

Chapter 1: Introduction

Purpose of Database Systems

View of Data

Data Models

Data Definition Language

Data Manipulation Language

Transaction Management

Storage Management

Database Administrator

Database Users

Overall System Structure

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

Purpose of Database System

In the early days, database applications were built on top of file systems

Drawbacks of using file systems to store data:

- Data redundancy and inconsistency
 - Multiple file formats, duplication of information in different files
- 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

Levels of Abstraction

Physical level describes how a record (e.g., customer) is stored.

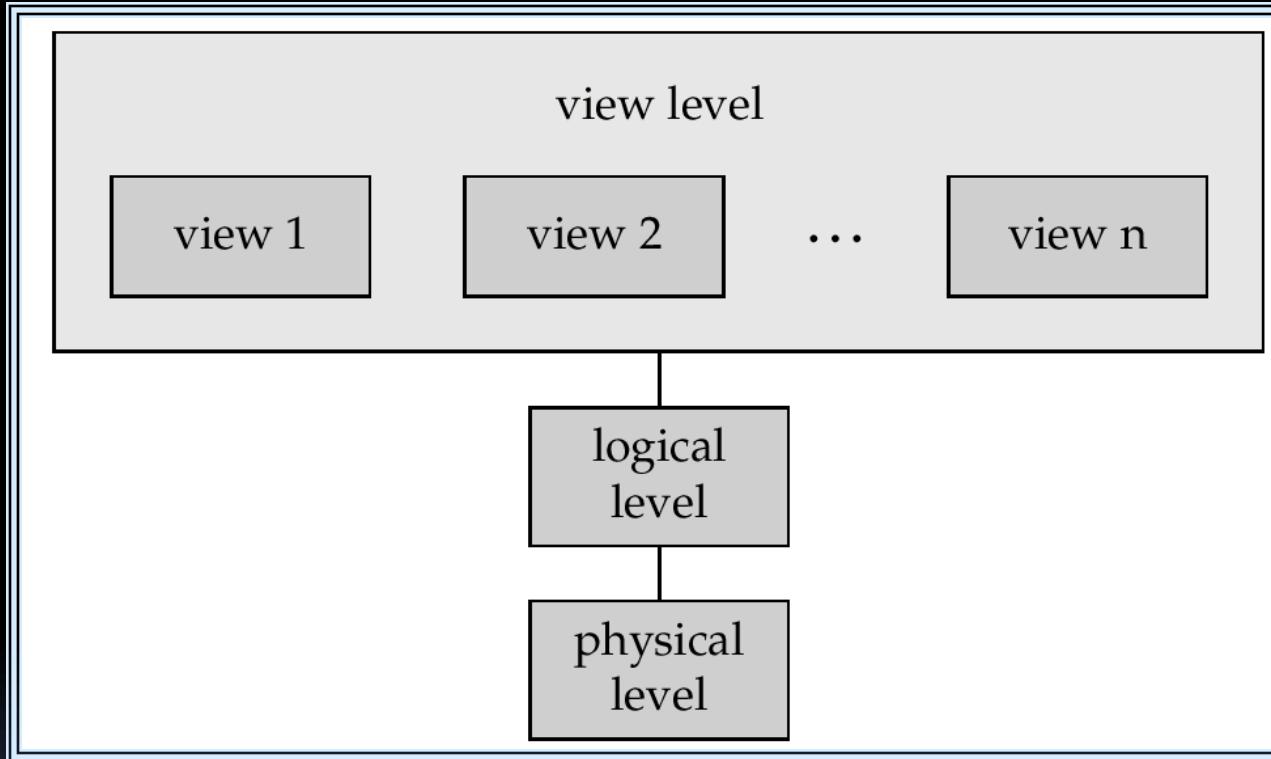
Logical level: describes data stored in database, and the relationships among the data.

```
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

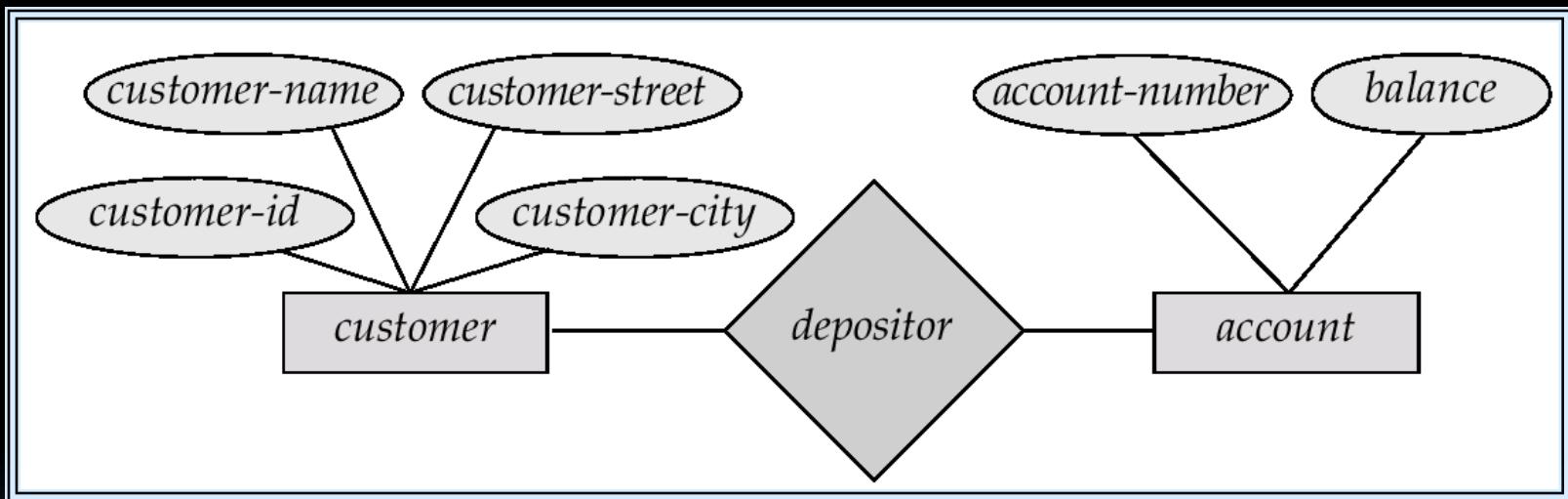
Relational model

Other models:

- object-oriented model
- semi-structured data models
- Older models: network model and hierarchical model

Entity-Relationship Model

Example of schema in the entity-relationship model



Entity Relationship Model (Cont.)

E-R model of real world

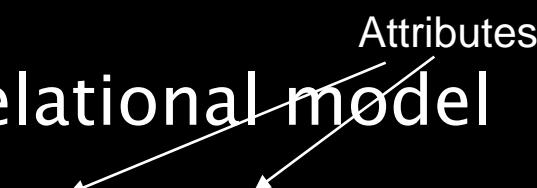
- Entities (objects)
 - E.g. customers, accounts, bank branch
- Relationships between entities
 - E.g. Account A-101 is held by customer Johnson
 - Relationship set *depositor* associates customers with accounts

Widely used for database design

- Database design in E-R model usually converted to design in the relational model (coming up next) which is used for storage and processing

Relational Model

Example of tabular data in the relational model



<i>Customer-id</i>	<i>customer-name</i>	<i>customer-street</i>	<i>customer-city</i>	<i>account-number</i>
192-83-7465	Johnson	Alma	Palo Alto	A-101
019-28-3746	Smith	North	Rye	A-215
192-83-7465	Johnson	Alma	Palo Alto	A-201
321-12-3123	Jones	Main	Harrison	A-217
019-28-3746	Smith	North	Rye	A-201

A Sample Relational Database

<i>customer-id</i>	<i>customer-name</i>	<i>customer-street</i>	<i>customer-city</i>
192-83-7465	Johnson	12 Alma St.	Palo Alto
019-28-3746	Smith	4 North St.	Rye
677-89-9011	Hayes	3 Main St.	Harrison
182-73-6091	Turner	123 Putnam Ave.	Stamford
321-12-3123	Jones	100 Main St.	Harrison
336-66-9999	Lindsay	175 Park Ave.	Pittsfield
019-28-3746	Smith	72 North St.	Rye

(a) The *customer* table

<i>account-number</i>	<i>balance</i>
A-101	500
A-215	700
A-102	400
A-305	350
A-201	900
A-217	750
A-222	700

(b) The *account* table

<i>customer-id</i>	<i>account-number</i>
192-83-7465	A-101
192-83-7465	A-201
019-28-3746	A-215
677-89-9011	A-102
182-73-6091	A-305
321-12-3123	A-217
336-66-9999	A-222
019-28-3746	A-201

(c) The *depositor* table

Data Definition Language (DDL)

Specification notation for defining the database schema

- E.g.

```
create table account(  
    account-number  char(10),  
    balance          integer)
```

DDL compiler generates a set of tables stored in a *data dictionary*

Data dictionary contains metadata (i.e., data about data)

- database schema
- Data *storage and definition* language
 - language in which the storage structure and access methods used by the database system are specified
 - Usually an extension of the data definition language

Data Manipulation Language (DML)

Language for accessing and manipulating the data organized by the appropriate data model

- DML also known as query language

Two classes of languages

- Procedural – user specifies what data is required and how to get those data
- Nonprocedural – user specifies what data is required without specifying how to get those data

SQL is the most widely used query language

SQL

SQL: widely used non-procedural language

- E.g. find the name of the customer with customer-id 192-83-7465

```
select customer.customer-name
      from customer
     where customer.customer-id = '192-83-7465'
```

- E.g. find the balances of all accounts held by the customer with customer-id 192-83-7465

