

## Chapter-1

### **Data:**

Data are the raw facts that can be obtained after some experiments or observations. Raw data is of no use until and unless we process it to find some useful information from it.

### **Database:**

A **database** is the collection of related persistent data and contains information relevant to an enterprise. The database is also called the repository or container for a collection of data files. For example, **university database** for maintaining information about students, courses and grades in university

### **Database Management System:**

Management of data involves a way to store data and also provides a mechanism for manipulation of that data. Database management systems are basically designed to manage large volume of information.

A **database system** is basically just a computerized record-keeping system. A database system involves four major components: **data, hardware, software, and users**. The **database management system (DBMS)** is the software that handles all access to the database. It is defined as the collection of interrelated data and a set of programs to access those data.

The primary goal of DBMS is to store and retrieve data in both **convenient** (easy method) and **efficient** (capable of performing well) manner. Some of the examples of DBMS are Oracle, SQL-Server, MySQL, MS Access etc.

### **Applications of DBMS:**

- Banking:  
To store information about customers, their account number, balance etc.
- Airlines:  
For reservations and schedule information.
- Telecommunication:  
To keep records of customers, call made, balance left, generating monthly bills etc.
- Universities:  
To keep records of students, courses, marks of students etc.
- Sales:  
To keep information of customers, products list, purchase information etc.
- Manufacturing:  
To store orders, tracking production of items etc.
- Human Resources:  
To keep records of employee, their salary, bonus etc.

## Characteristics of database approach:

The main characteristics of database approach are discussed below:

### 1. Self describing nature of database system:

The fundamental characteristic of the database approach is that the database system contains not only the database itself but also complete definition or description of the database structure. This definition is stored in the DBMS catalog which contains the information such as the structure of each file, the type and storage format of each data item. Eg.

Field	Type	Collation	Attributes	Null	Default	Extra	
<b>titel</b>	varchar(200)	latin1_general_ci		Yes	NULL		
<b>interpret</b>	varchar(200)	latin1_general_ci		Yes	NULL		
<b>jahr</b>	int(11)			Yes	NULL		
<b>id</b>	bigint(20)		UNSIGNED	No		auto_increment	

Row Statistics		Space usage	
Statements	Value	Type	Usage
<b>Format</b>	dynamic	<b>Data</b>	148 Bytes
<b>Collation</b>	latin1_general_ci	<b>Index</b>	2,048 Bytes
<b>Rows</b>	3	<b>Total</b>	2,196 Bytes
<b>Row length &amp;</b>	49		
<b>Row size &amp;</b>	732 Bytes		
<b>Next Autoindex</b>	6		
<b>Creation</b>	Oct 25, 2005 at 01:32 PM		
<b>Last update</b>	Oct 25, 2005 at 01:32 PM		

### 2. Insulation between programs and data :

In traditional file processing system, the structure of data file is embedded in the application program. Hence if we have to make any changes in the format of the data, the whole application program will also have to be changed.

But in the case of DBMS, database and the application program are separately situated. The structure of data files is stored in the DBMS catalog separately from the access programs. Hence in most cases DBMS access programs do not need such changes. We call this property as **program-data independence**.

For example, a file access program may be written in such a way that it can just access the name of a customer of any company but if we want to add another field, say, address of the customer, then in such a case this program will no longer be useful. But in the case of the DBMS, we just have to add the new field “address” in the catalog and the next time DBMS refers to the catalog, the new structure of the records will be accessed and hence we do not have to make change in the program.

### 3. Support of multiple view of the data:

Let us consider an example of **student information system** of a college, where all the data of student, courses, information of college etc are stored in a database. It is

obvious that there is more than one user of this system and also their interest is different. i.e. student are generally interested in finding out the marks obtained, whereas teachers are able to put marks, view information about students, similarly, visitors are able to find the information of the college.

A database typically has many users, and their perspective of viewing the database is different. In the example given above, every user is accessing data from the same database but what they see and can access is different. A view may be subset of the database or it may contain virtual data that is derived from the database files but is not stored explicitly.

#### **4. Sharing of data and multi user transaction processing:**

Database system should allow multiple users to access same database at the same time. For example, in online air ticket reservation system many users are accessing the same site at the same time. Hence for every seat, DBMS should ensure that only one user should be given access to reserve the seat.

Hence to ensure the correctness of this type of transaction DBMS must include concurrency control software to ensure that several users trying to update same data do so in a controlled manner so that the result of the updates is correct. In order to do so, DBMS uses lock based protocol and time stamp based protocol.

### **Purpose of Database System:**

Traditionally, file processing system was used to manage information. It stores data in various files of different application programs to extract or insert data to appropriate file. File processing system has several drawbacks due to which database management system is required. Database management system removes problems found in file processing system. Some of the major problems of file processing systems are:

#### **1. Data redundancy and inconsistency:**

In file processing system, different programmer creates files and writes application programs to access it. After a long period of time files may exist with different formats and application programs may be written in many different programming languages. Moreover, same information may be duplicated in several files. We have to pay for higher storage and access cost for such redundancy. It may lead database in inconsistent state because update made may be reflected in one file but it may not be reflected in another files where same information exist in another files.

#### **2. Difficulty in accessing data:**

In file processing system, we can not easily access required data stored in particular file.

For each new task we have to write a new application program. File processing system can not allow data to be retrieved in convenient and efficient manner.

#### **3. Integrity problem:**

In database, we required to enforce certain type consistency constraints to ensure the database correctness or to enforce certain business rules. It is in fact called integrity constraints (e.g. account balance > 0), integrity of database need not to be violated. In file processing system, integrity constraint becomes the part of application program.

Programmer need to write appropriate code to enforce it. When new constraints are required to add or change existing one, it is difficult to change program to enforce it.

#### 4. **Atomicity problem:**

Failures may lead database in an inconsistent state with partial updates. For example, failure occurs while transferring fund from account A to B. There would be the case that certain amount from account A is retrieved and it is updated but failure occurs just before it is deposited to account B, such case may lead database in inconsistent state.

#### 5. **Concurrent access problem:**

Concurrent accessed increase the overall performance of system providing fast response time but uncontrolled concurrent accesses can lead inconsistencies in system. File processing system allow concurrent access but it is unable to coordinate different application programs so database may lead in inconsistent state. E.g. two people reading a balance and updating it at the same time.

#### 6. **Security problem:**

Since file processing system consist large no. of application programs and it is added in ad hoc manner. So it is difficult to enforce security to each application to allow accessing only part of data/database for individual database users.

### **Data Abstraction:**

Data abstraction is the technique of hiding the complexity of the database to its users. There are three levels of data abstraction which are discussed below.

- **Physical Level or Internal Level:**

It is the lowest level of abstraction and describes *how* the data in the database are actually stored. This level describes complex low-level data structures in detail and is concerned with the way the data is physically stored. Data only exists at physical level.

- **Logical Level or Conceptual Level:**

This is the next higher level of abstraction and describes *what* data are stored in the database, and what relationships exist among those data. It describes the structure of whole database and hides details of physical storage structure. It concentrates on describing entities, data types, relationships, attributes and constraints. All of the views must be derivable from this conceptual schema.

- **View Level or External Level:**

It is the highest level of abstraction and is concerned with the way the data is seen by individual users. This level simplifies the users' interaction with the system. It includes a number of user views and hence is guided by the end user requirement. It describes only those part of the database in which the users are interested and hides rest of all from those users. Each user group refers to its own external schema.

Example:

*view level*

- View result
- View student information

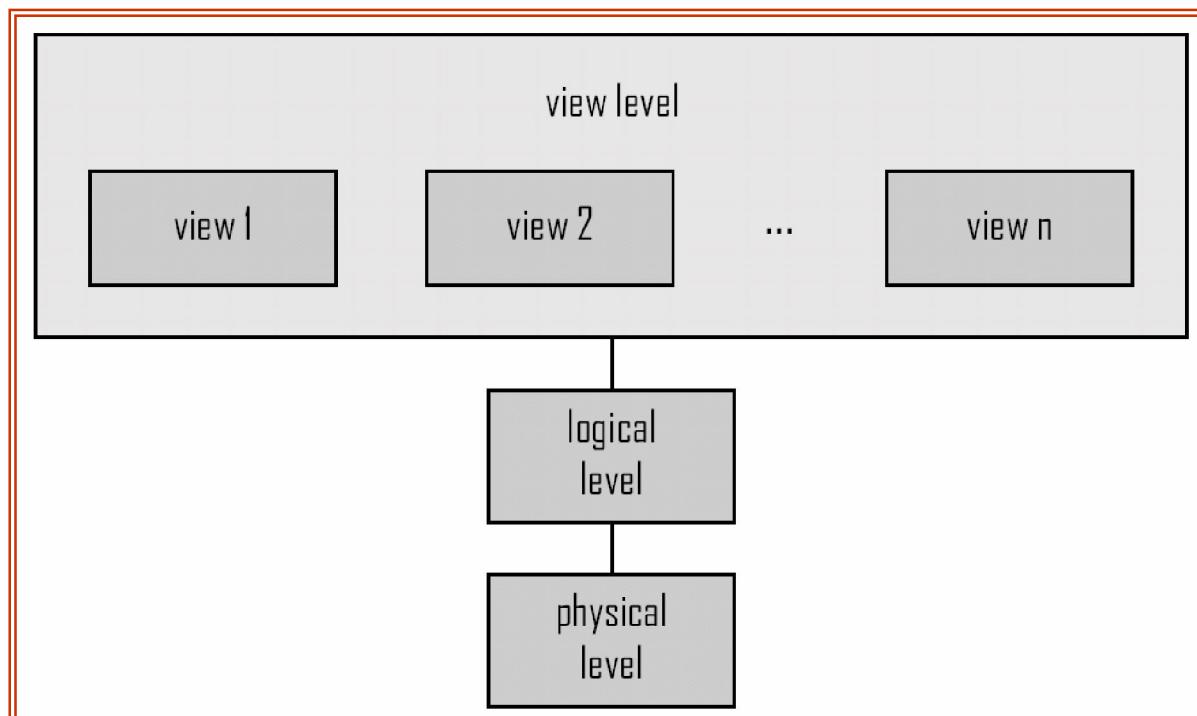
*logical level: entire database schema*

- Courses (CourseNo,CourseName,Credits,Dept)
- Student (StudentID,Lname,Fname,Level,Major)
- Grade (StudentID,CourseNo,mark)

*physical level:*

- how these tables are stored, how many bytes it required etc.

The DBMS must transform a request specified on an external schema into a request against the conceptual schema, and then into a request on the internal schema for processing over the database. The process of transforming requests and results between levels is called mapping.



**Fig : Data abstraction level**

## Instances and Schemas:

- *Instances:*

Databases change over time as information is inserted and deleted. The collection of information stored in the database at a particular moment is called an instance of the database. It is also known as database state.

- *Schema:*

The overall design of the database which is not expected to change frequently is called the database schema. There are three schemas, partitioned according to the levels of abstraction. The physical schema describes the database design at physical level. The logical schema describes the database design at the logical level. The schema at the view level is sometimes called subschema and describes the view of the database. A database may have several subschemas.

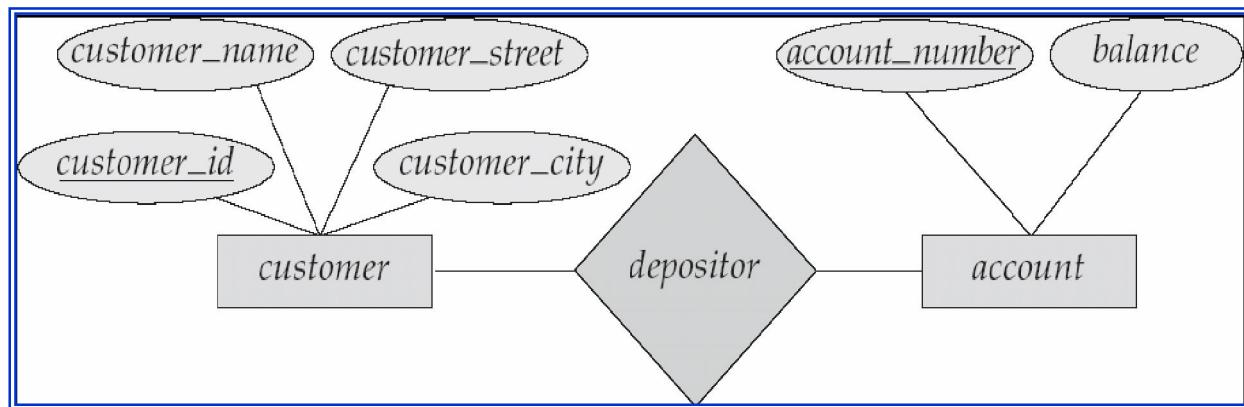
## Data Models:

The basic structure or design of the database is the **data model**. A data model is a collection of conceptual tools for describing data, data relationships, data semantics, and consistency constraints. Some data models are given below:

### Entity-Relationship Model:

**Entity-relationship (E-R) model** is a *high level* data model based on a perception of a real world that consists of collection of basic objects, called **entities**, and of **relationships** among these entities. An **entity** is a thing or object in the real world that 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 set of all entities of the same type is called an **entity set** and the set of all relationships of the same type is called a **relationship set**. Overall logical structure of a database can be expressed graphically by E-R diagram. The basic components of this diagram are:

- **Rectangles** (represent entity sets)
- **Ellipses** (represent attributes)
- **Diamonds** (represent relationship sets among entity sets)
- **Lines** (link attributes to entity sets and entity sets to relationship sets)



The figure shown above is an example of E-R diagram.

## Relational Model:

It is the current favorite model. The relational model is a *lower level* model that uses a collection of tables to represent both data and relationships among those data. Each table has multiple columns, and each column has a unique name. Each table corresponds to an entity set or relationship set, and each row represents an instance of that entity set or relationship set. Relationships link rows from two tables by embedding row identifiers (keys) from one table as attribute values in the other table. Structured query language (SQL) is used to manipulate data stored in tables.

<i>customer_id</i>	<i>customer_name</i>	<i>customer_street</i>	<i>customer_city</i>
192-83-7465	Johnson	12 Alma St.	Palo Alto
677-89-9011	Hayes	3 Main St.	Harrison
182-73-6091	Turner	123 Putnam Ave.	Stamford
321-12-3123	Jones	100 Main St.	Harrison
336-66-9999	Lindsay	175 Park Ave.	Pittsfield
019-28-3746	Smith	72 North St.	Rye

(a) The *customer* table

<i>account_number</i>	<i>balance</i>
A-101	500
A-215	700
A-102	400
A-305	350
A-201	900
A-217	750
A-222	700

(b) The *account* table

<i>customer_id</i>	<i>account_number</i>
192-83-7465	A-101
192-83-7465	A-201
019-28-3746	A-215
677-89-9011	A-102
182-73-6091	A-305
321-12-3123	A-217
336-66-9999	A-222
019-28-3746	A-201

(c) The *depositor* table

The relational data model is the most widely used data model, and a vast majority of current database systems are based on the relational model. The relational model is at a lower level of abstraction than the E-R model. Database designs are often carried out in the E-R model, and then translated to the relational model.

## **Other Data Models:**

### ***Object-based data models (Object-oriented and Object-relational):***

Object oriented data model is extension to E-R model with the notion of encapsulation, methods (functions) and object identity. It is based on collection of objects, like the E-R model. An object contains values stored in instance variables within the object.

### ***Network model:***

In network model, data are represented by the set of records and relationships among data are represented by links.

### ***Hierarchical model:***

Hierarchical model also represents data by a set of records but records are organized in hierarchical or order structure and database is a collection of such disjoint trees. The nodes of the tree represent record types. Hierarchical tree consists one root record type along with zero or more occurrences of its dependent subtree and each dependent subtree is again hierarchical.

## **Database Languages:**

### **Definition Language (DDL)**

Data definition language used to specify database scheme. For example, following DDL statement in SQL defines account relation.

```
create table account
(
    account_no  char(2),
    balance      integer
)
```

The execution of above DDL statement creates table account. Moreover, it updates special set of tables called data dictionary or data directory. Data dictionary contains meta data, that is data about data. For example table containing tables' information like table name, owner, created date, modified date etc refers data dictionary and contain information are example of meta data. Data definition language also allows defining storage structure and access methods for database system, such special set of DDL statement called data storage and definition language.

### **Data Manipulation Language (DML)**

Data manipulation language allow database user to access (query) and manipulate data. That is, DML is responsible for

- Create new information in the database

- Read information from the database
- Update information in the database
- Delete information in the database

DML established communication between user and database.

There are two types of DML

- (a) Procedural DML: user required to specify what data are needed and how they get those data.
- (b) Nonprocedural (Declarative) DML: user only required to what data needed without specifying how to get those data.

Declarative DMLs are usually easier to learn and use than procedural DMLs. However, since a user does not have to specify how to get data, the database system has to figure out an efficient means of accessing data. The DML component of SQL is nonprocedural. A query is statement requesting the retrieval of information. Special set of DML which only use to retrieve information from database called query language.

Example:

Select customer\_name

    from customer

where customer.customer\_id='c001'

This query retrieves those rows from table customer where the customer\_id=c01.

## Query Processing:

The steps of query processing are:

- Parsing and translation
- Optimization
- Evaluation

The first step that must be taken in query processing is to translate a given query into its internal form. In generating the internal form of the query, the parser checks the syntax of the users query, verifies that the relation names appearing in the query are names of the relations in the database. The system constructs the parse tree and then generates the relational algebra expression.

The given query can be computed in various ways. For example:

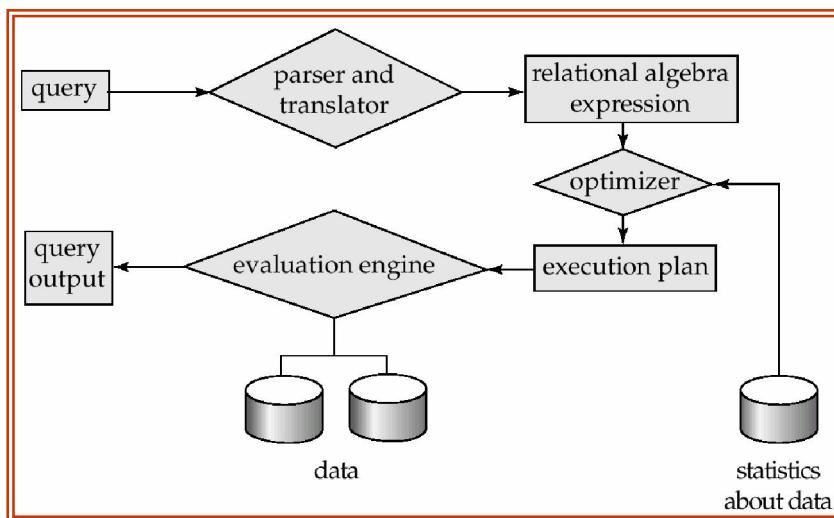
Select *balance* from *account*

Where *balance*<4000

This query can be translated into either of the following relational-algebra expressions.

- $\text{balance}(\text{balance} < 4000(\text{account}))$
- $\text{balance} < 4000(\text{balance}(\text{account}))$

the same query can be executed with different algorithms. The sequence of primitive operations that can be used to evaluate query is a query execution or query evaluation plan. Optimizer evaluates all the query plan and then finds out the cost of each query evaluation plan and selects the appropriate plan. Finally evaluation engine executes the plan and gives the output. The figure below shows the query processing steps.



**Fig: query processing steps**

### Database Users:

The various users of the database are discussed below:

1. naïve users:

Naïve users are the simple users who just uses the application that have been built previously. For example, a customer who uses ATM simply invokes the program which checks his username, password and balance: and finally allows to withdraw certain amount from his account.

2. application programmer:

These are the computer professionals who write application programs. They build user interfaces to interact with the database and hence it makes naïve users easy. Application programmers uses many application development tools for the quick development of the application.

3. sophisticated users:

Sophisticated users do not interact with the database through the application programmers since the interest of such users is vast and hence they generate their own query in a database query language. Analyst who submit queries to explore data in the database fall in this category.

4. specialized users:

Specialized users are the sophisticated users. These are highly advanced users and write specialized database applications. Scientists, researchers fall in this category.

### Database Administrators:

The person who has control over both data and the program that accesses those data are called **database administrator(DBA)**. The functions of database administrator are:

- schema definition:

The DBA creates the original structure of the database by using DDL.

- schema and physical organization modification:

The changes needed in any organization is analyzed and then appropriate change is made in the database schema by the DBA.

- granting of authorization:

By granting different types of the authorization, DBA can regulate which part of the database various users can access. The authorization information is kept in a special system structure that the database system consults whenever someone attempts to access the data in the system.

- runtime maintenance:

DBA periodically backup the database, ensures that enough free disk space is available for normal operations.

### **1.0 Database Management System**

Data are the raw facts that can be found after some experiment, observation or experience. Data itself do not provide any meaning but after processing it becomes information. The collection of related data organized in some specific manner is known as database. The database, its processing methods and the set of rules and conditions to be followed; collectively known as database management system (DBMS). Here, related data refers logically consistent facts of the real world. Random collection of data can not consider database. The primary goal of DBMS is to store and manage data both conveniently and efficiently. Database systems are generally designed to manage large volume of information. Management of data involves defining structure for storage of information and providing mechanisms for manipulation of information.

DBMS can also define as a general purpose software system that enables user to create, maintain and manipulate database. It provides fast and convenient access to information from data stored in database. DBMS interfaces with application programs so data contained in database can be accessed by multiple applications and users. Some popular DBMS softwares are: Oracle, SQL – Server, IBM-DB2, MySQL, MS Access, Sybase etc.

Some application areas of database system are:

- Banking: customer and their account info
- Airlines: reservations and schedules info
- Universities: student info, grades etc.
- Credit card transactions: for purchases on credit cards and generation of statements.
- Telecommunications: record of calls made
- Finance: for storing information about holding, sales and purchases etc.
- Sales: for customer, product and purchase information.
- Manufacturing: for management of supply chain.
- Human resources: for information about employee

### **1.2 Purpose of Database System**

Traditionally, file processing system was used to manage information. It stores data in various files of different application programs to extract or insert data to appropriate file.

File processing system has several drawbacks due to which database management system is required. Database management system removes problems found in file processing system. Some major problems of file processing systems are:

#### **1. Data redundancy and inconsistency**

In file processing system, different programmer creates files and writes application programs to access it. After a long period of time files may exist with different formats and application programs may written in many different programming languages. Moreover, same information may be duplicated in several files. We have to pay for higher storage and access cost for such redundancy. It may leads database in inconsistent state because update made in one file may reflected in one file but it may not reflected in another files where same information exist in another files.

#### **2. Difficulty in accessing data**

In file processing system, we can not easily access required data stored in particular file. For each new task we have to write a new application program. File processing system can not allow data to be retrieve in convenient and efficient manner.

## **1.0 |ntroduction**

---

### **3. Data isolation**

Since data are scatter in different files and data may stored in different format, so it is difficult to write program to retrieve appropriate data.

### **4. Integrity problem**

In database, we required to enforce certain type consistency constraints to ensure the database correctness or to enforce certain business rules. It is in fact called integrity constraints (e.g. account balance  $> 0$ ), integrity of database need not to be violated. In file processing system, integrity constraint becomes the part of application program. Programmer need to write appropriate code to enforce it. When new constraints are required to add or change existing one, it is difficult to change program to enforce it.

### **5. Atomicity problem**

Failures may lead database in an inconsistent state with partial updates. For example, failure occurs while transferring fund from account A to B. There would be the case that certain amount from account A is retrieved and it is updated but failure occurs just before it is deposited to account B, such case may lead database in inconsistent state.

### **6. Concurrent access problem**

Concurrent accessed increase the overall performance of system providing fast response time but uncontrolled concurrent accesses can lead inconsistencies in system. File processing system allow concurrent access but it is unable to coordinate different application programs so database may lead in inconsistent state. E.g. two people reading a balance and updating it at the same time

### **7. Security problems**

Since file processing system consist large no. of application programs and it is added in ad hoc manner. So it is difficult to enforce security to each application to allow accessing only part of data/database for individual database users.

## **1.3 Data Abstraction**

Data abstraction in database system is a mechanism to hide complexity of database. It allow database system to provides abstract view to database user. It hides how data are actually stored and maintain in database. Data abstraction simplifies users' interactions with the system.

There are three level of abstraction

### **Physical level**

It is a lowest level of abstraction. It describes how data are actually stored in database. It describes complex low level data structures in detail.

### **Logical Level**

This is a next highest level of abstraction. It describes what data are stored in database and what relationship exists among them. It describes entire database relatively in a simple structure. The user in logical level needs not to aware the complexity of physical level structure.

## 1.0 Introduction

### **View Level**

It is the highest level of abstraction. It describes only part of the entire database. It simplifies interaction with the system. It allows database system to provide many views for the same database. That is it allows each user/application to get different perspective of the database.

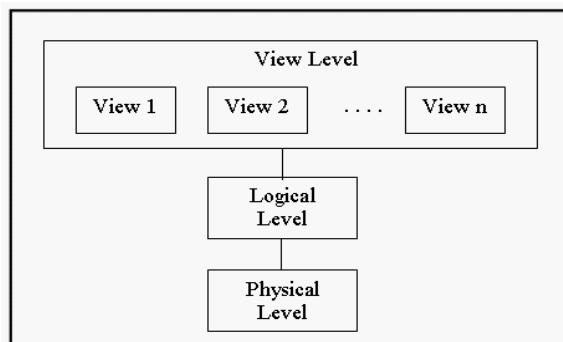


Figure1.3 Three level of abstraction

*Example:*

view level

- CS Majors
- Math Majors

logical level: entire database schema

- Courses (CourseNo,CourseName,Credits,Dept)
- Student (StudentID,Lname,Fname,Level,Major)
- Grade (StudentID,CourseNo,mark)

physical level:

- how these tables are stored, how many bytes it required etc.

## **1.4 Data Models**

Data models describe the underlying structure of database. It is a conceptual tool for describing data, relationship among data, data semantics and consistency constraints. There are several data models which can be group into three categories.

- (a) Object-based Logical Models.
- (b) Record-based Logical Models.
- (c) Physical Data Models.

### **1.4.1 Object-based Logical Models**

Object based logical model describe data at the logical and view levels. It has flexible structuring capabilities. It allows to specify data constraints explicitly. Under object-based logical model there are sever data models

- Entity-relationship model
- Object-oriented model

## 1.0 Introduction

### **1.4.1.1 Entity Relationship Model**

E-R model describes the design of database in terms of entities and relationship among them. An entity is a “thing” or “object” in real world that are distinguishable from other objects. An entity is described by a set of attributes.

For example

- Attributes account\_number and balance may describe entity “account”.
- Attributes customer\_id, customer\_name, customer\_city may describe entity “customer”.

A relationship is an association among several entities. For example, a depositor relationship associates a customer with each account he or she has.

The set of all entities of same type called entity set and similarly set of all relationship of the same type called relationship set.

E-R model graphically express overall logical structure of a database by an E-R diagram. Components of E-R diagram are as follows

rectangles: represent entity sets

ellipses: represent attributes

diamonds: represent relationships among entity sets

lines: link attributes to entity sets and entity sets to relationships

*Example:*

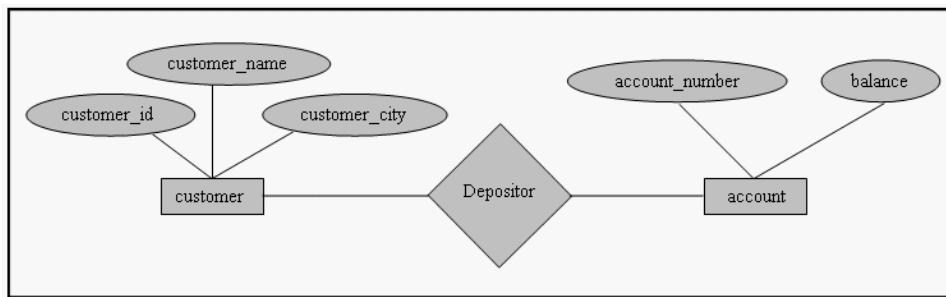


Figure 1.4.1.1.1 Sample E-R Diagram

Beside entities and relationship among them, E-R model has a capability to enforce constraints, mapping cardinalities which tell no. of entities to which another entity can be associated via relationship set. If each account must belong to only one customer, E-R model can express it. We discuss mapping cardinalities in detail in next chapter.

### **1.4.1.2 Object oriented model**

Object oriented data model is extension to E-R model with the notion of encapsulation, methods (functions) and object identity. It is based on collection of objects, like the E-R model. An object contains values stored in instance variables within the object. These values are themselves objects. That is, objects can contain objects to an arbitrarily deep level of nesting. An object also contains bodies of code that operate on the object. These bodies of code are

## 1.0 Introduction

called methods. Objects that contain the same types of values and the same methods are grouped into classes.

The only way in which one object can access the data of another object is by invoking the method of that other object. This is called sending a message to the object. Internal parts of the object, the instance variables and method code, are not visible externally.

*Example:*

Consider an object representing a bank account.

- The object may contain instance variables account\_number and balance.
- The object may contain a method pay-interest which adds interest to the balance.

Unlike entities in the E-R model, each object has its own unique identity. It is independent to the values it contains. Two objects containing the same values are distinct. Distinction is maintained in physical level by assigning distinct object identifier.

### **1.4.2 Record-based Logical Models**

As object based base logical model, record-based logical model also describes data at logical and view level. But it describes logical structure of database in more detail for implementation point of view. It describes database structure in terms of fixed-format records of different types. Each table contains records of a particular type. And each record type defines fixed number of fields or attributes. Each field is usually of a fixed length.

There are several languages which are used to express database queries and updates.

The three most widely-accepted models under record-based logical models are:

- Relational model
- Network model
- Hierarchical

#### **1.4.2.1 The Relational Model**

Relational model describes database design by a collection of tables (relations). It represents both data and their relationships among those data. Each table consist number of columns (attributes) with unique names. It is a most widely used data model. Relational model is lower level abstraction than E-R model. Database model are often carried out in E-R model and then translated into relational mode.

*Example:*

Previous describe E-R model can be express in relational model as follows

customer_id	customer_name	customer_city
C01	X	A
C02	Y	B
C03	Z	A
C04	X	A

(a) Customer relation

account_number	balance
A1	200
A2	300

customer_id	account_number
C01	A1
C02	A2
C03	A3

## 1.0 Introduction

A3	500
A4	500

(b) Account relation

(c) Depositor relation

Figure 1.4.2.1.1 Sample Relational database

### **1.4.2.2 The Network Model**

In network model, data are represented by the set of records and relationships among data are represented by links.

*Example:*

The above relational model can be express in network model as follows

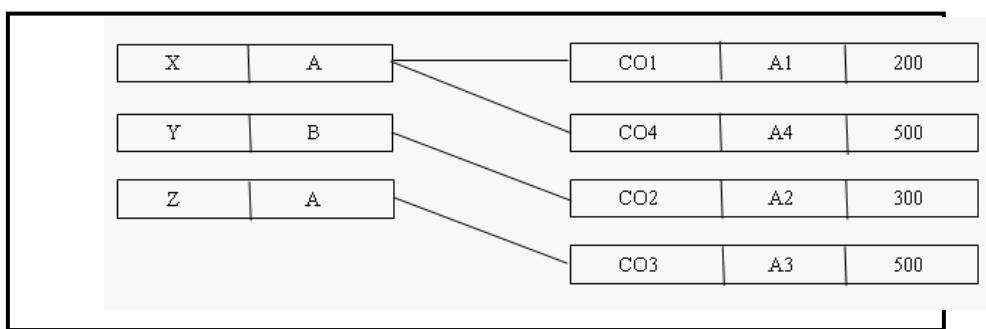


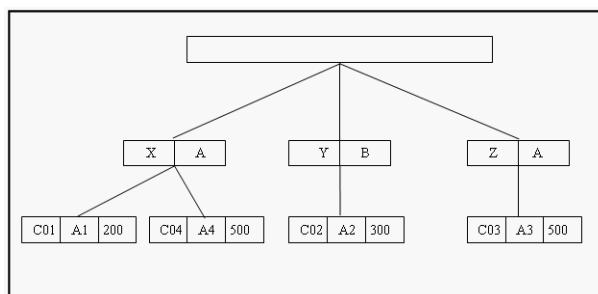
Figure 1.4.2.2.1 Sample Network Model

### **1.4.2.3 Hierarchical Model**

Hierarchical model is also represents data by a set of records but records are organized in hierarchical or order structure and database is a collection of such disjoint trees. The nodes of the tree represent record types. Hierarchical tree consists one root record type along with zero more occurrences of its dependent subtree and each dependent subtree is again hierarchical. In hierarchical model, no dependent record can occur without its parent record. Furthermore, no dependent record may be connected to more than one parent record.

*Example:*

The above network model can express in hierarchical model as follows



## **1.0 |ntroduction**

---

Figure 1.4.2.3.1 Sample Hierarchical Model

### **1.4.3 Physical Data Models**

Physical data models are used to describe data at the lowest level. Unifying model and frame memory are follows under physical data model. Here in this section we are not cover physical data model.

## **1.5 Instance and Schemas**

Database change over time as information is inserted, deleted and updated. The collection of information stored in database at a particular moment called an instance of the database. The overall design of the database is called database schema. Schemas are change infrequently, if at all.

According to the level of abstraction schema are divided into physical schema, logical schema and subschemas. The physical schema describes the database design at the physical level. Logical schema describes database design at the logical level. Database system may have several schemas at the view level, it is called subschemas (can be query), it describes different views of database.

