23IT404: DATA BASE MANAGEMENT SYSTEMS

Mr. V. Anil Kumar
Assistant Professor
Department of IT

What is data? A Historical perspective

A collection of facts from which conclusion may be drawn.

Data is often obtained as a result of recordings or observations.

- **Data is the plural form of datum.**
- **The temperature of the days is data.**

Temperature of the days

When this data is to be collected, a system or person monitors the daily temperatures and records it.

- Finally when it is to be converted into meaningful information, the patterns in the temperatures are analyzed and a conclusion about the temperature is arrived at.
- So information obtained is a result of analysis, communication, or investigation.

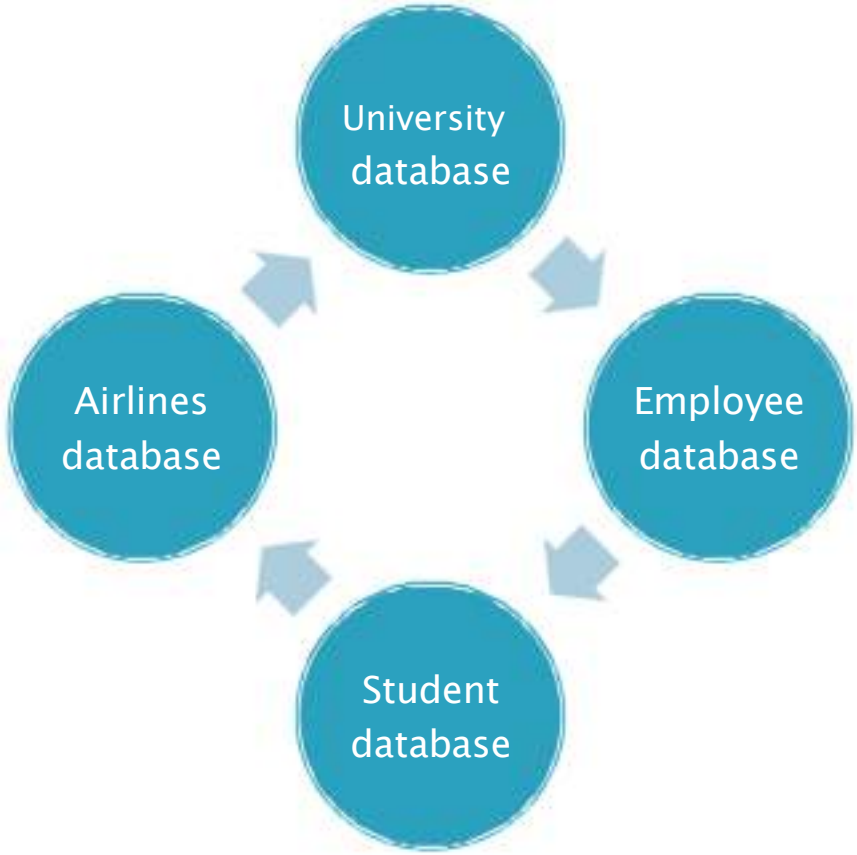
Properties of Data

- Data should be well organized.
- Data should be related.
- Data should be accessible in any order.
- One data should be stored minimum number of times.

What is a Database?

•Database is a collection of related data, that contains information relevant to an enterprise.

For example:



PROPERTIES OF A DATABASE

A database represents some aspect of the real world, sometimes called the mini world or the universe of discourse (UoD).

A database is a logically coherent collection of data with some inherent meaning.

A database is designed, built and populated with data for a specific purpose.

What is Database Management System (DBMS)?

A database management system (DBMS) is a collection of programs that enables users to create & maintain a database. It facilitates the definition, creation and manipulation of the database.

Definition – it holds only structure of database, not the data. It involves specifying the data types, structures & constraints for the data to be stored in the database.

Creation –it is the inputting of actual data in the database. It involves storing the data itself on some storage medium that is controlled by the DBMS.

Manipulation-it includes functions such as updation, insertion, deletion, retrieval of specific data and generating reports from the data.

Typical DBMS Functionality

- Define a database : in terms of data types, structures and constraints
- Construct or Load the Database: on a secondary storage medium
- Manipulating the database : querying, generating reports, insertions, deletions and modifications to its content
- Concurrent Processing and Sharing by a set of users and programs: yet, keeping all data valid and consistent

Typical DBMS Functionality

Other features:

- Protection or Security measures to prevent unauthorized access
- “Active” processing to take internal actions on data Presentation and Visualization of data

Database System

The database and the DBMS together is called the database system.

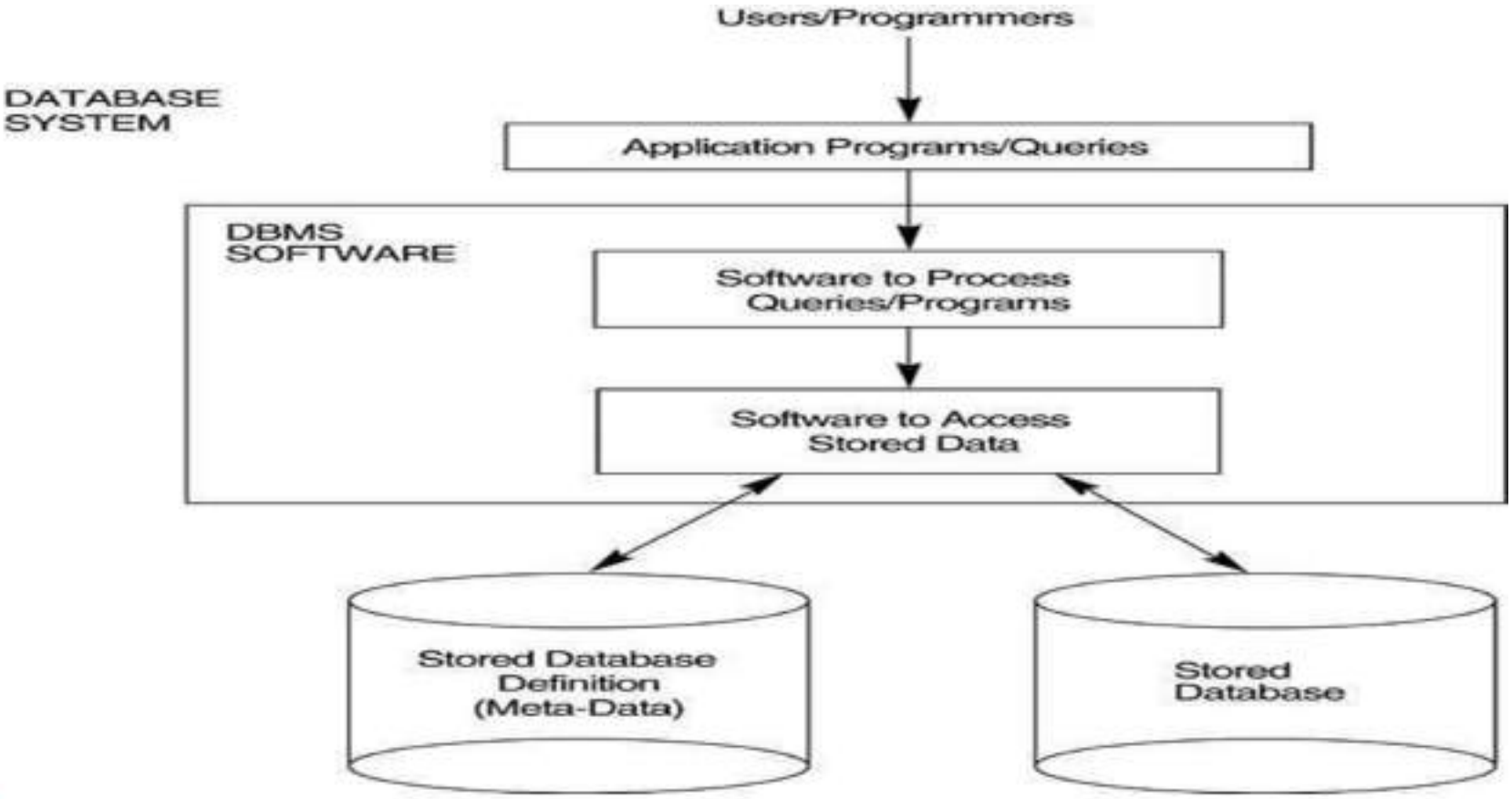
Database systems are designed to manage large bodies of information.

- It involves both defining structures for storage of information & providing mechanisms for the manipulation of information.

Database system must ensure the safety of the information stored.

Meta data- it is the data about the data. It contains the structure of the database as well as the physical location of the database.

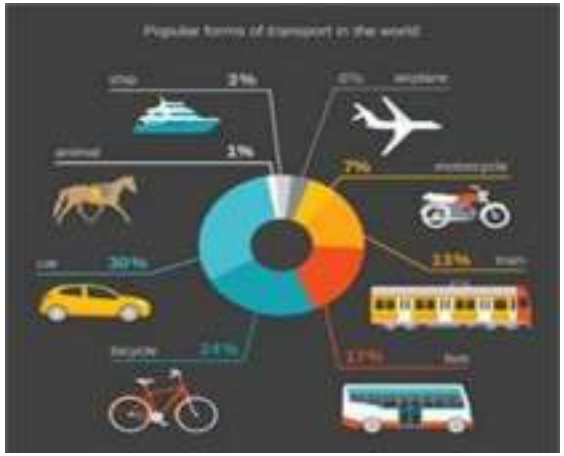
A simplified database system environment



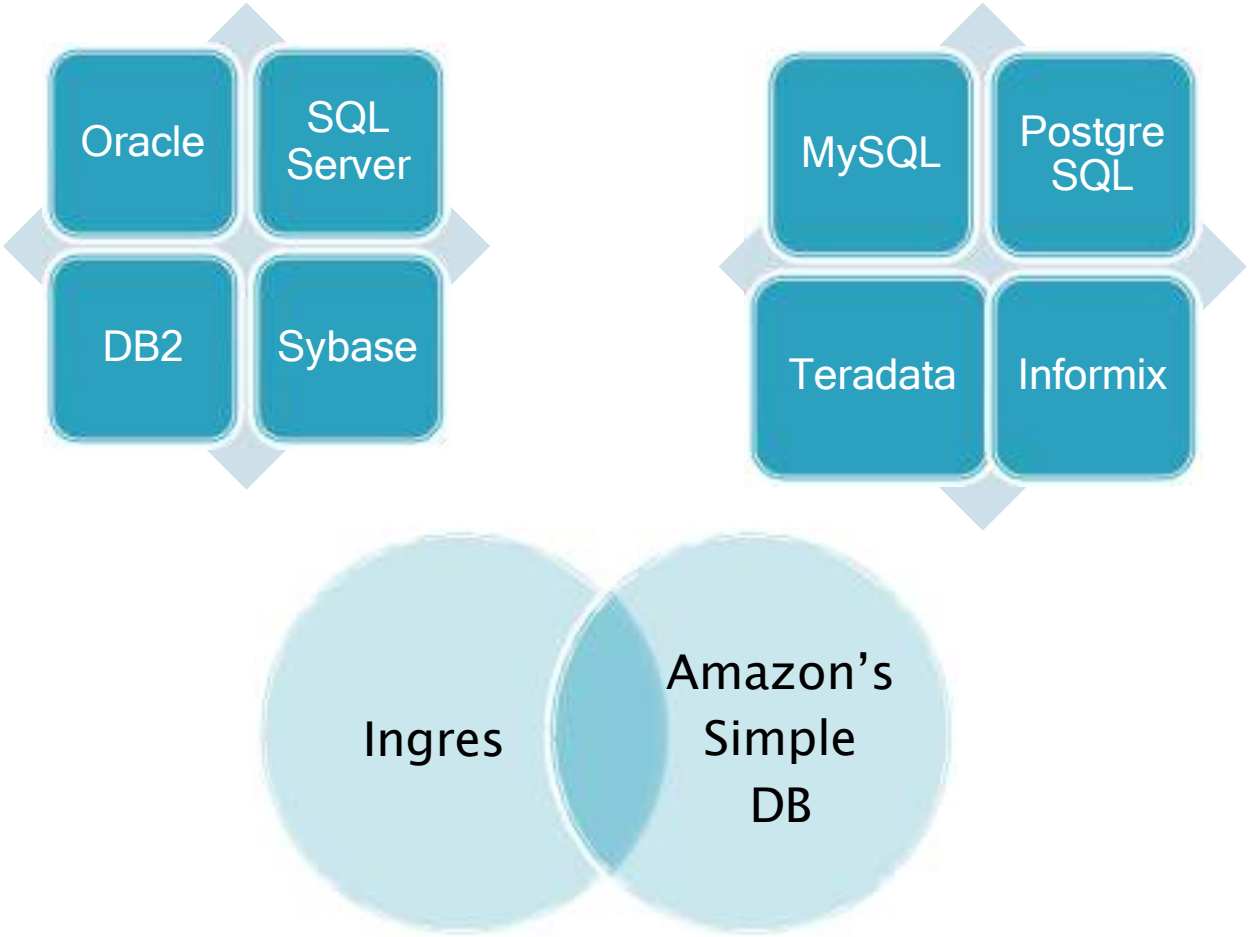
Database System Applications

- **Banking-** for customer information, accounts & loans, and banking transactions.
- **Airlines-**for reservations & schedule information.
- **Universities-**for student information, course registration and grades.
- **Credit card transactions-**for purchases on credit cards & generation of monthly statements.
- **Telecommunication-**for keeping records of calls made, generating monthly bills, maintaining balances, information about communication networks.
- **Finance-**for storing information about holdings, sales & purchases of financial instruments such as stocks & bonds.
- **Sales-**for customer, product and purchase information.
- **Manufacturing-**for management of supply chain & for tracking production of items in factories.
- **Human resources-**for information about employees, salaries, payroll taxes and benefits

Applications of DBMS



Database system



Traditional File systems

Before the evolution of DBMS, organizations used to store information in file systems.

- The system stores permanent records in various files & it need application program to extract records , or to add or delete records .

In traditional file processing, each user defines and implements the files needed for a specific application.

Traditional file system

For example, one user, the grade reporting office, may keep a file on students and their grades. Programs to print a student's transcript and to enter new grades into the file are implemented.

A second user, the accounting office, may keep track of students' fees and their payments.

- Although both users are interested in data about students, each user maintains separate files and programs to manipulate these files because each requires some data not available from the other user's files.

This redundancy in defining and storing data results in wasted storage space and in redundant efforts to maintain common data up-to-date.

File System VS DBMS

Data redundancy and inconsistency

- Different →Programmers
- → Files
- →Structures
- →Programming Languages
- Redundancy
- Higher Storage and access cost

Difficulty in accessing the data

- Find all the students who gained 25 points in Hacker rank
- Find out all students who cleared all the subjects
- File systems are NOT convenient and efficient
- Cannot offer needed data-retrieval easily

Data isolation

- Different -> Programmers
- -> file
- -> Structures
- Files are stored at different locations
- So data isolation is difficult in file system

Integrity problems

- Consistency constraints
- Ex: student cant have less attendance
- Easy to maintain integrity in software but not in file system
- IN DBMS new constraints can be added whenever required

Atomicity problems

- Difficult to ensure atomicity in file-processing
- Restoration of data is difficult

Concurrent-access anomalies

- Concurrency is good
- May lead to inconsistency
- Example: Accessing a Google document were permission is given to everyone to read & write the document

- Data access
- Authentication and Authorization
- Finance personnel should not access Academic records
- Enforcing access constraints in file systems is difficult

Advantages of DBMS

- Controlling Redundancy
- Restricting Unauthorized Access
- Providing Storage Structures for Efficient Query Processing
- Permitting Inference and Actions using Rules
- Providing Multiple User Interfaces
- Representing Complex Relationship among Data
- Enforcing Integrity Constraints
- Providing Backup and Recovery

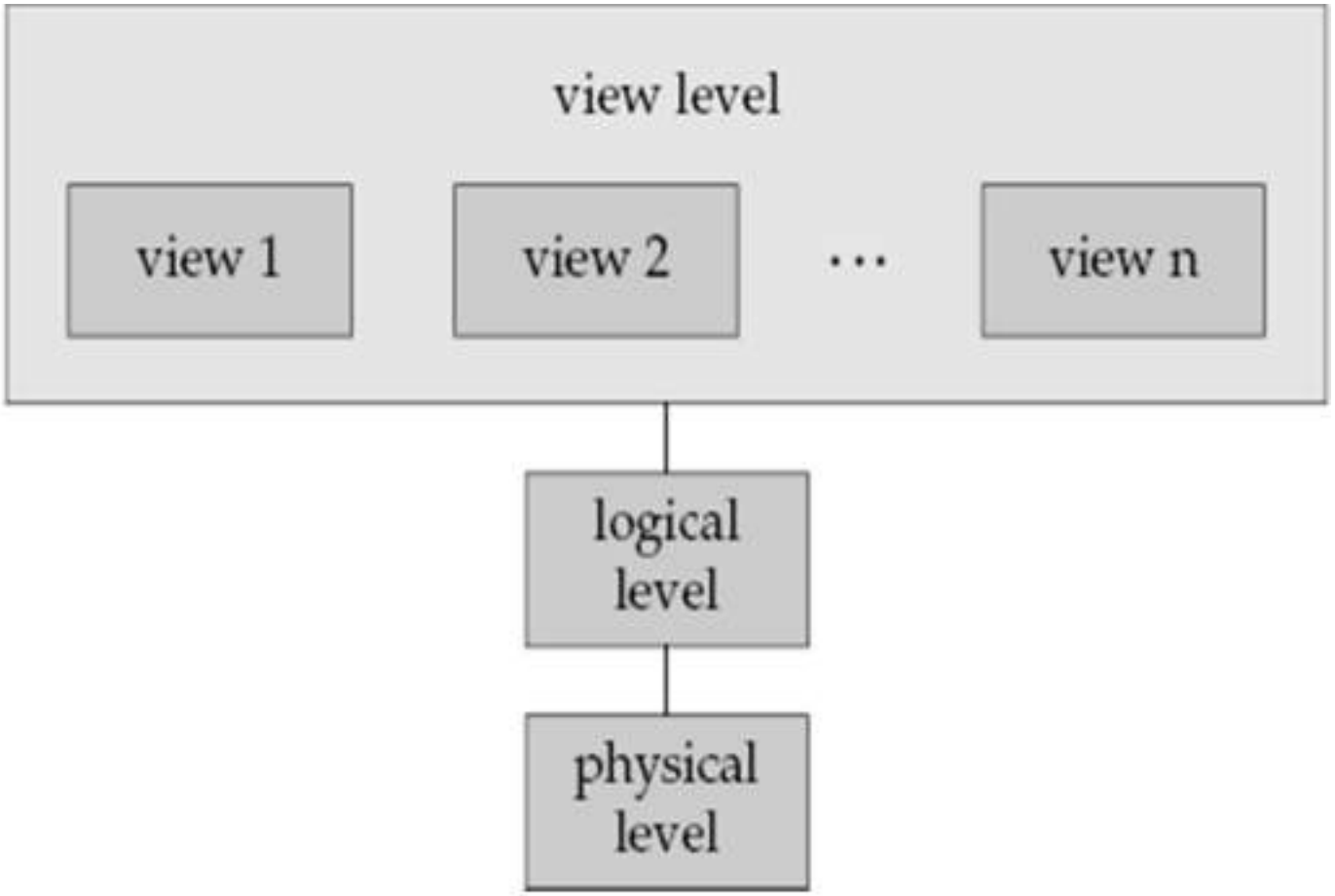
Data Abstraction

Data abstraction

Method of hiding the actual (complex) details from users is called as data abstraction.

That is, the DBMS system hides certain details of how the data are stored and maintained.

Levels of data abstraction



Physical level

It is the lowest level of abstraction & specifies how the data is actually stored.

Example:

A banking enterprise may have several such record types, including

Customer: with customer-id, customer-name, customer- street, customer-city

Account: with fields account-number and balance

Employee: with fields employee-name and salary

Logical level

It is the next level of abstraction & describes what data are stored in database & what relationship exists between various data.

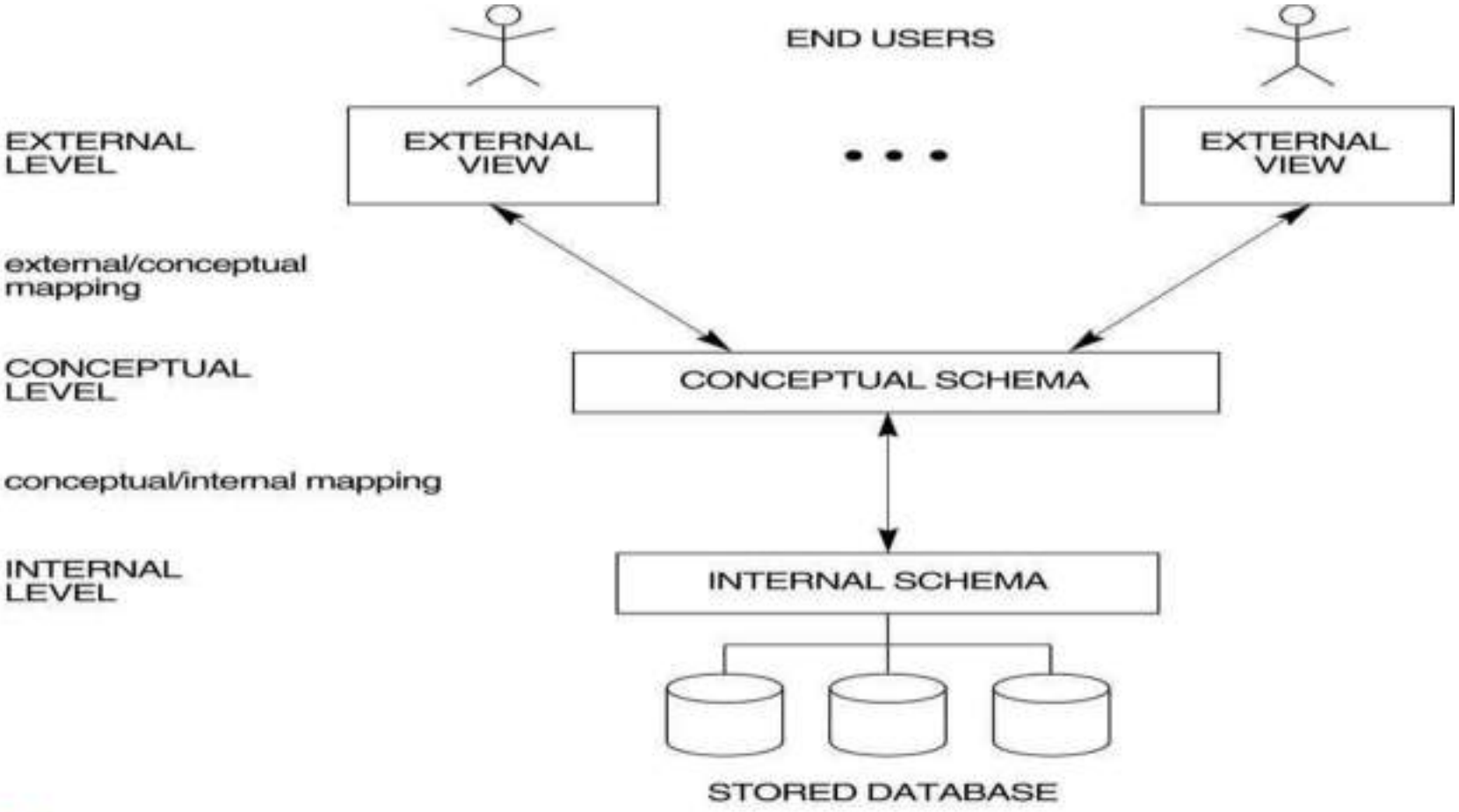
View level

This level contains the actual data which is shown to the users.

This is the highest level of abstraction & the user of this level need not know the actual details of data storage.

Structure of a DBMS

ANSI-SPARC 3-level DBMS Architecture



Three-schema architecture

here ANSI-SPARC stands for America National Standards Institute, Standards Planning And Requirements Committee.

The three-schema architecture is a convenient tool for the user to visualize the schema levels in a database system.

In this architecture, schemas can be defined at the following three levels:

- **Internal schema/Physical schema**
- **Conceptual schema**
- **External schema**

The internal level has an internal schema, which describes the physical storage structure of the database.

The conceptual level has a conceptual schema, which describes the structure of the whole database for a community of users. The conceptual schema hides the details of physical storage structures and concentrates on describing entities, data types, relationships, user operations, and constraints.

The external or view level includes a number of external schemas or user views.

The processes of transforming requests and results between levels are called mappings.

Example: university database

Physical schema:

- Relations stored as unordered files.
- Index on first column of students

Conceptual schema:

- Student (sid: string, name: string, age: number, percent: real)
- Courses (cid: string, cname: string, credits: number)
- Enrolled (sid: string, cid: string, grade: string)

External schema:

- Course_info(cid: string, enrollment: integer)

DATA INDEPENDENCE

DATA INDEPENDENCE

The changes can be made in one level without affecting the other levels that is called data independence.

Data independence is the capacity to change the schema at one level of a database system without having to change the schema at the next higher level.

Types of data independence

Physical data independence is the capacity to change the internal schema without having to change the conceptual (or external) schemas.

Logical data independence is the capacity to change the conceptual schema without having to change external schemas or application programs.

Entity- Relationship Model

Entity- Relationship Model

The E-R model is the most commonly used conceptual model.

In this model, the real world consists of a collection of basic objects called entities and the relationships among these objects.

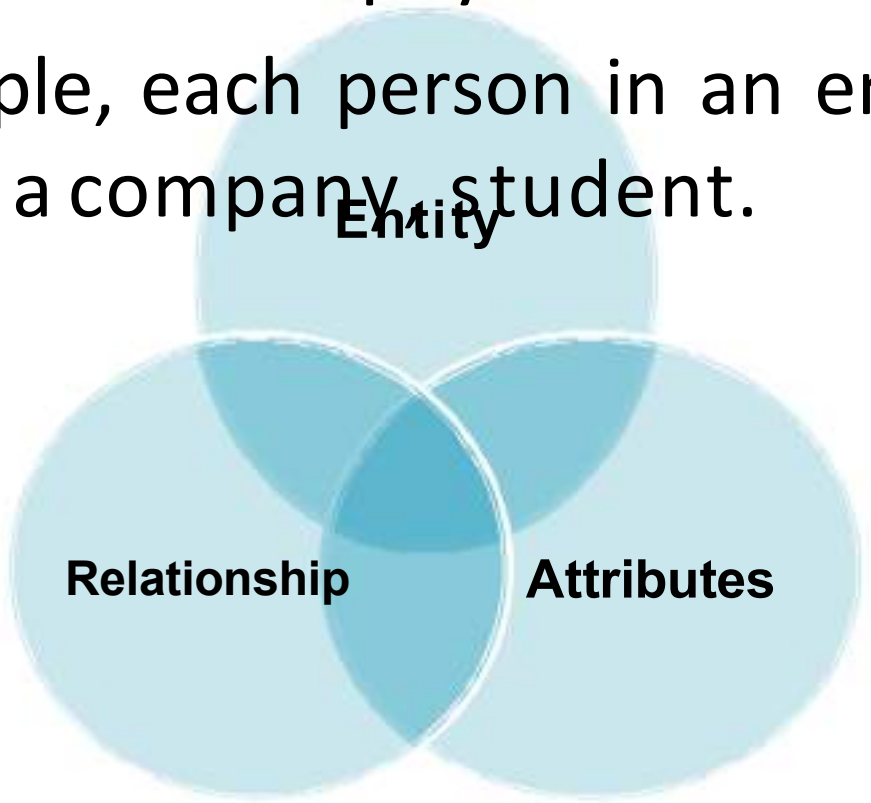
The end product of the modeling process is an entity-relationship diagram (ERD) or ER diagram.

But it is not implemented but design for creating the database.

The E-R data model employ three basic notions

Entity

- It is an object with a physical existence.
- For example, each person in an enterprise , car, house, a company, student.



Entity

It is an object with a physical existence.

For example, each person in an enterprise, car, house, a company, student.

Entity Type & Entity Sets

Entity Type – collection of entities that have the same attributes.

- Ex: STUDENT

Entity Set – The collection of all entities of a particular entity type.

Ex: Set of all rows 10 rows of STUDENT

STUDENT

Name	Age	Rollno

Graphical representation of entity sets

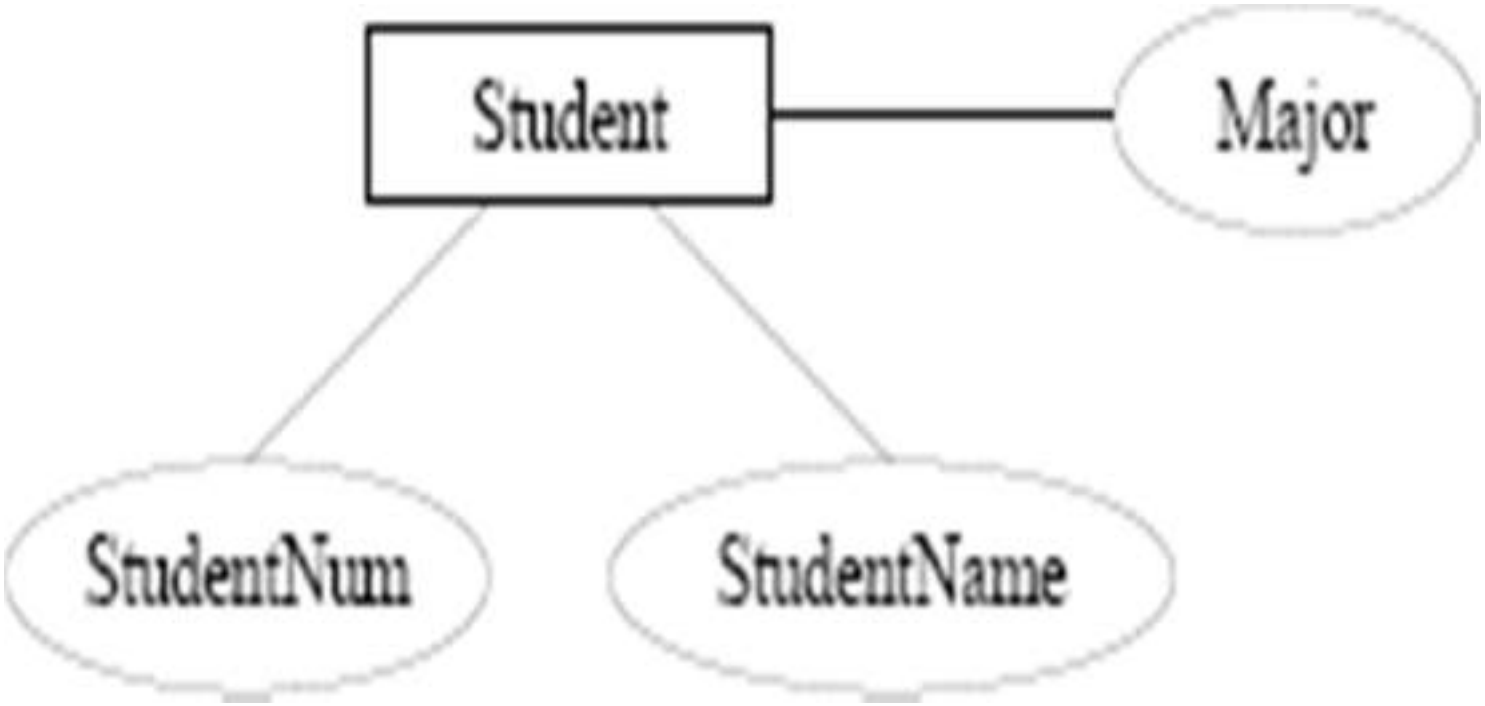
Attributes Closure

Attributes

Attributes are the particular properties that describe an entity.

Ex: A STUDENT entity may be described by student's name, student's roll_number.

Graphical representation of attributes



Types of Attributes

- Simple (Atomic) and Composite Attributes
- Single Valued & Multi-valued Attributes
- Stored and Derived Attributes
- Null Valued Attributes
- Complex Attributes

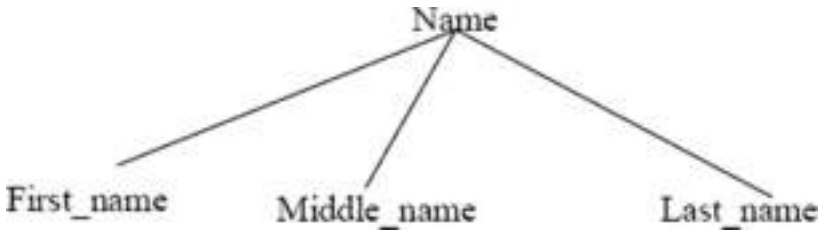
Simple (Atomic) and Composite Attributes

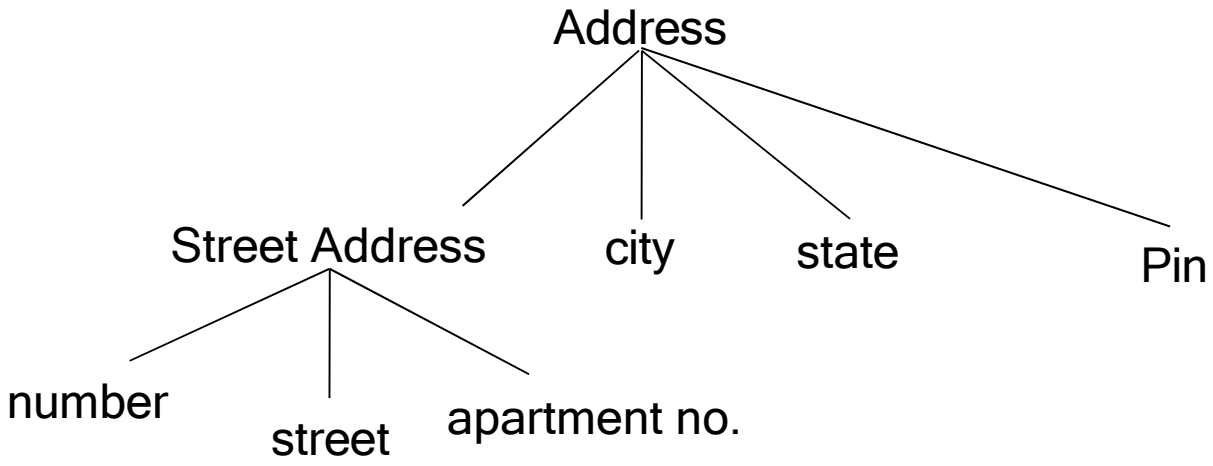
Simple attributes

- These are not divisible into parts.
- For example:
- Employee_Number and Age.

Composite attributes

- These can be divided into smaller subparts. These subparts represent basic attributes with independent meanings of their own.
- For example, take Name and address attributes.


Name
First_name
Middle_name
Last_name


Address
Street Address
city
state
Pin
number
street
apartment no.

Single Valued & Multi-valued Attributes

Single-valued attributes have a single value for particular entity.

Example: Roll_no, Age.

Multi-valued attributes may have more than one value for a single entity.

Example: Phone_no

Stored and Derived Attributes

Derived attribute is not stored in the database but it is derived from some attributes.

- Example: If DOB is stored in the database then we can calculate age of a student by subtracting DOB from current date.

Hence, in this case DOB is the stored attribute and age is considered as derived.

Null Valued Attributes

Null value is a value which is not inserted but it does not hold zero value.

The attributes which can have a null value called null valued attributes.

Example: Mobile_no attributes of a person may not be having mobile phones.

Complex Attributes

Complex composite attribute is a combination of and multi-valued attributes.

Complex attributes are represented by { } and composite attributes are represented by ( ).

Example1: Address_phone attribute will hold both the address and phone_no of any person.

Example2: {(2-A, St-5, Sec-4, Bhilai), 2398124}

Key attribute in an entity type

Key attributes will be having a unique value for each entity of that attribute.

It identifies every entity in the entity set. Key attribute will never be a null valued attribute.

Any composite attribute can also be a key attribute.

Example: roll_no, enrollment _no

There could be more than one key attributes for an entity type.

Domain of value set of an attribute

Domain of an attribute is the allowed set of values of that attribute.

Example: if attribute is ‘grade’, then its allowed values are A,B,C,F.

Grade ={A, B,C,F}

TYPES OF ENTITY TYPES

Strong entity type – Entity types that have at least one key attribute.

Weak entity type – Entity type that does not have any key attribute.

An entity in a weak entity type is identified by a relationship with a strong entity type and that relationship is called **Identifying Relationship** and that strong entity type is called the owner of the weak entity type.

TYPES OF ENTITY TYPES

Student

Roll No.	Name	Age
1	Rakesh	20
2	Nikhil	21
3	Nikhil	21

Secured

Marks

Name	M1	M2	M3
Nikhil	50	45	40
Nikhil	80	75	82

Identifying Relationship

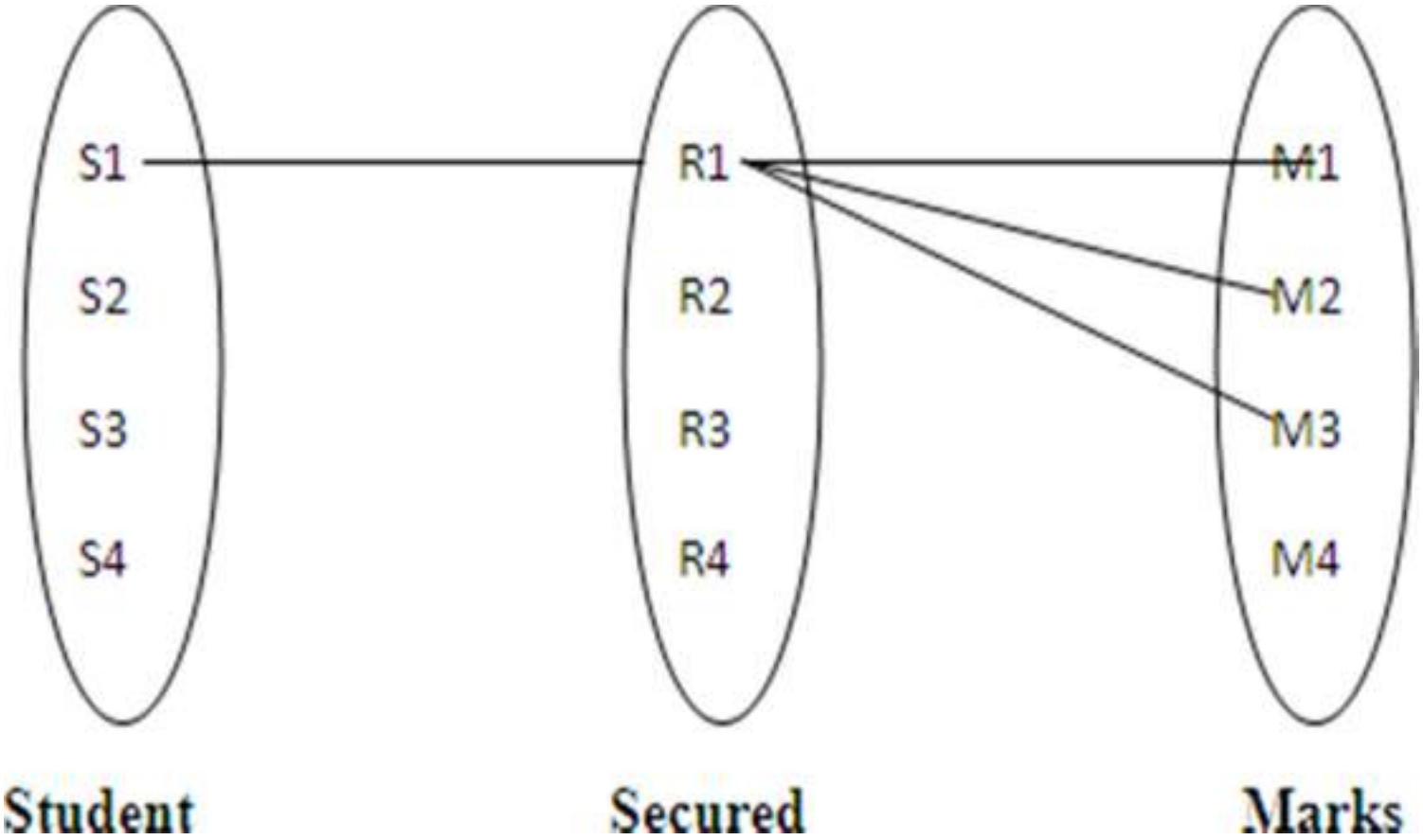
Relationship

Relationship

Relates two or more distinct entities with a specific meaning.