```
select account.balance
      from depositor, account
     where depositor.customer-id = '192-83-7465' and
           depositor.account-number = account.account-number
```

Application programs generally access databases through one of

- Language extensions to allow embedded SQL
- Application program interface (e.g. ODBC/JDBC) which allow SQL queries to be sent to a database

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

Database Administrator

Coordinates all the activities of the database system; the database administrator has a good understanding of the enterprise's information resources and needs.

Database administrator's duties include:

- Schema definition
- Storage structure and access method definition
- Schema and physical organization modification
- Granting user authority to access the database
- Specifying integrity constraints
- Acting as liaison with users
- Monitoring performance and responding to changes in requirements

Transaction Management

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

Transaction-management component ensures that the database remains in a consistent (correct) state despite system failures (e.g., power failures and operating system crashes) and transaction failures.

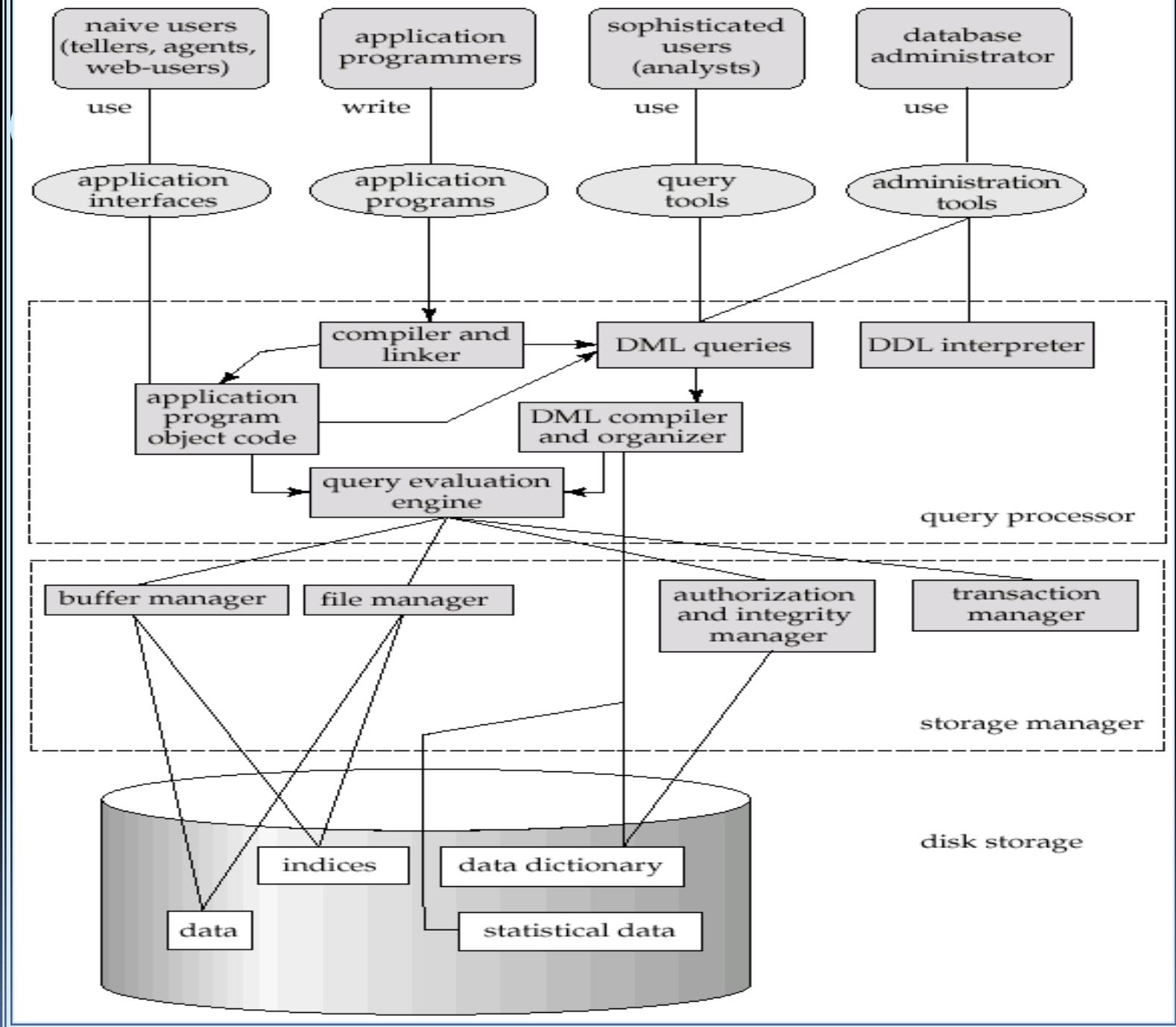
Concurrency-control manager controls the interaction among the concurrent transactions, to ensure the consistency of the database.

Storage Management

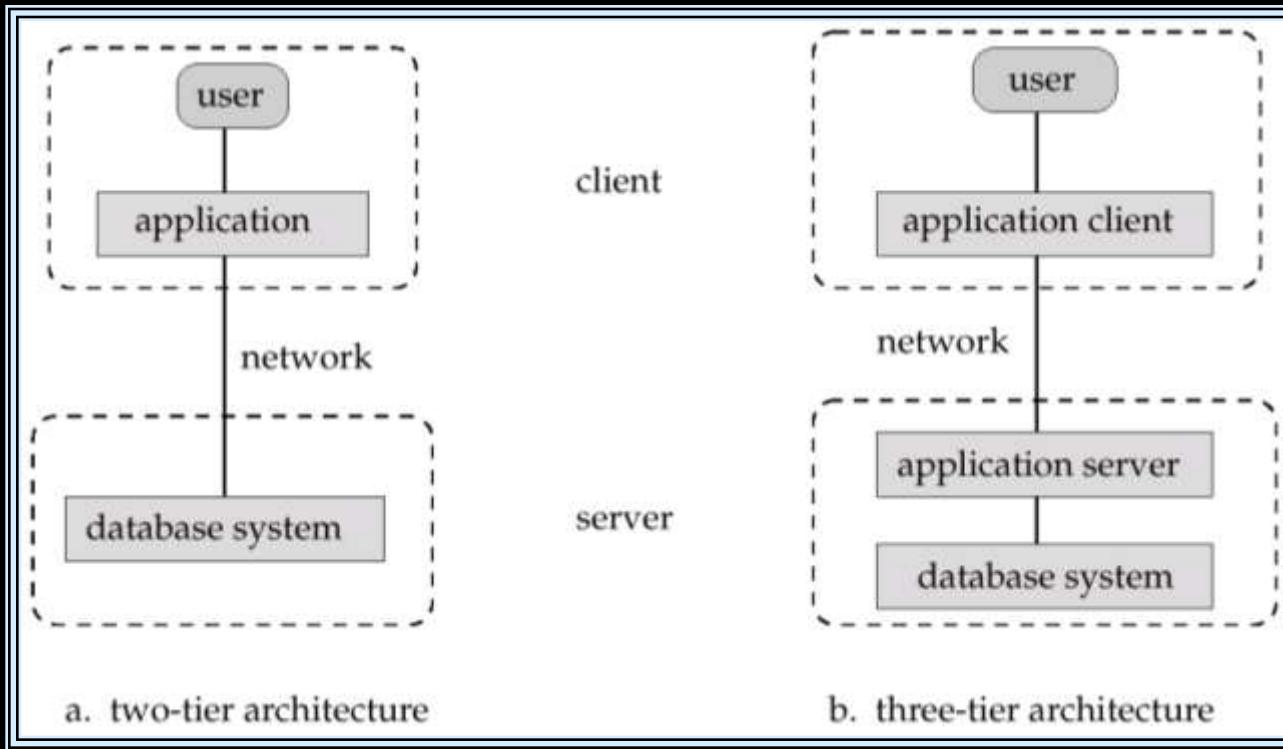
Storage manager is a program module that provides the interface between the low-level data stored in the database and the application programs and queries submitted to the system.

The storage manager is responsible to the following tasks:

- interaction with the file manager
- efficient storing, retrieving and updating of data



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”

Chapter 2: Entity–Relationship Model

- Entity Sets
- Relationship Sets
- Design Issues
- Mapping Constraints
- Keys
- E–R Diagram
- Extended E–R Features
- Design of an E–R Database Schema
- Reduction of an E–R Schema to Tables

Entity Sets

- A *database* can be modeled as:
 - a collection of entities,
 - relationship among entities.
- An *entity* is an object that exists and is distinguishable from other objects.
 - Example: specific person, company, event, plant
- Entities have *attributes*
 - Example: people have *names* and *addresses*
- An *entity set* is a set of entities of the same type that share the same properties.
 - Example: set of all persons, companies, trees, holidays

Entity Sets *customer* and *loan*

customer-id	customer-name	customer-street	customer-city	loan-number	amount
-------------	---------------	-----------------	---------------	-------------	--------

321-12-3123	Jones	Main	Harrison		
-------------	-------	------	----------	--	--

019-28-3746	Smith	North	Rye		
-------------	-------	-------	-----	--	--

677-89-9011	Hayes	Main	Harrison		
-------------	-------	------	----------	--	--

555-55-5555	Jackson	Dupont	Woodside		
-------------	---------	--------	----------	--	--

244-66-8800	Curry	North	Rye		
-------------	-------	-------	-----	--	--

963-96-3963	Williams	Nassau	Princeton		
-------------	----------	--------	-----------	--	--

335-57-7991	Adams	Spring	Pittsfield		
-------------	-------	--------	------------	--	--

L-17	1000
------	------

L-23	2000
------	------

L-15	1500
------	------

L-14	1500
------	------

L-19	500
------	-----

L-11	900
------	-----

L-16	1300
------	------

customer

loan

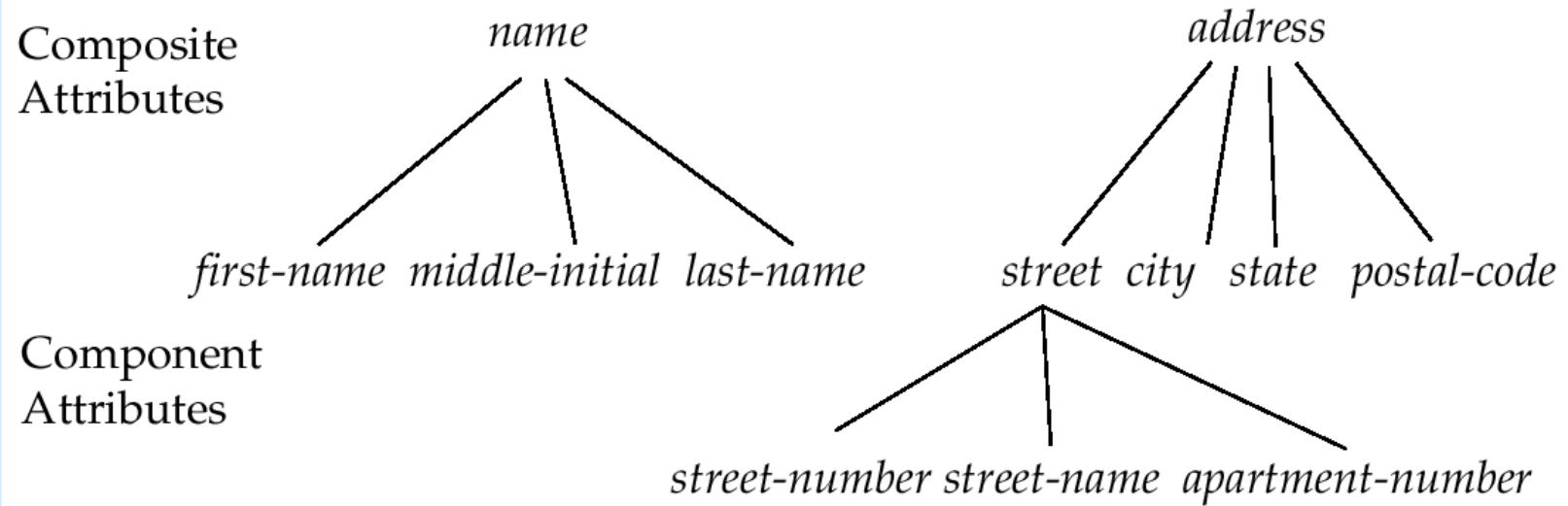
Attributes

- 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
 - E.g. multivalued attribute: phone-numbers
 - Derived attributes
 - Can be computed from other attributes
 - E.g. 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

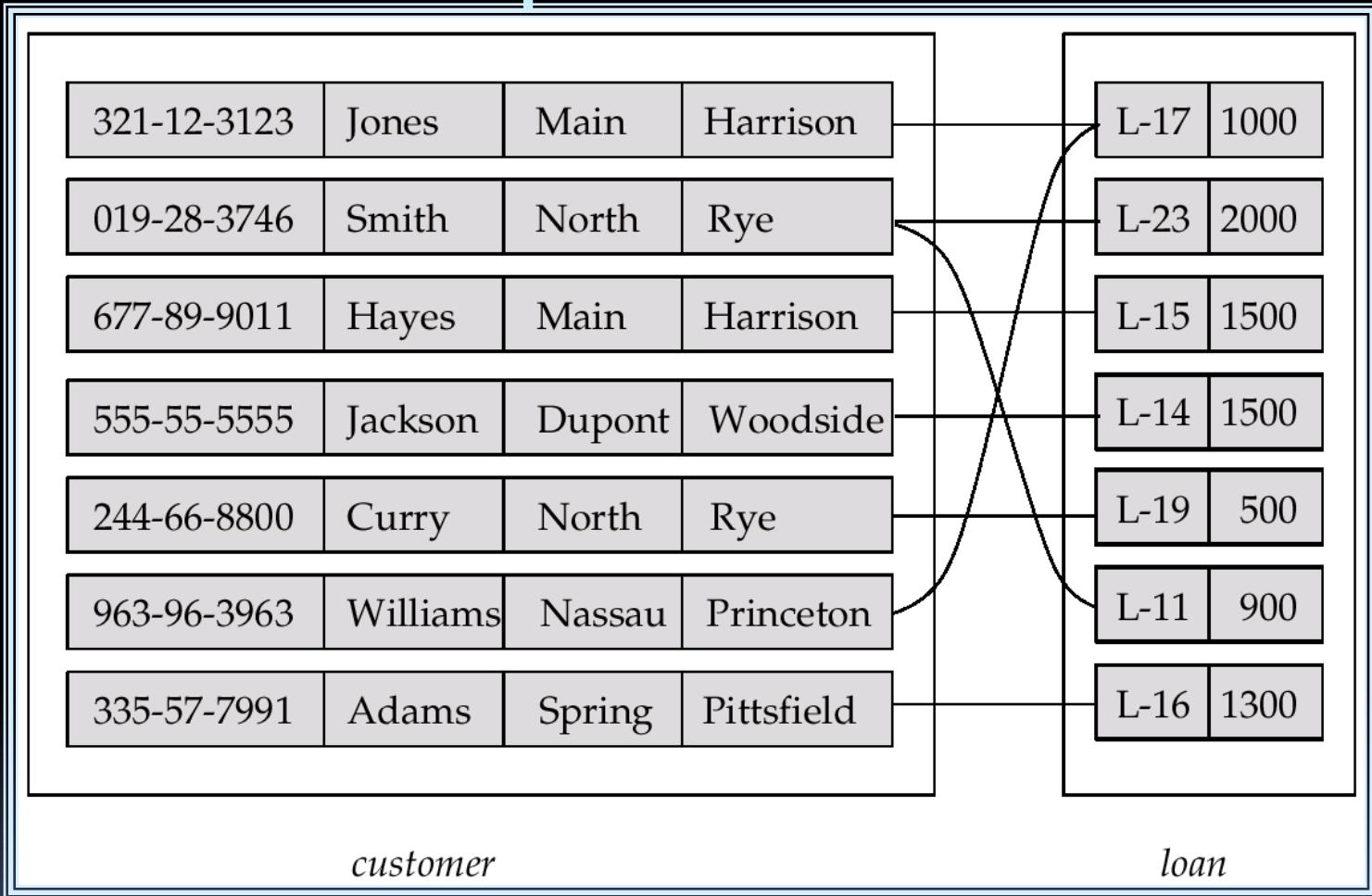
$$\{(e_1, e_2, \dots, e_n) \mid e_1 \in E_1, e_2 \in E_2, \dots, e_n \in E_n\}$$

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

- Example:

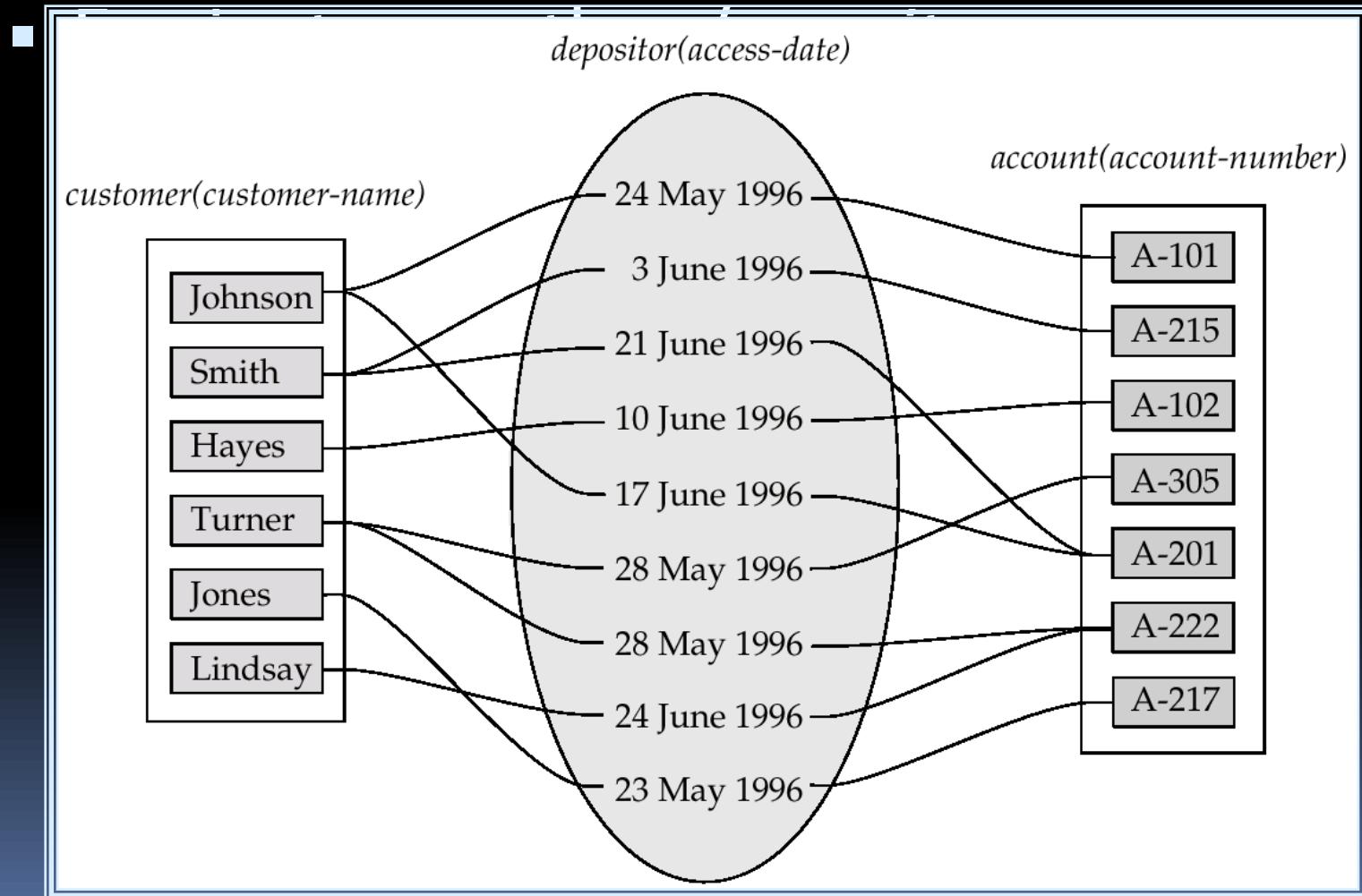
$(\text{Hayes}, \text{A-102}) \in \text{depositor}$

Relationship Set *borrower*



Relationship Sets (Cont.)

- An *attribute* can also be property of a relationship set.



Degree of a Relationship Set

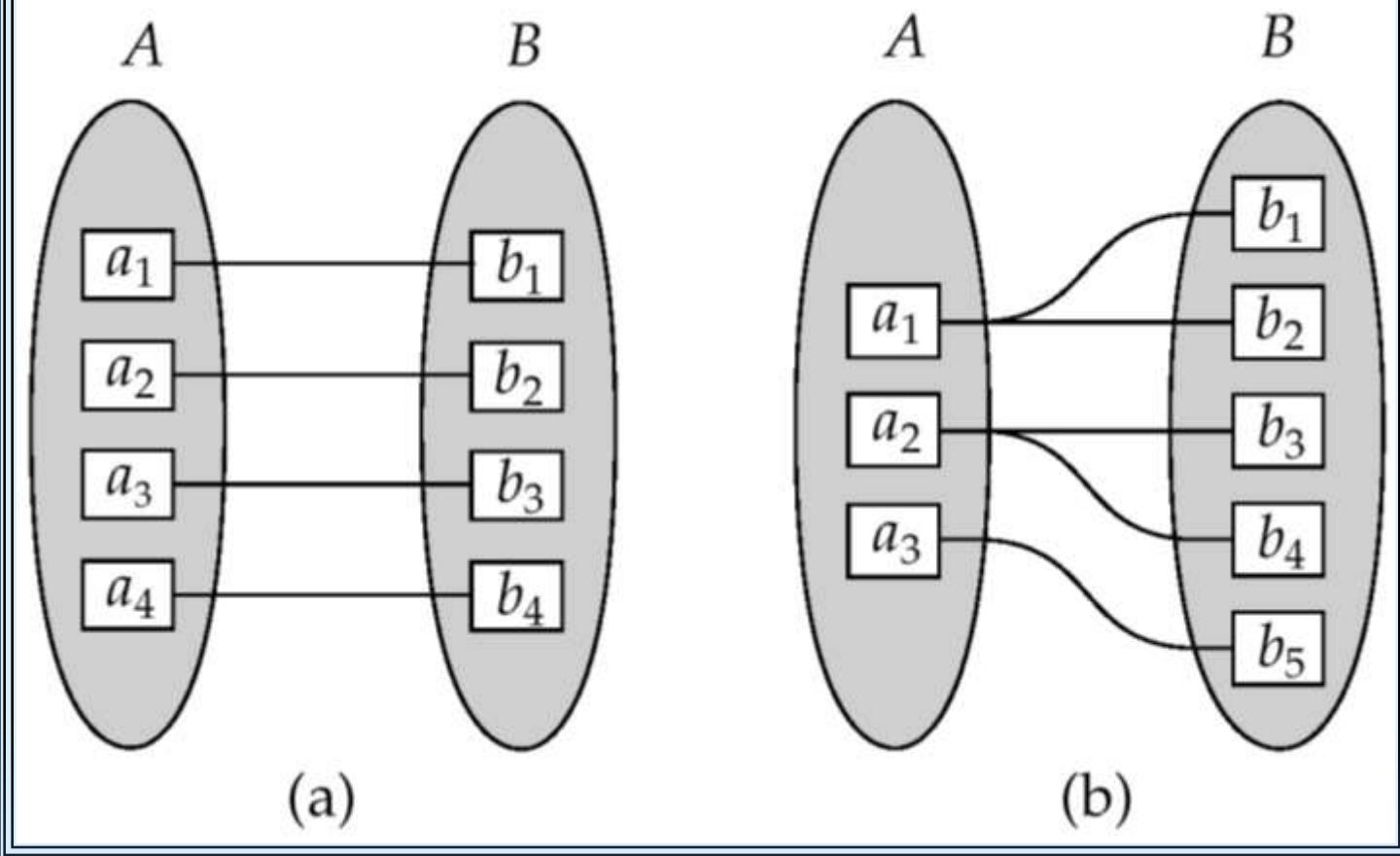
- Refers to number of entity sets that participate in a relationship set.
- Relationship sets that involve two entity sets are *binary* (or degree two). Generally, most relationship sets in a database system are binary.
- Relationship sets may involve more than two entity sets.

❖ E.g. 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*

Mapping Cardinalities

- 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

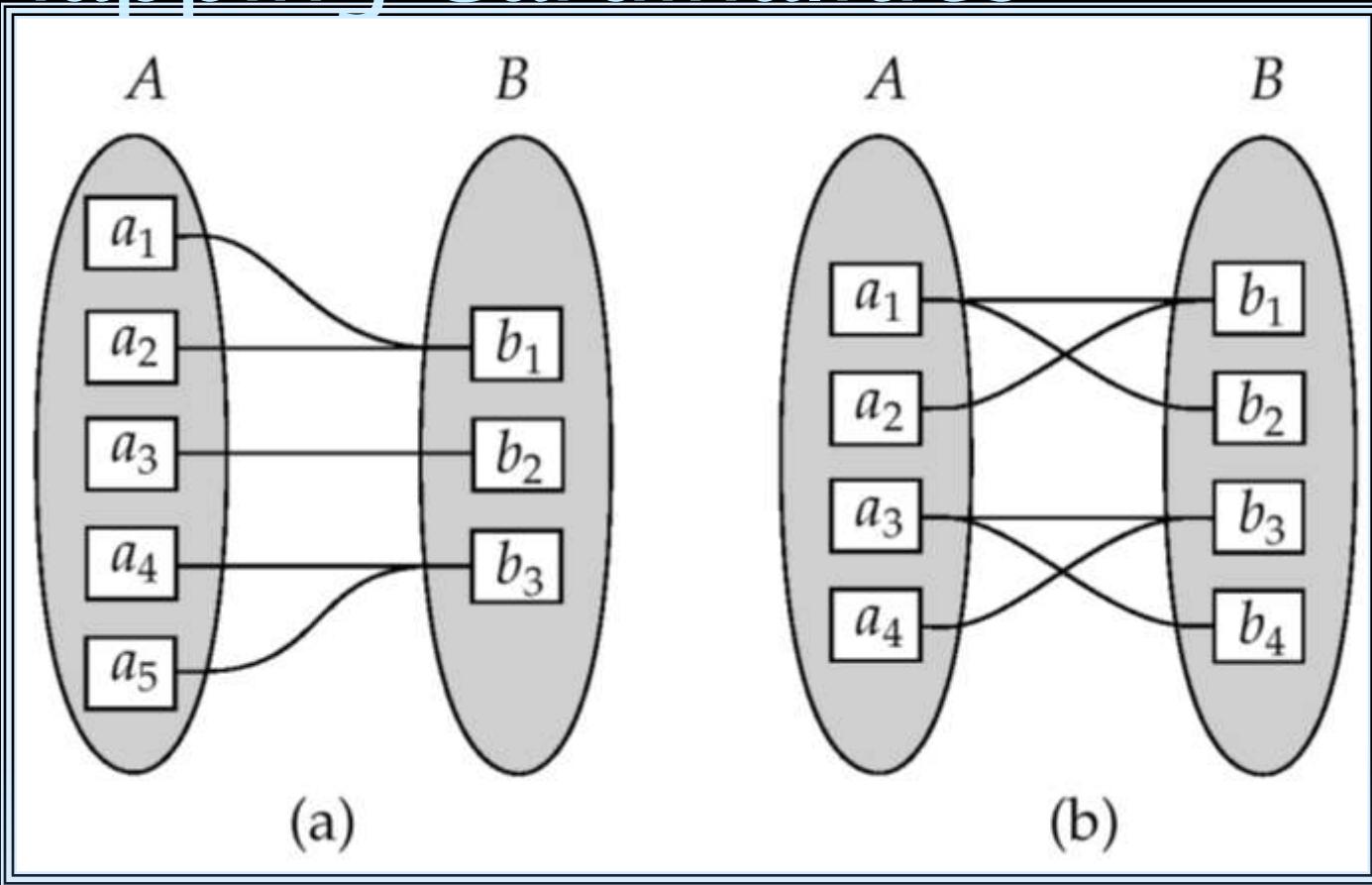


One to one

One to many

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

Mapping Cardinalities



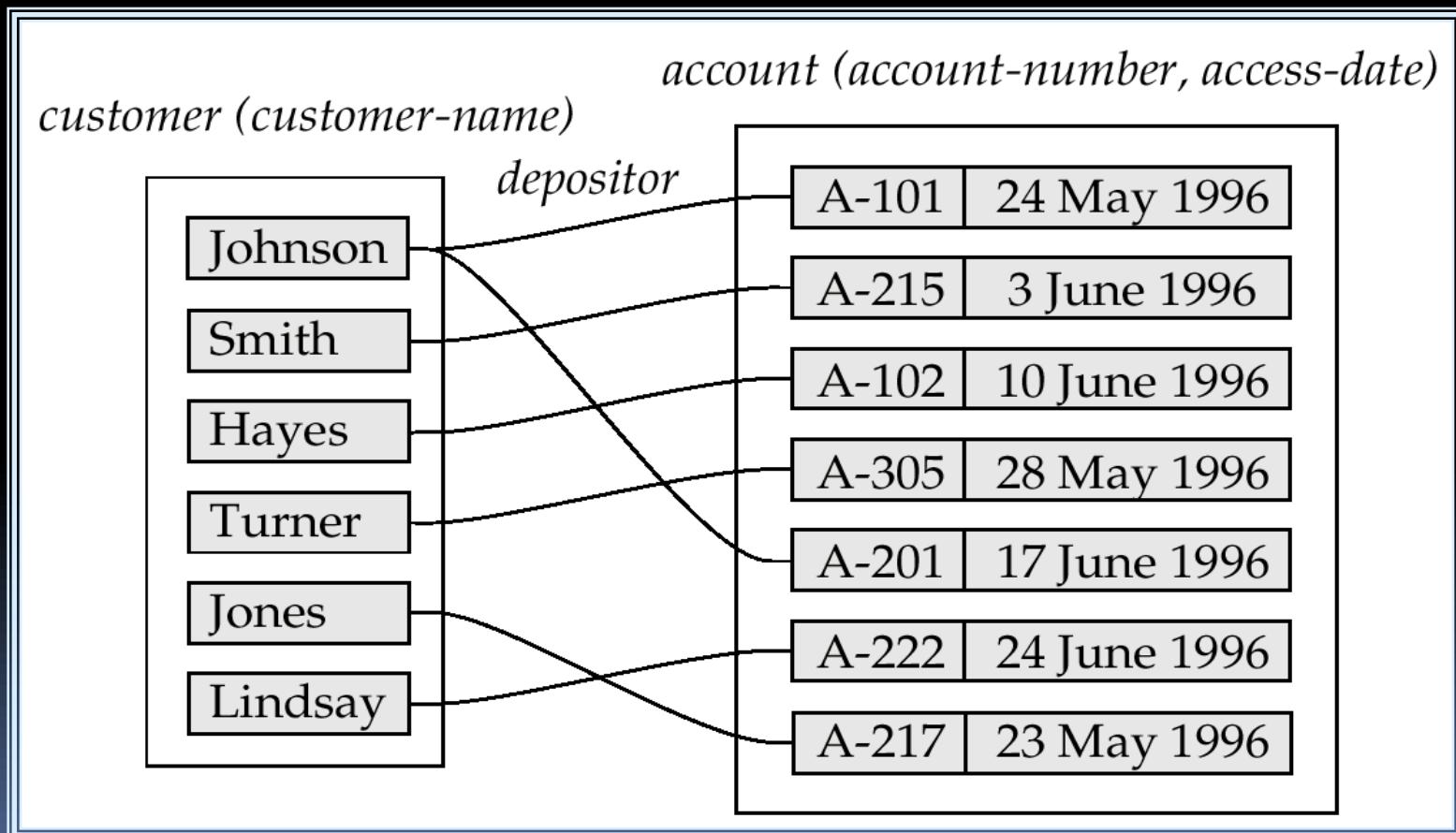
Many to one

Many to many

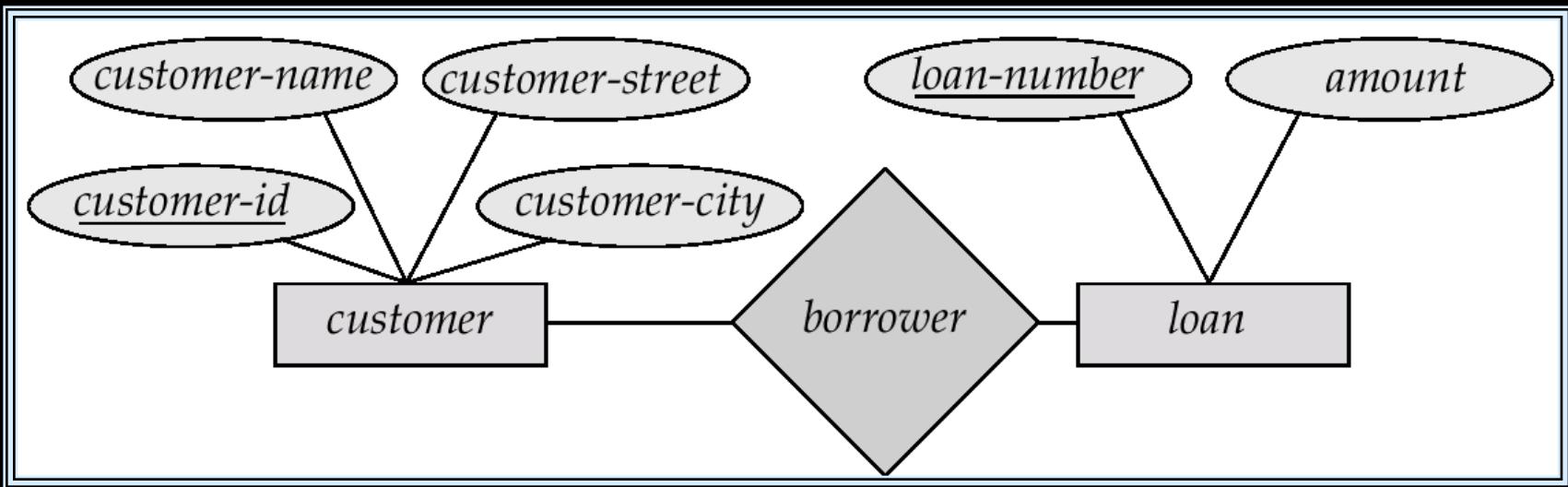
Note: Some elements in A and B may not be mapped to any elements in the other set

Mapping Cardinalities affect ER Design

- Can make *access-date* an attribute of account, instead of a relationship attribute, if each account can have only one customer
 - I.e., the relationship from account to customer is many to one, or equivalently, customer to account is one to many

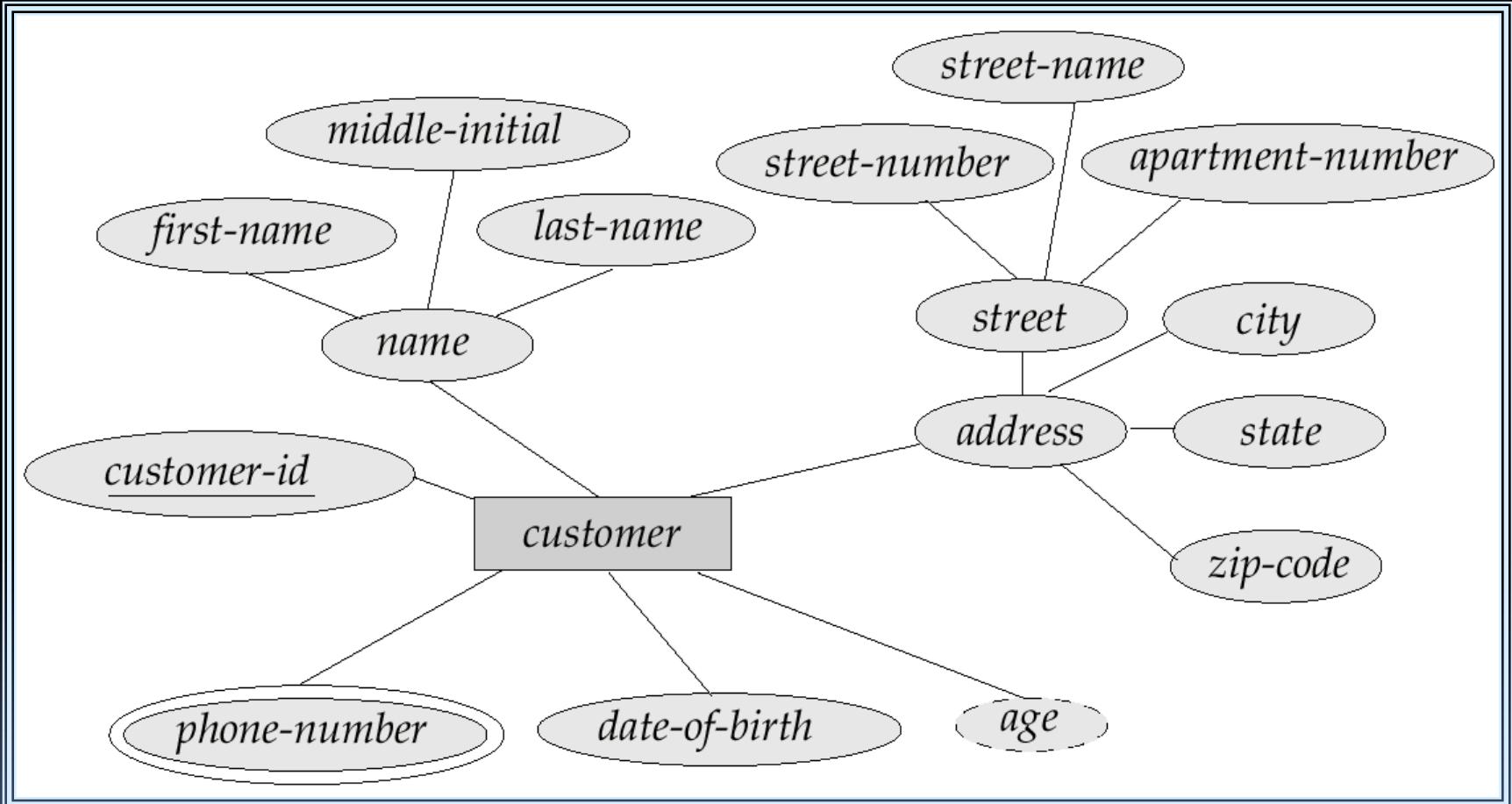


E-R Diagrams

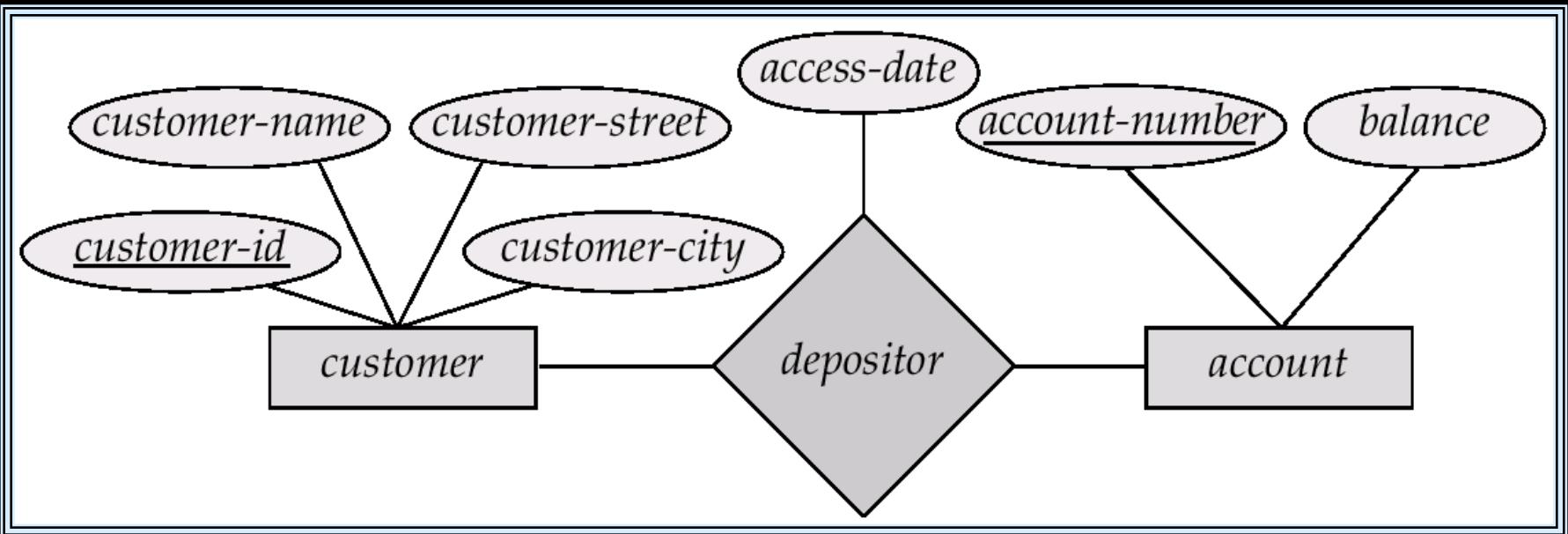


- **Rectangles** represent entity sets.
- **Diamonds** represent relationship sets.
- **Lines** link attributes to entity sets and entity sets to relationship sets.
- **Ellipses** represent attributes
 - **Double ellipses** represent multivalued attributes.
 - **Dashed ellipses** denote derived attributes.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

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

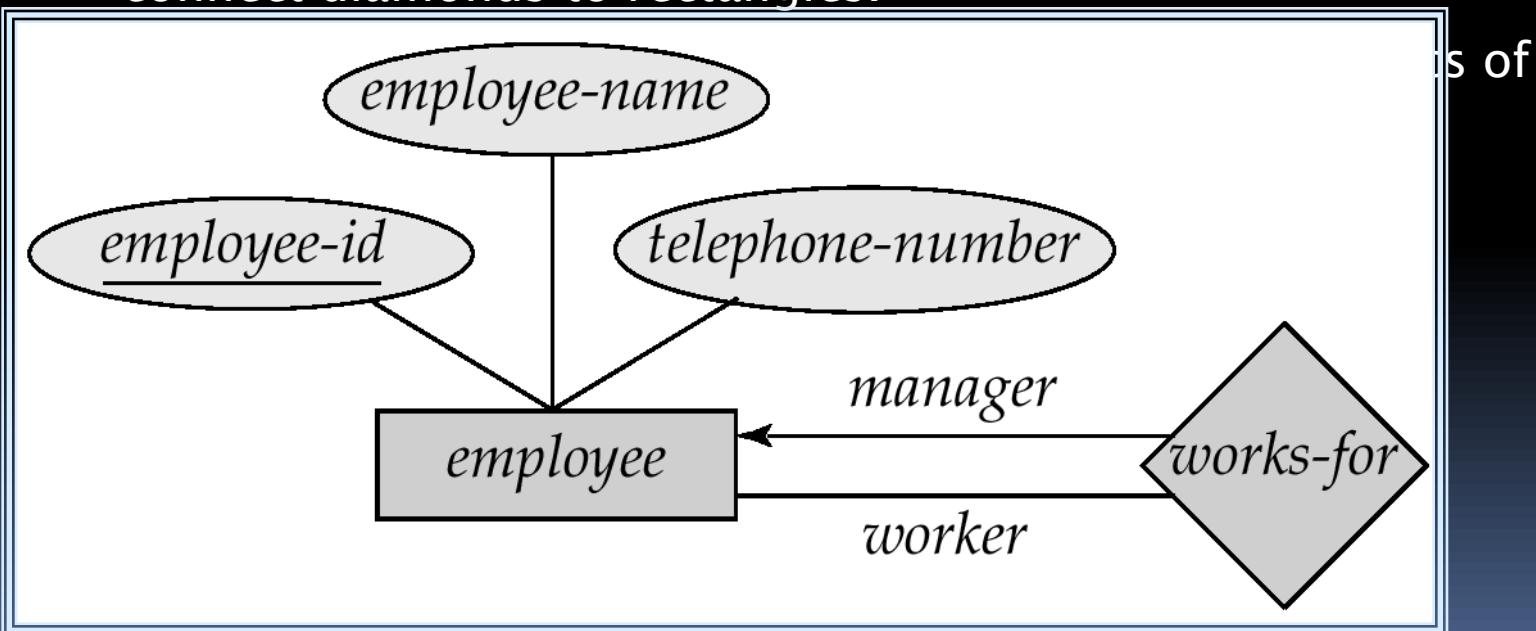


Relationship Sets with Attributes



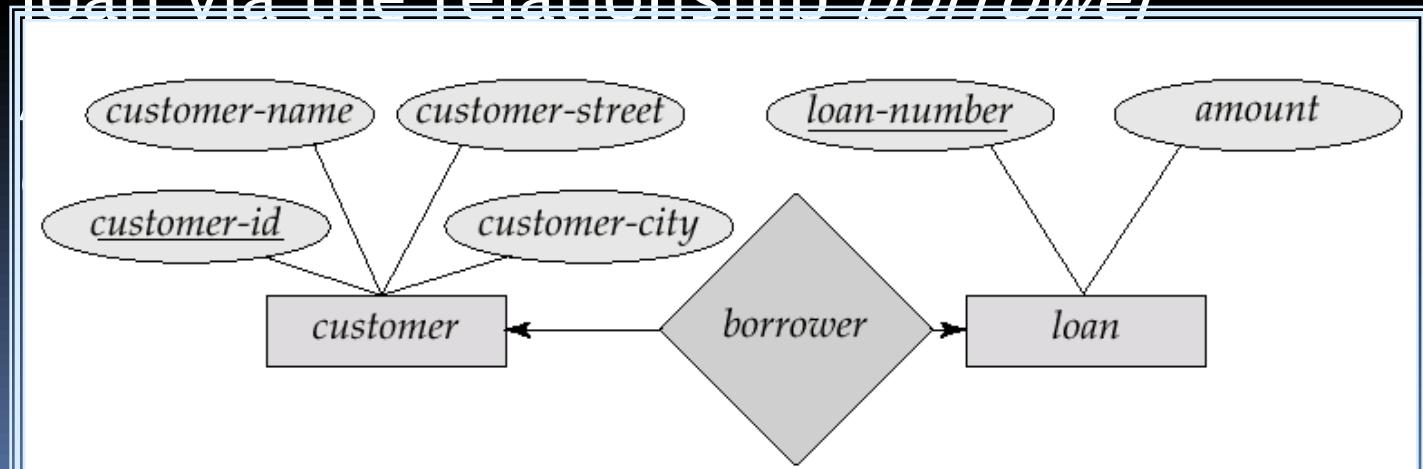
Roles

- 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.



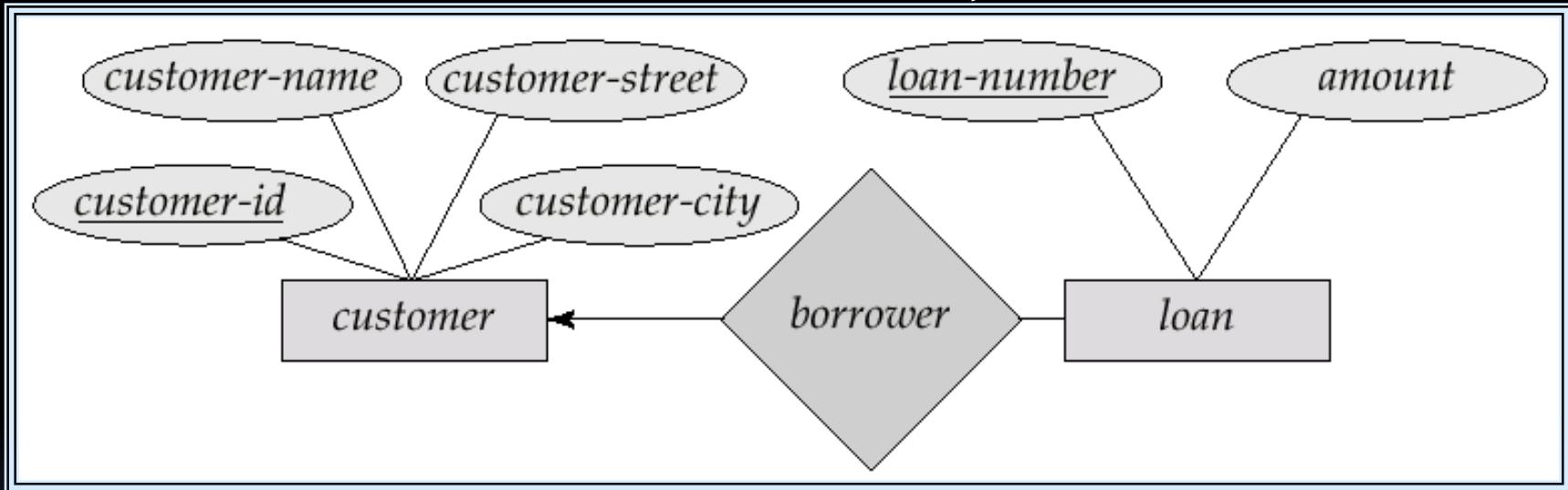
Cardinality Constraints

- We express cardinality constraints by drawing either a directed line (\rightarrow), signifying “one,” or an undirected line ($-$), signifying “many,” between the relationship set and the entity set.
- E.g.: One-to-one relationship:
 - A customer is associated with at most one loan via the relationship *borrower*



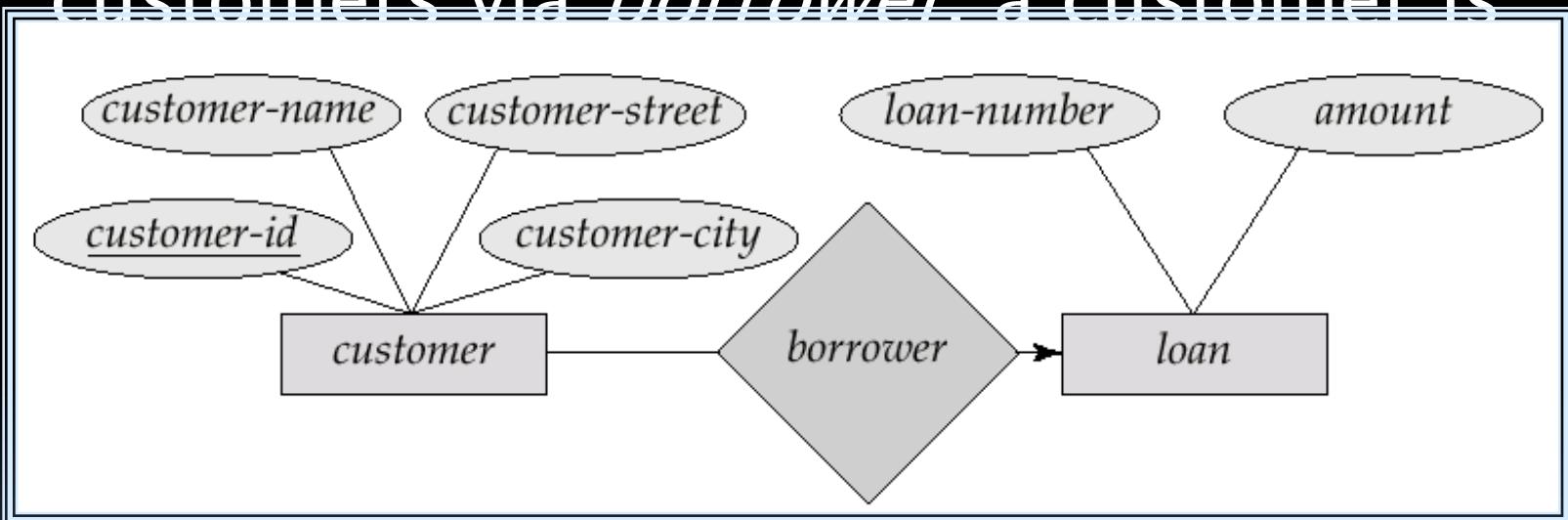
One-To-Many Relationship

- In the one-to-many relationship a loan is associated with at most one customer via *borrower*, a customer is

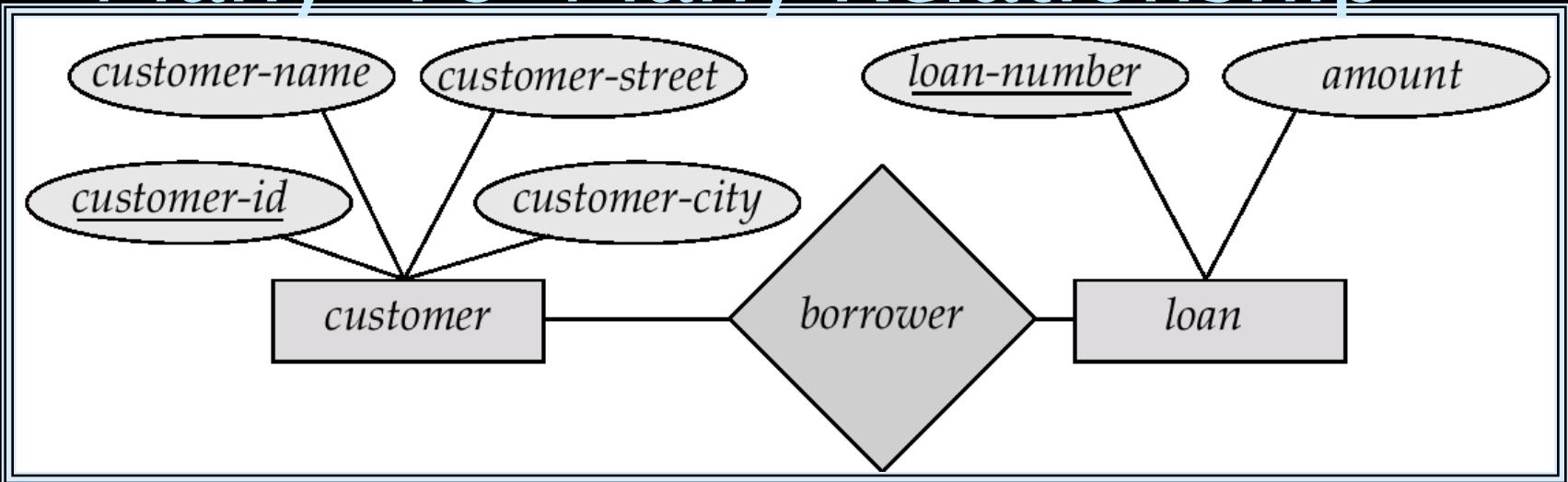


Many-To-One Relationships

- In a many-to-one relationship a loan is associated with several (including 0) ~~customers via *borrower*, a customer is~~



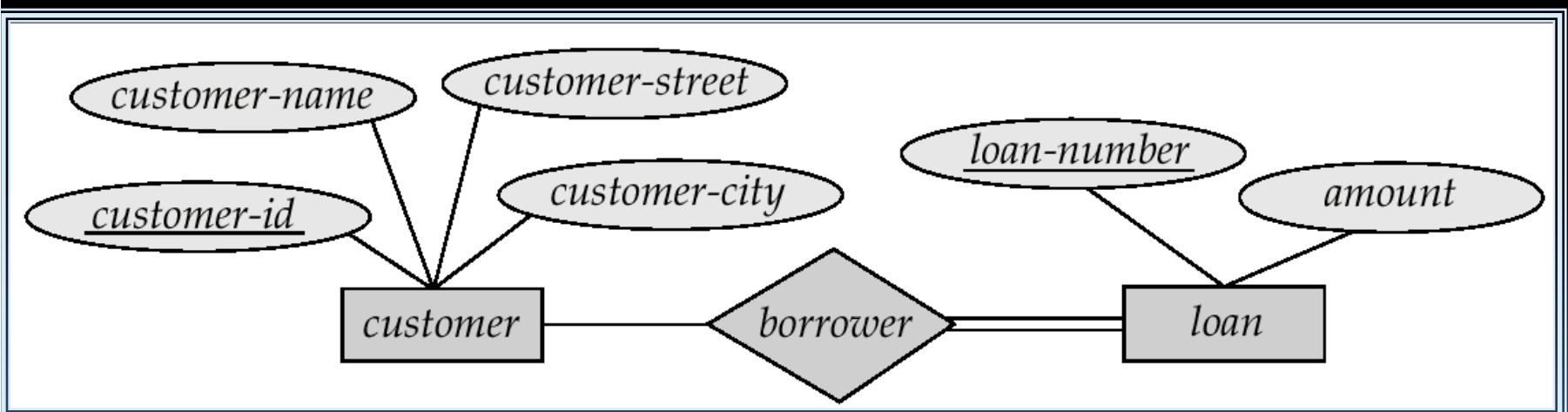
Many-To-Many Relationship



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

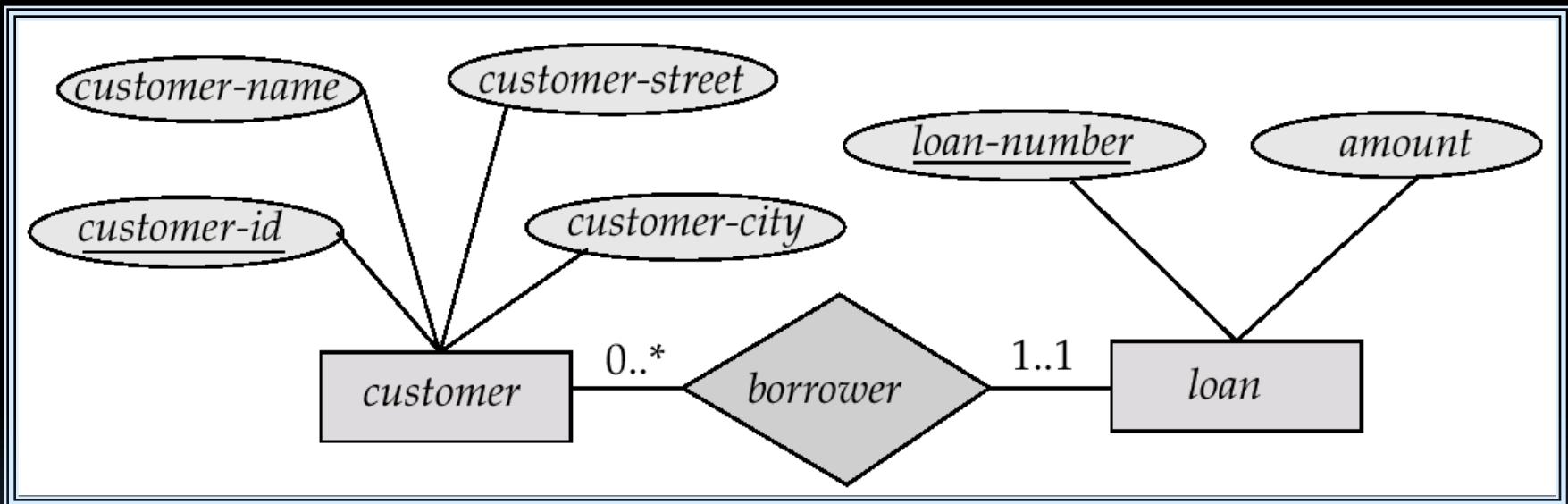
Participation of an Entity Set in

- 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
 - E.g. participation of *customer* in *borrower* is partial



Alternative Notation for Cardinality Limits

- Cardinality limits can also express participation constraints



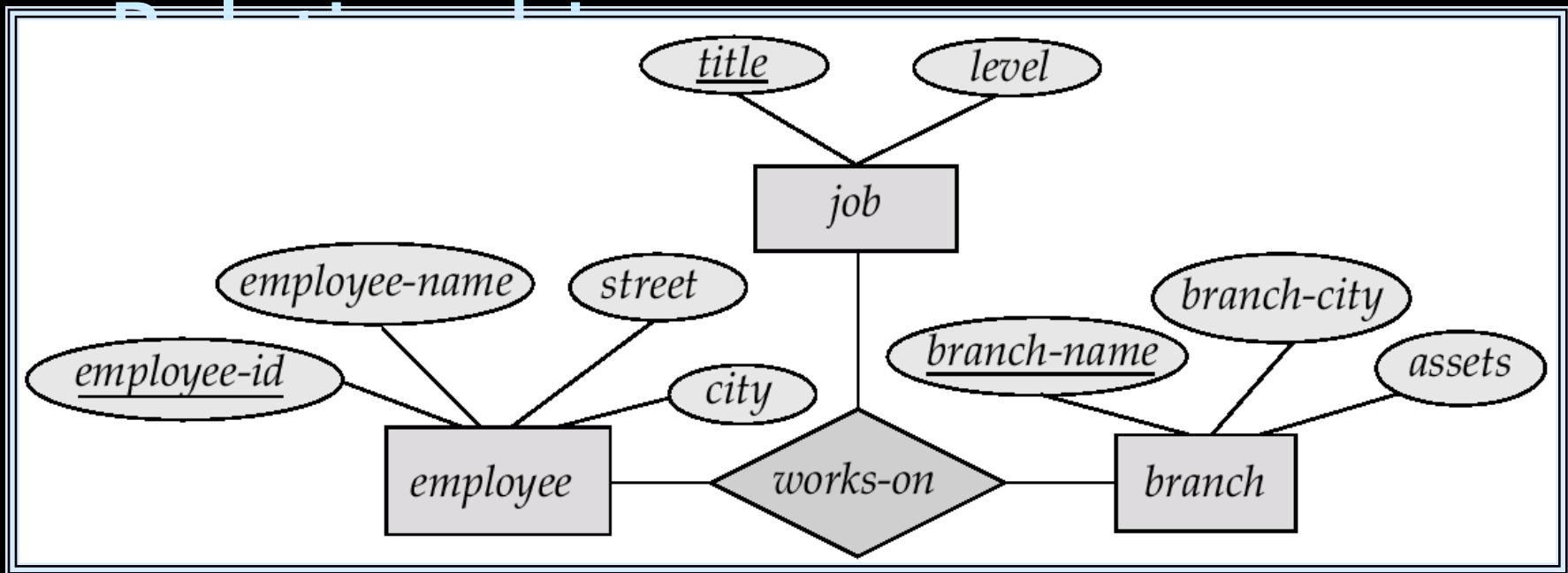
Keys

- A *super key* of an entity set is a set of one or more attributes whose values uniquely determine each entity.
- A *candidate key* of an entity set is a minimal super key
 - *Customer-id* is candidate key of *customer*
 - *account-number* is candidate key of *account*
- Although several candidate keys may exist, one of the candidate keys is selected to be the *primary key*.

Keys for Relationship Sets

- The combination of primary keys of the participating entity sets forms a super key of a relationship set.
 - $(customer\text{-}id, account\text{-}number)$ is the super key of *depositor*
 - *NOTE: this means a pair of entity sets can have at most one relationship in a particular relationship set.*
 - E.g. if we wish to track all access-dates to each account by each customer, we cannot assume a relationship for each access. We can use a multivalued attribute though

E-R Diagram with a Ternary



Cardinality Constraints on

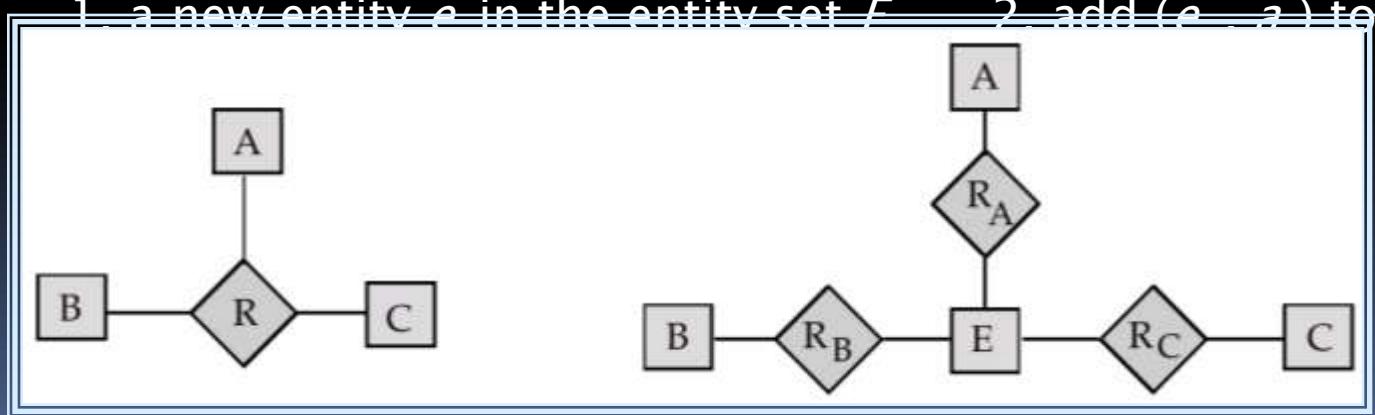
▪ **Ternary Relationship**

- We allow at most one arrow out of a ternary (or greater degree) relationship to indicate a cardinality constraint
- E.g. an arrow from *works-on* to *job* indicates each employee works on at most one job at any branch.
- If there is more than one arrow, there are two ways of defining the meaning.
 - E.g a ternary relationship R between A , B and C with arrows to B and C could mean
 - 1. each A entity is associated with a unique entity from B and C or

- **Binary Vs. Non-Binary Relationships**
 - Some relationships that appear to be non-binary may be better represented using binary relationships
 - E.g. A ternary relationship *parents*, relating a child to his/her father and mother, is best replaced by two binary relationships, *father* and *mother*
 - Using two binary relationships allows partial information (e.g. only mother being known)
 - But there are some relationships that are naturally non-binary
 - E.g. *works-on*

Converting Non-Binary Relationships to Binary Form

- In general, any non-binary relationship can be represented using binary relationships by creating an artificial entity set.
 - Replace R between entity sets A , B and C by an entity set E , and three relationship sets:
 1. R_A , relating E and A
 2. R_B , relating E and B
 3. R_C , relating E and C
 - Create a special identifying attribute for E
 - Add any attributes of R to E
 - For each relationship (a_i, b_i, c_i) in R , create
 - 1. a new entity e in the entity set E
 - 2. add $(e \rightarrow a_i)$ to R_A



Converting Non-Binary Relationships (Cont.)

- Also need to translate constraints
 - Translating all constraints may not be possible
 - There may be instances in the translated schema that cannot correspond to any instance of R
 - *Exercise: add constraints to the relationships R_A , R_B and R_C to ensure that a newly created entity corresponds to exactly one entity in each of entity sets A , B and C*
 - We can avoid creating an identifying attribute by making E a weak entity set (described shortly) identified by the three relationship sets

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

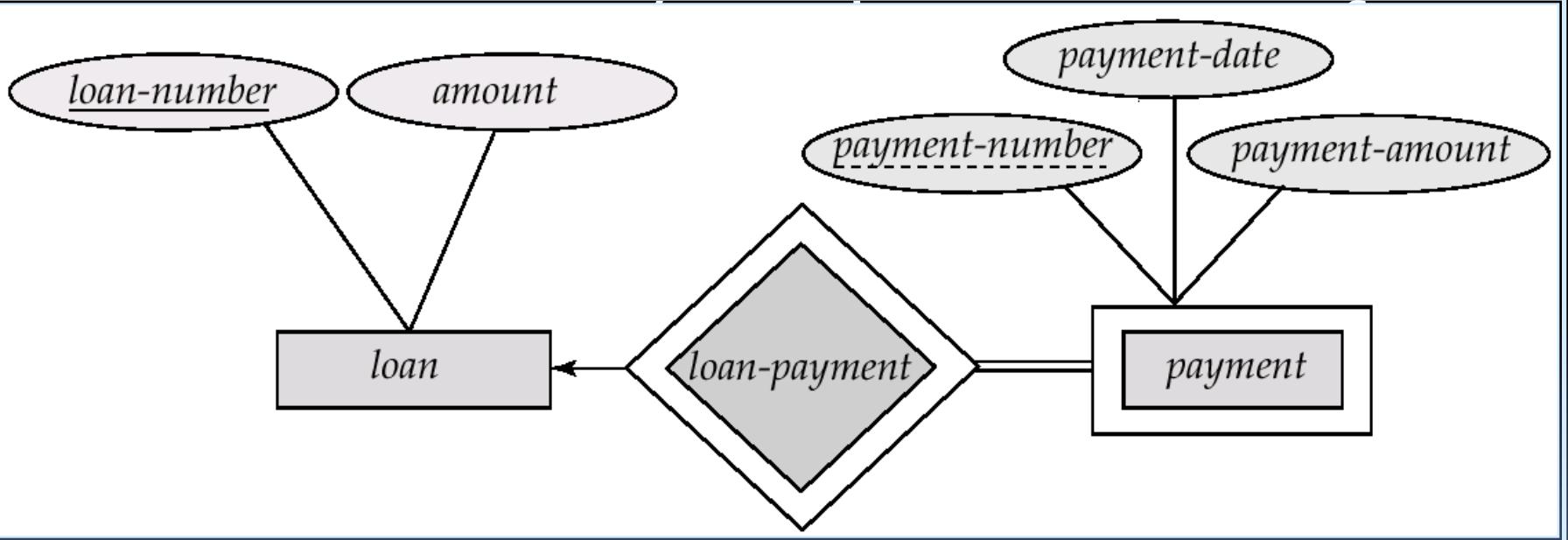
HOW ABOUT DOING
AN ER DESIGN
INTERACTIVELY ON
THE BOARD?
SUGGEST AN
APPLICATION TO BE
MODELED.

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

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.



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

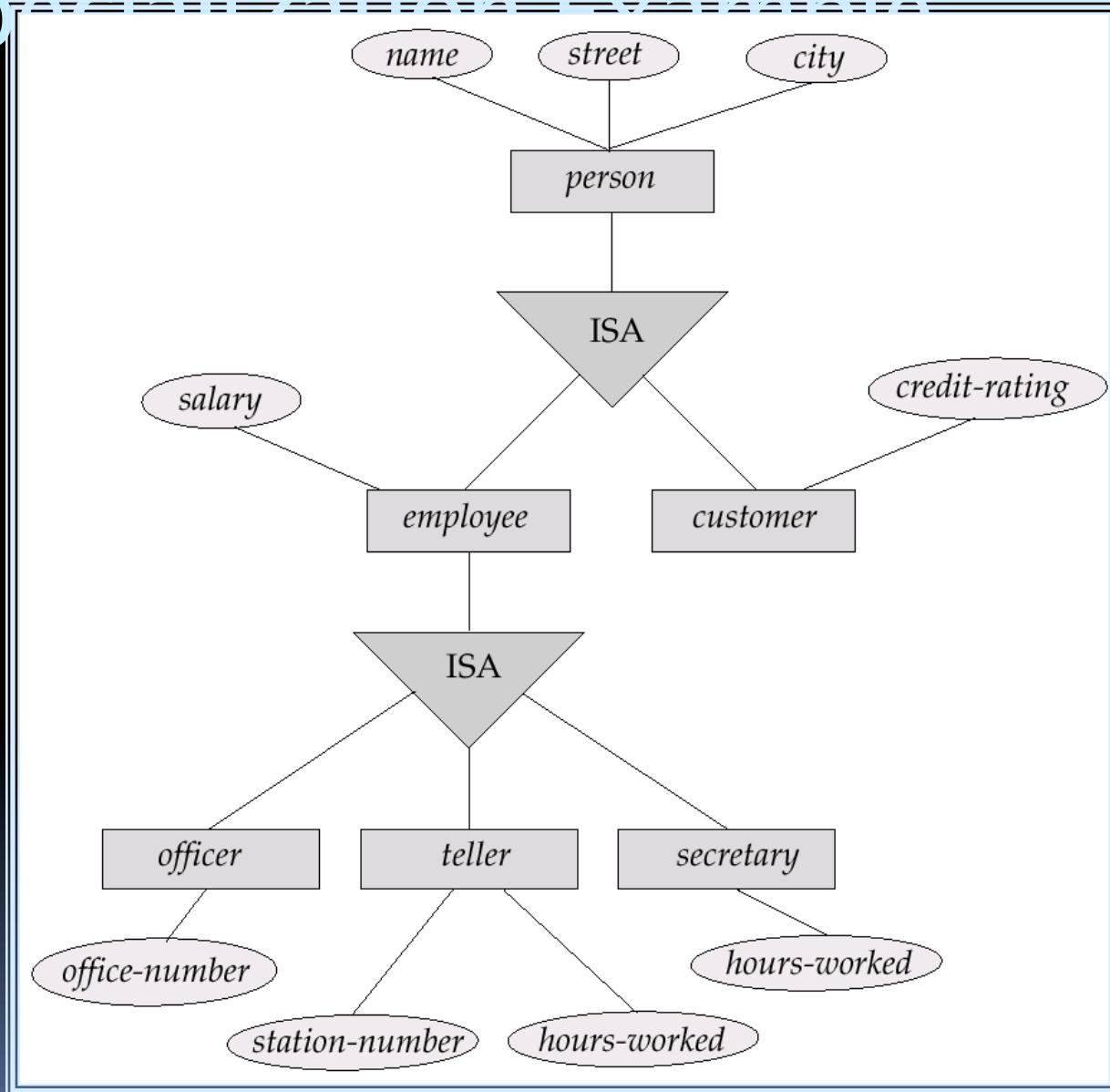
- In a university, a *course* is a strong entity and a *course-offering* can be modeled as a weak entity
- The discriminator of *course-offering* would be *semester* (including year) and *section-number* (if there is more than one section)
- If we model *course-offering* as a strong entity we would model *course-number* as an attribute.

Then the relationship with *course* would be implicit in the *course-number*

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

Specialization Example



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 (Contd.)

- 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

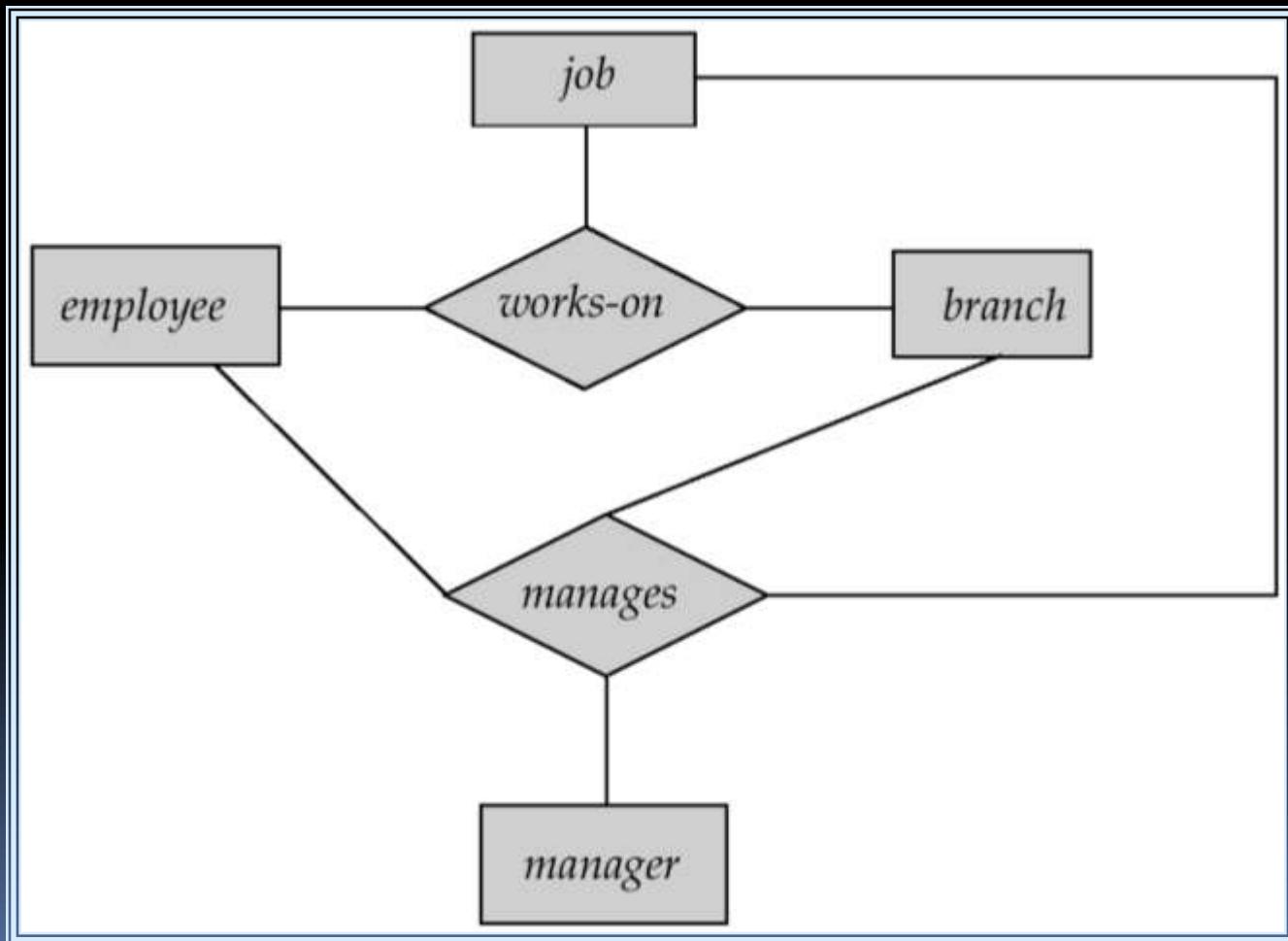
- Constraint on which entities can be members of a given lower-level entity set.
 - condition-defined
 - E.g. all customers over 65 years are members of *senior-citizen* entity set; *senior-citizen* ISA *person*.
 - user-defined
- Constraint on whether or not entities may belong to more than one lower-level entity set within a single generalization.
 - Disjoint
 - an entity can belong to only one lower-

Design Constraints on a Specialization/Generalization

- (Contd.) 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

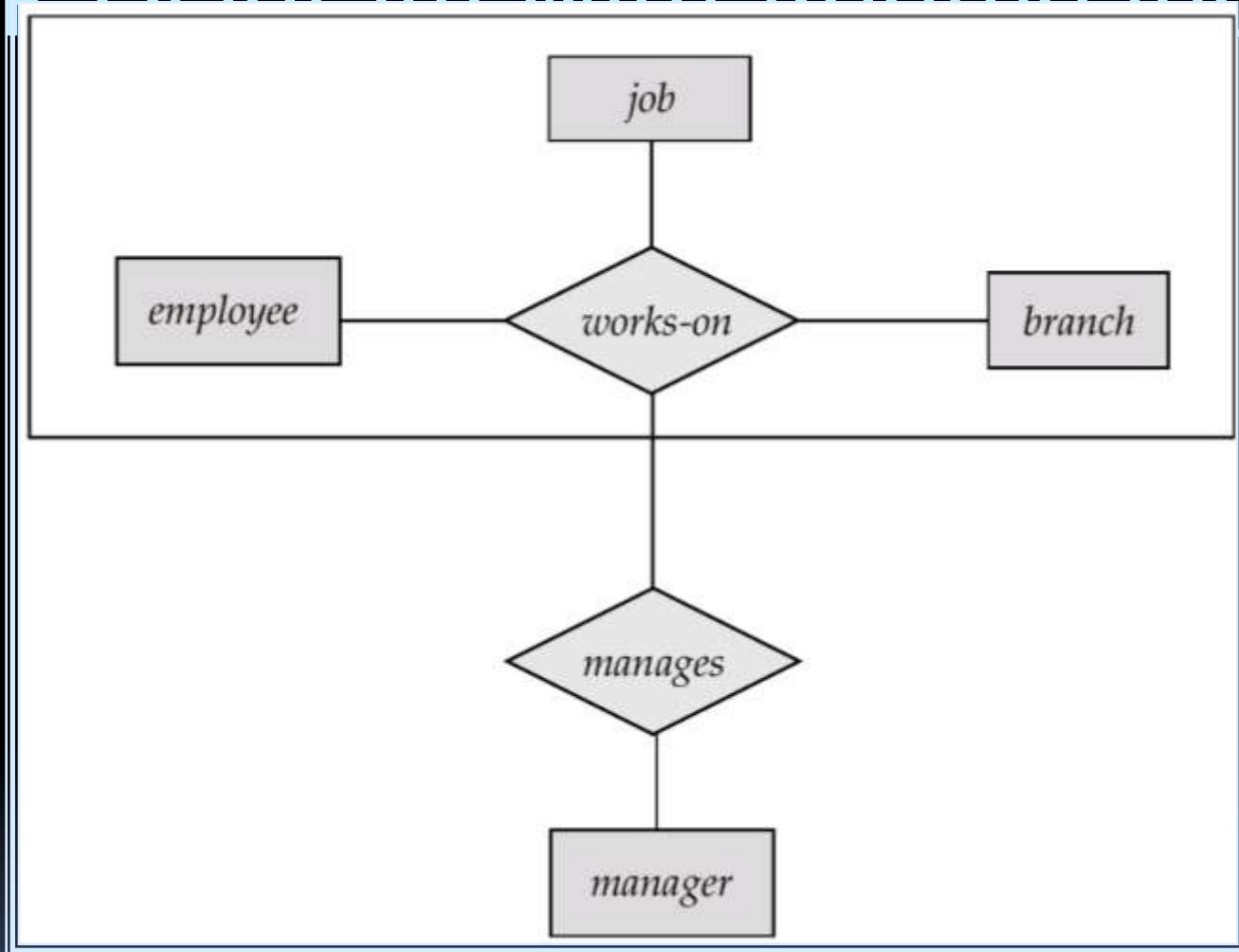
- Consider the ternary relationship *works-on*, which we saw earlier
- Suppose we want to record managers for tasks performed by an employee at a branch



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:

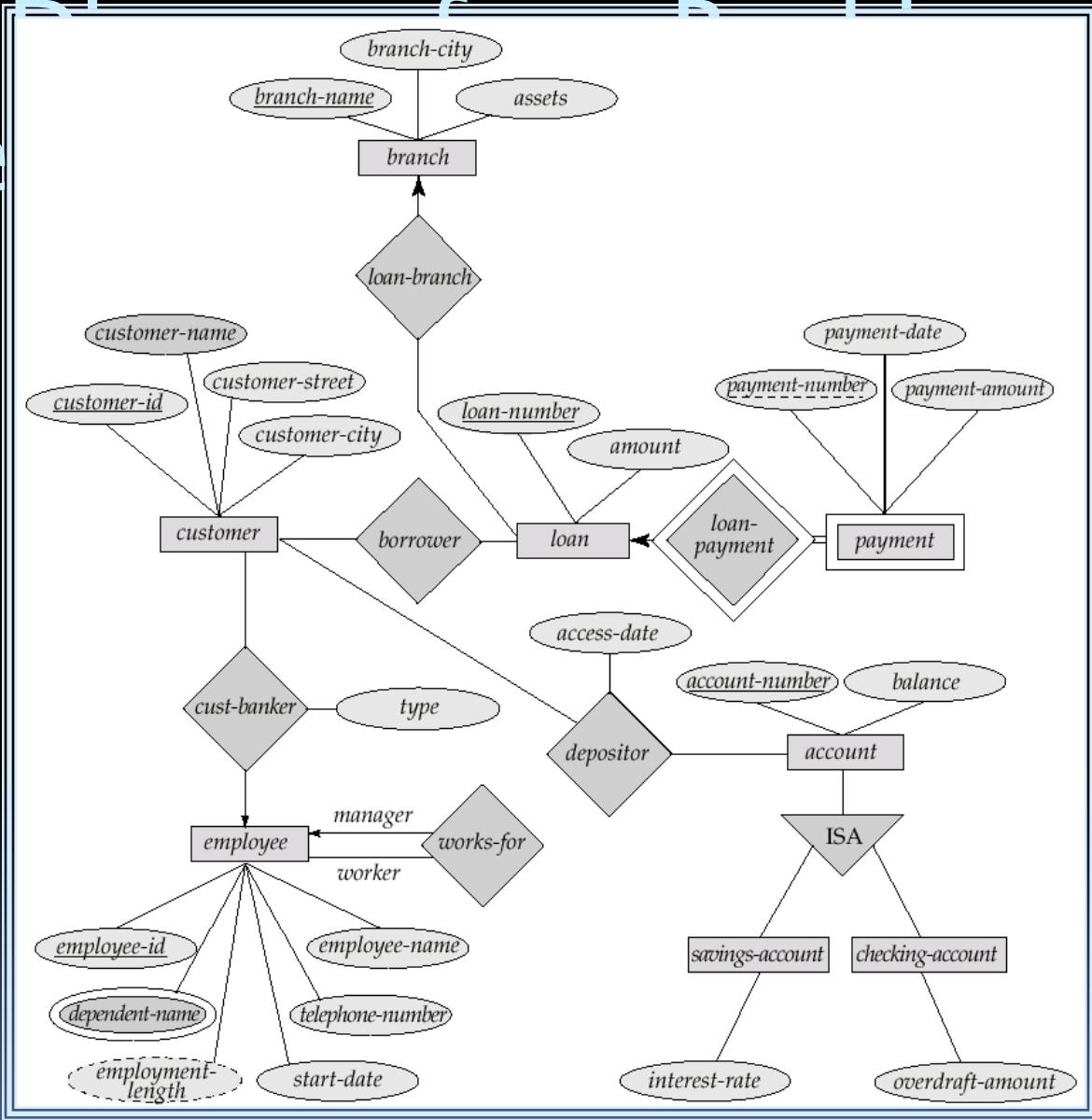
E-R Diagram With Aggregation



E-R Design Decisions

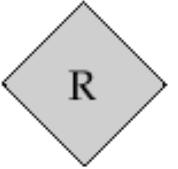
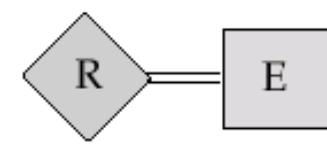
- The use of an attribute or entity set to represent an object.
- Whether a real-world concept is best expressed by an entity set or a relationship set.
- The use of a ternary relationship versus a pair of binary relationships.
- The use of a strong or weak entity set.
- The use of specialization/generalization – inheritance and its variants

E-R Enter

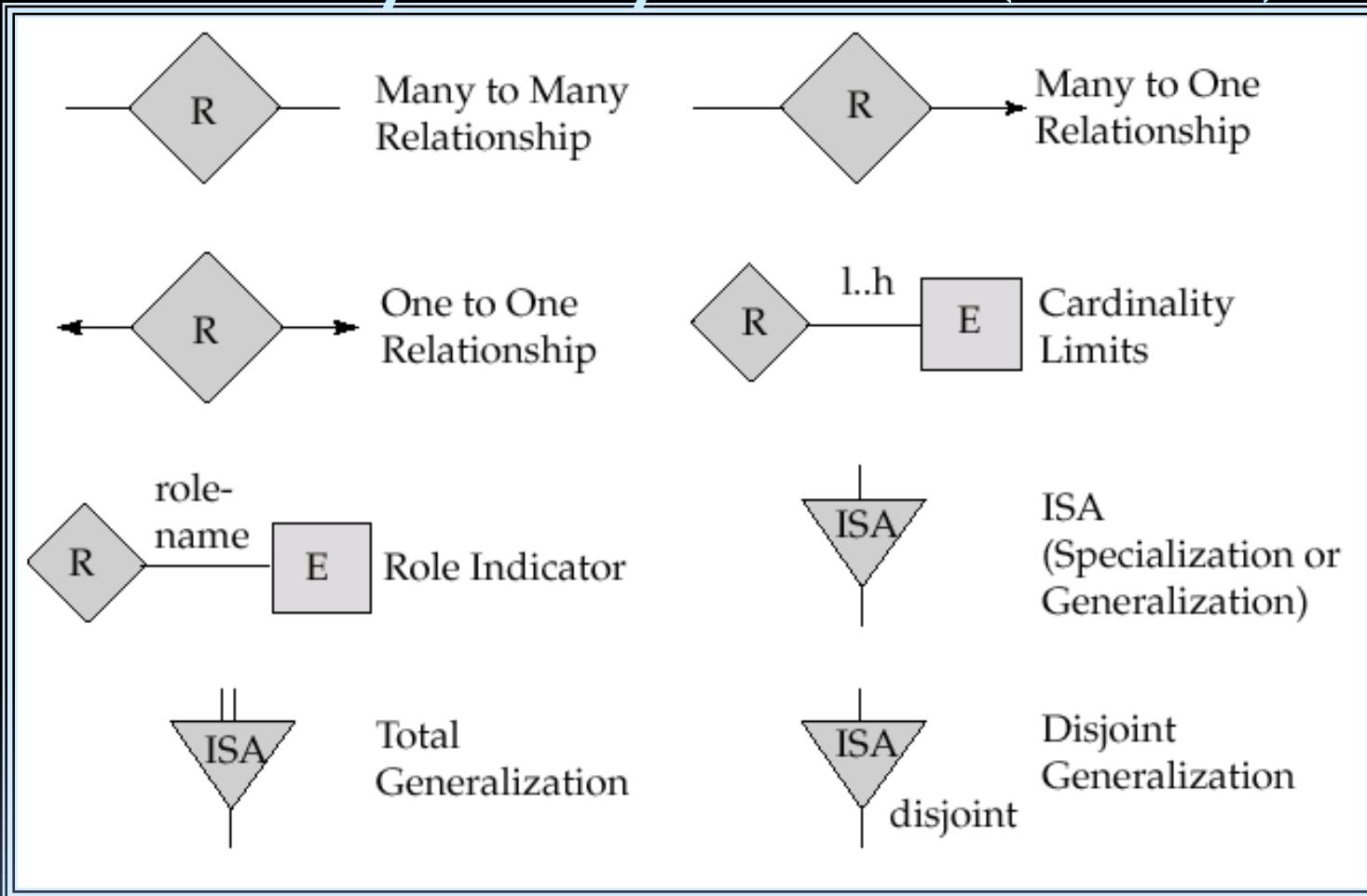


HOW ABOUT DOING
ANOTHER ER DESIGN
INTERACTIVELY ON
THE BOARD?

Summary of Symbols Used in E-R Notation

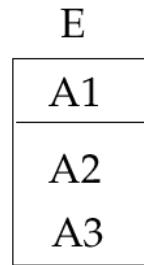
	Entity Set		Attribute
	Weak Entity Set		Multivalued Attribute
	Relationship Set		Derived Attribute
	Identifying Relationship Set for Weak Entity Set		Total Participation of Entity Set in Relationship
	Primary Key		Discriminating Attribute of Weak Entity Set

Summary of Symbols (Cont.)

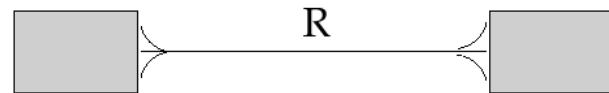
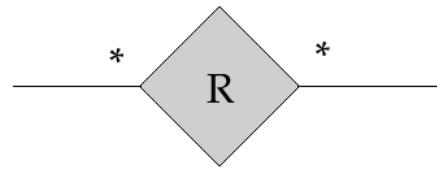


Alternative E-R Notations

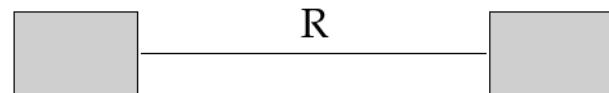
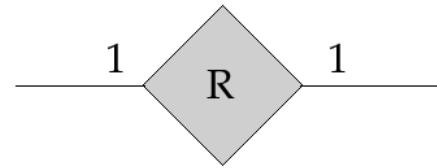
Entity set E with
attributes A1, A2, A3
and primary key A1



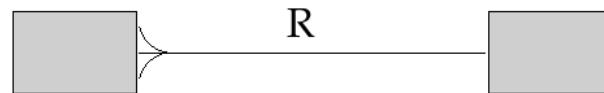
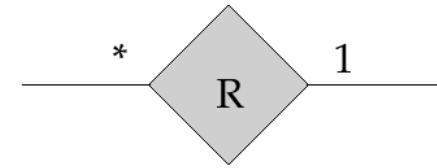
Many to Many
Relationship



One to One
Relationship



Many to One
Relationship

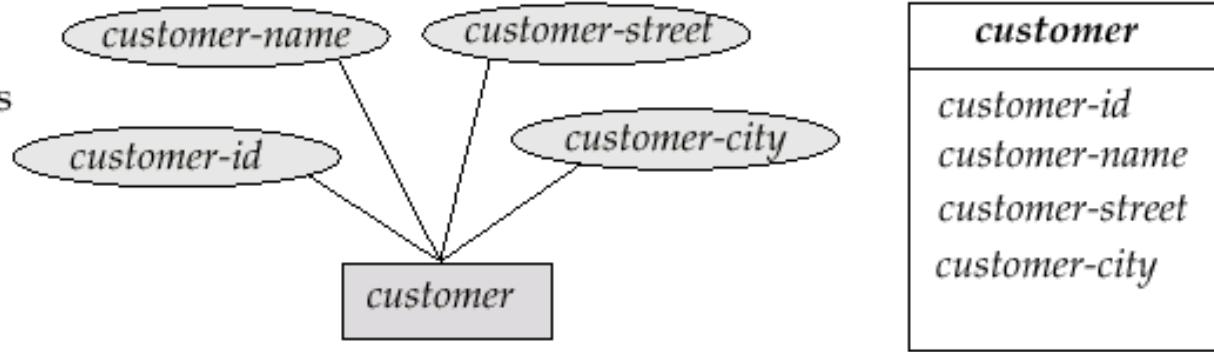


UML

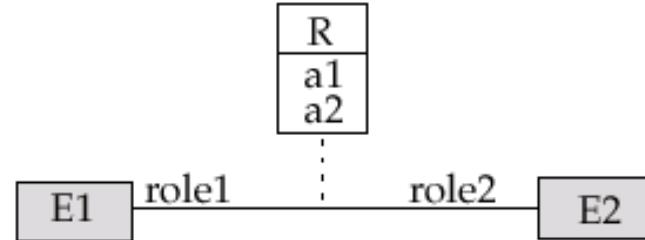
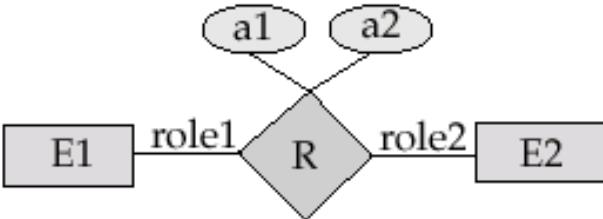
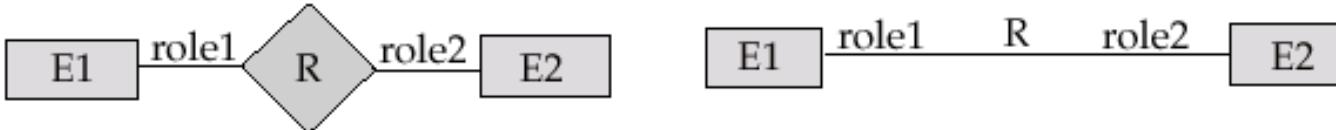
- UML: Unified Modeling Language
- UML has many components to graphically model different aspects of an entire software system
- UML Class Diagrams correspond to E-R Diagram, but several differences.

Summary of UML Class Diagram Notation

1. Entity sets and attributes



2. Relationships

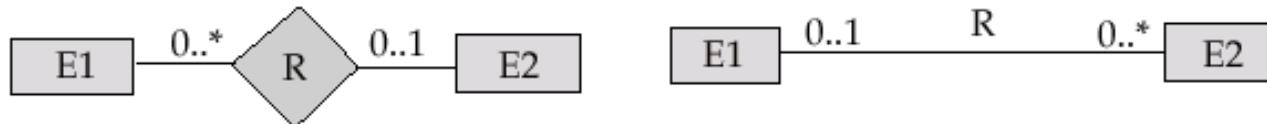


UML Class Diagrams (Contd.)

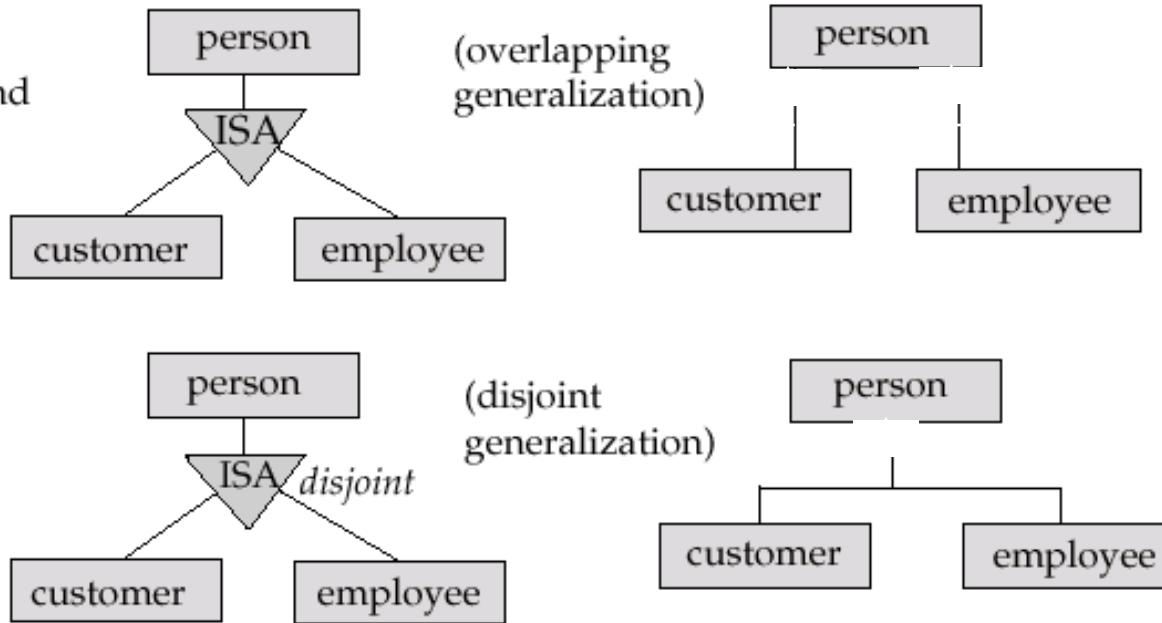
- Entity sets are shown as boxes, and attributes are shown within the box, rather than as separate ellipses in E-R diagrams.
- Binary relationship sets are represented in UML by just drawing a line connecting the entity sets. The relationship set name is written adjacent to the line.
- The role played by an entity set in a relationship set may also be specified by writing the role name on the line, adjacent to the entity set.

UML Class Diagram Notation

3. Cardinality constraints



4. Generalization and Specialization



*Note reversal of position in cardinality constraint depiction

*Generalization can use merged or separate arrows independent of disjoint/overlapping

UML Class Diagrams (Contd.)

- Cardinality constraints are specified in the form $/..h$, where $/$ denotes the minimum and h the maximum number of relationships an entity can participate in.
- Beware: the positioning of the constraints is exactly the reverse of the positioning of constraints in E-R diagrams.
- The constraint $0..*$ on the $E2$ side and $0..1$ on the $E1$ side means that each $E2$ entity can participate in at most one relationship, whereas each $E1$ entity can participate in many relationships; in other

Reduction of an E-R Schema to Tables

- Primary keys allow entity sets and relationship sets to be expressed uniformly as *tables* which represent the contents of the database.
- A database which conforms to an E-R diagram can be represented by a collection of tables.
- For each entity set and relationship set there is a unique table which is assigned the name of the corresponding entity set or

Representing Entity Sets as Tables

A strong entity set reduces to a table

<i>customer-id</i>	<i>customer-name</i>	<i>customer-street</i>	<i>customer-city</i>
019-28-3746	Smith	North	Rye
182-73-6091	Turner	Putnam	Stamford
192-83-7465	Johnson	Alma	Palo Alto
244-66-8800	Curry	North	Rye
321-12-3123	Jones	Main	Harrison
335-57-7991	Adams	Spring	Pittsfield
336-66-9999	Lindsay	Park	Pittsfield
677-89-9011	Hayes	Main	Harrison
963-96-3963	Williams	Nassau	Princeton

Composite and Multivalued

- Composite attributes are flattened out by creating a separate attribute for each component attribute
 - E.g. given entity set *customer* with composite attribute *name* with component attributes *first-name* and *last-name* the table corresponding to the entity set has two attributes
name.first-name and *name.last-name*
- A multivalued attribute M of an entity E is represented by a separate table EM
 - Table EM has attributes corresponding to the primary key of E and an attribute corresponding to multivalued attribute M

Representing Weak Entity Sets

- A weak entity set becomes a table that includes a column for the primary key of the identifying strong entity set

<i>loan-number</i>	<i>payment-number</i>	<i>payment-date</i>	<i>payment-amount</i>
L-11	53	7 June 2001	125
L-14	69	28 May 2001	500
L-15	22	23 May 2001	300
L-16	58	18 June 2001	135
L-17	5	10 May 2001	50
L-17	6	7 June 2001	50
L-17	7	17 June 2001	100
L-23	11	17 May 2001	75
L-93	103	3 June 2001	900
L-93	104	13 June 2001	200

Representing Relationship Sets as Tables

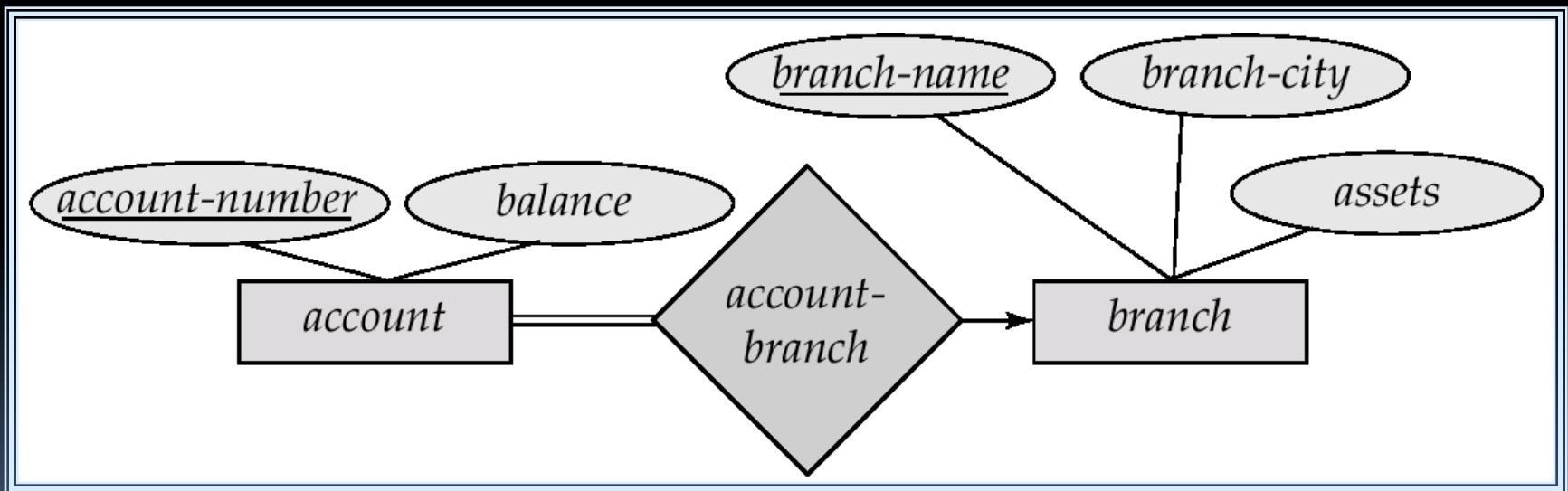
A many-to-many relationship set is represented as a table with columns for the primary keys of the two participating entity sets, and any descriptive attributes of the relationship set.

- E.g.: table for relationship set *borrower*

<i>customer-id</i>	<i>loan-number</i>
019-28-3746	L-11
019-28-3746	L-23
244-66-8800	L-93
321-12-3123	L-17
335-57-7991	L-16
555-55-5555	L-14
677-89-9011	L-15
963-96-3963	L-17

Redundancy of Tables

- Many-to-one and one-to-many relationship sets that are total on the many-side can be represented by adding an extra attribute to the many side, containing the primary key of the one side
- E.g.: Instead of creating a table for relationship *account-branch*, add an attribute *branch* to the entity set *account*



Redundancy of Tables (Cont.)

- For one-to-one relationship sets, either side can be chosen to act as the “many” side
 - That is, extra attribute can be added to either of the tables corresponding to the two entity sets
- If participation is *partial* on the many side, replacing a table by an extra attribute in the relation corresponding to the “many” side could result in null values
- The table corresponding to a

Representing Specialization as Tables

- Method 1:

- Form a table for the higher level entity
- Form a table for each lower level entity set, include primary key of higher level entity set and local attributes

table	table attributes
<i>person</i>	<i>name, street, city</i>
<i>customer</i>	<i>name, credit-rating</i>
<i>employee</i>	<i>name, salary</i>

- Drawback: getting information about, e.g., *employee* requires accessing two tables

Representing Specialization as Tables (Cont.)

- Method 2:
 - Form a table for each entity set with all local and inherited attributes

table	table attributes
<i>person</i>	<i>name, street, city</i>
<i>customer</i>	<i>name, street, city, credit-rating</i>
<i>employee</i>	<i>name, street, city, salary</i>

- If specialization is total, table for generalized entity (*person*) not required to store information
 - Can be defined as a “view” relation containing union of specialization tables
 - But explicit table may still be needed for foreign key constraints
- Drawback: street and city may be stored redundantly for persons who are both customers and employees

Relations Corresponding to Aggregation

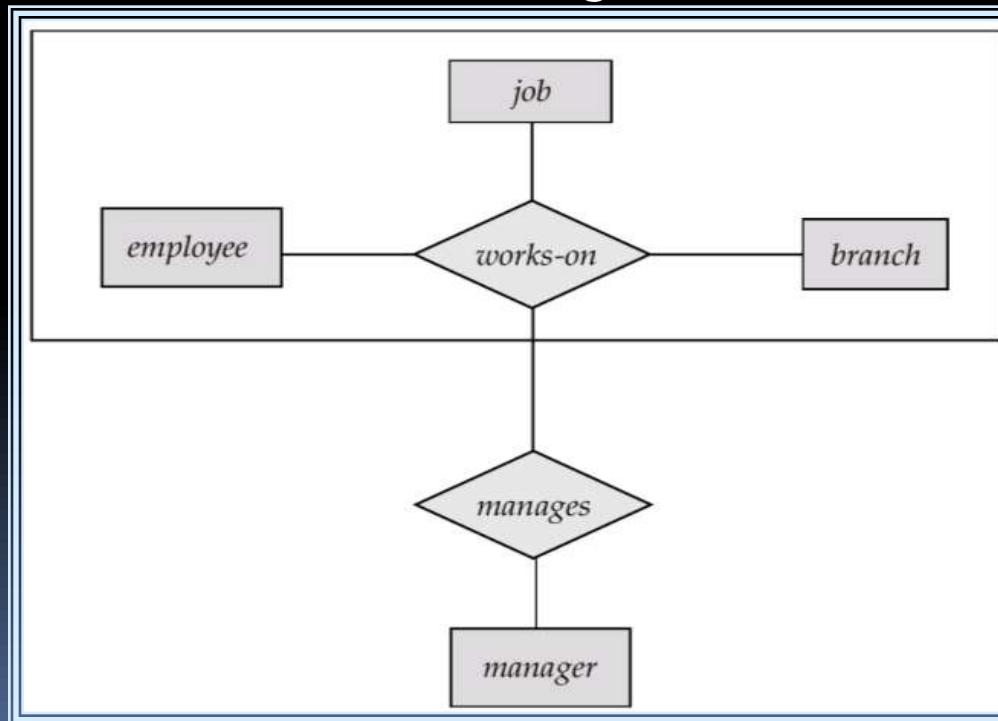
To represent aggregation, create a table containing

- primary key of the aggregated relationship,
- the primary key of the associated entity set
- Any descriptive attributes

Relations Corresponding to

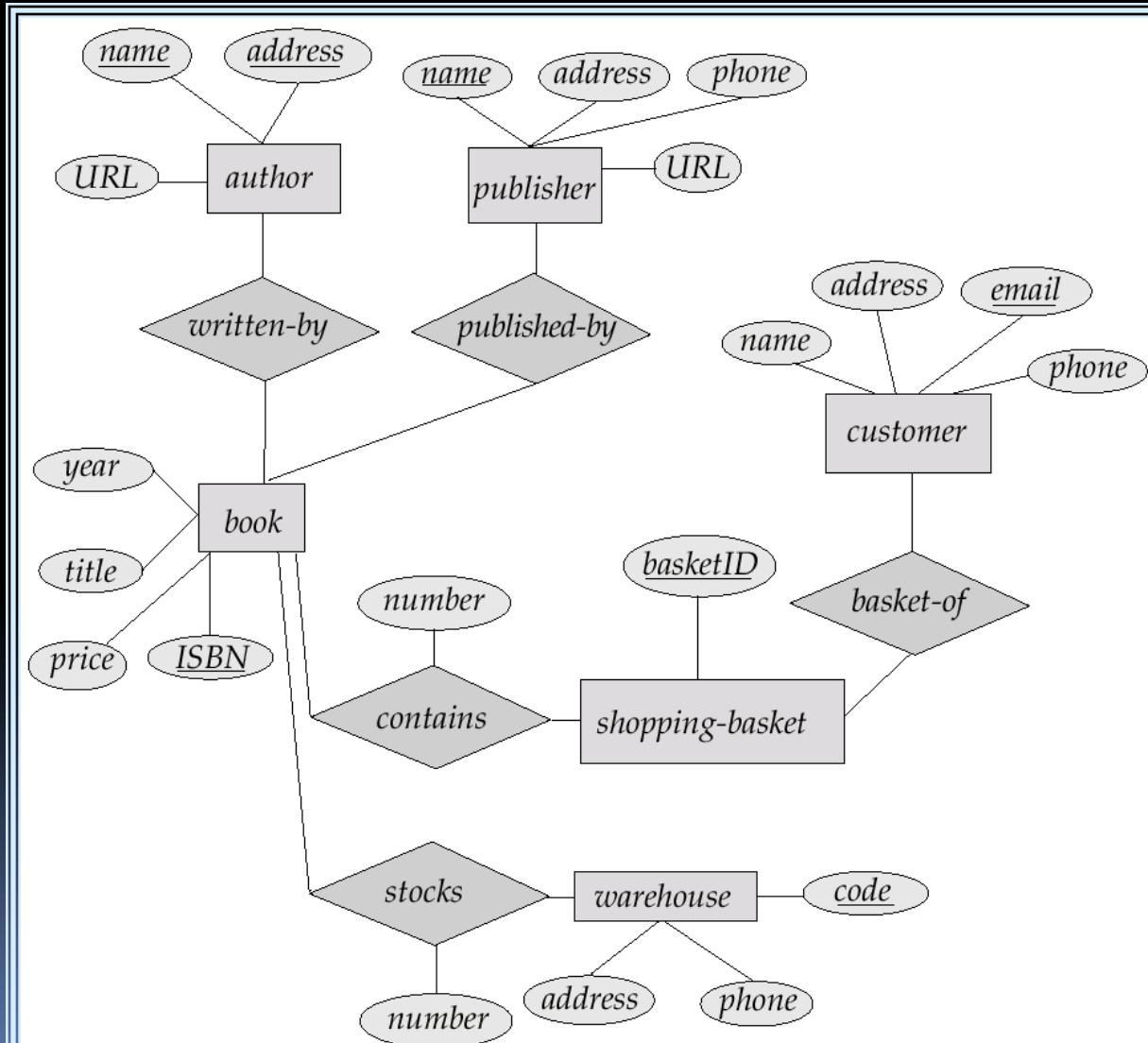
E.g. to represent aggregation *manages* between relationship *works-on* and entity set *manager*, create a table *manages(employee-id, branch-name, title, manager-name)*

Table *works-on* is redundant **provided** we are willing to store null values for attribute *manager-name* in table *manages*

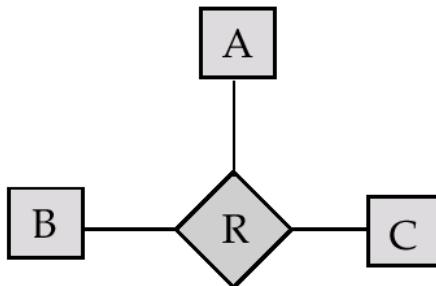


END OF CHAPTER 2

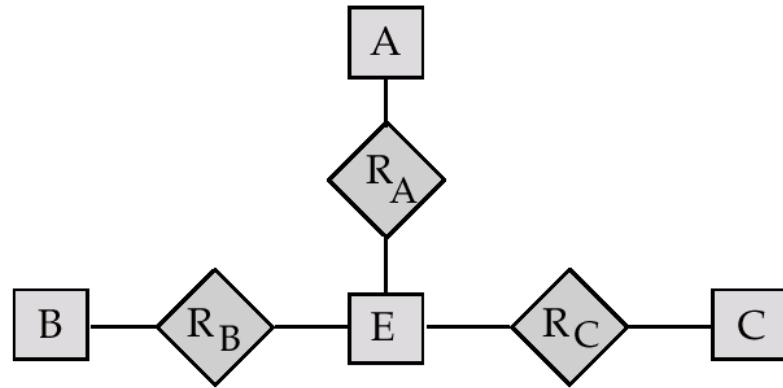
E-R Diagram for Exercise 2.10



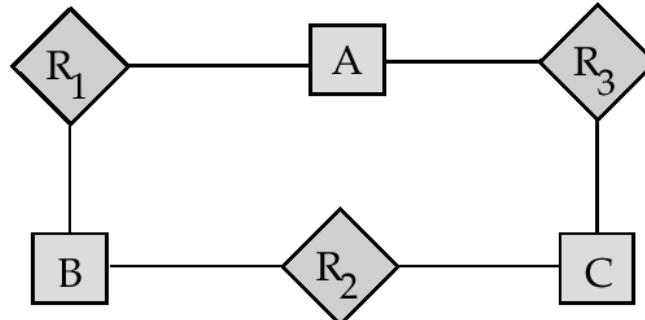
E-R Diagram for Exercise 2.15



(a)

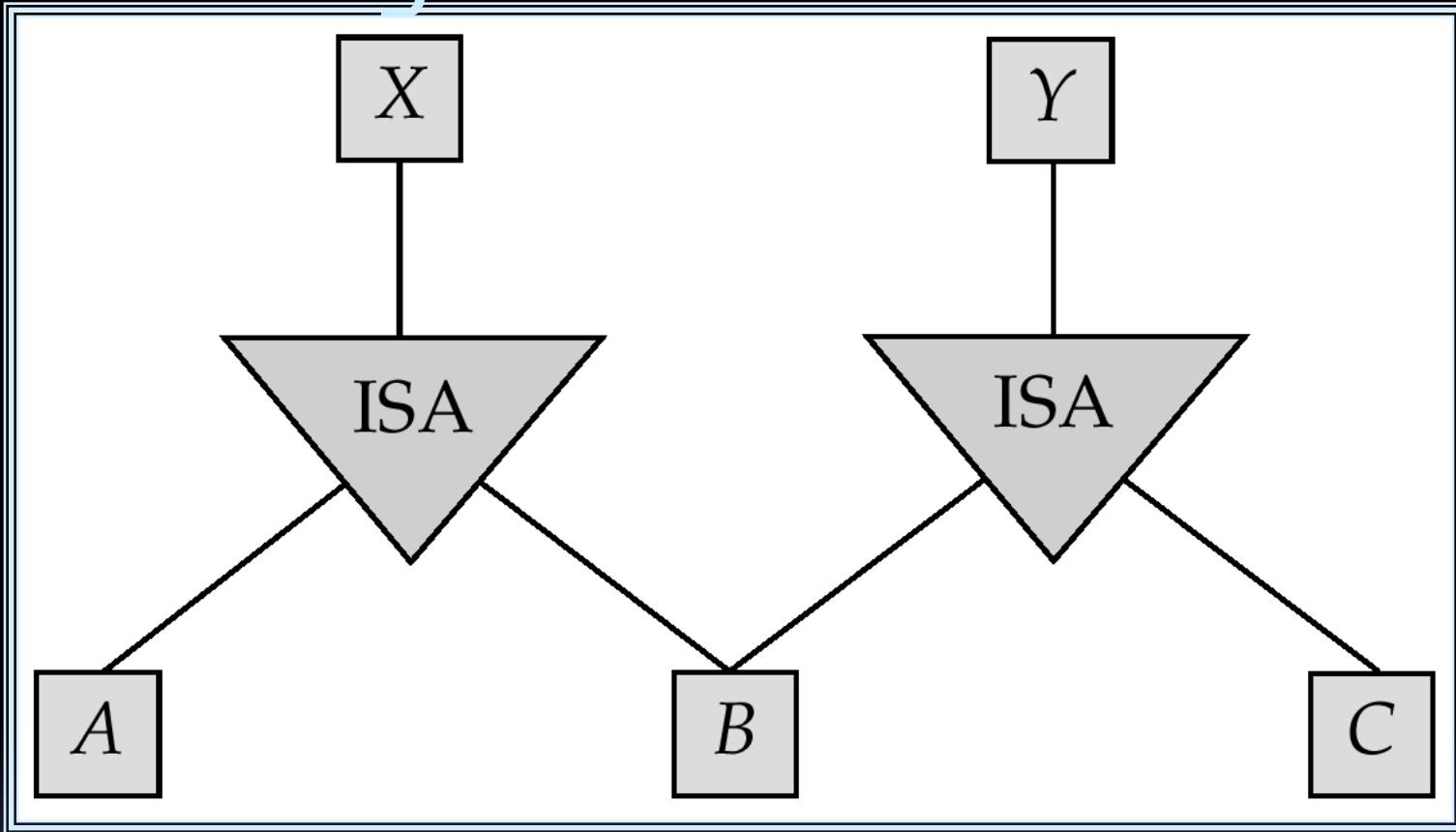


(b)

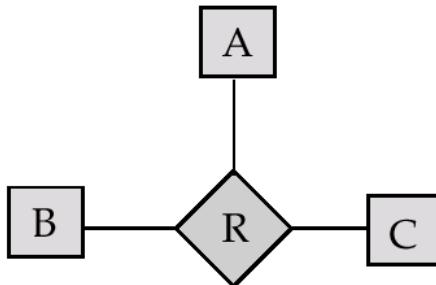


(c)

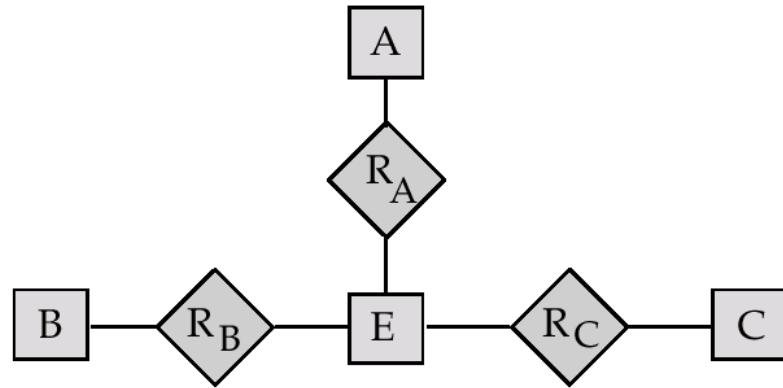
E-R Diagram for Exercise 2.22



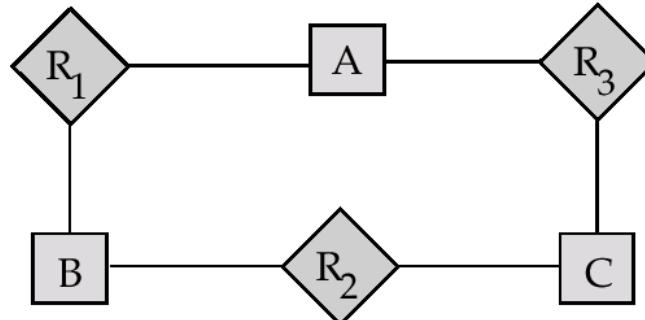
E-R Diagram for Exercise 2.15



(a)



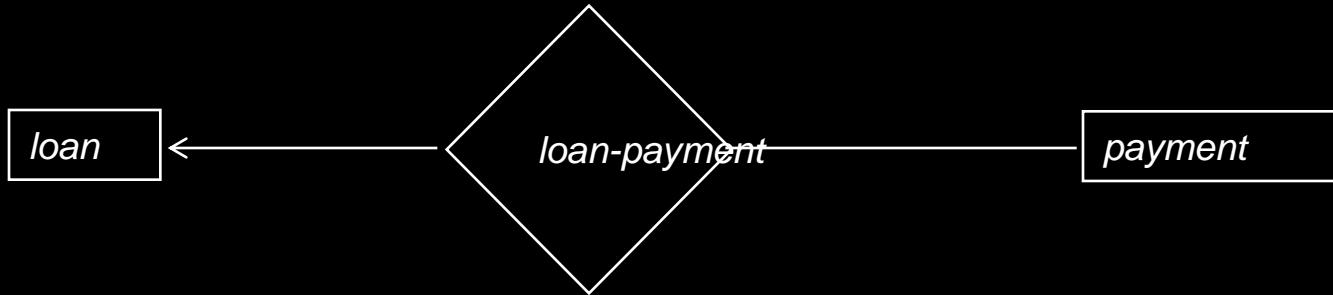
(b)



(c)

Existence Dependencies

- If the existence of entity x depends on the existence of entity y , then x is said to be *existence dependent* on y .
 - y is a *dominant entity* (in example below, *loan*)
 - x is a *subordinate entity* (in example below, *payment*)



If a *loan* entity is deleted, then all its associated *payment* entities must be deleted also.

Chapter 3: Relational Model

- Structure of Relational Databases
- Relational Algebra
- Tuple Relational Calculus
- Domain Relational Calculus
- Extended Relational-Algebra-Operations
- Modification of the Database
- Views

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 = {Jones, Smith, Curry, Lindsay}

customer-street = {Main, North, Park}

customer-city = {Harrison, Rye, Pittsfield}

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. multivalued attribute values are not atomic
 - E.g. composite attribute values are not 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

Relation Schema

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

E.g. *Customer-schema* =

*(customer-name,
customer-street, customer-city)*

- $r(R)$ is a *relation* on the *relation schema* R

E.g. *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 illustrates a relation instance table named 'customer'. The table has three columns labeled 'customer-name', 'customer-street', and 'customer-city'. The 'customer-name' column contains the values 'Jones', 'Smith', 'Curry', and 'Lindsay'. The 'customer-street' column contains 'Main', 'North', 'North', and 'Park'. The 'customer-city' column contains 'Harrison', 'Rye', 'Rye', and 'Pittsfield'. Arrows point from the text labels 'attributes (or columns)' to the column headers, and arrows point from the text labels 'tuples (or rows)' to the row data.

customer-name	customer-street	customer-city
Jones Smith Curry Lindsay	Main North North Park	Harrison Rye Rye Pittsfield

customer

Relations are Unordered

- Order of tuples is irrelevant (tuples may be stored in an arbitrary order)
- E.g. account relation with unordered tuples

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

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

E.g.: *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. two customers own an account)
 - the need for null values (e.g. represent a customer without an account)
- Normalization theory (Chapter 7) deals with how to design relational schemas

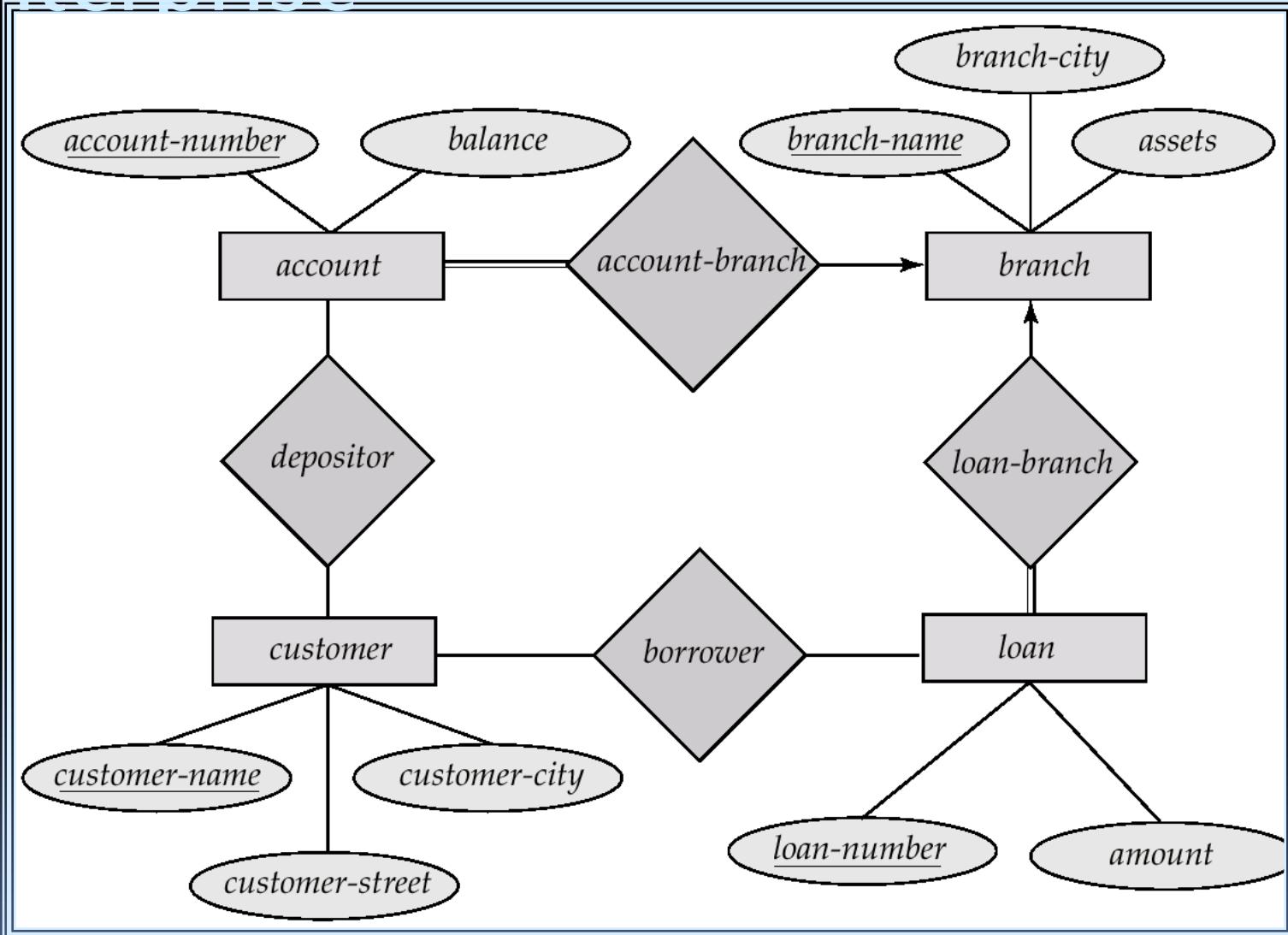
The *customer* Relation

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

The *depositor* Relation

<i>customer-name</i>	<i>account-number</i>
Hayes	A-102
Johnson	A-101
Johnson	A-201
Jones	A-217
Lindsay	A-222
Smith	A-215
Turner	A-305

E-R Diagram for the Banking Enterprise



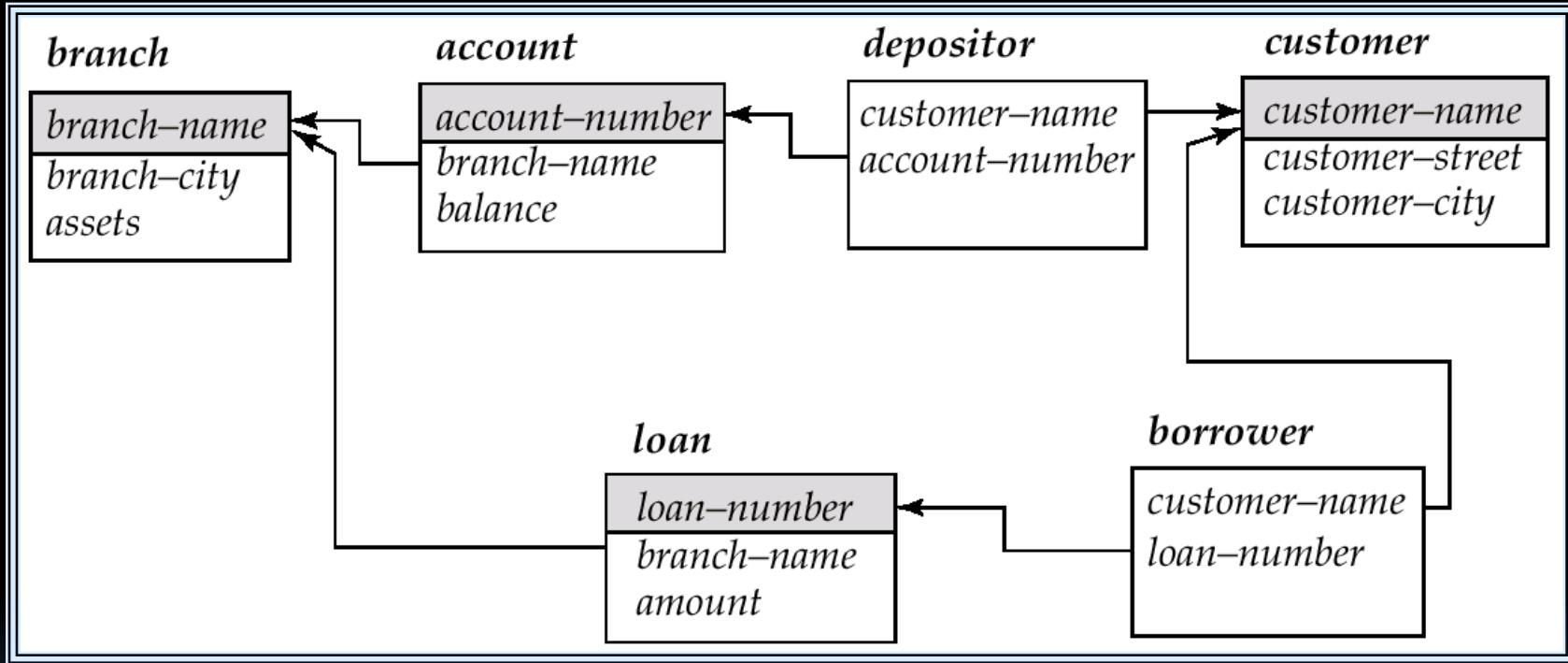
Keys

- Let $K \subseteq R$
- K is a *superkey* of R if values for K are sufficient to identify a unique tuple of each possible relation $r(R)$
 - by “possible r ” we mean a relation r that could exist in the enterprise we are modeling.
 - Example: $\{customer-name, customer-street\}$ and $\{customer-name\}$ are both superkeys of *Customer*, if no two customers can possibly have the same

Determining Keys from E-R Sets

- Strong entity set. The primary key of the entity set becomes the primary key of the relation.
- Weak entity set. The primary key of the relation consists of the union of the primary key of the strong entity set and the discriminator of the weak entity set.
- Relationship set. The union of the primary keys of the related entity sets becomes a super key of the

Schema Diagram for the Banking Enterprise



Query Languages

- Language in which user requests information from the database.
- Categories of languages
 - procedural
 - non-procedural
- “Pure” languages:
 - Relational Algebra
 - Tuple Relational Calculus
 - Domain Relational Calculus
- Pure languages form underlying basis for several query languages

Relational Algebra

- Procedural language
- Six basic operators
 - select
 - project
 - union
 - set difference
 - Cartesian product
 - rename
- The operators take one or more relations as inputs and give a new relation as a result

Select Operation - Example

- Relation r

A	B	C	D
□	□	1	7
□	□	5	7
□	□	12	3
□	□	23	10

- $\square_{A=B \wedge D > 5}(r)$

A	B	C	D
□	□	1	7
□	□	23	10

Select Operation

- Notation: $\sigma_p(r)$
- p is called the selection predicate
- Defined as:

$$\sigma_p(r) = \{t \mid t \in r \text{ and } p(t)\}$$

Where p is a formula in propositional calculus consisting of terms connected by : \wedge (and), \vee (or), \neg (not)

Each term is one of:

$\langle \text{attribute} \rangle \quad op$

$\langle \text{attribute} \rangle \text{ or } \langle \text{constant} \rangle$

where op is one of: $=, \neq, >, \geq,$

Project Operation - Example

- Relation r :

	A	B	C
□	10	1	
□	20	1	
□	30	1	
□	40	2	

- $\square_{A,C}(r)$

	A	C
□	1	
□	1	
□	1	
□	2	

	A	C
□	1	
□	1	
□	2	

=

	A	C
□	1	
□	1	
□	2	

Project Operation

- 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

Union Operation - Example

- Relations r and s :

A	B
□	1
□	2
□	1

r

A	B
□	2
□	3

s

$r \sqcup s$:

A	B
□	1
□	2
□	1
□	3

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* (e.g., 2nd column of r deals with the same type of

Set Difference Operation - Example

Relations /

A	B
□	1
□	2
□	1

r

A	B
□	2
□	3

s

$r - s:$

A	B
□	1
□	1

Set Difference Operation

- Notation $r - s$
- Defined as:

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

- 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

Cartesian-Product Operation-Example

Relations r, s:

A	B
□	1
□	2

r

C	D	E
□	10	a
□	10	a
□	20	b
□	10	b

s

$r \times s$:

A	B	C	D	E
□	1	□	10	a
□	1	□	10	a
□	1	□	20	b
□	1	□	10	b
□	2	□	10	a
□	2	□	10	a
□	2	□	20	b
□	2	□	10	b

Cartesian–Product Operation

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

$$r \times s = \{t q \mid t \in r \text{ and } q \in s\}$$

- Assume that attributes of $r(R)$ and $s(S)$ are disjoint. (That is,
 $R \cap S = \emptyset$).
- If attributes of $r(R)$ and $s(S)$ are not disjoint, then renaming must be used.

Composition of Operations

- Can build ~~each~~ using multiple operations
- Example: ~~each~~
- $r \times s$

A	B	C	D	E
□	1	□	10	a
□	1	□	10	a
□	1	□	20	b
□	1	□	10	b
□	2	□	10	a
□	2	□	10	a
□	2	□	20	b
□	2	□	10	b

A	B	C	D	E
□	1	□	10	a
□	2	□	20	a
□	2	□	20	b

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:

$$\rho_x(E)$$

returns the expression E under the name X

If a relational-algebra expression E has

Banking Example

branch (branch-name, branch-city, assets)

customer (customer-name, customer-street, customer-only)

account (account-number, branch-name, balance)

loan (loan-number, branch-name, amount)

depositor (customer-name, account-number)

Example Queries

- Find all loans of over \$1200

 - amount > 1200 (loan)

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

 - loan-number (□ amount > 1200 (loan))

Example Queries

- Find the names of all customers who have a loan, an account, or both, from the bank
□ customer-name (borrower) □ □ customer-name (depositor)
- Find the names of all customers who have a loan and an account at bank.
□ customer-name (borrower) □ □ customer-name (depositor)

Example Queries

- Find the names of all customers who have a loan at the Perryridge branch.
 - customer-name (□ branch-name = “Perryridge”
(□ borrower.loan-number = loan.loan-number(borrower x 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.
 - customer-name (□ branch-name = “Perryridge”
(□ borrower.loan-number = loan.loan-number(borrower x loan))) –
□ customer-name(depositor)

Example Queries

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

—Query 1

- customer-name(branch-name = “Perryridge” (
 - borrower.loan-number = loan.loan-number(borrower x loan)))

~

Query 2

- customer-name(loan.loan-number = borrower.loan-number(
 - branch-name = “Perryridge”(loan)) x borrower))

Example Queries

Find the largest account balance

- Rename *account* relation as *d*
- The query is:
 - $\text{balance}(\text{account}) - \text{account.balance}$
 - $(\text{account.balance} < d.\text{balance} \text{ (account} \times \rho_d \text{ (account)))}$

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:
 - $E_1 \cup E_2$
 - $E_1 - E_2$
 - $E_1 \times E_2$
 - $\sigma_P(E)$. P is a predicate on attributes in E .

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:
- $r \cap s = \{ t \mid t \in r \text{ and } t \in s \}$
- Assume:
 - r, s have the *same arity*
 - attributes of r and s are compatible
- Note: $r \cap s = r - (r - s)$

Set-Intersection Operation – Example

- Relation

A	B
□	1
□	2
□	1

r

A	B
□	2
□	3

s

A	B
□	2

- $r \cap s$

Natural-Join Operation

- Notation: $r \bowtie s$
- Let r and s be relations on schemas R and S respectively.
Then, $r \bowtie s$ is a relation on schema $R \cup S$ obtained as follows:
 - Consider each pair of tuples t_r from r and t_s from s .
 - If t_r and t_s have the same value on each of the attributes in $R \cap S$, add a tuple t to the result, where
 - \bowtie has the same value as t_r on r
 - t has the same value as t_s on s
- Example:

Natural Join Operation -

- Example s:

A	B	C	D
□	1	□	a
□	2	□	a
□	4	□	b
□	1	□	a
□	2	□	b

r

B	D	E
1	a	□
3	a	□
1	a	□
2	b	□
3	b	□

s

$r \bowtie s$

A	B	C	D	E
□	1	□	a	□
□	1	□	a	□
□	1	□	a	□
□	1	□	a	□
□	2	□	b	□

Division Operation

$r \square s$

- Suited to queries that include the phrase “for all”.
- Let r and s be relations on schemas R and S respectively where
 - $R = (A_1, \dots, A_m, B_1, \dots, B_n)$
 - $S = (B_1, \dots, B_n)$

The result of $r \div s$ is a relation on schema

$$R - S = (A_1, \dots, A_m)$$

$$r \div s = \{ t \mid t \in \prod_{A_i \in R} (t_i \wedge \forall u \in S (tu \in r)) \}$$

Division Operation - Example

Relations r, s:

A	B
	1
	2
	3
	1
	1
	1
	3
	4
	6
	1
	2

r

B
1
2

s

$r \sqcap s$:

A

Another Division Example

Relations r, s:

A	B	C	D	E
□	a	□	a	1
□	a	□	a	1
□	a	□	b	1
□	a	□	a	1
□	a	□	b	3
□	a	□	a	1
□	a	□	b	1
□	a	□	b	1

r

D	E
a	1
b	1

s

$r \square s$:

A	B	C
□	a	□
□	a	□

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 $S \subseteq R$

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

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

Example Queries

- Find all customers who have an account from at least the “Downtown” and the Uptown branches.

Query 1
 $\square \text{CN}(\square \text{BN}=\text{"Downtown"}(\text{depositor } \bowtie \text{account})) \bowtie$
 $\square \text{CN}(\square \text{BN}=\text{"Uptown"}(\text{depositor } \bowtie \text{account}))$
 \bowtie

where CN denotes customer-name and BN denotes branch-name.

Query 2

$\square \text{customer-name, branch-name } (\text{depositor } \bowtie \text{account})$
 $\quad \square \square \text{temp(branch-name) } \{(\text{"Downtown"}), (\text{"Uptown"})\}$

Example Queries

- Find all customers who have an account at all branches located in Brooklyn city.
 - $\exists \text{customer-name} \exists \text{branch-name} (\text{depositor} \bowtie \text{account})$
 - $\exists \text{branch-name} (\exists \text{branch-city} = \text{"Brooklyn"} (\text{branch}))$

Extended Relational-Algebra-Operations

- Generalized Projection
- Outer Join
- Aggregate Functions

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

Aggregate Functions and Operations

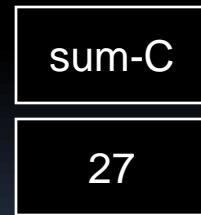
- Aggregation function takes a collection of values and returns a single value as a result.
 - avg: average value
 - min: minimum value
 - max: maximum value
 - sum: sum of values
 - count: number of values
- Aggregate operation in relational algebra

Aggregate Operation - Example

- Relation r :

A	B	C
□	□	7
□	□	7
□	□	3
□	□	10

$g_{\text{sum}(C)}(r)$



Aggregate Operation - Example

- Relation *account* grouped by *branch-name*

branch-name	account-number	balance
Perryridge	A-102	400
Perryridge	A-201	900
Brighton	A-217	750
Brighton	A-215	750
Redwood	A-222	700

branch-name g sum(balance) (account)

branch-name	balance
Perryridge	1300
Brighton	1500
Redwood	700

Aggregate Functions (Cont.)

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

Outer Join

- An extension of the join operation that avoids loss of information.
- Computes the join and then adds tuples from one relation that do not match tuples in the other relation to the result of the join.
- Uses *null* values:
 - *null* signifies that the value is unknown or does not exist
 - All comparisons involving *null* are (roughly speaking) false by definition.

Outer Join - Example

- Relation *loan*

loan-number	branch-name	amount
L-170	Downtown	3000
L-230	Redwood	4000
L-260	Perryridge	1700

- Relation borrower

customer-name	loan-number
Jones	L-170
Smith	L-230
Hayes	L-155

Outer Join - Example

■ Inner Join

loan \bowtie *Borrower*

loan-number	branch-name	amount	customer-name
L-170	Downtown	3000	Jones
L-230	Redwood	4000	Smith

■ Left Outer Join

loan $\leftarrow \bowtie$ *Borrower*

loan-number	branch-name	amount	customer-name
L-170	Downtown	3000	Jones
L-230	Redwood	4000	Smith
L-260	Perryridge	1700	null

Outer Join - Example

■ Right Outer Join

$loan \bowtie^R borrower$

loan-number	branch-name	amount	customer-name
L-170	Downtown	3000	Jones
L-230	Redwood	4000	Smith
L-155	null	null	Hayes

■ Full Outer Join

$loan \bowtie^F borrower$

loan-number	branch-name	amount	customer-name
L-170	Downtown	3000	Jones
L-230	Redwood	4000	Smith
L-260	Perryridge	1700	null
L-155	null	null	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
 - Is an arbitrary decision. Could have returned *null* as result instead

Null Values

- Comparisons with null values return the special truth value *unknown*
 - If *false* was used instead of *unknown*, then *not (A < 5)* would not be equivalent to $A \geq 5$
- Three-valued logic using the truth value *unknown*:
 - OR: $(\text{unknown or true}) = \text{true}$,
 $(\text{unknown or false}) = \text{unknown}$
 $(\text{unknown or unknown}) = \text{unknown}$
 - AND: $(\text{true and unknown}) = \text{unknown}$,
 $(\text{false and unknown}) = \text{false}$,
 $(\text{unknown and unknown}) = \text{unknown}$

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:

Deletion Examples

- Delete all account records in the Perryridge branch
$$\text{account} \sqcap \text{branch-name} = \text{"Perryridge"} \text{ (account)}$$

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

$$\text{loan} \sqcap \text{loan - amount} \sqcap 0 \text{and amount} \sqcap 50 \text{ (loan)}$$

- Delete all accounts at branches located in Needham.

$$r_1 \sqcap \text{branch-city} = \text{"Needham"} \text{ (account)}$$
 \times branch

$$r_2 \sqcap \text{branch-name, account-number, balance}$$
 (r_1)

$$r_3 \sqcap \text{customer-name, account-number}$$
 (r_2) depositor
account \sqcap account – r_2 \times
depositor \sqcap depositor – r_3

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

Insertion Examples

- Insert information in the database specifying that Smith has \$1200 in account A-973 at the Perryridge branch.
account account {("Perryridge", A-973, 1200)}
depositor depositor {("Smith", A-973)}
- Provide as a gift for all loan customers in the Perryridge branch, a \$200 savings account. Let the loan number serve as the account number for the new savings account.

$r_1 \sqsubseteq (\square_{\text{branch-name} = \text{"Perryridge"}} (\text{borrower} \bowtie \text{loan}))$
account account $\sqsubseteq \square_{\text{branch-name}, \text{account-number}, 200}(r_1)$
depositor depositor $\sqsubseteq \square_{\text{customer-name}, \text{loan-number}}(r_1)$

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

$$r \leftarrow \prod_{F_1, F_2, \dots, F_n} (r)$$

- Each F_i is either
 - the i th attribute of r , if the i th attribute is not updated, or,
 - if the attribute is to be updated F_i is an expression involving only constants and

Update Examples

- Make interest payments by increasing all balances by 5 percent
$$\text{account } \square \quad \square \text{AN, BN, BAL} * 1.05 \text{ (account)}$$

where AN, BN and BAL stand for account-number, branch-name and balance, respectively.

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

account \square $\square \text{AN, BN, BAL} * 1.06 (\square \text{BAL} \square 10000 \text{ (account)})$
 $\square \quad \square \text{AN, BN, BAL} * 1.05 (\square \text{BAL} \square 10000 \text{ (account)})$

Views

- In some cases, it is not desirable for all users to see the entire logical model (i.e., all the actual relations stored in the database.)
- Consider a person who ^{needs} to know a customer's loan number but has no need to see the loan amount. This person should see a relation described, in the relational algebra, by
$$\Pi_{customer-name, loan-number} (borrower \quad loan)$$
- Any relation that is not of the conceptual model but is made visible

View Definition

- A view is defined using the create view statement which has the form
create view v as <query expression>

where <query expression> is any legal relational algebra query expression. The view name is represented by v .

- Once a view is defined, the view name can be used to refer to the virtual relation that the view generates.

View Examples

- Consider the view (named *all-customer*) consisting of branches and their customers as

 - branch-name, customer-name (depositor \bowtie account)
 - □ branch-name, customer-name (borrower \bowtie loan)

- We can find all customers of the Perryridge branch by writing:

 - customer-name
 - (□ branch-name = “Perryridge” (all-customer))

Updates Through View

- Database modifications expressed as views must be translated to modifications of the actual relations in the database.
- Consider the person who needs to see all loan data in the *loan* relation except *amount*. The view given to the person, *branch-loan*, is defined as:
create view *branch-loan* as

$$\Pi_{\text{branch-name}, \text{loan-number}} (\text{loan})$$

Updates Through Views (Cont.)

- The previous insertion must be represented by an insertion into the actual relation *loan* from which the view *branch-loan* is constructed.
- An insertion into *loan* requires a value for *amount*. The insertion can be dealt with by either.
 - rejecting the insertion and returning an error message to the user.
 - inserting a tuple (“L-37”, “Perryridge”, *null*) into the *loan* relation
- Some updates through views are

Views Defined Using Other Views

- One view may be used in the expression defining another view
- A view relation v_1 is said to *depend directly* on a view relation v_2 if v_2 is used in the expression defining v_1
- A view relation v_1 is said to *depend on* view relation v_2 if either v_1 depends directly to v_2 or there is a path of dependencies from v_1 to v_2
- A view relation v is said to be *recursive* if it depends on itself.

View Expansion

- A way to define the meaning of views defined in terms of other views.
- Let view v_i be defined by an expression e_i that may itself contain uses of view relations.
- View expansion of an expression repeats the following replacement step:

repeat

Find any view relation v_i in e_i

Replace the view relation v_i by the

Tuple Relational Calculus

- A nonprocedural query language, where each query is of the form

$$\{t \mid P(t)\}$$

- It is the set of all tuples t such that predicate P is true for t
- t is a *tuple variable*, $t[A]$ denotes the value of tuple t on attribute A
- $t \in r$ denotes that tuple t is in relation r
- P is a *formula* similar to that of the predicate calculus

Predicate Calculus Formula

1. Set of attributes and constants
2. Set of comparison operators: (e.g.,
 $<$, \leq , $=$, \neq , $>$, \geq)
3. Set of connectives: and (\wedge), or (\vee),
not (\neg)
4. Implication (\Rightarrow): $x \Rightarrow y$, if x if true,
then y is true

$$x \Rightarrow y \equiv \neg x \vee y$$

5. Set of quantifiers:

- $\exists t \in r (Q(t)) \equiv$ "there exists" a tuple in t in

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)*

Example Queries

- Find the *loan-number*, *branch-name*, and *amount* for loans of over \$1200 { $t \mid t \sqsubseteq \text{loan} \sqcap t[\text{amount}] \sqgt 1200$ }
- Find the loan number for each loan of an amount greater than \$1200 { $t \mid s \sqsubseteq \text{loan} \wedge t[\text{loan-number}] = s[\text{loan-number}] \wedge s[\text{amount}] \sqgt 1200$ }

Notice that a relation on schema [loan-number] is implicitly defined by the query

Example Queries

- Find the names of all customers having a loan, an account, or both at the bank
 - {t | s \sqcup borrower(t[customer-name] = s[customer-name]) \sqcup . u \sqcup depositor(t[customer-name] = u[customer-name])
- Find the names of all customers who have a loan and an account at the bank
 - {t | s \sqcup borrower(t[customer-name] = s[customer-name]) \sqcup . u \sqcup depositor(t[customer-name] = u[customer-name])

Example Queries

- Find the names of all customers having a loan at the Perryridge branch

