



DATABASE DESIGN

BITS Pilani
Pilani Campus

SESSION 1





INTRODUCTION TO DATABASE SYSTEMS

Learning Objectives

- Basic terminologies in Database
- Advantages of data base system over conventional file system
- Architecture of data base system
- Three schema levels of data base system

Basic terminologies

Database and Database system

An organization contains different set of data.

The organization must have accurate and reliable data for effective decision making.

An organization can not run without data.

So the data in the organization should be maintained properly.

Database is used to maintain the data in proper way.

A data base is a collection of data and a database system is computerized record keeping system.

The data are stored in the secondary storage device (permanent Storage) in the form of files, tables, objects, etc.

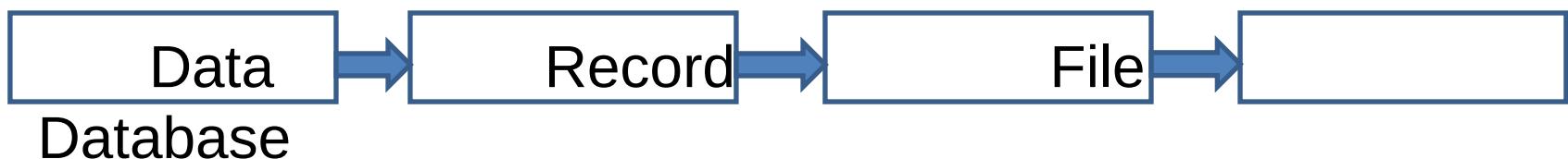
Basic terminologies

A **file** is a collection of one or more record(s) where a **record** is a collection of field values / attribute value.

A **field** is the lowest level of data item of a record or an entity.

The collection of inter-related files is called as the database.

The hierarchy is represented as follows:



Basic terminologies

A single database may contain more than one heterogeneous files.

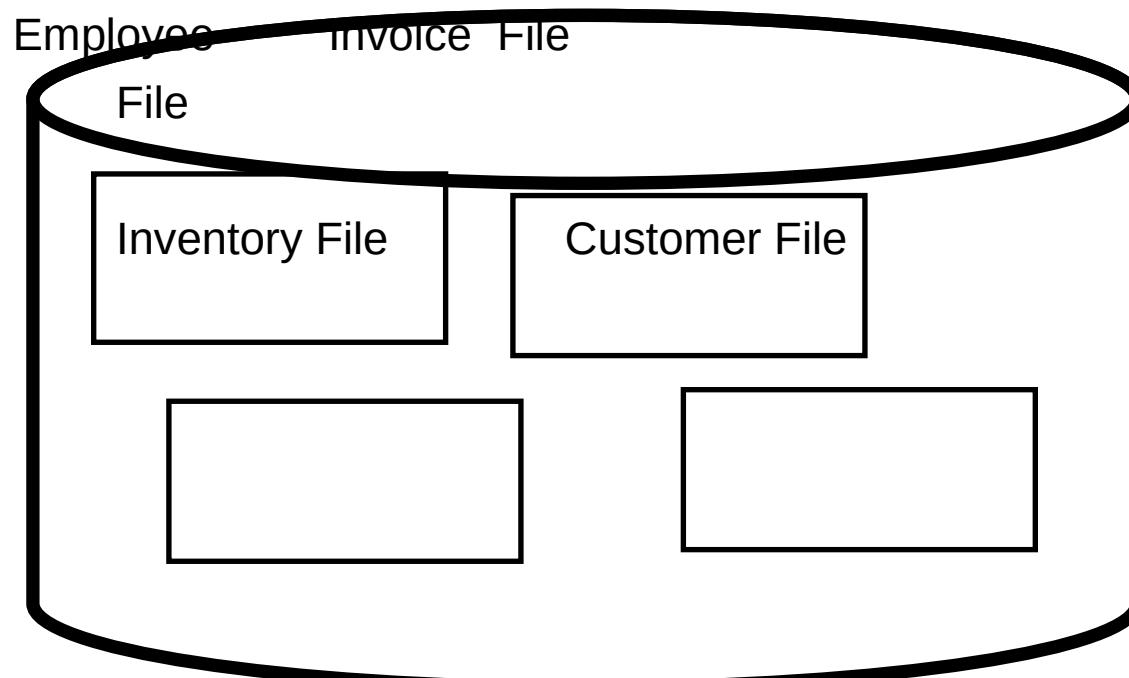
Example:

The database for a manufacturing company contains the files as

1. Employee file – It is a collection of records of all the employees working in the company
2. Customer file - It is a collection of records of all the customers of the company
3. Inventory file – It is a collection of records of maintaining the stock

Basic terminologies

4. Invoice file – It is a collection of records of bills of the customers
The database of the company can be represented as follows



Basic terminologies

Database management System – The software which is used to manage the data in the database is called as the database management system.

The management represents the following

- (i) Creating a new database for collecting the data
- (ii) Adding new data into the existing database
- (iii) Deleting existing data in the database
- (iv) Modifying an existing data in the database
- (v) Listing data of the database with or without using a condition.

Advantages of Database System



1. Redundancy can be reduced

When data are redundant, we face two problems.

They are

- i. Wastage of spaces in the disk
- ii. Inconsistency problem

In data base system, redundancy can be avoided with a mechanism called as Normalization.

Normalization is the procedure which removes the unwanted duplicate data by splitting the table into two or more tables. For Eg.

Assume that a table consists of the data of the students who are doing a particular course.

Advantages of Database System



Student table

Stu#	StuName	dept#	Coun#	CounName
S101	Ram	D1	ST002	Dr. Jackson
S102	Bala	D2	ST002	Dr. Jackson
S111	Madhu	D1	ST001	Dr. Venkat
S122	Divya	D1	ST001	Dr. Venkat
S115	Hema	D2	ST001	Dr. Venkat
S103	Sachin	D1	ST001	Dr. Venkat

Coun# - Counselor Id CounName – Counselor name

Advantages of Database System

The above table is split into two tables (Student and Staff) as follows:

Student

Stu#	StuName	dept#	Coun#
S101	Ram	D1	ST002
S102	Bala	D2	ST002
S111	Madhu	D1	ST001
S122	Divya	D1	ST001
S115	Hema	D2	ST001
S103	Sachin	D1	ST001

Staff

Coun#	CounName
ST001	Dr. Venkat
ST002	Dr. Jackson

Duplicate data related to staff is reduced

Advantages of Database System



2. Inconsistency is avoided

If the value of the attribute shows similar value when it is accessed by two different applications, then data is said to be Consistent.

Inconsistency may arise if they are redundant.

When duplication is avoided, there is no possibility of Inconsistency (Single copy has a single value only).

Though multiple copies are available and when a transaction accesses any one of the copy, a procedure in data base system called as "Propagating updates" updates the latest value in all the copies.

It avoids inconsistency.

Advantages of Database System

3. Data can be shared

Concurrently, data can be accessed by more than one application.

Database which share data to multiple applications is called as the distributed data base.

Distributed data base creates multiple mirror image copies of data and are shared with different applications that are executed in different locations.

When any modification is done on the mirror image copies, they automatically updated in the original copy of the data.

Example: Railway reservation system shares the data to different reservation counters.

Advantages of Database System



4. Data can be integrated

Data that exist in different files can be integrated (Combined) to answer the query of an application.

Example:

Assume that tables Student and Staff consist of the data about the students and staff who handle various courses.

To answer a query , “List the students who have scored more than 90% of marks in various subjects and the information of the staff who handled the subject”, we need the data from both the tables (Student and Staff)

Advantages of Database System



5. Security mechanism can be applied

Confidential data should be secured from unauthorized users.

To protect them from unauthenticated access, data base systems support a security mechanism known as Control access mechanism.

The control mechanism is divided into two types such as Software Controlling mechanism and hardware control mechanism.

Software Control mechanism – Data are protected with the help of a software.

Advantages of Database System



Examples:

- a. **Authentication** procedure which asks login name and password to distinguish authorized and others.
- b. **Encryption** procedure which encrypts the data into unknown format known as Cipher text so that it can be understood by the intruders

Hardware Control mechanism – Data are permitted to be accessed if the user should use physical component.

Example:

Usage of Access Entry card to enter into the prohibited area.

Advantages of Database System

6. Integrity can be maintained

The problem of integrity is the problem of ensuring that data in the database is correct.

When the input values (data) are given to be added into the database, there is a possibility of giving invalid values.

Invalid values create a lot of problem in the data base management.

Some conditions called as integrity constraints in Database system can be enforced at the time of creating the schema of the table.

These constraints check whether the inputs given by the user are valid or not.

Advantages of Database System



If they are invalid, the constraints produce appropriate error messages without storing the invalid values in to the table.

So there is no possibility of keeping Invalid values in the table.

Example: Assume that a table is created for keeping the information about employees working in hourly basis and one of the field is HOURS_WORKED which is used to calculate the wage.

We know that the field may have the value from 0 to 24.

If invalid value is given for the field constructing the record, database system displays the error message if integrity constraints are set.

Advantages of Database System



7. Conflicting requirements can be balanced

We know that data base can be accessed by more than one users and applications.

They may require different storage structures and access methods.

DBA chooses the best storage structure and access method so that performance of the system is improved.

8. Data Independence

In the conventional file system, all the database are dependent data.

Advantages of Database System



When the modification is to be performed on the storage structure, or if we need to make changes in the application of the data base also, the data is said to be **dependent data**.

If the application does not require any updating, the data is said to be **independent data**. Database system supports data dependence.

There are two types of data independence. They are

- i. Logical data independence
- ii. Physical data independence

Advantages of Database System



Logical data independence - When the modification is performed on the application of the data base, If we do not need any change in the storage structure, then the data dependence is called as Logical data independence

Physical data independence - When the modification is performed on the storage structure, If we do not need any change in the application of the data base, data dependence is called as Physical data independence

Architecture of Database system

Components of a database system

1. Data – Data which can be shared between multiple applications and integrated for a single application.

Example for Sharing, the Faculty file can be accessed by the HR department application related to the promotion or for doing some other HR related operations.

Same data can be accessed by the application of the Finance department to prepare payroll.

Example for integrating, Faculty and student files are to be accessed for preparing the list which requires data of both students and Faculty member.

Architecture of Database system

2. Hardware – The hardware components of the system that consists of

- a. SSD (Secondary Storage Device) such as Magnetic tape, Disk to hold data permanently
- b. Processors for executing data(ALU and Control Unit)
- c. Main memory for executing and storing data temporarily
- d. Network – the mechanism to connect more than one Computers

3. Software – It is a software which is used for performing the operation on the database.

It is called as the database management system (DBMS).

Architecture of Database system



The requests from the users to access the data in the database like adding / removing data, retrieving data and updating data are handled by the DBMS.

4. Users – The users in the database system are classified into four types as Application Programmers, Naïve users, Sophisticated Users and database administrator.

Application Programmers – They are the computer professionals who write the application program in a database language or a programming language to access the data in the database.

Architecture of Database system



Naïve Users are unsophisticated users who interact with the system by invoking one of the application programs that have been written by the application programmer.

Sophisticated Users are the users who interact without writing programs.

Instead, they form their requests in a query language.

They submit the query to the tool called as query processor.

The query processor breaks the query into multiple instructions that the storage manager can understand.

Architecture of Database system



Database Administrator (DBA) is the person who has a central control over the data and the programs that can access the data.

The functions of the DBA include

- **Schema definition** – Schema represents the overall structure of the database.
DBA is responsible for deciding what fields are to be used, their data types and their length.
- **Storage structure and access method** – DBA is responsible for deciding how the data are to be stored in the database (disk) and the way in which the data can be accessed (Sequential and random)

Architecture of Database system



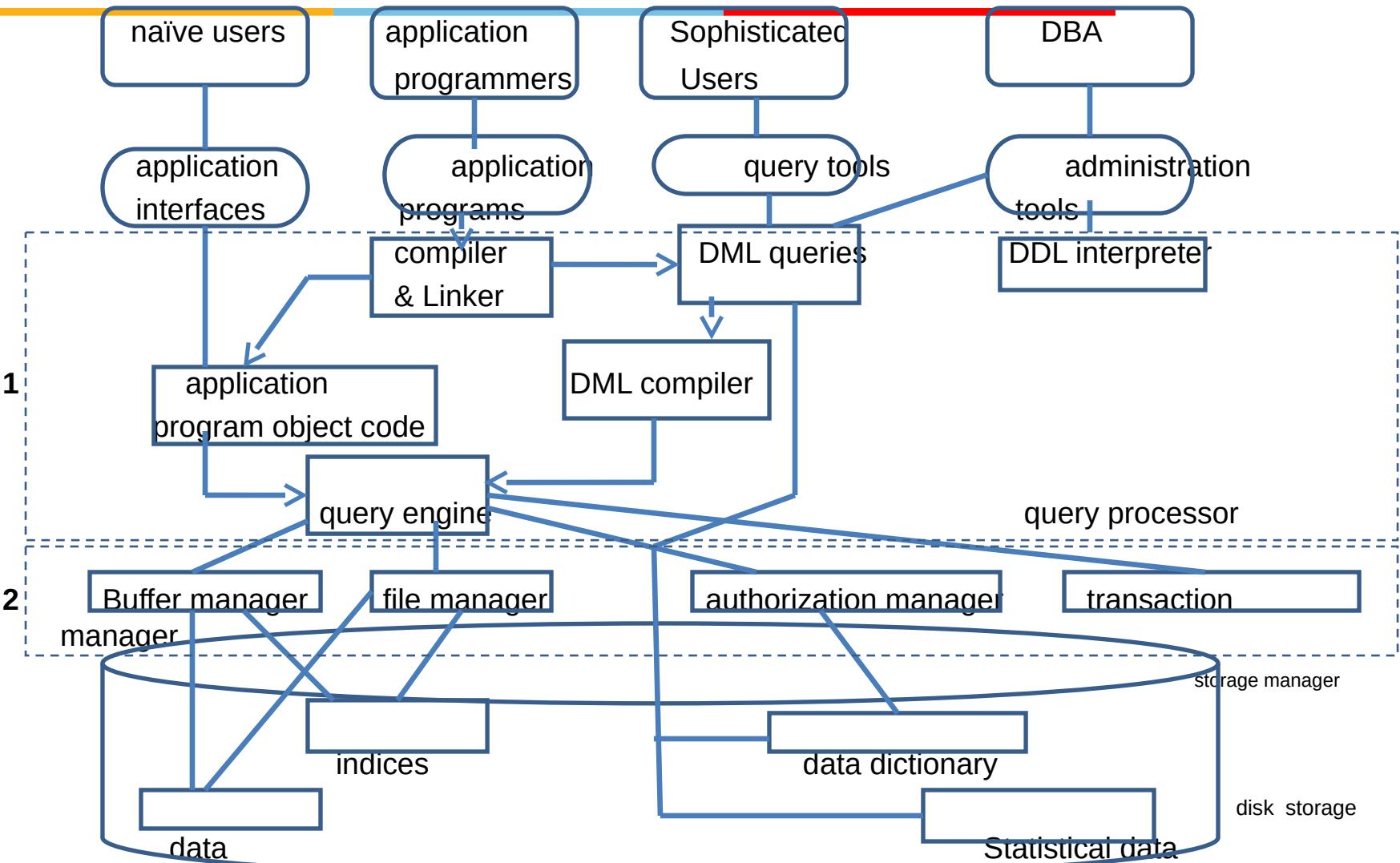
- **Modification in Schema definition and storage structure**

-

DBA is responsible for making any changes in the data structure of the database and storage structure

- **Granting authorization for data access** – DBA has the rights to grant the permission to the users who are authorized to access the data
- **Performing periodical operations** – DBA is responsible for performing routine maintenance activities such as
 - Doing back up periodically for preventing loss of data
 - Monitoring the performance
 - Ensuring that enough free disk space is available

Architecture of Database system



Architecture of Database system



5. Other components

The systems has some more components such as indices, data dictionary and Statistical data.

- *Indices* are the tables which are used for accessing the records randomly
- *Data dictionary* is also table which is the data about data. It contains information such as type, length, etc., about the data stored in the database
- *Statistical data* – These are the data which contain statistical information of the data (How often they are accessed , who had accessed frequently, etc.,)

DBMS Schemas: Internal, Conceptual, External



Database systems comprise of complex data structures. Thus, to make the system efficient for retrieval of data and reduce the complexity of the users, developers use the method of Data Abstraction.

There are mainly three levels of data abstraction:

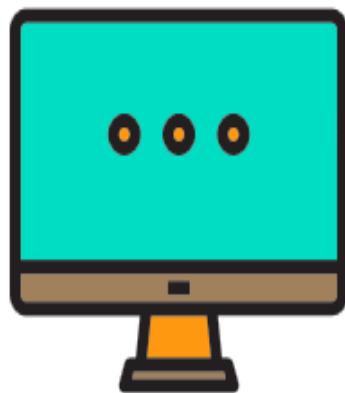
Internal Level: Actual PHYSICAL storage structure and access paths.

Conceptual or Logical Level: Structure and constraints for the entire database

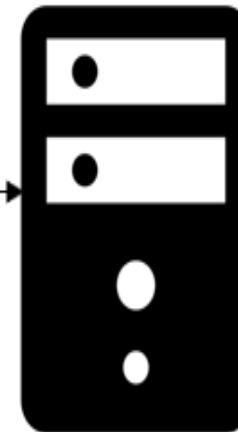
External or View level: Describes various user views

DBMS Schemas: Internal, Conceptual, External

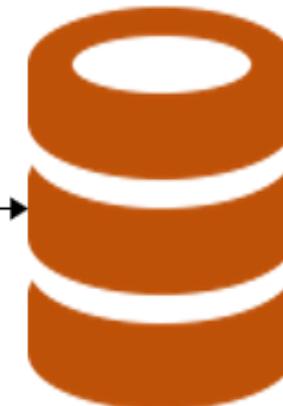
External Level/
View Level



Conceptual Level/
Logical Level



Internal Level



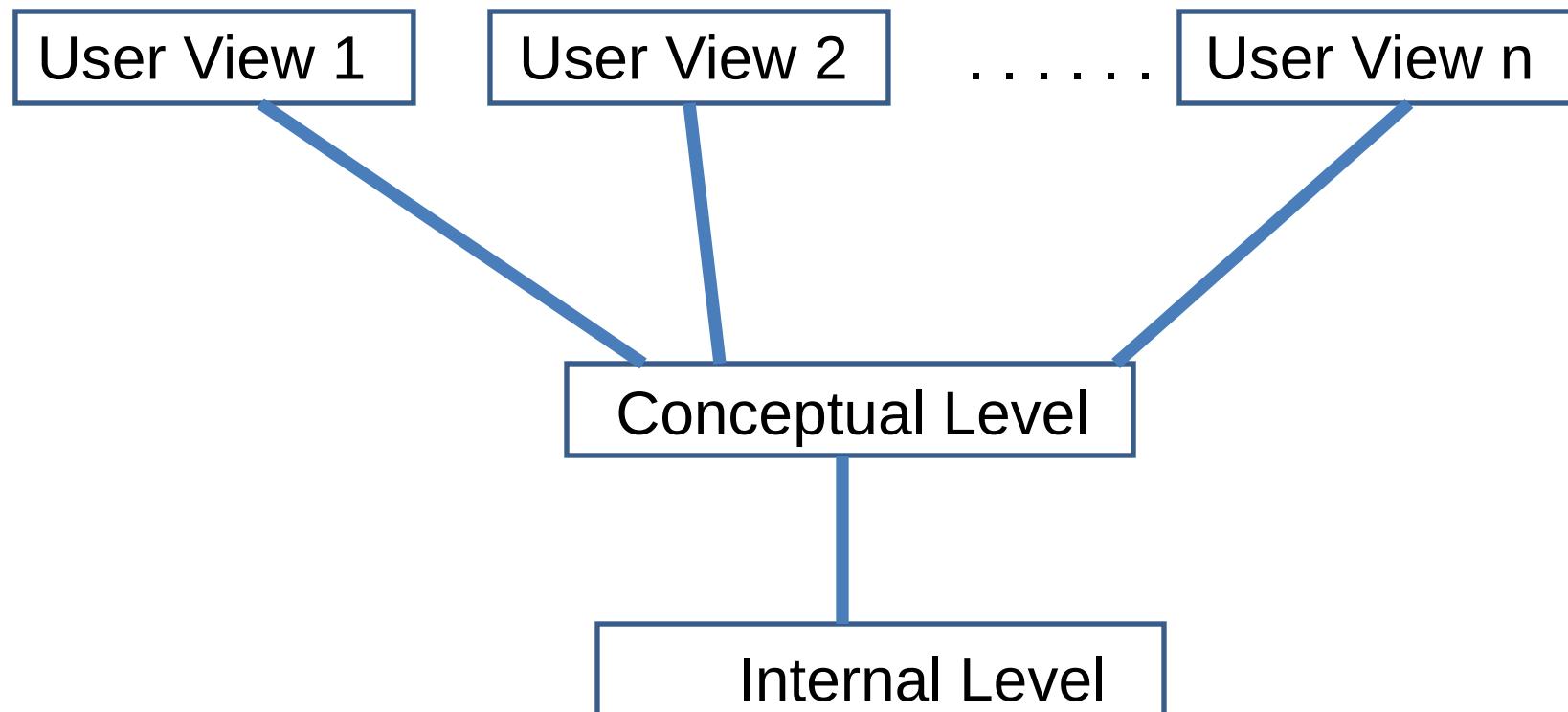
Client

Server

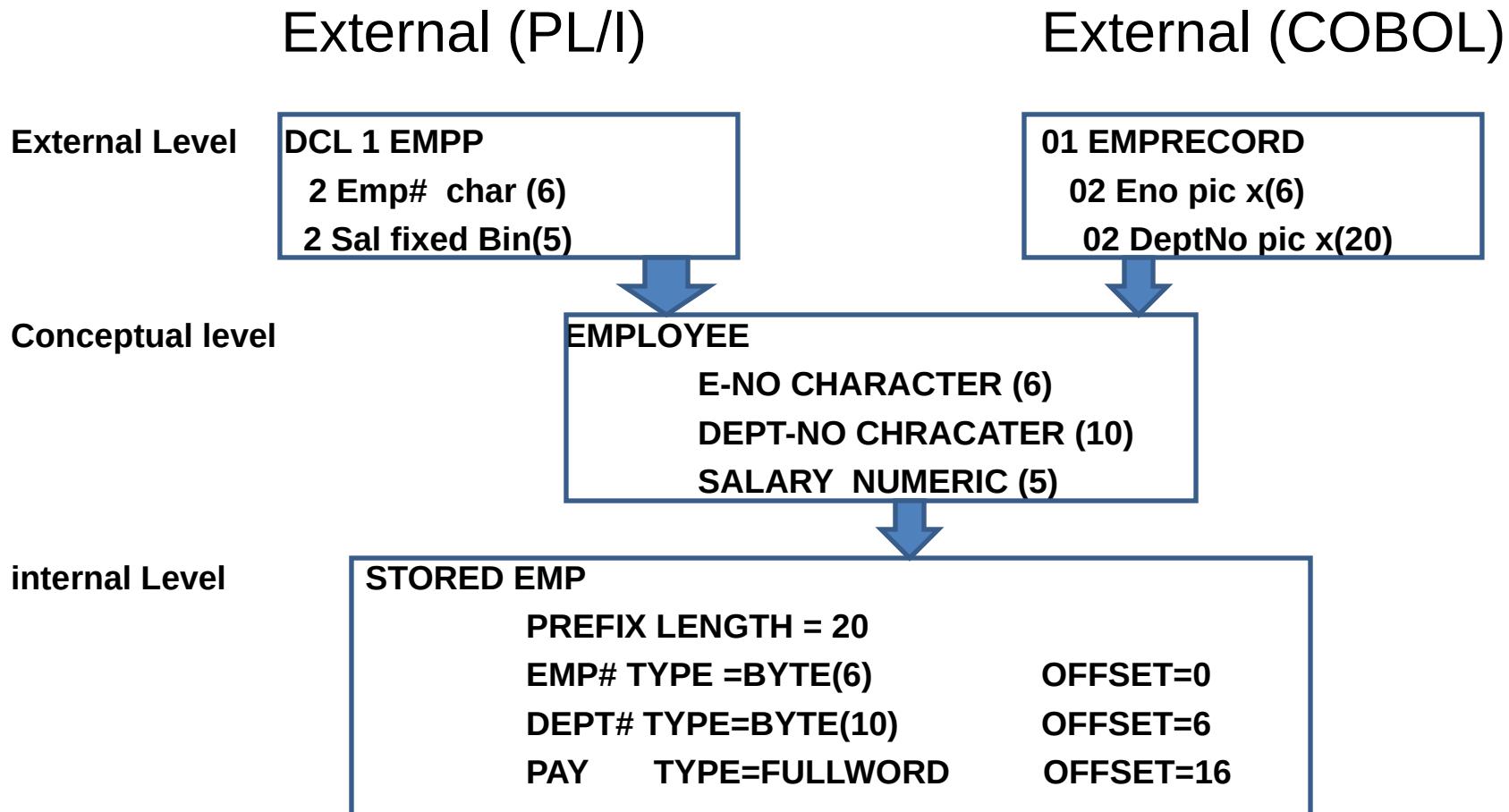
Database

Three Tier Architecture

DBMS Schemas: Internal, Conceptual, External



DBMS Schemas: Internal, Conceptual, External



DBMS Schemas: Internal, Conceptual, External

The purpose of each level is given below

Internal Level/Schema

The internal schema specifies the physical storage structure of the database.

The internal schema is a very low-level representation of the entire database.

It contains multiple occurrences of multiple types of internal record.

These records are also called "stored records".

Facts about Internal schema:

- The internal schema is the lowest level of data abstraction

DBMS Schemas: Internal, Conceptual, External

- It helps us to keep information about the actual representation of the entire database.
- It is like the actual storage of the data on the disk in the form of records
- The internal view tells us what data is stored in the database and how they are stored
- It never deals with the physical devices.
- Instead, internal schema views a physical device as a collection of physical pages

Conceptual Schema/Level

The conceptual schema mentions the Database structure of the whole database for the community of users.

DBMS Schemas: Internal, Conceptual, External

This schema hides information about the physical storage structures and focuses on describing data types, entities, relationships, etc.

This logical level lies between the user level and physical storage view.

However, there is only single conceptual view of a single database.

Facts about Conceptual schema:

- Defines all database entities, their attributes, and their relationships
- Security and integrity information

DBMS Schemas: Internal, Conceptual, External

- In the conceptual level, the data available to a user must be contained in or derivable from the physical level

External Schema/Level

An external schema specifies the part of the database which specific user is interested in.

It hides the unrelated details of the database from the user.

There may be "n" number of external views for each database.

Each external view is defined using an external schema, which consists of definitions of various types of external record of that specific view.

DBMS Schemas: Internal, Conceptual, External

An external view is just the content of the database as it is seen by some specific particular user.

For example, a user from the sales department will see only sales related data.

Facts about external schema:

- An external level is only related to the data which is viewed by specific end users.
- This level includes some external schemas.
- External schema level is nearest to the user
- The external schema describes the segment of the database which is needed for a certain user group and hides the remaining details from the database from the specific user group.

DBMS Schemas: Internal, Conceptual, External

Goal of three level schema of Database

Here, are some Objectives of using Three schema Architecture:

- Every user should be able to access the same data but able to see a customized view of the data.
- The user need not to deal directly with physical database storage detail.
- The DBA should be able to change the database storage structure without disturbing the user's views.
- The internal structure of the database should remain unaffected when changes made to the physical aspects of storage.

DBMS Schemas: Internal, Conceptual, External

Advantages of Database Schema

- We can manage data independent of the physical storage
- Faster Migration to new graphical environments
- DBMS Architecture allows us to make changes on the presentation level without affecting the other two layers
- As each tier is separate, it is possible to use different sets of developers.
- It is more secure as the client doesn't have direct access to the database business logic
- In case of failure of one-tier, no data loss is acquired, as we are always secure by accessing the other tier.

DBMS Schemas: Internal, Conceptual, External



Disadvantages Database Schema

- Complete DB Schema is a complex structure which is difficult to understand
- Difficult to set up and maintain
- The physical separation of the tiers can affect the performance of the Database

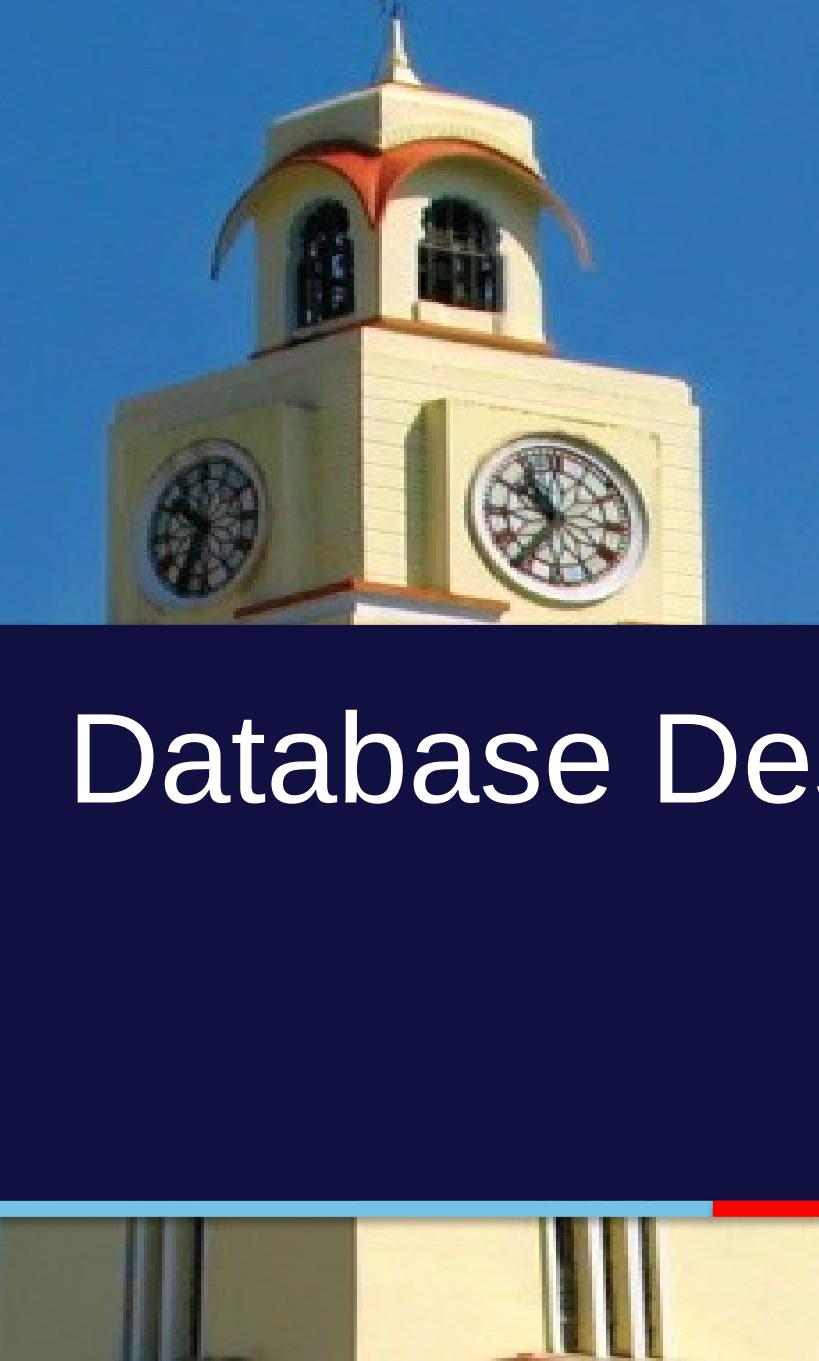
Thanks



BITS Pilani

Pilani | Dubai | Goa | Hyderabad

Database Design





Contact Session 2: Data Modelling

Database Design and ER Modelling

1. Steps in database Design Process
2. Concepts and notations
3. Relationships and constraints
4. Examples

Entity Relationship Model

- It describes the data involved in a real world enterprise in terms of objects and their relationship
- It illustrates the logical structure of databases .
- It is used in a phase called conceptual database design
- ER model is used mainly as a design tool and documentation for the system
- Peter Chen developed ERDs in 1976.

- An entity is an object or concept about which you want to store information.
- An entity may be an object with a physical existence (person, car) or with a conceptual existence (company, university).
- An entity has properties which are known as attributes of the entity.
- Attributes can be simple or composite.

Examples:

An entity STUDENT has the following attributes

StName (Student Name), Stu# (Student Number) , age and
dep(Department Name)

Entity 1

StName="Ranjith"
Stu#="S1001"
Age = 22
Dep="BIO"

Entity=2

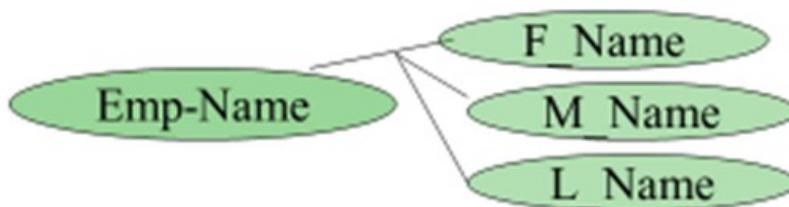
StName="Sachin"
Stu#="S1010"
Age = 23
Dep="CSE"

Attribute types

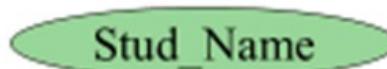
– Simple



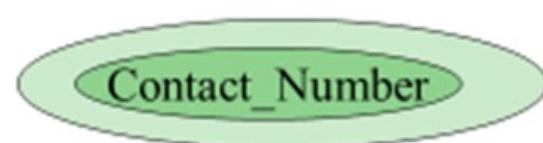
& Composite



– Single-valued

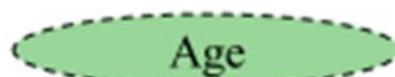


& Multi-valued



– Derived

* E.g Age: Age could be derived from date of Birth



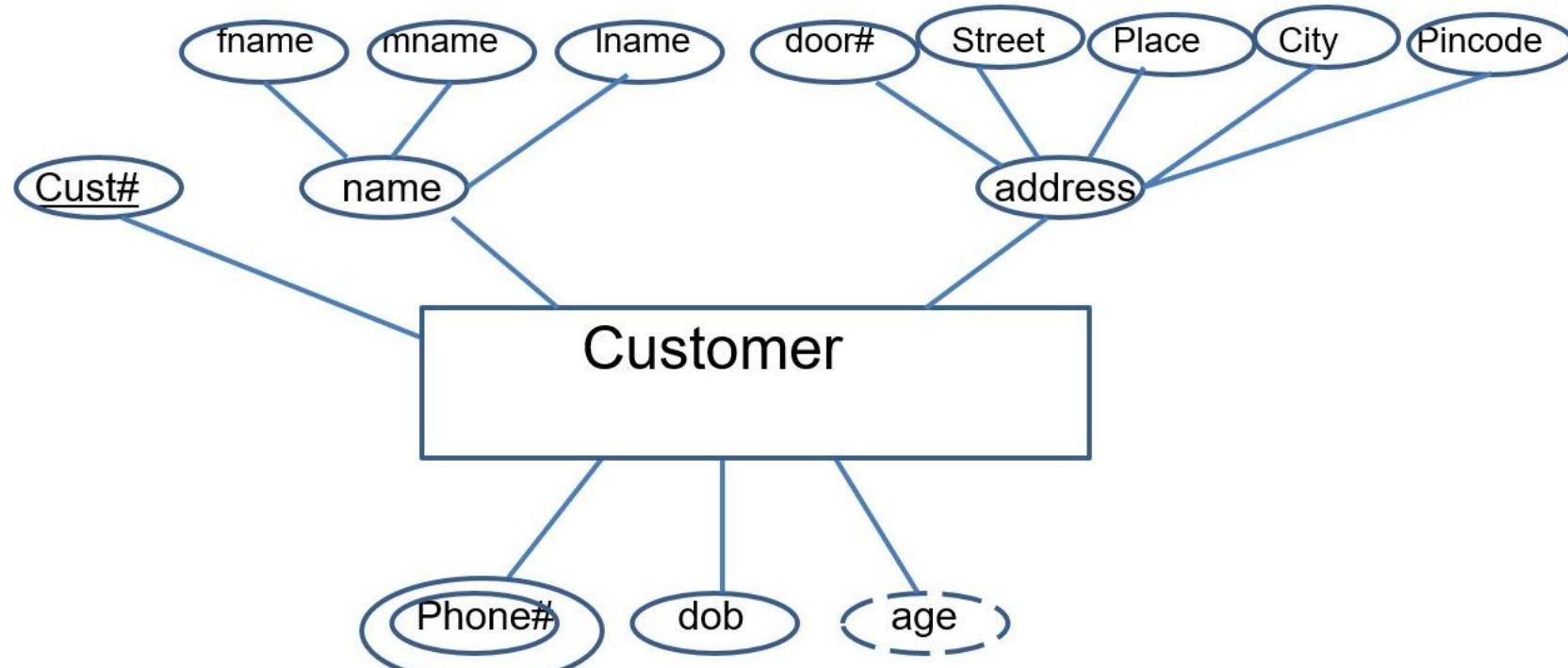
Entity Relationship Model

innovate

achieve

lead

The following is the ER diagram that consists of different types of attributes



•3. Relationship

- A relationship represents the association between two or more entities.
- Example: Assume that there are two entity sets: Customer and Loan as

Customer

Cust#	Cust Name	Address
C101	Mr.Ram	Chennai-5
C112	Mr.Raj	Chennai-12
C109	Mr.SreeRam	Chennai-1
C107	Mr.Bala	Chennai-2

Loan

Loan #	Amount
L1005	30000
L3002	23000
L1015	12000
L8019	10000

Entity Relationship Model

In the above diagram , there are two relationships.
First relationship is defined that Customer Mr. Ram has taken a loan of Rs.10000 and its number is L8019.
The second relationship shows that Mr. SreeRam has received the loan amount of Rs.30000 and its loan number is L1005.

Entity set

It is a set of entities of the same type that share same properties or attributes.

Example: A set of persons who are customers at a particular bank can be defined as the entity set **Customer** as in the following diagram

Entity Relationship Model



Customer

Cust#	Cust Name	Address	Balance(Rs)
C101	Mr.Ram	Chennai-5	28235
C112	Mr.Raj	Chennai-12	3140
C109	Mr.SreeRam	Chennai-1	112
C107	Mr.Bala	Chennai-2	12345

Symbols used for drawing ER diagrams and their usages are :

Rectangles - To represent entity sets

Ellipses - To represent attributes

Diamonds - To represent the relationship set

Line - To link attributes with the entity and entity to another entity

Double Ellipses – To represent multivalued attributes

Dotted Ellipses – To represent Derived attribute

Double rectangles – Representing weak entity sets

NOTATION FOR ER SCHEMAS

<u>Symbol</u>	<u>Meaning</u>
	ENTITY TYPE
	WEAK ENTITY TYPE
	RELATIONSHIP TYPE
	IDENTIFYING RELATIONSHIP TYPE
	ATTRIBUTE
	KEY ATTRIBUTE
	MULTIVALUED ATTRIBUTE
	COMPOSITE ATTRIBUTE
	DERIVED ATTRIBUTE
	TOTAL PARTICIPATION OF E₂ IN R
	CARDINALITY RATIO 1:N FOR E₁:E₂ IN R
	STRUCTURAL CONSTRAINT (min, max) ON PARTICIPATION OF E IN R

- Identifying Entities

Given a system description you can identify entities by:

- Select all nouns
- Eliminate one's not necessary
- Filter those for which system needs to store record.
- Identify the attributes that describes the entity

Entities are classified as:

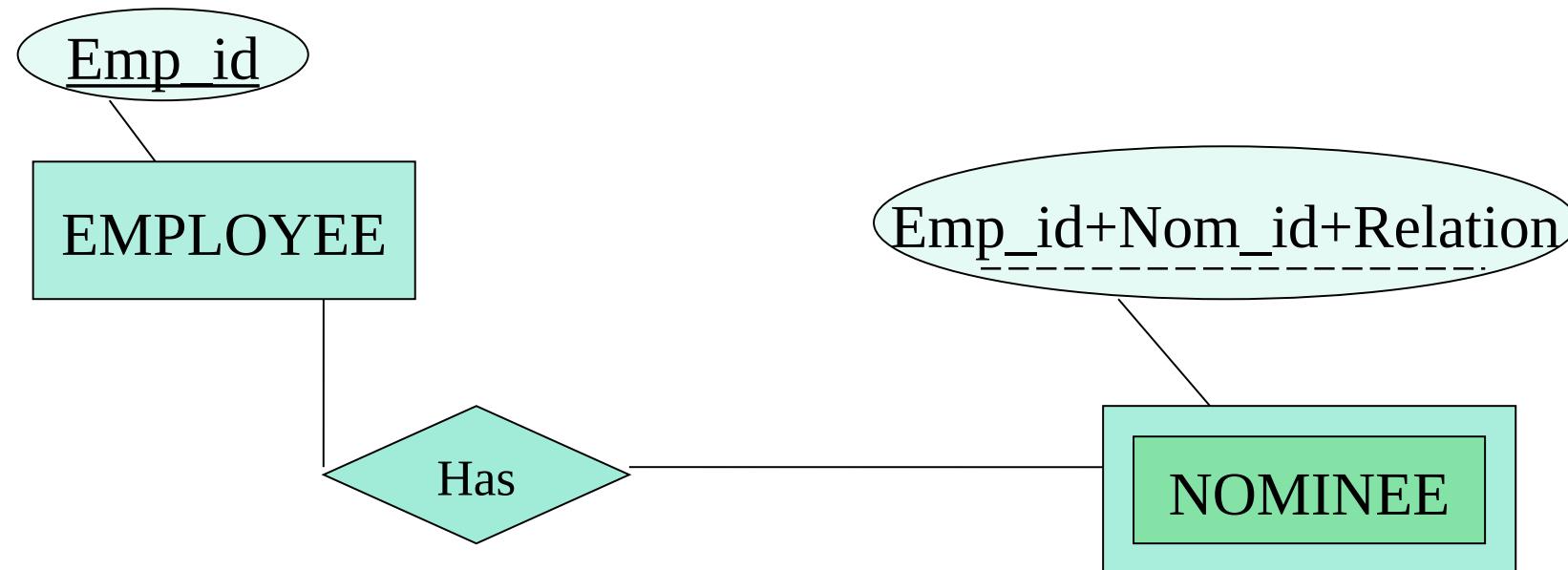
- **DOMINANT ENTITIES:** An entity that does not depend on another entity for its existence is a dominant or strong entity
- **WEAK ENTITIES:** An entity that depends on some other entity for its existence

Existence Dependencies

- If the existence of entity x depends on the existence of entity y , x is said to be existence dependent on y .
- Entity y is said to be a Dominant entity and x is said to be a subordinate entity

Weak Entity

- A weak entity does not have a primary key of its own
- You can use a combination of the primary key of the dominant entity & an attribute of the weak entity as the primary key of the weak entity.
- Thus a discriminator of Weak Entity + Prime attribute of the its Dominant Entity.



Relationships

- Entities are linked to each other through relationships.
Hence a relationship is an association between 2 entities.
- Eg:

Entity

SUPPLIER

Relationship

supplies

Entity

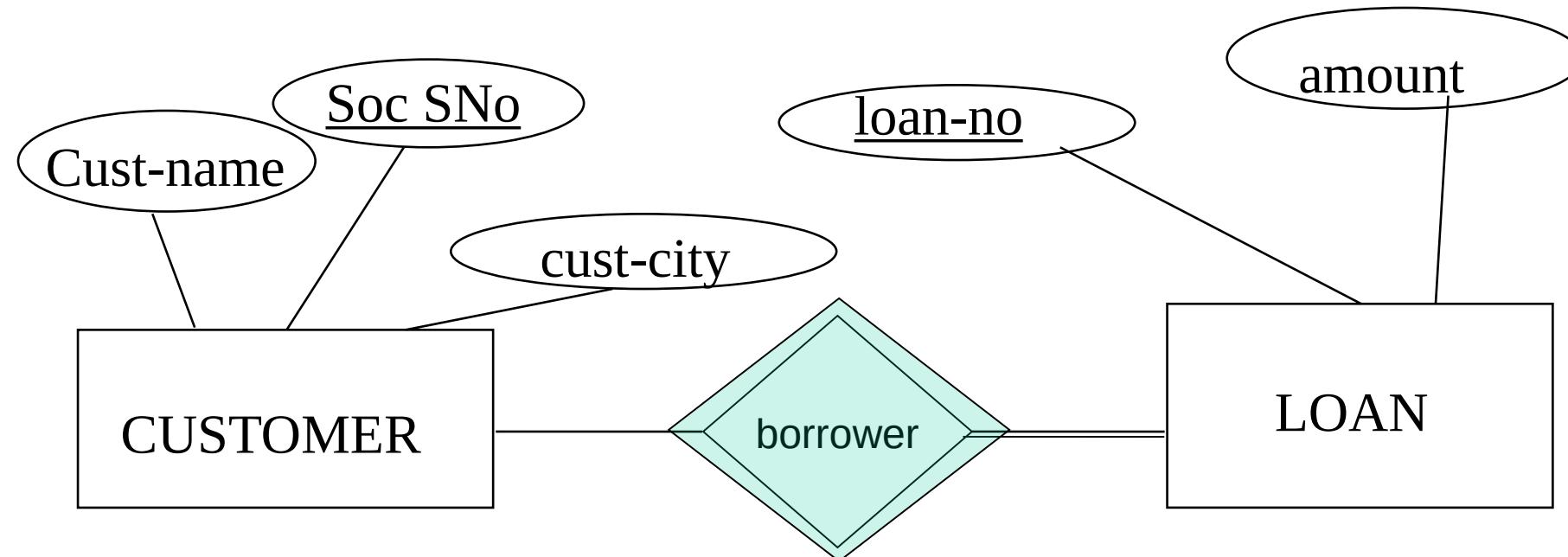
ITEM

Participation of an entity set in a relationship



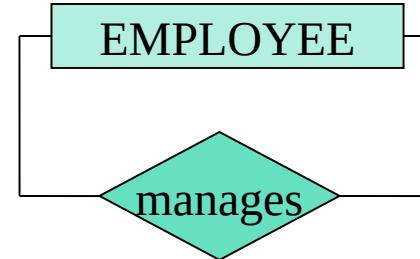
- **Total:** The participation of an entity set E in a relationship set R is said to be total if every entity in E participates in at least one relationship R
- **Partial:** Part of the set of entities participate in some relationship

Total & Partial Participation

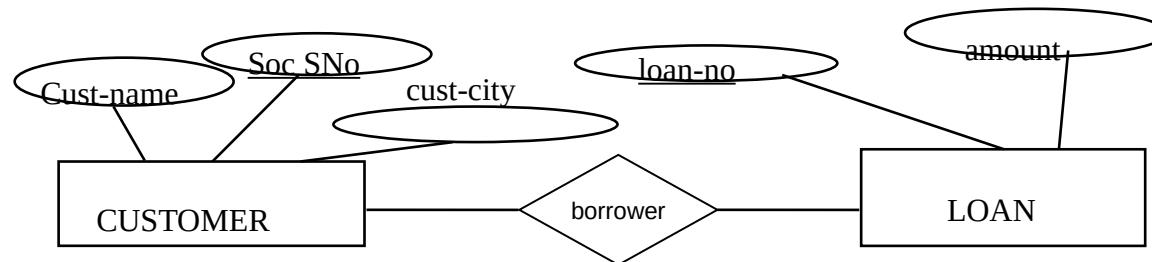


Degree of Relationship

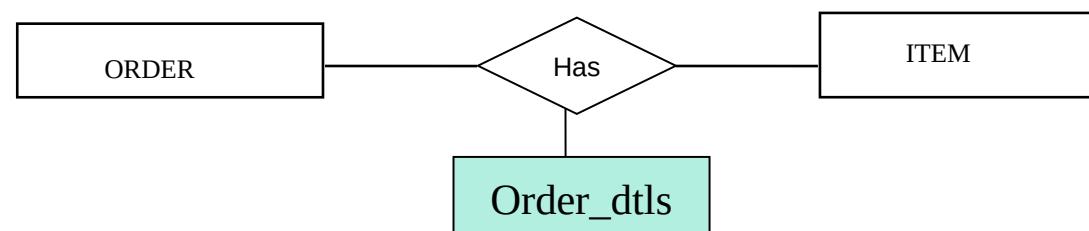
Unary : Degree 1



Binary : Degree 2



Ternary : Degree 3



N-ary : Degree n

Further Reading: <https://www.gatevidyalay.com/tag/total-and-partial-participation-in-dbsms/>

Cardinality Ratio/ Mapping Cardinality



- Mapping Cardinality express the number of entities to which another entity can be associated via a relationship set.
- Once the entities & relationship between the entities is identified you can represent them in a ER diagram

- 1-1

Supplier-----Key Person

- 1-M

Dept-----Emp

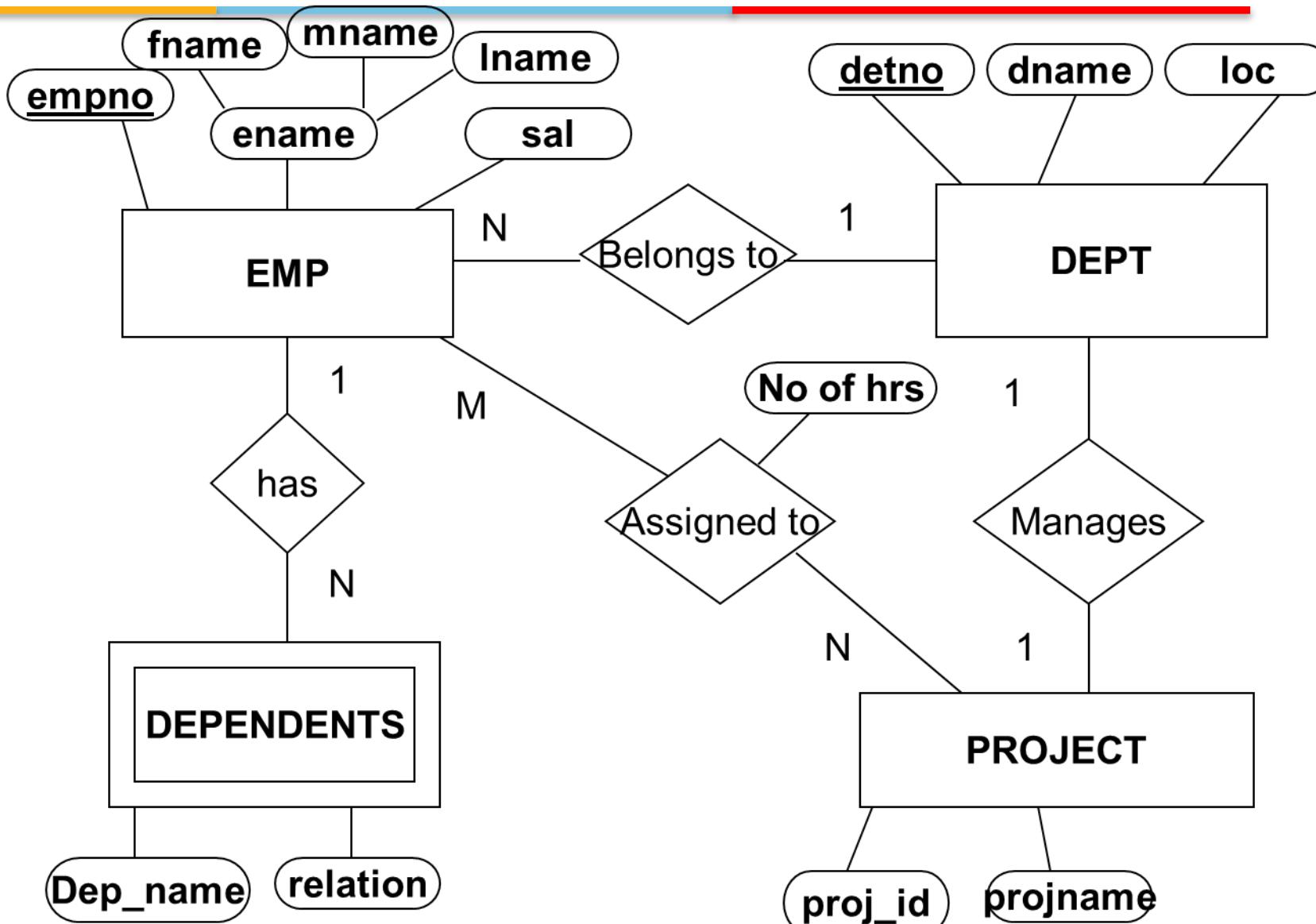
- M-1

Emp-----Dept

- M:M

Student----Subject

ER Example



ER ... (contd.)

This schema has 3 entity types

- EMPLOYEE
 - DEPT
 - PROJECT
- & one weak entity type DEPENDENT

There is M:N relationship between PROJECT and EMP entity sets

There is a 1:1 relationship between DEPT and PROJECT

There is a 1:M relationship between DEPT and EMP entity sets

No of hrs is an attribute on the relation “assigned to” between EMP and PROJECT

Steps to Create an ERD

Following are the steps to create an ERD.



Let's study them with an example:

In a university, a Student enrols in Courses.

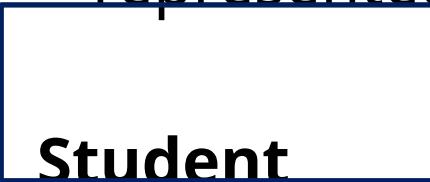
A student must be assigned to at least one or more Courses.

Each course is taught by a single Professor.

To maintain instruction quality, a Professor can deliver only one course

Step 1) Entity Identification - Identification of entities in the given application

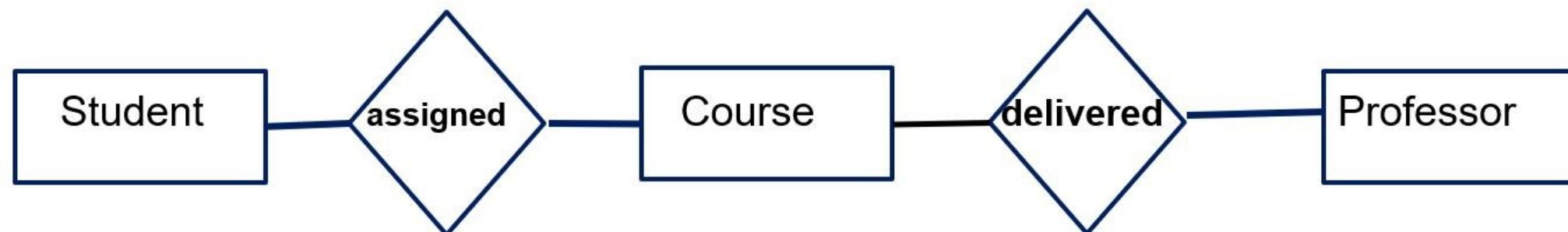
We have three entities **Student, Course and Professor** and are represented as follows



Step 2) Relationship Identification - Identifying the relationships between the entities

We have the following two relationships

1. The student is **assigned** a course (Relationship between a Student and the course)
2. Professor **delivers** a course (Relationship between a Course and the Professor)

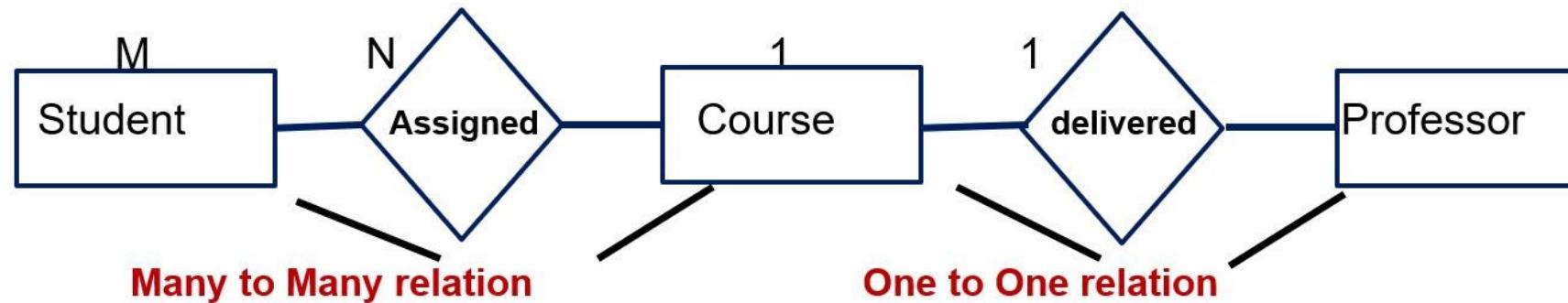


Step 3) Cardinality Identification – Identification of mapping cardinalities between the entities.

A student can be assigned **multiple** courses

A Professor can deliver only **one** course

A Course can be taken by only **one** Professor



Step 4) Identify Attributes – Identifying the attributes of each entity of the application and the primary key of the entity set.

*For Entity Set **Student**, the attributes are Name and Number in which Number is the Primary key*

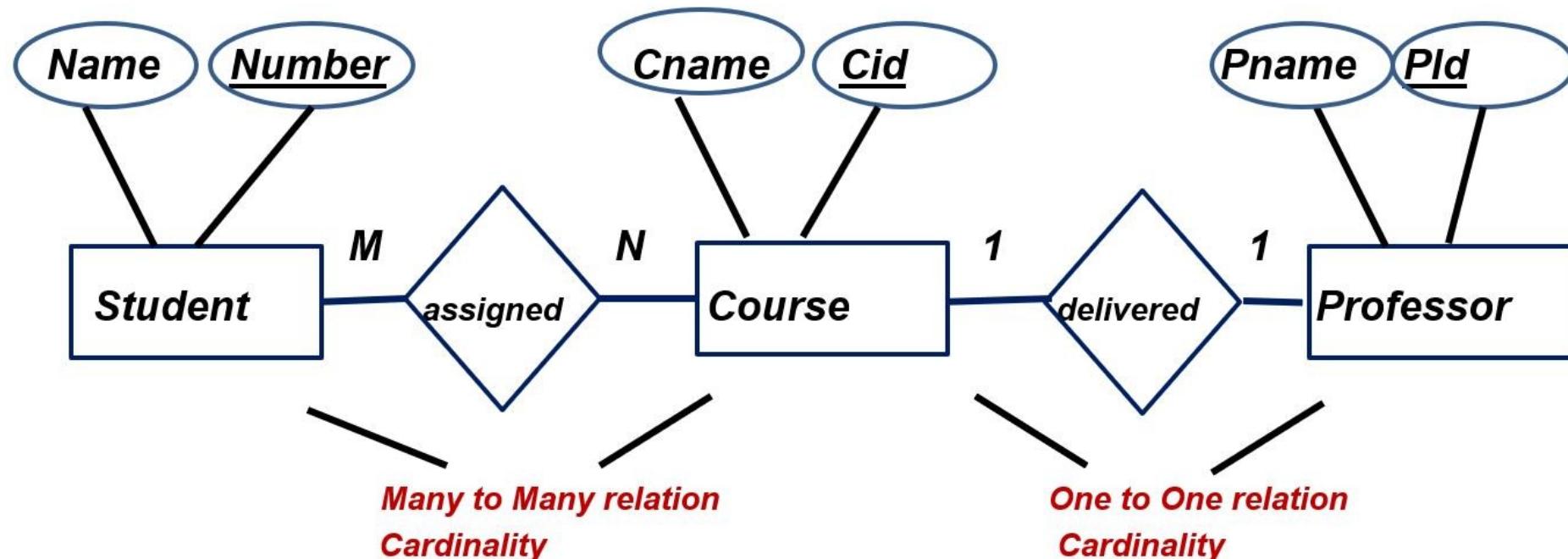
*For Entity Set **Course**, the attributes are Course Name and Course Id in which Course Id is the Primary key*

*For Entity Set **Professor**, the attributes are PName and PNumber in which PNumber is the Primary key*

Entity Relationship Model

Step 5) Create the ERD – Produce the Entity relationship diagram by linking all the attributes with the corresponding Entities.

The resultant ERD for the given application is



Enhanced features of ER Model

- Includes all modeling concepts of basic ER
- Additional concepts: subclasses / super classes, specialization/generalization, categories, attribute inheritance
- The resulting model is called the enhanced-ER or Extended ER (E2R or EER) model
- It is used to model applications more completely and accurately if needed
- It includes some object-oriented concepts, such as inheritance

Subclasses and Super classes

- An entity type may have additional meaningful sub groupings of its entities
- Example: EMPLOYEE may be further grouped into SECRETARY, ENGINEER, MANAGER, TECHNICIAN, SALARIED_EMPLOYEE, HOURLY_EMPLOYEE, ...
- Each of these groupings is a subset of EMPLOYEE entities
- Each is called a subclass of EMPLOYEE
- EMPLOYEE is the superclass for each of these subclasses
 - These are called superclass/subclass relationships.
 - Example: EMPLOYEE/SECRETARY, EMPLOYEE/TECHNICIAN
 - These are also called IS-A relationships (SECRETARY IS-A EMPLOYEE, TECHNICIAN IS-A EMPLOYEE, ...)

Attribute Inheritance in Superclass / Subclass Relationships

- An entity that is member of a subclass *inherits* all attributes of the entity as a member of the superclass
- It also inherits all relationships.

Specialization

Specialization is the process of defining a set of subclasses of a superclass

The set of subclasses is based upon some distinguishing characteristics of the entities in the superclass

Example: {SECRETARY, ENGINEER, TECHNICIAN} is a specialization of EMPLOYEE based upon *job type*.

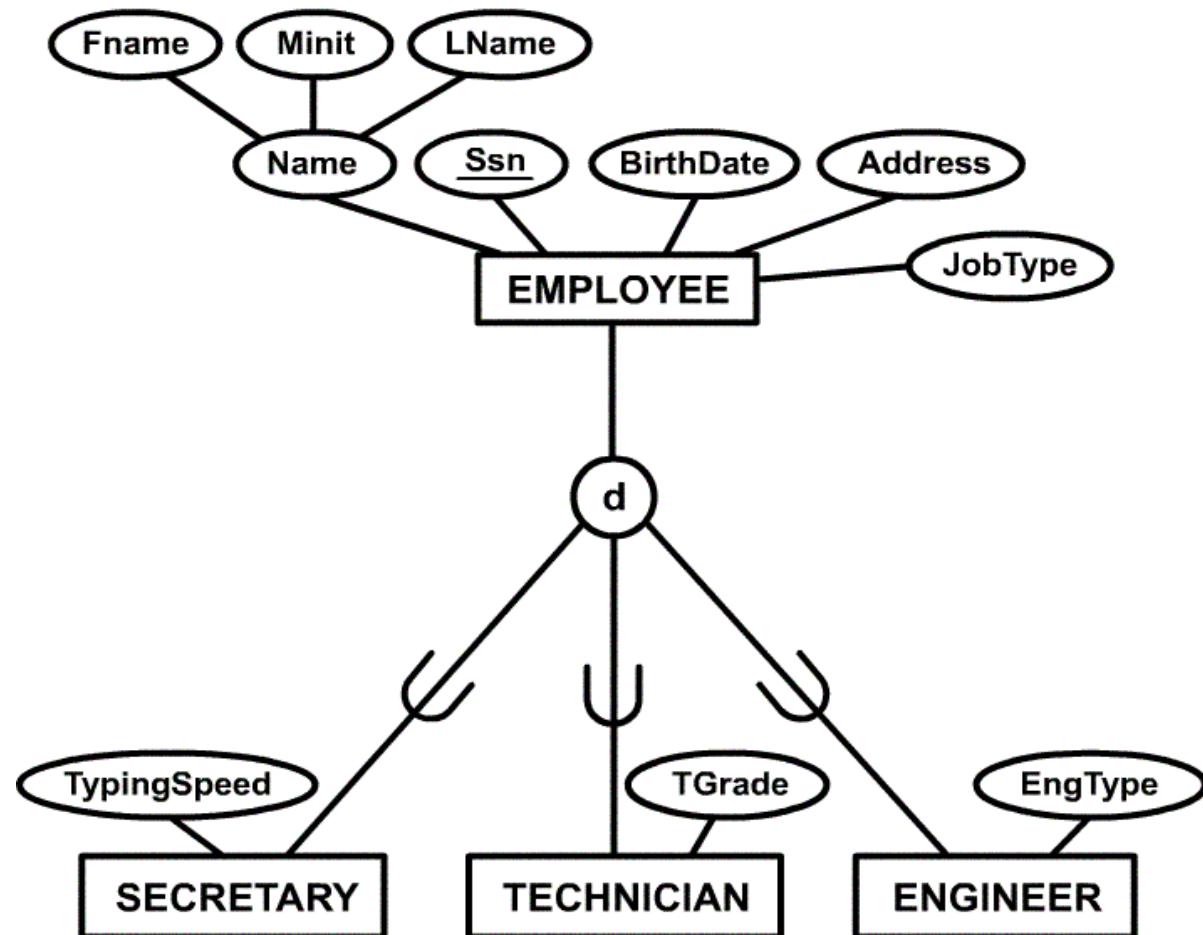
- May have several specializations of the same superclass

Specialization

Example: Another specialization of EMPLOYEE based on *method of pay* is {SALARIED_EMPLOYEE, HOURLY_EMPLOYEE}.

- Superclass/subclass relationships and specialization can be diagrammatically represented in EER diagrams
- Attributes of a subclass are called specific attributes. For example, TypingSpeed of SECRETARY
- The subclass can participate in specific relationship types. For example, BELONGS_TO of HOURLY_EMPLOYEE

Example of a Specialization



- The reverse of the specialization process
- Several classes with common features are generalized into a superclass; original classes become its subclasses
- Example: CAR, TRUCK generalized into VEHICLE; both CAR, TRUCK become subclasses of the superclass VEHICLE.
 - We can view {CAR, TRUCK} as a specialization of VEHICLE
 - Alternatively, we can view VEHICLE as a generalization of CAR and TRUCK

Constraints on Specialization and Generalization



- If we can determine exactly those entities that will become members of each subclass by a condition, the subclasses are called ***predicate-defined*** (or condition-defined) subclasses
 - Condition is a constraint that determines subclass members
 - Display a predicate-defined subclass by writing the predicate condition next to the line attaching the subclass to its superclass.
 - Two types of conditions or constraints
 - Attribute defined
 - User defined

Constraints on Specialization and Generalization

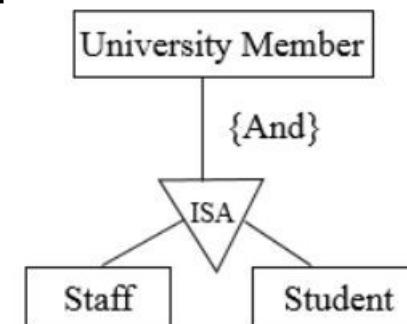
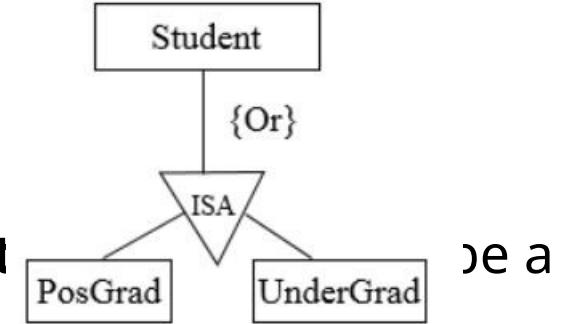


- If all subclasses in a specialization have membership condition on same attribute of the superclass, such specialization is called an ***attribute defined-specialization***
 - Attribute is called the defining attribute of the specialization
 - Example: JobType is the defining attribute of the specialization {SECRETARY, TECHNICIAN, ENGINEER} of EMPLOYEE
- If no condition determines membership, the subclass is called ***user-defined***
 - Membership in a subclass is determined by the database users by applying an operation to add an entity to the subclass
 - Membership in the subclass is specified individually for each entity in the superclass by the user. Eg. Dependent of an employee can be wife, parent, child

Two other conditions apply to a specialization/generalization:

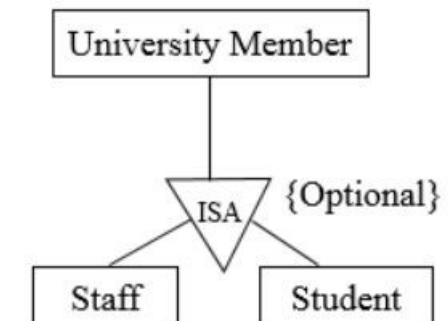
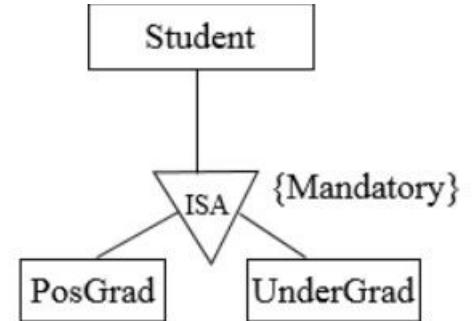
1. Disjoint Constraint:

- Specifies that the subclasses of the specialization must be **disjoint** (a member of at most one of the subclasses of the specialization)
- Specified by d in EER diagram
- If not disjointed, **overlapping constraint**; that is the same entity may be a member of more than one subclass of the specialization
- Specified by o in EER diagram



2. Completeness Constraint:

- **Total** specifies that every entity in the superclass must be a member of some subclass in the specialization/ generalization
- Shown in EER diagrams by a double line
- **Partial** participation - Not all entities are involved in the relationship.
- Partial participation is represented by single lines.

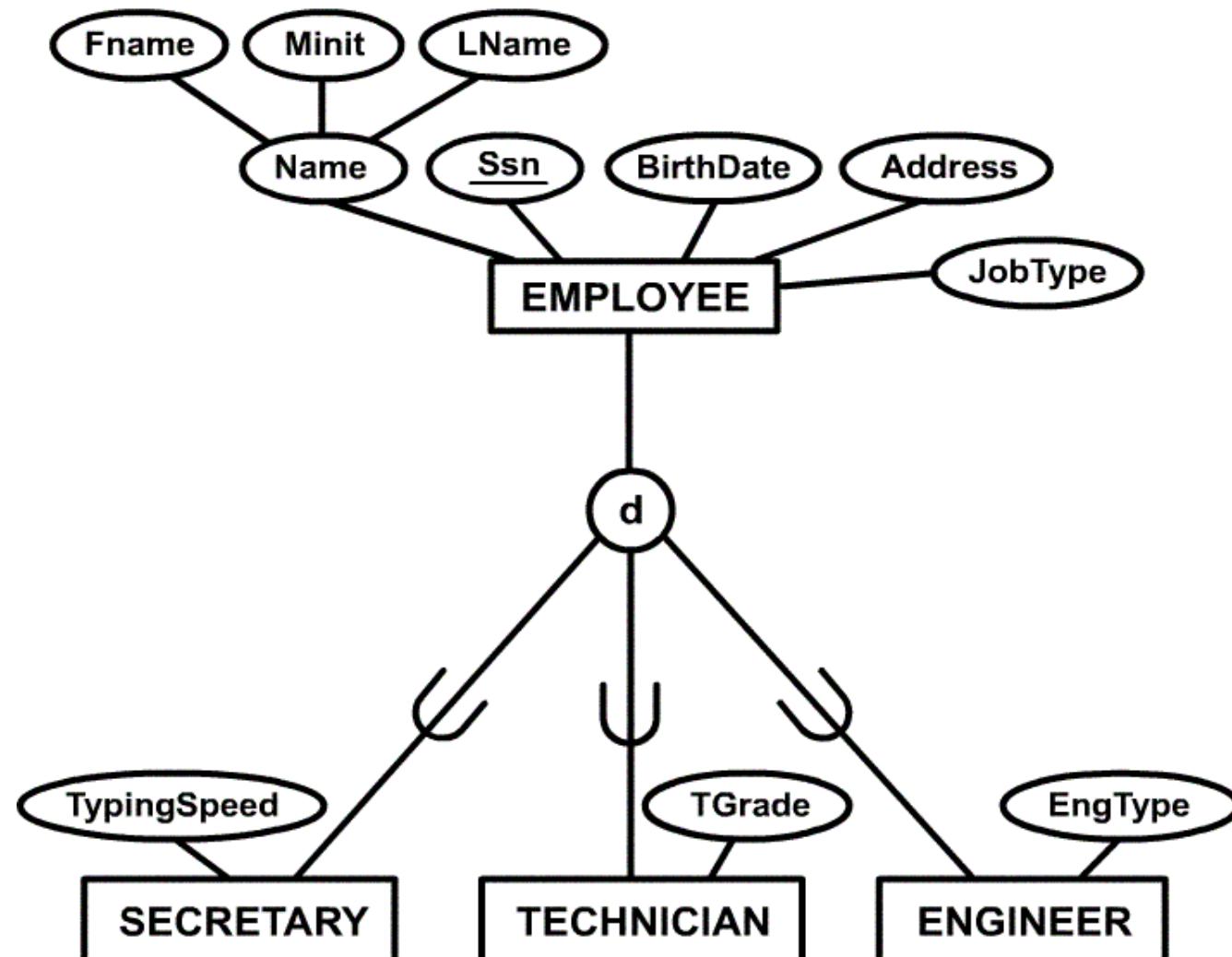


Constraints on Specialization and Generalization



- Hence, we have four types of specialization/generalization:
 - Disjoint, total
 - Disjoint, partial
 - Overlapping, total
 - Overlapping, partial
- Note: Generalization usually is total because the superclass is derived from the subclasses.

