UNIT – I

Introduction

A *database management system* (DBMS) consists of a collection of interrelated data and a set of programs to access that data. The collection of data is referred to as the *database*.

The primary goal of a DBMS is to provide an environment that is both *convenient* and *efficient* to use in retrieving and storing database information.

- Database systems are designed to manage large bodies of information.
- The management of data involves both storage of information and the manipulation of information.
- The database system must provide for the safety of the information stored, despite system crashes or attempts at unauthorized access.

1.1 Purpose of Database Systems

1.1.1 File Processing System:

The *file-processing system* is supported by a conventional operating system. Permanent records are stored in various files, and a number of different application programs are written to extract records from and add records to the appropriate files.

For example a part of a savings bank enterprise keeps information about all customers and savings accounts in permanent system files at the bank. Also, the system has a number of application programs that allow the user to manipulate the files, including:

- A program to debit or credit an account.
- A program to add a new account.
- A program to find the balance of an account.
- A program to generate monthly statements.

The application programs have been written by system programmers in response to the needs of the bank organization. New application programs are added to the system as the need arises.

1.1.2 Disadvantages of File Processing System

- **Data redundancy and inconsistency**

  Redundancy means duplication of data i.e., the same information may be duplicated in several places (files).

  Example, the address and phone number of a particular customer may appear in a file that consists of savings account records and in a file that consists of checking account records.

  Redundancy leads to higher storage and access cost and it will lead to data inconsistency (contradiction) - that is, the various copies of the same data may no longer agree.

  Example, a changed customer address may be reflected in savings account records but not elsewhere in the system.

- **Difficulty in accessing data**
  The file-processing environments do not allow needed data to be retrieved in a convenient and efficient manner. Better data retrieval systems must be developed for general use.

  Example: Suppose that one of the bank officers needs to find out the names of all customers who live within the city's 600001 zip code. The officer asks the data processing department to generate such a list. Since this request was not anticipated when the original system was designed, there is no application program on hand to meet it. There is, however, an application program to generate the list of *all* customers. The bank officer has now two choices: Either get the list of customers and extract the needed information manually, or ask the data processing department to have a system programmer write the necessary application program. Both alternatives are obviously unsatisfactory. Suppose that such a program is actually written and that, several days later, the same officer needs to trim that list to include only those customers with an account balance of `10,000 or more. As expected, a program to generate such a list does not exist. Again, the officer has the preceding two options, neither of which is satisfactory.

- **Data isolation**
  Since data is scattered in various files, and files may be in different formats, it is difficult to write new application programs to retrieve the appropriate data.

- **Integrity problems**
  The data values stored in the database must satisfy certain types of *consistency constraints.*
  Example, the balance of a bank account may never fall below a prescribed amount (say, `500). These constraints are enforced in the system by adding appropriate code in the various application programs. However, when new constraints are added, it is difficult to change the programs to enforce them.

- **Atomicity Problems**
  A computer system is subject to failure, like any other mechanical or electrical device. In many applications, it is crucial to ensure that, once a failure has occurred and has been detected, the data are restored to the consistent state that existed prior to the failure.

  Example, consider a program to transfer `50 from account A to B. If a system failure occurs during the execution of the program, it is possible that the `50 was removed from account A but was not credited to account B, resulting in an inconsistent database state. It is essential to database consistency that either both the credit and debit occur, or that neither occur. That is, the funds transfer must be *atomic* – it must happen in its entirety of not at all. It is difficult to ensure this property in a conventional file-processing system.

- **Concurrent access anomalies**
  Many systems allow multiple users to update the data simultaneously. That is an interaction of concurrent updates may result in inconsistent data. In order to guard against this possibility, some form of supervision must be maintained in the system.

Since data may be accessed by many different application programs which have not been previously coordinated, supervision is very difficult to provide.

Example: Consider bank account *A*, with `500. If two customers withdraw funds (say `50 and `100 respectively) from account *A* at about the same time, the result of the concurrent executions may leave the account in an incorrect (or inconsistent) state. In particular, the account may contain `450 or `400, rather than `350.

- **Security problems**
  Every user of the database system should not be able to access all the data. Since application programs are added to the system in an ad hoc manner, it is difficult to enforce such security constraints.
  Example, in a banking system, payroll personnel only is allowed to see that part of the database that has information about the various bank employees. They do not need access to information about customer accounts.

These difficulties, among others, have prompted the development of database management systems.

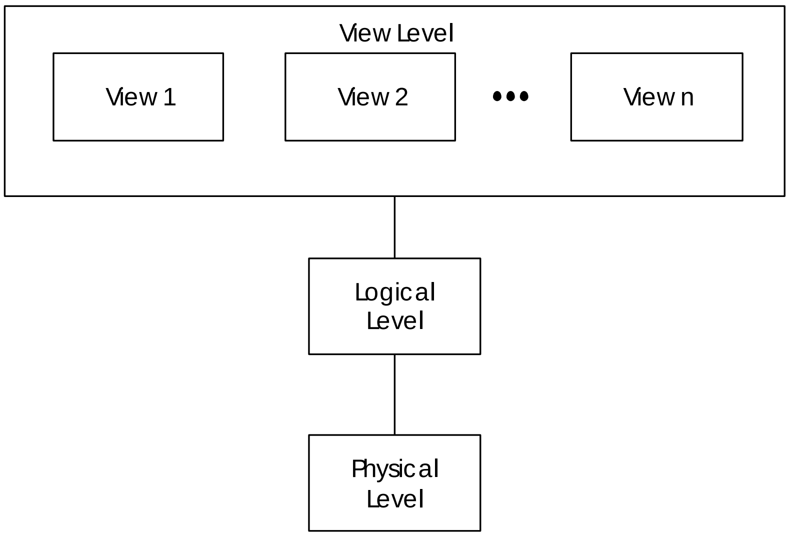
1.2 View of Data

A major purpose of a database system is to provide users with an *abstract* view of the data. The database system hides certain details of how the data is stored and maintained.

1.2.1 Data Abstraction

The database system must retrieve the data efficiently. The data represented in the database is designed with complex data structures. The complexity (how data stored and maintained) is hidden from the users through several levels of abstraction in order to simplify their interaction with the system.

There are three levels of abstraction:



The three levels of data abstraction

Physical level

Physical level is the lowest level of abstraction. This describes how the data are actually stored. Complex low-level data structures are described in detail at the physical level.

Conceptual level (or) Logical level

Logical level is the next-higher-level of abstraction. This describes what data are actually stored in the database, and the relationships that exist among the data. The entire database is described in terms of a small number of relatively simple structures. The logical level of abstraction is used by database administrators, who must decide what information is to be kept in the database.

View level

View level is the highest level of abstraction. This describes only part of the entire database.

Many users of the database system will not be concerned with all of the information. Instead, such users need only a part of the database. To simplify their interaction with the system, the view level of abstraction is defined. The system may provide many views for the same database.

Consider the following example for declaring a record data type in Pascal Language

```
type customer = record
        name: string;
        street: string;
        city: string;
        end;
```

This defines a new record called *customer* with three fields. Each field has a name and a type associated with it. A banking enterprise may have several such record types, including:

• *account,* with fields *number* and *balance.*
• *employee,* with fields *name* and *salary.*

- At the physical level, a *customer, account,* or *employee* record can be described as a block of consecutive storage locations (for example, words or bytes).
- At the logical level, each such record is described by a type definition, illustrated above, and the interrelationship among these record types is defined.
- Finally, at the view level, several views of the database are defined. For example, tellers in a bank see only that part of the database that has information on customer accounts. They cannot access information concerning salaries of employees.

1.2.2 Instances and Schemas

- The collection of information stored in the database at a particular moment in time is called an *instance* of the database.
- The overall design of the database is called the database *schema.*
- Schemas are changed infrequently, if at all.

The concept of a database schema corresponds to the programming language notion of type definition. A variable of a given type has a particular value at a given instant in time. Thus, the concept of the value of a variable in programming languages corresponds to the concept of an *instance* of a database schema.

Database systems have several schemas, partitioned according to the levels of abstraction.

The lowest level is the *physical schema;*
The intermediate level is the *logical schema;*
The highest level is a *subschema.*

The database systems support one physical schema, one logical schema, and several subschemas.

1.2.3 Data Independence

The ability to modify a schema definition in one level without affecting a schema definition in the next higher level is called *data independence.*

There are two levels of data independence:

i) Physical Data Independence
ii) Logical Data Independence

Physical data independence: Physical data independence is the ability to modify the physical schema without causing application programs to be rewritten.

Logical data independence: Logical data independence is the ability to modify the logical schema without causing application programs to be rewritten. Modifications at the logical level are necessary whenever the logical structure of the database is altered.

Logical data independence is more difficult to achieve than physical data independence since application programs are heavily dependent on the logical structure of the data they access.

1.3 Data Models

Data Model is a collection of conceptual tools for describing data, data relationships, data semantics, and consistency constraints.

There are three different groups in data models:

i) Object-based logical models
ii) Record-based logical models
iii) Physical data models.

1.3.1 Object-Based Logical Models

Object-based logical models are used in describing data at the logical and view levels. They provide fairly flexible capabilities and allow data constraints to be specified explicitly.

There are many different models,

i) The entity-relationship model
ii) The object-oriented model
iii) The semantic data model
iv) The functional data model

1.3.1.1 The Entity-Relationship Model

- An *entity* is a ‘thing’ or ‘object’ in the real world which is distinguishable from other objects.
- Entities are described in a database by a set of *attributes*.
- A *relationship* is an association among several entities.
- The entity-relationship (E-R) data model is a collection of basic objects called *entities,* and *relationships* among those objects.

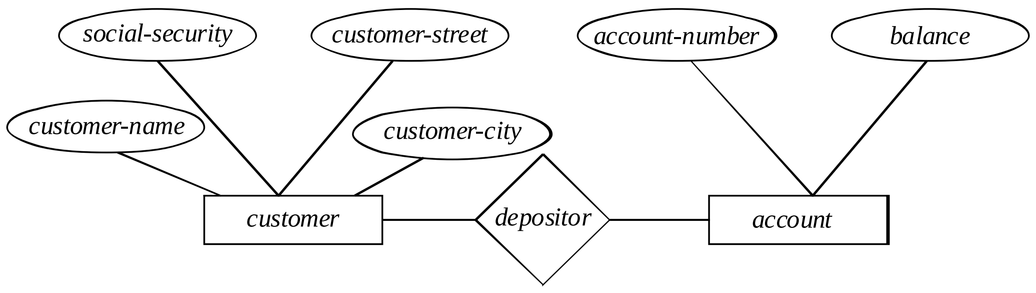
For example, the attributes *number* and *balance* describe one particular *account* in a bank.

For example, a *depositor* relationship associates a customer with each account that he or she has.

- The set of all entities of the same type and relationships of the same type are termed an *entity set* and *relationship set,* respectively.
- The number of entities to which another entity can be associated via a relationship set is *mapping cardinality.*

The overall logical structure of a database can be expressed graphically by an *E-R diagram,* which consists of the following components:

• Rectangles	- represent entity sets.
• Ellipses	- represent attributes.
• Diamonds	- represent relationships among entity sets.
• Lines	- link attributes to entity sets and entity sets to relationships.



A Sample E-R Diagram

1.3.1.2 The Object-Oriented Model

- The object-oriented model is based on a collection of objects.
- An object contains values stored in *instance variables* within the object.
- An object contains bodies of code that operate on the object.
- These bodies of code are called *methods.*
- Objects that contain the same types of values and the same methods are grouped together into *classes.*

1.3.2 Record-Based Logical Models

Record-based logical models are used in describing data at the logical and view levels. They are used both to specify the overall logical structure of the database and to provide a higher-level description of the implementation. In Record-based models the database is structured in fixed-format records. Each record type defines a fixed number of fields, or attributes, and each field is usually of fixed length.

The three most widely accepted data models are

i) The Relational Model
ii) The Network Model
iii) The Hierarchical Model

1.3.2.1 Relational Model

The relational model represents data and relationships among data by a collection of tables, each of which has a number of columns with unique names.

customer-name	*social-security*	*customer-street*	*Customer-city*	*account-number*
Johnson	192-83-7465	Alma	Palo Alto	A-101
Smith	019-28-3746	North	Rye	A-215
Hayes	677-89-9011	Main	Harrison	A-102
Turner	182-73-6091	Putnam	Stamford	A-305
Johnson	192-83-7465	Alma	Palo Alto	A-201
Jones	321-12-3123	Main	Harrison	A-217
Lindsay	336-66-9999	Park	Pittsfield	A-222
Smith	019-28-3746	North	Rye	A-201

Customer database

account-number	*Balance*
A-101	500
A-215	700
A-102	400
A-305	350
A-201	900
A-217	750
A-222	700

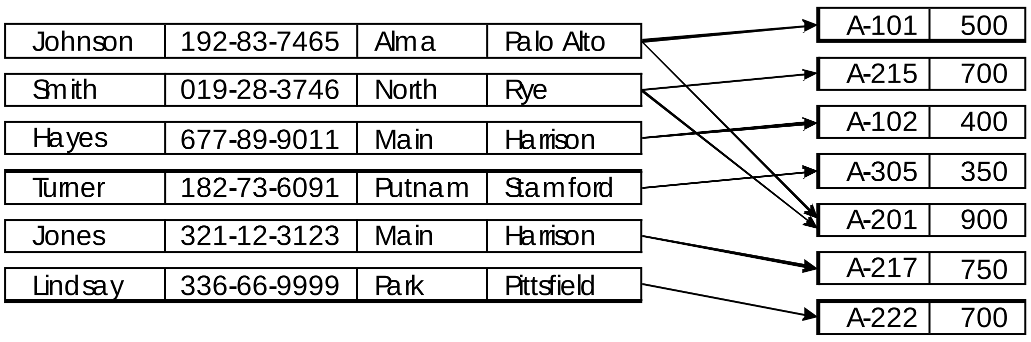
Account database

A sample Relational database

The above is a sample relational database showing customers and the accounts they have. It shows, for example, that customer Johnson, with social-security number 321-12-3123, lives on Main in Harrison, and has two accounts, one numbered A-101 with a ba1ance of `500, and the other numbered A-201 with a balance of `900. Note that customers Johnson and Smith share account number A-201 (they may share a business venture).

1.3.2.2 Network Model

Data in the network model are represented by collections of *records* and relationships among data are represented by *links,* which can be viewed as pointers.

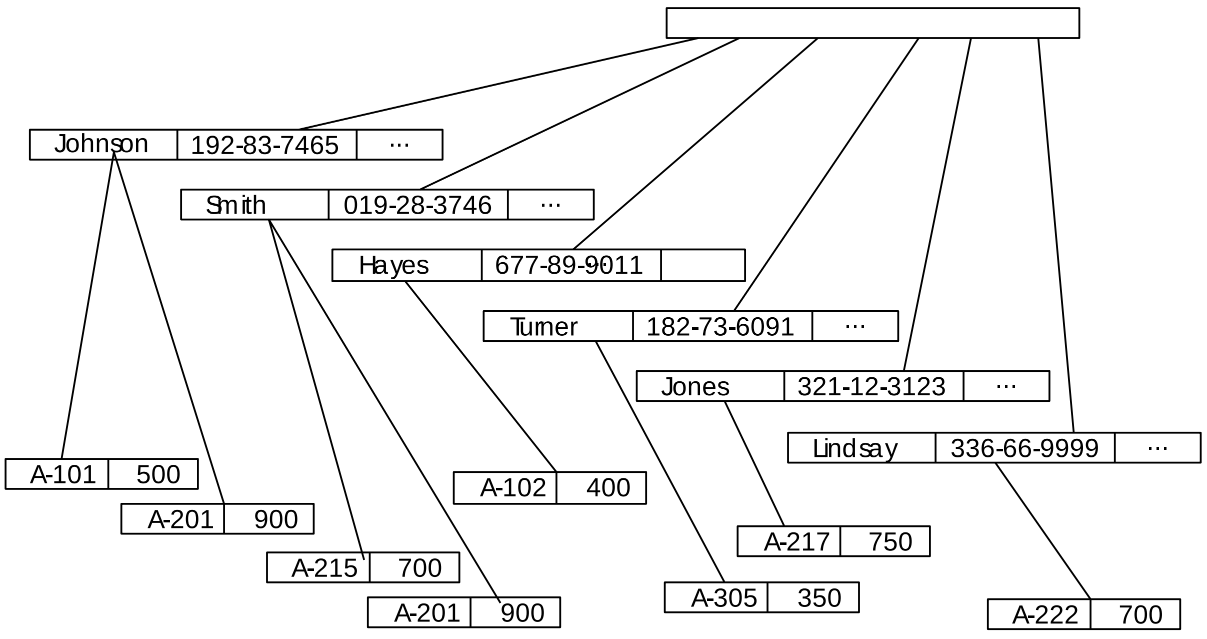


A sample Network database

The records in the database are organized as collections of arbitrary graphs.

1.3.2.3 Hierarchical Model

The hierarchical model is similar to the network model in the sense that data and relationships among data are represented by records and links, respectively.



A sample Hierarchical database

It differs from the network model in that the records are organized as collections of trees rather than arbitrary graphs.

1.3.2.4 Differences between the Models

The relational model differs from the network and hierarchical models in that it does not use pointers or links. Instead, the relational model relates records by the values they contain.

1.3.3 Physical Data Models

Physical data models are used to describe data at the lowest level i.e., in the Physical level.

Two of the widely known ones are:

i) Unifying model
ii) Frame memory

1.4 Database Languages

A database system provides two different types of languages: one to specify the database schema, and the other to express database queries and updates.

1.4.1 Data Definition Language

A database schema is specified by a set of definitions which are expressed by a special language called a *data definition language* (DDL).

The result of compilation of DDL statements is a set of tables which are stored in a special file called *data dictionary* (or *directory).*

A data directory is a file that contains *metadata;* that is, "data about data."

The storage structure and access methods used by the database system are specified by a set of definitions in a special type of DDL called a *data storage and definition* language.

1.4.2 Data Manipulation Language

A *data manipulation language* (DML) is a language that enables users to access or manipulate data.

Data manipulation in terms of

- The retrieval of information stored in the database
- The insertion of new information into the database
- The deletion of information from the database
- The modification of data stored in the database.

There are basically two types:

i) Procedural DMLs
ii) Nonprocedural DMLs

- **Procedural** DMLs require a user to specify *what* data is needed and *how* to get it.
- **Nonprocedural** DMLs require a user to specify *what* data is needed *without* specifying how to get it. Nonprocedural DMLs are easier to learn which is not efficient.

