Lecture Notes
on
Database Management System

Dharmendra Kumar

December 18, 2021

Contents

Chapter 1

INTRODUCTION TO DATABASE SYSTEMS

1.1 Database

- It is a collection of interrelated data of an enterprise in a particular subject.
- It is a collection of data in an organized manner in a persistent media so that storing and retrieving data will be easier.

For example, a university database might contain information about the following:-

Entities such as students, faculty, courses, and classrooms.
Relationships between entities, such as students' enrollment in courses, faculty teaching courses, and the use of rooms for courses.

1.2 Database Management System(DBMS)

- It is a collection of programs(software, which is used to create, manipulate(insert, delete, retrieve, update) data in database.
- The primary goal of a DBMS is to provide a way to store and retrieve database information that is both convenient and efficient.

1.3 Applications of DBMS

Databases are widely used. Here are some representative applications:

- **Banking:** In banking database, we record the information about customers, accounts, loans and banking transactions.
- **Railway reservation system:** we record the information about trains, reservation, passengers.
- **Airlines:** We record reservations and schedule information, flight information.
- **Universities:** We record student information, course registrations, and grades.

- **Credit card transactions:** We record purchases on credit cards and generation of monthly statements.
- **Telecommunication:** For keeping records of calls made, generating monthly bills, maintaining balances on prepaid calling cards, and storing information about the communication networks.
- **Sales:** For customer, product, and purchase information.
- **Human resources:** For information about employees, salaries, payroll taxes and benefits, and for generation of paychecks.
- **Hospital system:** We record information about doctors, patients, services available, rooms details, employees etc.
- **Hotel system:** We record information about rooms,customers and employees details.

1.4 Drawbacks of File system

To keep information in such file-processing system, there are a number of major disadvantages:-

Data redundancy: Data redundancy refers to the duplication of data, lets say we are managing the data of a college where a student is enrolled for two courses, the same student details in such case will be stored twice, which will take more storage than needed. Data redundancy often leads to higher storage costs and poor access time.

Data inconsistency: Data redundancy leads to data inconsistency, lets take the same example that we have taken above, a student is enrolled for two courses and we have student address stored twice, now lets say student requests to change his address, if the address is changed at one place and not on all the records then this can lead to data inconsistency.

Difficulty in accessing data: In file system the data is stored in the files. Whenever data has to be retrieved as per the requirements then a new application program has to be written. This is tedious process.

Data isolation: Because data are scattered in various files, and files may be in different formats, writing new application programs to retrieve the appropriate data is difficult.

Integrity problems: The data values stored in the database must satisfy certain types of consistency constraints.
For example, the balance of a bank account may never fall below a prescribed amount. These constraints are enforced in the system by adding appropriate code in the various application programs. However, when new constraints are added, it is difficult to change the programs to enforce them. The problem is compounded when constraints involve several data items from different files.

Atomicity problems: Atomicity of a transaction refers to “All or nothing”, which

means either all the operations in a transaction executes or none.
For example: Lets say Steve transfers 100$ to Negan's account. This transaction consists multiple operations such as debit 100$ from Steve's account, credit 100$ to Negan's account. Like any other device, a computer system can fail lets say it fails after first operation then in that case Steve's account would have been debited by 100$ but the amount was not credited to Negan's account, in such case the rollback of operation should occur to maintain the atomicity of transaction. It is difficult to achieve atomicity in file processing systems.

Concurrent-access anomalies: Concurrent access means multiple users can access database simultaneously. In such an environment, interaction of concurrent updates may result in inconsistent data.
Consider bank account A, containing $500. If two customers withdraw funds (say $50 and $100 respectively) from account A at about the same time, the result of the concurrent executions may leave the account in an incorrect (or inconsistent) state. There is no central control of data in classical file organization. So, the concurrent access of data by many users is difficult to implement.

Security problems: Data should be secured from unauthorised access, for example a student in a college should not be able to see the payroll details of the teachers, such kind of security constraints are difficult to apply in file processing systems.
Not every user of the database system should be able to access all the data.

1.5 Advantage of DBMS over file system

There are several advantages of Database management system over file system. Few of them are as follows:
No redundant data: Redundancy removed by data normalization. No data duplication saves storage and improves access time.

Data Consistency and Integrity: As we discussed earlier the root cause of data inconsistency is data redundancy, since data normalization takes care of the data redundancy, data inconsistency also been taken care of as part of it.

Data Security: It is easier to apply access constraints in database systems so that only authorized user is able to access the data. Each user has a different set of access thus data is secured from the issues such as identity theft, data leaks and misuse of data.

Privacy: Limited access means privacy of data.

Easy access to data: Database systems manages data in such a way so that the data is easily accessible with fast response times.

Easy recovery: Since database systems keeps the backup of data, it is easier to do a full recovery of data in case of a failure.

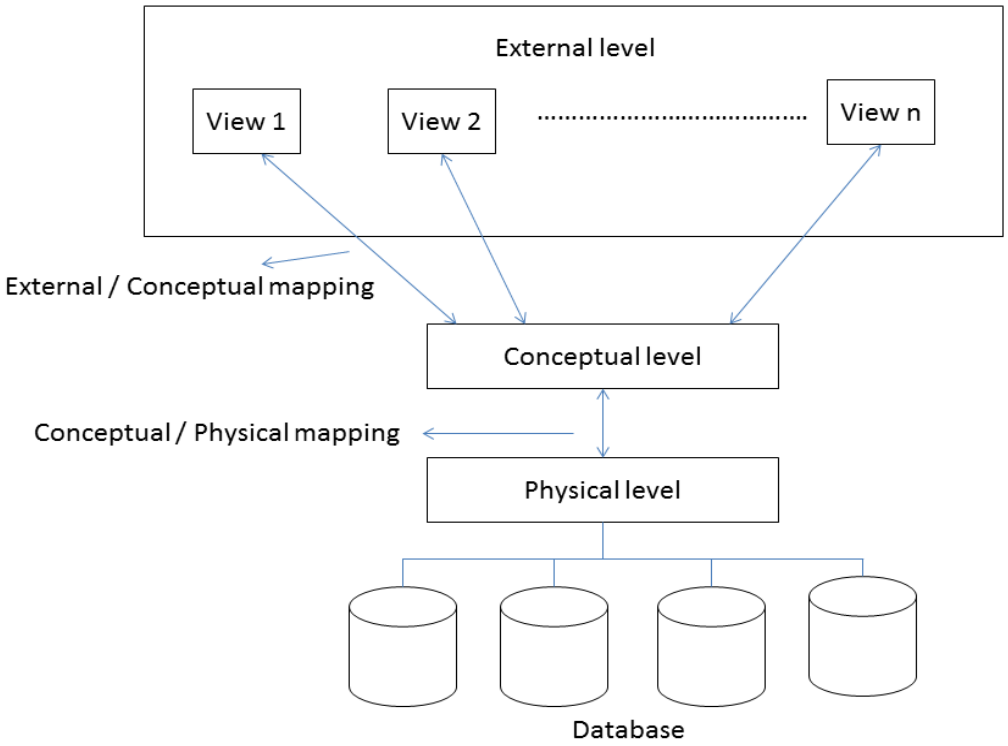
Flexible: Database systems are more flexible than file processing systems.

1.6 Disadvantages of DBMS

- DBMS implementation cost is high compared to the file system
- Complexity: Database systems are complex to understand
- Performance: Database systems are generic, making them suitable for various applications. However this feature affect their performance for some applications

1.7 Data abstraction model or Three level data abstraction architecture

Data abstraction means to hide the information. To hide the information from some users, three levels of abstraction is used.



This architecture has three levels:
1. External level
2. Conceptual level
3. Physical level

1. External level
It is also called view level. The reason this level is called "view" is because several users can view their desired data from this level which is internally fetched from database with the help of conceptual and internal level mapping. The user doesn't need to know the database schema details such as data structure, table definition etc. User is only concerned about data which is what returned back to the view level after it has been fetched from database (present at the internal level). External level is the "top level" of the

Three Level DBMS Architecture.

2. Conceptual level
It is also called logical level. The conceptual level describes the structure of the whole database. The conceptual schema hides the details of physical storage structures and concentrates on describing entities, data types, relationships, user operations, and constraints. Usually, a representational data model is used to describe the conceptual schema when a database system is implemented.

3. Physical level
This level is also known as internal level. This level describes how the data is actually stored in the storage devices. This level is also responsible for allocating space to the data. The internal level describes the physical storage structure of the database. The internal schema uses a physical data model and describes the complete details of data storage and access paths for the database. This is the lowest level of the architecture.

1.8 Instances and Schema

1.8.1 Schema

The overall design of the database is called the database schema.

Schema is of three types: Physical schema, logical schema and view schema.
Physical schema: The design of a database at physical level is called physical schema, how the data stored in blocks of storage is described at this level.
Logical schema: Design of database at logical level is called logical schema, programmers and database administrators work at this level, at this level data can be described as certain types of data records gets stored in data structures, however the internal details such as implementation of data structure is hidden at this level (available at physical level).
View schema: Design of database at view level is called view schema. This generally describes end user interaction with database systems.

1.8.2 Instance

The collection of information stored in the database at a particular moment is called an instance of the database.

1.9 Data Independence

Data independence can be explained using the three-schema architecture.
Data independence refers characteristic of being able to modify the schema at one level of the database system without altering the schema at the next higher level.
There are two types of data independence:

1.9.1 Logical Data Independence

- Logical data independence refers characteristic of being able to change the conceptual schema without having to change the external schema.
- Logical data independence is used to separate the external level from the conceptual view.
- If we do any changes in the conceptual view of the data, then the user view of the data would not be affected.
- Logical data independence occurs at the user interface level.

1.9.2 Physical Data Independence

- Physical data independence can be defined as the capacity to change the internal schema without having to change the conceptual schema.
- If we do any changes in the storage size of the database system server, then the Conceptual structure of the database will not be affected.
- Physical data independence is used to separate conceptual levels from the internal levels.
- Physical data independence occurs at the logical interface level.

1.10 Data Models

Data model is a collection of conceptual tools for describing data, data relationships, data semantics, and consistency constraints. Data models define how the logical structure of a database is modeled. Data models define how data is connected to each other and how they are processed and stored inside the system.
There are different types of data models are used in DBMS. Here, we are going to explain some data models.
(1) Entity relationship model
(2) Relational model
(3) Network model
(4) Hierarchical model

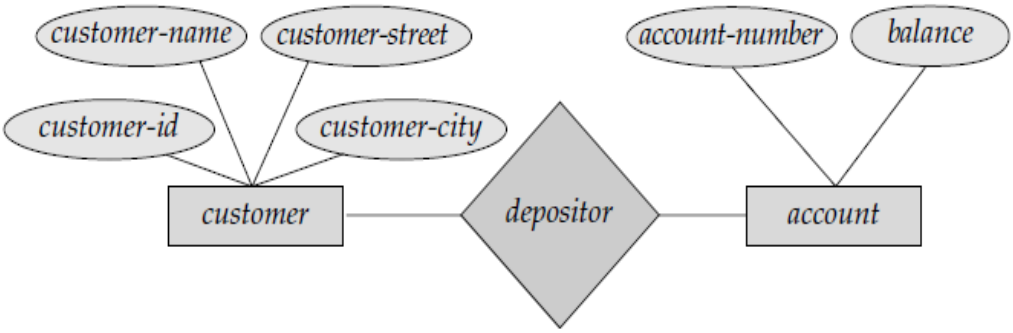
Entity-relationship model

The entity-relationship (E-R) data model is based on a perception of a real world that consists of a collection of basic objects, called entities, and of relationships among these objects. An entity is a "thing" or "object" in the real world that is distinguishable from other objects. For example, each person is an entity, and bank accounts can be considered as entities.
Entities are described in a database by a set of attributes. For example, the attributes account-number and balance may describe one particular account in a bank, and they form attributes of the account entity set. Similarly, attributes customer-name, customer-street address and customer-city may describe a customer entity.

A relationship is an association among several entities. For example, a depositor relationship associates a customer with each account that she has. The set of all entities of the same type and the set of all relationships of the same type are termed an entity set and relationship set, respectively.
The overall logical structure (schema) of a database can be expressed graphically by an E-R diagram, which is built up from the following components:

- **Rectangles**, which represent entity sets
- **Ellipses**, which represent attributes
- **Diamonds**, which represent relationships among entity sets
- **Lines**, which link attributes to entity sets and entity sets to relationships

For example, consider part of a database banking system consisting of customers and of the accounts that these customers have. Following figure shows the corresponding E-R diagram. The E-R diagram indicates that there are two entity sets, customer and account, with attributes as outlined earlier. The diagram also shows a relationship depositor between customer and account.



E-R diagram

Relational model

The relational model uses a collection of tables to represent both data and the relationships among those data. Each table has multiple columns, and each column has a unique name. Following figure presents a sample relational database comprising three tables: One shows details of bank customers, the second shows accounts, and the third shows which accounts belong to which customers.

The relational model is an example of a record-based model. Record-based models are so named because the database is structured in fixed-format records of several types. Each table contains records of a particular type. Each record type defines a fixed number of fields, or attributes. The columns of the table correspond to the attributes of the record type.

customer-id	*customer-name*	*customer-street*	*customer-city*
192-83-7465	Johnson	12 Alma St.	Palo Alto
019-28-3746	Smith	4 North St.	Rye
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

Customer table

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

Account table

customer-id	*account-number*
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

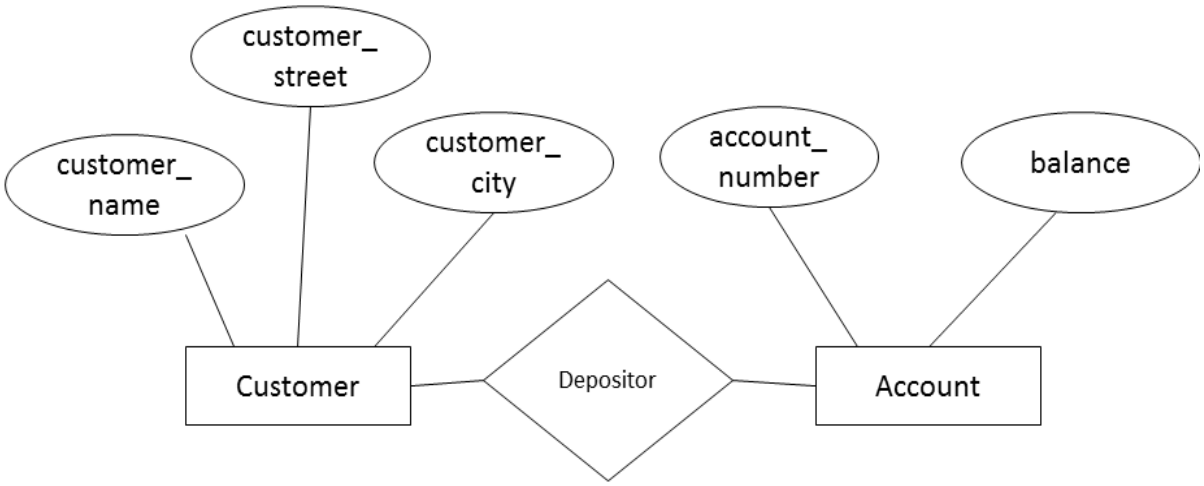
Depositor table

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.

Network model

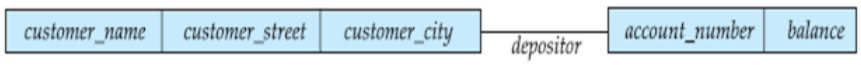
A network database consists of a collection of records connected to one another through links. A record is in many respects similar to an entity in the E-R model. Each record is a collection of fields (attributes), each of which contains only one data value. A link is an association between precisely two records. Thus, a link can be viewed as a restricted (binary) form of relationship in the sense of the E-R model.
A schema representing the design of a network database is said to be Data-structure diagram. Such a diagram consists of two basic components:
1. Boxes, which correspond to record types
2. Lines, which correspond to links
A data-structure diagram serves the same purpose as an E-R diagram; namely, it specifies the overall logical structure of the database. Every E-R diagrams can be transformed into their corresponding data-structure diagrams.

Example: Consider the following E-R diagram:-
This E-R diagram is consisting of two entity sets, customer and account, related through a binary, many-to-many relationship depositor, with no descriptive attributes. This diagram specifies that a customer may have several accounts, and that an account may belong to several different customers.
The data-structure diagram for the corresponding E-R diagram is the following:-
The record type customer corresponds to the entity set customer. It includes three fields—



E-R diagram

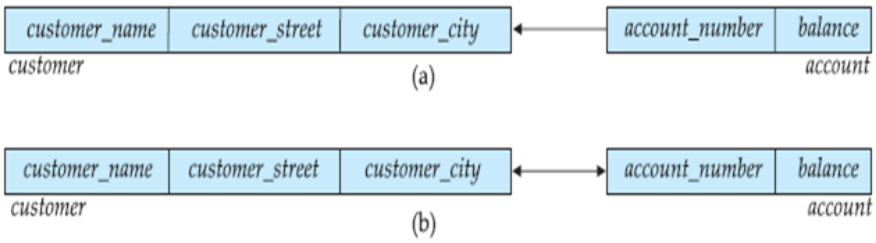
customer_name, customer_street, and customer_city. Similarly, account is the record type corresponding to the entity set account. It includes the two fields account_number and balance. Finally, the relationship depositor has been replaced with the link depositor.



Data-structure diagram

Figure 1.1: data-structure diagram

Here, the relationship depositor is many to many. If the relationship depositor were one to many from customer to account, then the link depositor would have an arrow pointing to customer record type. Similarly, if the relationship depositor were one to one, then the link depositor would have two arrows: one pointing to account record type and one pointing to customer record type.



Two data-structure diagram

A sample database corresponding to the data-structure diagram of Figure 1.1 is shown in Figure 1.2.

If a relationship includes descriptive attributes, the transformation from an E-R diagram to a data-structure diagram is more complicated. A link cannot contain any data value,

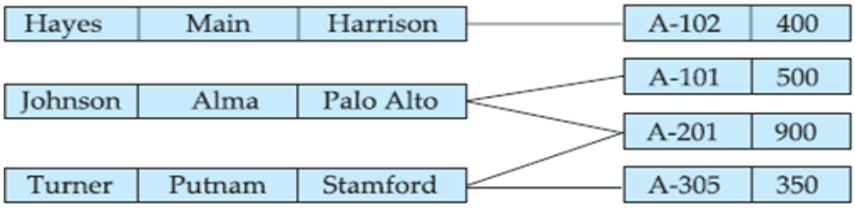
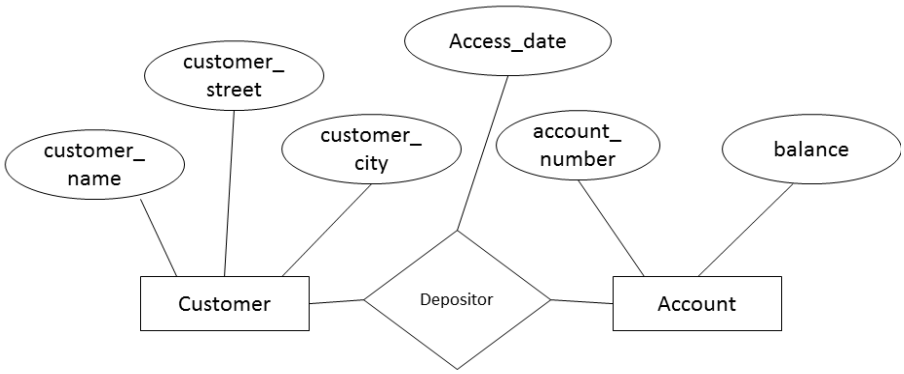


Figure 1.2: Sample database for data-structure diagram shown in figure 1.1

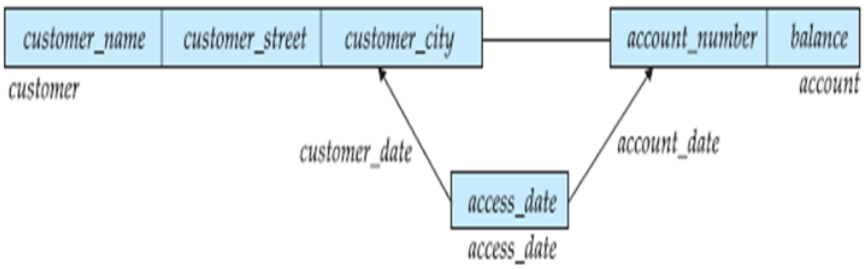
so a new record type needs to be created and links need to be established.

Example: Consider the following E-R diagram.



E-R diagram

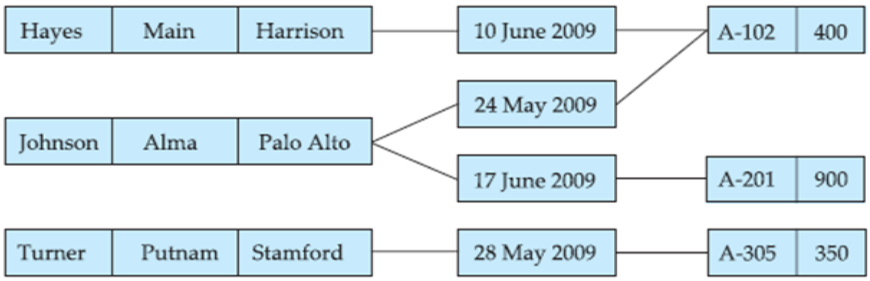
The data structure diagram of this E-R diagram is shown in the following figure:-



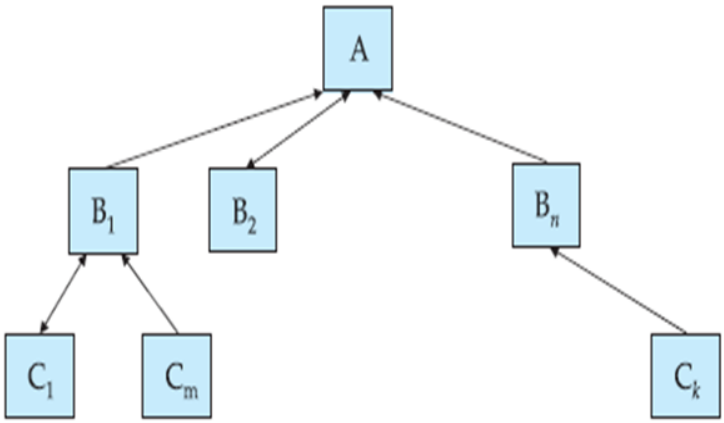
Sample database corresponding to above data-structure diagram is the following:-

Hierarchical model

A hierarchical database consists of a collection of records that are connected to each other through links. A record is similar to a record in the network model. Each record is a collection of fields (attributes), each of which contains only one data value. A link is an association between precisely two records. Thus, a link here is similar to a link in the network model.

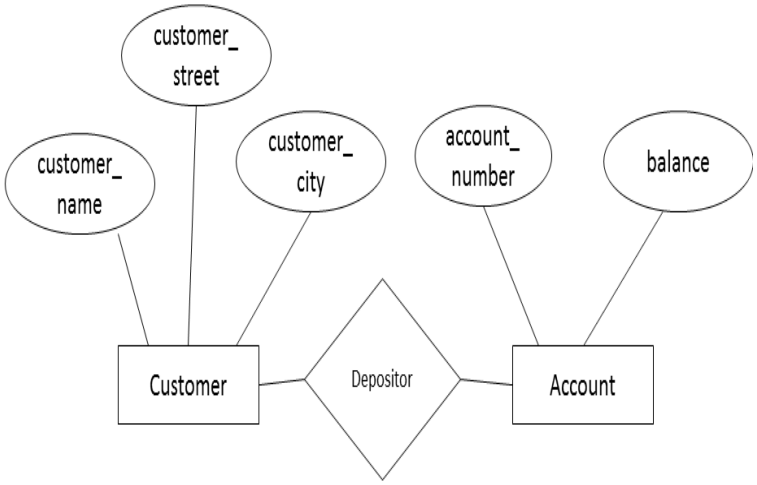


Consider a database that represents a customer-account relationship in a banking system. There are two record types: customer and account. The set of all customer and account records is organized in the form of a rooted tree, where the root of the tree is a dummy node. A hierarchical database is a collection of such rooted trees, and hence forms a forest. We shall refer to each such rooted tree as a database tree.
The schema for a hierarchical database is represented by tree-structured diagram. Such a diagram consists of two basic components:
1. Boxes, which correspond to record types
2. Lines, which correspond to links
A tree-structure diagram serves the same purpose as an entity–relationship (E-R) diagram; namely, it specifies the overall logical structure of the database. A tree structure diagram is similar to a data-structure diagram in the network model. The main difference is that, in the latter, record types are organized in the form of an arbitrary graph, whereas in the former, record types are organized in the form of a rooted tree. The relationships formed in the tree-structure diagram must be such that only one-to-many or one-to-one relationships exist between a parent and a child. The general form of a tree-structure diagram is shown in the following figure:-



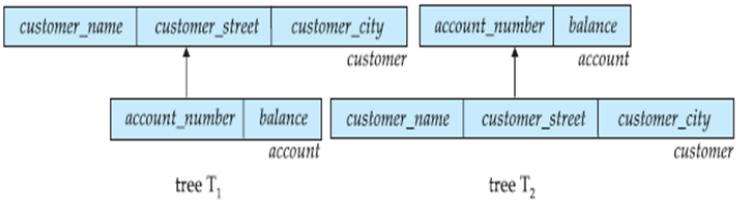
The database schema is represented as a collection of tree-structure diagrams. For each such diagram, there exists one single instance of a database tree. The root of this tree is a dummy node. The children of the dummy node are instances of the root record type in the tree-structure diagram. Each record instance may, in turn, have several children, which are instances of various record types, as specified in the corresponding tree-structure diagram. Every E-R diagram can be transformed into tree-structure diagram.

Example: Consider the following E-R diagram:-

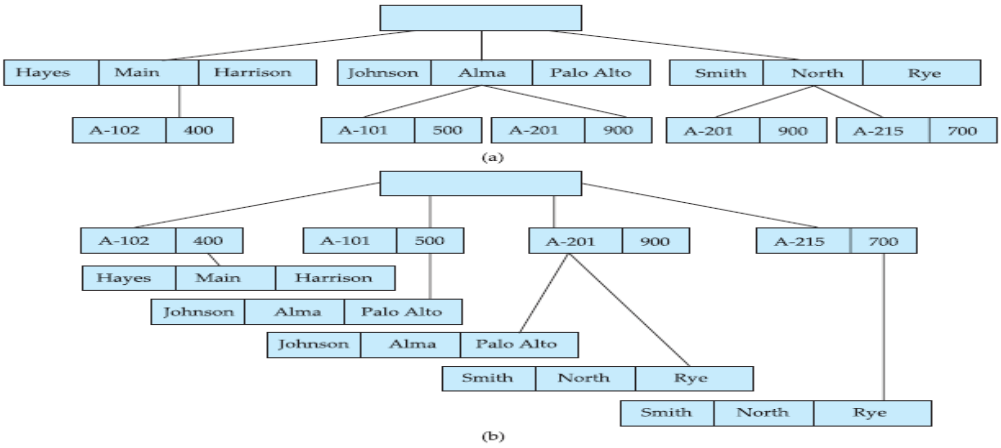


E-R diagram

Tree structure diagram for this E-R diagram is the following:-

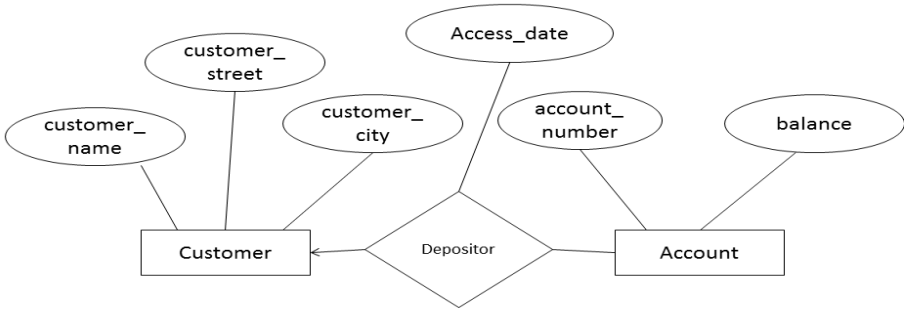


A sample database corresponding to the above tree-structure diagram is shown in the following figure:-



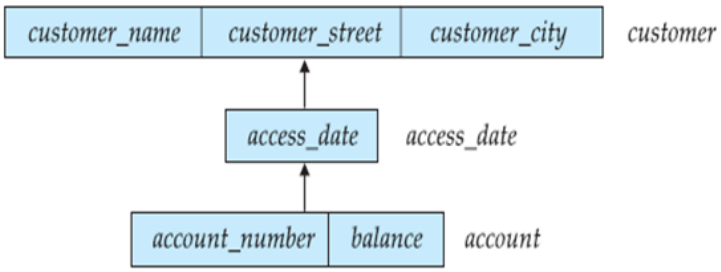
There are two database trees. The first tree (Figure a) corresponds to the tree-structure diagram T1; the second tree (Figure b) corresponds to the tree-structure diagram T2.

Example: Consider the following E-R diagram:-

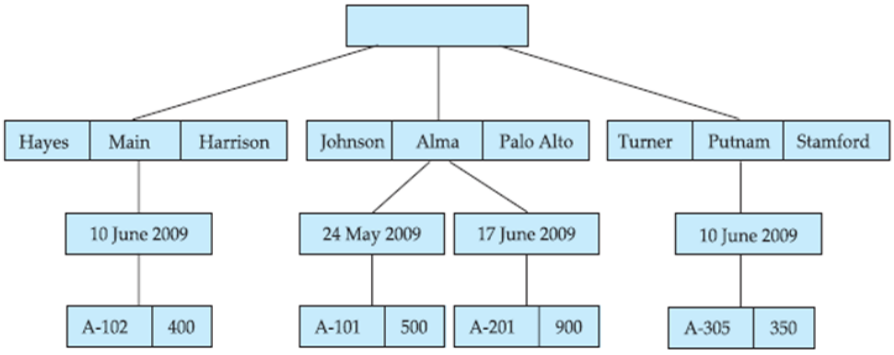


E-R diagram

Tree structure diagram for this E-R diagram is the following:-



A sample database corresponding to the above tree-structure diagram is shown in the following figure:-



1.11 Database languages

A database system provides a data definition language to specify the database schema and a data manipulation language to express database queries and updates. In practice, the data definition and data manipulation languages are not two separate languages; instead they simply form parts of a single database language, such as the widely used SQL language.

1.11.1 Data-Definition Language

We specify a database schema by a set of definitions expressed by a special language called a data-definition language (DDL).
It is a Set of SQL Commands used to create, modify, and delete database objects such as tables, views, indices etc. . It is normally used by DBA and Database engineers. It provides command like -

Command	Description
CREATE	to create objects in a database
ALTER	to modify existing data in a table
DROP	to delete objects from the database
TRUNCATE	to remove all records from the table

These are all the commands of Data Definition Language. These all Commands are shown in detail with their syntax and example:

1. CREATE :

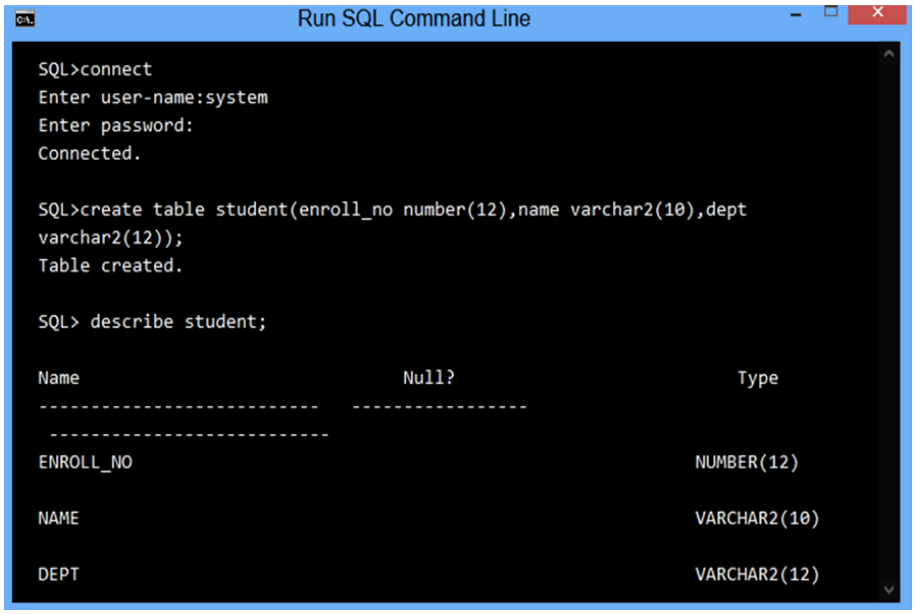
- The create command is used to create a table.
- A Table name should be unique, i.e., it must not match with existing tables.
- A Table name and column name must start with alphabet, must not match with reserved keywords, and should be combination of A-Z, a-z, 0-9, and '_' (underscore) having maximum length up to 30 characters.
- Each column definition requires name, data type and size for that column.
- Table name and column name are not case sensitive generally . But if they are enclosed within double quotes, then they are case sensitive.
- Each column definition is separated from other by a ',' (comma).
- The entire SQL statement is terminated with ';' (semi colon)

Syntax:
CREATE TABLE tablename(columnName1 datatype(size),columnName2 datatype(size),.....,
columnNameN datatype(size));

Example: Table creation is shown in the following figure:-

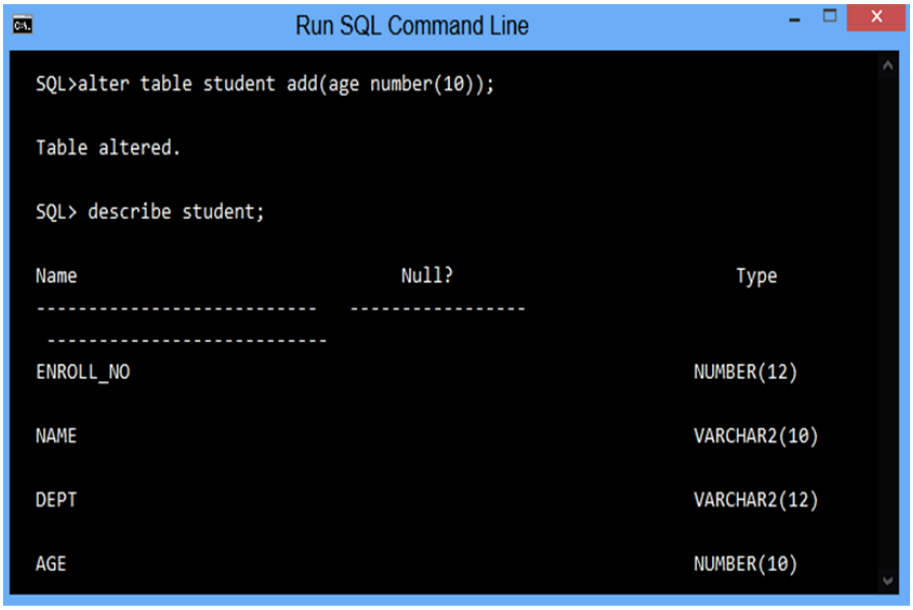
2. ALTER :

- Alter command used to modify structures of a table.
- Alter command can be used to add ,modify, or drop columns in a table.
- Alter command can be used for this purpose are described below:



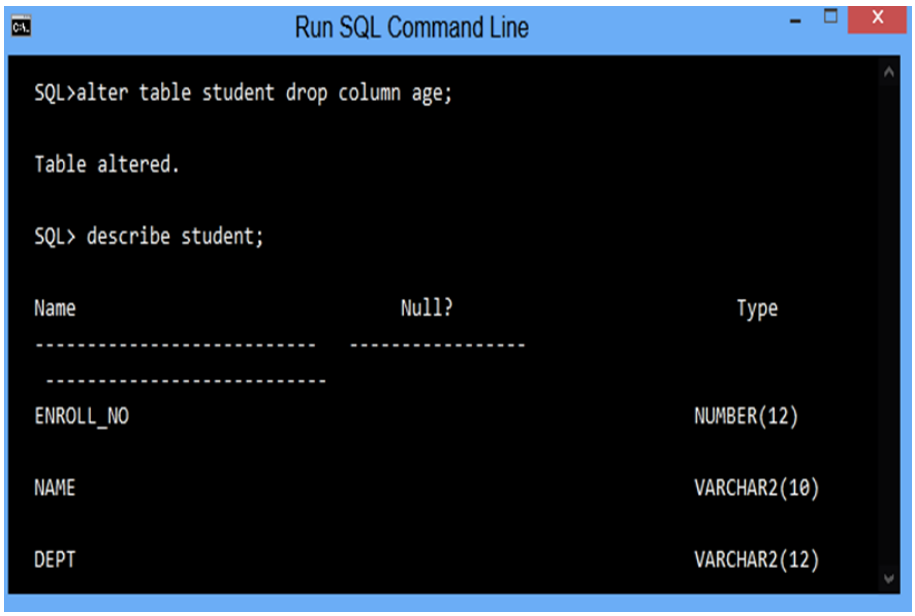
A. Adding New Columns :
This command adds a new columns in an existing table. Initially, this column will not contain any data. If required, data can be filled for this column using UPDATE command.
Syntax:

Alter table TableName Add (NewColumnName Datatype(size), NewColumnName Datatype(size)....);
Example: Addition of new column is shown in the following figure:-

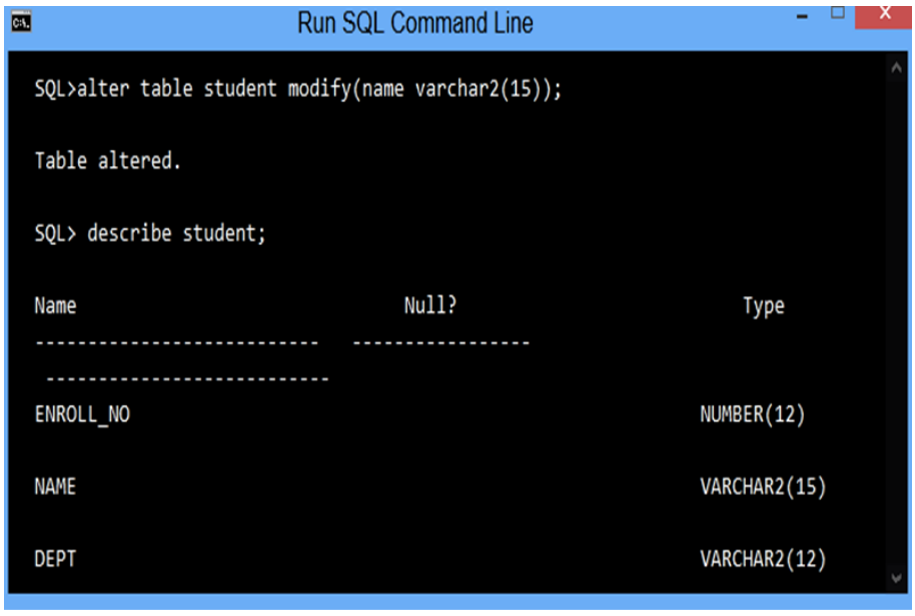


B. Dropping Columns :
This command deletes an existing column from the table along with the data held by that column.

Syntax:
Alter table TableName Drop column columnName;
Example: Deletion of a column from a table is shown in the following figure:-



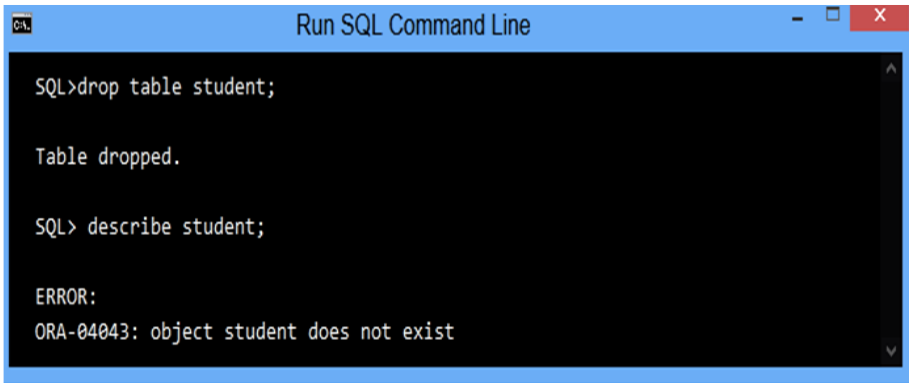
C. Modifying Columns :
This command sets newDatatype and newSize as datatype and size for specified column respectively. The main aim of this command is to modify or change the datatype and size of the column.
Syntax :
Alter table TableName Modify(columnName newDatatype(newSize));
Example:



3. DROP :

- DROP TABLE command is used to delete or destroy table from a database.
- The DROP TABLE command drops the specified table. This means, all records along with structure of the table will be destroyed.
- Care must be taken while using this command, as all records held within the table are lost and cannot be recovered.