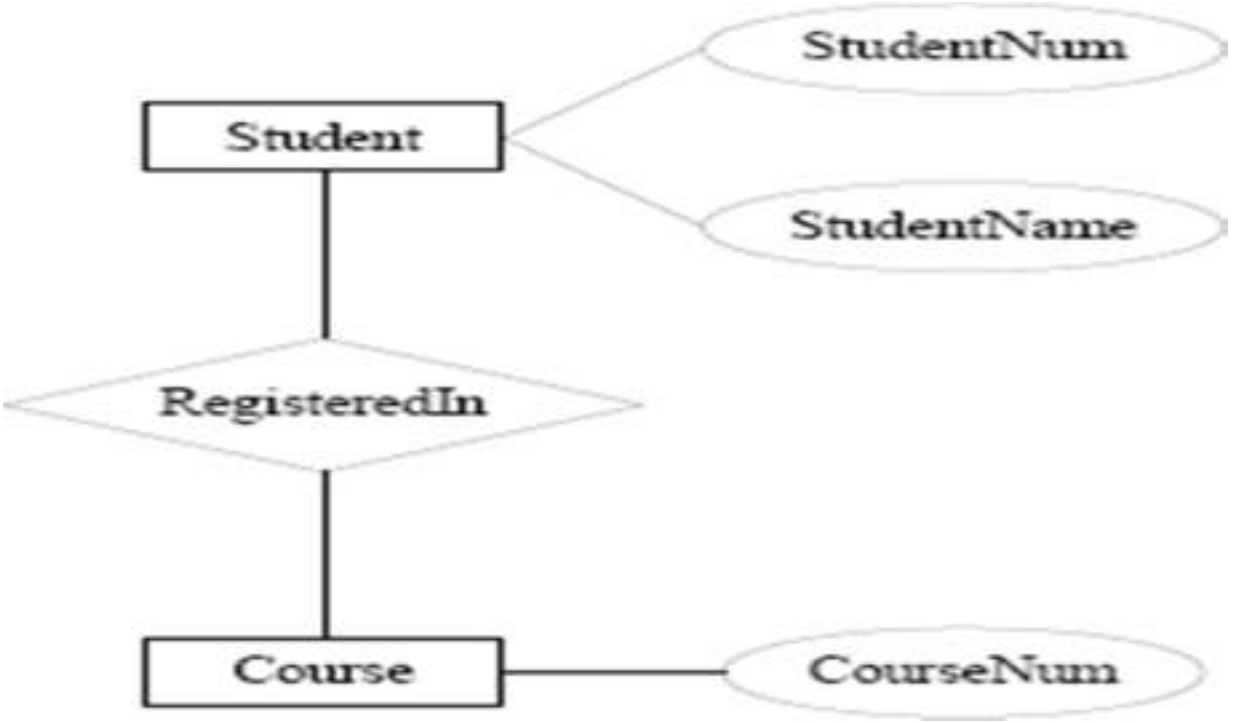
- For example, EMPLOYEE John works on the Product X PROJECT or EMPLOYEE Franklin manages the Research DEPARTMENT.

Terms used: Relationship type, Relationship set, Relationship instances.



Relationship type: secured
Relationship set: {R1, R2, R3, R4}
Relationship instances: R1

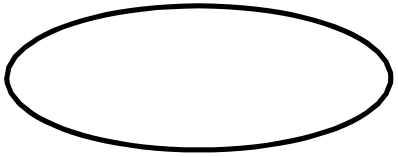
Graphical Representation of Relationship Sets



NOTATIONS USED IN E-R DIAGRAM

Entity Type

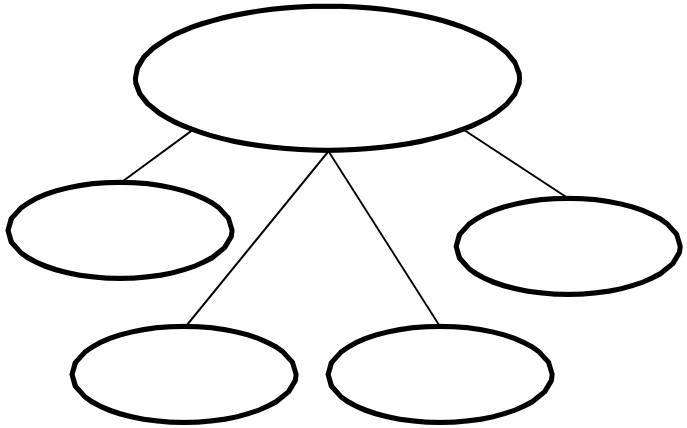
Weak Entity Type



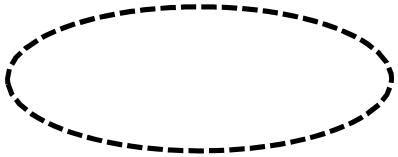
Attribute

Key Attribute

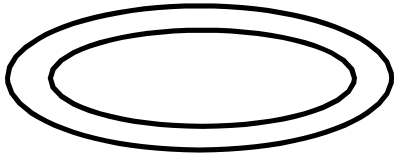
NOTATIONS USED IN E-R DIAGRAM



Composite Attribute

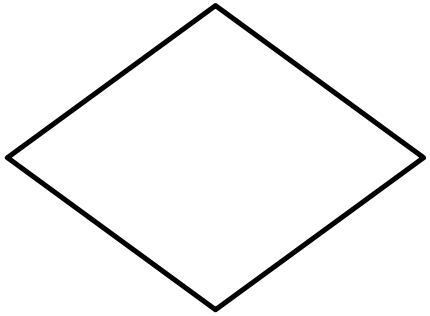


Derived Attribute

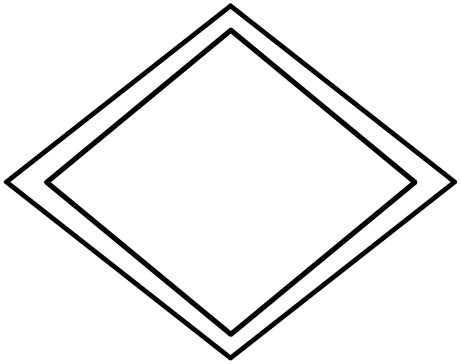


Multivalued Attribute

NOTATIONS USED IN E-R DIAGRAM



Relationship Type



Identifying Relationship

Constraints

Relationship types usually have certain Constraints.

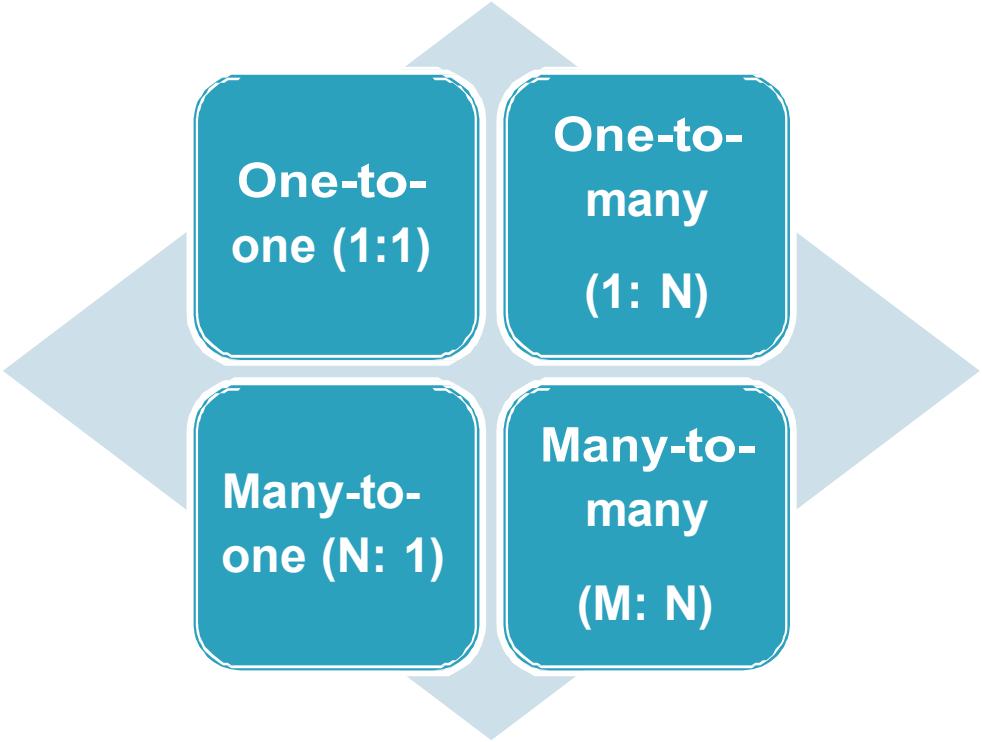
Two main types of relationship Constraints:

Mapping cardinalities
Participation constraints

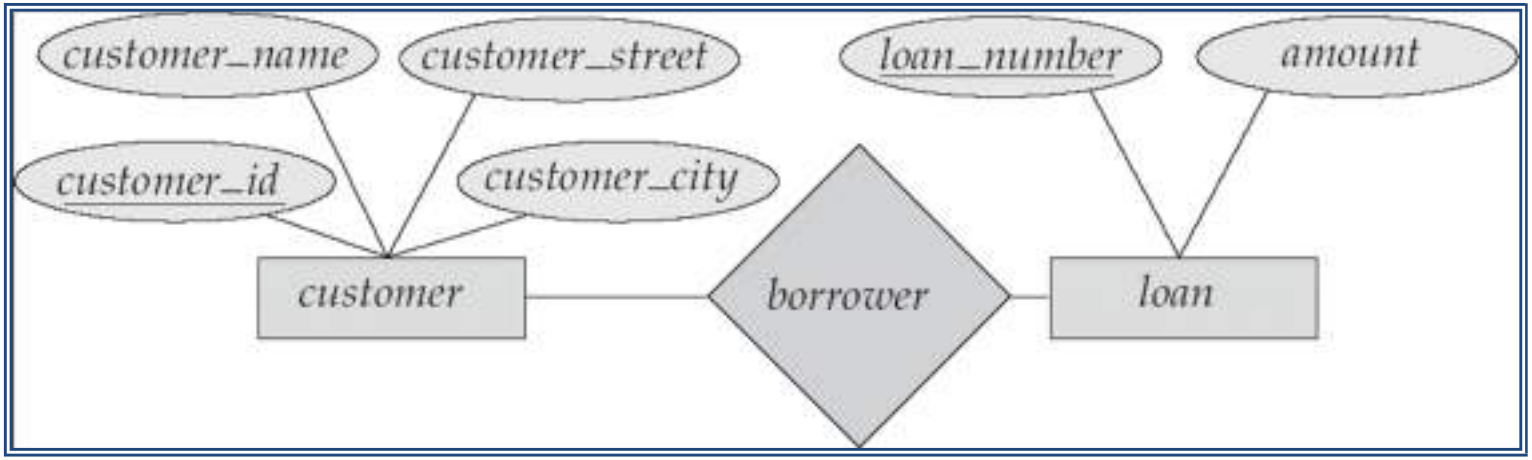
Mapping cardinalities, or cardinality ratios

Specifies the number of relationship instances that an entity can participate in.

Mapping Cardinalities

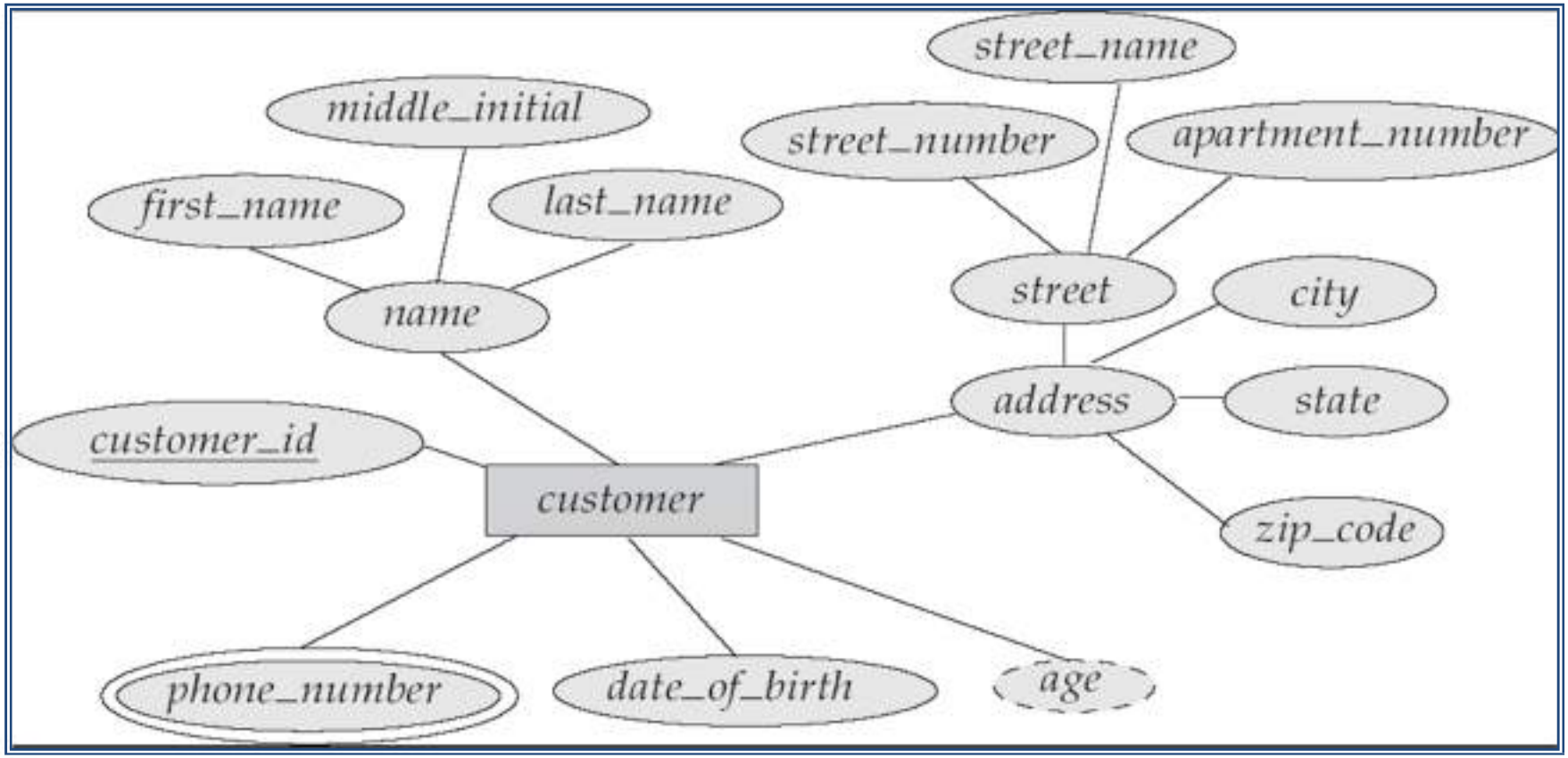


Example of E-R Diagrams

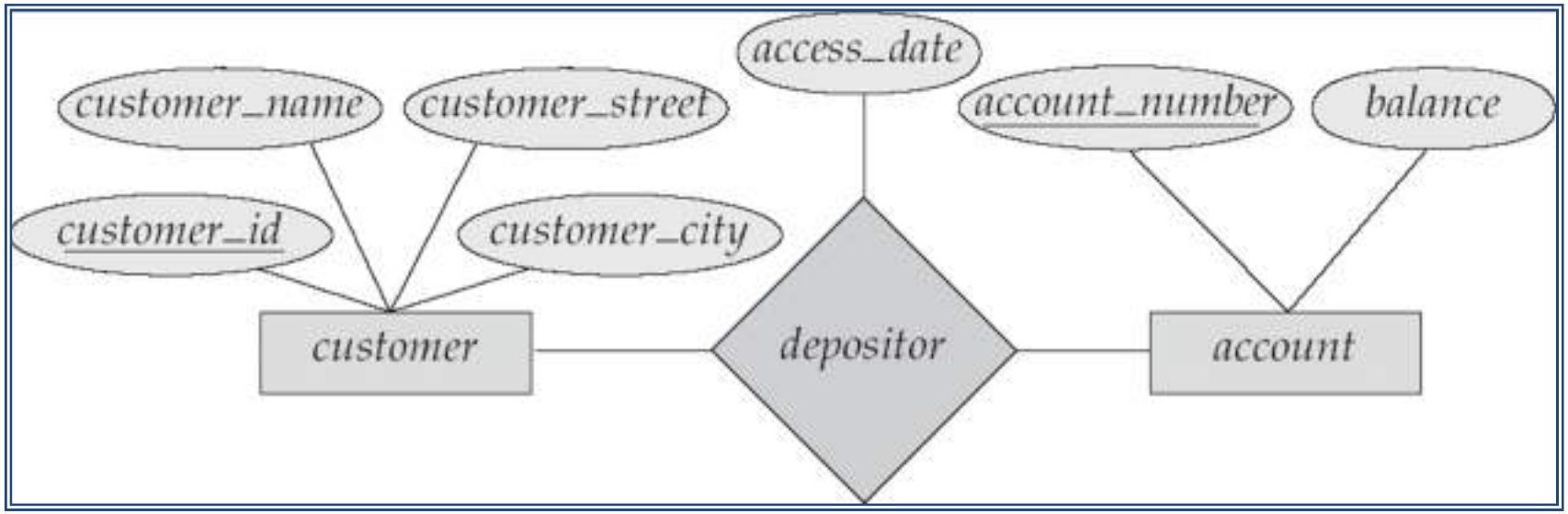


- Rectangles represent entity types.
- Diamonds represent relationship types.
- Lines link attributes to entity types and entity types to relationship types.
- Ellipses represent attributes
- Underline indicates primary key attributes (will study later)

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



Relationship Types with Attributes



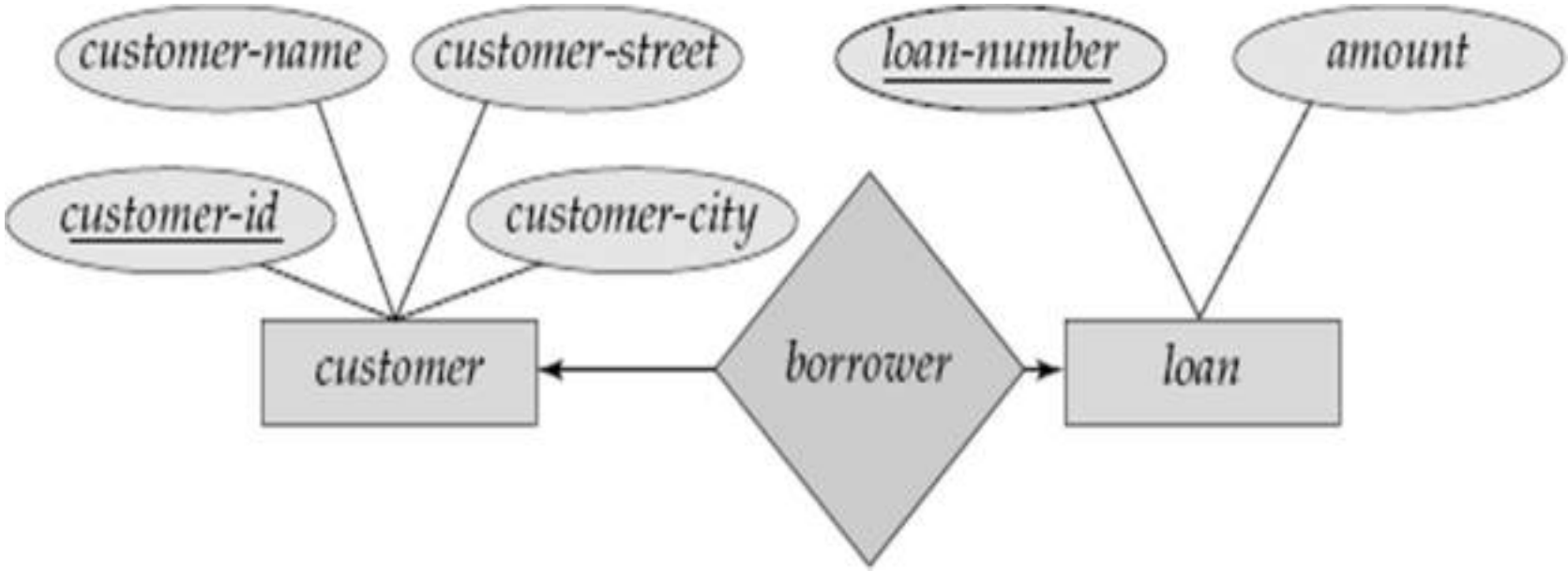
we have the access_date attribute attached to the relationship set depositor to specify the most recent date on which a customer accessed that account.

Cardinality ratio

We express cardinality ratio by drawing

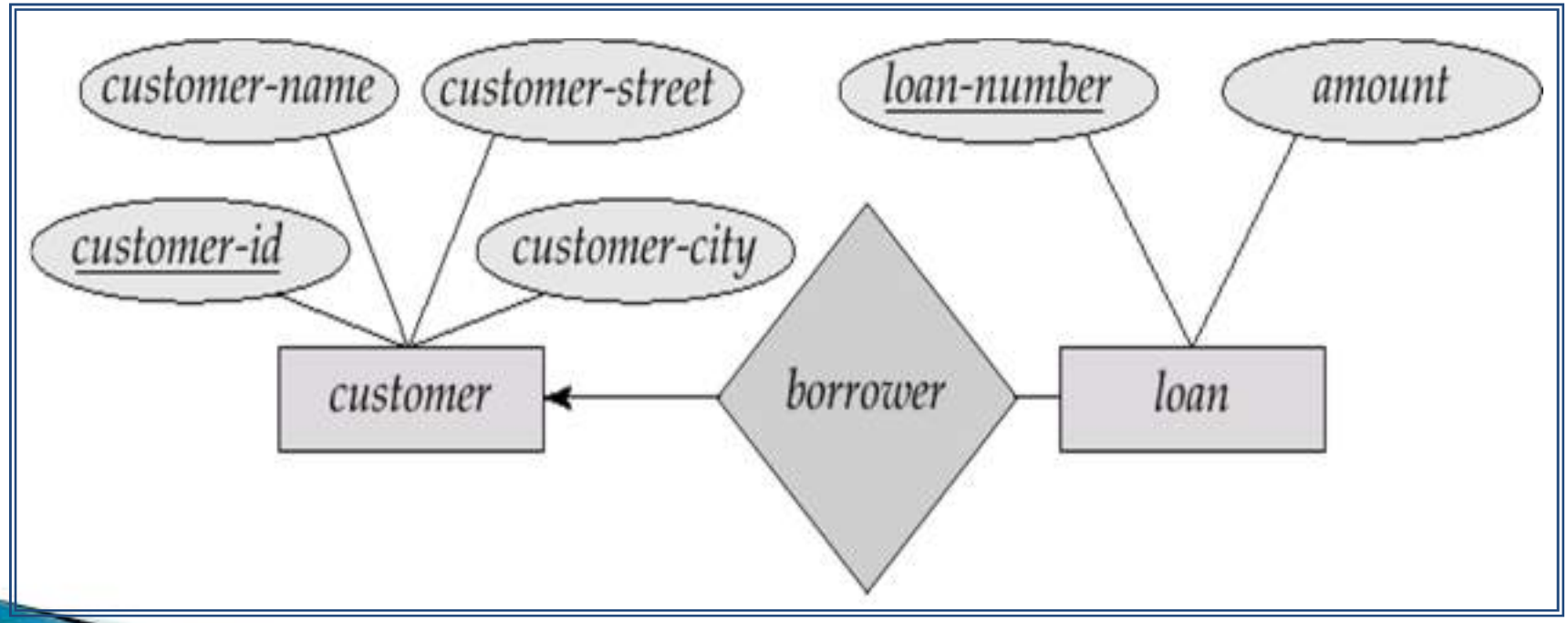
- directed line (→), signifying “one,” or an
- undirected line (—), signifying “many,”

One-To-One Relationship



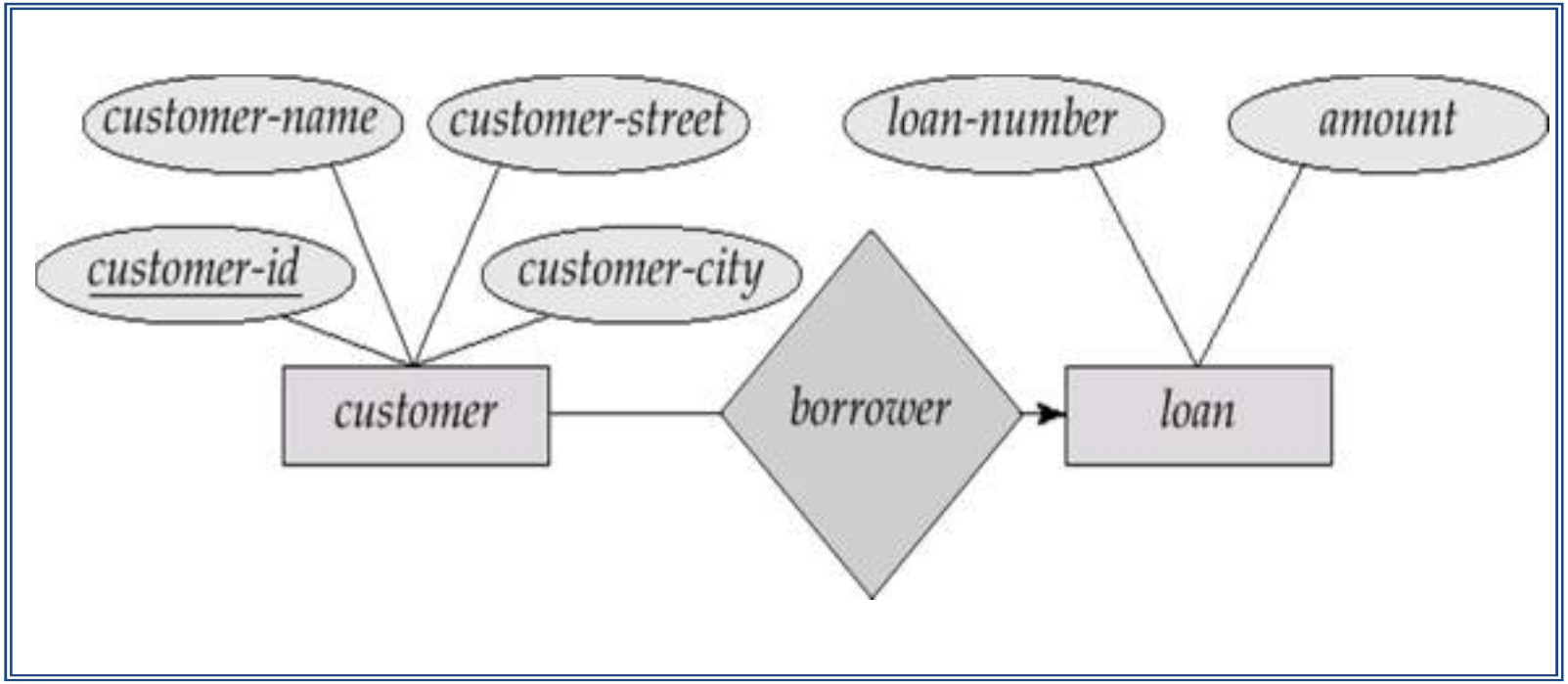
One-To-Many Relationship

In the one-to-many relationship a customer is associated with several loans via *borrower*



Many-To-One Relationships

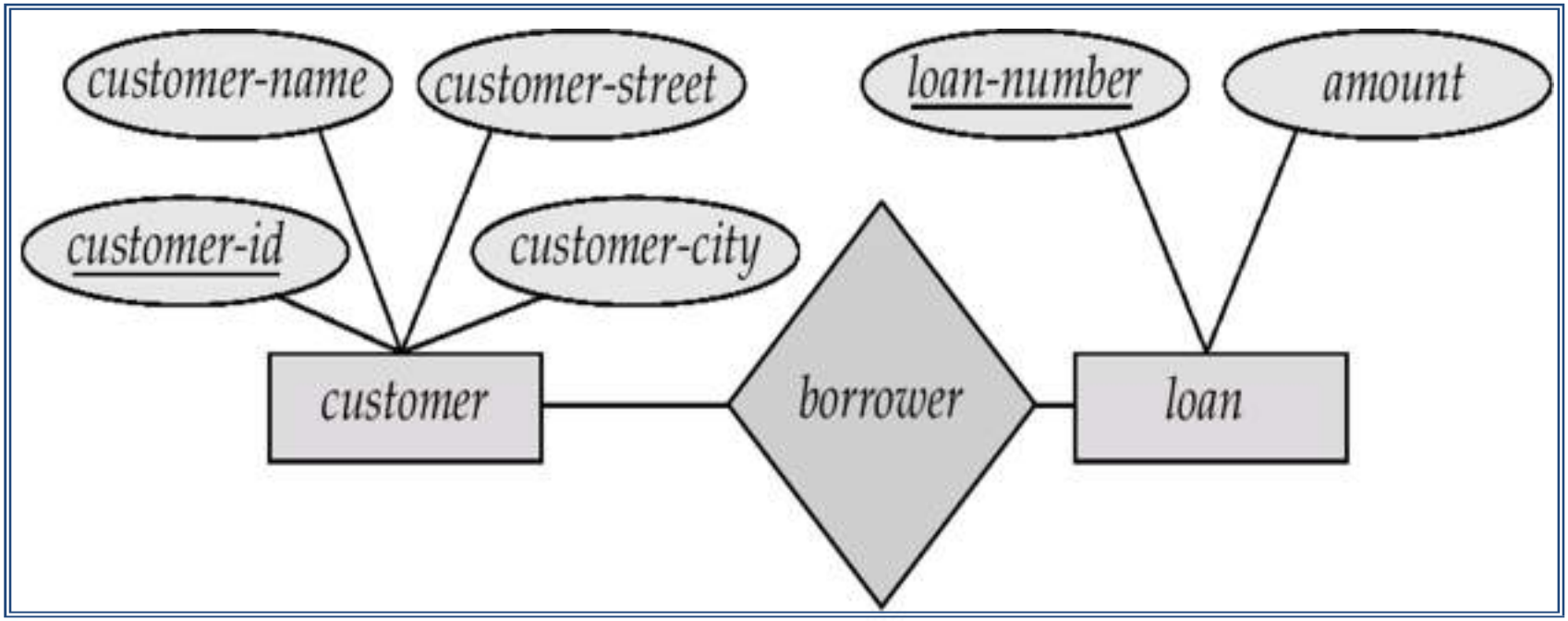
In a many-to-one relationship a loan is associated with several customers via *borrower.*



Many-To-Many Relationship

A customer is associated with several (possibly 0) loans via borrower

A loan is associated with several (possibly 0) customers via borrower

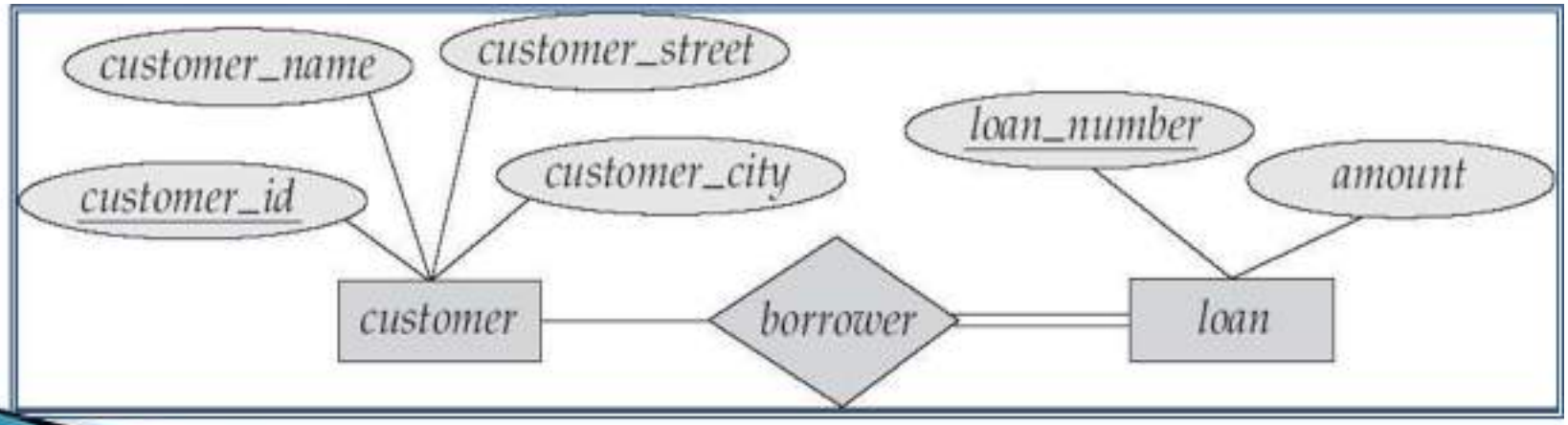


Find out the Cardinality ratio

- Prime minister-country
- classroom –students
- students –classroom
- customer -loan

Participation constraints

- **Total participation :** every entity in the entity type participates in at least one relationship in the relationship type
  - E.g. participation of loan in borrower is total
    - every loan must have a customer associated to it via borrower
- **Partial participation**: some entities may not participate in any relationship in the relationship type
  - Example: participation of customer in borrower is partial
    - some customers may not participate in any loan

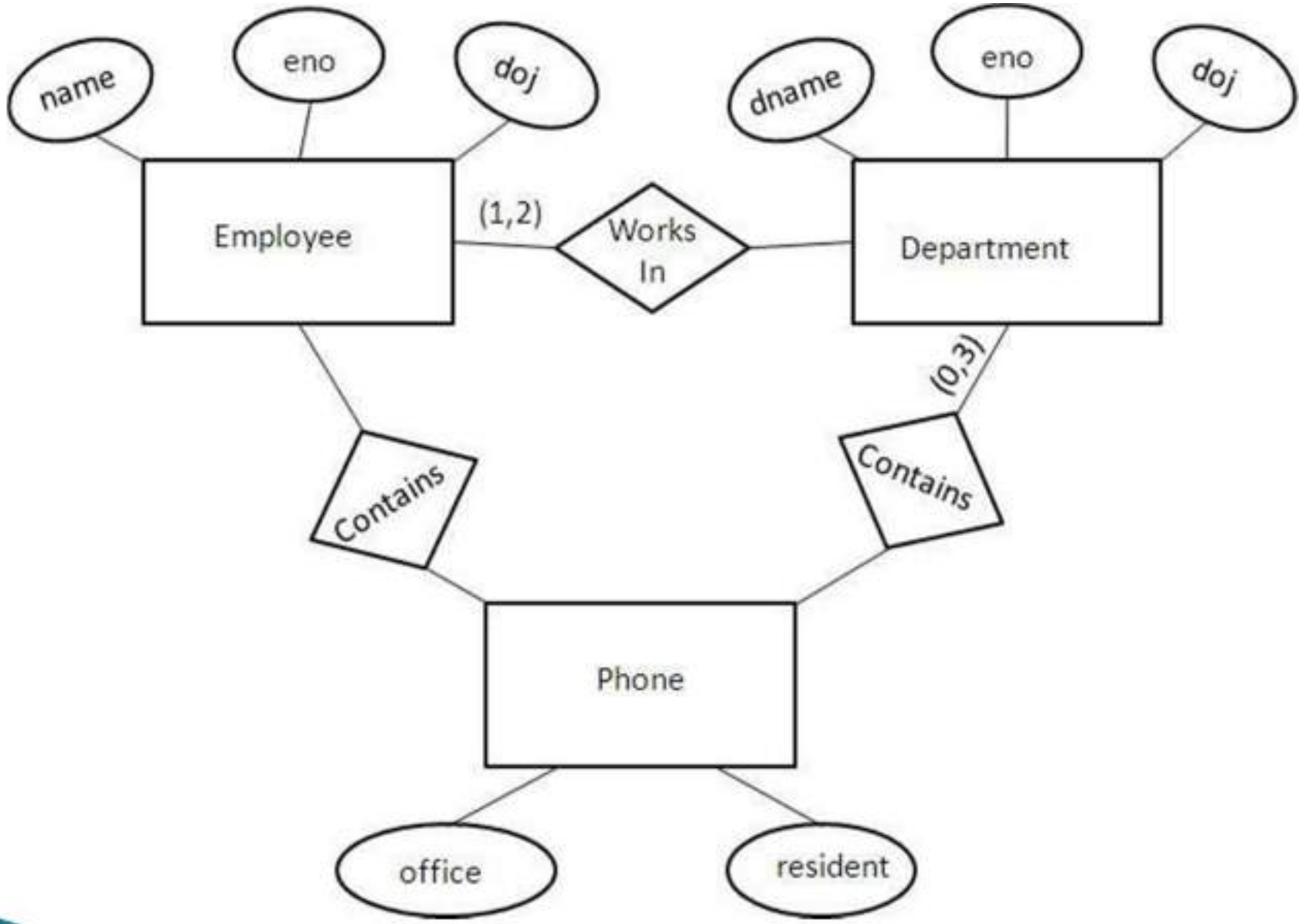


Steps in ER Modeling

- Identify the Entities
- Find relationships
- Identify the key attributes for every Entity
- Identify other relevant attributes
- Draw complete E-R diagram with all attributes including Primary Key

PROBLEMS ON E-R DIAGRAM

Question: An employee works in one department. The department contains phone, the employee also has phone. Assume that an employee works in maximum 2 departments or minimum one department. Each department must have maximum 3 phones or minimum zero phone. Design an E-R diagram for the above.

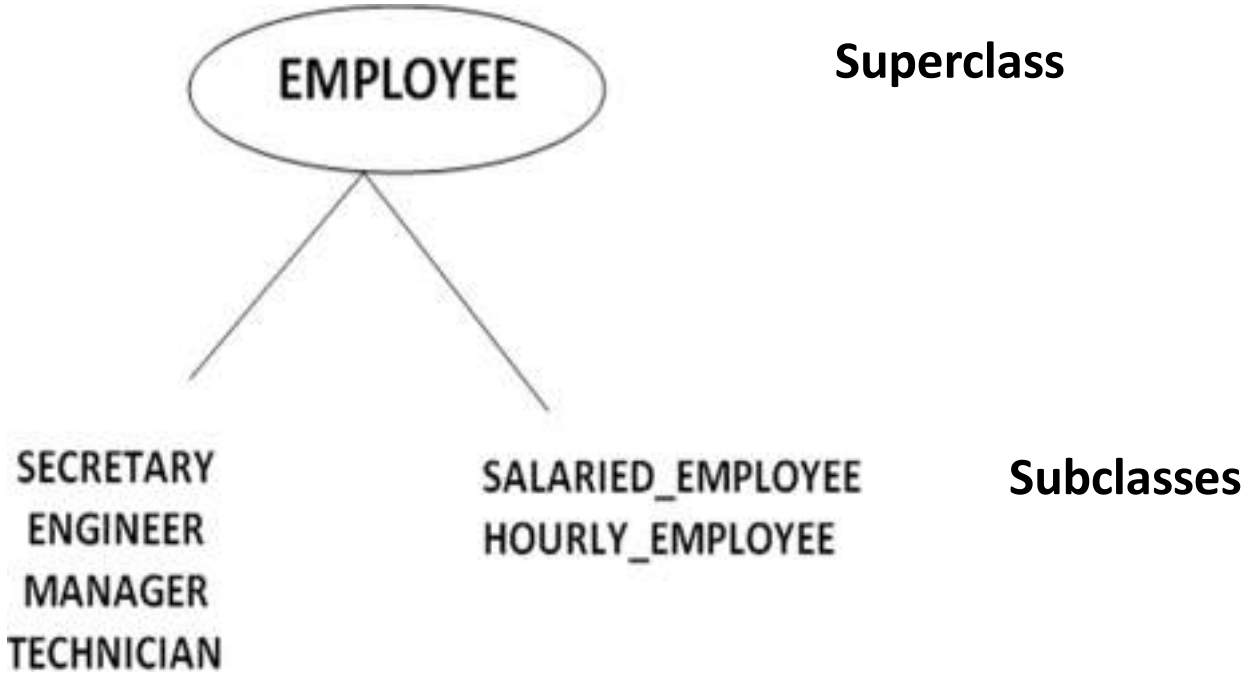

name
eno
doj
dname
eno
doj
Employee
(1,2)
Works In
Department
(0,3)
Contains
Contains
Phone
office
resident

EER (Enhanced Entity-Relationship)

- The EER model is a high-level or conceptual data model incorporating extensions to the original Entity- relationship (ER) model.
- EER includes all the concepts of ER model.
- EER=ER all the concepts + some extension
- Additionally it includes the concepts of
  - superclass and subclass
  - specialization and generalization.

Subclasses and Superclasses

An entity type may have additional meaningful subgroupings.



Subclasses and Inheritance

- An entity class B is said to be a subclass of another entity class A. if it shares an "is-a" relationship with A.

- Example 1: Car "is-a" Vehicle
- Example 2: Manager "is-a" Employee

- An entity of class B is said to be a **specialization** of entities of class A.
- Conversely, entities of class A are **generalizations** of class B entities.

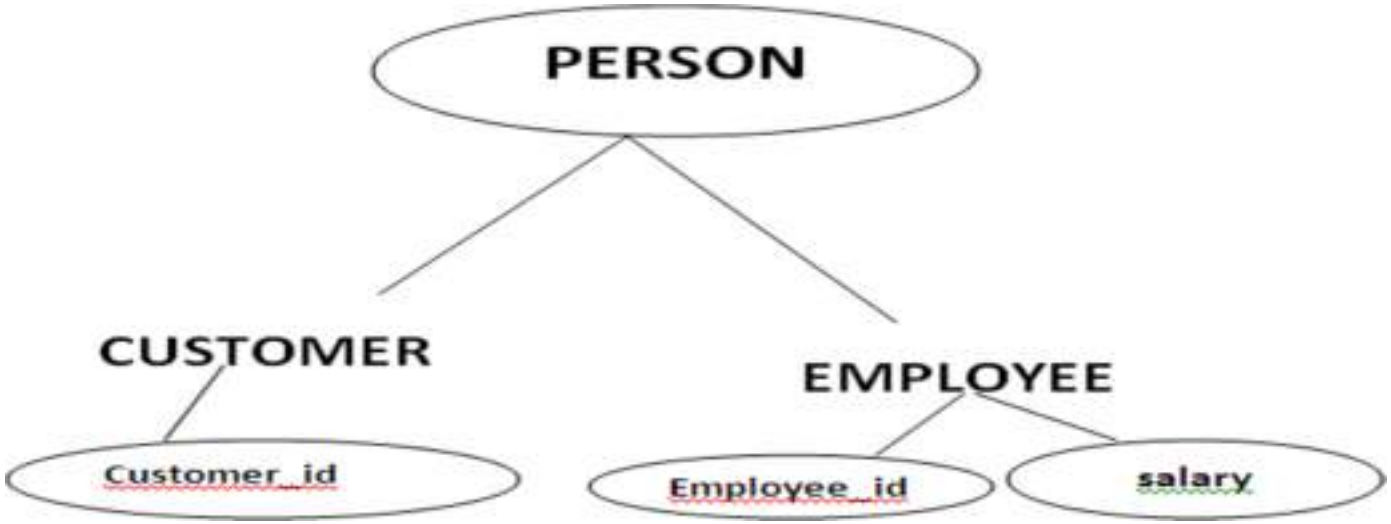
Specialization

Specialization is the process of defining a set of subclasses of a superclass. The set of subclasses is based upon some characteristics of the entities in the superclass.

