

# Overview of Database Management System



# Data vs. Information

- Data:
  - Raw facts; building blocks of information
  - Unprocessed information
- Information:
  - Data processed to reveal meaning
- Accurate, relevant, and timely information is key to good decision making
- Good decision making is key to survival in global environment

# Introducing the Database and the DBMS

- Database—shared, integrated computer structure that houses:
  - End user data (raw facts)
  - Metadata (data about data)

# Introducing the Database and the DBMS (continued)

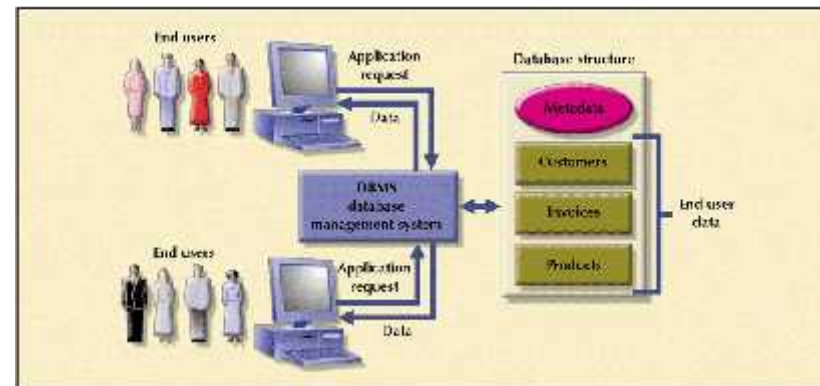
- DBMS (database management system):
  - Collection of programs that manages database structure and controls access to data
  - Possible to share data among multiple applications or users
  - Makes data management more efficient and effective

# DBMS Makes Data Management More Efficient and Effective

- End users have better access to more and better-managed data
  - Promotes integrated view of organization's operations
  - Probability of data inconsistency is greatly reduced
  - Possible to produce quick answers to ad hoc queries

# The DBMS Manages the Interaction Between the End User and the Database

FIGURE 1.2 THE DBMS MANAGES THE INTERACTION BETWEEN THE END USER AND THE DATABASE



# Database Management System (DBMS)

- Collection of interrelated data
- Set of programs to access the data
- DBMS contains information about a particular enterprise
- DBMS provides an environment that is both *convenient* and *efficient* to use.
- Database Applications:
  - Banking: all transactions
  - Airlines: reservations, schedules
  - Universities: registration, grades
  - Sales: customers, products, purchases
  - Manufacturing: production, inventory, orders, supply chain
  - Human resources: employee records, salaries, tax deductions
- Databases touch all aspects of our lives

# Evolution of Simple File System

- As number of databases increased, small file system evolved
- Each file used its own application programs
- Each file was owned by individual or department who commissioned its creation



# Problems with File System Data Management

- Every task requires extensive programming in a third-generation language (3GL)
  - Programmer must specify task and *how* it must be done
- Modern databases use fourth-generation language (4GL)
  - Allows user to specify what must be done *without specifying how* it is to be done

# Programming in 3GL

- Time-consuming, high-level activity
- Programmer must be familiar with physical file structure
- As system becomes complex, access paths become difficult to manage and tend to produce malfunctions
- Complex coding establishes precise location of files and system components and data characteristics

# Programming in 3GL (continued)

- Ad hoc queries are impossible
- Writing programs to design new reports is time consuming
- As number of files increases, system administration becomes difficult
- Making changes in existing file structure is difficult
- File structure changes require modifications in all programs that use data in that file

# Programming in 3GL (continued)

- Modifications are likely to produce errors, requiring additional time to “debug” the program
- Security features hard to program and therefore often omitted

# Structural and Data Dependence

- Structural dependence
  - Access to a file depends on its structure
- Data dependence
  - Changes in database structure affect program's ability to access data
  - Logical data format
    - How a human being views the data
  - Physical data format
    - How the computer “sees” the data

# Purpose of Database System

- To summaries ....
- Drawbacks of using file systems to store data
  - Data redundancy results in data inconsistency
    - Different and conflicting versions of the same data appear in different places
    - Data anomalies develop when required changes in redundant data are not made successfully
    - Data anomalies – Modification, Insertion, Deletion
  - Difficulty in accessing data
    - Need to write a new program to carry out each new task
  - Data isolation — multiple files and formats
  - Integrity problems
    - Integrity constraints (e.g. account balance > 0) become part of program code
    - Hard to add new constraints or change existing ones

# Purpose of Database Systems (Cont.)

- Drawbacks of using file systems (cont.)
  - Atomicity of updates
    - Failures may leave database in an inconsistent state with partial updates carried out
    - E.g. transfer of funds from one account to another should either complete or not happen at all
  - Concurrent access by multiple users
    - Concurrent accessed needed for performance
    - Uncontrolled concurrent accesses can lead to inconsistencies
      - E.g. two people reading a balance and updating it at the same time
  - Security problems
- Database systems offer solutions to all the above problems



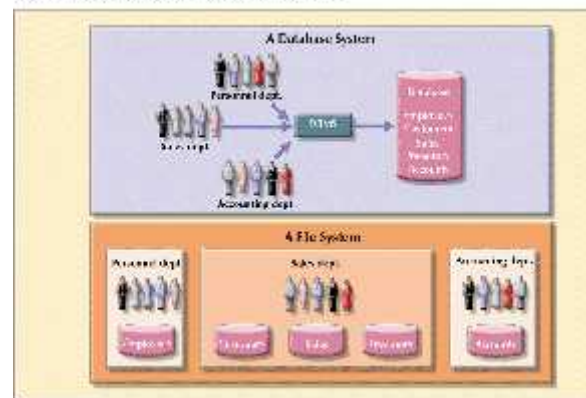
# Database vs. File System

- Problems inherent in file systems make using a database system desirable
- File system
  - Many separate and unrelated files
- Database
  - Logically related data stored in a single logical data repository



# Contrasting Database and File Systems

FIGURE 1.8 CONTRASTING DATABASE AND FILE SYSTEMS



# Basic File Terminology

**TABLE 1.1 BASIC FILE TERMINOLOGY**

TERM	DEFINITION
<b>Data</b>	"Raw" facts, such as a telephone number, a birth date, a customer name, and a year-to-date (YTD) sales value. Data have little meaning unless they have been organized in some logical manner. The smallest piece of data that can be "recognized" by the computer is a single character, such as the letter A, the number 5, or a symbol such as /. A single character requires one byte of computer storage.
<b>Field</b>	A character or group of characters (alphabetic or numeric) that has a specific meaning. A field is used to define and store data.
<b>Record</b>	A logically connected set of one or more fields that describes a person, place, or thing. For example, the fields that constitute a record for a customer named J.D. Rudd might consist of J.D. Rudd's name, address, phone number, date of birth, credit limit, and unpaid balance.
<b>File</b>	A collection of related records. For example, a file might contain data about vendors of ROBCOR Company, or a file might contain the records for the students currently enrolled at Gigantic University.

# DBMS Functions

- Performs functions that guarantee integrity and consistency of data
  - Data dictionary management
    - defines data elements and their relationships
  - Data storage management
    - stores data and related data entry forms, report definitions, etc.
  - Data transformation and presentation
    - translates logical requests into commands to physically locate and retrieve the requested data

# DBMS Functions (continued)

- Security management

- enforces user security and data privacy within database

- Multi-user access control

- creates structures that allow multiple users to access the data

- Backup and recovery management

- provides backup and data recovery procedures

# DBMS Functions (continued)

- Data integrity management
  - promotes and enforces integrity rules to eliminate data integrity problems
- Database access languages and application programming interfaces
  - provides data access through a query language
- Database communication interfaces
  - allows database to accept end-user requests within a computer network environment

# The Database System Environment

- Database system is composed of 5 main parts:
  1. Hardware
  2. Software
    - Operating system software
    - DBMS software
    - Application programs and utility software
  3. People
  4. Procedures
  5. Data

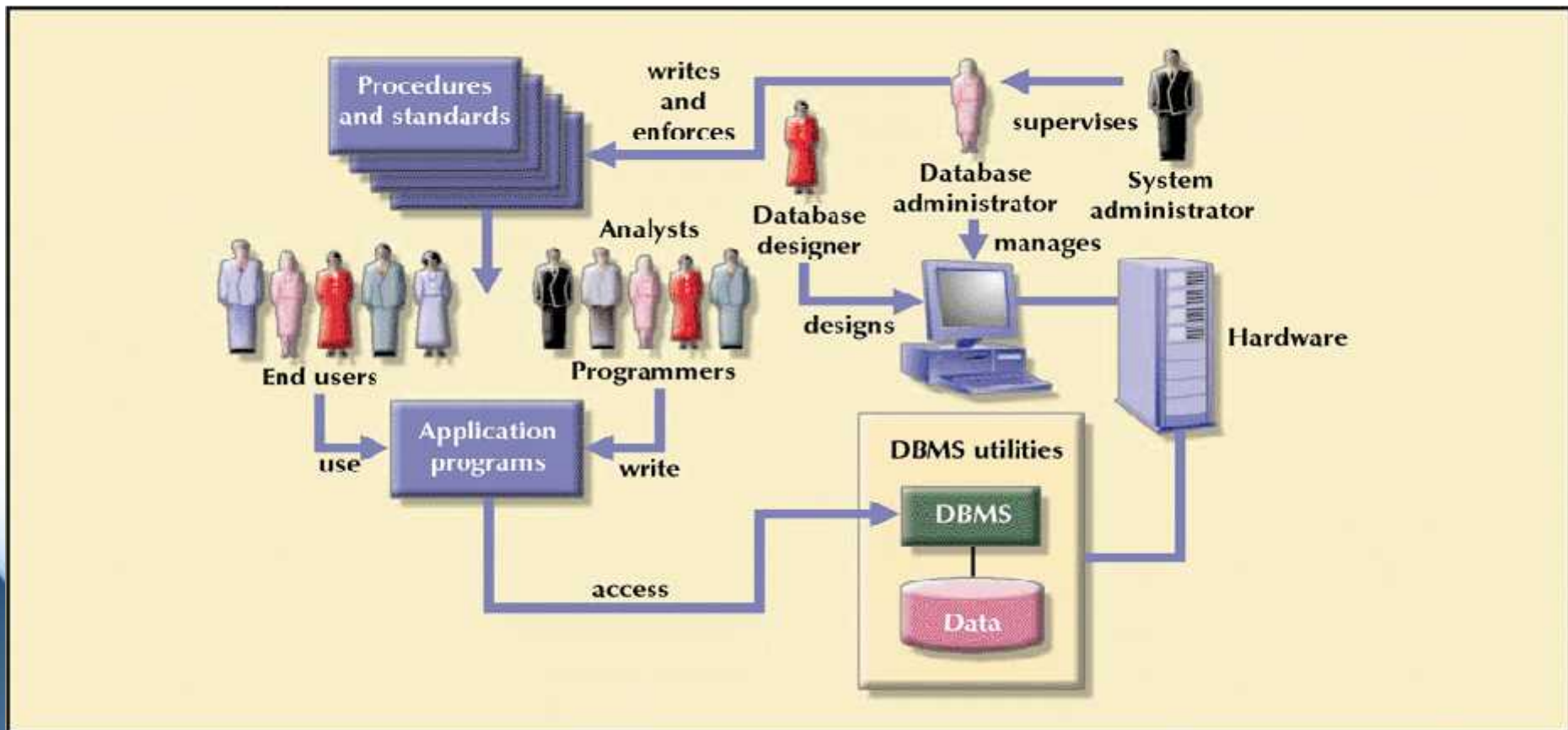
# Database Users

- Users are differentiated by the way they expect to interact with the system
- Application programmers – interact with system through DML calls
- Sophisticated users – form requests in a database query language
- Specialized users – write specialized database applications that do not fit into the traditional data processing framework
- Naïve users – invoke one of the permanent application programs that have been written previously
  - E.g. people accessing database over the web, bank tellers, clerical staff



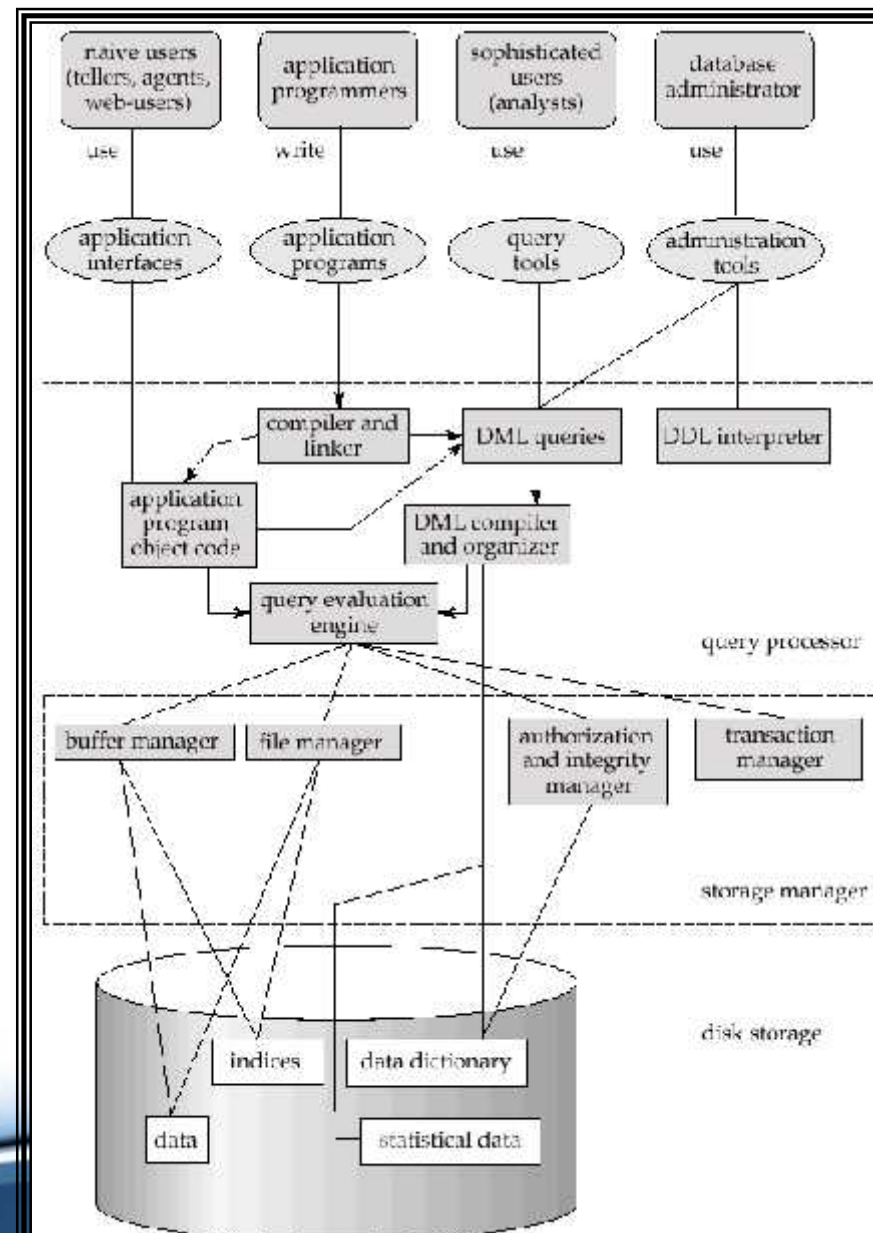
# The Database System Environment (continued)

FIGURE 1.7 THE DATABASE SYSTEM ENVIRONMENT

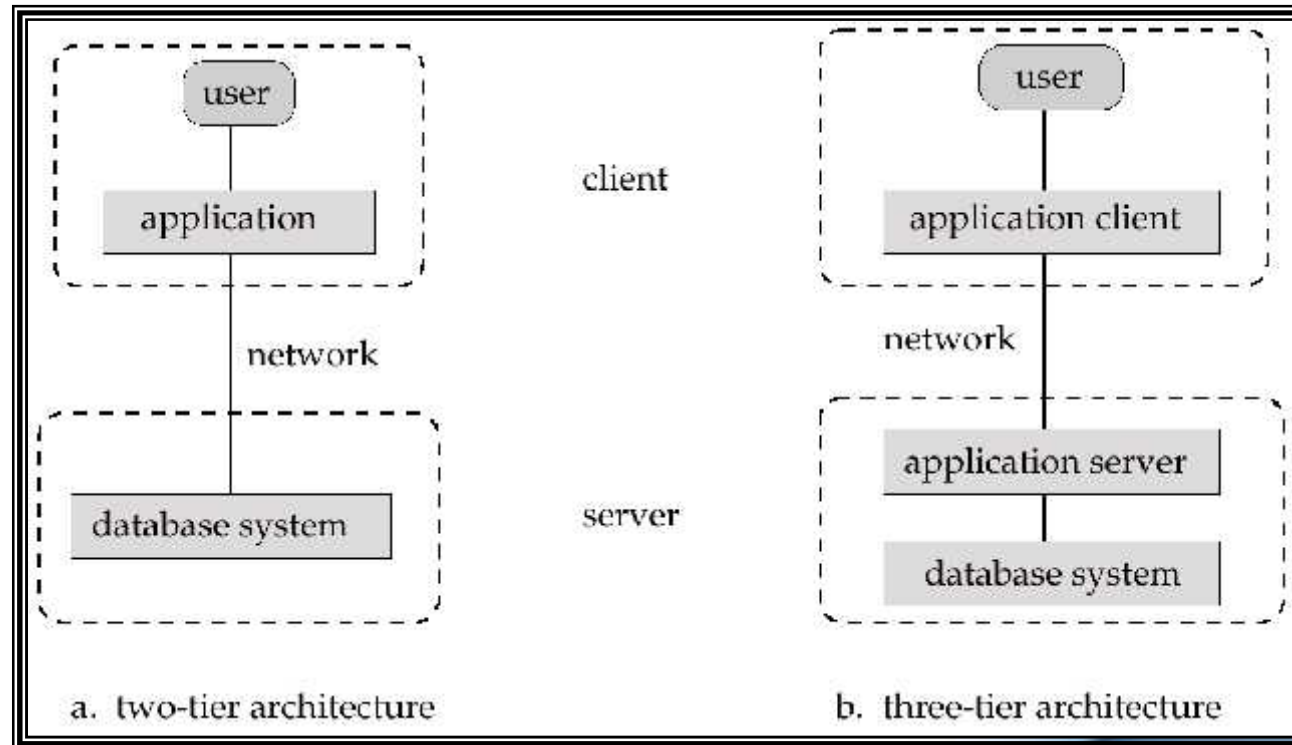




# Overall System Structure



# Application Architectures



- **Two-tier architecture:** E.g. client programs using ODBC/JDBC to communicate with a database
- **Three-tier architecture:** E.g. web-based applications, and applications built using “middleware”

# Levels of Abstraction

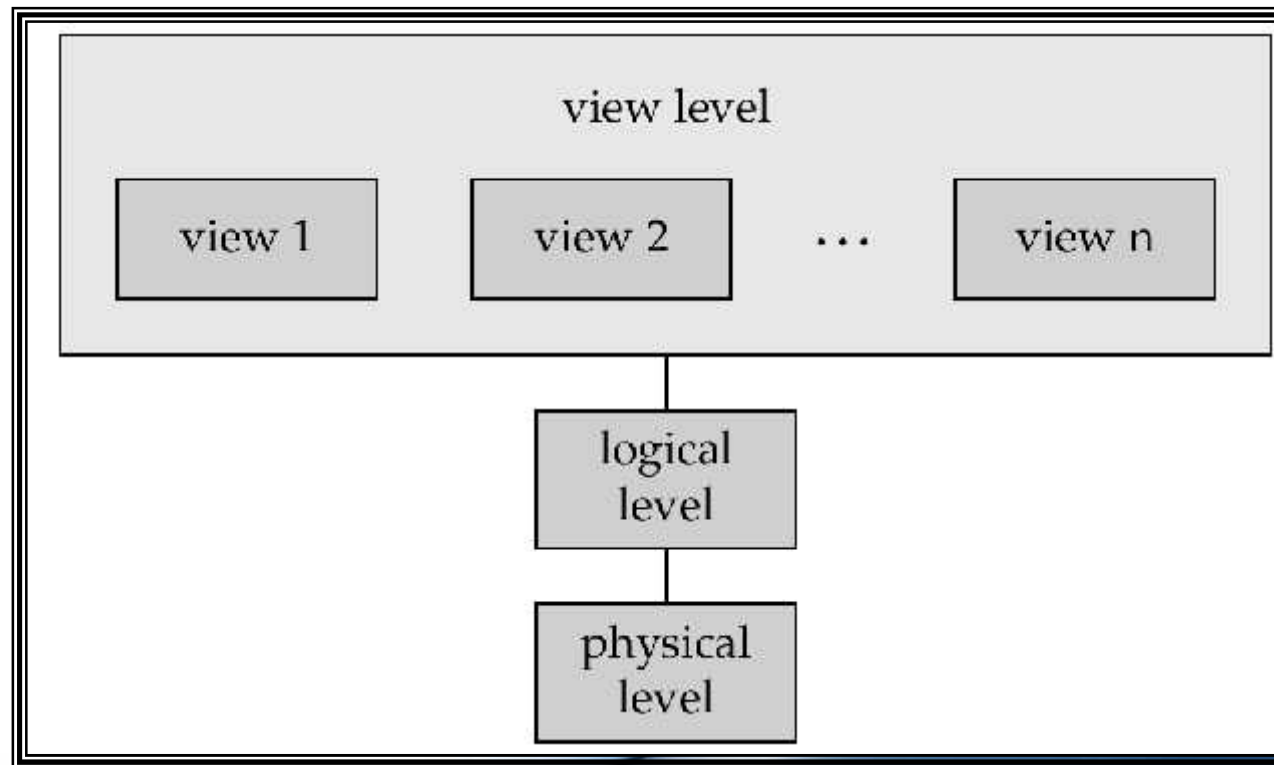
- Physical level describes how a record (e.g., customer) is stored.
- Logical level: describes data stored in database, and the relationships among the data.

```
type customer = record  
    name : string;  
    street : string;  
    city : integer;  
end;
```

- View level: application programs hide details of data types. Views can also hide information (e.g., salary) for security purposes.

# View of Data

An architecture for a database system



# Instances and Schemas

- Similar to types and variables in programming languages
- **Schema** – the logical structure of the database
  - e.g., the database consists of information about a set of customers and accounts and the relationship between them)
  - Analogous to type information of a variable in a program
  - **Physical schema**: database design at the physical level
  - **Logical schema**: database design at the logical level
- **Instance** – the actual content of the database at a particular point in time
  - Analogous to the value of a variable
- **Physical Data Independence** – the ability to modify the physical schema without changing the logical schema
  - Applications depend on the logical schema
  - In general, the interfaces between the various levels and components should be well defined so that changes in some parts do not seriously influence others.

# Data Models

- A collection of tools for describing
  - data
  - data relationships
  - data semantics
  - data constraints
- Entity-Relationship model
- Relational model
- Other models:
  - Object-Oriented model
  - Semi-structured data models
  - Older models: network model and hierarchical model

# The Importance of Data Models

- Data model
  - Relatively simple representation, usually graphical, of complex real-world data structures
  - Communications tool to facilitate interaction among the designer, the applications programmer, and the end user
- Good database design uses an appropriate data model as its foundation
- End-users have different views and needs for data
- Data model organizes data for various users



# The Evolution of Data Models

- Hierarchical
- Network
- Relational
- Entity relationship
- Object oriented



# The Hierarchical Model— Evolution

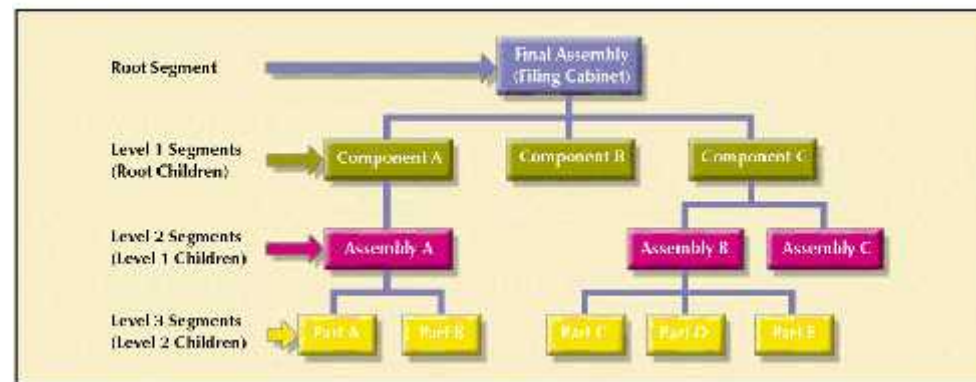
- GUAM (Generalized Update Access Method)
  - Based on the recognition that the many smaller parts would come together as components of still larger components
- Information Management System (IMS)
  - World's leading mainframe hierarchical database system in the 1970s and early 1980s

# The Hierarchical Model— Characteristics

- Basic concepts form the basis for subsequent database development
- Limitations lead to a different way of looking at database design
- Basic concepts show up in current data models
- Best understood by examining manufacturing process

# A Hierarchical Structure

FIGURE 2.1 A HIERARCHICAL STRUCTURE



# Hierarchical Structure— Characteristics

- Each parent can have many children
- Each child has only one parent
- Tree is defined by path that traces parent segments to child segments, beginning from the left
- Hierarchical path
  - Ordered sequencing of segments tracing hierarchical structure
- Preorder traversal or hierarchic sequence
  - “Left-list” path

# The Hierarchical Model

- Advantages
  - Conceptual simplicity
  - Database security
  - Data independence
  - Database integrity
  - Efficiency

# The Hierarchical Model (continued)

- Disadvantages
  - Complex implementation
  - Difficult to manage
  - Lacks structural independence
  - Complex applications programming and use
  - Implementation limitations
  - Lack of standards

# The Network Model

- Created to
  - Represent complex data relationships more effectively
  - Improve database performance
  - Impose a database standard
- Conference on Data Systems Languages (CODASYL) (In late 1960s)
- American National Standards Institute (ANSI)
- Database Task Group (DBTG)

# Crucial Database Components

- Schema
  - Conceptual organization of entire database as viewed by the database administrator
- Subschema
  - Defines database portion “seen” by the application programs that actually produce the desired information from data contained within the database
- Data Management Language (DML)
  - Define data characteristics and data structure in order to manipulate the data



# Data Management Language Components

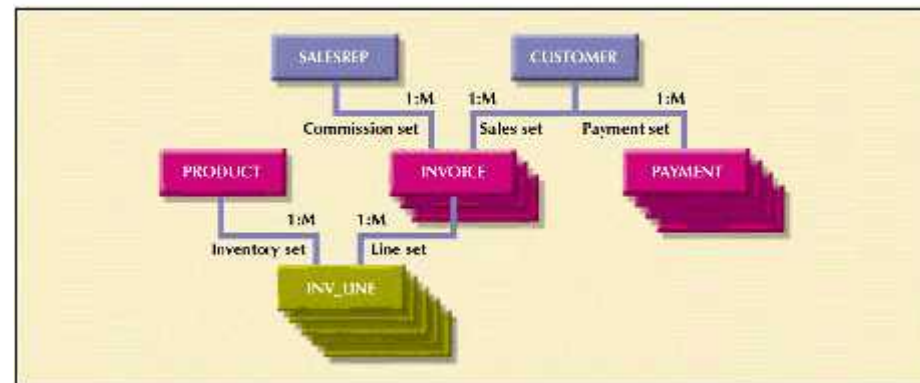
- Schema Data Definition Language (DDL)
  - Enables database administrator to define schema components
- Subschema DDL
  - Allows application programs to define database components that will be used
- DML
  - Manipulates database contents

# Network Model—Basic Structure

- Resembles hierarchical model
- Collection of records in 1:M relationships
- Set
  - Relationship
  - Composed of at least two record types
    - Owner
      - Equivalent to the hierarchical model's parent
    - Member
      - Equivalent to the hierarchical model's child

# A Network Data Model

FIGURE 2.3 A NETWORK DATA MODEL



# The Network Data Model

- Advantages
  - Conceptual simplicity
  - Handles more relationship types
  - Data access flexibility
  - Promotes database integrity
  - Data independence
  - Conformance to standards

# The Network Data Model

## (continued)

- Disadvantages
  - System complexity
  - Lack of structural independence

# The Relational Model

- Developed by Codd (IBM) in 1970
- Considered ingenious but impractical in 1970
- Conceptually simple
- Computers lacked power to implement the relational model
- Today, microcomputers can run sophisticated relational database software

# The Relational Model—Basic Structure

- Relational Database Management System (RDBMS)
- Performs same basic functions provided by hierarchical and network DBMS systems, plus other functions
- Most important advantage of the RDBMS is its ability to let the user/designer operate in a human logical environment



# The Relational Model— Basic Structure (continued)

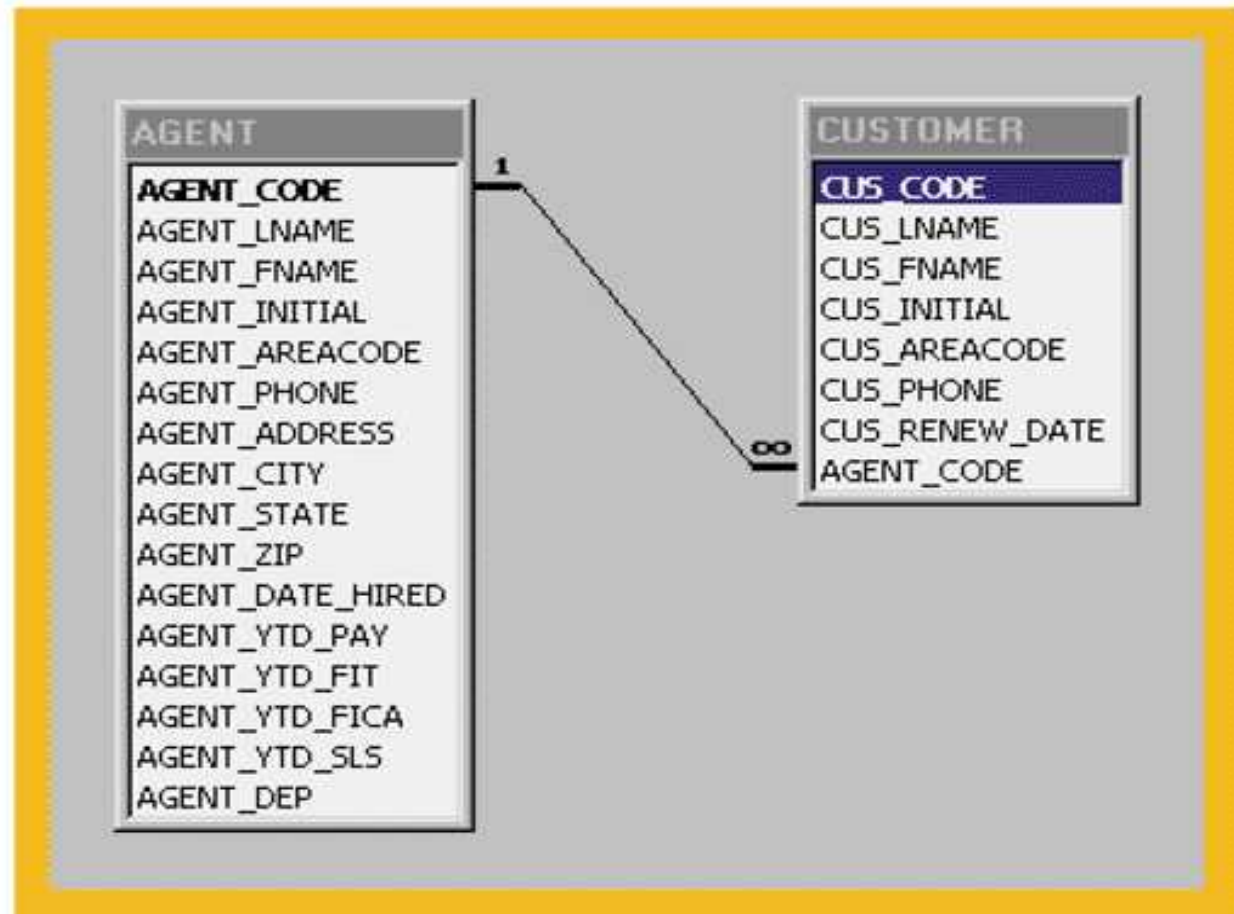
- Table (relations)
  - Matrix consisting of a series of row/column intersections
  - Related to each other by sharing a common entity characteristic
- Relational schema
  - Visual representation of relational database's entities, attributes within those entities, and relationships between those entities

# Relational Table

- Stores a collection of related entities
  - Resembles a file
- Relational table is purely logical structure
  - How data are physically stored in the database is of no concern to the user or the designer
  - This property became the source of a real database revolution

# A Relational Schema

FIGURE 2.5 A RELATIONAL SCHEMA



# Linking Relational Tables

FIGURE 2-6 LINKING RELATIONAL TABLES

Database name: Ch02\_Insurance Table name: AGENT (first six attributes)

AGENT_CODE	AGENT_NAME	AGENT_INITIAL	AGENT_INITIAL	AGENT_INITIAL	AGENT_INITIAL
500	Alex	Alex	B	7	3
502	Mike	Mike	F	6	5
503	John	John	T	6	5

Link through AGENT\_CODE

Table name: CUSTOMER

CUS_CODE	CUS_NAME	CUS_INITIAL	CUS_INITIAL	CUS_INITIAL	CUS_INITIAL	CUS_INITIAL	CUS_INITIAL	CUS_INITIAL
1000	Renee	Alex	A	010	004	2070	05-Aug-2004	500
1001	Dennis	Mike	K	713	804	1236	18-Jul-2004	501
1002	Smith	John	W	605	604	2765	29-Jan-2005	502
1003	Clarence	Paul	F	815	804	2189	14-Oct-2004	502
1004	Griffin	Myron	W	605	205	1077	27-Nov-2004	501
1005	Calvin	Alex	B	713	442	2281	22-Sep-2004	503
1006	Blower	James	G	010	287	1220	26-Mar-2004	500
1007	Wilson	George	W	815	200	2556	17-Jul-2004	503
1008	Peters	Anna	D	713	300	7105	07-Oct-2004	501
1009	Smith	Octo	K	815	207	2500	14-Nov-2004	503

# The Relational Model

- Advantages
  - Structural independence
  - Improved conceptual simplicity
  - Easier database design, implementation, management, and use
  - Ad hoc query capability
  - Powerful database management system

# The Relational Model (continued)

- Disadvantages
  - Substantial hardware and system software overhead
  - Can facilitate poor design and implementation
  - May promote “islands of information” problems

# The Entity Relationship Model

- Widely accepted and adapted graphical tool for data modeling
- Introduced by Chen in 1976
- Graphical representation of entities and their relationships in a database structure

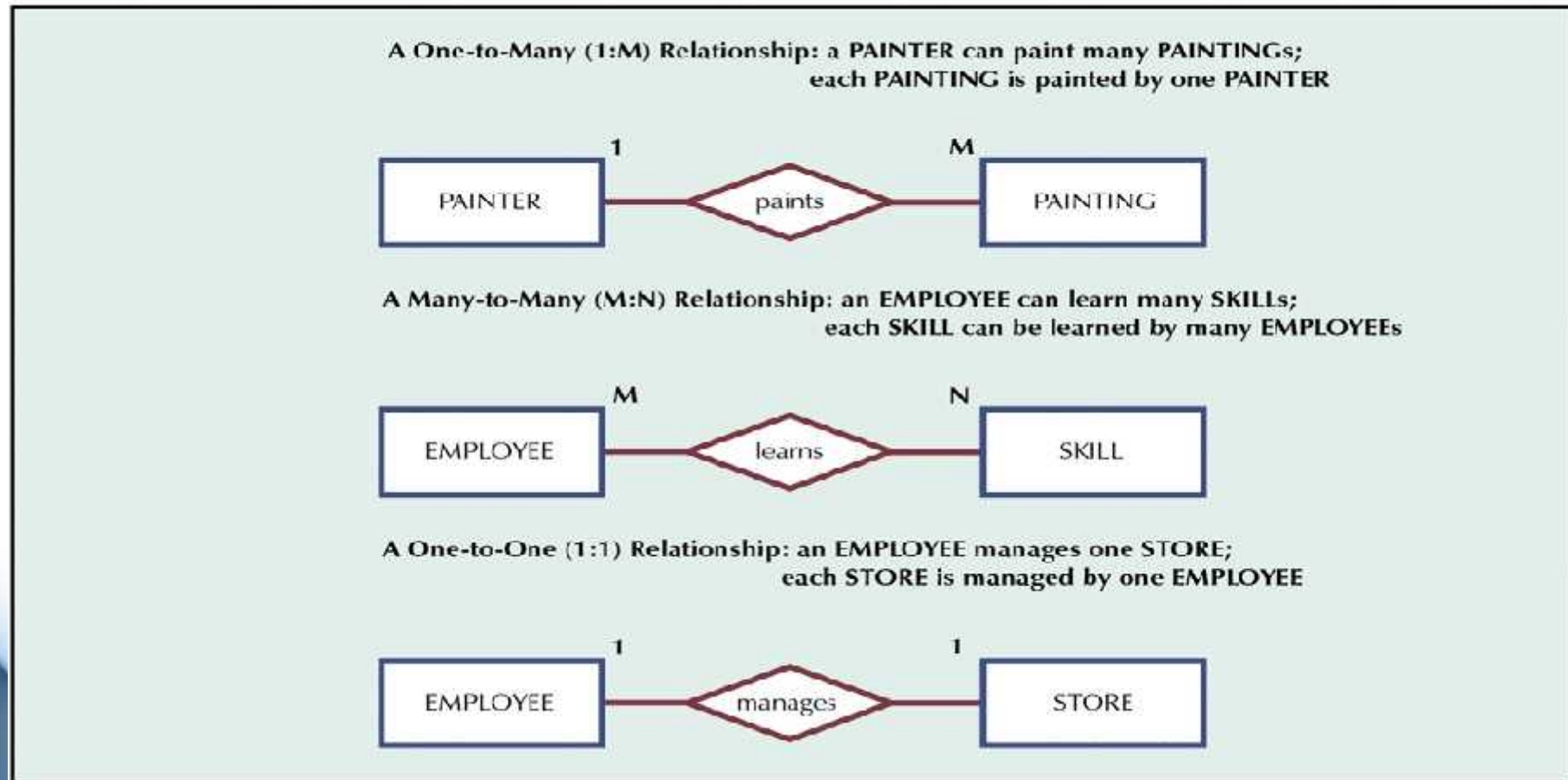


# The Entity Relationship Model— Basic Structure

- Entity relationship diagram (ERD)
  - Uses graphic representations to model database components
  - Entity is mapped to a relational table
- Entity instance (or occurrence) is row in table
- Entity set is collection of like entities
- Connectivity labels types of relationships
  - Diamond connected to related entities through a relationship line

# Relationships: The Basic Chen ERD

FIGURE 2.6 RELATIONSHIPS: THE BASIC CHEN ERD



# The Entity Relationship Model

- Advantages
  - Exceptional conceptual simplicity
  - Visual representation
  - Effective communication tool
  - Integrated with the relational data model

# The Entity Relationship Model

## (continued)

- Disadvantages
  - Limited constraint representation
  - Limited relationship representation
  - No data manipulation language
  - Loss of information content

# The Object Oriented Model

- Semantic data model (SDM) developed by Hammer and McLeod in 1981
- Modeled both data and their relationships in a single structure known as an object
- Basis of object oriented data model (OODM)
- OODM becomes the basis for the object oriented database management system (OODBMS)

# The Object Oriented Model

## (continued)

- Object is described by its factual content
  - Like relational model's entity
- Includes information about relationships between facts within object and relationships with other objects
  - Unlike relational model's entity
- Subsequent OODM development allowed an object to also contain operations
- Object becomes basic building block for autonomous structures

# Developments that Boosted OODM's Popularity

- Growing costs put a premium on code reusability
- Complex data types and system requirements became difficult to manage with a traditional RDBMS
- Became possible to support increasingly sophisticated transaction & information requirements
- Ever-increasing computing power made it possible to support the large computing overhead required