Example of disjoint partial Specialization

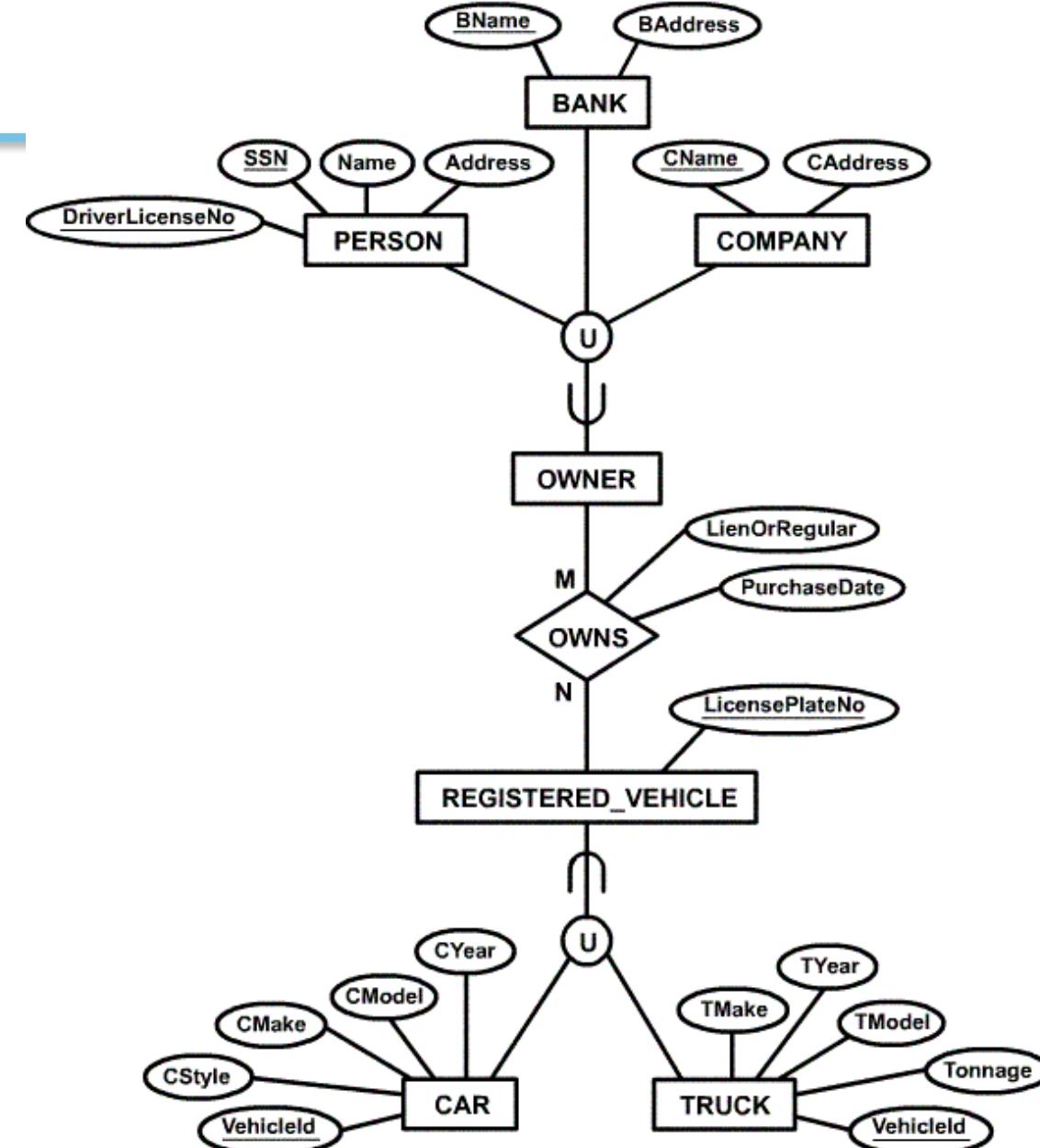


- All of the superclass/subclass relationships we have seen thus far have a single superclass
- A shared subclass is subclass in more than one distinct superclass/subclass relationships, where each relationships has a single superclass (multiple inheritance)
- In some cases, need to model a single superclass/subclass relationship with **more than one superclass**
- Super classes represent different entity types
- Such a subclass is called a category or UNION TYPE

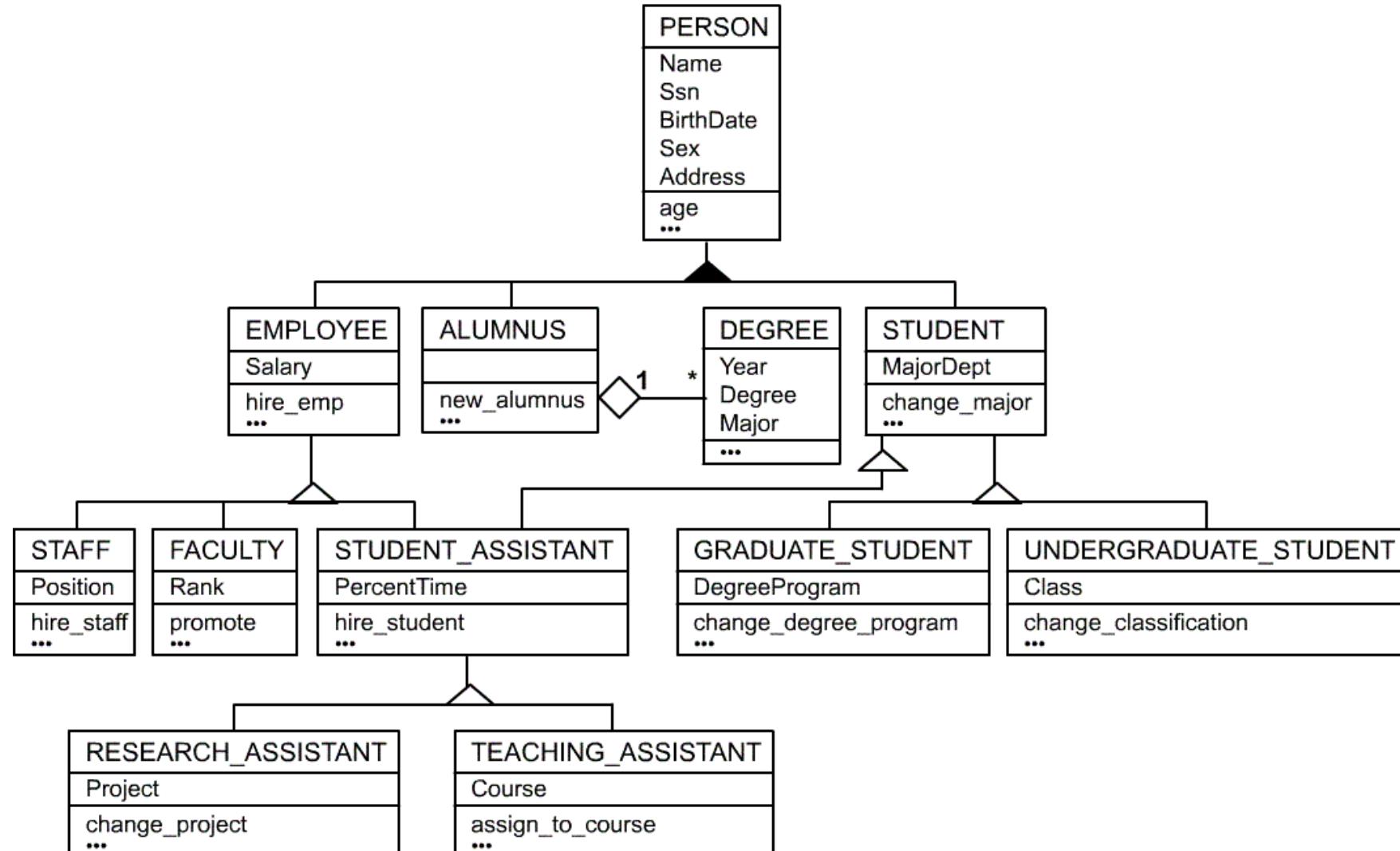
Categories (UNION TYPES)

- Example: Database for vehicle registration, vehicle owner can be a person, a bank (holding a lien on a vehicle) or a company.
- Category (subclass) OWNER is a subset of the union of the three super classes COMPANY, BANK, and PERSON
- A category member must exist in at least one of its super classes

Example of categories (UNION TYPES)



UML Example for Displaying Specialization / Generalization







BITS Pilani

Pilani | Dubai | Goa | Hyderabad

Database Design





Contact Session 3: Relational Model

Session Outline

- Relational Model Concepts
- Relational Model Constraints and Relational Database Schemas
- Update Operations and Dealing with Constraint Violations

Recap-QUIZ

Which type of entity cannot exist in the database unless another type of entity also exists in the database?

- A. Weak entity
- B. Strong entity
- C. Dependent entity
- D. Independent entity

Answer: A - Weak Entity - An entity that depends on some other entity for its existence

Recap-QUIZ

The attribute name could be structured as an attribute consisting of first name, middle initial, and last name. This type of attribute is called

- a) Simple attribute
- b) Composite attribute
- c) Multivalued attribute
- d) Derived attribute

Answer: B - Composite attribute

Recap-QUIZ

_____ express the number of entities to which another entity can be associated via a relationship set.

- a) Mapping Cardinality
- b) Relational Cardinality
- c) Participation Constraints
- d) None of the mentioned

Answer: A – Mapping cardinality - Mapping cardinality is **the maximum number of relationship instances in which an entity can participate**

The entity which is created through specialization is considered as

- a) Medium Class of entity
- b) Mini Class of entity
- c) Sub class of entity
- D) Super class of entity

Answer: Sub class of entity

- Generalization is

ANS: Grouping entity types, bottom up

Recap-Exercise

We wish to create a database for **BITS Data Science trainings**. For this, we must store data about the Participants and the Faculty. For each course participant, identified by an ID, we want to store her social security number, surname, age, sex, place of birth, employer's name, address and telephone number, previous employers (and periods employed), the courses attended and the final assessment for each course. If a participant is self-employed, we need to know her area of expertise, and, if appropriate, her title. For somebody who works for a company, we store the level and position held. Each course has a code and a title and any course can be given any number of times. Each time a particular course is given, we will call it an 'instance' of the course. For each instance, we represent the start date, the end date, and the number of participants. For each Faculty, we will show the SSN, surname, age, place of birth, the instance of the course currently being taught, those taught in the past and the courses that the faculty is qualified to teach. All the faculty's telephone numbers are also stored. An instructor can be permanently employed by BITS or can be a guest faculty.

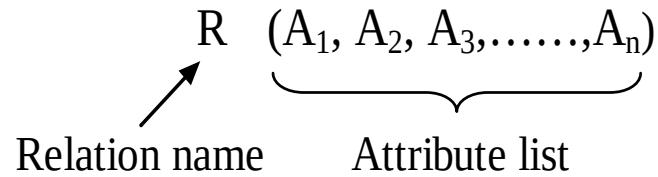
- 1) Identify the entity sets.
- 2) Identify the relationships .
- 3) Identify the structural constraints of all the relationships
- 4) Identify the specializations /generalizations involved.

Relational Model Concepts

- The relational Model of Data is based on the concept of a *Relation*
- A Relation is a mathematical concept based on the ideas of sets
- The model was first proposed by Dr. E.F. Codd of IBM Research in 1970

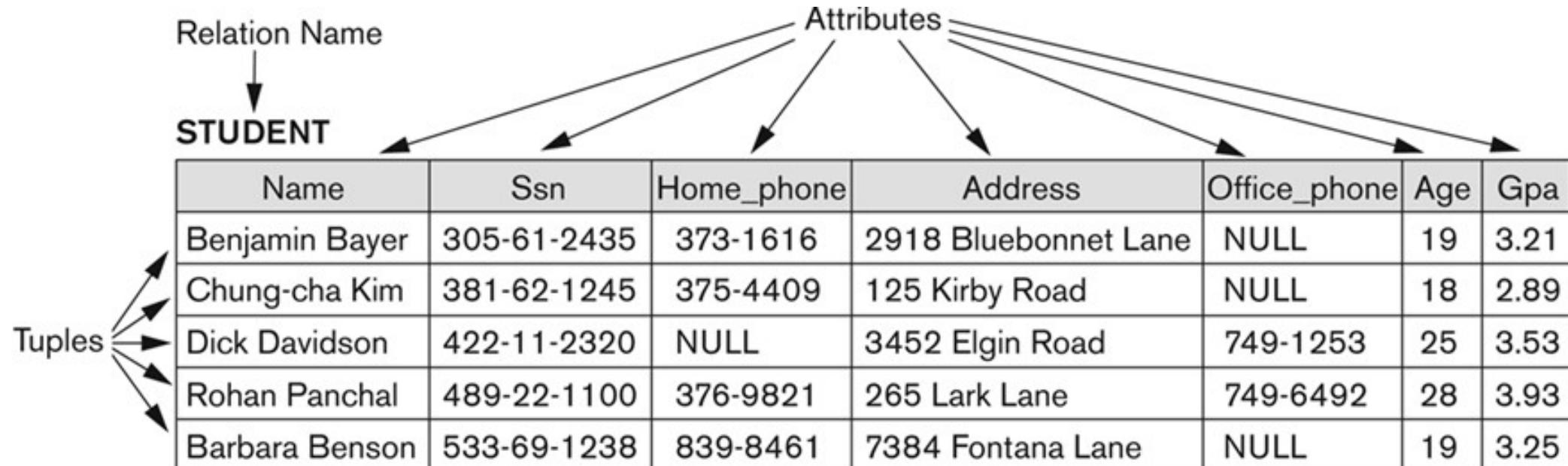
- Using this representational (or implementation) model we represent a database as collection of relations.
- The notion of relation here is different from the notion of relationship used in ER modeling.
- Relation is the main construct for representing data in relational model.
- Every relation consists of a relation schema and Relation instance.

Relation Schema is denoted by



- The number of columns in a relation is known as its ***degree or arity***.
- A relation of degree seven, which stores information about university students, would contain seven attributes describing each student. as follows:
 - STUDENT(Name, Ssn, Home_phone, Address, Office_phone, Age, Gpa)
 - Using the data type of each attribute, the definition is sometimes written as:
 - STUDENT(Name: string, Ssn: string, Home_phone: string, Address: string, Office_phone: string, Age: integer, Gpa: real)

Relational Model Characteristics



Relation instance or Relation State (r) of R (thought of as a table)

- Each *row* in the table represents a collection of related data.
- Each row contains facts about some entity of same entity-set.

$$R = (A_1, A_2, A_3, \dots, A_n)$$

$r(R)$ is a set of n tuples in R

$$r = \{t_1, t_2, t_3, \dots, t_n\}$$

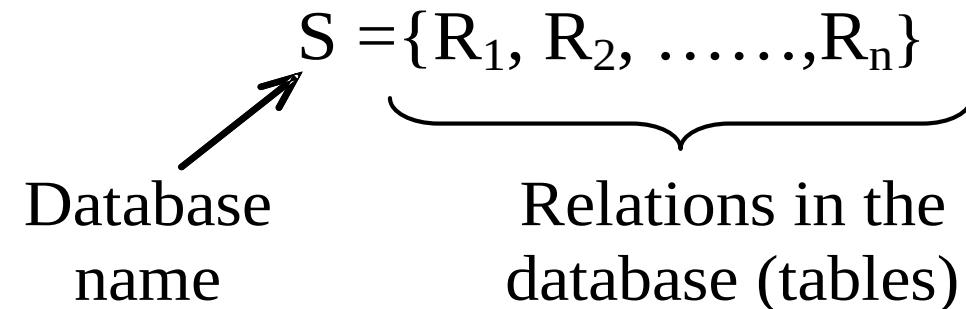
- r is an **instance** of R each t is a tuple and is an ordered list of **values**.
- $t = (v_1, v_2, \dots, v_n)$ where v_i is an element of domain of A_i
- Attribute A of relation R is accessed by notation - $R.A$.
Ex: Student (name, age, branch). Here Student is the relation name.
 Student.age - denotes age attribute of Student relation.

- Entities of each type/set are stored as rows in a single relation.
- Hence in general, a ***relation*** corresponds to a single entity type in ER diagram.
- In some cases a ***relationship*** between two entities can have some specific ***attributes*** which can be captured in a relation (table).
- A row is called a ***tuple***.
- The columns of the table represent attributes of that entity type.
- The column header is known as ***attribute*** or ***field***.
- ***Data type or format*** of an attribute: is the format of data for that attribute. Ex. Character strings, numeric, alphanumeric etc.
- Values that can appear in a column for any row is called the ***domain*** of that attribute.
 - Example: “USA_phone_numbers” are the set of 10 digit phone numbers valid in the U.S.

Relational Database Schema is denoted by

$$S = \{R_1, R_2, \dots, R_n\}$$

Database name Relations in the database (tables)



Relational Model

For example the following is the relation for the

Stu

Primary

Degree = 4 Cardinality = 6

Student

Key

Stname	St#	Department	Year
Sachin	S401	CSE	IV
Dravid	S303	Mech	III
Dhoni	S210	CSE	II
Kohli	S107	Bio	I
Sachin	S101	Production	I

Columns or Attributes

Tuples

Characteristics of a Relation

- Ordering of tuples is not significant.
- Ordering of values in a tuple is important.
- Values in a tuple under each column must be atomic (simple & single).
 - Example Multiple Phone numbers to be stored separately under different columns namely phone #1, phone#2, etc.
- NULL values : used to represent the values of attributes that may be unknown or may not apply to a tuple
- Interpretation (Meaning) of a Relation : a type of assertion
 - An assertion is a statement in SQL that ensures a certain condition will always exist in the database. Assertions are like column and table constraints.

Relational Model Characteristics

- **Key of a Relation:**
 - Each row has a value of a data item (or set of items) that uniquely identifies that row in the table
 - Called the *key*
 - In the STUDENT table, SSN is the key

Relational Model - Keys

Key – An attribute in a relation that is used to distinguish one tuple from another tuple is referred as the Key.

Different types of keys

1. Primary and Secondary keys

If the key uniquely identifies only one tuple/ record from all tuples / record is called as the **Primary Key**.

Otherwise the key is called as the **Secondary key**.

Example: In the above relation **Student**, the attributes Stu# and StName are the Primary and Secondary Keys respectively.

Relational Model - keys

2. Composite or concatenated key

When a table has no primary key for unique identification, we can combine more than one secondary keys to identify unique record/tuple, the key is called as the Composite or Concatenated keys.

Stname	Department	Year
Sachin	CSE	IV
Dravid	Mech	III
Dhoni	CSE	II
Kohli	Bio	I
Sachin	Production	I

StName+Department is
used as the Composite
Key

Relational Model - keys

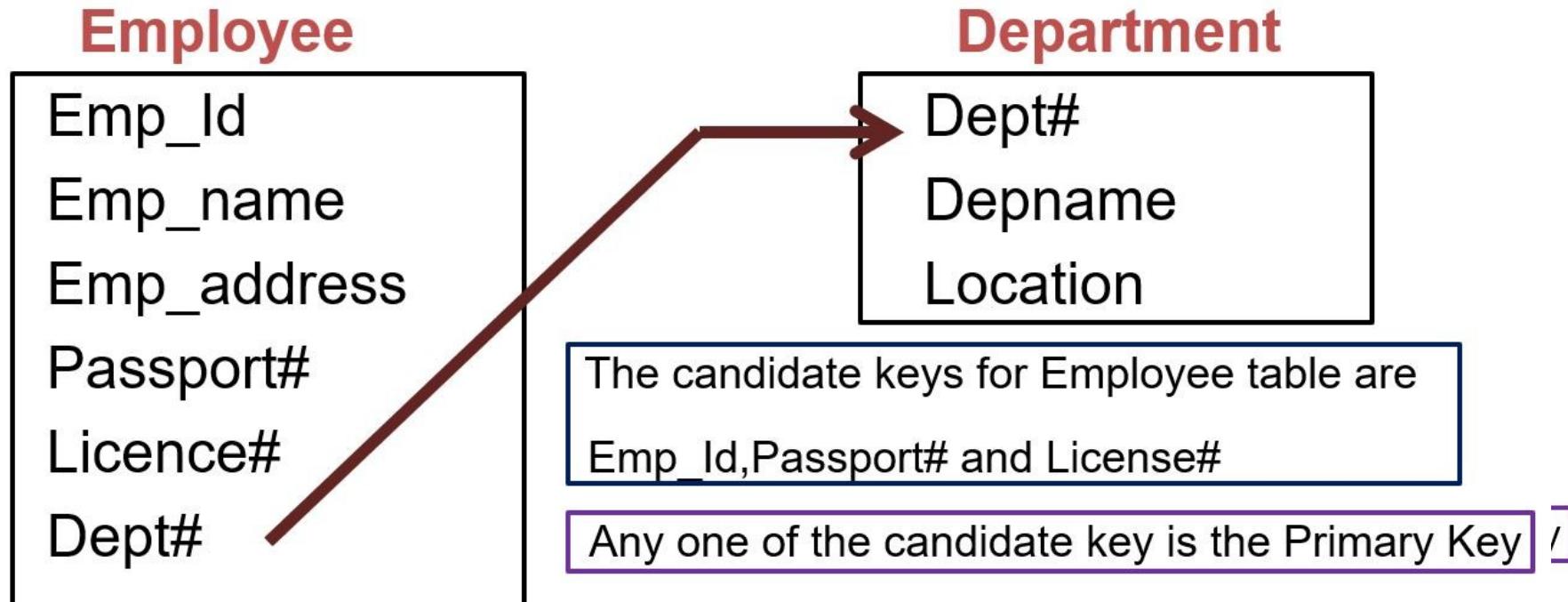
3. Candidate Key – It is an attribute or a set of attributes that can act as a primary key which can uniquely identify the record/tuple in a relation

4. Alternate key – If a relation has two separate primary key attributes say a1 and a2 and one of the attribute is used as the primary key, the other is called as alternate key for the primary key.

5. Foreign Key – It is one of the attribute in a table which is a primary key of another table.

The candidate, alternate and foreign keys are explained in the following tables

Relational Model - keys



Dep# is the foreign Key

Passport# or Licence# is the alternate Key for Emp_Id

Best Practices for creating a Relational Model

- Data need to be represented as a collection of relations
- Each relation should be depicted clearly in the table
- Rows should contain data about instances of an entity
- Columns must contain data about attributes of the entity
- Cells of the table should hold a single value
- Each column should be given a unique name
- No two rows can be identical
- The values of an attribute should be from the same domain

Terminology

Informal Terms	Formal Terms
Table	Relation
Column Header	Attribute
All possible Column Values	Domain
Row	Tuple
Table Definition	Schema of a Relation
Populated Table	State of the Relation

Relational Constraints

Constraints are restrictions on data of a relation.

- *Domain level Constraints* – Format of data Ex. Character numeric etc.
- *Entity integrity constraints* – Primary key, unique key
- *Referential integrity constraints* – Foreign key
- Semantic Constraints.

Relational Integrity Constraints

- Constraints are **conditions** that must hold on **all** valid relation states.
- There are three *main types* of constraints in the relational model:
 - **Key** constraints
 - **Entity integrity** constraints
 - **Referential integrity** constraints
- Another implicit constraint is the **domain** constraint
 - Every value in a tuple must be from the *domain of its attribute* (or it could be **null**, if allowed for that attribute)

Key Constraints

- **Superkey** of R:
 - Is a set of attributes SK of R with the following condition:
 - No two tuples in any valid relation state $r(R)$ will have the same value for SK
 - That is, for any distinct tuples t_1 and t_2 in $r(R)$, $t_1[SK] \neq t_2[SK]$
 - This condition must hold in *any valid state* $r(R)$
- **Key** of R:
- It is used to uniquely identify any record or row of data from the table.
 - ID is used as a key in the Student table because it is unique for each student. In the PERSON table, passport_number, license_number, SSN are keys since they are unique for each person.
- It is also used to establish and identify relationships between tables.
 - A key is a "minimal" superkey
 - That is, a key is a superkey K such that removal of any attribute from K results in a set of attributes that is not a superkey (does not possess the superkey uniqueness property)

Key Constraints (continued)

- Example: Consider the CAR relation schema
 - CAR(State, Reg#, SerialNo, Make, Model, Year)
 - CAR has two keys:
 - Key1 = {State, Reg#}
 - Key2 = {SerialNo}
 - Both are also superkeys of CAR
 - {SerialNo, Make} is a superkey but *not* a key.
- In general:
 - Any *key* is a *superkey* (but not vice versa)
 - Any set of attributes that *includes a key* is a *superkey*
 - A *minimal* superkey is also a key.

» Further Reading: <https://www.javatpoint.com/dbms-keys#:~:text=Super%20key%20is%20an%20attribute,can%20also%20be%20a%20key.>

CAR

License_number	Engine_serial_number	Make	Model	Year
Texas ABC-739	A69352	Ford	Mustang	02
Florida TVP-347	B43696	Oldsmobile	Cutlass	05
New York MPO-22	X83554	Oldsmobile	Delta	01
California 432-TFY	C43742	Mercedes	190-D	99
California RSK-629	Y82935	Toyota	Camry	04
Texas RSK-629	U028365	Jaguar	XJS	04

Key Constraints (continued)

- If a relation has several **candidate keys**, one is chosen arbitrarily to be the **primary key**.
 - The primary key attributes are underlined
- Example: Consider the CAR relation schema
 - CAR(State, Reg#, SerialNo, Make, Model, Year)
 - We chose SerialNo as the primary key

CAR

License_number	Engine_serial_number	Make	Model	Year
Texas ABC-739	A69352	Ford	Mustang	02
Florida TVP-347	B43696	Oldsmobile	Cutlass	05
New York MPO-22	X83554	Oldsmobile	Delta	01
California 432-TFY	C43742	Mercedes	190-D	99
California RSK-629	Y82935	Toyota	Camry	04
Texas RSK-629	U028365	Jaguar	XJS	04

- The primary key value is used to *uniquely identify* each tuple in a relation
 - Provides the tuple identity
- Also used to *reference* the tuple from another tuple
 - General rule: Choose as primary key the **smallest** of the candidate keys (in terms of size)
 - Not always applicable – choice is sometimes subjective

Exercise-1

Consider the relation

CLASS(Course#, Univ_Section#, Instructor_name, Semester, Building_code, Room#, Time_period, Weekdays, Credit_hours).

This represents classes taught in a university, with unique Univ_section#s. Identify what you think should be various candidate keys with justification.

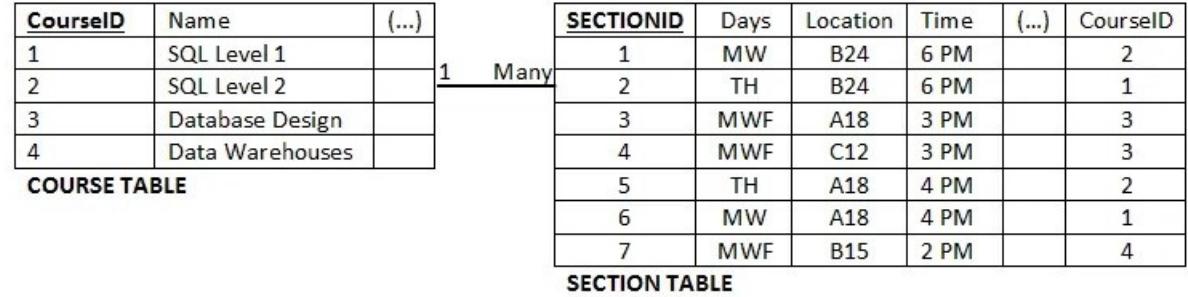
- **Entity Integrity:**
 - The *primary key attributes* PK of each relation schema R in S cannot have null values in any tuple of $r(R)$.
 - This is because primary key values are used to *identify* the individual tuples.
 - $t[PK] \neq \text{null}$ for any tuple t in $r(R)$
 - If PK has several attributes, null is not allowed in any of these attributes
 - Note: Other attributes of R may be constrained to disallow null values, even though they are not members of the primary key.

- A constraint involving **two** relations
- Used to specify a **relationship** among tuples in two relations:
 - The **referencing relation** and the **referenced relation**.

Referential Integrity

- Tuples in the **referencing relation** R1 have attributes FK (called **foreign key** attributes) that reference the primary key attributes PK of the **referenced relation** R2.
 - A tuple t_1 in R1 is said to **reference** a tuple t_2 in R2 if $t_1[FK] = t_2[PK]$.
- A referential integrity constraint can be displayed in a relational database schema as a directed arc from R1.FK to R2.

Referential Integrity



COURSE TABLE

CourseID	Name	(...)
1	SQL Level 1	
2	SQL Level 2	
3	Database Design	
4	Data Warehouses	

SECTION TABLE

SECTIONID	Days	Location	Time	(...)	CourseID
1	MW	B24	6 PM		2
2	TH	B24	6 PM		1
3	MWF	A18	3 PM		3
4	MWF	C12	3 PM		3
5	TH	A18	4 PM		2
6	MW	A18	4 PM		1
7	MWF	B15	2 PM		4

Example from a Course/Class database.

- Course(**CrsCode**, DeptCode, Description)
- Class(**CrsCode**, **Section**, ClassTime)
- The referential integrity constraint states that CrsCode in the Class table must match a valid CrsCode in the Course table.
- In this situation, it's not enough that the CrsCode and Section in the Class table make up the PK, we must also enforce referential integrity.
- When setting up referential integrity it is important that the PK and FK have the same data types and come from the same domain, otherwise the relational database management system (RDBMS) will not allow the join.

Referential Integrity (or foreign key) Constraint



- Statement of the constraint
 - The value in the foreign key column (or columns) FK of the **referencing relation** R1 can be **either**:
 - (1) a value of an existing primary key value of a corresponding primary key PK in the **referenced relation** R2, or
 - (2) a **null**.
 - In case (2), the FK in R1 should **not** be a part of its own primary key.

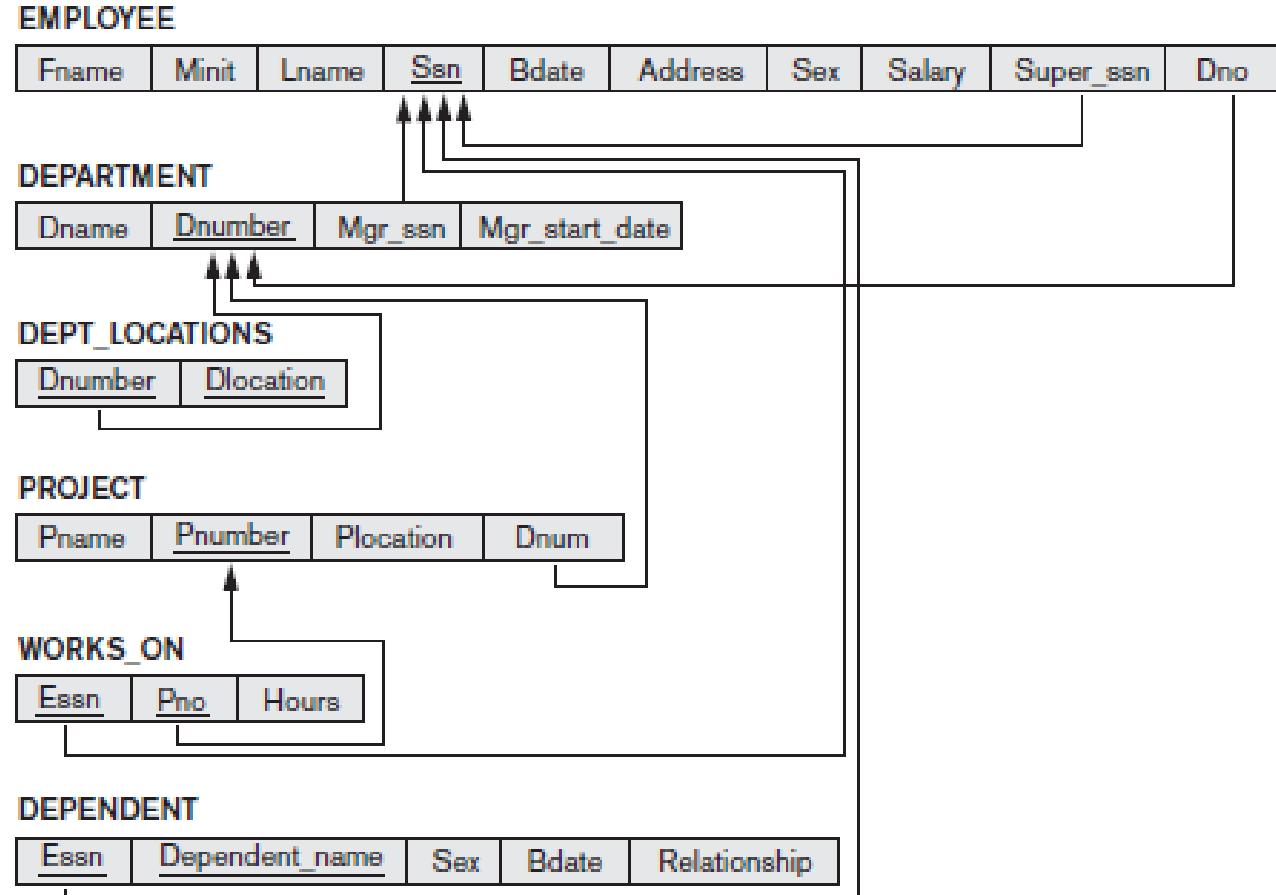
Other Types of Constraints

- Semantic Integrity Constraints:
 - based on application semantics (business rules, standards, constraints) and cannot be expressed by the model
 - Example: “the max. no. of hours per employee for all projects he or she works on is 56 hrs per week”

Displaying a relational database schema and its constraints

- Each relation schema can be displayed as a row of attribute names
- The name of the relation is written above the attribute names
- The primary key attribute (or attributes) will be underlined
- A foreign key (referential integrity) constraints is displayed as a directed arc (arrow) from the foreign key attributes to the referenced table
 - Can also point the primary key of the referenced relation for clarity

COMPANY DB-Relational Model



Exercise-2-Relational Model



- Consider the following relations for a database that keeps track of business trips of salespersons in a sales office:
 - SALESPERSON(Ssn, Name, Start_year, Dept_no)
 - TRIP(Ssn, From_city, To_city, Departure_date, Return_date, Trip_id)
 - EXPENSE(Trip_id, Account#, Amount)
- A trip can be charged to one or more accounts. Specify the foreign keys for this schema, stating any assumptions you make. Draw the Relational model

Consider the following relations for a database that keeps track of student enrollment in courses and the books adopted for each course:

STUDENT(SSN, Name, Major, Bdate)

COURSE(Course#, Cname, Dept)

ENROLL(SSN, Course#, Quarter, Grade)

BOOK_ADOPTION(Course#, Quarter, Book_ISBN)

TEXT(Book_ISBN, Book_Title, Publisher, Author)

Draw a relational schema diagram specifying the foreign keys for this schema.

Operations on Relations and constraints



- The following operations update and modify the database
- INSERT a tuple.
- DELETE a tuple.
- MODIFY a tuple.
- Integrity constraints should not be violated by the update operations.
- Several update operations may have to be grouped together.
- Updates may **propagate** (AKA cascading) to cause other updates automatically.
- This may be necessary to maintain integrity constraints.

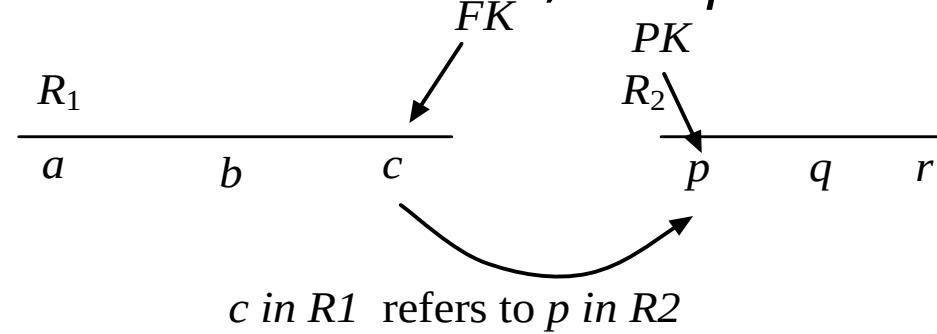
Update Operations on Relations



- In case of integrity violation, several actions can be taken:
 - Cancel the operation that causes the violation (RESTRICT or REJECT option)
 - Perform the operation but inform the user of the violation
 - Trigger additional updates so the violation is corrected (CASCADE option, SET NULL option)
 - Execute a user-specified error-correction routine

Operations on Relations and constraints

Actions need to be taken when FK is set , on operations like update, insert, and delete.



- To insert a tuple in R_1 where the value for c is not in p of R_2 , then don't allow.
- What if a tuple in R_2 is deleted:
Cascade, don't allow, set to default, set to null.
- What if update on R_2 's p happens:
Cascade, don't allow, set to default, set to null.

Operations on Relations and constraints



Operation on relations	Constraints need to be taken care
Insert	Null, Not Null, PK, unique, FK, format, Domain
Delete	FK
Update	Null, Not Null, PK, unique, FK, domain, and Semantic

Operations on Relations and constraints



Emp				Dept		
eid	ename	age	dno	dnum	dname	dloc
101	Kiran	28	10	10	Payroll	Che
105	Gopu	32	20	20	Prod	Hyd
121	Ahmed	29	20	30	Acct	Bgl
136	John	28	20			
138	Mike	30	30			
143	Ali	38	10			

Op : Care for constraints.

Insert :PK, Unique, null, notnull, format, FK

Update :PK, Unique, null, notnull, format, FK

Delete :No care

Op : Care for constraints.

Insert : PK, Unique, null, notnull, format

Update : PK, Unique, null, notnull, format, FK

Delete : FK

Possible violations for each operation

- INSERT may violate any of the constraints:
 - Domain constraint:
 - if one of the attribute values provided for the new tuple is not of the specified attribute domain
 - Key constraint:
 - if the value of a key attribute in the new tuple already exists in another tuple in the relation
 - Referential integrity:
 - if a foreign key value in the new tuple references a primary key value that does not exist in the referenced relation
 - Entity integrity:
 - if the primary key value is null in the new tuple

Possible violations for each operation

- DELETE may violate only referential integrity:
 - If the primary key value of the tuple being deleted is referenced from other tuples in the database
 - Can be remedied by several actions: RESTRICT, CASCADE, SET NULL
 - RESTRICT option: reject the deletion
 - CASCADE option: propagate the new primary key value into the foreign keys of the referencing tuples
 - SET NULL option: set the foreign keys of the referencing tuples to NULL
 - One of the above options must be specified during database design for each foreign key constraint

Possible violations for each operation

- UPDATE may violate domain constraint and NOT NULL constraint on an attribute being modified
- Any of the other constraints may also be violated, depending on the attribute being updated:
 - Updating the primary key (PK):
 - Similar to a DELETE followed by an INSERT
 - Need to specify similar options to DELETE
 - Updating a foreign key (FK):
 - May violate referential integrity
 - Updating an ordinary attribute (neither PK nor FK):
 - Can only violate domain constraints

Exercise-4

- Suppose that each of the following Update operations is applied directly to the database state shown below. Discuss *all* integrity constraints violated by each operation, if any, and the different ways of enforcing these constraints.

EMPLOYEE

Fname	Minit	Lname	Ssn	Bdate	Address	Sex	Salary	Super_ssn	Dno
John	B	Smith	123456789	1965-01-09	731 Fondren, Houston, TX	M	30000	333445555	5
Franklin	T	Wong	333445555	1955-12-08	638 Voss, Houston, TX	M	40000	888665555	5
Alicia	J	Zelaya	999887777	1968-01-19	3321 Castle, Spring, TX	F	25000	987654321	4
Jennifer	S	Wallace	987654321	1941-06-20	291 Berry, Bellaire, TX	F	43000	888665555	4
Ramesh	K	Narayan	666884444	1962-09-15	975 Fire Oak, Humble, TX	M	38000	333445555	5
Joyce	A	English	453453453	1972-07-31	5631 Rice, Houston, TX	F	25000	333445555	5
Ahmad	V	Jabbar	987987987	1969-03-29	980 Dallas, Houston, TX	M	25000	987654321	4
James	E	Borg	888665555	1937-11-10	450 Stone, Houston, TX	M	55000	NULL	1

WORKS_ON

Essn	Pho	Hours
123456789	1	32.5
123456789	2	7.5
666884444	3	40.0
453453453	1	20.0
453453453	2	20.0
333445555	2	10.0

- Insert <'Robert', 'F', 'Scott', '943775543', '1972-06-21', '2365 Newcastle Rd, Bellaire, TX', M, 58000, '888665555', 1> into EMPLOYEE.
- Delete the EMPLOYEE tuple with Ssn = '123456789'.

Exercise-5

Suppose that each of the following Update operations is applied directly to the database state shown below. Discuss *all* integrity constraints violated by each operation, if any, and the different ways of enforcing these constraints.

PROJECT

Pname	Pnumber	Plocation	Dnum
ProductX	1	Bellaire	5
ProductY	2	Sugarland	5
ProductZ	3	Houston	5
Computerization	10	Stafford	4
Reorganization	20	Houston	1
Newbenefits	30	Stafford	4

DEPARTMENT

Dname	Dnumber	Mgr_ssn	Mgr_start_date
Research	5	333445555	1988-05-22
Administration	4	987654321	1995-01-01
Headquarters	1	888665555	1981-06-19

Insert <'ProductA', 4, 'Bellaire', 2> into PROJECT.

Insert <'Production', 4, '943775543', '2007-10-01'> into DEPARTMENT.

Exercise-6

Suppose that each of the following Update operations is applied directly to the database state shown below. Discuss *all* integrity constraints violated by each operation, if any, and the different ways of enforcing these constraints.

WORKS_ON

<u>Essn</u>	<u>Pno</u>	Hours
123456789	1	32.5
123456789	2	7.5
666884444	3	40.0
453453453	1	20.0
453453453	2	20.0
333445555	2	10.0

EMPLOYEE

Fname	Minit	Lname	<u>Ssn</u>	Bdate	Address	Sex	Salary	Super_ssn	Dno
John	B	Smith	123456789	1965-01-09	731 Fondren, Houston, TX	M	30000	333445555	5
Franklin	T	Wong	333445555	1955-12-08	638 Voss, Houston, TX	M	40000	888665555	5
Alicia	J	Zelaya	999887777	1968-01-19	3321 Castle, Spring, TX	F	25000	987654321	4
Jennifer	S	Wallace	987654321	1941-06-20	291 Berry, Bellaire, TX	F	43000	888665555	4
Ramesh	K	Narayan	666884444	1962-09-15	975 Fire Oak, Humble, TX	M	38000	333445555	5
Joyce	A	English	453453453	1972-07-31	5631 Rice, Houston, TX	F	25000	333445555	5
Ahmad	V	Jabbar	987987987	1969-03-29	980 Dallas, Houston, TX	M	25000	987654321	4
James	E	Borg	888665555	1937-11-10	450 Stone, Houston, TX	M	55000	NULL	1

Insert <'677678989', NULL, '40.0'> into WORKS_ON.

Suppose that each of the following Update operations is applied directly to the database state shown below. Discuss *all* integrity constraints violated by each operation, if any, and the different ways of enforcing these constraints.

DEPENDENT

<u>Essn</u>	<u>Dependent_name</u>	Sex	Bdate	Relationship
333445555	Alice	F	1986-04-05	Daughter
333445555	Theodore	M	1983-10-25	Son
333445555	Joy	F	1958-05-03	Spouse
987654321	Abner	M	1942-02-28	Spouse
123456789	Michael	M	1988-01-04	Son
123456789	Alice	F	1988-12-30	Daughter
123456789	Elizabeth	F	1967-05-05	Spouse

WORKS_ON

<u>Essn</u>	<u>Pno</u>	Hours
123456789	1	32.5
123456789	2	7.5
666884444	3	40.0
453453453	1	20.0
453453453	2	20.0
333445555	2	10.0

Insert <'453453453', 'John', 'M', '1990-12-12', 'spouse'> into DEPENDENT.
Delete the WORKS_ON tuples with Essn = '333445555'.

- Suppose that each of the following Update operations is applied directly to the database state shown below. Discuss *all* integrity constraints violated by each operation, if any, and the different ways of enforcing these constraints.

PROJECT

Pname	<u>Pnumber</u>	Plocation	Dnum
ProductX	1	Bellaire	5
ProductY	2	Sugarland	5
ProductZ	3	Houston	5
Computerization	10	Stafford	4
Reorganization	20	Houston	1
Newbenefits	30	Stafford	4

DEPARTMENT

Dname	<u>Dnumber</u>	Mgr_ssn	Mgr_start_date
Research	5	333445555	1988-05-22
Administration	4	987654321	1995-01-01
Headquarters	1	888665555	1981-06-19

- Delete the PROJECT tuple with Pname = 'ProductX'.
- Modify the Mgr_ssn and Mgr_start_date of the DEPARTMENT tuple with Dnumber = 5 to '123456789' and '2007-10-01', respectively.

Exercise-9

- Suppose that each of the following Update operations is applied directly to the database state shown below. Discuss *all* integrity constraints violated by each operation, if any, and the different ways of enforcing these constraints.

EMPLOYEE

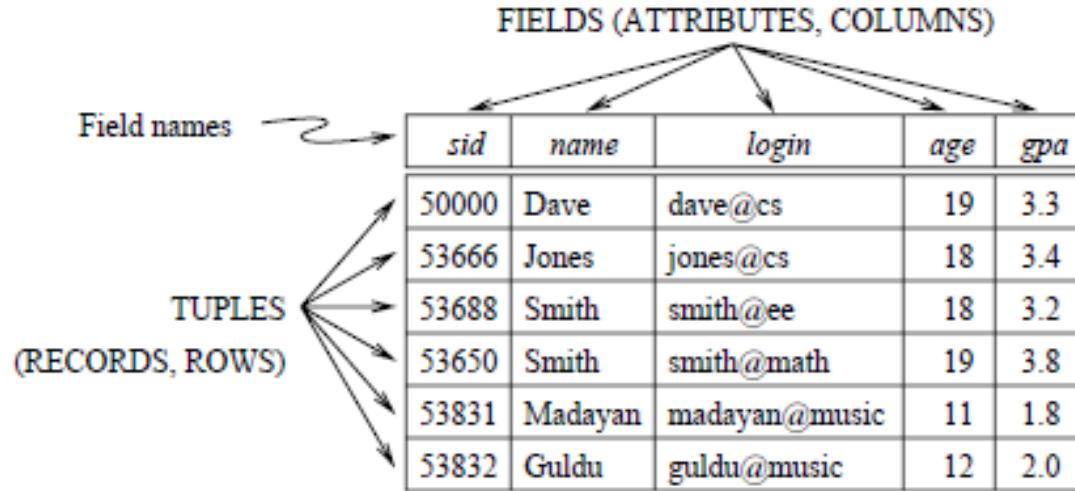
Fname	Minit	Lname	Ssn	Bdate	Address	Sex	Salary	Super_ssn	Dno
John	B	Smith	123456789	1965-01-09	731 Fondren, Houston, TX	M	30000	333445555	5
Franklin	T	Wong	333445555	1955-12-08	638 Voss, Houston, TX	M	40000	888665555	5
Alicia	J	Zelaya	999887777	1968-01-19	3321 Castle, Spring, TX	F	25000	987654321	4
Jennifer	S	Wallace	987654321	1941-06-20	291 Berry, Bellaire, TX	F	43000	888665555	4
Ramesh	K	Narayan	666884444	1962-09-15	975 Fire Oak, Humble, TX	M	38000	333445555	5
Joyce	A	English	453453453	1972-07-31	5631 Rice, Houston, TX	F	25000	333445555	5
Ahmad	V	Jabbar	987987987	1969-03-29	980 Dallas, Houston, TX	M	25000	987654321	4
James	E	Borg	888665555	1937-11-10	450 Stone, Houston, TX	M	55000	NULL	1

WORKS_ON

Essn	Pno	Hours
123456789	1	32.5
123456789	2	7.5
666884444	3	40.0
453453453	1	20.0
453453453	2	20.0
333445555	2	10.0

- Modify the Super_ssn attribute of the EMPLOYEE tuple with Ssn ='999887777' to '943775543'.
- Modify the Hours attribute of the WORKS_ON tuple with Essn ='999887777' and Pno = 10 to '5.0'.

Exercise-10



Give an example of an attribute (or set of attributes) that you can deduce is not a candidate key, based on this instance being legal.

Is there any example of an attribute (or set of attributes) that you can deduce is a candidate key, based on this instance being legal?

Presented Relational Model Concepts

- Definitions
- Characteristics of relations

Discussed Relational Model Constraints and Relational Database Schemas

- **Domain constraints'** - It makes sure that the data value entered for that particular column matches with the data type defined for that column .
- **Key constraints** - A unique constraint (also referred to as a unique key constraint) is a rule that forbids duplicate values in one or more columns within a table.
- A primary key constraint is a column or combination of columns that has the same properties as a unique constraint.
- **Entity integrity** - Entity integrity is concerned with ensuring that each row of a table has a unique and non-null primary key value;
- **Referential integrity** - You can use a primary key and foreign key constraints to define relationships between tables.
- The reference from a row in one table to another table must be valid.

Described the Relational Update Operations and Dealing with Constraint

Violations



BITS Pilani

Pilani | Dubai | Goa | Hyderabad

Database Design



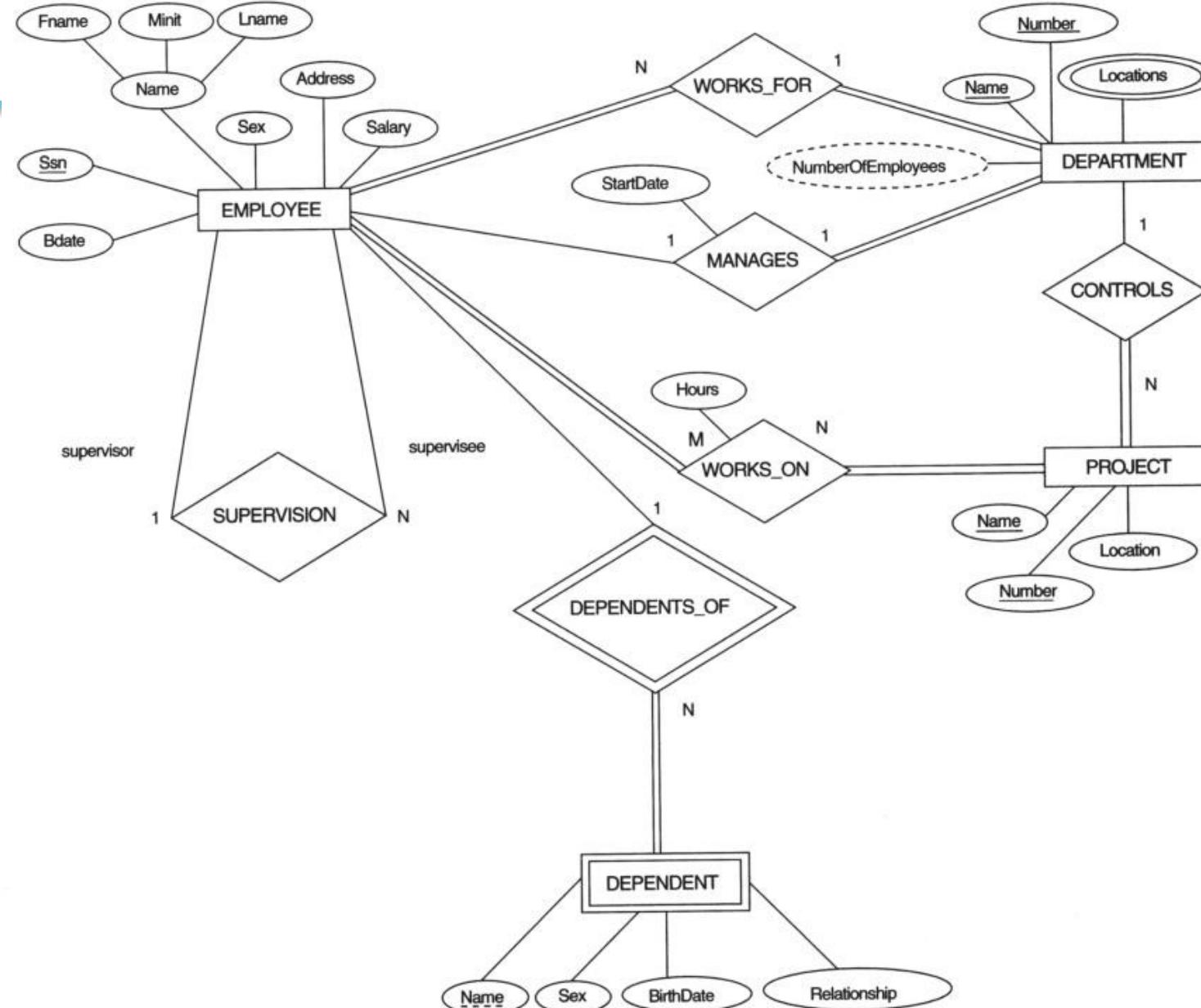


Contact Session 4: ER to Relational Mapping

ER to Relational Mapping

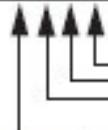
- 1.Mapping rules/guidelines for mapping ER constructs
- 2.Mapping rules/guidelines for mapping hierarchies
- 3.Examples

Entity Relationship Model: **COMPANY** database.



EMPLOYEE

Fname	Minit	Lname	<u>Ssn</u>	Bdate	Address	Sex	Salary	Super_ssn	Dno
-------	-------	-------	------------	-------	---------	-----	--------	-----------	-----


DEPARTMENT

Dname	<u>Dnumber</u>	Mgr_ssn	Mgr_start_date
-------	----------------	---------	----------------


DEPT_LOCATIONS

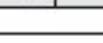
<u>Dnumber</u>	Dlocation
----------------	-----------

PROJECT

Pname	<u>Pnumber</u>	<u>Plocation</u>	Dnum
-------	----------------	------------------	------


WORKS_ON

<u>Essn</u>	<u>Pno</u>	Hours
-------------	------------	-------


DEPENDENT

<u>Essn</u>	Dependent_name	Sex	Bdate	Relationship
-------------	----------------	-----	-------	--------------

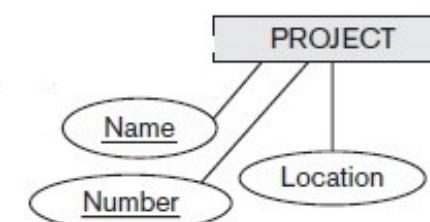
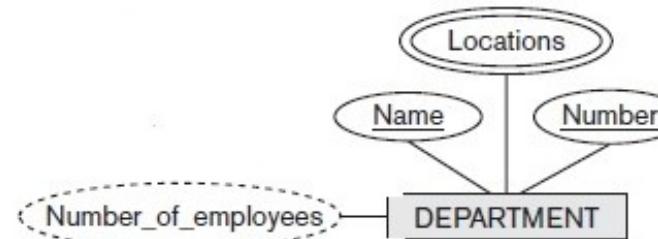
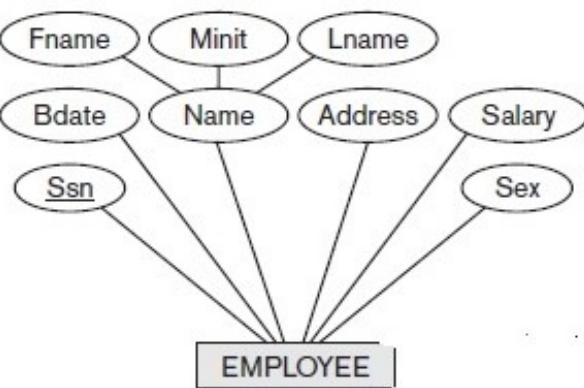
Result of mapping the COMPANY ER schema into a relational database schema.

Step 1: Mapping of Regular Entity Types

- For each regular (strong) entity type E in the ER schema, create a relation R that includes all the simple attributes of E .
- Include only the simple component attributes of a composite attribute
- Choose one of the key attributes of E as the primary key for R

Step 1: Mapping of Regular Entity Types

- If the chosen key of E is a composite, then the set of simple attributes that form it will together form the primary key of R
- If multiple keys were identified for E during the conceptual design, the information describing the attributes that form each additional key is kept in order to specify secondary (unique) keys of relation R.



EMPLOYEE

Fname	Minit	Lname	Ssn	Bdate	Address	Sex	Salary
-------	-------	-------	-----	-------	---------	-----	--------

DEPARTMENT

Dname	Dnumber
-------	---------

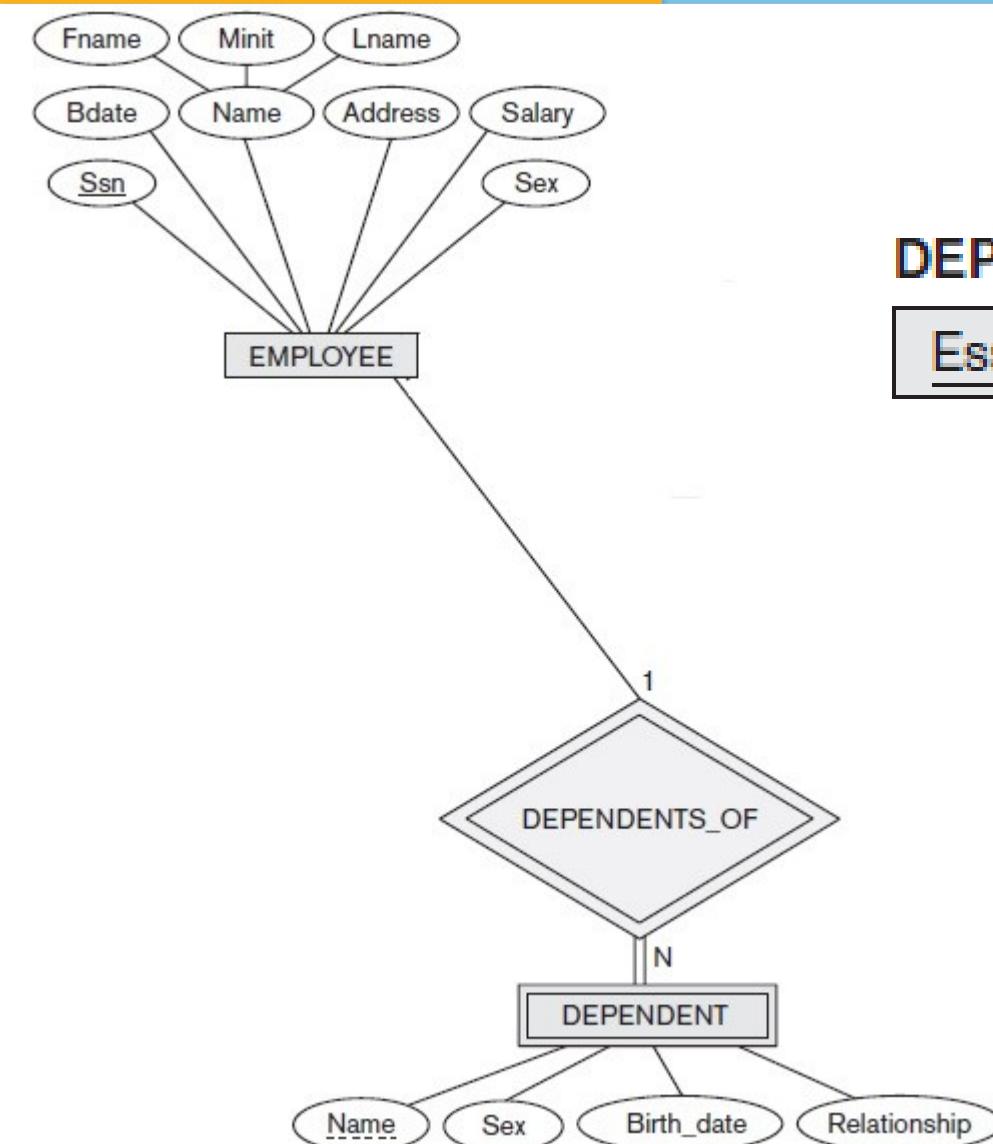
PROJECT

Pname	Pnumber	Plocation
-------	---------	-----------

Step 2: Mapping of Weak Entity Types

- For each weak entity type W in the ER schema with owner entity type E , create a relation R and include all simple attributes (or simple components of composite attributes) of W as attributes of R .
- In addition, include primary key of E as foreign key attributes of R (this takes care of mapping the identifying relationship type of W)
- The primary key of R is the combination of the primary key(s) of the owner(s) and the partial key of the weak entity type W (if any)

ER-to-Relational Mapping Algorithm



DEPENDENT

<u>Essn</u>	<u>Dependent_name</u>	<u>Sex</u>	<u>Bdate</u>	<u>Relationship</u>
-------------	-----------------------	------------	--------------	---------------------

Step 2: Mapping of Weak Entity Types

- If there is a weak entity type E2 whose owner is also a weak entity type E1, then E1 should be mapped before E2 to determine its primary key first
- It is common to choose the propagate (CASCADE) option for the referential triggered action on the foreign key in the relation corresponding to the weak entity type, since a weak entity has an existence dependency on its owner entity. This can be used for both ON UPDATE and ON DELETE.

Step 3: Mapping of Binary 1:1 Relationship Types

- For each binary 1:1 relationship type R in the ER schema, identify the relations S and T that correspond to the entity types participating in R .
- There are three possible approaches to represent this relationship types:
 - 1) Foreign key approach
 - 2) Merged relationship approach
 - 3) Cross-reference or relationship relation approach

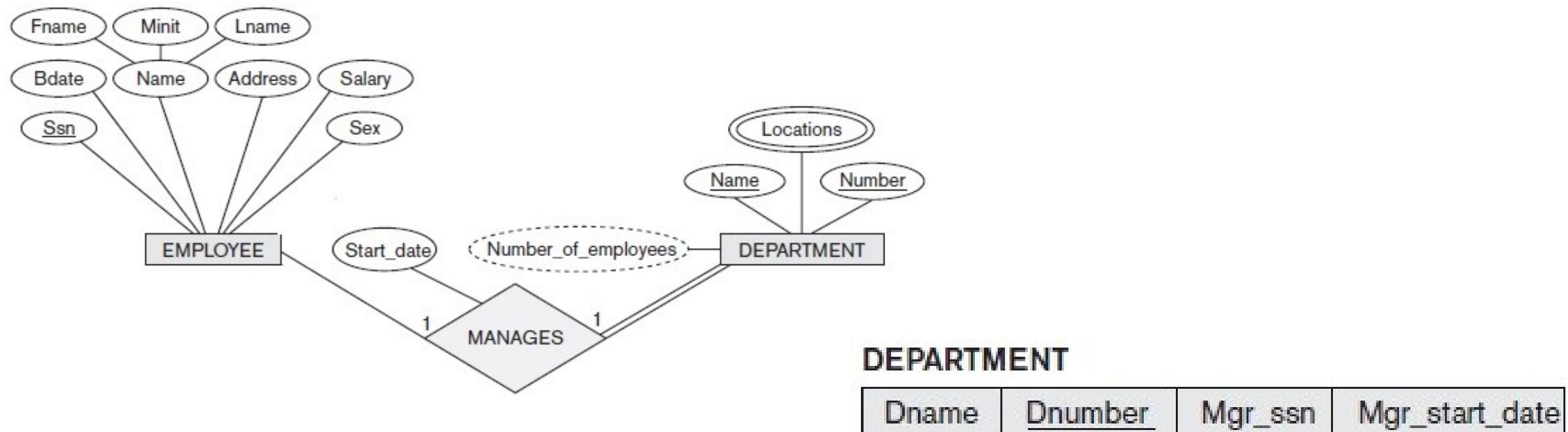
Step 3: Mapping of Binary 1:1 Relationship Types

1) Foreign key approach:

- Choose one of the relations—S, and include as a foreign key in S the primary key of T
- It is better to choose an entity type with total participation in R in the role of S
- Include all the simple attributes of the 1:1 relationship type R as attributes of S.

Step 3: Mapping of Binary 1:1 Relationship Types

1) Foreign key approach:



Step 3: Mapping of Binary 1:1 Relationship Types

2) Merged relationship approach :

- Mapping of a 1:1 relationship type is to merge the two entity types and the relationship into a single relation.
- This is possible when both participations are total, as this would indicate that the two tables will have the exact same number of tuples at all times.

Step 3: Mapping of Binary 1:1 Relationship Types

3) Cross-reference or relationship relation approach :

- set up a third relation R for the purpose of cross-referencing the primary keys of the two relations S and T representing the entity types.
- The relation R is called a relationship relation (or sometimes a lookup table), because each tuple in R represents a relationship instance that relates one tuple from S with one tuple from T.

Step 3: Mapping of Binary 1:1 Relationship Types

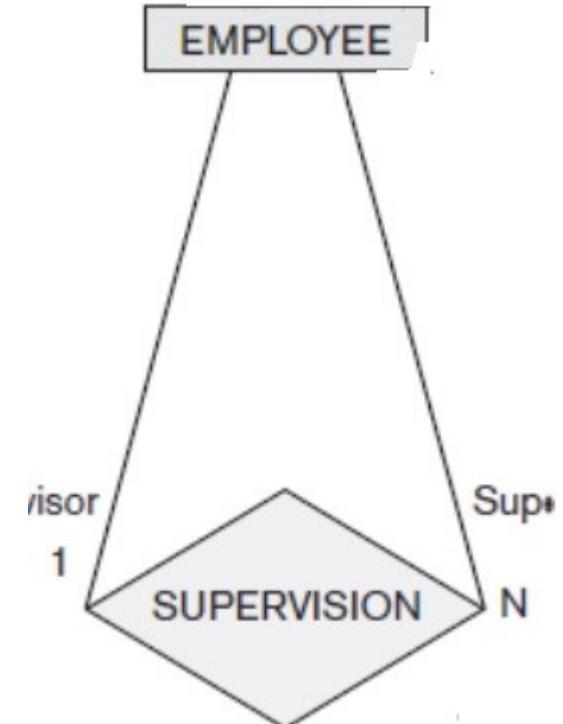
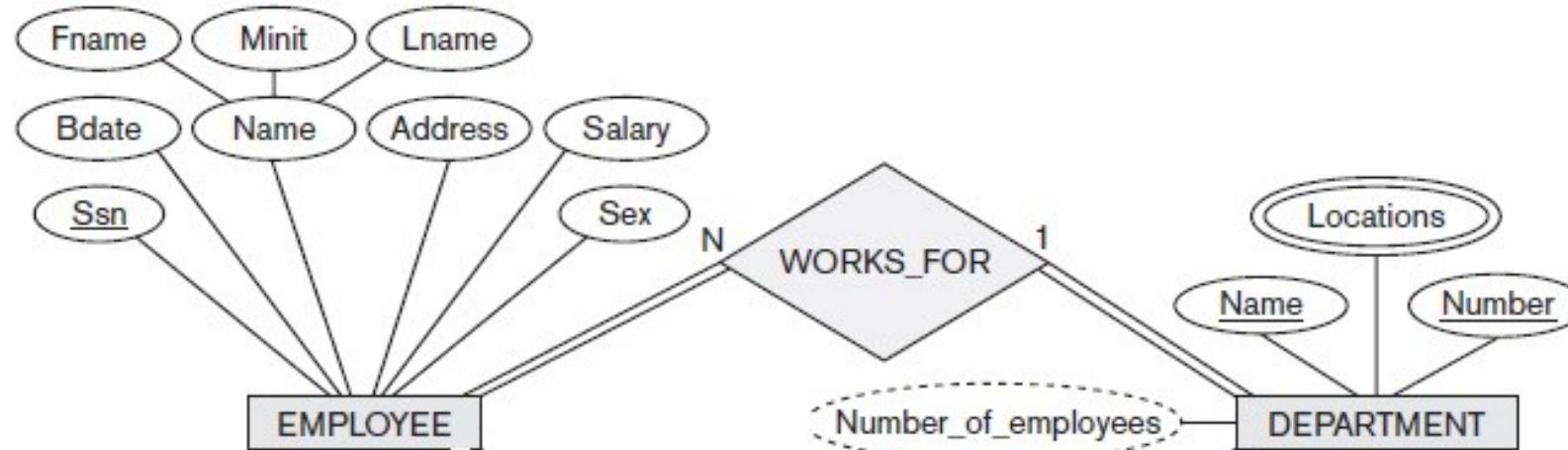
3) Cross-reference or relationship relation approach :

- The relation R will include the primary key attributes of S and T as foreign keys.
- The primary key of R will be one of the two foreign keys, and the other foreign key will be a unique key of R.
- The drawback is having an extra relation, and requiring an extra join operation when combining related tuples from the tables.

Step 4: Mapping of Binary 1:N Relationship Types

- For each regular binary 1:N relationship type R, identify the relation S that represents the participating entity type at the N-side of the relationship type
- Include as foreign key in S the primary key of the relation T that represents the other entity type participating in R
- Include any simple attributes of the 1:N relationship type as attributes of S.

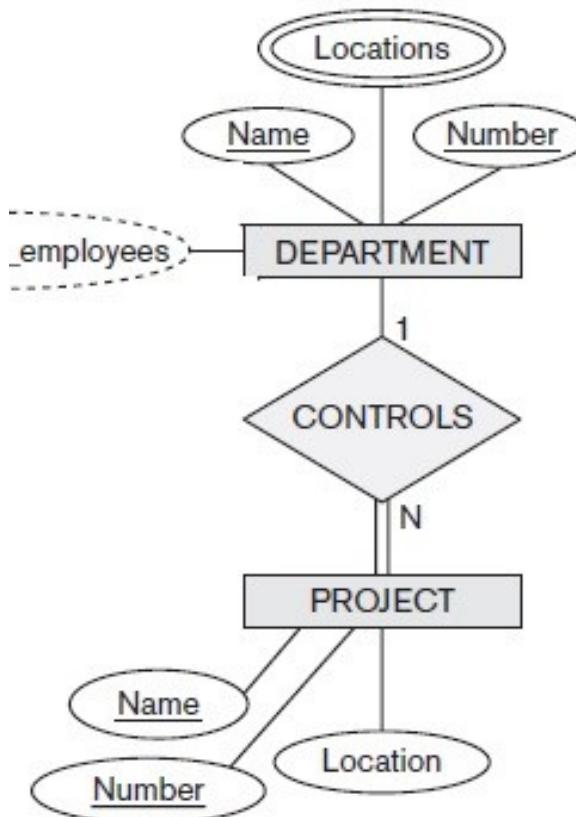
Step 4: Mapping of Binary 1:N Relationships in Tuples



EMPLOYEE

Fname	Minit	Lname	Ssn	Bdate	Address	Sex	Salary	Super_ssn	Dno
-------	-------	-------	-----	-------	---------	-----	--------	-----------	-----

Step 4: Mapping of Binary 1:N Relationship Types



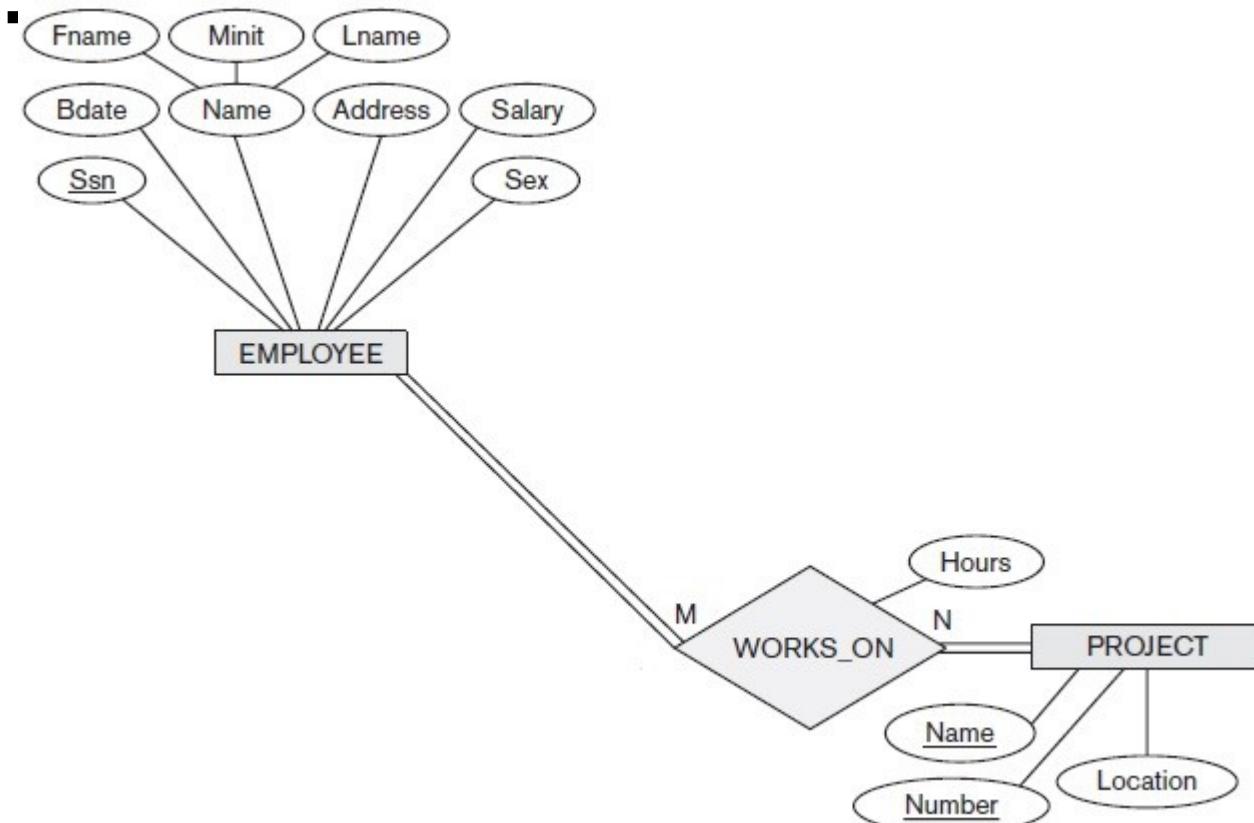
PROJECT

Pname	Pnumber	Plocation	Dnum
-------	---------	-----------	------

Step 5: Mapping of Binary M:N Relationship Types

- For each binary M:N relationship type R, create a new relation S to represent R.
- Include as foreign key attributes in S the primary keys of the relations that represent the participating entity types
- Their combination will form the primary key of S.
- Also include any simple attributes of the M:N relationship type as attributes of S.
- we must create a separate relationship relation S.

Step 5: Mapping of Binary M:N Relationship



WORKS_ON

<u>Essn</u>	<u>Pno</u>	Hours
-------------	------------	-------

Step 5: Mapping of Binary M:N Relationship Types

- The propagate (CASCADE) option for the referential triggered action should be specified on the foreign keys in the relation corresponding to the relationship R
- This is due to each relationship instance has an existence dependency on each of the entities it relates.
- This can be used for both ON UPDATE and ON DELETE.

Step 5: Mapping of Binary M:N Relationship Types

- Notice that we can always map 1:1 or 1:N relationships in a manner similar to M:N relationships by using the cross-reference (relationship relation) approach.
- This alternative is particularly useful when few relationship instances exist, in order to avoid NULL values in foreign keys.
- In this case, the primary key of the relationship relation will be only one of the foreign keys that reference the participating entity relations.