Query & Query Language

- A *query* is a statement requesting the retrieval of information.
- The portion of a DML that involves information retrieval is called a *query language.*

1.5 Database Manager

A *database manager* is a program module which provides the interface between the low-level data stored in the database and the application programs and queries.

The database manager is responsible for the following tasks:

Interaction with the file manager: The database manager translates the various DML statements into low-level file system commands. Thus, the database manager is responsible for the actual storing, retrieving, and updating of data in the database.
Integrity enforcement: The data values stored in the database must satisfy certain types of consistency constraints. The database manager determines whether updates to the database result in the violation of the constraint; if so, appropriate action must be taken.

For example, the number of hours an employee may work in one week may not exceed some specific limit (say, 80 hours). Such a constraint must be specified explicitly by the database administrator.

Security enforcement: It is the job of the database manager to enforce the security requirements. That is every database user need not have access to the entire content of the database.

Backup and recovery: A computer system, like any other mechanical or electrical device, is subject to failure. Causes of failure include disk crash, power failure, and software errors. In each of these cases, information concerning the database is lost. It is the responsibility of the database manager to detect such failures and restore the database to a state that existed prior to the occurrence of the failure. This is usually accomplished through the initiation of various backup and recovery procedures.

Concurrency control: When several users update the database concurrently, the consistency of data may no longer be preserved. Controlling the interaction among the concurrent users is another responsibility of the database manager.

1.6 Database Administrator

The person who is having central control (i.e., overall control) of both data and programs accessing that data is called the *database administrator* (DBA).

The various functions of the database administrator:

Schema definition: The original database schema is created by writing a set of definitions which are translated by the DDL compiler to a set of tables that are permanently stored in the *data dictionary.*

Storage structure and access method definition: Appropriate storage structures and access methods are created by writing a set of definitions which are translated by the data storage and definition language compiler.

Schema and physical organization modification: Modifications to either the database schema or the description of the physical storage organization, are accomplished by writing a set of definitions which are used by either the DDL compiler or the data storage and definition language compiler to generate modifications to the appropriate internal system tables (for example, the data dictionary).

Granting of authorization for data access. The database administrator regulates the access of the database, by granting different types of authorization to the various users.

Integrity constraint specification: Integrity constraints are kept in a special system structure that is consulted by the database manager whenever an update takes place in the system.

1.7 Database Users

A primary goal of a database system is to provide an environment for retrieving information from and storing new information into the database.

There are four different types of database system users;

Application programmers: Application programmers are computer professionals who interact with the system through DML calls, which are included in a program written in a *host* language. These programs are commonly referred to as *application programs.*

The DML syntax is usually quite different from the host language syntax. A special preprocessor, called the DML *pre compiler,* converts the DML statements to normal procedure calls in the host language. The resulting program is then run through the host language compiler, which generates appropriate object code.

There are special types of programming languages which combine control structures of Pascal-like languages with control structures for the manipulation of a database objects. These languages, sometimes called *fourth-generation languages,* often include special features to facilitate the generation of forms and the display of data on the screen. Most major commercial database systems include a fourth-generation language.

Sophisticated users: Sophisticated users interact with the system without writing programs. Instead, they form their requests in a database query language. Each such query is submitted to a *query processor* whose function is to take a DML statement and break it down into instructions that the database manager understands.

Specialized users: Some sophisticated users write specialized database applications that do not fit into the traditional data processing framework. Among these applications are computer-aided design systems, knowledge-base and expert systems, systems that store data with complex data types (for example, graphics data and audio data), and environment-modeling systems.

Naive users: Unsophisticated users interact with the system by invoking one of the permanent application programs that have been written previously.

For example, a bank teller who needs to transfer `50 from account *A* to account *B* would invoke a program called *transfer*. This program would ask the teller for the amount of money to be transferred, the account from which the money is being transferred, and the account to which the money is to be transferred.

1.8 Overall System Structure

A database system is partitioned into modules that deal with each of the responsibilities of the overall system.

The computer's operating system provides only the most basic services and the database system must build on that base. Thus, the design of a database system must include consideration of the interface between the database system and the operating system.

The functional components of a database system is divided into query processor components and storage manager components.

The query processor components include:

DML compiler: which translates DML statements in a query language into low-level instructions that the query evaluation engine understands. In addition, the DML compiler attempts to a user's request into an equivalent but more efficient form, thus finding a good strategy for executing the query.

Embedded DML precompiler: which converts DML statements embedded in an application program to normal procedure calls in the host language. The precompiler must interact with the query processor in order to generate the appropriate code.

DDL interpreter: which interprets DDL statements and records them in a set of tables containing *metadata*.

Query evaluation engine: which executes low-level instructions generated by the DML compiler.

The storage manager components provide the interface between the low-level data stored in the database and the application programs and queries submitted to the systems. the storage manager components include:

Authorization and integrity manager: which tests for the satisfaction of integrity constraints and checks the authority of users to access data.

Transaction manager: which ensures that the database remains in a consistent (correct) state despite system failures, and that concurrent transaction executions proceed without conflicting.

File manager: Manages the allocation of space on disk storage and the data structures used to represent information stored on disk.
Buffer manager: which is responsible for fetching data from disk storage into main memory, and deciding what data to cache in memory.

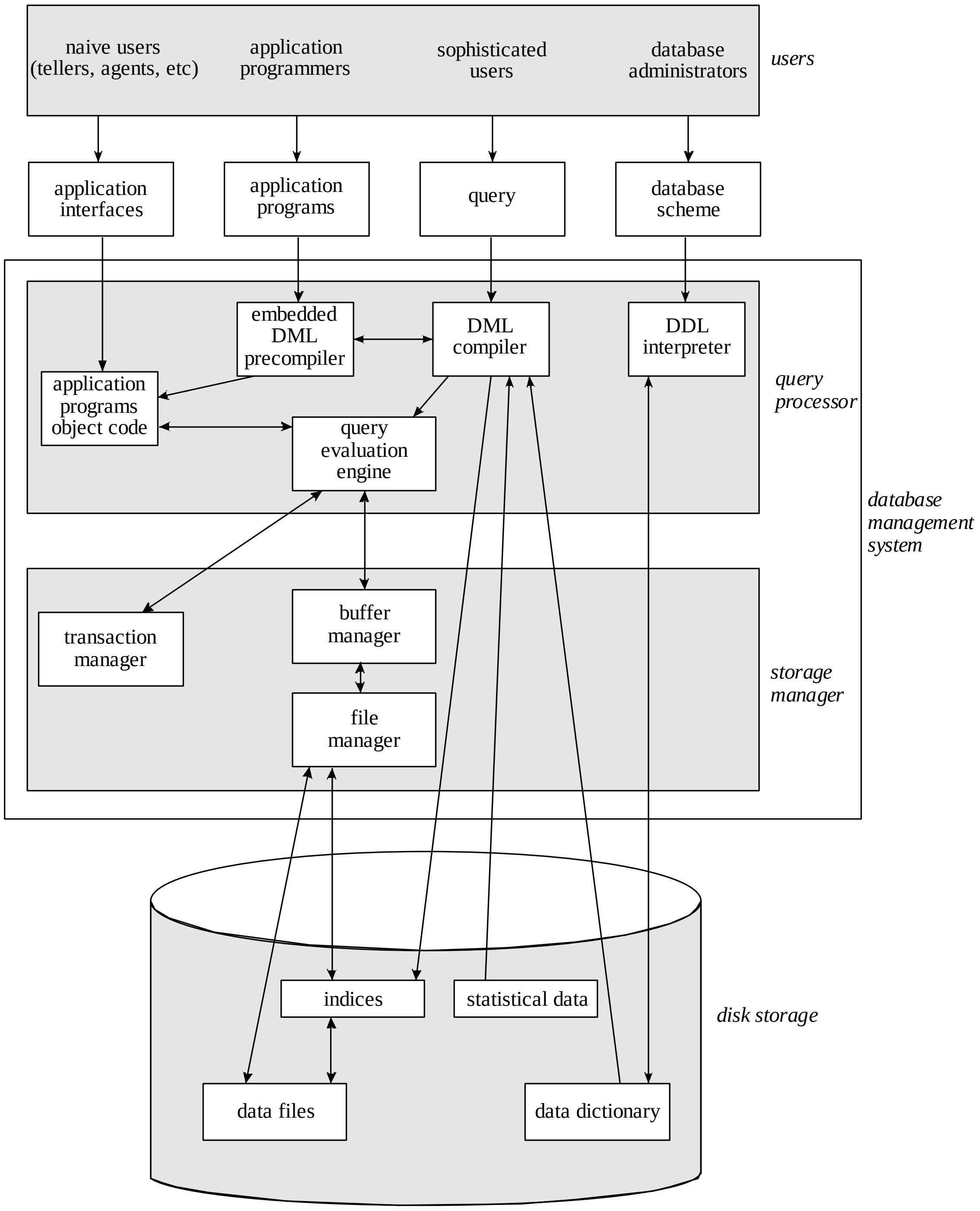
Several data structures are required as part of the physical system implementation:

Data files: which store the database itself.

Data dictionary: which, stores metadata about the structure of the database. The data dictionary is used heavily.

Indices: which provide for fast access to data items holding particular values.

Statistical data: which store statistical information about the data in the database. This information is used by the query processor to select efficient ways to execute a query



Overall System Structure

UNIT – II

Entity-Relationship Model

The entity-relationship (E-R) data model is a collection of basic objects called *entities,* and *relationships* among those objects.

Entities and Entity Sets

An ***entity*** is an object that exists and is distinguishable from other objects.

For example, John Harris with social security number 890-12-3456 is an entity, since it uniquely identifies one particular person in the universe. Similarly, account number 401 at the Redwood branch is an entity that uniquely identifies one particular account. An entity may be concrete, such as a person or a book, or it may be abstract, such as a holiday or a concept.

An ***entity set*** is a set of entities of the same type.

The set of all persons having an account at a bank, for example, can be defined as the entity set *customer.* Similarly, the entity set *account* might represent the set of all accounts in a particular bank.

An entity is represented by a set of *attributes.*

Possible attributes of the *customer* entity set are *customer-name, social-security, street,* and *customer-city.* Possible attributes of the *account* entity set are *account-number* and *balance.*

For each attribute there is a set of permitted values, called the ***domain*** of that attribute.

The domain of attribute *customer-name* might be the set of all text strings of a certain length. Similarly, the domain of attribute *account number* might be the set of all positive integers.

Every entity is described by a set of (attribute, data value) ***pairs***, one pair for each attribute of the entity set.

A particular *customer* entity is described by the set *{(name,* Harris), *(social-security,* 890-12-3456), *(street,* North), *(city,* Rye)}, which means the entity describes a person named Harris with social security number 890-12-3456, residing at North Street in Rye.

Attributes :

An attribute, as used in the E-R model, can be characterized by the following types:

- **Simple** and **Composite** attributes, the simple attributes cannot be divided into subparts, whereas in the case of composite of attributes, it can be divided into subparts.

  For example, *customer-name* could be structured as a composite attribute consisting of *first-name, middle-name* and *last-name.* The *account-number* attribute refers to only one account number.

- **Single-valued** and **multivalued** attributes. The attribute holds single value is called as single valued attribute. An attribute may have a set of values for a specific entity, such attribute is called as multivalued attribute.

For example, account-number holds only a single account number and an attribute dependent-name of an employee may have zero, one, or more dependents

- **Null attributes**. A null is used when an entity does not have a value for an attribute. A null value will have the meaning of "not applicable". Null can also designate that an attribute value is unknown which means either the value may be missing or not known.

- **Derived attribute**. The value for this type of attribute can be derived from the values of other related attributes or entities.

Relationships and Relationship Sets
A ***relationship*** is an association among several entities.

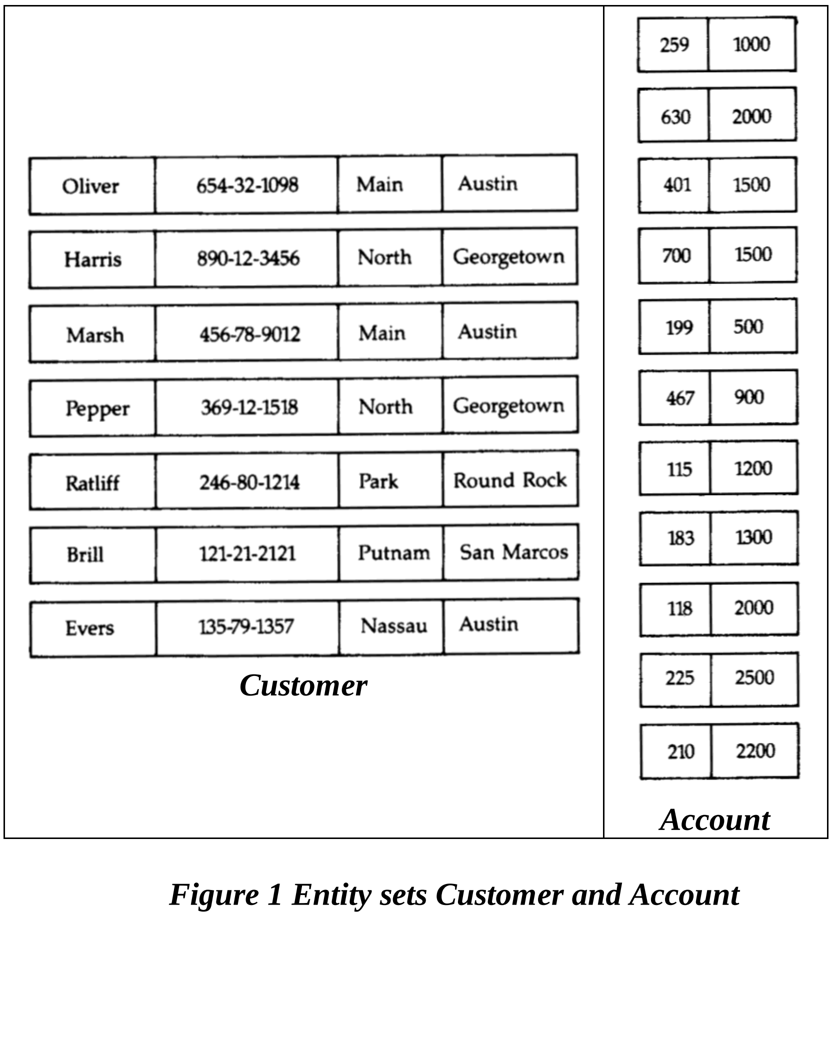
For example, we may define a relationship, which associates customer Harris with account 401. This specifies that Harris is a customer with bank account number 401.

A *relationship set* is a set of relationships of the same type.
Formally, it is a mathematical relation on $n \geq 2$ entity sets. If $E_1, E_2, \ldots, E_n$ are entity sets, then a relationship set R is a subset of

$$\{(e_1, e_2\ e_3 \ldots e_n)\ e_1 \in E_1, e_2 \in E_2, \ldots, e_n \in E_n\}$$

where $(e_1, e_2, \ldots, e_n)$ is a relationship.

Consider the two entity sets *customer* and *account* in Figure 1. We define the relationship set *CustAcct* to denote the association between customers and the bank accounts that they have. This association is depicted in Figure 2.



Oliver	654-32-1098	Main	Austin
Harris	890-12-3456	North	Georgetown
Marsh	456-78-9012	Main	Austin
Pepper	369-12-1518	North	Georgetown
Ratliff	246-80-1214	Park	Round Rock
Brill	121-21-2121	Putnam	San Marcos
Evers	135-79-1357	Nassau	Austin

Customer

259	1000
630	2000
401	1500
700	1500
199	500
467	900
115	1200
183	1300
118	2000
225	2500
210	2200

Account

Figure 1 Entity sets Customer and Account

If two entities are involved in a relationship then the relationship is known as ***binary relationship set***. Most of the relationship sets in a database system are binary.

If three entities are involved in a relationship then the relationship is known as ***ternary relationship set***.

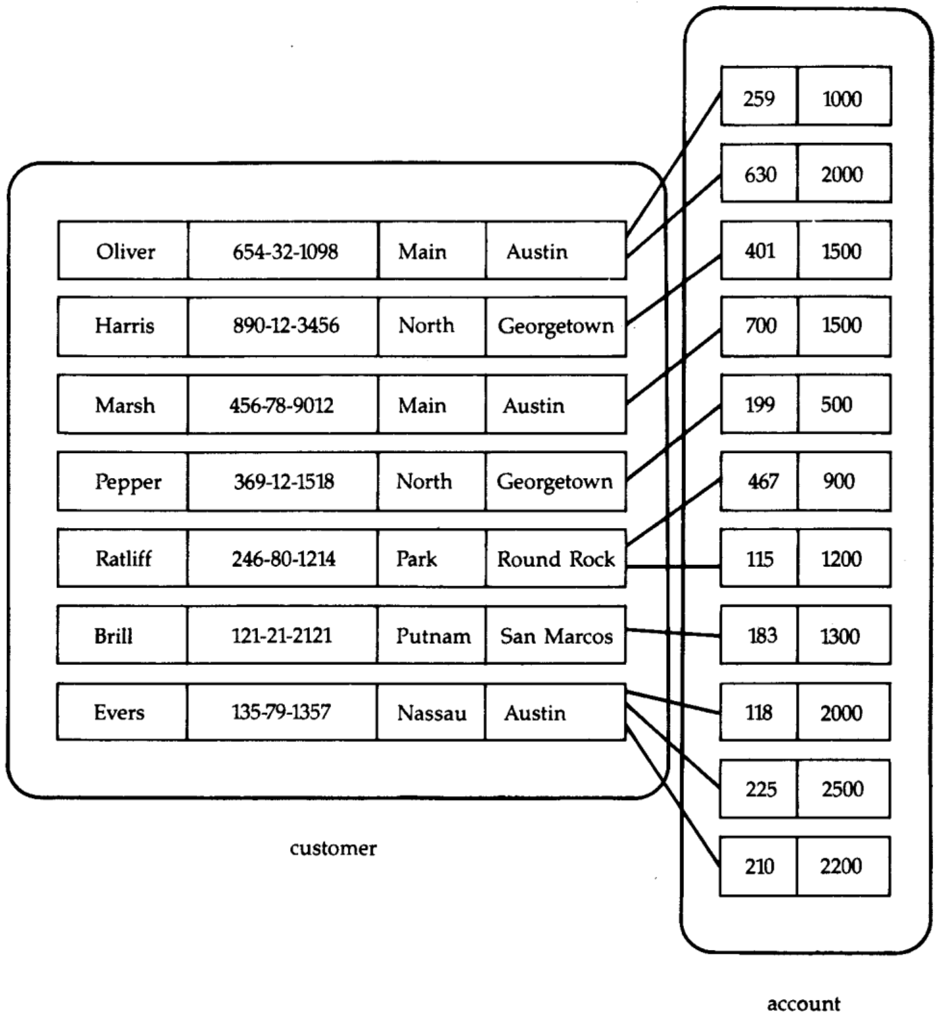


Figure 2 Relationship between Entity sets Customer and Account

The function that an entity plays in a relationship is called its ***role***.