It follows **top-down design** process.

Represented by a triangle component labeled ISA (E.g. customer "is a" person).

Example of Specialization



The specialization of person allows us to distinguish among persons according to whether they are employees or customers.

Generalization

Generalization is a simple inversion of specialization.

It is a **bottom-up** design process.

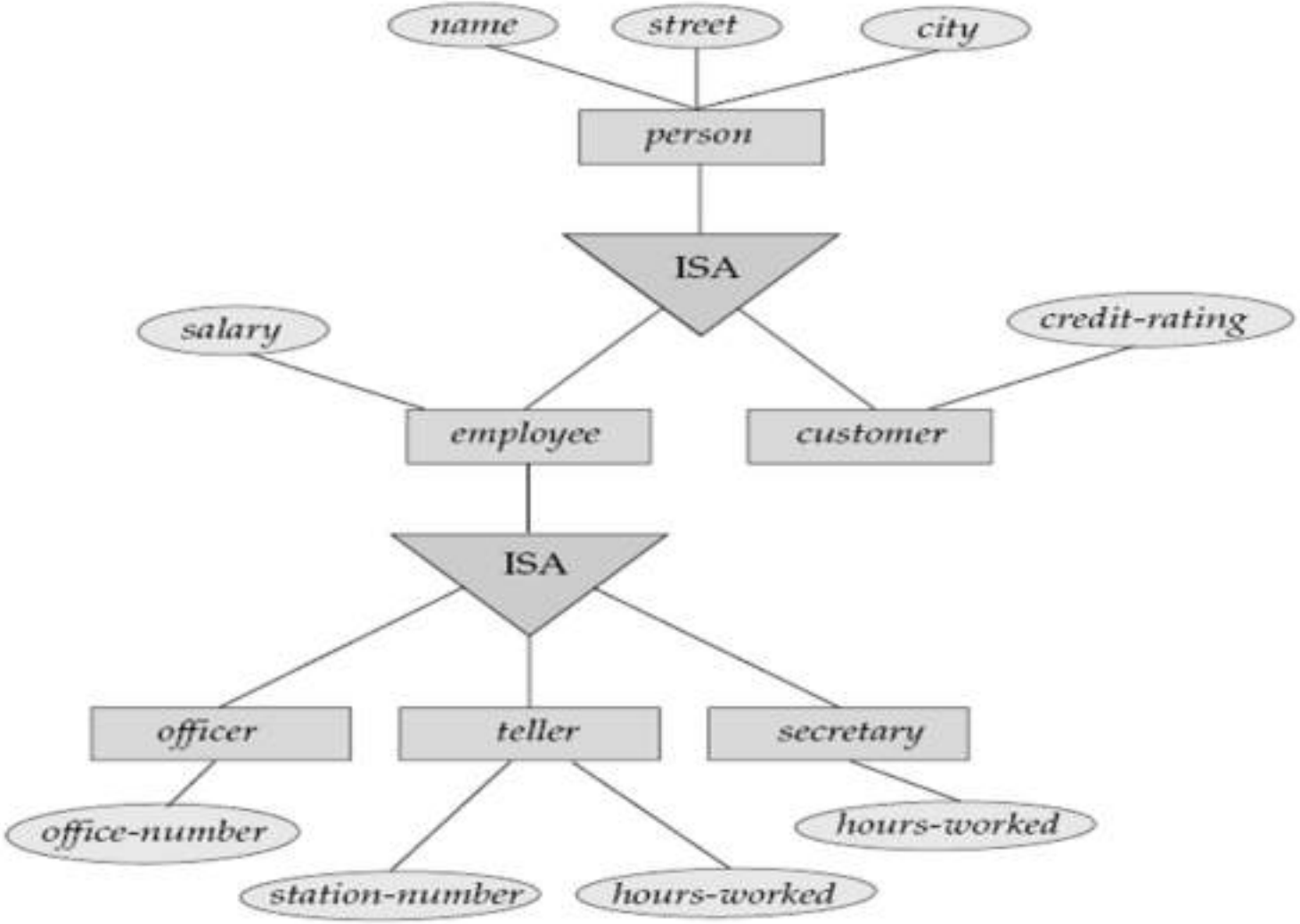
In this process multiple entity sets are synthesized into a higher-level entity set on the basis of common features.

For example, customer entity set with the attributes name, street, city, and customer-id, and an employee entity set with the attributes name, street, city, employee-id, and salary.

There are similarities between the customer entity set and the employee entity set in the sense that they have several attributes in common.

Differences in the two approaches may be characterized by their starting point and overall goal.

Specialization and generalization



Design Constraints on a Specialization/Generalization

Constraint on which entities can be members of a given lower-level entity set.

- **Condition-defined**
  - **User-defined**

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

- **Disjoint**
- **Overlapping**

Design Constraints on a Specialization/Generalization (Cont.)

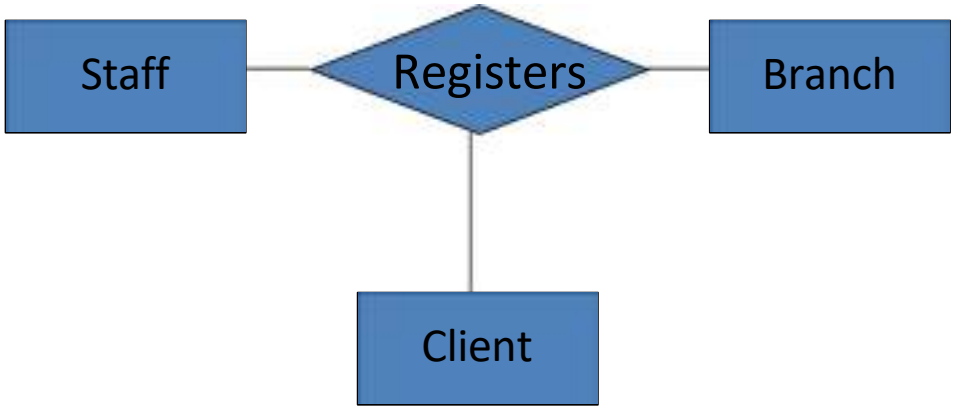
Completeness constraint -- specifies whether or not an entity in the higher-level entity set must belong to at least one of the lower-level entity sets within a generalization.

- **total** : an entity must belong to one of the lower-level entity sets. The account generalization is total: All account entities must be either a savings account or a checking account.

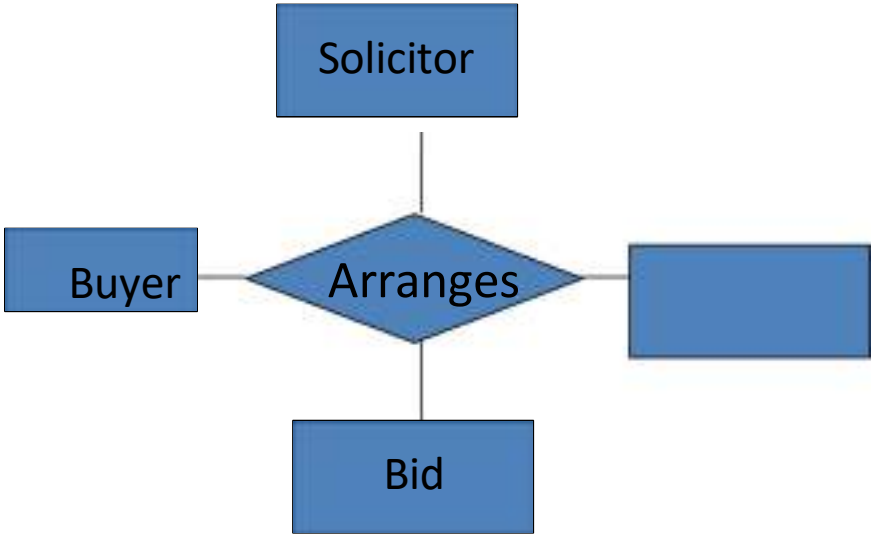
- **partial**: an entity need not belong to one of the lower- level entity sets. Since employees are assigned to a team only after 3 months on the job, some employee entities may not be members of any of the lower-level team entity sets.

Degree of Relationship Type

- It is the number of participating entity types in a relationship.
- A relationship of degree two is called **binary**, a relationship of degree three is called **ternary**...

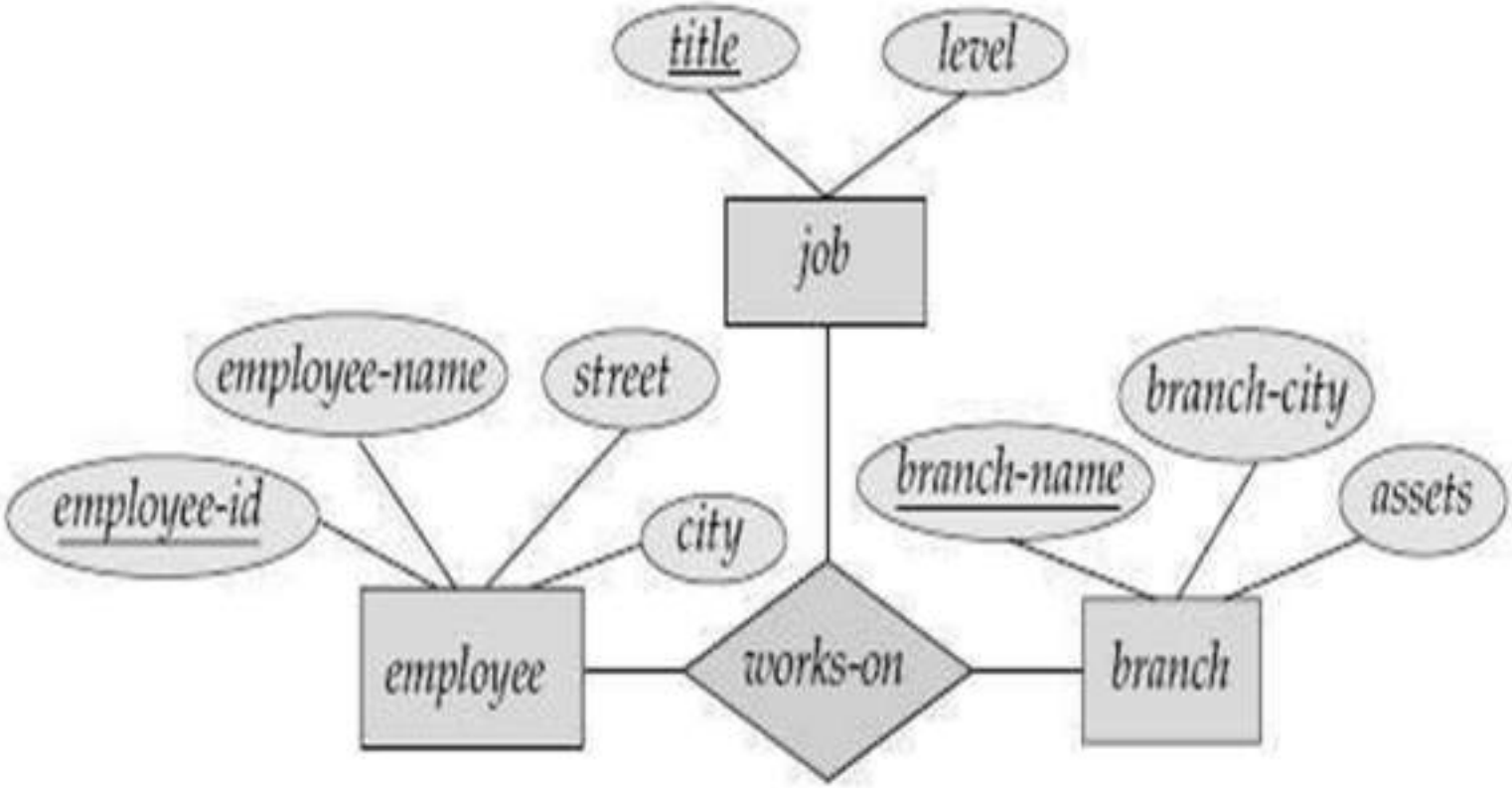


"Staff registers a client at a branch"

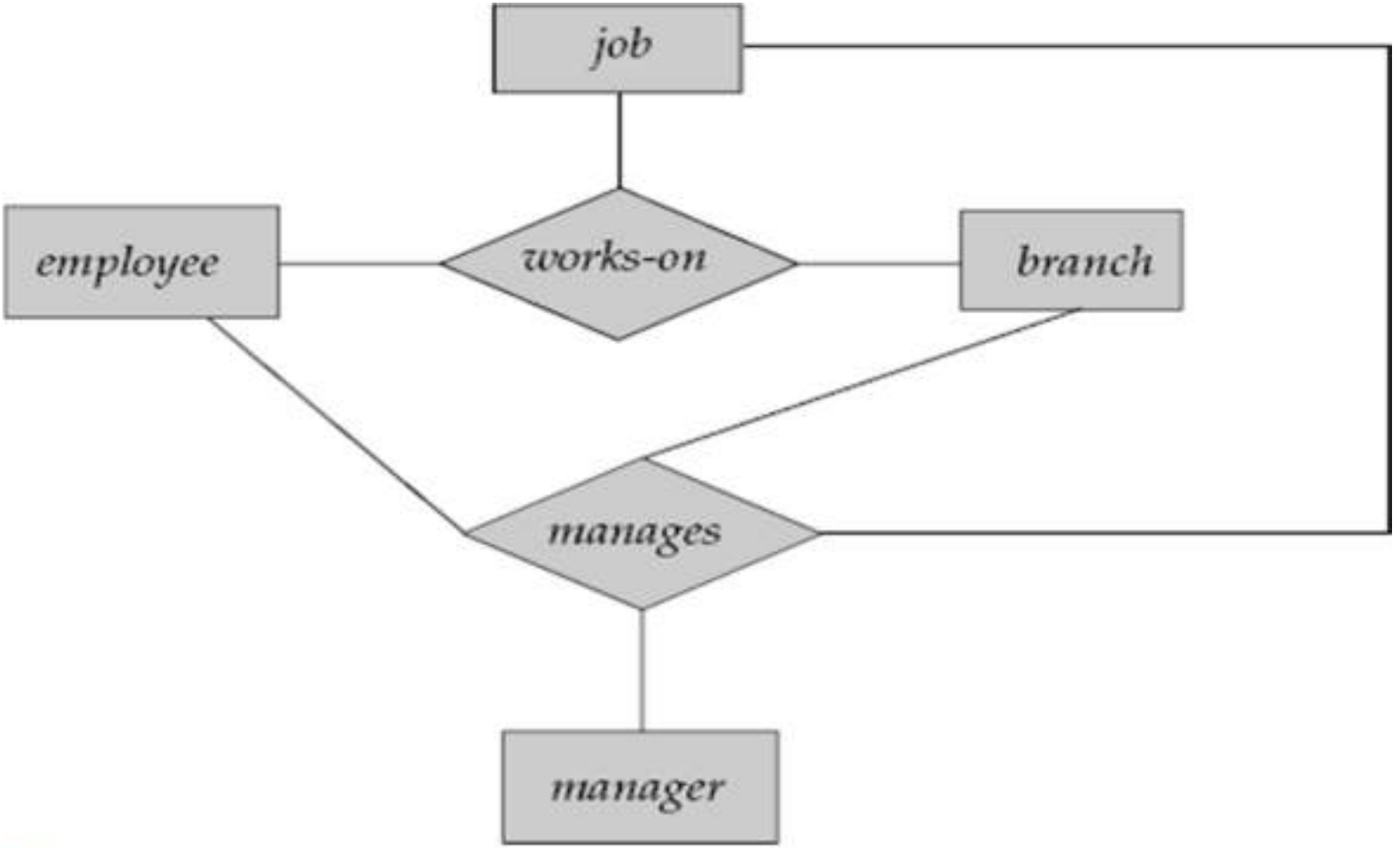


"A solicitor arranges a bid on behalf of a buyer supported by a financial institution"

A ternary relationship



E-R diagram with redundant relationships



Aggregation

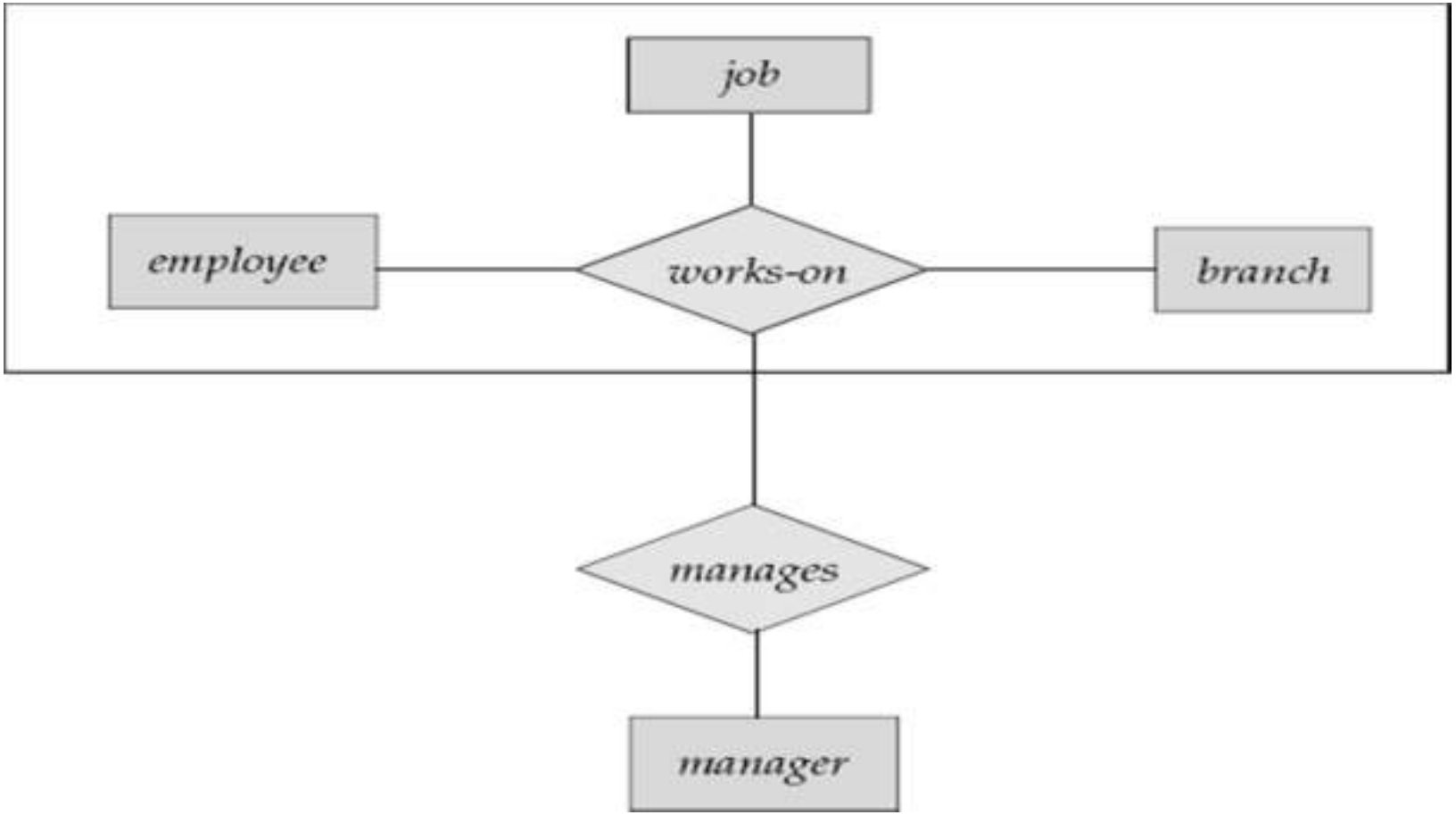
Aggregation is an abstraction through which relationships are treated as higher level entities.

- Thus, for our example, we regard the relationship set works-on (relating the entity sets employee, branch, and job) as a higher-level entity set called works-on.

Such an entity set is treated in the same manner as is any other entity set.

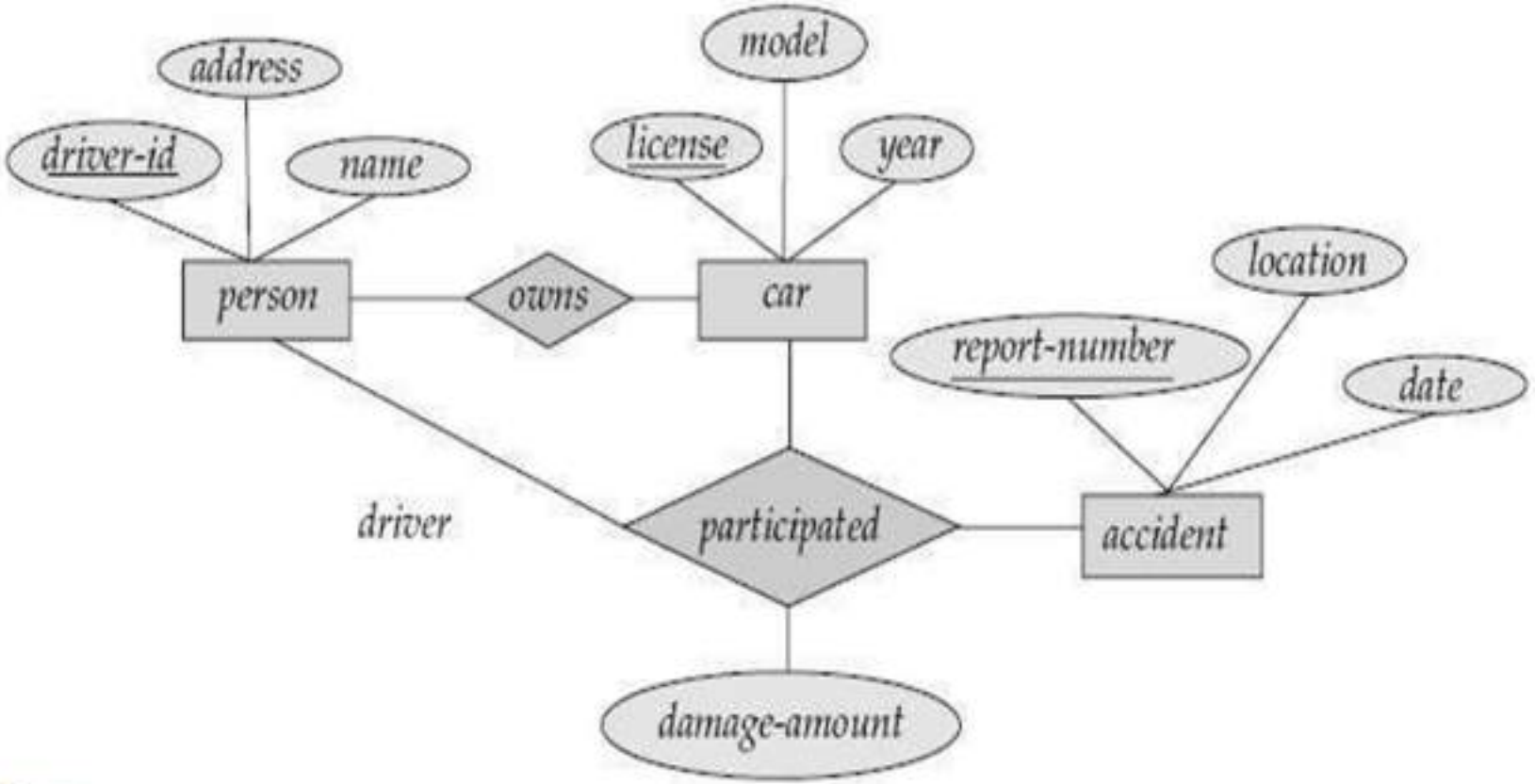
We can then create a binary relationship manages between works-on and manager to represent who manages what tasks.

E-R Diagram With Aggregation



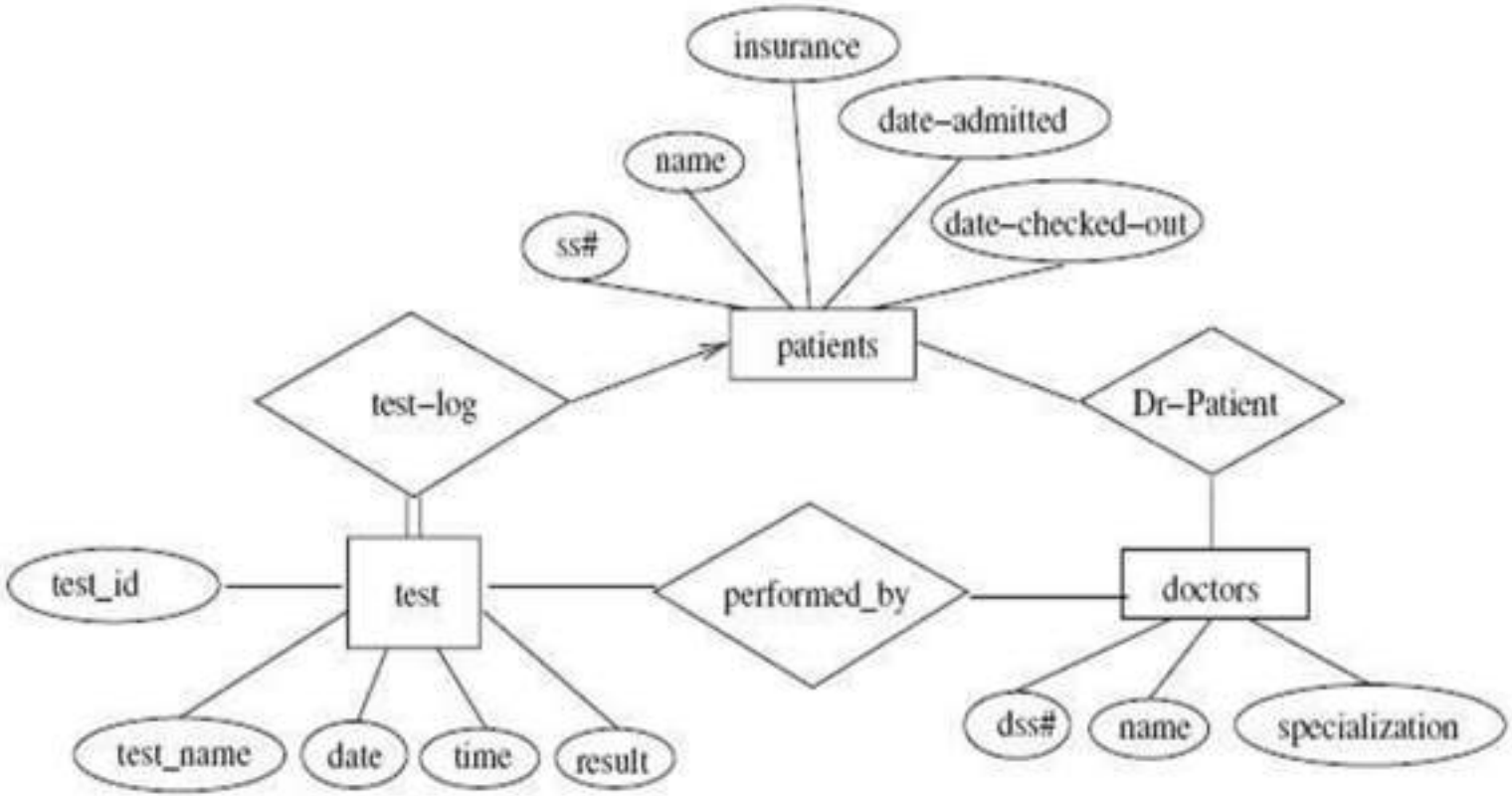
Question 1

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



Question 2

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



ER model question 1

Data in the database of a department store is as follows:

- Each employee is represented. The data about an employee are his employee number, name, address and the department he works for.
- Each department is represented. The data about departments are its name, employees, manager and items sold.
- Each item sold is represented. The data about items are its name, manufacturer, price, model number (assigned by the manufacturer) and an internal item number (assigned by the store).
- Each manufacturer is represented. The data about a manufacturer are its name, address, items supplied to the store and their prices.
- Give an ER diagram for this database. Indicate a key for each entity set and indicate which relationships are one-one and which are many-one.

ER model question 2

- In university database, an entity type section which describes the section offering course.
- The attribute of e section are section number, semester, year, course number, instructor, room no, weekday(domain is the possible combinations of weekday in which the sections are offered {9-9.50 AM, 10-10.50AM,.....3.30-4.50, 5.30-6.20PM etc}.
- assume that section number is unique for each course within a particular semester/year combination. There are several composite keys for section and some attributes are components of more than one key.
- Identify three composite keys, and show how they can be represented in an ER diagram schema.

ER model question 3

Create an ER diagram for the following:

- Each company operates four departments, and each department belongs to one company.
- Each department employs one or more employees and each employee work for one department.
- Each of the employee may or may not have one or more departments and each departments belong to one employee.
- Each employee may or may not have an employment history.
- Make assumptions if necessary and state them. Identify primary, candidate and foreign key.

RELATIONAL MODEL

Syllabus

- Relational data model concepts,
- Constraints,
- Relational algebra,
- Relational calculus,
- SQL: DDL, DML, DCL,
- View,
- Index
- Cursors and

What is data model?

A data model is a collection of concepts that can be used to describe the structure of a database.

Introduction

The entire database and more particularly the modern database uses a very important data model called as the Relational Model.

This model was proposed by Codd in the year 1970.

During that time the popular models were only *Network model* and *Hierarchical model*.

CODD's rules for RDBMS

1. Information Rule
2. Guaranteed Access Rule
3. Systematic Treatment of Null Values
4. Dynamic On-line Catalog Based on the Relational Model
5. Comprehensive Data Sublanguage Rule
6. View Updating Rule
7. High-level Insert, Update, and Delete
8. Physical Data Independence
9. Logical Data Independence
10. Integrity Independence
11. Distribution Independence
12. Non subversion Rule

Relation

The relational model represents the database as a collection of relations.

Each relation resembles a simple table, having a set of rows and set of columns.

Each relation consists of a relational schema and relational instance.

Relational Model Concepts

In the formal language terminology,

- **Row is called as- tuple**
- **Column header is called as-attribute**
- **Table is called as-relation**

The data type describing the types of values that can appear in each column is called a domain.

Relational Model Concepts

The number of attributes in a relation is the **degree** of a relation.

The number of tuples in a relation is called the cardinality of that relation.

Relation schema(R) is the description about the structure of any relation.

Relation state(r) is the data in the relation at any moment of time.

INFORMAL DEFINITIONS

RELATION: A table of values

A relation may be thought of as a **set of rows**.

A relation may be thought of as a **set of columns**.

Each row represents a fact that corresponds to a real-world **entity** or **relationship**.

Each row has a value of an item or set of items that uniquely identifies that row in the table.

Each column typically is called by its **column name** or **column header** or **attribute name**.

FORMAL DEFINITIONS

A **Relation** may be defined in multiple ways.

The **Schema** of a Relation: *R* (A1, A2,.....An)

- Relation schema *R* is defined over **attributes** A1, A2, ...An

For Example -

- CUSTOMER (Cust-id, Cust-name, Address, Phone#)

FORMAL DEFINITIONS

A relation may be regarded as a ***set of tuples*** (rows).

A **tuple** is an ordered set of values

Each value is derived from an appropriate domain.

Each row in the CUSTOMER table may be referred to as a tuple in the table and would consist of four values.

<632895, "John Smith", "101 Main St. Atlanta, GA 30332", "(404) 894-2000">
is a tuple belonging to the CUSTOMER relation.

FORMAL DEFINITIONS

A domain D is a set of atomic values. By atomic we mean that each value in the domain is indivisible as far as the relational model is concerned.

"USA_phone_numbers" are the set of 10 digit phone numbers valid in the U.S.

"Local_phone_numbers" are the set of 7 digit phone numbers.

Employe_ages: possible ages of employees of a company; each must be a value between 15 and 80 years old.

Academic_department_names: the set of Academic department names such as, computer science, economics, and physics, in a university.

FORMAL DEFINITIONS

A relation R is a subset of the Cartesian product of the domains that define R

- $R \subseteq (dom(A1)) \times (dom(A2)) \times \ldots (dom(An))$
- This actually gives the total number of tuples in the Cartesian product. R is represented as, R (A1, A2, . . ., An).

R: schema of the relation

Relational Integrity Constraints

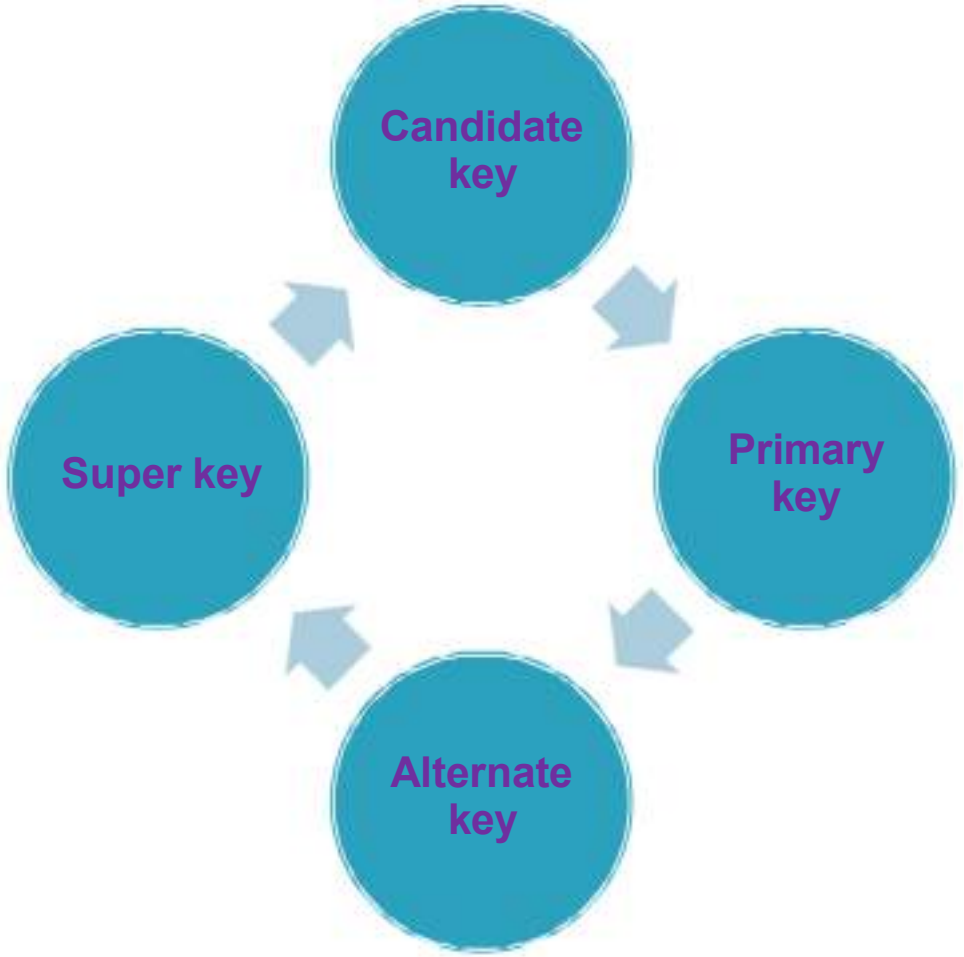
Constraints are the rules applied on the attribute.

Constraints on the relation are the constraints on the attributes of that relation.

There are three main types of constraints:

- **Key constraints**
- **Entity integrity constraints**
- **Referential integrity constraints**

Key Constraints



Alternate & Primary key

Alternate & Primary key is related with candidate key.

In a relation, primary key is a candidate key but only one key is the primary key & the left candidate keys are called alternate key.

AK=CK-PK

Super key

A super key is the superset of any candidate key.

Entity Integrity

The entity integrity constraint states that no primary key value can be null.

This is because the primary key value is used to identify individual tuples in a relation; having null values for the primary key implies that we cannot identify some tuples.

For example, if two or more tuples had null for their primary keys, we might not be able to distinguish them.

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.

$t[PK] \neq null$ for any tuple t in r(R)

Referential Integrity

Referential integrity is concerned with foreign keys. Primary key and foreign key relates two tables.

Referential integrity is a property, when satisfied, requires every value of one attribute of a relation to exist as a value of another attribute in a different relation.

Less formally: For referential integrity to hold, any field in a table that is declared a foreign key can contain only values from a parent table's primary key.

Student

Primary key

Roll_no	Enroll	Name	Addr
1			

MARKS

Foreign key

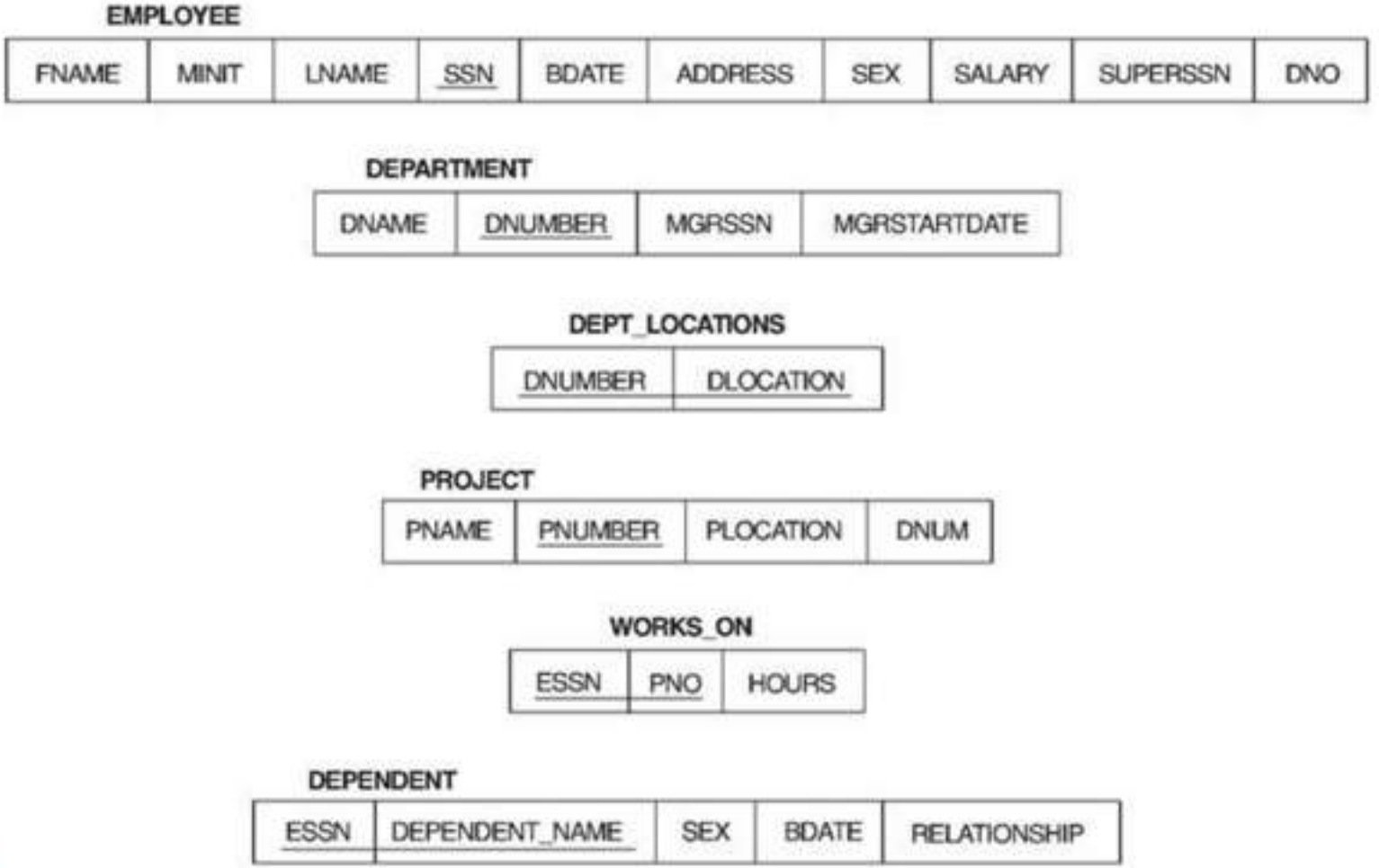
Roll_no	Marks
1	34
1	33
1	56
1	12

Referential Integrity Constraint

The value in the foreign key column FK of the referencing relation R1 can be either:

- **(1) a value of an existing primary key value of the corresponding primary key PK in the referenced relation R2,, or..**
- **(2) a null.**

Schema diagram for the COMPANY relational database schema; the primary keys are underlined.



Querying Relational Data Database Languages

Data definition language (DDL)

Storage definition language (SDL)

View definition language (VDL)

Data manipulation language (DML)

- High-level or nonprocedural DML or **set-at-a-time or set-oriented DMLs**
- Low-level or procedural DML or **record-at-a-time DMLs**

DDL

Once the design of a database is completed and a DBMS is chosen to implement the database, the first order of the day is to specify conceptual and internal schemas for the database and any mappings between the two.

In many DBMSs where no strict separation of levels is maintained, one language, called the **data definition language (DDL), is used by the DBA and by database designers to define** both schemas.

SDL

In DBMSs where a clear separation is maintained between the conceptual and internal levels, the DDL is used to specify the conceptual schema only.

Another language, the **storage definition language (SDL), is used to specify the internal schema.**

The mappings between the two schemas may be specified in either one of these languages.

VDL

For a true three-schema architecture, we would need a third language, the **view definition language (VDL), to specify user views and their mappings to the** conceptual schema, but in most DBMSs the DDL is used to define both conceptual and external schemas.

DML

Once the database schemas are compiled and the database is populated with data, users must have some means to manipulate the database.

Typical manipulations include retrieval, insertion, deletion, and modification of the data.

The DBMS provides a **data manipulation language (DML) for these** purposes.