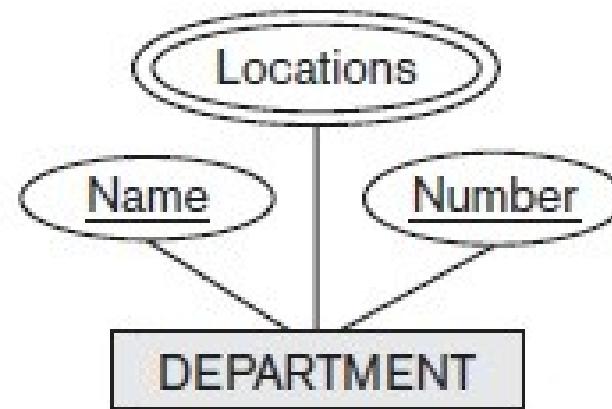
Step 5: Mapping of Binary M:N Relationship Types

- For a 1:N relationship, the primary key of the relationship relation will be the foreign key that references the entity relation on the N-side.
- For a 1:1 relationship, either foreign key can be used as the primary key of the relationship relation.

Step 6: Mapping of Multivalued Attributes

- For each multivalued attribute A, create a new relation R.
- This relation R will include an attribute corresponding to A, plus the primary key attribute K—as a foreign key in R—of the relation that represents the entity type or relationship type that has A as a multivalued attribute.
- The primary key of R is the combination of A and K.
- If the multivalued attribute is composite, we include its simple components.

Step 6: Mapping of Multivalued Attributes



DEPT_LOCATIONS

<u>Dnumber</u>	<u>Dlocation</u>
----------------	------------------

DEPARTMENT

<u>Dname</u>	<u>Dnumber</u>
--------------	----------------

Step 6: Mapping of Multivalued Attributes

- The propagate (CASCADE) option for the referential triggered action should be specified on the foreign key in the relation R corresponding to the multivalued attribute for both ON UPDATE and ON DELETE.
- We should also note that the key of R when mapping a composite, multivalued attribute requires some analysis of the meaning of the component attributes.

Step 6: Mapping of Multivalued Attributes

- In some cases, when a multivalued attribute is composite, only some of the component attributes are required to be part of the key of R
- These attributes are similar to a partial key of a weak entity type that corresponds to the multivalued attribute

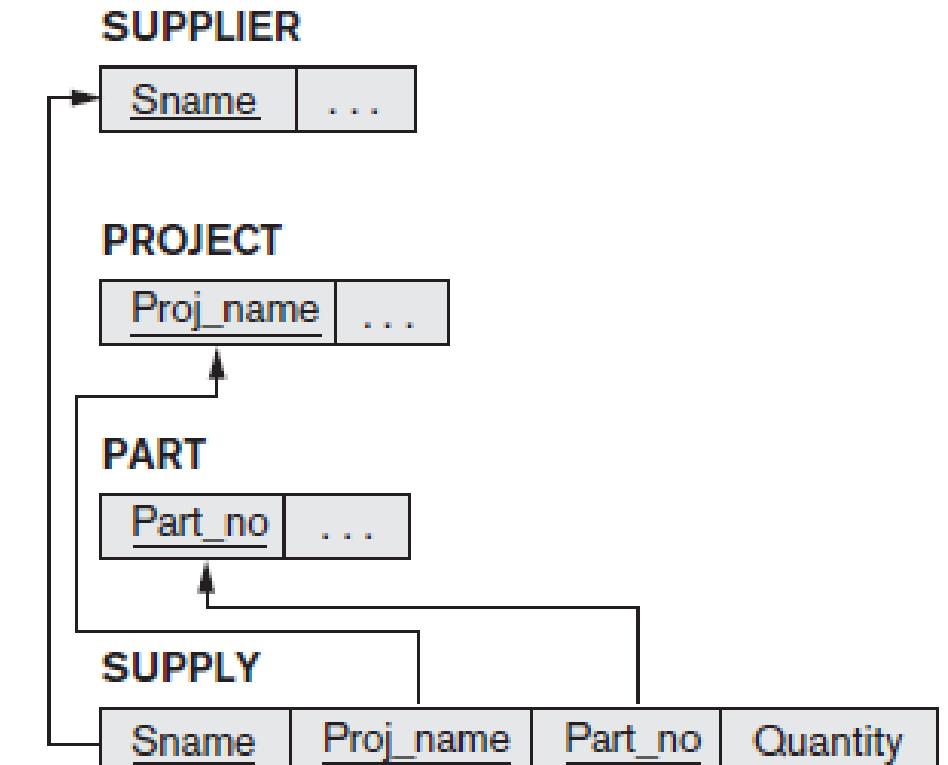
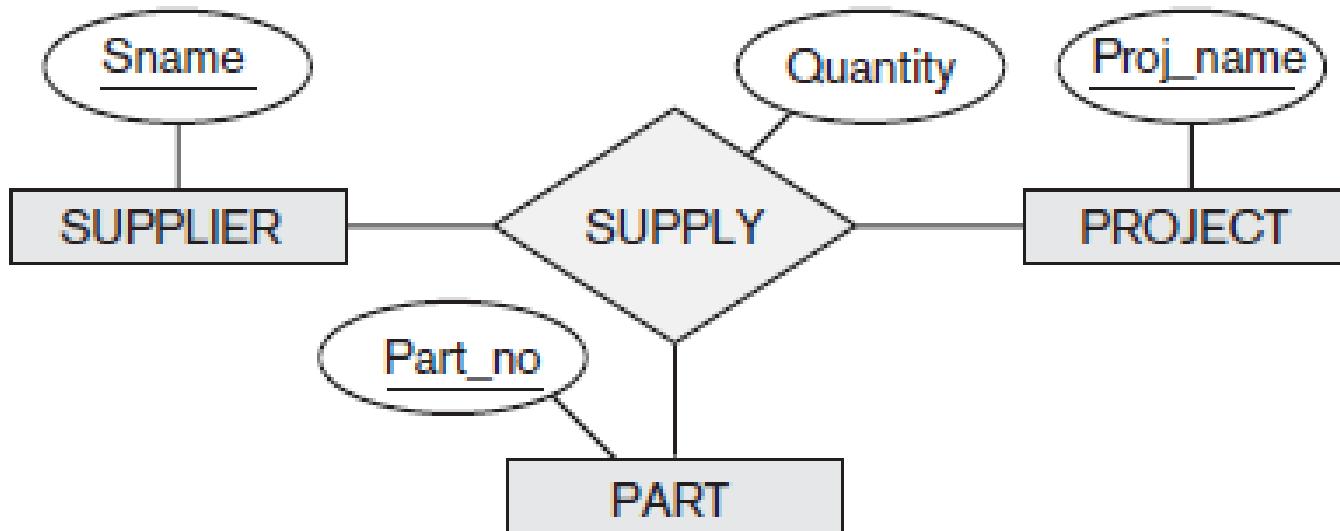
Step 7: Mapping of N-ary Relationship Types

- For each n-ary relationship type R, where $n > 2$, create a new relation S to represent R.
- Include as foreign key attributes in S the primary keys of the relations that represent the participating entity types.
- Also include any simple attributes of the n-ary relationship type as attributes of S.

Step 7: Mapping of N-ary Relationship Types

- The primary key of S is usually a combination of all the foreign keys that reference the relations representing the participating entity types.
- However, if the cardinality constraints on any of the entity types E participating in R is 1, then the primary key of S should not include the foreign key attribute that references the relation E corresponding to E

Step 7: Mapping of N-ary Relationship Types



Correspondence between ER and Relational Models

ER MODEL

Entity type

1:1 or 1:N relationship type

M:N relationship type

n -ary relationship type

Simple attribute

Composite attribute

Multivalued attribute

Value set

Key attribute

RELATIONAL MODEL

Entity relation

Foreign key (or *relationship* relation)

Relationship relation and two foreign keys

Relationship relation and n foreign keys

Attribute

Set of simple component attributes

Relation and foreign key

Domain

Primary (or secondary) key

Step 8: Mapping of Specialization/Generalization

$\text{Attrs}(R)$ to denote *the attributes of relation R*

$\text{PK}(R)$ to denote the *primary key of R*

Convert each specialization with m subclasses $\{S_1, S_2, \dots, S_m\}$ and (generalized) superclass C , where the attributes of C are $\{k, a_1, \dots, a_n\}$ and k is the (primary) key, into relation schemas using one of the following options:

8A: Multiple relations—superclass and subclasses.

8B: Multiple relations—subclass relations only.

8C: Single relation with one type attribute.

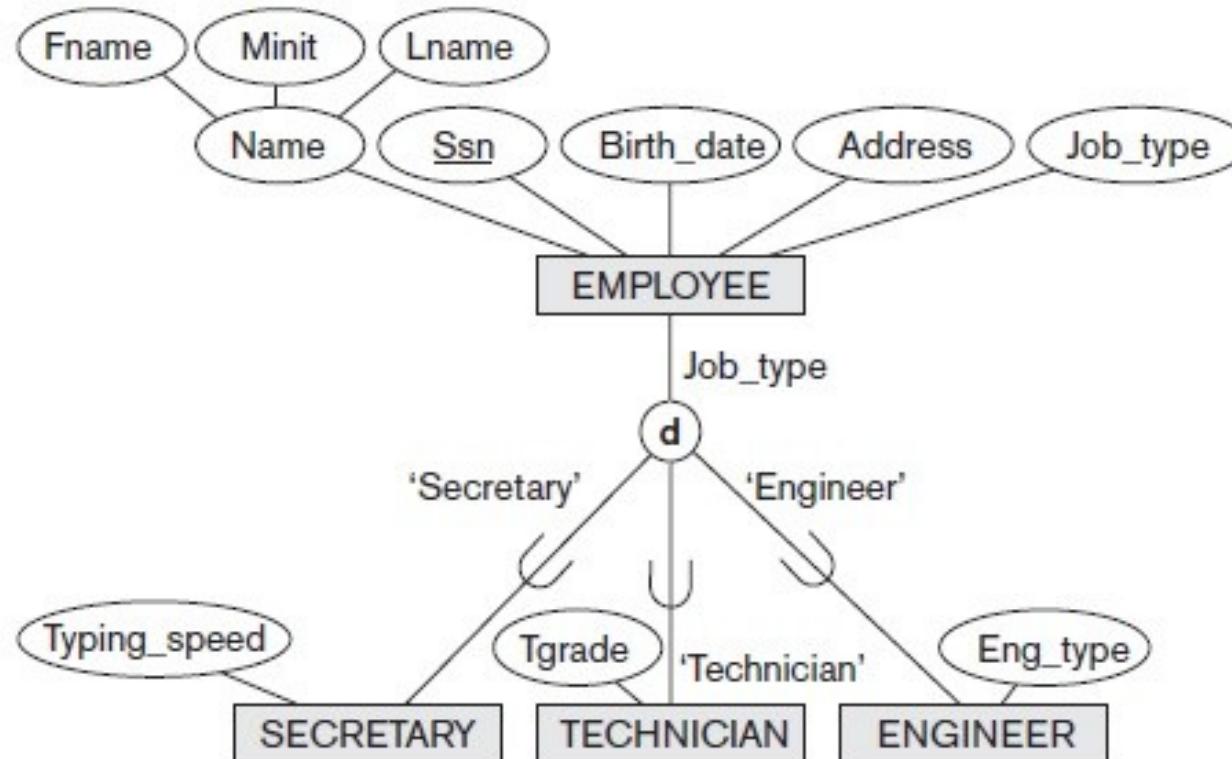
8D: Single relation with multiple type attributes.

Step 8: Mapping of Specialization/Generalization

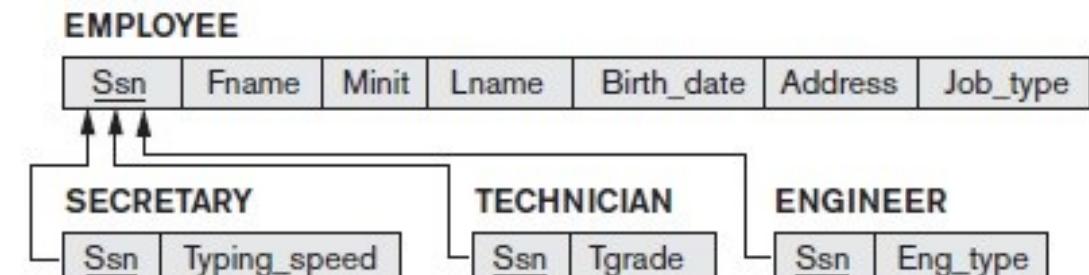
8A: Multiple relations—superclass and subclasses.

- Create a relation L for C with attributes $\text{Attrs}(L) = \{k, a_1, \dots, a_n\}$ and $\text{PK}(L) = k$.
- Create a relation L_i for each subclass S_i , $1 \leq i \leq m$, with the attributes $\text{Attrs}(L_i) = \{k\} \cup \{\text{attributes of } S_i\}$ and $\text{PK}(L_i) = k$.
- This option works for any specialization (total or partial, disjoint or overlapping).

Step 8: Mapping of Specialization/Generalization



ubclasses.



Step 8: Mapping of Specialization/Generalization

8B: Multiple relations— Subclasses relations only.

- Create a relation L_i for each subclass S_i , $1 \leq i \leq m$, with the attributes $Attrs(L_i) = \{ \text{attributes of } S_i \} \cup \{k, a_1, \dots, a_n\}$ and $PK(L_i) = k$.
- This option only works for a specialization whose subclasses are total (every entity in the superclass must belong to (at least) one of the subclasses).

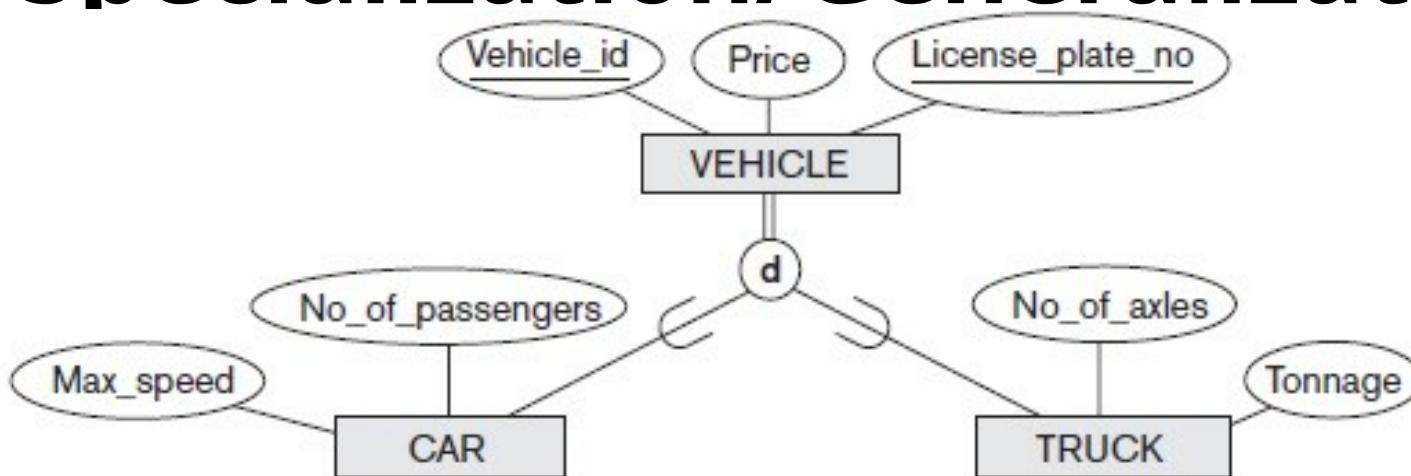
Step 8: Mapping of Specialization/Generalization

8B: Multiple relations— Subclasses relations only.

- Additionally, it is only recommended if the specialization has the disjointedness constraint.
- If the specialization is overlapping, the same entity may be duplicated in several relations.

Step 8: Mapping of Specialization/Generalization

ily.



CAR

Vehicle_id	License_plate_no	Price	Max_speed	No_of_passengers
------------	------------------	-------	-----------	------------------

TRUCK

Vehicle_id	License_plate_no	Price	No_of_axles	Tonnage
------------	------------------	-------	-------------	---------

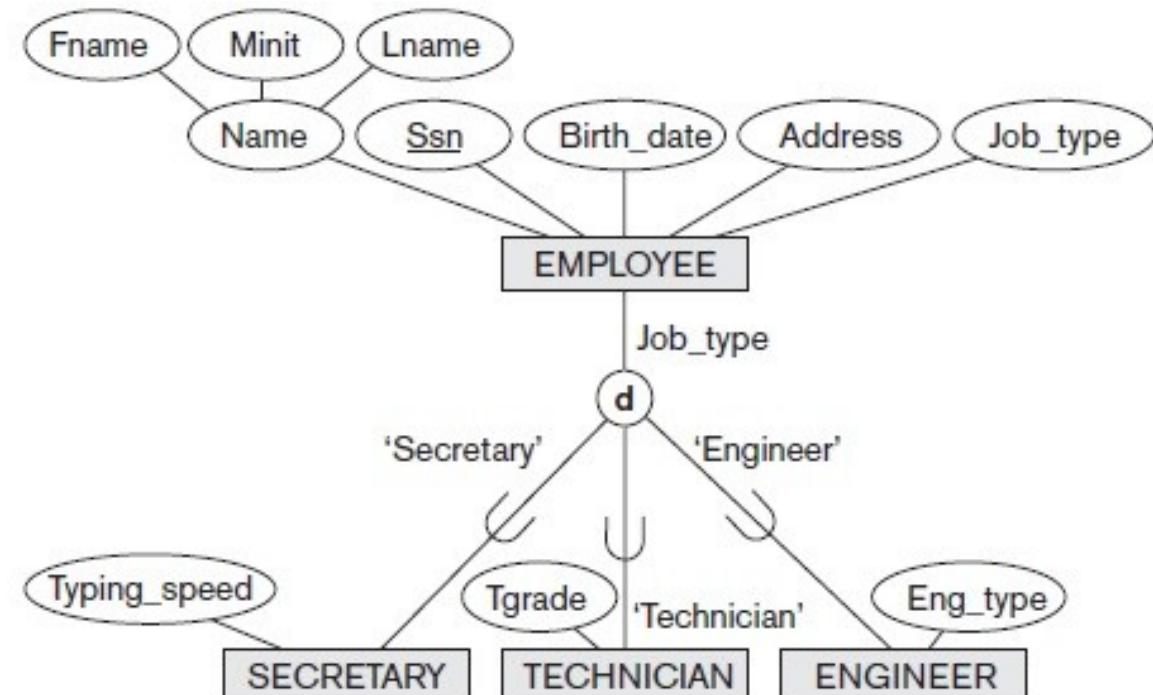
Step 8: Mapping of Specialization/Generalization

8C: Single relation with one type attribute.

- Create a single relation L with attributes $\text{Attrs}(L) = \{k, a_1, \dots, a_n\} \cup \{\text{attributes of } S_1\} \cup \dots \cup \{\text{attributes of } S_m\} \cup \{t\}$ and $\text{PK}(L) = k$.
- The attribute t is called a type (or discriminating) attribute whose value indicates the subclass to which each tuple belongs, if any.
- This option works only for a specialization whose subclasses are disjoint, and has the potential for

Step 8: Mapping of Specialization/Generalization

8C: Single relation with one type



EMPLOYEE

<u>Ssn</u>	Fname	Minit	Lname	Birth_date	Address	Job_type	Typing_speed	Tgrade	Eng_type
------------	-------	-------	-------	------------	---------	----------	--------------	--------	----------

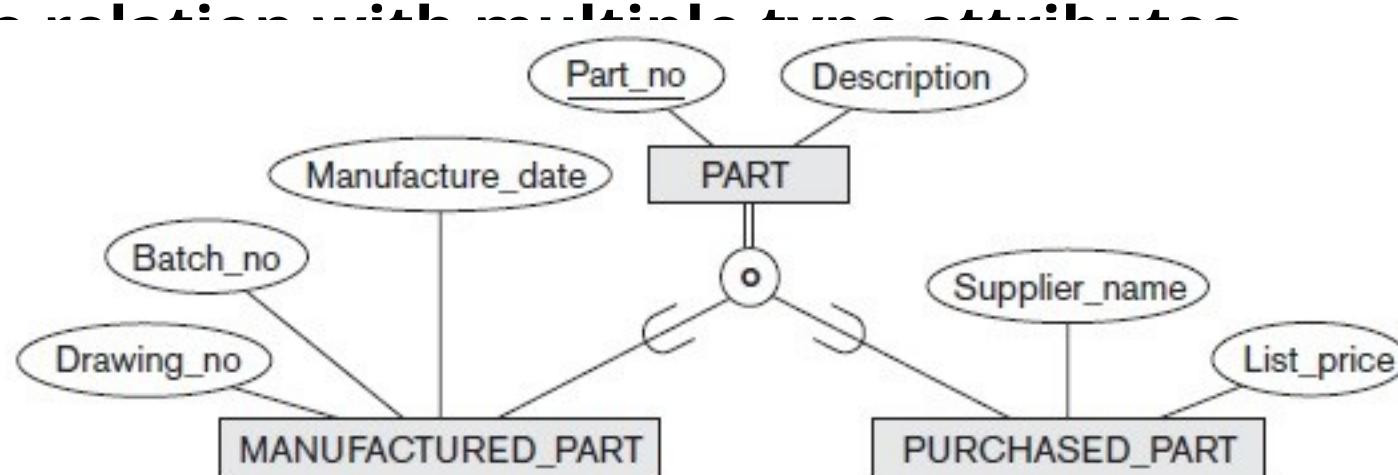
Step 8: Mapping of Specialization/Generalization

8D: Single relation with multiple type attributes.

- Create a single relation schema L with attributes $\text{Attrs}(L) = \{k, a_1, \dots, a_n\} \cup \{\text{attributes of } S_1\} \cup \dots \cup \{\text{attributes of } S_m\} \cup \{t_1, t_2, \dots, t_m\}$ and $\text{PK}(L) = k$.
- Each $t_i, 1 \leq i \leq m$, is a Boolean type attribute indicating whether a tuple belongs to subclass S_i .
- This option is used for a specialization whose subclasses are overlapping (but will also work for a disjoint specialization).

Step 8: Mapping of Specialization/Generalization

8D: Singl



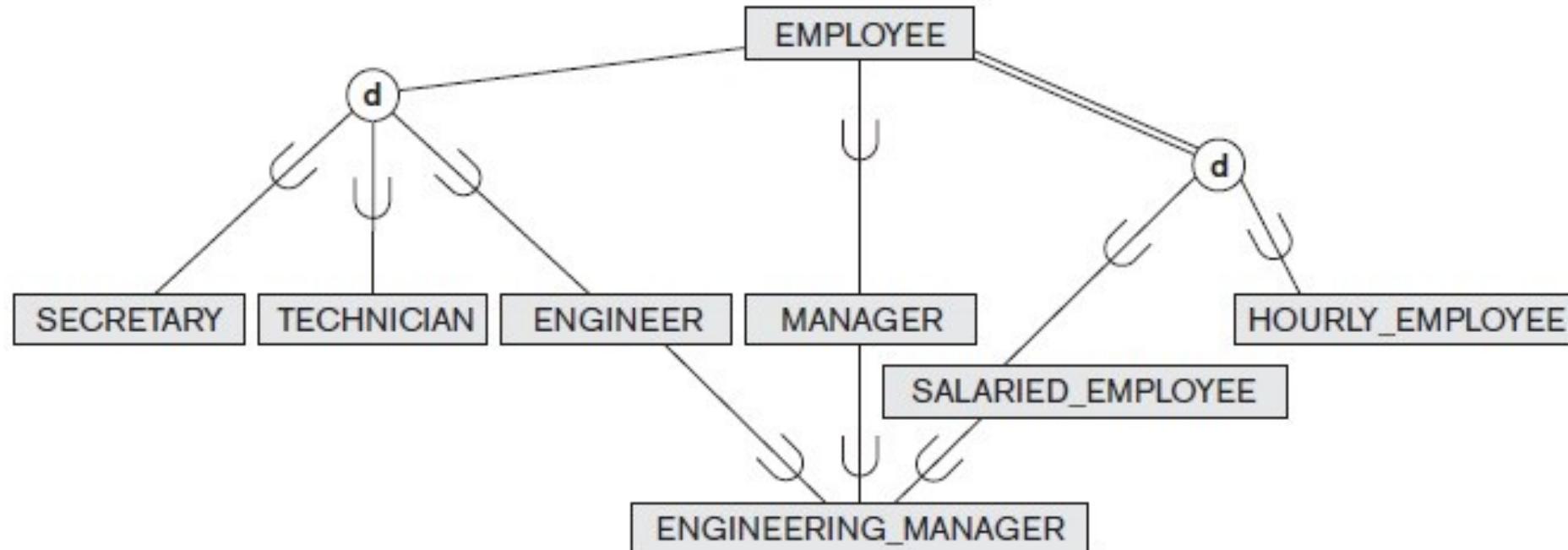
PART

Part_no	Description	Mflag	Drawing_no	Manufacture_date	Batch_no	Pflag	Supplier_name	List_price
---------	-------------	-------	------------	------------------	----------	-------	---------------	------------

Mapping of Shared Subclasses (Multiple Inheritance)

- A shared subclass, such as ENGINEERING_MANAGER in Figure 8.6, is a subclass of several superclasses, indicating multiple inheritance.
- These classes must all have the same key attribute; otherwise, the shared subclass would be modeled as a category (union type)
- We can apply any of the options discussed in step 8 to a shared subclass, subject to the restrictions discussed in step 8 of the mapping algorithm

Mapping of Shared Subclasses (Multiple)

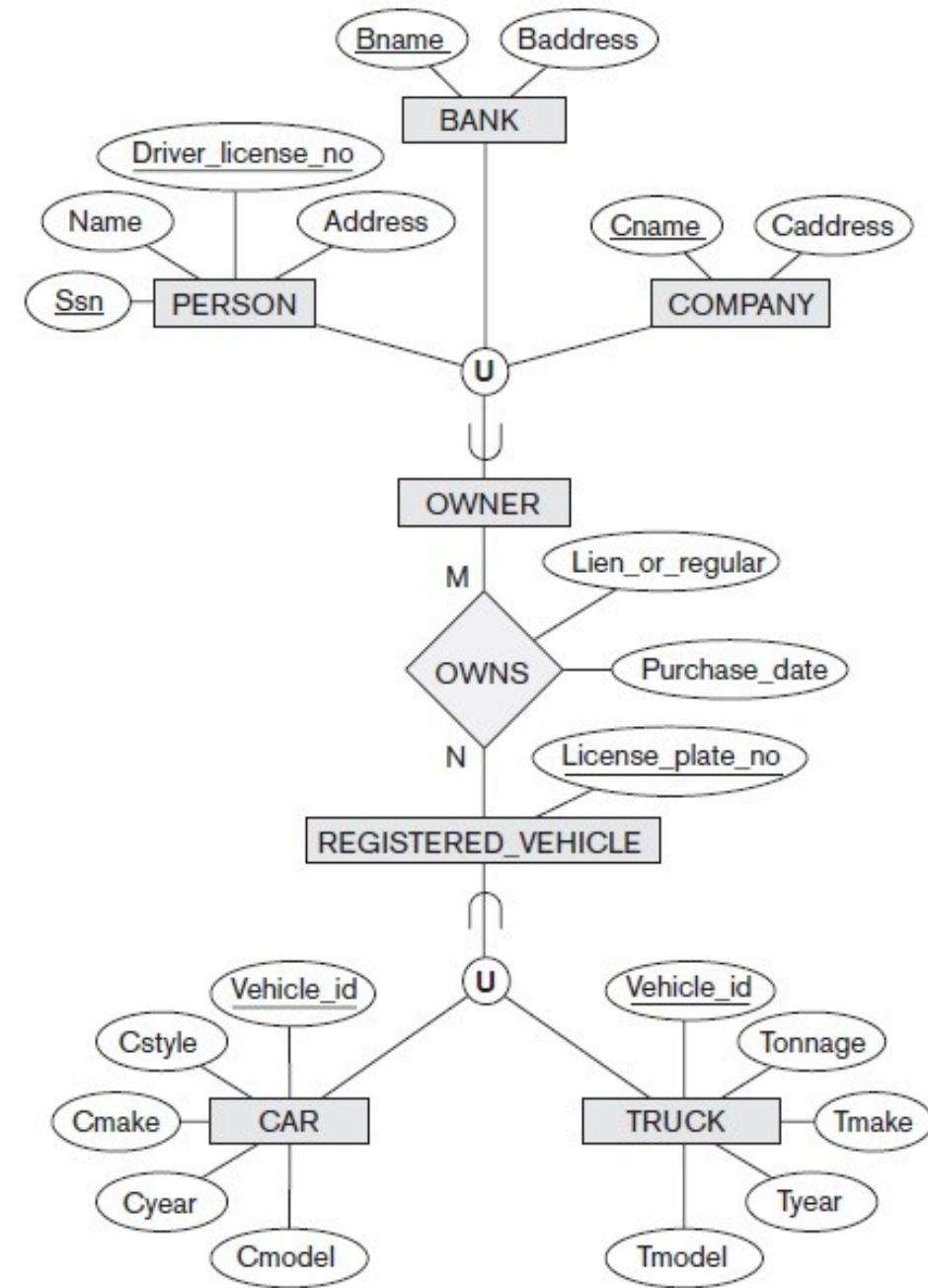
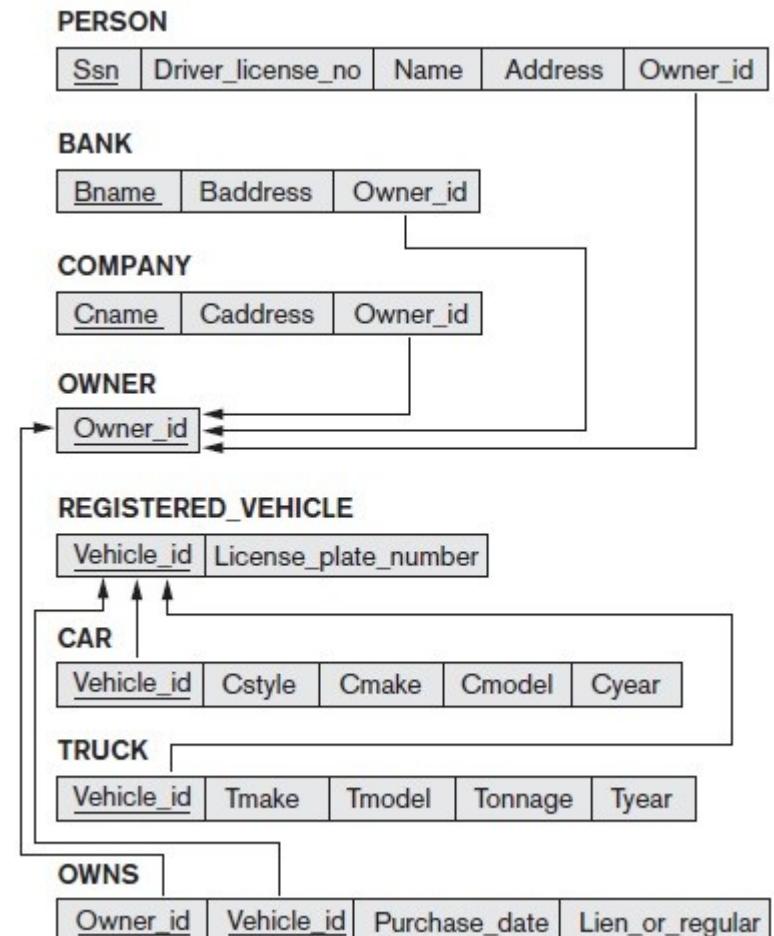


Step 9: Mapping of Categories (Union Types)

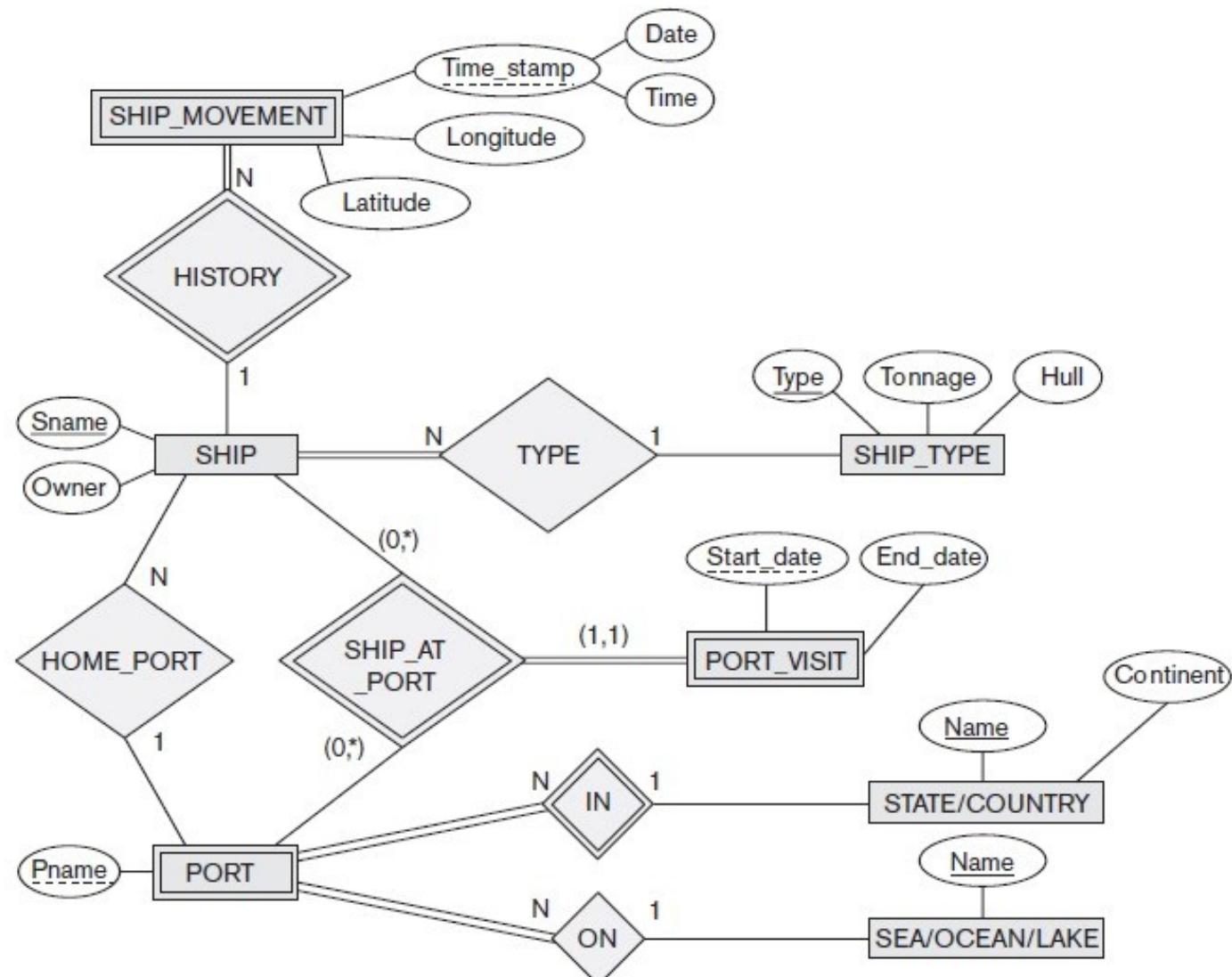
- A category (or union type) is a subclass of the union of two or more superclasses that can have different keys because they can be of different entity types
- For mapping a category whose defining superclasses have different keys, it is customary to specify a new key attribute, called a surrogate key, when creating a relation to correspond to the category.

Mapping EER Model Constructs

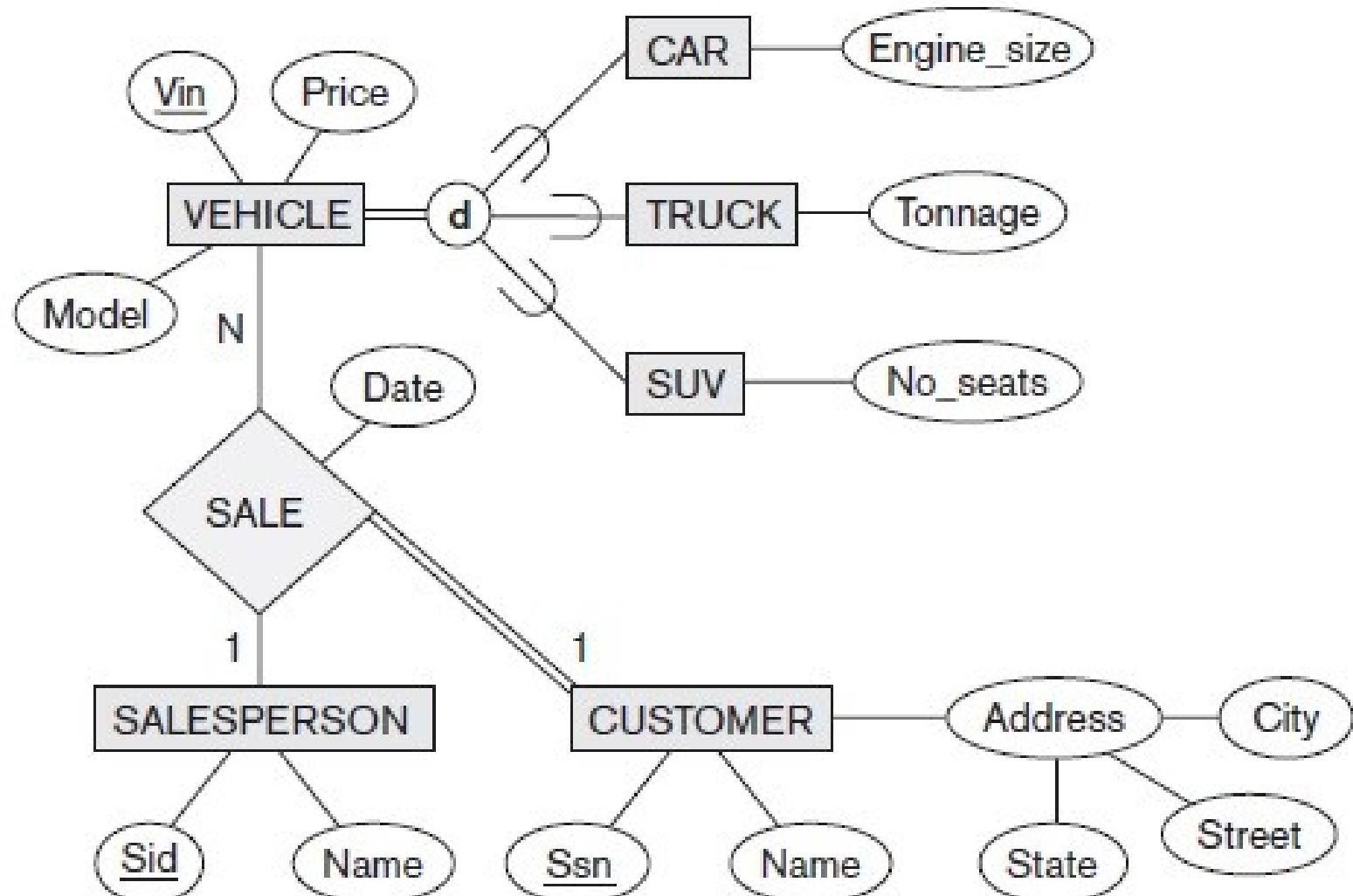
Step 9: Mapping of Categories (Union Type)



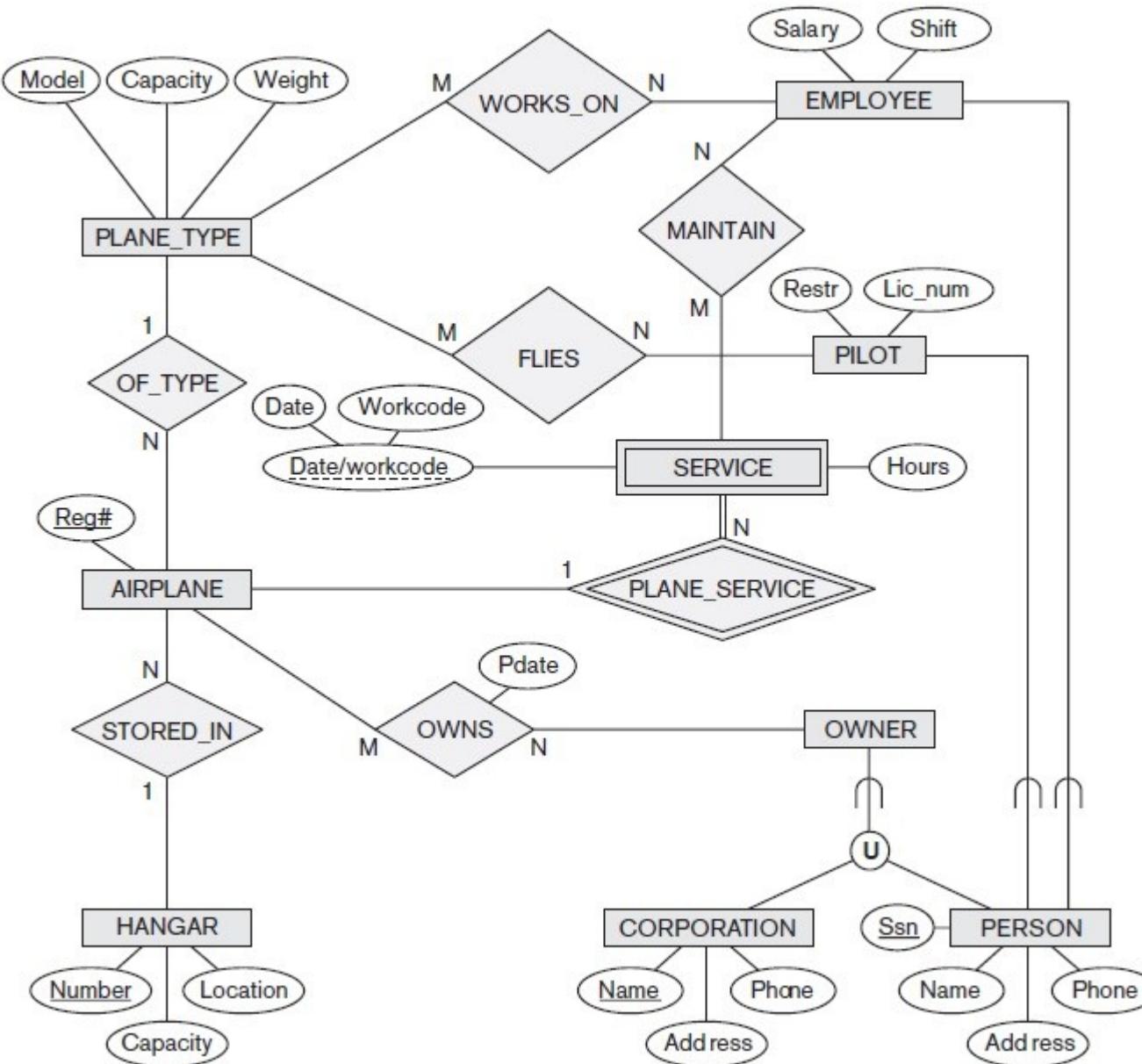
Example: 1. Map ER-Model to Relational Model



Example: 2. Map ER-Model to Relational Model



Example: 3. Map ER-Model to Relational Model





<https://www.youtube.com/watch?v=7LRH7DY1QbQ>



DATABASE DESIGN

BITS Pilani
Pilani Campus



SQL Commands and Relational Algebra

Learning Objectives

- SQL Commands
 - i. Definition
 - ii. Types of SQL commands
 - a. DDL commands
 - b. DML Commands
 - c. DQL commands
 - d. DCL commands and
 - e. TCL commands
- Setting the constraints in Create command
- Definition of Relational Algebra
- Types of Relational algebra
- Basic Operations in Relational Algebra
 - i. Select
 - ii. Project
 - iii. Union
 - iv. Set Difference
 - v. Cartesian product
 - vi. Rename

SQL Commands

SQL commands are instructions coded into SQL statements which communicate with the database to perform specific tasks, work, function and queries with data.

SQL commands are classified into five categories. They are

1. DDL(Data Definition Language) Commands
2. DML (Data manipulation Language) Commands
3. DQL (Data Query Language) Commands
4. DCL (Data Control Language) Commands
5. TCL (Transaction Control Language) Commands

SQL Commands

1. DDL commands

DDL commands are used to make or perform changes on the physical structure of any table residing inside a database. When these commands are executed, the changes in the table are saved immediately. The DDL commands are

- (i) CREATE TABLE
- (ii) ALTER TABLE
- (iii) DROP TABLE
- (iv) RENAME

SQL Commands

(i) **CREATE TABLE**- This command is used to create the structure of the table. The syntax of this command is

```
CREATE TABLE TABLENAME ( Attribute 1 data type, Attribute 2 data type, ..... . Attribute n data type);
```

There are various data types such as integer, number, double, varchar, varchar2, date, etc.,).

Example:

```
CREATE TABLE Student
( StName varchar2(20), StNum number (3),
Dept varchar2(20), Average number (7,2));
```

SQL Commands

The above command creates a table with four attributes as

StName	StNum	Dept	Average

(ii) ALTER TABLE- This command is used to alter the structure of the already created table such as changing size of the attribute, adding a new attribute or deleting an existing attribute. The syntax is

**ALTER TABLE Table name Add/Modify/Drop
(specification of the alteration);**

SQL Commands

Examples:

- a. `ALTER TABLE Student Add (dob date);` // To add a new attribute dob into the Student Table
- b. `ALTER TABLE Student Modify (average number(5, 2));` // Change the size of the attribute average in Student Table
- c. `ALTER TABLE Student drop Dept;` // To delete a attribute Dept from the Student Table

(iii) RENAME – It is used to rename the table or database name. The syntax of the command is

`RENAME Old table name to New table Name;`

SQL Commands

Example:

```
RENAME Student to Students; // The table Student  
is renamed as Students
```

(iv) DROP Command - This command is used to delete a table from the database.

Its syntax is

```
DROP TABLE table name;
```

Example:

```
DROP TABLE Students; // DROP command  
deletes the table names Students from the database  
permanently
```

SQL Commands

2. DML commands

Once the tables are created using DDL Commands, the manipulation inside those tables is done with the DML commands. The advantages of using DML command is, if we did any wrong changes, they can be rolled back easily. The DML Commands are

- (i)INSERT
- (ii)DELETE
- (iii)UPDATE
- (iv)LOCK

SQL Commands

(i) INSERT COMMAND – This DML command is used to insert a tuple / record into the table. The general format of the command is

```
INSERT INTO TABLE NAME (Attribute names )  
VALUES(the values of the attributes );
```

The String and Date values are to be enclosed in Single quotes.

Examples:

a. `INSERT INTO Students VALUES
('Yuvaraj' , 10001 , 'CSE' , 98.78);`

SQL Commands

The INSERT command inserts a new tuple into the table Students.

b. `INSERT INTO Students(StName, StNum, Average)
VALUES ('Bumrah', 10010, 88.75);`

The above INSERT command inserts a new tuple into the table Students in which Dept is filled with NULL. After executing the above commands, the Students table will be

StName	StNum	Dept	Average
Yuvaraj	10001	CSE	95.78
Bumrah	10010	NULL	88.75

SQL Commands

(ii) DELETE COMMAND – This DML command is used to delete a tuple / record from the table. The general format of the command is

```
DELETE * from TABLE NAME [WHERE  
Condition];
```

Here Condition is optional. If there is no condition, all the records / tuples are deleted. Otherwise, the records / tuples satisfying the condition are deleted.

Examples:

a. `DELETE * from Students;`

The DELETE command deletes all the records in the table.

SQL Commands

a. `DELETE from Students WHERE Average<=90;`

The `DELETE` command deletes all the student records who have got average ≤ 90 from the table.

(iii) UPDATE COMMAND – This DML command is used to Modify/change the content of a tuple / record in a table. The general format of the command is

`UPDATE TABLE NAME SET ATT = V WHERE
CONDITION;`

Where

ATT is the attribute which requires modification

V is a value or an expression

SQL Commands

Examples

a. UPDATE Student SET StName = 'Yuvaraj Singh' WHERE StName = 'Yuvaraj';

The UPDATE command modifies the StName, 'Yuvaraj' with 'Yuvaraj Singh'

b. UPDATE Student SET Average = Average + 2.5 WHERE StName = 'Yuvaraj Singh';

The UPDATE command updates the attribute Average with Average + 2.5. That is, 2.5 marks are added with Average for all the records / tuples.

SQL Commands

(iv) LOCK Command – This command is used to lock the privilege as either READ or WRITE. Its syntax is

```
LOCK TABLE Table name READ / WRITE;
```

Example;

```
LOCK TABLE Student READ ; // Locks the privilege as  
READ.
```

3. DQL commands

DQL commands are used for performing data selection in SQL. It is performed with SELECT command. Its syntax is

```
SELECT ATTRIBUTES FROM TABLE NAME WHERE  
CONDITION;
```

SQL Commands

Assume that a table has the tuples as follows:

StName	StNum	Dept	Average
Yuvraj	10001	CSE	98.78
Bumrah	10010	IT	88.75
ROY	10019	Mech	65.25
Foster	55001	CSE	50.55
BIJU	10111	IT	55.75
RAM	44448	CSE	44.00
David	12345	CSE	88.00

SQL Commands

Examples – a. SELECT StNum, Average FROM Students; // displays StNum and Average of all the records in Students table as

StNum	Average
10001	98.78
10010	88.75
10019	65.25
55001	50.55
101116	55.75
44448	44.00
12345	88.00

SQL Commands

b. `SELECT StNum, Average FROM Students where Average>=76;` // displays StNum and Average of all the records in Students table who have the average ≥ 76 as

StNum	Average
10001	98.78
10010	88.75
12345	88.00

c. `SELECT * FROM Students;` // displays all the records in Students table as

SQL Commands

StName	StNum	Dept	Average
Yuvraj	10001	CSE	98.78
Bumrah	10010	IT	88.75
ROY	10019	Mech	65.25
Foster	55001	CSE	50.55
BIJU	10111	IT	55.75
RAM	44448	CSE	44.00
David	12345	CSE	88.00

d. `SELECT * FROM Students where Average>=76;` // displays all the information of all the records in Students table who have the average ≥ 76 as

SQL Commands

StName	StNum	Dept	Average
Yuvraj	10001	CSE	98.78
Bumrah	10010	IT	88.75
David	12345	CSE	88.00

4. DCL commands

These command are used for granting and revoking permission to the users. The commands are GRANT and REVOKE. The syntax of GRANT command is

GRANT Privilege on Object to User name;

Example:

GRANT CREATE TABLE to UserMe;

SQL Commands

Granting the Create command privilege to the user UserMe

The syntax of REVOKE command is

REVOKE Privilege on Object from User name;

Example:

REVOKE CREATE TABLE from UserMe;

Revoking the Create command privilege from UserMe.

The privileges are Alter, Drop, Select, Update, Delete, etc.,

SQL Commands

5. TCL commands

These commands are applied to rollback and commit the changes what we have made in the database.

The TCL Commands are

- (i)ROLLBACK – It is used to Undo the changes made in the database
- (ii)Commit – It is used to save the changes made in the database.

Setting the Constraints

Integrity Constraints

We know that it is possible to set some conditions (constraints) at the time of creating the structure of the table to maintain the integrity.

Integrity checks the validity of the value given for the attribute of the table.

These constraints are called as Integrity Constraints.

When the values of the attributes are given, these constraints check whether they are valid or not.

If they are valid, the values will be permitted to construct the tuple.

Setting the Constraints

After the tuple/record is constructed, it can be added into the table.

If the value of the attribute is not valid(incorrect value) , the constraint generates the error code and is displayed on the user's screen without giving the permission to construct the tuple.

Setting the Constraints

The constraints can be set with various formats. Some of them are

Constraint Type 1. Setting an attribute as NOT NULL

This constraint is set to accept a value for mandatory attribute and it can be assigned a NULL value.

Example for setting this constraint is

CREATE TABLE Student

```
( StName varchar2(20) NOT NULL,  
  StNum number NOT NULL, Dept varchar2(20),  
          Average number (7,2));
```

Setting the Constraints

Constraint Type 2. Setting an attribute as a Primary Key.

The primary key is set for avoiding duplicate value for an attribute.

Example for setting this constraint is

```
CREATE TABLE Student
( StName varchar2(20), StNum number NOT
NULL, Dept varchar2(20), Average number
(7, 2) primary key(StNum) );
```

Setting the Constraints

Constraint Type 3. Setting the possible value (s) for an attribute.

The set of valid values to be assigned for an attribute is given by using a constraint called as CHECK.

Example2 for setting this constraint is

```
1.CREATE TABLE Student
( StName varchar2(20), StNum number NOT
NULL, Dept varchar2(20), Average number
(7,2) CHECK (Dept in
('CSE', 'IT', 'Mech', 'BIO'))); // valid
values for the attribute Dept is set
```

Setting the Constraints

```
2.CREATE TABLE Daily_wages  
(Name varchar2(20), Num number NOT NULL,  
Hrs_worked number (3,2), Wage number (4,2)  
CHECK (Hrs_worked between 0 and 24));
```

The valid value for attribute Hrs_worked is set between 0 and 24.

If the value for the attribute is given out of this range, the system generates the error message.

Relational Algebra

Relational Algebra in DBMS

Relational algebra is a **procedural** query language that performs operation on relational model.

The purpose of a query language is to retrieve data from database or perform various operations such as insert, update, delete on the data in the database.

Conversely ***relational calculus*** is a **non-procedural query language**.

It specifies what data to be retrieved from the database but doesn't specify how to retrieve it.

Relational Algebra

Types of operations in relational algebra

The types of operations in relational algebra are classified into broad two categories. They are

1. **Basic Operations** – these are the fundamental operations which are to be performed on the data.
2. **Derived Operations** – these are the advanced operations which are used for extension operations

1. Basic/Fundamental Operations in relational algebra are:

- i. Select (σ)
- ii. Project (Π)

Relational Algebra

- iii. Union (\cup)
- iv. Set Difference (-)
- v. Cartesian product (\times)
- vi. Rename (ρ)

2. The Derived Operations in relational algebra are:

- i. Natural Join (\bowtie)
- ii. Left, Right, Full outer join (\bowtie_L , \bowtie_R , \bowtie_F)
- iii. Intersection (\cap)
- iv. Division (\div)

BASIC OPERATIONS

i. Select Operator

Select Operator is denoted by sigma (σ) and it is used to find the tuples / rows in a relation / a table which satisfy the

given condition. **Syntax of Select Operator (σ)**

σ Condition/Predicate (Relation/Table name)

Example for Select Operator

Assume that a customer table has the following data
Table: CUSTOMER

Relational Algebra - BASIC OPERATIONS



Customer_Id	Customer_Name	Customer_City
C10100	Sachin	Agra
C10111	Laxman	Agra
C10115	Shewag	Noida
C10117	Virat	Delhi
C10118	Dravid	Delhi

If the Query is

σ Customer_City="Agra" (CUSTOMER) ,

the output will be

Relational Algebra - BASIC OPERATIONS



Customer_Id	Customer_Name	Customer_City
C10100	Sachin	Agra
C10111	Laxman	Agra

Note:- The Select operation in SQL is performed with SELECT command with a where clause.

The above operation is performed with SQL command as

```
Select * from CUSTOMER where  
Customer_city="Agra";
```

Relational Algebra

ii. Project Operator

Project operator is denoted by Π symbol and it is used to select desired columns / attributes from a table / a relation.

Syntax of Project Operator

Π column_name1, column_name2, . . . ,
column_nameN(table_name)

Example for Project Operator

Consider a table CUSTOMER with three columns, we want to fetch only two columns of the table,

Π Customer_Name, Customer_City (CUSTOMER)

The above relational algebra query selects and returns the columns specified from the table

Relational Algebra - BASIC OPERATIONS



Then the Output would be

Customer_Name	Customer_City
Sachin	Agra
Laxman	Agra
Shewag	Noida
Virat	Delhi
Dravid	Delhi

Note:- The corresponding SQL command is:

```
Select Customer_Name ,Customer_City  from  
CUSTOMER;
```

Relational Algebra - BASIC OPERATIONS



iii. Union Operator

Union operator is denoted by \cup symbol and it is used to select all the rows / tuples from two tables / relations.

Let us discuss union operator a bit more.

Assume that there are two relations R1 and R2 both have same columns and we want to select all the tuples /rows from these relations then we can apply the union operator on these relations.

Note: The rows / tuples that are present in both the tables will only appear once in the union set.

In short we can say that there are no duplicates present after the union operation.

Relational Algebra - BASIC OPERATIONS



Syntax of Union Operator: table_name1 \cup table_name2

Example for Union Operator

Assume that the tables COURSE and STUDENT have the data as follows

Table 1: COURSE

Course_Id	Student_Name	Student_Id
C101	Aditya	S901
C104	Aditya	S901
C106	Sachin	S911
C109	Yuvaraj	S921
C115	Mithali	S931

Relational Algebra

Table 2: STUDENT

Student_Id	Student_Name	Student_Age
S901	Aditya	19
S911	Sachin	18
S921	Yuvraj	19
S931	Mithali	17
S941	Dravid	16
S951	Rick	18

If we use the Query is executed with the intersection operator as
 $\Pi \text{ Student_Name } (\text{COURSE}) \cup \Pi \text{ Student_Name } (\text{STUDENT})$,
the Output table will be

Relational Algebra

Student_Name
Aditya
Dravid
Yuvaraj
Mithali
Rick
Sachin

Note: As we can see ~~there are no duplicates~~ names present in the output even though we had few common names in both the tables, also in the COURSE table we had the duplicate name itself.

Relational Algebra - BASIC OPERATIONS



iv. Intersection Operator

Intersection operator is denoted by \cap symbol and it is used to select common rows /tuples from two tables /relations.

Assume that we have two relations R1 and R2 both have same columns and we want to select all those tuples / rows that are present in both the relations, then in that case we can apply intersection operation on these two relations $R1 \cap R2$.

Relational Algebra - BASIC OPERATIONS



Note: Only those rows that are present in both the tables will appear in the result set.

Syntax of Intersection Operator

`table_name1 \cap table_name2`

Example for Intersection Operator

Let us take the same example that we have taken above operator.

If the Query is executed with the intersection operator as
 `Π Student_Name (COURSE) \cap Π Student_Name (STUDENT)`,

the Output table is shown as follows

Relational Algebra - BASIC OPERATIONS



Student Name
Aditya
Sachin
Yuvaraj
Mithali

The Equivalent SQL command is

Select Student_Name from Course **INTERSECT** Select Student_Name from STUDENT;

v. Set Difference - Set Difference is denoted by – symbol.

Let us assume that we have two relations R1 and R2 and we want to select all those tuples / rows that are present in Relation R1 but **not** present in Relation R2, this can be done using Set difference R1 – R2.

Relational Algebra - BASIC OPERATIONS



Example for Set Difference

Let us take the same tables COURSE and STUDENT that we have seen above.

Query:

Let us write a query to select those student names that are present in STUDENT table but not present in COURSE table.

$\Pi \text{ Student_Name } (\text{STUDENT}) - \Pi \text{ Student_Name } (\text{COURSE})$

Relational Algebra - BASIC OPERATIONS



Student Name
Dravid
Rick

The Equivalent SQL command is

```
Select Student_Name from STUDENT MINUS  
Select Student_Name from COURSE;
```

Relational Algebra - BASIC OPERATIONS



vi. Cartesian product

Cartesian Product is denoted by X symbol.

Assume that R1 and R2 are the given relations.

The Cartesian product of these two relations ($R1 \times R2$) would combine each tuple of first relation R1 with each tuple of second relation R2.

Syntax of Cartesian product (X)

$R1 \times R2;$

Relational Algebra - BASIC OPERATIONS



Example for Cartesian product

Table 1: R

Col_A	Col_B	Col_C
AA	100	
BB	200	
CC	300	

Table 2 : S

Col_X	Col_Y
XX	99
YY	11
ZZ	101

If the Query is to find the Cartesian product (RXS) of the tables R and S, the output will be

Relational Algebra - BASIC OPERATIONS



Col_A	Col_B	Col_X	Col_Y
AA	100	XX	99
AA	100	YY	11
AA	100	ZZ	101
BB	200	XX	99
BB	200	YY	11
BB	200	ZZ	101
CC	300	XX	99
CC	300	YY	11
CC	300	ZZ	101

Relational Algebra - BASIC OPERATIONS



Note: The number of rows in the output will always be the cross product of number of rows in each table.

In our example table 1 has 3 rows and table 2 has 3 rows, so the output has $3 \times 3 = 9$ rows.

vi. Rename

The rename operation is represented with the symbol ρ .

This operation is used to rename a relation or an attribute of a relation.

Rename Syntax:

$\rho(\text{new_relation_name}, \text{old_relation_name})$

Relational Algebra - BASIC OPERATIONS

Example for Rename - Let us say we have a table customer, we are fetching customer names and we are renaming the resulted relation to CUST_NAMES.

Table: CUSTOMER

Customer_Id	Customer_Name	Customer_City
C10100	Sachin	Agra
C10111	Laxman	Agra
C10115	Shewag	Noida
C10117	Virat	Delhi
C10118	Dravid	Delhi

Relational Algebra - BASIC OPERATIONS



If the Query is formed as

$\rho(\text{CUST_NAMES}, \Pi(\text{Customer_Name})(\text{CUSTOMER}))$,
the output will be

CUST_NAMES
Sachin
Raghu
Shewag
Ajeet
Dravid

Thanks



DATABASE DESIGN

BITS Pilani
Pilani Campus

SESSION 6



Relational Algebra and SQL Commands

Learning Objectives

- Derived Operations in Relational Algebra
- Different types of Joins
 - i. Inner Join
 - ii. Full Join
 - iii. Left Join
 - iv. Right Join
- Nested Query / Sub query
- SQL Aggregate Functions
- SQL EXISTS / NOT EXISTS Condition
- SQL command with GROUP By Clause
- SQL Command with HAVING clause

JOIN operation in SQL Commands



What are Joins?

JOINS in SQL are commands which are used to combine rows from two or more tables, based on an associated column between those tables. They are largely used when a user tries to extract data from tables which have one-to-many or many-to-many relationships between them.

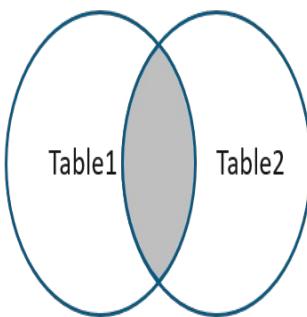
Types of Joins

There are 4 types of Joins. They are

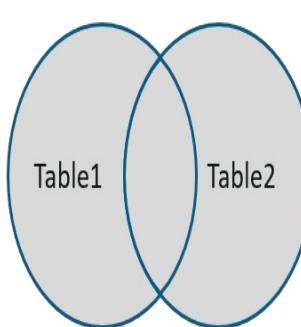
1. Inner Join
2. Full Join
3. Left Join
4. Right Join

JOIN operation in SQL Commands

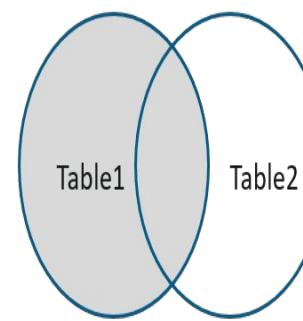
The following represents the various Joins



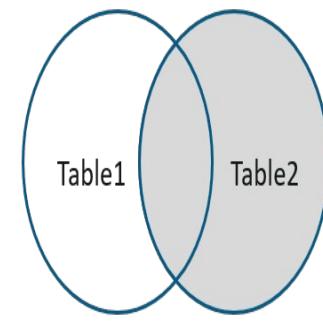
INNER JOIN



FULL JOIN



LEFT JOIN



RIGHT JOIN

To explain various Join operations, we consider two tables such as EMPLOYEE and PROJECT in which EMPLOYEE table consists of records of Employee and PROJECT contains the records of the information about various projects done by the employees

JOIN operation in SQL Commands

EMPLOYEE

EmpId	Name	Age	EmailId
E101	Mr. Dhoni	33	dhoni@abc.com
E102	Mr.Sachin	40	sachin@abc.com
E103	Mr.Rohit Sharma	30	hitman@abc.com
E104	Mr. Virat Kohli	28	virat_k@abc.com

PROJECT

ProjId	EmpId	Proj_Name	Start_date
P1	E102	Project A	07-2-2018
P2	E104	Project B	03-8-2018
P3	E102	Project C	01-1-2019
P4	E120	Project D	10-8-2019

JOIN operation in SQL

Commands – Inner Join

I. INNER JOIN

INNER join returns those records which have matching values in both tables. So, if you perform an INNER join operation between the Employee table and the Projects table,

all the tuples which have matching values in both the tables will be given as output.

Syntax:

```
SELECT Table1.Column1,Table1.Column2,Table2.Column1,....  
FROM Table1INNER JOIN Table2 ON  
Table1.MatchingColumnName =  
Table2.MatchingColumnName;
```

JOIN operation in SQL

Commands – Inner Join

Example: `SELECT EMPLOYEE.NAME,EMPLOYEE.EmpId, PROJECT.ProjId, PROJECT.Proj_Name FROM EMPLOYEE INNER JOIN PROJECT ON EMPLOYEE.EmpId = PROJECT.EmpId;`

The output of the above query is

Name	<u>EmpId</u>	ProjId	Proj_Name
Mr.Sachin	E102	P1	Project A
Mr. Virat Kohli	E104	P2	Project B
Mr.Sachin	E102	P3	Project C

II. FULL JOIN

Full Join or the Full Outer Join returns all those records which either have a match in the left(Table1) or the right(Table2) table. The **Syntax** for **FULL JOIN** is

SQL Commands

```
SELECT Table1.Column1,Table1.Column2,Table2.Column1,....  
FROM Table1 FULL JOIN Table2 ON  
Table1.MatchingColumnName =  
Table2.MatchingColumnName;
```

Example:

```
SELECT EMPLOYEE.NAME,EMPLOYEE.EmpId, PROJECT. ProjId,  
PROJECT. Proj_Name FROM EMPLOYEE  
FULL JOIN PROJECT ON  
EMPLOYEE.EmpId = PROJECT. EmpId;
```

If the above query is executed , we get the following result:

JOIN operation in SQL

Commands – Left Join

Name	EmpId	ProjId	Proj_Name
Mr. Dhoni	E101	NULL	NULL
Mr. Sachin	E102	P1	Project A
Mr. Sachin	E102	P3	Project C
Mr. Rohit Sharma	E103	NULL	NULL
Mr. Virat Kohli	E104	P2	Project B
NULL	NULL	P4	Project D

III. LEFT JOIN

The LEFT JOIN or the LEFT OUTER JOIN returns all the records from the left table and also those records which satisfy a condition from the right table. Also, for the records having no matching values in the right table, the output or the result-set

JOIN operation in SQL Commands – Left Join

will contain the NULL values.

Its Syntax:

```
SELECT Table1.Column1,Table1.Column2,Table2.Column1,....  
      FROM Table1 LEFT JOIN Table2 ON  
            Table1.MatchingColumnName =  
            Table2.MatchingColumnName;
```

Example:

```
SELECT EMPLOYEE.NAME,EMPLOYEE.EmpId, PROJECT. ProjId,  
      PROJECT. Proj_Name FROM EMPLOYEE  
      LEFT JOIN PROJECT ON  
            EMPLOYEE.EmpId = PROJECT. EmpId;
```

If the above query is executed , we get the following result:

JOIN operation in SQL

Commands – Right Join

EmpName	EmpId	ProjId	Proj_Name	
Mr. Dhoni	E101	NULL	NULL	
Mr.Sachin	E102	P1	Project A	
Mr.Sachin	E102	P3	Project C	
Mr.Rohit Sharma	E103	NULL	NULL	
Mr. Virat Kohli	E104	P2	Project B	

III. RIGHT JOIN

The RIGHT JOIN or the RIGHT OUTER JOIN returns all the records from the right table and also those records which satisfy a condition from the left table. Also, for the records having no matching values in the left table, the output or the result-set will contain the NULL values.

JOIN operation in SQL

Commands – Right Join

Its Syntax:

```
SELECT Table1.Column1,Table1.Column2,Table2.Column1,....  
      FROM Table1 RIGHT JOIN Table2 ON  
            Table1.MatchingColumnName =  
            Table2.MatchingColumnName;
```

Example:

```
SELECT EMPLOYEE.NAME,EMPLOYEE.EmpId, PROJECT. ProjId,  
      PROJECT. Proj_Name FROM EMPLOYEE  
      RIGHT JOIN PROJECT ON  
            EMPLOYEE.EmpId = PROJECT. EmpId;
```

If the above query is executed , we get the following result:

JOIN operation in SQL

Commands – Right Join

EmpName	EmpId	ProjId	Proj_Name
Mr.Sachin	E102	P1	Project A
Mr. Virat Kohli	E104	P2	Project B
Mr.Sachin	E102	P3	Project C
NULL	NULL	P4	Project D

SQL NESTED QUERY

SUBQUERY / NESTED QUERY

A subquery consists of two queries as out and inner queries. A subquery is used to return data that will be used in the main query as a condition to further restrict the data to be retrieved. Subqueries can be used with the SELECT, INSERT, UPDATE, and DELETE statements along with the operators like =, <, >, >=, <=, IN, BETWEEN, etc.

There are a few rules that subqueries must satisfy -

- Subqueries must be enclosed within parentheses.
- A subquery can have only one column in the SELECT clause, unless multiple columns are in the main query for the subquery to compare its selected columns.

SQL NESTED QUERY

- An ORDER BY command cannot be used in a subquery, although the main query can use an ORDER BY. The GROUP BY command can be used to perform the same function as the ORDER BY in a subquery.
- Subqueries that return more than one row can only be used with multiple value operators such as the IN operator.
- A subquery cannot be immediately enclosed in a set function.

1. Subqueries with the SELECT Statement

Subqueries are most frequently used with the SELECT statement. The syntax for a subquery is as follows -

SQL NESTED QUERY

```
SELECT column_name [, column_name ] FROM table1 [, table2 ]
  WHERE column_name OPERATOR
  (SELECT column_name [, column_name ]
   FROM table1 [, table2 ] [WHERE condition]);
```

Example

Consider the CUSTOMERS table having the following records –

ID	NAME	AGE	ADDRESS	SALARY
1	Ramesh	35	Ahmedabad	2000.00
2	Keshav	25	Delh	1500.00
3	kaushik	23	Kota	2000.00
4	Chaitanya	25	Mumbai	6500.00
5	Rajesh	27	Bhopal	8500.00
6	Kamal	22	MP	4500.00

SQL NESTED QUERY

Now, let us check the following subquery with a SELECT statement.

```
SQL> SELECT * FROM CUSTOMERS WHERE ID IN (SELECT ID FROM CUSTOMERS WHERE SALARY > 4500);
```

This would produce the following result.

ID	NAME	AGE	ADDRESS	SALARY
4	Chaitanya	25	Mumbai	6500.00
5	Rajesh	27	Bhopal	8500.00

2. Subqueries with the INSERT Statement

Subqueries also can be used with INSERT statements. The INSERT statement uses the data returned from the subquery to insert into another table. The selected data in the subquery

SQL NESTED QUERY

can be modified with any of the character, date or number functions. The basic syntax is as follows.

```
INSERT INTO table_name [ (column1 [, column2] ) ]  
    SELECT [ * | column1 [, column2] ]  
        FROM table1 [, table2 ]  
            [ WHERE VALUE OPERATOR ]
```

Example

Assume a table CUSTOMER_TEMP has similar structure as CUSTOMER table. To copy the complete CUSTOMER table into the CUSTOMER_TEMP table, we can use the following SQL subquery command

SQL NESTED QUERY

```
INSERT INTO CUSTOMER_TEMP  
SELECT * FROM CUSTOMER WHERE ID IN  
    (SELECT ID FROM CUSTOMER);
```

As usual, the inner query is executed first. When it is executed, it creates a table with all IDs from the CUSTOMER table. Then all the records of the CUSTOMER matching the condition are retrieved (in this case, all the records satisfy the condition) and are inserted into the table CUSTOMER_TEMP.

3. Subqueries with the UPDATE Statement

The subquery can be used in conjunction with the UPDATE statement. Either single or multiple columns in a table can be updated when using a subquery with the UPDATE statement.

SQL NESTED QUERY

The basic syntax is as follows.

UPDATE table

```
SET column_name = new_value [ WHERE OPERATOR  
[ VALUE ] (SELECT COLUMN_NAME FROM  
TABLE_NAME) [ WHERE condition ] ]
```

Example

Assume that we have CUSTOMER_TEMP table available which is backup of CUSTOMER table. The following example Increments the SALARY by 10 % in the CUSTOMERS table for all the customers whose AGE is greater than or equal to 27.

SQL NESTED QUERY

UPDATE CUSTOMER

```
SET SALARY = SALARY + SALARY * 0.10  
WHERE AGE IN  
(SELECT AGE FROM CUSTOMER_TEMP  
WHERE AGE >= 27 );
```

The inner query is executed first which retrieves all the ages of the records in the CUSTOMER table satisfying the condition (≥ 27). Then the outer query is performed to update the salary for the records. The resultant CUSTOMER table is

SQL NESTED QUERY

CUSTOMER

ID	NAME	AGE	ADDRESS	SALARY
1	Ramesh	35	Ahmedabad	2200.00
2	Keshav	25	Delhi	1500.00
3	kaushik	23	Kota	2000.00
4	Chaitanya	25	Mumbai	6500.00
5	Rajesh	27	Bhopal	9350.00
6	Kamal	22	MP	4500.00

4. Subqueries with the DELETE Statement

The subquery can be used in conjunction with the DELETE statement like with any other statements mentioned above.