A relationship may also have *descriptive attributes*.

For example, *date* could be an attribute of the *CustAcct* relationship set. This specifies the last date on which a customer has accessed the account. The *CustAcct* relationship among the entities corresponding to customer Harris and count 401 is described by *{(date,* 23 May 1990)}, which means that the last time Harris accessed account 401 was on 23 May 1990.

Mapping Constraints

An E-R enterprise schema may define certain constraints to which the contents of a database must conform. There are two constraints available mapping cardinalities and existence dependencies.

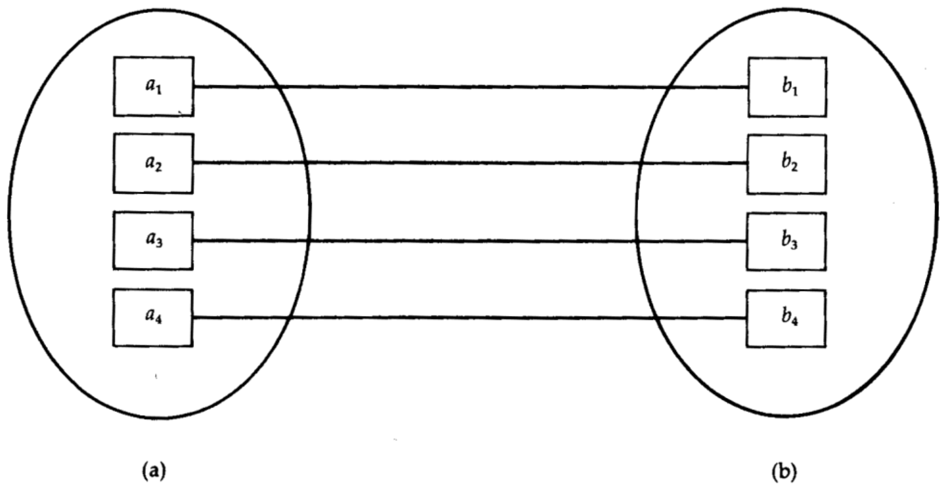
Mapping Cardinalities

Mapping cardinalities, or cardinality ratios, express the number of entities to which another entity can be associated via a relationship set.

Mapping cardinalities are most useful in describing binary relationship sets.

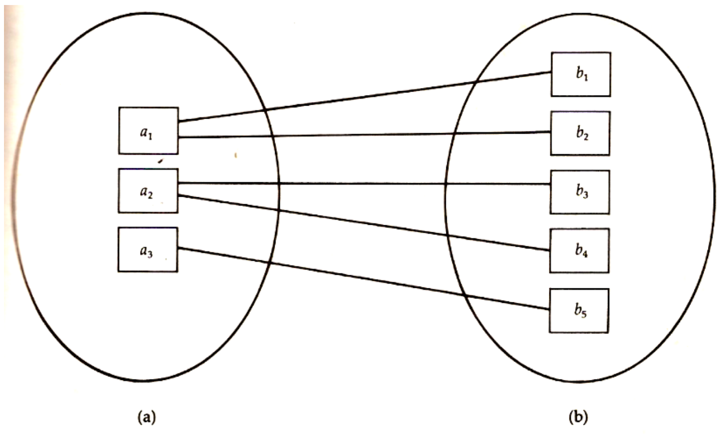
For a binary relationship set *R* between entity sets *A* and *B,* the mapping cardinality must be one of the following:

One-to-one: An entity in *A* is associated with at most one entity in *B,* and an entity in *B* is associated with at most one entity in *A.*



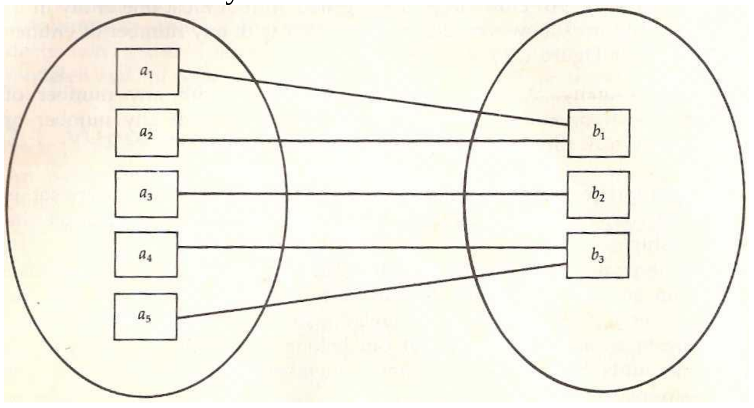
One-to-one relationship

One-to-many: An entity in *A* is associated with any number of entities in *B.* An entity in *B,* however, can be associated with at most one entity in *A.*



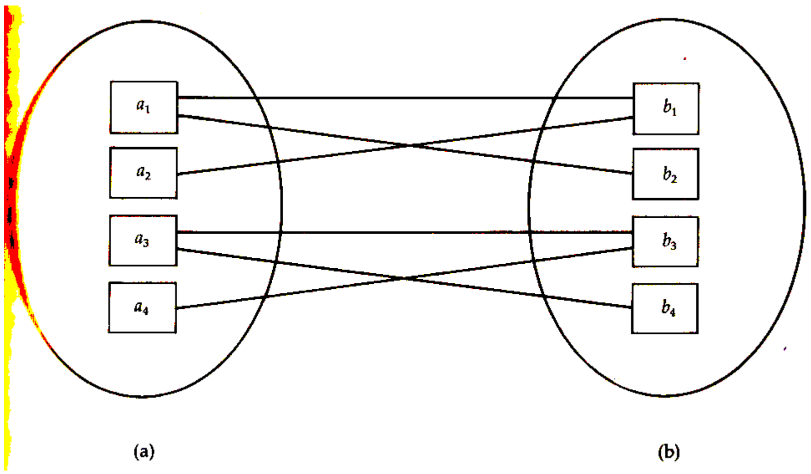
One-to-many relationship

Many-to-one. An entity in *A* is associated with at most one entity in *B.* An entity in *B,* however, can be associated with any number of entities in A.



Many-to-one relationship

Many-to-many: An entity in *A* is associated with any number of entities in *B,* and an entity in *B* is associated with any number of entities in *A.*



Many-to-many relationship

Keys:

A **superkey** is defined in the relational model of database organization as a set of attributes of a relation variable for which it holds that in all relations assigned to that variable, there are no two distinct tuples (rows) that have the same values for the attributes in this set. Equivalently a superkey can also be defined as a set of attributes of a relation schema upon which all attributes of the schema are functionally dependent.

In the relational model of databases, a **candidate key** of a relation is a minimal superkey for that relation; that is, a set of attributes such that

1. the relation does not have two distinct tuples (i.e. rows or records in common database language) with the same values for these attributes (which means that the set of attributes is a superkey)
2. there is no proper subset of these attributes for which (1) holds (which means that the set is minimal).

The constituent attributes are called **prime attributes**. Conversely, an attribute that does not occur in ANY candidate key is called a **non-prime attribute**.

Since a relation contains no duplicate tuples, the set of all its attributes is a superkey if NULL values are not used. It follows that every relation will have at least one candidate key.

The candidate keys of a relation tell us all the possible ways we can identify its tuples. As such they are an important concept for the design of database schema.

Definition: The primary key of a relational table uniquely identifies each record in the table. It can either be a normal attribute that is guaranteed to be unique (such as Social Security Number in a table with no more than one record per person) or it can be generated by the DBMS (such as a globally unique identifier, or GUID, in Microsoft SQL Server). Primary keys may consist of a single attribute or multiple attributes in combination.

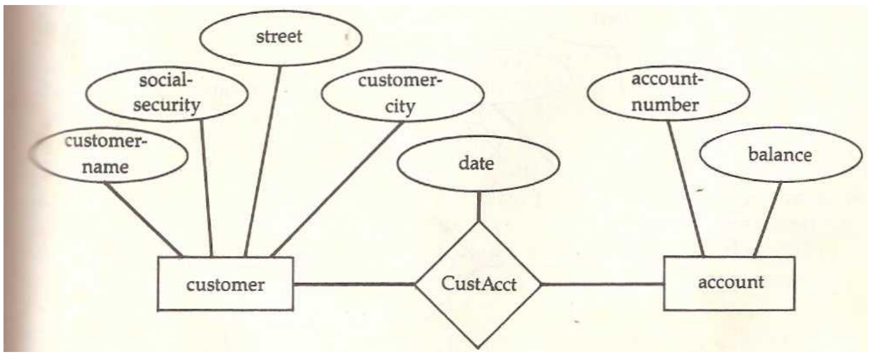
Examples:

Imagine we have a STUDENTS table that contains a record for each student at a university. The student's unique student ID number would be a good choice for a primary key in the STUDENTS table. The student's first and last name would not be a good choice, as there is always the chance that more than one student might have the same name.

Entity-Relationship Diagram

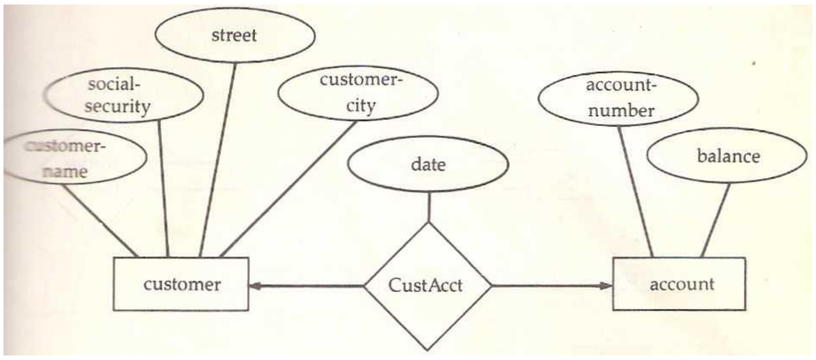
The overall logical structure of a database can be expressed graphically by an *E-R diagram*. The diagram consists the following components:

- **Rectangles,** which represent entity sets.
- **Ellipses,** which represent attributes.
- **Diamonds,** which represent relationship sets
- **Lines,** which link attributes to entity sets and entity sets to relationship sets.

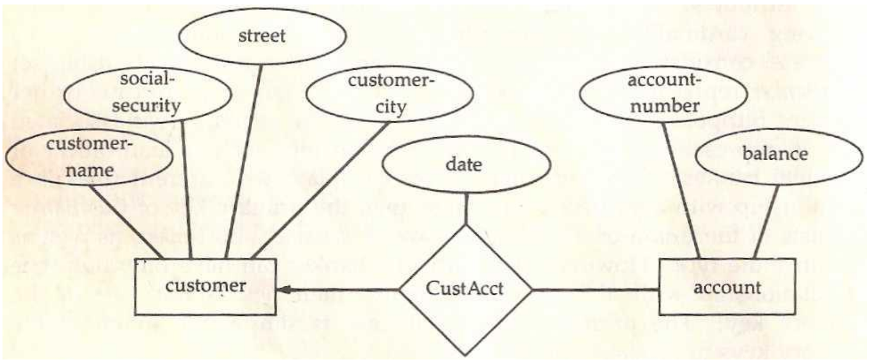


3. E-R Diagram

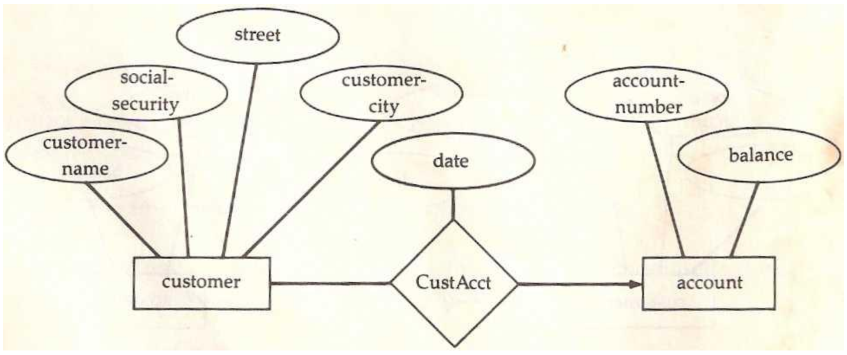
Consider the entity-relationship diagram in Figure 3, which consists two entity sets, *customer* and *account,* related through a binary relationship set *CustAcct.* The attributes associated with *customer* are *customer-name, social-security, street,* and *customer-city.* The attributes related with *account* are *account-number* and *balance.*



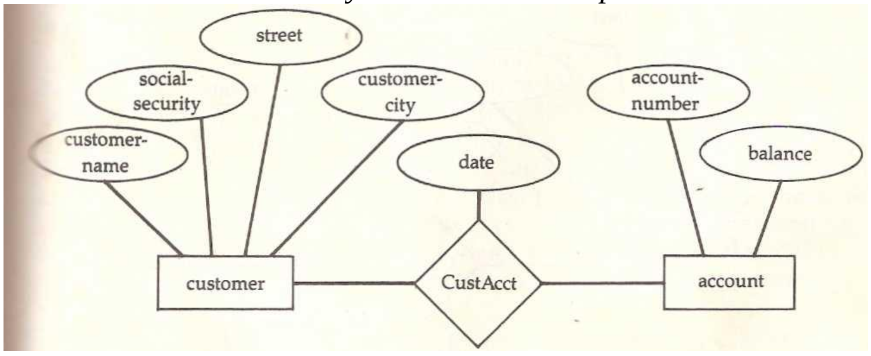
4. One-to-one relationship



5. One-to-many relationship



6. Many-to-one relationship



7. Many-to-many relationship

To distinguish among these, we shall draw a directed line (→) or an undirected line (—) between the relationship set.

A directed line from the relationship set *CustAcct* to the entity set *account* specifies that the *account* entity set participates in either a one-to-one or a many-to-one relationship with the *customer* entity set.

An undirected line from the relationship set *CustAcct* to the entity set *account* specifies that *account* entity set participates in either a many-to-many or one-to-many relationship with the *customer* entity set.

If the relationship set *CustAcct* were one-to-one, then both lines from *CustAcct* would have arrows, one pointing to the *account* entity set and one pointing to the *customer* entity set (Figure 4).

If the relationship set *CustAcct* were one-to-many from (*customer* to *account*) then the line from *CustAcct* to *customer* would be directed, with an arrow pointing to the *customer* entity set (Figure 5).

If the relationship set *CustAcct* were many-to-one from *customer* to *account,* then the line from *CustAcct* to *account* would have an arrow pointing to the *account* entity set (Figure 6).

Finally, the E-R diagram of Figure 7, we see that the relationship *CustAcct* is many-to-many.

<u>Reducing E-R diagram into tables</u>

<u>1.Tabular representation for strong entity sets:</u>

Let E be a strong entity set with descriptive attributes a1,a2,a3,……an. The entity is represented by a schema called E with ‘n’ distinct attributes. Each tuple in a relation on this table *loan* corresponds to one entity of the entity set E.

Consider the entity set *loan* of the E-R diagram. This entity set has two attributes, *loan_number and amount.*We represent entity set by a table called loan.

loan =(loan_number,amount). The row in the loan table, means that loan_number L-17 has a loan amount of Rs.2000. (L-17,2000)

Loan_number	Amount
L-11	1000
L-17	2000
L-14	4000
L-15	6000

The loan Table

NOTE: Since *loan_number* is a primary key of the entity set. It is also the primary key of relation table.

2.Tabular representation for weak entity sets:

Let A be a weak entity set with attributes a1,a2,a3,……an. Let B be a strong entity set on which A depends. Let the primary key of B consists of attributes b1,b2,b3,……bn.We can represent the entity set A by a relation schema called A with one attribute for each member of the set :

{a1,a2,a3,……an}U{b1,b2,b3,……bn}

Consider the entity set *payment* , it consists of three attributes. (*Payment_number* , *Payment_data and Payment_amount*.)

The primary key of the *loan* entity set on which payment depends is *loan-number*. Thus we represent payment by a table with four columns. i.e (*loan-number, payment-number*, *payment-date, payment-amount*).

3.Tabular representation for Realationship sets:

Let R be a relationship set, let a1,a2,a3,……am be the set of attributes formed by the union of the primary keys of each entity sets participating in R ,and let the descriptive attributes (if any) of R be b1,b2,b3,……bn. We represent the relationship set by a table called 'R' with one column for each attributes of the entity set.

{a1, a2, a3,……am}U{b1,b2,b3,……bn}

Example: Consider the relationship set *borrower* in the E.R diagram. This relationship involving the following two entity sets.

- *customer,* with the primary key *customer_id.*
- *loan,* with the primary key *loan_number.*

Since the relationship has no attributes, the *borrower* table has two columns *customer_id* and *loan_number.*

Borrower=(customer_id,loan_number)

customer_id	loan_number

1001	L-11
1002	L-13
1003	L-17
1004	L-14
1005	L-18

The Borrower Table

The primary key of the *borrower* relation is, the union of the primary keys attributes of customer loan.

GENERALIZATION:

Generalization is used to express commonality between the two entity sets which is a containment relationship that exists between the higher level entity sets and one or more lower level entity sets.

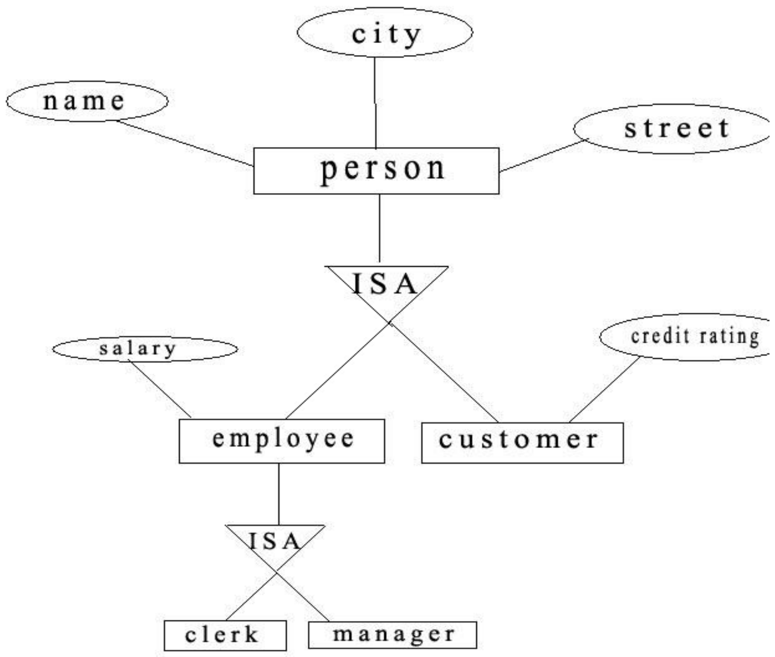
Generalization is depicted through a Triangle component labeled ISA. The Label ISA Stands for "is a" and also represents "is an".