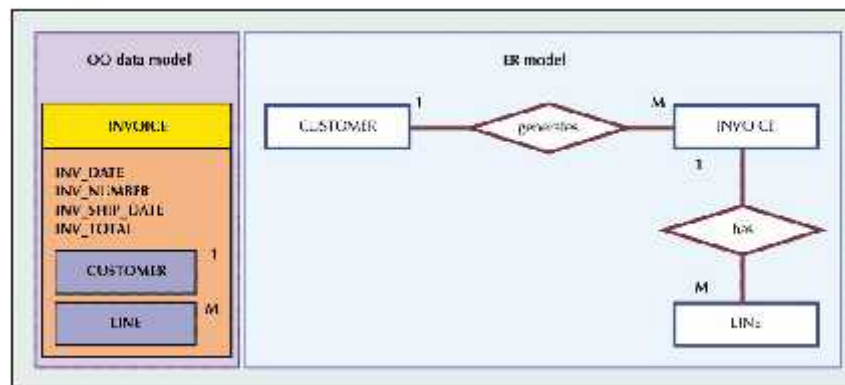


# Object Oriented Data Model— Basic Structure

- Object: abstraction of a real-world entity
- Attributes describe the properties of an object
- Objects that share similar characteristics are grouped in classes
- Classes are organized in a class hierarchy
- Inheritance is the ability of an object within the class hierarchy to inherit the attributes and methods of classes above it

# A Comparison of the OO Model and the ER Model

FIGURE 2.8 A COMPARISON OF THE OO MODEL AND THE ER MODEL



# The Object Oriented Model

- Advantages
  - Adds semantic content
  - Visual presentation includes semantic content
  - Database integrity
  - Both structural and data independence

# The Object Oriented Model

## (continued)

- Disadvantages
  - Slow pace of OODM standards development
  - Complex navigational data access
  - Steep learning curve
  - High system overhead slows transactions
  - Lack of market penetration

# Other Models

- Extended Relational Data Model (ERDM)
  - Semantic data model developed in response to increasing complexity of applications
  - DBMS based on the ERDM often described as an object/relational database management system (O/RDBMS)
  - Primarily geared to business applications

# Other Models (continued)

- Date's objections to ERDM label
  - Given proper support for domains, relational data models are quite capable of handling complex data
    - Therefore, capability that is supposedly being extended is already there
  - O/RDM label is not accurate because the relational data model's domain is not an object model structure

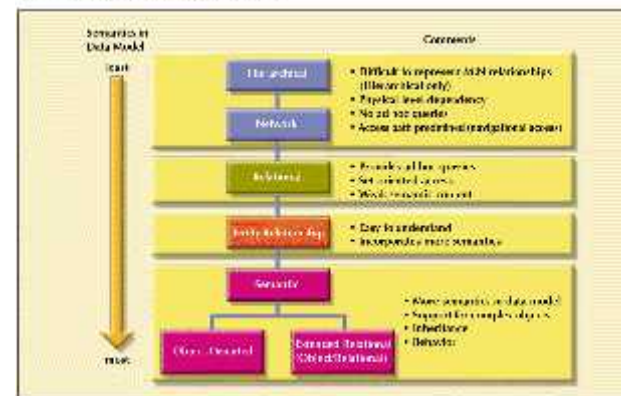
# Data Models: A Summary

- Each new data model capitalized on the shortcomings of previous models
- Common characteristics:
  - Conceptual simplicity without compromising the semantic completeness of the database
  - Represent the real world as closely as possible
  - Representation of real-world transformations (behavior) must be in compliance with consistency and integrity characteristics of any data model



# The Development of Data Models

FIGURE 2.9 The Development of Data Models



# Entity Relation Model

# The Entity Relationship Model

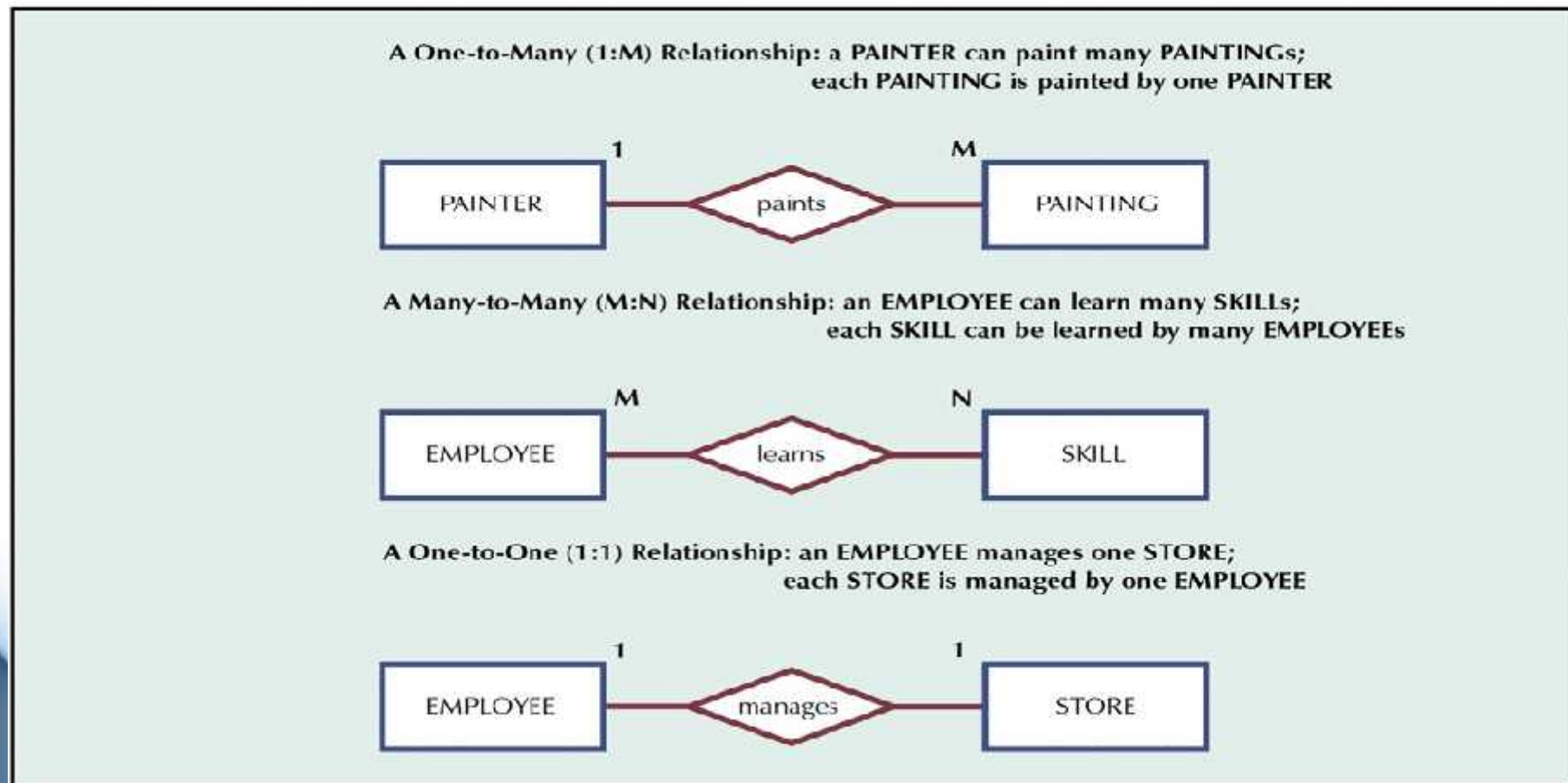
- Widely accepted and adapted graphical tool for data modeling
- Introduced by Chen in 1976
- Graphical representation of entities and their relationships in a database structure

# The Entity Relationship Model — Basic Structure

- ☐ Entity relationship diagram (ERD)
  - ☐ Uses graphic representations to model database components
  - ☐ Entity is mapped to a relational table
- ☐ Entity instance (or occurrence) is row in table
- ☐ Entity set is collection of like entities
- ☐ Connectivity labels types of relationships
  - ☐ Diamond connected to related entities through a relationship line

# Relationships: The Basic Chen ERD

FIGURE 2.6 RELATIONSHIPS: THE BASIC CHEN ERD



# Relationships: The Basic Crow's Foot ERD

FIGURE 2.7 RELATIONSHIPS: THE BASIC CROW'S FOOT ERD

**A One-to-Many (1:M) Relationship:** a PAINTER can paint many PAINTINGs;  
each PAINTING is painted by one PAINTER



**A Many-to-Many (M:N) Relationship:** an EMPLOYEE can learn many SKILLs;  
each SKILL can be learned by many EMPLOYEEs



**A One-to-One (1:1) Relationship:** an EMPLOYEE manages one STORE;  
each STORE is managed by one EMPLOYEE



# The Entity Relationship (ER) Model

- ER model forms the basis of an ER diagram
- ERD represents the conceptual database as viewed by end user
- ERDs depict the ER model's three main components:
  - ▣ Entities
  - ▣ Attributes
  - ▣ Relationships

# Entities

- ☐ Refers to the *entity set* and not to a single entity occurrence
- ☐ Corresponds to a table and not to a row in the relational environment
- ☐ In both the Chen and Crow's Foot models, an entity is represented by a rectangle containing the entity's name
- ☐ Entity name, a noun, is usually written in capital letters

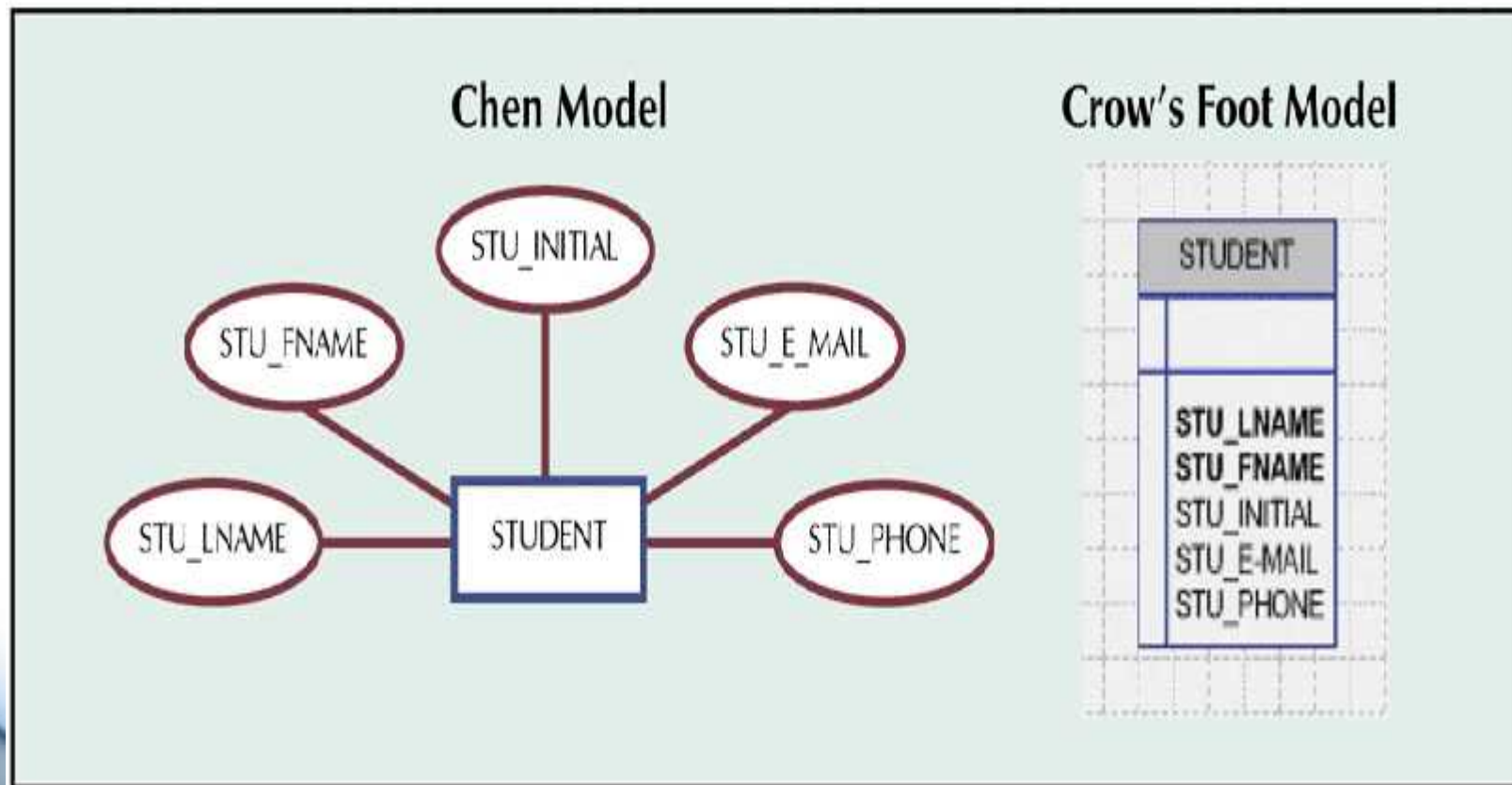


# Attributes

- ☐ Characteristics of entities
- ☐ In Chen model, attributes are represented by ovals and are connected to the entity rectangle with a line
- ☐ Each oval contains the name of the attribute it represents
- ☐ In the Crow's Foot model, the attributes are simply written in the attribute box below the entity rectangle

# The Attributes of the STUDENT Entity

FIGURE 4.1 THE ATTRIBUTES OF THE STUDENT ENTITY



# Attributes

- An entity is represented by a set of attributes, that is descriptive properties possessed by all members of an entity set.

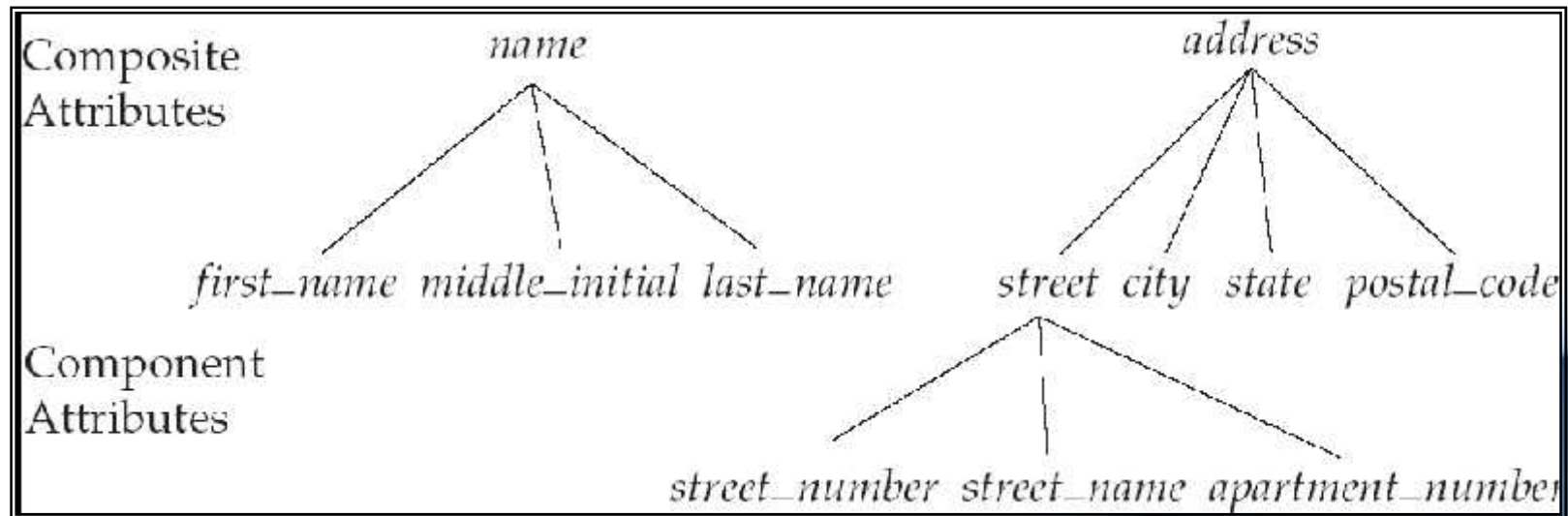
Example:

*customer = (customer\_id, customer\_name,  
customer\_street, customer\_city)*

*loan = (loan\_number, amount)*

- **Domain** – the set of permitted values for each attribute
- Attribute types:
  - *Simple* and *composite* attributes.
  - *Single-valued* and *multi-valued* attributes
    - Example: multivalued attribute: *phone\_numbers*
  - *Derived* attributes
    - Can be computed from other attributes
    - Example: age, given date\_of\_birth

# Composite Attributes



# Relationship Sets

- A **relationship** is an association among several entities

Example:

Hayes                  depositor                  A-102  
*customer* entity   *relationship set*   *account* entity

- A **relationship set** is a mathematical relation among  $n \geq 2$  entities, each taken from entity sets

$$\{(e_1, e_2, \dots, e_n) \mid e_1 \in E_1, e_2 \in E_2, \dots, e_n \in E_n\}$$

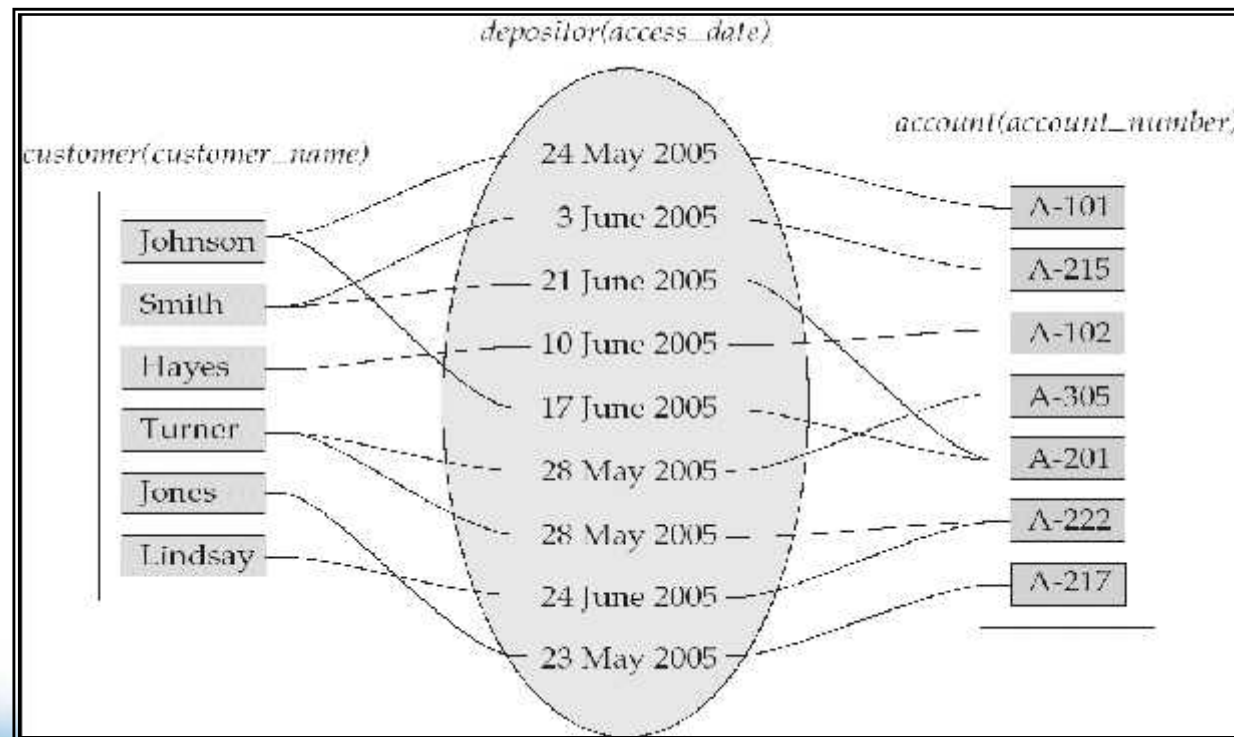
where  $(e_1, e_2, \dots, e_n)$  is a relationship

– Example:

$(\text{Hayes}, \text{A-102}) \in \text{depositor}$

# Relationship Sets

- An **attribute** can also be property of a relationship set.
- For instance, the *depositor* relationship set between entity sets *customer* and *account* may have the attribute *access-date*



# Degree of a Relationship Set

- Refers to number of entity sets that participate in a relationship set.
- Relationship sets that involve two entity sets are **binary** (or degree two). Generally, most relationship sets in a database system are binary.
- **Relationship sets may involve more than two entity sets.**
  - ▶ Example: Suppose employees of a bank may have jobs (responsibilities) at multiple branches, with different jobs at different branches. Then there is a ternary relationship set between entity sets *employee*, *job*, and *branch*
- Relationships between more than two entity sets are rare. Most relationships are binary. (More on this later.)

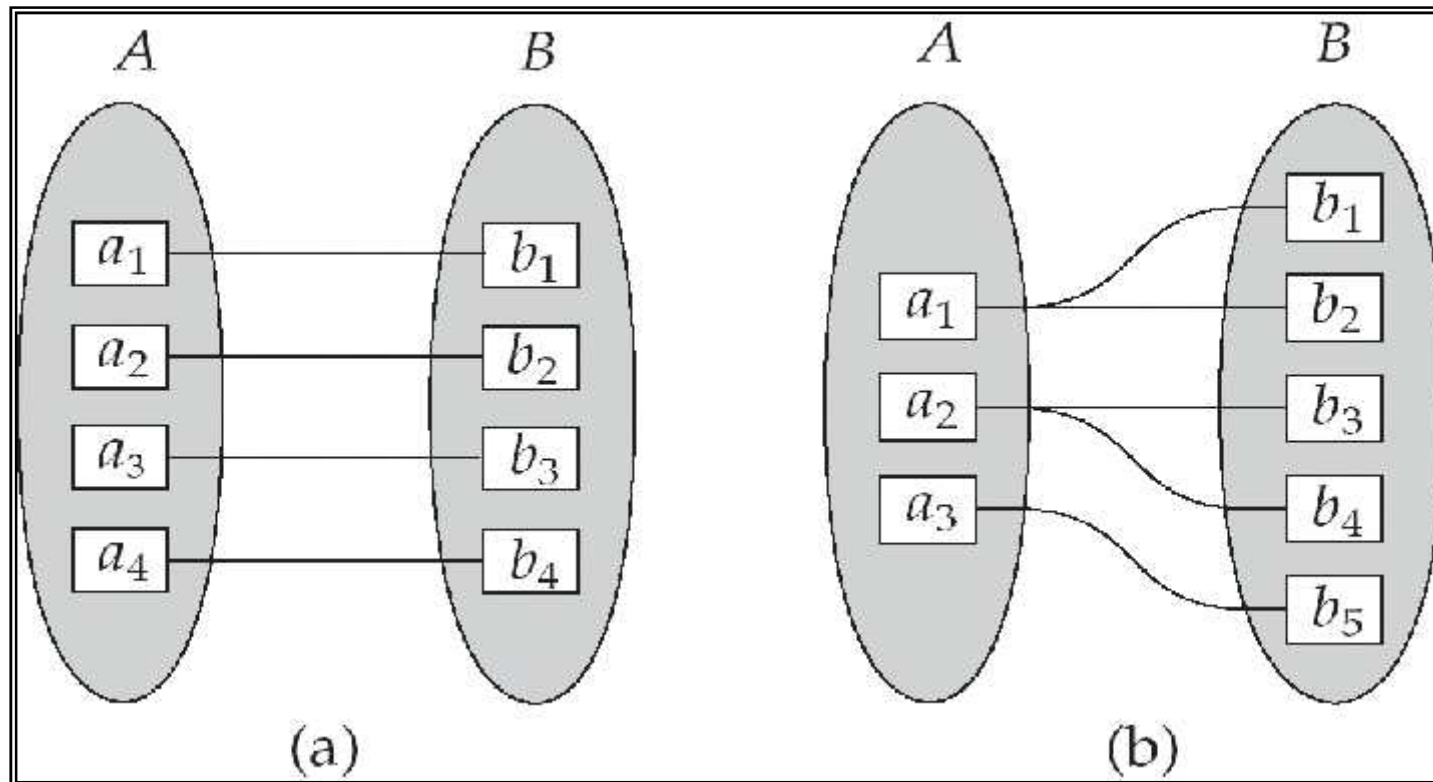


# Mapping Cardinality Constraints

- Express the number of entities to which another entity can be associated via a relationship set.
- Most useful in describing binary relationship sets.
- For a binary relationship set the mapping cardinality must be one of the following types:
  - One to one
  - One to many
  - Many to one
  - Many to many



# Mapping Cardinalities

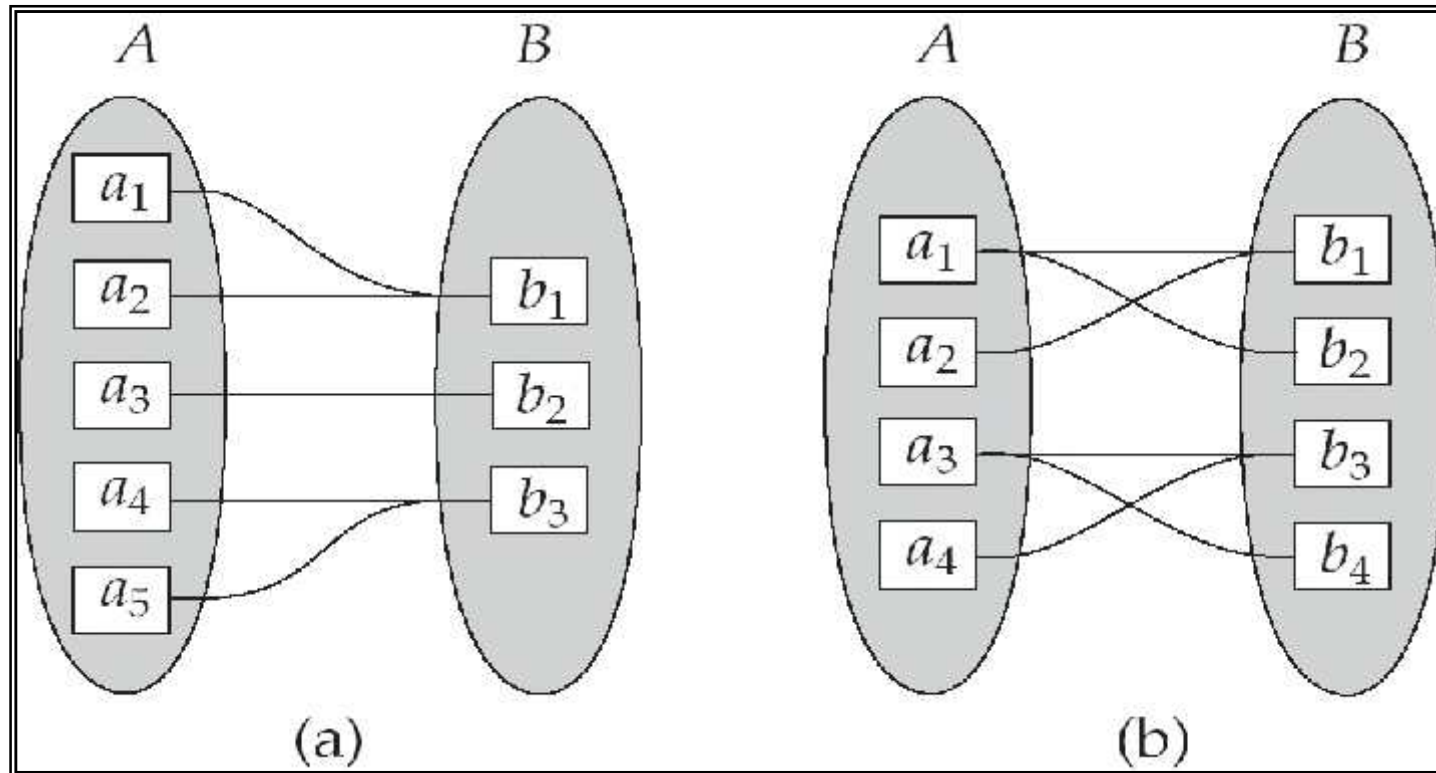


One to one

One to many

Note: Some elements in  $A$  and  $B$  may not be mapped to any elements in the other set

# Mapping Cardinalities



Many to one

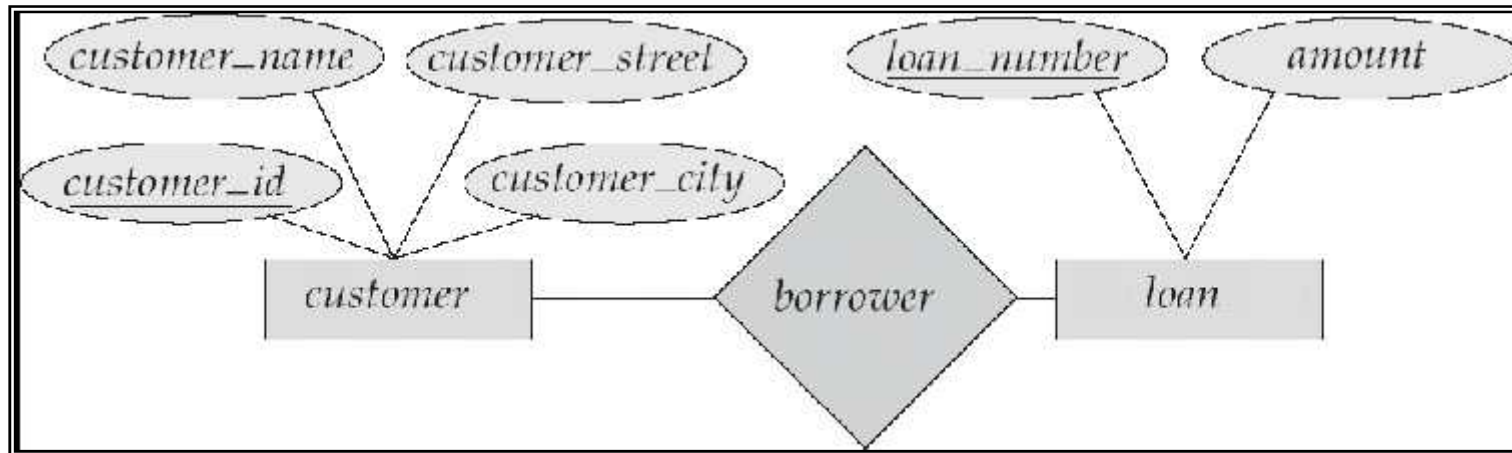
Many to many

Note: Some elements in  $A$  and  $B$  may not be mapped to any elements in the other set

# Primary Keys

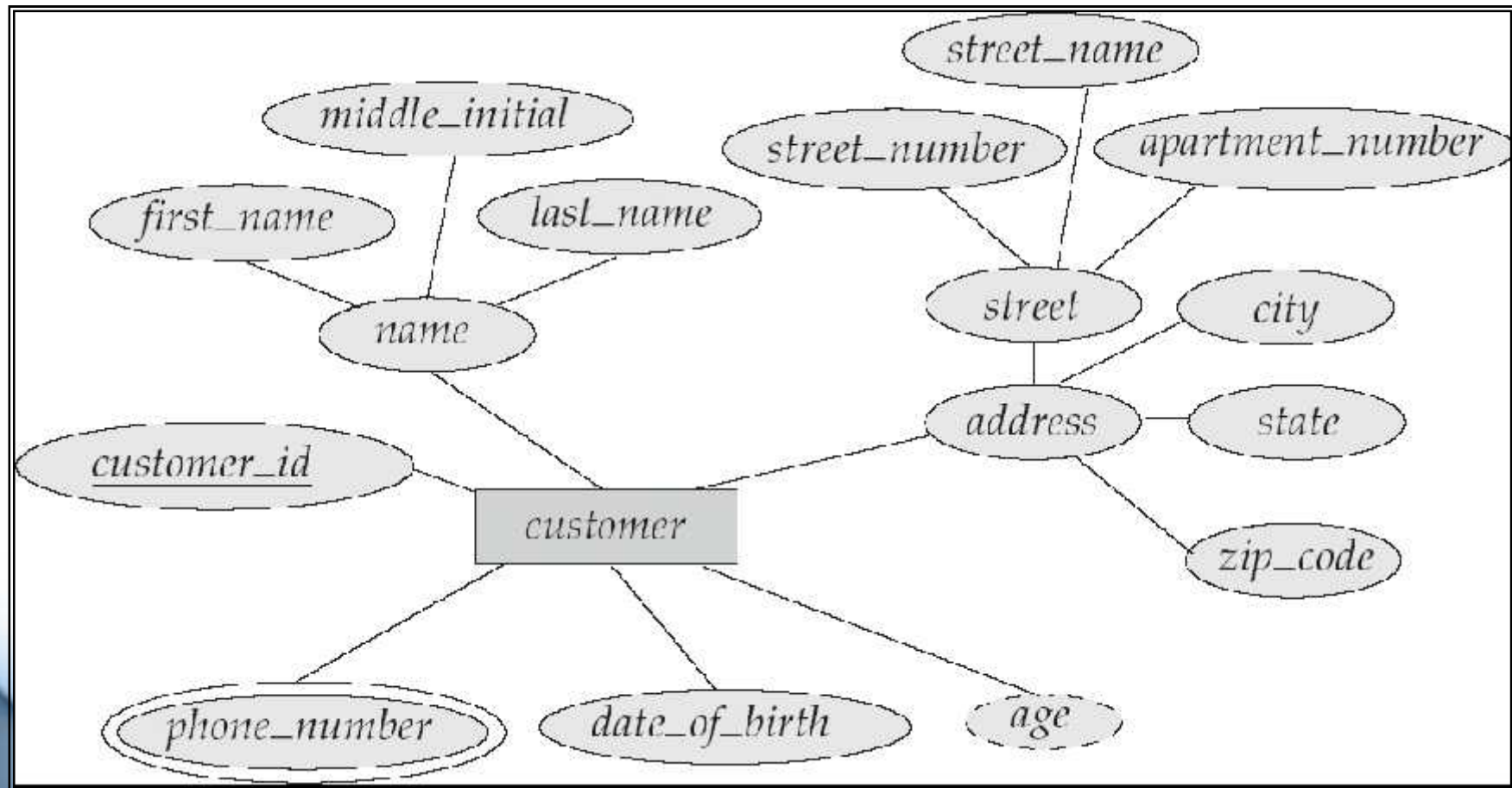
- Underlined in the ER diagram
- Key attributes are also underlined in frequently used table structure shorthand
- Ideally composed of only a single attribute
- Possible to use a *composite key*:
  - Primary key composed of more than one attribute

# E-R Diagrams

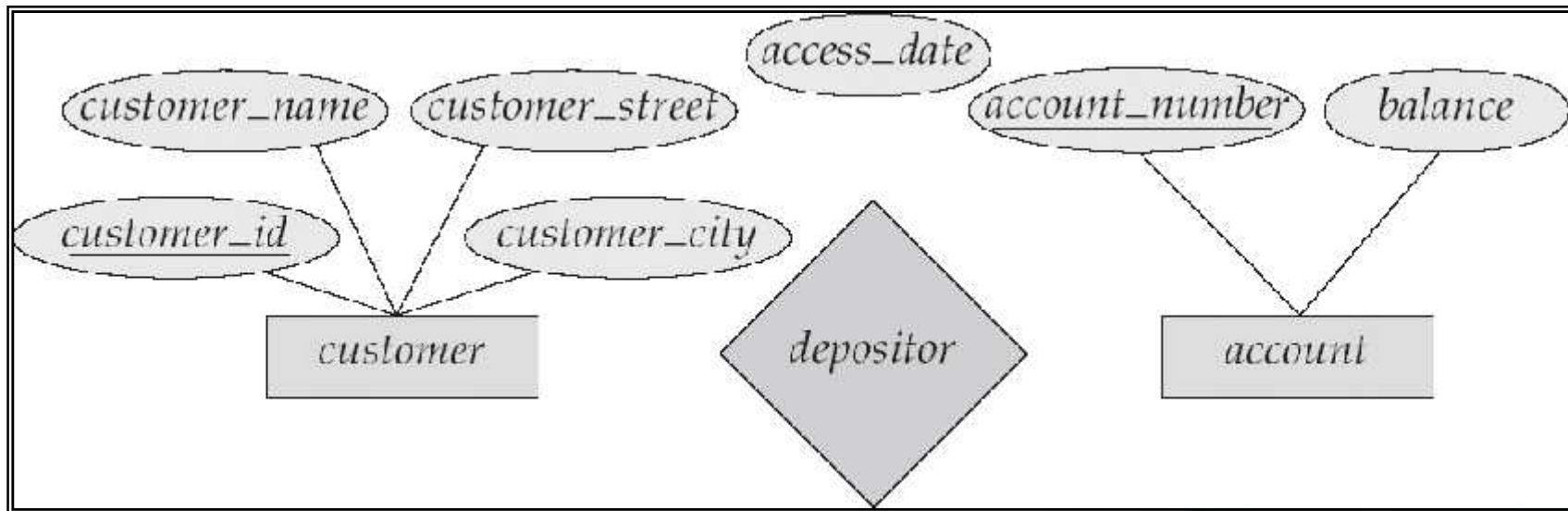


- Rectangles represent entity sets.
- Diamonds represent relationship sets.
- Lines link attributes to entity sets and entity sets to relationship sets.
- Ellipses represent attributes
  - Double ellipses represent multivalued attributes.
  - Dashed ellipses denote derived attributes.
- Underline indicates primary key attributes

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



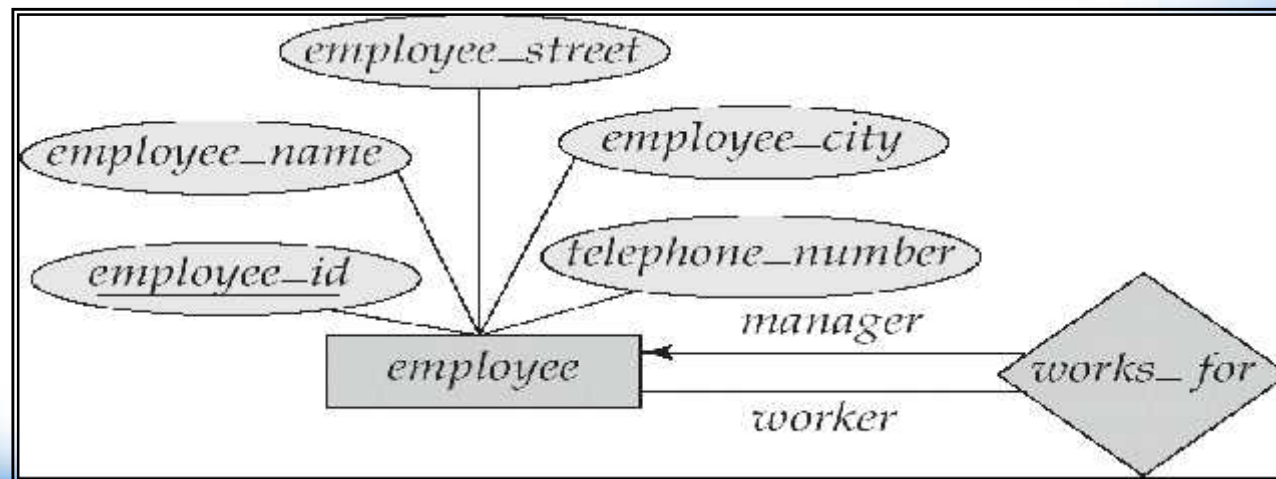
# Relationship Sets with Attributes





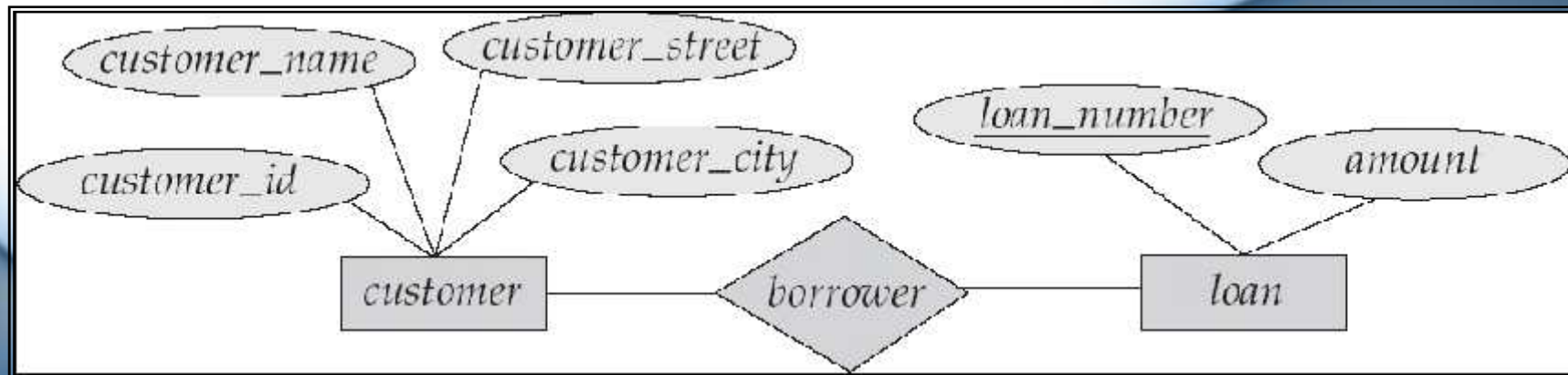
# Roles

- Entity sets of a relationship need not be distinct
- The labels “manager” and “worker” are called **roles**; they specify how employee entities interact via the works\_for relationship set.
- Roles are indicated in E-R diagrams by labeling the lines that connect diamonds to rectangles.
- Role labels are optional, and are used to clarify semantics of the relationship



# Participation of an Entity Set in a Relationship Set

- Total participation (indicated by double line): every entity in the entity set participates in at least one relationship in the relationship set
  - 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 set
  - Example: participation of customer in borrower is partial





# Multiplicity constraints on relationships

- Represents the number of occurrences of one entity that may relate to a single occurrence of an associated entity.
- Represents policies (called business rules) established by user or company.