```
{t |. s ☐ borrower(t[customer-name] = s[customer-name]
    ☐. u ☐ loan(u[branch-name] = "Perryridge"
        ☐ u[loan-number] = s[loan-number]))}
```

- Find the names of all customers who have a loan at the Perryridge branch, but no account at any branch of the bank

```
{t |. s ☐ borrower( t[customer-name] = s[customer-name]
    ☐. u ☐ loan(u[branch-name] = "Perryridge"
        ☐ u[loan-number] = s[loan-number]))
    ☐ not v ☐ depositor (v[customer-name] =
        t[customer-name]) }
```

Example Queries

- Find the names of all customers having a loan from the Perryridge branch, and the cities they live in
 - u □ borrower (u[loan-number] = s[loan-number])
 - t [customer-name] = u[customer-name]
 - v □ customer (u[customer-name] = v[customer-name])
 - t[customer-city] = v[customer-city]))})

Example Queries

- Find the names of all customers who have an account at all branches located in Brooklyn.
 - . $\exists c \text{ customer} . \exists s \text{ student} . \exists u \text{ user} . \exists t \text{ teacher} . \exists b \text{ branch} . \exists a \text{ account} . \exists d \text{ depositor} . c[\text{customer-name}] = c[\text{customer-name}] \wedge s[\text{branch-city}] = \text{"Brooklyn"} \wedge u[\text{branch-name}] = b[\text{branch-name}] \wedge a[\text{branch-name}] = b[\text{branch-name}] \wedge d[\text{customer-name}] = s[\text{customer-name}] \wedge s[\text{account-number}] = u[\text{account-number}] \wedge a[\text{account-number}] = u[\text{account-number}] \wedge d[\text{account-number}] = a[\text{account-number}])) \}$

Safety of Expressions

- It is possible to write tuple calculus expressions that generate infinite relations.
- For example, $\{t \mid \neg t \in r\}$ results in an infinite relation if the domain of any attribute of relation r is infinite
- To guard against the problem, we restrict the set of allowable expressions to safe expressions.
- An expression $\{t \mid P(t)\}$ in the tuple relational calculus is *safe* if every

Domain Relational Calculus

- A nonprocedural query language equivalent in power to the tuple relational calculus
- Each query is an expression of the form:

$$\{ < X_1, X_2, \dots, X_n > \mid P(X_1, X_2, \dots, X_n) \}$$

- X_1, X_2, \dots, X_n represent domain variables

Example Queries

- Find the *loan-number*, *branch-name*, and *amount* for loans of over \$1200
 $\{ \text{[l, b, a]} \mid \text{[l, b, a]} \text{ loan } a > 1200 \}$
- Find the names of all customers who have a loan of over \$1200
 $\{ \text{[c, l, b, a}} \mid \text{[c, l} \text{ borrower } \text{[l, b, a]} \text{ loan } a > 1200 \}$
- Find the names of all customers who have a loan from the Perryridge branch and the loan amount:
 $\{ \text{[c, a}} \mid \text{[l} \mid (\text{[c, l} \text{ borrower } \text{[b(l, b, a)} \text{ loan } b = \text{"Perryridge"})\}$
or $\{ \text{[c, a}} \mid \text{[l} \mid (\text{[c, l} \text{ borrower } \text{[l, "Perryridge", a}} \text{ loan})\}$

Example Queries

- Find the names of all customers having a loan, an account, or both at the Perryridge branch:

{ $\exists c \exists l \exists a \exists n$. c, l, a, n \sqsubseteq customer }
 . $b, a(l, b, a) \sqsubseteq$ loan $\sqcap b =$ "Perryridge") }
 . $a(c, a) \sqsubseteq$ depositor
 . $b, n(a, b, n) \sqsubseteq$ account $\sqcap b =$ "Perryridge") } }

- Find the names of all customers who have an account at all branches located in Brooklyn:

{ $\exists c \exists s \exists n$. c, s, n \sqsubseteq customer }
 . $x, y, z(x, y, z) \sqsubseteq$ branch $\sqcap y =$ "Brooklyn" }
 . $a, b(x, y, z) \sqsubseteq$ account $\sqcap c, a \sqsubseteq$ depositor } }

Safety of Expressions

$$\{ \langle x_1, x_2, \dots, x_n \rangle \mid P(x_1, x_2, \dots, x_n) \}$$

is safe if all of the following hold:

1. All values that appear in tuples of the expression are values from $\text{dom}(P)$ (that is, the values appear either in P or in a tuple of a relation mentioned in P).
2. For every “there exists” subformula of the form $\exists x (P_1(x))$, the subformula is true if and only if there

END OF CHAPTER 3



Result of σ *branch-name* = “Perryridge” (*loan*)

<i>loan-number</i>	<i>branch-name</i>	<i>amount</i>
L-15	Perryridge	1500
L-16	Perryridge	1300

Loan Number and the Amount of the Loan

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

Names of All Customers Who Have Either a Loan or an Account

customer-name

Adams
Curry
Hayes
Jackson
Jones
Smith
Williams
Lindsay
Johnson
Turner

Customers With An Account But No Loan

customer-name

Johnson
Lindsay
Turner

Result of *borrower* × *loan*

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

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

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

Result of $\Pi_{customer-name}$

customer-name

Adams
Hayes

Result of the Subexpression

<i>balance</i>
500
400
700
750
350

Largest Account Balance in the Bank

balance

900

Customers Who Live on the Same Street and In the Same City as Smith

customer-name

Curry
Smith

Customers With Both an Account and a Loan at the Bank

customer-name

Hayes
Jones
Smith

Result of $\Pi_{customer-name, loan-number, amount} (borrower \bowtie loan)$

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

Result of $\Pi_{branch-name}(\sigma_{customer-city = "Harrison"}(customer \bowtie account \bowtie depositor))$

<i>branch-name</i>
Brighton
Perryridge

Result of $\Pi_{branch-name}(\sigma_{branch-city} =$
“Brooklyn”(branch))

<i>branch-name</i>
Brighton
Downtown



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

<i>customer-name</i>	<i>branch-name</i>
Hayes	Perryridge
Johnson	Downtown
Johnson	Brighton
Jones	Brighton
Lindsay	Redwood
Smith	Mianus
Turner	Round Hill

The *credit-info* Relation

<i>customer-name</i>	<i>branch-name</i>
Hayes	Perryridge
Johnson	Downtown
Johnson	Brighton
Jones	Brighton
Lindsay	Redwood
Smith	Mianus
Turner	Round Hill

Result of $\Pi_{customer-name, (limit - credit-balance) as credit-available} (credit-info)$.

<i>customer-name</i>	<i>credit-available</i>
Curry	250
Jones	5300
Smith	1600
Hayes	0

The *pt-works* Relation

<i>employee-name</i>	<i>branch-name</i>	<i>salary</i>
Adams	Perryridge	1500
Brown	Perryridge	1300
Gopal	Perryridge	5300
Johnson	Downtown	1500
Loreena	Downtown	1300
Peterson	Downtown	2500
Rao	Austin	1500
Sato	Austin	1600

The *pt-works* Relation After Grouping

<i>employee-name</i>	<i>branch-name</i>	<i>salary</i>
Rao	Austin	1500
Sato	Austin	1600
Johnson	Downtown	1500
Loreena	Downtown	1300
Peterson	Downtown	2500
Adams	Perryridge	1500
Brown	Perryridge	1300
Gopal	Perryridge	5300

Result of $\text{branch-name} \Sigma \text{sum(salary)} \text{ (pt-works)}$

<i>branch-name</i>	<i>sum of salary</i>
Austin	3100
Downtown	5300
Perryridge	8100

Result of $\text{branch-name} \Sigma \text{sum salary}$,
 $\max(\text{salary})$ as max-salary (*pt-works*)

<i>branch-name</i>	<i>sum-salary</i>	<i>max-salary</i>
Austin	3100	1600
Downtown	5300	2500
Perryridge	8100	5300

The *employee* and *ft-works* Relations

<i>employee-name</i>	<i>street</i>	<i>city</i>
Coyote	Toon	Hollywood
Rabbit	Tunnel	Carrotville
Smith	Revolver	Death Valley
Williams	Seaview	Seattle

<i>employee-name</i>	<i>branch-name</i>	<i>salary</i>
Coyote	Mesa	1500
Rabbit	Mesa	1300
Gates	Redmond	5300
Williams	Redmond	1500

The Result of *employee*~~e~~ *ft-works*

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

The Result of *employee*~~X~~ *ft-works*

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

Result of *employee* \bowtie *ft-works*

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

Result of *employee*^{XN} *ft-works*

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

Tuples Inserted Into *loan* and *borrower*

<i>loan-number</i>	<i>branch-name</i>	<i>amount</i>
L-11	Round Hill	900
L-14	Downtown	1500
L-15	Perryridge	1500
L-16	Perryridge	1300
L-17	Downtown	1000
L-23	Redwood	2000
L-93	Mianus	500
<i>null</i>	<i>null</i>	1900

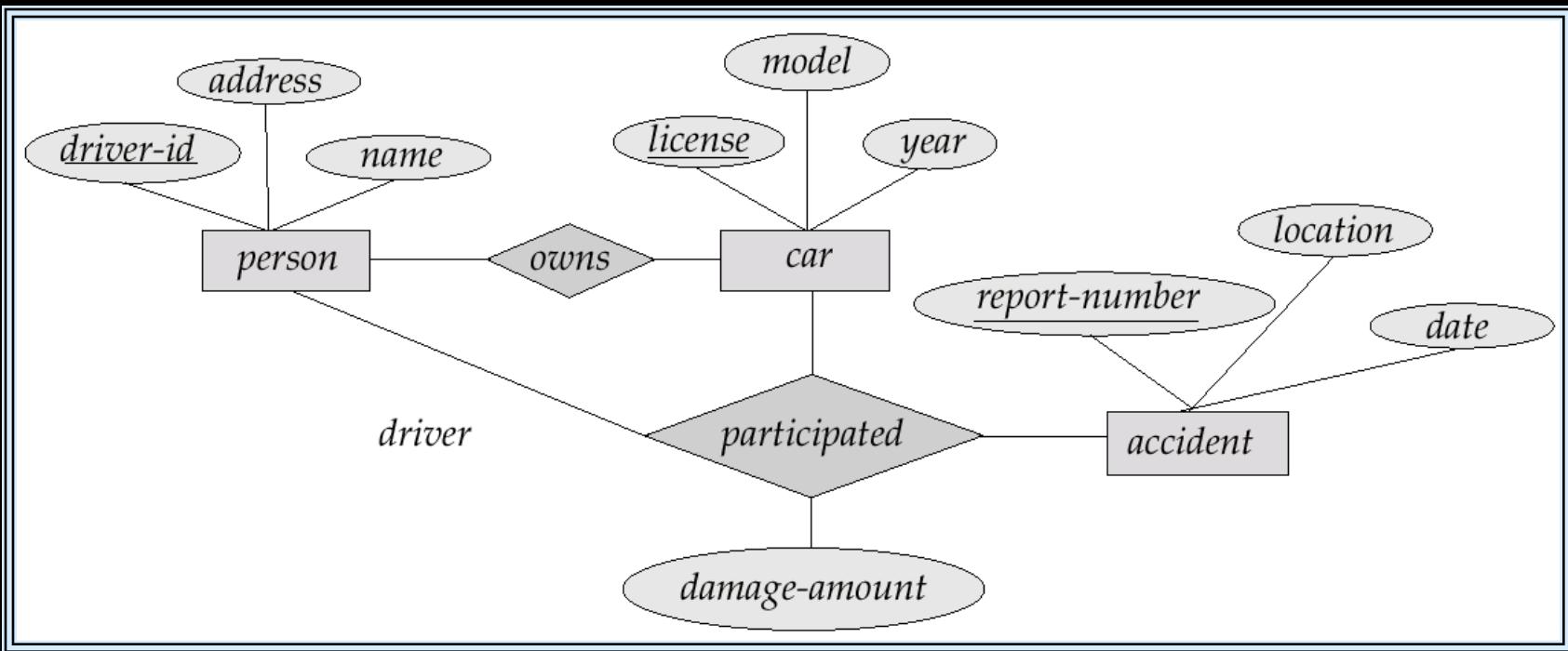
<i>customer-name</i>	<i>loan-number</i>
Adams	L-16
Curry	L-93
Hayes	L-15
Jackson	L-14
Jones	L-17
Smith	L-11
Smith	L-23
Williams	L-17
Johnson	<i>null</i>

Names of All Customers Who Have a Loan at the Perryridge Branch

customer-name

Adams
Hayes

E-R Diagram



The *branch* Relation

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

The *loan* Relation

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

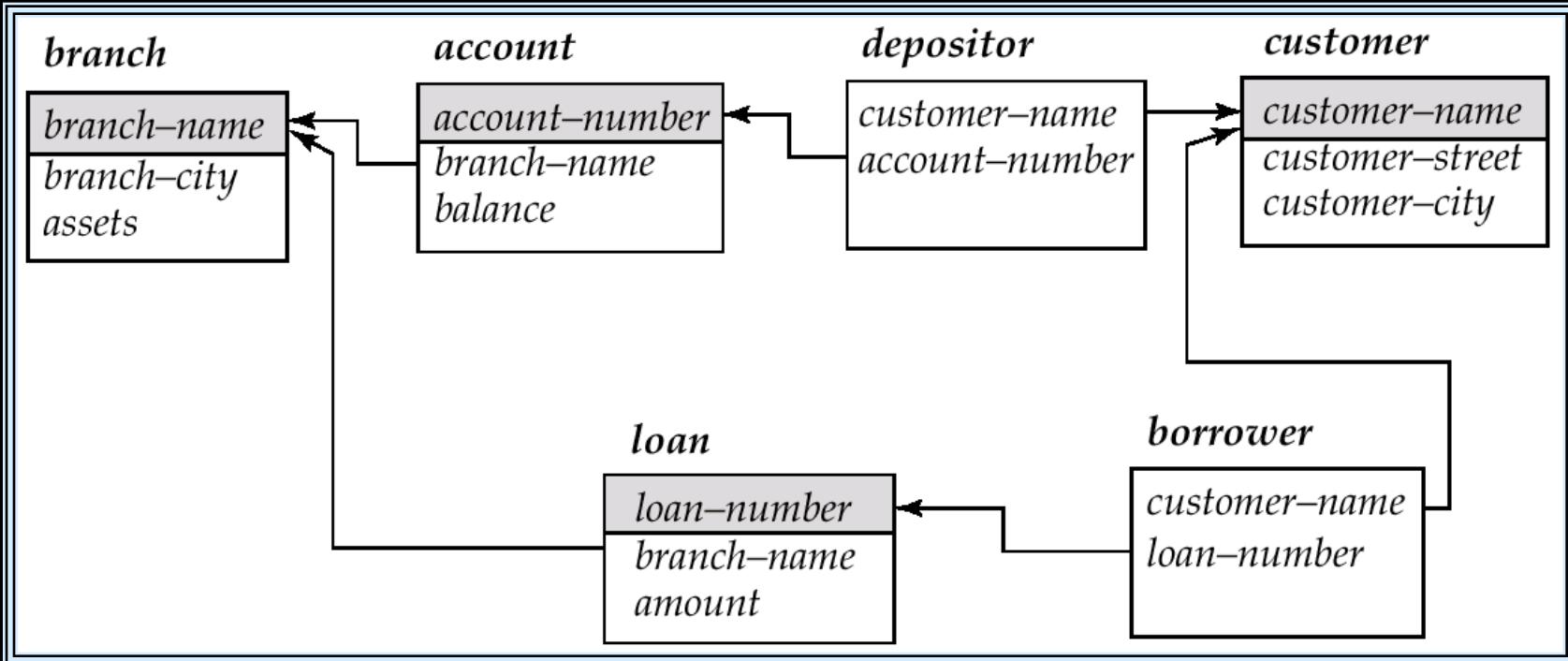
The *borrower* Relation

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

Chapter 4: SQL

- Basic Structure
- Set Operations
- Aggregate Functions
- Null Values
- Nested Subqueries
- Derived Relations
- Views
- Modification of the Database
- Joined Relations
- Data Definition Language
- Embedded SQL, ODBC and JDBC

Schema Used in Examples



Basic Structure

- SQL is based on set and relational operations with certain modifications and enhancements
- A typical SQL query has the form:
$$\begin{array}{l} \text{select } A_1, A_2, \dots, A_n \\ \text{from } r_1, r_2, \dots, r_m \\ \text{where } P \end{array}$$
 - A_i s represent attributes
 - r_j s represent relations
 - P is a predicate.
- This query is equivalent to the relational algebra expression.

$$\prod_{A1, A2, \dots, An} (\sigma_P (r_1 \times r_2 \times \dots \times r_m))$$

- The result of an SQL query is a relation.

The select Clause

- The select clause list the attributes desired in the result of a query
 - corresponds to the projection operation of the relational algebra
- E.g. find the names of all branches in the *loan* relation

```
select branch-name
      from loan
```
- In the “pure” relational algebra syntax, the query would be:

$$\Pi_{\text{branch-name}}(\textit{loan})$$

- NOTE: SQL does not permit the ‘-’ character in names,
 - Use, e.g., *branch_name* instead of *branch-name* in a real implementation.
 - We use ‘-’ since it looks nicer!
- NOTE: SQL names are case insensitive, i.e. you can use capital or small letters.
 - You may wish to use upper case where-ever we use bold font.

The select Clause (Cont.)

- SQL allows duplicates in relations as well as in query results.
- To force the elimination of duplicates, insert the keyword `distinct` after `select`.
- Find the names of all branches in the *loan* relations, and remove duplicates

`select distinct branch-name
from loan`

- The keyword `all` specifies that duplicates not be removed.

`select all branch-name
from loan`

The select Clause (Cont.)

- An asterisk in the select clause denotes “all attributes”
$$\text{select } * \\ \text{from } \textit{loan}$$
- The select clause can contain arithmetic expressions involving the operation, +, -, *, and /, and operating on constants or attributes of tuples.
- The query:

$$\text{select } \textit{loan-number}, \textit{branch-name}, \\ \textit{amount} * 100 \\ \text{from } \textit{loan}$$

would return a relation which is the same as the *loan* relations, except that the attribute *amount* is multiplied by 100.

The where Clause

- The where clause specifies conditions that the result must satisfy
 - corresponds to the selection predicate of the relational algebra.
- To find all loan number for loans made at the Perryridge branch with loan amounts greater than \$1200.

```
select loan-number
      from loan
            where branch-name = 'Perryridge' and
                  amount > 1200
```

- Comparison results can be combined using the logical connectives and, or, and not.
- Comparisons can be applied to results of arithmetic expressions.

The where Clause (Cont.)

- SQL includes a between comparison operator
- E.g. Find the loan number of those loans with loan amounts between \$90,000 and \$100,000 (that is, $\geq \$90,000$ and $\leq \$100,000$)

```
select loan-number
      from loan
     where amount between 90000 and 100000
```

The from Clause

- The from clause lists the relations involved in the query
 - corresponds to the Cartesian product operation of the relational algebra.
- Find the Cartesian product *borrower x loan*
- Find the name, loan number and loan amount of all customers having a loan at the Perryridge branch.

```
select *
from borrower, loan
select customer-name, borrower.loan-number, amount
from borrower, loan
where borrower.loan-number = loan.loan-number and
branch-name = 'Perryridge'
```

The Rename Operation

- The SQL allows renaming relations and attributes using the as clause:

old-name as new-name

- Find the name, loan number and loan amount of all customers; rename the column name *loan-number* as *loan-id*.
select customer-name, borrower.loan-number as loan-id, amount
from borrower, loan
where borrower.loan-number = loan.loan-number

Tuple Variables

- Tuple variables are defined in the from clause via the use of the as clause.
- Find the customer names and their loan numbers for all customers having a loan at some branch
select customer-name, T.loan-number, S.amount
from borrower as T, loan as S
where T.loan-number = S.loan-number
- Find the names of all branches that have greater assets than some branch located in Brooklyn.

```
select distinct T.branch-name  
from branch as T, branch as S  
where T.assets > S.assets and S.branch-city = 'Brooklyn'
```

String Operations

- SQL includes a string-matching operator for comparisons on character strings. Patterns are described using two special characters:
 - percent (%). The % character matches any substring.
 - underscore (_). The _ character matches any character.
- Find the names of all customers whose street includes the substring “Main”.
 - select *customer-name*
from *customer*
where *customer-street* like '%Main%'

- Match the name “Main%”
 - like 'Main\%' escape '\'
- SQL supports a variety of string operations such as
 - concatenation (using “||”)

Ordering the Display of Tuples

- List in alphabetic order the names of all customers having a loan in Perryridge branch

```
select distinct customer-name
  from borrower, loan
 where borrower loan-number =
 loan.loan-number and
       branch-name = 'Perryridge'
   order by customer-name
```

- We may specify desc for descending order or asc for ascending order, for each attribute; ascending order is the default

Duplicates

- In relations with duplicates, SQL can define how many copies of tuples appear in the result.
- *Multiset* versions of some of the relational algebra operators – given multiset relations r_1 and r_2 :
 1. $\sigma_\theta(r_1)$: If there are c_1 copies of tuple t_1 in r_1 , and t_1 satisfies selections σ_θ , then there are c_1 copies of t_1 in $\sigma_\theta(r_1)$.
 2. $\Pi_A(r_1)$: For each copy of tuple t_1 in r_1 , there is a copy of tuple $\Pi_A(t_1)$ in $\Pi_A(r_1)$.

Duplicates (Cont.)

- Example: Suppose multiset relations $r_1 (A, B)$ and $r_2 (C)$ are as follows:

$$r_1 = \{(1, a) (2, a)\} \quad r_2 = \{(2), (3), (3)\}$$

- Then $\Pi_B(r_1)$ would be $\{(a), (a)\}$, while $\Pi_B(r_1) \times r_2$ would be

$$\{(a, 2), (a, 2), (a, 3), (a, 3), (a, 3), (a, 3)\}$$

- SQL duplicate semantics:

```
select A1, A2, ..., An  
from r1 r2 ... r
```

Set Operations

- The set operations union, intersect, and except operate on relations and correspond to the relational algebra operations \cup , \cap , $-$.
- Each of the above operations automatically eliminates duplicates; to retain all duplicates use the corresponding multiset versions union all, intersect all and except all.

Suppose a tuple occurs m times in r

Set Operations

- Find all customers who have a loan, an account, or both.
(select customer-name from depositor)
union
(select customer-name from borrower)
- Find all customers who have both a loan and an account.
(select customer-name from depositor)
intersect
(select customer-name from borrower)
- Find all customers who have an account but no loan.
(select customer-name from depositor)
except
(select customer-name from borrower)

Aggregate Functions

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

avg: average value

min: minimum value

max: maximum value

sum: sum of values

count: number of values

Aggregate Functions (Cont.)

- Find the average account balance at the Perryridge branch.

```
from account  
where branch-name = 'Perryridge'
```

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

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

- Find the number of depositors in the bank.

```
select count (distinct customer-name)  
from depositor
```

Aggregate Functions - Group By

- Find the number of depositors for each branch

select branch-name, count (distinct customer-name)

from depositor, account

where depositor.account-number = account.account-number

group by branch-name

Note: Attributes in **select** clause outside of aggregate functions must appear in **group by** list

Aggregate Functions – Having Clause

- Find the names of all branches where the average account balance is more than

```
select branch-name, avg (balance)  
      from account  
      group by branch-name  
      having avg (balance) > 1200
```

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

Null Values

- It is possible for tuples to have a null value, denoted by *null*, for some of their attributes
- *null* signifies an unknown value or that a value does not exist.
- The predicate `is null` can be used to check for null values.
 - E.g. Find all loan number which appear in the *loan* relation with null values for *amount*.

```
select loan-number  
from loan
```

Null Values and Three Valued Logic

- Any comparison with *null* returns *unknown*
 - E.g. $5 < \text{null}$ or $\text{null} <> \text{null}$ or $\text{null} = \text{null}$
- Three-valued logic using the truth value *unknown*:
 - OR: (*unknown or true*) = *true*, (*unknown or false*) = *unknown*
(*unknown or unknown*) = *unknown*
 - AND: (*true and unknown*) = *unknown*,
(*false and unknown*) = *false*,

Null Values and Aggregates

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

Nested Subqueries

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

Example Query

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

```
select distinct customer-name
  from borrower
 where customer-name in (select customer-name
                           from depositor)
```

- Find all customers who have a loan at the bank but do not have an account at the bank

```
select distinct customer-name
  from borrower
 where customer-name not in (select customer-name
                           from depositor)
```

Example Query

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

~~select distinct customer-name
from borrower, loan~~

where *borrower.loan-number = loan.loan-number and
branch-name = “Perryridge” and
(branch-name, customer-name) in*

(select *branch-name, customer-name
from depositor, account
where depositor.account-number =
account.account-number)*

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

(Schema used in this example)

Set Comparison

- Find all branches that have greater assets than **some** branch located in Brooklyn.

```
select distinct T.branch-name  
from branch as T, branch as S  
where T.assets > S.assets and  
S.branch-city = 'Brooklyn'
```

- Same query using **> some** clause

```
select branch-name  
from branch  
where assets > some  
(select assets  
from branch  
where branch-city = 'Brooklyn')
```

Definition of Some Clause

- $F <\text{comp}> \text{some } r \Leftrightarrow \exists t \in r \text{ s.t.}$

$(F <\text{comp}> t)$

Where $\{\text{comp}\}$ can be:

$<, \leq, >$

0
5
6

\neq (read: 5 < some tuple in the relation)

$(5 < \text{some }) = \text{false}$

0
5

$(5 = \text{some }) = \text{true}$

0
5

$(5 \neq \text{some }) = \text{true}$ (since $0 \neq 5$)

$(= \text{some}) \equiv \text{in}$

However, $(\neq \text{some}) \equiv \text{not in}$



Definition of all Clause

- $F <\text{comp}> \text{all } r \Leftrightarrow \forall t \in r (F <\text{comp}>^t)$

$(5 < \text{all} \begin{array}{|c|}\hline 0 \\ \hline 5 \\ \hline 6 \\ \hline\end{array}) = \text{false}$

$(5 < \text{all} \begin{array}{|c|}\hline 6 \\ \hline 10 \\ \hline\end{array}) = \text{true}$

$(5 = \text{all} \begin{array}{|c|}\hline 4 \\ \hline 5 \\ \hline\end{array}) = \text{false}$

$(5 \neq \text{all} \begin{array}{|c|}\hline 4 \\ \hline 6 \\ \hline\end{array}) = \text{true} (\text{since } 5 \neq 4 \text{ and } 5 \neq 6)$

$(\neq \text{all}) \equiv \text{not in}$

However, $(= \text{all}) \equiv \text{in}$

/

Example Query

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

*select branch-name
from branch*

where assets > all

(select assets

from branch

where branch-city = 'Brooklyn')

Test for Empty Relations

- The exists construct returns the value true if the argument subquery is nonempty.
- $\text{exists } r \Leftrightarrow r \neq \emptyset$
- $\text{not exists } r \Leftrightarrow r = \emptyset$

Example Query

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

```
select distinct S.customer-name  
from depositor as S  
where not exists (
```

```
(select branch-name  
from branch  
where branch-city = 'Brooklyn')
```

except

```
(select R.branch-name  
from depositor as T, account as R  
where T.account-number = R.account-number and  
S.customer-name = T.customer-name))
```

- (Schema used in this example)
- Note that $X - Y = \emptyset \Leftrightarrow X \subseteq Y$
- Note: Cannot write this query using = all and its variants

Test for Absence of Duplicate

Tuples

- The `unique` construct tests whether a subquery has any duplicate tuples in its result.
- Find all customers who have at most one account at the Perryridge branch.

```
select T.customer-name
from depositor as T
where unique (
    select R.customer-name
    from account, depositor as R
    where T.customer-name =
R.customer-name and
R.account-number
```

Example Query

- Find all customers who have at least two accounts at the Perryridge branch.

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

- (Schema used in this example)

Views

- Provide a mechanism to hide certain data from the view of certain users.

To ^{create view} `v as <query expression>` we use the command:

where:

❖ `<query expression>` is any legal expression

❖ The view name is represented by `v`

Example Queries

- A view consisting of branches and their

~~Customers~~ *all-customer* as

```
(select branch-name, customer-name  
  from depositor, account  
 where depositor.account-number = account.account-number)  
 union  
(select branch-name, customer-name  
  from borrower, loan  
 where borrower.loan-number = loan.loan-number)
```

- Find all customers of the Perryridge branch

```
select customer-name  
  from all-customer  
 where branch-name = 'Perryridge'
```

Derived Relations

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

select *branch-name*, *avg-balance*
from (select *branch-name*, avg
(*balance*)
from *account*
group by *branch-name*)
as *result* (*branch-name*, *avg-*
balance)

With Clause

- With clause allows views to be defined locally to a query, rather than globally. Analogous to procedures in a programming language.
- Find all accounts with the maximum balance

```
with max-balance(value) as  
      select max (balance)  
            from account  
      select account-number
```

Complex Query using With

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

```
with branch-total(branch-name, value) as
  select branch-name, sum (balance)
    from account
   group by branch-name
with branch-total-avg(value) as
  select avg (value)
    from branch-total
select branch-name
  from branch-total, branch-total-avg
 where branch-total.value >= branch-total-avg.value
```

Modification of the Database – Deletion

Delete all account records at the Perryridge branch

delete from *account*
where *branch-name* =

‘Perryridge’

- Delete all accounts at every branch located in Needham city.

delete from *account*
where *branch-name* in (select *branch-name*

from *branch*
where *branch-city* =

‘Needham’)

Example Query

- Delete the record of all accounts with balances below the average at the bank.

```
delete.from account  
where balance < (select avg (balance)  
from account)
```

- ☞ Problem: as we delete tuples from *deposit*, the average balance changes
- ☞ Solution used in SQL:
 1. First, compute **avg** balance and find all tuples to delete
 2. Next, delete all tuples found above (without recomputing **avg** or retesting the tuples)

Modification of the Database – Insertion

- Add a new tuple to *account*

*insert into account
values ('A-9732',*

'Perryridge', 1200)

or equivalently

*insert into account (branch-name,
balance, account-number)*

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

- Add a new tuple to *account* with

Modification of the Database – Insertion

- Provide as a gift for all loan customers of the Perryridge branch, a \$200 savings account. Let the loan number serve as the account number for the new savings account

```
insert into account
select loan-number, branch-name,
200
from loan
where branch-name = 'Perryridge'
insert into depositor
select customer-name, loan-number
```

Modification of the Database – Updates

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

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

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

Case Statement for Conditional Updates

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

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

Update of a View

- Create a view of all loan data in *loan* relation, hiding the *amount* attribute

```
create view branch-loan as  
    select branch-name, loan-number  
        from loan
```

- Add a new tuple to *branch-loan*

```
insert into branch-loan  
    values ('Perryridge', 'L-307')
```

This insertion must be represented by the insertion of the tuple

(‘L-307’, ‘Perryridge’, *null*)

into the *loan* relation

- Updates on more complex views are difficult or impossible to translate, and hence are disallowed.
- Most SQL implementations allow updates only on simple views (without aggregates) defined on a single relation

Transactions

- A transaction is a sequence of queries and update statements executed as a single unit
 - Transactions are started implicitly and terminated by one of
 - **commit work:** makes all updates of the transaction permanent in the database
 - **rollback work:** undoes all updates performed by the transaction.
- Motivating example
 - Transfer of money from one account to another involves two steps:
 - deduct from one account and credit to another
 - If one step succeeds and the other fails, database is in an inconsistent state

Transactions (Cont.)

- In most database systems, each SQL statement that executes successfully is automatically committed.
 - Each transaction would then consist of only a single statement
 - Automatic commit can usually be turned off, allowing multi-statement transactions, but how to do so depends on the database system
 - Another option in SQL:1999: enclose statements within **begin atomic**

Joined Relations

- Join operations take two relations and return as a result another relation.
 - These additional operations are typically used as subquery expressions in the from clause
 - Join condition – defines which tuples in the two relations match. and what are pre defined in each relation that do not match any tuple in the other relation (based on the join
- | | |
|------------------|----------------------------------|
| Join Types | Join Conditions |
| inner join | natural |
| left outer join | on <predicate> |
| right outer join | using (A_1, A_2, \dots, A_n) |
| full outer join | |

Joined Relations - Datasets for Examples

<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

- Note: borrower information missing for L-260 and loan information missing for L-155

Joined Relations - Examples

- *loan inner join borrower on
loan.loan-number =*

loan-number	branch-name	amount	customer-name	loan-number
L-170	Downtown	3000	Jones	L-170
L-230	Redwood	4000	Smith	L-230

- *loan left outer join borrower on
loan.loan-number = borrower.loan-number*

loan-number	branch-name	amount	customer-name	loan-number
L-170	Downtown	3000	Jones	L-170
L-230	Redwood	4000	Smith	L-230
L-260	Perryridge	1700	null	null

Joined Relations - Examples

- *loan natural inner join 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

- loan **natural right outer join** *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	null	null	Hayes

Joined Relations - Examples

- *loan full outer join 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	null	null	Hayes

- Find all customers who have either an account or a loan (but not both) at the bank.

select *customer-name*

from (*depositor* natural full outer join *borrower*)
where *account-number* is null or *loan-number* is null

Data Definition Language (DDL)

Allows the specification of not only a set of relations but also information about each relation, including:

- The schema for each relation.
- The domain of values associated with each attribute.
- Integrity constraints
- The set of indices to be maintained for each relations.
- Security and authorization information for each relation.
- The physical storage structure of

Domain Types in SQL

- `char(n)`. Fixed length character string, with user-specified length n .
- `varchar(n)`. Variable length character strings, with user-specified maximum length n .
- `int`. Integer (a finite subset of the integers that is machine-dependent).
- `smallint`. Small integer (a machine-dependent subset of the integer domain type).
- `numeric(p,d)`. Fixed point number, with user-specified precision of p digits, with n digits to the right of decimal point.
- `real, double precision`. Floating point and double-precision floating point numbers, with machine-dependent precision.
- `float(n)`. Floating point number, with user-specified precision of at least n digits.
- Null values are allowed in all the domain types. Declaring an attribute to be not null prohibits null values for that attribute.
- `create domain` construct in SQL-92 creates user-defined domain types

`create domain person-name char(20) not null`

Date/Time Types in SQL (Cont.)

- date. Dates, containing a (4 digit) year, month and date
 - E.g. **date** ‘2001-7-27’
- time. Time of day, in hours, minutes and seconds.
 - E.g. **time** ’09:00:30’ **time**
’09:00:30.75’
- timestamp: date plus time of day
 - E.g. **timestamp** ‘2001-7-27
09:00:30.75’
- Interval: period of time
 - E.g. **Interval** ‘1’ day
 - Subtracting a date/time/timestamp value from another gives an interval value

Create Table Construct

- An SQL relation is defined using the create table command:

create table $r(A_1 D_1, A_2 D_2,$
 $\dots, A_n D_n,$

(integrity-
constraint₁),

...,

(integrity-
constraint_k))

- r is the name of the relation
- each A_i is an attribute name in the schema of relation r

Integrity Constraints in Create

- **not null**

Table

- primary key (A_1, \dots, A_n)
- check (P), where P is a predicate

Example: Declare *branch-name* as the primary key for *branch* and ensure that the values of *assets* are non-negative.

```
create table branch
  (branch-name char(15),
   branch-city  char(30)
   assets       integer,
   primary key (branch-name),
   check (assets >= 0))
```

primary key declaration on an attribute automatically ensures **not null** in SQL-92 onwards, needs to be explicitly stated in SQL-89

Drop and Alter Table Constructs

- The drop table command deletes all information about the dropped relation from the database.
- The alter table command is used to add attributes to an existing relation.

`alter table r add $A D$`

where A is the name of the attribute to be added to relation r and D is the domain of A .

- All tuples in the relation are assigned *null* as the value for the new attribute.

Embedded SQL

- The SQL standard defines embeddings of SQL in a variety of programming languages such as Pascal, PL/I, Fortran, C, and Cobol.
- A language to which SQL queries are embedded is referred to as a *host* language, and the SQL structures permitted in the host language comprise *embedded* SQL.
- The basic form of these languages follows that of the System R

Example Query

From within a host language, find the names and cities of customers with more than the variable *amount* dollars in some account.

- Specify the query in SQL and declare a *cursor* for it

EXEC SQL

```
declare c cursor for
select customer-name, customer-city
from depositor, customer, account
where depositor.customer-name =
customer.customer-name
and depositor account-number =
```

Embedded SQL (Cont.)

- The open statement causes the query to be evaluated

EXEC SQL open *c* END-EXEC

- The fetch statement causes the values of one tuple in the query result to be placed on host language variables.

EXEC SQL fetch *c* into :*cn*, :*cc* END-EXEC

Repeated calls to fetch get successive tuples in the query result

Updates Through Cursors

- Can update tuples fetched by cursor by declaring that the cursor is for update

```
declare c cursor for
    select *
    from account
    where branch-name = 'Perryridge'
    for update
```

- To update tuple at the current location of cursor

```
update account
set balance = balance + 100
where current of c
```

Dynamic SQL

- Allows programs to construct and submit SQL queries at run time.
- Example of the use of dynamic SQL from within a C program.

```
char * sqlprog = "update account  
    set balance =  
        balance * 1.05  
    where account-number  
        = ?"  
EXEC SQL prepare dynprog from  
    :sqlprog;
```

ODBC

Open DataBase Connectivity(ODBC) standard

- standard for application program to communicate with a database server.
- application program interface (API) to
 - open a connection with a database,
 - send queries and updates,
 - get back results.

Applications such as GUI, spreadsheets, etc. can use ODBC

ODBC (Cont.)

- Each database system supporting ODBC provides a "driver" library that must be linked with the client program.
- When client program makes an ODBC API call, the code in the library communicates with the server to carry out the requested action, and fetch results.
- ODBC program first allocates an SQL environment, then a database connection handle.
- Opens database connection using SQLConnect(). Parameters for SQLConnect:
 - connection handle,
 - the server to which to connect
 - the user identifier,
 - password
- Must also specify types of arguments:
 - SQL_NTS denotes previous argument is a null-terminated string.

ODBC Code

```
■ int ODBCexample()
{
    RETCODE error;
    HENV    env;    /* environment */
    HDBC    conn;   /* database connection */
    SQLAllocEnv(&env);
    SQLAllocConnect(env, &conn);
    SQLConnect(conn, "aura.bell-
        labs.com", SQL_NTS, "avi", SQL_NTS, "avipass
        wd", SQL_NTS);
    { .... Do actual work ... }

    SQLDisconnect(conn);
    SQLFreeConnect(conn);
    SQLFreeEnv(env);
```

ODBC Code (Cont.)

- Program sends SQL commands to the database by using SQLExecDirect
- Result tuples are fetched using SQLFetch()
- SQLBindCol() binds C language variables to attributes of the query result
 - When a tuple is fetched, its attribute values are automatically stored in corresponding C variables.
 - Arguments to SQLBindCol()
 - ODBC stmt variable, attribute position in query result
 - The type conversion from SQL to C.
 - The address of the variable.
 - For variable-length types like character arrays,
 - The maximum length of the variable
 - Location to store actual length when a tuple is fetched.

ODBC Code (Cont.)

- Main body of program

```
char branchname[80];
float balance;
int lenOut1, lenOut2;
HSTMT stmt;
SQLAllocStmt(conn, &stmt);
char * sqlquery = "select branch_name, sum
(balance)
from account
group by branch_name";
error = SQLExecDirect(stmt, sqlquery,
SQL_NTS);
if (error == SQL_SUCCESS) {
SQLBindCol(stmt, 1, SQL_C_CHAR,
```

More ODBC Features

■ Prepared Statement

- SQL statement prepared: compiled at the database
- Can have placeholders: E.g. insert into account values(?, ?, ?)
- Repeatedly executed with actual values for the placeholders

■ Metadata features

- finding all the relations in the database and
- finding the names and types of columns of a query result or a relation in the database.

■ By default, each SQL statement is treated as a separate transaction that is

ODBC Conformance Levels

- Conformance levels specify subsets of the functionality defined by the standard.

- Core
- Level 1 requires support for metadata querying
- Level 2 requires ability to send and retrieve arrays of parameter values and more detailed catalog information.

- SQL Call Level Interface (CLI) standard similar to ODBC interface, but with some minor differences.

JDBC

- JDBC is a Java API for communicating with database systems supporting SQL
- JDBC supports a variety of features for querying and updating data, and for retrieving query results
- JDBC also supports metadata retrieval, such as querying about relations present in the database and the names and types of relation attributes
- Model for communicating with the database:

JDBC Code

```
public static void JDBCexample(String dbid,  
String userid, String passwd)  
{  
    try {  
        Class.forName ("oracle.jdbc.driver.OracleDriver");  
        Connection conn =  
        DriverManager.getConnection(  
        "jdbc:oracle:thin:@aura.bell-  
        labs.com:2000:bankdb", userid, passwd);  
        Statement stmt = conn.createStatement();  
        ... Do Actual Work ....  
        stmt.close();  
        conn.close();  
    }  
    catch (SQLException sale) {
```

JDBC Code (Cont.)

- Update to database

```
try {  
    stmt.executeUpdate( "insert into account  
values  
        ('A-  
9732', 'Perryridge', 1200)");  
} catch (SQLException sqle) {  
    System.out.println("Could not insert tuple.  
" + sqle);  
}
```

- Execute query and fetch and print results

```
ResultSet rset = stmt.executeQuery( "select  
branch_name, avg(balance)
```

JDBC Code Details

- Getting result fields:
 - `rs.getString("branchname")` and
`rs.getString(1)` equivalent if branchname is
the first argument of select result.
- Dealing with Null values

```
int a = rs.getInt("a");
```

```
if (rs.wasNull()) System.out.println("Got null  
value");
```

Prepared Statement

- Prepared statement allows queries to be compiled and executed multiple times with different arguments

```
PreparedStatement pStmt =  
conn.prepareStatement(  
"insert into  
account values(?, ?, ?)");  
pStmt.setString(1,  
"A-9732");  
pStmt.setString(2, "Perryridge");  
pStmt.setInt(3, 1200);  
pStmt.executeUpdate();  
  
pStmt.setString(1, "A-9733");  
pStmt.executeUpdate();
```

Other SQL Features

- SQL sessions
 - client *connects* to an SQL server, establishing a session
 - executes a series of statements
 - *disconnects* the session
 - can *commit* or *rollback* the work carried out in the session
- An SQL environment contains several components, including a user identifier, and a *schema*, which identifies which of several schemas a session is using.

Schemas, Catalogs, and

Environments

- Three-level hierarchy for naming relations.
 - Database contains multiple **catalogs**
 - each catalog can contain multiple **schemas**
 - SQL objects such as relations and views are contained within a schema
- e.g. catalog5.bank-schema.account
- Each user has a default catalog and schema, and the combination is unique to the user.
- Default catalog and schema are set up for a connection

Procedural Extensions and Stored Procedures

- SQL provides a module language
 - permits definition of procedures in SQL, with if-then-else statements, for and while loops, etc.
 - more in Chapter 9
- Stored Procedures
 - Can store procedures in the database
 - then execute them using the **call** statement
 - permit external applications to operate on the database without knowing about internal details
- These features are covered in Chapter

EXTRA MATERIAL ON JDBC AND APPLICATION ARCHITECTURES



Transactions in JDBC

- As with ODBC, each statement gets committed automatically in JDBC
- To turn off auto commit use `conn.setAutoCommit(false);`
- To commit or abort transactions use `conn.commit()` or `conn.rollback()`
- To turn auto commit on again, use `conn.setAutoCommit(true);`

Procedure and Function Calls in JDBC

JDBC provides a class `CallableStatement` which allows SQL stored procedures/functions to be invoked.

```
CallableStatement cs1 =  
conn.prepareCall( "{call proc (?,?)}" );
```

```
CallableStatement cs2 =  
conn.prepareCall( "{? = call func (?,?)}" );
```

Result Set MetaData

- The class ResultSetMetaData provides information about all the columns of the ResultSet.
- Instance of this class is obtained by getMetaData() function of ResultSet.
- Provides Functions for getting number of columns, column name, type, precision, scale, table from which the column is derived etc.