There may be similarities between the *customer* entity set and the *employee* entity set by having several attributes in common.

EX: *person* is the higher level entity set and *customer* and *employee* are lower level entity sets

Higher and lower level entity sets may be designated by super class and subclass

The *person* entity is super class of the *customer* and *employee* subclasses



Total Generalization

Each higher level entity much belongs to lower level entity set.

Partial Generalization

Some higher level entity may not belong to any lower level entity set

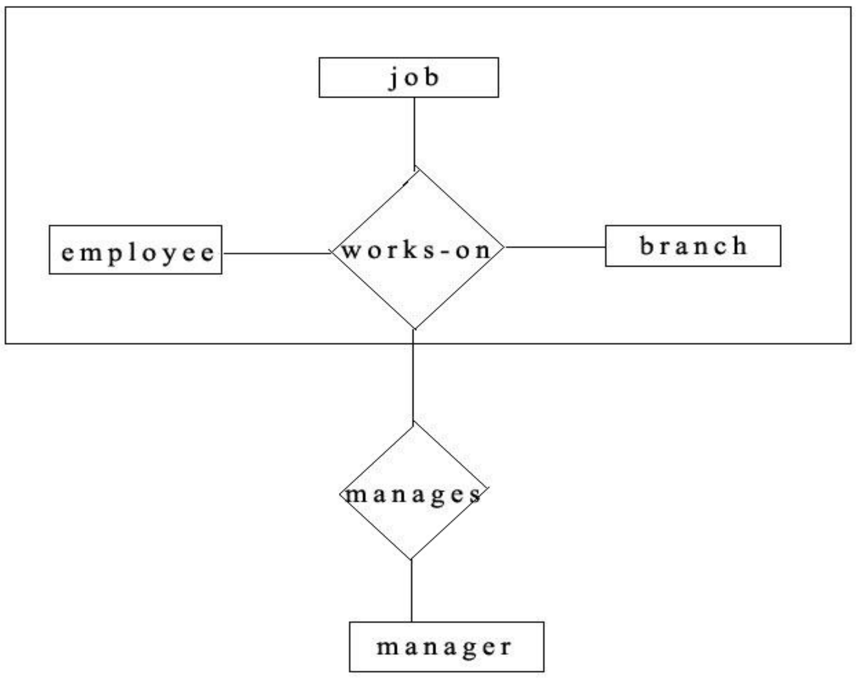
AGGREGATION:

It is not possible to express relationship among relationship in ER model. The solution is to use Aggregation. Aggregation is an abstraction through which relationship are treated a higher level entities. Transforming an ER Diagram which includes aggregation to a tabular form is straightforward.

EX: The relationship set *works-on* (relating the entity sets *employee, branch* and *job*) is consider as a higher level entity set

Then we can consider the higher level entity set as same as other entity sets

Now we can create the binary relationship (manages) between works-on and manager to represent who manages the task.



We cannot combine both relationships (works-on, manages) into single relationship becomes some (employee, branch, job) may not have a manager.

UNIT – III

Introduction:

The inputs and outputs of a query are relations. A query used to evaluate using instances of each input relation and it produces an instance of the output relation.

- Query language can be categorized as procedural or nonprocedural.
- In nonprocedural language, the user describes the information desired without giving a specific procedure for obtaining that information (ex: relational calculus)
- In procedural language, the user instructs the system to perform a sequence of operations on the database to compute the desired result (ex: relational algebra).

Relational-Algebra Operations:

The relational algebra is a procedural query language. It consists of a set of operations that take one or two relations as input and produce a new relation as their result.

The Select operation:

The Select operation selects tuples that satisfy a given predicate.

We use the lowercase Greek letter sigma (σ) to denote selection.

The predicate appears as a subscript to. The argument relation is in parenthesis after the

Thus, to select those tuples of the loan relation where the branch is "perryridge", we write,

$$\boxed{\sigma_{branch\_name} = \text{“perryridge”} \text{ (loan)}}$$

Loan relation

Loan_number	Branch_name	Amount
L-11	**Round hill**	**900**
L-14	**Downtown**	**1500**
L-15	**Perryridge**	**1500**
L-16	**Perryridge**	**1300**
L-17	**Downtown**	**1000**
L-23	**Redwood**	**2000**

The result of preceding query is,

Loan_number	Branch_name	Amount
L-15	**Perryridge**	**1500**
L-16	**Perryridge**	**1300**

We can find all tuples in which the amount is more than 1200,

$$\sigma_{\text{amount}} > 1200 \text{ (loan)}$$

Loan_number	Branch_name	Amount
L-14	**Downtown**	**1500**
L-15	**Perryridge**	**1500**
L-16	**Perryridge**	**1300**
L-23	**Redwood**	**2000**

The Project Operation:

The project operation allows us to produce this relation. The project operation is a unary operation that returns its argument relation, with certain attributes left out, because a relation is a set, any duplicate rows are eliminated.

Projection is denoted by the uppercase pi(Π). We list those attributes that we wish to appear in the result as a subscript to Π.

$$\Pi_{\text{loan\_number, amount}} \text{ (loan)}$$

The result of this query is,

Loan_number	Amount
L-11	**900**
L-14	**1500**
L-15	**1500**
L-16	**1300**
L-17	**1000**
L-23	**2000**

The Union Operation:

We need a query to find the names of all bank customers who have either an account or a loan or both.

The customer relation does not contain the information. Since a customer does not need to have either an account or a loan at the bank.

So we need the information in the depositor relation and in the borrower relation.

First, we have to write a query to find the names of all customers with a loan in the bank.

$$\Pi_{\text{customer\_name}} \text{ (borrower)}$$

Next, write a query to find the names of all customers with an account in the bank,

$$\Pi_{\text{customer\_name}} \text{ (depositor)}$$

Then, we need the union of these two sets, that is, we need all customer name that appear in either or both of the two relations.

We find these data by the binary operation Union denoted by "U".

$$\boxed{\Pi_{\text{customer\_name}}\text{(borrower)} \cup \Pi_{\text{customer\_name}}\text{(depositor)}}$$

The depositor relation

Customer_name	Account_number
Smith	**A-102**
John	**A-101**
Johnson	**A-201**
Jones	**A-217**
Lindsay	**A-222**
Brooks	**A-305**

Borrower relation

Customer_name	Loan_number
Jones	**L-16**
Smith	**L-93**
Williams	**L-15**
Johnson	**L-14**
Smith	**L-17**
John	**L-11**

The result relation for Union query is,

Customer_name
Smith
John
Johnson
Jones
Lindsay
Brooks
Williams

The Set-Difference Operation:

The Set-difference operation, denoted by _ , allows us to find tuples that are in one relation but are not in another.

The expression r-s produces a relation containing those tuples in r but not in s.

We can find all customers of the bank who have an account but not a loan by writing.

$$\Pi_{\text{customer\_name}}\text{(depositor)} - \Pi_{\text{customer\_name}}\text{(borrower)}$$

The result relation for this query is,

depositor relation Borrower relation

Customer_name	Account_number

Smith	**A-102**
John	**A-101**
Johnson	**A-201**
Jones	**A-217**
Lindsay	**A-222**
Brooks	**A-305**

Customer_name	Loan_number
Jones	**L-16**
Smith	**L-93**
Williams	**L-15**
Johnson	**L-14**
Smith	**L-17**
John	**L-11**

Customer_name
Johnson
Lindsay
Brooks

The Cartesian-Product Operation:

The Cartesian-Product operation, denoted by a cross(X), allows us to combine information from any two relations.

We write the Cartesian product of relations r_1 and r_2 as r_1 X r_2.

The same attribute name may appear in both r_1 and r_2, we need to devise a naming schema to distinguish between these attributes.

For example, the relation schema for r = borrower X loan is (borrower.customer_name, borrower.loan_number, loan.loan_number, loan.branch_name, loan.amount).

Borrower relation Loan relation

Customer_name	Loan_number
Adams	**L-16**
Jones	**L-17**
Smith	**L-23**

Loan_number	Branch_name	Amount
L-23	**Round hill**	**900**
L-17	**Downtown**	**1000**
L-16	**Perryridge**	**1300**

The result of borrower X loan is

Customer_name	borrower.loan_number	loan.loan_number	Branch_name	Amount
Adams	**L-16**	**L-23**	**Round hill**	**900**
Adams	**L-16**	**L-17**	**Downtown**	**1000**
Adams	**L-16**	**L-16**	**Perryridge**	**1300**
Jones	**L-17**	**L-23**	**Round hill**	**900**
Jones	**L-17**	**L-17**	**Downtown**	**1000**
Jones	**L-17**	**L-16**	**Perryridge**	**1300**
Smith	**L-23**	**L-23**	**Round hill**	**900**
Smith	**L-23**	**L-17**	**Downtown**	**1000**
Smith	**L-23**	**L-16**	**Perryridge**	**1300**

If we want to find the names of all customer who have a loan at the "Perryridge" branch. If we write,

σ branch_name = "Perryridge" (borrower X loan)

Customer_name	borrower.loan_number	loan.loan_number	Branch_name	Amount
Adams	**L-16**	**L-16**	**Perryridge**	**1300**
Jones	**L-17**	**L-16**	**Perryridge**	**1300**
Smith	**L-23**	**L-16**	**Perryridge**	**1300**

The Rename Operation:

The results of relational algebra expressions do not have a name that we can use to refers to them. It is useful to be able to give them names.

The rename operator, denoted by the lowercase Greek letter rho (ρ).

Given a relational algebra E, the expression, $\rho_x(E)$ returns the result of expression E under the name x.

A relation r by itself is a relational-algebra expression, thus we can also apply the rename operation to a relation r to get the same relation under a new name.

A second form of the rename operation is as follows, Assume that a relational algebra expression E has arity A. Then, the expression

$$\rho_{x(A_1, A_2, \ldots\ldots A_n)}(E)$$

returns the result of expression E under the name x, and with the attributes renamed to $A_1, A_2, \ldots.., A_n$.

The temporary relation that consists of the balances that are not the largest can be written as,

$$\Pi_{account.balance}(\sigma_{account.balance\ <\ d.balance}(account.\ \rho_d(account))).$$

This expression gives those balances in the account relation for which a larger balance appears somewhere in the account relation (renamed as d). The result contains all balances except the largest one.

Formal definition of the relational algebra

A basic expression in the relational algebra consists of either one of the following.

- ⇨ A relation in the database.
- ⇨ A constant relation

A constant relation is written by listing its tuples within {}.

For example, {(A-101, Downtown,500)(A-215,Mianus,700)}.

Relational-algebra expressions:

- ❈ $E_1 \cup E_2$
- ❈ $E_1 - E_2$
- ❈ $E_1 \times E_2$

- ❋ $\sigma_p(E_1)$, where p is a predicate on attribute in E_1.
- ❋ $\Pi_s(E_1)$, where s is a list consisting of some of the attributes in E_1.
- ❋ $P_x(E_1)$, where x is the new name for the result of E_1.

Additional Relational-Algebra Operations:

- ⇨ Set-Intersection operation
- ⇨ Natural-join operation
- ⇨ Division operation
- ⇨ Assignment operation

Set-Intersection Operation:

Set intersection operation is defined by the symbol (∩). If we want to find all customers who have both a loan and an account. Using set intersection, we can write

$$\Pi_{customer\_name}(borrower) \cap \Pi_{customer\_name}(depositor).$$

The result relation for this query is,

Customer_name	Account_number
Hayes	**A-102**
Johnson	**A-101**
Johnson	**A-201**
Jones	**A-217**
Lindsay	**A-222**
Smith	**A-305**

depositor relation

Customer_name	Loan_number
Adams	**L-16**
Curry	**L-93**
Hayes	**L-15**
Jackson	**L-14**
Jones	**L-17**
Smith	**L-11**

borrower relation

Customer_name
Hayes
Jones
Smith

Customers with both an account and a loan at the bank.

Natural-Join Operation:

The natural join is a binary operation that allows us to combine certain selections and a Cartesian product into one operation.

It is denoted by the join symbol $\bowtie$.

The natural join operation forms a Cartesian product of its two arguments, performs a selection forcing equality on those attributes that appear in both relation schemas and finally removes duplicate attributes. Find the names of all customers who have a loan at the bank and find the amount of the loan. We can express this query by using the natural join,

$$\Pi_{customer\_name,\ loan\_number,\ amount}\ (borrower \bowtie loan).$$

Result of this query,

Loan_number	Branch_name	Amount
L-11	**Round hill**	**900**
L-14	**Downtown**	**1500**
L-15	**Perryridge**	**1500**
L-16	**Perryridge**	**1300**
L-17	**Downtown**	**1000**
L-23	**Redwood**	**2000**
L-93	**Mianus**	**500**

Customer_name	Loan_number
Adams	**L-16**
Curry	**L-93**
Hayes	**L-15**
Jackson	**L-14**
Jones	**L-17**
Smith	**L-11**
Smith	**L-23**
Williams	**L-17**

Loan relation borrower relation

Customer_name	Loan_number	Amount
Adams	L-16	**1300**
Curry	L-93	**500**
Hayes	L-15	**1500**
Jackson	L-14	**1500**
Jones	L-17	**1000**
Smith	L-23	**2000**
Smith	L-11	**900**
Williams	L-17	**1000**

Result of $\Pi_{customer\_name,\ loan\_number,\ amount}\ (borrower \bowtie loan).$

Division Operation:

The division operation is denoted by ÷, is suited to queries that include the phrase "for all".

We want to find all customers who have an account at all the branches located in Brooklyn. We can obtain all branches in Brooklyn by the expression.

$$r_1 = \Pi_{branch\_name}\ (\sigma_{branch\_city\ =\ \text{"Brooklyn"}}(branch))$$

Branch_name	Branch_city	Assets

Brighton	Brooklyn	**7100000**
Downtown	Brooklyn	**9000000**
Minaus	Horseneck	**400000**
Northtown	Rye	**3700000**
Perryridge	Horseneck	**170000**
Redwood	Palo alto	**2100000**

Branch relation

Result of $\Pi_{branch\_name}(\sigma_{branch\_city} =$ "Brooklyn" (branch))

Branch_name
Brigton
Downtown

We can find all (customer_name, branch_name) pairs for which the customer has an account at a branch by writing,

$r_2 = \Pi_{customer\_name,branch\_name}$ (depositor $\bowtie$ account).

Account_number	Branch_name	Amount
A-101	Downtown	**500**
A-102	Perryridge	**400**
A-201	Brighton	**900**
A-215	Mianus	**700**
A-217	Brighton	**750**
A-222	Redwood	**700**
A-305	Roundhill	**350**

Account relation

Customer_name	Account_number
Hayes	**A-102**
Johnson	**A-101**
Johnson	**A-201**
Jones	**A-217**
Lindsay	**A-222**
Smith	**A-215**
Turner	**A-305**

depositor relation

Customer_name	Branch_name
Hayes	Perryridge
Johnson	Downtown
Johnson	Brighton
Jones	Brighton
Lindsay	Redwood
Smith	Mianus
Turner	Roundhill

Result of $\Pi_{customer\_name,branch\_name}$ (depositor $\bowtie$ account)

We need to find customer who appear in r_2 with every branch_name in r_1.

The operation that provides exactly those customers is the divide operation.

$\Pi_{customer\_name,branch\_name}$ (depositor $\bowtie$ account) $\div$ Π_{branch_name} (σ_{branch_city} ="Brooklyn"(branch))

The result of this expression is a relation that has the schema (customer_name) and that contains the tuple (johnson).

Assignment Operation:

It is convenient to write a relational-algebra expression by assigning parts of it to temporary relation variables.

The assignment operation denoted by $\longleftarrow$.

We could write r ÷ s as,

temp1 $\longleftarrow$ $\Pi_{R-S}(r)$

temp2 $\longleftarrow$ $\Pi_{R-S}((temp1 \times S) - \Pi_{R-S}, S(r))$

result = temp1 – temp2.

The evaluation of an assignment does not result in any relation being displayed to user, rather than result of expression to the right of the $\longleftarrow$ is assigned to the relation variable on the left of the $\longleftarrow$.

Extended Relational-Algebra Operations:

The basic relational-algebra operations have been extended in several ways,

- Generalized Projection
- Aggregate Functions
- Outer Join

Generalized Projection:

The generalized projection operation extends the projection operation by allowing arithmetic functions to be used in the projection list.

The generalized projection operation has the form,

$$\Pi_{F1,F2,\ldots,Fn}(E)$$

Where, E is any relational-algebra expression and each F1,F2,…Fn is an arithmetic expression.

A relation credit_info, which lists the credit limit and credit_balance on the account.

If we want to find how much more each person can spend, we can write the following expression,

$$\Pi_{customer\_name,limit\_credit\_balance}(credit\_info).$$

Credit_info relation

Customer_name	Limit	Credit_balance
Curry	2000	**1750**
Hayes	1500	**1500**
Jones	6000	**700**
Smith	2000	**400**

The attribute resulting from the expression limit_credit_balance does not have a name.

We can apply the rename operation to the result of generalized projection in order to give it a name.

As a notational convenience, remaining of attributes can be combined with generalized projection,

$\Pi_{customer\_name,}$(limit_credit_balance) as credit_available (credit_info).

The result of $\Pi_{customer\_name,}$(limit_credit_balance) as credit_available (credit_info)

Customer_name	Credit_available
Curry	250
Jones	5300
Smith	1600
Hayes	0

Aggregate Functions:

Aggregate functions take a collection of values and return a single value as a result. For example, The aggregate function **sum** takes a collection of values and returns the sum of the values. Thus, the function sum applied on the collection, {1, 1, 3, 4, 4, 11} returns the value 24.

The aggregate function **avg** returns the average of the values, it returns the value 4.

The aggregate function **count** returns the number of the elements in the collection and returns 6.

Other common aggregate functions include **min** and **max**, which return the minimum and maximum values in a collection, they return 1 an 11 respectively.