SQL NESTED QUERY

Its syntax is :

```
DELETE FROM TABLE_NAME [ WHERE OPERATOR [ VALUE ]  
    (SELECT COLUMN_NAME  
        FROM TABLE_NAME) [ WHERE Condition ]
```

Example

Assuming, we have a CUSTOMER_TEMP table available which is a backup of the CUSTOMER table. The following example deletes the records from the CUSTOMER table for all the customers whose AGE is greater than or equal to 25.

```
DELETE FROM CUSTOMER  
WHERE AGE IN (SELECT AGE FROM CUSTOMER_TEMP  
WHERE AGE >= 25 );
```

SQL NESTED QUERY

Now the CUSTOMER table is

CUSTOMER

<u>ID</u>	NAME	AGE	ADDRESS	SALARY
3	kaushik	23	Kota	2000.00
6	Kamal	22	MP	4500.00

SQL Aggregate Functions

AGGREGATE FUNCTIONS

The aggregate functions are the predefined functions to calculate average, sum and count the number of occurrences of the records satisfying a criterion. The aggregate functions are SQL COUNT(), AVG() and SUM() Functions

1. COUNT() function – It returns the number of rows that Matches a specified criterion.

Its syntax is

```
SELECT COUNT(column_name) FROM table_name  
      WHERE condition;
```

2. AVG() function – It returns the average value of a numeric column. Its syntax is

SQL Aggregate Functions

```
SELECT AVG(column_name)
      FROM table_name
            WHERE condition;
```

3. **SUM()** function - It returns the total sum of a numeric column. Its syntax is

```
SELECT SUM(column_name)
      FROM table_name
            WHERE condition;
```

Examples for the aggregate functions:

Assume that we are given a CUSTOMER table as follows:

SQL Aggregate Functions

CUSTOMER

<u>ID</u>	NAME	AGE	ADDRESS	SALARY
1	Ramesh	35	Ahmedabad	2200.00
2	Keshav	25	Delhi	1500.00
3	kaushik	23	Kota	2000.00
4	Chaitanya	25	Mumbai	6500.00
5	Rajesh	27	Bhopal	9800.00

Example 1: If we execute the SQL command

SELECT COUNT(ID) FROM CUSTOMER
 WHERE age >=25; , the output is

COUNT(ID)
4

SQL Aggregate Functions

Example 2: If we execute the SQL command as

```
SELECT SUM(salary)  
FROM CUSTOMER
```

WHERE age >=25; , the output is

SUM(salary)
20000

Example 3: If we execute the SQL command as

```
SELECT AVG(salary)  
FROM CUSTOMER
```

WHERE age >=25; , the output is

AVG(salary)
20000

SQL Exists / Not Exists Condition



SQL EXISTS

The EXISTS condition in SQL is used to check whether the result of a correlated nested query is empty (contains no tuples) or not. The result of EXISTS is a boolean value which is

either True or False. This condition can be used in a SELECT, UPDATE, INSERT or DELETE statement.

Its Syntax is:

SELECT column_name(s) FROM table_name WHERE

EXISTS

(SELECT column_name(s) FROM table_name
WHERE condition);

SQL Exists / Not Exists Condition

Example: Consider the following two relations “Customer” and “Order”.

CUSTOMER

CustId	CustName	Location
C001	Mr. Ram	Chennai - 10
C008	Ms. Geetha	Chennai-22
C110	Mr. Anand	Chennai-78

ORDER

OrderId	CustId	Amount
O1001	C008	12345
O1002	C001	50000
O998	C222	43000

SQL Exists / Not Exists Condition



Using EXISTS condition with SELECT statement

To fetch the name of the customers who placed atleast one order.

```
SELECT CustName  FROM Customer WHERE  
EXISTS (SELECT *  FROM Order WHERE  
        Customer.CustId = Order.CustId);
```

The output is

CustName
Ms.Geetha
Mr. Ram

SQL Exists / Not Exists Condition

Using NOT with EXISTS

To fetch name of the customers who has not placed any order.

```
SELECT CustName FROM Customer WHERE  
NOT EXISTS (SELECT * FROM Order WHERE  
Customer.CustId = Order.CustId);
```

CustName
Mr.Anand

SQL Command with Group By option



The GROUP BY clause is used in the SELECT statement . Optionally it is used in conjunction with aggregate functions to produce summary reports from the database. That's what it does, summarizing data from the database. The queries that contain the GROUP BY clause are called grouped queries and only return a single row for every grouped item.

GROUP BY Syntax

Now that we know what the GROUP By clause is, let's look at the syntax for a basic group by query.

SELECT statement... GROUP BY

column_name1[,column_name2,...] [HAVING condition];

SQL Command with Group By option



Where

- (i) "SELECT statement..." is the standard SQL SELECT statement
- (ii) "GROUP BY column_name1" is the clause that performs the grouping based on a particular column which is specified as column_name1.
- (iii) "[,column_name2,...]" is optional; represents other column names when the grouping is done on more than one columns.
- (iv) "[HAVING condition]" is optional; it is used to restrict the rows affected by the GROUP BY clause. It is similar to the WHERE clause.

SQL Command with Group By option

Grouping using a Single Column

Assume that a table **members** has the data as follows:

Name	MemNo	Gender	Age	Place
Ram	M1001	Male	25	Chennai-1
Hema	M1010	Female	23	Chennai-10
Amrutha	M1005	Female	19	Chennai-5
Akshitha	M1111	Female	24	Chennai-45
Anand	M0009	Male	45	Chennai-78

In order to help understand the effect of Group By clause, let us execute a simple query that returns all the gender entries from the members table.

```
SELECT Gender FROM members ;
```

SQL Command with Group By option



It produces the following output:

Gender
Male
Female
Female
Female
Male

Suppose we want to get the unique values for genders. We can use a following query -

```
SELECT Gender FROM members GROUP BY Gender;
```

It produces the following output:

SQL Command with Group By option



Gender
Male
Female

Note that it returns only two results. This is because we only have two gender types **Male** and **Female**. The GROUP BY clause grouped all the "Male" members together and returned only a single row for it. It is same with the "Female" members. If we use the SQL Query as

```
SELECT Gender,Avg(Age) FROM members GROUP BY  
          Gender;
```

it returns the average Age value of each Gender as

SQL Command with Group By option

Gender	Avg(Age)
Male	35
Female	22

Calculation for Male: $(25+45)/2$

Calculation for Female: $(23+19+24)/3$

Grouping using a Multiple Columns

Assume that a table **Orders** has the data as follows:

<u>SalesId</u>	<u>CustId</u>	Amount
S1000	C5001	8000
S1003	C5003	13000
S1000	C5010	28000
S1000	C5004	40000
S1000	C5001	70000
S1003	C5001	8000

SQL Command with Group By option



```
SELECT SalesId,CustId,Sum(Amount),Avg(Amount) FROM  
Orders GROUP BY SalesId, CustId;
```

it displays the output as

SalesId	CustId	Sum(Amount)	Avg(Amount)
S1000	C5001	78000	39000
S1000	C5010	28000	28000
S1000	C5004	40000	40000
S1003	C5003	13000	13000
S1003	C5001	8000	8000

In the above query, the records are grouped with two columns

/ attributes such as SalesId and CustId.

SQL Command with Group By option



If the same table is grouped with a single attribute as

```
SELECT SalesId,Sum(Amount),Average(Amount) FROM Orders  
GROUP BY SalesId;
```

it displays the output as

<u>SalesId</u>	Sum(Amount)	Average(Amount)
S1000	146000	36500
S1003	21000	10500

SQL Command with Having option



We know that WHERE clause is used to select and reject the individual rows from a table, the HAVING is also used to select

and reject row groups. The format of the HAVING clause parallels that of the WHERE clause, consisting of the keyword HAVING followed by the search condition. Thus, the HAVING clause specifies a search condition for groups.

If a query is asked as , “What is the average salesperson whose

order total more than Rs 30,000?”, we can use the HAVING clause as

```
SELECT SalesId,Sum(Amount),Average(Amount) FROM Orders  
GROUP BY SalesId HAVING Sum(Amount)>30000;
```

SQL Command with Having option



it displays the output as

<u>SalesId</u>	Sum(Amount)	Average(Amount)
S1000	146000	36500

Select Statement with Order By Clause

The Order By clause in the Select statement is used to sort the query result in an order of a column. The order may be in ascending order or descending order. The default order is ascending.

Assume that a table Student has the following data

StName	Stno	Dept	Percent
Dravid	S110	Mech	78
Meera	S200	Cse	92
Levin	S078	Mech	80
Leela	S012	Mech	62
Raman	S003	Mech	91
Pavithra	S235	Cse	77
Pavithra	S234	Cse	67

Select Statement with Order By Clause

If we want to sort the records of the Student table according to the department, we use the Order by clause as

Select * from Student order by Dept;

The output of the above query is				
StName	<u>Stno</u>	Dept	Percent	
Meera	S200	Cse	92	
Pavithra	S235	Cse	77	
Pavithra	S234	Cse	67	
Levin	S078	Mech	80	
Leela	S012	Mech	62	
Raman	S003	Mech	91	
Dravid	S110	Mech	78	

Select Statement with Order By Clause

If we want to sort the records of the Student table according to the department and Stno within the department, we use the Order by clause as **Select * from Student order by**

Dept,Stno;	The output of the above query is			
StName	<u>Stno</u>	Dept	Percent	
Meera	S200	Cse	92	
Pavithra	S234	Cse	67	
Pavithra	S235	Cse	77	
Raman	S003	Mech	91	
Leela	S012	Mech	62	
Levin	S078	Mech	80	
Dravid	S110	Mech	78	

Select Statement with Order By Clause

If we want to sort the Student table according to the descending values of percentage of the students, we use the SQL statement as **Select * from Student order by Percent desc;** The output of the above query is

StName	Stno	Dept	Percent
Meera	S200	Cse	92
Raman	S003	Mech	91
Levin	S078	Mech	80
Dravid	S110	Mech	78
Pavithra	S235	Cse	77
Pavithra	S234	Cse	67
Leela	S012	Mech	62

Thanks



BITS Pilani

Pilani | Dubai | Goa | Hyderabad

Database Design

SESSION 7





Contact Session 7:

Session Outline

- Exercises in Nested Query
- Features of Correlated Sub queries
- Trigger
- Views
- Substring handling and other features of SQL

- Some queries require that existing values in the database be fetched and then used in a comparison condition.
- Such queries can be conveniently formulated by using **nested queries**, which are complete select-from-where blocks within another SQL query.
- That other query is called the **outer query**.
- These nested queries can also appear in the **WHERE clause** or the **FROM clause** or the **SELECT clause** or other **SQL clauses** as needed.

Nested Queries - Example



A Nested query for make a list of all project numbers for projects that involve employee Smith either as worker or as a manager of the department that controls the project:

```
Q4A:  SELECT      DISTINCT Pnumber
        FROM        PROJECT
        WHERE       Pnumber IN
                    ( SELECT      Pnumber
                      FROM        PROJECT, DEPARTMENT, EMPLOYEE
                      WHERE       Dnum=Dnumber AND
                                  Mgr_ssn=Ssn AND Lname='Smith' )
                    OR
                    Pnumber IN
                    ( SELECT      Pno
                      FROM        WORKS_ON, EMPLOYEE
                      WHERE       Essn=Ssn AND Lname='Smith' );
```

Nested Queries - Example



- If a nested query returns a single attribute and a single tuple, the query result will be a single (scalar) value. In such cases, it is permissible to use = instead of IN for the comparison operator.
- To illustrate this, consider the following query:

```
SELECT DISTINCT Essn
  FROM WORKS_ON
 WHERE (Pno, Hours) IN ( SELECT Pno, Hours
                           FROM WORKS_ON
                          WHERE Essn = '123456789' );
```

This query will select the Essns of all employees who work the same (project, hours) combination on some project that employee 'John Smith' (whose Ssn = '123456789') works on.

Nested Queries - Example



Use other comparison operators to compare a single value v

- $= ANY$ (or $= SOME$) operator [equivalent to IN]
 - Returns TRUE if the value v is equal to some value in the set
- Other operators that can be combined with ANY (or $SOME$): $>$, $>=$, $<$, $<=$, and $<>$
- ALL : value must exceed all values from nested query

```
SELECT      Lname, Fname
FROM        EMPLOYEE
WHERE       Salary > ALL  ( SELECT      Salary
                           FROM        EMPLOYEE
                           WHERE       Dno=5 );
```

Above query returns the names of employees whose salary is greater than the salary of all the employees in department 5:

Nested Queries - Example



A reference to an unqualified attribute refers to the relation declared in the innermost nested query.

Example: Retrieve the name of each employee who has a dependent with the same first name and is the same sex as the employee.

```
SELECT      E.Fname, E.Lname
FROM        EMPLOYEE AS E
WHERE       E.Ssn IN      ( SELECT      D.Essn
                           FROM        DEPENDENT AS D
                           WHERE       E.Fname = D.Dependent_name
                           AND E.Sex = D.Sex );
```

In above query, we must qualify E.Sex because it refers to the Sex attribute of EMPLOYEE from the outer query, and DEPENDENT also has an attribute called Sex. If there were any unqualified references to Sex in the nested query, they would refer to the Sex attribute of DEPENDENT.

1. Correlated Nested Queries



- Whenever a condition in the WHERE clause of a nested query references some attribute of a relation declared in the outer query, the two queries are said to be correlated
- Queries that are nested using the = or IN comparison operator can be collapsed into one single block: Example:

■ Example 1.3A:

■ **Query:**

SELECT	E.Fname, E.Lname
FROM	EMPLOYEE AS E, DEPENDENT AS D
WHERE	E.Ssn=D.Essn AND E.Sex=D.Sex AND E.Fname=D.Dependent_name;

- **Correlated nested query**
 - **Evaluated once for each tuple in the outer query**

2. Introduction to Triggers in SQL



- CREATE TRIGGER statement
 - Used to monitor the database
- Typical trigger has three components which make it a rule for an “active database”.
 - **Event(s)**
 - **Condition**
 - **Action**

- AN EXAMPLE with standard Syntax.
- Suppose we want to check whenever an employee's salary is greater than the salary of his or her direct supervisor in the COMPANY database. Several events can trigger this rule: inserting a new employee record, changing an employee's salary, or changing an employee's supervisor. Suppose that the action to take would be to call an external stored procedure **SALARY_VIOLATION**, which will notify the supervisor

Introduction to Triggers: USE OF TRIGGERS



```
CREATE TRIGGER SALARY_VIOLATION
BEFORE INSERT OR UPDATE OF SALARY, SUPERVISOR_SSN
    ON EMPLOYEE
FOR EACH ROW
    WHEN ( NEW.SALARY > ( SELECT SALARY FROM EMPLOYEE
                            WHERE SSN = NEW.SUPERVISOR_SSN ) )
        INFORM_SUPERVISOR(NEW.Supervisor_ssN,
                           NEW.Ssn );
```

3. Views (Virtual Tables) in SQL



- Concept of a view in SQL
- Specification of Views in SQL
- View Implementation, View Update, and Inline Views
- Views as Authorization Mechanisms

- Concept of a view in SQL:

- Single table derived from other tables called the **defining tables**
- Considered to be a virtual table that is not necessarily populated

■ CREATE VIEW command

- Give table name, list of attribute names, and a query to specify the contents of the view
- In V1, attributes retain the names from base tables and In V2, attributes are assigned names

V1:	CREATE VIEW	WORKS_ON1
	AS SELECT	Fname, Lname, Pname, Hours
	FROM	EMPLOYEE, PROJECT, WORKS_ON
	WHERE	Ssn=Essn AND Pno=Pnumber;
V2:	CREATE VIEW	DEPT_INFO(Dept_name, No_of_emps, Total_sal)
	AS SELECT	Dname, COUNT (*), SUM (Salary)
	FROM	DEPARTMENT, EMPLOYEE
	WHERE	Dnumber=Dno
	GROUP BY	Dname;

- Once a View is defined, SQL queries can use the View relation in the FROM clause
- View is always up-to-date
 - Responsibility of the DBMS and not the user
- DROP VIEW** command
 - Dispose of a view

- Complex problem of efficiently implementing a view for querying.
- **Strategy1: Query modification** approach
 - Compute the view as and when needed. Do not store permanently
 - Modify view query into a query on underlying base tables
 - Disadvantage: inefficient for views defined via complex queries that are time-consuming to execute

Strategy 2: View materialization

- Physically create a temporary view table when the view is first queried.
- Keep that table on the assumption that other queries on the view will follow.
- Requires efficient strategy for automatically updating the view table when the base tables are updated.

Incremental update strategy for materialized views

- DBMS determines what new tuples must be inserted, deleted, or modified in a materialized view table.

- Multiple ways to handle materialization:
 - I. **Immediate update strategy** updates a view as soon as the base tables are changed
 - II. **Lazy update strategy** updates the view when needed by a view query
 - III. **Periodic update strategy** updates the view periodically (in the latter strategy, a view query may get a result that is not up-to-date). This is commonly used in Banks, Retail store operations, etc.

Update on a view defined on a single table without any aggregate functions

- Can be mapped to an update on underlying base table- possible if the primary key is preserved in the view

Update not permitted on aggregate views. E.g.,

```
UPDATE DEPT_INFO
SET Total_sal=100000
WHERE Dname='Research';
```

Cannot be processed because Total_sal is a computed value in the view definition

Views as authorization mechanism

SQL query authorization statements Views can be used to hide certain attributes or tuples from unauthorized users.

Suppose a certain user is only allowed to see employee information for employees who

work for department 5;

Then we can create the following view DEPT5EMP and grant the user the privilege to query the view but not the base table EMPLOYEE itself.

This user will only be able to retrieve employee information for employee tuples whose Dno = 5, and will not be able to see other employee tuples when the view is queried.

Views as authorization mechanism

```
CREATE VIEW DEPT5EMP AS  
SELECT *  
FROM EMPLOYEE  
WHERE Dno = 5;
```

In a similar manner, a view can restrict a user to only see certain columns; for

example, only the first name, last name, and address of an employee may be visible

as follows:

```
CREATE VIEW BASIC_EMP_DATA AS  
SELECT Fname, Lname, Address  
FROM EMPLOYEE;
```

Summary of SQL Syntax



Table 7.2 Summary of SQL Syntax

```
CREATE TABLE <table name> ( <column name><column type> [ <attribute constraint> ]
    { , <column name><column type> [ <attribute constraint> ] }
    [ <table constraint> { , <table constraint> } ] )
```

```
DROP TABLE <table name>
```

```
ALTER TABLE <table name> ADD <column name><column type>
```

```
SELECT [ DISTINCT ] <attribute list>
```

```
FROM ( <table name> { <alias> } | <joined table> ) { , ( <table name> { <alias> } | <joined table> ) }
[ WHERE <condition> ]
```

```
[ GROUP BY <grouping attributes> [ HAVING <group selection condition> ] ]
```

```
[ ORDER BY <column name> [ <order> ] { , <column name> [ <order> ] } ]
```

```
<attribute list> ::= ( * | ( <column name> | <function> ( ( [ DISTINCT ] <column name> | * ) ) )
    { , ( <column name> | <function> ( ( [ DISTINCT ] <column name> | * ) ) ) } )
```

```
<grouping attributes> ::= <column name> { , <column name> }
```

```
<order> ::= ( ASC | DESC )
```

```
INSERT INTO <table name> [ ( <column name> { , <column name> } ) ]
```

```
( VALUES ( <constant value> , { <constant value> } ) { , ( <constant value> { , <constant value> } ) }
| <select statement> )
```

Summary of SQL Syntax



Table 7.2 Summary of SQL Syntax

DELETE FROM <table name>

[WHERE <selection condition>]

UPDATE <table name>

SET <column name> = <value expression> { , <column name> = <value expression> }

[WHERE <selection condition>]

CREATE [UNIQUE] INDEX <index name>

ON <table name> (<column name> [<order>] { , <column name> [<order>] })

[CLUSTER]

DROP INDEX <index name>

CREATE VIEW <view name> [(<column name> { , <column name> })]

AS <select statement>

DROP VIEW <view name>

NOTE: The commands for creating and dropping indexes are not part of standard SQL.

- ❑ Complex SQL:
 - Nested queries, joined tables (in the FROM clause), outer joins, aggregate functions, grouping
- ❑ Handling semantic constraints with **CREATE ASSERTION** and **CREATE TRIGGER**
- ❑ **CREATE VIEW** statement and materialization strategies
- ❑ Schema Modification for the DBAs using **ALTER TABLE** , **ADD** and **DROP COLUMN**, **ALTER CONSTRAINT** etc.

SUBSTRING() Function

- The SUBSTRING() extracts a substring with a specified length starting from a location in an input string.
- The syntax of the SUBSTRING() function:
`SUBSTRING(input_string, start, length);`
- In this syntax:
 - `input_string` can be a character, binary, text, ntext, or image expression.
 - `start` is an integer that specifies the location where the returned substring starts. **Note that the first character in the `input_string` is 1, not zero.**
 - `length` is a positive integer that specifies the number of characters of the substring to be returned. The SUBSTRING() function raises an error if the length is negative. **If `start + length > the length of input_string, the substring will begin at the start and include the remaining characters of the input_string.`**

SUBSTRING() examples-1

A) Using SUBSTRING() function with literal strings

- This example extracts a substring with the length of 6, starting from the fifth character, in the 'SQL Server SUBSTRING' string.

```
SELECT SUBSTRING('SQL Server SUBSTRING', 5, 6) result;
```

- output:

result

Server

(1 row affected)

SUBSTRING() examples-2

B) Using SUBSTRING() function with table columns

- The example returns the names attribute in the first column, the first letter of the name attribute in the second column, and the third and fourth characters in the final column.

```
SELECT name, SUBSTRING(name, 1, 1) AS Initial ,
SUBSTRING(name, 3, 2) AS ThirdAndFourthCharacters
FROM sys.databases
WHERE database_id < 5;
```

- Output:

name	Initial	ThirdAndFourthCharacters
master	M	St
tempdb	T	Mp
model	M	De
msdb	M	Db

REPLACE() function

- Replace in SQL is a built-in function that allows you to replace all the incidents of a substring within a specified string with a new substring.
- To replace all occurrences of a substring within a string with a new substring.
- **Syntax**
`REPLACE(input_string, substring, new_substring);`
- In this syntax:
 - `input_string` is any string expression to be searched.
 - `substring` is the substring to be replaced.
 - `new_substring` is the replacement string.
- The REPLACE() function returns a new string in which all occurrences of the substring are replaced by the new_substring.
- It returns NULL if any argument is NULL.

Distinct Keyword

- The SQL DISTINCT keyword is used in conjunction with the SELECT statement to eliminate all the duplicate records and fetching only unique records.
- There may be a situation when you have multiple duplicate records in a table, it makes more sense to fetch only those unique records instead of fetching duplicate records.
- **Syntax**

```
SELECT DISTINCT column1, column2,....columnN  
FROM table_name  
WHERE [condition]
```

- The LIKE operator is used in a WHERE clause to search for a specified pattern in a column.
- There are two wildcards often used in conjunction with the LIKE operator:
 - The percent sign (%) represents zero, one, or multiple characters
 - The underscore sign (_) represents one, single character
- **You can also combine any number of conditions using AND or OR operators.**
- Syntax-

```
SELECT column1, column2, ...
  FROM table_name
 WHERE columnN LIKE pattern;
```

Examples :LIKE operators with '%' and '_' wildcards:

LIKE Operator	Description
WHERE CustomerName LIKE 'a%'	Finds any values that start with "a"
WHERE CustomerName LIKE '%a'	Finds any values that end with "a"
WHERE CustomerName LIKE '%or%'	Finds any values that have "or" in any position
WHERE CustomerName LIKE '_r%'	Finds any values that have "r" in the second position
WHERE CustomerName LIKE 'a_%"	Finds any values that start with "a" and are at least 2 characters in length
WHERE CustomerName LIKE 'a__%"	Finds any values that start with "a" and are at least 3 characters in length
WHERE ContactName LIKE 'a%o'	Finds any values that start with "a" and ends with "o"

Practice Exercise #1.1: (create)



Q1: Create the table suppliers that contains four columns: supplier_id, supplier_name, city, state and supplier_id is primary key

Solution:

```
CREATE TABLE suppliers
( supplier_id int NOT NULL,
  supplier_name char(50) NOT NULL,
  city char(50),
  state char(25),
  CONSTRAINT suppliers_pk PRIMARY KEY (supplier_id)
);
```

Practice Exercise #1.2 (insert)

Q2: Insert some records in the table suppliers that contains four columns: supplier_id, supplier_name, city, state and supplier_id is primary key

Solution:

```
INSERT INTO suppliers (supplier_id, supplier_name, city, state)
VALUES (100, 'Microsoft', 'Redmond', 'Washington'),
       (200, 'Google', 'Mountain View', 'California'),
       (300, 'Oracle', 'Redwood City', 'California'),
       (400, 'Kimberly-Clark', 'Irving', 'Texas'),
       (500, 'Tyson Foods', 'Springdale', 'Arkansas'),
       (600, 'SC Johnson', 'Racine', 'Wisconsin'),
       (700, 'Dole Food Company', 'Westlake Village', 'California'),
       (800, 'Flowers Foods', 'Thomasville', 'Georgia'),
       (900, 'Electronic Arts', 'Redwood City', 'California');
```

Practice Exercise #1.3: (select)

Q3: Based on the supplier table, select the unique city values that reside in the state of California and order the results in descending order by city:

- Solution:

```
SELECT DISTINCT city  
FROM suppliers  
WHERE state = 'California'  
ORDER BY city DESC;
```

- These are the results that you should see:

city
Westlake Village
Redwood City
Mountain View

Practice Exercise #1.4: (update)

Q4: Based on the suppliers table populated with the following data, update the city to 'Boise' and the state to "Idaho" for all records whose supplier_name is "Microsoft".

Solution: `UPDATE suppliers SET city = 'Boise',`

~~`state = 'Idaho' WHERE supplier_name = 'Microsoft'.`~~

The supplier table
would now look like thi

supplier_id	supplier_name	city	state
100	Microsoft	Boise	Idaho
200	Google	Mountain View	California
300	Oracle	Redwood City	California
400	Kimberly-Clark	Irving	Texas
500	Tyson Foods	Springdale	Arkansas
600	SC Johnson	Racine	Wisconsin
700	Dole Food Company	Westlake Village	California
800	Flowers Foods	Thomasville	Georgia
900	Electronic Arts	Redwood City	California

Practice Exercise #1.5: (delete)



Q5: Based on the suppliers table, delete the supplier record whose state is 'California' and supplier_name is not Google:

Solution: `DELETE FROM suppliers WHERE state = 'California'
AND supplier_name <> 'Google';`

There would be 3 records deleted and the suppliers table would now look like this:

supplier_id	supplier_name	city	state
100	Microsoft	Redmond	Washington
200	Google	Mountain View	California
400	Kimberly-Clark	Irving	Texas
500	Tyson Foods	Springdale	Arkansas
600	SC Johnson	Racine	Wisconsin
800	Flowers Foods	Thomasville	Georgia

Practice Exercise #2



Sample *employees* table for the queries:

```
CREATE TABLE employees
( employee_number int NOT NULL,
  last_name char(50) NOT NULL,
  first_name char(50) NOT NULL,
  salary int,
  dept_id int, emailid varchar(100),
  CONSTRAINT employees_pk PRIMARY KEY (employee_number)
);
```

Practice Exercise #2:



```
INSERT INTO employees
(employee_number, last_name, first_name, salary, dept_id, emailid)
VALUES
(1001, 'Smith', 'John', 62000, 500, 'smith@gmail.com'),
(1002, 'Anderson', 'Jane', 57500, 500, 'anderson@gmail.com'),
(1003, 'Everest', 'Brad', 71000, 501, 'everest@ny.com'),
(1004, 'Horvath', 'Jack', 42000, 501, 'horvath@yahoo.com');
(1005, 'Mohan', 'Radha', 55000, 502, 'mohan@yahoo.com');
```

Practice Exercise #2:

1. Select all fields from the *employees* table whose salary is less than or equal to \$52,500 (no sorting is required)
2. Select the details of the employee whose first name starts with B.
3. Print name and Id of employee whose email-id is in Gmail.
4. Select the details of the employee who work either for department 501 or 502.
5. Select all the employees working in department 501 in descending order of their salary.

Solution for Practice Exercise #2



- Solution 1:

SELECT * FROM employees WHERE salary <= 52500;

- Output:

employee_number	last_name	first_name	Salary	dept_id	Emailid
1004	Horvath	Jack	42000	501	horvath@yahoo.com

- Solution 2:

SELECT * FROM employees WHERE first_name like 'b%'

- Output:-

employee_number	last_name	first_name	salary	dept_id	emailid
1003	Everest	Brad	71000	501	Everest@ny.com

Solution for Practice Exercise #2



- Solution 3:

SELECT employee_number as E_no, last_name, first_name FROM employees where emailid like '%@gmail.com';

- Output

E_no	last_name	first_name
1001	smith	John
1002	Anderson	Jano

- Solution 4:

SELECT * FROM employees WHERE dept_id=501 or dept_id=502; or

SELECT * FROM employees WHERE dept_id in (501, 502)

Output:-

employee_number	last_name	first_name	salary	dept_id	emailid
1003	Everest	Brad	71000	501	Everest@ny.com
1004	Horvath	Jack	42000	501	Horvath@yahoo.com
1005	Mohan	Radha	55000	502	mohan@yahoo.com

Solution for Practice Exercise #2



- Solution 5:

`SELECT * FROM employees WHERE dept=501 ORDER BY salary DESC;`

- Output

employee_number	last_name	first_name	salary	dept_id	emailid
1003	Everest	Brad	71000	501	Everest@ny.com
1004	Horvath	Jack	42000	501	Horvath@yahoo.com

Practice Exercise #3

Sample *customers* table:

Cust_id	First_name	Last_name	Gender	Phone_number
100	Steven	Austin	M	515.123.4567
101	Neena	Singh	F	515.124.4568
102	Bruce	King	M	516.124.4569
103	David	Russell	M	590.123.4560
104	Den	Lee	M	590.124.4561
105	John	Gates	M	590.423.4565
106	Amit	Banda	M	515.124.4550
107	Sundar	Bhatt	M	590.124.4566
108	Clara	Doran	F	515.123.4562

Practice Exercise #3:

1. Write a query to update the portion of the phone_number in the customers table, within the phone number the substring '124' will be replaced by '999'.
2. Write a query to get the details of the customers where the length of the first name greater than or equal to 6.
3. Write a query to extract the last 4 character of phone numbers.
4. Write a query that displays the first name and the length of the first name for all customers whose name starts with the letters 'D', 'J' or 'N'. Give each column an appropriate label. Sort the results by the employees' first names.
5. Write a query to display the length of first name for employees where last name contain character 'a' after 2nd position.

Solution Practice Exercise #3:

- Solution Q1:

```
UPDATE customers SET phone_number = REPLACE(phone_number, '124',  
'999')  
WHERE phone_number LIKE '%124%';
```

- Solution Q2:

```
SELECT * FROM customers WHERE LENGTH(first_name) >= 6;
```

- Solution Q3:

```
SELECT SUBSTRING(phone_number, 9, 4) as 'Ph.no' FROM customers;
```

- Solution Q4:

```
SELECT first_name "Name", LENGTH(first_name) "Length" FROM customers  
WHERE first_name LIKE 'D%' OR first_name LIKE 'J%' OR first_name LIKE 'N%'  
ORDER BY first_name ;
```

- Solution Q5:

```
SELECT first_name, last_name FROM customers WHERE INSTR(last_name,'A')
```

Thank you



DATABASE DESIGN

BITS Pilani
Pilani Campus

SESSION - 8





DEPENDENCY AND NORMALIZATION

Learning Objectives

- What is Functional Dependency?
- Different types of dependencies
 - (i) Functionally dependent
 - (ii) Partially dependent
 - (iii) Transitive dependent
- What is Normalization ?
- Different types of Normal forms
 - i. First normal form
 - ii. Second normal form
 - iii. Third normal form
 - iv. Boyce Codd normal form
- Exercises

Functional Dependency

What is a Functional Dependency?

Functional Dependency (FD) determines the relation of one attribute to another attribute in a database management system (DBMS) system.

Functional dependency helps you to maintain the quality of data in the database. A functional dependency is denoted by an arrow \rightarrow .

If attribute A functionally determines attribute B we write this as $A \rightarrow B$.

Functional Dependency plays a vital role to find the difference between good and bad database design.

Assume that an employee table consists of the data about employees who are working in a company as

Functional Dependency

EMPLOYEE

<u>EmpNo</u>	EmpName	Dept	Salary
E101	Zaheer	HR	78000
E102	Basheer	Production	67000
E103	Naveen	Sales	54000
E110	Siva	System	60000
E112	Shankar	Production	39000

In this example, if we know the value of EmpNo(Employee number), we can obtain EmpName, Dept and salary.

By this, we can say that the EmpName, Dept and Salary are **functionally dependent** on Employee number.

Functional Dependency

It is represented as follows

EmpNo → EmpName, Dept ,Salary

In other words, we can say that for each EmpNo, there is a single value for attributes EmpName, Dept and Salary.

There are various types of functional dependencies:

1. Fully Functional dependency
2. Partial Functional dependency
3. Transitive Functional dependency

The dependencies which are used in normalization process are

1. Partial dependency and
2. Transitive dependency

Full Functional Dependency

In full functional dependency an attribute or a set of attributes uniquely determines another attribute or set of attributes.

If a relation R has attributes X, Y, Z with the dependencies $X \rightarrow Y$ and $X \rightarrow Z$ which states that those dependencies are fully functional.

Example:

In the given Employee relation, if we know the value of EmpNo(Employee number), we can obtain EmpName, Dept and salary.

By this, we can say that the EmpName, Dept and Salary are **fully functionally dependent** on Employee number.

Partial Functional Dependency

Partial Dependency

Assume that a table consists of two attributes are combined for forming a primary key.

All non key attributes may either depend on both the key attributes or depend on any one of the key attribute.

If all non key attributes in the table are functionally dependent on any one of the key attributes (not dependent on both), the dependency is called as partial dependency

Example :

Assume that STUDENT table consists of student information as follows:

Partial Functional Dependency

STUDENT **Key attributes** **Non-key attributes**

StuID	FacultyID	FacName	Course
S001	F1001	Mr. A	DBSA
S101	F1005	Mr. B	Datastructures
S006	F1001	Mr. A	DBSA
S010	F0024	Mr.C	OOPS
S001	F1111	Mr.D	Web
Technology			
S101	F0012	Mr. M	Unix
S101	F0024	Mr.C	OOPS

Partial Functional Dependency

In the above table, the non key attribute FacName is functionally dependent on only FacultyID (not on the other key attribute StuID).

So the table has Partial Dependency.

Note :- If a table has a single key attribute, it does not have partial dependency by default.

Transitive dependency

Transitive dependency

We know that a table consists of both key attributes and non key attributes in which the key attributes are used to identify /locate a record /a tuple uniquely.

All the non key attributes are functionally dependent on the key attribute (s).

There are some instances in which a non key attribute may depend on another non key attribute.

If a table has such type of attribute, the table is said to be having transitive dependency.

Transitive dependency

Example: Assume that a STUDENT table consists of the student records as follows:

STUDENT

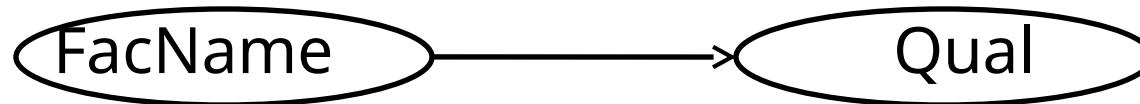
<u>StuID</u>	<u>FacultyID</u>	FacName	Qual
S001	F1001	Mr. A	ME , PhD
S101	F1005	Mr. B	ME
S006	F1001	Mr. A	ME , PhD
S010	F0024	Mr.C	Mtech

In the above table, StuID and FacultyID are the key attributes.

FacName and Qual are the non key attributes.

Functional Dependency

The non key attribute Qual is functionally dependent on another non key attribute, FacName.



Hence, the table STUDENT has transitive dependency.

If a table has some dependencies like partial dependency and transitive dependency, there are lot of redundancy (duplication).

Functional Dependency

Advantages of Functional Dependency

The advantages of functional dependency are

(i) Functional Dependency enables to identify and avoid data redundancy.

Therefore same data do not repeat at multiple locations in the database

(ii) It helps you to maintain the quality of data in the database

(iii) It helps you to define meanings and constraints of databases

(iv) It helps you to identify bad designs

(v) It helps you to find the facts regarding the database design

Data Normalization

Data normalization is the process of organizing data in a database in order to minimize redundancy and increase data integrity.

Functional dependencies play an important part in data normalization.

With the help of functional dependencies we are able to identify the primary key, candidate key in a table which in turns helps in normalization.

NORMALIZATION

Normalization is the analysis of functional dependency between attributes of user views.

It reduces the complex user view to a set of small and stable subgroups of fields / relations.

This process helps to design a logical data model known as conceptual data model.

There are different levels of normalization as listed below:

1. **Un normalized form (UNF)** – The relation is in *un normalized form* if it satisfies all the properties of a relation
2. **First normal form (1NF)** – The relation is said to be in the *first normal form* if it is already in the un normalized form and has no repeating group.

NORMALIZATION

3. Second normal form (2NF) – The relation is said to be in the *second normal form* if it is already in the first normalized form and has no partial dependency.

In a relation having more than one key field , a sub set of non – key fields may depend on all the key fields but another subset / a particular non – key fields may depend on only one of the key fields (ie., not depend on all the key fields).

Such dependency is called as the partial dependency.

4. Third normal form (3NF) – The relation is said to be in the *third normal form* if it is already in the second form and has no transitive dependency.

In a relation, there may be dependency among non – key fields.

Such dependency is called as transitive dependency.

NORMALIZATION

5. Boyce - Codd form (BCNF) - The relation is said to be in the *Boyce - codd normal form* if it is already in the third normal form and every determinant is a candidate key. It is a stricter version of the third normal form. A determinant is any field (simple or composite field) on which some other field is fully functionally dependent.

NORMALIZATION - Example

CASE PROBLEM

Let us consider the two report requirements namely , INVOICE and REORDER REPORT of XYZ book house as shown in the following figures:

XYZ Book House						
Chennai – 600 001						
Customer N0		: C10001				
Customer Name		: WIPRO Systems Date : XX / XX / XX				
Customer Address		: Chennai - 125				
Book# Rs.	Title	Author's Name	No.of copies	Price (Rs.)	Amount	
B10001	Production and Operations Management	Benher	50	250		
B10010	System Simulation	Gordon	400	8000		
B10011	Engineering Mathematics	Benher	40	300		12000

Invoice of a book company

NORMALIZATION

The XYZ book house supplies the books to the institutions which frequently buy books from them.

The customers are very keen on the data item namely author.

Normalization of Invoice report.

The invoice report is represented in the relation mentioned above.

The relation is named as INVOICE.

This is in un normalized form.

It is represented as

INVOICE (Cus_no, Cus_name, Cus_Add, Book_No, Title, Author_name, Auth_coun, Unit_price)

NORMALIZATION

The amount in the last column of the invoice for each book can be computed by multiplying the number of copies (Qty) with their respective unit price.

Similarly, the grand total at the bottom of the invoice representing the invoice amount can be calculated by summing the entries in the last column of the invoice.

Similarly, the date at the top right corner of the invoice can be printed using the system clock.

A sample instance of the data of the INVOICE relation is shown in the following table.

This is nothing but the file containing all the data items of the INVOICE report.

NORMALIZATION

Sample Invoice Form

Cus_no	Cus_name	Cus_Add	Book_No	Title	Author_name	Auth_coun	Unit_price
C1001	SIST	Chennai-1	B1001	Applied OR	Manmohan	INDIA	355
C1001	SIST	Chennai-1	B1010	OOPS with C++	Schildt	US	450
C1001	SIST	Chennai-1	B2000	Java	Herbert	US	525
C1002	RMK	Chennai-10	B2000	Java	Herbert	US	525

The above relation is in un normalized form.

FIRST NORMAL FORM

First normal form (1NF) – The relation is said to be in the *first normal form* if it is already in the un normalized form and has no repeating group.

In the above relation the fields Cus_no , Cus_name and Cus_Add are forming repaeating group.

NORMALIZATION

This will result in redundancy of data for the first three fields.

It leads to inconsistency of data.

Hence the relation can be divided into two sub relations to remove the repeating group as

CUSTOMER (Cus_no , Cus_name and Cus_Add)

CUSTOMER-BOOK(Cus_no , Book_No , Title , Author_name , Auth_coun, Unit_price)

The corresponding relations in their normalized form are as follows:

NORMALIZATION

CUSTOMER relation [Relation 2]

<u>Cus_no</u>	Cus_name	Cus_Add
C1001	SIST	Chennai-1
C1001	SIST	Chennai-1
C1001	SIST	Chennai-1
C1002	RMK	Chennai-10

CUSTOMER-BOOK relation [Relation 3]

<u>Cus_no</u>	<u>Book_No</u>	Title	Author_name	Auth_coun	Unit_price
C1001	B1001	OOPS with C++	Manmohan	INDIA	355
C1001	B1010	Applied OR	Schildt	US	450
C1001	B2000	Java	Herbert	US	525
C1002	B2000	Java	Herbert	US	525

Now each relation is in 1NF.

NORMALIZATION

SECOND NORMAL FORM

The relation is said to be in the *second normal form* if it is already in the first normalized form and has no partial dependency.

In relation 2 (CUSTOMER relation), the number of key fields is only one and hence there is no scope for partial dependency.

The absence of partial dependency in a relation takes it to the 2NF without any modification.

But in relation 1 (CUSTOMER-BOOK relation), the field Qty depends on Cus_No as well as Book_No.

But the non key fields (Title , Author_name, Unit_price) depend only on Book_No.

This situation is an example of partial dependency.

NORMALIZATION

Partial dependency will result into insertion update and deletion anomalies.

Insertion Anomaly :- In the relation 3, if we want to insert data of a new book , there must be at least one customer buying it.

It means that a new book can be inserted into the CUSTMOER-BOOK relation only when the first customer buys the book.

Update Anomaly :- In the relation, if we want to change any non key fields , it will result into inconsistency because the same is available in more than one record.

This is an example for updation anomaly.

NORMALIZATION

Deletion Anomaly :- Once the customer order is fully satisfied, the customer's record should be deleted.

If the customer is the only person buying a particular book , the information of the book will not be available when his/her record is deleted.

To overcome this, relation 3 is divided into two relations as relation 4 and relation 5 where both of them are in the 2nd Normal form as

SALES(Cus_no , Book_No Qty)

BOOK-AUTHOR(Book_No , Title , Author_name , Unit_price)

NORMALIZATION

The corresponding relations are:

SALES relation (Relation 4)

<u>Cus_no</u>	<u>Book_No</u>
C1001	B1001
C1001	B1010
C1001	B2000
C1002	B2000

BOOK-AUTHOR relation (Relation 5)

<u>Book_No</u>	Title	Author_name	Auth_coun
B1001	Applied OR	Manmohan	INDIA
B1010	OOPS with C++	Schildt	US
B2000	Java	Herber	US
B2000	Java	Herbert	US

In the relations , there are no partial dependencies.

NORMALIZATION

So both the relations are in second normal form.

THIRD NORMAL FORM

The relation is said to be in the *third normal form* if it is already in the second form and has no transitive dependency.

In a relation, there may be dependency among non - key fields.

Such dependency is called as transitive dependency.

In relation 4 (SALES relation), there is only one non - key field and hence there is no scope for any dependency between the non - key fields.

It means that there is no transitive dependency.

Hence the relation 3 can be treated to be in 3NF without any modification. But in relation 5 (BOOK-AUTHOR relation), the field Auth_Coun depends on author's name.

NORMALIZATION

This situation is an example of the transitive dependency. Transitive dependency will result into insertion update and deletion anomalies.

Insertion Anomaly :- In case, the book has resident authors who are in the process of developing new books, it will be difficult to include the author's detail in relation 5 .

It means that there should be at least one published book to insert the details of a resident author.

Update Anomaly :- Since the author's details are duplicated , there will be significant redundancy and inconsistency.

Some problem arises when the updation is done.

Deletion Anomaly :- If the only book of a resident author is not reprinted , then the respective author's data is lost.

NORMALIZATION

To overcome these anomalies, relation 5 can be divided into two relations as

BOOK (Book_No , Title , Author_name)

AUTHOR(Author_name , Author_Coun)

The corresponding relations are:

BOOK relation (Relation 6)

<u>Book_No</u>	Title	Author_name
B1001	Applied OR	Manmohan
B1010	OOPS with C++	Schildt
B2000	Java	Herbert
B2000	Java	Herbert

NORMALIZATION

AUTHOR relation (Relation 7)

<u>Author_name</u>	Auth_coun
Manmohan	INDIA
Schildt	US
Herbert	US
Herbert	US

In the relations , there are no partial dependencies.

So both the relations are in second normal form.

For practical applications , it is sufficient to normalize up to 3NF.

NORMALIZATION

4. Boyce Codd Normal form (BCNF)

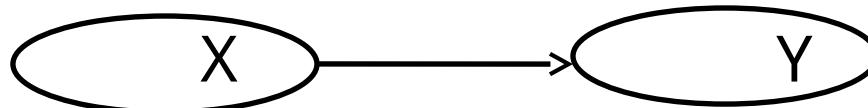
A table is said to be in **Boyce Codd Normal form** iff it is already in third normal form and every determinant in the relation is a candidate key.

Candidate Key – It is an attribute or a collection of attributes that uniquely identify a record

Determinant- The attribute on the left hand side of the arrow in representing the functional dependency is called as the determinant.

Let Y is an attribute that is functionally dependent on another attribute X and is denoted as

NORMALIZATION



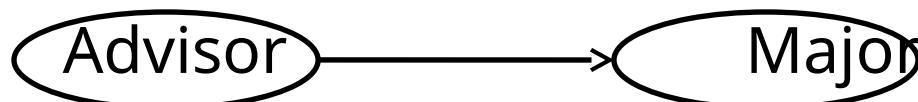
where the attribute X is the determinant

Example: Assume that a student table consists of student data with the following restrictions:

- (i) An advisor can handle only one major subject
- (ii) A Major Subject may have multiple advisors

According to the restrictions, we identify that Major is functionally dependent on Advisor.

So we have the representation



NORMALIZATION

The table is given as

<u>StuID</u>	<u>AdvisorID</u>	Major_Sub	Major_Grade
S1	A1	M1	A+
S1	A2	M1	A+
S3	A2	M1	B
S4	A1	M1	C
S5	A2	M1	A

The table is in third normal form because it has no repeating group, no partial dependency and transitive dependency.

NORMALIZATION

But it is not in BCNF because the determinant (Advisor) is not a candidate key (ie. it is only a part of the key) because the combination of the attributes StuID and AdvisorID is used as the candidate key.

To make the table to be in Boyce Codd Normal form , we have to split the table into two tables such as

STUDENT (StudID, AdvisorId, Major_Grade)

Advisor(AdvisorID, Major_Sub)

The records of these tables are

NORMALIZATION

STUDENT

<u>StuID</u>	<u>AdvisorID</u>	Major_Grade
S1	A1	A+
S1	A2	A+
S3	A2	B
S4	A1	C
S5	A2	A

ADVISOR

<u>AdvisorID</u>	Major_Sub
A1	M1
A2	M1

In both the tables every determinant is a candidate key. So the tables are in Boyce Codd Normal form

NORMALIZATION

ADVANTAGES OF NORMALIZATION

Here we can see why normalization is an attractive prospect in RDBMS concepts.

- 1) A smaller database can be maintained as normalization eliminates the duplicate data. Overall size of the database is reduced as a result.
- 2) Better performance is ensured which can be linked to the above point. As databases become lesser in size, the passes through the data becomes faster and shorter thereby improving response time and speed.
- 3) Narrower tables are possible as normalized tables will be fine-tuned and will have lesser columns which allows for more data records per page.

NORMALIZATION

- 4) Fewer indexes per table ensures faster maintenance tasks (index rebuilds).
- 5) Also realizes the option of joining only the tables that are needed.

DISADVANTAGES OF NORMALIZATION

- 1) More tables to join as by spreading out data into more tables, the need to join table's increases and the task becomes more tedious.
- 2) Data model becomes extremely difficult to query against as the data model is optimized for applications

Exercise

Let us take an EMPLOYEE table consisting of the employee records as follows:

EMPLOYEE

<u>EmpId</u>	EmpName	DeptId	Salary
E001	Nikhil	D100	76000
E002	Suresh	D106	45200
E003	Kamal	D100	55000
E004	Kavya	D101	49000
E005	Ramya	D108	56240
E006	Keshav	D101	34500
E007	Midhun	D100	49000
E010	Saran	D101	34800

Exercise

Query 1: Write SQL command to display all employee records who are getting salary between Rs.40000 and Rs.60000

Solution:

Select * from EMPLOYEE where salary >40000 AND salary <60000;

Query 2: Write SQL command to display all employee records Whose name begins with 'K'

Solution:

Select * from EMPLOYEE where EmpName LIKE 'K%';

Query 3: Write SQL command to display the number of departments in the company

Solution:

Select distinct(count(DeptId)) from EMPLOYEE;

Exercise

Query 4: Write SQL command to add 5% increment in salary those who are getting the salary \leq average salary

Solution:

Update EMPLOYEE

```
set salary=salary+0.05*salary where  
salary<=(select avg(salary) from EMPLOYEE);
```

Query 5: Write SQL command to display the DeptId that have less than 2 people in it

Solution:

```
Select DeptId, count(DeptId) from EMPLOYEE where  
group by DeptId having count(DeptId)<2;
```

Exercise

Query 6: Write SQL command to fetch departments along with the total salaries paid for each department

Solution:

Select DeptId,sum(salary) from EMPLOYEE group by DeptId;

Query 7: Write SQL command to display the name and salary of the employees who earn more than EmpId 'E004'

Solution:

Select EmpName,salary from EMPLOYEE where
salary > (select salary from EMPLOYEE where
EmpId='E004');

Exercise

Query 8: Write SQL command to fetch the list of employees with the same salary

Solution:

```
Select distinct E1.EmpId,E1.EmpName,E1.Salary from  
EMPLOYEE E1,EMPLOYEE E2 where  
E1.salary=E2.salary and E1.EmpId!=E2.EmpId;
```

Query 9: Write SQL command to display the employee records whose names are neither 'Ramya' nor 'Midhun'

Solution:

```
Select * from EMPLOYEE where  
EmpName not in ('Ramya','Midhun');
```

Exercise

Assume that a Store has different branches to sell the products.

It does the sales through internet also. The data are given in the following tables:

STORES		
<u>Store location</u>	Sales	<u>Date</u>
Chennai	280050	5-5-2020
Bangalore	432000	6-5-2020
Madurai	120000	5-5-2020
Chennai	490000	6-5-2020
Bangalore	343250	10-5-2020
Chennai	123000	15-5-2020

Exercise

INTERNET

<u>Date</u>	Sales
5-5-2020	45000
7-5-2020	34300
8-5-2020	10450
13-5-202	23325
16-5-2020	45000

Examples:

1. If we want to find out all the dates where there is a sales transaction, the SQL command is

`SELECT Date FROM STORES UNION SELECT Date FROM INTERNET;`

Exercise

2. If we want to find out all the dates where there are both store sales and internet sales, we use the following SQL command

```
SELECT Date FROM STORES INTERSECT  
        SELECT Date FROM INTERNET;
```

3. We want to find out all the dates where there are store sales, but no internet sales. To do so, we use the following SQL statement:

```
SELECT Date FROM STORES MINUS  
        SELECT Date FROM INTERNET;
```

Thanks



DATABASE DESIGN

BITS Pilani
Pilani Campus

SESSION 9



STORAGE DEVICES AND FILE ORGANIZATION

Learning Objectives

- Different types of storage devices
 - (i) Magnetic storage (ii) Optical Storage (iii) Flash memory (iv) Online and Cloud (v) Paper Storage
- What is file organization
- Objectives of file organization
- Types of file organization
 - (i) Sequential (ii) Heap (iii) ISAM (iv) B+ tree
 - (v) Cluster (vi) Hashing File Organization

STORAGE DEVICES

Devices which are used to store data is known as Storage device.

In computers, the storage devices are classified into two categories as Primary storage devices and Secondary storage devices.

The devices which are used to store data temporarily, are called primary storage devices.

Example : RAM -Random Access Memory and is a volatile Memory to store temporarily.

The devices which are used to store data permanently , are known as secondary storage devices .

Example: A hard disk -It is a non volatile memory to store data permanently

STORAGE DEVICES

A secondary storage can be removable, internal, or external. Alternatively referred to as **digital storage**, **storage**, **storage media**, or **storage medium**, a **storage device** is any hardware capable of holding information either temporarily or permanently.

The following picture shows an example of an external Secondary storage device, [Drobo](#).



STORAGE DEVICES

The secondary storage are further classified into the following categories.

1. Magnetic storage
2. Optical Storage
3. Flash memory
4. Online and Cloud
5. Paper Storage

1. Magnetic storage devices - Today, magnetic storage is one of the most common types of storage used with computers. This technology is found mostly on extremely large hybrid hard drives. Examples for magnetic storage devices are : Floppy disks, Hard Drive, Magnetic strip, Tape cassette, etc.,

STORAGE DEVICES

2. Optical storage devices - Another common type of storage is optical storage, which uses lasers and lights as its method of reading and writing data.

Examples are: CD-ROM disc, CD-R, CD-RW, Blue-Ray disc, DVD-R, DVD-RW, etc.,

3. Flash memory devices - Flash memory has replaced most magnetic and optical media because it is cheaper, more efficient and reliable device.

Examples are : Memory card, Compact Flash, USB Flash drive, Jump drive, SD card, etc.,

4. Online and cloud - Storing data online and in cloud storage is becoming popular as people need to access their data from more than one device.

Examples are Cloud Storage and Network media

STORAGE DEVICES

5. Paper storage - Early computers had no method of using any of the technologies above for storing information and had to rely on paper. Today, these forms of storage are rarely used or found.

Examples are : OMR and Punched Cards

Note : A hard copy could be considered a form of paper storage that is still widely used, although it cannot be easily used to input data back into a computer without the aid of OCR.

FILE ORGANIZATION

File – A file is named collection of related information that is recorded on secondary storage such as magnetic disks, magnetic tapes and optical disks.

What is File Organization?

File Organization refers to the logical relationships among various records that constitute the file, particularly with respect to the means of identification and access to any specific record.

In simple terms, Storing the files in certain order is called file Organization.

File Structure refers to the format of the label and data blocks and of any logical control record.

FILE ORGANIZATION

Objectives of File Organization

- a. Optimal selection records – records should be accessed as fast as possible
- b. Easy Transaction – Any Insert , Update or Delete operation on records in the file should be performed easily without any difficulty
- c. Avoid duplication – There should not be any duplicate record
- d. Efficient Storing – Records of the file should be stored efficiently in the file so that the cost to perform any operation on the records in the file should be minimum.

FILE ORGANIZATION

Types of File Organizations

There are various methods to organize the files.

Each method has its own advantages and disadvantages on the basis of access or selection .

Thus it is the responsibility of the programmer to decide the best suited file organization method .

Some types of File Organizations are :

1. Sequential File Organization
2. Heap File Organization
3. ISAM method
4. B+ Tree File Organization
5. Clustered File Organization
6. Hash File Organization

FILE ORGANIZATION

We will be discussing each of the file Organizations along with differences and advantages/ disadvantages of each file Organization methods.

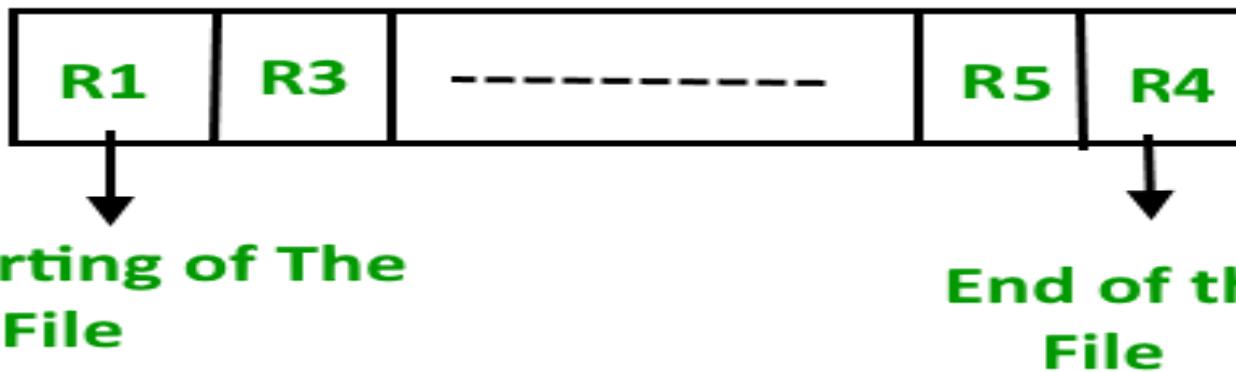
I. Sequential File Organization

The simplest method for file Organization is the Sequential method. In this method the file are stored one after another in a sequential / linear manner.

There are two ways to implement the sequential method:

1. Pile File Method – This method is quite simple, in which we store the records in a sequence i.e one after other in the order in which they are inserted into the tables.

FILE ORGANIZATION

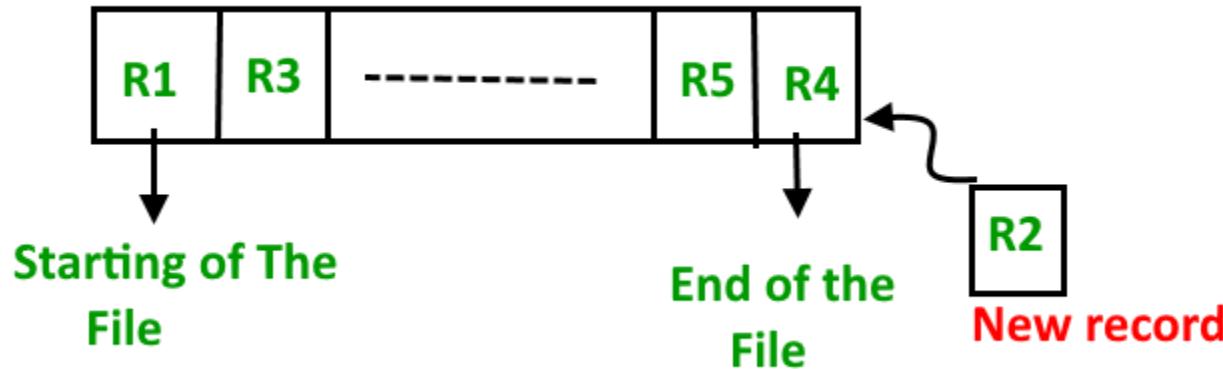


Insertion of new record into the file

Let R1, R3 and so on upto R5 and R4 be records in the sequence.

Here, records are nothing but the tuples or rows in any table. Suppose a new record R2 has to be inserted in the sequence, then it is simply placed at the end of the file as follows

FILE ORGANIZATION



2. Sorted File Method -In this method, As the name itself suggest whenever a new record has to be inserted, it is always inserted in a sorted (ascending or descending) manner. Sorting of records may be based on any primary key or any other key.

FILE ORGANIZATION

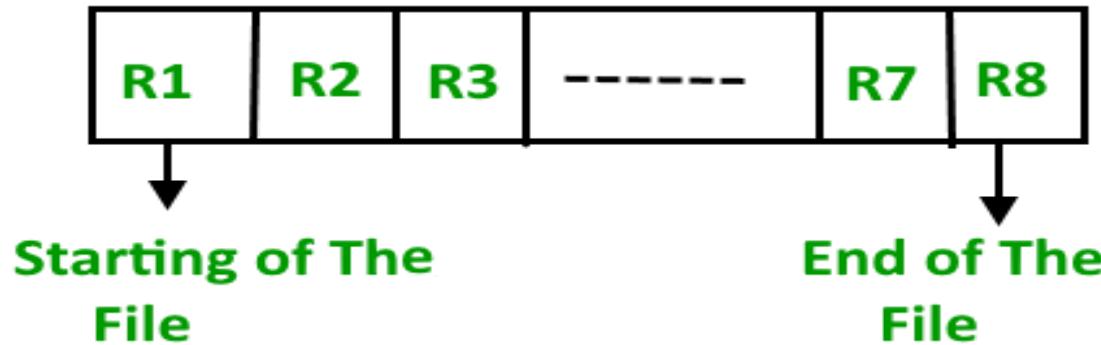


a. Insertion of new record into the file

Let us assume that there is a pre existing sorted sequence of four records R1, R3, and so on up to R7 and R8.

Suppose a new record R2 has to be inserted in the sequence, then it will be inserted at the end of the file and then it will sort the sequence .

FILE ORGANIZATION



Advantages of Sequential File Organization

- Fast and efficient method for huge amount of data.
- It is the simplest design.
- Files can be easily stored in sequential access device like magnetic tapes.
- So these devices are cheaper storage mechanisms.

FILE ORGANIZATION

Disadvantages of Sequential File Organization

- Time is wasted as we cannot go to a particular record randomly but we have to move in a sequential manner. It takes more time
- Sorted file method is inefficient method because it takes much time and space for sorting records(moving the records).

II. Heap File Organization

It is also the simplest form of file organization as sequential file organization.

The records are inserted at the end of the file.

This method does not require any sorting or ordering of the records.

Once the data block is full , the record to be inserted may be stored in any new block.

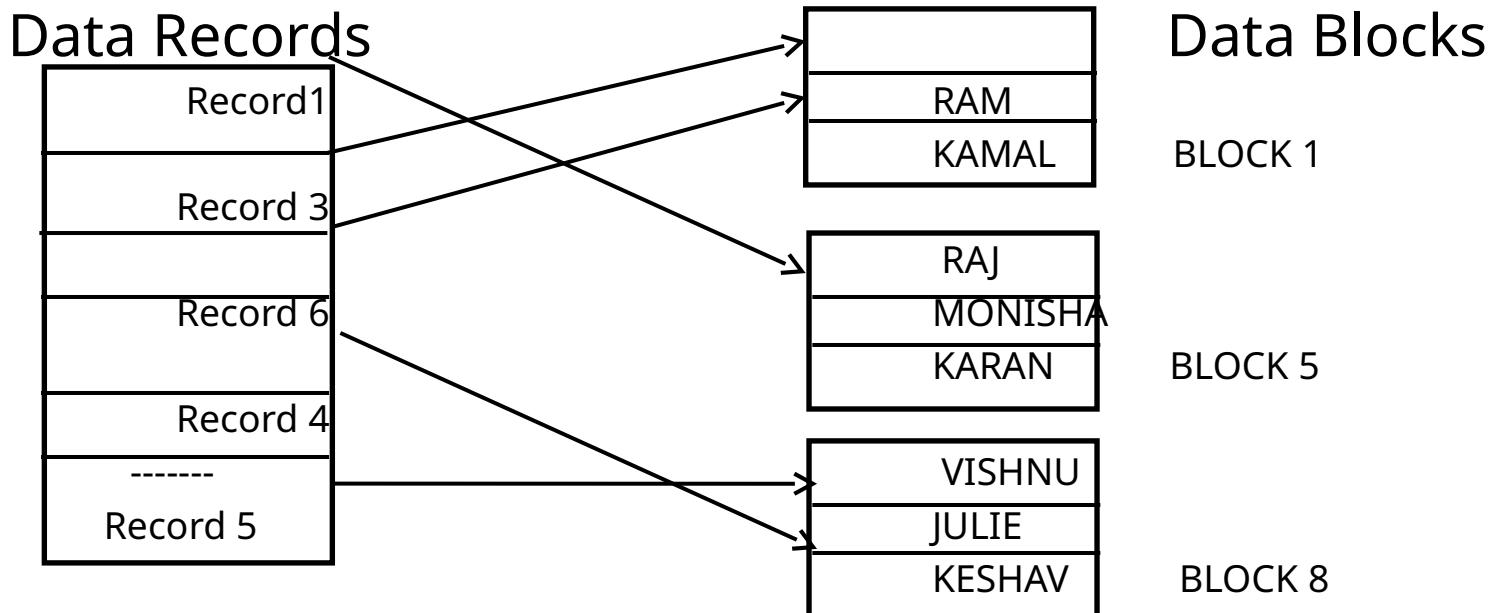
FILE ORGANIZATION

The new block need not be the very next block.

This method is also like the sequential method but data blocks to store are not selected sequentially.

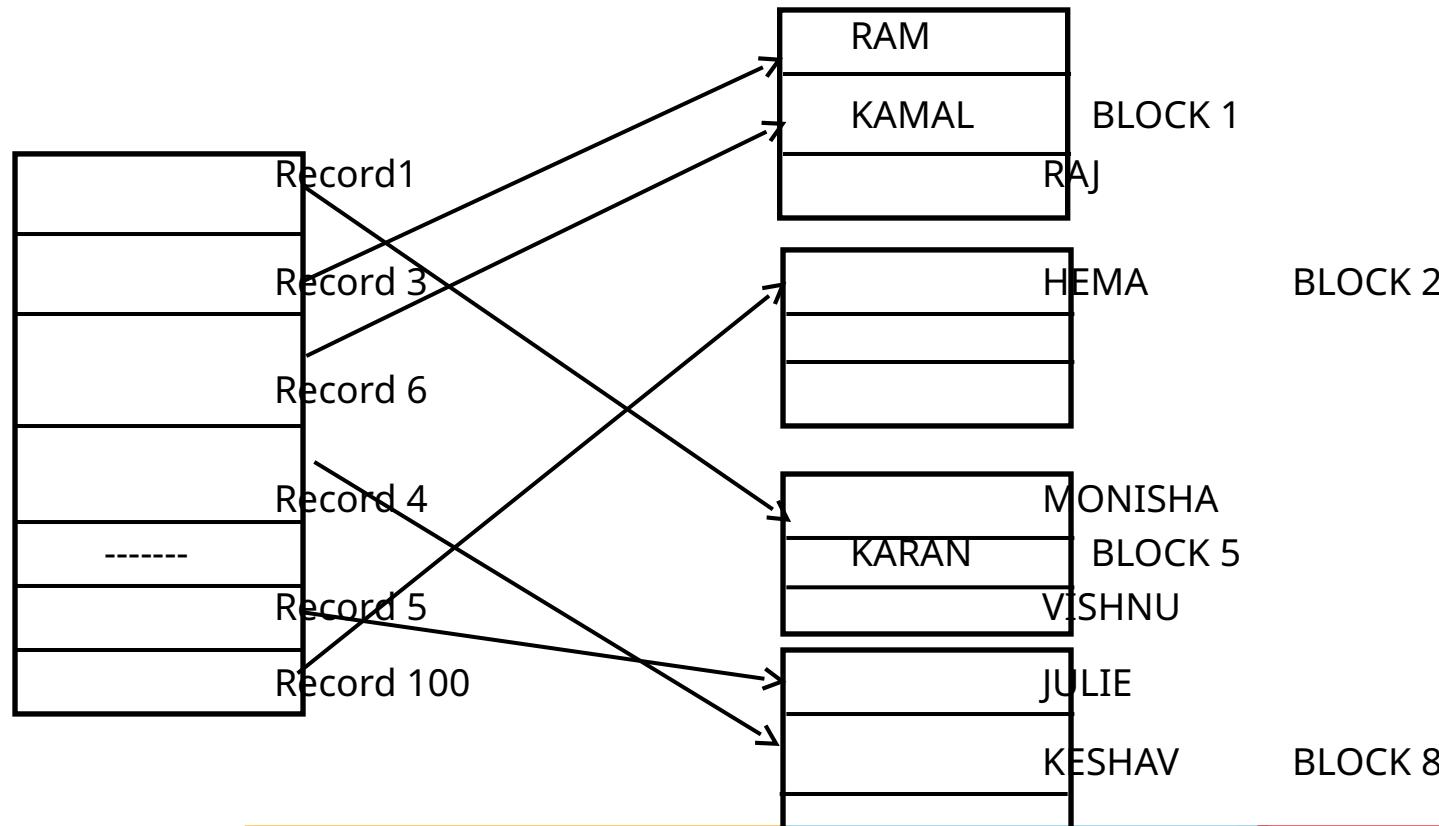
Any block can be selected and it is the responsibility of the DBMS to store and manage the records in the file.

Assume that the records are stored as follows:



FILE ORGANIZATION

If a new record is to be inserted , it can be inserted in any of the blocks 2,3,4,6 and 7 or block 8 which has a space to insert the record. When the record (record 100) is inserted , the file is organized in the following format:



FILE ORGANIZATION

If we want to search, delete or update data in heap file organization then we will traverse the data from the beginning of the file till we get the requested record.

Thus if the database is very huge, searching, deleting or updating the record will take a lot of time.

Advantages of Heap File Organization

- Fetching and retrieving records is faster than sequential record because they can be accessed randomly
- When there is a huge number of data needs to be loaded into the database at a time, then this method of file Organization is best suited.

Disadvantages of Heap File Organization

Problem of unused memory blocks.

Inefficient for larger databases.

FILE ORGANIZATION

III. Indexed Sequential Access Method (ISAM) file organization

ISAM is an advanced sequential file organization method. The records are stored according to the primary key of the records in the file.

For each primary key , an index value is generated and mapped with the record.

Index is nothing but the address of the record of the file.

Index is a table which has two columns representing Primary key and the address of the record stored in the file.

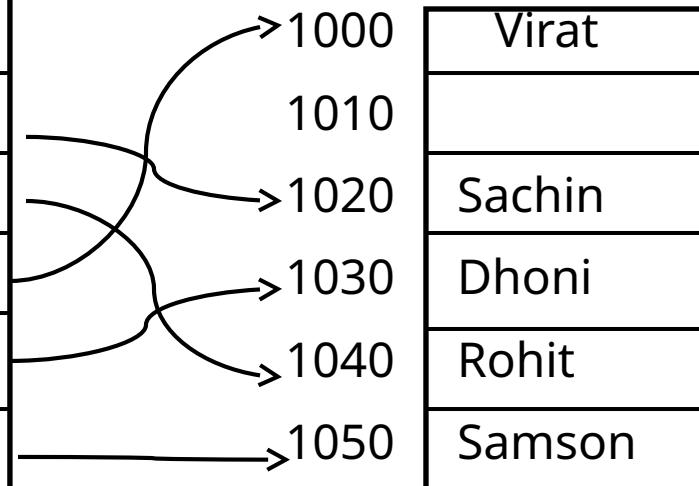
The records are stored as in the following diagram

FILE ORGANIZATION

Data Records

Record#	Primary Key
R1	Sachin
R2	Rohit
R3	Virat
R4	Dhoni
R5	Samson

Data blocks in the file



Primary Key

INDEX Table

Address

Sachin	1020
Rohit	1040
Virat	1000
Dhoni	1030
Samson	1050

FILE ORGANIZATION

Advantages

- It is faster because the records can be accessed randomly since we know their addresses with the help of index table
- It supports range retrieval and partial retrieval. Index is based on the primary key value, we can retrieve data for the given range of value. In the same way, partial value can also be easily searched ie. , Students name starts with 'Sa'

Disadvantages

- Extra space is required for storing index
 - When the records are inserted, the file have to be restructured to maintain the sequence
 - When a record is deleted, the space occupied by it should be released. Otherwise it reduces its performance.
-

FILE ORGANIZATION

IV. B+ Tree File Organization

It is an advanced method of ISAM file organization.

It uses the same concept of key – index combination to store the records but is represented in the tree structure.

B+ tree is similar to binary search tree but it can have more than two leaf nodes.

It stores all the records only at the leaf nodes.

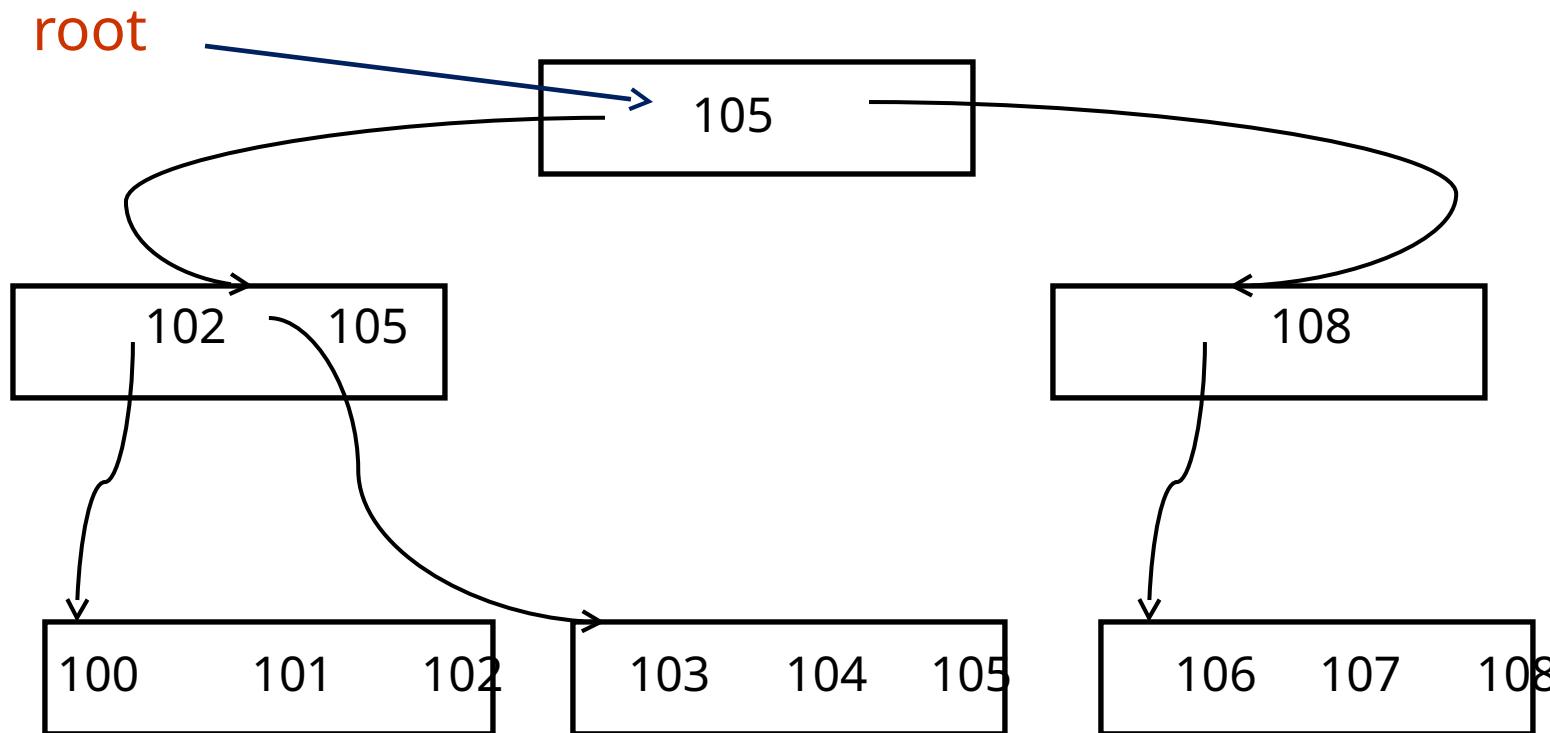
All non terminal nodes (other than leaves) are used to keep the pointer (address) to leaf nodes and are not used for storing the records.