```
ResultSetMetaData rsmd =  
rs.getMetaData ( );  
for ( int i = 1; i <=
```

Database Meta Data

- The class DatabaseMetaData provides information about database relations
- Has functions for getting all tables, all columns of the table, primary keys etc.
- E.g. to print column names and types of a relation

```
DatabaseMetaData dbmd = conn.getMetaData();
ResultSet rs = dbmd.getColumns( null, "BANK-DB", "account",
"%");
//Arguments: catalog, schema-pattern, table-pattern,
column-pattern
// Returns: 1 row for each column, with several attributes
such as
//           COLUMN_NAME, TYPE_NAME, etc.
while ( rs.next() ) {
    System.out.println( rs.getString("COLUMN_NAME") ,
                        rs.getString("TYPE_NAME"));
}
```

- There are also functions for getting information such as
 - Foreign key references in the schema

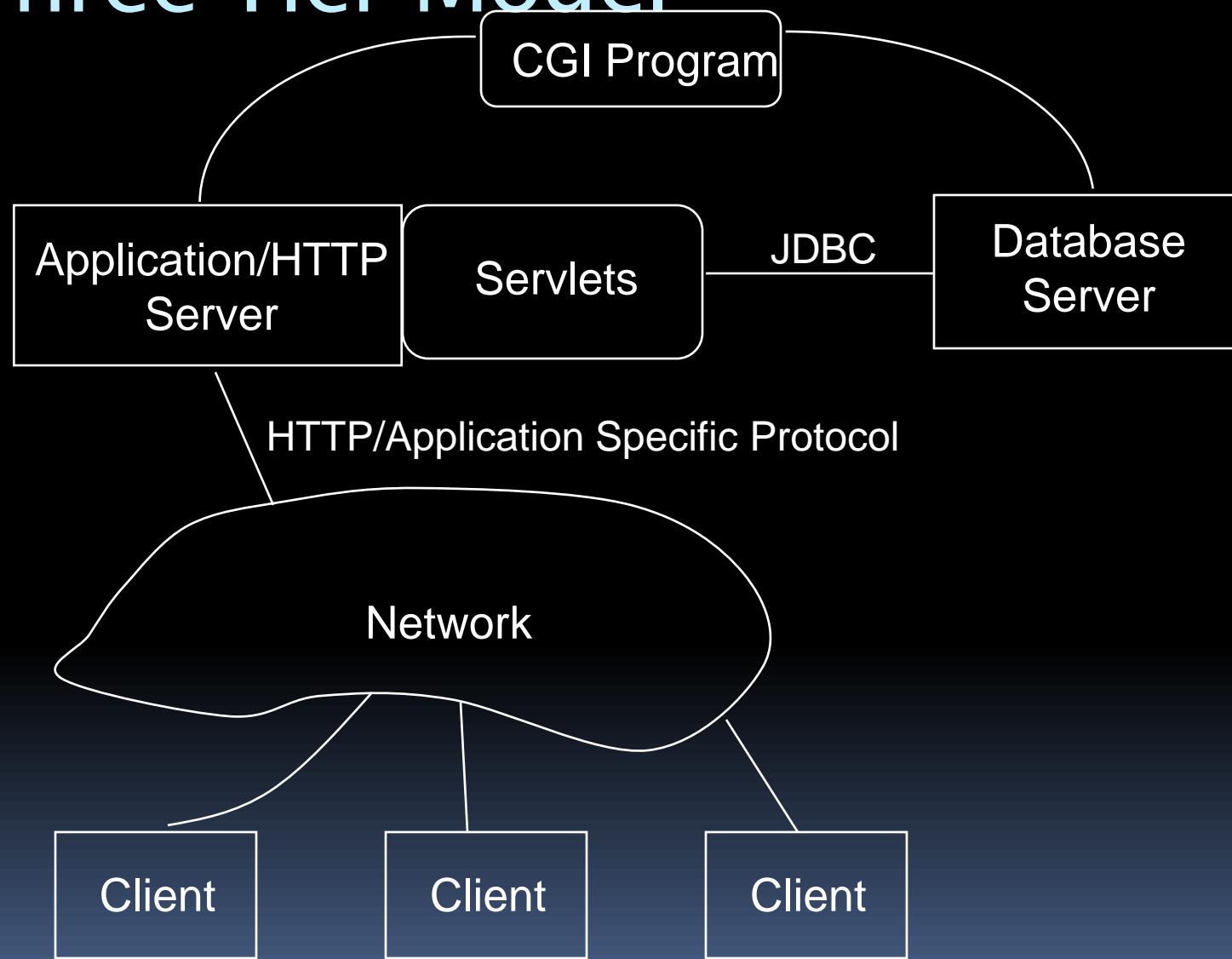
Application Architectures

- Applications can be built using one of two architectures
 - Two tier model
 - Application program running at user site directly uses JDBC/ODBC to communicate with the database
 - Three tier model
 - Users/programs running at user sites communicate with an application server. The application server in turn communicates with the database

Two-tier Model

- E.g. Java code runs at client site and uses JDBC to communicate with the backend server
- Benefits:
 - flexible, need not be restricted to predefined queries
- Problems:
 - Security: passwords available at client site, all database operation possible
 - More code shipped to client
 - Not appropriate across organizations, or in large ones like universities

Three Tier Model



Three-tier Model (Cont.)

- E.g. Web client + Java Servlet using JDBC to talk with database server
- Client sends request over http or application-specific protocol
- Application or Web server receives request
- Request handled by CGI program or servlets
- Security handled by application at server
 - Better security
 - Fine granularity security

END OF CHAPTER



The *loan* and *borrower* Relations

<i>loan-number</i>	<i>branch-name</i>	<i>amount</i>	<i>customer-name</i>	<i>loan-number</i>
L-170	Downtown	3000	Jones	L-170
L-230	Redwood	4000	Smith	L-230
L-260	Perryridge	1700	Hayes	L-155
<i>loan</i>			<i>borrower</i>	

The Result of *loan* inner join
borrower on *loan.loan-number* =
borrower.loan-number

Γ-Σ30	Keewoob	₹000	Smith	Γ-Σ30
Γ-ΣΔ0	Dominion	3000	Jones	Γ-ΣΔ0
JOIN-NNNNNN	NNNN-NNNN	NNNN	NNNN-NNNN	JOIN-NNNNNN

The Result of *loan* left outer join *borrower* on *loan-number*

<i>loan-number</i>	<i>branch-name</i>	<i>amount</i>	<i>customer-name</i>	<i>loan-number</i>
L-170	Downtown	3000	Jones	L-170
L-230	Redwood	4000	Smith	L-230
L-260	Perryridge	1700	<i>null</i>	<i>null</i>

The Result of *loan* natural inner join *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

Join Types and Join Conditions

Join types

inner join

left outer join

right outer join

full outer join

Join Conditions

natural

on < predicate >

using (A_1, A_1, \dots, A_n)

The Result of *loan* natural right outer join *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

The Result of *loan* full outer join *borrower* using(*loan-number*)

<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

SQL Data Definition for Part of the Bank Database

```
create table customer
  (customer-name  char(20),
   customer-street  char(30),
   customer-city    char(30),
   primary key (customer-name))
```

```
create table branch
  (branch-name     char(15),
   branch-city      char(30),
   assets          integer,
   primary key (branch-name),
   check (assets >= 0))
```

```
create table account
  (account-number  char(10),
   branch-name     char(15),
   balance         integer,
   primary key (account-number),
   check (balance >= 0))
```

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

Chapter 5: Other Relational Languages

- Query-by-Example (QBE)
- Datalog

Query-by-Example (QBE)

- Basic Structure
- Queries on One Relation
- Queries on Several Relations
- The Condition Box
- The Result Relation
- Ordering the Display of Tuples
- Aggregate Operations
- Modification of the Database

QBE — Basic Structure

- A graphical query language which is based (roughly) on the domain relational calculus
- Two dimensional syntax – system creates templates of relations that are requested by users
- Queries are expressed “by example”

QBE Skeleton Tables for the Bank Example

<i>branch</i>	<i>branch-name</i>	<i>branch-city</i>	<i>assets</i>

<i>customer</i>	<i>customer-name</i>	<i>customer-street</i>	<i>customer-city</i>

<i>loan</i>	<i>loan-number</i>	<i>branch-name</i>	<i>amount</i>

QBE Skeleton Tables (Cont.)

<i>borrower</i>	<i>customer-name</i>	<i>loan-number</i>

<i>account</i>	<i>account-number</i>	<i>branch-name</i>	<i>balance</i>

<i>depositor</i>	<i>customer-name</i>	<i>account-number</i>

Queries on One Relation

- Find all loan numbers at the Perryridge branch.

<i>loan</i>	<i>loan-number</i>	<i>branch-name</i>	<i>amount</i>
	P._x	Perryridge	

- _x is a variable (optional; can be omitted in above query)
- P. means print (display)
- duplicates are removed by default
- To retain duplicates use P.ALL

<i>loan</i>	<i>loan-number</i>	<i>branch-name</i>	<i>amount</i>
	P.ALL.	Perryridge	

Queries on One Relation (Cont.)

- Display full details of all loans

Method 1:

<i>loan</i>	<i>loan-number</i>	<i>branch-name</i>	<i>amount</i>

Method 2: Shorthand notation

<i>loan</i>	<i>loan-number</i>	<i>branch-name</i>	<i>amount</i>
P.			

Queries on One Relation (Cont.)

- Find the loan number of all loans with a loan amount of more than \$700

<i>loan</i>	<i>loan-number</i>	<i>branch-name</i>	<i>amount</i>
	P.		>700

- Find names of all branches that are

<i>branch</i>	<i>branch-name</i>	<i>branch-city</i>	<i>assets</i>
	P.	¬ Brooklyn	

Queries on One Relation (Cont.)

- Find the loan numbers of all loans made jointly to Smith and Jones.

<i>borrower</i>	<i>customer-name</i>	<i>loan-number</i>
	“Smith”	P. $_x$
	“Jones”	$_x$

- Find all customers who live in the same city as Jones

<i>customer</i>	<i>customer-name</i>	<i>customer-street</i>	<i>customer-city</i>
	P. $_x$ Jones		$-y$ $-y$

Queries on Several Relations

- Find the names of all customers who have a loan from the

<i>loan</i>	<i>loan-number</i>	<i>branch-name</i>	<i>amount</i>
	$_x$	Perryridge	
<i>borrower</i>	<i>customer-name</i>	<i>loan-number</i>	
	P. $_y$	$_x$	

Queries on Several Relations

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

<i>depositor</i>	<i>customer-name</i>	<i>account-number</i>
	P. $_x$	

<i>borrower</i>	<i>customer-name</i>	<i>loan-number</i>
	$_x$	

Negation in QBE

- Find the names of all customers who have an account at the bank, but do not have a loan from the bank.

<i>depositor</i>	<i>customer-name</i>	<i>account-number</i>
	P. $_x$	
<i>borrower</i>	<i>customer-name</i>	<i>loan-number</i>
\neg	$_x$	

\neg means “there does not exist”

Negation in QBE (Cont.)

- Find all customers who have at least two accounts

<i>depositor</i>	<i>customer-name</i>	<i>account-number</i>
	P_x $_x$	$_y$ $\neg _y$

\neg means “not equal to”

The Condition Box

- Allows the expression of constraints on domain variables that are either inconvenient or impossible to express within the skeleton tables.
- Complex conditions can be used in

<i>borrower</i>	<i>customer-name</i>	<i>loan-number</i>
	$_n$	P_x

conditions

$_n = \text{Smith} \text{ or } _n = \text{Jones}$

Condition Box (Cont.)

<i>branch</i>	<i>branch-name</i>	<i>branch-city</i>	<i>assets</i>
	P.	$\neg x$	

conditions

$\neg x = (\text{Brooklyn} \text{ or } \text{Queens})$

Condition Box (Cont.)

- Find all account numbers with a balance between \$1,300 and \$1,500

<i>account</i>	<i>account-number</i>	<i>branch-name</i>	<i>balance</i>
	P.		$-x$

conditions

$$\begin{aligned} -x &\geq 1300 \\ -x &\leq 1500 \end{aligned}$$

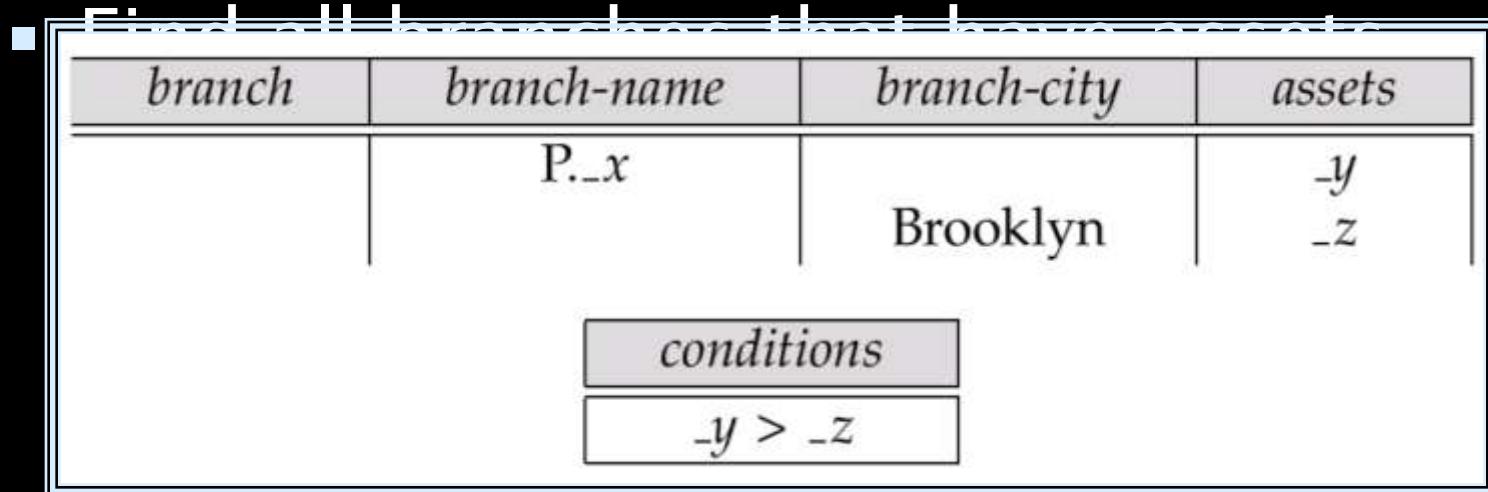
- Find all account numbers with a balance between \$1,300 and \$2,000 but not exactly \$1,500.

<i>account</i>	<i>account-number</i>	<i>branch-name</i>	<i>balance</i>
	P.		$-x$

conditions

$$-x = (\geq 1300 \text{ and } \leq 2000 \text{ and } \neg 1500)$$

Condition Box (Cont.)

■ 

<i>branch</i>	<i>branch-name</i>	<i>branch-city</i>	<i>assets</i>
	P. $_x$	Brooklyn	$_y$ $_z$

conditions

$_y > _z$

The Result Relation

- Find the *customer-name*, *account-number*, and *balance* for all customers who have an account at the Perryridge branch.
 - We need to:
 - Join *depositor* and *account*.
 - Project *customer-name*, *account-number* and *balance*.
 - To accomplish this we:
 - Create a skeleton table, called *result*, with attributes *customer-name*, *account-number* and *balance*.

The Result Relation (Cont.)

- The resulting query is:

<i>account</i>	<i>account-number</i>	<i>branch-name</i>	<i>balance</i>
	$-y$	Perryridge	$-z$
<i>depositor</i>	<i>customer-name</i>	<i>account-number</i>	
	$-x$		$-y$
<i>result</i>	<i>customer-name</i>	<i>account-number</i>	<i>balance</i>
P.	$-x$	$-y$	$-z$

Ordering the Display of Tuples

- AO = ascending order; DO = descending order.

<i>depositor</i>	<i>customer-name</i>	<i>account-number</i>
	P.AO.	

<i>account</i>	<i>account-number</i>	<i>branch-name</i>	<i>balance</i>
	P.AO(1).	Perryridge	P.DO(2).

with each sort operator (AO or DO) an integer surrounded by parentheses

Aggregate Operations

- The aggregate operators are AVG, MAX, MIN, SUM, and CNT
- The above operators must be postfixed with “ALL”
(e.g., SUM.ALL.or AVG.ALL._x) to

<i>account</i>	<i>account-number</i>	<i>branch-name</i>	<i>balance</i>
		Perryridge	P.SUM.ALL.

- E.g. Find the total balance of all the accounts maintained at the Perryridge branch.

Aggregate Operations (Cont.)

- UNQ is used to specify that we want to eliminate duplicates

- ~~Find the total number of customers~~

<i>depositor</i>	<i>customer-name</i>	<i>account-number</i>
	P.CNT.UNQ.	

Query Examples

account	account-number	branch-name	balance
		P.G.	P.AVG.ALL._x

- The “G” in “P.G” is analogous to SQL’s **group by** construct
- The “ALL” in the “P.AVG.ALL” entry in the *balance* column ensures that all balances are considered
- To find the average account balance at only those branches where the average account balance is more than \$1,200, we simply add the condition box:

<i>conditions</i>
AVG.ALL._x > 1200

Query Example

- Find all customers who have an account at all branches located in Brooklyn.
 - Approach: for each customer, find the number of branches in Brooklyn at which they have accounts, and compare with total number of branches in Brooklyn
- In the query on the next page
 - **QBE does not provide subquery functionality, so both above tasks have to be combined in a single query.**
 - CNT.UNQ.ALL. w specifies the number of distinct branches in Brooklyn.
Note: The variable w is not connected to other variables in the query
 - CNT.UNQ.X specifies the number of distinct branches in Brooklyn at which customer x has an account.
 - Can be done for this query, but there are queries that require subqueries and cannot be expressed in QBE always be done.

Query Example (Cont.)

<i>depositor</i>	<i>customer-name</i>	<i>account-number</i>	
	P.G._ <i>x</i>	- <i>y</i>	
<i>account</i>	<i>account-number</i>	<i>branch-name</i>	<i>balance</i>
	- <i>y</i>	- <i>z</i>	
<i>branch</i>	<i>branch-name</i>	<i>branch-city</i>	<i>assets</i>
	- <i>z</i> - <i>w</i>	Brooklyn Brooklyn	

conditions

CNT.UNQ._*z* =
CNT.UNQ._*w*

Modification of the Database – Deletion

- Deletion of tuples from a relation is expressed by use of a D. command. In the case where we delete information in only some of the columns, null

<i>customer</i>	<i>customer-name</i>	<i>customer-street</i>	<i>customer-city</i>
D.	Smith		

~~Delete customer Smith~~

<i>branch</i>	<i>branch-name</i>	<i>branch-city</i>	<i>assets</i>
	Perryridge		D.

- Delete the *branch-city* value of the branch whose name is “Perryridge”

Deletion Query Examples

- Delete all loans with a loan amount between \$1300 and \$1500.

~~For consistency we have to delete~~

<i>loan</i>	<i>loan-number</i>	<i>branch-name</i>	<i>amount</i>
D.	$-y$		$-x$

<i>borrower</i>	<i>customer-name</i>	<i>loan-number</i>
D.		$-y$

conditions

$-x = (\geq 1300 \text{ and } \leq 1500)$

Deletion Query Examples

(Cont.) Delete all accounts at branches located in Brooklyn.

<i>account</i>	<i>account-number</i>	<i>branch-name</i>	<i>balance</i>
D.	-y	-x	
<i>depositor</i>	<i>customer-name</i>	<i>account-number</i>	
D.			-y
<i>branch</i>	<i>branch-name</i>	<i>branch-city</i>	<i>assets</i>
	-x	Brooklyn	

Modification of the Database – Insertion

- Insertion is done by placing the `I.` operator in the query expression.
- Insert the fact that account A-9732 at the Perryridge branch has

<i>account</i>	<i>account-number</i>	<i>branch-name</i>	<i>balance</i>
I.	A-9732	Perryridge	700

Modification of the Database – Insertion (Cont.)

Provide as a gift for all loan customers of the Perryridge branch, a new \$200 savings account for every loan account they have, with the loan number serving as the account number for the new savings account.

<i>account</i>	<i>account-number</i>	<i>branch-name</i>	<i>balance</i>
I.	$-x$	Perryridge	200

<i>depositor</i>	<i>customer-name</i>	<i>account-number</i>
I.	$-y$	$-x$

<i>loan</i>	<i>loan-number</i>	<i>branch-name</i>	<i>amount</i>
	$-x$	Perryridge	

<i>borrower</i>	<i>customer-name</i>	<i>loan-number</i>
	$-y$	$-x$

Modification of the Database – Updates

- Use the U. operator to change a value in a tuple without changing *all* values in the tuple. QBE does not allow users to update the primary key fields

<i>branch</i>	<i>branch-name</i>	<i>branch-city</i>	<i>assets</i>
	Perryridge		U.10000000

Perryridge branch to \$10,000,000.

<i>account</i>	<i>account-number</i>	<i>branch-name</i>	<i>balance</i>
			U._x * 1.05

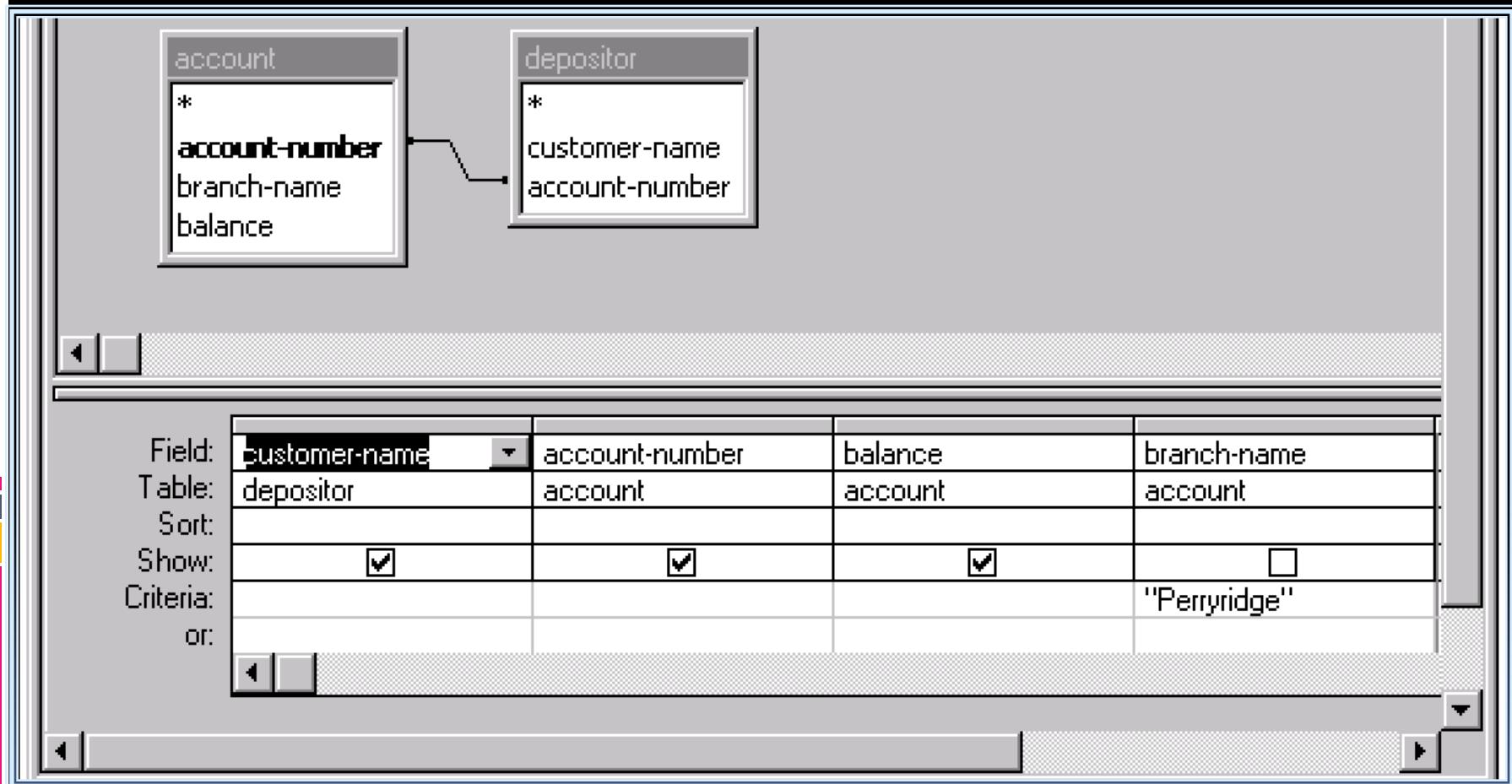
- Increase all balances by 5 percent.

Microsoft Access QBE

- Microsoft Access supports a variant of QBE called Graphical Query By Example (GQBE)
- GQBE differs from QBE in the following ways
 - Attributes of relations are listed vertically, one below the other, instead of horizontally
 - Instead of using variables, lines (links) between attributes are used to specify that their values should be the same.
 - Links are added automatically on the basis of attribute name, and the user can then add

An Example Query in Microsoft Access QBE

- Example query: Find the *customer-name*, *account-number* and *balance* for all accounts at the Perryridge branch



An Aggregation Query in Access QBE

- Find the *name*, *street* and *city* of all customers who have more than one account at the bank

The screenshot shows the Microsoft Access Query by Example (QBE) grid. At the top, there are two tables: "customer" and "depositor". The "customer" table has fields: * (selected), customer-name, customer-street, and customer-city. The "depositor" table has fields: * (selected), customer-name, and account-number. A relationship line connects the customer-name field in the "customer" table to the customer-name field in the "depositor" table.

The QBE grid below defines the query:

Field:	customer-name	customer-street	customer-city	account-number
Table:	customer	customer	customer	depositor
Total:	Group By	Group By	Group By	Count
Sort:				
Show:	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>
Criteria:				>1
or:				

Aggregation in Access QBE

- The row labeled Total specifies
 - which attributes are group by attributes
 - which attributes are to be aggregated upon (and the aggregate function).
 - For attributes that are neither group by nor aggregated, we can still specify conditions by selecting **where** in the Total row and listing the conditions below
- As in SQL, if group by is used, only group by attributes and aggregate results can be output

Datalog

- Basic Structure
- Syntax of Datalog Rules
- Semantics of Nonrecursive Datalog
- Safety
- Relational Operations in Datalog
- Recursion in Datalog
- The Power of Recursion

Basic Structure

- Prolog-like logic-based language that allows recursive queries; based on first-order logic.
- A Datalog program consists of a set of *rules* that define views.
- Example: define a view relation *v1* containing account numbers and balances for accounts at the Perryridge branch with a balance of over \$700.

v1(A, B) :- account(A, ‘Perryridge’, B), B > 700.

- Retrieve the balance of account

Example Queries

- Each rule defines a set of tuples that a view relation must contain.
 - E.g. $v1(A, B) :- account(A, \text{``Perryridge''}, B), B > 700$ is read as
 - for all A, B
 - if $(A, \text{``Perryridge''}, B) \in account$ and $B > 700$
 - then $(A, B) \in v1$
- The set of tuples in a view relation is then defined as the union of all the sets of tuples defined by the rules for the view relation.
- Example:

$interest-rate(A, 5) :- account(A, N,$

Negation in Datalog

- Define a view relation c that contains the names of all customers who have a deposit but no loan at the bank:
 $c(N) :- depositor(N, A), \text{not } is-borrower(N).$
 $is-borrower(N) :- borrower(N, L).$
- NOTE: using $\text{not } borrower(N, L)$ in the first rule results in a different meaning, namely there is some loan L for which N is not a borrower.

■ To prevent such confusion, we require all

Named Attribute Notation

- Datalog rules use a positional notation, which is convenient for relations with a small number of attributes
- It is easy to extend Datalog to support named attributes.
 - E.g., $v1$ can be defined using named attributes as

$v1(account-number A, balance B) :-$
 $account(account-number A, branch-$
 $name 'Perryridge', balance B),$
 $B > 700.$

Formal Syntax and Semantics of Datalog

- We formally define the syntax and semantics (meaning) of Datalog programs, in the following steps
 1. We define the syntax of predicates, and then the syntax of rules
 2. We define the semantics of individual rules
 3. We define the semantics of non-recursive programs, based on a layering of rules
 4. It is possible to write rules that can generate an infinite number of tuples in the view relation. To prevent this, we

Syntax of Datalog Rules

- A *positive literal* has the form

$$p(t_1, t_2 \dots, t_n)$$

- p is the name of a relation with n attributes
- each t_i is either a constant or variable
- A *negative literal* has the form

$$\text{not } p(t_1, t_2 \dots, t_n)$$

- Comparison operations are treated as positive predicates

- E.g. $X > Y$ is treated as a predicate $>(X, Y)$
- “ $>$ ” is conceptually an (infinite) relation that contains all pairs of values such that $x > y$

Syntax of Datalog Rules (Cont.)

- *Rules* are built out of literals and have the form:

$$p(t_1, t_2, \dots, t_n) :- L_1, L_2, \dots, L_m.$$

head *body*

- each of the L_i 's is a literal
- head - the literal $p(t_1, t_2, \dots, t_n)$
- body - the rest of the literals
- A *fact* is a rule with an empty body, written in the form:

Semantics of a Rule

- A *ground instantiation of a rule* (or simply *instantiation*) is the result of replacing each variable in the rule by some constant.
 - Eg. Rule defining $v1$

$v1(A, B) :- account(A, "Perryridge", B), B > 700.$

- An instantiation above rule:

$v1("A-217", 750) :- account("A-217", "Perryridge", 750),$
 $750 > 700.$

- The body of rule instantiation R' is *satisfied* in a set of facts (database instance) / if

Semantics of a Rule (Cont.)

- We define the set of facts that can be inferred from a given set of facts / using rule R as:

$\text{infer}(R, I) = \{p(t_1, \dots, t_n) \mid \text{there is a}$
 $\text{ground instantiation } R' \text{ of } R$
 $\text{where } p(t_1, \dots, t_n)$ is
 $\text{the head of } R', \text{ and}$
 $\text{the body of } R' \text{ is}$
 $\text{satisfied in } I\}$

- Given a set of rules $\mathcal{R} = \{R_1, R_2, \dots, R_n\}$, we define

$$\text{infer}(\mathfrak{R}, I) = \text{infer}(R_1, I) \cup \text{infer}(R_2, I) \cup \dots$$

Layering of Rules

- Define the interest on each account in Perryridge

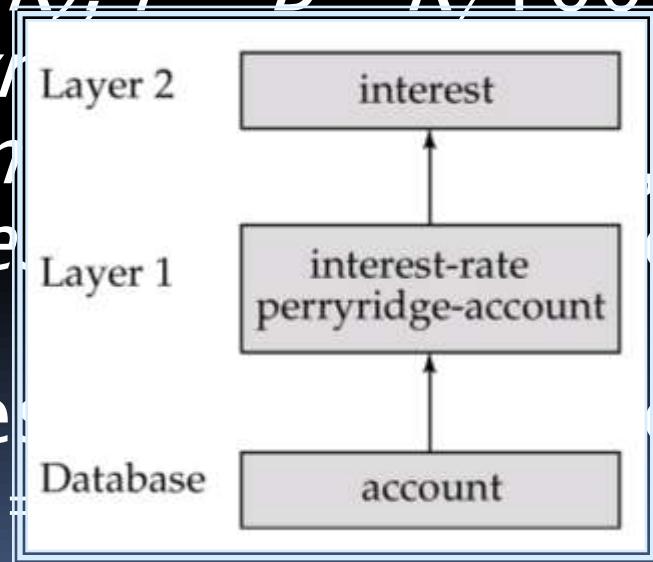
interest(A, I) :- perryridge-account(A,B),

*interest-rate(A,R), I = B * R/100.*

perryridge-account(A,B), B < interest-rate(A,R).

interest(A,B), B > interest-rate(A,R).

interest(A,B), B >



B) :- count(N, A,
B).
count(N, A,

- Layering of the view relations

Layering Rules (Cont.)

Formally:

- A relation is a layer 1 if all relations used in the bodies of rules defining it are stored in the database.
- A relation is a layer 2 if all relations used in the bodies of rules defining it are either stored in the database, or are in layer 1.
- A relation p is in layer $i + 1$ if
 - it is not in layers 1, 2, ..., i
 - all relations used in the bodies of rules defining a p are either stored in the

Semantics of a Program

Let the layers in a given program be $1, 2, \dots, n$. Let \mathfrak{R}_i denote the set of all rules defining view relations in layer i .

- Define $I_0 = \text{set of facts stored in the database}$.
- Recursively define $I_{i+1} = I_i \cup \text{infer}(\mathfrak{R}_{i+1}, I_i)$
- The set of facts in the view relations defined by the program (also called the semantics of the program) is given by the set of facts I_n corresponding to the highest layer n .

Note: Can instead define semantics using view expansion like in relational algebra, but above definition is better for handling extensions such as recursion.

Safety

- It is possible to write rules that generate an infinite number of answers.

$$gt(X, Y) :- X > Y$$
$$\text{not-in-loan}(B, L) :- \text{not } loan(B, L)$$

To avoid this possibility Datalog rules must satisfy the following conditions.

- Every variable that appears in the head of the rule also appears in a non-arithmetic positive literal in the body of the rule.

Relational Operations in Datalog

- Project out attribute *account-name* from account.

$query(A) :- account(A, N, B).$

- Cartesian product of relations r_1 and r_2 .

$query(X_1, X_2, \dots, X_n, Y_1, Y_1, Y_2, \dots, Y_m) :-$

$r_1(X_1, X_2, \dots, X_n), r_2(Y_1, Y_2, \dots, Y_m).$

- *Union of relations* r_1 and r_2 .

Updates in Datalog

- Some Datalog extensions support database modification using + or - in the rule head to indicate insertion and deletion.
- E.g. to transfer all accounts at the Perryridge branch to the Johnstown branch, we can write

```
+ account(A, "Johnstown", B) :- account  
(A, "Perryridge", B).  
- account(A, "Perryridge", B) :- account  
(A, "Perryridge", B)
```

Recursion in Datalog

- Suppose we are given a relation $manager(X, Y)$ containing pairs of names X, Y such that Y is a manager of X (or equivalently, X is a direct employee of Y).
- Each manager may have direct employees, as well as indirect employees
 - Indirect employees of a manager, say Jones, are employees of people who are

Semantics of Recursion in Datalog

- Assumption (for now): program contains no negative literals
- The view relations of a recursive program containing a set of rules \mathcal{R} are defined to contain exactly the set of facts / computed by the iterative procedure *Datalog-Fixpoint*

```
procedure Datalog-Fixpoint  
  / = set of facts in the  
  database  
  repeat
```

Example of Datalog-FixPoint Iteration

<i>employee-name</i>	<i>manager-name</i>
Alon	Barinsky
Barinsky	Estovar
Corbin	Duarte
Duarte	Jones
Estovar	Jones
Jones	Klinger
Rensal	Klinger

Iteration number	Tuples in <i>empl-jones</i>
0	
1	(Duarte), (Estovar)
2	(Duarte), (Estovar), (Barinsky), (Corbin)
3	(Duarte), (Estovar), (Barinsky), (Corbin), (Alon)
4	(Duarte), (Estovar), (Barinsky), (Corbin), (Alon)

A More General View

- Create a view relation *empl* that contains every tuple (X, Y) such that X is directly or indirectly managed by Y .

empl(X, Y) :- manager(X, Y).

empl(X, Y) :-

manager(X, Z), empl(Z, Y)

- Find the direct and indirect employees of Jones.

? *empl(X, “Jones”).*

- Can define the view *empl* in another

The Power of Recursion

- Recursive views make it possible to write queries, such as transitive closure queries, that cannot be written without recursion or iteration.
 - Intuition: Without recursion, a non-recursive non-iterative program can perform only a fixed number of joins of manager with itself
 - This can give only a fixed number of levels of managers
 - Given a program we can construct a database with a greater number of levels of managers on which the program will not work

Recursion in SQL

- SQL:1999 permits recursive view definition
- E.g. query to find all employee-manager pairs

with recursive *empl*(*emp*, *mgr*) as (
 select *emp*, *mgr*
 from *manager*
 union
 select *manager.emp*,
 empl.mgr

Monotonicity

- A view V is said to be monotonic if given any two sets of facts I_1 and I_2 such that $I_1 \subseteq I_2$, then $E_V(I_1) \subseteq E_V(I_2)$, where E_V is the expression used to define V .
- A set of rules R is said to be monotonic if $I_1 \subseteq I_2$ implies $\text{infer}(R, I_1) \subseteq \text{infer}(R, I_2)$,
- Relational algebra views defined using only the operations: \prod , σ , \times , \cup , \cap and

Non-Monotonicity

- Procedure *Datalog-Fixpoint* is sound provided the rules in the program are monotonic.
 - Otherwise, it may make some inferences in an iteration that cannot be made in a later iteration. E.g. given the rules

$a :- \text{not } b.$

$b :- c.$

$c.$

Then a can be inferred initially, before b is inferred, but not later.

- We can extend the procedure to

Stratified Negation

- A Datalog program is said to be stratified if its predicates can be given layer numbers such that
 1. For all positive literals, say q , in the body of any rule with head, say, p
$$p(..) :- \dots, q(..), \dots$$
then the layer number of p is greater than or equal to the layer number of q
 2. Given any rule with a negative literal
$$p(..) :- \dots, \text{not } q(..), \dots$$
then the layer number of p is strictly greater than the layer number of q
- Stratified programs do not have recursion mixed with negation

Non-Monotonicity (Cont.)

- There are useful queries that cannot be expressed by a stratified program
 - E.g., given information about the number of each subpart in each part, in a part–subpart hierarchy, find the total number of subparts of each part.
 - A program to compute the above query would have to mix aggregation with recursion
 - However, so long as the underlying data (part–subpart) has no cycles, it is possible to write a program that mixes aggregation

Forms and Graphical User Interfaces

- Most naive users interact with databases using form interfaces with graphical interaction facilities
 - Web interfaces are the most common kind, but there are many others
 - Forms interfaces usually provide mechanisms to check for correctness of user input, and automatically fill in fields given key values
 - Most database vendors provide convenient mechanisms to create forms interfaces, and to link form actions to

Report Generators

- Report generators are tools to generate human-readable summary reports from a database
 - They integrate database querying with creation of formatted text and graphical charts
 - Reports can be defined once and executed periodically to get current information from the database.
 - Example of report (next page)
 - Microsoft's Object Linking and Embedding (OLE) provides a convenient way of

A Formatted Report

Acme Supply Company Inc. Quarterly Sales Report

Period: Jan. 1 to March 31, 2001

Region	Category	Sales	Subtotal
North	Computer Hardware	1,000,000	1,500,000
	Computer Software	500,000	
	All categories		
South	Computer Hardware	200,000	600,000
	Computer Software	400,000	
	All categories		
Total Sales			2,100,000

END OF CHAPTER



QBE Skeleton Tables for the Bank Example

<i>branch</i>	<i>branch-name</i>	<i>branch-city</i>	<i>assets</i>

<i>customer</i>	<i>customer-name</i>	<i>customer-street</i>	<i>customer-city</i>

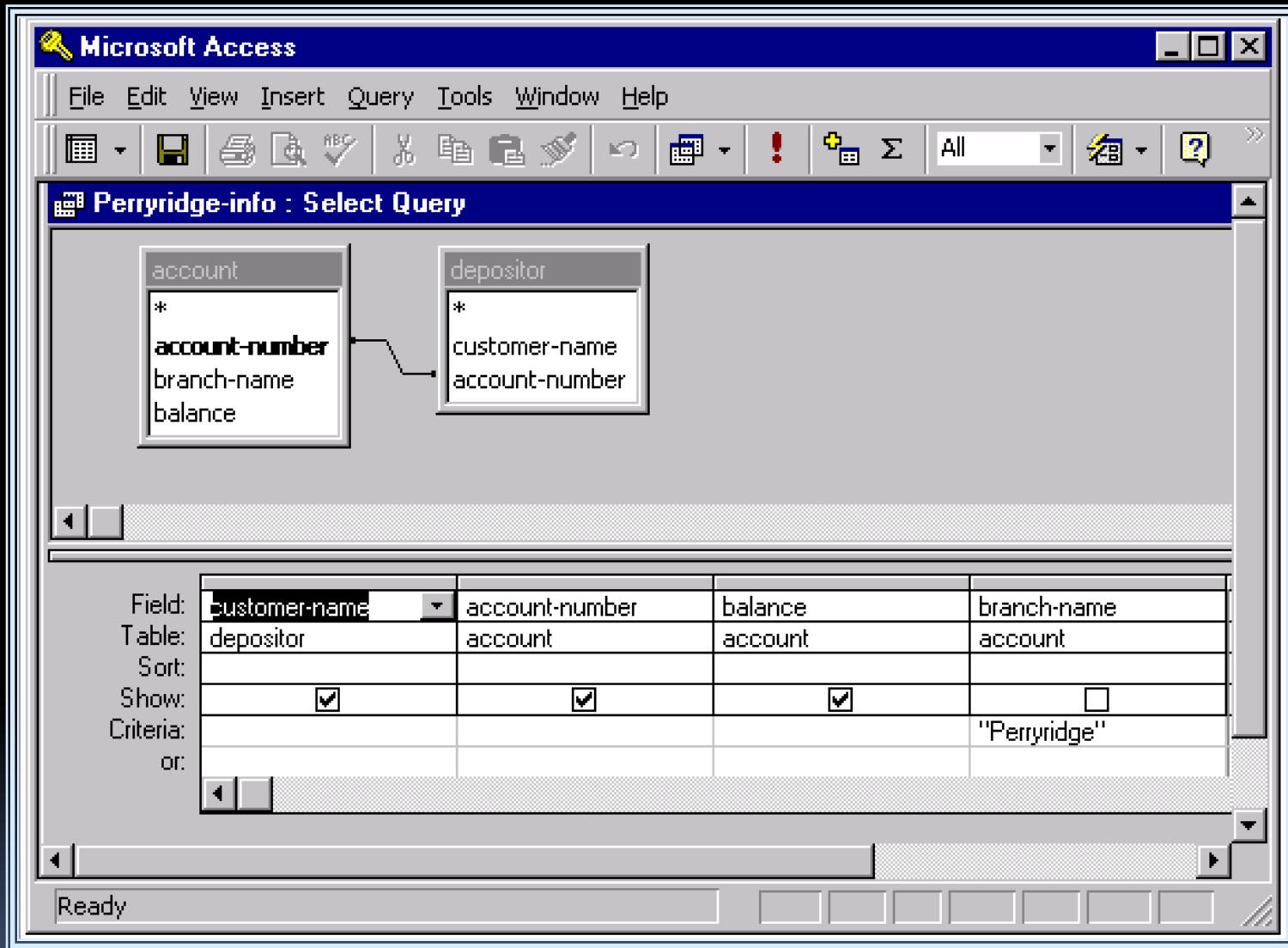
<i>loan</i>	<i>loan-number</i>	<i>branch-name</i>	<i>amount</i>

<i>borrower</i>	<i>customer-name</i>	<i>loan-number</i>

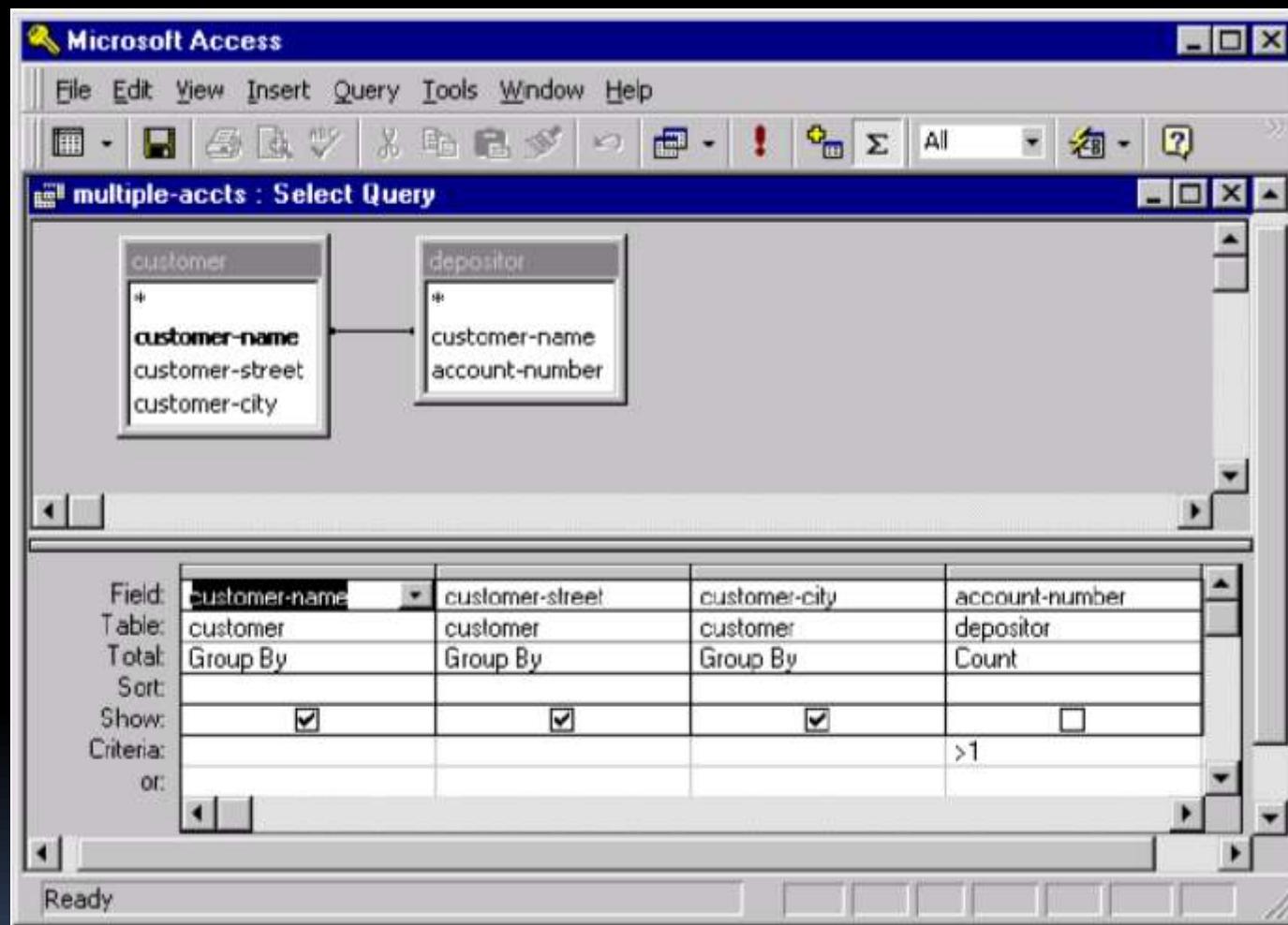
<i>account</i>	<i>account-number</i>	<i>branch-name</i>	<i>balance</i>

<i>depositor</i>	<i>customer-name</i>	<i>account-number</i>

An Example Query in Microsoft Access QBE



An Aggregation Query in Microsoft Access QBE



The *account* Relation

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

The v/l Relation

<i>account-number</i>	<i>balance</i>
A-201	900
A-217	750

Result of $\text{infer}(R, I)$

<i>account-number</i>	<i>balance</i>
A-201	900
A-217	750

<i>loan</i>	<i>loan-number</i>	<i>branch-name</i>	<i>amount</i>
	P.	Perryridge	

<i>loan</i>	<i>loan-number</i>	<i>branch-name</i>	<i>amount</i>
	$_x$	Perryridge	

<i>branch</i>	<i>branch-name</i>	<i>branch-city</i>	<i>assets</i>
I.	Capital	Queens	

conditions

$$-_y \geq 2 *-_z$$

Chapter 6: Integrity and Security

■ Domain Constraints

- Referential Integrity
- Assertions
- Triggers
- Security
- Authorization
- Authorization in SQL

Domain Constraints

- Integrity constraints guard against accidental damage to the database, by ensuring that authorized changes to the database do not result in a loss of data consistency.
- Domain constraints are the most elementary form of integrity constraint.
- They test values inserted in the database, and test queries to ensure that the comparisons make sense.
- New domains can be created from existing data types
 - E.g. `create domain Dollars numeric(12, 2)`
`create domain Pounds numeric(12,2)`
- We cannot assign or compare a value of type Dollars to a value of type Pounds.
 - However, we can convert type as below
`(cast r.A as Pounds)`
(Should also multiply by the dollar-to-pound conversion-rate)

Domain Constraints (Cont.)

- The check clause in SQL-92 permits domains to be restricted:
 - Use **check** clause to ensure that an hourly-wage domain allows only values greater than a specified value.

```
create domain hourly-wage numeric(5,2)
constraint value-test check(value > = 4.00)
```

 - The domain has a constraint that ensures that the hourly-wage is greater than 4.00
 - The clause **constraint *value-test*** is optional; useful to indicate which constraint an update violated.
- Can have complex conditions in domain check
 - ```
create domain AccountType char(10)
constraint account-type-test
check (value in ('Checking', 'Saving'))
```
  - ```
check (branch-name in (select branch-name from
branch))
```

Referential Integrity

- Ensures that a value that appears in one relation for a given set of attributes also appears for a certain set of attributes in another relation.
 - Example: If “Perryridge” is a branch name appearing in one of the tuples in the *account* relation, then there exists a tuple in the *branch* relation for branch “Perryridge”.
- Formal Definition
 - Let $r_1(R_1)$ and $r_2(R_2)$ be relations with primary keys K_1 and K_2 respectively.
 - The subset α of R_2 is a *foreign key* referencing K_1 in relation r_1 , if for every t_2 in r_2 there must be a tuple t_1 in r_1 such that $t_1[K_1] = t_2[\alpha]$.
 - Referential integrity constraint also called subset dependency since its can be written as
$$\Pi_\alpha(r_2) \subseteq \Pi_{K_1}(r_1)$$

Referential Integrity in the E-R Model

- Consider relationship set R between entity sets E_1 and E_2 . The relational schema for R includes the primary keys K_1 of E_1 and K_2 of E_2 .
Then K_1 and K_2 form foreign keys on the relational schemas for E_1 and E_2 respectively.
- Weak entity sets are also a source of referential integrity constraints.
 - For the relation schema for a weak entity set must include the primary key attributes of the entity set on which it depends

Checking Referential Integrity on Database Modification

The following tests must be made in order to preserve the following referential integrity constraint:

$$\Pi_{\alpha}(r_2) \subseteq \Pi_K(r_1)$$

- Insert. If a tuple t_2 is inserted into r_2 , the system must ensure that there is a tuple t_1 in r_1 such that $t_1[K] = t_2[\alpha]$. That is

$$t_2[\alpha] \in \Pi_K(r_1)$$

- Delete. If a tuple, t_1 is deleted from r_1 , the system must compute the set of tuples in r_2 that reference t_1 :

$$\sigma_{\alpha = t_1[K]}(r_2)$$

If this set is not empty

- either the delete command is rejected as an error, or
- the tuples that reference t_1 must themselves be deleted (cascading deletions are possible).

Database Modification (Cont.)

- Update. There are two cases:
 - If a tuple t_2 is updated in relation r_2 and the update modifies values for foreign key α , then a test similar to the insert case is made:
 - Let t_2' denote the new value of tuple t_2 . The system must ensure that
$$t_2'[\alpha] \in \Pi_K(r_1)$$
 - If a tuple t_1 is updated in r_1 , and the update modifies values for the primary key (K), then a test similar to the delete case is made:
 1. The system must compute
$$\sigma_{\alpha = t_1[K]}(r_2)$$
using the old value of t_1 (the value before the update is applied).
 2. If this set is not empty
 1. the update may be rejected as an error, or
 2. the update may be cascaded to the tuples in the set, or
 3. the tuples in the set may be deleted.

Referential Integrity in SQL

- Primary and candidate keys and foreign keys can be specified as part of the SQL create table statement:
 - The **primary key** clause lists attributes that comprise the primary key.
 - The **unique key** clause lists attributes that comprise a candidate key.
 - The **foreign key** clause lists the attributes that comprise the foreign key and the name of the relation referenced by the foreign key.
- By default, a foreign key references the primary key attributes of the referenced table
foreign key (*account-number*) references *account*
- Short form for specifying a single column as foreign key
account-number* char (10) references *account
- Reference columns in the referenced table can be explicitly specified
 - but must be declared as primary/candidate keys**foreign key (*account-number*) references *account*(*account-number*)**

Referential Integrity in SQL - Example

```
create table customer
  (customer-name char(20),
   customer-street char(30),
   customer-city    char(30),
   primary key (customer-name))

create table branch
  (branch-name    char(15),
   branch-city char(30),
   assets        integer,
   primary key (branch-name))
```

Referential Integrity in SQL - Example (Cont.)

```
create table account
  (account-number char(10),
   branch-name    char(15),
   balance        integer,
   primary key (account-number),
   foreign key (branch-name)
   references branch)
```

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

Cascading Actions in SQL

```
create table account
```

```
    . . .
```

```
        foreign key(branch-name) references  
        branch
```

```
            on delete cascade  
            on update cascade
```

```
    . . .)
```

- Due to the on delete cascade clauses, if a delete of a tuple in *branch* results in referential-integrity constraint violation, the delete “cascades” to the

Cascading Actions in SQL

- If there is a chain of foreign-key dependencies across multiple relations, with on delete cascade specified for each dependency, a deletion or update at one end of the chain can propagate across the entire chain.
- If a cascading update to delete causes a constraint violation that cannot be handled by a further cascading operation, the system aborts the transaction.
 - As a result, all the changes caused by the transaction and its cascading actions are

Referential Integrity in SQL (Cont.)

- Alternative to cascading:
 - **on delete set null**
 - **on delete set default**
- Null values in foreign key attributes complicate SQL referential integrity semantics, and are best prevented using **not null**
 - if any attribute of a foreign key is null, the tuple is defined to satisfy the foreign key constraint!

Assertions

- An *assertion* is a predicate expressing a condition that we wish the database always to satisfy.
- An assertion in SQL takes the form

```
create assertion <assertion-name>
check <predicate>
```
- When an assertion is made, the system tests it for validity, and tests it again on every update that may violate the assertion

Assertion Example

- The sum of all loan amounts for each branch must be less than the sum of all account balances at the branch.

create assertion *sum-constraint*
check

(not exists (select * from *branch*
 where (select
 sum(*amount*) from *loan*
 where
 loan.branch-name =

Assertion Example

- Every loan has at least one borrower who maintains an account with a minimum balance of \$1000.00

create assertion *balance-constraint* check

(not exists (

select * from *loan*

where not exists (

select *

from *borrower, depositor, account*

where *loan.loan-number* =

borrower.loan-number

Triggers

- A trigger is a statement that is executed automatically by the system as a side effect of a modification to the database.
- To design a trigger mechanism, we must:
 - Specify the conditions under which the trigger is to be executed.
 - Specify the actions to be taken when the trigger executes.
- Triggers introduced to SQL standard

Trigger Example

- Suppose that instead of allowing negative account balances, the bank deals with overdrafts by
 - setting the account balance to zero
 - creating a loan in the amount of the overdraft
 - giving this loan a loan number identical to the account number of the overdrawn account
- The condition for executing the trigger is an update to the *account*

Trigger Example in SQL:1999

```
create trigger overdraft-trigger after
update on account
referencing new row as nrow
for each row
when nrow.balance < 0
begin atomic
    insert into borrower
        (select customer-name, account-
number
         from depositor
         where nrow.account-number =
               depositor.account-
number);
```

Triggering Events and Actions

in SQL Triggering event can be insert, delete or update

- Triggers on update can be restricted to specific attributes
 - E.g. `create trigger overdraft-trigger after update of balance on account`
- Values of attributes before and after an update can be referenced
 - referencing old row as : for deletes and updates
 - referencing new row as : for inserts and updates

Statement Level Triggers

- Instead of executing a separate action for each affected row, a single action can be executed for all rows affected by a transaction
 - Use **for each statement** instead of **for each row**
 - Use **referencing old table** or **referencing new table** to refer to temporary tables (called *transition tables*) containing the affected rows
 - Can be more efficient when dealing with SQL statements that update a large

External World Actions

- We sometimes require external world actions to be triggered on a database update
 - E.g. re-ordering an item whose quantity in a warehouse has become small, or turning on an alarm light,
- Triggers cannot be used to directly implement external-world actions, BUT
 - Triggers can be used to record actions-to-be-taken in a separate table
 - Have an external process that repeatedly scans the table, carries out external-world actions and deletes action from table
- E.g. Suppose a warehouse has the

External World Actions (Cont.)

Create trigger *reorder-trigger* after update
of *amount* on *inventory*

referencing old row as *orow*, new row as
nrow

or each row

```
when nrow.level <= (select level  
from minlevel  
where minlevel.item  
= orow.item)
```

```
and orow.level > (select level  
from minlevel  
where
```

```
minlevel.item = orow.item)
```

begin

insert into *orders*

Triggers in MS-SQLServer

create trigger *overdraft-trigger* on
account

for update

as

if inserted.*balance* < 0

begin

insert into *borrower*

(select *customer-name, account-number*

from *depositor*, inserted

where inserted.*account-number*

=

depositor.account-number)

When Not To Use Triggers

Triggers were used earlier for tasks such as

- maintaining summary data (e.g. total salary of each department)
- Replicating databases by recording changes to special relations (called **change** or **delta** relations) and having a separate process that applies the changes over to a replica

There are better ways of doing these now:

- Databases today provide built in materialized view facilities to maintain summary data
- Databases provide built in support for

Security

- Security – protection from malicious attempts to steal or modify data.
 - Database system level
 - Authentication and authorization mechanisms to allow specific users access only to required data
 - We concentrate on authorization in the rest of this chapter
 - Operating system level
 - Operating system super-users can do anything they want to the database! Good operating system level security is required.
 - Network level: must use encryption to prevent

Security (Cont.)

- Physical level
 - Physical access to computers allows destruction of data by intruders; traditional lock-and-key security is needed
 - Computers must also be protected from floods, fire, etc.
 - More in Chapter 17 (Recovery)
- Human level
 - Users must be screened to ensure that authorized users do not give access to intruders
 - Users should be trained on password selection and secrecy

Authorization

Forms of authorization on parts of the database:

- Read authorization – allows reading, but not modification of data.
- Insert authorization – allows insertion of new data, but not modification of existing data.
- Update authorization – allows modification, but not deletion of data.
- Delete authorization – allows deletion of data.

Authorization (Cont.)

Forms of authorization to modify the database schema:

- Index authorization – allows creation and deletion of indices.
- Resources authorization – allows creation of new relations.
- Alteration authorization – allows addition or deletion of attributes in a relation.
- Drop authorization – allows deletion of relations.

Authorization and Views

- Users can be given authorization on views, without being given any authorization on the relations used in the view definition
- Ability of views to hide data serves both to simplify usage of the system and to enhance security by allowing users access only to data they need for their job
- A combination of relational-level security and view-level security can be used to limit a user's access to

View Example

- Suppose a bank clerk needs to know the names of the customers of each branch, but is not authorized to see specific loan information.
 - Approach: Deny direct access to the *loan* relation, but grant access to the view *cust-loan*, which consists only of the names of customers and the branches at which they have a loan.
 - The *cust-loan* view is defined in SQL as follows:

```
create view cust-loan as
select branchname, customer-
```

View Example (Cont.)

- The clerk is authorized to see the result of the query:

```
select *  
from cust-loan
```

- When the query processor translates the result into a query on the actual relations in the database, we obtain a query on *borrower* and *loan*.
- Authorization must be checked on the clerk's query before query processing replaces a view by the definition of the view.

Authorization on Views

- Creation of view does not require resources authorization since no real relation is being created
- The creator of a view gets only those privileges that provide no additional authorization beyond that he already had.
- E.g. if creator of view *cust-loan* had only read authorization on *borrower* and *loan*, he gets only read authorization on *cust-loan*

Granting of Privileges

- The passage of authorization from one user to another may be represented by an authorization graph.
- The nodes of this graph are the users.
- The root of the graph is the database administrator
- Consider graph for update authorization on loan.
- An edge $U_i \rightarrow U_j$ indicates that user U_i has granted update authorization on loan to U_j .

Authorization Grant Graph

- *Requirement:* All edges in an authorization graph must be part of some path originating with the database administrator
- If DBA revokes grant from U_1 :
 - Grant must be revoked from U_4 since U_1 no longer has authorization
 - Grant must not be revoked from U_5 since U_5 has another authorization path from DBA through U_2
- Must prevent cycles of grants with no path from the root:
 - DBA → U_1 , U_1 → U_2 , U_2 → U_3 , U_3 → U_4 , U_4 → U_5 , U_5 → U_1

Security Specification in SQL

- The grant statement is used to confer authorization

```
grant <privilege list>  
    on <relation name or view name>
```

```
to <user list>
```

- <user list> is:

- a user-id
- *public*, which allows all valid users the privilege granted
- A role (more on this later)

- Granting a privilege on a view does not imply granting any privileges on the

Privileges in SQL

- select: allows read access to relation, or the ability to query using the view
 - Example: grant users U_1 , U_2 , and U_3 **select** authorization on the *branch* relation:
 $\text{grant select on } branch \text{ to } U_1, U_2, U_3$
- insert: the ability to insert tuples
- update: the ability to update using the SQL update statement
- delete: the ability to delete tuples.
- references: ability to declare foreign keys when creating relations.
- usage: In SQL-92; authorizes a user to use a specified domain
- all privileges: used as a short form for all the allowable privileges

Privilege To Grant Privileges

- with grant option: allows a user who is granted a privilege to pass the privilege on to other users.

- Example:

grant select on *branch* to U_1 , with grant option
gives U_1 the **select** privileges on branch and
allows U_1 to grant this
privilege to others

Roles

- Roles permit common privileges for a class of users can be specified just once by creating a corresponding “role”
- Privileges can be granted to or revoked from roles, just like user
- Roles can be assigned to users, and even to other roles
- SQL:1999 supports roles
 - `create role teller`
 - `create role manager`

■ `grant select on branch to teller`

■ `grant update (balance) on account to teller`

Revoking Authorization in SQL

- The revoke statement is used to revoke authorization.

```
revoke<privilege list>
```

```
on <relation name or view name> from  
<user list> [restrict|cascade]
```

- Example:

