be somewhat more conceptual, such as CS A-341, or an email address.

 **attributes**: These are used to describe a particular entity (e.g. name, SS#, height).

 **domain**: Each attribute comes from a specified domain (e.g., name may be a 20 character string; SS# is a nine-digit integer)

 **entity set**: a collection of similar entities (i.e., those which are distinguished using the same set of attributes. As an example, I may be an entity, whereas Faculty might be an entity set to which I belong. Note that entity sets need not be disjoint. I may also be a member of Staff or of Softball Players.

 **key**: a minimal set of attributes for an entity set, such that each entity in the set can be uniquely identified. In some cases, there may be a single attribute (such as SS#) which serves as a key, but in some models you might need multiple attributes as a key ("Bob from Accounting"). There may be several possible candidate keys. We will generally designate one such key as the **primary key**.

### ER diagrams

It is often helpful to visualize an ER model via a diagram. There are many variant conventions for such diagrams; we will adapt the one used in the text.

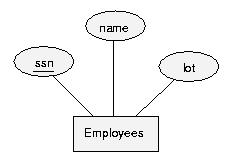
#### diagram conventions

 An entity set is drawn as a rectangle. http://cs.slu.edu/~goldwasser/class/slu/csa341/2003_Fall/lectures/design/entity.gif

 Attributes are drawn as ovals. http://cs.slu.edu/~goldwasser/class/slu/csa341/2003_Fall/lectures/design/attribute.gif

 Attributes which belong to the primary key are underlined. http://cs.slu.edu/~goldwasser/class/slu/csa341/2003_Fall/lectures/design/key.gif

#### diagram example (Figure 2.1 of text)



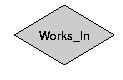
### Relationships

A **relationship** is an association among two or more entities. The relationship must be uniquely identified by the participating entities.

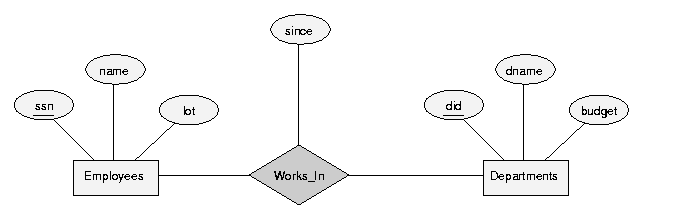
A relationship can also have **descriptive attributes**, to record additional information about the relationship (as opposed to about any one participating entity).

For example, I am an entity, as is the Department of Math/CS. A relationship exists in that I work in that department.

Similarly, a **relationship set** is a set of "similar" relationships (the similarity is based on the type of underlying entities involved in each such relationship) For example, if you have an entity set Employees and another entity set Departments, you might define a relationship set Works\_In which associates members of those two entity sets.

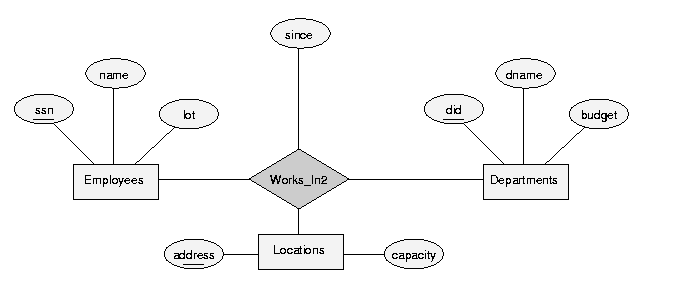
In the ER diagrams, we will draw a relationship set is drawn as a shaded diamond: 

#### Figure 2.2 of text:



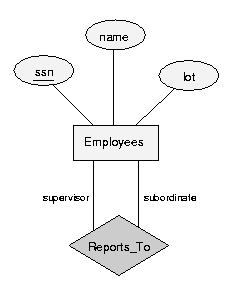
#### Ternary Relationship Set

A relationship set need not be an association of precisely two entities; it can involve three or more when applicable. Here is another example from the text, in which a store has multiple locations.



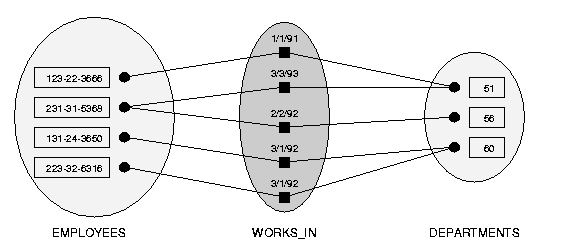
#### Using several entities from same entity set

A relationship might associate several entities from the same underlying entity set, such as in the following example, Reports\_To. In this case, an additional **role indicator** (e.g., "supervisor") is used in the diagram to further distinguish the two similar entities.



### Specifying additional constraints

If you took a 'snapshot' of the relationship set at some instant in time, we will call this an **instance**. It can be diagramed separately, as in Figure 2.3 of the text.