# Multiplicity constraints

- The most common degree for relationships is binary.
- Binary relationships are generally referred to as being:
  - one-to-one (1:1)
  - one-to-many (1:\*)
  - many-to-many (\*:\*)

# Summary of multiplicity constraints

**Table 7.1** A summary of ways to represent multiplicity constraints.

Alternative ways to represent multiplicity constraints	Meaning
0..1	Zero or one entity occurrence
1..1 (or just 1)	Exactly one entity occurrence
0..* (or just *)	Zero or many entity occurrences
1..*	One or many entity occurrences
5..10	Minimum of 5 up to a maximum of 10 entity occurrences
0, 3, 6–8	Zero or three or six, seven, or eight entity occurrences

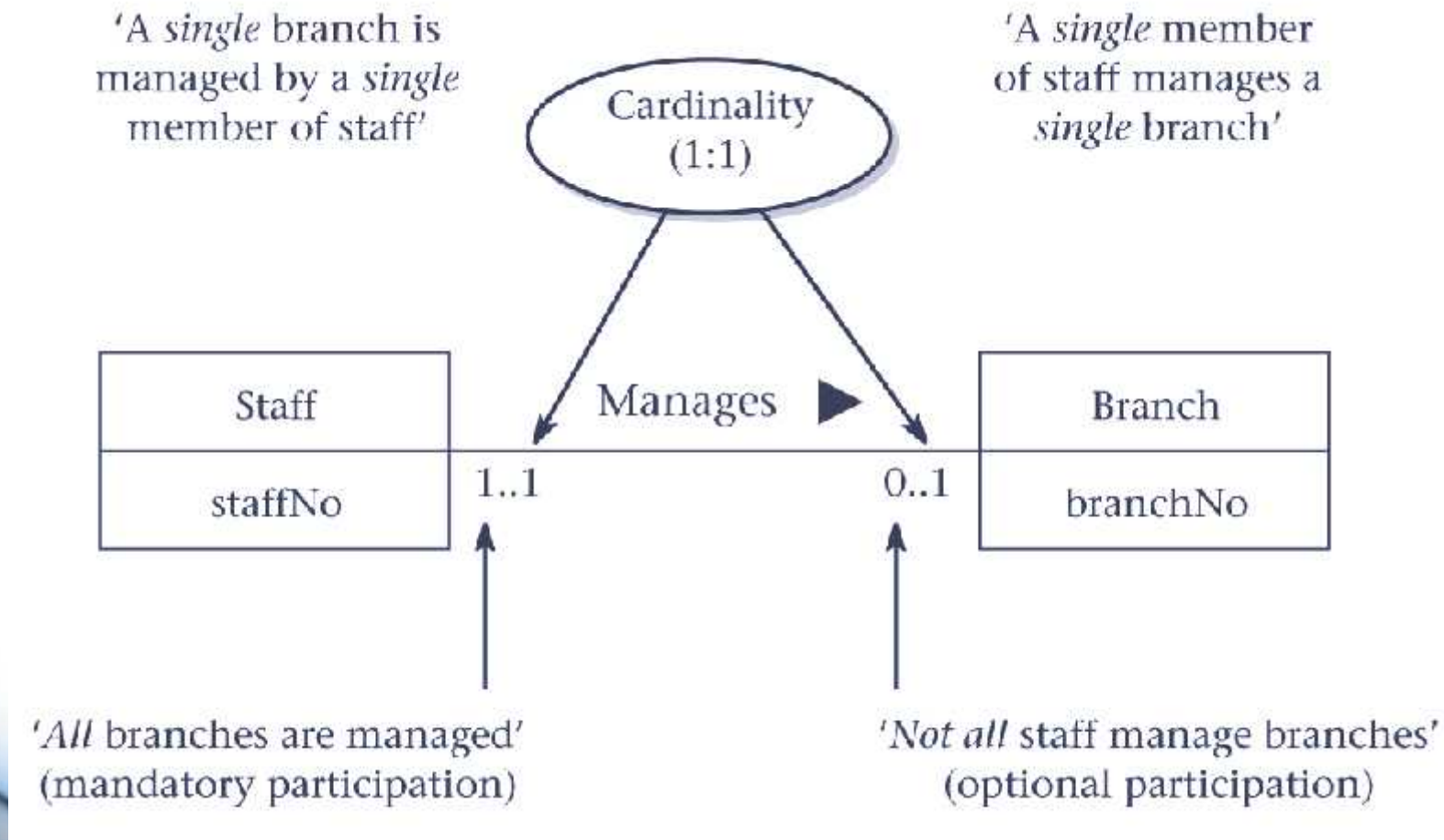
# Multiplicity

- Made up of two types of restrictions on relationships:
  - cardinality
  - and participation

# Multiplicity

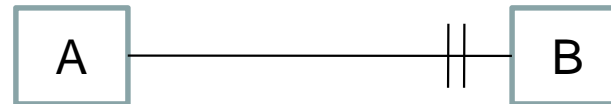
- Cardinality
  - Describes the number of possible relationships for each participating entity.
- Participation
  - Determines whether all or only some entity occurrences participate in a relationship.

# Multiplicity as cardinality and participation constraints

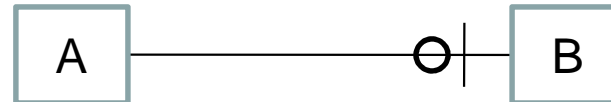


# Relationship Cardinalities

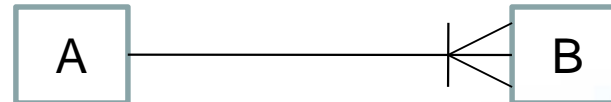
**A is related to one and only one B**



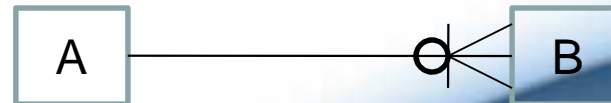
**A is related to zero or one B**



**A is related to one or more B**



**A is related to zero, one or more B**



**A is related to more than one B**



# Design Issues

- **Use of entity sets vs. attributes**

Choice mainly depends on the structure of the enterprise being modeled, and on the semantics associated with the attribute in question.

- **Use of entity sets vs. relationship sets**

Possible guideline is to designate a relationship set to describe an action that occurs between entities

- **Binary versus  $n$ -ary relationship sets**

Although it is possible to replace any nonbinary ( $n$ -ary, for  $n > 2$ ) relationship set by a number of distinct binary relationship sets, a  $n$ -ary relationship set shows more clearly that several entities participate in a single relationship.

- **Placement of relationship attributes**

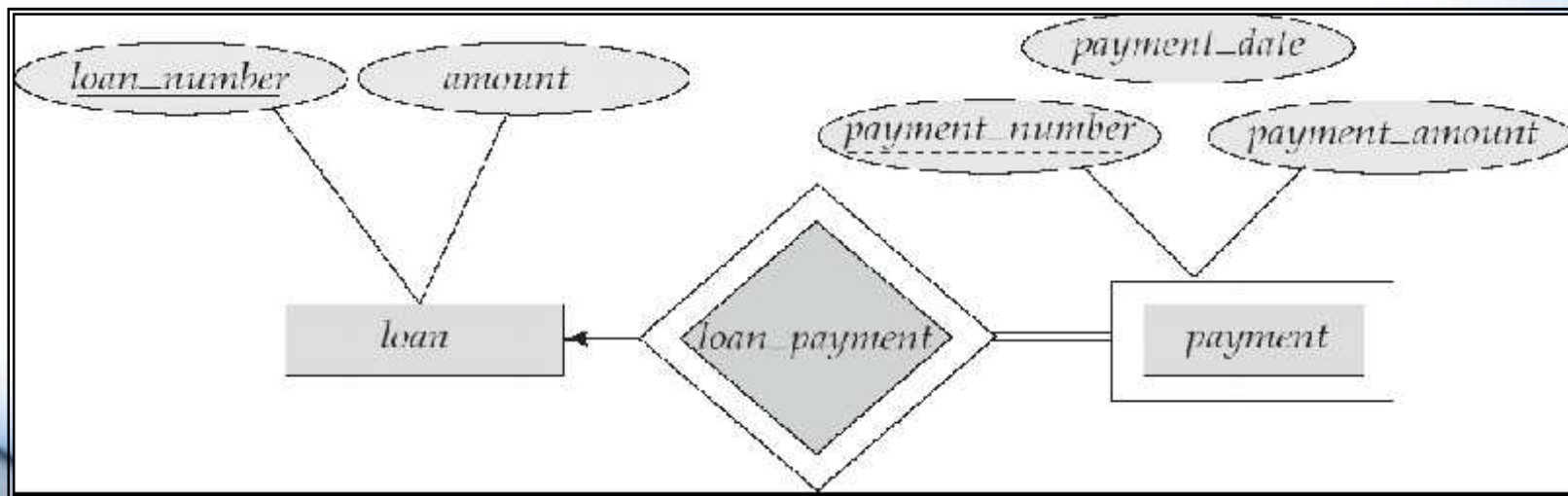


# Weak Entity Sets

- An entity set that does not have a primary key is referred to as a **weak entity set**.
- The existence of a weak entity set depends on the existence of a **identifying entity set**
  - it must relate to the identifying entity set via a total, one-to-many relationship set from the identifying to the weak entity set
  - Identifying relationship depicted using a double diamond
- The **discriminator** (*or partial key*) of a weak entity set is the set of attributes that distinguishes among all the entities of a weak entity set.
- The primary key of a weak entity set is formed by the primary key of the strong entity set on which the weak entity set is existence dependent, plus the weak entity set's discriminator.

# Weak Entity Sets (Cont.)

- We depict a weak entity set by double rectangles.
- We underline the discriminator of a weak entity set with a dashed line.
- *payment\_number* – discriminator of the *payment* entity set
- Primary key for *payment* – (*loan\_number*, *payment\_number*)



# Weak Entity Sets (Cont.)

- Note: the primary key of the strong entity set is not explicitly stored with the weak entity set, since it is implicit in the identifying relationship.
- If *loan\_number* were explicitly stored, *payment* could be made a strong entity, but then the relationship between *payment* and *loan* would be duplicated by an implicit relationship defined by the attribute *loan\_number* common to *payment* and *loan*

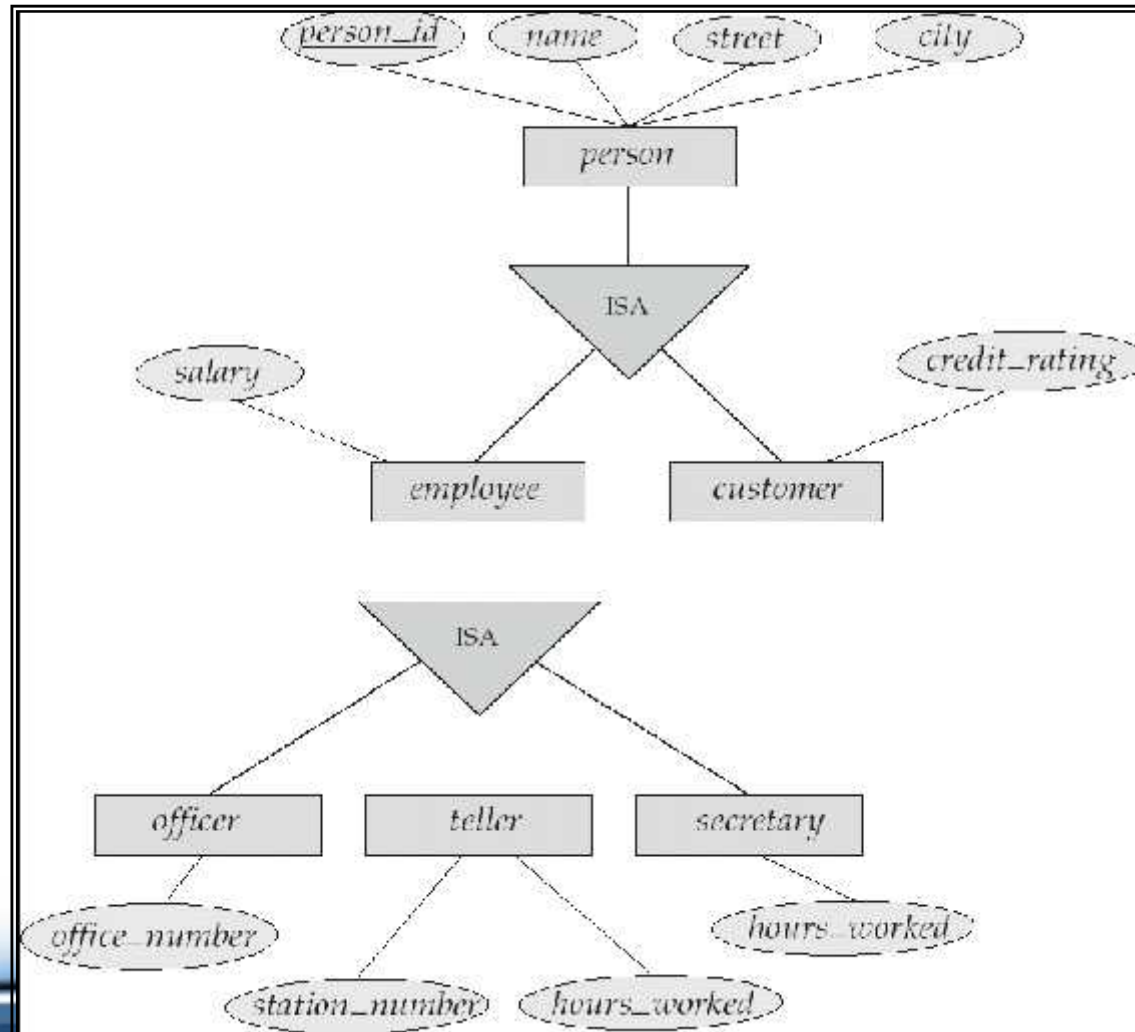
# More Weak Entity Set Examples

- In a university, a *course* is a strong entity and a *course\_offering* can be modeled as a weak entity
- The discriminator of *course\_offering* would be *semester* (including year) and *section\_number* (if there is more than one section)
- If we model *course\_offering* as a strong entity we would model *course\_number* as an attribute.  
Then the relationship with *course* would be implicit in the *course\_number* attribute

# Extended E-R Features: Specialization

- Top-down design process; we designate subgroupings within an entity set that are distinctive from other entities in the set.
- These subgroupings become lower-level entity sets that have attributes or participate in relationships that do not apply to the higher-level entity set.
- Depicted by a *triangle* component labeled ISA (E.g. *customer* “is a” *person*).
- **Attribute inheritance** – a lower-level entity set inherits all the attributes and relationship participation of the higher-level entity set to which it is linked.

# Specialization Example



# Extended ER Features: Generalization

- **A bottom-up design process** – combine a number of entity sets that share the same features into a higher-level entity set.
- Specialization and generalization are simple inversions of each other; they are represented in an E-R diagram in the same way.
- The terms specialization and generalization are used interchangeably.



# Specialization and Generalization

- Can have multiple specializations of an entity set based on different features.
- E.g. *permanent\_employee* vs. *temporary\_employee*, in addition to *officer* vs. *secretary* vs. *teller*
- Each particular employee would be
  - a member of one of *permanent\_employee* or *temporary\_employee*,
  - and also a member of one of *officer*, *secretary*, or *teller*
- The ISA relationship also referred to as **superclass - subclass** relationship

# Design Constraints on a Specialization/Generalization

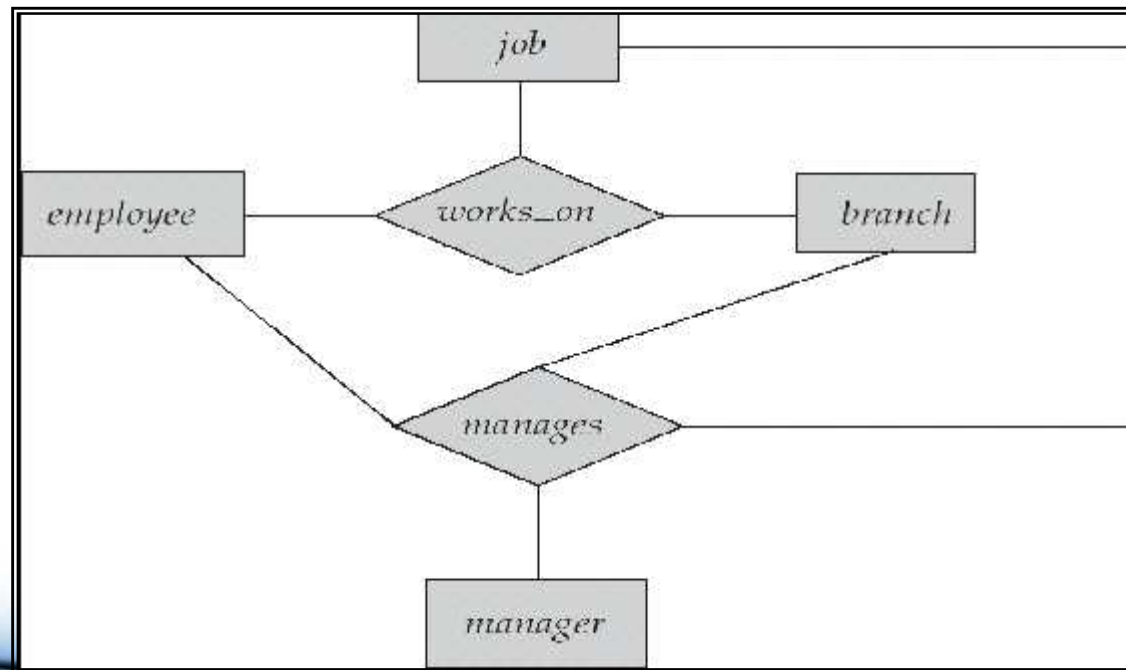
- Constraint on which entities can be members of a given lower-level entity set.
  - condition-defined
    - Example: all customers over 65 years are members of *senior-citizen* entity set; *senior-citizen* ISA *person*.
  - user-defined
- Constraint on whether or not entities may belong to more than one lower-level entity set within a single generalization.
  - **Disjoint**
    - an entity can belong to only one lower-level entity set
    - Noted in E-R diagram by writing *disjoint* next to the ISA triangle
  - **Overlapping**
    - an entity can belong to more than one lower-level entity set

# 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
  - **partial**: an entity need not belong to one of the lower-level entity sets

# Aggregation

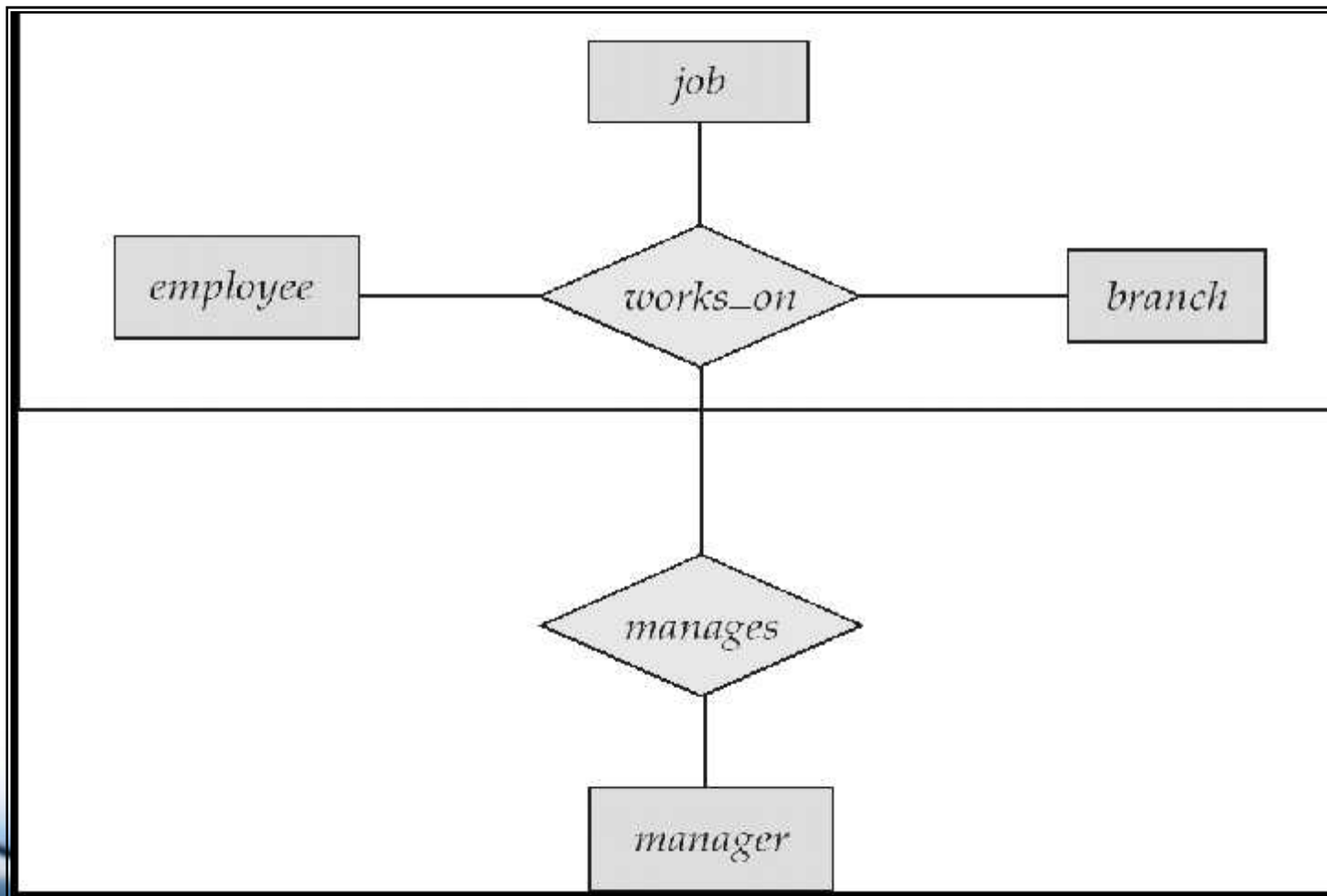
- Consider the ternary relationship *works\_on*, which we saw earlier
- Suppose we want to record managers for tasks performed by an employee at a branch



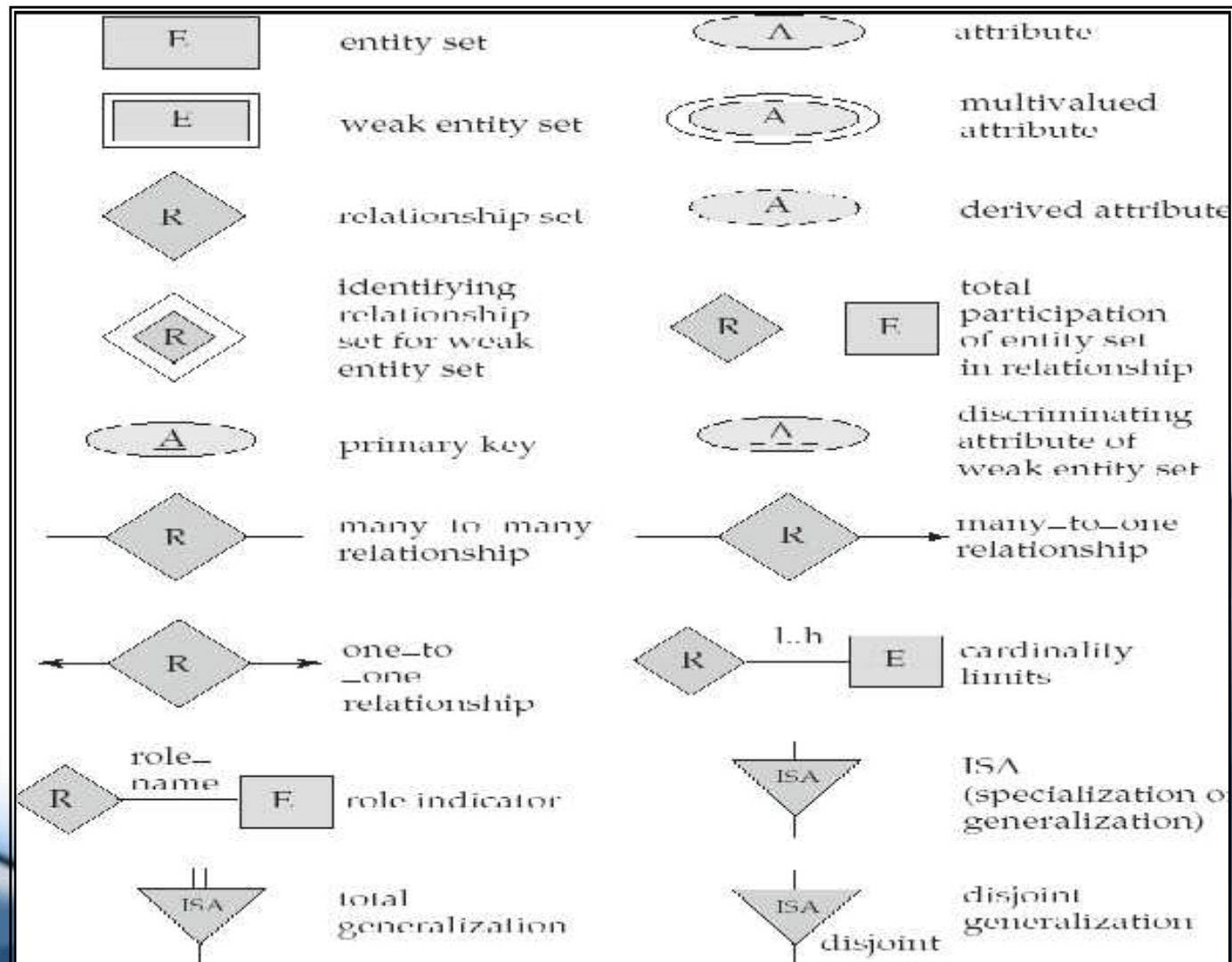
# Aggregation (Cont.)

- Relationship sets *works\_on* and *manages* represent overlapping information
  - Every *manages* relationship corresponds to a *works\_on* relationship
  - However, some *works\_on* relationships may not correspond to any *manages* relationships
    - So we can't discard the *works\_on* relationship
- Eliminate this redundancy via *aggregation*
  - Treat relationship as an abstract entity
  - Allows relationships between relationships
  - Abstraction of relationship into new entity
- Without introducing redundancy, the following diagram represents:
  - An employee works on a particular job at a particular branch
  - An employee, branch, job combination may have an associated manager

# E-R Diagram With Aggregation



# Summary of Symbols Used in E-R Notation





# RELATIONAL MODEL

# A Logical View of Data

- Relational model
  - Enables us to view data *logically* rather than *physically*
  - Reminds us of simpler file concept of data storage
- Table
  - Has advantages of structural and data independence
  - Resembles a file from conceptual point of view
  - Easier to understand than its hierarchical and network database predecessors

# Tables and Their Characteristics

- **Table:** two-dimensional structure composed of rows and columns
- Contains group of related entities → an entity set
  - Terms *entity set* and *table* are often used interchangeably
- Table also called a *relation* because the relational model's creator, Codd, used the term *relation* as a synonym for table

# Example of a Relation

<i>account_number</i>	<i>branch_name</i>	<i>balance</i>
A-101	Downtown	500
A-102	Perryridge	400
A-201	Brighton	900
A-215	Mianus	700
A-217	Brighton	750
A-222	Redwood	700
A-305	Round Hill	350

# Basic Structure

Formally, given sets  $D_1, D_2, \dots, D_n$  a **relation**  $r$  is a subset of

$$D_1 \times D_2 \times \dots \times D_n$$

Thus, a relation is a set of  $n$ -tuples  $(a_1, a_2, \dots, a_n)$  where each  $a_i \in D_i$

Example: If

- $customer\_name = \{\text{Jones, Smith, Curry, Lindsay, ...}\}$   
/\* Set of all customer names \*/
- $customer\_street = \{\text{Main, North, Park, ...}\}$  /\* set of all street names \*/
- $customer\_city = \{\text{Harrison, Rye, Pittsfield, ...}\}$  /\* set of all city names \*/

Then  $r = \{$   
    (Jones, Main, Harrison),  
    (Smith, North, Rye),  
    (Curry, North, Rye),  
    (Lindsay, Park, Pittsfield)  $\}$

is a relation over

$customer\_name \times customer\_street \times customer\_city$

# Attribute Types

Each attribute of a relation has a name

The set of allowed values for each attribute is called the **domain** of the attribute

Attribute values are (normally) required to be **atomic**; that is, indivisible

- E.g. the value of an attribute can be an account number, but cannot be a set of account numbers

Domain is said to be atomic if all its members are atomic

The special value *null* is a member of every domain

The null value causes complications in the definition of many operations

- We shall ignore the effect of null values in our main presentation and consider their effect later



# Characteristics of a Relational Table

**TABLE 3.1** CHARACTERISTICS OF A RELATIONAL TABLE

- 1 A table is perceived as a two-dimensional structure composed of rows and columns.
- 2 Each table row (**tuple**) represents a single entity occurrence within the entity set.
- 3 Each table column represents an attribute, and each column has a distinct name.
- 4 Each row/column intersection represents a single data value.

**TABLE 3.1** CHARACTERISTICS OF A RELATIONAL TABLE (CONTINUED)

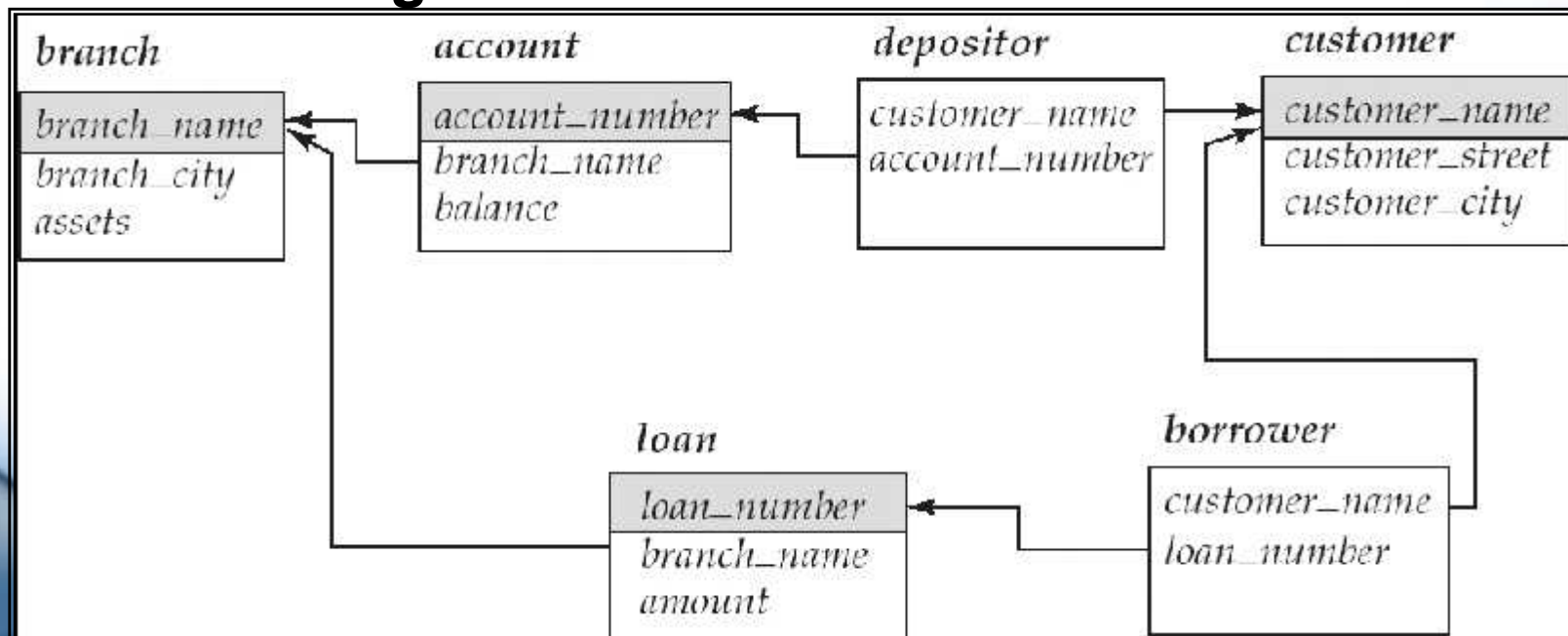
- 5 All values in a column must conform to the same data format. For example, if the attribute is assigned an integer data format, all values in the column representing that attribute must be integers.
- 6 Each column has a specific range of values known as the **attribute domain**.
- 7 The order of the rows and columns is immaterial to the DBMS.
- 8 Each table must have an attribute or a combination of attributes that uniquely identifies each row.

Table 3.1



# Relational Schema

- A Relational schema is a textual representation of the database tables where each table is described by its name followed by the list of its attributes in the parenthesis. The primary key attribute(s) are underlined.
- **Schema diagram**



# Relation Schema

$A_1, A_2, \dots, A_n$  are *attributes*

$R = (A_1, A_2, \dots, A_n)$  is a *relation schema*

Example:

*Customer\_schema* = (*customer\_name*,  
*customer\_street*, *customer\_city*)

$r(R)$  denotes a *relation*  $r$  on the *relation schema*  $R$

Example:

*customer* (*Customer\_schema*)

# Relation Instance

- The current values (*relation instance*) of a relation are specified by a table
- An element  $t$  of  $r$  is a *tuple*, represented by a *row* in a table

The diagram shows a table representing a relation instance. The table has three columns and four rows. Arrows point from the text 'attributes (or columns)' to the column headers. Another set of arrows points from the text 'tuples (or rows)' to the four rows of the table.

<i>customer_name</i>	<i>customer_street</i>	<i>customer_city</i>
Jones	Main	Harrison
Smith	North	Rye
Curry	North	Rye
Lindsay	Park	Pittsfield

*customer*

# Database

- A database consists of multiple relations
- Information about an enterprise is broken up into parts, with each relation storing one part of the information

*account* : stores information about accounts

*depositor* : stores information about which customer  
owns which account

*customer* : stores information about customers

- Storing all information as a single relation such as  
*bank(account\_number, balance, customer\_name, ..)*  
results in
  - repetition of information
    - e.g. if customers has several accounts
  - the need for null values
    - e.g., to represent a customer without an account
- Normalization theory deals with how to design relational schemas

# Keys

- Consists of one or more attributes that determine other attributes
- **Primary key (PK)** is an attribute (or a combination of attributes) that uniquely identifies any given entity (row)
- Key's role is based on determination
  - If you know the value of attribute A, you can look up (determine) the value of attribute B
  - The shorthand notation is  $A \rightarrow B$
  - If A determines B, C, and D then  $A \rightarrow B, C, D$ .

# Keys

- A multi-attribute key is known as **Composite key**.
- Any attribute that is part of a key is known as a key is known as a **key attribute**.
- A **Superkey** is a set of one or more attributes that, taken collectively, allow us to identify uniquely a tuple in the relation.
- Ex. **Customer\_id** attribute of the relation **Customer**.
- **Customer\_id, Cust\_name** attributes of the relation Customer.
- **Customer\_id** with or without additional attributes , can be a superkey even when the additional attributes are redundant.

# Keys

- A **Candidate Key** can be described as a superkey without redundancies, i.e. minimal superkey
- Ex. Customer\_id, Cust\_name is a superkey, but it is not a candidate key.
- Primary key is the candidate key chosen to be the unique row identifier.
- Primary key is a superkey as well as a candidate key.



# Keys

- Foreign key (FK)
  - An attribute whose values match primary key values in the related table
- Referential integrity
  - FK contains a value that refers to an existing valid tuple (row) in another relation
- Secondary key
  - Key used strictly for data retrieval purposes

# Relational Database Keys

**TABLE 3.3** RELATIONAL DATABASE KEYS

KEY TYPE	DEFINITION
<b>Superkey</b>	An attribute (or combination of attributes) that uniquely identifies each entity in a table.
<b>Candidate key</b>	A minimal superkey. A superkey that does not contain a subset of attributes that is itself a superkey.
<b>Primary key</b>	A candidate key selected to uniquely identify all other attribute values in any given row. Cannot contain null entries.
<b>Secondary key</b>	An attribute (or combination of attributes) used strictly for data retrieval purposes.
<b>Foreign key</b>	An attribute (or combination of attributes) in one table whose values must either match the primary key in another table or be null.

# Integrity Rules

**TABLE 3.4** INTEGRITY RULES

ENTITY INTEGRITY	DESCRIPTION
<b>Requirement</b>	All primary key entries are unique, and no part of a primary key may be null.
<b>Purpose</b>	Guarantees that each entity will have a unique identity and ensures that foreign key values can properly reference primary key values.
<b>Example</b>	No invoice can have a duplicate number, nor can it be null. In short, all invoices are uniquely identified by their invoice number.
REFERENTIAL INTEGRITY	DESCRIPTION
<b>Requirement</b>	A foreign key may have either a null entry—as long as it is not a part of its table's primary key—or an entry that matches the primary key value in a table to which it is related. (Every non-null foreign key value <i>must</i> reference an <i>existing</i> primary key value.)
<b>Purpose</b>	Makes it possible for an attribute NOT to have a corresponding value, but it will be impossible to have an invalid entry. The enforcement of the referential integrity rule makes it impossible to delete a row in one table whose primary key has mandatory matching foreign key values in another table.
<b>Example</b>	A customer might not (yet) have an assigned sales representative (number), but it will be impossible to have an invalid sales representative (number).

# Query Languages

- Language in which user requests information from the database.
- Categories of languages
  - Procedural
  - Non-procedural, or declarative
- “Pure” languages:
  - Relational algebra
  - Tuple relational calculus
  - Domain relational calculus
- Pure languages form underlying basis of query languages that people use.

# Relational Algebra

- Procedural language
- Six basic operators
  - select:  $\sigma$
  - project:  $\Pi$
  - union:  $\cup$
  - set difference:  $-$
  - Cartesian product:  $\times$
  - rename: ...
- The operators take one or two relations as inputs and produce a new relation as a result.



# Select Operation

- The **Select** operation selects tuples that satisfy a given predicate.
- Notation:  $\uparrow_p(r)$
- $p$  is called the **selection predicate**
- To select those tuples of the *Loan* relation where the branch is “Andheri” write

$$\uparrow_{branch\_name="Andheri"}(loan)$$

- Comparisons using  $=$ ,  $\neq$ ,  $>$ ,  $\geq$ ,  $<$ ,  $\leq$  is allowed in the selection predicate.
- Ex. Tuples in which the amount lent is more than Rs.1200

$$\uparrow_{amount > 1200}(loan)$$

- We can use the connectives  $\wedge$  (**and**),  $\vee$  (**or**),  $\neg$  (**not**)
- Ex. Tuples pertaining to loans of more than Rs. 1200 made by Andheri branch.

$$\uparrow_{branch\_name="Andheri" \wedge amount > 1200}(loan)$$

# Select Operation – Example

■ Relation r

A	B	C	D
r	r	1	7
r	s	5	7
s	s	12	3
s	s	23	10

■  $\sigma_{A=B \wedge D > 5}(r)$

A	B	C	D
r	r	1	7
s	s	23	10



# Project Operation

- The Project operation is a unary operation that returns its argument relation, with certain attributes left out.
- Notation:  $\Pi_{A_1, A_2, \dots, A_k}(r)$   
where  $A_1, A_2$  are attribute names and  $r$  is a relation name.
- The result is defined as the relation of  $k$  columns obtained by erasing the columns that are not listed
- Duplicate rows removed from result, since relations are sets
- Example: To eliminate the *branch\_name* attribute of loan

$$\Pi_{loan\_number, amonut}(loan)$$

# Project Operation – Example

- Relation  $r$ :

$A$	$B$	$C$
$r$	10	1
$r$	20	1
$s$	30	1
$s$	40	2

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

$A$	$C$
$r$	1
$r$	1
$s$	1
$s$	2

=

$A$	$C$
$r$	1
$s$	1
$s$	2

# Composition of Relational Operations

- Find those customers who live in Mulund.  
write:

$$\Pi_{customer\_name} (\sigma_{Customer\_city = "Mulund"} (Customer))$$

- Instead of giving the name of the relation as the argument of the projection operation, we give an expression that evaluates the relation.
- Relation-algebra operation can be composed together into a relational-algebra expression.