```
revoke select on branch from  $U_1, U_2, U_3$   
cascade
```

- Revocation of a privilege from a user may cause other users also to lose that privilege; referred to as cascading of the revoke.

Revoking Authorization in SQL

- <privilege-list> may be all to revoke all privileges the revoker may hold.
- If <revoker-list> includes public all users lose the privilege except those granted it explicitly.
- If the same privilege was granted twice to the same user by different grantors, the user may retain the privilege after the revocation.
- All privileges that depend on the privilege being revoked are also revoked.

Limitations of SQL

- **Authorization**
 - E.g. we cannot restrict students to see only (the tuples storing) their own grades
- With the growth in Web access to databases, database accesses come primarily from application servers.
 - End users don't have database user ids, they are all mapped to the same database user id
- All end-users of an application (such as a web application) may be mapped to a single database user
- The task of authorization in above cases falls on the application program, with no support from SQL
 - Benefit: fine grained authorizations, such as to individual tuples, can be implemented by the application.
 - Drawback: Authorization must be done in application code, and may be dispersed all

Audit Trails

- An audit trail is a log of all changes (inserts/deletes/updates) to the database along with information such as which user performed the change, and when the change was performed.
- Used to track erroneous/fraudulent updates.
- Can be implemented using triggers, but many database systems provide direct support.

Encryption

- Data may be *encrypted* when database authorization provisions do not offer sufficient protection.
- Properties of good encryption technique:
 - Relatively simple for authorized users to encrypt and decrypt data.
 - Encryption scheme depends not on the secrecy of the algorithm but on the secrecy of a parameter of the algorithm called the encryption key.
 - Extremely difficult for an intruder to determine the encryption key.

Encryption (Cont.)

- *Data Encryption Standard* (DES) substitutes characters and rearranges their order on the basis of an encryption key which is provided to authorized users via a secure mechanism. Scheme is no more secure than the key transmission mechanism since the key has to be shared.
- Advanced Encryption Standard (AES) is a new standard replacing DES, and is based on the Rijndael algorithm, but is also dependent on shared secret keys
- *Public-key encryption* is based on each user having two keys:
 - *public key* – publicly published key used to encrypt data, but cannot be used to decrypt data
 - *private key* -- key known only to individual user, and used to decrypt data. Need not be transmitted to the site doing encryption.

Encryption scheme is such that it is impossible or extremely hard to decrypt data given only the public key.

Authentication

- Password based authentication is widely used, but is susceptible to sniffing on a network
- Challenge-response systems avoid transmission of passwords
 - DB sends a (randomly generated) challenge string to user
 - User encrypts string and returns result.
 - DB verifies identity by decrypting result
 - Can use public-key encryption system by DB sending a message encrypted using user's public key, and user decrypting and sending the message back

Digital Certificates

- Digital certificates are used to verify authenticity of public keys.
- Problem: when you communicate with a web site, how do you know if you are talking with the genuine web site or an imposter?
 - Solution: use the public key of the web site
 - Problem: how to verify if the public key itself is genuine?
- Solution:
 - Every client (e.g. browser) has public keys of a few root-level **certification authorities**
 - A site can get its name/URL and public key signed by a certification authority: signed

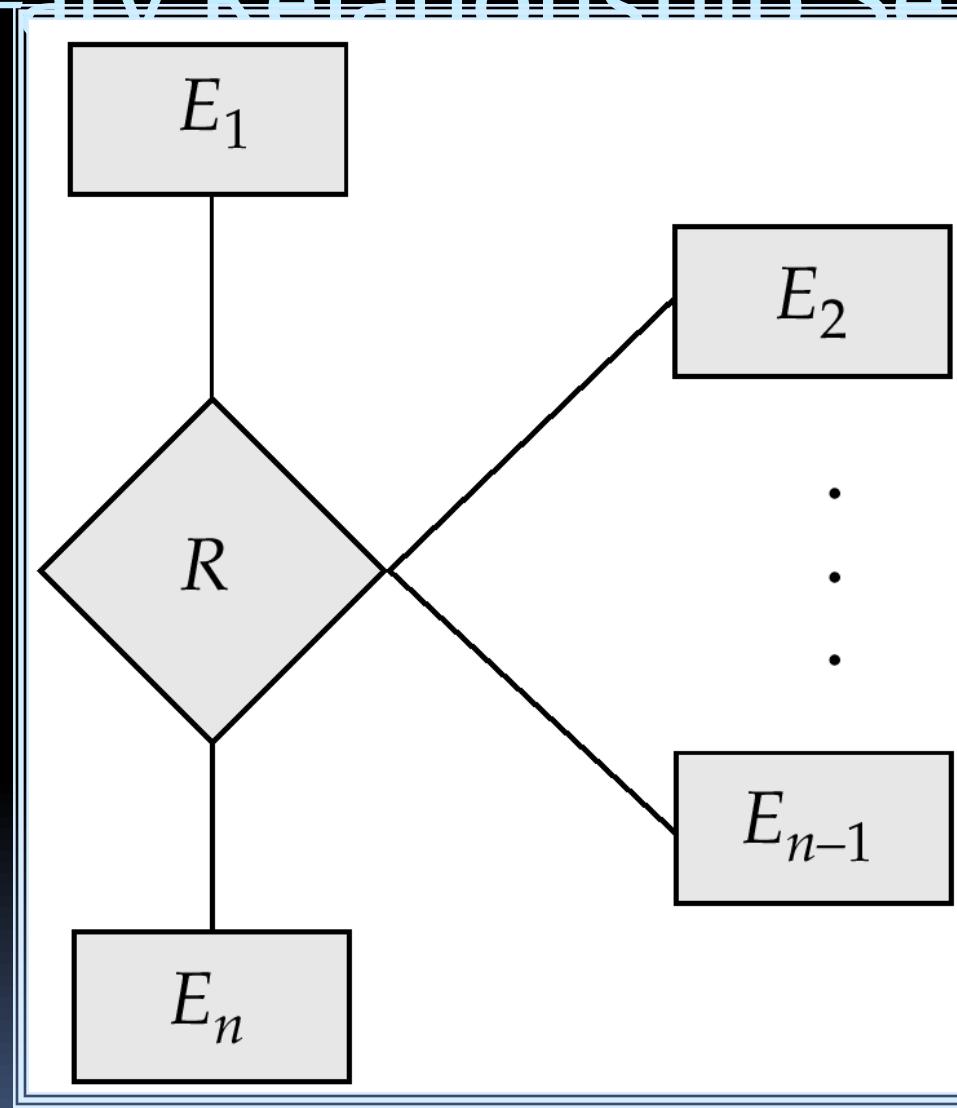
END OF CHAPTER



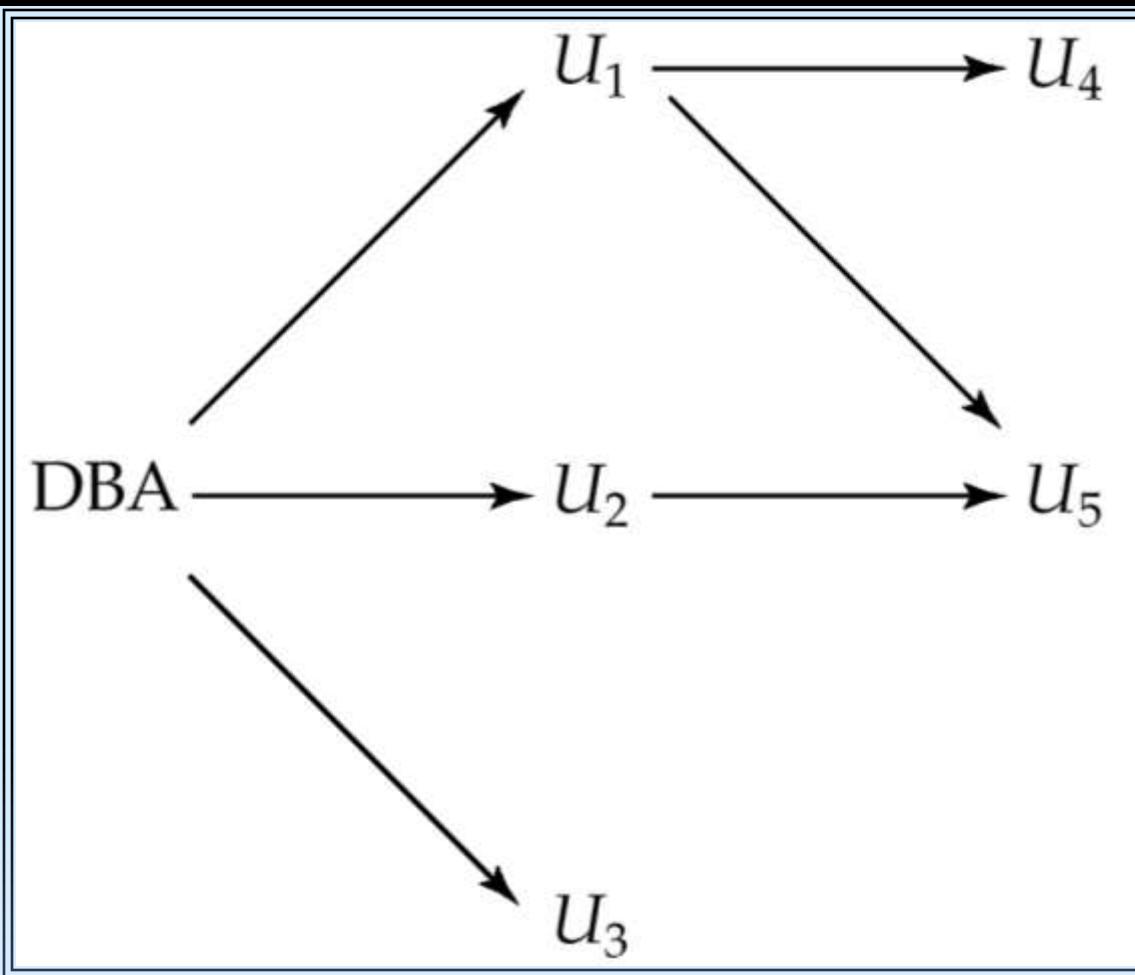
Statistical Databases

- Problem: how to ensure privacy of individuals while allowing use of data for statistical purposes (e.g., finding median income, average bank balance etc.)
- Solutions:
 - System rejects any query that involves fewer than some predetermined number of individuals.
 - * Still possible to use results of multiple overlapping queries to deduce data about an individual
 - *Data pollution* -- random falsification of

An n -ary Relationship Set

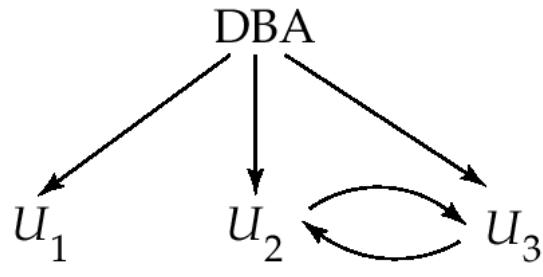


Authorization-Grant Graph

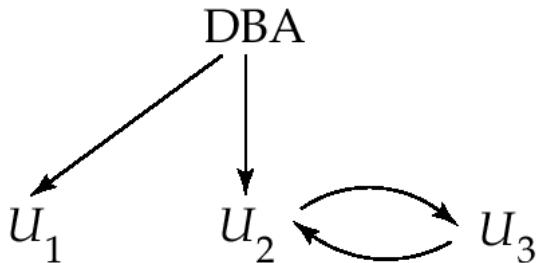


Attempt to Defeat Authorization

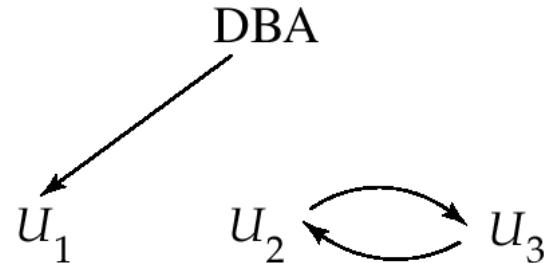
Devocation



(a)

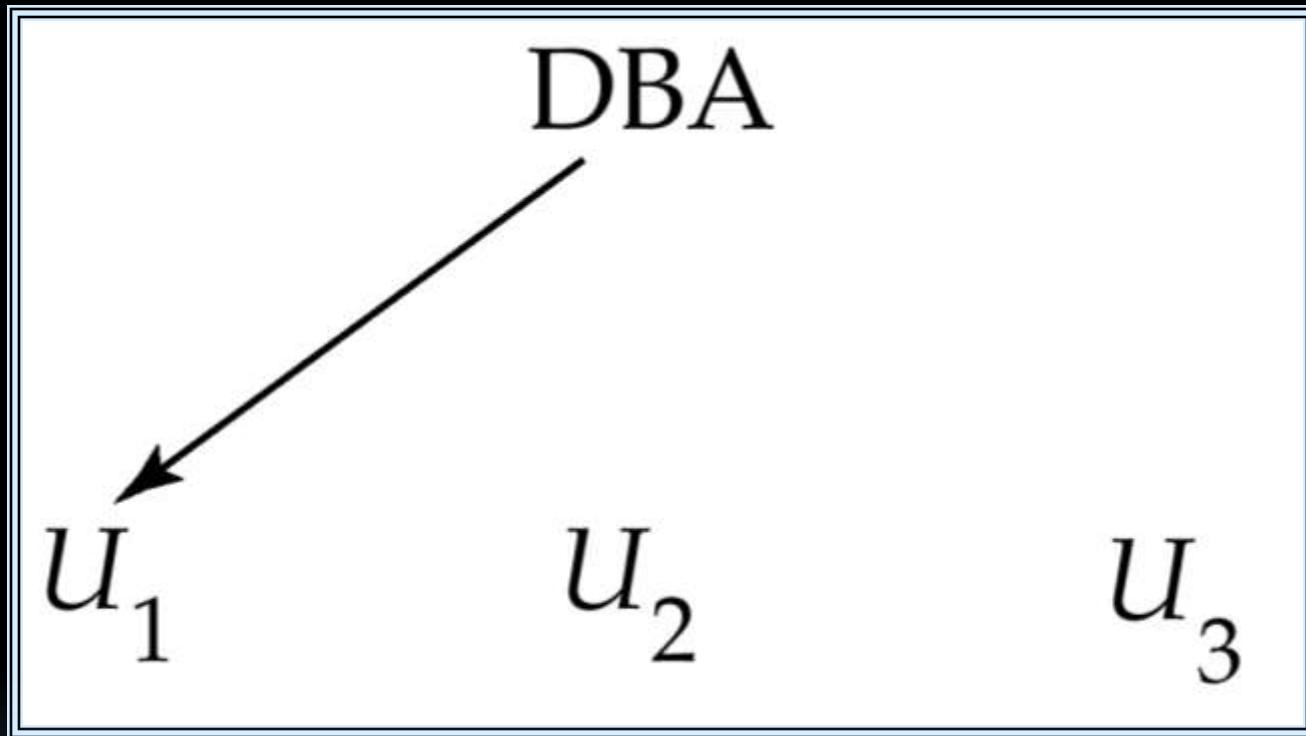


(b)



(c)

Authorization Graph



Physical Level Security

- Protection of equipment from floods, power failure, etc.
- Protection of disks from theft, erasure, physical damage, etc.
- Protection of network and terminal cables from wiretaps non-invasive electronic eavesdropping, physical damage, etc.

solutions:

- Replicated hardware:
 - mirrored disks, dual busses, etc.
 - multiple access paths between every pair of devices
- Physical security: locks, police, etc.
- Software techniques to detect physical security breaches.

Human Level Security

- Protection from stolen passwords, sabotage, etc.
- Primarily a management problem:
 - Frequent change of passwords
 - Use of “non-guessable” passwords
 - Log all invalid access attempts
 - Data audits
 - Careful hiring practices

Operating System Level Security

- Protection from invalid logins
- File-level access protection (often not very helpful for database security)
- Protection from improper use of “superuser” authority.
- Protection from improper use of privileged machine instructions.

Network-Level Security

- Each site must ensure that it communicate with trusted sites (not intruders).
- Links must be protected from theft or modification of messages
- Mechanisms:
 - Identification protocol (password-based),
 - Cryptography.

Database-Level Security

- Assume security at network, operating system, human, and physical levels.
- Database specific issues:
 - each user may have authority to read only part of the data and to write only part of the data.
 - User authority may correspond to entire files or relations, but it may also correspond only to parts of files or relations.
- Local autonomy suggests site-level authorization control in a distributed database.
- Global control suggests centralized control.

CHAPTER 7: RELATIONAL DATABASE DESIGN



Chapter 7: Relational Database Design

- First Normal Form
- Pitfalls in Relational Database Design
- Functional Dependencies
- Decomposition
- Boyce–Codd Normal Form
- Third Normal Form
- Multivalued Dependencies and Fourth Normal Form
- Overall Database Design Process

First Normal Form

- Domain is atomic if its elements are considered to be indivisible units
 - Examples of non-atomic domains:
 - Set of names, composite attributes
 - Identification numbers like CS101 that can be broken up into parts
- A relational schema R is in first normal form if the domains of all attributes of R are atomic
- Non-atomic values complicate storage and encourage redundant (repeated) storage of data
 - E.g. Set of accounts stored with each customer, and set of owners stored with each account
 - We assume all relations are in first normal form (revisit this in Chapter 9 on Object Relational Databases)

First Normal Form (Contd.)

- Atomicity is actually a property of how the elements of the domain are used.
 - E.g. Strings would normally be considered indivisible
 - Suppose that students are given roll numbers which are strings of the form *CS0012* or *EE1127*
 - If the first two characters are extracted to find the department, the domain of roll numbers is not atomic.
 - Doing so is a bad idea: leads to encoding of information in application program rather than in the database.

Pitfalls in Relational Database

Design

Relational database design requires that we find a “good” collection of relation schemas. A bad design may lead to

- Repetition of Information.
- Inability to represent certain information.
- Design Goals:
 - Avoid redundant data
 - Ensure that relationships among attributes are represented
 - Facilitate the checking of updates for violation of database integrity constraints.

Example

- Consider the relation schema:

*Lending-schema = (branch-name, branch-city, assets,
customer-name, loan-number, amount)*

<i>branch-name</i>	<i>branch-city</i>	<i>assets</i>	<i>customer-name</i>	<i>loan-number</i>	<i>amount</i>
Downtown	Brooklyn	9000000	Jones	L-17	1000
Redwood	Palo Alto	2100000	Smith	L-23	2000
Perryridge	Horseneck	1700000	Hayes	L-15	1500
Downtown	Brooklyn	9000000	Jackson	L-14	1500

- Data for *branch-name*, *branch-city*, *assets* are repeated for each loan that a branch makes
 - Wastes space
 - Complicates updating, introducing possibility of inconsistency of *assets* value
- Null values
 - Cannot store information about a branch if no loans exist
 - Can use null values, but they are difficult to handle.

Decomposition

- Decompose the relation schema *Lending-schema* into:
Branch-schema = (*branch-name*, *branch-city*, *assets*)
Loan-info-schema = (*customer-name*, *loan-number*,
 branch-name,
 amount)
- All attributes of an original schema (R) must appear in the decomposition (R_1, R_2):

$$R = R_1 \cup R_2$$

- Lossless-join decomposition.
For all possible relations r on schema R

$$r = \prod_{R_1}(r) \quad \prod_{R_2}(r)$$



Example of Non Lossless-Join Decomposition

- Decomposition of $R = (A, B)$

$$R_1 = (A) \quad R_2 = (B)$$

A	B
α	1
α	2
β	1

r

A
α
β

$\Pi_A(r)$

B
1
2

$\Pi_{B(r)}$

$\Pi_A(r) \bowtie \Pi_B(r)$

A	B
α	1
α	2
β	1
β	2

Goal — Devise a Theory for the Following

- Decide whether a particular relation R is in “good” form.
- In the case that a relation R is not in “good” form, decompose it into a set of relations $\{R_1, R_2, \dots, R_n\}$ such that
 - each relation is in good form
 - the decomposition is a lossless-join decomposition
- Our theory is based on:
 - functional dependencies
 - multivalued dependencies

Functional Dependencies

- Constraints on the set of legal relations.
- Require that the value for a certain set of attributes determines uniquely the value for another set of attributes.
- A functional dependency is a generalization of the notion of a *key*.

Functional Dependencies

- Let R be a relation schema
- (Cont.) $\alpha \subseteq R$ and $\beta \subseteq R$
- The functional dependency

$$\alpha \rightarrow \beta$$

holds on R if and only if for any legal relations $r(R)$, whenever any two tuples t_1 and t_2 of r agree on the attributes α , they also agree on the attributes β . That is,

$$t_1[\alpha] = t_2[\alpha] \Rightarrow t_1[\beta] = t_2[\beta]$$

- Example: Consider $r(A,B)$ with the following instance of r .

1	4
1	5
3	7

- On this instance, $A \rightarrow B$ does NOT hold, but $B \rightarrow A$ does hold.

Functional Dependencies (Cont.)

K is a superkey for relation schema R if and only if $K \rightarrow R$

K is a candidate key for R if and only if

- $K \rightarrow R$, and
- for no $\alpha \subset K$, $\alpha \rightarrow R$

Functional dependencies allow us to express constraints that cannot be expressed using superkeys. Consider the schema:

Loan-info-schema = (customer-name, loan-number, branch-name, amount).

Use of Functional Dependencies

- We use functional dependencies to:
 - test relations to see if they are legal under a given set of functional dependencies.
 - If a relation r is legal under a set F of functional dependencies, we say that r satisfies F .
 - specify constraints on the set of legal relations
 - We say that F holds on R if all legal relations on R satisfy the set of functional dependencies F .
- Note: A specific instance of a relation schema may satisfy a functional dependency even if the functional

Functional Dependencies (Cont.)

- A functional dependency is trivial if it is satisfied by all instances of a relation
 - *E.g.*
 - $\text{customer-name}, \text{loan-number} \rightarrow \text{customer-name}$
 - $\text{customer-name} \rightarrow \text{customer-name}$
 - In general, $\alpha \rightarrow \beta$ is trivial if $\beta \subseteq \alpha$

Closure of a Set of Functional

Given a set F of functional dependencies, there are certain other functional dependencies that are logically implied by F .

- E.g. If $A \rightarrow B$ and $B \rightarrow C$, then we can infer that $A \rightarrow C$
- The set of all functional dependencies logically implied by F is the *closure* of F .
- We denote the *closure* of F by F^+ .
- We can find all of F^+ by applying Armstrong's Axioms:
 - if $\beta \subseteq \alpha$, then $\alpha \rightarrow \beta$

Example

- $R = \{A, B, C, G, H, I\}$

$$\begin{aligned}F = \{ & A \rightarrow B \\& A \rightarrow C \\& CG \rightarrow H \\& CG \rightarrow I \\& B \rightarrow H \}\end{aligned}$$

- some members of F^+

- $A \rightarrow H$
 - by transitivity from $A \rightarrow B$ and $B \rightarrow H$
- $AG \rightarrow I$
 - by augmenting $A \rightarrow C$ with G , to get $AG \rightarrow CG$ and then transitivity with $CG \rightarrow I$
- $CG \rightarrow HI$
 - from $CG \rightarrow H$ and $CG \rightarrow I$: “union rule” can be inferred from $\{A, B, C, G, H, I\}$

Procedure for Computing F^+

- To compute the closure of a set of functional dependencies F :

$$F^+ = F$$

repeat

 for each functional dependency f in

$$F^+$$

 apply reflexivity and
 augmentation rules on f

 add the resulting functional
 dependencies to F^+

 for each pair of functional

 dependencies f and g in F

Closure of Functional Dependencies (Cont.)

- We can further simplify manual computation of F^+ by using the following additional rules.
 - If $\alpha \rightarrow \beta$ holds and $\alpha \rightarrow \gamma$ holds, then $\alpha \rightarrow \beta\gamma$ holds (**union**)
 - If $\alpha \rightarrow \beta\gamma$ holds, then $\alpha \rightarrow \beta$ holds and $\alpha \rightarrow \gamma$ holds (**decomposition**)
 - If $\alpha \rightarrow \beta$ holds and $\beta \rightarrow \delta$ holds, then $\alpha\beta \rightarrow \delta$ holds (**pseudotransitivity**)

The above rules can be inferred from Armstrong's axioms.

Closure of Attribute Sets

- Given a set of attributes α , define the *closure* of α under F (denoted by α^+) as the set of attributes that are functionally determined by α under F :

$$\alpha \rightarrow \beta \text{ is in } F^+ \quad \square \quad \beta \subseteq \alpha^+$$

- Algorithm to compute α^+ , the closure of α under F

result := α ;

while (changes to *result*) do

for each $\beta \rightarrow \gamma$ in F do

begin

Example of Attribute Set

$R = \{A, B, C, G, H, I\}$

$F = \{A \rightarrow B, A \rightarrow C, CG \rightarrow H, CG \rightarrow I, B \rightarrow H\}$

- $(AG)^+$
 1. $result = AG$
 2. $result = ABCG$ ($A \rightarrow C$ and $A \rightarrow B$)
 3. $result = ABCGH$ ($CG \rightarrow H$ and $CG \subseteq AGBC$)
 4. $result = ABCGHI$ ($CG \rightarrow I$ and $CG \subseteq AGBCH$)
- Is AG a candidate key?
 1. Is AG a super key?
 1. Does $AG \rightarrow R? == \text{Is } (AG)^+ \supseteq R$
 2. Is any subset of AG a superkey?
 1. Does $A \rightarrow R? == \text{Is } (A)^+ \supseteq R$
 2. Does $G \rightarrow R? == \text{Is } (G)^+ \supseteq R$

Uses of Attribute Closure

There are several uses of the attribute closure algorithm:

- Testing for superkey:
 - To test if α is a superkey, we compute α^+ , and check if α^+ contains all attributes of R .
- Testing functional dependencies
 - To check if a functional dependency $\alpha \rightarrow \beta$ holds (or, in other words, is in F^+), just check if $\beta \subseteq \alpha^+$.
 - That is, we compute α^+ by using attribute closure, and then check if it contains β .
 - Is a simple and cheap test, and very useful

Canonical Cover

- Sets of functional dependencies may have redundant dependencies that can be inferred from the others
 - Eg: $A \rightarrow C$ is redundant in: $\{A \rightarrow B, B \rightarrow C, A \rightarrow C\}$
 - Parts of a functional dependency may be redundant
 - E.g. on RHS: $\{A \rightarrow B, B \rightarrow C, A \rightarrow CD\}$ can be simplified to
$$\{A \rightarrow B, B \rightarrow C, A \rightarrow D\}$$
 - E.g. on LHS: $\{A \rightarrow B, B \rightarrow C, AC \rightarrow D\}$ can be simplified to

Extraneous Attributes

- Consider a set F of functional dependencies and the functional dependency $\alpha \rightarrow \beta$ in F .
 - Attribute A is extraneous in α if $A \in \alpha$ and F logically implies $(F - \{\alpha \rightarrow \beta\}) \cup \{(\alpha - A) \rightarrow \beta\}$.
 - Attribute A is extraneous in β if $A \in \beta$ and the set of functional dependencies $(F - \{\alpha \rightarrow \beta\}) \cup \{\alpha \rightarrow (\beta - A)\}$ logically implies F .
- *Note:* implication in the opposite direction is trivial in each of the cases above, since a “stronger” functional dependency always implies a weaker one

Testing if an Attribute is Extraneous

- Consider a set F of functional dependencies and the functional dependency $\alpha \rightarrow \beta$ in F .
- To test if attribute $A \in \alpha$ is extraneous in α
 1. compute $(\{\alpha\} - A)^+$ using the dependencies in F
 2. check that $(\{\alpha\} - A)^+$ contains A ; if it does, A is extraneous
- To test if attribute $A \in \beta$ is extraneous in β

Canonical Cover

- A *canonical cover* for F is a set of dependencies F_c such that
 - F logically implies all dependencies in F_c , and
 - F_c logically implies all dependencies in F , and
 - No functional dependency in F_c contains an extraneous attribute, and
 - Each left side of functional dependency in F_c is unique.

- To compute a canonical cover for F : repeat

 Use the union rule to replace any dependencies in F

$\alpha_1 \rightarrow \beta_1$ and $\alpha_1 \rightarrow \beta_2$ with $\alpha_1 \rightarrow \beta_1, \beta_2$

Example of Computing a Canonical Cover

$$\begin{aligned} R &= \{A, B, C\} \\ F &= \{A \rightarrow BC \\ &\quad B \rightarrow C \\ &\quad A \rightarrow B \\ &\quad AB \rightarrow C\} \end{aligned}$$

- Combine $A \rightarrow BC$ and $A \rightarrow B$ into $A \rightarrow BC$
 - Set is now $\{A \rightarrow BC, B \rightarrow C, AB \rightarrow C\}$
- A is extraneous in $AB \rightarrow C$
 - Check if the result of deleting A from $AB \rightarrow C$ is implied by the other dependencies
 - Yes: in fact, $B \rightarrow C$ is already present!
 - Set is now $\{A \rightarrow BC, B \rightarrow C\}$
- C is extraneous in $A \rightarrow BC$
 - Check if $A \rightarrow C$ is logically implied by $A \rightarrow B$ and the other dependencies
 - Yes: using transitivity on $A \rightarrow B$ and $B \rightarrow C$.
 - Can use attribute closure of A in more complex cases
- The canonical cover is:
 - $A \rightarrow B$
 - $B \rightarrow C$

Goals of Normalization

- Decide whether a particular relation R is in “good” form.
- In the case that a relation R is not in “good” form, decompose it into a set of relations $\{R_1, R_2, \dots, R_n\}$ such that
 - each relation is in good form
 - the decomposition is a lossless-join decomposition
- Our theory is based on:
 - functional dependencies
 - multivalued dependencies

Decomposition

- Decompose the relation schema *Lending-schema* into:

Branch-schema = (*branch-name*, *branch-city*, *assets*)

Loan-info-schema = (*customer-name*, *loan-number*,
branch-name, *amount*)

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

$$R = R_1 \cup R_2$$

- Lossless-join decomposition.

For all possible relations r on schema R

$$r = \Pi_{R1}(r) \ \ \ \Pi_{R2}(r)$$

- A decomposition of R into R_1 and R_2 is lossless join if and only if at least one of the following dependencies is in F^+ :

- $R_1 \cap R_2 \rightarrow R_1$
- $R_1 \cap R_2 \rightarrow R_2$

Example of Lossy-Join

Decomposition

- Lossy-Join decompositions result in information loss.
- Example: Decomposition of $R = (A, \overline{B})$

	A	B
α	1	
α	2	
β	1	

r

$R_1 (A)$

R_1	A	
	α	
	β	

$\Pi_A(r)$

$\Pi_B(r) = (B)$

$\Pi_B(r)$	B	
	1	
	2	

$\Pi_A(r) \bowtie \Pi_B(r)$

	A	B
α	1	
α	2	
β	1	
β	2	

Normalization Using Functional Dependencies

When we decompose a relation schema R with a set of functional dependencies F into R_1, R_2, \dots, R_n we want

- Lossless-join decomposition: Otherwise decomposition would result in information loss.
- No redundancy: The relations R_i preferably should be in either Boyce-Codd Normal Form or Third Normal Form.
- Dependency preservation: Let F_i be the set of dependencies F^+ that include only attributes in R_i .
 - Preferably the decomposition should be dependency preserving, that is, $(F_1 \cup F_2 \cup \dots \cup F_n)^+ = F^+$
 - Otherwise, checking updates for violation of

Example

- $R = (A, B, C)$
 $F = \{A \rightarrow B, B \rightarrow C\}$
 - Can be decomposed in two different ways
- $R_1 = (A, B), R_2 = (B, C)$
 - Lossless-join decomposition:
 $R_1 \cap R_2 = \{B\}$ and $B \rightarrow BC$
 - Dependency preserving
- $R_1 = (A, B), R_2 = (A, C)$ ✘
 - Lossless-join decomposition:
 $R_1 \cap R_2 = \{A\}$ and $A \rightarrow AB$
 - Not dependency preserving

Testing for Dependency

- To ~~check if a dependency $\alpha \rightarrow \beta$ is preserved~~ in a decomposition of R into R_1, R_2, \dots, R_n we apply the following simplified test (with attribute closure done w.r.t. F)
 - $result = \alpha$
while (changes to $result$) do
 for each R_i in the decomposition
 $t = (result \cap R_i)^+ \cap R_i$
 $result = result \cup t$

- $result$ contains all attributes in β , then the functional dependency $\alpha \rightarrow \beta$ is preserved.

Boyce-Codd Normal Form

A relation schema R is in BCNF with respect to a set F of functional dependencies if for all functional dependencies in F^+ of the form $\alpha \rightarrow \beta$, where $\alpha \subseteq R$ and $\beta \subseteq R$, at least one of the following holds:

- $\alpha \rightarrow \beta$ is trivial (i.e., $\beta \subseteq \alpha$)
- α is a superkey for R

Example

- $R = (A, B, C)$
 $F = \{A \rightarrow B$
 $B \rightarrow C\}$
Key = {A}
- R is not in BCNF
- Decomposition $R_1 = (A, B), R_2 = (B, C)$
 - R_1 and R_2 in BCNF
 - Lossless-join decomposition
 - Dependency preserving

Testing for BCNF

- To check if a non-trivial dependency
 $\cdot \alpha \rightarrow \beta$ causes a violation of BCNF
 1. compute α^+ (the attribute closure of α), and
 2. verify that it includes all attributes of R , that is, it is a superkey of R .
- Simplified test: To check if a relation schema R is in BCNF, it suffices to check only the dependencies in the given set F for violation of BCNF, rather than checking all dependencies in F^+ .
 - If none of the dependencies in F causes a violation of BCNF, then none of the dependencies in F^+ will cause a violation of BCNF either.

BCNF Decomposition Algorithm

```
result := {R};  
done := false;  
compute  $F^+$ ;  
while (not done) do
```

```
    if (there is a schema  $R_i$  in result that is not in BCNF)
```

```
        then begin
```

```
            let  $\alpha \rightarrow \beta$  be a nontrivial functional  
            dependency that holds on  $R_i$   
            such that  $\alpha \rightarrow R_i$  is not in  $F^+$ ,  
            and  $\alpha \cap \beta = \emptyset$ ;
```

```
            result := (result -  $R_i$ )  $\cup$  ( $R_i - \beta$ )  $\cup$  ( $\alpha, \beta$ );
```

```
        end
```

```
    else done := true;
```

Note: each R_i is in BCNF, and decomposition is lossless-join.

Example of BCNF

- $R = (\text{branch-name}, \text{branch-city}, \text{assets}, \text{customer-name}, \text{loan-number}, \text{amount})$
 $F = \{\text{branch-name} \rightarrow \text{assets} \text{ branch-city}$
 $\text{loan-number} \rightarrow \text{amount} \text{ branch-name}\}$
Key = {loan-number, customer-name}
- Decomposition
 - $R_1 = (\text{branch-name}, \text{branch-city}, \text{assets})$
 - $R_2 = (\text{branch-name}, \text{customer-name}, \text{loan-number}, \text{amount})$
 - $R_3 = (\text{branch-name}, \text{loan-number}, \text{amount})$

Testing Decomposition for

- **BCNF**: Check if a relation R_i in a decomposition of R is in BCNF,
 - Either test R_i for BCNF with respect to the restriction of F to R_i (that is, all FDs in F^+ that contain only attributes from R_i)
 - or use the original set of dependencies F that hold on R , but with the following test:
 - for every set of attributes $\alpha \subseteq R_i$, check that α^+ (the attribute closure of α) either includes no attribute of $R_i - \alpha$, or includes all attributes of R_i .
 - If the condition is violated by some $\alpha \rightarrow \beta$ in F , the dependency
$$\alpha \rightarrow (\alpha^+ - \alpha) \cap R_i$$

BCNF and Dependency

Preservation

It is not always possible to get a BCNF decomposition that is dependency preserving

- $R = J, K, L$
 $F = \{JK \rightarrow L$
 $L \rightarrow K\}$

Two candidate keys = JK and JL

- R is not in BCNF
- Any decomposition of R will fail to preserve

$$JK \rightarrow L$$

Third Normal Form: Motivation

- There are some situations where
 - BCNF is not dependency preserving, and
 - efficient checking for FD violation on updates is important
- Solution: define a weaker normal form, called Third Normal Form.
 - Allows some redundancy (with resultant problems; we will see examples later)
 - But FDs can be checked on individual relations without computing a join.
 - There is always a lossless-join, dependency-preserving decomposition into 3NF.

Third Normal Form

- A relation schema R is in third normal form (3NF) if for all:

$$\alpha \rightarrow \beta \text{ in } F^+$$

at least one of the following holds:

- $\alpha \rightarrow \beta$ is trivial (i.e., $\beta \in \alpha$)
- α is a superkey for R
- Each attribute A in $\beta - \alpha$ is contained in a candidate key for R .

(*NOTE*: each attribute may be in a different candidate key)

- If a relation is in BCNF it is in 3NF

3NF (Cont.)

- Example
 - $R = \{J, K, L\}$
 $F = \{JK \rightarrow L, L \rightarrow K\}$
 - Two candidate keys: JK and JL
 - R is in 3NF
 - $JK \rightarrow L$ JK is a superkey
 - $L \rightarrow K$ K is contained in a candidate key
- BCNF decomposition has (JL) and (LK)
 - Testing for $JK \rightarrow L$ requires a join
- There is some redundancy in this schema
- Equivalent to example in book:
 - Banker-schema = (branch-name, customer-name, banker-name)
 - banker-name \rightarrow branch name
 - branch name customer-name \rightarrow banker-name

Testing for 3NF

- Optimization: Need to check only FDs in F , need not check all FDs in F^+ .
- Use attribute closure to check for each dependency $\alpha \rightarrow \beta$, if α is a superkey.
- If α is not a superkey, we have to verify if each attribute in β is contained in a candidate key of R
 - this test is rather more expensive, since it involve finding candidate keys
 - testing for 3NF has been shown to be NP-hard

3NF Decomposition Algorithm

Let F_c be a canonical cover for F ;

$i := 0$;

for each functional dependency $\alpha \rightarrow \beta$
in F_c do

if none of the schemas R_j , $1 \leq j \leq i$
contains $\alpha \beta$

then begin

$i := i + 1$;

$R_i := \alpha \beta$

end

if none of the schemas R_j , $1 \leq j \leq i$
contains a candidate key for R
then begin

3NF Decomposition Algorithm (Cont.)

- Above algorithm ensures:
 - each relation schema R_i is in 3NF
 - decomposition is dependency preserving and lossless-join
 - Proof of correctness is at end of this file ([click here](#))

Example

- Relation schema:
 $\text{Banker-info-schema} = (\text{branch-name}, \text{customer-name}, \text{banker-name}, \text{office-number})$
- The functional dependencies for this relation schema are:
 $\text{banker-name} \rightarrow \text{branch-name}$
 $\text{office-number} \rightarrow \text{customer-name}$
 $\text{branch-name} \rightarrow \text{banker-name}$
- The key is:

Applying 3NF to *Banker-info-schema*

- The for loop in the algorithm causes us to include the following schemas in our decomposition:

Banker-office-schema =
(banker-name, branch-name,
 office-number)

Banker-schema = *(customer-*
name, branch-name,
 banker-name)

- Since *Banker-schema* contains a candidate key for

Comparison of BCNF and 3NF

- It is always possible to decompose a relation into relations in 3NF and
 - the decomposition is lossless
 - the dependencies are preserved
- It is always possible to decompose a relation into relations in BCNF and
 - the decomposition is lossless
 - it may not be possible to preserve dependencies.

Comparison of BCNF and 3NF (Cont.)

- Example of problems due to redundancy in 3NF

	J	L	K
$R =$	j_1	l_1	k_1
$F = \{$	j_2	l_1	k_1
	j_3	l_1	k_1
	<i>null</i>	l_2	k_2

- A schema that is in 3NF but not in BCNF has the problems of
- repetition of information (e.g., the relationship l_1, k_1)
 - need to use null values (e.g., to represent the relationship l_2, k_2 where there is no corresponding value for J).

Design Goals

- Goal for a relational database design is:
 - BCNF.
 - Lossless join.
 - Dependency preservation.
- If we cannot achieve this, we accept one of
 - Lack of dependency preservation
 - Redundancy due to use of 3NF
- Interestingly, SQL does not provide a direct way of specifying functional

Testing for FDs Across

- If decomposition is not dependency preserving, we can have an extra materialized view for each dependency $\alpha \rightarrow \beta$ in F_c that is not preserved in the decomposition
- Relations
- The materialized view is defined as a projection on $\alpha \beta$ of the join of the relations in the decomposition
- Many newer database systems support materialized views and database system maintains the view when the relations are updated.
 - No extra coding effort for programmer.
- The functional dependency $\alpha \rightarrow \beta$ is expressed by declaring α as a candidate key on the materialized view.
- Checking for candidate key cheaper than checking $\alpha \rightarrow \beta$
- BUT:
 - Space overhead: for storing the materialized view
 - Time overhead: Need to keep materialized view up to date when relations are updated
 - Database system may not support key declarations on materialized views

Multivalued Dependencies

- There are database schemas in BCNF that do not seem to be sufficiently normalized
- Consider a database $\text{classes}(\text{course}, \text{teacher}, \text{book})$ such that $(c,t,b) \in \text{classes}$ means that t is qualified to teach c , and b is a required textbook for c
- The database is supposed to list for each course the set of teachers any one of which can be the course's

Multivalued Dependencies

(Cont.)

<i>course</i>	<i>teacher</i>	<i>book</i>
database	Avi	DB Concepts
database	Avi	Ullman
database	Hank	DB Concepts
database	Hank	Ullman
database	Sudarshan	DB Concepts
database	Sudarshan	Ullman
operating systems	Avi	OS Concepts
operating systems	Avi	Shaw
operating systems	Jim	OS Concepts
operating systems	Jim	Shaw

classes

- There are no non-trivial functional dependencies and therefore the relation is in BCNF
- Insertion anomalies – i.e., if Sara is a new teacher that can teach database, operating systems, and computer organization, then we have to insert three rows in the relation.

Multivalued Dependencies

- Therefore, it is better to decompose *classes* into:

course	teacher
database	Avi
database	Hank
database	Sudarshan
operating systems	Avi
operating systems	Jim

teaches

course	book
database	DB Concepts
database	Ullman
operating systems	OS Concepts
operating systems	Shaw

text

We shall see that these two relations are in Fourth Normal Form (4NF)

Multivalued Dependencies

(MVDs) be a relation schema and let
 $\alpha \subseteq R$ and $\beta \subseteq R$. The
multivalued dependency

$$\alpha \twoheadrightarrow \beta$$

holds on R if in any legal relation $r(R)$, for all pairs for tuples t_1 and t_2 in r such that $t_1[\alpha] = t_2[\alpha]$, there exist tuples t_3 and t_4 in r such that:

$$t_1[\alpha] = t_2[\alpha] = t_3[\alpha] = t_4[\alpha] \quad t_3[\beta] = t_1[\beta] \quad \twoheadrightarrow$$

MVD (Cont.)

- Tabular representation of $\alpha \rightarrow \beta$

	α	β	$R - \alpha - \beta$
t_1	$a_1 \dots a_i$	$a_{i+1} \dots a_j$	$a_{j+1} \dots a_n$
t_2	$a_1 \dots a_i$	$b_{i+1} \dots b_j$	$b_{j+1} \dots b_n$
t_3	$a_1 \dots a_i$	$a_{i+1} \dots a_j$	$b_{j+1} \dots b_n$
t_4	$a_1 \dots a_i$	$b_{i+1} \dots b_j$	$a_{j+1} \dots a_n$

Example

- Let R be a relation schema with a set of attributes that are partitioned into 3 nonempty subsets.

Y, Z, W

- We say that $Y \rightarrow\!\!\! \rightarrow Z$ (Y multidetermines Z) if and only if for all possible relations $r(R)$

$< y_1, z_1, w_1 > \in r$ and $< y_2, z_2, w_2 > \in r$

Example (Cont.)

- In our example:

course $\rightarrow\!\!\!$ teacher

course $\rightarrow\!\!\!$ book

- The above formal definition is supposed to formalize the notion that given a particular value of Y (*course*) it has associated with it a set of values of Z (*teacher*) and a set of values of W (*book*), and these two sets are in some sense independent of each other.
- Note:

Use of Multivalued Dependencies

- We use multivalued dependencies in two ways:
 1. To test relations to determine whether they are legal under a given set of functional and multivalued dependencies
 2. To specify constraints on the set of legal relations. We shall thus concern ourselves *only* with relations that satisfy a given set of functional and multivalued dependencies.
- If a relation r fails to satisfy a given

Theory of MVDs

- From the definition of multivalued dependency, we can derive the following rule:
 - If $\alpha \rightarrow \beta$, then $\alpha \twoheadrightarrow \beta$

That is, every functional dependency is also a multivalued dependency

- The closure D^+ of D is the set of all functional and multivalued dependencies logically implied by D .
 - We can compute D^+ from D , using \twoheadrightarrow the formal definitions of functional

Fourth Normal Form

- A relation schema R is in 4NF with respect to a set D of functional and multivalued dependencies if for all multivalued dependencies in D^+ of the form $\alpha \twoheadrightarrow \beta$, where $\alpha \subseteq R$ and $\beta \subseteq R$, at least one of the following hold:
 - $\alpha \twoheadrightarrow \beta$ is trivial (i.e., $\beta \subseteq \alpha$ or $\alpha \cup \beta = R$)
 - α is a superkey for schema R
- If a relation is in 4NF it is in BCNF

Restriction of Multivalued Dependencies

- The restriction of D to R_i is the set D_i consisting of
 - All functional dependencies in D^+ that include only attributes of R_i
 - All multivalued dependencies of the form
$$\alpha \twoheadrightarrow (\beta \cap R_i)$$
where $\alpha \subseteq R_i$ and $\alpha \twoheadrightarrow \beta$ is in D^+

4NF Decomposition Algorithm

result := {R};

done := false;

compute D⁺;

Let D_i denote the restriction of D^+ to R_i

while (not *done*)

 if (there is a schema R_i in *result* that
 is not in 4NF) then

 begin

 let $\alpha \twoheadrightarrow \beta$ be a nontrivial multivalued
 dependency that holds

 on R_i such that $\alpha \rightarrow R_i$ is not in
 D_i , and $\alpha \cap \beta = \emptyset$;



Example

$F = \{ A \rightarrow\!\!\!\rightarrow B$

$B \rightarrow\!\!\!\rightarrow HI$

$CG \rightarrow\!\!\!\rightarrow H \}$

- R is not in 4NF since $A \rightarrow\!\!\!\rightarrow B$ and A is not a superkey for R
- Decomposition
 - a) $R_1 = (A, B)$ $(R_1$ is in 4NF)
 - b) $R_2 = (A, C, G, H, I)$ $(R_2$ is not in 4NF)
 - c) $R_3 = (C, G, H)$ $(R_3$ is in 4NF)
 - d) $R_4 = (A, C, G, I)$ $(R_4$ is not in 4NF)
- Since $A \rightarrow\!\!\!\rightarrow B$ and $B \rightarrow\!\!\!\rightarrow HI$, $A \rightarrow\!\!\!\rightarrow HI$, $A \rightarrow\!\!\!\rightarrow I$
 - e) $R_5 = (A, I)$ $(R_5$ is in 4NF)
 - f) $R_6 = (A, C, G)$ $(R_6$ is in 4NF)

Further Normal Forms

- Join dependencies generalize multivalued dependencies
 - lead to **project-join normal form (PJNF)** (also called **fifth normal form**)
- A class of even more general constraints, leads to a normal form called **domain-key normal form**.
- Problem with these generalized constraints: are hard to reason with, and no set of sound and complete set of inference rules exists.

Overall Database Design

- We have assumed schema R is given
Process
 - R could have been generated when converting E-R diagram to a set of tables.
 - R could have been a single relation containing *all* attributes that are of interest (called **universal relation**).
 - Normalization breaks R into smaller relations.
 - R could have been the result of some ad hoc design of relations, which we then test/convert to normal form.

ER Model and Normalization

- When an E-R diagram is carefully designed, identifying all entities correctly, the tables generated from the E-R diagram should not need further normalization.
- However, in a real (imperfect) design there can be FDs from non-key attributes of an entity to other attributes of the entity
 - E.g. *employee* entity with attributes *department-number* and *department-address*, and an FD *department-number*

Universal Relation Approach

- Dangling tuples – Tuples that “disappear” in computing a join.
 - Let $r_1 (R_1)$, $r_2 (R_2)$, ..., $r_n (R_n)$ be a set of relations
 - A tuple r of the relation r_i is a dangling tuple if r is not in the relation:
$$\prod_{R_i} (r_1 \quad r_2 \quad \dots \quad r_n)$$

- The relation $r_1 \quad r_2 \quad \dots \quad r_n$ is called a *universal relation* since it involves all the attributes in the “universe” defined by

$$R_1 \cup R_2 \cup \dots \cup R_n$$

Universal Relation Approach

- Dangling tuples may occur in practical database applications.
- They represent incomplete information
- E.g. may want to break up information about loans into:
 - (branch-name, loan-number)
 - (loan-number, amount)
 - (loan-number, customer-name)
- Universal relation would require null values and have dangling tuples

Universal Relation Approach

- (Contd.) A particular decomposition defines a restricted form of incomplete information that is acceptable in our database.
 - Above decomposition requires at least one of customer-name, branch-name or amount in order to enter a loan number without using null values
 - Rules out storing of customer-name, amount without an appropriate loan-number (since it is a key, it can't be null either!)
- Universal relation requires unique

Denormalization for

- May want to use non-normalized schema for performance
- E.g. displaying *customer-name* along with *account-number* and *balance* requires join of *account* with *depositor*
- Alternative 1: Use denormalized relation containing attributes of *account* as well as *depositor* with all above attributes
 - faster lookup
 - Extra space and extra execution time for updates

Other Design Issues

- Some aspects of database design are not caught by normalization
- Examples of bad database design, to be avoided:
Instead of *earnings(company-id, year, amount)*, use
 - *earnings-2000, earnings-2001, earnings-2002*, etc., all on the schema (*company-id, earnings*).
 - Above are in BCNF, but make querying across years difficult and needs new table each year

PROOF OF CORRECTNESS OF 3NF DECOMPOSITION ALGORITHM



Correctness of 3NF Decomposition Algorithm

- 3NF decomposition algorithm is dependency preserving (since there is a relation for every FD in F_c)
- Decomposition is lossless join
 - A candidate key (C) is in one of the relations R_i in decomposition
 - Closure of candidate key under F_c must contain all attributes in R .
 - Follow the steps of attribute closure algorithm to show there is only one tuple in the join result for each tuple in R ,

Correctness of 3NF Decomposition Algorithm (Contd.)

Claim: if a relation R_i is in the decomposition generated by the above algorithm, then R_i satisfies 3NF.

- Let R_i be generated from the dependency $\alpha \rightarrow \beta$
- Let $\gamma \rightarrow B$ be any non-trivial functional dependency on R_i . (We need only consider FDs whose right-hand side is a single attribute.)
- Now, B can be in either β or α but not in both. Consider each case separately

Correctness of 3NF Decomposition (Contd.)

- Case 1: If B in β :
 - If γ is a superkey, the 2nd condition of 3NF is satisfied
 - Otherwise α must contain some attribute not in γ
 - Since $\gamma \rightarrow B$ is in F^+ it must be derivable from F_c , by using attribute closure on γ .
 - Attribute closure not have used $\alpha \rightarrow \beta$ – if it had been used, α must be contained in the attribute closure of γ , which is not possible, since we assumed γ is not a superkey.
 - Now using $\alpha \rightarrow \beta$ ($\beta \subseteq \{B\}$) and $\gamma \rightarrow B$ we can

Correctness of 3NF

Decomposition (Contd.)

- Case 2: B is in α .
 - Since α is a candidate key, the third alternative in the definition of 3NF is trivially satisfied.
 - In fact, we cannot show that γ is a superkey.
 - This shows exactly why the third alternative is present in the definition of 3NF.

Q.E.D.

END OF CHAPTER



Sample *lending* Relation

<i>branch-name</i>	<i>branch-city</i>	<i>assets</i>	<i>customer-name</i>	<i>loan-number</i>	<i>amount</i>
Downtown	Brooklyn	9000000	Jones	L-17	1000
Redwood	Palo Alto	2100000	Smith	L-23	2000
Perryridge	Horseneck	1700000	Hayes	L-15	1500
Downtown	Brooklyn	9000000	Jackson	L-14	1500
Mianus	Horseneck	400000	Jones	L-93	500
Round Hill	Horseneck	8000000	Turner	L-11	900
Pownal	Bennington	300000	Williams	L-29	1200
North Town	Rye	3700000	Hayes	L-16	1300
Downtown	Brooklyn	9000000	Johnson	L-18	2000
Perryridge	Horseneck	1700000	Glenn	L-25	2500
Brighton	Brooklyn	7100000	Brooks	L-10	2200

Sample Relation r

A	B	C	D
a_1	b_1	c_1	d_1
a_1	b_2	c_1	d_2
a_2	b_2	c_2	d_2
a_2	b_2	c_2	d_3
a_3	b_3	c_2	d_4

The *customer* Relation

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

The *loan* Relation

<i>loan-number</i>	<i>branch-name</i>	<i>amount</i>
L-17	Downtown	1000
L-23	Redwood	2000
L-15	Perryridge	1500
L-14	Downtown	1500
L-93	Mianus	500
L-11	Round Hill	900
L-29	Pownal	1200
L-16	North Town	1300
L-18	Downtown	2000
L-25	Perryridge	2500
L-10	Brighton	2200

The *branch* Relation

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

The Relation *branch-customer*

<i>branch-name</i>	<i>branch-city</i>	<i>assets</i>	<i>customer-name</i>
Downtown	Brooklyn	9000000	Jones
Redwood	Palo Alto	2100000	Smith
Perryridge	Horseneck	1700000	Hayes
Downtown	Brooklyn	9000000	Jackson
Mianus	Horseneck	400000	Jones
Round Hill	Horseneck	8000000	Turner
Pownal	Bennington	300000	Williams
North Town	Rye	3700000	Hayes
Downtown	Brooklyn	9000000	Johnson
Perryridge	Horseneck	1700000	Glenn
Brighton	Brooklyn	7100000	Brooks

The Relation *customer-loan*

<i>customer-name</i>	<i>loan-number</i>	<i>amount</i>
Jones	L-17	1000
Smith	L-23	2000
Hayes	L-15	1500
Jackson	L-14	1500
Jones	L-93	500
Turner	L-11	900
Williams	L-29	1200
Hayes	L-16	1300
Johnson	L-18	2000
Glenn	L-25	2500
Brooks	L-10	2200



The Relation *branch-customer* *customer-loan*

<i>branch-name</i>	<i>branch-city</i>	<i>assets</i>	<i>customer-name</i>	<i>loan-number</i>	<i>amount</i>
Downtown	Brooklyn	9000000	Jones	L-17	1000
Downtown	Brooklyn	9000000	Jones	L-93	500
Redwood	Palo Alto	2100000	Smith	L-23	2000
Perryridge	Horseneck	1700000	Hayes	L-15	1500
Perryridge	Horseneck	1700000	Hayes	L-16	1300
Downtown	Brooklyn	9000000	Jackson	L-14	1500
Mianus	Horseneck	400000	Jones	L-17	1000
Mianus	Horseneck	400000	Jones	L-93	500
Round Hill	Horseneck	8000000	Turner	L-11	900
Pownal	Bennington	300000	Williams	L-29	1200
North Town	Rye	3700000	Hayes	L-15	1500
North Town	Rye	3700000	Hayes	L-16	1300
Downtown	Brooklyn	9000000	Johnson	L-18	2000
Perryridge	Horseneck	1700000	Glenn	L-25	2500
Brighton	Brooklyn	7100000	Brooks	L-10	2200

An Instance of *Banker-schema*

<i>customer-name</i>	<i>banker-name</i>	<i>branch-name</i>
Jones	Johnson	Perryridge
Smith	Johnson	Perryridge
Hayes	Johnson	Perryridge
Jackson	Johnson	Perryridge
Curry	Johnson	Perryridge
Turner	Johnson	Perryridge

Tabular Representation of

$\alpha \rightarrow\rightarrow \beta$

	α	β	$R - \alpha - \beta$
t_1	$a_1 \dots a_i$	$a_{i+1} \dots a_j$	$a_{j+1} \dots a_n$
t_2	$a_1 \dots a_i$	$b_{i+1} \dots b_j$	$b_{j+1} \dots b_n$
t_3	$a_1 \dots a_i$	$a_{i+1} \dots a_j$	$b_{j+1} \dots b_n$
t_4	$a_1 \dots a_i$	$b_{i+1} \dots b_j$	$a_{j+1} \dots a_n$

Relation *bc*: An Example of Reduncy in a BCNF Relation

<i>loan-number</i>	<i>customer-name</i>	<i>customer-street</i>	<i>customer-city</i>
L-23	Smith	North	Rye
L-23	Smith	Main	Manchester
L-93	Curry	Lake	Horseneck

An Illegal bc Relation

<i>loan-number</i>	<i>customer-name</i>	<i>customer-street</i>	<i>customer-city</i>
L-23	Smith	North	Rye
L-27	Smith	Main	Manchester

Decomposition of *loan-info*

<i>branch-name</i>	<i>loan-number</i>
Round Hill	L-58
<i>loan-number</i>	<i>amount</i>
<i>loan-number</i>	<i>customer-name</i>
L-58	Johnson

Relationship of Exercise 7.4

A	B	C
a_1	b_1	c_1
a_1	b_1	c_2
a_2	b_1	c_1
a_2	b_1	c_3

Chapter 8: Object-Oriented Databases

■ Need for Complex Data Types

- The Object-Oriented Data Model
- Object-Oriented Languages
- Persistent Programming Languages
- Persistent C++ Systems

Need for Complex Data Types

- Traditional database applications in data processing had conceptually simple data types
 - Relatively few data types, first normal form holds
- Complex data types have grown more important in recent years
 - E.g. Addresses can be viewed as a
 - Single string, or
 - Separate attributes for each part, or
 - Composite attributes (which are not in first normal form)
 - E.g. it is often convenient to store multivalued attributes as-is, without creating a separate relation to store the values in first normal form
- Applications
 - computer-aided design, computer-aided software engineering
 - multimedia and image databases, and document/hypertext databases.

Object-Oriented Data Model

- Loosely speaking, an object corresponds to an entity in the E-R model.
- The *object-oriented paradigm* is based on *encapsulating* code and data related to an object into single unit.
- The object-oriented data model is a logical data model (like the E-R model).
- Adaptation of the object-oriented programming paradigm (e.g., Smalltalk, C++) to database systems.

Object Structure

- An object has associated with it:
 - A set of **variables** that contain the data for the object. The value of each variable is itself an object.
 - A set of **messages** to which the object responds; each message may have zero, one, or more *parameters*.
 - A set of **methods**, each of which is a body of code to implement a message; a method returns a value as the *response* to the message
- The physical representation of data is visible only to the **implementor** of the object
- Messages and responses provide the only external interface to an object.
- The term **message** does not necessarily imply **physical message passing**. Messages can be implemented as procedure invocations.

Messages and Methods

- Methods are programs written in general-purpose language with the following features
 - only variables in the object itself may be referenced directly
 - data in other objects are referenced only by sending *messages*.
- Methods can be read-only or update methods
 - Read-only methods do not change the value of the object
- Strictly speaking, every attribute of an entity must be represented by a variable and two methods, one to read and the other to update the attribute
 - e.g., the attribute *address* is represented by a variable *address* and two messages *get-address* and *set-address*.
 - For convenience, many object-oriented data models permit direct access to variables of other objects.

Object Classes

- Similar objects are grouped into a class; each such object is called an instance of its class
- All objects in a class have the same
 - Variables, with the same types
 - message interface
 - methodsThe may differ in the values assigned to variables
- Example: Group objects for people into a *person* class
- Classes are analogous to entity sets in the E-R model

Class Definition Example

```
class employee{  
    /*Variables */  
    string name;  
    string address;  
    date start-date;  
    int salary;  
    /* Messages */  
    int annual-salary();  
    string get-name();  
    string get-address();  
    int set-address(string new-address);  
    int employment-length();  
};
```

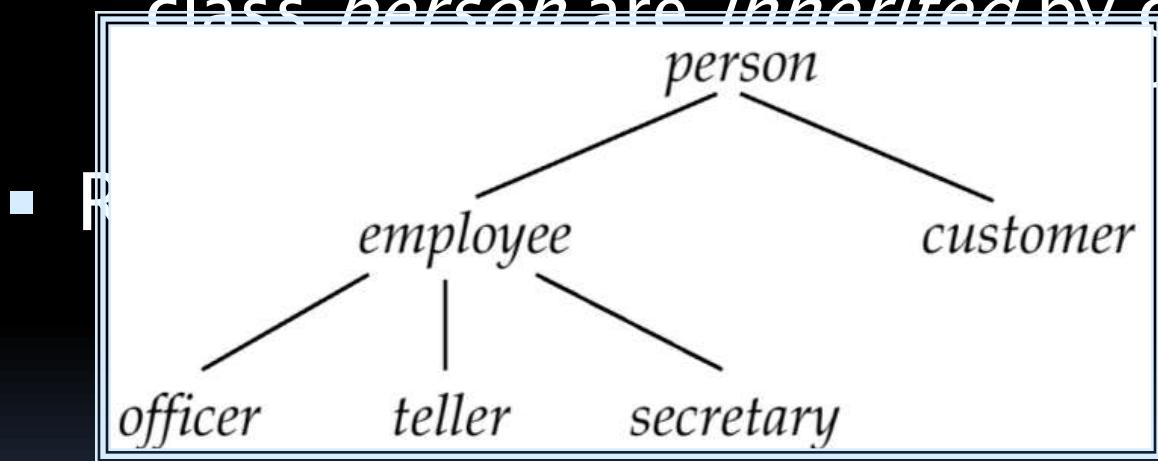
- Methods to read and set the other variables are also needed with strict encapsulation
- Methods are defined separately
 - E.g. `int employment-length() { return today() - start-date;}`
`int set-address(string new-address) { address = new-address;}`

Inheritance

- E.g., class of bank customers is similar to class of bank employees, although there are differences
 - both share some variables and messages, e.g., *name* and *address*.
 - But there are variables and messages specific to each class e.g., *salary* for employees and *credit-rating* for customers.
- Every employee is a person; thus *employee* is a specialization of *person*
- Similarly, *customer* is a specialization of *person*.
- Create classes *person*, *employee* and *customer*
 - variables/messages applicable to all persons associated with class *person*.
 - variables/messages specific to employees associated with class *employee*; similarly for *customer*

Inheritance (Cont.)

- Place classes into a specialization/IS-A hierarchy
 - variables/messages belonging to class *person* are *inherited* by class



Note analogy with ISA Hierarchy in the E-R model

Class Hierarchy Definition

```
class person {
    string    name;
    string    address;
};

class customer isa person {
    int credit-rating;
};

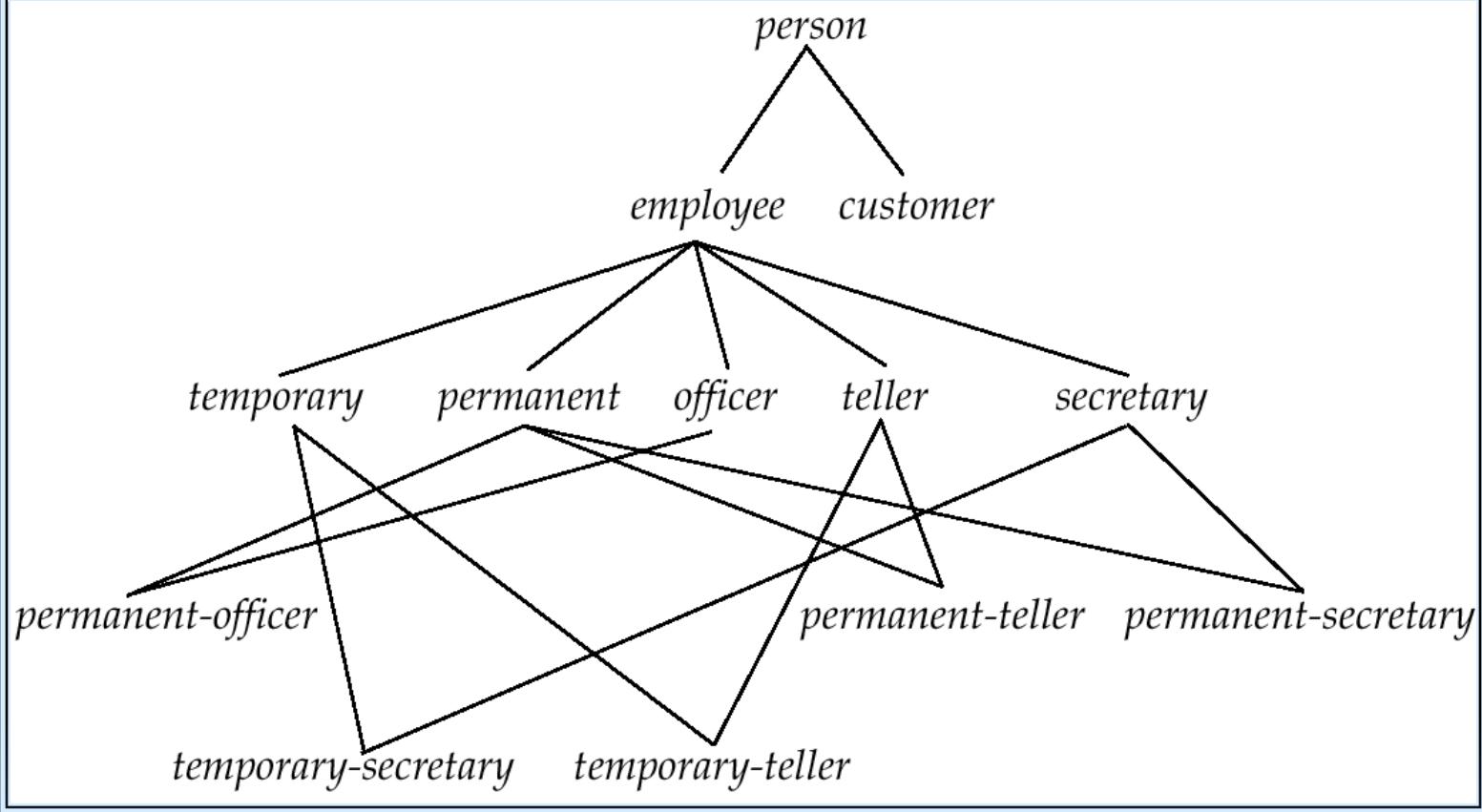
class employee isa person {
    date start-date;
    int salary;
};

class officer isa employee {
```

Class Hierarchy Example (Cont.)

- Full variable list for objects in the class *officer*:
 - *office-number, expense-account-number*: defined locally
 - *start-date, salary*: inherited from *employee*
 - *name, address*: inherited from *person*
- Methods inherited similar to variables.
- Substitutability — any method of a class, say *person*, can be invoked equally well with any object belonging to any subclass, such as subclass *officer* of *person*.
- Class extent: set of all objects in the class. Two options:
 1. Class extent of *employee* includes all *officer, teller* and *secretary* objects.
 2. Class extent of *employee* includes only

Example of Multiple Inheritance



Class DAG for banking example.

Multiple Inheritance

- With multiple inheritance a class may have more than one superclass.
 - The class/subclass relationship is represented by a **directed acyclic graph (DAG)**
 - Particularly useful when objects can be classified in more than one way, which are independent of each other
 - E.g. temporary/permanent is independent of Officer/secretary/teller
 - Create a subclass for each combination of subclasses
 - Need not create subclasses for combinations that are not possible in the database being modeled
 - A class inherits variables and methods from all its superclasses
 - There is potential for ambiguity when a variable/message N with

More Examples of Multiple

- **Inheritance** Conceptually, an object can belong to each of several subclasses
 - A *person* can play the roles of *student*, a *teacher* or *footballPlayer*, or any combination of the three
 - E.g., student teaching assistant who also play football
- Can use multiple inheritance to model “roles” of an object
 - That is, allow an object to take on any one or more of a set of types
- But many systems insist an object should have a most-specific class

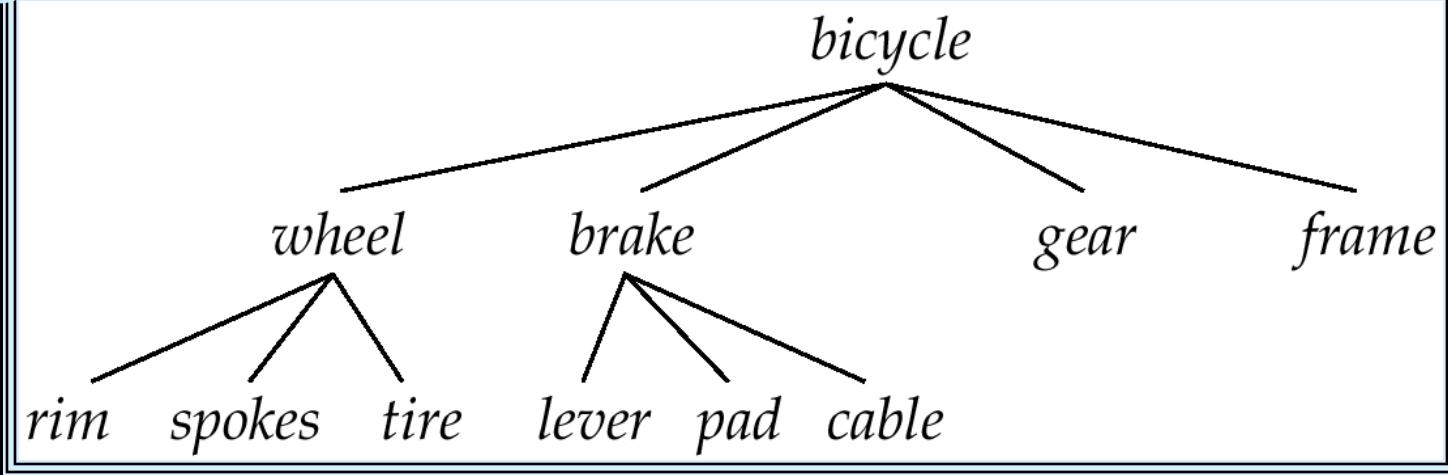
Object Identity

- An object retains its identity even if some or all of the values of variables or definitions of methods change over time.
- Object identity is a stronger notion of identity than in programming languages or data models not based on object orientation.
 - Value - data value; e.g. primary key value used in relational systems.
 - Name - supplied by user; used for variables in procedures.
 - Built-in - identity built into data model

Object Identifiers

- Object identifiers used to uniquely identify objects
 - Object identifiers are unique:
 - no two objects have the same identifier
 - each object has only one object identifier
 - E.g., the *spouse* field of a *person* object may be an identifier of another *person* object.
 - can be stored as a field of an object, to refer to another object.
 - Can be
 - system generated (created by database) or
 - external (such as social-security number)

Object Containment



- Each component in a design may contain other components
- Can be modeled as containment of objects. Objects containing other objects are called composite objects.
- Multiple levels of containment create a containment hierarchy

Object-Oriented Languages

- Object-oriented concepts can be used in different ways
 - Object-orientation can be used as a design tool, and be encoded into, for example, a relational database
 - ▢ analogous to modeling data with E-R diagram and then converting to a set of relations)
 - The concepts of object orientation can be incorporated into a programming language that is used to manipulate the database.
 - **Object-relational systems** – add complex types and object-orientation

Persistent Programming

- Persistent Programming languages allow objects to be created and stored in a database, and used directly from a programming language
 - allow data to be manipulated directly from the programming language
 - No need to go through SQL.
 - No need for explicit format (type) changes
 - format changes are carried out transparently by system
 - Without a persistent programming language, format changes becomes a burden on the programmer
 - More code to be written
 - More chance of bugs

Persistent Prog. Languages (Cont.)

- Drawbacks of persistent programming languages
 - Due to power of most programming languages, it is easy to make programming errors that damage the database.
 - Complexity of languages makes automatic high-level optimization more difficult.
 - Do not support declarative querying as well as relational databases

Persistence of Objects

- Approaches to make transient objects persistent include establishing
 - Persistence by Class – declare all objects of a class to be persistent; simple but inflexible.
 - Persistence by Creation – extend the syntax for creating objects to specify that that an object is persistent.
 - Persistence by Marking – an object that is to persist beyond program execution is marked as persistent before program termination.

Object Identity and Pointers

- A persistent object is assigned a persistent object identifier.
- Degrees of permanence of identity:
 - Intraprocedure – identity persists only during the executions of a single procedure
 - Intraprogram – identity persists only during execution of a single program or query.
 - Interprogram – identity persists from one program execution to another, but may change if the storage organization is changed

Object Identity and Pointers

(Cont.) In O-O languages such as C++, an object identifier is actually an in-memory pointer.

- Persistent pointer – persists beyond program execution
 - can be thought of as a pointer into the database
 - E.g. specify file identifier and offset into the file
 - Problems due to database reorganization have to be dealt with by keeping forwarding pointers

Storage and Access of Persistent Objects

How to find objects in the database:

- Name objects (as you would name files)
 - Cannot scale to large number of objects.
 - Typically given only to class extents and other collections of objects, but not objects.
- Expose object identifiers or persistent pointers to the objects
 - Can be stored externally.
 - All objects have object identifiers.
- Store collections of objects, and allow programs to iterate over the

Persistent C++ Systems

- C++ language allows support for persistence to be added without changing the language
 - Declare a class called Persistent_Object with attributes and methods to support persistence
 - Overloading – ability to redefine standard function names and operators (i.e., +, -, the pointer deference operator ->) when applied to new types
 - Template classes help to build a type-safe type system supporting collections and persistent types.
- Providing persistence without extending the C++ language is

ODMG C++ Object Definition

The Object Database Management Group is an industry consortium aimed at standardizing object-oriented databases

- in particular persistent programming languages
- Includes standards for C++, Smalltalk and Java
- ODMG-93
- ODMG-2.0 and 3.0 (which is 2.0 plus extensions to Java)
 - Our description based on ODMG-2.0
- ODMG C++ standard avoids changes

ODMG Types

- Template class `d_Ref<class>` used to specify references (persistent pointers)
- Template class `d_Set<class>` used to define sets of objects.
 - Methods include `insert_element(e)` and `delete_element(e)`
- Other collection classes such as `d_Bag` (set with duplicates allowed), `d_List` and `d_Varray` (variable length array) also provided.

ODMG C++ ODL: Example

```
class Branch : public d_Object {  
    ....  
}  
  
class Person : public d_Object {  
public:  
    d_String    name;      // should not use String!  
    d_String    address;  
};  
  
class Account : public d_Object {  
private:  
    d_Long     balance;  
public:  
    d_Long     number;  
    d_Set <d_Ref<Customer>> owners;  
    int       find_balance();  
    int       update_balance(int delta);  
};
```

ODMG C++ ODL: Example

(Cont.)

```
class Customer : public Person {  
public:  
    d_Date           member_from;  
    d_Long          customer_id;  
    d_Ref<Branch> home_branch;  
    d_Set <d_Ref<Account>> accounts; };
```

Implementing Relationships

- Relationships between classes implemented by references
- Special reference types enforces integrity by adding/removing inverse links.
 - Type `d_Rel_Ref<Class, InvRef>` is a reference to Class, where attribute `InvRef` of Class is the inverse reference.
 - Similarly, `d_Rel_Set<Class, InvRef>` is used for a set of references
- Assignment method (=) of class `d_Rel_Ref` is overloaded

Implementing Relationships

- E.g.