Consider the following table consists of the student records in which StuId is used as the primary key:

FILE ORGANIZATION

<u>StuID</u>	StName	Department
100	Ramesh	Production
101	Hemanth	CSE
102	Jerin	CSE
103	Levin	IT
104	Baskar	CSE
105	Santhosh	IT
106	Meena	Mech
107	David	Production
108	Geetha	CSE
B+ tree is represented as follows:		

FILE ORGANIZATION



Leaves representing the records

FILE ORGANIZATION

- ✓ The above tree has one main node called as the root node. The root node has the key value 105
- ✓ Leaf node contains the data stored in the sorted order
- ✓ Intermediary layers does not have the records actually but they are the pointers to the left child having the data less than or equal to the data in the present node
- ✓ All leaf nodes are balanced. All leaf nodes should exist in the same level

To insert or to delete the records , do the operations on the leaf nodes after searching process finds the correct location to insert.

There should be some adjustments and or insertion on the intermediary nodes.

FILE ORGANIZATION

Advantages of B+ Tree File Organization

- Tree traversal is easier and faster.
- Searching becomes easy as all records are stored only in leaf nodes and are sorted sequential linked list.
- There is no restriction on B+ tree size. It may grows/shrink as the size of data increases/decreases.

Disadvantages of B+ Tree File Organization

- Inefficient for static tables.

FILE ORGANIZATION

V. Cluster File Organization

In cluster file organization, two or more related tables/records are stored within same file known as clusters.

These files will have two or more tables in the same data block and the key attributes which are used to map these table together are stored only once.

Thus it lowers the cost of searching and retrieving various records in different files as they are now combined and kept in a single cluster.

For example we have two tables or relation Employee and Department.

These table are related to each other.

FILE ORGANIZATION

EMPLOYEE

EmpID	EmpName	Address	DeptId
E101	MOHINDER	DELHI	D105
E102	SUNIL	BOMBAY	D102
E103	ROY	CHENNAI	D101
E104	SHYAM	CHENNAI	D103
E105	DRAVID	BANGALORE	D101
E106	SASTRI	BOMBAY	D102
E107	MAMTA	BANGALORE	D104
E108	KAVYA	CHENNAI	D103
E109	RAMYA	TRIVANDRAM	D102

DEPARTMENT

DeptId	DeptName
D101	SALES
D102	PRODUCTION
D103	IT
D104	HR
D105	QM

Therefore these table are allowed to combine using a join operation and can be seen in a cluster file.

FILE ORGANIZATION

CLUSTER KEY



EmpID	EmpName	Address	DeptId	DeptName
E101	MOHINDER	DELHI	D105	QM
E102	SUNIL	BOMBAY	D102	PRODUCTION
E103	ROY	CHENNAI	D101	SALES
E104	SHYAM	CHENNAI	D103	IT
E105	DRAVID	BANGALORE	D101	SALES
E106	SASTRI	BOMBAY	D102	PRODUCTION
E107	MAMTA	BANGALORE	D104	HR
E108	KAVYA	CHENNAI	D103	IT
E109	RAMYA	TRIVANDRAM	D102	PRODUCTION

If we have to insert, update or delete any record we can directly do so. Data is sorted based on the primary key or the key with which searching is done. **Cluster key** is the key with which joining of the table is performed.

FILE ORGANIZATION

VI. Hashing File Organization

It is an efficient technique to directly search the location of desired data on the disk without using index structure.

Data is stored at the data blocks whose address is generated by using hash function.

The memory location where these records are stored is called as data block or data bucket.

Hash File Organization :

Data bucket – Data buckets are the memory locations where the records are stored.

These buckets are also considered as Unit Of Storage.

FILE ORGANIZATION

Hash Function - Hash function is a mapping function that maps all the set of search keys to actual record address.

Generally, hash function uses primary key to generate the hash index - address of the data block.

Hash function can be simple mathematical function to any complex mathematical function.

Hash Index-The prefix of an entire hash value is taken as a hash index.

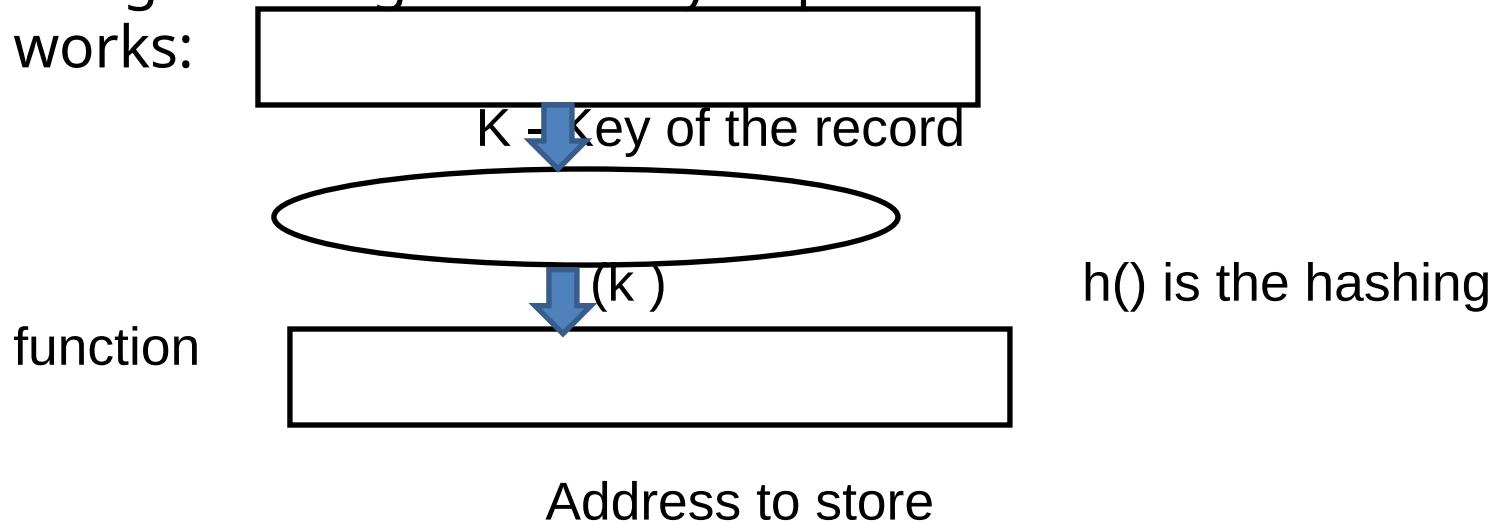
Every hash index has a depth value to signify how many bits are used for computing a hash function.

These bits can address 2^n buckets.

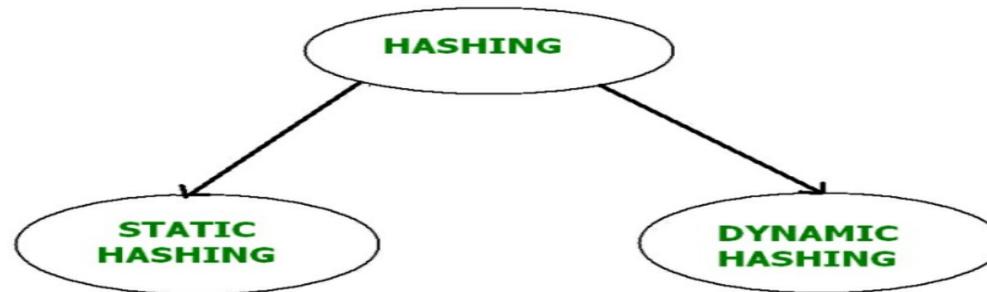
When all these bits are consumed, then the depth value is increased linearly and twice the buckets are allocated.

FILE ORGANIZATION

Below given diagram clearly depicts how hash function works:



Hashing



FILE ORGANIZATION

I. STATIC HASHING

In static hashing, when a search-key value is provided, the hash function always computes the same address.

For example, if we want to generate address for STUDENT_ID = 76 using mod (5) hash function, it always result in the same bucket address 1.

There will not be any changes to the bucket address here.

Hence number of data buckets in the memory for this static hashing remains constant throughout.

Operations –

Insertion – When a new record is inserted into the table, The hash function h generate a bucket address for the new record

based on its hash key K . Bucket address = $h(K)$

FILE ORGANIZATION

Searching - When a record needs to be searched, The same hash function is used to retrieve the bucket address for the record.

For Example, if we want to retrieve whole record for ID 76, and if the hash function is mod (5) on that ID, the bucket address generated would be 1. Then we will directly go to address 1 and retrieve the whole record for ID 76.

Here ID acts as a hash key.

Deletion - If we want to delete a record, Using the hash function we will first fetch the record which is supposed to be deleted.

Then we will remove the records for that address in memory.

FILE ORGANIZATION

Updation – The data record that needs to be updated is first searched using hash function, and then the data record is updated.

Now, If we want to insert some new records into the file but if the data bucket address generated by the hash function is not empty or the data already exists in that address.

This becomes a critical situation to handle.

This situation in the static hashing is called **bucket overflow**.

How will we insert data in this case?

There are several methods provided to overcome this situation.

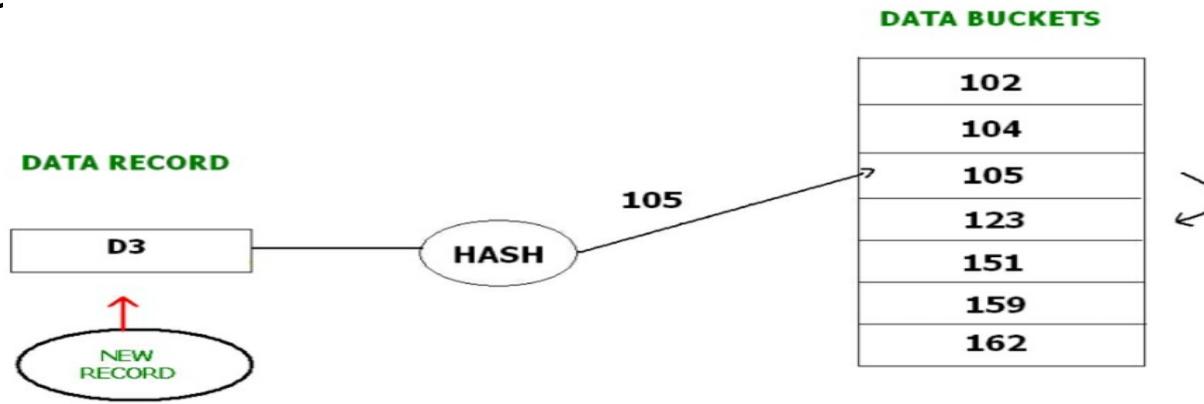
Some commonly used methods are discussed below:

FILE ORGANIZATION

a. Open Hashing

In Open hashing method, next available data block is used to enter the new record, instead of overwriting older one. This method is also called linear probing. For example, D3 is a new

record which needs to be inserted , the hash function generates address as 105. But it is already full. So the system searches next available data bucket 123 and assigns D3 to it.



FILE ORGANIZATION

b. Closed hashing

In Closed hashing method, a new data bucket is allocated with same address and is linked after the full data bucket.

This method is also known as overflow chaining.

For example, we have to insert a new record D3 into the tables.

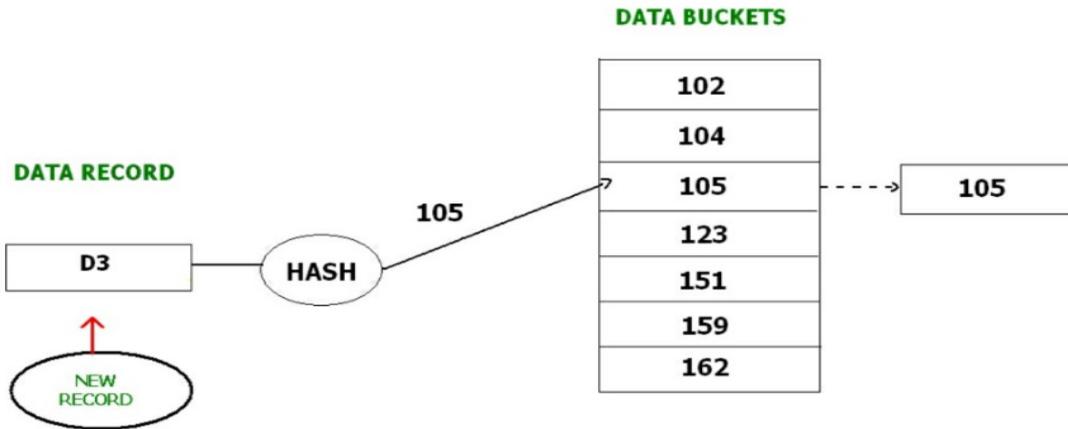
The static hash function generates the data bucket address as 105.

But this bucket is full to store the new data.

In this case, a new data bucket is added at the end of 105 data bucket and is linked to it.

Then new record D3 is inserted into the new bucket.

FILE ORGANIZATION



C. Quadratic probing :

Quadratic probing is very much similar to open hashing or linear probing.

Here, The only difference between old and new bucket is linear. Quadratic function is used to determine the new bucket address.

FILE ORGANIZATION

d. Double Hashing :

Double Hashing is another method similar to linear probing. Here the difference is fixed as in linear probing, but this fixed difference is calculated by using another hash function.

That's why the name is double hashing.

II. DYNAMIC HASHING

- The dynamic hashing method is used to overcome the problems of static hashing like bucket overflow.
- In this method, data buckets grow or shrink as the records increases or decreases. This method is also known as Extendable hashing method.
- This method makes hashing dynamic, i.e., it allows insertion or deletion without resulting in poor performance.

FILE ORGANIZATION

Search a key

- First, find the hash address of the key.
- Check how many bits are used in the directory, and these bits are called as i .
- Take the least significant i bits of the hash address.
- This gives an index of the directory.
- Using the index, go to the directory and find bucket address where the record might be.

Insert a new record

- Firstly, you have to follow the same procedure for retrieval, ending up in some bucket.
- If there is still space in that bucket, then place the record in it.

FILE ORGANIZATION

- If the bucket is full, then we will split the bucket and redistribute the records.

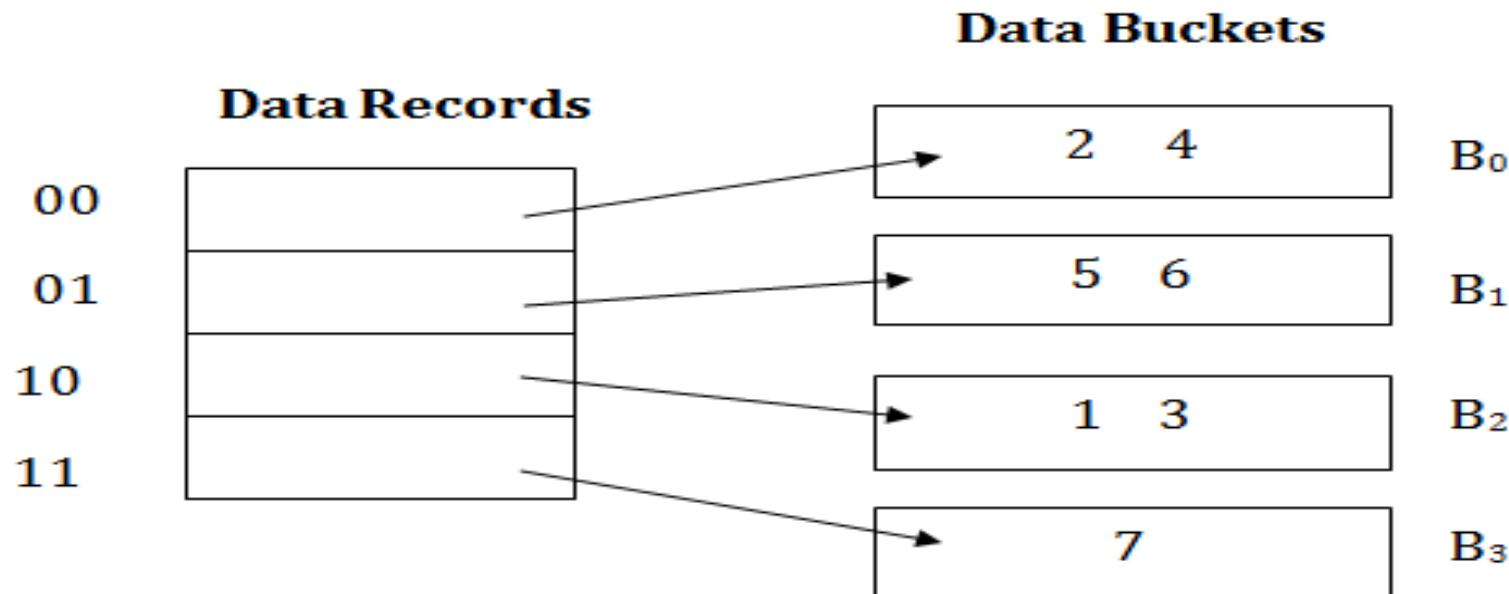
For example:

Consider the following grouping of keys into buckets, depending on the prefix of their hash address:

Key	Hash address
1	11010
2	00000
3	11110
4	00000
5	01001
6	10101
7	10111

FILE ORGANIZATION

The last two bits of 2 and 4 are 00. So it goes into bucket B0.
The last two bits of 5 and 6 are 01, so it goes into bucket B1.
The last two bits of 1 and 3 are 10, so it goes into bucket B2.
The last two bits of 7 are 11, so it goes into B3.



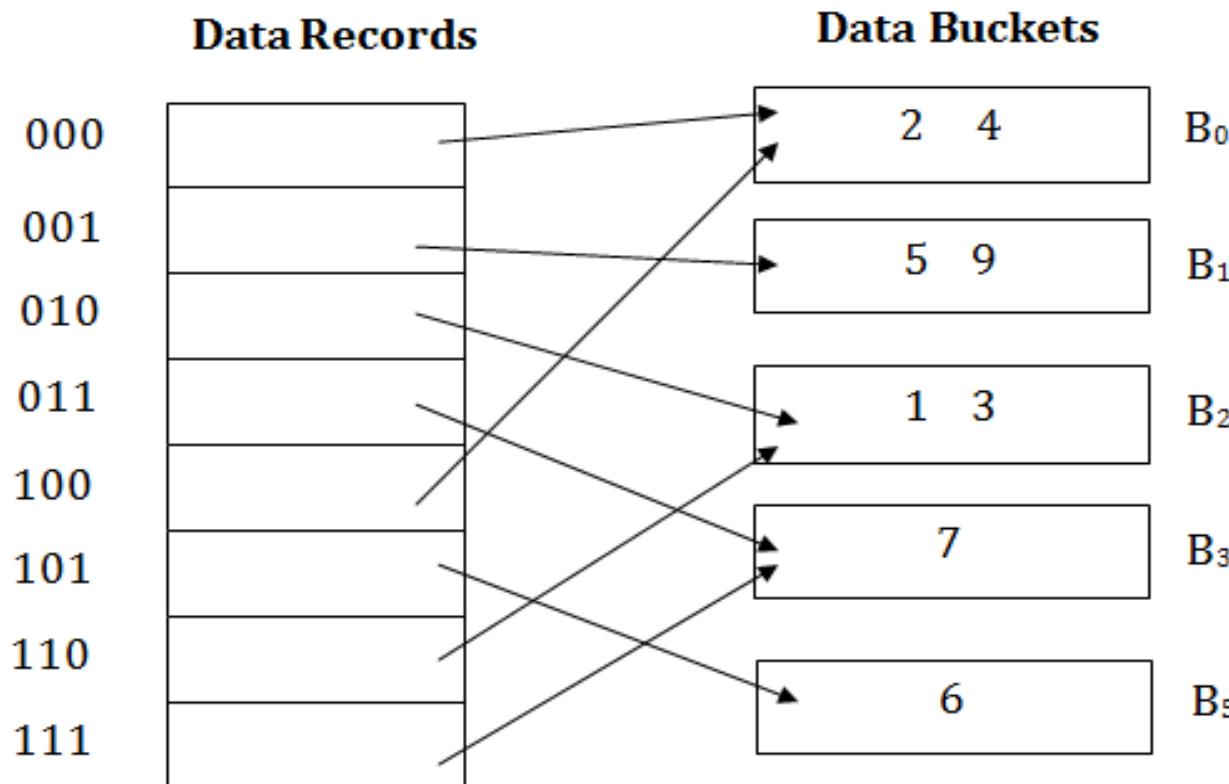
FILE ORGANIZATION

Insert key 9 with hash address 10001 into the above structure:

- Since key 9 has hash address 10001, it has to go into the first bucket. But bucket B1 is full, so it will get split.
- The splitting will separate 5, 9 from 6 since last three bits of 5, 9 are 001, so it will go into bucket B1, and the last three bits of 6 are 101, so it will go into bucket B5.
- Keys 2 and 4 are still in B0. The record in B0 pointed by the 000 and 100 entry because last two bits of both the entry are 00.
- Keys 1 and 3 are still in B2. The record in B2 pointed by the 010 and 110 entry because last two bits of both the entry are 10.

FILE ORGANIZATION

Key 7 are still in B3. The record in B3 pointed by the 111 and 011 entry because last two bits of both the entry are 11.



FILE ORGANIZATION

Advantages of dynamic hashing

- In this method, the performance does not decrease as the data grows in the system.
- In this method, memory is well utilized as it grows and shrinks with the data. There will not be any unused memory lying.
- This method is good for the dynamic database where data grows and shrinks frequently.

Disadvantages of dynamic hashing

- In this method, if the data size increases then the bucket size is also increased. These addresses of data will be maintained in the bucket address table. This is because the data address will keep changing as buckets grow and shrink. If there is a huge increase in data, maintaining the bucket address table becomes tedious.
- In this case, the bucket overflow situation will also occur. But it might take little time to reach this situation than static hashing.

Thanks



BITS Pilani

Pilani | Dubai | Goa | Hyderabad

Database Design





Contact Session 10: Indexing used in Databases & **B+ trees**

- Storage Types and Characteristics
- Secondary Storage Devices
- Placing File Records on Disk
- Hashing Techniques
- Other Primary File Organizations

Reference : Chapter 16 - Disk Storage, Basic File Structures, Hashing, and Modern Storage Architectures (Ramez Elmasri & Shamkant B. Navathe, Fundamentals of Database Education, 7th Edition, 2017)

- **Buffering of Blocks**
- **Indexing used in Databases with examples.**
- **Understanding B+ trees and**
- **Constructing B+ trees with examples**
- **Construction of B+ trees and Problem Solving**

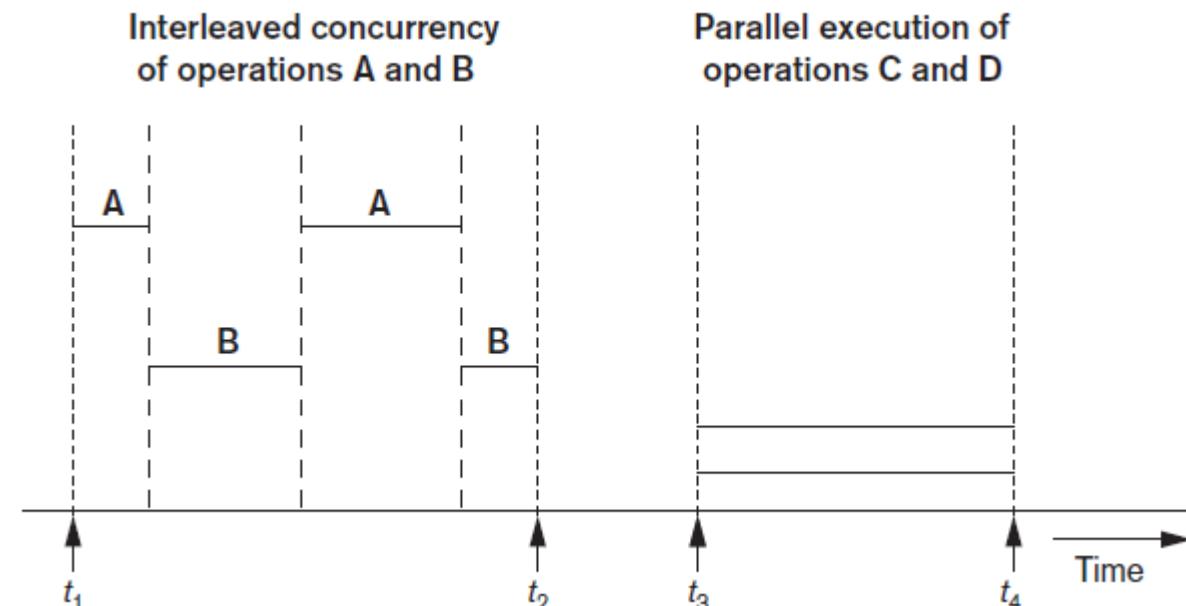
Reference : Chapter 17 - Indexing Structures for Files and Physical Database Design (Ramez Elmasri & Shamkant B. Navathe, *Fundamentals of Database Systems*, Pearson Education, 7th Edition, 2017)

Buffering of Blocks

A buffer is a memory location used by a database management system (DBMS) to temporarily hold data that has recently been accessed or updated in the database.

This buffer, often referred to as a database buffer, acts as a link between the programs accessing the data and the physical storage devices.

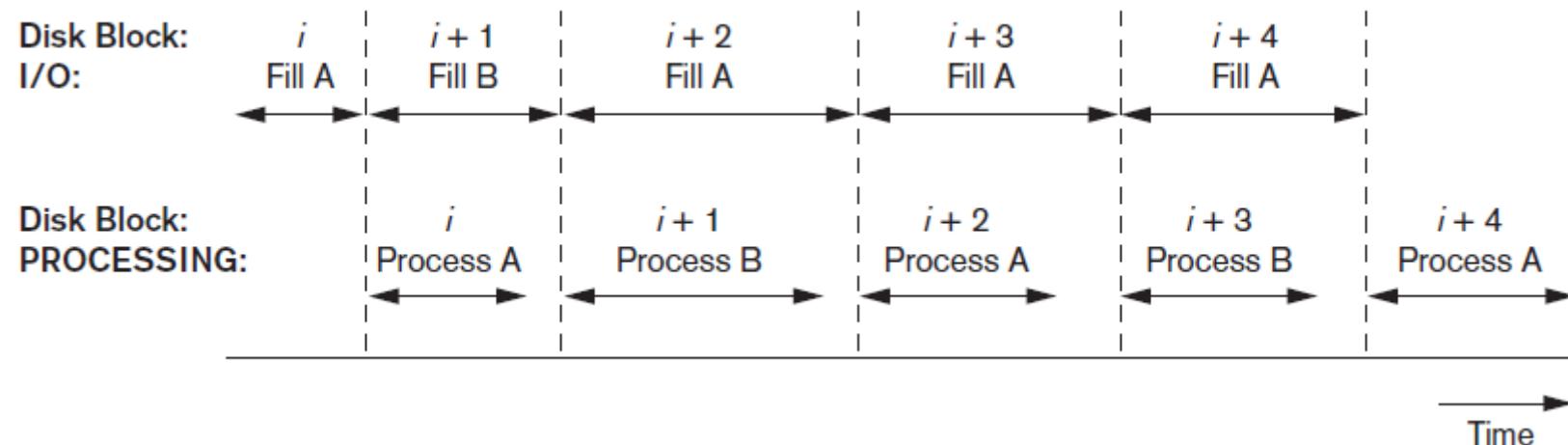
Buffering is most useful when processes can run concurrently in parallel



Interleaved concurrency versus parallel execution

Buffering of Blocks (cont'd.)

Double buffering can be used to read continuous stream of blocks



Use of two buffers, A and B, for reading from disk

- Buffer management information
 - Pin count
 - Dirty bit
 - Buffer replacement strategies
 - Least recently used (LRU)
 - Clock policy
 - First-in-first-out (FIFO)
- Why do we need Buffer replacement strategies?

Buffer Management and Replacement Strategies

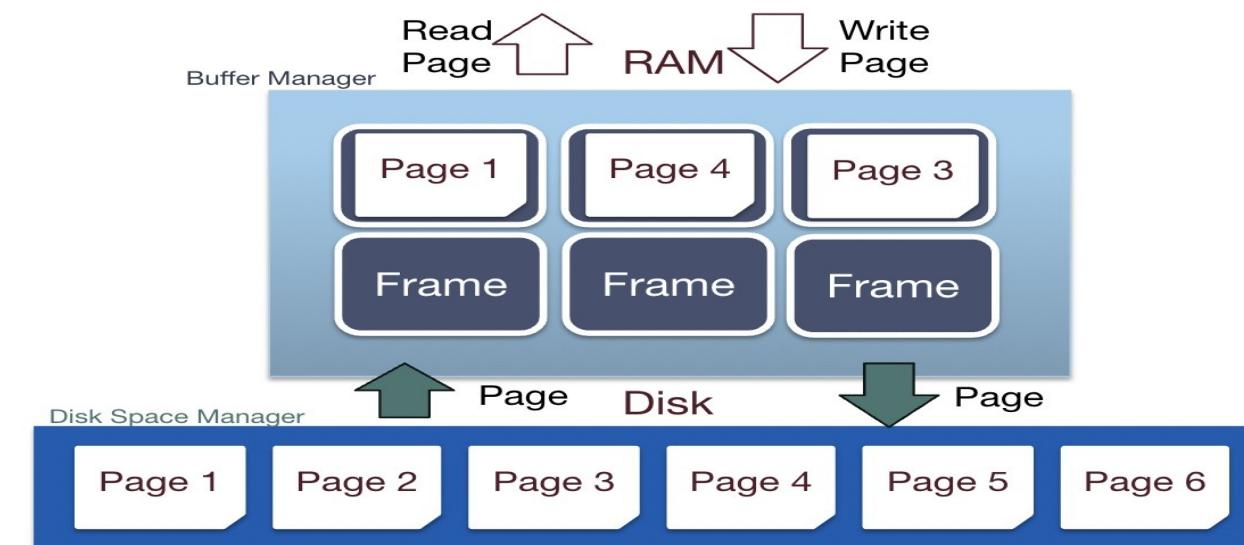


The buffer manager is responsible for managing pages in memory and processing page requests from the file and index manager.

Since space on memory is limited, so we cannot afford to store all pages in the buffer pool.

The buffer manager is responsible for the eviction policy, or choosing which pages to evict when space is filled up.

When pages are evicted from memory or new pages are read in to memory, the buffer manager communicates with the disk space manager to perform the required disk operations.



Buffer Pool : Memory is converted into a buffer pool by partitioning the space into frames that pages can be placed in. A buffer frame can hold the same amount of data as a page can (so a page fits perfectly into a frame). To efficiently track frames, the buffer manager allocates additional space in memory for a metadata table.

Frame ID	Page ID	Dirty Bit	Pin Count
0	5	1	3
1	3	0	1
2	10	1	0
3			

The table tracks 4 pieces of information:

- 1. Frame ID** that is uniquely associated with a memory address
- 2. Page ID** for determining which page a frame currently contains
- 3. Dirty Bit** for verifying whether or not a page has been modified
- 4. Pin Count** for tracking the number of requestors currently using a page

The records of data that we store are of variable length – Why?

This may be due to the following reasons

Reasons for variable-length records

- One or more fields have variable length
- One or more fields are repeating
- One or more fields are optional
- File contains records of different types

Record Blocking and Spanned Versus Unspanned Records



- File records allocated to disk blocks
- Two strategies can be used for allocation
- Spanned records
 - Larger than a single block
 - Pointer at end of first block points to block containing remainder of record
- Unspanned
 - Records not allowed to cross block boundaries

Record Blocking and Spanned Versus Unspanned Records (cont'd.)

Blocking factor

- Average number of records per block for the file

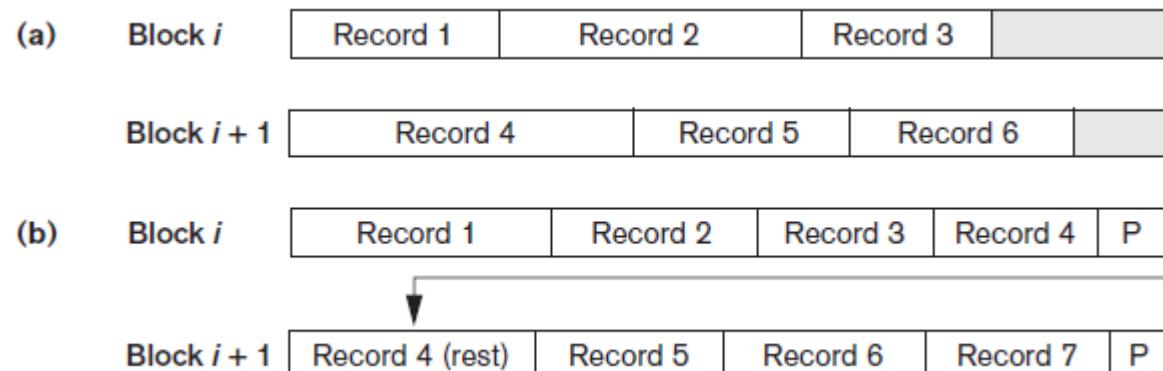


Figure 16.6 Types of record organization (a) Unspanned (b) Spanned

Record Blocking and Spanned Versus Unspanned Records (cont'd.)



- Allocating file blocks on disk
 - Contiguous allocation
 - Linked allocation
 - Indexed allocation

File header or descriptor is used to allocate and locate files in the memory
- File header (file descriptor)
 - Contains file information needed by system programs
 - Disk addresses
 - Format descriptions

Operations on Files



- Retrieval operations
 - No change to file data
- Update operations
 - File change by insertion, deletion, or modification
- Records selected based on selection condition

- Index / Indices are used to speed up record retrieval in response to certain search conditions
- Index structures provide secondary access paths
- Any field can be used to create an index
 - Multiple indexes can be constructed
- Most indexes based on ordered files
 - Tree data structures organize the index

17.1 Types of Single-Level Ordered Indexes



- Ordered index similar to index in a textbook
- Indexing field (attribute)
 - Index stores each value of the index field with list of pointers to all disk blocks that contain records with that field value
- Values in index are ordered
- Primary index
 - Specified on the ordering key field of ordered file of records

➤ Clustering index

- Used if numerous records can have the same value for the ordering field

➤ Secondary index

- Can be specified on any non-ordering field
- Data file can have several secondary indexes

- Ordered file with two fields
 - Primary key, $K(i)$
 - Pointer to a disk block, $P(i)$
- One index entry in the index file for each **block** in the data file
- Indexes may be dense or sparse
 - Dense index has an index entry for every search key value in the data file
 - Sparse index has entries for only some search values

Example: To create a primary index on the ordered file say for a set of students or employees, use the Name field as primary key, because that is the ordering key field of the file (assuming that each value of Name is unique).

Each entry in the index has a Name value and a pointer.

The first three index entries are as follows:

$\langle K(1) = (\text{Aaron, Ed}), P(1) = \text{address of block 1} \rangle$
 $\langle K(2) = (\text{Adams, John}), P(2) = \text{address of block 2} \rangle$
 $\langle K(3) = (\text{Alexander, Ed}), P(3) = \text{address of block 3} \rangle$

Primary Indexes (cont'd.)

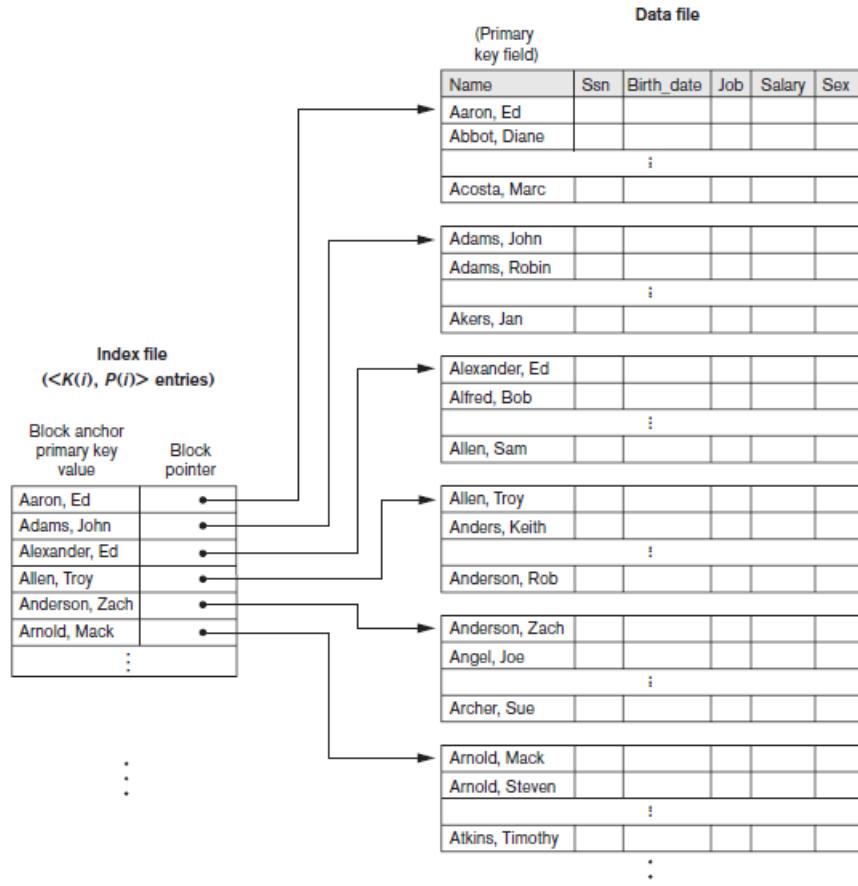


Figure 17.1 Primary index on the ordering key field of the file shown in previous slide

Example 1.

Suppose that we have an ordered file with $r = 3,00,000$ records stored on a disk with block size $B = 4,096$ bytes.

File records are of fixed size and are unspanned, with record length $R = 100$ bytes.

The blocking factor for the file would be $bfr = \lfloor (B/R) \rfloor = \lfloor (4,096/100) \rfloor = 40$ records per block.

The number of blocks needed for the file is $b = \lceil (r/bfr) \rceil = \lceil (300,000/40) \rceil = 7,500$ blocks.

A binary search on the data file would need approximately

$$\lceil \log_2 b \rceil = \lceil \log_2 7,500 \rceil = 12.87 \cong 13 \text{ block accesses.}$$

Now suppose that the ordering key field of the file is $V = 9$ bytes long, a block pointer is $P = 6$ bytes long, and we have constructed a primary index for the file.

The size of each index entry is $R_i = (9 + 6) = 15$ bytes, so the blocking factor for the index is $bfri = \lfloor (B/R_i) \rfloor = \lfloor (4,096/15) \rfloor = 273$ entries per block.

The total number of index entries r_i is equal to the number of blocks in the data file, which is 7,500.

The number of index blocks is hence $b_i = \lceil (r_i/bfri) \rceil = \lceil (7,500/273) \rceil = 28$ blocks.

To perform a binary search on the index file would need $\lceil (\log_2 b_i) \rceil = \lceil (\log_2 28) \rceil = 5$ block accesses.

To search for a record using the index, we need one additional block access to the data file for a total of $5 + 1 = 6$ block accesses—an improvement over binary search on the data file, which required 13 disk block accesses.

Primary Indexes (cont'd.)

- Major problem: insertion and deletion of records
 - Move records around and change index values
 - Solutions
 - Use unordered overflow file (\equiv Heap)
 - Use linked list of overflow records

➤ Clustering field

- File records are physically ordered on a non-key field without a distinct value for each record

➤ Ordered file with two fields

- Same type as clustering field
- Disk block pointer

Clustering Indexes (cont'd.)

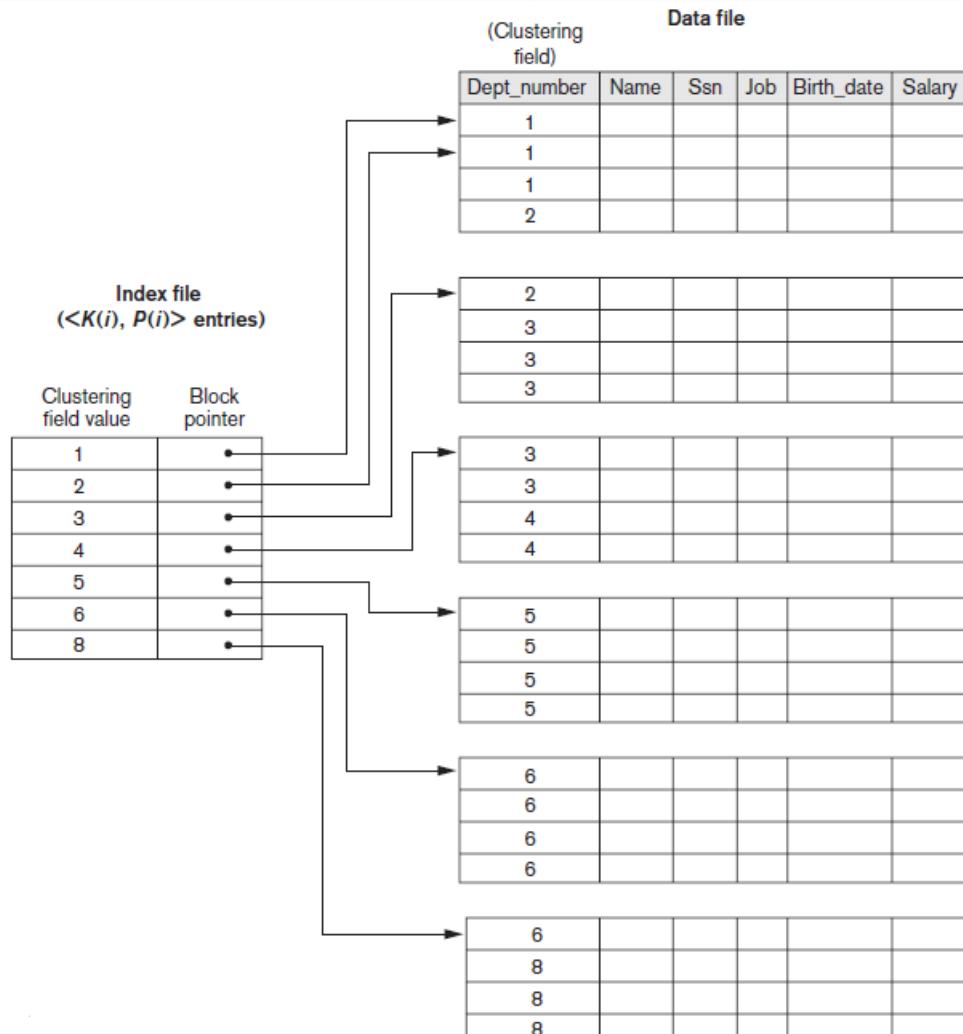


Figure 17.2 A clustering index on the `Dept_number` ordering nonkey field of an `EMPLOYEE` file

Example 2.

Suppose that we consider the same ordered file with $r = 300,000$ records stored on a disk with block size $B = 4,096$ bytes.

Imagine that it is ordered by the attribute Zipcode and there are 1,000 zip codes in the file (with an average 300 records per zip code, assuming even distribution across zip codes.)

The index in this case has 1,000 index entries of 11 bytes each (5-byte Zipcode and 6-byte block pointer) with a blocking factor $bfri = \lfloor (B/Ri) \rfloor = \lfloor (4,096/11) \rfloor = 372$ index entries per block.

The number of index blocks is hence $bi = \lceil (ri/bfri) \rceil = \lceil (1,000/372) \rceil = 3$ blocks.

To perform a binary search on the index file would need $\lceil (\log_2 bi) \rceil = \lceil (\log_2 3) \rceil = 2$ block accesses.

Again, this index would typically be loaded in main memory (occupies 11,000 or 11 Kbytes) and takes negligible time to search in memory.

One block access to the data file would lead to the first record with a given zip code.

- **Provide secondary means of accessing a data file**
 - Some primary access exists
- **Ordered file with two fields**
 - Indexing field, $K(i)$
 - Block pointer or record pointer, $P(i)$
- **Usually need more storage space and longer search time than primary index**
 - Improved search time for arbitrary record

Secondary Indexes (cont'd.)

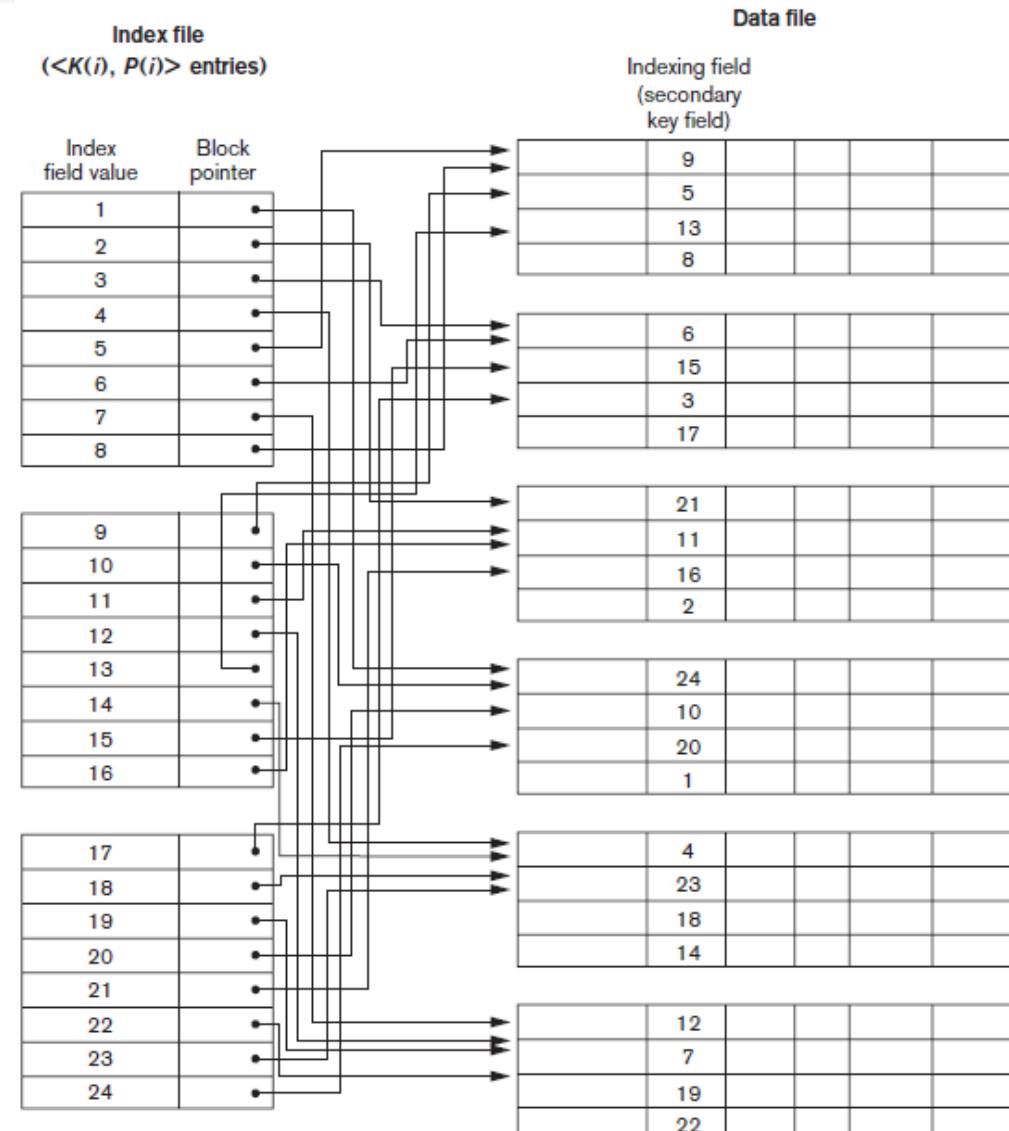
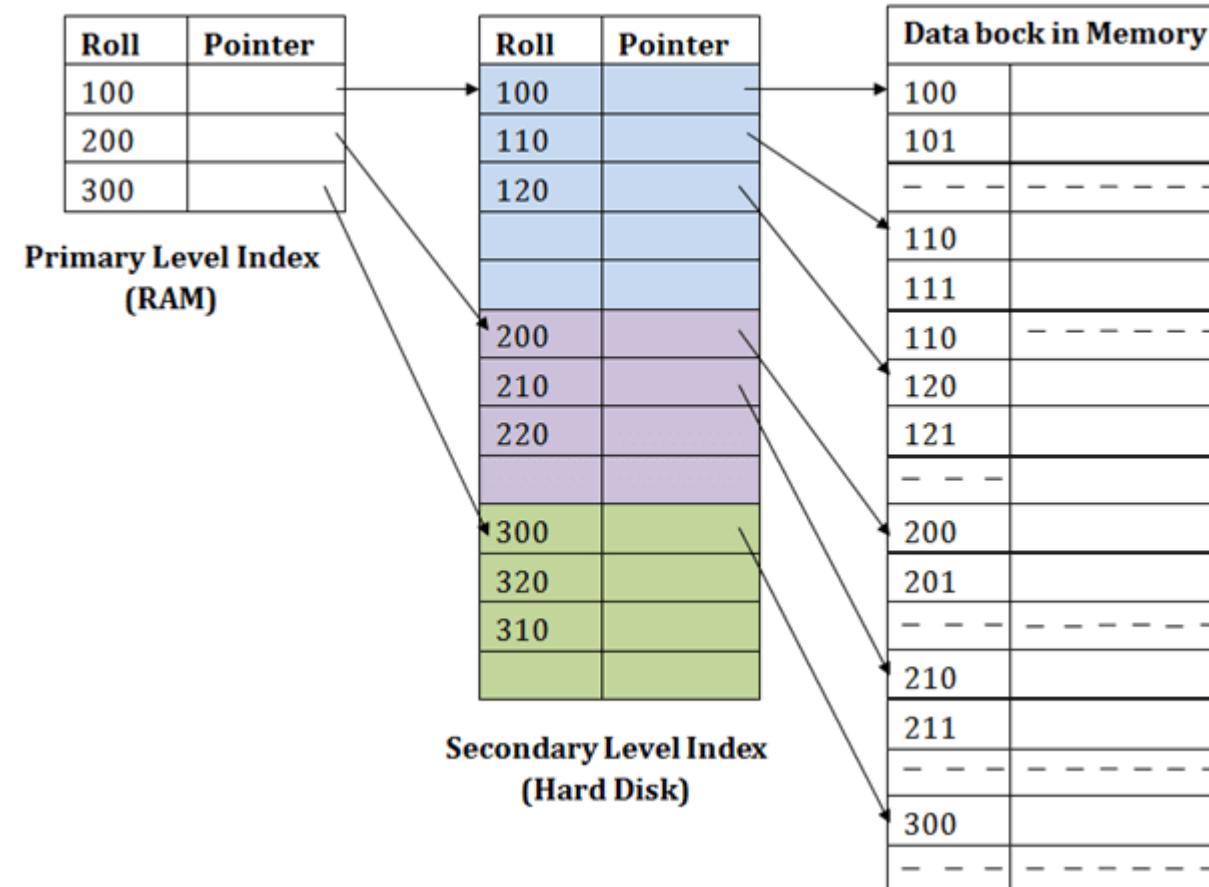


Figure 17.4 Dense secondary index (with block pointers) on a nonordering key field of a file.

Secondary Indexes (cont'd.)



Example 3.

Consider the file of Example 1 with $r = 300,000$ fixed-length records of size $R = 100$ bytes stored on a disk with block size $B = 4,096$ bytes.

The file has $b = 7,500$ blocks, as calculated in Example 1.

Suppose we want to search for a record with a specific value for the secondary key—a non-ordering key field of the file that is **V = 9 bytes long**.

Without the secondary index, to do a linear search on the file would require $b/2 = 7,500/2 = 3,750$ block accesses on the average.

Suppose that we construct a secondary index on that non-ordering key field of the file.

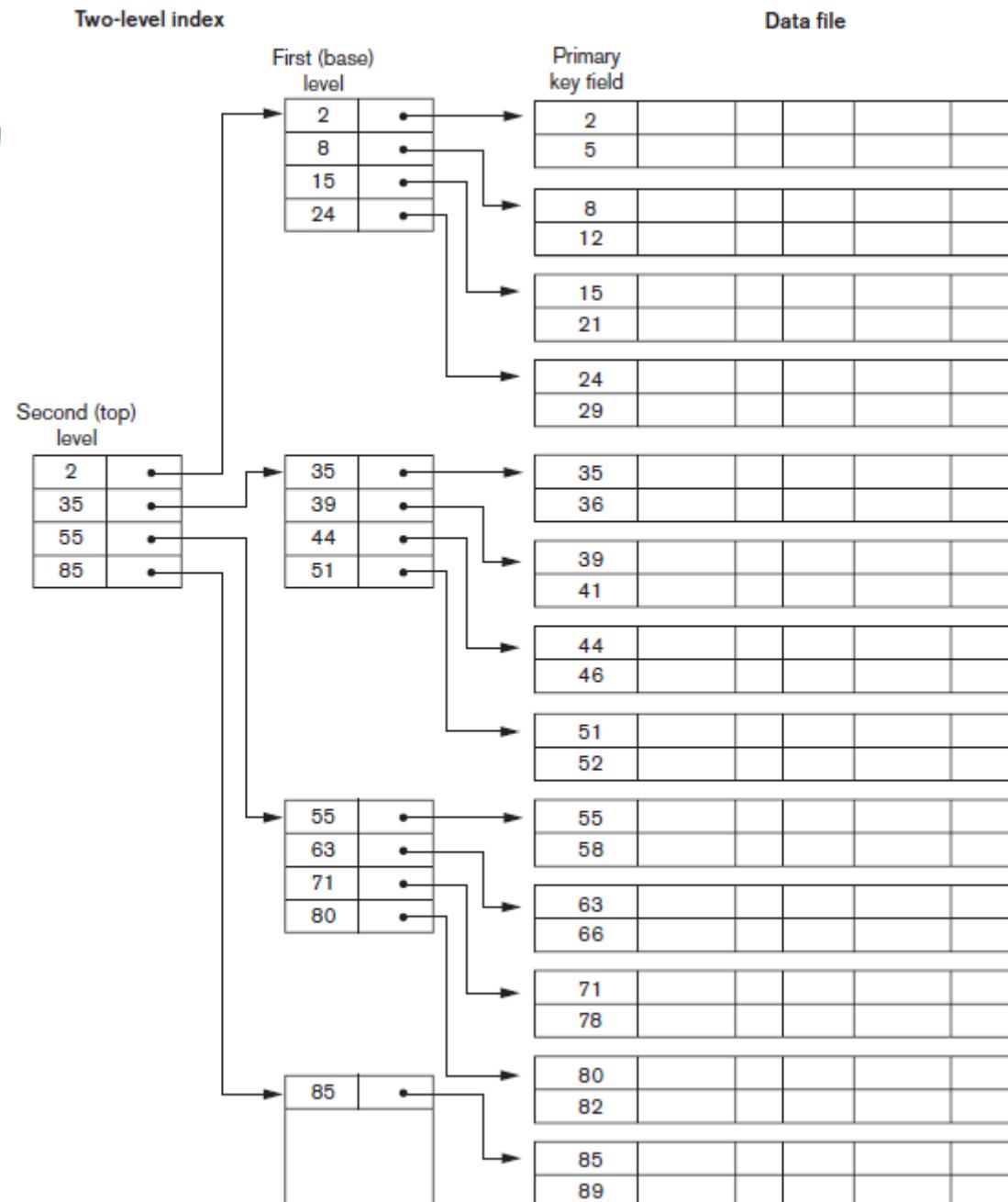
As in Example 1, a block pointer is $P = 6$ bytes long, so each index entry is $R_i = (9 + 6) = 15$ bytes, and the blocking factor for the index is $bfri = \lfloor (B/R_i) \rfloor = \lfloor (4,096/15) \rfloor = 273$ index entries per block.

In a dense secondary index such as this, the total number of index entries ri is equal to the number of records in the data file, which is 300,000.

The number of blocks needed for the index is hence $bi = \lceil (ri/bfri) \rceil = \lceil (300,000/273) \rceil = 1,099$ blocks. (As compared to 13 block accesses on the Primary index on the ordering key field and 2 block accesses using clustering index)

- **Designed to greatly reduce remaining search space as search is conducted**
- **Index file**
 - Considered first (or base level) of a multilevel index
- **Second level**
 - Primary index to the first level
- **Third level**
 - Primary index to the second level

Figure 17.6 A two-level primary index resembling ISAM (indexed sequential access method) organization



Example 4.

Suppose that the dense secondary index of Example 3 is converted into a multilevel index.

We calculated the index blocking factor $bfri = 273$ index entries per block, which is also the fan-out for the multilevel index; the number of first-level blocks $b1 = 1,099$ blocks was also calculated.

The number of second-level blocks will be $b2 = \lceil (b1/fo) \rceil = \lceil (1,099/273) \rceil = 5$ blocks, and the number of third level blocks will be $b3 = \lceil (b2/fo) \rceil = \lceil (5/273) \rceil = 1$ block.

Hence, the third level is the top level of the index, and $t = 3$.

To access a record by searching the multilevel index, we must access one block at each level plus one block from the data file, so we need $t + 1 = 3 + 1 = 4$ block accesses.

Compare this to Example 3, where 12 block accesses were needed when a single-level index and binary search were used.

B-Tree and B⁺-Tree

A B-tree is an M-way search tree with two properties :

1. It is perfectly balanced: every leaf node is at the same depth
2. Every internal node other than the root, is at least half-full, i.e. $M/2-1 \leq \#keys \leq M-1$
3. Every internal node with k keys has $k+1$ non-null children

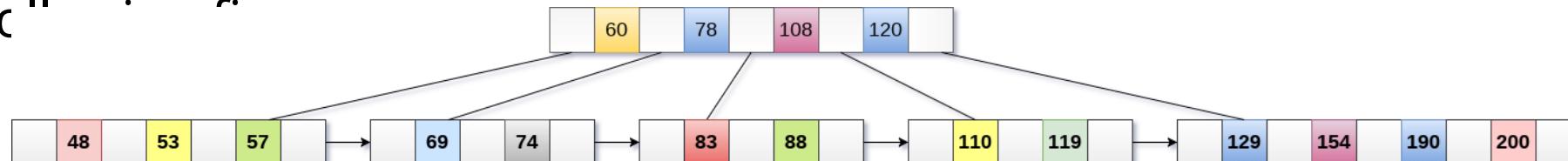
- Same structure as B-trees.
- Dictionary pairs are in leaves only.
- Leaves form a doubly-linked list.
- Remaining nodes have following structure:

$$j \ a_0 \ k_1 \ a_1 \ k_2 \ a_2 \dots k_j \ a_j$$

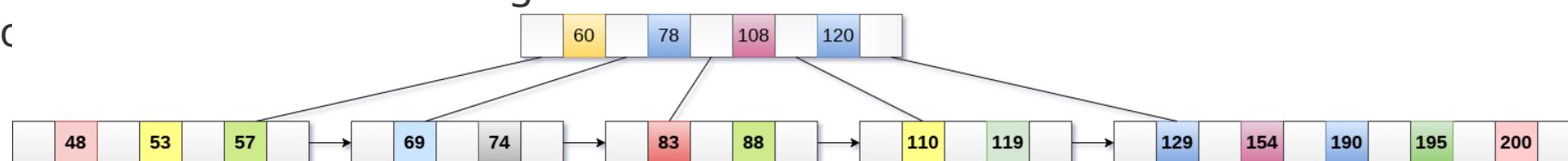
- j = number of keys in node.
- a_i is a pointer to a subtree.
- $k_i \leq$ smallest key in subtree a_i and $>$ largest in a_{i-1} .

- B+ Tree is an extension of B Tree which allows efficient insertion, deletion and search operations.
- In B Tree, Keys and records both can be stored in the internal as well as leaf nodes.
- Whereas, in B+ tree, records (data) can only be stored on the leaf nodes while internal nodes can only store the key values.
- The leaf nodes of a B+ tree are linked together in the form of a singly linked lists to make the search queries more efficient.
- B+ Tree are used to store the large amount of data which can not be stored in the main memory.
- Due to the fact that, size of main memory is always limited, the internal nodes (keys to access records) of the B+ tree are stored in the main memory whereas, leaf nodes are stored in the secondary memory.

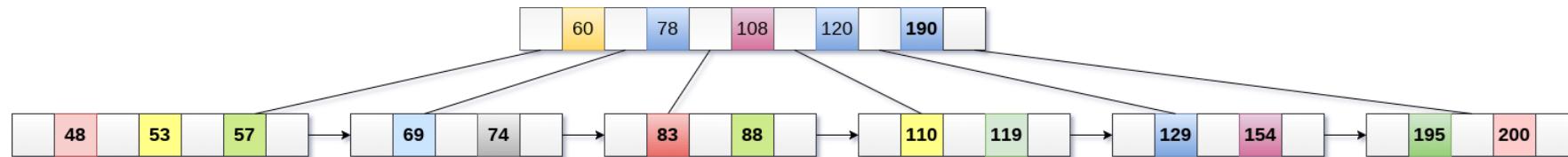
- Insertion in B+ Tree
- **Step 1:** Insert the new node as a leaf node
- **Step 2:** If the leaf doesn't have required space, split the node and copy the middle node to the next index node.
- **Step 3:** If the index node doesn't have required space, split the node and copy the middle element to the next index page.
- Example :
- Insert the value 195 into the B+ tree of order 5 shown in the figure



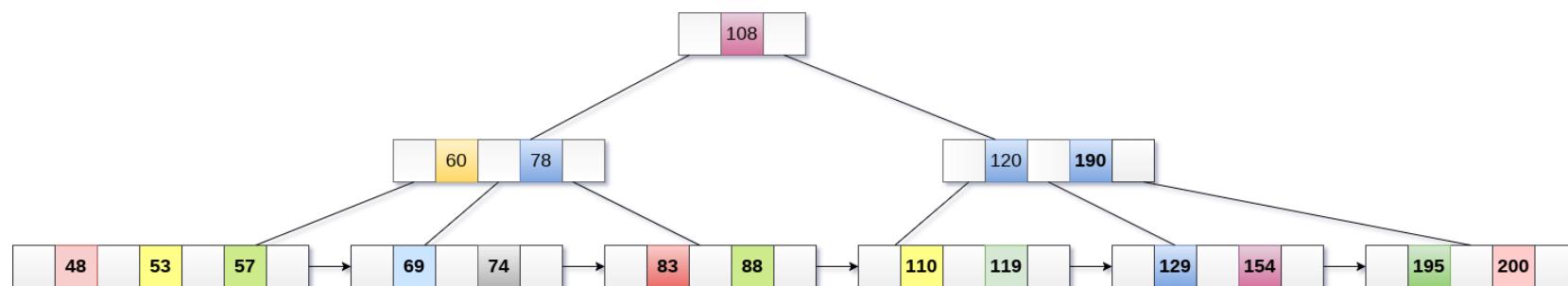
195 will be inserted in the right sub-tree of 120 after 190. Insert it at the desired position



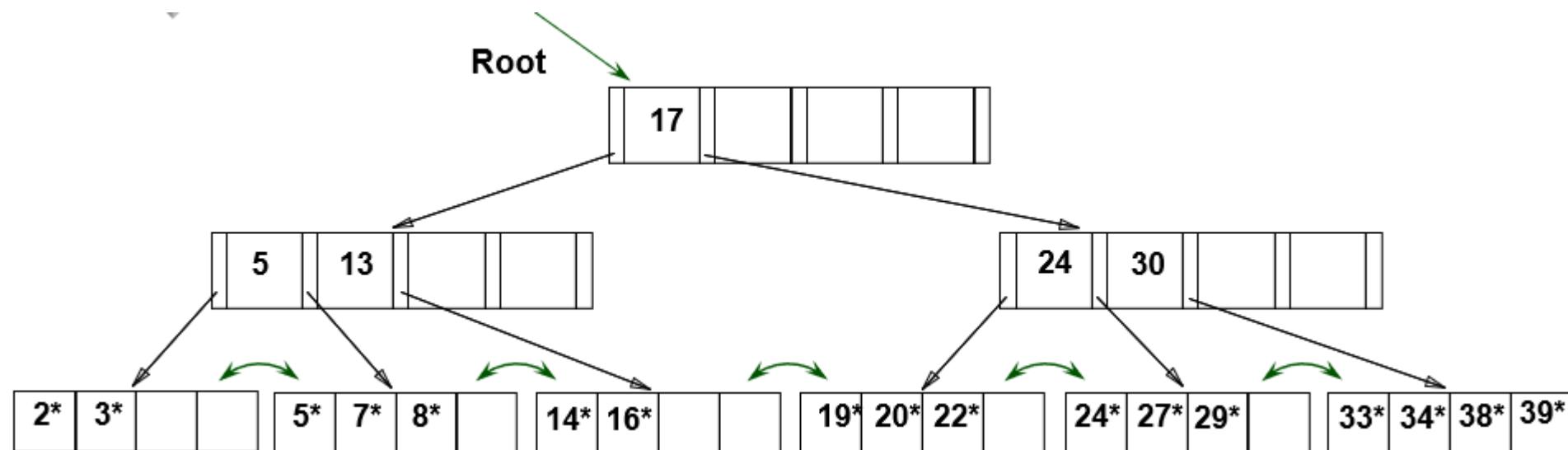
- The node contains greater than the maximum number of elements i.e. 4, therefore split it and place the median node up to the parent.



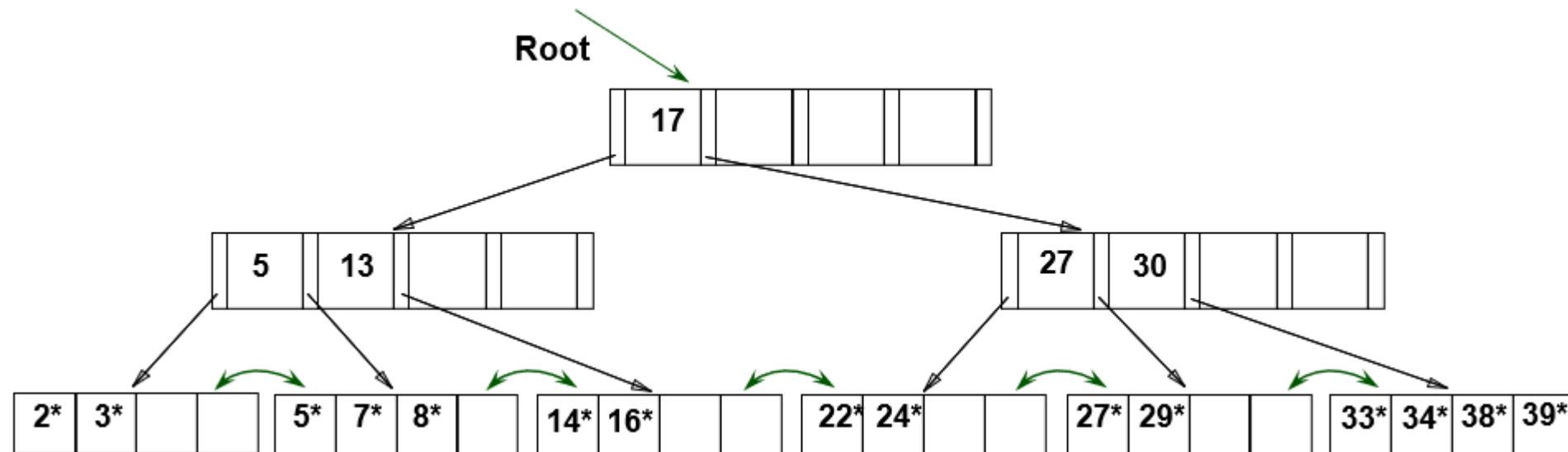
Now, the index node contains 6 children and 5 keys which violates the B+ tree properties, therefore we need to split it, shown as follows.



Example 2- B⁺ Tree After Inserting 8*



Example Tree After (Inserting 8*, Then) Deleting 19* and 20* ...



➤ Tree data structure terminology

- Tree is formed of nodes
- Each node (except root) has one parent and zero or more child nodes
- Leaf node has no child nodes
 - **Unbalanced if leaf nodes occur at different levels**
- Non-leaf node called internal node
- Sub-tree of node consists of node and all descendant nodes

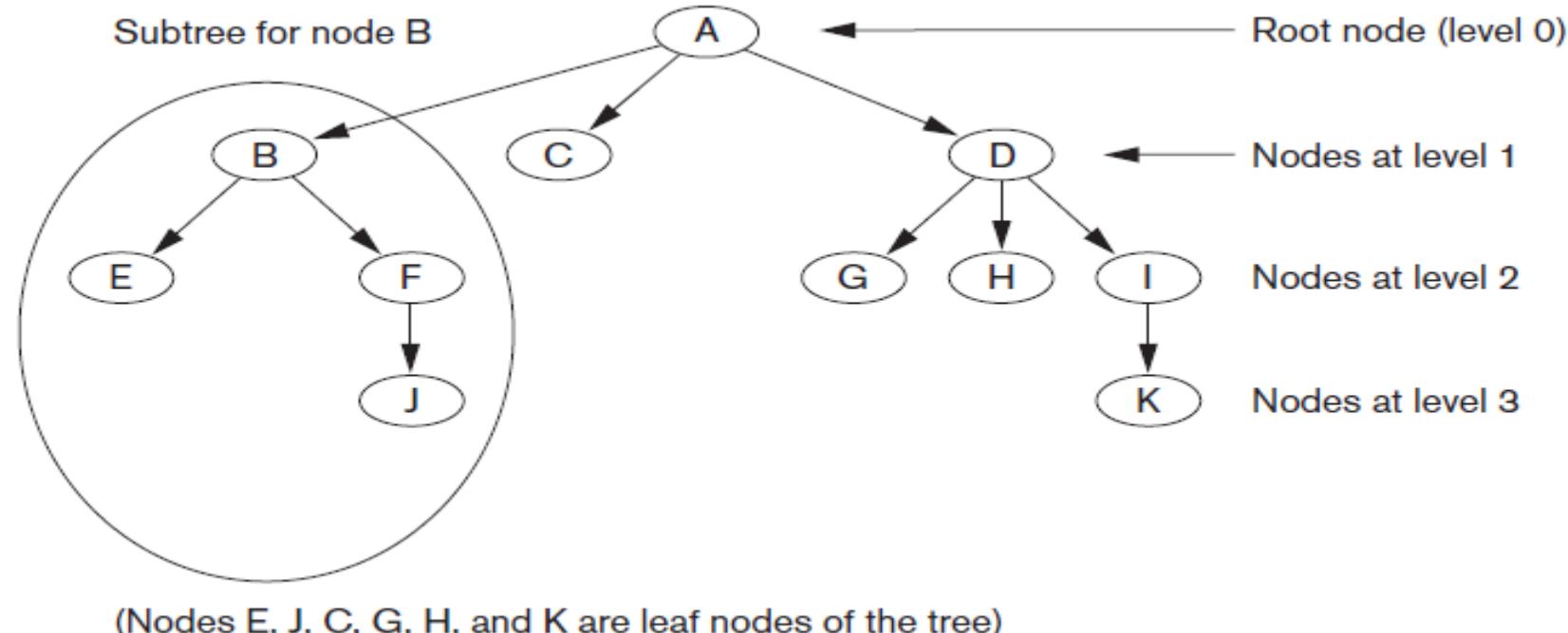


Figure 17.7 A tree data structure that shows an unbalanced tree

- Search tree used to guide search for a record
 - Given value X of one of record's fields

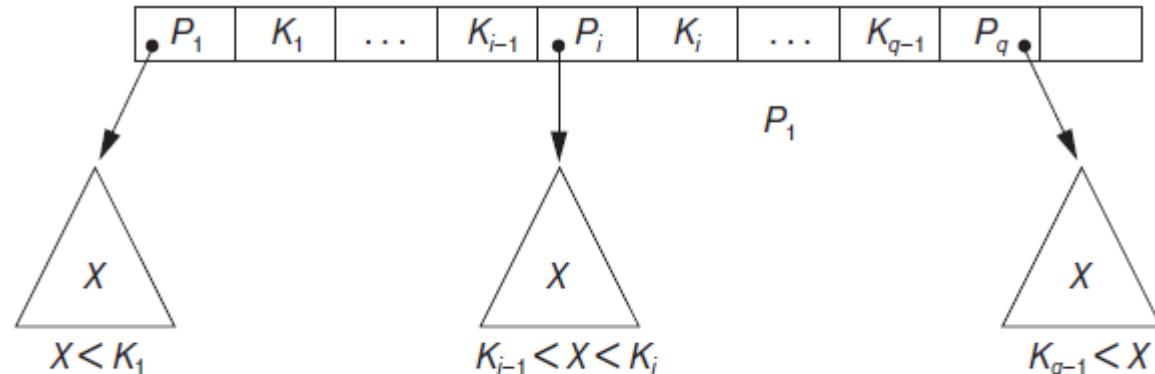


Figure 17.8 A node in a search tree with pointers to subtrees below it

- Algorithms necessary for inserting and deleting search values into and from the tree

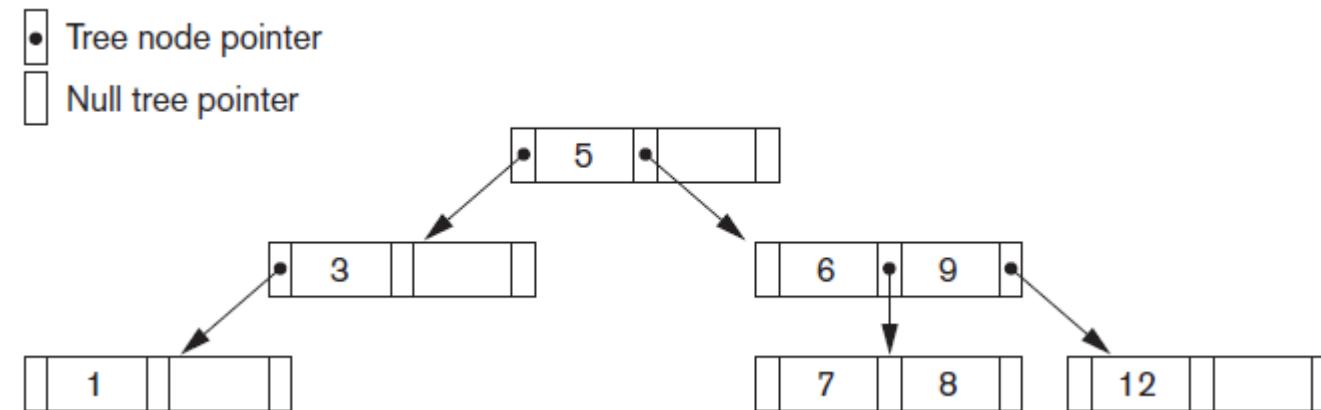


Figure 17.9 A search tree of order $p = 3$

- Provide multi-level access structure
- Tree is always balanced
- Space wasted by deletion never becomes excessive
 - Each node is at least half-full
- Each node in a B-tree of order p can have at most $p-1$ search values

B-Tree Structures

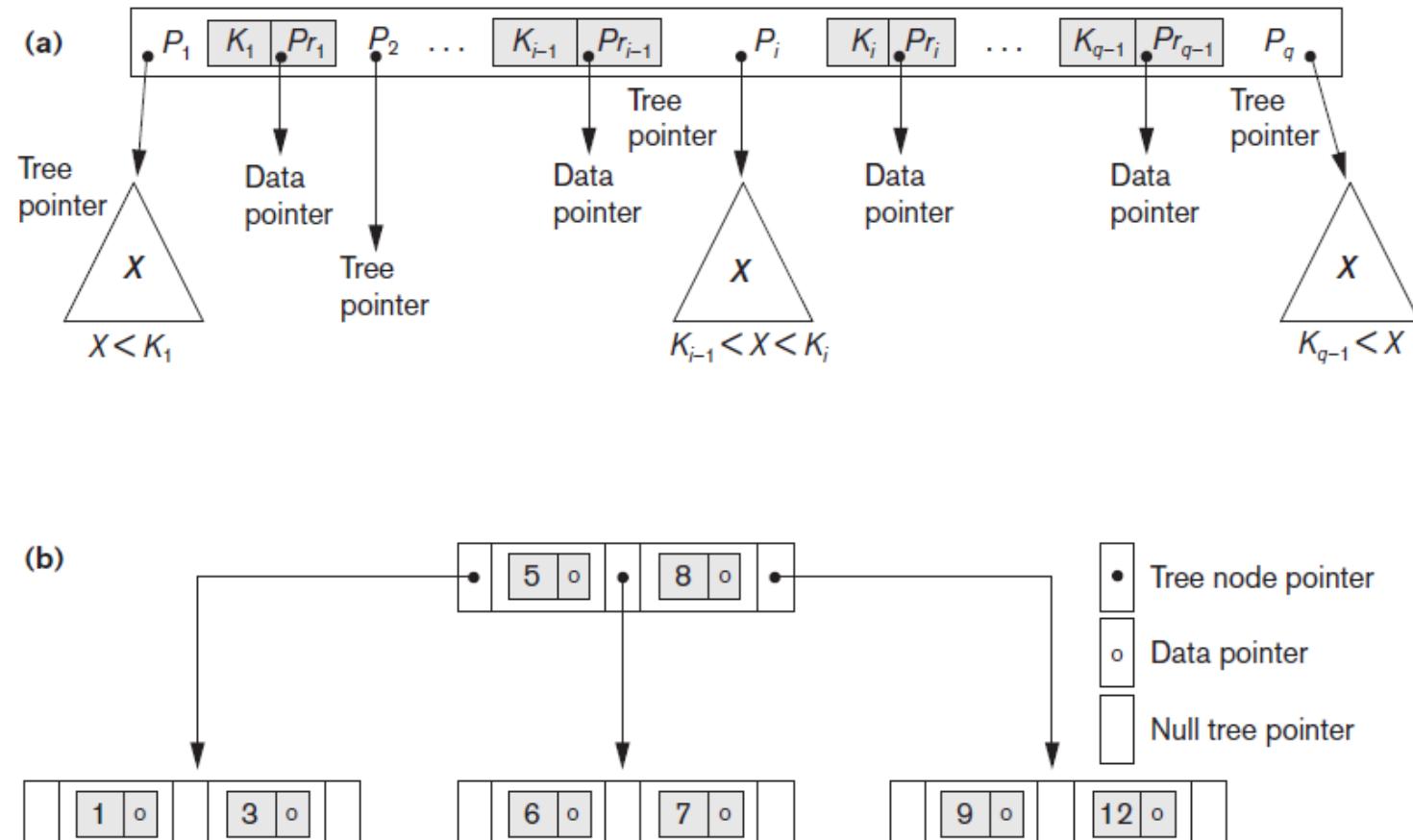


Figure 17.10 B-tree structures
(a) A node in a B-tree with $q-1$ search values
(b) A B-tree of order $p=3$. The values were inserted in the order 8, 5, 1, 7, 3, 12, 9, 6

➤ Data pointers stored only at the leaf nodes

- Leaf nodes have an entry for every value of the search field, and a data pointer to the record if search field is a key field
- For a non-key search field, the pointer points to a block containing pointers to the data file records

➤ Internal nodes

- Some search field values from the leaf nodes repeated to guide search

B⁺ -Trees (cont'd.)

