Dr. DEEPAK NEDUNURI

ASSOCIATE PROFESSOR
DEPARTMENT OF CSE

Chapter-1
1. AN OVERVIEW OF DATABASE MANAGEMENT SYSTEMS

1. OVERVIEW OF DATABASE SYSTEMS:

Database:
→A **database** is a collection of data, typically describing the activities of one or more related organizations.
→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.

DATABASE MANAGEMENT SYSTEMS (DBMS):
→A **database management system**, or **DBMS**, is software designed to assist in maintaining and utilizing large collections of data, and the need for such systems, as well as their use, is growing rapidly.

→A **DBMS** is a collection of interrelated data and a set of programs to access those data. **DBMS** is a system which allows inserting, deleting and updating database. The primary goal of a **DBMS** is to provide a way to store and retrieve database information that is both convenient and efficient.

Database + Management System = DBMS.

→**DBMS** is the program that organizes and maintains the information and database application is the program that makes it possible to view, retrieve and update information stored in the DBMS.

→**Oracle, Microsoft Access 2000, FoxPro by Microsoft, dBase V by Borland, Paradox by Borland** etc., are the various **DBMS packages.**

DATABASE SYSTEM APPLICATIONS:
1. Banking: This application is very useful for maintain customer information, accounts, loans and banking transactions.
2. Universities: DBMS is used for maintaining student's records, course registration and grades in any university.
3. Airlines: DBMS is used for reservation and schedule information.
4. Railway Reservation: DBMS is used for checking the availability of reservation in different trains, tickets etc.
5. Finance: DBMS is used for storing information about holidays, sales and purchase of financial instruments.
6. Telecommunication: DBMS is used for keeping records of calls made, generating monthly bills etc.
7. Sales: DBMS is used for customer, product and purchase information.
8. Credit Card Transaction: DBMS is used for purchases on credit card and generation of monthly statements.

9. Human Resources: For information about employees, salaries, payroll taxes and benefits and for generation of paychecks.

2. FILE SYSTEM VS. DBMS: (******************)

→**DBMS (Database Management System)** and **File System** are two ways that could be used to manage, store, retrieve and manipulate data.

→A **File System** is a collection of raw data files stored in the hard-drive whereas **DBMS** is a bundle of applications that is dedicated for managing data stored in databases. It is the integrated system used for managing digital databases, which allows the storage of database content, creation/ maintenance of data, search and other functionalities. Both systems can be used to allow the user to work with data in a similar way. A File System is one of the earliest ways of managing data. But due the shortcomings present in using a File System to store electronic data, Database Management Systems came in to use sometime later, as they provide mechanisms to solve those problems. But it should be noted that, even in a DBMS, data are eventually (physically) stored in some sort of files.

Differences between FS vs DBMS:

(Disadvantages of File system over DBMS)

1. Data Redundancy:
→ In FMS, as various copies of same data resides, the same information is duplicated in several files indicating the redundancy [waste copies] of data, which leads to higher storage and access cost.

For example: The addresses of customers will be present in the file maintaining information about customers holding savings account and also the address of the customers will be present in file maintaining the current account. Even when same customers have a saving account and current account his address will be present at two places.
→ In **DBMS**, as only one copy of data record resides, the same information may not be duplicated avoiding the redundancy of data, leading to minimum storage and access cost.
2. Data Inconsistency:
→In FMS, data redundancy leads to greater problem than just wasting the storage i.e. it may lead to inconsistent data. Same data which has been repeated at several places may not match after it has been updated at some places.

For example: Suppose the customer requests to change the address for his account in the Bank and the Program is executed to update the saving bank account file only but his current bank account file is not updated. Afterwards the addresses of the same customer present in saving bank account file and current bank account file will not match. Moreover there will be no way to find out which address is latest out of these two.

→In **DBMS** the related data resides in the same storage location minimizing data inconsistency.

3. Difficult in Accessing Data:
→In FMS, for generating ad hoc reports the programs will not already be present and only options present will to write a new program to generate requested report or to work manually. This is going to take impractical time and will be more expensive.

For example: Suppose all of sudden the administrator gets a request to generate a list of all the customers holding the saving banks account who lives in particular locality of the city.

Administrator will not have any program already written to generate that list but say he has a program which can generate a list of all the customers holding the savings account. Then he can either provide the information by going thru the list manually to select the customers living in the particular locality or he can write a new program to generate the new list. Both of these ways will take large time which would generally be impractical.

→ In **DBMS**, to access the appropriate data, the DBMS consists of one or more programs to extract the needed information.

<u>4. Data Isolation:</u>
→ In **FMS**, as data is scattered in different formats, it is difficult to write new programs to retrieve the appropriate data.

For example: Suppose the Address in Saving Account file have fields: ***Add line1, Add line2, City, State, Pin*** while the fields in address of Current account are: ***House No., Street No., Locality*, City, State, Pin**. Administrator is asked to provide the list of customers living in a particular locality. Providing consolidated list of all the customers will require looking in both files. But they both have different way of storing the address. Writing a program to generate such a list will be difficult.

→ In **DBMS**, as data resides in same storage location i.e., related files in different groups, it is easy to write new programs to retrieve the appropriate data.

<u>5. Integrity Problems:</u>
→ In **FMS**, to develop the new consistent range in the existing system, the appropriate code must be added in various application problems.

For example: An account should not have balance less than Rs. 500. To enforce this constraint appropriate check should be added in the program which add a record and the program which withdraw from an account. Suppose later on this amount limit is increased then all those check should be updated to avoid inconsistency. These time to time changes in the programs will be great headache for the administrator.

→In **DBMS**, to develop the new consistent range in the existing system, the appropriate code must be added in one application program that access all data at one time.

<u>6. Concurrent Access:</u>
→In FMS, when more than one users are allowed to process the database. If in that environment two or more users try to update a shared data element at about the same time then it may result into inconsistent data.

For example: Suppose Balance of an account is Rs. 500. And User A and B try to withdraw Rs 100 and Rs 50 respectively at almost the same time using the Update process.

Update:
1. Read the balance amount.
2. Subtract the withdrawn amount from balance.
3. Write updated Balance value.

Suppose A performs Step 1 and 2 on the balance amount i.e., it reads 500 and subtract100 from it. But at the same time B withdraws Rs 50 and he performs the Update process and he also reads the balance as 500 subtract 50 and writes back 450. User A will also write his updated Balance amount as 400. They may update the Balance value in any order depending on various reasons concerning to system being used by both of the users. So finally the balance will be either equal to 400 or 450. Both of these values are wrong for the updated balance and so now the balance amount is having inconsistent value forever.

→In **DBMS**, to improve the overall performance of the data management, the multiple users are allowed to update the data concurrently where same data file is accessed by multiple users and the last change remains permanent.

7. Atomicity:
→ In FMS, on subject to the failure of the computer system, it is crucial to ensure that the processing data are restored to the consistent state that exist prior to the failure or not i.e., the change must be atomic, it must happen entirely or not at all.

→In DBMS, on subject to the failure of computer system, the software ensures that the processing data are restored to the consistent state that exist prior to the failure i.e., the change does not happen at all.

8. Security and Access Control:
→In FMS, database should be protected from unauthorized users. Every user should not be allowed to access every data. Since application programs are added to the system For example: The Payroll Personnel in a bank should not be allowed to access accounts information of the customers.

→In DBMS, as a security property, every user of DBMS is able to access only needed data and strictly not allowing accessing all the data.

3. ADVANTAGES OF DBMS(*******)

1. Data Independence: Application programs should be as independent as possible from details of data representation and storage. The DBMS can provide an abstract view of the data to insulate application code from such details.

2. Efficient Data Access: A DBMS utilizes a variety of sophisticated techniques to store and retrieve data efficiently. This feature is especially important if the data is stored on external storage devices.

3. Data Integrity: Integrity constraints such as unique account number for each person should be identified. Data may be updated incorrectly, if integrity is not maintained.
There are two types of integrity constraints,
i) Primary integrity or primary key = Not null + Unique values.
ii) Referential integrity or Foreign key = Value from the primary key + Duplicate values may be there.

4. Security: Different constraints can be there for each type of access right to each module of information in database, thus ensuring that access to database is through proper channel by proper person.

5. Less Redundancy: Database Administrator (DBA) has the centralized control of the data, so it will avoid the inconsistency because now we have to make the changes at one place only.

6. Data Sharing: Many application programs use the same data simultaneously and the referred to as data sharing. Due to this feature of database, the storage capacity is reduced to minimum. These advantages make database system very popular.

7. Concurrent Access and Crash recovery: Database has backup so that recover of data from software and hardware failure can be made, such that if system crashes in mid of transactions database should be restored to the safe state and concurrency is maintained.

8. Reduced Application Development Time: DBMS supports many important functions that are common to many applications accessing data stored in the DBMS.

Chapter-2
INTRODUCTION TO DATABASE SYSTEM ARCHITECTURE

Chapter-2: Database system architecture, Introduction- The Three Levels of Architecture-The External Level- the Conceptual Level- the Internal Level-Mapping- the Database Administrator-The Database Management Systems-Client/Server Architecture.

1. DATA MODELS: (***************)

→A **data model** is a collection of high level data description constructs that hide many low-level storage details.

→A ***data model*** is a collection of concepts for describing data.
→Underlying the structure of a data base is the data model. The collection of conceptual tools for describing data, data relationships, and data semantics.

→**Data models** define how the logical structure of a database is modelled. Data Models are fundamental entities to introduce abstraction in a DBMS. Data models define how data is connected to each other and how they are processed and stored inside the system.

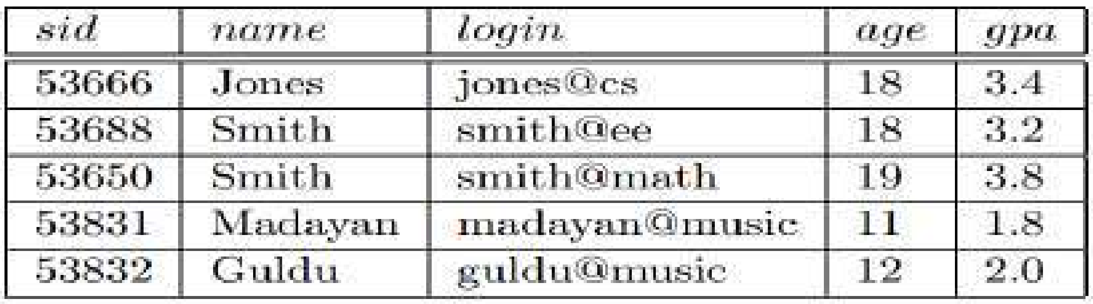
The Relational model:

→A description of data in terms of a data model is called a **schema**. In the relational model, the schema for a relation specifies its **name**, the name of each **field** (or **attribute** or **column**), and the type of each field.

→**As an example**, student information in a university database may be stored in a relation with the following **schema:**

Students (sid: string, name: string, login: string, age: integer, gpa: real)

→The preceding schema says that each record in the Students relation has five fields, with field names and types as indicated.2, an example instance of the Students relation appears in **Figure 1.1**.

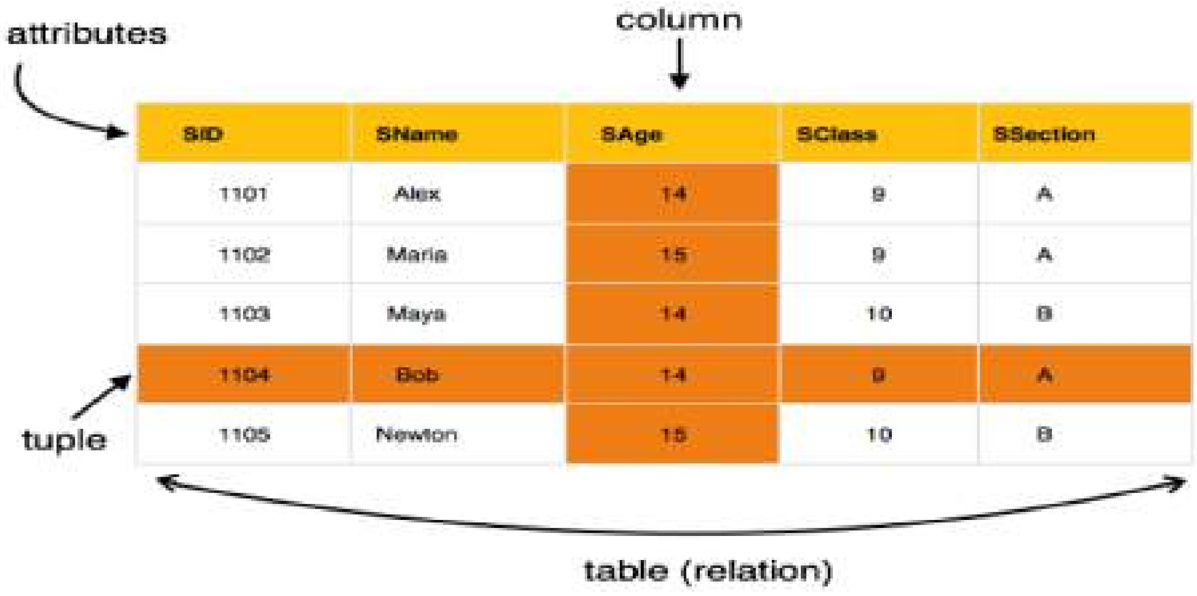


sid	*name*	*login*	*age*	*gpa*
53666	Jones	jones@cs	18	3.4
53688	Smith	smith@ee	18	3.2
53650	Smith	smith@math	19	3.8
53831	Madayan	madayan@music	11	1.8
53832	Guldu	guldu@music	12	2.0

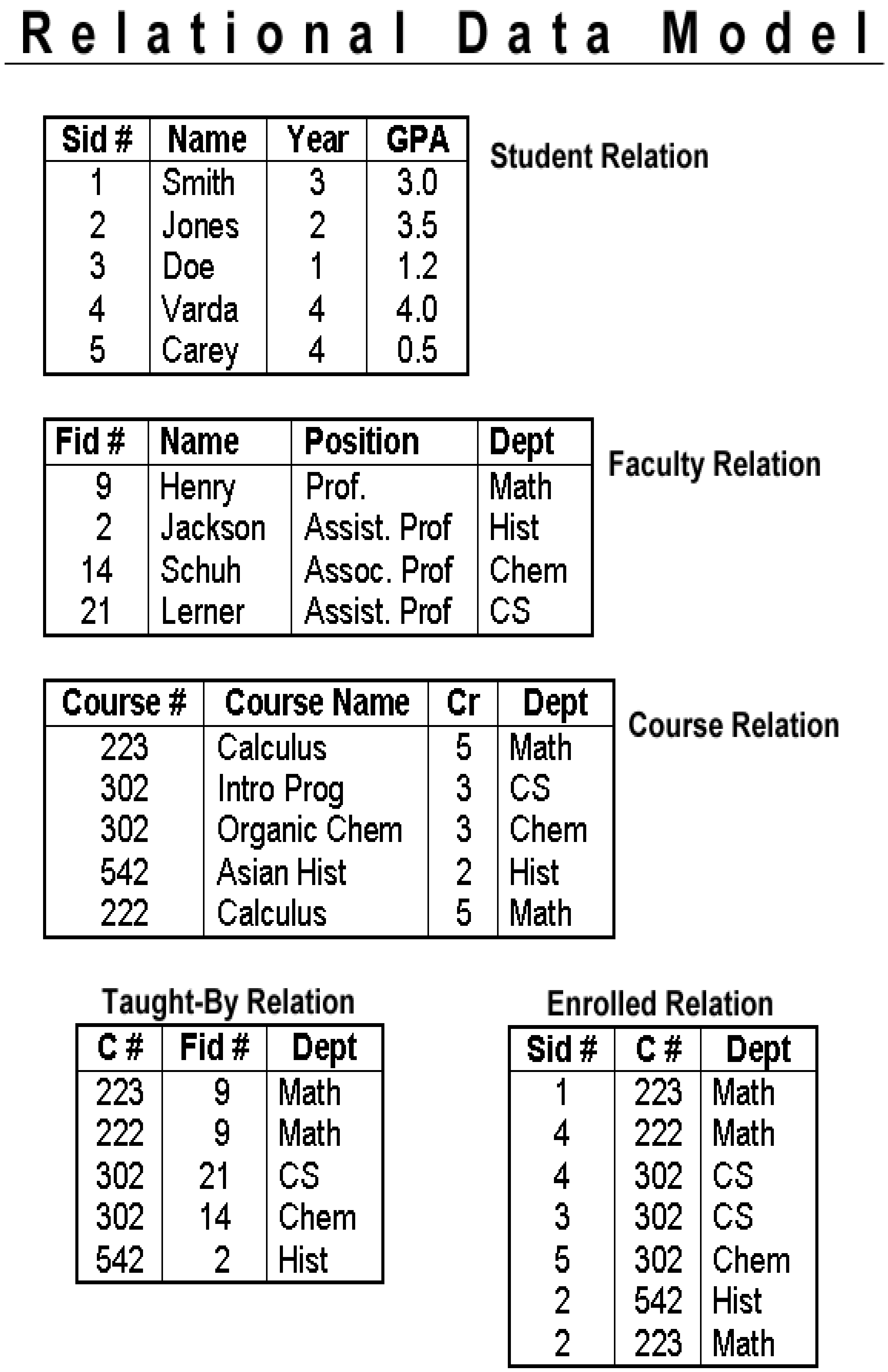
→In **relational data model**, data exists in two dimensional tables known as relations. A relation (table) consists of unique attributes (columns) and tuples (rows).
→The description of data in terms of tables is called as **relations.**

Chapter-2
INTRODUCTION TO DATABASE SYSTEM ARCHITECTURE

→Consider the following **relational model** shown in **tables.**



SID	SName	SAge	SClass	SSection
1101	Alex	14	9	A
1102	Maria	15	9	A
1103	Maya	14	10	B
1104	Bob	14	9	A
1105	Newton	15	10	B



Relational Data Model

Student Relation

Sid #	Name	Year	GPA
1	Smith	3	3.0
2	Jones	2	3.5
3	Doe	1	1.2
4	Varda	4	4.0
5	Carey	4	0.5

Faculty Relation

Fid #	Name	Position	Dept
9	Henry	Prof.	Math
2	Jackson	Assist. Prof	Hist
14	Schuh	Assoc. Prof	Chem
21	Lerner	Assist. Prof	CS

Course Relation

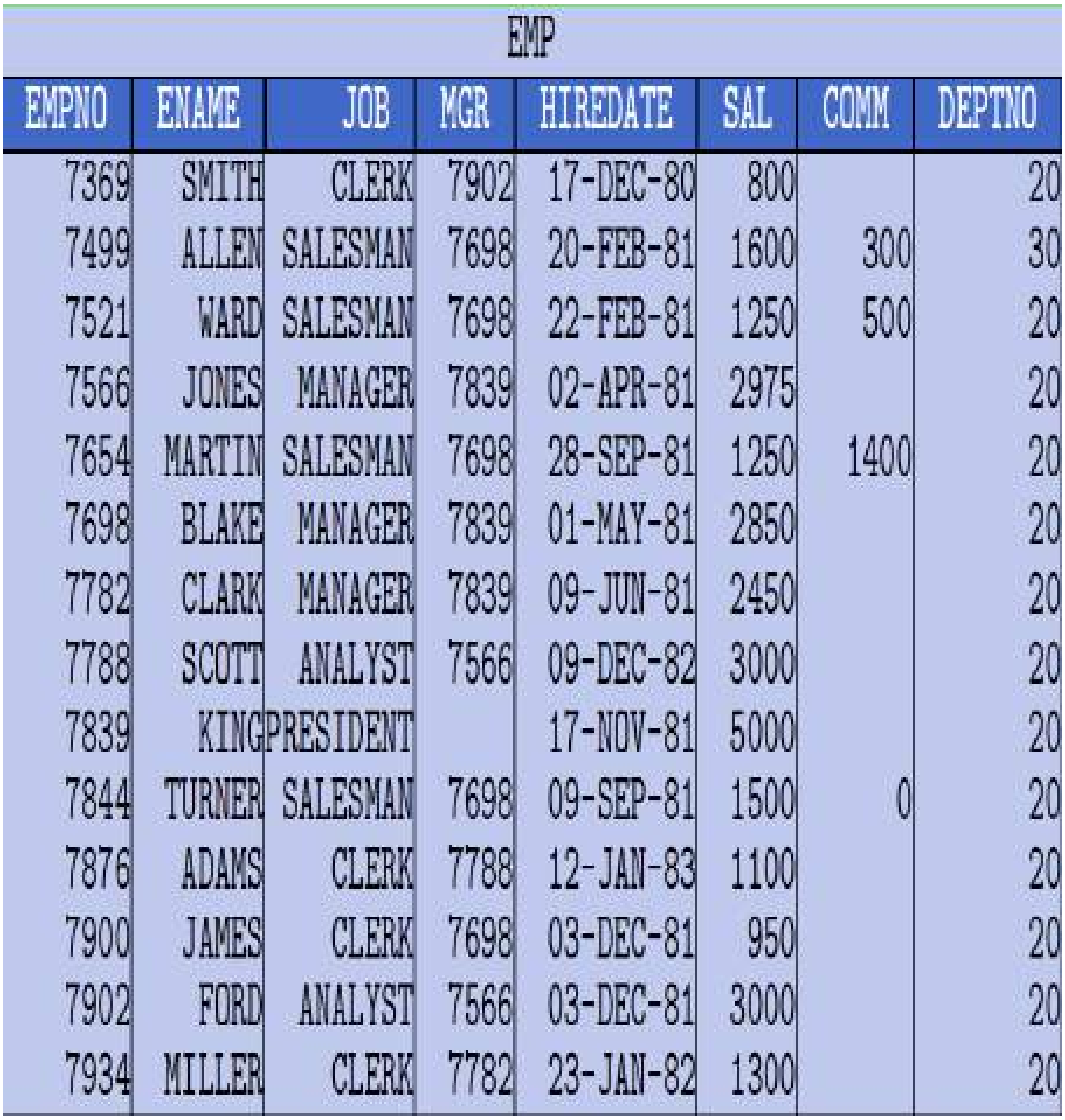
Course #	Course Name	Cr	Dept
223	Calculus	5	Math
302	Intro Prog	3	CS
302	Organic Chem	3	Chem
542	Asian Hist	2	Hist
222	Calculus	5	Math

Taught-By Relation

C #	Fid #	Dept
223	9	Math
222	9	Math
302	21	CS
302	14	Chem
542	2	Hist

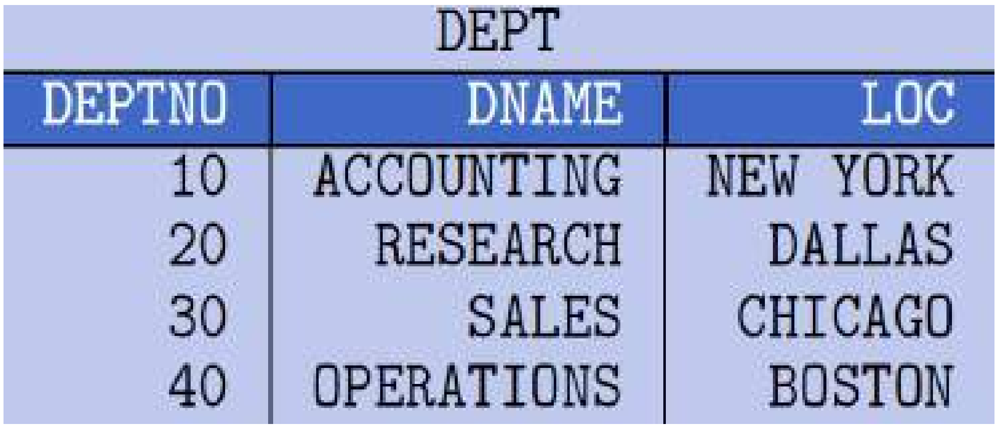
Enrolled Relation

Sid #	C #	Dept
1	223	Math
4	222	Math
4	302	CS
3	302	CS
5	302	Chem
2	542	Hist
2	223	Math



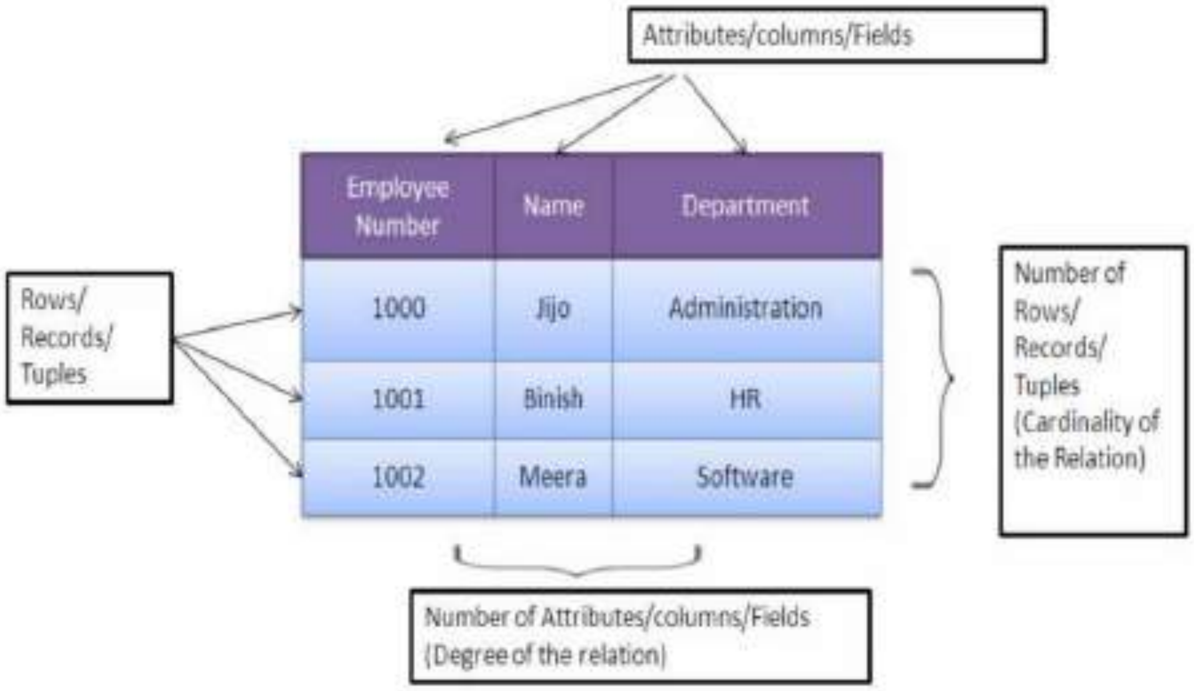
EMP

EMPNO	ENAME	JOB	MGR	HIREDATE	SAL	COMM	DEPTNO
7369	SMITH	CLERK	7902	17-DEC-80	800		20
7499	ALLEN	SALESMAN	7698	20-FEB-81	1600	300	30
7521	WARD	SALESMAN	7698	22-FEB-81	1250	500	20
7566	JONES	MANAGER	7839	02-APR-81	2975		20
7654	MARTIN	SALESMAN	7698	28-SEP-81	1250	1400	20
7698	BLAKE	MANAGER	7839	01-MAY-81	2850		20
7782	CLARK	MANAGER	7839	09-JUN-81	2450		20
7788	SCOTT	ANALYST	7566	09-DEC-82	3000		20
7839	KING	PRESIDENT		17-NOV-81	5000		20
7844	TURNER	SALESMAN	7698	09-SEP-81	1500	0	20
7876	ADAMS	CLERK	7788	12-JAN-83	1100		20
7900	JAMES	CLERK	7698	03-DEC-81	950		20
7902	FORD	ANALYST	7566	03-DEC-81	3000		20
7934	MILLER	CLERK	7782	23-JAN-82	1300		20



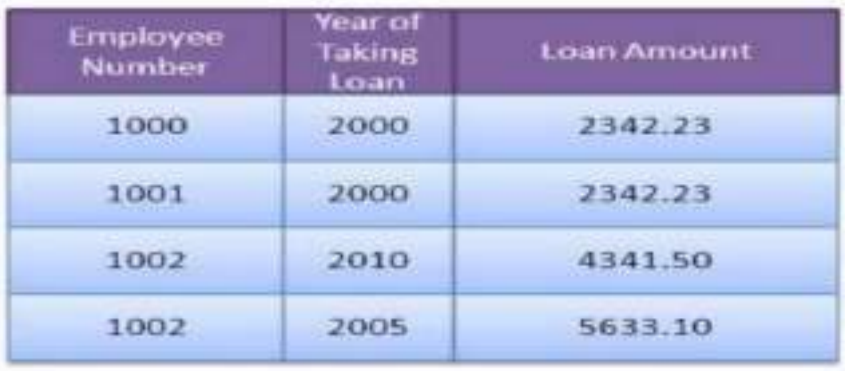
DEPT

DEPTNO	DNAME	LOC
10	ACCOUNTING	NEW YORK
20	RESEARCH	DALLAS
30	SALES	CHICAGO
40	OPERATIONS	BOSTON


Attributes/columns/Fields
Rows/ Records/ Tuples
Number of Rows/ Records/ Tuples (Cardinality of the Relation)
Number of Attributes/columns/Fields (Degree of the relation)

Employee Number	Name	Department
1000	Jijo	Administration
1001	Binish	HR
1002	Meera	Software

employee_details table



Employee Number	Year of Taking Loan	Loan Amount
1000	2000	2342.23
1001	2000	2342.23
1002	2010	4341.50
1002	2005	5633.10

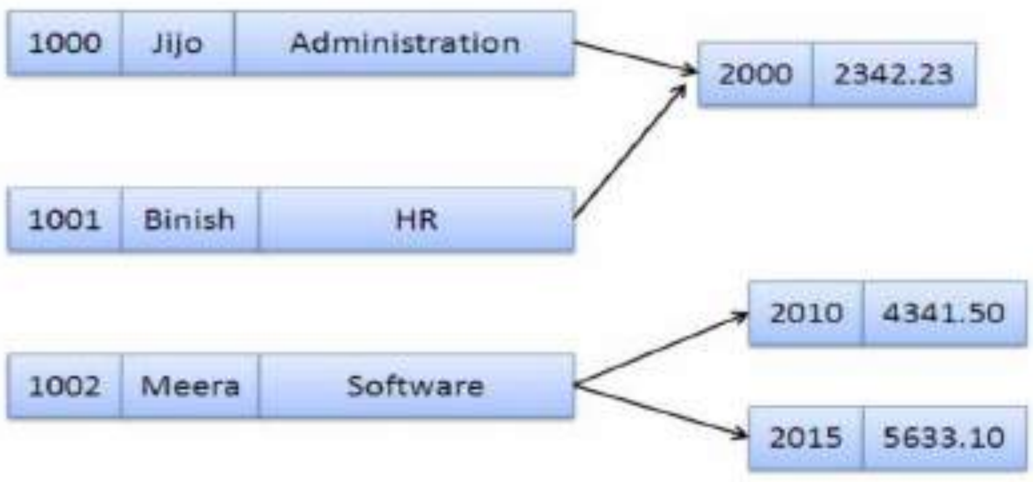
employee_loan_details table

The main highlights of the relational model are:
1. Data is stored in tables called **relations**.
2. Relations can be normalized.
3. In normalized relations, values saved are atomic values.
4. Each row in a relation contains a unique value.
5. Each column in a relation contains values from a same domain.

The Network Model:
→Data in the **network model** are represented by collection of records and relationships among data are connected by links. The links can be viewed as pointers.

The records in the database are represented in the form of graphs.

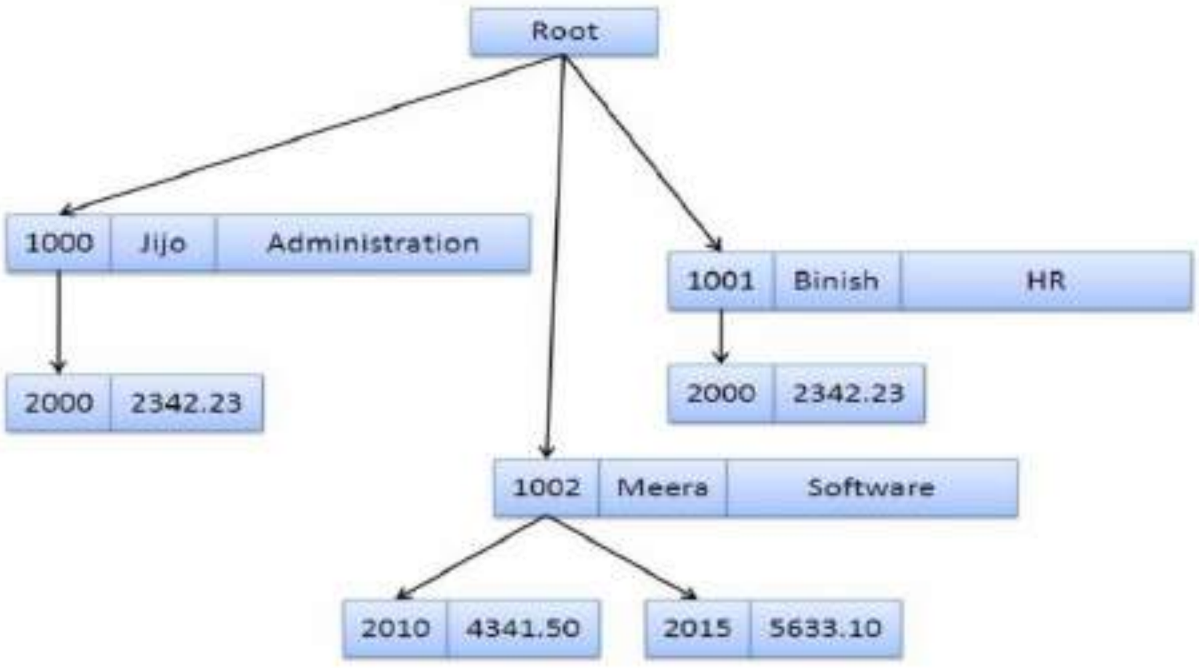
2. Network data model



Hierarchical Model:

→**Hierarchical model** is same as the network model i.e., data in the hierarchical model also represented by collection of records and relationships among data are connected by links. The links can be viewed as pointers. But, the difference from network model is that, the records in the database are represented in the form of trees.

1. Hierarchical data model



b) VIEW OF DATA:

Levels of Abstraction in a DBMS: (**********)

→**Data abstraction** is a process of representing the essential features without including implementation details.

Three levels of abstraction are:

Chapter-2
INTRODUCTION TO DATABASE SYSTEM ARCHITECTURE

1. External / View Level (or) External / View Schema.

2. Conceptual / Logical Level (or) Conceptual / Logical Schema.

3. Physical /Internal Level (or) Physical /Internal Schema.

1. External / View Level:
→This is the highest level of data abstraction. This level describes the user interaction with database system.

→**External Level** is described by a schema i.e. it consists of definition of logical records and relationship in the external view. It also contains the method of deriving the objects in the external view from the objects in the conceptual view.

2. Conceptual / Logical Level:
→This is the middle level of 3-level data abstraction architecture. It describes what data is stored in database.

→**Conceptual Level** represents the entire database. Conceptual schema describes the records and relationship included in the Conceptual view. It also contains the method of deriving the objects in the conceptual view from the objects in the internal view.

3. Physical /Internal Level:
→This is the lowest level of data abstraction. It describes how data is actually stored in database. You can get the complex data structure details at this level.

→**Internal level** indicates how the data will be stored and described the data structures and access method to be used by the database. It contains the definition of stored record and method of representing the data fields and access aid used.

Chapter-2

INTRODUCTION TO DATABASE SYSTEM ARCHITECTURE

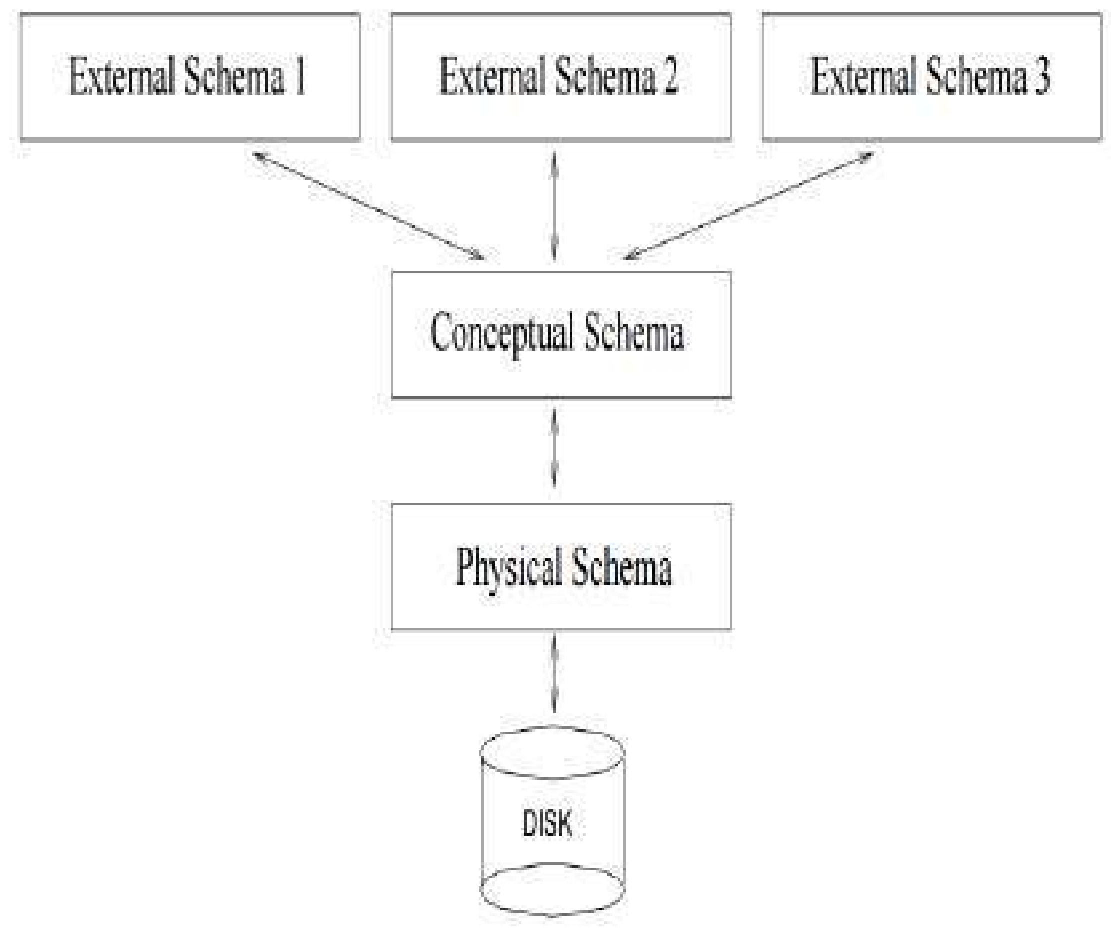


Figure 1.2 Levels of Abstraction in a DBMS

→**Example**: Let's say we are storing customer information in a customer table.
At **physical level** these records can be described as blocks of storage (bytes, gigabytes, terabytes etc.) in memory. These details are often hidden from the programmers.

At the **logical level** these records can be described as fields and attributes along with their data types, their relationship among each other can be logically implemented. The programmers generally work at this level because they are aware of such things about database systems.

At **view level**, user just interact with system with the help of GUI and enter the details at the screen, they are not aware of how the data is stored and what data is stored; such details are hidden from them.

Data Independence: (**********************)

The ability to modify a scheme definition in one level without affecting a scheme definition in a higher level is called data independence.

→Two kinds of data independence:

1. Logical Data independence
2. Physical data Independence

1. Logical Data independence:
→**Logical data independence** is the ability to modify the logical schema i.e., conceptual schema, which decides what information is to be kept in the database without affecting the next higher level schema i.e., external schema application program.

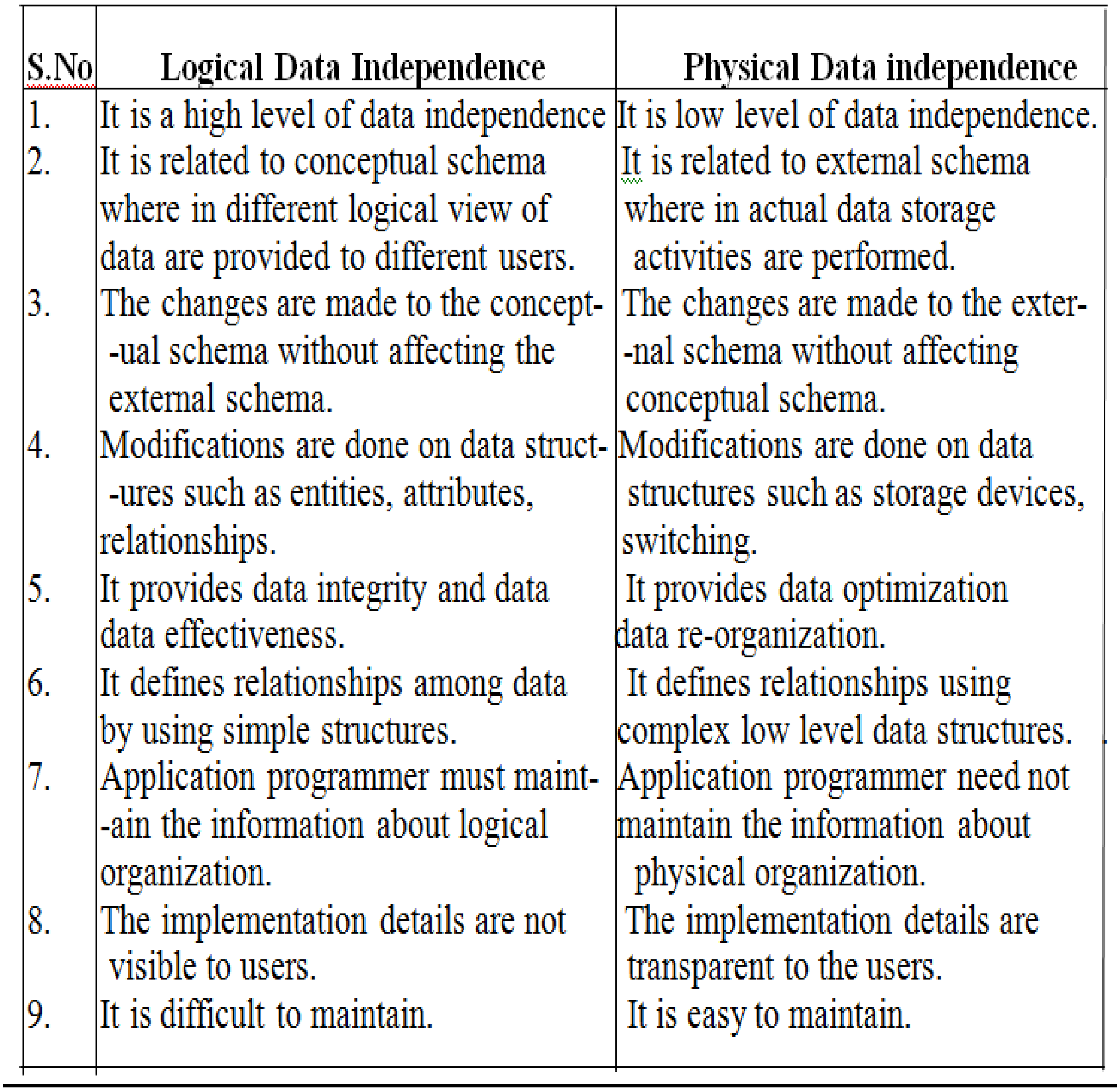
→ **Logical data independence** refers to the immunity (protection) of the external schemas to changes in the conceptual schema. Changes to the conceptual schema such as the addition or removal of new entities, attributes (or) relationships should be possible without having to change existing external schemas or having rewrite the application programs.

→**Logical data independence** is a kind of mechanism, which liberalizes itself from actual data stored on the disk. If we do some changes on table format, it should not change the data residing on the disk.

2. Physical data Independence:
→**Physical data independence** is the ability to modify the physical schema i.e., internal schema which describes the physical storage devices or structure to store the data without affecting the conceptual schema application programs.

→ Physical data independence refers to the immunity of the conceptual schema to changes in the internal schema. Changes to the internal schema such as using different file organizations or storage structures, using different storage devices, modifying indexes or hashing algorithms should be possible without having to change the conceptual or external schema.

Comparison between Logical and physical Data Independence:

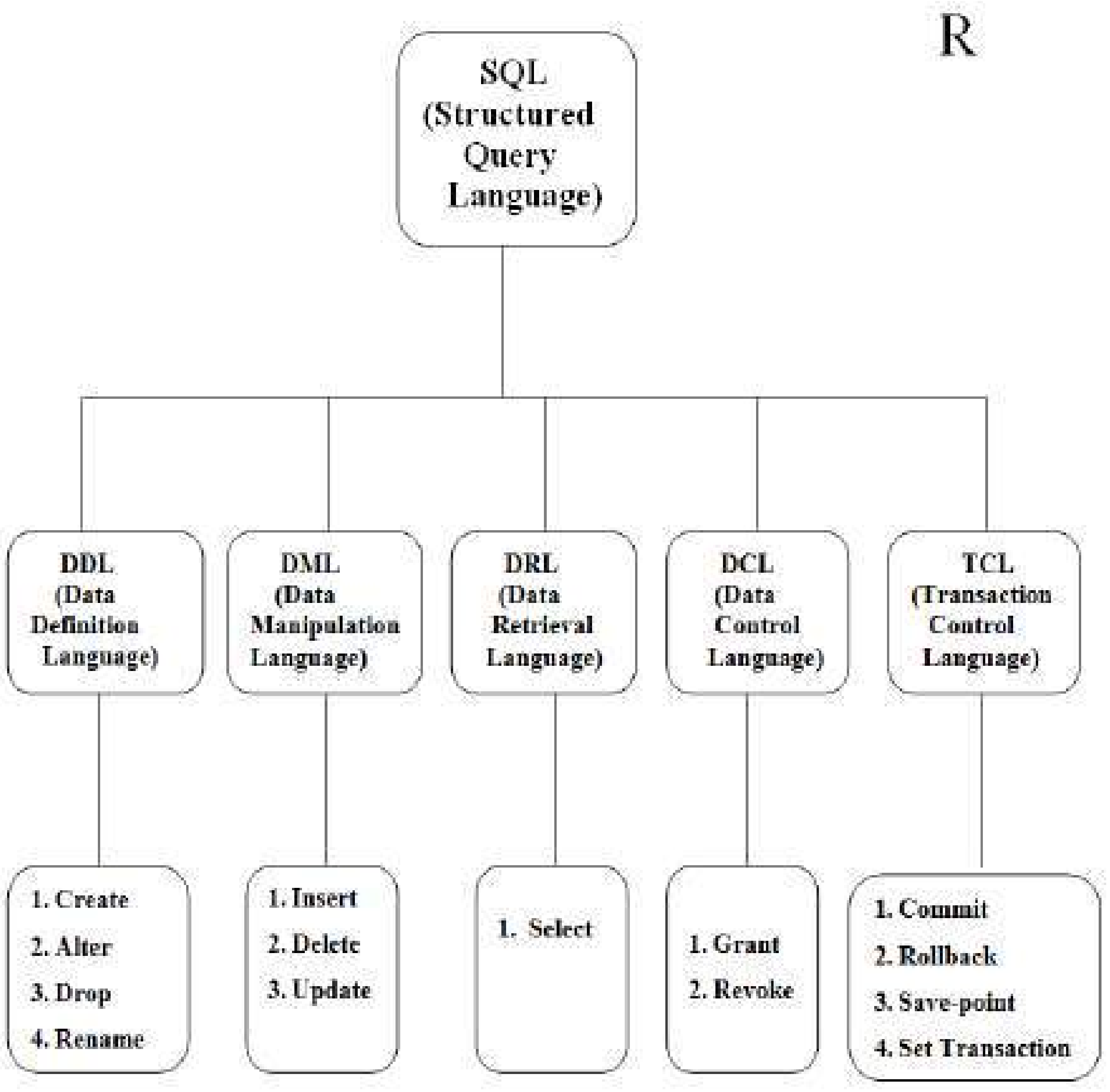


S.No	Logical Data Independence	Physical Data independence
1.	It is a high level of data independence	It is low level of data independence.
2.	It is related to conceptual schema where in different logical view of data are provided to different users.	It is related to external schema where in actual data storage activities are performed.
3.	The changes are made to the concept- -ual schema without affecting the external schema.	The changes are made to the exter- -nal schema without affecting conceptual schema.
4.	Modifications are done on data struct- -ures such as entities, attributes, relationships.	Modifications are done on data structures such as storage devices, switching.
5.	It provides data integrity and data data effectiveness.	It provides data optimization data re-organization.
6.	It defines relationships among data by using simple structures.	It defines relationships using complex low level data structures.
7.	Application programmer must maint- -ain the information about logical organization.	Application programmer need not maintain the information about physical organization.
8.	The implementation details are not visible to users.	The implementation details are transparent to the users.
9.	It is difficult to maintain.	It is easy to maintain.

2. DATABASE LANGUAGES:

Database languages are used for read, update and store data in database. There are several such **languages** that can be used for this purpose; one of them is SQL (Structured Query **Language**).

SQL (Structural query Language):


R
SQL (Structured Query Language)
DDL (Data Definition Language)
DML (Data Manipulation Language)
DRL (Data Retrieval Language)
DCL (Data Control Language)
TCL (Transaction Control Language)
1. Create
2. Alter
3. Drop
4. Rename
1. Insert
2. Delete
3. Update
1. Select
1. Grant
2. Revoke
1. Commit
2. Rollback
3. Save-point
4. Set Transaction

→**SQL** is a standard language for accessing databases.

→It is classified into five types:-

1). DDL (Data Definition Language):- Create, Alter, Drop

2). DML (Data Manipulation Language): Insert, delete, Update

3). DCL (Data Control Language):-Grant, Revoke

4). TCL (Transaction Control Language):- Commit, Rollback, Save- point, Set Transaction

5). DRL (Data Retrieval Language):- Select

Data Definition Language (DDL):

→It is a language that allows the users to define data and their relationship to other types of data. It is mainly used to create files, databases, data dictionary and tables within databases.

→It is also used to specify the structure of each table, set of associated values with each attribute, integrity constraints, security and authorization information for each table and physical storage structure of each table on disk.

→The following table gives an overview about usage of DDL statements in SQL:

S.No	Need and Usage	The SQL DDL Statement
1	Create schema objects	CREATE
2	Alter schema objects	ALTER
3	Delete schema objects	DROP
4	Rename schema objects	RENAME

Data Manipulation Language (DML):
→It is a language that provides a set of operations to support the basic data manipulation operations on the data held in the databases. It allows users to insert, update, delete and retrieve data from the database. The part of DML that involves data retrieval is called a query language.
→The following table gives an overview about the usage of DML statements in SQL:

S.No	Need and Usage	The SQL DML Statement
1	Remove rows from tables or views	DELETE
2	Add new rows of data into table or view	INSERT
3	Change column values in existing rows of a table or view	UPDATE

Data Control Language (DCL):
→DCL statements control access to data and the database using statements such as GRANT and REVOKE. A privilege can either be granted to a User with the help of GRANT statement. The privileges assigned can be SELECT, ALTER, DELETE, EXECUTE, INSERT, INDEX etc. In addition to granting of privileges, you can also revoke (taken back) it by using REVOKE command.

→The following table gives an overview about the usage of DCL statements in SQL:

S.No	Need and Usage	The SQL DCL Statement
1	Gives user access privileges to database	GRANT
2	Take back permissions from user	REVOKE

Transition Control Language (TCL):
→TCL commands are used to manage transactions in database. These are used to manage the changes made by DML statements. It also allows statements to be grouped together into logical transactions.

S.No	Need and Usage	The SQL TCL Statement
1	Save work done	COMMIT
2	Identify a point in a transaction to which you can later rollback	SAVEPOINT
3	Restore database to original since the last commit	ROLLBACK
4	Change transition options like isolation level and what rollback segment to use	SET TRANSITION

Data Retrieval Language (DRL) (or) **Data Query Language (DQL):**
→DRL will be used for the retrieval of the data from database. In order to see the present in the database, we will use DRL statement. We have only one DRL statement.
→SELECT is the only DRL statement in SQL.
→Select is DRL/DQL i.e., data retrieval language.

S.No	Need and Usage	The SQL DMCL Statement
1	Retrieve data from one or more tables	SELECT

Chapter-2
INTRODUCTION TO DATABASE SYSTEM ARCHITECTURE

Database Users and ADMINISTRATOR:

Database Users and User Interfaces: (**************)

→**Database users** are the one who really use and take the benefits of database. There will be different types of users depending on their need and way of accessing the database.

1. **Application Programmers -** They are the developers who interact with the database by means of DML queries. These DML queries are written in the application programs like C, C++, JAVA, Pascal etc. These queries are converted into object code to communicate with the database. For example, writing a C program to generate the report of employees who are working in particular department will involve a query to fetch the data from database. It will include an embedded SQL query in the C Program.
2. **Sophisticated Users -** They are database developers, who write SQL queries to select/insert/delete/update data. They do not use any application or programs to request the database. They directly interact with the database by means of query language like SQL. These users will be scientists, engineers, analysts who thoroughly study SQL and DBMS to apply the concepts in their requirement. In short, we can say this category includes designers and developers of DBMS and SQL.
3. **Specialized Users -** These are also sophisticated users, but they write special database application programs. They are the developers who develop the complex programs to the requirement.
4. **Stand-alone Users -** These users will have stand –alone database for their personal use. These kinds of database will have readymade database packages which will have menus and graphical interfaces.
5. **Native Users -** these are the users who use the existing application to interact with the database. For example, online library system, ticket booking systems, ATMs etc which has existing application and users use them to interact with the database to fulfill their requests.

Database Administrator (DBA): (***************)

→**DBA is a person having control over data and program. DBA is a person having all the authorities and responsibilities of database.**

→A **Database administrator (DBA)** manages a DBMS for an enterprise. The DBA designs schemas, provide security, restores the system after a failure, and periodically tunes the database to meet changing user needs. Application programmers develop applications that use DBMS functionality to access and manipulate data, and end users invoke these applications.

→**The DBA is responsible for many critical tasks**:

1. Installing and upgrading the DBMS Servers: - DBA is responsible for installing a new DBMS server for the new projects. He is also responsible for upgrading these servers as there are new versions comes in the market or requirement. If there is any failure in up gradation of the existing servers, he should be able revert the new changes back to the older version, thus maintaining the DBMS working. He is also responsible for updating the service packs/ fixes/ patches to the DBMS servers.

2. Design and implementation: - Designing the database and implementing is also DBA's responsibility. He should be able to decide proper memory management, file organizations, error handling, log maintenance etc for the database.

3. Performance tuning: - Since database is huge and it will have lots of tables, data, constraints and indices, there will be variations in the performance from time to time. Also, because of some designing issues or data growth, the database will not work as expected. It is

responsibility of the DBA to tune the database performance. He is responsible to make sure all the queries and programs works in fraction of seconds.
4. Migrate database servers: - Sometimes, users using oracle would like to shift to SQL server or Netezza. It is the responsibility of DBA to make sure that migration happens without any failure, and there is no data loss.
5. Backup and Recovery: - Proper backup and recovery programs needs to be developed by DBA and has to be maintained him. This is one of the main responsibilities of DBA. Data/objects should be backed up regularly so that if there is any crash, it should be recovered without much effort and data loss.
6. Security: - DBA is responsible for creating various database users and roles, and giving them different levels of access rights.
7. Documentation: - DBA should be properly documenting all his activities so that if he quits or any new DBA comes in, he should be able to understand the database without any effort. He should basically maintain all his installation, backup, recovery, security methods. He should keep various reports about database performance.

3. TRANSACTION MANAGEMENT:

→A **transaction** is a collection of operations that performs a single logical function in a database application

→A **transaction** is any one execution of a user program in a DBMS. (Executing the same program several times will generate several transactions.) This is the basic unit of change as seen by the DBMS: Partial transactions are not allowed, and the effect of a group of transactions is equivalent to some serial execution of all transactions.
Transactions should posses the following (ACID) properties:
Transactions should possess several properties. These are often called the **ACID properties,** and they should be enforced by the concurrency control and recovery methods of the DBMS. The following are the ACID properties:

1. **Atomicity:** A transaction is an atomic unit of processing; it is either performed in its entirety or not performed at all

2. **Consistency preservation:** A transaction is consistency preserving if its complete execution take(s) the database from one consistent state to another.

3. **Isolation:** A transaction should appear as though it is being executed in isolation from other transactions. That is, the execution of a transaction should not be interfered with by any other transactions executing concurrently.
4. **Durability or permanency:** The changes applied to the database by a committed transaction must persist in the database. These changes must not be lost because of any failure.

4. DBMS STRUCTURE(************************)

Chapter-2
INTRODUCTION TO DATABASE SYSTEM ARCHITECTURE

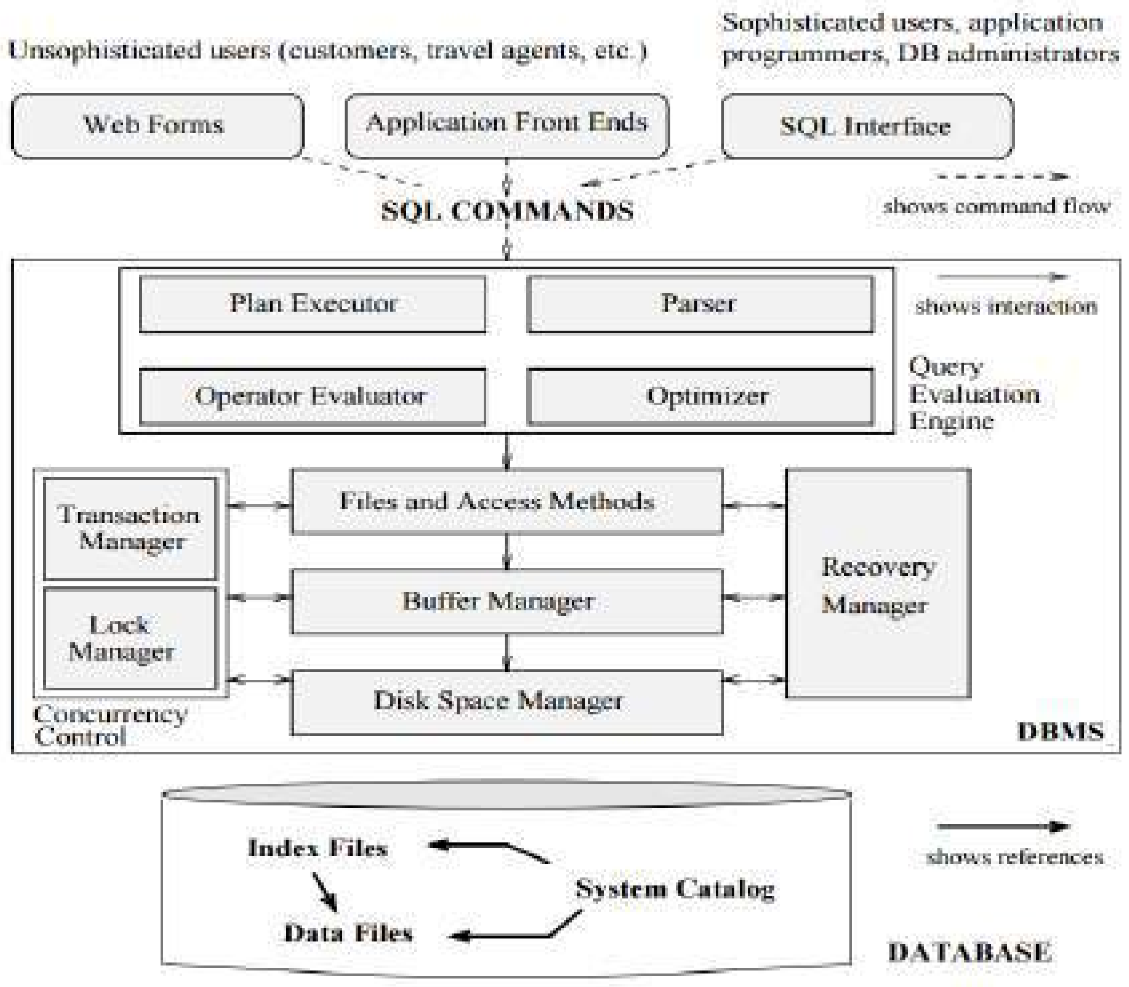


Figure 1.3 Architecture of a DBMS

→**Figure 1.3** shows the structure (with some simplification) of a typical DBMS based on the relational data model.

→The DBMS accepts **SQL commands** generated from a variety of user interfaces, produces query evaluation plans, executes these plans against the database, and returns the answers. (This is a simplification: SQL commands can be embedded in host language application programs, e.g., Java or COBOL programs. We ignore these issues to concentrate on the core DBMS functionality.)

→When a user issues a query, the parsed query is presented to a **query optimizer**, which uses information about how the data is stored to produce an efficient execution plan for evaluating the query. An **execution plan** is a blueprint for evaluating a query, and is usually represented as a tree of relational operators.

→The code that implements relational operators sits on top of the **file and access methods layer**. This layer includes a variety of software for supporting the concept of a file, which, in a DBMS, is a collection of pages or a collection of records. This layer typically supports a heap file, or file of unordered pages, as well as indexes.