The collections on which aggregate functions operate can have multiple occurrences of a value; the order in which the values appear in not relevant. Such collections Are called **multisets.**

Sets are a special case of multisets where there is only one copy of each element.

Pt_works relation

Employee_name	Branch_name	Salary
Adams	**Perryridge**	**1500**
Brown	**Perryridge**	**1300**
Gopal	**Perryridge**	**5300**
Johnson	**Downtown**	**1500**
Loreena	**Downtown**	**1300**
Peterson	**Downtown**	**2500**
Rao	**Austin**	**1500**

If we want to find out the total sum of salaries of all the part-time employees in the bank. The relational-algebra expression for this query,

$$_{sum(salary)}\ \textbf{(pt\_works)}$$

The symbol is the letter G in calligraphic font.

The result of the expression above is a relation with a single attribute, containing a single row with a numerical value corresponding to the sum of all the salaries of all employees.

count-distinct(branch_name) (pt_works)

We want to find tee total salary sum of all part_time employees at each branch of the bank separately rather than the sum for the entire bank.

We need to partition the relation pt_works in to groups based on the branch and to apply the aggregate function on each group.

The pt_works relation after grouping

Employee_name	Brance_name	Salary
Rao	Austin	1500
Sat		
Johnson	Downtown	1500
Loreena	Downtown	1300
Peterson	Downtown	2500
Adams	Perryridge	1500
Brown	Perryridge	1300
Gopal	Perryridge	5300

branch_name sum(salary)(pt_works)

In this expression, the attribute branc_name indicates the input relation pt_works must be divided in to groups based on the value of branch name.

The expression sum (salary) indicates that for each group of tuples, the aggregation function sum must be applied on the collection of values of the salary attribute.

Result of branch name **sum(salary) (pt_works)**

Branch_name	sum(Salary)
Austin	1500
Downtown	5300
Perryridge	8100

The general form of aggregation operation g is as follows,

G1,G2,……,Gn F1(A1), F2(A2), ….,Fm(Am)(E)

Where E is any relational_algebra expression;

G1,G2…,Gn constitute a list of attributes on which to group

Fi is an aggregate function.

branch_name sum(salary) as sum_salary, max(salary) as max_salary(pt_works)

Result of this query is,

Branch_name	Sum_salary	Max_salary
Austin	1500	1500
Downtown	5308	2500`

Perryridge	8100	5300

Outer Join:

The outer join operation is an extension of the join operation to deal with missing information.

Emp_name	Street	City
Kumar	Sakthi	Hollywood
Raja	Tunnel	Trichy
Smith	Kk	Salam
Williams	Nehru	Seattle

Employee Relation

ft_works relation

A possible approach would be to use the natural join operation as follows

Emp_name	Branch_name	Salary
Kumar	Mesa	1500
Raja	Mesa	1300
Smith	Redmond	5300
Williams	Redmond	1500

employee ⋈ **ft_works**

The result is

Emp_name	Street	City	Branch_name	Salary
Kumar	Sakthi	Hollywood	Mesa	1500
Raja	Tunnel	Trichy	Mesa	1300
Williams	Nehru	Seattle	Redmond	1500

In the result, we have lost the street and city information about smith, because the tuple describing smith is absent from ft_works relation.

Similarly, we have lost the branch name and salary information about Gates.

We can use the outer_join operation to avoid this loss of information.

There are actually three forms of the operation

- Left outer join denoted ⟕
- Right outer join denoted ⟖
- Full outer join denoted ⟗

Left Outer join:

The **left outer join** ⟕ takes all tuples in the left relation that did not match with any tuple in the right relation, pads the tuples with null values for all other attributes from the right relation and adds them ti the result of natural join.

Result of **employee** ⟕ **ft_works**

Emp_name	Street	City	Branch_name	Salary
Kumar	Sakthi	Hollywood	Mesa	1500
Raja	Tunnel	Trichy	Mesa	1300

Williams	Nehru	Seattle	Redmond	1500
Smith	Kk	Salam	Null	null

Right Outer join:

The **right outer join** ⟖ is symmetric with the left outer join. It pads tuples from the right relation that did not match any from the left relation with nulls and adds them to the result of the natural join.

The result of **employee ⟖ ft_works**

Emp_name	**Street**	**City**	**Branch_name**	**Salary**
Kumar	Sakthi	Hollywood	Mesa	1500
Raja	Tunnel	Trichy	Mesa	1300
Williams	Nehru	Seattle	Redmond	1500
Gate	Null	Null	Redmond	5300

Full Outer join:

The **full outer join** ⟗ does both of these operations, padding tuples from the left relation that did not match any from the right relation, as well as tuples from the right relation that did not match any from the left relation and adding them to the result of the join.

Result of **employee ⟗ ft_works**

Emp_name	**Street**	**City**	**Branch_name**	**Salary**
Kumar	Sakthi	Hollywood	Mesa	1500
Raja	Tunnel	Trichy	Mesa	1300
Williams	Nehru	Seattle	Redmond	1500
Smith	Kk	Salam	Null	Null
Gate	Null	Null	Redmond	5300

Relation Language:

Two formal language such as

- Tuple Relational Calculus
- Domain Relational Calculus

Which are declarative query languages based on mathematical logic.

Tuple Relational Calculus:

The Tuple Relational Calculus is a non-procedural query language. It describe the desired information without giving a specific procedure for obtaining that information.

A Query in the tuple relational calculus is expressed as

$$\{t \mid p(T)\}$$

That is, it is the set of all tuples t such that predicate P is true for t.

Formal Definition:

A Tuple relational calculus expression is of the form,

$\{t \mid p(T)\}$

Where P is a formula

Several tuple variables may appear in a formula.

A tuple Variable is said to be a free variable, it is quantified by a $\exists$

$t \in loan \wedge \exists s \in customer(t[branch\_name]=s[branch\_name])$

t is a free variable, tuple variable s is said to be a bound variable.

A tuple relational calculus formula is built up out of atoms.

An atom has one of the following forms

- **$s \in r$**, where s is a tuple variable and r is a relation.
- **$s[x]\ \Theta\ u[y]$**, where s and n are tuple variable, x is an attribute on which s is defined, y is an attribute on which u is defined, and Θ is a comparison operator ($<, \leq, =, \neq, >, \geq$).
- **$s[x]\ \Theta\ c$**, where s is a tuple variable, x is an attribute on which s is defined, Θ is a comparison operator and c is a constant in the domain of attribute x.

We build up formulae from atoms by using the following rules:

- An atom is a formula.
- If P_1 is a formula, then so are $\neg P_1$ and (P_1).
- If P1 and P_2 are formulae, then so are $P_1 \vee P_2$, $P_1 \wedge P_2$, $P_1 \Rightarrow P_2$.
- If $P_1(s)$ is a formula containing a free tuple variable s, and r is a relation then,

$\exists s \in r(P_1(s))$ and $\forall s \in r(P_1(s))$

Example queries:

Find the branch-name, loan number, customer name and amount for loans over \$1200:

$\{t \mid t \in loan \wedge t[amount] > 1200\}$.

We need those tuples on (loan_number) such that there is a tuple in loan with the amount attribute > 1200.

To express this request, we need the construct "there exists" from mathematical logic, the notation,

$\exists t \in r(Q(t))$

means "there exist a tuple t in relation r such that predicate Q(t) is true".

Using this notation, we can write the query "find the loan_number for each loan of an amount greater than \$1200" as

$\{t \mid \exists s \in loan\ (t[loan\_number] = s[loan\_number] \wedge s[amount] > 1200)\}$.

Safety of Expression:

A tuple-relational calculus expression may generate an infinite relation.

Suppose, we write the expression,

$\{t \mid \neg(t \in loan)\}$

There are infinitely many tuples that are not in loan. Most of these tuples contain values that do not even appear in the database. We do not wish to allow such expression, so we define a restriction of the tuple relational calculus formula, P.

The domain of P, denoted dom(P) is the set of all values referenced by P.

For example,

dom(t $\in$ loan $\wedge$ t[amount] > 1200) is the set containing 1200 as well as the set values appearing in loan.

Domain Relational Calculus:

Domain relational calculus uses domain variables that take on values from an attributes domain, rather than values for an entire tuple.

Formal definition:

An expression in the domain relational calculus os of the form,

$$\{<x1, x2, \ldots, xn> \mid P(x1, x2, \ldots, xn)\}$$

Where x1, x2, ……., xn represent domain variables.

P represents a formula composed of atoms.

An atom in the domain relational calculus has one of the following forms:

- **<x1, x2,…….., xn> $\in$ r**, where r is a relation on n attributes and x1, x2, …., xn are domain variables or domain constants.
- **x θ y**, where x and y are domain variables and θ is a comparison operator.
- **x θ c**, where x is a domain variable, θ is a comparison operator and c is a constant in the domain of the attribute for which x is a domain variable.

We build up formula from atoms by using the following rules.

- An atom is formula.
- If P_1 is a formula, then so are $\neg P_1$ and (P_1).
- If P_1 and P_2 are formulae, then so are $P_1 \vee P_2$, $P_1 \wedge P_2$, $P_1 \Rightarrow P_2$.
- If P_1(s) is a formula in x, where x is a free domain variable, then

$$\exists x\, (P_1(x)) \text{ and } \forall x\, (P_1(x)).$$

Example queries:

Find the loan number, branch name and amount for loans of over $1200:

$$\{<l,b,a> \mid <l,b,a> \in loan \wedge a > 1200 \}.$$

Find all loan numbers for loans with an amount greater than $1200:

$$\{<l> \mid \exists b,a\, (<l,b,a> \in loan \wedge a > 1200)\}.$$

Safety of Expression:

In the tuple relational calculus, it is possible to write expressions that may generate an infinite relation.

A similar situation arises for the domain relational calculus.

An expression such as,

$\{<l,b,a> \mid \neg(<l,b,a> \in loan)\}$ is unsafe, because it allows values in the result that are not in the domain of the expression.

For the domain relational calculus, we must be concerned also about the form of formula,

$$\{<x> \mid \exists y\,(<x,y> \in r) \land \exists z(\neg(<x,z> \in r) \land P(x,z))\}$$

UNIT – 4 NORMALIZATION

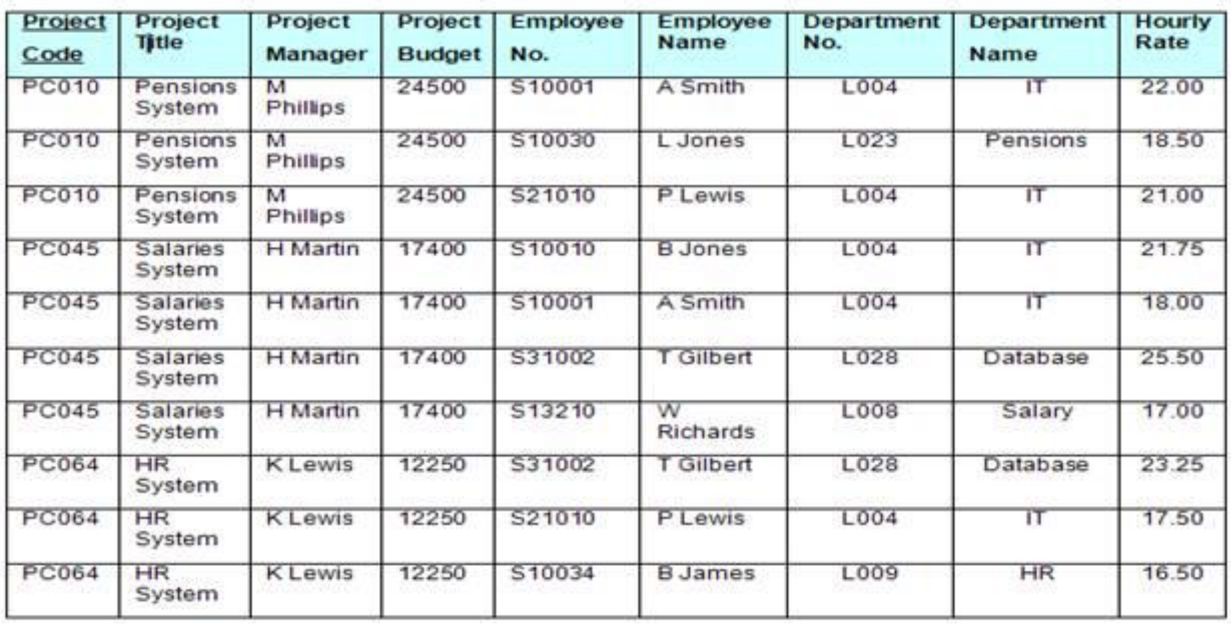
Step 1:

- Create column headings for the table for each data item on the report (**ignoring any calculated fields**). A calculated field is one that can be derived from other information on the form. In this case **total staff** and **average hourly rate**.

- Enter sample data into table. (This data is not simply the data on the report but a representative sample. In this example it shows several employees working on several projects. In this company the same employee can work on different projects and at a different hourly rate.)

- Identify a **key** for the table (and underline it).

- Remove duplicate data. (In this example, for the chosen key of Project Code, the values for Project Code, Project Title, Project Manager and Project Budget are duplicated if there are two or more employees working on the same project. **Project Code** chosen for the key and duplicate data, associated with each project code, is removed. Do not confuse duplicate data with repeating attributes which is described in the next step.



Project Code	Project Title	Project Manager	Project Budget	Employee No.	Employee Name	Department No.	Department Name	Hourly Rate
PC010	Pensions System	M Phillips	24500	S10001	A Smith	L004	IT	22.00
PC010	Pensions System	M Phillips	24500	S10030	L Jones	L023	Pensions	18.50
PC010	Pensions System	M Phillips	24500	S21010	P Lewis	L004	IT	21.00
PC045	Salaries System	H Martin	17400	S10010	B Jones	L004	IT	21.75
PC045	Salaries System	H Martin	17400	S10001	A Smith	L004	IT	18.00
PC045	Salaries System	H Martin	17400	S31002	T Gilbert	L028	Database	25.50
PC045	Salaries System	H Martin	17400	S13210	W Richards	L008	Salary	17.00
PC064	HR System	K Lewis	12250	S31002	T Gilbert	L028	Database	23.25
PC064	HR System	K Lewis	12250	S21010	P Lewis	L004	IT	17.50
PC064	HR System	K Lewis	12250	S10034	B James	L009	HR	16.50

Step 2:

Transform a table of unnormalised data into first normal form (1NF). any repeating attributes to a new table. A repeating attribute is a data field within the UNF relation that may occur with multiple values for a single value of the key. The process is as follows:

- Identify repeating attributes.

- Remove these repeating attributes to a new table together with a copy of the key from the UNF table.

- Assign a key to the new table (and underline it). The key from the original unnormalised table always becomes part of the key of the new table. A compound key is created. The value for this key must be unique for each entity occurrence.

Notes:

- After removing the duplicate data the repeating attributes are easily identified.
- In the previous table the Employee No, Employee Name, Department No, Department Name and Hourly Rate attributes are repeating. That is, there is potential for more than one occurrence of these attributes for each project code. These are the repeating attributes and have been to a new table together with a copy of the original key (ie: Project Code).
- A key of Project Code and Employee No has been defined for this new table. This combination is unique for each row in the table.

1NF Tables: Repeating Attributes Removed

Project Code	Project Title	Project Manager	Project Budget
PC010	Pensions System	M Phillips	24500
PC045	Salaries System	H Martin	17400
PC064	HR System	K Lewis	12250

-

Project Code	Employee No.	Employee Name	Department No.	Department Name	Hourly Rate
PC010	S10001	A Smith	L004	IT	22.00
PC010	S10030	L Jones	L023	Pensions	18.50
PC010	S21010	P Lewis	L004	IT	21.00
PC045	S10010	B Jones	L004	IT	21.75
PC045	S10001	A Smith	L004	IT	18.00
PC045	S31002	T Gilbert	L028	Database	25.50
PC045	S13210	W Richards	L008	Salary	17.00
PC064	S31002	T Gilbert	L028	Database	23.25
PC064	S21010	P Lewis	L004	IT	17.50
PC064	S10034	B James	L009	HR	16.50

Step 3:

Transform 1NF data into second normal form (2NF). Remove any -key attributes (partial Dependencies) that only depend on part of the table key to a new table.

What has to be determined "is field A dependent upon field B or vice versa?" This means: "Given a value for A, do we then have only one possible value for B, and vice versa?" If the answer is yes, A and B should be put into a new relation with A becoming the primary key. A should be left in the original relation and marked as a foreign key.

Ignore tables with (a) a simple key or (b) with no non-key attributes (these go straight to 2NF with no conversion).

The process is as follows:

Take each non-key attribute in turn and ask the question: is this attribute dependent on **one part** of the key?

- If yes, remove the attribute to a new table with a **copy** of the **part** of the key it is dependent upon. The key it is dependent upon becomes the key in the new table. Underline the key in this new table.

- If no, check against other part of the key and repeat above process

- If still no, ie: not dependent on either part of the key, keep attribute in current table.

Notes:

- The first table went straight to 2NF as it has a simple key (Project Code).

- Employee name, Department No and Department Name are dependent upon Employee No only. Therefore, they were moved to a new table with Employee No being the key.

- However, Hourly Rate is dependent upon both Project Code and Employee No as an employee may have a different hourly rate depending upon which project they are working on. Therefore it remained in the original table.

2NF Tables: Partial Key Dependencies Removed

Project Code	**Project Title**	**Project Manager**	**Project Budget**
PC010	Pensions System	M Phillips	24500
PC045	Salaries System	H Martin	17400
PC064	HR System	K Lewis	12250

•

Project Code	**Employee No.**	**Hourly Rate**		**Employee No.**	**Employee Name**	**Department No.**	**Department Name**
PC010	S10001	22.00		S10001	A Smith	L004	IT
PC010	S10030	18.50		S10030	L Jones	L023	Pensions
PC010	S21010	21.00		S21010	P Lewis	L004	IT
PC045	S10010	21.75		S10010	B Jones	L004	IT
PC045	S10001	18.00		S31002	T Gilbert	L028	Database
PC045	S31002	25.50		S13210	W Richards	L008	Salary
PC045	S13210	17.00		S10034	B James	L009	HR
PC064	S31002	23.25					
PC064	S21010	17.50					
PC064	S10034	16.50					

Step 4:

Data in second normal form (2NF) into third normal form (3NF).

Remove to a new table any non-key attributes that are more dependent on other non-key attributes than the table key.