Logical schema is more important for the development of application programs. Programmer constructs applications by using logical schema. The physical schema is hidden under the logical schema and it can change without affecting application programs.

## **1.6 Data Independence**

Data independence is an ability to modify a schema definition in one level without affecting scheme definition in higher level. There are two types of data independence.

### *Physical data independence*

It is an ability to modify the physical scheme without causing application programs to be rewritten

Modification at this level usually required for performance improvement reason.

### *Logical data independence*

It is an ability to modify the conceptual/logical scheme without causing application programs to be rewritten. Logical scheme needs to modify if we required to modify logical structure of database. Logical data independence is harder to achieve since application programs are usually dependent on logical structure of the data.

## **1.7 Database Languages**

Database system provides two languages

- (a) Data Definition Language and
- (b) Data Manipulation Language

## 1.0 |ntroduction

But in practice, data definition language and data manipulation language are not separate languages.

### **1.7.1 Data Definition Language (DDL)**

Data definition language used to specify database scheme. For example, following DDL statement in SQL defines account relation.

```
create table account
(
    account_no  char(2),
    balance      integer
)
```

The execution of above DDL statement creates table account. Moreover, it updates special set of tables called data dictionary or data directory. Data dictionary contains meta data, that is data about data. For example table containing tables' information like table name, owner, created date, modified date etc refers data dictionary and contain information are example of meta data.

Data definition language also allows to define storage structure and access methods for database system, such special set of DDL statement called *data storage and definition language*.

### **1.7.2 Data Manipulation Language (DML)**

Data manipulation language allow database user to access (query) and manipulate data. That is, DML is responsible for

- retrieval of information from the database
- insertion of new information into the database
- deletion of information in the database
- modification of information in the database

DML established communication between user and database.

There are two types of DML

- (a) Procedural DML: user required to specify what data are needed and how they get those data.
- (b) Nonprocedural (Declarative) DML: user only required to what data needed without specifying how to get those data.

Declarative DMLs are usually easier to learn and use than procedural DMLs. However, since a user does not have to specify how to get data, the database system has to figure out an efficient means of accessing data. The DML component of SQL is nonprocedural.

A query is statement requesting the retrieval of information. Special set of DML which only use to retrieve information from database called *query language*.

*Example:*

```
Select customer_name
      from customer
     where customer.customer_id='c001'
```

## **1.0 |ntroduction**

---

This query retrieves those rows from table customer where the customer\_id=c01.

### **1.8 Database Manager**

The 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 submitted to the system. Since database required lots of storage space so it must be stored on disks. Data need to moved between disk and main memory as needed.

Since the goal of database system is to simplify and facilitate access to data providing optimal performance as far as possible. So the database manager module is responsible for

- Interaction with the file manager: responsible to translate DML statements into low-level file system commands for storing, retrieving and updating data in the database.
- Integrity enforcement: responsible to check any updates in the database do not violate consistency constraints.(e.g. no bank account balance below \$25).
- Security enforcement: responsible to ensure that users only have access to information they are permitted to see.
- Backup and recovery: Detecting failures due to power failure, disk crash, software errors, etc., and restoring the database to its state before the failure.
- Concurrency control: responsible to preserving data consistency when there are concurrent users.

### **1.9 Database Administrator**

The database administrator is a person having central control over data and programs accessing that data. Database administrator has the following responsibility:

- Schema definition: responsible for the creation of original database schema. So DBA is responsible to write data definition statements in DDL.
- Storage structure and access method definition: DBA is responsible to write a set of definitions to define storage and access method using storage and access.
- Schema and physical organization modification: DBA is responsible for modification of schema and to reflect the changes in schema or to improve the performance physical organization may need to be change.
- Granting authorization for data access: DBA is responsible to grant different types of authorization for data access to various users.
- Routine maintenance:
  - Periodically backing up the database ensuring enough free disk space available for normal operations and upgrading disk space as required.
  - Monitoring jobs running on the database and ensuring that performance is not degraded too much.

### **1.10 Database Users**

There are four different types of database users, they are differentiated according to their interaction with the system. Moreover, there are different types of user interfaces for different types of users.

## **1.0 |ntroduction**

---

### **(a) Naïve Users:**

Naïve users are unsophisticated users who interact with the system by invoking one of the application programs that are already written. For example, banks teller who needs to transfer fund from one account to another invoking a program called transfer. This program asks the teller for the amount of money to be transferred, and account to which the money is to be transferred.

The typical user interface for the native user is a form interface, where user can fill appropriate fields of the form. Native users may also simply read reports generated from the database.

### **(b) application programmers:**

Application programmers are computer professional who write application programs. Application programmers may choose any programming tool to develop user interfaces. They can also used RAD tools that enable an application programmer to construct forms and reports without writing the program. There are also special type of programming languages that combine imperative control structures (e.g. for loops, while loops and if-then-else statements) with the statements of data manipulation language. These languages are sometimes called fourth generation languages. It often includes special features to facilitate the generation of forms and display data on the screen. Most major commercial database system includes a fourth generation language.

### **(c) sophisticated users:**

Sophisticated user interact with system without writing programs but they requests by writing queries in database using DML query language. This query goes to query processor and it converted into instructions for the database manager module.

### **(d) Specialized users:**

Specialized users are responsible to write special database application programs it could be computer-aided design systems, knowledge based and expert systems that store data with complex data types (e.g. graphics data, audio/video data).

## **1.11 Overall System Structure**

The functional component of the database system is divided into storage manager and query processor component.

### **1.11.1 Storage Manager**

Storage manager is a program module that provides interface between the low level data stored in the database and the application programs and queries submitted to the system. The storage manager is responsible for the interaction with the file manager. The storage manager various DML statements into low level file system command. And it is responsible for storing, retrieving, and updating data in the database.

## **1.0 |ntroduction**

---

Storage manager consist following components:

*Authorization and integrity manager:* responsible to ensure integrity constraint does not violate and checks the authority of users to access data.

*Transaction Manager:* responsible to ensure database remain inconsistent state even system failure occurs. It is also responsible to manage concurrent transactions so that they could not conflict, which also helps to ensure consistency of database.

*File Manager:* responsible to manage the allocation of space on disk storage and the data structures used to represent information stored on disk.

*Buffer Manager:* responsible for fetching data from disk storage into main memory, and decides what data to cache in main memory.

The storage manager implements several data structure for physical system implementation:

*Data files:* stores database itself,

*Data dictionary:* stores meta data about structure of database, in particular schema of database.

*Indices:* provides fast access to data items that holds particular values.

### **1.11.2 Query processor**

The query processor is responsible to simplify and facilitate access data. It is responsible to translate updates and queries written in nonprocedural language at the logical level, into an efficient sequence of operations at the physical level.

The query processor component includes the following components:

*DDL interpreter:* responsible to interprets DDL statements and records the definitions in the data dictionary.

*DML Compiler:* responsible to translate DML statements in a query language into low level instructions that query evaluation engine understands. Query is generally translated into no. of alternative evaluation plans that produce the same result. It is also responsible for query optimization; it required to select the lowest cost evaluation plan among the alternatives

*Query evaluation:* responsible to execute low level instruction generated by DML compiler.

## 1.0 |ntroduction

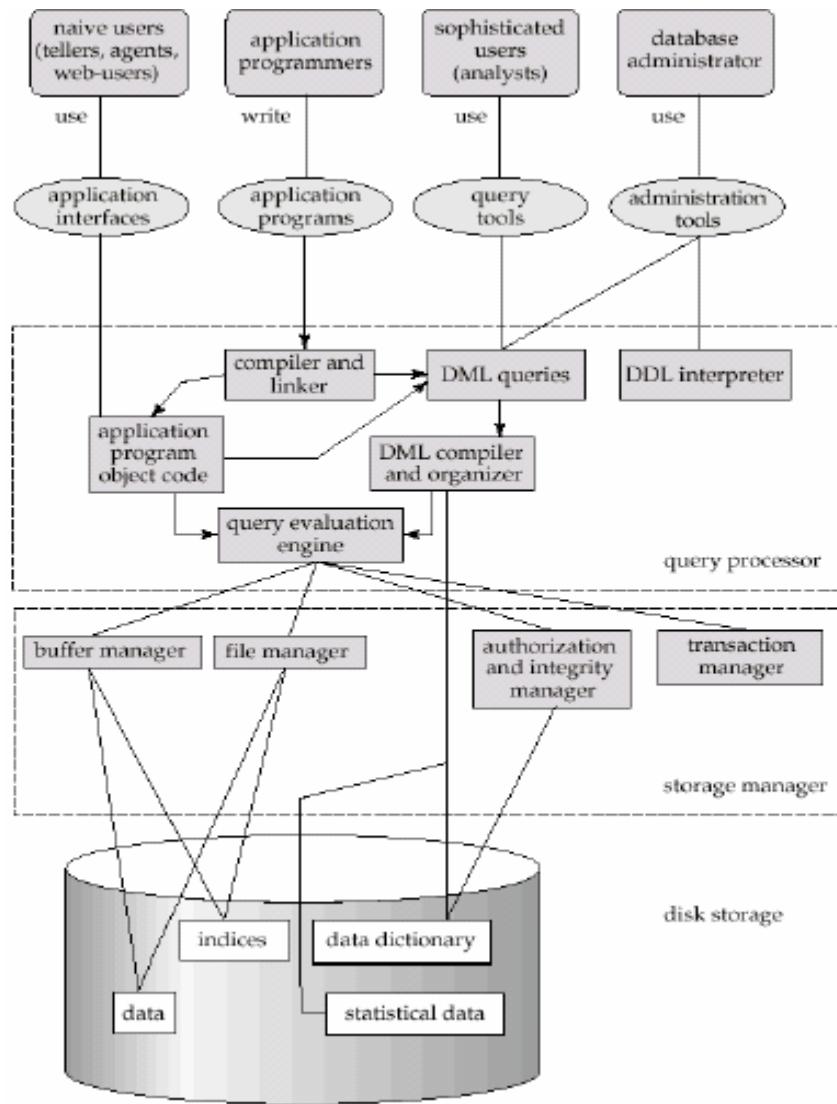


Figure 1.11.1 Overall database system structure

## 1.12 Advantages and disadvantages of DBMS

### 1.12.1 Advantages of DBMS

*Data independence:* DBMS provides abstract view of data. Application programs are independent from details of data representation and storage.

*Efficient data access:* DBMS provides variety of sophisticated techniques to store and retrieve data efficiently.

*Data integrity and security:* DBMS allow to enforce integrity constraints on data. For example before inserting salary information for an employee, DBMS can enforce integrity constraint to

## **1.0 |ntroduction**

---

check salary is not exceeded department budget. DBMS can also enforce access controls, what data is visible to what class of users.

*Data administration:* DBMS provides centralized administration of data. It is appropriate when several no. of database user shares data. It improves the overall performance of database system.

*Concurrent access and crash recovery:* DBMS has a capability manage concurrent access. It schedules concurrent access to the data in such a manner that user fell data is being accessed by only one user at a time. Moreover, DBMS protects users from the effects of system failures.

*Reduced application development time:* since DBMS supports many important functions that are common to many applications accessing data stored in database. It provides high level interface to data and facilitates quick development of applications.

### **1.12.2 Disadvantage of DBMS**

- Complex architecture of DBMS software
- DBMS software cost
- Since DBMS is optimized certain kind of workloads (e.g. answering complex queries or handling many concurrent requests) its performance may not appropriate for certain specialized applications.
- Abstract view of data presented by DBMS may not match for certain applications. For example, relational databases does not supports flexible analysis of text data
- If specialized performance or data manipulation requirements are central to an application, DBMS is not appropriate for such application. The added benefits of a DBMS (e.g. flexible querying, security, concurrent access and crash recovery) may not require for applications.

## 2. Entity-Relationship Model

---

### 2. Entity-Relationship Model

Entity-relationship model describes data involves in real world in terms of object and their relationships. It is widely used for initial database design. It describes overall structure of database. E-R model is in fact, semantic data model which describes the meaning of data. It has a capability to map the meanings and interactions of real world objects on to the conceptual schema.

#### 2.1 Entity and Entity Sets

An entity is a "thing" or "object" in the real world that is distinguishable from another object. For example:

- Specific customer , Particular course in university

Entities can be described by a set of properties called *attributes*. For example: customer\_id, customer\_name, customer\_address are attributes for entity customer. Similarly, course\_id, course\_name are attributes for entity course.

An entity set is a set of entities of the same type that share the same properties. Entities of an entity set has same set of attributes.

For example

- Set of all customer
- Set of all courses in an university

C01	X	Kathmandu
C02	Y	Lalitpur
C03	Z	Kathmandu

Figure 2.1.1 instance of customer entity set

#### 2.2 Attributes

In simple, attribute is descriptive property of entity set. Set of attributes describes entity set. For example

```
customer = (customer-id, customer-name, customer-city)
account=( account_number, balance)
loan = (loan_number, amount)
```

The set of permitted values for an attribute called domain of that attribute.

For example

- set of all text strings of certain length for customer\_name is domain of that attribute.
- set of all strings like "A-n" where n is a positive integer for account\_number is domain of that attribute.

Formally, an attribute is a function which maps an entity set into a domain. Every entity is described by a set of (attribute, data value) pairs. There is one pair for each attribute of the entity set. For example: particular customer entity in customer entity set can describe by the set of pairs (customer\_id, c01), (customer\_name, x) and, (customer\_city,Kathmandu).

## 2. Entity-Relationship Model

### 2.2.1 Types of Attributes

#### ***Simple and Composite attribute***

Attribute which can not be divide into subparts (i.e. into other attributes) called simple attribute. For example, customer\_id in customer entity set is simple attribute, since it can not divide into sub attributes.

Attribute that can further divide into subparts called composite attribute. For example, customer\_name in customer entity set is composite attribute since it can be divided into sub attributes: customer\_fname, customer\_mname and customer\_lname. Composite attributes helps to group related attributes, which makes modeling clearer.

#### ***Single-valued and Multivalued attributes***

Attribute that can take only one value in every entry called singled-valued attribute. For example, attribute customer\_name in customer entity set is single-valued attribute since it can not contain more than one customer name in any entry. An attribute that can take more than one values in any entry called multivalued attribute. For example, in a customer entity set attribute customer\_phonenumber is multivalued attribute since customer may have zero or one or several phone number.

#### ***Stored and Derived attribute***

Attribute whose values can be derived from the values of other related attributes or entities called derived attribute. For example, in customer entity set, attribute age is derived attribute if customer entity set has attribute date\_of\_birth. We can derive age of customer from date\_of\_birth and current\_date. Here the attribute date\_of\_birth is stored attribute and the attribute age is derived attribute. The value of derived attribute is not stored, it is computed when required.

## 2.3 Relationships and Relationship Sets

A relationship is an association among two or more entities. A relationship set is a set of relationship of same type.

Formally, if  $E_1, E_2, \dots, E_n$  ( $n \geq 2$ ) are entity sets then a relationship set  $R$  is a subset of  
 $\{(e_1, e_2, \dots, e_n) | e_1 \in E_1, e_2 \in E_2, \dots, e_n \in E_n\}$

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

Example: For two entity sets customer and account, we can define relationship set depositor which associates each customer to their corresponding account he/she has.

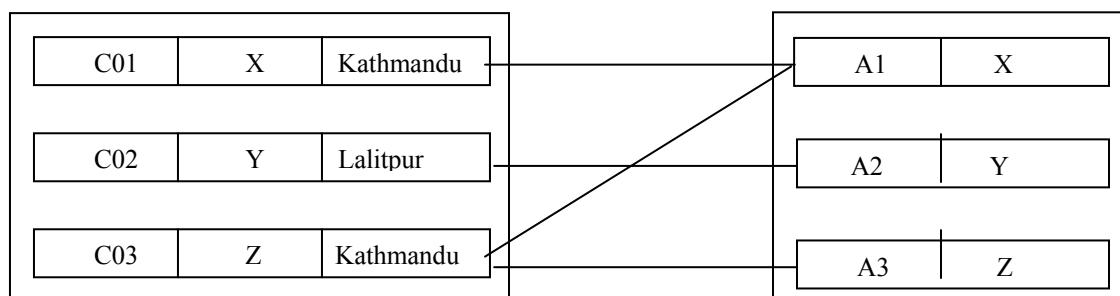


Figure: instance of depositor relationship set

## 2. Entity-Relationship Model

The relationship set may have descriptive attributes, which generally used to record information about the relationship. For example: in depositor relationship with entity sets customer and account we can associate attribute access\_date to the relationship set depositor to specify the most recent date on which a customer accessed an account.

### ***Recursive relationship set***

The function that an entity plays in relationship called that entity's role. In relationship set, usually roles of entity's are not specified but it is useful if meaning of relationship need to clear. In such case, same entity may participate in relationship set more than once with different roles. This type of relationship set called recursive relationship set.

Example:

Let us consider entity set employee that records information about all employees of bank. We may have relationship set "work-for" as recursive relationship set because same employee may play worker as well as manager.

### ***Binary and ternary relationship set***

Relationship set that involves only two entity sets known as binary-relationship set. For example: depositor relationship set is a binary relationship set where relationship set involves only two entity set "customer" and "account".

Most relationship sets in database system are binary. However relationship set may involve in more than two entity sets. Relationship set that involves three entity sets known as ternary relationship. For example: the relationship set "work-on" among employee, branch and job is example of ternary relationship.

The no. of entity sets that participate in relationship set refers degree of relationship set. Here degree of ternary relationship is 3.

## 2.4 Constraints in E-R Model

E-R model has a capability to enforce constraints. Two most important type of constraints in E-R model are-

Mapping Cardinalities (Cardinality ratio), Participation Constraints

### ***Mapping Cardinalities***

Mapping Cardinalities describes no. of entities to which another entity can be associated via relationship set. Mapping cardinalities are most useful in describing binary relationship sets but it can also describe relationship sets that involve more than two entity sets. For binary relationship set between entity set A and B mapping cardinality must one of the following.

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

**One to many:** An entity in A is associated with zero or more entities in B but entity in B can be associated with at most one entity in A.

**Many to one:** An entity in A is associated with at most one entity in B but an entity in B can be associated with zero or more entities in A.

## 2. Entity-Relationship Model

**Many to many:** An entity in A is associated with zero or more entities in B, and an entity in B is associated with zero or more entities in A.

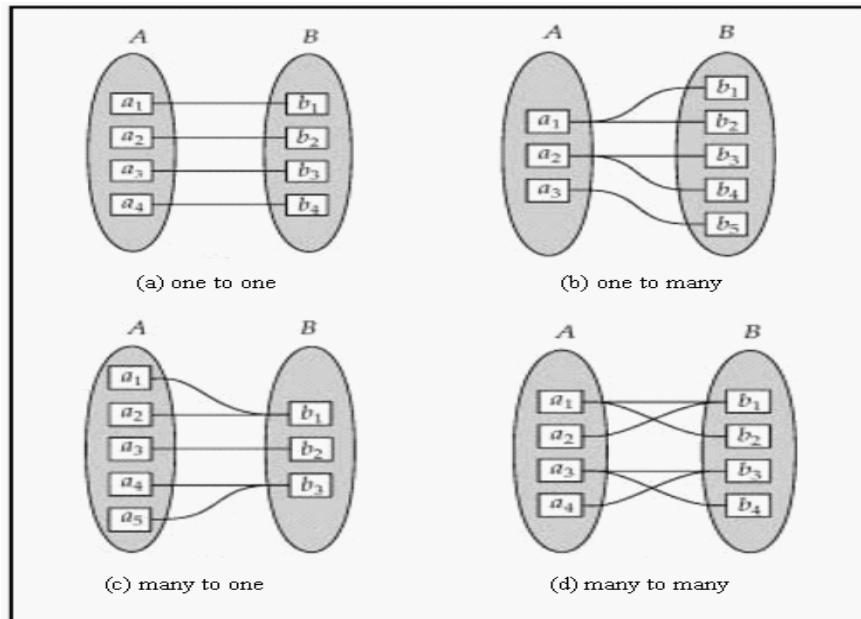


Figure: Mapping Cardinalities

The appropriate mapping cardinality for a particular relationship set depend upon real world situation that the relationship set is going to modeling. For example: in borrower relationship set, if loan can belong to any one customer and customer can have several loan then relationship set from customer to loan is one to many. If a loan can belong to several customer (loans taken jointly by several customers) then relationship set is many to many.

### ***Participation Constraints***

The participation of an entity set E in a relationship set R is said to be *total* if every entities in E participates in at least one relationship in R. If only some entities in E participate in relationship in R, then participation of entity set E in relationship set R is said to be *partial*. The participation of loan in the relationship set borrower is total but customer entity set in borrower relationship set is partial since not all customers necessarily take loan from bank, customer may also those who are only account holder. Such participation constraint can be express by E-R model. We will discuss it in later section.

## 2.5 Keys

The concept of key is important to distinguish one entity from another and one relationship from another relationship. In fact, values of attributes distinguish one entity from another entity. To distinguish one entity from another entity in entity set there must exist attribute/s whose values must not duplicate in entity set. It ensures no two entities in an entity set can exist with same values for all attributes.

## 2. Entity-Relationship Model

### Super key

A super key is a set of one or more attributes which uniquely identifies an entity in entity set. For example: in customer relation single attribute `customer_id` is sufficient to uniquely identify one customer entity to another. So `customer_id` is a superkey in a customer relation. Since combination of `customer_id` and `customer_name` can also uniquely identifies one customer entity to another. So combination of attributes `{customer_id, customer_name}` is also superkey in relation customer. But single attribute `customer_name` can not superkey in relation customer because customer name only can not uniquely identify one customer entity to another, there would be number of customers having same name.

The above example of supekey shows that superkey may contains extraneous attributes. That is, if K is superkey then any superset of K is superkey.

### Candidate key

The minimal superkey called candidate key. That is, candidate key is a superkey but its proper subset is not superkey. For example: `customer_id` is a candidate key in customer relation. Similarly `account_id` is a candidate key in account relation.

### Primary key

In a relation, it is possible that we can choose distinct set of attributes as a candidate key. For example: in customer we can choose single attribute `{customer_id}` or set attributes `{customer_name, customer_city}` as candidate key. Candidate key chosen by database designer for particular relation known as primary key.

## 2.6 Primary Keys for Relationship Sets

Suppose R is a relationship set involving entity sets  $E_1, E_2, \dots, E_n$ . Lets consider primary-key( $E_i$ ) is a set of attributes that form primary key in each entity sets  $E_i$ . ( $i=0,1, \dots, n$ ). Then set of attributes

Primary-key ( $E_1$ )  $\cup$  primary-key ( $E_2$ )  $\cup \dots \cup$  primary-key ( $E_n$ )  
describes individual relationship in relationship set R.

Since relationship set may also consist another attributes (e.g. descriptive attributes) so assume relationship set consist attributes `{a1, a2, ..., an}` then set of attributes

Primary-key ( $E_1$ )  $\cup$  primary-key ( $E_2$ )  $\cup \dots \cup$  primary-key ( $E_n$ )  $\cup \{a_1, a_2, \dots, a_n\}$   
also describes individual relationship in relationship set R.

In both of the above case, the set of attributes

Primary-key ( $E_1$ )  $\cup$  primary-key ( $E_2$ )  $\cup \dots \cup$  primary-key ( $E_n$ )  
form super key in relationship set R.

This indicates that primary keys of entity sets those involves in relationship set form super key in relationship set. For example: in relationship set depositor, `{customer_id, account_number}` are attributes and each attribute `customer_id` and `account_number` are primary key in relation customer and account respectively. If relationship set consist declarative attribute say `access_date` then it can also consider as attributes of relationship set depositor.

## 2. Entity-Relationship Model

The selection of primary key in relationship set depends up on mapping cardinality of that relationship set. To illustrate this, let us consider entity sets customer and account, and relationship set depositor with declarative attribute access\_date. Suppose that relationship set is many to many. Then primary key of depositor relationship set is combination of primary keys of customer and account. If customer can have only one account, that is, if relationship set is many to one from customer to account the primary key of depositor is simply primary key of customer (i.e customer\_id). Similarly, if each account is own by at most one customer, that is, relationship set is many to one from account to customer then primary key of depositor is simply the primary key of account (i.e. account\_id). If relationship set is one to one then we can choose either customer\_id or account\_id as primary key.

For non-binary relationship set where no cardinality constraints are define then superkey is only one possible candidate key. So it needs to be chosen as a primary key.

### 2.7 Entity Relationship Diagram (ERD)

We already discuss that E-R Diagram has a capability to describes overall logical structure of database graphically in brief. In this section we discuss E-R diagram capabilities in detail

Major components of E-R diagram are:

- Rectangles:** representing entity sets.
- Ellipses:** representing attributes.
- Diamonds:** representing relationship sets.
- Lines:** linking attributes to entity sets and entity sets to relationship sets.
- Double ellipses:** represents multivalued attributes.
- Dashed ellipses:** represents derived attributes
- Double lines:** represents total participation of entity in relationship sets
- Double rectangles:** represents weak entity sets (discuss in later section)
- Underline:** indicates primary key attributes

#### *Representation of relationship set in E-R Diagram*

The relationship set borrower having two entity sets customer and loan can be express as

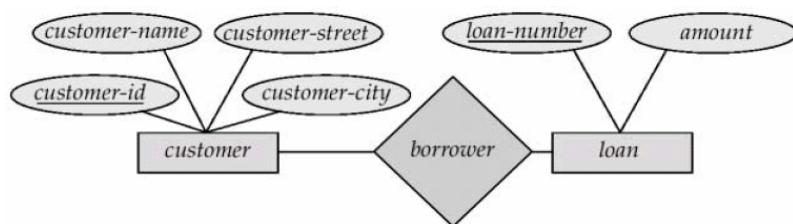


Figure: borrower relationship set in E-R Diagram

#### *Representation of mapping cardinalities of relationship set in E-R Diagram*

The directed lines in E-R diagram are used to specify mapping cardinalities of relationship sets. The directed line ( $\rightarrow$ ) tells "one," and an undirected line ( $-$ ) tells "many" between the relationship set and the entity set.

## 2. Entity-Relationship Model

### One-to-one relationship representation

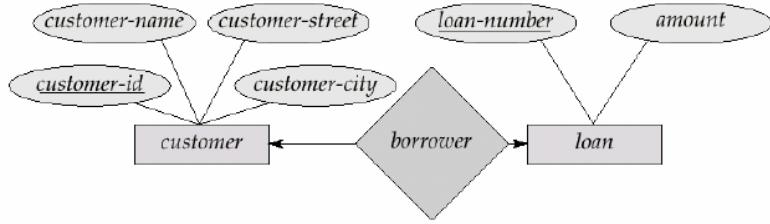


Figure: one to one relationship set in E-R Diagram

This indicates a customer is associated with at most one loan via the relationship borrower and a loan is associated with at most one customer via borrower.

### One to many relationship representation

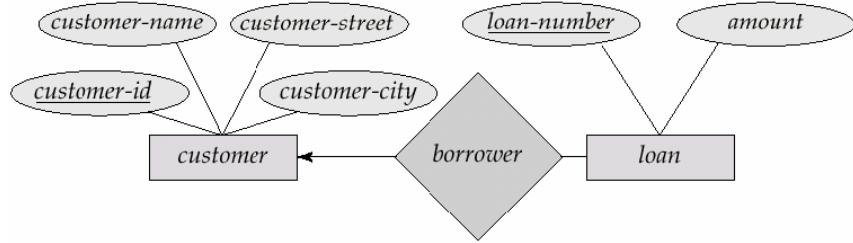


Figure: one to many relationship set in E-R Diagram

This indicates a loan is associated with at most one customer via borrower and a customer is associated with several (including 0) loans via borrower.

### Many to one representation

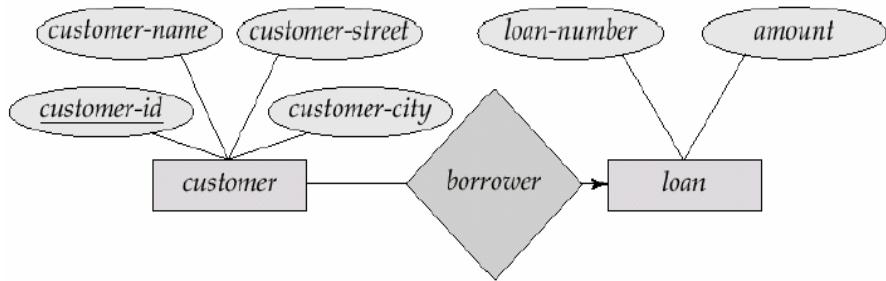


Figure: many to one relationship set in E-R Diagram

This indicates that a loan is associated with several (including 0) customers via borrower but a customer is associated with at most one loan via borrower.

## 2. Entity-Relationship Model

### Many to many relationship representation

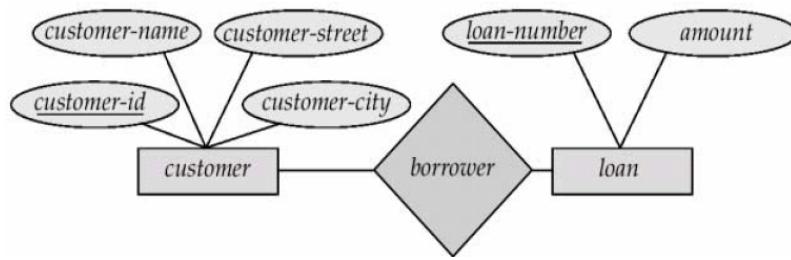


Figure: many to many relationship set in E-R Diagram

This indicates a customer is associated with several (including 0) loans via borrower and a loan is associated with several (including 0) customers via borrower

### Representation of Composite, Multivalued, and Derived Attributes in E-R Diagram

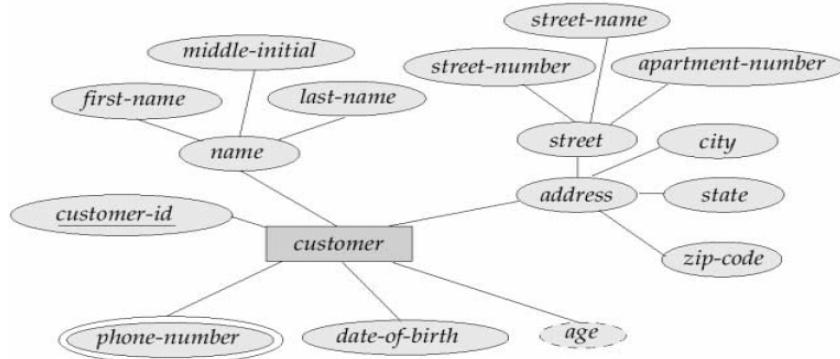


Figure: E-R Diagram with Composite, Multivalued, and Derived Attributes

Here attribute "name" is composite attribute, "phone\_number" is multivalued attribute and "age" is derived attribute.

### Representation of roles (recursive relationship set) in E-R Diagram

Roles in E-R diagrams are indicated by labeling the lines that connect diamonds to rectangles. Role labels are optional, and are used to clarify semantics of the relationship

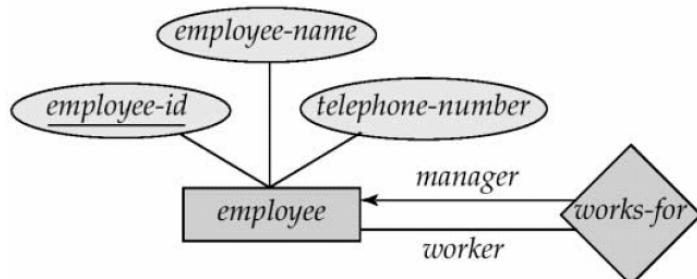


Figure: Roles in E-R Diagram

## 2. Entity-Relationship Model

Here relationship set “work-for” is recursive relationship set and the labels “manager” and “worker” are roles.

### ***Representation of non binary relationship Set in E-R Diagram***

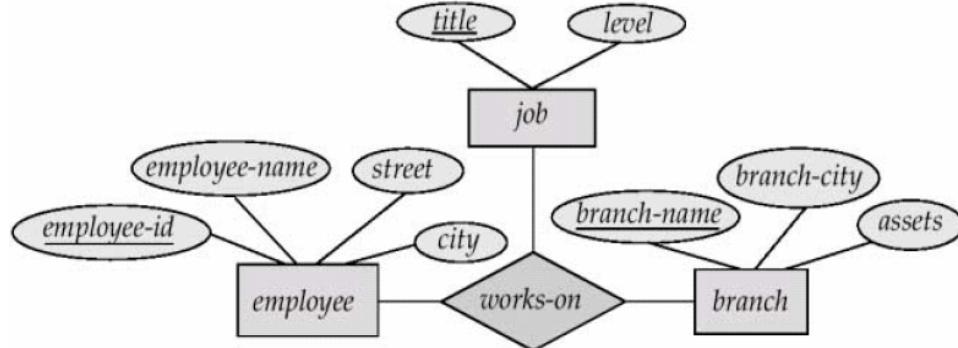


Figure: ternary relationship set in E-R Diagram

In ternary (or higher degree) relationship set there are one arrow to indicate a cardinality constraint. For example: an arrow from works-on to job indicates each employee works on at most one job at any branch. If there is more than one arrow, there are two ways of defining the meaning. Suppose a ternary relationship R between A, B and C with arrows to B and C this can be interoperate as

- each entity in A is associated with a unique entity in B and C or
- each pair of entities (A, B) is associated with a unique entity in C, and each pair of entities (A, C) is associated with a unique entity in B.

### ***Representation of participation constraints of entity set in relationship set***

In E-R diagram total participation of entity set in relationship set indicated by double line between that relationship set and entity set, single line indicates partial participation.

