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).

  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.

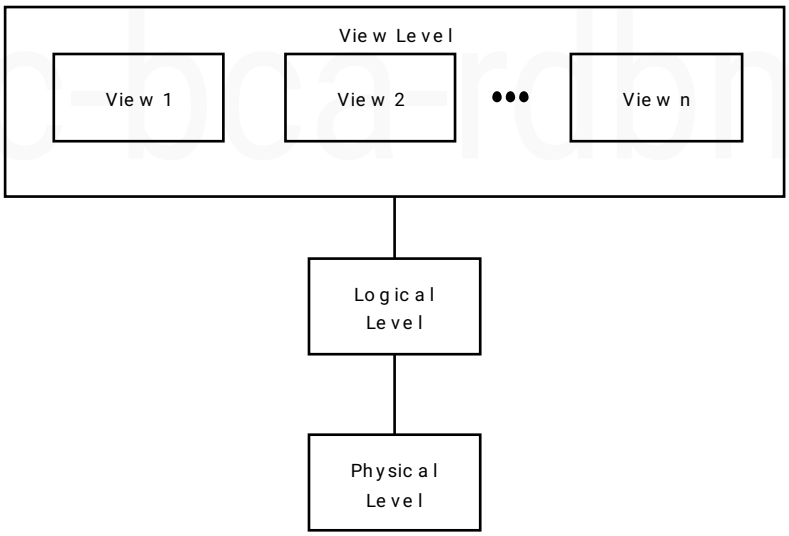
1.2 View of Data

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 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.

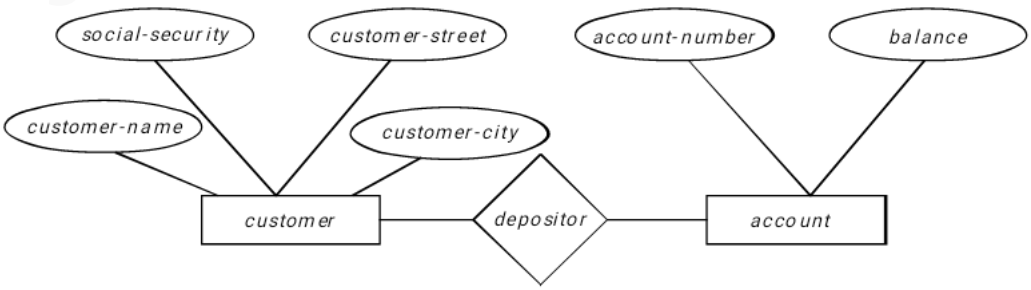
For example, the attributes *number* and *balance* describe one particular *account* in a bank.

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.

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

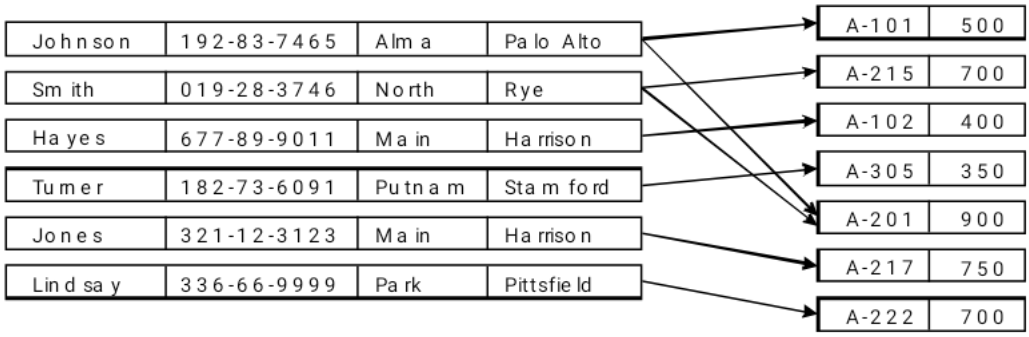
A sample Relational database

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.

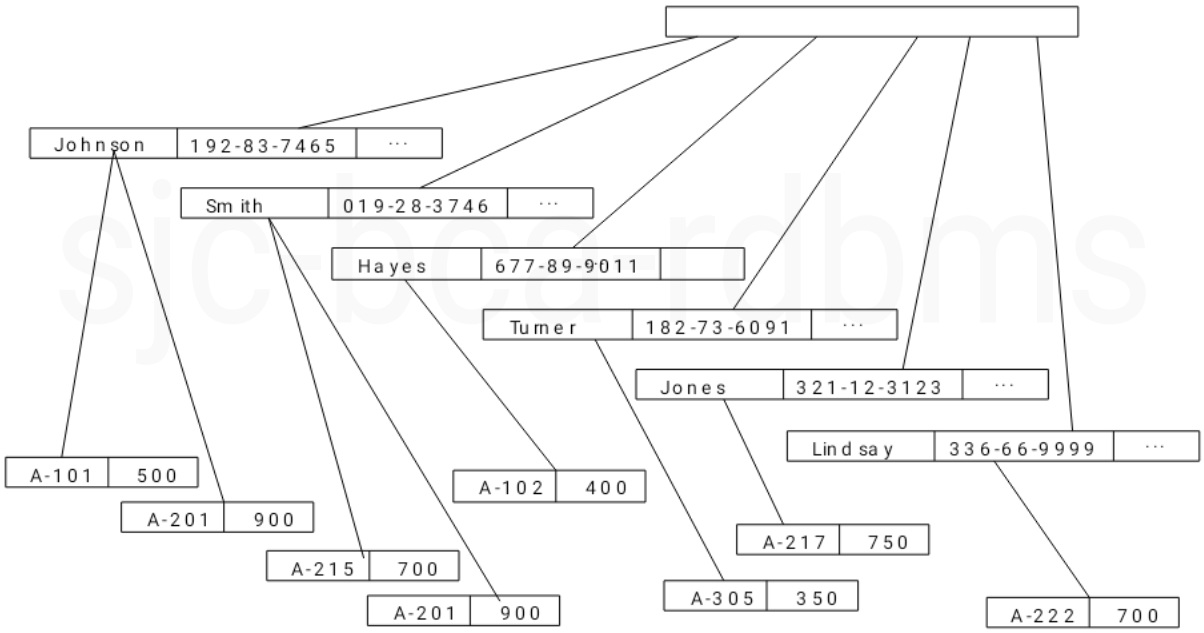


A sample Network database

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 type 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

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 *precompiler,* 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 are 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.

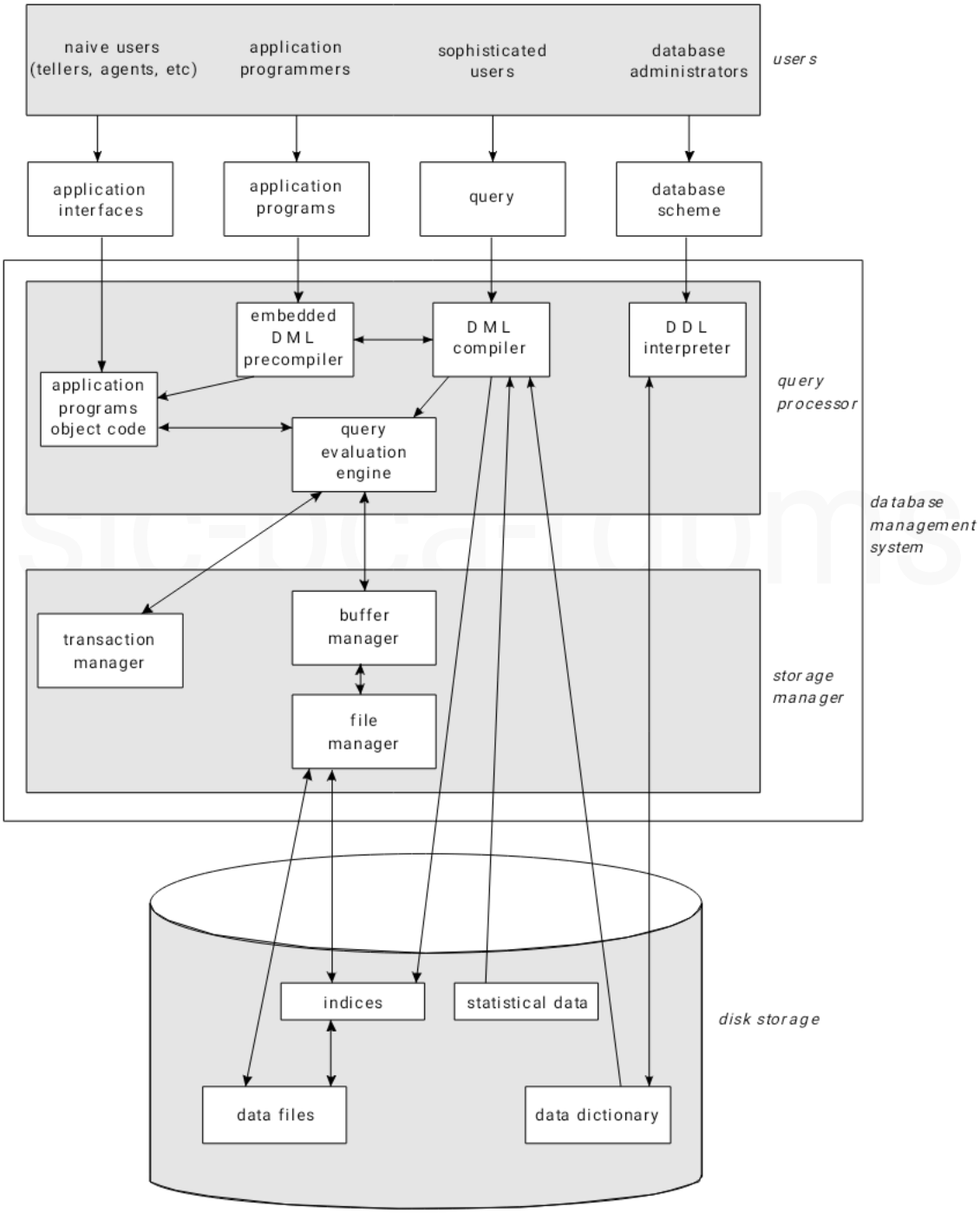
Several data structures are required as part of the physical system implementation:

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 *accountnumber* 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:

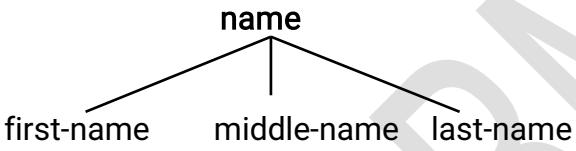
- **Simple** and **Composite** attributes, the simple attributes cannot be divided into subparts, whereas in the case of composite of attributes, it can be divided into subparts.

  Ex:

  Simple attribute:

  The *account-number* attribute refers to only one account number.

  Composite attribute:



- **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.

  Ex: age,given date_of_ birth.

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

$$\{(e1, e2\ e3 \ldots en)\ e1 \varepsilon\ E1, e2\ \varepsilon\ E2, \ldots, en\ \varepsilon\ En\}$$

where $(e1, e2, \ldots, en)$ 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.