A (binary) relationship set can further be classified as either

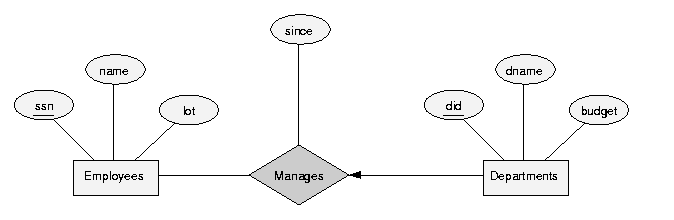
 **many-to-many**

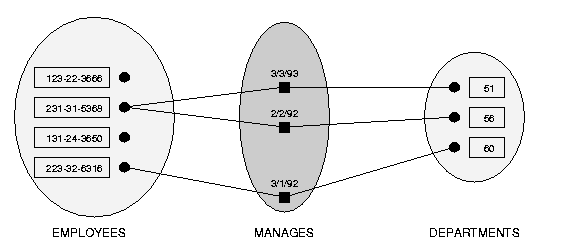
 **one-to-many**

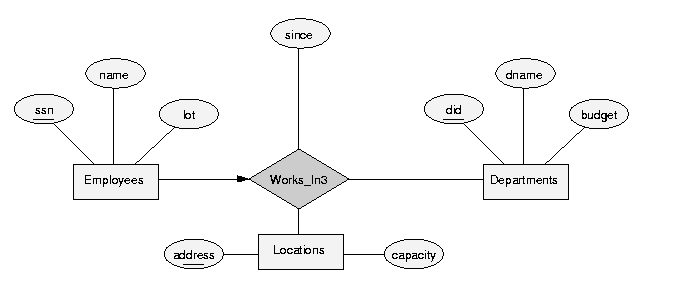
 **one-to-one**

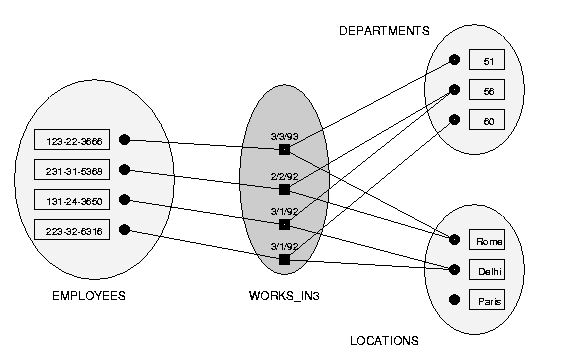
based on whether an individual entity from one of the underlying sets is allowed to be in more than one such relationship at a time. The above figure contains a many-to-many relationship, as departments may employ more than one person at a time, and an individual person may be employed by more than one department.

Sometimes, an additional constraint exists for a given relationship set, that any entity from one of the associated sets appears in at most one such relationship. For example, consider a relationship set "Manages" which associates departments with employees. If a department cannot have more than one manager, this is an example of a one-to-many relationship set (it may be that an individual manages multiple departments).

This type of constraint is called a **key constraint**. It is represented in the ER diagrams by drawing an arrow from an entity set E to a relationship set R when each entity in an instance of E appears in at most one relationship in (a corresponding instance of) R.   


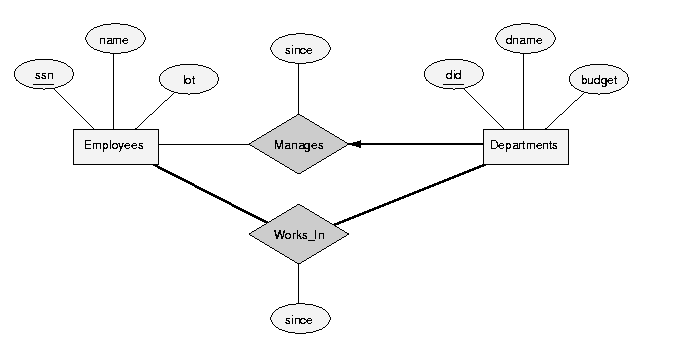
An instance of this relationship is given in Figure 2.7.   


If both entity sets of a relationship set have key constraints, we would call this a "one-to-one" relationship set. In general, note that key constraints can apply to relationships between more than two entities, as in the following example.   


An instance of this relationship:   


#### Participation Constraints

Recall that a key constraint requires that each entity of a set be required to participate in at most one relationship. Dual to this, we may ask whether each entity of a set be required to participate in at least one relationship.

If this is required, we call this a **total participation constraint**; otherwise the participation is **partial**. In our ER diagrams, we will represent a total participation constraint by using a thick line.   


#### Weak Entities

There are times you might wish to define an entity set even though its attributes do not formally contain a key (recall the definition for a key).

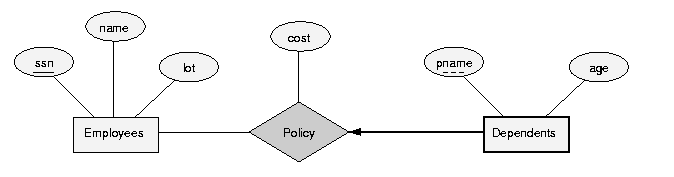
Usually, this is the case only because the information represented in such an entity set is only interesting when combined through an **identifying relationship set** with another entity set we call the **identifying owner**.

We will call such a set a **weak entity set**, and insist on the following:

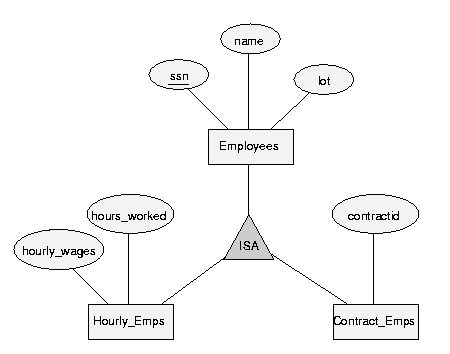
 The weak entity set must exhibit a key constraint with respect to the identifying relationship set.

 The weak entity set must have total participation in the identifying relationship set.