# Union Operation

- Notation:  $r \cup s$
- Defined as:

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

- For  $r \cup s$  to be valid.
  1.  $r, s$  must have the **same arity** (same number of attributes)
  2. The attribute domains must be **compatible** (example: 2<sup>nd</sup> column of  $r$  deals with the same type of values as does the 2<sup>nd</sup> column of  $s$ )
- Example: to find all customers with either an account or a loan

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

# Union Operation – Example

- Relations  $r$ ,  $s$ :

$A$	$B$
$r$	1
$r$	2
$s$	1

$r$

$A$	$B$
$r$	2
$s$	3

$s$

■  $r \cup s$ :

$A$	$B$
$r$	1
$r$	2
$s$	1
$s$	3

# Set Difference Operation

- Notation  $r - s$
- Defined as:

$$r - s = \{t \mid t \in r \textbf{ and } t \notin s\}$$

- Set differences must be taken between **compatible** relations.
  - $r$  and  $s$  must have the same arity
  - attribute domains of  $r$  and  $s$  must be compatible
- Example: to find all customers of the bank who have an account but not a loan

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

# Set Difference Operation – Example

- Relations

A	B
r	1
r	2
s	1

*r*

A	B
r	2
s	3

*s*

■  $r - s$ :

A	B
r	1
s	1



# Cartesian-Product Operation

- Notation  $r \times s$
- The Cartesian product of relations  $r_1$  and  $r_2$  as

$$r_1 \times r_2$$

$$r = \text{borrower} \times \text{loan}$$

- Assume that we have  $n_1$  tuples in  $r_1$  and  $n_2$  tuples in  $r_2$ . Then, there are  $n_1 * n_2$  ways of choosing a pair of tuples – one tuple from each relation, so there are  $n_1 * n_2$  tuples in  $r$ .
- For some tuples  $t$  in  $r$ , it may be that  $t[\text{borrower.loanno}] \neq t[\text{loan.loanno}]$

# Cartesian-Product Operation – Example

■ Relations  $r$ ,  $s$ :

$A$	$B$
$r$	1
$s$	2

$r$

$C$	$D$	$E$
$r$	10	$a$
$s$	10	$a$
$s$	20	$b$
$x$	10	$b$

$s$

■  $r \times s$ :

$A$	$B$	$C$	$D$	$E$
$r$	1	$r$	10	$a$
$r$	1	$s$	10	$a$
$r$	1	$s$	20	$b$
$r$	1	$x$	10	$b$
$s$	2	$r$	10	$a$
$s$	2	$s$	10	$a$
$s$	2	$s$	20	$b$
$s$	2	$x$	10	$b$

# Composition of Operations

- Can build expressions using multiple operations
- Example:  $\sigma_{A=C}(r \times s)$

- $r \times s$

A	B	C	D	E
r	1	r	10	a
r	1	s	10	a
r	1	s	20	b
r	1	x	10	b
s	2	r	10	a
s	2	s	10	a
s	2	s	20	b
s	2	x	10	b

- $\sigma_{A=C}(r \times s)$

A	B	C	D	E
r	1	r	10	a
s	2	s	10	a
s	2	s	20	b

# Cartesian-Product Operation

- To find the names of all customers who have a loan at the Andheri branch.

*branch\_name = "Andheri"(borrower X loan)*

- To get tuples of borrower X loan that pertain to customers who have a loan at the Andheri branch.

*borrower.loanno= loan.loanno*

*( branch\_name = "Andheri"(borrower X loan))*

- Finally we want only customer\_name, we do projection:

*customer\_name ( borrower.loanno= loan.loanno*

*( branch\_name = "Andheri"(borrower X loan)))*

# Rename Operation

- Allows us to name, and therefore to refer to, the results of relational-algebra expressions.
- Allows us to refer to a relation by more than one name.
- Example:

$$\dots_X(E)$$

returns the expression  $E$  under the name  $X$

- If a relational-algebra expression  $E$  has arity  $n$ , then

$$\dots_X(A_1, A_2, \dots, A_n)(E)$$

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

# Illustration - Rename Operation

- Find the largest account balance in the bank
  - Two steps:
    - Compute a temporary relation consists of those balances that are not the largest
    - Take the set difference between the relation *balance(account)* and the temporary relation just computed.

# Illustration - Rename Operation

- Step 1:
  - To compare the values of all account balances – compute the Cartesian product  $account \times account$
  - Use the rename operation to rename one reference to the account relation.  
 $account.balance ( \quad account.balance < d.balance ( account \times_d (account)))$
  - This expression gives those balances in the  $account$  relation for which a larger balance appears somewhere in the  $account$  relation (renamed as  $d$ )
  - The result contains all the balances *except* the largest one.



# Illustration - Rename Operation

- Step 2:
  - To find the largest account balance in the bank

*balance(account) –*  
*account.balance ( account.balance < d.balance ( account X*  
*(account)))* <sub>d</sub>

# Banking Example

*branch (branch\_name, branch\_city, assets)*

*customer (customer\_name, customer\_street,  
customer\_city)*

*account (account\_number, branch\_name, balance)*

*loan (loan\_number, branch\_name, amount)*

*depositor (customer\_name, account\_number)*

*borrower (customer\_name, loan\_number)*

# Example Queries

- Find all loans of over \$1200

$$\sigma_{amount > 1200} (loan)$$

- Find the loan number for each loan of an amount greater than \$1200

$$\Pi_{loan\_number} (\sigma_{amount > 1200} (loan))$$

- Find the names of all customers who have a loan, an account, or both, from the bank

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

# Example Queries

- Find the names of all customers who have a loan at the Perryridge branch.

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

- Find the names of all customers who have a loan at the Perryridge branch but do not have an account at any branch of the bank.

$$\Pi_{customer\_name} (\sigma_{branch\_name = "Perryridge"} (\sigma_{borrower.loan\_number = loan.loan\_number} (borrower \times loan))) - \Pi_{customer\_name} (depositor)$$

# Example Queries

- Find the names of all customers who have a loan at the Perryridge branch.

- Query 1

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

- Query 2

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

# Example Queries

- Find the largest account balance
  - Strategy:
    - Find those balances that are *not* the largest
      - Rename *account* relation as *d* so that we can compare each account balance with all others
    - Use set difference to find those account balances that were *not* found in the earlier step.
  - The query is:

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

# Formal Definition

- A basic expression in the relational algebra consists of either one of the following:
  - A relation in the database
  - A constant relation
- Let  $E_1$  and  $E_2$  be relational-algebra expressions; the following are all relational-algebra expressions:
  - $E_1 \cup E_2$
  - $E_1 - E_2$
  - $E_1 \times E_2$
  - $\sigma_P(E_1)$ ,  $P$  is a predicate on attributes in  $E_1$
  - $\Pi_S(E_1)$ ,  $S$  is a list consisting of some of the attributes in  $E_1$
  - $\rho_x(E_1)$ ,  $x$  is the new name for the result of  $E_1$



# Additional Operations

We define additional operations that do not add any power to the relational algebra, but that simplify common queries.

- Set intersection
- Natural join
- Division
- Assignment

# Set-Intersection Operation

- Notation:  $r \cap s$
- Defined as:
- $r \cap s = \{ t \mid t \in r \textbf{ and } t \in s \}$
- Assume:
  - $r, s$  have the *same arity*
  - attributes of  $r$  and  $s$  are compatible
- Note:  $r \cap s = r - (r - s)$
- Find all customers who have both a loan and an account

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

# Set-Intersection Operation – Example

- Relation  $r$ ,  $s$ :

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

$r$

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

$s$

- $r \cap s$

A	B
$\alpha$	2

# The Natural-Join Operation

- Allows us to combine certain selections and a Cartesian Product into one operation
- Denoted by the **join** symbol
- Selection forcing **equality** on those attributes that **appear in both relation** schemas and finally removes duplicate attributes.
- Find the names of all customers who have a loan at the bank, along with the loan number and the loan amount.

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

- The above query using the natural join as follows:

$\Pi_{\text{customer\_name}, \text{loan.loan\_number}, \text{amout}} (\text{borrower} \bowtie \text{loan})$



# Natural Join Operation – Example

- Relations  $r$ ,  $s$ :

$A$	$B$	$C$	$D$
$r$	1	$r$	$a$
$s$	2	$x$	$a$
$x$	4	$s$	$b$
$r$	1	$x$	$a$
$u$	2	$s$	$b$

$r$

$B$	$D$	$E$
1	$a$	$r$
3	$a$	$s$
1	$a$	$x$
2	$b$	$u$
3	$b$	$e$

$s$

■  $r \bowtie s$

$A$	$B$	$C$	$D$	$E$
$r$	1	$r$	$a$	$r$
$r$	1	$r$	$a$	$x$
$r$	1	$x$	$a$	$r$
$r$	1	$x$	$a$	$x$
$r$	2	$s$	$b$	$u$
$u$				

# The Natural-Join Operation

- Find the names of all branches with customers who have an account in the bank and who live in Andheri.

$\Pi_{\text{branch\_name}}(\sigma_{\text{customer-city}=\text{"Andheri"}}(\text{Customer} \bowtie \text{Account Depositor}))$

- Find all customers who have both a loan and an account at the bank

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

# Division Operation

- Notation:  $r \div s$
- Suited to queries that include the phrase “for all”.
- Find all customers who have an account at all the branches located in “Andheri”

$r1 = \Pi \text{ branch\_name}(\sigma_{\text{branch\_city}=\text{“Andheri”}}(\text{Branch}))$

- Find all (customer\_name, branch\_name) for which the customer has an account at a branch by

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

- Find customers who appear in r2 with every branch name in r1

$r2 = \Pi \text{ customer\_name, branch\_name } (\text{depositor} \bowtie \text{account}) \div \Pi \text{ branch\_name}(\sigma_{\text{branch\_city}=\text{“Andheri”}}(\text{Branch}))$



# Division Operation (Cont.)

- Property
  - Let  $q = r \div s$
  - Then  $q$  is the largest relation satisfying  $q \times s \subseteq r$
- Definition in terms of the basic algebra operation  
Let  $r(R)$  and  $s(S)$  be relations, and let  $S \subseteq R$

$$r \div s = \Pi_{R-S}(r) - \Pi_{R-S}((\Pi_{R-S}(r) \times s) - \Pi_{R-S,S}(r))$$

To see why

- $\Pi_{R-S,S}(r)$  simply reorders attributes of  $r$
- $\Pi_{R-S}(\Pi_{R-S}(r) \times s) - \Pi_{R-S,S}(r)$  gives those tuples  $t$  in  $\Pi_{R-S}(r)$  such that for some tuple  $u \in s$ ,  $tu \notin r$ .

R1 =

<i>branch_name</i>
Brighton
Downtown

R2 =

<i>customer_name</i>	<i>branch_name</i>
Hayes	Perryridge
Johnson	Downtown
Johnson	Brighton
Jones	Brighton
Lindsay	Redwood
Smith	Mianus
Turner	Round Hill

Result = A relation that has the schema (customer\_name) and that contains the tuple (Johnson)

# Division Operation – Example

■ Relations  $r, s$ :

$A$	$B$
$r$	1
$r$	2
$r$	3
$r$	1
$s$	1
$x$	1
$u$	1
$u$	3
$u$	4
$è$	6
$è$	1
$s$	2

$r$

$B$
1
2

$s$

■  $r \div s$ :

$A$
$r$
$s$

# Another Division Example

■ Relations  $r, s$ :

$A$	$B$	$C$	$D$	$E$
$r$	$a$	$r$	$a$	$1$
$r$	$a$	$x$	$a$	$1$
$r$	$a$	$x$	$b$	$1$
$s$	$a$	$x$	$a$	$1$
$s$	$a$	$x$	$b$	$3$
$x$	$a$	$x$	$a$	$1$
$x$	$a$	$x$	$b$	$1$
$x$	$a$	$s$	$b$	$1$

$r$

$D$	$E$
$a$	$1$
$b$	$1$

$s$

■  $r \div s$ :

$A$	$B$	$C$
$r$	$a$	$x$
$x$	$a$	$x$

# Assignment Operation

- The assignment operation ( $\leftarrow$ ) provides a convenient way to express complex queries.
  - Write query as a sequential program consisting of
    - a series of assignments
    - followed by an expression whose value is displayed as a result of the query.
  - Assignment must always be made to a temporary relation variable.
- Example: Write  $r \div s$  as

$temp1 \leftarrow \Pi_{R-S}(r)$

$temp2 \leftarrow \Pi_{R-S}((temp1 \times s) - \Pi_{R-S,S}(r))$

$result = temp1 - temp2$

- The result to the right of the  $\leftarrow$  is assigned to the relation variable on the left of the  $\leftarrow$ .
- May use variable in subsequent expressions.

# Extended Relational-Algebra-Operations

- Generalized Projection
- Aggregate Functions
- Outer Join

# Generalized Projection

- Extends the projection operation by allowing arithmetic functions to be used in the projection list.

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

- $E$  is any relational-algebra expression
- Each of  $F_1, F_2, \dots, F_n$  are arithmetic expressions involving constants and attributes in the schema of  $E$ .
- Given relation *credit\_info*(*customer\_name*, *limit*, *credit\_balance*), find how much more each person can spend:

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



# Aggregate Functions and Operations

- **Aggregation function** takes a collection of values and returns a single value as a result.

**avg:** average value

**min:** minimum value

**max:** maximum value

**sum:** sum of values

**count:** number of values

- **Aggregate operation** in relational algebra

$$G_1, G_2, \dots, G_n \left[ F_1(A_1), F_2(A_2), \dots, F_n(A_n) \right] (E)$$

$E$  is any relational-algebra expression

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

# Aggregate Operation – Example

- Relati  
on  $r$ :

$A$	$B$	$C$
$r$	$r$	7
$r$	$s$	7
$s$	$s$	3
$s$	$s$	10

■  $g_{\text{sum}(c)}(r)$

$\text{sum}(c)$
27

# Aggregate Operation – Example

- Relation *account* grouped by

*branch\_name*

<i>branch_name</i>	<i>account_number</i>	<i>balance</i>
Perryridge	A-102	400
Perryridge	A-201	900
Brighton	A-217	750
Brighton	A-215	750
Redwood	A-222	700

*branch\_name*  $\mathcal{G}$  **sum**(*balance*) (*account*)




<i>branch_name</i>	<b>sum</b> ( <i>balance</i> )
Perryridge	1300
Brighton	1500
Redwood	700

# Aggregate Functions (Cont.)

- Result of aggregation does not have a name
  - Can use rename operation to give it a name
  - For convenience, we permit renaming as part of aggregate operation

*branch\_name* **g** **sum**(balance) **as** sum\_balance (*account*)

# Outer Join

- An extension of the join operation that avoids loss of information.
- Computes the join and then adds tuples from one relation that does not match tuples in the other relation to the result of the join.
- There are three forms
  - Left outer join
  - Right outer join 
  - Full outer join 
- Uses *null* values: 
  - *null* signifies that the value is unknown or does not exist

# Outer Join – Example

- Relation *loan*

<i>loan_number</i>	<i>branch_name</i>	<i>amount</i>
L-170	Downtown	3000
L-230	Redwood	4000
L-260	Perryridge	1700

- Relation *borrower*

<i>customer_name</i>	<i>loan_number</i>
Jones	L-170
Smith	L-230
Hayes	L-155

# Outer Join – Example

- Join

*loan* ⋈ *borrower*

<i>loan_number</i>	<i>branch_name</i>	<i>amount</i>	<i>customer_name</i>
L-170	Downtown	3000	Jones
L-230	Redwood	4000	Smith

- Left Outer Join

*loan* ⋈<sub>L</sub> *borrower*

<i>loan_number</i>	<i>branch_name</i>	<i>amount</i>	<i>customer_name</i>
L-170	Downtown	3000	Jones
L-230	Redwood	4000	Smith
L-260	Perryridge	1700	<i>null</i>



# Outer Join – Example

## ■ Right Outer Join

*loan* ⋈<sub>R</sub> *borrower*

<i>loan_number</i>	<i>branch_name</i>	<i>amount</i>	<i>customer_name</i>
L-170	Downtown	3000	Jones
L-230	Redwood	4000	Smith
L-155	<i>null</i>	<i>null</i>	Hayes

## ■ Full Outer Join

*loan* ⋈<sub>F</sub> *borrower*

<i>loan_number</i>	<i>branch_name</i>	<i>amount</i>	<i>customer_name</i>
L-170	Downtown	3000	Jones
L-230	Redwood	4000	Smith
L-260	Perryridge	1700	<i>null</i>
L-155	<i>null</i>	<i>null</i>	Hayes

# 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 result of any arithmetic expression involving *null* is *null*.
- Aggregate functions simply ignore null values (as in SQL)
- For duplicate elimination and grouping, null is treated like any other value, and two nulls are assumed to be the same (as in SQL)

# Modification of the Database

- The content of the database may be modified using the following operations:
  - Deletion
  - Insertion
  - Updating
- All these operations are expressed using the assignment operator.

# Deletion

- A delete request is expressed similarly to a query, except instead of displaying tuples to the user, the selected tuples are removed from the database.
- Can delete only whole tuples; cannot delete values on only particular attributes
- A deletion is expressed in relational algebra by:

$$r \leftarrow r - E$$

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

# Deletion Examples

- Delete all account records in the Perryridge branch.

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

- Delete all loan records with amount in the range of 0 to 50

$loan \leftarrow loan - \sigma_{amount \leq 50}(loan)$

# Insertion

- To insert data into a relation, we either:
  - specify a tuple to be inserted
  - write a query whose result is a set of tuples to be inserted
- in relational algebra, an insertion is expressed by:

$$r \leftarrow r \cup E$$

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

- The insertion of a single tuple is expressed by letting  $E$  be a constant relation containing one tuple.

# Insertion Examples

- Insert information in the database specifying that Smith has \$1200 in account A-973 at the Perryridge branch.

$account \leftarrow account \cup \{("A-973", "Perryridge", 1200)\}$

$depositor \leftarrow depositor \cup \{("Smith", "A-973")\}$



# Updating

- A mechanism to change a value in a tuple without changing *all* values in the tuple
- Use the generalized projection operator to do this task

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

- Each  $F_i$  is either
  - the  $i^{\text{th}}$  attribute of  $r$ , if the  $i^{\text{th}}$  attribute is not updated, or,
  - if the attribute is to be updated  $F_i$  is an expression, involving only constants and the attributes of  $r$ , which gives the new value for the attribute

# Update Examples

- Make interest payments by increasing all balances by 5 percent.

$account \leftarrow \Pi_{account\_number, branch\_name, balance * 1.05} (account)$

- Pay all accounts with balances over \$10,000 6 percent interest and pay all others 5 percent

$account \leftarrow \Pi_{account\_number, branch\_name, balance * 1.06} (\sigma_{BAL > 10000} (account)) \cup \Pi_{account\_number, branch\_name, balance * 1.05} (\sigma_{BAL \leq 10000} (account))$

# Query Processing and Optimisation

Unit - 5

# Query Processing

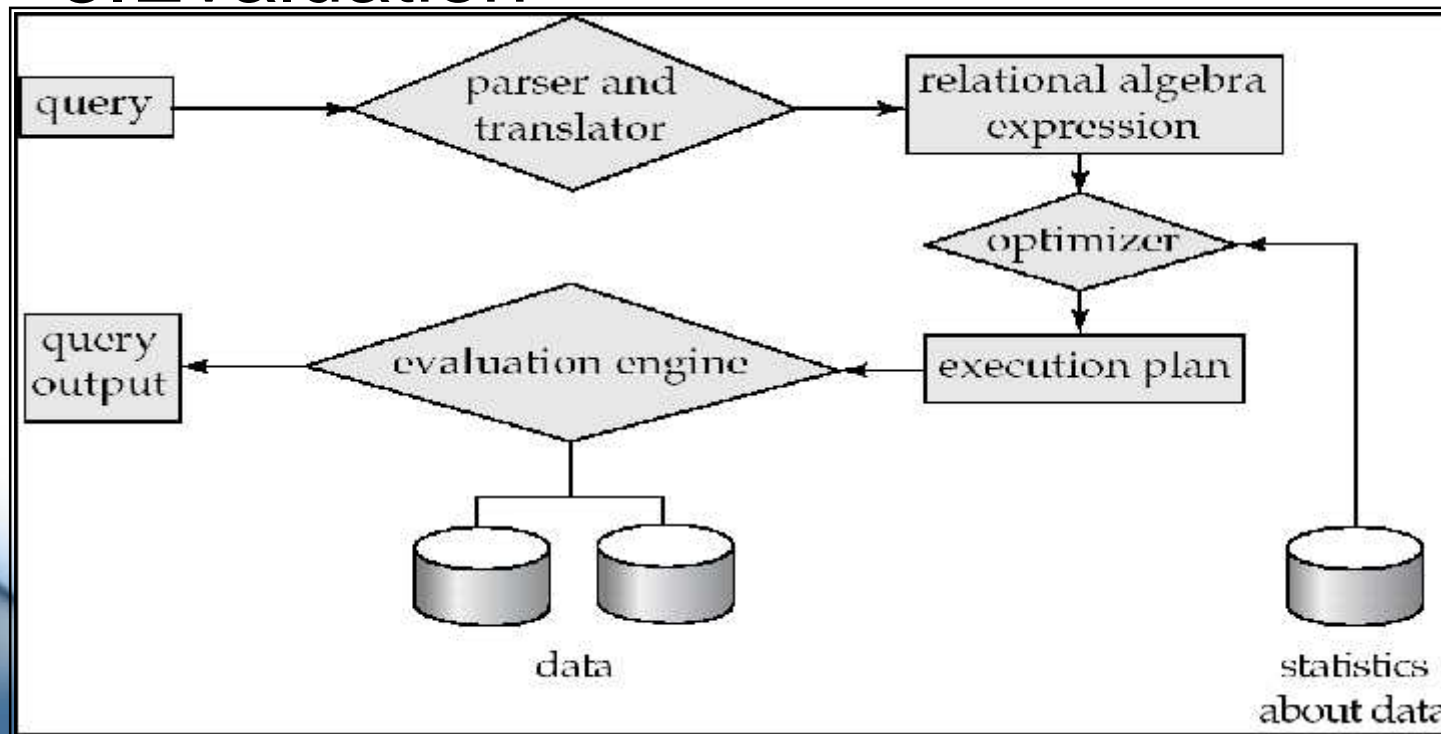
- **Query Processing** is the procedure of transforming a high-level query (such as SQL) into a correct and efficient execution plan expressed in low-level language that performs the required retrievals and manipulations in the database.

# Basic Steps in Query Processing

1. Parsing and translation

2. Optimization

3. Evaluation

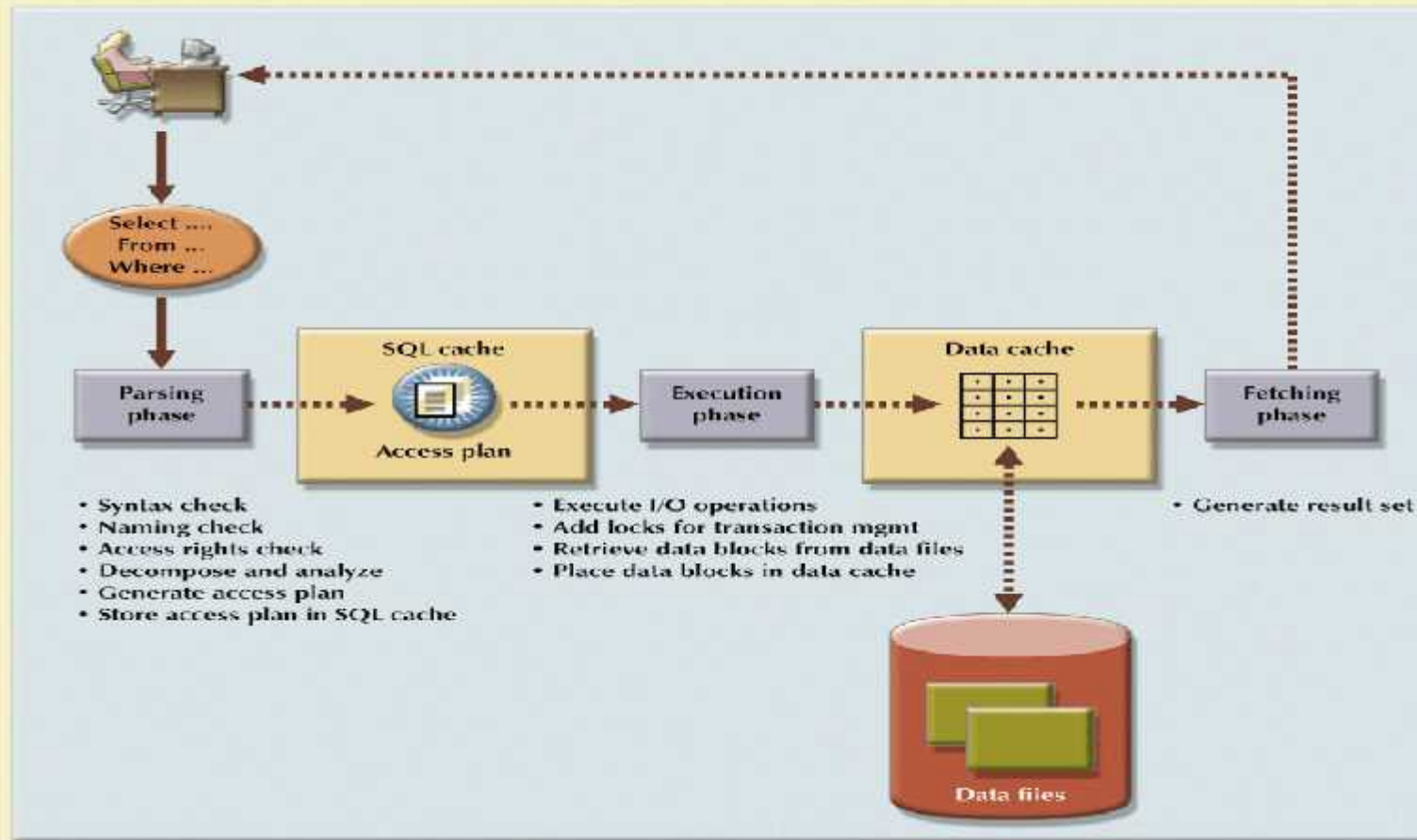


# SQL Parsing Phase

- Breaking down (parsing) query into smaller units and transforming original SQL query into slightly different version of original SQL code
  - Fully equivalent
    - Optimized query results are always the same as original query
  - More efficient
    - Optimized query will almost always execute faster than original query

# SQL Parsing Phase (continued)

FIGURE 11.2 Query processing



# SQL Parsing Phase (continued)

- Query optimizer analyzes SQL query and finds most efficient way to access data
- Access plans are DBMS-specific and translate client's SQL query into series of complex I/O operations required to read the data from the physical data files and generate result set



# SQL Parsing Phase (continued)

TABLE  
11.3

Sample DBMS Access Plan I/O Operations

OPERATION	DESCRIPTION
Table Scan (Full)	Reads the entire table sequentially, from the first row to the last row, one row at a time (slowest)
Table Access (Row ID)	Reads a table row directly, using the row ID value (fastest)
Index Scan (Range)	Reads the index first to obtain the row IDs and then accesses the table rows directly (faster than a full table scan)
Index Access (Unique)	Used when a table has a unique index in a column
Nested Loop	Reads and compares a set of values to another set of values, using a nested loop style (slow)
Merge	Merges two data sets (slow)
Sort	Sorts a data set (slow)

# SQL Execution Phase

- All I/O operations indicated in access plan are executed

# SQL Fetching Phase

- Rows of resulting query result set are returned to client
- DBMS may use temporary table space to store temporary data

# Query Processing and Optimization

- Syntax Analyzer
- Query Decomposition
- Query Optimization
- Cost estimation in query optimization
- Pipelining and Materialization
- Structure of Query Execution Plans

# Syntax Analyzer

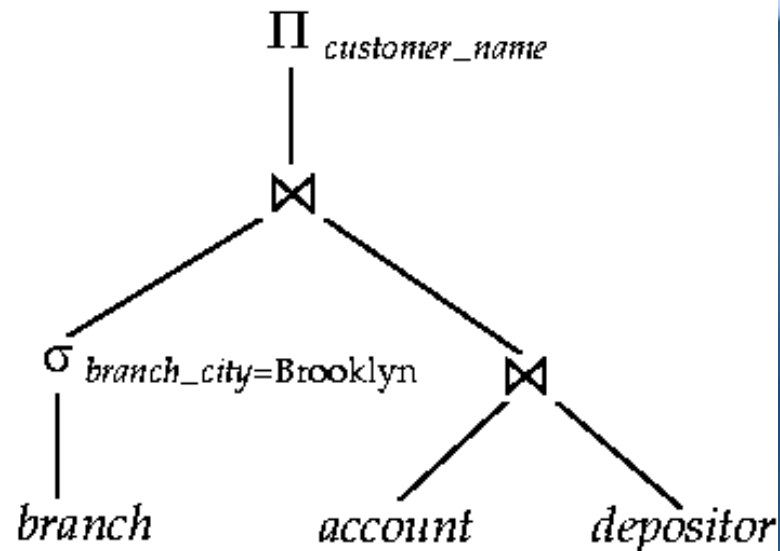
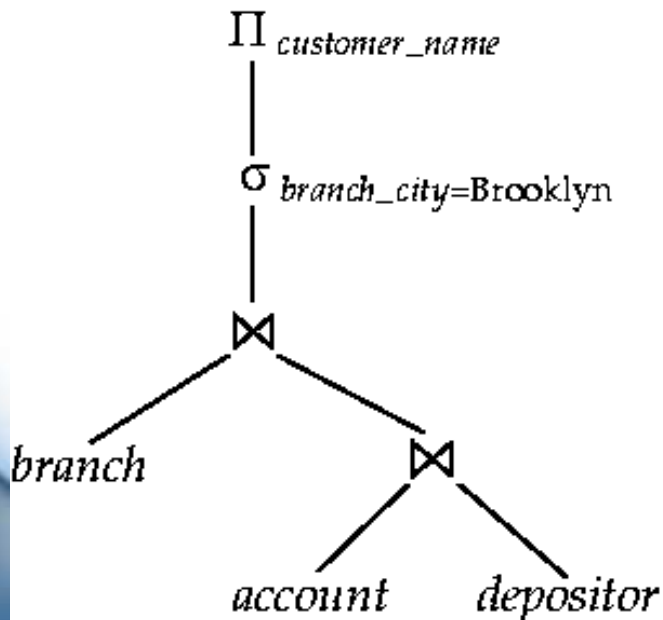
- Takes the query form users
- Parses it into tokens
- Analyses the tokens and their order as per the rule of the language grammar
- If an error then query rejected and error code generated.

# Query Decomposition

- Aims to transform a high-level query into a relationl-algebra query to check – query is syntactically and semantically correct.
- The five stages of query decomposition are:
  - Query Analysis
  - Query Normalization
  - Semantic Analysis
  - Query simplifier
  - Query restructuring

# Query Analysis

- Query is validated – using system catalogues, to ensure that all database objects referred are defined in the database.
- Following two notations are used
  - **Query –tree notation** (Relational algebra tree)
  - Query Graph notation



# Query Normalization

- Goal is to avoid redundancy.
- Converts the query into a normalised form that can be more easily manipulated.
- A set of equivalency rules is applied so that the projection and selection operations are simplified.
- Equivalence Rule



# Semantic Analyzer

- Checks Correctness and Contradiction
- Correctness
  - For ex. Missing join specification do not contribute to the generation of the result.
- Contradiction
  - If query predicate cannot be satisfied by any tuple
  - Select ename from emp where job="Clerk" and job="Manager"

# Query Simplifier

- Objective is to transform query to semantically equivalent but more easily and efficiently computed forms.
- Detects redundant predicate used in original query.
- Eliminates those without changing the meaning of the query.

## Query Restructuring

- Final stage of query decomposition
- Query can be restructured to give more efficient implementation
- Transformation rules are used.

# Query Optimization

- A query typically has many possible execution strategies, and the process of choosing a suitable one for processing a query is known as ***Query Optimization***.
- The primary goal is of choosing an efficient execution strategy for processing a query
- The basic issues in query optimization are:
  - How to use available indexes.
  - How to use memory to accumulate information and perform intermediate steps such as sorting
  - How to determine the order in which joins should be performed.

# Query Optimization

- There are two main techniques for implementing query optimization:
  - Based on *heuristic rules* for ordering the operations in query execution strategy.
  - Based on systematic estimation of the cost of different execution strategies and choosing the execution plan with the lowest cost estimate – *Cost-based query optimization*.

# Cost-based Query Optimization

- Cost difference between evaluation plans for a query can be enormous
  - E.g. seconds vs. days in some cases
- Steps in **cost-based query optimization**
  1. Generate logically equivalent expressions using **equivalence rules**
  2. Annotate resultant expressions to get alternative query plans
  3. Choose the cheapest plan based on **estimated cost**
- Estimation of plan cost based on:
  - Statistical information about relations. Examples:
    - number of tuples, number of distinct values for an attribute
  - Statistics estimation for intermediate results
    - to compute cost of complex expressions
  - Cost formulae for algorithms, computed using statistics

# Cost components of Query Execution

- The cost of executing a query includes the following components:
  - Access cost to secondary storage
  - Storage cost
  - Computation cost
  - Memory uses cost
  - Communication cost

# Cost components of Query Execution

**The DBMS is expected to hold the following information in its system catalogue:**

- The number of tuples in relation R
- The average record size in relation R
- The number of blocks requires to store relation R
- The number of tuples of R that fit into one block
- Primary access method of each file
- Primary access attributes for each file
- The number of levels of each multi-level index I (Primary, secondary or clustering)
- The number of first-level index blocks
- The number of distinctive values that appear for attribute A in relation R
- The minimum and maximum possible values for attribute in relation R



# Classification of Physical Storage Media

- ☐ Speed with which data can be accessed
- ☐ Cost per unit of data
- ☐ Reliability
  - ☐ data loss on power failure or system crash
  - ☐ physical failure of the storage device
- ☐ Can differentiate storage into:
  - ☐ **volatile storage:** loses contents when power is switched off
  - ☐ **non-volatile storage:**
    - Contents persist even when power is switched off.
    - Includes secondary and tertiary storage, as well as batter- backed up main-memory.



# Physical Storage Media

- ❑ **Cache** – fastest and most costly form of storage; volatile; managed by the computer system hardware.
- ❑ **Main memory:**
  - ❑ fast access (10s to 100s of nanoseconds; 1 nanosecond =  $10^{-9}$  seconds)
  - ❑ generally too small (or too expensive) to store the entire database
    - capacities of up to a few Gigabytes widely used currently
    - Capacities have gone up and per-byte costs have decreased steadily and rapidly (roughly factor of 2 every 2 to 3 years)
  - ❑ **Volatile** — contents of main memory are usually lost if a power failure or system crash occurs.