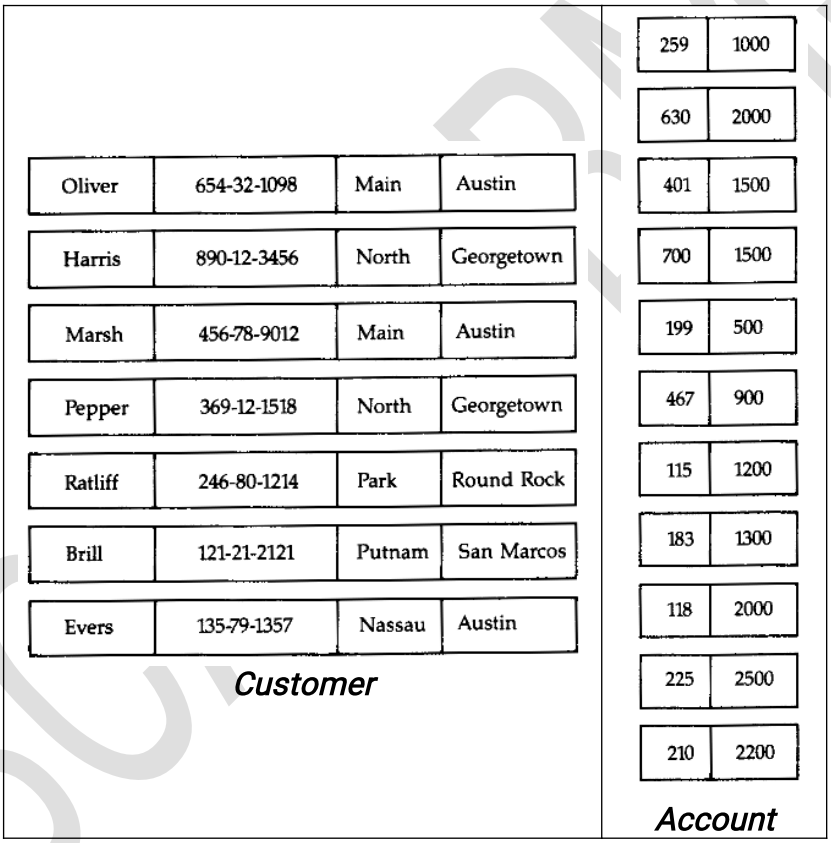


Figure 1 Entity sets Customer and Account

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***.

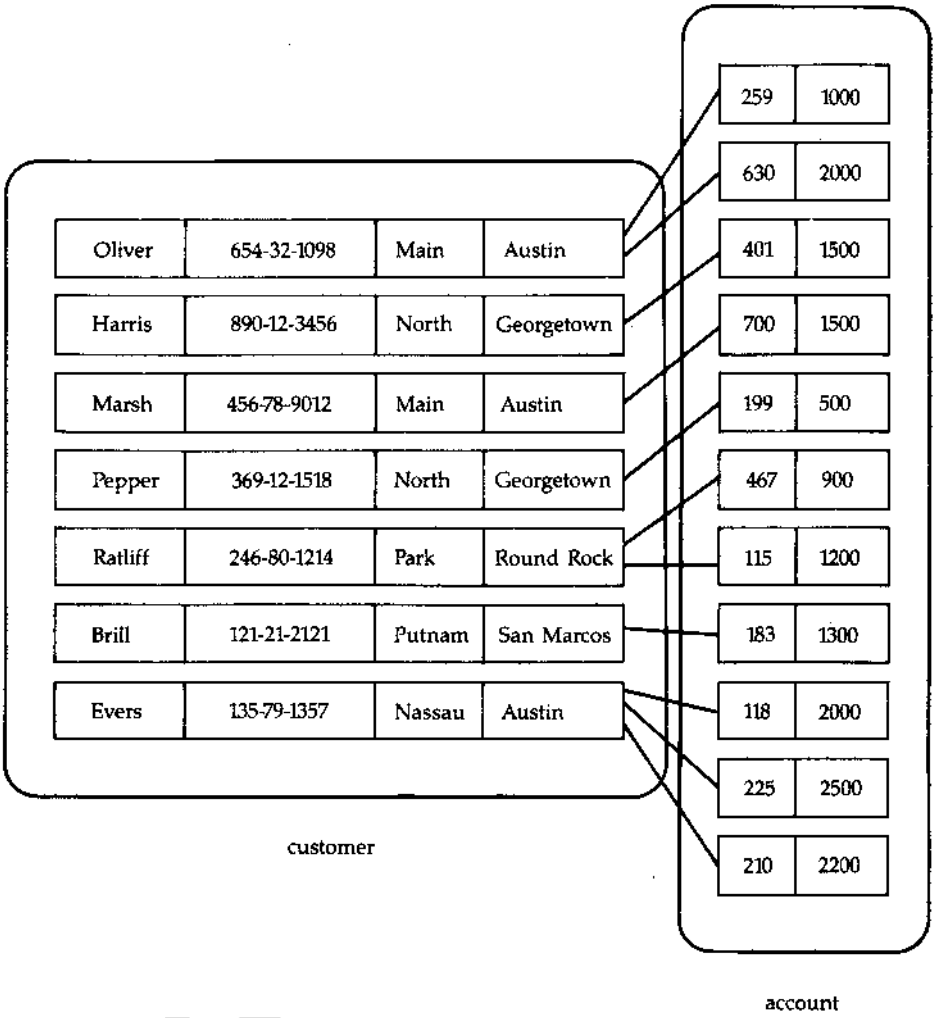


Figure 2 Relationship between Entity sets Customer and Account

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.

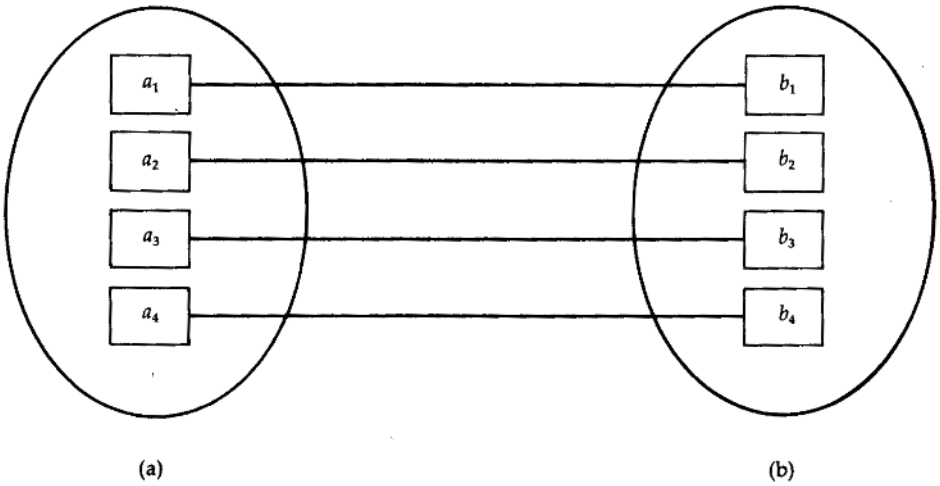
Mapping Cardinalities

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.

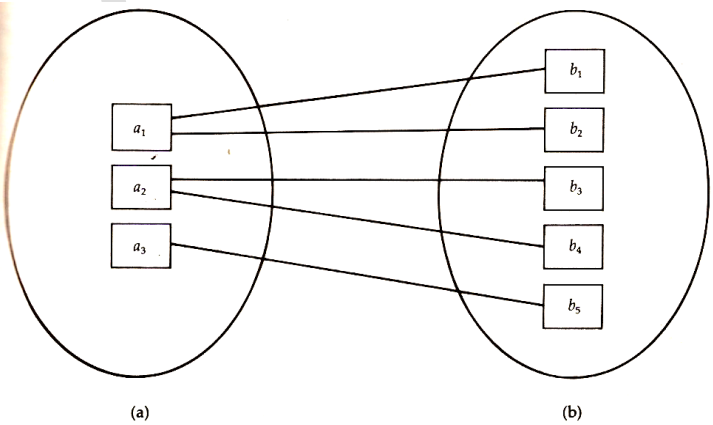
For a binary relationship set *R* between entity sets *A* and *B,* the mapping cardinality must be one of the following:

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.*



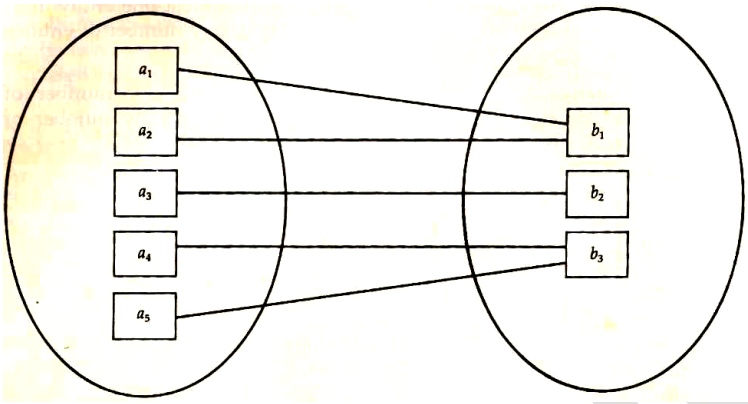
One-to-one relationship

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.*



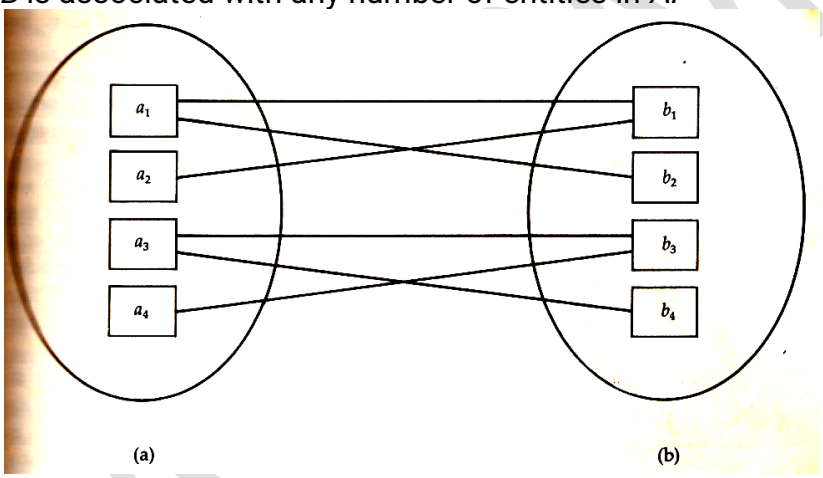
One-to-many relationship

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.*



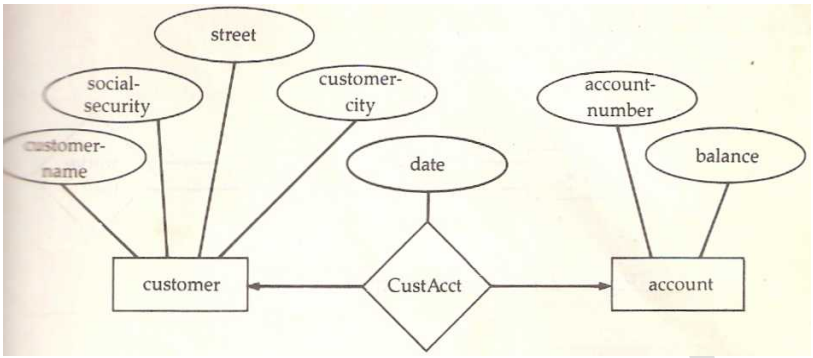
Many-to-many relationship

Consider the entity-relationship diagram in below Figure, 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.*

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

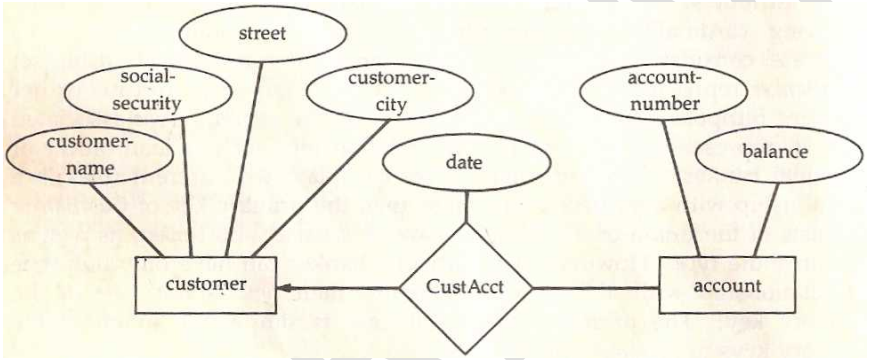
A directed line from the relationship set *CustAcct* to the entity set *account* specifies that the *account* entity set participates in either a one-to-one or a many-to-one relationship with the *customer* entity set.

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.