Together, this assures us that we can uniquely identify each entity from the weak set by considering the primary key of its identifying owner together with a **partial key** from the weak entity.

In our ER diagrams, we will represent a weak entity set by outlining the entity and the identifying relationship set with dark lines. The required key constraint and total participation are diagrammed with our existing conventions. We underline the partial key with a dotted line.   


#### Class Hierarchies

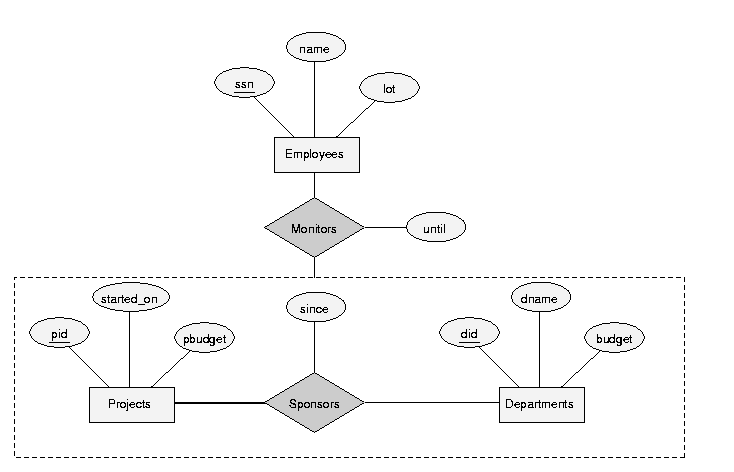
As with object-oriented programming, it is often convenient to classify an entity sets as a subclass of another. In this case, the child entity set inherits the attributes of the parent entity set. We will denote this scenario using an **"ISA"** triangle, as in the following ER diagram:   


Furthermore, we can impose additional constraints on such subclassing. By default, we will assume that two subclasses of an entity set are disjoint. However, if we wish to allow an entity to lie in more than one such subclass, we will specify an **overlap constraint**. (e.g. "Contract\_Emps OVERLAPS Senior\_Emps")

Dually, we can ask whether every entity in a superclass be required to lie in (at least) one subclass. By default we will not assume not, but we can specify a **covering constraint** if desired. (e.g. "Motorboats AND Cards COVER Motor\_Vehicles")

#### Aggregation

Thus far, we have defined relationships to be associations between two or more entities. However, it sometimes seems desirable to define a new relationship which associates some entity with some other existing relationship. To do this, we will introduce a new feature to our model called **aggregation**. We identifying an existing relationship set by enclosing it in a larger dashed box, and then we will allow it to participate in another relationship set.

A motivating example follows:   


## Conceptual Design with the ER Model

It is most important to recognize that there is more than one way to model a given situation. Our next goal is to start to compare the pros and cons of common choices.

####  Should a concept be modeled as an entity or an attribute?

 Consider the scenario, if we want to add address information to the Employees entity set? We might choose to add a single attribute address to the entity set. Alternatively, we could introduce a new entity set, Addresses and then a relationship associating employees with addresses. What are the pros and cons?

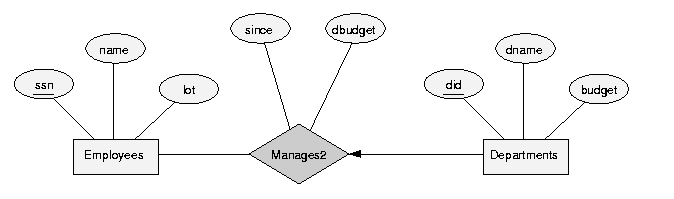
Adding a new entity set is more complex model. It should only be done when there is need for the complexity. For example, if some employees have multiple address to be associated, then the more complex model is needed. Also, representing addresses as a separate entity would allow a further breakdown, for example by zip code or city.

 What if we wanted to modify the Works\_In relationship to have both a start and end date, rather than just a start date. We could add one new attribute for the end date; alternatively, we could create a new entity set Duration which represents intervals, and then the Works\_In relationship can be made ternary (associating an employee, a department and an interval). What are the pros and cons?

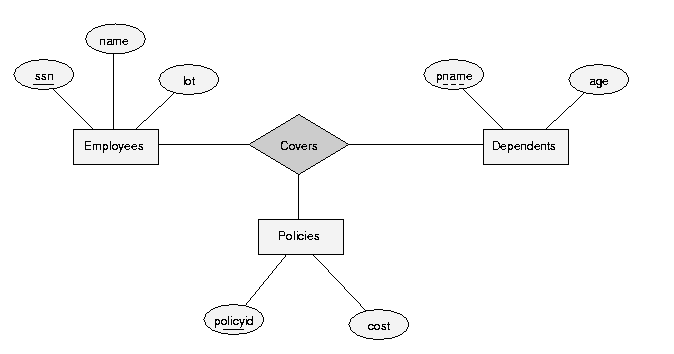
If the duration is described through descriptive attributes, only a single such duration can be modeled. That is, we could not express an employment history involving someone who left the department yet later returned.

####  Should a concept be modeled as an entity or a relationship?

Consider a situation in which a manager controls several departments. Let's presume that a company budgets a certain amount (budget) for each department. Yet it also wants managers to have access to some discretionary budget (dbudget). There are two corporate models. A discretionary budget may be created for each individual department; alternatively, there may be a discretionary budget for each manager, to be used as she desires.

Which scenario is represented by the following ER diagram? If you want the alternate interpretation, how would you adjust the model?