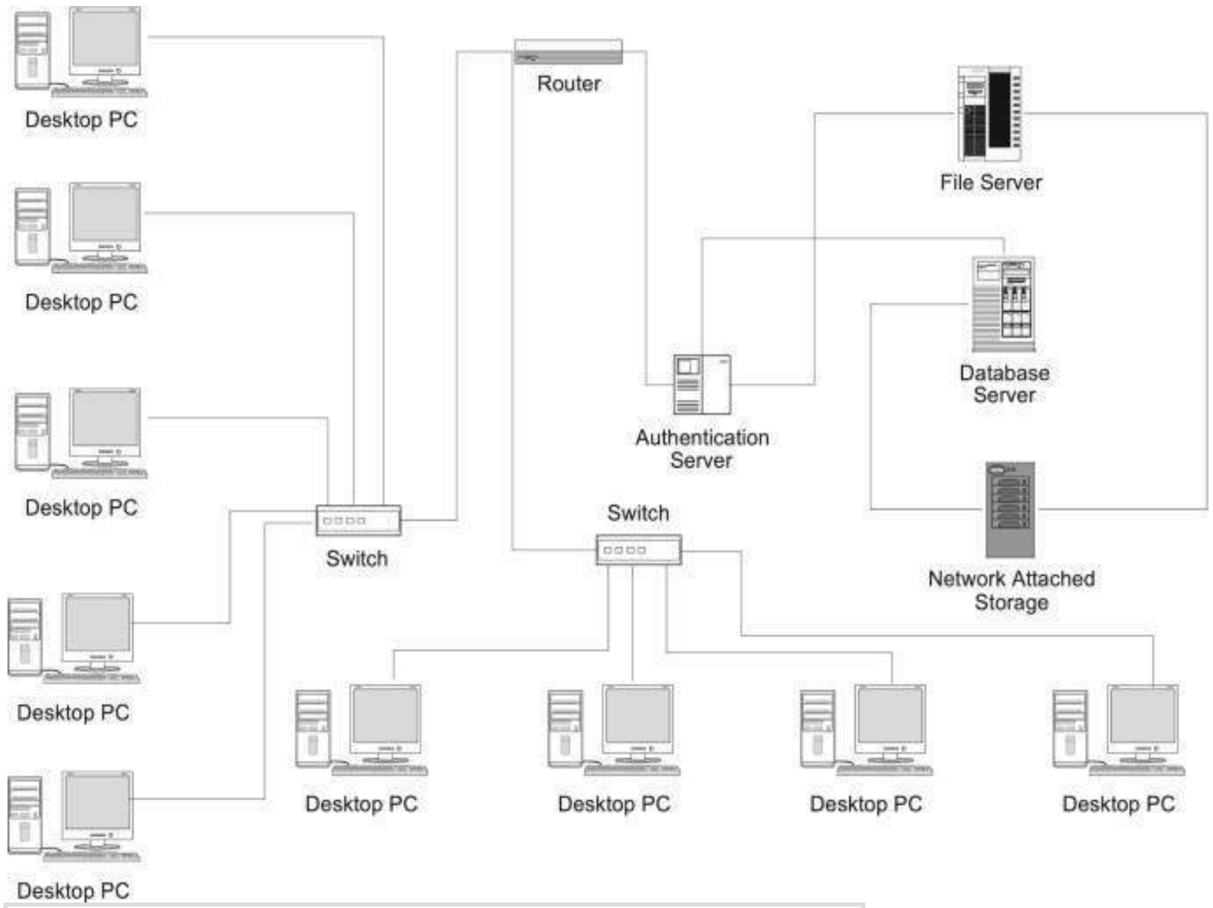
→The files and access methods layer code sits on top of the **buffer manager**, which brings pages in from disk to main memory as needed in response to read requests.

→The lowest layer of the DBMS software deals with management of space on disk, where the data is stored. Higher layers allocate, de-allocate, read, and write pages through (routines provided by) this layer, called the **disk space manager**.

→The DBMS supports concurrency and crash recovery by carefully scheduling user requests and maintaining a log of all changes to the database. DBMS components associated with concurrency control and recovery include the **transaction manager**, which ensures that transactions request and release locks according to a suitable locking protocol and schedules the execution transactions; the **lock manager**, which keeps track of requests for locks and grants locks on database objects when they become available; and the recovery manager, which is responsible for maintaining a log, and restoring the system to a consistent state after a crash. **The disk space manager, buffer manager, and file and access method layers** must interact with these components.

Client/Server Architecture:

Client/server architecture shares the data processing chores between a server—typically a high-end workstation—and clients, which are usually PCs. PCs have significant processing power and are therefore capable of taking raw data returned by the server and formatting it for output. Application programs are stored and executed on the PCs. Network traffic is reduced to data manipulation requests sent from the PC to the database server and raw data returned as a result of that request. The result is significantly less network traffic and theoretically better performance.
Today's client/server architectures exchange messages over local area networks (LANs). Although a few older Token Ring LANs are still in use, most of today's LANs are based on Ethernet standards. As an example, take a look at the small network in Figure 1-3. The database runs on its on server (the database server), using additional disk space on the network attached storage device. Access to the database is controlled not only by the DBMS itself but by the authentication server.



- Figure Small LAN with network-accessible database server.

Client/server architecture is similar to the traditional centralized architecture in that the DBMS resides on a single computer. In fact, many of today's mainframes actually function as large, fast servers. The need to handle large data sets still exists, although the location of some of the processing has changed.
Because client/server architecture uses a centralized database server, it suffers from the same reliability problems as the traditional centralized architecture: If the server goes down, data access is cut off. However, because the “terminals” are PCs, any data downloaded to a PC can be processed without access to the server.

Chapter-3
ENTITY RELATIONSHIP MODEL

UNIT-II:

The E/R Models, The Relational Model, Relational Calculus, Introduction to Database Design, Database Design and Er Diagrams-Entities Attributes, and Entity Sets-Relationship and Relationship Sets-Conceptual Design With the Er Models, The Relational Model Integrity Constraints Over Relations- Key Constraints –Foreign Key Constraints-General Constraints, Relational Algebra and Calculus, Relational Algebra- Selection and Projection- Set Operation, Renaming – Joins- Division- More Examples of Queries, Relational Calculus, Tuple Relational Calculus- Domain Relational Calculus.

1. OVERVIEW OF DATABASE DESIGN: (**************)

→The **database design process** can be divided into six steps. The **ER model** is most relevant to the first three steps:

(1) **Requirements Analysis:** The very first step in designing a database application is to understand what data is to be stored in the database, what applications must be built on the database and what operations must be performed on the database. In other words, we must find out what the users want from the database. This process involves discussions with user groups, a study of the current operating environment, how it is expected to change an analysis of any available documentation on existing applications and so on.

(2) **Conceptual Database Design:** The information gathered in the requirements analysis step is used to develop a high-level description of the data to be stored in the database, along with the constraints that are known to hold over this data. The goal is to create a description of the data that matches to how both users and developers think of the data. This facilities discussion among all the people involved in the design process i.e., developers and as well as users who have no technical background. In simple words, the conceptual database design phase is used in drawing ER model.

(3) **Logical Database Design:** We must implement our database design and convert the conceptual database design into a database schema (a description of data) in the data model (a collection of high level data description constructs that hide many low level storage details) of the DBMS. We will consider only consider relational DBMSs, and therefore, the task in the logical design step is to convert the conceptual database design in the form of an ER schema (Entity-Relationship schema) into a relational database schema.

(4) **Schema Refinement:** The fourth step in database design is to analyze the collection of relations in our relational database schema to identify future problems, and to refine (clear) it.

(5) **Physical Database Design:**. This step may simply involve building indexes on some tables and clustering some tables, or it may involve redesign of parts of the database schema obtained from the earlier design steps.

(6) **Application and Security Design:** Any software project that involves a DBMS must consider applications that involve processes and identify the entities.

Example: Users, user groups, departments, etc.

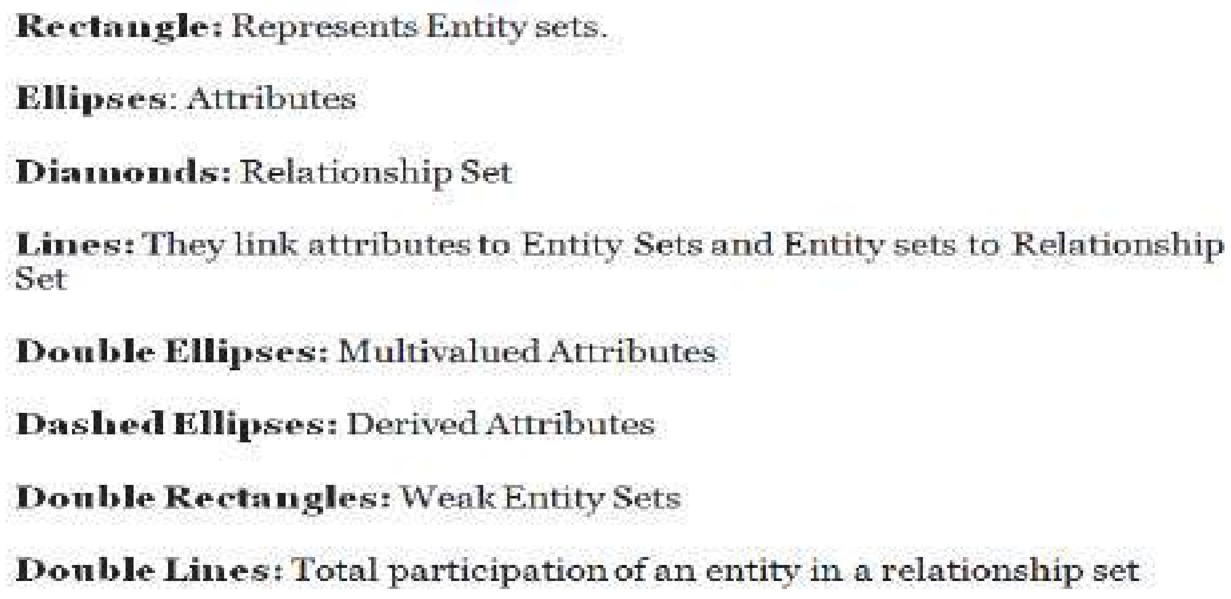
→We must describe the role of each entity in every process. As a security design, for each role, we must identify the parts of the database that just not is accessible and we must take steps to ensure that these access rules are enforced.

<u>**Conceptual design:**</u>

- **What are the entities and relationships in the enterprise?**
- **What information about these entities and relationships should be stored in the database?**
- **What are the integrity constraints or business rules that hold?**
- **A database schema in the ER Model can be represented pictorially (ER diagrams)**
- **An ER diagram can be mapped into a relational schema**

<u>E-R MODEL:</u>
→An **entity–relationship model (ER model)** is a systematic way of describing and defining a business process. An ER model is typically implemented as a database.
→The main components of E-R model are: entity set and relationship set.

→Here are the geometric shapes and their meaning in an E-R Diagram –



Rectangle: Represents Entity sets.

Ellipses: Attributes

Diamonds: Relationship Set

Lines: They link attributes to Entity Sets and Entity sets to Relationship Set

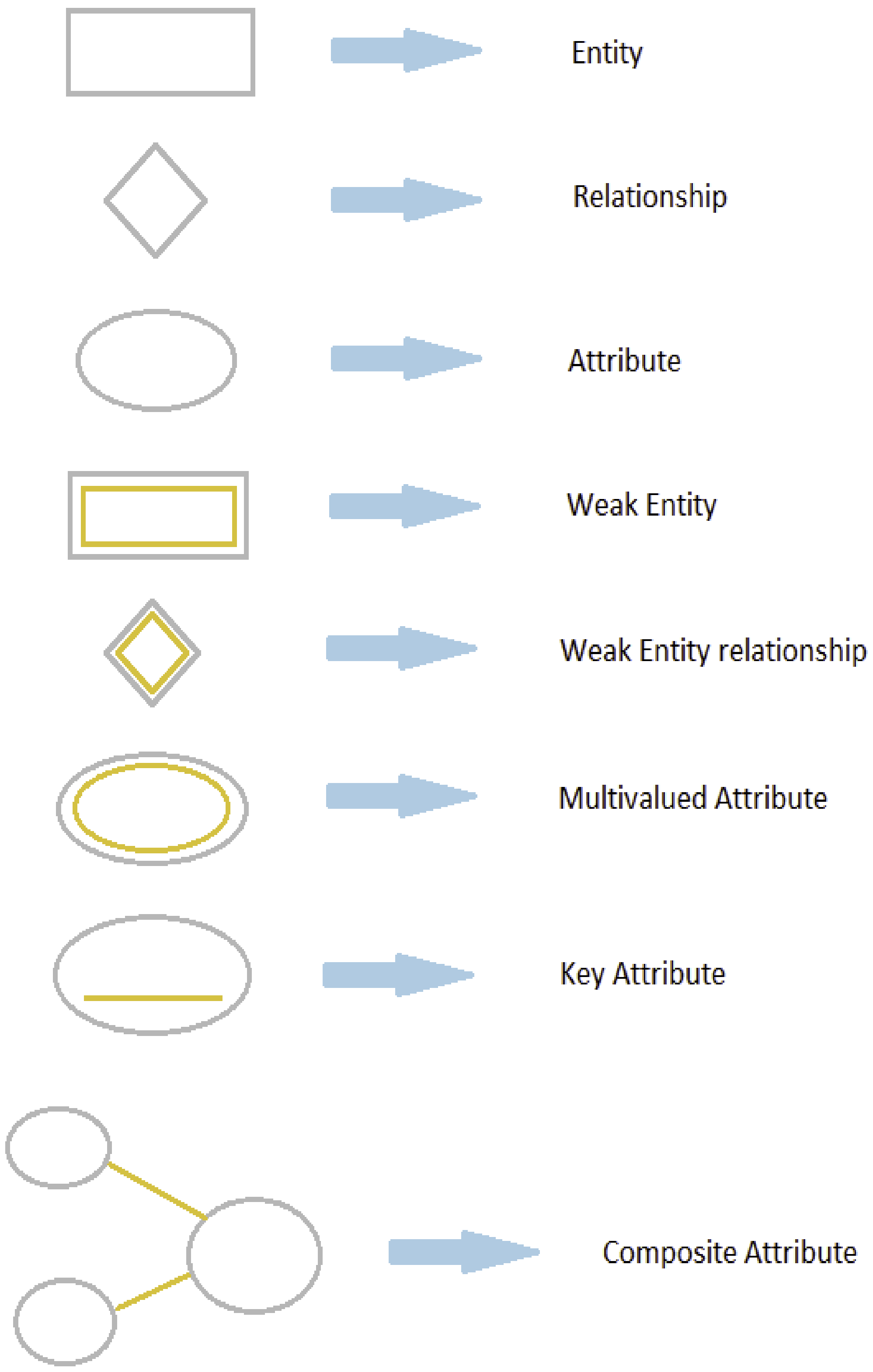
Double Ellipses: Multivalued Attributes

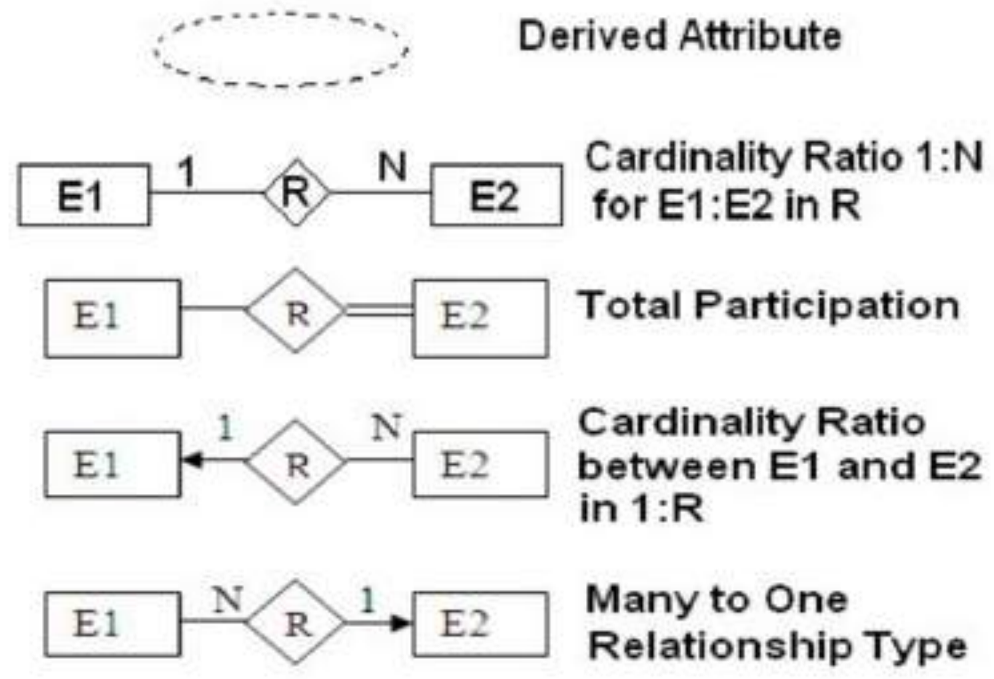
Dashed Ellipses: Derived Attributes

Double Rectangles: Weak Entity Sets

Double Lines: Total participation of an entity in a relationship set

Symbols and Notations




Derived Attribute
E1
1
R
N
E2
Cardinality Ratio 1:N for E1:E2 in R
E1
R
E2
Total Participation
E1
1
R
N
E2
Cardinality Ratio between E1 and E2 in 1:R
E1
N
R
1
E2
Many to One Relationship Type

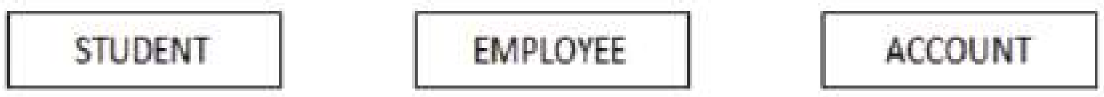
2. ENTITIES, ATTRIBUTES, AND ENTITY SETS:

ENTITIES:

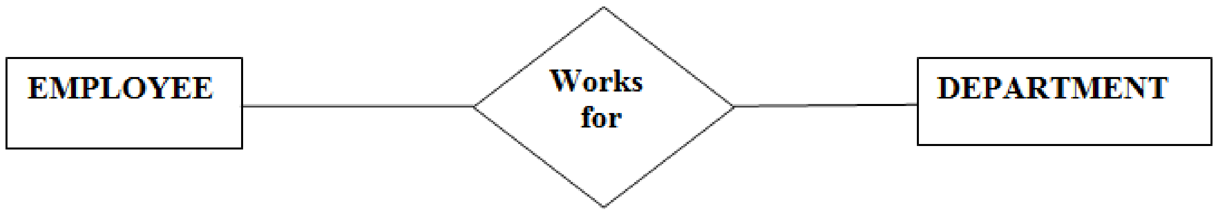
An **Entity** is an object that exists and is distinguishable from other objects.

Example: Specific person, Company, Event, Plant, Building, Room, Chair, Course, Employee etc.

In E-R Diagram, an **entity** is represented using rectangles. Name of the Entity is written inside the rectangle.
→**Examples**: STUDENT, EMPLOYEE, ACCOUNT etc.



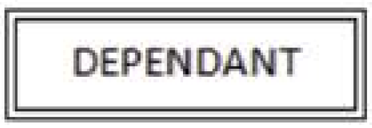
A **Strong entity** is represented by simple rectangle as shown above.
→Consider an **example** of an Organization. Employee, Manager, Department, Product and many more can be taken as entities from an Organization.



A **Weak entity** is an entity that depends on another entity. Weak entity doesn't have key attribute of their own. Double rectangle represents weak entity.

Examples: CLASS_SECTION, DEPENDANT etc.

CLASS_SECTION



An **<u>Entity set</u>** is a set of entities of the same type that share the same properties.
→**An Entity set** is a collection of similar entities.
Examples: set of all persons, companies, Job positions, Courses, Academic staff, Managers, Employees etc.
–All entities in an entity set have the same set of attributes. (Until we consider ISA hierarchies, anyway!)
–Each entity set has a *key*.
–Each attribute has a *domain*.

→The **Employees entity set** with attributes ssn, name, and lot is shown in Figure 2.1. An entity set is represented by a rectangle, and an attribute is represented by an oval. Each attribute in the **primary key** is underlined.

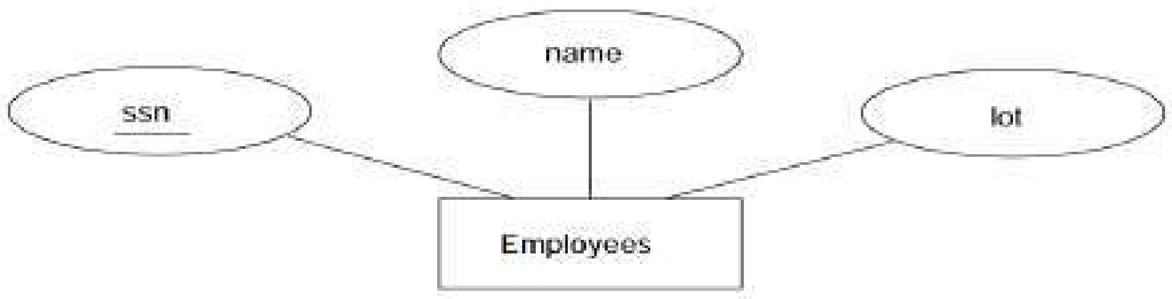
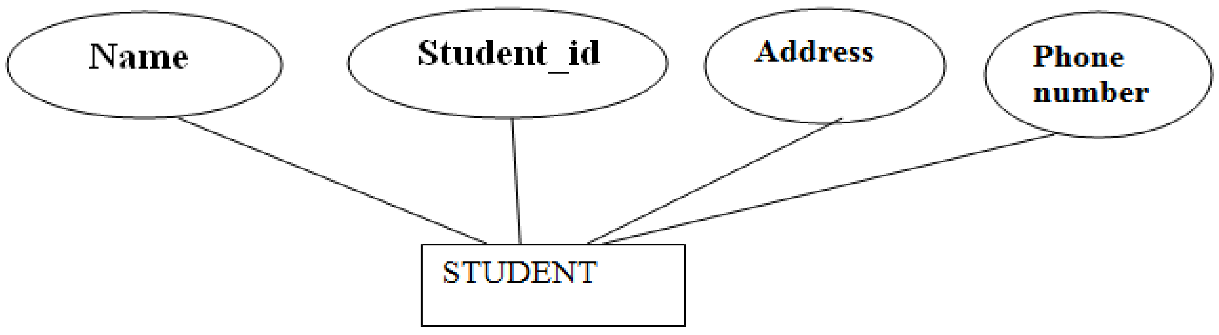


Figure 2.1 The Employees Entity Set

ATTRIBUTES:

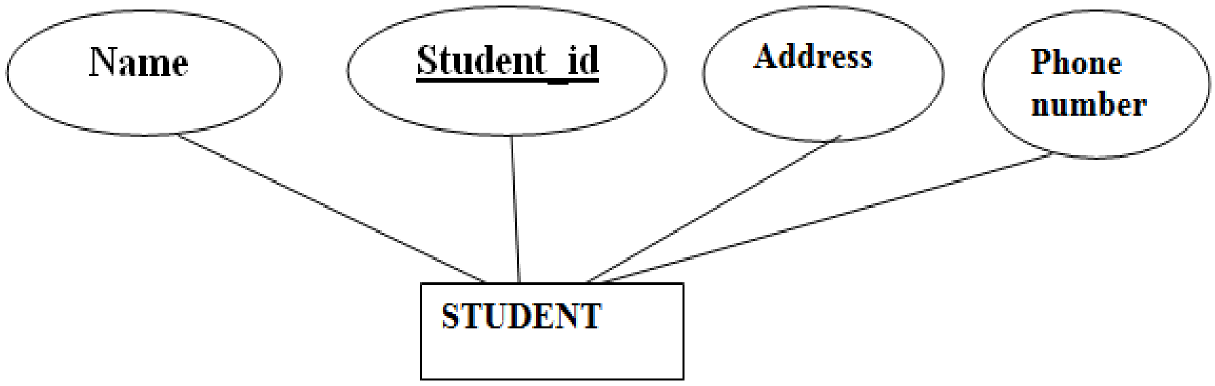
An entity is represented by a set of **attributes. Attributes are** descriptive properties possessed by each member of an entity set.

An **Attribute** describes a property or characteristics of an entity. For example, Name, Age, Address etc can be attributes of a Student. An attribute is represented using eclipse.

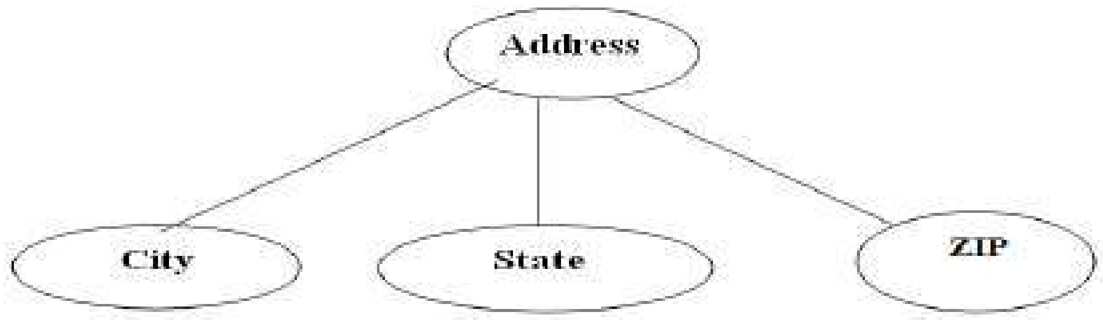


Example:
Customer=(customer_id, customer_name, customer_street, customer_city)
loan = (loan_number, amount)

Key Attribute:
Key attribute represents the main characteristics of an Entity. It is used to represent Primary key. Ellipse with underlying lines represents Key Attribute.



Composite Attribute:
An attribute can also have their own attributes. These attributes are known as **Composite** attribute.
→ Composite attributes can be divided into subparts. For example, an attribute name could be structured as a composite attribute consisting of first-name, middle-initial, and last-name.
Attribute Divided into sub parts. Eg. Name (First name, Middle Name, last name)

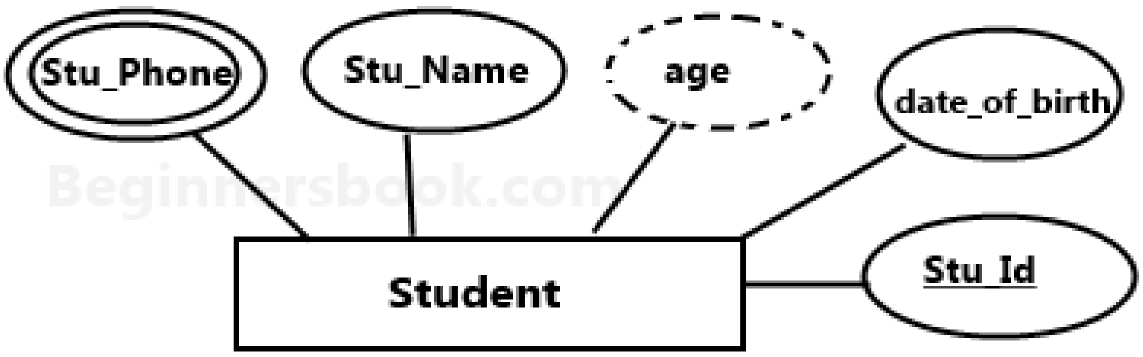


Multivalued Attributes: (********************)
An attribute that can hold multiple values is known as multivalued attribute. We represent it with double ellipses in an E-R Diagram. E.g. A person can have more than one phone numbers so the phone number attribute is multivalued.

There may be instances where an attribute has a set of values for a specific entity. Consider an employee entity set with the attribute phone-number. An employee may have zero, one, or several phone numbers, and different employees may have different numbers of phones. This type of attribute is said to be attribute having more than one values. Eg. Phone Number.

Derived Attribute: A derived attribute is one whose value is dynamic and derived from another attribute. It is represented by dashed ellipses in an E-R Diagram. E.g. Person age is a derived attribute as it changes over time and can be derived from another attribute (Date of birth).

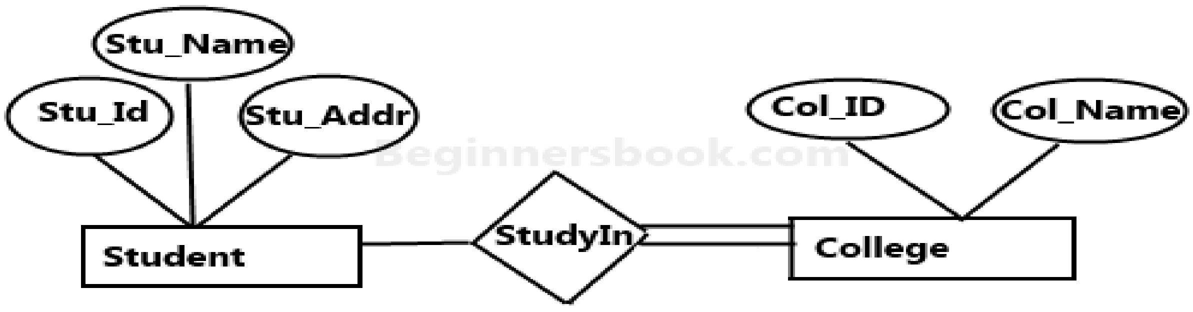
E-R diagram with multivalued and derived attributes:



Total Participation of an Entity set:

A total participation of an entity set represents that each entity in entity set must have at least one relationship in a relationship set. For example: In the below diagram each college must have at least one associated student.

.



E-R Digram with total participation of College entity set in StudyIn relationship Set - This indicates that each college must have atleast one associated Student.

3. RELATIONSHIPS AND RELATIONSHIP SETS: (****)

→A **relationship** is an association (connection) among (between) two or more entities.

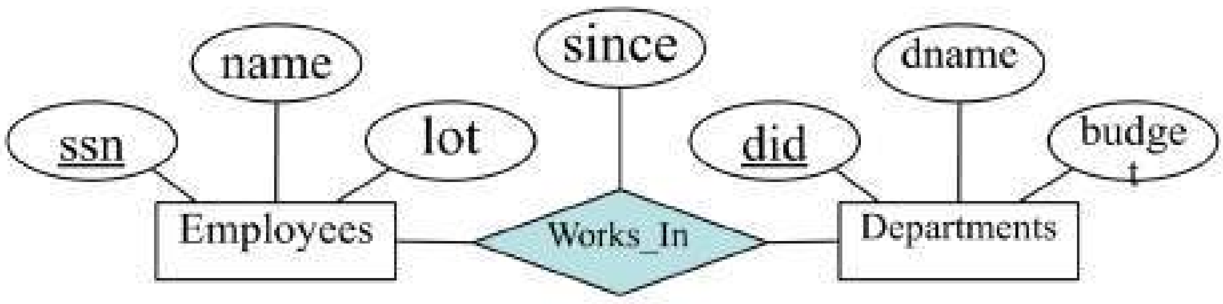


Figure 2.2 The Works_In Relationship Set

Example: We may have the relation Works_In among entities, Employees and Departments i.e., an Employee Works_In a Department.

→A **relationship set** is a collection of similar relationships or we collect a set of similar relationships into a relationship set.
A relationship set can be thought of as a set of n-tuples:
{(e1, ..., en) | e1 ∈ E1, ..., en ∈ En}

Each n-tuple denotes a relationship involving n entities e1 through en, where entity ei is in entity set Ei. Note that several relationship sets might involve the same entity sets. For example, we can also have a Manages relationship set involving Employees and Departments.

A **relationship** can also have **descriptive attributes**. **Descriptive attributes** are used to record information about the relationship, rather than about any one of the participating entities.

Example: In Works_In relationship 'since' attribute captures information about participating entities Employees and Departments.
But, for a given employee-department pair, we cannot have more than one associated 'since' attribute value.

→An instance of a relationship (or) relationship instance set is a set of relationships. An instance can be thought of as a 'snapshot' (a short description) of the relationship set at some instant in time.
An instance of the Works In relationship set is shown in **Figure 2.3.** Each Employees entity is denoted by its **ssn**, and each Departments entity is denoted by its **did**, for simplicity.

The 'since' value is shown beside each relationship as 'many-to-many' relationships and total participation i.e., the employee with ssn (123-22-3666) Works_In did (51) since 1/1/91, similarly the employee with ssn (231-31-5368) Works_In did (51) since 3/3/93 and so on.

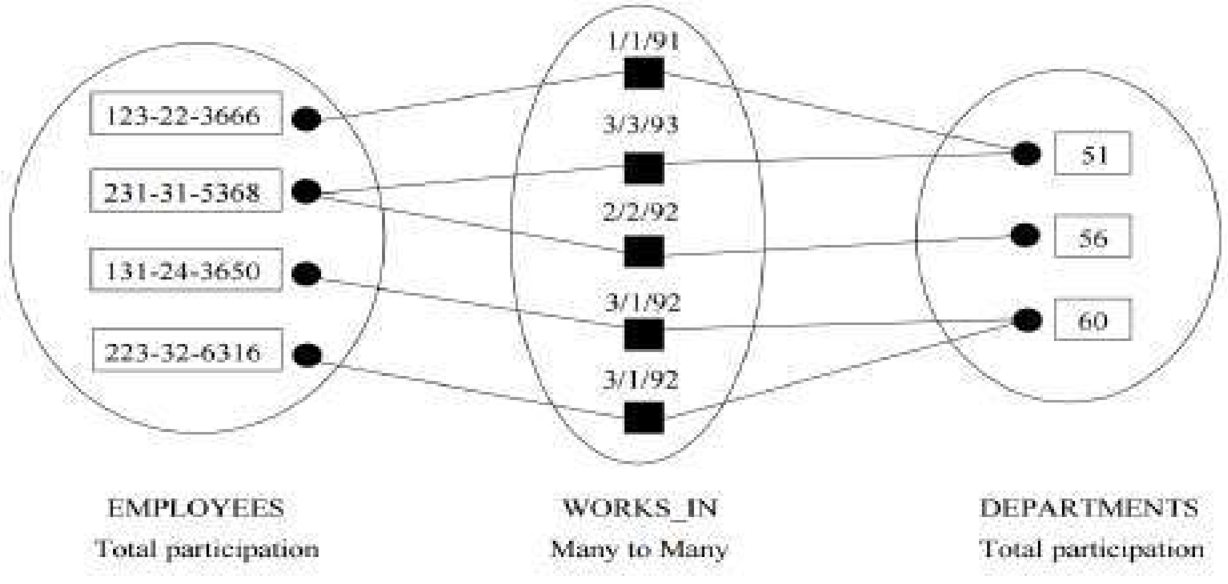


Figure 2.3 An Instance of the Works_In Relationship Set

→**Ternary relationship** is an association (connection) between three entities an employee, a department, and a location.
Example: Each department has offices in several location and we want to record the location at which each employee works.

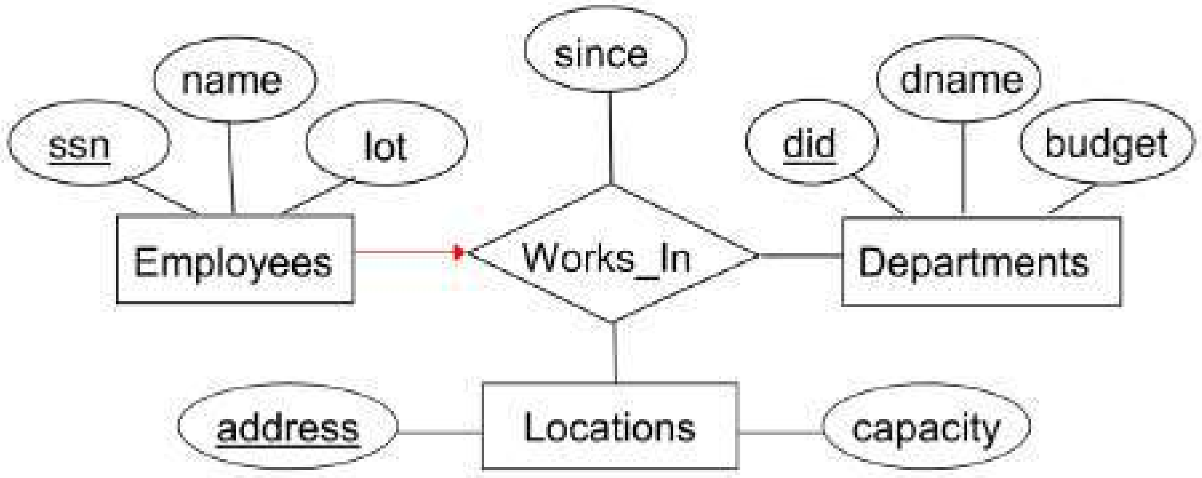


Figure 2.4 A Ternary Relationship set

→The entity sets that participate in a relationship set need not be only one. Sometimes a relationship might involve two entities in the same entity set.

For example, consider the Reports_To relationship set that is shown in **Figure 2.5**. Since employees report to other employees, every relationship in Reports_To is of the form (emp1, emp2), where both emp1 and emp2 are entities in Employees.

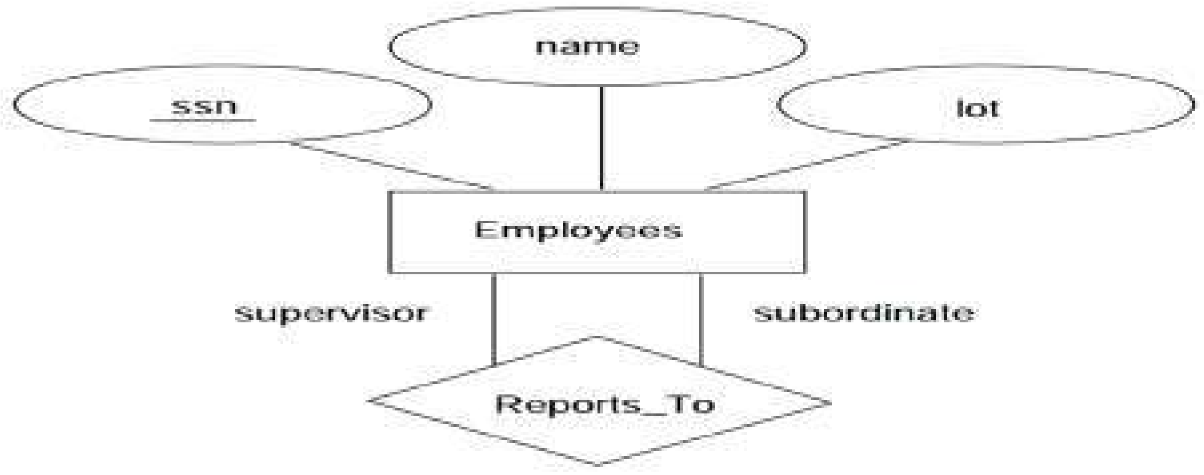


Figure 2.5 The Reports_To Relationship set

However, they play different roles: emp1 reports to the managing employee emp2, which is reflected in the role indicators supervisor and subordinate in **Figure 2.5.**

If an entity set plays more than one role, the role indicator concatenated with an attribute name from the entity set gives us a unique name for each attribute in the relationship set. For example, the Reports To relationship set has attributes corresponding to the ssn of the supervisor and the ssn of the subordinate, and the names of these attributes are supervisor ssn and subordinate ssn.

4. ADDITIONAL FEATURES OF ER MODEL:

→Following constructs are the features in the ER Model that allows us to describe some common properties of the data in expressing ER Model.

Key Constraints: (*********)

Consider the Works_In relationship shown in **Figure 2.2.**

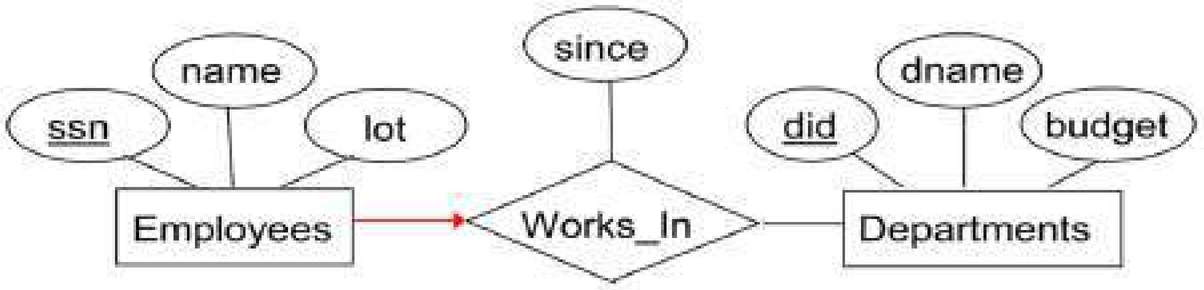


Figure 2.2 The Works_In Relationship Set

An employee can work in several departments, and a department can have several employees, as illustrated in the Works_In instance shown in **Figure 2.3.**

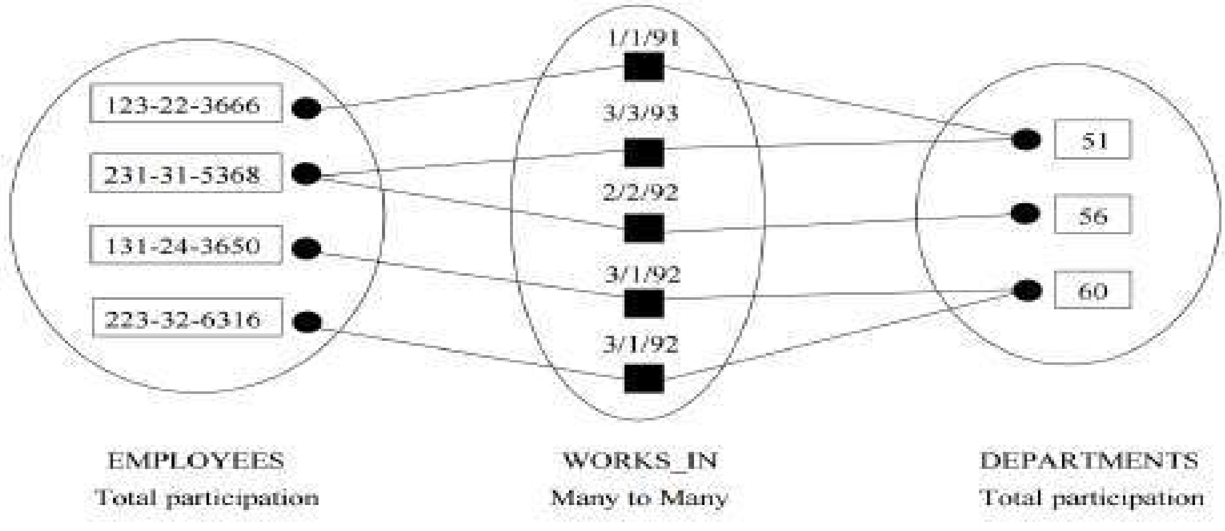


Figure 2.3 An Instance of the Works_In Relationship Set

Here, Employee 231-31-5368 has worked in Department 51 since 3/3/93 and in Department 56 since 2/2/92. Department 51 has two employees. Thus one department can have many employees.

But, if we want to have only one employee in department, then it is an example of **Kay constraint.**

Example: Consider another relationship set called Manages between the Employees and Departments entity sets as in the **Figure 2.6**.

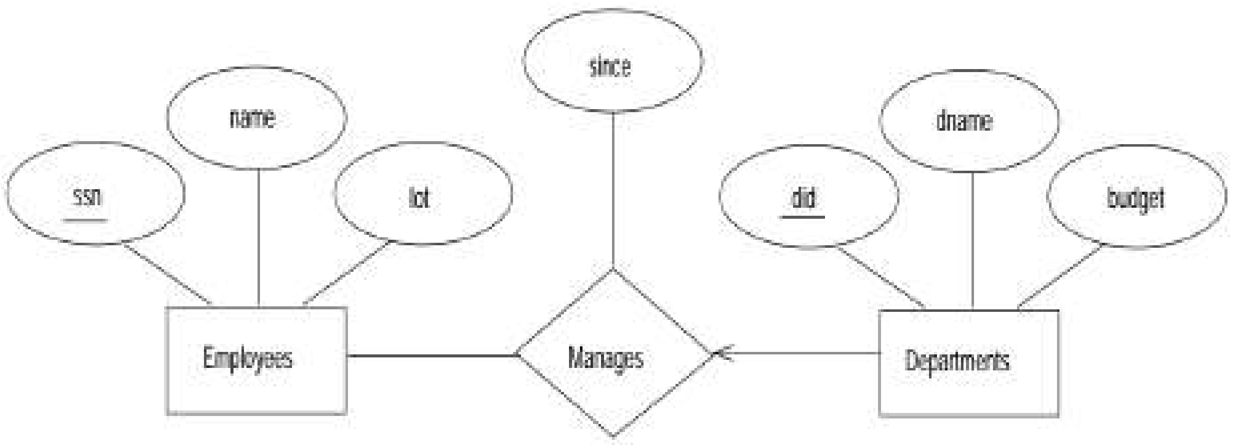


Figure 2.6 Key Constraint on Manages

Here, each department can have only one manager. The restriction that each department can have only one manager is an example of key constraints. This restriction is indicated in the above ER diagram by using an arrow from departments to manages, such that a department can have only one manager.

An instance of the Manages relationship set is shown in **Figure 2.7.** While this is also a potential instance for the Works In relationship set, the instance of Works In shown in Figure 2.3 violates the key constraint on Manages.

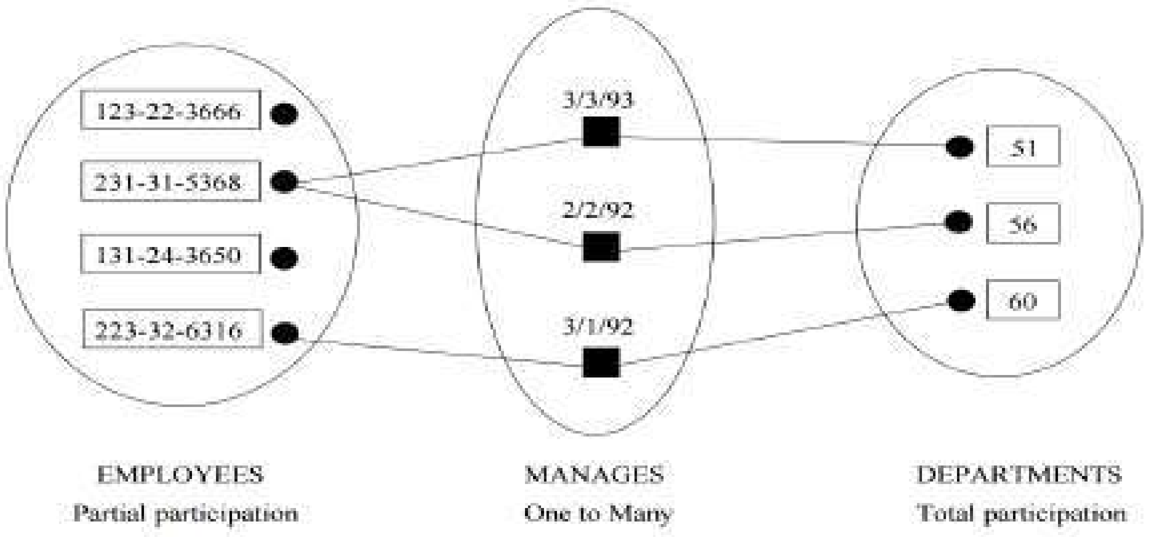


Figure 2.7 An Instance of the Manages Relationship Set

Key Constraints for Ternary Relationships:

In **Figure 2.8**, we show a ternary relationship with key constraints. Each employee works in at most one department, and at a single location.

An instance of the Works_In3 relationship set is shown in **Figure 2.9**. Notice that each department can be associated with several employees and locations, and each location can be associated with several departments and employees; however, each employee is associated with a single department and location.

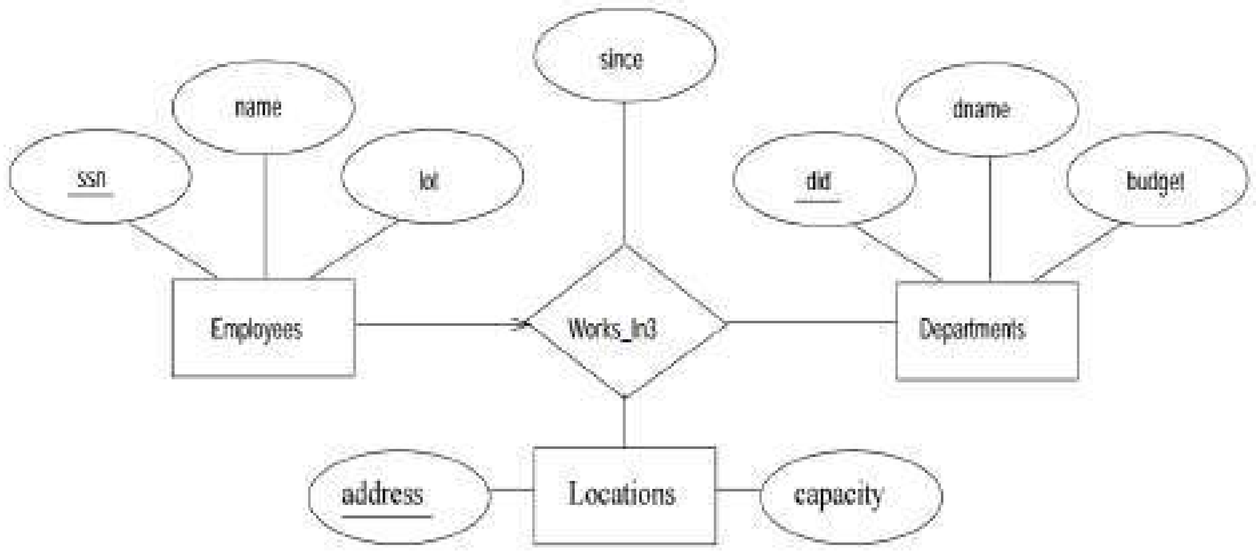


Figure 2.8 A Ternary Relationship Set with Key Constraints

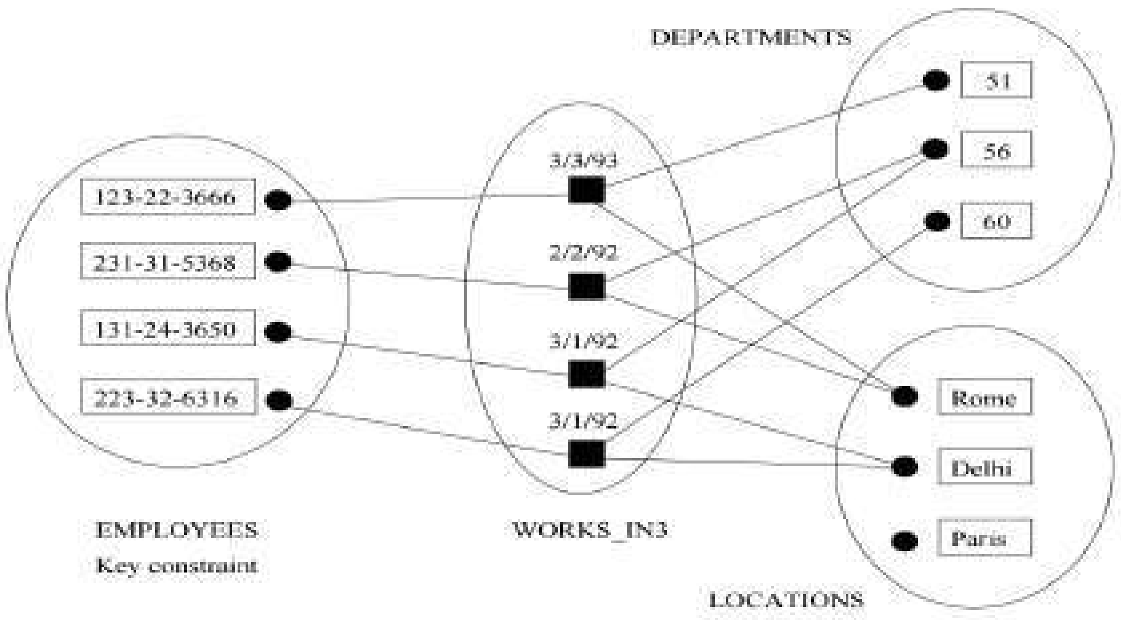


Figure 2.9 An Instance of Works_In3

<u>**Participation Constraints: (***********)**</u>

The ER diagram in **Figure 2.10** shows both the Manages and Works_In relationship sets and all the given constraints. If the participation of an entity set in a relationship set is total, the two are connected by a thick line; independently, the presence of an arrow indicates a key constraint. The instances of Works_In and Manages shown in Figures 2.3 and 2.7 satisfy all the constraints in **Figure 2.10.**

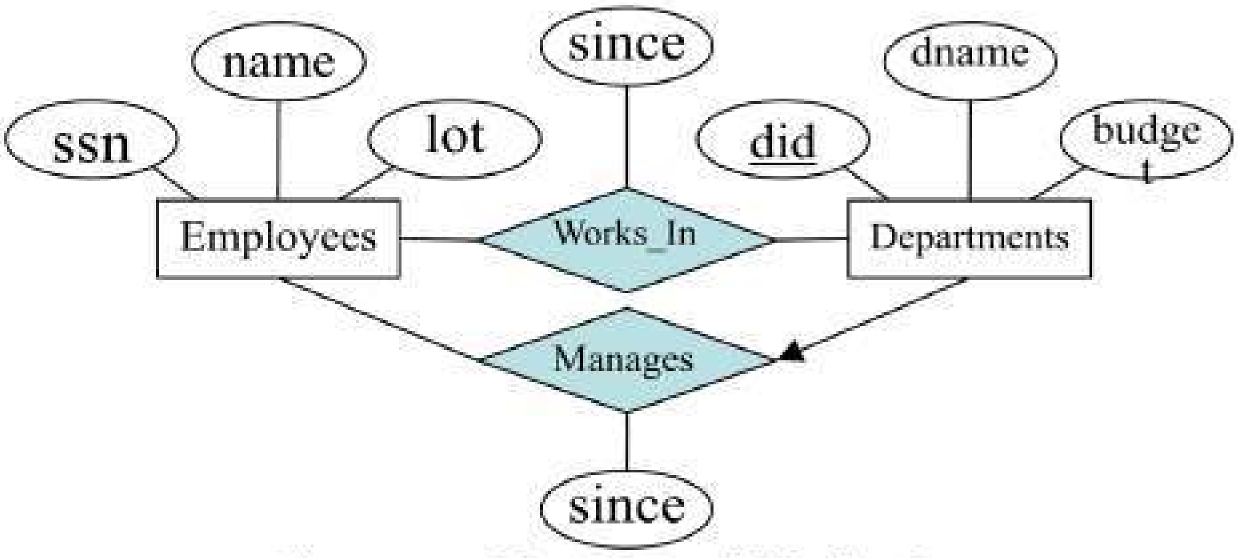


Figure 2.10 Manages and Works_In

<u>**Weak Entities:**</u>

An entity set attributes that does not have a primary key within them, is termed as a **weak entity set**. As an example, consider the entity set Dependents, which has the two attributes pname and age, illustrated with the ER diagram as shown in **Figure 2.11**

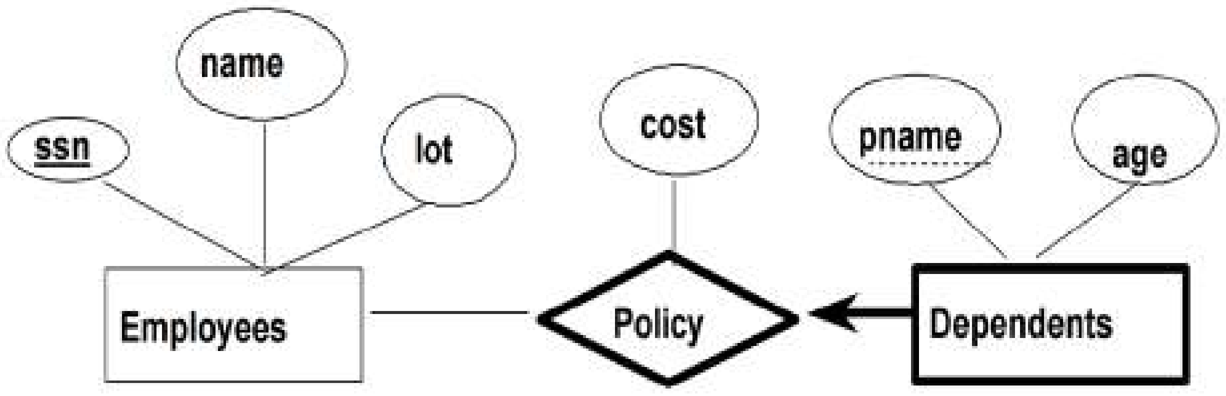


Figure 2.11 A Weak Entity Set

A dependent is an example of a weak entity set. A weak entity can be identified uniquely only by considering some of its attributes in conjunction with the primary key of another entity, which is called the identifying owner.

The following restrictions must hold:
→The owner entity set and the weak entity set must participate in a one-to-many relationship set (one owner entity is associated with one or more weak entities, but each weak entity has a single owner).
→This relationship set is called the identifying relationship set of the weak entity set. The weak entity set must have total participation in the identifying relationship set.

The Dependents weak entity set and its relationship to Employees is shown in **Figure 2.11.** The total participation of Dependents in Policy is indicated by linking them with a dark line. The arrow from Dependents to Policy indicates that each Dependents entity appears in at most one Policy relationship. To underscore the fact that Dependents is a weak entity and Policy is its identifying relationship, we draw both with dark lines. To indicate that pname is a partial key for Dependents, we underline it using a broken line. This means that there may well be two dependents with the same pname value.

Class Hierarchies:
To classify the entities in an entity set into subclass entity is known as **class hierarchies.**
Example: we might want to classify Employees entity set into subclass entities Hourly_Emps entity set and a Contract _Emps entity set to distinguish the basis on which they are paid. Then the class hierarchy is illustrated as shown in **Figure 2.12.**

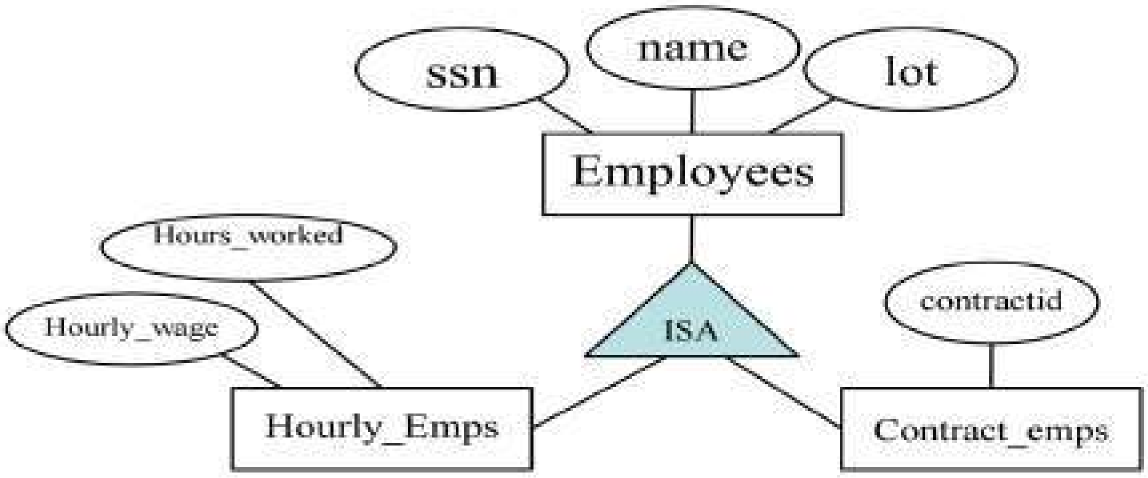


Figure 2.12 Class hierarchy

This **class hierarchy** illustrates the inheritance concept. Where, the subclass attributes ISA (read as: is a) superclass attributes, indicating the "is a" relationship (inheritance concept).
Therefore, the attributes defined for a Hourly_Emps entity set are the attributes of Hourly_Emps plus attributes of employees (because subclass can have superclass properties).
Likewise the attributes defined for a Contract_Emps set are the attributes of Contract_Emps plus attributes of Employees.

→A **class hierarchy** can be viewed in one of two ways:
Specialization:
→An **employee is** specialized into subclasses. Specialization is the process of identifying subsets (subclasses) of an entity set (the superclass) that share some distinguishing characteristics. Here, the superclass (Employees) is defined first, the subclasses (Hourly_Emps, Contract_Emps etc.) are defined next and subclass-specific attributes and relationship sets are then added.

Generalization:
→Generalization is the process of identifying(defining) some generalized (common) characteristics of a collection of (two or more) entity sets and creating a new entity set that contains entities possessing these common characteristics. Here, the subclasses (Hourly_Emps, Contract_Emps, etc.,) are defined first the superclass (Employees) is defined next.
In shortly, Hourly_Emps and Contract_Emps are generalized by Employees.

→The **class hierarchy** can specify two kinds of constraints. They are
Overlapped Constraints:
Overlap constraints determine whether two subclasses are allowed to contain the same entity.
Example: can Akbar be both a Hourly_Emps entity and a Contract_Emps entity? The answer is no.
Other example, can Akbar be both a Contract_Emps entity and a Senior_Emps entity (among them) the answer is, Yes.
Thus, this is a specialization hierarchy property. We denote this by writing "Contract_Emps overlaps Senior_Emps".

Covering Constraints:
Covering constraints determine whether the entities in the subclasses collectively include all entities in the superclass.

Example: Should every Employees entity be a Hourly_Emps or Contract_Emps? The answer is, No. He can be a Daily_Emps.
→Other example, should every Motor_Vehicle (superclass) be a bike (subclass) or a car (subclass)? The answer is yes.
Thus generalization hierarchies' property is that every instance of a superclass is an instance of a subclass.
We denote this by writing "bikes and cars cover Motor_Vehicles"

AGGREGATION: (*************************)

→Used when we have to model a relationship involving (entity sets and) a relationship set.
→**Aggregation** allows us to indicate that a relationship set (identified through a dashed box) participates in another relationship set.
→**Aggregation** allows a relationship set to be treated as an entity set for purposes of participation in (other) relationship sets.
This is illustrated in **Figure**, with a dashed box around **Sponsors** (and its participating entity sets) used to denote aggregation. This effectively allows us to treat Sponsors as an entity set for purposes of defining the **Monitors** relationship set.