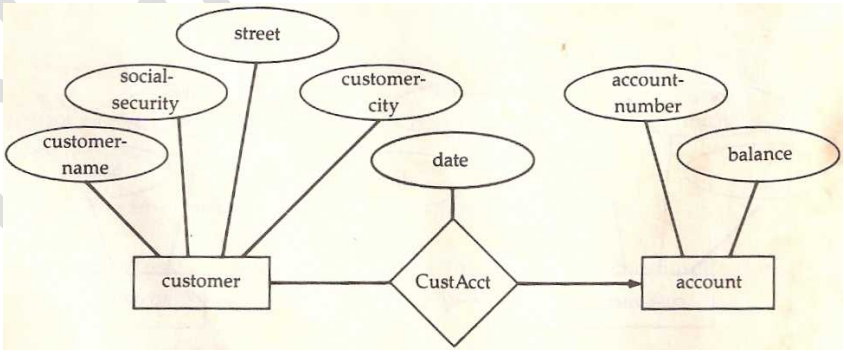
1. One-to-one relationship

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 1).



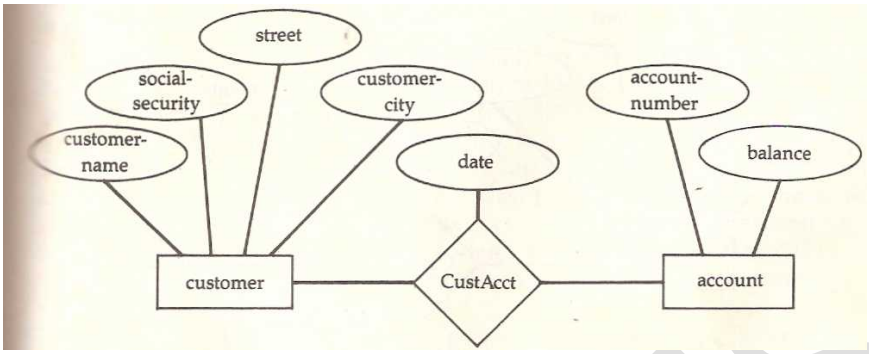
2. One-to-many relationship

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 2).



3. Many-to-one relationship

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 3).



4. Many-to-many relationship

Finally, the E-R diagram of Figure 4, we see that the relationship *CustAcct* is many-to-many.

Keys

Key plays an important role in relational database; it is used to distinguish entities from each other. Keys also help uniquely identify relationships, and thus distinguish relationships from each other.

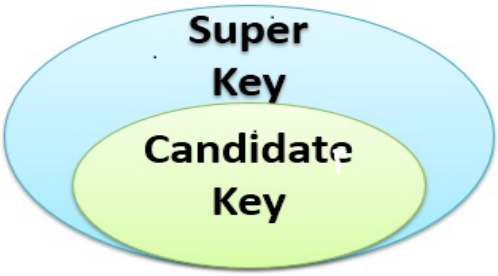
Super key

Super key is a set of one or more attributes; it allows us to identify uniquely an entity in the entity set.

Ex: the customer_id attribute of the entity set customer is used to distinguish one customer entity from another. Thus customer-id is the super key. Similarly the combination of customer_name and customer_id is a super key for the entity set customer.The customer_name attribute is not a super key, because several people might have the same name.

Candidate key

The minimal super keys are called candidate keys. Super keys which are having no proper subset is called candidate keys. It is also is used to identify uniquely an entity in entity set. All candidate keys are super keys, but all super keys need not to be a candidate keys.

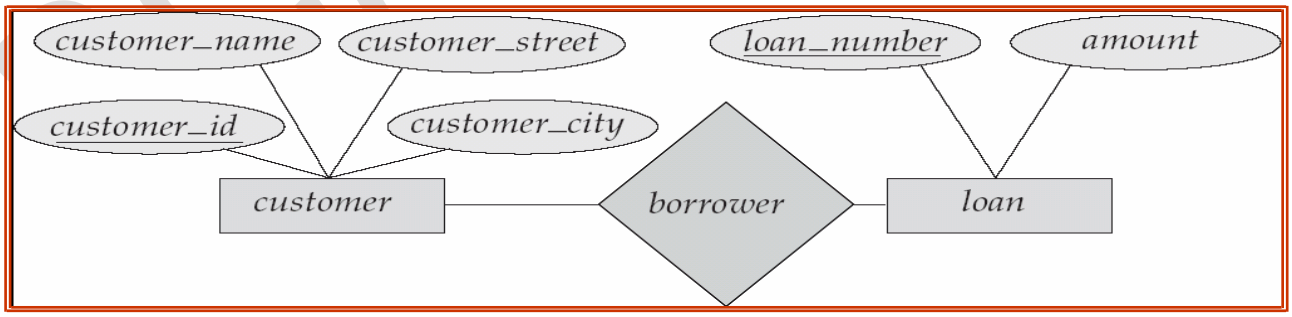


Ex: Combination of customer_name and customer_address is sufficient to distinguish among members from the customer entity set. Then both { customer_id} and { customer_name ,customer_address} are candidate keys. Although the attributes customer_id and customer_name together can distinguish customer entities, their combination does not form a candidate key, since the attribute customer_id alone is a candidate key.

The term primary key is used to denote a candidate key that is chosen by the database designer as the principal means of identifying entities within an entity set. A key (primary, candidate and super) is a property of the entity set.

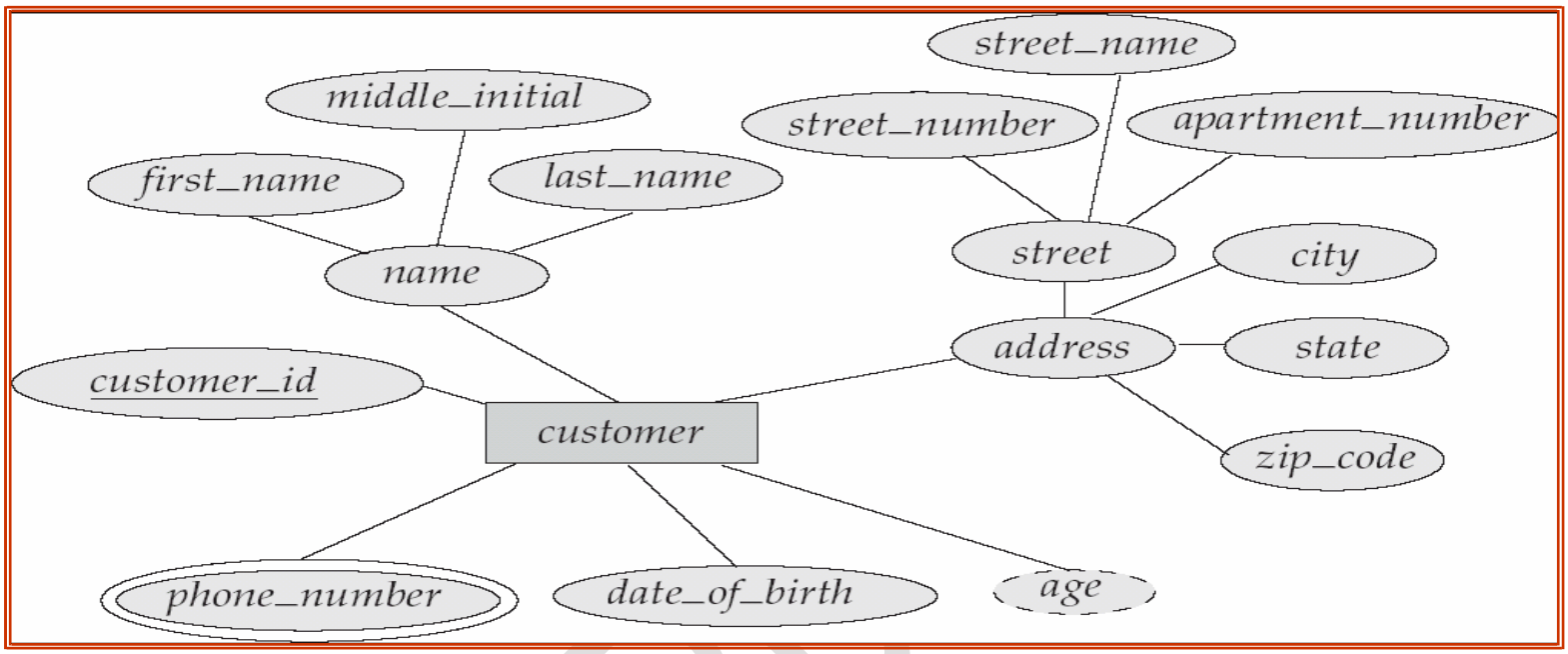
Entity-Relationship Diagram

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

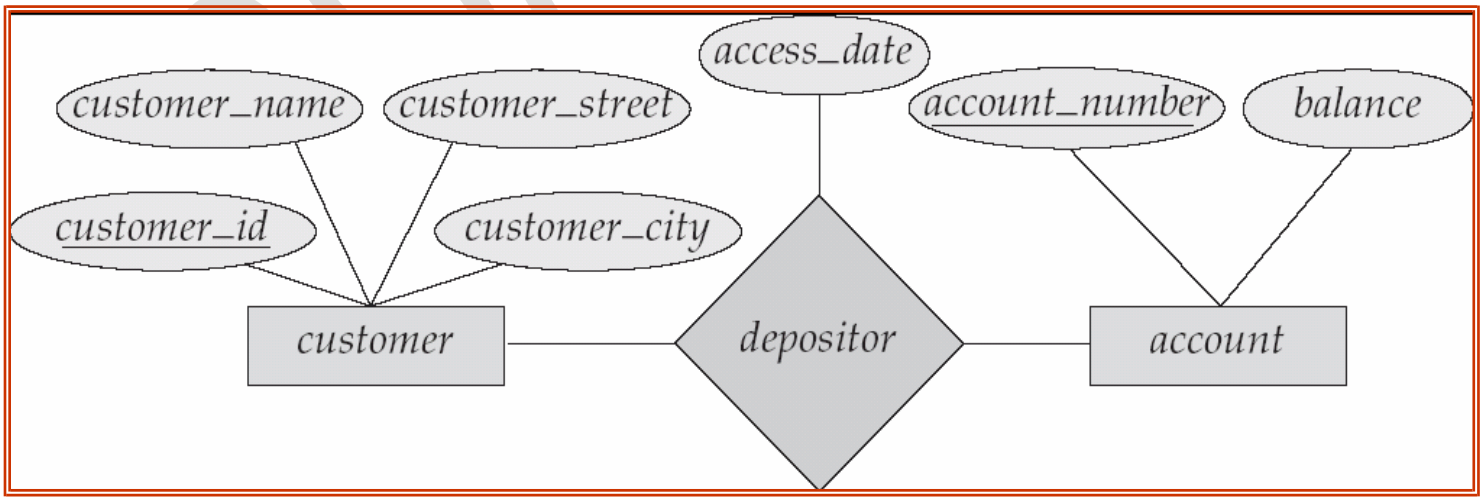


- **Rectangles,** which represent entity sets.
- **Diamonds,** which represent relationship sets
- **Lines,** which link attributes to entity sets and entity sets to relationship sets.
- **Ellipses,** which represent attributes.