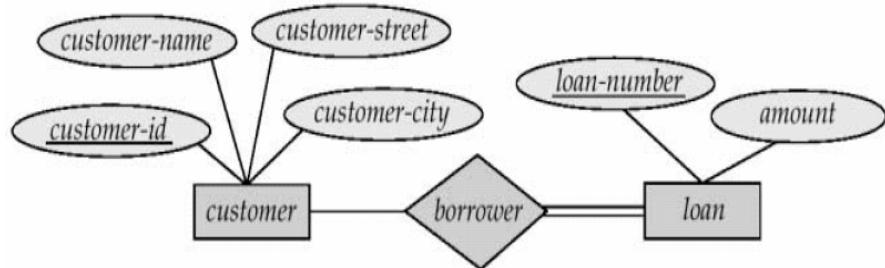


Figure: participation constraints in E-R Diagram

This indicates participation of *loan* in *borrower relationship* is total and participation of *customer* in *borrower relationship* is partial. That is, every loan must have a customer associated to it via borrower but all customers may not associate to loan.

## 2. Entity-Relationship Model

E-R diagram can also specify more complex constraints. It can specify cardinality limits. That is, it can specify how many no. of times each entity may participates in relationships in relationship set. An edge between an entity set and binary relationship can have an associated minimum and maximum cardinality in the form l..h, where l is the minimum and h is the maximum cardinality. A minimum value 1 indicates total participation of the entity set in the relationship set. A maximum value 1 indicates that the entity participates in at most one relationship, while maximum value \* indicates no limit. That is label 1..\* on an edge is equivalent to double line.

Example:

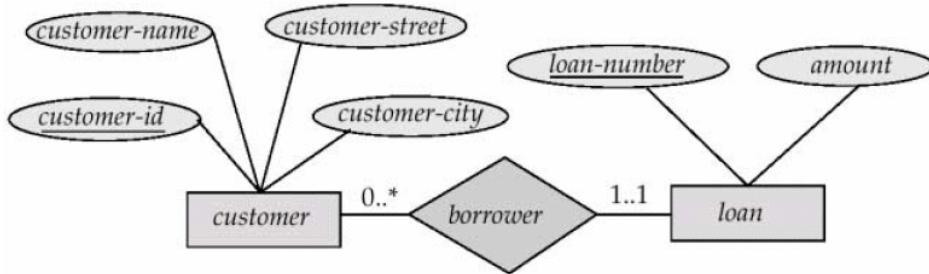


Figure: Cardinality limits on relationship sets

Here, edge between loan and borrower has cardinality limit 1..1. This indicates loan must have exactly one associated customer. The cardinality limit 0..\* on the edge from customer to borrower indicates that customer can have zero or more loans. Thus the relationship borrower is one to many from customer to loan and further the participation of loan in borrower is total.

### 2.8. Weak Entity Sets and their representation in E-R Diagram

Entity set that does not have primary key known as weak entity set and entity set that has a primary key known as strong entity set.

Let us consider entity set

Payment= (payment\_number,payment\_date,payment\_amount)

Here, payment numbers are typically sequence of numbers, starting from 1 and generated for each loan. Thus although each payment entry is distinct, payments for different loan may share the same payment number. Thus, this entity set does not have a primary key; it is a weak entity set

For weak entity set to be meaningful, it must be associated with another entity set called identifying or owner entity set. The relationship between weak entity set and identifying set known as identifying relationship. The identifying relationship is many to one from the weak entity set to identifying entity set, and participation of weak entity set in relationship is total.

In our example, identifying entity set for weak entity set payment is loan, and a relationship loan-payment associates payment entities with their corresponding loan entities in identifying relationship.

Although, a weak entity set does not have a primary key, it contains attribute or set of attributes that distinguishes all entities of a weak entity set called *discriminator of weak entity*

## 2. Entity-Relationship Model

*set or partial key.* In payment weak entity set payment\_number is a discriminator since for each loan, payment number uniquely identifies one single payment for that loan.

The primary key of weak entity set can be composed by combining primary key of identifying entity set and the weak entity set's discriminator. For weak entity set payment primary key is {loan\_number,payment\_number}, where loan\_number is primary key of identifying entity set loan, and payment\_number is discriminator of weak entity set payment.

In E-R diagram, weak entity set represented by double rectangles. The discriminator of a weak entity set is represented by underline attribute with a dashed line and double outlined diamond represents identifying relationship.

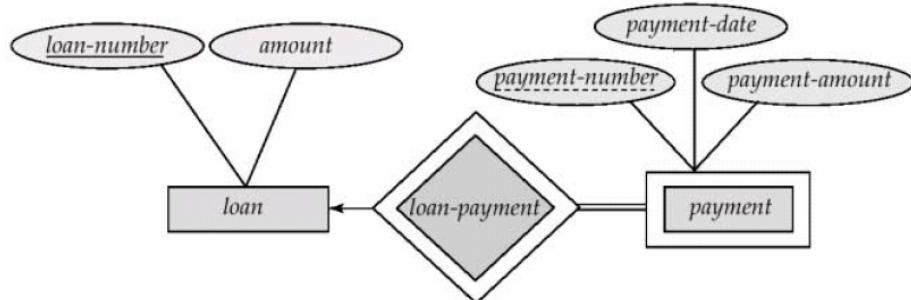


Figure: E-R diagram with weak entity set.

## 2. Entity-Relationship Model

### 2.9 Extended E-R Features

#### 2.9.1 Specialization

Specialization follows top down design approach. Entity sets are subgroups in distinct entity sets. For example entity set person with attributes name, street and city can further subgroup into two entities sets customer and employee. Each of these person types can be described by a set of attributes that includes all the attributes of entity set person plus all possible attributes of itself. For example, customer entity set can further be described by a set of attributes: customer\_id, enroll\_date etc. Similarly entity attributes can further be described by a set of attributes: employee\_id, salary etc. The process of sub groupings within an entity set is called specialization. We can apply specialization repeatedly to refine a design schema. For instance bank employees may be further classified into officer, teller or secretary.

In E-R diagram, specialization can be represented by a triangle component labeled ISA. The label ISA stands for "is a". For example customer is a person, officer is an employee etc. The ISA relationship also called super class-subclass relationship.

#### 2.9.2 Generalization

Generalization follows bottom-up approach in which multiple entity sets are synthesized into higher-level entity set on the basis of common features. For example, the database designer may have first identified a customer entity set with the attributes: name, street, city and customer\_id and employee entity set with the attributes name, street, city, employee\_id and salary. In both entities some attributes are common. These similarities between these two entities can be expressed by generalization.

During the course of database design or E-R schema for enterprise database designer may use both specialization and generalization process. Specialization and generalization in E-R diagram represent by a same way. The terms specialization and generalization are used interchangeably.

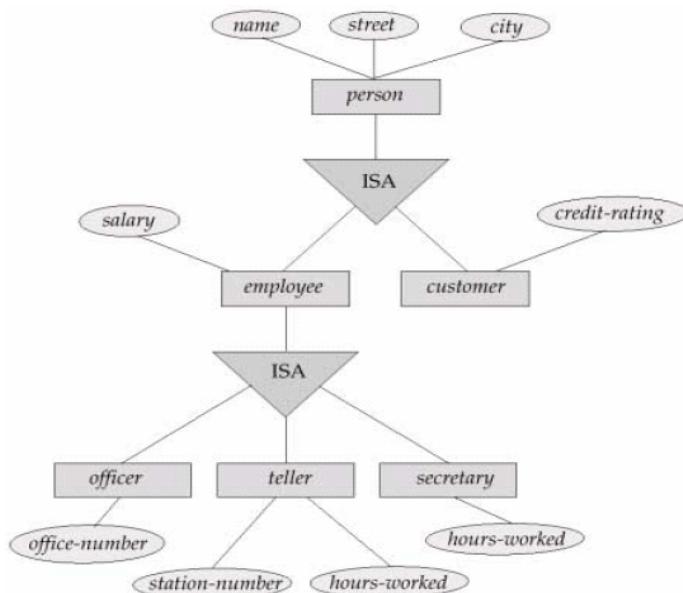


Figure: Specialization and generalization.

## 2. Entity-Relationship Model

### 2.9.3 Constraints on Generalization and Specialization

Database designer can enforce certain constraints in generalization and specialization. These constraints are as follows

(a) Constraint on which entities can be member of a given lower level entity set

Membership of lower-level entity can be express one of the following way

**Condition defined:**

Database designer can define condition to lower level entity set that must follows entities in higher level for member of lower level entity set. Suppose account is a higher level entity set with attribute account\_type. Assume that saving\_account and checking\_account are two lower level entity set. To specify which entities in account belongs which lower level entity set database designer can specify condition for each lower level entity set. For saving\_account entity set, database designer can enforce membership condition account\_type='saving account' and for checking\_account entity set, database designer can enforce membership condition account\_type='checking account'. This type of specialization/generalization known as *attribute defined*.

**User defined:**

It does not enforce any membership condition in lower-level entity set. Database user itself assigns entities into other entity set. For instance, we can assume that after 3 month of employment, bank employees are assign in one of the available workgroups.

(b) Constraint on whether or not entities may belong to more than one lower-level entity set within a single generalization.

To indicate this constraint lower level entity set may one of the following

**Disjoint:**

Disjointness constraints enforce an entity can belong to only one lower level entity set. In previous account example, entities in account can either belong to saving\_account or checking\_account, can not both. In E-R diagram it can be express by writing disjoint next to the ISA triangle.

**Overlapping**

In overlapping generalization/specialization, same entity may belong to more than one lower-level entity set. For example: employee may involve in more than one workgroups. Same people may customer as well as employee. Lower level entity overlap is default case.

(c) Completeness constraint: specifies whether or not an entity in the higher-level entity set must belong to at least one of the lower-level entity sets within a generalization/specialization.

This constraint may specify as follows

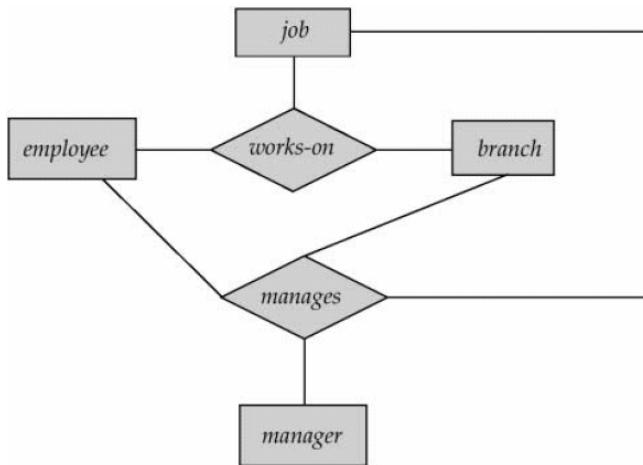
**Total generalization/specialization:** tells each higher level entities must belong to one of the lower-level entity sets. In E-R diagram total generalization/specialization specifies by using a double line to connect the box representing higher level entity set to the triangle symbol.

**Partial generalization/specialization:** tell not all entities in higher level need to belong to one of the lower-level entity sets. Partial generalization is default.

## 2. Entity-Relationship Model

### 2.10 Aggregation

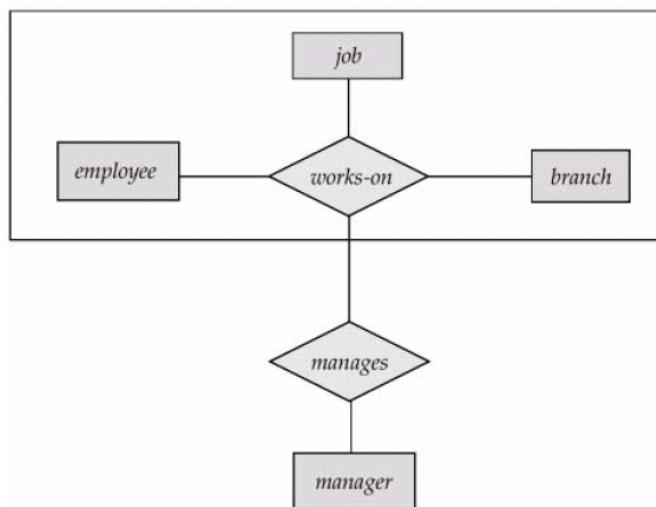
E-R model can not express relationship among relationship. To illustrate this, let us consider quaternary relationship manages among employee, branch, job and manager. Its main job is to record managers who manages particular job/task perform by particular employee at particular branch.



E-R diagram with redundant relationships.

This quaternary relationship is required since binary relationship between manager and employee can not represent required information. This E-R diagram is able to represent the required information but information are redundant since every employee, branch and job exist both relationship set "work-on" and "manages". Here aggregation is better to represent such information.

Aggregation is in fact an abstraction it treats relationships as higher level entities. In our example, it treats relationship set work-on (including entity set employee, branch and job) as entity set. So now we can create binary relationship set "manages" between work-on and manager. This removes redundant information.



E-R Diagram with aggregation

## **2. Entity-Relationship Model**

### **2.11 Design of an E-R Database Schema**

E-R data model provides much flexibility in designing database schema. Database designer can select wide range of alternatives. Database designer must make following decisions:

- Whether to use an attribute or entity set to represent an object.
- Whether a real-world concept is best to express by an entity set or a relationship set
- Whether to use a ternary relationship or pair of binary relationship.
- Whether to use strong or weak entity set.
- Whether to use generalization or specialization. Generalization/Specialization provides modularity in the design.
- Whether aggregation is better to use or not.

#### **2.11.1 Database Design Phase**

##### **(a) user specification requirements**

In the initial phase of database design, database designer need to characterize what data needs for database users and how the database is structured to fulfill these requirements. Database designer needs to interact with domain experts and users to carry out this task.

##### **(b) Conceptual design**

In this phase, database designer need to choose appropriate data model to translate the requirement into conceptual schema of the database. The conceptual design describes detail overview of enterprise. E-R model can be used to develop conceptual schema. In terms of E-R model, conceptual schema specifies all entity sets, relationship sets, attributes, and mapping constraints. Conceptual schema is also able to describe functional requirements of the enterprise. In functional requirements user can describes kind of operations that will be performed on data.

##### **(c) Logical design phase:**

In this phase database designer need to maps the high level conceptual schema onto the implementation data model of the database system.

##### **(d) Physical design phase**

In this phase, database designer specifies physical features of the database. These features include form of file organization and storage structure.

### **2.12 Reduction of an E-R Schema to Tables**

We can represent the E-R database schema by a set of tables. Each entity sets and each relationship sets in E-R schema can be represented by their corresponding tables. Each attributes of entity sets and relationship sets are mapped as columns of their corresponding tables. Similarly constraints specified in E-R diagram such as primary key, cardinality constraints etc are mapped to tables generated from E-R diagram. In fact, representing E-R schema into tables is converting E-R model of database into relational model.

## 2. Entity-Relationship Model

### 2.12.1 Tabular representation of Strong Entity Sets

Strong entity set represent by a table with all attributes of its. Let E be a strong entity set with attributes  $a_1, a_2, \dots, a_n$  then it represent by a table E with n distinct columns, each of which correspond to one of the attributes of strong entity set E. Each row in this table corresponds to entity of entity set E.

For example: entity set account with account\_number and balance can be represented by table "account" as

account_number	balance
A1	200
A2	300
A3	500
A4	500

Figure: The account table

Suppose  $D_1$  denotes the set of all account numbers, and  $D_2$  denotes all balances. Each row of the account table contains 2-tuple  $(v_1, v_2)$ , where  $v_1$  is loan number (i.e.  $v_1$  is in set  $D_1$ ) and  $v_2$  is balance (i.e.  $v_2$  is in set  $D_2$ ). In general, we can say that account table contains only a subset of all possible rows. We can refer set of all possible rows of account table as Cartesian product of  $D_1$  and  $D_2$ , denoted by  $D_1 \times D_2$ .

In general, table contains n columns, then set of all possible rows in that table can express by Cartesian product of  $D_1, D_2, \dots, D_n$ , denoted by  $D_1 \times D_2 \times \dots \times D_{n-1} \times D_n$ .

### 2.12.2 Tabular representation of Weak Entity Sets

Let A be a weak entity set with attributes  $a_1, a_2, \dots, a_m$ . Let B be a strong entity set on which A depend on. Let primary key of B consist attributes  $b_1, b_2, \dots, b_n$ . Now weak entity set A can be represent by table A with attributes  $\{a_1, a_2, \dots, a_m\} \cup \{b_1, b_2, \dots, b_n\}$ .

Example: Let us consider weak entity set payment with attributes; payment\_number, payment\_date, and payment\_amount. Consider loan is a strong entity set on which weak entity set payment depends on and suppose that loan\_number is a primary key of loan entity set. Now weak entity set payment can be represent with attributes: loan\_number, payment\_number, payment\_date, and payment\_amount.

loan_number	Payment_number	Payment_date	balance
L12	5	02-04-2005	100
L15	7	12-04-2005	250
L15	8	15-04-2005	150
L19	9	03-05-2005	500

Figure: The payment table

## 2. Entity-Relationship Model

### 2.12.3 Tabular representation of Relationship Sets

Let R be a relationship set with set of attributes  $a_1, a_2, \dots, a_m$  formed by union of primary keys of each entity sets that participates in R. Assume that R consist descriptive attributes  $b_1, b_2, \dots, b_n$ . Now, relationship set R represent by table R with attributes  $\{a_1, a_2, \dots, a_m\} \cup \{b_1, b_2, \dots, b_n\}$ .

Example; consider a relationship set borrower that involves two entity sets

- customer with primary key customer\_id
- loan with primary key loan\_number

Since relationship set does not consist any descriptive attribute, relationship set borrower can represent by table borrower with attributes customer\_id and loan\_number.

customer_id	loan_number
C1	L1
C2	L2
C3	L3
C4	L4

Figure: The borrower relationship set

# 3.0 Relational Model

---

## 3.0 Relational Model

Relational database model is a primary data model for commercial data-processing applications. It is popular because of its simplicity; it provides simple but powerful way of representing data. It also supports complex query. Database in a relational model is simply a collection of one or more relations, where each relation is represented by table with rows and columns. It allows simple high level languages to query data.

### 3.1 Structure of Relational Databases

In relational model, table is a major construct for representing data. Relational database consist set of tables. A row in a table represents a relationship among set of values. So table can be refers as a collection of such relationship.

#### 3.1.1 Basic Construct

To illustrate the basic structure of database, let us consider a table "account"

account_number	branch_name	balance
A-1	Kathmandu	500
A-2	Lalitpur	300
A-3	Kathmandu	700
A-4	Bhaktpur	600

Figure: The account table

The columns of table "account" are: account\_number, brance\_name and balance refer attributes. The set of permitted values for each attribute known as domain of that attribute. For example: set of all accounts numbers is a domain of attribute account\_number.

Let  $D_1$  denotes set of all account number,  $D_2$  denotes set of all branch names and  $D_3$  denotes set of all balances. Any row of table "account" must consist 3-tuple  $(v_1, v_2, v_3)$ , where  $v_1$  is account number (i.e.  $v_1$  is a domain of  $D_1$ ),  $v_2$  is branch name (i.e.  $v_2$  is a domain of  $D_2$ ), and  $v_3$  is account balance (i.e.  $v_3$  is a domain of  $D_3$ ). In general, we can express table "account" will contain only subset of all possible rows. That is, "account" is a subset of

$$D_1 \times D_2 \times D_3$$

In general, a table of  $n$  attributes must be subset of

$$D_1 \times D_2 \times D_3 \times \dots \times D_{n-1} \times D_n$$

This implies that definition of table is almost similar to the definition of relation in mathematics. In mathematics, relationship is a subset of Cartesian product of a list of domains. Therefore, in relational model, table can be refer as a relation and row of table can be refers as tuple. We can define tuple variable to represent a tuple. The above account relation consist seven tuples. Assume that tuple variable  $t$  represents first tuple of the relation. Then, the notation  $t[\text{account\_number}]$  indicates value of  $t$  on account\_number attribute. That is,  $t[\text{account\_number}] = "A-1"$ . Similarly  $t[\text{branch}] = "Kathmandu"$ , and  $t[\text{balance}] = 500$ . We can also represent it as follow:  $t[1]$  where 1 indicated first attribute of relation; that is "account\_number". Therefore,  $t[1] = "A-1"$ . In relational model, we can also express relation as a set of tuples. So definitely,  $t \in r$ .

# **3.0 Relational Model**

---

For each relations r, domains of all attributes of r must be atomic. The domain is said to be **atomic** if elements of domain is indivisible unit. Domains of multivalued and composite attributes are **nonatomic**.

Several attributes may have same domain. Suppose we have two relation customer with customer\_name as one of its attribute and employee is another relation with one of its attribute as employee\_name. It is possible that attribute customer\_name and employee\_name may have same domain. If we look attributes: customer\_name and branch\_name of relations customer and account respectively, at physical level, their domain may be same, both are defined by set of character string. But at logical level, customer\_name and branch\_name must have distinct domain. Null value is a member of any possible domain. It signifies value is unknown or does not exist. Domain for customer\_phoneno of customer entity set may null, meaning is that particular customer does not have any phone number or phone no. is not available.

## **3.1.2 Database Schema**

A relation schema is a list of attributes and their corresponding domains. For example, the relation schema for relation customer is express as

Customer-schema = (customer\_id, customer\_name, customer\_city)

We may also specify domains of attributes as

Customer-schema = (customer\_id: integer, customer\_name: string,  
customer\_city:string)

We may state customer is a relation on Customer-schema by

customer(Customer-schema)

Values or data contain in relation change when it is updated. Relation instance is a snapshot of data in relation at particular time. But, in general, we simply say relation even it is actually relation instance.

In terms of relation, relational database is a collection of relations and relational database schema is a collection of schemas for relations in database. It describes logical design of database. The instance of relational database is collection of relation instances. It is actually a snapshot of data in database at a particular time.

Database schemas for banking enterprise

Branch-schema = (branch\_name,branch\_city,assets)  
Account\_schema = (account\_number,branch\_name,balance)  
Customer-schema = (customer\_id, customer\_name, customer\_street, customer\_city)  
Depositor-schema = (customer\_id, account\_number)  
Loan-schema = (loan\_number, branch\_name, amount)  
Borrower-schema = (customer\_id, loan\_no)

## **3.1.3 Keys**

In relational model, keys (superkey, candidate key and primary key) play important roles. For example: in Branch-schema, {branch\_name} and {branch\_name, branch\_city} are superkey. Since {branch\_name} itself is a superkey, {branch\_name, branch\_city} can not

## 3.0 Relational Model

candidate key in Branch-schema. In Branch-schema, {branch\_name} is a single candidate key so ultimately it is a primary key of Branch-schema.

Let R be a relation schema and  $K \subseteq R$ . If K is superkey for R then it restricts relations  $r(R)$  in which no two distinct tuples have same values on all attribute in K. That is, if  $t_1$  and  $t_2$  are in  $r$  and  $t_1 \neq t_2$  then  $t_1[K] \neq t_2[K]$ .

Relational database schema can be derived from an E-R schema where primary key for relation schema is primary key of entity or relationship set from which relation schema is derived. If relation is derived from strong entity set then primary key for relation is primary key of that strong entity set. Similarly, if relation is derived from weak entity set then primary key for relation is union of primary key of strong entity set and discriminator of weak entity set. And the relation consist

- all attributes of weak entity set
- primary key of the strong entity set on which weak entity set depends

In relational model, relationship set is also represented by a relation. Its primary key is depends up on mapping cardinalities of relationship set. If relationship is many to many then primary key of relation representing relationship set is a union of primary keys of related entity sets. If relationship is one to one then primary key of relation representing relationship set is primary key of any one related entity set. Similarly, if relationship is many to one then primary key of relation representing relationship set is the primary key of the "many" entity set.

In relational model, one relational schema may contain primary key of another relation schema. If relation schema  $r_1$  contains primary key of another relation schema  $r_2$  then this attribute (i.e. PK in  $r_2$ ) in  $r_1$  called *foreign key*. The relation  $r_1$  called referencing relation (detail table) of the foreign key dependency and  $r_2$  called referenced relation (master table).

Example:

In Branch-schema, branch\_name is a primary key. In Account-schema, branch\_name is a foreign key referencing Branch-schema. This implies, in any database instance, any tuple  $t_a$  in account relation, there must be some tuple,  $t_b$  in branch relation such that the value of the branch\_name attribute of  $t_a$  is same as the values of the primary key, branch\_name of  $t_b$ .

### 3.1.4 Schema Diagram

Schema diagram is a graphical representation of database schema along with primary key and foreign key dependencies.

In schema diagram, each relation is represented by box where attributes are listed inside box and relation name is specified above it. Primary key in relation is place above the horizontal line that crosses the box. Foreign key in schema diagram appear as arrow from the foreign key attributes of the referencing relation to the primary key of the referenced relation.

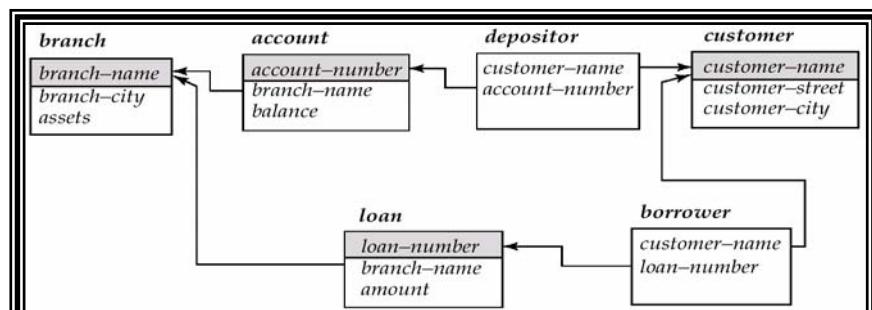


Figure: Schema diagram

# **3.0 Relational Model**

---

**Note:** The difference between E-R diagram and schema diagram is, E-R diagram do not shows the foreign key but schema diagram shows it explicitly.

## **3.1.5 Query Languages**

Query language is a language through which user request information from database. Query languages can be categories in procedural and non procedural. In procedural query language, user required to specify sequence of operations to system to compute desired information where as in nonprocedural query language, user need to specify required information without specifying special procedure for obtaining that information.

Most commercial relational database system offers query language, both procedural and non procedural. SQL is most popular nonprocedural query language.

In this section we discuss pure query language: relational algebra, tuple relational calculus and domain relational calculus. These query languages can not commercially use by people but it describes fundamental techniques for extracting data from database and provides basis for commercial query language.

## **3.2 Relational algebra**

The relational algebra is a procedural query language. It consist set of operation that takes one or more relations as inputs and produce a new relation as output. The fundamental operations in relational algebra are selection, projection, union, set difference, Cartesian product, and rename. Set intersection, natural join, division and assignments other operations of relational algebra which can be define in terms of fundamental operations.

### **3.2.1 Fundamental Operations**

The fundamental operations selection, projection and rename on one relation so they called unary operations. Others operations union, set difference and Cartesian product operates on pairs of relations and so called binary operations.

#### ***The Selection Operation***

The Select Operation selects tuples that satisfy a given predicate. Select is denoted by a lowercase Greek letter sigma ( $\sigma$ ), with the predicate appearing as a subscript. The relation is specifying within parentheses after  $\sigma$ . That is, general structure of selection is

$$\sigma_p(r)$$

where p is selection predicate.

Formally, selection operation define as

$$\sigma_p(r) = \{t | t \in r \text{ and } p(t)\}$$

where p is formula in propositional calculus consisting terms connected by connectives: ^ (and),  $\vee$  (or),  $\neg$  (not). Each term is in the format

$$<\text{attribute}>\text{op}<\text{attribute}>\text{or } <\text{constant}>$$

where op is one of the comparison operators:  $=, \neq, <, \leq, >, \geq$

Examples:

1. Select those tuples of loan relation where the branch is Kathmandu.

$$\sigma_{\text{branch\_name}=\text{"Kathmandu"}(\text{loan})}$$

$$\text{output} = \{t | t[\text{branch\_name}] = \text{Kathmandu}\}$$

## **3.0 Relational Model**

---

2. Find all tuples in loan relation in which amount loan is more than 5000

$$\sigma_{\text{amount} > 5000}(\text{loan})$$

3. Find all tuples in loan relation where amount is more than 5000 and branch is Kathmandu.

$$\sigma_{\text{branch\_name} = \text{"Kathmandu"} \wedge \text{amount} > 5000}(\text{loan})$$

### ***The projection Operation***

The projection operation retrieves tuples for specified attributes of relation. It eliminates duplicate tuples in relation. The projection is denoted by uppercase Greek letter pi ( $\Pi$ ). We need to specify attributes that we wish to appear in the result as a subscript to  $\Pi$ .

The general structure of projection is

$$\Pi_{A_1, A_2, \dots, A_k}(r)$$

where  $A_1, A_2, \dots, A_k$  are attributes of relation  $r$ .

Example: Find account number and their balance from account relation

$$\Pi_{\text{account\_number}, \text{balance}}(\text{account})$$

$$\text{output} == \{t \mid t[\text{account\_number}, \text{balance}]\}$$

### **Composition of relational operations**

Relational algebra operations can be composed together into relational-algebra expression. This required for complicated query.

Example: Find those customers who stay in Kathmandu.

$$\Pi_{\text{customer\_name}}(\sigma_{\text{customer\_city} = \text{"Kathmandu"}}(\text{customer}))$$

### ***Union Operation***

Let  $r$  and  $s$  are two relations then their union defines as

$$r \cup s = \{t \mid t \in r \text{ or } t \in s\}$$

For  $r \cup s$  to be valid, it must hold

- $r, s$  must have same arity (same number of attributes)
- The attribute domain must be compatible (e.g. domain of  $i^{\text{th}}$  column of  $r$  must deals with same type of domain of  $i^{\text{th}}$  column of  $s$ )

Example: Find all customers with either account or loan.

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

### ***Set difference Operation***

The set difference allows us to find tuples that are in one relation but not in another relation. The expression  $r - s$  produces a relation containing those tuples in  $r$  but not in  $s$ .

## 3.0 Relational Model

Formally, let  $r$  and  $s$  are two relations then their difference  $r-s$  define as  
 $r-s = \{t \mid t \in r \text{ and } t \notin s\}$

The set difference must be taken between compatible relations. For  $r-s$  to be valid, it must hold

- $R$  and  $s$  must have the same arity
- Attribute domains of  $r$  and  $s$  must be compatible

Example: Find all customer of the bank who have account but not loan

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

### ***Cartesian Product Operation***

The Cartesian product operation denoted by cross ( $\times$ ). It allows us to combine information from any two relations. Cartesian product of two relations  $r$  and  $s$ , denoted by  $r \times s$  returns a relation instance whose schema contains all the fields of  $r$  (in same order as they appear in  $r$ ) followed all field of  $s$  (in the same order as they appear in  $s$ ). The result of  $r \times s$  contains one tuples  $\langle r, s \rangle$  (concatenation of tuples of  $r$  and  $s$ ) for each pair tuples  $t \in r$ ,  $q \in s$ . Formally,

$$r \times s = \{ \langle t, q \rangle \mid t \in r \text{ and } q \in s \}$$

Example 1:

A	B
$\alpha$	1
$\beta$	2

C	D	E
$\alpha$	10	a
$\beta$	10	a
$\beta$	20	b
$\gamma$	10	b

Relation  $r$

Relation  $s$

$r \times s$ :

A	B	C	D	E
$\alpha$	1	$\alpha$	10	a
$\alpha$	1	$\beta$	10	a
$\alpha$	1	$\beta$	20	b
$\alpha$	1	$\gamma$	10	b
$\beta$	2	$\alpha$	10	a
$\beta$	2	$\beta$	10	a
$\beta$	2	$\beta$	20	b
$\beta$	2	$\gamma$	10	b

$\sigma_{A=a}(r \times s)$ :

A	B	C	D	E
$\alpha$	1	$\alpha$	10	a
$\alpha$	1	$\beta$	10	a
$\alpha$	1	$\beta$	20	b
$\alpha$	1	$\gamma$	10	b

### 3.0 Relational Model

---

Example 2:

customer_name	loan_number
X	L01
Y	L02

Relation borrower

loan_number	branch_name	amount
L01	B1	5000
L02	B2	6000

Relation loan

Query: Find all customer who taken loan from branch "B1".

$$\Pi_{\text{customer\_name}}(\sigma_{\text{borrower.loan\_number}=\text{loan.loan\_number}}(\sigma_{\text{branch\_name}=\text{"B1"}}(\text{borrower} \times \text{loan})))$$

Process:

borrower $\times$ loan

customer_name	borrower.loan_number	loan.loan_number	branch_name	amount
X	L01	L01	B1	5000
X	L01	L02	B2	6000
Y	L02	L01	B1	5000
Y	L02	L02	B2	6000

$$\sigma_{\text{branch\_name}=\text{"B1"}}(\text{borrower} \times \text{loan})$$

customer_name	borrower.loan_number	loan.loan_number	branch_name	amount
X	L01	L01	B1	5000
Y	L02	L01	B1	5000

$$\sigma_{\text{borrower.loan\_number}=\text{loan.loan\_number}}(\sigma_{\text{branch\_name}=\text{"B1"}}(\text{borrower} \times \text{loan}))$$

customer_name	borrower.loan_number	loan.loan_number	branch_name	amount
X	L01	L01	B1	5000

$$\Pi_{\text{customer\_name}}(\sigma_{\text{borrower.loan\_number}=\text{loan.loan\_number}}(\sigma_{\text{branch\_name}=\text{"B1"}}(\text{borrower} \times \text{loan})))$$

customer_name
X

# 3.0 Relational Model

## The Rename Operation

The result of relational-algebra expression does not have a name to refer it. It is better to give name to result relation. The rename operator is denoted by lower case Greek letter rho ( $\rho$ ). Rename operation in relation-algebra expressed as

$$\rho_x(E)$$

where E is a relational algebra expression and x is name for result relation. It returns the result of expression E under the name x.