```
extern const char _owners[ ], _accounts[  
];
```

```
class Account : public d.Object {
```

```
....
```

```
    d_Rel_Set <Customer, _accounts>  
owners;  
}
```

```
// .. Since strings can't be used in  
templates ...
```

```
const char _owners= "owners";  
const char _accounts= "accounts";
```

ODMG C++ Object Manipulation

~~Language~~ persistent versions of C++ operators such as new(db)

```
d_Ref<Account> account = new(bank_db,  
"Account") Account;
```

- new allocates the object in the specified database, rather than in memory.
- The second argument ("Account") gives typename used in the database.
- Dereference operator -> when applied on a d_Ref<Account> reference loads the referenced object in memory (if not already present) before continuing with usual C++ dereference.
- Constructor for a class - a special

ODMG C++ OML: Database and Object Functions

- Class **d_Database** provides methods to
 - open a database:
open(databaseName)
 - give names to objects:
set_object_name(object, name)
 - look up objects by name:
lookup_object(name)
 - rename objects:
rename_object(oldname, newname)
 - close a database (**close()**);
- Class **d_Object** is inherited by all persistent classes

ODMG C++ OML: Example

```
int create_account_owner(String name, String  
Address){  
    Database bank_db.obj;  
    Database * bank_db= & bank_db.obj;  
    bank_db =>open("Bank-DB");  
    d.Transaction Trans;  
    Trans.begin();
```

```
d_Ref<Account> account = new(bank_db)  
Account;  
d_Ref<Customer> cust = new(bank_db)  
Customer;  
cust->name = name;
```

ODMG C++ OML: Example

- ~~(Class)~~ Extents maintained automatically in the database.
- To access a class extent:

```
d_Extent<Customer>  
customerExtent(bank_db);
```
- Class d_Extent provides method

```
d_Iterator<T> create_iterator()
```

to create an iterator on the class extent
- Also provides `select(pred)` method to return iterator on objects that satisfy selection predicate pred.
- Iterators help step through objects in a

ODMG C++ OML: Example of

```
iterator<Customer> customers() {  
    Database bank_db_obj;  
    Database * bank_db = &bank_db_obj;  
    bank_db->open ("Bank-DB");  
    d_Transaction Trans; Trans.begin ();
```

```
    d_Extent<Customer>  
    all_customers(bank_db);  
    d_Iterator<d_Ref<Customer>> iter;  
    iter = all_customers->create_iterator();  
    d_Ref <Customer> p;  
    while{iter.next (p))
```

ODMG C++ Binding: Other

Features

- Declarative query language OQL, looks like SQL

- Form query as a string, and execute it to get a set of results (actually a bag, since duplicates may be present)

```
d_Set<d_Ref<Account>> result;  
d_OQL_Query q1("select a  
from Customer c,  
c.accounts a  
where c.name='Jones'  
and
```

```
a.find_balance() > 100");  
d_oql_execute(q1, result);
```

- Provides error handling mechanism

Making Pointer Persistence Transparent

- Drawback of the ODMG C++ approach:
 - Two types of pointers
 - Programmer has to ensure `mark_modified()` is called, else database can become corrupted
- ObjectStore approach
 - Uses *exactly* the same pointer type for in-memory and database objects
 - Persistence is transparent applications
 - Except when creating objects
 - Same functions can be used on in-memory objects and database objects

Persistent Java Systems

- ODMG-3.0 defines extensions to Java for persistence
 - Java does not support templates, so language extensions are required
- Model for persistence: persistence by reachability
 - Matches Java's garbage collection model
 - Garbage collection needed on the database also
 - Only one pointer type for transient and persistent pointers
- Class is made persistence capable by running a post-processor on object

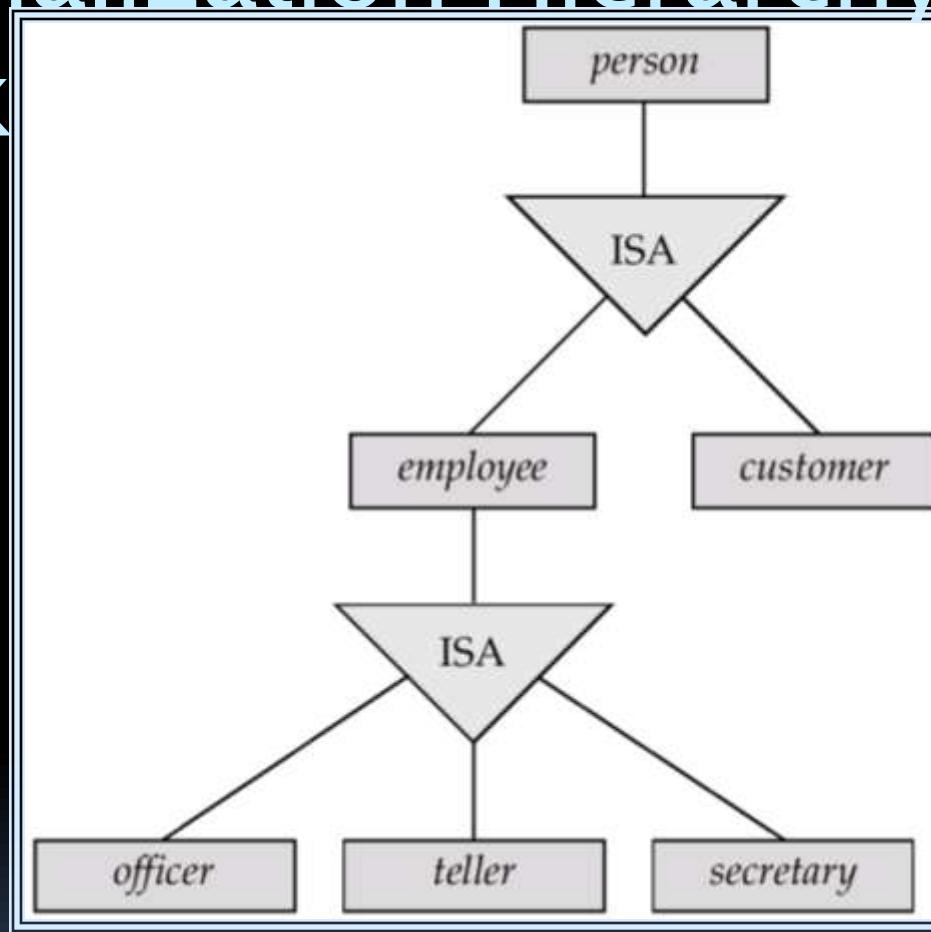
ODMG Java

- Transaction must start accessing database from one of the root object (looked up by name)
 - finds other objects by following pointers from the root objects
- Objects referred to from a fetched object are allocated space in memory, but not necessarily fetched
 - Fetching can be done lazily
 - An object with space allocated but not yet fetched is called a hollow object
 - When a hollow object is accessed, its data is fetched from disk.

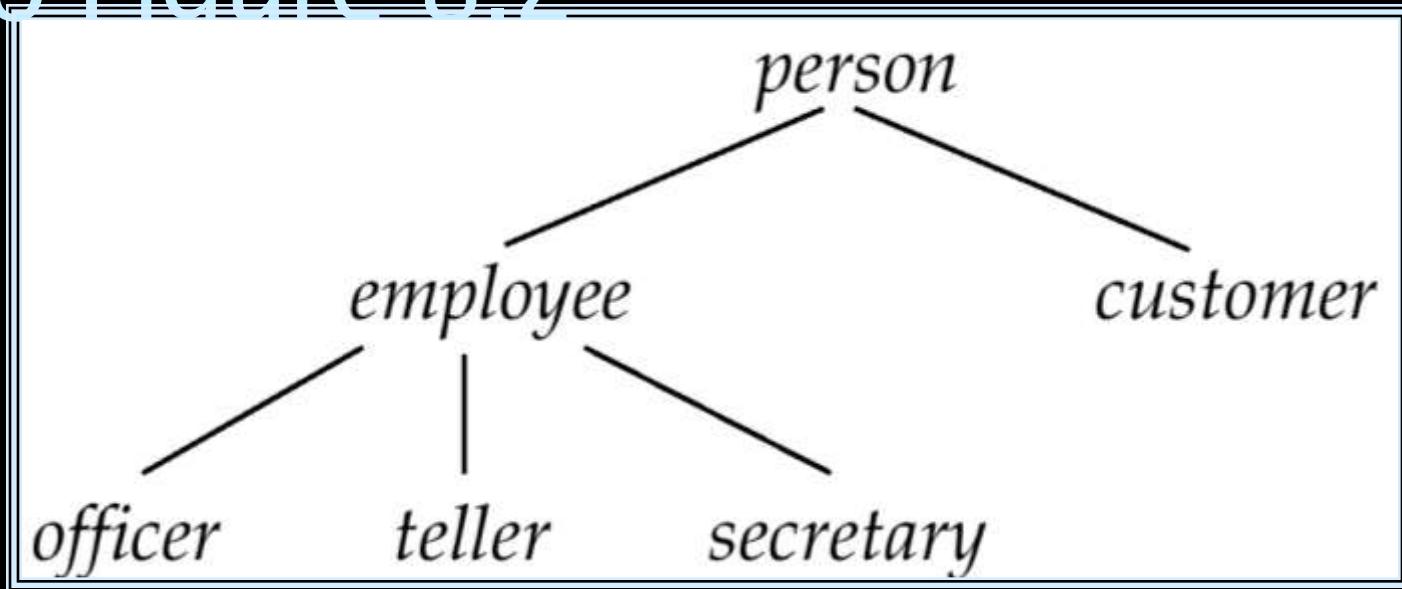
END OF CHAPTER



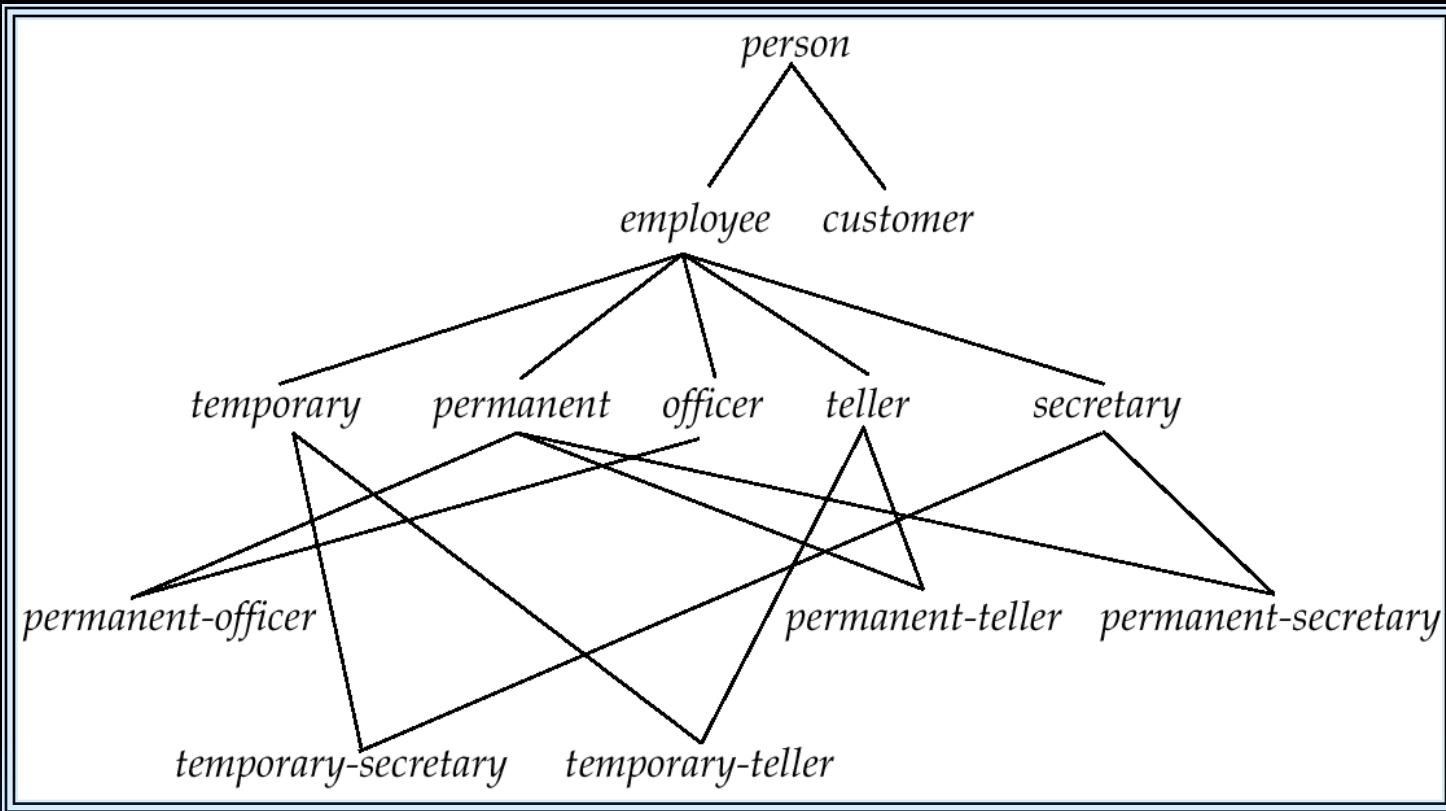
Specialization Hierarchy for the Bank



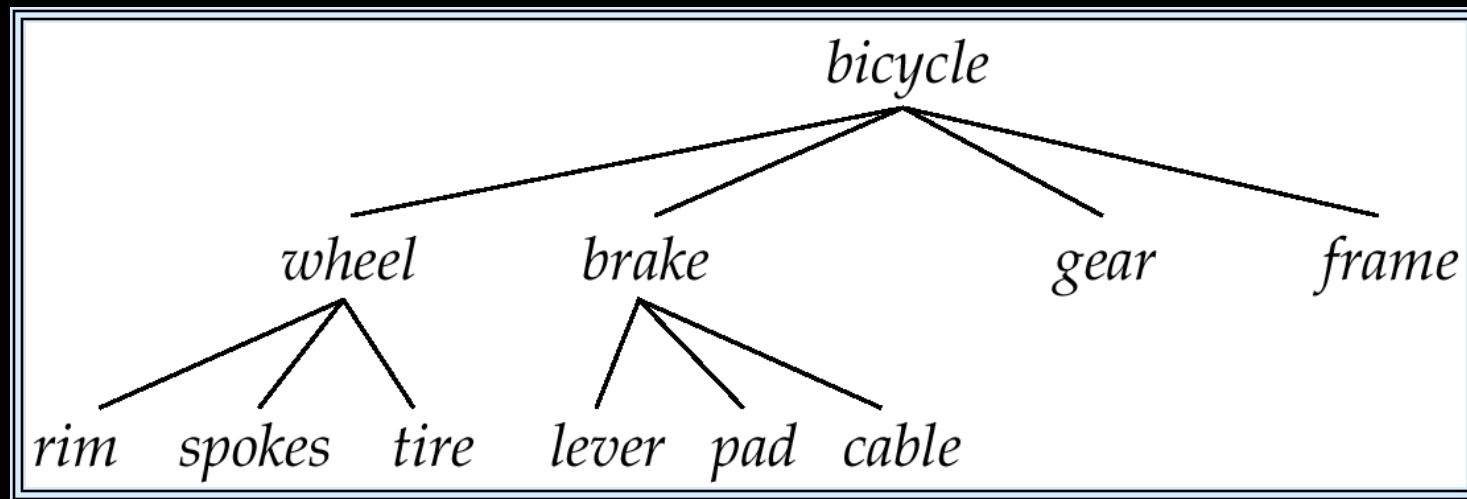
Class Hierarchy Corresponding to Figure 8.2



Class DAG for the Bank Example



Containment Hierarchy for Bicycle–Design Database



Chapter 9: Object-Relational Databases

- Nested Relations
- Complex Types and Object Orientation
- Querying with Complex Types
- Creation of Complex Values and Objects
- Comparison of Object-Oriented and Object-Relational Databases

Object–Relational Data Models

- Extend the relational data model by including object orientation and constructs to deal with added data types.
- Allow attributes of tuples to have complex types, including non-atomic values such as nested relations.
- Preserve relational foundations, in particular the declarative access to data, while extending modeling power.
- Upward compatibility with existing relational languages.

Nested Relations

- Motivation:
 - Permit non-atomic domains (atomic = indivisible)
 - Example of non-atomic domain: set of integers, or set of tuples
 - Allows more intuitive modeling for applications with complex data
- Intuitive definition:
 - allow relations whenever we allow atomic (scalar) values — relations within relations
 - Retains mathematical foundation of relational model
 - Violates first normal form.

Example of a Nested Relation

- Example: library information system
- Each book has
 - title,
 - a set of authors,
 - Publisher, and
 - a set of keywords
- Non-1NF relation *books*

<i>title</i>	<i>author-set</i>	<i>publisher</i> (<i>name, branch</i>)	<i>keyword-set</i>
Compilers	{Smith, Jones}	(McGraw-Hill, New York)	{parsing, analysis}
Networks	{Jones, Frick}	(Oxford, London)	{Internet, Web}

1NF Version of Nested Relation

- 1NF version of *books*

<i>title</i>	<i>author</i>	<i>pub-name</i>	<i>pub-branch</i>	<i>keyword</i>
Compilers	Smith	McGraw-Hill	New York	parsing
Compilers	Jones	McGraw-Hill	New York	parsing
Compilers	Smith	McGraw-Hill	New York	analysis
Compilers	Jones	McGraw-Hill	New York	analysis
Networks	Jones	Oxford	London	Internet
Networks	Frick	Oxford	London	Internet
Networks	Jones	Oxford	London	Web
Networks	Frick	Oxford	London	Web

flat-books

4NF Decomposition of Nested

- Remove awkwardness of *flat-books* by assuming that the following multivalued dependencies hold:
 - $\text{title} \rightarrow\!\!\! \rightarrow \text{author}$
 - $\text{title} \rightarrow\!\!\! \rightarrow \text{keyword}$
 - $\text{title} \rightarrow\!\!\! \rightarrow \text{pub-name}, \text{pub-branch}$
- Decompose *flat-doc* into 4NF using the schemas:
 - $(\text{title}, \text{author})$
 - $(\text{title}, \text{keyword})$
 - $(\text{title}, \text{pub-name}, \text{pub-branch})$

4NF Decomposition of *flat*-

<i>title</i>	<i>author</i>
Compilers	Smith
Compilers	Jones
Networks	Jones
Networks	Frick

authors

<i>title</i>	<i>keyword</i>
Compilers	parsing
Compilers	analysis
Networks	Internet
Networks	Web

keywords

<i>title</i>	<i>pub-name</i>	<i>pub-branch</i>
Compilers	McGraw-Hill	New York
Networks	Oxford	London

books4

Problems with 4NF Schema

- 4NF design requires users to include joins in their queries.
- 1NF relational view *flat-books* defined by join of 4NF relations:
 - eliminates the need for users to perform joins,
 - but loses the one-to-one correspondence between tuples and documents.
 - And has a large amount of redundancy
- Nested relations representation is much more natural here.

Complex Types and SQL:1999

- Extensions to SQL to support complex types include:
 - Collection and large object types
 - Nested relations are an example of collection types
 - Structured types
 - Nested record structures like composite attributes
 - Inheritance
 - Object orientation
 - Including object identifiers and references
- Our description is mainly based on the SQL:1999 standard

Not fully implemented in any database

Collection Types

- Set type (not in SQL:1999)

```
create table books (
    ....
    keyword-set setof(varchar(20))
    ....
)
```

- Sets are an instance of collection types.
Other instances include
 - Arrays (are supported in SQL:1999)
 - E.g. *author-array* **varchar(20) array[10]**
 - Can access elements of array in usual fashion:
 - *E.g. author-array[1]*
 - Multisets (not supported in SQL:1999)
 - I.e., unordered collections, where an element may occur multiple times
 - Nested relations are sets of tuples

Large Object Types

- Large object types
 - **clob**: Character large objects

book-review **clob**(10KB)
 - **blob**: binary large objects

image blob(10MB)
movie blob (2GB)
- JDBC/ODBC provide special methods to access large objects in small pieces
 - Similar to accessing operating system files
 - Application retrieves a **locator** for the large object and then manipulates the large object from the host language

Structured and Collection Types

- Structured types can be declared and used in SQL

```
create type Publisher as  
  (name          varchar(20),  
   branch        varchar(20))
```

```
create type Book as  
  (title         varchar(20),  
   author-array  varchar(20) array [10],  
   pub-date      date,  
   publisher     Publisher,  
   keyword-set   setof(varchar(20)))
```

- Note: **setof** declaration of keyword-set is not supported by SQL:1999
- Using an array to store authors lets us record the order of the authors
- Structured types can be used to create tables
 - create table *books* of *Book*
 - Similar to the nested relation books, but

Structured and Collection Types (Cont.)

- Structured types allow composite attributes of E-R diagrams to be represented directly.
- Unnamed row types can also be used in SQL:1999 to define composite attributes
 - **E.g.** we can omit the declaration of type *Publisher* and instead use the following in declaring the type *Book*

```
publisher row (name varchar(20),  
            branch
```

```
            varchar(20))
```

- Similarly, collection types allow multivalued attributes of E-R

. Structured Types (Cont.)

We can create tables without creating an intermediate type

- For example, the table *books* could also be defined as follows:

```
create table books
  (title varchar(20),
   author-array varchar(20) array[10],
   pub-date date,
   publisher Publisher
   keyword-list setof(varchar(20)))
```

- Methods can be part of the type definition of a structured type:

```
create type Employee as (
  name varchar(20),
  salary integer)

  method giveraise (percent integer)
```

- We create the method body separately

```
create method giveraise (percent integer) for Employee
begin
  set self.salary = self.salary + (self.salary * percent) / 100;
end
```

■ Creation of Values of Complex Types

- Values of structured types are created using constructor functions
 - E.g. *Publisher('McGraw-Hill', 'New York')*
 - Note: a value is **not** an object
- SQL:1999 constructor functions
 - E.g.
create function *Publisher(n varchar(20), b varchar(20))*
returns *Publisher*
begin
 set *name=n;*
 set *branch=b;*
end
 - Every structured type has a default constructor with no arguments, others can be defined as required
- Values of row type can be constructed by listing values in parantheses
 - E.g. given row type **row (name varchar(20), branch varchar(20))**

Creation of Values of Complex Types

- Array construction
 - array[**Silberschatz**, `Korth', `Sudarshan']
- Set value attributes (not supported in SQL:1999)
 - **set**(v1, v2, ..., vn)
- To create a tuple of the *books* relation
 - (`Compilers',
array[`Smith', `Jones'],
Publisher(`McGraw-Hill', `New
York'),
set(`parsing', `analysis'))
- To insert the preceding tuple into the relation *books*

■ Inheritance

Suppose that we have the following type definition for people:

```
create type Person  
  (name varchar(20),  
   address varchar(20))
```

- Using inheritance to define the student and teacher types

```
create type Student  
under Person  
  (degree      varchar(20),  
   department varchar(20))
```

```
create type Teacher  
under Person
```

Multiple Inheritance

- SQL:1999 does not support multiple inheritance
- If our type system supports multiple inheritance, we can define a type for teaching assistant as follows:

create type *Teaching Assistant*
under *Student, Teacher*

- To avoid a conflict between the two occurrences of *department* we can rename them

create type *Teaching Assistant*
under
Student with (*department* as
...)

Table Inheritance

- Table inheritance allows an object to have multiple types by allowing an entity to exist in more than one table at once.
- *E.g. people* table: create table *people* of *Person*
- We can then define the *students* and *teachers* tables as subtables of *people*

create table *students* of *Student*

under *people*

create table *teachers* of *Teacher*

under *people*

- Each tuple in a subtable (e.g. *students* and *teachers*) is implicitly present in its supertables (e.g. *people*)

Table Inheritance: Roles

- Table inheritance is useful for modeling roles
- permits a value to have multiple types, without having a most-specific type (unlike type inheritance).
 - e.g., an object can be in the *students* and *teachers subtables* simultaneously, without having to be in a subtable *student-teachers* that is under both *students* and *teachers*
 - object can gain/lose roles: corresponds to inserting/deleting object from a suhtable

Table Inheritance: Consistency Requirements

- Consistency requirements on subtables and supertables.
 - Each tuple of the supertable (e.g. *people*) can correspond to at most one tuple in each of the subtables (e.g. *students* and *teachers*)
 - Additional constraint in SQL:1999:
All tuples corresponding to each other (that is, with the same values for inherited attributes) must be derived from one tuple (inserted into one table).
 - That is, each entity must have a most specific type
 - We cannot have a tuple in *people*

Table Inheritance: Storage Alternatives

- Storage alternatives
 1. Store only local attributes and the primary key of the supertable in subtable
 - Inherited attributes derived by means of a join with the supertable
 2. Each table stores all inherited and locally defined attributes
 - Supertables implicitly contain (inherited attributes of) all tuples in their subtables
 - Access to all attributes of a tuple is faster: no join required
 - If entities must have most specific type,

Reference Types

- Object-oriented languages provide the ability to create and refer to objects.
- In SQL:1999
 - References are to tuples, and
 - References must be scoped,
 - I.e., can only point to tuples in one specified table
- We will study how to define references first, and later see how to use references

Reference Declaration in

E.g. define a type *Department* with a field *name* and a field *head* which is a reference to the type *Person*, with table *people* as scope

```
create type Department(  
    name varchar(20),  
    head ref(Person) scope people)
```

- We can then create a table *departments* as follows

```
create table departments of  
Department
```

- We can omit the declaration scope

Initializing Reference Typed

- In Oracle, to create a tuple with a reference value, we can first create the tuple with a null reference and then set the reference separately by using the function `ref(p)` applied to a tuple variable

- E.g. to create a department with name CS and head being the person named John, we use

```
insert into departments
values ('CS', null)
```

```
update departments
```

```
set head = (select ref(p)
```

```
from people as p
```

Initializing Reference Typed Values (Cont.)

SQL:1999 does not support the `ref()` function, and instead requires a special attribute to be declared to store the object identifier

- The self-referential attribute is declared by adding a `ref is` clause to the `create table` statement:

`create table people of Person
ref is oid system generated`

- Here, `oid` is an attribute name, not a keyword.

- To get the reference to a tuple, the subquery shown earlier would use
`select n oid`

User Generated Identifiers

- SQL:1999 allows object identifiers to be user-generated
 - The type of the object-identifier must be specified as part of the type definition of the referenced table, and
 - The table definition must specify that the reference is user generated
 - E.g.

```
create type Person
  (name varchar(20)
   address varchar(20))
  ref using varchar(20)
create table people of Person
  ref is oid user generated
```

User Generated Identifiers

■ We can then use the identifier value
(Cont.) when inserting a tuple into
departments

- Avoids need for a separate query to retrieve the identifier:

E.g. *insert into departments values(`CS`, `02184567`)*

- It is even possible to use an existing primary key value as the identifier, by including the ref from clause, and declaring the reference to be derived

create type Person

(name varchar(20) primary key,

address varchar(20))

ref from(name)

Path Expressions

- Find the names and addresses of the heads of all departments:

```
select head->name, head->address  
from departments
```
- An expression such as “head->name” is called a path expression
- Path expressions help avoid explicit joins
 - If department head were not a reference, a join of *departments* with *people* would be required to get at the address
 - Makes expressing the query much easier for the user

Querying with Structured Types

- Find the title and the name of the publisher of each book.

```
select title, publisher.name  
from books
```

Note the use of the dot notation to access fields of the composite attribute (structured type) *publisher*

Collection-Value Attributes

- Collection-valued attributes can be treated much like relations, using the keyword `unnest`
 - The *books* relation has array-valued attribute *author-array* and set-valued attribute *keyword-set*
- To find all books that have the word “database” as one of their keywords,

```
select title
      from books
      where 'database' in (unnest(keyword-set))
```

- Note: Above syntax is valid in SQL:1999, but the only collection type supported by

Collection Valued Attributes

- We can access individual elements of an array by using indices
 - E.g. If we know that a particular book has three authors, we could write:

```
select author-array[1], author-  
array[2], author-array[3]  
from books  
where title = `Database System  
Concepts'
```

Unnesting

- The transformation of a nested relation into a form with fewer (or no) relation-valued attributes us called unnesting.
- E.g.

```
select title, A as author, publisher.name
as pub_name,
      publisher.branch as pub_branch,
K as keyword
from books as B, unnest(B.author-array) as A, unnest (B.keyword-list) as K
```

Nesting

- Nesting is the opposite of unnesting, creating a collection-valued attribute
- NOTE: SQL:1999 does not support nesting
- Nesting can be done in a manner similar to aggregation, but using the function `set()` in place of an aggregation operation, to create a set
- To nest the *flat-books* relation on the attribute *keyword*:

```
select title, author, Publisher(pub_name, pub_branch) as publisher,  
      set(keyword) as keyword-list  
from flat-books  
groupby title, author, publisher
```
- To nest on both authors and keywords:

```
select title, set(author) as author-list,  
      Publisher(pub_name, pub_branch) as publisher,  
      set(keyword) as keyword-list  
from flat-books  
groupby title, publisher
```

Nesting (Cont.)

- Another approach to creating nested relations is to use subqueries in the select clause.

select title,

(select author

from flat-books as M

where M.title=O.title) as author-

set,

Publisher(pub-name, pub-branch)

as publisher,

(select keyword

from flat-books as N

where N.title = O.title) as

Functions and Procedures

- SQL:1999 supports functions and procedures
 - Functions/procedures can be written in SQL itself, or in an external programming language
 - Functions are particularly useful with specialized data types such as images and geometric objects
 - E.g. functions to check if polygons overlap, or to compare images for similarity
 - Some databases support **table-valued functions**, which can return a relation as a result
- SQL:1999 also supports a rich set of

SQL Functions

- Define a function that, given a book title, returns the count of the number of authors (on the 4NF schema with relations *books4* and *authors*).

```
create function author-count(name  
varchar(20))  
returns integer  
begin  
    declare a-count integer;  
    select count(author) into a-  
count  
    from authors
```

SQL Methods

- Methods can be viewed as functions associated with structured types
 - They have an implicit first parameter called **self** which is set to the structured-type value on which the method is invoked
 - The method code can refer to attributes of the structured-type value using the **self** variable
 - E.g. **self.a**

SQL Functions and Procedures

The *author-count* function could instead be written as procedure:
(cont.)

```
create procedure author-count-proc (in title varchar(20),
                                         out a-count integer)
begin
    select count(author) into a-count
        from authors
        where authors.title = title
end
```

- Procedures can be invoked either from an SQL procedure or from embedded SQL, using the call statement.
 - E.g. from an SQL procedure

```
declare a-count integer;
call author-count-proc('Database systems Concepts', a-count);
```

- SQL:1999 allows more than one function/procedure of the same name (called name overloading), as long as the number of arguments differ, or at least the types of the arguments differ

External Language

Functions / **P**rocedures
functions and procedures written in
other languages such as C or C++

- Declaring external language
procedures and functions

```
create procedure author-count-
proc(in title varchar(20),
```

```
out count integer)
language C
external name'
```

External Language Routines

Benefits of external language

functions/procedures:
(Cont.)

- more efficient for many operations, and more expressive power

Drawbacks

- Code to implement function may need to be loaded into database system and executed in the database system's address space
 - risk of accidental corruption of database structures
 - security risk, allowing users access to unauthorized data
- There are alternatives, which give good security at the cost of potentially worse performance
- Direct execution in the database system's space is used when efficiency is more important than security

Security with External Language

- To deal with security problems
 - **Routines**
 - Use **sandbox** techniques
 - that is use a safe language like Java, which cannot be used to access/damage other parts of the database code
 - Or, run external language functions/procedures in a separate process, with no access to the database process' memory
 - Parameters and results communicated via inter-process communication
 - Both have performance overheads
 - Many database systems support both above approaches as well as direct executing in database system address space

Procedural Constructs

- SQL:1999 supports a rich variety of procedural constructs
- Compound statement
 - is of the form **begin** ... **end**,
 - may contain multiple SQL statements between **begin** and **end**.
 - Local variables can be declared within a compound statements
- While and repeat statements

```
declare n integer default 0;
```

```
while n < 10 do
```

```
    set n = n+1
```

```
end while
```

```
repeat
```

```
    set n = n - 1
```

```
until n = 0
```

```
end repeat
```

Procedural Constructs (Cont.)

- For loop
 - Permits iteration over all results of a query
 - E.g. find total of all balances at the Perryridge branch

```
declare n integer default 0;  
for r as  
    select balance from account  
        where branch-name = 'Perryridge'  
do  
    set n = n + r.balance  
end for
```

Procedural Constructs (cont.)

Conditional statements (if-then-else)

E.g. To find sum of balances for each of three categories of accounts (with balance <1000, >=1000 and <5000, >= 5000)

```
if r.balance < 1000
    then set l = l + r.balance
elseif r.balance < 5000
    then set m = m + r.balance
else set h = h + r.balance
end if
```

- SQL:1999 also supports a case statement similar to C case statement
- Signaling of exception conditions, and declaring handlers for exceptions

```
declare out_of_stock condition
declare exit handler for out_of_stock
begin
...
    .. signal out-of-stock
end
```

- The handler here is **exit** -- causes enclosing begin..end to be exited

Comparison of O-O and O-R

- Summary of strengths of various database systems:
 - Databases
 - Relational systems
 - simple data types, powerful query languages, high protection.
 - Persistent-programming-language-based OODBs
 - complex data types, integration with programming language, high performance.
 - Object-relational systems
 - complex data types, powerful query languages, high protection.

Finding all employees of a manager

- Procedure to find all employees who work directly or indirectly for *mgr*
- Relation *manager(empname, mgrname)* specifies who directly works for whom
- Result is stored in *emp(name)*

create procedure *findEmp*(in *mgr* char(10))

begin

 create temporary table *newemp(name* char(10));

 create temporary table *temp(name* char(10));

 insert into *newemp* -- store all direct employees of *mgr* in
newemp

 select *empname*

 from *manager*

 where *mgrname* = *mgr*

Finding all employees of a manager(cont.)

```
repeat
    insert into empl          -- add all
    new employees found to empl
    select name
    from newemp;
    insert into temp           -- find all employees
    of people already found
    (select manager.empname
        from newemp, manager
        where newemp.empname =
              manager.mgrname;
     )
    except (                   -- but remove those
    who were found earlier
    select empname
    from empl
```

END OF CHAPTER



A Partially Nested Version of the *flat-books* Relation

<i>title</i>	<i>author</i>	<i>publisher</i> (pub-name, pub-branch)	<i>keyword-set</i>
Compilers	Smith	(McGraw-Hill, New York)	{parsing, analysis}
Compilers	Jones	(McGraw-Hill, New York)	{parsing, analysis}
Networks	Jones	(Oxford, London)	{Internet, Web}
Networks	Frick	(Oxford, London)	{Internet, Web}

CHAPTER 10: XML



Introduction

- XML: Extensible Markup Language
- Defined by the WWW Consortium (W3C)
- Originally intended as a document markup language not a database language
 - Documents have tags giving extra information about sections of the document
 - E.g. `<title> XML </title> <slide> Introduction ...</slide>`
 - Derived from SGML (Standard Generalized Markup Language), but simpler to use than SGML
 - Extensible unlike HTML

XML Introduction (Cont.)

- The ability to specify new tags, and to create nested tag structures made XML a great way to exchange data, not just documents.
 - Much of the use of XML has been in data exchange applications, not as a replacement for HTML
- Tags make data (relatively) self-documenting
 - E.g.

```
<bank>
  <account>
    <account-number> A-101
    </account-number>
```

XML: Motivation

- Data interchange is critical in today's networked world
 - Examples:
 - Banking: funds transfer
 - Order processing (especially inter-company orders)
 - Scientific data
 - Chemistry: ChemML, ...
 - Genetics: BSML (Bio-Sequence Markup Language), ...
 - Paper flow of information between organizations is being replaced by electronic flow of information

Each application uses its own set

XML Motivation (Cont.)

- Earlier generation formats were based on plain text with line headers indicating the meaning of fields
 - Similar in concept to email headers
 - Does not allow for nested structures, no standard “type” language
 - Tied too closely to low level document structure (lines, spaces, etc)
- Each XML based standard defines what are valid elements, using
 - XML type specification languages to specify the syntax
 - DTD (Document Type Descriptors)

Structure of XML Data

- Tag: label for a section of data
- Element: section of data beginning with *<tagname>* and ending with matching *</tagname>*
- Elements must be properly nested
 - Proper nesting
 - *<account> ... <balance> </balance></account>*
 - Improper nesting
 - *<account> ... <balance> </account></balance>*

Example of Nested Elements

```
<bank-1>
  <customer>
    <customer-name> Hayes </customer-name>
    <customer-street> Main </customer-street>
    <customer-city>    Harrison </customer-city>
    <account>
      <account-number> A-102 </account-number>
      <branch-name>    Perryridge </branch-name>
      <balance>        400 </balance>
    </account>
    <account>
      ...
    </account>
  </customer>
  .
  .
</bank-1>
```

Motivation for Nesting

- Nesting of data is useful in data transfer
 - Example: elements representing customer-id, customer name, and address nested within an order element
- Nesting is not supported, or discouraged, in relational databases
 - With multiple orders, customer name and address are stored redundantly
 - normalization replaces nested structures in each order by foreign key into table storing customer name and address information
 - Nesting is supported in object-relational

Structure of XML Data (Cont.)

- Mixture of text with sub-elements is legal in XML.
 - Example:

```
<account>
    This account is seldom used any more.
    <account-number> A-
102</account-number>
    <branch-name> Perryridge</branch-
name>
    <balance>400 </balance>
</account>
```
 - Useful for document markup, but discouraged for data representation

Attributes

- Elements can have attributes
 - ```
<account acct-type = "checking">
 <account-number> A-102
</account-number>
 <branch-name> Perryridge
</branch-name>
 <balance> 400 </balance>
</account>
```
- Attributes are specified by *name=value* pairs inside the starting tag of an element
- An element may have several attributes, but each attribute name can only occur once

# Attributes Vs. Subelements

- Distinction between subelement and attribute
  - In the context of documents, attributes are part of markup, while subelement contents are part of the basic document contents
  - In the context of data representation, the difference is unclear and may be confusing
    - Same information can be represented in two ways
      - <account account-number = “A-101”> ...

# More on XML Syntax

- Elements without subelements or text content can be abbreviated by ending the start tag with a `/>` and deleting the end tag
  - `<account number="A-101" branch="Perryridge" balance="200 />`
- To store string data that may contain tags, without the tags being interpreted as subelements, use CDATA as below
  - `<![CDATA[<account> ... </account>]]>`
    - Here, `<account>` and `</account>` are treated as just strings

# Namespaces

- XML data has to be exchanged between organizations
- Same tag name may have different meaning in different organizations, causing confusion on exchanged documents
- Specifying a unique string as an element name avoids confusion
- Better solution: use `unique-name:element-name`
- Avoid using long unique names all over document by using XML Namespaces

# XML Document Schema

- Database schemas constrain what information can be stored, and the data types of stored values
- XML documents are not required to have an associated schema
- However, schemas are very important for XML data exchange
  - Otherwise, a site cannot automatically interpret data received from another site
- Two mechanisms for specifying XML schema

# Document Type Definition (DTD)

- The type of an XML document can be specified using a DTD
- DTD constraints structure of XML data
  - What elements can occur
  - What attributes can/must an element have
  - What subelements can/must occur inside each element, and how many times.
- DTD does not constrain data types
  - All values represented as strings in XML
- DTD syntax

# Element Specification in DTD

- Subelements can be specified as
  - names of elements, or
  - #PCDATA (parsed character data), i.e., character strings
  - EMPTY (no subelements) or ANY (anything can be a subelement)
- Example

```
<! ELEMENT depositor (customer-name account-number)>
<! ELEMENT customer-name (#PCDATA)>
<! ELEMENT account-number (#PCDATA)>
```

- Subelement specification may have regular expressions

```
<!ELEMENT bank ((account | customer | depositor)+)>
```

# Bank DTD

```
<!DOCTYPE bank [
 <!ELEMENT bank ((account | customer | depositor)+)>
 <!ELEMENT account (account-number branch-name
balance)>
 <! ELEMENT customer(customer-name customer-street
 customer-
city)>
 <! ELEMENT depositor (customer-name account-number)>
 <! ELEMENT account-number (#PCDATA)>
 <! ELEMENT branch-name (#PCDATA)>
 <! ELEMENT balance(#PCDATA)>
 <! ELEMENT customer-name(#PCDATA)>
 <! ELEMENT customer-street(#PCDATA)>
 <! ELEMENT customer-city(#PCDATA)>
]>
```

# Attribute Specification in DTD

- Attribute specification : for each attribute
  - Name
  - Type of attribute
    - CDATA
    - ID (identifier) or IDREF (ID reference) or IDREFS (multiple IDREFs)
      - more on this later
  - Whether
    - mandatory (#REQUIRED)
    - has a default value (value),
    - or neither (#IMPLIED)
- Examples
  - <!ATTLIST account acct-type CDATA “checking”>

# IDs and IDREFs

- An element can have at most one attribute of type ID
- The ID attribute value of each element in an XML document must be distinct
  - Thus the ID attribute value is an object identifier
- An attribute of type IDREF must contain the ID value of an element in the same document
- An attribute of type IDREFS contains a set of (0 or more) ID values. Each ID

# Bank DTD with Attributes

- Bank DTD with ID and IDREF attribute types.

```
<!DOCTYPE bank-2[
 <!ELEMENT account (branch, balance)>
 <!ATTLIST account
 account-number ID # REQUIRED
 owners IDREFS # REQUIRED>
 <!ELEMENT customer(customer-name, customer-
street,
customer-city)>
 <!ATTLIST customer
 customer-id ID # REQUIRED
 accounts IDREFS # REQUIRED>
 ... declarations for branch, balance, customer-
name,
 customer-street and customer-
city]
```

# XML data with ID and IDREF attributes

```
<bank-2>
 <account account-number="A-401"
 owners="C100 C102">
 <branch-name> Downtown </branch-name>
 <balance> 500 </balance>
 </account>
 <customer customer-id="C100"
accounts="A-401">
 <customer-name>Joe </customer-name>
 <customer-street> Monroe </customer-street>
 <customer-city> Madison</customer-city>
 </customer>
 <customer customer-id="C102"
accounts="A-401 A-402">
 <customer-name> Mary </customer-name>
 <customer-street> Erin </customer-street>
 <customer-city> Newark </customer-city>
 </customer>
</bank-2>
```

# Limitations of DTDs

- No typing of text elements and attributes
  - All values are strings, no integers, reals, etc.
- Difficult to specify unordered sets of subelements
  - Order is usually irrelevant in databases
  - $(A \mid B)^*$  allows specification of an unordered set, but
    - Cannot ensure that each of A and B occurs only once

# XML Schema

- XML Schema is a more sophisticated schema language which addresses the drawbacks of DTDs. Supports
  - Typing of values
    - E.g. integer, string, etc
    - Also, constraints on min/max values
  - User defined types
  - Is itself specified in XML syntax, unlike DTDs
    - More standard representation, but verbose
  - Is integrated with namespaces

# XML Schema Version of Bank

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
<xsd:element name="bank" type="BankType"/>
<xsd:element name="account">
 <xsd:complexType>
 <xsd:sequence>
 <xsd:element name="account-number" type="xsd:string"/>
 <xsd:element name="branch-name" type="xsd:string"/>
 <xsd:element name="balance"
type="xsd:decimal"/>
 </xsd:sequence>
 </xsd:complexType>
</xsd:element>
..... definitions of customer and depositor
<xsd:complexType name="BankType">
 <xsd:sequence>
 <xsd:element ref="account" minOccurs="0"
maxOccurs="unbounded"/>
 <xsd:element ref="customer" minOccurs="0"
maxOccurs="unbounded"/>
 <xsd:element ref="depositor" minOccurs="0"
maxOccurs="unbounded"/>
 </xsd:sequence>
</xsd:complexType>
</xsd:schema>
```

# Querying and Transforming XML Data

- Translation of information from one XML schema to another
- Querying on XML data
- Above two are closely related, and handled by the same tools
- Standard XML querying/translation languages
  - XPath
    - Simple language consisting of path expressions
  - XSLT
    - Simple language designed for translation

# Tree Model of XML Data

- Query and transformation languages are based on a tree model of XML data
- An XML document is modeled as a tree, with nodes corresponding to elements and attributes
  - Element nodes have children nodes, which can be attributes or subelements
  - Text in an element is modeled as a text node child of the element
  - Children of a node are ordered according to their order in the XML document
  - Element and attribute nodes (except for the root node) have a single parent, which is an element node

# XPath

- XPath is used to address (select) parts of documents using path expressions
- A path expression is a sequence of steps separated by “/”
  - Think of file names in a directory hierarchy
- Result of path expression: set of values that along with their containing elements/attributes match the specified path

# XPath (Cont.)

- The initial “/” denotes root of the document (above the top-level tag)
- Path expressions are evaluated left to right
  - Each step operates on the set of instances produced by the previous step
- Selection predicates may follow any step in a path, in [ ]
  - E.g.    /bank-2/account[balance > 400]
    - returns account elements with a balance value greater than 400
    - /bank-2/account[balance] returns account

# Functions in XPath

- XPath provides several functions
  - The function **count()** at the end of a path counts the number of elements in the set generated by the path
    - E.g. **/bank-2/account[customer/count() > 2]**
      - Returns accounts with > 2 customers
    - Also function for testing position (1, 2, ..) of node w.r.t. siblings
  - Boolean connectives **and** and **or** and function **not()** can be used in predicates
  - IDREFs can be referenced using

# More XPath Features

Operator “|” used to implement union

- E.g. `/bank-2/account/id(@owner) | /bank-2/loan/id(@borrower)`
  - gives customers with either accounts or loans
  - However, “|” cannot be nested inside other operators.
- “//” can be used to skip multiple levels of nodes
  - E.g. `/bank-2//customer-name`
    - finds any `customer-name` element *anywhere* under the `/bank-2` element, regardless of the element in which it is contained.

A step in the path can go to:

parents, siblings, ancestors and descendants

# XSLT

- A stylesheet stores formatting options for a document, usually separately from document
  - E.g. HTML style sheet may specify font colors and sizes for headings, etc.
- The XML Stylesheet Language (XSL) was originally designed for generating HTML from XML
- XSLT is a general-purpose transformation language
  - Can translate XML to XML and XML to

# XSLT Templates

- Example of XSLT template with match and select part

```
<xsl:template match="/bank-
2/customer">
 <xsl:value-of select="customer-name"/>
 </xsl:template>
 <xsl:template match="*"/>
```

- The match attribute of `xsl:template` specifies a pattern in XPath
- Elements in the XML document matching the pattern are processed by the actions within the `xsl:template` element
  - `xsl:value-of` selects (outputs) specified values (here, `customer-name`)
  - For elements that do not match any

# XSLT Templates (Cont.)

- If an element matches several templates, only one is used
  - Which one depends on a complex priority scheme/user-defined priorities
  - We assume only one template matches any element

# Creating XML Output

- Any text or tag in the XSL stylesheet that is not in the xsl namespace is output as is
- E.g. to wrap results in new XML elements.

```
<xsl:template match="/bank-
2/customer">
 <customer>
 <xsl:value-of select="customer-name"/>
 </customer>
 </xsl:template>
 <xsl:template match="*"/>
```

- Example output:

```
<customer> Joe </customer>
```

# Creating XML Output (Cont.)

- Note: Cannot directly insert a **xsl:value-of** tag inside another tag
  - E.g. cannot create an attribute for `<customer>` in the previous example by directly using **xsl:value-of**
  - XSLT provides a construct **xsl:attribute** to handle this situation
    - **xsl:attribute** adds attribute to the preceding element
    - E.g. `<customer>`

```
<xsl:attribute name=“customer-id”>
<xsl:value-of select = “customer-
id”/>
</xsl:attribute>
</customer>
```

results in output of the form

# Structural Recursion

- Action of a template can be to recursively apply templates to the contents of a matched element
- E.g.

```
<xsl:template match="/bank">
 <customers>
 <xsl:template apply-templates/>
 </customers >
</xsl:template>
<xsl:template match="/customer">
 <customer>
 <xsl:value-of select="customer-name"/>
 </customer>
</xsl:template>
<xsl:template match="*"/>
```

- Example output:

```
<customers>
 <customer> John </customer>
 <customer> Mary </customer>
</customers>
```

# Joins in XSLT

- XSLT **keys** allow elements to be looked up (indexed) by values of subelements or attributes

- Keys must be declared (with a name) and, the key() function can then be used for lookup. E.g.

- ```
<xsl:key name="acctno" match="account"
          use="account-number"/>
```
 - ```
<xsl:value-of select=key("acctno", "A-101")
```

- Keys permit (some) joins to be expressed in XSLT

```
<xsl:key name="acctno" match="account" use="account-number"/>
<xsl:key name="custno" match="customer" use="customer-name"/>
<xsl:template match="depositor">
 <cust-acct>
 <xsl:value-of select=key("custno", "customer-name")/>
 <xsl:value-of select=key("acctno", "account-number")/>
 </cust-acct>
</xsl:template>
<xsl:template match="*"/>
```

# Sorting in XSLT

- Using an `xsl:sort` directive inside a template causes all elements matching the template to be sorted
  - Sorting is done before applying other templates
- E.g.

```
<xsl:template match="/bank">
 <xsl:apply-templates
select="customer">
 <xsl:sort select="customer-
name"/>
 </xsl:apply-templates>
</xsl:template>
<xsl:template match="customer">
```

# XQuery

- XQuery is a general purpose query language for XML data
- Currently being standardized by the World Wide Web Consortium (W3C)
  - The textbook description is based on a March 2001 draft of the standard. The final version may differ, but major features likely to stay unchanged.
- Alpha version of XQuery engine available free from Microsoft
- XQuery is derived from the Quilt query language, which itself borrows from SQL, XQL and XML-QL
- XQuery uses a
  - for ... let ... where .. result ...syntax
  - for      ⇔ SQL from
  - where ⇔ SQL where

# FLWR Syntax in XQuery

- For clause uses XPath expressions, and variable in for clause ranges over values in the set returned by XPath
- Simple FLWR expression in XQuery
  - find all accounts with balance > 400, with each result enclosed in an `<account-number> .. </account-number>` tag

```
for $x in /bank-2/account
let $acctno := $x/@account-number
where $x/balance > 400
return <account-number> $acctno </account-
number>
```

- Let clause not really needed in this query, and selection can be done In XPath - `($x/balance > 400)`

# Path Expressions and Functions

- Path expressions are used to bind variables in the for clause, but can also be used in other places
  - E.g. path expressions can be used in **let** clause, to bind variables to results of path expressions
- The function **distinct()** can be used to remove duplicates in path expression results
- The function **document(name)** returns root of named document

## ■ Joins

- Joins are specified in a manner very similar to SQL

```
for $a in /bank/account,
 $c in /bank/customer,
 $d in /bank/depositor
 where $a/account-number =
 $d/account-number
 and $c/customer-name =
 $d/customer-name
 return <cust-acct> $c $a </cust-
 acct>
```

- The same query can be expressed with the selections specified as XPath selections:

```
for $a in /bank/account
```

# Changing Nesting Structure

- The following query converts data from the flat structure for **bank** information into the nested structure used in **bank-1**

```
<bank-1>
for $c in /bank/customer
return
<customer>
$c/*
for $d in /bank/depositor[customer-name = $c/customer-
name],
$a in /bank/account[account-number=$d/account-
number]
return $a
</customer>
</bank-1>
```

- $\$c/*$  denotes all the children of the node to which  $\$c$  is bound, without the enclosing top-level tag

# XQuery Path Expressions

- `$c/text()` gives text content of an element without any subelements/tags
- XQuery path expressions support the “->” operator for dereferencing IDREFs
  - Equivalent to the `id()` function of XPath, but simpler to use
  - Can be applied to a set of IDREFs to get a set of results
  - June 2001 version of standard has changed “->” to “=>”

# Sorting in XQuery

- Sortby clause can be used at the end of any expression. E.g. to return customers sorted by name

```
for $c in /bank/customer
return <customer> $c/* </customer>
sortby(name)
```

- Can sort at multiple levels of nesting (sort by customer-name, and by account-number within each customer)

```
<bank-1>
for $c in /bank/customer
return
<customer>
```

# Functions and Other XQuery

- ~~User-defined~~ functions with the type system of XMLSchema

```
function balances(xs:string $c)
returns list(xs:numeric) {
 for $d in /bank/depositor[customer-
name = $c],
 $a in /bank/account[account-
number=$d/account-number]
 return $a/balance
}
```

- Types are optional for function parameters and return values

# Application Program Interface

- There are two standard application program interfaces to XML data:
  - SAX (Simple API for XML)
    - Based on parser model, user provides event handlers for parsing events
      - E.g. start of element, end of element
      - Not suitable for database applications
  - DOM (Document Object Model)
    - XML data is parsed into a tree representation
    - Variety of functions provided for traversing the DOM tree
    - E.g.: Java DOM API provides Node class with methods
      - getparentNode( ), getChild( ),
      - getNextSibling( )
      - getAttribute( ), getData( ) (for text node)
      - getElementsByTagName( ), ...
    - Also provides functions for updating DOM tree

# Storage of XML Data

- XML data can be stored in
  - Non-relational data stores
    - Flat files
      - Natural for storing XML
      - But has all problems discussed in Chapter 1 (no concurrency, no recovery, ...)
    - XML database
      - Database built specifically for storing XML data, supporting DOM model and declarative querying
      - Currently no commercial-grade systems
  - Relational databases
    - Data must be translated into relational form
    - Advantage: mature database systems

# Storage of XML in Relational Databases

- Alternatives:
  - String Representation
  - Tree Representation
  - Map to relations

# String Representation

- Store each top level element as a string field of a tuple in a relational database
  - Use a single relation to store all elements, or
  - Use a separate relation for each top-level element type
    - E.g. account, customer, depositor relations
      - Each with a string-valued attribute to store the element
- Indexing:
  - Store values of subelements/attributes to be indexed as extra fields of the relation,

# String Representation (Cont.)

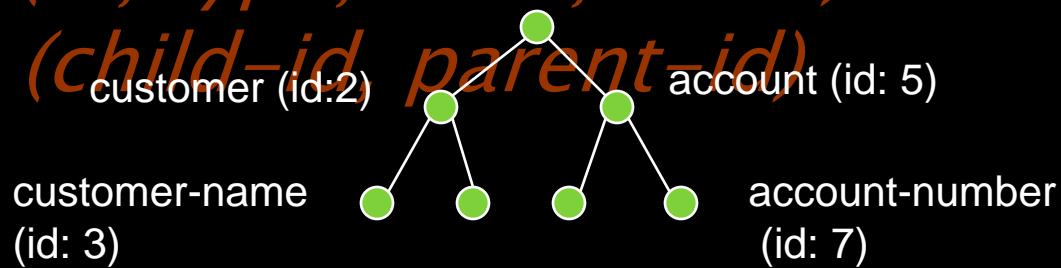
- Benefits:
  - Can store any XML data even without DTD
  - As long as there are many top-level elements in a document, strings are small compared to full document
    - Allows fast access to individual elements.
- Drawback: Need to parse strings to access values inside the elements
  - Parsing is slow.

# Tree Representation

- Tree representation: model XML data as tree and store using relations

*nodes(id, type, label, value)*

*child (child-id, parent-id)*



- Each element/attribute is given a unique identifier
- Type indicates element/attribute

# Tree Representation (Cont.)

- Benefit: Can store any XML data, even without DTD
- Drawbacks:
  - Data is broken up into too many pieces, increasing space overheads
  - Even simple queries require a large number of joins, which can be slow

# Mapping XML Data to Relations

## ■ Map to relations

- If DTD of document is known, can map data to relations
- A relation is created for each element type
  - Elements (of type #PCDATA), and attributes are mapped to attributes of relations
  - More details on next slide ...

## ■ Benefits:

- Efficient storage
- Can translate XML queries into SQL, execute efficiently, and then translate SQL results back to XML
- Drawbacks: need to know DTD,

# Mapping XML Data to Relations

- Relation created for each element type
  - (Cont.)
    - An id attribute to store a unique id for each element
    - A relation attribute corresponding to each element attribute
    - A parent-id attribute to keep track of parent element
      - As in the tree representation
      - Position information ( $i^{\text{th}}$  child) can be stored too
  - All subelements that occur only once can become relation attributes

# Mapping XML Data to Relations

- E.g. For bank-1 DTD with **account** elements nested within **customer** elements, create relations
  - **customer**(id, parent-id, customer-name, customer-stret, customer-city)
    - parent-id can be dropped here since parent is the sole root element
    - All other attributes were subelements of type #PCDATA, and occur only once
  - **account** (id, parent-id, account-number, branch-name, balance)
    - parent-id keeps track of which customer an account occurs under
    - Same account may be represented many times with different parents

# Chapter 11: Storage and File Structure

- Magnetic Disks
- RAID
- Tertiary Storage
- Storage Access
- File Organization
- Organization of Records in Files

## ■ 3. DATA STORAGE AND INDEXING

Storage & File Structure-Disks-RAID-File Organization-Indexing

& Hashing-B-TREE-B-Tree-Static Hashing-Dynamic Hashing-Multiple Key Access

Databases

# 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

# 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

# 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 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

# 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
- Hard disks vs floppy disks
- Capacities range up to roughly 100 GB currently

# Physical Storage Media (Cont.)

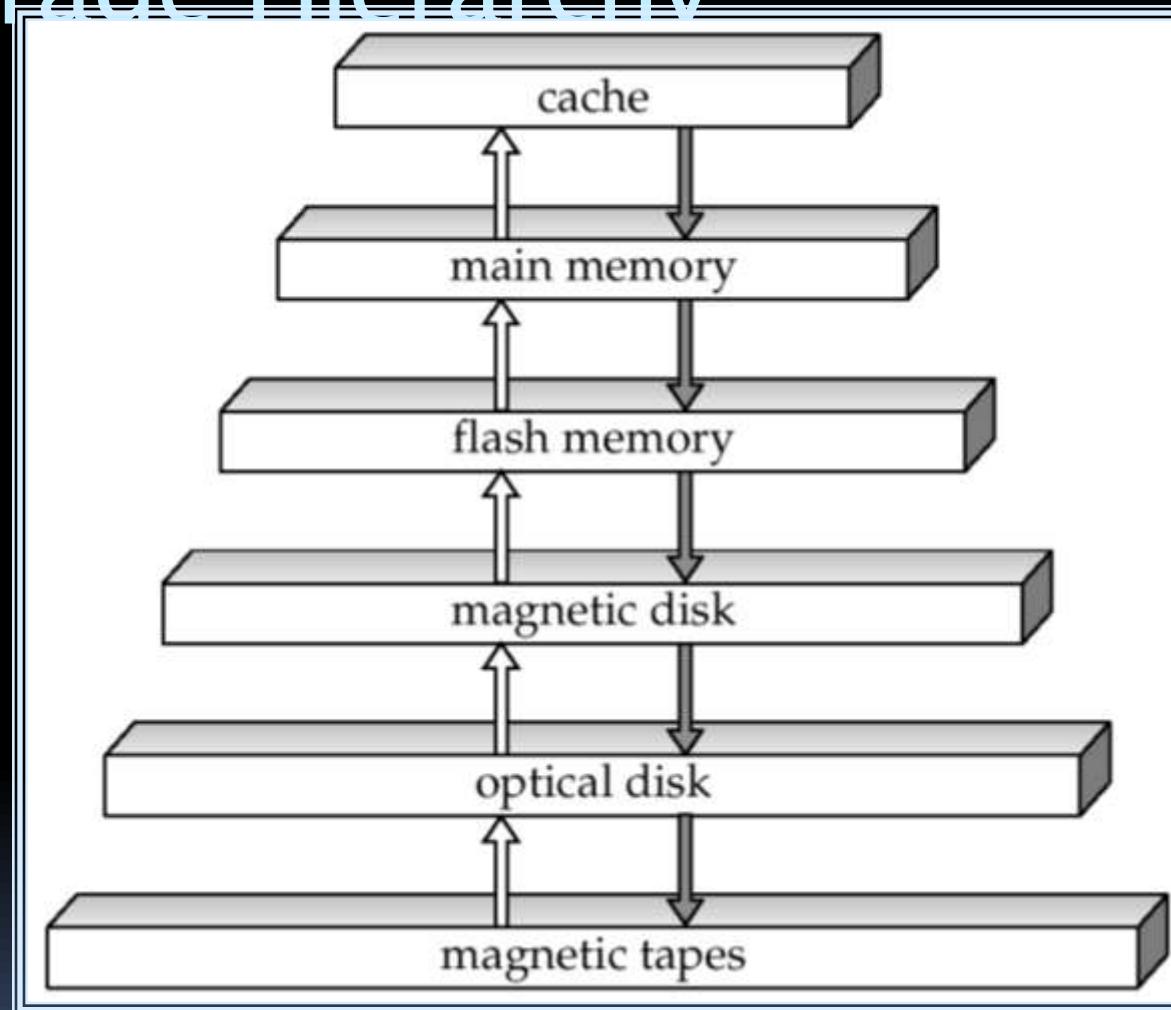
- Optical storage
  - non-volatile, data is read optically from a spinning disk using a laser
  - CD-ROM (640 MB) and DVD (4.7 to 17 GB) most popular forms
  - Write-one, read-many (WORM) optical disks used for archival storage (CD-R and DVD-R)
  - Multiple write versions also available (CD-RW, DVD-RW,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

# Physical Storage Media (Cont.)

- Tape storage

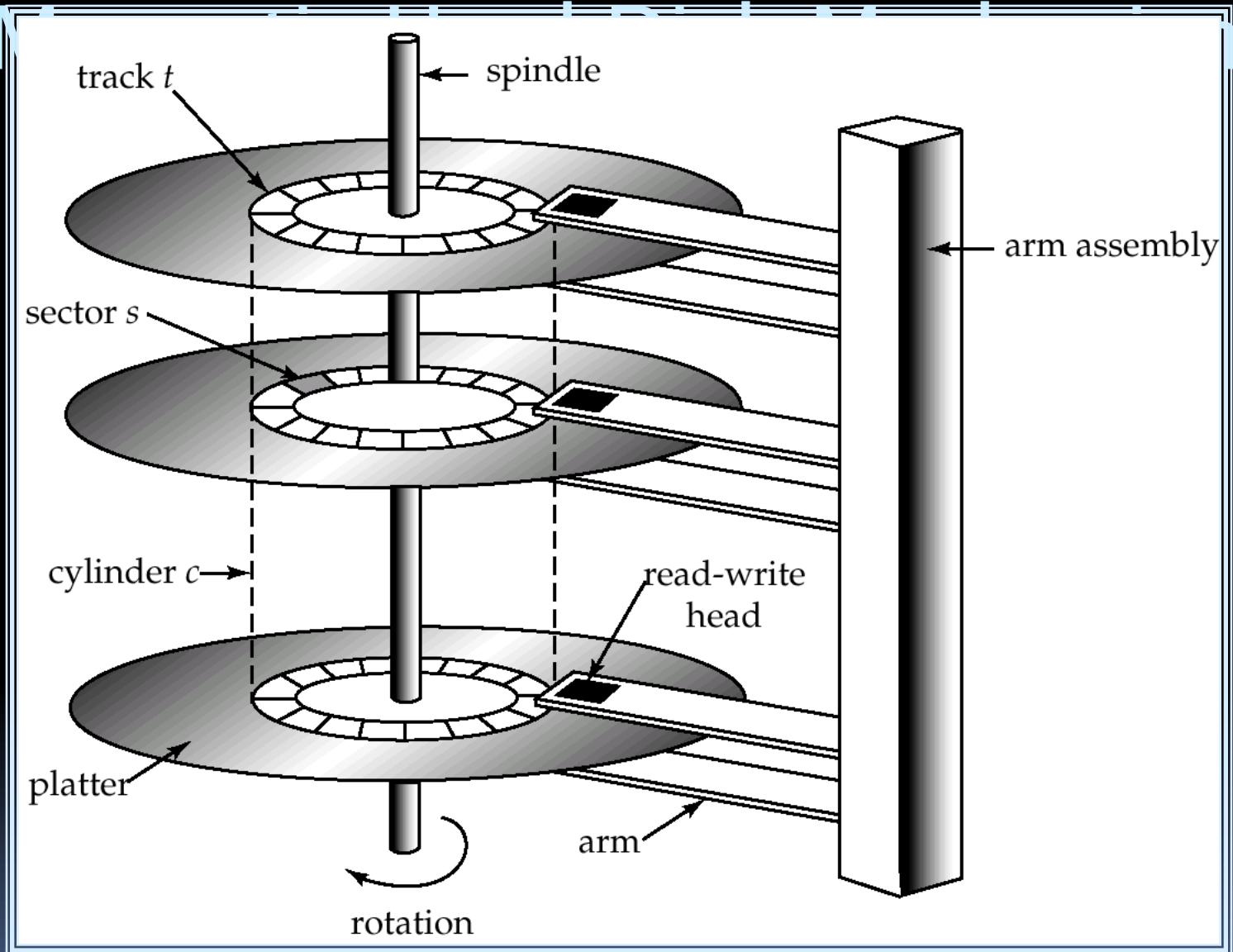
- non-volatile, used primarily for backup (to recover from disk failure), and for archival data
- **sequential-access** - much slower than disk
- very high capacity (40 to 300 GB tapes available)
- tape can be removed from drive ⇒ storage costs much cheaper than disk, but drives are expensive
- Tape jukeboxes available for storing

# 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**



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

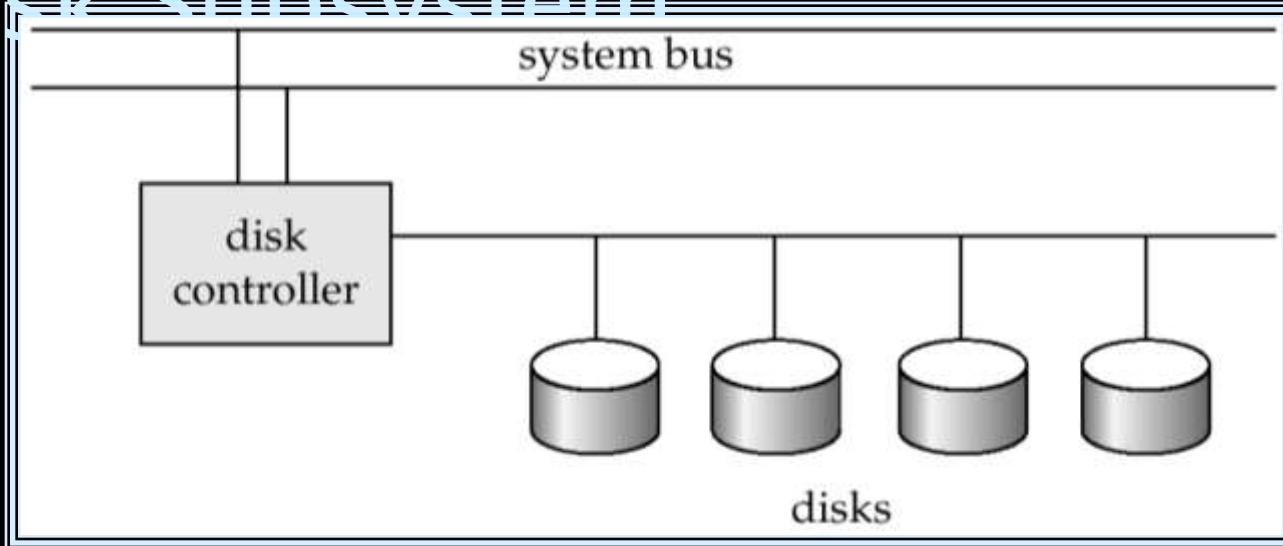
# Magnetic Disks

- Read-write head
  - Positioned very close to the platter surface (almost touching it)
  - Reads or writes magnetically encoded information.
- Surface of platter divided into circular tracks
  - Over 16,000 tracks per platter on typical hard disks
- Each track is divided into sectors.
  - A sector is the smallest unit of data that can be read or written.
  - Sector size typically 512 bytes
  - Typical sectors per track: 200 (on inner tracks) to 400 (on outer tracks)
- To read/write a sector
  - disk arm swings to position head on right

# Magnetic Disks (Cont.)

- Earlier generation disks were susceptible to head-crashes
  - Surface of earlier generation disks had metal-oxide coatings which would disintegrate on head crash and damage all data on disk
  - Current generation disks are less susceptible to such disastrous failures, although individual sectors may get corrupted
- Disk controller - interfaces between the computer system and the disk drive hardware.
  - accepts high-level commands to read or write a sector

# Disk Subsystem



- Multiple disks connected to a computer system through a controller
  - Controllers functionality (checksum, bad sector remapping) often carried out by individual disks; reduces load on controller
- Disk interface standards families
  - ATA (AT adaptor) range of standards

# 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.)

# 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 30,000 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

# 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

# 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

# Optimization of Disk Block

- 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

# 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

# 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

# Improvement in Performance via

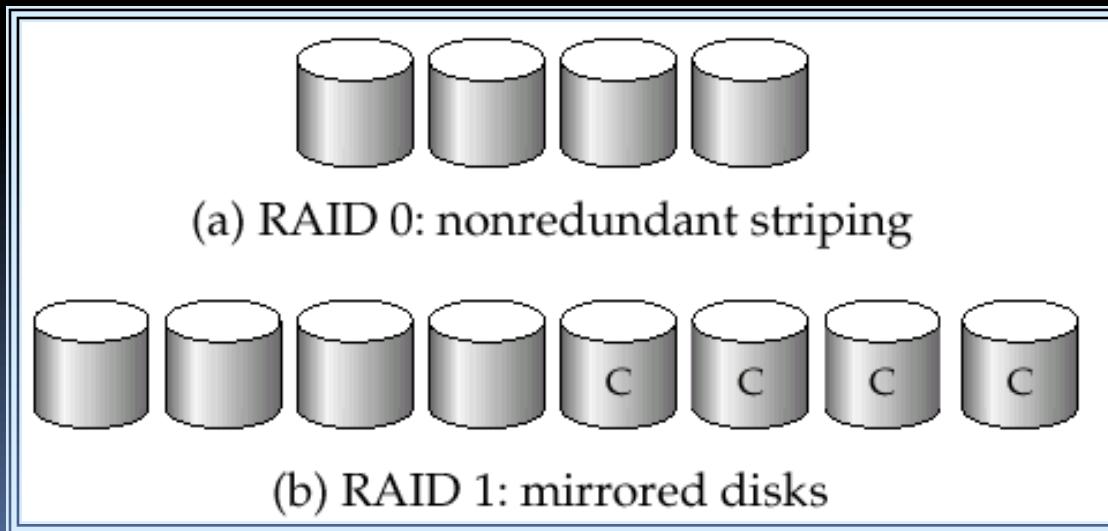
## Parallelism

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.

# RAID Levels

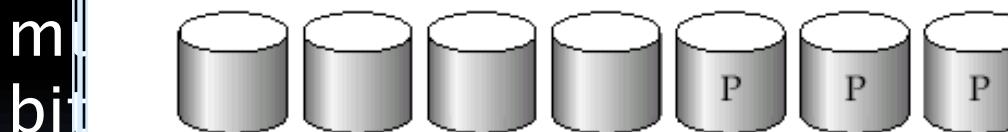
- Schemes to provide redundancy at lower cost by using disk striping combined with parity bits
- RAID Level 0: Block striping; non-redundant.
  - Different RAID organizations, or RAID levels, have differing cost, performance and reliability characteristics
    - Used in high-performance applications where data lost is not critical.
- RAID Level 1: Mirrored disks with block striping
  - Offers best write performance.
  - Popular for applications such as storing log files in a database system.



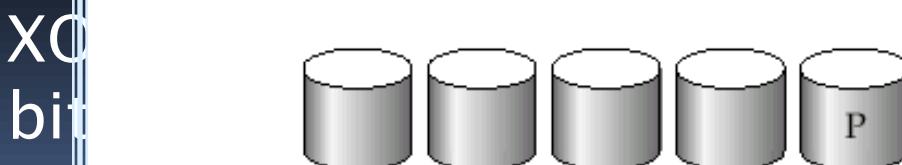
# RAID Levels (Cont.)

- RAID Level 2: Memory-Style Error-Correcting-Codes (ECC) with bit striping.
- RAID Level 3: Bit-Interleaved Parity
  - a single parity bit is enough for error correction, not just detection, since we know which disk has failed

- When writing data, corresponding parity bits



- To compute parity



(d) RAID 3: bit-interleaved parity

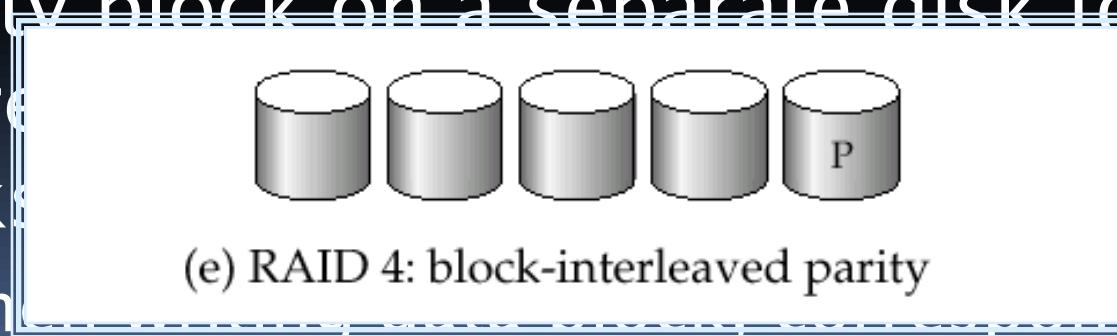
# RAID Levels (Cont.)

- RAID Level 3 (Cont.)

- Faster data transfer than with a single disk, but fewer I/Os per second since every disk has to participate in every I/O.
- Subsumes Level 2 (provides all its benefits, at lower cost).

- RAID Level 4: Block-Interleaved Parity; uses block-level striping, and keeps a parity block on a separate disk for correction. It is also known as **bit-interleaved parity**.

- While reading, the entire data block must be read from all the data disks, and the corresponding block of parity bits must also be computed.



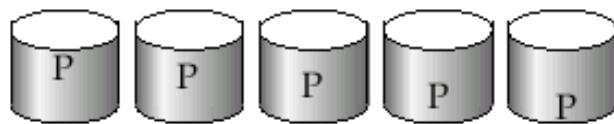
# RAID Levels (Cont.)

- RAID Level 4 (Cont.)
  - Provides higher I/O rates for independent block reads than Level 3
    - block read goes to a single disk, so blocks stored on different disks can be read in parallel
  - Provides high transfer rates for reads of multiple blocks than no-striping
  - Before writing a block, parity data must be computed
    - Can be done by using old parity block, old value of current block and new value of

# RAID Levels (Cont.)

- RAID Level 5: Block-Interleaved Distributed Parity; partitions data and parity among all  $N + 1$  disks, rather than storing data in  $N$  disks

and



\$ (f) RAID 5: block-interleaved distributed parity

or  $n$ th  
 $n \bmod 5$ )

+ with the data blocks stored on the

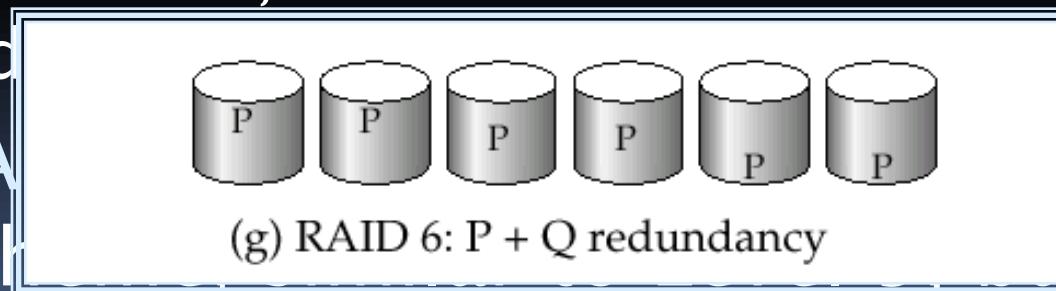
other

P0	0	1	2	3
4	P1	5	6	7
8	9	P2	10	11
12	13	14	P3	15
16	17	18	19	P4

# RAID Levels (Cont.)

- RAID Level 5 (Cont.)
  - Higher I/O rates than Level 4.
    - Block writes occur in parallel if the blocks and their parity blocks are on different disks.
  - Subsumes Level 4: provides same benefits, but avoids bottleneck of parity disk.

- RAID 6
  - It stores extra redundant information to guard against multiple disk failures



# Choice of RAID Level

- Factors in choosing RAID level
  - Monetary cost
  - Performance: Number of I/O operations per second, and bandwidth during normal operation
  - Performance during failure
  - Performance during rebuild of failed disk
    - Including time taken to rebuild failed disk
- RAID 0 is used only when data safety is not important
  - E.g. data can be recovered quickly from other sources
- Level 2 and 4 never used since they are subsumed by 3 and 5
- Level 3 is not used anymore since bit-striping forces single block reads to access all disks, wasting disk arm movement

# Choice of RAID Level (Cont.)

- Level 1 provides much better write performance than level 5
  - Level 5 requires at least 2 block reads and 2 block writes to write a single block, whereas Level 1 only requires 2 block writes
  - Level 1 preferred for high update environments such as log disks
- Level 1 had higher storage cost than level 5
  - disk drive capacities increasing rapidly (50%/year) whereas disk access times have decreased much less (x 3 in 10 years)
  - I/O requirements have increased

# Hardware Issues

- Software RAID: RAID implementations done entirely in software, with no special hardware support
- Hardware RAID: RAID implementations with special hardware
  - Use non-volatile RAM to record writes that are being executed
  - Beware: power failure during write can result in corrupted disk
    - E.g. failure after writing one block but

# Hardware Issues (Cont.)

- Hot swapping: replacement of disk while system is running, without power down
  - Supported by some hardware RAID systems,
  - reduces time to recovery, and improves availability greatly
- Many systems maintain spare disks which are kept online, and used as replacements for failed disks immediately on detection of failure

# Optical Disks

- Compact disk-read only memory (CD-ROM)
  - Disks can be loaded into or removed from a drive
  - High storage capacity (640 MB per disk)
  - High seek times or about 100 msec (optical read head is heavier and slower)
  - Higher latency (3000 RPM) and lower data-transfer rates (3–6 MB/s) compared to magnetic disks
- Digital Video Disk (DVD)
  - DVD-5 holds 4.7 GB , and DVD-9 holds 8.5 GB
  - DVD-10 and DVD-18 are double sided formats with capacities of 9.4 GB and 17 GB

# Magnetic Tapes

- Hold large volumes of data and provide high transfer rates
  - Few GB for DAT (Digital Audio Tape) format, 10–40 GB with DLT (Digital Linear Tape) format, 100 GB+ with Ultrium format, and 330 GB with Ampex helical scan format
  - Transfer rates from few to 10s of MB/s
- Currently the cheapest storage medium
  - Tapes are cheap, but cost of drives is very high
- Very slow access time in comparison to magnetic disks and optical disks
  - limited to sequential access.
  - Some formats (Accelis) provide faster seek

# 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

# Buffer Manager

- Programs call on the buffer manager when they need a block from disk.
  1. If the block is already in the buffer, the requesting program is given the address of the block in main memory
  2. If the block is not in the buffer,
    1. the buffer manager allocates space in the buffer for the block, replacing (throwing out) some other block, if required, to make space for the new block.
    2. The block that is thrown out is written back to disk only if it was modified since the most recent time that it was written to/fetched from the disk.