Double ellipses represent multivalued attributes.
Dashed ellipses denote derived attributes.
Underline indicates primary key attributes



E-R Diagram with Composite, Multivalued, and Derived Attributes



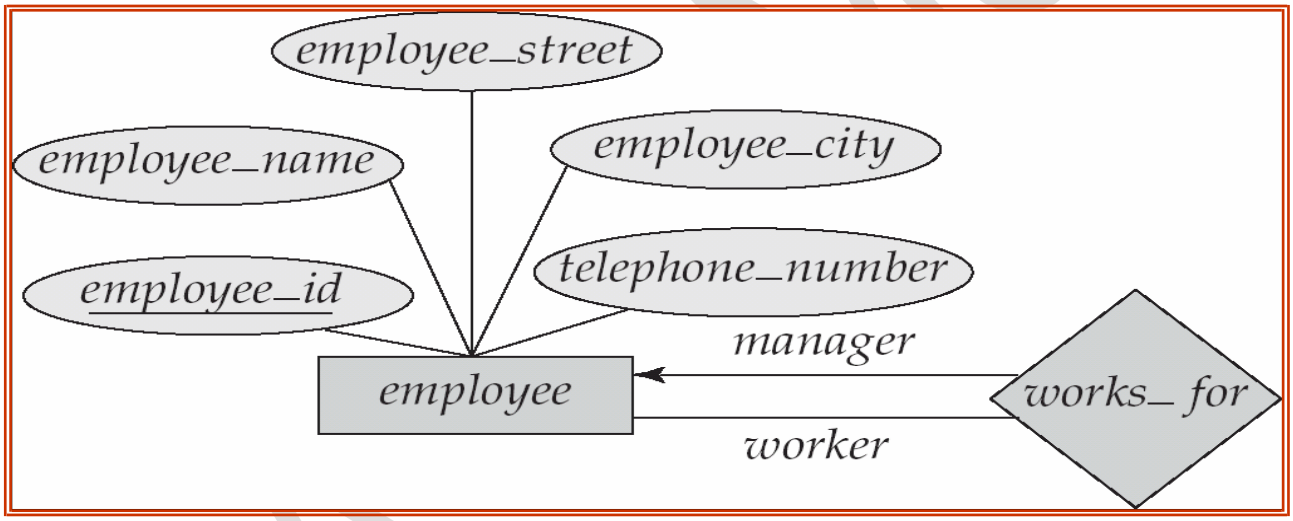
Relationship Sets with Attributes

Roles

The labels “manager” and “worker” are called **roles**; they specifyhow employee entities interact via the works_for relationship set.

Roles are indicated in E-R diagrams by labeling the lines that connect diamonds to rectangles.

Role labels are optional, and are and are used to clarify semantics of the relationship.



Participation of an Entity Set in a Relationship Set

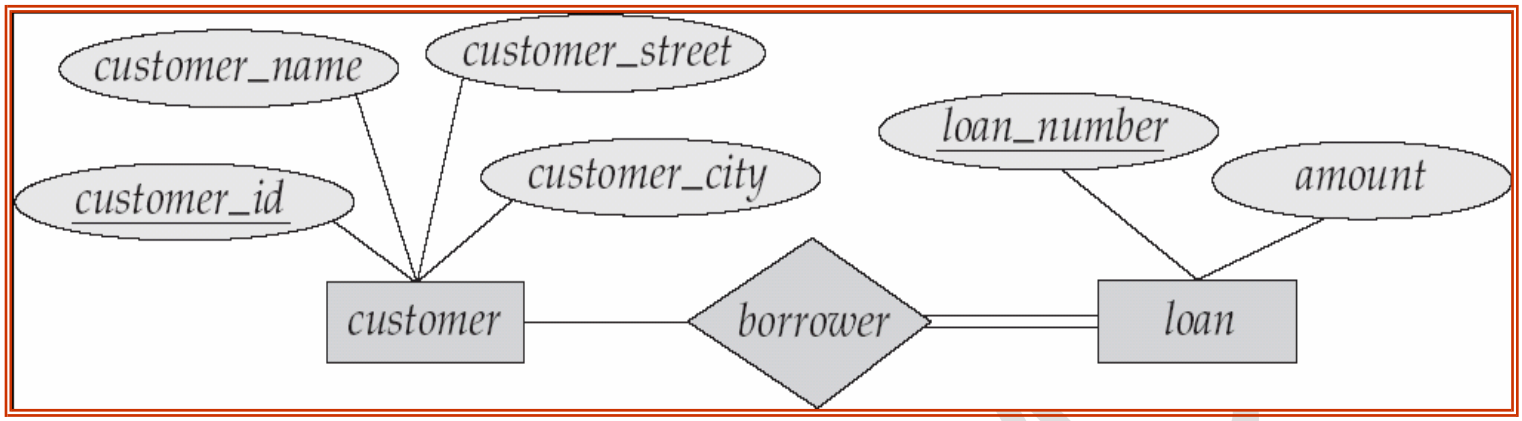
Total participation (indicated by double line): every entity in the entity set participates in at least one relationship in the relationship set

E.g. participation of loan in borrower is total

Every loan must have a customer associated to it via borrower

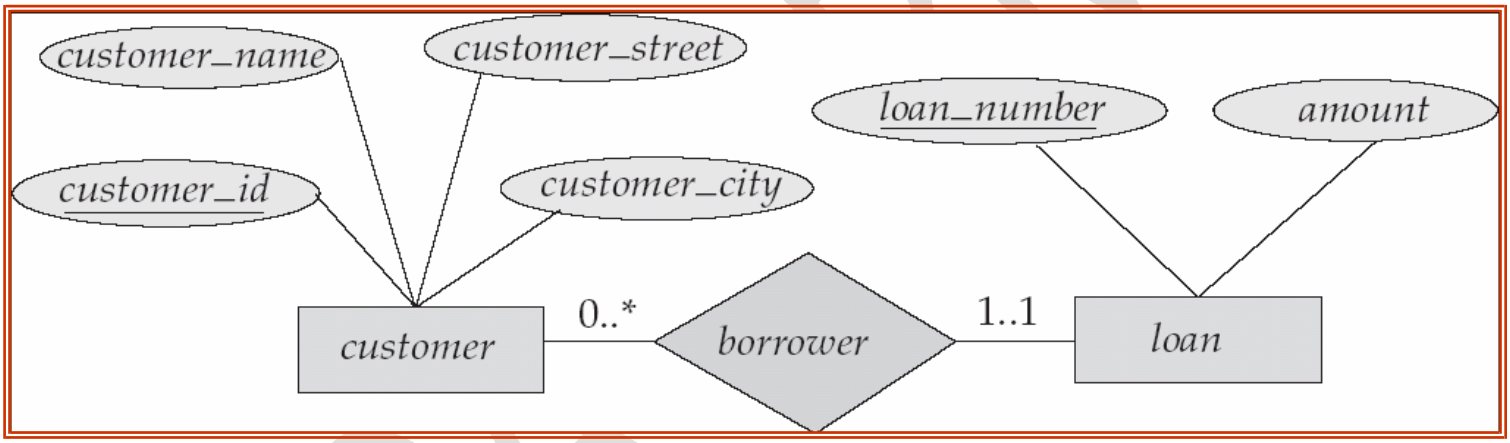
Partial participation: some entities may not participate in any relationship in the relationship set

Example: participation of customer in borrower is partial

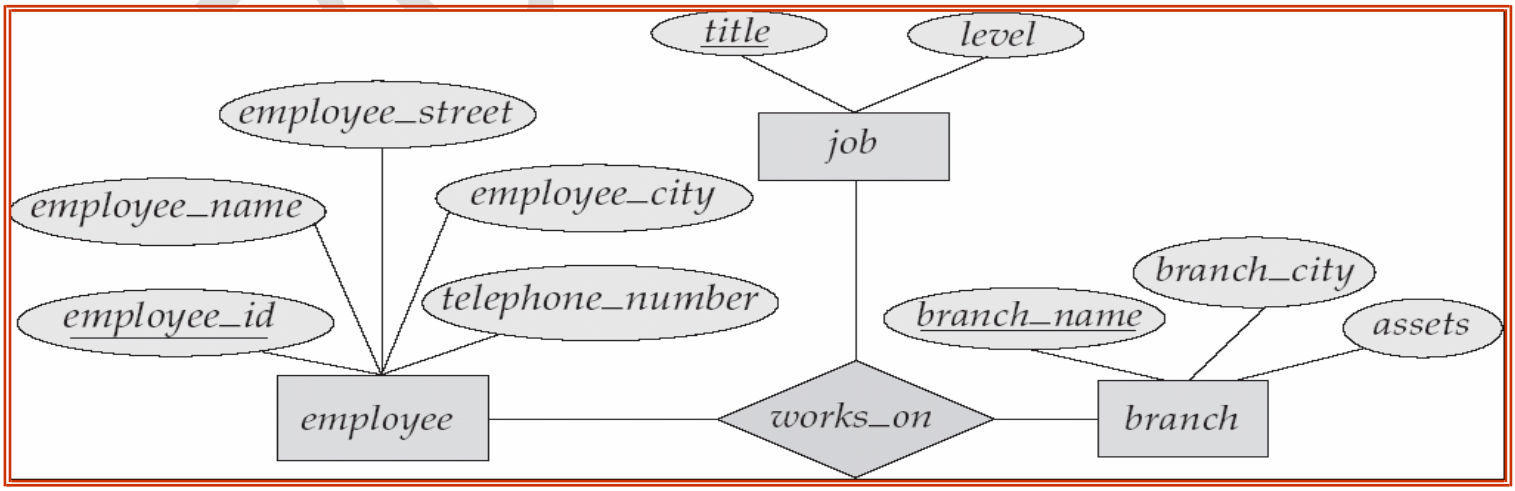


Alternative Notation for Cardinality Limits

Cardinality limits can also express participation constraints



E-R Diagram with a Ternary Relationship



Weak Entity Sets

An entity set that does not have a primary key is referred to as a**weak entity set**.

The existence of a weak entity set depends on the existence of a identifying entity set.

The **discriminator** (*or partial key)* of a weak entity set is the set of attributes that distinguishes among all the entities of a weak entityset.

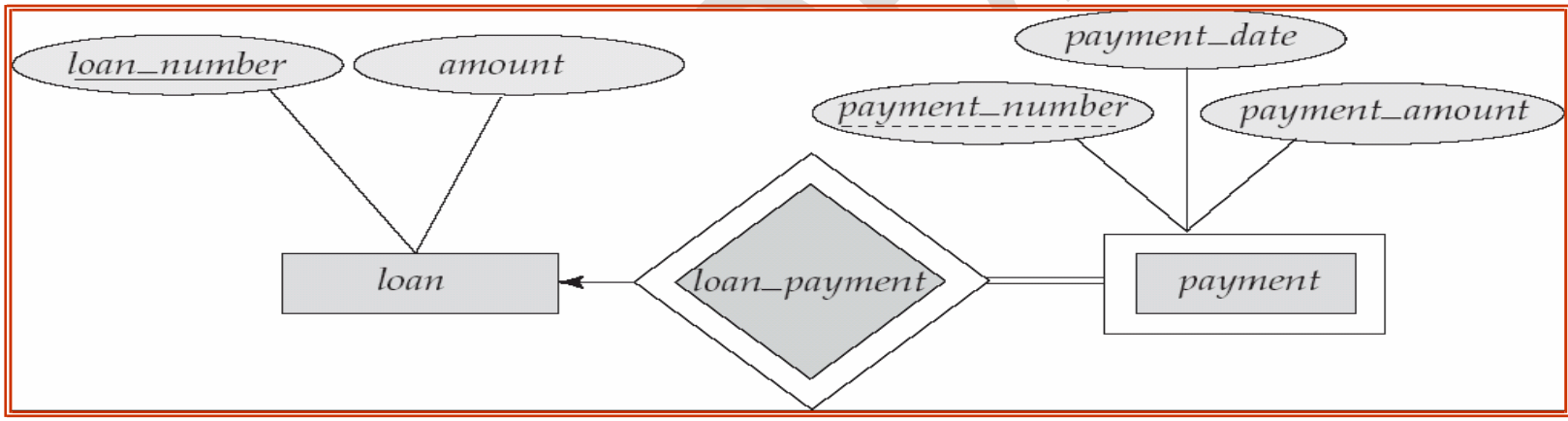
The primary key of a weak entity set is formed by the primary key of the strong entity set on which the weak entity set is existence dependent, plus the weak entity set's discriminator.

We depict a weak entity set by double rectangles.

We underline the discriminator of a weak entity set with a dashed line.