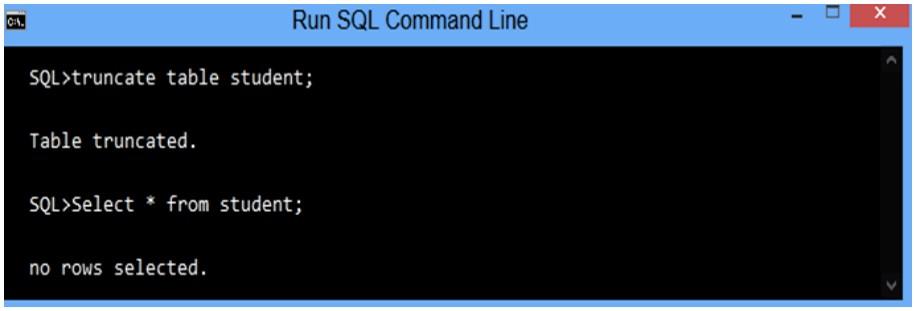
Syntax :
drop table tablename;
Example:



4. TRUNCATE :

- TRUNCATE TABLE is used to delete all data from a table.
- Logically, this is equivalent to DELETE statement that deletes all rows without using WHERE clause.
- TRUNCATE operation drops and re-creates the table. This is must faster than deleting all rows one by one.
- The deleted records cannot be recovered in truncate operation. while in delete operation, deleted records can be recovered using ROLLBACK statement.

Syntax :
truncate table tablename;
Example:



1.11.2 Data Manipulation Language

Data manipulation is

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

A data manipulation language (DML) is a language that enables users to access or manipulate data as organized by the appropriate data model. There are basically two types:

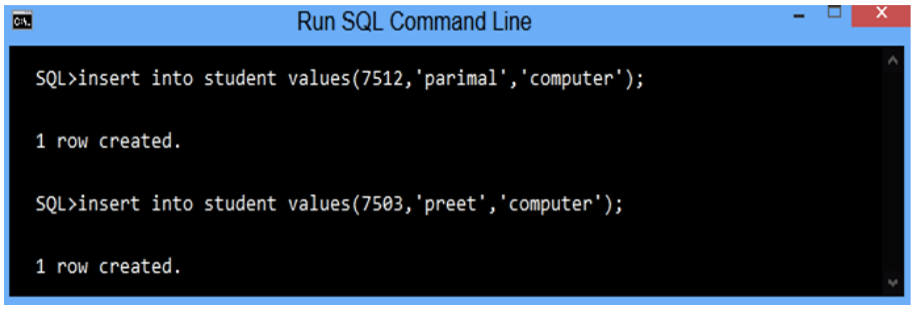
- Procedural DMLs require a user to specify what data are needed and how to get those data.
- Declarative DMLs (also referred to as non-procedural DMLs) require a user to specify what data are needed without specifying how to get those data.

A query is a statement requesting the retrieval of information. The portion of a DML that involves information retrieval is called a query language.
Following commands in SQL are used to manipulate database.
These all Commands are shown in detail with their syntax and example:

Command	Description
INSERT	insert data into table
SELECT	retrieve data from the table
UPDATE	modify existing data in the table
DELETE	delete records from the table

1. INSERT :

- The INSERT command is used insert the data into a table or create a new row in table.
- To insert user data into tables, "INSERT INTO ..." SQL statement is used. and stores the inserted values into respective columns.

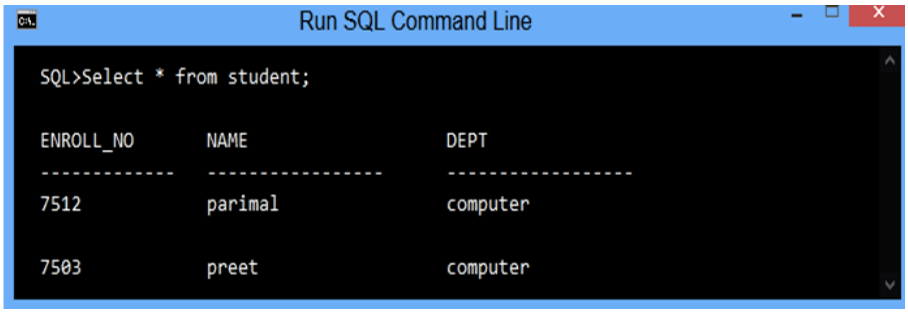


Syntax :
insert into tablename (column1, column2, columnN) Values (expression1, expression2, ... , expressionN); Example:
Example:

2. SELECT :

- The SELECT command is used to retrieve selected rows from one or more tables and displays on the screen. It is most widely used and required statement among all others in SQL.
- Once a table is loaded with user data, these data can be retrieved in number of different manners.
- Here, an asterisk (' * ') is used as the meta character, and it indicates all the columns of a given table.
- This statement retrieves all the columns and all the rows of the table.

Syntax :
select * from tablename;
Example:



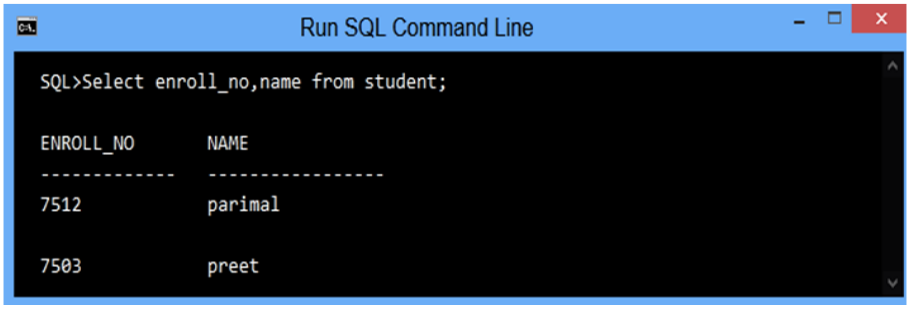
There are three ways of table data filtering as given below:
(A) Selected columns, All Rows
(B) Selected Rows, All Columns
(C) Selected columns, Selected Rows

Variations of the basic SELECT statement can be used to retrieve selected data as described below:

A. SELECTED COLUMNS, ALL ROWS

- This Statement retrieves only selected columns as specified with SELECT clause.
- This Statement retrieves all the row of the table.

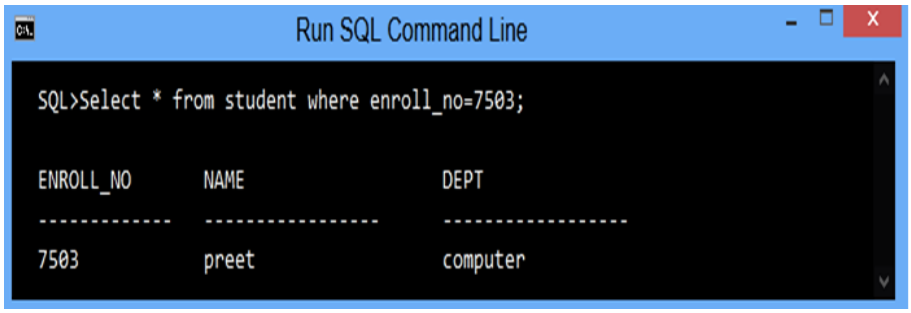
Syntax :
select column1, column2, ..., columnN from tablename;
Example:



B. SELECTED ROWS, ALL COLUMNS

- This Statement retrieves all the columns of the table.
- This statement retrieves only specific rows that specify the condition given with WHERE clause.
- Multiple conditions can be combined with logical operators such as AND and OR.

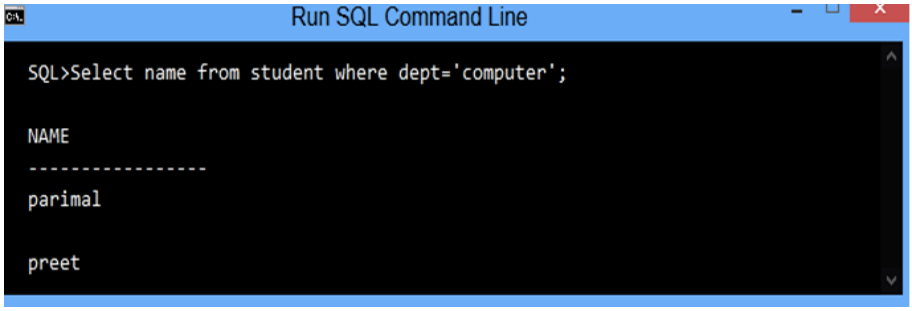
Syntax :
select * from tablename WHERE condition;
Example:



C. SELECTED ROWS, SELECTED COLUMNS

- This Statement retrieves only Selected columns as specified with SELECT clause.
- Also, retrieves only specific rows that specify the condition given with WHERE clause.

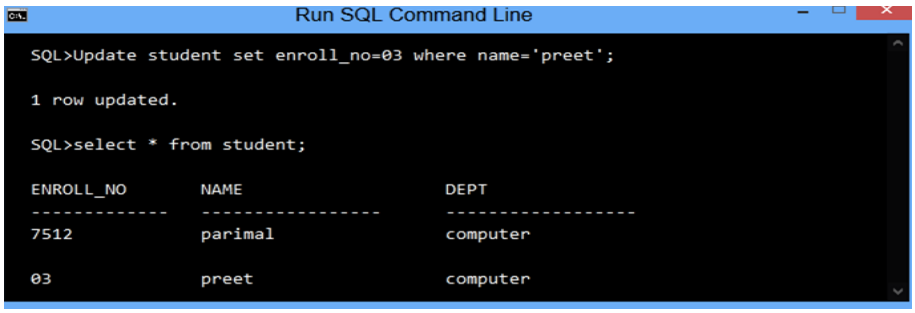
Syntax :
select column1, column2, ..., columnN from tablename WHERE condition;
Example:



3. UPDATE :

- The UPDATE command can be used to change or modify the data values in a table.
- It can be used to update either all rows or a set of rows from a table.
- This update command updates all rows from the table, and displays message regarding how many rows have been updated.
- The SET clause specifies which column data to modify.
- An expression can be a constant value, a variable, or some expression and it specifies the new value for related columns.
- You can update specific rows by the WHERE clause, and displays message regarding how many rows have been updated.

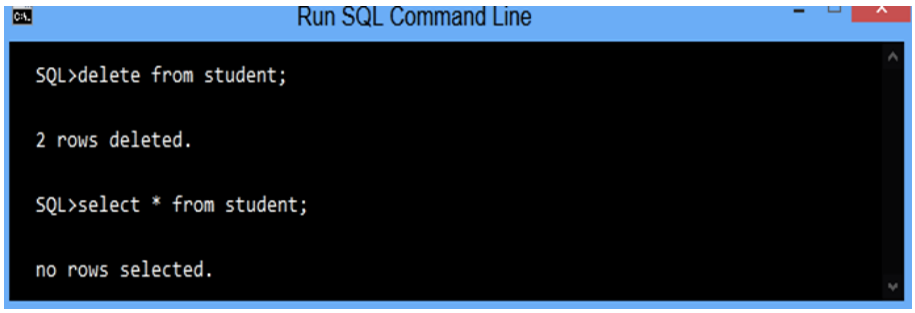
Syntax :
update tablename set column1=expression1,column2=expression2 where condition;
Example:



4. DELETE :

- The DELETE command can be used to remove either all rows of a table, or a set of rows from a table.
- The DELETE command deletes all rows from the table, and displays message regarding how many rows have been deleted.
- The DELETE command deletes rows from the table that satisfy the condition provided by WHERE clause. It also displays message regarding how many rows have been deleted.

Syntax :
delete from tablename;
Example:



1.11.3 Data Control Language

Data Control Language(DCL) is used to control privileges in Database. To perform any operation in the database, such as for creating tables or views, a user needs privileges.
In DCL, we have following two commands:
GRANT: It is used to provide any user access privileges or other priviliges for the database.
REVOKE: It is used to take back permissions from any user.

1.11.4 Transaction Control Language

Transaction Control Language(TCL) commands are used to manage transactions in the database. These are used to manage the changes made to the data in a table by DML statements. It also allows statements to be grouped together into logical transactions.
In TCL, we have following three command-

COMMIT Command: COMMIT command is used to permanently save any transaction into the database.
When we use any DML command like INSERT, UPDATE or DELETE, the changes made

by these commands are not permanent, until the current session is closed. The changes made by these commands can be rolled back.
To avoid that, we use the COMMIT command to mark the changes as permanent.
Following is commit command's syntax,

COMMIT;

ROLLBACK command
This command restores the database to last commited state. It is also used with SAVEPOINT command to jump to a savepoint in an ongoing transaction. If we have used the UPDATE command to make some changes into the database, and realise that those changes were not required, then we can use the ROLLBACK command to rollback those changes, if they were not commited using the COMMIT command.
Following is rollback command's syntax,

ROLLBACK TO savepoint_name;

SAVEPOINT command
SAVEPOINT command is used to temporarily save a transaction so that you can rollback to that point whenever required.
Following is savepoint command's syntax,

SAVEPOINT savepoint_name;

Example:

INSERT INTO class VALUES(5, 'Rahul');
COMMIT;
UPDATE class SET name = 'Abhijit' WHERE id = '5';
SAVEPOINT A;
INSERT INTO class VALUES(6, 'Chris');
SAVEPOINT B;
INSERT INTO class VALUES(7, 'Bravo');
SAVEPOINT C;
SELECT * FROM class;

1.12 Database Users and Administrators

People who work with a database can be categorized as database users or database administrators.

1.12.1 Database Users

There are four different types of database-system users, differentiated by the way they expect to interact with the system. Different types of user interfaces have been designed for the different types of users.

Naive users/Parametric users
Naive users are unsophisticated users who interact with the system by invoking one of the application programs that have been written previously. For example, a bank teller who needs to transfer $50 from account A to account B invokes a program called transfer.

Application programmers
Application programmers are computer professionals who write application programs. Application programmers can choose from many tools to develop user interfaces. Rapid application development (RAD) tools are tools that enable an application programmer to construct forms and reports without writing a program.

Sophisticated users
Sophisticated users interact with the system without writing programs. Instead, they form their requests in a database query language. They submit each such query to a query processor, whose function is to break down DML statements into instructions that the storage manager understands. Analysts who submit queries to explore data in the database fall in this category. They use some of the tools like Online Analytical Processing(OLAP), Data mining.

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

Casual end users
These are the users who occasionally access the database but they require different information each time. They use a sophisticated database query language basically to specify their request and are typically middle or level managers or other occasional browsers. For example: High level Managers who access the data weekly or biweekly.

1.12.2 Database Administrator

A person who has such central control of both the data and the programs that access those data over the system is called a database administrator (DBA). The functions of a DBA are the followings:-

- **Schema definition:** The DBA creates the original database schema by executing a set of data definition statements in the DDL.
- **Storage structure and access-method definition:** DBA decides what structure to be used to store the data and what method to be used to access that.
- **Schema and physical-organization modification:** The DBA carries out changes to the schema and physical organization to reflect the changing needs of the organization, or to alter the physical organization to improve performance.

- **Granting of authorization for data access:** By granting different types of authorization, the database administrator can regulate which parts of the database various users can access.

- **Routine maintenance:** Examples of the database administrator's routine maintenance activities are:

  - Periodically backing up the database, either onto tapes or onto remote servers, to prevent loss of data in case of disasters such as flooding.
  - Ensuring that enough free disk space is available for normal operations, and upgrading disk space as required.
  - Monitoring jobs running on the database and ensuring that performance is not degraded by very expensive tasks submitted by some users.

1.13 Database System Structure

A database system is partitioned into modules that deal with each of the responsibilities of the overall system. The functional components of a database system can be broadly divided into the storage manager and the query processor components.

1.13.1 Storage Manager

A storage manager is a program module that provides the 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 storing, retrieving, and updating data in the database. The components of storage manager includes the following modules:

Authorization and integrity manager: It checks the integrity constraints and the authority of users to access data.

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

File manager: It manages the allocation of space on disk storage and the data structures used to represent information stored on disk.

Buffer manager: It is responsible for fetching data from disk storage into main memory, and deciding what data to cache in main memory. The buffer manager is a critical part of the database system, since it enables the database to handle data sizes that are much larger than the size of main memory.

The storage manager implements several data structures as part of the physical system implementation:

- **Data files,** which store the database itself.

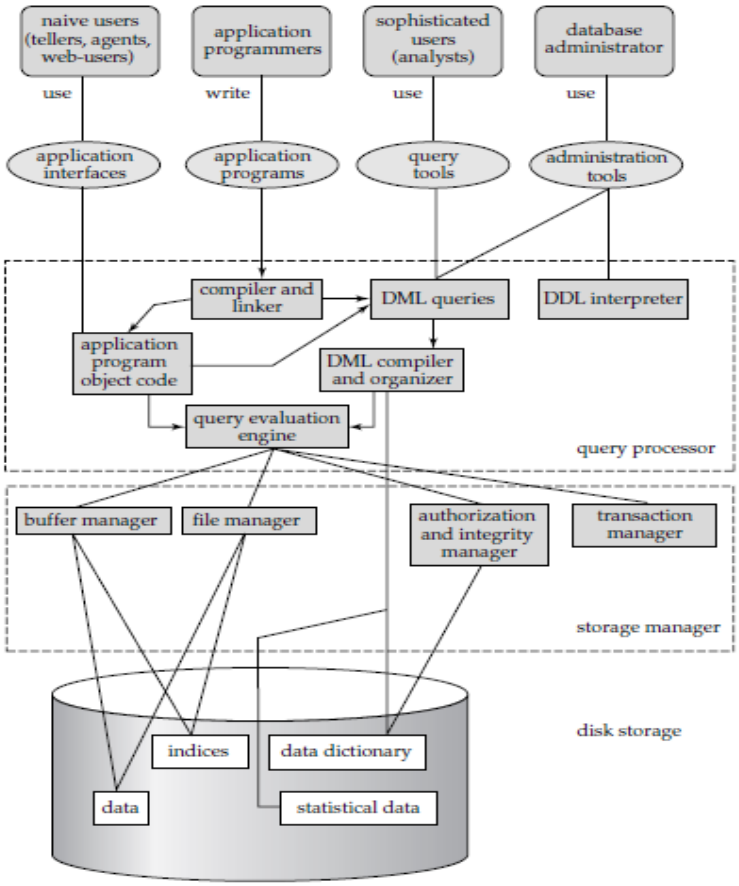


Figure 1.3: Database system structure

- **Data dictionary,** which stores metadata about the structure of the database, in particular the schema of the database.

- **Indices,** which provide fast access to data items that hold particular values.

1.13.2 Query Processor

The query processor includes the following components:-

DDL interpreter: It interprets DDL statements and records the definitions in the data dictionary.

DML compiler: It translates DML statements in a query language into an evaluation plan consisting of low-level instructions that the query evaluation engine understands. The DML compiler also performs query optimization.

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

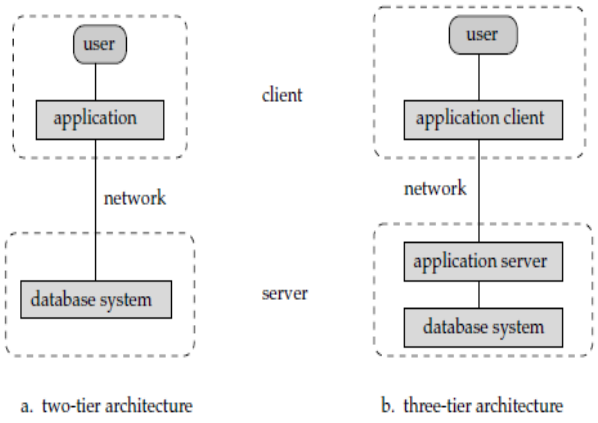


Figure 1.4: Two-tier and three-tier architectures

1.14 Application Architectures

Most users of a database system today are not present at the site of the database system, but connect to it through a network. We can therefore differentiate between client machines, on which remote database users work, and server machines, on which the database system runs. Database applications are usually partitioned into two or three parts, as in Figure .

In a **two-tier architecture,** the application is partitioned into a component that resides at the client machine, which invokes database system functionality at the server machine through query language statements. Application program interface standards like ODBC and JDBC are used for interaction between the client and the server.

In contrast, in a **three-tier architecture,** the client machine acts as merely a front end and does not contain any direct database calls. Instead, the client end communicates with an application server, usually through a forms interface. The application server in turn communicates with a database system to access data. The business logic of the application, which says what actions to carry out under what conditions, is embedded in the application server, instead of being distributed across multiple clients.

Three-tier applications are more appropriate for large applications, and for applications that run on the World Wide Web.

Chapter 2

E-R Model

2.1 E-R Model

ENTITY RELATIONSHIP (ER) MODEL is a high-level conceptual data model diagram. ER modeling helps you to analyze data requirements systematically to produce a well-designed database. The Entity-Relationship model represents real-world entities and the relationship between them. It is considered a best practice to make ER model before implementing your database.
The E-R data model uses three important components: entity sets, relationship sets, and attributes.

2.1.1 Entity Sets

An entity is a "thing" or "object" in the real world that is distinguishable from all other objects. For example, each person in an enterprise is an entity.
An entity set is a set of entities of the same type that share the same properties, or attributes. The set of all persons who are customers at a given bank, for example, can be defined as the entity set customer.

2.1.2 Attributes

Attributes are descriptive properties possessed by each member of an entity set. For example, possible attributes of the customer entity set are customer-id, customer-name, customer-street, and customer-city.
In real life, there would be further attributes, such as street number, apartment number, state, postal code, and country.
Possible attributes of the loan entity set are loan-number and amount.
For each attribute, there is a set of permitted values, called the domain, or value set, of that attribute. The domain of attribute customer-name might be the set of all text strings of a certain length.
An attribute, as used in the E-R model, can be characterized by the following attribute types.

Simple and composite attributes
An attribute is said to be simple attribute if it can not be divided into parts. An attribute which can be divide into parts is said to be composite attribute.

For example, an attribute **name** and address are the composite attributes. An attribute **roll-number** is a simple attribute.

Single-valued and multi-valued attributes
The attribute for which we have a single value for each entity is called a single valued attribute and the attributes for which we have a set of values for a particular entity is called a multi-valued attribute.
For example, loan-number, age are single valued attributes whereas mobile-no, dependent-name are multi-valued attributes.

Derived attribute
If the value for the attribute is derived from the values of other related attributes, then this attribute is said to be derived attribute.
For example, Age attribute is derived from attribute date-of-birth, Experience attribute deried from date-of-joining etc. The value of the derived attribute is not stored but is computed when required.

Null values: An attributes takes null value when an entity does not have a value for it(for example middle-name) or if the value is not known.

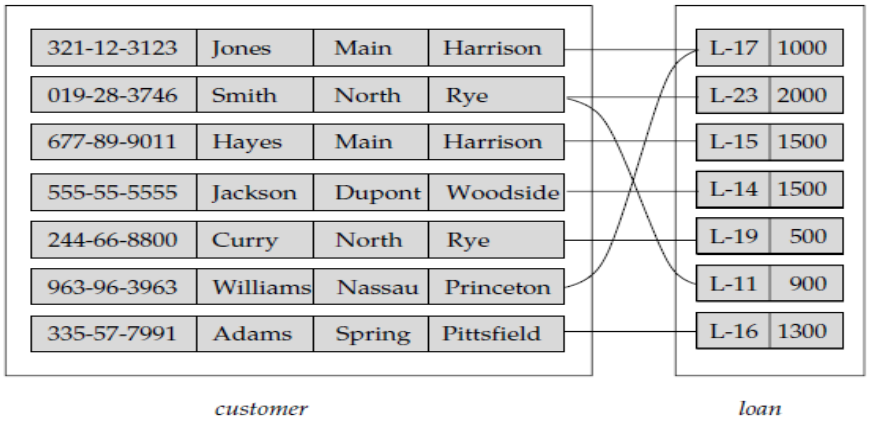
2.1.3 Relationship set

- **A relationship** is an association among several entities.
- **For example,** we can define a relationship that associates customer Hayes with loan L-15. This relationship specifies that Hayes is a customer with loan number L-15.
- A relationship set is a set of relationships of the same type.
- It is a mathematical relation on more than one entity sets. If $E_1, E_2, ..., E_n$ are entity sets, then a relationship set R is a subset of $\{(e_1, e_2, ..., e_n)!e_1 \in E_1, e_2 \in E_2, ..., e_n \in E_n\}$ where $(e_1, e_2, ..., e_n)$ is a relationship. Here, entity sets $E_1, E_2, ..., E_n$ participate in relationship R.

Example: Consider the two entity sets customer and loan in following Figure. We define the relationship set borrower to denote the association between customers and the bank loans that the customers have.

Descriptive attributes: A relationship set may have some new attributes which are not presents in the entity sets are called descriptive attributes. It describes about the relationship set.
Example: Consider a relationship set depositor with entity sets customer and account.We could associate the attribute access-date to that relationship to specify the most recent date on which a customer accessed an account.

Ternary relationship: The relationship between three entity sets is known as a



ternary relationship. For example, the relationship works-on between the entity sets employee, branch and job can be a ternary relationship.

Degree of relationship: The number of entity sets that participate in a relationship set is known as degree of relationship.

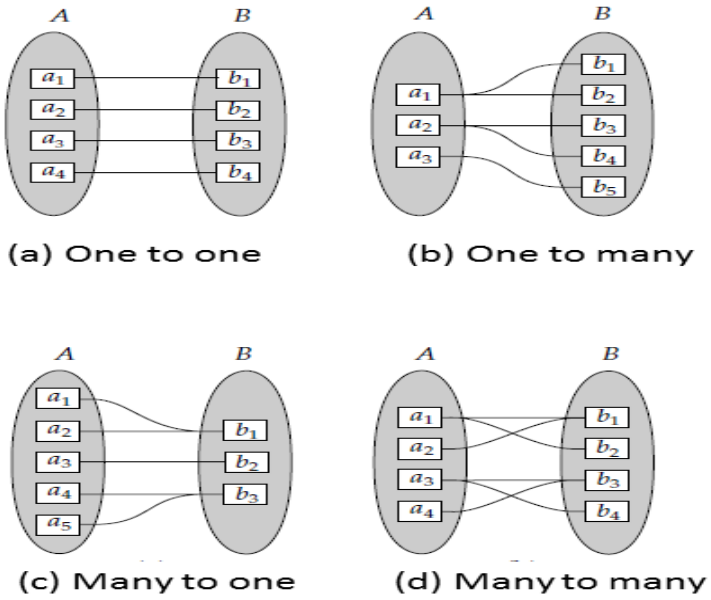
2.2 Constraints

An E-R enterprise schema may define certain constraints to which the contents of a database must conform. In this section, we examine mapping cardinalities and participation constraints, which are two of the most important types of constraints.

2.2.1 Mapping cardinality

Mapping cardinalities express the number of entities to which another entity can be associated via a relationship set.
For a binary relationship set R between entity sets A and B, the mapping cardinality must be one of the following:

- **One to one:** An entity in A is associated with at most one entity in B, and an entity in B is associated with at most one entity in A.
- **One to many:** An entity in A is associated with any number (zero or more) of entities in B. An entity in B, however, 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. An entity in B, however, can be associated with any number (zero or more) of entities in A.
- **Many to many:** An entity in A is associated with any number (zero or more) of entities in B, and an entity in B is associated with any number (zero or more) of entities in A.



2.2.2 Participation Constraints

- The participation of an entity set E in a relationship set R is said to be **total** if every entity in E participates in at least one relationship in R.
- If only some entities in E participate in relationships in R, the participation of entity set E in relationship R is said to be **partial**.

Example: Consider relationship borrower between two entity sets customer and loan. Participation of loan entity set in this relationship borrower is the total. And participation of customer entity set in this relationship borrower is the partial.

2.3 Keys

A key allows us to identify a set of attributes that suffice to distinguish entities from each other. Keys also help uniquely identify relationships, and thus distinguish relationships from each other.

2.3.1 Keys defined on entity sets

Superkey
A superkey is a set of one or more attributes that, taken collectively, allow us to identify uniquely an entity in the entity set.

Candidate key
A superkey is said to be a candidate key if no proper subset of it is a superkey.
In another words, a minimal superkeys are a candidate keys.

Primary key
Out of all possible candidate keys, the database designer chooses one as a primary key. That is, a primary key is a candidate key that is chosen by the database designer.

Alternate key
The candidate keys which are not selected as a primary key, are called alternate keys.

Composite key
Any key is said to be composite key if it consists of more than one attributes.

Example: Consider entity set student with attributes (rollNo, name, branch, address, mobileNo).
Superkeys: {rollNo}, {rollNo, name}, {rollNo, address}, {mobileNo},{name, mobileNo}.
Candidate keys: {rollNo}, {mobileNo}.
Primary key: {rollNo}
Alternate key: {mobileNo}.
Composite keys: {rollNo, name}, {rollNo,address},{name, mobileNo}.

2.3.2 Relationship keys

Let R be a relationship set involving entity sets $E_1, E_2, ..., E_n$. Let primary-key(E_i) denote the set of attributes that forms the primary key for entity set E_i.

- If the relationship set R has no attributes associated with it, then the set of attributes primary-key(E_1) $\cup$ primary-key(E2) $\cup \cdots \cup$ primary-key(En) describes an individual relationship in set R.

- If the relationship set R has attributes a1, a2, $\cdots$, am associated with it, then the set of attributes primary-key(E_1) $\cup$ primary-key(E_2) $\cup \cdots \cup$ primary-key(E_n) $\cup$ $\{a_1, a_2, ..., a_m\}$ describes an individual relationship in set R.

In both of the above cases, the set of attributes primary-key(E_1) $\cup$ primary-key(E_2) $\cup \cdots \cup$ primary-key(E_n) forms a **superkey** for the relationship set.

The structure of the **primary key** for the relationship set depends on the mapping cardinality of the relationship set.
Example: consider the entity sets customer and account, and the relationship set depositor, with attribute access-date.

- **Case 1:** If the relationship set is many to many. Then the primary key of depositor consists of the union of the primary keys of customer and account.

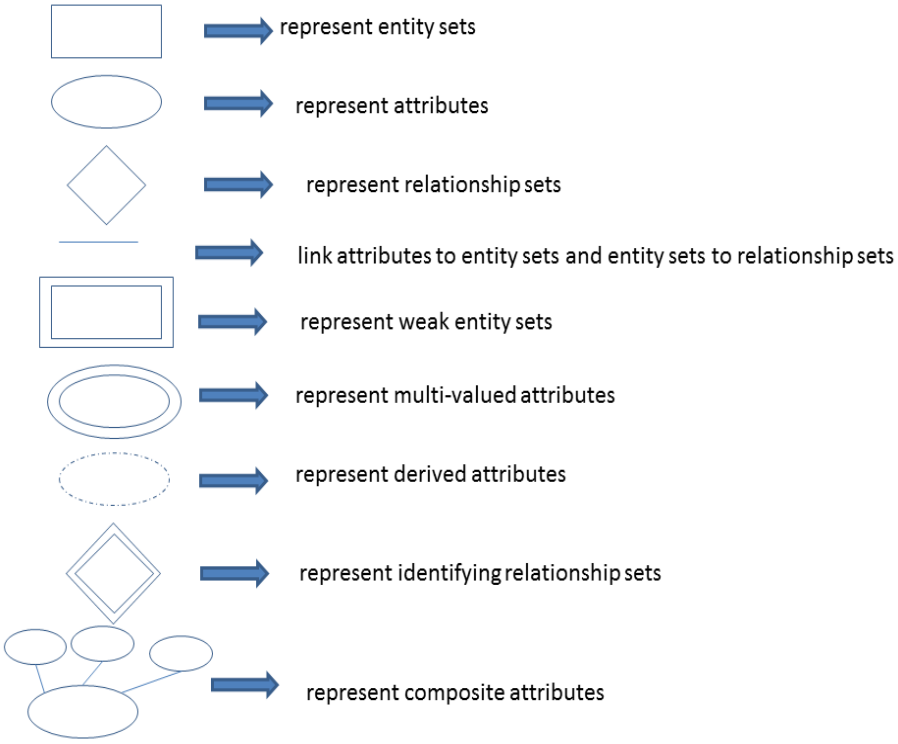
- **Case 2:** If the depositor relationship is many to one from customer to account, then the primary key of depositor will be the primary key of customer.

- **Case 3:** If the depositor relationship is many to one from account to customer, then the primary key of depositor will be the primary key of account.

- **Case 4:** If the depositor relationship is one to one from customer to account, then the primary key of depositor will be either the primary key of customer or the primary key of account.

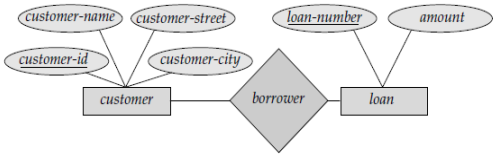
2.4 Entity-Relationship Diagram

An E-R diagram can express the overall logical structure of a database graphically. Such a diagram consists of the following major components:

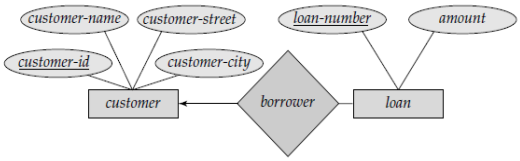


2.4.1 Some E-R diagram examples

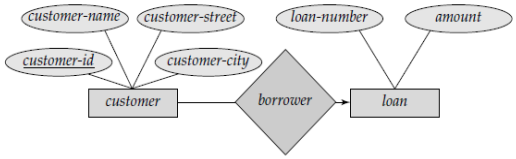
1. E-R diagram showing many to many relationship between customer and loan entity set.



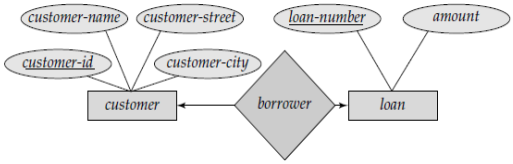
2. E-R diagram showing one to many relationship between customer and loan entity set.



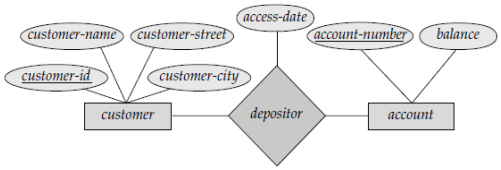
3. E-R diagram showing many to one relationship between customer and loan entity set.