What has to be determined is "is field A dependent upon field B or vice versa?" This means: "Given a value for A, do we then have only one possible value for B, and vice versa?" If the answer is yes, then A and B should be put into a new relation, with A becoming the primary key. A should be left in the original relation and marked as a foreign key.

Ignore tables with zero or only one non-key attribute (these go straight to 3NF with no conversion).

The process is as follows: If a non-key attribute is more dependent on another non-key attribute than the table key:

- Move the **dependent** attribute, together with a **copy** of the non-key attribute upon which it is dependent, to a new table.

- Make the non-key attribute, upon which it is dependent, the key in the new table. Underline the key in this new table.

- **Leave** the non-key attribute, upon which it is dependent, in the original table and mark it a **foreign key** (*).

Notes:

- The project team table went straight from 2NF to 3NF as it only has one non-key attribute.

- Department Name is more dependent upon Department No than Employee No and therefore was moved to a new table. Department No is the key in this new table and a foreign key in the Employee table.

3NF Tables: Non-Key Dependencies Removed

Project Code	**Project Title**	**Project Manager**	**Project Budget**
PC010	Pensions System	M Phillips	24500
PC045	Salaries System	H Martin	17400
PC064	HR System	K Lewis	12250

•

Project Code	**Employee No.**	**Hourly Rate**
PC010	S10001	22.00
PC010	S10030	18.50
PC010	S21010	21.00
PC045	S10010	21.75
PC045	S10001	18.00
PC045	S31002	25.50
PC045	S13210	17.00
064	S31002	23.25
PC064	S21010	17.50
PC064	S10034	16.50

Employee No.	**Employee Name**	**Department No. ***
S10001	A Smith	L004
S10030	L Jones	L023
S21010	P Lewis	L004
S10010	B Jones	L004
S31002	T Gilbert	L023
S13210	W Richards	L008
S10034	B James	L0009

-

Department No.	**Department Name**
L004	IT
L023	Pensions
L028	Database
L008	Salary
L009	HR

Summary of Normalization Rules:

That is the complete process. Having started off with an unnormalised table we finished with four normalised tables in 3NF. You will notice that duplication has been removed (apart from the keys needed to establish the links between those tables).

The process may look complicated. However, if you follow the rules **completely**, and **donot** miss out any steps, then you should arrive at the correct solution. If you omit a rule there is a high probability that you will end up with too few tables or incorrect keys.

The following normal forms were discussed in this section:

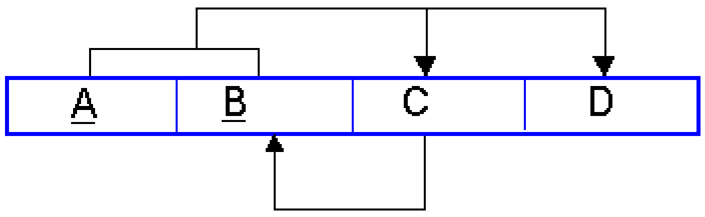
1. **First normal form:** A table is in the first normal form if it contains no repeating columns.

2. **Second normal form:** A table is in the second normal form if it is in the first normal form and contains only columns that are dependent on the whole (primary) key.

3. **Third normal form:** A table is in the third normal form if it is in the second normal form and all the non-key columns are dependent only on the primary key. If the value of a non-key column is dependent on the value of another non-key column we have a situation known as transitive dependency. This can be resolved by removing the columns dependent on non-key items to another table.

Boyce Codd Normal Form

Usually tables that are in Third Normal Form are already in Boyce Codd Normal Form. Boyce Codd Normal Form (BCNF) is considered a special condition of third Normal form. A table is in BCNF if every determinant is a candidate key. A table can be in 3NF but not in BCNF. This occurs when a non key attribute is a determinant of a key attribute. The dependency diagram may look like the one below



The table is in 3NF. A and B are the keys and C and D depend on both A and B. There are no transitive dependencies existing between the non key attributes, C and D.

The table is not in BCNF because a dependency exists between C and B. In other words if we know the value of C we can determine the value of B.

We can also show the dependencies as

A B → C D

C → B

Fourth normal form :

Fourth normal form (4NF) is a normal form used in database normalization. Introduced by Ronald Fagin in 1977, 4NF is the next level of normalization after Boyce–Codd normal form (BCNF). Whereas the second, third, and Boyce–Codd normal forms are concerned with functional dependencies, 4NF is concerned with a more general type of dependency known as a multivalued dependency. A Table is in 4NF if and only if, for every one of its non-trivial multivalued dependencies $X \twoheadrightarrow Y$, X is a super key—that is, X is either a candidate key or a superset thereof

Multivalued Dependencies:

If the column headings in a relational database table are divided into three disjoint groupings X, Y, and Z, then, in the context of a particular row, we can refer to the data beneath each group of headings as x, y, and z respectively. A multivalued dependency $X \twoheadrightarrow Y$ signifies that if we choose any x actually occurring in the table (call this choice x_c), and compile a list of all the x_c y z combinations that occur in the table, we will find that x_c is associated with the same y entries regardless of z. So essentially the presence of z provides no useful information to constrain the possible values of y.

A **trivial multivalued dependency** $X \twoheadrightarrow Y$ is one where either Y is a subset of X, or X and Y together form the whole set of attributes of the relation.

A functional dependency is a special case of multivalued dependency. In a functional dependency $X \rightarrow Y$, every x determines exactly one y, never more than one.

UNIT - V
SQL COMMANDS

5.1 DDL, DML, DCL OPERATIONS

SQL commands are instructions used to communicate with the database to perform specific task that work with data. SQL commands can be used not only for searching the database but also to perform various other functions like, for example, you can create tables, add data to tables, or modify data, drop the table, set permissions for users. SQL commands are grouped into four major categories depending on their functionality:

☺ **Data Definition Language (DDL)** - These SQL commands are used for creating, modifying, and dropping the structure of database objects. The commands are CREATE, ALTER, DROP, RENAME, and TRUNCATE.
☺ **Data Manipulation Language (DML)** - These SQL commands are used for storing, retrieving, modifying, and deleting data. These commands are SELECT, INSERT, UPDATE, and DELETE.
☺ **Transaction Control Language (TCL)** - These SQL commands are used for managing changes affecting the data. These commands are COMMIT, ROLLBACK, and SAVEPOINT.
☺ **Data Control Language (DCL)** - These SQL commands are used for providing security to database objects. These commands are GRANT and REVOKE.

SQL SELECT STATEMENT

The most commonly used SQL command is SELECT statement. The SQL SELECT statement is used to query or retrieve data from a table in the database. A query may retrieve information from specified columns or from all of the columns in the table. To create a simple SQL SELECT Statement, you must specify the column(s) name and the table name. The whole query is called SQL SELECT Statement.

Syntax of SQL SELECT Statement:

SELECT column_list FROM table-name [WHERE clause]
[GROUP BY Clause][HAVING Clause][ORDER BY Clause];

Table-name is the name of the table from which the information is retrieved.

- column_list includes one or more columns from which data is retrieved.
- The code within the brackets is optional.

Database table student_details;

id	**first_name**	**last_name**	**age**	**subject**	**games**
100	Rahul	Sharma	10	Science	Cricket
101	Anjali	Bhagwat	12	Maths	Football
102	Stephen	Fleming	09	Science	Cricket
103	Shekar	Gowda	18	Maths	Badminton
104	Priya	Chandra	15	Economics	Chess

NOTE: These database tables are used here for better explanation of SQL commands. In reality, the tables can have different columns and different data.
For example, consider the table student_details. To select the first name of all the students the query would be like:

```
SELECT first_name FROM student_details;
```

NOTE: The commands are not case sensitive. The above SELECT statement can also be written as "select first_name from students_details;"
You can also retrieve data from more than one column. For example, to select first name and last name of all the students.

```
SELECT first_name, last_name FROM student_details;
```

You can also use clauses like WHERE, GROUP BY, HAVING, ORDER BY with SELECT statement. We will discuss these commands in coming chapters.
NOTE: In a SQL SELECT statement only SELECT and FROM statements are mandatory. Other clauses like WHERE, ORDER BY, GROUP BY, HAVING are optional.
How to use expressions in SQL SELECT Statement?

- Expressions combine many arithmetic operators, they can be used in SELECT, WHERE and ORDER BY Clauses of the SQL SELECT Statement.
- Here we will explain how to use expressions in the SQL SELECT Statement. About using expressions in WHERE and ORDER BY clause, they will be explained in their respective sections.
- The operators are evaluated in a specific order of precedence, when more than one arithmetic operator is used in an expression. The order of evaluation is: parentheses, division, multiplication, addition, and subtraction. The evaluation is performed from the left to the right of the expression.

For example: If we want to display the first and last name of an employee combined together, the SQL Select Statement would be like

```
SELECT first_name || ' ' || last_name FROM employee;
```

Output:

```
first_name   ||      ||     last_name

Rahul                       Sharma
Anjali                      Bhagwat
Stephen                     Fleming
Shekar                      Gowda
Priya                       Chandra
```

You can also provide aliases as below.

```
SELECT first_name || ' ' || last_name AS emp_name FROM employee;
```

Output:

```
emp_name

Rahul       Sharma
Anjali      Bhagwat
Stephen     Fleming
Shekar      Gowda
Priya       Chandra
```

SQL Alias

SQL Aliases are defined for columns and tables. Basically aliases is created to make the column selected more readable.

For Example: To select the first name of all the students, the query would be like:

Aliases for columns:

```
SELECT first_name AS Name FROM student_details;   (or)
```

```
SELECT first_name Name FROM student_details;
```

In the above query, the column first_name is given a alias as 'name'. So when the result is displayed the column name appears as 'Name' instead of 'first_name'.

Output:

```
Name

Rahul       Sharma
Anjali      Bhagwat
Stephen     Fleming
Shekar      Gowda
Priya       Chandra
```

Aliases for tables:

```
SELECT s.first_name FROM student_details s;
```

In the above query, alias 's' is defined for the table student_details and the column first_name is selected from the table.

Aliases is more useful when

- There are more than one tables involved in a query,
- Functions are used in the query,
- The column names are big or not readable,
- More than one columns are combined together

SQL WHERE CLAUSE

The WHERE Clause is used when you want to retrieve specific information from a table excluding other irrelevant data. For example, when you want to see the information about students in class 10th only then you do need the information about the students in other class. Retrieving information about all the students would increase the processing time for the query.

So SQL offers a feature called WHERE clause, which we can use to restrict the data that is retrieved. The condition you provide in the WHERE clause filters the rows retrieved from the table and gives you only those rows which you expected to see. WHERE clause can be used along with SELECT, DELETE, UPDATE statements.

<u>Syntax of SQL WHERE Clause:</u>

```
WHERE {column or expression} comparison-operator value
```

<u>Syntax for a WHERE clause with Select statement is:</u>

```
SELECT column_list FROM table-name WHERE condition;
```

- column or expression - Is the column of a table or a expression
- comparison-operator - operators like = < > etc.
- value - Any user value or a column name for comparison

For Example: To find the name of a student with id 100, the query would be like:

```
SELECTfirst_name,last_nameFROMstudent_detailsWHERE id = 100;
```

Comparison Operators and Logical Operators are used in WHERE Clause. These operators are discussed in the next chapter.

NOTE: Aliases defined for the columns in the SELECT statement cannot be used in the WHERE clause to set conditions. Only aliases created for tables can be used to reference the columns in the table.

<u>How to use expressions in the WHERE Clause?</u>

Expressions can also be used in the WHERE clause of the SELECT statement.

For example: Lets consider the employee table. If you want to display employee name, current salary, and a 20% increase in the salary for only those products where the percentage increase in salary is greater than 30000, the SELECT statement can be written as shown below

```
SELECTname,salary,salary*1.2ASnew_salaryFROMemployeeWHEREsalary*1.2>30000;
```

<u>Output:</u>

name	salary	new_salary
Hrithik	35000	37000
Harsha	35000	37000
Priya	30000	360000

NOTE: Aliases defined in the SELECT Statement can be used in WHERE Clause.

<u>**SQL INSERT Statement**</u>

The INSERT Statement is used to add new rows of data to a table.

We can insert data to a table in two ways,

1) Inserting the data directly to a table.

Syntax for SQL INSERT is:

```
INSERT INTO TABLE_NAME[(col1,col2,col3,...coln)]
```

```
VALUES (value1, value2, value3,...valueN);
```

- col1, col2,...colN -- the names of the columns in the table into which you want to insert data.

While inserting a row, if you are adding value for all the columns of the table you need not specify the column(s) name in the sql query. But you need to make sure the order of the values is in the same order as the columns in the table. The sql insert query will be as follows

```
INSERT INTO TABLE_NAME VALUES (value1, value2, value3,...valuen);
```

For Example: If you want to insert a row to the employee table, the query would be like,

```
INSERT INTO employee (id, name, dept, age, salary location) VALUES (105, 'Srinath',
'Aeronautics', 27, 33000);
```

NOTE:When adding a row, only the characters or date values should be enclosed with single quotes.

If you are inserting data to all the columns, the column names can be omitted. The above insert statement can also be written as,

```
INSERT INTO employee VALUES (105, 'Srinath', 'Aeronautics', 27, 33000);
```

Inserting data to a table through a select statement.

Syntax for SQL INSERT is:

```
INSERT INTO table_name[(column1,column2,...columnN)]
SELECTcolumn1,column2,...columnN
FROM table_name [WHERE condition];
```

For Example: To insert a row into the employee table from a temporary table, the sql insert query would be like,

```
INSERT INTO employee (id, name, dept, age, salary location) SELECT emp_id,
emp_name, dept, age, salary, location FROM temp_employee;
```

If you are inserting data to all the columns, the above insert statement can also be written as,

```
INSERT INTO employee SELECT * FROM temp_employee;
```

NOTE: We have assumed the temp_employee table has columns emp_id, emp_name, dept, age, salary, location in the above given order and the same datatype.

<u>IMPORTANT NOTE:</u>

1) When adding a new row, you should ensure the data type of the value and the column matches
2) You follow the integrity constraints, if any, defined for the table.

<u>SQL UPDATE Statement</u>

The UPDATE Statement is used to modify the existing rows in a table.

The Syntax for SQL UPDATE Command is:

```
UPDATE table_name SET column_name1=value1, column_name2=value2, .. [WHERE
condition]
```

- table_name - the table name which has to be updated.
- column_name1, column_name2.. - the columns that gets changed.
- value1, value2... - are the new values.

NOTE: In the Update statement, WHERE clause identifies the rows that get affected. If you do not include the WHERE clause, column values for all the rows get affected.

For Example: To update the location of an employee, the sql update query would be like,

```
UPDATEemployeeSETlocation='Mysore'WHERE id = 101;
```

To change the salaries of all the employees, the query would be,

```
UPDATEemployeeSET salary = salary + (salary * 0.2);
```

SQL DELETE STATEMENT

The DELETE Statement is used to delete rows from a table.
The Syntax of a SQL DELETE statement is:

```
DELETE FROM table_name [WHERE condition];
```

- table_name -- the table name which has to be updated.

NOTE:The WHERE clause in the sql delete command is optional and it identifies the rows in the column that gets deleted. If you do not include the WHERE clause all the rows in the table is deleted, so be careful while writing a DELETE query without WHERE clause.
For Example: To delete an employee with id 100 from the employee table, the sql delete query would be like,

```
DELETE FROM employee WHERE id = 100;
```

To delete all the rows from the employee table, the query would be like,

```
DELETE FROM employee;
```

SQL TRUNCATE STATEMENT

The SQL TRUNCATE command is used to delete all the rows from the table and free the space containing the table.
Syntax to TRUNCATE a table:

```
TRUNCATE TABLE table_name;
```

For Example: To delete all the rows from employee table, the query would be like,

```
TRUNCATE TABLE employee;
```

Difference between DELETE and TRUNCATE Statements:

DELETE Statement: This command deletes only the rows from the table based on the condition given in the where clause or deletes all the rows from the table if no condition is specified. But it does not free the space containing the table.
TRUNCATE statement: This command is used to delete all the rows from the table and free the space containing the table.

SQL DROP STATEMENT:

The SQL DROP command is used to remove an object from the database. If you drop a table, all the rows in the table is deleted and the table structure is removed from the database. Once a table is dropped we cannot get it back, so be careful while using RENAME command. When a table is dropped all the references to the table will not be valid.
Syntax to drop a sql table structure:

```
DROP TABLE table_name;
```

For Example: To drop the table employee, the query would be like

```
DROP TABLE employee;
```

Difference between drop and truncate statement:

If a table is dropped, all the relationships with other tables will no longer be valid, the integrity constraints will be dropped, grant or access privileges on the table will also be dropped, if want use the table again it has to be recreated with the integrity constraints, access privileges and the relationships with other tables should be established again. But, if a table is truncated, the table structure remains the same, therefore any of the above problems will not exist.

SQL ALTER TABLE STATEMENT

The SQL ALTER TABLE command is used to modify the definition (structure) of a table by modifying the definition of its columns. The ALTER command is used to perform the following functions.

1)Add, drop, modify table columns
2)Add and drop constraints
3) Enable and Disable constraints

Syntax to add a column

```
ALTER TABLE table_name ADD column_name datatype;
```

For Example: To add a column "experience" to the employee table, the query would be like

```
ALTER TABLE employee ADD experience number(3);
```

Syntax to drop a column

```
ALTER TABLE table_name DROP column_name;
```

For Example: To drop the column "location" from the employee table, the query would be like

```
ALTER TABLE employee DROP location;
```

Syntax to modify a column

```
ALTER TABLE table_name MODIFY column_name datatype;
```

For Example: To modify the column salary in the employee table, the query would be like

```
ALTER TABLE employee MODIFY salary number(15,2);
```

SQL RENAME COMMAND

The SQL RENAME command is used to change the name of the table or a database object.
If you change the object's name any reference to the old name will be affected. You have to manually change the old name to the new name in every reference.

Syntax to rename a table

```
RENAME old_table_name To new_table_name;
```

For Example: To change the name of the table employee to my_employee, the query would be like

```
RENAME employee TO my_employee;
```

5.2 SQL INTEGRITY CONSTRAINTS

Integrity Constraints are used to apply business rules for the database tables.
The constraints available in SQL are Foreign Key, Not Null, Unique, Check.
Constraints can be defined in two ways

1. the constraints can be specified immediately after the column definition. This is called column-level definition.
2. The constraints can be specified after all the columns are defined. This is called table-level definition.