Payment_number – discriminator of the *payment* entity set

Primary key for *payment* – (*loan_number, payment_number*)



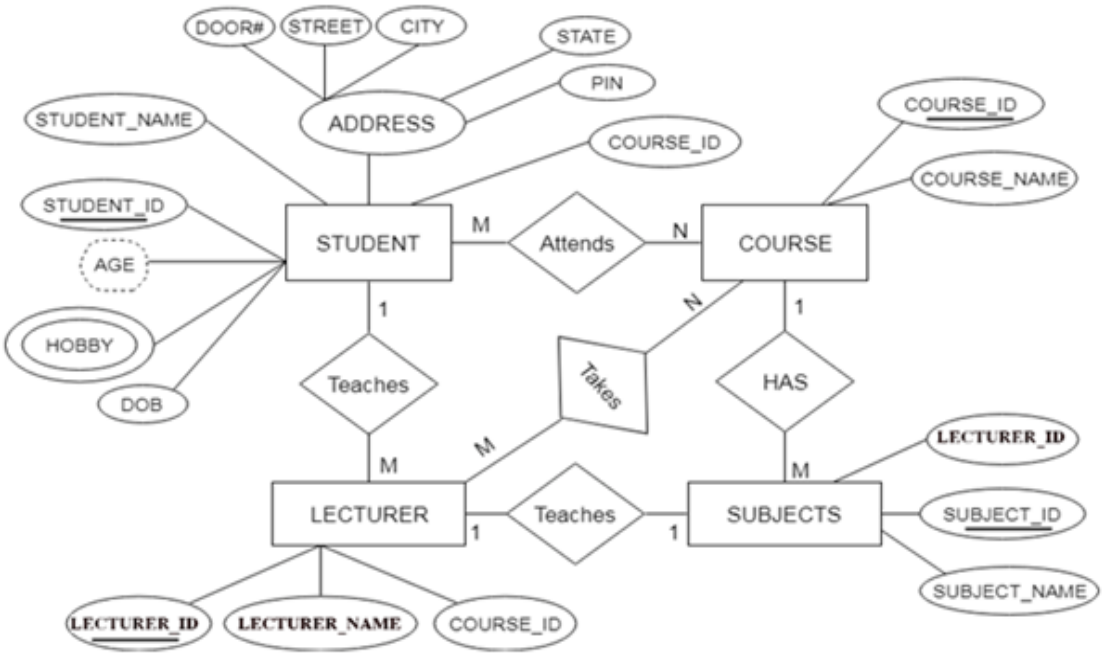
Reducing E-R diagram to Tables

Database can be represented using the notations and these notations can be reduced to collection of tables.

In the database, every entity set or relationship set can be represented in tabular form.

The ER diagram is given below:

v



There are some points for converting the ER diagram into table:

Entity type becomes a table.

In the given ER diagram, LECTURE, STUDENT, SUBJECT and COURSE forms individual tables.

All single valued attribute becomes a column for the table.

In the STUDENT entity, STUDENT_NAME and STUDENT_ID form the column of STUDENT table. Similarly COURSE_NAME and COURSE_ID form the column of COURSE table and so on.

Key attribute of the entity type represented by the primary key.

In the given ER diagram, COURSE_ID, STUDENT_ID, SUBJECT_ID and LECTURE_ID are the key attribute of the entity.

Multivalued attribute are represented by separate table.

In the student table, a hobby is a multivalued attribute. So it is not possible to represent multiple values in a single column of STUDENT table. Hence we create a table STUD_HOBBY with column name STUDENT_ID and HOBBY. Using both the column, we create a composite key.

Composite attribute represented by components.

In the given ER diagram, student address is a composite attribute. It contains CITY, PIN, DOOR#, STREET and STATE. In the STUDENT table, these attributes can merge as individual column.

Derived attributes are not considered in the table.

In the STUDENT table, Age is the derived attribute. It can be calculated at any point of time by calculating the difference between current date and Date of Birth.

Using these rules, you can covert ER diagram into tables and columns and assign the mapping between the tables. Table structure for the given ER diagram is as below:

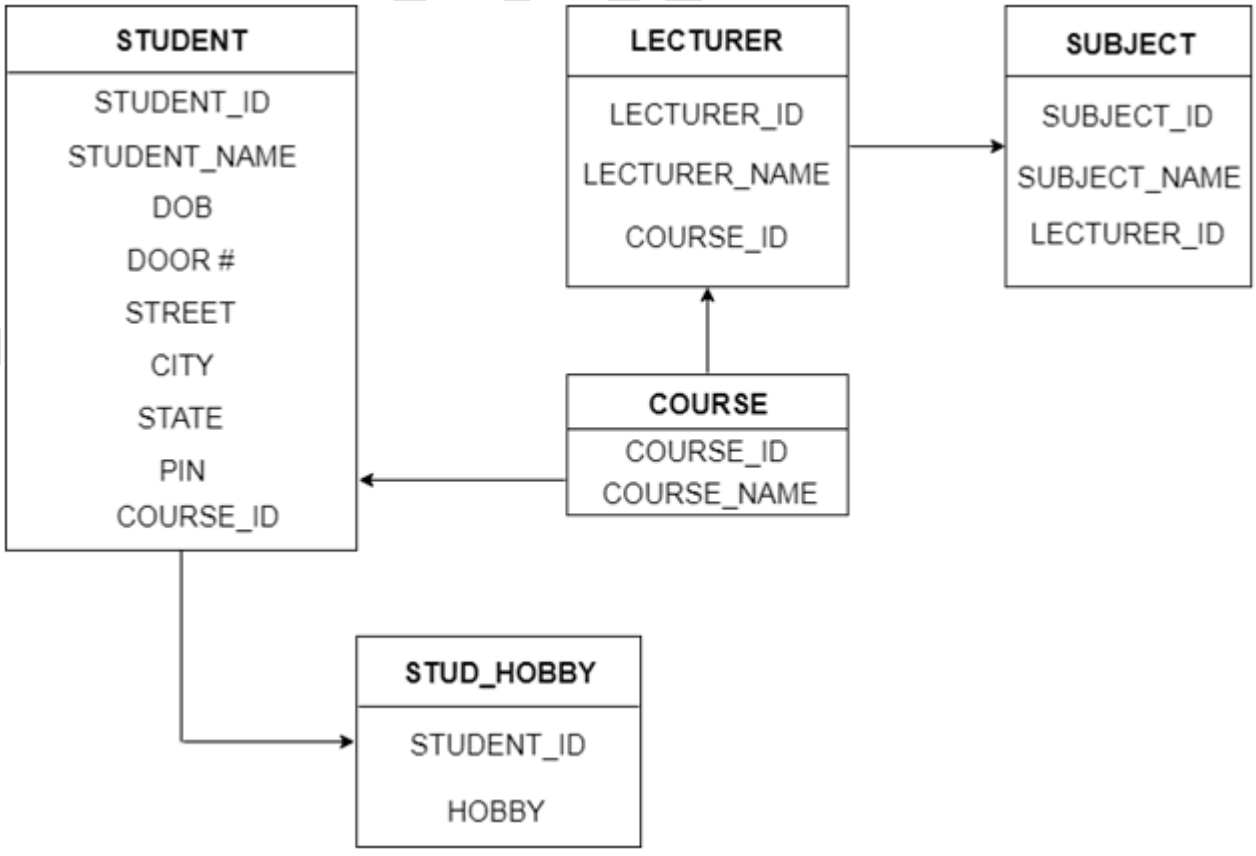


Figure: Table structure

UNIT-3
Normalization

Data Normalization is the process of organizing the data in the database. Normalization is a systematic approach of decomposing tables to eliminate data redundancy and undesirable characteristics like Insertion, Update, Deletion anomalies

It is a multi-step process that puts data into tabular form by removing duplicated data from the relation tables.

Problem without Normalization

Ex:

Consider a table Library_info for the Library Management which contains all the columns in a single table(raw format).

Library_info(Transaction_no,Lent_date,Customer_id,Customer_name,contact_no,address,Book_id,Copy_no,Title,Price)

Library_info

Trans_no	Lent date	Cid	C_name	C_ no	address	Book_id	Copy no	Title	Price
1	1/3/14	101	Raj	23478	pondy	3312	7	Rdbms	300
1	1/3/14	101	Raj	43679	pondy	4127	2	Network	250
2	4/5/14	105	Kumar	34567	cuddalore	5555	1	OS	300
3	7/8/14	113	Joy	22335	panruti	0022	4	Digital	350
4	9/9/14	109	Sam	17890	cuddalore	5555	3	OS	250
4	9/9/14	109	Sam	17890	cuddalore	3312	5	Rdbms	300

- ✔ Duplication of Data –The same data is listed in multiple lines of the database. (Customer name, address and contact no should be repeated for every transaction)
- ✔ Insert Anomaly – A record about an entity cannot be inserted into the table without first inserting information about another entity. (Cannot enter a customer

information without entering the book details)

- ✔ Delete Anomaly – A record cannot be deleted without deleting a record about a related entity. (Cannot delete a book details without deleting all of the customer's information)

- ✔ Update Anomaly – Cannot update information without changing information in many places.(To update customer information, it must be updated for each book the customer has rented)

First Normal Form(I NF):(Avoid Redundancy)

- If the table does not have repeating groups then it is said to be in first normal form (I NF).
- The value should be atomic in the sense that it cannot be decomposed into smaller pieces.
- Separate the repeating groups into new tables.
- The new table must include the key column of the Original table.

Form A(Key1,simple columns,(group1,A,B,C),(group2,X,Y))

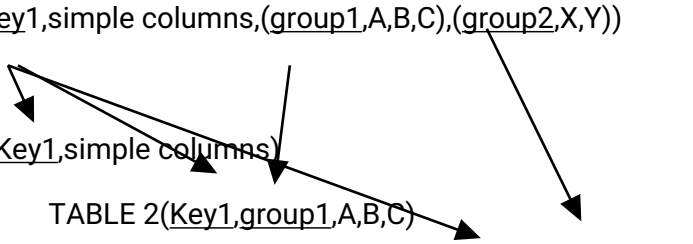


TABLE 1(Key1,simple columns)

TABLE 2(Key1,group1,A,B,C)

TABLE 3(Key1,group2,X,Y)

Ex:

Considering the table Library_info it contains repeating data. so the table should be decomposed based on repeating groups. Here Transaction no and book id is a primary key column ,by having this key column a new table can be created.

The new table is as follows:

Table_1(Transaction_no,Lent_date,Customer_id,Customer_name,contact_no,address)

Table_2(Transaction_no,Book_id,Copy_no,Title,Price)

Second Normal Form (2 NF): (Functional Dependency)

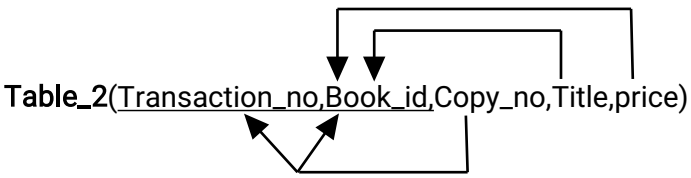
- Remove Partial Dependencies.
- ***Functional Dependency***: The value of one attribute in a table is determined entirely by the value of another.
- ***Partial Dependency*** A type of functional dependency where an attribute is functionally dependent on only part of the primary key (primary key must be a composite key).
- Create separate table with the functionally dependent data and the part of the key on which it depends.

Ex:

Problems existing in **Table_2**.

- ✔ Duplication of Data -The title of the book and rent has been repeated.
- ✔ Delete Anomaly-The Transaction details cannot be deleted without deleting the book details.
- ✔ Insert Anomaly -Cannot able to insert the Book details without Transaction .

Considering the Table_2, all the non key columns does not dependent on the entire key columns. Hence the table should be decomposed with the functionally dependent columns based on the key which it depends.



The new table is as follows:

Trans_books(Transaction_no,Book_id,Copy_no)

Book_info(Book_id,Title,price)

Third Normal Form(3 NF):(Transitive Dependency)

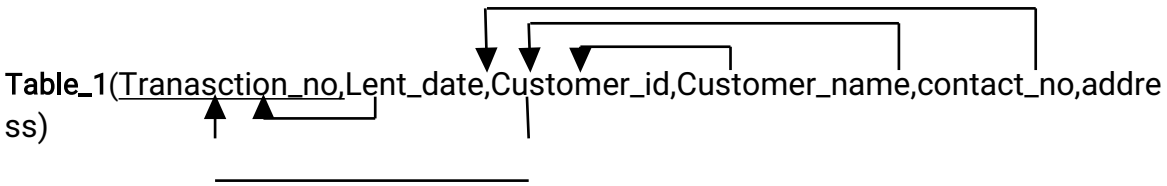
- Remove transitive dependencies.
- ***Transitive Dependency*** A type of functional dependency where an attribute is functionally dependent on an attribute other than the primary key. Thus its value is only indirectly determined by the primary key.
- Create a separate table containing the attribute and the fields that are functionally dependent on it.