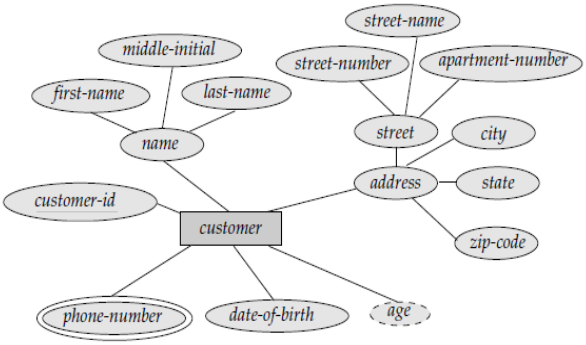
4. E-R diagram showing one to one relationship between customer and loan entity set.



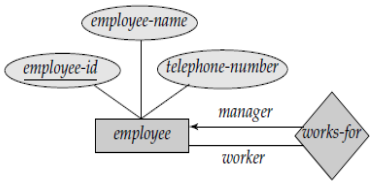
5. E-R diagram showing descriptive attributes associated with a relationship set.



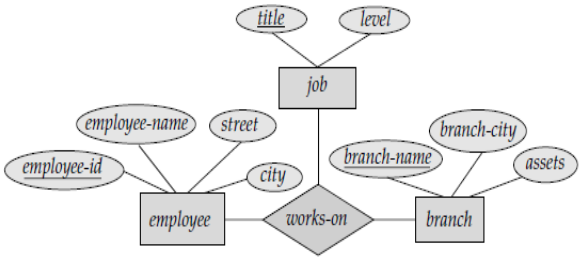
6. E-R diagram showing composite, multi-valued and derived attributes.



7. E-R diagram showing role indicator.



8. E-R diagram showing ternary relationship.

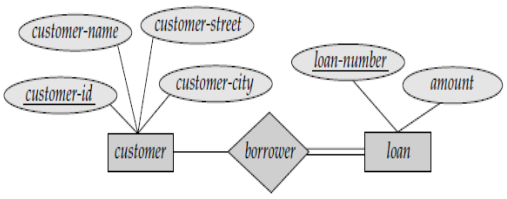


We permit at most one arrow out of a relationship set, since an E-R diagram with two or more arrows out of a non-binary relationship set can be interpreted in two ways. Suppose there is a relationship set R between entity sets $A_1, A_2, ..., A_n$, andthe only arrows are on the edges to entity sets $Ai + 1, A_{i+2}, ..., A_n$. Then, the two possible interpretations are:

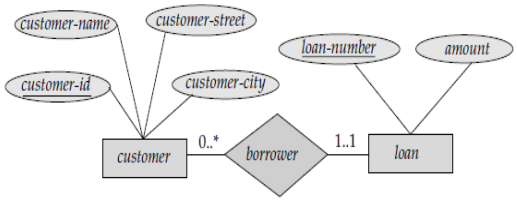
(a) A particular combination of entities from $A_1, A_2, ..., A_i$ can be associated with at most one combination of entities from $Ai + 1, A_{i+2}, ..., A_n$. Thus, the primary key for the relationship R can be constructed by the union of the primary keys of $A_1, A_2, ..., A_i$.

(b) For each entity set $A_k, i < k \leq n$, each combination of the entities from the other entity sets can be associated with at most one entity from A_k. Each set $\{A_1, A_2, ..., A_{k-1}, A_{k+1}, ..., A_n\}$, for $i < k \leq$ n, then forms a candidate key.

9. E-R diagram showing total participation.



10. E-R diagram showing alternative notation for cardinality limits.

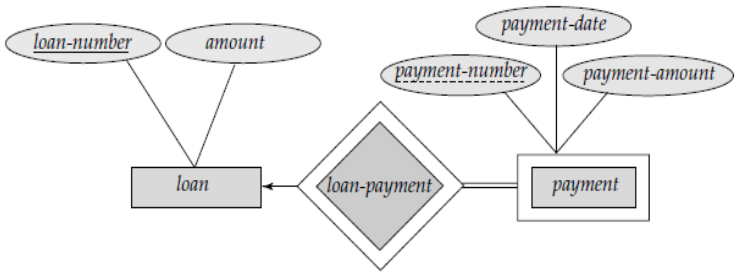


- An edge between an entity set and a binary relationship set can have an associated minimum and maximum cardinality, shown in the form l..h, where l is the minimum and h the maximum cardinality.

- A minimum value of 1 indicates total participation of the entity set in the relationship set.
- A maximum value of 1 indicates that the entity participates in at most one relationship, while a maximum value * indicates no limit.
- Note that a label 1..* on an edge is equivalent to a double line.

2.5 Weak Entity Sets

- An entity set that does not have a primary key is referred to as a **weak entity set**.
- An entity set that has a primary key is termed as a strong entity set.
- The existence of weak entity set depends on the existence of an **identifying or owner entity set**.
- The relationship associating the weak entity set with identifying entity set is called an **identifying relationship set**.
- The identifying relationship is many to one from the weak entity set to the identifying entity set, and the participation of the weak entity set in the relationship is total.
- The **discriminator** of a weak entity set is a set of attributes that distinguishes among all the entities in weak entity set that depends on one strong entity set. It is also said to be **partial key**.
- The primary key of a weak entity set is formed by the primary key of the strong entity set on which the weak entity set is dependent plus the weak entity set's discriminator.
- Weak entity set is represented by double rectangle.
- The discriminator of a weak entity set is shown with dashed lines.
- The identifying relationship set is represented by double diamonds



In this E-R diagram, payment is a weak entity set and loan is a strong entity set. loan-payment is an identifying entity set. Discriminator of payment is payment-number. Primary key of payment will be {loan-number,payment-number}.

2.6 Extended E-R Features

In this section, we discuss the extended E-R features of specialization, generalization, higher- and lower-level entity sets, attribute inheritance, and aggregation.

2.6.1 Specialization

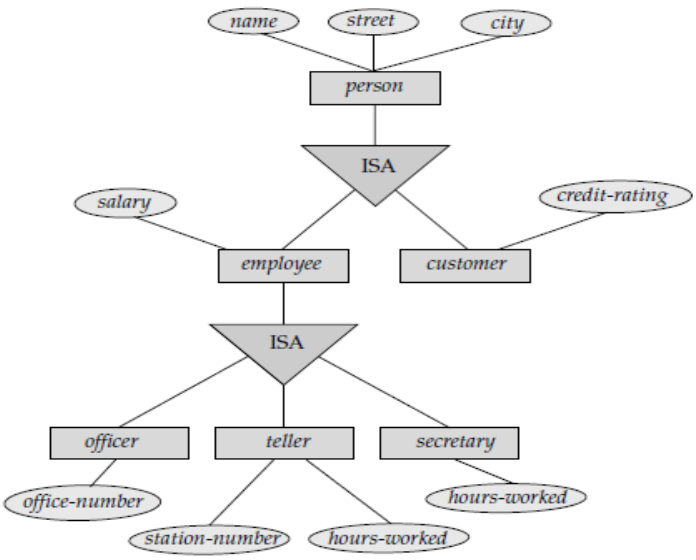
The process of designating subgroupings within an entity set is called specialization.

Example: Consider an entity set person, with attributes name, street, and city. A person may be further classified as one of the following:

- customer
- employee

Each of these person types is described by a set of attributes that includes all the attributes of entity set person plus possibly additional attributes. For example, customer entities may be described further by the attribute customer-id, whereas employee entities may be described further by the attributes employee-id and salary.

In terms of an E-R diagram, specialization is depicted by a triangle component labeled ISA, as shown in the following figure. The label ISA stands for "is a". The ISA relationship may also be referred to as a superclass-subclass relationship. Higher- and lower-level entity sets are depicted as regular entity sets that is, as rectangles containing the name of the entity set.



2.6.2 Generalization

- Generalization is a containment relationship that exists between a higher-level entity set and one or more lower-level entity sets.
- Generalization is a bottom-up design process that combines the number of entity sets that share the same features into higher level entity sets.

- Generalization and specialization are simple inversions of each other. They are represented in an E-R diagram in the same way.
- In E-R diagram, generalization is also represented by triangle symbol "ISA" just like specialization.
- Same E-R diagram is used to represent specialization and generalization.

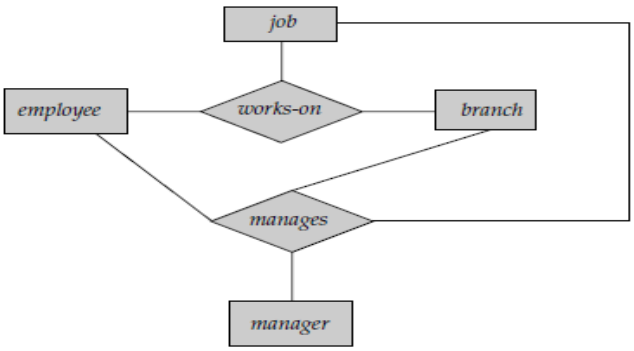
Example: In generalization process, database designer may have first identified a customer entity set with the attributes name, street, city, and customer-id, and an employee entity set with the attributes name, street, city, employee-id, and salary.

There are similarities between the customer entity set and the employee entity set in the sense that they have several attributes in common. This commonality can be expressed by generalization.

2.6.3 Aggregation

- Aggregation is an abstraction through which relationships are treated as a higher level entity sets and can participate in relationships.
- Aggregation allows us to indicate that a relationship set participates in another relationship sets.

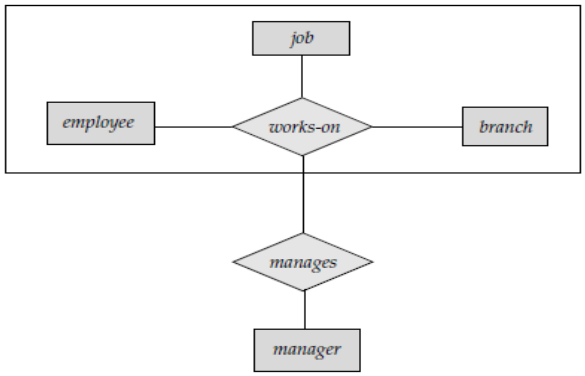
Example: Consider the following E-R diagram:-



E-R diagram with redundant relationships

There is redundant information in the above figure, however, since every employee, branch, job combination in manages is also in works-on. If the manager were a value rather than an manager entity,we could instead make manager a multi-valued attribute of the relationship works-on.

Using aggregation, the relationship set works-on (relating the entity sets employee, branch, and job) is treated as a higher-level entity set called works-on. Such an entity set is treated in the same manner as is any other entity set. We can then create a binary relationship manages between works-on and manager to represent who manages what tasks. It is shown in the following figure:-

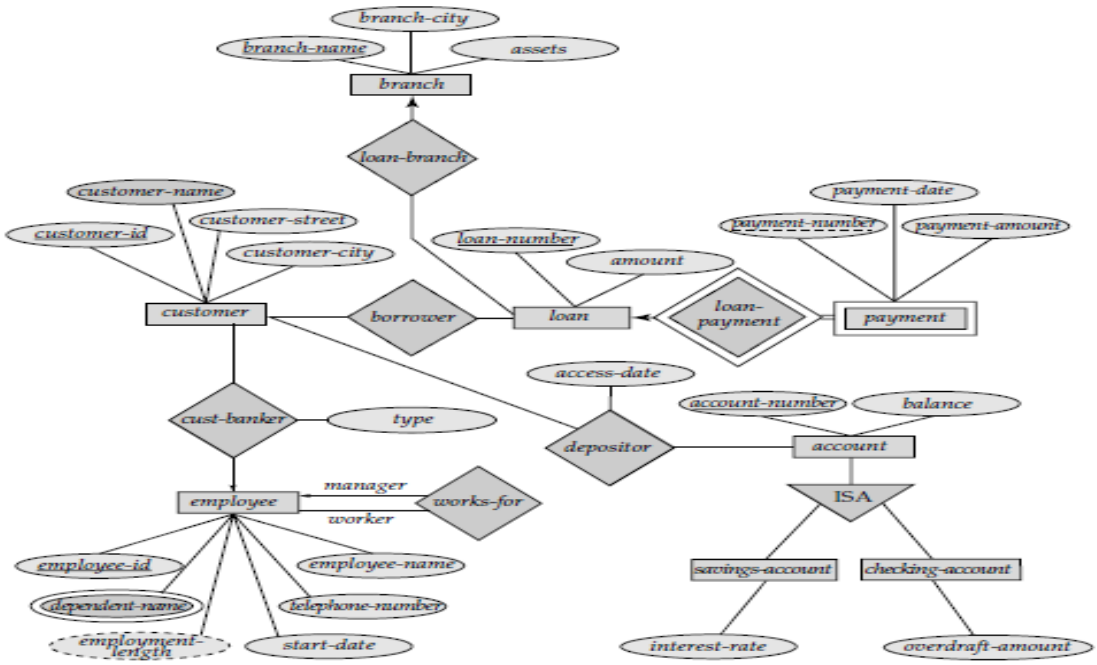


E-R diagram with aggregation

Example:
Draw the E-R model or diagram for banking enterprise.
Solution: To design E-R model corresponding to any enterprise, we follow the following steps:-

- **Data requirements**
- **Entity Sets Designation**
- **Relationship Sets Designation**
- **Identify attributes**
- **Identify mapping cardinality of each relationship**

E-R diagram for banking enterprise is the following:-



E-R diagram for banking enterprise

2.7 Reduction of an E-R Schema to Tables

Every E-R schema or diagram can be converted into set of tables. In this section, we describe how an E-R schema can be represented by tables. The constraints specified in an E-R diagram, such as primary keys and cardinality constraints, are mapped to constraints on the tables generated from the E-R diagram.

2.7.1 Tabular Representation of Strong Entity Sets

Let E be a strong entity set with descriptive attributes $a_1, a_2, ..., a_n$. We represent this entity by a table called E with n distinct columns, each of which corresponds to one of the attributes of E. Each row in this table corresponds to one entity of the entity set E.

Example:
Consider a strong entity set **loan** with attributes loan-number and amount. The table corresponding to loan will be the following

loan-number	**amount**
L-1	10000
L-2	15000
L-3	20000

Table 2.1: loan table

2.7.2 Tabular Representation of Weak Entity Sets

Let A be a weak entity set with attributes $a_1, a_2, ..., a_m$. Let B be the strong entity set on which A depends. Let the primary key of B consist of attributes $b_1, b_2, ..., b_n$. We represent the entity set A by a table called A with one column for each attribute of the set:

$$\{a_1, a_2, ..., a_m\} \cup \{b_1, b_2, ..., b_n\}$$

Example:
Consider the weak entity set payment with three attributes: payment-number, payment-date, and payment-amount. It depends on strong entity set loan. The primary key of loan is loan-number. Thus, we represent payment entity set by a table with four columns labeled loan-number, payment-number, payment-date, and payment-amount. The table corresponding to payment will be the following

loan-number	**payment-number**	**payment-date**	**payment-amount**
L-1	1	9 May 20018	5000
L-2	3	29 May 20018	6000
L-2	4	20 June 20018	9000

Table 2.2: payment table

2.7.3 Tabular Representation of Relationship Sets

Let R be a relationship set, let $a_1, a_2, ..., a_m$ be the set of attributes formed by the union of the primary keys of each of the entity sets participating in R, and let the descriptive attributes (if any) of R be $b_1, b_2, ..., b_n$. We represent this relationship set by a table called R with one column for each attribute of the set:

$\{a_1, a_2, ..., a_m\} \cup \{b_1, b_2, ..., b_n\}$

Example:
Consider the relationship set borrower. This relationship set involves the following two entity sets:

- customer, with the primary key customer-id
- loan, with the primary key loan-number

Assume relationship set borrower has no attributes. Then, the borrower table has two columns, labeled customer-id and loan-number. Borrower table is the following:-

customer-id	loan-number
C-34	L-1
C-45	L-2
C-20	L-3

Table 2.3: borrower table

2.7.4 Tabular Representation corresponding to Composite Attributes

We handle composite attributes by creating a separate attribute for each of the component attributes; we do not create a separate column for the composite attribute itself.

Example:
Suppose address is a composite attribute of entity set customer, and the components of address are street and city. The table generated from customer would then contain columns address-street and address-city; there is no separate column for address.

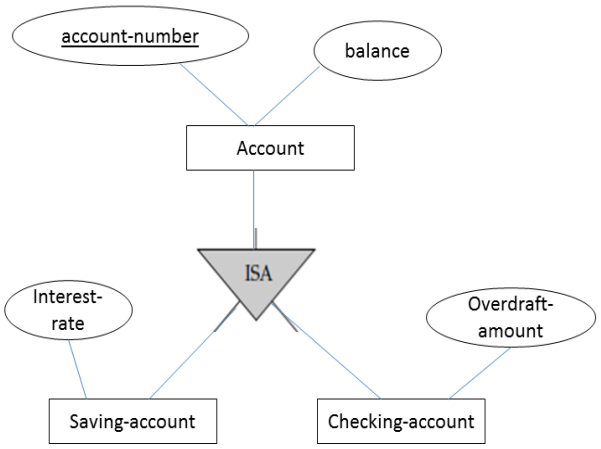
2.7.5 Tabular Representation corresponding to Multivalued Attributes

For a multivalued attribute M, we create a table T with a column C that corresponds to M and columns corresponding to the primary key of the entity set or relationship set of which M is an attribute.

Example:
Consider the above E-R diagram for banking enterprise. The diagram includes the multivalued attribute dependent-name. For this multivalued attribute, we create a table dependent-name, with columns dname, referring to the dependent-name attribute of employee, and employee-id, representing the primary key of the entity set employee. Each dependent of an employee is represented as a unique row in the table.

2.7.6 Tabular Representation of Generalization

There are two different methods for transforming to a tabular form an E-R diagram that includes generalization. Consider the following E-R diagram:-



We simplify it by including only the first tier of lower-level entity sets, that is, savings-account and checking-account.

1. Create a table for the higher-level entity set. For each lower-level entity set, create a table that includes a column for each of the attributes of that entity set plus a column for each attribute of the primary key of the higher-level entity set.
   For the above E-R diagram, we have three tables:

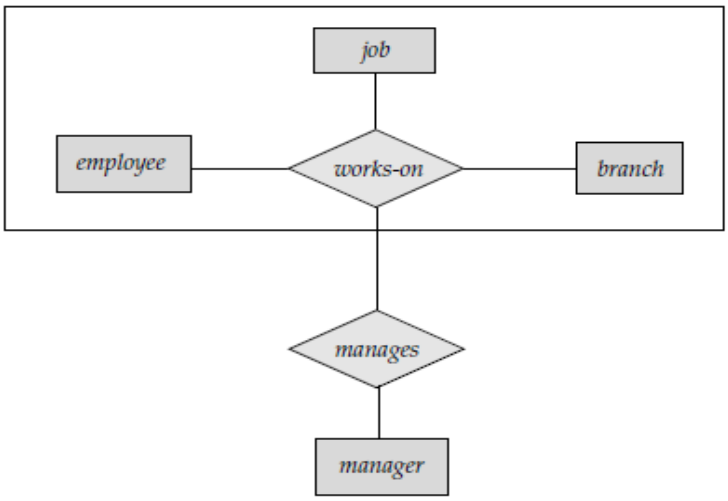
   - account, with attributes account-number and balance
   - savings-account, with attributes account-number and interest-rate
   - checking-account, with attributes account-number and overdraft-amount

2. If the generalization is disjoint and complete, that is, if no entity is a member of two lower-level entity sets directly below a higher-level entity set, and if every entity in the higher level entity set is also a member of one of the lower-level entity sets. Here, do not create a table for the higher-level entity set. Instead, for each lower-level entity set, create a table that includes a column for each of the attributes of that entity set plus a column for each attribute of the higher-level entity set.
   For the above E-R diagram, we have two tables:

   - savings-account, with attributes account-number, balance, and interest-rate
   - checking-account, with attributes account-number, balance, and overdraft-amount

2.7.7 Tabular Representation of Aggregation

Transforming an E-R diagram containing aggregation to a tabular form is straightforward. Consider the diagram in the following figure:-



E-R diagram with aggregation

The table for the relationship set manages between the aggregation of works-on and the entity set manager includes a column for each attribute in the primary keys of the entity set manager and the relationship set works-on. It would also include a column for any descriptive attributes, if they exist, of the relationship set manages. We then transform the relationship sets and entity sets within the aggregated entity.

2.8 Exercise

1. Construct an E-R diagram for a car-insurance company whose customers own one or more cars each. Each car has associated with it zero to any number of recorded accidents.

2. Construct an E-R diagram for a hospital with a set of patients and a set of medical doctors. Associate with each patient a log of the various tests and examinations conducted.

3. A university registrar's office maintains data about the following entities:
   (a) courses, including number, title, credits, syllabus, and prerequisites;
   (b) course offerings, including course number, year, semester, section number, instructor(s), timings, and classroom;
   (c) students, including student-id, name, and program; and
   (d) instructors, including identification number, name, department, and title.

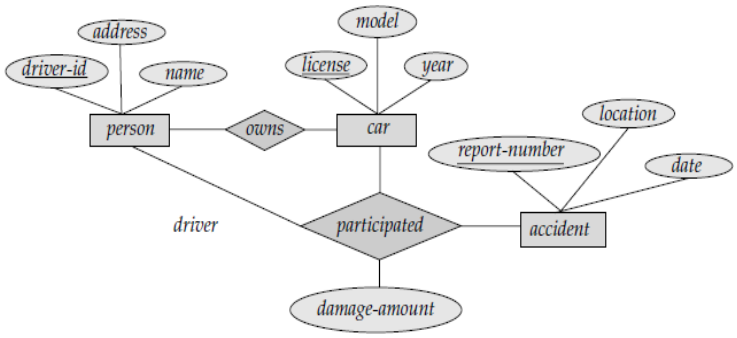
   Further, the enrollment of students in courses and grades awarded to students in each course they are enrolled for must be appropriately modeled.
   Construct an E-R diagram for the registrar's office. Document all assumptions that you make about the mapping constraints.

4. Construct appropriate tables for each of the E-R diagrams in Exercises 1 to 3.

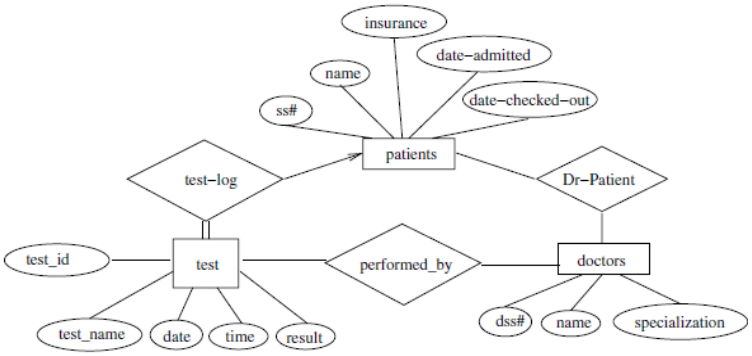
5.

2.8.1 Solution of Exercise

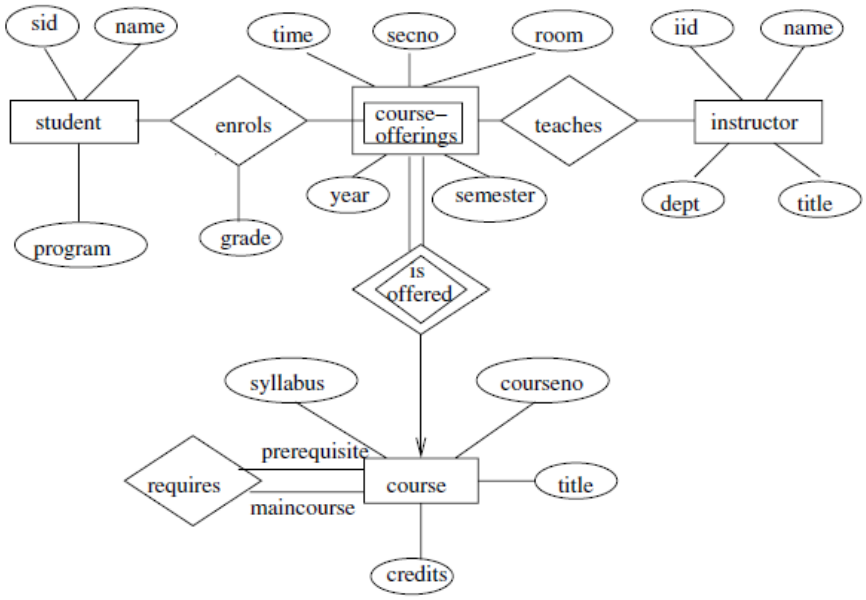
1. An E-R diagram for a car-insurance company is the following:-



2. An E-R diagram for a hospital with a set of patients and a set of medical doctors is the following:-



3. An E-R diagram for the registrar's office is the following:-



The assumptions made are :

(a) A class meets only at one particular place and time. This E-R diagram cannot model a class meeting at different places at different times.

(b) There is no guarantee that the database does not have two classes meeting at the same place and time.

4. Appropriate tables for each of the E-R diagrams used in exercises 1 to 3 are the following:-

(1) Car insurance tables:
person (driver-id, name, address)
car (license, year, model)
accident (report-number, date, location)
participated(driver-id, license, report-number, damage-amount)

(2) Hospital tables:
patients (patient-id, name, insurance, date-admitted, date-checked-out)
doctors (doctor-id, name, specialization)
test (testid, testname, date, time, result)
doctor-patient (patient-id, doctor-id)
test-log (testid, patient-id)
performed-by (testid, doctor-id)

(3) University registrar's tables:
student (student-id, name, program)
course (courseno, title, syllabus, credits)
course-offering (courseno, secno, year, semester, time, room)
instructor (instructor-id, name, dept, title)
enrols (student-id, courseno, secno, semester, year, grade)
teaches (courseno, secno, semester, year, instructor-id)
requires (maincourse, prerequisite)

Chapter 3

Relational Data Model

Relational data model represents the database as a collection of relations. A relation is nothing but a table of values. Every row in the table represents a collection of related data values. These rows in the table denote a real-world entity or relationship.

3.1 Some concepts related with relational data model

Relation: A relation is a table with columns and rows.

Field: A column in a table is called the field of a relation.

Tuple: It is nothing but a single row of a table, which contains a single record.

Relation schema: A relation schema represents the name of the relation with its attributes. If $A_1, A_2, ..., A_n$ are attributes then R= $(A_1, A_2, ..., A_n)$ is a relation schema.
Example: A relation schema student with their attributes are like the followings:-
student=(rollNo, name,branch,contactNo).

Relation Instance: Relation instance is a finite set of tuples in the RDBMS system. Relation instances never have duplicate tuples.

Attribute domain: The set of all possible values of a relation is said to be domain of an attribute.

Degree: The total number of attributes exist in the relation is called the degree of the relation.

Cardinality: Total number of rows present in the table.

Atomic values: A value is said to be atomic if it is not divisible.

Note: Domain of an attribute is said to be atomic if all its possible values are atomic i.e. not divisible.

3.2 Integrity Constraints

- Integrity constraints are a set of rules. It is used to maintain the quality of information.
- Integrity constraints ensure that the data insertion, updating, and other processes have to be performed in such a way that data integrity is not affected.
- Thus, integrity constraint is used to guard against accidental damage to the database.

3.2.1 Types of Integrity Constraint

1. Domain Constraints
2. Entity Integrity Constraints
3. Referential Integrity Constraints
4. Key Constraints

Domain constraints

- Domain constraints can be defined as the definition of a valid set of values for an attribute.
- The data type of domain includes string, character, integer, time, date, currency, etc. The value of the attribute must be available in the corresponding domain.

Example: Consider the following table

ID	NAME	SEMENSTER	AGE
1000	Tom	1st	17
1001	Johnson	2nd	24
1002	Leonardo	5th	21
1003	Kate	3rd	19
1004	Morgan	8th	A

Not allowed. Because AGE is an integer attribute

Entity integrity constraints

- The entity integrity constraint states that primary key value can't be null.
- This is because the primary key value is used to identify individual rows in relation and if the primary key has a null value, then we can't identify those rows.
- A table can contain a null value other than the primary key field.

Example: Consider the following table

EMPLOYEE

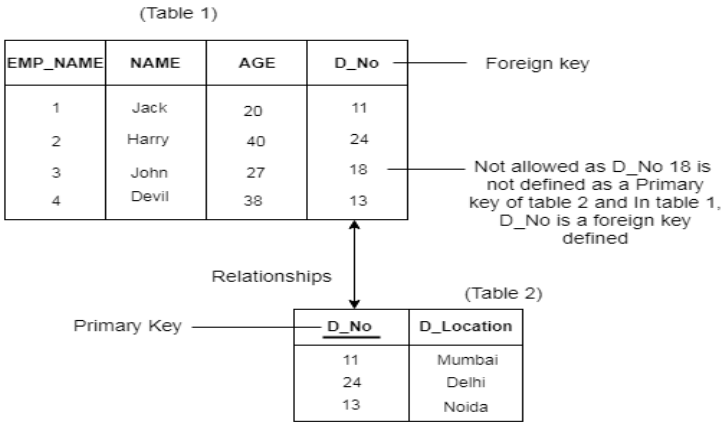
EMP_ID	EMP_NAME	SALARY
123	Jack	30000
142	Harry	60000
164	John	20000
	Jackson	27000

Not allowed as primary key can't contain a NULL value

Referential Integrity Constraints

- A referential integrity constraint is specified between two tables.
- In the Referential integrity constraints, if a foreign key in Table 1 refers to the Primary Key of Table 2, then every value of the Foreign Key in Table 1 must be null or be available in Table 2.

Example: Consider the following table



Key constraints

- Keys in the entity set is used to identify an entity within its entity set uniquely.
- An entity set can have multiple keys, but out of which one key will be the primary key. A primary key can contain a unique and null value in the relational table.

Example: Consider the following table

ID	NAME	SEMENSTER	AGE
1000	Tom	1st	17
1001	Johnson	2nd	24
1002	Leonardo	5th	21
1003	Kate	3rd	19
1002	Morgan	8th	22

Not allowed. Because all row must be unique

3.3 Foreign key

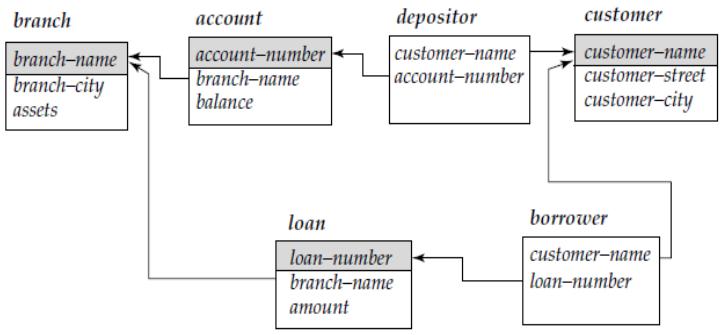
- Consider R and S are two tables. An attribute A of table R is said to be foreign key of R if A is the primary key in S.
- A foreign key is a column (or combination of columns) in a table whose values must match values of a column in some other table.
- FOREIGN KEY constraints enforce referential integrity, which essentially says that if column value A refers to column value B, then column value B must exist.

Example: Consider two tables Employee(ID, Name, Dept-ID) and Department(Dept-ID, Dept-name).
Here, attribute Dept-ID in Employee table is a foreign key because Dept-ID in Department table is a primary key.

3.4 Schema Diagram

- A database schema, along with primary key and foreign key dependencies, can be depicted pictorially by schema diagrams.
- Each relation appears as a box, with the attributes listed inside it and the relation name above it. If there are primary key attributes, a horizontal line crosses the box, with the primary key attributes listed above the line. Foreign key dependencies appear as arrows from the foreign key attributes of the referencing relation to the primary key of the referenced relation.

Example: Following figure shows the schema diagram for our banking enterprise.



3.5 Query Languages

A query language is a language in which a user requests information from the database. Query languages can be categorized as either procedural or non-procedural.

In a procedural language, the user instructs the system to perform a sequence of operations on the database to compute the desired result.

In a non-procedural language, the user describes the desired information without giving a specific procedure for obtaining that information.

In this chapter, we will study following three languages:-

1. Relational algebra
2. Tuple relational calculus
3. Domain relational calculus

In these languages, relational algebra is a procedural but tuple and domain relational calculus are non-procedural languages.

Consider the following banking database. We will write all the queries for this database.

branch

branch-name	*branch-city*	*assets*
Brighton	Brooklyn	7100000
Downtown	Brooklyn	9000000
Mianus	Horseneck	400000
North Town	Rye	3700000
Perryridge	Horseneck	1700000
Pownal	Bennington	300000
Redwood	Palo Alto	2100000
Round Hill	Horseneck	8000000

customer

customer-name	*customer-street*	*customer-city*
Adams	Spring	Pittsfield
Brooks	Senator	Brooklyn
Curry	North	Rye
Glenn	Sand Hill	Woodside
Green	Walnut	Stamford
Hayes	Main	Harrison
Johnson	Alma	Palo Alto
Jones	Main	Harrison
Lindsay	Park	Pittsfield
Smith	North	Rye
Turner	Putnam	Stamford
Williams	Nassau	Princeton

account

account-number	*branch-name*	*balance*
A-101	Downtown	500
A-102	Perryridge	400
A-201	Brighton	900
A-215	Mianus	700
A-217	Brighton	750
A-222	Redwood	700
A-305	Round Hill	350

depositor

customer-name	*account-number*
Hayes	A-102
Johnson	A-101
Johnson	A-201
Jones	A-217
Lindsay	A-222
Smith	A-215
Turner	A-305

loan

loan-number	*branch-name*	*amount*
L-11	Round Hill	900
L-14	Downtown	1500
L-15	Perryridge	1500
L-16	Perryridge	1300
L-17	Downtown	1000
L-23	Redwood	2000
L-93	Mianus	500

borrower

customer-name	*loan-number*
Adams	L-16
Curry	L-93
Hayes	L-15
Jackson	L-14
Jones	L-17
Smith	L-11
Smith	L-23
Williams	L-17

Figure 3.1: Banking database

3.6 Relational Algebra

- The relational algebra is a procedural query language.
- It consists of a set of operations that take one or two relations as input and produce a new relation as their result.
- The fundamental operations in the relational algebra are select, project, union, set difference, Cartesian product, and rename. In addition to the fundamental operations, there are several other operations—namely, set intersection, natural join, division, and assignment.

3.6.1 Fundamental Operations

- The select, project, and rename operations are called unary operations, because they operate on one relation.
- The other three operations operate on pairs of relations and are, therefore, called binary operations.

The Select Operation

The select operation selects tuples that satisfy a given predicate. We use the lowercase Greek letter sigma (σ) to denote selection. The predicate appears as a subscript to σ. The argument relation is in parentheses after the σ. That is,

$$\sigma_P(r)$$

Here, r is a name of a relation and P is a predicate.

Example: Select those tuples of the loan relation where the branch is "Perryridge".
Solution: $\sigma_{branch-name="Perryridge"}(loan)$

loan-number	branch-name	amount
L-15	Perryridge	1500
L-16	Perryridge	1300

Example: Find all tuples in which the amount lent is more than $1200.
Solution: $\sigma_{amount>1200}(loan)$

Note: In general, we allow comparisons using $=, \neq, <, \leq, >, \geq$ in the selection predicate. Furthermore, we can combine several predicates into a larger predicate by using the connectives and ($\wedge$), or ($\vee$), and not ($\neg$).

Example: Find those tuples pertaining to loans of more than $1200 made by the "Perryridge" branch.
Solution: $\sigma_{(branch-name="Perryridge")\wedge(amount>1200)}(loan)$

The Project Operation

The project operation is used to select columns of a table. It is denoted by Π. We list those attributes that we wish to appear in the result as a subscript to Π. The argument relation follows in parentheses.

$$\Pi_{A,B,C}(r)$$

Here, r is a name of a relation and A, B, C are the attributes corresponding selected column.

Example: List all loan numbers and the amount of the loan.
Solution: $\Pi_{loan-number,amount}(loan)$

loan-number	amount
L-11	900
L-14	1500
L-15	1500
L-16	1300
L-17	1000
L-23	2000
L-93	500

Example: Find those customers who live in Harrison.
Solution: $\Pi_{customer-name}(\sigma_{city="Harrison"}(Customer))$

The Union Operation

Union operation will be applied if we have to find elements which are belong into either of one relation.
For a union operation r ∪ s to be valid, we require that two conditions hold:

1. The relations r and s must be of the same arity. That is, they must have the same number of attributes.
2. The domains of the i^{th} attribute of r and the i^{th} attribute of s must be the same, for all i.

Example: Find the names of all bank customers who have either an account or a loan or both.
Solution: $\Pi_{customer}(depositer) \cup \Pi_{customer}(borrower)$

customer-name
Adams
Curry
Hayes
Jackson
Jones
Smith
Williams
Lindsay
Johnson
Turner

The Set Difference Operation

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

Example: Find all customers of the bank who have an account but not a loan.
Solution: $\Pi_{customer}(depositer) - \Pi_{customer}(borrower)$

customer-name
Johnson
Lindsay
Turner

As with the union operation, we must ensure that set differences are taken between compatible relations. Therefore, for a set difference operation r - s to be valid, we require that the relations r and s be of the same arity, and that the domains of the i^{th} attribute of r and the i^{th} attribute of s be the same.