####  Should we use binary or ternary relationships?

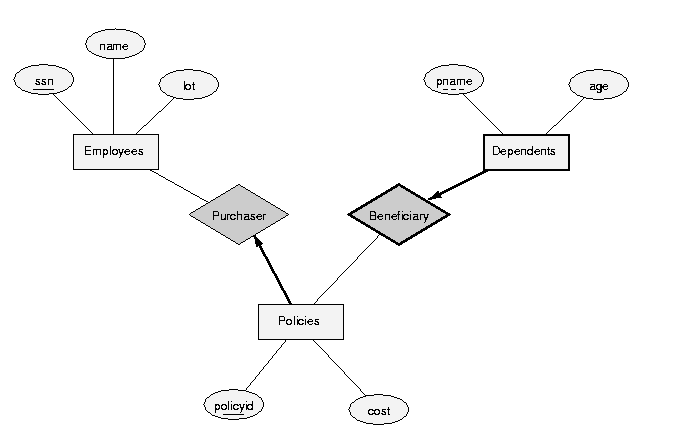
Consider the following ER diagram, representing insurance policies owned by employees at a company. Each employee can own several polices, each policy can be owned by several employees, and each dependent can be covered by several policies.   


What if we wish to model the following additional requirements:

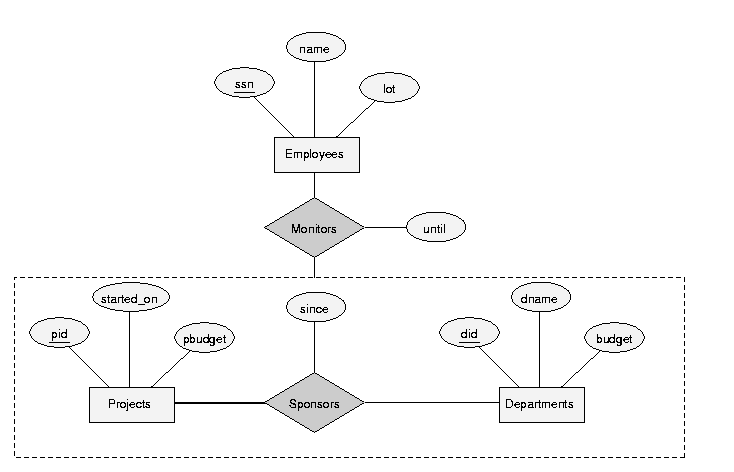
 A policy cannot be owned jointly by two or more employees.

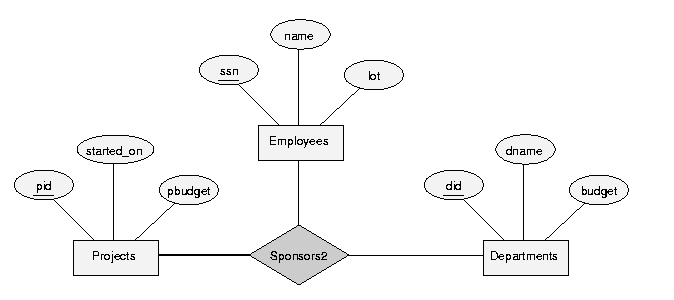
 Every policy must be owned by some employee.

 Dependents is a weak entity set, and each dependent entity is uniquely identified by taking pname in conjunction with the policyid of a policy entity (which, intuitively, covers the given dependent).

The best way to model this is to switch away from the ternary relationship set, and instead use two distinct binary relationship sets.   


####  Should we use aggregation?

Consider again the following ER diagram:   


If we did not need the until or since attributes. In tihs case, we could model the identical setting using the following ternary relationship:   


Let's compare these two models. What if we wanted to add an additional constraint to each, that each sponsorship (of a project by a department) be monitored by at most one employee. Can you add this constraint to either of the above models?

# ****Relational Set Operators in DBMS****

One of the characteristics of RDBMS is that it should support all the transaction on the records in the table by means relational operations. That means it should have strong query language which supports relational algebra. There are three main relational algebras on sets – UNION, SET DIFFERENCE and SET INTERSECT.  The same is implemented in database query language using set operators.

Relational set operators are used to combine or subtract the records from two tables. These operators are used in the SELECT query to combine the records or remove the records. In order to set operators to work in database, it should have same number of columns participating in the query and the datatypes of respective columns should be same. This is called **Union Compatibility**. The resulting records will also have same number of columns and same datatypes for the respective column.

### There are 3 main set operators used in the query language.

## UNION

It combines the similar columns from two tables into one resultant table. All columns that are participating in the UNION operation should be Union Compatible. This operator combines the records from both the tables into one. If there are duplicate values as a result, then it eliminates the duplicate. The resulting records will be from both table and distinct.

Suppose we have to see the employees in EMP\_TEST and EMP\_DESIGN tables. Suppose we don’t have UNION operator. What we will be doing is, select the records from EMP\_TEST. Copy it into some file. Then select the records from EMP\_DESIGN and copy it to the same file as previous. Thus we will get the result in one file. If we are using UNION, then it will combine both the results from tables in to one set.