Types of DML

A **high-level or nonprocedural DML can be used on its own to** specify complex database operations in a concise manner. High-level DMLs, such as SQL, can specify and retrieve many records in a single DML statement and are hence called **set-at-a-time or set-oriented DMLs.**

A **low-level** or **procedural DML** typically retrieves individual records or objects from the database and processes each separately. Low-level DMLs are also called **record-at-a-time DMLs because of this** property.

Comprehensive integrated language

In current DBMSs, the preceding types of languages are usually *not considered distinct languages;* rather, a comprehensive integrated language is used that includes constructs for conceptual schema definition, view definition, and data manipulation.

A typical example of a comprehensive database language is the SQL relational database language, which represents a combination of DDL, VDL, and DML, as well as statements for constraint specification and schema evolution.

VIEWS

Introduction

After a table is created and populated with data, it may become necessary to prevent all users from accessing all columns of a table, for data security reasons.

Consider the employee table

Employee(fname, lname,SSN, Bdate address,dno, salary, supSSN)

The company does not want to show the salary of employee to others.

For that reason, company will create a view for the others that view will contain all data except the salary of employee.

Introduction

Views are logical tables of data extracted from existing tables.

Views do not store any data.

It can be queried just like a table, but does not require disk space .

View stores it ‘s definition in Oracle system catalog.

A view contains no data of its own but it like a window through which data from tables can be viewed or changed.

When a reference is made to a view, its definition is scanned, the base table is opened and the view created on the top of the base table.

It can be used to hide sensitive columns. To do this the owner of the view must grant the select privilege on the view and revoke the privileges on the underlying tables.

Views

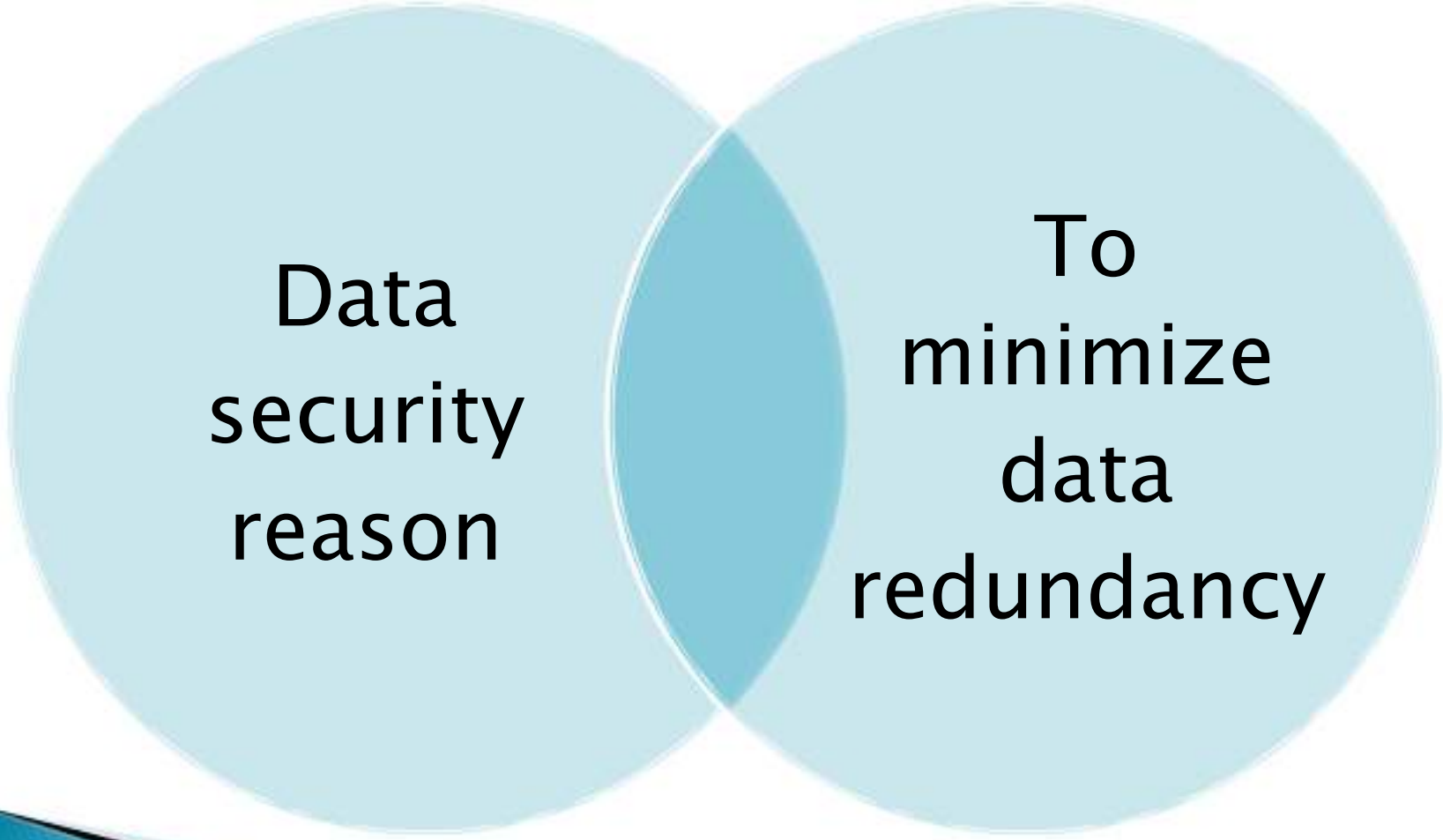
A **view** provides a mechanism to hide certain data from the view of certain users.

A view is a **virtual** table.

Some views are used for looking at table data, called as read-only view.

A view that is used to look at table data as well as insert, update and delete table data is called an updatable view.

Reasons why views are created



Advantage of View

- Data independence
- Improved security
- **Reduced complexity**: Instead of forcing your users to learn the SQL JOIN syntax you might wish to provide a view that runs a commonly requested SQL statement.
  Convenience
- **Customization:** If you wish to display some computed values or column names formatted differently than the base table columns, you can do so by creating views.

Disadvantage of View

1. **Update restriction**
2. **Structure restriction** (new attribute in the base table)
3. **Performance :** Even though views can be a great tool for securing and customizing data, they can be slow.

Creation of views

- **Create** view view-name AS
- **SELECT** column-name, column-name
- **FROM** table-name
- **WHERE** column_name =expression list;

Example

- Create view v_emp AS
- Select fname, lname,SSN, Bdate, address,dno, supSSN from employee;

- Create view v_emp AS
- Select * from employee;

To display the view

- Select *from emp_view;

To view the columns in a view

- Desc view_name

Renaming the columns of a view

- Create view v_emp AS
- Select fname, lname,SSN, Bdate, address add1 ,dno dnumber, supSSN from employee;

Selecting a data set from a view

- Once a view has been created, it can be queried exactly like a base table.

```
select column-name, column_name
FROM view_name;
```

Updateable view

Views on which data manipulation can be done are called updateable views.

Views defined from single table

If the user wants to INSERT records with the help of a view, then the PRIMARY KEY columns and all the NOT NULL columns must be included in the view.

The user can UPDATE, DELETE records with the help of a view even if the PRIMARY KEY column and NOT NULL columns are excluded from the view definition.

INSERT, UPDATE, DELETE

Insert into view_name values (… .....);

Update view_name set bdate=…. Where fname=…..;

Delete from view name where fname=……;

Common restrictions on updateable views

For the view to be updateable the view definition must not include:

- **Aggregate functions**
- **DISTINCT, GROUP BY or HAVING clause**
- **Sub-queries**
- **Constants, strings or value expression like price*.5**
- **UNION, INTERSECT or MINUS clause**

Destroying a view

Drop view
view_name;

Relational Algebra

Relational Algebra

Relational algebra is a procedural query language.

A query language is a language in which a user requests information from the database.

The basic set of operations for the relational model is known as the relational algebra.

Unary Relational Operations

These operations are called unary operations, because they operate on one relation.

select: σ(sigma)

project: ∏(pi)

rename: *ρ (rho)*

SELECT Operation

SELECT operation is used to select a subset of the tuples from a relation that satisfy a **selection condition**.

Syntax: $\sigma_{\text{<selection condition>}}(R)$

where the symbol **σ** (sigma) is used to denote the select operator. The selection condition appears as a subscript to *σ.*

Write the Relational Algebra expression

- Select those tuples of the loan relation where the branch is “Perryridge”
- Find tuples in which the amount lent is more than $1200.
- Find those tuples pertaining to loans of more than $1200 made by the Perryridge branch.

PROJECT Operation

This operation selects certain columns from the table and discards the other columns.

Syntax: $\Pi_{<attribute\ list>}(R)$

where Π (pi) is the symbol used to represent the project operation and <attribute list> is the desired list of attributes from the attributes of relation R.

Write the Relational Algebra expression

1. List all loan numbers and the amount of the Loan.

Loan Relation

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

Give the Relational Algebra expression

1. Find those customers who live in Harrison.

customer relation

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

Rename Operation

The rename operation is represented by the lowercase Greek letter ρ (rho).

Renaming operation is very useful in the case of union and join operation.

In improves the readability and better understandability.

We rename relations as well as the attribute.

Syntax for rename:

$\rho_{S(B1, B2, ..., Bn)}(R)$ is a renamed relation S based on R with column names $B_1, B_2, ,.....B_n$.

$\rho_S(R)$ is a renamed relation S based on R.

$\rho_{(B1, B2, ..., Bn)}(R)$ is a renamed column names $B_1, B_2, ,.....B_n$ based on R.

Rename Operation

ρ_{EMP1} (EMP)

In this example we only rename the relation from EMP to EMP1.

$\rho_{EMP1(Name1,dept1)}$ (EMP)

In this example we rename EMP as EMP1 and attributes Name and dept as Name1 and dept1.

$_{(Name1,dept1)}$ (EMP)

In this example we rename the attributes only.

EMP

Name	dept
Animesh	CSE
Jyoti	CSE
Kalpna	IT
Anamika	IT

After Rename

EMP1

Name 1	Dept 1
Animesh	CSE
Jyoti	CSE
Kalpna	IT
Anamika	IT


THANK
YOU

UNIT-3

What is SQL

- When a user wants to get some information from a database file, he can issue a **query**.
- A query is a user–request to retrieve data or information with a certain condition.
- SQL is a query language that allows user to specify the conditions.

DDL (Data Definition Language)

Data Definition Language (DDL) statements are used to define the database structure or schema. Some examples:

- CREATE – to create objects in the database
- ALTER – alters the structure of the database
- DROP – delete objects from the database
- TRUNCATE – remove all records from a table, including all spaces allocated for the records are removed
- RENAME – rename an object

DML (Data Manipulation Language)

- **Data Manipulation Language** (DML) statements are used for managing data within schema objects.

Some examples:

- SELECT - retrieve data from the a database
- INSERT - insert data into a table
- UPDATE - updates existing data within a table
- DELETE - deletes all records from a table, the space for the records remain

INSERT COMMAND

Des:

This command is used to enter (input) data into the created table.

Syntax:

INSERT INTO
<TableName> (<ColumnName1>,<ColumnName2>)
VALUES(<Expression1>,<Expression2>);

Ex:

INSERT INTO
STUDENT(ROLL_NO,NAME,BRANCH,PERCENT)
VALUES('CS05111', 'AAA', 'CO.SC', 84.45);

Modification of the Database – Insertion

- Add a new tuple to *account*

  insert into *account*

  values ('A-9732', 'Perryridge', 1200)

  or equivalently

  insert into *account* (*branch_name, balance, account_number*)

  values ('Perryridge', 1200, 'A-9732')

- Add a new tuple to *account* with *balance* set to null

  insert into *account*

  values ('A-777','Perryridge', *null*)

Modification of the Database – Updates

- Increase all accounts with balances over $10,000 by 6%, all other accounts receive 5%.
  - Write two **update** statements:

update *account*
set *balance* = *balance* * 1.06
where *balance* > 10000

update *account*
set *balance* = *balance* * 1.05
where *balance* ≤ 10000

Case Statement for Conditional Updates

- Same query as before: Increase all accounts with balances over $10,000 by 6%, all other accounts receive 5%.

```
update account
set balance = case
                when balance <= 10000
then balance *1.05
                else balance * 1.06
              end
```

DCL (Data Control Language)

Data Control Language (DCL) statements. Some examples:

- GRANT – to allow specified users to perform specified tasks.
- REVOKE – to cancel previously granted or denied permissions.
- COMMIT – save work done
- ROLLBACK – restore database to original since the last COMMIT

- The following privileges can be GRANTED TO or REVOKED FROM a user or role:
- SELECT
- INSERT
- UPDATE
- DELETE

Priviledge

The rights that allow the use of some or all Oracle's resources on the server are called Priviledge.

Granting the Priviledge

Objects that are created by the user are owned and controlled by the user. If a user wishes to access any of the objects belonging to other user , the owner of the object will have to give the permissions for such access. This is called Granting the Priviledge.

Revoking of Priviledges

Privileges once given can be taken by the owner of the object . This is called evoking of priviledge.

SQL GRANT Command

- SQL GRANT is a command used to provide access or privileges on the database objects to the users.

The Syntax for the GRANT command is:

- GRANT <privilege_name>
  ON <object_name >
  TO {user_name }
  [WITH GRANT OPTION];

- ***privilege_name*** is the access right or privilege granted to the user. Some of the access rights are ALL, EXECUTE, and SELECT.
- ***object_name*** is the name of an database object like TABLE, VIEW, STORED PROC and SEQUENCE.
- ***user_name*** is the name of the user to whom an access right is being granted.
- ***user_name*** is the name of the user to whom an access right is being granted.
- ***WITH GRANT OPTION*** – allows a user to grant access rights to other users.

SQL REVOKE Command:

- The REVOKE command removes user access rights or privileges to the database objects.
- The Syntax for the REVOKE command is:
- REVOKE privilege name
  ON object name
  FROM {username }

- The set operations **union**, **intersect**, and **except** operate on relations and correspond to the relational algebra operations ∪, ∩, −.
- Each of the above operations automatically eliminates duplicates; to retain all duplicates use the corresponding multi set versions **union all**, **intersect all** and **except all**.

Set Operations

eg. 15 Find all customers who have a loan, an account, or both:

(select *customer_name* **from** *depositor*)
union
(select *customer_name* **from** *borrower)*

eg. 16 Find all customers who have both a loan and an account.

(**select** *customer_name* **from** *depositor*)
intersect
(select *customer_name* **from** *borrower)*

eg. 17 Find all customers who have an account but no loan.

(**select** *customer_name* **from** *depositor*)
except
(select *customer_name* **from** *borrower)*

Aggregate Functions

- These functions operate on the multiset of values of a column of a relation, and return a value

  avg: average value
  min: minimum value
  max: maximum value
  sum: sum of values
  count: number of values

Aggregate Functions (Cont.)

eg. 18 Find the average account balance at the Perryridge branch.

```
select avg (balance)
        from account
        where branch_name = 'Perryridge'
```

- **eg. 19** Find the number of tuples in the *customer* relation.

```
select count (*) from customer
```

- **eg. 20** Find the number of depositors in the bank.

```
select count (distinct customer_name)
        from depositor
```

Group by clause

- This is used to produce summary information for a subset of the table .
- This will produce a single row for all the rows that met a particular condition.

Employee

ID	F_NAME	L_NAME	DOB	SALARY
1	vaibhav	bharade	19-OCT-90	30000
2	pradeep	verma	19-OCT-91	30000
3	navin	singh	19-OCT-92	30000
4	vibhu	bharade	19-OCT-93	30000
5	anish	bharadwaj	19-OCT-94	30000
6	minu	choudhary	27-JUN-86	68888
7	ankita	choudhary	27-JAN-86	68888
1	vaibhav	bharade	19-OCT-80	5000
2	pradeep	verma	20-OCT-80	5000

Grouping

eg. 21 To sum the salary of each employee group by id.

```
SELECT id, sum(salary) FROM employee GROUP
BY id;
```

ID	SUM(SALARY)
1	35000
6	68888
2	35000
4	30000
5	30000
3	30000

Grouping

eg. 22 Find the average of the salary of each employee group by id.

```
SELECT id, avg(salary) FROM employee GROUP
BY id;
```

Result

ID	AVG(SALARY)
1	17500
6	68888
2	17500
4	30000
5	30000
3	30000

Grouping

eg. 23 Find the minimum of the salary of each employee group by id.

```
SELECT id, min(salary) FROM employee
group by id;
```

Result

ID	MIN(SALARY)
1	5000
6	68888
2	5000
4	30000
5	30000
3	30000

Grouping

eg. 24 Find the maximum of the salary of each employee group by id.

SELECT id, max(salary) FROM employee group by id;

Result

ID	MAX(SALARY)
1	30000
6	68888
2	30000
4	30000
5	30000

Grouping

eg. 25 List the number of id present in employee table.

SELECT id, count(id) FROM employee group by id;

ID	COUNT(ID)
1	2
6	1
2	2
4	1
5	1
3	1
7	1

Aggregate Functions – Having Clause

eg. 26 Find the names of all branches where the average account balance is more than $1,200.

select *branch_name,* **avg** (*balance*)
from *account*
group by *branch_name*
having avg (*balance*) > 1200

Note: predicates in the **having** clause are applied after the formation of groups whereas predicates in the **where** clause are applied before forming groups

Null Values

- It is possible for tuples to have a null value, denoted by *null*, for some of their attributes
- *null* signifies an unknown value or that a value does not exist.
- The predicate **is null** can be used to check for null values.

eg. 27 Find all loan number which appear in the *loan* relation with null values for *amount.*

select *loan_number*
from *loan*
where *amount* **is null**

Null Values and Aggregates

- Total all loan amounts

  select sum (*amount*)**from** *loan*
  - Above statement ignores null amounts
  - Result is *null* if there is no non-null amount
- All aggregate operations except **count(*)** ignore tuples with null values on the aggregated attributes.

Nested Sub queries

- SQL provides a mechanism for the nesting of sub queries.
- A **sub query** is a **select-from-where** expression that is nested within another query.
- A common use of sub queries is to perform tests for set membership, set comparisons, and set cardinality.

Example Query

eg. 28 Find all customers who have both an account and a loan at the bank.

```
select distinct customer_name
        from borrower
        where customer_name in (select customer_name
                                from depositor )
```

eg. 29 Find all customers who have a loan at the bank but do not have an account at the bank

```
select distinct customer_name
        from borrower
        where customer_name not in (select customer_name
                                    from depositor )
```

Example Query

eg. 30 Find all customers who have both an account and a loan at the Perryridge branch

```
select distinct customer_name
        from borrower, loan
        where borrower.loan_number = loan.loan_number and
          branch_name = 'Perryridge'  and
          (branch_name, customer_name ) in
                (select branch_name, customer_name
                  from depositor, account
                  where depositor.account_number =
                      account.account_number )
```

- Note: Above query can be written in a much simpler manner. The formulation above is simply to illustrate SQL features.

Set Comparison

eg. 31 Find all branches that have greater assets than some branch located in Brooklyn.

```
select distinct  T.branch_name
        from branch as T, branch as S
        where  T.assets > S.assets and
                S.branch_city = 'Brooklyn'
```

- Same query using > **some** clause

```
select branch_name
        from branch
        where assets > some
                (select assets
                 from branch
                 where branch_city = 'Brooklyn')
```

Example Query

eg. 32 Find the names of all branches that have greater assets than all branches located in Brooklyn.

```
select branch_name
        from branch
        where assets > all
                (select assets
                from branch
                where branch_city = 'Brooklyn')
```

Example Query

eg. 33 Find all customers who have an account at all branches located in Brooklyn.

```
select distinct S.customer_name
        from depositor as S
        where not exists (
                (select branch_name
                from branch
                where branch_city = 'Brooklyn')
                except
                (select R.branch_name
                from depositor as T, account as R
                where T.account_number = R.account_number and
                        S.customer_name = T.customer_name ))
```

With Clause

- The **with** clause provides a way of defining a temporary view whose definition is available only to the query in which the **with** clause occurs.

eg. 34 Find all accounts with the maximum balance

with *max_balance* (*value*) **as**
select max (*balance*)
from *account*
select *account_number*
from *account, max_balance*
where *account.balance = max_balance.value*

Complex Queries using With Clause

eg. 35 Find all branches where the total account deposit is greater than the average of the total account deposits at all branches.

```
with branch_total (branch_name, value) as
        select branch_name, sum (balance)
        from account
        group by branch_name
with branch_total_avg (value) as
        select avg (value)
        from branch_total
select branch_name
from branch_total, branch_total_avg
where branch_total.value >= branch_total_avg.value
```

DATABASE TRIGGERS

Aggregate Functions

- These functions operate on the multiset of values of a column of a relation, and return a value

  avg: average value
  min: minimum value
  max: maximum value
  sum: sum of values
  count: number of values

Aggregate Functions (Cont.)

eg. 18 Find the average account balance at the Perryridge branch.

```
select avg (balance)
        from account
        where branch_name = 'Perryridge'
```

- **eg. 19** Find the number of tuples in the *customer* relation.

```
select count (*) from customer
```

- **eg. 20** Find the number of depositors in the bank.

```
select count (distinct customer_name)
        from depositor
```

Group by clause

- This is used to produce summary information for a subset of the table .
- This will produce a single row for all the rows that met a particular condition.

Employee

ID	F_NAME	L_NAME	DOB	SALARY
1	vaibhav	bharade	19-OCT-90	30000
2	pradeep	verma	19-OCT-91	30000
3	navin	singh	19-OCT-92	30000
4	vibhu	bharade	19-OCT-93	30000
5	anish	bharadwaj	19-OCT-94	30000
6	minu	choudhary	27-JUN-86	68888
7	ankita	choudhary	27-JAN-86	68888
1	vaibhav	bharade	19-OCT-80	5000
2	pradeep	verma	20-OCT-80	5000

Grouping

eg. 21 To sum the salary of each employee group by id.

```
SELECT id, sum(salary) FROM employee GROUP
BY id;
```

Result

ID	SUM(SALARY)
1	35000
6	68888
2	35000
4	30000
5	30000
3	30000

Grouping

eg. 22 Find the average of the salary of each employee group by id.

```
SELECT id, avg(salary) FROM employee GROUP
BY id;
```

Result

ID	AVG(SALARY)
1	17500
6	68888
2	17500
4	30000
5	30000
3	30000

Grouping

eg. 23 Find the minimum of the salary of each employee group by id.

```
SELECT id, min(salary) FROM employee
group by id;
```

Result

ID	MIN(SALARY)
1	5000
6	68888
2	5000
4	30000
5	30000
3	30000

Grouping

eg. 24 Find the maximum of the salary of each employee group by id.

SELECT id, max(salary) FROM employee group by id;

Result

ID	MAX(SALARY)
1	30000
6	68888
2	30000
4	30000
5	30000

Grouping

eg. 25 List the number of id present in employee table.

SELECT id, count(id) FROM employee group by id;

ID	COUNT(ID)
1	2
6	1
2	2
4	1
5	1
3	1
7	1

Aggregate Functions – Having Clause

eg. 26 Find the names of all branches where the average account balance is more than $1,200.

select *branch_name,* **avg** (*balance*)
from *account*
group by *branch_name*
having avg (*balance*) > 1200

Note: predicates in the **having** clause are applied after the formation of groups whereas predicates in the **where** clause are applied before forming groups

Introduction

- The Oracle server allows the user to define procedures that are implicitly executed, when an insert, update or delete is issued against a table.
- These procedures are called database triggers.
- The major issue that make these triggers stand-alone is that they are fired implicitly(i.e. internally) by the Oracle server itself and not explicitly called by the user.

Use of database triggers

- A trigger can permit DML statements against a table only if they are issued, during regular business hours or on predetermined weekdays.
- A trigger can also be used to keep an audit trail of a table along with the operation performed and the time on which the operation was performed.
  - It can be used to prevent invalid transactions.
  - Enforce complex security authorizations.

How to apply database triggers

- A trigger has three basic parts
  - A triggering event or statement
  - A trigger restriction
  - A trigger action

A triggering event or statement

- It is a SQL statement that causes a trigger to be fired.
- It can be INSERT, UPDATE or DELETE statement for a specific table.

A trigger restriction

- A trigger restriction specifies a Boolean (logical) expression that must be TRUE for the trigger to fire.
- It is an option available for triggers that are fired for each row.
- Its function is to conditionally control the execution of a trigger.
- A trigger restriction is specified using a WHEN clause.

A trigger action

- A trigger action is the PL/SQL code to be executed when a triggering statement is encountered and any trigger restriction evaluates to TRUE.
- The PL/SQL block can contain SQL and PL/SQL statements, can define PL/SQL language constructs and can call stored procedures.

Row triggers

- A row trigger is fired each time a row in the table is affected by the triggering statement.
- For example, if an UPDATE statement updates multiple rows of a table, a row trigger is fired once for each row affected by the UPDATE statement.
- If the triggering statement affects no rows, the trigger is not executed at all.
- Row triggers should be used when some processing is required whenever a triggering statement affects a single row in a table.

Statement triggers

- A statement trigger is fired once on behalf of the triggering statement, independent of the number of rows the triggering statement affects.
- Statement triggers should be used when a triggering statement affects rows in a table but the processing required is completely independent of the number of rows affected.

Before V/s after triggers

- When defining a trigger it is necessary to specify the trigger timing, i.e. specifying when the triggering action is to be executed in relation to the triggering statement.
- Before and after apply to both row and the statement triggers.

Before triggers

- Before triggers execute the trigger action before the triggering statement. These types of triggers are commonly used in the following situation:
  - Before triggers are used when the trigger action should determine whether or not the triggering statement should be allowed to complete. By using a before trigger, one can eliminate unnecessary processing of the triggering statement.
  - Before triggers are used to drive specific column values before completing a triggering INSERT or UPDATE statement.

After triggers

- AFTER trigger executes the trigger action after the triggering statement is executed. These types of triggers are commonly used in the following situation:
  - AFTER triggers are used when you want the triggering statement to complete before executing the trigger action.
  - If a before trigger is already present, an AFTER trigger can perform different actions on the same triggering statement.

Combinations triggers

- **BEFORE statement trigger**: before executing the triggering statement, the trigger action is executed.
- **BEFORE row trigger**: before modifying each row affected by the triggering statement and before appropriate integrity constraints, the trigger is executed if the trigger restriction either evaluated to TRUE or was not included.
- **AFTER statement trigger:** after executing the triggering statement and applying any deferred integrity constraints, the trigger action is executed.
- **AFTER row trigger:** after modifying each row affected by the triggering statement and possibly applying appropriate integrity constraints, the trigger action is executed for the current row if the trigger restriction either evaluates to TRUE or was not included. Unlike BEFORE row triggers, AFTER row triggers have rows locked.

Creating a Triggers

CREATE OR REPLACE TRIGGER triggername
{ BEFORE, AFTER}
{DELETE, INSERT,UPDATE [OF column,…..]}
ON tablename
[REFERENCING {OLD AS old, NEW AS new}]
[FOR EACH ROW [WHEN condition]]

Keyword and parameters

- **OR REPLACE**: recreates the trigger if it already exists. This option can be used to change the definition of an existing trigger without first dropping it.
- **Triggername:** is the name of the trigger to be cretated.
- **BEFORE**: indicates that the Oracle server fires the trigger before executing the triggering statement.
- **AFTER:** indicates that the Oracle server fires the trigger after executing the triggering statement.

Keyword and parameters

- **DELETE**: indicates that the Oracle server fires the trigger whenever a DELETE statement removes a row from the table.
- **INSERT:** indicates that the Oracle server fires the trigger whenever an INSERT statement adds a row to table.
- **UPDATE:** indicates that the Oracle server fires the trigger whenever an UPDATE statement changes a value in any column of the table.

Keyword and parameters

- **ON:** Specifies the name of the table, which the trigger is to be created.
- **REFERENCING:** specifies correlation names.
- **FOR EACH ROW:** designates the trigger to be a row trigger.
- **WHEN:** specifies the trigger restriction.

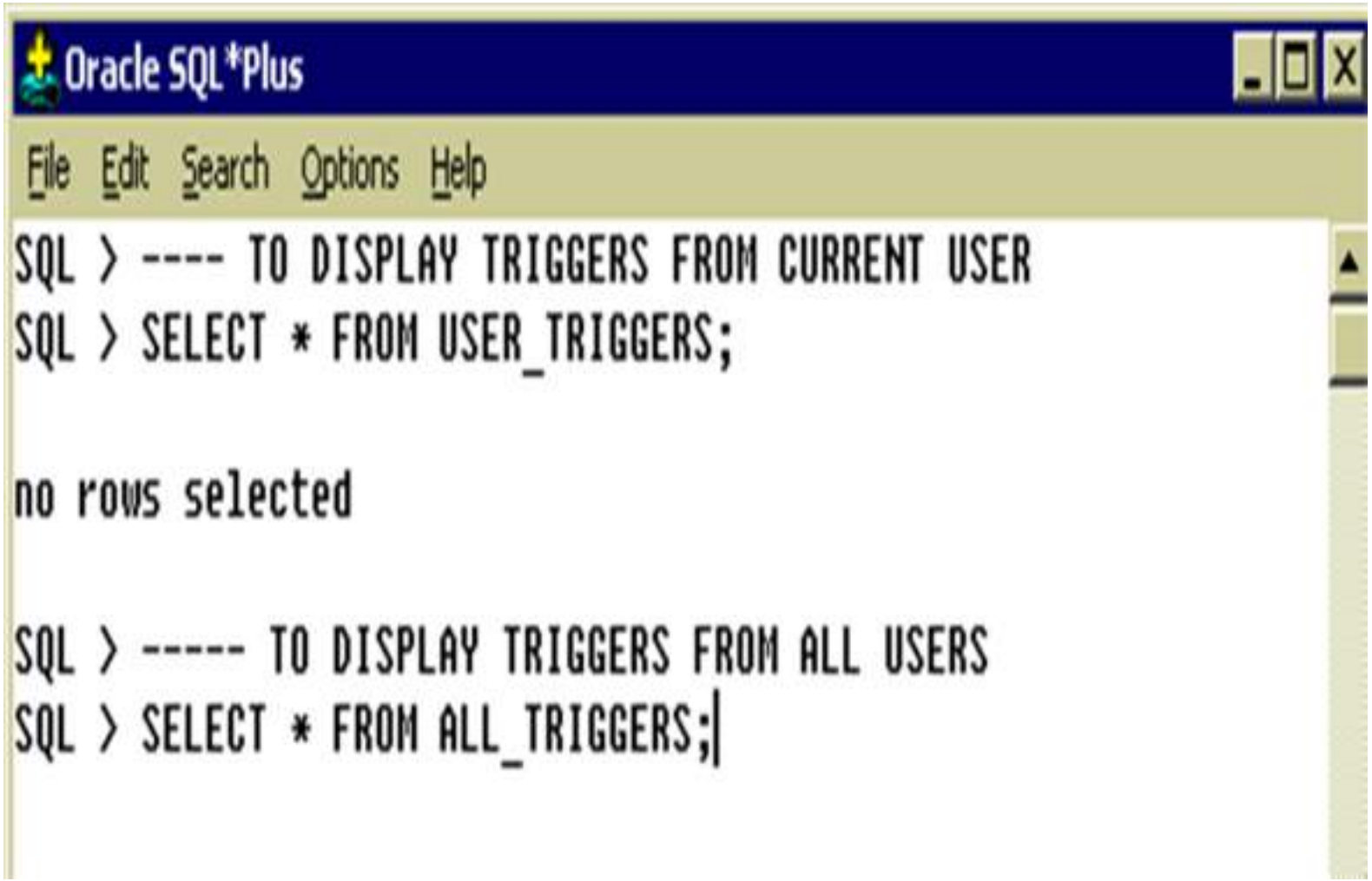
Creating Triggers

- CREATE OR REPLACE TRIGGER Print_salary_changes BEFORE DELETE OR INSERT OR UPDATE ON Emp_tab FOR EACH ROW WHEN (new.Empno > 0)

```
  DECLARE sal_diff number;
  BEGIN sal_diff := :new.sal - :old.sal;
dbms_output.put('Old salary: ' || :old.sal);
dbms_output.put(' New salary: ' || :new.sal);
dbms_output.put_line(' Difference ' ||
sal_diff);
  END;
```

Points to ponder

- A trigger cannot include COMMIT, SAVEPOINT and ROLLBACK.
- We can use only one trigger of a particular type .
- A table can have any number of triggers.


Oracle SQL*Plus
File Edit Search Options Help
SQL > ---- TO DISPLAY TRIGGERS FROM CURRENT USER
SQL > SELECT * FROM USER_TRIGGERS;
no rows selected
SQL > ----- TO DISPLAY TRIGGERS FROM ALL USERS
SQL > SELECT * FROM ALL_TRIGGERS;

Viewing Defined Triggers

To view a list of all defined triggers, use:

- select trigger_name from user_triggers;
- For more details on a particular trigger:
- select trigger_type, triggering_event, table_name, referencing_names, trigger_body from user_triggers where trigger_name = '<trigger_name>';

Disabling Triggers

- To disable or enable a trigger:
- alter trigger <trigger_name> {disable|enable};

Schema Refinement Problems caused by Redundancy

INFORMAL DESIGN GUIDELINES FOR RELATIONAL SCHEMAS

- Semantics of attributes.
- Reducing the redundant values in tuples.
- Reducing the null values in tuples.
- Disallowing the possibility of generating spurious tuples.

Semantics of attributes

- Semantics – specifies how to interpret the attribute values stored in a tuple of the relation.
- Name of the attribute must have some meaning.
- Relationship among the relations must be clear.

Employee	Emp_id	Emp_name	Street	City
Company	Company_id	Company_name	C_City	
Works	Emp_id	Company_id	Salary	
Manages	Manager_id	Manager_name	Emp_id	

GUIDELINES 1:

- Design a relation schema so that it is easy to explain its meaning.
- Do not combine attributes from multiple entity types into a single relation.

Reducing the redundant values in tuples

- One goal of schema design is to minimize the storage space that the base relations occupy.
- Grouping attributes into relation has a significant effect on storage space.
- Mixing attributes of multiple entities may cause problems
- Information is stored redundantly wasting storage
- Problems with update anomalies
  - Insertion anomalies
  - Deletion anomalies
  - Modification anomalies

Emp_com

Emp_id	Emp_name	Street	City	Company_id	Company_name	C_City

Combining the 2 relations ‘Employee’ and ‘Company’ into one relation namely: ‘Emp_com’

Insertion Anomalies

Insertion anomalies can be differentiated into two types:

- To insert a new tuple into emp_com, either include the attribute values for the company that the employee works for, or null if the employee does not work for a company yet.
- To insert a new company that has no employees as yet in the emp_com relation. Place the null values in the attributes for employee.

Deletion Anomalies

- If we delete from emp_com, an employee tuple that represent the last employee working for a particular company, the information concerning that company, is lost from the database.

Modification Anomalies

In Emp_com if the value of the attributes for a particular company is to be changed, the update is required for all tuples of 'emp_com' relation who work in that company, otherwise database will become inconsistent.

GUIDELINE 2 : Design a schema that does not suffer from the insertion, deletion and update anomalies. If there are any present, then ensure that applications that update the database will operate correctly.

Null Values in Tuples

If many of the attributes do not apply to all tuples in the relation , we end up with many nulls in those tuples. Nulls can have multiple interpretations such as,

- Attribute not applicable or invalid
- Attribute value unknown (may exist)
- Value known to exist, but unavailable

For example, if only 10 percent of employees have individual offices, there is little justification for including an attribute office_number in the employee relation.

GUIDELINE 3: Relations should be designed such that their tuples will have as few NULL values as possible

Generation of SpuriousTuples

- At the time of joining of two tables the extra tuples which are included in the join are extra, i.e. which are not required are spurious tuples or dangling tuples or wrong tuple.

GUIDELINE 4 : design relation schemas so that they can be joined with equality conditions on attributes that are either primary keys or foreign keys in a way that guarantees that no spurious tuples are generated.

Functional Dependencies

- Functional dependencies (FDs) are used to specify formal measures of the "goodness" of relational designs.
- A Functional dependency define the properties of the database schema.
- FDs are **constraints** that are derived from the meaning and interrelationships of the data attributes.
- An FD is a property of the attributes in the schema R.
- The constraint must hold on *every relation instance* r(R).

FUNCTIONAL DEPENDENCIES

- A Functional dependency denoted by X → Y between two sets of attributes X and Y that are subsets of R specifies a constraint on the possible tuple that can form a relation state r of R.
- The constraint is for any two tuples t1 and t2 in r if t1[X] = t2[X] then they have t1[Y] = t2[Y].
- This means the value of X component of a tuple uniquely determines the value of component Y.

Full functional dependency

- For a relation schema R and a FD X->Y is a **full functional dependent** if you remove any attribute from X the dependency does not hold any more.
- {X-A}-> Y is no longer true.

Partial dependency

- A **partial dependency** X → Y, then there is some attribute on X that can be removed from X and yet the dependency stills holds.

Transitive dependency

- Given a relation R with the functional dependencies F, if there a set of attribute Z that are neither a primary or candidate key and both X → Y and Y → Z holds, , then Z is **transitively dependent** on X.
- (X->Z) is transitive dependent

TRIVIAL AND NON-TRIVIAL FD

- If X -> Y hold and X is a superset of Y, then this is called a Trivial Functional Dependency. All FDs which are not trivial is called Non-trivial FD.

Example:

- {Employee ID, EmployeeAddress} → {EmployeeAddress} is trivial.

PRIME & NON-PRIME ATTRIBUTES

If an attribute is a candidate key or a subset of candidate key then it is called a prime attribute.

For ex,

A	B	C	D

'A' is a candidate key.

Combination of C & D is a candidate key.

Hence; A, C, D are prime attributes and B is a non-prime attribute.

Inference Rules for FDs(Armstrong Axioms)

- Axioms, or rules of inference, provide a simpler technique for reasoning about functional dependencies.
- **Armstrong's axioms** are a set of axioms (inference rules) used to infer all the functional dependencies on a relational database.
- The very first Armstrong is the person who gives a set of inference rules.

Armstrong's inference rules

The following six rules (IR1 through IR6) are well-known inference rules for FDs.

1. IR1 – Reflective rule: $Y \subseteq X$ then $X \rightarrow Y$.
2. IR2 – Augmentation rule: $X \rightarrow Y$, then $XZ \rightarrow YZ$.
3. IR3 – Transitive rule: $\{X \rightarrow Y, Y \rightarrow Z\}$ then $X \rightarrow Z$

Inference Rules for FDs (contd...

Some **additional inference rules** that are useful:

(**Union**) If X -> Y and X -> Z, then X -> YZ

(**Decomposition**) If X -> YZ, then X -> Y and X -> Z

(**Psuedotransitivity**) If X -> Y and WY -> Z, then WX -> Z

Questions on inference rules for FD

1. Let set of FDs are given

F={ A->B,C->X, BX->Z}

Prove or disprove that AC->Z

Sol n:

C->X and BX->Z then **CB->Z**

By rule A6: Pseudo transitivity

A->B(given) and **CB->Z then AC->Z**

By rule A6: Pseudo transitivity

Questions on inference rules for FD

2. Let set of FDs are given

F={ A->B,C->D, C is subset of B}

Prove or disprove that A->C

Sol n:

C is subset of B then **B->C**

By rule A1: Reflexivity

A->B(given) and **B->C then A-> C**

By rule A3: Transitivity

Questions on inference rules for FD

3. Let set of FDs are given

F={ A->B,BC->D}

Prove or disprove that AC->D

Sol n:

A->B and BC->D then **AC->D**

By rule A6: Pseudo transitivity

Questions on inference rules for FD

4. Let set of FDs are given

F={ A->B,BC->D}

Prove or disprove that AD-> B

Sol n:

A->B then **AD->BD**

By rule A2: Augmentation

AD->BD then AD->B and AD->D

By rule A5: Decomposition

Questions on inference rules for FD

5. Let set of FDs are given

F={ A->B,A->C,BC->D}

Prove that A->D

Sol n:

A->B and A->C then **A->BC**

By rule A4: union

A->BC and BC->D(given) **then A->D**

By rule A3: Transitivity

UN-NORMALIZED RELATION

A table is said to be un-normalized if each row contains multiple set of values for some of the columns, these multiple values in a single row are also called non-atomic values.

UN-NORMALIZED RELATION

SID	Sname	Saddress	Phoneno	DOB
101	Animesh	Bhopal	276666	17-08-87
102	Ravi	Delhi	566666	25-06-78
103	Nishant	Indore	665555 453333 656566	04-3-56
104	Ganesh	Bhopal	555555 876544	12-5-78

Introduction to Normalization

- **Normalization** is the process of decomposing unsatisfactory "bad" relations by breaking up their attributes into smaller relations while ensuring data integrity, eliminating data redundancy and minimizing the insertion, deletion and update anomalies.

Objective of normalization

- To make it feasible to represent any relation in the database.
- To free relations from undesirable insertion, update and deletion anomalies.

Benefits of Normalization

- Improve storage efficiency
- Quicker updates
- Less data inconsistency
- Clearer data relationships
- Easier to add data
- Flexible Structure

Normal form

- Basically the normal form of data indicate how much redundancy is in that data.
- Types of normal form
  - 1NF
  - 2NF
  - 3NF
  - BCNF **(Boyce-Codd Normal Form)**
  - 4NF
  - 5NF (PJNF)
  - DKNF(Domain Key Normal Form)

Normal Forms

1NF	*Keys; No repeating groups*
2NF	*No partial dependencies*
3NF	*No transitive dependencies*
BCNF	*Determinants are candidate keys*
4NF	*No multivalued dependencies*

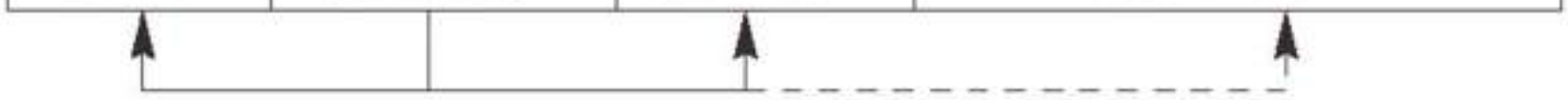
First Normal Form

- It states that the value of any attribute in any relation must be a **single atomic value**.
- In other words, only one value is associated with each attribute and the value is not a set of values or a list of values (No composite Attributes).