1) SQL Primary key:

This constraint defines a column or combination of columns which uniquely identifies each row in the table.
Syntax to define a Primary key at column level:

column name datatype [CONSTRAINT constraint_name] PRIMARY KEY

Syntax to define a Primary key at table level:

[CONSTRAINT constraint_name] PRIMARY KEY (column_name1,column_name2,..)

column_name1, column_name2 are the names of the columns which define the primary Key.
The syntax within the bracket i.e. **[CONSTRAINT constraint_name]** is optional.
For Example: To create an employee table with Primary Key constraint, the query would be like.

Primary Key at table level:

CREATE TABLE employee (id number(5) PRIMARY KEY,name char(20),dept char(10), age number(2), salary number(10),location char(10));
or
CREATE TABLE employee (id number(5) CONSTRAINT emp_id_pk PRIMARY KEY, name char(20),dept char(10),age number(2),salary number(10),location char(10));

Primary Key at table level:

CREATE TABLE employee (id number(5), name char(20),dept char(10),age number(2), salary number(10),location char(10),CONSTRAINT emp_id_pk PRIMARY KEY (id));

2) SQL Foreign key or Referential Integrity :

This constraint identifies any column referencing the PRIMARY KEY in another table. It establishes a relationship between two columns in the same table or between different tables. For a column to be defined as a Foreign Key, it should be a defined as a Primary Key in the table which it is referring. One or more columns can be defined as Foreign key.

Syntax to define a Foreign key at column level:
[CONSTRAINT constraint_name] REFERENCES Referenced_Table_name(column_name)

Syntax to define a Foreign key at table level:
[CONSTRAINT constraint_name] FOREIGN KEY(column_name) REFERENCES referenced_table_name(column_name);

For Example:
1) Lets use the "product" table and "order_items".

<u>**Foreign Key at column level:**</u>
CREATE TABLE product (product_id number(5) CONSTRAINT pd_id_pk PRIMARY KEY, product_name char(20),supplier_name char(20),unit_price number(10));

CREATE TABLE order_items (order_id number(5) CONSTRAINT od_id_pk PRIMARY KEY,product_id number(5) CONSTRAINT pd_id_fk REFERENCES, product(product_id),product_name char(20), supplier_name char(20),unit_price number(10));

<u>**Foreign Key at table level:**</u>
CREATE TABLE order_items(order_id number(5) ,product_id number(5), product_name char(20),supplier_name char(20),unit_price number(10)CONSTRAINT od_id_pk PRIMARY KEY(order_id),CONSTRAINT pd_id_fk FOREIGN KEY(product_id) REFERENCES product(product_id));

<u>**3) SQL Not Null Constraint :**</u>
This constraint ensures all rows in the table contain a definite value for the column which is specified as not null. Which means a null value is not allowed.

<u>Syntax to define a Not Null constraint:</u>
[CONSTRAINT constraint name] NOT NULL

For Example**: To create a employee table with Null value, the query would be like**
CREATE TABLE employee(id number(5),name char(20) CONSTRAINT nm_nn NOT NULL,dept char(10),age number(2),salary number(10),location char(10));

<u>**4) SQL Unique Key:**</u>
This constraint ensures that a column or a group of columns in each row have a distinct value. A column(s) can have a null value but the values cannot be duplicated.
<u>Syntax to define a Unique key at column level:</u>

[CONSTRAINT constraint_name] UNIQUE
Syntax to define a Unique key at table level:

[CONSTRAINT constraint_name] UNIQUE(column_name)
CREATE TABLE employee(id number(5) PRIMARY KEY,name char(20),dept char(10), age number(2),salary number(10),location char(10) UNIQUE);
(or)
CREATE TABLE employee (id number(5) PRIMARY KEY,name char(20),dept char(10), age number(2), salary number(10),location char(10) CONSTRAINT loc_un UNIQUE);

<u>Unique Key at table level:</u>
CREATE TABLE employee (id number(5) PRIMARY KEY,name char(20),dept char(10),age number(2),salary number(10),location char(10),CONSTRAINT loc_un UNIQUE(location));

<u>**5) SQL Check Constraint :**</u>

This constraint defines a business rule on a column. All the rows must satisfy this rule. The constraint can be applied for a single column or a group of columns.

Syntax to define a Check constraint:
[CONSTRAINT constraint_name] CHECK (condition)

For Example: **In the employee table to select the gender of a person, the query would be like**
Check Constraint at column level:
CREATE TABLE employee (id number(5) PRIMARY KEY, name char(20), dept char(10), age number(2), gender char(1) CHECK (gender in ('M','F')), salary number(10), location char(10));
Check Constraint at table level:
CREATE TABLE employee (id number(5) PRIMARY KEY, name char(20), dept char(10), age number(2), gender char(1), salary number(10), location char(10), CONSTRAINT gender_ck CHECK (gender in ('M','F')));

BUILT IN FUNCTIONS:

SQL provides a number of predefined functions that can be called from within an sql statement. It is used to manipulate data items.

Advantage of functions:

- Functions can be used to perform complex calculations on data.
- Functions can modify individual data items.
- Functions can very easily manipulate output for group of rows.
- Functions can alter date formats for display.

Types of function:

1. **Character functions**
2. **Arithmetic functions**
3. **Other functions**
4. **Date functions**
5. **Aggregate /group functions**
6. **Conversion functions**

5.3 STRING FUNCTION

These function all take arguments in the character family and return character values. The majority of the functions return a **varchar2**values, except where noted. The return type of character function is subject to the same restrictions as the base database type, namely that varchar2 values are limited to 2000 characters and char values are limited to 255 characters. When in procedural statements, they can be assigned to either varchar2 or char pl/sql variables.

Types:
CHR(X)
CONCAT(STRING1,STRING2)
LOWER(STRING)
UPPER(STRING)

LPAD(CHAR1,N[CHAR2])
RPAD(CHAR1,N[CHAR2])
SOUNDEX(STRING)
LTRIM(STRING,'CHAR/S')
RTRIM(STRING,'CHAR/S')
REPLACE(STRING,SEARCH_STR[,REPLACE_STR])
SUBSTR(STRING,M[,N])
TRANSLATE(STRING,FROM_STR,TO_STR)
ASCII(STRING)
INSTR(STRING,CHAR)
LENGTH(STRING)
INITCAP(STRING)

CHR:
Syntax: CHR(x)
Purpose returns the character that has the values equivalent to x in the database character set. CHR and ASCII are opposite functions. CHR returns the character given the character number, and ASCII returns the character number given the character.
Example: **SELECT CHR(37)a,CHR(100)b,CHR(101)c from dual;**
Output:

```
A       B       C
-       -       ..
%       d       e
```

Dual is small oracle table created for testing functions or doing quick calculations which has on row and column. Since oracle's many function work on both columns on literals. Some oracle function use just literals in these situations **DUAL** table can be used.

CONCAT:
Syntax: CONCAT (String1,String2)
Purpose returns string1 concatenated with string2. This function is identical to the || operator.
Example: **SELECT CONCAT('Information','technology')"computer" from dual;**
Output:

```
        Computer
        -------------
    Information technology
```

REPLACE:
Syntax: REPLACE (string, search_str [replace_str])
Purpose: returns string with every occurrence of search_str replaced with replace_str. If replace_str is not specified, all occurrences of search_str are removed. REPLACE is a superset of the functionality provided by TRANSLATE.

5.4 NUMBER FUNTIONS

These functions take number arguments and return number values. **ABS**
Types:
ABS(N)
CEIL(N)

FLOOR(N)
MODE(M,N)
POWER(M,N)
SORT(N)
TRUNC(M,[N])
ROUND(M,[N])
EXP(N)

<u>ABS(X)</u>
Syntax: ABS(X)
Purpose returns the absolute value of x.
Example:
SELECT ABS(-7),ABS(7) FROM DUAL;
<u>Output</u>

```
        ABS(-7)        ABS(7)
        -------- -     ----------
               7              7
```

<u>CEIL</u>
Syntax: CEIL(X)
Purpose returns the Smallest integer greater than or equal to x.
Example:
SELECT CEIL(18.1),CEIL(-18.1) FROM DUAL;
<u>Output</u>

```
        CEIL(18.1)    CEIL(-18.1)
        -------- -     ----------
               19             -18
```

<u>FLOOR</u>
Syntax: FLOOR(X)
Purpose returns the largest integer equal to or less than x.
Example:
SELECT FLOOR(-23.5),FLOOR(23.5) FROM DUAL;
<u>Output</u>

```
        FLOOR(-23.5)          FLOOR(23.5)
        -------- -     ----------
               -24            23
```

<u>5.5 SELECT DISTINCT VALUES</u>

It avoids repeatation from the sql table.
It avoids duplication of data.
Example:
SQL> select distinct(eid) from child;

```
       EID
----------
       100
       200
       201
       202
```

```
      203

SQL> select * from child;

            EID    ITEMNO ITEMNAME   AMOUNT DATEODT
          ---------- ----------    ----------         ----------     ---------
            100       5          Pen            26            20-DEC-00
            200       6          Pencil         13            21-DEC-01
            201       4          Eraser          5            12-JAN-00
            202       5          Gum            23            12-DEC-00
            203       6          Paper          23            17-DEC-00
            100      20           hh            55            12-DEC-00

6 rows selected.

SQL> select distinct(eid) from child;

      EID
----------
      100
      200
      201
      202
      203
```

5.6 WORKING WITH NULL VALUES

A column value is NULL if it does not exist. The IS NULL operator is used to display all the rows for columns that do not have a value.

- Null values are not 0 or blank.
- It represents an unknown or inapplicable value.
- It cannot be compared using the relational and/or logical operators.
- The special operator **IS** used with the keyword **NULL** locate null values.

SELECT ENAME FROM EMP WHERE COMM. IS NULL;
(The above command will list the employees who is not getting commission).

SELECT EMPNO,ENAME,JOB,SALFROM EMP WHERE MGR IS NULL;
(the above command will list the employee who does not report to anybody(manager is NULL).

5.7 PSEUDO COLUMNS

Pseudo columns are additional functions that can be called only from SQL statement. Syntactically, they are treated like columns in a table. However, they don't actually exist in the same way that table columns do. They are:

- **Currtval**
- **Nextval**
- **Rowid**
- **Rownum**

Currval and nextval:

These two pseudo columns are used with sequences. A Sequence is an Oracle object that is used to generate unique numbers. Currval returns the current value of the sequence and Nextval increments the sequence and returns the new value. Both Currval and Nextval return NUMBER values.

Syntax for using Nextval:

SEQUENCE_NAME.NEXTVAL

Syntax for using Currval:

SEQUENCE_NAME.CURRVAL

Rowid:

The Rowid pseudo column is used the select list of query. It return the Rowid of that particular row. The external format of a Rowid is an 18 characters string. All Rowid will be unique for that table. No Rowid will be duplicated in a table. Using a Rowid you can identify a row individually. The following steps illustrate this:

Let us create the table called ROWID_SAMPLE

**Step 1: CREATE TABLE ROWID_SAMPLE (ID NUMBER (3),
NAME VARCHAR2(20), FEES NUMBER);**

Table Created.
Let us insert some duplicate rows in ROWID_SAMPLE table.

**Step 2:INSERT INTO ROWID_SAMPLE VALUES(1, 'RANJITH', 4500);
INSERT INTO ROWID_SAMPLE VALUES(2, 'ANNAMALAI', 6000);
INSERT INTO ROWID_SAMPLE VALUES(1, 'RANJITH', 4500);
INSERT INTO ROWID_SAMPLE VALUES(7, 'URMILA', 7000);
INSERT INTO ROWID_SAMPLE VALUES(2, 'ANNAMALAI', 6000);**

Note: In the above insert statement rows 1,3 and 2,5 are called duplicate rows.
Let us display the records from ROWID_SAMPLE table.

Step 3:SQL>SELECT * FROM ROWID_SAMPLE;

Output:

```
ID        NAME                    FEES

1       RANJITH             4500
2       ANNAMALAI           6000
3       RANJITH             4500
4       URMILA              7000
5       ANNAMALAI           6000
```

Now, let us select the rowid from ROWID_SAMPLE table.

Step 4: SQL>SELECT ROWID FROM ROWID_SAMPLE;

Output:

```
                 ROWID

AAADAEAABAAAJNVAAA
AAADAEAABAAAJNVAAB
AAADAEAABAAAJNVAAC
AAADAEAABAAAJNVAAD
AAADAEAABAAAJNVAAE
```

Rownum

Rownum will return the current roe number in a query. It is useful for limiting the total number or rows, and it is used primarily in the WHERE clause of queries and SET clause of UPDATE statements. For example if we want to display the first 5 records from EMP table, then we would query like:

SQL> SELECT EMPNO, ENAME, SAL FROM EMP WHER ROWNUM <=5;

Output:

```
EMPNO    ENAME     SAL

7369     SMITH     1300
7499     ALLEN     2100
7521     WARD      1750
7566     JONES     2975
7624     MARTIN    1750
```

The first row has ROWNUM 1, the second has ROWNUM 2, and so on.

5.8 GROUPING AND ORDERING DATA

Group functions are those statistical functions which gives information about a group of values taken as whole. The aggregate functions produce a single value for a entire group or table. In all group function, NULLs are ignored. These functions are valid in the select list of a query and the GROUP BY clause only.

COUNT:

Syntax: **COUNT**(*|[DISTINCT|ALL]col)

Purpose count function determines the number of rows or non-null column values. If *is passed then the total number of rows is returned.
Example: **SELECT COUNT(*)FROM EMP;**
<u>Output:</u>

```
COUNT(*)
-----------
   14
```

<u>MAX:</u>
Syntax: **MAX**([DISTINCT|ALL]col)
Purpose returns the maximum value of the select list item. **DISTINCT** and **All** have no effect, since the maximum value would be the same in either case.
Example: **SELECT MAX(SAL)FROM EMP;**
<u>Output:</u>

```
MAX(SAL)
-----------
  5000
```

<u>MIN:</u>
Syntax: **MIN**([DISTINCT|ALL]col)
Purpose returns the minimum value of the select list item. **DISTINCT** and **All** have no effect, since the minimum value would be the same in either case.
Example: **SELECT MIN(COMM)"MIN-COMM",MIN(SAL)"MIN-SAL"FROM EMP;**
<u>Output:</u>

```
MIN-COMM          MIN-SAL
-----------       -----------
    0                 800
```

<u>SUM:</u>
Syntax: **SUM**([DISTINCT|ALL]col)
Purpose returns the Sum of the value for the select list item.
Example: **SELECT SUM(SAL)FROM EMP;**
<u>Output:</u>

```
SUM(SAL)
-----------
  29025
```

<u>AVG:</u>
Syntax: **AVG**([DISTINCT|ALL]col)
Purpose returns the average of the column values.
Example: **SELECT AVG(SAL)FROM EMP;**
<u>Output:</u>

```
AVG(SAL)
-----------
20373.2143
```

<u>Order by clause:</u>
Displaying the results in sorted order

1. Sql uses the **ORDER BY** clause to impose an order on the result of a query.
2. **ORDER BY** clause is used with **SELECT** statement.
3. One or more columns and/or expressions can be specified in **ORDER BY** clause.

The **ORDERBY** clause orders the query output according to the values in one or more selected columns. Multiple columns are ordered one with another, and the user can specify whether to order them in ascending or descending order. Ascending is default.

Syntax:
SELECT<COLUMNS>FROM <TABLE-NAME>ORDERBY[<COLUMN-NAME>,<COLUMN-NAME>]ASC/DESC;
Example:
SELECT*FROM EMP ORDERBY ENAME;

Sorting the result on multiple columns:

Suppose after sorting the emp table in ascending order of their **DEPTNO,** we wanted to sort further with in each salary in descending order.

Example:
SELECT EMPNO, ENAME, DEPTNO, SAL, FROM EMP ORDER BY DEPTNO, SAL DESC;

Ordering output by column number:
In place of column names, we can use number to indicate the fields being used to order the output. These numbers will refer no to order of the columns in the table, but to their orders in the output.
Example:
SELECT EMPNO, ENAME, SAL, FROM EMP ORDER BY 2;

Group by clause;
The group by clause is parallel to the order by clause. Order by acts upon rows wheras group by acts upon groups. So far we were using group function (**MIN, MAX, AVG, SUM, COUNT, Etc**.)to find something about the whole set of rows.

Example:
SELECT MAX(SAL), MIN(SAL), AVG(SAL), COUNT(*), SUM(SAL) FROM EMP;
Output:
```
MAX(SAL)   MIN(SAL)    AVG(SAL)    COUNT(*)    SUM(SAL)
---------  ---------   ----------  ---------   ----------
  5000        800      2073.2143       14        29025
```

Grouping rows based on multiple columns:
You can group the rows of the table based on more than one column.

Example:
SELECT DEPTNO,JOB,COUNT(*)FROM EMP GROUP BY DEPTNO,JOB;
Output:
```
DEPTNO          JOB          COUNT(*)
------------    ------- -    --------------
10              CLERK            1
10              MANAGER          1
10              PRESIDENT        1
20              ANALYST          2
20              CLERK            2
```

5.9 SQL SUBQUERY

Sub query or Inner query or Nested query is a query in a query. A subquery is usually added in the WHERE Clause of the sql statement. Most of the time, a subquery is used when you know how to search for a value using a SELECT statement, but do not know the exact value. Subqueries are an alternate way of returning data from multiple tables.Subqueries can be used with the following sql statements along with the comparision operators like =, <, >, >=, <= etc.

Advantages of sub-queries:
1. Sub-queries allow a developer to built powerful command out of simple ones.
2. The nested sub queries are very useful when you need to select rows from a table with a condition that depends on the data in the table itself.

- **SELECT**
- **INSERT**
- **UPDATE**
- **DELETE**

For Example:
1) Usually, a subquery should return only one record, but sometimes it can also return multiple records when used with operators like IN, NOT IN in the where clause. The query would be like,

```
SELECT first_name, last_name, subject
FROM student_details
WHERE games NOT IN ('Cricket', 'Football');
```

The output would be similar to:

first_name	last_name	subject
Shekar	Gowda	Badminton
Priya	Chandra	Chess

2) Lets consider the student_details table which we have used earlier. If you know the name of the students who are studying science subject, you can get their id's by using this query below,

```
SELECT id, first_name
FROM student_details
WHERE first_name IN ('Rahul', 'Stephen');
```

but, if you do not know their names, then to get their id's you need to write the query in this manner,

```
SELECT id, first_name
FROM student_details
WHERE first_name IN (SELECT first_name
FROM student_details
WHERE subject= 'Science');
```

Output:

id	first_name
100	Rahul
102	Stephen

In the above sql statement, first the inner query is processed first and then the outer query is processed.