# Physical Storage Media (Cont.)

## □ **Flash memory**

- ▣ Data survives power failure
- ▣ Data can be written at a location only once, but location can be erased and written to again
  - Can support only a limited number (10K – 1M) of write/erase cycles.
  - Erasing of memory has to be done to an entire bank of memory
- ▣ Reads are roughly as fast as main memory
- ▣ But writes are slow (few microseconds), erase is slower
- ▣ Cost per unit of storage roughly similar to main memory
- ▣ Widely used in embedded devices such as digital cameras
- ▣ Is a type of EEPROM (Electrically Erasable Programmable Read-Only Memory)

# Physical Storage Media (Cont.)

- **Magnetic-disk**

- Data is stored on spinning disk, and read/written magnetically
- Primary medium for the long-term storage of data; typically stores entire database.
- Data must be moved from disk to main memory for access, and written back for storage
  - Much slower access than main memory (more on this later)
- **direct-access** – possible to read data on disk in any order, unlike magnetic tape
- Capacities range up to roughly 400 GB currently
  - Much larger capacity and cost/byte than main memory/flash memory
  - Growing constantly and rapidly with technology improvements (factor of 2 to 3 every 2 years)
- Survives power failures and system crashes
  - disk failure can destroy data, but is rare

# Physical Storage Media (Cont.)

## □ **Optical storage**

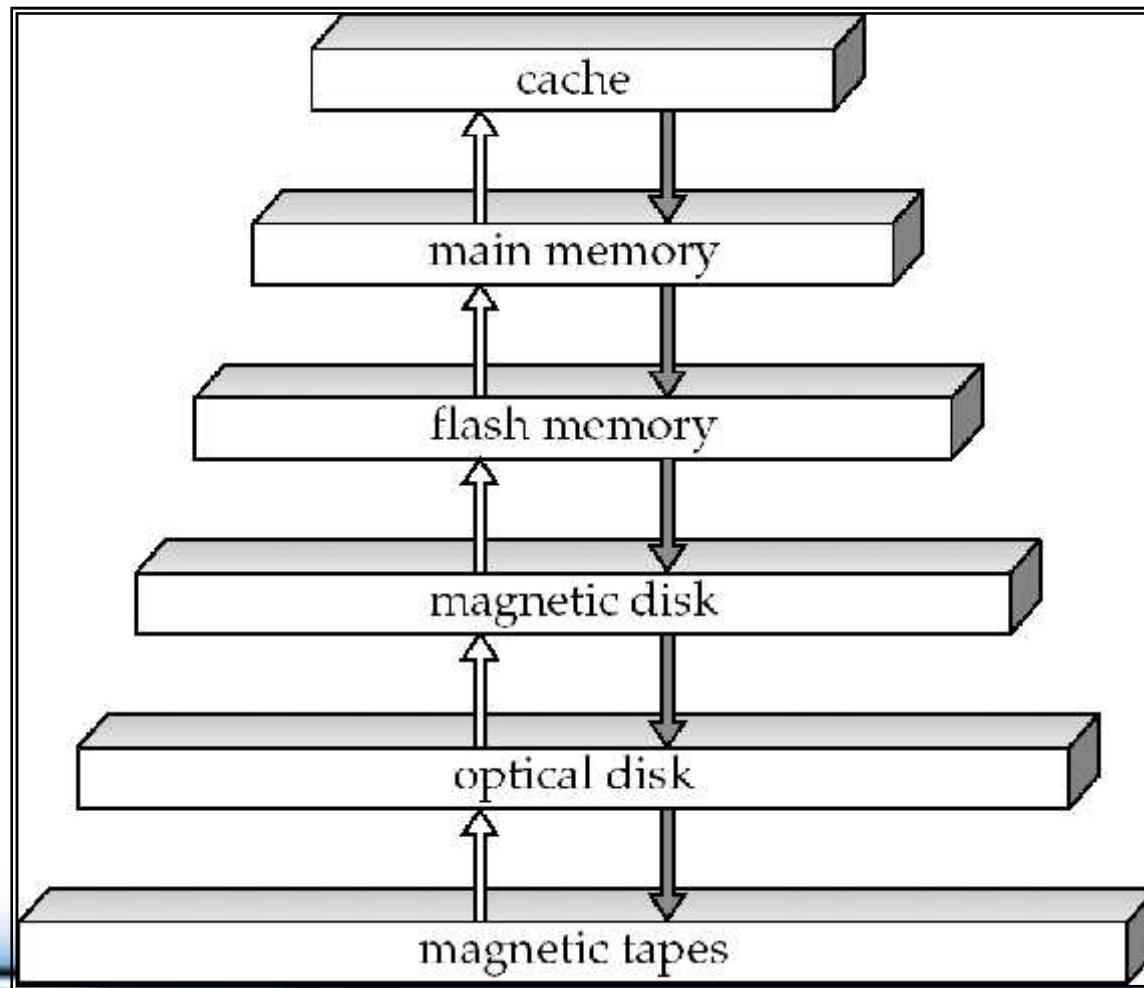
- ▣ non-volatile, data is read optically from a spinning disk using a laser
- ▣ CD-ROM (640 MB) and DVD (4.7 to 17 GB) most popular forms
- ▣ Write-one, read-many (WORM) optical disks used for archival storage (CD-R, DVD-R, DVD+R)
- ▣ Multiple write versions also available (CD-RW, DVD-RW, DVD+RW, and DVD-RAM)
- ▣ Reads and writes are slower than with magnetic disk
- ▣ **Juke-box** systems, with large numbers of removable disks, a few drives, and a mechanism for automatic loading/unloading of disks available for storing large volumes of data

# Physical Storage Media (Cont.)

## □ **Tape storage**

- non-volatile, used primarily for backup (to recover from disk failure), and for archival data
- **sequential-access** – much slower than disk
- very high capacity (40 to 300 GB tapes available)
- tape can be removed from drive  $\Rightarrow$  storage costs much cheaper than disk, but drives are expensive
- Tape jukeboxes available for storing massive amounts of data
  - hundreds of terabytes (1 terabyte =  $10^9$  bytes) to even a petabyte (1 petabyte =  $10^{12}$  bytes)

# Storage Hierarchy

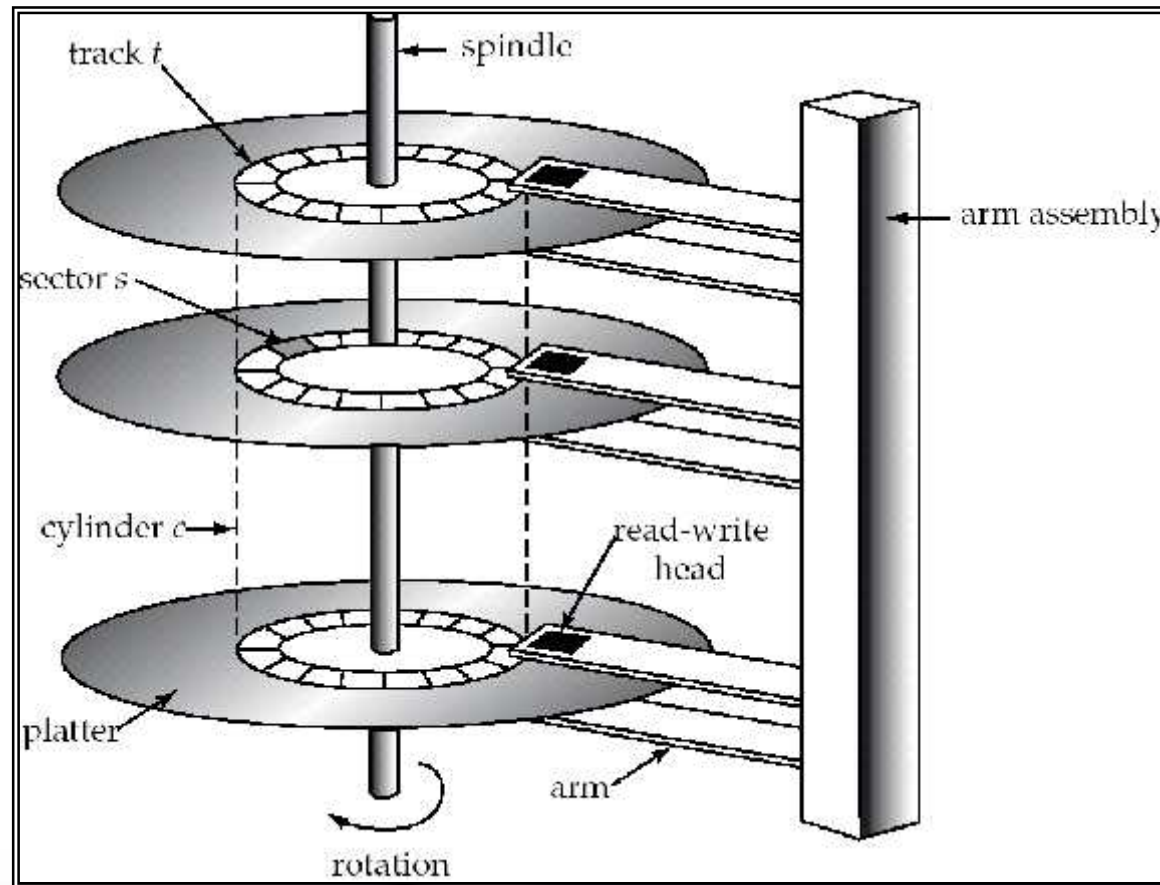


# Storage Hierarchy (Cont.)

- **primary storage:** Fastest media but volatile (cache, main memory).
- **secondary storage:** next level in hierarchy, non-volatile, moderately fast access time
  - also called **on-line storage**
  - E.g. flash memory, magnetic disks
- **tertiary storage:** lowest level in hierarchy, non-volatile, slow access time
  - also called **off-line storage**
  - E.g. magnetic tape, optical storage



# Magnetic Hard Disk Mechanism



**NOTE: Diagram is schematic, and simplifies the structure of actual disk drives**



# Performance Measures of Disks

- **Access time** – the time it takes from when a read or write request is issued to when data transfer begins. Consists of:
  - **Seek time** – time it takes to reposition the arm over the correct track.
    - Average seek time is 1/2 the worst case seek time.
      - Would be 1/3 if all tracks had the same number of sectors, and we ignore the time to start and stop arm movement
    - 4 to 10 milliseconds on typical disks
  - **Rotational latency** – time it takes for the sector to be accessed to appear under the head.
    - Average latency is 1/2 of the worst case latency.
    - 4 to 11 milliseconds on typical disks (5400 to 15000 r.p.m.)
- **Data-transfer rate** – the rate at which data can be retrieved from or stored to the disk.
  - 25 to 100 MB per second max rate, lower for inner tracks
  - Multiple disks may share a controller, so rate that controller can handle is also important
    - E.g. ATA-5: 66 MB/sec, SATA: 150 MB/sec, Ultra 320 SCSI: 320 MB/s
    - Fiber Channel (FC2Gb): 256 MB/s

# Performance Measures (Cont.)

- **Mean time to failure (MTTF)** – the average time the disk is expected to run continuously without any failure.
  - Typically 3 to 5 years
  - Probability of failure of new disks is quite low, corresponding to a “theoretical MTTF” of 500,000 to 1,200,000 hours for a new disk
    - E.g., an MTTF of 1,200,000 hours for a new disk means that given 1000 relatively new disks, on an average one will fail every 1200 hours
  - MTTF decreases as disk ages

# Optimization of Disk-Block Access

- **Block** – a contiguous sequence of sectors from a single track
  - ▣ data is transferred between disk and main memory in blocks
  - ▣ sizes range from 512 bytes to several kilobytes
    - Smaller blocks: more transfers from disk
    - Larger blocks: more space wasted due to partially filled blocks
    - Typical block sizes today range from 4 to 16 kilobytes
- **Disk-arm-scheduling** algorithms order pending accesses to tracks so that disk arm movement is minimized
  - ▣ **elevator algorithm** : move disk arm in one direction (from outer to inner tracks or vice versa), processing next request in that direction, till no more requests in that direction, then reverse direction and repeat

# Optimization of Disk Block Access (Cont.)

- **File organization** – optimize block access time by organizing the blocks to correspond to how data will be accessed
  - E.g. Store related information on the same or nearby cylinders.
  - Files may get **fragmented** over time
    - E.g. if data is inserted to/deleted from the file
    - Or free blocks on disk are scattered, and newly created file has its blocks scattered over the disk
    - Sequential access to a fragmented file results in increased disk arm movement
  - Some systems have utilities to defragment the file system, in order to speed up file access

# Optimization of Disk Block Access (Cont.)

- **Nonvolatile write buffers** speed up disk writes by writing blocks to a non-volatile RAM buffer immediately
  - Non-volatile RAM: battery backed up RAM or flash memory
    - Even if power fails, the data is safe and will be written to disk when power returns
  - Controller then writes to disk whenever the disk has no other requests or request has been pending for some time
  - Database operations that require data to be safely stored before continuing can continue without waiting for data to be written to disk
  - *Writes can be reordered to minimize disk arm movement*
- **Log disk** – a disk devoted to writing a sequential log of block updates
  - Used exactly like nonvolatile RAM
    - Write to log disk is very fast since no seeks are required
    - No need for special hardware (NV-RAM)
- File systems typically reorder writes to disk to improve performance
  - **Journaling file systems** write data in safe order to NV-RAM or log disk
  - Reordering without journaling: risk of corruption of file system data

# RAID

- **RAID: Redundant Arrays of Independent Disks**

- disk organization techniques that manage a large numbers of disks, providing a view of a single disk of
  - high capacity and high speed by using multiple disks in parallel, and
  - high reliability by storing data redundantly, so that data can be recovered even if a disk fails
- The chance that some disk out of a set of  $N$  disks will fail is much higher than the chance that a specific single disk will fail.
  - E.g., a system with 100 disks, each with MTTF of 100,000 hours (approx. 11 years), will have a system MTTF of 1000 hours (approx. 41 days)
  - Techniques for using redundancy to avoid data loss are critical with large numbers of disks
- Originally a cost-effective alternative to large, expensive disks
  - I in RAID originally stood for “inexpensive”
  - Today RAIDs are used for their higher reliability and bandwidth.
    - The “I” is interpreted as independent



# Improvement of Reliability via Redundancy

- **Redundancy** – store extra information that can be used to rebuild information lost in a disk failure
- E.g., **Mirroring** (or **shadowing**)
  - Duplicate every disk. Logical disk consists of two physical disks.
  - Every write is carried out on both disks
    - Reads can take place from either disk
  - If one disk in a pair fails, data still available in the other
    - Data loss would occur only if a disk fails, and its mirror disk also fails before the system is repaired
      - Probability of combined event is very small
        - » Except for dependent failure modes such as fire or building collapse or electrical power surges
- Mean time to data loss depends on mean time to failure, and mean time to repair
  - E.g. MTTF of 100,000 hours, mean time to repair of 10 hours gives mean time to data loss of  $500 \times 10^6$  hours (or 57,000 years) for a mirrored pair of disks (ignoring dependent failure modes)



# Improvement in Performance via Parallelism

- Two main goals of parallelism in a disk system:
  1. Load balance multiple small accesses to increase throughput
  2. Parallelize large accesses to reduce response time.
- Improve transfer rate by striping data across multiple disks.
- **Bit-level striping** – split the bits of each byte across multiple disks
  - In an array of eight disks, write bit  $i$  of each byte to disk  $i$ .
  - Each access can read data at eight times the rate of a single disk.
  - But seek/access time worse than for a single disk
    - Bit level striping is not used much any more
- **Block-level striping** – with  $n$  disks, block  $i$  of a file goes to disk  $(i \bmod n) + 1$ 
  - Requests for different blocks can run in parallel if the blocks reside on different disks
  - A request for a long sequence of blocks can utilize all disks in parallel

# Storage Access

- ☐ A database file is partitioned into fixed-length storage units called **blocks**. Blocks are units of both storage allocation and data transfer.
- ☐ Database system seeks to minimize the number of block transfers between the disk and memory. We can reduce the number of disk accesses by keeping as many blocks as possible in main memory.
- ☐ **Buffer** – portion of main memory available to store copies of disk blocks.
- ☐ **Buffer manager** – subsystem responsible for allocating buffer space in main memory.

# Buffer Manager

□ Programs call on the buffer manager when they need a block from disk.

1. If the block is already in the buffer, buffer manager returns the address of the block in main memory

2. If the block is not in the buffer, the buffer manager

1. Allocates space in the buffer for the block

Replacing (throwing out) some other block, if required, to make space for the new block.

Replaced block written back to disk only if it was modified since the most recent time that it was written to/fetched from the disk.

2. Reads the block from the disk to the buffer, and returns the address of the block in main memory to requester.

# File Organization

- The database is stored as a collection of *files*. Each file is a sequence of *records*. A record is a sequence of fields.
  - One approach:
    - assume record size is fixed
    - each file has records of one particular type only
    - different files are used for different relations
- This case is easiest to implement; will consider variable length records later.

# Fixed-Length Records

- Simple approach:
  - Store record  $i$  starting from byte  $n ; (i - 1)$ , where  $n$  is the size of each record.
  - Record access is simple but records may cross blocks
    - Modification: do not allow records to cross block boundaries
- Deletion of record  $i$ :  
alternatives:
  - move records  $i + 1, \dots, n$  to  $i, \dots, n - 1$
  - move record  $n$  to  $i$
  - do not move records, but link all free records on a *free list*

record 0	A-102	Perryridge	400
record 1	A-305	Round Hill	350
record 2	A-215	Mianus	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
record 8	A-218	Perryridge	700

# Free Lists

- ☐ Store the address of the first deleted record in the file header.
- ☐ Use this first record to store the address of the second deleted record, and so on
- ☐ Can think of these stored addresses as pointers since they “point” to the location of a record.
- ☐ More space efficient representation: reuse space for normal attributes of free records to store pointers. (No pointers stored in in-use records.)

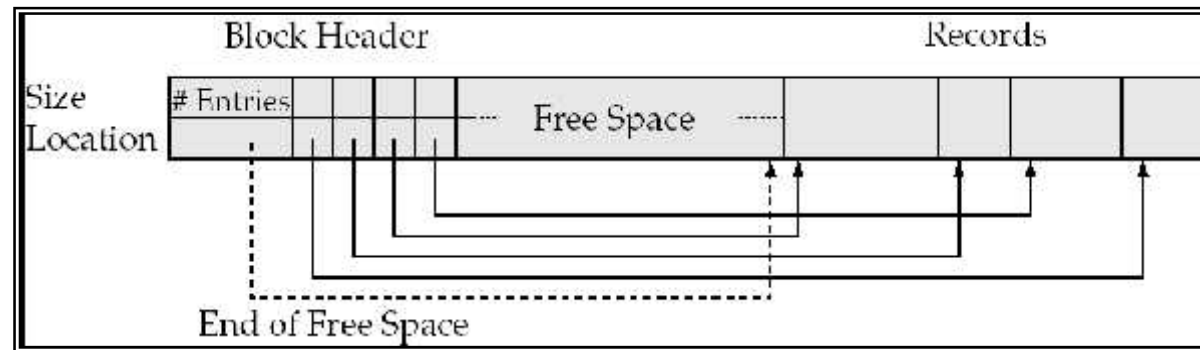
header					
record 0	A-102	Perryridge	400		
record 1					
record 2	A-215	Mianus	700		
record 3	A-101	Downtown	500		
record 4					
record 5	A-201	Perryridge	900		
record 6					
record 7	A-110	Downtown	600		
record 8	A-218	Perryridge	700		

# Variable-Length Records

- Variable-length records arise in database systems in several ways:
  - Storage of multiple record types in a file.
  - Record types that allow variable lengths for one or more fields.
  - Record types that allow repeating fields (used in some older data models).



# Variable-Length Records: Slotted Page Structure



- ☐ Slotted page header contains:
  - ☒ number of record entries
  - ☒ end of free space in the block
  - ☒ location and size of each record
- ☐ Records can be moved around within a page to keep them contiguous with no empty space between them; entry in the header must be updated.
- ☐ Pointers should not point directly to record — instead they should point to the entry for the record in header.

# Organization of Records in Files

- ❑ **Heap** – a record can be placed anywhere in the file where there is space
- ❑ **Sequential** – store records in sequential order, based on the value of the search key of each record
- ❑ **Hashing** – a hash function computed on some attribute of each record; the result specifies in which block of the file the record should be placed
- ❑ Records of each relation may be stored in a separate file. In a **multitable clustering file organization** records of several different relations can be stored in the same file
  - ❑ Motivation: store related records on the same block to minimize I/O

# Sequential File Organization

- ☐ Suitable for applications that require sequential processing of the entire file
- ☐ The records in the file are ordered by a search-key

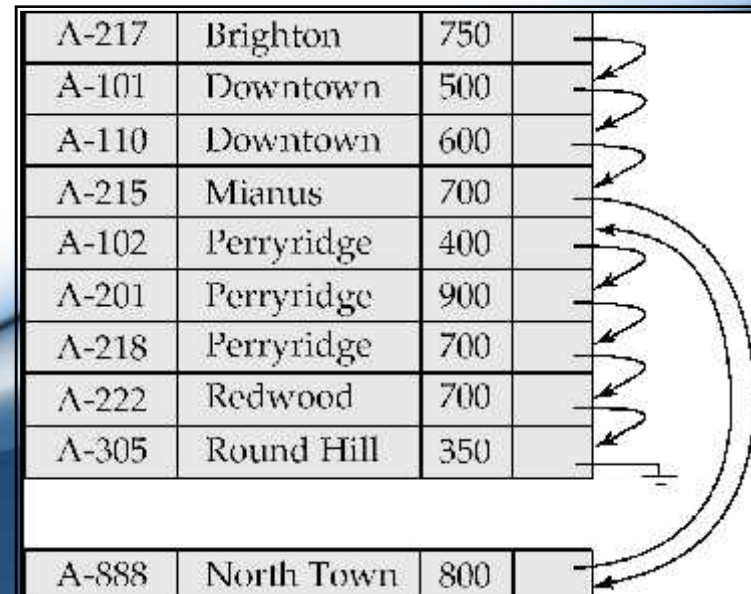
A-217	Brighton	750	
A-101	Downtown	500	
A-110	Downtown	600	
A-215	Mianus	700	
A-102	Perryridge	400	
A-201	Perryridge	900	
A-218	Perryridge	700	
A-222	Redwood	700	
A-305	Round Hill	350	



# Sequential File Organization (Cont.)

- Deletion – use pointer chains
- Insertion – locate the position where the record is to be inserted
  - if there is free space insert there
  - if no free space, insert the record in an overflow block
  - In either case, pointer chain must be updated
- Need to reorganize the file from time to time to restore sequential order

A-217	Brighton	750	
A-101	Downtown	500	
A-110	Downtown	600	
A-215	Mianus	700	
A-102	Perryridge	400	
A-201	Perryridge	900	
A-218	Perryridge	700	
A-222	Redwood	700	
A-305	Round Hill	350	
A-888	North Town	800	



# Multitable Clustering File Organization

Store several relations in one file using a **multitable clustering** file organization

<i>customer_name</i>	<i>account_number</i>
Hayes	A-102
Hayes	A-220
Hayes	A-503
Turner	A-305

<i>customer_name</i>	<i>customer_street</i>	<i>customer_city</i>
Hayes	Main	Brooklyn
Turner	Putnam	Stamford

## Multitable Clustering File Organization (cont.)

Multitable clustering organization of *customer* and *depositor*:

Hayes	Main	Brooklyn
Hayes	A-102	
Hayes	A-220	
Hayes	A-503	
Turner	Putnam	Stamford
Turner	A-305	

- good for queries involving *depositor* ⋈ *customer*, and for queries involving one single customer and his accounts
- bad for queries involving only *customer*
- results in variable size records
- Can add pointer chains to link records of a particular relation



# Data Dictionary Storage

Data dictionary (also called system catalog) stores metadata; that is, data about data, such as

- Information about relations
  - names of relations
  - names and types of attributes of each relation
  - names and definitions of views
  - integrity constraints
- User and accounting information, including passwords
- Statistical and descriptive data
  - number of tuples in each relation
- Physical file organization information
  - How relation is stored (sequential/hash/...)
  - Physical location of relation
- Information about indices



# Data Dictionary Storage (Cont.)

- Catalog structure
  - Relational representation on disk
  - specialized data structures designed for efficient access, in memory
- A possible catalog representation:

$$Relation\_metadata = (\underbrace{relation\_name, number\_of\_attributes,}_{storage\_organization, location})$$
$$\text{Attribute\_metadata} = (\text{attribute\_name}, \text{relation\_name}, \text{domain\_type}, \text{position}, \text{length})$$

```
User_metadata = (user_name, encrypted_password, group)
```

```
Index_metadata = (index_name, relation_name, index_type,
                  index_attributes)
```

```
View_metadata = (view_name, definition)
```

# Indexing and Hashing

- Basic Concepts
- Ordered Indices
- B<sup>+</sup>-Tree Index Files
- B-Tree Index Files
- Static Hashing
- Dynamic Hashing
- Comparison of Ordered Indexing and Hashing
- Index Definition in SQL
- Multiple-Key Access

# Basic Concepts

- Indexing mechanisms used to speed up access to desired data.
  - E.g., author catalog in library
- **Search Key** - attribute to set of attributes used to look up records in a file.
- An **index file** consists of records (called **index entries**) of the form

Index files are typically much smaller than the original file

- Two basic kinds of indexes:

search-key

pointer

- **Ordered indices:** search keys are stored in sorted order
- **Hash indices:** search keys are distributed uniformly across “buckets” using a “hash function”.

# Index Evaluation Metrics

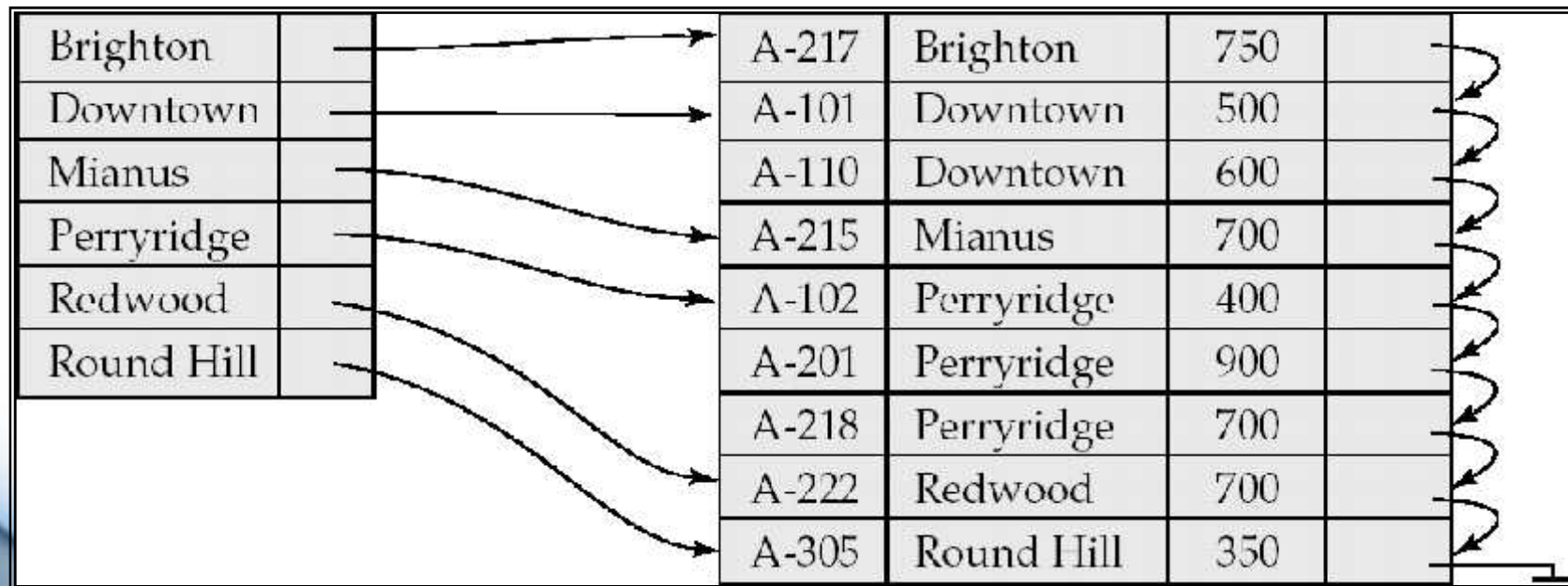
- Access types supported efficiently. E.g.,
  - records with a specified value in the attribute
  - or records with an attribute value falling in a specified range of values.
- Access time
- Insertion time
- Deletion time
- Space overhead

# Ordered Indices

- In an **ordered index**, index entries are stored sorted on the search key value. E.g., author catalog in library.
- **Primary index**: in a sequentially ordered file, the index whose search key specifies the sequential order of the file.
  - Also called **clustering index**
  - The search key of a primary index is usually but not necessarily the primary key.
- **Secondary index**: an index whose search key specifies an order different from the sequential order of the file. Also called non-clustering index.
- Index-sequential file: ordered sequential file with a primary index.

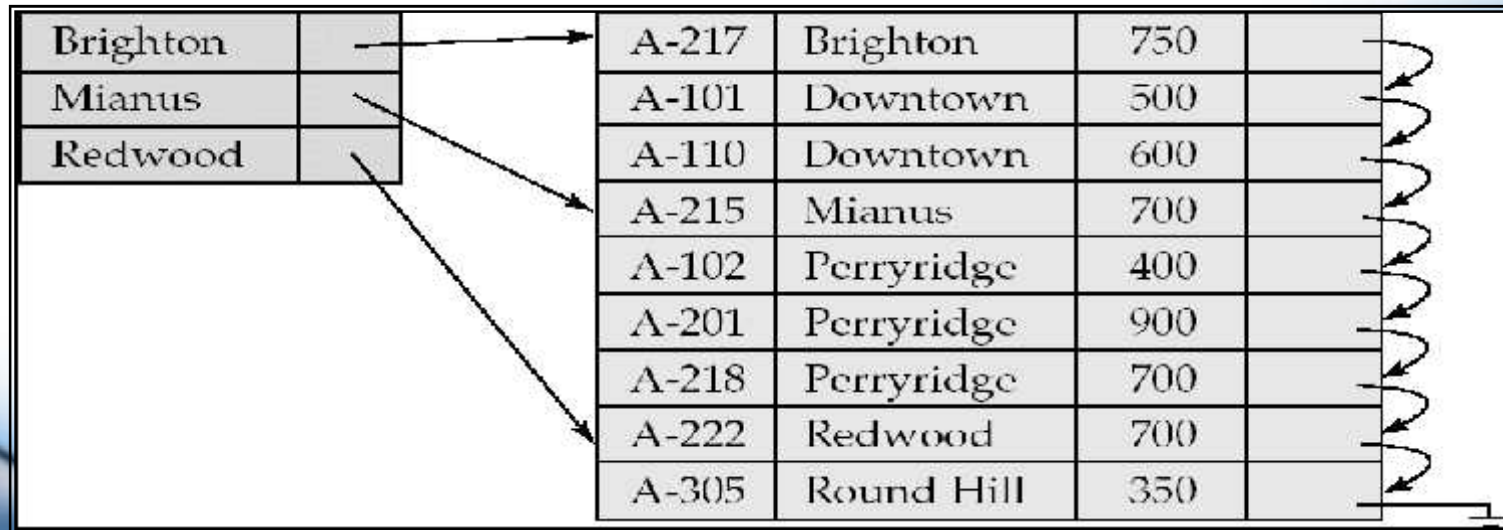
# Dense Index Files

- Dense index — Index record appears for every search-key value in the file.



# Sparse Index Files

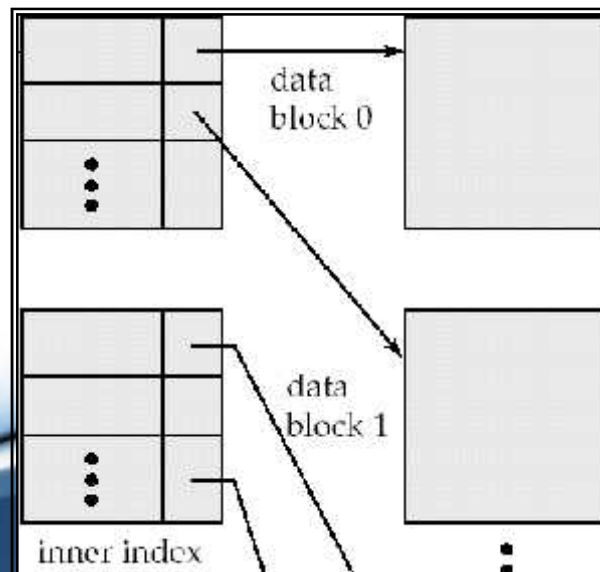
- Sparse Index: contains index records for only some search-key values.
  - Applicable when records are sequentially ordered on search-key
- To locate a record with search-key value  $K$  we:
  - Find index record with largest search-key value  $< K$
  - Search file sequentially starting at the record to which the index record points





# Sparse Index Files (Cont.)

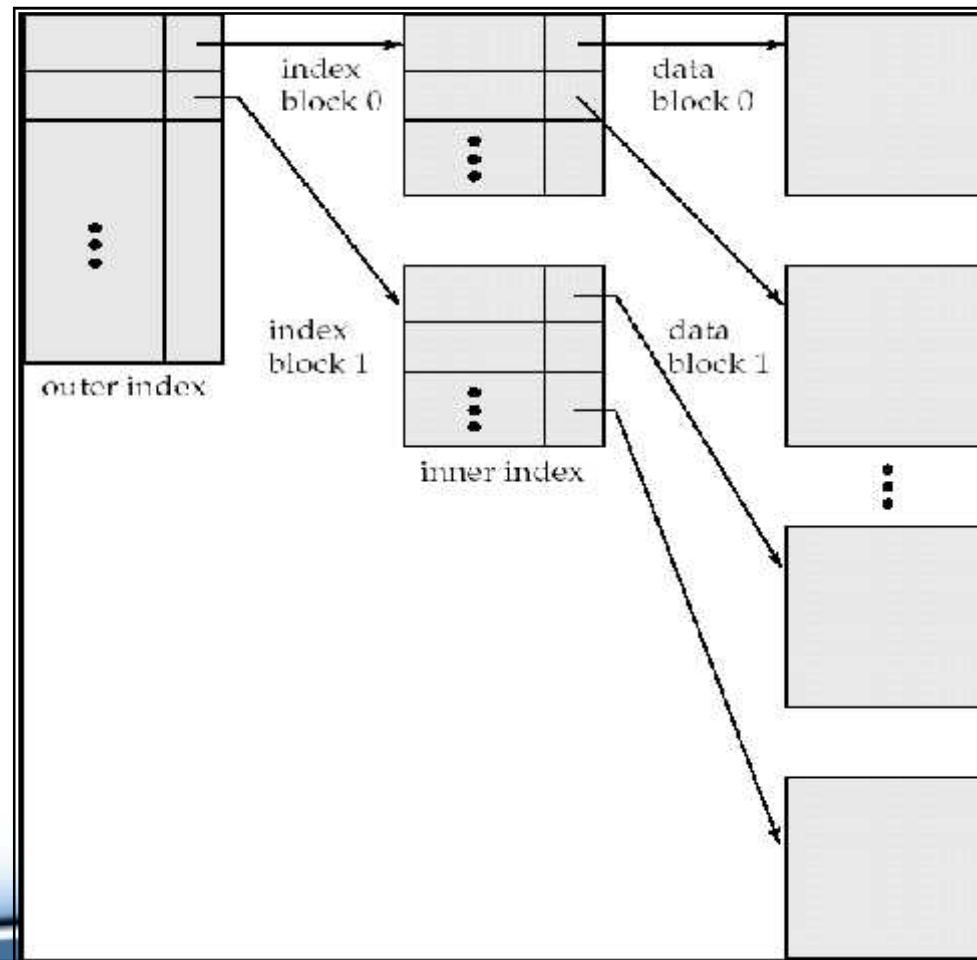
- Compared to dense indices:
  - Less space and less maintenance overhead for insertions and deletions.
  - Generally slower than dense index for locating records.
- **Good tradeoff:** sparse index with an index entry for every block in file, corresponding to least search-key value in the block.



# Multilevel Index

- If primary index does not fit in memory, access becomes expensive.
- Solution: treat primary index kept on disk as a sequential file and construct a sparse index on it.
  - outer index – a sparse index of primary index
  - inner index – the primary index file
- If even outer index is too large to fit in main memory, yet another level of index can be created, and so on.
- Indices at all levels must be updated on insertion or deletion from the file.

## Multilevel Index (Cont.)



# Index Update: Deletion

- If deleted record was the only record in the file with its particular search-key value, the search-key is deleted from the index also.
- Single-level index deletion:
  - **Dense indices** – deletion of search-key: similar to file record deletion.
  - **Sparse indices** –
    - if an entry for the search key exists in the index, it is deleted by replacing the entry in the index with the next search-key value in the file (in search-key order).
    - If the next search-key value already has an index entry, the entry is deleted instead of being replaced.

Brighton		→	A-217	Brighton	750	
Mianus		→	A-101	Downtown	500	
Redwood		→	A-110	Downtown	600	
			A-215	Mianus	700	
			A-102	Perryridge	400	
			A-201	Perryridge	900	
			A-218	Perryridge	700	
			A-222	Redwood	700	
			A-305	Round Hill	350	

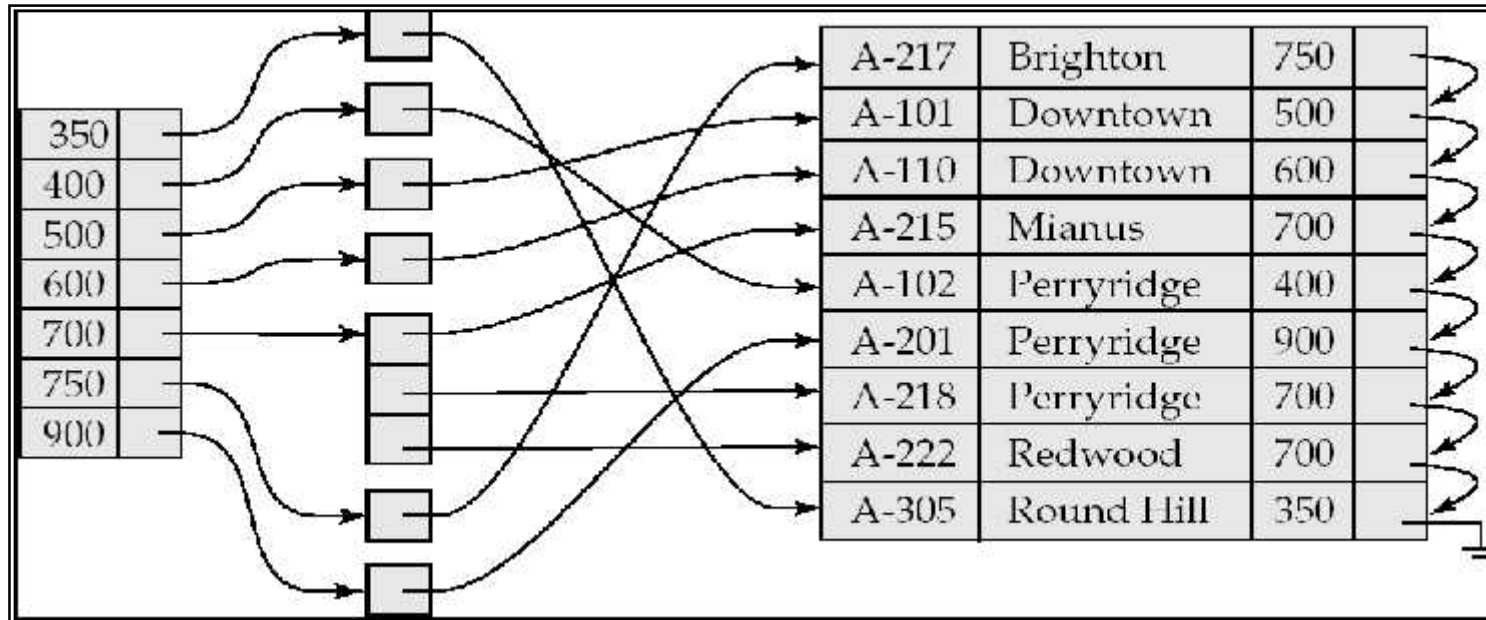
# Index Update: Insertion

- Single-level index insertion:
  - Perform a lookup using the search-key value appearing in the record to be inserted.
  - **Dense indices** – if the search-key value does not appear in the index, insert it.
  - **Sparse indices** – if index stores an entry for each block of the file, no change needs to be made to the index unless a new block is created.
    - If a new block is created, the first search-key value appearing in the new block is inserted into the index.
- Multilevel insertion (as well as deletion) algorithms are simple extensions of the single-level algorithms

# Secondary Indices

- Frequently, one wants to find all the records whose values in a certain field (which is not the search-key of the primary index) satisfy some condition.
  - Example 1: In the *account* relation stored sequentially by account number, we may want to find all accounts in a particular branch
  - Example 2: as above, but where we want to find all accounts with a specified balance or range of balances
- We can have a secondary index with an index record for each search-key value

# Secondary Indices Example



## Secondary index on *balance* field of *account*

- Index record points to a bucket that contains pointers to all the actual records with that particular search-key value.
- Secondary indices have to be dense



# Primary and Secondary Indices

- Indices offer substantial benefits when searching for records.
- BUT: Updating indices imposes overhead on database modification --when a file is modified, every index on the file must be updated,
- Sequential scan using primary index is efficient, but a sequential scan using a secondary index is expensive
  - Each record access may fetch a new block from disk
  - Block fetch requires about 5 to 10 milliseconds
    - versus about 100 nanoseconds for memory access

# B<sup>+</sup>-Tree Index Files

B<sup>+</sup>-tree indices are an alternative to indexed-sequential files.

- Disadvantage of indexed-sequential files
  - performance degrades as file grows, since many overflow blocks get created.
  - Periodic reorganization of entire file is required.
- Advantage of B<sup>+</sup>-tree index files:
  - automatically reorganizes itself with small, local, changes, in the face of insertions and deletions.
  - Reorganization of entire file is not required to maintain performance.
- (Minor) disadvantage of B<sup>+</sup>-trees:
  - extra insertion and deletion overhead, space overhead.
- Advantages of B<sup>+</sup>-trees outweigh disadvantages
  - B<sup>+</sup>-trees are used extensively

# B<sup>+</sup>-Tree Index Files (Cont.)

A B<sup>+</sup>-tree is a rooted tree satisfying the following properties:

- All paths from root to leaf are of the same length
- Each node that is not a root or a leaf has between  $\lceil n/2 \rceil$  and  $n$  children.
- A leaf node has between  $\lceil (n-1)/2 \rceil$  and  $n-1$  values
- Special cases:
  - If the root is not a leaf, it has at least 2 children.
  - If the root is a leaf (that is, there are no other nodes in the tree), it can have between 0 and  $(n-1)$  values.