Ex:

Problems existing in **Table_1**.

- ✔ Duplication of Data- The customer details has been repeated for each and every transaction.
- ✔ Insert Anomaly - Customer details cannot be inserted without inserting Transaction details.
- ✔ Delete Anomaly-The Transaction details cannot be deleted without deleting the Customer details
- ✔ Update Anomaly- To update customer information, it must be updated for each book the customer has rented

Considering the Table_1, the table is not in Third Normal Form since some of the columns dependent on the Customer id which is not a Primary key column. Hence the table should been decomposed dependent on an attribute other than the primary key column.



The new table is as follows:

Trans_cust(Tranasction_no,Lent_date,Customer_id)

Customer(Customer_id,Customer_name,contact_no,address)

After Completion of Three Normalization, the following tables have been created.

Trans_cust(Tranasction_no,Lent_date,Customer_id)

Customer(Customer_id,Customer_name,contact_no,address)

Trans_books(Transaction_no,Book_id,Copy_no)

Book_info(Book_id,Title,price)

BOYCE - CODD NORMAL FORM

To solve the additional problems in the table BCNF is defined.

BCNF is an extension of 3rd Normal Form (3NF). Since it has been termed 3.5NF. 3NF states that all data in a table must depend only on that table's primary key, and not on any other field in the table. At first glance it would seem that BCNF and 3NF is the same thing. However, in some rare cases it does happen that a 3NF table is not BCNFcompliant. This may happen in tables with two or more overlapping composite candidate keys.

A secondary relationship between columns within the table can cause problem with duplication and loss of data.

Ex:

Consider the table **SST** having three columns.

SST

Studentno	Major	Advisor
1	Os	Godbole
2	Rdbms	Abraham
3	Network	Martin
1	Rdbms	Abraham
4	C	Balagurusamy
2	C	Yeshwantkanithkar
5	Os	Godbole
4	Digital	Darwin

SST(Student_no,Major,Advisor)

From the business rules, it is clear that the table is in 3 NF from the rule (c) the non key columns depend on the primary key. But the problem arises because of the business rule (d) since the Advisor determines the Major(hidden dependency).

- ✔ Insert Anomaly -It is not possible to record the fact that DARWIN can advise Digital until we have a student majoring in DIGITAL to whom we can assign DARWIN as a advisor.

- ✔ Delete Anomaly- If the record of Student no 2is deleted we lose not only information of student 2 but also the fact that MARTIN advices in NETWORK.

BCNF prevents the problems by stating that any dependency must be explicitly shown. The solution is to decompose the table to make dependency explicit.

The new table is as follows:

Student_Advisor

STUDENTNO	ADVISOR
1	Godbole
1	Silberchatz
2	Abraham
2	Yeshwantkanithkar
3	Martin
4	Balagurusamy
4	Abraham
5	Godbole

ADVISOR	MAJOR

Godbole	Os
Silberchatz	Rdbms
Abraham	Rdbms
Yeshwantkanithkar	C
Martin	Network
Balagurusamy	C

Advisor_Subject

Fourth Normal Form(Multi-valued Dependency)

A table is in fourth normal form (4NF) if and only if it is in BCNF and contains no more than one multi-valued dependency.

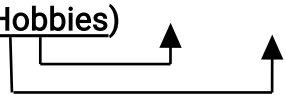
Anomalies can occur in relations in BCNF if there is more than one multi-valued dependency.

If A--->B and A--->C but B and C are unrelated, i.e. A--->(B,C) is false, then we have more than one multi-valued dependency

Ex: **Stu_info**

Studentno	Skills	Hobbies
1	Programming	Cricket
1	Programming	Gardening
1	Management	Cricket
1	Management	Gardening
2	Management	Foot ball
2	Management	Singing
2	Designing	Football
2	Designing	Singing

Stu_info(Studentno,Skills,Hobbies)



This table is difficult to maintain since adding a new hobby requires multiple new rows corresponding to each skill. This problem is created by the pair of multi-valued dependencies STUDENT NO--->SKILLS and STUDENTNO--->HOBBIES. A much better alternative would be to decompose STU_INFO into two relations:

Skills Hobbies

Studentno	Skills
1	Programming
1	Management
2	Management
2	Designing
2	Singing

Studentno	Hobbies
1	Cricket
1	Gardening
2	Football

****

SQL | DDL, DML, DCL and TCL Commands

Structured Query Language(SQL) as we all know is the database language by the use of which we can perform certain operations on the existing database and also we can use this language to create a database. SQL uses certain commands like Create, Drop, Insert etc. to carry out the required tasks.

These SQL commands are mainly categorized into four categories as discussed below:

1. **DDL(Data Definition Language) :** DDL or Data Definition Language actually consists of the SQL commands that can be used to define the database schema. It simply deals with descriptions of the database schema and is used to create and modify the structure of database objects in database.
   Examples of DDL commands:
   - **CREATE** – is used to create the database or its objects (like table, index, function, views, store procedure and triggers).
     Syntax for CREATE TABLE
     ```
     CREATE TABLE table_name (column1 datatype, column2 datatype,
         column3 datatype,  ....);
     ```
     Syntax for CREATE USER
     CREATE USER *username* IDENTIFIED BY*password;*
     Syntax for CREATE DATABASE
     CREATE DATABASE *database-name;*
   - **DROP** – is used to delete objects from the database.
     Syntax for DROP TABLE
     ```
     DROP TABLE table_name;
     ```
   - **ALTER**-is used to alter the structure of the database.
     1. **Alter Table – ADD column**
        Syntax
        ```
        ALTER TABLE table_name
        ADD column_name datatype;
        ```
     2. **Alter Table – Drop column**
        Syntax
        ```
        ALTER TABLE table_name
        DROP COLUMN column_name;
        ```
     3. **Alter Table – Modify column**
        Syntax
        ALTER TABLE *table_name*
        MODIFY COLUMN *column_name datatype*;
        or
        (for oracle version 10G and above)

        ALTER TABLE *table_name*
        MODIFY *column_name datatype*;

   - **TRUNCATE**–is used to remove all records from a table, including all spaces allocated for the records are removed.
   - **COMMENT** –is used to add comments to the data dictionary.
   - **RENAME** –is used to rename an object existing in the database.

2. **DML(Data Manipulation Language) :** The SQL commands that deals with the manipulation of data present in database belong to DML or Data Manipulation Language and this includes most of the SQL statements.
   Examples of DML:
   - **SELECT** – is used to retrieve data from the database.
     Syntax
     SELECT *column1*, *column2*, ...
     FROM *table_name*;
     Select with where clause
     SELECT *column1*, *column2*, ...
     FROM *table_name where column_name=value*;
   - **INSERT** – is used to insert data into a table.
     Syntax
     INSERT INTO *table_name* (*column1*, *column2*, *column3*, ...)
     VALUES (*value1*, *value2*, *value3*, ...);
   - **UPDATE** – is used to update existing data within a table.
     Syntax
     UPDATE *table_name*
     SET *column1 = value1*, *column2 = value2*, ...
     WHERE *condition*;
   - **DELETE** – is used to delete records from a database table.
     Syntax
     DELETE FROM *table_name* WHERE *condition*;
3. **DCL(Data Control Language) :** DCL includes commands such as GRANT and REVOKE which mainly deals with the rights, permissions and other controls of the database system.
   Examples of DCL commands:
   - **GRANT**-gives user's access privileges to database.
     Syntax
     GRANT CONNECT, RESOURCE TO *user-name;*
     REVOKE-withdraw user's access privileges given by using the GRANT command.
     Syntax
     `Revoke connect,resource from jus;`

4. **TCL(transaction Control Language) :** TCL commands deals with the transaction within the database.
   Examples of TCL commands:
   - **COMMIT**– commits a Transaction.
   - **ROLLBACK**– rollbacks a transaction in case of any error occurs.
   - **SAVEPOINT**–sets a savepoint within a transaction.
   - **SET TRANSACTION**–specify characteristics for the transaction.

Working with NULL Values

What is a NULL Value?

A field with a NULL value is a field with no value.

- If a field in a table is optional, it is possible to insert a new record or update a record without adding a value to this field. Then, the field will be saved with a NULL value.
- A null value is different from a zero value or a field that contains spaces. A filed with a NULL value is one that has been left blank during record creation.

How to Test for NULL Values?

It is not possible to test for NULL values with comparison operators, such as =, <, or <>.

We will have to use the IS NULL and IS NOT NULL operators instead.

IS NULL Syntax

SELECT *column_names*
FROM *table_name*
WHERE *column_name* IS NULL;

IS NOT NULL Syntax

SELECT *column_names*
FROM *table_name*
WHERE *column_name* IS NOT NULL;

Joins

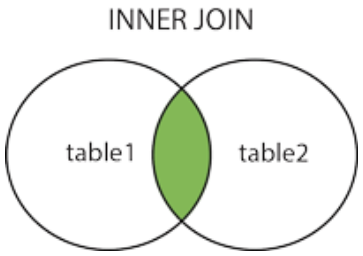
SQL JOIN

A JOIN clause is used to combine rows from two or more tables, based on a related column between them.

Different Types of SQL JOINs

Here are the different types of the JOINs in SQL:

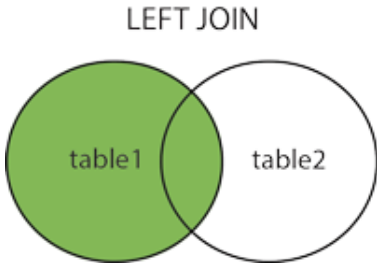
- **(INNER) JOIN**: Returns records that have matching values in both tables



Example

```
SELECT Orders.OrderID, Customers.CustomerName, Orders.OrderDate
FROM Orders
INNER JOIN Customers ON Orders.CustomerID=Customers.CustomerID;
```

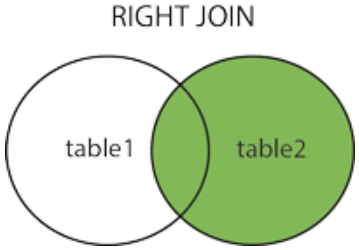
- **LEFT (OUTER) JOIN**: Returns all records from the left table, and the matched records from the right table



Example

```
SELECT column_name(s)
FROM table1
LEFT JOIN table2
ON table1.column_name = table2.column_name;
```

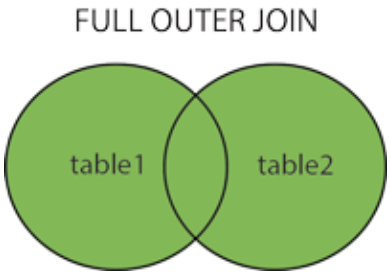
- **RIGHT (OUTER) JOIN**: Returns all records from the right table, and the matched records from the left table



Example

```
SELECT column_name(s)
FROM table1
RIGHT JOIN table2
ON table1.column_name = table2.column_name;
```

- **FULL (OUTER) JOIN**: Returns all records when there is a match in either left or right table



Example

SELECT *column_name(s)*
FROM *table1*
FULL OUTER JOIN *table2*
ON *table1.column_name = table2.column_name*
WHERE *condition*;

Self JOIN

A self JOIN is a regular join, but the table is joined with itself.

Syntax

SELECT *column_name(s)*
FROM *table1 T1, table1 T2*
WHERE *condition*;

Pseudonames or SQL Aliases

SQL aliases are used to give a table, or a column in a table, a temporary name.

Aliases are often used to make column names more readable.

An alias only exists for the duration of the query.