(a)

DEPARTMENT

DNAME	DNUMBER	DMGRSSN	DLOCATIONS



(b)

DEPARTMENT

DNAME	DNUMBER	DMGRSSN	DLOCATIONS
Research	5	333445555	{Bellaire, Sugarland, Houston}
Administration	4	987654321	{Stafford}
Headquarters	1	888665555	{Houston}

(c)

DEPARTMENT

DNAME	DNUMBER	DMGRSSN	DLOCATION
Research	5	333445555	Bellaire
Research	5	333445555	Sugarland
Research	5	333445555	Houston
Administration	4	987654321	Stafford
Headquarters	1	888665555	Houston

DNAME	DNUMBER	DMGRSSN	DLOCATION1	DLOCATION2	DLOCATION3
Research	5	333445555	Bellaire	Sugarland	Houston
Administration	4	987654321	Stafford	Null	Null
Headquarters	1	888665555	Houston	Null	Null

DNAME	DNUMBER	DMGRSSN
Research	5	333445555
Administration	4	987654321
Headquarters	1	888665555

DNUMBER	DLOCATION
5	Bellaire
5	Sugarland
5	Houston
4	Stafford
1	Houston

SECOND NORMAL FORM

- A relation schema R is in second normal form (2NF) if it is in the 1NF and if all non-prime attributes of R are fully functionally dependent on the primary keys.
- R can be decomposed into 2NF relations via the process of 2NF normalization.

Steps to convert 1NF to 2NF

- First find all candidate keys.
- Determine all the prime and nonprime attributes separately.
- Check that no nonprime attribute is partially dependent on any key.

Sales order relation

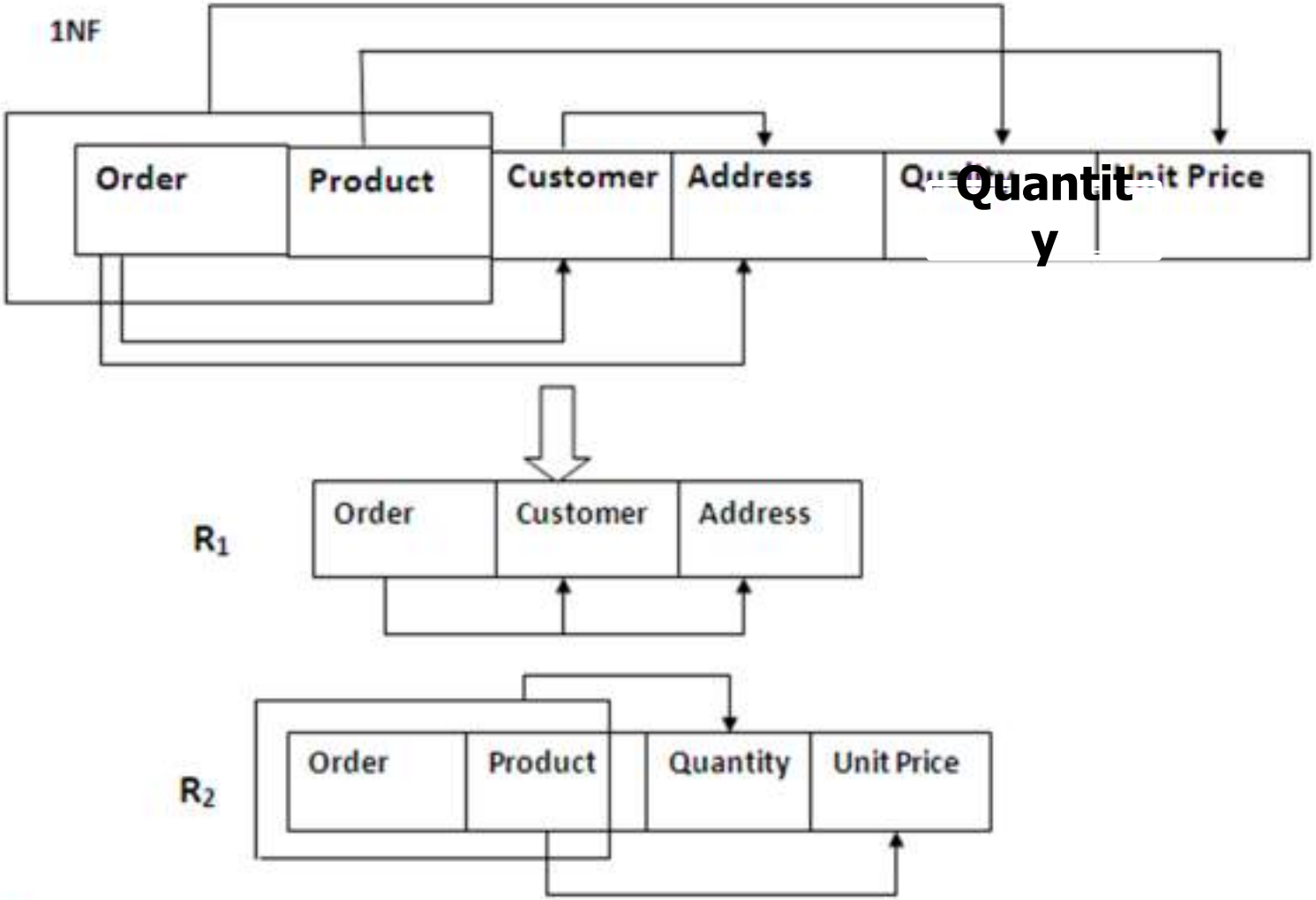
Order	Product	Customer	Address	Qua-ntity	Unit price
S1	P1	Animesh	Bhopal	300	500
S1	P2	Animesh	Bhopal	100	700
S2	P3	Ganesh	Raipur	500	200
S3	P4	Rahul	Guna	200	1000
S3	P1	Rahul	Guna	450	500
S4	P5	Nishant	Indore	400	900
S5	P2	Ranjit	Jabalpur	250	700

2 NF

- 2NF means no partial dependencies on the key. But we have
- {order}-> {customer , address}
- {product}-> {unit price}

So the relation is in 1NF but not in 2NF. So we convert the relation sale order in 2NF.

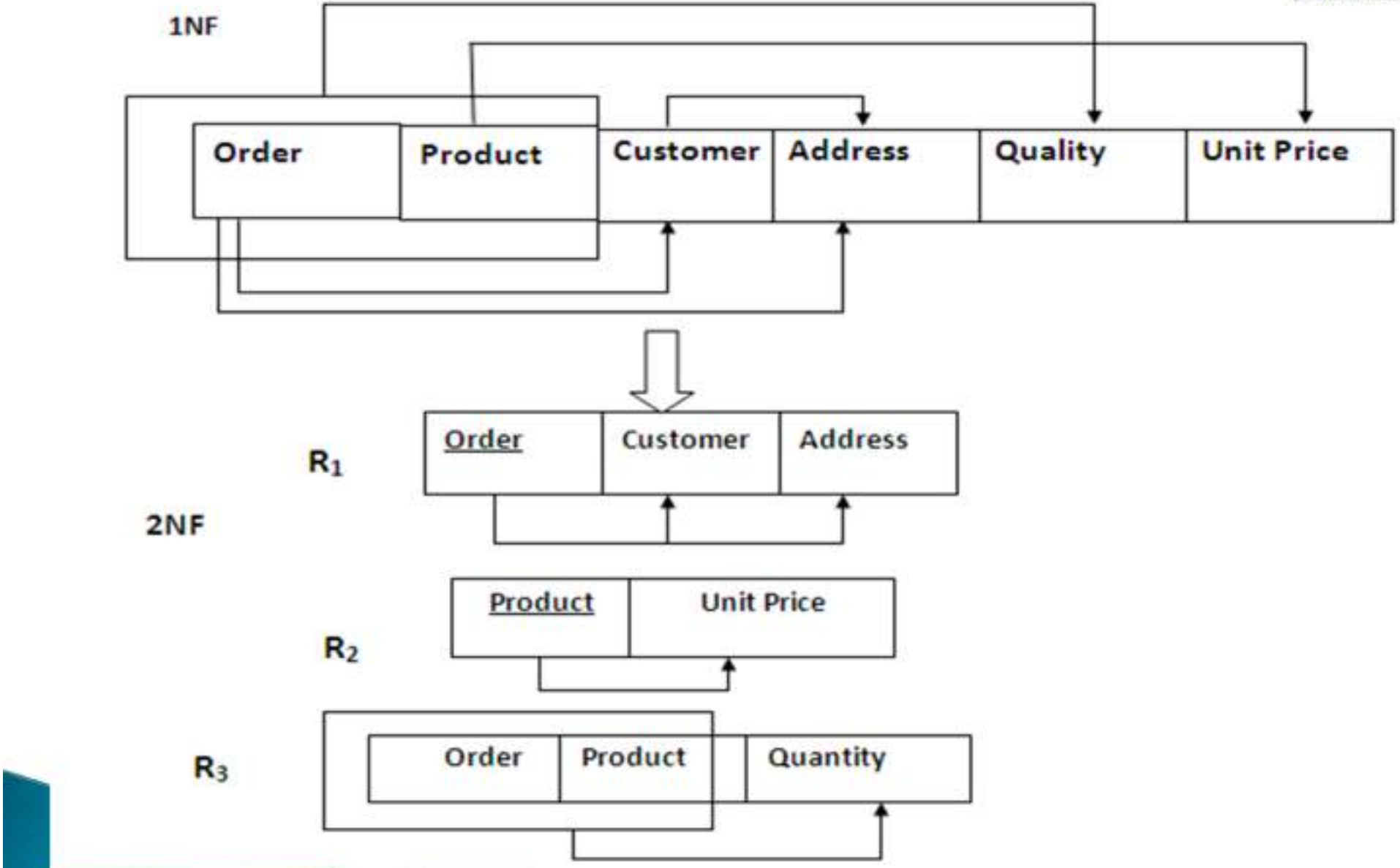
Conversion of 1NF to 2NF



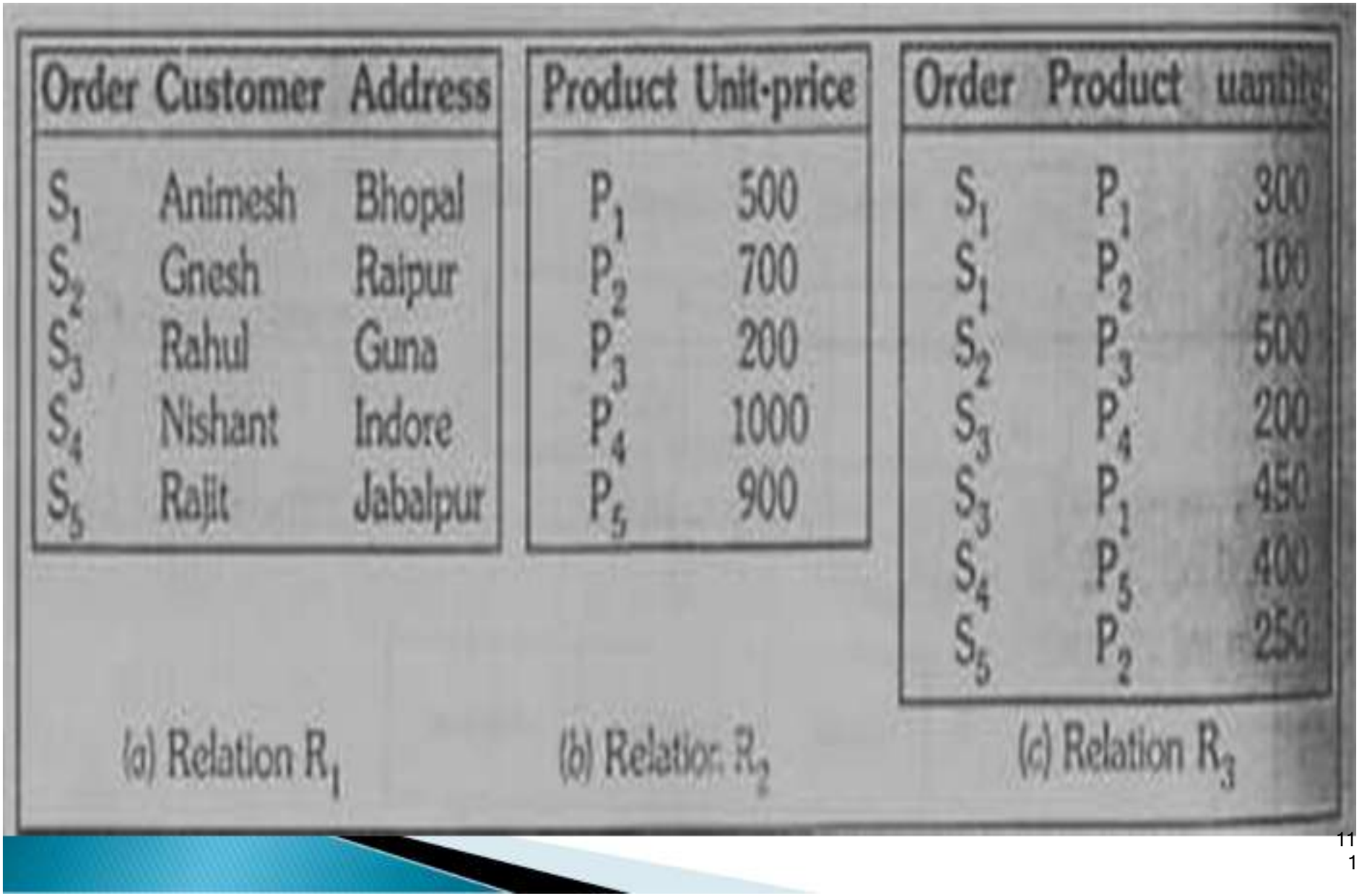
Continued.....

- R_1 is now in 2NF, but there is still partial FD in R_2.
- Product-> unit price

Relation in 2NF



Sample value



Order	Customer	Address
S_1	Animesh	Bhopal
S_2	Gnesh	Raipur
S_3	Rahul	Guna
S_4	Nishant	Indore
S_5	Rajit	Jabalpur

(a) Relation R_1

Product	Unit-price
P_1	500
P_2	700
P_3	200
P_4	1000
P_5	900

(b) Relation R_2

Order	Product	uantity
S_1	P_1	300
S_1	P_2	100
S_2	P_3	500
S_3	P_4	200
S_3	P_1	450
S_4	P_5	400
S_5	P_2	250

(c) Relation R_3

Lossy decomposition

- There should be no loss of information due to decomposition.

Lossless Join decomposition

Recombination using relational join produces exactly same as pre decomposition.

Lossless-Join

Condition for lossless join.

1) All attributes of an original schema (R) must appear in the decomposition(R_1 , R_2)

$R = R_1 \cup R_2$

2) At least one of the following functional dependencies holds

- If $R_1 \cap R_2 = R_1$
- If $R_1 \cap R_2 = R_2$

Dependency preservation

- Dependency preservation is another important requirement since a dependency is a constraint on the database and if X-> Y holds than we know that the two attributes are closely related and it would be useful if both attributes appeared in the same relation so that the dependency can be checked easily.

Dependency preservation

- Dependency preservation is another important requirement since a dependency is a constraint on the database and if X-> Y holds than we know that the two attributes are closely related and it would be useful if both attributes appeared in the same relation so that the dependency can be checked easily.

Dependency preservation Condition

- If R is decomposed into X and Y then

$$(F_x \cup F_y)^+ = F^+$$

Irreducible sets(minimal sets or canonical cover) of FD

- A canonical cover F_c for F is a set of dependencies such that F logically implies all dependencies in F_c, and F_c logically implies all dependencies in F.
- A set of FDs S is irreducible if and only if
  - The right-hand side (the dependent) of each FD in S involves just one attribute.
  - The left-hand side (the determinant) of each FD in S is irreducible.

Normalization

- Normalization is the process of putting one fact in one appropriate place.
- This optimizes updates at the expense of retrievals.
- When a fact is stored in only one place, then data retrieving is easy. But, When a fact is stored in different places, then retrieving many different but related facts usually requires going to many different places.
- This tends to slow the retrieval process.
- There are two strategies for dealing in this type of situation.

1st method

- The preferred method is to keep the logical design normalized, but allow the DBMS to store additional redundant information on disk to optimize query response.
- In this case it is the DBMS software's responsibility to ensure that any redundant copies are kept consistent.
- This method is often implemented in SQL as indexed views(Microsoft SQL) or materialized views(oracle).

2nd method

- The more usual approach is to denormalize the logical data design.
- With care this can achieve a similar improvement in query response, but at a cost-it is now the database designer's responsibility to ensure that the denormalized database does not become inconsistence.
- This is done by creating rules in the database called constraints, that specify how the redundant copies of information must be kept synchronized.

Denormalization

- A denormalized data model is not the same as a data model that has not been normalized, denormalization should only take place after a satisfactory level of normalization has taken and that any required constrainits and/ or rules have been created to deal with the inherent anomalies in the design.

Multivalued Dependencies

- A multivalued dependency (MVD) exists between two fields X and Y, when a distinct value of X is directly associated with two or more values of Y.
- An MVD is represented as X Y and can be read as:
  - "the value of X determines multiple values of Y"

    Or $\twoheadrightarrow$
  - "multiple values of Y are functionally dependent on the value of X.

Teaching database

Course	Book	Lecturer
AHA	Silberschatz	John D
AHA	Nederpelt	William M
AHA	Silberschatz	William M
AHA	Nederpelt	John D
AHA	Silberschatz	Christian G
AHA	Nederpelt	Christian G
OSO	Silberschatz	John D
OSO	Silberschatz	William M

Types of MVD

- A multi-valued dependency can be further defined as being trivial or nontrivial.
  - A MVD A ->> B in relation R is defined as being trivial if (a) B is a subset of A *or* (b) $A \cup B = R$.
  - A MVD is defined as being nontrivial if neither (a) nor (b) are satisfied.

Definition MVD

- The multivalued dependency X->> Y holds in a relation R if whenever we have two tuples of R that agree in all the attributes of X, then we can swap their Y components and get two new tuples that are also in R.
- If two tuples t_1 and t_2 exist in 'r' such that $t_1[X] = t_2[X]$, then two tuples t_3 and t_4 should also exist in 'r' with the following properties,
  - $t_1[X] = t_2[X] = t_3[X] = t_4[X]$
  - $t_1[Y] = t_3[Y]$ and $t_2[Y] = t_4[Y]$
  - $t_1[Z] = t_4[Z]$ and $t_2[Z] = t_3[Z]$

Inference Rules for Multivalued Dependencies

Rule 1: **Complementation Rule**

It states that if in a relation R, X->-> Y exists then X->->R-(X UY) is true.

Rule 2: **Augmentation Rule**

It states that if in a relation R, X->->Y and T $\subseteq$ R And P $\subseteq$ T, then TX->-> PY

Rule 3: **Transitive Rule**

If X->->Y and Y->->P, then X->->(P-Y)

Rule 4: **Replication Rule**

If X -> Y then X->> Y but not vice versa is not true .

Inference Rules for Multivalued Dependencies

Rule 5: **Coalescence Rule**

If X->->Y and $P \subseteq Y$, then there exists Q such that
(a) $Q \subseteq R$ (b) $Q \cap Y$ is empty (c) Q->P then X->P

Rule 6: **Union Rule**

If X->->Y and X->->P, then X->-> $P \cup Y$ or X->->PY

Rule 7: **Intersection Rule**

If X->->Y and X->->P, then X->-> $Y \cap P$

Rule 8: **Difference Rule**

If X->->Y and X->->P, then X->-> (Y – P) and X->-> (P- Y)

Fourth Normal Form (4NF)

A relation schema R is in **4NF** with respect to a set of dependencies F if,

- It is in BCNF and
- For every nontrivial multivalued dependency $X \text{—>>} Y$, X is a superkey — that is, X is either a candidate key or a superset thereof.

Pizza Delivery Permutations

Restaurant	Pizza Variety	Delivery Area
Vincenzo's Pizza	Thick Crust	Springfield
Vincenzo's Pizza	Thick Crust	Shelbyville
Vincenzo's Pizza	Thin Crust	Springfield
Vincenzo's	ThinCrust	Shelbyville
Elite Pizza	Thin Crust	Capital City
Elite Pizza	Stuffed Crust	Capital City
A1 Pizza	Thick Crust	Springfield
A1 Pizza	Thick Crust	Shelbyville
A1 Pizza	Thick Crust	Capital City
A1 Pizza	Stuffed Crust	Springfield
A1 Pizza	Stuffed Crust	Shelbyville
A1 Pizza	Stuffed Crust	Capital City

- The table has no non-key attributes because its only key is {Restaurant, Pizza Variety, Delivery Area}.
- The problem is that the table features two non-trivial multivalued dependencies on the {Restaurant} attribute (which is not a superkey).
- The dependencies are:

{Restaurant} →→ {Pizza Variety}

{Restaurant} →→ {Delivery Area}

To satisfy 4NF, we must place the facts about varieties offered into a different table from the facts about delivery areas:

Varieties By Restaurant

Restaurant	Pizza Variety
Vincenzo's Pizza	Thick Crust
Vincenzo's Pizza	Thin Crust
Elite Pizza	Thin Crust
Elite Pizza	Stuffed Crust
A1 Pizza	Thick Crust
A1 Pizza	Stuffed Crust

Delivery Areas By Restaurant

Restaurant	Delivery Area
Vincenzo's Pizza	Springfield
Vincenzo's Pizza	Shelbyville
Elite Pizza	Capital City
A1 Pizza	Springfield
A1 Pizza	Shelbyville
A1 Pizza	Capital City

Transaction Concept

Example

Consider the following fund transfer transaction

- Begin transaction
- Transfer 100 from Account X to Account Y
- Commit

Operations in this transaction

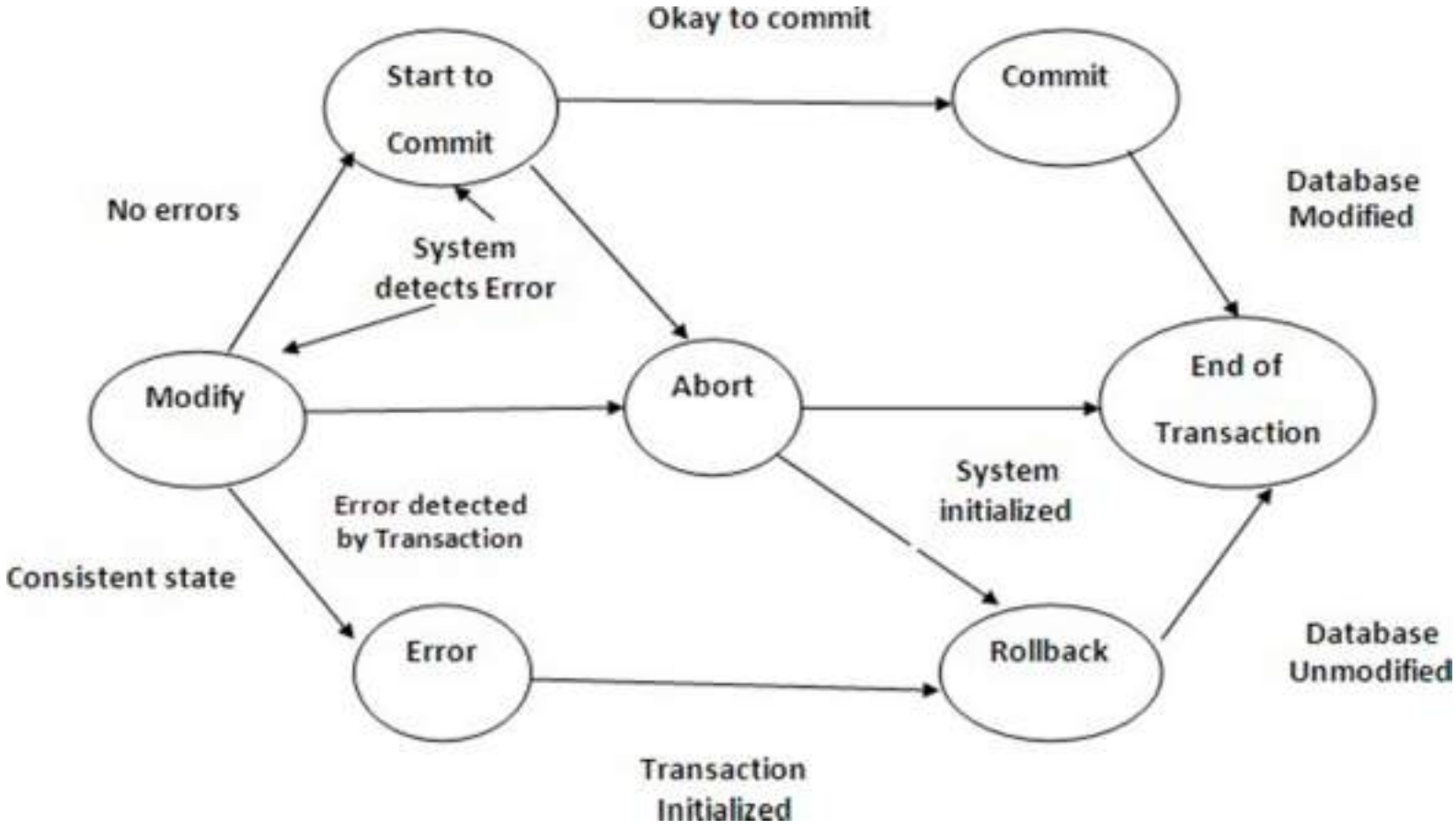
1. Read Account X from disk
2. If balance is less than 100, return with an appropriate message.
3. Subtract 100 from balance of Account X.
4. Write Account X back to disk.
5. Read Account Y from disk.
6. Add 100 to balance of Account Y.
7. Write Account Y back to disk.

Transaction Processing

- Transaction processing systems are the systems with large databases and hundreds of concurrent users that are executing database transaction.
- Example: systems for reservation, banking, credit card processing etc.

Transaction State - Implementation of Atomicity and Durability

States of a Transaction



Properties of a Transaction

- Atomicity
- Consistency
- Isolation
- Durability

This is also called as ACID property.

Atomicity

- Atomicity requires that database modifications must follow an "all or nothing" rule.
- Each transaction is said to be atomic. If one part of the transaction fails, the entire transaction fails and the database state is left unchanged.
- A transaction implies that it will run to completion as an indivisible unit, at the end of which either no changes have occurred to the database or the database has been changed in a consistent manner.

Consistency

- The consistency property of a transaction implies that if the database was in a consistent state before the start of a transaction, then on termination of a transaction the database will also be in a consistent state.

Isolation

- The isolation property of a transaction indicates that actions performed by a transaction will be isolated or hidden from outside the transaction until the transaction terminates.
- This property gives the transaction a measure of relative independence.

Durability

- The durability property guarantees that, once a transaction completes successfully, all the updates that it carried out on the database persist, even if there is a system failure after the transaction completes execution.

Schedule 1 (serial schedule)

- Let T_1 transfer \$50 from A to B, and T_2 transfer 10% of the balance from A to B.
- A serial schedule in which T_1 is followed by T_2:
  A=1000, B=2000

T_1	T_2
read(A)	
$A := A - 50$	
write (A)	
read(B)	
$B := B + 50$	
write(B)	
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
	read(B)
	$B := B + temp$
	write(B)

Result after schedule

- A=855
- B=2145

Concurrent - Executions - Serializability - Recoverability

Concurrency control mechanism

- **Optimistic**
- **Pessimistic**
- **Semi-optimistic**

Major goals of Concurrency control

Concurrency control mechanisms are usually designed to achieve some of, or all the following goals:

I. **Serializability**

II. **Recoverability**

Serializability

- A schedule is serializable if it is equivalent to a serial schedule.
- **Types of serializability**
  1. conflict serializability
  2. view serializability

Conflict operation

- Two operations conflict if:
  - They are issued by different transactions,
  - They operate on the same data item, and
  - At least on of them is a write operation

I_i = **read**(Q), I_j = **read**(Q). I_i and I_j don't conflict

I_i = **read**(Q), I_j = **write**(Q). They conflict

I_i = **write**(Q), I_j = **read**(Q). They conflict

I_i = **write**(Q), I_j = **write**(Q). They conflict

Conflict equivalent

- If a schedule S can be transformed into a schedule S' by a series of swaps of non-conflicting instructions, we say that S and S' are **conflict equivalent.**

Conflict Serializability

- An execution is conflict-serializable if it is conflict equivalent to a serial schedule.

Conflict Serializability (Cont.)

- Schedule 3 can be transformed into Schedule 6, a serial schedule where T_2 follows T_1, by series of swaps of non-conflicting instructions.
  - Therefore Schedule 3 is conflict serializable.

T_1	T_2
read(A)	
write(A)	
	read(A)
	write(A)
read(B)	
write(B)	
	read(B)
	write(B)

Schedule 3

T_1	T_2
read(A)	
write(A)	
read(B)	
write(B)	
	read(A)
	write(A)
	read(B)
	write(B)

Schedule 6

We continue to swap nonconflicting instructions:

- Swap *write(A) instruction of T2 with the read(B) instruction of T1.*
- *Swap the read(B) instruction of T1 with the read(A) instruction of T2.*
- *Swap the write(B) instruction of T1 with the write(A) instruction of T2.*
- *Swap the write(B) instruction of T1 with the read(A) instruction of T2.*

Lock-Based Protocols

Lock-Based Protocols

- A lock is a mechanism to control concurrent access to a data item.
- Data items can be locked in two modes :
  1. ***Exclusive (X) mode***: Data item can be both read as well as written. X-lock is requested using **lock-X** instruction.
  2. ***Shared (S) mode***: Data item can only be read. S-lock is requested using **lock-S** instruction.
- Lock requests are made to concurrency-control manager.
- Transaction can proceed only after request is granted.

The Two-Phase Locking Protocol

This is a protocol which ensures serializable schedules. This protocol requires that each transaction issue lock and unlock requests in two phases:

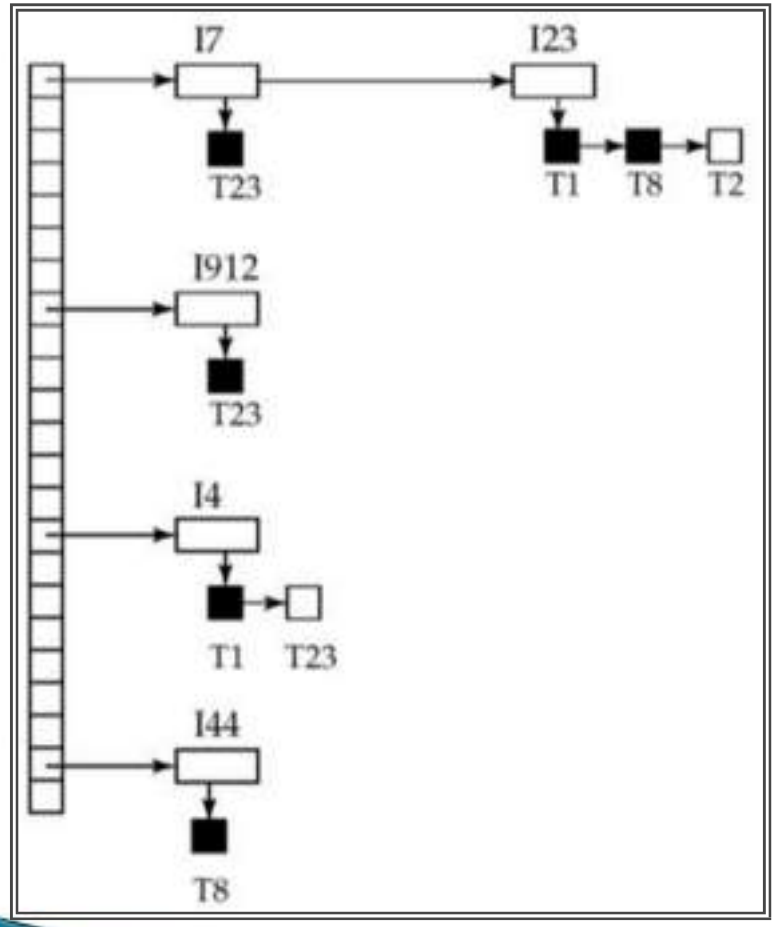
- Phase 1: Growing Phase
  - A transaction may obtain locks, but may not release any lock
- Phase 2: Shrinking Phase
  - A transaction may release locks, but may not obtain any new locks

Lock Conversions

- A mechanism for **upgrading** a shared lock to an exclusive lock, and **downgrading** an exclusive lock to a shared lock.
- We denote conversion from shared to exclusive modes by **upgrade, and from** exclusive to shared by **downgrade.**

- Two-phase locking with lock conversions:
  - First Phase:
    - can acquire a lock-S on item
    - can acquire a lock-X on item
    - can convert a lock-S to a lock-X (upgrade)
  - Second Phase:
    - can release a lock-S
    - can release a lock-X
    - can convert a lock-X to a lock-S (downgrade)

Lock Table



- Black rectangles indicate granted locks, white ones indicate waiting requests.
- Lock table also records the type of lock granted or requested.
- New request is added to the end of the linked list of requests for the data item, and granted if it is compatible with all earlier locks.
- Unlock requests result in the request being deleted, and later requests are checked to see if they can now be granted.
- If transaction aborts, all waiting or granted requests of the transaction are deleted
  - lock manager may keep a list of locks held by each transaction, to implement this efficiently.

Timestamp-Based Protocols

Timestamp-Based Protocols

- Each transaction is issued a timestamp when it enters the system.
- If an old transaction T_i has time-stamp TS(T_i), a new transaction T_j is assigned time-stamp TS(T_j) such that TS(T_i) <TS(T_j).
- The protocol manages concurrent execution such that the time-stamps determine the serializability order.

Timestamp-Based Protocols (Cont.)

There are two simple methods for implementing this scheme:

- **Use the value of the system clock as the timestamp; that is, a transaction's timestamp** is equal to the value of the clock when the transaction enters the system.
- **Use a logical counter that is incremented after a new timestamp has been** assigned; that is, a transaction's timestamp is equal to the value of the counter when the transaction enters the system.

Disadvantages

- Each value stored in the database requires two additional time stamp fields.
- One for the last time the field was read and one for the last update.
- Time stamping thus increases the memory needs and the database processing overhead.

Validation-Based Protocol / Optimistic concurrency control

Validation-Based Protocol

- Execution of transaction T_i is done in three phases.

1. **Read phase:** During this phase, the system executes transaction T_i . It reads the values of the various data items and stores them in variables local to T_i . It performs all write operations on temporary local variables, without updates of the actual database.
2. **Validation phase:** Transaction T_i performs a validation test to determine whether it can copy to the database the temporary local variables that hold the results of write operations without causing a violation of serializability.
3. **Write phase:** If transaction T_i succeeds in validation to the database. Otherwise, the system rolls back T_i .

(step 2), then the system applies the actual updates

to the database. Otherwise, the system rolls back T_i.

Validation-Based Protocol (Cont.)

- To perform the validation test, we need to know when the various phases of transactions T_i took place.

1. Start(T_i): the time when T_i started its execution.
2. Validation(T_i): the time when T_i finished its read phase and started its validation phase.
3. Finish(T_i): the time when T_i finished its write phase.

Schedule Produced by Validation

- Example of schedule produced using validation

T_{14}	T_{15}
read(B)	
	read(B) $B := B-50$ **read**(A) $A := A+50$
read(A) (*validate*) **display** ($A+B$)	
	(*validate*) **write** (B) **write** (A)

Recovery and Atomicity, Log - Based Recovery

CRASH RECOVERY

- Database recovery is the process of restoring a database to the consistent state that existed before the failure.

Failure Classification

- **Transaction failure**
- **System crash**
- **Disk failure**

Transaction failure

- **Logical errors**: transaction cannot complete due to some internal error condition, such as bad input, data not found, overflow, or resource limit exceeded.
- **System errors**: the database system must terminate an active transaction due to an error condition (e.g., deadlock)

System crash

- There is a hardware malfunction, or a bug in the database software or the operating system, that causes the loss of the content of volatile storage, and brings transaction processing to a halt.
- The content of nonvolatile storage remains intact, and is not corrupted.

Disk failure

- A head crash or similar disk failure destroys all or part of disk storage.

Storage Structure

Storage types

- **Volatile storage**:
  - does not survive system crashes
  - examples: main memory, cache memory
- **Nonvolatile storage**:
  - survives system crashes
  - examples: disk, tape, flash memory, non-volatile (battery backed up) RAM
- **Stable storage**:
  - a mythical form of storage that survives all failures
  - approximated by maintaining multiple copies on distinct nonvolatile media

Recovery techniques

- Log-based recovery
  - Deferred Database Modification
  - Immediate Database Modification
- Shadow Paging

Log-Based Recovery

- The most widely used structure for recording database modifications is the log.
- The log is a sequence of **log records**, and maintains a record of all the update activities in the database.
  i. When transaction T_i starts, it registers itself by writing $<T_i$ **start**$>$
  ii. $<T_i, X, V_1, V_2>$ is written, where V_1 is the value of X before the write, and V_2 is the value to be written to X.
  iii. When T_i finishes it's last statement, the log record $<T_i$ **commit**$>$ is written.
  iv. $<T_i$ **abort**$>$, transaction T_i *has aborted.*

Deferred Database Modification

- The **deferred database modification** scheme records all modifications to the log, but defers all the **write**s to after partial commit.
- Assume that transactions execute serially
- Transaction starts by writing $<T_i$ ***start***$>$ record to log.
- A **write**(X) operation results in a log record $<T_i, X, V>$ being written, where V is the new value for X
  - Note: old value is not needed for this scheme
- The write is not performed on X at this time, but is deferred.
- When T_i partially commits, $<T_i$ **commit**$>$ is written to the log.
- Finally, the log records are read and used to actually execute the previously deferred writes.

State of the log and database corresponding to T_0 and T_1

Log	Database
$<T_0$ start$>$	
$<T_0, A, 950>$	
$<T_0, B, 2050>$	
$<T_0$ commit$>$	
	$A = 950$
	$B = 2050$
$<T_1$ start$>$	
$<T_1, C, 600>$	
$<T_1$ commit$>$	
	$C = 600$

Immediate Database Modification

- The immediate-modification technique allows database modifications to be output to the database while the transaction is still in the active state.
- Before a transaction T_i *starts its execution, the system writes the record* $<T_i$ *start*$>$ to the log.
- During its execution, any write(X) *operation by* T_i *is preceded by the writing* of $< T_i, X, V_1, V_2>$ to the log.
- When T_i *partially commits, the* system writes the record $< T_i$ *commit*$>$ *to the log.*

Recovery with Concurrent Transactions

Checkpoints

- Problems in recovery procedure as discussed are:
  1. Searching the entire log is time-consuming
  2. We might unnecessarily redo transactions which have already written their updates into the database.

Checkpoints (Cont.)

To reduce these types of overhead, we introduce checkpoints.

The system periodically performs **checkpoints, which require** the following sequence of actions to take place:

1. Output all log records currently residing in main memory onto stable storage.
2. Output all modified buffer blocks to the disk.
3. Write a log record <**checkpoint**> onto stable storage.