Since a relation r is itself a relational-algebra expression thus, the rename operation can also apply to rename the relation r (i.e. to get same relation under a new name). Rename operation can also used to rename attributes of relation. Assume a relational algebra expression E has arity n. Then expression

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

returns the result of expression E under the name x and it renames attributes to A1,A2, . ,An.

Example 1: Find the largest account balance in the bank.

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

Process:

account_number	balance
A1	500
A2	600
A3	700

Relation account

Account_number	balance
A1	500
A2	600
A3	700

Relation d

$$\text{account} \times \rho_d(\text{account})$$

account.account_number	account.balance	d.balance
A1	500	500
A1	500	600
A1	500	700
A2	600	500
A2	600	600
A2	600	700
A3	700	500
A3	700	600
A3	700	700

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

Account.balance

# 3.0 Relational Model

500
600

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

balance
500
600
700

account.balance
500
600

Output:

balance
700

Example 2: Find the names of all customers who live on the same street and in the same city as smith.

$$\Pi_{\text{customer.customer}}(\sigma_{\text{customer.customer\_street} = \text{smith\_add.street} \wedge \text{customer.customer\_city} = \text{smith\_add.city}} (\text{customer} \times \rho_{\text{smith\_add(street,city)}} (\Pi_{\text{customer\_street}, \text{customer\_city}} (\sigma_{\text{customer\_name} = \text{"smith"}}(\text{customer}))))))$$

## 3.2.2 Formal Definition of Relational Algebra

Basic expressions in relational algebra are

- Relation in a database
- A constant relation
  - A constant relation is expression by listing its tuples
    - $\{(A01, "B1", 500)(A02, "B2", 600)\}$

From the basic expression we can construct other expression. Let E1 and E2 be relational algebra expression, then following are also relational-algebra expression.

$$E_1 \cup E_2$$
$$E_1 - E_2$$
$$E_1 \times E_2$$
$$\sigma_p(E_1)$$
, p is a predicate on attributes in E1.
$$\Pi_s(E_1)$$
, s is a list of some attributes in E1
$$\rho_x(E_1)$$
, x is the new name for the result of E1

## 3.2.3 Additional Operations

The fundamental operations of the relational algebra are sufficient to express any relational algebra query. But for complex query it is difficult. Additional operations (set intersection, natural join, division, assignment) simplify the common queries.

### ***Set-intersection operation***

Let r and s are two relation having same arity and attributes of r and s are compatible then their intersection  $r \cap s$  define as

$$r \cap s = \{t | t \in r \text{ and } t \in s\}$$

In terms of fundamental operation of relational algebra it can express as

# 3.0 Relational Model

$$r \cap s = r - (r - s)$$

Example 1:

A	B
$\alpha$	1
$\alpha$	2
$\beta$	1

Relation r

A	B
$\alpha$	2
$\beta$	3

relation s

A	B
$\alpha$	2

$r \cap s$ :

Example 2: Find all customer who have both loan and account

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

## The natural join operation

The natural join operation generally needs to simplify queries that required a Cartesian product. The natural join allow to combine certain selections and a Cartesian product into one operation. It is denoted by symbol



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 schema and finally removes duplicate attributes.

Formally, let  $r$  and  $s$  are two relations on schema  $R$  and  $S$  respectively then  $r \bowtie s$  is a relation on schema  $R \cup S$ . That is,

$$r \bowtie s = \Pi_{R \cup S} (\sigma_{r.A_1=s.A_1 \wedge r.A_2=s.A_2 \wedge \dots \wedge r.A_n=s.A_n} (r \times s))$$

where  $\{A_1, A_2, \dots, A_n\}$  are common attributes in  $R$  and  $S$ .

Example 1:

Let

$$\begin{aligned} R &= (A, B, C, D) \\ S &= (E, B, D) \end{aligned}$$

Now,

Result schema =  $(A, B, C, D, E)$

$r \bowtie s$  is define as

$$\Pi_{r.A, r.B, r.C, r.D, s.E} (\sigma_{r.B=s.B \wedge r.D=s.D} (r \times s))$$

### 3.0 Relational Model

Suppose r and s are two relation as follow

A	B	C	D
$\alpha$	1	$\alpha$	a
$\beta$	2	$\gamma$	a
$\gamma$	4	$\beta$	b
$\alpha$	1	$\gamma$	a
$\delta$	2	$\beta$	b

B	D	E
1	a	$\alpha$
3	a	$\beta$
1	a	$\gamma$
2	b	$\delta$
3	b	$\epsilon$

Relation r

relation s

$r \bowtie s$ :

A	B	C	D	E
$\alpha$	1	$\alpha$	a	$\alpha$
$\alpha$	1	$\alpha$	a	$\gamma$
$\alpha$	1	$\gamma$	a	$\alpha$
$\alpha$	1	$\gamma$	a	$\gamma$
$\delta$	2	$\beta$	b	$\delta$

Example 2: Find the names of all customer who have a loan at the bank

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

Same query is express by fundamental operation as follow

$\Pi_{\text{customer\_name}}(\sigma_{\text{borrower.loan\_number} = \text{loan.loan\_number}}((\text{borrower} \times \text{loan}))$

Example 3: Find names of all branches with customer who have account in the bank and who live in Kathmandu.

$\Pi_{\text{customer\_name}}(\sigma_{\text{customer\_city} = \text{"Kathmandu"}} \text{customer} \bowtie \text{account} \bowtie \text{depositor})$

Example 4: Find all customers who have both loan and account at the bank.

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

In the set intersection form this can be express as

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

**Note 1:** Let r(R) and s(S) e relations without any attributes in common. That is  $R \cap S = \emptyset$  then

$r \bowtie s = r \times s$

#### Note 2: Theta Join

The theta join is an extension to the natural join operations that allow us to combine a selection and a Cartesian product into a single operation with predicate on attributes.

Let relation r(R) and s(S), and  $\Theta$  be a predicate on attributes in the schema  $R \cup S$  then theta join operation  $r \bowtie \Theta s$  is defined as

# 3.0 Relational Model

$$r \bowtie \Theta s = \sigma_{\Theta}(r \times s)$$

Example: Find all customers who have loan and stay in Kathmandu.

$$\Pi_{\text{customer\_name}(\text{borrower}) \bowtie_{\text{customer\_city}=\text{"Kathmandu"}} \text{loan}}$$

## Division Operation

Let  $r$  and  $s$  be the relations on schemas  $R$  and  $S$  respectively where  $S \subseteq R$  (i.e. every attributes of schema  $S$  is also in schema  $R$ ) then  $r \div s$  is a relation on schema  $(R-S)$ , define as

$$r \div s = \{t | t \in \Pi_{R-S}(r) \wedge \forall u \in s (tu \in r)\}$$

Example:

A	B
$\alpha$	1
$\alpha$	2
$\alpha$	3
$\beta$	1
$\gamma$	1
$\delta$	1
$\delta$	3
$\delta$	4
$\epsilon$	6
$\epsilon$	1
$\beta$	2

B
1
2

Relation r

Relation s

$$r \div s :$$

A
$\alpha$
$\beta$

Example 2:

A	B	C	D	E
$\alpha$	a	$\alpha$	a	1
$\alpha$	a	$\gamma$	a	1
$\alpha$	a	$\gamma$	b	1
$\beta$	a	$\gamma$	a	1
$\beta$	a	$\gamma$	b	3
$\gamma$	a	$\gamma$	a	1
$\gamma$	a	$\gamma$	b	1
$\gamma$	a	$\beta$	b	1

D	E
a	1
b	1

Relation r

Relation s

Example: Find all customers who have an account at all the branches located in Kathmandu.

## **3.0 Relational Model**

---

We can obtain all branches in Kathmandu by expression

$$r1 = \Pi_{\text{branch\_name}}(\sigma_{\text{cbranch\_city}=\text{"Kathmandu"}}(\text{branch}))$$

We can obtain all (customer\_name, branch\_name) pair for the customer who have account.

$$r2 = \Pi_{\text{customer\_name}, \text{branch\_name}}(\text{depositor} \bowtie \text{account})$$

The required customer can obtain from

$$r2 \div r1$$

That is,

$$\Pi_{\text{customer\_name}, \text{branch\_name}}(\text{depositor} \bowtie \text{account}) \div \Pi_{\text{branch\_name}}(\sigma_{\text{cbranch\_city}=\text{"Kathmandu"}}(\text{branch}))$$

### ***The assignment operations***

The assignment operation provides convenient way to express complex query. The assignment operation denoted by  $\leftarrow$ , works like assignment in programming language. The evaluation of an assignment does not result any relation being displayed to the user. But the result of the expression to the right of the  $\leftarrow$  is assigned to the relation variable. This relation variable may used in subsequent expressions. With the assignment expression, a query can be written as a sequential program consisting a series of assignments followed by an expression whose value is displayed as the result of the query.

Example: Find all customer who taken loan from bank as well as he/she has bank account.

$$\text{Temp1} \leftarrow \Pi_{\text{customer\_name}}(\text{borrower})$$

$$\text{Temp2} \leftarrow \Pi_{\text{customer\_name}}(\text{depositor})$$

$$\text{result} \leftarrow \text{Temp1} \cap \text{Temp2}$$

### **3.2.4 Extended Relational-Algebra operations**

Generalized projection, outer join and aggregation function are extension on basic relational algebra operation.

#### ***Generalized Projection***

Generalized projection operation allows arithmetic functions in the projection list. The general structure is

$$\Pi_{F_1, F_2, \dots, F_n}(E)$$

where E is any relational algebra expression. Each  $F_1, F_2, \dots, F_n$  are arithmetic expression involving constraints and attributes in the schema of E.

Example 1:

Suppose a relation

$$\text{credit\_info}(\text{customer\_name}, \text{credit\_limit}, \text{credit\_balance})$$

Query: Find how much more each person can spend.

$$\Pi_{\text{customer\_name}, \text{credit\_limit} - \text{credit\_balance}}(\text{credit\_info})$$

## 3.0 Relational Model

The resulting attribute from credit\_limit – credit\_balance does not have name; its name can be specify as below

$$\Pi_{\text{customer\_name}, \text{credit\_limit} - \text{credit\_balance}} \text{as credit\_available}(\text{credit\_info})$$

Example 2:

Suppose relation

$$\text{employee}(\text{employee\_id}, \text{ename}, \text{salary})$$

Find employee and their corresponding bonus, assume that bonus for each employee is 10% of his/her salary.

$$\Pi_{\text{ename}, \text{salary} * 1.10} \text{as bonus}(\text{employee})$$

### **Aggregate function and operations**

Aggregate function takes a collection of values and return as a single value as a result.

Some aggregate functions are

- AVG: average value
- MIN: minimum value
- SUM: sum of values
- Count: number of values

The aggregate operation in relational algebra denoted by the symbol  $\mathfrak{g}$  (i.e.  $\mathfrak{g}$  is the letter G in calligraphic font). The general structure is

$$\mathfrak{g}_{G_1, G_2, \dots, G_n} F_1(A_1), F_2(A_2), \dots, F_n(A_n)(E)$$

where E is any relational algebra expression.

- $G_1, G_2, \dots, G_n$  is a list of attributes on which to group (it could be empty)
- Each  $F_i$  is aggregate function.
- Each  $A_i$  is an attribute name

Example:

A	B	C
$\alpha$	$\alpha$	7
$\alpha$	$\beta$	7
$\alpha$	$\beta$	3
$\beta$	$\beta$	10

Relation r

$$\mathfrak{g}_{\text{sum}(C)}(r): \quad \boxed{\begin{array}{|c|}\hline \text{Sum-C} \\ \hline 27 \\ \hline \end{array}}$$

Example 2: Find the balance to each branch.

$$\text{branch\_name } \mathfrak{g}_{\text{sum(blance)}} \text{ as sum-blance}(\text{account})$$

Example 3: Find no. of account in each branch

# 3.0 Relational Model

branch\_name  count(account\_number) (account)

## Outer join

The outer-join operation is extension to natural join. It has a capability to deal with missing information. There are three form of outer-join operation

(a) Left outer-join ( $\bowtie_L$ )

- Takes all tuples in the left relation. If there are any tuples in right relation that does not match with tuple in left relation, simply pad these right relation tuples with null.
- Add them to the result of the left outer-join.

(b) Right outer-join ( $\bowtie_R$ )

- Takes all tuples in the right relation. If there are any tuples in the left relation that does not match with tuple in right relation, simply pad left relation tuples with null.
- Add them to the result of the left outer-join.

(c) Full outer-join ( $\bowtie_F$ )

- Pad tuples from the left relation that did not match any from the right relation
- Pad tuples from the right relation that did not match any from the left relation
- Add them to the result of full outer-join.

Example:

Consider relations

loan_number	branch_name	amount
L01	B1	500
L02	B2	600
L05	B1	700

Relation loan

customer_name	loan_number
X	L01
Y	L02
Z	L07

Relation borrower

## Natural join (Inner join)

Loan  $\bowtie$  borrower

loan_number	branch_name	amount	customer_name
L01	B1	500	X
L02	B2	600	Y

# 3.0 Relational Model

## Left outer-join

loan  $\bowtie$  borrower

loan_number	branch_name	amount	customer_name
L01	B1	500	X
L02	B2	600	Y
L05	B1	700	null

## Right outer-join

Loan  $\bowtie$  borrower

loan_number	branch_name	amount	customer_name
L01	B1	500	X
L02	B2	600	Y
L07	null	null	Z

## Full outer-join

Loan  $\bowtie$  borrower

loan_number	branch_name	amount	customer_name
L01	B1	500	X
L02	B2	600	Y
L05	B1	700	null
L07	null	null	Z

## 3.2.5 Null Values

Tuples may not have any values for some of the attributes. At that time the attribute is said to have null value and is denoted by null. It simplifies an unknown value or a value does not exist. The result of any arithmetic or comparison involving null is null. There are often more than one possible way of dealing with null values, as a result our definition can sometimes be arbitrary. Therefore arithmetic operations and comparison on null values should avoid if possible.

Comparison involving nulls may occur inside Boolean expression: AND, OR and NOT operations. Boolean operation deal with null value is as follow.

AND: (true and null)=null  
(false and null)=false  
(null and null)=null

OR: (null or true)=true  
(null or false)=null  
(null or null)=null

NOT: (not null)=null

## **3.0 Relational Model**

---

**How different relation operations deal with null values ?**

### **Select**

The select operation evaluates predicate  $p$  in  $\sigma_p(E)$  on each tuple  $t$  in  $E$ . If predicate returns true value then  $t$  is added to the result. Otherwise predicate returns null or false and  $t$  is not added to the result.

### **Join**

In natural join,  $r \bowtie s$ , if two tuples  $t_r \in r$  and  $t_s \in s$ , both have a null values in common attributes then tuples do not match.

### **Projection**

The projection treats null just like any other value when eliminating duplicates. If two tuples in the projection result are exactly the same, and both have nulls in the same fields, they are treated as duplicated. For example

A	B	C
$\alpha$	null	$\gamma$
$\alpha$	null	$\gamma$

Here, projection assumes these two tuples are duplicates.

This is arbitrary decision since without knowing the actual value, we can not tell two instances of null are duplicates or not.

### **Union, intersection and difference**

These operation also treats null value as any other values when eliminating duplicates. These operations treat tuples that have same values on all fields as duplicates even if some of the fields have null values in both tuples. This decision is arbitrary, especially in the case of intersection and difference since the actual values (if any) represented by nulls are same.

### **Generalized projection**

It treats null value same as projection.

### **Outer join**

Outer join operation behaves null value just like natural join operation. In outer join, tuples that do not occur in the natural join result may be added to the result of outer join padded with nulls.

## **3.2.6 Modification of the database**

Insertion, deletion and updating operations are responsible for database modification.

### **Deletion**

A delete request is expressed similarly to the query, except instead of displaying tuples, the selected tuples are removed from the databases. Delete request can delete only whole tuples, can not delete values on only particular attributes. Deletion is expressed as

$$r \leftarrow r - E$$

where  $r$  is a relation and  $E$  is a relational algebra query.

Example 1: Delete all account in the "B1" branch.

$$\text{account} \leftarrow \text{account} - \sigma_{\text{branch\_name} = "B1"}(\text{account})$$

Example 2: Delete all records with account in the range of 0 to 5.

$$\text{loan} \leftarrow \text{loan} - \sigma_{\text{amount} > 0 \text{ and } \text{amount} \leq 50}(\text{loan})$$

### **insertion**

## **3.0 Relational Model**

---

to insert data into relation we can either specify a tuple to be inserted or write a query whose result is a set of tuples to be inserted. In relational-algebra, an insertion is express by

$$r \leftarrow r \cup E$$

where  $r$  is a relation and  $E$  is a relational algebra expression.

Example: insert information in the database specifying customer "X" has 5000 in account A1 at the kathmandu city.

```
account ← account ∪ {(A1,"kathmandu",5000)}  
depositor ← depositor ∪ {("x",A1)}
```

### ***Updating***

Updating allow to change a value in a tuple without changing all values in tuple. In relational algebra, updating express by

$$r \leftarrow \Pi_{F_1, F_2, \dots, F_n}(r)$$

where each  $F_i$  is either

- $i^{\text{th}}$  attributes of  $r$ , if the  $i^{\text{th}}$  attribute is not updated or
- expression involving only constant and attributes of  $r$ , if the attribute is to be updated. It gives the new value for the attribute.

Example 1: increase balance by 5% to all branches.

```
account ← \Pi_{account_number, branch_name, balance * 1.05}(account)
```

Example 2: Increase the balance by 6% for those account which balance is over 5000 and for the rest of account increase balance by 5%.

```
account ← \Pi_{account_number, branch_name, balance * 1.06}(\sigma_{balance > 5000}(account))  
          \cup \Pi_{account_number, branch_name, balance * 1.05}(\sigma_{balance \leq 5000}(account))
```

## **3.2.7 Views**

For a security reason, it is not desirable for all users to see the entire logical model of database (i.e. all actual relations stored in database). We may required certain data to be hidden from users. We may wish to create a personalized collection of relations that is better matched to certain user's intuition than whole database. For example, an employee in advertising department might to see a relation consisting of the customers who have either an account or loan at the bank and the branches with which they do business. In relational algebra it is express by

$$\begin{aligned} &\Pi_{branch\_name, customer\_name}(depositor \bowtie account) \cup \Pi \\ &branch\_name, customer\_name(borrower \bowtie loan) \end{aligned}$$

Any relation that is not the part of the logical model, but it is made visible to a user as a virtual relation called a view. There would no. of views for any given set of actual relations.

## **3.2.8 View Definition**

View is defined by using the create view statement. The general structure is

Create view <view name> as <query expression>

where <query expression> is any legal relational-algebra query expression. Once a view is defined, it can refer by its virtual name, called view name.

Example: Create a view "all-customer" consisting branches and their customers.

Create view all-customer as

## 3.0 Relational Model

$$\Pi_{branch\_name, customer\_name} (depositor \bowtie account) \cup \Pi_{branch\_name, customer\_name} (borrower \bowtie loan)$$

We can query on this view as in other relation. For example,

Query: find all customer of "B1" branch.

$$\Pi_{customer\_name} (\sigma_{branch\_name = "B1"} (all-customer))$$

View definition is not same as creating a new relation evaluating the query express. It is actually substitution to query expression from which it is defined. If the view is stored, it may become out of date. If the relations used to be define it are modified. To avoid this, database system stores the definition of view itself, rather than the result of evaluation of relational-algebra expression that defines the view. Whenever we evaluate the query, the view relation is recomputed.

Certain database system stores result of evaluation of the relational algebra expression that defines view. If the actual relation used in the view definition change, the view need to kept up to date; such view called *materialized view*. The process of keeping the view up to date called *view maintenance*.

### 3.2.9 Updates through view and null values

Although views are useful tool for queries, it gives serious problem if we allow insertion, deletion and updates from view. Any modification made by view must be translated to actual relations in database. It is difficult task. In some case, it is not possible Let us consider example to illustrate this.

Consider a view loan-branch which is provided to clerk to see all loan data in loan relation, except loan amount.

Create view loan-branch as

$$\Pi_{loan\_number, branch\_name} (loan)$$

Assume that clerk try to insert loan information and write

$$loan-branch \leftarrow loan-branch \cup \{(L01, "B1")\}$$

In fact, this insertion must made to the relation loan. However to insert a tuple into loan, we must have value for account. To deal with this problem we may choose to options

- Reject insertion and return an error message to the user
- Insert tuple (L01, "B1", null) into a loan relation.

Let us consider another view to illustrate another problem with modification of the database through view.

Consider a view loan-info providing loan amount for each loan of customer

Create view loan-info as

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

Assume that following insertion is perform to view

$$loan-info \leftarrow loan-info \cup \{"X", 5000\}$$

Since view loan-info is created from multiple relation (i.e customer\_name is taken from borrower, account is taken from loan). So we have to insert tuple ("X", null) into borrower and (null, null, 5000) into loan. This insertion may leads several complication, loan can be taken with out loan number so it is impossible to map loan and their corresponding customer in actual relation. If loan\_number is define as primary key in loan relation then this insertion is not possible. Because of these problems, modification on view is not permitted generally.

# 3.0 Relational Model

---

## 3.2.10 Tuple relational calculus

Tuple relational calculus is nonprocedural query language. It describes desired information without specifying procedure for obtaining that information. The general structure of query in relational calculus is express as

$$\{t | p(t)\}$$

read as; set of all tuples t such that predicate p is true for t.

General notation

- t is a tuple variables.
- T[A] denotes the value of tuple ton attribute A.
- $t \in r$  denotes the tuple t in relation r.
- p is a formula similar to the predicate calculus. Predicate calculus formula consist
  - set of attributes and constant
  - set of comparison operator (e.g.  $<$ ,  $\leq$ ,  $=$ ,  $\neq$ ,  $>$ ,  $\geq$ )
  - set of connectives: and ( $\wedge$ ), or ( $\vee$ ), not ( $\neg$ )
  - implication ( $\Rightarrow$ ):  $X \Rightarrow Y$  (i.e. if X is true, then Y is true)
  - set of quantifiers
    - $\exists t \in r(Q(t))$  : "there exist" a tuple t in relation r such that predicate Q(t) is true.
    - $\forall \exists t \in r(Q(t))$  : Q is true "for all" tuples t in relation r

Example queries

1. Find the loan number, branch name and amount for loan of over 5000  
 $\{t | t \in \text{loan} \wedge t[\text{amount}] > 5000\}$
2. Find the loan number for each loan of account greater than 1200.  
 $\{t | \exists s \in \text{loan}(t[\text{loan\_number}] = s[\text{loan\_number}] \wedge s[\text{amount}] > 1200)\}$
3. Find the names of all customer having a loan, an account or n=both at the bank.  
 $\{t | \exists s \in \text{borrower}(t[\text{customer\_name}] = s[\text{customer\_name}]) \vee \exists u \in \text{depositor}(t[\text{customer\_name}] = u[\text{customer\_name}])\}$
4. Find the names of all customers who have a loan and account at the bank.  
 $\{t | \exists s \in \text{borrower}(t[\text{customer\_name}] = s[\text{customer\_name}]) \wedge \exists u \in \text{depositor}(t[\text{customer\_name}] = u[\text{customer\_name}])\}$
5. Find the names of all customers having a loan at the "B1' branch.  
 $\{t | \exists s \in \text{borrower}(t[\text{customer\_name}] = s[\text{customer\_name}]) \wedge \exists u \in \text{loan}(u[\text{branch\_name}] = "B1" \wedge u[\text{loan\_number}] = s[\text{loan\_numnber}])\}$
6. Find the names of all customers who have a loan at the "B1" branch, bt no. amount at any branch of the bank.  
 $\{t | \exists s \in \text{borrower}(t[\text{customer\_name}] = s[\text{customer\_name}]) \wedge \exists u \in \text{loan}(u[\text{branch\_name}] = "B1" \wedge u[\text{loan\_number}] = s[\text{loan\_numnber}] \wedge \text{not } \exists v \in \text{depositor}(v[\text{customer\_name}] = t[\text{customer}]))\}$
7. Find the names of all customer and their city they stay having a loan from the branch "b1"  
 $\{t | \exists s \in \text{loan}(s[\text{branch\_name}] = "B1" \wedge \exists u \in \text{borrower}(u[\text{loan\_number}] = s[\text{loan\_number}] \wedge t[\text{customer\_name}] = u[\text{customer\_name}]) \wedge \exists v \in \text{customer}(u[\text{customer\_name}] = v[\text{customer\_name}] \wedge t[\text{customer\_city}] = v[\text{customer\_city}]))\}$
8. Find the names of all customers who have an account at all branch located in "Kathmandu".  
 $\{t | \exists c \in \text{customer}(t[\text{customer\_name}] = c[\text{customer\_name}]) \wedge \forall s \in \text{branch}(s[\text{branch\_city}] = "Kathmandu" \Rightarrow \exists u \in \text{account}(s[\text{branch\_name}] = u[\text{branch\_name}] \wedge \exists s \in \text{depositor}(t[\text{customer\_name}] = s[\text{customer\_name}] \wedge s[\text{account\_number}] = u[\text{account\_number}]))\}$

## **4.0 Structure Query Language (SQL)**

---

---

### **4.0 Structure Query Language (SQL)**

SQL was developed in 1970's in an IBM laboratory "San Jose Research Laboratory" (now the Amaden Research center). SQL is derived from the SEQUEL one of the database language popular during 1970's. SQL established itself as the standard relational database language. Two standard organization (ANSI) and International standards organization (ISO) currently promote SQL standards to industry.

In 1986 ANSI & ISO published an SQL standard called SQL-86. In 1987, IBM published its own corporate SQL standard, the system application Architecture Database Interface (SAA-SQL). In 1989, ANSI published extended standard for SQL called, SQL-89. The next version was SQL-92, and the recent version is SQL: 1999.

### **4.1 Basic Term and Terminology**

**Query:** is a statement requesting the retrieval of information.

**Query language:** language through which user request information from database. These languages are generally higher level language than programming language.

The two types of query language are:

#### **(i) Procedural language**

- User instructs the system to perform sequence of operation on the database to compete the desired result. Example : relational algebra

#### **ii) Non- procedural language**

- User describes the desired information without giving a specific procedure for obtaining that desired information.
- Examples: tuple relational calculus and domain relational calculus.

### **4.2 Database Languages**

Two types of database language

1. Data Definition Language (DDL)
2. Data Manipulation Language (DML)

#### **Data Definition Language**

- Specifies the database schema.
- For e.g.: The following statement in SQL defines relation named `student'.

```
CREATE TABLE student
(
    student_id VARCHAR2(3),
    address      VARCHAR(30)
);
```

The execution of this DDL statement creates the `student' table. It also updates a special set of tables called *data dictionary* or *data directory*. A data dictionary contains metadata, that is data about data. The schema of table is an example of metadata. A database system consults data dictionary or data directory.

Through the set of special type of DDL, called data storage definition language, we may specify the storage structure (like size of database, size of table etc) and access methods.

## 4.0 Structure Query Language (SQL)

---

The DDL allow to enforce constraints in the database. For example: student\_id should begin with 'S', address could not be null etc.

```
CREATE TABLE STUDENT
(
    student_id  VARCHAR2 (3),
    address      VARCHAR2 NOT NULL,
    CONSTRAINT ch_student_id CHECK (student_id LIKE 'S%')
);
```

The database systems check these constraints every time the database is updated.

### Data Manipulation Language:

- A data manipulation language is a language that enables users to access or manipulate the data in database. The data manipulation means :
  - Retrieval of information stored in database.
  - The insertion of new information into database.
  - Deletion of information from database.
  - Modification of data in database.

Two types of data manipulation languages are:

- Procedural DML: User need to specify what data are needed to retrieve (modify) and how to retrieve those data.
- Non Procedural (Declarative DML) : User requires to specify what data are needed to retrieve without specifying how to get (retrieve) those data.
  - Non procedural DML are easier to understand and use than procedural DML, since user does not have to specify how to get data from database.
  - The DML component of SQL is non procedural language.

Example: Consider a simple relational database.

customer_id	customer_name	customer_address
C001	Smith	Kathmandu
C002	John	Bhaktapur
C003	Semi	Lalitpur
C004	Ivan	Kathmandu

The customer table

account_no	balance
A101	900
A102	300
A103	200
A104	700

The account table

customer_id	account_no
C001	A001
C002	A002
C003	A004
C004	A004

The depositor table

## 4.0 Structure Query Language (SQL)

---

---

*Some queries and their equivalent SQL statement*

**Query: Find the name of customer whose customer\_id C001.**

```
SELECT customer.customer_name FROM customer  
      WHERE customer.customer_id = `C001`;  
            OR  
SELECT customer_name FROM customer  
      WHERE customer_id = `C001`;
```

**Note:** We don't need to specify the table name while referencing column\_name if we are taking column from only one table.

**Query: Find the name and balance of the customer.**

```
SELECT customer.customer_name,account.balance  
      FROM customer,account.balance  
            WHERE customer.customer_id= depositor.customer_id  
                  AND depositor.acount_no = account.account_no;
```

**Problem : Insert record to customer table.**

```
customer_id : C005  
customer_name : MICHAEL  
address : KATHMANDU
```

```
INSERT INTO customer (customer_id, customer_name, address)  
      VALUES (`C005` , `MICHAEL` , `KATHMANDU` ) ;  
            OR  
INSERT INTO Customer values (`C005` , `MICHAEL` , `KATHMANDU` );
```

**Note:** Column name need not to specify if we are going to insert values for all columns of table.

**Query: Delete record from depositor whose customer\_id is `C004`.**

```
:  
      DELETE FROM depositor WHERE Customer_id = `C004`;
```

**What happen if we execute DELETE statement as below?**

```
DELETE FROM depositor;  
      • Deletes all records from the table `depositor`
```

**Problem: If you attempt to delete all records of customer from customer table, what happen ?**

- You cannot delete all records, only when "account" and "deposit" tables are empty or only when these table contains records that are not related to the customer.

**Query: Increase the balance by 5% in account table whose account no is `A101` or current balance is only 200.**

```
UPDATE account  
      SET balance = balance + (balance + 0.05)  
      WHERE account_no = `A1001` OR balance = 200;
```

## **4.0 Structure Query Language (SQL)**

---

---

Note:

Sometimes database languages are also categorized with Data Control Language.

That is

1. Data Definition Language (DDL)
2. Data Control Language (DCL)
3. Data Manipulation (DML)

Data Control language is a language that controls the behavior of database.

In SQL, COMMIT, ROLLBACK, commands go under Data Control Language.

- COMMIT: Saves the changes made to database.
- ROLLBACK: Undo changes to database from the current state database to last commit state.

**Question: Why we need query language, even we have formal languages?**

(formal languages, relational algebra, relational calculus )

- The formal languages provides concise notation for representing query. But commercial database system requires more user friendly query language. This is a main reason for why we need query languages, even we have formal languages.
- The formal languages form the basis for data manipulation language of DBMS only but DBMS/commercial DBMS also supports data definition capabilities as well as data manipulation capabilities.
- SQL is a most popular and powerful query language. It can do much more than just query database. It can define structure of data, modify data in database and allow to specify security constraints.
- SQL is a truly non procedural language. It has all features of relational algebra, relational calculus as well as its own powerful features.

### **4.3 Different parts of SQL Language**

#### **1. Data Definition Language (DDL):**

SQL DDL provides commands for defining relation schemas, deleting schema, deleting relations and modifying relational schemas.

Example:

CREATE, ALTER, DROP

```
CREATE TABLE dept
(
    dept no NUMBER(2) PRIMARY KEY,
    dname  VARCHAR2(20) NOT NULL
);
```

```
CREATE TABLE emp
(
    empno NUMBER(5) PRIMARY KEY,
    deptno NUMBER(3),
    ename VARCHAR2(10) NOT NULL,
    sal NUMBER(5) NOT NULL,
    CONSTRAINT fk_emp_dept FOREIGN KEY(deptno) REFERENCES dept
```

## **4.0 Structure Query Language (SQL)**

---

---

);

```
ALTER TABLE dept ADD (loc VARCHAR2(10));
ALTER TABLE emp MODIFY (empno NUMBER(10));
ALTER TABLE emp ADD UNIQUE (ename);
DROP TABLE emp;
DROP constraint fk_emp_dept;
```

### **2. Data Manipulation Language ( DML):**

The SQL DML includes query language based on relational algebra and relational calculus. It includes commands for insert tuples, delete tuples and modify tuples in database.

Example: INSERT, DELETE, UPDATE, SELECT etc. statements.

### **3. View Definition:**

The SQL DDL includes for defining views e.g.

Syntax:

```
CREATE VIEW <view name> AS
(<query expression>);
```

### **4. Transaction Control: (Data Control Language):**

SQL includes commands for specifying integrity constraints that the data stored in database must satisfy.

### **5. Embedded SQL and dynamic SQL:**

Embedded SQL & dynamic SQL dynamic SQL is that SQL with general purpose programming language; such as C, C++, JAVA, COBAL, PASCAL, FORTRAN.

### **6. Integrity:**

SQL DDL includes commands for specifying integrity constraints that the data stored in database must satisfy.

### **7. Authorization:**

SQL DDL commands used for specifying access right to relation relations and views.