The Cartesian-Product Operation

The Cartesian-product operation, denoted by a cross ($\times$), allows us to combine information from any two relations.We write the Cartesian product of relations r_1 and r_2 as $r_1 \times r_2$.

customer-name	*borrower. loan-number*	*loan. loan-number*	*branch-name*	*amount*
Adams	L-16	L-11	Round Hill	900
Adams	L-16	L-14	Downtown	1500
Adams	L-16	L-15	Perryridge	1500
Adams	L-16	L-16	Perryridge	1300
Adams	L-16	L-17	Downtown	1000
Adams	L-16	L-23	Redwood	2000
Adams	L-16	L-93	Mianus	500
Curry	L-93	L-11	Round Hill	900
Curry	L-93	L-14	Downtown	1500
Curry	L-93	L-15	Perryridge	1500
Curry	L-93	L-16	Perryridge	1300
Curry	L-93	L-17	Downtown	1000
Curry	L-93	L-23	Redwood	2000
Curry	L-93	L-93	Mianus	500
Hayes	L-15	L-11		900
Hayes	L-15	L-14		1500
Hayes	L-15	L-15		1500
Hayes	L-15	L-16		1300
Hayes	L-15	L-17		1000
Hayes	L-15	L-23		2000
Hayes	L-15	L-93		500
...	...	...	...	...
...	...	...	...	...
...	...	...	...	...
Smith	L-23	L-11	Round Hill	900
Smith	L-23	L-14	Downtown	1500
Smith	L-23	L-15	Perryridge	1500
Smith	L-23	L-16	Perryridge	1300
Smith	L-23	L-17	Downtown	1000
Smith	L-23	L-23	Redwood	2000
Smith	L-23	L-93	Mianus	500
Williams	L-17	L-11	Round Hill	900
Williams	L-17	L-14	Downtown	1500
Williams	L-17	L-15	Perryridge	1500
Williams	L-17	L-16	Perryridge	1300
Williams	L-17	L-17	Downtown	1000
Williams	L-17	L-23	Redwood	2000
Williams	L-17	L-93	Mianus	500

borrower x loan

Example: Find the names of all customers who have a loan at the Perryridge branch.
Solution: $\Pi_{customer-name}(\sigma_{(borrower.loan-no=loan.loan-no)\wedge(branch-name=\text{"}Perryridge\text{"})}(borrower \times loan)$

customer-name	*borrower. loan-number*	*loan. loan-number*	*branch-name*	*amount*
Adams	L-16	L-15	Perryridge	1500
Adams	L-16	L-16	Perryridge	1300
Curry	L-93	L-15	Perryridge	1500
Curry	L-93	L-16	Perryridge	1300
Hayes	L-15	L-15	Perryridge	1500
Hayes	L-15	L-16	Perryridge	1300
Jackson	L-14	L-15	Perryridge	1500
Jackson	L-14	L-16	Perryridge	1300
Jones	L-17	L-15	Perryridge	1500
Jones	L-17	L-16	Perryridge	1300
Smith	L-11	L-15	Perryridge	1500
Smith	L-11	L-16	Perryridge	1300
Smith	L-23	L-15	Perryridge	1500
Smith	L-23	L-16	Perryridge	1300
Williams	L-17	L-15	Perryridge	1500
Williams	L-17	L-16	Perryridge	1300

Result of $\sigma_{branch-name = \text{"Perryridge"}}(borrower \times loan)$.

customer-name
Adams
Hayes

Example: Consider the following tables S_1, S_2 and R_1:
Compute the following operations. (a) $S_1 \cup S_2$ (b) $S_1 \cap S_2$ (c) $S_1 - S_2$ (d) $S_1 \times R_1$

sid	*sname*	*rating*	*age*
22	Dustin	7	45.0
31	Lubber	8	55.5
58	Rusty	10	35.0

Instance S_1 of sailors

sid	*sname*	*rating*	*age*
28	yuppy	9	35.0
31	Lubber	8	55.5
44	guppy	5	35.0
58	Rusty	10	35.0

Instance S_2 of sailors

sid	*bid*	*day*
22	101	10/10/96
58	103	11/12/96

Instance R_1 of Reserves

Solution: Result of all the operations are shown as the following:-

sid	*sname*	*rating*	*age*
22	Dustin	7	45.0
31	Lubber	8	55.5
58	Rusty	10	35.0
28	yuppy	9	35.0
44	guppy	5	35.0

$S_1 \cup S_2$

sid	*sname*	*rating*	*age*
31	Lubber	8	55.5
58	Rusty	10	35.0

$S_1 \cap S_2$

sid	*sname*	*rating*	*age*
22	Dustin	7	45.0

$S_1 - S_2$

(*sid*)	*sname*	*rating*	*age*	(*sid*)	*bid*	*day*
22	Dustin	7	45.0	22	101	10/10/96
22	Dustin	7	45.0	58	103	11/12/96
31	Lubber	8	55.5	22	101	10/10/96
31	Lubber	8	55.5	58	103	11/12/96
58	Rusty	10	35.0	22	101	10/10/96
58	Rusty	10	35.0	58	103	11/12/96

$S_1 \text{ X } R_1$

The Rename Operation

Unlike relations in the database, the results of relational-algebra expressions do not have a name that we can use to refer to them. It is useful to be able to give them names; the rename operator, denoted by the lowercase Greek letter rho (ρ).
Given a relational-algebra expression E, the expression

$$\rho_x(E)$$

returns the result of expression E under the name x.
A second form of the rename operation is as follows. Assume that a relational algebra expression E has arity n. Then, the expression

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

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

Example: Find the largest account balance in the bank.
Solution: Our strategy is to (1) compute first a temporary relation consisting of those balances that are not the largest and (2) take the set difference between the relation $\Pi_{balance}(account)$ and the temporary relation just computed, to obtain the result.

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

balance
500
400
700
750
350

Figure 3.2: Result of $\Pi_{account.balance}(\sigma_{account.balance<d.balance}(account \times \rho_d(account)))$

balance
900

Figure 3.3: Largest balance

Example: Find the names of all customers who live on the same street and in the same city as Smith.
Solution: In this query, first we find Smith's street and city. Second, we match Smith's street and city with other customer street and city. If match found then we select that customer.
$\Pi_{customer-name}(\sigma_{customer.customer-street=smith.street \wedge customer.customer-city=smith.city}(customer \times \rho_{smith(street,city)}(\Pi_{customer-street,customer-city}(\sigma_{customer-name="Smith"}(customer)))))$

customer-name
Curry
Smith

3.6.2 Additional Operations

Intersection Operation

Example: Find all customers who have both a loan and an account.
Solution: $\Pi_{customer}(depositer) \cap \Pi_{customer}(borrower)$

customer-name
Hayes
Jones
Smith

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

Join

Join is a combination of a Cartesian product followed by a selection process. A Join operation pairs two tuples from different relations, if and only if a given join condition is satisfied.

Types of join operations

Theta (θ) Join or Condition Join

Theta join combines tuples from different relations provided they satisfy the theta condition. The join condition is denoted by the symbol θ.

$R_1 \bowtie_\theta R_2$ R_1 and R_2 are relations having attributes $(A_1, A_2, .., A_n)$ and $(B_1, B_2, .., B_n)$ such that the attributes don't have anything in common, that is $R_1 \cap R_2 = \phi$.
Theta join can use all kinds of comparison operators.

Equijoin

When Theta join uses only equality comparison operator, it is said to be equijoin. The above example corresponds to equijoin.

Example: Theta join and Equijoin operations are shown as following:-

(sid)	*sname*	*rating*	*age*	*(sid)*	*bid*	*day*
22	Dustin	7	45.0	58	103	11/12/96
31	Lubber	8	55.5	58	103	11/12/96

$$S1 \bowtie_{S1.sid<R1.sid} R1$$

sid	*sname*	*rating*	*age*	*bid*	*day*
22	Dustin	7	45.0	101	10/10/96
58	Rusty	10	35.0	103	11/12/96

$$S1 \bowtie_{R.sid=S.sid} R1$$

Natural-Join

Natural join does not use any comparison operator. It does not concatenate the way a Cartesian product does. We can perform a Natural Join only if there is at least one common attribute that exists between two relations. In addition, the attributes must have the same name and domain.
Natural join acts on those matching attributes where the values of attributes in both the relations are same.

Example: Natural join operation is shown as following:-

- Relations r, s:

A	B	C	D
α	1	α	a
β	2	γ	a
γ	4	β	b
α	1	γ	a
δ	2	β	b

r

B	D	E
1	a	α
3	a	β
1	a	γ
2	b	δ
3	b	∈

s

- r ⋈ s

A	B	C	D	E
α	1	α	a	α
α	1	α	a	γ
α	1	γ	a	α
α	1	γ	a	γ
δ	2	β	b	δ

Example: Find the names of all customers who have a loan at the bank, along with the loan number and the loan amount.
Solution: Query without using natural join is
$\Pi_{customer-name,loan-number,amount}(\sigma_{borrower.loan-number=loan.loan-number}(borrower \times loan))$

Equialent query using natural join is
$\Pi_{customer-name,loan-number,amount}(borrower \bowtie loan)$

customer-name	loan-number	amount
Adams	L-16	1300
Curry	L-93	500
Hayes	L-15	1500
Jackson	L-14	1500
Jones	L-17	1000
Smith	L-23	2000
Smith	L-11	900
Williams	L-17	1000

Example: Find the names of all branches with customers who have an account in the bank and who live in Harrison.
Solution: $\Pi_{branch-name}(\sigma_{customer-city="Harrison"}(customer \bowtie depositor \bowtie account))$

branch-name
Brighton
Perryridge

Note: If there is no common attributes between two relations, then natural-join and Cartesian product is equal.

The Division Operation

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

We are describing division operation through an example. Consider two relation instances A and B in which A has (exactly) two fields x and y and B has just one field y, with the same domain as in A. We define the division operation A/B as the set of all x values (in the form of unary tuples) such that for every y value in (a tuple of) B, there is a tuple $< x, y >$ in A.

Another way to understand division is as follows. For each x value in (the first column of) A, consider the set of y values that appear in (the second field of) tuples of A with that x value. If this set contains (all y values in) B, then the x value is in the result of A/B.

Example: Division operation is explain in the following figure:-

A

sno	*pno*
s1	p1
s1	p2
s1	p3
s1	p4
s2	p1
s2	p2
s3	p2
s4	p2
s4	p4

B1

pno
p2

B2

pno
p2
p4

B3

pno
p1
p2
p4

A/B1

sno
s1
s2
s3
s4

A/B2

sno
s1
s4

A/B3

sno
s1

Example: Find all customers who have an account at all the branches located in Brooklyn.
Solution: In this query, we will apply the division operator. For this we have to find numerator and denominator of the query. If numerator is N and denominator is D then final query will be N÷ D.
In this query, denominator is all the branches located in "Brooklyn". Query for this is
D = $\Pi_{branch-name}(\sigma_{branch-city="Brooklyn"}(branch))$

branch-name
Brighton
Downtown

Result of $\Pi_{branch\text{-}name}(\sigma_{branch\text{-}city = \text{"Brooklyn"}}(branch))$.

And numerator is all the customers who have an account with their branch name. Query for this is
N = $\Pi_{customer-name,branch-name}(depositor \bowtie account)$

customer-name	*branch-name*
Hayes	Perryridge
Johnson	Downtown
Johnson	Brighton
Jones	Brighton
Lindsay	Redwood
Smith	Mianus
Turner	Round Hill

Result of $\Pi_{customer\text{-}name,\ branch\text{-}name}\ (depositor \bowtie\ account)$

Therefore, the final query is

N ÷ D.

Customer-name
Johnson

Result of final query

Note: Let r(R) and s(S) be given, with S ⊆ R:
$r \div s = \Pi_{R-S}(r) - \Pi_{R-S}((\Pi_{R-S}(r) \times s) - \Pi_{R-S,S}(r))$

Assignment Operation

It is denoted by ←. If E is a relational algebra query expression, then we can assigned it as like the following:-

r ← E

3.6.3 Example:

Consider the following database which consists of three tables.

sid	*sname*	*rating*	*age*
22	Dustin	7	45.0
29	Brutus	1	33.0
31	Lubber	8	55.5
32	Andy	8	25.5
58	Rusty	10	35.0
64	Horatio	7	35.0
71	Zorba	10	16.0
74	Horatio	9	35.0
85	Art	3	25.5
95	Bob	3	63.5

Instance S_3 of sailors

sid	*bid*	*day*
22	101	10/10/98
22	102	10/10/98
22	103	10/8/98
22	104	10/7/98
31	102	11/10/98
31	103	11/6/98
31	104	11/12/98
64	101	9/5/98
64	102	9/8/98
74	103	9/8/98

Instance R_2 of Reserves

bid	*bname*	*color*
101	Interlake	blue
102	Interlake	red
103	Clipper	green
104	Marine	red

Instance B_1 of Boats Reserves

Write the following queries in relational algebra:

1. Find the names of sailors who have reserved boat 103.
   Solution: $\Pi_{sname}((\sigma_{bid=103}(Reserves)) \bowtie Sailors)$

2. Find the names of sailors who have reserved a red boat.
   Solution: $\Pi_{sname}((\sigma_{color="red"}(Boats)) \bowtie Reserves \bowtie Sailors)$

3. Find the colors of boats reserved by Lubber.
   Solution: $\Pi_{color}((\sigma_{sname="Lubber"}(Sailors)) \bowtie Reserves \bowtie Boats)$

4. Find the names of sailors who have reserved at least one boat.
   Solution: $\Pi_{sname}(Sailors \bowtie Reserves)$

5. Find the names of sailors who have reserved a red or a green boat.
   Solution: $temp \leftarrow \Pi_{sname,color}(Sailors \bowtie Reserves \bowtie Boat)$
   $\Pi_{sname}(\sigma_{color="red"}(temp)) \cup \Pi_{sname}(\sigma_{color="green"}(temp))$

6. Find the names of sailors who have reserved a red and a green boat.
   Solution: $temp \leftarrow \Pi_{sname,color}(Sailors \bowtie Reserves \bowtie Boat)$
   $\Pi_{sname}(\sigma_{color="red"}(temp)) \cap \Pi_{sname}(\sigma_{color="green"}(temp))$

7. Find the names of sailors who have reserved at least two boats.
   Solution: $temp \leftarrow \Pi_{sid,sname,bid}(Sailors \bowtie Reserves)$
   $\Pi_{sname}(\sigma_{temp1.sid=r.sid \wedge temp1.bid \neq r.bid}(temp \times \rho_r(temp)))$

8. Find the sids of sailors with age over 20 who have not reserved a red boat.
   Solution: $\Pi_{sid}(\sigma_{age>20}(Sailors)) - \Pi_{sid}((\sigma_{color="red"}(Boats)) \bowtie Reserves \bowtie Sailors)$

9. Find the names of sailors who have reserved all boats.
   Solution: $N \leftarrow \Pi_{sname,bid}(Sailors \bowtie Reserves)$
   $D \leftarrow \Pi_{bid}(Boat)$
   Therefore, final query is $N \div D$

10. Find the names of sailors who have reserved all boats called Interlake.
    Solution: $N \leftarrow \Pi_{sname,bid}(Sailors \bowtie Reserves)$
    $D \leftarrow \Pi_{bid}(\sigma_{bname="Interlake"}(Boat))$
    Therefore, final query is $N \div D$

3.6.4 Extended Relational-Algebra Operations

Generalized Projection

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

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

Where E is any relational-algebra expression, and each of $F_1, F_2, ..., F_n$ is an arithmetic expression involving constants and attributes in the schema of E. As a special case, the arithmetic expression may be simply an attribute or a constant.

Aggregate Functions

Aggregate functions take a collection of values and return a single value as a result. For example, the aggregate function **sum** takes a collection of values and returns the sum of the values. Following aggregate functions are used.

1. sum
2. avg
3. count
4. min
5. max

Example: Find the sum of all account balances.
Solution: Query for this will be

$$\mathcal{G}_{sum(balance)}(account)$$

The symbol $\mathcal{G}$ is the letter G in calligraphic font; read it as "calligraphic G." The relational-algebra operation $\mathcal{G}$ signifies that aggregation is to be applied, and its subscript specifies the aggregate operation to be applied.

Example: Find the sum of all account balances of each branch.

Solution: Query for this will be
$_{branch-name}\mathcal{G}_{sum(balance)}(account)$

Example: Find the number of depositors.
Solution: Query for this will be
$\mathcal{G}_{count(customer-name)}(depositor)$

Outer Join

The outer-join operation is an extension of the join operation to deal with missing information. There are actually three forms of the operation: left outer join, denoted ⟕; right outer join, denoted ⟖ ; and full outer join, denoted ⟗ . All three forms of outer join compute the join, and add extra tuples to the result of the join.

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

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

The full outer join(⟗) does both of those operations, padding tuples from the left relation that did not match any from the right relation, as well as tuples from the right relation that did not match any from the left relation, and adding them to the result of the join.
Example: Consider the following two relations Employee and FT-works:-

employee-name	*street*	*city*
Coyote	Toon	Hollywood
Rabbit	Tunnel	Carrotville
Smith	Revolver	Death Valley
Williams	Seaview	Seattle

Employee table

employee-name	*branch-name*	*salary*
Coyote	Mesa	1500
Rabbit	Mesa	1300
Gates	Redmond	5300
Williams	Redmond	1500

FT-works table

Natural join and left outer join are the followings:-

employee-name	*street*	*city*	*branch-name*	*salary*
Coyote	Toon	Hollywood	Mesa	1500
Rabbit	Tunnel	Carrotville	Mesa	1300
Williams	Seaview	Seattle	Redmond	1500

Employee ⋈ FT-works

employee-name	*street*	*city*	*branch-name*	*salary*
Coyote	Toon	Hollywood	Mesa	1500
Rabbit	Tunnel	Carrotville	Mesa	1300
Williams	Seaview	Seattle	Redmond	1500
Smith	Revolver	Death Valley	*null*	*null*

Employee ⟕ FT-works

Right outer join and Full outer join are the followings:-

employee-name	*street*	*city*	*branch-name*	*salary*
Coyote	Toon	Hollywood	Mesa	1500
Rabbit	Tunnel	Carrotville	Mesa	1300
Williams	Seaview	Seattle	Redmond	1500
Gates	*null*	*null*	Redmond	5300

Employee ⟖ FT-works

employee-name	*street*	*city*	*branch-name*	*salary*
Coyote	Toon	Hollywood	Mesa	1500
Rabbit	Tunnel	Carrotville	Mesa	1300
Williams	Seaview	Seattle	Redmond	1500
Smith	Revolver	Death Valley	*null*	*null*
Gates	*null*	*null*	Redmond	5300

Employee ⟗ FT-works

3.6.5 Modification of the Database

Deletion

We can delete only whole tuples; we cannot delete values on only particular attributes. In relational algebra a deletion is expressed by

$$r \leftarrow r - E$$

Where r is a relation and E is a relational-algebra query.

Example: Delete all of Smith's account records.
Solution: $depositor \leftarrow depositor - \sigma_{customer-name}(depositor)$

Example: Delete all loans with amount in the range 0 to 50.
Solution: $loan \leftarrow loan - \sigma_{amount \geq 0 \ \wedge \ amount \leq 50}(loan)$

Example: Delete all accounts at branches located in Needham.
Solution:

$$account \leftarrow account - r$$

Where r is

$$r \leftarrow \Pi_{account-number,branch-name,balance}(\sigma_{branch-city="Needham"}(branch \bowtie account))$$

Insertion

To insert data into a relation, we either specify a tuple to be inserted or write a query whose result is a set of tuples to be inserted. Obviously, the attribute values for inserted tuples must be members of the attribute's domain. Similarly, tuples inserted must be of the correct arity. The relational algebra expresses an insertion by
$r \leftarrow r \cup E$
where r is a relation and E is a relational-algebra expression.

Example: Suppose that we wish to insert the fact that Smith has $1200 in account A-973 at the Perryridge branch.
Solution:

$$depositor \leftarrow depositor \cup ("Smith", "A-973")$$
$$account \leftarrow account \cup ("A-973", "Perryridge", 1200)$$

Updating

To update value of a particular row into a relation, we write the following type of query:-

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

where each F_i is either the i^{th} attribute of r, if the i^{th} attribute is not updated, or, if the attribute is to be updated, F_i is an expression, involving only constants and the attributes of r, that gives the new value for the attribute.
If we want to select some tuples from r and to update only them, we can use the following expression; here, P denotes the selection condition that chooses which tuples to update:

$$r \leftarrow \Pi_{F_1,F_2,...,F_n}(\sigma_P(r)) \cup (r - \sigma_P(r))$$

Example: Suppose that interest payments are being made, and that all balances are to be increased by 5 percent.
Solution: $account \leftarrow \Pi_{account-number,branch-name,balance*1.05}(account)$

Example: Suppose that accounts with balances over $10,000 receive 6 percent interest, whereas all others receive 5 percent.
Solution: $account \leftarrow \Pi_{account-number,branch-name,balance*1.06}(\sigma_{balance>10000}(account)) \cup \Pi_{account-number,branch-name,balance*1.05}(\sigma_{balance \leq 10000}(account))$

3.6.6 Views

Any relation that is not part of the logical model, but is made visible to a user as a virtual relation, is called a view.
We define a view by using the create view statement. To define a view, we must give the view a name, and must state the query that computes the view. The form of the create view statement is

create view v as ¡query expression¿ where¡query expression¿is any legal relational-algebra query expression. The view name is represented by v.

Example: Consider the view consisting of branches and their customers. We wish this view to be called all-customer. We define this view as follows:

create view all-customer as $\Pi_{branch-name,customer-name}(depositor \bowtie account)$
$\cup \Pi_{branch-name,customer-name}(borrower \bowtie loan)$

Once we have defined a view, we can use the view name to refer to the virtual relation that the view generates. Using the view all-customer, we can find all customers of the Perryridge branch by writing

$Pi_{customer-name}(\sigma_{branch-name=\backslash Perryridge"}(all - customer))$

3.7 The Tuple Relational Calculus

A query in the tuple relational calculus is expressed as

{ t ! P(t)}

that is, it is the set of all tuples t such that predicate P is true for t. We use t[A] to denote the value of tuple t on attribute A, and we use t ∈ r to denote that tuple t is in relation r.

3.7.1 Example Queries

- Find the branch-name, loan-number, and amount for loans of over $1200.
  Solution: $\{ t \mid t \in loan \wedge t[amount] > 1200 \}$

- Find the loan number for each loan of an amount greater than $1200.
  Solution: $\{ t \mid \exists\ s \in loan(t[loan - number] = s[loan - number] \wedge s[amount] > 1200) \}$

- Find the names of all customers who have a loan from the Perryridge branch.
  Solution: $\{ t \mid \exists\ s \in borrower(t[customer - name] = s[customer - name] \wedge \exists\ s \in loan(u[loan{-}number] = s[loan{-}number] \wedge u[branch{-}name] = "Perryridge"))\}$

- Find all customers who have a loan, an account, or both at the bank.
  Solution: $\{ t \mid \exists s \in borrower(t[customer - name] = s[customer - name]) \vee\ \exists u \in depositor(t[customer - name] = u[customer - name])\}$

- Find those customers who have both an account and a loan at the bank.
  Solution: $\{ t \mid \exists s \in borrower(t[customer - name] = s[customer - name]) \wedge\ \exists u \in depositor(t[customer - name] = u[customer - name])\}$

- Find all customers who have an account at the bank but do not have a loan from the bank.
  Solution: $\{ t \mid \exists s \in depositor(t[customer-name] = s[customer-name]) \wedge \neg\exists u \in borrower(t[customer-name] = u[customer-name])\}$

- Find all customers who have an account at all branches located in Brooklyn.
  Solution: $\{ t \mid \forall u \in branch(u[branch{-}city] = "Brooklyn" \Rightarrow \exists s \in depositor(t[customer-name] = s[customer-name] \wedge \exists w \in account(w[account-number] = s[account-number] \wedge w[branch-name] = u[branch-name])))\}$

Note:

1. The formula P $\Rightarrow$ Q means "P implies Q"; that is, "if P is true, then Q must be true."

2. Note that P $\Rightarrow$ Q is logically equivalent to $\neg$P $\vee$ Q.

3.8 Domain Relational Calculus

It uses domain variables that take on values from an attributes domain, rather than values for an entire tuple. The domain relational calculus, however, is closely related to the tuple relational calculus.
An expression in the domain relational calculus is of the form

$$\{< x_1, x_2, ..., x_n > \mid P(x_1, x_2, ..., x_n)\}$$

where $x_1, x_2, ..., x_n$ represent domain variables. P represents a formula composed of atoms, as was the case in the tuple relational calculus.

3.8.1 Example Queries

- Find the branch-name, loan-number, and amount for loans of over \$1200.
  Solution: $\{ < l, b, a > \mid < l, b, a > \in loan \wedge a > 1200 \}$

- Find the loan number for each loan of an amount greater than \$1200.
  Solution: $\{ < l > \mid \exists\, b, a\ (< l, b, a > \in loan \wedge a > 1200) \}$

- Find the names of all customers who have a loan from the Perryridge branch and find the loan amount.
  Solution: $\{ < c, a > \mid \exists\ l(< c, l > \in borrower \wedge \exists b\ (< l, b, a > \in loan \wedge b = "Perryridge"))\}$

- Find all customers who have a loan, an account, or both at the Perryridge branch.
  Solution: $\{< c > \mid \exists l(< c, l > \in borrower \wedge \exists b, a(< l, b, a > \in loan \wedge b = "Perryridge")) \vee \exists a(< c, a > \in depositor \wedge \exists b, n(< a, b, n > \in account \wedge b = "Perryridge"))\}$

- Find all customers who have an account at all branches located in Brooklyn.
  Solution: $\{< c > \mid \forall x, y, z(< x, y, z > \in branch \wedge y = "Brooklyn" \Rightarrow \exists a, b(< a, x, b > \in account \wedge < c, a > \in depositor))\}$

3.9 Exercise

1. Consider the following relational database, where the primary keys are underlined.
   employee (person-name, street, city)
   works (person-name, company-name, salary)
   company (company-name, city)
   manages (person-name, manager-name)

   Give an expression in the relational algebra to express each of the following queries:

   (a) Find the names of all employees who work for First Bank Corporation.

   (b) Find the names and cities of residence of all employees who work for First Bank Corporation.

   (c) Find the names, street address, and cities of residence of all employees who work for First Bank Corporation and earn more than $10,000 per annum.

   (d) Find the names of all employees in this database who live in the same city as the company for which they work.

   (e) Find the names of all employees who live in the same city and on the same street as do their managers.

   (f) Find the names of all employees in this database who do not work for First Bank Corporation.

   (g) Find the names of all employees who earn more than every employee of Small Bank Corporation.

   (h) Assume the companies may be located in several cities. Find all companies located in every city in which Small Bank Corporation is located.

   (i) Find the company with the most employees.

   (j) Find the company with the smallest payroll.

   (k) Find those companies whose employees earn a higher salary, on average, than the average salary at First Bank Corporation.

   Solution:

   (a) $\Pi_{person-name}(\sigma_{company-name="First\ Bank\ Corporation"}(works))$

   (b) $\Pi_{person-name,city}(\sigma_{company-name="First\ Bank\ Corporation"}(employee \bowtie works))$

   (c) $\Pi_{person-name,street,city}(\sigma_{company-name="First\ Bank\ Corporation" \wedge salary>10000}(employee \bowtie works))$

   (d) $\Pi_{person-name}(employee \bowtie works \bowtie company))$

   (e) $temp \leftarrow \Pi_{manager-name,street,city}(\sigma_{employee.person-name=manages.manager-name}(employee \times manages)$
   $\Pi_{person-name}(employee \bowtie manages \bowtie temp)$

   (f) $\Pi_{person-name}(\sigma_{company-name \neq "FirstBankCorporation"}(works))$

   (g) $\Pi_{person-name}(works) - \Pi_{works.person-name}(\sigma_{works.salary \leq r.salary}(works \times (\rho_r(\sigma_{company-name="Small\ Bank\ corporation"}(works)))))$

   (h) $company \div \Pi_{city}(\sigma_{company-name="Small\ Bank\ Corporation"}(company)))$

(i) $temp \leftarrow {}_{company-name}\mathcal{G}_{count(person-name)\ as\ person-count}(works)$
$\Pi_{company-name}(\sigma_{temp.person-count=r.max-employee}(temp \times$
$\rho_r(\mathcal{G}_{max(person-count)\ as\ max-employee}(temp))))$

(j) $temp \leftarrow {}_{company-name}\mathcal{G}_{sum(salary)\ as\ payroll}(works)$
$\Pi_{company-name}(\sigma_{temp.payroll=r.min-payroll}(temp \times \rho_r(\mathcal{G}_{min(payroll)\ as\ min-payroll}(temp))))$

(k) $temp1 \leftarrow \mathcal{G}_{avg(salary)\ as\ avg-salary}(\sigma_{company-name=\text{"}First\ Bank\ Corporation\text{"}}(works)$
$temp2 \leftarrow {}_{company-name}\ \mathcal{G}_{avg(salary)\ as\ avg-salary}(works)$
$\Pi_{company-name}(\sigma_{temp2.avg-salary>temp1.avg-salary}(temp2 \times temp1))$

2. Consider the relational database of previous question. Give an expression in the relational algebra for each request:

(a) Modify the database so that Jones now lives in Newtown.

(b) Give all employees of First Bank Corporation a 10 percent salary raise.

(c) Give all managers in this database a 10 percent salary raise.

(d) Give all managers in this database a 10 percent salary raise, unless the salary would be greater than $100,000. In such cases, give only a 3 percent raise.

(e) Delete all tuples in the works relation for employees of Small Bank Corporation.

Solution:

(a) $employee \leftarrow \Pi_{person-name,street,\text{"}Newtown\text{"}}(\sigma_{person-name=\text{"}Jones\text{"}}(Employee)) \cup (employee - \sigma_{person_name=\text{"}Jones\text{"}}(employee))$

(b) $works \leftarrow \Pi_{person-name,company-name,salary*1.1}(\sigma_{company-name=\text{"}FirstBankCorporation\text{"}}(works)) \cup (works - \sigma_{company-name=\text{"}FirstBankCorporation\text{"}}(works)$

(c) $temp \leftarrow \Pi_{works.person-name,company-name,salary}(\sigma_{works.person-name=manages.manages-name}(works \times manages))$
$works \leftarrow (works - temp) \cup \Pi_{works.person-name,comapny-name,salary*1.1}(temp)$

(d) $temp1 \leftarrow \Pi_{works.person-name,company-name,salary}(\sigma_{works.person-name=manages.manages-name}(works \times manages))$
$temp2 \leftarrow \Pi_{works.person-name,company-name,salary*1.03}(\sigma_{salary*1.1>100000}(temp1))$
$temp2 \leftarrow temp2 \cup \Pi_{works,person-name,company-name,salary*1.1}(\sigma_{salary*1.1\leq 100000}(temp1))$
$works \leftarrow (works - temp1) \cup temp2$

(e) $works \leftarrow works - \sigma_{company-name=\text{"}Small\ Bank\ Corporation\text{"}}(works)$

3. Let the following relation schemas be given:
R = (A, B, C) and S = (D, E, F)
Let relations r(R) and s(S) be given. Give an expression in the tuple relational calculus that is equivalent to each of the following:

(a) $\Pi_A(r)$

(b) $\sigma_{B=17}(r)$

(c) r×s

(d) $\Pi_{A,F}(\sigma_{C=D}(r \times s))$

Solution:

(a) $\{t \mid \exists u \in r(t[A] = u[A])\}$

(b) $\{t \mid t \in r \wedge t[B] = 17\}$

(c) $\{t \mid \exists u \in r(t[A] = u[A] \wedge t[B] = u[B] \wedge t[C] = u[C] \wedge \exists w \in s(t[D] = w[D] \wedge t[E] = w[E] \wedge t[F] = w[F]))\}$

(d) $\{t \mid \exists u \in r(t[A] = u[A] \wedge \exists w \in s(t[F] = w[F] \wedge u[C] = w[D]))\}$

4. Let R = (A, B, C), and let r_1 and r_2 both be relations on schema R. Give an expression in the domain relational calculus that is equivalent to each of the following:

(a) $\Pi_A(r_1)$

(b) $\sigma_{B=17}(r_1)$

(c) $r_1 \cup r_2$

(d) $r_1 \cap r_2$

(e) $r_1 - r_2$

(f) $\Pi_{A,B}(r_1) \bowtie \Pi_{B,C}(r_2)$

Solution:

(a) $\{< a > \mid \exists b, c(< a, b, c > \in r_1)\}$

(b) $\{< a, b, c > \mid \ < a, b, c > \in r_1 \wedge b = 17)\}$

(c) $\{< a, b, c > \mid \ < a, b, c > \in r_1 \vee < a, b, c > \in r_2\}$

(d) $\{< a, b, c > \mid \ < a, b, c > \in r_1 \wedge < a, b, c > \in r_2\}$

(e) $\{< a, b, c > \mid \ < a, b, c > \in r_1 \wedge < a, b, c > \notin r_2\}$

(f) $\{< a, b, c > \mid \exists p, q(< a, b, p > \in r_1 \wedge < q, b, c > \in r_2)\}$

5. Let R = (A, B) and S = (A, C), and let r(R) and s(S) be relations. Write relational-algebra expressions equivalent to the following domain-relational calculus expressions:

(a) $\{< a > \mid \exists b(< a, b > \in r \wedge b = 17)\}$

(b) $\{< a, b, c > \mid < a, b > \in r \wedge < a, c > \in s\}$

(c) $\{< a > \mid \exists b(< a, b > \in r) \vee \forall c(\exists d(< d, c > \in s) \Rightarrow < a, c > \in s)\}$

(d) $\{< a > \mid \exists c(< a, c > \in s \wedge \exists b_1, b_2(< a, b_1 > \in r \wedge < c, b_2 > \in r \wedge b_1 > b_2))\}$

Solution:

(a) $\Pi_A(\sigma_{B=17}(r))$

(b) $r \bowtie s$

(c) $\Pi_A(r) \cup (s \div \Pi_C(s))$

(d) $\Pi_{r.A}(\sigma_{r.B>r1.B}((r \bowtie s) \times (\rho_{r1}(r))))$

6. Given two relations R_1 and R_2, where R_1 contains N_1 tuples,R_2 contains N_2 tuples, and $N_2 > N_1 > 0$, give the minimum and maximum possible sizes (in tuples) for the result relation produced by each of the following relational algebra expressions. In each case, state any assumptions about the schemas for R_1 and R_2 that are needed to make the expression meaningful:

(a) $R_1 \cup R_2$

(b) $R_1 \cap R_2$

(c) $R_1 - R_2$

(d) $R_1 \times R_2$

(e) $\sigma_{a=5}(R_1)$

(f) $\Pi_a(R_1)$

(g) $R_1 \div R_2$

Solution:

(a) Minimum number of tuples = N_2
Maximum number of tuples = $N_1 + N_2$

(b) Minimum number of tuples = 0
Maximum number of tuples = N_1

(c) Minimum number of tuples = 0
Maximum number of tuples = N_1

(d) Minimum number of tuples = $N_1 * N_2$
Maximum number of tuples = $N_1 * N_2$

(e) Assume attribute a in R_1 is a primary key. In this case
Minimum number of tuples = 0
Maximum number of tuples = 1

Assume attribute a in R_1 is not a primary key. In this case
Minimum number of tuples = 0
Maximum number of tuples = N_1

(f) Assume attribute a in R_1 is a primary key. In this case
Minimum number of tuples = N_1
Maximum number of tuples = N_1

Assume attribute a in R_1 is not a primary key. In this case
Minimum number of tuples = 1
Maximum number of tuples = N_1

(g) Minimum number of tuples = 0
Maximum number of tuples = 0

7. Consider the following schema:
Suppliers(sid: integer, sname: string, address: string)
Parts(pid: integer, pname: string, color: string)
Catalog(sid: integer, pid: integer, cost: real)

The key fields are underlined, and the domain of each field is listed after the field name. Thus sid is the key for Suppliers, pid is the key for Parts, and sid and pid together form the key for Catalog. The Catalog relation lists the prices charged for parts by Suppliers. Write the following queries in relational algebra, tuple relational calculus, and domain relational calculus:

(a) Find the names of suppliers who supply some red part.

(b) Find the sids of suppliers who supply some red or green part.

(c) Find the sids of suppliers who supply some red part or are at 221 Packer Street.

(d) Find the sids of suppliers who supply some red part and some green part.

(e) Find the sids of suppliers who supply every part.

(f) Find the sids of suppliers who supply every red part.

(g) Find the sids of suppliers who supply every red or green part.

(h) Find the sids of suppliers who supply every red part or supply every green part.

(i) Find pairs of sids such that the supplier with the first sid charges more for some part than the supplier with the second sid.

(j) Find the pids of parts that are supplied by at least two different suppliers.

(k) Find the pids of the most expensive parts supplied by suppliers named Yosemite Sham.