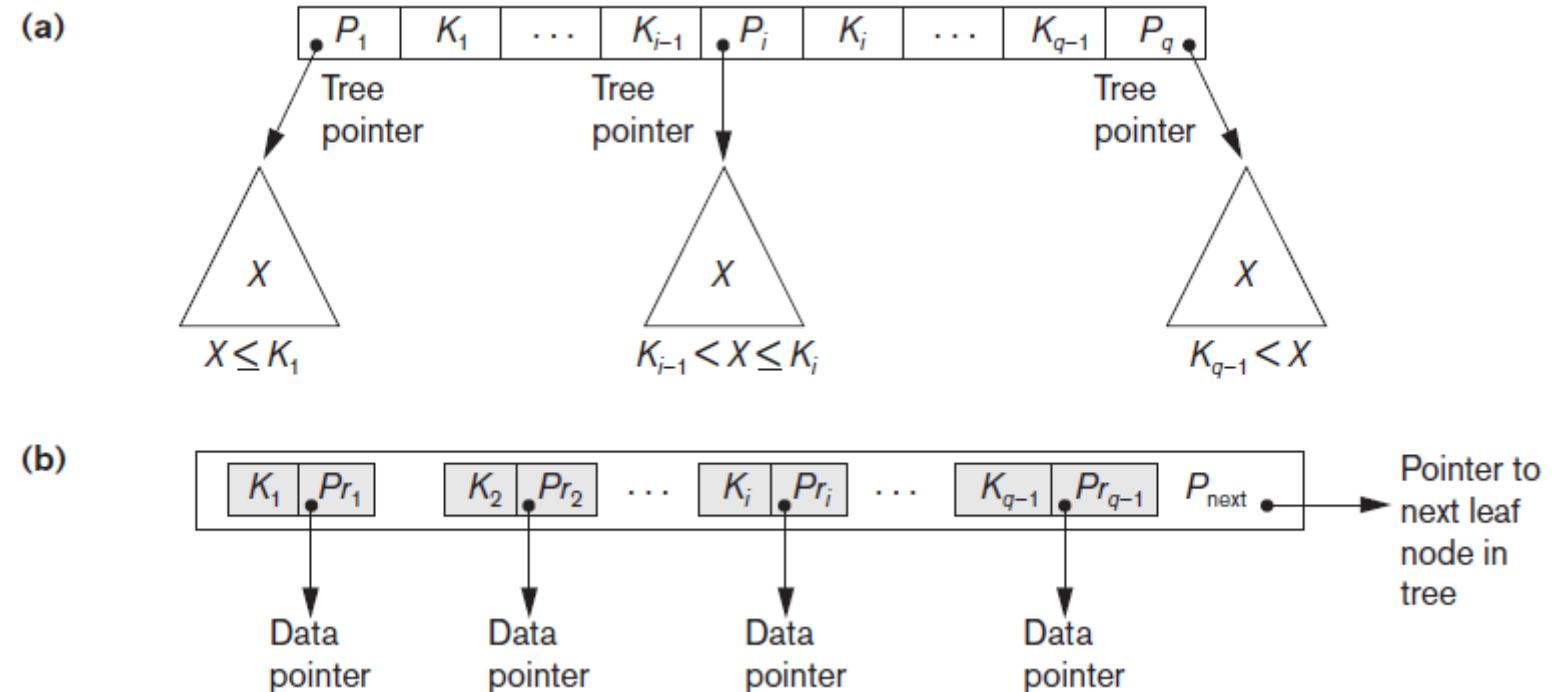


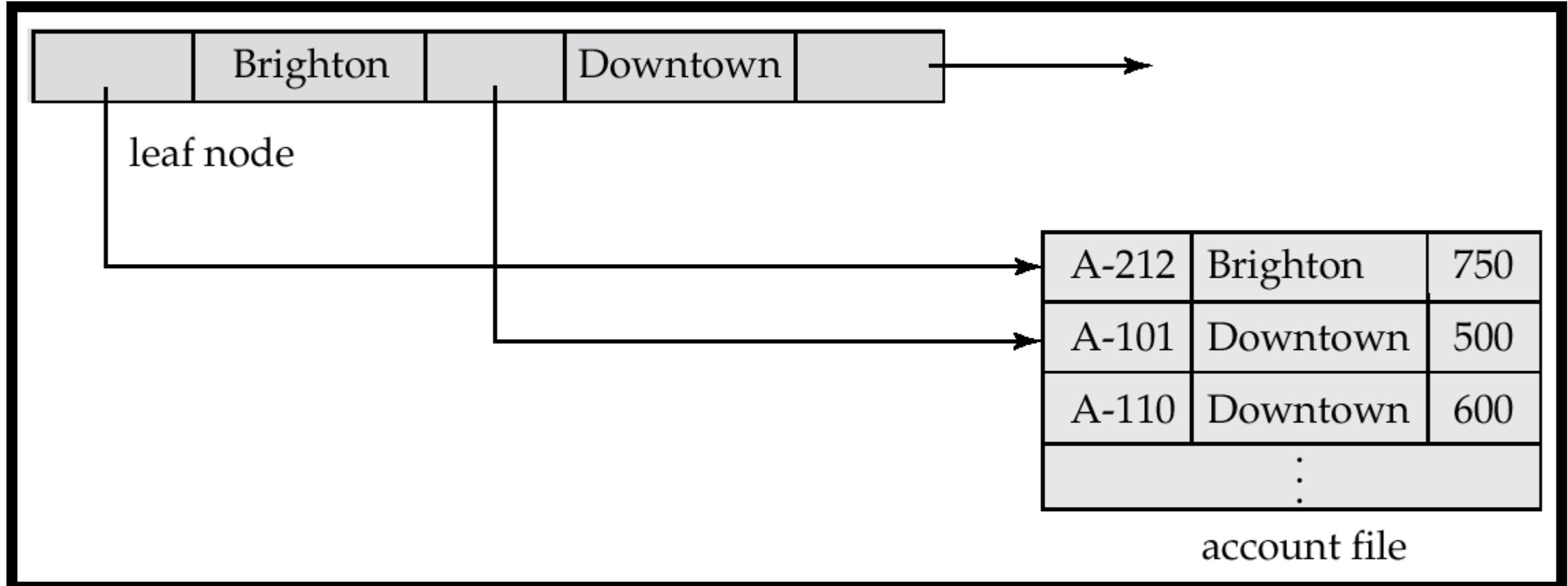
Figure 17.11 The nodes of a B+-tree (a) Internal node of a B+-tree with $q-1$ search values (b) Leaf node of a B+-tree with $q-1$ search values and $q-1$ data pointers

Field Value K, Using a B⁺ -Tree

```
n ← block containing root node of B+-tree;  
read block n;  
while (n is not a leaf node of the B+-tree) do  
    begin  
        q ← number of tree pointers in node n;  
        if K ≤ n.K1 (*n.Ki refers to the ith search field value in node n*)  
            then n ← n.P1 (*n.Pi refers to the ith tree pointer in node n*)  
        else if K > n.Kq-1  
            then n ← n.Pq  
        else begin  
            search node n for an entry i such that n.Ki-1 < K ≤ n.Ki;  
            n ← n.Pi;  
            end;  
        read block n  
    end;  
    search block n for entry (Ki, Pri) with K = Ki; (* search leaf node *)  
    if found  
        then read data file block with address Pri and retrieve record  
    else the record with search field value K is not in the data file;
```

Algorithm 17.2 Searching for a record with search key field value K, using a B+ -Tree

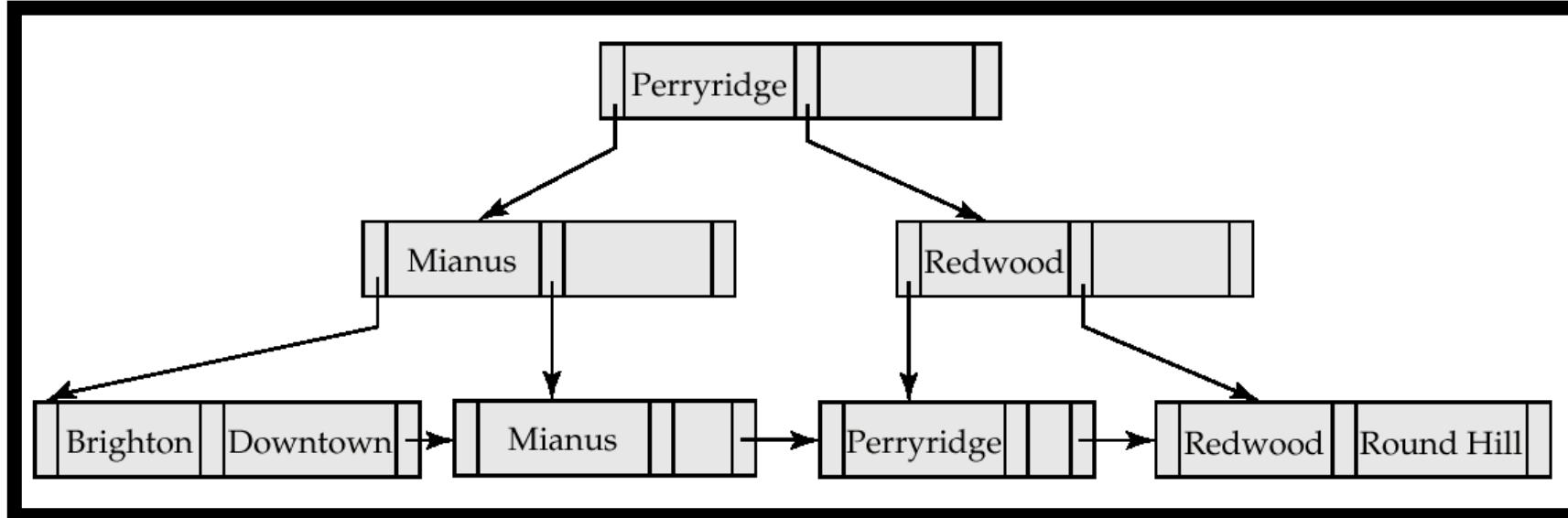
Leaf Nodes in B⁺-Trees



- All the search-keys in the subtree to which P_1 points are less than K_1
- For $2 \leq i \leq n - 1$, all the search-keys in the subtree to which P_i points have values greater than or equal to K_{i-1} and less than K_{m-1}

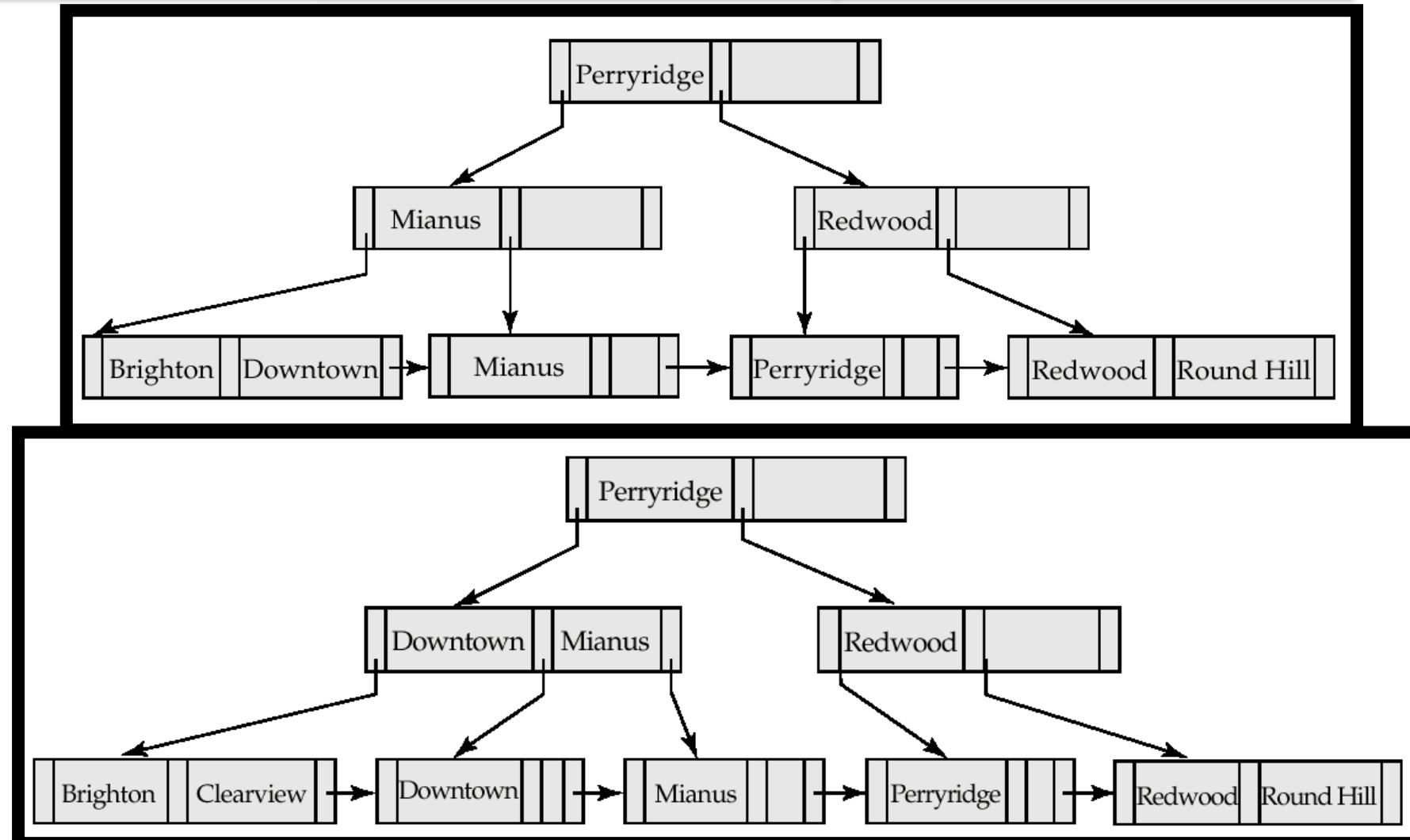


Example of a B⁺-tree



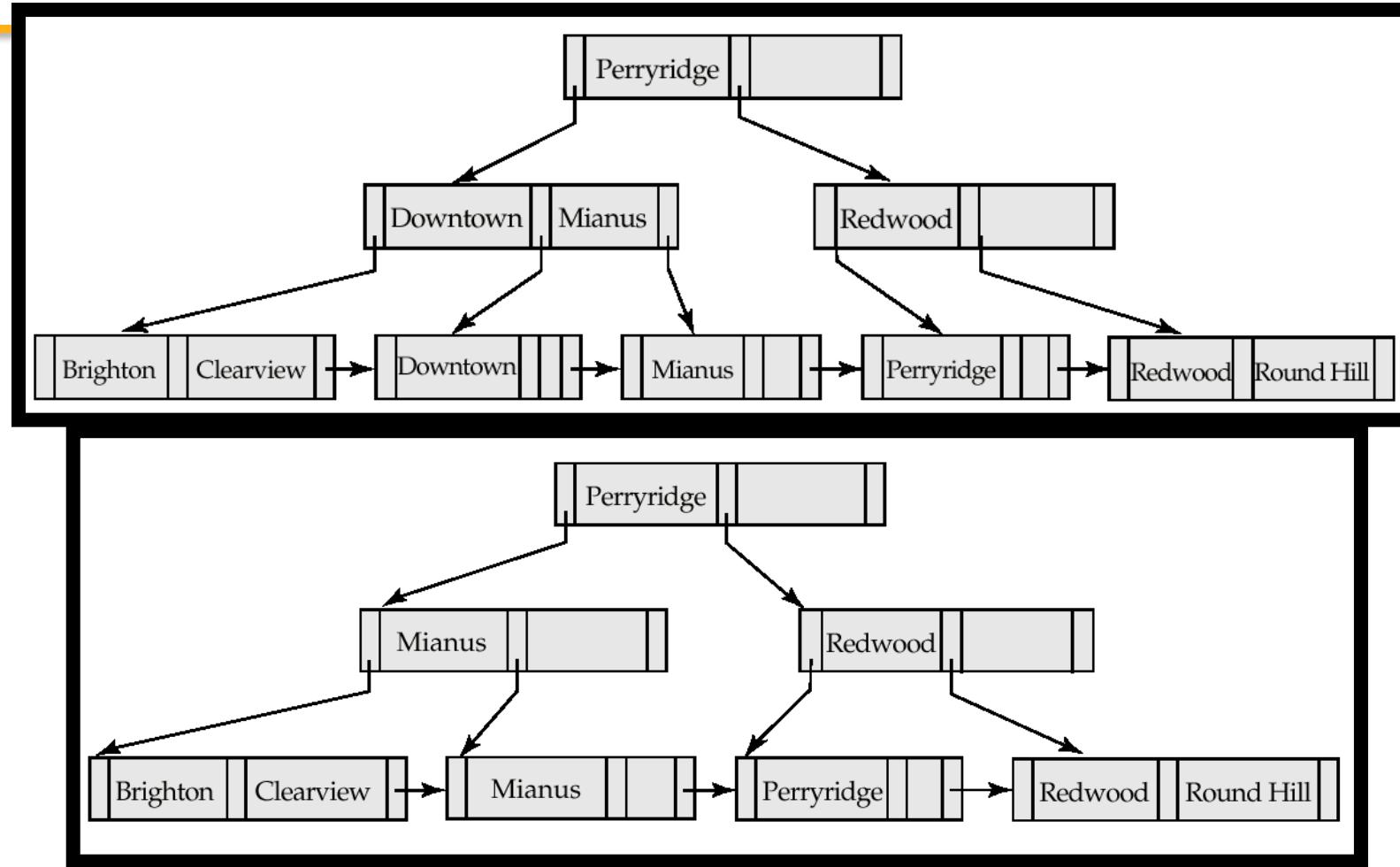
B⁺-tree for account file (n = 3)

Updates on B⁺-Trees: Insertion



B⁺-Tree before and after insertion of “Clearview”

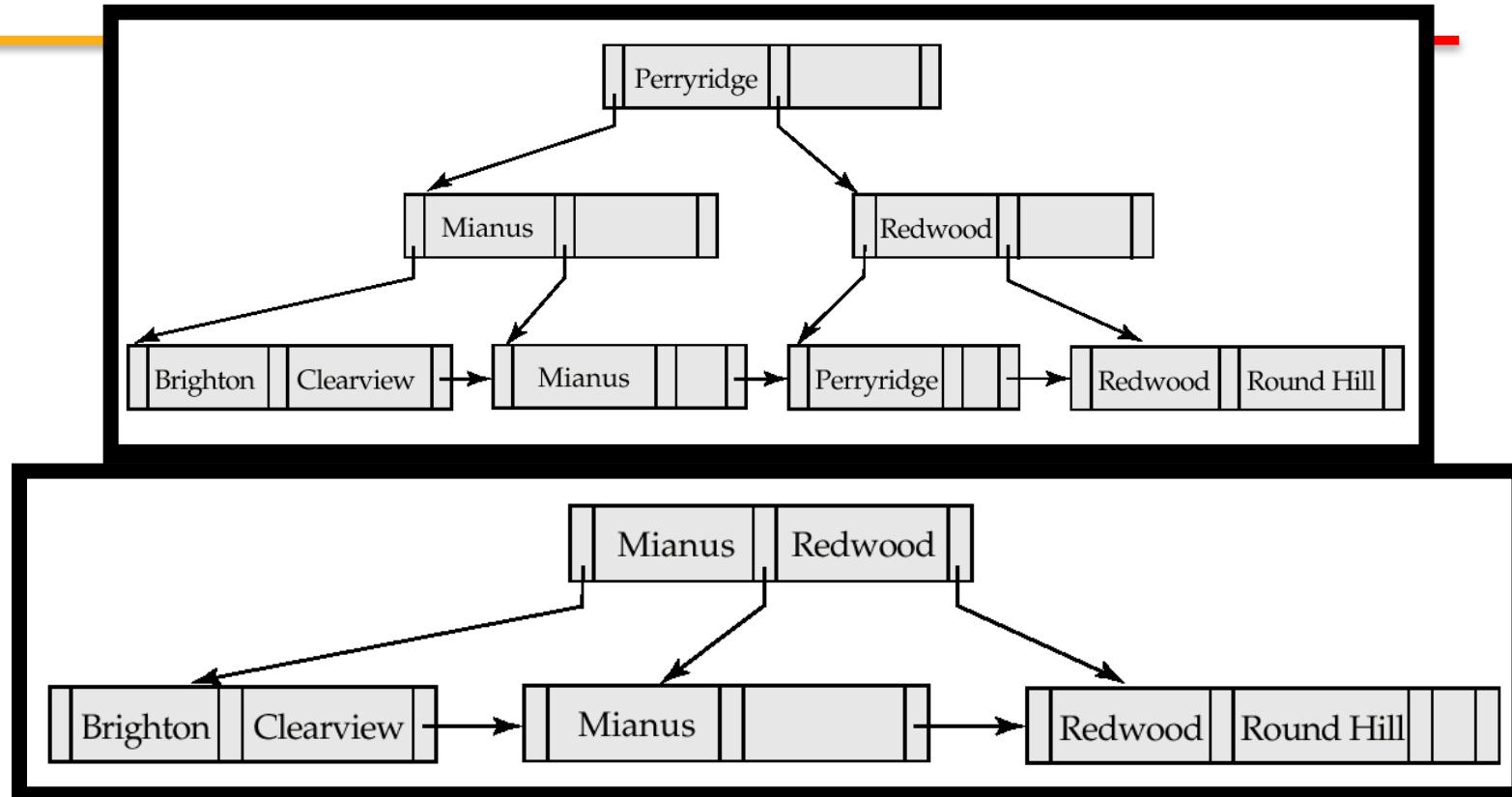
Examples of B⁺-Tree Deletion



Before and after deleting “Downtown”

- The removal of the leaf node containing “Downtown” did not result in its parent having too little pointers. So the cascaded deletions stopped with the deleted leaf node’s parent.

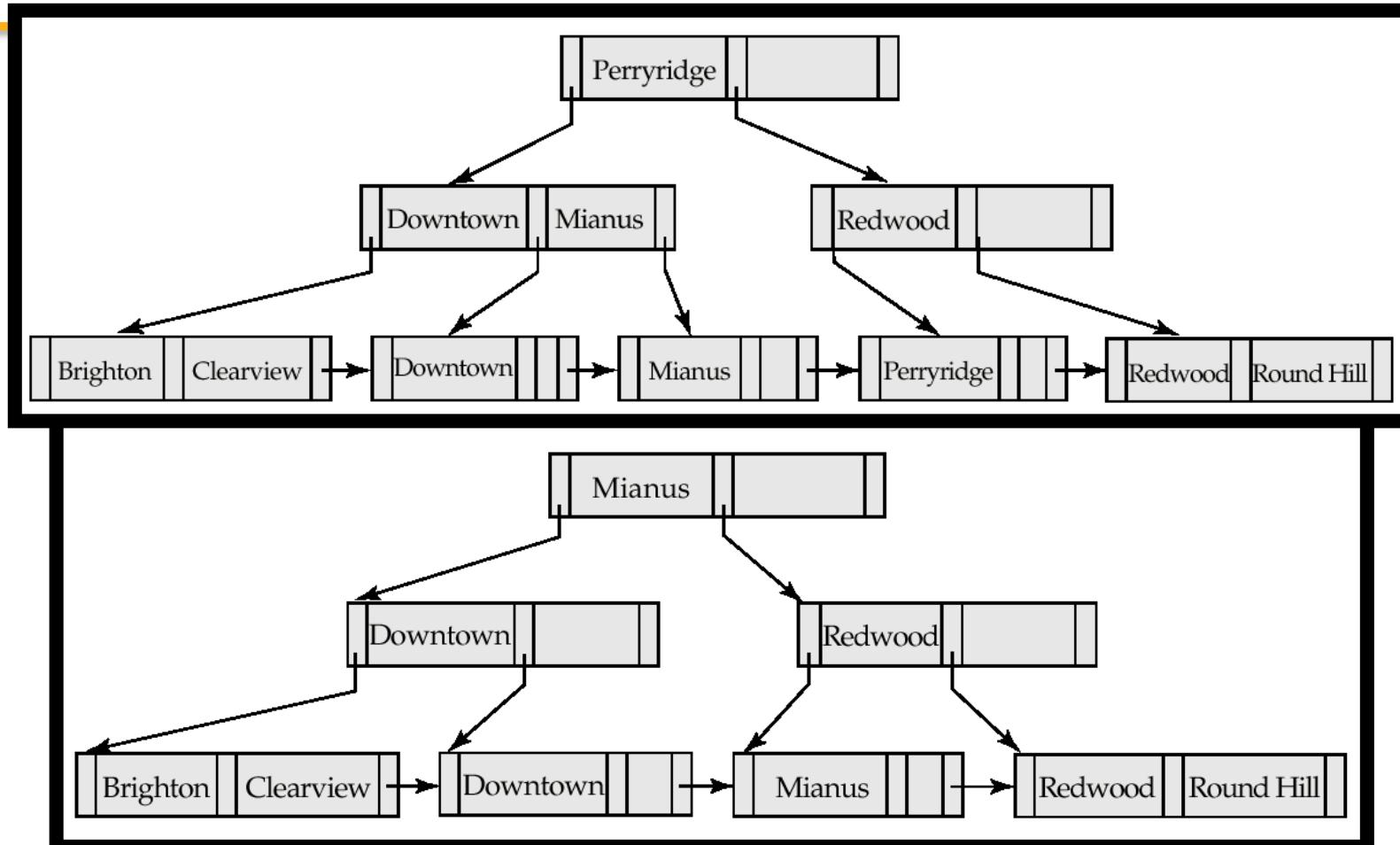
Examples of B⁺-Tree Deletion



Deletion of “Perryridge” from result of previous example

- Node with “Perryridge” becomes underfull (actually empty, in this special case) and merged with its sibling.
- As a result “Perryridge” node’s parent became underfull, and was merged with its sibling (and an entry was deleted from their parent)
- Root node then had only one child, and was deleted and its child became the new root node

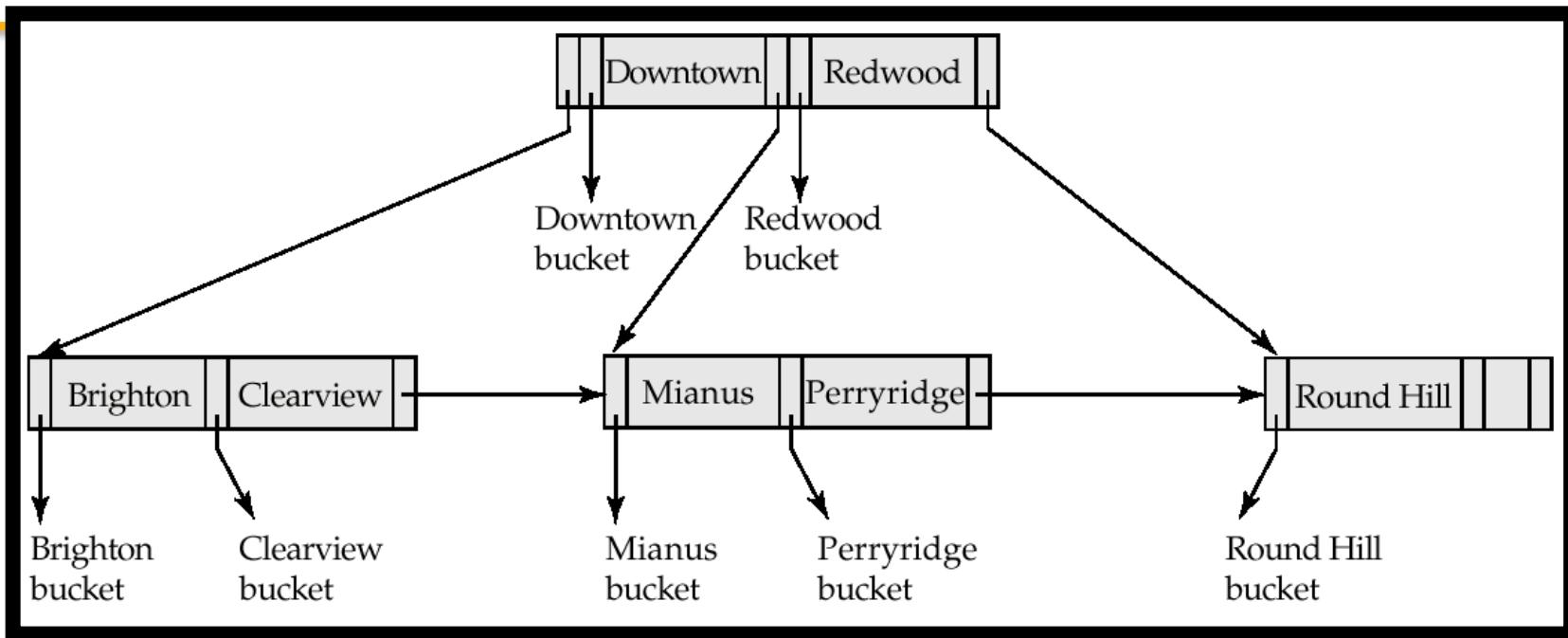
Example of B⁺-tree Deletion



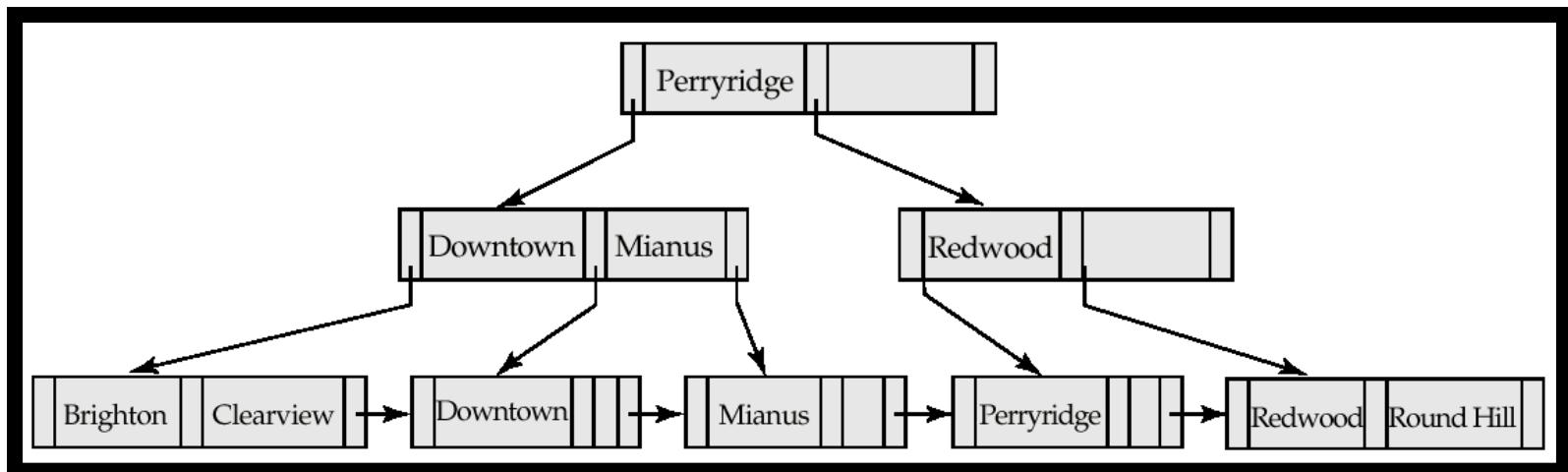
Before and after deletion of "Perryridge" from earlier

- Parent of leaf containing Perryridge became underfull, and borrowed a pointer from its left sibling
- Search-key value in the parent's parent changes as a result

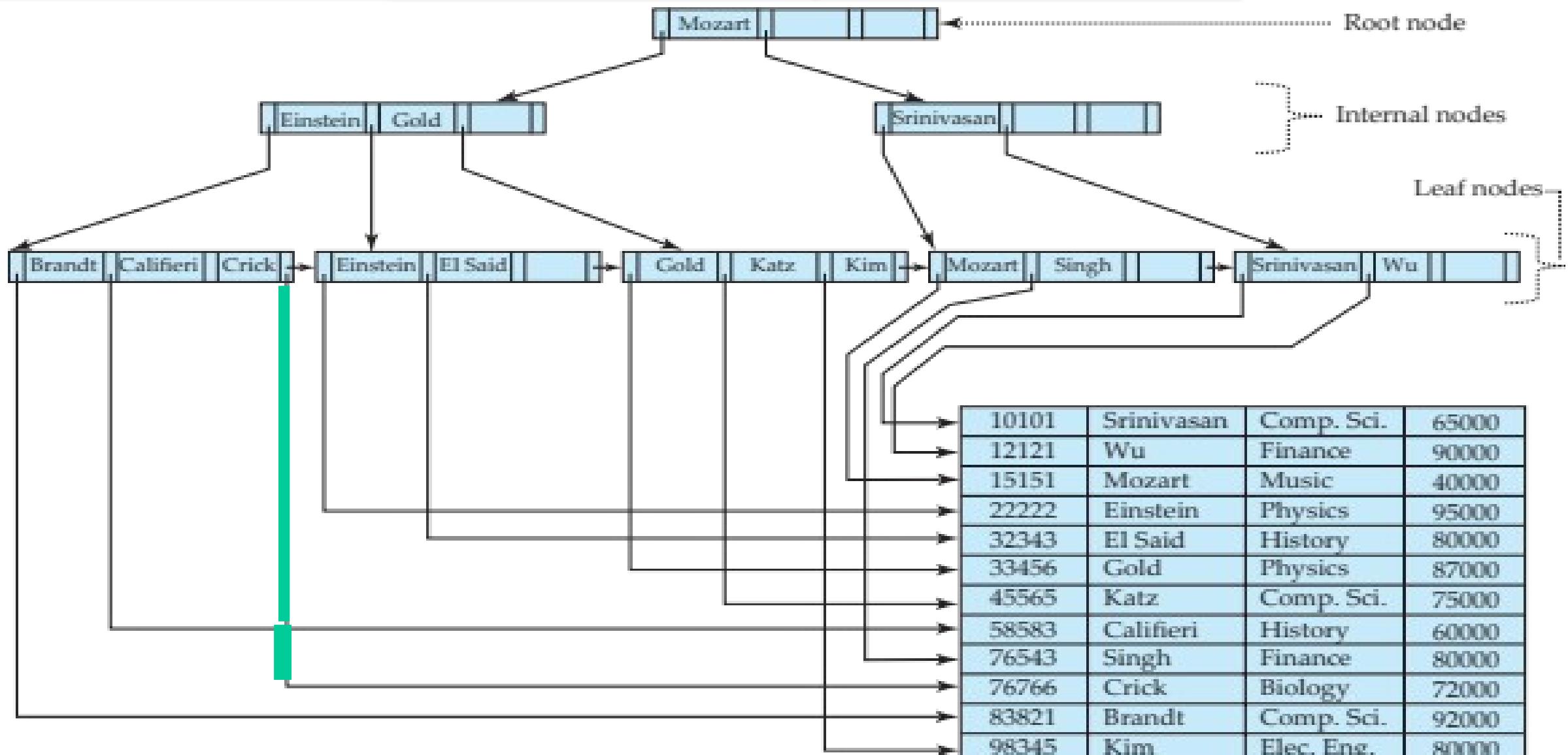
B-Tree Index File Example



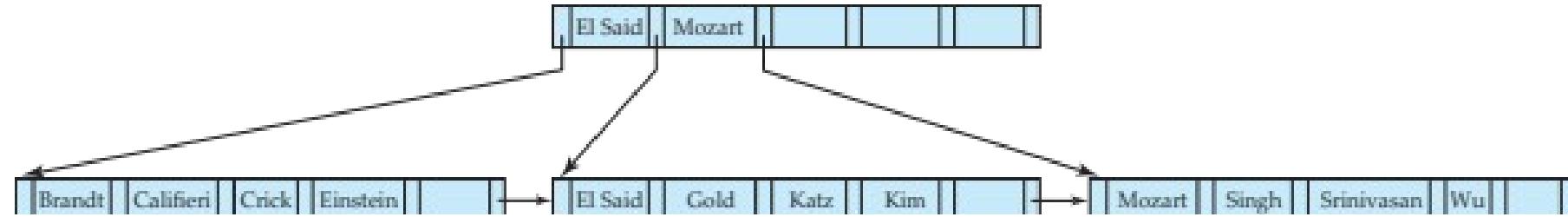
B-tree (above) and B+-tree (below) on same data



Example of B⁺-Tree

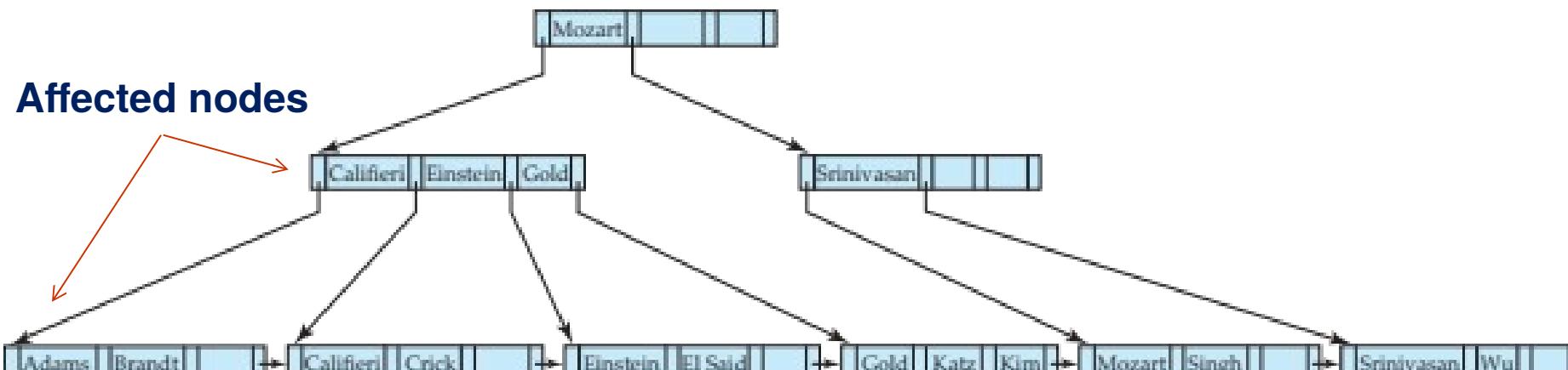
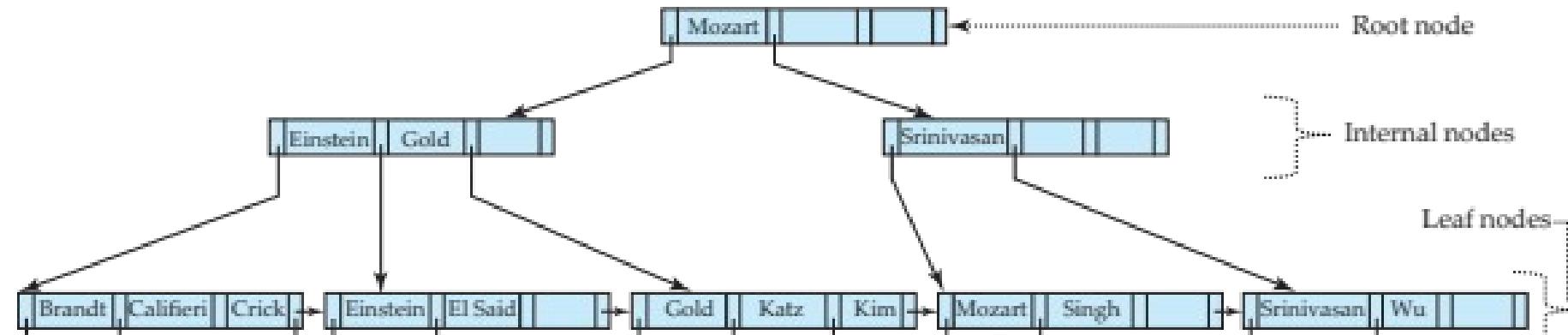


Example of B⁺-tree



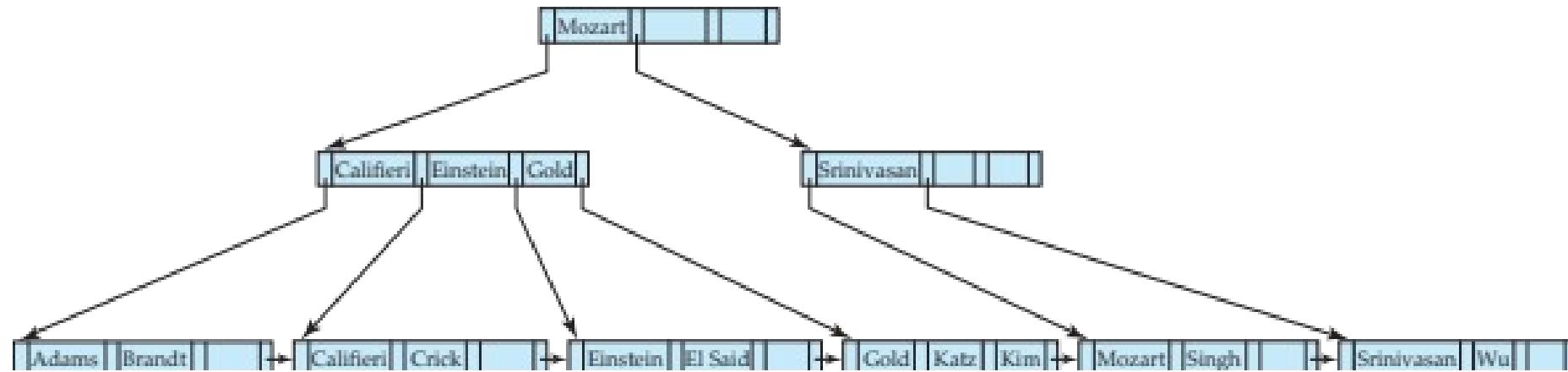
B⁺-tree for *instructor* file ($n = 6$)

B⁺-Tree Insertion

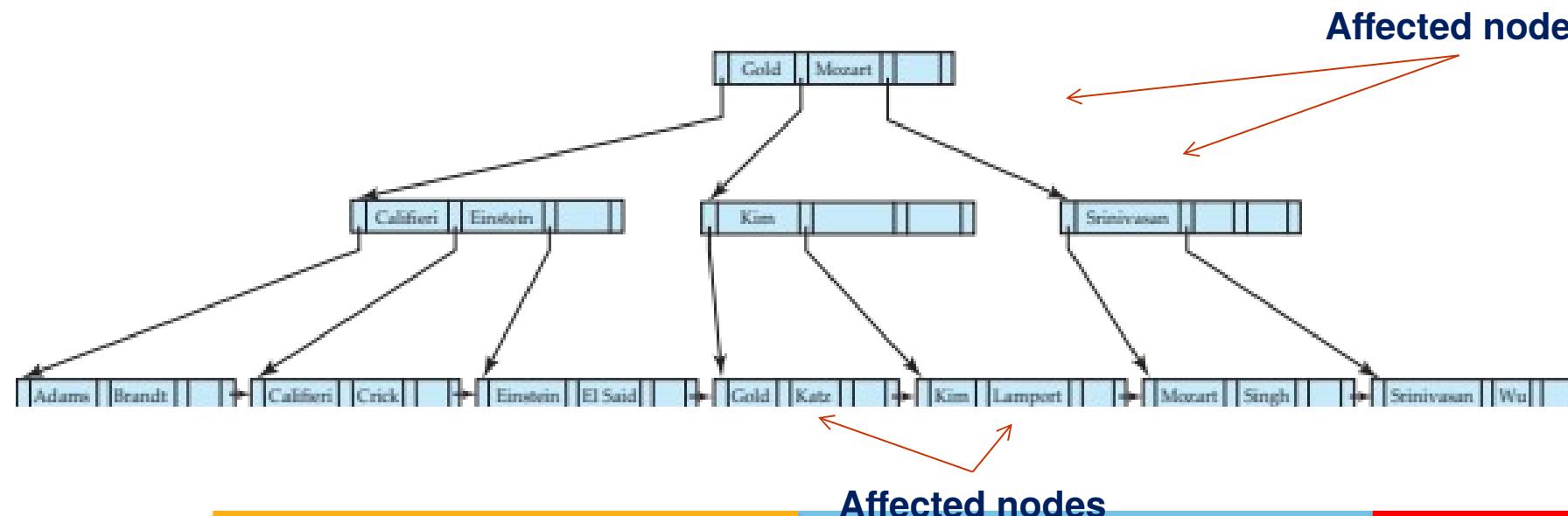


B⁺-Tree before and after insertion of "Adams"

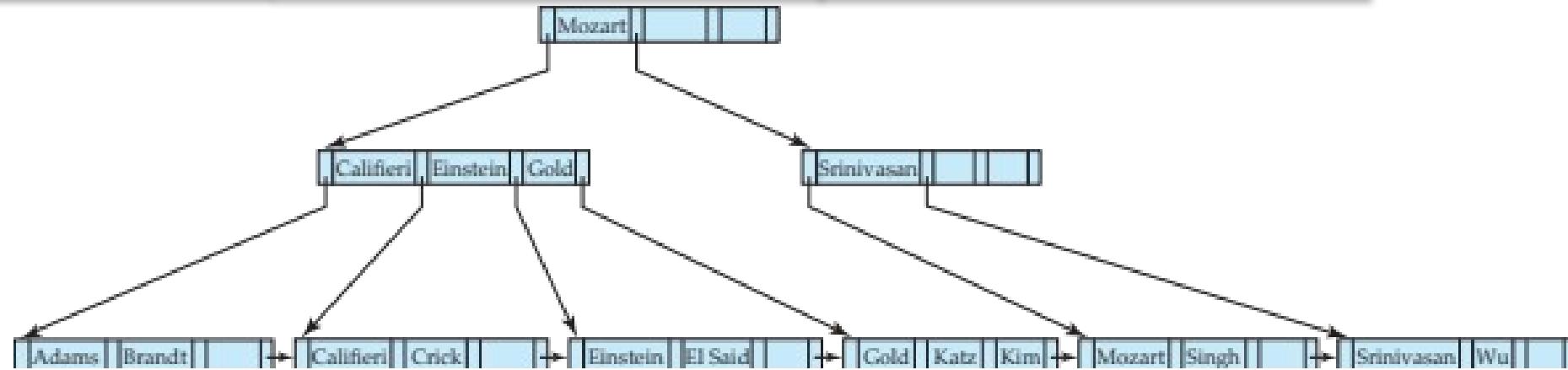
B⁺-Tree Insertion



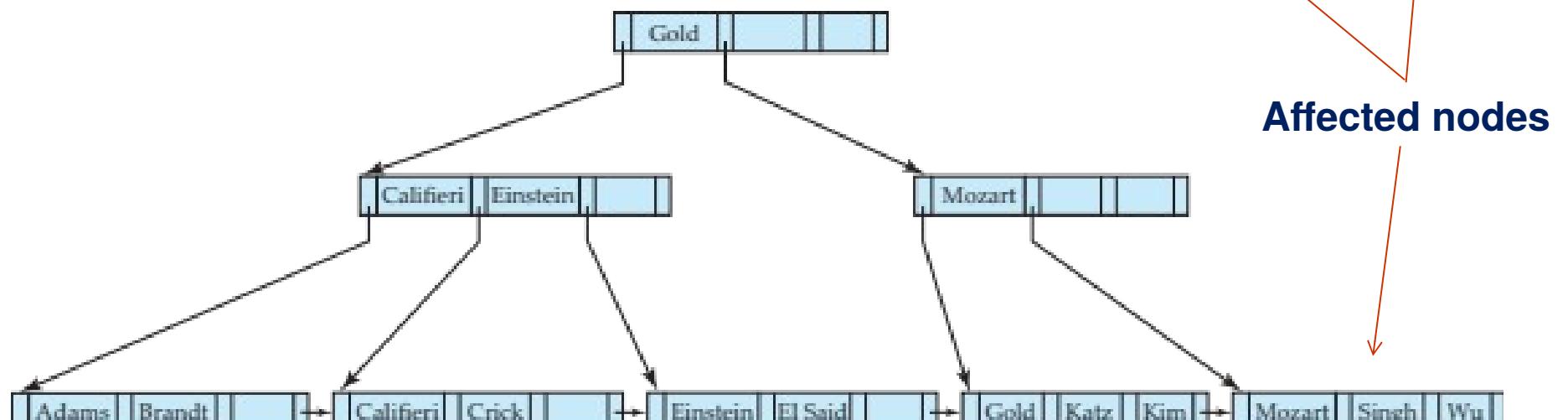
B⁺-Tree before and after insertion of “Lamport”



Examples of B⁺-Tree Deletion



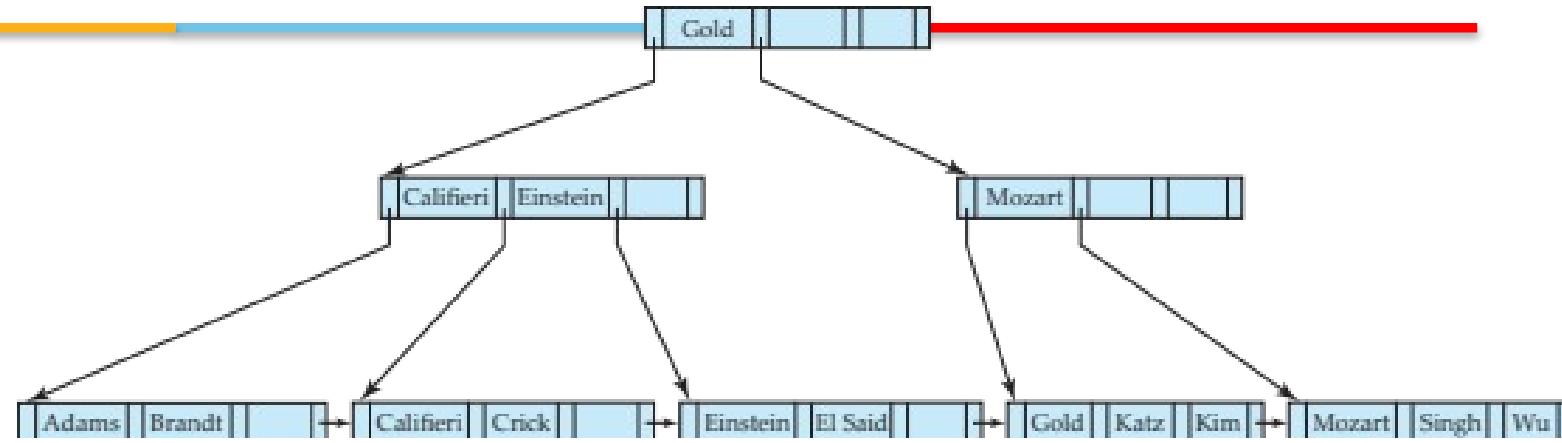
Before and after deleting “Srinivasan”



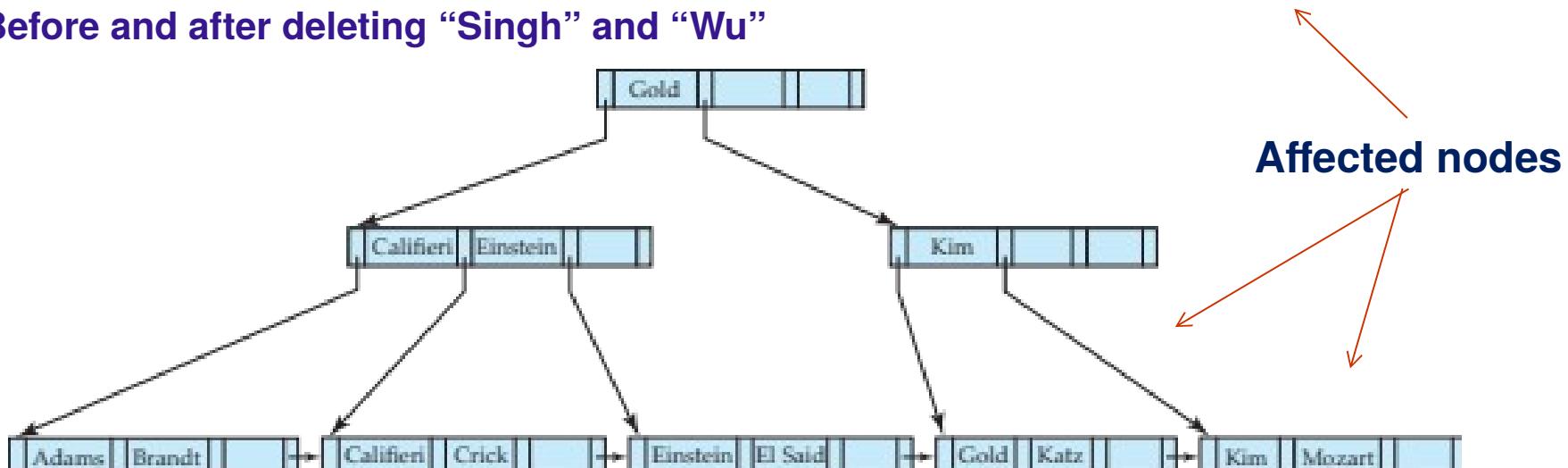
Affected nodes

- Deleting “Srinivasan” causes **merging** of under-full leaves

Examples of B⁺-Tree Deletion

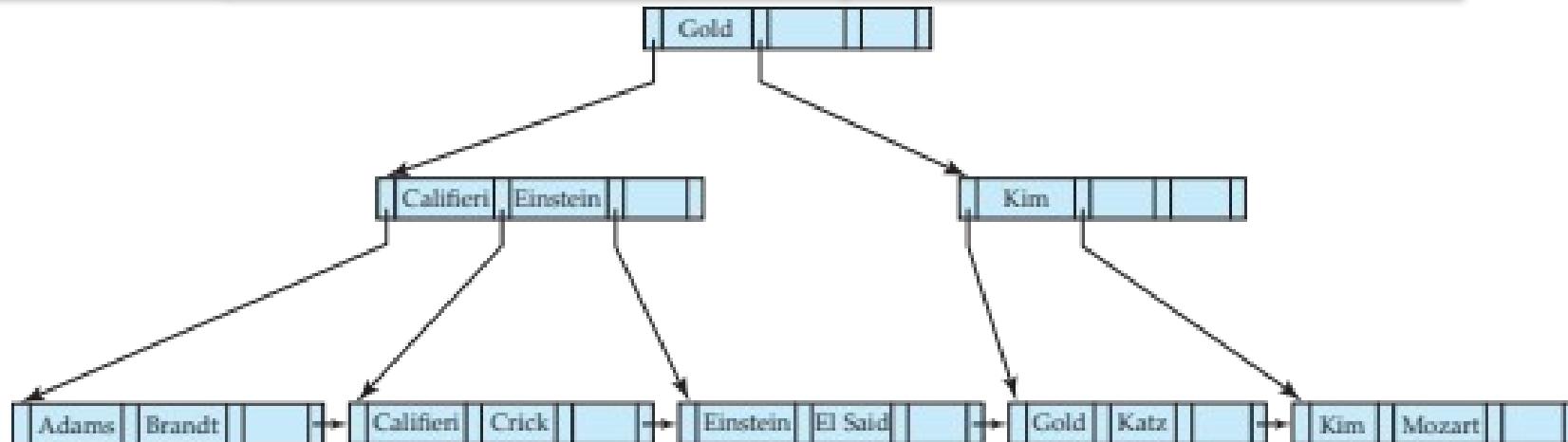


Before and after deleting “Singh” and “Wu”

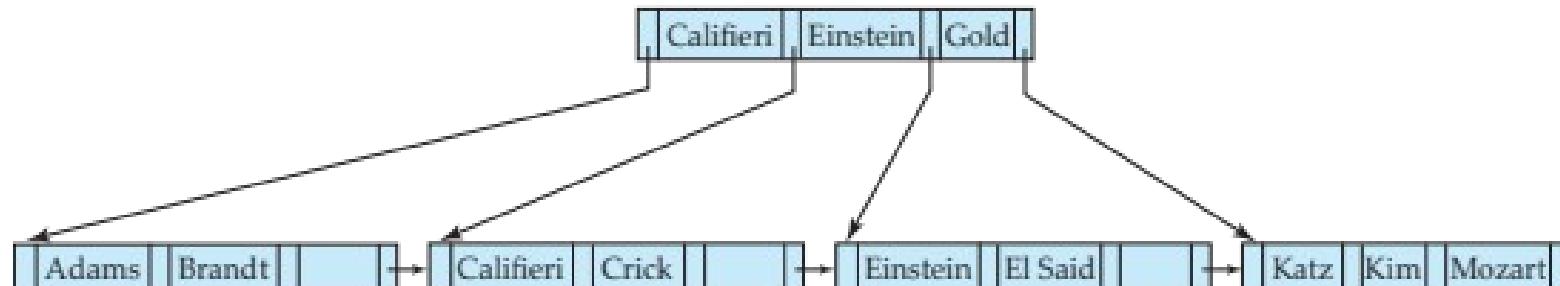


- Leaf containing Singh and Wu became underfull, and **borrowed a value** Kim from its left sibling
- Search-key value in the parent changes as a result

Example of B⁺-tree Deletion (Cont.)

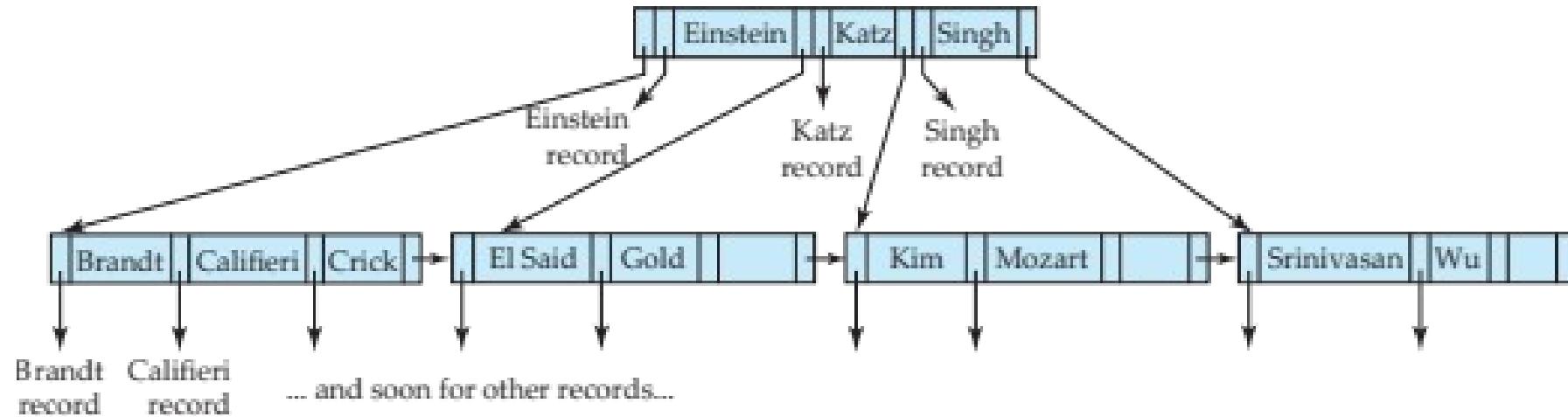


Before and after deletion of "Gold"

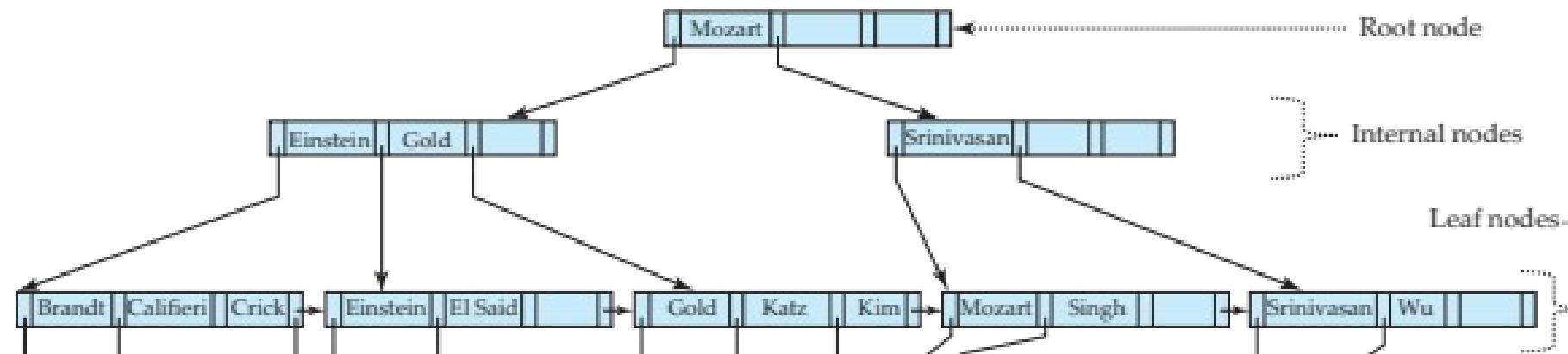


- Node with Gold and Katz became underfull, and was merged with its sibling
- Parent node becomes underfull, and is merged with its sibling
 - Value separating two nodes (at the parent) is pulled down when merging
- Root node then has only one child, and is deleted

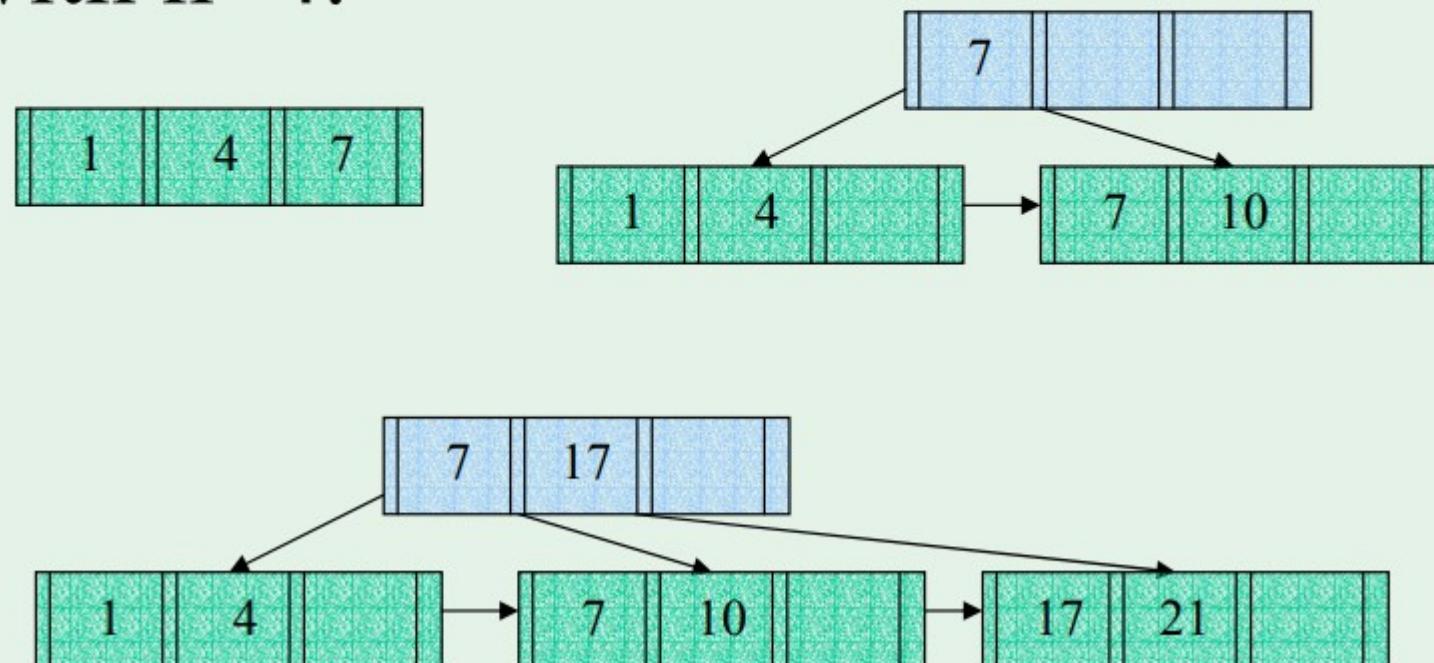
B-Tree Index File Example



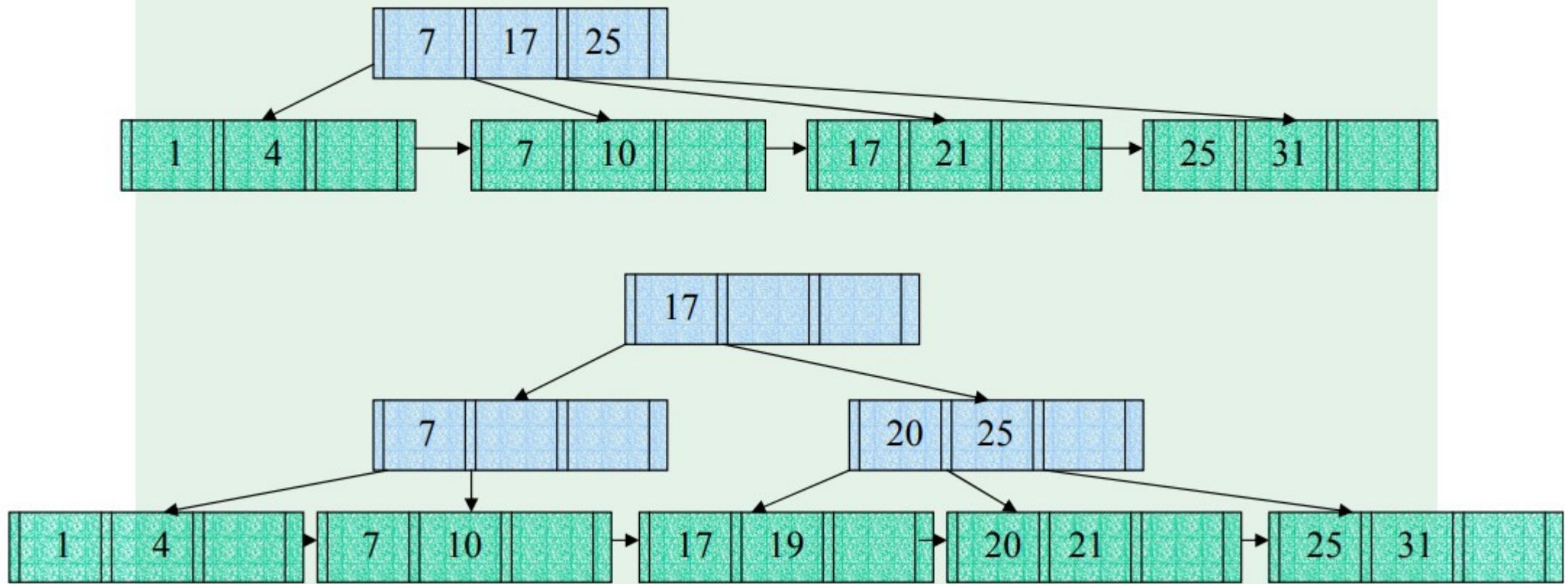
B-tree (above) and B+-tree (below) on same data



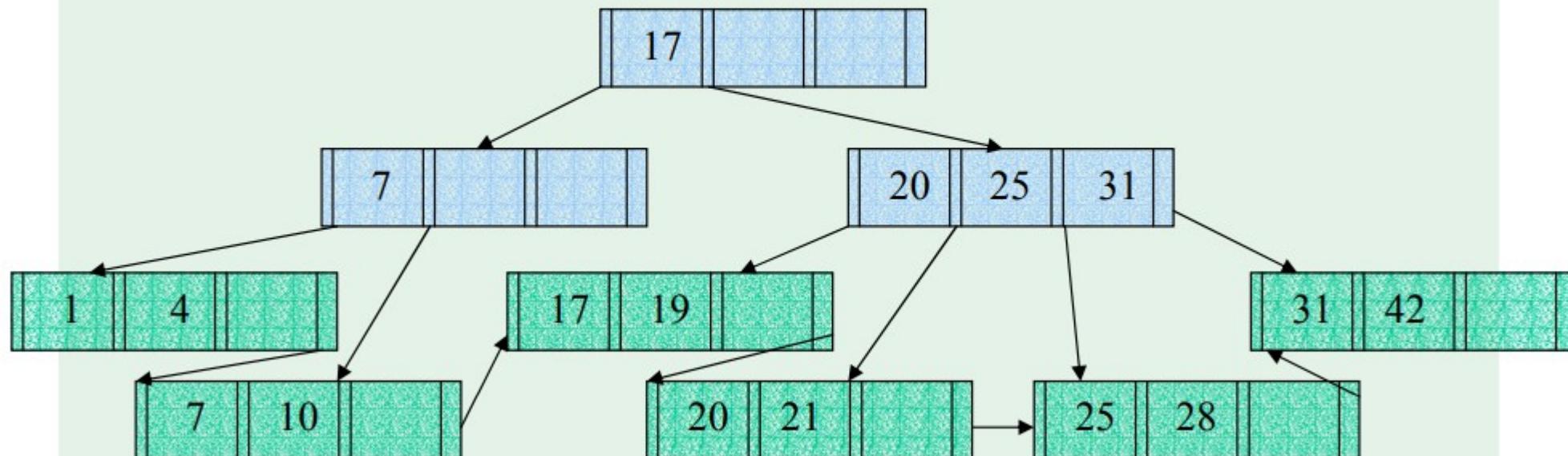
- Example 1: Construct a B⁺ tree for (1, 4, 7, 10, 17, 21, 31, 25, 19, 20, 28, 42) with n=4.