## **4.4 General overview of SQL:**

Though SQL user / programmer / DBA can perform the following task:

- Create database.
- Modify a database structure.
- Add user permissions to database or tables.
- Changes system security.
- Query a database for a retrieval of information.
- Updates the contents of information.

Note: Commands in SQL are not necessarily a question, request to the database. It could be a command to do one of the following.

- Build or delete a table.
- Insert, Modify or delete rows or fields.
- Search several tables for specific information and returns the result in specific order.
- Modify security information.

## 4.0 Structure Query Language (SQL)

---

Note: Commands in SQL are not case-sensitive. But generally conversation is write a keywords as a capital and other should be in small letter.

### Data Manipulation Language in SQL:

SQL provides the following basic data manipulation statements: SELECT, UPDATE, DELETE and INSERT.

The select statements:

- The SELECT statement is most commonly used SQL statement. It is only a data retrieval statement in SQL.

- The basic syntax for select statement is

```
SELECT [DISTINCT / ALL ] <attributes> 1  
      FROM <relations> 2  
      [WHERE <predicate>] 3
```

1. <attribute> -> columns name

2. <relations> -> tables name

3. <predicated> -> conditions

- SELECT, FROM are necessary clause.
- WHERE is optional clause.
- DISTINCT / ALL are optional clause.
- SELECT clause used to list the attributes that required in the result in query.
- FROM clause list the relation/s from where specified attributes are to be selected.
- WHERE clause are used to specify the condition/s while we require retrieving particular data. One or more condition can be specified using where clause by using SQL logical connectives can be any comparison operators <, <=, >, >=, = and < >. SQL also includes BETWEEN comparisons.
- DISTINCT key word is used to eliminate duplicate value.
- ALL key word is used to explicitly allow duplicates.

Example: Assumed simple relational database is as follows.

customer_id	customer_name	customer_address
C001	Smith	Kathmandu
C002	John	Bhaktapur
C003	Semi	Lalitpur
C004	Ivan	Kathmandu

The customer table

account_no	balance
A101	900
A102	300
A103	200
A104	700

The account table

customer_id	account_no
C001	A001
C002	A002
C003	A004
C004	A004

The depositor table

**a. Find all customer names.**

```
SELECT customer_name FROM customer;
```

**b. Find the different customer address (location).**

```
SELECT DISTINCT customer_address FROM customer;
```

**c. Find all address of customer.**

```
SELECT ALL customer_address FROM customer;
```

**d. Find customer\_id and its corresponding customer name.**

```
SELECT customer_id, customer_name FROM customer;
```

## **4.0 Structure Query Language (SQL)**

---

---

**e. Find customer detail.**

```
SELECT *FROM customer (* indicates all attributes)
```

**f. List name and address of customer who stay in "KATHMANDU".**

```
SELECT customer_name, customer_address, FROM customer  
WHERE customer_address = "KATHMANDU";
```

**g. List all customer whose name should be "smith" and address should be "Kathmandu".**

```
SELECT customer_name FROM customer  
WHERE customer_name = 'smith'  
OR customer_address = 'Kathmandu'
```

**h. What would be the output if statement like**

```
SELECT customer_name FROM customer  
WHERE customer_name FROM customer  
OR customer_address = 'Kathmandu'
```

**i. List account no, balance whose balance is between 200 to 700.**

```
SELECT account_no, balance FROM account  
WHERE balance BETWEEN 200 AND 700;
```

**j. What happen if we execute the statement?**

```
SELECT account_no, balance FROM account  
WHERE balance NOT BETWEEN 200 AND 700;
```

**k. Write SQL statement for (i) using only AND logical connectives and comparison operatives.**

```
SELECT account_no, balance FROM account  
WHERE balance <= 700 AND balance >= 200;
```

Note: We can retrieve the information from multiple tables; there should be a common attribute between two tables. i.e., table should be related and we require to join condition.

**l. List the customer\_id, account\_id and balance whose balance is more than 300.**

```
SELECT depositor.customer_id, depositor.account_no, account.balance  
FROM depositor, account  
WHERE depositor.account_no = account.account_no  
AND account.balance > 300;
```

**m. List all customers and corresponding balance.**

```
SELECT c.customer_name, a.balance  
FROM customer c, account a, depositor d  
WHERE d.account_no = a.account_no  
AND d.customer_id = c.customer_id;
```

**Renaming attribute and relations:**

- In previous example relations customer, account, depositor are renamed respectively c, a and d.

## 4.0 Structure Query Language (SQL)

---

- We can also rename attributes, it is required when we are taking attribute from multiple column or we need arithmetic operations in the statement or when we need to give appropriate name for (column name) attribute name.
- To rename attribute SQL provides as clause or we can simply rename attribute or relation without as clause.

Examples:

- a. SELECT account\_no as account number FROM account;  
OR  
SELECT account\_no "Account number" FROM account;
- b. SELECT account\_no, balance, balance+ (balance\*0.05) as Account Number, Balance,  
Increase salary FROM account;  
OR  
SELECT account\_no "Account number", balance "Balance:",  
Balance+(balance\*0.05) "Increased salary" FROM account;
- c. SELECT c. customer\_name, a. balance  
FROM customer c, account a, depositor d  
WHERE d. account\_no = a. account\_no  
AND d.customer\_id = c.customer\_id;  
OR  
SELECT c. customer\_name, a. balance  
FROM customer as c, account as a, depositor as d  
WHERE d.account\_no = a.account\_no  
AND d.customer\_id = c.customer\_id;

### String operations:

String pattern matching operation on SQL can be performed by `like' operator and we can describe the patterns by two spherical character.

1. Percent (%): matches any substrings.
2. underscore (\_): matches any characters.

Example:

`I%' matches any string beginning with I.  
IVAN -> valid / match.  
INDIA -> valid / match  
NEPALI -> invalid / does not match.

` % VAN% match any string containing `VAN' as substring.  
IVAN, Mr IVAN, DEVAN, DEVANGAR are all valid.  
`---' matches any strings of exactly three character.  
`---%' matches any string of atleast three character.

Moreover,

Like `ab\%cd%' matches all string beginning with "ab%cd".  
Like `ab\\cd%' matches all string beginning with "ab\cd".

Problems:

## **4.0 Structure Query Language (SQL)**

---

---

List those customers whose name begin with character `S'

```
SELECT customer_name FROM customer.  
      WHERE customer_name LIKE `S%';
```

**Note:**

customer\_name like `S%H'

- List all customer name whose name begin with `S' and end with `H'

customer\_name like` ----%N'

- List all customer name whose name must contain at least five character and end with character `N'

### **Ascending and descending records in SQL:**

- ORDER BY clause used for ascending or descending records( or list items).
- To specify sort order we may specify desc for descending\_order or asc ascending orders. By default order by clause list item in ascending order.
- Moreover, ordering can be performed on multiple attributes.

Example: 1

```
SELECT distinct customer_name FROM customer ORDER BY customer_name;
```

- Lists name of customer in alphabetic order by customer name.

Example: 2

```
SELECT DISTINCT customer_name FROM customer ORDER BY customer_name  
DESC;
```

- Lists name of customer in descending alphabetic order.

Note: select from customer order by 2;

Here 2 indicates second column in table "customer". This SQL statement is equivalent to first example's SQL statement.

Example: 3

Suppose want list account information in descending order by balance but if say some balance are same and in such case if we want to order account information by order\_no in ascending order then we have order record by performing ordering on multiple attributes. The SQL statement likes,

```
SELECT *FROM account  
      ORDER BY balance DESC, account ASC;
```

### **Set operation**

- basic set operation are union(u), intersection(n) and difference(\_). These operation also can be performed by using union, intersection and minus (except) clause respectively.

## 4.0 Structure Query Language (SQL)

---

### The union operation

- the union operation can be performed by using union clause.

Example: consider two reactions

client_id	name
C001	Ammit
C002	Ajay
C003	Rohit
C004	Ammit

supplier_id	name	city
S001	ASHOK	Kathmandu
S002	MANISH	Bhaktapur
S003	MANOJ	Kathmandu
S004	MANISH	Bhaktapur

The table client

The table supplier

**List the id and name of the client and supplier who stay in city 'Kathmandu'.**

Select supplier\_id "ID", name "Name" from supplier  
where city = 'Kathmandu'

UNION

SELECT client\_id "ID", name "Name" from client  
where city = 'Kathmandu'

- Proceed as follow:  
Output from 1<sup>st</sup> SQL statement

ID	Name
S001	Ashok
S002	Manoj

Output from 2<sup>nd</sup> SQL statement

ID	Name
C001	Ammit
C002	Ammit

Hence, the resulting output is

ID	Name
C001	Ammit
C002	Ammit
C003	Ashok
C004	Manoj

Note: if we retrieve only one column say name without duplicate name then corresponding statement like

Select name from supplier where city = 'Kathmandu'

UNION

Select name from client where city = 'Kathmandu';

- o this is unlike select clause, union operation automatically eliminates duplicates. If we want to retain all duplicates, we must replace union all.

## 4.0 Structure Query Language (SQL)

Example:

```
select name from supplier where city = 'Kathmandu'.
```

The output would be

Name
Ammit
Ammit
Ashok
Manoj

### *The intersection operation*

- the intersection operation can be performed by using INTERSECT clause
- consider relation as follow:

SALESMAN

salesman_id	name	city
S001	Manish	Kathmandu
S002	Manoj	Lalitpur
S003	Ammit	Bhaktapur
S004	Rabin	Kathmandu

order_no	Order_date	salesman_id
0001	10-JAN-98	S001
0002	12-FEB-98	S002
0003	13-FEB-98	S001
0004	18-MAR-98	S001
0005	19-MAR-98	S002

The salesman table  
sales\_order table

The

Retrieve salesman name who stay in Kathmandu and who must sales at least order.

```
SELECT salesman_id, name
      from salesman
     where city = 'Kathmandu'
```

salesman_id	name
S001	Manish
S004	Rabin

### INTERSECT

```
SELECT salesman. salesman_id
      from salesman, sales_order
     Where      salesman_id
           sales_order. salesman_id;
```

salesman_id	name
S001	Manish
S002	Manoj
S001	Manish
S002	Manoj

The resulting output is

salesman_id	name
S001	Manish

- The INTERSECT operation also automatically eliminates duplicates. So, here only one record is delayed in output. If we want to retain all duplicates we must replace INTERSECT by INTERSECT ALL.

```
SELECT salesman_id, name from SELECT salesman
                                where city = 'Kathmandu'
INTERSECT ALL
SELECT salesman. salesman_id, salesman name
      from salesman, sales_order
 WHERE salesman. salesman_id = sales_order salesman_id;
```

## 4.0 Structure Query Language (SQL)

---

---

### ***The difference operation***

The difference operation can be perform in SQL by using except or minus clause.

Example: in previous example, find the salesman\_id, name who stay in Kathmandu but they do not sales any order.

```
SELECT salesman_id, name from salesman
    where city = 'Kathmandu'
EXCEPT
SELECT salesman. salesman_id, salesman name
    from salesman, sales_order
        Where salesman. salesman_id = sales_order. salesman_id;
        OR
SELECT salesman_id, name from salesman
    where city = 'Kathmandu'
MINUS
SELECT salesman. salesman_id, salesman name
    FROM salesman, sales_order
        Where salesman salesman_id = sales_order salesman_id;
```

The output is

salesman_id	name
S004	Rabin

NOTE: Except operation also automatically eliminates duplicates so it want to return all duplicates, we must write EXCEPT ALL instead of EXCEPT.

Problem: consider a relation schema as follow

```
branch(#branch_name, branch_city, assets)
account(#account_number, branch_name, balance)
customer(#customer_name, customer_street, customer_city)
depositor(customer_name, account_number)
loan(#loan_number, branch_name, amount)
borrower(customer_name, loan_number)
```

1. Find all customer who have a loan, account or both at the bank .

```
SELECT customer nameFrom depositor
UNION
SELECT customer name From borrower;
```

2. Find all customers who have both loan and account at the bank.

```
SELECT DISTINCT customer name from depositor
INTERSECT
SELECT DISTINCT customer name from borrower;
```

3. Find all customers who have account but no loan at the bank.

```
SELECT DISTINCT customer name from depositor
EXCEPT
SELECT customer name from borrower;
```

## **4.0 Structure Query Language (SQL)**

---

---

### ***Aggregate Function***

Aggregate functions are those functions that take a set of values as input and return a single value. SQL consist many built in aggregate function. Some are:

1. AVERAGE: AVG
2. MAXIMUM: MAX
3. MINIMUM: MIN
4. TOTAL: SUM
5. COUNT: COUNT

The input to AVG and SUM must be a set of numbers and other aggregate function can be operate by non numeric data types, it may be strings, not necessary numbers.

#### **AVG:**

- Syntax: AVG (<DISTINCT\ALL>; n)  
• returns average of n, ignoring null values.

Example: find the average balance in Kathmandu branch.

```
SELECT AVG (balance) From account  
WHERE branch_name = 'KATHMANDU';
```

There would be a situation that we may have to use aggregate function not only a single set of tuples we may have to use with group of set of tuples, we can specify this by using GROUP BY clause in SQL. That is, group by clause specifies group rows based on distinct values that exist in specified column when we use GROUP BY clause we can not use WHERE clause to specify condition, we must have to use HAVING clause to specify the condition. That is, GROUP BY and HAVING clause. But GROUP BY or HAVING clause act on record sets rather than individual records.

Example: find the average account balance at each branch

```
SELECT branch_name , avg (balance)  
      FROM account  
      GROUP BY branch_name;
```

#### **MIN:**

- Syntax: MIN (<DISTINCT\ALL>; n)  
• returns minimum value of n.

Example: find the minimum balance of each branch.

```
SELECT branch_name, min (balance) "Minimum Balance"  
      FROM account  
      GROUP BY branch_name;
```

#### **MAX:**

- Syntax: MAX (<DISTINCT ALL>; n)  
• returns maximum value of n

Example: find the maximum balance in each branch.

```
SELECT branch_name, max (balance) "Maximum Balance"  
      FROM account
```

## **4.0 Structure Query Language (SQL)**

---

---

GROUP BY branch\_name;

### **COUNT:**

Syntax: COUNT (<DISTINCT ALL>; n)

- returns numbers of rows where n is not null.

NOTE: COUNT (\*)

- returns numbers of rows in the table, including duplicates and those with nulls.

Example: find the numbers of depositor for each branch.

NOTE: each depositor may have numbers of account so we must count depositor only once thus we write query as below:

```
SELECT branch_name, count (DISTINCT Customer_name)
      From depositor, account
            WHERE depositor account number = account. Account_number
                  GROUP BY branch_name;
```

NOTE: if we need to specify the condition (predicates) after GROUP BY clause, we need having clause.

PROBLEM: find only those branches where the average account balance is more than 1200.

```
SELECT branch_name, AVG (balance) FROM account
      GROUP BY branch_name having AVG (balance) > 1200;
```

PROBLEM: find the no. of customer in customer table.

```
SELECT COUNT (*) FROM Customer;
```

NOTE: If a WHERE clause and HAVING clause both appears in the same query, SQL executes predicate in the WHERE clause first and if it satisfied the only it executes predicate of GROUP BY clause.

PROBLEM: Find the average balance for each customer who lives in KATHMANDU and has at least three accounts.

```
SELECT depositor customer_name, AVG (balance)
      FROM depositor, account , customer
            WHERE depositor account_number = account account_number
                  Depositor customer_name = customer customer_name
                  Customer_city = 'KATHMANDU'
            GROUP BY depositor, customer_name
                  HAVING COUNT (DISTINCT depositor account_number) > = 3;
```

### **SUM**

Syntax: SUM ([DISTINCT/ ALL] n)

- return sum of value of n

Example: find total loan amount for each branch

```
SELECT branch_name, SUM (amount)
      FROM loan
            GROUP BY branch_name;
```

## **4.0 Structure Query Language (SQL)**

---

---

### **NULL VALUES**

- In SQL, NULL values all to indicate absence of information for the value of attribute.
- SQL provides special key word NULL in a predicate to test for NULL values. Not NULL in predicate use to test absence of NULL values.

Example: find the balance of each branch whose balance is not empty.

```
SELECT branch_name, balance  
      FROM account  
 WHERE balance is NOT NULL;
```

Example: List the account number, branch name and balance whose balance is empty.

```
SELECT *FROM account  
 WHERE balance is NULL;
```

- The feature of SQL that handles NULL values has important application but some time it gives unpredictable result. For example, an arithmetic expression involving (+, -, \*, or /), if any of the input values is NULL value (except IS NULL Q IS NOT NULL).
- If NULL values exist in the processing of aggregate operation, it makes process complicated.  
Example: SELECT AVG (amount) FROM loan;
- This query calculates the average loan amount that is not empty so if there exist empty amount then calculated average amount is not valid. Except COUNT(\*) function all aggregate function ignores NULL values in input.
- The COUNT ( ) function return if count value is empty and all other aggregate function returns NULL if it found empty value.

Example: consider a table 'emp' as below:

Empno	Sal	Comm
10	100	
20	200	50
30	300	20

Suppose the SQL statement are as below

- a. SELECT COUNT (\*) FROM emp;  
returns count is equal to 3  
That is           count(\*)  
-----  
                  3
- b. SELECT COUNT (comm.) FROM emp WHERE empno = 10;  
returns           count(comm.)  
-----  
                  0
- c. SELECT SUM(COMM) FROM emp WHERE empno = 10;  
return           sum(comm.)  
-----  
                  (nothing)

## 4.0 Structure Query Language (SQL)

---

- d. SELECT SAL+COMM FROM emp WHERE empno = 10;  
return            sal+comm.  
-----  
                  (nothing)
- here the result is unpredictable. In this case result should be 100. To handle such unpredictable situation SQL provides NVL ( ) function  
Example: SELECT sal+nvl(comm,0)/100
  - nvl function returns 0 when comm is found empty. If user do not specify the value for any column (attribute) SQL place null values in these columns. The null value is different from zero. That is null value is not equivalent to value zero.
  - A NULL value will evaluate to NULL in any expression.  
Example: NULL multiply by 10 is NULL.
  - If the column has a NULL value, SQL ignores the unique Foreign key, check constraints that are attached to the column.
  - If any field define as NOT NULL, it does not allow to ignore this field, user must insert value, that is NOT NULL is itself a constraint while it specify in table.

### Nested Subqueries

- SQL provides sub\_query facility. A sub\_query is a SQL statement that appears inside another SQL statement. It is also called nested sub queries or simply nested query.
- Sub\_query use to perform tests for set membership, make set comparisons and determine set cardinality.
- Sub\_query can be used with SELECT, INSERT, UPDATE and DELETE statement.

Example: find all branch name where depositor account number is 'A005' .

```
SELECT branch_name FROM account
WHERE account.account_number = (SELECT account_number FROM
depositor
WHERE account_number
= 'A005');
```

- When sub\_query return more than one values the we required to test whether value written by first query is match/exist or not within values return by sub\_query.
- IN and NOT IN connectives are useful to test in such condition. That is IN connectives use to test for set membership and NOT IN connectives use to test for absence of set membership.

Example: find those customers who are borrowers from the bank and who are also account holder.

```
SELECT DISTINCT customer_name FROM borrower
WHERE customer_name IN (SELECT customer_name FROM
depositor);
```

Example: find all customer who do have loan at the bank but do not have an amount at the bank.

```
SELECT DISTINCT customer_name FROM borrower
WHERE customer_name NOT IN (SELECT customer_name FROM
depositor);
```

## **4.0 Structure Query Language (SQL)**

---

Example: list the name of customers who have a loan at the bank and whose name neither SMITH nor JONE

```
SELECT DISTINCT Customer_name FROM borrower  
WHERE customer_name NOT IN ('SMITH', 'JONE');
```

Example: find all customers who have both an account and loan at Kathmandu branch.

```
SELECT DISTINCT Customer_name FROM borrower, loan  
WHERE borrower loan number = loan loan_number and  
branch_name = 'KATHMANDU'  
AND (branch_name, customer_name) IN (SELECT  
branch_name, customer_name  
FROM depositor account  
WHERE depositor account_number = account  
account_name);
```

### **SET COMPARISON**

Nested sub\_query have an ability to compare set.

Example: Find the names of all branches that have assets greater than those of at least one branch located in 'Kathmandu'.

The simple SQL statement is

```
SELECT DISTINCT B1 branch_name FROM branch B1, branch B2  
WHERE B1 assets > B2 assets  
AND B2.branch_city = 'Kathmandu'.
```

The same query can be written by using subquery as

```
SELECT branch_name FROM branch  
WHERE assets > some (select assets FROM branch  
WHERE branch_city = 'Kathmandu');
```

- here, >some comparison in the where clause of the outer value return by sub\_query.
- SQL also allow <some, >=some, =some and < > some comparison
- Not that = some is identical to IN. but < > some is not same as NOT IN.

Example: Find the names of all branches that have an assets value greater than that of each branch in 'KATHMANDU'.

NOTE: The construct > all corresponds to the phase 'greater than all'

```
SELECT branch_name FROM branch  
WHERE assets > all (SELECT assets FROM branch  
WHERE branch city = 'KATHMANDU')
```

NOTE: SQL also allow <all, <=all, >=all, =all and < >all comparison.

Example: find the branch that has the highest average balance.

NOTE that we can not use MAX {AVG (balance)}, since aggregate function can not be composed in SQL. So we first need to find all average balances and need to nest

## **4.0 Structure Query Language (SQL)**

---

it as subquery of another query that finds those branches for which average balance is greater than or equal to all average balances.

```
SELECT branch_name FROM account
    GROUP BY branch_name
        HAVING AVG (balance) >=all (SELECT AVG (balance) FROM account
            GROUP BY
branch_name);
```

### **TEST EMPTY RELATIONSHIP**

SQL has a feature for testing whether a subquery return any value or not. The exists construct returns true if subquery returns values.

Example: find all customers who have both an account and loan at the bank.

```
SELECT customer_name FROM borrower
WHERE exists (SELECT *FROM depositor
                WHERE depositor.customer_name = borrower
customer_name);
```

We can test non existence of values (tuples) in sub\_query by using not exists construct.

Example: find all customers who have an account at all branches located in 'KATHMANDU'.

```
SELECT DISTINCT d1.customer_name
    FROM depositor as d1
WHERE not exists { (SELECT branch_name FROM branch
                    WHERE branch_city = 'KATHMANDU')
Except
    (SELECT d2. branch_name FROM depositor as d2, account as a
        WHERE d2. Account_no. = a. Account_no.
            AND d1. Customer_name = d2.
Customer_name)};
```

### **Test for the absence of Duplicate Tuples**

SQL has a feature for testing whether the subquery has any duplicate Tuples in its results.

The UNIQUE construct true if a subquery contains no duplicate Tuples.

Example: find all customers who have at most one account at the KATHMANDU branch.

```
SELECT d.customer_name FROM depositor d1
WHERE UNIQUE (SELECT d2. customer_name FROM account depositor d2
                WHERE d1. customer_name = d2. customer_name
                    AND d2. account_no. = account. Account_no.
                        AND account. Branch_name = 'KATHMANDU');
```

NOTE: using NOT UNIQUE construct, we can test the existence of duplicate tuples.

Example: find all customers who have at least two account at the KATHMANDU branch.

## **4.0 Structure Query Language (SQL)**

---

---

```
SELECT DISTINCT d1.customer_name FROM depositor as d1
    WHERE UNIQUE (SELECT d2. customer_name FROM account depositor d2
        WHERE d1. customer_name = d2. customer_name
        AND d2. account_no. = account. Account_no.
        AND account. Branch_name = 'KATHMANDU');
```

### **Complex Queries**

There are several way of composing query: derived relation and the with clause are ways of composing complex queries.

#### **Derived Relation**

- SQL have a feature that it allow sub\_query expression to used in the FROM clause.
- If we use such expression then we must give result relation name and we can remove the attributes.

Example: find the average account of those branches where the average account balance is greater than 1200.

```
SELECT branch_name, avg_balance
    FROM {SELECT branch_name, avg (balance) FROM account GROUP BY
branch_name}
        AS branch_avg (branch_name, avg_balance)
            WHERE avg_balance > 1200;
```

#### **The with clause**

- The with clause introduced in SQL: 1999 and is currently supported by only some database.
- The with clause makes query logic clear.

Example: find all branches where the total account deposit is less than the average deposits at all branches.

```
WITH branch_total (branch_name, value) as
    SELECT branch_name, SUM (balance) FROM account
        GROUP BY branch_name
    WITH branch_total_avg (value) as
        SELECT avg (value) FROM branch_total
    SELECT branch_name FROM branch_total, brnch_total_avg
        WHERE branch_total.value >= branch_total_avg.value;
```

## **UPDATE STATEMENT**

- The UPDATE statement is used to modify one or more records in specified relation. The records to be modify are specified by a predicate in the WHERE clause and new value of the column (s) to be modified is specified by a SET clause.

The syntax is

```
UPDATE <relation>
    SET <attribute with new value>
    [WHERE <predicate>];
```

Example: increase the balance of all branches by 5%

```
UPDATE account
```

## **4.0 Structure Query Language (SQL)**

---

---

```
SET balance = balance * 1.05;  
OR  
UPDATE account  
SET balance = (balance) + (balance * 0.05);
```

Example: increase balance of those branches whose current balance is less than or equal to 1000 by 5%

```
UPDATE account  
SET balance = balance * 1.05  
WHERE balance <= 1000;
```

Example: decrease the balance by 5% on accounts whose balance is greater than average.

```
UPDATE account  
SET balance = balance - (balance * 0.05)  
WHERE balance > {SELECT average (balance) FROM account};
```

NOTE: SQL provides case construct, which can be perform multiple updates with a single UPDATE statements

The syntax is:

```
Case  
When predicate 1 then result 1  
When predicate 2 then result 2  
When predicate n then result n  
Else result  
END
```

Example: UPDATE account

```
SET balance = case  
When balance <= 1000 then balance * 1.05  
else balance * 1.06  
end;
```

### **DELETE STATEMENT**

The DELETE statement use to delete one or more records from relations. The records to be deleted are specified by the predicate in the WHERE clause.

The Syntax is:

```
DELETE <relation> [WHERE <Predicate>]
```

Note: Delete statement can operate only one relation. It can not delete records of multiple relation.

Example: Delete all records from loan relation.

```
DELETE FROM loan;
```

Example: Delete all records from account relation whose branch is located in Kathmandu.

```
DELETE FROM account  
WHERE branch_name = 'KATHMANDU';
```

## **4.0 Structure Query Language (SQL)**

---

Example: Delete all loan with loan amount between 1000 and 11500;

```
DELETE FROM account  
WHERE branch_name IN (SELECT branch_name FROM branch)  
      WHERE branch_city = 'KATHMANDU');
```

Example: Delete all records of account with balance below the average

```
DELETE FROM account  
WHERE balance < (SELECT avg (balance) FROM account);
```

### **INSERT STATEMENT**

The INSERT statement used to insert a new records into a specified relation

Syntax:

```
INSERT INTO <relation>  
VALUES (<values list>)
```

Another form:

```
INSERT INTO <relation> (<target columns>)  
VALUES (<values list>)
```

The attributes values that are going to insert must be match order by corresponding attributes and also must be matched data type of value and corresponding attribute data type.

Example: insert one record in account relation.

```
INSERT INTO account  
VALUES ('A_0009', 'LALITPUR', 1500);  
OR  
INSERT INTO account (account_no., branch_name, balance )  
VALUES ('A_0009', 'LALITPUR', 1500);
```

we can also insert records on the basis of query

Example: INSERT INTO account

```
SELECT loan_number = 'KATHMANDU';
```

It inserts the records in account relation taking account\_number, brancg\_name from loan relation whose branch is located in Kathmandu and for all records balance is constant 500.

Example: INSERT INTO depositor

```
SELECT customer_name, loan_number FROM borrower, loan  
WHERE borrower . loan_number = loan . loan_number  
AND branch_name = 'KATHMANDU';
```

Insert Tuple (customer\_name, loan\_number) into the depositor relation for each customer who has a loan in Kathmandu branch with loan number.

Example: INSERT INTO account

```
SELECT *FROM account;
```

Insert infinite number of Tuples.

### **Joined Relations**

- one of the most powerful feature of SQL is its capability to gather and manipulate data from several relations.

## **4.0 Structure Query Language (SQL)**

---

- If SQL does not provide this feature we must have to store all the data elements in a single relations for each application. We have to store same data in several relations.
- The join statements of SQL enables to design smaller, more specific relations that are easier to maintain than larger relations.
- There are several methods for joining relations. Some methods are not useful for application and some are very useful.

### **1. Cross Join**

Joins two or more tables without relation between them or without condition in the where clause. The result is the Cartesian product of two or more table's attributes

Examples: consider two relations

Table 1:

<u>row</u>	<u>remarks</u>
1.	Table 1
2.	Table 1

Table 2: <u>row</u>	<u>remarks</u>
1.	Table 2
2.	Table 2

The cross join statement is

```
SELECT *FROM Table 1,Table 2;
```

Output is:

<u>row</u>	<u>remarks</u>	<u>row</u>	<u>remarks</u>
1	Table 1	1	Table 2
1	Table 1	2	Table 2
2	Table 1	1	Table 2
2	Table 1	2	Table 2

- cross join is normally not useful but it illustrates the basic combining property of all join types.

### **Equi Join (natural inner join)**

Equi joins methods join relations based on the equality. It has very important application in commercial database application.

Example: consider

Relation : dept

#empno	E name	job	sal	comm	Dept. no
E001	SMITH	Manager	7000	500	10
E002	JONE	Engineer	6000	3000	20
E003	MICLE	Engineer	5000	2000	20
E004	JACK	Accountant	3000	500	40

relation: emp

#dept. no	Depo. name	Loc
10	Management	Kathmandu
20	Technical	Kathmandu
30	Marketing	Bhaktapur
40	Account	Lalitpur

Example: list all employee name, department name and salary

## 4.0 Structure Query Language (SQL)

---

```
SELECT e.ename, d.dname FROM emp e, dept d  
WHERE e.deptno = d.deptno;
```

Output:

ename	Dname
SMITH	MANAGEMENT
JONE	TECHNICAL
MICLE	TECHNICAL
JACK	ACCOUNT

We can further qualify this query by adding more condition on where clause.

```
Example: SELECT e.ename, d.dname  
        FROM emp e , dept d  
        WHERE e.deptno = d.deptno  
        AND e.sal >5000      ORDER By e.ename;
```

Example: consider relations

Loan_number	Depo. name	Loc
L_170	Management	Kathmandu
L_230	Technical	Kathmandu
L_260	Marketing	Bhaktapur

Relation: loan

Customer_name	Loan_nuber
JONE	L_170
SMITH	L_230
MICLE	L_155

relation: borrower

```
SELECT *FROM loan, borrower  
WHERE loan.Loan_number = borrower.loan_number;
```

### Non – Equi join (Outer join)

SQL also supports non equi\_join. That is, this method joins tables (relations) based on non equality. But equi\_join is far more common than non equi\_join

```
e.g. Select e.ename, d.dname, d.dname d.deptno  
      FROM emp e, dept d  
      WHERE e.deptno > d.deptno;
```

Output:

Ename	dname	dept no
JONE	MANAGEMENT	10
MICLE	MANAGEMENT	10
JACK	MANAGEMENT	10
JACK	TECHNICAL	20
JACK	MARKETING	30

Here, this information is not so useful.

# How from the output? Lets look

Case 1: equi\_join

```
SELECT e.ename, d.dname, e.deptno "emp deptno", d.deptno "deptno"  
      FROM emp e, dept d  
      WHERE e.deptno;
```

OUTPUT:	ename	dname	emp deptno	deptno
	SMITH	MANAGER	10	10

## 4.0 Structure Query Language (SQL)

---

JONE	TECHNICAL	20	20
MICLE	TECHNCAL	20	20
JACK	ACCOUNT	40	40

REMARKS: output is selected from the Cartesian product of two relations

Emp (e.ename, e.deptno)  
Dept (d.dname, d.deptno)  
such that both department no is equal only.

Case 2: (Non equi join)

```
SELECT e.ename, d.dname, e.deptno "emp deptno", d.deptno "dept deptno"  
FROM emp e, dept  
WHERE e. dept> d.dept no;
```

Output:	ename	dname	empdeptno	detno
	JOHN	Management	20	10
	MICHAEL	Management	20	10
	JACK	Management	40	10
	JACK	Technical	40	20
	JACK	Markeing	40	30

..  
Remarks:

Output is selected from the Cartesian product of two relations

emp ( e.ename, e.dept no)  
Dept (d.dname, d.deptno)

Such that emp table of department no is greater than department table of department no.

Case 3 (Non equi join)

```
SELECT e.ename , d.dname e.dept no "emp deptno"  
FROM emp e , dept d  
WHERE e. deptno> 10;
```

Output:

ename	dname	Emp deptno	
JONE	MANAGEMENT	20	>10
MICLE	MANAGEMENT	20	>10
JACK	MANAGEMENT	40	>10
JONE	TECHNICAL	20	>10

REMARKS: output is selected from the Cartesian product of two relation

Emp (e.ename, e.deptno)  
Dept (e.ename, e.deptno)

Such that emp table of deptno is always greater than 10.

Three types of outer join

1. Left outer join
2. Right outer join
3. Full outer join

## 4.0 Structure Query Language (SQL)

---

Consider two relations

Loan

Loan_no.	Branch_name	amount
L-170	KATHMANDU	3000
L-230	BHAKTAPUR	4000
L-260	LALITPUR	1700

Borrower

Customer_name	Loan_number
JONE	L-170
SMITH	L-230
MICLE	L-155

- a. `SELECT loan . loan_number, loan . branch_name, loan . amount, borrower . loan_number loan`

Left outer join borrower on loan . loan\_number = borrower . loan\_number

Loan_no.	Branch_name	amount	Customer_name
L-170	KATHMANDU	3000	JONE
L-230	BHAKTAPUR	4000	SMITH
L-260	LALITPUR	1700	null

Remarks: Tuple from the left hand side relation that do not math any Tuple in the right side relation are padded with null.