# Buffer-Replacement Policies

- Most operating systems replace the block least recently used (LRU strategy)
- Idea behind LRU – use past pattern of block references as a predictor of future references
- Queries have well-defined access patterns (such as sequential scans), and a database system can use the information in a user's query to predict future references
  - LRU can be a bad strategy for certain access patterns involving repeated scans of data

- # Buffer-Replacement Policies
- (Cont.)
- Pinned block - memory block that is not allowed to be written back to disk.
  - Toss-immediate strategy - frees the space occupied by a block as soon as the final tuple of that block has been processed
  - Most recently used (MRU) strategy - system must pin the block currently being processed. After the final tuple of that block has been processed, the block is unpinned, and it becomes the most recently used block.

# 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

# 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 block boundaries

■ Deletion of record alternatives:

- move records  $i + 1$ , to  $i, \dots, n - 1$
- move record  $n$  to  $i$
- do not move records, but link all free records on a

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.)

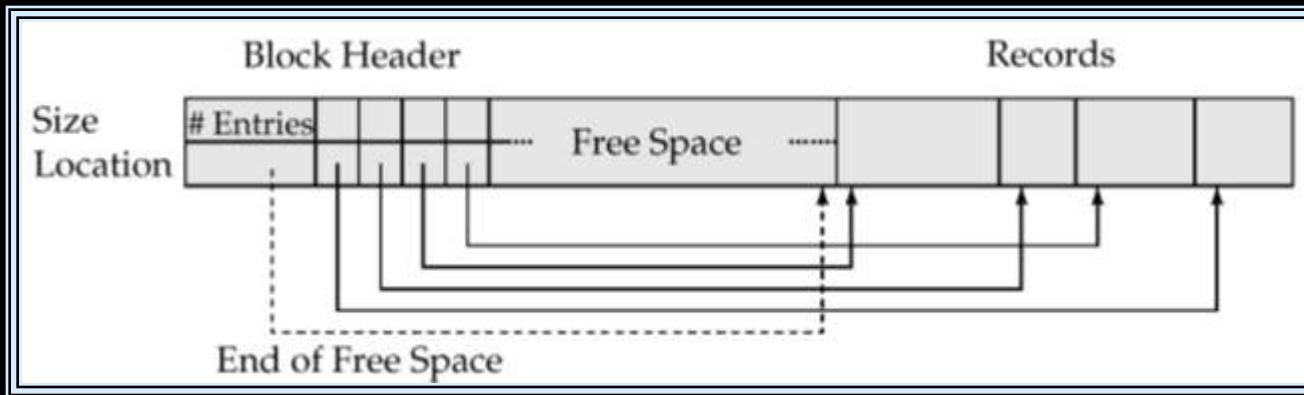
header			
record 0	A-102	Perryridge	400
record 1			
record 2	A-215	Mianus	700
record 3	A-101	Downtown	500
record 4			
record 5	A-201	Perryridge	900
record 6			
record 7	A-110	Downtown	600
record 8	A-218	Perryridge	700

The diagram illustrates a linked list of deleted records. The 'header' row contains the address of the first deleted record (400). The 'record 0' row contains the address of the second deleted record (700). The 'record 1' row contains the address of the third deleted record (500). The 'record 2' row contains the address of the fourth deleted record (900). The 'record 3' row contains the address of the fifth deleted record (600). The 'record 4' row is empty. The 'record 5' row contains the address of the sixth deleted record (700). The 'record 6' row is empty. The 'record 7' row contains the address of the seventh deleted record (400). The 'record 8' row is empty.

# 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).
- Byte string representation
  - Attach an *end-of-record* ( $\perp$ ) control character to the end of each record
  - Difficulty with deletion
  - Difficulty with growth

# 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.

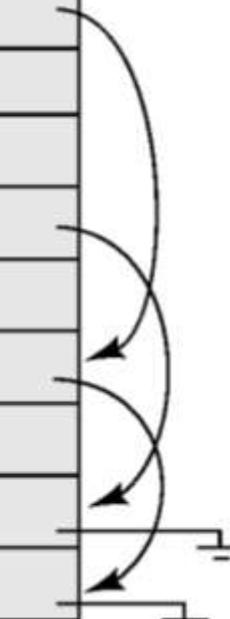
# Variable-Length Records (Cont.)

- Fixed-length representation:
  - reserved space
  - pointers
- Reserved space - can use fixed-length records of a known length

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

# Pointers

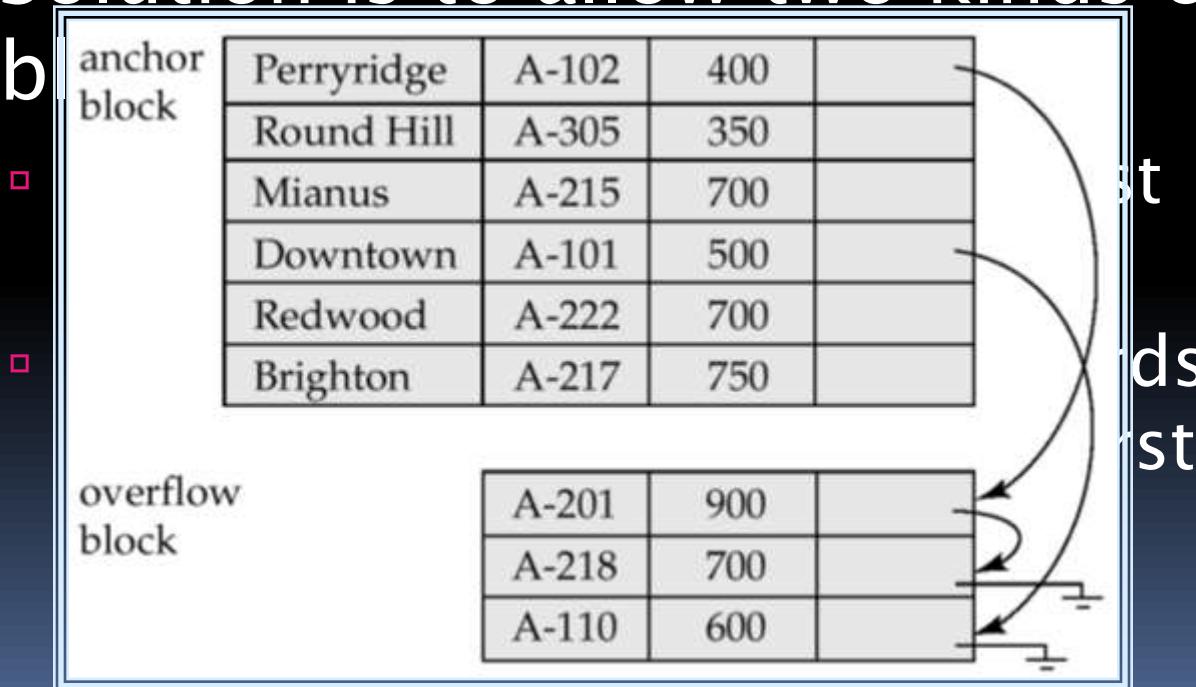
	Address	Value	Next
0	Perryridge	A-102	400
1	Round Hill	A-305	350
2	Mianus	A-215	700
3	Downtown	A-101	500
4	Redwood	A-222	700
5		A-201	900
6	Brighton	A-217	750
7		A-110	600
8		A-218	700



- Pointer method
  - A variable-length record is represented by a list of fixed-length records, chained together via pointers.
  - Can be used even if the maximum record length is not known

# Pointer Method (Cont.)

- Disadvantage to pointer structure; space is wasted in all records except the first in a chain.
- Solution is to allow two kinds of



# 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

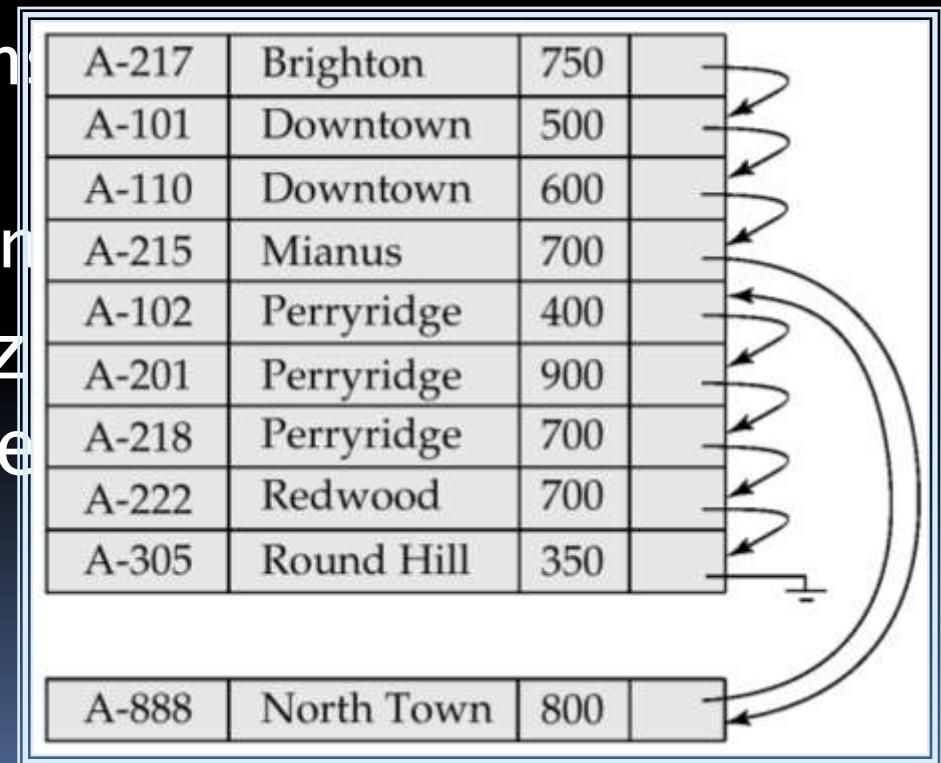
# Sequential File Organization

Suitable for applications that require sequential processing of the entire file

- The records in the file are

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

- # Sequential File Organization
- Deletion - use pointer chains  
**(Cont.)**
  - Insertion - locate the position where the record is to be inserted
    - if there is free space insert there
    - if no free space, in overflow block
    - In either case, point
  - Need to reorganize from time to time sequential order



# Clustering File Organization

Simple file structure stores each relation in a separate file

- Can instead store several relations in one file using a clustering file organization of
- E.g.,  
*CUST*

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

# 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

# Data Dictionary Storage (Cont.)

- Catalog structure: can use either
  - specialized data structures designed for efficient access
  - a set of relations, with existing system features used to ensure efficient access

The latter alternative is usually preferred

- A possible catalog representation:
  - $\text{Relation-metadata} = (\underline{\text{relation-name}}, \underline{\text{number-of-attributes}}, \underline{\text{storage-organization}}, \underline{\text{location}})$
  - $\text{Attribute-metadata} = (\underline{\text{attribute-name}}, \underline{\text{relation-name}}, \underline{\text{domain-type}}, \underline{\text{position}}, \underline{\text{length}})$

$\text{User-metadata} = (\underline{\text{user-name}}, \underline{\text{encrypted-password}}, \underline{\text{group}})$

$\text{Index-metadata} = (\underline{\text{index-name}}, \underline{\text{relation-name}}, \underline{\text{index-type}}, \underline{\text{index-attributes}})$

$\text{View-metadata} = (\underline{\text{view-name}}, \underline{\text{definition}})$

# Mapping of Objects to Files

- Mapping objects to files is similar to mapping tuples to files in a relational system; object data can be stored using file structures.
- Objects in O-O databases may lack uniformity and may be very large; such objects have to managed differently from records in a relational system.
  - Set fields with a small number of elements may be implemented using data structures such as linked lists.
  - Set fields with a larger number of elements

# Mapping of Objects to Files

- Objects are identified by an object identifier (OID); the storage system needs a mechanism to locate an object given its OID (this action is called dereferencing).
  - **logical identifiers** do not directly specify an object's physical location; must maintain an index that maps an OID to the object's actual location.
  - **physical identifiers** encode the location of the object so the object can be found directly. Physical OIDs typically have the following parts:
    1. a volume or file identifier
    2. a page identifier within the volume or

# Management of Persistent

- Physical OIDs may be a unique identifier. This identifier is stored in the object also and is used to

Physical Object Identifier	Location	Unique-Id	Data
Volume.Block.Offset Unique-Id	519.56850.1200	51	.....

Object	Good OID	519.56850.1200	51
Unique-Id Data	Bad OID	519.56850.1200	50

(a) General structure (b) Example of use

# Management of Persistent

- Implement persistent pointers using  
OIDs; persistent pointers are  
substantially longer than are in-  
memory pointers

- Pointer swizzling cuts down on cost  
of locating persistent objects already  
in-memory.
- Software swizzling (swizzling on  
pointer deference)
  - When a persistent pointer is first  
dereferenced, the pointer is **swizzled**  
(replaced by an in-memory pointer) after  
the object is located in memory.

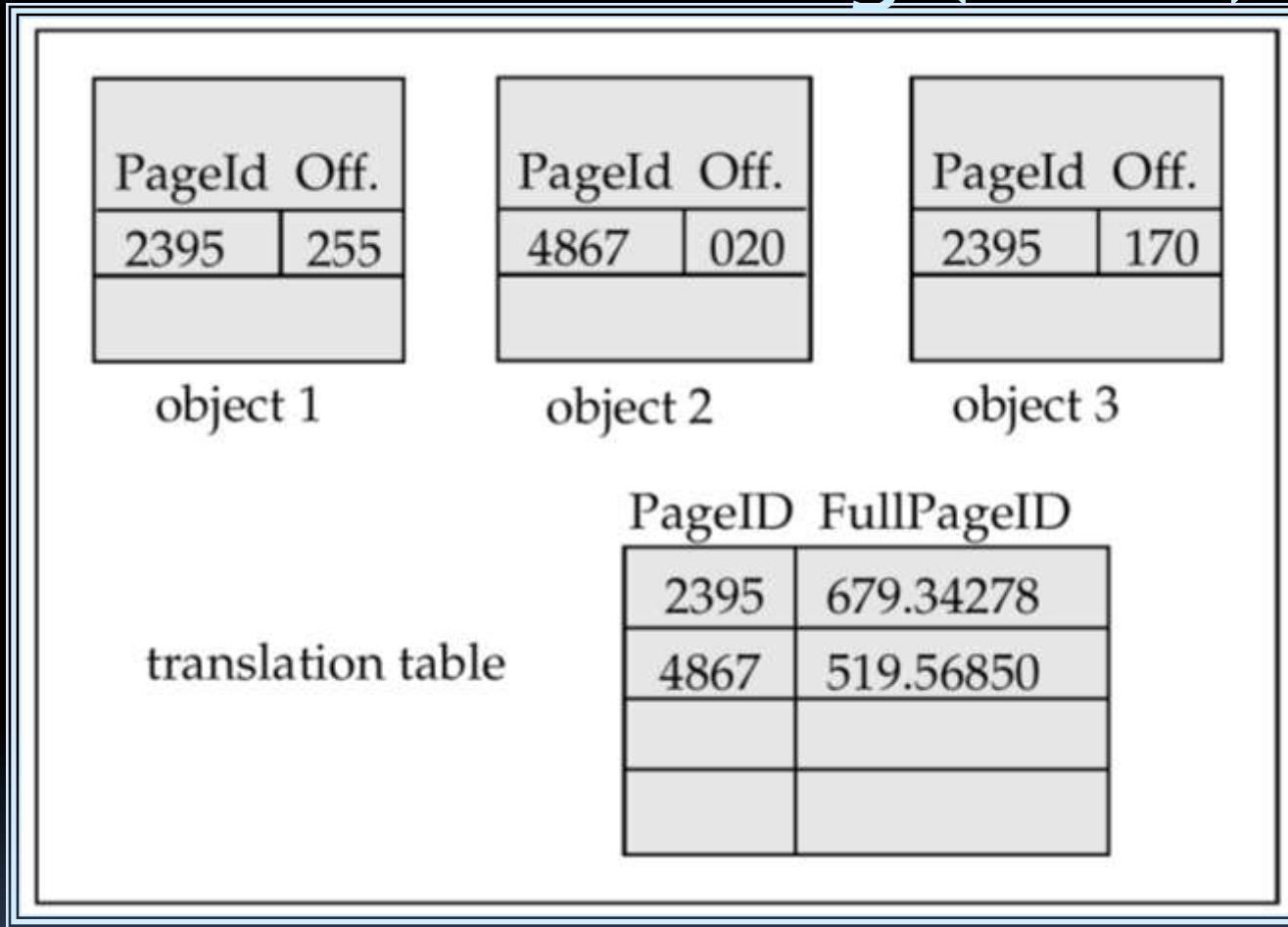
# Hardware Swizzling

- With hardware swizzling, persistent pointers in objects need the same amount of space as in-memory pointers — extra storage external to the object is used to store rest of pointer information.
- Uses virtual memory translation mechanism to efficiently and transparently convert between persistent pointers and in-memory pointers.
- All persistent pointers in a page

# Hardware Swizzling

- Persistent pointer is conceptually split into two parts: a page identifier, and an offset within the page.
  - The page identifier in a pointer is a short indirect pointer: Each page has a translation table that provides a mapping from the short page identifiers to full database page identifiers.
  - Translation table for a page is small (at most 1024 pointers in a 4096 byte page with 4 byte pointer)
  - Multiple pointers in page to the same

# Hardware Swizzling (Cont.)



- Page image before swizzling  
(page located on disk)

# Hardware Swizzling (Cont.)

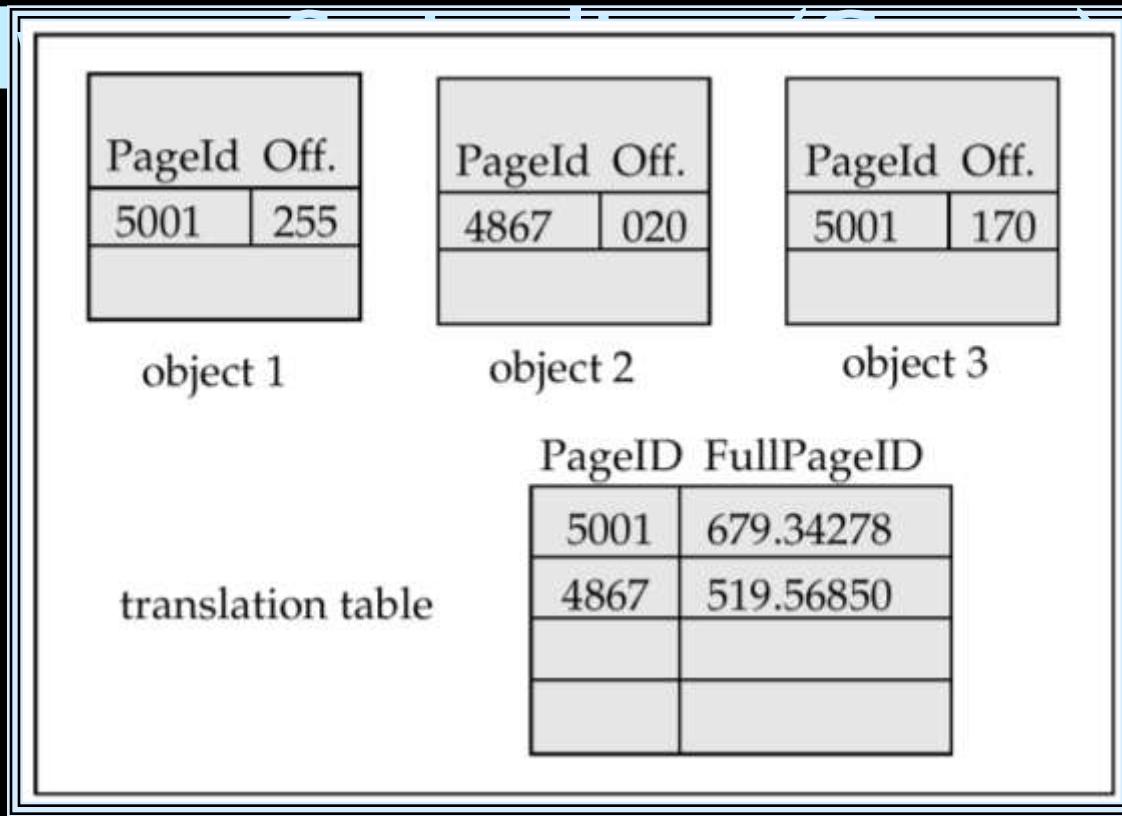
- When system loads a page into memory the persistent pointers in the page are *swizzled* as described below
  - Persistent pointers in each object in the page are located using object type information
  - For each persistent pointer ( $p_i, o_i$ ) find its full page ID  $P_i$ 
    - If  $P_i$  does not already have a virtual memory page allocated to it, allocate a virtual memory page to  $P_i$  and read-protect the page
      - Note: there need not be any physical space (whether in memory or on disk swap-space) allocated for the virtual memory page at this point. Space can be allocated later if (and when)  $P_i$  is accessed. In this case read

# Hardware Swizzling (Cont.)

When an in-memory pointer is dereferenced, if the operating system detects the page it points to has not yet been allocated storage, or is read-protected, a segmentation violation occurs.

- The `mmap()` call in Unix is used to specify a function to be invoked on segmentation violation
- The function does the following when it is invoked
  1. Allocate storage (swap-space) for the page containing the referenced

# Hard



## Page image after swizzling

- Page with short page identifier 2395 was allocated address 5001. Observe change in pointers and translation table.

# Hardware Swizzling (Cont.)

- After swizzling, all short page identifiers point to virtual memory addresses allocated for the corresponding pages
  - functions accessing the objects are not even aware that it has persistent pointers, and do not need to be changed in any way!
  - can reuse existing code and libraries that use in-memory pointers
- After this, the pointer dereference that triggered the swizzling can continue
- Optimizations:
  - If all pages are allocated the same address as in the short page identifier, no changes in object code are required.

# Disk versus Memory Structure of Objects

- The format in which objects are stored in memory may be different from the formal in which they are stored on disk in the database. Reasons are:
  - software swizzling – structure of persistent and in-memory pointers are different
  - database accessible from different machines, with different data representations
  - Make the physical representation of objects in the database independent of the machine and the compiler.
  - Can transparently convert from disk

# Large Objects

- Large objects : binary large objects (blobs) and character large objects (clobs)
  - Examples include:
    - text documents
    - graphical data such as images and computer aided designs audio and video data
- Large objects may need to be stored in a contiguous sequence of bytes when brought into memory.

# Modifying Large Objects

- If the application requires insert/delete of bytes from specified regions of an object:
  - B<sup>+</sup>-tree file organization (described later in Chapter 12) can be modified to represent large objects
  - Each leaf page of the tree stores between half and 1 page worth of data from the object
- Special-purpose application programs outside the database are used to manipulate large objects:
  - Text data treated as a byte string

END OF CHAPTER



# File Containing *account*

Records

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

# File of Figure 11.6, with Record 2 Deleted

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

# File of Figure 11.6, With Record 2 deleted and Final Record Moved

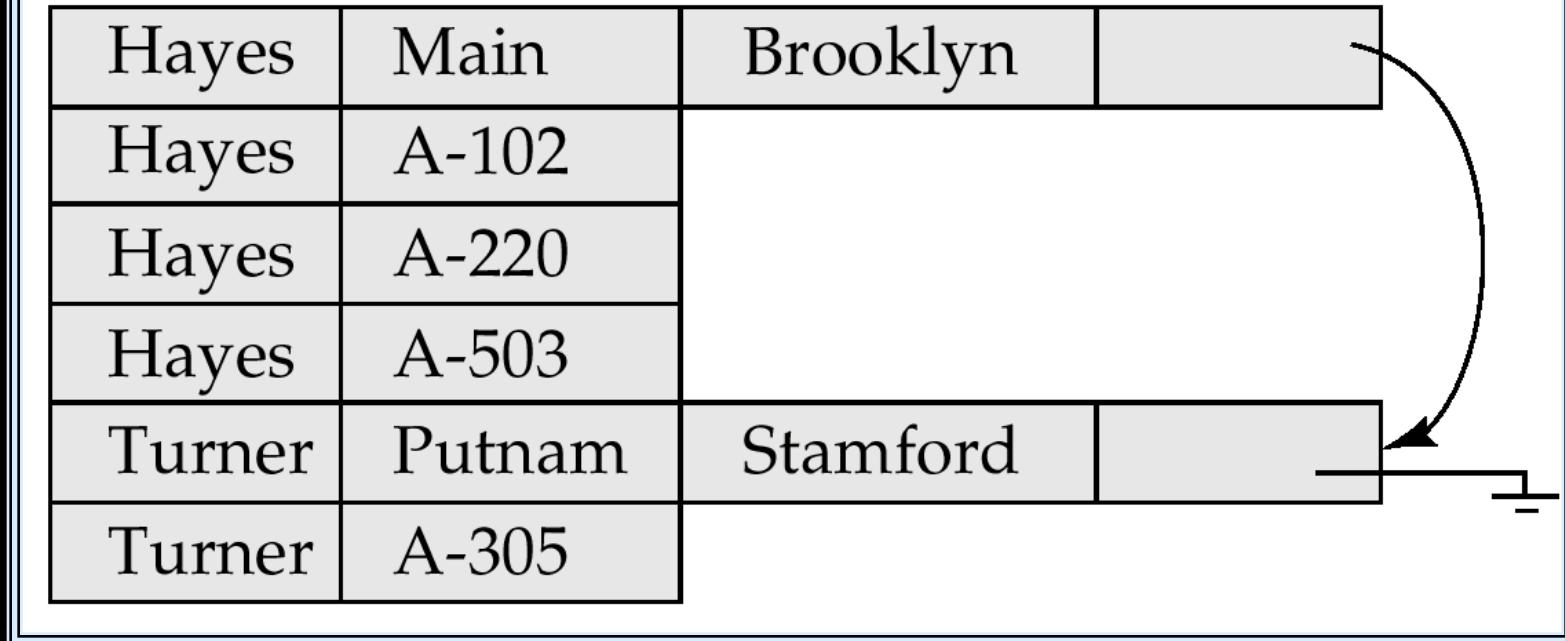
record 0	A-102	Perryridge	400
record 1	A-305	Round Hill	350
record 8	A-218	Perryridge	700
record 3	A-101	Downtown	500
record 4	A-222	Redwood	700
record 5	A-201	Perryridge	900
record 6	A-217	Brighton	750
record 7	A-110	Downtown	600

# Byte-String Representation of Variable-Length Records

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

# Clustering File Structure

# Clustering File Structure With Pointer Chaining



# The *depositor* Relation

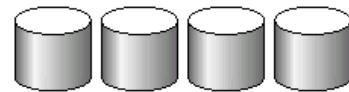
<i>customer-name</i>	<i>account-number</i>
Hayes	A-102
Hayes	A-220
Hayes	A-503
Turner	A-305

# The *customer* Relation

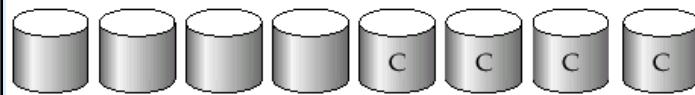
<i>customer-name</i>	<i>customer-street</i>	<i>customer-city</i>
Hayes	Main	Brooklyn
Turner	Putnam	Stamford

# Clustering File Structure

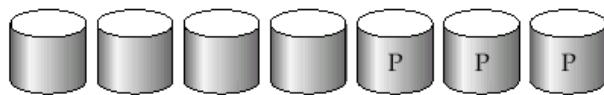
Hayes	Main	Brooklyn
Hayes	A-102	
Hayes	A-220	
Hayes	A-503	
Turner	Putnam	Stamford
Turner	A-305	



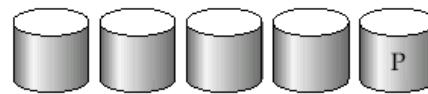
(a) RAID 0: nonredundant striping



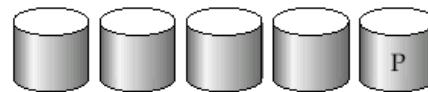
(b) RAID 1: mirrored disks



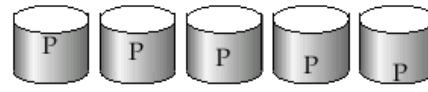
(c) RAID 2: memory-style error-correcting codes



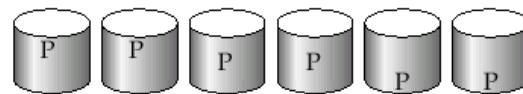
(d) RAID 3: bit-interleaved parity



(e) RAID 4: block-interleaved parity



(f) RAID 5: block-interleaved distributed parity



(g) RAID 6: P + Q redundancy

# Chapter 12: Indexing and Hashing

- Basic Concepts
- Ordered Indices
- B+-Tree Index Files
- B-Tree Index Files
- Static Hashing
- Dynamic Hashing

Organization-Indexing & Hashing-B+ TREE-B Tree-Static Hashing-Dynamic Hashing-Multiple Key Access

- 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.
  - search-key      pointer
- An index file consists of records (called index entries) of the form
- Index files are typically much smaller than the original file

# 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

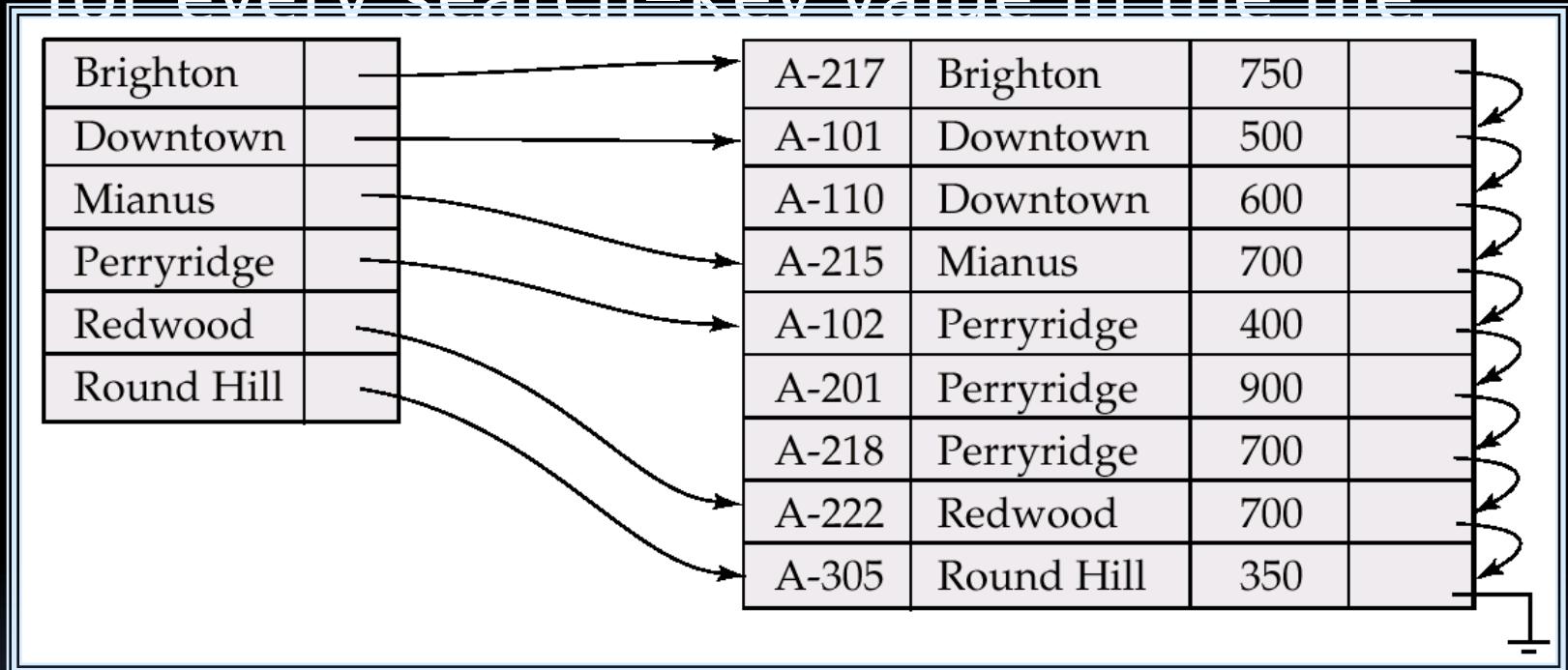
# Ordered Indices

Indexing techniques evaluated on basis of:

- 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

# Dense Index Files

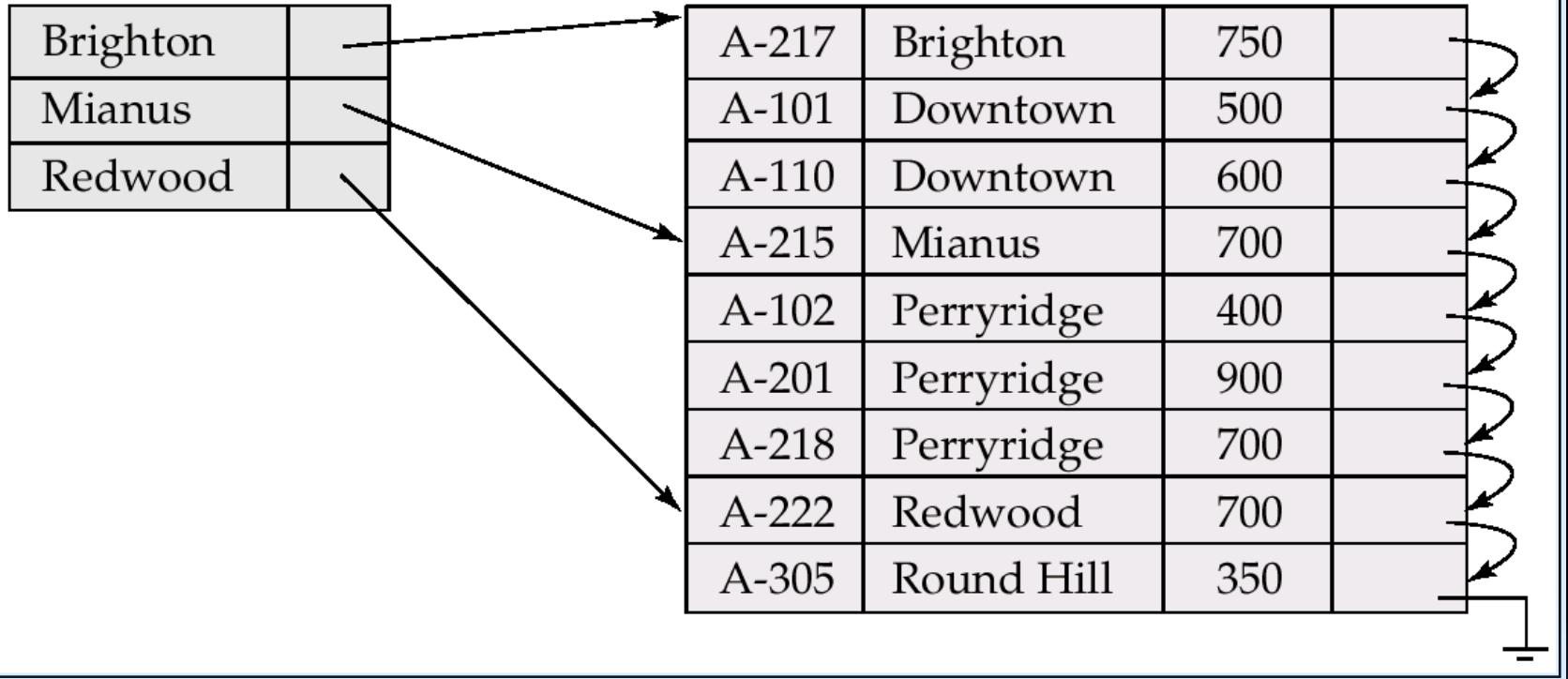
- Dense index — Index record appears for every search-key value in the file.



# 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
- Less space and less maintenance

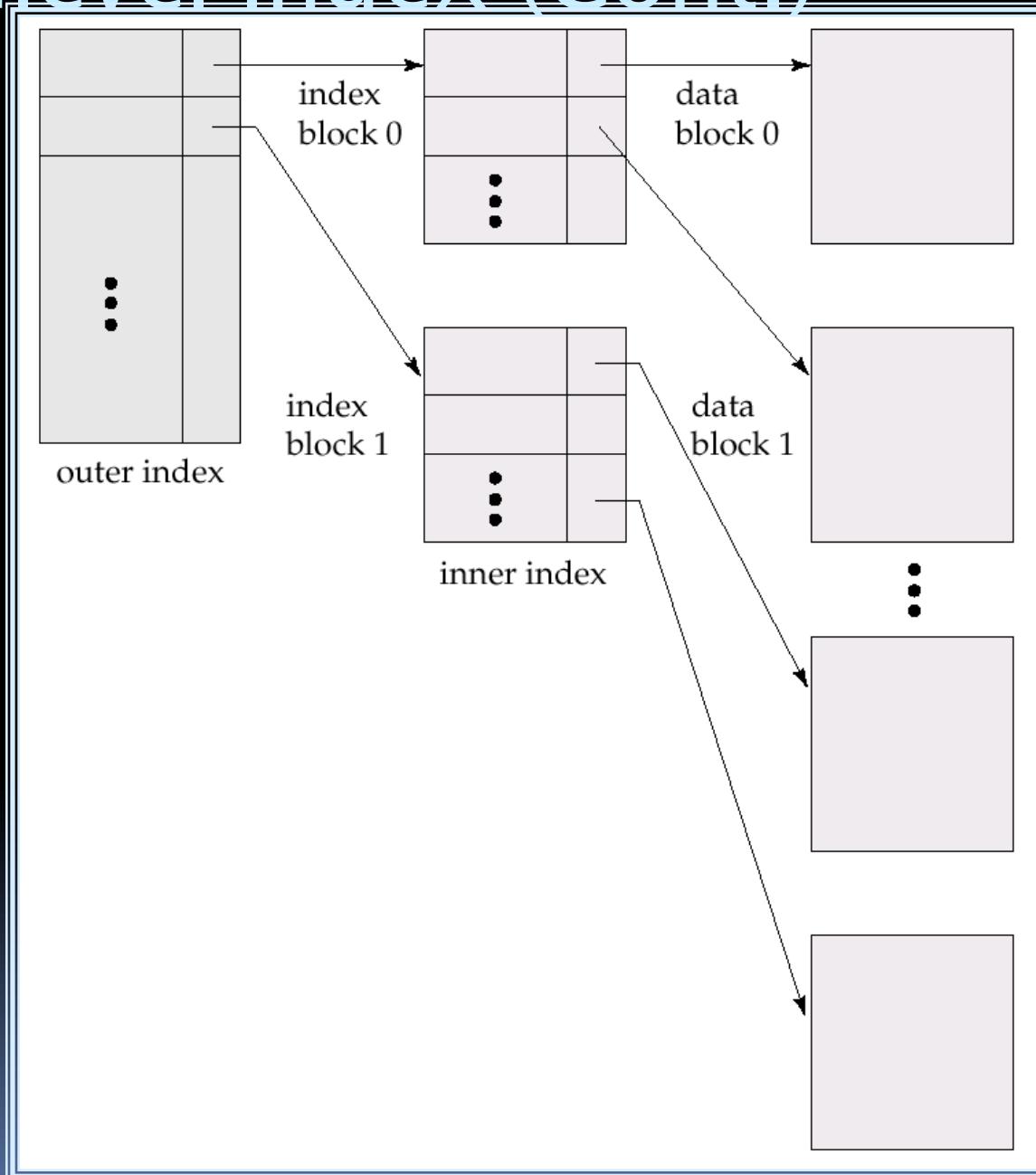
# Example of Sparse Index Files



# Multilevel Index

- If primary index does not fit in memory, access becomes expensive.
- To reduce number of disk accesses to index records, 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

# 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 is 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

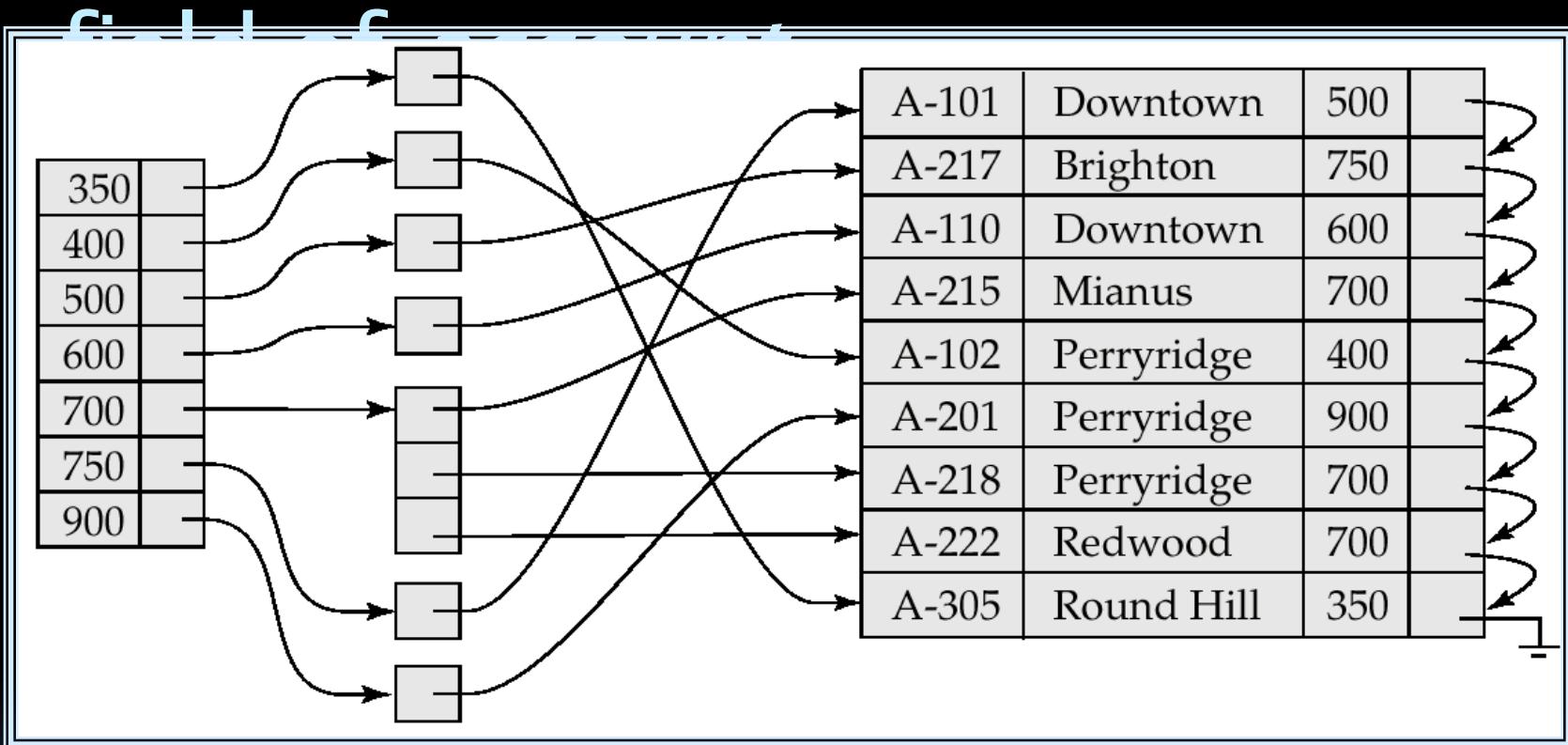
# Index Update: Insertion

- 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. In this case, the first search-key value appearing in the new block is inserted into the index.

# 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* database 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

# Secondary Index on *balance*



# Primary and Secondary Indices

- Secondary indices have to be dense.
- Indices offer substantial benefits when searching for records.
- When a file is modified, every index on the file must be updated, Updating indices imposes overhead on database modification.
- Sequential scan using primary index is efficient, but a sequential scan using a secondary index is expensive

# 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.

# 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  $[n/2]$  and  $n$  children.
- A leaf node has between  $[(n-1)/2]$  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 children), it has at least one value.

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



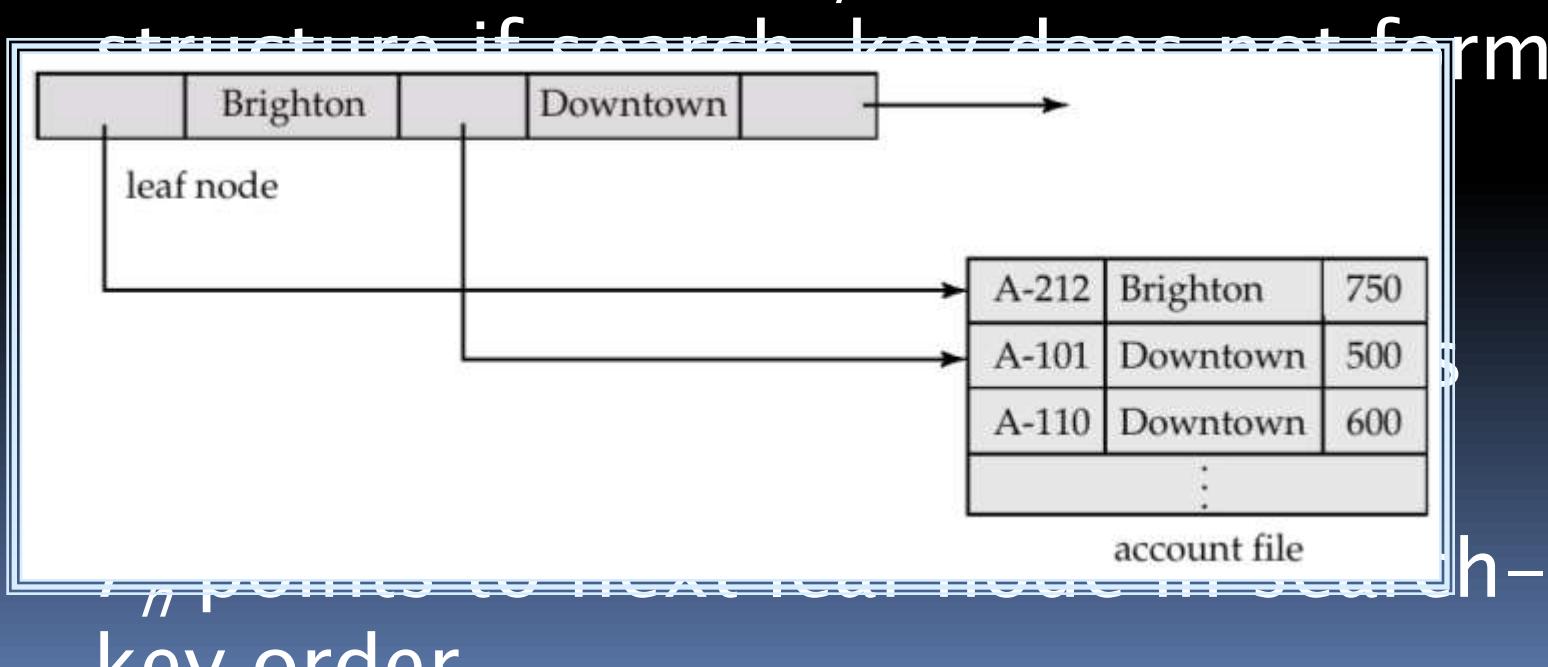
- $K_i$  are the search-key values
- $P_i$  are pointers to children (for non-leaf nodes) or pointers to records or buckets of records (for leaf nodes).
- The search-keys in a node are ordered

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

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

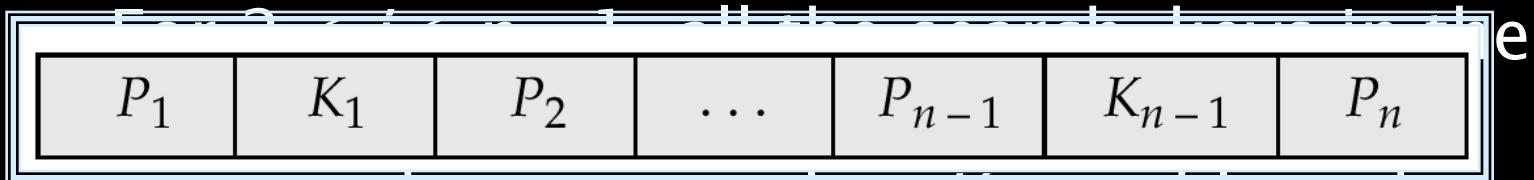
Properties of a leaf node:

- 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



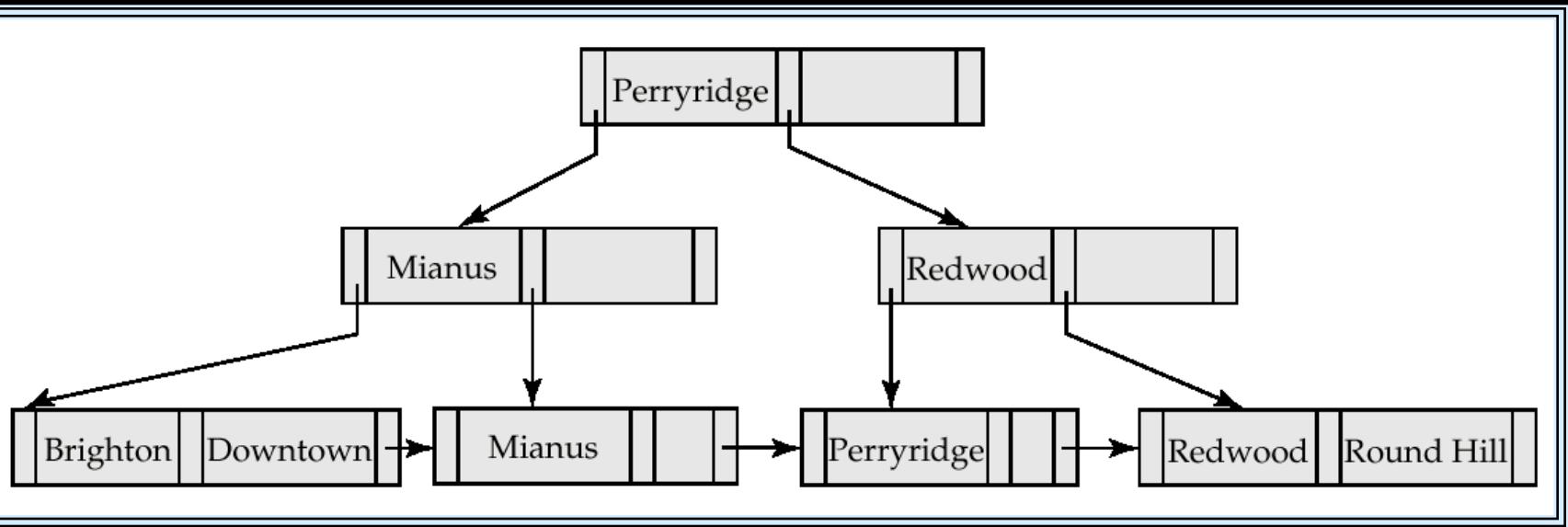
# Non-Leaf Nodes in B<sup>+</sup>-Trees

- 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  $K_1$



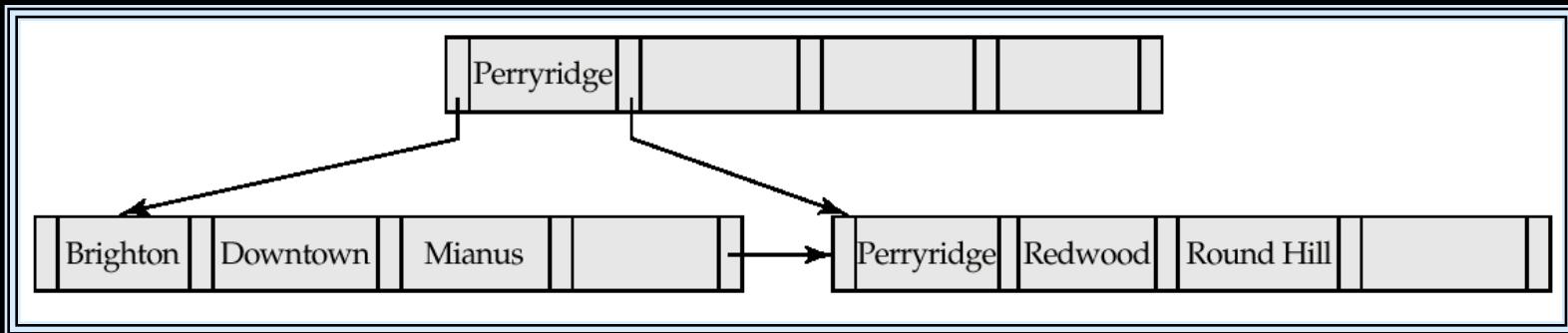
greater than or equal to  $K_{i-1}$  and less than  $K_{m-1}$

# 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  $n$

# 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 (logarithmic in the size of the main file), thus searches can be conducted efficiently.
- Insertions and deletions to the main file can be handled efficiently, as the

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

- Find all records with a search-key value of  $k$ .
  1. Start with the root node
    1. Examine the node for the smallest search-key value  $> k$ .
    2. If such a value exists, assume it is  $K_j$ . Then follow  $P_j$  to the child node
    3. Otherwise  $k \geq K_{m-1}$ , where there are  $m$  pointers in the node. Then follow  $P_m$  to the child node.
  2. If the node reached by following the pointer above is not a leaf node, repeat the above procedure on the node, and follow the corresponding pointer.

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

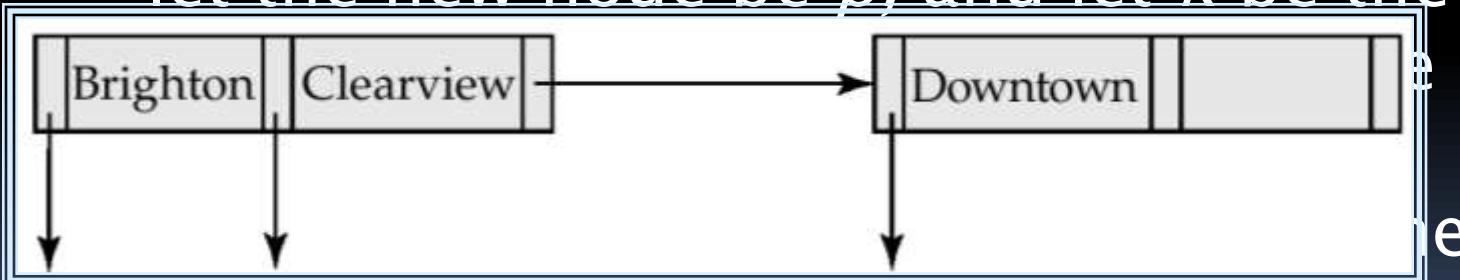
- In processing a query, a path is traversed in the tree from the root to some leaf node.
- If there are  $K$  search-key values in the file, the path is no longer than  $\lceil \log_{\lceil n/2 \rceil}(K) \rceil$ .
- A node is generally the same size as a disk block, typically 4 kilobytes, and  $n$  is typically around 100 (40 bytes per index entry).
- With 1 million search key values

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

- Find the leaf node in which the search-key value would appear
- If the search-key value is already there in the leaf node, record is added to file and if necessary a pointer is inserted into the bucket.
- If the search-key value is not there, then add the record to the main file and create a bucket if necessary.  
Then:
  - If there is room in the leaf node, insert

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

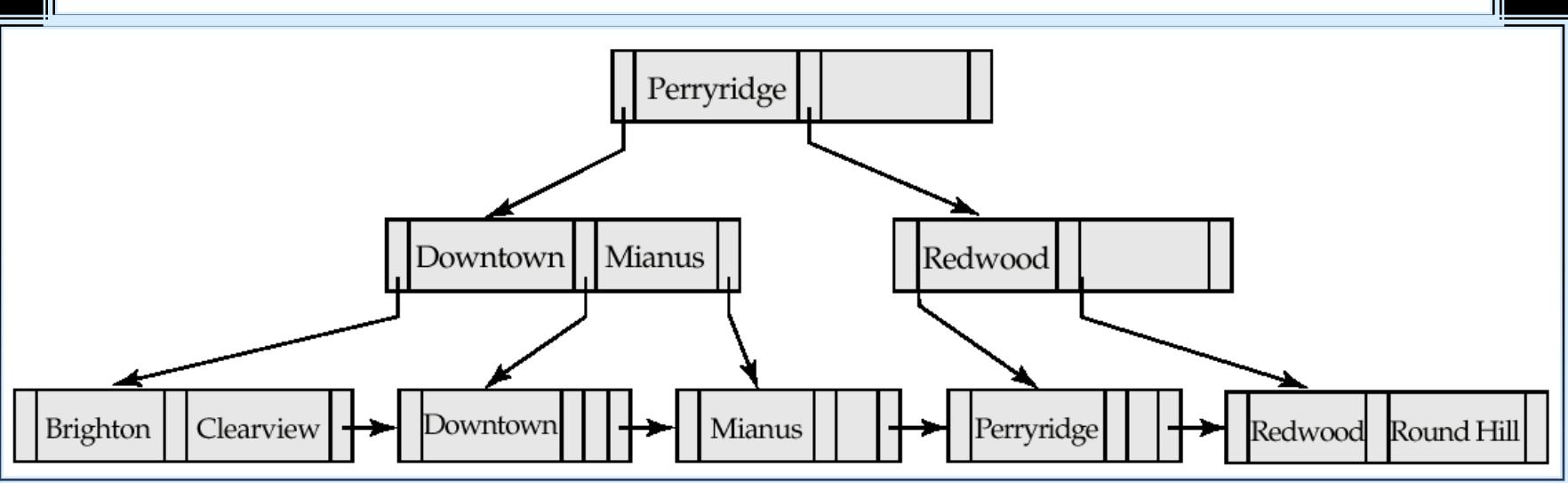
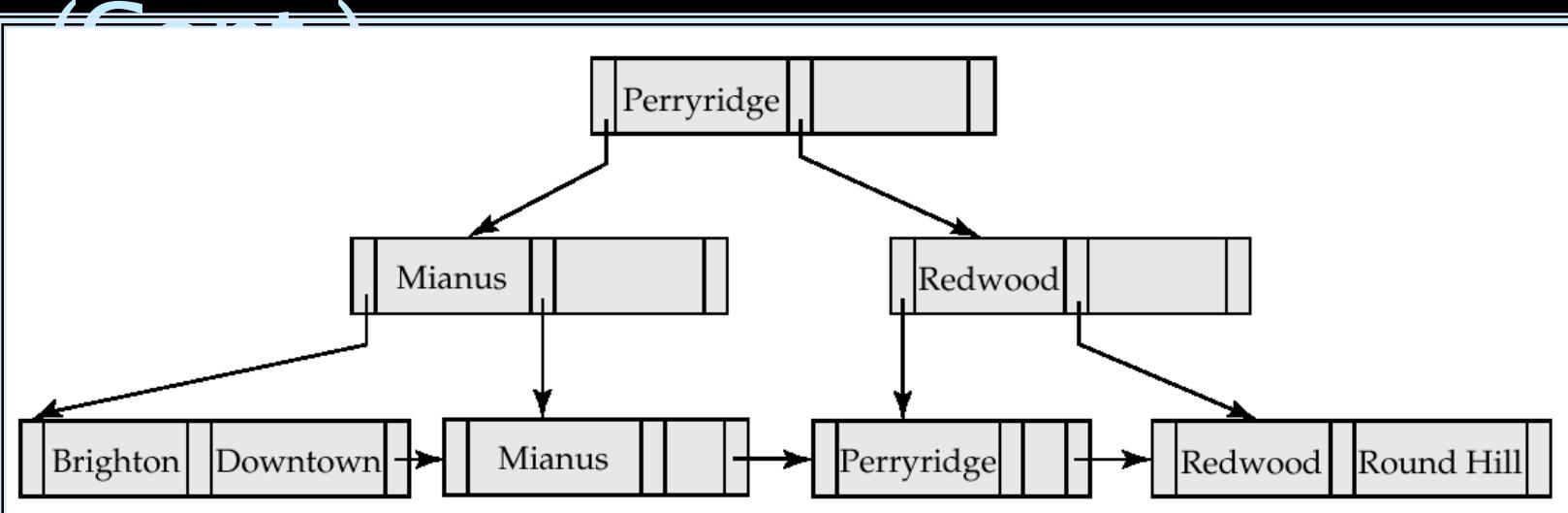
- Splitting a node:
  - take the  $n$ (search-key value, pointer) pairs (including the one being inserted) in sorted order. Place the first  $\lceil n/2 \rceil$  in the original node, and the rest in a new node.
  - let the new node be  $p$ , and let  $k$  be the



Result of splitting node containing Brighton and Downtown on inserting Clearview

- The splitting of nodes proceeds upwards till a node that is not full is

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



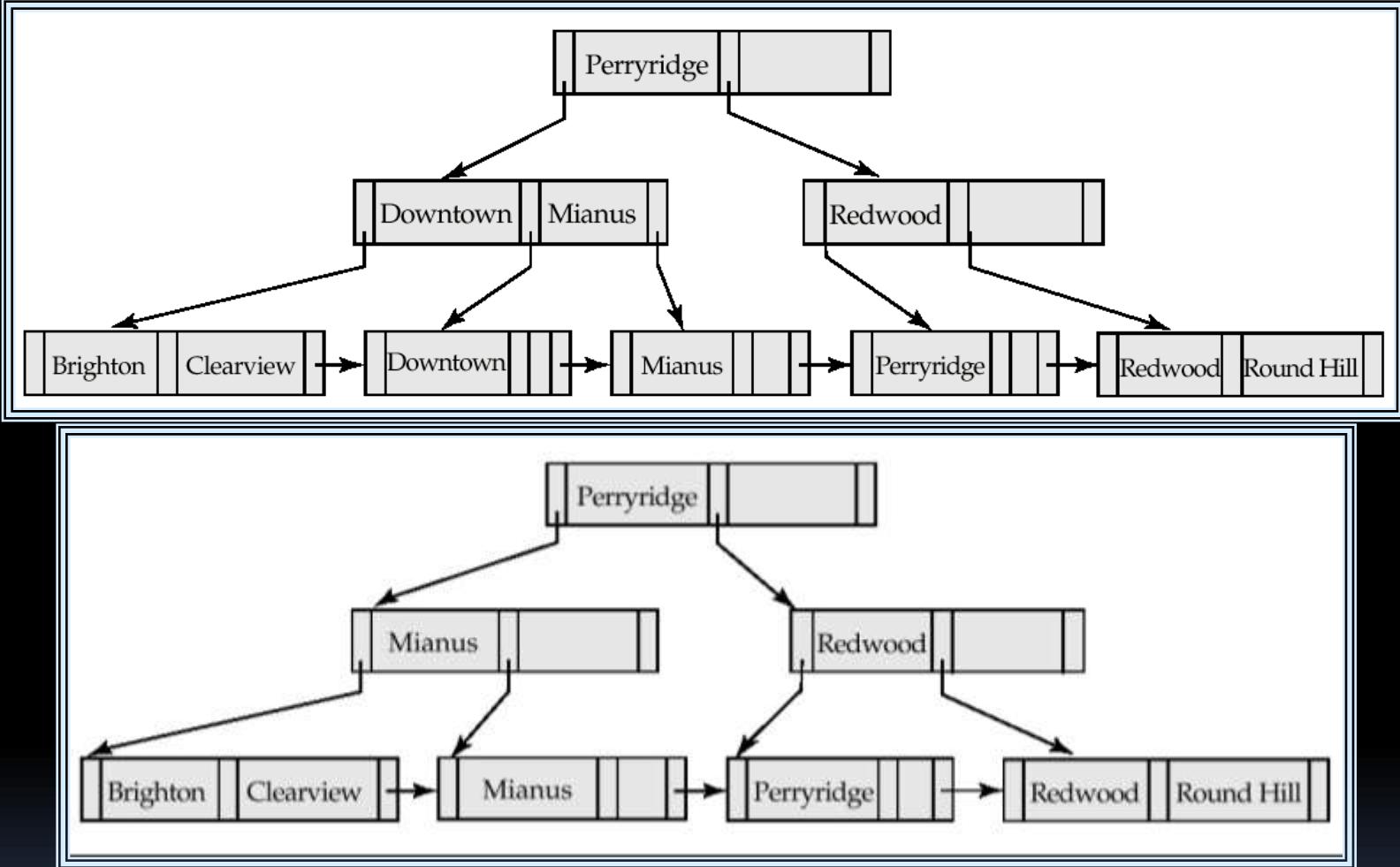
B<sup>+</sup>-Tree before and after insertion of “Clearview”

# 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
  - Insert all the search-key values in the node into a single node.

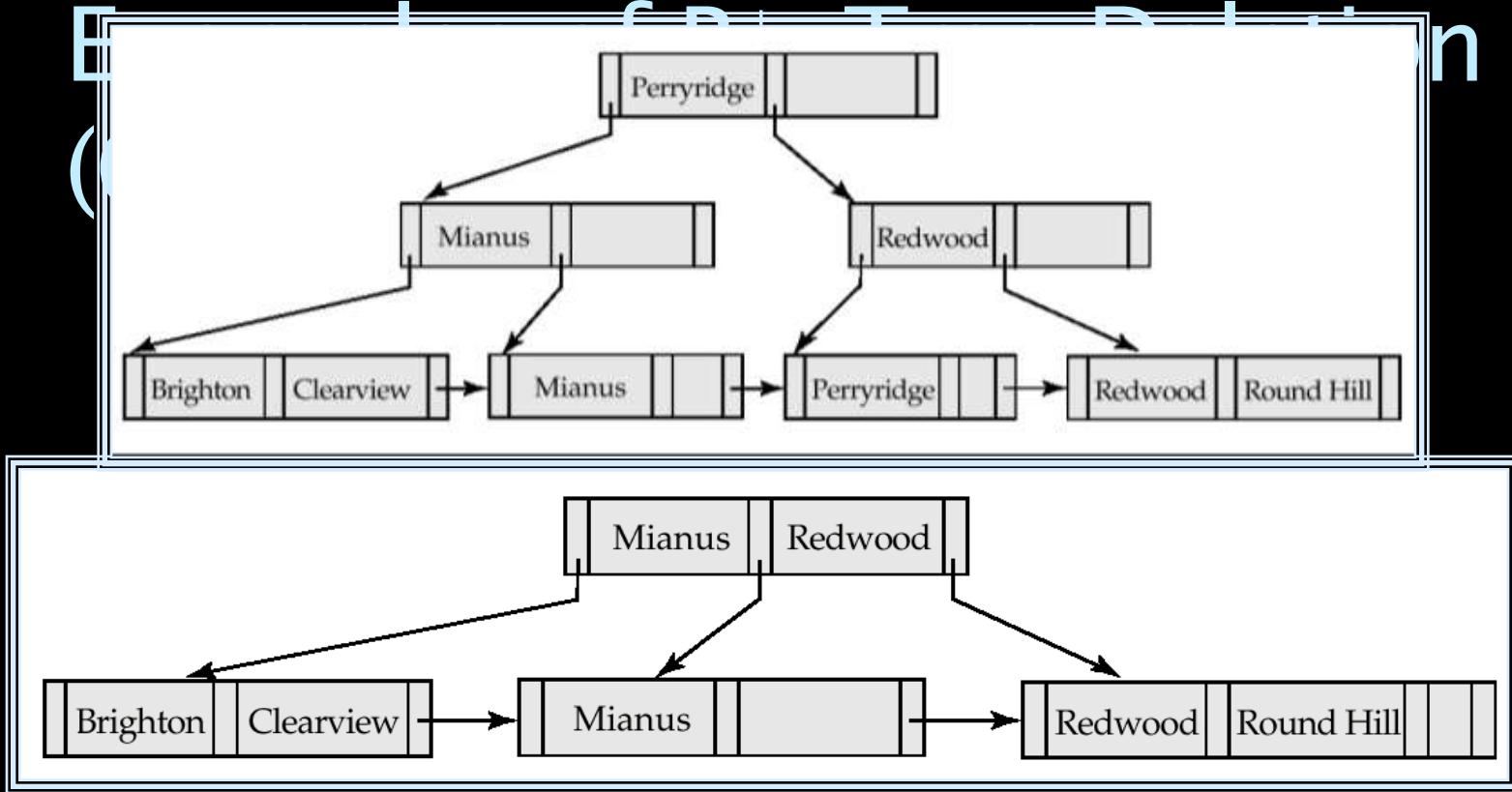
# Updates on B<sup>+</sup>-Trees: Deletion

- Otherwise, 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
  - 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



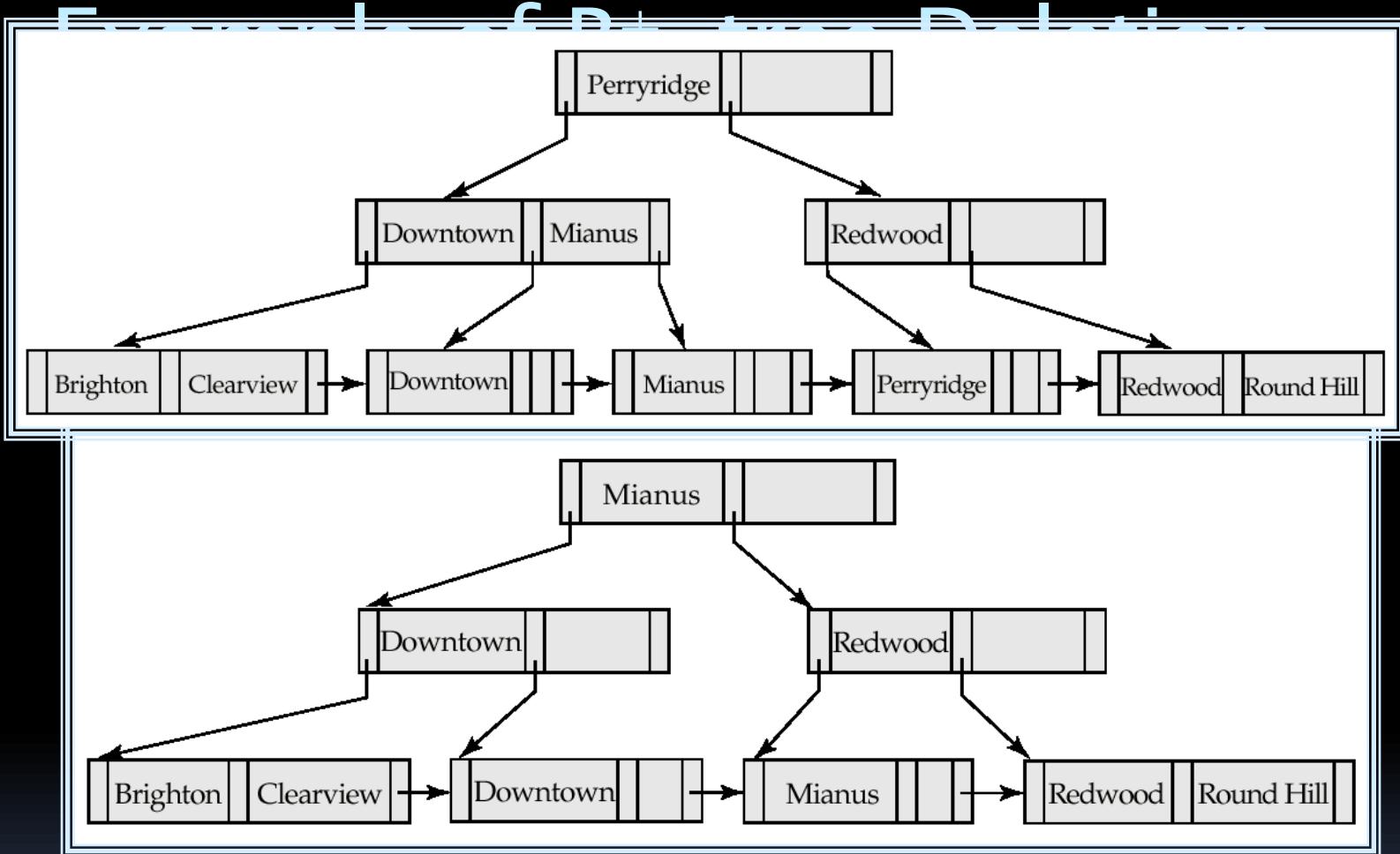
Before and after deleting “Downtown”

The removal of the leaf node containing “Downtown” did not result in its parent having too little pointers. So the cascaded deletions stopped with the deleted leaf node’s parent.