- 1, 4, 7, 10, 17, 21, 31, 25, 19, 20, 28, 42



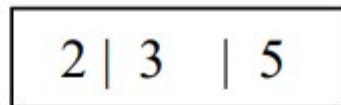
- 1, 4, 7, 10, 17, 19, 21, 25, 28, 31, 42



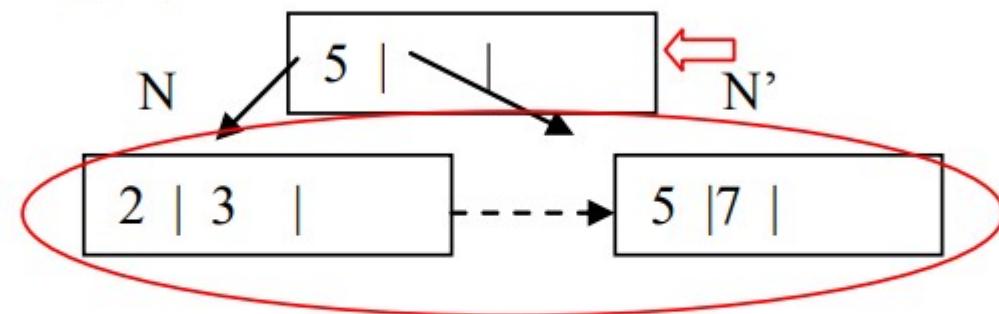
EXERCISE 1

- Construct a **B⁺** tree for the following set of values (2, 3, 5, 7, 11, 17, 19, 23, 29, 31)
- Assume that the tree is initially empty and values are inserted in ascending order.

B⁺tree: Insert 2, 3, 5



B⁺tree: Insert 7 (leaf split)

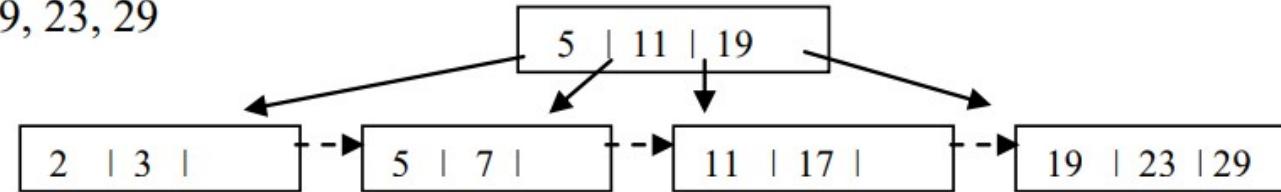


Split a leaf node:

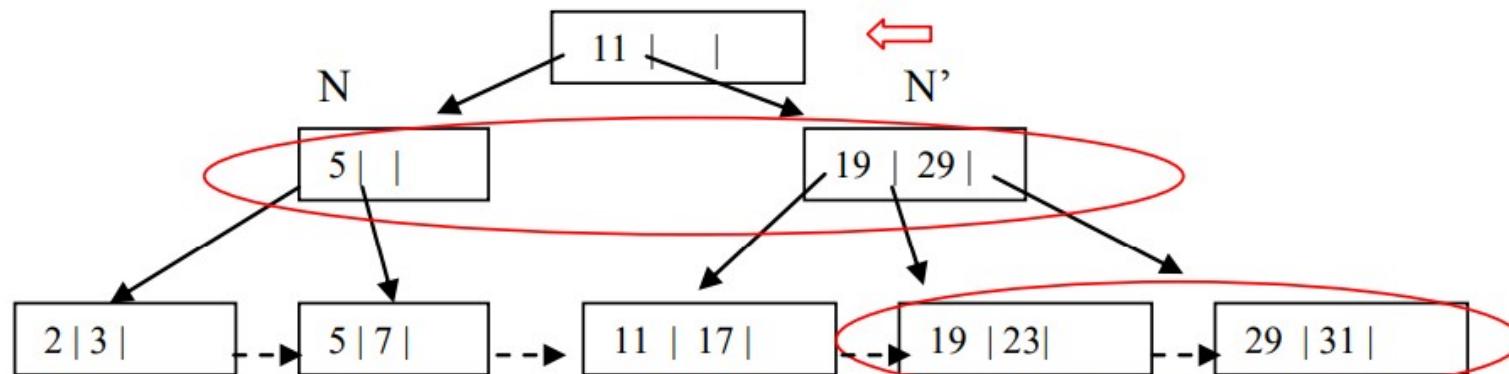
Let K_1, \dots, K_m be the set of keys in the ascending order (i.e. 2, 3, 5, 7)

- Node N: $K_1, \dots, K_{\lceil m/2 \rceil - 1}, K_{\lceil m/2 \rceil}$ (i.e. 2, 3)
- Node N': $K_{\lceil m/2 \rceil + 1}, \dots, K_m$ (i.e. 5, 7)
- Insert $K_{\lceil m/2 \rceil + 1}$ into the parent node (i.e. 5)

B+tree: Insert 11, 17, 19, 23, 29



B+tree: Insert 31 (leaf node split + non leaf node split)



Split an internal node N:

Let K_1, \dots, K_m be the set of keys in the ascending order (i.e. 5, 11, 19, 29)

- Node N: $K_1, \dots, K_{\lceil m/2 \rceil - 1}$ (i.e. 5)
- Node N': $K_{\lceil m/2 \rceil + 1}, \dots, K_m$ (i.e. 19, 29)
- Insert $K_{\lceil m/2 \rceil}$ into the parent node of N (i.e. 11)

- [Indexing In DBMS | Why Indexing is used | With Examples | Basics 1/2 – YouTube](#)
- [Types Of Indexing In DBMS With Examples | Indexing Basics 2/2 – YouTube](#)
- [classification indexing | DBMS – YouTube](#)
- [indexing in database example | dbms – YouTube](#)
- [B+ tree in database | Introduction & Example | DBMS | Bhanu Priya – YouTube](#)
- <https://www.javatpoint.com/b-tree-vs-bplus-tree#:~:text=Let's%20understand%20the%20property%20through,that%20is%20not%20a%20B%20tree>
-





BITS Pilani

Pilani | Dubai | Goa | Hyderabad

Database Design





Contact Session 12:

Transaction Processing,

Concurrency Control and Schedules

Objective:

Understanding the Transaction Processing System (TPS), characteristics of the transaction and notions underlying TPS and concurrent transaction execution, Schedules and Serializability.

Contents:

- Introduction to Transactions in DBMS
- Transaction Model
- Concurrent Transactions and Issues
- Understanding Schedules and Serializability
- Concurrency Control and Conflict Serializability
- Test for Conflict Serializability

Types of DBMS

Single-User: at most one user at a time can use the system

Multiuser: many users can use the system concurrently.

Multiprogramming: allows the computer to execute multiple programs at the same time.

Interleaving: keeps the CPU busy when a process requires an input or output operation, the CPU switched to execute another process rather than remaining idle during I/O time.

Most of the theory concerning concurrency control in databases is developed in terms of interleaved concurrency

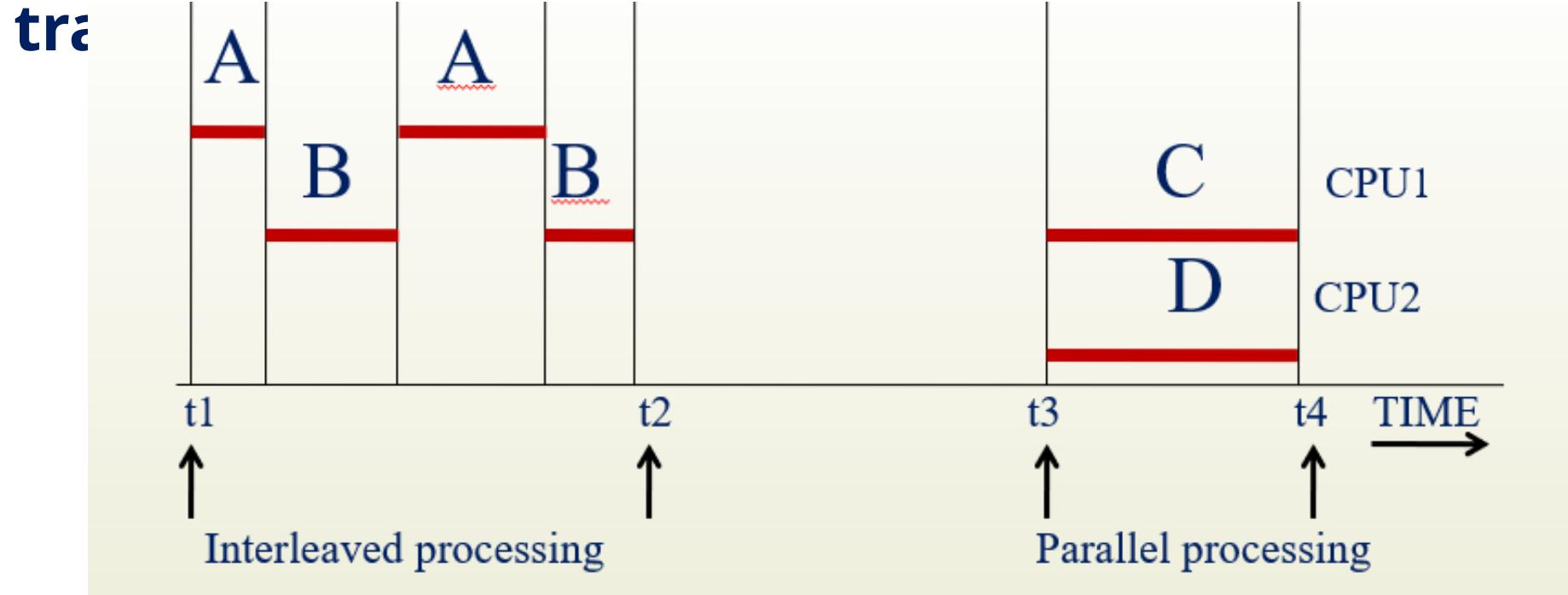
Execution of transactions in DBMS : Sequential Vs Concurrent Execution Sequential Execution

When transactions are strictly executed one after the other it is termed sequential execution

Concurrent Execution

When two or more transactions are interleaved, then we say that it is a concurrent execution

Interleaved Vs parallel processing of the concurrent



If the computer system has multiple hardware processors (CPUs), **parallel processing** of multiple processes is possible, as illustrated by processes C and D in the above figure.

Transaction

serves as a way to describe logical units of database processing that involve one or more database access operations.

Transaction Processing Systems:

The system with large databases and hundreds of concurrent users that are executing database transactions.

E.g., Banking, Airline reservations, credit card processing, etc.,

Operations on Transaction:

- All database access operations between **Begin Transaction** and **End Transaction** statements are considered one logical transaction.
- If the database operations in a transaction do not update the database but only retrieve data, the transaction is called a **read-only transaction**.

Basic database access operations :

- **read_item(X)** : reads a database item X into program variable.
- **Write_item(X)** : Writes the value of program variable X into the database item X.

- **Example:**

Suppose an employee of a bank transfers Rs 800 from X's account to Y's account. This small transaction contains several low-level tasks:

X's Account

Open_Account(X)

Old_Balance = X.balance

New_Balance = Old_Balance - 800

X.balance = New_Balance

Close_Account(X)

Y's Account

Open_Account(Y)

Old_Balance = Y.balance

New_Balance = Old_Balance + 800

Y.balance = New_Balance

Close_Account(Y)

Transaction

A **transaction** is an atomic unit of work that is either completed in its entirety or not done at all.

- ✓ For recovery purposes, the system needs to keep track of when the transaction starts, terminates, and commits or aborts.
- ✓ **The recovery manager keeps track of the following operations :**
 - **BEGIN_TRANSACTION**
 - **READ OR WRITE**
 - **END_TRANSACTION**
 - **COMMIT_TRANSACTION**
 - **ROLLBACK**

Transaction states

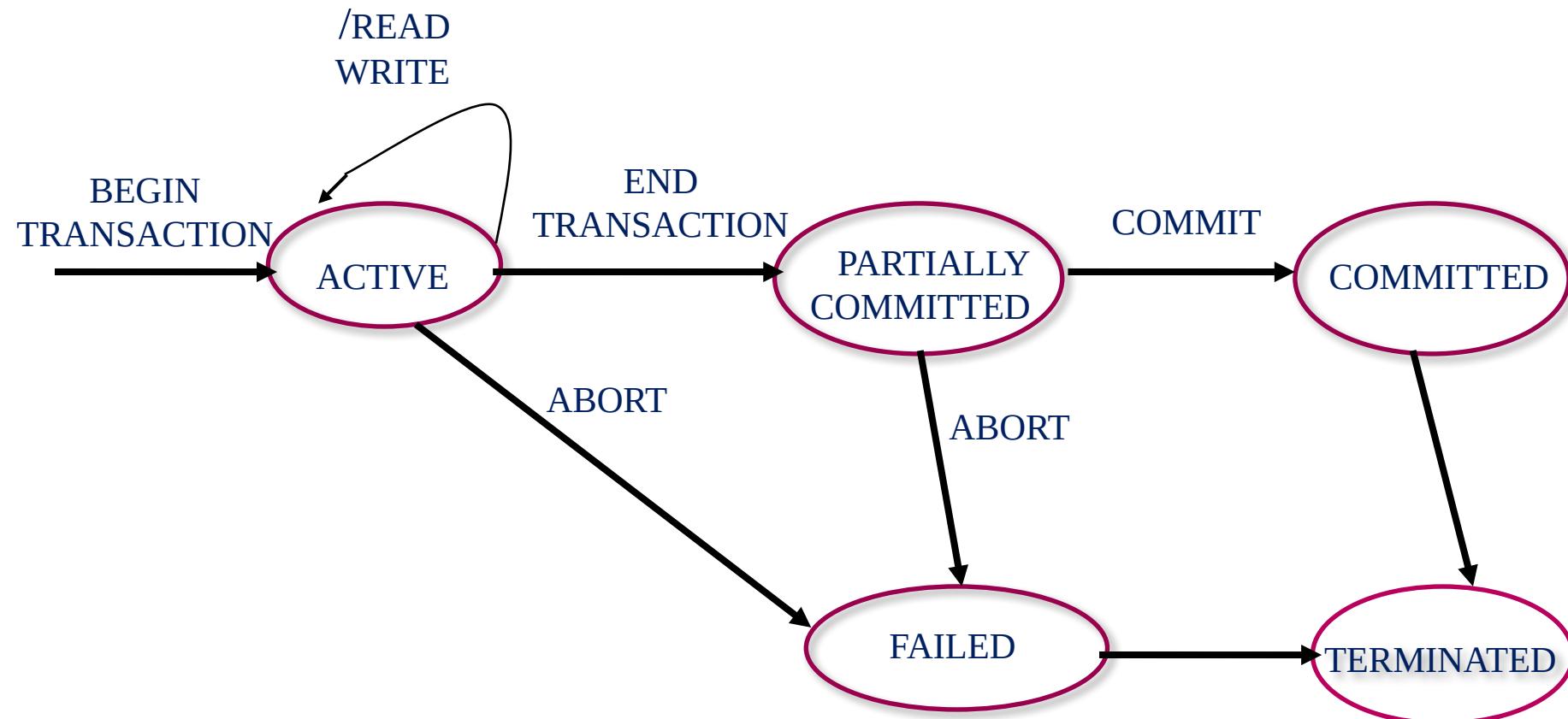


Figure 1: State transition diagram illustrating the states for transaction execution

The Recovery manager keeps track of the following operations:

Begin Transaction: This is where transaction execution starts. When a transaction begins, it is in its **active state**.

Read/Write: These describe the read and write operations that are performed on database items as part of a transaction.

Commit Transaction: This indicates that the transaction has been completed successfully, allowing any changes (updates) made by the transaction to be safely committed to the database and not undone.

Partially Committed State: When the last statement is executed but the result is not written to the database, this state is achieved.

Failed State: After discovering that the normal execution cannot be continued a transaction is aborted and reaches failed state. In order to ensure atomicity property, failed transactions should have no effect on the database

Difference between Failed and Partially committed state

If all the 'read and write' operations are performed without any error then it goes to the "partially committed state"; if any instruction fails, it goes to the "failed state". After completion of all the read and write operation the changes are made in main memory or local buffer.

Rollback/ Abort: This indicates that the transaction failed and that any database modifications or effects caused by the transaction must be undone. As a result, the database must be restored to the state it was in just before the transaction started.

End Transaction: This indicates that read and write transaction activities have finished and that transaction execution has reached its completion.

- It's reached after the transaction's failure or success
- It may be required at this point to determine if the transaction's changes can be permanently applied to the database, or whether the transaction must be cancelled due to a violation of concurrency control or another

The Recovery manager keeps track of the following operations...

Undo: It works in a similar way as rollback, however, it only affects a single operation rather than the entire transaction.

Redo:

This specifies that specific transaction activities must be performed in order to

guarantee that a committed transaction's operations have been successfully

implemented in the database.

Force writing a log:

- Before a transaction reaches its commit point, any portion of the log that has not been written to the disk yet must now be written to the

Why Recovery is needed?

There are several possible reasons for a transaction to fail:

- **A computer failure:** A **hardware, software, or network error occurs** in the computer system during transaction execution.
- **A transaction or system error:** Some operations in the transaction may cause it to fail such as **integer overflow or division by zero**. Transaction failure may also occur because of erroneous parameter values or because of a logical programming error. In addition, **the user may interrupt the transaction** during its execution.
- **Local errors or exception conditions**

Detected by the transaction. For example, an **insufficient account balance** in a banking database may cause a transaction, such as a fund withdrawal from that account, to be cancelled.

Why Recovery is needed?

- **Concurrency control enforcement:** The concurrency control method may decide to abort the transaction. Because it **violates serializability or because several transactions are in a state of deadlock**
- **Disk failure:** all disks or some disk blocks may lose their data because of a **disk read/write head crash**
- **Physical problems:** **Disasters, theft, fire, etc.**

Therefore, the system must keep sufficient information to recover from the failure.

System Log

The System Log

The system **maintains a log** to keep track of all transaction operations that affect the values of database items. This log may be needed to recover from failures.

Types of log records :

- **[start_transaction,T]** : indicates that transaction T has started execution.
- **[write_item,T,X,old_value,new_value]** : indicates that transaction T has changed the value of database item X from old_value to new_value.
(new_value may not be recorded)
- **[read_item,T,X]**: indicates that transaction T has read the value of database item X.
(read_item may not be recorded)
- **[commit,T]**: transaction T has been recorded permanently.
- **[abort,T]**: indicates that transaction T has been aborted.

ACID should be enforced by the concurrency control and recovery methods of the DBMS.

Atomicity Consistency Isolation Durability (ACID) properties of transactions :

- **Atomicity:** a transaction is an atomic unit of processing; it is either performed entirely or not performed at all.
- **(It is the responsibility of recovery)**
- **Consistency :** transfer the database from one consistent state to another consistent state [compatible or in agreement with something---standard and effect over time]
- **(It is the responsibility of the applications and DBMS to maintain the constraints)**

Transaction Properties

- **Isolation:** the execution of the transaction should be isolated from other transactions (Locking) - **(It is the responsibility of concurrency control sub system)**
 - * Isolation level:
 - Level 0 – (No dirty reads):** This means the database will not read any of the uncommitted values
 - Level 1 (No lost update) –** A lost update happens when one transaction overwrites the changes made by another transaction without seeing them.
 - Level 2 (No dirty+ no lost)**
 - Level 3 (level 2+repeatable reads) –Repeatable reads** - This is the most restrictive isolation level. The transaction holds read locks on all rows it references and writes locks on referenced rows for update and delete actions. Since other transactions cannot read, update or delete these rows, consequently it avoids non-repeatable read.
 - **Durability:** committed transactions must persist in the database,
 - i.e. those changes must not be lost because of any failure.

Concurrency Control

In a database management system (DBMS), concurrency control handles simultaneous access to a database.

It prevents two or more users from simultaneously accessing the same record and **serializes** transactions for backup and recovery.

Why concurrency control is essential?

Several problems can occur when concurrent transactions execute in an uncontrolled manner:

- **The Lost Update Problem**
- **The Temporary Update (or Dirty Read) Problem**
- **The Incorrect Summary Problem**
- **Unrepeatable Read problem**

Lost Update Problem:

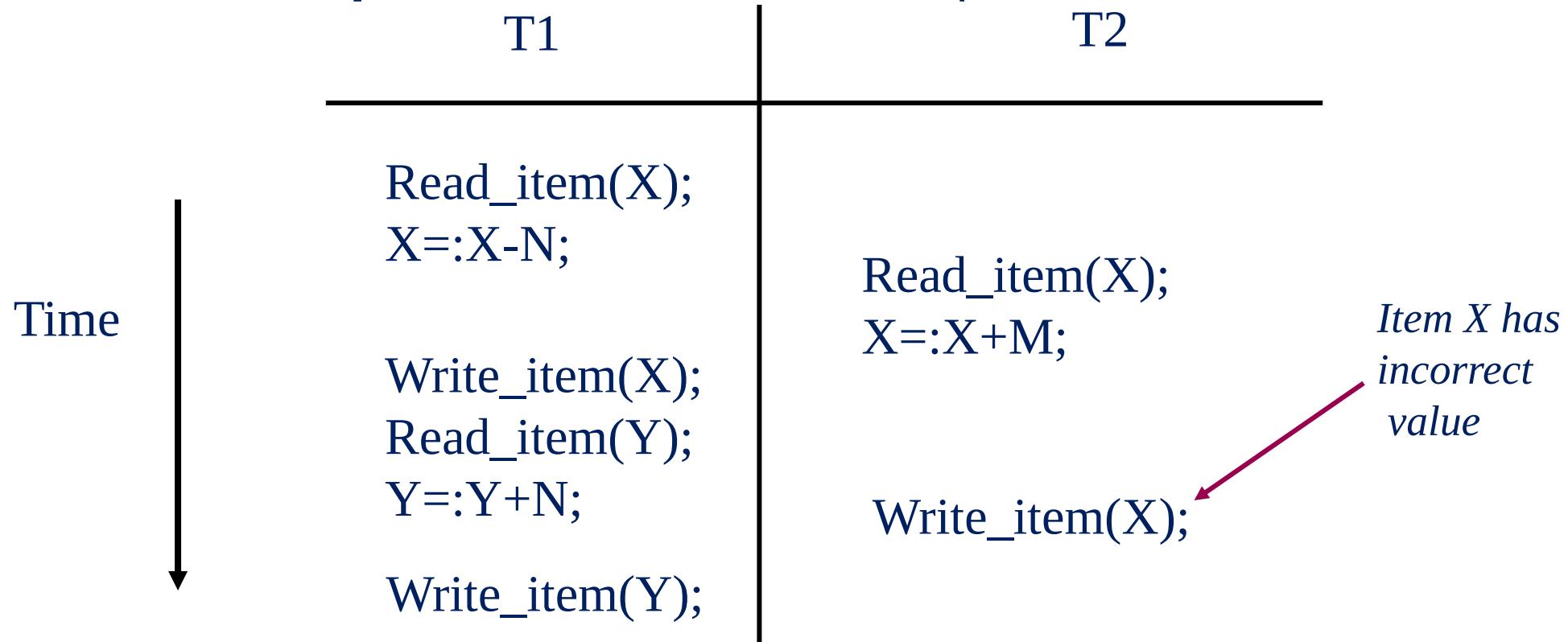
In the lost update problem, an update done to a data item by a transaction is lost as it is overwritten by the update done by another transaction.

Example 1

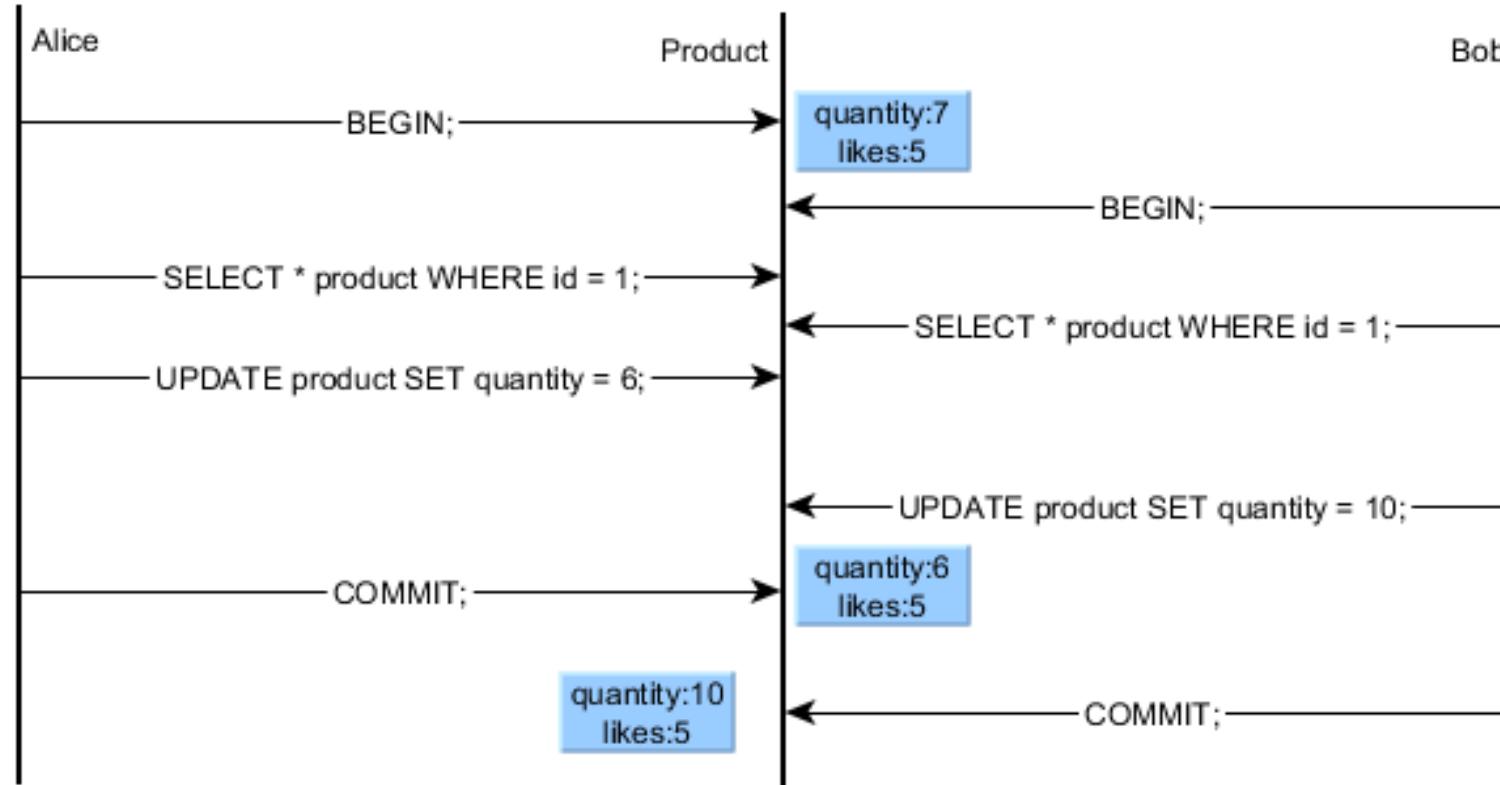
T1	T2
<code>read_item(X) X = X + N</code>	<code>X = X + 10 write_item(X)</code>

In the above example, transaction 2 changes the value of X but it will get overwritten by the write commit by transaction 1 on X (*not shown in the image above*). Therefore, the update done by transaction 2 will be lost. Basically, the write commit done by the **last transaction** will overwrite all previous write commits.

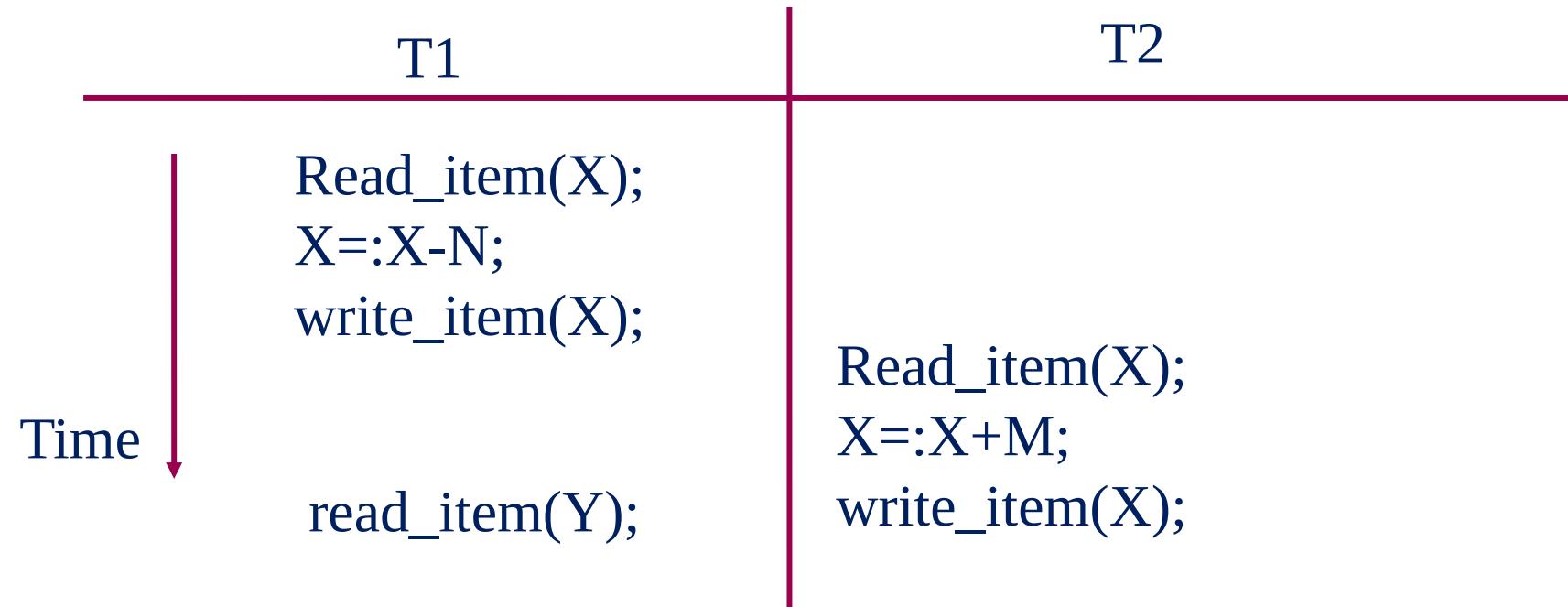
1. The Lost Update Problem : Example 2:



Example: 2 A lost update occurs when two different transactions are trying to update the same column on the same row within a database at the same time

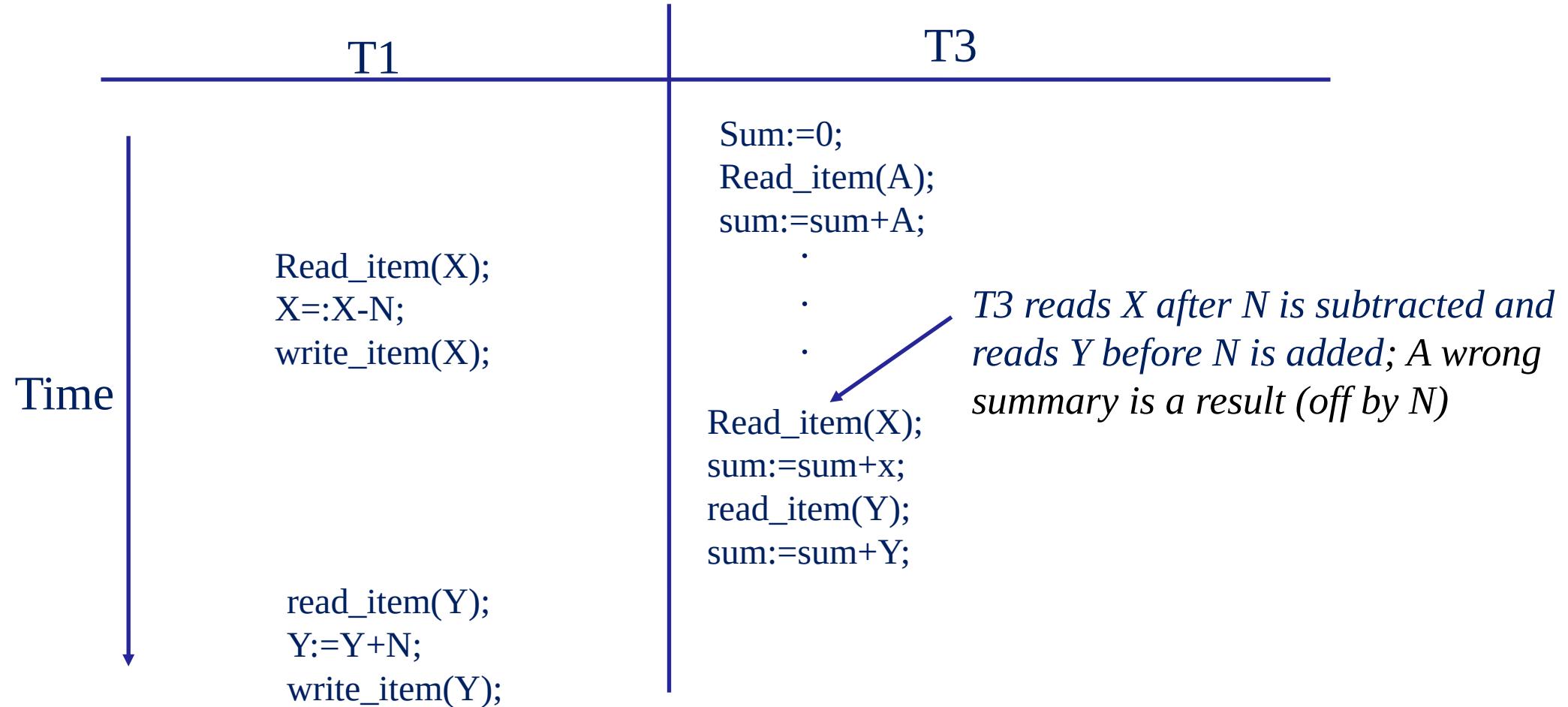


2. The temporary Update (or Dirty read) problem



T1 fails and must rollback, meanwhile T2 has read the temporary value

3. The incorrect summary problem



4. Unrepeatable Read problem: A transaction reads items twice with two different values because it was changed by another transaction **between the two reads**.

*Also known as **Inconsistent Retrievals Problem** that occurs when in a transaction, two different values are read for the same database item.*

T1 reads X again, however T2 has changed the value of X after the first read

T1	T2
read-item (X); read-item (X) X:=X-N; write-item (X);	read-item (X); X:=X+M; write-item (X);

5. Phantom Read Problem: The phantom read problem happens when a transaction reads a variable once but then encounters an error stating that the variable does not exist when it tries to read it again.

T1	T2
Read (A)	
	Read (A)
Delete (A)	
	Read (A)

Table 1

In the example above Table 1, after transaction 2 reads the variable A, transaction 1 deletes the variable A without the knowledge of transaction 2. As a result, when transaction 2 attempts to read A, it is unable to do so.

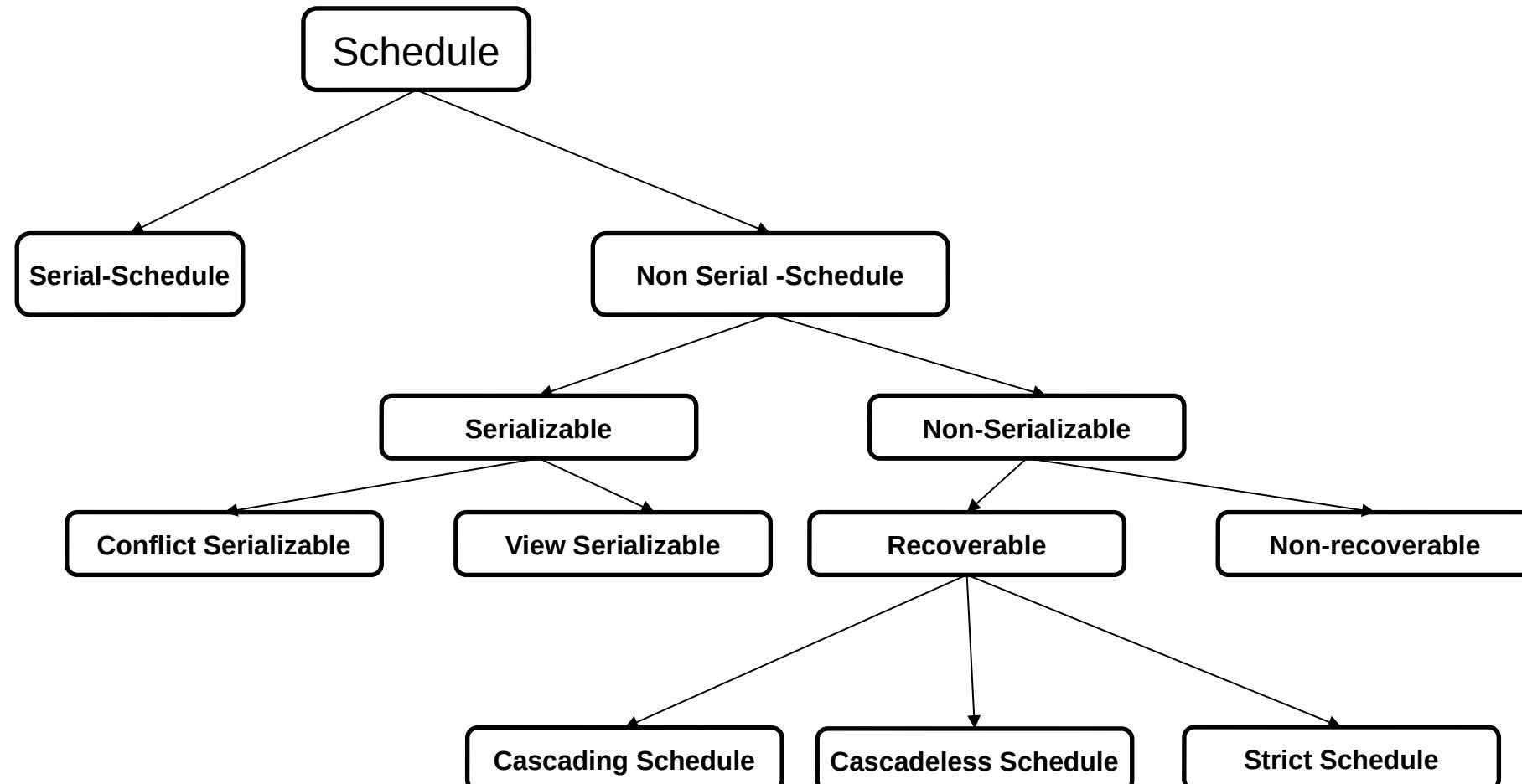
Schedule(or History) : A series of operations from one transaction to another transaction is known as a schedule. It is used to preserve the order of the operation in each of the individual transaction.

Schedule(or History) S of n transactions T_1, T_2, \dots, T_n is an ordering of operations of the transactions with the following conditions :

- for each transaction T_i that participates in S , the operations of T_i in S must appear in the same order in which they occur in T_i
- the operations from other transactions T_j can be interleaved with the operations of T_i in S .

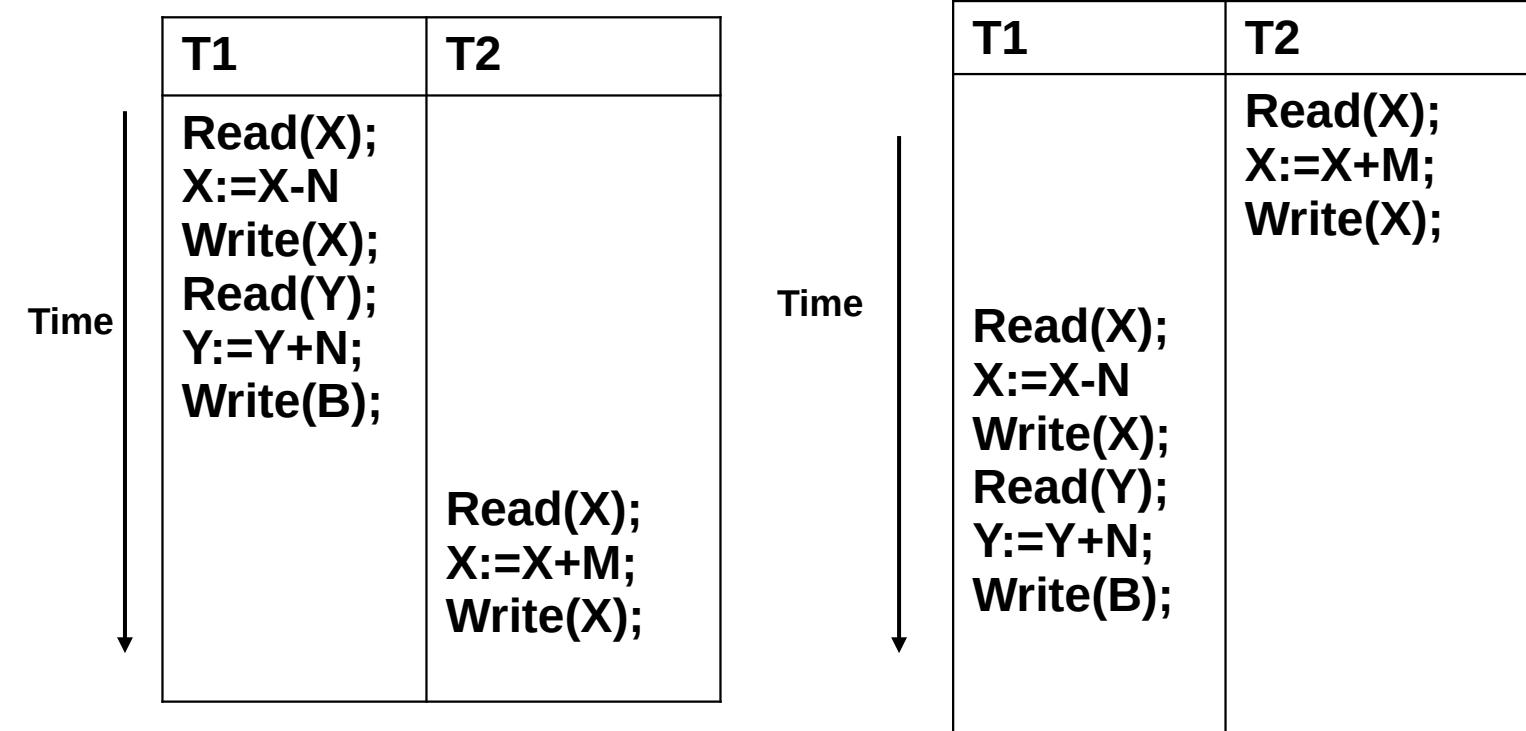
The symbols **r,w,c**, and **a** are used for the operations **read_item**, **write_item**, **commit**, and **abort** respectively.

Transaction Schedules & Serializability



Serial Schedule

- In the serial schedule, when the first transaction completes its cycle, then the next transaction is executed.
- All the transactions execute serially one after the other.
- When one transaction executes, no other transaction is allowed to execute.



The diagram illustrates two serial transaction schedules, A and B, represented as tables. Each table has two columns: T1 and T2. A vertical double-headed arrow between the two tables is labeled 'Time', indicating the sequence of events over time.

T1	T2
Read(X); X:=X-N Write(X); Read(Y); Y:=Y+N; Write(B);	Read(X); X:=X+M; Write(X);

T1	T2
Read(X); X:=X-N Write(X); Read(Y); Y:=Y+N; Write(B);	Read(X); X:=X+M; Write(X);

Characteristics

- Serial schedules are always-
- Consistent
 - Recoverable
 - Cascadeless
 - Strict

Figure (a)
Schedule A

Schedule A shows the serial schedule where T1 is followed by T2.

Figure (b)
Schedule B

Schedule B shows the serial schedule where T2 is followed by T1.

Non-serial Schedule

- If interleaving of operations is allowed, then there will be a non-serial schedule.
- It contains many possible orders in which the system can execute the individual operations of the transactions.
- In the given figure (c) and (d), Schedule C and Schedule D are the non-serial schedules. It has interleaving of operations.

Characteristics-

Non-serial schedules are **NOT** always-

- Consistent
- Recoverable
- Cascadeless
- Strict

Non-serial Schedule

Time ↓

T1	T2
Read(X); X:=X-N	
	Read(X); X:=X+M;
Write(X); Read(Y);	Write(X);

Figure (c)
Schedule C

Time ↓

T1	T2
Read(X); X:=X-N	
Write(X);	Read(X); X:=X+M;

Figure (d)
Schedule D

Serializable schedule

- The serializability of schedules is used to find non-serial schedules that allow the transaction to execute concurrently without interfering with one another.
- It identifies which schedules are correct when executions of the transaction have to interleave their operations.
- A non-serial schedule will be serializable if its result is equal to the result of its transactions executed serially.

These are of two types:

- i) **Conflict Serializable**
- ii) **View Serializable**

Note: A concurrent schedule whose result is the same as that of a serial schedule is called a **concurrent serializable schedule**

i) **Conflict Serializable** : A schedule S is said to be conflict serializable if it is *conflict equivalent* to some serial schedule S'.

Conflict Equivalent: Two schedules are said to be conflict equivalent when one can be transformed to another by swapping **non-conflicting** operations.

Conflict Operations:

For transactions T1 & T2 the order of read operation on any data element does not matter.

{T1R(Q), T2R(Q)} or **{T2R(Q), T1R(Q)}** does not matter. The result is the same and **does not lead to any conflict**.

Conflicting Operations:

But **{T1R(Q), T2W(Q)}** is **not same as {T2W(Q), T1R(Q)}**

If **I_i** and **I_j** are the operations (instructions) of two different transactions on the same data item, and at least one of these instructions is a WRITE operation then we say that **I_i** and **I_j** **are conflict operations**.

Here, Q is the data element. **I** stands for instruction and **i** and **j** are transactions.

Hence it is evident that if ***we swap non-conflicting operations of a concurrent schedule, it will not affect the final result.***

Example

T ₁	T ₂
R(A)	
W(A)	R(A)
	W(A)
R(B)	
W(B)	R(B)
	W(B)

(S₁)
Concurrent schedule
with T₁ & T₂
accessing A, B (data
item)

T ₁	T ₂
R(A)	
W(A)	R(A)
	W(A)
R(B)	
W(B)	R(B)
	W(B)

(S₂)
Swap W(A) in T₂ with
R(B) in T₁ (because
they are non
conflicting)

T ₁	T ₂
R(A)	
W(A)	R(A)
R(B)	
W(B)	R(B)
	W(B)

(S₃)
Swap R(A) of T₂ with
R(B) of T₁ and W(A)
in T₂ with W(B) in T₁
(Since non
conflicting)

T ₁	T ₂
R(A)	
W(A)	R(A)
R(B)	
W(B)	R(B)
	W(B)

(S₄)
Swap R(A) of T₂ with
W(B) of T₁
Now, the final schedule (S₄) is
a serial schedule

Conflict Equivalent Schedules

We say S and S' are conflict equivalent if a schedule S can be transformed into a schedule S' by swapping non-conflicting instructions.

Furthermore, if a schedule S is **conflict equivalent** to a serial schedule, it is said to be **conflict serializable**.

In the preceding example, **S_4 is a serial schedule that is conflict equivalent to S_1** .

As a result, S_1 is a conflict serializable schedule.

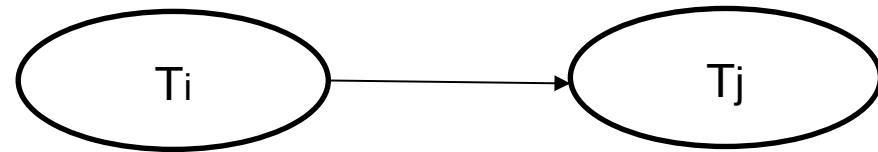
T_1	T_2
$R(\theta)$	
	$W(\theta)$

Now, in this schedule, we can not perform any swap between instructions of T_1 and T_2 . Hence, it is **not conflict serializable**

Test for Conflict Serializability

- ✓ Assume a schedule S.
- ✓ Let's create a graph called a **precedence graph/serialization graph** for S.
- ✓ $G = (V, E)$ is a pair in this graph, where V is a collection of vertices and E is a set of edges.
- ✓ All of the transactions in the schedule are represented by a set of vertices.
- ✓ All edges $T_i \rightarrow T_j$ that satisfy one of the three conditions are included in the set of edges:
 - i. Create a node $T_i \rightarrow T_j$ if T_i executes write (Q) before T_j executes read (Q).
 - ii. Create a node $T_i \rightarrow T_j$ if T_i executes read (Q) before T_j executes write (Q).
 - iii. Create a node $T_i \rightarrow T_j$ if T_i executes write (Q) before T_j executes write (Q).

Precedence Graph for Schedule S:

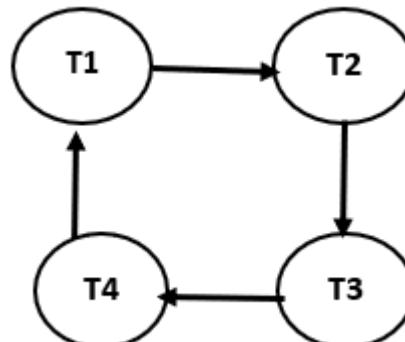


- ✓ If a precedence graph contains a single edge $Ti \rightarrow Tj$, then all the instructions of Ti are executed before the first instruction of Tj is executed.
- ✓ If a precedence graph for schedule S contains a cycle, then S is non-serializable. If the precedence graph has no cycle, then S is known as serializable.

Example 1

Sh1: $r1(J); r2(K); w2(J); r3(L); w3(K); w4(L);$

T1	T2	T3	T4
$R(J)$			
	$R(K)$		
	$W(J)$		
	$R(L)$		
	$W(K)$		
		$W(L)$	
		$R(M)$	
			$W(M)$

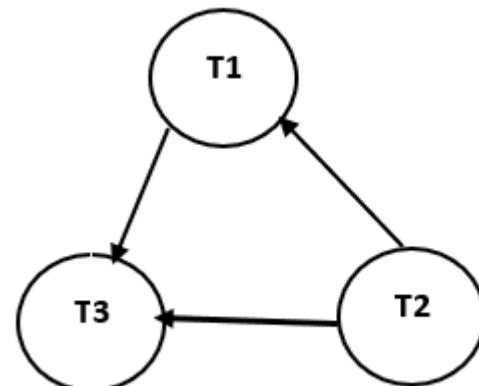


Cycle formed in the precedence graph in the given schedule Sh1.

Therefore, it is “not conflict serializable”.

Example 2

Sh2: r1(A); r2(B); r2(C); w1(B); w3(A); w3(C);



There is no cycle formed in the precedence graph in the given schedule Sh2.

Therefore, it is “Conflict serializable”.

Conflict equivalent:

- ✓ If the order of any two conflicting operations is the same in both schedules,
they are said to be conflict equivalent.
- ✓ In the conflict equivalent, one can be transformed into another by swapping
non-conflicting operations.
- ✓ Two schedules are said to be conflict equivalent if and only if:
 - They contain the same set of transactions.
 - If each pair of conflict operations are ordered in the same way.

- ✓ In the following example,

S2 is conflict equivalent to S1

(S1 can be converted to S2 by swapping non-conflicting operations).

T1	T2
Read(X);	
Write(X);	Read(X);
	Write(X);
Read(Y);	
Write(Y);	
	Read(Y);
	Write(Y);

Fig 1 Schedule S1

Schedule S1 becomes

T1	T2
Read(X);	
Write(X);	
Read(Y);	
Write(Y);	
	Read(X);
	Write(X);
	Read(Y);
	Write(Y);

Fig 2 Schedule S2

T1	T2
Read(X);	
Write(X);	
Read(Y);	
Write(Y);	
	Read(X);
	Write(X);
	Read(Y);
	Write(Y);

Fig 3 After swapping of non-conflict operations

- Schedule S2 is a serial schedule because, in this, all operations of T1 are performed before starting any operation of T2.
- Schedule S1 can be transformed into a serial schedule by swapping non-conflicting operations of S1.
- After swapping of non-conflict operations the schedule S1 (fig3), since S1 is conflict serializable)

Result Equivalent:

When two schedules produce the same final state of the database, they are said to be ***result equivalent***.

Note:

- Being serializable is not the same as being serial
- Being serializable implies that the schedule is the correct schedule.
 - It will leave the database in a consistent state.
 - The interleaving is appropriate and will result in a state as if the transactions were serially executed, yet will achieve efficiency due to concurrent execution.

ii) View Serializable

A schedule is view serializable if it is ***view equivalent*** to a serial schedule.

View Equivalent Schedules

Two schedules S and S' (where the same set of transactions participate in both schedules) are said to be view equivalent if the following ***three conditions are met***:

- a) For each data item Q , if the transaction T_i reads the initial value of Q in S , then transaction T_i must in schedule S' , also read the initial value of Q .
- b) For each data item Q , if transaction T_i executes read (Q) in S , and the value produced by transaction T_j (if any) then transaction T_i must in schedule S' also read the value of Q that was produced by transaction T_j
- c) For each data item Q , the transaction if any that performs the final write(Q) operation in schedule S must perform the final write(Q) operation in schedule S'

Now, we say that a schedule S is view serializable if it is view equivalent to a serial schedule

Transaction Schedules & Serializability



Non-Serial

S1

T1	T2
Read(X); Write(X);	
	Read(X); Write(X);
Read(Y); Write(Y);	Read(Y); Write(Y);
	Read(Y); Write(Y);

Serial

S2

T1	T2
Read(X); Write(X);	
	Read(Y); Write(Y);
;	Read(X); Write(X);
	;
	Read(Y); Write(Y);

We can say that given schedule s1 is **view serializable**
if we can prove that they are **view equivalent**.

Note: Every conflict serializable schedule is view serializable. But not all view serializable schedules are conflict serializable.

Non-Serializability in DBMS

A non-serial schedule, which is not serializable is called as non-serializable schedule.

Non-serializable schedules may or may not be consistent or recoverable.

Non-serializable schedule is divided into types:

- i. **Recoverable schedule**
- ii. **Non-recoverable schedule**

Recoverable Schedule

Due to a software issue, system crash, or hardware malfunction, a transaction may not finish completely. The unsuccessful transaction must be rolled back in this scenario. However, the value generated by the unsuccessful transaction may have been utilized by another transaction. As a result, we must also roll back those transactions.

Non-recoverable/ Irrecoverable Schedule

T1	T1's Buffer space	T2	T2's Buffer space	Database
Read(X);	X = 5000			X = 5000
X = X - 250;	X = 4750			X = 5000
Write(X);	X = 4750			X = 4750
		Read(X);	X = 4750	X = 4750
		A = A + 1000;	X = 5750	X = 4750
		Write(X);	X = 5750	X = 5750
		Commit;		
Failure Point				
Commit;				

Table 1

The above Table 1 shows a schedule that has two transactions.

T1 reads and writes the value of A and that value is read and written by T2.

T2 commits but later on, T1 fails. Due to the failure, we have to rollback T1.

T2 should also be rollback because it reads the value written by T1, but T2 can't be rollback because it is already committed. So this type of schedule is known as

"non-recoverable/irrecoverable" schedule.

Irrecoverable schedule: The schedule will be irrecoverable if T_j reads the updated value of T_i and T_i committed before T_j commit.

Recoverable with Cascade Rollback

T1	T1'S Buffer space	T2	T2's Buffer space	Database
Read(X);	X = 5000			X = 5000
X = X - 250;	X = 4750			X = 5000
Write (X);	X = 4750			X = 4750
		Read(X);	X = 4750	X = 4750
		A = A + 1000;	X = 5750	X = 4750
		Write (X);	X = 5750	X = 5750
Failure Point				
Commit;				
		Commit;		

Table 2

The above Table 2 shows a schedule with two transactions.

Transaction T1 reads and writes A, and that value is read and written by transaction T2. But later on, T1 fails. Due to this, we have to rollback T1.

T2 should be rollback because T2 has read the value written by T1. As it has not committed before T1 commits so we can rollback transaction T2 as well. So it is **recoverable with cascade rollback**.

Recoverable with cascading rollback: The schedule will be recoverable with cascading rollback if T_j reads the updated value of T_i . Commit of T_j is delayed till commit of T_i .

Cascadeless Recoverable Schedule

T1	T1'S Buffer space	T2	T2's Buffer space	Database
Read(X);	X = 5000			X = 5000
X = X - 250;	X = 4750			X = 5000
Write (X);	X = 4750			X = 4750
Commit;		Read (X);	X = 4750	X = 4750
		A = A + 1000;	X = 5750	X = 4750
		Write (X);	X = 5750	X = 5750
		Commit;		

Table 3

The above Table 3 shows a schedule with two transactions. Transaction T1 reads and write A and commits, and that value is read and written by T2. So this is a ***cascade less recoverable schedule***.

Strict Recoverable Schedule

When a transaction is not allowed to access or write data until the last transaction that has written it is committed or aborted, the schedule is referred to as Strict Schedule.

T1	T2
Read(X);	
	Read(X);
Write(X);	
Commit;	
	Write(X);
	Read(X);
	Commit;

Here, transaction T2 reads/writes the written value of transaction T1 only after the transaction T1 commits. Hence, the schedule is a ***strict schedule***.

Table 4

Let's have two transactions T1 and T2 in Table 4.

The write operation of transaction T1 precedes the read or write operation of transaction T2, so the commit or abort operation of transaction T1 should also precede the read or write of T2.

The Strict schedule allows only committed read and write operations.

This schedule implements more restrictions than cascadeless schedule.

SUMMARY

- ✓ **Transactions in DBMS**
- ✓ **Concurrent Transactions and Issues**
- ✓ **Schedules in DBMS**
 - Types of schedules
 - ❖ **Serial Schedule**
 - ❖ **Non-serial Schedule**
 - **Serializable**
 - Conflict Serializable & Test for Conflict Serializability
 - View Serializable
 - **Non-serializable**
 - **Non-Recoverable Schedule**
 - **Recoverable Schedule**
 - Cascading Schedule
 - Cascadeless Schedule
 - Strict Schedule





BITS Pilani

Pilani | Dubai | Goa | Hyderabad

Database Design



BITS Pilani

Pilani | Dubai | Goa | Hyderabad

Contact Session 13:

**Concurrency Control, Lock-based Protocols and Deadlocks,
Database Recovery**

Contents:

- Concurrency control and Implementing Serializability
- Lock-based protocols and Deadlocks
- Timestamp based protocols
- Introduction to Database recovery and Log-based recovery
- Checkpointing
- Shadow paging

Let's have a quick recap!!

Transaction

- Both concurrency control and crash recovery are based on the concept of a **transaction**.
- A transaction is a set of SQL statements chosen by the user.
- Example:
Transfer \$100 from one account to another:

```
BEGIN transaction
read balance from first account
add $100 to first account balance
write balance to first account
read balance from second account
verify balance to see if it contains at least $100
    if not, ABORT transaction
subtract $100 from second account
write balance to second account
COMMIT transaction
```

The ACID Properties of Transactions

- **Atomicity:** A transaction happens in its entirety or not at all
 - What if the OS crashed after \$100 was deposited to the first account?
 - The *recovery manager* of the DBMS must assure that the \$100 is withdrawn from the first account.
- **Consistency:** If the DB starts in a consistent state, (this notion is defined by the user; some of it may be enforced by integrity constraints) the transaction will transform it into a consistent state.
 - What if a transaction just deposited \$100 into an account?
 - The *programmer* must ensure that all transactions are consistent.
- **Isolation:** Each transaction is isolated from other transactions.
The effect on the DB is as if the transaction executed by itself.
 - What if another transaction computed the total bank balance after \$100 was deposited to the first account?
 - The *concurrency control subsystem* must ensure that all transactions run in isolation, unless the DBA chooses a less strict level of isolation.
- **Durability:** If a transaction commits, its changes to the database state persist
 - What if, after the commit, the OS crashed before the withdrawal was written to disk?
 - The *recovery manager* must assure that the withdrawal was at least logged.

Concurrency

- First we will study **isolation** of transactions, ensured by the **concurrency control** subsystem.
- Isolation is a problem only when **multiple users** are accessing the **same data**, and their actions **interleave**.
- Why is concurrency necessary?
 - **Applications** demand it
 - Better **utilization of resources**: While one user/transaction is reading the disk, another can be using the CPU or reading another disk. This results in better **throughput and response time**.

Let's start today's session!!

Purpose of Concurrency Control

- To enforce Isolation (through mutual exclusion) among conflicting transactions.
- To preserve database consistency through consistency preserving execution of transactions.
- To resolve read-write and write-write conflicts.

Example:

- In concurrent execution environment if T1 conflicts with T2 over a data item A, then the existing concurrency control decides if T1 or T2 should get the A and if the other transaction is rolled-back or waits.

Two-Phase Locking Techniques

- Locking is an operation which secures
 - (a) permission to Read
 - (b) permission to Write a data item for a transaction.
- Example:
 - Lock (X). Data item X is locked on behalf of the requesting transaction.
- Unlocking is an operation which removes these permissions from the data item.
- Example:
 - Unlock (X): Data item X is made available to all other transactions.
- Lock and Unlock are Atomic operations.

Two-Phase Locking Techniques: Essential components

- Two locks modes:
 - (a) shared (read) (b) exclusive (write).
 - Shared mode: shared lock (X)
 - More than one transaction can apply share lock on X for reading its value but no write lock can be applied on X by any other transaction.
 - Exclusive mode: Write lock (X)
 - Only one write lock on X can exist at any time and no shared lock can be applied by any other transaction on X.
 - Conflict matrix

		Read	Write
Read	Read	Y	N
	Write	N	N

Two-Phase Locking Techniques: Essential components

- **Lock Manager:**
 - Managing locks on data items.
- **Lock table:**
 - Lock manager uses it to store the identity of transaction locking a data item, the data item, lock mode and pointer to the next data item locked. One simple way to implement a lock table is through linked list.

Transaction ID	Data item id	lock mode	Ptr to next data item
T1	X1	Read	Next

Two-Phase Locking Techniques: Essential components

- Database requires that all transactions should be well-formed. A transaction is well-formed if:
 - It must lock the data item before it reads or writes to it.
 - It must not lock an already locked data items and it must not try to unlock a free data item.

Two-Phase Locking Techniques: Essential components

- The following code performs the lock operation:

```
B:if LOCK (X) = 0 (*item is unlocked*)
  then LOCK (X) ← 1 (*lock the item*)
  else begin
    wait (until lock (X) = 0) (until
      the lock manager wakes up the transaction);
  goto B
end;
```

Two-Phase Locking Techniques: Essential components

- The following code performs the unlock operation:

```
LOCK (X) ← 0 (*unlock the item*)
if any transactions are waiting then
    wake up one of the waiting transactions;
```

Two-Phase Locking Techniques: Essential components

- The following code performs the read operation:

```
B: if LOCK (X) = "unlocked" then
    begin LOCK (X) ← "read-locked";
        no_of_reads (X) ← 1;
    end
    else if LOCK (X) ← "read-locked" then
        no_of_reads (X) ← no_of_reads (X) +1
    else begin wait (until LOCK (X) = "unlocked" and
        the lock manager wakes up the transaction);
        go to B
    end;
```

Two-Phase Locking Techniques: Essential components

- The following code performs the write lock operation:

```
B: if LOCK (X) = “unlocked” then
    begin LOCK (X) ← “write-locked”;
        no_of_writes (X) ← 1;
    end
else if LOCK (X) ← “read-locked” then
    begin wait (until LOCK (X) = “unlocked” and
        the lock manager wakes up the transaction);
        go to B
    end;
```

Two-Phase Locking Techniques: Essential components

- The following code performs the unlock operation:

```
if LOCK (X) = "write-locked" then
begin LOCK (X) ← "unlocked";
    wakes up one of the transactions, if any
end
else if LOCK (X) ← "read-locked" then
begin
    no_of_reads (X) ← no_of_reads (X) -1
    if no_of_reads (X) = 0 then
begin
    LOCK (X) = "unlocked";
    wake up one of the transactions, if any
end
end;
```

Two-Phase Locking Techniques: Essential components

- Lock conversion
 - Lock upgrade: existing read lock to write lock
 - if T_i has a read-lock (X) and T_j has no read-lock (X) ($i \neq j$) then
 - convert read-lock (X) to write-lock (X)
 - else
 - force T_i to wait until T_j unlocks X
 - Lock downgrade: existing write lock to read lock
 - T_i has a write-lock (X) (*no transaction can have any lock on X*)
 - convert write-lock (X) to read-lock (X)

Two-Phase Locking Techniques: The algorithm

- Two Phases:
 - (a) Locking (Growing)
 - (b) Unlocking (Shrinking).
- **Locking (Growing) Phase:**
 - A transaction applies locks (read or write) on desired data items one at a time.
- **Unlocking (Shrinking) Phase:**
 - A transaction unlocks its locked data items one at a time.
- **Requirement:**
 - For a transaction these two phases must be **mutually exclusive**, that is, during locking phase unlocking phase must not start and during unlocking phase locking phase must not begin.

Two-Phase Locking Techniques: The algorithm

T1

```
read_lock (Y);  
read_item (Y);  
unlock (Y);  
write_lock (X);  
read_item (X);  
X:=X+Y;  
write_item (X);  
unlock (X);
```

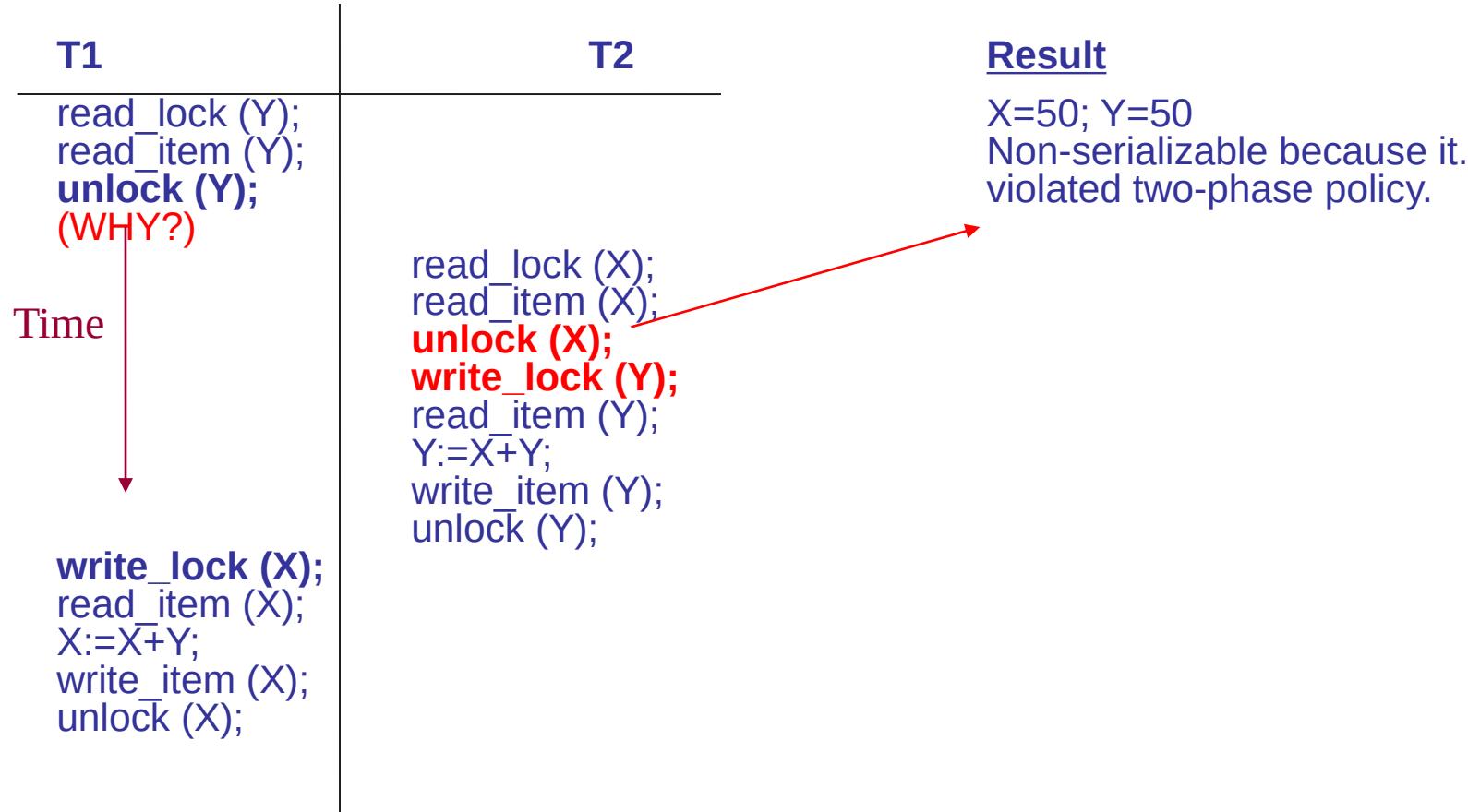
T2

```
read_lock (X);  
read_item (X);  
unlock (X);  
Write_lock (Y);  
read_item (Y);  
Y:=X+Y;           T2 followed by T1  
write_item (Y);  
unlock (Y);
```

Result

Initial values: X=20; Y=30
Result of serial execution
T1 followed by T2
X=50, Y=80.
Result of serial execution
T2 followed by T1
X=80, Y=50

Two-Phase Locking Techniques: The algorithm



Conditions for Two-phase locking protocol



- The two-phase locking divides the transaction execution phase into three components.
- Phase 1: When the transaction's execution begins in the first part, it requests permission for the lock it requires.
- Phase 2: The transaction then obtains all of the locks in the second part.
- Phase 3: The third phase begins when the transaction's first lock is released.
- The transaction cannot demand any new locks in the third phase. It only unlocks the locks that have been acquired.

Conditions for Two-phase locking protocol



- A transaction follows the two-phase locking protocol if locking and unlocking can be done in two phases. The two phases of the two-phase protocol are:
 - **Growing phase:** During the growth phase, new locks on data items may be acquired, but none can be released.
 - **Shrinking phase:** Existing locks may be released, but no new locks can be acquired during the shrinking phase.
- **Note:** If lock conversion is allowed, the following phase may take place:
 - Upgrading of lock (from S(a) → X(a)) is allowed in the growing phase.
 - Downgrading of lock (from X(a) → S(a)) must be done in the shrinking phase.
- **Lock Point:** It is a point at which the growing phase comes to a close, i.e., when a transaction obtains the last lock it requires.
- Let's look at an example:

Conditions for Two-phase locking protocol



	T1	T2
1	Lock - S(A)	
2		Lock - S(A)
3	Lock - X(B)	
4
5	Unlock(A)	
6		Lock - X(C)
7	Unlock(B)	
8		Unlock(A)
9		Unlock(C)
10

The way unlocking and locking works in two-phase locking is:

Transaction T1:

- The growing phase is from steps 1-3.
- The shrinking phase is from steps 5-7.
- The locking point is at point 3.

Transaction T2:

- The growing phase is from steps 2-6.
- The shrinking phase is from steps 8-9.
- The locking point is at point 6.

The type mentioned above of 2-PL is Basic 2PL. It ensures Serializability, but it does not prevent Cascading Rollback and Deadlock.

Two-Phase Locking Techniques: The algorithm

T'1

```
read_lock (Y);  
read_item (Y);  
write_lock (X);  
unlock (Y);  
read_item (X);  
X:=X+Y;  
write_item (X);  
unlock (X);
```

T'2

```
read_lock (X);  
read_item (X);  
Write_lock (Y);  
unlock (X);  
read_item (Y);  
write_item (Y);  
unlock (Y);
```

T1 and T2 follow two-phase policy but they are subject to deadlock, which must be dealt with.

Two-Phase Locking Techniques: The algorithm

- Two-phase policy generates two locking algorithms
 - (a) **Basic**
 - (b) **Conservative**
- **Conservative:**
 - Prevents deadlock by locking all desired data items before transaction begins execution.
- **Basic:**
 - Transaction locks data items incrementally. This may cause deadlock which is dealt with.
- **Strict:tp**
- A more stricter version of Basic algorithm where unlocking is performed after a transaction terminates (commits or aborts and rolled-back).
- This is the most commonly used two-phase locking algorithm.

Dealing with Deadlock and Starvation

▪ Deadlock

T'1

```
read_lock (Y);  
read_item (Y);
```

```
write_lock (X);  
(waits for X)
```

T'2

```
read_lock (X);  
read_item (Y);
```

```
write_lock (Y);  
(waits for Y)
```

T1 and T2 did follow two-phase policy but they are deadlock

▪ Deadlock (T'1 and T'2)

Dealing with Deadlock and Starvation

■ **Deadlock prevention**

- A transaction locks all data items it refers to before it begins execution.
- This way of locking prevents deadlock since a transaction never waits for a data item.
- The conservative two-phase locking uses this approach.

Dealing with Deadlock and Starvation

■ **Deadlock detection and resolution**

- In this approach, deadlocks are allowed to happen.
- The scheduler maintains a wait-for-graph for detecting cycle.
- If a cycle exists, then one transaction involved in the cycle is selected (victim) and rolled-back.
- A wait-for-graph is created using the lock table.
- As soon as a transaction is blocked, it is added to the graph.
- When a chain like: T_i waits for T_j waits for T_k waits for T_i or T_j occurs, then this creates a cycle.
- One of the transaction should be rolled back.

Dealing with Deadlock and Starvation

■ **Deadlock avoidance**

- There are many variations of two-phase locking algorithm.
- Some avoid deadlock by not letting the cycle to complete.
- That is as soon as the algorithm discovers that blocking a transaction is likely to create a cycle, it rolls back the transaction.
- Wound-Wait and Wait-Die algorithms use timestamps to avoid deadlocks by rolling-back victim.

Dealing with Deadlock and Starvation

■ Starvation

- Starvation occurs when a particular transaction consistently waits or restarted and never gets a chance to proceed further.
- In a deadlock resolution it is possible that the same transaction may consistently be selected as victim and rolled-back.
- This limitation is inherent in all priority based scheduling mechanisms.
- In Wound-Wait scheme a younger transaction may always be wounded (aborted) by a long running older transaction which may create starvation.

Timestamp based concurrency control algorithm

■ **Timestamp**

- A monotonically increasing variable (integer) indicating the age of an operation or a transaction.
- A larger timestamp value indicates a more recent event or operation.
- Timestamp based algorithm uses timestamp to serialize the execution of concurrent transactions.

Timestamp based concurrency control algorithm

■ Basic Timestamp Ordering

- 1. Transaction T issues a `write_item(X)` operation:
 - If $\text{read_TS}(X) > \text{TS}(T)$ or if $\text{write_TS}(X) > \text{TS}(T)$, then an younger transaction has already read the data item so abort and roll-back T and reject the operation.
 - If the condition in part (a) does not exist, then execute `write_item(X)` of T and set `write_TS(X)` to `TS(T)`.
- 2. Transaction T issues a `read_item(X)` operation:
 - If $\text{write_TS}(X) > \text{TS}(T)$, then an younger transaction has already written to the data item so abort and roll-back T and reject the operation.
 - If $\text{write_TS}(X) \leq \text{TS}(T)$, then execute `read_item(X)` of T and set `read_TS(X)` to the larger of `TS(T)` and the current `read_TS(X)`.