Alias Column Syntax

SELECT *column_name* AS *alias_name*
FROM *table_name;*

ROWID, ROWNUM

An Oracle ROWID uniquely identifies where a row resides on disk.

The information in a ROWID gives Oracle everything he needs to find your row, the disk number, the cylinder, block and offset into the block.

The ROWNUM is a "pseudo-column", a placeholder that you can reference in SQL*Plus. The ROWNUM can be used to write specialized SQL and tune SQL.

For example, to only display the first-10 rows, you might apply a ROWNUM filter:

```
select *
from (
  select * from my_view where alert_level=3 order by alert_time desc)
where
   rownum<=10;
```

The difference between ROWNUM and ROWID is that ROWNUM is temporary while ROWID is permanent. Another difference is that ROWID can be used to fetch a row, while ROWNUM only has meaning within the context of a single SQL statement, a way of referencing rows within a fetched result set.

STRING FUNCTIONS

1. LENGTH(*string*) : The functions gives the length of the string that is indicated as parameter.

Return type **:** INTEGER

Return data : Number of characters in *string*

Syntax :

length('NuoDB')

Example:

Select length('NuoDB') from dual;

output : 5

2. **Lower(***string***)** : converts the string indicated as parameter into lower case.

Return type : string

Return data : string in lowercase

Syntax:

lower('NuoDB')

Example:

select lower(‘NuoDB’)from dual;

Output : nuodb

3. **LTRIM(*string*))** : Removes leading spaces from the start of the string indicated as parameter in the function.

Return type : string

Return data : string without trailing white spaces.

syntax

ltrim(' NuoDB ')

Example:

select ltrim(‘ NuoDB’) from dual;

Output : nuodb

4. **RTRIM(*string*))** : Removes trailing spaces from the end of the string indicated as parameter in the function.

Return type : string

Return data : string without leading white spaces.

syntax

rtrim(' NuoDB ')

Example:

select rtrim(‘ NuoDB ’)from dual;

Output : nuodb

5. **TRIM(*string*))** : Removes trailing spaces from the end of the string indicated as parameter in the function.

Return type : string

Return data : string without leading white spaces.

Syntax

```
TRIM([ LEADING | TRAILING | BOTH ] [string1] FROM string2)
```

Example:

Select trim(both 'x' from 'xTomxx') from dual; // o/p Tom

select trim(leading 'x' from 'xTomxx') from dual; // o/p Tomxx

6. **SUBSTR(*string, int , int*):** Returns a subset of *string* starting from *start_pos*, optionally for *length*

Return type: String

Return data: part of the string

Syntax:

SUBSTR(*string, start_pos* [,*length*])

Example

Select substring('alphabet',3,2)**from dual;**

Output

ph

7. **CONCAT(*string, string*):** joins two strings together

Return type: String
Return data: concatenated strings

Syntax:

CONCAT(*string, string***):**

Example

Select concat('hello','world')**from dual;**

Output

Helloworld

8. INITCAP(*string)*

Return type : String

Return Data : Initial letter capitalized String

Syntax:

Initcap('string')

Example

select inticap('programming') from dual;

output

Programming

9. **INSTR(*string,char,int)***

RETURN TYPE :

```
select Instr('st.josephs college','o',1) from dual;
```

AGGREGATE FUNCTIONS

DATE FUNCTIONS

SET OPERATIONS

UNION, INTERSECTION AND MINUS

1. UNION Operator

The UNION operator is used to combine the result-set of two or more SELECT statements.

- Each SELECT statement within UNION must have the same number of columns
- The columns must also have similar data types
- The columns in each SELECT statement must also be in the same order

UNION Syntax

SELECT *column_name(s)* FROM *table1*
UNION
SELECT *column_name(s)* FROM *table2*;0

UNION ALL Syntax

The UNION operator selects only distinct values by default. To allow duplicate values, use UNION ALL:

SELECT *column_name(s)* FROM *table1*
UNION ALL
SELECT *column_name(s)* FROM *table2*;

2. INTERSECTION

The INTERSECTION operator is used to combine the result-set of two or more SELECT statements. It retrieves the common records from the given two tables.

- Each SELECT statement within INTERSECTION must have the same number of columns
- The columns must also have similar data types
- The columns in each SELECT statement must also be in the same order

SYNTAX

```
select attribute1,attribute2,… from table1
  intersect
select attribute1,attribute2 from table2;
```

Example

```
select empno,empname from emp
  intersect
  select empno,empname from emp1;
```

3. MINUS

The MINUS operator is used to combine the result-set of two or more SELECT statements. It retrieves the common records from the given two tables and all the rest records from table1 will be displayed.

- Each SELECT statement within MINUS must have the same number of columns
- The columns must also have similar data types
- The columns in each SELECT statement must also be in the same order

Syntax

Select attribute1,attribute2,…from table1
Minus
Select attribute1,attribute2,…from table2;

Example

```
select empno,empname from emp
 minus
select empno,empname from emp1;
```

VIEW

In SQL, a view is a virtual table based on the result-set of an SQL statement.

A view contains rows and columns, just like a real table. The fields in a view are fields from one or more real tables in the database.

You can add SQL functions, WHERE, and JOIN statements to a view and present the data as if the data were coming from one single table.

CREATE VIEW Syntax

CREATE VIEW *view_name* AS
SELECT *column1*, *column2*, ...
FROM *table_name*
WHERE *condition*;

Note: A view always shows up-to-date data! The database engine recreates the data, using the view's SQL statement, every time a user queries a view.

UNIT- 5
PL/SQL

PL/SQL is the procedural extension to SQL with design features of programming languages.
Data Manipulation and query statements of SQL are included with procedural units of code.
Block Structure PL/SQL programs are divided up into structures known as blocks, with each block containing PL/ SQL and SQL statements. A PL/SQL block has the following structure:

Benefits of PL/SQL

- PL/SQL is portable.
- We can declare variables.
- We can program with procedural language control structures.
- PL/SQL can handle errors.
- Improved data security and integrity.
- Improved performance and code clarity.
- Easy Maintenance.

PL/SQL Block Structure

```
DECLARE                - Optional
      declaration_statements
BEGIN                  - Mandatory
      executable_statements
EXCEPTION              - Optional
      exception_handling_statements
END;                   -Mandatory
```

PL/SQL is a block-structured language ,it means that it can be divided into logical blocks .A PL/SQL block consists of up to three sections: declarative(optional), executable (required),and exception handling (optional).

Declarative(Optional) - Contains all variables ,constants ,cursors and user-defined exceptions that are referenced in the executable sections and declarative sections.

Executable(Mandatory) - Contains SQL statements to manipulate data in the database and PL/SQL statements to manipulate data in the block.

Exception Handling(Optional) - Specifies the actions to perform when errors and abnormal conditions arise in the executable section.

A PL/SQL block can be anonymous block, procedure, or function.

Anonymous PL/SQL block is the basic ,unnamed unit of a PL/SQL program.

Procedures and Functions can be compiled separately and stored permanently in a

oracle database ,ready to be executed.

Declaring & Initialization of PL/SQL variables

Examples:

```
v_empno  NUMBER;
v_empname  VARCHAR(15) := 'jus';
v_deptno  NUMBER  NOT  NULL  := 20;
v_comm  CONSTANT NUMBER :=140;
```

Declaring Variables with the % TYPE Attribute

Syntax:

```
identifier          Table .column_name % TYPE;
```

Ex:

```
v_empname        emp.empname%TYPE;
```

The variable v_empname is defined to be of the same data type as the empname column in the emp table .%TYPE provides the data type of a database column.

Declaring Variables with the % ROWTYPE Attribute

Syntax:

```
identifier          Table % ROWTYPE;
```

Ex:

```
v_emp        emp%ROWTYPE;
```

Using Bind Variables

To reference a bind variable in PL/SQL,we must prefix its name with a colon(:).

```
VARIABLE  g_salary  NUMBER
BEGIN
   SELECT salary  INTO : g_salary FROM  emp  WHERE empno = 101;
END;
/
PRINT g_salary
```

Example of Anonymous Block(PL/SQL Block):

```
SET SERVER OUTPUT ON

DECLARE
v_empno   NUMBER;
v_empname  emp.empname%TYPE;
BEGIN
v_empno := &empno;
SELECT empname  INTO  v_empname FROM  emp WHERE empno = v_empno;
```

```
DBMS_OUTPUT. PUT_LINE(v_empname);
END;
```

Conditional Logic

We can use the IF, THEN, ELSE, ELSIF, and END IF keywords to perform conditional logic:

```
IF condition1 THEN
      statements1;
ELSIF condition2 THEN
      statements2 ;
ELSE
      statements3 ;
END IF;
```

where

- condition1 and condition2 are Boolean expressions that evaluate to true or false.
- statements1, statements2, and statements3 are PL/SQL statements.

Loops

We can use a loop to run statements zero or more times. There are three types of loops in PL/SQL:

- FOR loops run a predetermined number of times.

Simple Loops

Simple loops run until you explicitly end the loop.The syntax for a simple loop is as follows:

```
LOOP
   statements
END LOOP;
```

To end the loop,we can use either an EXIT or an EXIT WHEN statement. The EXIT statement ends a loop immediately; the EXIT WHEN statement ends a loop when a specified condition occurs.

The following example shows a simple loop. A variable named v_counter is initialized to 0 prior to the beginning of the loop. The loop adds 1 to v_counter and exits when v_counter is equal to 5 using an EXIT WHEN statement.

```
v_counter := 0;
 LOOP
v_counter := v_counter + 1;
EXIT WHEN v_counter = 5;
 END LOOP;
```

WHILE Loops

A WHILE loop runs until a specified condition occurs. The syntax for a WHILE loop is as follows:

```
WHILE  condition
          LOOP
      statements
END LOOP;
```

The following example shows a WHILE loop that executes while the v_counter variable is less than 6:

```
v_counter := 0;
WHILE v_counter < 6
LOOP
     v_counter := v_counter + 1;
 END LOOP;
```

FOR Loops

A FOR loop runs a predetermined number of times; you determine the number of times the loop runs by specifying the lower and upper bounds for a loop variable. The loop variable is then incremented (or decremented) each time around the loop. The syntax for a FOR loop is as follows:

```
FOR loop_variable IN [REVERSE] lower_bound ..upper_bound
LOOP
     statements
END LOOP;
```

Example

```
FOR  v_counter  IN  1..5
LOOP
     DBMS_OUTPUT.PUT_LINE(v_counter);
END LOOP;
```

The following example uses REVERSE:

```
FOR v_counter IN REVERSE 1..5
 LOOP
    DBMS_OUTPUT.PUT_LINE(v_counter);
 END LOOP;
```

Cursors

- Cursor is a private SQL work area.
- Cursor is used to fetch rows returned by a query. we can retrieve the rows into the cursor using a query and then fetch the rows one at a time from the cursor.

They are two types of cursors:

i) Implicit cursors - The oracle server uses implicit cursors to parse and execute your SQL statements.

ii) Explicit cursors- Explicit cursors are explicitly declared by the programmer.

SQL Cursor Attributes

Using SQL Attributes we can test the outcome of the SQL statements.

SQL%ROWCOUNT -- Number of rows affected by the most recent SQL statement(an integer value).
SQL%FOUND -- Boolean attribute that evaluates to TRUE if the most recent SQL statement affects one or more rows.

SQL%NOTFOUND-- Boolean attribute that evaluates to TRUE if the most recent SQL statement does not affect any rows.

SQL%ISOPEN--Always evaluates to FALSE because PL/SQL closes implicit cursors immediately after they are executed.

Example of using Implicit Cursor

```
VARIABLE rows_deleted VARCHAR(25)
DECLARE
v_empid emp.empid%type;
BEGIN
v_empid := &v_empid;
DELETE FROM emp WHERE empid = v_empid;
: rows_deleted := ( SQL%ROWCOUNT || ' rows deleted.' ) ;
END;
/
PRINT rows_deleted
```

Explicit Cursor

The following five steps are followed when using a explicit cursor:

1. Declare variables to store the column values for a row.
2. Declare the cursor, which contains a query.
3. Open the cursor.
4. Fetch the rows from the cursor one at a time and store the column values in the variables declared in Step 1. You would then do something with those variables, such as display them on the screen, use them in a calculation, and so on.
5. Close the cursor.