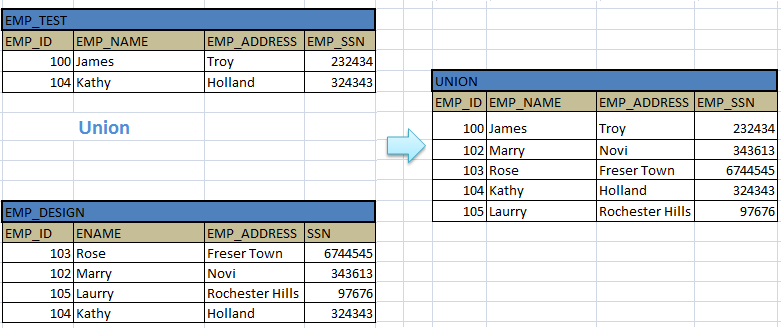
SELECT EMP\_ID, EMP\_NAME, EMP\_ADDRESS, EMP\_SSN

FROM EMP\_TEST

UNION

SELECT EMP\_ID, EMP\_NAME, EMP\_ADDRESS, EMP\_SSN

FROM EMP\_DESIGN;



We can notice that Result will have same column names as first query. Duplicate record – 104 from EMP\_TEST and EMP\_DESIGN are showed only once in the result set. Records are sorted in the result.

## UNION ALL

This operation is also similar to UNION, but it does not eliminate the duplicate records. It shows all the records from both the tables. All other features are same as UNION. We can have conditions in the SELECT query. It need not be a simple SELECT query.

Look at the same example below with UNION ALL operation.

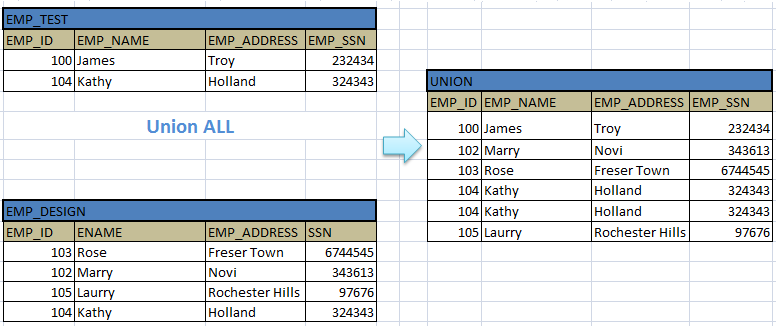
SELECT EMP\_ID, EMP\_NAME, EMP\_ADDRESS, EMP\_SSN

FROM EMP\_TEST

UNION ALL

SELECT EMP\_ID, EMP\_NAME, EMP\_ADDRESS, EMP\_SSN

FROM EMP\_DESIGN;



## INTERSECT

This operator is used to pick the records from both the tables which are common to them. In other words it picks only the duplicate records from the tables. Even though it selects duplicate records from the table, each duplicate record will be displayed only once in the result set. It should have UNION Compatible columns to run the query with this operator.

 Same example above when used with INTERSECT operator, gives below result.

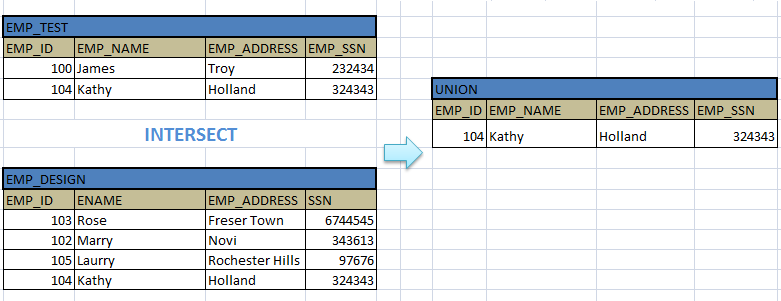
SELECT EMP\_ID, EMP\_NAME, EMP\_ADDRESS, EMP\_SSN

FROM EMP\_TEST

INTERSECT

SELECT EMP\_ID, EMP\_NAME, EMP\_ADDRESS, EMP\_SSN

FROM EMP\_DESIGN;



We have INTERSECT ALL operator too. But it is same as INTERSET. There is no difference between them like we have between UNION and UNION ALL.

## MINUS

This operator is used to display the records that are present only in the first table or query, and doesn’t present in second table / query. It basically subtracts the first query results from the second.

Let us see the same example with MINUS operator.

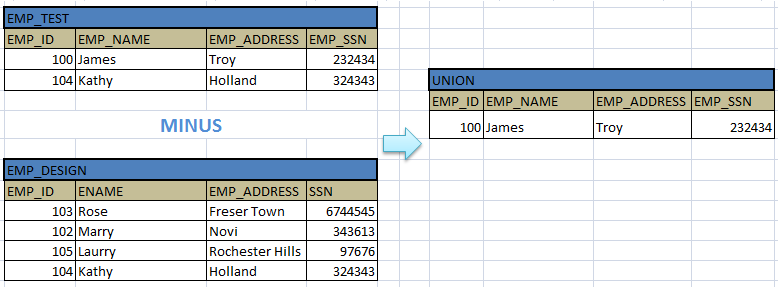
SELECT EMP\_ID, EMP\_NAME, EMP\_ADDRESS, EMP\_SSN

FROM EMP\_TEST

MINUS

SELECT EMP\_ID, EMP\_NAME, EMP\_ADDRESS, EMP\_SSN

FROM EMP\_DESIGN;



We can notice in the above result that only the records that do not exists in EMP\_DESIGN are displayed in the result. The record which appears in both the tables is eliminated. Similarly, the records that appear in second query but not in the first query are also eliminated

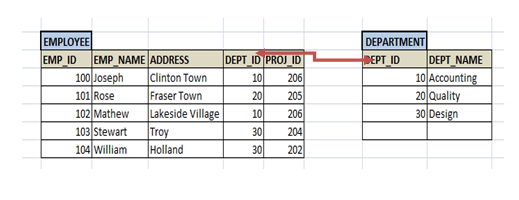
# ****Relational Models****

## Logical view of Data

In the database, all the records are stored in the physical memory in the form of bits in the files. But the user will see all these in the form of rows and columns. This user view is called logical view of the data. User will not have any information about how it will be stored in the memory. He views the data in the form of tables. This is easy for anyone to understand.