`SELECT loan. loan_number, loan . brach_name, loan.amount, borrower . Customer_name loan`

`right outer join borrower on loan.loan_number = borrower. Loan_number;`

Loan_no.	Branch_name	amount	Loan_no.	Customer_name
L-170	KATHMANDU	3000	L-170	JONE
L-230	BHAKTAPUR	4000	L-230	SMITH
L-155	null	null	null	MICLE

Remarks: Tuples from the right hand-side that do not match any Tuple in the left hand side relation are padded with nulls.

`SELECT loan. loan_number, loan. branch_name, loan. amount, borrower . customer_name loan`

`Full outer join burrower on loan.loan_number = borrower. Loan_number;`

loan number	branch name	amount	customer name
L - 170	Kathmandu	3000	JONES
L - 230	Bhaktapur	4000	SMITH
L - 260	Lalitpur	1700	NULL
L -150	NULL	NULL	MICHALE

*Self join*

- In some situation, we may need necessary to join a table to itself as we are joining two separate tables, this is known as self-join
- In a self join two rows from the same table combine to form a result row.
- Example: Consider a relation `employee` as below.

Empno	name	manager no
E001	Smith	E002
E002	Michale	E005
E003	John	E004
E004	Ivan	
E005	Scott	

## **4.0 Structure Query Language (SQL)**

---

---

Retrieve the names of employees and the names of their respective manager from the employee relation.

```
SELECT emp.name , mgr.name "Manager"  
      FROM employee emp, employee mgr  
     WHERE emp.Manager no = mgr.emp_no;
```

Output:

Name	Manager
Smith	Michael
Michael	Scott
John	Ivan

Process:

EMP			MGR		
Emp no	name	manager no	emp no	name	manager no
E001	Smith	E002	E001	Michael	E002
E002	Michael	E005	E002	Scott	E005
E003	John	E04	E003	Ivan	E004

Name	Manager
Smith	Michael
Michael	Scott
John	Ivan

### **Data definition Language in SQL**

- In SQL, DDL specifies set of relations (tables) in a database.
- SQL DDL also allows to specify
  - Integrity constraints.
  - Index on relations
  - Security and authorization for each relation
  - Physical storage structure of each relation
- Basic statement in Data Definition Language are CREATE, DROP, ALTER

### **Some Domain types in SQL ( Data type in SQL )**

CHAR(n): fixed length character string with user specified length n.

VARCHAR2(n) : A variable length character string with user specified length n. Full form is character varying and to indicate version of the domain.

NUMBER(n): holds fixed number specified length n.

NUMBER(P,S): holds fixed or floating point numbers

P determines the maximum length of data, and S

Determines the number of places to the right of decimal.

If S is not specified then default is zero ; in such case or specified 0, it can not hold floating point number.

INT : An integer , small int

FLOAT(n) : A floating point number , with precision of at least n digits.

DATE : Represents date and time. The standard format is DD-MM-YY

LONG: Used to store variable length character strings containing up to 2 GB.

RAN/ LONG RAW : Used to store binary data such as digitized picture or image. It can contain up to 2 GB.

## **4.0 Structure Query Language (SQL)**

---

---

### ***Schema Definition in SQL***

- CREATE TABLE command is used to create relation

Syntax :

```
CREATE TABLE <relation name>
(    A1D1 , A2D2, ......., AnDn ,
    [< integrity constraint 1>]
    [< integrity constraint K>]
);
```

- Here, Ai is the name of attributes.
- Di is the domain type (or data type)

And integrity constraint includes:

#### **PRIMARY KEY :**

Primary key is an attribute or combination of multiple attributes that uniquely identifies records.

If a primary key is a combination of multiple attributes called composite primary key. A primary key attributes are required NOT NULL AND UNIQUE. That is primary key attribute cannot be left null and it cannot contain duplicate values.

NOT NULL / UNIQUE: Attribute can be specified NOT NULL attribute or unique attribute.

**FOREIGN KEY:** Any column (attribute) of table (relation) can be specified as a foreign key if it is a common attribute between relations where we are going to establish a relationship.

- In one relation (master table) it should be primary key and in another table (detail table) some attribute should be foreign key.
- Primary key and foreign key together used to establish the relationship between the two relations.
- The concept of primary key and foreign key is very important in RDBMS.

CHECK(P) : Check clause specifies the predicate P that must satisfy specified condition.

Example: SQL data definition for the simple banking database.

```
CREATE TABLE customer
Customer_name VARCHAR2(20) NOT NULL,
Customer_location VARCHAR2 (20)
Constraint PK_Cname Primary key (Customer_name));
```

```
CREATE TABLE branch
(
branch_name VARCHAR 2(15),
branch_city VARCHAR2(30) DEFAULT "KATHMANDU",
assets NUMBER (5) ,
CONSTRAINT PK_branch_name PRIMARY KEY (branch_name),
CONSTRAINT ch_acbal CHECK (balance >=0)
);
```

## **4.0 Structure Query Language (SQL)**

---

---

```
CREATE TABLE depositor
(
    customer_name VARCHAR2(20),
    account_no CHAR(10)
    CONSTRAINT fk_depositor_cname
        FOREIGN KEY(customer_name) REFERENCES customer,
    CONSTRAINT PK_cname_accno PRIMARY KEY (customer_name, account_no)
);

CREATE TABLE loan
(
    loan_no CHAR(10) PRIMARY KEY ,
    branch_name VARCHAR2(15) NOT NULL,
    amount NUMBER (5),
    CONSTRAINT fk_loan_branch_name
        FOREIGN KEY (branch_name) REFERENCES branch,
    CHECK (amount>=0)
);
CREATE TABLE borrower
(
    customer_name  VARCHAR2(20),
    loan_no        CHAR(10),
    CONSTRAINT fk_cname FOREIGN KEY (customer_name) REFERENCES customer
);

.
```

Example of using UNIQUE key and DEFAULT value

e.g. CREATE TABLE student

```
( 
    student_id NUMBER (3) PRIMARY KEY,
    name        CHAR (20) UNIQUE,
    degree      CHAR (15) DEFAULT 'Master',
    CHECK (degree IN ('Bachelors', 'Master', 'Doctorate'))
);
```

### ***Drop statement***

- Drop table statement used to drop the relation.

Syntax:

```
DROP TABLE <relation name>;
```

### ***DROP user statement***

- DROP USER statement is used to drop the user.

Syntax:

```
DROP USER <user name> [USER CASCADE];
```

### ***ALTER TABLE statement***

- Alter Table command is used to add or modify attributes to the existing relation.

Syntax:

```
ALTER TABLE <relation> ADD (attribute domain type);
```

## 4.0 Structure Query Language (SQL)

---

```
ALTER TABLE <relation> MODIFY (attribute domain type);
ALTER TABLE <relation> DROP CONSTRAINT <constraint_name>;
ALTER TABLE <relation> DROP COLOUMN <coloumn_name>;
```

Examples:

```
ALTER TABLE customer ADD PRIMARY KEY (customer_name);
ALTER TABLE customer ADD (customer_adds VARCHAR2(23));
ALTER TABLE customer MODIFY (customer_adds VARCHAR2(32) NOT NULL);
ALTER TABLE customer DROP PRIMARY KEY;
ALTER TABLE customer DROP CONSTRAINT fk_cname;
ALTER TABLE customer DROP COLOUMN customer_adds
```

### View

- View is a virtual table, it does not contain actual data it is map to the base table/s.
- When tables are created or populated with data, we may require to prevent all user from accessing all columns of table. So for the data security reasons view are created.
- One alternative solution is to create several table having appropriate no of columns and assigned each user to each table. This provides well data security but it keeps redundant data in tables. So it is not useful practically. So, views are generally created instead of it . It reduces redundant data.
- View can be created from a single table hiding some column/s or from the multiple tables mapping all or some of the columns of the base tables. View is the simple and effective way of hiding columns of tables for security reason.
- When view is referenced then only it holds data, so it reduces redundant data.
- When view is used to manipulate table , the underlying base e table/s are completely invisible. This adds level of data security.
- Since view can be created from multiple tables so it makes easy to query multiple tables because we can simply query views instead of query multiple tables.
- View may be read only or updateable view. Read only view only allow to read data from view. Updatable view allow insert, update and delete on view.
- If view is created from multiple tables it won't be updateable.
- If view is created without primary key and null columns then value/record can be inserted in view.
- The general syntax for view is

```
CREATE VIEW <view name> AS < query expression> ;
```

Example: creating view from single table.

```
CREATE VIEW vw_emp AS
    SELECT empno, ename, job FROM emp;
View column can be renamed as below:
CREATE VIEW vw_emp AS
    SELECT empno "Employee no", ename " Employee name"
        Job "work" FROM emp;
```

- Creating view from multiple tables.

```
CREATE VIEW vw_emp_info AS
    SELECT e.empno, e.ename, e.ejob, d.dname
        FROM emp e, dept d
    WHERE e.empno = d.dept no AND e.sal>1000;
```

### Common restrictions on view

- We cannot use delete statement on multiple table view.
- We cannot use insert statement unless all NOT NULL columns On underlying table are included.
- View must be created from single table to allow insert or update on view.

## **4.0 Structure Query Language (SQL)**

---

---

- If we use DISTINCT clause to create views, we cannot update or insert records within that view.

### ***Common application of views***

- Provides user security functions.
- Simplifies the constructions of complex queries.
- Summarize data from multiple tables.

### ***Common restrictions on updatable views***

- For the views to be updateable, the view definitions must not include.
- Aggregate function.
- DISTINCT, GROUP BY or HAVING clause.
- Sub – queries.
- Constraints, strin or value expression like bal\*1.05
- UNION, INTERSECT, OR MINUS/EXCEPT clause.
- View can be destroyed by using DROP VIEW command.

Syntax:

```
DROP VIEW <view name>;  
e.g. DROP view vw_emp;
```

### **Transactions:**

- A transaction consists of sequence of query / or updateable statements.
- SQL standard specifies, the transaction begins implicitly when SQL statement is executed and one of the following SQL statement must end with transaction commands.

COMMIT:

It commits (save) the current transaction changes on database / table/s by update statements. After the transaction is committed, a new transaction is automatically started.

ROLLBACK:

It rollback (undo) the current transaction. That is, it undo all the update performed by SQL statements. Thus database state is restored to what it was before the first statements of the transaction were executed.

- If program terminates without executing either of the commands commit or rollback. The updates or changes to database are either committed or rollback. This depends upon SQL implementation.
- In many SQL implementations, if transactions are continued and at the same moment if the system is restarted or fails then transaction is rollback.

## 5.0 Integrity Constraints

---

---

## 5.0 Integrity Constraints

- Integrity constraints are those constraints in database system which guard against invalid database operations or accidental damage to the database, by ensuring that authorized changes to the database. It does not allow to loss of data consistency in database, it ensures database consistency.
- In fact, integrity constraints provide a way of ensuring that changes made to the database by authorized users do not result in a loss of data consistency.
- Example of integrity constraints in E-R model
  - Key declaration: candidate key, primary key
  - Form of relationship : mapping cardinalities: one to one, one to many etc
- In database management system we can enforce any arbitrary predicate as integrity constraints but it adds overhead to the database system so its cost should be evaluated, as far as possible integrity constraint should with minimal overhead.

## 5.1 Domain Constraints

- Set of all possible values for attribute known as its domain. Domain constraints enforce attribute should hold only particular types of attributes. A domain of possible values should be associated with every attribute. Domain constraints are the most elementary form of integrity constraint. It is tested by database system whenever a new data item is entered into database. System test values inserted in the database and test queries to ensure that the comparisons make sense.

### Domain types in SQL

SQL standard supports a variety of built in domain types including:

- Char (n): A fixed length character string with user specified length n.
  - Varchar(n): A variable length string with user specified maximum length n.
  - Int: An integer (Machine dependant).
  - Smallint: A small integer.
  - Numeric (p,d): A fixed point number with user specified precision. Where (.) is counted in p.
  - Real, double precision: Floating point and double precision floating point numbers.
  - Float (n): A floating point number with precision of at least n digits.
  - Date: A calendar date containing a four digit year, month and day of the month.
  - Time: The time of a day, in hours, minutes and seconds.
  - Timestamp: A combination of date and time.
- New domains can be created from existing data types
  - E.g. `create domain Dollars numeric(12, 2)`  
`create domain Pounds numeric(12,2)`
  - The **check** clause in SQL allow domains to be restricted

Example 1

```
create domain salary-rate numeric(5)
constraint value-test check(value > = 5000)
```

The domain constraint ensures that the hourly-rate must greater than 5000

The clause **constraint value-test** is optional but useful to indicate which constraint an update violated.



## 5.0 Integrity Constraints

---

---

Example 2:

- ```
create domain AccountType char(10)
  constraint account-type-test
    check (value in ('Checking', 'Saving'))
```

Example 3:

```
create domain account-number char(10)
constraint account-number-null-test check(value not null)
```

## 5.2 Referential Integrity

- Referential integrity is a condition which Ensures that a value that appears in one relation for a given set of attributes also appears for a certain set of attributes in another relation.

### Example

If "B1" is a branch name appearing in one of the tuples in the *account* relation, then there exists a tuple in the *branch* relation where "B1" exist for branch name attribute.

.

### Example:

Consider two relation department and employee as follows

```
department(deptno#,dname)
employee(empno#,ename,deptno)
```

- Deletion of particular department from department table also need to delete records of employees they belongs to that particular department or delete need not be allow if there is any employee that is associated to that particular department that we are going to delete.
- Any update made in deptno in department table deptno in employee must be updated automatically.
- This implies primary key acts as a referential integrity constraint in a relation.

### Formal Definition

- Let  $r_1(R_1)$  and  $r_2(R_2)$  be relations with primary keys  $K_1$  and  $K_2$  respectively.  
The subset  $\alpha$  of  $R_2$  is a **foreign key** referencing  $K_1$  in relation  $r_1$ , if for every  $t_2$  in  $r_2$  there must be a tuple  $t_1$  in  $r_1$  such that  $t_1[K_1] = t_2[\alpha]$ .
- Referential integrity constraint also called subset dependency since its can be written as  
$$\Pi_\alpha (r_2) \subseteq \Pi K_1 (r_1)$$

### 5.2.1 Referential integrity in E-R Model

- Consider relationship set  $R$  between entity sets  $E_1$  and  $E_2$ . The relational schema for  $R$  includes the primary keys  $K_1$  of  $E_1$  and  $K_2$  of  $E_2$ . Then  $K_1$  and  $K_2$  form foreign keys on the relational schemas for  $E_1$  and  $E_2$  respectively that leads referential integrity constraint.
- Weak entity sets are also a source of referential integrity constraints. A weak entity set must include the primary key attributes of the entity set on which it depends



## 5.0 Integrity Constraints

---

---

### 5.2.2 Database modification

- The following tests must be made in order to preserve the following referential integrity constraint:  
 $\Pi_\alpha(r_2) \subseteq \Pi_K(r_1)$
- Insert. If a tuple  $t_2$  is inserted into  $r_2$ , the system must ensure that there is a tuple  $t_1$  in  $r_1$  such that  $t_1[K] = t_2[\alpha]$ . That is  
 $t_2[\alpha] \in \Pi_K(r_1)$
- Delete. If a tuple,  $t_1$  is deleted from  $r_1$ , the system must compute the set of tuples in  $r_2$  that reference  $t_1$ :  
 $\sigma_\alpha = t_1[K](r_2)$
- If this set is not empty
  - either the delete command is rejected as an error, or
  - the tuples that reference  $t_1$  must themselves be deleted (cascading deletions are possible).
- Update: There are two cases:
  - If a tuple  $t_2$  is updated in relation  $r_2$  and the update modifies values for foreign key  $\alpha$ , then a test similar to the insert case is made:
    - Let  $t_2'$  denote the new value of tuple  $t_2$ . The system must ensure that  
 $t_2'[\alpha] \in \Pi_K(r_1)$
  - If a tuple  $t_1$  is updated in  $r_1$ , and the update modifies values for the primary key ( $K$ ), then a test similar to the delete case is made:
    1. The system must compute  
 $\sigma_\alpha = t_1[K](r_2)$   
using the old value of  $t_1$  (the value before the update is applied).
    2. If this set is not empty
      1. the update may be rejected as an error, or
      2. the update may be cascaded to the tuples in the set, or
      3. the tuples in the set may be deleted.

### 5.2.3 Referential integrity in SQL

- Using the SQL Create table statement we can enforce
  - Primary key
  - Unique.
  - Foreign key

Example:

```
create table customer
(
    customer-name    char(20),
    customer-street   char(30),
    customer-city     char(30),
    primary key (customer-name)
)
```

```
create table branch
(
    branch-name      char(15),
    branch-city char(30),
    assets        integer,
    primary key (branch-name)
)
```



## 5.0 Integrity Constraints

---

---

```
create table account
(
    account-number  char(10),
    branch-name     char(15),
    balance         integer,
    primary key (account-number),
    foreign key (branch-name) references branch
)

create table depositor
(
    customer-name   char(20),
    account-number  char(10),
    primary key (customer-name, account-number),
    foreign key (account-number) references account,
    foreign key (customer-name) references customer
)
Cascading actions
```

Syntax

```
create table account
    .
    .
    .
    foreign key(branch-name) references branch
        on delete cascade
        on update cascade
    . . .
```

**on delete cascade**: if a delete of a tuple in *branch* results referential-integrity constraint violation, it also delete tuples in relation account that refers to the branch that was deleted.

**on update cascade** : if a update of a tuple in *branch* results referential-integrity constraint violation, it updates tuples in relation account that refers to the branch that was updated.

### 5.3 Assertion

- An assertion is a predicate expressing a condition we wish the database to always satisfy
- Domain constraints, functional dependency and referential integrity are special forms of assertion.
- If a constraint cannot be expressed in these forms, we use an assertion
  - e.g.
    - Sum of loan amounts for each branch is less than the sum of all account balances at the branch.
    - Every loan customer keeps a minimum of \$1000 in an account.
- General syntax for creating assertion in SQL is
  - **create assertion <assertion-name> check <predicate>**
- Example 1: sum of loan amounts for each branch is less than the sum of all account balances at the branch.



## 5.0 Integrity Constraints

---

---

```
create assertion sum-constraint check
  (not exists (select * from branch
    where (select sum(amount) from loan
      where loan.branch-name = branch.branch-name)
    >= (select sum(amount) from account
      where loan.branch-name = branch.branch-name)))
```

Example 2: every customer must have minimum balance 1000 in an account who are loan holder

```
create assertion balance-constraint check
  (not exists (
    select * from loan
    where not exists (
      select * from borrower, depositor, account
      where loan.loan-number = borrower.loan-number
        and borrower.customer-name = depositor.customer-name
        and depositor.account-number = account.account-number
        and account.balance >= 1000)))
```

- o When an assertion is created, the system tests it for validity. If the assertion is valid then only allow further modification. if test found assertion is violated then it can not go ahead.
- o Assertion testing may introduce a significant amount of overhead, especially if the assertions are complex; hence assertions should be used with great care.

## 5.4 Trigger

A trigger is a statement that is automatically executed by the system as a side effect of a modification to the database. While writing a trigger we must specify

- o conditions under which the trigger is executed
- o actions to be taken when trigger executes

Triggers re useful mechanism to perform certain task automatically when certain condition/s met. Sometime trigger is also called rule or action rule.

Basic syntax for trigger  
CREATE OR REPLACE TRIGGER <TRIGGER NAME>  
{BEFORE,AFTER}  
  {INSERT|DELETE|UPDATE [OF column, . .]} ON <table name>  
[REFERENCING {OLD AS <old>, NEW AS <new>}]  
[FOR EACH ROW [WHEN <condition>]]  
DECLARE  
  Variable declaration;  
BEGIN  
  ...  
END;



## 5.0 Integrity Constraints

---

---

Example 1: maintaining log

```
emp(empno,ename,sal)
emp_log(empno,ename,sal,operation_perform,userid,opr_date)
```

```
Create or replace emp_operation_log
After update or delete on emp
For each row
Declare
    oper varchar2(8);
    v_empno emp.empno%type;
    v_ename emp.ename%type;
    v_sal emp.sal%type;
begin
    if updating then
        oper:='Update';
    end if;
    if deleting then
        oper:='Delete';
    end if;

    v_empno:=old.empno;
    v_ename:=old.ename;
    v_sal:=old.sal;
    insert into emp_log values(v_empno,v_ename,v_sal,oper,user,sysdate);
end;
```

Example 2:

- o Suppose that instead of allowing negative account balances, the bank deals with overdrafts by
  - o setting the account balance to zero
  - o creating a loan in the amount of the overdraft providing same loan number as a account number of the overdrawn account

```
create trigger overdraft-trigger
after update on account
referencing new row as nrow
for each row
when nrow.balance < 0
begin
    insert into borrower
        (select customer-name, account-number
         from depositor
         where nrow.account_number = depositor.account-number);
    insert into loan values
        (nrow.account_number, nrow.branch-name, nrow.balance);
    update account
        set balance = 0
        where account.account_number = nrow.account_number
end;
```

## 5.0 Integrity Constraints

---

---

### 5.6 Functional Dependencies

- Functional dependencies are constraints on the set of legal relations. It defines attributes of relation, how they are related to each other.
- It determines unique value for a certain set of attributes to the value for another set of attributes that is functional dependency is a generalization of the notation of key.
- Functional dependencies are interrelationship among attributes of a relation.

#### Definition:

For a given relation R with attribute X and Y, Y is said to be functionally dependent on X, if given value for each X uniquely determines the value of the attribute in Y. X is called determinant of the functional dependency (FD) and functional dependency denoted by  $X \rightarrow Y$ .

Example 1: consider a relation supplier

Supplier(supplier\_id#, sname, status, city)

Here, sname, status and city are functionally dependent on supplier\_id. Meaning is that each supplier id uniquely determines the value of attributes supplier name, supplier status and city. This can be express by

Supplier.supplier\_id  $\rightarrow$  supplier.sname  
Supplier.supplier\_id  $\rightarrow$  supplier.status  
Supplier.supplier\_id  $\rightarrow$  supplier.city

Or simply,

supplier\_id  $\rightarrow$  sname  
supplier\_id  $\rightarrow$  status  
supplier\_id  $\rightarrow$  city

Question: is following functional dependency is valid ?

sname  $\rightarrow$  status  
sname  $\rightarrow$  city

Answer: it is true only if sname is unique, otherwise false.

Valid case

| sname | status |
|-------|--------|
| X     | Good   |
| Y     | Good   |

Invalid case

| sname | status |
|-------|--------|
| X     | Good   |
| Y     | Good   |
| X     | Bad    |

Example 2: Consider a relation student-info



## 5.0 Integrity Constraints

---

---

Student-info(name#, course#, phone\_no, major, prof, grade)

That is, {name, course} is composite primary key

This relation has the following functional dependencies  
 $\{\text{name} \rightarrow \text{phone\_no}, \text{name} \rightarrow \text{major}, \text{name, course} \rightarrow \text{grade}, \text{course} \rightarrow \text{prof}\}$

### Functional dependency $X \rightarrow Y$ satisfied on the relation R/ hold on R

FD  $X \rightarrow Y$  is satisfied on relation R if the cardinality of  $\Pi_Y(\sigma_{x=x}(r))$  is at most one. That is if, two tuples  $t_i$  and  $t_j$  of R have the same X value then the corresponding value of Y must identical.

Let R be a relational schema

$$\alpha \subseteq R \text{ and } \beta \subseteq R$$

then the functional dependency  $\alpha \rightarrow \beta$  holds on R iff for any legal relation r(R), whenever any two tuples  $t_1$  and  $t_2$  of r agree on the attributes  $\alpha$  then they also agree on the attributes  $\beta$ . That is, if  $t_1[\alpha] = t_2[\alpha]$  then  $t_1[\beta] = t_2[\beta]$ .

### 5.6.1 Application of Functional dependencies

Functional dependencies are applicable

- o To test the relation whether they are legal under a given set of functional dependency.
  - o Let r is a relation and F is a given set of functional dependencies. If r satisfies F, then we determine that r is legal under a given set of functional dependency F
- o To specify the constraints for the legal relation
  - o We say that f holds on R if all legal relations on R satisfy the set of functional dependencies F.

### 5.6.2 Types of Functional Dependencies

#### Trivial functional dependency

Functional dependencies are said to be trivial if it satisfied by all relations.

For example:

- o  $A \rightarrow A$  is trivial. It satisfied by all relation involving attribute A
- o  $AB \rightarrow A$  is trivial. It satisfied by all relations involving attribute A.

In general, A functional dependency of the form  $\alpha \rightarrow \beta$  is trivial if  $\beta \subseteq \alpha$ .

Verification:

Consider a relation r

| A              | B              | C              | D              |
|----------------|----------------|----------------|----------------|
| a <sub>1</sub> | b <sub>1</sub> | c <sub>1</sub> | d <sub>1</sub> |
| a <sub>1</sub> | b <sub>2</sub> | c <sub>1</sub> | d <sub>2</sub> |
| a <sub>2</sub> | b <sub>2</sub> | c <sub>2</sub> | d <sub>2</sub> |
| a <sub>2</sub> | b <sub>2</sub> | c <sub>2</sub> | d <sub>3</sub> |
| a <sub>3</sub> | b <sub>3</sub> | c <sub>2</sub> | d <sub>4</sub> |

$t_1 \rightarrow$   
 $t_2 \rightarrow$

## 5.0 Integrity Constraints

---

---

$t_1[AB]=a_2b_2, t_2[AB]=a_2b_2$  agree

$t_1[A]=a_2$   $t_2[A]=a_2$  agree

Here,  $t_1[AB]=t_2[AB] \Rightarrow t_1[A]=t_2[A]$ . This implies  $AB \rightarrow A$  is satisfied.

### Fully functionally dependency

For a given relation schema R, FD  $X \rightarrow Y$ , Y is said to be fully functionally dependent on X if there is no Z (where Z is a proper subset of X) such that  $Z \rightarrow Y$ .

Example: Let us consider relational schema  $R=(A,B,C,D,E,H)$  with the FDs  
 $F=\{A \rightarrow BC, CD \rightarrow E, C \rightarrow E, CD \rightarrow E, CD \rightarrow AH, ABH \rightarrow BD, DH \rightarrow BC\}$

- Here, the FD  $A \rightarrow BC$  is left reduced, so clearly, BC is fully functionally dependent on A (because there is no possible proper subset of only element A)
- Here, the FDs  $CD \rightarrow E$ ,  $C \rightarrow E$  where E is functionally dependent on CD and again E is functionally dependent on subset of CD. That is C (i.e.  $C \rightarrow E$ ). Hence E is not fully functionally dependent on CD.

Example: Consider a relation sales

Sales (product\_id#, sales\_date#, quantity, product\_name)

With the following functional dependencies

$F=\{\text{product\_id}, \text{sales\_date} \rightarrow \text{quantity}, \text{product\_id} \rightarrow \text{quantity}, \text{product\_id} \rightarrow \text{product\_name}\}$

- Here, FDs  $\text{product\_id}, \text{sales\_date} \rightarrow \text{quantity}$ ,  $\text{product\_id} \rightarrow \text{quantity}$ ,  $\text{quantity}$  is not fully functional dependent on  $\text{product\_id}, \text{sales\_date}$ .
- Here, functional dependency  $\text{product\_id} \rightarrow \text{product\_name}$ ,  $\text{product\_name}$  is fully functional dependent on  $\text{product\_id}$ .

### Partial functional dependency

For a given relation schema R with set of functional dependency F on attribute of R. Let K as a candidate key in R. if X is a proper subset of K and X and  $X \rightarrow A$  then A is said to be partially dependent on K.

Example: Consider a relation schema 'student\_course\_info'

student\_course\_info(name#, course#, grade, phone\_no, major, course\_department)

with the following FDs

```
{name→phone_no, major  
course→course_department,  
name, course→grade  
}
```

Here {name, course} is a candidate key. Here grade is fully functionally dependent on {name, course}. If there is a possible FD name→grade then we can not say grade is fully

## 5.0 Integrity Constraints

---

---

functionally dependent on {name, course}. Here phone\_no, major and course\_department are partially dependent on {name, course}

### Transitive dependency

For a given relational schema R with set of functional dependency F. Let X and Y be the subset of r and let A be the attribute of R s.t.  $X \not\subset Y$ ,  $A \not\subset XY$ . If the functional dependencies  $\{X \rightarrow Y, Y \rightarrow A\}$  implies by F (i.e.  $X \rightarrow Y \rightarrow A$ ) then A is said to be transitively dependent on X.

Example:

Let us consider relational schema 'prof\_info'

$\text{prof\_info} = (\text{prof\_name}\#, \text{department\_name}, \text{head\_of\_department})$

with the set functional dependency

$F = \{\text{prof\_name} \rightarrow \text{department\_name}, \text{department\_name} \rightarrow \text{head\_of\_department}\}$

Here  $\text{prof\_name} \rightarrow \text{department\_name} \rightarrow \text{head\_of\_department}$  so head\_of\_department is transitively dependent on the key prof\_name.

Example:

Let  $R = (A, B, C, D, E)$  and FDs  $F = \{AB \rightarrow C, B \rightarrow D, C \rightarrow E\}$

Here AB act a candidate key and E is transitively dependent on the key AB, since  $AB \rightarrow C \rightarrow E$ .

### 5.6.3 Closure of Set of Functional Dependencies

For a given set of functional dependencies F, there are certain other functional dependencies that are logically implies by F. (i.e. if  $A \rightarrow B$  and  $B \rightarrow C$ , then we can write  $A \rightarrow C$ ). the set of all functional dependencies logically implies F is the closure of F. Closure of F is denoted by  $F^+$ .

We can find all of  $F^+$  by applying Armstrong's Axioms:

- o if  $\beta \subseteq \alpha$  then  $\alpha \rightarrow \beta$  or  $\alpha \rightarrow \alpha$  (reflexive)
- o if  $\alpha \rightarrow \beta$  then  $\gamma \alpha \rightarrow \gamma \beta$  (augmentation)
- o if  $\alpha \rightarrow \beta$  and  $\beta \rightarrow \gamma$  then  $\alpha \rightarrow \gamma$  (transitivity)

Example: Let  $R = (A, B, C, G, H, I)$

$F = \{A \rightarrow B, A \rightarrow C, CG \rightarrow H, CG \rightarrow I, B \rightarrow H\}$

Compute closure of  $F^+$ .

Closure of  $F^+$  computed as follow:

- o  $A \rightarrow H$ 
  - o by transitivity  $A \rightarrow B$  and  $B \rightarrow H$
- o  $AG \rightarrow I$ 
  - o By augmenting  $A \rightarrow C$  with G we get  $AG \rightarrow CG$  and then by transitivity with  $CG \rightarrow I$  we get  $AG \rightarrow I$
- o  $CG \rightarrow HI$
- o From  $CG \rightarrow H$  and  $CG \rightarrow I$  "union rule" can be inferred from definition of functional dependency ot
  - Augmentation of  $CG \rightarrow I$  to infer  $CG \rightarrow CGI$ , argumentation of  $CG \rightarrow H$  to infer  $CGI \rightarrow HI$ , and then transitivity.

Hence,  $F^+ = \{A \rightarrow A, B \rightarrow B, C \rightarrow C, H \rightarrow H, G \rightarrow G, I \rightarrow I, A \rightarrow B,$



## 5.0 Integrity Constraints

---

---

A→C, CG→H, CG→I, CG→HI, B→H, A→H,  
AG→I, CG→Hi  
}

here , first six FDs obtain by reflexive axiom.

We can further simplify the the computation of F+ by using the following addition rule.

- (a) if  $\alpha \rightarrow \beta$  holds and  $\alpha \rightarrow \gamma$  holds, then  $\alpha \rightarrow \beta \gamma$  (Additivity or union rule)
- (b) if  $\alpha \rightarrow \beta \gamma$  holds then  $\alpha \rightarrow \beta$  holds and  $\alpha \rightarrow \gamma$  holds (projectivity/decomposition)
- (c) if  $\alpha \rightarrow \beta$  holds and  $\gamma \beta \rightarrow \delta$  holds then  $\alpha \gamma \rightarrow \delta$  holds (pseudotransitivity)

Examples: Let R=(A,B,C,D) and F={A→B,A→C,BC→D} then compute F<sup>+</sup>.

- Since A→B and A→C then by union rule A→BC.
- Since BC→D, then by projective/decomposition B→D, C→D. Again by transitivity A→B & B→D  $\Rightarrow$  A→D and A→C and C→D  $\Rightarrow$  A→D.
- Hence, F<sup>+</sup> = {A→A, B→B, C→C, D→D, A→B, A→C, BC→D, B→D, C→D, A→D}

### 5.6.4 Attribute Closure

The closure of X under a set of functional dependencies F, written as X+, is the set of attributes {A<sub>1</sub>,A<sub>2</sub>, . . . A<sub>m</sub>} such that the FD X→A<sub>i</sub> for A<sub>i</sub>∈X<sup>+</sup> follows from F by the inference axioms for functional dependencies.

Example:

Let X=BCD and F={A→BC,CD→E,E→C,D→AEH,ABH→BD,DH→BC}. Compute the closure X<sup>+</sup> of X under F.