Solution:

(a) **Relational algebra query is**
$\Pi_{sname}(Suppliers \bowtie Catalog \bowtie \Pi_{pid}(\sigma_{color="red"}(Parts)))$

Tuple relational calculus query is
$\{ t \mid \exists s \in Suppliers(t[sname] = s[sname] \wedge \exists u \in Catalog(s[sid] = u[sid] \wedge \exists w \in Parts(u[pid] = w[pid] \wedge w[color] = "red")))\}$

Domain relational calculus query is
$\{< b > \mid \exists a, c(< a, b, c >\in Suppliers \wedge \exists d, e(< a, d, e >\in Catalog \wedge \exists f, g(< d, f, g >\in Parts \wedge g = "red")))\}$

(b) **Relational algebra query is**
$\Pi_{sid}(\Pi_{pid}(\sigma_{color="red" \vee color="green"(Parts)} \bowtie Catalog))$

Tuple relational calculus query is
$\{ t \mid \exists u \in Catalog(t[sid] = u[sid] \wedge \exists w \in Parts(u[pid] = w[pid] \wedge (w[color] = "red" \vee w[color] = "green")))\}$

Domain relational calculus query is
$\{< a > \mid \exists b, c(< a, b, c >\in Catalog \wedge \exists d, e(< b, d, e >\in Parts \wedge (e = "red" \vee e = "green")))\}$

(c) **Relational algebra query is**
$\Pi_{sid}(\sigma_{color="red"}(Catalog \bowtie Parts)) \cup \Pi_{sid}(\sigma_{address="221\ Packer\ Street"}(Suppliers))$

Tuple relational calculus query is
$\{ t \mid \exists u \in Catalog(t[sid] = u[sid] \wedge \exists w \in Parts(u[pid] = w[pid] \wedge w[color] = "red"))) \vee \exists s \in Suppliers(t[sid] = s[sid] \wedge s[address] = "220\ Packer\ Street")\}$

Domain relational calculus query is
$\{< a > \mid \exists b, c(< a, b, c > \in Catalog \wedge \exists d, e(< b, d, e > \in Parts \wedge e = "red")) \vee \exists b, c(< a, b, c > \in Suppliers \wedge c = "220\ Packer\ Street")\}$

(d) **Relational algebra query is**
$\Pi_{sid}(\sigma_{color="red"}(Catalog \bowtie Parts)) \cap \Pi_{sid}(\sigma_{color="green"}(Catalog \bowtie Parts))$

Tuple relational calculus query is
$\{\ t \mid \exists u \in Catalog(t[sid] = u[sid] \wedge \exists w \in Parts(u[pid] = w[pid] \wedge w[color] = "red"))) \wedge \exists u \in Catalog(t[sid] = u[sid] \wedge \exists w \in Parts(u[pid] = w[pid] \wedge w[color] = "green")))\}$

Domain relational calculus query is
$\{< a > \mid \exists b, c(< a, b, c > \in Catalog \wedge \exists d, e(< b, d, e > \in Parts \wedge e = "red")) \wedge \exists b, c(< a, b, c > \in Catalog \wedge \exists d, e(< b, d, e > \in Parts \wedge e = "green"))\}$

(e) **Relational algebra query is**
$\Pi_{sid,pid}(Catalog) \div \Pi_{pid}(Parts)$

Tuple relational calculus query is
$\{\ t \mid \forall s \in Parts \Rightarrow \exists u \in Catalog(t[sid] = u[sid] \wedge s[pid] = u[pid])\}$

Domain relational calculus query is
$\{\ < s >\ \mid \forall a, b, c(< a, b, c > \in Parts \Rightarrow \exists d(< s, a, d > \in Catalog))\}$

(f) **Relational algebra query is**
$\Pi_{sid,pid}(Catalog) \div \Pi_{pid}(Parts)$

Tuple relational calculus query is
$\{\ t \mid \forall s \in Parts \Rightarrow \exists u \in Catalog(t[sid] = u[sid] \wedge s[pid] = u[pid])\}$

Domain relational calculus query is
$\{\ < s >\ \mid \forall a, b, c(< a, b, c > \in Parts \Rightarrow \exists d(< s, a, d > \in Catalog))\}$

(g) **Relational algebra query is**
$\Pi_{sid,pid}(Catalog) \div \Pi_{pid}(\sigma_{color="red" \vee color="green"}(Parts))$

Tuple relational calculus query is
Domain relational calculus query is

(h) **Relational algebra query is**
$\Pi_{sid,pid}(Catalog) \div \Pi_{pid}(\sigma_{color="red"}(Parts)) \vee \Pi_{sid,pid}(Catalog) \div \Pi_{pid}(\sigma_{color="green"}(Parts))$

Tuple relational calculus query is
Domain relational calculus query is

(i) **Relational algebra query is**
$\Pi_{r.sid,s.sid}(\sigma_{r.pid=s.pid \wedge r.sid \neq s.sid \wedge r.cost > s.cost}(\rho_r(Catalog) \times \rho_s(Catalog)))$

Tuple relational calculus query is
Domain relational calculus query is

(j) **Relational algebra query is**
$\Pi_{r.pid}(\sigma_{r.pid=s.pid \wedge r.sid \neq s.sid}(\rho_r(Catalog) \times \rho_s(Catalog))$

Tuple relational calculus query is
Domain relational calculus query is

(k) **Relational algebra query is**
$r \leftarrow \Pi_{pid,cost}(\sigma_{sname="YosemiteSham"}(Suppliers) \bowtie Catalog)$
$\Pi_{pid}(r) - \Pi_{r.pid}(\sigma_{r.cost<s.cost}(r \times \rho_s(r)))$

Tuple relational calculus query is
Domain relational calculus query is

8. Consider the Supplier-Parts-Catalog schema from the previous question. State what the following queries compute:

(a) $\Pi_{sname}(\Pi_{sid}(\sigma_{color="red"}(Parts)) \bowtie (\sigma_{cost<100}(Catalog)) \bowtie Suppliers)$

(b) $\Pi_{sname}(\Pi_{sid}((\sigma_{color="red"}(Parts)) \bowtie (\sigma_{cost<100}(Catalog)) \bowtie Suppliers))$

(c) $(\Pi_{sname}((\sigma_{color="red"}(Parts)) \bowtie (\sigma_{cost<100}(Catalog)) \bowtie Suppliers)) \cap$
$(\Pi_{sname}((\sigma_{color="green"}(Parts)) \bowtie (\sigma_{cost<100}(Catalog)) \bowtie Suppliers))$

(d) $(\Pi_{sid}((\sigma_{color="red"}(Parts)) \bowtie (\sigma_{cost<100}(Catalog)) \bowtie Suppliers)) \cap$
$(\Pi_{sid}((\sigma_{color="green"}(Parts)) \bowtie (\sigma_{cost<100}(Catalog)) \bowtie Suppliers))$

(e) $\Pi_{sname}((\Pi_{sid,sname}((\sigma_{color="red"}(Parts)) \bowtie (\sigma_{cost<100}(Catalog)) \bowtie Suppliers)) \cap$
$(\Pi_{sid,sname}((\sigma_{color="green"}(Parts)) \bowtie (\sigma_{cost<100}(Catalog)) \bowtie Suppliers)))$

Solution:

(a) Find the Supplier names of the suppliers who supply a red part that costs less than 100 dollars.

(b) This Relational Algebra statement does not return anything because of the sequence of projection operators. Once the sid is projected, it is the only field in the set. Therefore, projecting on sname will not return anything.

(c) Find the Supplier names of the suppliers who supply a red part that costs less than 100 dollars and a green part that costs less than 100 dollars.

(d) Find the Supplier ids of the suppliers who supply a red part that costs less than 100 dollars and a green part that costs less than 100 dollars.

(e) Find the Supplier names of the suppliers who supply a red part that costs less than 100 dollars and a green part that costs less than 100 dollars.

9. Consider the following relations containing airline flight information:
Flights(flno: integer, from: string, to: string, distance: integer, departs: time, arrives: time)
Aircraft(aid: integer, aname: string, cruisingrange: integer)
Certified(eid: integer, aid: integer)
Employees(eid: integer, ename: string, salary: integer)

   Note that the Employees relation describes pilots and other kinds of employees as well; every pilot is certified for some aircraft (otherwise, he or she would not qualify as a pilot), and only pilots are certified to fly.
Write the following queries in relational algebra, tuple relational calculus, and domain relational calculus. Note that some of these queries may not be expressible in relational algebra (and, therefore, also not expressible in tuple and domain relational calculus)! For such queries, informally explain why they cannot be expressed.

   (a) Find the eids of pilots certified for some Boeing aircraft.

   (b) Find the names of pilots certified for some Boeing aircraft.

   (c) Find the aids of all aircraft that can be used on non-stop flights from Bonn to Madras.

   (d) Identify the flights that can be piloted by every pilot whose salary is more than $100,000. (Hint: The pilot must be certified for at least one plane with a sufficiently large cruising range.)

   (e) Find the names of pilots who can operate some plane with a range greater than 3,000 miles but are not certified on any Boeing aircraft.

   (f) Find the eids of employees who make the highest salary.

   (g) Find the eids of employees who make the second highest salary.

   (h) Find the eids of pilots who are certified for the largest number of aircraft.

   (i) Find the eids of employees who are certified for exactly three aircraft.

   (j) Find the total amount paid to employees as salaries.

   Solution:

   (a) **Relational algebra query is**
   $\Pi_{eid}(Certified \bowtie \sigma_{aname="Boeing"}(Aircraft))$

   (b) **Relational algebra query is**
   $\Pi_{ename}(Employees \bowtie Certified \bowtie \sigma_{aname="Boeing"}(Aircraft))$

   (c) **Relational algebra query is**
   $\Pi_{aid}(\sigma_{cruisingrange>distance}(Aircraft \times \sigma_{from="Bonn" \wedge to="Madras"}(Flights)))$

   (d) **Relational algebra query is**
   $\Pi_{flno}(\sigma_{cruisingrange>distance \wedge salary>100000}(Flights \bowtie Aircraft \bowtie Certified \bowtie Employee))$

   (e) **Relational algebra query is**
   $\Pi_{ename}(Employees \bowtie Certified \bowtie \sigma_{cruisingrange>3000 \wedge aname \neq "Boeing"}(Aircraft))$

(f) **Relational algebra query is**
The approach to take is first find all the employees who do not have the highest salary. Subtract these from the original list of employees and what is left is the highest paid employees.
$\Pi_{eid}(Employees) - \Pi_{r.eid}(\sigma_{r.salary<s.salary}(\rho_r(Employees) \times \rho_s(Employees)))$

(g) **Relational algebra query is**
$temp \leftarrow \Pi_{r.eid,r.salary}(\sigma_{r.salary<s.salary}(\rho_r(Employees) \times \rho_s(Employees)))$
$\Pi_{eid}(temp) - \Pi_{temp.eid}(\sigma_{temp.salary<s.salary}(temp \times \rho_s(temp)))$

(h) **Relational algebra query is**
$temp \leftarrow \ _{eid}\mathcal{G}_{count(aid)\ as\ count-aid}(Certified)$
$temp1 \leftarrow \mathcal{G}_{max(count-aid)\ as\ max}(temp)$
$\Pi_{eid}(\sigma_{count-aid=max}(temp \times temp1))$

(i) **Relational algebra query is**
$temp \leftarrow \ _{eid}\mathcal{G}_{count(aid)\ as\ count-aid}(Certified)$
$\Pi_{eid}(\sigma_{count-aid=3}(temp))$

(j) **Relational algebra query is**
$\mathcal{G}_{sum(salary)}(Employees)$

10. What is an unsafe query? Give an example and explain why it is important to disallow such queries.
Solution: An unsafe query is a query in relational calculus that has an infinite number of results. An example of such a query is:

$$\{S \mid \neg(S \in Sailors)\}$$

The query is for all things that are not sailors which of course is everything else. Clearly there is an infinite number of answers, and this query is unsafe. It is important to disallow unsafe queries because we want to be able to get back to users with a list of all the answers to a query after a finite amount of time.

Chapter 4

SQL

4.1 Basic Structure

The basic structure of an SQL expression consists of three clauses: select, from, and where.

- The select clause corresponds to the projection operation of the relational algebra. It is used to list the attributes desired in the result of a query.
- The from clause corresponds to the Cartesian-product operation of the relational algebra. It lists the relations to be scanned in the evaluation of the expression.
- The where clause corresponds to the selection predicate of the relational algebra. It consists of a predicate involving attributes of the relations that appear in the from clause.

A typical SQL query has the form

select $A_1, A_2, ..., A_n$
from $r_1, r_2, ..., r_m$
where P

Each A_i represents an attribute, and each r_i a relation. P is a predicate. The query is equivalent to the relational-algebra expression

$$\Pi_{A_1,A_2,...,A_n}(\sigma_P(r_1 \times r_2 \times ..... \times r_m))$$

4.1.1 Schema Definition in SQL

We define an SQL relation by using the **create table** command:

create table $r(A_1\ D_1, A_2\ D_2, ..., A_n\ D_n, < integrity - constraint_1 >, ..., < integrity - constraint_k >)$ where r is the name of the relation, each A_i is the name of an attribute in the schema of relation r, and D_i is the domain type of values in the domain of attribute A_i. The allowed integrity constraints include

- **primary key** $(A_{j_1}, A_{j_2}, ..., A_{j_m})$**:** The primary key specification says that attributes $A_{j_1}, A_{j_2}, ..., A_{j_m}$ form the primary key for the relation. The primary key attributes are required to be non-null and unique; that is, no tuple can have a null value for a primary key attribute, and no two tuples in the relation can be equal on all the primary-key attributes. Although the primary key specification is optional, it is generally a good idea to specify a primary key for each relation.

- **check(P):** The check clause specifies a predicate P that must be satisfied by every tuple in the relation.

Example: Consider the following definition of tables:-

- 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),
  check (assets $\geq$ 0))

- create table account
  (account-number char(10),
  branch-name char(15),
  balance integer,
  primary key (account-number),
  check (balance $\geq$ 0))

- create table depositor
  (customer-name char(20),
  account-number char(10),
  primary key (customer-name, account-number))

- create table student
  (name char(15) not null,
  student-id char(10),
  degree-level char(15),
  primary key (student-id),
  check (degree-level in ('Bachelors', 'Masters', 'Doctorate')))

Note: SQL also supports an integrity constraint
unique $(A_{j_1}, A_{j_2}, ..., A_{j_m})$
The unique specification says that attributes $A_{j_1}, A_{j_2}, ..., A_{j_m}$ form a candidate key.

4.1.2 Some queries

Consider the following relation schemas:-

Branch-schema = (branch-name, branch-city, assets)
Customer-schema = (customer-name, customer-street, customer-city)
Loan-schema = (loan-number, branch-name, amount)
Borrower-schema = (customer-name, loan-number)
Account-schema = (account-number, branch-name, balance)
Depositor-schema = (customer-name, account-number)

- Find the names of all branches in the loan relation.
  Solution:
  select branch-name
  from loan

- Find all loan numbers for loans made at the Perryridge branch with loan amounts greater that $1200.
  Solution:
  select loan-number
  from loan
  where branch-name = 'Perryridge' and amount $>$ 1200

- Find the loan number of those loans with loan amounts between $90,000 and $100,000.
  Solution:
  select loan-number
  from loan
  where amount $\leq$ 100000 and amount $\geq$ 90000

- For all customers who have a loan from the bank, find their names,loan numbers and loan amount.
  Solution:
  select customer-name, borrower.loan-number, amount
  from borrower, loan
  where borrower.loan-number = loan.loan-number

- Find the customer names, loan numbers, and loan amounts for all loans at the Perryridge branch.
  Solution:
  select customer-name, borrower.loan-number, amount
  from borrower, loan
  where borrower.loan-number = loan.loan-number and branch-name = 'Perryridge'

-

4.1.3 Rename operation

SQL provides a mechanism for renaming both relations and attributes. It uses the **as** clause, taking the form:
old-name as new-name
The **as** clause can appear in both the select and from clauses.

Example:

- select customer-name, borrower.loan-number **as** loan-id, amount
  from borrower, loan
  where borrower.loan-number = loan.loan-number

- For all customers who have a loan from the bank, find their names, loan numbers, and loan amount
  Solution:
  select customer-name, T.loan-number, S.amount
  from borrower **as** T, loan **as** S
  where T.loan-number = S.loan-number

- Find the names of all branches that have assets greater than at least one branch located in Brooklyn.
  Solution:
  select distinct T.branch-name
  from branch **as** T, branch **as**S
  where T.assets > S.assets and S.branch-city = 'Brooklyn'

4.1.4 String Operations

The most commonly used operation on strings is pattern matching using the operator like. We describe patterns by using two special characters:

- Percent (%): The % character matches any substring.
- Underscore (_): The _ character matches any character.

Patterns are case sensitive; that is, uppercase characters do not match lowercase characters, or vice versa. To illustrate pattern matching, we consider the following examples:

- 'Perry%' matches any string beginning with "Perry".
- '%idge%' matches any string containing "idge" as a substring, for example, 'Perryridge', 'Rock Ridge', 'Mianus Bridge', and 'Ridgeway'.
- '___' matches any string of exactly three characters.
- '___%' matches any string of at least three characters.

Example: Find the names of all customers whose street address includes the substring 'Main'.
Solution:
select customer-name
from customer
where customer-street like '%Main%'

4.1.5 Ordering the Display of Tuples

To display the result in the sorted order, we use the **order by** clause.
Example: To list in alphabetic order all customers who have a loan at the Perryridge branch
Solution:
select distinct customer-name
from borrower, loan
where borrower.loan-number = loan.loan-number and
branch-name = 'Perryridge'
order by customer-name

Example:
select *
from loan
order by amount desc, loan-number asc

4.1.6 Set Operations

The SQL operations union, intersect, and except operate on relations and correspond to the relational algebra operations $\cup$, $\cap$, and -.

- Find all customers having a loan, an account, or both at the bank.
  Solution:
  (select customer-name
  from depositor)
  union
  (select customer-name
  from borrower)

- Find all customers who have both a loan and an account at the bank.
  Solution:
  (select customer-name
  from depositor)
  intersect
  (select customer-name
  from borrower)

- Find all customers who have an account but no loan at the bank.
  Solution:
  (select customer-name
  from depositor)
  except
  (select customer-name
  from borrower)

4.1.7 Aggregate Functions

Aggregate functions are functions that take a collection (a set or multiset) of values as input and return a single value. SQL offers five built-in aggregate functions:

1. Average: avg
2. Minimum: min
3. Maximum: max
4. Total: sum
5. Count: count

Example:

- Find the average account balance at the Perryridge branch.
  Solution:
  select avg (balance)
  from account
  where branch-name = 'Perryridge'

- Find the average account balance at each branch.
  Solution:
  select branch-name, avg (balance)
  from account
  group by branch-name

- Find the number of depositors for each branch.
  Solution:
  select branch-name, count (distinct customer-name)
  from depositor, account
  where depositor.account-number = account.account-number
  group by branch-name

- Find the branches where the average account balance is more than $1200.
  Solution:
  select branch-name, avg (balance)
  from account
  group by branch-name
  having avg (balance) > 1200

- Find the average balance for each customer who lives in Harrison and has at least three accounts.
  Solution:
  select depositor.customer-name, avg (balance)
  from depositor, account, customer

where depositor.account-number = account.account-number and depositor.customer-name = customer.customer-name and customer-city = 'Harrison'
group by depositor.customer-name
having count (distinct depositor.account-number) ≥ 3

4.1.8 Nested Subqueries

- Find all customers who have both a loan and an account at the bank.
  Solution:
  select distinct customer-name
  from borrower
  where customer-name in (select customer-name from depositor)

- Find all customers who have both an account and a loan at the Perryridge branch.
  Solution:
  select distinct customer-name
  from borrower, loan
  where borrower.loan-number = loan.loan-number and
  branch-name = 'Perryridge' and (branch-name, customer-name)
  in (select branch-name, customer-name
  from depositor, account
  where depositor.account-number = account.account-number)

- Find all customers who do have a loan at the bank, but do not have an account at the bank.
  Solution:
  select distinct customer-name
  from borrower
  where customer-name not in (select customer-name from depositor)

- Find the names of all branches that have assets greater than those of at least one branch located in Brooklyn.
  Solution:
  select branch-name
  from branch
  where assets > some (select assets
  from branch
  where branch-city = 'Brooklyn')

- Find the names of all branches that have an asset value greater than that of each branch in Brooklyn.
  Solution:
  select branch-name
  from branch
  where assets > all (select assets

```
                from branch
                where branch-city = 'Brooklyn')
```

- Finds those branches for which the average balance is greater than or equal to all average balances.
  Solution:
  ```
  select branch-name
  from account
  group by branch-name
  having avg (balance) ≥ all (select avg (balance)
                              from account
                              group by branch-name)
  ```

4.1.9 Test for Empty Relations

The **exists** construct returns the value **true** if the argument subquery is nonempty.

Example: Find all customers who have both an account and a loan at the bank. **Solution:**

```
select customer-name
from borrower
where exists (select *
              from depositor
              where depositor.customer-name = borrower.customer-name)
```

We can test for the nonexistence of tuples in a subquery by using the not exists construct. We can use the not exists construct to simulate the set containment (that is, superset) operation: We can write "relation A contains relation B" as "not exists (B except A)."

Example: Find all customers who have an account at all the branches located in Brooklyn. **Solution:**

```
select distinct S.customer-name
from depositor as S
where not exists ((select branch-name
                   from branch
                   where branch-city = 'Brooklyn')
                   except
                   (select R.branch-name
                   from depositor as T, account as R
                   where T.account-number = R.account-number and
                   S.customer-name = T.customer-name))
```

4.1.10 Test for the Absence of Duplicate Tuples

SQL includes a feature for testing whether a subquery has any duplicate tuples in its result. The **unique** construct returns the value **true** if the argument subquery contains no duplicate tuples.

Example: Find all customers who have at most one account at the Perryridge branch.
Solution:

```
select T.customer-name
from depositor as T
where unique (select R.customer-name
              from account, depositor as R
              where T.customer-name = R.customer-name and
              R.account-number = account.account-number and
              account.branch-name = 'Perryridge')
```

We can test for the existence of duplicate tuples in a subquery by using the **not unique** construct.

Example: Find all customers who have at least two accounts at the Perryridge branch.
Solution:

```
select distinct T.customer-name
from depositor T
where not unique (select R.customer-name
                  from account, depositor as R
                  where T.customer-name = R.customer-name and
                  R.account-number = account.account-number and
                  account.branch-name = 'Perryridge')
```

4.1.11 Some other complex queries

- Find the average account balance of those branches where the average account balance is greater than $1200.
  Solution

```
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 > 120
```

- Find the maximum across all branches of the total balance at each branch.
  Solution

```
select max(tot-balance)
from (select branch-name, sum(balance)
from account
group by branch-name) as branch-total (branch-name, tot-balance)
```

4.1.12 Example

Consider the following database schemas and corresponding its database:-

Sailors(sid: integer, sname: string, rating: integer, age: real)
Boats(bid: integer, bname: string, color: string)
Reserves(sid: integer, bid: integer, day: date)

sid	*sname*	*rating*	*age*
22	Dustin	7	45.0
29	Brutus	1	33.0
31	Lubber	8	55.5
32	Andy	8	25.5
58	Rusty	10	35.0
64	Horatio	7	35.0
71	Zorba	10	16.0
74	Horatio	9	35.0
85	Art	3	25.5
95	Bob	3	63.5

Instance S_3 of sailors

sid	*bid*	*day*
22	101	10/10/98
22	102	10/10/98
22	103	10/8/98
22	104	10/7/98
31	102	11/10/98
31	103	11/6/98
31	104	11/12/98
64	101	9/5/98
64	102	9/8/98
74	103	9/8/98

Instance R_2 of Reserves

bid	*bname*	*color*
101	Interlake	blue
102	Interlake	red
103	Clipper	green
104	Marine	red

Instance B_1 of Boats Reserves

Write the following queries in SQL:-

1. Find the names of sailors who have reserved boat number 103.
   Solution:
   SELECT S.sname
   FROM Sailors S, Reserves R
   WHERE S.sid = R.sid AND R.bid=103

2. Find all sailors with a rating above 7.
   Solution:

SELECT S.sid, S.sname, S.rating, S.age
FROM Sailors AS S
WHERE S.rating > 7

3. Find the sids of sailors who have reserved a red boat.
   Solution:
   SELECT R.sid
   FROM Boats B, Reserves R
   WHERE B.bid = R.bid AND B.color = 'red'

4. Find the names of sailors who have reserved a red boat.
   Solution:
   SELECT S.sname
   FROM Sailors S, Reserves R, Boats B
   WHERE S.sid = R.sid AND R.bid = B.bid AND B.color = 'red'

5. Find the colors of boats reserved by Lubber.

   Solution:
   SELECT B.color
   FROM Sailors S, Reserves R, Boats B
   WHERE S.sid = R.sid AND R.bid = B.bid AND S.sname = 'Lubber'

6. Find the names of sailors who have reserved at least one boat.

   Solution:
   SELECT S.sname
   FROM Sailors S, Reserves R
   WHERE S.sid = R.sid

7. Compute increments for the ratings of persons who have sailed two different boats on the same day.
   Solution:
   SELECT S.sname, S.rating+1 AS rating
   FROM Sailors S, Reserves R1, Reserves R2
   WHERE S.sid = R1.sid AND S.sid = R2.sid
   AND R1.day = R2.day AND R1.bid <> R2.bid

8. Find the ages of sailors whose name begins and ends with B and has at least three characters.
   Solution:
   SELECT S.age
   FROM Sailors S

WHERE S.sname LIKE 'B_%B'

9. Find the names of sailors who have reserved a red or a green boat. **Solution:**
SELECT S.sname
FROM Sailors S, Reserves R, Boats B
WHERE S.sid = R.sid AND R.bid = B.bid
AND (B.color = 'red' OR B.color = 'green')

   This query can also be written as following:-
SELECT S.sname FROM Sailors S, Reserves R, Boats B WHERE S.sid = R.sid AND R.bid = B.bid AND B.color = 'red' UNION SELECT S2.sname FROM Sailors S2, Boats B2, Reserves R2 WHERE S2.sid = R2.sid AND R2.bid = B2.bid AND B2.color = 'green'

10. Find the names of sailors who have reserved both a red and a green boat. **Solution:**
SELECT S.sname
FROM Sailors S, Reserves R1, Boats B1, Reserves R2, Boats B2
WHERE S.sid = R1.sid AND R1.bid = B1.bid
AND S.sid = R2.sid AND R2.bid = B2.bid
AND B1.color='red' AND B2.color = 'green'

    This query can also be written as following:-
SELECT S.sname
FROM Sailors S, Reserves R, Boats B
WHERE S.sid = R.sid AND R.bid = B.bid AND B.color = 'red'
INTERSECT
SELECT S2.sname
FROM Sailors S2, Boats B2, Reserves R2
WHERE S2.sid = R2.sid AND R2.bid = B2.bid AND B2.color = 'green'

11. Find the sids of all sailors who have reserved red boats but not green boats.

    Solution:
SELECT S.sid
FROM Sailors S, Reserves R, Boats B
WHERE S.sid = R.sid AND R.bid = B.bid AND B.color = 'red'
EXCEPT
SELECT S2.sid
FROM Sailors S2, Reserves R2, Boats B2
WHERE S2.sid = R2.sid AND R2.bid = B2.bid AND B2.color = 'green'

12. Find the names of sailors who have not reserved a red boat.
    Solution:
SELECT S.sname
FROM Sailors S

```
WHERE S.sid NOT IN ( SELECT R.sid
                     FROM Reserves R
                     WHERE R.bid IN ( SELECT B.bid
                                      FROM Boats B
                                      WHERE B.color = 'red' ))
```

13. Find sailors whose rating is better than some sailor called Horatio.
    Solution:

```
SELECT S.sid
FROM Sailors S
WHERE S.rating > ANY ( SELECT S2.rating
                       FROM Sailors S2
                       WHERE S2.sname = 'Horatio' )
```

14. Find the sailors with the highest rating.
    Solution:

```
SELECT S.sid
FROM Sailors S
WHERE S.rating >= ALL ( SELECT S2.rating
                        FROM Sailors S2 )
```

15. Find the names of sailors who have reserved all boats.
    Solution:

```
SELECT S.sname
FROM Sailors S
WHERE NOT EXISTS (( SELECT B.bid
                    FROM Boats B )
                    EXCEPT
                    (SELECT R.bid
                    FROM Reserves R
                    WHERE R.sid = S.sid ))
```

    An alternative way to do this query without using EXCEPT follows:

```
SELECT S.sname
FROM Sailors S
WHERE NOT EXISTS ( SELECT B.bid
                   FROM Boats B
                   WHERE NOT EXISTS ( SELECT R.bid
                                      FROM Reserves R
                                      WHERE R.bid = B.bid
                                            AND R.sid = S.sid ))
```

16. Find the average age of all sailors.
    Solution:

```
SELECT AVG (S.age)
```

```
FROM Sailors S
```

17. Find the average age of sailors with a rating of 10.
    Solution:
    ```
    SELECT AVG (S.age)
    FROM Sailors S
    WHERE S.rating = 10
    ```

18. Find the name and age of the oldest sailor.
    Solution:
    ```
    SELECT S.sname, S.age
    FROM Sailors S
    WHERE S.age = ( SELECT MAX (S2.age)
                    FROM Sailors S2 )
    ```

19. Count the number of different sailor names.
    Solution:
    ```
    SELECT COUNT ( DISTINCT S.sname )
    FROM Sailors S
    ```

20. Find the names of sailors who are older than the oldest sailor with a rating of 10.
    Solution:
    ```
    SELECT S.sname
    FROM Sailors S
    WHERE S.age > ( SELECT MAX ( S2.age )
                    FROM Sailors S2
                    WHERE S2.rating = 10 )
    ```

    This query could alternatively be written as follows:
    ```
    SELECT S.sname
    FROM Sailors S
    WHERE S.age > ALL ( SELECT S2.age
                        FROM Sailors S2
                        WHERE S2.rating = 10 )
    ```

21. Find the age of the youngest sailor for each rating level.
    Solution:
    ```
    SELECT S.rating, MIN (S.age)
    FROM Sailors S
    GROUP BY S.rating
    ```

22. Find the age of the youngest sailor who is eligible to vote (i.e., is at least 18 years old) for each rating level with at least two such sailors.
    Solution:
    ```
    SELECT S.rating, MIN (S.age) AS minage
    ```

```
FROM Sailors S
WHERE S.age >= 18
GROUP BY S.rating
HAVING COUNT (*) > 1
```

23. For each red boat, find the number of reservations for this boat.
Solution:
```
SELECT B.bid, COUNT (*) AS sailorcount
FROM Boats B, Reserves R
WHERE R.bid = B.bid AND B.color = 'red'
GROUP BY B.bid
```

24. Find the average age of sailors for each rating level that has at least two sailors.
Solution:
```
SELECT S.rating, AVG (S.age) AS average
FROM Sailors S
GROUP BY S.rating
HAVING COUNT (*) > 1
```

25. Find the average age of sailors who are of voting age (i.e., at least 18 years old) for each rating level that has at least two sailors.
Solution:
```
SELECT S.rating, AVG ( S.age ) AS average
FROM Sailors S
WHERE S. age >= 18
GROUP BY S.rating
HAVING 1 < ( SELECT COUNT (*)
                FROM Sailors S2
                WHERE S.rating = S2.rating )
```

26. Find the average age of sailors who are of voting age (i.e., at least 18 years old) for each rating level that has at least two such sailors.
Solution:
```
SELECT S.rating, AVG ( S.age ) AS average
FROM Sailors S
WHERE S. age > 18
GROUP BY S.rating
HAVING 1 < ( SELECT COUNT (*)
                FROM Sailors S2
                WHERE S.rating = S2.rating AND S2.age >= 18 )
```

27. Find those ratings for which the average age of sailors is the minimum over all ratings.
Solution:
```
SELECT S.rating
FROM Sailors S
WHERE AVG (S.age) = ( SELECT MIN (AVG (S2.age))
                        FROM Sailors S2
                        GROUP BY S2.rating )
```

4.1.13 Cursor

We can declare a cursor on any relation or on any SQL query (because every query returns a set of rows). Once a cursor is declared, we can **open** it (which positions the cursor just before the first row); **fetch** the next row; **move** the cursor (to the next row, to the row after the next n, to the first row, or to the previous row, etc., by specifying additional parameters for the **FETCH** command); or **close** the cursor. Thus, a cursor essentially allows us to retrieve the rows in a table by positioning the cursor at a particular row and reading its contents.

4.1.14 TRIGGERS AND ACTIVE DATABASES

A trigger is a procedure that is automatically invoked by the DBMS in response to specified changes to the database, and is typically specified by the DBA. A database that has a set of associated triggers is called an active database. A trigger description contains three parts:

Event: A change to the database that activates the trigger.
Condition: A query or test that is run when the trigger is activated.
Action: A procedure that is executed when the trigger is activated and its condition is true.
A trigger can be thought of as a 'daemon' that monitors a database, and is executed when the database is modified in a way that matches the event specification. An insert, delete or update statement could activate a trigger, regardless of which user or application invoked the activating statement; users may not even be aware that a trigger was executed as a side effect of their program.
A condition in a trigger can be a true/false statement (e.g., all employee salaries are less than $100,000) or a query. A query is interpreted as true if the answer set is nonempty, and false if the query has no answers. If the condition part evaluates to true, the action associated with the trigger is executed.
A trigger action can examine the answers to the query in the condition part of the trigger, refer to old and new values of tuples modified by the statement activating the trigger, execute new queries, and make changes to the database. In fact, an action can even execute a series of data-definition commands (e.g., create new tables, change authorizations) and transaction-oriented commands (e.g., commit), or call host language procedures.

4.1.15 Exercise

1. Consider the following employee database:-

   employee (employee-name, street, city)
   works (employee-name, company-name, salary)
   company (company-name, city)
   manages (employee-name, manager-name)

   where the primary keys are underlined. Give an expression in SQL for each of the following queries.

(a) Find the names of all employees who work for First Bank Corporation.

(b) Find the names and cities of residence of all employees who work for First Bank Corporation.

(c) Find the names, street addresses, and cities of residence of all employees who work for First Bank Corporation and earn more than $10,000.

(d) Find all employees in the database who live in the same cities as the companies for which they work.

(e) Find all employees in the database who live in the same cities and on the same streets as do their managers.

(f) Find all employees in the database who do not work for First Bank Corporation.

(g) Find all employees in the database who earn more than each employee of Small Bank Corporation.

(h) Assume that the companies may be located in several cities. Find all companies located in every city in which Small Bank Corporation is located.

(i) Find all employees who earn more than the average salary of all employees of their company.

(j) Find the company that has the most employees.

(k) Find the company that has the smallest payroll.

(l) Find those companies whose employees earn a higher salary, on average, than the average salary at First Bank Corporation.

Solution:

(a) select employee-name
from works
where company-name = 'First Bank Corporation'

(b) select e.employee-name, city
from employee e, works w
where w.company-name = 'First Bank Corporation' and w.employee-name = e.employee-name

(c) select *
from employee
where employee-name in
(select employee-name
from works
where company-name = 'First Bank Corporation' and salary > 10000)

(d) select e.employee-name
from employee e, works w, company c
where e.employee-name = w.employee-name and e.city = c.city and
w.company-name = c.company-name

(e) select P.employee-name
from employee P, employee R, manages M
where P.employee-name = M.employee-name and M.manager-name = R.employee-name and P.street = R.street and P.city = R.city

(f) select employee-name
from works
where company-name ≠ 'First Bank Corporation'

(g) select employee-name
from works
where salary > all
(select salary
from works
where company-name = 'Small Bank Corporation')

(h) select S.company-name
from company S
where not exists ((select city
from company
where company-name = 'Small Bank Corporation')
except
(select city
from company T
where S.company-name = T.company-name))

(i) select employee-name
from works T
where salary > (select avg (salary)
from works S
where T.company-name = S.company-name)

(j) select company-name
from works
group by company-name
having count (distinct employee-name) >= all
(select count (distinct employee-name)
from works
group by company-name)

(k) select company-name
from works
group by company-name
having sum (salary) <= all (select sum (salary)
from works
group by company-name)

(l) select company-name
from works
group by company-name
having avg (salary) > (select avg (salary)
from works
where company-name = 'First Bank Corporation')

2. Consider the employee database of the previous questions. Give an expression in SQL for each of the following queries.

   (a) Modify the database so that Jones now lives in Newtown.

   (b) Give all employees of First Bank Corporation a 10 percent raise.

   (c) Give all managers of First Bank Corporation a 10 percent raise.

   (d) Give all managers of First Bank Corporation a 10 percent raise unless the salary becomes greater than $100,000; in such cases, give only a 3 percent raise.

   (e) Delete all tuples in the works relation for employees of Small Bank Corporation.

   Solution:

   (a)
   ```
   update employee
   set city = 'Newton'
   where person-name = 'Jones'
   ```

   (b)
   ```
   update works
   set salary = salary * 1.1
   where company-name = 'First Bank Corporation'
   ```

   (c)
   ```
   update works
   set salary = salary * 1.1
   where employee-name in (select manager-name from manages)
   and company-name = 'First Bank Corporation'
   ```

   (d)
   ```
   update works
   set salary = salary * 1.03
   where employee-name in (select manager-name from manages)
   and salary * 1.1 > 100000
   and company-name = 'First Bank Corporation'

   update works
   set salary = salary * 1.1
   where employee-name in (select manager-name from manages)
   and salary * 1.1 <= 100000
   and company-name = 'First Bank Corporation'
   ```

   (e)
   ```
   delete works
   where company-name = 'Small Bank Corporation'
   ```

3. Consider the following employee database:-

   person (driver-id, name, address)
   car (license, model, year)
   accident (report-number, date, location)
   owns (driver-id, license)

participated (driver-id, license, report-number, damage-amount)

where the primary keys are underlined. Construct the following SQL queries for this relational database.

(a) Find the total number of people who owned cars that were involved in accidents in 1989.

(b) Find the number of accidents in which the cars belonging to "John Smith" were involved.

(c) Add a new accident to the database; assume any values for required attributes.

(d) Delete the Mazda belonging to "John Smith".

(e) Update the damage amount for the car with license number "AABB2000" in the accident with report number "AR2197" to $3000.

Solution:

(a)
```
select count (distinct name)
from accident, participated, person
where accident.report-number = participated.report-number
and participated.driver-id = person.driver-id
and date between date '1989-00-00' and date '1989-12-31'
```

(b)
```
select count (distinct *)
from accident
where exists
        (select *
        from participated, person
        where participated.driver-id = person.driver-id
        and person.name = 'John Smith'
        and accident.report-number = participated.report-number)
```

(c) We assume the driver was "Jones," although it could be someone else. Also, we assume "Jones" owns one Toyota. First we must find the license of the given car. Then the participated and accident relations must be updated in order to both record the accident and tie it to the given car. We assume values "Berkeley" for location, '2001-09-01' for date and date, 4007 for reportnumber and 3000 for damage amount.

```
insert into accident
values (4007, '2001-09-01', 'Berkeley')

insert into participated
select o.driver-id, c.license, 4007, 3000
from person p, owns o, car c
where p.name = 'Jones' and p.driver-id = o.driver-id and
o.license = c.license and c.model = 'Toyota'
```

(d) delete car
where model = 'Mazda' and license in
(select license
from person p, owns o
where p.name = 'John Smith' and
p.driver-id = o.driver-id)

(e) update participated
set damage-amount = 3000
where report-number = "AR2197" and driver-id in
(select driver-id
from owns
where license = "AABB2000")

4. Consider the following relations:

Student(snum: integer, sname: string, major: string, level: string, age: integer)
Class(cname: string, meets at: time, room: string, fid: integer)
Enrolled(snum: integer, cname: string)
Faculty(fid: integer, fname: string, deptid: integer)

Write the following queries in SQL. No duplicates should be printed in any of the answers.