For example, STUDENT records will be scattered in the memory in various data blocks in the form of bits. But when user fetches the data, he will view it in the form of table. Different subset of records can be created out of main table and named. This named subset of records in a database is known as views. This is also can be viewed as tables.

The tables in the database are related to one another by using the mapping / relation. This is achieved by using one of the parent table columns in the child table. The parent table column in the child table is known as foreign key column.



## Constraints

Constraints are the conditions forced on the columns of the table to meet the data integrity. We have seen above what types of data integrities exists in the database. Now let see what constraints can be applied on tables so that data integrity is met.

## NOT NULL

This constraint forces the column to have non-null value. We cannot enter/update any NULL value into such columns. It must have valid value all the time. For example, each student in STUDENT table should have class specified. No student can exist without class. Hence class column in the STUDENT table can be made NOT NULL.

CREATE TABLE STUDENT (STUDENT\_ID NUMBER (10) **NOT NULL**  
STUDENT\_NAME VARCHAR2 (50) NOT NULL,  
AGE NUMBER);

## UNIQUE

This constraint ensures, the column will have unique value for each row. The column value will not repeat for any other rows in the table.

Passport number of individual person is unique. Hence passport column in the PERSON table is made UNIQUE. It avoids duplicate entry of passport number to other persons.

CREATE TABLE STUDENT (STUDENT\_ID NUMBER (10) NOT NULL **UNIQUE**  
STUDENT\_NAME VARCHAR2 (50) NOT NULL,  
AGE NUMBER);

**OR**

CREATE TABLE STUDENT (STUDENT\_ID NUMBER (10) NOT NULL,  
STUDENT\_NAME VARCHAR2 (50) NOT NULL,  
AGE NUMBER  
**CONSTRAINT uc\_StdID UNIQUE (STUDENT\_ID));**

## PRIMARY KEY

This constraint is another type of UNIQUE constraint. This constraint forces the column to have unique value and using which, we can uniquely determine each row.

As we have seen in STUDENT example, STUDENT\_ID is the primary key in STUDENT tables.

CREATE TABLE STUDENT (STUDENT\_ID NUMBER (10) NOT NULL**PRIMARY KEY**,  
STUDENT\_NAME VARCHAR2 (50) NOT NULL,  
AGE NUMBER);

**OR**

CREATE TABLE STUDENT (STUDENT\_ID NUMBER (10) NOT NULL,  
STUDENT\_NAME VARCHAR2 (50) NOT NULL,  
AGE NUMBER  
**CONSTRAINT pk\_StdID PRIMARY KEY (STUDENT\_ID));**

## FOREIGN KEY

This constraint helps to map two or more tables in the database. It enforces parent-child relationship in the DB. Foreign key in the child table is the column which is a primary key in the parent table.

For example, each employee works for some department. Hence to map employee and department tables, we have to have DEPARMENT\_ID of DEPARTMENT table in EMPLOYEE table too.  DEPARTMENT\_ID is the primary key in DEPARTMENT table (Parent table) and is foreign key in EMPLOYEE table (Child table).

CREATE TABLE EMPLOYEE (EMPLOYEE\_ID VARCHAR2 (10) PRIMARY KEY,  
EMP\_NAME VARCHAR2 (50),  
DOB DATE,  
……  
DEPT\_ID NUMBER  
**CONSTRAINT fk\_DeptId FOREIGN KEY (DEPT\_ID) REFERENCES DEPARTMENT (DEPARTMENT\_ID));**

## CHECK

This constraint is used to check for specific conditions on the column. For example, if age has to be entered between 25 and 32, we can use CHECK Constraint. This will not allow to enter the age<25 and age>32.

CREATE TABLE STUDENT (STUDENT\_ID NUMBER (10) NOT NULL,  
STUDENT\_NAME VARCHAR2 (50) NOT NULL,  
AGE NUMBER **CHECK** (AGE >= 25 and AGE<= 32));

## DEFAULT

This constraint specifies the default value to be entered when no value is entered to it. Suppose whenever we enter an entry in the STUDENT table, apart from Student details we also have to store the date when it is being entered. This entry would always be SYSDATE. Instead of entering it each time when we do an entry, if we set the default value of this column as SYSDATE, this column will be always inserted with SYSDATE. If we need to override this value with any other date, then we have to explicitly insert the new date.

CREATE TABLE STUDENT (STUDENT\_ID NUMBER (10) NOT NULL,  
STUDENT\_NAME VARCHAR2 (50) NOT NULL,  
AGE NUMBER,  
....  
CREATED\_DATE DATE **DEFAULT** SYSDATE);

## Creating Constraints and Keys in SQL

As we saw above constraints are used on the table to make sure the records and attributes entered are correct in that contest. It makes sure that there is no incorrect data being entered. If there is any mismatch or wrong data being entered, then the transaction will be rejected. How do we create all these constraints in SQL? Either they are created when we create the table for the first time or it can be constructed after table is being created.

The general syntax for creating a constraint when a table is created is as shown below.