- initialize X<sup>+</sup>:=BCD.
- Since left hand side of the FD CD→E is a subset of X<sup>+</sup> (i.e CD $\subseteq$ X<sup>+</sup>), X<sup>+</sup> is augmented by the right hand side of the FD (i.e. E) thus now X<sup>+</sup>:=BCDE.
- Similarly, D $\subseteq$ X<sup>+</sup>, the right hand side of the FD D→AEH is added to X<sup>+</sup>. Hence now X<sup>+</sup>:=ABCDEH.
- Now X<sup>+</sup> can not be augmented any further because no FDs left hand side is subset of X<sup>+</sup>.

### Application of Attribute Closure

1. Testing superkey  
To test  $\alpha$  is a superkey we compute  $\alpha^+$  and check whether  $\alpha^+$  contains all attributes of R. if so  $\alpha$  is a superkey, otherwise not.
2. Testing functional dependencies  
To check a functional dependency  $\alpha \rightarrow \beta$  holds check whether  $\beta \subseteq \alpha^+$ . If so  $\alpha \rightarrow \beta$ ; otherwise not.



## **6.0 Relational Database Design**

---

### **6.0 Relational Database Design**

#### **6.1 Pitfall of relational model**

- The main goal of relational database is to create / find good collection of relational schemas. Such that database should allow us to store data / information without unnecessary redundancy and should also allow to retrieve information easily. That is the goal of relational database design should concentrated
  - To avoid redundant data from database.
  - To ensure that relationships among attributes are represented.
  - To facilitate the checking of updates for the violation of database integrity constraints.
- A bad relational database design may lead to:
  - Repetition of information.
    - That is, it leads data redundancy in database, so obviously it requires much space.
  - Inability to represent certain information.

Example 1: Consider the relational schema

Branch\_loan = ( branch\_name, branch\_city, assets, customer\_name, loan\_no, amount)

| branch_name | branch city    | assets | customer name | loan no | amount |
|-------------|----------------|--------|---------------|---------|--------|
| kathmandu   | baneshwor      | 25000  | rohan         | L - 15  | 3000   |
| Lalitpur    | patan          | 19000  | mohan         | L - 17  | 5000   |
| Kathmandu   | baneshwor      | 25000  | raju          | L - 19  | 10000  |
| Pokhara     | Prithibi nagar | 17000  | manoj         | L - 10  | 7000   |
| Lalitpur    | patan          | 19000  | swikar        | L - 30  | 9000   |

#### **Redundancy:**

- Data for branch\_name, branch\_city, assets are repeated for each loan that provides by bank.
- Storing information several times leads waste of storage space / time.
- Data redundancy leads problem in Insertion, deletion and update.

#### **Insertion problem**

- We cannot store information about a branch without loan information, we can use null values for loan information, but they are difficult to handle.

#### **Deletion problem**

- In this example, if we delete the information of Manoj (i.e. DELETE FROM branch\_loan WHERE customer\_name = 'Manoj';), we cannot obtain the information of pokhara branch (i.e. we don't know branch city of pokhara, total asset of pokhara branch etc.

#### **Update problem**

- Since data are duplicated so multiple copies of same fact need to update while updating one. It increases the possibility of data inconsistency. When we made update in one copy there is possibility that only some of the multiple copies are updated but not all, which lead data/database in inconsistent state..



## **6.0 Relational Database Design**

---

### **6.2 Decomposition**

The idea of decomposition is break down large and complicated relation in no. of simple and small relations which minimized data redundancy. It can be consider principle to solve the relational model problem.

#### **Definition**

The decomposition of relation schema  $R = (A_1, A_2, \dots, A_n)$  is a set of relation schema  $\{R_1, R_2, \dots, R_m\}$ , such that  $R_i \subseteq R \quad \forall 1 \leq i \leq m$  and  $R_1 \cup R_2 \cup \dots \cup R_m = R$ .

That is all attributes of an original schema ( $R$ ) must appear in the decomposition ( $R_1, R_2$ ).  
That is,  $R = R_1 \cup R_2$ . if  $R \neq R_1 \cup R_2$  then such decomposition called lossey join decomposition.  
That is,  $R \neq \Pi_{R_1}(R) \bowtie \Pi_{R_2}(R)$ . Decomposition should **lossless join decomposition**.

A decomposition of relation schema  $R$  into  $R_1$  and  $R_2$  is lossless join iff at least one of the following dependencies is in  $F^+$ .

$$\begin{aligned} R_1 \cap R_2 &\rightarrow R_1 \\ R_1 \cap R_2 &\rightarrow R_2 \end{aligned}$$

Example 1: The problems in the relational schema branch\_loan (illustrated in above example) can be resolved if we replace it with the following relation schemas.

Branch (# branch\_name, branch\_city, assets)  
Loan (customer\_name, loan\_number, branch\_name, amount)

Example 2: Consider the relation schema to store the information a student maintain by the university.

Student\_info (#name, course, phone\_no, major, prof, grade)

| name   | course | phone_no | major            | prof   | grade |
|--------|--------|----------|------------------|--------|-------|
| John   | 353    | 374537   | Computer Science | Smith  | A     |
| Scott  | 329    | 427993   | Mathematics      | James  | S     |
| John   | 328    | 374537   | Computer Science | Adams  | A     |
| Allen  | 432    | 729312   | Physics          | Blake  | C     |
| Turner | 523    | 252731   | Chemistry        | Miller | B     |
| John   | 320    | 374537   | Computer Science | Martin | A     |
| Scott  | 328    | 727993   | Mathematics      | Ford   | B     |

#### **Problems:**

##### **Redundancy:**

- Data for major and phone\_no of student are stored several times in the database once for each course that is taken by a student.

##### **Complicates updating:**

- Multiple copies of some facts may lead to update which leads possibility of inconsistency.
- Here, change of phone\_no of John is required we need to update three records (tuples) corresponding to the student John. If one of the three tuples is not changed there will be inconsistency in the date.



## **6.0 Relational Database Design**

---

### **Complicate insertion**

- If this is only the relation in the database showing the association between a faculty member and the course he / she teaches, then the information that a given professor is teaching a given course cannot be entered in the database unless a student is registered in the course.

### **Deletion Problem:**

- If the only one student is registered in a given course then the information as to which professor is offering the course will be lost if this is only the relation in the database showing the association between the faculty member and the course he / she teaches. If database have another relation that establishes the relationship between a course, the deletion of 4<sup>th</sup> & 5<sup>th</sup> tuple in this relation will not cause the information about the councils teach to be lost.

### **Solution?**

#### **⇒ Decomposition**

The problems in this relation schema `student_info` can be resolved if we it with the following relation schemas.

Students (#name, phone\_no, major)  
Transcript (#name, course, grade)  
Teacher (course, prof)

- Here, first relation schema gives the phone number and major subject of each student such information will be stored only once for each student. Thus, any changes in the phone number will thus require changes in only one tuple of this relation
- The second relation schema 'TRANSCRIP' stores the grade of each student in each course. So, to insert information about student phone number, major and the course teaches by which the professor.
- Third relation schema records the teacher of each course.

### **What is the problem of such [REDACTED] decomposition?**

- One of the disadvantages of original relation [REDACTED] schema 'student\_info' with these true relational schemas is that retrieval of certain information required to performed natural join operation.
- We know that to resolve the problem of bad relational database design we need to decompose relation but the problem is that how to decompose relation. That is, we require to process decomposition of relation that may give good relational database. Normalization gives the approach for designing the best relational database under the five normal forms.

## **6.0 Relational Database Design**

---

### **6.3 Normalization:**

Normalization is the process of reducing redundant data / information in the database by decomposing the long relation. It is an approach for designing reliable database system. Normalization theory is built under the concept of the normal form. There are several normal form called

- First Normal Form. (1NF)
- Second Normal Form (2NF)
- Third Normal Form (3NF)
- Fourth Normal Form (4NF)
- Fifth Normal Form (5NF)

A relation is said to be in particular normal form if it satisfies the set of this particular normal form if it satisfies the set of this particular normal form's constraints. In practical, third normal form is sufficient to design reliable database system.

Normalization theory is based on:

- Functional dependencies
- Multi valued dependencies.

#### **6.3.1 Needs of normalization**

In database, there is considerable amount of data redundancy and also serious insertion, deletion and update. Thus to reduce the data redundancy as far as possible and for easy insert, delete and update operation in relational database system we require normalization. It does not allow any inconsistency in database system.

#### **6.3.2 Objectives of Normalization**

Dr. Codd suggested five normal forms for relational database design without data redundancy and serious insertion, update and deletion. According to him, the objectives of normalization are:

1. To free a collection of relations from undesirable insertion, update and deletion.
2. To move the relational model more informative to use.
3. To increase the lifetime of application programs.

#### **6.3.3 Properties of relation after normalization**

1. No data value should be duplicate in different row or tuple.
2. A value must be specified for every attribute in a tuple.
3. Important information should not be accidentally lost.
4. When new data / record are inserted to relation, other relations in database should not be affected.



## **6.0 Relational Database Design**

---

### **6.3.4 Non -Normalized Relation**

A relation schema R is said to be non-normalized relation if any domain of attributes of R are non atomic.

Example:

| faculty  | prof | course                |             |
|----------|------|-----------------------|-------------|
|          |      | course_name           | department  |
| computer | X    | DBMS                  | Computer    |
|          |      | Software Engineering  | Computer    |
|          |      | Microprocessor        | Electronics |
|          | Y    | Mathematical Analysis | Math        |
| Physics  | Z    | Probability Theory    | Stat        |
|          | Z    | Modern Physic         | Physics     |
|          | P    | Dynamics              | Physics     |
|          |      | Vector Analysis       | Math        |

Figure: non-normalized relation

- Here, the attribute of relation schema, that is course is divided into course\_name and department. That is, domain of attribute course is non atomic.
- Here, each row contains multiple set of values.

### **6.3.5 Key and non key attribute (prime and non prime attribute)**

An attribute A in relation R is said to be key attribute if A is any part of candidate key; otherwise A is called non key attribute.

Example: Consider a relational schema 'student\_course\_info'

Student\_course\_info=(#name, course, grade, phone, major, department)

With the following FDs

$F=\{name \rightarrow phone\_no, major, course\}$   $Math\{department, name, course\}$   $Math\{grade\}$

- Here, name and course are key attributes and phone\_no, major and department are non key attribute.

Example: Let  $R=(A,B,C,D,E)$  and FDs  $F=\{AB \rightarrow C, B \rightarrow D, C \rightarrow E\}$

- Here AB is composite primary key (composite candidate key) so attribute A and B are key attributes and attributes C, D and E are non key attributes.

## **6.4 Normal Forms**

### **6.4.1 First Normal Form**

- A relation schema R is said to be in first normal form if domain of all attributes of R are atomic. That is, for any relational schema R to be in first normal form, each attribute of relation not divisible and each row must contain single set of values for all attributes.
- The previous relational schema 'course\_info' can be covert into first normal form as below



## 6.0 Relational Database Design

---

| prof | course_name           | faculty  | department  |
|------|-----------------------|----------|-------------|
| X    | DBMS                  | computer | Computer    |
| X    | Software Engineering  | computer | Computer    |
| X    | Microprocessor        | computer | Electronics |
| Y    | Mathematical Analysis | computer | Math        |
| Y    | Probability Theory    | computer | Stat        |
| Z    | Modern Physic         | Physics  | Physics     |
| Z    | Dynamics              | Physics  | Physics     |
| P    | Vector Analysis       | Physics  | Math        |

Figure: course\_info relation in first normal form

- First normal form enforce relation in tabular form.  
Here, FDs  $F=\{prof \rightarrow faculty, course\_name \rightarrow department\}$
- Here key attributes are prof and course and key of relation is  $\{prof, course\_name\}$
- The representation of relation 'course\_info' in first normal form has several drawbacks.  
Major problem is data redundancy.

### Data redundancy

- Given professor assigned to corresponding faculty is repeated a number of times.
- Given course offered by corresponding department is repeated number of times.
- Data redundancy leads problem in insertion, deletion and update.

### Update problem

- If professor change faculty to teach then we must change all rows where that professor appears, it may leads inconsistency in database.

### Deletion problem

- If we have to delete the professor who teach only one subject (course\_name) in such case deletion cause the loss of information about the department to which the course belongs.

### Insertion problem

- Suppose department introduce new course (i.e. course\_name) such information can not insert without professor name and assigned faculty. This is possible inserting only null values instead of professor name and his/her assigned faculty.

### 6.4.2 Second Normal Form

- A relation schema R is said to be in second normal form (2NF) if it is first normal form and all non key attributes are fully functionally dependent on relation key (s).
- A second normal from does not allow partial dependency between non key attribute and relation key (s). But it does nor enforce that non key attribute may not functionally dependent on another non key attribute. (In fact, 2NF that does not allow such dependency called relation in 3NF).

Example: Let us consider the relation 'student\_info'

Student-info=(#name,course,phone\_no,major,prof,grade)  
with the functional dependencies

$$F=\{name \rightarrow phone\_no, name \rightarrow major, course \rightarrow prof, name, course \rightarrow grade\}$$

- Here, in this relation grade is fully functionally dependent on key (name, course). But phone\_no and major are partially dependent on name and prof and prof is partially dependent on course.
- Since, second normal form does not allow partial dependency so the given relation 'student\_info' is only in 1NF but not in 2NF.



## **6.0 Relational Database Design**

---

**Remarks:** The relation can be converted into 2NF as below:

Student-general-info=(name,phone\_mno,major)

Transcript=(#name,course,grade)

Teacher=(#course,prof)

- Here the functional dependencies on relations are as below:

*Student-general-info*

$$F=\{name \rightarrow phone\_no, name \rightarrow major\}$$

Phone\_no and major are fully functional dependent on key 'name' of relation schema 'student-general-info'. So the is in 2NF. Moreover, here is no functional dependency between non key attributes (phone\_no & major). So this relation is also in 3NF (discuss later)

*Transcript*

$$F=\{name, course \rightarrow grade\}$$

Here grade is fully functionally dependent on relation key (name, course). So this relation is in 2NF. Moreover, here is no functional dependency between non key attributes because here is only one non key attribute grade.

*Teacher*

$$F=\{course \rightarrow prof\}$$

Here. Prof is fully functionally dependent on relation key course. So this relation is in 2NF. Moreover, here is no functional dependency between non key attributes. Here is only one non key attributes prof.

### **6.4.3 Third Normal Form**

- A relation R is said to be in third normal form if it is in 2NF, every non key attributes in non transitively and fully functionally dependent on the every candidate key.
- That is, relational schema in 3NF does not allow partial dependency, transitive dependency. For any relation that is in 3NF, it must in 2NF, every non key attribute must fully functionally dependent on relation key (s) and relation must not exist partial dependency, transitive dependency and no functional dependency between non key attribute.
- The problems with a relation schema that is not in 3NF (problems with a relational schema in 2NF) are
  - If a relation schema R contains a transitive dependency  $Z \rightarrow X \rightarrow A$ , we can not insert an values for x in relation along with X value. That is, we can not independently record the fact that for each value of X there is one value of A (insertion problem). Similarly, deletion of a  $Z \rightarrow A$  association also required the deletion of  $X \rightarrow A$  association (deletion problem).
  - If a relation schema R contains a partial dependency (That is, an attribute A dependent on subset X of the key K of R.. i.e K is key of R,  $X \subset K$  and  $X \rightarrow A$ ) then association between X and A (i.e.  $X \rightarrow A$ ) can not express (insert) unless remaining part of K is present in a tuple, remaining part of k must also be express (insert) since K is a key of relation, these part can not be null.
  - In the above relation 'course\_detail'  
 $Course\_detail=(#course,prof,#room,enroll\_limit)$



## **6.0 Relational Database Design**

---

There is a transitive dependency,  $\text{course} \rightarrow \text{room} \rightarrow \text{enroll\_limit}$ . We can eliminate this transitive dependency by decomposing the relation 'course\_detail' into the relations  
Course\_prof\_info=(#course,prof,enroll\_limit)  
Fds={course→prof, course→enroll\_limit}

Course\_room\_info=(course,room)  
Fds=no functional dependency exist because course and room both are here foreign key.

That is, all relations in 3NF are as below:

*Given relation in 1NF*

Course\_detail=(#course,prof,room,room\_capacity,enroll\_limit)  
Fds={course→(prof,room,room\_capacity,enroll\_limit), room→room\_capacity}

*Relations in 2NF*

Course\_detail=(#course,prof,room,enroll\_limit)  
FDs={course→prof, course→room, course→enroll\_limit, room→enroll\_limit}

Room\_detail=(#room,room\_capacity)  
Fds={room→room\_capacity}

*Relations in 3NF*

Course\_prof\_info=(#course,prof,enroll\_limit)  
FDs={course→prof, course→enroll\_limit}

Course\_room\_info=(course,room)  
No FDs

Room\_detail=(#room,room\_capacity)  
Fds={room→room\_capacity}

## **6.5 Boyce Codd Normal Form (BCNF)**

A relation schema R is said to be in Boyce Codd normal form if it is in 3NF and every FDs in  $F^+$  should be in the form  $X \rightarrow A$  where  $X \subseteq S$  and  $A \in S$  and at least one of the following condition hold:

- (a)  $X \rightarrow A$  is a trivial FD (i.e.  $A \in X$ ) or
- (b)  $X \rightarrow R$  (i.e. X is a superkey of R, here X called determinant of functional dependency  $X \rightarrow R$ )

- The BCNF imposes stronger constraints on the type of FDs allow in a relation.
  - It allows only those non trivial functional dependencies whose determinants are candidate sperkeys of relation. But in case of 3NF, it allow non trivial FDs whose determinant is not a candidate superkey if right-hand side of FDs contained a candidate key.
  - That is BCNF enforce more stronger constraints than 3NF.

Example: Let us consider the relation schema "grade" which is in 3NF.

Grade=(#name,student\_id,course,grade)



## **6.0 Relational Database Design**

---

- Assume that , each student has unique name and unique student\_id then set of FDs={name,course}  $\rightarrow_{\text{grade}}$ ,  
 $\text{student\_id},\text{course} \rightarrow \text{grade}, \text{name} \rightarrow \text{student\_id}, \text{student\_id} \rightarrow \text{name}$
- Here this relation has two candidate keys {name,course} and {student\_id,course}. Each of those composite key has common attribute course.
- Here the relation grade is not in BCNF because the dependencies {student\_id→name} and {name→student\_id} are non trivial and there determinants are not superkey of relation grade.

*Drawback of this relation (which is not 3NF)*

- Data redundancy: The association between name and corresponding student\_id are repeated.
- Update problem: Any changes in one of these attribute value (name or student\_id) has to reflected all tuples; otherwise there will be inconsistency in the database.
- Insertion problem: The student\_id can not be associated with the student name unless the student has registered in a course.
- Deletion problem: The association between student\_id and student name is lost if the student deletes all courses he/she is registered in.
- Solution ?  
 $\text{student\_info}=(\# \text{student\_id}, \text{name})$   
 $\text{grade}(\# \text{student\_id}, \text{course}, \text{grade})$

These problems of relation schema in 3NF occur since relation may have overlapping candidate keys. BCNF removes this problem. So it is stronger than 3NF.

Example: Let us consider the relational schema

$\text{Student\_info}=(\text{student\_id}, \text{name}, \text{phone\_no}, \text{major})$

where student\_id, name and phone\_no assumed to be unique in this relation. The following FDs satisfy this relation.

FDs={student\_id→name,student\_id→phone\_no,student\_id→major, name→student\_id, name→phone\_no, name→major, phone\_no→student\_id, phone\_no→name, phone\_no→major}

Here each non trivial FD involves candidate key as determinant. Hence the relation 'student\_info' is in BCNF.



## Concurrency Control

This chapter deals with the study of concurrency control techniques which are used to ensure the isolation property of concurrently executing transactions. All the schemes presented here assume that the schedules are serializable.

### **Lock based protocol:-**

One way through which we can assume serializability is to access the variables in mutually exclusive manner. i.e. if one transaction is accessing a data item, no other transaction can modify that data item. To implement the notion mentioned above, we implement a technique of using lock on data item .i.e. transaction can access a data item only if it holds lock on that data item.

### **Types of lock:-**

#### **1. Binary lock:-**

A binary lock can have two states i.e. locked and unlocked (1 for locked state and 0 for unlocked state). A transaction requests access to an item Q by first issuing LOCK (Q) operations. If  $\text{LOCK}(Q) = 1$ , the transaction is forced to wait and if  $\text{LOCK}(Q) = 0$ , it is set to 1 and the transaction is allowed to access item Q. When transaction finishes with the operation, it finally issues unlock operation by setting  $\text{LOCK}(Q)$  to 0. The rule followed by binary locking scheme is as follows:

- A transaction T must issue the operation lock (Q) before any read (Q) or write (Q) operations.
- A transaction T must issue unlock (Q) after all read (Q) and write (Q) operations are completed.
- A transaction T will not issue any lock (Q) if it already holds lock on item Q.
- A transaction T will issue unlock (Q) iff it already holds a lock on data item Q.

#### **2. Shared/Exclusive or Read/Write lock:-**

Binary lock as discussed earlier is too restrictive for database items. But we should allow multiple transactions to access the same item if they all access Q for reading purpose only. Hence shared/exclusive lock is desired.

Ø Shared: - if  $T_i$  holds shared mode lock on Q, it can only read and cannot write. Multiple shared mode lock can exist for the same data item Q.

Ø Exclusive: - if  $T_i$  holds exclusive mode lock on Q, then it can both read as well as write. There can exist only one exclusive mode lock on a data item Q.

Let A and B are the two arbitrary lock modes. If transaction  $T_i$  can be granted a lock on Q immediately inspite of the presence of the mode B lock on Q, then we say mode A is compatible with mode B. Hence lock compatibility matrix can be summarized as shown below:

|   |       |       |
|---|-------|-------|
|   | S     | X     |
| S | true  | false |
| X | false | false |

Note that the shared mode lock is only compatible with the shared mode lock. Hence at any time more than one shared mode lock can exist on the same data item.

To access a data item, transaction  $T_i$  must first lock that item. If data item is already locked in an incompatible mode, then the concurrency control manager do not grant the lock until all incompatible locks held by other transactions have been released. Thus  $T_i$  is made to wait until all incompatible locks held by other transactions have been released. Any transaction  $T_i$  may unlock the data item that it had locked in earlier point but unlocking data item immediately may not ensure serializability. For example consider the schedule as shown below(A=1000 & B=2000):

| <b>T1</b> | <b>T2</b>    | <b>Concurrency control manager</b> | <b>Value</b>       |
|-----------|--------------|------------------------------------|--------------------|
| Lock-X(B) |              |                                    |                    |
|           |              | Grant-X(B,T1)                      |                    |
| Read(B)   |              |                                    | 2000               |
| B:=B-50   |              |                                    |                    |
| Write(B)  |              |                                    | 1950               |
| Unlock(B) |              |                                    |                    |
|           | Lock-S(A)    |                                    |                    |
|           |              | Grant-S(A,T2)                      |                    |
|           | Read(A)      |                                    | 1000               |
|           | Unlock(A)    |                                    |                    |
|           | Lock-S(B)    |                                    |                    |
|           |              | Grant-S(B,T2)                      |                    |
|           | Read(B)      |                                    | 1950               |
|           | Unlock(B)    |                                    |                    |
|           | Display(A+B) |                                    | <b><u>2950</u></b> |
| Lock-X(A) |              |                                    |                    |
|           |              | Grant-X(A,T1)                      |                    |
| Read(A)   |              |                                    | 1000               |
| A:=A+50   |              |                                    |                    |
| Write(A)  |              |                                    | 1050               |
| Unlock(A) |              |                                    |                    |

As we can see that unlocking B too early has caused the database in inconsistent state. i.e. transaction T2 should display 3000 as a result but it is now showing 2950. Hence from here we can conclude that unlocking data items too early is not desirable. Hence delaying data unlocking can solve this problem. i.e.

|                                                                                                                                   |                                                                                                     |
|-----------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------|
| $T_{20}:$<br>Lock-X(B)<br>Read(B)<br>B:=B-50<br>Write(B)<br>Lock-X(A)<br>Read(A)<br>A:=A+50<br>Write(A)<br>Unlock(B)<br>Unlock(A) | $T_{30}:$<br>Lock-S(A)<br>Read(A)<br>Lock-S(B)<br>Read(B)<br>Display(A+B)<br>Unlock(A)<br>Unlock(B) |
|-----------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------|

Hence, from the schedule shown above, it becomes sure that no undesirable result is displayed.

### Two phase locking protocol:-

This is a protocol which ensures conflict-serializable schedules. This protocol requires that each transaction issue lock and unlock requests in different phases. There are two phases as discussed below:

#### Phase 1: Growing Phase

- transaction may obtain locks
- transaction may not release locks

#### Phase 2: Shrinking Phase

- transaction may release locks
- transaction may not obtain locks

For example:

Transaction  $T_{20}$  and  $T_{30}$  are the example of two phase locking protocol. Initially a transaction is in growing phase and can acquire locks as per its need. Then the transaction reaches its lock point i.e. it is the point where a transaction acquires its final lock. Finally when transaction unlocks any data item, it enters into shrinking phase and no more locks can be acquired again.

The protocol assures serializability. It can be proved that the transactions can be serialized in the order of their **lock points** (i.e. the point where a transaction acquired its final lock). Two-phase locking *does not* ensure freedom from deadlocks.

Cascading roll-back is possible under two-phase locking. To avoid this, follow a modified protocol called **strict two-phase locking**. Here a transaction must hold all its exclusive locks till it commits/aborts.

**Rigorous two-phase locking** is even stricter: here *all* locks are held till commit/abort. In this protocol transactions can be serialized in the order in which they commit.

Lock conversion:

Two-phase locking with lock conversions:

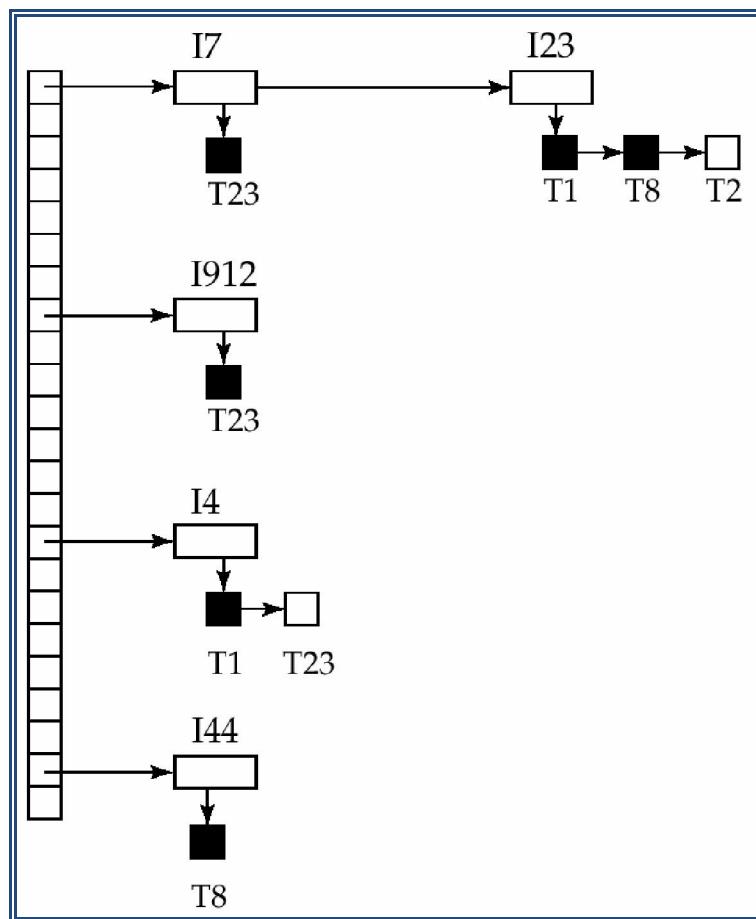
- Growing Phase:

- can acquire a lock-S on item
- can acquire a lock-X on item
- can convert a lock-S to a lock-X (upgrade)

- Shrinking Phase:
  - can release a lock-S
  - can release a lock-X
  - can convert a lock-X to a lock-S (downgrade)

### Implementation of locks:-

A **lock manager** can be implemented as a separate process to which transactions send lock and unlock requests. The lock manager replies to a lock request by sending a lock grant messages (or a message asking the transaction to roll back, in case of a deadlock). The requesting transaction waits until its request is answered. The lock manager maintains a data-structure called a **lock table** to record granted locks and pending requests. The lock table is usually implemented as an in-memory hash table indexed on the name of the data item being locked.



- Black rectangles indicate granted locks, white ones indicate waiting requests
- Lock table also records the type of lock granted or requested
- New request is added to the end of the queue of requests for the data item, and granted if it is compatible with all earlier locks
- Unlock requests result in the request being deleted, and later requests are checked to see if they can now be granted
- If transaction aborts, all waiting or granted requests of the transaction are deleted
  - lock manager may keep a list of locks held by each transaction, to implement this efficiently

## Timestamp ordering protocol:-

Each transaction is issued a timestamp when it enters the system. If an old transaction  $T_i$  has time-stamp  $TS(T_i)$ , a new transaction  $T_j$  is assigned time-stamp  $TS(T_j)$  such that  $TS(T_i) < TS(T_j)$ . The protocol manages concurrent execution such that the time-stamps determine the serializability order. In order to assure such behavior, the protocol maintains *for each data Q two timestamp values:*

- W-timestamp ( $Q$ ) is the **largest time-stamp** of any transaction that executed write ( $Q$ ) successfully.
- R-timestamp ( $Q$ ) is the **largest time-stamp** of any transaction that executed read ( $Q$ ) successfully.

### Rules:

- $\emptyset$   $T_i$  issues read( $Q$ ):
  - i.  $TS(T_i) < W\text{-timestamp}(Q)$   
Reject  $T_i$ , rollback
  - ii.  $TS(T_i) \geq W\text{-timestamp}(Q)$   
Commit read instruction, update R-timestamp ( $Q$ ) = maximum  $TS(T_i)$
- $\emptyset$   $T_i$  issues write( $Q$ ):
  - i.  $TS(T_i) < R\text{-timestamp}(Q)$   
Reject write ( $Q$ ), roll back  $T_i$
  - ii.  $TS(T_i) < W\text{-timestamp}(Q)$   
Reject write ( $Q$ ), roll back  $T_i$
  - iii. Otherwise execute write( $Q$ ) instruction  
Update  $W\text{-timestamp}(Q) = TS(T_i)$

Example:

| $T_{14}$      | $T_{15}$     | R-TS  | W-TS  |
|---------------|--------------|-------|-------|
| Read(B)       |              | B-14  | ----- |
|               | Read(B)      | B-15  | ----- |
|               | B:=B-50      | ----- | ----- |
|               | Write(B)     | ----- | B-15  |
| Read(A)       |              | A-14  | ---   |
|               | Read(A)      | A-15  | ----  |
| Display (A+B) |              |       |       |
|               | A:=A+50      |       |       |
|               | Write(A)     | ----- | A-15  |
|               | Display(A+B) |       |       |

Problem with timestamp-ordering protocol:

Suppose  $T_i$  aborts, but  $T_j$  has read a data item written by  $T_i$ . Then  $T_j$  must abort; if  $T_j$  had been allowed to commit earlier, the schedule is not recoverable. Further, any transaction that has read a data item written by  $T_j$  must abort. This can lead to cascading rollback that is, a chain of rollbacks.

### Thomas' Write Rule:

Modified version of the timestamp-ordering protocol in which obsolete **write** operations may be ignored under certain circumstances is called Thomas' write rule. Consider schedule that is shown below:

| <b>T<sub>16</sub></b> | <b>T<sub>17</sub></b> |
|-----------------------|-----------------------|
| Read(Q)               |                       |
|                       | Write(Q)              |
| Write(Q)              |                       |

If we apply timestamp ordering protocol in the above schedule: the read (Q) instruction of T<sub>16</sub> is executed and similarly the Write (Q) instruction of transaction T<sub>17</sub> also gets committed and W-timestamp (Q) = TS (T<sub>17</sub>). Now when write (Q) instruction of T<sub>16</sub> is executed we find that

TS (T<sub>16</sub>) < W-timestamp (Q); hence we have to roll back transaction T<sub>16</sub>.

In this case when T<sub>16</sub> attempts to write data item Q, it is attempting to write an obsolete value of {Q} because: any transaction Ti with TS (Ti) < TS (T<sub>17</sub>) that attempts read (Q) will be rolled back, since TS (Ti) < W-timestamp (Q). And also any transaction having TS (Tj) > TS (T<sub>17</sub>) must read the value of Q written by T<sub>17</sub>. Hence, rather than rolling back T<sub>16</sub> as the timestamp ordering protocol would have done, this {write} operation can be ignored. Otherwise this protocol is the same as the timestamp ordering protocol. Hence Thomas' Write Rule allows greater potential concurrency. Hence the modified rule for Thomas' Write rule is:

- Ø Ti issues write(Q):
  - i. TS(Ti) < R-timestamp(Q)  
Reject write (Q), roll back Ti
  - ii. TS(Ti) < W-timestamp(Q)  
Ti is attempting to write obsolete value of Q hence write (Q) operation is ignored.
  - iii. Otherwise execute write(Q) instruction  
Update W-timestamp (Q) = TS (Ti)

### Validation Based protocols:-

According to this protocol execution of transaction T<sub>i</sub> is done in two or three phases in its lifetime, depending on whether it is a read-only or an update transaction.

1. **Read phase:** Transaction T<sub>i</sub> reads the values of the various data items and writes only to temporary local variables of T<sub>i</sub> without updates to the actual database.
2. **Validation phase:** Transaction T<sub>i</sub> performs a “validation test” to determine if local variables can be written to the database without violating serializability.
3. **Write phase:** If T<sub>i</sub> is validated, the updates are applied to the database; otherwise, T<sub>i</sub> is rolled back.