3) Sub query can be used with INSERT statement to add rows of data from one or more tables to another table. Lets try to group all the students who study Maths in a table 'maths_group'.

```
INSERT INTO maths_group(id, name)
SELECT id, first_name || ' ' || last_name
FROM student_details WHERE subject= 'Maths'
```

4) A subquery can be used in the SELECT statement as follows. Lets use the product and order_items table defined in the sql_joins section.
select p.product_name, p.supplier_name, (select order_id from order_items where product_id = 101) as order_id from product p where p.product_id = 101

product_name	supplier_name	order_id
Television	Onida	5103

CORRELATED SUBQUERY

A query is called correlated subquery when both the inner query and the outer query are interdependent. For every row processed by the inner query, the outer query is processed as well. The inner query depends on the outer query before it can be processed.

```
SELECT p.product_name FROM product p
WHERE p.product_id = (SELECT o.product_id FROM order_items o
WHERE o.product_id = p.product_id);
```

NOTE:
1) You can nest as many queries you want but it is recommended not to nest more than 16 subqueries in oracle.
2) If a subquery is not dependent on the outer query it is called a non-correlated subquery.

5.10 SQL JOINS

Using sql joints, you can retrieve data more than one table or view using the keys[primary&foreign] references
The Syntax for joining two tables is:

```
SELECT col1, col2, col3...
FROM table_name1, table_name2
WHERE table_name1.col2 = table_name2.col1;
```

The select statements contains the columns to retrieve and may came form two or more tables. If the selected column exists both table specify the table[table. column]. You have to specify the tables in the from clause and the joint condition is done in the where clause. In the where clause you need the table name and dot followed by column name. the column name in the where clause is the joint column[keys]. And &Or also normally used to make multiple joint conditions. There are about four basic types of joints equality joints self joints, outer joints and inequality

Equality joints:

Equality joints happened when two tables are joined based on values in one table being equal to values in another table.

Example:

Product table

Product id	**name**	**description**	**price**	**cost**
100000000	printerinkjet300	colourprinter	120	80
100000001	printer1220cxi	inkjetprinter	200	130
100000002	printerphoto890	inkjetprinter	250	200
100000003	printerphoto890	inkjetprinter	300	270

Inventory table

Product id	**qty_on_hand**	**qty_on_order**	**min_req**	**mx_req**
100000000	20	0	10	25
100000001	10	5	2	15
100000002	2	10	1	12
100000003	1	15	1	12

The query to perform equality joints might look like this:

```
SELECT product.product_id,name,price,qty_on_hand,qty_on_order
FROM product, inventory
WHERE product.product_id=inventory.product_id;
```

Product table is specified in the select statement to issue the product_id and the reason is , product id exist both table and if you do not specified which table to select from, you will receive ambiguous error. The query will select all the selected rows from both tables since there is always product_id equal to product_id in the other table.

Outer joints

Outer joint is joint condition where all the rows of a table are selected along with their matching rows in other table .
For example ,you might want to select all your customers along with their order. If they have orders .**+sign** is used in the WHERE clause beside the child table with in parentheses. The following is sql joint statements to select every customer in the customer table along with their orders.if they have order and if they do not and order it will select blanks .

```
SELECT customer.customer_id,firstname,lastname,item_id,qty_ordered,price
FROM customer,order
WHERE customer.customer_id=order.customer_id(+);
```

Self joints

Self joint is joint of a table by it self. For example if you want to retriew customers whom ordered same product twice or more assuming there is num_order that keeps track the number of orders customers made. Here is how you would do this using a self joints:

```
SELECT o1.customer_id
FROM ordero1,ordero2
WHERE o1.item_id=o2.item_id and o1.num_order>1;
```

This query simplify created two table alias, o1 and o2 which represents two copies of the table order then compares if item_id exists both table when order is placed two or more times by a customer.

Inequality joints

Inequality joints is when each record in the one table is joined with every record in the second table using operators.SQL Non equi joins It is a sql join condition which makes use of some comparison operator other than the equal sign like **>, <, >=, <= .**It's opposite of inner joints. This type of joint is rarely used since joint columns are keys & inequality comparison of the keys has no meaning full applications.

```
SELECT firstname||''lastname "full name"
FROM customer,order
WHERE customer.customer_id<order.customer_id;
```

SET OPERATOR

Sometimes you need to combine information of similar type from one/more tables. For example if you want to combine the results of to queries, fetch the common returned by two queries or to get a subset from the mainset we use set operators. Dataypes of corresponding columns must be the same.

Set operators are:

- Intersect
- Union
- Union all
- Minus

5.11 INTERSECT

The intersect operator returns the row that are common between two sets of rows.

Example:

```
SELECT DEPTNO FROM DEPT INTESECT SELECT DEPTNO FROM EMP;
```

Output:

```
DEPTNO
   10
   20
   30
   40
```

5.12 UNION

The union clause merges the outputs of two or more queries into a single set of rows and columns.

Example:

```
SELECT DEPTNO FROM DEPT UNION SELECT DEPTNO FROM EMP;
```

Output:

```
DEPTNO
  10
  20
  30
  40
```

5.13 UNION ALL

The union ALL clause repeat the outputs of two or more queries into a single set of rows and columns.

Example:

```
SELECT DEPTNO FROM DEPT UNION ALL SELECT DEPTNO FROM EMP;
```

Output:

```
DEPTNO
   10
```

```
20
30
40
20
30
10
20
40
10
20
```

5.14 MINUS

The minus operator returns the rows unique to first query.

Example:

```
SELECT DEPTNO FROM DEPT MINUS SELECT DEPTNO FROM EMP;
```

Output:

```
DEPTNO
  40
```

5.15 SQL Index

Index in sql is created on existing tables to retrieve the rows quickly.

When there are thousands of records in a table, retrieving information will take a long time. Therefore indexes are created on columns which are accessed frequently, so that the information can be retrieved quickly. Indexes can be created on a single column or a group of columns. When a index is created, it first sorts the data and then it assigns a ROWID for each row.

Syntax to create Index:

```
CREATE INDEX index_name  ON table_name (column_name1,column_name2...);
```

Syntax to create SQL unique Index:

```
CREATE UNIQUE INDEX index_name ON table_name (column_name1,column_name2...);
```

- index_name is the name of the INDEX.
- table_name is the name of the table to which the indexed column belongs.
- column_name1, column_name2.. is the list of columns which make up the INDEX.
- In Oracle there are two types of SQL index namely, implicit and explicit.

Implicit Indexes:

They are created when a column is explicity defined with PRIMARY KEY, UNIQUE KEY Constraint.

Explicit Indexes:

They are created using the "create index.. " syntax.

NOTE:

1. Even though sql indexes are created to access the rows in the table quickly, they slow down DML operations like INSERT, UPDATE, DELETE on the table, because the indexes and tables both are updated along when a DML operation is performed. So use indexes only on columns which are used to search the table frequently.
2. Is is not required to create indexes on table which have less data.
3. In oracle database you can define up to sixteen (16) columns in an INDEX.

DCL commands are used to enforce database security in a multiple user database environment. Two types of DCL commands are GRANT and REVOTE. Only Database Administrator's or owner's of the database object can provide/remove privileges on a databse object.

Rebuilding and Index
Oracle provides a fast index rebuild capability that allows you to recreate an index without having to drop the existing index.

Syntax:ALTER INDEX<INDEX-NAME>REBUILD;

Dropping an Index:
You can drop an index using a DOP INDEX command.

Syntax:
DROP INDEX<INDEX-NAME>;
If the index name is ENAME_INDEX then
Example:
DROP INDEX ENAME_INDEX;
Note: Whenever you drop an index the associated table's data is un-affected.

5.16 CLUSTERS:

Apart from using indexes for faster retrieval, CLUSTER also improves performance for faster retrieval of data from the table.
Creating of Cluster
Cluster can be created using CREATE CLUSTER command.

Syntax:
CREATE CLUSTER<CLUSTER-NAME> (COMMON-COLUMN-NAME DATATYPE);

Now we shall try the whole example using DEPT and EMP table.

Example:
CREATE CLUSTER DEPT_EMP_CLUSTER (DEPTNO NUMBER (2));
Creating Clustered Tables
Note: Only new tables can be created in a cluster. Existing tables cannot be moved into the cluster.

Example for creating DEPT table with cluster:
CREATE TABLE DEPT(DEPTNO NUMBER(2) PRIMARY KETY, DNAME VARCHAR2(20), LOC VARCHAR2(20))CLUSTER DEPT_EMP_CLUSTER(DEPTNO);

Example for creating EMP table with cluster:
CREATE TABLE EMP(EMPNO NUMBER(4) PRIMARY KEY, ENAME VARCHAR2(20) NOT NULL, JOB VARCHAR2(20), DEPTNO NUMBER(2), SAL NUMBER(5)) CLUSTER DEPT_EMP_CLUESTER(DEPTNO);

Creating Cluster Indexes
In a clustered index, the actual data is sorted in the indexed order.
Syntax:
CREATE INDEX <INDED-NAME> ON CLUSETER <CLUSTER –NAME>;

Example: **CREATE INDEX DEPT_EMP_INDEX ON CLUSTER DEPT_EMP_CLUSTER;**

Dropping a Cluster:
DROP CLUSTER command is used to remove a cluster from the data dictionary.

Syntax:
DROP CLUSTER <CLUSTER-NAME>;

Example:
DROP CLUSTER DEPT_EMP_CLUSTER;
To drop a cluster which contains one or more clustered table, then first the member tables should be dropped or INCLUDING TABLES option can be included in the DROP CLUSTER command. The following example illustrated this:
Syntax: DROP CLUSTER<CLUSTER-NAME> INCLUDING TABLES;

Example: DROP CLUSTER DEPT_EMP_CLUSTER INCLUDING TABLES;

5.17 SQL VIEWS

A view is a custom-tailored presentation of data form one or more tables. A view is nothing but a stored query. View is like a window through which data on a table can be viewed or changed. It is a virtual table that is – does not have any data of its own, but derives the data from the table it is associated with. It manipulates data in the underlying base table.

Restrictions for creating view

You cannot include currval or nextval pseudo columns in the view query.
If you include rowed, Rownum pseudo columns in the view's query, they must have aliases in the view's query.
DML operations cannot be issued against a view whose underlying query contains:

- **Group/aggregate function**
- **Group by or having clause**
- **Sub-quires**
- **Set operators**
- **The distinct operator**

The Syntax to create a sql view is

CREATE VIEW view name AS SELECT column_list FROM table_name
[WHERE condition];

view name is the name of the VIEW.
The SELECT statement is used to define the columns and rows that you want to display in the view.

For Example: to create a view on the product table the sql query would be like
CREATE VIEW view_product AS SELECT product_id, product_name FROM product;
Advantage of view

1. It provides additional level of security to the table y restrictiong access to a predefined set roles and columns of a table. For instance, in the view definition the columns with sensitive information may be excluding from the view definition.
2. It hides data complexity. For instance a view cab be created from multiple table using joins. However, the view hides the fact that data originates from different tables.
3. As a view does not store an data the redundancy problem does not arise.

4. View allows users to make simple queries to retrieve the3 result forms complicated queries. For example: views allow users to select information from multiple tables without knowing how to perform JOIN.
5. View stores complex queries.

5.18 SEQUENCE:

A Sequence is a database object used to generate unique integers for use as primary keys. A sequence is created through the CREATE SEQUENCE command. The Pseudo-column NEXTVAL is used to extract successive sequence numbers from a specified sequence. The Pseudo-column CURRVAL is used to refer to a sequence number that is the current sequence number. Sequences can be altered and dropped.

Creating the Sequence

Syntax:

```
CREATE SEQUENCE <SEQUENCE NAME>
INCREMENT BY <N>]
START WITH <M>]
MAXVALUE N | NOMAXVALUE]
MINVALUE N | NOMINVALUE]
CYCLE/NOCYCLE
CACHE/NOCACHE
```

All clauses are optional.

<SEQUENCE NAME> is the name of the sequence. <N> is the increment specified by the user. The default for INCREMENT BY is 1; the user can specify the increment. START WITH is the number with which the sequence will begin. MINVALUE sequence will generate lower bound. Default is 1 for ascending sequences and 10e27-1 for descending sequences. MAXVALUE provides upper bound for sequence. Default is 1 for descending sequence and 10e27-1 for ascending sequences. Any attempts to generate a sequence number once the upper limit is reached returns an error.

Example:

```
CREATE SEQUENCE EMP_NUMBER INCREMENT BY 1 START WITH 1000;
```

This will create a sequence EMP_NUMBER that can be accessed by INSERT and UPDATE.

5.19 SYNONYM:

A synonym is an alias name for a table, view etc., since a synonym is nothing but an alias it requires no storage other than its definition. Synonym are often used for security and convenience, because they can
Mask the name and owner of the object.
Provide location transparency for remote object of a distributed database.

Creating the Synonym

Syntax: CREATE SYNONYM <SYNONYM-NAME> FOR <OBJECT>;

Example: CREATE SYNONYM EMP_SYS FOR EMP;

The above example will create a synonym called EMP_SYS for EMP table.

Object name may be a table or view or cluster or a synonym itself.

Dropping a synonym:
Like Views, Synonyms can dropped as and when required. The following example illustrates this.

Syntax: DROP SYNONYM <SYNONYM_NAME>;

Where SYNONYM_NAME should be an existing Synonym.

Example: DROP SYNONYM EXP_SYS;
The above statement drops the Synonym from the database.

5.20 ROLES:

Roles are a collection of privileges or access rights. When there are many users in a database it becomes difficult to grant or revoke privileges to users. Therefore, if you define roles, you can grant or revoke privileges to users, thereby automatically granting or revoking privileges. You can either create Roles or use the system roles pre-defined by oracle.
Some of the privileges granted to the system roles are as given below:

System Role	Privileges Granted to the Role
CONNECT	CREATE TABLE, CREATE VIEW, CREATE SYNONYM, CREATE SEQUENCE, CREATE SESSION etc.
RESOURCE	CREATE PROCEDURE, CREATE SEQUENCE, CREATE TABLE, CREATE TRIGGER etc. The primary usage of the RESOURCE role is to restrict access to database objects.
DBA	ALL SYSTEM PRIVILEGES

Creating Roles:
The Syntax to create a role is:

CREATE ROLE role_name
[IDENTIFIED BY password];

For example: To create a role called "developer" with password as "pwd",the code will be as follows

CREATE ROLE testing
[IDENTIFIED BY pwd];

It's easier to GRANT or REVOKE privileges to the users through a role rather than assigning a privilege directly to every user. If a role is identified by a password, then, when you GRANT or REVOKE privileges to the role, you definitely have to identify it with the password.
We can GRANT or REVOKE privilege to a role as below.
For example: To grant CREATE TABLE privilege to a user by creating a testing role:
First, create a testing Role

CREATE ROLE testing;

Second, grant a CREATE TABLE privilege to the ROLE testing. You can add more privileges to the ROLE.

GRANT CREATE TABLE TO testing;

Third, grant the role to a user.

GRANT testing TO user1;

To revoke a CREATE TABLE privilege from testing ROLE, you can write:

REVOKE CREATE TABLE FROM testing;

The Syntax to drop a role from the database is as below:

DROP ROLE role_name;

For example: To drop a role called developer, you can write:

DROP ROLE testing;

5.21 PRIVILEGES

Privileges: Privileges defines the access rights provided to a user on a database object. There are two types of privileges.

1) **System privileges** - This allows the user to CREATE, ALTER, or DROP database objects.

2) **Object privileges** - This allows the user to EXECUTE, SELECT, INSERT, UPDATE, or DELETE data from database objects to which the privileges apply.

Few CREATE system privileges are listed below:

System Privileges	**Description**
CREATE object	allows users to create the specified object in their own schema.
CREATE ANY object	allows users to create the specified object in any schema.

The above rules also apply for ALTER and DROP system privileges.

Few of the object privileges are listed below:

Object Privileges	**Description**
INSERT	allows users to insert rows into a table.
SELECT	allows users to select data from a database object.
UPDATE	allows user to update data in a table.
EXECUTE	allows user to execute a stored procedure or a function.

5.22 SQL GRANT COMMAND

SQL GRANT is a command used to provide access or privileges on the database objects to the users.

The Syntax for the GRANT command is:

GRANT privilege_name
ON object_name
TO {user_name |PUBLIC |role_name}
[WITH GRANT OPTION];

privilege_name is the access right or privilege granted to the user. Some of the access rights are ALL, EXECUTE, and SELECT.

object_name is the name of an database object like TABLE, VIEW, STORED PROC and SEQUENCE.

user_name is the name of the user to whom an access right is being granted.

user_name is the name of the user to whom an access right is being granted.

PUBLIC is used to grant access rights to all users.

ROLES are a set of privileges grouped together.

WITH GRANT OPTION - allows a user to grant access rights to other users.

For Example: GRANT SELECT ON employee TO user1;This command grants a SELECT permission on employee table to user1.You should use the WITH GRANT option carefully because for example if you GRANT SELECT privilege on employee table to user1 using the WITH GRANT option, then user1 can GRANT SELECT privilege on employee table to another user, such as user2 etc. Later, if you REVOKE the SELECT privilege on employee from user1, still user2 will have SELECT privilege on employee table.

5.23 SQL REVOKE COMMAND:

The REVOKE command removes user access rights or privileges to the database objects.
The Syntax for the REVOKE command is:

REVOKE privilege_name
ON object_name
FROM {user_name |PUBLIC |role_name}

For Example: REVOKE SELECT ON employee FROM user1;This command will REVOKE a SELECT privilege on employee table from user1.When you REVOKE SELECT privilege on a table from a user, the user will not be able to SELECT data from that table anymore. However, if the user has received SELECT privileges on that table from more than one users, he/she can SELECT from that table until everyone who granted the permission revokes it. You cannot REVOKE privileges if they were not initially granted by you.
Privileges and Roles:
Privileges: Privileges defines the access rights provided to a user on a database object. There are two types of privileges.

1) System privileges - This allows the user to CREATE, ALTER, or DROP database objects. 2) Object privileges - This allows the user to EXECUTE, SELECT, INSERT, UPDATE, or DELETE data from database objects to which the privileges apply.

Few CREATE system privileges are listed below:

System Privileges	Description
CREATE object	allows users to create the specified object in their own schema.
CREATE ANY object	allows users to create the specified object in any schema.

The above rules also apply for ALTER and DROP system privileges.
Few of the object privileges are listed below:

Object Privileges	Description
INSERT	allows users to insert rows into a table.
SELECT	allows users to select data from a database object.
UPDATE	allows user to update data in a table.
EXECUTE	allows user to execute a stored procedure or a function.