CREATE TABLE table\_name (  
                                      Column1 **DATATYPE (SIZE) CONSTRAINT\_TYPE**,  
                                      Column2 **DATATYPE (SIZE) CONSTRAINT\_ TYPE**,  
                                      ....  
                                    ); --where CONSTRAINT\_TYPE can be NOT NULL, PRIMARY KEY, CHECK, DEFAULT etc.

Few examples of creating constraints as above is given below:

CREATE TABLE STUDENT (  
STD\_ID NUMBER (10) **NOT NULL PRIMARY KEY**,  
STD\_NAME VARCHAR2 (255) **NOT NULL**,  
ADDRESS VARCHAR2 (255),  
DATE\_OF\_BIRTH DATE); -- Example of NOT NULL and PRIMARY KEY Constraint

CREATE TABLE PERSON (  
SSN\_NUM NUMBER (10) **NOT NULL UNIQUE**,  
PERSON\_NAME VARCHAR2 (255) **NOT NULL**,  
ADDRESS VARCHAR2 (255),  
AGE NUMBER **NOT NULL CHECK (AGE>=18)**,  
LICENCE NUMBER,  
DEPT\_ID NUMBER **DEFAULT 10**); -- Example of NOT NULL, UNIQUE, DEFAULT AND CHECK Constraint

For a given column, single or multiple constraints can be applied. Constraints can be created on the existing tables as below.

**ALTER TABLE** table\_name **ADD** **CONSTRAINT** constraint\_name **CONSTRAINT\_TYPE;**

**ALTER TABLE**PERSON**ADD CONSTRAINT**checkAge**CHECK (AGE >= 18);**

**ALTER TABLE**STUDENT **ADD CONSTRAINT**stdPrimaryKey**PRIMARY KEY (**STD\_ID**);**

**ALTER TABLE**STUDENT **ADD CONSTRAINT**stdUniqueKey ***UNIQUE* (**STD\_ID**);**

**ALTER TABLE**STUDENT **ADD CONSTRAINT**stdUniqueKey ***NOT NULL* (**STD\_NAME**);**

## Drop the Constraint

**ALTER TABLE** table\_name **DROP CONSTRAINT** constraint\_name;

**ALTER TABLE**PERSON**DROP CONSTRAINT**checkAge**;**

**ALTER TABLE**STUDENT **DROP CONSTRAINT**stdPrimaryKey**;**

**ALTER TABLE**STUDENT **DROP CONSTRAINT**stdUniqueKey**;**

**ALTER TABLE**STUDENT **DROP CONSTRAINT**stdUniqueKey**;**

## Create Foreign Key Constraint

Creating a foreign key constraint is little different. In order to create a foreign key constraint, we need to have parent table created first. Then, when we create child table, we can have foreign key constraint mentioned as below.

**CREATE TABLE** table\_name (

                                            Column1**DATATYPE (SIZE),**

Column2**DATATYPE (SIZE)**

                                               ……

**CONSTRAINT**constraint\_name**FOREIGN KEY (**column\_name**) REFERENCES**parent\_table**(**parent\_column\_name**));**

### Example:

**CREATE TABLE** EMPLOYEE (EMPLOYEE\_ID VARCHAR2 (10) **PRIMARY KEY,**

EMP\_NAME VARCHAR2 (50),

DOB DATE,

……

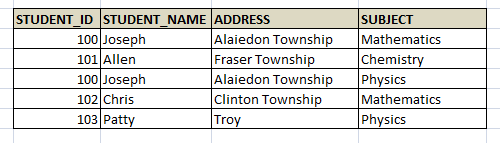
DEPT\_ID NUMBER

**CONSTRAINT**fk\_DeptId**FOREIGN KEY (DEPT\_ID) REFERENCES**DEPARTMENT**(**DEPARTMENT\_ID**)**);

# Data Integrity

When we are designing a database, there is lot of factors to be concentrated on. We need to make sure that all the required datas are distributed among right tables and there is no duplication/missing data. The space utilised for the appropriately for the database. Time taken for each query is minimal and so on.

Imagine we have a STUDENT table with Student details and the subjects that he has opted for. If we observe the table below, Joseph has opted for two subjects - Mathematics and Physics. That is fine. But what is wrong in below table? His address is repeated each time, which is not necessary and waste of space. This is called redundancy and is not allowed in a database.



Similarly, if we have to insert one more record for Allen, then we have to enter all his details into the above table. But what is the guarantee that all his details are entered correctly? There could be a mistake and hence leading to mismatch in his details. But who will later say which entry is correct? No one! Hence the data in DB is wrong.

Same issue can happen when we update the data. If we update address one of the record, and leave other record for Joseph above, again a data mismatch.

And when we delete a data, say for Chris, who is having only one entry, whole of his information is lost!

Imagine there are two entities - Employee and Department, and they are not properly related by means of foreign key. What would be the result? We can enter as many department as we want to an employee for whom department may not exists at all in Department table! So mapping the tables appropriately is also a very important factor.

Data integrity ensures, all the above mentioned issues are not injected into the database while it is designed. It guarantees that database is perfect and complete.

# Types of Integrity Constraints

**There are five types of data integrity constraints**

## Domain Constraint

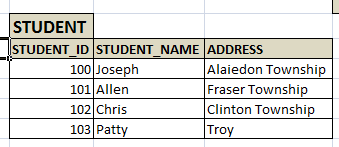
Here each columns of a table are verified so that correct data is entered into column. For example, numeric data is entered into a NUMBER column and not any character. In a DATE column, correct dates are entered and not any invalid values.