Timestamp based concurrency control algorithm

■ Strict Timestamp Ordering

- 1. Transaction T issues a `write_item(X)` operation:
 - If $TS(T) > \text{read_TS}(X)$, then delay T until the transaction T' that wrote or read X has terminated (committed or aborted).
- 2. Transaction T issues a `read_item(X)` operation:
 - If $TS(T) > \text{write_TS}(X)$, then delay T until the transaction T' that wrote or read X has terminated (committed or aborted).

Timestamp based concurrency control algorithm

■ Thomas's Write Rule

- If $\text{read_TS}(X) > \text{TS}(T)$ then abort and roll-back T and reject the operation.
- If $\text{write_TS}(X) > \text{TS}(T)$, then just ignore the write operation and continue execution. This is because the most recent writes counts in case of two consecutive writes.
- If the conditions given in 1 and 2 above do not occur, then execute $\text{write_item}(X)$ of T and set $\text{write_TS}(X)$ to $\text{TS}(T)$.

Databases Recovery

Purpose of Database Recovery

- To bring the database into the last consistent state, which existed prior to the failure.
- To preserve transaction properties (Atomicity, Consistency, Isolation and Durability).

Example:

- If the system crashes before a fund transfer transaction completes its execution, then either one or both accounts may have incorrect value.
- Thus, the database must be restored to the state before the transaction modified any of the accounts.

Types of Failure

- The database may become unavailable for use due to
 - **Transaction failure:** Transactions may fail because of incorrect input, deadlock, incorrect synchronization.
 - **System failure:** System may fail because of addressing error, application error, operating system fault, RAM failure, etc.
 - **Media failure:** Disk head crash, power disruption, etc.

Transaction Log

- For recovery from any type of failure data values prior to modification (BFIM - BeFore Image) and the new value after modification (AFIM – AFter Image) are required.
- These values and other information is stored in a sequential file called Transaction log. A sample log is given below. Back P and Next P point to the previous and next log records of the same transaction.

T ID	Back P	Next P	Operation	Data item	BFIM	AFIM
T1	0	1	Begin			
T1	1	4	Write	X	X = 100	X = 200
T2	0	8	Begin			
T1	2	5	W	Y	Y = 50	Y = 100
T1	4	7	R	M	M = 200	M = 200
T3	0	9	R	N	N = 400	N = 400
T1	5	nil	End			

Data Update

- **Immediate Update:** As soon as a data item is modified in cache, the disk copy is updated.
- **Deferred Update:** All modified data items in the cache are written either after a transaction ends its execution or after a fixed number of transactions have completed their execution.
- **Shadow update:** The modified version of a data item does not overwrite its disk copy but is written at a separate disk location.
- **In-place update:** The disk version of the data item is overwritten by the cache version.

Data Caching

- Data items to be modified are first stored into database cache by the Cache Manager (CM) and after modification they are flushed (written) to the disk.
- The flushing is controlled by **Modified** and **Pin-Unpin** bits.
 - **Pin-Unpin**: Instructs the operating system not to flush the data item.
 - **Modified**: Indicates the AFIM of the data item.

Transaction Roll-back (Undo) and Roll-Forward (Redo)

- To maintain atomicity, a transaction's operations are redone or undone.
 - **Undo:** Restore all BFIMs on to disk (Remove all AFIMs).
 - **Redo:** Restore all AFIMs on to disk.
- Database recovery is achieved either by performing only Undos or only Redos or by a combination of the two.
- These operations are recorded in the log as they happen.

Database recovery techniques



(a)

T_1	T_2	T_3
read_item(A)		
read_item(D)	read_item(B)	
write_item(D)	write_item(B)	
	read_item(D)	read_item(C)
		write_item(B)
		read_item(A)
		write_item(A)

Figure 19.1

Illustrating cascading rollback (a process that never occurs in strict or cascadeless schedules).

(a) The read and write operations of three transactions.

(b) System log at point of crash. (c) Operations before the crash.

Database recovery techniques



(b)

	A	B	C	D
	30	15	40	20
*	[start_transaction, T_3]			
	[read_item, T_3, C]			
	[write_item, $T_3, B, 15, 12$]		12	
	[start_transaction, T_2]			
**	[read_item, T_2, B]			
	[write_item, $T_2, B, 12, 18$]		18	
	[start_transaction, T_1]			
	[read_item, T_1, A]			
**	[read_item, T_1, D]			
	[write_item, $T_1, D, 20, 25$]			25
	[read_item, T_2, D]			
	[write_item, $T_2, D, 25, 26$]			26
	[read_item, T_3, A]			

← System crash

Figure 19.1

Illustrating cascading rollback (a process that never occurs in strict or cascadeless schedules).

(a) The read and write operations of three transactions.

(b) System log at point of crash. (c) Operations before the crash.

* T_3 is rolled back because it did not reach its commit point.

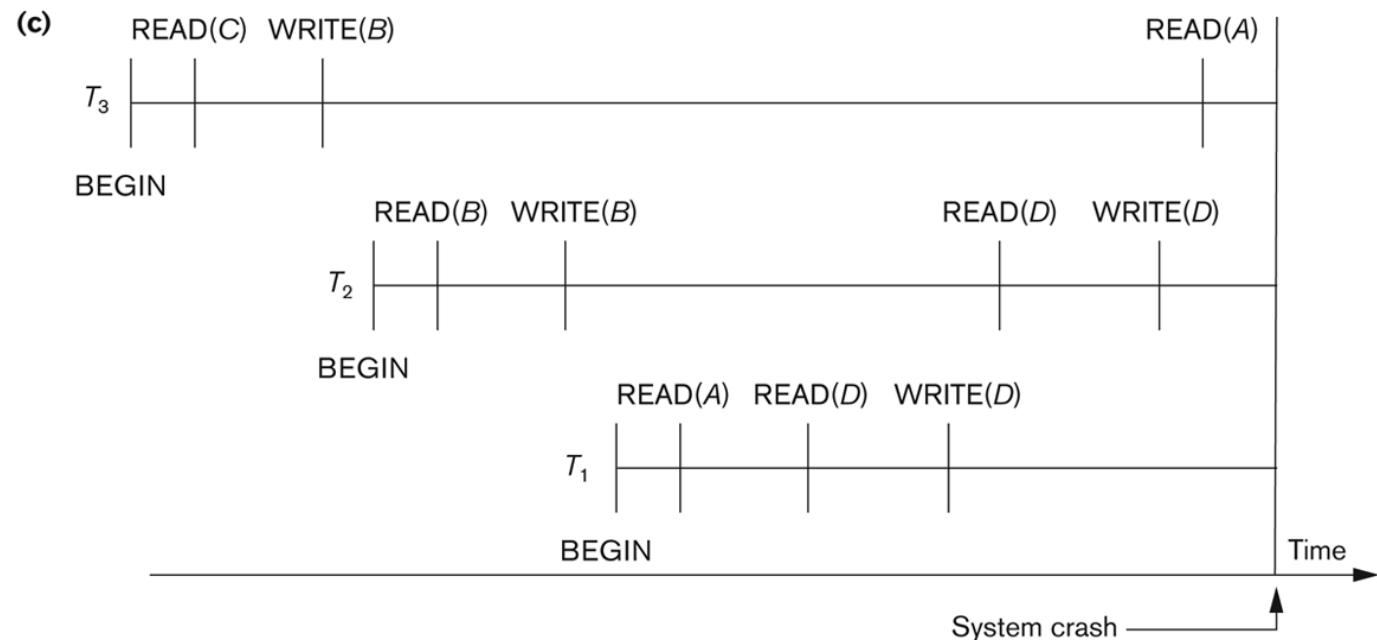
** T_2 is rolled back because it reads the value of item B written by T_3 .

Roll-back: One execution of T1, T2 and T3 as recorded in the log.

Figure 19.1

Illustrating cascading rollback (a process that never occurs in strict or cascadeless schedules).

- (a) The read and write operations of three transactions.
 - (b) System log at point of crash.
 - (c) Operations before the crash.



Write-Ahead Logging

- When **in-place** update (immediate or deferred) is used then log is necessary for recovery and it must be available to recovery manager. This is achieved by **Write-Ahead Logging (WAL)** protocol. WAL states that
 - **For Undo:** Before a data item's AFIM is flushed to the database disk (overwriting the BFIM) its BFIM must be written to the log and the log must be saved on a stable store (log disk).
 - **For Redo:** Before a transaction executes its commit operation, all its AFIMs must be written to the log and the log must be saved on a stable store.

Checkpointing

- Time to time (randomly or under some criteria) the database flushes its buffer to database disk to minimize the task of recovery.
- The following steps defines a checkpoint operation:
 1. Suspend execution of transactions temporarily.
 2. Force write modified buffer data to disk.
 3. Write a [checkpoint] record to the log, save the log to disk.
 4. Resume normal transaction execution.
- During recovery redo or undo is required to transactions appearing after [checkpoint] record.

Steal/No-Steal and Force/No-Force

- Possible ways for flushing database cache to database disk:
 1. Steal: Cache can be flushed before transaction commits.
 2. No-Steal: Cache cannot be flushed before transaction commit.
 3. Force: Cache is immediately flushed (forced) to disk.
 4. No-Force: Cache is deferred until transaction commits
- These give rise to four different ways for handling recovery:
 - Steal/No-Force (Undo/Redo)
 - Steal/Force (Undo/No-redo)
 - No-Steal/No-Force (Redo/No-undo)
 - No-Steal/Force (No-undo/No-redo)

Recovery Scheme

- Deferred Update (No Undo/Redo)
 - The data update goes as follows:
 - A set of transactions records their updates in the log.
 - At commit point under WAL scheme, these updates are saved on database disk.
 - After reboot from a failure the log is used to redo all the transactions affected by this failure.
 - No undo is required because no AFIM is flushed to the disk before a transaction commits.

- Deferred Update in a single-user system
There is no concurrent data sharing in a single user system.
- The data update goes as follows:
 - A set of transactions records their updates in the log.
 - At commit point under WAL scheme these updates are saved on database disk.
- After reboot from a failure the log is used to redo all the transactions affected by this failure.
- No undo is required because no AFIM is flushed to the disk before a transaction commits.

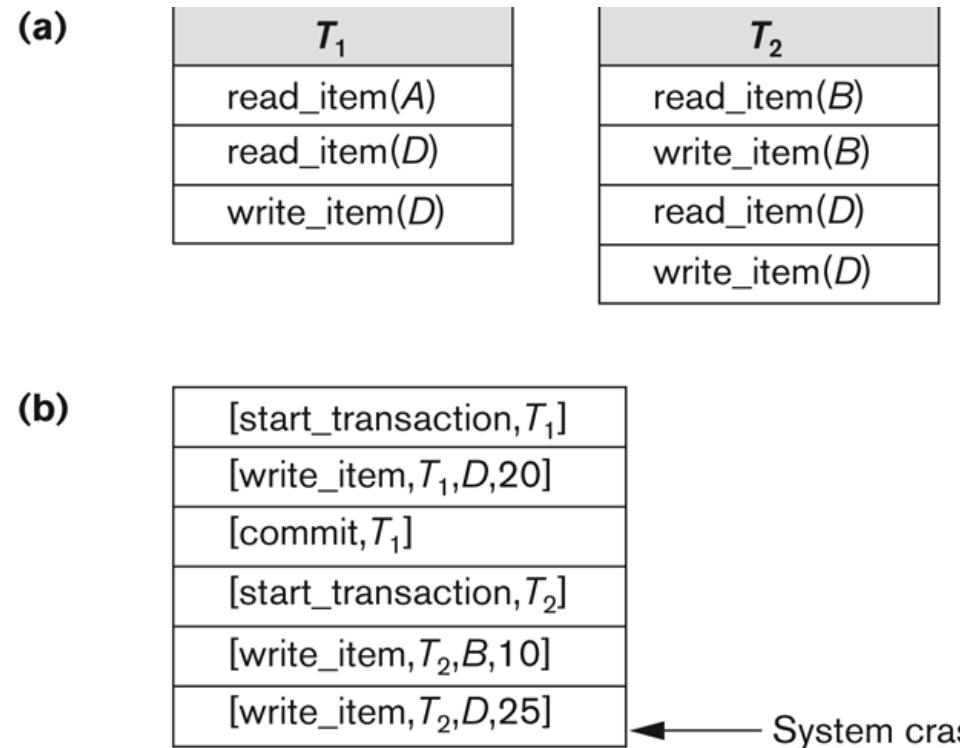


Figure 19.2

An example of recovery using deferred update in a single-user environment. (a) The READ and WRITE operations of two transactions. (b) The system log at the point of crash.

The [write_item,...] operations of T_1 are redone.
 T_2 log entries are ignored by the recovery process.

Deferred Update with concurrent users

- This environment requires some concurrency control mechanism to guarantee **isolation** property of transactions. In a system recovery, transactions which were recorded in the log after the last checkpoint were **redone**. The recovery manager may scan some of the transactions recorded before the checkpoint to get the AFIMs.

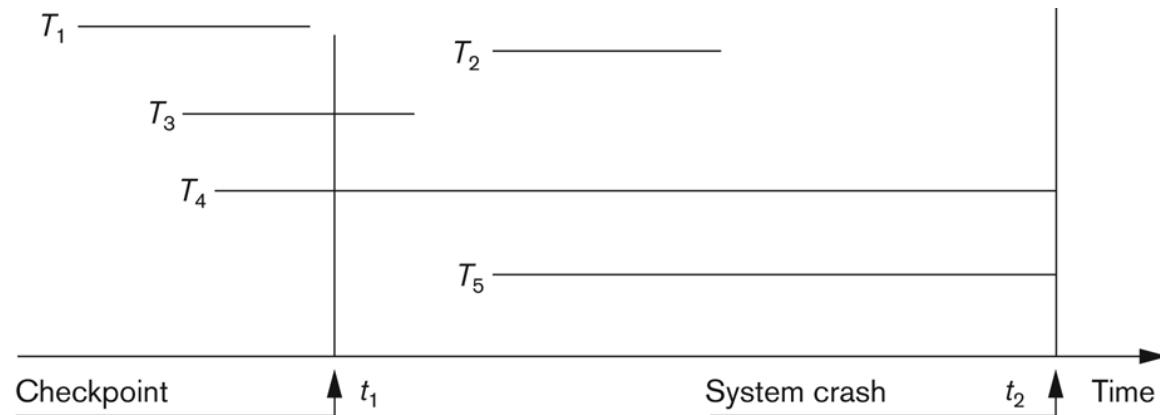


Figure 19.3
An example of recovery in a multi-user environment

(a)

T_1	T_2	T_3	T_4
read_item(A)	read_item(B)	read_item(A)	read_item(B)
read_item(D)	write_item(B)	write_item(A)	write_item(B)
write_item(D)	read_item(D)	read_item(C)	read_item(A)

(b)

[start_transaction, T_1]
[write_item, T_1 , D , 20]
[commit, T_1]
[checkpoint]
[start_transaction, T_4]
[write_item, T_4 , B , 15]
[write_item, T_4 , A , 20]
[commit, T_4]
[start_transaction, T_2]
[write_item, T_2 , B , 12]
[start_transaction, T_3]
[write_item, T_3 , A , 30]
[write_item, T_2 , D , 25]

← System crash

T_2 and T_3 are ignored because they did not reach their commit points.

T_4 is redone because its commit point is after the last system checkpoint.

Figure 19.4

An example of recovery using deferred update with concurrent transactions.

(a) The READ and WRITE operations of four transactions. (b) System log at the point of crash.

Deferred Update with concurrent users

- Two tables are required for implementing this protocol:
 - **Active table:** All active transactions are entered in this table.
 - **Commit table:** Transactions to be committed are entered in this table.
- During recovery, all transactions of the **commit** table are redone and all transactions of **active** tables are ignored since none of their AFIMs reached the database.
- It is possible that a **commit** table transaction may be **redone** twice but this does not create any inconsistency because of a redone is "**idempotent**", that is, one redone for an AFIM is equivalent to multiple redone for the same AFIM.

Recovery Techniques Based on Immediate Update

■ **Undo/No-redo Algorithm**

- In this algorithm AFIMs of a transaction are flushed to the database disk under WAL before it commits.
- For this reason the recovery manager **undoes** all transactions during recovery.
- No transaction is **redone**.
- It is possible that a transaction might have completed execution and ready to commit but this transaction is also **undone**.

Recovery Techniques Based on Immediate Update

■ **Undo/Redo Algorithm (Single-user environment)**

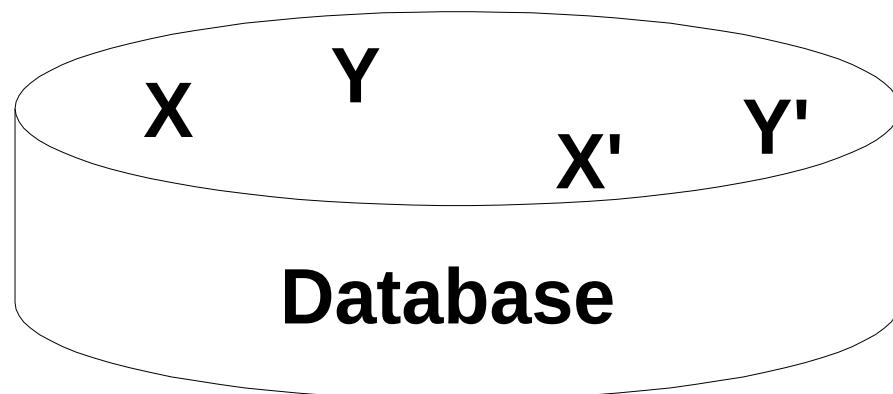
- Recovery schemes of this category apply **undo** and also **redo** for recovery.
- In a single-user environment no concurrency control is required but a log is maintained under WAL.
- Note that at any time there will be one transaction in the system and it will be either in the **commit** table or in the **active** table.
- The recovery manager performs:
 - **Undo** of a transaction if it is in the **active** table.
 - **Redo** of a transaction if it is in the **commit** table.

Recovery Techniques Based on Immediate Update

- **Undo/Redo Algorithm (Concurrent execution)**
- Recovery schemes of this category applies **undo** and also **redo** to recover the database from failure.
- In concurrent execution environment a concurrency control is required and log is maintained under WAL.
- Commit table records transactions to be committed and active table records active transactions.
- To minimize the work of the recovery manager checkpointing is used.
- The recovery performs:
 - **Undo** of a transaction if it is in the **active** table.
 - **Redo** of a transaction if it is in the **commit** table.

Shadow Paging

- The AFIM does not overwrite its BFIM but recorded at another place on the disk. Thus, at any time a data item has AFIM and BFIM (Shadow copy of the data item) at two different places on the disk.



X and Y: Shadow copies of data items
X' and Y': Current copies of data items

Shadow Paging

- To manage access of data items by concurrent transactions two directories (current and shadow) are used.
 - The directory arrangement is illustrated below. Here a page is a data item.

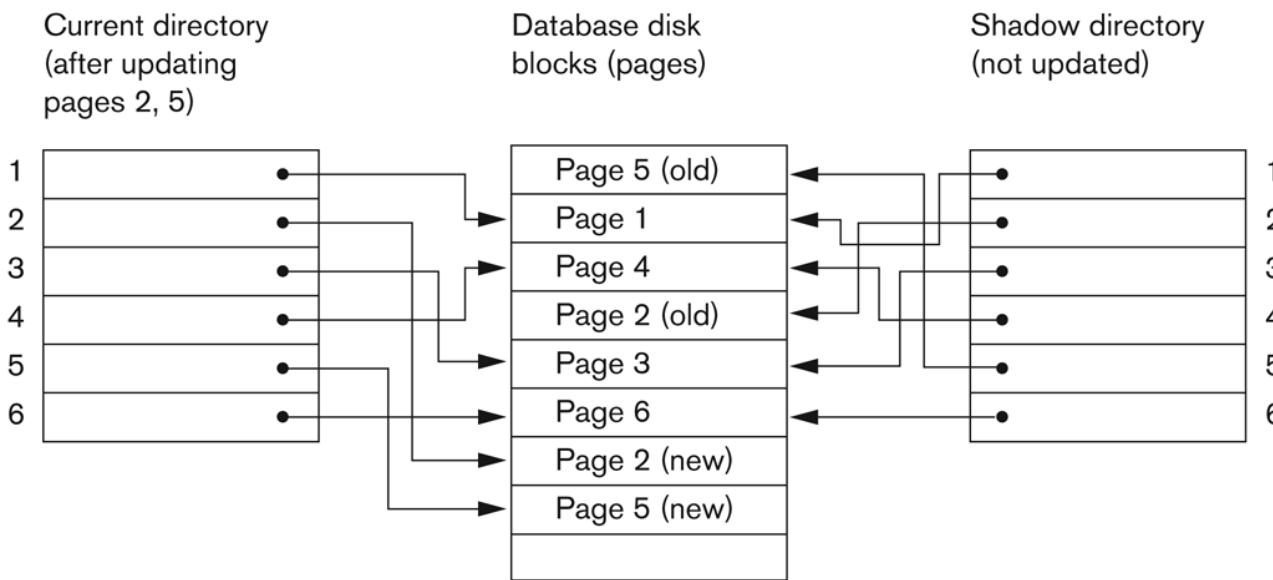


Figure 19.5
An example of shadow paging.



Thank you....



BITS Pilani
Pilani Campus

Database Design





Triggers in SQLite

SQLite Trigger

SQLite **Triggers** are database callback functions, which are automatically performed/invoked when a specified database event occurs. Following are the important points about SQLite triggers -

- SQLite trigger may be specified to fire whenever a DELETE, INSERT or UPDATE of a particular database table occurs or whenever an UPDATE occurs on one or more specified columns of a table.
- At this time, SQLite supports only FOR EACH ROW triggers, not FOR EACH STATEMENT triggers.
- Hence, explicitly specifying FOR EACH ROW is optional.
- Both the WHEN clause and the trigger actions may access elements of the row being inserted, deleted, or updated using references of the form **NEW.column-name** and **OLD.column-name**, where column-name is the name of a column from the table that the trigger is associated with.

SQLite Trigger

- If a WHEN clause is supplied, the SQL statements specified are only executed for rows for which the WHEN clause is true.
- If no WHEN clause is supplied, the SQL statements are executed for all rows.
- The BEFORE or AFTER keyword determines when the trigger actions will be executed relative to the insertion, modification, or removal of the associated row.
- Triggers are automatically dropped when the table that they are associated with is dropped.
- The table to be modified must exist in the same database as the table or view to which the trigger is attached and one must use just **tablename** not **database.tablename**.
- A special SQL function RAISE() may be used within a trigger-program to raise an exception.

When do we need SQLite triggers

An SQLite trigger is a named database object that is executed automatically when an [INSERT](#), [UPDATE](#) or [DELETE](#) statement is issued against the associated table.

You often use triggers to enable sophisticated auditing.

For example, you want to log the changes in the sensitive data such as salary and address whenever it changes.

In addition, you use triggers to enforce complex business rules centrally at the database level and prevent invalid transactions.

SQLite Trigger - Syntax

Following is the basic syntax of creating a **trigger**.

```
CREATE TRIGGER trigger_name [BEFORE|AFTER] event_name ON table_name
BEGIN -- Trigger logic goes here....END;
```

Here, **event_name** could be *INSERT*, *DELETE*, and *UPDATE* database operation on the mentioned table **table_name**.

You can optionally specify **FOR EACH ROW** after table name.

Following is the syntax for creating a trigger on an *UPDATE* operation on one or more specified columns of a table.

```
CREATE TRIGGER trigger_name [BEFORE|AFTER] UPDATE OF column_name ON
table_nameBEGIN -- Trigger logic goes here....END;
```

SQLite CREATE TRIGGER statement



To create a new trigger in SQLite, you use the CREATE TRIGGER statement as follows:

```
CREATE TRIGGER [IF NOT EXISTS] trigger_name
[BEFORE | AFTER | INSTEAD OF] [INSERT | UPDATE | DELETE]
ON table_name
[WHEN condition]
BEGIN
statements;
END;
```

SQLite CREATE TRIGGER statement



In this syntax:

- First, specify the name of the trigger after the CREATE TRIGGER keywords.
- Next, determine when the trigger is fired such as BEFORE, AFTER, or **INSTEAD OF**. You can create BEFORE and AFTER triggers on a table. However, you can only create an **INSTEAD OF** trigger on a view.
- Then, specify the event that causes the trigger to be invoked such as INSERT, UPDATE, or DELETE.
- After that, indicate the table to which the trigger belongs.
- Finally, place the trigger logic in the BEGIN END block, which can be any valid SQL statements.

SQLite CREATE TRIGGER statement

If you combine the time when the trigger is fired and the event that causes the trigger to be fired, you have a total of 9 possibilities:

- BEFORE INSERT
- AFTER INSERT
- BEFORE UPDATE
- AFTER UPDATE
- BEFORE DELETE
- AFTER DELETE
- INSTEAD OF INSERT
- INSTEAD OF DELETE
- INSTEAD OF UPDATE

SQLite - Trigger

Suppose you use a UPDATE statement to update 10 rows in a table, the trigger that associated with the table is fired 10 times. This trigger is called FOR EACH ROW trigger. If the trigger associated with the table is fired one time, we call this trigger a FOR EACH STATEMENT trigger.

As of version 3.9.2, SQLite only supports FOR EACH ROW triggers. It has not yet supported the FOR EACH STATEMENT triggers.

If you use a condition in the WHEN clause, the trigger is only invoked when the condition is true. In case you omit the WHEN clause, the trigger is executed for all rows.

Notice that if you [drop a table](#), all associated triggers are also deleted. However, if the trigger references other tables, the trigger is not removed or changed if other tables are removed or updated.

For example, a trigger references to a table named people, you drop the people table or rename it, you need to manually change the definition of the trigger.

SQLite - Trigger

You can access the data of the row being inserted, deleted, or updated using the OLD and NEW references in the form: OLD.column_name and NEW.column_name. the OLD and NEW references are available depending on the event that causes the trigger to be fired.

The following table illustrates the rules.:

Action	Reference
INSERT	NEW is available
UPDATE	Both NEW and OLD are available
DELETE	OLD is available

SQLite Trigger - Examples

Let us consider a case where we want to keep audit trail for every record being inserted in COMPANY table, which we create newly as follows (Drop COMPANY table if you already have it).

```
sqlite> CREATE TABLE COMPANY( ID INT PRIMARY KEY NOT NULL, NAME TEXT NOT NULL, AGE INT NOT NULL, ADDRESS CHAR(50), SALARY REAL);
```

To keep audit trial, we will create a new table called AUDIT where the log messages will be inserted, whenever there is an entry in COMPANY table for a new record.

```
sqlite> CREATE TABLE AUDIT( EMP_ID INT NOT NULL, ENTRY_DATE TEXT NOT NULL);
```

Here, ID is the AUDIT record ID, and EMP_ID is the ID which will come from COMPANY table and DATE will keep timestamp when the record will be created in COMPANY table.

SQLite Trigger - Examples

Now let's create a trigger on COMPANY table as follows -

```
sqlite> CREATE TRIGGER audit_log AFTER INSERT ON COMPANY BEGIN INSERT INTO AUDIT(EMP_ID,  
ENTRY_DATE) VALUES (new.ID, datetime('now'));END;
```

Now, we will start actual work, Let's start inserting record in COMPANY table which should result in creating an audit log record in AUDIT table. Create one record in COMPANY table as follows -

```
sqlite> INSERT INTO COMPANY (ID,NAME,AGE,ADDRESS,SALARY)VALUES (1, 'Paul', 32, 'California',  
20000.0);
```

This will create one record in COMPANY table, which is as follows -

ID	NAME	AGE	ADDRESS	SALARY
----	------	-----	---------	--------

1	Paul	32	California	20000.0
---	------	----	------------	---------

Same time, one record will be created in AUDIT table. This record is the result of a trigger, which we have created on INSERT operation in COMPANY table.

Similarly, you can create your triggers on UPDATE and DELETE operations based on your requirements.

EMP_ID	ENTRY_DATE
--------	------------

1	2013-04-05 06:26:00
---	---------------------

SQLite Trigger - Listing Triggers

You can list down all the triggers from **sqlite_master** table as follows -

```
sqlite> SELECT name FROM sqlite_master WHERE type = 'trigger';
```

The above SQLite statement will list down only one entry as follows -

```
name-----audit_log
```

If you want to list down triggers on a particular table, then use AND clause with table name as follows -

```
sqlite> SELECT name FROM sqlite_master WHERE type = 'trigger' AND tbl_name = 'COMPANY';
```

The above SQLite statement will also list down only one entry as follows -

```
name-----audit_log
```

Dropping Triggers

Following is the DROP command, which can be used to drop an existing trigger.

```
sqlite> DROP TRIGGER trigger_name;
```

SQLite Trigger - Examples

Let's [create a new table](#) called leads to store all business leads of the company.

```
CREATE TABLE leads (  
    id integer PRIMARY KEY,  
    first_name text NOT NULL,  
    last_name text NOT NULL,  
    phone text NOT NULL,  
    email text NOT NULL,  
    source text NOT NULL  
);
```

SOLite BEFORE INSERT trigger example



1) SQLite BEFORE INSERT trigger example

Suppose you want to validate the email address before inserting a new lead into the leads table. In this case, you can use a BEFORE INSERT trigger.

First, create a BEFORE INSERT trigger as follows:

```
CREATE TRIGGER validate_email_before_insert_leads
```

```
    BEFORE INSERT ON leads
```

```
    BEGIN
```

```
    SELECT
```

```
    CASE
```

```
    WHEN NEW.email NOT LIKE '%_@_%._%' THEN
```

```
        RAISE (ABORT,'Invalid email address')
```

```
    END; END;
```

SOLite BEFORE INSERT trigger example



We used the NEW reference to access the email column of the row that is being inserted.

To validate the email, we used the **LIKE** operator to determine whether the email is valid or not based on the email pattern. If the email is not valid, the RAISE function aborts the insert and issues an error message.

Second, insert a row with an invalid email into the leads table.

```
INSERT INTO leads (id,first_name,last_name,email,phone,source)
```

```
VALUES(1,'John','Doe','jjj','4089009334','New Your');
```

SQLite issued an error: "Invalid email address" and aborted the execution of the insert.

SOLite BEFORE INSERT trigger example



Third, insert a row with a valid email.

```
INSERT INTO leads (first_name, last_name, email, phone)
```

```
VALUES ('John', 'Doe', 'john.doe@sqlitetutorial.net', '4089009334');
```

Because the email is valid, the insert statement executed successfully.

```
SELECT first_name, last_name, email, phone FROM leads;
```

	first_name	last_name	email	phone
▶	John	Doe	john.doe@sqlitetutorial.net	4089009334

SQLite AFTER UPDATE trigger example

SQLite AFTER UPDATE trigger example

The phones and emails of the leads are so important that you can't afford to lose this information.

For example, someone accidentally updates the email or phone to the wrong ones or even delete it.

To protect this valuable data, you use a trigger to log all changes which are made to the phone and email.

First, [create a new table](#) called lead_logs to store the historical data.

```
CREATE TABLE lead_logs (id INTEGER PRIMARY KEY, old_id int, new_id int, old_phone text,  
new_phone text,  
old_email text, new_email text, user_action text, created_at text);
```

SQLite AFTER UPDATE trigger example

Second, create an AFTER UPDATE trigger to log data to the lead_logs table whenever there is an update in the email or phone column.

```
CREATE TRIGGER log_contact_after_update AFTER UPDATE ON leads WHEN old.phone <> new.phone
```

```
OR old.email <> new.email
```

```
BEGIN
```

```
    INSERT INTO lead_logs (old_id,new_id,old_phone, new_phone,old_email,new_email,user_action,  
    created_at )VALUES (old.id,new.id,old.phone,new.phone, old.email,  
    new.email, 'UPDATE',  
    DATETIME('NOW'))
```

```
);
```

```
END;
```

SQLite AFTER UPDATE trigger example

You notice that in the condition in the WHEN clause specifies that the trigger is invoked only when there is a change in either email or phone column.

Third, update the last name of John from Doe to Smith.

```
UPDATE leads SET last_name = 'Smith' WHERE id = 1;
```

The trigger log_contact_after_update was not invoked because there was no change in email or phone.

Fourth, update both email and phone of John to the new ones.

```
UPDATE leads SET phone = '4089998888', email = 'john.smith@sqlitetutorial.net' WHERE id = 1;
```

SQLite AFTER UPDATE trigger example

If you check the log table, you will see there is a new entry there.

```
SELECT old_phone, new_phone, old_email, new_email, user_action FROM lead_logs;
```

old_phone	new_phone	old_email	new_email	user_action
► 4089009334	4089998888	john.doe@sqlitetutorial.	john.smith@sqlitetutorial.net	UPDATE

Exercise

Develop the AFTER INSERT and AFTER DELETE triggers to log the data in the lead_logs table as an exercise.

SQLite Trigger – Example 3

Assuming that customer records are stored in the "customers" table, and that order records are stored in the "orders" table, the following UPDATE trigger ensures that all associated orders are redirected when a customer changes his or her address:

```
CREATE TRIGGER update_customer_address UPDATE OF address ON customers
BEGIN
    UPDATE orders SET address = new.address WHERE customer_name = old.name;
END;
```

With this trigger installed, executing the statement:

```
UPDATE customers SET address = '1 Main St.' WHERE name = 'Jack Jones';
```

causes the following to be automatically executed:

```
UPDATE orders SET address = '1 Main St.' WHERE customer_name = 'Jack Jones';
```

SQLite Trigger – Example 3

For an example of an INSTEAD OF trigger, consider the following schema:

```
CREATE TABLE customer(
    cust_id INTEGER PRIMARY KEY,
    cust_name TEXT,
    cust_addr TEXT
);

CREATE VIEW customer_address AS SELECT cust_id, cust_addr FROM customer;

CREATE TRIGGER cust_addr_chng
INSTEAD OF UPDATE OF cust_addr ON customer_address
BEGIN
    UPDATE customer SET cust_addr=NEW.cust_addr
    WHERE cust_id=NEW.cust_id;  END;
```

SQLite Trigger – Example 3

With the schema above, a statement of the form:

```
UPDATE customer_address SET cust_addr=$new_address WHERE cust_id=$cust_id;
```

```
UPDATE customer_address SET cust_addr='Mumbai' WHERE cust_id=1;
```

Causes the customer.cust_addr field to be updated for a specific customer entry that has customer.cust_id equal to the \$cust_id parameter.

Note how the values assigned to the view are made available as field in the special "NEW" table within the trigger body.



BITS Pilani
Pilani Campus

SQLite3 - Java



Java Applications – Java Database connectivity

Installation - SQLite3

- **Step 1** - Go to [SQLite download page](#), and download precompiled binaries from Windows section.
- **Step 2** - Download sqlite-shell-win32-* .zip and sqlite-dll-win32-* .zip zipped files.
- **Step 3** - Create a folder C:\>sqlite and unzip above two zipped files in this folder, which will give you sqlite3.def, sqlite3.dll and sqlite3.exe files.
- **Step 4** - Add C:\>sqlite in your PATH environment variable and finally go to the command prompt and issue sqlite3 command, which should display the following result.
- C:\>sqlite3
- SQLite version 3.7.15.2 2013-01-09 11:53:05
- Enter ".help" for instructions
- Enter SQL statements terminated with a ";"
- sqlite>

Installation – sqlite-jdbc driver

- Download latest version of *sqlite-jdbc-(VERSION).jar* from [sqlite-jdbc](#) repository.
- Add downloaded jar file *sqlite-jdbc-(VERSION).jar* in your class path, or you can use it along with -classpath option as explained in the following examples.

Connect to database

```
import java.sql.*;  
public class SQLiteJDBC {  
    public static void main( String args[] ) {  
        Connection c = null;  
        try {  
            Class.forName("org.sqlite.JDBC");  
            c = DriverManager.getConnection("jdbc:sqlite:Lab2.db");  
        } catch ( Exception e ) {  
            System.err.println( e.getClass().getName() + ": " + e.getMessage() );  
            System.exit(0);  
        }  
        System.out.println("Opened database successfully");  
    } }
```

Compile and execute as follows:

```
C:>javac SQLiteJDBC.java  
C:>java -classpath ".;sqlite-jdbc-3.27.2.1.jar" SQLiteJDBC
```

Create Table

```
import java.sql.*;  
public class CreateTableCompany {  
    public static void main( String args[] ) {  
        Connection c = null;  
        Statement stmt = null;  
        try {  
            Class.forName("org.sqlite.JDBC");  
            c = DriverManager.getConnection("jdbc:sqlite:Lab2.db");  
            System.out.println("Opened database successfully");  
            stmt = c.createStatement();  
            String sql = "CREATE TABLE COMPANYNew " +  
                "(ID INT PRIMARY KEY    NOT NULL," +  
                " NAME      TEXT    NOT NULL," +  
                " AGE       INT     NOT NULL," +  
                " ADDRESS    CHAR(50), " +  
                " SALARY     REAL)";  
            stmt.executeUpdate(sql);  
            stmt.close();  
            c.close();  
        } catch ( Exception e ) {  
            System.err.println( e.getClass().getName()  
                + ":" + e.getMessage() );  
            System.exit(0);  
        }  
        System.out.println("Table created  
successfully");  
    }  
}
```

Insert data into the created table

```
import java.sql.*;  
public class InsertIntoCompany {  
    public static void main( String args[] ) {  
        Connection c = null;  
        Statement stmt = null;  
        try {  
            Class.forName("org.sqlite.JDBC");  
            c = DriverManager.getConnection("jdbc:sqlite:Lab2.db");  
            c.setAutoCommit(false);  
            System.out.println("Opened database successfully");  
            stmt = c.createStatement();  
            String sql = "INSERT INTO COMPANYNew (ID,NAME,AGE,ADDRESS,SALARY) " +  
                "VALUES (1, 'Paul', 32, 'California', 20000.00 );";  
            stmt.executeUpdate(sql);  
        } catch (Exception e) {  
            e.printStackTrace();  
        }  
    }  
}
```

Insert data into the created table

```
sql = "INSERT INTO COMPANYNew (ID,NAME,AGE,ADDRESS,SALARY) " +"VALUES (2, 'Allen', 25, 'Texas',  
15000.00 );";  
stmt.executeUpdate(sql);  
sql = "INSERT INTO COMPANYNew (ID,NAME,AGE,ADDRESS,SALARY) " + "VALUES (3, 'Teddy', 23,  
'Norway', 20000.00 );";  
stmt.executeUpdate(sql);  
sql = "INSERT INTO COMPANYNew (ID,NAME,AGE,ADDRESS,SALARY) " + "VALUES (4, 'Mark', 25, 'Rich-  
Mond ', 65000.00 );";  
stmt.executeUpdate(sql);  
stmt.close();  
c.commit();  
c.close();  
} catch ( Exception e ) {  
System.err.println( e.getClass().getName() + ": " + e.getMessage() );  
System.exit(0);  
}  
System.out.println("Records created successfully"); } }
```

Fetch data from created table

```
import java.sql.*;  
public class SelectCompany {  
    public static void main( String args[] ) {  
        Connection c = null;  
        Statement stmt = null;  
        try {  
            Class.forName("org.sqlite.JDBC");  
            c = DriverManager.getConnection("jdbc:sqlite:test.db");  
            c.setAutoCommit(false);  
            System.out.println("Opened database successfully");  
            stmt = c.createStatement();  
            ResultSet rs = stmt.executeQuery( "SELECT * FROM COMPANYNew;" );
```

Fetch data from created table

```
while ( rs.next() ) {  
    int id = rs.getInt("id");  
    String name = rs.getString("name");  
    int age = rs.getInt("age");  
    String address = rs.getString("address");  
    float salary = rs.getFloat("salary");  
    System.out.println( "ID = " + id );  
    System.out.println( "NAME = " + name );  
    System.out.println( "AGE = " + age );  
    System.out.println( "ADDRESS = " + address );  
    System.out.println( "SALARY = " + salary );  
    System.out.println();  
}  
}
```

Fetch data from created table

```
rs.close();
stmt.close();
c.close();
} catch ( Exception e ) {
    System.err.println( e.getClass().getName() + ": " + e.getMessage() );
    System.exit(0);
}
System.out.println("Operation done successfully");
}
```



Thank you



Database Systems and Applications

BITS Pilani

Pilani | Dubai | Goa | Hyderabad



BITS Pilani

Pilani | Dubai | Goa | Hyderabad

Contact Session 14:

**Database Security, Access Control Mechanisms, to
SQL Query Processing**

Objectives:

- To understand Database Security with Access control schemes;
- To understand basics of SQL query optimization
- To understand optimization algorithms to perform certain operations.

Contents:

- Introduction to Database Security
- Access Control Mechanisms
- Introduction to SQL Query Processing
- Algorithms for Sorting and Joining
- Algorithms for SELECT, PROJECT and Set operations
- Algorithms for Aggregation and Grouping
- query optimization,

Let's have a quick recap!!

- Concurrency control schemes
- Implementing serializability with locks and timestamps;
- Deadlocks with suitable examples

Main Types of Database Security:



- Enforce security of portions of a database against unauthorized access
 - Database Security and Authorization Subsystem
- Prevent unauthorized persons from accessing the system itself
 - Access Control
- Control the access to databases
 - Database Security
- Protect sensitive data that is being transmitted via some type of communications
 - Data Encryption

Database Security and Authorization Subsystem



- **Discretionary Security Mechanisms**
 - concerned with defining, modeling, and enforcing access to information
- **Mandatory Security Mechanisms** for Multilevel Security
 - requires that data items and users are assigned to certain security labels.

Mandatory access control (MAC) is a model of access control where the operating system provides users with access based on data confidentiality and user clearance levels. In this model, access is granted on a need to know basis: users have to prove a need for information before gaining access.

MAC is considered the most secure of all access control models. Access rules in this model are manually defined by system administrators and strictly enforced by the operating system or security kernel. Regular users can't alter security attributes even for data they've created.

Database Security and Authorization Subsystem



Discretionary access control is an identity-based access control model that provides users with a certain amount of control over their data.

Data owners (document creators or any users authorized to control data) can define access permissions for specific users or groups of users.

In other words, whom to give access to and what privileges to grant are decided at the resource owner's discretion.

Access permissions for each piece of data are stored in an access control list (ACL).

An administrator creates this list when a user grants access to somebody.

The list can be generated automatically.

An ACL includes users and groups that may access data and the levels of access they have.

A system administrator can also enforce an ACL. In this case, the ACL acts as a security policy, and regular users can't edit or overrule it.

Mandatory Access Control



Elements:

OBJECTS

(file, database, port, etc.)



CLASSIFICATIONS

--class(object)--

SUBJECTS

(Users)



CLEARANCE

--clear(subject)--



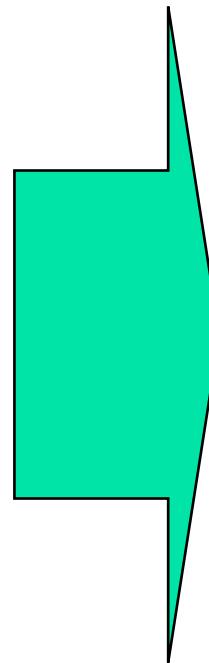
Levels: Top Secret, Secret, Confidential, Unclassified

Mandatory Access Control



Rules:

- Simple Property:
subject s is allowed to read data item d if $\text{clear}(s) \geq \text{class}(d)$
- $*$ -property:
subject s is allowed to write data item d if $\text{clear}(s) \leq \text{class}(d)$
(Less intuitive / reasonable rule)



- Simple Property protects information from **unauthorized access**
- $*$ -property protects data from **contamination or unauthorized modification**

Preventing SQL Injection



- SQL injection is a **code injection technique that might destroy your database**. SQL injection is one of the most common web hacking techniques.

SQL Injection attacks (or SQLi) alter SQL queries, injecting malicious code by exploiting application vulnerabilities.

Successful SQLi attacks allow attackers to modify database information, access sensitive data, execute admin tasks on the database, and recover files from the system.

In some cases attackers can issue commands to the underlying database operating system.

- The severe impact of these attacks makes it critical for developers to adopt practices that prevent SQL injection, such as parameterized queries, stored procedures, and rigorous input validation.

Preventing SQL Injection



- To prevent SQL injection attacks, use prepared statements (instead of creating query strings from input parameters)
 - ```
PreparedStatement pstmt= conn.prepareStatement("select balance from account where account_number =?");
pstmt.setString(1,acct_number);
pstmt.execute();
```

    - (assume that conn is an already open connection to the database)
- Alternatives:
  - use stored procedures
  - use a function that removes special characters (such as quotes) from strings
- Refer for SQL injections:  
<https://portswigger.net/web-security/sql-injection#:~:text=Some%20common%20SQL%20injection%20examples,data%20from%20different%20database%20tables>

- Have password to database, can update anything!
  - Digital signatures by end users can help in some situations.
  - A digital signature confirms that any macros, code modules, and other executable components in the database originated with the signer and that no one has altered them since the database was signed. This helps people who use the database decide whether to trust it and its content.
    - E.g. low update rate data such as land records, birth/death data
- Application program has database password
  - Seize control of the application program → can do anything to the database
  - Solution:
    - Don't give database password to development team
    - keep password in a configuration file on live server, accessible to only a few system administrators
- Ongoing research on trusted applications
- Applications with special privileges that perform security-related functions are called trusted applications.
  - E.g. OS computes checksum on application to verify corruption
  - Allows file-system access only to trusted applications

# Detecting Corruption



- Audit trails: record of all (update) activity on the database: who did what, when
  - Application-level audit trail
    - Helps detect fraudulent activities by users
    - Independent audit section to check all updates
    - BUT: DBAs can bypass this level
      - E.g. audit trail apparently deleted in New Delhi auto-rickshaw license case by malicious users with DBA access
  - Database level audit trail
    - Database needs to ensure these can't be turned off, and turned on again after doing damage
    - Supported by most commercial database systems
    - But required DBAs with knowledge of application to monitor at this level
  - Keep archival copies and cross check periodically

# Access Control in Application Layer

innovate

achieve

lead

- Authorization in application layer vs. database layer
  - Benefits
    - fine grained authorizations, such as to individual tuples, can be implemented by the application.
    - authorizations based on business logic easier to code at application level
  - Drawback:
    - Authorization must be done in application code, and may be dispersed all over an application
    - Hard to check or modify authorizations
    - Checking for absence of authorization loopholes becomes very difficult since it requires reading large amounts of application code

# Introduction Query optimization

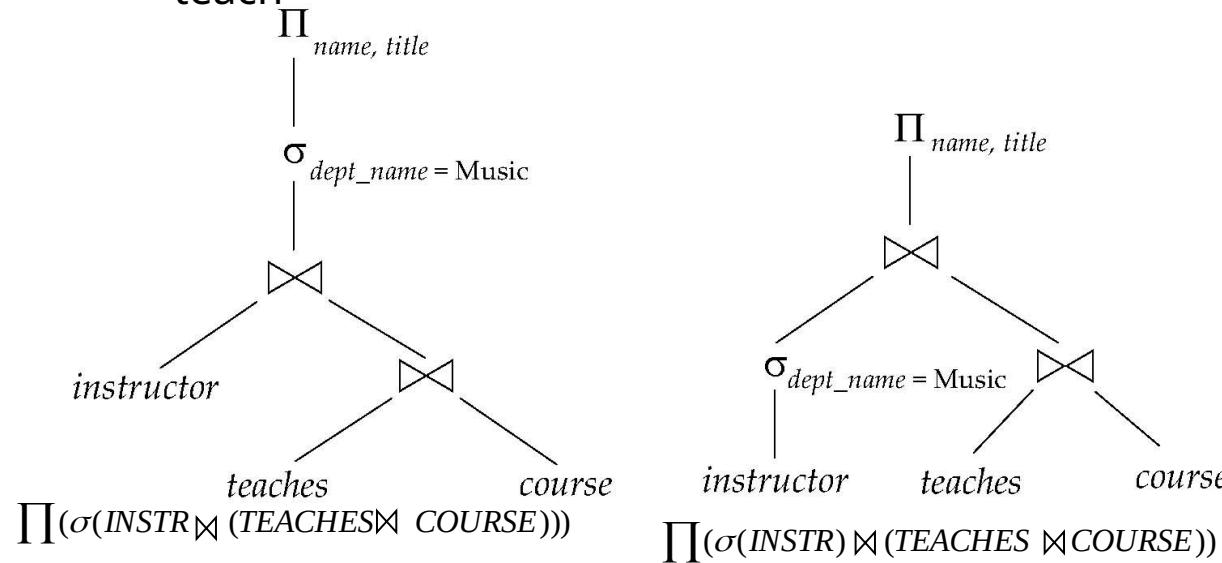


- **Query optimization:** finding the “best” **query execution plan (QEP)** among the **many** possible ones
  - User is not expected to write queries efficiently (DBMS optimizer takes care of that)
- Alternative ways to execute a given query – 2 levels
  - Equivalent relational algebra expressions
  - Different implementation choices for each relational algebra operation
    - Algorithms, indices, coordination between successive operations, ...

INSTR(i\_id, name, dept\_name, ...)  
COURSE(c\_id, title, ...)  
TEACHES(i\_id, c\_id, ...)

```
SELECT I.name, C.title
FROM INSTR I, COURSE C,
TEACHES T
WHERE I.i_id = T.i_id
AND T.c_id = C.c_id
AND dept_name="Music"
```

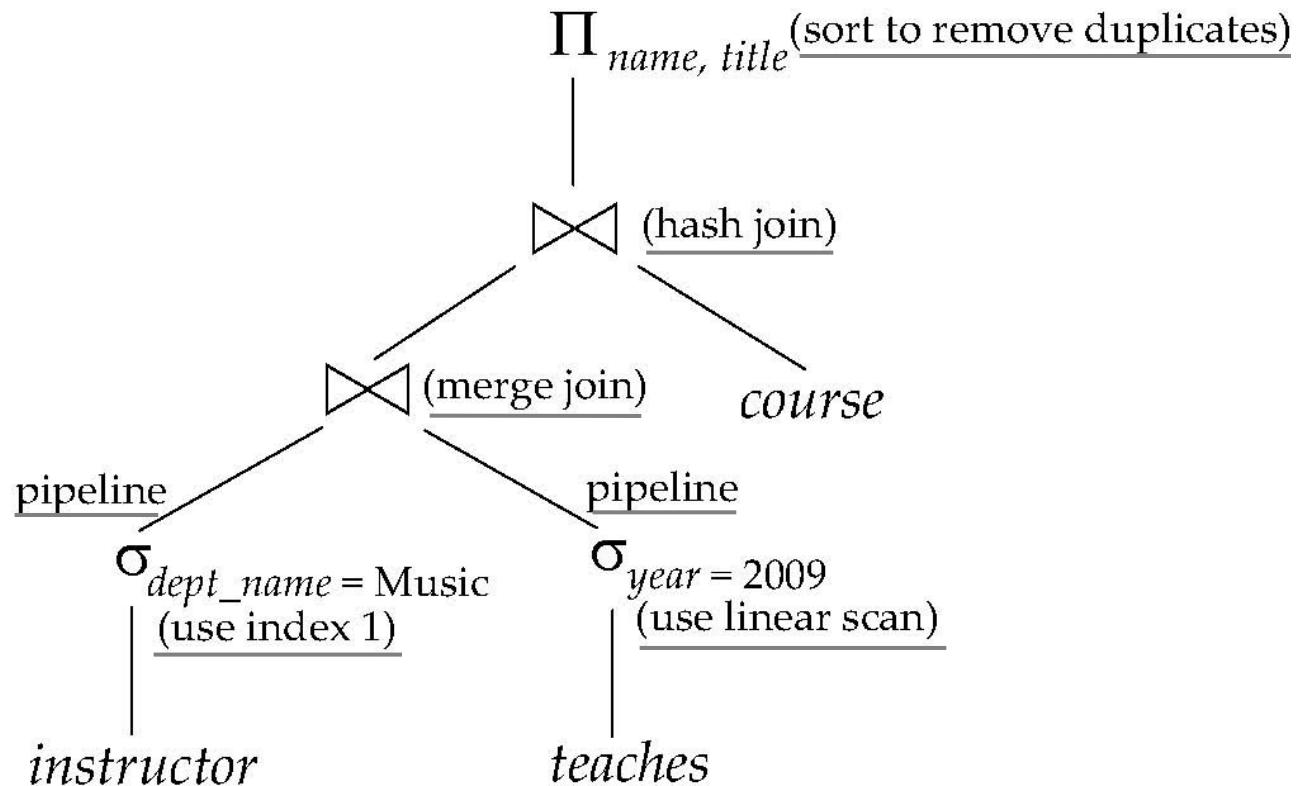
The name of all instructors in the department of Music together with the titles of all courses they teach



# Introduction (Cont.)



- A **query evaluation plan (QEP)** defines exactly what algorithm is used for each operation, and how the execution of the operations is coordinated



- Find out how to view query execution plans on your favorite database. For SQLite follow the link:

[https://www.sqlite.org/eqp.html#:~:text=sqlite%3E%20CREATE%20INDEX%20i2%20ON,COVERING%20INDEX%20i2%20\(a%3D%3F\)&text=The%20order%20of%20the%20entries%20indicates%20the%20nesting%20order](https://www.sqlite.org/eqp.html#:~:text=sqlite%3E%20CREATE%20INDEX%20i2%20ON,COVERING%20INDEX%20i2%20(a%3D%3F)&text=The%20order%20of%20the%20entries%20indicates%20the%20nesting%20order)

# Introduction (Cont.)



- Cost difference between query evaluation plans can be enormous
  - E.g. seconds vs. days in some cases
  - It is worth spending time in finding “best” QEP
- Steps in **cost-based query optimization**
  1. Generate logically equivalent expressions using **equivalence rules**
  2. Annotate in all possible ways resulting expressions to get alternative QEP
  3. Evaluate/estimate the cost (execution time) of each QEP
  4. Choose the cheapest QEP based on **estimated cost**
- Estimation of QEP cost based on:
  - Statistical information about relations (stored in the **Catalog**)
    - number of tuples, number of distinct values for an attribute
  - Statistics estimation for intermediate results
    - to compute cost of complex expressions
  - Cost formulae for algorithms, computed using statistics

# Transformation of Relational Expressions

innovate

achieve

lead

- Two relational algebra expressions are said to be **equivalent** if the two expressions generate the same set of tuples on every *legal* database instance
  - Note: order of tuples is irrelevant (and also order of attributes)
  - We don't care if they generate different results on databases that violate integrity constraints (e.g., uniqueness of keys)
- In SQL, inputs and outputs are multisets (un-ordered) of tuples
  - Two expressions in the multiset version of the relational algebra are said to be equivalent if the two expressions generate the same multiset of tuples on every legal database instance
  - We focus on relational algebra and treat relations as sets
- An **equivalence rule** states that expressions of two forms are equivalent
  - One can replace an expression of first form by one of the second form, or vice versa

# Equivalence Rules



1. Conjunctive selection operations can be deconstructed into a sequence of individual selections.

$$\sigma_{\theta_1 \wedge \theta_2}(E) = \sigma_{\theta_1}(\sigma_{\theta_2}(E))$$

2. Selection operations are commutative.

$$\sigma_{\theta_1}(\sigma_{\theta_2}(E)) = \sigma_{\theta_2}(\sigma_{\theta_1}(E))$$

3. Only the last in a sequence of projection operations is needed, the others can be omitted

$$\Pi_{L_1}(\Pi_{L_2}(\dots(\Pi_{L_n}(E))\dots)) = \Pi_{L_1}(E)$$

where  $L_1 \subseteq L_2 \subseteq \dots \subseteq L_n$

4. Selections can be combined with Cartesian products and theta joins.

a.  $\sigma_{\theta}(E_1 \times E_2) = E_1 \bowtie_{\theta} E_2$

b.  $\sigma_{\theta_1}(E_1 \bowtie_{\theta_2} E_2) = E_1 \bowtie_{\theta_1 \wedge \theta_2} E_2$

# Equivalence Rules (Cont.)



5. Theta-join (and thus natural joins) operations are commutative.  $\bowtie$   $\bowtie$

$$E_1 \underset{\theta}{\bowtie} E_2 = E_2 \underset{\theta}{\bowtie} E_1$$

(but the order is important for efficiency)

6. (a) Natural join operations are associative:

$$(E_1 \underset{\theta}{\bowtie} E_2) \underset{\theta}{\bowtie} E_3 \neq E_1 \underset{\theta}{\bowtie} (E_2 \underset{\theta}{\bowtie} E_3)$$

(again, the order is important for efficiency)

(b) Theta joins are associative in the following manner:

$$\bowtie(E_1 \underset{\theta_1}{\bowtie} E_2) \underset{\theta_2 \wedge \theta_3}{\bowtie} E_3 = E_1 \underset{\theta_1 \wedge \theta_3}{\bowtie} (E_2 \underset{\theta_2}{\bowtie} E_3)$$

where  $\theta_1$  involves attributes from only  $E_1$  and

$E_2$

and  $\theta_2$  involves attributes from only  $E_2$  and

$E_3$

# Equivalence Rules (Cont.)



7. (a) Selection distribute over theta join in the following manner:

$$\sigma_{\theta_1}(E_1 \bowtie_{\theta} E_2) = (\sigma_{\theta_1}(E_1)) \bowtie_{\theta} E_2$$

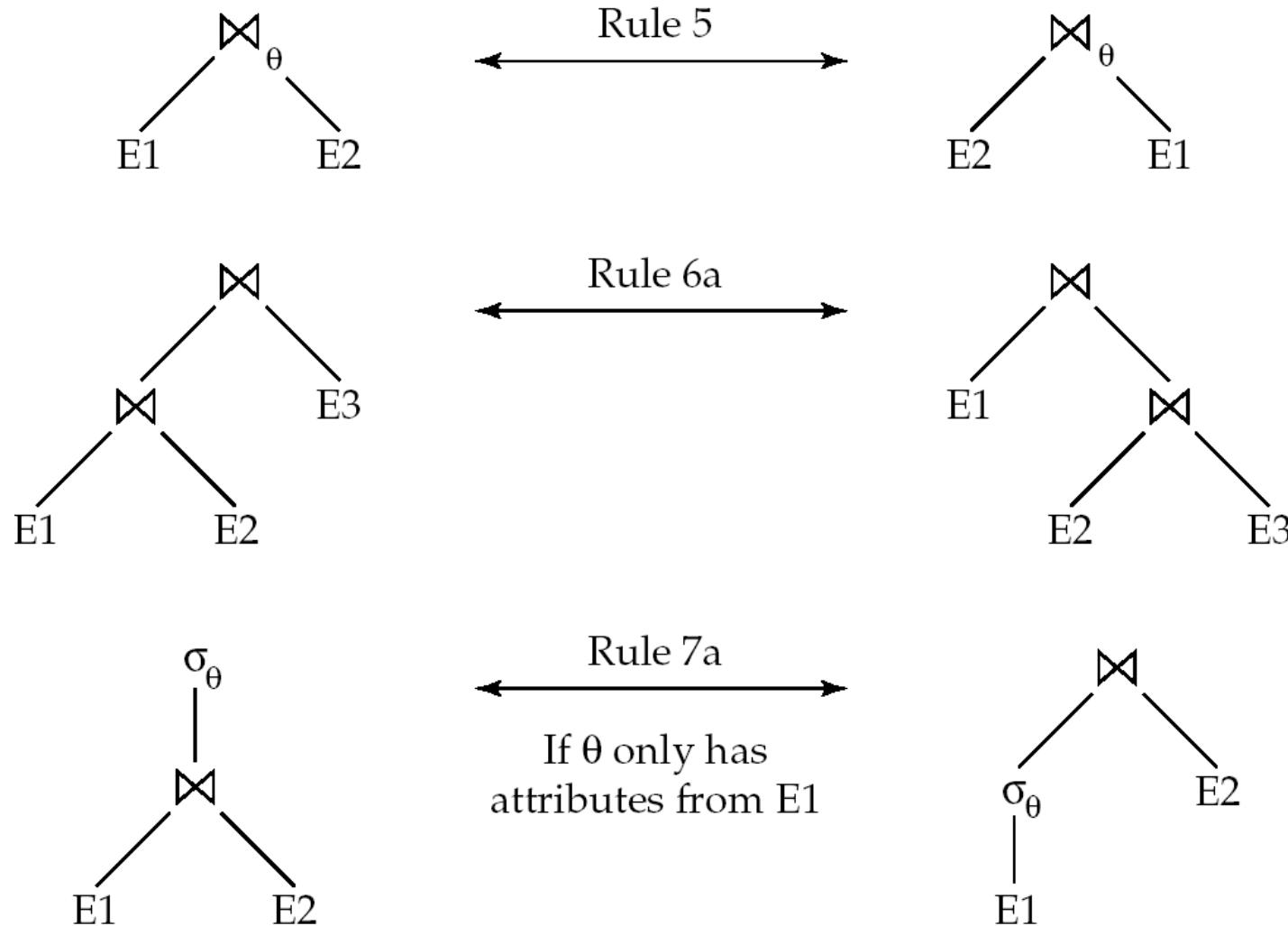
where  $\theta_1$  involves attributes from only  $E_1$

- (b) Complex selection distribute over theta join in the following manner:

$$\sigma_{\theta_1 \wedge \theta_2}(E_1 \bowtie_{\theta} E_2) = (\sigma_{\theta_1}(E_1)) \bowtie_{\theta} (\sigma_{\theta_2}(E_2))$$

where  $\theta_1$  involves attributes from only  $E_1$   
and  $\theta_2$  involves attributes from only  $E_2$

# Pictorial Depiction of Equivalence Rules



# Exercise



## ■ Disprove the equivalence

$$(R \bowtie S) \bowtie T = R \bowtie (S \bowtie T)$$

Definition (**left outer join**): the result of a left outer join  $T \bowtie R$      $S$  is a super-set of the result of the join  $T \bowtie R \bowtie S$  in that all tuples in  $T'$  appear in  $T$ . In addition,  $T$  preserve those tuples that are lost in the join, by creating tuples in  $T$  that are filled with *null* values

| <i>STUD</i> | <b>stud_id</b> | <b>name</b> | <b>surname</b> |
|-------------|----------------|-------------|----------------|
|             | 1              | gino        | bianchi        |
|             | 2              | filippo     | neri           |
|             | 3              | mario       | rossi          |

| <i>TAKES</i> | <b>stud_id</b> | <b>course</b> | <b>grade</b> |
|--------------|----------------|---------------|--------------|
|              | 1              | Math          | 30           |
|              | 2              | DB            | 22           |
|              | 2              | Logic         | 30           |

| <i>STUD</i> $\bowtie$ <i>TAKES</i> | <b>stud_id</b> | <b>name</b> | <b>surname</b> | <b>course</b> | <b>grade</b> |
|------------------------------------|----------------|-------------|----------------|---------------|--------------|
|                                    | 1              | gino        | bianchi        | Math          | 30           |
|                                    | 2              | filippo     | neri           | DB            | 22           |
|                                    | 2              | filippo     | neri           | Logic         | 30           |

*TAKES*  $\bowtie$  *STUD*    **???**

# Solution



- Disprove the equivalence  $(R \bowtie S) \bowtie T = R \bowtie (S \bowtie T)$

| R |       |
|---|-------|
| A | $A_R$ |
| 1 | 1     |

| S |       |
|---|-------|
| A | $A_S$ |
| 2 | 1     |

| T |       |
|---|-------|
| A | $A_T$ |
| 1 | 1     |

$R \bowtie S$

| A | $A_R$ | $A_S$ |
|---|-------|-------|
| 1 | 1     | null  |

$S \bowtie T$

| A | $A_S$ | $A_T$ |
|---|-------|-------|
| 2 | 1     | null  |

$(R \bowtie S) \bowtie T$

| A | $A_R$ | $A_S$ | $A_T$ |
|---|-------|-------|-------|
| 1 | 1     | null  | 1     |

$R \bowtie (S \bowtie T)$

| A | $A_R$ | $A_S$ | $A_T$ |
|---|-------|-------|-------|
| 1 | 1     | null  | null  |

# Equivalence derivability and minimality

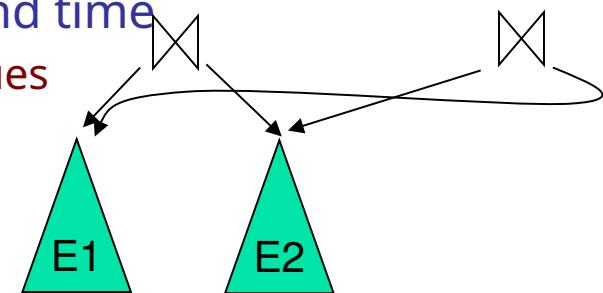


- Some equivalence can be derived from others
  - example: 2 can be obtained from 1 (exploiting commutativity of conjunction)
  - 7b can be obtained from 1 and 7a
- Optimizers use **minimal** sets of equivalence rules

# Enumeration of Equivalent Expressions



- Query optimizers use equivalence rules to **systematically** generate expressions equivalent to the given one
- Can generate all equivalent expressions as follows:
  - Repeat (starting from the set containing only the given expression)
    - apply all applicable equivalence rules on every sub-expression of every equivalent expression found so far
    - add newly generated expressions to the set of equivalent expressions
- Until no new equivalent expressions are generated
- The above approach is very expensive in space and time
  - Space: efficient expression-representation techniques
    - 1 copy is stored for shared sub-expressions
  - Time: partial generation
    - Dynamic programming
    - Greedy techniques (select best choices at each step)
    - Heuristics, e.g., single-relation operations (selections, projections) are pushed inside (performed earlier)



# Cost Estimation



- Cost of each operator computed as described in Chapter 15 <sup>\*</sup>
  - Need statistics of input relations
    - E.g. number of tuples, sizes of tuples
- Statistics are collected in the **Catalog**
- Inputs can be results of sub-expressions
  - Need to estimate statistics of expression results
    - Based on values taken by the attributes
  - Estimation of size of intermediate results
    - Based on # of tuple in input to successive operations
  - Estimation of number of distinct values in intermediate results
    - selectivity rate of successive selection operations i.e. # of select operations
- Statistics are not totally accurate
  - Information in the catalog might be not always up-to-date (delay)
  - A precise estimate for intermediate results might be impossible to compute

# Statistical Information for Cost Estimation

innovate

achieve

lead

- Statistics information is maintained in the **Catalog**
- The catalog is itself stored in the database as relation(s)
- It contains:
  - $n_r$ : number of tuples in a relation  $r$
  - $b_r$ : number of blocks containing tuples of  $r$
  - $l_r$ : size of a tuple of  $r$  (in bytes)
  - $f_r$ : blocking factor of  $r$  – i.e., the number of tuples of  $r$  that fit into one block
  - $V(A, r)$ : number of distinct values that appear in  $r$  for set of attributes  $A$ 
    - $V(A, r) = \prod_A(r)$  – if  $A$  is a key, then  $V(A, r) = n_r$
  - $\min(A, r)$ : smallest value appearing in relation  $r$  for set of attribute  $A$ ;
  - $\max(A, r)$ : largest value appearing in relation  $r$  for set of attribute  $A$ ;
  - statistics about indices (height of  $B^*$ -trees, number of blocks for leaves, ...)
- We assume tuples of  $r$  are stored together physically in a file; then:  $b_r = \lceil n_r / f_r \rceil$
- Information not always up-to-date
  - Catalog is not updated to every DB change (done during periods of light system load)

# Selection Size Estimation



- # of records that will satisfy the selection predicate (aka selection condition)
- $\sigma_{A=v}(r)$  *(we are assuming that v actually is present in A)*
  - $n_r / V(A, r)$
  - 1 if A is key
- $\sigma_{A \leq v}(r)$  *(case  $\sigma_{A \geq v}(r)$  is symmetric)*
  - 0 if  $v < \min(A, r)$
  - $n_r$  if  $v \geq \max(A, r)$
  - $n_r * \frac{v - \min(A, r)}{\max(A, r) - \min(A, r)}$  otherwise
  - In absence of statistical information or when v is unknown at time of cost estimation (e.g., v is computed at run-time by the application using the DB), then we assume
    - $n_r / 2$
- If histograms are available, we can do more precise estimates
  - use values for restricted ranges instead of  $n_r$ ,  $V(A, r)$ ,  $\min(A, r)$ ,  $\max(A, r)$
- A histogram is a graph that shows the frequency of numerical data using rectangles. The height of a rectangle (the vertical axis) represents the distribution frequency of a variable (the amount, or how often that variable appears). The width of the rectangle (horizontal axis) represents the value of the variable (for instance, minutes, years, or ages). Histograms make cost estimates accurately.

# Complex Selection Size Estimation

innovate

achieve

lead

## Conjunction $E = \sigma_{\theta_1 \wedge \theta_2 \wedge \dots \wedge \theta_n}(r)$

- we compute  $s_i$  = \*size selection for  $\theta_i$   $(i = 1, \dots, n)$
- selectivity rate (SR\*\*)** of  $\sigma_{\theta_i}(r)$ :  $SR(\sigma_{\theta_i}(r)) = s_i / n_r$   $(i = 1, \dots, n)$
- $SR(E) = \prod_i (SR(\sigma_{\theta_i}(r))) = s_1 / n_r * \dots * s_n / n_r$   $\prod_i$  is multiplication with  $i = 1, \dots, n$
- # of record for  $E = n_r * SR(E) = n_r * \frac{s_1 * s_2 * \dots * s_n}{(n_r)^n}$

## Disjunction $E = \sigma_{\theta_1 \vee \theta_2 \vee \dots \vee \theta_n}(r) = \sigma_{\neg\theta_1 \wedge \neg\theta_2 \wedge \dots \wedge \neg\theta_n}(r) [$

- $SR(E) = 1 - SR(\sigma_{\neg\theta_1 \wedge \neg\theta_2 \wedge \dots \wedge \neg\theta_n}(r))$
- $SR(\sigma_{\neg\theta_1 \wedge \neg\theta_2 \wedge \dots \wedge \neg\theta_n}(r)) = (1 - s_1 / n_r) * \dots * (1 - s_n / n_r)$
- # of record for  $E = n_r * SR(E) =$

$$n_r * \left[ 1 - \left(1 - \frac{s_1}{n_r}\right) * \left(1 - \frac{s_2}{n_r}\right) * \dots * \left(1 - \frac{s_n}{n_r}\right) \right]$$

## Negation $E = \sigma_{\neg\theta}(r)$

- # of record for  $E = n_r - \# \text{ of record for } \sigma_{\theta}(r)$

\*This is the operation of selecting particular column/s from a table. Hence its size estimate is equal to number of distinct values of column/s present. \*\*The selectivity is the fraction of rows in a table or partition that is chosen by the predicate.

# Join Size Estimation

- # of records that will be included in the result
- (*cartesian product*)  $r \times s$ : 
$$\# \text{ of records} = n_r * n_s$$
- (*natural join on attribute A*)  $r \bowtie s$ :
  - for each tuple  $t_r$  of  $r$  there are in average  $n_s / V(A,s)$  many tuples of  $s$  selected
  - thus, 
$$\# \text{ of records} = n_r * n_s / V(A,s)$$
  - by switching the role of  $r$  and  $s$  we get 
$$\# \text{ of records} = n_r * n_s / V(A,r)$$
  - lowest is more accurate estimation 
$$\# \text{ of records} = n_r * n_s / \max\{ V(A,r), V(A,s) \}$$
  - histograms can be used for more accurate estimations
    - histograms must be on join attributes, for both relations, and with same ranges
    - use values for restricted ranges instead of  $n_r$ ,  $n_s$ ,  $V(A,r)$ ,  $V(A,s)$  and then sum estimations for each range
  - if  $A$  is key for  $r$ , then 
$$\# \text{ of records} \leq n_s$$
 (and vice versa)
    - in addition, if  $A$  is not null in  $s$ , then 
$$\# \text{ of records} = n_s$$
 (and vice versa)
- (*theta join*)  $r \bowtie_\theta s$ 
  - $$r \bowtie_\theta s = \sigma_\theta(r \times s)$$
 use formulas for cartesian product and selection

# Size Estimation for Other Operations

innovate

achieve

lead

- projection (no duplications):  $\# \text{ of records} = V(A, r)$
- aggregation  ${}_G \gamma_F (r)$   $\# \text{ of records} = V(G, r)$
- set operations
  - between selections on same relation
    - es.:  $\sigma_{\theta_1}(r) \cup \sigma_{\theta_2}(r) = \sigma_{\theta_1 \vee \theta_2}(r)$
  - $r \cup s$   $\# \text{ of records} = n_r + n_s$
  - $r \cap s$   $\# \text{ of records} = \min \{ n_r, n_s \}$
  - $r - s$   $\# \text{ of records} = n_r$
- outer join
  - left outer join  $\# \text{ of records} = \# \text{ of records for inner join} + n_r$
  - right outer join  $\# \text{ of records} = \# \text{ of records for inner join} + n_s$
  - full outer join  $\# \text{ of records} = \# \text{ of records for inner join} + n_r + n_s$

# Estimation for Number of Distinct Values

innovate

achieve

lead

- # distinct values in the result for expression  $E$  and attribute (or set of attributes)  $A$ :  $V(A, E)$
- Selection  $E = \sigma_\theta(r)$ 
  - $V(A, E)$  is a specific value for some conditions – e.g.,  $A=3$  or  $3 < A \leq 6$
  - condition  $A < v$  (or  $A > v$ ,  $A \geq v$ , ... ) 
$$V(A, E) = V(A, r) * \text{selectivity rate of the selection}$$
  - otherwise 
$$V(A, E) = \min \{ n_E, V(A, r) \}$$
- Join  $E = r \bowtie s$ 
  - $A$  only contains attributes from  $r$  
$$V(A, E) = \min \{ n_E, V(A, r) \}$$
  - $A$  only contains attributes from  $s$  
$$V(A, E) = \min \{ n_E, V(A, s) \}$$
  - $A$  contains attributes  $A1$  from  $r$  and attributes  $A2$  from  $s$ 
$$V(A, E) = \min \{ n_E, V(A1, r) * V(A2 - A1, s), V(A2, s) * V(A1 - A2, r) \}$$

# Choice of Evaluation Plans



- Must consider the interaction of evaluation techniques when choosing evaluation plans
  - choosing the cheapest algorithm for each operation independently may not yield best overall algorithm. E.g.
    - merge-join may be costlier than hash-join, but may provide a sorted output which reduces the cost for an outer level aggregation
    - nested-loop join may provide opportunity for pipelining
- Practical query optimizers incorporate elements of the following two broad approaches:
  1. Search all the plans and choose the best plan in a cost-based fashion
  2. Uses heuristics (shortcuts) to choose a plan