# B<sup>+</sup>-Tree Node Structure

- Typical node



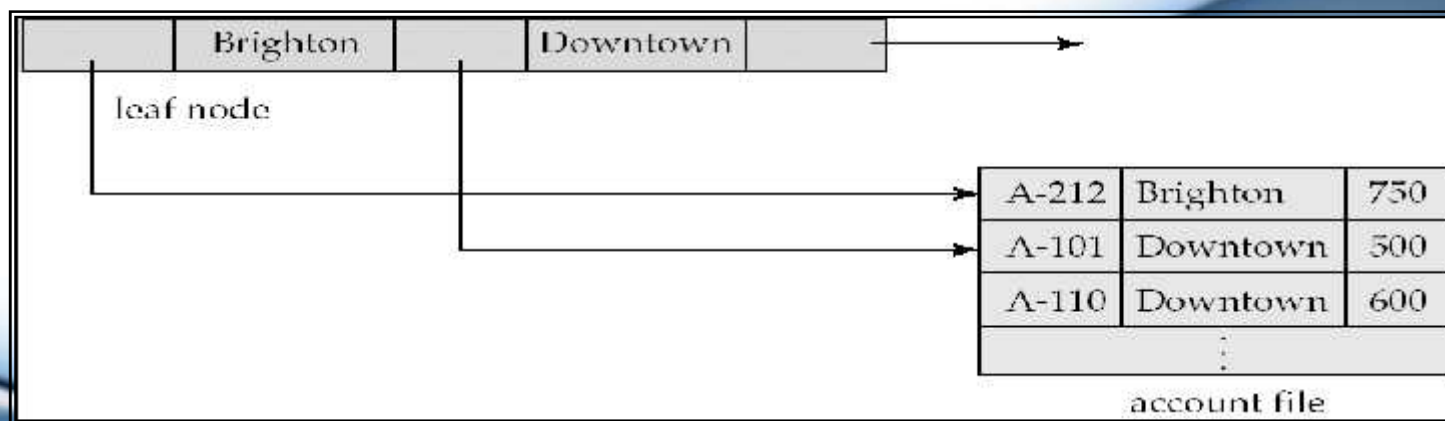
- $K_i$  are the search-key values
  - $P_i$  are pointers to children (for non-leaf nodes) or pointers to records or buckets of records (for leaf nodes).
- The search-keys in a node are ordered

$$K_1 < K_2 < K_3 < \dots < K_{n-1}$$

# Leaf Nodes in B<sup>+</sup>-Trees

Properties of a leaf node:

- For  $i = 1, 2, \dots, n-1$ , pointer  $P_i$  either points to a file record with search-key value  $K_i$ , or to a bucket of pointers to file records, each record having search-key value  $K_i$ . Only need bucket structure if search-key does not form a primary key.
- If  $L_i, L_j$  are leaf nodes and  $i < j$ ,  $L_i$ 's search-key values are less than  $L_j$ 's search-key values
- $P_n$  points to next leaf node in search-key order

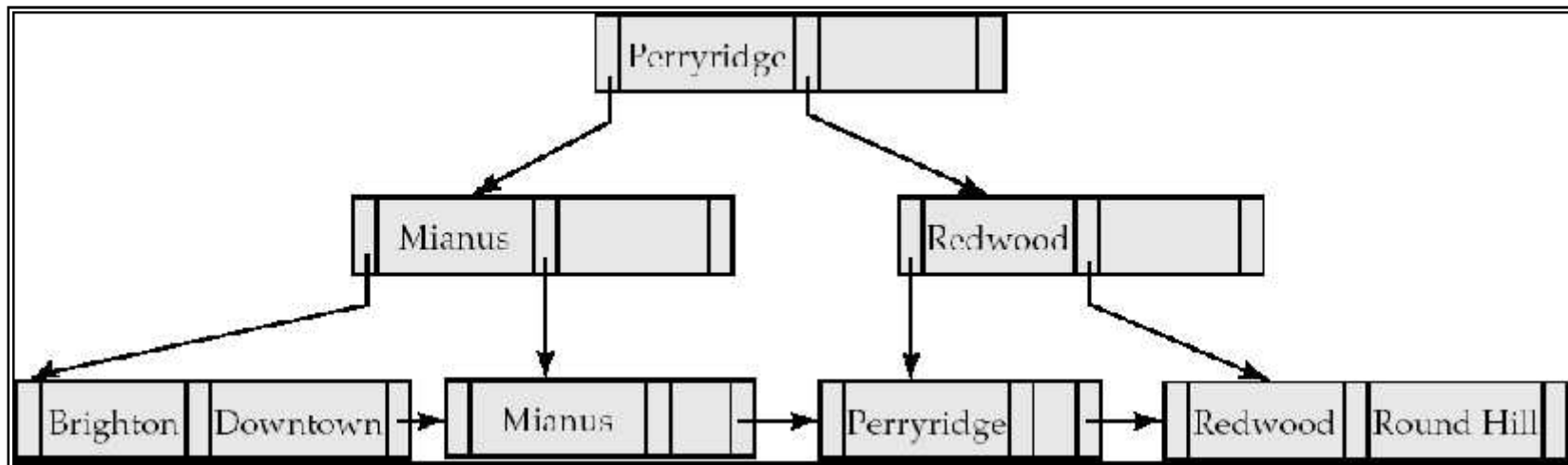


# Non-Leaf Nodes in B<sup>+</sup>-Trees

- Non leaf nodes form a multi-level sparse index on the leaf nodes. For a non-leaf node with  $m$  pointers:
  - All the search-keys in the subtree to which  $P_1$  points are less than  $K_1$
  - For  $2 \leq i \leq n - 1$ , all the search-keys in the subtree to which  $P_i$  points have values greater than or equal to  $K_{i-1}$  and less than  $K_i$
  - All the search-keys in the subtree to which  $P_n$  points have values greater than or equal to  $K_{n-1}$



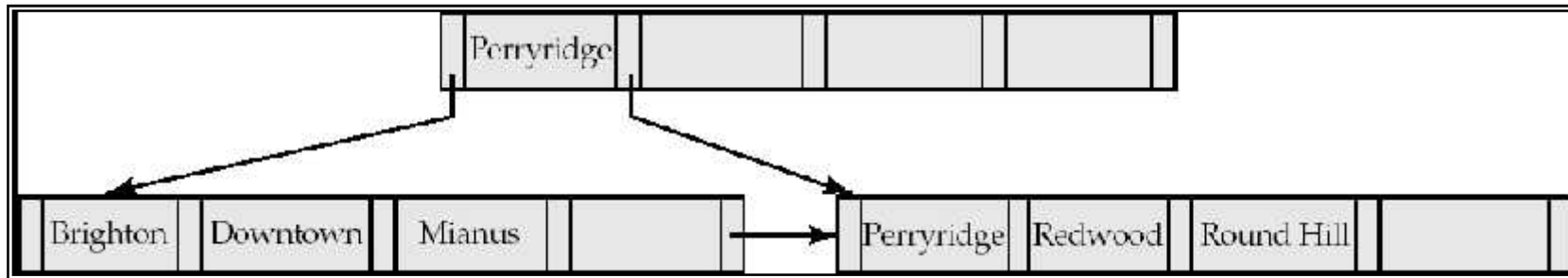
# Example of a B<sup>+</sup>-tree



B<sup>+</sup>-tree for *account* file ( $n = 3$ )



# Example of B<sup>+</sup>-tree



B<sup>+</sup>-tree for *account* file ( $n = 5$ )

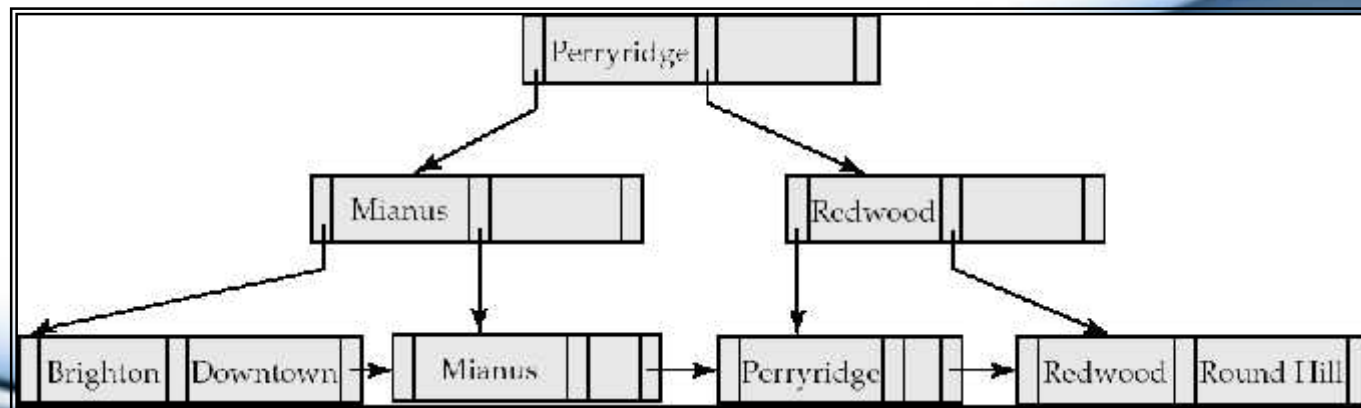
- Leaf nodes must have between 2 and 4 values ( $\lceil (n-1)/2 \rceil$  and  $n-1$ , with  $n = 5$ ).
- Non-leaf nodes other than root must have between 3 and 5 children ( $\lceil n/2 \rceil$  and  $n$  with  $n = 5$ ).
- Root must have at least 2 children.

# Observations about B<sup>+</sup>-trees

- Since the inter-node connections are done by pointers, “logically” close blocks need not be “physically” close.
- The non-leaf levels of the B<sup>+</sup>-tree form a hierarchy of sparse indices.
- The B<sup>+</sup>-tree contains a relatively small number of levels
  - Level below root has at least  $2 \cdot \lceil n/2 \rceil$  values
  - Next level has at least  $2 \cdot \lceil n/2 \rceil \cdot \lceil n/2 \rceil$  values
  - .. etc.
  - If there are  $K$  search-key values in the file, the tree height is no more than  $\lceil \log_{\lceil n/2 \rceil}(K) \rceil$
  - thus searches can be conducted efficiently.
- Insertions and deletions to the main file can be handled efficiently, as the index can be restructured in logarithmic time (as we shall see).

# Queries on B<sup>+</sup>-Trees

- Find all records with a search-key value of  $k$ .
  - $N = \text{root}$
  - Repeat
    - Examine  $N$  for the smallest search-key value  $> k$ .
    - If such a value exists, assume it is  $K_i$ . Then set  $N = P_i$
    - Otherwise  $k \geq K_{n-1}$ . Set  $N = P_n$Until  $N$  is a leaf node
  - If for some  $i$ , key  $K_i = k$  follow pointer  $P_i$  to the desired record or bucket.
  - Else no record with search-key value  $k$  exists.



# Queries on B<sup>+</sup>-Trees (Cont.)

- If there are  $K$  search-key values in the file, the height of the tree is no more than  $\lceil \log_{\lceil n/2 \rceil}(K) \rceil$ .
- A node is generally the same size as a disk block, typically 4 kilobytes
  - and  $n$  is typically around 100 (40 bytes per index entry).
- With 1 million search key values and  $n = 100$ 
  - at most  $\log_{50}(1,000,000) = 4$  nodes are accessed in a lookup.
- Contrast this with a balanced binary tree with 1 million search key values — around 20 nodes are accessed in a lookup
  - above difference is significant since every node access may need a disk I/O, costing around 20 milliseconds

# Updates on B<sup>+</sup>-Trees: Insertion

1. Find the leaf node in which the search-key value would appear
2. If the search-key value is already present in the leaf node
  1. Add record to the file
  2. If necessary add a pointer to the bucket.
3. If the search-key value is not present, then
  1. add the record to the main file (and create a bucket if necessary)
  2. If there is room in the leaf node, insert (key-value, pointer) pair in the leaf node
  3. Otherwise, split the node (along with the new (key-value, pointer) entry) as discussed in the next slide.

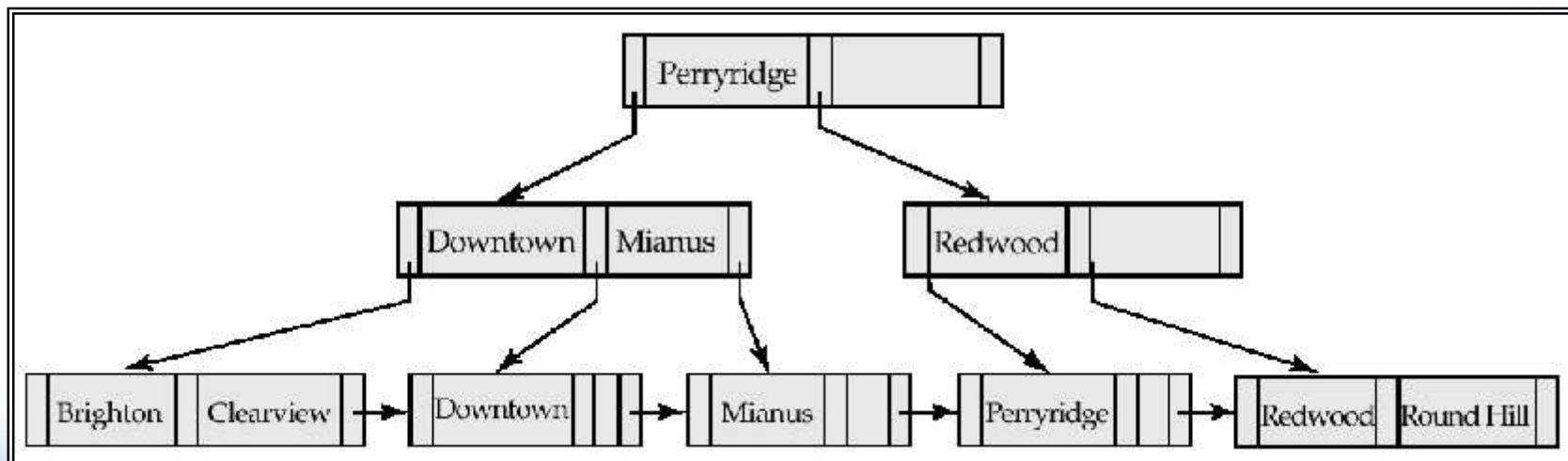
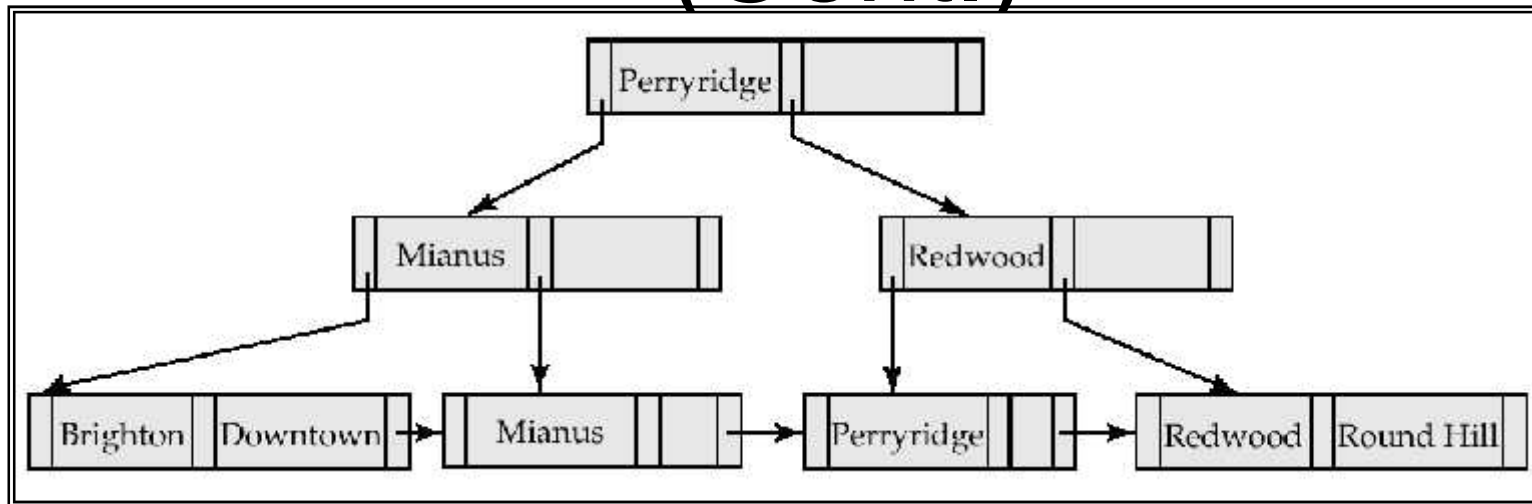
# Updates on B<sup>+</sup>-Trees: Insertion (Cont.)

- Splitting a leaf node:
  - take the  $n$  (search-key value, pointer) pairs (including the one being inserted) in sorted order. Place the first  $\lceil n/2 \rceil$  in the original node, and the rest in a new node.
  - let the new node be  $p$ , and let  $k$  be the least key value in  $p$ . Insert  $(k, p)$  in the parent of the node being split.
  - If the parent is full, split it and **propagate** the split further up.
- Splitting of nodes proceeds upwards till a node that is not full is found.
  - In the worst case the root node may be split increasing the height of the tree by 1.

Result of splitting node containing Brighton and Downtown on inserting Clearview

Next step: insert entry with (Downtown, pointer-to-new-node) into parent

# UPDATES ON B-TREES. INSERTION (Cont.)

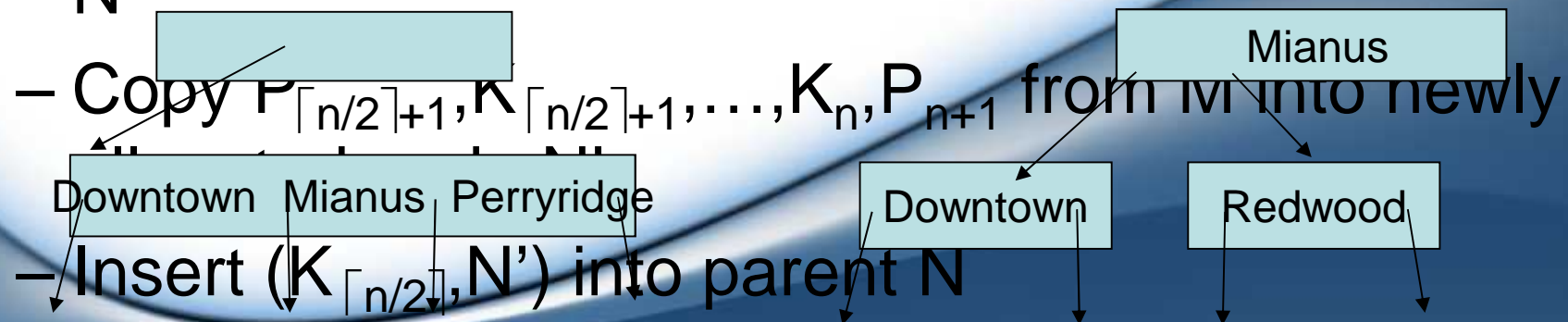


B<sup>+</sup>-Tree before and after insertion of "Clearview"



# Insertion in B<sup>+</sup>-Trees (Cont.)

- Splitting a non-leaf node: when inserting (k,p) into an already full internal node N
  - Copy N to an in-memory area M with space for n+1 pointers and n keys
  - Insert (k,p) into M
  - Copy  $P_1, K_1, \dots, K_{\lceil n/2 \rceil - 1}, P_{\lceil n/2 \rceil}$  from M back into node N



- **Read pseudocode in book!**

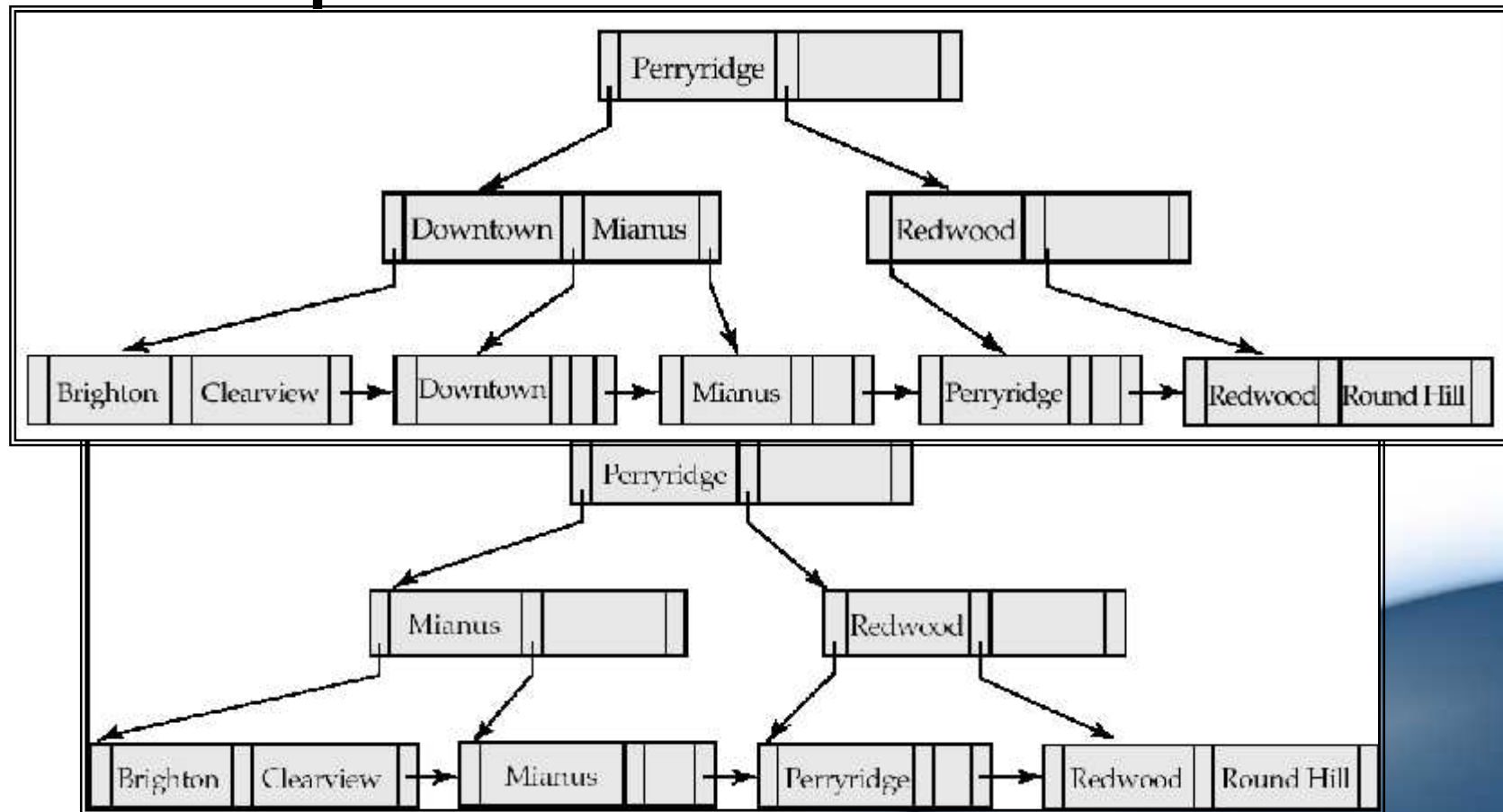
# Updates on B<sup>+</sup>-Trees: Deletion

- Find the record to be deleted, and remove it from the main file and from the bucket (if present)
- Remove (search-key value, pointer) from the leaf node if there is no bucket or if the bucket has become empty
- If the node has too few entries due to the removal, and the entries in the node and a sibling fit into a single node, then ***merge siblings:***
  - Insert all the search-key values in the two nodes into a single node (the one on the left), and delete the other node.
  - Delete the pair  $(K_{i-1}, P_i)$ , where  $P_i$  is the pointer to the deleted node, from its parent, recursively using the above procedure.

# Updates on B<sup>+</sup>-Trees: Deletion

- Otherwise, if the node has too few entries due to the removal, but the entries in the node and a sibling do not fit into a single node, then **redistribute pointers**:
  - Redistribute the pointers between the node and a sibling such that both have more than the minimum number of entries.
  - Update the corresponding search-key value in the parent of the node.
- The node deletions may cascade upwards till a node which has  $\lceil n/2 \rceil$  or more pointers is found.
- If the root node has only one pointer after deletion, it is deleted and the sole child becomes the root.

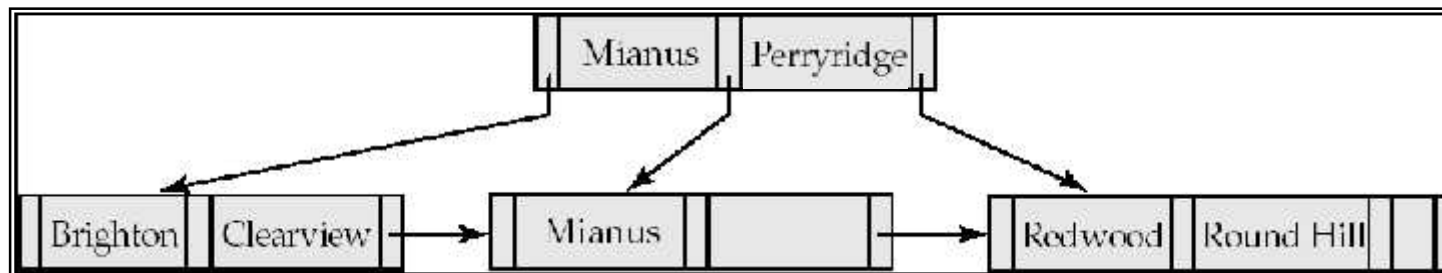
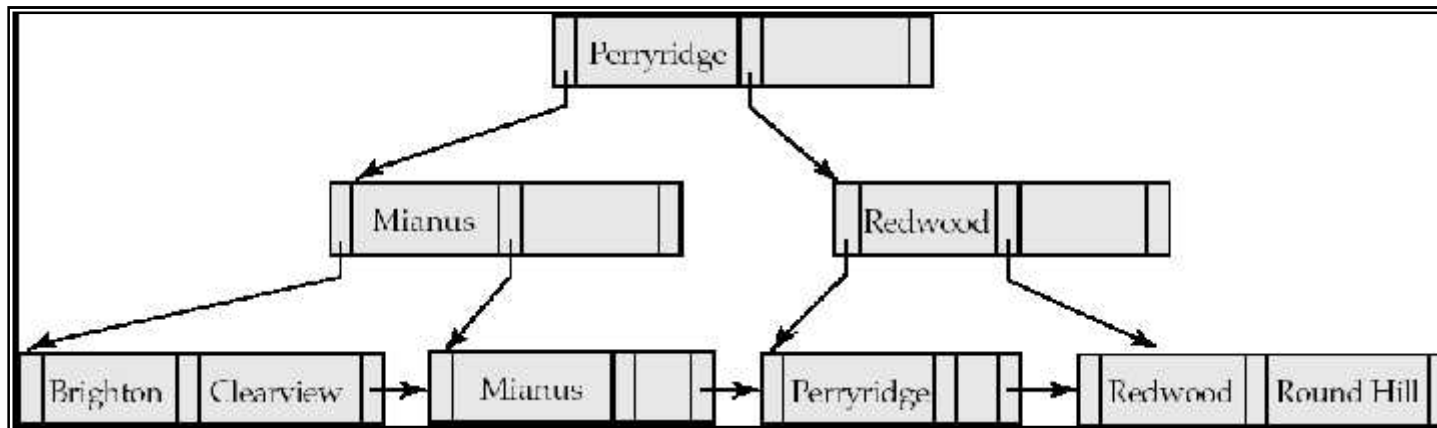
# Examples of B<sup>+</sup>-Tree Deletion



Before and after deleting "Downtown"

- Deleting "Downtown" causes merging of under-full leaves
  - leaf node can become empty only for  $n=3$ !

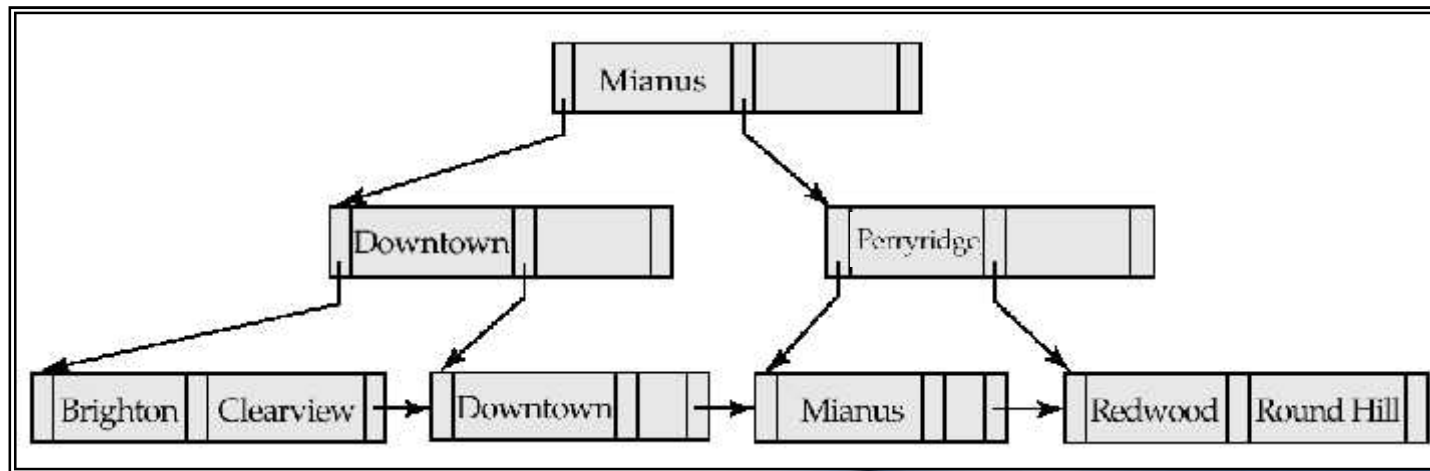
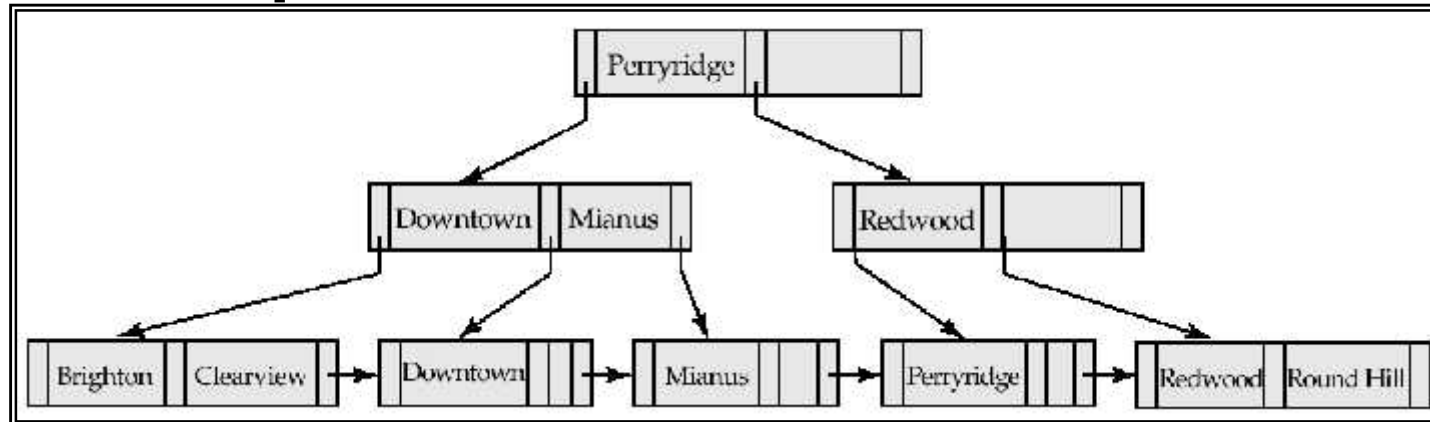
# Examples of B<sup>+</sup>-Tree Deletion



Deletion of "Perryridge" from result of previous example

- Leaf with "Perryridge" becomes underfull (actually empty, in this special case) and merged with its sibling.
- As a result "Perryridge" node's parent became underfull, and was merged with its sibling
  - Value separating two nodes (at parent) moves into merged node
  - Entry deleted from parent
- Root node then has only one child, and is deleted

# Example of B<sup>+</sup>-tree Deletion



Before and after deletion of “Perryridge” from earlier example

- Parent of leaf containing Perryridge became underfull, and borrowed a pointer from its left sibling
- Search-key value in the parent's parent changes as a result

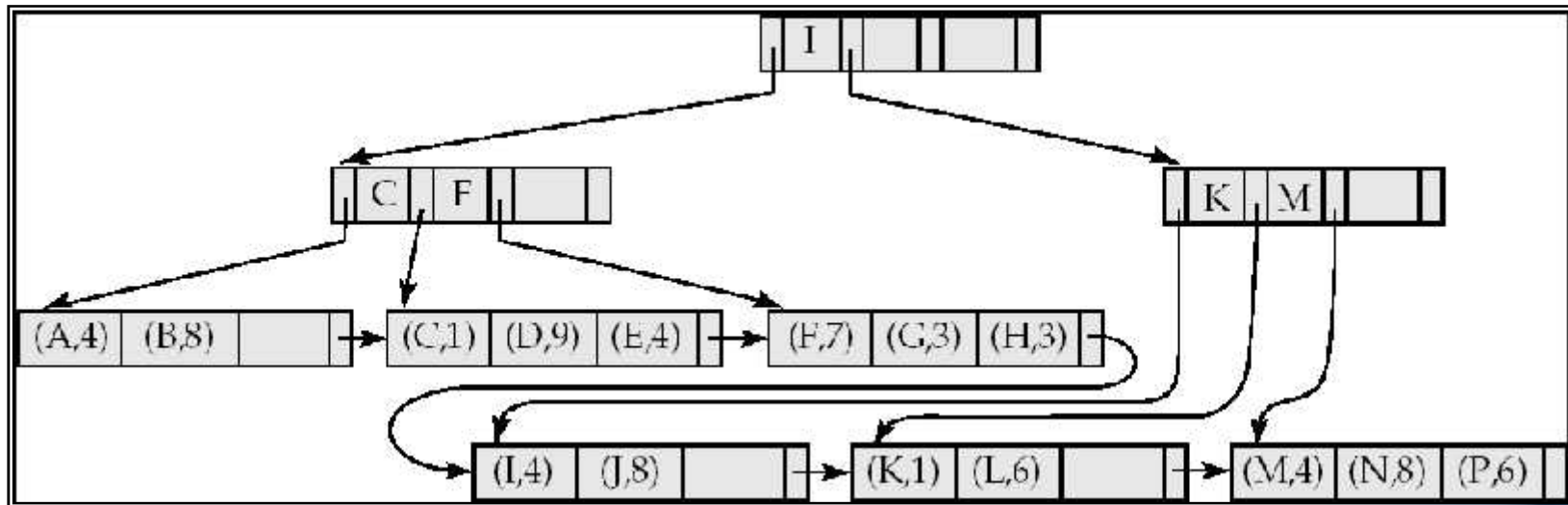


# B<sup>+</sup>-Tree File Organization

- Index file degradation problem is solved by using B<sup>+</sup>-Tree indices.
- Data file degradation problem is solved by using B<sup>+</sup>-Tree File Organization.
- The leaf nodes in a B<sup>+</sup>-tree file organization store records, instead of pointers.
- Leaf nodes are still required to be half full
  - Since records are larger than pointers, the maximum number of records that can be stored in a leaf node is less than the number of pointers in a nonleaf node.
- Insertion and deletion are handled in the same way as insertion and deletion of entries in a B<sup>+</sup>-tree index.



# B<sup>+</sup>-Tree File Organization (Cont.)



Example of B<sup>+</sup>-tree File Organization

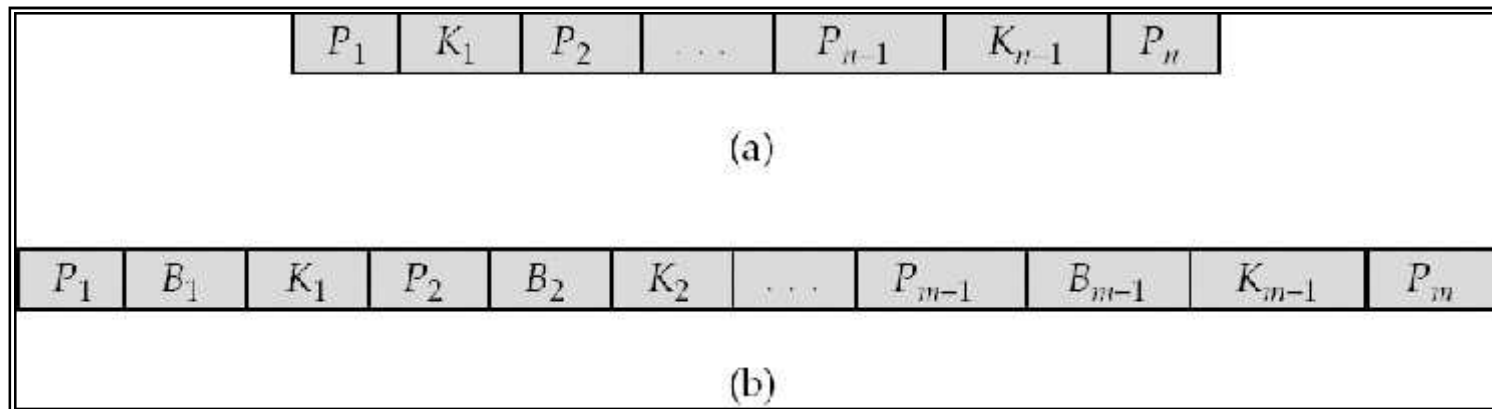
- Good space utilization important since records use more space than pointers.
- To improve space utilization, involve more sibling nodes in redistribution during splits and merges
  - Involving 2 siblings in redistribution (to avoid split / merge where possible) results in each node having at least  $\lfloor 2n/3 \rfloor$  entries

# Indexing Strings

- Variable length strings as keys
  - Variable fanout
  - Use space utilization as criterion for splitting, not number of pointers
- **Prefix compression**
  - Key values at internal nodes can be prefixes of full key
    - Keep enough characters to distinguish entries in the subtrees separated by the key value
      - E.g. “Silas” and “Silberschatz” can be separated by “Silb”
  - Keys in leaf node can be compressed by sharing common prefixes

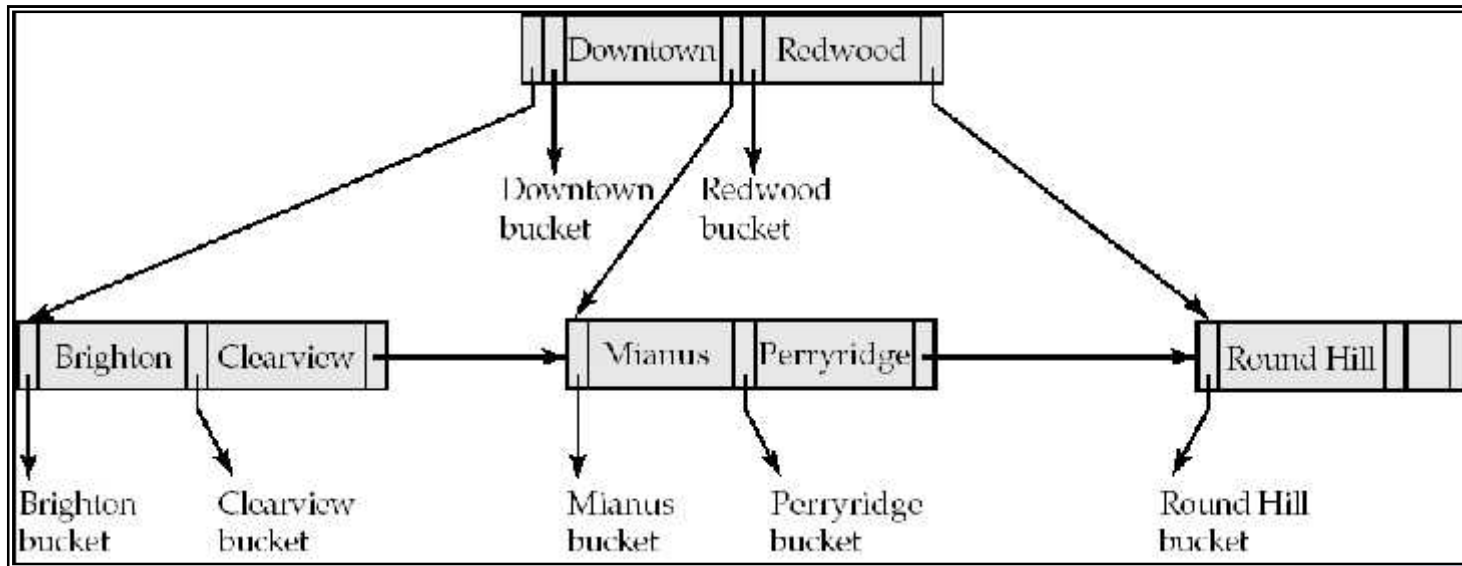
# B-Tree Index Files

- Similar to B+-tree, but B-tree allows search-key values to appear only once; eliminates redundant storage of search keys.
- Search keys in nonleaf nodes appear nowhere else in the B-tree; an additional pointer field for each search key in a nonleaf node must be included.
- Generalized B-tree leaf node

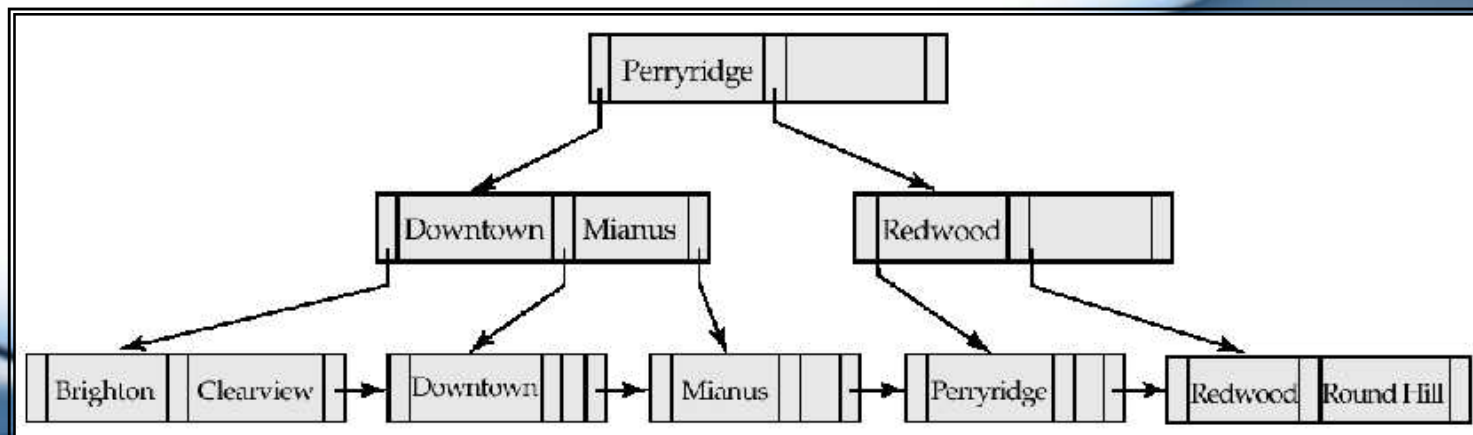


- Nonleaf node – pointers  $B_i$  are the bucket or file record pointers.