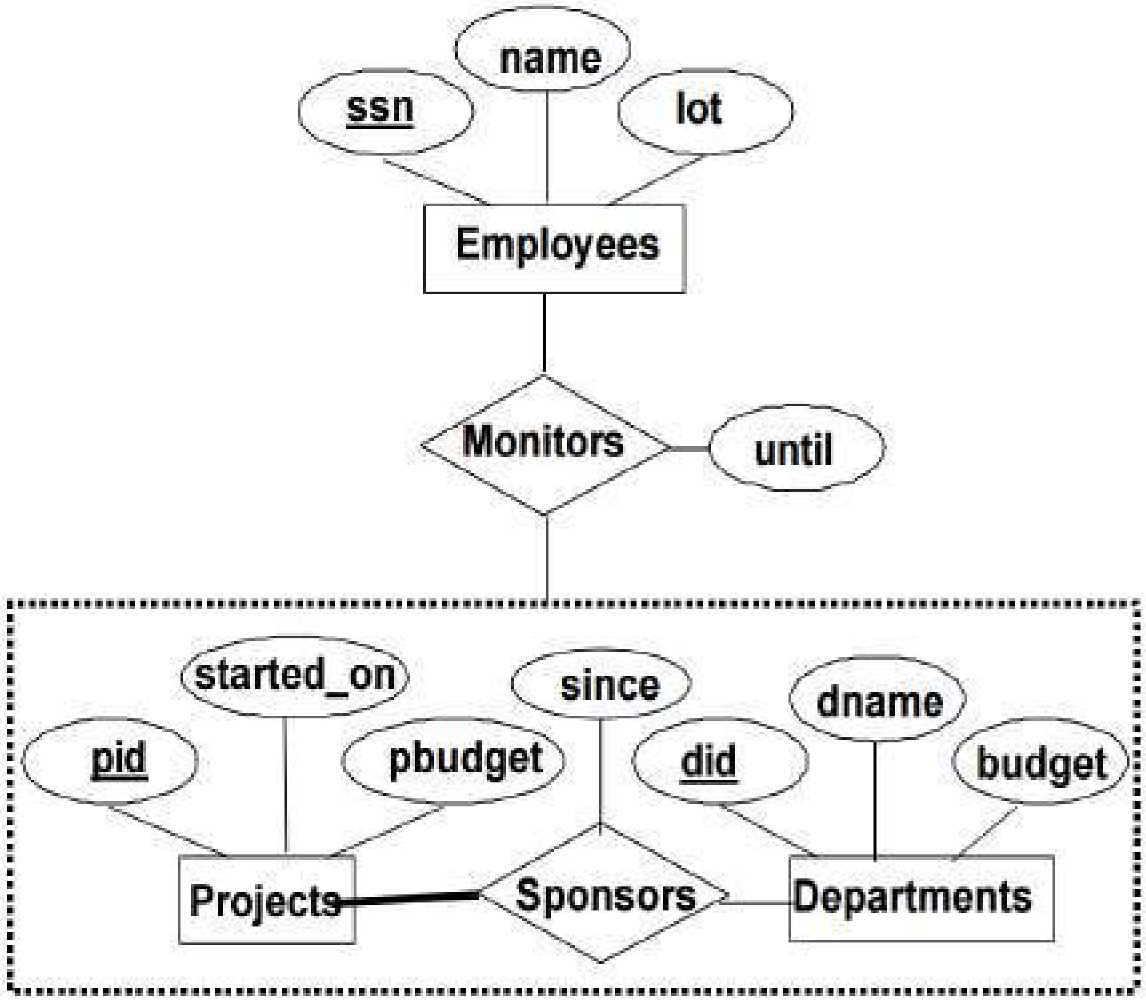
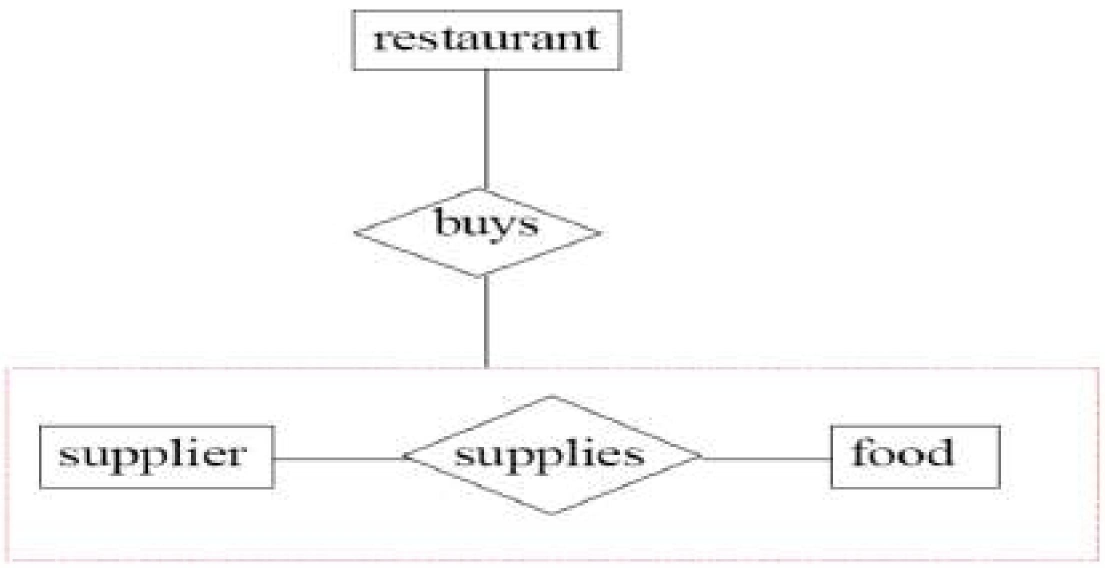


Figure. Aggregation

Restaurant Example:



Note:

→For Ternary relationship, it can only see which specific restaurant buys what kind food from which supplier.

→For Aggregation, you have more information about which supplier supplies a food item. Any restaurant needs that item can choose from that list.

Uses of Aggregation:

We use an aggregation, when we need to express a relationship among relationships. Thus, there are really two distinct relationships, sponsors and monitors, each with its own attributes.

Example: The Monitors relationship has an attribute until that records the ending date until when the employee is appointed as the sponsorship monitoring.

5. CONCEPTUAL DATABASE DESIGN WITH ER- DIAGRAMS:

Developing an ER diagram presents several **design choices**, including the following:

- **Should a concept be modeled as an entity or an attribute?**
- **Should a concept be modeled as an entity or a relationship?**
- **What are the relationship sets and their participating entity sets? Should we use binary or ternary relationships?**
- **Should we use aggregation**?

Entity Vs Attributes:

While identifying the attributes of an entity set, it is sometimes not clear, whether a property should be modeled as an attribute or as an entity set.

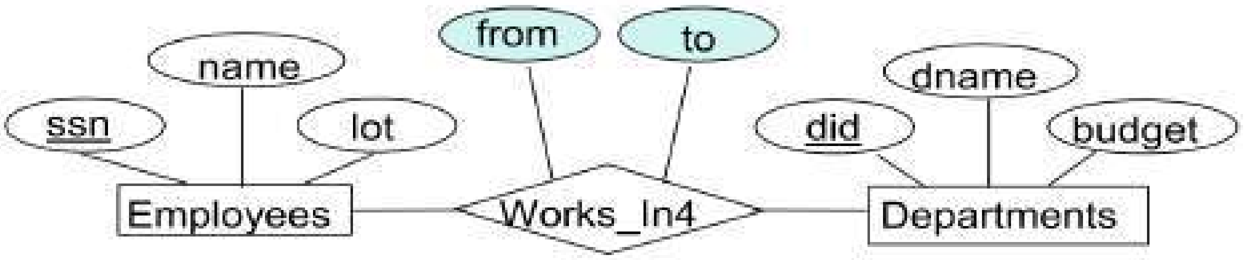
→Should **address** be an attribute of Employees or an entity (connected to Employees by a relationship)?

→Depends upon the use we want to make of address information, and the semantics of the data:

- If we have several addresses per employee, address must be an entity (since attributes cannot be set-valued).
- If the structure (city, street, etc.,) is important, e.g., we want to retrieve employees in a given city, address must be modeled as an entity (since attributes values are atomic).

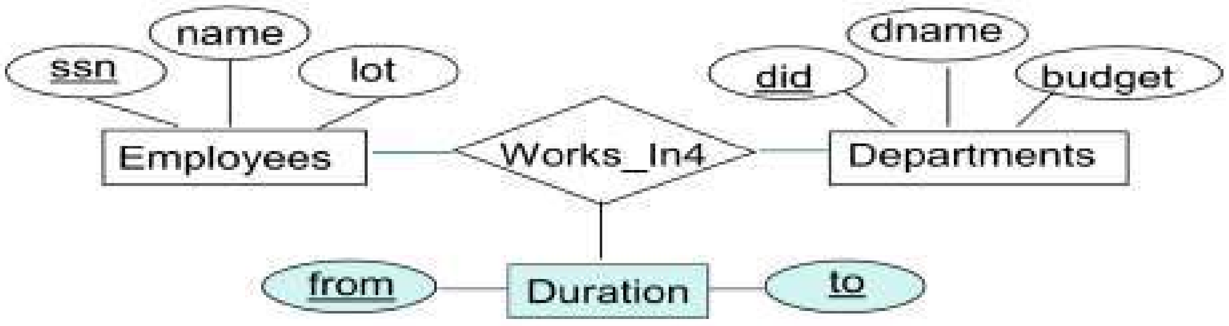
- **Works-In4** does not allow an employee to work in a department for two or more periods. A relationship is uniquely identified by the participating entities.

$$R = \{(e_1, \ldots, e_n) \mid e_1 \in E_1, \ldots, e_n \in E_n\}$$



- Similar to the problem of wanting to record several working periods for an employee in Work_In4. We want to record **several values of the descriptive attributes for each instance of this relationship.** Accomplished by introducing new entity set**, Duration.**

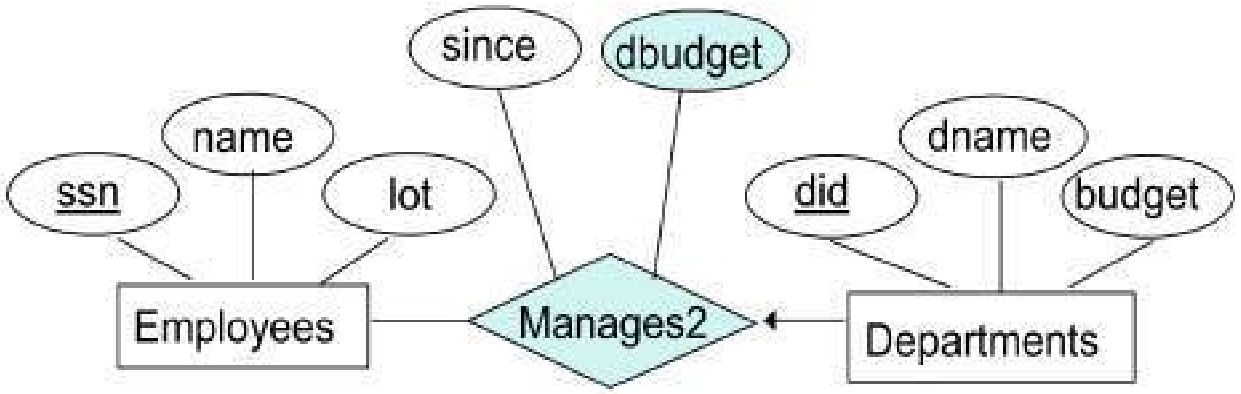
.



Entity Vs Relationship:

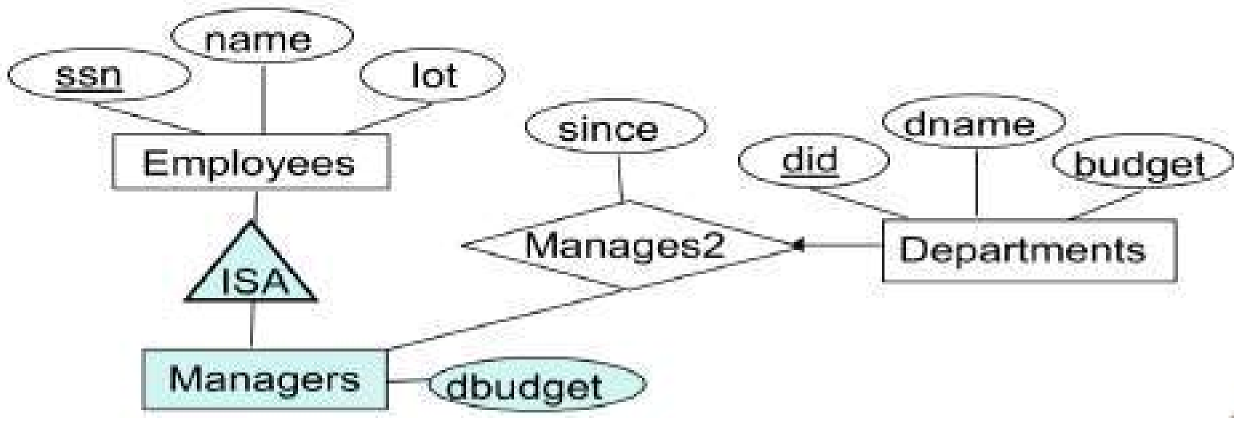
It is not always clear whether an object is best expressed by an entity set or a relationship set.

Example: If a manager gets a separate discretionary budget (dbudget) for each department he or she manages.



What if a manager gets a discretionary budget that covers all managed departments?

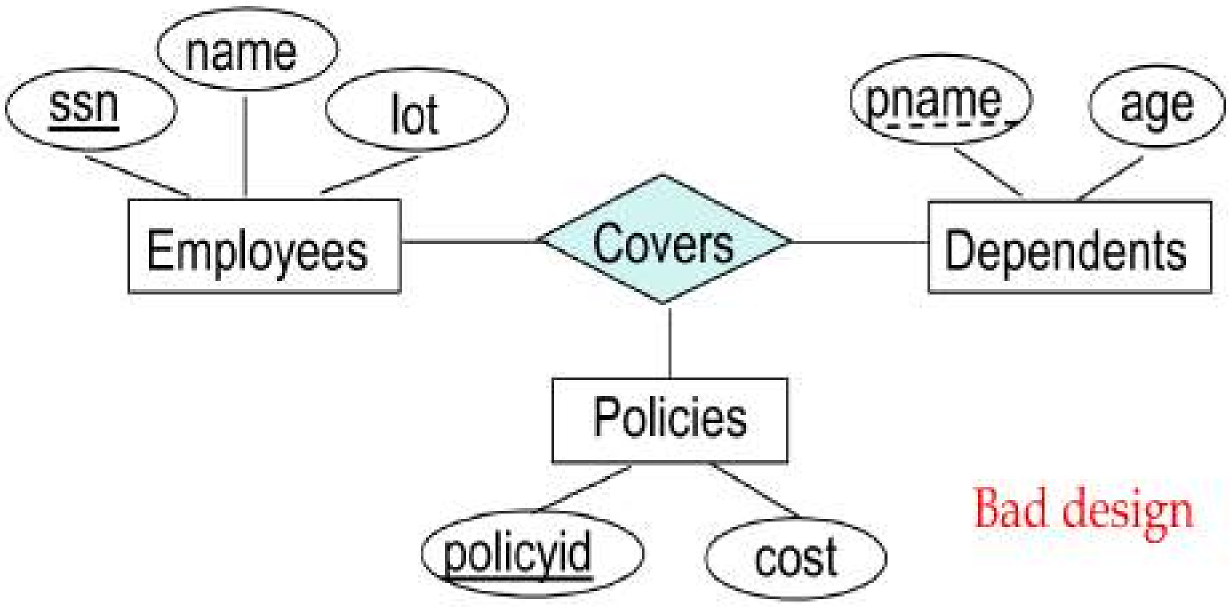
- **Redundancy:** dbudget stored for each department managed by the manager.
- **Misleading:** suggests **dbudge**t is associated with department – manager combination.



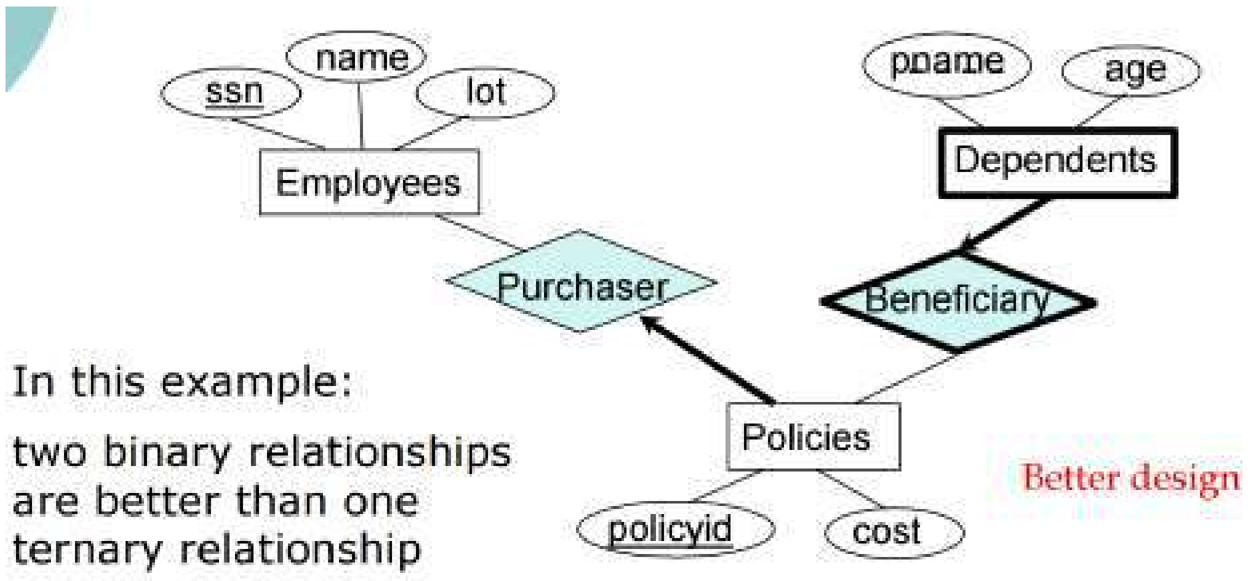
Binary versus Ternary Relationships:
It is always possible to replace a non-binary (n-array, for n>2) relationship set by a number of distinct binary relationship sets.

- A **Bad design** below if:
  Each policy is owned by just 1 employee, and, Dependents is a weak entity set, and each dependent is tied to the covering policy.

  Bad design: Policies involves in two relationships.



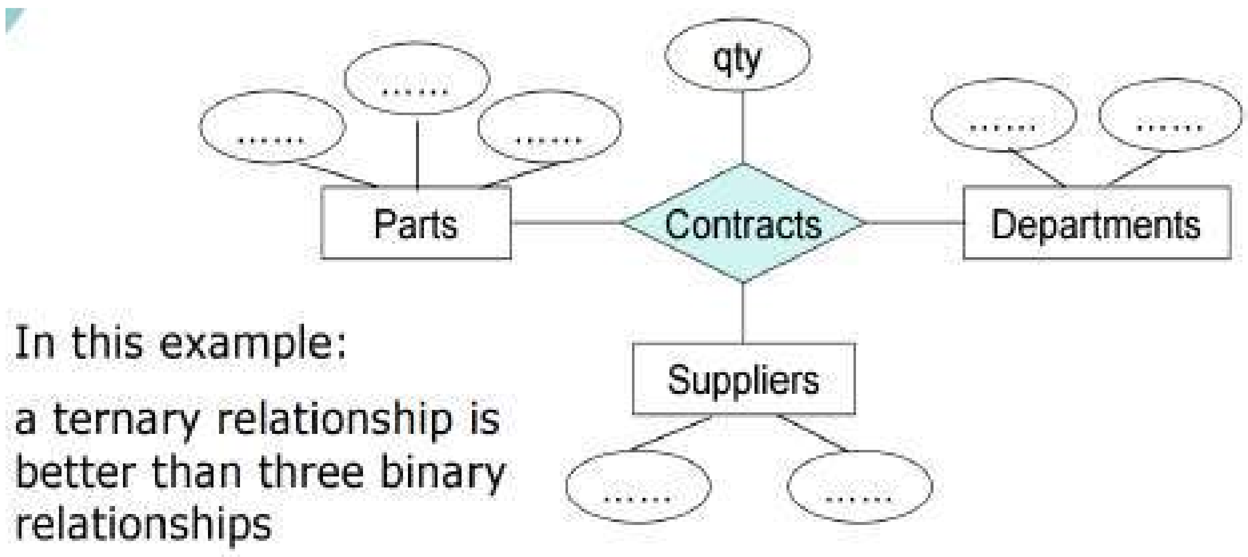
What are the additional constraints in 2nd diagram?
Better design:



In this example:

two binary relationships are better than one ternary relationship

Another example: The contract specifies that a supplier will supply some quantity of a part to a department.



In this example:

a ternary relationship is better than three binary relationships

<u>**Aggregation Vs Ternary Relationships:**</u>

→The choice between using aggregation or a ternary relationship is mainly determined by the existence of a relationship that relates a relationship set to an entity set (or second relationship set). The choice may also be guided by certain integrity constraints that we want to express.
→ The **monitors** is a distinct relationship, with a descriptive attribute. (In Figure)

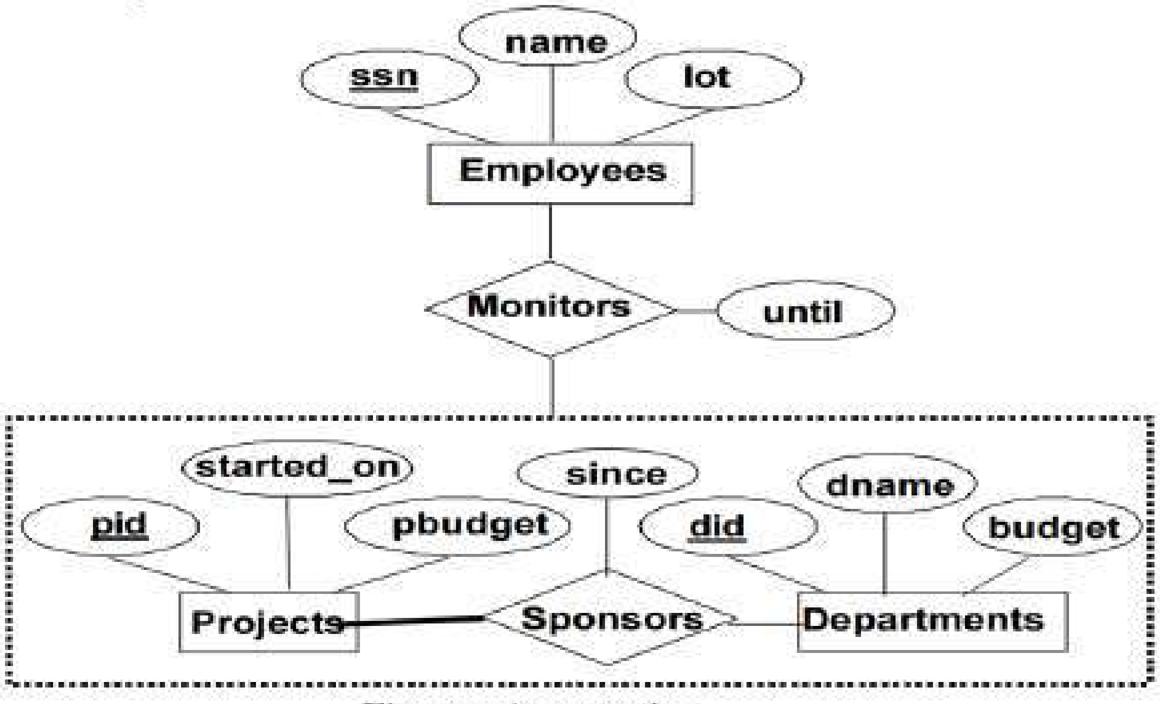


Figure. Aggregation

→If we don't need to record the ***until*** attribute of Monitors, then we might reasonably use a **ternary relationship. (In Figure)**

→Also, it can say that each sponsorship is monitored by <u>at most one employee</u>.

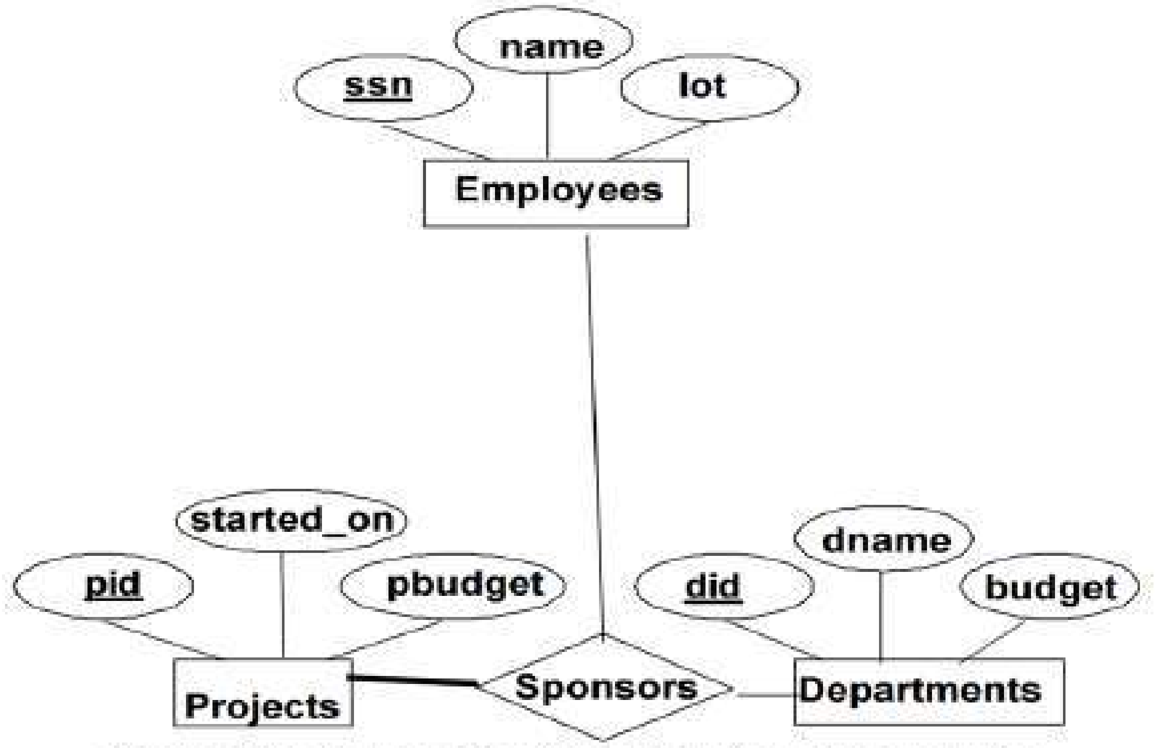


Figure. Using Ternary Relationship Instead of Aggregation.

<u>ER DIAGRAMS EXAMPLES</u>

1) Entity Relationship (ER) Modeling - Learn with a Complete Example

Here we are going to design an Entity Relationship (ER) model for a **college database.** Say we have the following statements.

1. **A college contains many departments**
2. **Each department can offer any number of courses**
3. **Many instructors can work in a department**
4. **An instructor can work only in one department**
5. **For each department there is a Head**
6. **An instructor can be head of only one department**
7. **Each instructor can take any number of courses**
8. **A course can be taken by only one instructor**
9. **A student can enroll for any number of courses**
10. **Each course can have any number of students**

Step 1: Identify the Entities

What are the entities here?

From the statements given, the entities are

1. Department
2. Course
3. Instructor
4. Student

Step 2: Identify the relationships

1. One department offers many courses. But one particular course can be offered by only one department. hence the cardinality between department and course is One to Many (1:N)
2. One department has multiple instructors. But instructor belongs to only one department. Hence the cardinality between department and instructor is One to Many (1:N)
3. One department has only one head and one head can be the head of only one department. Hence the cardinality is one to one. (1:1)
4. One course can be enrolled by many students and one student can enroll for many courses. Hence the cardinality between course and student is Many to Many (M:N)
5. One course is taught by only one instructor. But one instructor teaches many courses. Hence the cardinality between course and instructor is Many to One (N :1)

Step 3: Identify the key attributes

- "**Departmen_Name**" can identify a department uniquely. Hence Department_Name is the key attribute for the Entity "Department".
- **Course_ID** is the key attribute for "Course" Entity.
- **Student_ID** is the key attribute for "Student" Entity.
- **Instructor_ID** is the key attribute for "Instructor" Entity.

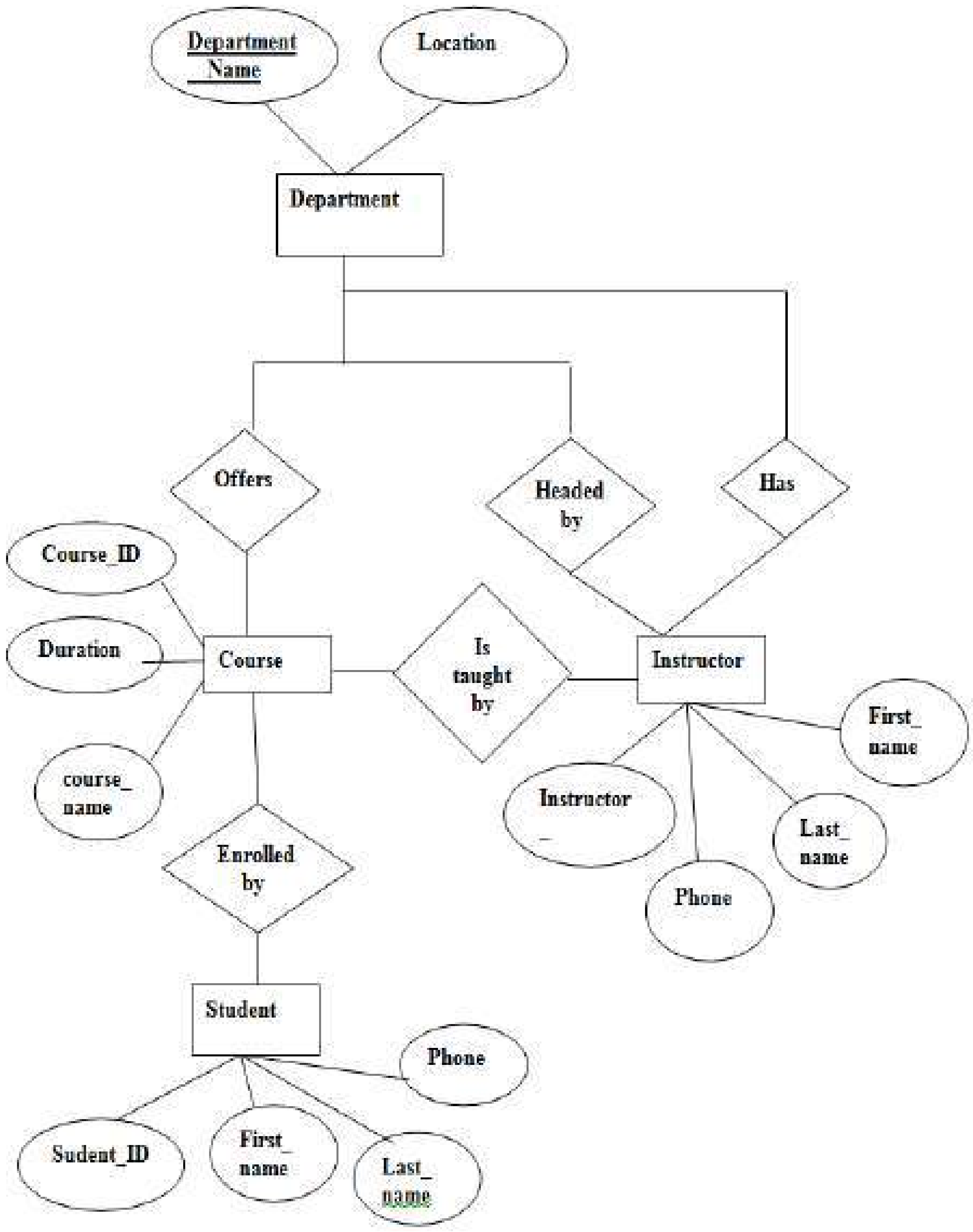
Step 4: Identify other relevant attributes

- For the department entity, other attributes are location

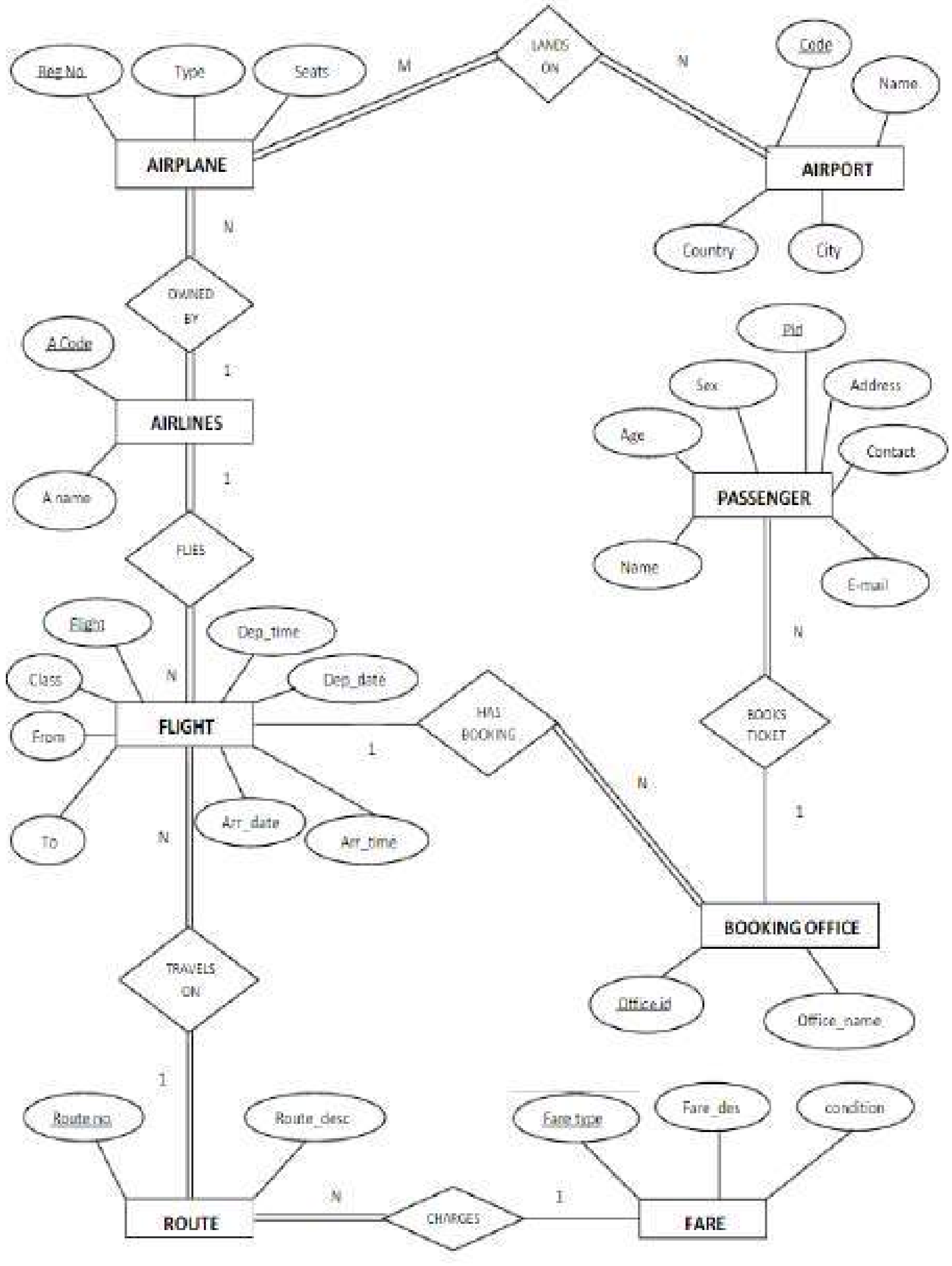
- For course entity, other attributes are course_name, duration
- For instructor entity, other attributes are first_name, last_name, phone
- For student entity, first_name, last_name, phone

Step 5: Draw complete ER diagram
By connecting all these details, we can now draw ER diagram as given below.

2) ER DIGRAM FOR COLLEGE DATABSAE



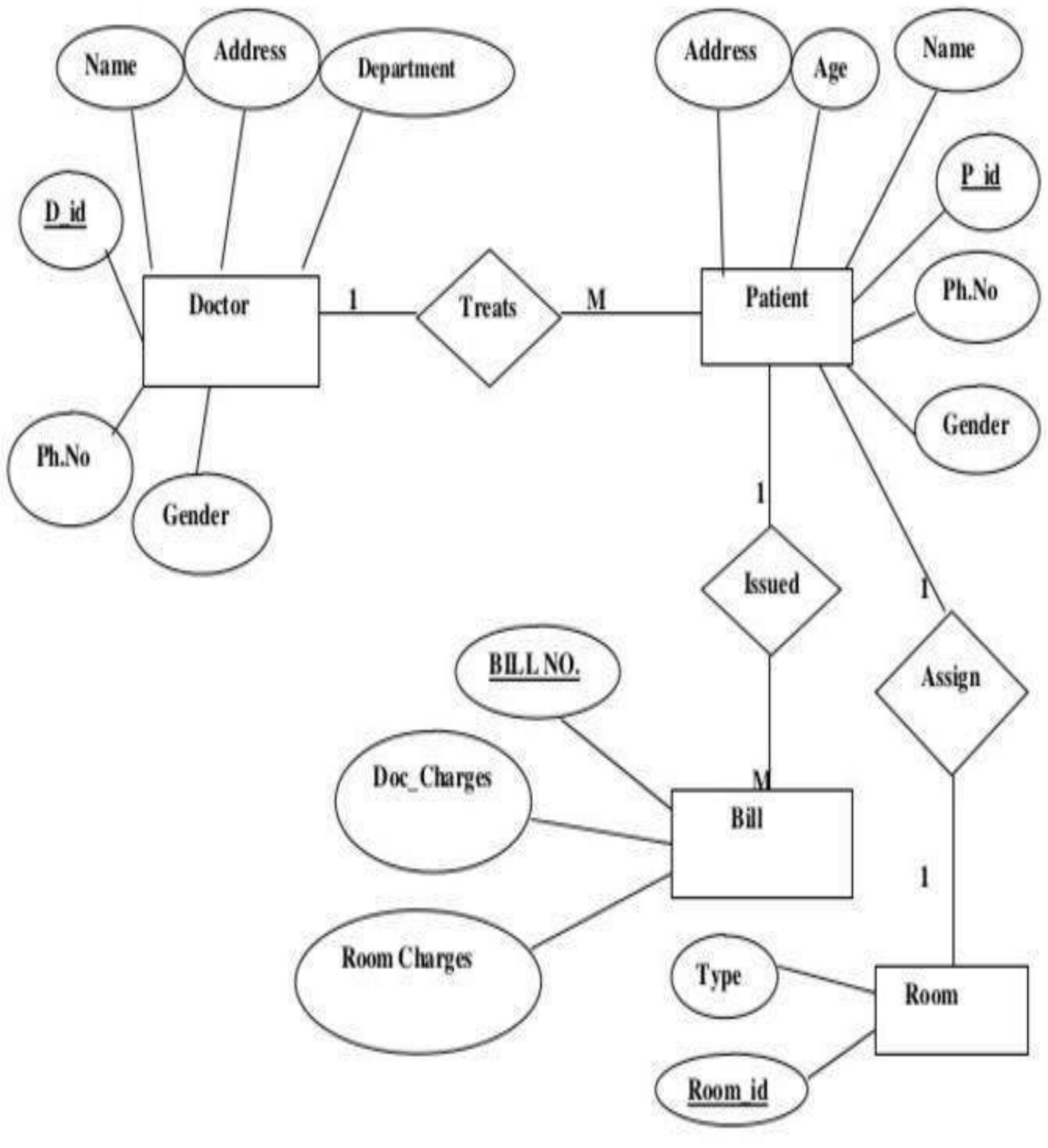
3) ER DIGRAM FOR AIRLINES - RESERVATION SYSTEM



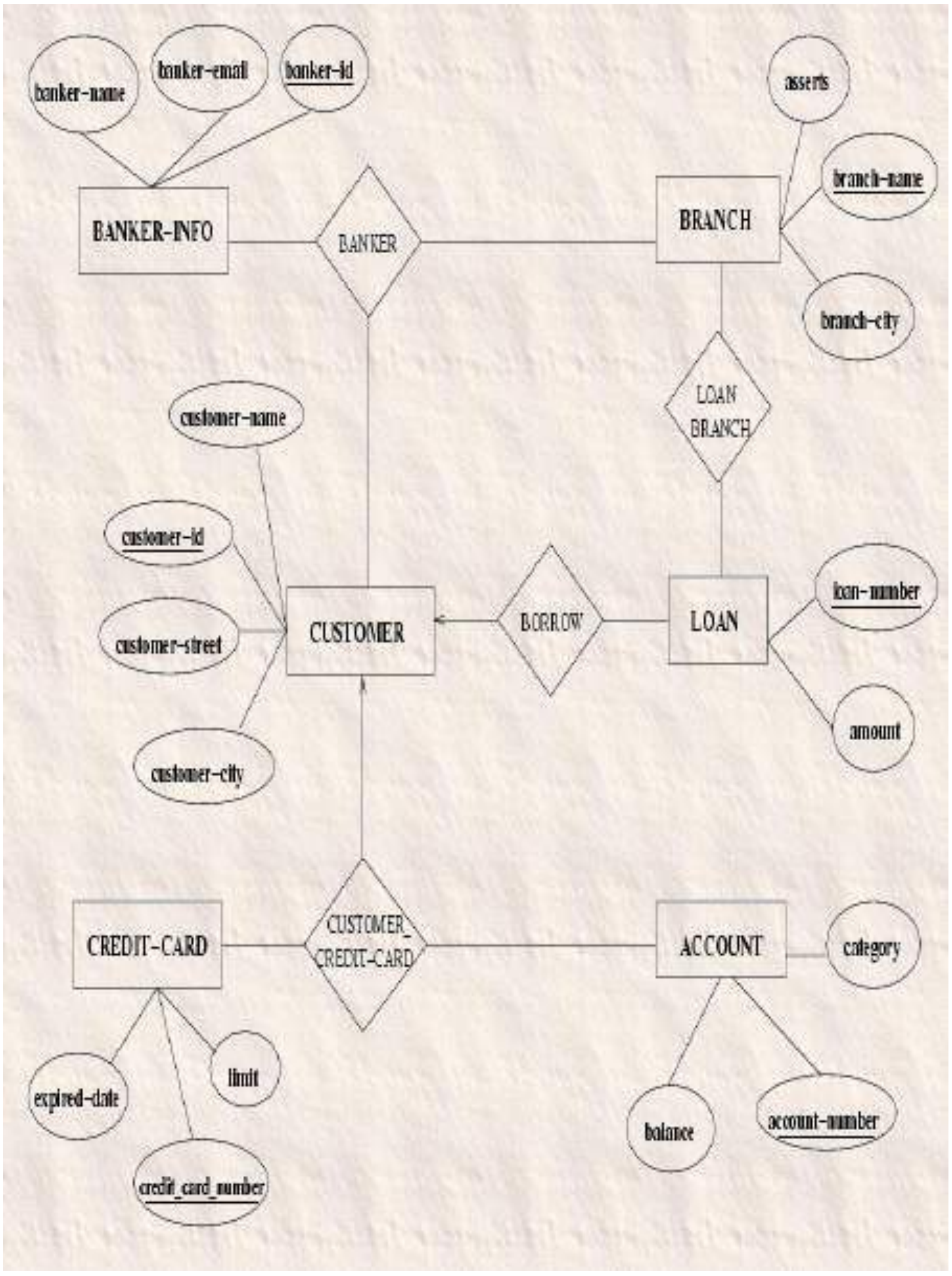
4) ER Diagram for Hospital Management System:

5) ER Diagram for Banking System

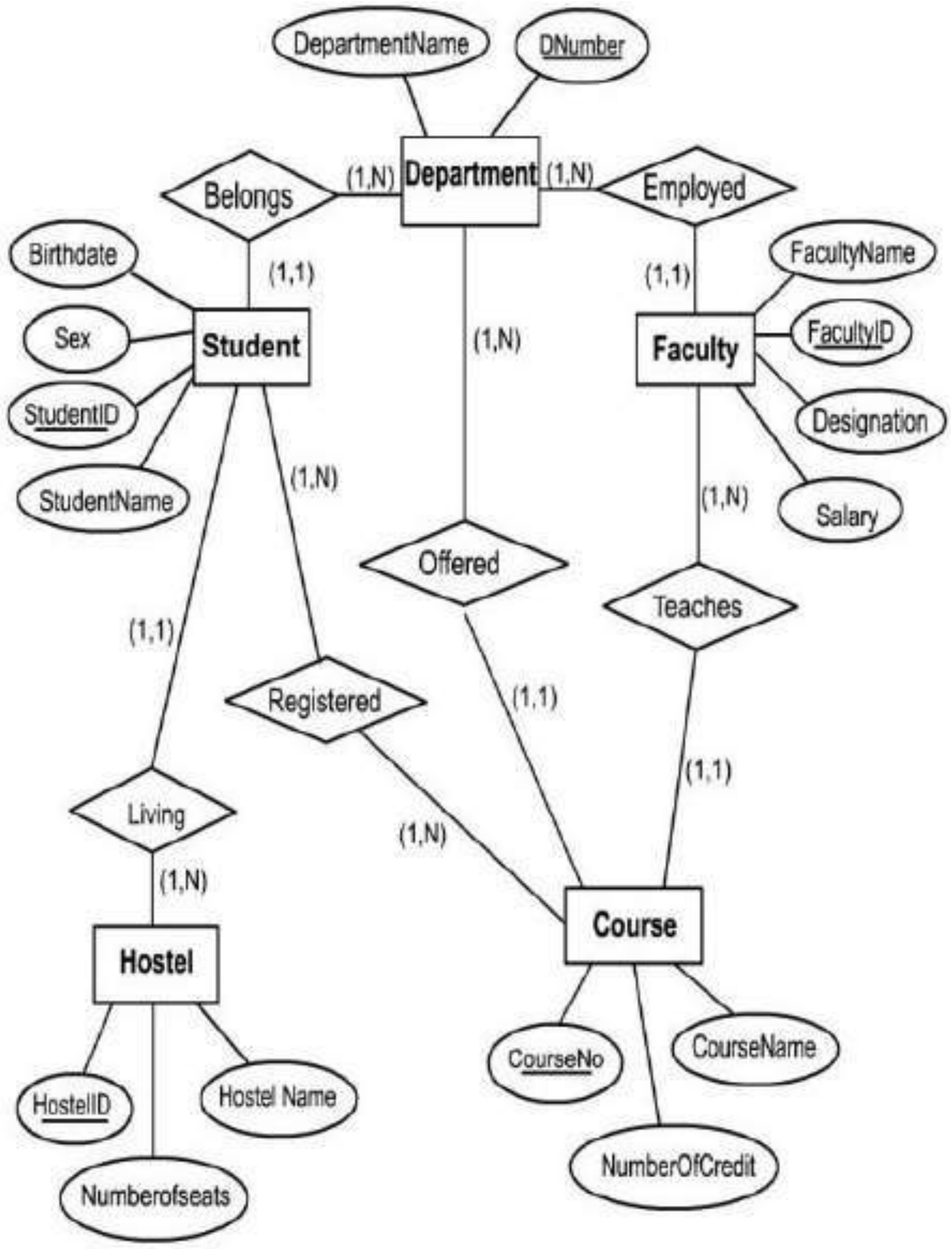
E-R Diagram of Hospital Management System



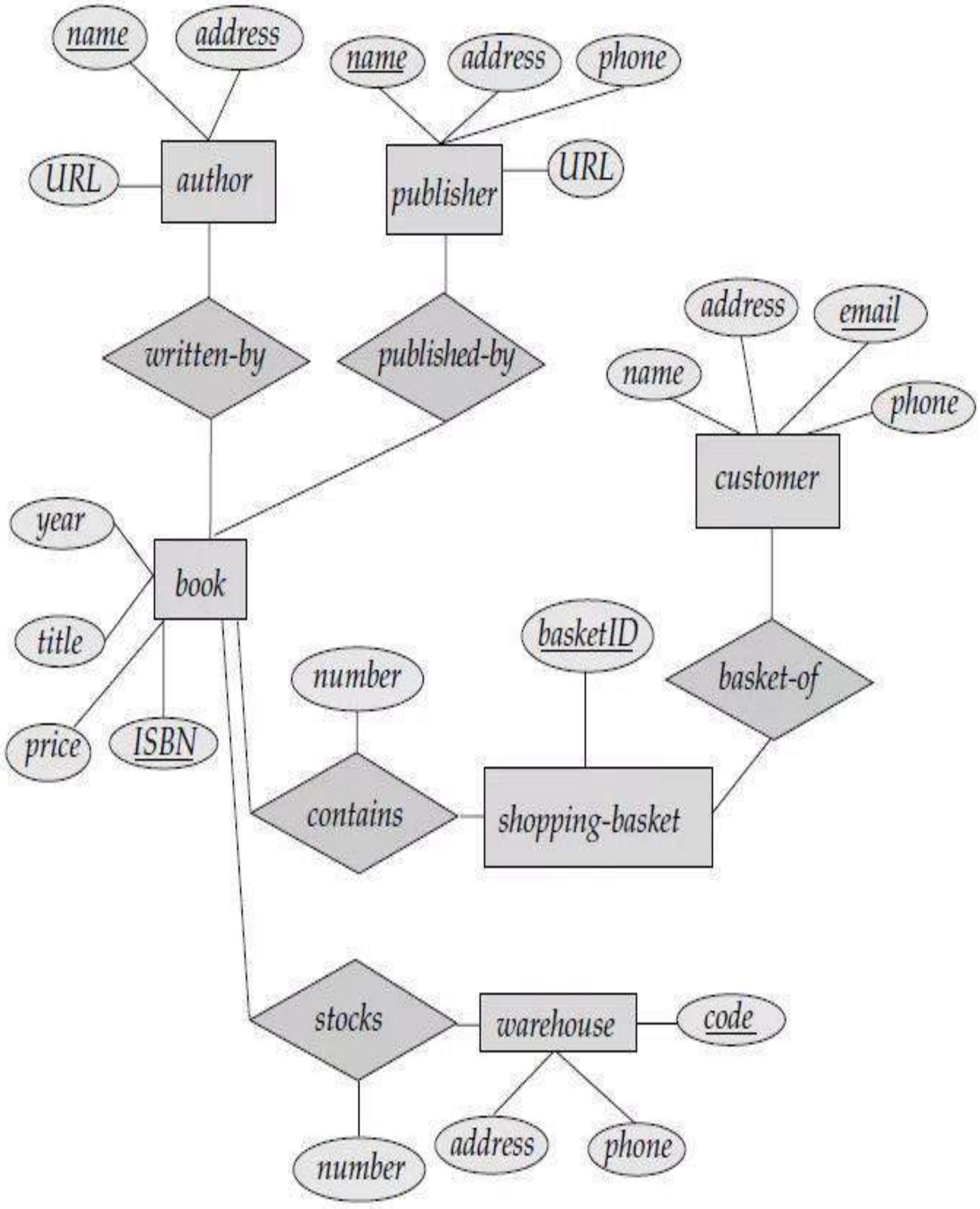
5) ER Diagram for Banking System



6) ER Diagram for College management system



7) ER Diagram for Online Book Store

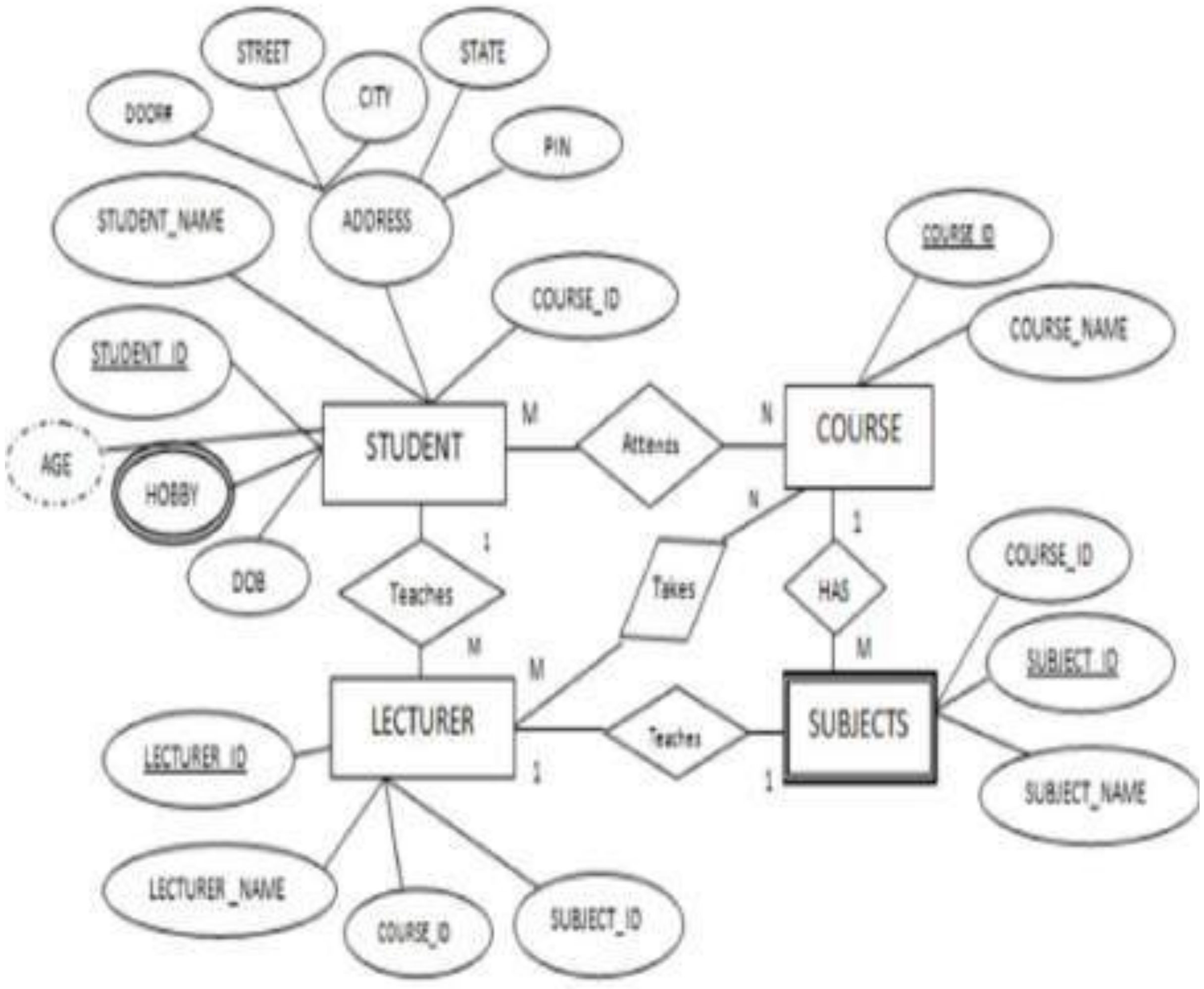


ER Diagram for Online BookStore

TRANSFORM ER DIAGRAM INTO TABLES

There are various steps involved in converting it into tables and columns. Each type of entity, attribute and relationship in the diagram takes their own depiction here. Consider the

ER diagram below and will see how it is converted into tables, columns and mappings.



The basic rule for converting the ER diagrams into tables is

- **Convert all the Entities in the diagram to tables.**

All the entities represented in the rectangular box in the ER diagram become independent tables in the database. In the below diagram, STUDENT, COURSE, LECTURER and SUBJECTS forms individual tables.

- **All single valued attributes of an entity is converted to a column of the table**

All the attributes, whose value at any instance of time is unique, are considered as columns of that table. In the STUDENT Entity, STUDENT_ID, STUDENT_NAME form the columns of STUDENT table. Similarly, LECTURER_ID, LECTURER_NAME form the columns of LECTURER table. And so on.

- **Key attribute in the ER diagram becomes the Primary key of the table.**

In diagram above, STUDENT_ID, LECTURER_ID, COURSE_ID and SUB_ID are the key attributes of the entities. Hence we consider them as the primary keys of respective table.

- **Declare the foreign key column, if applicable.**

In the diagram, attribute COURSE_ID in the STUDENT entity is from COURSE entity. Hence add COURSE_ID in the STUDENT table and assign it foreign key constraint. COURSE_ID and SUBJECT_ID in LECTURER table forms the foreign key column. Hence by declaring the foreign key constraints, mapping between the tables are established.

- **Any multi-valued attributes are converted into new table.**

A hobby in the Student table is a multi-valued attribute. Any student can have any number of hobbies. So we cannot represent multiple values in a single column of STUDENT table. We need to store it separately, so that we can store any number of hobbies, adding/ removing / deleting hobbies should not create any redundancy or anomalies in the system. Hence we create a separate table STUD_HOBBY with STUDENT_ID and HOBBY as its columns. We create a composite key using both the columns.

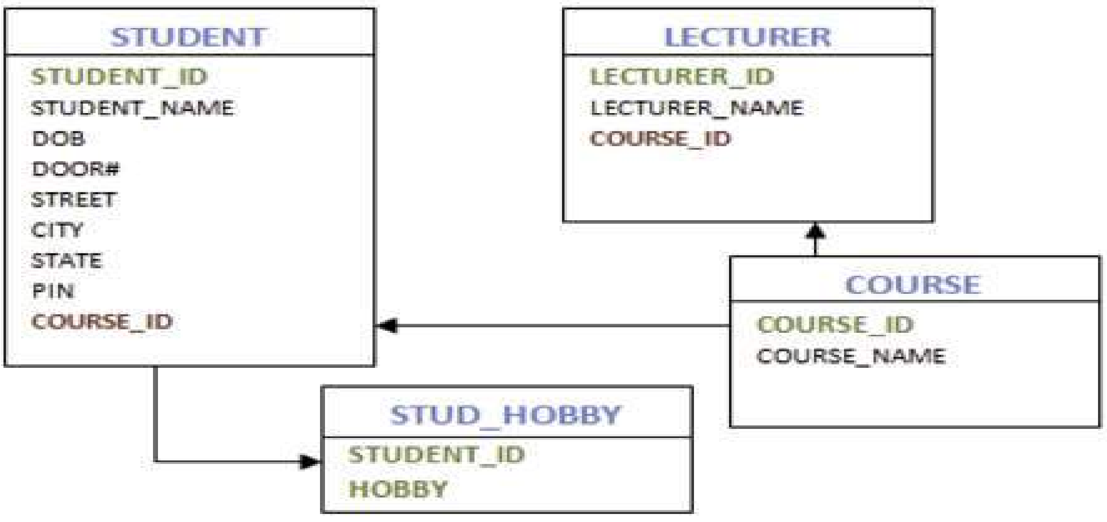
- **Any composite attributes are merged into same table as different columns.**

In the diagram above, Student Address is a composite attribute. It has Door#, Street, City, State and Pin. These attributes are merged into STUDENT table as individual columns.

- **One can ignore derived attribute, since it can be calculated at any time.**

In the STUDENT table, Age can be derived at any point of time by calculating the difference between DateOfBirth and current date. Hence we need not create a column for this attribute. It reduces the duplicity in the database.

→These are the very basic rules of converting ER diagram into tables and columns, and assigning the mapping between the tables. Table structure at this would be as below:



<u>RELATIONAL MODEL:</u>

→The **Relational Database** is a collection of one or more relations, where each relation is a table with rows and columns.

→The main construct for representing data in the relational model is a relation (table). A relation consists of a **<u>relation schema and a relation instance.</u>** The relation instance is a table, and the relation schema describes the column heads for the table.

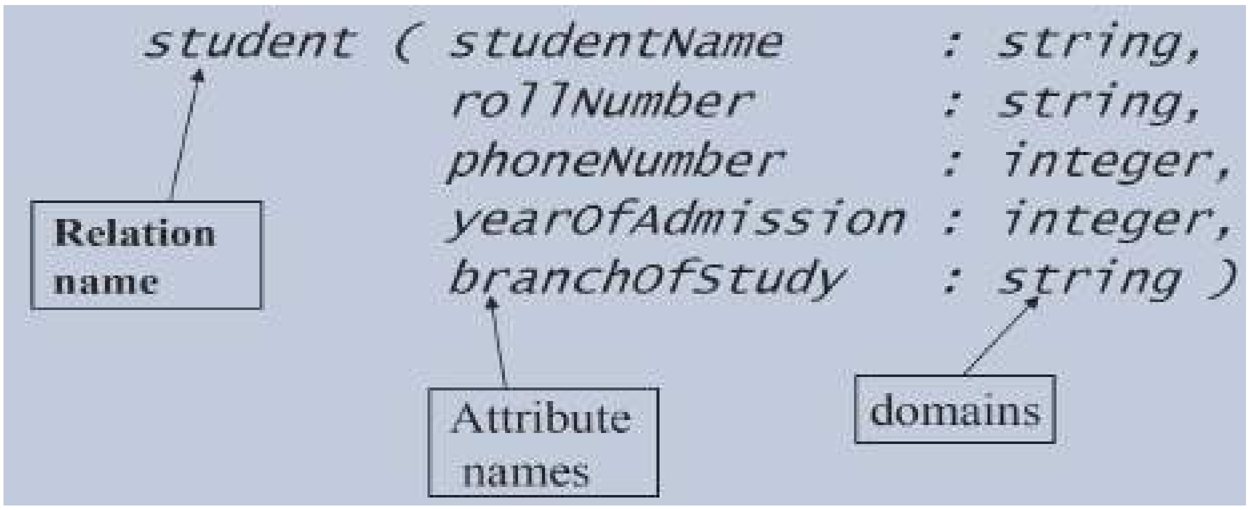
→The **schema** specifies the relation's name, the name of each field (or column, or attribute), and the domain of each field. A **domain** is referred to in a relation schema by the domain name and has a set of associated values.