(a) Find the names of all Juniors (Level = JR) who are enrolled in a class taught by I. Teach.

(b) Find the age of the oldest student who is either a History major or is enrolled in a course taught by I. Teach.

(c) Find the names of all classes that either meet in room R128 or have five or more students enrolled.

(d) Find the names of all students who are enrolled in two classes that meet at the same time.

(e) Find the names of faculty members who teach in every room in which some class is taught.

(f) Find the names of faculty members for whom the combined enrollment of the courses that they teach is less than five.

(g) Print the Level and the average age of students for that Level, for each Level.

(h) Print the Level and the average age of students for that Level, for all Levels except JR.

(i) Find the names of students who are enrolled in the maximum number of classes.

(j) Find the names of students who are not enrolled in any class.

(k) For each age value that appears in Students, find the level value that appears most often.

For example, if there are more FR level students aged 18 than SR, JR, or SO students aged 18, you should print the pair (18, FR).

Solution:

(a)
```
SELECT DISTINCT S.Sname
FROM Student S, Class C, Enrolled E, Faculty F
WHERE S.snum = E.snum AND E.cname = C.name AND C.fid = F.fid AND
F.fname = 'I.Teach' AND S.level = 'JR'
```

(b)
```
SELECT MAX(S.age)
FROM Student S
WHERE (S.major = 'History')
OR S.snum IN (SELECT E.snum
FROM Class C, Enrolled E, Faculty F
WHERE E.cname = C.name AND C.fid = F.fid
AND F.fname = 'I.Teach' )
```

(c)
```
SELECT C.name
FROM Class C
WHERE C.room = 'R128'
OR C.name IN (SELECT E.cname
FROM Enrolled E
GROUP BY E.cname
HAVING COUNT (*) >= 5)
```

(d)
```
SELECT DISTINCT S.sname
FROM Student S
WHERE S.snum IN (SELECT E1.snum
FROM Enrolled E1, Enrolled E2, Class C1, Class C2
WHERE E1.snum = E2.snum AND E1.cname <> E2.cname
AND E1.cname = C1.name
AND E2.cname = C2.name AND C1.meets at = C2.meets at)
```

(e)
```
SELECT DISTINCT F.fname
FROM Faculty F
WHERE NOT EXISTS (( SELECT *
FROM Class C )
EXCEPT
(SELECTC1.room
FROM Class C1
WHERE C1.fid = F.fid ))
```

(f)
```
SELECT DISTINCT F.fname
FROM Faculty F
WHERE 5 > (SELECT COUNT (E.snum)
FROM Class C, Enrolled E
WHERE C.name = E.cname
```

AND C.fid = F.fid)

(g) SELECT S.level, AVG(S.age)
FROM Student S
GROUP BY S.level

(h) SELECT S.level, AVG(S.age)
FROM Student S
WHERE S.level ¡¿ 'JR'
GROUP BY S.level

(i) SELECT F.fname, COUNT(*) AS CourseCount
FROM Faculty F, Class C
WHERE F.fid = C.fid
GROUP BY F.fid, F.fname
HAVING EVERY (C.room = 'R128')

(j) SELECT DISTINCT S.sname
FROM Student S
WHERE S.snum IN (SELECT E.snum
FROM Enrolled E
GROUP BY E.snum
HAVING COUNT (*) >= ALL (SELECT COUNT (*)
FROM Enrolled E2
GROUP BY E2.snum))

(k) SELECT DISTINCT S.sname
FROM Student S
WHERE S.snum NOT IN (SELECT E.snum
FROM Enrolled E)

(l) SELECT S.age, S.level
FROM Student S
GROUP BY S.age, S.level,
HAVING S.level IN (SELECT S1.level
FROM Student S1
WHERE S1.age = S.age
GROUP BY S1.level, S1.age
HAVING COUNT (*) >= ALL (SELECT COUNT (*)
FROM Student S2
WHERE s1.age = S2.age
GROUP BY S2.level, S2.age))

5. Consider the following schema:

Suppliers(sid: integer, sname: string, address: string)

Parts(pid: integer, pname: string, color: string)
Catalog(sid: integer, pid: integer, cost: real)

The Catalog relation lists the prices charged for parts by Suppliers. Write the following queries in SQL:

(a) Find the pnames of parts for which there is some supplier.

(b) Find the snames of suppliers who supply every part.

(c) Find the snames of suppliers who supply every red part.

(d) Find the pnames of parts supplied by Acme Widget Suppliers and by no one else.

(e) Find the sids of suppliers who charge more for some part than the average cost of that part (averaged over all the suppliers who supply that part).

(f) For each part, find the sname of the supplier who charges the most for that part.

(g) Find the sids of suppliers who supply only red parts.

(h) Find the sids of suppliers who supply a red part and a green part.

(i) Find the sids of suppliers who supply a red part or a green part.

Solution:

(a)
```
SELECT pname
FROM Parts, Catalog
WHERE Parts.pid = Catalog.pid
```

(b)
```
SELECT Sname
FROM Suppliers
WHERE NOT EXISTS ( SELECT pid
                   FROM Part)
                   EXCEPT
                   ( SELECT pid
                   FROM Catalog
                   WHERE Suppliers.sid = Catalog.sid)
```

(c)
```
SELECT Sname
FROM Suppliers
WHERE NOT EXISTS ( SELECT pid
                   FROM Part
                   WHERE color = 'red')
                   EXCEPT
                   ( SELECT pid
                   FROM Catalog
                   WHERE Suppliers.sid = Catalog.sid)
```

(d)
```
SELECT pname FROM Parts, Catalog, Suppliers
WHERE Parts.pid = Catalog.pid AND Catalog.sid = Suppliers.sid AND
```

```
sname = 'Acme Widget' AND
pid NOT IN (SELECT pid
            FROM Catalog, Suppliers
            WHERE Catalog.sid = Suppliers.sid AND
                  sname <> 'Acme Widget')
```

(e)
```
SELECT sid
FROM Catalog
WHERE cost > (SELECT avg(cost)
              FROM Catalog as T
              WHERE Catalog.pid = T.pid)
```

(f)
```
SELECT pid, sname
FROM Suppliers, Catalog
WHERE Suppliers.sid = Catalog.sid AND cost = (SELECT max(cost)
                                              FROM Catalog as T
                                              WHERE Catalog.pid = T.pid)
```

(g)
```
SELECT sid
FROM Catalog, Parts
WHERE Catalog.pid = Parts.pid AND
      NOT EXISTS ((SELECT pid
                   FROM Parts as P
                   WHERE Parts.pid = P.pid)
                   EXCEPT
                   (SELECT pid
                   FROM Parts
                   WHERE color = 'red'))
```

(h)
```
(SELECT sid
FROM Catalog, Parts
WHERE Catalog.pid = Parts.pid AND color = 'red')
INTERSECT
(SELECT sid
FROM Catalog, Parts
WHERE Catalog.pid = Parts.pid AND color = 'green')
```

(i)
```
(SELECT sid
FROM Catalog, Parts
WHERE Catalog.pid = Parts.pid AND color = 'red')
UNION
(SELECT sid
FROM Catalog, Parts
WHERE Catalog.pid = Parts.pid AND color = 'green')
```

6. The following relations keep track of airline flight information:

Flights(flno: integer, from: string, to: string, distance: integer, departs: time, arrives: time, price: integer)
Aircraft(aid: integer, aname: string, cruisingrange: integer)
Certified(eid: integer, aid: integer)

Employees(eid: integer, ename: string, salary: integer)

Note that the Employees relation describes pilots and other kinds of employees as well; every pilot is certified for some aircraft, and only pilots are certified to fly. Write each of the following queries in SQL.

(a) Find the names of aircraft such that all pilots certified to operate them earn more than 80,000.

(b) For each pilot who is certified for more than three aircraft, find the eid and the maximum cruisingrange of the aircraft that he (or she) is certified for.

(c) Find the names of pilots whose salary is less than the price of the cheapest route from Los Angeles to Honolulu.

(d) For all aircraft with cruisingrange over 1,000 miles, find the name of the aircraft and the average salary of all pilots certified for this aircraft.

(e) Find the names of pilots certified for some Boeing aircraft.

(f) Find the aids of all aircraft that can be used on routes from Los Angeles to Chicago.

(g) Identify the flights that can be piloted by every pilot who makes more than$100,000. (Hint: The pilot must be certified for at least one plane with a sufficiently large cruisingrange.)

(h) Print the enames of pilots who can operate planes with cruisingrange greater than 3,000 miles, but are not certified on any Boeing aircraft.

(i) A customer wants to travel from Madison to New York with no more than two changes of flight. List the choice of departure times from Madison if the customer wants to arrive in New York by 6 p.m.

(j) Compute the difference between the average salary of a pilot and the average salary of all employees (including pilots).

(k) Print the name and salary of every nonpilot whose salary is more than the average salary for pilots.

Solution:

(a)
```
SELECT DISTINCT A.aname
FROM Aircraft A
WHERE A.Aid IN (SELECT C.aid
                FROM Certified C, Employees E
                WHERE C.eid = E.eid AND
                NOT EXISTS ( SELECT *
                             FROM Employees E1
                             WHERE E1.eid = E.eid AND E1.salary < 80000))
```

(b)
```
SELECT C.eid, MAX (A.cruisingrange)
FROM Certified C, Aircraft A
WHERE C.aid = A.aid
GROUP BY C.eid
HAVING COUNT (*) > 3
```

(c)
```
SELECT DISTINCT E.ename
FROM Employees E
WHERE E.salary < ( SELECT MIN (F.price)
                   FROM Flights F
                   WHERE F.from = 'Los Angeles' AND F.to = 'Honolulu' )
```

(d)
```
SELECT Temp.name, Temp.AvgSalary
FROM ( SELECT A.aid, A.aname AS name, AVG (E.salary) AS AvgSalary
       FROM Aircraft A, Certified C, Employees E
       WHERE A.aid = C.aid AND C.eid = E.eid
             AND A.cruisingrange > 1000
       GROUP BY A.aid, A.aname ) AS Temp
```

(e)
```
SELECT DISTINCT E.ename
FROM Employees E, Certified C, Aircraft A
WHERE E.eid = C.eid AND C.aid = A.aid AND A.aname LIKE 'Boeing
```

(f)
```
SELECT A.aid
FROM Aircraft A
WHERE A.cruisingrange > ( SELECT MIN (F.distance)
                         FROM Flights F
                         WHERE F.from = 'Los Angeles' AND F.to = 'Chicago'
)
```

(g)
```
SELECT DISTINCT F.from, F.to
FROM Flights F
WHERE NOT EXISTS ( SELECT *
                   FROM Employees E
                   WHERE E.salary > 100000 AND
                   NOT EXISTS (SELECT *
                               FROM Aircraft A, Certified C
                               WHERE A.cruisingrange > F.distance
                               AND E.eid = C.eid AND A.aid = C.aid))
```

(h)
```
SELECT DISTINCT E.ename
FROM Employees E
WHERE E.eid IN ( SELECT C.eid
                 FROM Certified C
                 WHERE EXISTS ( SELECT A.aid
                                FROM Aircraft A
                                WHERE A.aid = C.aid AND
                                      A.cruisingrange > 3000 )
                       AND NOT EXISTS ( SELECT A1.aid
                                        FROM Aircraft A1
                                        WHERE A1.aid = C.aid
                                        AND A1.aname LIKE 'Boeing%'))
```

(i)
```
SELECT F.departs
FROM Flights F
WHERE F.flno IN ( ( SELECT F0.flno
                    FROM Flights F0
                    WHERE F0.from = 'Madison' AND F0.to = 'New York'
                          AND F0.arrives < '18:00' )
                    UNION
                  ( SELECT F0.flno
                  FROM Flights F0, Flights F1
                  WHERE F0.from = 'Madison' AND F0.to <> 'New York'
                       AND F0.to = F1.from AND F1.to = 'New York'
                      AND F1.departs > F0.arrives AND F1.arrives < '18:00'
)
                    UNION
                    ( SELECT F0.flno
                    FROM Flights F0, Flights F1, Flights F2
                    WHERE F0.from = 'Madison' AND F0.to = F1.from
                          AND F1.to = F2.from AND F2.to = 'New York'
                          AND F0.to <> 'New York'
                          AND F1.to <> 'New York'
                          AND F1.departs > F0.arrives
                          AND F2.departs> F1.arrives
                          AND F2.arrives < '18:00' ))
```

(j)
```
SELECT Temp1.avg - Temp2.avg
FROM (SELECT AVG (E.salary) AS avg
        FROM Employees E
        WHERE E.eid IN (SELECT DISTINCT C.eid
                        FROM Certified C )) AS Temp1,
        (SELECT AVG (E1.salary) AS avg
        FROM Employees E1 ) AS Temp2
```

(k)
```
SELECT E.ename, E.salary
FROM Employees E
WHERE E.eid NOT IN ( SELECT DISTINCT C.eid
                     FROM Certified C )
      AND E.salary > ( SELECT AVG (E1.salary)
                       FROM Employees E1
                       WHERE E1.eid IN ( SELECT DISTINCT C1.eid
                                         FROM Certified C1 ) )
```

7. Consider the following relational schema. An employee can work in more than one department; the pct time field of the Works relation shows the percentage of time that a given employee works in a given department.

   Emp(eid: integer, ename: string, age: integer, salary: real)
   Works(eid: integer, did: integer, pct time: integer)

Dept(did: integer, budget: real, managerid: integer)

Write the following queries in SQL:

(a) Print the names and ages of each employee who works in both the Hardware department and the Software department.

(b) For each department with more than 20 full-time-equivalent employees (i.e., where the part-time and full-time employees add up to at least that many full-time employees), print the did together with the number of employees that work in that department.

(c) Print the name of each employee whose salary exceeds the budget of all of the departments that he or she works in.

(d) Find the managerids of managers who manage only departments with budgets greater than $1,000,000.

(e) Find the enames of managers who manage the departments with the largest budget.

(f) If a manager manages more than one department, he or she controls the sum of all them budgets for those departments. Find the managerids of managers who control more than $5,000,000.

(g) Find the managerids of managers who control the largest amount.

Solution:

(a)

(b)

(c)

(d)

(e)

(f)

(g)

(h)

Chapter 5

Relational Database Design

5.1 Functional dependency

Consider a relation schema R, and let $\alpha \subseteq R$ and $\beta \subseteq R$. The functional dependency $\alpha \rightarrow \beta$ holds on relation schema R if, in any legal relation r(R), for all pairs of tuples t_1 and t_2 in r such that $t_1[\alpha] = t_2[\alpha]$, then $t_1[\beta] = t_2[\beta]$ must also satisfy with in r(R).

Super key: A subset α of a relation schema R is said to be super key of R if $\alpha \rightarrow R$ holds.

Candidate key: A subset α of a relation schema R is said to be super key of R if
(1) α should be super key of R i.e. $\alpha \rightarrow R$.
(2) There should not exist any proper subset K of α such that K $\rightarrow$ R.

Example: Consider the following relation :-

A	B	C	D
a_1	b_1	c_1	d_1
a_1	b_2	c_1	d_2
a_2	b_2	c_2	d_2
a_2	b_2	c_2	d_3
a_3	b_3	c_2	d_4

Find out which functional dependencies are satisfied.

Solution: Observe that A$\rightarrow$C is satisfied. There are two tuples that have an A value of a_1. These tuples have the same C value namely, c_1. Similarly, the two tuples with an A value of a_2 have the same C value, c_2. There are no other pairs of distinct tuples that have the same A value. The functional dependency C $\rightarrow$ A is not satisfied, however. To see that it is not, consider the tuples $t_1 = (a_2, b_3, c_2, d_3)$ and $t_2 = (a_3, b_3, c_2, d_4)$. These two tuples have the same C values, c_2, but they have different A values, a_2 and a_3, respectively. Thus, we have found a pair of tuples t_1 and t_2 such that $t_1[C] = t_2[C]$, but $t_1[A] \neq t_2[A]$.
Some other functional dependencies which satisfied are the following:-

AB → C, D → B, BC → A, CD → A, CD → B, AD → B, AD → C.

5.1.1 Trivial functional dependency

A functional dependency $\alpha \rightarrow \beta$ is said to be trivial if $\beta \subseteq \alpha$.

Some trivial functional dependencies are the following:- ABC → C, CD → C, A → A.

5.1.2 Closure of a Set of Functional Dependencies

Consider F is a set of functional dependencies defined on relation schema R.
Closure of F is the set of all the functional dependencies which are logically implied(or derived) from F. It is denoted by F^+.

5.1.3 Armstrong's axioms

Following three rules are said to be Armstrong's axioms.

- **Reflexivity rule:** If α is a set of attributes and $\beta \subseteq \alpha$, then $\alpha \rightarrow \beta$ holds.
- **Augmentation rule:** If $\alpha \rightarrow \beta$ holds and γ is a set of attributes, then $\gamma\alpha \rightarrow \gamma\beta$ holds.
- **Transitivity rule:** If $\alpha \rightarrow \beta$ holds and $\beta \rightarrow \gamma$ holds, then $\alpha \rightarrow \gamma$ holds.

Some additional rules are the following:-

- **Union rule:** If $\alpha \rightarrow \beta$ and $\alpha \rightarrow \gamma$ holds, then $\alpha \rightarrow \beta\gamma$ holds.
- **Decomposition rule:** If $\alpha \rightarrow \beta\gamma$ holds then $\alpha \rightarrow \beta$ and $\alpha \rightarrow \gamma$ holds.
- **Pseudo transitivity rule:** If $\alpha \rightarrow \beta$ holds and $\gamma\beta \rightarrow \delta$ holds, then $\gamma\alpha \rightarrow \delta$ holds.

Example: Consider relation schema R = (A, B, C, G, H, I) and the set F of functional dependencies A → B, A → C, CG → H, CG → I, B → H. We list several members of F+ here:

- A → H. Since A → B and B → H hold, we apply the transitivity rule.
- CG → HI . Since CG → H and CG → I , the union rule implies that CG → HI.
- AG → I. Since A → C and CG → I, the pseudo transitivity rule implies that AG → I holds.

Algorithm to compute F^+ using Armstrong's axioms
In this algorithm, the input will be F and R. It is computed by following algorithm:-
Note: The left-hand and right-hand sides of a functional dependency are both subsets of R. Since a set of size n has 2^n subsets, therefore there are a total of $2 \times 2^n = 2^{n+1}$ possible functional dependencies, where n is the number of attributes in R.

Input: F and R
Output: F^+
$F^+ \leftarrow F$
repeat
  for *for each functional dependency f in* F^+ **do**
    apply reflexivity and augmentation rules on f
    add the resulting functional dependencies to F^+
  end
  for *each pair of functional dependencies* f_1 *and* f_2 *in* F^+ **do**
    if f_1 *and* f_2 *can be combined using transitivity rule* **then**
      add the resulting functional dependency to F^+
    end
  end
until F^+ *does not change any further*;

Algorithm 1: A procedure to compute F+

5.1.4 Closure of attribute sets

Consider relation schema R and a set of functional dependencies F. Let $\alpha \subseteq$ R.
The closure of α is the set of all the attributes of R which are logically determined by α under a set F. It is denoted by α^+.
The closure of α is computed by following algorithm:-

Input: α and F
Output: α^+ = result
$result \leftarrow \alpha$
while *changes to result* **do**
  for *each functional dependency* $\beta \rightarrow \gamma$ *in* F **do**
    if $\beta \subseteq result$ **then**
      $result \leftarrow result \cup \gamma$
    end
  end
end

Algorithm 2: An algorithm to compute α^+, the closure of α under F

Example: Consider relation schema R = (A, B, C, G, H, I) and the set F of functional dependencies A $\rightarrow$ B, A $\rightarrow$ C, CG $\rightarrow$ H, CG $\rightarrow$ I, B $\rightarrow$ H. Compute the closure of {A,G}, {C,G} and {A}.
Solution:

$\{A, G\}^+ = \{A,G\}$
$= \{A,B,C,G\}$
$= \{A,B,C,G,H,I\}$

Therefore, $\{A, G\}^+ = \{A,B,C,G,H,I\}$

$\{C, G\}^+ = \{C,G\}$
$= \{C,G,H,I\}$

Therefore, $\{C, G\}^+ = \{C,G,H,I\}$

$\{A\}^+ = \{A\}$
$= \{A,B,C\}$
$= \{A,B,C,H\}$
Therefore, $\{A\}^+ = \{A,B,C,H\}$
Uses or applications of attribute closure
There are several uses of the attribute closure:

- To test if α is a superkey, we compute α^+, and check if α^+ contains all attributes of R.
- We can check if a functional dependency $\alpha \rightarrow \beta$ holds by checking if $\beta \subseteq \alpha^+$.
- It gives us an alternative way to compute F^+: For each $\gamma \subseteq R$, we find the closure γ^+, and for each $S \subseteq \gamma^+$, we output a functional dependency $\gamma \rightarrow S$.

5.1.5 Canonical Cover

Before defining canonical cover, first we are going to define some concepts related with it.
Extraneous attribute: Consider a set F of functional dependencies and the functional dependency $\alpha \rightarrow \beta$ in F.

- Attribute A is said to be extraneous in α if A $\in \alpha$, and F logically implies (F - $\{\alpha \rightarrow \beta\}$) $\cup$ $\{(\alpha - A) \rightarrow \beta\}$.
- Attribute A is said to be extraneous in β if A $\in \beta$, and the set of functional dependencies (F - $\{\alpha \rightarrow \beta\}$) $\cup$ $\{\alpha \rightarrow (\beta - A)\}$ logically implies F.

Example: Consider F = { AB $\rightarrow$ C and A $\rightarrow$ C}. Find extraneous attributes in F.
Solution: Consider the functional dependency AB $\rightarrow$ C. In this dependency, right hand side contains single attribute, therefore right hand side has no extraneous attribute.
Now, consider left hand side. Now, we check A is extraneous attribute or not.
Eliminate A from FD, AB $\rightarrow$ C. We get B $\rightarrow$ C. Clearly, B $\rightarrow$ C can not be derived from F, therefore A is not extraneous attribute.
Now, we check B is extraneous attribute or not.
Eliminate B from FD, AB $\rightarrow$ C. We get A $\rightarrow$ C. Clearly, A $\rightarrow$ C is derived from F, therefore B is an extraneous attribute.
Consider the functional dependency A $\rightarrow$ C. In this dependency, left hand and right hand side contains single attribute, therefore this FD has no extraneous attribute.

Example: Consider F = {AB $\rightarrow$ CD and A $\rightarrow$ C}. Find extraneous attributes in F.
Solution: Consider the functional dependency AB $\rightarrow$ CD. In this FD, both sides may contain extraneous attributes.
Now, consider left hand side. Now, we check A is extraneous attribute or not.
Eliminate A from FD, AB $\rightarrow$ CD. We get B $\rightarrow$ CD. Clearly, B $\rightarrow$ CD can not be derived from F, therefore A is not extraneous attribute.
Now, we check B is extraneous attribute or not.
Eliminate B from FD, AB $\rightarrow$ CD. We get A $\rightarrow$ CD. Clearly, A $\rightarrow$ CD can not be derived from F, therefore B is not extraneous attribute.

Now, we check C is extraneous attribute or not.
Eliminate C from FD, AB → CD. We get AB → D. Clearly, Set F' = {AB → D, A → C} derives set F, therefore C is an extraneous attribute.
Now, we check D is extraneous attribute or not.
Eliminate D from FD, AB → CD. We get AB → C. Clearly, Set F' = {AB → C, A → C} can not derive set F, therefore D is not an extraneous attribute.
Consider the functional dependency A → C. In this dependency, left hand and right hand side contains single attribute, therefore this FD has no extraneous attribute.

Redundant functional dependency A functional dependency $\alpha \rightarrow \beta$ in F is said to be redundant if after eliminating $\alpha \rightarrow \beta$ from F, we get a set of functional dependency F' equivalent to F. That is, $F^+ = F'^+$.
Example: Consider F = {A → B, B → C, and A → C}. In this set F, FD A → C} is redundant because it is derived from A → B and B → C using transitivity rule.

Canonical Cover:
Canonical cover is defined for a set F of functional dependencies.
Canonical cover of F is the minimal set of functional dependencies equivalent to F that is canonical cover is a set of functional dependencies equivalent to F which does not contain any extraneous attribute and redundant FD. It is denoted by F_c.
A canonical cover for a set of functional dependencies F can be computed by following algorithm.

Input: F
Output: F_c
$F_c \leftarrow F$
repeat
  Use the union rule to replace any dependencies in F_c of the form $\alpha_1 \rightarrow \beta_1$ and $\alpha_1 \rightarrow \beta_2$ with $\alpha_1 \rightarrow \beta_1\beta_2$.
  Find a functional dependency $\alpha \rightarrow \beta$ in F_c with an extraneous attribute either in α or in β.
  /* Note: the test for extraneous attributes is done using F_c, not F */
  If an extraneous attribute is found, delete it from $\alpha \rightarrow \beta$.
until F_c *does not change any further*;

Algorithm 3: Computing canonical cover

Example: Consider the following set F of functional dependencies on relation schema R = (A,B,C):
A → BC
B → C
A → B
AB → C
Compute the canonical cover for F.
Solution:

- There are two functional dependencies with the same set of attributes on the left side of the arrow:

A → BC, A → B
We combine these functional dependencies using union rule into A → BC.

- A is extraneous in AB → C because F logically implies (F - {AB → C}) ∪ {B → C}. This assertion is true because B → C is already in our set of functional dependencies.

- C is extraneous in A → BC, since A → BC is logically implied by A → B and B → C.

Thus, canonical cover of F is
F_c = {A → B, B → C}.

Note: A canonical cover might not be unique.

Example: Consider the following set F of functional dependencies on relation schema R = (A,B,C):
A → BC
B → AC
C → AB
Compute the canonical cover for F.
Solution: If we apply the extraneity test to A → BC, we find that both B and C are extraneous under F. However, it is incorrect to delete both. The algorithm for finding the canonical cover picks one of the two, and deletes it. Then,

1. If C is deleted, we get the set F' = {A → B, B → AC, and C → AB}. Now, B is not extraneous in the right hand side of A → B under F'. Continuing the algorithm, we find A and B are extraneous in the right hand side of C → AB, leading to two canonical covers
   F_c = {A → B, B → C, and C → A}, and
   F_c = {A → B, B → AC, and C → B}.

2. If B is deleted, we get the set F' = {A → C, B → AC, and C → AB}. This case is symmetrical to the previous case, leading to the canonical covers
   F_c = {A → C, C → B, and B → A}, and
   F_c = {A → C, B → C, and C → AB}.

5.1.6 Decomposition

Let R be a relation schema. A set of relation schemas $\{R_1, R_2, ..., R_n\}$ is a decomposition of R if

$$R = R_1 \cup R_2 \cup ..... \cup R_n$$

That is, $\{R_1, R_2, ..., R_n\}$ is a decomposition of R if, for i = 1, 2, . . . , n, each R_i is a subset of R, and every attribute in R appears in at least one R_i.
Let r be a relation on schema R, and let $r_i = \Pi_{R_i}(r)$ for i = 1, 2, . . . , n. That is, $\{r_1, r_2, ..., r_n\}$ is the database that results from decomposing R into $\{R_1, R_2, ..., R_n\}$. It is always the case that

$$r \subseteq r_1 \bowtie r_2 \bowtie \bowtie r_n.$$

A decomposition $\{R_1, R_2, ..., R_n\}$ of R is a lossless-join decomposition if, for all relations

r on schema R,

$$r = \Pi_{R_1}(r) \bowtie \Pi_{R_2}(r) \bowtie ..... \bowtie \Pi_{R_n}(r).....(1)$$

If equation (1) is not satisfied, then this decomposition is said to be lossy decomposition.

5.1.7 Desirable Properties of Decomposition

Lossless-Join Decomposition
Let R be a relation schema, and let F be a set of functional dependencies on R. Let R_1 and R_2 form a decomposition of R. This decomposition is a lossless-join decomposition of R if at least one of the following functional dependencies is in F+:

- $R_1 \cap R_2 \rightarrow R_1$
- $R_1 \cap R_2 \rightarrow R_2$

In other words, if $R_1 \cap R_2$ forms a superkey of either R_1 or R_2, then the decomposition of R is a lossless-join decomposition.
Eample: Consider the following schema R and F.
R = (branch-name, branch-city, assets, customer-name, loan-number, amount)
F is the following :-
branch-name → branch-city assets
loan-number → amount branch-name
R is decomposed into following relation schemas:-
Branch-schema = (branch-name, branch-city, assets)
Loan-info-schema = (branch-name, customer-name, loan-number, amount)
Is this decomposition lossless or lossy?
Solution:
Branch-schema ∩ Loan-info-schema = {branch-name}
Clearly, branch-name is a super key of Branch-schema. Therefore this decomposition is lossless.

Dependency Preservation
Let F be a set of functional dependencies on a schema R, and let $R_1, R_2, ..., R_n$ be a decomposition of R. Let $F_1, F_2, ..., F_n$ is the set of dependencies corresponding to $R_1, R_2, ..., R_n$.
Let F' = $F_1 \cup F_2 \cup ... \cup F_n$
If F' = F, then the decomposition will be functionally dependency preserve.
If F' ≠ F , then the decomposition may or may not be functionally dependency preserve. In this case, compute F^+ and F'^+. If $F^+ = F'^+$, then the decomposition will be functionally dependency preserve otherwise not.

Eample: Consider the following schema R and F.
R = (branch-name, branch-city, assets, customer-name, loan-number, amount)
F is the following :-
branch-name → branch-city assets
loan-number → amount branch-name
R is decomposed into following relation schemas:-
Branch-schema = (branch-name, branch-city, assets)
Loan-info-schema = (branch-name, customer-name, loan-number, amount)
Is this decomposition functionally dependency preserve?

Solution:
Let F_1 and F_2 are the set of functional dependencies corresponding to Branch-schema and Loan-info-schema respectively. Therefore
F_1 = { branch-name → branch-city assets} and
F_2 = { loan-number → amount branch-name}
Now, F' = $F_1 \cup F_2$ = {branch-name → branch-city assets, loan-number → amount branch-name}
Clearly, F' = F. Therefore this decomposition is functionally dependency preserve.

5.2 Normalization

Normalization is a database design technique that reduces data redundancy and eliminates undesirable characteristics like Insertion, Update and Deletion Anomalies. Normalization rules divides larger tables into smaller tables and links them using relationships. The purpose of Normalization is to eliminate redundant (repetitive) data and ensure data is stored logically.

5.2.1 Anomalies in DBMS

Anomalies are problems that can occur in poorly planned, unnormalized databases where all the data is stored in one table (a flatfile database). There are three types of anomalies that occur when the database is not normalized. These are – **insertion, update and deletion** anomaly. Let's take an example to understand this. Consider a relation Emp-Dept.

E#	Ename	Address	D#	Dname	Dmgr#
123456789	Akhilesh	Ghaziabad	5	Research	333445555
333445555	Ajay	Kanpur	5	Research	333445555
999887777	Shreya	Lucknow	4	Administration	987654321
987654321	Sanjay	Mirjapur	4	Administration	987654321
666884444	Om Prakash	Lucknow	5	Research	333445555
453453453	Manish	Delhi	5	Research	333445555
987987987	Ishani	Prayagraj	4	Administration	987654321
888665555	Garvita	Prayagraj	1	Headquarters	888665555

Table 5.1: Emp-Dept

Insertion anomaly: Let us assume that a new department has been started by the organization but initially there is no employee appointed for that department, then the tuple for this department cannot be inserted into this table as the E# will have NULL, which is not allowed as E# is primary key.
This kind of a problem in the relation where some tuple cannot be inserted is known as insertion anomaly.

Deletion anomaly:
Now consider there is only one employee in some department and that employee leaves the organization, then the tuple of that employee has to be deleted from the table, but

in addition to that the information about the department also will get deleted.
This kind of a problem in the relation where deletion of some tuples can lead to loss of some other data not intended to be removed is known as deletion anomaly.

Modification/update anomaly:
Suppose the manager of a department has changed, this requires that the Dmgr# in all the tuples corresponding to that department must be changed to reflect the new status. If we fail to update all the tuples of the given department, then two different records of employee working in the same department might show different Dmgr# leading to inconsistency in the database.
This is known as modification/update anomaly.

5.2.2 Normalization

To overcome these anomalies we need to normalize the data. In the next section we will discuss different types of normal forms. These normal forms are:-

1. First normal form(1NF)
2. Second normal form(2NF)
3. Third normal form(3NF)
4. Boyce-Codd normal form(BCNF)
5. Fourth normal form(4NF)
6. Fifth normal form(5NF)

First normal form(1NF):
As per the rule of first normal form, an attribute (column) of a table cannot hold multiple values. It should hold only atomic values.
Consider the following table:-
This table is not in 1NF as the rule says "each attribute of a table must have atomic

emp_id	emp_name	emp_address	emp_mobile
101	Herschel	New Delhi	8912312390
102	Jon	Kanpur	8812121212 , 9900012222
103	Ron	Chennai	7778881212
104	Lester	Bangalore	9990000123 , 8123450987

Table 5.2: Employee

(single) values", the emp_mobile values for employees Jon & Lester violates that rule.
To make the table complies with 1NF we should have the data like this:
Now this table is in 1NF.

Second normal form(2NF)
Consider a relation schema R with set of functional dependencies F. A relation schema R is said to be in second normal form(2NF) if

emp_id	emp_name	emp_address	emp_mobile
101	Herschel	New Delhi	8912312390
102	Jon	Kanpur	8812121212
102	Jon	Kanpur	9900012222
103	Ron	Chennai	7778881212
104	Lester	Bangalore	9990000123
104	Lester	Bangalore	8123450987

Table 5.3: Revised Employee table

(1) It is in 1NF.
(2) Every non-prime attribute is fully functionally dependent on candidate key.

Prime attribute An attribute which is a part of any candidate key is said to be prime attribute. And which is not part of any candidate key is said to be non-prime attribute.

Example: Consider a relation schema R = (A,B,C,D) with only one candidate key as {A,B}. In this case, A and B are prime attributes. C and D are non-prime attributes.

Partial functional dependent
A partial dependency means if the non-key attributes depend on the part of candidate key then it is said to be partial dependency.
An attribute is fully functional dependent on a set of attributes α, if it is functionally dependent on only α and not on any of its proper subset.

Note: A relation with a single-attribute primary key is automatically in at least 2NF.

Example: Consider following functional dependencies in relation R(A, B, C, D)
AB $\rightarrow$ C
BC $\rightarrow$ D
Is this relation in 2NF?
Solution:
First find candidate keys. Here candidate key is A,B. Clearly non-prime attributes are C and D. From functional dependencies, C and D are fully functional dependent, therefore R is in 2NF.

Example: Consider a relation- R (V , W , X , Y , Z) with functional dependencies-
VW $\rightarrow$ XY
Y $\rightarrow$ V
WX $\rightarrow$ YZ
Is this relation in 2NF?
Solution:
Here candidate keys are VW, WX and WY. Therefore non-prime attributes are Z. Clearly Z is fully dependent on candidate keys. Therefore R is in 2NF.

Example: Consider relation R(A, B, C, D, E) with set of following functional dependencies
A $\rightarrow$ BC,

CD → E,
B → D,
E → A
Is this relation in 2NF?
Solution:
Here candidate keys are A, E, CD, BC. Clearly all the attributes belong into some candidate keys, therefore no non-prime attribute. Therefore, R is in 2NF.

Third normal form(3NF)
A relation schema R is in third normal form (3NF) with respect to a set F of functional dependencies if, for all functional dependencies in F^+ of the form $\alpha \rightarrow \beta$, where $\alpha \subseteq R$ and $\beta \subseteq R$, at least one of the following holds:

- $\alpha \rightarrow \beta$ is a trivial functional dependency.
- α is a super key for R.
- Each attribute A in $\beta - \alpha$ is contained in a candidate key for R.

In other words, we can define 3NF as following:-
A relation schema R with set of functional dependencies F is said to in 3NF if
(1) It is in 2NF.
(2) No non-prime attribute transitively depends on any candidate key.

Example: Consider a relation- R (V , W , X , Y , Z) with functional dependencies-
VW → XY
Y → V
WX → YZ
Is this relation in 3NF?
Solution:
Here candidate keys are VW, WX and WY. Therefore non-prime attributes are Z. Clearly Z is fully dependent on candidate keys. Therefore R is in 2NF.
Now, Z is not transitively dependent on any candidate key. Therefore, it is in 3NF.

Example: Consider relation R(A, B, C, D, E) with set of following functional dependencies
A → BC,
C → E,
B → D,
Is this relation in 3NF?
Solution:
Here candidate key is A. Therefore, non-prime attributes are B, C, D, E. Since candidate key contains single attribute, therefore R is in 2NF.
Now, Clearly attributes D and E are transitively dependent on candidate key A, therefore R is not in 3NF.