Imagine we have a table where date fields are stored as character and we have to copy this date field into a new table where this column is defined as DATE. What happens here is most of the data from the original table will not be loaded into new table, as there is mismatch in the data stored. i.e.; the original table will have dates in 22 March 2015 format which new table will not accept as date. In the foremost case, if there was domain integrity, original table would not have such dates and would have preserved the integrity of data in the original table itself.

## Entity Integrity

This integrity ensures that each record in the table is unique and has primary key which is not NULL.  That means, there is no duplicate record or information of data in a table and each records are uniquely identified by non null attribute of the table.

In a STUDENT table, each student should be a different from other and there will not be duplicate records. Also, STUDENT\_ID which is a primary key in the table has non-null values for each of the record.



## Column Constraint

This constraint ensures that the values entered into a column are correct by means of business rules. Say, there is an age column and its value is negative which is not correct. This constraint refines from entering wrong age. Similar example of such constraint is salary cannot be negative; employee number will be in a given range etc. These are business rules/requirements that specify what kind of values could be entered into each column.

This constraint is different from domain constraint as here it checks for the validity of the data being entered- like correct age is being entered; Correct Employee Id is entered etc. In the domain constraint, it checks, whether correct set of data being is entered - like Date is entered into date column, Number is entered into number column etc.

## User-Defined Integrity Constraint

Imagine, while entering a salary of an employee, we need to check if his salary is less than his manager. Though this is similar to column constraint, we cannot direct insist this constraint on the column as the system does not know who his manager is. We need to check for his manager's salary first, if it is more than his employer, then we will insert the data. For this we manually need to write code. This kind of constraints is called User-Defined Integrity Constraint.

## Referential  Integrity

As we discussed for Employee and Department tables, if they are not mapped correctly, there would be a data mismatch. It will allow us to enter a department for an employee which does not exist. It will allow us to delete a department for which employees are working. What would be the result? Employees without any department are not correct. Or updating any department number in the employee table will result in a department which does not exist at all. All these cases will lead to mismatch and invalid data in the database.

Hence to ensure above all cases are met, proper relationship has to be defined between the related tables by means of primary and foreign keys. i.e.; every foreign key in the table should be a primary key in the related table.

**Specifying Foreign Key Constraints in SQL**

CREATE TABLE Enrolled ( sid CHAR(20), cid CHAR(20), grade CHAR(10), PRIMARY KEY

(sid, cid), FOREIGN KEY (sid) REFERENCES Students )

**Enforcing Integrity Constraints**

Consider the instance S1 of Students shown in Figure 3.1. The following insertion violates the primary

key constraint because there is already a tuple with the sid 53688, and it will be rejected by the DBMS:

INSERT INTO Students (sid, name, login, age, gpa) VALUES (53688, ‘Mike’, ‘mike@ee’, 17,

3.4)

The following insertion violates the constraint that the primary key cannot contain null:

INSERT INTO Students (sid, name, login, age, gpa) VALUES (null, ‘Mike’, ‘mike@ee’, 17,

3.4)

**Querying Relational Data**

A**relational database query** is a question about the data, and the answer consists of a new

relation containing the result. For example, we might want to ﬁnd all students younger than 18 or

all students enrolled in Reggae203. A**query language** is a specialized language for writing queries.

SQL is the most popular commercial query language for a relational DBMS. Consider the instance of the

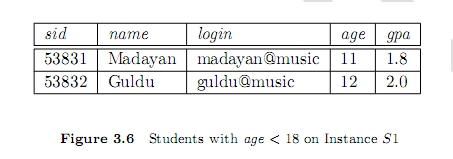
Students relation shown in Figure 3.1. We can retrieve rows corresponding to students who are younger

than 18 with the following SQL query:

SELECT \* FROM Students S WHERE S.age < 18

The symbol \* means that we retain all ﬁelds of selected tuples in the result. The condition S.age

< 18 in the WHERE clause speciﬁes that we want to select only tuples in which the age ﬁeld has

a value less than 18. This query evaluates to the relation shown in Figure 3.6.

**Introduction To Views**

A**view** is a table whose rows are not explicitly stored in the database but are computed as needed from a

**view deﬁnition**. Consider the Students and Enrolled relations. Suppose that we are often interested in

ﬁnding the names and student identiﬁers of students who got a grade of B in some course, together with

the cid for the course. We can deﬁne a view for this purpose. Using SQL notation:

CREATE VIEW B-Students (name, sid, course) AS SELECT S.sname, S.sid, E.cid FROM Students S,

Enrolled E WHERE S.sid = E.sid AND E.grade = ‘B’

**Destroying/Altering Tables and Views**

To destroy views, use the DROP TABLE command. For example, DROP TABLE Students RESTRICT

destroys the Students table unless some view or integrity constraint refers to Students; if so, the command

fails. If the keyword RESTRICT is replaced by CASCADE, Students is dropped and any referencing

views or integrity constraints are (recursively) dropped as well; one of these two keywords must always

be speciﬁed. A view can be dropped using the DROP VIEW command, which is just like DROP TABLE.

ALTER TABLE modiﬁes the structure of an existing table. To add a column called maiden-name to

Students, for example, we would use the following command:

ALTER TABLE Students ADD COLUMN maiden-name CHAR(10)

The deﬁnition of Students is modiﬁed to add this column, and all existing rows are padded with null

values in this column. ALTER TABLE can also be used to delete columns and to add or drop integrity constraints on a table.