Example of student information in a university database to illustrate the parts of a relation schema:

Students (sid: string, name: string, login: string, age: integer, gpa: real)

The field named sid has a domain named string. The set of values associated with domain string is the set of all character strings.

Example2:

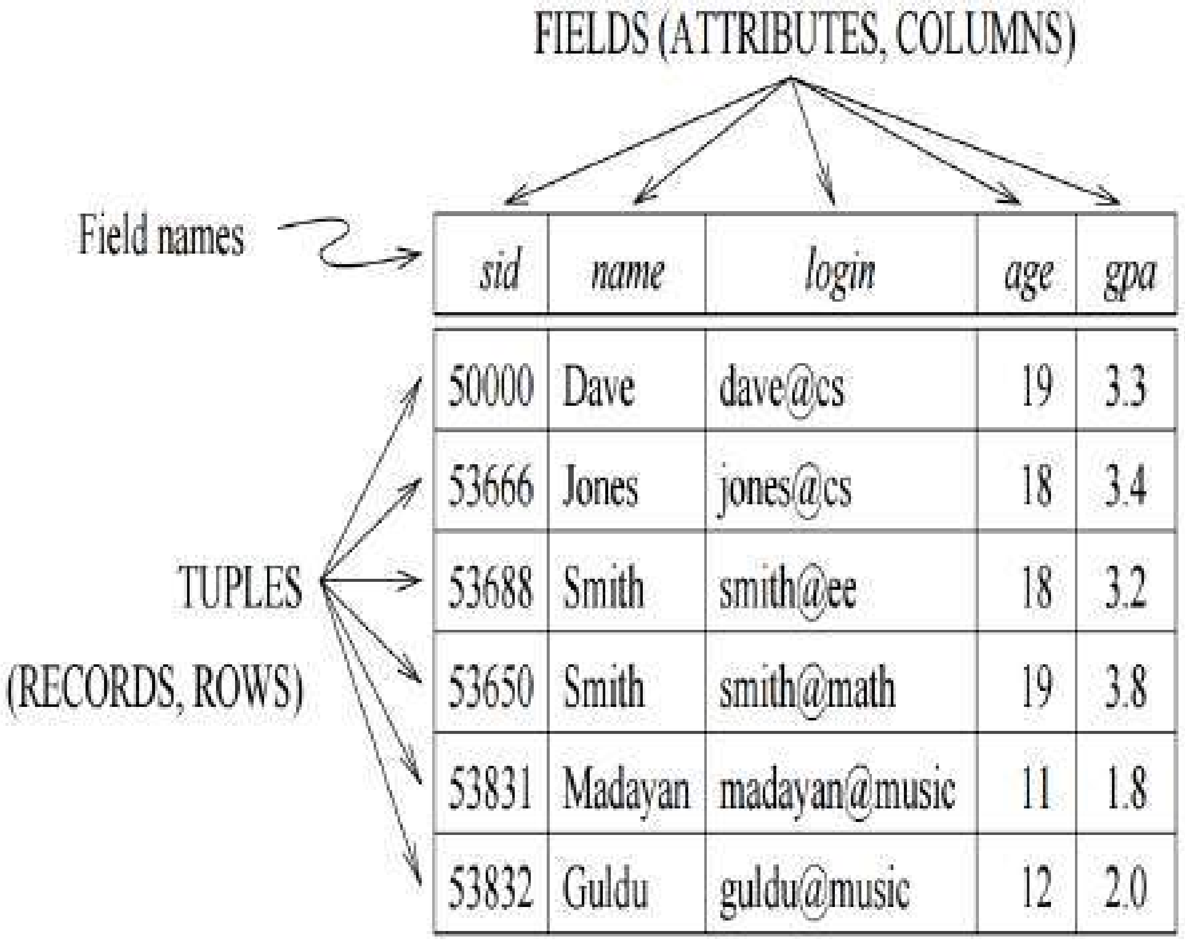


Domain–set of *atomic* (or *indivisible*) values –data type

→An **instance of a relation** is a set of **tuples**, also called **records**, in which each tuple has the same number of fields as the relation schema. A relation instance can be thought of as a table in which each tuple is a row, and all rows have the same number of fields

An instance of the Students relation appears in **Figure 3.1.**

The instance S1 contains six tuples and has, as we expect from the schema, five fields. Note that no two rows are identical. This is a requirement of the relational model—each relation is defined to be a set of unique tuples or rows.


FIELDS (ATTRIBUTES, COLUMNS)
Field names
TUPLES
(RECORDS, ROWS)

sid	*name*	*login*	*age*	*gpa*
50000	Dave	dave@cs	19	3.3
53666	Jones	jones@cs	18	3.4
53688	Smith	smith@ee	18	3.2
53650	Smith	smith@math	19	3.8
53831	Madayan	madayan@music	11	1.8
53832	Guldu	guldu@music	12	2.0

Figure 3.1 An Instance S1 of the Students Relation

Cardinality = 3, degree = 5, all rows distinct.

→**Domain constraints** are so fundamental in the relational model that we will henceforth consider only relation instances that satisfy them; therefore, relation instance means relation instance that satisfies the domain constraints in the relation schema.

→The **degree**, also called **arity**, of a relation is the number of fields. The **cardinality of a relation instance** is the number of tuples in it. In **Figure 3.1**, the **degree of the relation** (the number of columns) is five, and the **cardinality** of this instance is six.

→A **relation schema** specifies the domain of each field or column in the relation instance. These **domain constraints** in the schema specify an important condition that we want each instance of the relation to satisfy: The values that appear in a column must be drawn from the domain associated with that column. Thus, the domain of a field is essentially the type of that field, in programming language terms, and restricts the values that can appear in the field.

More formally, let R(f_1:D1, …, f_n:Dn) be a relation schema, and for each f_i, $1 \leq i \leq n$, let Dom_i be the set of values associated with the domain named Di. An instance of R that satisfies the domain constraints in the schema is a set of tuples with n fields:

$$\{ \langle f_1 : d_1, \ \ldots, f_n : d_n \rangle \mid d_1 \in Dom_1, \ \ldots, d_n \in Dom_n \}$$

Another Example:

sid	*name*	*login*	*age*	*gpa*
53831	Madayan	madayan@music	11	1.8
53832	Guldu	guldu@music	12	2.0
53688	Smith	smith@ee	18	3.2
53650	Smith	smith@math	19	3.8
53666	Jones	jones@cs	18	3.4
50000	Dave	dave@cs	19	3.3

Figure 3.2 An Alternative Representation of Instance *S1* of Students

→A **relational database** is a collection of relations with distinct relation names. The **relational database schema** is the collection of schemas for the relations in the database. For example, University database with relations called Students, Faculty, Courses, Rooms, Enrolled, Teaches, and Meets In. An **instance of a relational database** is a collection of relation instances, one per relation schema in the database schema; of course, each relation instance must satisfy the domain constraints in its schema.

Creating and Modifying Relations Using SQL-92:
The SQL-92 language standard uses the word table to denote relation. The subset of SQL that supports the creation, deletion, and modification of tables is called the Data Definition Language (DDL).

Domain Types in SQL:

1. **char (n):** Fixed length character string, with user-specified length n.
2. **varchar (n) (or) character varying):** Variable length character strings, with user-specified maximum length n.
3. **int or integer:** An integer (a finite subset of the integers that is machine dependent).
4. **smallint:** a small integer (a machine-dependent subset of the integer domain type).
5. **numeric(p,d):** Fixed point number, with user-specified precision of p digits, with n digits to the right of decimal point.
6. **Real (or) double precision:** Floating point and double-precision floating point numbers, with machine-dependent precision.
7. **float (n):** Floating point number, with user-specified precision of at least n digits.
8. **date:** a calendar date, containing four digit year, month, and day of the month.
9. **time**: the time of the day in hours, minutes, and seconds.

→The **CREATE TABLE** statement is used to define a new table. To create the Students relation, we can use the following statement:

```
CREATE TABLE Students ( sid    CHAR(20),
                        name   CHAR(30),
                        login  CHAR(20),
                        age    INTEGER,
                        gpa    REAL )
```

→Tuples are inserted using the **INSERT** command. We can insert a single tuple into the Students table as follows:

```
INSERT
INTO    Students   (sid, name, login, age, gpa)
VALUES  (53688, 'Smith', 'smith@ee', 18, 3.2)
```

→We can delete tuples using the **DELETE** command. We can delete all Students tuples with name equal to Smith using the command:

```
DELETE
FROM    Students S
WHERE   S.name = 'Smith'
```

→We can modify the column values in an existing row using the **UPDATE** command. For example, we can increment the age and decrement the gpa of the student with sid 53688:

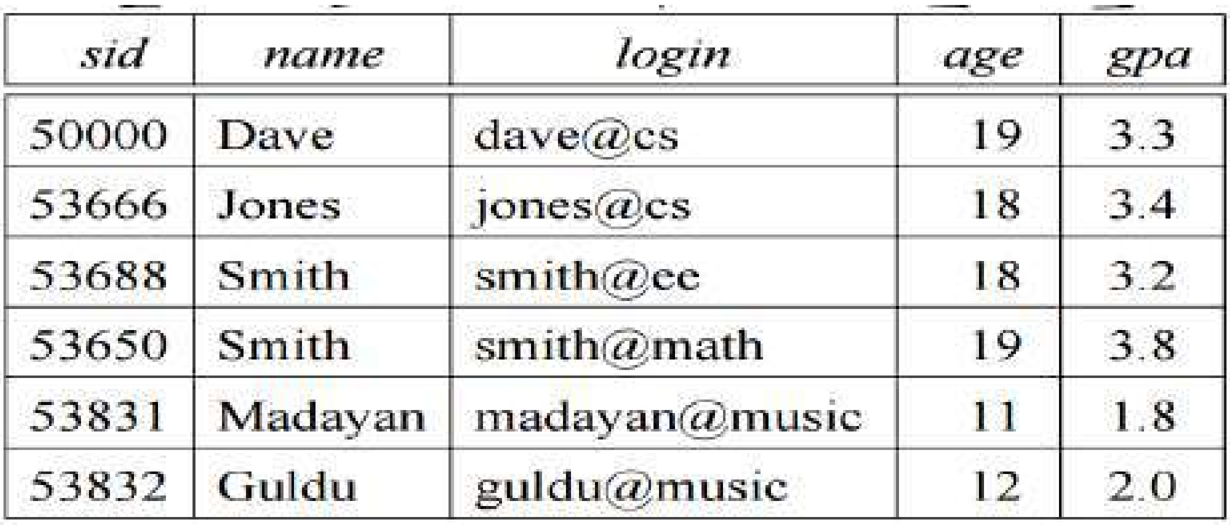
```
UPDATE  Students S
SET     S.age = S.age + 1, S.gpa = S.gpa - 1
WHERE   S.sid = 53688
```

→The **WHERE** clause is applied first and determines which rows are to be modified. The **SET** clause then determines how these rows are to be modified.

consider the following variation of the previous query:

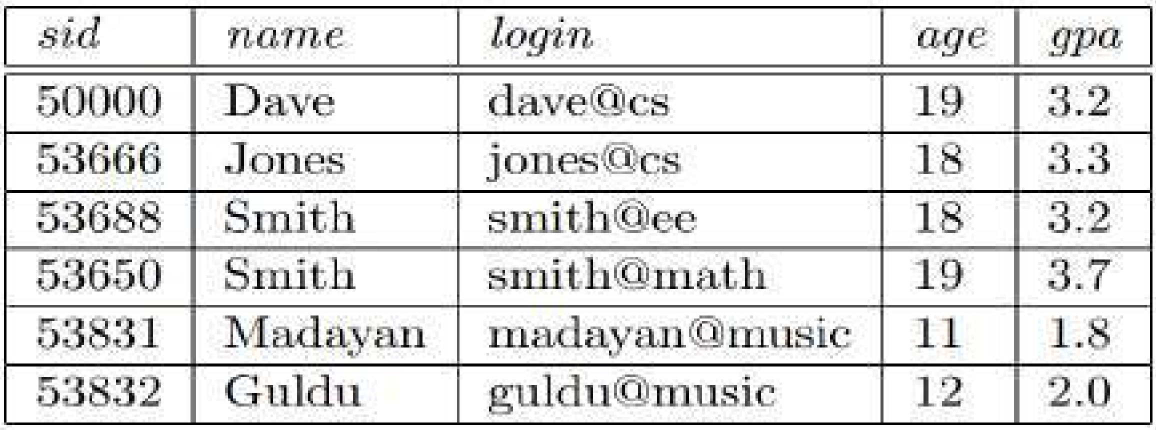
```
UPDATE  Students S
SET     S.gpa = S.gpa - 0.1
WHERE   S.gpa >= 3.3
```

→If this query is applied on the instance S1 of Students shown in **Figure 3.1**, we obtain the instance shown in **Figure 3.3.**



sid	*name*	*login*	*age*	*gpa*
50000	Dave	dave@cs	19	3.3
53666	Jones	jones@cs	18	3.4
53688	Smith	smith@ee	18	3.2
53650	Smith	smith@math	19	3.8
53831	Madayan	madayan@music	11	1.8
53832	Guldu	guldu@music	12	2.0

Figure 3.1 An Instance *S1* of the Students Relation



sid	*name*	*login*	*age*	*gpa*
50000	Dave	dave@cs	19	3.2
53666	Jones	jones@cs	18	3.3
53688	Smith	smith@ee	18	3.2
53650	Smith	smith@math	19	3.7
53831	Madayan	madayan@music	11	1.8
53832	Guldu	guldu@music	12	2.0

Figure 3.3 Students Instance *S1* after Update

2. INTEGRITY CONSTRAINTS OVER RELATIONS:(****)

An **integrity constraint (IC)** is a condition that ensures the correct insertion of the data and prevents unauthorized data access thereby preserving the consistency of the data.

For **example**, the roll number of a student cannot be a decimal value. The database enforces the constraint that the instance of roll number can have only integer values.

Integrity constraints are specified and enforced at different times:

1. When the DBA or end user defines a database schema, he or she specifies the ICs that must hold on any instance of this database.
2. When a database application is run, the DBMS checks for violations and disallows changes to the data that violate the specified ICs.

There are three types of integrity constraints in addition to domain constraint. They are:

1. **Key Constraints**
2. **Foreign Key Constraints.**
3. **General Constraints.**

<u>1). KEY CONSTRAINTS:</u>
→A **key constraint** is a statement that a certain minimal subset of the fields of a relation is a unique identifier for a tuple.
→Consider the Students relation and the constraint that no two students have the same student id. This IC is an example of a **key constraint**.

TYPES OF KEY CONSTRAINTS:

1. **Candidate key**
2. **Super key**
3. **Primary key**
4. **Foreign key**

<u>1. CANDIDATE KEY:</u>
→A candidate key is a collection of fields/columns/attributes that uniquely identifies a tuple.
→Let us take a closer look at the above definition of a (candidate) key.
→There are two parts to the definition:

1. Two distinct tuples in a legal instance (an instance that satisfies all ICs, including the key constraint) cannot have identical values in all the fields of a key.
2. No subset of the set of fields in a key is a unique identifier for a tuple.

<u>Example:</u> In "**customer**" relation the attribute "**cid**" is a key, it uniquely defines a tuple in a relation. No two rows in a relation "**customer**" can have the same "**cid"** value.
→The set of attributes that form a **candidate key** need not be all keys. The attributes may be treated as candidates to be taken as key.

<u>Example:</u> The set (**cid, cname**) is a **candidate key** which means either **cid** or **cname** can be taken as key but not both. Each of them independently and uniquely identifies a particular row. The alternate keys are candidate keys that are not taken as keys.

2. COMPOSITE KEY:

→Composite key consist of more than one attribute that uniquely identifies a tuple in a relation. All the attributes that form a set of keys and all of them taken together determines a unique row in a table.

Example: The set (**cid, accno**) is a **composite key** which maintains the uniqueness of each row. Both **cid, accno** are taken as keys.

3. SUPER KEY:

A **super key** is a combination of both **candidate key** and **composite key**. That is a set of attributes or a single attribute that uniquely identifies a tuple in a relation.

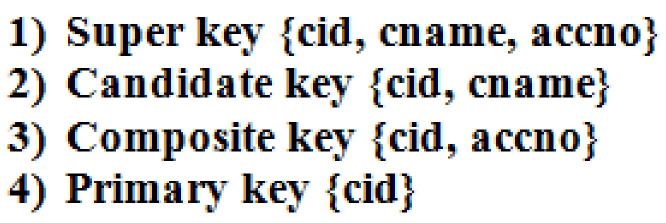
Example: Consider the **super key** {**cid, accno, cname**}
Here, all the three attributes taken together can identify a particular record or a combination of any two attributes can identify a particular record or any one of the attribute can identify a particular record.

4. PRIMARY KEY:

Only a single attribute can uniquely identify a particular record. More specifically, it can be defined as the candidate key, which has been selected as key to identify unique records.

Example: "cid" attribute in "customer" relation can be treated as PRIMARY KEY.

→Summary of Key (With respect to "customers" relation)



1) Super key {cid, cname, accno}
2) Candidate key {cid, cname}
3) Composite key {cid, accno}
4) Primary key {cid}

Specifying Key Constraints in SQL-92:

→In SQL, we can eliminate the chances of inserting duplicate data by using a unique constraint. This constraint helps the user to insert unique values for the columns which have been declared as unique, forming a candidate key any one of the columns among them can be declared as primary by using **primary key** constraints.

→**Example**, Consider the creation of "Students" table.

```
CREATE TABLE Students ( sid   CHAR(20),
                        name  CHAR(30),
                        login CHAR(20),
                        age   INTEGER,
                        gpa   REAL,
                        UNIQUE (name, age),
                        CONSTRAINT StudentsKey PRIMARY KEY (sid) )
```

This example shows the creation of Students table with attributes sid, name, login, age, gpa, unique key is used on columns name and age which ensures that the values inserted in these columns are unique. The last line of declaration defines a primary key constraint.

→The syntax used for defining constraint is,

```
CONSTRAINT constraint-name PRIMARY KEY (key)
i.e., CONSTRAINT StudentsKey PRIMARY KEY (sid)
```

The line declares sid as **primary key** for **Students relation**. If the user inserts repeated values for **”sid”** then error occurs and constraint-name is return indicating violation of constraint.

---

2). Foreign Key Constraints(*******)

→A **foreign key (FK)** is a column or combination of columns that is used to establish and enforce a link between the data in two tables. You can create a foreign key by defining a **FOREIGN KEY constraint** when you create or modify a table.

→In a **foreign key reference**, a link is created between two tables when the column or columns that hold the primary key value for one table are referenced by the column or columns in another table. This column becomes a foreign key in the second table.

→Suppose that in addition to Students, we have a second relation:

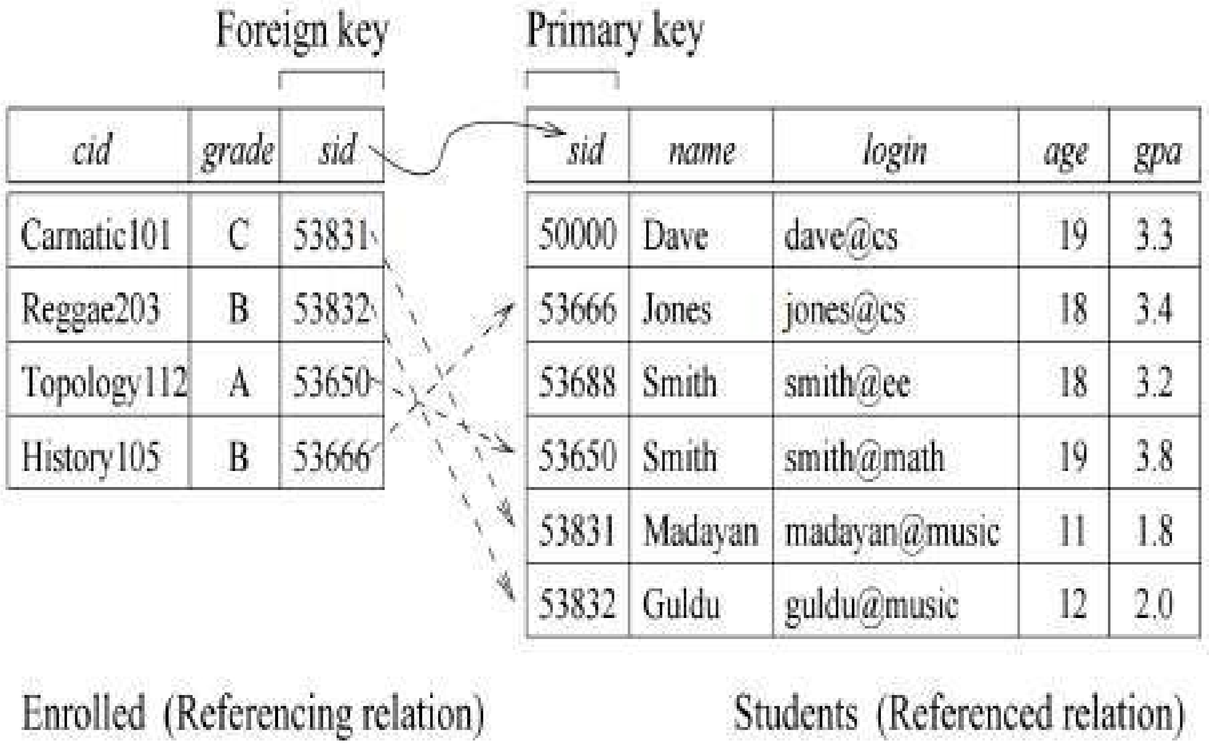
Enrolled (sid: string, cid: string, grade: string)

→To ensure that only bona fide students can enroll in courses, any value that appears in the sid field of an instance of the Enrolled relation should also appear in the sid field of some tuple in the Students relation. The sid field of Enrolled is called a foreign key and refers to Students. The **foreign key** in the **referencing relation** (Enrolled) must match the primary key of the **referenced relation** (Students), i.e., it must have the same number of columns and compatible data types, although the column names can be different.

→This constraint is illustrated in **Figure 3.4.** As the figure shows, there may well be some students who are not referenced from Enrolled (e.g., the student with sid =50000).

→However, every **sid** value that appears in the instance of the Enrolled table appears in the primary key column of a row in the Students table.

→A **FOREIGN KEY constraint** does not have to be linked only to a PRIMARY KEY constraint in another table; it can also be defined to reference the columns of a **UNIQUE constraint** in another table. A FOREIGN KEY constraint can contain null values; however, if any column of a composite FOREIGN.


Foreign key
Primary key
Enrolled (Referencing relation)
Students (Referenced relation)

cid	*grade*	*sid*
Carnatic101	C	53831
Reggae203	B	53832
Topology112	A	53650
History105	B	53666

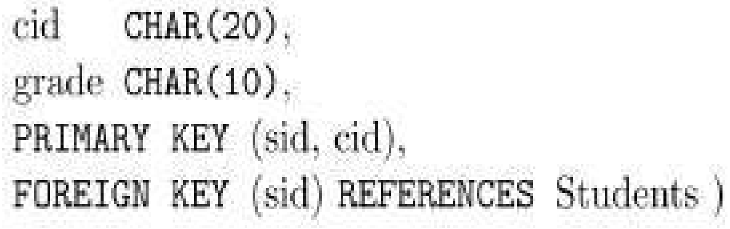
sid	*name*	*login*	*age*	*gpa*
50000	Dave	dave@cs	19	3.3
53666	Jones	jones@cs	18	3.4
53688	Smith	smith@ee	18	3.2
53650	Smith	smith@math	19	3.8
53831	Madayan	madayan@music	11	1.8
53832	Guldu	guldu@music	12	2.0

Figure 3.4 Referential Integrity

Specifying Foreign Key Constraints in SQL:

Let us define Enrolled(*sid:* `string`, *cid:* `string`, *grade:* `string`):

```
CREATE TABLE Enrolled ( sid    CHAR(20),
```



```
cid    CHAR(20),
grade CHAR(10),
PRIMARY KEY (sid, cid),
FOREIGN KEY (sid) REFERENCES Students )
```

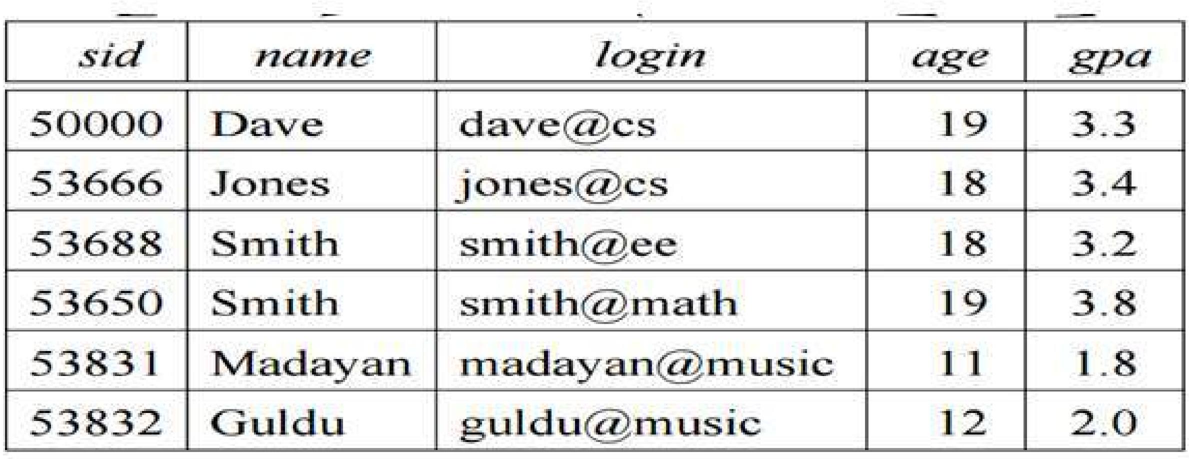
The statement **FOREIGN KEY (sid) REFERENCES Students** means that the foreign key sid uses primary id sid of employee relation as a reference. Every tuple with sid must match a tuple in Students relation.

The **foreign key constraint** states that every **sid** value in Enrolled must also appear in Students, that is, sid in Enrolled is a **foreign key** referencing Students.

3). General Constraints:

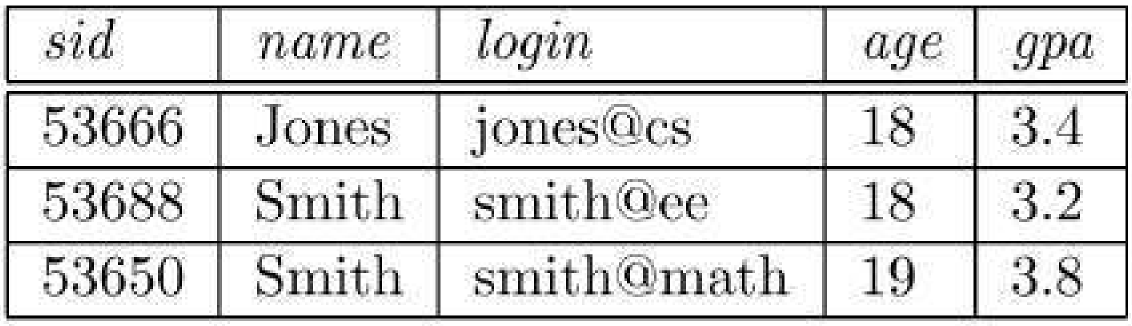
Domain, primary key, and foreign key constraints are considered to be a fundamental part of the relational data model and are given special attention in most commercial systems. Sometimes, however, it is necessary to specify more general constraints.

E**xample**: we may require that student ages be within a certain range of values; given such an IC specification, the DBMS will reject inserts and updates that violate the constraint. This is very useful in preventing data entry errors. If we specify that all students must be at least 16 years old, then age are valid cases i.e., legal instance. Rest of all the others having lesser than 16 years are called as invalid cases i.e., illegal instance. Instance of Students shown in **Figure 3.1** is illegal because two students are underage. If we disallow the insertion of these two tuples, we have a legal instance, as shown in **Figure 3.5.**



sid	*name*	*login*	*age*	*gpa*
50000	Dave	dave@cs	19	3.3
53666	Jones	jones@cs	18	3.4
53688	Smith	smith@ee	18	3.2
53650	Smith	smith@math	19	3.8
53831	Madayan	madayan@music	11	1.8
53832	Guldu	guldu@music	12	2.0

Figure 3.1 An Instance $S1$ of the Students Relation



sid	*name*	*login*	*age*	*gpa*
53666	Jones	jones@cs	18	3.4
53688	Smith	smith@ee	18	3.2
53650	Smith	smith@math	19	3.8

Figure 3.5 An Instance *S*2 of the Students Relation

→The IC that students must be older than 16, is known as an extended domain constraint, because we are restricting age values more stringently (strictly), than by simply using a standard domain such as integer.

→In general, constraints domain, primary and foreign key constraints can also specify the maximum limit.
Example: we require a student whose age is greater than 18 must have a gpa greater than 3.

→There are two types of general constraints. They are

<u>1. Table Constraints:</u> These are applied on a particular table and are checked every table whenever that specific table is updated.
<u>2. Assertions:</u> These assertions are applied on collection of tables and are checked every time whenever theses tables are applied.

<u>3. ENFORCING INTEGRITY CONSTRAINTS:</u>

→**Integrity Constraints(IC)** are the rules that when applied on relations restricts the insertion of incorrect data and also helps to prevent deletion and updating of consistent data that may lead to loss of data integrity. And, therefore one should be very careful when applying integrity constraints on relations.

The operations such as insertion, deletion and updating must be discarded if they are found to violate integrity constraints. This section provides a brief on different violations of ICs and also the solutions to handle these violations.

Consider the instance S1 of Students shown in **Figure 3.1**. The following insertion violates the primary key constraint because there is already a tuple with the **sid 53688**, and it will be rejected by the DBMS:

```
INSERT
INTO    Students  (sid, name, login, age, gpa)
VALUES  (53688, 'Mike', 'mike@ee', 17, 3.4)
```

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

```
INSERT
INTO    Students  (sid, name, login, age, gpa)
VALUES  (null, 'Mike', 'mike@ee', 17, 3.4)
```

→Deletion does not cause a violation of domain, primary key or unique constraints.
→However, an update can cause violations, similar to an insertion:

```
UPDATE  Students S
SET     S.sid = 50000
WHERE   S.sid = 53688
```

This update violates the primary key constraint because there is already a tuple with **sid 50000.**

→In addition to the instance S1 of Students, consider the instance of Enrolled shown in Figure 3.4. Deletions of Enrolled tuples do not violate referential integrity, but insertions of Enrolled tuples could. The following insertion is illegal because there is no student with **sid 51111:**

```
INSERT
INTO    Enrolled  (cid, grade, sid)
VALUES  ('Hindi101', 'B', 51111)
```

EXAMPLE:

```
CREATE TABLE Enrolled ( sid    CHAR(20),
                        cid    CHAR(20),
                        grade  CHAR(10),
                        PRIMARY KEY (sid, cid),
                        FOREIGN KEY (sid) REFERENCES Students
                                    ON DELETE CASCADE
                                    ON UPDATE NO ACTION )
```

→This example explains the options when delete or update operation are performed. These options are included as a part of foreign key declaration. No action is the default option which means both update and delete operations are rejected.

1) On Delete Cascade: Means when a row is deleted from Students relation, then all the rows referred to this deleted row in Enrolled relation must also be deleted.

2) On Update Cascade: Means when updations are carried in Students relation for the primary key attribute then all these updations must also be carried out in Enrolled.

3) On Delete Set Default: Means when a row is deleted in Students, then that row in Enrolled relation can be set to same default value.

4) On Delete Set Null: Means on deleting the row in Students the same row can be assigned a NULL value in Enrolled relation.

NOTE: SQL even provides the facility to delay the applications of constraints on relation and also immediate application of constraints. This is possible with these two additional constraints,

1) **Deferred mode**
2) **Immediate mode**

The syntax for this constraint is,

SET CONSTRAINT Constraint-name DEFERRED

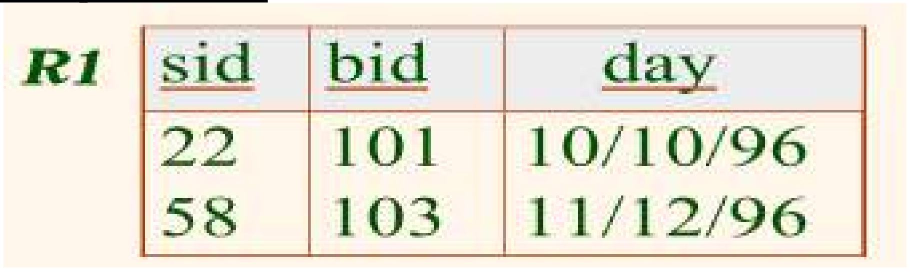
SET CONSTRAINT Constraint-name IMMEDIATE

→Usually, constraints are checked at the end of SQL statements and if the constraints are violated then the statements are rejected. But with differed constraint, constraint checks are postponed and are checked at the time of commit.

<u>RELATIONAL ALGEBRA:</u>

→**Relational algebra** is a procedural query language, which takes instances of relations as input and yields instances of relations as output. It uses operators to perform queries. An operator can be either **unary** or **binary**. They accept relations as their input and yield relations as their output. Relational algebra is performed recursively on a relation and intermediate results are also considered relations.

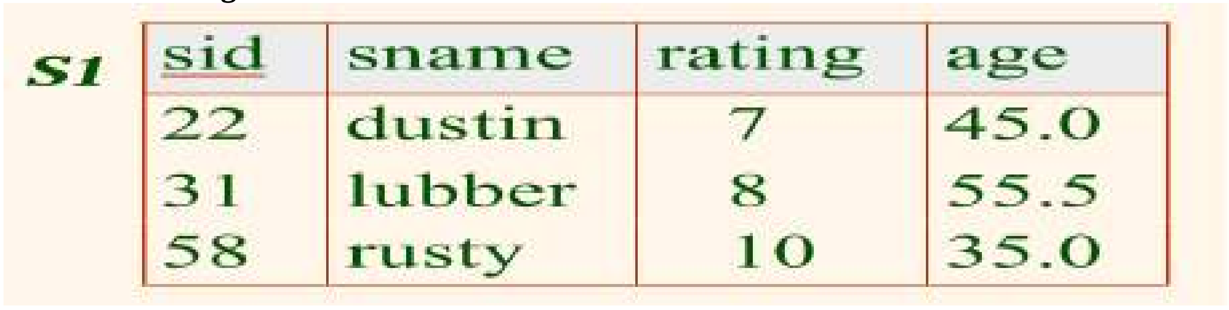
<u>Example schemas:</u>
Sailors (sid: integer, sname: string, rating: integer, age: real)
Boats (bid: integer, bname: string, color: string)
Reserves (sid: integer, bid: integer, day: date)
<u>Example Instances:</u>



R1

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

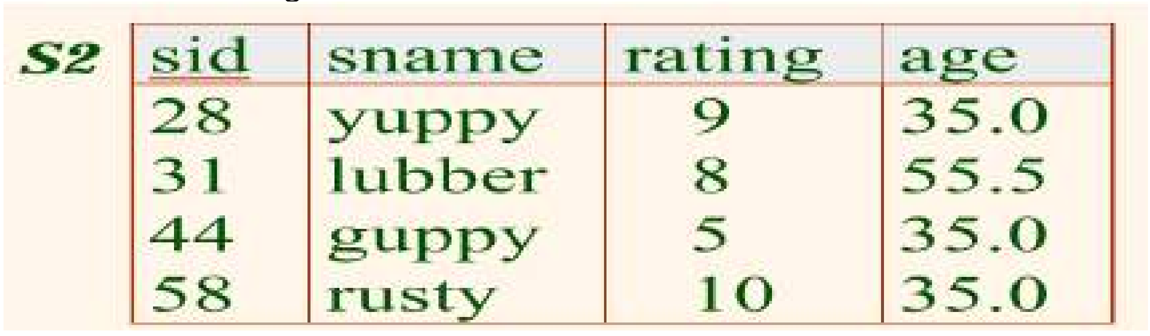
Figure 4.1 Instance S1 of Sailors



S1

sid	sname	rating	age
22	dustin	7	45.0
31	lubber	8	55.5
58	rusty	10	35.0

Figure 4.2 Instance S2 of Sailors



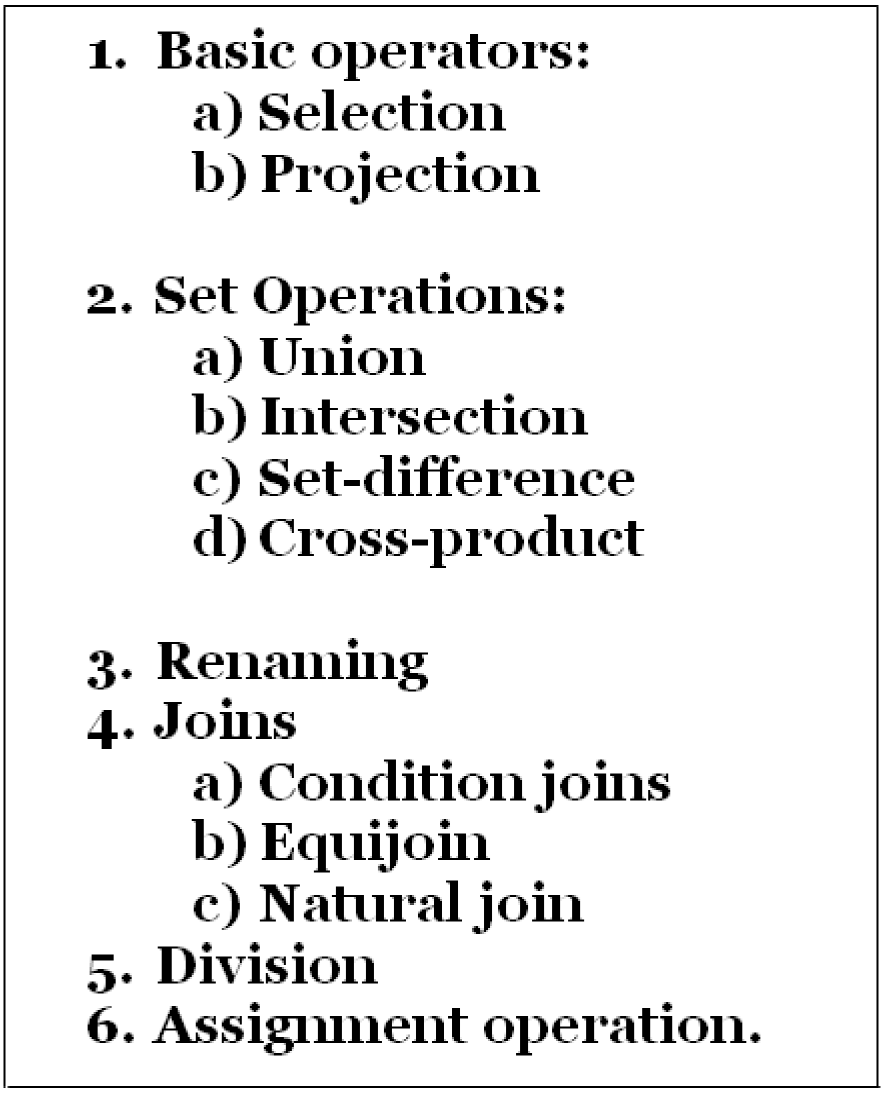
S2

sid	sname	rating	age
28	yuppy	9	35.0
31	lubber	8	55.5
44	guppy	5	35.0
58	rusty	10	35.0

Figure 4.3 Instance R1 of Reserves

→The "Sailors" and "Reserves" relations are our examples. We'll use positional or named field notation, assume that names of fields in query results are `inherited' from names of fields in query input relations.

→**<u>The fundamental operations of relational algebra are:</u>**

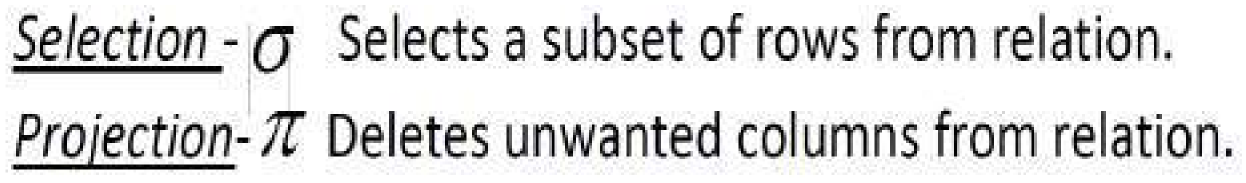

1. Basic operators:
   a) Selection
   b) Projection

2. Set Operations:
   a) Union
   b) Intersection
   c) Set-difference
   d) Cross-product

3. Renaming
4. Joins
   a) Condition joins
   b) Equijoin
   c) Natural join
5. Division
6. Assignment operation.

1. Selection and Projection:

→Relational algebra includes operators to select rows from a relation (σ) and to project columns (π). These operations allow us to manipulate data in a single relation.


Selection - σ Selects a subset of rows from relation.

Projection- π Deletes unwanted columns from relation.

SELECTION (σ):

The **selection operation** is a unary operation. This is used to find horizontal subset of relation or tuples of relation.
It selects tuples that satisfy the given predicate from a relation. It is denoted by **sigma** (**σ**).

Notation − $\sigma_p(r)$

Where **σ** stands for selection predicate and **r** stands for relation. *p* is prepositional logic formula which may use connectors like **and, or,** and **not**. These terms may use relational operators like − =, ≠, ≥, < , >, ≤.

Example: If you want all the **Sailors having rating more than 8** from instance S2 of Sailors. The query is,

$$\sigma_{rating > 8}(S2)$$

The result is shown in **Figure 4.4**

sid	*sname*	*rating*	*age*
28	yuppy	9	35.0
58	Rusty	10	35.0

Figure 4.4 $\sigma_{rating>8}(S2)$

<u>PROJECTION (π):</u>

The **projection operation** is a unary operation which applies only on a single relation at a time. This is used to select vertical subset of relation (i.e., columns of table)
It projects column(s) that satisfy a given predicate. It is denoted by **pi (π).**

Notation – $\prod_{A1, A2, An}$ **(r)**

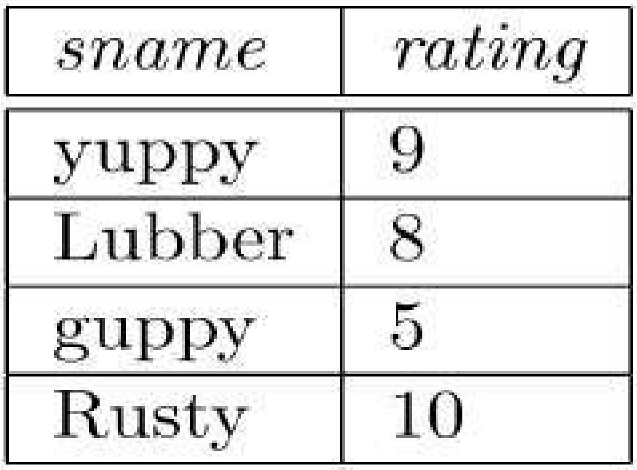
Where A_1, A_2 , A_n are attribute names of relation **r**.

Duplicate rows are automatically eliminated, as relation is a set.

Example: If you can find out **all sailors names and ratings** from instance S2 of Sailors. The query is,

$$\pi_{sname,rating}(S2)$$

The result is shown in **Figure 4.5**



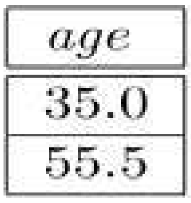
sname	*rating*
yuppy	9
Lubber	8
guppy	5
Rusty	10

Figure 4.5 $\pi_{sname,rating}(S2)$

→ Suppose that we wanted to find out **only the ages of sailors**. The expression

$$\pi_{age}(S2)$$

evaluates to the relation shown in **Figure 4.6.**



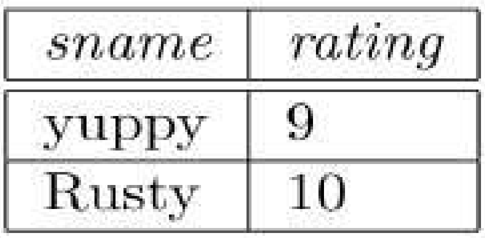
age
35.0
55.5

Figure 4.6 $\pi_{age}(S2)$

→ For example, we can compute the **names and ratings of highly rated sailors** by combining two of the preceding queries. The expression

$$\pi_{sname,rating}(\sigma_{rating>8}(S2))$$

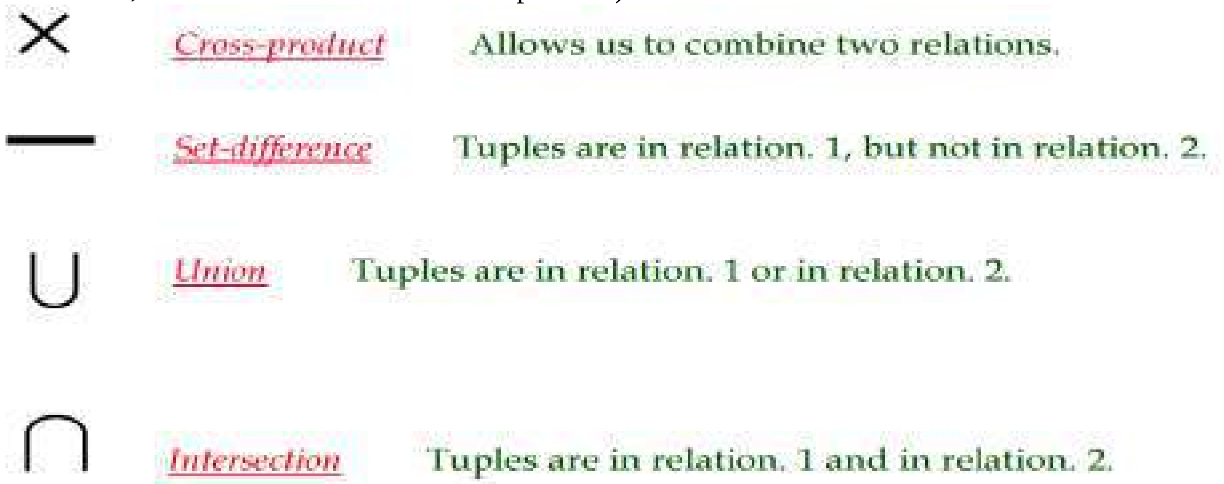
produces the result shown in **Figure 4.7.** It is obtained by applying the selection to S2 (to get the relation shown in Figure 4.4) and then applying the projection.



sname	*rating*
yuppy	9
Rusty	10

Figure 4.7 $\pi_{sname,rating}(\sigma_{rating>8}(S2))$

<u>2. SET OPERATIONS:</u>
The relational algebraic operations can be divided into basic set oriented operations (Union, Intersection, Set difference and Cartesian product).



$\times$	*Cross-product*	Allows us to combine two relations.
$-$	*Set-difference*	Tuples are in relation. 1, but not in relation. 2.
$\cup$	*Union*	Tuples are in relation. 1 or in relation. 2.
$\cap$	*Intersection*	Tuples are in relation. 1 and in relation. 2.

<u>The UNION (∪) Operation:</u>
→R∪S returns a relation instance containing all tuples that occur in either relation instance R or relation instance S (or both). R and S must be union-compatible, and the schema of the result is defined to be identical to the schema of R.

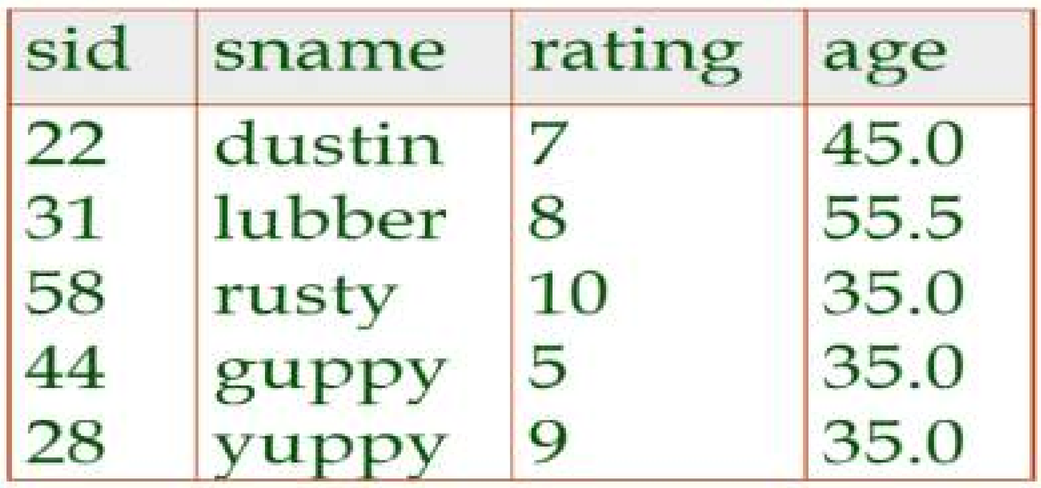
Notation − R U S

→<u>Two relation instances are said to be **union-compatible** if the following conditions hold:</u>

- They have the same number of the fields, and
- Corresponding fields, taken in order from left to right, have the same domains.
- Duplicate tuples are automatically eliminated.

Note that field names are not used in defining union-compatibility. For convenience, we will assume that the fields of R ∪ S inherit names from R, if the fields of R have names.

→The union of S1 and S2 is shown in **Figure 4.8**. Fields are listed in order; field names are also inherited from S1. S2 has the same field names, of course, since it is also an instance of Sailors. In general, fields of S2 may have different names; recall that we require only domains to match. Note that the result is a set of tuples. Tuples that appear in both S1 and S2 appear only once in S1 ∪ S2. Also, S1 ∪ R1 is not a valid operation because the two relations are not union-compatible.



sid	sname	rating	age
22	dustin	7	45.0
31	lubber	8	55.5
58	rusty	10	35.0
44	guppy	5	35.0
28	yuppy	9	35.0

Figure 4.8 S1 ∪ S2

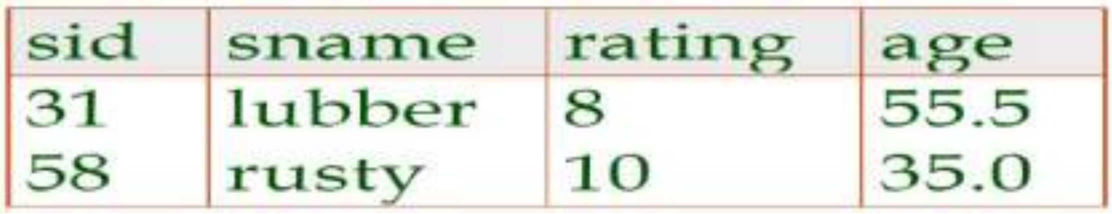
The INTERSECTION (∩) Operation:

→R∩S returns a relation instance containing all tuples that occur in both R and S. The relations R and S must be **union-compatible**, and the schema of the result is defined to be identical to the schema of R.

Notation − **R ∩ S**

If the relations contain nothing as common then the result will be an empty relation. Rules of set union operations are also applicable here.

→The intersection of S1 and S2 is shown in **Figure 4.9.**



sid	sname	rating	age
31	lubber	8	55.5
58	rusty	10	35.0

Figure 4.9 S1 ∩ S2

The SET-DIFFERENCE (−) Operation:

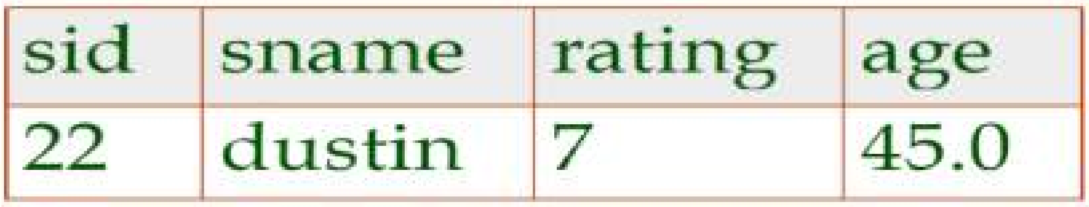
→**R−S** returns a relation instance containing all tuples that occur in R but not in S. The relations R and S must be union-compatible, and the schema of the result is defined to be identical to the schema of R.

The result of set difference operation is tuples, which are present in one relation but are not in the second relation. It removes the common tuples of two relations and produces a new relation having rest of the tuples of first relation.

Notation − **R − S**

→ It finds all the tuples that are present in **R** but not in **S**.

→The set-difference S1 − S2 is shown in **Figure 4.10.**



sid	sname	rating	age
22	dustin	7	45.0

Figure 4.10 S1 - S2

<u>The CROSS-PRODUCT (×) Operation:</u>

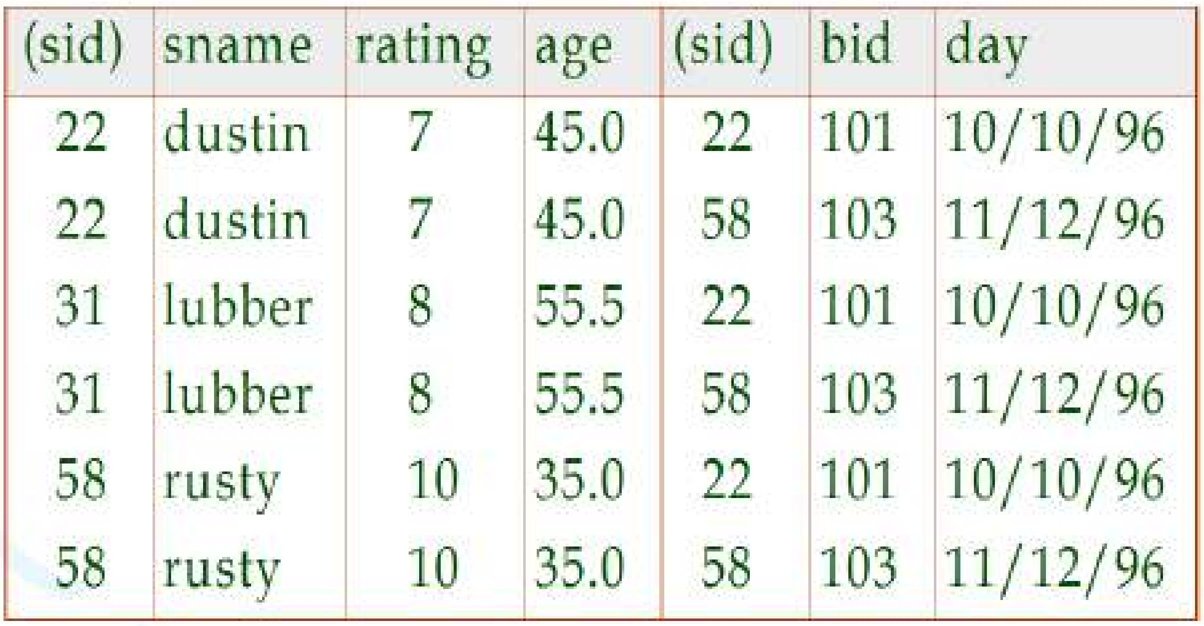
→**R ×S** returns a relation instance whose schema contains all the fields of R (in the same order as they appear in R) followed by all the fields of S (in the same order as they appear in S). The result of R × S contains one tuple hr, si (the concatenation of tuples r and s) for each pair of tuples r ∈ R, s ∈ S. The cross-product operation is sometimes called **Cartesian product**.

→We will use the convention that the fields of R × S inherit names from the corresponding fields of R and S. It is possible for both R and S to contain one or more fields having the same name; this situation creates a naming conflict. The corresponding fields in R × S are unnamed and are referred to solely by position.

→ It combines information of two different relations into one.

Notation − R X S

→The result of the cross-product **S1 × R1** is shown in **Figure 4.11.** Because R1 and S1 both have a field named sid, by our convention on field names, the corresponding two fields in S1 × R1 are unnamed, and referred to solely by the position in which they appear in **Figure 4.11**. The fields in S1 × R1 have the same domains as the corresponding fields in R1 and S1. In **Figure 4.11 sid** is listed in parentheses to emphasize that it is not an inherited field name; only the corresponding domain is inherited.



(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

Figure 4.11 S1 × R1

3. Renaming (ρ): (********)

The rename (*ρ)* operation is a unary operation which is used to give names to relational algebra expressions.

The results of relational algebra are also relations but without any name. The rename operation allows us to rename the output relation. 'Rename' operation is denoted with small Greek letter **rho** *ρ*.

Suppose, you want to find Cartesian product of a relation with itself then by using rename operator we give an alias name to that relation. Now, you can easily multiply that relation with its alias. It is helpful in removing ambiguity.

Notation − ρ_{x} (E)

→Where the result of expression **E** is saved with name of **x**.

For example, the expression **ρ(C (1 → sid1, 5 → sid2), S1 × R1)** returns a relation that contains the tuples shown in **Figure 4.11** and has the following schema:

C (sid1: integer, **sname**: string, **rating**: integer, **age**: real, **sid2**: integer, **bid**: integer, **day**: dates).

4. Joins: (********)

The join operation is one of the most useful operations in relational algebra and is the most commonly used way to combine information from two or more relations.

The join operation denoted by "join" or "⋈", is a relational algebra operation, which is used to combine (join) two relations like Cartesian-product but finally removes duplicate attributes (same column to only one column) and makes the operations (selection, projection etc.,) very simple. In simple words, we can say that join connects relations on columns containing comparable information.

There are three types of joins. Namely, they are

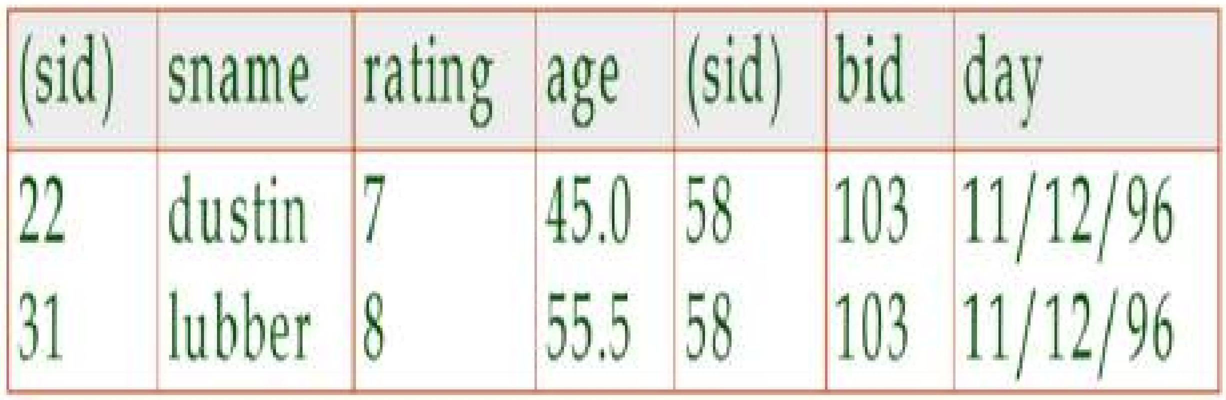
1. **Condition Joins**
2. **Equi Join** and
3. **Natural join**.

<u>1. Condition Joins:</u>
→The most general version of the join operation accepts a join condition c and a pair of relation instances as arguments, and returns a relation instance. The join condition is identical to a selection condition in form. The operation is defined as follows:

$$R \bowtie_{c} S = \sigma_{c}(R \times S)$$

Thus ⋈ *is defined to be* a cross-product followed by a selection. Note that the condition c can (and typically does) refer to attributes of both R and S. The reference to an attribute of a relation, say R, can be by position (of the form R.i) or by name (of the form R.name).

→**<u>Example:</u>** the result of $S1 \bowtie_{S1.sid < R1.sid} R1$ is shown in **Figure 4.12**. Because **sid** appears in both S1 and R1, the corresponding fields in the result of the cross-product S1 × R1 (and therefore in the result of $S1 \bowtie_{S1.sid < R1.sid} R1$ are unnamed. Domains are inherited from the corresponding fields of S1 and R1.



(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

Figure 4.12 $S1 \bowtie_{S1.sid < R1.sid} R1$

- ***Result schema*** same as that of cross-product.
- Fewer tuples than cross-product, might be able to compute more efficiently
- Sometimes called a ***theta-join.***

2. Equijoin:

→ It is a special case of condition join where the condition *c* contains only ***equalities.***

→**Equijoin** is same as condition join, the only difference is that, equijoin uses equity '=' operator to join the two relations.

The schema of the result of an equijoin contains the fields of R (with the same names and domains as in R) followed by the fields of S that do not appear in the join conditions. If this set of fields in the result relation includes two fields that inherit the same name from R and S, they are unnamed in the result relation.

We illustrate $S1 \bowtie_{R.sid=S.sid} R1$ in **Figure 4.13**. Notice that only one field called sid appears in the result.

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

Figure 4.13 $S1 \bowtie_{R.sid=S.sid} R1$

→**Result schema** similar to cross-product, but only one copy of fields for which equality is specified.