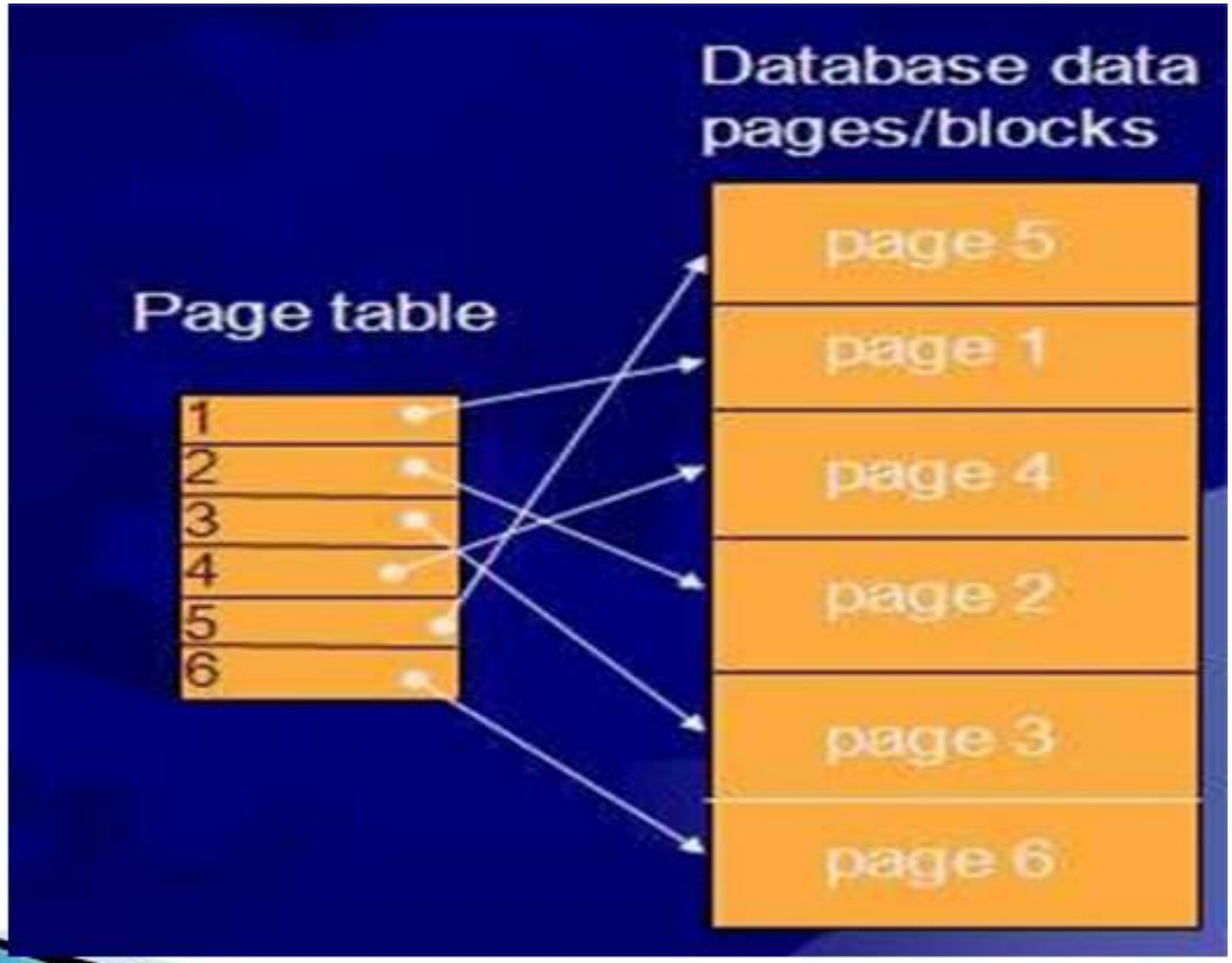
Recovery scheme

- During recovery we need to consider only the most recent transaction T_i that started before the checkpoint, and transactions that started after T_i.
  1. Scan backwards from end of log to find the most recent <**checkpoint**> record.
  2. Continue scanning backwards till a record <T_i **start**> is found.
  3. Once the system has identified transaction T_i *the redo and undo operations need* to be applied to only transaction T_i *and all transactions* T_j *that started executing* after transaction T_i.
  4. For all transactions (starting from T_i or later) with no <T_i **commit**>, execute **undo(***T_i***).**
  5. Scanning forward in the log, for all transactions starting from T_i or later with a <T_i **commit**>, execute **redo(***T_i***).**

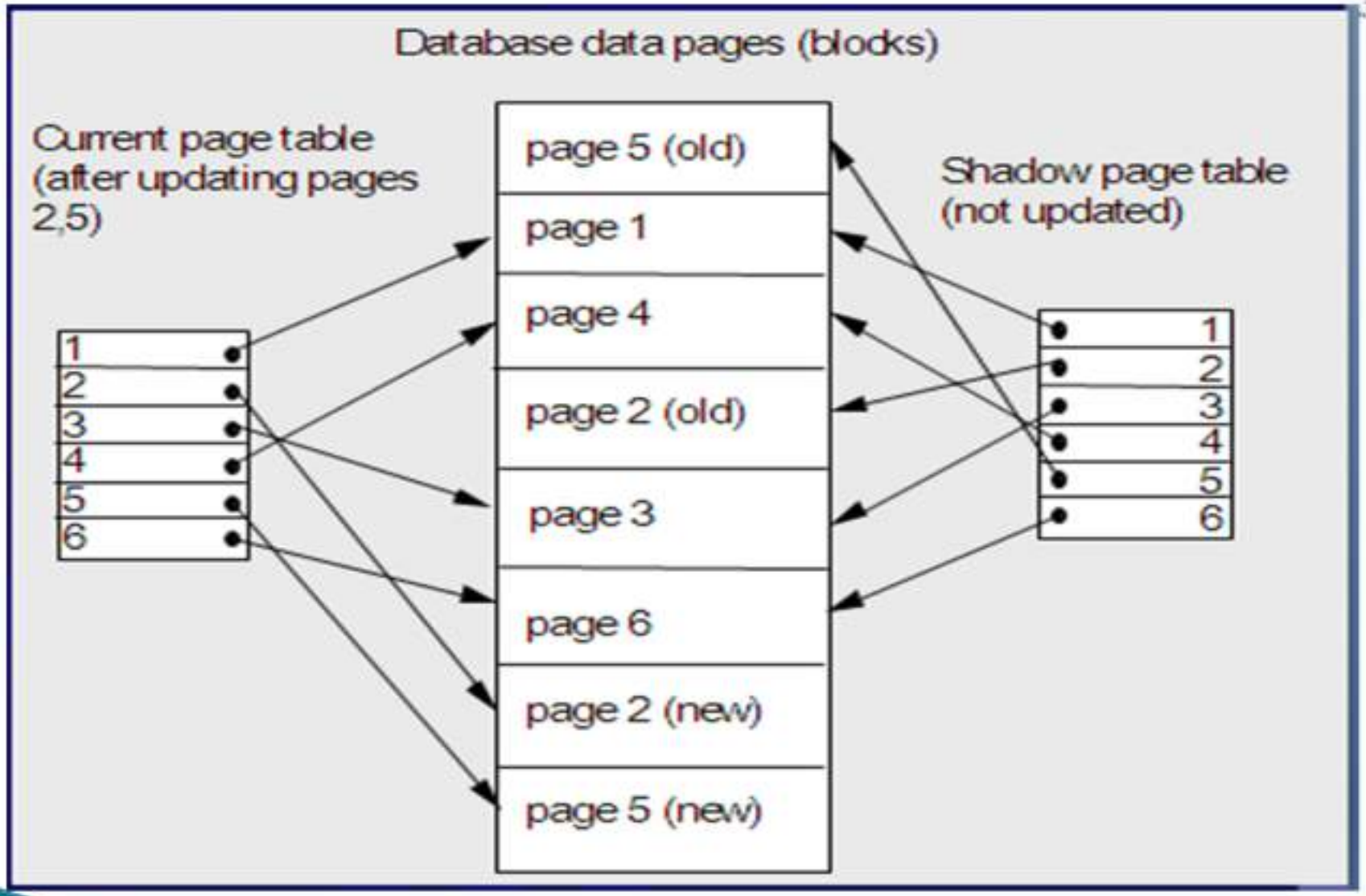
Shadow Paging

- An alternative to log-based crash-recovery techniques is **shadow paging.**
- The database is partitioned into some number of fixed-length blocks, which are referred to as **pages.**
- Assume that there are n pages, numbered 1 through n.
- **Page table, contains all the pages.**

Sample Page Table



Example of Shadow Paging



Committing a transaction

1. Ensure that all buffer pages in main memory are output to disk.

2. Output the current page table to disk.

3. Discard previous shadow page.

4. Pages not pointed to from current/shadow page table should be freed (garbage collected).

To recover from a failure

- the state of the database before transaction execution is available through the shadow page table
- discard current page table
- that state is recovered by reinstating the shadow page table to become the current page table once more

Advantages of shadow-paging over log-based schemes

1. The overhead of log-record output is eliminated.
2. Recovery from crashes is significantly faster (since no undo or redo operations are needed).

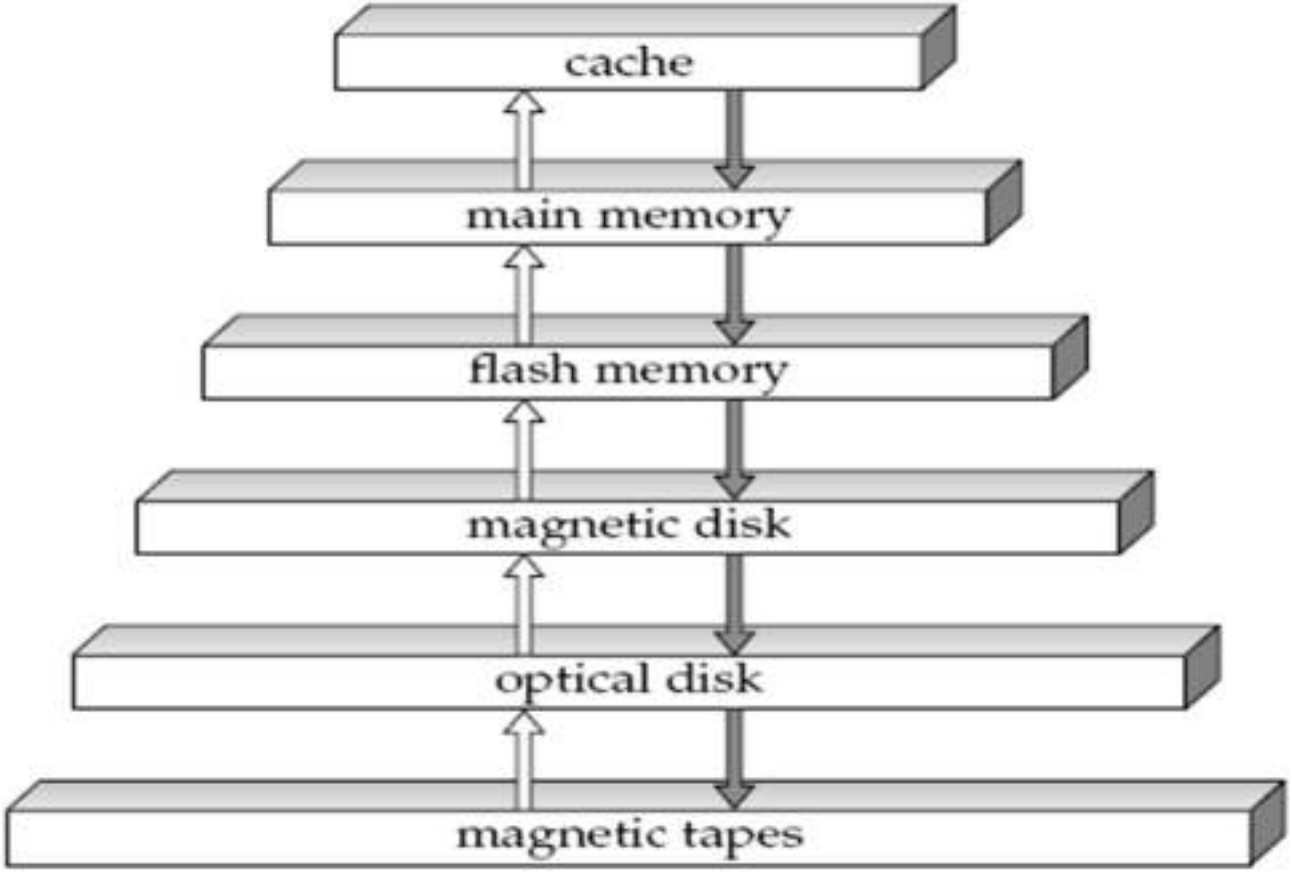
Disadvantages

- Copying the entire page table is very expensive
  - Can be reduced by using a page table structured like a B^+–tree
    - No need to copy entire tree, only need to copy paths in the tree that lead to updated leaf nodes
- Commit overhead is high even with above extension
  - Need to flush every updated page, and page table
- Data gets fragmented (related pages get separated on disk)
- After every transaction completion, the database pages containing old versions of modified data need to be garbage collected
- Hard to extend algorithm to allow transactions to run concurrently
  - Easier to extend log based schemes


THANK
YOU

Data on External Storage

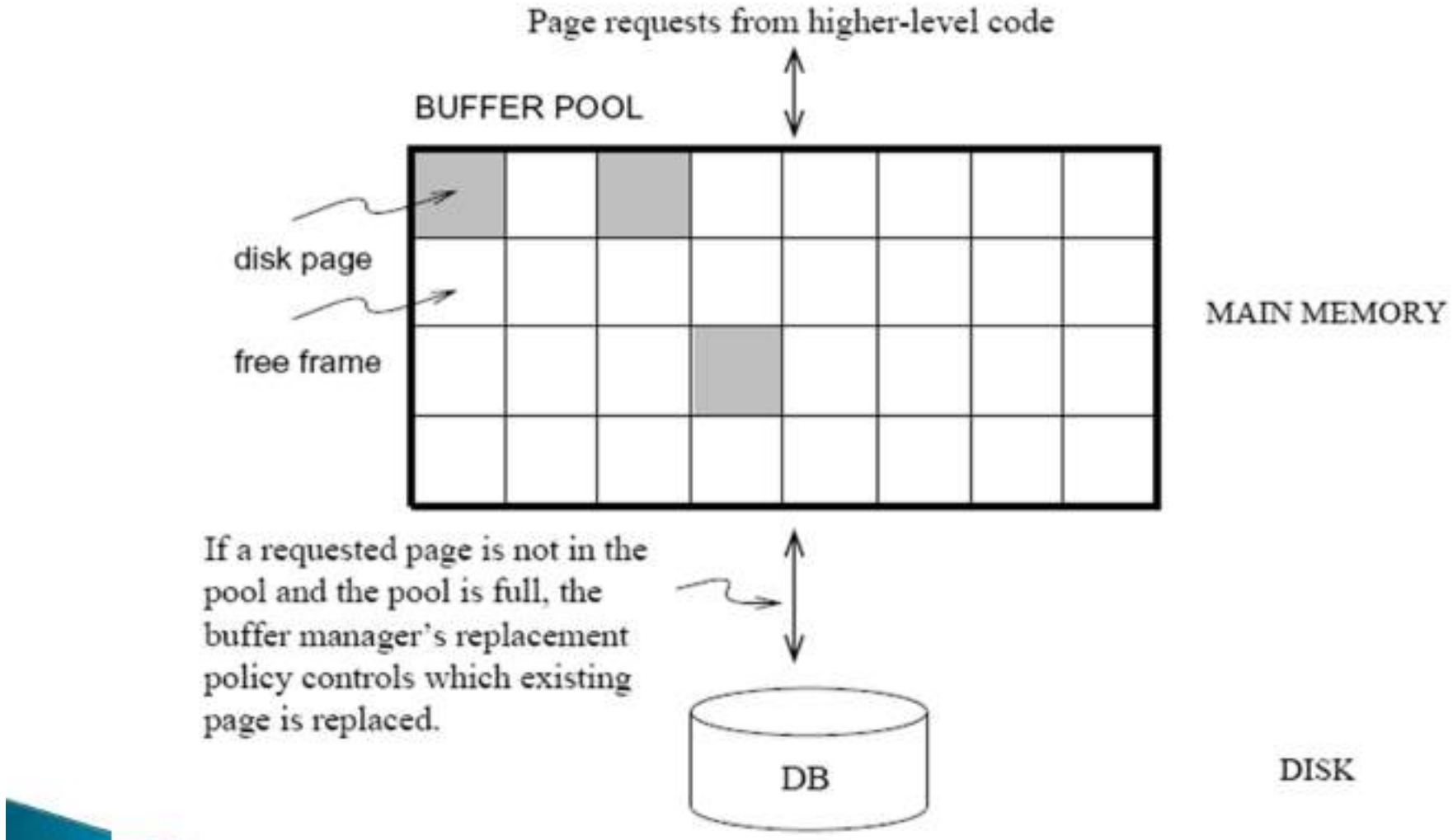
Storage-device hierarchy



Buffer Manager

- Files reside permanently on disks.
- Each file is partitioned into fixed-length storage units called **blocks.**
- The **buffer** is the part of main memory available for storage of copies of disk blocks.
- The subsystem responsible for the allocation of buffer space is called the **buffer manager.**
- The **buffer manager is** the software layer that is responsible for bringing pages from disk to main memory as needed.
- The buffer manager manages the available main memory by partitioning it into a collection of pages, which we collectively refer to as the **buffer pool.**
- **The main** memory pages in the buffer pool are called **frames; it is convenient to think of them** as slots that can hold a page.

The buffer pool



Buffer Manager techniques

- **Buffer replacement strategy:** When there is no room left in the buffer, a block must be removed from the buffer. Most operating systems use a least recently used (LRU) scheme.

- **Pinned blocks:** Most recovery systems require that a block should not be written to disk while an update on the block is in progress. A block that is not allowed to be written back to disk is said to be pinned.

- **Forced output of blocks**: There are situations in which it is necessary to write back the block to disk, even though the buffer space that it occupies is not needed. This write is called the forced output of a block.

Record Structure

- The database is stored as a collection of files.
- Each file is a sequence of records.
- A record is a sequence of fields.

Types of records

- **Fixed-Length Records**: every record in the file has exactly the same size (in bytes).
- **Variable-Length Records:** different records in the file have different sizes.

Fixed-Length Records

Let us consider a file of account records for bank database.

Each record of this file is defined as:

Account-number: char (10);

Branch-name: char (22);

Balance: real; //Real size=8

Record size= 10+22+8= 40 bytes

A simple approach is to use the first 40 bytes for the first record, the next 40 bytes for the second record, and so on.

There are two problems with this simple approach:

1. It is difficult to delete a record from this structure. The space occupied by the record to be deleted must be filled with some other record of the file, or we must have a way of marking deleted records so that they can be ignored.

2. Unless the block size happens to be a multiple of 40 (which is unlikely), some records will cross block boundaries. That is, part of the record will be stored in one block and part in another. It would thus require two block accesses to read or write such a record.

Deletion of record 1st approach

- When a record is deleted, we could move the record that came after it into the space occupied by the deleted record, and so on, until every record following the deleted record has been moved ahead. Such an approach requires moving a large number of records.

record 0	A-102	Perryridge	400
record 1	A-305	Round Hill	350
record 3	A-101	Downtown	500
record 4	A-222	Redwood	700
record 5	A-201	Perryridge	900
record 6	A-217	Brighton	750
record 7	A-110	Downtown	600
record 8	A-218	Perryridge	700

Deletion of record 2nd approach

- It might be easier simply to move the final record of the file into the space occupied by the deleted record. It is undesirable to move records to occupy the space freed by a deleted record, since doing so requires additional block accesses.

record 0	A-102	Perryridge	400
record 1	A-305	Round Hill	350
record 8	A-218	Perryridge	700
record 3	A-101	Downtown	500
record 4	A-222	Redwood	700
record 5	A-201	Perryridge	900
record 6	A-217	Brighton	750
record 7	A-110	Downtown	600

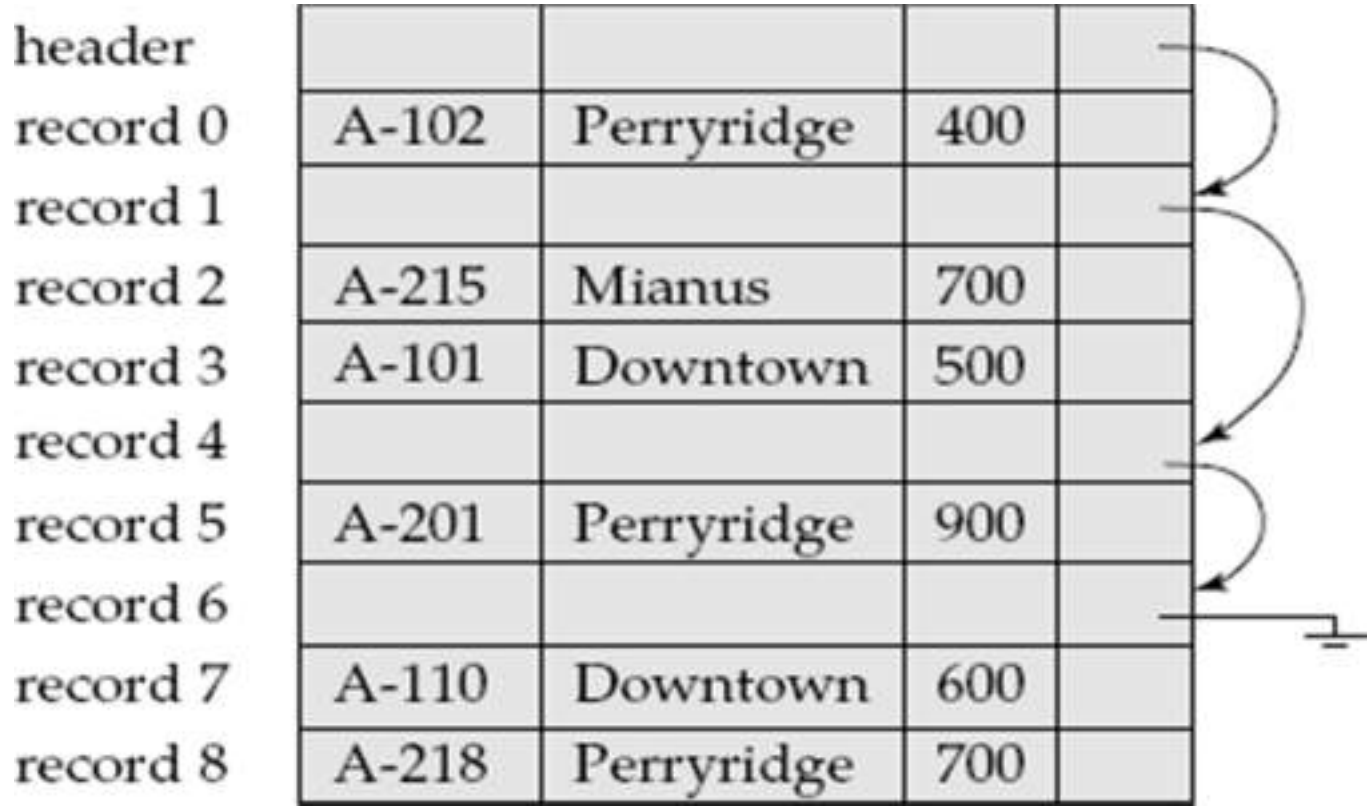
Deletion of record 3rd approach

- Since insertions tend to be more frequent than deletions, it is acceptable to leave open the space occupied by the deleted record, and to wait for a subsequent insertion before reusing the space.
- A simple marker on a deleted record is not sufficient, since it is hard to find this available space when an insertion is being done.
- Thus, we need to introduce an additional structure.

File header

- At the beginning of the file, we allocate a certain number of bytes as a **file header.**
- The header will contain a variety of information about the file.
- For now, all we need to store there is the address of the first record whose contents are deleted.
- We use this we can think of these stored addresses as pointers, since they point to the location of a record.
- The deleted records thus form a linked list, which is often referred to as a **free list.**
- On insertion of a new record, we use the record pointed to by the header.
- We change the header pointer to point to the next available record.
- If no space is available, we add the new record to the end of the file.

Continued.....



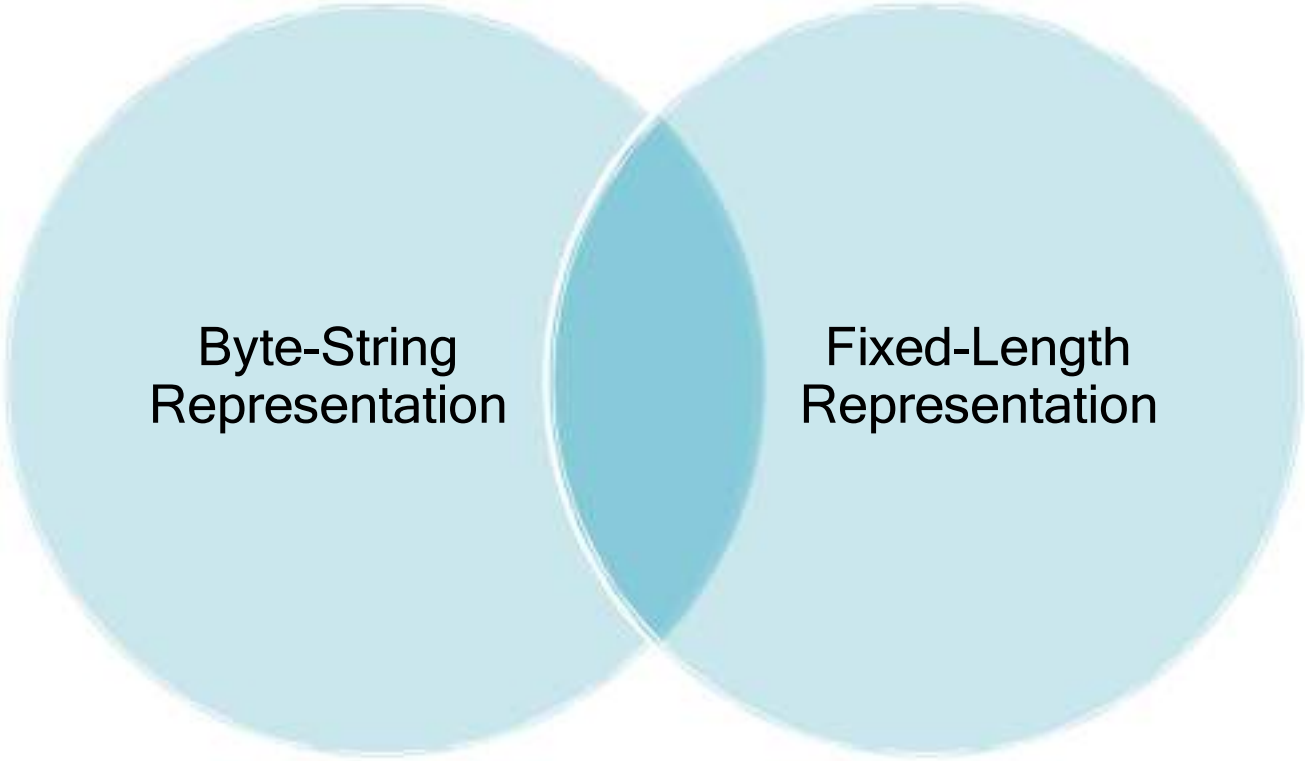
header				
record 0	A-102	Perryridge	400	
record 1				
record 2	A-215	Mianus	700	
record 3	A-101	Downtown	500	
record 4				
record 5	A-201	Perryridge	900	
record 6				
record 7	A-110	Downtown	600	
record 8	A-218	Perryridge	700	

Variable-Length Records

Variable-length records arise in database systems in several ways:

- Record types that allow variable lengths for one or more fields.
- Record types that allow repeating fields (used in some older data models).

Techniques for implementing variable-length records

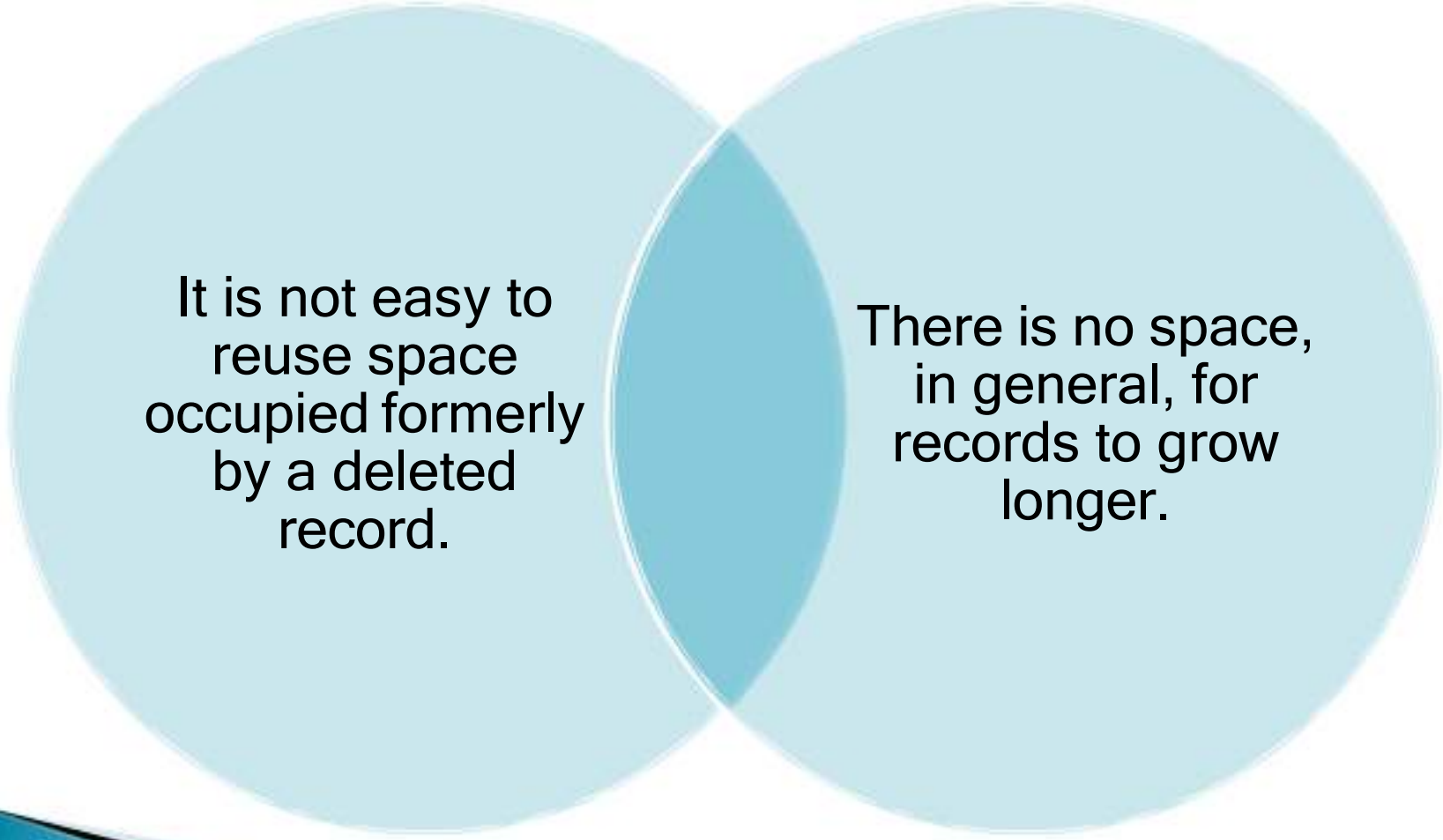


Byte-String Representation

- A simple method for implementing variable-length records is to attach a special end-of-record (⊥) symbol to the end of each record.

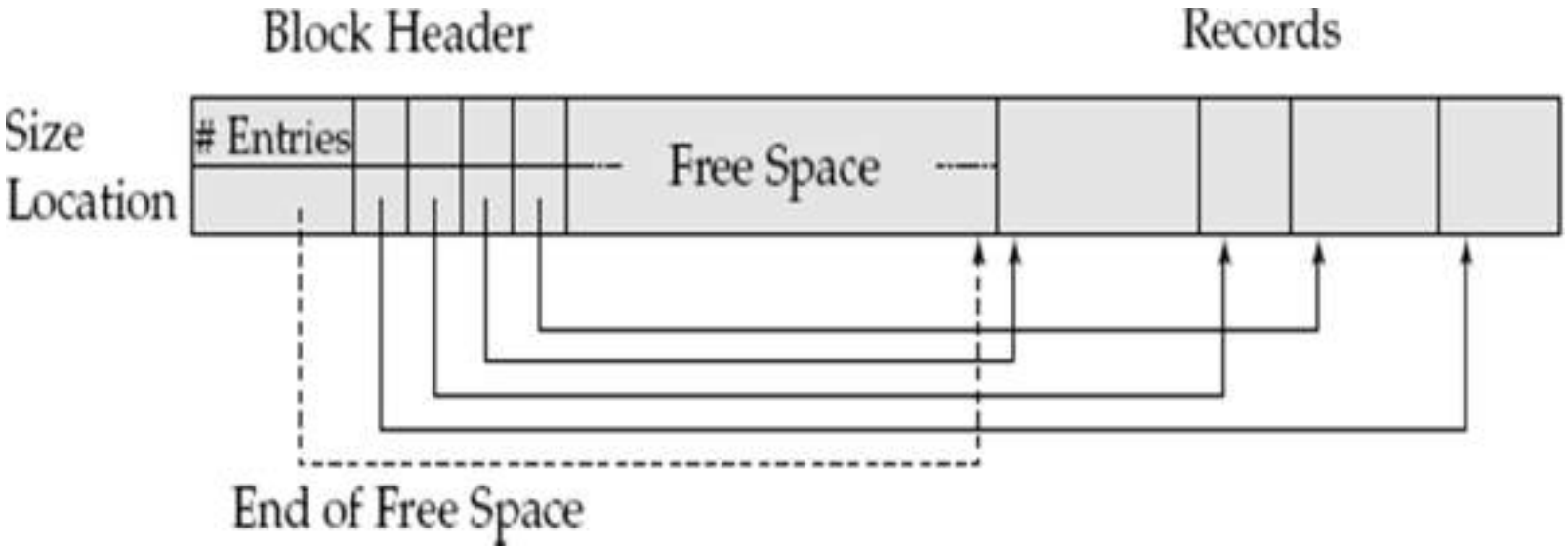
0	Perryridge	A-102	400	A-201	900	A-218	700	⊥
1	Round Hill	A-305	350	⊥				
2	Mianus	A-215	700	⊥				
3	Downtown	A-101	500	A-110	600	⊥		
4	Redwood	A-222	700	⊥				
5	Brighton	A-217	750	⊥				

Byte-string representation disadvantages:



Slotted-page structure

- A modified form of the byte-string representation, called the **slotted-page structure**, is commonly used for organizing records within a single block.



- There is a header at the beginning of each block, containing the following information:

1. The number of record entries in the header
2. The end of free space in the block
3. An array whose entries contain the location and size of each record

- The actual records are allocated contiguously in the block, starting from the end of the block.
- The free space in the block is contiguous, between the final entry in the header array, and the first record.
- If a record is inserted, space is allocated for it at the end of free space, and an entry containing its size and location is added to the header.
- If a record is deleted, the space that it occupies is freed, and its entry is set to deleted.

Fixed-Length Representation

- Another way to implement variable-length records efficiently in a file system is to use one or more fixed-length records to represent one variable-length record.

There are two ways of doing this:

1. **Reserved space**: If there is a maximum record length that is never exceeded, we can use fixed-length records of that length. Unused space is filled with a special null, or end-of-record, symbol.
2. **List representation**: We can represent variable-length records by lists of fixed length records, chained together by pointers.

Reserved space method

0	Perryridge	A-102	400	A-201	900	A-218	700
1	Round Hill	A-305	350	⊥	⊥	⊥	⊥
2	Mianus	A-215	700	⊥	⊥	⊥	⊥
3	Downtown	A-101	500	A-110	600	⊥	⊥
4	Redwood	A-222	700	⊥	⊥	⊥	⊥
5	Brighton	A-217	750	⊥	⊥	⊥	⊥

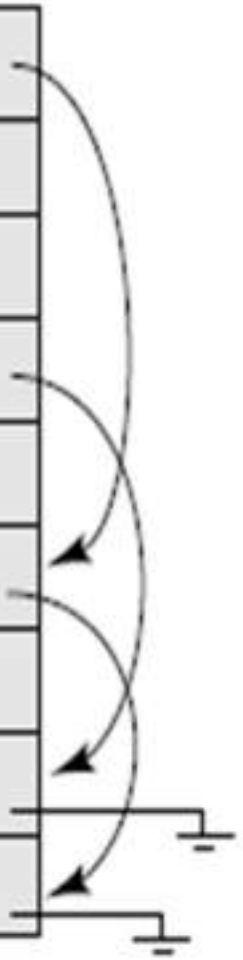
Reserved-space method

- The reserved-space method is useful when most records have a length close to the maximum.

- Otherwise, a significant amount of space may be wasted.

List representation method

0	Perryridge	A-102	400	
1	Round Hill	A-305	350	
2	Mianus	A-215	700	
3	Downtown	A-101	500	
4	Redwood	A-222	700	
5		A-201	900	
6	Brighton	A-217	750	
7		A-110	600	
8		A-218	700	



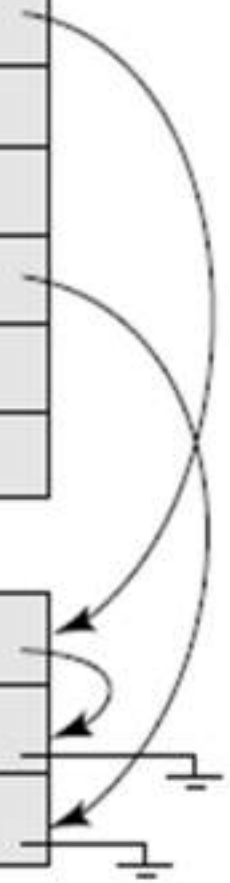
Anchor-block and overflow-block structures

anchor block

Perryridge	A-102	400	
Round Hill	A-305	350	
Mianus	A-215	700	
Downtown	A-101	500	
Redwood	A-222	700	
Brighton	A-217	750	

overflow block

A-201	900	
A-218	700	
A-110	600	



File Organization and Indexing - Clustered Indexes

File organization

- File organization includes the way records and blocks are placed on the storage medium.
- A fi**le organization is a way of arranging the records in a file when the file is stored** on disk.
- There are two types of file organization
  - Primary File Organizations
  - Secondary File Organizations

Primary File Organizations

- Unordered or Heap or Pile Files
- Ordered or Sorted or sequential Files
- Hash or Direct Files

Unordered or Heap or Pile Files

- Records are placed in the file in the order in which they are inserted.
- Inserting a new record is very efficient.
- Searching can be done by linear search (inefficient).
- Deletion is very inefficient.

Ordered or Sorted or sequential Files

- It store records in sequential order, based on the value of the **search key** of each record.
- An attribute or set of attribute used to look up records in a file is called a **search key**.

Advantages of Ordered Files

- Reading of the records in order of the **ordering field** is extremely efficient, because no sorting is required.

- Finding the next record is fast.

Disadvantages of Ordered Files

- Searches on non-ordering fields are inefficient.
- Insertion and deletion of records are very expensive.

Hash or Direct Files

- Hash function computed on some attribute of each record; the result specifies where record should be placed.
- The pages in a hashed file are grouped into buckets.
- Given a bucket number, the hashed file structure allows us to find the primary page for that bucket.
- The bucket to which a record belongs can be determined by applying a special function called a hash function, to the search field(s).
- Insertions and deletions are fast.

Secondary File Organizations

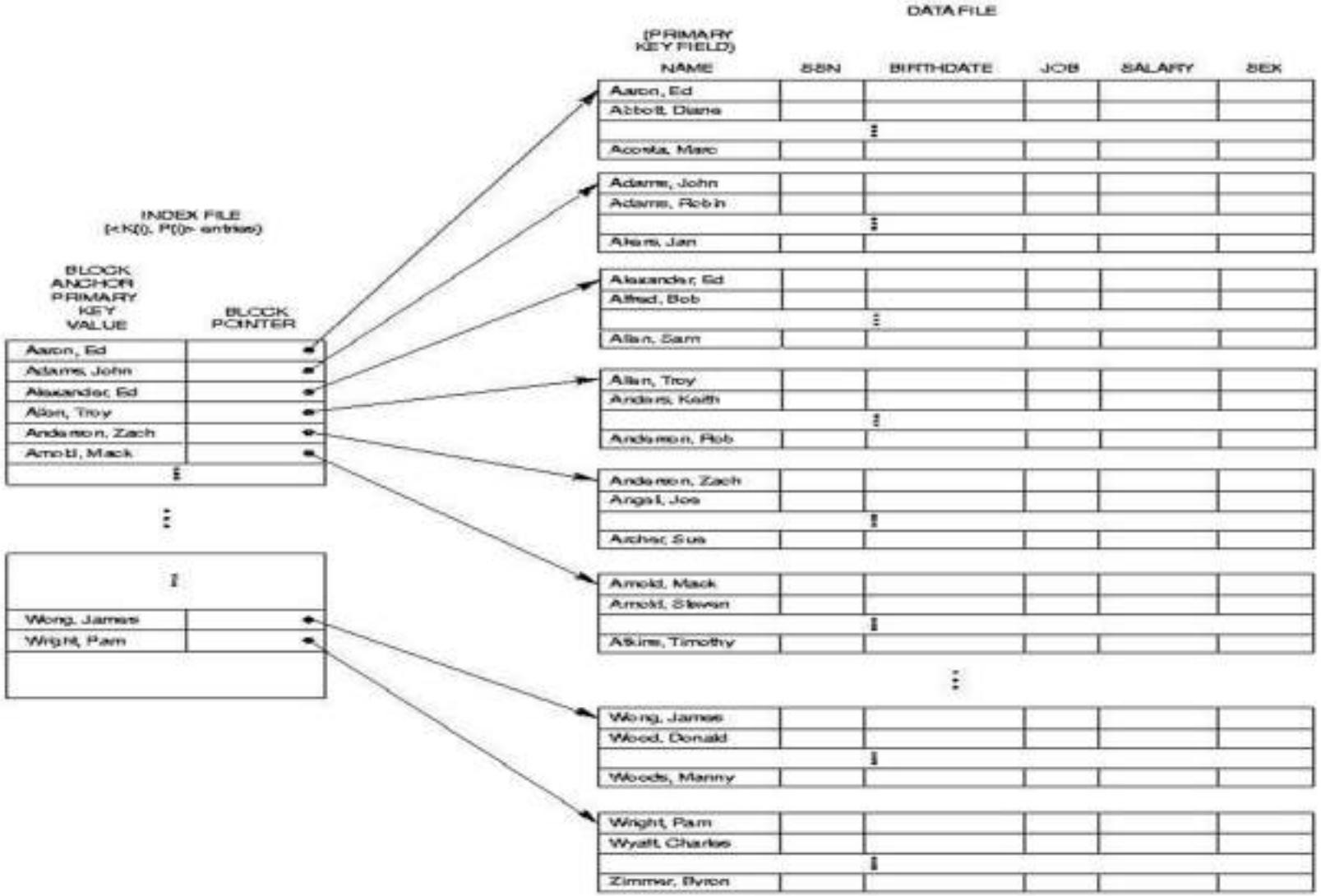
- Secondary file organization uses the index to access the records.
- An index is a data structure that speeds up certain operations on a file.
- An index for a file in a database system works in the same way as the index in any textbook.
- If we want to learn about a particular topic (specified by a word or a phrase) , we can search for the topic in the index at the back of the book.
- Indexes provide faster access to data.

Types of Indexes

- **Single-level ordered indexes**
  - Primary indexes
  - Secondary indexes
  - Clustering indexes
- **Multi-level Indexes**
- **Dynamic Multi-level indexes using B-trees and B+-trees**

Primary indexes

- An index is called a primary index if the search key includes the primary key.
- A **Primary Index** is constructed of two parts: The **first field** is the same data type of the **primary key** of a file block of the data file and the **second field** is file **block pointer**.