# B-Tree Index File Example



B-tree (above) and B+-tree (below) on same data



# B-Tree Index Files (Cont.)

- Advantages of B-Tree indices:
  - May use less tree nodes than a corresponding B<sup>+</sup>-Tree.
  - Sometimes possible to find search-key value before reaching leaf node.
- Disadvantages of B-Tree indices:
  - Only small fraction of all search-key values are found early
  - Non-leaf nodes are larger, so fan-out is reduced. Thus, B-Trees typically have greater depth than corresponding B<sup>+</sup>-Tree
  - Insertion and deletion more complicated than in B<sup>+</sup>-Trees
  - Implementation is harder than B<sup>+</sup>-Trees.
- Typically, advantages of B-Trees do not outweigh disadvantages.

# Multiple-Key Access

- Use multiple indices for certain types of queries.
- Example:  
**select** *account\_number*  
**from** *account*  
**where** *branch\_name* = "Perryridge" **and** *balance* = 1000
- Possible strategies for processing query using indices on single attributes:
  1. Use index on *branch\_name* to find accounts with branch name Perryridge; test *balance* = 1000
  2. Use index on *balance* to find accounts with balances of \$1000; test *branch\_name* = "Perryridge".
  3. Use *branch\_name* index to find pointers to all records pertaining to the Perryridge branch. Similarly use index on *balance*. Take intersection of both sets of pointers obtained.

# Indices on Multiple Keys

- **Composite search keys** are search keys containing more than one attribute
  - E.g. (*branch\_name*, *balance*)
- Lexicographic ordering:  $(a_1, a_2) < (b_1, b_2)$  if either
  - $a_1 < b_1$ , or
  - $a_1 = b_1$  and  $a_2 < b_2$



# Indices on Multiple Attributes

Suppose we have an index on combined search-key  
(*branch\_name*, *balance*).

- With the **where** clause  
    **where** *branch\_name* = "Perryridge" **and** *balance* = 1000  
the index on (*branch\_name*, *balance*) can be used to fetch only records that satisfy both conditions.
  - Using separate indices is less efficient — we may fetch many records (or pointers) that satisfy only one of the conditions.
- Can also efficiently handle  
    **where** *branch\_name* = "Perryridge" **and** *balance* < 1000
- But cannot efficiently handle  
    **where** *branch\_name* < "Perryridge" **and** *balance* = 1000
  - May fetch many records that satisfy the first but not the second condition

# Non-Unique Search Keys

- Alternatives:
  - Buckets on separate block (bad idea)
  - List of tuple pointers with each key
    - Extra code to handle long lists
    - Deletion of a tuple can be expensive if there are many duplicates on search key (why?)
    - Low space overhead, no extra cost for queries
  - Make search key unique by adding a record-identifier
    - Extra storage overhead for keys
    - Simpler code for insertion/deletion
    - Widely used

# Other Issues in Indexing

- **Covering indices**
  - Add extra attributes to index so (some) queries can avoid fetching the actual records
    - Particularly useful for secondary indices
      - Why?
  - Can store extra attributes only at leaf
- Record relocation and secondary indices
  - If a record moves, all secondary indices that store record pointers have to be updated
  - Node splits in B<sup>+</sup>-tree file organizations become very expensive
  - *Solution:* use primary-index search key instead of record pointer in secondary index
    - Extra traversal of primary index to locate record
      - Higher cost for queries, but node splits are cheap
    - Add record-id if primary-index search key is non-unique

# Hashing

# Static Hashing

- A **bucket** is a unit of storage containing one or more records (a bucket is typically a disk block).
- In a **hash file organization** we obtain the bucket of a record directly from its search-key value using a **hash function**.
- Hash function  $h$  is a function from the set of all search-key values  $K$  to the set of all bucket addresses  $B$ .
- Hash function is used to locate records for access, insertion as well as deletion.
- Records with different search-key values may be mapped to the same bucket; thus entire bucket has to be searched sequentially to locate a record.

# Example of Hash File Organization

Hash file organization of *account* file, using *branch\_name* as key  
(See figure in next slide.)

- There are 10 buckets,
- The binary representation of the  $i$ th character is assumed to be the integer  $i$ .
- The hash function returns the sum of the binary representations of the characters modulo 10
  - E.g.  $h(\text{Perryridge}) = 5$      $h(\text{Round Hill}) = 3$   
 $h(\text{Brighton}) = 3$

# Example of Hash File Organization

Hash file organization of *account* file, using *branch\_name* as key (see previous slide for details).

bucket 0			bucket 5		
			A-102	Perryridge	400
			A-201	Perryridge	900
			A-218	Perryridge	700
bucket 1			bucket 6		
bucket 2			bucket 7		
			A-215	Mianus	700
bucket 3			bucket 8		
A-217	Brighton	750	A-101	Downtown	500
A-305	Round Hill	350	A-110	Downtown	600
bucket 4			bucket 9		
A-222	Redwood	700			



# Hash Functions

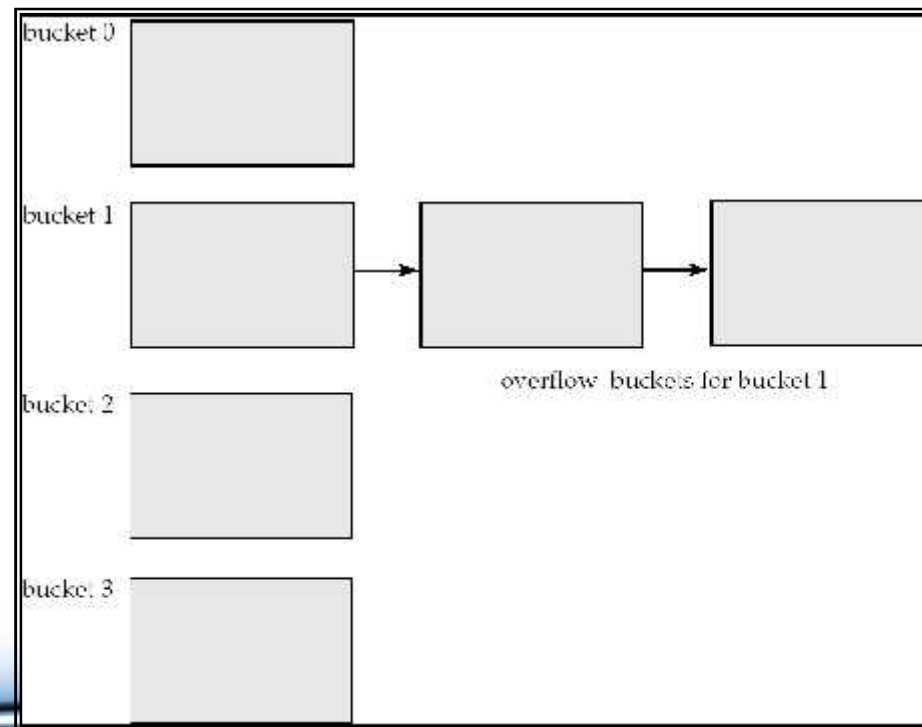
- Worst hash function maps all search-key values to the same bucket; this makes access time proportional to the number of search-key values in the file.
- An ideal hash function is **uniform**, i.e., each bucket is assigned the same number of search-key values from the set of *all* possible values.
- Ideal hash function is **random**, so each bucket will have the same number of records assigned to it irrespective of the *actual distribution* of search-key values in the file.
- Typical hash functions perform computation on the internal binary representation of the search-key.
  - For example, for a string search-key, the binary representations of all the characters in the string could be added and the sum modulo the number of buckets could be returned. .

# Handling of Bucket Overflows

- Bucket overflow can occur because of
  - Insufficient buckets
  - Skew in distribution of records. This can occur due to two reasons:
    - multiple records have same search-key value
    - chosen hash function produces non-uniform distribution of key values
- Although the probability of bucket overflow can be reduced, it cannot be eliminated; it is handled by using *overflow buckets*.

# MANAGING OF BUCKET OVERFLOWS (Cont.)

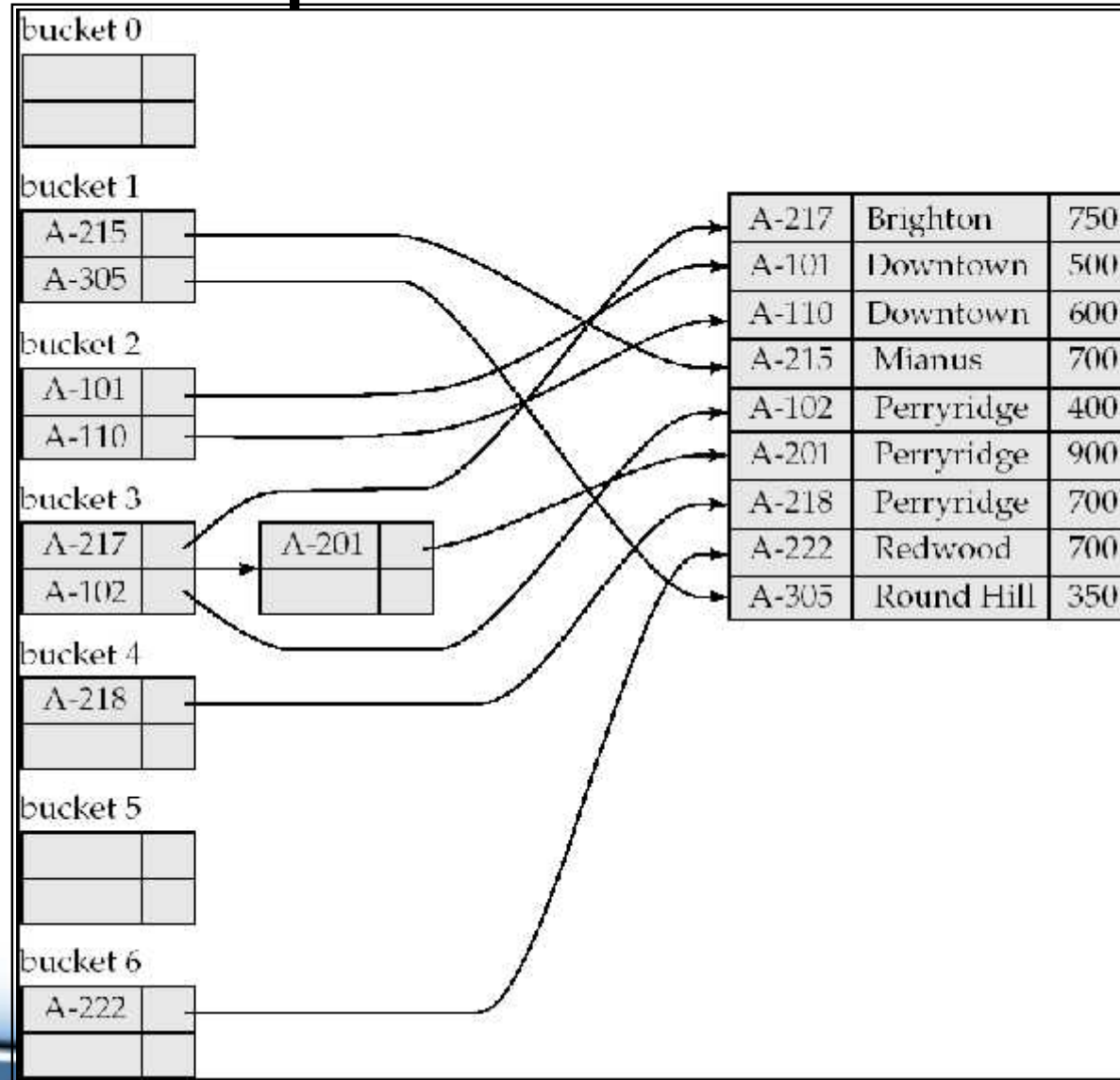
- Overflow chaining – the overflow buckets of a given bucket are chained together in a linked list.
- Above scheme is called closed hashing.
  - An alternative, called open hashing, which does not use overflow buckets, is not suitable for database applications.



# 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



# Deficiencies of Static Hashing

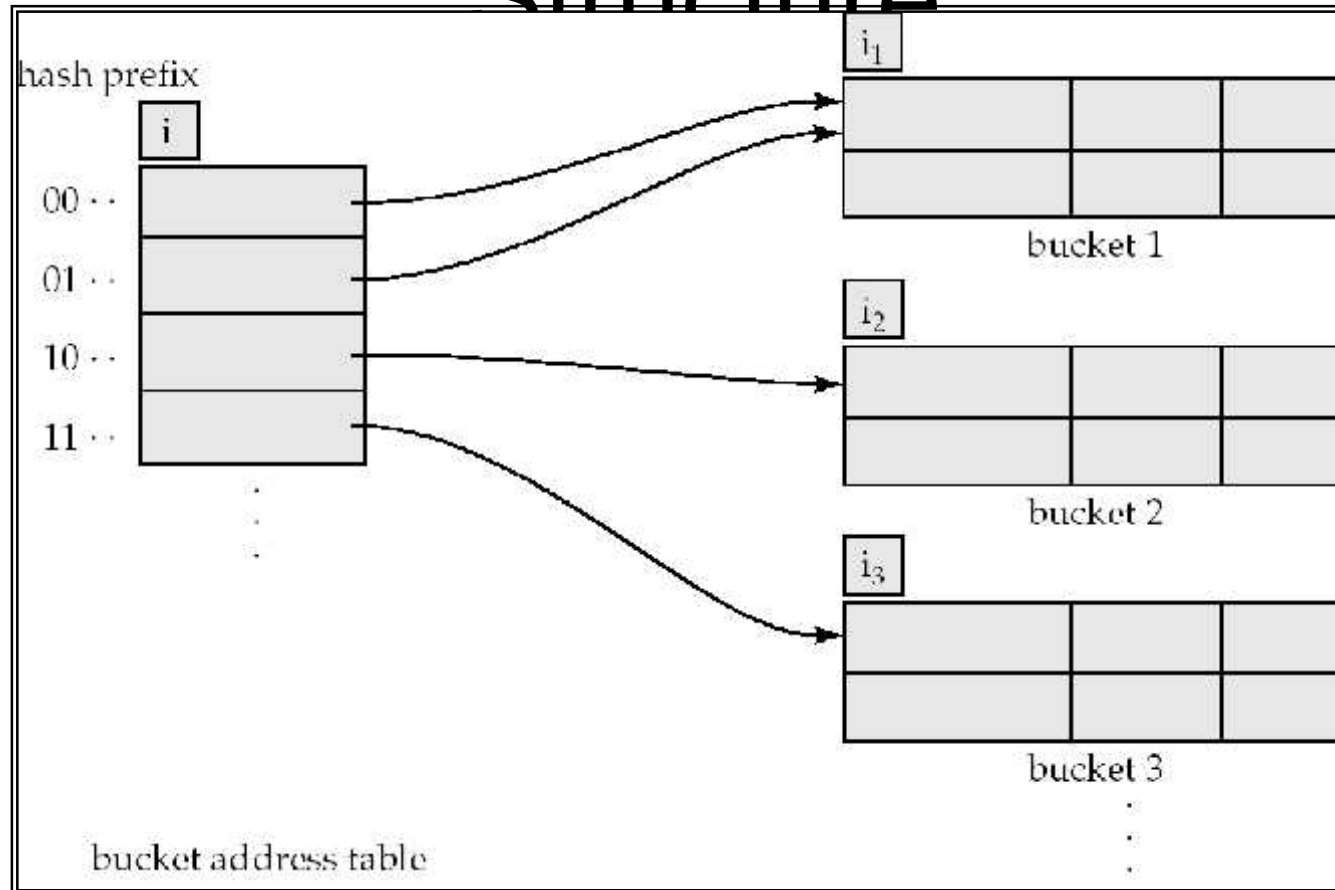
- In static hashing, function  $h$  maps search-key values to a fixed set of  $B$  of bucket addresses. Databases grow or shrink with time.
  - If initial number of buckets is too small, and file grows, performance will degrade due to too much overflows.
  - If space is allocated for anticipated growth, a significant amount of space will be wasted initially (and buckets will be underfull).
  - 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  $K_j$ :

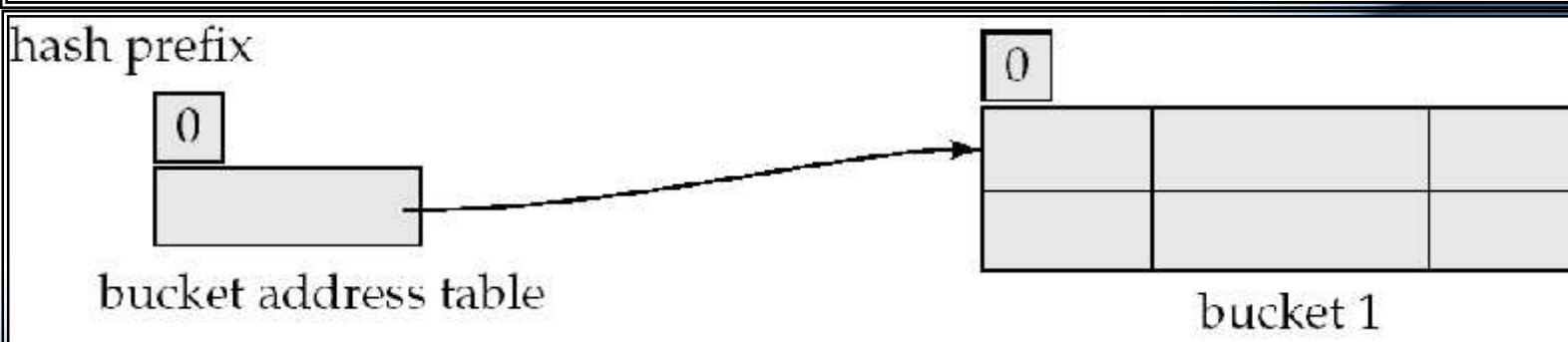
- If  $i > i_j$  (more than one pointer to bucket  $j$ )
  - allocate a new bucket  $z$ , and set  $i_j = i_z = (i_j + 1)$
  - Update the second half of the bucket address table entries originally pointing to  $j$ , to point to  $z$
  - remove each record in bucket  $j$  and reinsert (in  $j$  or  $z$ )
  - recompute 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

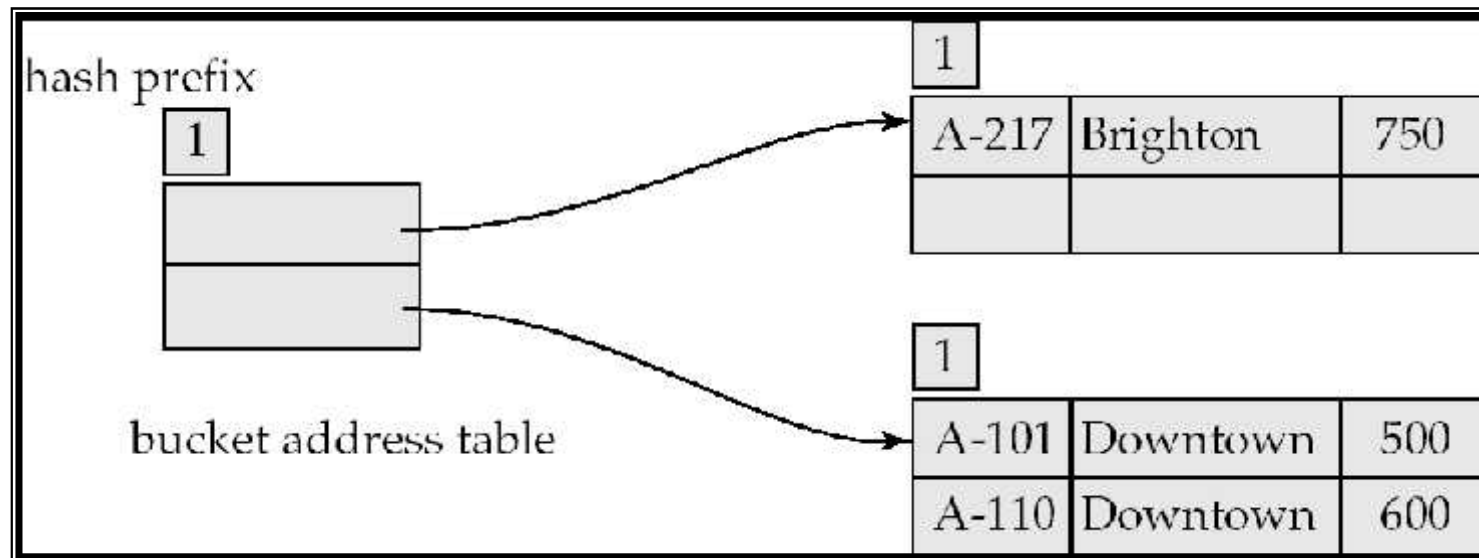
$branch\_name$	$h(branch\_name)$
Brighton	0010 1101 1111 1011 0010 1100 0011 0000
Downtown	1010 0011 1010 0000 1100 0110 1001 1111
Mianus	1100 0111 1110 1101 1011 1111 0011 1010
Perryridge	1111 0001 0010 0100 1001 0011 0110 1101
Redwood	0011 0101 1010 0110 1100 1001 1110 1011
Round Hill	1101 1000 0011 1111 1001 1100 0000 0001



## Initial Hash structure, bucket size = 2

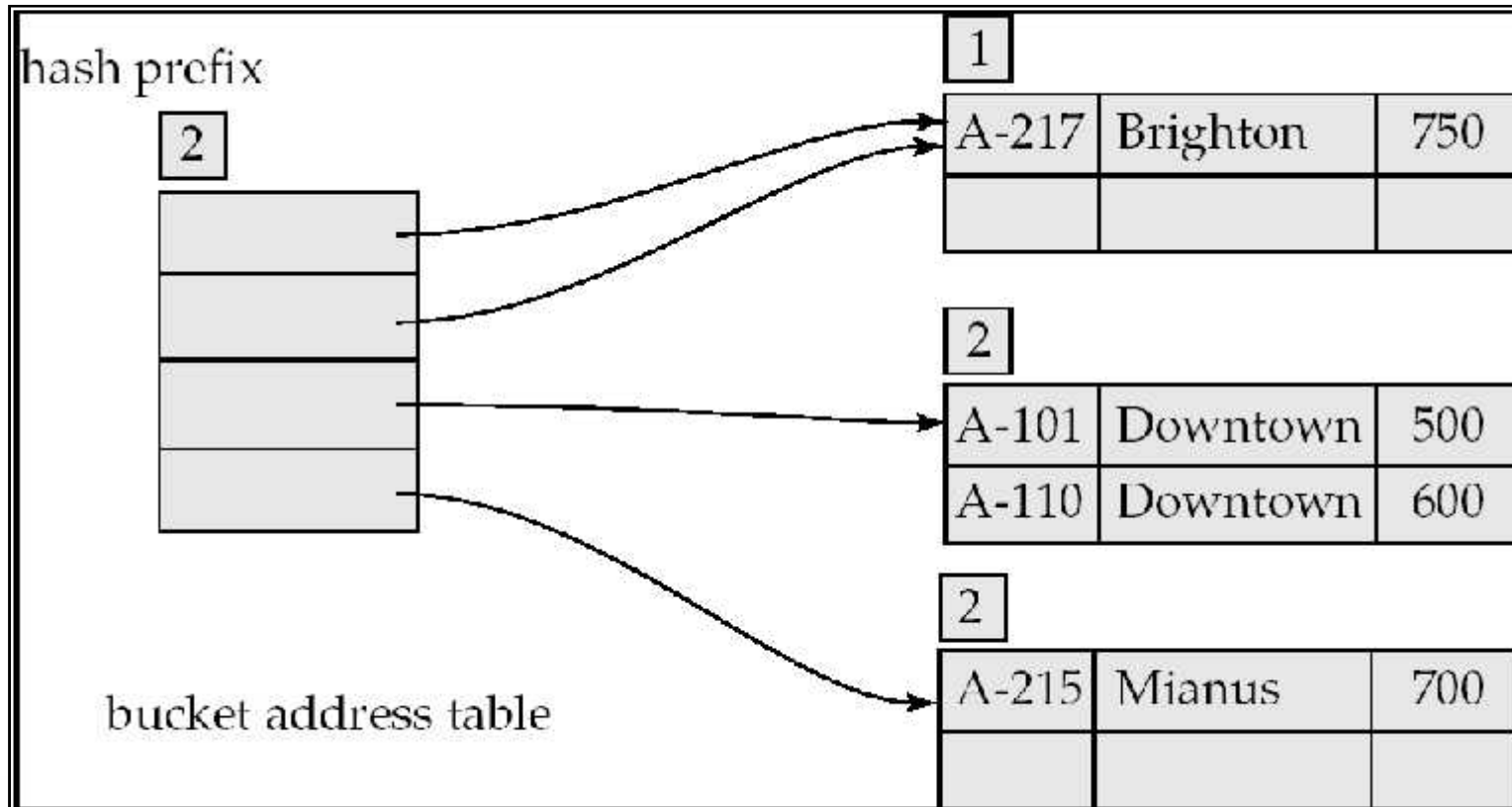
# Example (Cont.)

- Hash structure after insertion of one Brighton and two Downtown records



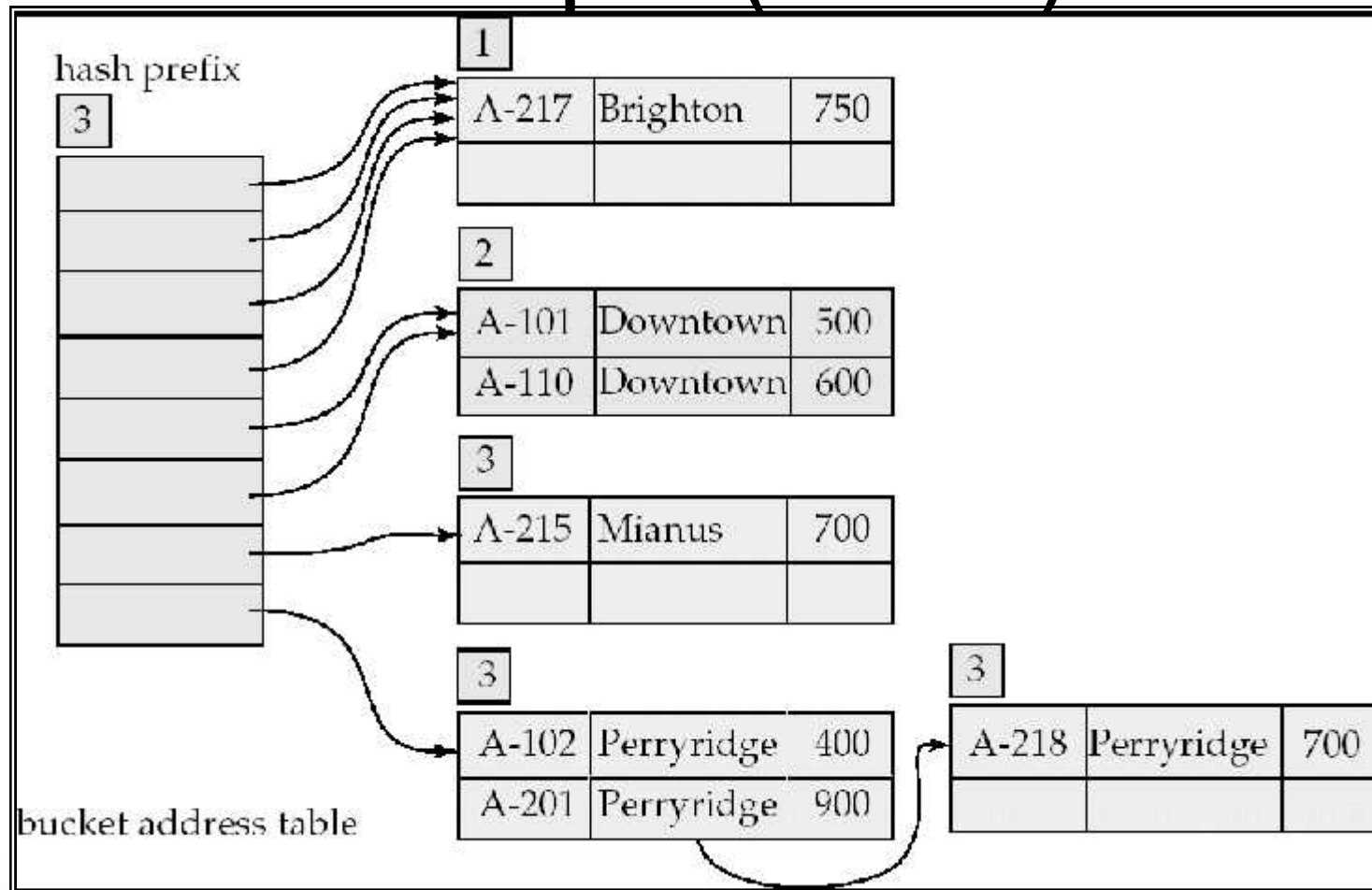
# Example (Cont.)

Hash structure after insertion of Mianus record





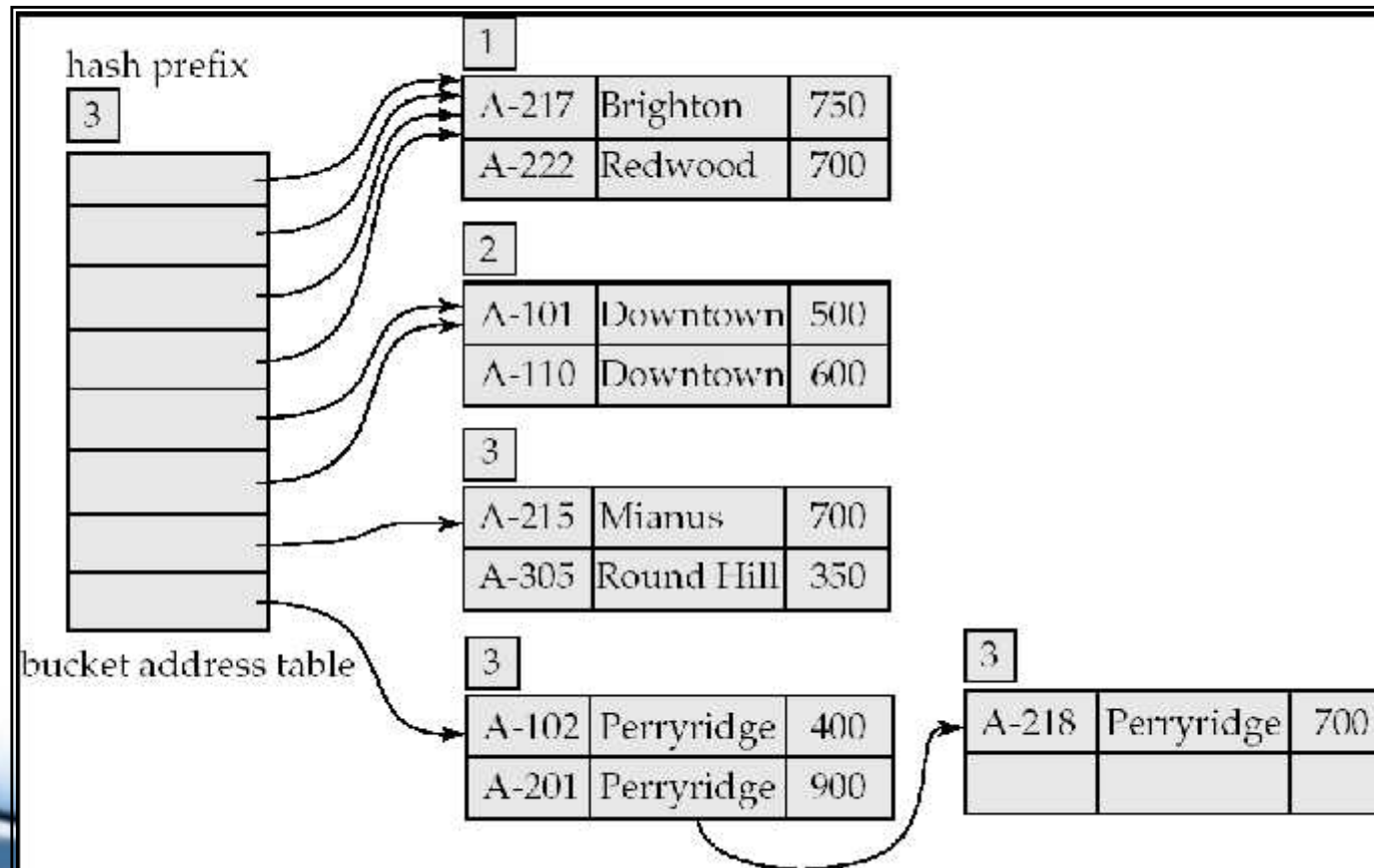
# Example (Cont.)



Hash structure after insertion of three Perryridge records

# Example (Cont.)

- Hash structure after insertion of Redwood and Round Hill records



# Extendable Hashing vs. Other Schemes

- Benefits of extendable hashing:
  - Hash performance does not degrade with growth of file
  - Minimal space overhead
- Disadvantages of extendable hashing
  - Extra level of indirection to find desired record
  - Bucket address table may itself become very big (larger than memory)
    - Cannot allocate very large contiguous areas on disk either
    - Solution: B<sup>+</sup>-tree structure to locate desired record in bucket address table
  - Changing size of bucket address table is an expensive operation
- Linear hashing is an alternative mechanism
  - Allows incremental growth of its directory (equivalent to bucket address table)
  - At the cost of more bucket overflows

# Comparison of Ordered Indexing and Hashing

- Cost of periodic re-organization
- Relative frequency of insertions and deletions
- Is it desirable to optimize average access time at the expense of worst-case access time?
- Expected type of queries:
  - Hashing is generally better at retrieving records having a specified value of the key.
  - If range queries are common, ordered indices are to be preferred
- In practice:
  - PostgreSQL supports hash indices, but discourages use due to poor performance
  - Oracle supports static hash organization, but not hash indices
  - SQLServer supports only B<sup>+</sup>-trees

# Bitmap Indices

- Bitmap indices are a special type of index designed for efficient querying on multiple keys
- Records in a relation are assumed to be numbered sequentially from, say, 0
  - Given a number  $n$  it must be easy to retrieve record  $n$ 
    - Particularly easy if records are of fixed size
- Applicable on attributes that take on a relatively small number of distinct values
  - E.g. gender, country, state, ...
  - E.g. income-level (income broken up into a small number of levels such as 0-9999, 10000-19999, 20000-50000, 50000-infinity)
- A bitmap is simply an array of bits

# Bitmap Indices (Cont.)

- In its simplest form a bitmap index on an attribute has a bitmap for each value of the attribute
  - Bitmap has as many bits as records
  - In a bitmap for value v, the bit for a record is 1 if the record has the value v for the attribute, and is 0 otherwise

record number	name	gender	address	income_level	Bitmaps for gender		Bitmaps for income_level	
0	John	m	Perryridge	L1	m	1 0 0 1 0	L1	1 0 1 0 0
1	Diana	f	Brooklyn	L2	f	0 1 1 0 1	L2	0 1 0 0 0
2	Mary	f	Jonestown	L1			L3	0 0 0 0 1
3	Peter	m	Brooklyn	L4			L4	0 0 0 1 0
4	Kathy	f	Perryridge	L3			L5	0 0 0 0 0

# Bitmap Indices (Cont.)

- Bitmap indices are useful for queries on multiple attributes
  - not particularly useful for single attribute queries
- Queries are answered using bitmap operations
  - Intersection (and)
  - Union (or)
  - Complementation (not)
- Each operation takes two bitmaps of the same size and applies the operation on corresponding bits to get the result bitmap
  - E.g.  $100110 \text{ AND } 110011 = 100010$   
 $100110 \text{ OR } 110011 = 110111$   
 $\text{NOT } 100110 = 011001$
  - Males with income level L1:  $10010 \text{ AND } 10100 = 10000$ 
    - Can then retrieve required tuples.
    - Counting number of matching tuples is even faster



# Bitmap Indices (Cont.)

- Bitmap indices generally very small compared with relation size
  - E.g. if record is 100 bytes, space for a single bitmap is 1/800 of space used by relation.
    - If number of distinct attribute values is 8, bitmap is only 1% of relation size
- Deletion needs to be handled properly
  - Existence bitmap to note if there is a valid record at a record location
  - Needed for complementation
    - $\text{not}(A=v)$ : *(NOT bitmap-A-v) AND ExistenceBitmap*
- Should keep bitmaps for all values, even null value
  - To correctly handle SQL null semantics for  $\text{NOT}(A=v)$ :
    - intersect above result with *(NOT bitmap-A-Null)*

# Efficient Implementation of Bitmap Operations

- Bitmaps are packed into words; a single word and (a basic CPU instruction) computes and of 32 or 64 bits at once
  - E.g. 1-million-bit maps can be and-ed with just 31,250 instruction
- Counting number of 1s can be done fast by a trick:
  - Use each byte to index into a precomputed array of 256 elements each storing the count of 1s in the binary representation
    - Can use pairs of bytes to speed up further at a higher memory cost
  - Add up the retrieved counts
- Bitmaps can be used instead of Tuple-ID lists at leaf levels of B<sup>+</sup>-trees, for values that have a large number of matching records
  - Worthwhile if  $> 1/64$  of the records have that value, assuming a tuple-id is 64 bits
  - Above technique merges benefits of bitmap and B<sup>+</sup>-tree indices

# Index Definition in SQL

- Create an index

**create index** <index-name> **on** <relation-name>  
(<attribute-list>)

E.g.: **create index** *b-index* **on** *branch(branch\_name)*

- Use **create unique index** to indirectly specify and enforce the condition that the search key is a candidate key is a candidate key.
  - Not really required if SQL **unique** integrity constraint is supported
- To drop an index

**drop index** <index-name>
- Most database systems allow specification of type of index, and clustering.

# Transaction Management and Concurrency Control

# What is a Transaction?

- Any action that reads from and/or writes to a database may consist of:
  - Simple SELECT statement to generate list of table contents
  - Series of related UPDATE statements to change values of attributes in various tables
  - Series of INSERT statements to add rows to one or more tables
  - Combination of SELECT, UPDATE, and INSERT statements

# What is a Transaction?

- Transaction is logical unit of work that must be either entirely completed or aborted
- Successful transaction changes database from one consistent state to another
  - One in which all data integrity constraints are satisfied
- Most real-world database transactions are formed by two or more database requests
  - Equivalent of a single SQL statement in an application program or transaction

# Transaction Concept

- A **transaction** is a *unit* of program execution that accesses and possibly updates various data items.
- A transaction must see a consistent database.
- During transaction execution the database may be temporarily inconsistent.
- When the transaction completes successfully (is committed), the database must be consistent.
- After a transaction commits, the changes it has made to the database persist, even if there are system failures.
- Multiple transactions can execute in parallel.
- Two main issues to deal with:
  - Failures of various kinds, such as hardware failures and system crashes
  - Concurrent execution of multiple transactions



# ACID Properties

A **transaction** is a unit of program execution that accesses and possibly updates various data items. To preserve the integrity of data the database system must ensure:

- **Atomicity.** Either all operations of the transaction are properly reflected in the database or none are.
- **Consistency.** Execution of a transaction in isolation preserves the consistency of the database.

# ACID Properties

- **Isolation.** Although multiple transactions may execute concurrently, each transaction must be unaware of other concurrently executing transactions. Intermediate transaction results must be hidden from other concurrently executed transactions.
- **Durability.** After a transaction completes successfully, the changes it has made to the database persist, even if there are system failures.