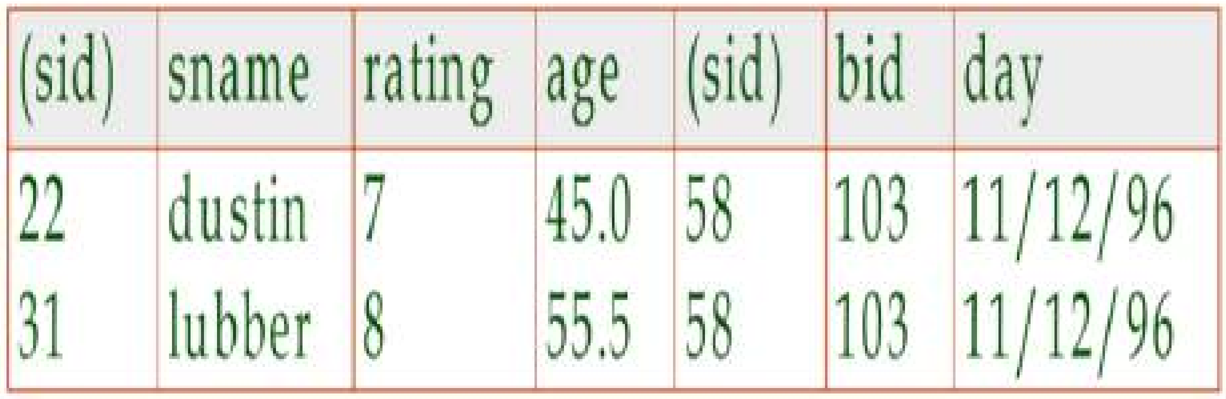
3. Natural Join:

→**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.

→Special case of the join operation R ⋈ S is an equijoin in which equalities are specified on all fields having the same name in R and S. In this case, we can simply omit the join condition; the default is that the join condition is a collection of equalities on all common fields. This special case a **natural join**, and with this result is guaranteed not to have two fields with the same name.

The equijoin expression $S1 \bowtie_{R.sid=S.sid} R1$ is actually a natural join and can simply be denoted as S1⋈ R1, since the only common field is sid. If the two relations have no attributes in common, S1 ⋈ R1 is simply the cross-product.



(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

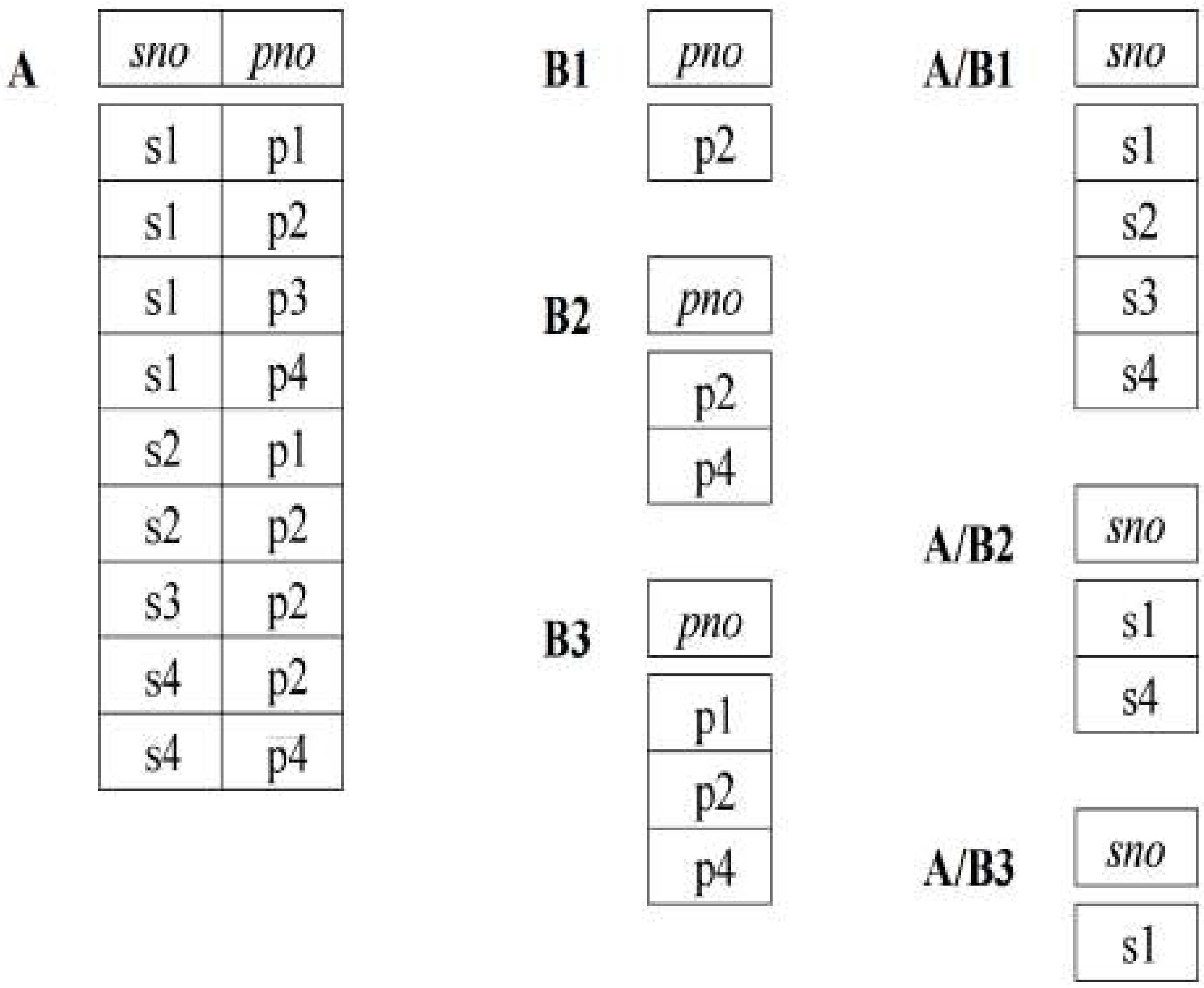
Figure. S1⋈ R1

5. Division:

→The **division operator** is useful for expressing certain kinds of queries that include the phrase "for all'. It is denoted by **(/)**. It is alike the inverse of Cartesian product.

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, the x value is in the result of A/B.
An analogy with integer division may also help to understand division. For integers A and B, A/B is the largest integer Q such that $Q * B \leq A$. For relation instances A and B, A/B is the largest relation instance Q such that $Q \times B \subseteq A$.
Division is illustrated in **Figure 4.14.**



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

Figure 4.14 Examples Illustrating Division

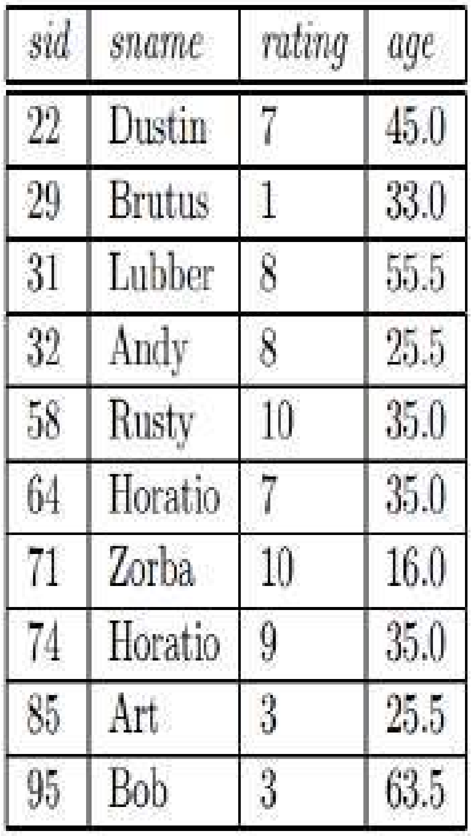
<u>Expressing A/B Using Basic Operators:</u>

- Division is not essential op; just a useful shorthand.
  - (Also true of joins, but joins are so common that systems implement joins specially.)
- *Idea*: For *A/B*, compute all *x* values that are not `disqualified' by some *y* value in *B*.
  - *x* value is *disqualified* if by attaching *y* value from *B*, we obtain an *xy* tuple that is not in *A*.

Disqualified x values $\pi_x((\pi_x(A) \times B) - A)$

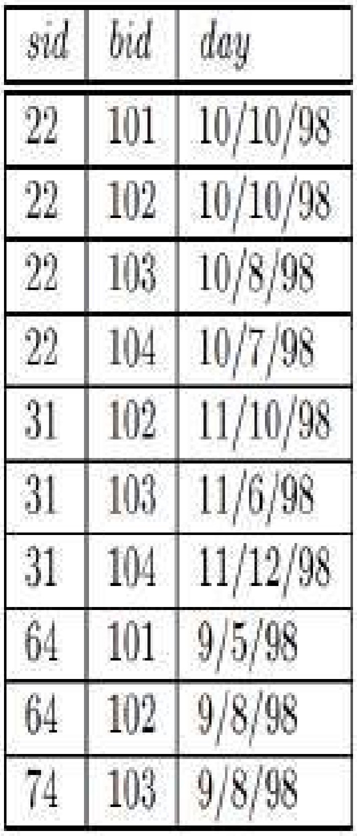
A/B: $\pi_x(A)$ − all disqualified tuples

<u>Examples of Relational Algebra Queries:</u>



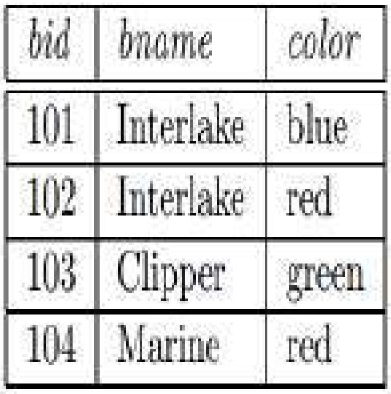
sid	*sname*	*rating*	*age*
22	Dustin	7	45.0
29	Brutus	1	33.0
31	Lubber	8	55.5
32	Andy	8	25.5
58	Rusty	10	35.0
64	Horatio	7	35.0
71	Zorba	10	16.0
74	Horatio	9	35.0
85	Art	3	25.5
95	Bob	3	63.5

Figure 4.15 An Instance *S3* 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

Figure 4.16 An Instance *R2* of Reserves



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

Figure 4.17 An Instance *B1* of Boats

(Q1) Find names of sailors who have reserved boat 103.

Solution 1:

→This query can be written as follows,

$$\pi_{sname}((\sigma_{bid=103} Reserves) \bowtie Sailors)$$

→First, we compute the set of tuples in Reserves with bid = 103 and then take the natural join of this set with Sailors. This expression can be evaluated on instances of Reserves and Sailors. Evaluated on the instances R2 and S3, it yields a relation that contains just one field, called sname, and three tuples **<Dustin>**, **<Horatio>,** and **<Lubber>.** (Observe that there are two sailors called Horatio, and only one of them has reserved a red boat.)

→We can break this query into smaller pieces using the **renaming operator ρ**:

Solution 2:

$$\rho\ (Temp1,\ \sigma_{bid=103}\ Reserves)$$

$$\rho\ (Temp2,\ Temp1 \bowtie Sailors)$$

$$\pi_{sname}(Temp2)$$

→Because we are only using ρ to give names to intermediate relations, the renaming list is optional and is omitted. **Temp1** denotes an intermediate relation that identifies reservations of boat 103. **Temp2** is another intermediate relation, and it denotes sailors who have made a reservation in the set Temp1. The instances of these relations when evaluating this query on the instances R2 and S3 are illustrated in **Figures 4.18 and 4.19**. Finally, we extract the sname column from Temp2.

sid	bid	day
22	103	10/8/98
31	103	11/6/98
74	103	9/8/98

Figure 4.18 Instance of *Temp1*

sid	sname	rating	age	bid	day
22	Dustin	7	45.0	103	10/8/98
31	Lubber	8	55.5	103	11/6/98
74	Horatio	9	35.0	103	9/8/98

Figure 4.19 Instance of *Temp2*

Solution 3:

$$\pi_{sname}(\sigma_{bid=103}(Reserves \bowtie Sailors))$$

(Q2) Find the names of sailors who have reserved a red boat.

→This query can be written as follows,

$$\pi_{sname}((\sigma_{color='red'} Boats) \bowtie Reserves \bowtie Sailors)$$

→This query involves a series of two joins. First we choose (tuples describing) red boats. Then we join this set with Reserves (natural join, with equality specified on the bid column) to identify reservations of red boats. Next we join the resulting intermediate relation with Sailors (natural join, with equality specified on the sid column) to retrieve the names of sailors who have made reservations of red boats. Finally, we project the sailors' names. The answer, when evaluated on the instances B1, R2 and S3, contains the names Dustin, Horatio, and Lubber.

→An equivalent expression is:

$$\pi_{sname}(\pi_{sid}((\pi_{bid}\sigma_{color='red'} Boats) \bowtie Reserves) \bowtie Sailors)$$

(Q3) Find the colors of boats reserved by Lubber.

$$\pi_{color}((\sigma_{sname='Lubber'} Sailors) \bowtie Reserves \bowtie Boats)$$

→This query is very similar to the query we used to compute sailors who reserved red boats. On instances B1, R2, and S3, the query will return the colors green and red.

(Q4) Find the names of sailors who have reserved at least one boat.

$$\pi_{sname}(Sailors \bowtie Reserves)$$

→The join of Sailors and Reserves creates an intermediate relation in which tuples consist of a Sailors tuple 'attached to' a Reserves tuple. A Sailors tuple appears in (some tuple of) this intermediate relation only if at least one Reserves tuple has the same sid value, that is, the sailor has made some reservation. The answer, when evaluated on the **instances B1, R2 and S3**, contains the three tuples **<Dustin>, <Horatio>,** and **<Lubber>.** Even though there are two sailors called Horatio who have reserved a boat, the answer contains only one copy of the tuple **<Horatio>,** because the answer is a relation, i.e., a set of tuples, without any duplicates.

(Q5) Find the names of sailors who have reserved a red or a green boat.

$$\rho(Tempboats, (\sigma_{color='red'} Boats) \cup (\sigma_{color='green'} Boats))$$
$$\pi_{sname}(Tempboats \bowtie Reserves \bowtie Sailors)$$

→We identify the set of all boats that are either red or green (**Tempboats**, which contains boats with the bids 102, 103, and 104 on instances B1, R2, and S3). Then we join with Reserves to identify sids of sailors who have reserved one of these boats; this gives us sids 22, 31, 64, and 74 over our example instances. Finally, we join (an intermediate relation containing this set of sids) with Sailors to find the names of Sailors with these sids. This gives us the names **Dustin, Horatio,** and **Lubber** on the instances B1, R2, and S3.

→Another equivalent definition is the following:

$$\rho(Tempboats, (\sigma_{color='red' \vee color='green'} Boats))$$
$$\pi_{sname}(Tempboats \bowtie Reserves \bowtie Sailors)$$

(Q6) Find the names of sailors who have reserved a red and a green boat.

It is tempting to try to do this by simply replacing ∪ by ∩ in the definition of Tempboats:

$$\rho(Tempboats2, (\sigma_{color='red'}Boats) \cap (\sigma_{color='green'}Boats))$$
$$\pi_{sname}(Tempboats2 \bowtie Reserves \bowtie Sailors)$$

→However, this solution is **incorrect**—it instead tries to compute sailors who have reserved a boat that is both red and green.

→The **correct approach** is to find **sailors who have reserved a red boat, then sailors who have reserved a green boat**, and then take the intersection of these two sets:

$$\rho(Tempred, \pi_{sid}((\sigma_{color='red'}Boats) \bowtie Reserves))$$
$$\rho(Tempgreen, \pi_{sid}((\sigma_{color='green'}Boats) \bowtie Reserves))$$
$$\pi_{sname}((Tempred \cap Tempgreen) \bowtie Sailors)$$

→The two temporary relations compute the **sids** of sailors, and their intersection identifies sailors who have reserved both red and green boats. On instances B1, R2, and S3, the **sids** of sailors who have reserved a red boat are 22, 31, and 64. The **sids** of sailors who have reserved a green boat are 22, 31, and 74. Thus, sailors 22 and 31 have reserved both a red boat and a green boat; their names are Dustin and Lubber.
→This formulation of Query Q6 can easily be adapted to find sailors who have reserved red or green boats (Query Q5); just replace ∩ by ∪:

$$\rho(Tempred, \pi_{sid}((\sigma_{color='red'}Boats) \bowtie Reserves))$$
$$\rho(Tempgreen, \pi_{sid}((\sigma_{color='green'}Boats) \bowtie Reserves))$$
$$\pi_{sname}((Tempred \cup Tempgreen) \bowtie Sailors)$$

(Q7) Find the names of sailors who have reserved at least two boats.

$$\rho(Reservations, \pi_{sid,sname,bid}(Sailors \bowtie Reserves))$$
$$\rho(Reservationpairs(1 \rightarrow sid1, 2 \rightarrow sname1, 3 \rightarrow bid1, 4 \rightarrow sid2,$$
$$5 \rightarrow sname2, 6 \rightarrow bid2), Reservations \times Reservations)$$
$$\pi_{sname1}\sigma_{(sid1=sid2)\wedge(bid1\neq bid2)}Reservationpairs$$

→First we compute tuples of the form < **sid, sname, bid**>, where sailor sid has made a reservation for boat bid; this set of tuples is the temporary relation Reservations. Next we find all pairs of Reservations tuples where the same sailor has made both reservations and the boats involved are distinct. Here is the central idea: In order to show that a sailor has reserved two boats, we must find two Reservations tuples involving the same sailor but distinct boats. Over instances B1, R2, and S3, the sailors with **sids** 22, 31, and 64 have each reserved at least two boats. Finally, we project the names of such sailors to obtain the answer, containing the names **Dustin, Horatio,** and **Lubber.**

(Q8) Find the sids of sailors with age over 20 who have not reserved a red boat.

$$\pi_{sid}(\sigma_{age>20}Sailors) -$$
$$\pi_{sid}((\sigma_{color='red'}Boats) \bowtie Reserves \bowtie Sailors)$$

→This query illustrates the use of the set-difference operator. Again, we use the fact that sid is the key for Sailors. We first identify sailors aged over 20 (over instances B1, R2, and S3, sids 22, 29, 31, 32, 58, 64, 74, 85, and 95) and then discard those who have reserved a red boat (sids 22, 31, and 64), to obtain the answer (sids 29, 32, 58, 74, 85, and 95). If we want to compute the names of such sailors, we must first compute their sids (as shown above), and then join with Sailors and project the sname values.

(Q9) Find the names of sailors who have reserved all boats.

→The use of the word all (or every) is a good indication that the division operation might be applicable:

$$\rho(Tempsids, (\pi_{sid,bid}Reserves)/(\pi_{bid}Boats))$$
$$\pi_{sname}(Tempsids \bowtie Sailors)$$

→The intermediate relation Tempsids is defined using division, and computes the set of sids of sailors who have reserved every boat (over instances B1, R2, and S3, this is just sid 22). Notice how we define the two relations that the **division operator (/)** is applied to—the first relation has the schema **(sid, bid)** and the second has the schema **(bid).** Division then returns all sids such that there is a tuple **<sid, bid>** in the first relation for each bid in the second. Joining Tempsids with Sailors is necessary to associate names with the selected sids; for sailor 22, the name is Dustin.

(Q10) Find the names of sailors who have reserved all boats called Interlake.

$$\rho(Tempsids, (\pi_{sid,bid}Reserves)/(\pi_{bid}(\sigma_{bname='Interlake'}Boats)))$$
$$\pi_{sname}(Tempsids \bowtie Sailors)$$

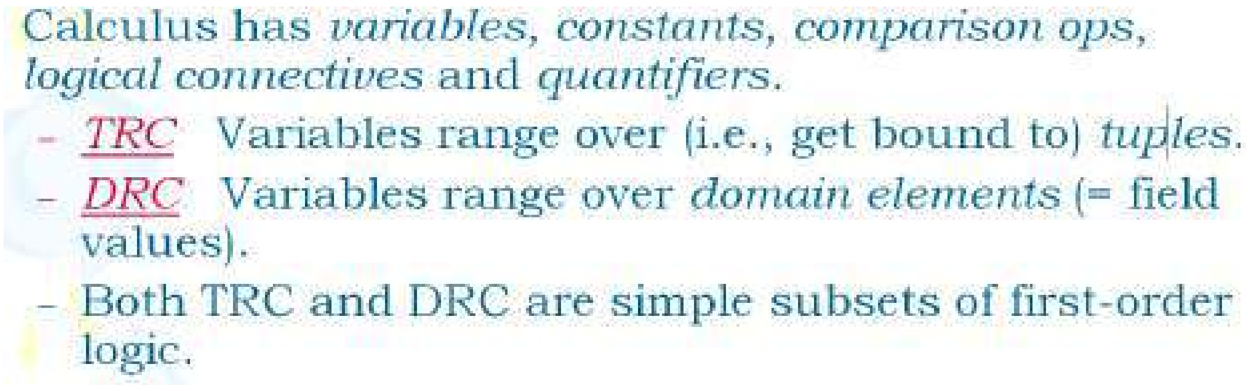
→The only difference with respect to the previous query is that now we apply a selection to Boats, to ensure that we compute only bids of boats named Interlake in defining the second argument to the division operator. Over instances B1, R2, and S3, Tempsids evaluates to sids 22 and 64, and the answer contains their names, Dustin and Horatio.

2. RELATIONAL CALCULUS:

→**Relational calculus** is an alternative to relational algebra. In contrast to the algebra, which is procedural, the calculus is nonprocedural, or declarative, in that it allows us to describe the set of answers without being explicit about how they should be computed.
→**Relational calculus** has had a big influence on the design of commercial query languages such as **SQL** and, especially, **Query-by-Example (QBE).**

→Relational calculus is of two types.

1. **Tuple relational calculus (TRC)**
2. **Domain relational calculus (DRC)**



Calculus has *variables*, *constants*, *comparison ops*, *logical connectives* and *quantifiers*.
- *TRC* Variables range over (i.e., get bound to) *tuples*.
- *DRC* Variables range over *domain elements* (= field values).
- Both TRC and DRC are simple subsets of first-order logic.

1. Tuple Relational Calculus:

→A **tuple variable** is a variable that takes on tuples of a particular relation schema as values. That is, every value assigned to a given tuple variable has the same number and type of fields.
→A **tuple relational calculus** query has the form **{T | p (T)},** where **T** is a tuple variable and **p(T)** denotes a formula that describes T; we will shortly define formulas and queries rigorously. The result of this query is the set of all tuples t for which the formula p(T) evaluates to true with T = t. The language for writing formulas p(T) is thus at the heart of TRC and is essentially a simple subset of first-order logic.

→**As a simple example**, consider the following query:

(Q11) Find all sailors with a rating above 7.

$$\{S \mid S \in Sailors \wedge S.rating > 7\}$$

→When this query is evaluated on an instance of the Sailors relation, the tuple variable S is instantiated successively with each tuple, and the test **S.rating>7** is applied. The answer contains those instances of S that pass this test. On instance S3 of Sailors, the answer contains Sailors tuples with sid 31, 32, 58, 71, and 74.

<u>Syntax of TRC Queries:</u>
→Let **Rel** be a relation name, **R and S** be tuple variables, a an attribute of R, and b an attribute of S. Let op denote an operator in the set {, =, ≤, ≥, 6=}.
→An atomic formula is one of the following:

- $R \in Rel$
- $R.a$ op $S.b$
- $R.a$ op $constant$, or $constant$ op $R.a$

→A **formula** is recursively defined to be one of the following, where p and q are themselves formulas, and p(R) denotes a formula in which the variable R appears:

- any atomic formula
- $\neg p$, $p \wedge q$, $p \vee q$, or $p \Rightarrow q$
- $\exists R(p(R))$, where R is a tuple variable
- $\forall R(p(R))$, where R is a tuple variable

→In the last two clauses above, the quantifiers ∃ and ∀ are said to bind the variable R. A variable is said to be free in a formula or **subformula** (a formula contained in a larger formula) if the (sub) formula does not contain an occurrence of a quantifier that binds it.

→A **TRC query** is defined to be expression of the form {T | p (T)}, where T is the only free variable in the formula p.

<u>Semantics of TRC Queries:</u>
→A query is evaluated on a given instance of the database. Let each free variable in a formula F be bound to a tuple value. For the given assignment of tuples to variables, with respect to the given database instance, F evaluates to (or simply 'is') true if one of the following holds:

1. F is an atomic formula R ∈ Rel, and R is assigned a tuple in the instance of relation Rel.
2. F is a comparison R.a op S.b, R.a op constant, or constant op R.a, and the tuples assigned to R and S have field values R.a and S.b that make the comparison true.
3. F is of the form ¬p, and p is not true; or of the form p ∧ q, and both p and q are true; or of the form p ∨ q, and one of them is true, or of the form p ⇒ q and q is true whenever4 p is true.
4. F is of the form ∃R(p(R)), and there is some assignment of tuples to the free variables in p(R), including the variable R, 5 that makes the formula p(R) true.

5. F is of the form ∀R(p(R)), and there is some assignment of tuples to the free variables in p(R) that makes the formula p(R) true no matter what tuple is assigned to R.

<u>Examples of TRC Queries:</u>

(Q12) Find the names and ages of sailors with a rating above 7.

$$\{P \mid \exists S \in Sailors(S.rating > 7 \wedge P.name = S.sname \wedge P.age = S.age)\}$$

→This query illustrates that P is considered to be a tuple variable with two fields required, name and age. That is the result of this query is a relation with two fields, name and age. The atomic formulas P.name = S.sname and P.age = S.age give values to the fields of an answer tuple P.
→On instances B1, R2, and S3, the answer is the set of tuples **<Lubber, 55.5>, <Andy, 25.5>, <Rusty, 35.0>, <Zorba, 16.0>, and <Horatio, 35.0>.**

(Q13) Find the sailor name, boat id, and reservation date for each reservation.

$$\{P \mid \exists R \in Reserves \ \ \exists S \in Sailors$$
$$(R.sid = S.sid \wedge P.bid = R.bid \wedge P.day = R.day \wedge P.sname = S.sname)\}$$

→For each Reserves tuple, we look for a tuple in Sailors with the same sid. Given a pair of such tuples, we construct an answer tuple P with field's sname, bid, and day by copying the corresponding fields from these two tuples. This query illustrates how we can combine values from different relations in each answer tuple. The answer to this query on instances B1, R2, and S3 is shown in **Figure 4.20.**

sname	*bid*	*day*
Dustin	101	10/10/98
Dustin	102	10/10/98
Dustin	103	10/8/98
Dustin	104	10/7/98
Lubber	102	11/10/98
Lubber	103	11/6/98
Lubber	104	11/12/98
Horatio	101	9/5/98
Horatio	102	9/8/98
Horatio	103	9/8/98

Figure 4.20 Answer to Query Q13

(Q1) Find the names of sailors who have reserved boat 103.

$$\{P \mid \exists S \in Sailors\ \exists R \in Reserves(R.sid = S.sid \wedge R.bid = 103 \wedge P.sname = S.sname)\}$$

→This query can be read as follows: "Retrieve all sailor tuples for which there exists a tuple in Reserves, having the same value in the sid field, and with bid = 103." That is, for each sailor tuple, we look for a tuple in Reserves that shows that this sailor has reserved boat 103. The answer tuple P contains just one field, sname.

(Q2) Find the names of sailors who have reserved a red boat.

$$\{P \mid \exists S \in Sailors\ \ \exists R \in Reserves(R.sid = S.sid \wedge P.sname = S.sname \\ \wedge \exists B \in Boats(B.bid = R.bid \wedge B.color = 'red'))\}$$

This query can be read as follows: "Retrieve all sailor tuples S for which there exist tuples R in Reserves and B in Boats such that S.sid = R.sid, R.bid = B.bid, and B.color ='red'. Another way to write this query, which corresponds more closely to this reading, is as follows:

$$\{P \mid \exists S \in Sailors \;\; \exists R \in Reserves \;\; \exists B \in Boats$$
$$(R.sid = S.sid \wedge B.bid = R.bid \wedge B.color = 'red' \wedge P.sname = S.sname)\}$$

(Q7) Find the names of sailors who have reserved at least two boats.

$$\{P \mid \exists S \in Sailors \;\; \exists R1 \in Reserves \;\; \exists R2 \in Reserves$$
$$(S.sid = R1.sid \wedge R1.sid = R2.sid \wedge R1.bid \neq R2.bid \wedge P.sname = S.sname)\}$$

(Q9) Find the names of sailors who have reserved all boats.

$$\{P \mid \exists S \in Sailors \;\; \forall B \in Boats$$
$$(\exists R \in Reserves(S.sid = R.sid \wedge R.bid = B.bid \wedge P.sname = S.sname))\}$$

This query was expressed using the division operator in relational algebra. Notice how easily it is expressed in the calculus. “Find sailors S such that for all boats B there is a Reserves tuple showing that sailor S has reserved boat B.”

(Q14) Find sailors who have reserved all red boats.

$$\{S \mid S \in Sailors \wedge \forall B \in Boats$$
$$(B.color = 'red' \Rightarrow (\exists R \in Reserves(S.sid = R.sid \wedge R.bid = B.bid)))\}$$

This query can be read as follows: For each candidate (sailor), if a boat is red, the sailor must have reserved it. That is, for a candidate sailor, a boat being red must imply the sailor having reserved it. Observe that since we can return an entire sailor tuple as the answer instead of just the sailor’s name, we have avoided introducing a new free variable (e.g., the variable P in the previous example) to hold the answer values. On instances B1, R2, and S3, the answer contains the Sailors tuples with sids 22 and 31.

We can write this query without using implication, by observing that an expression of the form p ⇒ q is logically equivalent to ¬p ∨ q:

$$\{S \mid S \in Sailors \wedge \forall B \in Boats$$
$$(B.color \neq 'red' \vee (\exists R \in Reserves(S.sid = R.sid \wedge R.bid = B.bid)))\}$$

This query should be read as follows: “Find sailors S such that for all boats B, either the boat is not red or a Reserves tuple shows that sailor S has reserved boat B.”

2. Domain Relational Calculus:

→A **domain variable** is a variable that ranges over the values in the domain of some attribute (e.g., the variable can be assigned an integer if it appears in an attribute whose domain is the set of integers). A DRC query has the form {<x1, x2,...,xn> | p(<x1, x2,...,xn>)}, where each xi is either a domain variable or a constant and p(<x1, x2,...,xn>) denotes a DRC formula whose only free variables are the variables among the xi, 1 ≤ i ≤ n. The result of this query is the set of all tuples <x1, x2,...,xn> for which the formula evaluates to true.

→A DRC formula is defined in a manner that is very similar to the definition of a TRC formula. The main difference is that the variables are now domain variables. Let op denote an operator in the set {, =, ≤, ≥, 6=} and let X and Y be domain variables.
→An atomic formula in DRC is one of the following:

- $\langle x_1, x_2, \ldots, x_n\rangle \in Rel$, where Rel is a relation with n attributes; each x_i, $1 \le i \le n$ is either a variable or a constant.
- X op Y
- X op *constant*, or *constant* op X

→A **formula** is recursively defined to be one of the following, where p and q are themselves formulas, and p(X) denotes a formula in which the variable X appears:

- any atomic formula
- $\neg p$, $p \wedge q$, $p \vee q$, or $p \Rightarrow q$
- $\exists X(p(X))$, where X is a domain variable
- $\forall X(p(X))$, where X is a domain variable

<u>Examples of DRC Queries:</u>

(Q11) Find all sailors with a rating above 7.

$$\{\langle I, N, T, A\rangle \mid \langle I, N, T, A\rangle \in Sailors \wedge T > 7\}$$

This differs from the TRC version in giving each attribute a (variable) name. The condition <I, N, T, A> ∈ Sailors ensures that the domain variables I, N, T, and A are restricted to be fields of the same tuple. In comparison with the TRC query, we can say T > 7 instead of S.rating > 7, but we must specify the tuple <I, N, T, A> in the result, rather than just S.

(Q1) Find the names of sailors who have reserved boat 103.

$$\{\langle N\rangle \mid \exists I,T,A(\langle I,N,T,A\rangle \in Sailors$$
$$\wedge \exists Ir,Br,D(\langle Ir,Br,D\rangle \in Reserves \wedge Ir = I \wedge Br = 103))\}$$

→Only the sname field is retained in the answer and that only N is a free variable. We use the notation ∃Ir, Br, D(...) as a shorthand for ∃Ir(∃Br(∃D(...))). Very often, all the quantified variables appear in a single relation, as in this example. An even more compact notation in this case is ∃hIr, Br, Di ∈ Reserves. With this notation, which we will use henceforth, the above query would be as follows:

$$\{\langle N\rangle \mid \exists I,T,A(\langle I,N,T,A\rangle \in Sailors$$
$$\wedge \exists \langle Ir,Br,D\rangle \in Reserves(Ir = I \wedge Br = 103))\}$$

The comparison with the corresponding TRC formula should now be straightforward. This query can also be written as follows; notice the repetition of variable I and the use of the constant 103:

$$\{\langle N\rangle \mid \exists I,T,A(\langle I,N,T,A\rangle \in Sailors$$
$$\wedge \exists D(\langle I,103,D\rangle \in Reserves))\}$$

(Q2) Find the names of sailors who have reserved a red boat.

$$\{\langle N\rangle \mid \exists I,T,A(\langle I,N,T,A\rangle \in Sailors$$
$$\wedge \exists \langle I,Br,D\rangle \in Reserves \wedge \exists \langle Br,BN,'red'\rangle \in Boats)\}$$

(Q7) Find the names of sailors who have reserved at least two boats.

$$\{\langle N\rangle \mid \exists I,T,A(\langle I,N,T,A\rangle \in Sailors \wedge$$
$$\exists Br1,Br2,D1,D2(\langle I,Br1,D1\rangle \in Reserves \wedge \langle I,Br2,D2\rangle \in Reserves \wedge Br1 \neq Br2$$

(Q9) Find the names of sailors who have reserved all boats.

$$\{\langle N\rangle \mid \exists I,T,A(\langle I,N,T,A\rangle \in Sailors \wedge$$
$$\forall B,BN,C(\neg(\langle B,BN,C\rangle \in Boats) \vee$$
$$(\exists \langle Ir,Br,D\rangle \in Reserves(I = Ir \wedge Br = B))))\}$$

This query can be read as follows: “Find all values of N such that there is some tuple **< I, N, T, A>** in Sailors satisfying the following condition: for every **<B,BN,C>,** either this is not a tuple in Boats or there is some tuple < Ir, Br, D> in Reserves that proves that Sailor I has reserved boat B.” The ∀ quantifier allows the domain variables B, BN, and C to range over all values in their respective attribute domains, and the pattern **‘¬(<B,BN,C> ∈ Boats)∨’** is necessary to restrict attention to those values that appear in tuples of Boats. This pattern is common in DRC formulas, and the notation **∀<B,BN,C> ∈ Boats** can be used as a shorthand instead. This is similar to the notation introduced earlier for ∃. With this notation the query would be written as follows:

$$\{\langle N\rangle \mid \exists I,T,A(\langle I,N,T,A\rangle \in Sailors \wedge \forall\langle B,BN,C\rangle \in Boats$$
$$(\exists\langle Ir,Br,D\rangle \in Reserves(I = Ir \wedge Br = B)))\}$$

(Q14) Find sailors who have reserved all red boats.

$$\{\langle I,N,T,A\rangle \mid \langle I,N,T,A\rangle \in Sailors \wedge \forall\langle B,BN,C\rangle \in Boats$$
$$(C ='red' \Rightarrow \exists\langle Ir,Br,D\rangle \in Reserves(I = Ir \wedge Br = B))\}$$

Chapter-4
4. Queries, Constraints, Triggers

Queries, Constraints, Triggers: The Form of Basic SQL Query, Union, Intersect, and Except, Nested Queries, Aggregate Operators, Null Values, Complex Integrity Constraints in SQL, Triggers and Active Database.

1. Basic SQL, Query:

→**Structured Query Language (SQL)** is the most widely used commercial relational database language. It was originally developed at IBM in the SEQUEL-XRM and System-R projects (1974–1977).

The **SQL language** has several aspects to it:

1. The Data Definition Language (DDL): This subset of SQL supports the creation, deletion, and modification of definitions for tables and views. Integrity constraints can be defined on tables, either when the table is created or later. The DDL also provides commands for specifying access rights or privileges to tables and views. Although the standard does not discuss indexes, commercial implementations also provide commands for creating and deleting indexes.

2. The Data Manipulation Language (DML): This subset of SQL allows users to pose queries and to insert, delete, and modify rows.

3. Embedded and dynamic SQL: Embedded SQL features allow SQL code to be called from a host language such as C or COBOL. Dynamic SQL features allow a query to be constructed (and executed) at run-time.

4. Triggers: The new SQL:1999 standard includes support for triggers, which are actions executed by the DBMS whenever changes to the database meet conditions specified in the trigger.

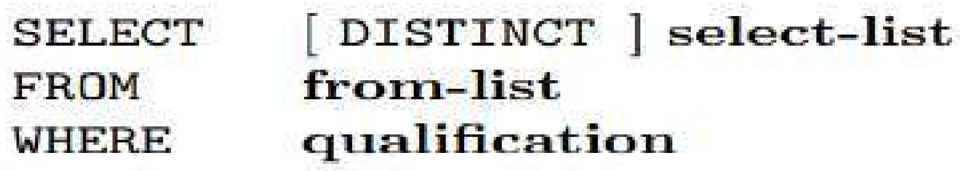
5. Security: SQL provides mechanisms to control users' access to data objects such as tables and views.

6. Transaction management: Various commands allow a user to explicitly control aspects of how a transaction is to be executed.

7. Client-server execution and remote database access: These commands control how a client application program can connect to an SQL database server, or access data from a database over a network.

2. THE FORM OF A BASIC SQL QUERY:

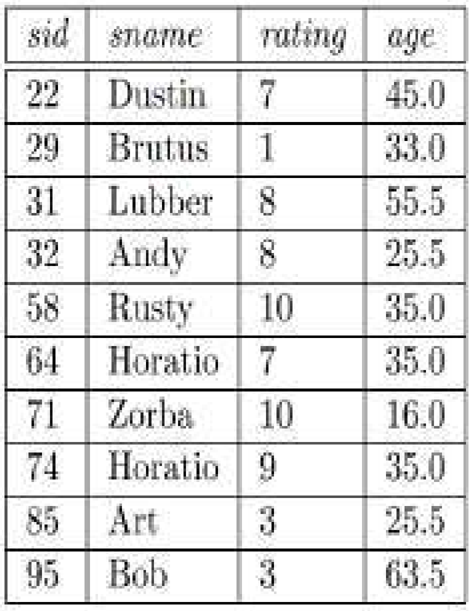
→The basic form of an SQL query is as follows:



```
SELECT    [ DISTINCT ] select-list
FROM      from-list
WHERE     qualification
```

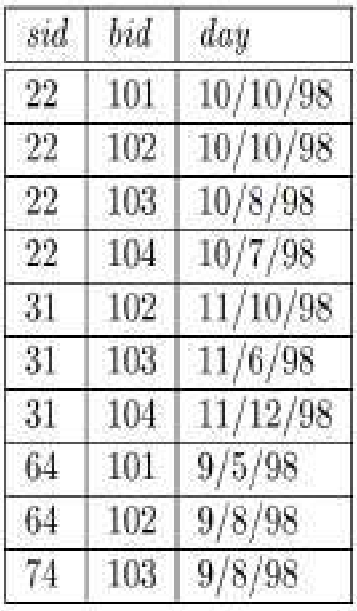
→**Such a query intuitively corresponds** to a relational algebra expression involving selections, projections, and cross-products.

→Every query must have a SELECT clause, which specifies columns to be retained in the result, and a FROM clause, which specifies a cross-product of tables. The optional WHERE clause specifies selection conditions on the tables mentioned in the FROM clause.



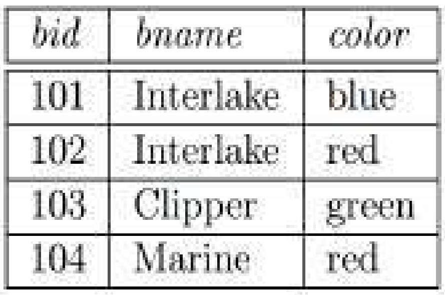
sid	*sname*	*rating*	*age*
22	Dustin	7	45.0
29	Brutus	1	33.0
31	Lubber	8	55.5
32	Andy	8	25.5
58	Rusty	10	35.0
64	Horatio	7	35.0
71	Zorba	10	16.0
74	Horatio	9	35.0
85	Art	3	25.5
95	Bob	3	63.5

Figure 5.1 An Instance *S3* 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

Figure 5.2 An Instance *R2* of Reserves



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

Figure 5.3 An Instance *B1* of Boats

- The from-list in the **FROM clause** is a list of table names. A table name can be followed by a range variable; a range variable is particularly useful when the same table name appears more than once in the from-list.
- The **select-list** is a list of (expressions involving) column names of tables named in the from-list. Column names can be prefixed by a range variable.
- The qualification in the **WHERE clause** is a Boolean combination (i.e., an expression using the logical connectives AND, OR, and NOT) of conditions of the form expression op expression, where op is one of the comparison operators {<=, =, <>, >=, >}. 2 An expression is a column name, a constant, or an (arithmetic or string) expression.
- The **DISTINCT keyword** is optional. It indicates that the table computed as an answer to this query should not contain duplicates, that is, two copies of the same row. The default is that duplicates are not eliminated.

<u>SELECT Clause:</u>
→Let us consider a simple query:

(Q15) Find the names and ages of all sailors.

```
SELECT DISTINCT S.sname, S.age
FROM   Sailors S
```

→The answer is a set of rows, each of which is a pair **<sname, age>.** If two or more sailors have the same name and age, the answer still contains just one pair with that name and age. This query is equivalent to applying the projection operator of relational algebra.

→The answer to this query with and without the keyword DISTINCT on instance S3 of Sailors is shown in **Figures 5.4 and 5.5.** The only difference is that the tuple for Horatio appears twice if DISTINCT is omitted; this is because there are two sailors called Horatio and age 35

sname	*age*
Dustin	45.0
Brutus	33.0
Lubber	55.5
Andy	25.5
Rusty	35.0
Horatio	35.0
Zorba	16.0
Art	25.5
Bob	63.5

Figure 5.4 Answer to Q15

sname	*age*
Dustin	45.0
Brutus	33.0
Lubber	55.5
Andy	25.5
Rusty	35.0
Horatio	35.0
Zorba	16.0
Horatio	35.0
Art	25.5
Bob	63.5

Figure 5.5 Answer to Q15 without DISTINCT

(Q11) Find all sailors with a rating above 7.

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

→This query uses the optional keyword **AS** to introduce a range variable. Incidentally, when we want to retrieve all columns, as in this query, SQL provides convenient shorthand: We can simply write **SELECT ***. This notation is useful for interactive querying, but it is poor style for queries that are intended to be reused and maintained.

<u>Conceptual evaluation strategy:</u>

1. Compute the cross-product of the tables in the from-list.
2. Delete those rows in the cross-product that fail the qualification conditions.
3. Delete all columns that do not appear in the select-list.
4. If DISTINCT is specified, eliminate duplicate rows.

We illustrate the conceptual evaluation strategy using the following query:

(Q1) Find the names of sailors who have reserved boat number 103.

It can be expressed in SQL as follows.

```
SELECT S.sname
FROM   Sailors S, Reserves R
WHERE  S.sid = R.sid AND R.bid=103
```

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

Figure 5.6 Instance *R3* of Reserves

sid	*sname*	*rating*	*age*
22	dustin	7	45.0
31	lubber	8	55.5
58	rusty	10	35.0

Figure 5.7 Instance *S4* of Sailors

→The first step is to construct the cross-product S4 × R3, which is shown in **Figure 5.8.**

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

Figure 5.8 $S4 \times R3$

→The second step is to apply the qualification S.sid = R.sid AND R.bid=103. This step eliminates all but the last row from the instance shown in **Figure 5.8.**

→The third step is to eliminate unwanted columns; only sname appears in the SELECT clause. This step leaves us with the result shown in **Figure 5.9**, which is a table with a single column and, as it happens, just one row.

sname
rusty

Figure 5.9 Answer to Query Q1 on R3 and S4

<u>**Examples of Basic SQL Queries:**</u>

(Q16) Find the sids of sailors who have reserved a red boat.

```
SELECT   R.sid
FROM     Boats B, Reserves R
WHERE    B.bid = R.bid AND B.color = 'red'
```

(Q2) Find the names of sailors who have reserved a red boat.

```
SELECT   S.sname
FROM     Sailors S, Reserves R, Boats B
WHERE    S.sid = R.sid AND R.bid = B.bid AND B.color = 'red'
```

(Q3) Find the colors of boats reserved by Lubber.

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

(Q4) Find the names of sailors who have reserved at least one boat.

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

<u>**Expressions and Strings in the SELECT Command:**</u>

(Q17) Compute increments for the ratings of persons who have sailed two different boats on the same day.

```
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
```

→Also, each item in a qualification can be as general as expression1 = expression2.

```
SELECT S1.sname AS name1, S2.sname AS name2
FROM   Sailors S1, Sailors S2
WHERE  2*S1.rating = S2.rating-1
```

(Q18) Find the ages of sailors whose name begins and ends with B and has at least three characters.

```
SELECT S.age
FROM   Sailors S
WHERE  S.sname LIKE 'B_%B'
```

The only such sailor is Bob, and his age is 63.5.

<u>3. UNION, INTERSECT, AND EXCEPT:</u>

→The **UNION operation** combines two relations and automatically eliminates the duplicate tuples.
→The **INTERSECT operation** finds the common tuples of two relations and eliminates the duplicate tuples.
→The **EXCEPT operation** finds the tuples which are in one relation but not in the other relation and automatically eliminates duplicate tuples.

(Q5) Find the names of sailors who have reserved a red or a green boat.

```
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')
```

→The OR query (Query Q5) can be rewritten as follows:

```
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'
```

(Q6) Find the names of sailors who have reserved both a red and a green boat.

```
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'
```

➔AND query (Query Q6) can be rewritten as follows:

```
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'
```

(Q19) Find the sids of all sailors who have reserved red boats but not green boats.

```
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'
```

```
SELECT  R.sid
FROM    Boats B, Reserves R
WHERE   R.bid = B.bid AND B.color = 'red'
EXCEPT
SELECT  R2.sid
FROM    Boats B2, Reserves R2
WHERE   R2.bid = B2.bid AND B2.color = 'green'
```

(Q20) Find all sids of sailors who have a rating of 10 or have reserved boat 104.

```
SELECT  S.sid
FROM    Sailors S
WHERE   S.rating = 10

UNION
SELECT  R.sid
FROM    Reserves R
WHERE   R.bid = 104
```

<u>4. NESTED QUERIES:</u>

<u>Introduction to Nested Queries:</u>

(Q1) Find the names of sailors who have reserved boat 103.

```
SELECT  S.sname
FROM    Sailors S
WHERE   S.sid IN ( SELECT  R.sid
                   FROM    Reserves R
                   WHERE   R.bid = 103 )
```

(Q2) Find the names of sailors who have reserved a red boat.

```
SELECT  S.sname
FROM    Sailors S
WHERE   S.sid IN ( SELECT  R.sid
                   FROM    Reserves R
                   WHERE   R.bid IN ( SELECT  B.bid
                                      FROM    Boats B
                                      WHERE   B.color = 'red' )
```

(Q21) Find the names of sailors who have not reserved a red boat.

```
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' )
```

<u>Correlated Nested Queries:</u>

(Q1) Find the names of sailors who have reserved boat number 103.

```
SELECT  S.sname
FROM    Sailors S
WHERE   EXISTS ( SELECT *
                 FROM   Reserves R
                 WHERE  R.bid = 103
                        AND  R.sid = S.sid )
```

<u>Set-Comparison Operators:</u>

(Q22) Find sailors whose rating is better than some sailor called Horatio.

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

(Q23) Find sailors whose rating is better than every sailor called Horatio.

→We can obtain all such queries with a simple modification to Query Q22: just replace ANY with ALL in the WHERE clause of the outer query.

(Q24) Find the sailors with the highest rating.

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

More Examples of Nested Queries:

(Q6) Find the names of sailors who have reserved both a red and a green boat.

```
SELECT  S.sname
FROM    Sailors S, Reserves R, Boats B
WHERE   S.sid = R.sid AND R.bid = B.bid AND B.color = 'red'
        AND S.sid IN ( SELECT  S2.sid
                       FROM    Sailors S2, Boats B2, Reserves R2
                       WHERE   S2.sid = R2.sid AND R2.bid = B2.bid
                               AND B2.color = 'green' )
```

```
SELECT  S3.sname
FROM    Sailors S3
WHERE   S3.sid IN (( SELECT  R.sid
                     FROM    Boats B, Reserves R
                     WHERE   R.bid = B.bid AND B.color = 'red' )
                     INTERSECT
                     (SELECT R2.sid
                     FROM    Boats B2, Reserves R2
                     WHERE   R2.bid = B2.bid AND B2.color = 'green' ))
```

(Q9) Find the names of sailors who have reserved all boats.

```
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 ))
```

```
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 ))
```

5. AGGREGATE OPERATORS:

→Aggregate functions operate on a multiset of values and return a single value. Typical aggregate functions are **min, max, sum, count,** and **avg.**

→These features represent a significant extension of relational algebra.
→SQL supports five aggregate operations, which can be applied on any column, say A, of a relation:

1. **COUNT ([DISTINCT] A):** The number of (unique) values in the A column.

2. **SUM ([DISTINCT] A):** The sum of all (unique) values in the A column.

3. **AVG ([DISTINCT] A):** The average of all (unique) values in the A column.

4. **MAX (A):** The maximum value in the A column.

5. **MIN (A):** The minimum value in the A column.

Examples:

(Q25) Find the average age of all sailors

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

On instance S3, the average age is 37.4.

(Q26) Find the average age of sailors with a rating of 10.

```
SELECT AVG (S.age)
FROM   Sailors S
WHERE  S.rating = 10
```

There are two such sailors, and their average age is 25.5. MIN (or MAX) can be used instead of AVG in the above queries to find the age of the youngest (oldest) sailor.

(Q27) Find the name and age of the oldest sailor.

Consider the following attempt to answer this query:

```
SELECT  S.sname, MAX  (S.age)
FROM    Sailors S
```

```
SELECT  S.sname, S.age
FROM    Sailors S
WHERE   S.age = ( SELECT  MAX  (S2.age)
                  FROM  Sailors S2 )
```

```
SELECT  S.sname, S.age
FROM    Sailors S
WHERE   ( SELECT  MAX  (S2.age)
          FROM  Sailors S2 ) = S.age
```

(Q28) Count the number of sailors.

```
SELECT COUNT (*)
FROM   Sailors S
```

(Q29) Count the number of different sailor names.

```
SELECT  COUNT  ( DISTINCT  S.sname )
FROM    Sailors S
```

(Q30) Find the names of sailors who are older than the oldest sailor with a rating of 10.

```
SELECT  S.sname
FROM    Sailors S
WHERE   S.age > ( SELECT  MAX  ( S2.age )
                  FROM    Sailors S2
                  WHERE   S2.rating = 10 )
```

```
SELECT  S.sname
FROM    Sailors S
WHERE   S.age > ALL ( SELECT  S2.age
                      FROM    Sailors S2
                      WHERE   S2.rating = 10 )
```

The GROUP BY and HAVING Clauses:

→**Group by clause** is used to group the results of a SELECT query based on one or more columns. It is also used with SQL functions to group the result from one or more tables.

→Syntax for using Group by clause is as follows,

```
SELECT     [ DISTINCT ] select-list
FROM       from-list
WHERE      qualification
GROUP BY   grouping-list
HAVING     group-qualification
```

- The select-list in the SELECT clause consists of (1) a list of column names and (2) a list of terms having the form aggop (column-name) AS new-name. The optional AS new-name term gives this column a name in the table that is the result of the query. Any of the aggregation operators can be used for aggop.
- Every column that appears in (1) must also appear in grouping-list. The reason is that each row in the result of the query corresponds to one group, which is a collection of rows that agree on the values of columns in grouping-list. If a column appears in list (1), but not in grouping-list, it is not clear what value should be assigned to it in an answer row.

- The expressions appearing in the group-qualification in the HAVING clause must have a single value per group. The intuition is that the HAVING clause determines whether an answer row is to be generated for a given group. Therefore, a column appearing in the group-qualification must appear as the argument to an aggregation operator, or it must also appear in grouping-list.

- If the GROUP BY clause is omitted, the entire table is regarded as a single group.

For example, consider the following query.

(Q31) Find the age of the youngest sailor for each rating level.

```
SELECT     S.rating, MIN (S.age)
FROM       Sailors S
GROUP BY   S.rating
```

(Q32) 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.

```
SELECT    S.rating, MIN (S.age) AS minage
FROM      Sailors S
WHERE     S.age >= 18
GROUP BY  S.rating
HAVING    COUNT (*) > 1
```

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

Figure 5.10 Instance *S3* of Sailors

rating	*age*
7	45.0
1	33.0
8	55.5
8	25.5
10	35.0
7	35.0
9	35.0
3	25.5
3	63.5

Figure 5.11 After Evaluation Step 3

rating	*age*
1	33.0
3	25.5
3	63.5
7	45.0
7	35.0
8	55.5
8	25.5
9	35.0
10	35.0

Figure 5.12 After Evaluation Step 4

rating	*minage*
3	25.5
7	35.0
8	25.5

Figure 5.13 Final Result in Sample Evaluation

<u>More Examples of Aggregate Queries:</u>

(Q33) For each red boat, find the number of reservations for this boat.

```
SELECT    B.bid, COUNT (*) AS sailorcount
FROM      Boats B, Reserves R
WHERE     R.bid = B.bid AND B.color = 'red'
GROUP BY  B.bid
```

```
SELECT    B.bid, COUNT (*) AS sailorcount
FROM      Boats B, Reserves R
WHERE     R.bid = B.bid
GROUP BY  B.bid
HAVING    B.color = 'red'
```

(Q34) Find the average age of sailors for each rating level that has at least two sailors.

```
SELECT    S.rating, AVG (S.age) AS avgage
FROM      Sailors S
GROUP BY  S.rating
HAVING    COUNT (*) > 1
```

```
SELECT    S.rating, AVG ( S.age ) AS avgage
FROM      Sailors S
GROUP BY  S.rating
HAVING    1 < ( SELECT COUNT (*)
                FROM   Sailors S2
                WHERE  S.rating = S2.rating )
```

→After identifying groups based on rating, we retain only groups with at least two sailors. The answer to this query on instance S3 is shown in **Figure 5.14.**

rating	avgage
3	44.5
7	40.0
8	40.5
10	25.5

Figure 5.14 Q34 Answer

rating	avgage
3	45.5
7	40.0
8	40.5
10	35.0

Figure 5.15 Q35 Answer

rating	avgage
3	45.5
7	40.0
8	40.5

Figure 5.16 Q36 Answer

(Q35) 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.

```
SELECT     S.rating, AVG ( S.age ) AS avgage
FROM       Sailors S
WHERE      S. age >= 18
GROUP BY   S.rating
HAVING     1 < ( SELECT COUNT (*)
                 FROM   Sailors S2
                 WHERE  S.rating = S2.rating )
```

(Q36) 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.

```
SELECT     S.rating, AVG ( S.age ) AS avgage
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 )
```

```
SELECT     S.rating, AVG ( S.age ) AS avgage
FROM       Sailors S
WHERE      S. age > 18
GROUP BY   S.rating
HAVING     COUNT (*) > 1
```

```
SELECT  Temp.rating, Temp.avgage
FROM    ( SELECT     S.rating, AVG ( S.age ) AS  avgage,
                     COUNT (*) AS  ratingcount
          FROM       Sailors S
          WHERE      S. age > 18
          GROUP BY   S.rating ) AS  Temp
WHERE  Temp.ratingcount > 1
```

(Q37) Find those ratings for which the average age of sailors is the minimum overall ratings.

```
SELECT   S.rating
FROM     Sailors S
WHERE    AVG (S.age) = ( SELECT    MIN (AVG (S2.age))
                         FROM      Sailors S2
                         GROUP BY  S2.rating )
```

```
SELECT Temp.rating, Temp.avgage
FROM   ( SELECT   S.rating, AVG (S.age) AS avgage,
         FROM     Sailors S
         GROUP BY S.rating) AS Temp
WHERE  Temp.avgage = ( SELECT MIN (Temp.avgage) FROM Temp )
```

The answer to this query on instance $S3$ is $\langle 10, 25.5 \rangle$.

As an exercise, the reader should consider whether the following query computes the same answer, and if not, why:

```
SELECT   Temp.rating, MIN ( Temp.avgage )
FROM     ( SELECT   S.rating, AVG (S.age) AS avgage,
           FROM     Sailors S
           GROUP BY S.rating ) AS Temp
GROUP BY Temp.rating
```

5. NULL VALUES:

→The SQL **NULL** is the term used to represent a missing value. A NULL value in a table is a value in a field that appears to be blank.

→A field with a NULL value is a field with no value. It is very important to understand that a NULL value is different than a zero value or a field that contains spaces.

→The basic syntax of **NULL** while creating a table:

```
SQL> CTREATE TABLE CUSTOMERS (
        ID              INT                 NOT NULL,
        NAME            VARCHAR2 (20)       NOT NULL,
        AGE             INT                 NOT NULL,
       ADDRESS          CHAR (25),
       SALARY           DECIMAL (18, 2),
       PRIMARY KEY (ID)
   );
```

→Here, **NOT NULL** signifies that column should always accept an explicit value of the given data type. There are two columns, where we did not use NOT NULL, which means these columns could be NULL.
→A field with a NULL value is one that has been left blank during record creation.
Example:
The NULL value can cause problems when selecting data, however, →because when comparing an unknown value to any other value, the result is always unknown and not included in the final results.
→You must use the **IS NULL** or **IS NOT NULL** operators in order to check for a NULL value.
→Consider the following table, CUSTOMERS having the following records:

ID	NAME	AGE	ADDRESS	SALARY
1	ANUPAMA	52	Ahmadabad	2000.00
2	ABEELA	49	Delhi	1500.00
3	RANIA	45	Kota	2000.00
4	KAVYA	47	Mumbai	6500.00
5	NAGINA	41	Bhopal	8500.00
6	NAJAH	48	Jaipur	
7	RAMEESHA	50	Indore	

→Now, following is the usage of **IS NOT NULL** operator:

```
SQL> SELECT ID, NAME, AGE, ADDRESS, SALARY
     FROM CUSTOMERS
     WHERE SALARY IS NOT NULL;
```

→This would produce the following result:

ID	NAME	AGE	ADDRESS	SALARY
1	ANUPAMA	52	Ahmadabad	2000.00
2	ABEELA	49	Delhi	1500.00
3	RANIA	45	Kota	2000.00
4	KAVYA	47	Mumbai	6500.00
5	NAGINA	41	Bhopal	8500.00

→Now, following is the usage of **IS NULL** operator:

```
SQL> SELECT ID, NAME, AGE, ADDRESS, SALARY
       FROM CUSTOMERS
       WHERE SALARY IS  NULL;
```

→This would produce the following result:

ID	NAME	AGE	ADDRESS	SALARY
6	NAJAH	48	Jaipur	
7	RAMEESHA	50	Indore	

1. Comparison Using Null Values

→It is difficult to perform comparison of valid values with **NULL** values if two valued Logic **TRUE** or **FALSE** is used. Therefore to avoid this issue three valued logic **TRUE, FALSE** or **UNKNOWN** must be used with **NULL** value.

→Consider a comparison such as **rating = 8.** If this is applied to the row for **Dan**, is this condition **TRUE** or **FALSE?** Since **Dan's rating** is UNKNOWN, it is reasonable to say that this comparison should evaluate to the value **UNKNOWN.** In fact, this is the case for the comparisons rating > 8 and rating < 8 as well. If we compare two null values using <, <, =, <> and so on, the result is always **UNKNOWN.**

→For example, if we have null in two distinct rows of the sailor relation, any comparison returns **UNKNOWN.**

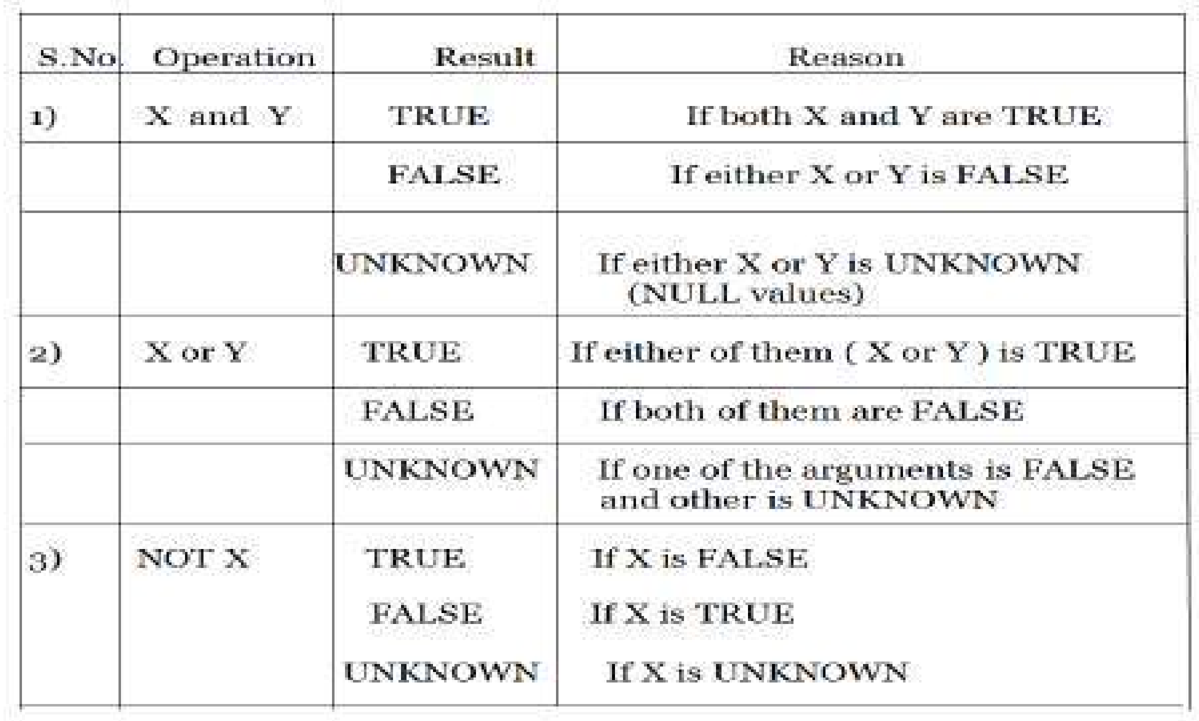
→SQL provides a special comparison operator **IS NULL** to test whether a column value is null; for example, we can say **rating IS NULL**, which would evaluate to **TRUE** on the row representing **Dan**. We can also say **rating IS NOT NULL**, which would evaluate to **FALSE** on the row for **Dan.**

2. Logical Connectives AND, OR, and NOT:

→Logical connectives with **NULL** values must be defined using three valued logic wherein expressions evaluates to three values (i.e., to **TRUE, FALSE,** or **UNKNOWN**).