DATA FILE
(PRIMARY KEY FIELD)
NAME
SSN
BIRTHDATE
JOB
SALARY
SEX
INDEX FILE
BLOCK ANCHOR PRIMARY KEY VALUE
BLOCK POINTER
Aaron, Ed
Adams, John
Alexander, Ed
Allen, Troy
Anderson, Zach
Arnold, Mack
Wong, James
Wright, Pam

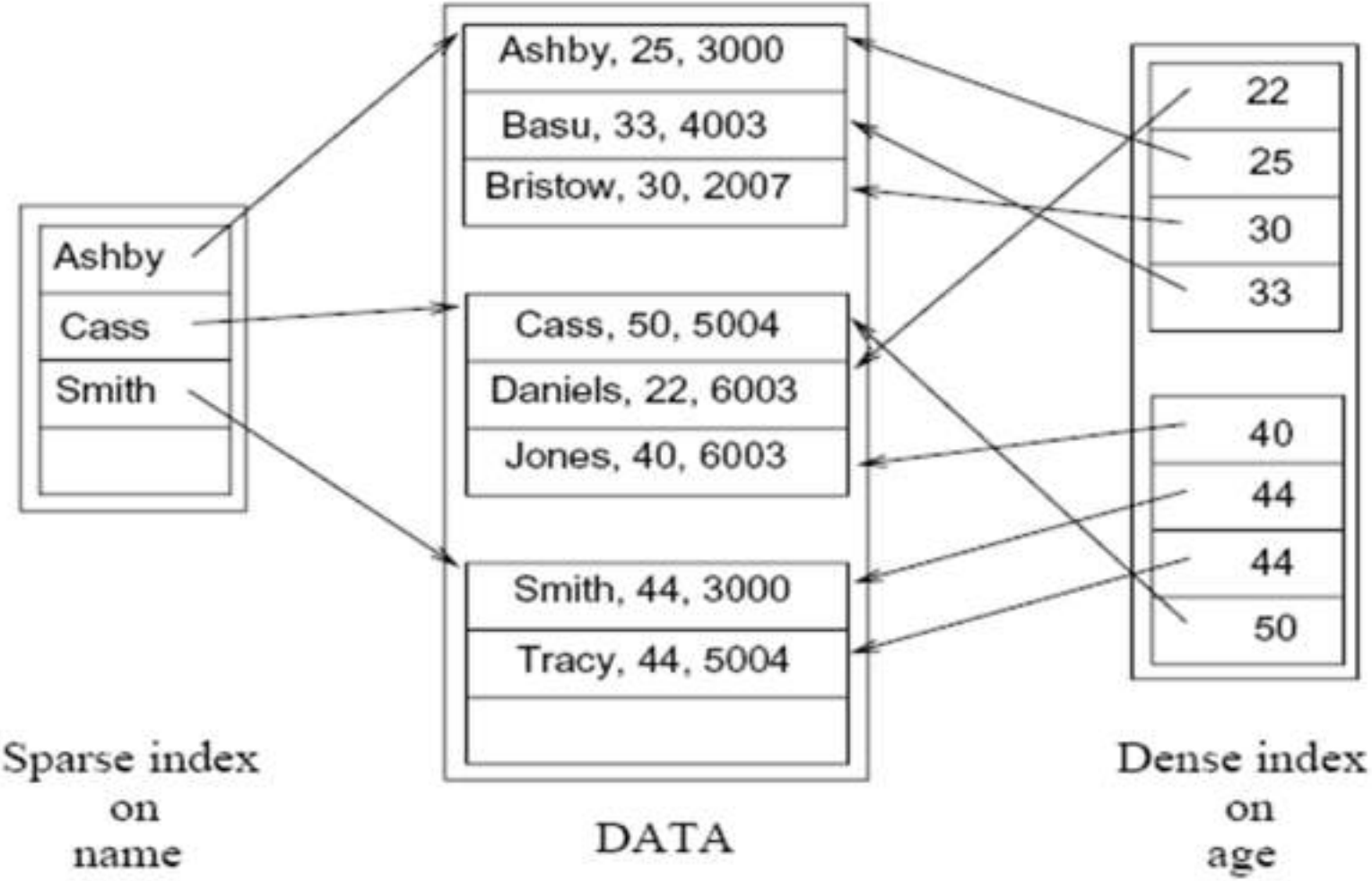
Problem with a primary index

A major problem with a primary index—as with any ordered file—is insertion and deletion of records.

Indexes can also be characterized as

- Dense: A **dense index has an index entry for every** search key value (and hence every record) in the data file.
- Sparse (nondense): A **sparse (or nondense) index, on the other** hand, has index entries for only some of the search values.
- A primary index is hence a nondense (sparse) index, since it includes an entry for each disk block of the data file rather than for every search value (or every record).

Sparse versus Dense Indexes



Clustering Indexes

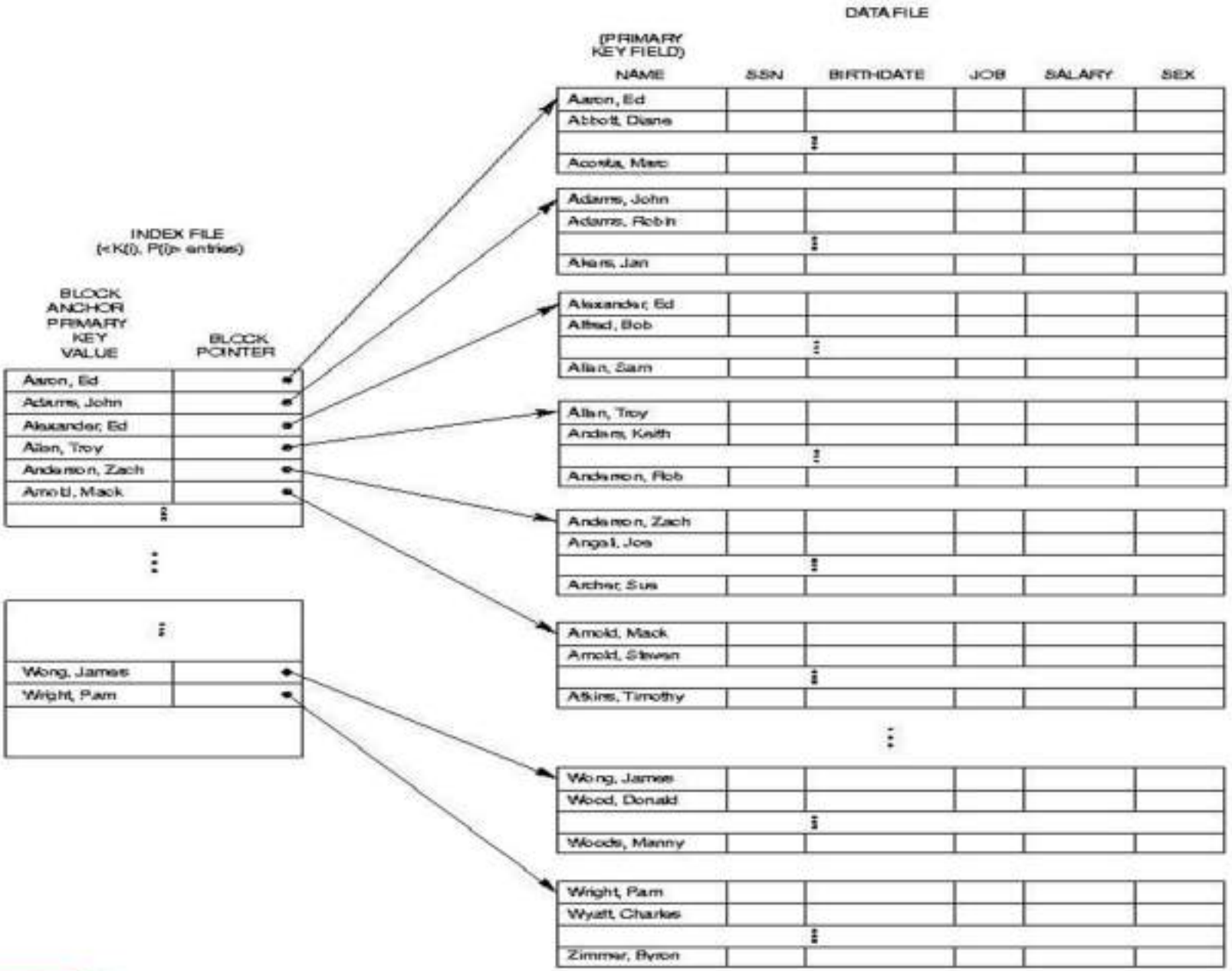
- If records of a file are physically ordered on a non key field—which does not have a distinct value for each record—that field is called the **clustering field.**

- A clustering index is also an ordered file with two fields; the first field is of the same type as the clustering field of the data file, and the second field is a block pointer.

Primary Index and Secondary Index

Primary indexes

- An index is called a primary index if the search key includes the primary key.
- A **Primary Index** is constructed of two parts: The **first field** is the same data type of the **primary key** of a file block of the data file and the **second field** is file **block pointer**.


DATA FILE
(PRIMARY KEY FIELD)
NAME
SSN
BIRTHDATE
JOB
SALARY
SEX
INDEX FILE
(<K(i), P(i)> entries)
BLOCK ANCHOR PRIMARY KEY VALUE
BLOCK POINTER
Aaron, Ed
Adams, John
Alexander, Ed
Allen, Troy
Anderson, Zach
Arnold, Mack
Wong, James
Wright, Pam

Problem with a primary index

A major problem with a primary index as with any ordered file is insertion and deletion of records.

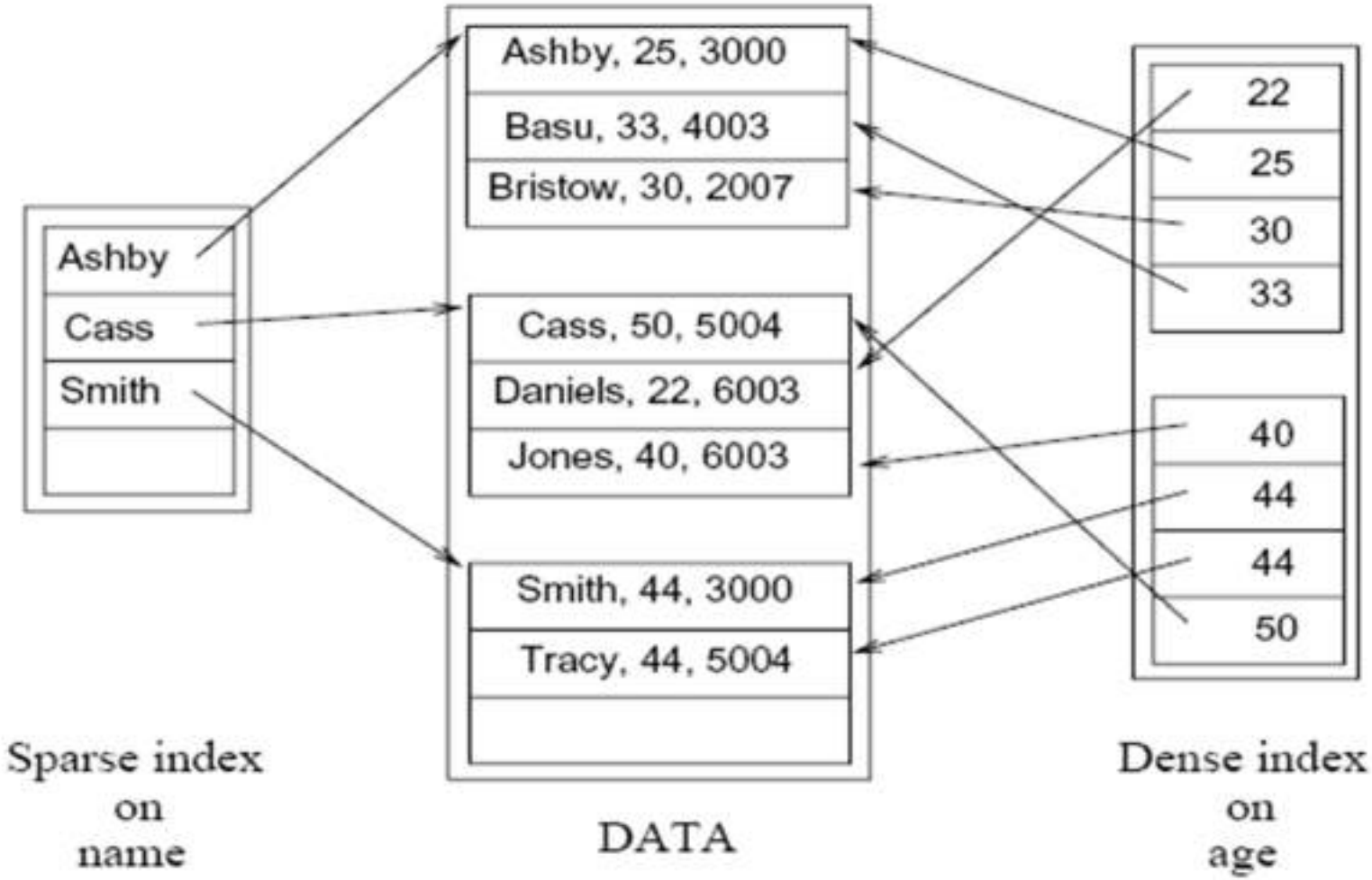
Indexes can also be characterized as

- Dense: A **dense index has an index entry for every** search key value (and hence every record) in the data file.

- Sparse (non-dense): A **sparse (or non-dense) index, on the other** hand, has index entries for only some of the search values.

- A primary index is hence a non-dense (sparse) index, since it includes an entry for each disk block of the data file rather than for every search value (or every record).

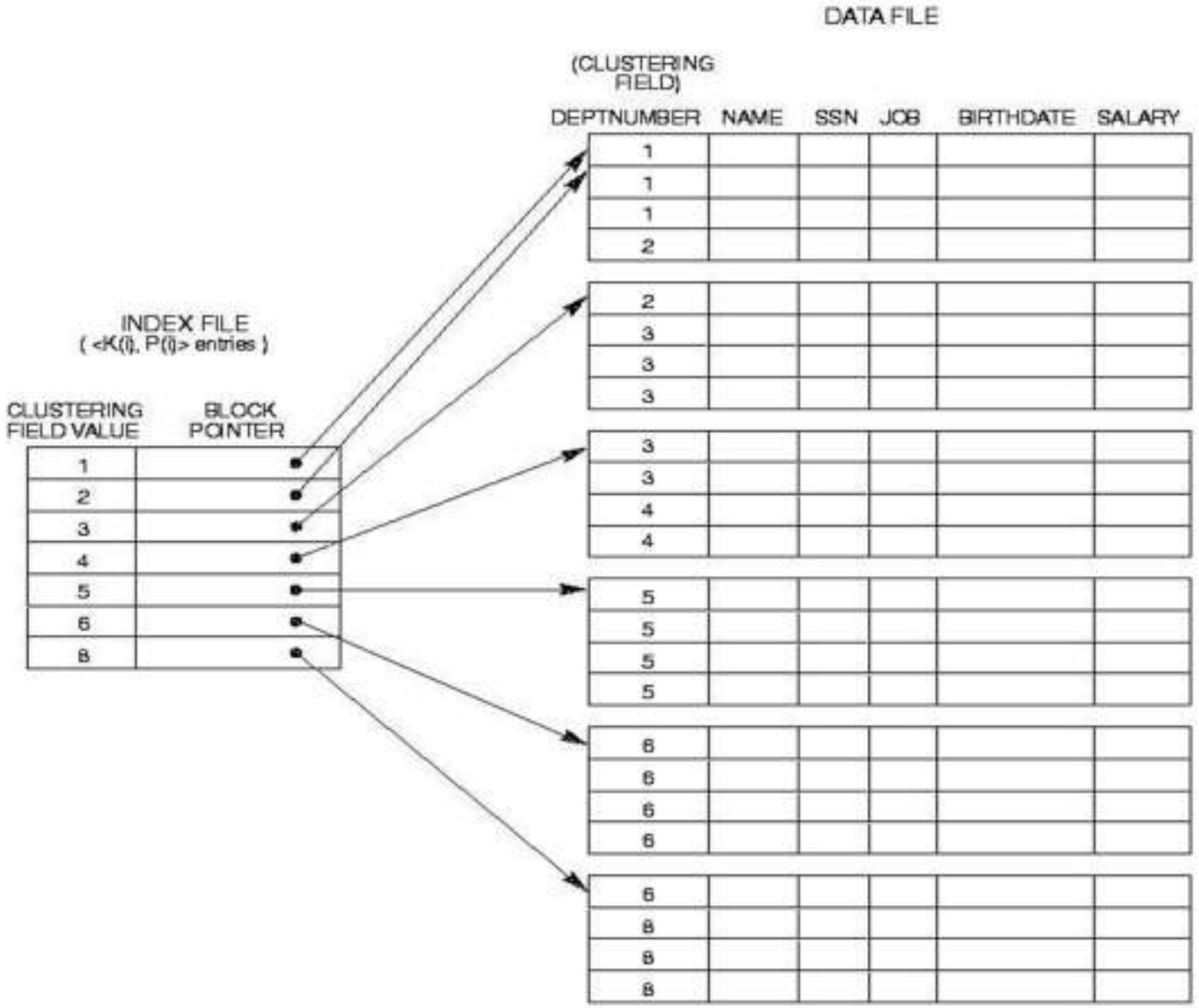
Sparse versus Dense Indexes



Clustering Indexes

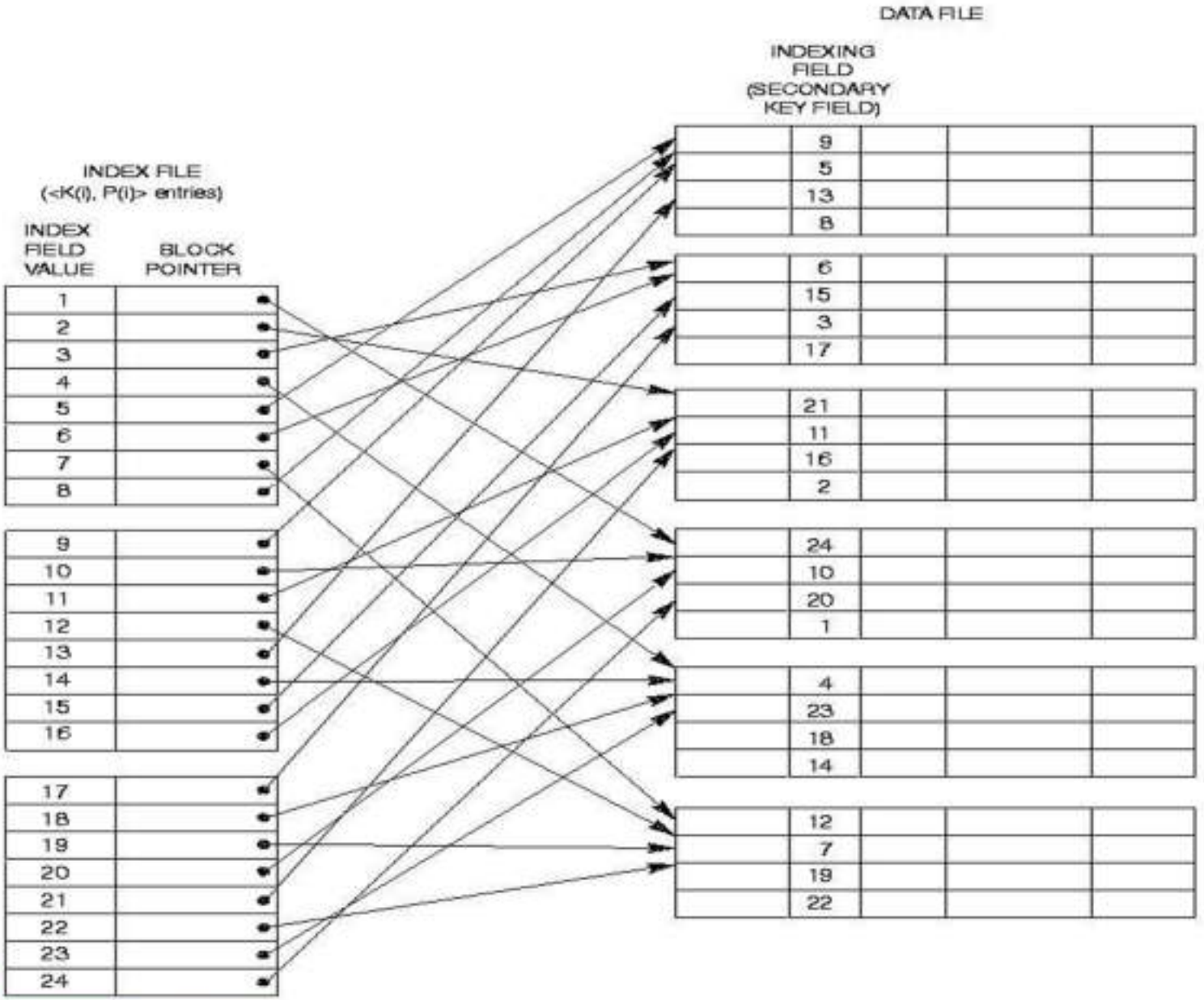
- If records of a file are physically ordered on a non-key field—which does not have a distinct value for each record—that field is called the **clustering field.**

- A clustering index is also an ordered file with two fields; the first field is of the same type as the clustering field of the data file, and the second field is a block pointer.


DATA FILE
(CLUSTERING FIELD)
DEPTNUMBER
NAME
SSN
JOB
BIRTHDATE
SALARY
INDEX FILE
(<K(i), P(i)> entries)
CLUSTERING FIELD VALUE
BLOCK POINTER
1
2
3
4
5
6
8

Secondary Indexes

- A **Secondary Index** is an **ordered file** with two fields.
- The **first** is of the same data type as some **non-ordering field** and the second is either a block or a record pointer.
- If the **entries** in this non-ordering field **must be unique** this field is sometime referred to as a **Secondary Key**. This results in a dense index.


DATA FILE
INDEXING FIELD (SECONDARY KEY FIELD)
INDEX FILE (<K(i), P(i)> entries)
INDEX FIELD VALUE
BLOCK POINTER
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
9
5
13
8
6
15
3
17
21
11
16
2
24
10
20
1
4
23
18
14
12
7
19
22

Comparison between indexes

	Number of (First-level) Index Entries	Dense or Nondense	Block Anchoring on the Data File
Primary	Number of blocks in data file	Nondense	Yes
Clustering	Number of distinct index field values	Nondense	Yes/no
Secondary (key)	Number of records in data file	Dense	No

Index data Structures

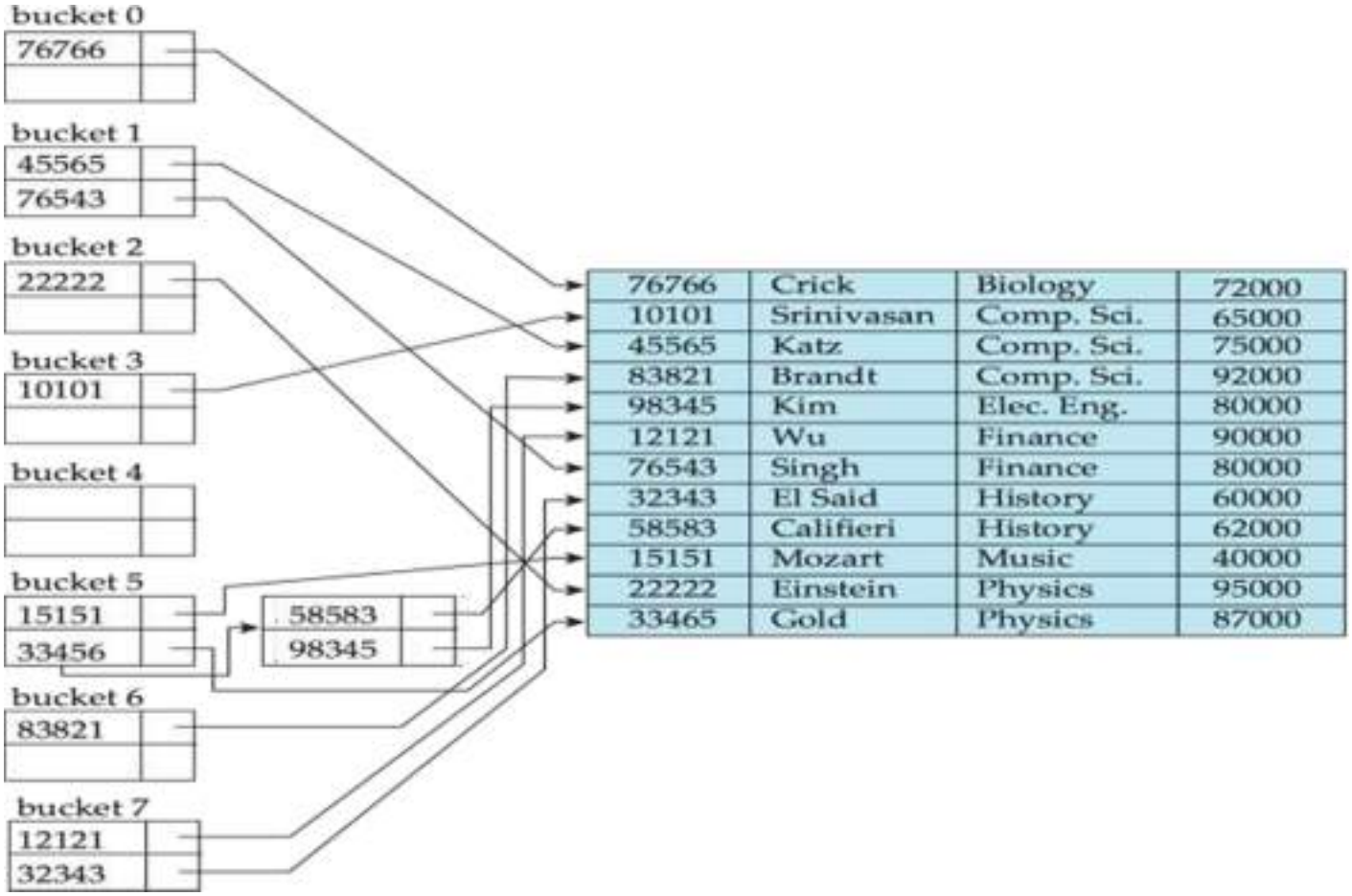
–

Hash Based Indexing

Hash Indices

- Hashing can be used not only for file organization, but also for index-structure creation.
- A **hash index** organizes the search keys, with their associated record pointers, into a hash file structure.
- Strictly speaking, hash indices are always secondary indices
  - if the file itself is organized using hashing, a separate primary hash index on it using the same search-key is unnecessary.
  - However, we use the term hash index to refer to both secondary index structures and hash organized files.

Example of Hash Index



hash index on *instructor,* on attribute *ID*

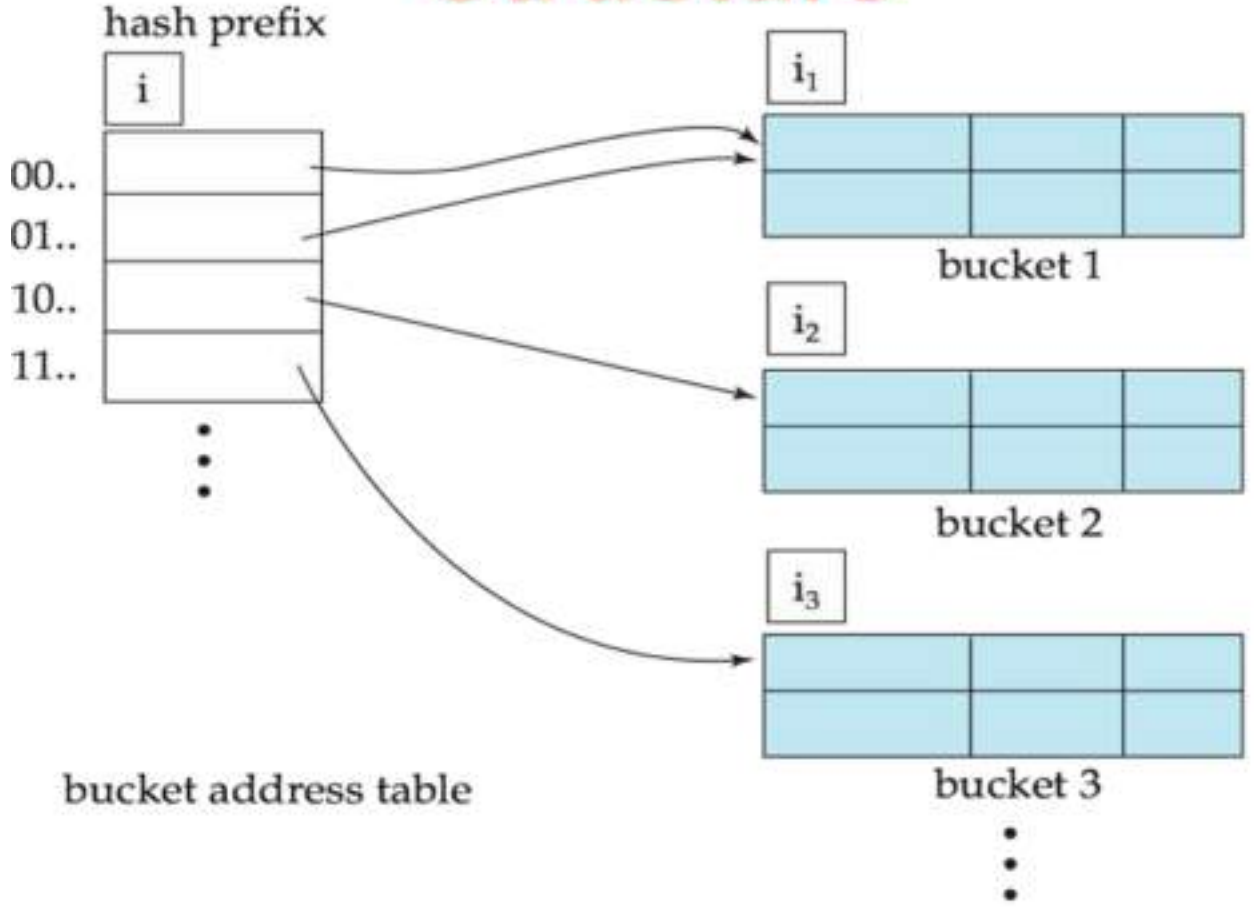
Deficiencies of Static Hashing

- In static hashing, function *h* maps search-key values to a fixed set of *B* of bucket addresses. Databases grow or shrink with time.
  - If initial number of buckets is too small, and file grows, performance will degrade due to too much overflows.
  - If space is allocated for anticipated growth, a significant amount of space will be wasted initially (and buckets will be under full).
  - If database shrinks, again space will be wasted.
- One solution: periodic re-organization of the file with a new hash function
  - Expensive, disrupts normal operations
- Better solution: allow the number of buckets to be modified dynamically.

Dynamic Hashing

- Good for database that grows and shrinks in size
- Allows the hash function to be modified dynamically
- **Extendable hashing** – one form of dynamic hashing
  - Hash function generates values over a large range – typically *b*-bit integers, with $b = 32$.
  - At any time use only a prefix of the hash function to index into a table of bucket addresses.
  - Let the length of the prefix be *i* bits, $0 \leq i \leq 32$.
    - Bucket address table size = 2^{i}. Initially $i = 0$
    - Value of *i* grows and shrinks as the size of the database grows and shrinks.
  - Multiple entries in the bucket address table may point to a bucket (why?)
  - Thus, actual number of buckets is $< 2^{i}$
    - The number of buckets also changes dynamically due to coalescing and splitting of buckets.

General Extendable Hash Structure



In this structure, $i_2 = i_3 = i$, whereas $i_1 = i - 1$ (see next slide for details)

Use of Extendable Hash Structure

- Each bucket *j* stores a value i_j
  - All the entries that point to the same bucket have the same values on the first i_j bits.
- To locate the bucket containing search-key K_j:
  1. Compute $h(K_j) = X$
  2. Use the first *i* high order bits of *X* as a displacement into bucket address table, and follow the pointer to appropriate bucket
- To insert a record with search-key value K_j
  - follow same procedure as look-up and locate the bucket, say *j*.
  - If there is room in the bucket *j* insert record in the bucket.
  - Else the bucket must be split and insertion re-attempted (next slide.)
    - Overflow buckets used instead in some cases (will see shortly)

Insertion in Extendable Hash Structure (Cont)

- To split a bucket j when inserting record with search-key value K_j:
- If $i > i_j$ (more than one pointer to bucket j)
  - allocate a new bucket z, and set $i_j = i_z = (i_j + 1)$
  - Update the second half of the bucket address table entries originally pointing to j, to point to z
  - remove each record in bucket j and reinsert (in j or z)
  - Re-compute new bucket for K_j and insert record in the bucket (further splitting is required if the bucket is still full)
- If $i = i_j$ (only one pointer to bucket j)
  - If i reaches some limit b, or too many splits have happened in this insertion, create an overflow bucket
  - Else
    - increment i and double the size of the bucket address table.
    - replace each entry in the table by two entries that point to the same bucket.
    - recompute new bucket address table entry for K_j

      Now $i > i_j$ so use the first case above.

Deletion in Extendable Hash Structure

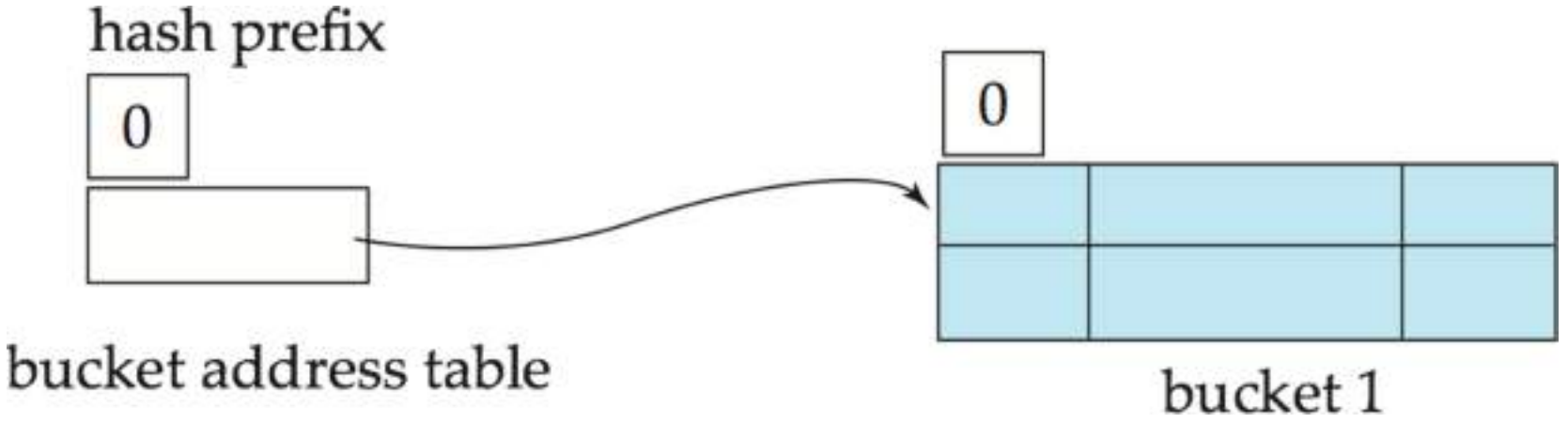
- To delete a key value,
  - locate it in its bucket and remove it.
  - The bucket itself can be removed if it becomes empty (with appropriate updates to the bucket address table).
  - Coalescing of buckets can be done (can coalesce only with a "*buddy*" bucket having same value of i_j and same $i_j - 1$ prefix, if it is present)
  - Decreasing bucket address table size is also possible
    - Note: decreasing bucket address table size is an expensive operation and should be done only if number of buckets becomes much smaller than the size of the table

Use of Extendable Hash Structure: Example

dept_name	h(*dept_name*)
Biology	0010 1101 1111 1011 0010 1100 0011 0000
Comp. Sci.	1111 0001 0010 0100 1001 0011 0110 1101
Elec. Eng.	0100 0011 1010 1100 1100 0110 1101 1111
Finance	1010 0011 1010 0000 1100 0110 1001 1111
History	1100 0111 1110 1101 1011 1111 0011 1010
Music	0011 0101 1010 0110 1100 1001 1110 1011
Physics	1001 1000 0011 1111 1001 1100 0000 0001

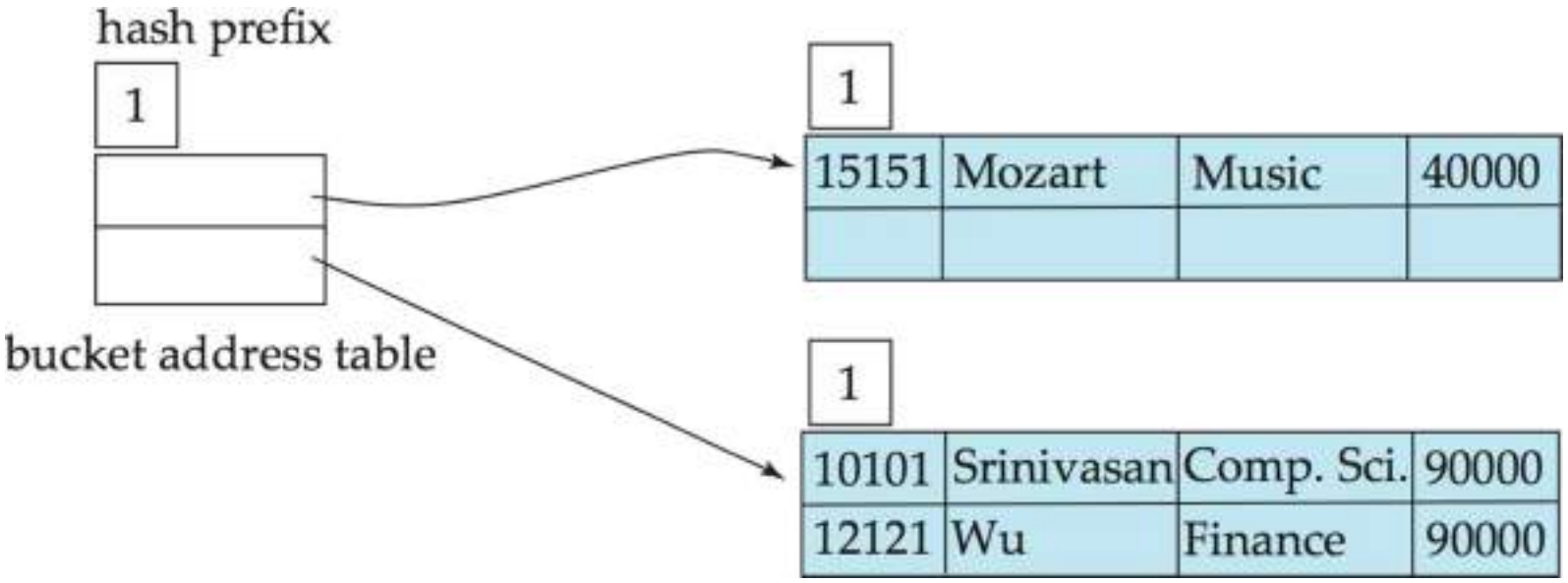
Example (Cont.)

- Initial Hash structure; bucket size = 2



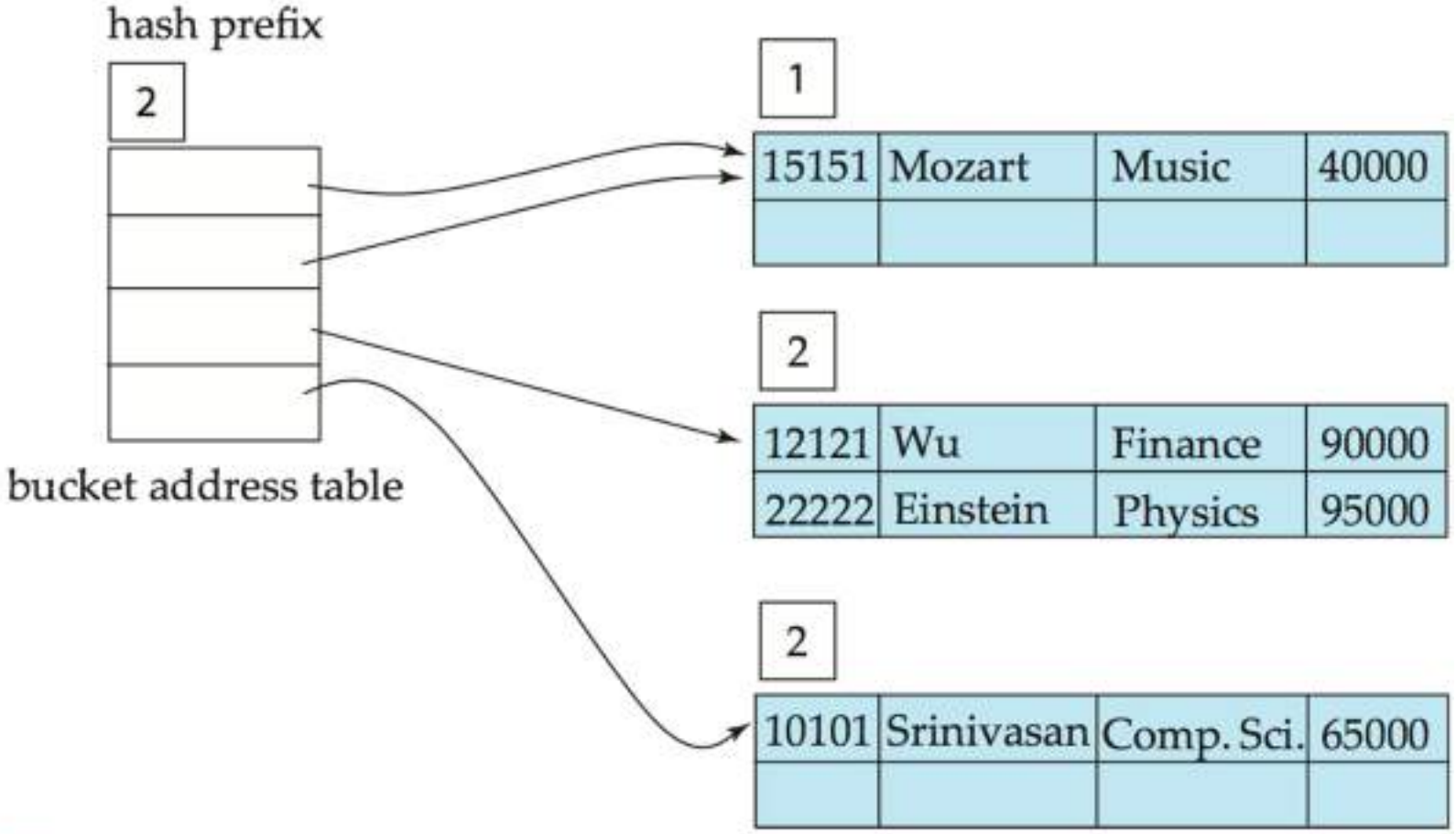
Example (Cont.)