Each transaction must go through the three phases in the order shown. But, the phases of concurrently executing transactions can be interleaved. To perform the validation test, each transaction T<sub>i</sub> has 3 timestamps

- **Start (T<sub>i</sub>):** the time when T<sub>i</sub> started its execution.
- **Validation (T<sub>i</sub>):** the time when T<sub>i</sub> entered its validation phase.
- **Finish (T<sub>i</sub>):** the time when T<sub>i</sub> finished its write phase.

Serializability order is determined by the timestamp-ordering technique, using the value of **Validation (T<sub>i</sub>)**. Thus the value TS (T<sub>i</sub>) = **Validation (T<sub>i</sub>)**. Hence, if TS (T<sub>j</sub>) < TS (T<sub>k</sub>), then any produced schedule must be equivalent to a serial schedule in which transaction T<sub>j</sub> appears before the transaction T<sub>k</sub>.

#### Validation Test for Transaction T<sub>j</sub>:

If for all T<sub>i</sub> with TS (T<sub>i</sub>) < TS (T<sub>j</sub>) either one of the following condition holds:

- **finish(T<sub>i</sub>) < start(T<sub>j</sub>)**
- **start (T<sub>j</sub>) < finish(T<sub>i</sub>) < validation(T<sub>j</sub>)** and the set of data items written by T<sub>i</sub> does not intersect with the set of data items read by T<sub>j</sub>.

then validation succeeds and  $T_j$  can be committed. Otherwise, validation fails and  $T_j$  is aborted.

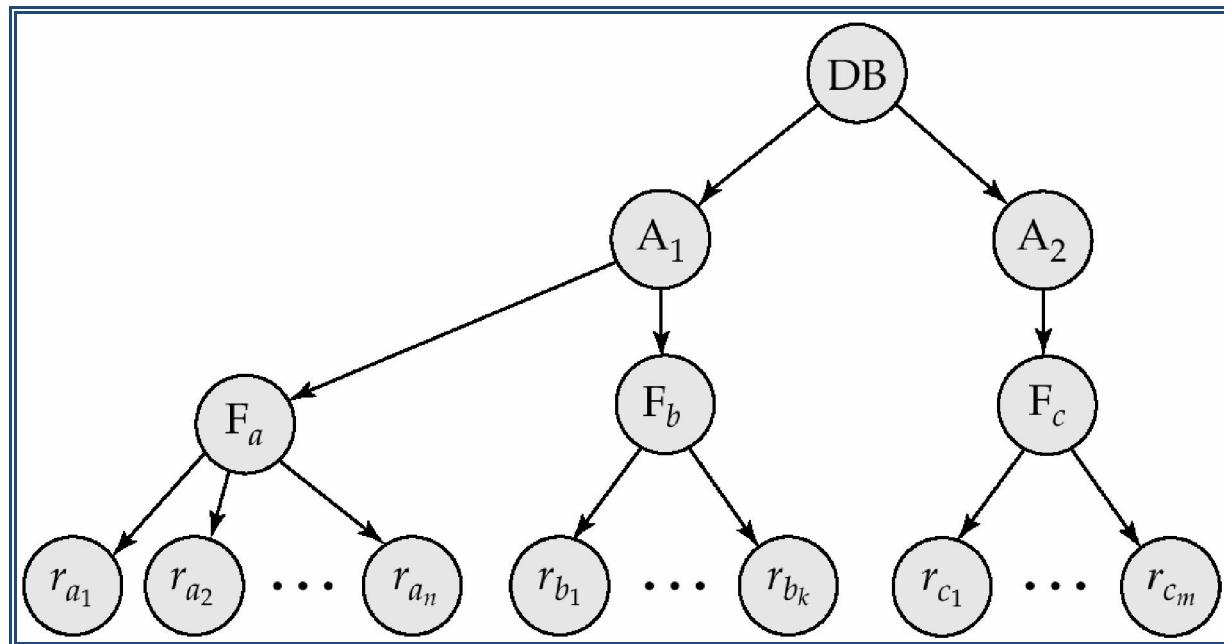
### Multiple granularity:-

The concurrency control scheme discussed so far treated each individual data item as a single unit on which concurrency was supposed to be maintained. But there are circumstances where grouping several data items together and treating them as a single unit is an advantage while working on them. For example suppose that any transaction wants to access the entire data items of the database, in such case, if we use locking technique then it is clear to us that we must lock each and every data item before we use them and finally unlock each and every data item after we complete our work on them. It is obvious that while locking and unlocking data items we are consuming some amount of time and if database contains large number of data items then this locking and unlocking time is significant and hence slows down the execution process. Hence it becomes better if the transaction can issue a single lock request to lock the entire database or only certain part of it.

Hence to apply this notion in concurrency control techniques, system need to define multiple levels of granularity. The levels, starting from the coarsest (top) level are

- *database*
- *area*
- *file*
- *record*

The figure below shows an example of multiple granularities.



In addition to S and X lock modes, there are three additional lock modes with multiple granularity:

- **intention-shared** (IS): indicates explicit locking at a lower level of the tree but only with shared locks.
- **intention-exclusive** (IX): indicates explicit locking at a lower level with exclusive or shared locks
- **shared and intention-exclusive** (SIX): the sub tree rooted by that node is locked explicitly in shared mode and explicit locking is being done at a lower level with exclusive-mode locks.

When a transaction locks a node, the transaction also has implicitly locked all the descendants of that node in the same lock mode. Intention locks allow a higher level node to be locked in S or X mode without having to check all descendent nodes. Hence, the compatibility matrix for all lock modes is as shown below:

|     | IS | IX | S | SIX | X |
|-----|----|----|---|-----|---|
| IS  | ✓  | ✓  | ✓ | ✓   | ✗ |
| IX  | ✓  | ✓  | ✗ | ✗   | ✗ |
| S   | ✓  | ✗  | ✓ | ✗   | ✗ |
| SIX | ✓  | ✗  | ✗ | ✗   | ✗ |
| X   | ✗  | ✗  | ✗ | ✗   | ✗ |

Transaction  $T_i$  can lock a node  $Q$ , using the following rules:

1. The lock compatibility matrix must be observed.
2. The root of the tree must be locked first, and may be locked in any mode.
3. A node  $Q$  can be locked by  $T_i$  in S or IS mode only if the parent of  $Q$  is currently locked by  $T_i$  in either IX or IS mode.
4. A node  $Q$  can be locked by  $T_i$  in X, SIX, or IX mode only if the parent of  $Q$  is currently locked by  $T_i$  in either IX or SIX mode.
5.  $T_i$  can lock a node only if it has not previously unlocked any node (that is,  $T_i$  is two-phase).
6.  $T_i$  can unlock a node  $Q$  only if none of the children of  $Q$  are currently locked by  $T_i$ .

Observe that locks are acquired in root-to-leaf order, whereas they are released in leaf-to-root order.

## Multiversion schemes:-

In order to maintain serializability, the techniques studied so far either delayed the transaction or totally aborted the operation. This could lead to undoing of significant work and hence there is more loss of time. Hence to avoid this, old copy of each data is kept in a system. i.e. in multiversion concurrency control scheme, each write( $Q$ ) creates a new version of  $Q$  and while reading, the concurrency control manager selects one of the version of  $Q$  to be read in the manner which ensures serializability.

### 1. Multiversion time stamp ordering:

With every data item  $Q$ , there exists sequence of newer version of that data item  $Q_k$ . There may exist 'n' version of  $Q$ . Each version contains field as described below:

- Content – the value of version( $Q_k$ )

- W-timestamp – timestamp of version that created  $Q_k$
- R-timestamp – largest time stamp of any transaction that read  $Q_k$

Suppose that transaction  $T_i$  issues read (Q) or write (Q) operation. Let  $Q_k$  denote the version of Q whose write timestamp is the largest write time stamp less than or equal to TS ( $T_i$ ).

- a. If transaction  $T_i$  issues read (Q), the value read is content of  $Q_k$  i.e. value returned is the content of version whose timestamp is the largest TS less than or equal to TS ( $T_i$ ).
- b. If  $T_i$  issues write(Q)
  - if  $TS(T_i) < R\text{-timestamp}(Q_k)$  the roll back  $T_i$ .
  - else if  $TS(T_i) = W\text{-timestamp}(Q_k)$  the overwrite the content of  $Q_k$ .
  - else create new version of Q

Example:

| T1       | T2        | Version | W-TS | R-TS |
|----------|-----------|---------|------|------|
| Read A   |           |         |      |      |
| A:=A+50  |           |         |      |      |
| Write(A) |           | A1=1050 | 1    | 1    |
|          | Read(A)   | A1=1050 | 1    | 2    |
|          | A:=A-50   |         |      |      |
|          | Write(A)  | A2=1000 | 2    | 2    |
|          | A:=A+1000 |         |      |      |
|          | Write(A)  | A2=2000 | 2    | 2    |

#### To delete the data item version no longer in use:

Compares the timestamp of every version with oldest version of transaction remaining and if  $TS(Q_k) < TS(T_i)_{\text{oldest}}$  then delete  $Q_k$ .

## 2. Multiversion two phase locking:

This protocol uses Multiversion timestamp protocol with the advantage of two phase locking protocol. This protocol differentiates between read only transaction and update transaction. Every transaction is assigned a timestamp before it is executed.

When  $T_i$  issues read (Q) instruction, it reads the value of appropriate version of  $Q_k$  based on the Multiversion time stamp ordering protocol.

When update transaction gets chance to execute, for reading purpose it gets a shared lock and reads the largest version of that item. When it wants to update an item, it gets exclusive lock on item and writes a newer version of that item and its timestamp is set initially to infinite. When update is completed, it sets the timestamp of that version of data and updates the counter.

Versions are deleted in a manner like that of Multiversion timestamp ordering protocol.

## Deadlock handling:

The state where exists a set of waiting transactions such that every transaction in the set is waiting for another transaction in the set is called deadlock state. Consider the following two transactions:

| $T_1$                         | $T_2$                                             |
|-------------------------------|---------------------------------------------------|
| <b>lock-X(×)</b><br>write (×) |                                                   |
| <b>lock-X(Y)</b>              | <b>lock-X(Y)</b><br>write (Y)<br><b>lock-X(×)</b> |

There are two principal methods for dealing with the *deadlock* problem:

**Deadlock prevention protocol:** We use this protocol to ensure that the system will *never* enter into a deadlock state.

**Deadlock detection and deadlock recovery scheme:** Alternatively, we can allow the system to enter a deadlock state, and then try to recover by using a deadlock detection and deadlock recovery scheme.

### There are two approaches to deadlock prevention.

- No cyclic waits can occur by ordering the requests for locks, or requiring all locks to be acquired together.
- Performs transaction rollback instead of waiting for a lock.

#### Schemes under the first approach:

Each transaction locks all its data items before it begins execution (pre- declaration). Impose partial ordering of all data items and require that a transaction can lock data items only in the order specified by the partial order (graph-based protocol).

#### Schemes under the second approach:

Use preemption and transaction rollbacks. In preemption, when a transaction  $T_2$  requests a lock that transaction  $T_1$  holds, the lock granted to  $T_1$  may be **preempted** by rolling back of  $T_1$  and granting of the lock to  $T_2$ . To control the preemption, we assign a unique timestamp to each transaction.

- Ø **Wait-die scheme:** This is nonpreemptive scheme. Older transaction may wait for younger one to release data item. Younger transactions never wait for older ones; they are rolled back instead. A transaction may die several times before acquiring needed data item.
- Ø **Wound-wait scheme:** This is preemptive scheme. Older transaction *wounds* (forces rollback) of younger transaction instead of waiting for it. Younger transactions may wait for older ones. May be fewer rollbacks than *wait-die* scheme.

Both in *wait-die* and in *wound-wait* schemes, a rolled back transaction is restarted with its original timestamp. Older transactions thus have precedence over newer ones, and **starvation** is hence avoided.

#### Ø **Timeout-Based Schemes:**

A transaction waits for a lock only for a specified amount of time. After that, the wait times out and the transaction is rolled back. Thus deadlocks are not possible. Simple to implement; but starvation is possible.

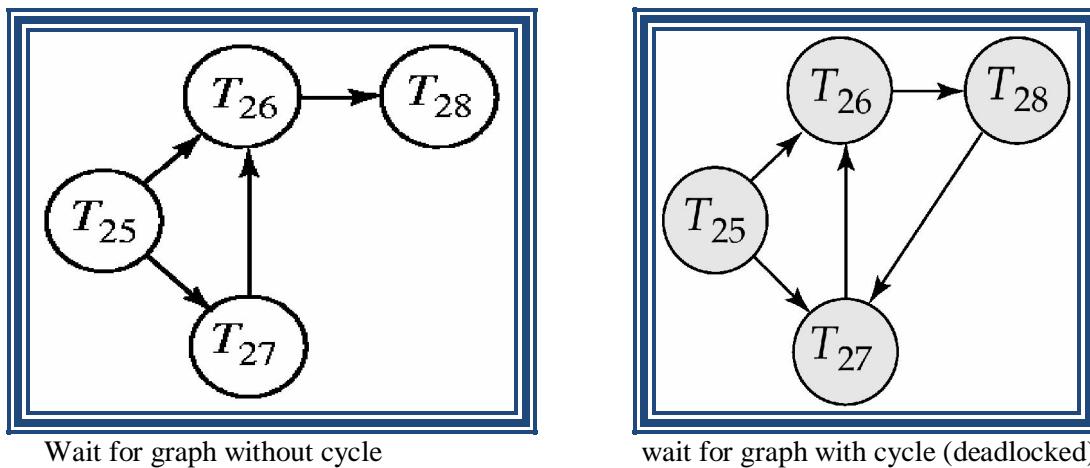
## Dead lock detection and recovery:

Deadlocks can be described as a *wait-for graph*, which consists of a pair  $G = (V, E)$ ,

$V$  is a set of vertices (all the transactions in the system)

$E$  is a set of edges; each element is an ordered pair  $T_i \rightarrow T_j$ .

If  $T_i \rightarrow T_j$  is in  $E$ , then there is a directed edge from  $T_i$  to  $T_j$ , implying that  $T_i$  is waiting for  $T_j$  to release a data item. When  $T_i$  requests a data item currently being held by  $T_j$ , then the edge  $T_i \rightarrow T_j$  is inserted in the wait-for graph. This edge is removed only when  $T_j$  is no longer holding a data item needed by  $T_i$ . *The system is in a deadlock state if and only if the wait-for graph has a cycle.* DBMS must invoke a deadlock-detection algorithm periodically to look for cycles.



**When deadlock is detected three actions need to be taken to recover from the deadlock:**

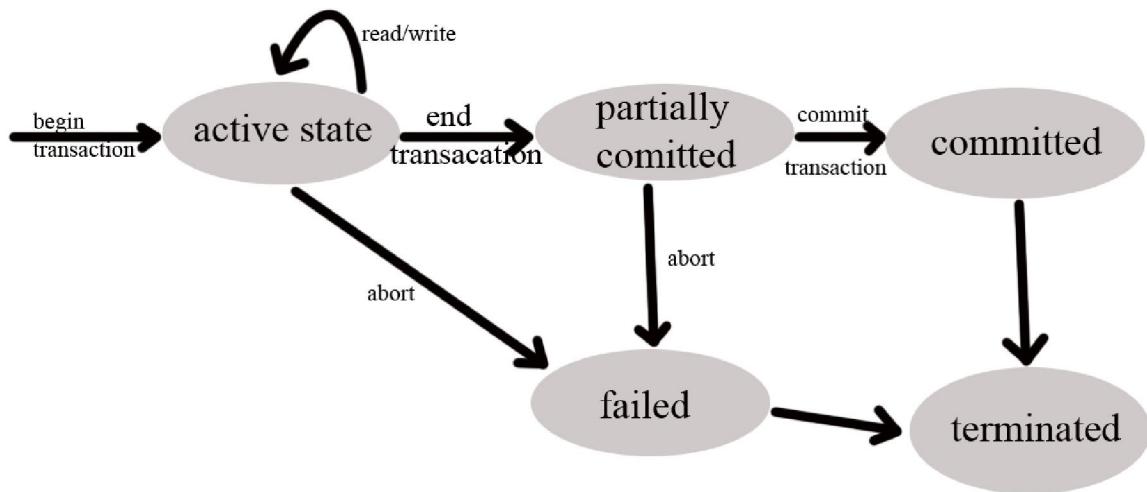
- Ø **Selection of a victim:** Some transaction will have to rolled back to break deadlock. Select that transaction as victim that will incur minimum cost.
- Ø **Rollback:** determines how far to roll back transaction i.e. it can be rolled back to its entirety or partially.
- Ø **Starvation:** Starvation happens if same transaction is always chosen as victim. Include the number of rollbacks in the cost factor to avoid starvation.

## Transaction processing:

### Transaction:

- It is the atomic unit of work that is either completed in its entirety or not at all
- It may also be defined as the unit or piece of program execution that accesses and possibly updates the various data items.
- Operations recorded by the recovery manager from beginning to end of transaction are:
  - Begin\_transaction: notes the start of transaction.
  - Read/write: specify read or write operation on database.
  - End transaction: specify read or write has ended and transaction has ended. At this point it is necessary to check whether the transaction should be committed or aborted.
  - Commit\_transaction: signals successful end of transaction.
  - Rollback or abort: signals unsuccessful transaction.

State transition diagram illustrating the state of transaction execution.



### Desirable properties of transaction:

To ensure integrity of data, we require the database system maintain the following properties of the transaction.

1. Atomicity: either all operations of a transaction is reflected in database or none.
2. Consistency: execution of a transaction in isolation preserves consistency of database.
3. Isolation: even though multiple transactions are executing concurrently, each transaction is unaware of other transaction.
4. Durability: after a transaction completes, the changes it has made persists.

These properties are also called **ACID** properties.

Example:

```
Let, Ti: read(A);
      A:=A-50;
      Write(A);
      Read(B);
      B:=B+50;
      Write(B);
```

Consistency:

In this case consistency requirement here is that the total sum of A and B should always be same i.e. before and after the transaction is executed. In this case consistency property of the transaction prevents the unwanted creation of a destroying of money.

Ensuring consistency for an individual transaction is the responsibility of the application programmer who codes the transaction.

Atomicity:

Suppose that failure occur after write(A) operation but before write(B) operation. Hence the values reflected in the database becomes A=950 and B=2000, which is not the true value. Hence transaction should be completed in its entirety or not at all.

Durability:

Once the execution of the transaction completed successfully and the user who initiated the transaction has been notified that the transfer of the fund has taken place, it must be the case that no system failure will result in a loss of data corresponding to this transfer of funds.

Isolation:

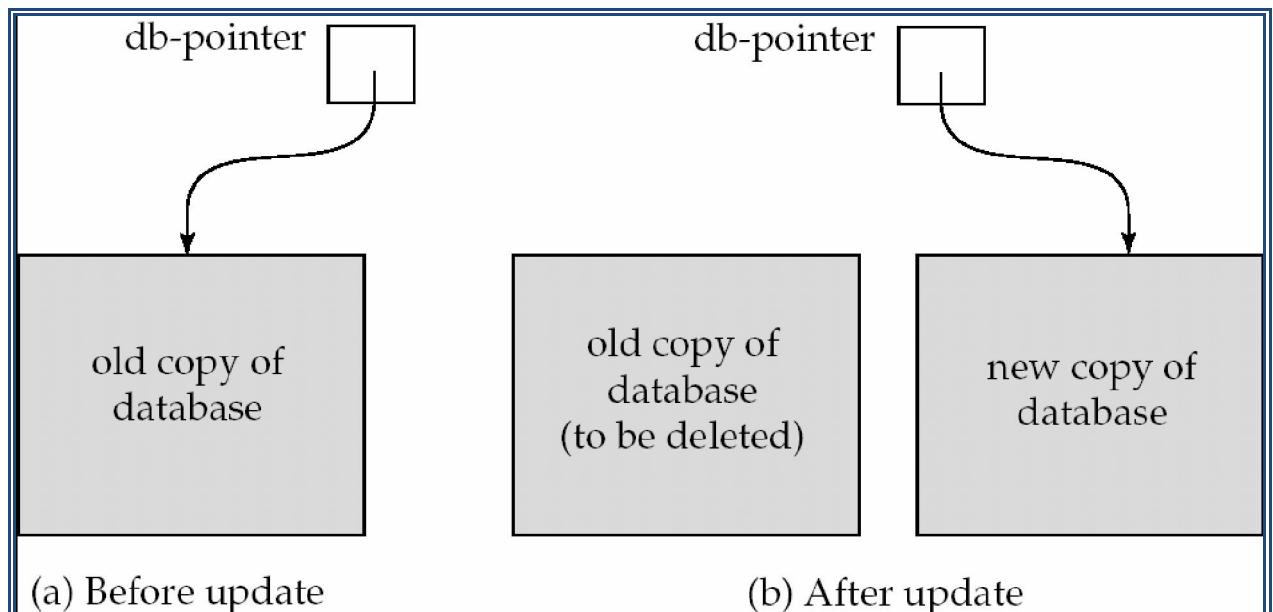
Let a transaction T1 transfer Rs 50 from account A to B. While T1 reads and subtracts 50 from account A but before it was updated to account B a new transaction (T2) interrupts which reads from account A and B. the value read by transaction T2 is A=950 and B=2000 (initial values of A=1000 and B=2000).

Hence in order to maintain consistency isolation of transaction is required. In order to maintain isolation serial execution of transaction and use of concurrency control mechanism should be used.

## Implementation of Atomicity and Durability: Shadow copy scheme.

Assume:

- Only one transaction is active at a time.
- Database is simply a file on disk.



- If transaction completes successfully, it is updated in the new copy and then db-pointer is updated. Now this new copy becomes the current copy of database and old copy is deleted.
- If transaction is unsuccessful, then new copy is deleted and old copy is again the current one.
- The implementation actually depends on the write to db-pointer being atomic.i.e either all its bytes are written or none of its bytes are written.

Drawbacks:

- Implementation is extremely inefficient in the context of large database since executing single transaction requires copying the entire database.
- Does not allow transaction to execute concurrently.

## Concurrent Executions:

Transaction processing system usually allows multiple transactions to run concurrently but running multiple transactions at the same time has complications with the preservation of consistency of the data. There are various techniques by which we can conquer this problem. Hence reasons for using multiple transactions at the same time are as discussed below:

### 1. Improved throughput and resource utilization:

Since we know that a transaction comprises of so many steps like reading from disk, using CPU. Hence we take this feature as gratitude and use it for the parallelism of transaction. For example, if one transaction is using CPU, another can access the database files at the same time. Hence number of transactions executed in a given amount of time increases which is also known as improved throughput. And it also increases the

utilization of resources. i.e. at the same time more than one source can be used concurrently.

## 2. Reduced waiting time:

The total execution time of a transaction depends upon the complexity of the transaction. Hence the execution time for different transaction is different. In a scenario where long transaction is followed by short one, in the case of transactions running serially, short transaction will have to wait for long time in order to complete. Hence running these transactions concurrently reduces the average response time (average time for a transaction to be completed after it has been submitted).

Let us consider some schedules and study them briefly:

|             |             |         |
|-------------|-------------|---------|
| T1: read(A) |             | A=1000  |
| A:=A-50     |             | A=950   |
| Write(A)    |             | A=950   |
| Read(B)     |             | B=2000  |
| B:=B+50     |             | B=2050  |
| Write(B)    |             | B=2050  |
|             | T2: read(A) | A=950   |
|             | temp:=A*0.1 | temp=95 |
|             | A:=A-temp   | A=855   |
|             | Write(A)    | A=855   |
|             | Read(B)     | B=2050  |
|             | B:=B+temp   | B=2145  |
|             | Write(B)    | B=2145  |

Fig 1: Schedule 1

In the schedule shown above consistency by transferring Rs 50 from account A to account B is preserved. Since  $(A+B)_{initial} = (A+B)_{final}$

Similarly, if the order of execution of transaction is reversed.i.e. execute T2 first and then execute T1, then also  $(A+B)_{initial} = (A+B)_{final}$   
Hence consistency of data is preserved.

|             |  |        |
|-------------|--|--------|
| T1: read(A) |  | A=1000 |
| A:=A-50     |  | A=950  |

|                                |                                                          |                                    |
|--------------------------------|----------------------------------------------------------|------------------------------------|
| Write(A)                       |                                                          | A=950                              |
|                                | T2:      read(A)<br>temp:=A*0.1<br>A:=A-temp<br>Write(A) | A=950<br>temp=95<br>A=855<br>A=855 |
| Read(B)<br>B:=B+50<br>Write(B) |                                                          | B=2000<br>B=2050<br>B=2050         |
|                                | Read(B)<br>B:=B+temp<br>Write(B)                         | B=2050<br>B=2145<br>B=2145         |

Fig: schedule 2

Here

$$(A+B)_{\text{initial}} = (A+B)_{\text{final}}$$

Hence consistency is preserved.

|                                            |                                                                     |                                                |
|--------------------------------------------|---------------------------------------------------------------------|------------------------------------------------|
| T1:      read(A)<br>A:=A-50                |                                                                     | A=1000<br>A=950                                |
|                                            | T2:      read(A)<br>temp:=A*0.1<br>A:=A-temp<br>Write(A)<br>Read(B) | A=1000<br>temp=100<br>A=900<br>A=900<br>B=2000 |
| Write(A)<br>Read(B)<br>B:=B+50<br>Write(B) |                                                                     | A=950<br>B=2000<br>B=2050<br>B=2050            |
|                                            | B:=B+temp<br>Write(B)                                               | B=2150<br>B=2150                               |

Fig: Schedule 3

Here,

$$\begin{aligned}(A+B) \text{ initial} &= 1000+2000 = 3000 \\(A+B) \text{ final} &= 950 + 150 = 3100\end{aligned}$$

Hence consistency of data in schedule 3 is not maintained.

## Serializability:

**Basic Assumption** – Each transaction preserves database consistency. Thus serial execution of a set of transactions preserves database consistency. A (possibly concurrent) schedule is serializable if it is equivalent to a serial schedule. Different forms of schedule equivalence give rise to the notions of:

1. conflict serializability
2. view serializability

## Conflict serializability:

We ignore operations other than **read** and **write** instructions, and we assume that transactions may perform arbitrary computations on data in local buffers in between reads and writes. Our simplified schedules consist of only **read** and **write** instructions.

Instructions  $l_i$  and  $l_j$  of transactions  $T_i$  and  $T_j$  respectively, **conflict** if and only if there exists some item  $Q$  accessed by both  $l_i$  and  $l_j$ , and at least one of these instructions wrote  $Q$ .

- |                                                        |                                 |
|--------------------------------------------------------|---------------------------------|
| 1. $l_i = \text{read}(Q)$ , $l_j = \text{read}(Q)$ .   | $l_i$ and $l_j$ don't conflict. |
| 2. $l_i = \text{read}(Q)$ , $l_j = \text{write}(Q)$ .  | They conflict.                  |
| 3. $l_i = \text{write}(Q)$ , $l_j = \text{read}(Q)$ .  | They conflict                   |
| 4. $l_i = \text{write}(Q)$ , $l_j = \text{write}(Q)$ . | They conflict                   |

Intuitively, a conflict between  $l_i$  and  $l_j$  forces a (logical) temporal order between them.

If  $l_i$  and  $l_j$  are consecutive in a schedule and they do not conflict, their results would remain the same even if they had been interchanged in the schedule. Schedule 3 can be transformed into Schedule 6, a serial schedule where  $T_2$  follows  $T_1$ , by series of swaps of non-conflicting instructions.

The diagram illustrates the transformation of Schedule 3 into Schedule 6. On the left, a double-headed arrow points from a table labeled "Schedule 3" to another table labeled "schedule 6".

**Schedule 3:**

| $T_1$               | $T_2$               |
|---------------------|---------------------|
| read(A)<br>write(A) | read(A)<br>write(A) |
| read(B)<br>write(B) | read(B)<br>write(B) |

**schedule 6:**

| $T_1$                                      | $T_2$                                      |
|--------------------------------------------|--------------------------------------------|
| read(A)<br>write(A)<br>read(B)<br>write(B) | read(A)<br>write(A)<br>read(B)<br>write(B) |

Therefore Schedule 3 is conflict serializable.

## View serializability:

Let  $S$  and  $S'$  be two schedules with the same set of transactions.  $S$  and  $S'$  are **view equivalent** if the following three conditions are met:

1. For each data item  $Q$ , if transaction  $T_i$  reads the initial value of  $Q$  in schedule  $S$ , then transaction  $T_i$  must, in schedule  $S'$ , also read the initial value of  $Q$ .
2. For each data item  $Q$  if transaction  $T_i$  executes **read( $Q$ )** in schedule  $S$ , and that value was produced by transaction  $T_j$  (if any), then transaction  $T_i$  must in schedule  $S'$  also read the value of  $Q$  that was produced by transaction  $T_j$ .
3. For each data item  $Q$ , the transaction (if any) that performs the final **write( $Q$ )** operation in schedule  $S$  must perform the final **write( $Q$ )** operation in schedule  $S'$ .

As can be seen, view equivalence is also based purely on **reads** and **writes** alone. A schedule  $S$  is **view serializable** if it is view equivalent to a serial schedule. Every conflict serializable schedule is also view serializable. Below is a schedule which is view-serializable but *not* conflict serializable.

## Recoverable schedules:

| $T_8$                        | $T_9$                       |
|------------------------------|-----------------------------|
| <b>read(<math>A</math>)</b>  |                             |
| <b>write(<math>A</math>)</b> |                             |
| <b>read(<math>B</math>)</b>  | <b>read(<math>A</math>)</b> |

Let us consider the schedule as shown. Transaction  $T_9$  gets chance to get executed after the **write( $A$ )** instruction of transaction  $T_8$ , and since it has only one read instruction  $T_9$  gets committed before  $T_8$  does. Now at this point let us suppose that  $T_9$  is committed and immediately after that system crashes leading to situation where  $T_8$  is in failed state whereas  $T_9$  is committed. Since the value read by  $T_9$  was written by transaction  $T_8$  hence we must abort both  $T_8$  and  $T_9$  to satisfy the atomicity property of transaction. But, since  $T_9$  is already committed, it cannot be aborted which ultimately leads us to the state from where we cannot recover from failure of transaction  $T_8$ .

From the example given above what we can conclude is that a recoverable schedule is the one where, for each pair of transaction  $T_i$  and  $T_j$  such that  $T_j$  reads data item previously written by  $T_i$ , the commit operation of  $T_i$  appears before the commit operation of  $T_j$ .

## Cascadeless Schedules:

| $T_{10}$                     | $T_{11}$                     | $T_{12}$                    |
|------------------------------|------------------------------|-----------------------------|
| <b>read(<math>A</math>)</b>  |                              |                             |
| <b>read(<math>B</math>)</b>  |                              |                             |
| <b>write(<math>A</math>)</b> | <b>read(<math>A</math>)</b>  |                             |
|                              | <b>write(<math>A</math>)</b> |                             |
|                              |                              | <b>read(<math>A</math>)</b> |

Even if schedule is recoverable, to recover correctly from the failure of transaction  $T_i$ , we may have to roll back several transactions. Such situation occurs if transactions have read data written by  $T_i$ . Consider the following schedules where none of the transaction has yet committed. If  $T_{10}$ , fails,  $T_{11}$  and

$T_{12}$  must also be rolled back. This phenomena in which single transaction leads to a series of transaction rollback is called cascading rollback.

Since cascading rollback leads to undoing of significant amount of work, hence cascading rollback is undesirable. A cascadeless schedule is one where, for each pair of transaction  $T_i$  and  $T_j$  such that  $T_j$  reads the data item previously written by  $T_i$ , the commit operation of  $T_i$  appears before the read operation of  $T_j$ .

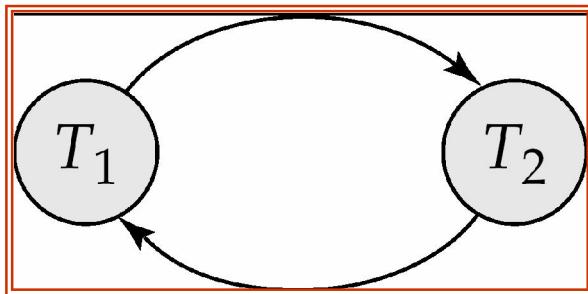
### Testing for serializability:

Consider some schedule of a set of transactions  $T_1, T_2, \dots, T_n$ . **Precedence graph** — a direct graph where the vertices are the transactions (names). We draw an arc from  $T_i$  to  $T_j$  if the two transaction conflict, and  $T_i$  accessed the data item on which the conflict arose earlier. We may label the arc by the item that was accessed.



For example, the precedence graph for schedule 1 contains the single stage  $T_1 \rightarrow T_2$ , since all the instructions of  $T_1$  are executed before the first instruction of the  $T_2$  transaction and hence the precedence graph of the schedule 1 becomes fig (a) as shown above. Similarly if  $T_2$  is followed by  $T_1$  then the precedence graph becomes fig (b). Since both fig doesn't have cycle, they are conflict serializable.

The precedence graph for the schedule 3 is as shown below. It contains the edge  $T_1 \rightarrow T_2$ , because  $T_1$  executes read(A) before  $T_2$  executes write(A). It also contains the edge  $T_2 \rightarrow T_1$ , because  $T_2$  executes read(B) before  $T_1$  executes write(B).



Since the precedence graph of schedule 3 has a loop, schedule 3 is not conflict serializable.