→Now, the Boolean expressions such as **rating = 8 OR age < 40 and rating = 8 AND age < 40.** Considering the row for **Dan age** < **40**, the first expression evaluates to **TRUE** the value of **rating**; the second can only say **UNKNOWN.**

→The given **table** will give you a better understanding of logical operators when used with **NULL** values. Point to note here is that we are using a three valued logic **TRUE, FALSE** or **UNKNOWN** i.e., the logical condition applied may evaluate to any one of them (**UNKNOWN** is used in case of **NULL** values).

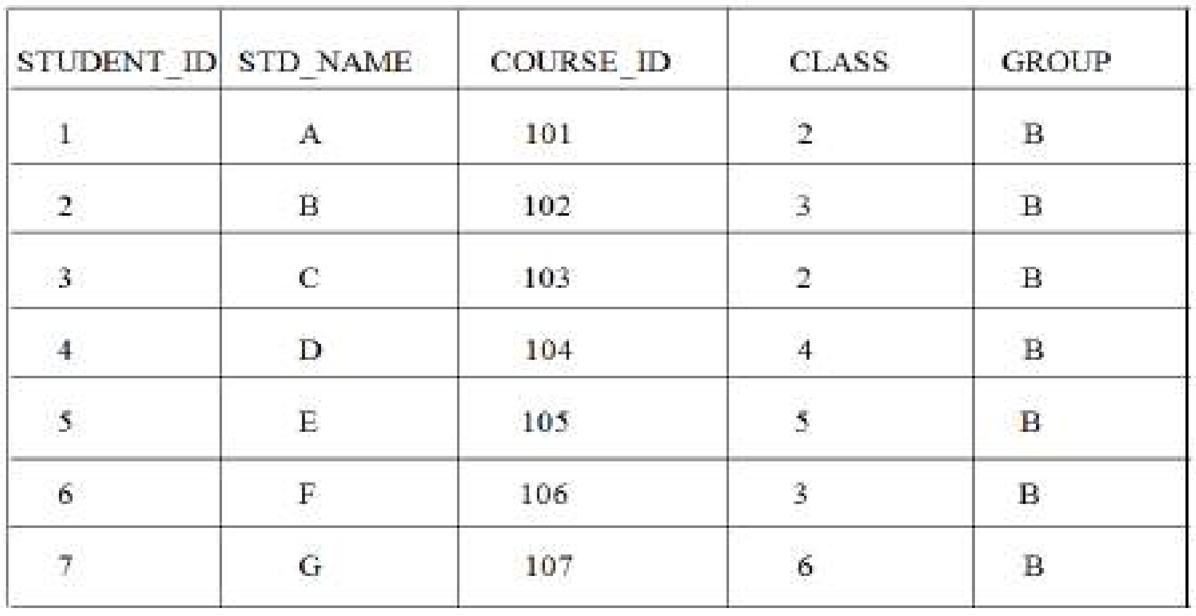


S.No.	Operation	Result	Reason
1)	X and Y	TRUE	If both X and Y are TRUE
		FALSE	If either X or Y is FALSE
		UNKNOWN	If either X or Y is UNKNOWN (NULL values)
2)	X or Y	TRUE	If either of them (X or Y) is TRUE
		FALSE	If both of them are FALSE
		UNKNOWN	If one of the arguments is FALSE and other is UNKNOWN
3)	NOT X	TRUE FALSE UNKNOWN	If X is FALSE If X is TRUE If X is UNKNOWN

Table. Logical Operators

<u>3. Impact on SQL Constructs:</u>

→As many Boolean expressions are used in SQL, it is necessary to understand the impact of NULL values on these constructs.



STUDENT_ID	STD_NAME	COURSE_ID	CLASS	GROUP
1	A	101	2	B
2	B	102	3	B
3	C	103	2	B
4	D	104	4	B
5	E	105	5	B
6	F	106	3	B
7	G	107	6	B

Table. Student Table

<u>→Example:</u>

List all names of students who belongs to group 'B'

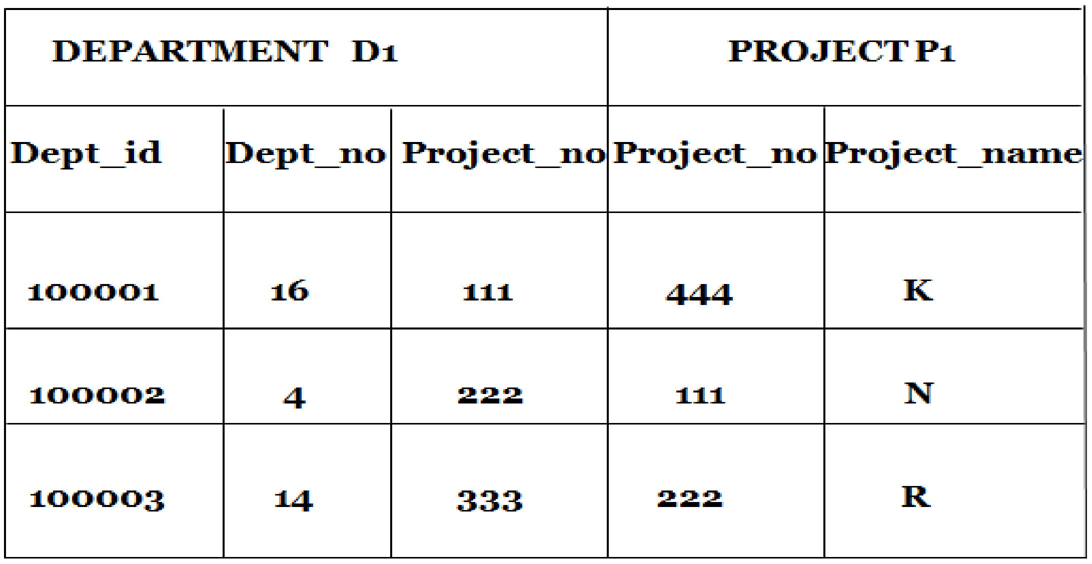
SELECT   *
<u>FROM</u>   STUDENT S
<u>WHERE</u>   S.group = 'B';

→This solution will result in the set of tuples that satisfies the '**WHERE**' condition and all other tuples that does not satisfy this condition are ignored in addition to these tuples. Tuples with **NULL** values are also ignored because for them the condition evaluates to **FALSE** or **UNKNOWN**. This elimination of rows that resulted unknown, makes the queries that

involves **EXISTS** and/or **UNIQUE** much more simple, easy to understand and makes the evaluation of these queries (nested queries especially) much easier.
→We know that the comparison of any two fields with NULL values for equality is an **UNKNOWN** value. But when it comes to **(=)** equality operator, the two **NULL** value attributes are treated as equal. If a field contains two **NULL** values then that is considered as duplicate values. Two tuples are said to be duplicates if they hold the same value or if they hold **NULL** values. So, the comparison of **NULL** values with the **"="** operator always results in **TRUE**.
→The result of all the **arithmetic operators (+, -, %, /, *)** results in an **UNKNOWN** value (**NULL**) if any one of the argument is a NULL value. Similarly, with all the aggregate operators the result is NULL if these operators are applied a NULL value. Aggregate functions simply delete the **NULL** values and then returns the result of aggregate operators i.e., **SUM, AVG, MIN, MAX, COUNT(DISTICT)** i.e., simply delete/ignore the NULL values and returns the result of other **NOT NULL** tuples. Only exception in aggregate operator is **COUNT (*)** which does not ignore/delete the **NULL** values, it counts them and then return the number of tuples in the table.

4. Outer Joins: (*************)
→We need to use outer joins to include all the tuples from the participating relations in the resulting relation.
→This is the special case of **"join"** operator which considers the NULL values. Generally **"join"** operations performs the cross product of two tables and apply certain join condition. Then it selects those rows from the cross product that satisfied the given condition. But with outer joins, DBMS allows to us select those rows which are common (satisfies the given) and even those rows that does not satisfies the given condition.

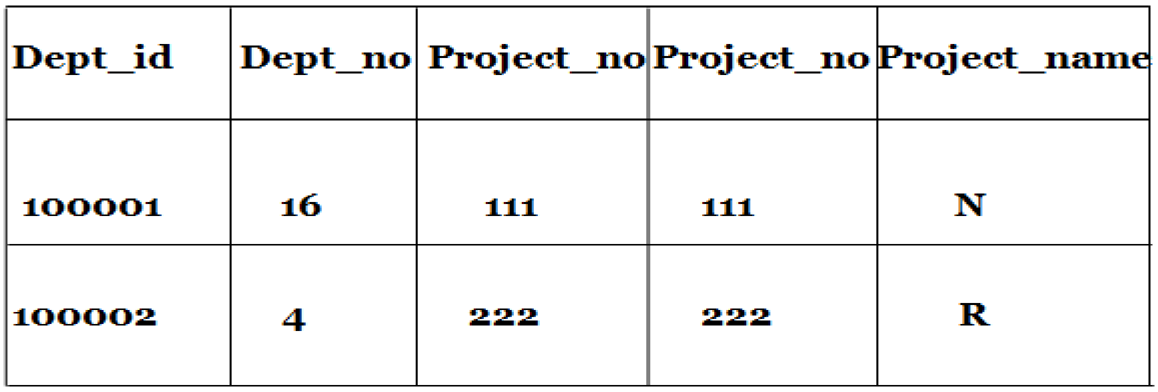
→To understand this, consider simple **instances of Project and Department** as shown in **table.**



DEPARTMENT D1			PROJECT P1	
Dept_id	Dept_no	Project_no	Project_no	Project_name
100001	16	111	444	K
100002	4	222	111	N
100003	14	333	222	R

TABLE . Intances of PROJECT and DEPARTMENT Table.

→If we perform join operation on these two tables,
SELECT * D1, * P1
FROM DEPARTMENT D1, PROJECT P1
WHERE D1.Project_no = P1.Project_no;
→The result of this statement is shown in **Table 1.**



Dept_id	Dept_no	Project_no	Project_no	Project_name
100001	16	111	111	N
100002	4	222	222	R

TABLE 1. Table Showing the Simple Join Operation.

→The **Table 1** shows the simple join operation of two tables, only those rows are selected that satisfied the condition. However, if we want to include those rows that do not satisfy the condition, then we can use the concept of **OUTER JOINS**.

→There are three types of **OUTER JOINS**. They are,

1. **LEFT OUTER JOIN**
2. **RIGHT OUTER JOIN**
3. **FULL OUTER JOIN**

1. LEFT OUTER JOIN:

→**LEFT OUTER JOIN** lists all those rows which are common to both the tables and also all those unmatched rows of the table which is specified at the left hand side.

Example:

```
SELECT   * D1, * P1
FROM     DEPARTMENT D1   LEFT OUTER JOIN PROJECT P1
WHERE    D1.Project_no = P1.Project_no;
```

→The result of this statement is shown in **Table 1A.**

DEPARTMENT D1			PROJECT P1	
Dept_id	Dept_no	Project_no	Project_no	Project_name
100001	16	111	111	N
100002	4	222	222	R
100003	14	333	NULL	NULL

TABLE 1A. Table Showing the LEFT OUTER JOIN Operation.

→So, the **LEFT OUTER JOIN** resulted in relations that have common rows from both the tables and also the row which does not have match in the other table. The values of the attributes corresponding to second table are **NULL** values.

<u>2. RIGHT OUTER JOIN:</u>

→**RIGHT OUTER JOIN** is same as the **LEFT OUTER JOIN** but the only difference is the unmatched rows of second table (specified on the right hand side) are listed along with the common rows of both the tables.

SELECT   * D1, * P1
FROM   DEPARTMENT D1   RIGHT OUTER JOIN PROJECT P1
WHERE   D1.Project_no = P1.Project_no;

→The result of this statement is shown in **Table 2B.**

DEPARTMENT D1			PROJECT P1	
Dept_id	Dept_no	Project_no	Project_no	Project_name
NULL	NULL	NULL	444	K
100001	16	111	111	N
100002	4	222	222	R

TABLE 2B. Table Showing the RIGHT OUTER JOIN.

→The values of attributes for the first table are declared as **NULL**.

<u>3. FULL OUTER JOIN:</u>

→**FULL OUTER JOIN** is same as the **RIGHT OUTER JOIN** and **LEFT OUTER JOIN** but only difference is unmatched rows of both tables are listed along with the common rows of the tables.

SELECT   * D1, * P1
FROM   DEPARTMENT D1   FULL OUTER JOIN PROJECT P1
WHERE   D1.Project_no = P1.Project_no;

→The result of this statement is shown in **Table 2C.**

DEPARTMENT D1			PROJECT P1	
Dept_id	**Dept_no**	**Project_no**	**Project_no**	**Project_name**
100001	16	111	111	N
100002	4	222	222	R
100003	14	333	NULL	NULL
NULL	NULL	NULL	444	K

TABLE 3C. Table Showing the FULL OUTER JOIN.

→In this relation as you can see all the matched and unmatched columns of both the tables are displayed, the values for the unmatched attributes are entered as **NULL**.

<u>5. Disallowing Null Values:</u>

→These fields can take on NULL values, if they are not declared as **NOT NULL**. We can restrict the insertion of NULL values for the field by declaring that field as **NOT NULL**. This means that the field cannot take **NULL** values. For the **PRIMARY KEY** Constraint i.e., the field which is declared as **PRIMARY KEY** is also declared as **NOT NULL**. This declaration is implicit declaration done by DBMS.

```
CREATE TABLE STUDENT (Sid INT NOT NULL,
                       Sname CHAR(10) NOT NULL
                      Project VARCHAR2 (15),
                      Class INT,
                      PRIMARY KEY(Sid));
```

→In this declaration i.e., creation of **STUDENT Table,** Sid is the **PRIMARY KEY** hence it must be UNIQUE and it should not be **NULL**. Project field indicates the Project taken up by the student .This field can take **NULL** values as it is possible

<u>6. Embedded SQL: (****************)</u>

→**SQL** provides a powerful declarative query language. Writing queries in SQL is usually much easier than coding the same queries in a general - purpose programming language. However, a programmer must have access to a database from a general purpose programming language for at least two reasons:

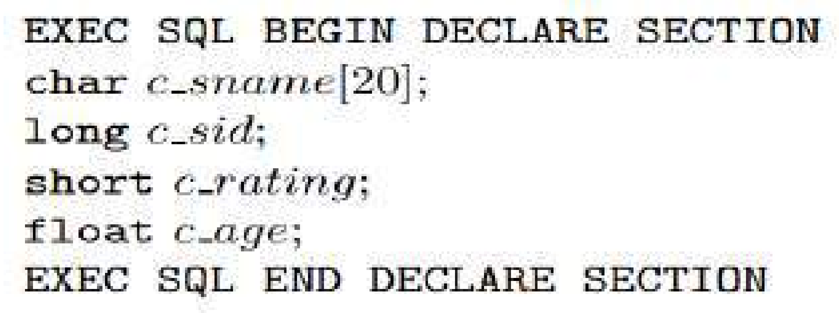
- Not all queries can be expressed in SQL, since SQL does not provide the full expressive power of a general-purpose language. That is, there exist queries that can be expressed in a language such as C, Java, or COBOL that cannot be expressed in SQL. To write such queries, we can embed SQL within a more powerful language.

- Non-declarative actions—such as printing a report, interacting with a user, or sending the results of a query to a graphical user interface—cannot be done from within SQL. Applications usually have several components, and querying or updating data is only one component; other components are written in general-purpose programming languages. For an integrated application, the programs written in the programming language must be able to access the database.

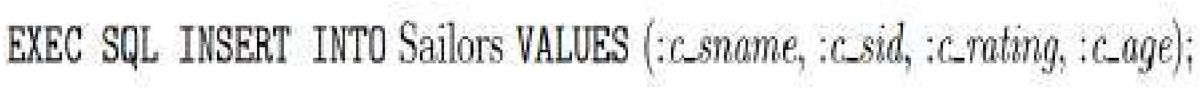
1. Declaring Variables and Exceptions:
→The SQL standard defines embeddings of SQL in a variety of programming languages such as **C, Java,** and **COBOL**.
→A language to which SQL queries are embedded is referred to as a **host language**, and the SQL structures permitted in the host language comprise embedded SQL.

→SQL statements can refer to variables defined in the host program. Such host-language variables must be prefixed by a colon (:) in SQL statements and must be declared between the commands **EXEC SQL BEGIN DECLARE SECTION** and **EXEC SQL END DECLARE SECTION.** The declarations are similar to how they would look in a C program and, as usual in C, are separated by semicolons.

→**For example,** we can declare variables **c_sname, c_sid, c_rating, and c_age** (with the initial c used as a naming convention to emphasize that these are host language variables) as follows:

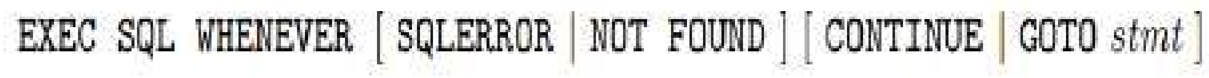

```
EXEC SQL BEGIN DECLARE SECTION
char c_sname[20];
long c_sid;
short c_rating;
float c_age;
EXEC SQL END DECLARE SECTION
```

2. Embedding SQL Statements:
→All SQL statements that are embedded within a host program must be clearly marked, with the details dependent on the host language; in **C**, SQL statements must be pre- fixed by **EXEC SQL**. An SQL statement can essentially appear in any place in the host language program where a host language statement can appear.
→**As a simple example**, the following embedded SQL statement inserts a row, whose column values are based on the values of the host language variables contained in it, into the **Sailors relation**:



```
EXEC SQL INSERT INTO Sailors VALUES (:c_sname, :c_sid, :c_rating, :c_age);
```

→The **SQLSTATE** variable should be checked for errors and exceptions after each embedded SQL statement. SQL provides the **WHENEVER** command to simplify this tedious task:



```
EXEC SQL WHENEVER [ SQLERROR | NOT FOUND ] [ CONTINUE | GOTO stmt ]
```

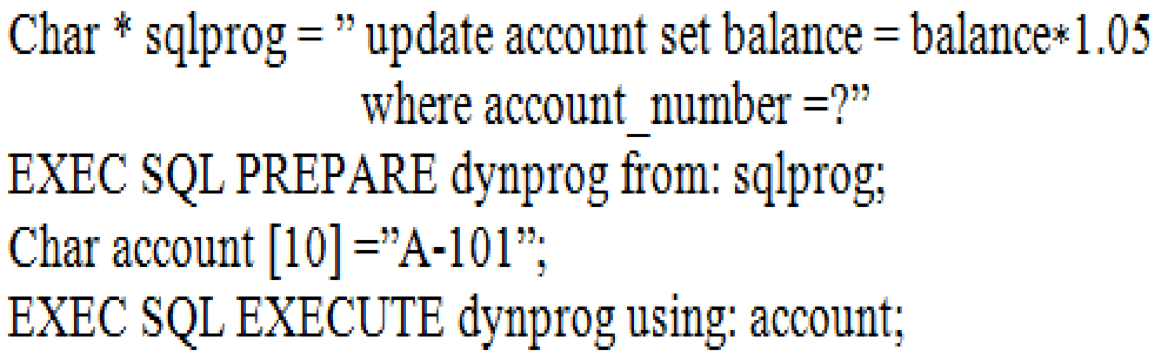
→The intent is that after each embedded SQL statement is executed, the value of **SQLSTATE** should be checked. If **SQLERROR** is specified and the value of **SQLSTATE** indicates an exception, control is transferred to stmt, which is presumably responsible for error/exception handling. Control is also transferred to stmt if **NOT FOUND** is specified and the value of S**QLSTATE** is 02000, which denotes **NO DATA**.

7. Dynamic SQL: (****************)

→The dynamic SQL component of SQL allows programs to construct and submit SQL queries at run time.

→Using dynamic SQL, programs can create SQL queries as strings at run time (perhaps based on input from the user) and can either have them executed immediately or have they prepared for subsequent use. Preparing a dynamic SQL statement compiles it, and subsequent uses of the prepared statement use the compiled version.

→SQL defines standards for embedding dynamic SQL calls in a host language, such as C, as in the following example.



```
Char * sqlprog = " update account set balance = balance*1.05
                    where account_number =?"
EXEC SQL PREPARE dynprog from: sqlprog;
Char account [10] ="A-101";
EXEC SQL EXECUTE dynprog using: account;
```

→The dynamic SQL program contains a? which is a place holder for a value that is provided when the SQL program is executed?

8. CURSORS: (************)

→A major problem in embedding SQL statements in a host language like C is that an impedance mismatch occurs because SQL operates on sets of records, whereas languages like C do not cleanly support a set-of-records abstraction. The solution is to essentially provide a mechanism that allows us to retrieve rows one at a time from a relation. This mechanism is called a **cursor.**

→A **cursor** is a temporary work area created in the system memory when a SQL statement is executed. A **cursor** contains information on a select statement and the rows of data accessed by it.
→This temporary work area is used to store the data retrieved from the database, and manipulate this data. A **cursor** can hold more than one row, but can process only one row at a time. The set of rows the cursor holds is called the ***active* set**.
→ 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.

1. Basic Cursor Definition and Usage:

→Cursors enable us to examine in the host language program a collection of rows computed by an embedded SQL statement:

- We usually need to open a cursor if the embedded statement is a **SELECT** (i.e., a query)..
- **INSERT, DELETE,** and **UPDATE** statements typically don't require a cursor, although some variants of DELETE and UPDATE do use a cursor.

→ **As an example,** we can **find the name and age of a sailor**, specified by assigning a value to the host variable **c_sid** as follows:

```
EXEC SQL SELECT S.sname, S.age
         INTO   :c_sname, :c_age
         FROM   Sailors S
         WHERE  S.sid = :c_sid;
```

→The INTO clause allows us to assign the columns of the single answer row to the host variables **c_sname** and **c_age.**

→Computes the names and ages of all sailors with a rating greater than the current value of the host variable c_minrating?

```
SELECT S.sname, S.age
FROM   Sailors S
WHERE  S.rating > :c_minrating
```

→This query returns a collection of rows, not just one row. The solution is to use a **cursor:**

```
DECLARE sinfo CURSOR FOR
SELECT S.sname, S.age
FROM   Sailors S
WHERE  S.rating > :c_minrating;
```

→This code can be included in a C program, and once it is executed, the cursor **sinfo** is defined. Subsequently, we can **open the cursor**:

```
OPEN sinfo;
```

→ We can use the **FETCH** command to read the first row of **cursor sinfo** into host language variables:

```
FETCH sinfo INTO :c_sname, :c_age;
```

→When we are done with a cursor**, we can close it:**

```
CLOSE sinfo;
```

2. Properties of Cursors:
→The general form of a cursor declaration is:

```
DECLARE cursorname [INSENSITIVE] [SCROLL] CURSOR FOR
        some query
        [ ORDER BY order-item-list ]
        [ FOR READ ONLY | FOR UPDATE ]
```

→ A **cursor** can be declared to be a read-only cursor (**FOR READ ONLY**) or, if it is a cursor on a base relation or an updatable view, to be an updatable cursor (**FOR UPDATE**).
→If it is updatable, simple variants of the **UPDATE** and **DELETE** commands allow us to update or delete the row on which the cursor is positioned.

→**For example**, if **sinfo** is an updatable cursor and is open, we can execute the following statement:

```
UPDATE Sailors S
SET    S.rating = S.rating - 1
WHERE  CURRENT of sinfo;
```

→ This embedded SQL statement **modifies** the rating value of the row currently pointed to by cursor sinfo; similarly, we can delete this row by executing the next statement:

```
DELETE Sailors S
WHERE  CURRENT of sinfo;
```

→A cursor is updatable by default unless it is a **scrollable** or **insensitive cursor**, in which case it is read-only by default.

→If the keyword **SCROLL** is specified, the cursor is scrollable, which means that variants of the FETCH command can be used to position the cursor in very flexible ways; otherwise, only the basic FETCH command, which retrieves the next row, is allowed
→If the keyword **INSENSITIVE** is specified, the cursor behaves as if it is ranging over a private copy of the collection of answer rows.

→**For example,** while we are fetching rows using the sinfo cursor, we might modify rating values in Sailor rows by concurrently executing the command:

```
UPDATE Sailors S
SET    S.rating = S.rating - 1
```

→The order-item-list is a list of order-items; an order-item is a column name, optionally followed by one of the keywords ASC or DESC.

→Suppose that a cursor is opened on this query, with the clause:

```
ORDER BY minage ASC, rating DESC
```

The answer is sorted first in ascending order by **minage**, and if several rows have the same **minage** value, these rows are sorted further in descending order by rating. The cursor would fetch the rows in the order shown in **Figure 5.18.**

rating	*minage*
8	25.5
3	25.5
7	35.0

Figure 5.18 Order in which Tuples Are Fetched

<u>8. ODBC AND JDBC: (************)</u>

ODBC and **JDBC**, short for **<u>Open DataBase Connectivity</u>** <u>and</u> **<u>Java DataBase Connectivity</u>**, also enable the integration of SQL with a general-purpose programming language. Both ODBC and JDBC expose database capabilities in a standardized way to the application programmer through an **application programming interface (API).**

<u>ODBC:</u>

→The **<u>Open Database Connectivity (ODBC)</u>** standard defines a way for an application program to communicate with a database server.

→**ODBC** defines an application program interface (**API**) that applications can use to open a connection with a database, send queries and updates, and get back results. Applications such as graphical user interfaces, statistics packages, and spreadsheets can make use of the same ODBC API to connect to any database server that supports **ODBC**.

```
int ODBCexample()
{
      RETCODE error;
      HENV env; /* environment */
      HDBC conn; /* database connection */

      SQLAllocEnv(&env);
      SQLAllocConnect(env, &conn);
      SQLConnect(conn, "aura.bell-labs.com", SQL_NTS, "avi", SQL_NTS,
                        "avipasswd", SQL_NTS);
      {
            char branchname[80];
            float balance;
            int lenOut1, lenOut2;
            HSTMT stmt;

            SQLAllocStmt(conn, &stmt);
            char * sqlquery = "select branch_name, sum (balance)
                                    from account
                                    group by branch_name";
            error = SQLExecDirect(stmt, sqlquery, SQL_NTS);
            if (error == SQL_SUCCESS) {
                  SQLBindCol(stmt, 1, SQL_C_CHAR, branchname , 80, &lenOut1);
                  SQLBindCol(stmt, 2, SQL_C_FLOAT, &balance, 0 , &lenOut2);
                  while (SQLFetch(stmt) >= SQL_SUCCESS) {
                        printf (" %s %g\n", branchname, balance);
                  }
            }
      }
      SQLFreeStmt(stmt, SQL_DROP);
      SQLDisconnect(conn);
      SQLFreeConnect(conn);
      SQLFreeEnv(env);
}
```

Figure 4.9 ODBC code example.

JDBC:

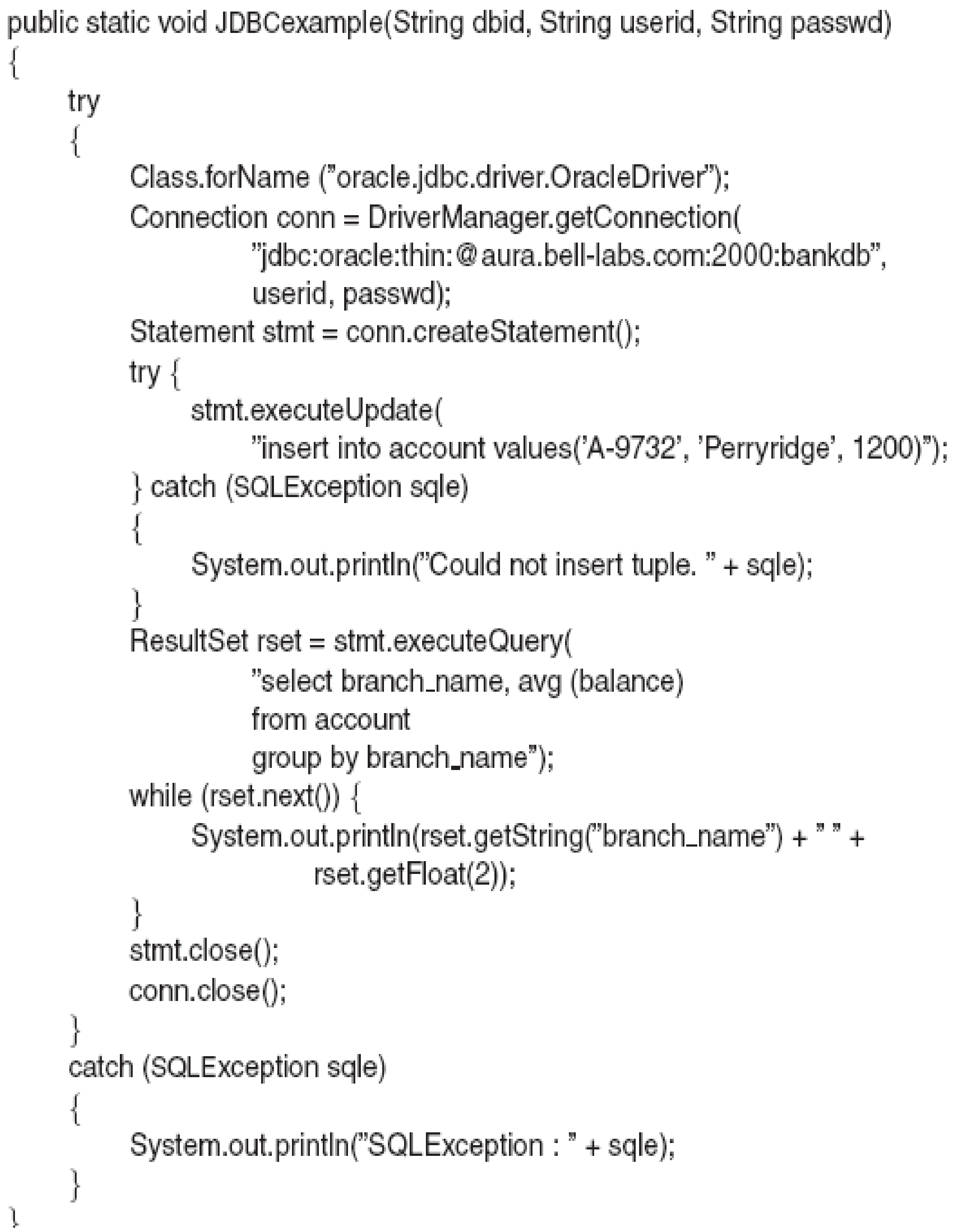
→The **JDBC (**"**Java Database Connectivity**") standard defines an API that Java programs can use to connect to database servers.

→Java Database Connectivity (JDBC) is an application program interface (API) specification for connecting programs written in Java to the data in popular databases.

→JDBC supports a variety of features for querying and updating data, and for retrieving query results.

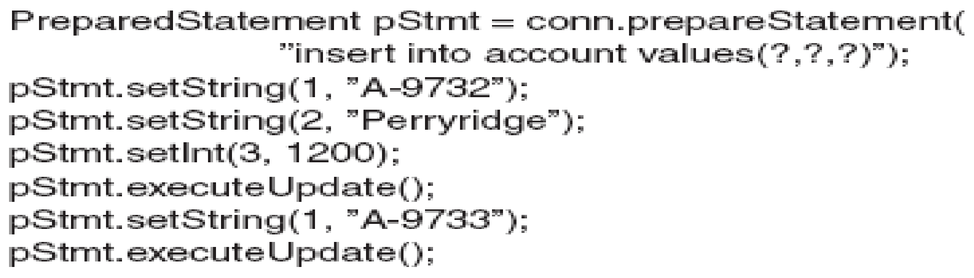
→JDBC also supports metadata retrieval, such as querying about relations present in the database and the names and types of relation attributes.

→**Figure 4.10** shows an example Java program that uses the JDBC interface.



```
public static void JDBCexample(String dbid, String userid, String passwd)
{
      try
      {
            Class.forName ("oracle.jdbc.driver.OracleDriver");
            Connection conn = DriverManager.getConnection(
                        "jdbc:oracle:thin:@aura.bell-labs.com:2000:bankdb",
                        userid, passwd);
            Statement stmt = conn.createStatement();
            try {
                  stmt.executeUpdate(
                        "insert into account values('A-9732', 'Perryridge', 1200)");
            } catch (SQLException sqle)
            {
                  System.out.println("Could not insert tuple. " + sqle);
            }
            ResultSet rset = stmt.executeQuery(
                        "select branch_name, avg (balance)
                        from account
                        group by branch_name");
            while (rset.next()) {
                  System.out.println(rset.getString("branch_name") + " " +
                              rset.getFloat(2));
            }
            stmt.close();
            conn.close();
      }
      catch (SQLException sqle)
      {
            System.out.println("SQLException : " + sqle);
      }
}
```

→The code fragment in **Figure 4.11** shows how prepared statements can be used.



```
PreparedStatement pStmt = conn.prepareStatement(
                  "insert into account values(?,?,?)");
pStmt.setString(1, "A-9732");
pStmt.setString(2, "Perryridge");
pStmt.setInt(3, 1200);
pStmt.executeUpdate();
pStmt.setString(1, "A-9733");
pStmt.executeUpdate();
```

Figure 4.11 Prepared statements in JDBC code.

Architecture:

→The architecture of ODBC/JDBC has four main components:

1. **The application**
2. **The driver manager**

3. **Several data source specific drivers, and**
4. **The corresponding data sources.**

→The **application** initiates and terminates the connection with the data source. It sets transaction boundaries, submits SQL statements, and retrieves the results—all through a well-defined interface as specified by the ODBC/JDBC API. The primary goal of the driver manager is to load ODBC/JDBC drivers and to pass ODBC/JDBC function calls from the application to the correct driver.
→The **driver manager** also handles ODBC/JDBC initialization and information calls from the applications and can log all function calls. In addition, the driver manager performs some rudimentary error checking. →The **driver** establishes the connection with the data source. In addition to submitting requests and returning request results, the driver translates data, error formats, and error codes from a form that is specific to the data source into the ODBC/JDBC standard.
→The **data source** processes commands from the driver and returns the results.

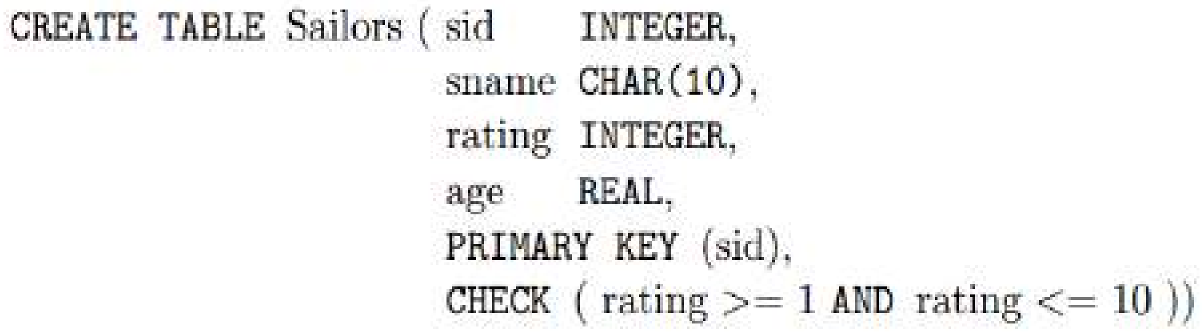
→**Drivers** in **JDBC** are classified into four types depending on the architectural relationship between the application and the data source:
1. **Type I (bridges)**: This type of driver translates JDBC function calls into function calls of another API that is not native to the DBMS. An example is an ODBCJDBC bridge. In this case the application loads only one driver, namely the bridge.
2. **Type II (direct translation to the native API):** This driver translates JDBC function calls directly into method invocations of the API of one specific data source. The driver is dynamically linked, and is specific to the data source.
3. **Type III (network bridges):** The driver talks over a network to a middle-ware server that translates the JDBC requests into DBMS-specific method invocations. In this case, the driver on the client site (i.e., the network bridge) is not DBMS specific.
4. **Type IV (direct translation over sockets)**: Instead of calling the DBMS API directly, the driver communicates with the DBMS through Java sockets. In this case the driver on the client side is DBMS-specific.

9. COMPLEX INTEGRITY CONSTRAINTS IN SQL:

Integrity constraints need not only be applied on single columns, they can also be applied on single table or group of tables (called **assertions**).

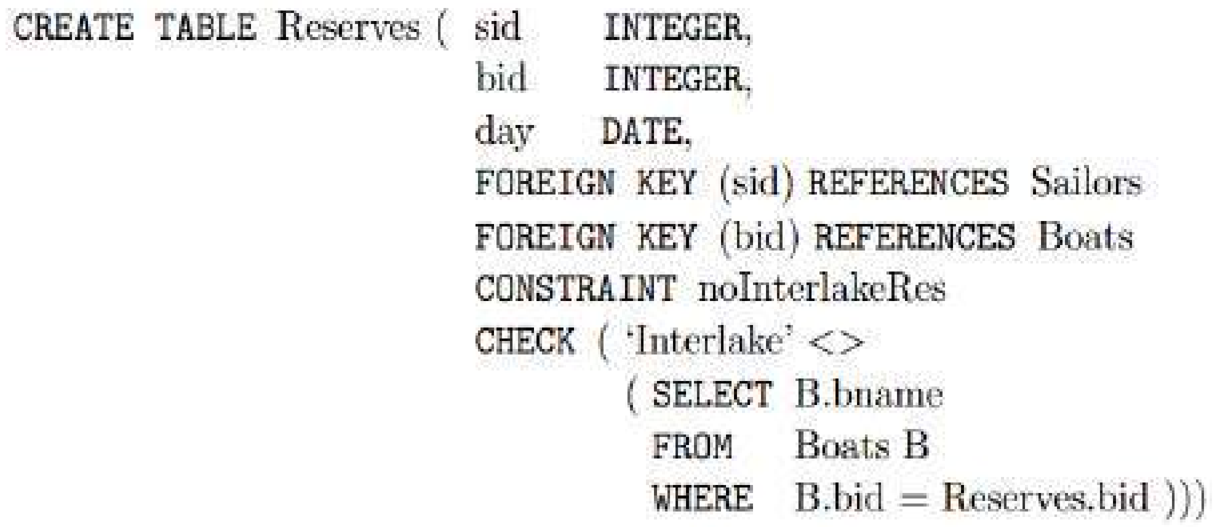
Constraints over a Single Table:

We can specify complex constraints over a single table using table constraints, which have the form CHECK conditional-expression. For example, to ensure that rating must be an integer in the range 1 to 10, we could use:



```
CREATE TABLE Sailors ( sid     INTEGER,
                       sname   CHAR(10),
                       rating  INTEGER,
                       age     REAL,
                       PRIMARY KEY (sid),
                       CHECK ( rating >= 1 AND rating <= 10 ))
```

To enforce the constraint that Interlake boats cannot be reserved, we could use:

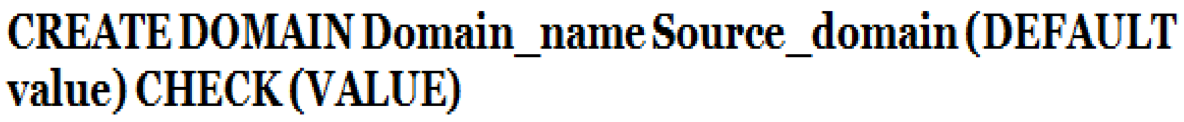


```
CREATE TABLE Reserves ( sid      INTEGER,
                        bid      INTEGER,
                        day      DATE,
                        FOREIGN KEY (sid) REFERENCES Sailors
                        FOREIGN KEY (bid) REFERENCES Boats
                        CONSTRAINT noInterlakeRes
                        CHECK ( 'Interlake' <>
                                ( SELECT B.bname
                                  FROM   Boats B
                                  WHERE  B.bid = Reserves.bid )))
```

When a row is inserted into Reserves or an existing row is modified, the conditional expression in the CHECK constraint is evaluated. If it evaluates to false, the command is rejected.

<u>Domain Constraints:</u> (**************)
→A user can define a new domain using the CREATE DOMAIN statement, which makes use of CHECK constraints.

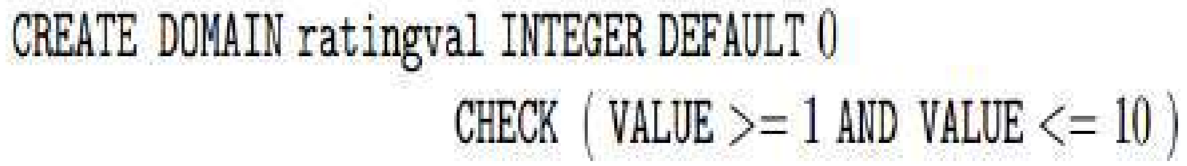
→The syntax for creating a new domain is,



```
CREATE DOMAIN Domain_name Source_domain (DEFAULT
value) CHECK (VALUE)
```

CREATE DOMAIN: A statement or keyword used to define a new domain.
Domain_name : Name of the new domain.
Source_domain: Name of the source domain from which new domain is derived.
DEFAULT value: We can also provide default values for the domains.
CHECK: This option is used to restrict the values in the particular field (for which a new domain is specified). This option provides a condition that must be checked by all the tuples of the column.
VALUE: The key word is used to provide a value to a domain variable.

Example:



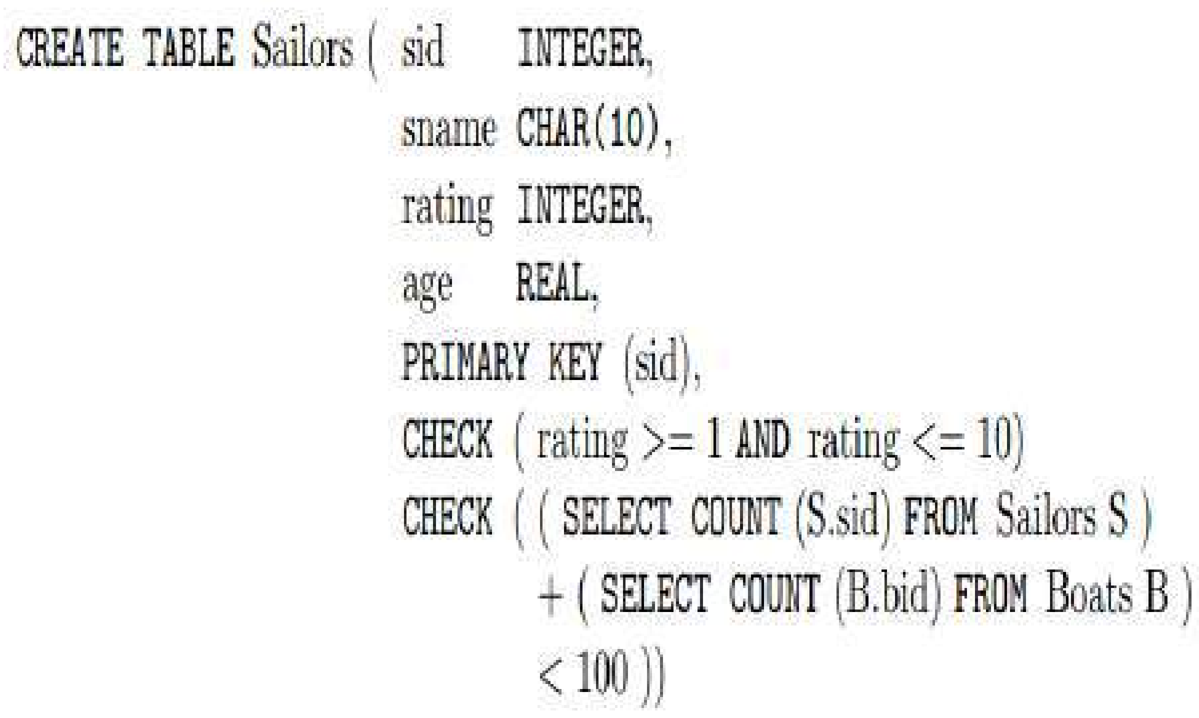
```
CREATE DOMAIN ratingval INTEGER DEFAULT 0
                        CHECK ( VALUE >= 1 AND VALUE <= 10 )
```

INTEGER is the base type for the domain **ratingval**, and every **ratingval** value must be of this type. Values in **ratingval** are further restricted by using a **CHECK** constraint; in defining this constraint, we use the keyword **VALUE** to refer to a value in the domain.

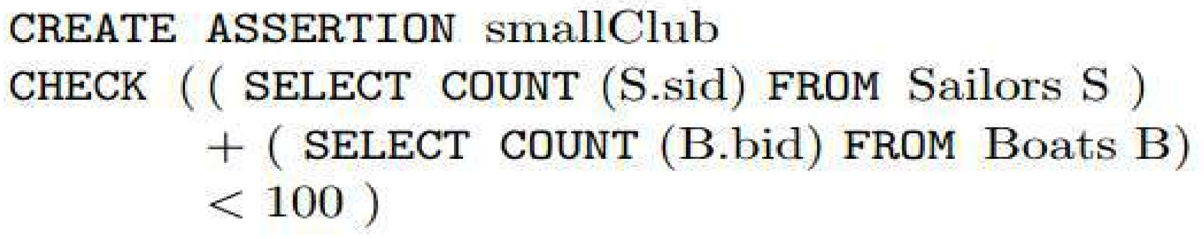
<u>Assertions: ICs over Several Tables:</u>

→**Assertions** are group of tables on which a constraint is applied. Unlike table constraints which are applied on single table, assertions are applied on multiple tables.

Example:
As an example, suppose that we wish to enforce the constraint that the <u>number of boats plus the number of sailors</u> should be <u>less than 100</u>. We could try the following table constraint:


```
CREATE TABLE Sailors ( sid    INTEGER,
                       sname  CHAR(10),
                       rating INTEGER,
                       age    REAL,
                       PRIMARY KEY (sid),
                       CHECK ( rating >= 1 AND rating <= 10)
                       CHECK ( ( SELECT COUNT (S.sid) FROM Sailors S )
                           + ( SELECT COUNT (B.bid) FROM Boats B )
                           < 100 ))
```

→This solution suffers from two drawbacks. It is associated with **Sailors**, although it involves **Boats** in a completely symmetric way. More important, if the <u>**Sailors** table is empty</u>, this constraint is defined (as per the semantics of table constraints) to always hold, even if we <u>have more than 100 rows</u> in <u>**Boats**</u>! We could extend this constraint specification to check that Sailors is nonempty, but this approach becomes very cumbersome. The best solution is to create an **assertion**, as follows:


```
CREATE ASSERTION smallClub
CHECK (( SELECT COUNT (S.sid) FROM Sailors S )
       + ( SELECT COUNT (B.bid) FROM Boats B)
       < 100 )
```

<u>10. TRIGGERS AND ACTIVE DATABASES:(*****)</u>

→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.**
→<u>A **trigger** description contains three parts:</u>

1. **Event**

2. **Condition**
3. **Action**

1. **Event:** A change to the database that activates the trigger.

→Event describes the modifications done to the database which lead to the activation of trigger. The following are fall under the category of events,

i) Inserting, updating, deleting columns of the tables or rows of tables may activate the trigger.
ii) Creating, altering or dropping any database object may also lead to activation of triggers.
iii) An error message or user log-on or log-off may also activate the trigger.

2. **Condition:** A query or test that is run when the trigger is activated.

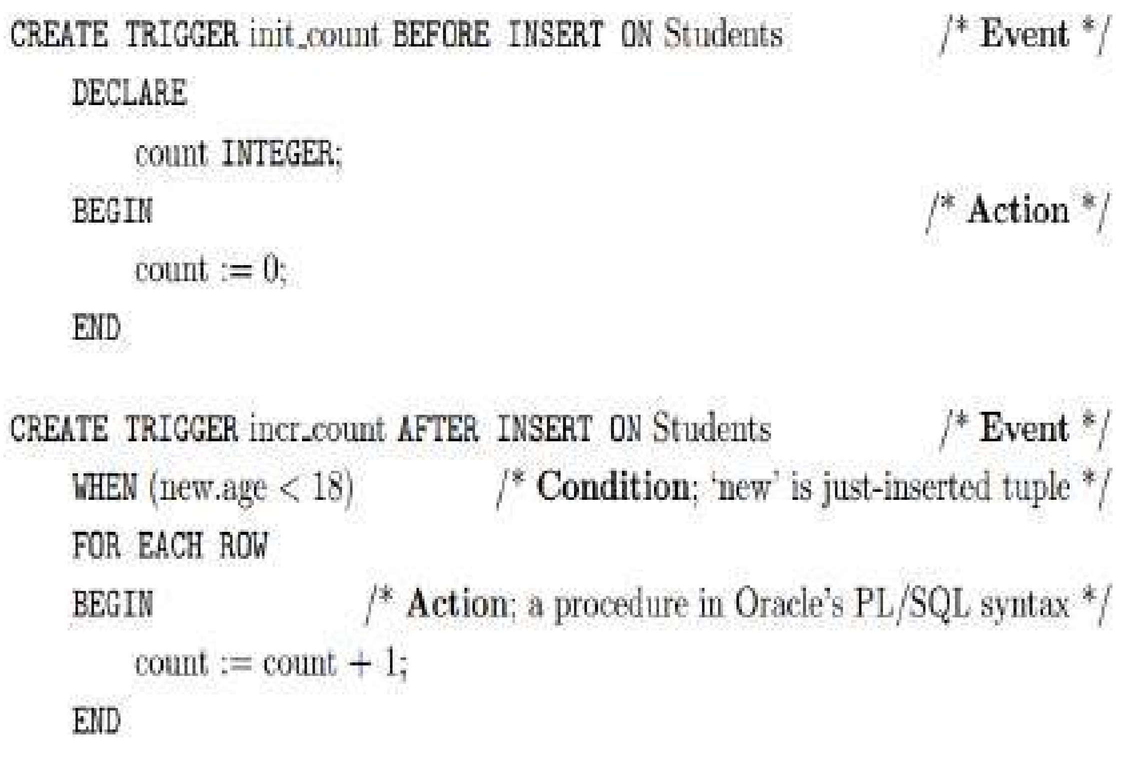
→Conditions are used to specify whether the particular action must be performed or not. If the condition is evaluated to true then the respective action is taken otherwise the action is rejected.

3. **Action:** A procedure that is executed when the trigger is activated and its condition is true.

→The examples shown in **Figure 5.19**

→The trigger called **init_count** initializes a counter variable before every execution of an INSERT statement that adds tuples to the Students relation.

→The trigger called **incr_count** increments the counter for each inserted tuple that satisfies the condition age < 18.



```
CREATE TRIGGER init_count BEFORE INSERT ON Students            /* Event */
    DECLARE
        count INTEGER;
    BEGIN                                                      /* Action */
        count := 0;
    END

CREATE TRIGGER incr_count AFTER INSERT ON Students             /* Event */
    WHEN (new.age < 18)         /* Condition; 'new' is just-inserted tuple */
    FOR EACH ROW
    BEGIN              /* Action; a procedure in Oracle's PL/SQL syntax */
        count := count + 1;
    END
```

Figure 5.19 Examples Illustrating Triggers

→A **row-level trigger** is activated for each modified record, a **statement-level trigger** is activated only once per INSERT command.

Advantages of trigger:

1) Triggers can be used as an alternative method for implementing referential integrity constraints.
2) By using triggers, business rules and transactions are easy to store in database and can be used consistently even if there are future updates to the database.
3) It controls on which updates are allowed in a database.
4) When a change happens in a database a trigger can adjust the change to the entire database.
5) Triggers are used for calling stored procedures.

10. DESIGNING ACTIVE DATABASES:

→Active database contains a set of triggers and therefore it becomes quite difficult to maintain active database.
→What triggers are activated in what order can be hard to understand because a statement can activate more than one trigger and the action of one trigger can activate other triggers.

Why Triggers Can Be Hard to Understand:

→In an active database system, when the DBMS is about to execute a statement that modifies the database, it checks whether some trigger is activated by the statement. If so, the DBMS processes the trigger by evaluating its condition part, and then (if the condition evaluates to true) executing its action part.
→If a statement activates more than one trigger, the DBMS typically processes all of them, in some arbitrary order. The execution of this action part of a trigger may in turn activate another trigger. In particular, the execution of the action part of a trigger could again activate the same trigger; such triggers are called **recursive triggers**. The potential for such **chain activations**, and the unpredictable order in which a DBMS processes **activated triggers**, can make it difficult to understand the effect of a collection of triggers.

Constraints versus Triggers:

→Triggers are more flexible than integrity constraints and the potential uses of triggers go beyond maintaining database integrity.
→Triggers are used to maintain the data integrity in the database. Whenever a change (update, insert or delete) is done in a database, a trigger can be used to indicate that change. There are several uses of triggers.

- To maintain data integrity.
- To identity the unusual events that occurs in a database.
- For security checks and also for auditing.

Chapter-5

5. SCHEMA REFINEMENT(NORMAL FORMS)

Schema Refinement (Normalization) : Purpose of Normalization or schema refinement, concept of functional dependency, normal forms based on functional dependency(1NF, 2NF and 3 NF), concept of surrogate key, Boyce-codd normal form(BCNF), Lossless join and dependency preserving decomposition, Fourth normal form(4NF).

1. SCHEMA REFINEMENT (*************)

→A schema can be defined as a complete description of database. The specifications for database schema are provided during the database design stage and this schema does not change frequently.

→ **Schema Refinement** is a technique of organizing the data in the database. It is a systematic approach of decomposing tables to eliminate data redundancy and undesirable characteristics like Insertion, Update and Deletion Anomalies.

→**Schema refinement** is the process that re-defines (refining) the schema of a relation so as to solve the problems caused by redundantly storing the information.

→**Redundancy** refers to repetition of same data or duplicate copies of same data stored in different locations.

→The **Schema Refinement** refers to refine the schema by using some technique. The best technique of schema refinement is **decomposition**.

→**Decomposition** can eliminate the redundancy.

1. Problems Caused by Redundancy :((***********)

→Redundancy is a data organization issue. It allows unnecessary duplication of data to be stored within the database. If modifications are performed to redundant data, then it is necessary to perform the same modification in multiple fields of database.

→Storing the same information redundantly, that is, in more than one place within a database, can lead to several problems:

1. **Redundant storage.**
2. **Update anomalies.**
3. **Insertion anomalies.**
4. **Deletion anomalies.**

→Schema diagram for Employee database is as follows,

→**Example:**

Employee Emp_id, Emp_name, Emp_section_id, Job_section, grade.

Emp_id	Emp_name	Emp_section_id	Job_section	grade
100001	ANUPAMA	111	CLERK	D
100002	RAMEESHA	222	Secretary	B
100002	RAMEESHA	222	Secretary	B
100002	RAMEESHA	222	Secretary	B
100003	NAGINA	333	MANAGER	A
100001	ANUPAMA	111	CLERK	D
100004	KAVYA	444	Asst.Manager	C

TABLE. An Instance of the Employee relation.

→Consider the above database **table**. The three tuples with **Emp_id 100002** and two tuples with **Emp_id 100001** repeat the same name and same job section information. The repetition wastes space as well as causes data inconsistency i.e., this redundant data may lead to **loss of data integrity.**

→For example, some update operation is being carried out, entering new record for an employee with id **100002.** This must be done multiple time i.e., it must be done for each file witch stores the employees details. This leads to redundant storage i.e., the same information is stored multiple times.

1. Redundant storage: Some information is stored repeatedly.

2. Update anomalies: If one copy of such repeated data is updated, an inconsistency is created unless all copies are similarly updated.
→If the update operation is performed, for example, the Emp_section_id 268 is updated to 520 and this correction is made only to the first record of the database, then this may lead to inconsistent data unless all the copies in the database are updated. This is referred to as update anomalies. The changes must be done to all the copies of data.

3. Insertion anomalies: It may not be possible to store some information unless some other information is stored as well.
→For example, if a new employee record is being entered, who has not yet assigned an Emp_id, now if we assume that the null values are not allowed, then it impossible to enter the new record unless the new employee has been assigned an Emp_id. This is called insertion anomalies.

4. Deletion anomalies: It may not be possible to delete some information without losing some other information as well.
→For example, if we want to delete the grade entries where grade is equal to 'A' then all the information of Emp_section_id 268 will be deleted/loss.

2. Use of Decompositions:
→**Decomposion** is the solution to the problem caused by data redundancy. **Decomposition** means breaking up the large schema into smaller multiple Schemas. **Decomposition** helps to remove all the anomalies and helps to maintain data integrity.

→We can restrict redundancy in **Employee** database by dividing it into two smaller relations/Schemas as in **table1R** and **Table2R.**

→Now we can easily update **Emp_section_id** in the **Schema Section** without bothering about the updations in the other tuples. To insert a new tuple, we can directly insert the new record in the Schema section (With the help of **Emp_section-id**) even if the new employee has not yet been assigned the **Emp_id**. To delete the entry with the **grade** equal to 'A', we can do it directly on the **Section schema** which does not lead to loss of other information. Thus, **decomposion eliminates the Problems caused by different anomalies**.

Emp_id	Emp_name	Job_section	grade
100001	ANUPAMA	CLERK	D
100002	RAMEESHA	Secretary	B
100002	RAMEESHA	Secretary	B
100002	RAMEESHA	Secretary	B
100003	NAGINA	MANAGER	A
100001	ANUPAMA	CLERK	D
100004	KAVYA	Asst.Manager	C

TABLE1R. An Instance of the Employee relation.

Emp_section_id	grade
100001	D
100002	B
100003	A
100004	C

TABLE 2R. An Instance of the Section Relation

2. FUNCTIONAL DEPENDENCIES :(**********)

→**FD** is defined as the attributes of a table is said to be dependent on each other when an attribute of a table uniquely identifies another attribute of the same table.

→A **functional dependency (FD)** is a kind of IC (Integrity Constraints) that generalizes the concept of a key. Let **R** be a relation schema and let **X** and **Y** be nonempty sets of attributes in **R**. We say that an instance r of R satisfies the **FD X → Y** if the following holds for every pair of **tuples t1 and t2 in r**:

If **t1.X = t2.X**, **then t1.Y = t2.Y**

→That is, an **FD X → Y** says that is two tuples values in attributes X, must be in association with two tuples values in attributes Y.

→If column **X** of a table uniquely identifies the column Y of same table then it can represented as **X → Y** (Attribute Y is functionally dependent on attribute X).

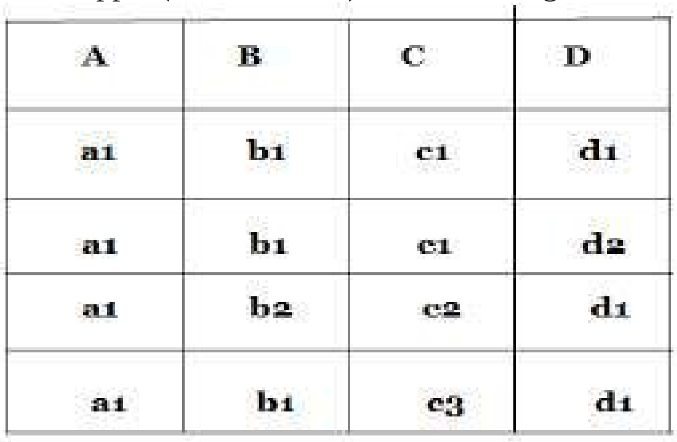
→Consider the relation Employee in which **Emp_id** and **Social_securiy_number** are the two attributes. Here, if the value of **Social_securiy_number** is used to determine the value of **Emp_id** then **Emp_id** is said to be functionally dependent on **Social_securiy_number**. This dependency can be diagrammatically shown as,

Emp_id → Social_securiy_number

Examples:

1. The **Table** illustrates the meaning of the **FD AB → C** by showing an instance that satisfies this dependency.

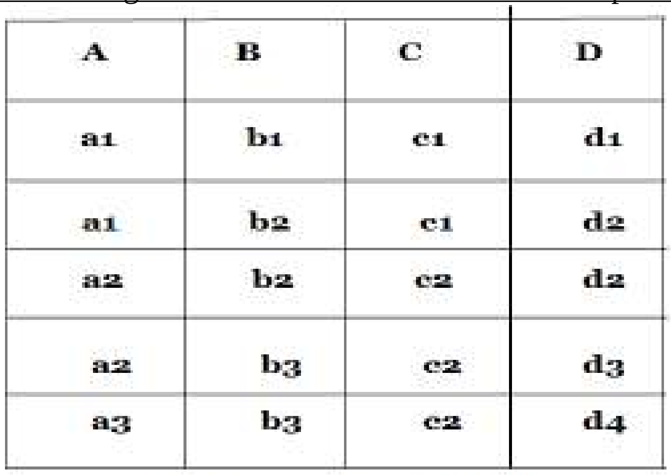
→Here, the first two tuples shows that an FD is not the same as a key constraint, although the **FD** is not violated, **AB** is clearly not a key for the relation. The third and fourth tuples illustrate that if two tuples differ in either the A field or the **B** field, they can differ in the **C** field without violating the **FD**.
→Suppose, if we add a tupple **(a1, b1, c3, d2)** to the resulting instance would violate **FD.**



A	B	C	D
a1	b1	c1	d1
a1	b1	c1	d2
a1	b2	c2	d1
a1	b1	c3	d1

TABLE. An Instance that satisfies AB → C.