Example: The relation schema Student_Performance (name, courseNo, rollNo, grade) has the following FDs:
name,courseNo → grade
rollNo,courseNo → grade

name → rollNo
rollNo → name
Is this relation in 3NF?
Solution:
Here candidate keys are {name, courseNo} and {rollNo, courseNo}. Therefore non-prime attributes are grade. Clearly, grade is fully functional dependent on candidate keys, therefore this relation schema is in 2NF.
Now, clearly grade is not transitively dependent on candidate keys, therefore this relation schema is in 3NF.

Boyce-codd normal form (BCNF)
A relation schema R is in BCNF with respect to a set F of functional dependencies if, for all functional dependencies in F^+ of the form $\alpha \rightarrow \beta$, where $\alpha \subseteq R$ and $\beta \subseteq R$, at least one of the following holds:

- $\alpha \rightarrow \beta$ is a trivial functional dependency.
- α is a super key for R.

Example: The relation schema Student_Performance (name, courseNo, rollNo, grade) has the following FDs:
name,courseNo → grade
rollNo,courseNo → grade
name → rollNo
rollNo → name
Is this relation in BCNF?
Solution:
Consider FD, name,courseNo → grade. Clearly, {name, courseNo} is a super key, therefore this satisfy 2nd criteria of BCNF.
Consider FD, rollNo,courseNo → grade. Clearly, {rollNo, courseNo} is a super key, therefore this satisfy 2nd criteria of BCNF.
Consider FD, name → rollNo. Clearly, this functional dependency not satisfy any condition of BCNF. Therefore, this is not in BCNF.

Example: Consider the following relation schemas and their respective functional dependencies:
Customer = (customer-name, customer-street, customer-city)
customer-name → customer-street, customer-city
Branch = (branch-name, assets, branch-city)
branch-name → assets, branch-city
Loan = (branch-name, customer-name, loan-number, amount)
loan-number → amount, branch-name

Clearly, Customer and Branch schema are in BCNF, because left side of functional dependency **customer-name → customer-street, customer-city** and **branch-name → assets, branch-city** is super key.
But Loan schema is not in BCNF, because left side of functional dependency **loan-number → amount, branch-name** is not super key and this functional dependency is also not trivial.

5.2.3 Decomposition into Normal form

Decomposition into 2NF
Example:
Let R = { A, B, C, D} and F = { AB → C, B → D }.
Is this relation schema in 2NF? If not then decompose it into 2NF.
Solution:
Here, Primary key = {A,B}.
Clearly, this is not in 2NF because partial dependency holds.
Now, we decompose R into R_1 and R_2 as the following:-
R_1 = (A, B, C), F_1 = {AB → C}
R_2 = (B, D), F_2 = {B → D}
Now, R_1 and R_2 are in 2NF.

Example:
Student-course-info(Name, Course, Grade, Phone-no, Major, Course-dept)
F = { Name → Phone-no Major, Course → Course-dept, Name Course → Grade }
Is this relation schema in 2NF? If not then decompose it into 2NF.
Solution:
Here, Primary key = {Name,Course}.
Clearly, this is not in 2NF because partial dependency holds.
Now, we decompose R into R_1, R_2 and R_3 as the following:-
R_1 = (Name, Phone-no, Major), F_1 = {Name → Phone-no Major}
R_2 = (Course, Course-dept), F_2 = {Course → Course-dept}
R_3 = (Name, Course, Grade), F_3 = {Name Course → Grade}
Now, R_1, R_2 and R_3 are in 3NF.

Decomposition into 3NF
Example:
Let R = { A, B, C, D} and F = { A → B, A → C, B → D }.
Is this relation schema in 3NF? If not then decompose it into 3NF.
Solution:
Here, Primary key = {A}.
Clearly, this is not in 3NF because transitivity dependency holds.
Now, we decompose R into R_1 and R_2 as the following:-
R_1 = (A, B, C), F_1 = {A → B, A → C}
R_2 = (B, D), F_2 = {B → D}
Now, R_1 and R_2 are in 3NF.

Example:
Let Banker-info = { branch-name, customer-name, banker-name, office-number} and F = { banker-name → branch-name office-number, customer-name branch-name → banker-name}.
Is this relation schema in 3NF? If not then decompose it into 3NF.
Solution:
Here, Primary key = {customer-name, branch-name}.
Clearly, this is not in 3NF because transitivity dependency holds.
Now, we decompose relation Banker-info into R_1 and R_2 as the following:-

R_1 = (banker-name, branch-name, office-number), F_1 = {banker-name → branch-name office-number}
R_2 = (customer-name, branch-name, banker-name), F_2 = {customer-name branch-name → banker-name}
Now, R_1 and R_2 are in 3NF.

Decomposition into BCNF
If R is not in BCNF, we can decompose R into a collection of BCNF schemas $R_1, R_2, ..., R_n$ by the algorithm. The decomposition that the algorithm generates is not only in BCNF, but is also a lossless-join decomposition.

Input: R and F
Output: result
$result \leftarrow R$
done = false
compute F^+
while *not done* **do**
  if *(there is a schema R_i in result that is not in BCNF)* **then**
    let $\alpha \rightarrow \beta$ be a nontrivial functional dependency that holds on R_i such that $\alpha \rightarrow R_i$ is not in F^+, and $\alpha \cap \beta = \phi$
    $result \leftarrow (result - R_i) \cup (R_i - \beta) \cup (\alpha, \beta)$
  end
  else
    done = true
  end
end

Algorithm 4: BCNF decomposition algorithm

Example:
Consider the following schema:-
Lending-schema = (branch-name, branch-city, assets, customer-name, loan-number, amount)
F = { branch-name → assets branch-city, loan-number → amount branch-name}
Is this relation schema in BCNF? If not then decompose it into BCNF.
Solution:
Here, Primary key = {customer-name, loan-number}.
Clearly, this is not in BCNF.
Now, we decompose relation Lending-schema into R_1 and R_2 as the following:-
R_1 = (customer-name, loan-number, amount, branch-name)
F_1 = { loan-number → amount branch-name}
R_2 = (branch-name, assets, branch-city)
F_2 = { branch-name → assets branch-city}
Clearly, R_1 is not in BCNF. Therefore, we again decompose R_1.
Now, we decompose relation R_1 into R_3 and R_4 as the following:-
R_3 = (customer-name, loan-number)
$F_3 = \phi$
R_4 = (loan-number, amount, branch-name)
F_4 = { loan-number → amount branch-name}

Now, the final relation schema are R_2, R_3 and R_4. All these are in BCNF.

5.2.4 Algorithm to check if a decomposition is lossless

Input: A relation schema R($A_1, A_2, ..., A_n$) and a decomposition D = $\{R_1, R_2, ..., R_m\}$ and a set F of functional dependencies.

1. Create a matrix S with one row i for each relation R_i in D, and one column j for each attribute a_j in R.

2. Set S(i,j) = a_j , if relation schema R_i contains attribute A_j otherwise set S(i,j) = b_{ij}.

3. .

change ← *true*
while *(change)* **do**
  for *each functional dependency* $\alpha \rightarrow \beta$ *in* F **do**
    if *(row i and j exists such that the same symbol appears in each column corresponding to attribute in* α*)* **then**
      if *one of the symbols in the* β *column is* a_r **then**
        Make other symbol to be a_r
      end
      else if *the symbols are* b_{pk} *and* b_{qk} **then**
        Make both of them b_{pk}
      end
    end
    else
      change = false
    end
  end
end

4. If there exists a row in which all the symbols are a's, then the decomposition has the lossless. Otherwise decomposition has lossy.

Example:
Consider the following schema:-
R=(SSN, Ename, Pnumber, Pname, Plocation, Hours)
F= { SSN → Ename, Pnumber → Pname Plocation, SSN pnumber → Hours }
R_1 = (SSN, Ename)
R_2 = (Pnumber, Pname, Plocation)
R_3 = (SSN, Pnumber, Hours)
Find out decomposition of R in R_1, R_2, R_3 is lossless or lossy.
Solution:
First construct matrix. It will be order of 3×6.
The initialization table will be the following:-

	SSN	Ename	Pnumber	Pname	Plocation	Hours
R_1	a_1	a_2	b_{13}	b_{14}	b_{15}	b_{16}
R_2	b_{21}	b_{22}	a_3	a_4	a_5	b_{26}
R_3	a_1	b_{32}	a_3	b_{34}	b_{35}	a_6

After the first iteration , the table will be the following:-

	SSN	Ename	Pnumber	Pname	Plocation	Hours
R_1	a_1	a_2	b_{13}	b_{14}	b_{15}	b_{16}
R_2	b_{21}	b_{22}	a_3	a_4	a_5	b_{26}
R_3	a_1	a_2	a_3	a_4	a_5	a_6

After the second iteration, the table will not changed. Therefore, the above table is the final table.
Since in this table, row 3 contains only a's symbol, therefore this decomposition is lossless.

Example:
Consider the following schema:-
R=(A, B, C, D, E)
F= { AB $\rightarrow$ CD, A $\rightarrow$ E, C $\rightarrow$ D }
R_1 = (A, B, C)
R_2 = (B, C, D)
R_3 = (C, D, E)
Is the decomposition of R in R_1, R_2, R_3 is lossless or lossy?
Solution:
First construct matrix. It will be order of 3×5.
The initialization table will be the following:-

	A	B	C	D	E
R_1	a_1	a_2	a_3	b_{14}	b_{15}
R_2	b_{21}	a_2	a_3	a_4	b_{25}
R_3	b_{31}	b_{32}	a_3	a_4	a_5

After the first iteration , the table will be the following:-

	A	B	C	D	E
R_1	a_1	a_2	a_3	a_4	b_{15}
R_2	b_{21}	a_2	a_3	a_4	b_{25}
R_3	b_{31}	b_{32}	a_3	a_4	a_5

After the second iteration, the table will not changed. Therefore, the above table is the final table.
Since in this table, no row contains only a's symbol, therefore this decomposition is lossy.

5.3 Multivalued dependency

Let R be a relation schema and let $\alpha \subseteq R$ and $\beta \subseteq R$. The multivalued dependency $\alpha \rightarrow\rightarrow \beta$ holds on R if in any legal relation r(R), for all pairs of t_1 and t_2 in r such that $t_1[\alpha] = t_2[\alpha]$, there exists two tuples t_3 and t_4 in r such that

$$t_1[\alpha] = t_2[\alpha] = t_3[\alpha] = t_4[\alpha]$$
$$t_3[\beta] = t_1[\beta]$$
$$t_4[\beta] = t_1[\beta]$$
$$t_3[R - \beta] = t_2[R - \beta]$$
$$t_4[R - \beta] = t_1[R - \beta]$$

5.3.1 Trivial multivalued dependency

A multivalued dependency $\alpha \rightarrow\rightarrow \beta$ is said to be trivial if $\beta \subseteq \alpha$ or $\beta \cup \alpha = R$.

Note: If $\alpha \rightarrow \beta$, then $\alpha \rightarrow\rightarrow \beta$. That is, every functional dependency is also a multivalued dependency.

Example: Consider the following relation schema.

loan-number	customer-name	customer-street	customer-city
L-23	Smith	North	Rye
L-23	Smith	Main	Manchester
L-93	Curry	Lake	Horseneck

In this table, multivalued dependency

customer-name $\rightarrow\rightarrow$ customer-street customer-city

holds.

5.3.2 Axioms for Multivalued dependency

(1) Replication rule:
$X \rightarrow Y \Rightarrow X \rightarrow\rightarrow Y$

(2) Reflexivity rule:
$X \rightarrow\rightarrow X$

(3) Augmentation rule:
$X \rightarrow\rightarrow Y \Rightarrow XZ \rightarrow\rightarrow Y$

(4) Union rule:
$X \rightarrow\rightarrow Y$ and $X \rightarrow\rightarrow Z \Rightarrow X \rightarrow\rightarrow YZ$

(5) Complementation rule:
$X \rightarrow\rightarrow Y \Rightarrow X \rightarrow\rightarrow (R - X - Y)$

(6) Transitivity rule:
$X \rightarrow\rightarrow Y$ and $Y \rightarrow\rightarrow Z \Rightarrow X \rightarrow\rightarrow (Z - Y)$

(7) Intersection rule:
$X \rightarrow\rightarrow Y$ and $X \rightarrow\rightarrow Z \Rightarrow X \rightarrow\rightarrow (Y \cap Z)$

(8) Difference rule:
$X \rightarrow\rightarrow Y$ and $X \rightarrow\rightarrow Z \Rightarrow X \rightarrow\rightarrow (Y - Z)$ *and* $X \rightarrow\rightarrow (Z - Y)$

(9) Pseudo transitivity rule:

$X \rightarrow\rightarrow Y$ and $XY \rightarrow\rightarrow Z \Rightarrow X \rightarrow\rightarrow (Z - Y)$

(10) Coalescence rule:
Given that $W \subseteq Y$ and $Y \cap Z = \phi$, and if $X \rightarrow\rightarrow Y$ and $Z \rightarrow W$ then $X \rightarrow W$.

5.3.3 Closure under Multivalued dependency

Let D be the set of functional and multivalued dependencies.
The closure of D is the set of all functional and multivalued dependencies that are logically implied by D. It is denoted by D^+.

Example: Consider R = (A, B, C, G, H, I)
and F =$\{A \rightarrow\rightarrow B, B \rightarrow\rightarrow HI, CG \rightarrow H\}$
Find some members of D^+.
Solution: Some members of D^+ are the following:-

- $A \rightarrow\rightarrow CGHI$, By complementation rule in $A \rightarrow\rightarrow B$
- $A \rightarrow\rightarrow HI$, By transitivity rule in $A \rightarrow\rightarrow B$ and $B \rightarrow\rightarrow HI$
- $B \rightarrow H$, By coalescence rule in $B \rightarrow\rightarrow HI$ and $CG \rightarrow H$
- $A \rightarrow\rightarrow CG$, By difference rule in $A \rightarrow\rightarrow CGHI$ and $A \rightarrow\rightarrow HI$.

5.4 Fourth Normal Form(4NF)

A relation schema R is in fourth normal form(4NF) with respect to a set D of functional and multivalued dependencies if, for all multivalued dependencies in D^+ of the form $\alpha \rightarrow\rightarrow \beta$, where $\alpha \subseteq R$ and $\beta \subseteq R$, at least one of the following holds:

- $\alpha \rightarrow\rightarrow \beta$ is a trivial multivalued dependency.
- α is a super key for schema R.

Note: Every 4NF schema is in BCNF.

5.4.1 Decomposition in to 4NF

Following algorithm is used to decompose schema R into 4NF. **Example:** Consider the following relation schema.

loan-number	customer-name	customer-street	customer-city
L-23	Smith	North	Rye
L-23	Smith	Main	Manchester
L-93	Curry	Lake	Horseneck

Input: Relation schema R and set D
Output: $R_1, R_2, ..., R_m$
$result \leftarrow R$
done=false
Compute D^+
while *not done* **do**
  if *(there is a schema R_i in result that is not in 4NF w.r.t. D_i)* **then**
    let $\alpha \rightarrow\rightarrow \beta$ be a nontrivial multivalued dependency that holds on R_i
     such that $\alpha \rightarrow R_i$ is not in D_i, and $\alpha \cap \beta = \phi$
    $result \leftarrow (result - R_i) \cup (R_i - \beta) \cup (\alpha, \beta)$
  end
  else
    done = true
  end
end

Algorithm 5: BCNF decomposition algorithm

Find out this table is in 4NF or not. If not, then decompose it into 4NF.
Solution: In this table, multivalued dependency
customer-name $\rightarrow\rightarrow$ customer-street customer-city
holds.
Clearly, neither this multivalued dependency is trivial nor customer-name is super key. Therefore, this table is not in 4NF.
By using above algorithm, this table is decomposed as
R_1= (customer-name, loan-number)
R_2= (customer-name, customer-street, customer-city)
Now, R_1 and R_2 are in 4NF.

5.5 Join dependency

Given a relation schema R. let $R_1, R_2, ..., R_n$ are the projections of R. A relation r(R) satisfies the join dependency $*(R_1, R_2, ..., R_n)$, iff the join of the projection of r on R_i, $1 \leq i \leq n$, is equal to r.

$$r = \Pi_{R_1} \bowtie \Pi_{R_2} \bowtie \Pi_{R_3} \bowtie ..... \bowtie \Pi_{R_n}$$

5.5.1 Trivial join dependency

A join dependency is trivial if one of the projections of R is R itself.

5.5.2 Project join normal form(PJNF) or 5NF

Consider a relation schema R and D is the set of functional, multivalued and join dependencies.
The relation R is in Project join normal form with respect to D if for every join dependency $*(R_1, R_2, ..., R_n)$, either of the following holds:-
(i) The join dependency is trivial.
(ii) Every R_i is a super key of R.

5.5.3 Exercise

1. Consider R = (A, B, C, D, E, F, G) and
   F= $\{A \rightarrow B,\ BC \rightarrow F,\ BD \rightarrow EG,\ AD \rightarrow C,\ D \rightarrow F,\ BEG \rightarrow FA\}$
   Calculate the following:-
   (a) $(A)^+$
   (b) $(ACEG)^+$
   (c) $(BD)^+$

2. Consider R = (A, B, C, D, E) and
   F= $\{A \rightarrow B,\ BC \rightarrow E,\ ED \rightarrow A\}$
   (a) List all the candidate keys for R.
   (b) Is R in third normal form?
   (c) Is R in BCNF?

3. Consider R = (A, B, C, D, E, F) and
   F= $\{AB \rightarrow C,\ C \rightarrow B,\ ABD \rightarrow E,\ AD \rightarrow C,\ F \rightarrow A\}$
   The decomposition of R is
   D=$\{R_1(B,C),\ R_2(A,C),\ R_3(A,B,D,E),\ R_4(A,B,D,F)\}$
   Check whether the decomposition is lossless or lossy.

4. Consider R = (V, W, X, Y, Z) and
   F= $\{Z \rightarrow V,\ W \rightarrow Y,\ XY \rightarrow Z,\ V \rightarrow WX\}$
   Sate whether the following decomposition of schema R is lossless join decomposition. Justify your answer.
   (i) R_1 = (V,W,X) and R_2 = (V,Y,Z)
   (ii) R_1 = (V,W,X) and R_2 = (X,Y,Z)

5. Consider R = (A, B, C, D, E, F, G, H, I, J) and
   F= $\{AB \rightarrow C,\ A \rightarrow DE,\ B \rightarrow F,\ F \rightarrow GH,\ D \rightarrow IJ\}$
   Is R in 2NF? If not, then decompose it into 2NF.

6. Consider R = (A, B, C, D, E) and
   F= $\{A \rightarrow BC,\ CD \rightarrow E,\ B \rightarrow D,\ E \rightarrow A\}$
   (i) List all the candidate keys for R.
   (ii) Compute the canonical cover.

7. Consider R = (A, B, C, D, E) and
   F= $\{A \rightarrow\!\!\rightarrow BC,\ B \rightarrow\!\!\rightarrow CD,\ E \rightarrow\!\!\rightarrow AD\}$
   Is R in 4NF? If not, then decompose it into 4NF.

8. Consider R = (A, B, C, D, E, F, G, H) and
   F= $\{AB \rightarrow C,\ BC \rightarrow D,\ E \rightarrow F,\ G \rightarrow F,\ H \rightarrow A,\ FG \rightarrow H\ \}$
   Is the decomposition of R into $R_1(A,B,C,D)$, $R_2(A,B,C,E,F)$, $R_3(A,D,F,G,H)$ lossless? Is it dependency preserving?

9. Define partial functional dependency. Consider the following two sets of functional dependencies F = A →C, AC →D, E →AD, E →H and G = A →CD, E →AH. Check whether or not they are equivalent.

10. Define Minimal Cover. Suppose a relation R (A,B,C) has FD set F = A → B, B →C, A → C, AB → B, AB → C, AC → B convert this FD set into minimal cover.

11. Write the difference between 3NF and BCNF. Find normal form of relation R(A,B,C,D,E) having FD set F= A→B,BC→E,ED→A.

5.6 AKTU Examination Questions

1. Explain normalization. What is normal form?
2. Describe Multivalued dependency.
3. What are the different types of anomalies associated with database?
4. Why do we normalize database?
5. Define partial functional dependency. Consider the following two sets of functional dependencies F = A →C, AC →D, E →AD, E →H and G = A →CD, E →AH. Check whether or not they are equivalent.
6. Define Minimal Cover. Suppose a relation R (A,B,C) has FD set F = A → B, B →C, A → C, AB → B, AB → C, AC → B convert this FD set into minimal cover.
7. Write the difference between 3NF and BCNF. Find normal form of relation R(A,B,C,D,E) having FD set F= A→B,BC→E,ED→A.
8. Define 2 NF.
9. Write difference between BCNF Vs 3 NF.
10. Short Notes of the Following
    (i)MVD or JD
    (ii) Normalization with advantages
11. Explain 1NF, 2NF, 3NF and BCNF with suitable example.
12. Consider the universal relation schema R = (A, B, C, D, E, F, G, H, I, J) and F= $\{AB \rightarrow C, A \rightarrow DE, B \rightarrow F, F \rightarrow GH, D \rightarrow IJ\}$
    Determine the keys for R? Decompose R into 2NF.
13. Distinguish between functional dependency and multivalued dependency.
14. Define functional dependency. What do you mean by lossless decomposition ? Explain with suitable example how function dependencies can be used to show that decomposition is lossless.

5.7 Gate questions

1. From the following instance of a relation schema R(A,B,C), we can conclude that

    (a) A functionally determines B and B functionally determines C
    (b) A functionally determines B and B does not functionally determines C
    (c) B does not functionally determines C
    (d) A does not functionally determines B and B does not functionally determines

A	B	C
1	1	1
1	1	0
2	3	2
2	3	2

C

2. Consider the following functional dependencies in a database.
   Date_of_birth → Age, Age → Eligibility
   Name → Roll_number, Roll_number → Name
   Course_number → Course_name, Course_number → Instructor
   (Roll_number, Course_number) → grade

   The relation (Roll_number, Name, Date_of_birth, Age) is in
   (a) Second normal form but not in third normal form
   (b) Third normal form but not in BCNF
   (c) BCNF
   (d) None of the above

3. The relation schema student performance (name, courseNo, rollNo, grade) has the following functional dependencies:-
   (name, courseNo) → grade
   (rollNo, courseNo) → grade
   name → rollNo
   rollNo → name

   The highest normal form of this relation schema is
   (a) 2NF (b) 3NF (c) BCNF (d) 4NF

Chapter 6

Transaction

6.1 Transaction

- A transaction is a unit of program execution that accesses and possibly updates various data items.
- A transaction is an action or sequence of actions. It is performed by a single user to perform operations for accessing the contents of the database.

Example: Suppose an employee of bank transfers Rs. 800 from X's account to Y's account. This small transaction contains several low-level tasks:
X's Account:
Open_Account(X)
Old_Balance = X.balance
New_Balance = Old_Balance - 800
X.balance = New_Balance
Close_Account(X)

Y's Account:
Open_Account(Y)
Old_Balance = Y.balance
New_Balance = Old_Balance + 800
Y.balance = New_Balance
Close_Account(Y)

Operations of Transaction:

Following are the main operations of transaction:

Read(X): Read operation is used to read the value of X from the database and stores it in a buffer in main memory.
Write(X): Write operation is used to write the value back to the database from the buffer.

Let's take an example to debit transaction from an account which consists of following operations:

1. Read(X);
2. X = X - 500;
3. Write(X);

Let's assume the value of X before starting of the transaction is 4000.

- The first operation reads X's value from database and stores it in a buffer.
- The second operation will decrease the value of X by 500. So buffer will contain 3500.
- The third operation will write the buffer's value to the database. So X's final value will be 3500.

But it may be possible that because of the failure of hardware, software or power, etc. that transaction may fail before finished all the operations in the set.

For example: If in the above transaction, the debit transaction fails after executing operation 2 then X's value will remain 4000 in the database which is not acceptable by the bank.

To solve this problem, we have two important operations:

Commit: It is used to save the work done permanently.

Rollback: It is used to undo the work done.

6.2 Properties of Transaction

To ensure integrity of the data, we require that the database system maintain the following properties of the transactions:

1. **Atomicity:** Either all operations of the transaction are reflected properly in the database, or none are.
2. **Consistency:**
   - This means that integrity constraints must be maintained so that the database is consistent before and after the transaction. It refers to the correctness of a database.
   - Execution of a transaction in isolation (that is, with no other transaction executing concurrently) preserves the consistency of the database.
3. **Isolation:**
   - In isolation, if the transaction T1 is being executed and using the data item X, then that data item can't be accessed by any other transaction T2 until the transaction T1 ends.
   - It shows that the data which is used at the time of execution of a transaction cannot be used by the second transaction until the first one is completed.

4. **Durability:** After a transaction completes successfully, the changes it has made to the database persist, even if there are system failures.

These properties are often called the ACID properties.

Example: Let Ti be a transaction that transfers $50 from account A to account B. This transaction can be defined as

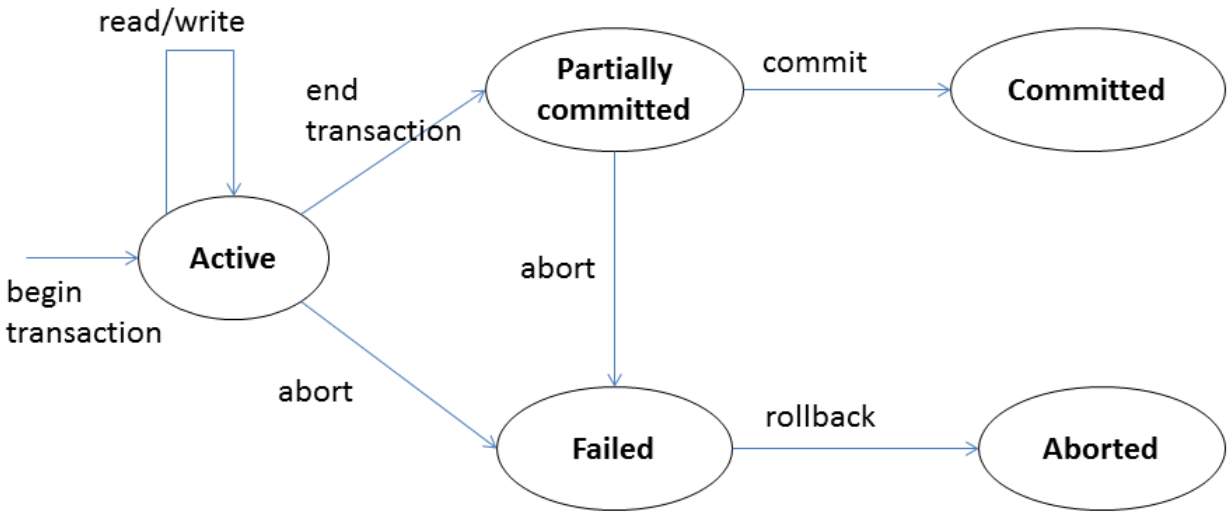
Ti: read(A);
A := A - 50;
write(A);
read(B);
B := B + 50;
write(B).

6.3 Transaction State

A transaction must be in one of the following states:

- **Active:** This is the initial state. The transaction stays in this state while it is executing.
- **Partially committed:** Transaction enter into this state after the final statement has been executed.
- **Failed:** Transaction enter into this state after the discovery that normal execution can no longer proceed.
- **Aborted:** Transaction enter into this state after the transaction has been rolled back and the database has been restored to its state prior to the start of the transaction.
- **Committed:** Transaction enter into this state after successful completion.

The state diagram corresponding to a transaction is the following:-



State diagram for the execution of a transaction

When transaction enters the aborted state, at this point, the system has two options:

- It can restart the transaction, but only if the transaction was aborted as a result of some hardware or software error that was not created through the internal logic of the transaction. A restarted transaction is considered to be a new transaction.
- It can kill the transaction. It usually does so because of some internal logical error that can be corrected only by rewriting the application program, or because the input was bad, or because the desired data were not found in the database.

6.4 Schedule

A schedule is a sequence of instructions of all the transactions in which order these instructions will execute.
There are two types of schedule. (1) Serial schedule (2) Concurrent schedule
Serial schedule: The serial schedule is a type of schedule where one transaction is executed completely before starting another transaction. In the serial schedule, when the first transaction completes its cycle, then the next transaction is executed. **Concurrent schedule:** If interleaving of operations is allowed, then the schedule will be concurrent schedule.

Example: Consider the following two transactions:-
Let T1 and T2 be two transactions that transfer funds from one account to another. Transaction T1 transfers $50 from account A to account B. It is defined as

T1: read(A);
A := A - 50;
write(A);
read(B);

B := B + 50;
write(B).

Transaction T2 transfers 10 percent of the balance from account A to account B. It is defined as

T2: read(A);
temp := A * 0.1;
A := A - temp;
write(A);
read(B);
B := B + temp;
write(B).

Now we make following four schedules for these transactions.
schedule-1, schedule-2, schedule-3, and schedule-4.

Schedule-1

T_1	T_2
read(A)	
$A := A - 50$	
write (A)	
read(B)	
$B := B + 50$	
write(B)	
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
	read(B)
	$B := B + temp$
	write(B)

Schedule-2

T_1	T_2
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
	read(B)
	$B := B + temp$
	write(B)
read(A)	
$A := A - 50$	
write(A)	
read(B)	
$B := B + 50$	
write(B)	

Schedule-3

T_1	T_2
read(A)	
$A := A - 50$	
write(A)	
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
read(B)	
$B := B + 50$	
write(B)	
	read(B)
	$B := B + temp$
	write(B)

Schedule-4

T_1	T_2
read(A)	
$A := A - 50$	
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
	read(B)
write(A)	
read(B)	
$B := B + 50$	
write(B)	
	$B := B + temp$
	write(B)

6.5 Serializability

To ensure consistency of database system, we must make a serializable schedule. Here, we will study two types of Serializability. These are (1) Conflict Serializability (2) View Serializability.

6.5.1 Conflict Serializability

Conflict Instructions:
Consider a schedule S in which there are two consecutive instructions I_i and I_j, of transactions T_i and T_j , respectively (i≠j). If I_i and I_j operates on same data item such as Q, then these instructions will be conflicting instructions if any one of the following is satisfied.

(1) I_i = read(Q), and I_j = write(Q).
(2) I_i = write(Q), and I_j = read(Q).
(3) I_i = write(Q), and I_j = write(Q).

In all other cases, instructions I_i and I_j will be non-conflicting.

Conflict equivalent: Two schedules S and S' are said to be conflict equivalent if a schedule S can be transformed in to a schedule S' by using swapping of non-conflicting instructions.

Conflict serializable: A schedule S is said to be conflict serizaliabe if it is conflict equivalent to a serial schedule.

Some examples:

Example: Check schedule-1, schedule-2, schedule-3 and schedule-4 are conflict serializable or not.
Solution:
(1) Clearly schedule-1 and schedule-2 are serial schedules,therefore these schedules are conflict serializable.
(2) Consider schedule-3 i.e.

Schedule-3

T_1	T_2
read(A)	
$A := A - 50$	
write(A)	
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
read(B)	
$B := B + 50$	
write(B)	
	read(B)
	$B := B + temp$
	write(B)

In this schedule, instruction read(B) in transaction T_1 is non-conflicting with instructions read(A) and write(A) in transaction T_2, therefore we can swap instructions read(B) in T_1 and write(A)in T_2. Similarly, we can swap instructions read(B) in T_1 and read(A)in T_2.

Similarly, instruction write(B) in transaction T_1 is non-conflicting with instructions read(A) and write(A) in transaction T_2, therefore we can swap instructions write(B) in T_1 and write(A)in T_2. Similarly, we can swap instructions write(B) in T_1 and read(A)in T_2.

Therefore, using swapping, schedule-3 can be transformed in to a serial schedule-1 i.e. T_1T_2. Therefore schedule-3 is conflict serializable.

(3) Consider schedule-4 i.e.

Schedule-4

T_1	T_2
read(A)	
$A := A - 50$	
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
	read(B)
write(A)	
read(B)	
$B := B + 50$	
write(B)	
	$B := B + temp$
	write(B)

In this schedule, instruction write(A) in transaction T_1 and instruction write(A) in transaction T_2 are conflicting instructions, therefore, we can not swap these two instructions. Therefore, schedule-4 can not be transformed in to any serial schedule. Hence, schedule-4 is not conflict serializable.

Example: Consider the following schedule-5:-

Schedule-5

T_3	T_4
read(Q)	
	write(Q)
write(Q)	

Is this schedule conflict serializable?

Solution:

Clearly, in this schedule, instruction read(Q) in transaction T_3 and instruction write(Q) in transaction T_4 are conflicting instructions, therefore, we can not swap these two instructions.

Similarly, instruction write(Q) in transaction T_3 and instruction write(Q) in transaction T_4 are conflicting instructions, therefore, we can not swap these two instructions.

In this situation, this schedule can not be transformed into any serial schedule. Therefore, the schedule-5 is non-conflict serializable.

Example: Consider the following schedule-6:-

Schedule-6

T_1	T_5
read(A)	
$A := A - 50$	
write(A)	
	read(B)
	$B := B - 10$
	write(B)
read(B)	
$B := B + 50$	
write(B)	
	read(A)
	$A := A + 10$
	write(A)

Is this schedule conflict serializable?

Solution:

In this schedule, instruction read(B) in transaction T_1 and instruction write(B) in transaction T_5 are conflicting instructions, therefore, we can not swap these two instructions. Hence, this schedule can not be transformed into serial schedule T_1T_5.

Similarly, instruction write(A) in transaction T_1 and instruction read(A) in transaction T_5 are conflicting instructions, therefore, we can not swap these two instructions. Hence, this schedule can not be transformed into serial schedule T_5T_1.

Therefore, this schedule-6 is non-conflict serializable.

6.5.2 View Serializability

View equivalent: Consider two schedules S and S', where the same set of transactions participates in both schedules. The schedules S and S' are said to be view equivalent if following three conditions are satisfied:

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 if that value was produced by a write(Q) operation executed by transaction T_j , then the

read(Q) operation of transaction T_i must, in schedule S', also read the value of Q that was produced by the same write(Q) operation of 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'.

View serializable: A schedule S is said to be view serizaliabe if it is view equivalent to a serial schedule.

Example: Consider the following schedule-3:-

Schedule-3

T_1	T_2
read(A)	
$A := A - 50$	
write(A)	
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
read(B)	
$B := B + 50$	
write(B)	
	read(B)
	$B := B + temp$
	write(B)

Is this schedule view serializable?

Solution:

First, we check the view equivalent of schedule-3 with serial schedule-1 i.e. T_1T_2.

First consider data item A.

Clearly, in schedule-3 and schedule-1, transaction T_1 reads the initial value of A, therefore condition-1 is satisfied for data item A.

Clearly, in schedule-3, transaction T_2 reads the value of A written by T_1. In schedule-1, transaction T_2 reads the value of A written by T_1. Clearly same order of write-read occur in both schedule, therefore second condition is also satisfied.

Clearly, in schedule-3, final write(A) operation is performed by transaction T_2. In schedule-1, final write(A) operation is also performed by transaction T_2. Therefore, third condition is also satisfied.

Clearly all the three conditions are satisfied for data item A.

Now, we will check all the three conditions for data item B.

Clearly, in schedule-3 and schedule-1, transaction T_1 reads the initial value of B, therefore condition-1 is satisfied for data item B.

Clearly, in schedule-3, transaction T_2 reads the value of B written by T_1. In schedule-1, transaction T_2 reads the value of B written by T_1. Clearly same order of write-read occur in both schedule, therefore second condition is also satisfied.

Clearly, in schedule-3, final write(B) operation is performed by transaction T_2. In schedule-1, final write(B) operation is also performed by transaction T_2. Therefore, third condition is also satisfied.

Clearly all the three conditions are satisfied for data item B.

Since all the three conditions are satisfied for each data item in both schedules 1 and 3,

therefore both schedule-3 is view equivalent to serial schedule-1. Therefore, schedule-3 is view serializable.‘

Example: Consider the following schedule-4:-

Schedule-4

T_1	T_2
read(A)	
$A := A - 50$	
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
	read(B)
write(A)	
read(B)	
$B := B + 50$	
write(B)	
	$B := B + temp$
	write(B)

Is this schedule view serializable?

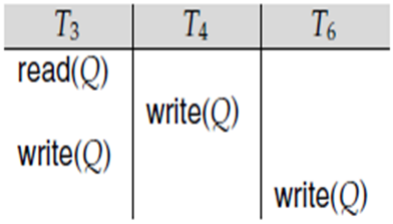
Solution:

First, we check the view equivalent of schedule-4 with serial schedule-1 i.e. T_1T_2.
First consider data item A.
Clearly, in schedule-4 and schedule-1, transaction T_1 reads the initial value of A, therefore condition-1 is satisfied for data item A.
Clearly, in schedule-4, there is no transaction which reads the value of A written by any transaction. Therefore second condition is also satisfied.
Clearly, in schedule-4, final write(A) operation is performed by transaction T_1. But, in schedule-1, final write(A) operation is performed by transaction T_2. Therefore, third condition is not satisfied.
Therefore, schedule-4 is not view equivalent to a serial schedule T_1T_2.

Now, we check the view equivalent of schedule-4 with serial schedule-2 i.e. T_2T_1.
First consider data item A.
Clearly, in schedule-4, transaction T_1 reads the initial value of A, but in serial schedule-2, transaction T_2 reads the initial value of A. Therefore condition-1 is not satisfied for data item A.
Therefore, schedule-4 is not view equivalent to a serial schedule T_1T_2.
Therefore, schedule-4 is not view serializable.‘

Example: Consider the following schedule-7:-

Schedule-7



Is this schedule view serializable?

Solution:

Clearly this schedule is view equivalent to serial schedule $T_3T_4T_6$. Therefore, this schedule is view serializable.

6.6 Testing for Serializability

In this section we are going to study a simple and efficient method for determining conflict serializability of a schedule. This method is explained as following.