- Hash structure after insertion of “Mozart”, “Srinivasan”, and “Wu” records



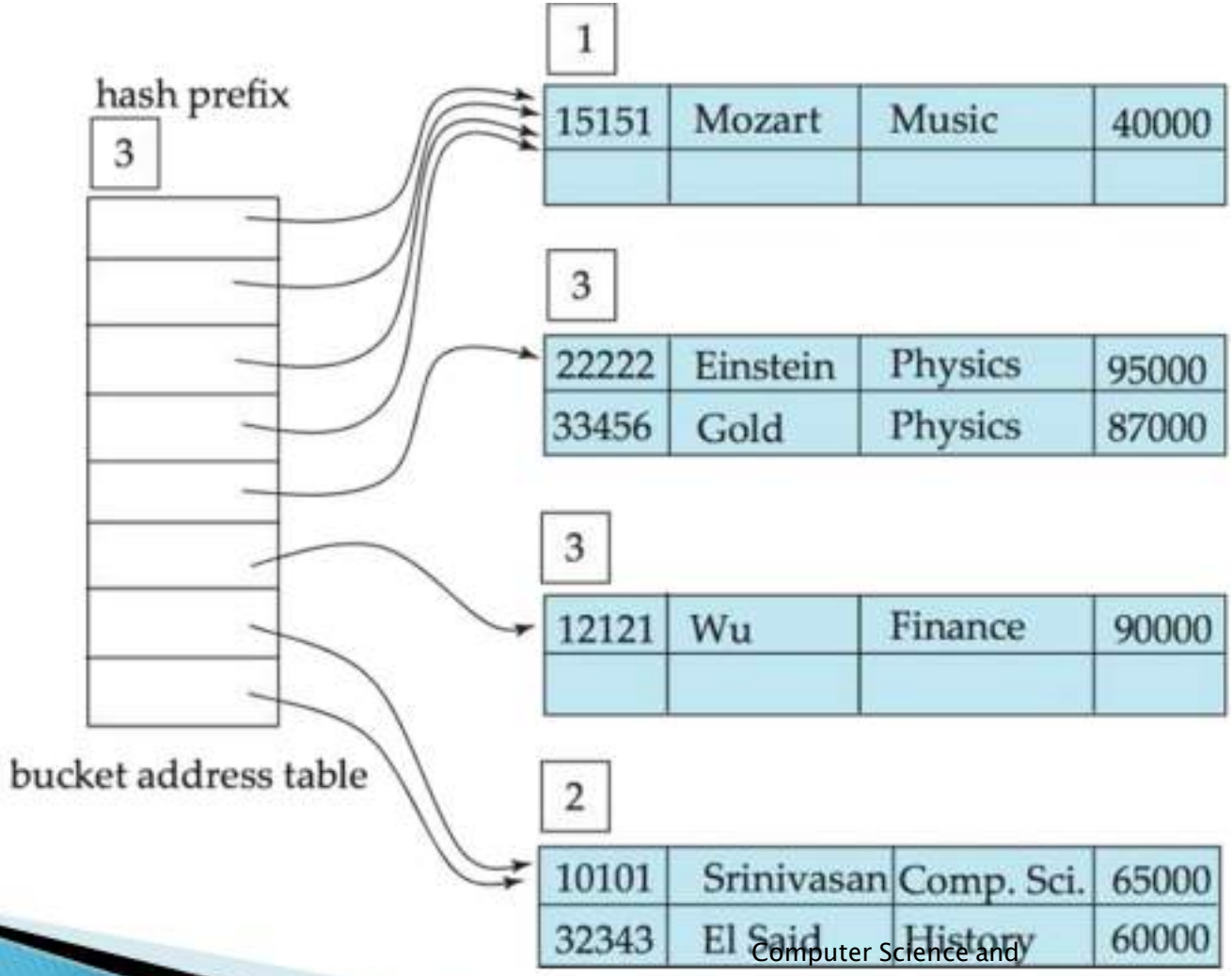
Example (Cont.)

- Hash structure after insertion of Einstein record



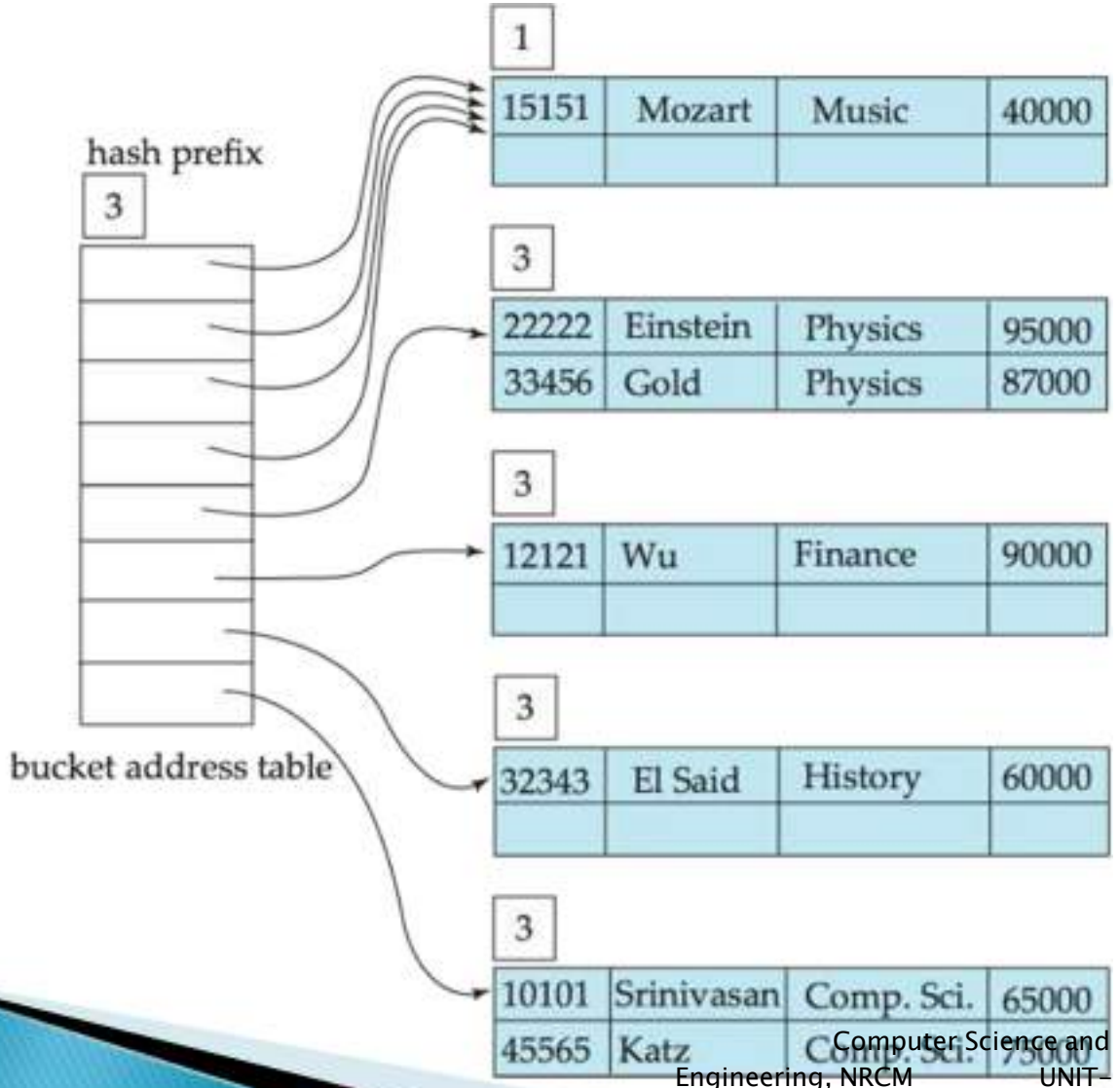
Example (Cont.)

- Hash structure after insertion of Gold and El Said records

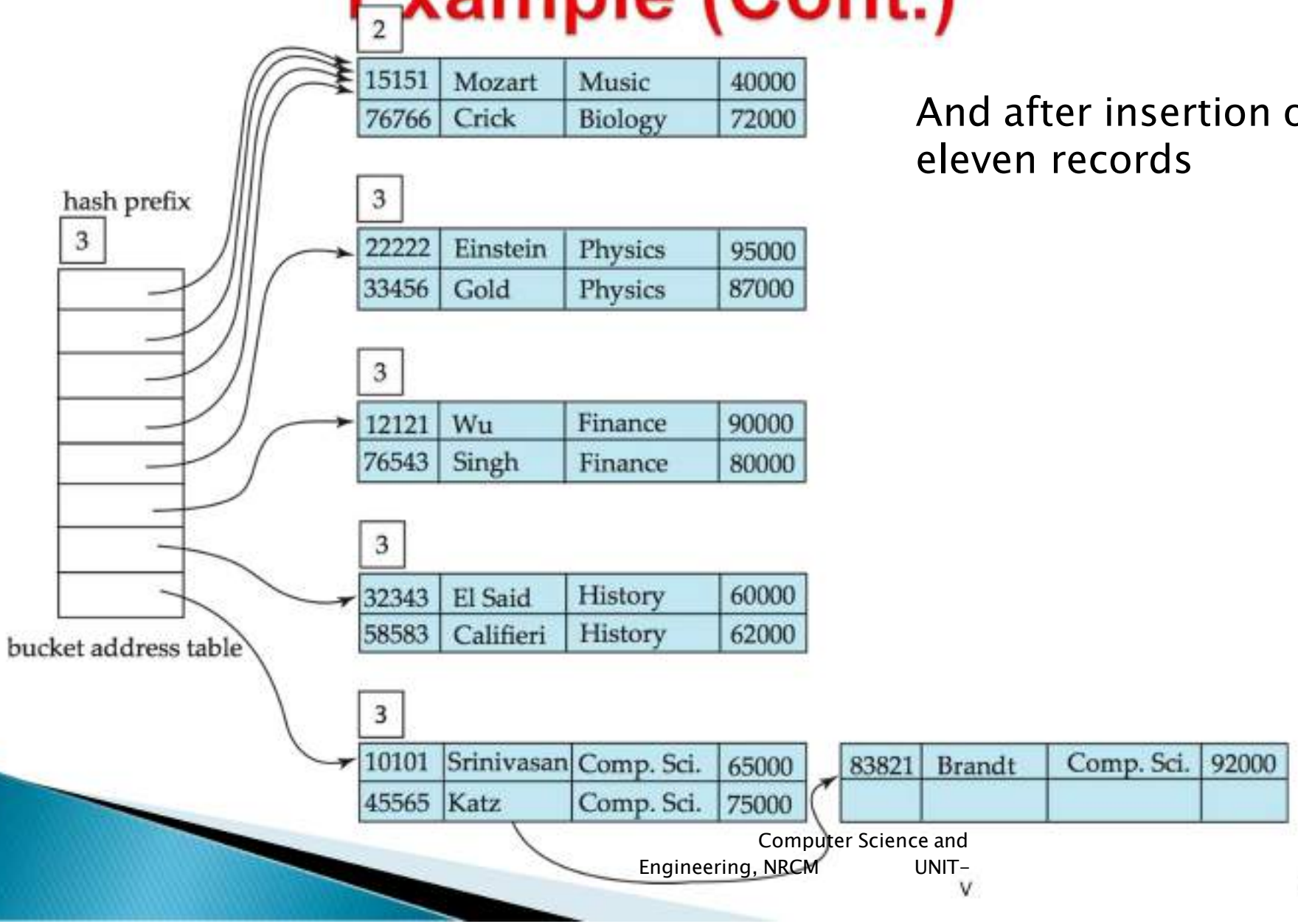


Example (Cont.)

- Hash structure after insertion of Katz record

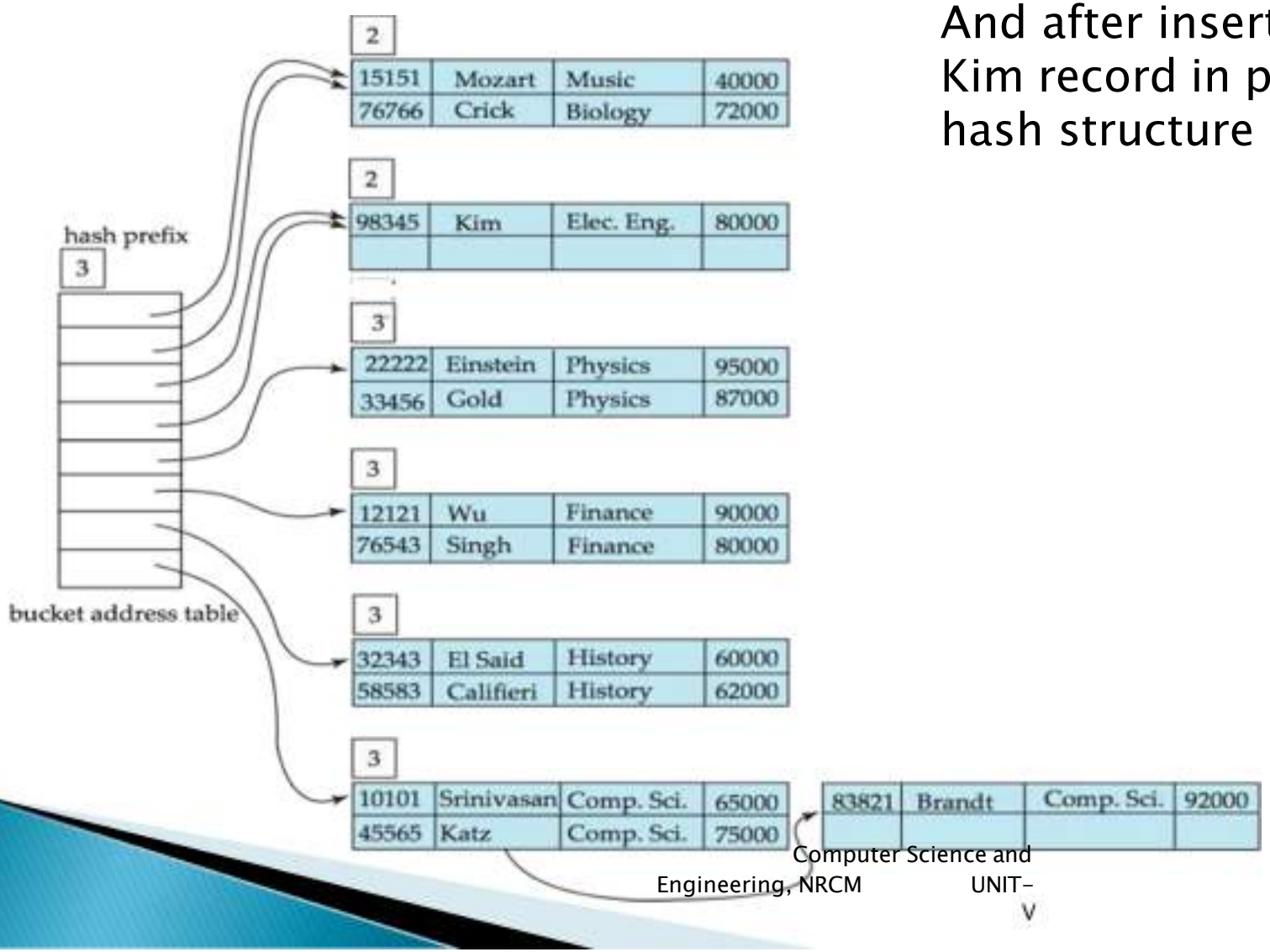


Example (Cont.)



And after insertion of eleven records

Example (Cont.)



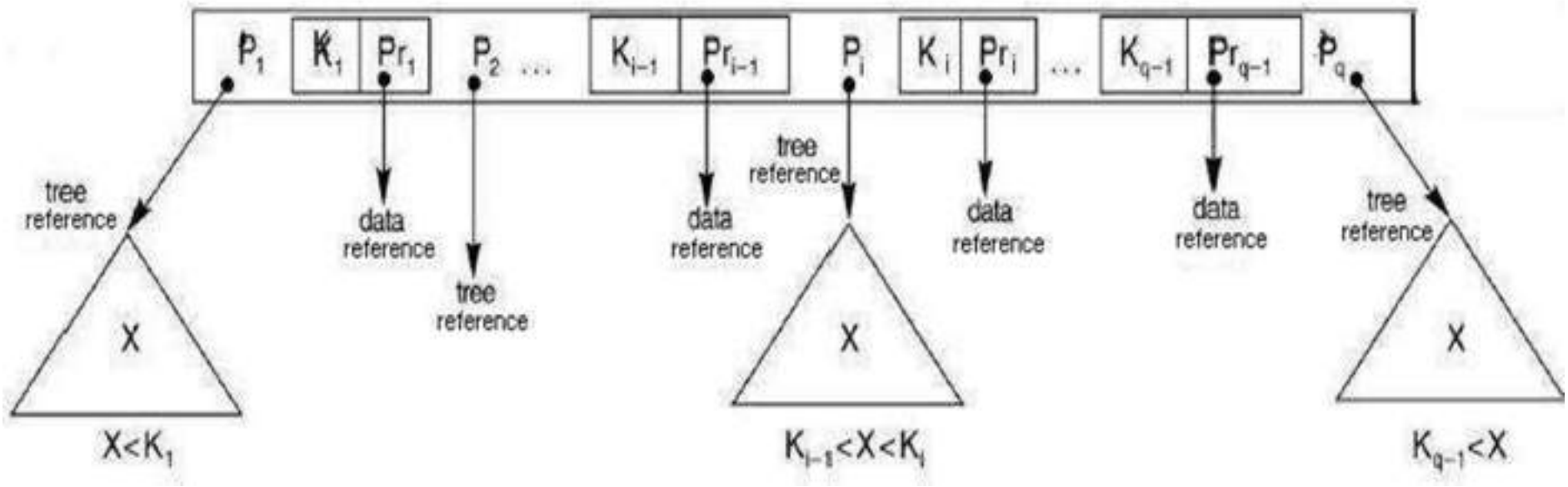
And after insertion of Kim record in previous hash structure

B tree

A B-tree of order m (the maximum number of children for each node) is a tree which satisfies the following properties:

- Every node has at most m children.
- Every node (except root and leaves) has at least $^{m}/_{2}$ children.
- The root has at least two children if it is not a leaf node.
- All leaves appear in the same level, and carry information.
- A non-leaf node with k children contains k–1 keys.
- B-trees are always height balanced, with all leaf nodes at the same level.
- Update and search operations affect only a few disk pages, so performance is good.

The node structure of B tree

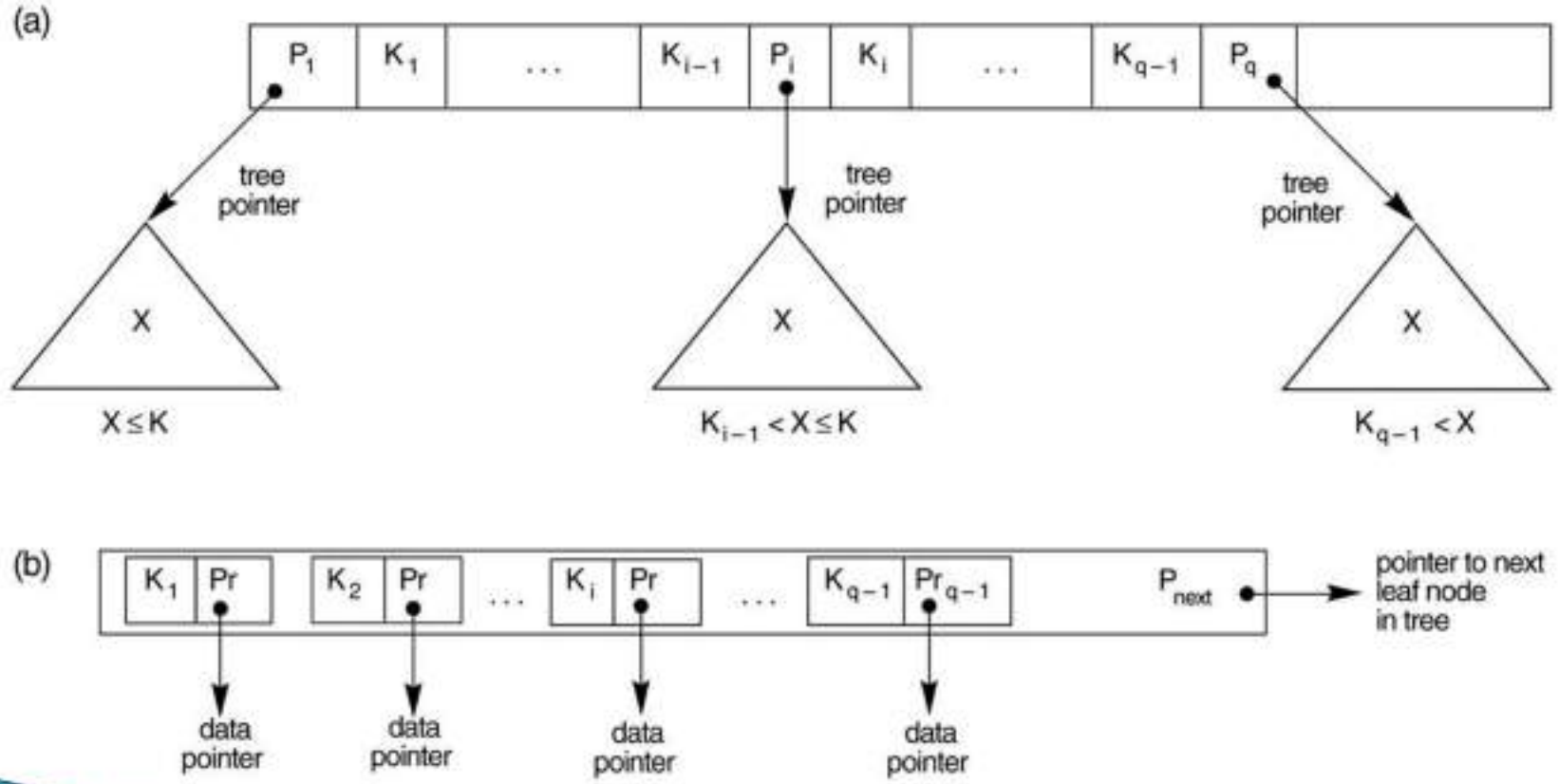


- Note:
  - Corresponding to each key there is a data reference that refers to the data record for that key in secondary memory.
  - In our representations we will omit the data references.

B^+ tree

- Similar to B trees, with a few slight differences.
- B^+ tree is a balanced tree in which the internal nodes direct the search and the leaf nodes contain the data entries.

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.



Difference between B-tree and B+-tree

- In a B-tree, pointers to data records exist at all levels of the tree. In a B+-tree, all pointers to data records exists at the leaf-level nodes
- A B+-tree can have less levels (or higher capacity of search values) than the corresponding B-tree.
- In a B-tree insertion and deletion are simple process. In B+ tree insertion and deletion are complex process.
- In a B-tree, internal nodes store both the keys and data value. In B+ tree Internal nodes store just keys.

Comparison of File Organizations

Comparing File Organizations

- Heap files (random order; insert at eof)
- Sorted files, sorted on *<age, sal>*
- Clustered B+ tree file, Alternative (1), search key *<age, sal>*
- Heap file with unclustered B + tree index on search key *<age, sal>*
- Heap file with unclustered hash index on search key *<age, sal>*

Operations to Compare

- Scan: Fetch all records from disk
- Equality search (e.g., “age = 30”)
- Range selection (e.g., “age > 30”)
- Insert a record
- Delete a record

Parameters of the Analysis

	B = # data pages	R = #records/page	D = disk page I/O time	C = process single record	H = apply Hash function	F = index tree fan-out
Typical value			15 mlsec	100 nanosec	100 nanosec	100

Assumptions in Our Analysis

- Heap Files:
  - Equality selection on key; exactly one match.
- Sorted Files:
  - Files compacted after deletions.
  - Clustered files: pages typically 67% full.
  - ⇒ Total number pages needed = 1.5 B.
- Indexes:
  - Alt (2), (3): data entry size = 10% size of record
  - Hash: No overflow buckets.
    - 80% page occupancy.
    - ⇒ Index size = 1.25 B data size.
    - ⇒ #data entries/page = 10 (0.8R) = 8R.
  - Tree: 67% page occupancy of index pages (this is typical).
    - ⇒ #leaf pages = (1.5 B) 0.1 = 0.15 B.
    - ⇒ #data entries/page = 10 (0.67R) = 6.7R.

Scanning Cost

- Heap file: B(D + RC).
  - for each page (B)
  - Read the page (D)
  - For each record (R), process the record (C).
- Sorted File: B(D + RC).
  - Have to go through all pages.
- Clustered File: 1.5B (D+RC).
  - Pages only 67% full.
- Unclustered Tree Index: >BR(D+C). Bad!
  - for each record (BR)
  - retrieve page and find record (D + C).

Exercise for Group Work

1. Estimate how long an equality search takes in
   (i) a heap file (ii) a sorted file (iii) a hash file, hashed on the search key, with at most one record matching the search key (i.e., the search is on a key field).

2. Estimate how long an insertion takes in
   (i) a heap file (ii) a sorted file (iii) a hash file.

	B = # data pages	R = #records/page	D = disk page I/O time	C = process single record	H = apply Hash function	F = index tree fan-out
Typical value			15 msec	100 nanosec	100 nanosec	100

Cost of Operations

	(a) Scan	(b) Equality	(c) Range	(d) Insert	(e) Delete
(1) Heap					
(2) Sorted					
(3) Clustered					
(4) Unclustered Tree index					
(5) Unclustered Hash index					

Indexes and Performance Tuning

8.5 Indexes and Performance Tuning

- One of a DBA's most important duties is to deal with performance problems
- One of the most effective methods for improving performance is to choose the best indexes for the given workload
- Why not index everything?
  - What are the two costs of an index?
- Part of a DBA's job: choose indexes so the database will "run fast".
  - What information does the DBA need, to do this?

Choice of Indexes

- What indexes should we create?
  - Which relations should have indexes? What field(s) should be the search key? Should we build several indexes?
- What kinds of indexes should we create?
  - Clustered? Hash/tree?
- Before creating an index, must also consider the impact on updates in the workload!
  - Trade-off: Indexes can make queries go faster, updates slower. Require disk space, too.
- Also consider dropping indexes

8.5.1 Understanding the Workload

- For each query in the workload:
  - Which relations does it access?
  - Which attributes are retrieved?
  - Which attributes are involved in selection/join conditions? How selective are these conditions likely to be?
- For each update in the workload:
  - Which attributes are involved in selection/join conditions? How selective are these conditions likely to be?
  - The type of update (INSERT/DELETE/UPDATE), and the attributes that are affected.

Composite Search Keys

- To retrieve Emp records with *age*=30 AND *sal*=4000, an index on <*age,sal*> would be better than an index on *age* or an index on *sal*.
  - Choice of index key orthogonal to clustering etc.
- If condition is: 20<*age*<30 AND 3000<*sal*<5000:
  - Clustered tree index on <*age,sal*> or <*sal,age*> is best.
- If condition is: *age*=30 AND 3000<*sal*<5000:
  - Clustered <*age,sal*> index much better than <*sal,age*> index!

- Composite indexes are larger, updated more often.

Indexes in the real world

- Types of indexes supported
  - Oracle, SQLServer and DB2 support only B+Tree indexes. Postgres supports hash indexes but does not recommend using them.
  - MySQL?
  - Everyone uses hash indexes for hash joins, but they are constructed on the fly.

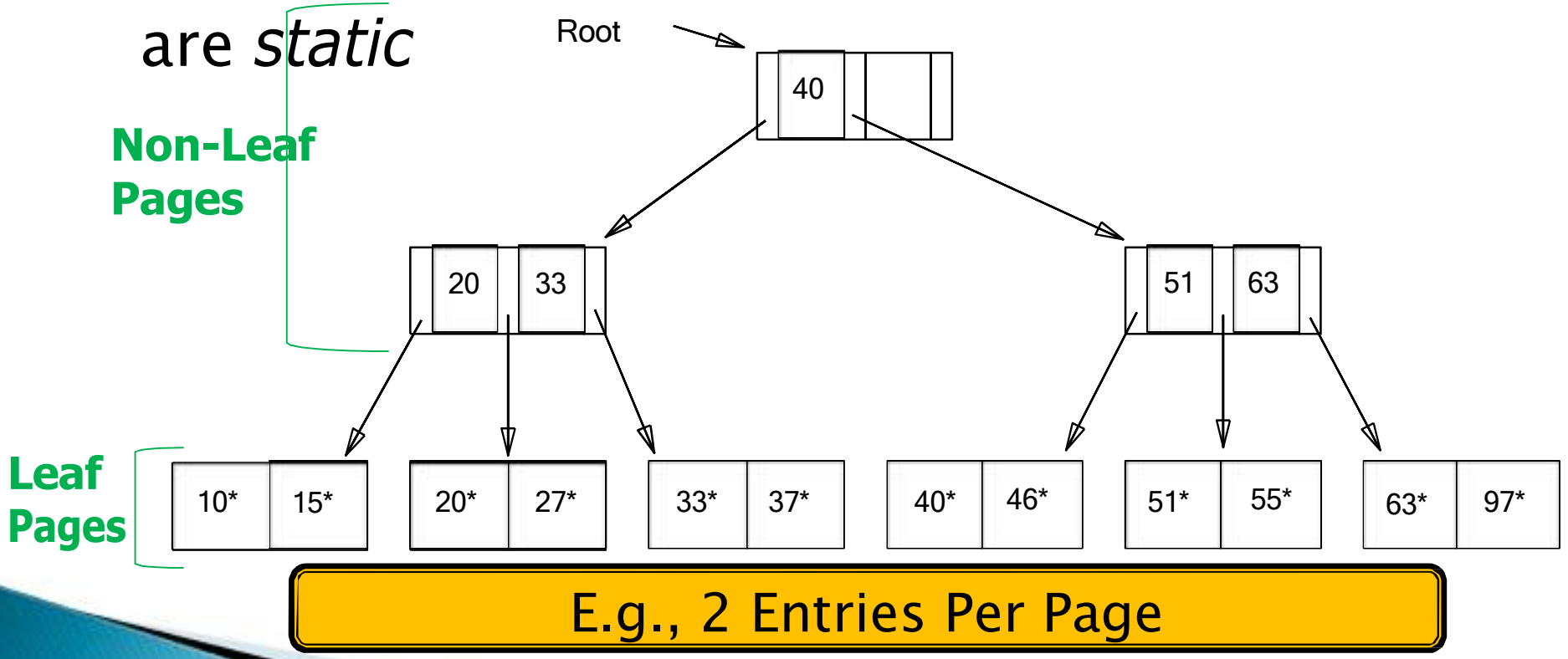
Clustering in the real world

- Oracle, SQL Server:
  - Declares a clustered index on any primary key.
  - It uses alternative 1.
- DB2, Postgres
  - The user can define an index to be clustered.
  - The DBMS uses alternative 2.
  - At first the index will be clustered.
  - If you also specify a percentage of free space, it will place subsequent inserts/updates (Postgres:updates only) in sorted order so the index should stay clustered for a while.
  - It's up to you to recluster the index if overflow chains get to be long.
- MySQL

Indexed Sequential Access Methods (ISAM)

ISAM Trees

- Indexed Sequential Access Method (ISAM) trees are *static*



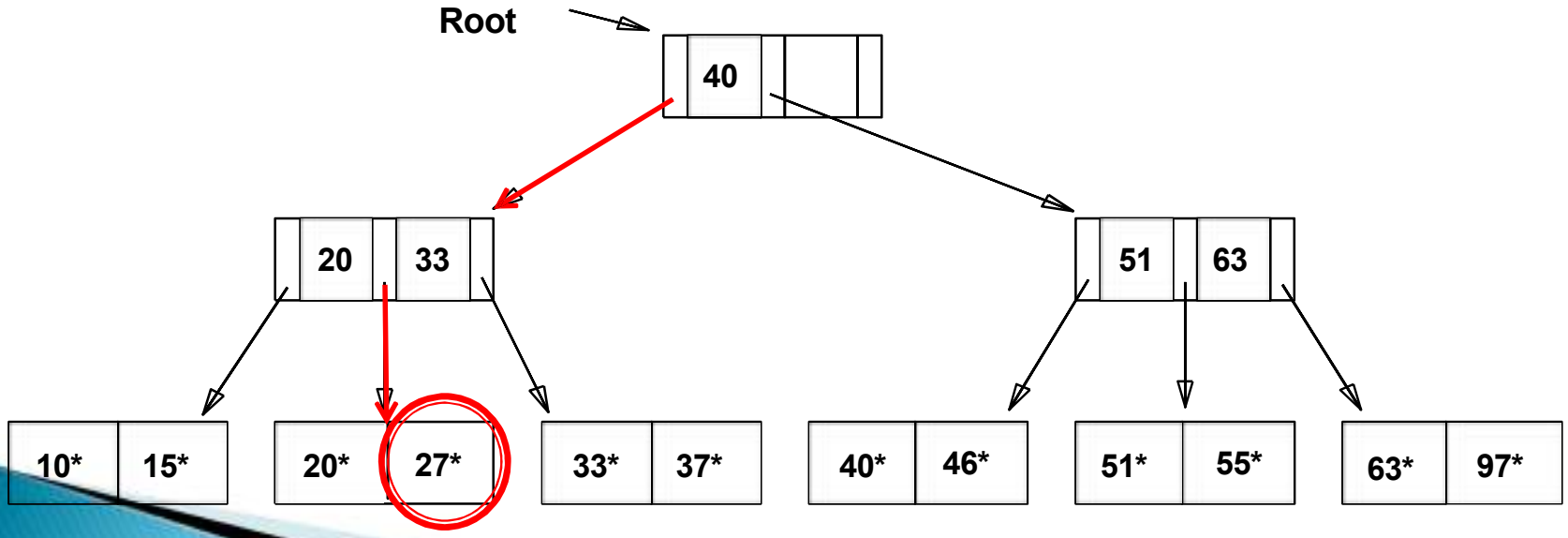
ISAM File Creation

- How to create an ISAM file?
  - All leaf pages are allocated *sequentially* and *sorted* on the search key value
  - If Alternative (2) or (3) is used, the data records are created and *sorted* before allocating leaf pages
  - The non-leaf pages are subsequently allocated

ISAM: Searching for Entries

- Search begins at root, and key comparisons direct it to a leaf

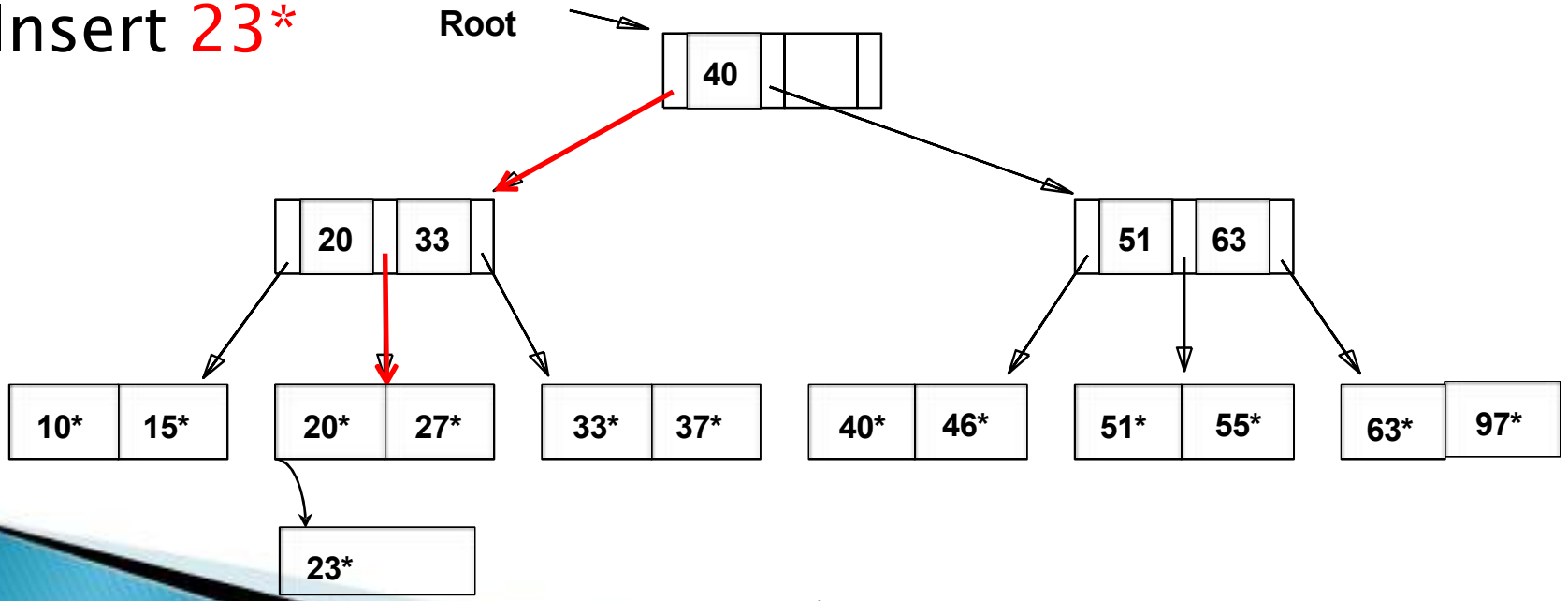
- Search for 27*



ISAM: Inserting Entries

- The appropriate page is determined as for a search, and the entry is inserted (with overflow pages added if necessary)

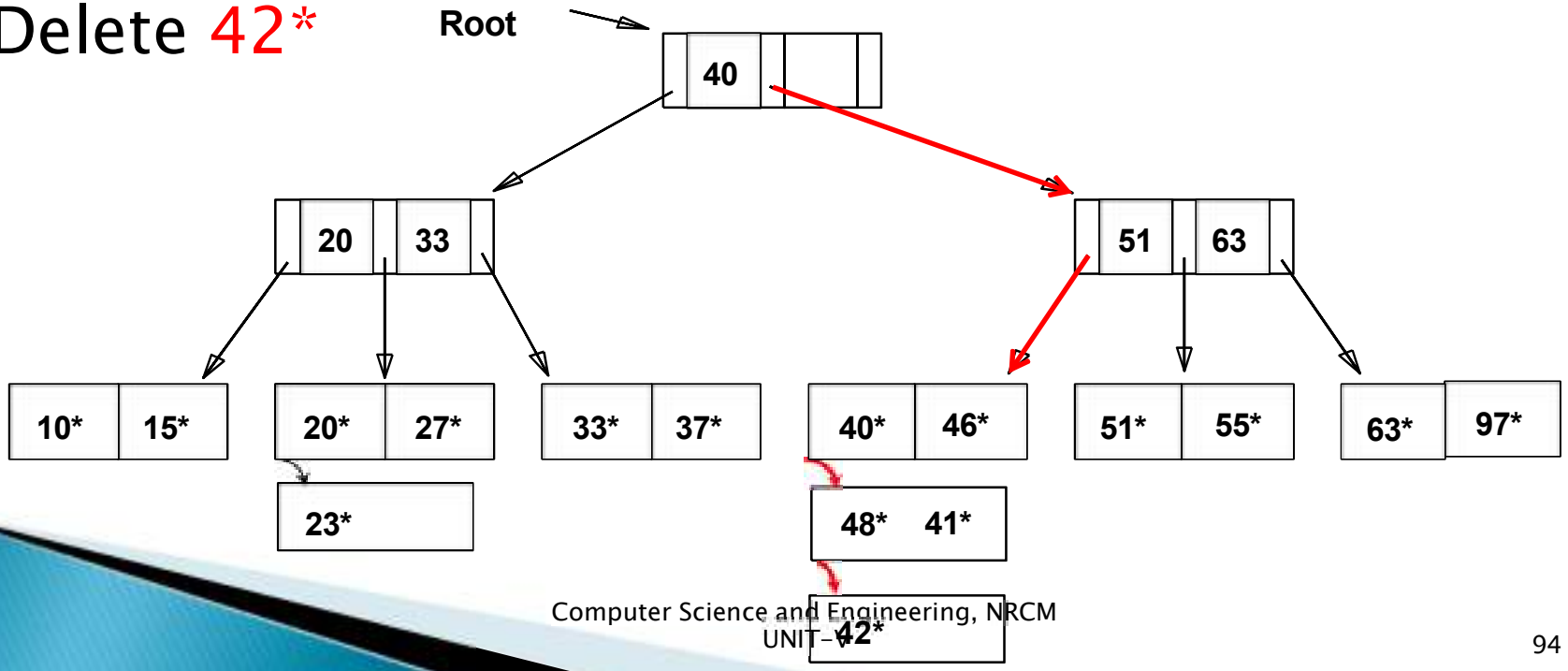
- Insert 23*



ISAM: Deleting Entries

- The appropriate page is determined as for a search, and the entry is deleted (*with ONLY overflow pages removed when becoming empty*)

- Delete 42*



ISAM: Some Issues

- Once an ISAM file is created, insertions and deletions affect only the contents of leaf pages (i.e., *ISAM is a static structure*!)
- Since index-level pages are *never* modified, there is no need to *lock* them during insertions/deletions
  - Critical for concurrency!
- Long overflow chains can develop easily
  - The tree can be initially set so that ~20% of each page is free
- If the data distribution and size are relatively static, ISAM might be a good choice to pursue