2. Let us consider the following relation **r** to see which functional dependencies are satisfied.



A	B	C	D
a1	b1	c1	d1
a1	b2	c1	d2
a2	b2	c2	d2
a2	b3	c2	d3
a3	b3	c2	d4

TABLE1. Sample relation.

→Observe that **FD A→C** is satisfied. Because, there are two tuples that have an A value as a1 and these tuples have the same C value as c1. Similarly, the other two tuples with an A value of a2 have the same C value c2. But, there are no tuples of **A** attribute having same value in A tuples as well as in corresponding B tuples or D tuples. Thus, the **FD A→B** and **A→D** are not satisfied.
→Moreover, the functional dependency **C→A** is not satisfied. Because, consider the last two tuples **t1 = (a2, b3, c2, d3) and t2 = (a3, b3, c2, d4).** These two tuples have the same C value c2, but they have different A values, a2 and a3, respectively.

Trivial Functional Dependency:
→Some **functional dependencies** are said to be **trivial functional dependencies** because they are satisfied by all relations.

Example: FD A→A is satisfied by all relations involving attribute A. By reading the definition of functional dependency once again, we see that, for all tuples t1 and t2 such that t1 [A] = t2 [A], it is the case that t1 [A] =t2 [A].

Student_Id → Student_Id &
Student_Name → Student_Name are **trivial dependencies**.

3. REASONING ABOUT FD's:
→If a set of FDs are given over a relation R, then several additional FDs satisfies over R, but whenever all of the given set of FDs are satisfied.
Example: Consider the following relation,

Employees (Eno, Ename, Salary, Rating, Did, Since);

With this relation, the given **FDs** are,

1) **FD Eno→Did** should satisfy and
2) **FD Did → Rating** should satisfy.

Therefore, if two tuples have the same **Eno** value, they must have the same **Did** value (from the first FD), and because they have the same **Did** value, they must also have the same **Rating** value (from the second FD).Therefore, the **FD Eno→Rating** also satisfies on Employees.
As a result, we say that **FD** f is implied by a given set F of **FDs**, if f holds on every relation instance that satisfies all dependencies in F.

1. Closure of a Set of FDs:
→The set of all FDs implied by a given set F of FDs is called the closure of F and is denoted as F+.
→The following three rules, called **Armstrong's Axioms,** can be applied repeatedly to compute all FDs implied by a set F of FDs. Here, we use X, Y, and Z to denote sets of attributes over a relation schema R:

Rule 1:
Reflexivity: If $X \supseteq Y$, then $X \rightarrow Y$.
Rule 2:
Augmentation: If $X \rightarrow Y$, then $XZ \rightarrow YZ$ for any Z.
Rule 3:
Transitivity: If $X \rightarrow Y$ and $Y \rightarrow Z$, then $X \rightarrow Z$.
→It is convenient to use some additional rules while reasoning about F^+.
→**Armstrong's axioms** are said to be complete because all the FDs in closure F^+ are computed by the repeated application of all these rules.
→Armstrong's Axioms are 'sound' because they do not generate wrong dependencies

because they generate only ‘dependencies which are in the closure of **F**$^+$
→In addition to the above rules some other rules and also applied.
Rule 4:
Union: If **X** → **Y** and **X** → **Z**, then **X** → **YZ.**
Rule 5:
Decomposition: If **X** → **YZ**, then **X** → **Y** and **X** → **Z.**
Rule 6:
Pseudo-transitivity: If **X → Y** and **Y→P**, then **XZ → P**

Example: Consider the following relation,

Sale (Productid, Date, Customer, Vender, Street)

→Now, the Schema for this Particular relation can be represented as **PDCVS** which implies the sale of product with Product with **Production (P)**, on **Date (D)** to the **Customer (C)** is done by the **Vendor (V)**, who resides in the **Street(S).**

→The following IC (Integrity Constraints) are applied to this relation,

1. **Productid P** is a key. All the attributes are dependent on key.
   P → PDCVS.
2. A **Vendor** Purchases a given **Product** on a single **date**. **VP → D.**
3. The **Customer** purchases almost one **Product** from the **Vendor**. **VC→P.**
4. Every **Vendor** is associated with their own **Street**. **V→S.**
5. Every **Product** belongs to one of the **Vendor**. **P→V**.

→In addition to these FDs, several other Fds also hold in the Closure of given FDs.

1) From **P→V, V→S** and transivity rule, we get **P→S.**
2) From **V→S** and augmentation rule, we get **VP→ PS**.
3) From **VP→ PS, VP → D** and transitivity rule, we get **VP → PDS**.
4) From **VC → P, P→ PDCVS** and transitivity rule, we get, **VC → PDCVS.**

→Decomposition can be applied so as to compute many other FD’s like. From
P→PDCVS and decomposition, we get the following FDs.
P→ D, P→P, P→ C, P→ V, P → S.

2. Closure of Attribute Sets:
→If we want to check whether a given dependency, say, **X** → **Y**, is in the **closure of a set** F of FDs. Then we have to compute the attribute closure X+ with respect to F, which is the set of attributes A such that
X → **A** can be computed using the **Armstrong Axioms**. The algorithm for computing the attribute closure of a set X of attributes is shown in **Figure 15.6.**

```
$closure = X$;
repeat until there is no change: {
        if there is an FD $U \rightarrow V$ in $F$ such that $U \subseteq closure$,
            then set $closure = closure \cup V$
        }
```

Figure 15.6 Computing the Attribute Closure of Attribute Set X

→This algorithm can be modified to find keys by starting with **set X** containing a single

attribute and stopping as soon as closure contains all attributes in the relation schema.
→By varying the starting attribute and the order in which the algorithm considers FDs, we can obtain all candidate keys.

4. NORMAL FORMS: (****************)

→**Normal Form** is a state of a relation that results by decomposing that relation for a good design to avoid redundancy.
→**Normalization** is a process of deciding which attributes should be grouped together in decomposing a given relation into smaller relations.
→**Normalization** is a process of decomposing relations to produce smaller, `well-structured relations.
→**Normalization** is a tool to validate and improve a logical design, so that it satisfies certain constraints that avoid unnecessary duplication of data.

→**Normalization** technique involves a sequence of rules that are employed to test individual relations so that the database can be normalized to any degree. The main objective of normalization is to refine the design of database in order to remove data maintaining anomalies, reduce data redundancy and to eliminate data inconsistency.

→The process of **Normalization** is based on the concept of Normal forms. Each and every normal form has its own set of properties and constraints. A relation is said to be in particular normal form only if it satisfies all the properties of normal form associated with that normal form. These properties are usually applied on the attributes of the relation and also on the relationship that exist between these relations.
Different types of Normal Forms are as follows,

1. **First Normal Form**
2. **Second Normal Form**
3. **Third Normal Form**
4. **Boyce-Codd Normal form**
5. **Fourth Normal Form**

1. First Normal Form (1NF)(**************)

→"Any multi-valued attributes (also called repeating groups) have been removed, so there is a single value at the intersection of each row and column of the table".

Emp_id	Emp_name	Emp_section_id	Emp_address	Dependents
100001	ANUPAMA	401	Rajahmundry	Mother, Father
100002	ABEELA	402	Nandigama	Father, Mother, Sister
100003	RANIA	403	Kovvuru	Brother, Sister
100004	KAVYA	404	Vijayawada	Father, Mother, Sister
100005	NAGINA	405	Kakinada	Mother, Brother

TABLE 11R. An Instance of the Employee relation.

Emp_id	Emp_name	Emp_section_id	Emp_address	Dependents
100001	ANUPAMA	401	Rajahmundry	Mother
100001	ANUPAMA	401	Rajahmundry	Father
100002	ABEELA	402	Nandigama	Father
100002	ABEELA	402	Nandigama	Mother
100002	ABEELA	402	Nandigama	Sister
100003	RANIA	403	Kovvuru	Brother
100003	RANIA	403	Kovvuru	Sister
100004	KAVYA	404	Vijayawada	Father
100004	KAVYA	404	Vijayawada	Mother
100004	KAVYA	404	Vijayawada	Sister
100005	NAGINA	405	Kakinada	Mother
100005	NAGINA	405	Kakinada	Brother

TABLE 22R. An Instance of the Employee in 1NF.

→A relation schema is said to be in first normal form if the attributes values in the relation are atomic, i.e., there should be no repeated values in a particular column. A attribute is said to be value atomic value if it contains only a single, unique value. A relation is said to contain atomic values, if there is an unique value of data item for any given row and column intersection.

Example: Consider an Employee relation with the additional attribute dependents as shown in **Table. 11R**.

→Here, the column dependents have non atomic values. In order to convert this relation into **1NF**, we have to convert these non atomic values to atomic values.

→The **Table. 22R** shows the relation **“Employee”** in **1NF**.

→Now the **relation Employee** is in **1NF** since the column dependents have atomic value. But other attributes i.e., **Emp_id, Emp_section_id, Emp_name, and Emp_address** are all repeating and forming a group called repeated groups. That is, for each value of attribute ‘dependents’ the values are repeating. However, the rule of **1NF** says that any repeated group in a relation must be eliminated as it gives rise to data redundancy. So, in order to delete the repeated groups from the table, the table must be decomposed into other smaller tables by providing a link to the decomposed table (link specifies the parent table which is which is decomposed into child tables).

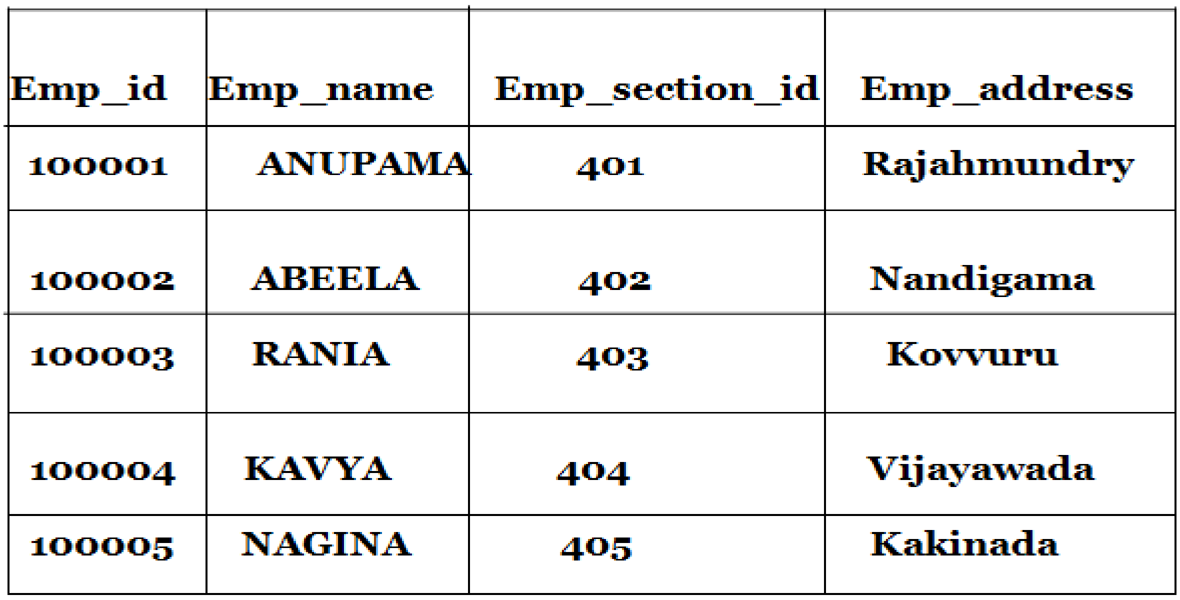
→**For example**, the above relation **“Employee**” can be decomposed into two tables namely,

1. **Emp**
2. **Emp_dependents.**

→Each of these tables do have their own primary keys. For table **“Emp”**
the **primary keys** are **Emp_id** and for table **“Emp_dependents”** primary key is **S.no.** Attribute **‘Emp_id’** is present in both tables which specify the link between two tables and the original table from which the tables are derived.

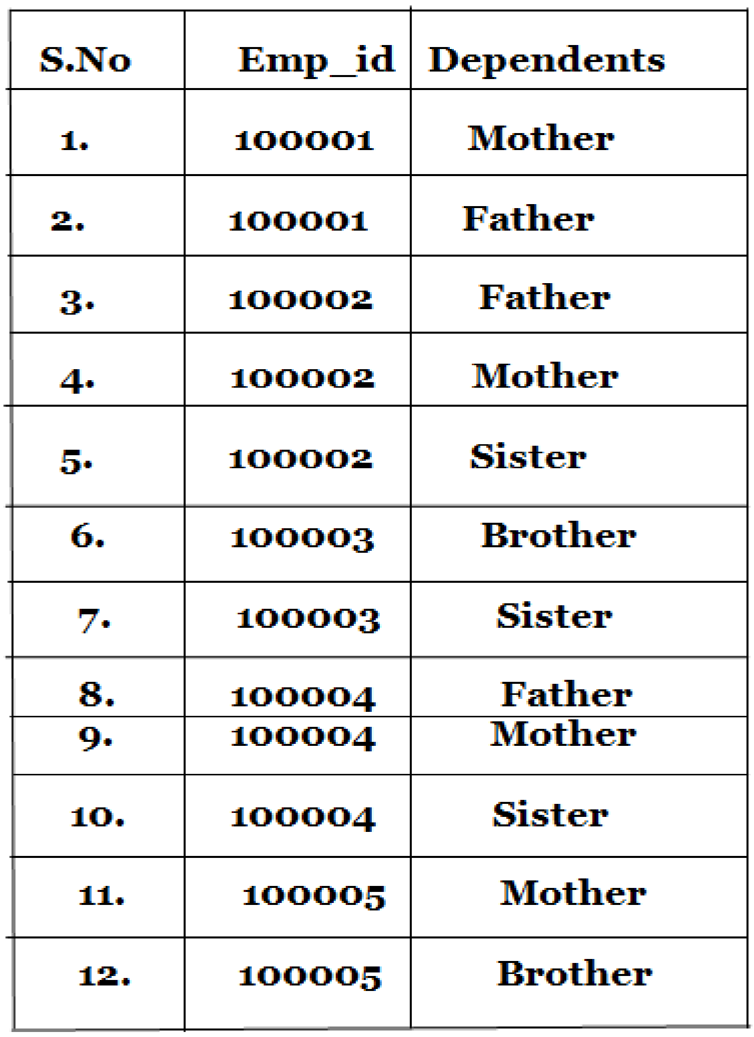
→The most important point to remember is that a relation in database must always be in first normal form.

<u>Drawbacks of 1NF:</u> The main drawback of 1NF is redundancy of data.



Emp_id	Emp_name	Emp_section_id	Emp_address
100001	ANUPAMA	401	Rajahmundry
100002	ABEELA	402	Nandigama
100003	RANIA	403	Kovvuru
100004	KAVYA	404	Vijayawada
100005	NAGINA	405	Kakinada

TABLE 33R. An Instance of the Emp relation.



S.No	Emp_id	Dependents
1.	100001	Mother
2.	100001	Father
3.	100002	Father
4.	100002	Mother
5.	100002	Sister
6.	100003	Brother
7.	100003	Sister
8.	100004	Father
9.	100004	Mother
10.	100004	Sister
11.	100005	Mother
12.	100005	Brother

ΓABLE 44R. An Instance of the Emp_dependents.

2. Second Normal Form (2NF):(***********)

→"Any partial functional dependencies have been removed".

→A relation is said to be **2NF** if it is in **1NF** and every non key attribute is fully functionally dependent on primary key attributes.

→If any one of the following condition is satisfied, then a relation (which is in **1NF**) is in **2NF:**

1). There should be only one attribute associated with the primary key.

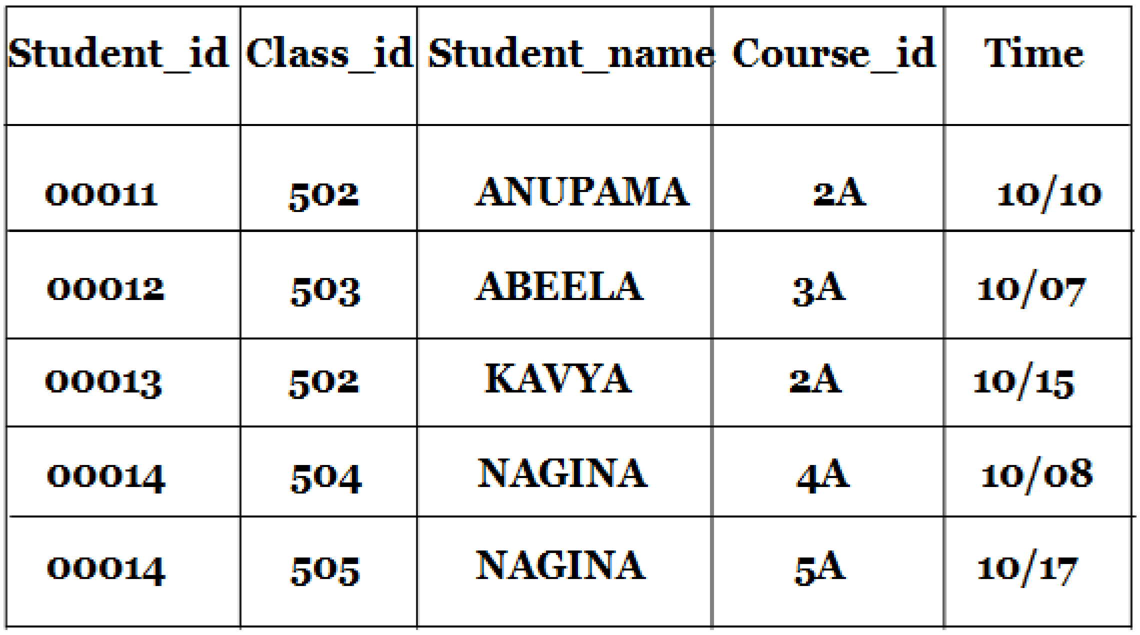
2). There must be no non-key attributes in the relation.

→Non-key attribute must be functionally dependent on the set of primary key attributes i.e., partial dependencies must not exist.

Example: Consider an example of **Student Relation.**

Student (Student_id, Class_id, Student_name, Course_id, Time)

(Student_id, Class-id) is the PRIMARY KEY.



Student_id	Class_id	Student_name	Course_id	Time
00011	502	ANUPAMA	2A	10/10
00012	503	ABEELA	3A	10/07
00013	502	KAVYA	2A	10/15
00014	504	NAGINA	4A	10/08
00014	505	NAGINA	5A	10/17

TABLE 55R. An Instance of the Student relation.

→An instance of **Student Relation** as shown in **Table 55R.**
→A student can attend different course in different classes at different times.

→The above relation is not in 2NF, as the name of the student can be determined by **Student_id.** Therefore a non key attribute (name) is functionally dependent on a part of key (Student_id) i.e., partial dependency exist due to which of the following problems are encountered.
1). Data Redundancy: The name of the student is repeated every time he/she takes a different course due to which loss of data integrity occurs. This is because the relation will show different rows of information for the same student.
2). Update Anomalies: If name of the student is updated, then the entire tuple of the student must be updated. Thus, giving rise due to update anomalies.
3). Insertion Anomalies: This also leads to insertion anomalies because if the student is not attending any classes then there will be no rows in which to keep the student's name.

→Therefore, to solve these problems, the Student relation is broken down into two sub tables (child) both of which are in **2NF.**

1) Student (Student_id, Class_id, Course_id, Time)

2) Student1 (Student_id, Student_name)

Where,
(Student_id, Class_id) is the **COMPOSITE KEY**.

Student_id	Class_id	Course_id	Time
00011	502	2A	10/10
00012	503	3A	10/07
00013	502	2A	10/15
00014	504	4A	10/08
00014	505	5A	10/17

TABLE R1. Student1 Relation which is in 2NF.

These two relations (**TABLE R1** and **TABLE R2**) are called projections of the original relation. A projection in a relation selects certain attributes from the original relation and presents them in a new relation. These two projections are in **2NF** and also solve all the problems listed above.

Student_id	Student_name
00011	ANUPAMA
00012	ABEELA
00013	KAVYA
00014	NAGINA

TABLE R2. Student2 Relation which is in 2NF.

→ The following steps are considered for decomposing a **non-2NF relation** into **2NF relation,**

1) Create a new relation by using the attributes from the offending FD as the attributes of the new relation (i.e., eliminate "Student_name" from the original table).
Student_id → Student_name (Name is now fully functionally dependant on key).
2) Make determinant (left hand side of the equation) i.e., Student_id as the **PRIMARY (determinant) KEY** of the new relation.
3) Delete the attribute on the right hand side (Student_name) from the original relation.
4) Repeat the steps (1, 2, 3) if more than one FD prevents the relation from being in **2NF**.
5) Place all the attributes functionally dependent on the determinant (if it is appearing in more than one FD) as non key attributes in a relation having **determinant key.**

3. Third Normal Form (3NF):(**********)

→"Any transitive dependencies have been removed".
→A relation is said to be **3NF** if it is in **2NF** and does not have transitivity dependencies.
→A relation is said to be in **3NF** if every determinant is a key i.e., for each and every functional dependency **FD: A → B**, A is a Key.
→If any relation is in **3NF**, then the default that relation is in 2NF.
→Consider the same relation "Student" as discussed before but with additional attribute "**Fee**" [For a particular course].

Student (Student_id, Student_name, Course_id, Fee)

Student_id	Student_name	Course_id	Fee
00011	ANUPAMA	2A	5,000
00012	ABEELA	3A	3,500
00013	KAVYA	2A	5,000
00014	NAGINA	4A	4,500

TABLE R11. An Instance of Student relation.

Consider the following **FDs,**

FD : Student_id → Course_id

FD : Student_id → Fee

These two FDs are in **3NF** since both of them satisfies the **3NF** criterion i.e., the determinant should be a PRIMARY KEY. These two FDs are in **3NF**, which also concludes that these FDs are in **2NF.**
Now consider an another FD,

FD : Course_id → Fee

→"**Fee**" attribute is functionally dependent upon the **Course_id** but this Course_id is not **PRIMARY KEY** and hence this FD violates the **3NF** criterion and therefore the relation "Student" is not in **3NF** (because according to the definition of **3NF** for every **FD : A → B** , A should be a **PRIMARY KEY**).

→The following is the list of problems which arises when a relation is not in 3NF form,
1) The attribute "**Fee**" is repeated for every row where the **Course_id** is same. This leads to data redundancy and also wastage of storage space.
2) If the "**Fee**" of the course is updated, then every such row must be updated leading to update anomalies. If we delete the attribute Fee then we may lose the data giving rise to delete anomalies.
3) If there are no **Course_ids** to be entered then there are no rows in which to keep the attribute "**Fee"** leading to insertion anomalies.

→All the problems discussed above are similar to the problems of **2NF**. In order to solve these problems, the relation must be converted to **3NF** form. The following are the steps involved in the process of conversion

Student (Student_id, Student_name, Course_id, Fee)

FD : Student_id → Course_id

FD : Student_id → Fee

FD : Student_id

FD : Course_id → Fee

→The last FD is also called as Transitivity Dependency which occurs when a non key attribute [(**Course_id**) attribute which is not a **PRIMARY KEY**] is functionally dependent upon the other non key attributes **(Fee).**

1) The first step towards the conversion is the removal of the attribute, which is on the right hand side of the FD, violates the conditions of **3NF** i.e., eliminating **"Fee"** attribute from original relation which gives rise to **relation – Student1RR Course_id** is the **FOREIGN KEY** references **Student2RR**.

Student1RR (Student_id, Student_name, Course_id).

2) From another relation, **Student2RR**, which consist of attributes of the following FD.

FD : Course_id → Fee.

The determinant of this FD is the **PRIMARY KEY** of the new relation.

Studen2RR (Course_id , Fee)

Student_id	Student_name	Course_id
00011	ANUPAMA	2A
00012	ABEELA	3A
00013	KAVYA	2A
00014	NAGINA	4A

TABLE. Student1RR Relation which is in 3NF.

Course_id	Fee
2A	5,000
3A	3,500
4A	4,500

TABLE. Student2RR Relation which is in 3NF.

→To Summarize, We Can Say that if a relation is in 3NF then it follows that it is also in 2NF and also in 1NF.

3NF → 2NF → 1NF

4. Boyce-Codd Normal Form (BCNF):(************)

→Any remaining anomalies that result from functional dependencies have been removed.

→ Let **R** be a relation schema, **X** be a subset of the attributes of R, and let A be an attribute of R. R is in **Boyce-Codd normal form** if for every **FD X → A** that holds over R, one of the following statements is true:

A ∈ X; that is, it is a **trivial FD**, or

X is a **superkey.**

→In **Boyce Codd's normal form** the attributes on the left hand side of functional dependency must be a candidate key (A candidate key is a minimal set of attributes whose values uniquely identify an entity in the set, we designate one of them as the primary key. In simple words, two or more primary keys together form a composite key/candidate key).

→According to **Codd's rules**, a relation schema R is in **BCNF** if it is satisfies **3NF.** But, when a relation has more than one candidate key, anomalies may result even though that relation is in **3NF.**

Example: Consider the **Student_advisor** relation, which is in **3NF** but not in **BCNF** as shown in Table.

The **Student_advisor** relation holds following functional dependencies,

FD : (Sid, Major) → Advisor, Major_gpa.....(1) FD

FD : Advisor → Major .... (2) FD

Sid	Major	Advisor	Major_gpa
100001	Electricals	ANUPAMA	9.0
100002	Computers	ABEELA	9.5
100003	Electronics	KAVYA	10.0
100004	Information	NAGINA	9.6

TABLE. The Student_Advisor Relation.

1) **FD** satisfies **BCNF** because the above relation has (Sid, Major) as a **PRIMARY KEY** i.e., the primary key for this relation is composite primary key. Whereas, the Advisor and Major_gpa are functionally dependent on the above composite primary key. The above relation also contains the constraint that a given student can have more than one major, where for each Major a student has exactly one **Advisor** and one **Major_gpa**.

2) FD satisfies **BCNF** because Major is functionally dependent on **Advisor.** That is, each **Advisor** advises exactly one **major.**

Therefore, by observing carefully the following FD's,

FD : (Sid, Major) → Advisor, Major_gpa…..(1) FD

FD : Advisor → Major………….(2) FD

We conclude that, the above relation is in **3NF** but not in **BCNF.**

<u>Converting a Relation to BCNF:</u>

A relation that is in **3NF** (but not in **BCNF**) can be converted to relations in BCNF using a simple two step process as follows,

<u>Step1:</u> The given relation is modified so that the determinant (left hand attribute) in the relation which is not a candidate key becomes a component of the primary key of the new modified relation. The attribute that is functionally dependent on that determinant becomes a non-key attribute. The result is as follows,

FD : (Sid, Advisor) → Major, Major_gpa ……(1) FD

FD : Advisor → Major…….. (2) FD

Observe that the determinant Advisor becomes part of the composite primary key instead of major in (1) FD. And the attribute Major which is functionally dependent on Advisor, becomes a non-key attribute in (2) FD.

<u>Step2:</u> In the above (1) FD and (2) FD, there is a partial dependency and conversion process of step2 is to decompose the relation with new FD's to eliminate the partial functional dependency.

→Thus, the **Student_advisor relation** is decomposed into smaller relations as shown in following two tables.

Sid	Advisor	Major_gpa
100001	ANUPAMA	9.0
100002	ABEELA	9.5
100003	KAVYA	10.0
100004	NAGINA	9.6

TABLE. The Student_Advisor1 Relation.

Advisor	Major
ANUPAMA	Electricals
ABEELA	Computers
KAVYA	Electronics
NAGINA	Information

TABLE. The Student_Advisor2 Relation.

→Where **Sid, Advisor** is the **composite primary key** of **Student_advisor1 relation** and **Advisor** is the **primary key** of **Student_advisor2**.

5. PROPERTIES OF DECOMPOSITIONS:

1) Lossless-Join Decomposition

2) Dependency-Preserving Decomposition

1. Lossless Join decomposition:(*********)

Lossless join decomposition is one of the properties of decompositions. This is dependency also called as **non-additive** or **non-less join dependency**. In a more specific way, lossless join dependency can be defined as the one which generates no additional tuples when the **natural 'join" operation** is performed on the decomposed relation schemas.
Example: Consider the Student Relation.

Student (Sudent_id, Student_name, Location)

This relation can be broken down into two relations as follows,

1) Location (Student_id, Location) and

2) Name (Student_id, Student_name)

→When we perform a natural join operation on these two schemas, the original **"Student"** relation is sustained i.e.,

Location $\bowtie$ Name → Student

Sid	Location	Student_name
100001	Rajahmundry	ANUPAMA
100002	Nandigama	ABEELA
100003	Kovvuru	KAVYA
100004	Vijayawada	NAGINA

TABLE. An instance of Student relation.

Sid	Location	Sid	Student_name
100001	Rajahmundry	100001	ANUPAMA
100002	Nandigama	100002	ABEELA
100003	Kovvuru	100003	KAVYA
100004	Vijayawada	100004	NAGINA

TABLE. Combination of Location and Name Relation.

Location $\bowtie$ Name → Student

Sid	Location	Student_name
100001	Rajahmundry	ANUPAMA
100002	Nandigama	ABEELA
100003	Kovvuru	KAVYA
100004	Vijayawada	NAGINA

TABLE. After performing Join operation Location and Name Relation.

→No additional tuples are generated and neither data is duplicated nor is data lost. Therefore the relation Student is lossless.

2. Dependency Preserving Decomposition:

→R is a relational schema that is decomposed into Schemes **R_1, R_2 …., R_n** with Attributes, **A, B, C….**by applying all the steps of normalization. Let F be the set of functional dependencies that hold over **R and F_1, F_2,** F_3…..be the set of functional dependencies that hold over **R_1, R_2 …., R_n** respectively. Fi refers to the attributes of **R_i** where **i = 1, 2, 3…..**

→If R is decomposed into two relations **R_1 and R2** with an attribute set of S1 and S2. Then the projection of **R on R_1** can be defined as a set of functional dependencies in the closure of **F^+** consisting of the attributes present in **R_1**. The notation for representation of a Projection of **F on S1 in F_{s1}.** Similarly, the projection of **F on S2 in F_{s2}.**

→Now , the relation R with a set of FDs F is decomposed into two relations **R_1 and R_2** with an attribute set of **S1 and S2** is said to be dependency Preserving if, **$(F_{s1} \cup F_{s2})^+ = F^+$.**

→The union of Closure Set of two projections must be equal to the Closure Set of dependencies of the original relation.

→In other words, it can be said that relation R is decomposed into projections **R_1, R_2,, R_3** in such a way that the enforcement of constraint to set **of F_1, F_2, ….., F_i**
is altogether equal to enforcing the constraint on the original set F. Thus, the decomposition is said to be "**dependency preserving".**

EXAMPLE: Consider an example where the relation Z is decomposed into relations. The attribute set of Z consist of PRQ and the attribute set of decomposed relations consist of PQ and QR. The set of FDs that hold over Z includes **P → Q, Q → R** and **R → P.**
F_{PQ} - Set of attributes of the relation contains **P → Q** and **Q → P.**
F_{QR} - Contains **Q → R** and **R → Q.**

Now, the original set of functional dependencies consist of,

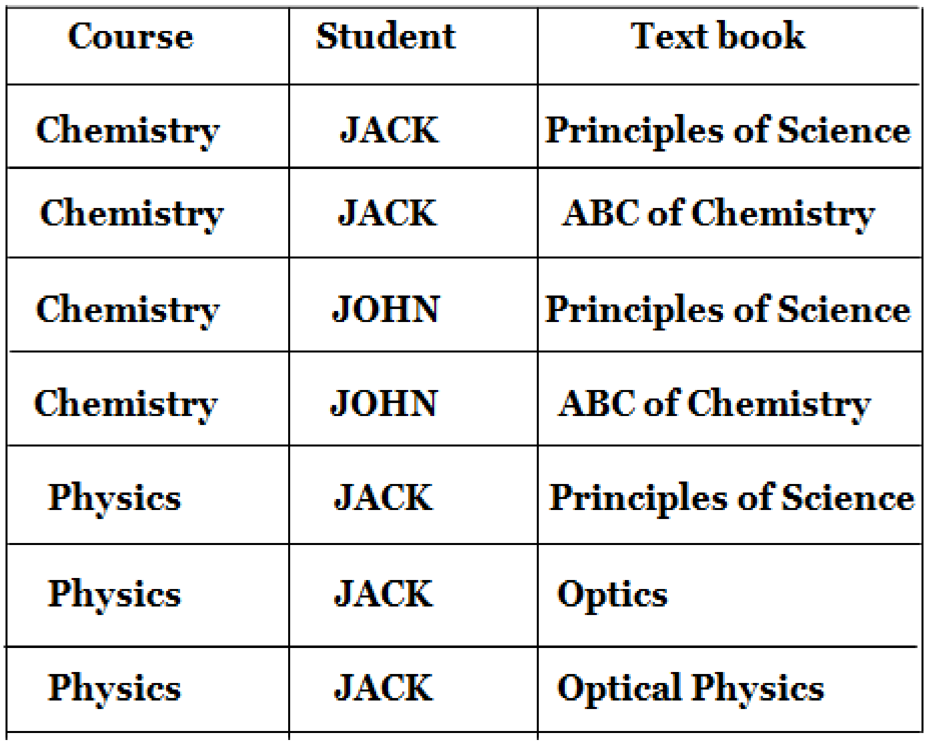
1) **F**
2) **P → Q**
3) **Q → R**
4) **R → P**

→Now, the union of closure of $\mathbf{F_{PQ}}$ and $\mathbf{F_{QR}}$ includes $\mathbf{F_{PQ}}$ **U** $\mathbf{F_{QR}}$
P → Q, Q → P, Q → R, R → Q
With **R → Q, Q → P** and **Transitivity**
We get, **R → P**
Thus, the decomposition is said to preserve the dependency.

6. OTHER KINDS OF DEPENDENCIES:

1. Multivalued Dependencies (**************)

To understand the concept of multivalued dependencies, consider the following relation.



Course	Student	Text book
Chemistry	JACK	Principles of Science
Chemistry	JACK	ABC of Chemistry
Chemistry	JOHN	Principles of Science
Chemistry	JOHN	ABC of Chemistry
Physics	JACK	Principles of Science
Physics	JACK	Optics
Physics	JACK	Optical Physics

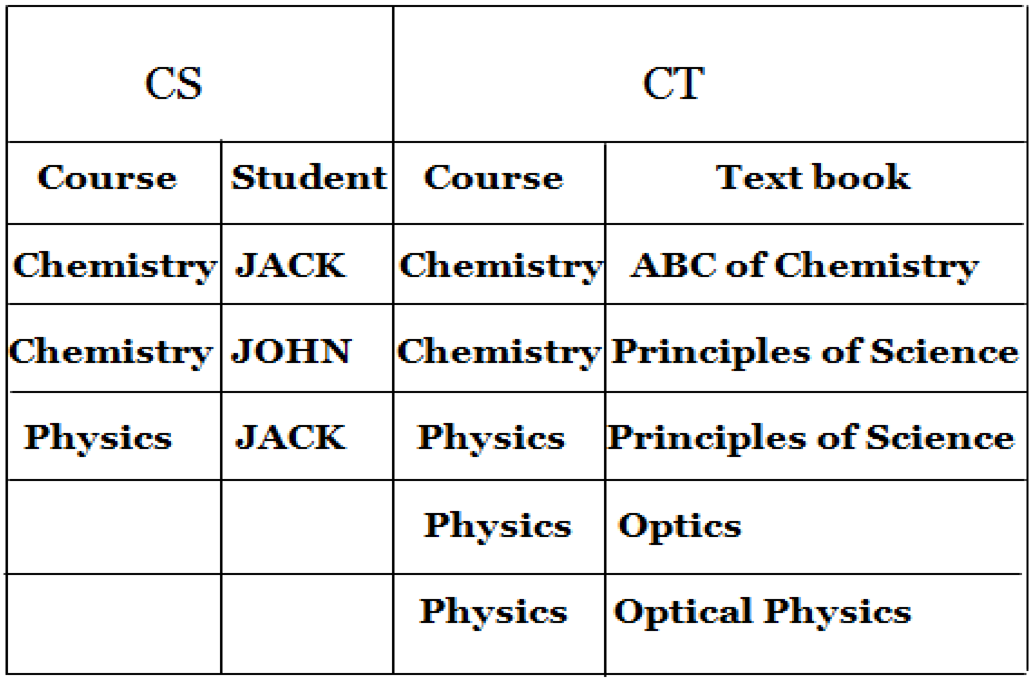
TABLE. BCNF Relation with Redundancy.

→Here, each of the tuple means that the Course ‘C’ is taken by the Student ‘S’ and the text book ‘T’ is the one which is recommended. The attributes Student and Text book are independent of each other. Any number of students can refer any Text book and take any Course. The composite key for this relation consist of (CST).

→Since all the attributes are part of key, this relation is in BCNF and therefore, there is no use of decomposing it further. We can also notice that much of the data is being repeated. That is, the Text book for chemistry is ABC of chemistry is repeated for each student.

→This redundancy again gives rise to update anomalies. By decomposing this relation into two schemas with attributes CS and CT, we can deal with redundancy.

→It is worth noticing that the redundancy is due to the fact that Students and Text books are independent of each other. Such type of constraint is an example of multivalued dependency of MVD.



CS		CT	
Course	Student	Course	Text book
Chemistry	JACK	Chemistry	ABC of Chemistry
Chemistry	JOHN	Chemistry	Principles of Science
Physics	JACK	Physics	Principles of Science
		Physics	Optics
		Physics	Optical Physics

TABLE. Decomposing CST Relation as CS and CT.

→By decomposing the relation we can eliminate the update anomalies. For example, if we want to enter the data that a new Student is taking Chemistry Course then we needs to enter a single tuple in the relation CS.

MVDs are generalization of functional dependencies. They can be represented as,
Course→ → Student
Course→ → Text book
This read as "Student is multi dependent on course" or "Course multi determines Text book".
The meaning of MVD **Course→ → Student** is,

There exists a set of Students corresponding to each Course C i.e., for a Course C and a Text book B, the set of Students matching the pair (C, B) in CST depends on the value of C only, it is independent of the value B. A multivalued dependency can be defined as, if X, Y, Z are the attribute sub sets of attribute set of relation A then Y is said to be multi dependent on X if for every instance of A, a set of Y values matching a given pair (X value, Z value) depends only on the X value and does not depend on Z value.
X→ → Y

→Every **FD** is an **MVD**,
i.e., **X → → Y** this means **X → → Y**
Five rules are used to compute additional **FDs** and **MVDs.** They are

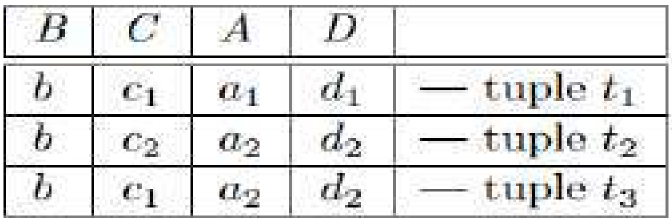
1) **MVD Complementation**: If **A → → B**, then **A→ → R - AB.**
2) **MVD Augmentation**: If **A → → B** and **C** ⊆ **D** then **AD → → BC**.
3) **MVD Transitivity**: If **A→ → B** and **B→→C** then **A→ → (C-B)**
4) **Replication:** If **A→→B**, then **A→→B**
5) **Coalescence**: If **A→ → B** and there is a **C** such than **C** ∩ **B** is empty.
C→D and **D** ⊆ **B**, then **A→D.**

<u>Fourth Normal Form:</u>
Fourth Normal form is a direct generalization of BCNF. Let R be a relation schema, A and B be attributes of R, and F be a set of dependencies that includes both FDs and MVDs. Then R

is said to be in Fourth Normal Form (4NF) if for every **MVD A→ → B** that holds over R, one of the following statements is true,

1) **B ⊆ A or AB = R, or**
2) **A is a super key.**

<u>Example</u>: Consider a relation schema ABCD and suppose that the FD A→BCD and the MVD B→→C are given as shown in Figure 15.5



B	C	A	D	
b	c_1	a_1	d_1	— tuple t_1
b	c_2	a_2	d_2	— tuple t_2
b	c_1	a_2	d_2	— tuple t_3

Figure 15.15 Three Tuples from a Legal Instance of *ABCD*

→**Figure 15.15** shows three tuples from relation ABCD that satisfies the given MVD B →→ C. From the definition of an MVD, given tuples t1 and t2, it follows that tuple t3 must also be included in the above relation. Now, consider tuples t2 and t3. From the given FD A → BCD and the fact that these tuples have the same A-value, we can compute that c1 = c2. Therefore, we see that the FD B → C must hold over ABCD whenever the FD A → BCD and the MVD B →→ C holds. If B → C holds, the relation ABCD is not in BCNF but the relation is in4NF.

<u>Uses of 4NF:</u> The Fourth normal form is useful because it overcomes the problems of various approaches in which it represents the multivalued attributes in a single relation.

<u>Join Dependencies :(************)</u>

A join dependency refers to a constraint that is provided on a group of relations over a database schema.

A join dependency is a further generalization of MVDs. A join dependency (JD) ⋈ **{R1, ... , R_n}** is said to hold over a relation **R if R1, ... , R_n** is a lossless-join decomposition of R. An MVD X →→ Y over a relation R can be expressed as the join dependency⋈ **{XY, X (R−Y)}.** As an example, in the CTB relation, the **MVD C →→ T** can be expressed as the **join dependency⋈ {CT, CB}.**

Example: Consider a schema table called Project with the three attributes i.e., **Studentid, Studentname, Projectname and Projecttype.** With this information, it can be clearly noted that the following relations can be derived.

1) **Studentname depends on Studentid**
2) **Projectname depends on Studentid**
3) **Projectname depends on Studentid.**

Thus, these three relations can be expressed by using a join dependency relationship, which is as follows,

***((Studentid, Studentname), (Studentid, Projectname), (Studentid, Projecttype))**

If every Student does a different Project i.e., every Student deals with a different Projecttype then there exists a Join dependency, which is as follows,

***((Studentid, Studentname), (Studentid, Projecttype), (Studentname, Projecttype), (Studentid, Projectname))**

(OR)

***((Studentid, Studentname, Projecttype), (Studentid, Projectname)).**

<u>Fifth Normal Form:</u>

A relation schema R is said to be in **fifth normal form (5NF)** if for every **JD ⋈{R1, ... , Rn**} that holds over R, one of the following statements is true: **Ri = R** for some i, or The JD is implied by the set of those FDs over R in which the left side is a key for R.

→Intuitively, we must be able to show that the decomposition of R into **{R1, ... , Rn}** is lossless-join whenever the key dependencies (FDs in which the left side is a key for R) hold.

⋈ {R1, ... , Rn} is a **trivial JD** if **Ri = R** for some i; such a JD always holds.

The following result helps us to safely ignore join dependency information.

"If a relation schema is in 3NF and each of its keys consists of a single attribute, it is also in 5NF".

The conditions identified in this result are sufficient for a relation to be in 5NF, but not necessary.

Chapter-6

6. TRANSACTION MANAGEMENT

1. TRANSACTION CONCEPT:

→A **Transaction** is a set of changes that must all be made together. It is a collection of operations that form a single logical unit of work. It must be either completed or entirely apported to ensure the consistency and integrity of database.
→A **Transaction** is an execution of a user program and is seen by the DBMS as a series or list of actions i.e., the actions that can be executed by a transaction includes the read and write operations of database.

For Example - A transfer of money from one bank account to another requires two changes to the database both must succeed or fail together.
Consider the example - You are working on a system for a bank. A customer goes to the ATM and instructs it to transfer RS 1000 from saving to a checking account. This simple transaction requires 2 steps.

1). Subtracting the money from the savings account balance.
2). Adding the money to the checking account balance.

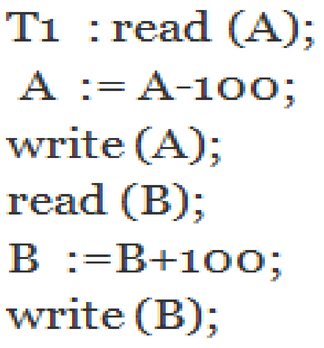
The code to create this transaction will require two updates to the database. There will be two SQL statements-one UPDATE command to decrease the balance in savings and a second UPDATE command to increase the balance in the checking account. Both changes must be made successfully. Thus a transaction is defined as a set of changes that must be made together.

PROCESS OF TRANSACTION:

→The transaction is executed as a series of reads and writes of database objects.

1). READ OPERATION: To read a database object, it is first brought into main memory from disk and then its value is copied into a program variable.

2).WRITE OPERATION: To write a database object, memory copy of the object is first modified and then written to disk.
Example: Let T1 be a transaction that transfers $100 from account A to account B. This transaction can be illustrated as follows,



```
T1  : read (A);
 A  := A-100;
write (A);
read (B);
B  :=B+100;
write (B);
```

PROPERTIES OF TRANSACTION (ACID PROPERTIES):

(*************************************)

→**ACID** (**Atomicity, Consistency, Isolation,** and **Durability**) is a set of properties that guarantee that database transactions are processed reliably.

→ There are four important properties of transactions that a DBMS must ensure to maintain data in the face of concurrent access and system failures:

A - Atomicity
C - Consistency
I - Isolation
D - Durability

1. Atomicity-(all or nothing):

→Users should be able to regard the execution of each transaction as **atomic:** either all actions are carried out or none are. Users should not have to worry about the effect of incomplete transactions (say, when a system crash occurs).

→A transaction is said to be **atomic** if a transaction always executes all its action in one step or not executes any action at all.

Example: Suppose **Anupama Parameswaran** had Rs50000 in her account and **Red Queen** has Rs20000 in her account. Now Anupama transfers a amount of Rs5000 to **Red Queen.** A transaction debits the amount from **Anupama's** account, but before it could be credited to **Red Queen**, if there is a failure then transaction would result in loss of Rs5000.

→Because, the amount is deducted from **Anupama's** account but it is not added to **Red Queen's** account. This leaves the data in an inconsistent state.

→If there is failure during transaction execution, then measures taken to get back the data in a form which was in, before transaction. This is taken care of by management component.

2. Consistency (No violation of Integrity Constraints):

→Each transaction, run by itself with no concurrent execution of other transactions, must preserve the consistency of the database. This property is called **consistency,** and the DBMS assumes that it holds for each transaction. Ensuring this property of a transaction is the responsibility of the user.

→A transaction must preserve the **consistency** of a database.

Example: Consider a transaction that involves transfer of amount. If amount is debited from account **'Anupama'** and credited to account **'Red Queen'**, after the transaction the sum **Anupama + Red Queen** should be the same as it was before transaction. However at an intermediate stage, where the amount is deducted from **Anupama** but not yet credited to **Red Queen**, the sum **Anupama + Red Queen** would not be same and it need not be.

→It is the responsibility of application program to ensure consistency.

3. Isolation-(concurrent changes invisibles):

→Users should be able to understand a transaction without considering the effect of other concurrently executing transactions, even if the DBMS interleaves the actions of several transactions for performance reasons. This property is sometimes referred to as isolation: Transactions are **isolated**, or **protected**, from the effects of concurrently scheduling other transactions.

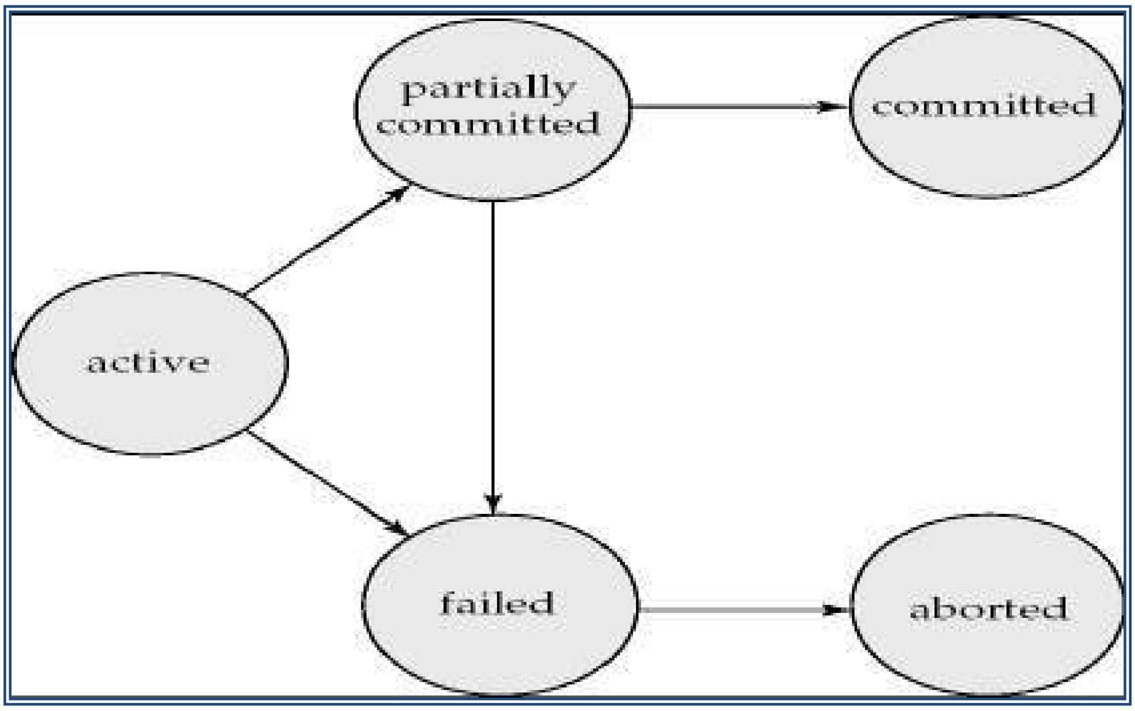
→The transaction must behave as if they are executed in **isolation**. In other words if transaction are executed concurrently the result must be same.

T1 Subtracts 500 from Anupama
T1 Adds 100 to Red Queen
T2 Subtracts 500 from Red Queen
T2 Adds 100 to Anupama

→If these operations are performed in order, isolation is maintained otherwise there will be an error.

4. Durability-(committed update persist):
→Once the DBMS informs the user that a transaction has been successfully completed, its effects should persist even if the system crashes before all its changes are reflected on disk. This property is called **durability.**
→The effect of complete or committed transaction should persist even after a crash.
→The recovery-management component of database systems ensures the durability of transaction.

STATES OF TRANSACTION:



→A transaction must be in one of the following states.
1. Active-In this state, the transaction is being executed. This is the initial state of every transaction.
2. Partially committed- When a transaction executes its final operation, it is said to be in a partially committed state.
3. Failed-A transaction is said to be in a failed state if any of the checks made by the database recovery system fails. A failed transaction can no longer proceed further.
4. Aborted-If any of the checks fails and the transaction has reached a failed state, then the recovery manager rolls back all its write operations on the database to bring the database back to its original state where it was prior to the execution of the transaction. Transactions in this state are called aborted. The database recovery module can select one of the two operations after a transaction aborts

- **Re-start** the transaction
- **Kill the transaction**

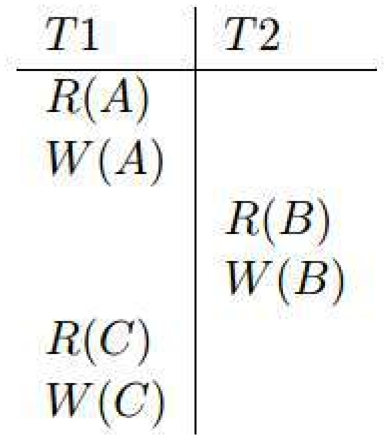
5. Committed-If a transaction executes all its operations successfully, it is said to be committed. All its effects are now permanently established on the database system.

2. TRANSACTIONS AND SCHEDULES:

→A **transaction** is seen by the DBMS as a series, or list, of actions. The actions that can be executed by a transaction include reads and writes of database objects. A transaction can also be defined as a set of actions that are partially ordered. That is, the relative order of some of the

actions may not be important. In order to concentrate on the main issues, we will treat transactions (and later, schedules) as a list of actions.
→A **schedule** is a list of actions (**reading, writing, aborting, or committing**) from a set of transactions, and the order in which two actions of a transaction T appear in a schedule must be the same as the order in which they appear in T. In short, <u>a schedule represents an actual or potential execution sequence.</u>

→Consider the schedule of an execution order for actions of two transactions T1 and T2 as shown in **Figure 18.1**.

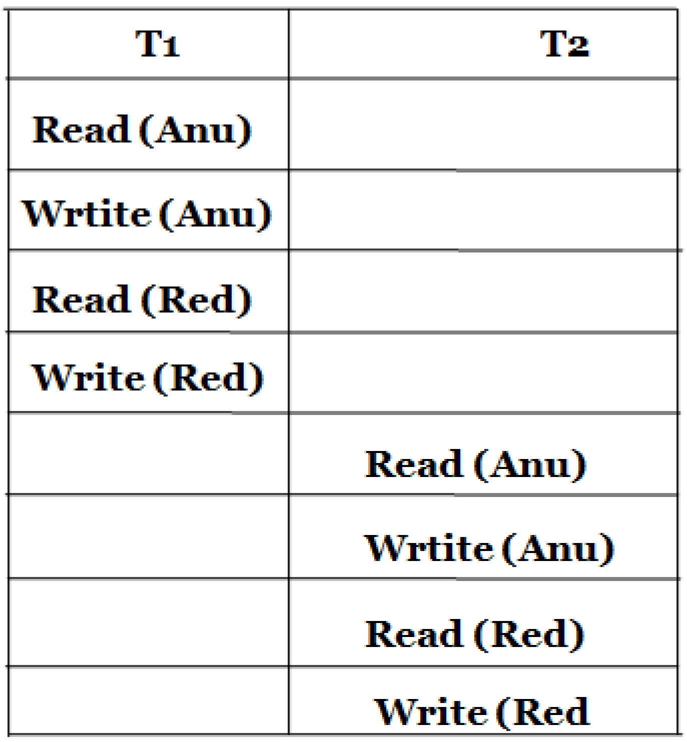


$T1$	$T2$
$R(A)$	
$W(A)$	
	$R(B)$
	$W(B)$
$R(C)$	
$W(C)$	

Figure 18.1 A Schedule Involving Two Transactions

→ We must forward in a schedule as row-wise, i.e., from one row to the next row and so on. Thus, a schedule describes the actions of transactions performed by the DBMS. In addition to these actions, a transaction may carry out other actions, such as reading or writing from operating system files, evaluating arithmetic expressions, and so on. Note that the schedule in **Figure 18.1** does not contain an abort commit action for either transaction because it is not a complete schedule.

→A schedule that contains either an abort or a commit for each transaction whose actions are listed in it is called a **complete schedule**. A **complete schedule** must contain all the actions of every transaction that appears in it.
<u>**Serial Schedule:**</u>
→**Serial Schedule** is a Schedule wherein the transactions are executed one after another sequentially. All the instructions belonging to a transaction appears together in Serial Schedule. The number of Serial Schedules generated for a given Schedule depends on the number of transactions (i.e., if there are K-transactions, then K! serial schedules are generated).



T1	T2
Read (Anu)	
Wrtite (Anu)	
Read (Red)	
Write (Red)	
	Read (Anu)
	Wrtite (Anu)
	Read (Red)
	Write (Red

TABLE. Serial schedule

<u>Non-Serial Schedule:</u>

→If multiple transactions are executed concurrently, then the schedule is not a **Serial Schedule**. In concurrent execution, the operating system initially executes few instructions of first transaction and then CPU performs Context Switching and executes the instructions of second transaction. Later it switches back to first transaction and instructions of second transaction. Later it switches back to first transaction and executes the remaining instructions and so on. If several transactions are executed concurrently, then CPU time is shared synchronously between all these transactions.

→In concurrent execution, transactions may be interleaved. Due to this, there is a possibility that more than one execution sequence may exist. However, it is not possible to know the number of instructions that are executed in a transaction before CPU switches to another transaction.

→It is not always true that concurrent execution leads to consistent state i.e., they may be schedules that may leads to incorrect result.

→Example of one such Schedule is given in a table.

T1	T2
Read (Anu)	
Anu: = Anu-200	
	Read (Anu)
	Anu: = Anu – (Anu*20/100)
	Write (Anu)
	Read (Red)
Write (Anu)	
Read (Red)	
Read: = Red+200	
Write (Red)	
	Red: =Red*(Red*20/100)
	Write (Red)

TABLE. Serial S1

→Since, the execution of above Schedule results in incorrect state, therefore the sum of both accounts is not stored.

3. CONCURRENT EXECUTION OF TRANSACTIONS(**)

→The DBMS interleaves the actions of different transactions to improve performance, in terms of increased throughput or improved response times for short transactions, but not all interleaving should be allowed.

1. Motivation for Concurrent Execution:

→The schedule shown in **Figure 18.1** represents an interleaved execution of the two transactions. Ensuring transaction isolation while permitting such concurrent execution is difficult, but is necessary for performance reasons.

Advantages of Concurrent Execution of Transaction:

The DBMS interleaves the actions of different transactions to improve performance of system as discussed below:

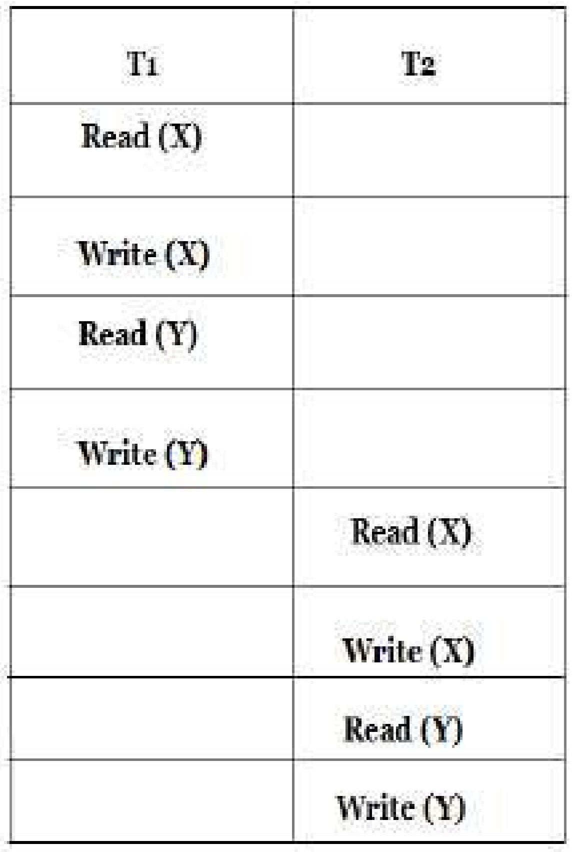
<u>1). Improved Throughput</u>: Consider that transaction are performed in serial order and active transaction is waiting for a page to be read in from disk, then instead of CPU waiting for a page, it can process another transaction. This is because Input/output activity can be done in parallel with the CPU activity. The overlapping of Input/output activities of CPU reduces the amount of time disks and processors are idle and increases system throughput (the average number of transaction completed in a given time.)

<u>2). Reduced Waiting time</u>: Interleaved execution of a short transaction with a long transaction usually allows the short transaction to complete quickly. In serial execution a short transaction could get stuck behind a long transaction leading to unpredictable delays in response time or average time taken to complete a transaction.

<u>Serializability:</u>

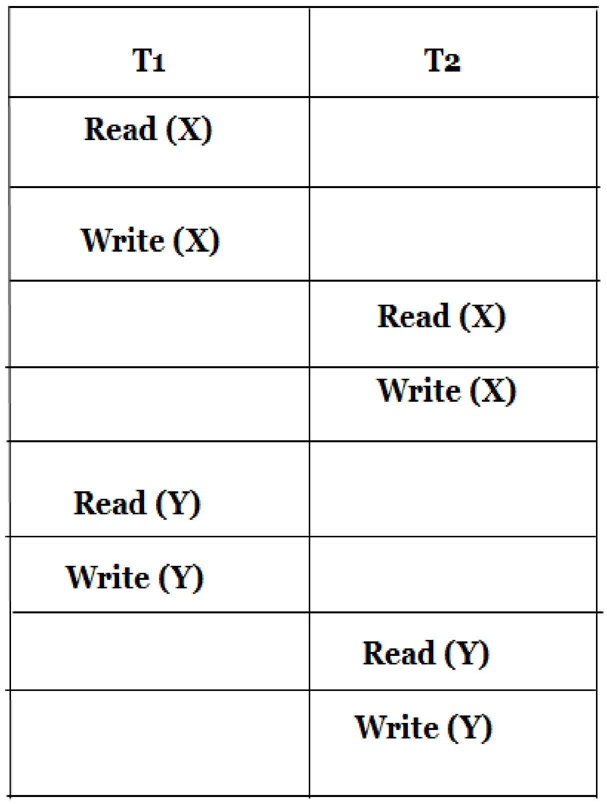
→A **Schedule** 'S' of 'n' transactions is **serializable** if it is equivalent to some serial schedule of the same 'n' transactions. A **serializable schedule** contains the correctness of serial schedule while ascertaining better CPU utilization of parallel schedule.

→**Serializability** is a widely accepted standard that ensures the consistency of a schedule. A schedule is consistent if and only if it is serializable. A schedule is said to be serializable if the interleaved transactions produces the result, which is equivalent to the result produced by executing individual transactions separately (i.e., a schedule is serializable, if it produces the same result as that of a serial schedule).



T1	T2
Read (X)	
Write (X)	
Read (Y)	
Write (Y)	
	Read (X)
	Write (X)
	Read (Y)
	Write (Y)

Table. Serial Schedule



T1	T2
Read (X)	
Write (X)	
	Read (X)
	Write (X)
Read (Y)	
Write (Y)	
	Read (Y)
	Write (Y)

Table. Schedule for the two transactions when interleaved.