Consider a schedule S. We construct a directed graph, called a **precedence graph**, from S. This graph consists of a pair G = (V, E), where V is a set of vertices and E is a set of edges. The set of vertices consists of all the transactions participating in the schedule. The set of edges consists of all edges $T_i \rightarrow T_j$ for which one of three conditions holds:

1. T_i executes write(Q) before T_j executes read(Q).

2. T_i executes read(Q) before T_j executes write(Q).

3. T_i executes write(Q) before T_j executes write(Q).

If the precedence graph for S has a cycle, then schedule S is not conflict serializable. If the graph contains no cycles, then the schedule S is conflict serializable.

A **serializability order** of the transactions can be obtained through topological sorting, which determines a linear order consistent with the partial order of the precedence graph.

Example: Consider the following schedule-3:-

Schedule-3

T_1	T_2
read(A)	
$A := A - 50$	
write(A)	
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
read(B)	
$B := B + 50$	
write(B)	
	read(B)
	$B := B + temp$
	write(B)

Construct precedence graph for it.

Solution:

Precedence graph for above schedule will be the following:-

Precedence graph for schedule-3

Clearly this graph does not contain any cycle. Therefore, this schedule is conflict serializable.

Example: Consider the following schedule-4:-

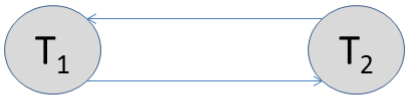
Schedule-4

T_1	T_2
read(A)	
$A := A - 50$	
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
	read(B)
write(A)	
read(B)	
$B := B + 50$	
write(B)	
	$B := B + temp$
	write(B)

Construct precedence graph for it.

Solution:

Precedence graph for above schedule will be the following:-



Precedence graph for schedule-4

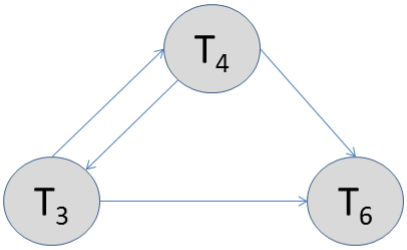
Clearly this graph contains a cycle. Therefore, this schedule is not conflict serializable.

Example: Consider the following schedule-7:-

Schedule-7

T_3	T_4	T_6
read(Q)		
	write(Q)	
write(Q)		
		write(Q)

Construct precedence graph for it.
Solution:
Precedence graph for above schedule will be the following:-



Precedence graph for schedule-7

Clearly this graph contains a cycle. Therefore, this schedule is not conflict serializable.

Note: If a schedule is conflict serializable then it is also view serializable. But converse need not be true.

6.7 Recoverability

If a transaction T_i fails, for whatever reason, we need to undo the effect of this transaction to ensure the atomicity property of the transaction. In a system that allows concurrent execution, it is necessary also to ensure that any transaction T_j that is dependent on T_i (that is, T_j has read data written by T_i) is also aborted. To achieve this surety, we need to place restrictions on the type of schedules permitted in the system.

In the following sections, we will study two schedules which are acceptable from the viewpoint of recovery from transaction failure.

6.7.1 Recoverable Schedules

A schedule is said to be recoverable schedule if for each pair of transactions T_i and T_j such that T_j reads a data item previously written by T_i, the commit operation of T_i appears before the commit operation of T_j.

Example: Is the following schedule recoverable?

T_8	T_9
read(A)	
write(A)	
	read(A)
read(B)	

Solution:
Clearly, in this schedule transaction T_9 reads the value of A written by transaction T_8, but T_9 commits before T_8. Therefore this schedule is not recoverable schedule.

6.7.2 Cascadeless Schedules

A schedule is said to be cascadeless schedule if for each pair of transactions T_i and T_j such that T_j reads a data item previously written by T_i, the commit operation of T_i appears before the read operation of T_j.

Example: Is the following schedule cascadeless?

T_{10}	T_{11}	T_{12}
read(A)		
read(B)		
write(A)		
	read(A)	
	write(A)	
		read(A)

Solution:
Clearly, in this schedule transaction T_{11} reads the value of A written by transaction T_{10} and T_{10} commits before the read operation of T_{11}.
Similarly, transaction T_{12} reads the value of A written by transaction T_{11} and T_{11} commits before the read operation of T_{12}.
Therefore this schedule is cascadeless schedule.

Note: Every cascadeless schedule is also recoverable schedule. But converse need not be true.

6.8 Exercise

1. Consider the following two transactions:

   T_1: read(A);
   read(B);
   if A = 0 then B := B + 1;
   write(B)
   T_2: read(B);
   read(A);
   if B = 0 then A := A + 1;
   write(A)

   Let the consistency requirement be A = 0 $\vee$ B = 0, with A = B = 0 the initial values.

   (a) Show that every serial execution involving these two transactions preserves the consistency of the database.

   (b) Show a concurrent execution of T_1 and T_2 that produces a non-serializable schedule.

   (c) Is there a concurrent execution of T_1 and T_2 that produces a serializable schedule?

2. Which of the following schedules is (conflict) serializable? For each serializable schedule, determine the equivalent serial schedules.

   (a) $r_1(X); r_3(X); w_1(X); r_2(X); w_3(X);$

   (b) $r_1(X); r_3(X); w_3(X); w_1(X); r_2(X);$

   (c) $r_3(X); r_2(X); w_3(X); r_1(X); w_1(X);$

   (d) $r_3(X); r_2(X); r_1(X); w_3(X); w_1(X);$

3. Consider the three transactions T_1, T_2, and T_3, and the schedules S_1 and S_2 given below. Draw the serializability (precedence) graphs for S_1 and S_2 and state whether each schedule is serializable or not. If a schedule is serializable, write down the equivalent serial schedule(s).

   T_1: $r_1(X); r_1(Z); w_1(X);$
   T_2: $r_2(Z); r_2(Y); w_2(Z); w_2(Y);$
   T_3: $r_3(X); r_3(Y); w_3(Y);$
   S_1: $r_1(X); r_2(Z); r_1(Z); r_3(X); r_3(Y); w_1(X); w_3(Y); r_2(Y); w_2(Z); w_2(Y);$
   S_2: $r_1(X); r_2(Z); r_3(X); r_1(Z); r_2(Y); r_3(Y); w_1(X); w_2(Z); w_3(Y); w_2(Y);$

6.9 AKTU previous year questions

1. What do you mean by Conflict Serializable Schedule?

2. What do you understand by ACID properties of transaction ? Explain in details.

3. Define Transaction and explain its properties with suitable example.

4. What is schedule? What are its types? Explain view serializable and cascadeless schedule with suitable example of each.

5. Which of the following schedules are conflicts serializable? For each serializable schedule find the equivalent serial schedule.
   S_1: $r_1(x); r_3(x); w_3(x); w_1(x); r_2(x)$
   S_2: $r_3(x); r_2(x); w_3(x); r_1(x); w_1(x)$
   S_3: $r_1(x); r_2(x); r_3(y); w_1(x); r_2(z); r_2(y); w_2(y)$

6. Explain I in ACID Property.

7. Define schedule.

8. What do you mean by serializability? Discuss the conflict and view serialzability with example. Discuss the testing of serializability also.

9. What do you mean by Transaction? Explain transaction property with detail and suitable example.

10. What is serializability? How it is tested?

11. State the properties of transaction.

12. What is transaction? Draw a state diagram of a transaction showing its state. Explain ACID properties of a transaction with suitable examples.

13. What are the schedules? What are the differences between conflict serialzability and view serialzability ? Explain with suitable examples what are cascadeless and recoverable schedules?

Chapter 7

Concurrency Control

When several transactions execute concurrently in the database, however, the isolation property may no longer be preserved. To ensure that it is, the system must control the interaction among the concurrent transactions. The mechanism used to control the interaction of transactions is called concurrency control scheme.
There are number of concurrency control schemes.

1. Lock based protocol
2. Time stamp based protocol
3. Validation based protocol
4. Multiple granularity protocol
5. Multi-version protocol

7.1 Lock based protocol

A lock is a mechanism to control concurrent access to a data item.
Data item can be locked in two modes.
Shared mode(S): If a transaction T_i has obtained a shared-mode lock on item Q, then T_i can read, but cannot write, Q.
Exclusive mode(S): If a transaction T_i has obtained an exclusive-mode lock on item Q, then T_i can both read and write Q.

Note: The transaction makes the lock request to the concurrency control manager. Transaction can process only after lock request is granted.

Compatibility function
Given a set of lock modes, we can define a compatibility function on them as follows.
Let A and B represent arbitrary lock modes. Suppose that a transaction T_i requests a lock of mode A on item Q on which transaction T_j ($T_i \neq T_j$) currently holds a lock of mode B. If transaction T_i can be granted a lock on Q immediately, in spite of the presence of the mode B lock, then we say mode A is compatible with mode B. Such a function can be represented conveniently by a matrix. An element comp(A, B) of the matrix has the value **true** if and only if mode A is compatible with mode B.

Compatibility Matrix

	S	X
S	true	false
X	false	false

Note:

- A transaction requests a shared lock on data item Q by executing the lock-S(Q) instruction.
- A transaction requests an exclusive lock on data item Q by executing the lock-X(Q) instruction.
- To unlock the data item Q, we use unlock(Q) instruction.

Note:
To access a data item, transaction T_i must first lock that item. If the data item is already locked by another transaction in an incompatible mode, the concurrency control manager will 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.

Example: Consider the following two transactions T_1 and T_2 with locking modes.

Transaction T_1 and T_2

T_1:
lock-X(B);
read(B);
$B := B - 50$;
write(B);
unlock(B);
lock-X(A);
read(A);
$A := A + 50$;
write(A);
unlock(A).

T_2:
lock-S(A);
read(A);
unlock(A);
lock-S(B);
read(B);
unlock(B);
display($A + B$).

Consider the following schedule-1 of these transactions.

Schedule-1

T_1	T_2	concurrency-control manager
lock-X(B)		
		grant-X(B, T_1)
read(B)		
$B := B - 50$		
write(B)		
unlock(B)		
	lock-S(A)	
		grant-S(A, T_2)
	read(A)	
	unlock(A)	
	lock-S(B)	
		grant-S(B, T_2)
	read(B)	
	unlock(B)	
	display($A + B$)	
lock-X(A)		
		grant-X(A, T_2)
read(A)		
$A := A + 50$		
write(A)		
unlock(A)		

Suppose that the values of accounts A and B are $100 and 200$, respectively. If these two transactions are executed serially, either in the order T_1, T_2 or the order T_2, T_1, then transaction T_2 will display the value \$300. If, however, these transactions are executed concurrently, then schedule 1 is possible. In this case, transaction T_2 displays \$250, which is incorrect. The reason for this mistake is that the transaction T_1 unlocked data item B too early, as a result of which T_2 saw an inconsistent state. **Example:** Consider the following two transactions T_3 and T_4 with locking modes.

Transaction T_3 and T_4

T_3: lock-X(B);
read(B);
$B := B - 50$;
write(B);
lock-X(A);
read(A);
$A := A + 50$;
write(A);
unlock(B);
unlock(A).

T_4: lock-S(A);
read(A);
lock-S(B);
read(B);
display($A + B$);
unlock(A);
unlock(B).

Consider the following schedule-1 of these transactions.

Schedule-2

T_3	T_4
lock-X(B)	
read(B)	
$B := B - 50$	
write(B)	
	lock-S(A)
	read(A)
	lock-S(B)
lock-X(A)	

Consider the partial schedule-2 for T_3 and T_4. Since T_3 is holding an exclusive-mode lock on B and T_4 is requesting a shared-mode lock on B, T_4 is waiting for T_3 to unlock B. Similarly, since T_4 is holding a shared-mode lock on A and T_3 is requesting an exclusive-mode lock on A, T_3 is waiting for T_4 to unlock A. Thus, we have arrived at a state where neither of these transactions can ever proceed with its normal execution. This situation is called deadlock.

Note: When deadlock occurs, the system must roll back one of the two transactions.

Locking protocol This is the set of rules indicating when a transaction may lock and unlock each of the data items.
Note: A schedule S is legal under a given locking protocol if S is a possible schedule for a set of transactions that follow the rules of the locking protocol.
Note: A locking protocol ensures conflict serializability if and only if all legal schedules are conflict serializable.

Starvation

Suppose a transaction T_2 has a shared-mode lock on a data item, and another transaction T_1 requests an exclusive-mode lock on the data item. Clearly, T_1 has to wait for T_2 to release the shared-mode lock. Meanwhile, a transaction T_3 may request a shared-mode lock on the same data item. The lock request is compatible with the lock granted to T_2, so T_3 may be granted the shared-mode lock. At this point T_2 may release the lock, but still T_1 has to wait for T_3 to finish. But again, there may be a new transaction T_4 that requests a shared-mode lock on the same data item, and is granted the lock before T_3 releases it. In fact, it is possible that there is a sequence of transactions that each requests a shared-mode lock on the data item, and each transaction releases the lock a short while after it is granted, but T_1 never gets the exclusive-mode lock on the data item. The transaction T_1 may never make progress, and is said to be starved. This situation is said to be **starvation**.

7.1.1 Two-phase locking protocol

This protocol requires that each transaction issue lock and unlock requests in two phases:
1. Growing phase: A transaction may obtain locks, but may not release any lock.
2. Shrinking phase: A transaction may release locks, but may not obtain any new locks.

Initially, a transaction is in the growing phase. The transaction acquires locks as needed. Once the transaction releases a lock, it enters the shrinking phase, and it can issue no more lock requests.
Example: Transactions T_3 and T_4 are locked in two phase. While, transactions T_1 and T_2 are not locked in two phase.

Note: Two-phase locking protocol ensures conflict serializability. The serializability order of transactions will be based on lock point in the transactions.

Lock point: Lock point of a transaction is a point in the schedule where the transaction has obtained its final lock (the end of its growing phase).

Note: Two-phase locking does not ensure freedom from deadlock.

Observe that transactions T_3 and T_4 are in two phase, but, in schedule 2, they are deadlocked.

Note: In addition to being serializable, schedules should be cascadeless. Cascading rollback may occur under two-phase locking.
Example: Consider the partial schedule in the following figure:-

Partial schedule under
two-phase locking protocol

T_5	T_6	T_7
lock-X(A)		
read(A)		
lock-S(B)		
read(B)		
write(A)		
unlock(A)		
	lock-X(A)	
	read(A)	
	write(A)	
	unlock(A)	
		lock-S(A)
		read(A)

Each transaction observes the two-phase locking protocol, but the failure of T_5 after the read(A) step of T_7 leads to cascading rollback of T_6 and T_7.
Note: Cascading rollbacks can be avoided by a modification of two-phase locking called the strict two-phase locking protocol.

Strict two-phase locking protocol
This protocol requires not only that locking be two phase, but also that all exclusive-mode locks taken by a transaction be held until that transaction commits.
Rigorous two-phase locking protocol
Another variant of two-phase locking is the rigorous two-phase locking protocol, which requires that all locks be held until the transaction commits.
Note: With rigorous two-phase locking, transactions can be serialized in the order in which they commit.

Lock Conversion
Upgrade: We denote conversion from shared to exclusive modes by upgrade.
Downgrade: We denote conversion from exclusive to shared by downgrade.

Note: Lock conversion cannot be allowed arbitrarily. Rather, upgrading can take place in only the growing phase, whereas downgrading can take place in only the shrinking phase.
Note: Strict two-phase locking and rigorous two-phase locking (with lock conversions) are used extensively in commercial database systems.

Note: A simple but widely used scheme automatically generates the appropriate lock and unlock instructions for a transaction, on the basis of read and write requests from the transaction:

- When a transaction T_i issues a read(Q) operation, the system issues a lock- S(Q) instruction followed by the read(Q) instruction.
- WhenT_i issues a write(Q) operation, the system checks to see whether T_i already holds a shared lock on Q. If it does, then the system issues an upgrade(Q) instruction, followed by the write(Q) instruction. Otherwise, the system issues a lock-X(Q) instruction, followed by the write(Q) instruction.
- All locks obtained by a transaction are unlocked after that transaction commits or aborts.

7.1.2 Graph-Based Protocols

For this type of protocol, we need some prior knowledge of database. To acquire such prior knowledge, we impose a partial ordering $\rightarrow$ on the set D = $\{d_1, d_2, ..., d_n\}$ of all data items. If $d_i \rightarrow d_j$, then any transaction accessing both d_i and d_j must access d_i before accessing d_j .

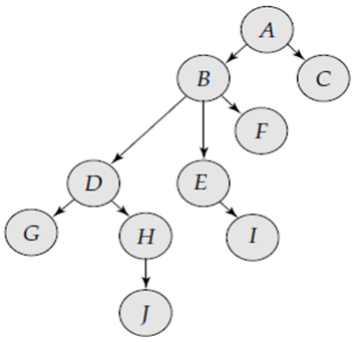
The partial ordering implies that the set D may now be viewed as a directed acyclic graph, called a database graph. Here, we will consider graph with rooted tree. Therefore, we will study tree protocol.

In the **tree protocol,** the only lock instruction allowed is lock-X. Each transaction T_i can lock a data item at most once, and must observe the following rules:

1. The first lock by T_i may be on any data item.
2. Subsequently, a data item Q can be locked by T_i only if the parent of Q is currently locked by T_i.
3. Data items may be unlocked at any time
4. A data item that has been locked and unlocked by T_i cannot subsequently be relocked by T_i.

All schedules that are legal under the tree protocol are conflict serializable.

Example: Consider the database graph of the following figure:-



The following four transactions follow the tree protocol on this graph. We show only the lock and unlock instructions:

T_{10}: lock-X(B); lock-X(E); lock-X(D); unlock(B); unlock(E); lock-X(G); unlock(D); unlock(G).
T_{11}: lock-X(D); lock-X(H); unlock(D); unlock(H).
T_{12}: lock-X(B); lock-X(E); unlock(E); unlock(B).
T_{13}: lock-X(D); lock-X(H); unlock(D); unlock(H).

One possible schedule in which these four transactions participated appears in the following figure:-

T_{10}	T_{11}	T_{12}	T_{13}
lock-X(B)			
	lock-X(D)		
	lock-X(H)		
	unlock(D)		
lock-X(E)			
lock-X(D)			
unlock(B)			
unlock(E)			
		lock-X(B)	
		lock-X(E)	
	unlock(H)		
lock-X(G)			
unlock(D)			
			lock-X(D)
			lock-X(H)
			unlock(D)
			unlock(H)
		unlock(E)	
		unlock(B)	
unlock (G)			

Observe that the schedule in this figure is conflict serializable. It can be shown not only that the tree protocol ensures conflict serializability, but also that this protocol ensures freedom from deadlock.
The tree protocol in this figure does not ensure recoverability and cascadelessness.

Advantage:

1. The tree-locking protocol has an advantage over the two-phase locking protocol in that, unlike two-phase locking, it is deadlock-free, so no rollbacks are required.

2. The tree-locking protocol has another advantage over the two-phase locking protocol in that unlocking may occur earlier. Earlier unlocking may lead to shorter waiting times, and to an increase in concurrency.

7.1.3 Timestamp-Based Protocols

Timestamps With each transaction T_i in the system, we associate a unique fixed timestamp, denoted by TS(T_i). This timestamp is assigned by the database system before the transaction Ti starts execution. If a transaction T_i has been assigned timestamp TS(T_i),

and a new transaction T_j enters the system, then TS(T_i) ¡ TS(T_j). There are two simple methods for implementing this scheme:

1. Use the value of the system clock as the timestamp; that is, a transaction's timestamp is equal to the value of the clock when the transaction enters the system.

2. Use a logical counter that is incremented after a new timestamp has been assigned; that is, a transaction's timestamp is equal to the value of the counter when the transaction enters the system.

The timestamps of the transactions determine the serializability order. Thus, if TS(T_i) ¡ TS(T_i), then the system must ensure that the produced schedule is equivalent to a serial schedule in which transaction T_i appears before transaction T_j.

To implement this scheme, we associate with each data item Q two timestamp values:

- **W-timestamp(Q)** denotes the largest timestamp of any transaction that executed write(Q) successfully.

- **R-timestamp(Q)** denotes the largest timestamp of any transaction that executed read(Q) successfully.

These timestamps are updated whenever a new read(Q) or write(Q) instruction is executed.

Timestamp-Ordering Protocol

The timestamp-ordering protocol ensures that any conflicting read and write operations are executed in timestamp order. This protocol operates as follows:

1. Suppose that transaction T_i issues read(Q).

   (a) If TS(T_i) $<$ W-timestamp(Q), then the read operation is rejected, and T_i is rolled back.

   (b) If TS(T_i) $\geq$ W-timestamp(Q), then the read operation is executed, and R-timestamp(Q) is set to the maximum of R-timestamp(Q) and TS(T_i).

2. Suppose that transaction T_i issues write(Q).

   (a) If TS(T_i) $<$ R-timestamp(Q), then the system rejects the write operation and rolls T_i back.

   (b) If TS(Ti) $<$ W-timestamp(Q), then the system rejects this write operation and rolls T_i back.

   (c) Otherwise, the system executes the write operation and sets W-timestamp(Q) to TS(T_i).

If a transaction T_i is rolled back by the concurrency-control scheme as result of issuance of either a read or write operation, the system assigns it a new timestamp and restarts it.

Example: Consider transactions T_{14} and T_{15}. Transaction T_{14} displays the contents of

accounts A and B:

T_{14}: read(B);
read(A);
display(A + B).

Transaction T_{15} transfers $50 from account A to account B, and then displays the contents of both:

T_{15}: read(B);
B := B - 50;
write(B);
read(A);
A := A + 50;
write(A);
display(A + B).

Following schedule is possible under timestamp ordering protocol.

T_{14}	T_{15}
read(B)	
	read(B)
	$B := B - 50$
	write(B)
read(A)	
	read(A)
display($A + B$)	
	$A := A + 50$
	write(A)
	display($A + B$)

Note:

1. The timestamp-ordering protocol ensures conflict serializability.
2. This protocol also ensures freedom from deadlock.
3. There is a possibility of starvation.
4. This protocol can generate schedules that are not recoverable.

Thomas' Write Rule

The modification to the timestamp-ordering protocol, called Thomas' write rule, is this: Suppose that transaction Ti issues write(Q).

1. If TS(T_i) ¡ W-timestamp(Q), then the read operation is rejected, and T_i is rolled back.

2. If TS(T_i) ¡ W-timestamp(Q), then T_i is attempting to write an obsolete value of Q. Hence, this write operation can be ignored.

3. Otherwise, the system executes the write operation and sets W-timestamp(Q) to TS(T_i).

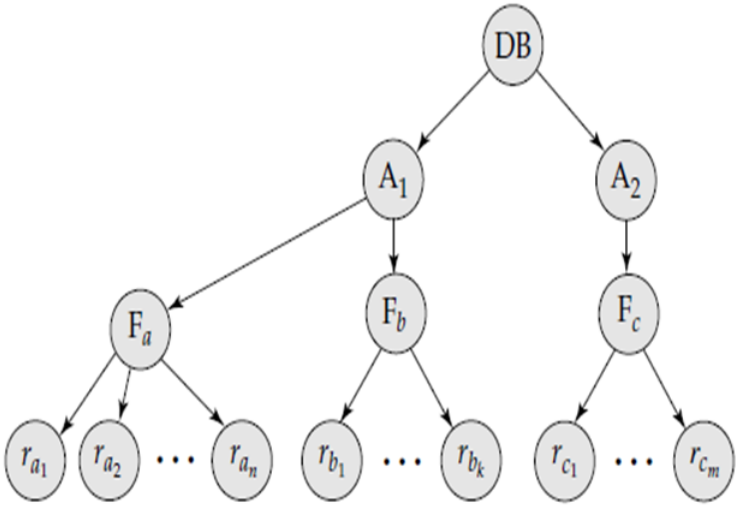
Example: Consider following schedule:-

T_{16}	T_{17}
read(Q)	
	write(Q)
write(Q)	

Clearly, this schedule is not conflict serializable and, thus, is not possible under any of two-phase locking, the tree protocol, or the timestamp-ordering protocol. Under Thomas' write rule, the write(Q) operation of T_{16} would be ignored. The result is a schedule that is view equivalent to the serial schedule ¡ T_{16}, T_{17}¿.

7.2 Multiple Granularity

Consider the following granularity hierarchy.



This tree consists of four levels of nodes. The highest level represents the entire database. Below it are nodes of type area; the database consists of exactly these areas. Each area in turn has nodes of type file as its children. Each area contains exactly those files that are its child nodes. No file is in more than one area. Finally, each file has nodes of type record. As before, the file consists of exactly those records that are its child nodes, and no record can be present in more than one file.

This protocol uses the following compatibility matrix to lock the data items.

Lock Compatibility Matrix

	IS	IX	S	SIX	X
IS	true	true	true	true	false
IX	true	true	false	false	false
S	true	false	true	false	false
SIX	true	false	false	false	false
X	false	false	false	false	false

There is an intention mode associated with shared mode, and there is one with exclusive mode. If a node is locked in intention-shared (IS) mode, explicit locking is being done at a lower level of the tree, but with only shared-mode locks. Similarly, if a node is locked in intention-exclusive (IX) mode, then explicit locking is being done at a lower level, with exclusive-mode or shared-mode locks. Finally, if a node is locked in shared and intention-exclusive (SIX) mode, the sub-tree rooted by that node is locked explicitly in shared mode, and that explicit locking is being done at a lower level with exclusive-mode locks.

The multiple-granularity locking protocol, which ensures serializability, is this: Each transaction T_i can lock a node Q by following these rules:

1. It must observe the lock-compatibility function shown in above matrix.

2. It must lock the root of the tree first, and can lock it in any mode.

3. It can lock a node Q in S or IS mode only if it currently has the parent of Q locked in either IX or IS mode.

4. It can lock a node Q in X, SIX, or IX mode only if it currently has the parent of Q locked in either IX or SIX mode.

5. It can lock a node only if it has not previously unlocked any node (that is, T_i is two phase).

6. It can unlock a node Q only if it currently has none of the children of Q locked.

Clearly, the multiple-granularity protocol requires that locks be acquired in top-down (root-to-leaf) order, whereas locks must be released in bottom-up (leaf-to-root) order.

Example:
Consider the tree shown in the above figure and these transactions:

- Suppose that transaction T_{18} reads record r_{a_2} in file F_a. Then, T_{18} needs to lock the database, area A_1, and F_a in IS mode (and in that order), and finally to lock r_{a_2} in S mode.

- Suppose that transaction T_{19} modifies record r_{a_9} in file F_a. Then, T_{19} needs to lock the database, area A_1, and file F_a in IX mode, and finally to lock r_{a_2} in X mode.

- Suppose that transaction T_{20} reads all the records in file F_a. Then, T_{20} needs to lock the database and area A_1 (in that order) in IS mode, and finally to lock F_a in S mode.

- Suppose that transaction T_{21} reads the entire database. It can do so after locking the database in S mode.

Clearly, transactions T_{18}, T_{20}, and T_{21} can access the database concurrently. Transaction T_{19} can execute concurrently with T_{18}, but not with either T_{20} or T_{21}.

This protocol enhances concurrency and reduces lock overhead. It is particularly useful in applications that include a mix of

- Short transactions that access only a few data items

- Long transactions that produce reports from an entire file or set of files

Note: Deadlock is possible in this protocol.

7.3 Multiversion Schemes

In multiversion concurrency control schemes, each write(Q) operation creates a new version of Q. When a transaction issues a read(Q) operation, the concurrencycontrol manager selects one of the versions of Q to be read. The concurrency-control scheme must ensure that the version to be read is selected in a manner that ensures serializability.

7.3.1 Multiversion Timestamp Ordering

With each data item Q, a sequence of versions $< Q_1, Q_2, ..., Q_m >$ is associated. Each version Q_k contains three data fields:

- **Content** is the value of version Q_k.

- **W-timestamp(Q_k)** is the timestamp of the transaction that created version Q_k.

- **R-timestamp(Q_k)** is the largest timestamp of any transaction that successfully read version Q_k.

A transaction T_i creates a new version Q_k of data item Q by issuing a write(Q) operation. The content field of the version holds the value written by T_i. The system initializes the W-timestamp and R-timestamp to TS(T_i). It updates the R-timestamp value of Q_k whenever a transaction T_j reads the content of Q_k, and R-timestamp(Q_k) ¡ TS(T_j).

The multiversion timestamp-ordering scheme operates as follows. Suppose that transaction T_i issues a read(Q) or write(Q) operation. Let Q_k denote the version of Q whose write timestamp is the largest write timestamp less than or equal to TS(T_i).

1. If transaction T_i issues a read(Q), then the value returned is the content of version Q_k.

2. If transaction T_i issues write(Q), and if TS(T_i)¡R-timestamp(Q_k), then the system rolls back transaction T_i. On the other hand, if TS(T_i) = W-timestamp(Q_k), the system overwrites the contents of Q_k; otherwise it creates a new version of Q.

Versions that are no longer needed are removed according to the following rule. Suppose that there are two versions, Q_k and Q_j , of a data item, and that both versions have a W-timestamp less than the timestamp of the oldest transaction in the system. Then, the older of the two versions Q_k and Q_j will not be used again, and can be deleted.

Note:

1. The multiversion timestamp-ordering scheme ensures serializability.

2. The multiversion timestamp-ordering scheme does not ensure recoverability and cascadelessness.

7.4 Deadlock Handling

A system is in a deadlock state if there exists a set of transactions such that every transaction in the set is waiting for another transaction in the set. More precisely, there exists a set of waiting transactions $\{T_0, T_1, ..., T_n\}$ such that T_0 is waiting for a data item that T_1 holds, and T_1 is waiting for a data item that T_2 holds, and . . ., and T_{n-1} is waiting for a data item that T_n holds, and T_n is waiting for a data item that T_0 holds. None of the transactions can make progress in such a situation.

There are two principal methods for dealing with the deadlock problem. We can use a deadlock prevention protocol to ensure that the system will never enter a deadlock state. 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.

Note: Prevention is commonly used if the probability that the system would enter a deadlock state is relatively high; otherwise, detection and recovery are more efficient.

7.4.1 Deadlock Prevention

Two different deadlock prevention schemes using timestamps have been proposed:

1. **wait–die:** This scheme is a non-preemptive technique. When transaction T_i requests a data item currently held by T_j , T_i is allowed to wait only if it has a timestamp smaller than that of T_j (that is, T_i is older than T_j). Otherwise, T_i is rolled back (dies).
   For example, suppose that transactions T_1, T_2, and T_3 have timestamps 5, 10, and 15, respectively. If T_1 requests a data itemheld by T_2, then T_1 will wait. If T_3 requests a data item held by T_2, then T_3 will be rolled back.

2. **wound–wait:** This scheme is a preemptive technique. It is a counterpart to the wait–die scheme. When transaction T_i requests a data item currently held by T_j , T_i is allowed to wait only if it has a timestamp larger than that of T_j (that is, T_i is younger than T_j). Otherwise, T_j is rolled back (T_j is wounded by T_i).

Returning to our example, with transactions T_1, T_2, and T_3, if T_1 requests a data item held by T_2, then the data item will be preempted from T_2, and T_2 will be rolled back. If T_3 requests a data item held by T_2, then T_3 will wait.

7.4.2 Deadlock Detection and Recovery

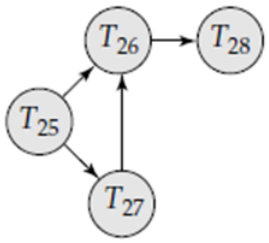
If a system does not employ some protocol that ensures deadlock freedom, then a detection and recovery scheme must be used. An algorithm that examines the state of the system is invoked periodically to determine whether a deadlock has occurred. If one has, then the system must attempt to recover from the deadlock.

Deadlock Detection

To identify the deadlock is present in the system, we use a directed graph called **wait-for-graph**.
In this graph, vertices are corresponding to transactions. When transaction T_i requests a data item currently being held by transaction T_j , then the edge $T_i \rightarrow T_j$ is inserted in the wait-for graph. This edge is removed only when transaction $T_j$is no longer holding a data item needed by transaction T_i.

A deadlock exists in the system if and only if the wait-for graph contains a cycle. Each transaction involved in the cycle is said to be deadlocked. To detect deadlocks, the system needs to maintain the wait-for graph, and periodically to invoke an algorithm that searches for a cycle in the graph.

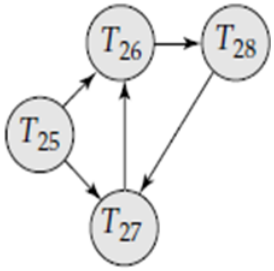
Example: Consider the wait-for graph show in the following figure,



which depicts the following situation:

- Transaction T_{25} is waiting for transactions T_{26} and T_{27}.
- Transaction T_{27} is waiting for transaction T_{26}.
- Transaction T_{26} is waiting for transaction T_{28}.

Since the graph has no cycle, the system is not in a deadlock state.
Suppose now that transaction T_{28} is requesting an item held by T_{27}. The edge $T_{28} \rightarrow T_{27}$is added to the wait-for graph, resulting in the new system state in following figure.



This time, the graph contains the cycle
$T_{26} \rightarrow T_{28} \rightarrow T_{27} \rightarrow T_{26}$.
implying that transactions T_{26}, T_{27}, and T_{28} are all deadlocked.

Recovery from Deadlock

When a detection algorithm determines that a deadlock exists, the system must recover from the deadlock. The most common solution is to roll back one or more transactions to break the deadlock. Three actions need to be taken:

1. **Selection of a victim:** Given a set of deadlocked transactions, we must determine which transaction (or transactions) to roll back to break the deadlock.We should roll back those transactions that will incur the minimum cost. Unfortunately, the term minimum cost is not a precise one. Many factors may determine the cost of a rollback, including

   (a) How long the transaction has computed, and how much longer the transaction will compute before it completes its designated task.

   (b) How many data items the transaction has used.

   (c) How many more data items the transaction needs for it to complete.

   (d) How many transactions will be involved in the rollback.

2. **Rollback:** Once we have decided that a particular transaction must be rolled back, we must determine how far this transaction should be rolled back.
   The simplest solution is a total rollback:

3. **Starvation:** In a system where the selection of victims is based primarily on cost factors, it may happen that the same transaction is always picked as a victim. As a result, this transaction never completes its designated task, thus there is starvation. We must ensure that transaction can be picked as a victim only a (small) finite number of times. The most common solution is to include the number of rollbacks in the cost factor.

7.4.3 The Phantom Phenomenon

Consider transaction T_{29} that executes the following SQL query on the bank database:

```
select sum(balance)
from account
where branch-name = 'Perryridge'
```

Transaction T_{29} requires access to all tuples of the account relation pertaining to the Perryridge branch.
Let T_{30} be a transaction that executes the following SQL insertion:

```
insert into account
values (A-201, 'Perryridge', 900)
```

Let S be a schedule involving T_{29} and T_{30} . We expect there to be potential for a conflict for the following reason:

- If T_{29} uses the tuple newly inserted by T_{30} in computing sum(balance), then T_{29} read a value written by T_{30}. Thus, in a serial schedule equivalent to S, T_{30} must come before T_{29}.

- If T_{29} does not use the tuple newly inserted by T_{30} in computing sum(balance), then in a serial schedule equivalent to S, T_{29} must come before T_{30}.

The second of these two cases is curious. T_{29} and T_{29} do not access any tuple in common, yet they conflict with each other! In effect, T_{29} and T_{29} conflict on a phantom tuple. If concurrency control is performed at the tuple granularity, this conflict would go undetected. This problem is called the **phantom phenomenon**.
To prevent the phantom phenomenon, we allow T_{29} to prevent other transactions from creating new tuples in the account relation with branch-name = "Perryridge."

7.5 AKTU Examination Questions

1. Define Concurrency Control.

2. Explain the phantom phenomena. Discuss a Time Stamp Protocol that avoids the phantom phenomena.

3. Discuss about deadlock prevention schemes.

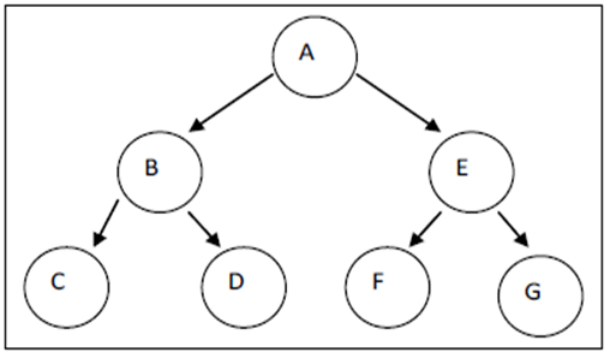
4. Explain Concurrency Control. Why it is needed in database system?

5. What is deadlock? What are necessary conditions for it? How it can be detected and recovered?

6. Explain two phase locking protocol with suitable example.

7. Write the salient features of graph based locking protocol with suitable example.

8. What do you mean by multiple granularity? How the concurrency is maintained in this case. Write the concurrent transactions for the following graph.



- T_1 wants to access Item C in read mode
- T_2 wants to access item D in Exclusive mode
- T_3 wants to read all the children of item B
- T_4 wants to access all items in read mode

9. Define Exclusive Lock.
10. What is Two phase Locking (2PL)? Describe with the help of example.
11. What are multi version schemes of concurrency control? Describe with the help of an example. Discuss the various Time stamping protocols for concurrency control also.
12. Define timestamp.
13. Discuss about the deadlock prevention schemes.
14. Explain the following protocols for concurrency control.
    i) Lock based protocols
    ii) Time Stamp based protocols
15. What are the pitfalls of lock-based protocol?
16. Describe major problems associated with concurrent processing with examples. What is the role of locks in avoiding these Problems.
17. Explain the phantom phenomenon. Devise a time stamp based protocol that avoids the phantom phenomenon.
18. What do you mean by multiple granularties? How it is implemented in transaction system?