Example of using Explicit Cursor

```
DECLARE
v_emp emp%ROWTYPE ;
CURSOR emp_cursor IS SELECT * FROM emp WHERE deptno=10;
BEGIN
OPEN emp_cursor;
LOOP
FETCH emp_cursor INTO v_emp ;
EXIT WHEN emp_cursor %NOT FOUND;
DBMS_OUTPUT.PUT_LINE(v_emp.empno ||','||v_emp.empname||','||v_emp.salary);
```

```
END LOOP;
CLOSE emp_cursor;
END;
```

Cursors and FOR Loops

We can use a FOR loop to access the rows in a cursor. When we do this, we don't have to explicitly open and close the cursor—the FOR loop does this automatically.

Example

```
DECLARE
CURSOR  emp_cursor  IS SELECT * FROM emp  WHERE  deptno=10;
BEGIN
FOR v_emp IN emp_cursor
LOOP
DBMS_OUTPUT.PUT_LINE(v_emp.empno ||','||v_emp.empname||','||v_emp.salary);
END LOOP;
END;
```

Exceptions

Exceptions are used to handle run-time errors in your PL/SQL code.

Exception can be divided into following types.

- Predefined
- Non Predefined
- User Defined

Predefined and Non Predefined are implicitly raised by the oracle server where as User Defined is explicitly raised by the user.

Trapping Exceptions

Syntax;

```
EXCEPTION
WHEN exception1    THEN
    statement1;
    statement2;
WHEN exception2     THEN              ---Optional
   statement 1;
   statement 2;
WHEN OTHERS       THEN             ---Optional
  statement 1;
```

Sample Predefined exceptions:

- NO_DATA_ FOUND --- Single row SELECT returned no data.

- TOO_MANY_ROWS---Single row SELECT returned more than one row.
- ZERO_DIVIDE ---- Attempt to divide by zero.
- INVALID_CURSOR --- Illegal cursor operation occurred.
- DUP_VAL_ON_INDEX---Attempt to insert a duplicate value.

The RAISE_APPLICATION_ERROR procedure

This procedure is used to communicate a predefined exception interactively by returning non standard error code and error message.

In the syntax:

error_number --- is a user-specified number for the exception between -20000 and -20999.

error message --- is the user-specified message for the exception.

Example

```
DECLARE
   v_empno     emp.empno%TYPE ;
   v_emp       emp%ROWTYPE ;
  BEGIN
   v_empno :=  &empno;
  SELECT  *  INTO v_emp  FROM emp WHERE  empno= v_empno;
  DBMS_OUTPUT. PUT_LINE (v_emp.empno
||','||v_emp.empname||','||v_emp.salary);
  EXCEPTION
       WHEN  NO_DATA_FOUND     THEN
            DBMS_OUTPUT . PUT_LINE(' The data is not present ' );
        WHEN TOO_MANY _ROWS   THEN
           RAISE_APPLICATION_ERROR(-20010, ' More than one row is present ');

END;
```

PROCEDURE

- ✔ A Procedure is a named PL/SQL block which contains a group of SQL and PL/SQL statements.
- ✔ A Procedure is type of subprogram that performs an action.
- ✔ A Procedure can be stored in the database ,as a schema object ,for repeated execution
- ✔ It provides modularity ,reusability ,extensibility and maintainability

Syntax for Creating Procedures

```
CREATE  [OR REPLACE ]  PROCEDURE procedure_name
[ (parameter1 [mode1] datatype1,  parameter2 [mode2] datatype2,.....) ]
IS/AS
PL/SQL Block;
```

Example:

```
CREATE OR REPLACE  PROCEDURE  raise_salary (p_empid  IN  emp.empid%TYPE)
 IS
 BEGIN
 UPDATE emp SET salary = (salary *1.10)  where empid  = p_empid ;
 END ;
   /
Procedure created.
```

To Invoke a procedure ,use the EXECUTE command.

```
EXECUTE   raise_salary (102);
```

Formal Versus Actual Parameters

- Formal parameters: variable declared in the parameter list of a subprogram specification
  Example:
  CREATE PROCEDURE raise_salary(p_empid IN emp.empid%TYPE)
  .....
  END;

- Actual parameters: variables or expressions referenced in the parameter list of a subprogram call
  Example:
  raise_salary(102);

To view the compilation errors
Ex:

```
SHOW ERRORS   OR  SHOW ERRORS raise_salary
```

To view the source code of the procedure from the data dictionary
Ex:

```
SELECT  text  FROM  user_source  WHERE  name = ' RAISE_SALARY ';
```

Removing a Procedure
Syntax:

```
DROP PROCEDURE  procedure_name;
```

Ex:

```
DROP PROCEDURE raise_salary;
```

FUNCTION

- ✔ Function is a named PL/SQL block that returns a value.
- ✔ A Function can be stored in a database as a schema object for repeated execution.
- ✔ A Function is called as an part of an SQL expression.

Syntax for Creating Functions

```
CREATE [ OR REPLACE ] FUNCTION function_name
[ (parameter1 [mode1] datatype1,  parameter2 [mode2] datatype2,.....) ]
RETURN  datatype
 IS/AS
PL/SQL Block;
```

Example:

```
CREATE OR REPLACE  FUNCTION get_salary(p_empid  emp.empid%TYPE)
RETURN  NUMBER
IS
v_salary   emp.salary % TYPE;
BEGIN
SELECT salary INTO v_salary FROM emp  WHERE empid = p_empid;
RETURN  v_salary;
END;
/
Function created.
```

Executing Functions

```
VARIABLE  g_salary NUMBER
EXECUTE  :g_salary : = get_salary(103);
PRINT g_salary
```

Example of Invoking Function in SQL Expressions;

```
CREATE OR REPLACE FUNCTION  tax_calc(p_value  IN  NUMBER)
RETURN  NUMBER
IS
BEGIN
   RETURN (p_value * 0.08);
END;
/
Function created.

SELECT  empno, empname ,salary, tax_calc(salary) as tax  FROM  emp;
```

Removing Functions

Syntax:

```
      DROP  FUNCTION   function_name;
```

Ex:

```
     DROP  FUNCTION  tax_calc;
```

PACKAGES

- ✔ Group logically related PL/SQL types ,items ,and subprograms.
- ✔ It consists of two parts:
  - Specification
  - Body
- ✔ Packages cannot be invoked ,parameterized ,or nested .
- ✔ It allows the Oracle server to read multiple objects into memory at once.

A package specification can exist without a package body ,but a package body cannot exist without a package specification.

Advantages of using Packages

- ✔ Overloading is possible by using Package.
- ✔ Better Performance can be obtained since the entire package is loaded into the memory when the package is first referenced.
- ✔ Persistency of Variables and Cursors.
- ✔ Security is provided by not allowing to access and by hiding the private constructs in the package body

Syntax for creating Package specification:

```
CREATE [OR REPLACE] PACKAGE package_name
{IS | AS}
   public type and item declarations
  subprogram_specification
END package_name;
```

Syntax for creating Package Body specification:

```
CREATE [OR REPLACE] PACKAGE BODY package_name
{IS | AS}
  package_body
END package_name;
```

Example: Creating Pacakage specification

```
CREATE OR REPLACE PACKAGE  salary_package
IS
g_sal NUMBER := 5000;
PROCEDURE  reset_salary (p_sal  IN NUMBER);
END  salary_package;
```

Example: Creating Pacakage Body

```
CREATE OR REPLACE PACKAGE BODY salary_package
IS
FUNCTION  validate_salary(p_sal  IN NUMBER)
RETURN  BOOLEAN
IS
v_max_sal  NUMBER ;
BEGIN
SELECT MAX(salary) INTO v_max_sal  FROM emp;
IF   p_sal  > v_max_sal  THEN
RETURN (FALSE)
ELSE
RETURN (TRUE);
END  IF;
END  validate_salary ;
PROCEDURE  reset_salary (p_sal  IN NUMBER)
IS
BEGIN
IF  validate_salary(p_sal)  THEN
g_sal := p_sal ;
ELSE
RAISE_APPLICATION_ERROR(-20222,'Invalid salary ');
END  IF;
END reset_salary;
END salary_package;
/
Package created;
```

Invoking Package Constructs

```
Execute salary_package.reset_salary(10000);
```

Example for Declaring a Bodiless Package

```
CREATE OR REPLACE PACKAGE  global_consts
 IS
mile_2_kilo   COSTANT  NUMBER := 1.6093;
kilo_2_mile   CONSTANT  NUMBER:= 0.6214;
yard_2_meter CONSTANT  NUMBER := 0.9144;
meter_2_yard  CONSTANT NUMBER := 1.0936;
END global_consts;
/
Package created;
```

Invoking Package Constructs

```
Execute  DBMS_OUTPUT.PUT_LINE( '20 miles = '|| 20 * global_consts.mile_2_kilo || '
km');
```

Removing Packages

To remove the package body

Syntax:

DROP PACKAGE BODY *package_name*;

To remove the package specification and the body

Syntax:

DROP PACKAGE *package_name*;

TRIGGERS

- ✔ A Trigger is a PL/SQL block or a PL/SQL procedure associated with a table ,view , schema , or the database.
- ✔ It executes implicitly whenever a particular event takes place.
- ✔ Trigger can be divided into Two types.
    - o Application Trigger: Fires whenever an event occurs with a particular application.
    - o Database Trigger: Fires whenever a data event (such as DML) or a system event (such as logon or shutdown) occurs on a schema or database.

Creating DML Triggers

A triggering statement contains :

- ✔ Trigger timing
    - For table : BEFORE ,AFTER
    - For view : INSTEAD OF
- ✔ Triggering event : INSERT,UPDATE, OR DELTE
- ✔ Table name : On table ,view
- ✔ Trigger type: Row or Statement
- ✔ WHEN clause: Restricting condition
- ✔ Trigger body : PL/SQL block

Trigger timing

BEFORE : Execute the trigger body before the triggering DML event on a table.
AFTER : Execute the trigger body after the triggering DML event on a table.
INSTEAD OF : Execute the trigger body instead of triggering statement. This is used for views that are not modifiable.

Triggering event

The triggering event can contain one ,or two or all three of these DML operations.

INSERT OR UPDATE OR DELETE

Trigger type

Statement: The Trigger body executes once for the triggering event. It is default
Row : The Trigger body executes once for each row affected by the triggering event.

Trigger Body

The Trigger body is a PL/SQL or a call to a procedure.

Creating DML Statement Trigger

Syntax:

```
CREATE OR [ REPLACE ] TRIGGER  trigger_name
timing
 event 1  [ OR event 2 OR event3]
ON  table_name
trigger_body
```

Example:

```
CREATE  OR REPLACE TRIGGER secure_emp
BEFORE  INSERT OR UPDATE OR DELETE ON emp
BEGIN
IF TO_CHAR( SYSDATE , 'DY' ) =  'sun'  THEN
RAISE_APPLICATION_ERROR (-20198,'You are not allowed to do any manipulation on sunday');
END IF;
END;
```

Using OLD and NEW Qualifiers

Data operation	Old value	New Value
INSERT	Null	Inserted value
UPDATE	Value before update	Value after update
DELETE	Value before delete	Null

- The OLD and NEW qualifiers are available only in ROW level Triggers.
- Prefix these qualifiers with a colon(:) in every SQL and PL/SQL statement.
- There is no colon(:) prefix if the qualifiers are referenced in the WHEN restricting condition.

Example:

```
CREATE OR REPLACE TRIGGER restrict_sal
BEFORE  UPDATE ON emp
FOR EACH ROW
BEGIN
IF :OLD.SAL  >  :NEW.SAL THEN
```

```
RAISE_APPLICATION_ERROR(-20222, ' You are not allowed to decrement the salary ');
END IF;
END;
/
```

Dropping a Trigger

Syntax:

```
DROP TRIGGER trigger_name;
```

Ex:

```
DROP TRIGGER restrict_sal;
```