→The above two schedules produce the same result, these schedules are said to be serializable. The transaction may be interleaved in any order and DBMS doesn't provide any guarantee about the order in which they are executed.
→The two different types of serializability are

1) **Conflict Serializability**
2) **View Serializability**

1. Conflict Serializability:

→Any given concurrent schedule is said to be Conflict serializable if and only if it is **CONFLICT EQUALENT** to one of the possible **serial schedule.**

→Consider a schedule S1, consisting of two successive instructions IA and IB belonging to transactions TA and TB refer to different data items then it is very easy to swap these instructions.

→The result of swapping these instructions doesn't have any impact on the remaining instructions in the schedule. If IA and IB refers to same data item then the following four cases must be considered,

Case 1 : $I_A = read(x)$, $I_B = read(x)$,
Case 2 : $I_A = read(x)$, $I_B = write(x)$,
Case 3 : $I_A = write(x)$, $I_B = read(x)$,
Case 4 : $I_A = write(x)$, $I_B = write(x)$,

Case 1: Here, both I_A and I_B are read instructions. In this case, the execution order of the instructions is not considered since the same data item x is read by both the transactions T_A and T_B.

Case 2: Here, I_A and I_B are read and write instructions respectively. If the execution order of instructions is $I_A \rightarrow I_B$, then transaction T_A cannot read the value written by transaction TB in instruction IB. but order is $I_B \rightarrow I_A$, then transaction T_A can read the value written by transaction T_B. Therefore in this case, the execution order of the instructions is important.

Case 3: Here, I_A and I_B are write and read instructions respectively. If the execution order of instructions is $I_A \rightarrow I_B$, then transaction T_B can read the value written by transaction T_A, but order is $I_B \rightarrow I_A$, then transaction T_B cannot read the value written by transaction T_B. Therefore in this case, the execution order of the instructions is important.

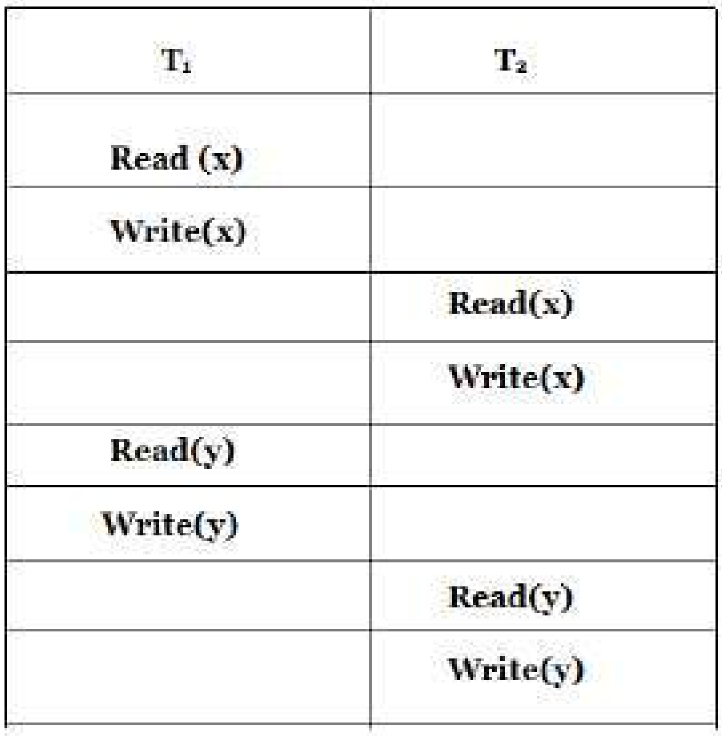
Case 4: Here, both I_A and I_B are write instructions. In this case, the execution order of the instructions doesn't matter. If a read operation is performed before the write operation, then the data item which was already stored in the database is read.

→Two instructions **I_A** and **I_B** are said to be **conflicting** if and only if,

1) They represent the operations performed by two different transactions on the same data item.
2) Atleast one (among **I_A** and **I_B**) is a write operation.

→Let us consider the following **Schedule S_2**.

Here, the write(x) operation of T_1 conflicts with read(x) operation of T_2. But the write(x) operation of T_2 doesn't conflict with read(y) operation of T_2, since these instructions perform their operations on two different data items.



T_1	T_2
Read (x)	
Write(x)	
	Read(x)
	Write(x)
Read(y)	
Write(y)	
	Read(y)
	Write(y)

Table. Schedule S_2.

<u>CONFLICT EQUIVALENT:</u>
The order of the instructions I_A and I_B can be swapped when,

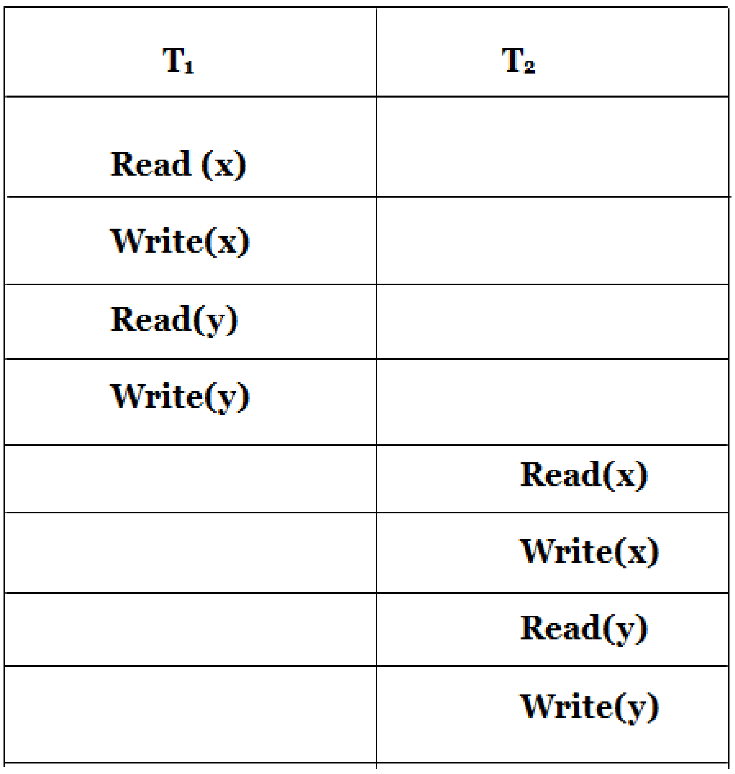
1) $\mathbf{I_A}$ and $\mathbf{I_B}$ belong to two different transactions.
2) $\mathbf{I_A}$ and $\mathbf{I_B}$ are not conflicting instructions.

→The execution order of the instruction in **Schedule S_2** are swapped, so as to generate a new **Schedule S_2^1** which is equivalent **Schedule S_2**. **S_2** is said to be equivalent to **S_2^1** because the order of executing the instructions in **S_2^1** is similar to the execution order in **S_2** except for instructions **I_A** and **I_B** **(i.e., the** order of **I_A** and **I_B** is not considered.

→In the above **Schedule S_2** write(x) instruction of T_2 can be swapped with read(y) instruction of T_1 (since both instructions are not conflict). This **Schedule S_2** can be transferred into **Schedule S_2^1** by swapping the instructions of T_1 and T_1 in the following manner:

1) Read(y) instruction of T_1 doesn't conflict with read(x) instruction of T_2. Therefore these instructions can be swapped so as to generate new **Schedule S_2^1.**
2) Write(y) instruction of T_1 can be swapped with write(x) instruction of T_2. Since, these are non-conflicting instructions.
3) Write(y) instruction of T_1 can further be swapped with read(x) instruction of T_2.

→<u>After performing the swapping, a **Schedule S_2^1** is produced which is a **Serial Schedule.**</u>



T_1	T_2
Read (x)	
Write(x)	
Read(y)	
Write(y)	
	Read(x)
	Write(x)
	Read(y)
	Write(y)

Table. Schedule S_2^1

Since **Schedule S_2** is equivalent to **Schedule S_2^1**, therefore **S_2** and **S_2^1** are said to be conflict equivalent.

→ The conflict equivalence leads to another notion called conflict serializability. A schedule say S2 is said to be Conflict Serializable, if it is conflict equivalent with the serial schedule.

→In the above example, **Schedule S_2** is **Conflict Serializable**, as it is **Conflict Equivalence** with **Serial Schedule**.

2. View Serializability:

→Any given concurrent schedule is said to be View serializable if and only if it is **VIEW EQUALENT** to one of the possible **serial schedule.**

→Two schedules S_1 and S_1^1 consisting of some set of transactions are said to be view equivalent, if the following conditions are satisfied,

1) If a transaction T_A in schedule S_1 performs the read operation on the initial value of data item x, then the same transaction in schedule S_1^1 must also perform the read operation on the initial value of x.

2) If a transaction T_A in schedule S1reads the value x, which was written by transaction T_B, then T_A in schedule S_1^1must also perform the read the value x written by transaction T_B.

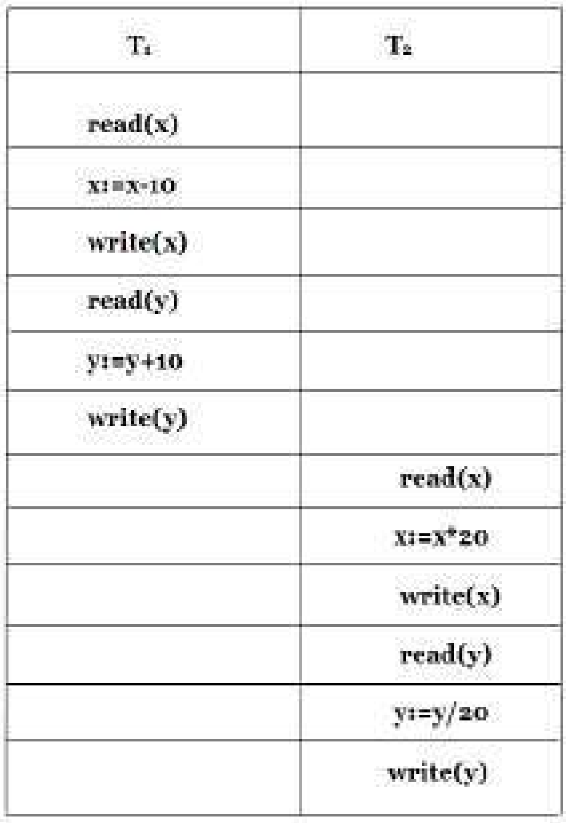
3) If a transaction T_A in schedule S1performs the final write operation on data item x, then the same transaction in schedule S_1^1 must also perform the final write operation on x.

→Let us consider the following schedules that are view equivalent.

→**Schedule S_4** is **view equivalent** to **Schedule S_5** since the value of x and y read by transaction T_2, is generated by T_1 in both **S_4** and **S_5**

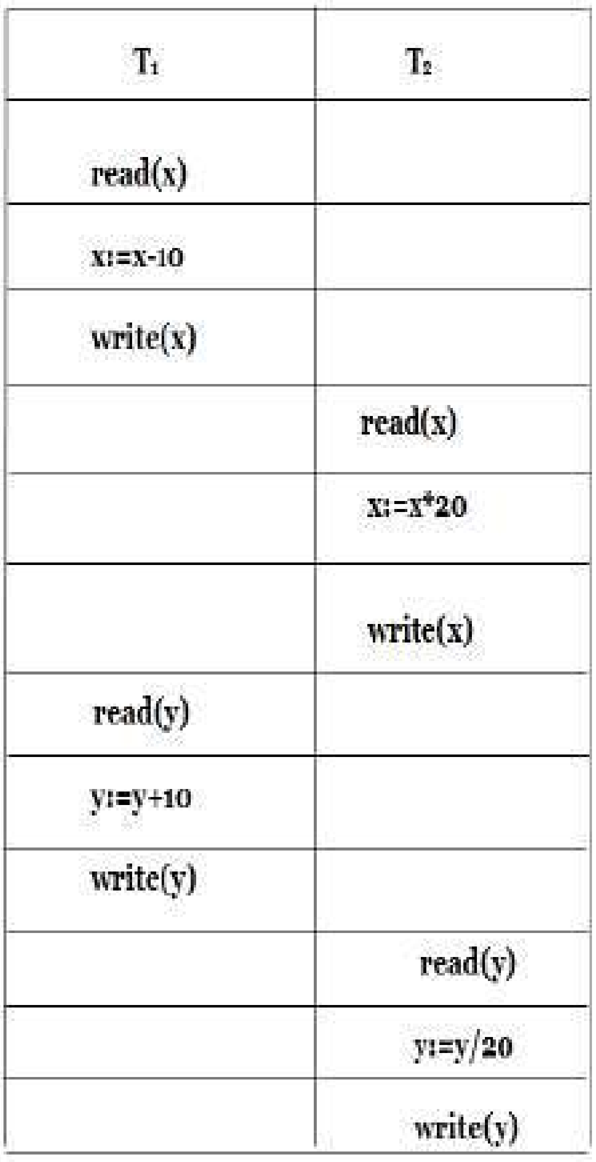
→The view equivalence leads to another notion called view serializability. A schedule say S is said to be view Serializable, if it is view equivalent with the serial schedule.

→Every conflict Serializable schedule is view Serializable but every view Serializable is not conflict Serializable.



T_1	T_2
read(x)	
x:=x-10	
write(x)	
read(y)	
y:=y+10	
write(y)	
	read(x)
	x:=x*20
	write(x)
	read(y)
	y:=y/20
	write(y)

Table. Schedule S_4



T_1	T_2
read(x)	
x:=x-10	
write(x)	
	read(x)
	x:=x*20
	write(x)
read(y)	
y:=y+10	
write(y)	
	read(y)
	y:=y/20
	write(y)

Table. Schedule S_5

3. Anomalies Due to Interleaved Execution (***********)

→The Schedule, involving two transactions shown in the **Table R1** represents an interleaved execution of the two transactions.

1) While one transaction is waiting for a page to be read from disk, the CPU can process another transaction. This is because I/O activity can be done in parallel with CPU activity in a

computer. Overlapping I/O and CPU activity reduces the amount of tome and increases system throughput which is the average number of transactions completed in a given time.
2) Interleaved execution of a short transaction with a long transaction usually allows the short transaction to complete quickly. In Serial execution, a short transaction could get stuck behind a long transaction, leading to unpredictable delays in response time or average time taken to complete a transaction.

T_1	T_2
read(A)	
write(A)	
	read(B)
	write(B)
read(C)	
write(C)	

Table. Interleaved Execution of Two Transactions.

→There are three main situations when the actions of two transactions T1 and T2 conflict with each other in the interleaved execution on the same data object.

→The three anomalies associated with interleaved execution are as follows,

1) Write-Read (WR) Conflict: Reading Uncommitted Data.

2) Read-Write (RW) Conflict: Unrepeatable Reads.

3) Write-Write (WW) Conflict: Overwriting Uncommitted Data.

Reading Uncommitted Data (WR Conflicts):

→The first Source of anomalies is that a transaction **T_2** could read a database object A has been just modified by another transaction **T_1**, which has not yet committed; such a read is called a **dirty read** or **reading uncommitted data.**

T_1	T_2
read(A)	
A:=A-100	
write(A)	
	read(A)
	A:=A+0.06A
	write(A)
	read(B)
	B:=B+0.06B
	write(B)
	commit
read(B)	
B:=B+100	
write(B)	
commit	

Table. Reading Uncommitted Data

Example: Consider two transactions **T_1** and **T_2** where **T_1** Stands for transferring$100 from **A** to **B** and **T_2** stands for incrementing both **A** and **B** by 6% of their accounts. Suppose that their actions are interleaved as follows,

1) **T_1** deducts $100 from account **A**, then immediately.
2) **T_2** reads accounts of **A** and **B** adds 6% interest to each and then.
3) **T_1** adds $100 to account **B**.

→This corresponding Schedule is illustrated as shown in above **Table.**
→The Problem here is **T_2** has added incorrect 6% interest to each **A** and **B.** Because before commitment that $100 is deducted from **A**, it has added 6% to account **A** and before commitment that $100 is credited to **B**, it has added 6% to account **B**. Thus, the result of this Schedule is different from the result of the other Schedule which is Serializable first **T_1,** then **T_2.**

Unrepeatable Reads (RW Conflicts):
→The second source of anomalies is that a transaction **T_2** could change the value of an object **A** that has been read by a transaction **T_1** and **T_2** is still in progress. This situation causes a

problem that, if **T_1** tries to read the value of **A** again, it will get a different result, even though it has not modified **A** in the meantime .But, this situation could not Arise in a serial execution of two transactions. This is called as **unrepeatable read.**

Example: Suppose that both **T_1** and **T_2** read the same value of **A**, Say 5. Then **T_1** has incremented A value 6 but before commitment as **A** value 6, **T_2** has decremented value from 5 to 4. Thus, instead of answer of **A** value as 5, i.e., from 6 to 5 we got an answer 4 which is incorrect.

<u>Overwriting Uncommitted Data (WW Conflicts):</u>

→The third source of anomalies is that a transaction T2 could overwrite the value of an object A, which has already been modified by a transaction **T_1**, while **T_1** is still in progress.

Example: Suppose that A and B are two employees and their salaries must be Kept equal. Transaction **T_1** sets their salaries to $1000 and transaction **$T_2$** sets their salaries to $2000.

→The following interleaving of the actions **T_1** and **T_2** occurs,

1) **T_1** sets A's salary to $1000, at the same time, **T_2** sets B's salary to $2000.
2) **T_1** sets B's salary to $1000, at the same time, **T_2** sets A's salary to $2000.
3) As a result, A's salary is set to $2000 and B's salary is set to $1000, i.e., the result is not identical.

→Neither transaction reads a salary value before writing it; such a write is called a **blind write.**

→The above example is the best example of blind write because **T_1** and **T_2** are concentrating only on writing but not on reading.

<u>4. Schedules Involving Aborted Transactions:</u>

→All actions of aborted transactions are to be undone and we can therefore imagine that they were never carried out to begin with.

<u>Example:</u> Suppose that transaction T1 deducts $100 from account A then immediately before committing A's new value the transaction T2 reads the current values of accounts A and B and adds 6% interest to each, then commits, but incidentally T1 is aborted. So, we get incorrect result of transaction T2 because T1 was aborted in the middle of the process and T2 has taken incorrect value of A by T1 and added 6%. We say that such a schedule is unrecoverable schedule. The corresponding schedule is shown in **Table**.

T_1	T_2
read(A)	
A:=A-100	
write(A)	
	read(A)
	A:=A+0.06A
	write(A)
	read(B)
	B:=B+0.06B
	write(B)
	commit
abort	

Table. Unrecoverable Schedule

→Whereas, a recoverable schedule is one in which transactions read only the changes of committed transactions.

<u>4. LOCK-BASED CONCURRENCY CONTROL(***)</u>

LOCK: A **lock** is a mechanism to control concurrent access to a data item.

→A **lock** is nothing but a mechanism that tells the DBMS whether a particular data item is being used by any transaction for read/write purpose.

→There are two types of operations, i.e. read and write, whose basic natures are different, the locks for read and write operation may behave differently.

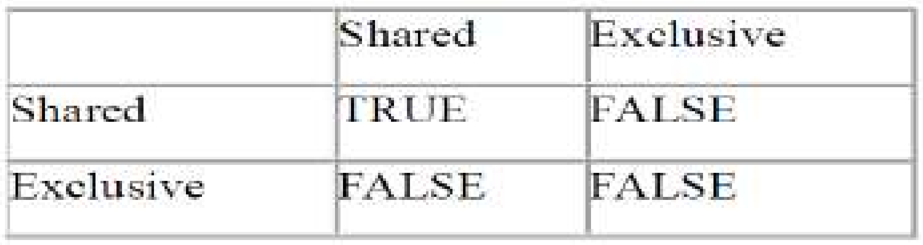
→The simple rule for locking can be derived from here. If a transaction is reading the content of a sharable data item, then any number of other processes can be allowed to read the content of the same data item. But if any transaction is writing into a sharable data item, then no other transaction will be allowed to read or write that same data item.

<u>Types of LOCKS:</u>

1).Shared Lock (S): A transaction may acquire shared lock on a data item P in order to read its content. The lock is shared in the sense that any other transaction can acquire the shared lock on that same data item P for reading purpose.

2).Exclusive Lock (X): A transaction may acquire exclusive lock on a data item P in order to both read/write into it. The lock is excusive in the sense that no other transaction can acquire any kind of lock (either shared or exclusive) on that same data item P.

The relationship between Shared and Exclusive Lock can be represented by the following table which is known as **Lock Matrix**.



	Shared	Exclusive
Shared	TRUE	FALSE
Exclusive	FALSE	FALSE

LOCKING PROTOCOLS:

→A **locking protocol** is a set of rules to be followed by each transaction (and enforced by the DBMS), in order to ensure that even though actions of several transactions might be interleaved, the net effect is identical to executing all transactions in some serial order.

Two Phase Locking (2PL) Protocol:(**********)

→The use of **locks** has helped us to create neat and clean concurrent schedule. The Two Phase Locking Protocol defines the rules of how to acquire the locks on a data item and how to release the locks.

→The **Two-Phase locking protocol** is a concurrency control protocol that guarantees serializability between the schedules.

→In 2PL, the transactions can be divided into two phases. They are

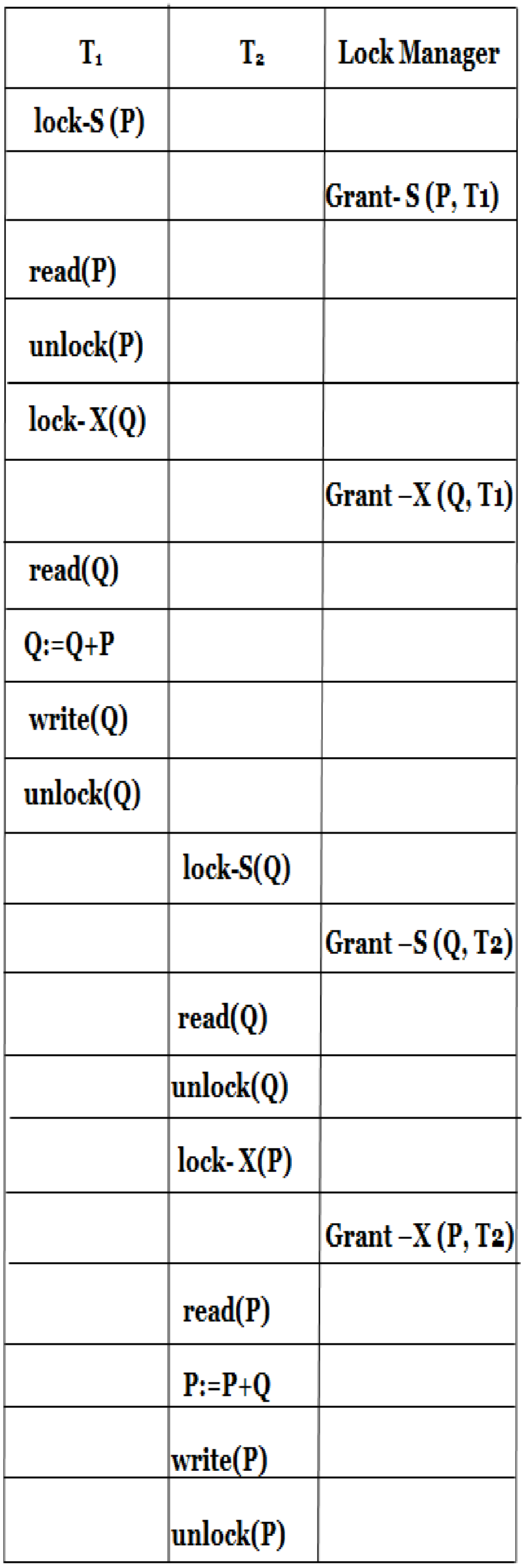
1). Growing (Lock Acquisition) Phase: In this phase, a transaction acquires new locks, but cannot release them.

2). Shrinking (Lock Releasing) Phase: In this phase, transaction releases the existing locks, but cannot acquire any new locks.

→Whenever, a transaction enters the system, it is said to be in growing phase, where it acquire the locks as per the requirement. But, when the locks are released, transaction enters the second phase i.e., shrinking phase where no more lock requests are processed.
→In applied in **2PL**, then lock upgrading and downgrading can be done during growing and shrinking phase respectively.

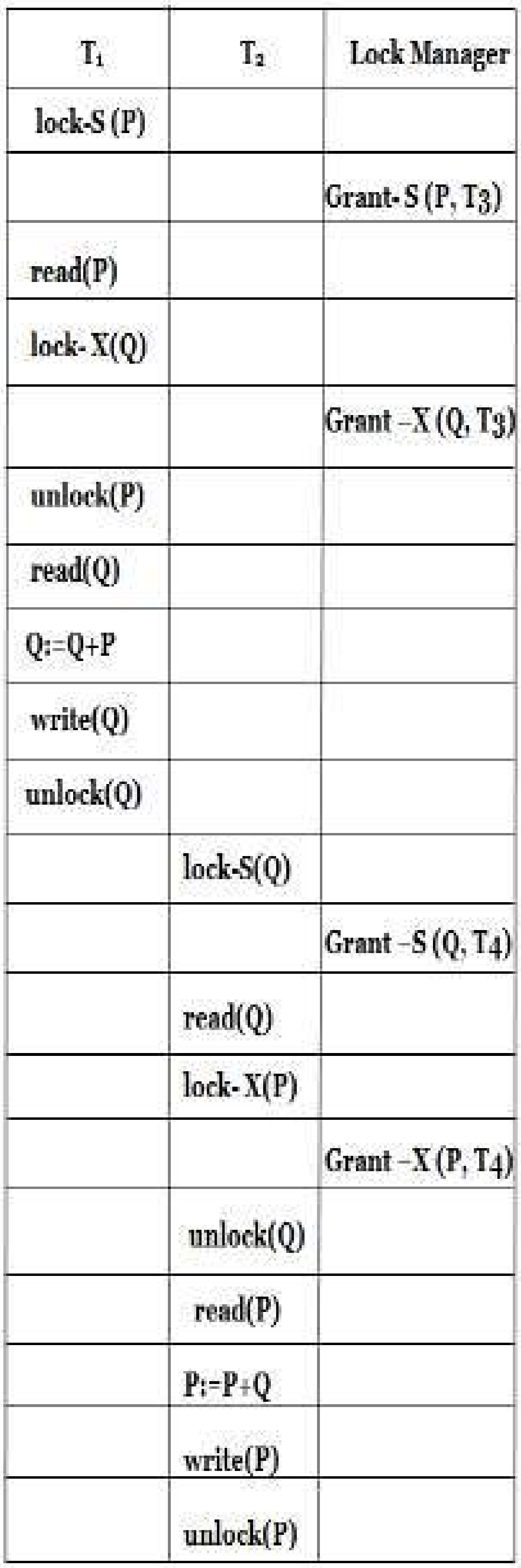
→The transactions in **Schedule S_{10}** (T_1 and T_2) doesn't obey 2PL protocol since the **lock- X (Q)** is executed after the execution of **unlock (P)** operation in T1. Similarly, the **lock- X (P)** operation is executed after the execution of **unlock (Q)** operation in T_2.

→On the other hand, the transactions in **Schedule S_{11}** follows 2PL protocol since **lock- X (Q)** operation is executed before the execution of **unlock (P)** instruction and **lock- X(P)** is executed before the execution of **unlock (Q).**



T_1	T_2	Lock Manager
lock-S (P)		
		Grant- S (P, T_1)
read(P)		
unlock(P)		
lock- X(Q)		
		Grant –X (Q, T_1)
read(Q)		
Q:=Q+P		
write(Q)		
unlock(Q)		
	lock-S(Q)	
		Grant –S (Q, T_2)
	read(Q)	
	unlock(Q)	
	lock- X(P)	
		Grant –X (P, T_2)
	read(P)	
	P:=P+Q	
	write(P)	
	unlock(P)	

Table. Schedule S10



T_1	T_2	Lock Manager
lock-S (P)		
		Grant- S (P, T_3)
read(P)		
lock- X(Q)		
		Grant –X (Q, T_3)
unlock(P)		
read(Q)		
Q:=Q+P		
write(Q)		
unlock(Q)		
	lock-S(Q)	
		Grant –S (Q, T_4)
	read(Q)	
	lock- X(P)	
		Grant –X (P, T_4)
	unlock(Q)	
	read(P)	
	P:=P+Q	
	write(P)	
	unlock(P)	

Table. Schedule S11

Strict Two Phase Locking (S2PL) Protocol:(*******)

→**Strict 2PL** is one of the most popular variations of **2PL** protocol. Schedules following **S2PL** protocol are compatible with the schedules following **2PL** protocol that posses strictness property.
→**S2PL** is designed to overcome the cascading rollback problem of **2PL**.
→A transaction follows **S2PL** if

1) It is compatible with **2PL** and

2) It doesn't releases the exclusive locks until the transaction either commits or aborts.

→The **S2PL** permits release of exclusive locks only at the end of transaction, in order to ensure recoverability and cascadelessness of the resulting Schedules. It guarantees Conflict Serializability.

→ **The Strict Two Phase Locking Protocol** disallows interleaving of transactions, if two transactions access completely different parts of the database, then they proceed without interruption on their ways. If two transactions access same object of the database, then their actions are ordered serially i.e., all actions of locked transactions are completed first, then this lock is released and other transactions can now proceed.

Example:

T_1	T_2
read(P)	
P:=P+10	
write(P)	
	read(P)
	T:=P*20/100
	P:=P+T
	write(P)
	read(Q)
	R:=Q*20/100
	Q:=Q+R
	write(Q)
read(Q)	
Q:=Q+10	
write(Q)	

Table. Interleaved Schedule

→ Let T_1 and T_2 be two transactions. T_1 increments the Value of P and Q by 10 and T_2 increment them by 20%. If the initial Value of P, Q is 10, them after Serial execution, the final value of P is 24 and Q is 14. On the other hand, if the transactions are interleaved, then final value of P is 24 and Q is 22.

→Such anomalies can be avoided by using S2PL. B When T_1 Wishes to Operate on 'P'. It has to acquire the lock on P which means that T_2 cannot interleave T_1.

T_1	T_2
lock- X(P)	
read(P)	
P:=P+10	
write(P)	
lock-X(Q)	
read(Q)	
Q:=Q+10	
write(Q)	
commit	
	lock-X(P)
	read(P)
	T:=P*20/100
	P:=P+T
	write(P)
	lock-X(Q)
	read(Q)
	R:=Q*20/100
	Q:=Q+R
	write(Q)
	commit

Table. Schedule Following S2PL

When the transactions T_1 and T_2 are executed using **S2PL**, then the transactions cannot be interleaved leading to Consistent State.

Advantages:

1) Recoverability is ensured since cascadeless Schedules are generated.
2) It is relatively simple to implement.

Disadvantages:

Concurrency is reduced, since the Schedules generated are subset of Schedules generated using **2PL.**

<u>CONCURRENCY CONTROL:</u>

<u>1. LOCK MANAGEMENT:</u>

→The part of the DBMS that keeps track of the locks issued to transactions is called the **lock manager.** The lock manager maintains a **lock table**, which is a hash table with the data object identifier as the key. The DBMS also maintains a descriptive entry for each transaction in a

transaction table, and among other things, the entry contains a pointer to a list of locks held by the transaction.

→A **lock table entry** for an object—which can be a page, a record, and so on, depending on the DBMS—contains the following information: the number of transactions currently holding a lock on the object (this can be more than one if the object is locked in shared mode), the nature of the lock (shared or exclusive), and a pointer to a queue of lock requests.

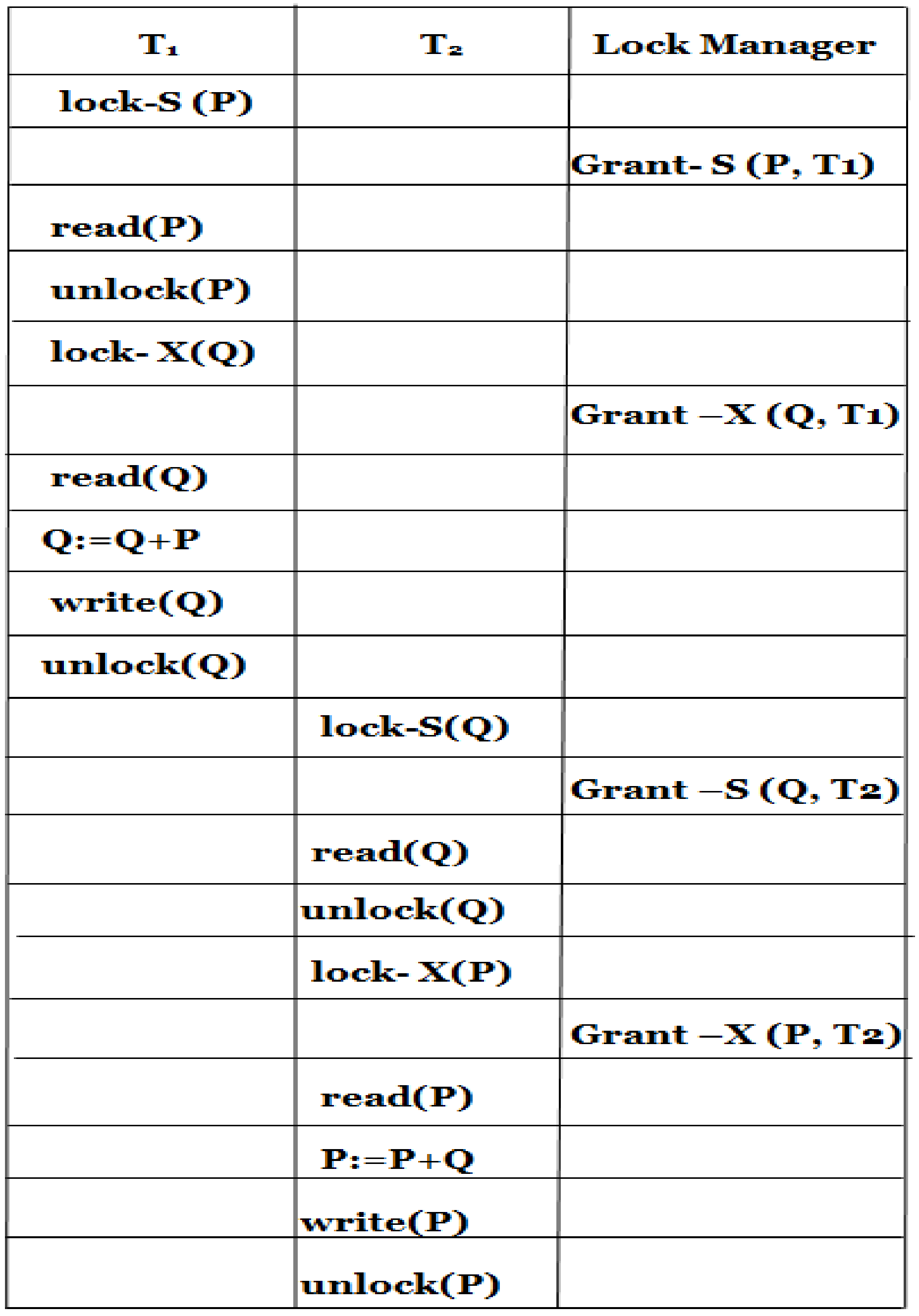
1. Implementing Lock and Unlock Requests:

→According to the **Strict 2PL protocol**, before a transaction T reads or writes a database object O, it must obtain a shared or exclusive lock on O and must hold on to the lock until it commits or aborts. When a transaction needs a lock on an object, it issues a lock request to the **lock manager**:

1. If a **shared lock** is requested, the queue of requests is empty, and the object is not currently locked in exclusive mode, the lock manager grants the lock and updates the lock table entry for the object (indicating that the object is locked in shared mode and incrementing the number of transactions holding a lock by one).

2. If an **exclusive lock is** requested, and no transaction currently holds a lock on the object (which also implies the queue of requests is empty), the lock manager grants the lock and updates the lock table entry.

3. Otherwise, the requested lock cannot be immediately granted, and the lock request is added to the queue of lock requests for this object. The transaction requesting the lock is suspended.



T_1	T_2	Lock Manager
lock-S (P)		
		Grant- S (P, T1)
read(P)		
unlock(P)		
lock- X(Q)		
		Grant –X (Q, T1)
read(Q)		
Q:=Q+P		
write(Q)		
unlock(Q)		
	lock-S(Q)	
		Grant –S (Q, T2)
	read(Q)	
	unlock(Q)	
	lock- X(P)	
		Grant –X (P, T2)
	read(P)	
	P:=P+Q	
	write(P)	
	unlock(P)	

Table. Schedule S10

Atomicity of Locking and Unlocking:

The implementation of **lock** and **unlock** commands must ensure that these are atomic operations. To ensure atomicity of these operations when several instances of the lock manager code can execute concurrently, access to the **lock table** has to be guarded by an operating system synchronization mechanism such as a semaphore.

2. Deadlocks:

→Consider the following example: transaction **T_1** gets an exclusive lock on object A, **T_2** gets an exclusive lock on B, **T_1** requests an exclusive lock on B and is queued, and **T_2** requests an exclusive lock on A and is queued. Now, **T_1** is waiting for **T_2** to release its lock and **T_2** is waiting for **T_1** to release its lock! Such a cycle of transactions waiting for locks to be released is called a **deadlock**.

→Clearly, these two transactions will make no further progress. They hold locks that may be required by other transactions. The DBMS must either prevent or detect (and resolve) such deadlock situations.

Deadlock Prevention: (***********)

→We can prevent deadlocks by giving each transaction a priority and ensuring that lower priority transactions are not allowed to wait for higher priority transactions (or vice versa). One way to assign priorities is to give each transaction a timestamp when it starts up. The lower the timestamp, the higher the transactions priority, that is, the oldest transaction has the highest priority.

→If a transaction $\mathbf{T}_i$ requests a lock and transaction $\mathbf{T}_j$ holds a conflicting lock, the lock manager can use one of the following two policies:

1. Wait-die:
2. Wound-wait

Wait-die:

→ If $\mathbf{T}_i$ has higher priority, it is allowed to wait; otherwise it is aborted.

→In the wait-die scheme, lower priority transactions can never wait for higher priority transactions. In the wound-wait scheme, higher priority transactions never wait for lower priority transactions. In either case no deadlock cycle can develop.

For example, if a transaction requires a resource that is already in use by another transaction,

- If a transaction is requesting a lock on the resource and is found to be of an older time stamp than the transaction which has the resource locked, it is not terminated.
- If a transaction is requesting a lock on the resource and is found to be of a younger time stamp than the transaction which has the resource locked, it is terminated.

Wound-wait:

→If $\mathbf{T}_i$ has higher priority, abort $\mathbf{T}_j$; otherwise $\mathbf{T}_i$ waits.

→The wait-die scheme is non-preemptive; only a transaction requesting a lock can be aborted. As a transaction grows older (and its priority increases), it tends to wait for more and more younger transactions. A younger transaction that conflicts with an older transaction may be repeatedly aborted (a disadvantage with respect to wound-wait), but on the other hand, a transaction that has all the locks it needs will never be aborted for deadlock reasons (an advantage with respect to wound-wait, which is preemptive).

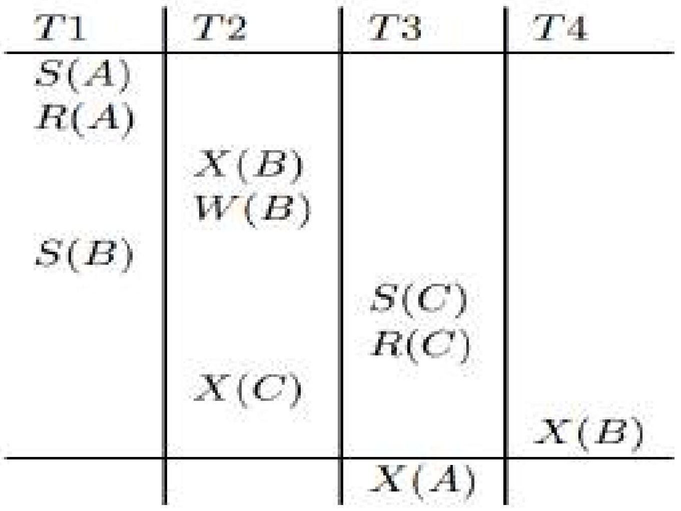
For example, if a transaction requires a resource that is already in use by another transaction,

- If a transaction is requesting a lock on the resource, it is not terminated and made to wait for the resource to be available.
- If a transaction is requesting a lock on the resource and is found to be of an older time stamp than the transaction which is in line for the resource, it can terminate the younger transaction and take over the resource. The time stamp with a younger time stamp is then rebooted.

Deadlock Detection:

→When a transaction $\mathbf{T}_i$ is suspended because a lock that it requests cannot be granted, it must wait until all transactions $\mathbf{T}_j$ that currently hold conflicting locks release them. The lock manager maintains a structure called a waits-for graph to detect deadlock cycles. The nodes correspond to active transactions, and there is an arc from **T$_i$** to **T$_j$** if (and only if) **T$_i$** is waiting for **T$_j$** to release a lock. The lock manager adds edges to this graph when it queues lock requests and removes edges when it grants lock requests.

→Consider the schedule shown in **Figure 19.3**. The last step, shown below the line, creates a cycle in the waits-for graph. **Figure 19.4** shows the waits-for graph before and after this step.



$T1$	$T2$	$T3$	$T4$
$S(A)$			
$R(A)$			
	$X(B)$		
	$W(B)$		
$S(B)$			
		$S(C)$	
		$R(C)$	
	$X(C)$		
			$X(B)$
		$X(A)$	

Figure 19.3 Schedule Illustrating Deadlock

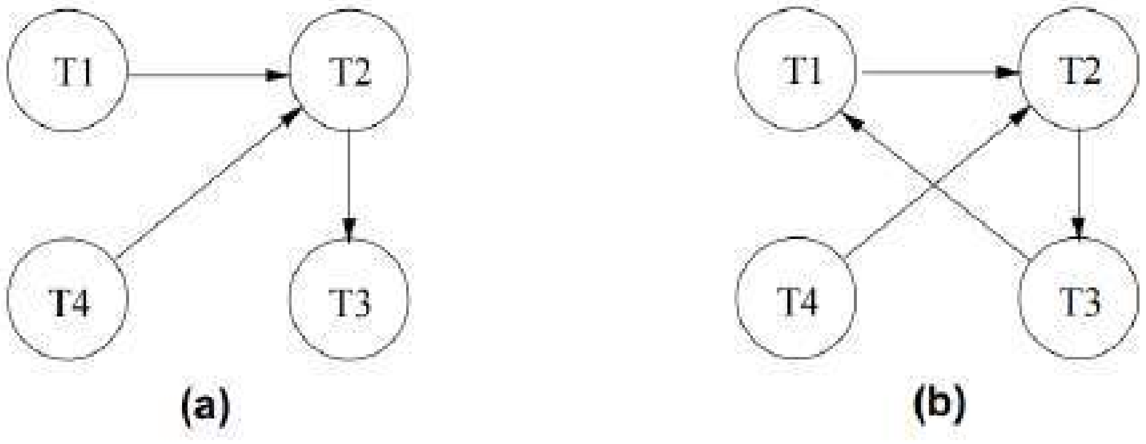


Figure 19.4 Waits-for Graph before and after Deadlock

→The **waits-for graph** is periodically checked for cycles, which indicate deadlock. A deadlock is resolved by aborting a transaction that is on a cycle and releasing its locks; this action allows some of the waiting transactions to proceed.

→As an alternative to maintaining a **waits-for graph**, a simplistic way to identify deadlocks is to use a timeout mechanism: if a transaction has been waiting too long for a lock, we can assume (pessimistically) that it is in a deadlock cycle and abort it.

2. SPECIALIZED LOCKING TECHNIQUES:

(***)

1. Dynamic Databases and the Phantom Problem:

Dynamic Database: A database with “value-based” relationships where typically the relationship is specified at retrieval time and locations of related records are discovered during retrieval.

Example: Both Independent Logical File (ILF) and relational database are value-based.

→**Phantom problem:** Phantom problem is the situation that a transaction retrieves a collection of objects twice and sees different results even though it does not modify any of these objects itself and it follows the strict two phase locking protocol.

→**Phantom problem** is a specific problem to dynamic database, so it cannot occur in a database, where the set of database objects is fixed and only the values of objects can be changed.

Example: Transaction T1 scans the Sailors relation to find the oldest sailor for each of the rating levels 1 and 2. First, T1 identifies and locks all pages (assuming that page-level locks are set) containing sailors with rating 1 and then finds the age of the oldest sailor, which is, say, 71. Next, transaction T2 inserts a new sailor with rating 1 and age 96. Observe that this new Sailors record can be inserted onto a page that does not contain other sailors with rating 1; thus, an exclusive lock on this page does not conflict with any of the locks held by T1. T2 also locks the page containing the oldest sailor with rating 2 and deletes this sailor (whose age is, say, 80). T2 then commits and releases its locks.

→Finally, transaction T1 identifies and locks pages containing (all remaining) sailors with rating 2 and finds the age of the oldest such sailor, which is, say, 63.

→The result of the interleaved execution is that ages 71 and 63 are printed in response to the query. If T1 had run first, then T2, we would have gotten the ages 71 and 80; if T2 had run first, then T1, we would have gotten the ages 96 and 63. Thus, the result of the interleaved execution is not identical to any serial execution of T1 and T2, even though both transactions follow Strict 2PL and commit!

→The problem is that T1 assumes that the pages it has locked include all pages containing Sailors records with rating 1, and this assumption is violated when T2 inserts a new such sailor on a different page. . T1’s semantics requires it to identify all such records, but locking pages that contain such records at a given time does not prevent new “phantom” records from being added on other pages. T1 has therefore not locked the set of desired Sailors records.

→<u>This phantom problem can be handled by the following techniques:</u>

1). If there is no index, and all pages in the file must be scanned, T1 must somehow ensure that no new pages are added to the file, in addition to locking all existing pages.

2). If there is a dense index1 on the rating field, T1 can obtain a lock on the index page—again, assuming that physical locking is done at the page level—that contains a data entry with rating=1. If there are no such data entries, that is, no records with this rating value, the page that would contain a data entry for rating=1 is locked, in order to prevent such a record from being inserted. Any transaction that tries to insert a record with rating=1 into the Sailors relation must insert a data entry pointing to the new record into this index page and is blocked until T1 releases its locks. This technique is called <u>**index locking.**</u>

→Index locking is a special case of a more general concept called predicate locking. In our example, the lock on the index page implicitly locked all Sailors records that satisfy the logical predicate rating=1. More generally, we can support implicit locking of all records that match an arbitrary predicate. General predicate locking is expensive to implement and is therefore not commonly used.

2. Concurrency Control in B+ Trees:

→An **insertion** or **deletion** may lock a node, unlock it and subsequently relock it. Furthermore, a lookup that runs concurrently with split or coalescence operation may find that the desired search key has been moved to the right node by the **split** or **coalescence** operation.

→We illustrate **B+ tree locking** using the tree shown in **Figure 19.5.**

→To **search** for the data entry 38*, a transaction T1 must obtain an S lock on node A, read the contents and determine that it needs to examine node B, obtain an S lock on node B and release the lock on A, then obtain an S lock on node C and release the lock on B, then obtain an S lock on node D and release the lock on C.

→Thus, T1 always maintain a lock on one node in the path, in order to force new transactions that want to read or modify nodes on the same path to wait until the current transaction is done.

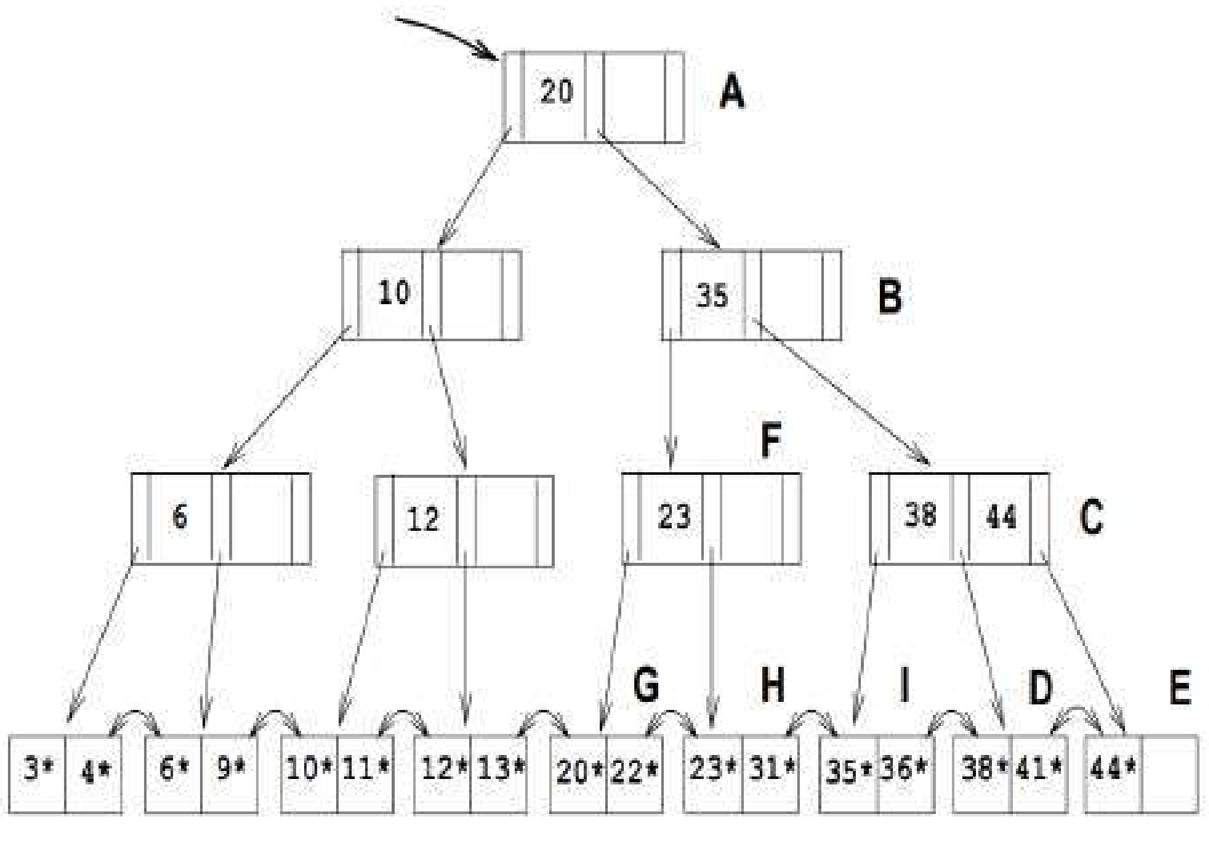


Figure 19.5 B+ Tree Locking Example

→To **delete** for data entry 38*, a transaction T1 must also traverse the path from the root to node D and is forced to wait until T1 delete 38*.

→**To insert** data entry 45*, a transaction must obtain an S lock on node A, obtain an S lock on node B and release the lock on A, then obtain an S lock on node C (observe that the lock on B is not released, because C is full!), then obtain an X lock on node E and release the locks on C and then B. Because node E has space for the new entry 45* is inserted in the node E.

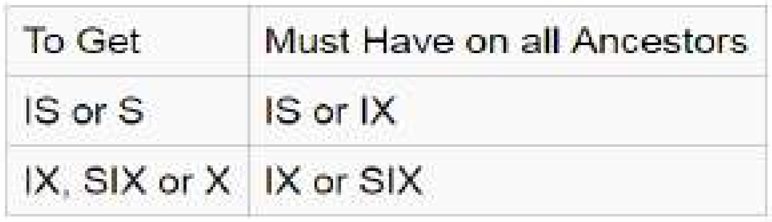
→Thus, the **B$^+$ tree** locking illustrates the potential for efficient locking protocols as a very important special case.

3. Multiple-Granularity Locking:

→Another **specialized locking strategy** is called **multiple-granularity locking**, and it allows us to efficiently set locks on objects that contain other objects. This protocol can ensure searializability.

→In addition to **shared (S) locks** and **exclusive (X) locks** from other locking schemes, like strict two-phase locking, MGL also uses ***intention shared*** and ***intention exclusive*** **locks. IS** locks conflict with **X** locks, while **IX** locks conflict with **S** and **X** locks. The **null lock (NL)** is compatible with everything.

→To lock a node in S (or X), MGL has the transaction lock on all of its ancestors with IS (or IX), so if a transaction locks a node in S (or X), no other transaction can access its ancestors in X (or S and X). This protocol is shown in the following table:



To Get	Must Have on all Ancestors
IS or S	IS or IX
IX, SIX or X	IX or SIX

→**Multiple granularity locking** is usually used with **Non-strict two phase locking** to guarantee serializability. MGL uses lock escalation to determine granularity lock on a node and its ancestors.

→Transaction Ti can lock a node Q, using the following rules:

1. The lock compatibility matrix must be observed.
2. The root of the tree must be locked first, and may be locked in any mode.
3. A node Q can be locked by Ti in S or IS mode only if the parent of Q is currently locked by T_i in either IX or IS mode.
4. A node Q can be locked by T_i in X, SIX, or IX mode only if the parent of Q is currently locked by T_i in either IX or SIX modes.
5. T_i can lock a node only if it has not previously unlocked any node (that is, T_i is two-phase).
6. T_i can unlock a node Q only if none of the children of Q are currently locked by T_i. ! Observe that locks are acquired in root-to-leaf order, whereas they are released in leaf-to-root order.

→Let us consider the following transaction for illustrating **Multiple Granularity Locking Protocol.**

1. If transaction **T_1** wants to read Student **S_{2b}** record present in Course file **Cr_{2b}**, then the transaction must send a lock request to lock manager requesting that it wants to acquire lock on university database, college area node **C_1**, course file **Cr_{2b}** in **IS mode** and a lock on S_{2b} node on **S mode**.
2. If a transaction **T_2** wants to perform write operation on student record **S_{3c}** present in Course file **Cr_{3c}** then the transaction must send a lock request that specifies that it wants to acquire lock on database, College area node **C_2**, on Course file **Cr_{3c}** in **IX mode** and a lock on **S_{3c}** node in **X mode.**
3. If a transaction **T_3** wants to read all the student records in file **Cr_{2b}**,then the transaction must send a lock request, which specifies that it wants to acquire lock on database U and on course area **C_1** in **IS mode** and then on file **Cr_{2b}** in **S mode.**
4. If a transaction **T_4** wants to read the entire database then it must send a lock request which specifies that it wants to acquire lock on the database U in **S mode.**

Advantages:

- The Concurrency level is improved.
- Overhead associated with locking single data item is reduced.

→This Protocol is basically used in an application that consists of both short transactions (these are those transactions that are capable of accessing fewer data items) as well as long transactions (these are those transactions that generate reports from a Complete file).

3. CONCURRENCY CONTROL WITHOUT LOCKING

(*************************)

1. Optimistic Concurrency Control (Validation Based Protocols):

→**Validation techniques** are also called as **Optimistic techniques**.
→Locking protocols take a pessimistic approach to conflicts between transactions and use either transaction abort or blocking to resolve conflicts.

→In **optimistic concurrency control,** the basic premise is that most transactions will not conflict with other transactions, and the idea is to be as permissive as possible in allowing transactions to execute.
→Transactions proceed in three phases:

1. Read Phase

2. Validation Phase and

3. Write Phase

1. Read: The transaction executes, reading values from the database and writing to a private workspace.

2. Validation: If the transaction decides that it wants to commit, the DBMS checks whether the transaction could possibly have conflicted with any other concurrently executing transaction. If there is a possible conflict, the transaction is aborted; its private workspace is cleared and it is restarted.

3. Write: If validation determines that there are no possible conflicts, the changes to data objects made by the transaction in its private workspace are copied into the database.

→Each transaction is assigned three time stamps as follows,
i) When execution is initiated **I(T)**
ii) At the start of the validation phase **V(T)**
iii) At the end of the validation phase **E(T)**

Qualifying conditions for successful validation:
→Consider two transactions, **transaction TA**, **transaction TB** and let the **timestamp of transaction TA** is less than the **timestamp of transaction TB** i.e., TS **(TA) < TS (TB)** then,

1) Before the start of transaction TB, transaction TA must complete its execution. i.e., **E(TA) < I(TB)**
2) The values written by transaction TA must not be necessarily matched with the values read by transaction TB. TA must execute the write phase before TB initiate the execution of validation phase, i.e., **I(TB) < E(TA) < V(TB)**
3) If transaction TA starts its execution before transaction TB completes, then the write phase of transaction TB must be finished before transaction TA starts the validation phase.

Advantages:

i) The efficiency of optimistic techniques lie in the scarcity of the conflicts.
ii) It doesn’t cause the significant delays.
iii) Cascading rollbacks never occurs.

Disadvantages:

i) Wastage in processing time during the rollback of aborting transactions which are very long.
ii) Hence, when one process is in its critical section (a portion of its code), no other process is allowed to enter. This is the principal of mutual exclusion.

2. Timestamp-Based Concurrency Control:

→Timestamp ordering technique is a method that determines the serializability order of different transactions in a schedule. This can be determined by having prior knowledge about the order in which the transactions are executed.

Timestamps:

→Timestamp denoted by TS(TA) is an identifier that specifies the start time of transaction and is generated by DBMS. It uniquely identifies the transaction in a schedule. The timestamp of older transaction (TA) is less than the timestamp of a newly entered transaction (TB) i.e., TS(TA) < TS(TB).

→In timestamp-based concurrency control method, transactions are executed based on priorities that are assigned based on their age. If an instruction IA of transaction TA conflicts with an instruction IB of transaction TB then it can be said that IA is executed before IB if and only if TS(TA) < TS(TB) which implies that older transactions have higher priority in case of conflicts.

Ways of generating Timestamps:

→Timestamps can be generated by using,

i) System Clock: When a transaction enters the system, then it is assigned a timestamp which is equal to the time in the system clock.
ii) Logical Counter: When a transaction enters the system, then it is assigned a timestamp which is equal to the counter value that is incremented each time for a newly entered transaction.

→Every individual data item x consists of the following two timestamp values,

i) WTS(x) (W-Timestamp(x)): It represents the highest timestamp value of the transaction that successfully executed the *write* instruction on x.

ii) RTS(x) (R-Timestamp(x)):It represents the highest timestamp value of the transaction that successfully executed the *read* instruction on x.

Timestamp Ordering Protocol:

→This protocol guarantees that the execution of read and write operations that are conflicting is done in timestamp order.

Working of Timestamp Ordering Protocol:

→The Time stamp ordering protocol ensures that any conflicting read and write operations are executed in time stamp order. This protocol operates as follows:

1) If TA executes read(x) instruction, then the following two cases must be considered,

i) TS(TA) < WTS(x)

ii) TS(TA) WTS(x)

<u>Case 1:</u> If a **transaction TA** wants to read the initial value of some **data item x** that had been overwritten by some younger transaction then, the **transaction TA** cannot perform the read operation and therefore the transaction must be rejected. Then the **transaction TA** must be <u>rolled back and restarted with a new timestamp.</u>

<u>Case 2:</u> If a **transaction TA** wants to read the initial value of some **data item x** that had not been updated then the transaction can execute the read operation. Once the value has been read, changes occur in the read timestamp value **(RTS(x))** which is set to the largest value of **RTS(x) and TS**

2) <u>If TA executes write(x) instruction,</u> then the following three cases must be considered,

i) TS(TA) < RTS(x)

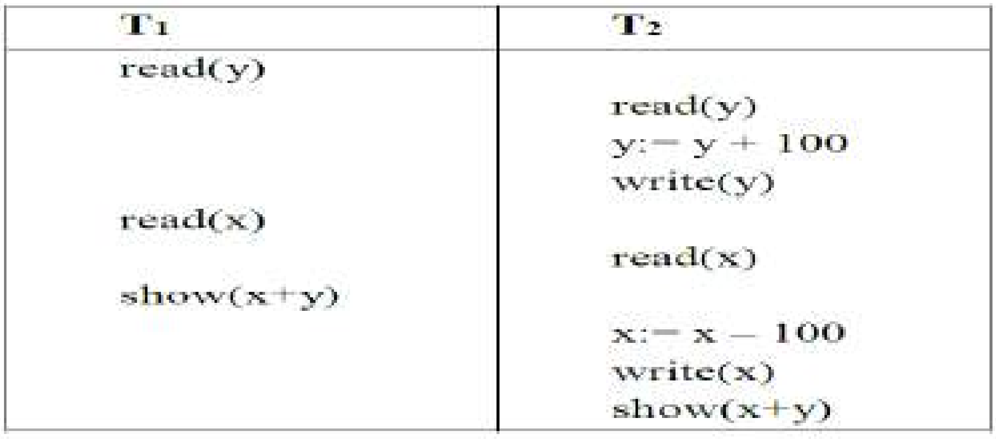
ii) TS(TA) < WTS(x)

iii) TS(TA) >WTS(x)

<u>Case 1:</u> If a **transaction TA** wants to write the value of some **data item x** on which the read operation has been performed by some younger transaction, then the transaction cannot execute the write operation. This is because the value of data item x that is being generated by **TA** was required previously and therefore, the system assumes that the value will never be generated. The write operation is thereby rejected and the **transaction TA** must be <u>rolled back and should be restarted with new timestamp value.</u>

<u>Case 2:</u> If a **transaction TA** wants to write a new value to some **data item x,** which was overwritten by some younger transaction, then the transaction cannot execute the write operation as it may lead to inconsistency of data item. Therefore, the <u>write operation is rejected and the transaction should be rolled back with a new timestamp value.</u> Ignoring outdated writes is called the **Thomas Write Rule.**

<u>Case 3:</u> If a **transaction TA** wants to write a new value on some **data item x** that was not updated by a younger transaction, then the transaction can executed the write operation. Once the value has been written, changes occur on **WTS(x)** value which is set to the value of **TS(TA).**



T1	T2
read(y)	
	read(y)
	y:= y + 100
	write(y)
read(x)	
	read(x)
show(x+y)	
	x:= x − 100
	write(x)
	show(x+y)

The above schedule can be executed under the timestamp protocol when **TS (T1) < TS(T2).**