# Example of Fund Transfer

- Transaction to transfer Rs.500 from account A to account B:
  1. **read**(A)
  2.  $A := A - 500$
  3. **write**(A)
  4. **read**(B)
  5.  $B := B + 500$
  6. **write**(B)
- **Atomicity requirement** — if the transaction fails after step 3 and before step 6, the system should ensure that its updates are not reflected in the database, else an inconsistency will result.
- **Consistency requirement** – the sum of A and B is unchanged by the execution of the transaction.

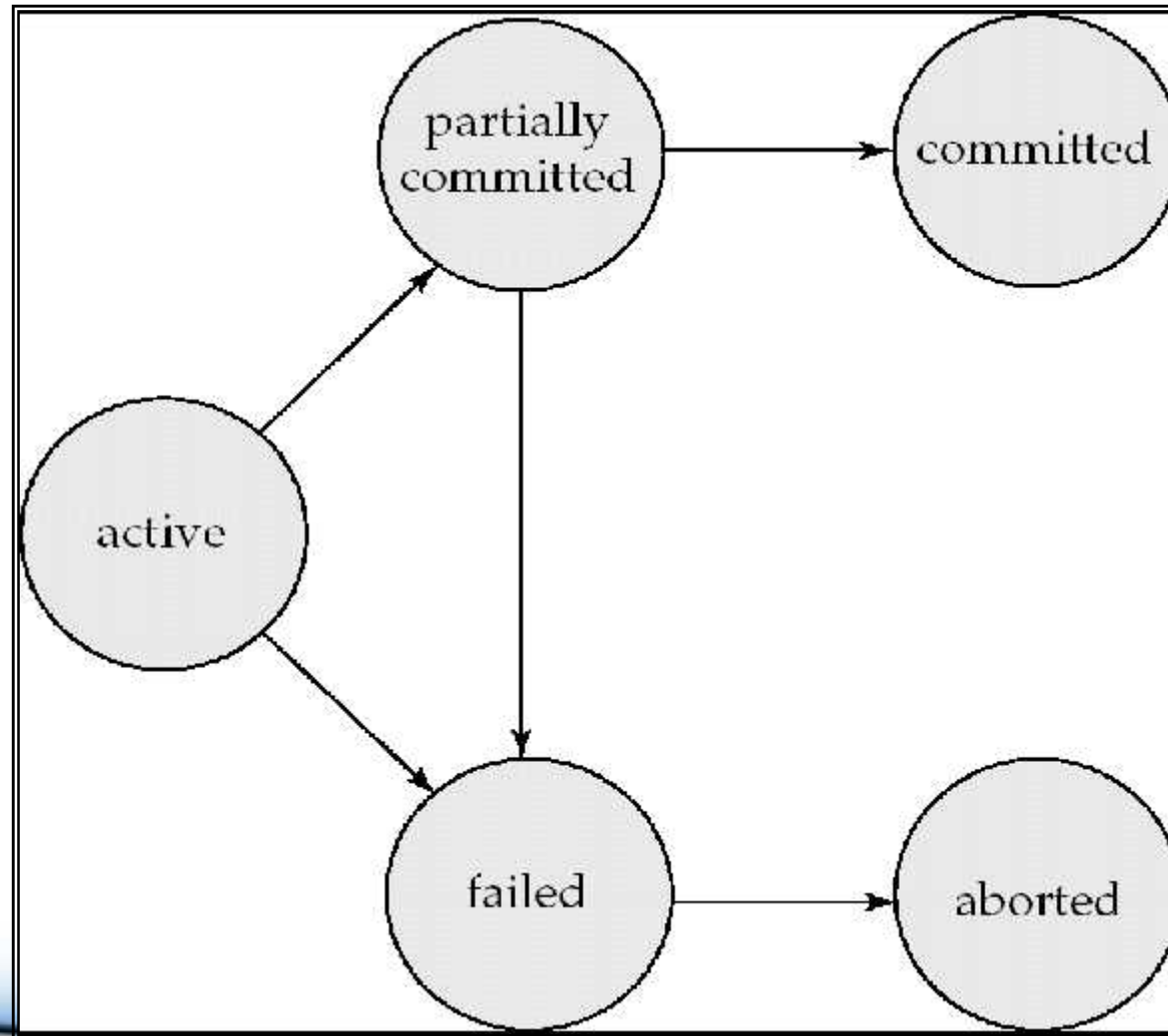
# Example of Fund Transfer (Cont.)

- **Isolation requirement** — if between steps 3 and 6, another transaction is allowed to access the partially updated database, it will see an inconsistent database (the sum  $A + B$  will be less than it should be).
  - Isolation can be ensured trivially by running transactions **serially**, that is one after the other.
  - However, executing multiple transactions concurrently has significant benefits, as we will see later.
- **Durability requirement** — once the user has been notified that the transaction has completed (i.e., the transfer of the Rs.500 has taken place), the updates to the database by the transaction must persist despite failures.

# Transaction State

- **Active** – the initial state; the transaction stays in this state while it is executing
- **Partially committed** – after the final statement has been executed.
- **Failed** - after the discovery that normal execution can no longer proceed.
- **Aborted** – after the transaction has been rolled back and the database restored to its state prior to the start of the transaction. Two options after it has been aborted:
  - restart the transaction; can be done only if no internal logical error
  - kill the transaction
- **Committed** – after successful completion.

# Transaction State



# Transaction Properties

- Serializability
  - Ensures that concurrent execution of several transactions yields consistent results



# Transaction Management with SQL

- ANSI has defined standards that govern SQL database transactions
- Transaction support is provided by two SQL statements: COMMIT and ROLLBACK

# Transaction Management with SQL

- ANSI standards require that, when a transaction sequence is initiated by a user or an application program, it must continue through all succeeding SQL statements until one of four events occurs
  - COMMIT statement is reached
  - ROLLBACK statement is reached
  - End of program is reached
  - Program is abnormally terminated


# The Transaction Log

- Transaction log stores:
  - A record for the beginning of transaction
  - For each transaction component (SQL statement):
    - Type of operation being performed (update, delete, insert)
    - Names of objects affected by transaction
    - “Before” and “after” values for updated fields
    - Pointers to previous and next transaction log entries for the same transaction
  - Ending (COMMIT) of the transaction

# The Transaction Log (continued)

TABLE 10.1 A Transaction Log

TRX ID	TRX NUM	PREV PTR	NEXT PTR	OPERATION	TABLE	ROW ID	ATTRIBUTE	BEFORE VALUE	AFTER VALUE
341	101	Null	352	START	****Start transaction				
352	101	341	363	UPDATE	PRODUCT	1550-QWE	PROD_QOH	25	23
363	101	352	365	UPDATE	CUSTOMER	10011	CUST_BALANCE	525.75	615.73
365	101	363	Null	COMMIT	****End of transaction				



**TRX ID** = Transaction log record ID  
**TRX NUM** = Transaction number  
 (Note: The transaction number is automatically assigned by the DBMS.)

**PTR** = Pointer to a transaction log record ID

# Concurrency Control

- Coordination of simultaneous transaction execution in a multiprocessing database system
- Objective is to ensure serializability of transactions in a multiuser database environment

# Concurrency Control (continued)

- Simultaneous execution of transactions over a shared database can create several data integrity and consistency problems
  - Lost updates
  - Uncommitted data
  - Inconsistent retrievals

# Lost Updates

TABLE  
10.2

Normal Execution of Two Transactions

TIME	TRANSACTION	STEP	STORED VALUE
1	T1	Read PROD_QOH	35
2	T1	$\text{PROD\_QOH} = 35 + 100$	
3	T1	Write PROD_QOH	135
4	T2	Read PROD_QOH	135
5	T2	$\text{PROD\_QOH} = 135 - 30$	
6	T2	Write PROD_QOH	105

- Product Table's attribute is a product's quantity on hand
- Assume a product whose PROD\_QOH value is 35
- Assume two concurrent transactions T1 and T2
- T1: Purchase 100 units
- T2: Sell 30 units



# Lost Updates (continued)

TABLE  
10.3

Lost Updates

TIME	TRANSACTION	STEP	STORED VALUE
1	T1	Read PROD_QOH	35
2	T2	Read PROD_QOH	35
3	T1	$\text{PROD\_QOH} = 35 + 100$	
4	T2	$\text{PROD\_QOH} = 35 - 30$	
5	T1	Write PROD_QOH (Lost update)	135
6	T2	Write PROD_QOH	5

- Suppose a transaction is able to read a product's PROD\_QOH value from the table before a previous transaction has been committed.
- The first transaction T1 has not yet been committed when the second transaction T2 is executed.
- T2 still operates on the value 35, and its subtraction yields 5 in memory.
- T1 writes the value 135 to disk, which is promptly overwritten by T2
- In short, the addition of 100 units is "lost" during the process.

# Uncommitted Data

TABLE  
10.4

Correct Execution of Two Transactions

TIME	TRANSACTION	STEP	STORED VALUE
1	T1	Read PROD_QOH	35
2	T1	$\text{PROD\_QOH} = 35 + 100$	
3	T1	Write PROD_QOH	135
4	T1	*****ROLLBACK*****	35
5	T2	Read PROD_QOH	35
6	T2	$\text{PROD\_QOH} = 35 - 30$	
7	T2	Write PROD_QOH	5

# I Incommitted Data (continued)

TABLE  
10.5

An Uncommitted Data Problem

TIME	TRANSACTION	STEP	STORED VALUE
1	T1	Read PROD_QOH	35
2	T1	$\text{PROD\_QOH} = 35 + 100$	
3	T1	Write PROD_QOH	135
4	T2	Read PROD_QOH (Read uncommitted data)	135
5	T2	$\text{PROD\_QOH} = 135 - 30$	
6	T1	***** ROLLBACK *****	35
7	T2	Write PROD_QOH	105

The phenomenon of Uncommitted data occurs when two transactions, T1 and T2, are executed concurrently and the first transaction T1 is rolled back after the second transaction T2 has already accessed the uncommitted data.

# Inconsistent Retrievals

TABLE  
10.6

Retrieval During Update

TRANSACTION T1		TRANSACTION T2	
SELECT	SUM(PROD_QOH)	UPDATE	PRODUCT
FROM	PRODUCT	SET	PROD_QOH = PROD_QOH + 10
		WHERE	PROD_CODE = '1546-QQ2'
		UPDATE	PRODUCT
		SET	PROD_QOH = PROD_QOH - 10
		WHERE	PROD_CODE = '1558-QW1'
		COMMIT;	

- Inconsistent retrievals occur when a transaction calculates some summary functions over a set of data while other transactions are updating the data.
- The transaction might read some data before they are changed and other data after they are changed.
- Thereby yielding inconsistent results.

# Inconsistent Retrievals (continued)

TABLE  
10.7 Transaction Results: Data Entry Correction

	BEFORE	AFTER
PROD_CODE	PROD_QOH	PROD_QOH
11QER/31	8	8
13-Q2/P2	32	32
1546-QQ2	15	$(15 + 10) \rightarrow 25$
1558-QW1	23	$(23 - 10) \rightarrow 13$
2232-QTY	8	8
2232-QWE	6	6
Total	92	92



# Inconsistent Retrievals (continued)

TABLE 10.8 Inconsistent Retrievals.

TIME	TRANSACTION	ACTION	VALUE	TOTAL
1	T1	Read PROD_QOH for PROD_CODE = '11QER/31'	8	8
2	T1	Read PROD_QOH for PROD_CODE = '13-Q2/P2'	32	40
3	T2	Read PROD_QOH for PROD_CODE = '1546-QQ2'	15	
4	T2	PROD_QOH = 15 + 10		
5	T2	Write PROD_QOH for PROD_CODE = '1546-QQ2'	25	
6	T1	Read PROD_QOH for PROD_CODE = '1546-QQ2'	25	(After) 65
7	T1	Read PROD_QOH for PROD_CODE = '1558-QW1'	23	(Before) 68
8	T2	Read PROD_QOH for PROD_CODE = '1558-QW1'	23	
9	T2	PROD_QOH = 23 - 10		
10	T2	Write PROD_QOH for PROD_CODE = '1558-QW1'	13	
11	T2	**** COMMIT ****		
12	T1	Read PROD_QOH for PROD_CODE = '2232-QTY'	8	96
13	T1	Read PROD_QOH for PROD_CODE = '2232-QWF'	6	102

# The Scheduler

- Special DBMS program
  - Purpose is to establish order of operations within which concurrent transactions are executed
- Interleaves execution of database operations to ensure serializability and isolation of transactions



# The Scheduler (continued)

- Bases its actions on concurrency control algorithms
- Ensures computer's central processing unit (CPU) is used efficiently
- Facilitates data isolation to ensure that two transactions do not update same data element at same time

# The Scheduler (continued)

**TABLE 10.9** Read/Write Conflict Scenarios: Conflicting Database Operations Matrix

	TRANSACTIONS		RESULT
	T1	T2	
Operations	Read	Read	No conflict
	Read	Write	Conflict
	Write	Read	Conflict
	Write	Write	Conflict

# Concurrency Control with Locking Methods

- Lock
  - Guarantees exclusive use of a data item to a current transaction
  - Required to prevent another transaction from reading inconsistent data
- Lock manager
  - Responsible for assigning and policing the locks used by transactions

# Lock Granularity

- Indicates level of lock use
- Locking can take place at following levels:
  - Database
  - Table
  - Page
  - Row
  - Field (attribute)

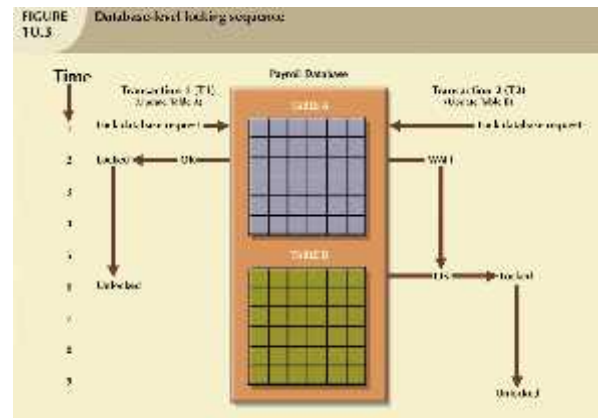
# Lock Granularity (continued)

- Database-level lock
  - Entire database is locked
- Table-level lock
  - Entire table is locked
- Page-level lock
  - Entire diskpage is locked

# Lock Granularity (continued)

- Row-level lock
  - Allows concurrent transactions to access different rows of same table, even if rows are located on same page
- Field-level lock
  - Allows concurrent transactions to access same row, as long as they require use of different fields (attributes) within that row

# Lock Granularity (continued)

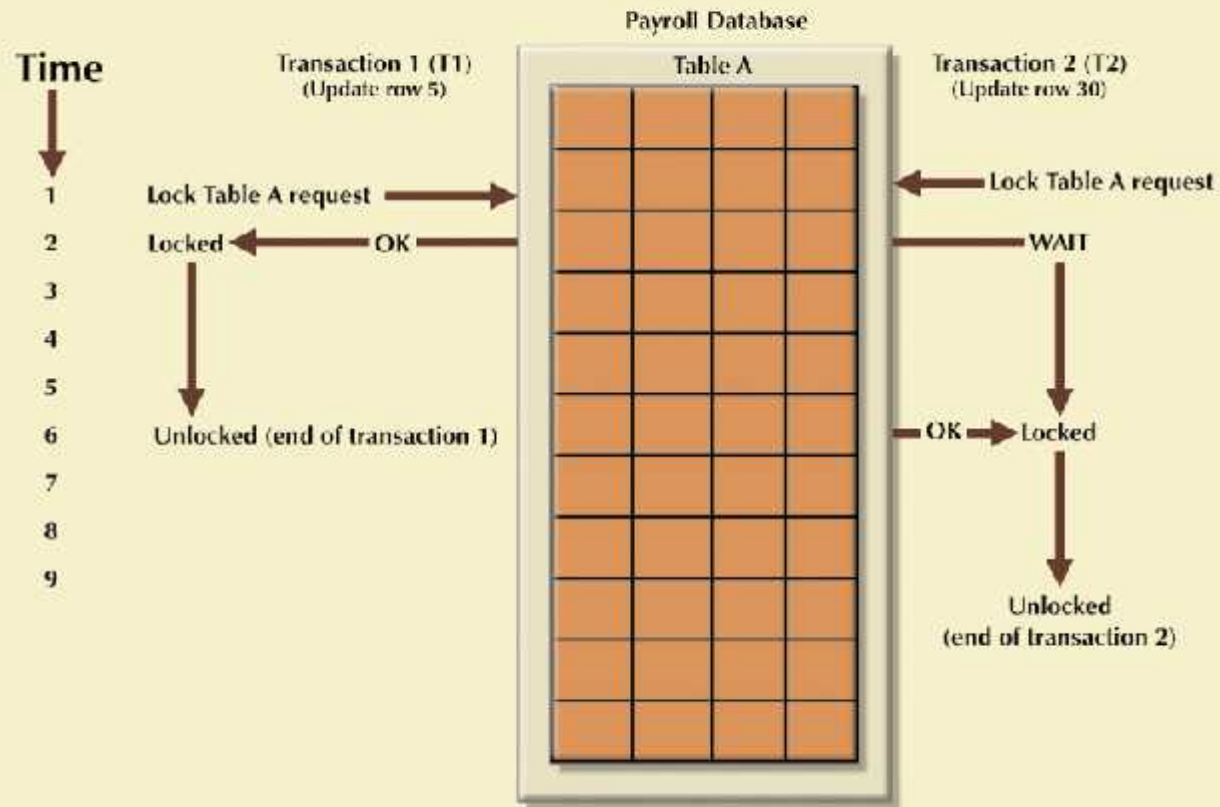




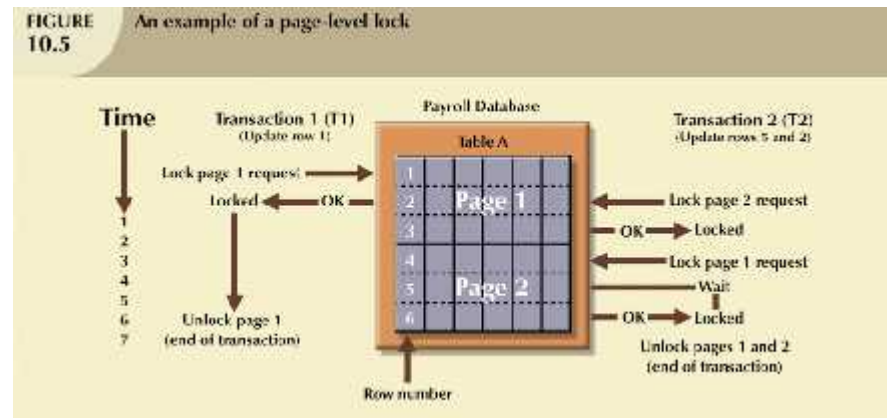
# Lock Granularity (continued)

FIGURE 10.4

An example of a table-level lock



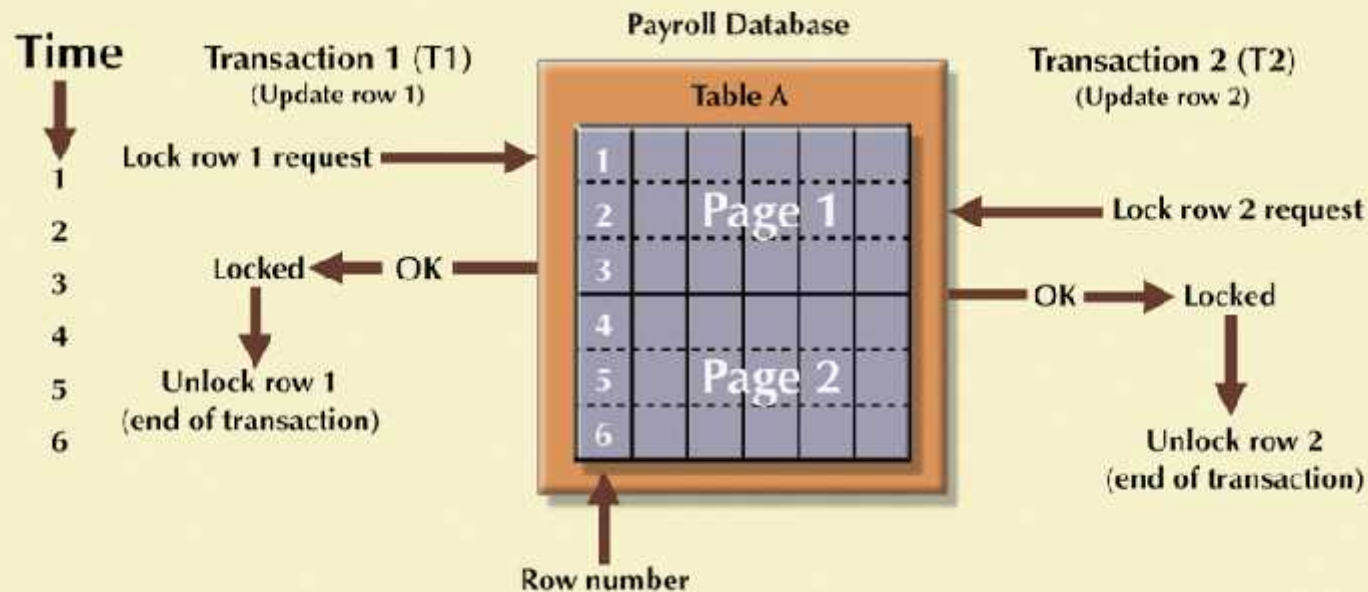
# Lock Granularity (continued)



# Lock Granularity (continued)

FIGURE 10.6

An example of a row-level lock



# Lock Types

- Binary lock
  - Has only two states: locked (1) or unlocked (0)
- Exclusive lock
  - Access is specifically reserved for transaction that locked object
  - Must be used when potential for conflict exists
- Shared lock
  - Concurrent transactions are granted Read access on basis of a common lock

# Locking problems

- Although locks prevent serious data inconsistencies, they lead to
  - The resulting transaction schedule may not be serializable.
  - The schedule may create deadlocks.
- Both problems are manageable
  - Serializability is guaranteed through a locking protocol known as Two-Phase Locking
  - Deadlock can be managed by using deadlock detection and prevention techniques.

# Two-Phase Locking

- Defines how transactions acquire and relinquish locks.
- It guarantees serializability, but does not prevent deadlocks.
- The two phases are:
  - A Growing Phase:
    - In which a transaction acquires all required locks without unlocking any data.
    - Once all locks have been acquired, the transaction is in its locked point
  - A Shrinking Phase:
    - In which a transaction releases all locks and cannot obtain any new lock.

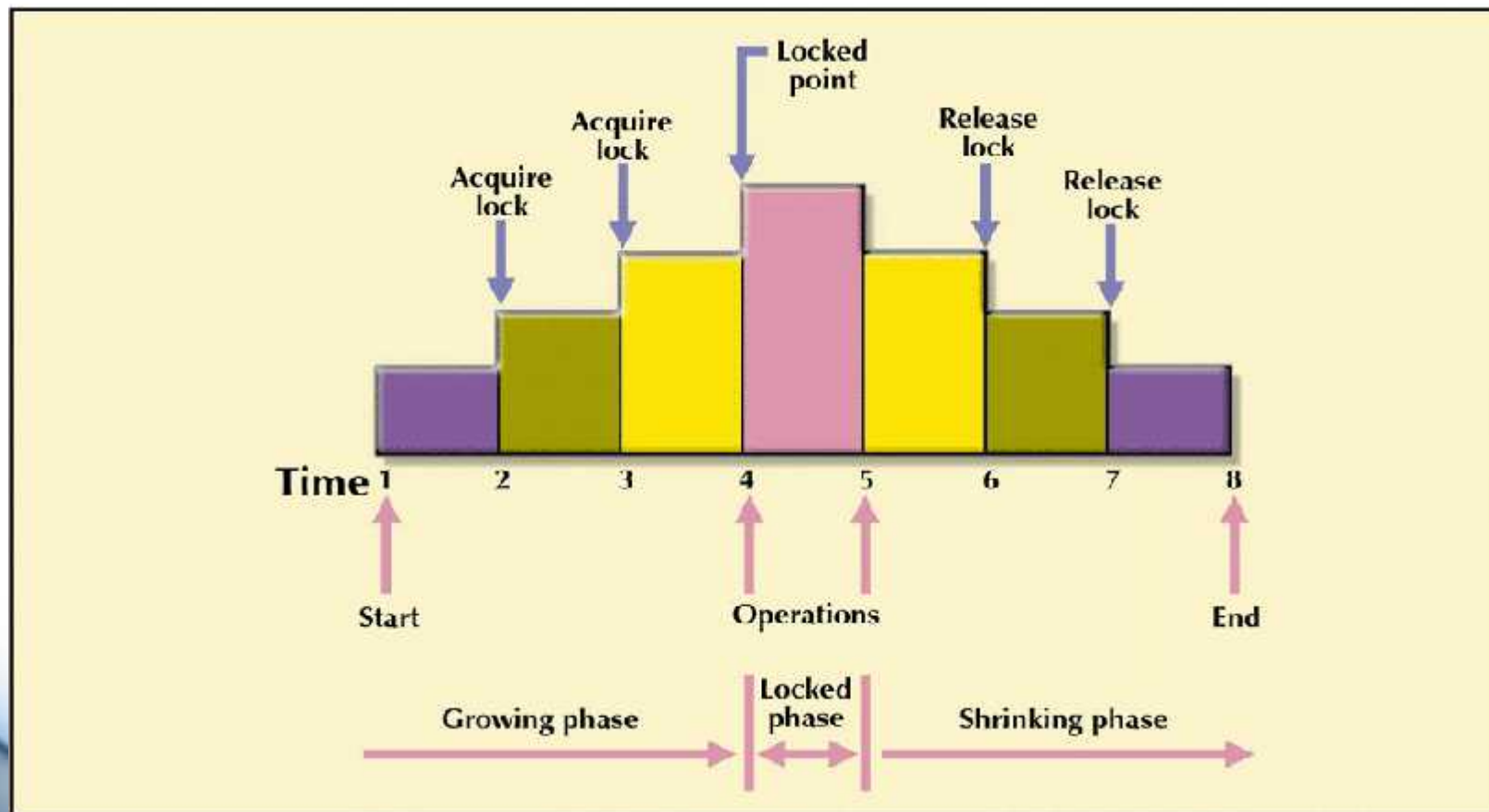
# Two-Phase Locking

- The Two-Phase Locking protocol is governed by the following rules
  - Two transactions cannot have conflicting locks.
  - No unlock operation can precede a lock operation in the same transaction.
  - No data are affected until all locks are obtained – i.e until the transaction is in its locked point.



# Two-Phase Locking Protocol

FIGURE 9.7 TWO-PHASE LOCKING PROTOCOL



# Deadlocks

- Condition that occurs when two transactions wait for each other to unlock data
- Possible only if one of the transactions wants to obtain an exclusive lock on a data item
  - No deadlock condition can exist among *shared* locks
- Control through
  - Prevention
  - Detection
  - Avoidance

# How a Deadlock Condition Is Created

**TABLE 9.11** HOW A DEADLOCK CONDITION IS CREATED

TIME	TRANSACTION	REPLY	LOCK STATUS	
			Data X	Data Y
0			Unlocked	Unlocked
1	T1: LOCK(X)	OK	Unlocked	Unlocked
2	T2: LOCK(Y)	OK	Locked	Unlocked
3	T1: LOCK(Y)	WAIT	Locked	Locked
4	T2: LOCK(X)	WAIT	Locked	Locked
5	T1: LOCK(Y)	WAIT	Locked	Locked
6	T2: LOCK(X)	WAIT	Locked	Locked
7	T1: LOCK(Y)	WAIT	Locked	Locked
8	T2: LOCK(X)	WAIT	Locked	Locked
9	T1: LOCK(Y)	WAIT	Locked	Locked
...	.....	.....	.....	.....
...	.....	.....	.....	.....
...	.....	.....	.....	.....
...	.....	.....	.....	.....

Deadlock

# Concurrency Control with Time Stamping Methods

- Assigns global unique time stamp to each transaction
- Produces explicit order in which transactions are submitted to DBMS
- Uniqueness
  - Ensures that no equal time stamp values can exist
- Monotonicity
  - Ensures that time stamp values always increase

# Wait/Die and Wound/Wait Schemes

- Wait/die
  - Older transaction waits and the younger is rolled back and rescheduled
- Wound/wait
  - Older transaction rolls back the younger transaction and reschedules it

# Wait/Die and Wound/Wait Schemes (continued)

TABLE 10.12 Wait/Die and Wound/Wait Concurrency Control Schemes

TRANSACTION REQUESTING LOCK	TRANSACTION OWNING LOCK	WAIT/DIE SCHEME	WOUND/WAIT SCHEME
T1 (11548789)	T2 (19562545)	<ul style="list-style-type: none"><li>• T1 waits until T2 is completed and T2 releases its locks.</li></ul>	<ul style="list-style-type: none"><li>• T1 preempts (rolls back) T2.</li><li>• T2 is rescheduled using the same time stamp.</li></ul>
T2 (19562545)	T1 (11548789)	<ul style="list-style-type: none"><li>• T2 dies (rolls back).</li><li>• T2 is rescheduled using the same time stamp.</li></ul>	<ul style="list-style-type: none"><li>• T2 waits until T1 is completed and T1 releases its locks.</li></ul>

# Concurrency Control with Optimistic Methods

- Optimistic approach
  - Based on the assumption that the majority of database operations do not conflict
  - Does not require locking or time stamping techniques
  - Transaction is executed without restrictions until it is committed
  - Phases are read, validation, and write



# Database Recovery Management

- Database recovery
  - Restores database from a given state, usually inconsistent, to a previously consistent state
  - Based on the atomic transaction property
    - All portions of the transaction must be treated as a single logical unit of work, in which all operations must be applied and completed to produce a consistent database
  - If transaction operation cannot be completed, transaction must be aborted, and any changes to the database must be rolled back (undone)

# Transaction Recovery

- Important aspects that affect the recovery process
  - The **write-ahead-log protocol** - ensures that transaction logs are written before any database data are actually updated.
  - **Redundant transaction logs** – DBMSs keep several copies of the transaction log
  - Database **Buffers**
  - Database **Checkpoints** – A database checkpoint is an operation in which the DBMS writes all of its updated buffers to disk

# Transaction Recovery

- Makes use of deferred-write and write-through
- Deferred write or Deferred update
  - Transaction operations do not immediately update the physical database
  - Only the transaction log is updated
  - Database is physically updated only after the transaction reaches its commit point using the transaction log information

# Transaction Recovery (continued)

- Write-through or immediate update
  - Database is immediately updated by transaction operations during the transaction's execution, even before the transaction reaches its commit point

# Security and Authorization

# Introduction to DB Security

- **Secrecy:** Users should not be able to see things they are not supposed to.
  - E.g., A student can't see other students' grades.
- **Integrity:** Users should not be able to modify things they are not supposed to.
  - E.g., Only instructors can assign grades.
- **Availability:** Users should be able to see and modify things they are allowed to.

# Access Controls

- A **security policy** specifies who is authorized to do what.
- A **security mechanism** allows us to enforce a chosen security policy.
- Two main mechanisms at the DBMS level:
  - Discretionary access control
  - Mandatory access control



# Discretionary Access Control

- Based on the concept of access rights or **privileges** for objects (tables and views), and mechanisms for giving users privileges (and revoking privileges).
- Creator of a table or a view automatically gets all privileges on it.
  - DMBS keeps track of who subsequently gains and loses privileges, and ensures that only requests from users who have the necessary privileges (at the time the request is issued) are allowed.
- SQL supports discretionary access control through the GRANT and REVOKE

# GRANT Command

GRANT **privileges** ON object **TO** users [WITH GRANT OPTION]

- The following **privileges** can be specified:
  - ❖ **SELECT**: Can read all columns (including those added later via ALTER TABLE command).
  - ❖ **INSERT**(col-name): Can insert tuples with non-null or non-default values in this column.
    - ❖ INSERT means same right with respect to all columns.
  - ❖ **DELETE**: Can delete tuples.
  - ❖ **REFERENCES** (col-name): Can define foreign keys (in other tables) that refer to this column.

# GRANT Command

GRANT privileges ON object TO users [WITH GRANT OPTION]

- If a user has a privilege with the GRANT OPTION, can pass privilege on to other users (with or without passing on the GRANT OPTION).
- Only owner can execute CREATE, ALTER, and DROP.

# GRANT and REVOKE of Privileges

- ▶ GRANT INSERT, SELECT ON EMP TO RAM  
RAM can query EMP or insert tuples into it.
- ▶ GRANT DELETE ON EMP TO MOHAN WITH GRANT OPTION  
MOHAN can delete tuples, and also authorize others to do so.
- ▶ GRANT UPDATE (*rating*) ON EMP TO MOHIT  
MOHIT can update (only) the *rating* field of EMP tuples.
- ▶ GRANT SELECT ON VIEWEMP TO GOPAL, RADHA  
This does NOT allow the GOPAL and RADHA to query EMP table directly!
- ▶ **REVOKE:** When a privilege is revoked from X, it is also revoked from all users who got it *solely* from X.

# REVOKE privilege

- The Syntax of the REVOKE command is as follows
- REVOKE [ GRANT OPTION FOR]  
privileges ON object FROM users  
{ RESTRICT| CASCADE }
  - REVOKE INSERT, SELECT ON EMP FROM RAM
  - REVOKE DELETE ON EMP FROM MOHAN

# DBMS maintaining GRANT

- Privilege descriptor – is added to a table, when a GRANT is executed.
  - Such Privilege descriptors are maintained by the DBMS
  - It specifies following: the *Grantor* of the privilege, the *Grantee* who receives the privilege, the *Granted privilege* and whether the *Grant option* is included.
- Authorization Graph – the effect of series of GRANT commands
  - The nodes are users – authorization IDs
  - The arcs indicates how privileges are passed.
  - The arc is labeled with descriptor for the GRANT command

# GRANT/REVOKE on Views

- If the creator of a view loses the `SELECT` privilege on an underlying table, the view is dropped!
- If the creator of a view loses a privilege held with the grant option on an underlying table, (s)he loses the privilege on the view as well; so do users who were granted that privilege on the view!



# GRANT/REVOKE on Views

- If the creator of a view gains additional privileges on the underlying tables, (S)/he automatically gains additional privileges on view
- The distinction between the REFERENCES and SELECT privilege is important.

# Views and Security

- ▶ Views can be used to present necessary information (or a summary), while hiding details in underlying relation(s).
- ▶ Creator of view has a privilege on the view if (s)he has the privilege on all underlying tables.
- ▶ Together with GRANT/REVOKE commands, views are a very powerful access control tool.

# Role-Based Authorization

- In SQL-92, privileges are actually assigned to **authorization ids**, which can denote a single user or a group of users.
- In SQL:1999 (and in many current systems), privileges are assigned to **roles**.
  - Roles can then be granted to users and to other roles.
  - Reflects how real organizations work.
  - A **Role** is a named collection of database access privileges that authorize a user to connect to the database and use the database system resources.
    - **CONNECT**: allows a user to connect to the database and create and modify tables, views and other data-related objects
    - **RESOURCE**: allows a user to create triggers, procedures and other data management objects
    - **DBA**: gives the user database administration privileges.

# Internet-Oriented Security

- Key Issues: User authentication and trust.
  - When DB must be accessed from a secure location, password-based schemes are usually adequate.
- For access over an external network, trust is hard to achieve.
  - If someone with Sam's credit card wants to buy from you, how can you be sure it is not someone who stole his card?
  - How can Sam be sure that the screen for entering his credit card information is indeed yours, and not some rogue site spoofing you (to steal such information)? How can he be sure that sensitive information is not "sniffed" while it is being sent over the network to you?
- *Encryption* is a technique used to address these issues.

# Encryption

- “Masks” data for secure transmission or storage
  - $\text{Encrypt}(\text{data}, \text{encryption key}) = \text{encrypted data}$
  - $\text{Decrypt}(\text{encrypted data}, \text{decryption key}) = \text{original data}$
  - Without decryption key, the encrypted data is meaningless gibberish
- **Symmetric Encryption:**
  - Encryption key = decryption key; all authorized users know decryption key (a weakness).
  - DES, used since 1977, has 56-bit key; AES has 128-bit (optionally, 192-bit or 256-bit) key
- **Public-Key Encryption:** Each user has two keys:
  - User’s public encryption key: Known to all
  - Decryption key: Known only to this user
  - Used in RSA scheme (Turing Award!)

# Certifying Servers: SSL, SET

- If Amazon distributes their public key, Sam's browser will encrypt his order using it.
  - So, only Amazon can decipher the order, since no one else has Amazon's private key.
- But how can Sam (or his browser) know that the public key for Amazon is genuine? The SSL protocol covers this.
  - Amazon contracts with, say, Verisign, to issue a certificate <Verisign, Amazon, amazon.com, public-key>
  - This certificate is stored in encrypted form, encrypted with Verisign's *private* key, known only to Verisign.
  - Verisign's public key is known to all browsers, which can therefore decrypt the certificate and obtain Amazon's public key, and be confident that it is genuine.
  - The browser then generates a temporary *session key*, encodes it using Amazon's public key, and sends it to Amazon.
  - All subsequent msgs between the browser and Amazon are encoded using symmetric encryption (e.g., DES), which is more efficient than public-key encryption.
- What if Sam doesn't trust Amazon with his credit card information?
  - Secure Electronic Transaction protocol: 3-way communication between Amazon, Sam, and a trusted server, e.g., Visa.



# Authenticating Users

- Amazon can simply use password authentication, i.e., ask Sam to log into his Amazon account.
  - Done after SSL is used to establish a session key, so that the transmission of the password is secure!
  - Amazon is still at risk if Sam's card is stolen and his password is hacked. Business risk ...
- Digital Signatures:
  - Sam encrypts the order using his *private* key, then encrypts the result using Amazon's public key.
  - Amazon decrypts the msg with their private key, and then decrypts the result using Sam's public key, which yields the original order!
  - Exploits interchangeability of public/private keys for encryption/decryption
  - Now, no one can forge Sam's order, and Sam cannot claim that someone else forged the order.



# Mandatory Access Control

- Based on system-wide policies that cannot be changed by individual users.
  - Each **DB object** is assigned a **security class**.
  - Each **subject** (user or user program) is assigned a **clearance** for a security class.
  - Rules based on security classes and clearances govern who can read/write which objects.
- Most commercial systems do not support mandatory access control. Versions of some DBMSs do support it; used for specialized (e.g., military) applications.

# Why Mandatory Control?

- Discretionary control has some flaws, e.g., the *Trojan horse* problem:
  - Dick creates Horsie and gives INSERT privileges to Justin (who doesn't know about this).
  - Dick modifies the code of an application program used by Justin to additionally write some secret data to table Horsie.
  - Now, Justin can see the secret info.
- The modification of the code is beyond the DBMSs control, but it can try and prevent the use of the database as a **channel** for secret information.

# Bell-LaPadula Model

- Objects (e.g., tables, views, tuples)
- Subjects (e.g., users, user programs)
- Security classes:
  - Top secret (TS), secret (S), confidential (C), unclassified (U):  $TS > S > C > U$
- Each object and subject is assigned a class.
  - Subject S can read object O only if  $class(S) \geq class(O)$  (Simple Security Property)
  - Subject S can write object O only if  $class(S) \leq class(O)$  (\*-Property)

# Intuition

- Idea is to ensure that information can never flow from a higher to a lower security level.
- E.g., If Dick has security class C, Justin has class S, and the secret table has class S:
  - Dick's table, Horsie, has Dick's clearance, C.
  - Justin's application has his clearance, S.
  - So, the program cannot write into table Horsie.
- The mandatory access control rules are applied in addition to any discretionary controls that are in effect.

# Multilevel Relations

<u>bid</u>	bname	color	class
101	Salsa	Red	S
102	Pinto	Brown	C

- Users with S and TS clearance will see both rows; a user with C will only see the 2<sup>nd</sup> row; a user with U will see no rows.
- If user with C tries to insert <101,Pasta,Blue,C>:
  - Allowing insertion violates key constraint
  - Disallowing insertion tells user that there is another object with key 101 that has a class > C!
  - Problem resolved by treating class field as part of key.

# polyinstantiation

- The presence of data objects that appear to have different values to users with different clearance is called polyinstantiation.

# Mandatory - drawback

- Rigidity
- Policies are set by system administrator
- Classification mechanism are not flexible enough



# Summary

- Three main security objectives: secrecy, integrity, availability.
- DB admin is responsible for overall security.
  - Designs security policy, maintains an **audit trail**, or history of users' accesses to DB.
- Two main approaches to DBMS security: discretionary and mandatory access control.
  - Discretionary control based on notion of privileges.
  - Mandatory control based on notion of security classes.

# THANK YOU