Deletion of “Perryridge” from result of previous example

- Node with “Perryridge” becomes underfull (actually empty, in this special case) and merged with its sibling.



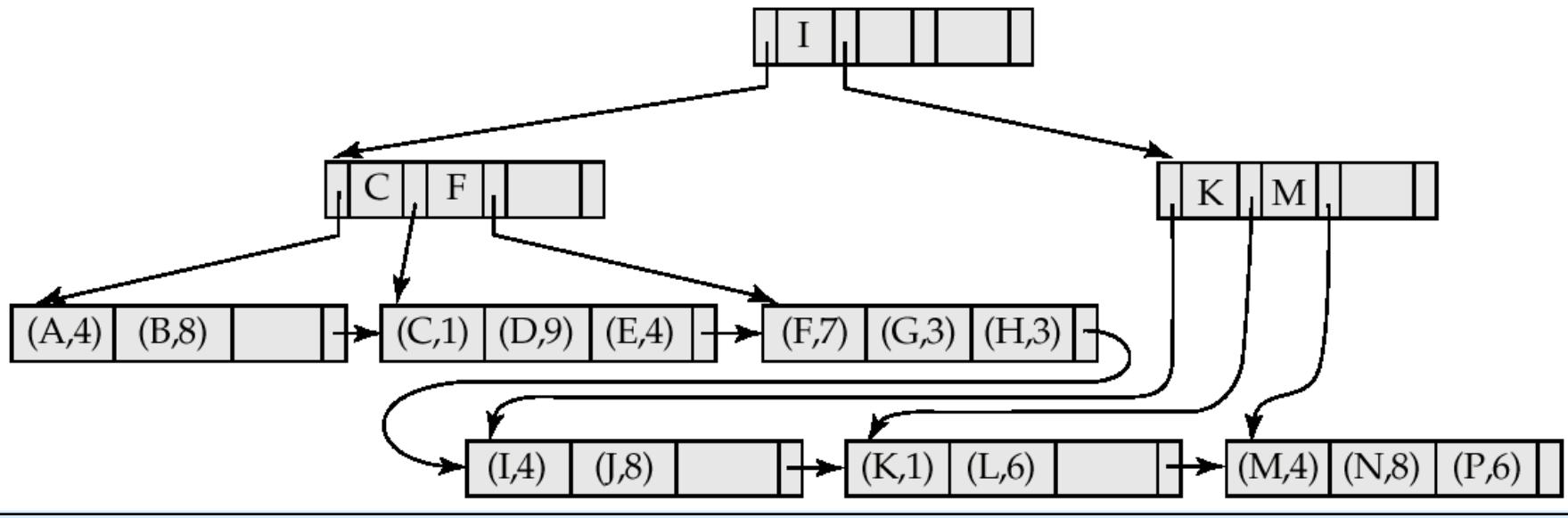
Before and after deletion of “Perryridge” from earlier example

- 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.
- 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

# R+ - Tree File Organization



Example of B+-tree File Organization

Good space utilization important since records use more space than pointers.

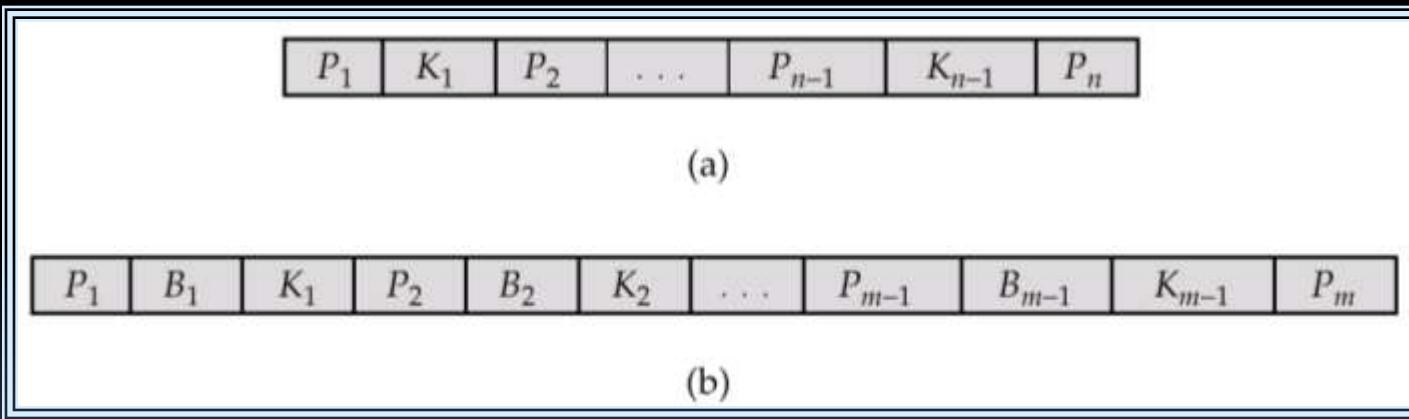
To improve space utilization, involve more sibling nodes in redistribution during splits and merges

- Involving 2 siblings in redistribution (to avoid split / merge where possible) results in each node having at least  $\lfloor 2n/3 \rfloor$  entries

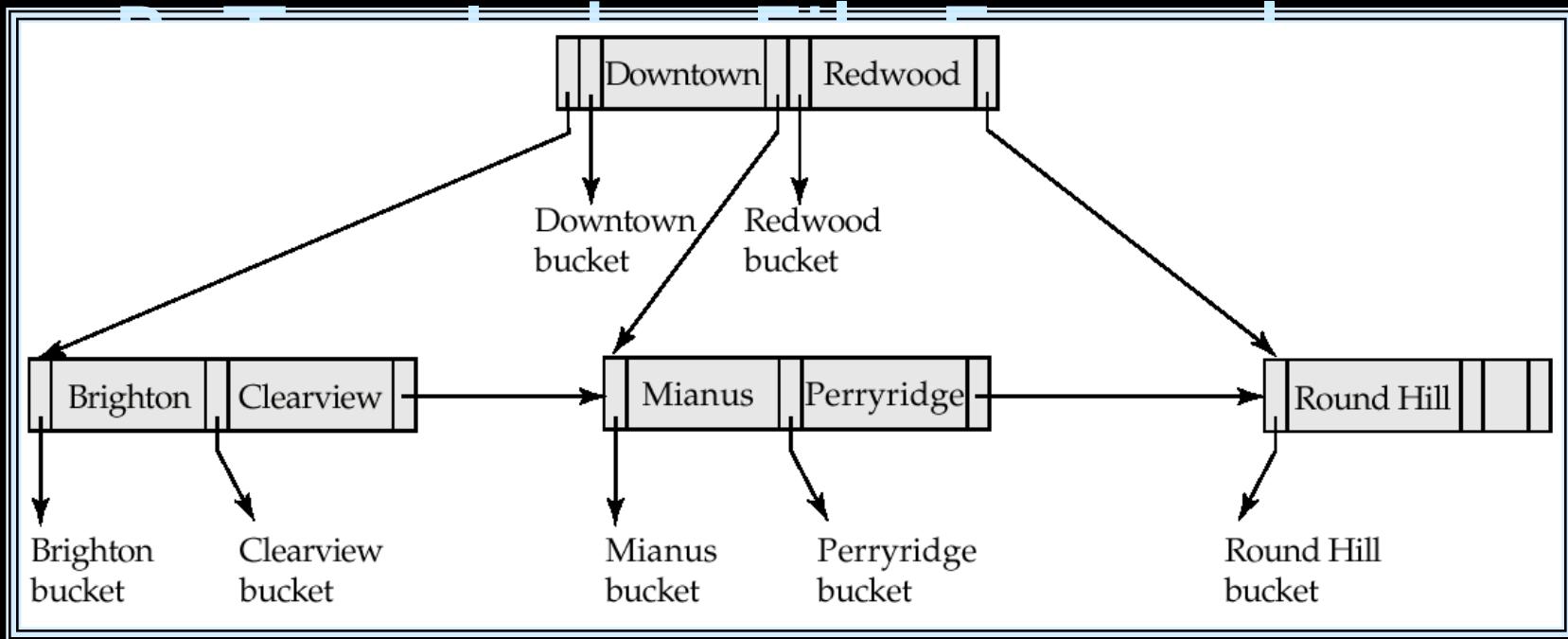
$$\lfloor 2n/3 \rfloor$$

# B-Tree Index Files

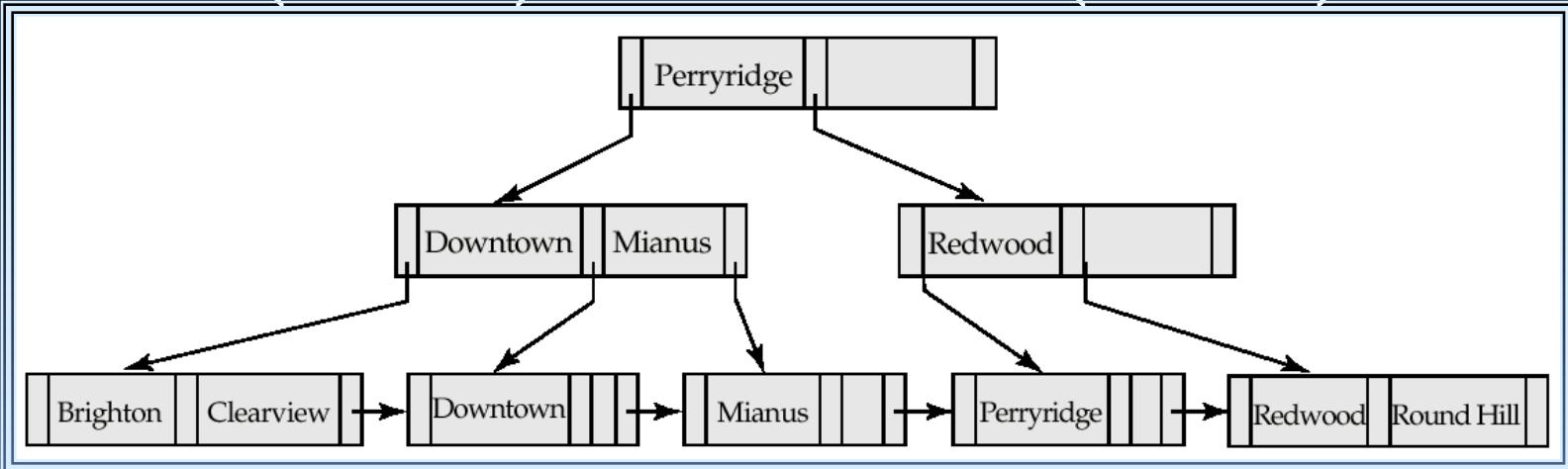
- Similar to B+-tree, but B-tree allows search-key values to appear only once; eliminates redundant storage of search keys.
- Search keys in nonleaf nodes appear nowhere else in the B-tree; an additional pointer field for each search key in a nonleaf node must be included.
- Generalized B-tree leaf node



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



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



## 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

# 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

# Example of Hash File Organization (Cont.)

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.  $h(\text{Perryridge}) = 5 \quad h(\text{Round Hill}) = 3$   
 $h(\text{Brighton}) = 3$

Hash file organization of account file, using branch-name as key  
(see previous slide for details).

# Example of Hash File Organization

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

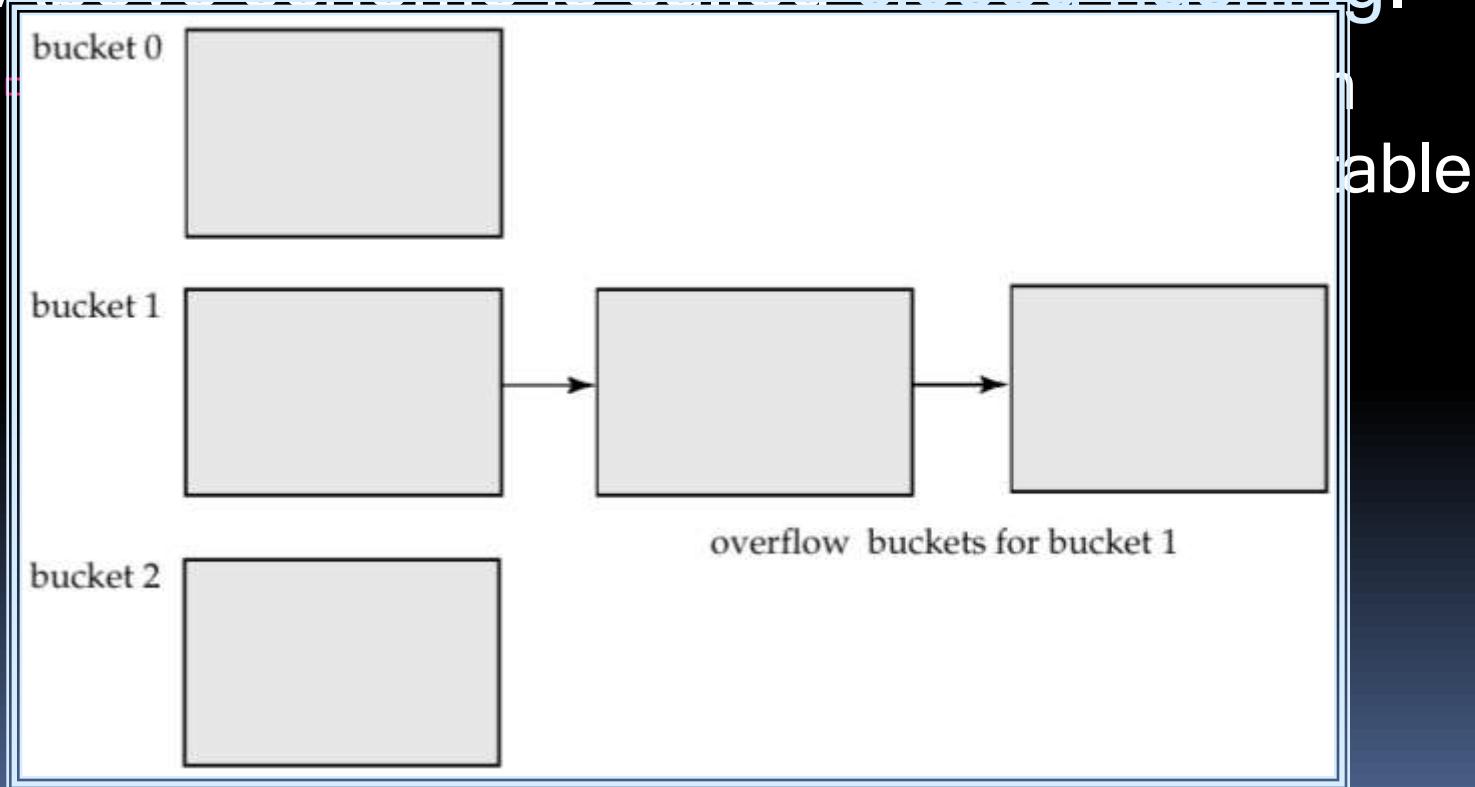
- Worst hash function maps all search-key values to the same bucket; this makes access time proportional to the number of search-key values in the file.
- An ideal hash function is uniform, i.e., each bucket is assigned the same number of search-key values from the set of all possible values.
- Ideal 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.

# 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.

# Handling of Bucket Overflows (Cont.)

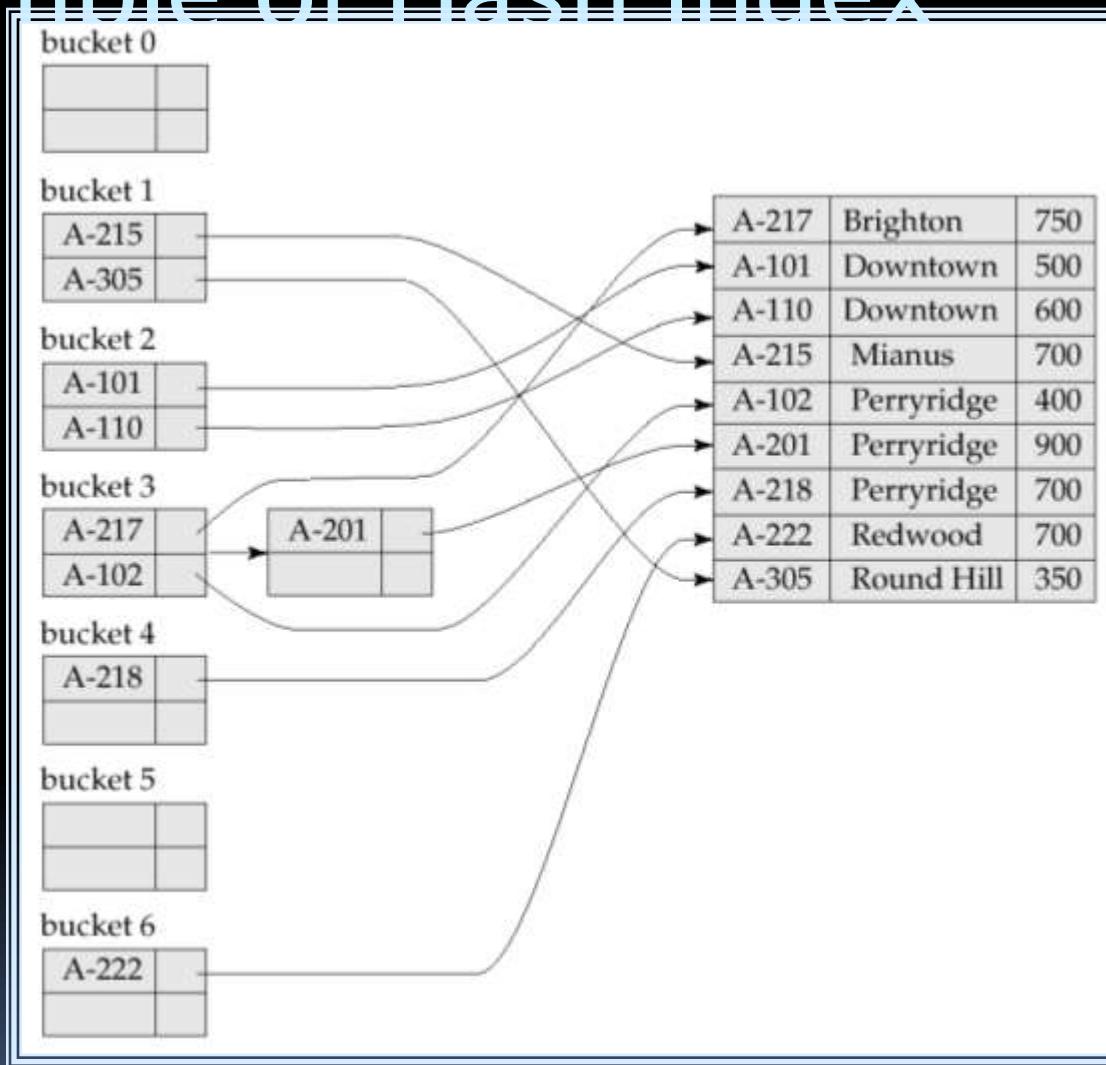
- Overflow chaining - the overflow buckets of a given bucket are chained together in a linked list.
- Above scheme is called closed hashing.



# 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.

# Example of Hash Index



# Deficiencies of Static Hashing

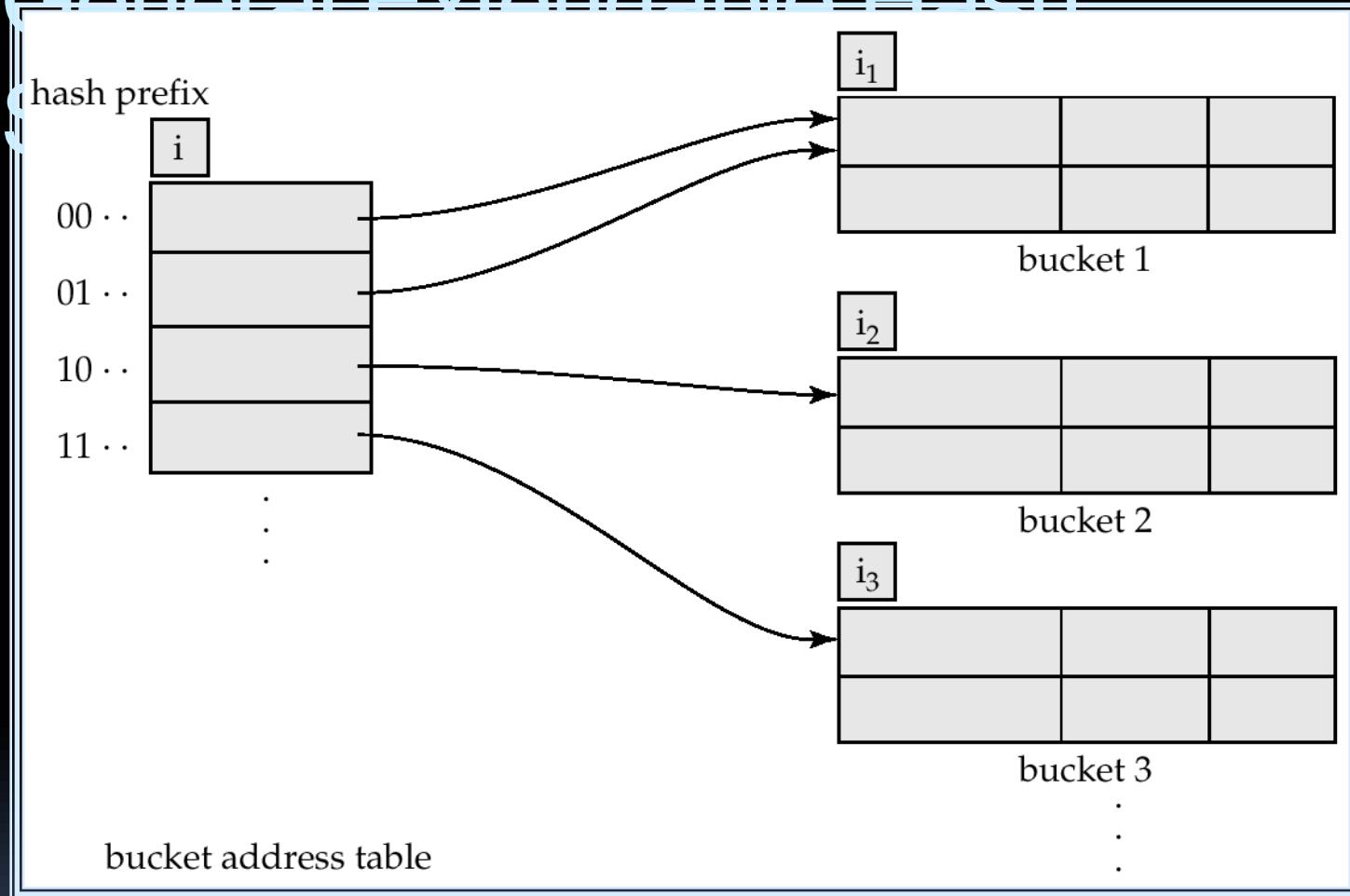
- In static hashing, function  $h$  maps search-key values to a fixed set of  $B$  of bucket addresses.
  - Databases grow with time. If initial number of buckets is too small, performance will degrade due to too much overflows.
  - If file size at some point in the future is anticipated and number of buckets allocated accordingly, significant amount of space will be wasted initially.
  - If database shrinks, again space will be wasted.
  - One option is periodic re-organization of the file with a new hash function, but it is very

# 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

# General Extendable Hash



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  $l_j$ ; 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

# Updates in Extendable Hash

To split a bucket  $j$  when inserting record with search-key value  $K_j$ :

- **Structure (more than one pointer to bucket  $j$ )**

- allocate a new bucket  $z$ , and set  $i_j$  and  $i_z$  to the old  $i_j -+ 1$ .
  - make the second half of the bucket address table entries pointing to  $j$  to point to  $z$
  - remove and reinsert each record in bucket  $j$ .
  - 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  $j$ )

- increment  $i$  and double the size of the bucket address table.
  - replace each entry in the table by two entries

# Updates in Extendable Hash

## Structure (Cont.)

When inserting a value, if the bucket is full after several splits (that is,  $i$  reaches some limit  $b$ ) create an overflow bucket instead of splitting bucket entry table further.

- 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

# Use of Extendable Hash Structure:

Ex

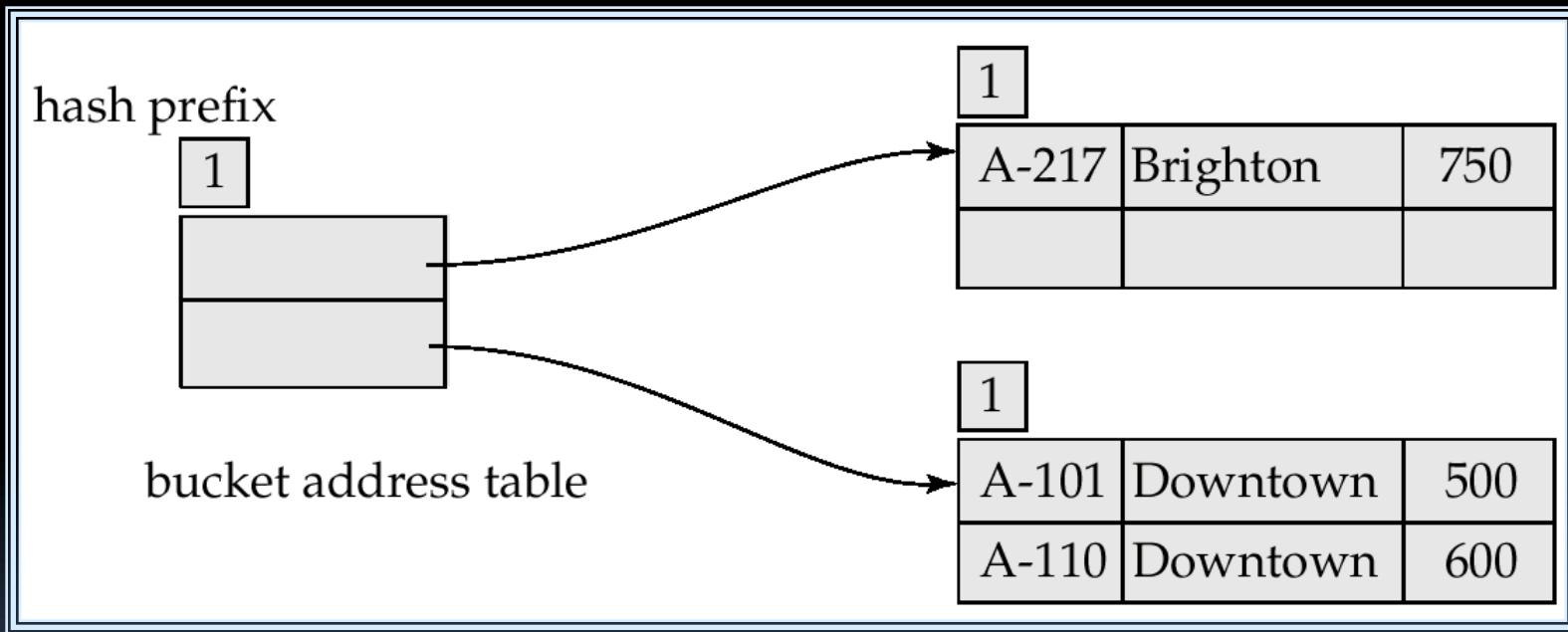
<i>branch-name</i>	$h(\text{branch-name})$
Brighton	0010 1101 1111 1011 0010 1100 0011 0000
Downtown	1010 0011 1010 0000 1100 0110 1001 1111
Mianus	1100 0111 1110 1101 1011 1111 0011 1010
Perryridge	1111 0001 0010 0100 1001 0011 0110 1101
Redwood	0011 0101 1010 0110 1100 1001 1110 1011
Round Hill	1101 1000 0011 1111 1001 1100 0000 0001



Initial Hash structure, bucket size = 2

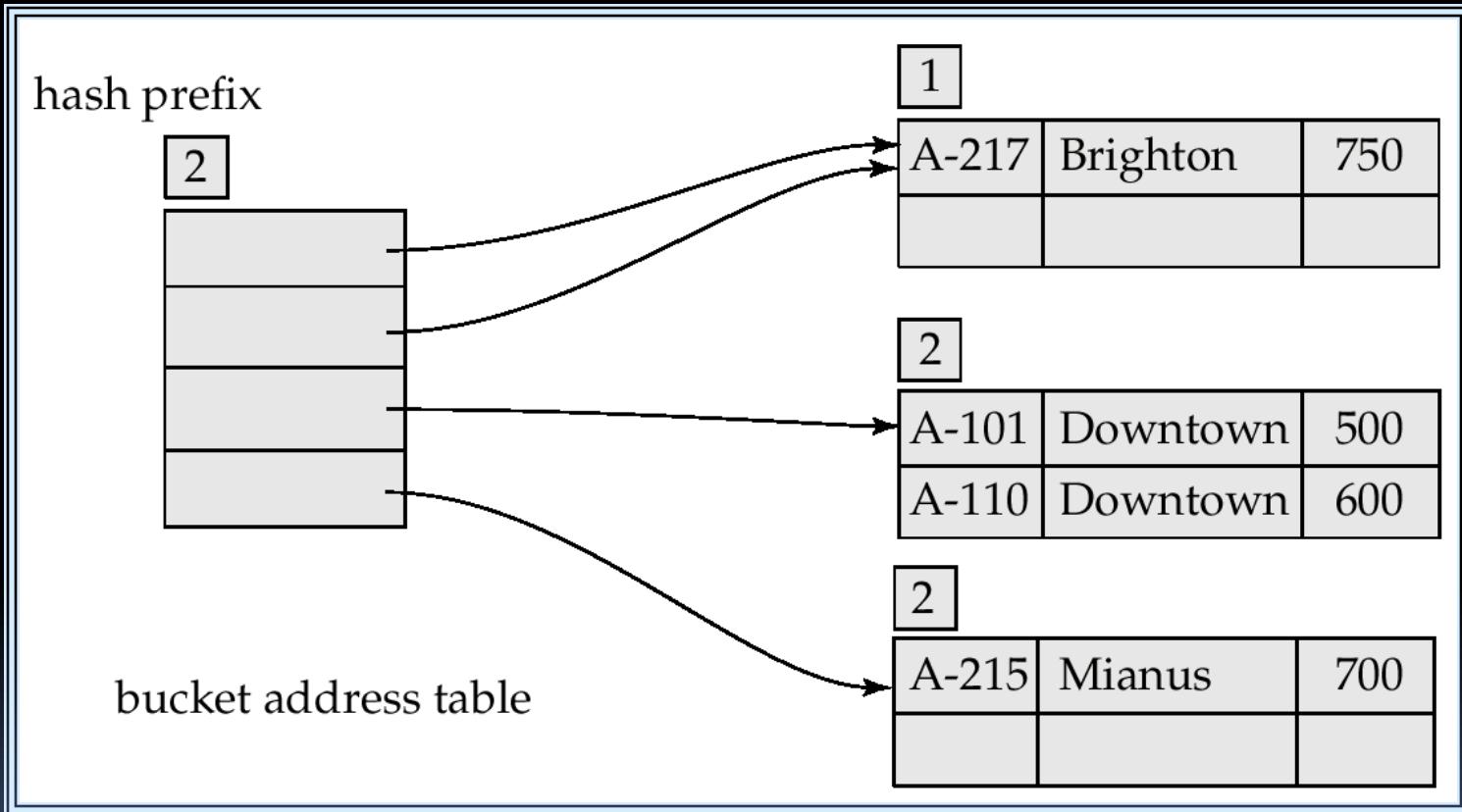
# Example (Cont.)

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

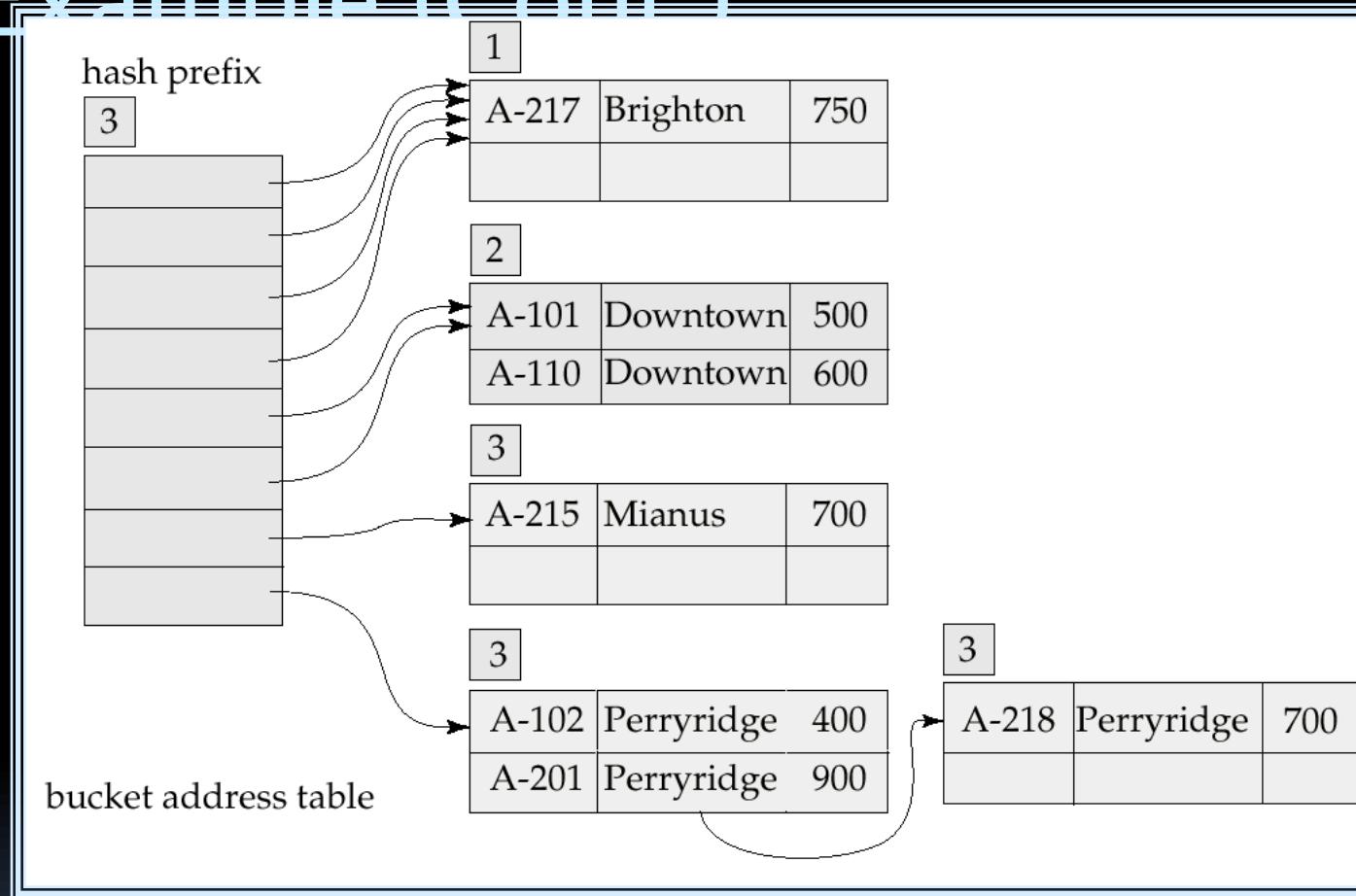


# Example (Cont.)

Hash structure after insertion of Mianus record

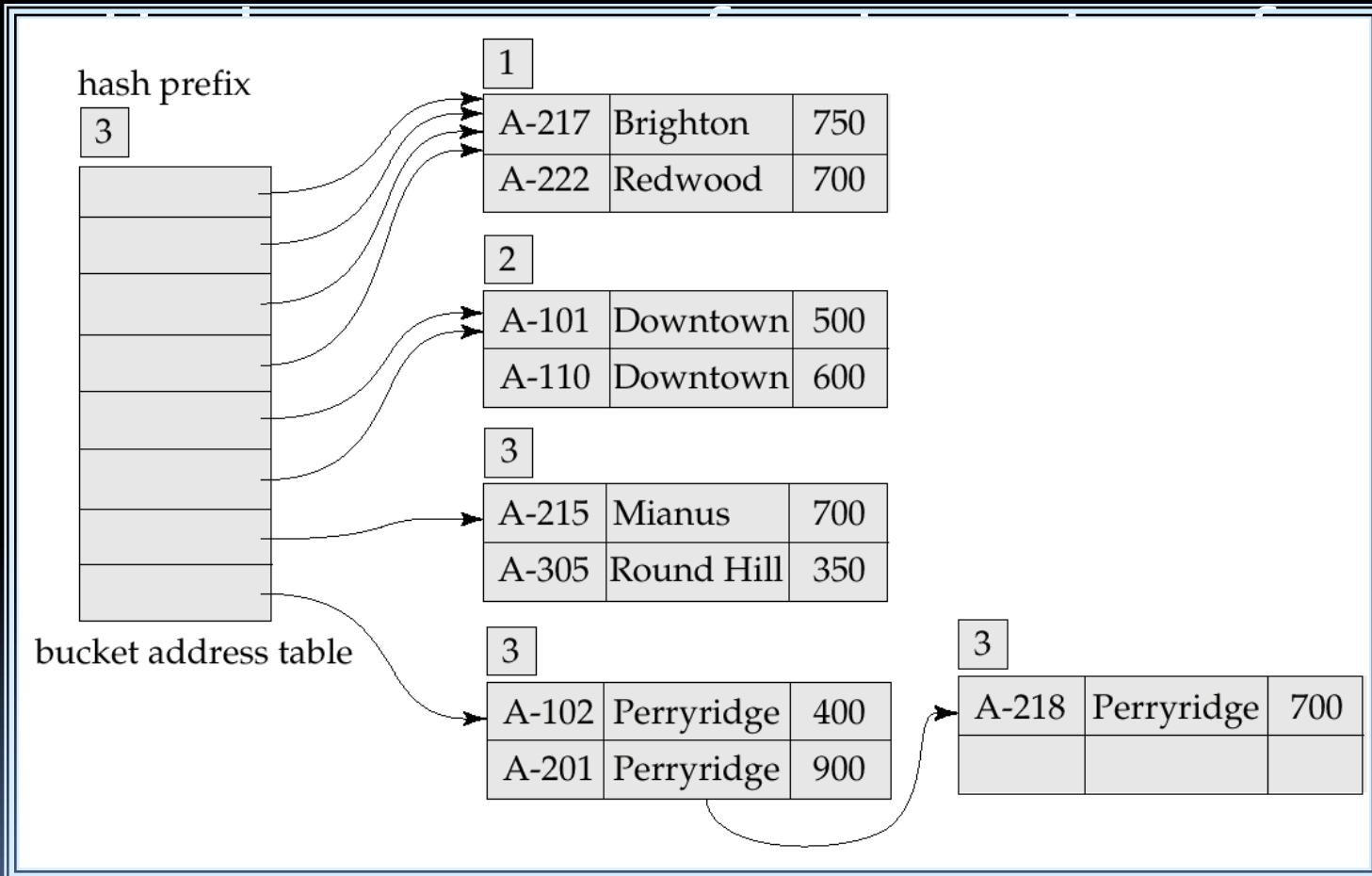


# Example (Cont.)



Hash structure after insertion of three Perryridge records

# Example (Cont.)



# 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)
    - Need a tree structure to locate desired record in the structure!
  - Changing size of bucket address table is an

# 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

# 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

# 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 balances of \$1000; test *branch-name* = "Perryridge".

# Indices on Multiple Attributes

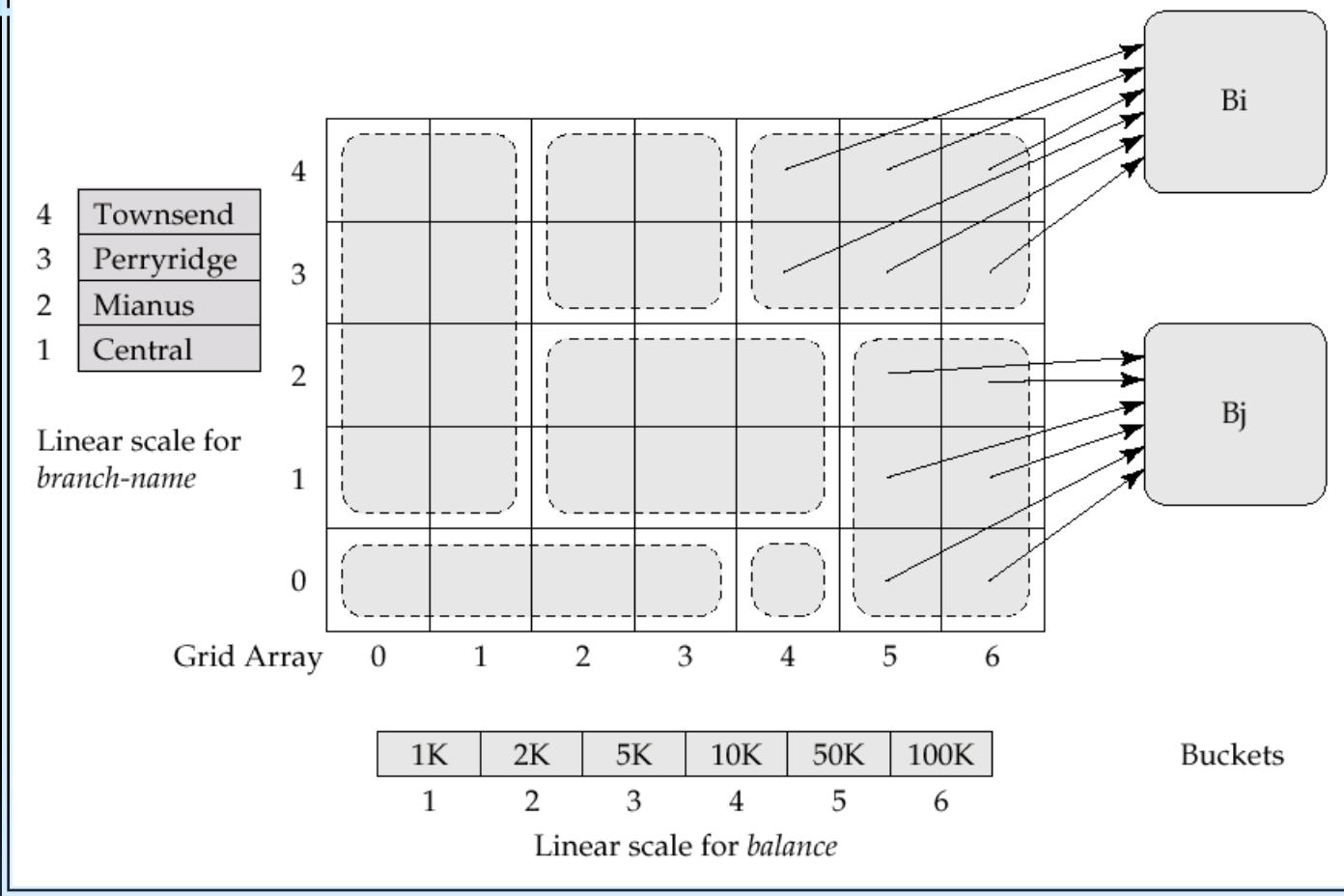
Suppose we have an index on combined search-key  
*(branch-name, balance)*.

- With the where clause  
where *branch-name* = “Perryridge”  
and *balance* = 1000  
the index on the combined search-key will 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”

# Grid Files

- Structure used to speed the processing of general multiple search-key queries involving one or more comparison operators.
- The grid file has a single grid array and one linear scale for each search-key attribute. The grid array has number of dimensions equal to number of search-key attributes.
- Multiple cells of grid array can point to same bucket
- To find the bucket for a search-key value locate the row and column of

# Example Grid File for account



# Queries on a Grid File

- A grid file on two attributes  $A$  and  $B$  can handle queries of all following forms with reasonable efficiency
  - $(a_1 \leq A \leq a_2)$
  - $(b_1 \leq B \leq b_2)$
  - $(a_1 \leq A \leq a_2 \wedge b_1 \leq B \leq b_2),.$
- E.g., to answer  $(a_1 \leq A \leq a_2 \wedge b_1 \leq B \leq b_2)$ , use linear scales to find corresponding candidate grid array cells, and look up all the buckets pointed to from those cells.

# Grid Files (Cont.)

- During insertion, if a bucket becomes full, new bucket can be created if more than one cell points to it.
  - Idea similar to extendable hashing, but on multiple dimensions
  - If only one cell points to it, either an overflow bucket must be created or the grid size must be increased
- Linear scales must be chosen to uniformly distribute records across cells.
  - Otherwise there will be too many

# 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

# Bitmap Indices (Cont.)

- In its simplest form a bitmap index on an attribute has a bitmap for each value of the attribute

record number	<i>name</i>	<i>gender</i>	<i>address</i>	<i>income-level</i>	Bitmaps for <i>gender</i>	Bitmaps for <i>income-level</i>														
0	John	m	Perryridge	L1	<table border="1"><tr><td>m</td><td>1 0 0 1 0</td></tr><tr><td>f</td><td>0 1 1 0 1</td></tr></table>	m	1 0 0 1 0	f	0 1 1 0 1	<table border="1"><tr><td>L1</td><td>1 0 1 0 0</td></tr><tr><td>L2</td><td>0 1 0 0 0</td></tr><tr><td>L3</td><td>0 0 0 0 1</td></tr><tr><td>L4</td><td>0 0 0 1 0</td></tr><tr><td>L5</td><td>0 0 0 0 0</td></tr></table>	L1	1 0 1 0 0	L2	0 1 0 0 0	L3	0 0 0 0 1	L4	0 0 0 1 0	L5	0 0 0 0 0
m	1 0 0 1 0																			
f	0 1 1 0 1																			
L1	1 0 1 0 0																			
L2	0 1 0 0 0																			
L3	0 0 0 0 1																			
L4	0 0 0 1 0																			
L5	0 0 0 0 0																			
1	Diana	f	Brooklyn	L2																
2	Mary	f	Jonestown	L1																
3	Peter	m	Brooklyn	L4																
4	Kathy	f	Perryridge	L3																

# 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 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
    - $\text{not}(A=v)$ :  $(NOT \text{ bitmap-}A-v) AND \text{ExistenceBitmap}$
- Should keep bitmaps for all values, even

# 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 anded 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

END OF CHAPTER



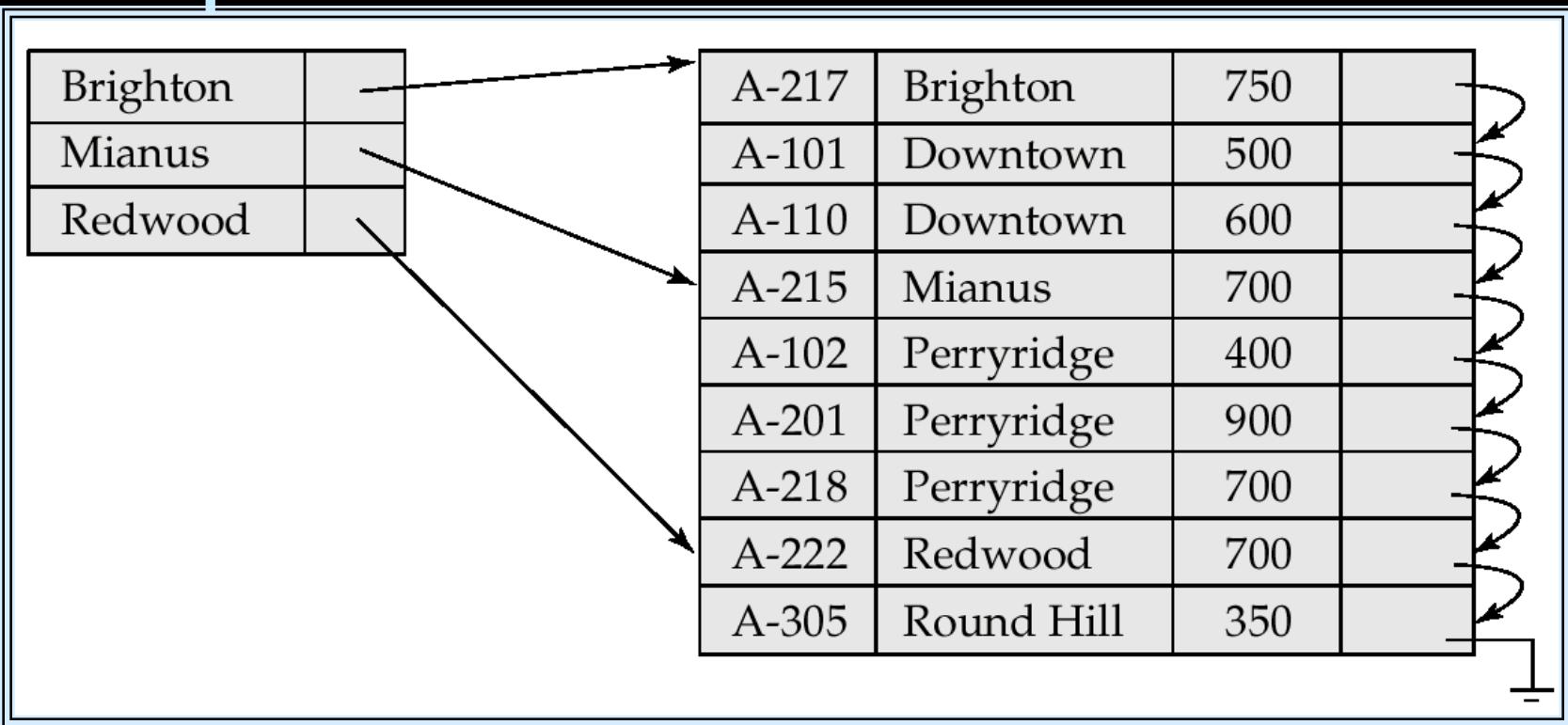
# Partitioned Hashing

- Hash values are split into segments that depend on each attribute of the search-key.  
 $(A_1, A_2, \dots, A_n)$  for  $n$  attribute search-key
- Example:  $n = 2$ , for *customer*, search-key being (*customer-street*, *customer-city*)  

search-key value	hash
<i>value</i>	

(Main, Harrison)	101
------------------	-----

# Sequential File For *account*



# Deletion of “Perryridge” From the B<sup>+</sup>-Tree of Figure 12.12

# Sample account File

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

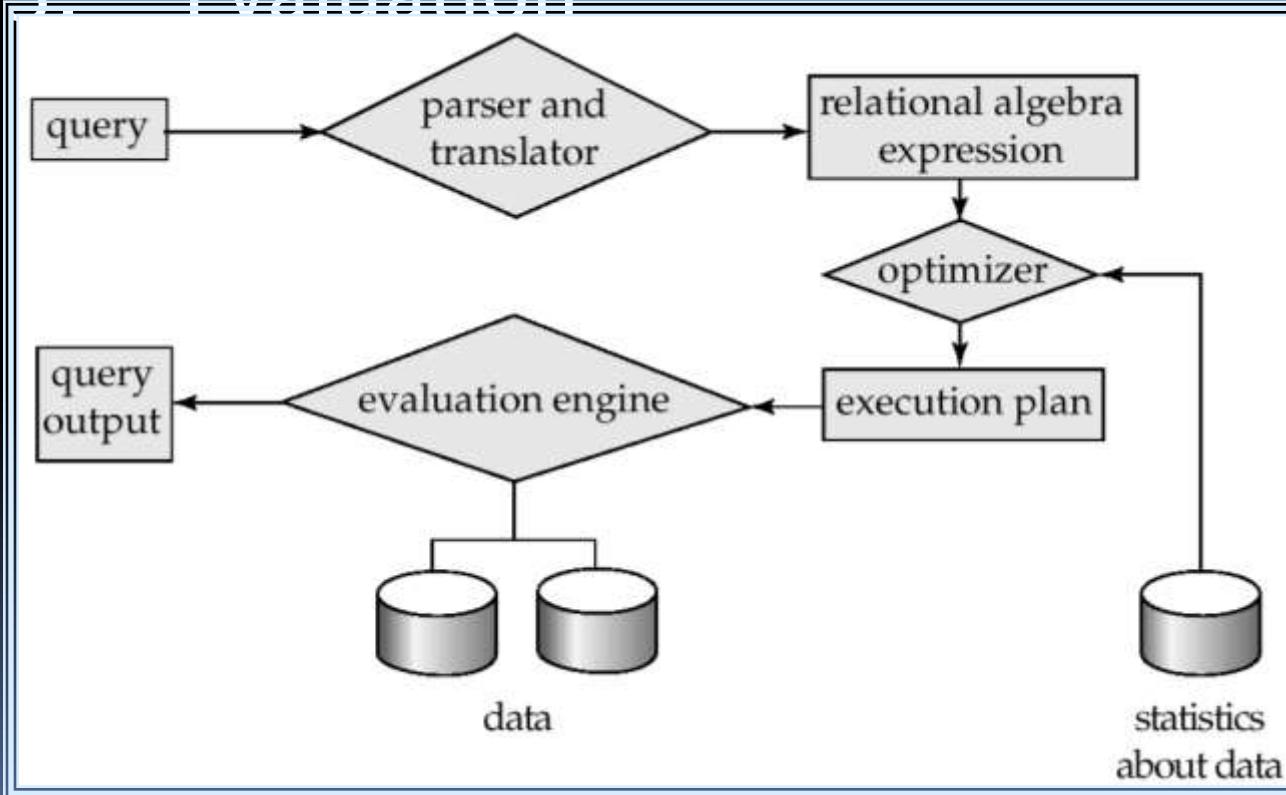
# Chapter 13: Query Processing

- Overview
- Measures of Query Cost
- Selection Operation
- Sorting
- Join Operation
- Other Operations
- Evaluation of Expressions



# Basic Steps in Query Processing

1. Parsing and translation
2. Optimization
3. Evaluation



# Basic Steps in Query Processing (Cont.)

## ▪ Parsing and translation

- translate the query into its internal form. This is then translated into relational algebra.
- Parser checks syntax, verifies relations

## ▪ Evaluation

- The query-execution engine takes a query-evaluation plan, executes that plan, and returns the answers to the query.

# Basic Steps in Query Processing :

## Optimization

- A relational algebra expression may have many equivalent expressions
  - E.g.,  $\sigma_{balance < 2500}(\Pi_{balance}(account))$  is equivalent to  
 $\Pi_{balance}(\sigma_{balance < 2500}(account))$
- Each relational algebra operation can be evaluated using one of several different algorithms
  - Correspondingly, a relational-algebra expression can be evaluated in many ways.
- Annotated expression specifying detailed evaluation strategy is called an evaluation plan

# Basic Steps: Optimization (Cont.)

Query Optimization: Amongst all equivalent evaluation plans choose the one with lowest cost.

- Cost is estimated using statistical information from the database catalog
  - e.g. number of tuples in each relation, size of tuples, etc.

In this chapter we study

- How to measure query costs
- Algorithms for evaluating relational algebra operations
- How to combine algorithms for individual

# Measures of Query Cost

- Cost is generally measured as total elapsed time for answering query
  - Many factors contribute to time cost
    - *disk accesses, CPU, or even network communication*
- Typically disk access is the predominant cost, and is also relatively easy to estimate.  
Measured by taking into account
  - Number of seeks \* average-seek-cost
  - Number of blocks read \* average-

# Measures of Query Cost (Cont.)

- For simplicity we just use *number of block transfers from disk* as the cost measure
  - We ignore the difference in cost between sequential and random I/O for simplicity
  - We also ignore CPU costs for simplicity
- Costs depends on the size of the buffer in main memory
  - Having more memory reduces need for disk access
  - Amount of real memory available to buffer depends on other concurrent OS processes, and hard to determine ahead of actual

# Selection Operation

- File scan – search algorithms that locate and retrieve records that fulfill a selection condition.
- Algorithm A1 (*linear search*). Scan each file block and test all records to see whether they satisfy the selection condition.
  - Cost estimate (number of disk blocks scanned) =  $b_r$ 
    - $b_r$  denotes number of blocks containing records from relation  $r$

# Selection Operation (Cont.)

- A2 (*binary search*). Applicable if selection is an equality comparison on the attribute on which file is ordered.
  - Assume that the blocks of a relation are stored contiguously
  - Cost estimate (number of disk blocks to be scanned):
    - $\lceil \log_2(b_r) \rceil$  — cost of locating the first tuple by a binary search on the blocks
    - *Plus* number of blocks containing records that satisfy selection condition
      - Will see how to estimate this cost in Chapter

# Selections Using Indices

- Index scan – search algorithms that use an index
  - selection condition must be on search-key of index.
- A3 (*primary index on candidate key, equality*). Retrieve a single record that satisfies the corresponding equality condition
  - $Cost = HT_i + 1$
- A4 (*primary index on nonkey, equality*) Retrieve multiple records.
  - Records will be on consecutive blocks
  - $Cost = HT_i + \text{number of blocks containing retrieved records}$
- A5 (*equality on search-key of secondary index*).
  - Retrieve a single record if the search-key is a candidate key
    - $Cost = HT_i + 1$

# Selections Involving

- Can implement selections of the form  $\sigma_{A \leq v}(r)$  or  $\sigma_{A \geq v}(r)$  by using
  - a linear file scan or binary search,
  - or by using indices in the following ways:
- A6 (*primary index, comparison*).  
(Relation is sorted on A)
  - For  $\sigma_{A \geq v}(r)$  use index to find first tuple  $\geq v$  and scan relation sequentially from there
  - For  $\sigma_{A \leq v}(r)$  just scan relation sequentially till first tuple  $> v$ ; do not use index
- A7 (*secondary index, comparison*).
  - For  $\sigma_{A \geq v}(r)$  use index to find first index entry  $\geq v$  and scan index sequentially from there, to find pointers to records.
  - For  $\sigma_{A \leq v}(r)$  just scan leaf pages of index

# Implementation of Complex

Conjunction:  $\sigma_{\theta_1} \wedge \theta_2 \wedge \dots \wedge \theta_n(r)$

- **Selections**
- A8 (*conjunctive selection using one index*).
  - Select a combination of  $\theta_i$  and algorithms A1 through A7 that results in the least cost for  $\sigma_{\theta_i}(r)$ .
  - Test other conditions on tuple after fetching it into memory buffer.
- A9 (*conjunctive selection using multiple-key index*).
  - Use appropriate composite (multiple-key) index if available.
- A10 (*conjunctive selection by intersection of identifiers*).

# Algorithms for Complex

- Disjunction:  $\sigma_{\theta_1 \vee \theta_2 \vee \dots \vee \theta_n}(r)$ .
- Selections
- All (*disjunctive selection by union of identifiers*).
  - Applicable if *all* conditions have available indices.
    - Otherwise use linear scan.
  - Use corresponding index for each condition, and take union of all the obtained sets of record pointers.
  - Then fetch records from file
- Negation:  $\sigma_{\neg \theta}(r)$ 
  - Use linear scan on file
  - If very few records satisfy  $\neg \theta$ , and an index is

# Sorting

- We may build an index on the relation, and then use the index to read the relation in sorted order. May lead to one disk block access for each tuple.
- For relations that fit in memory, techniques like quicksort can be used. For relations that don't fit in memory, external sort-merge is a good choice.

# External Sort-Merge

Let  $M$  denote memory size (in pages).

**1. Create sorted runs.** Let  $i$  be 0 initially.

Repeatedly do the following till the end of the relation:

- (a) Read  $M$  blocks of relation into memory
- (b) Sort the in-memory blocks
- (c) Write sorted data to run  $R_i$ ; increment  $i$ .

Let the final value of  $i$  be  $N$

**2. Merge the runs (N-way merge).** We assume

(for now) that  $N < M$ .

**1.** Use  $N$  blocks of memory to buffer input runs, and 1 block to buffer output. Read the first block of each run into its buffer page

**2. repeat**

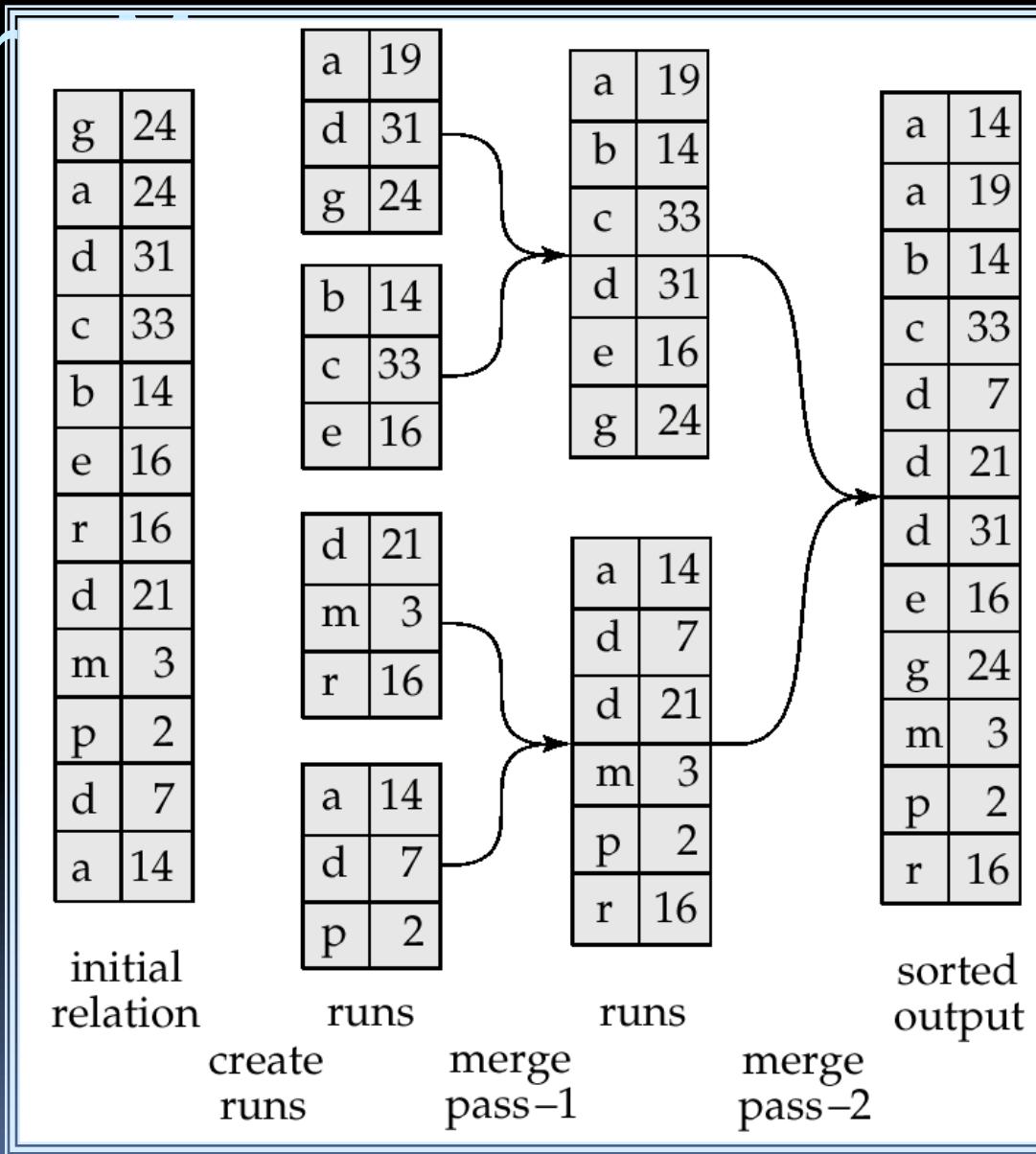
**1.** Select the first record (in sort order) among all buffer pages

**2.** Write the record to the output buffer. If the output buffer is full write it to disk

# External Sort-Merge (Cont.)

- If  $i \geq M$ , several merge *passes* are required.
  - In each pass, contiguous groups of  $M - 1$  runs are merged.
  - A pass reduces the number of runs by a factor of  $M - 1$ , and creates runs longer by the same factor.
    - E.g. If  $M=11$ , and there are 90 runs, one pass reduces the number of runs to 9, each 10 times the size of the initial runs
    - Repeated passes are performed till all runs have been merged into one.

# Example: External Sorting Using Sorter



# External Merge Sort (Cont.)

- Cost analysis:
  - Total number of merge passes required:  $\lceil \log_{M-1}(b_r/M) \rceil$ .
  - Disk accesses for initial run creation as well as in each pass is  $2b_r$ 
    - for final pass, we don't count write cost
      - we ignore final write cost for all operations since the output of an operation may be sent to the parent operation without being written to disk

Thus total number of disk accesses for external sorting:

# Join Operation

- Several different algorithms to implement joins
  - Nested-loop join
  - Block nested-loop join
  - Indexed nested-loop join
  - Merge-join
  - Hash-join
- Choice based on cost estimate
- Examples use the following information

# Nested-Loop Join

- To compute the theta join  $r \times_s \theta s$   
for each tuple  $t_r$  in  $r$  do begin  
  for each tuple  $t_s$  in  $s$  do begin  
    test pair  $(t_r, t_s)$  to see if they satisfy  
    the join condition  $\theta$   
    if they do, add  $t_r \cdot t_s$  to the result.  
  end  
end
- $r$  is called the outer relation and  $s$  the inner relation of the join.
- Requires no indices and can be used with any kind of join condition.

# Nested-Loop Join (Cont.)

- In the worst case, if there is enough memory only to hold one block of each relation, the estimated cost is

$$n_r * b_s + b_r$$

disk accesses.

- If the smaller relation fits entirely in memory, use that as the inner relation. Reduces cost to  $b_r + b_s$  disk accesses.
- Assuming worst case memory availability cost estimate is
  - $5000 * 400 + 100 = 2,000,100$  disk accesses with *depositor* as outer relation, and

# Block Nested-Loop Join

- Variant of nested-loop join in which every block of inner relation is paired with every block of outer relation.

```
for each block B_r of r do begin
 for each block B_s of s do begin
 for each tuple t_r in B_r do
```

```
begin
```

```
 for each tuple t_s in B_s do
begin
```

Check if  $(t_r, t_s)$  satisfy  
the join condition

if they do, add  $t_r \cdot t_s$  to

# Block Nested-Loop Join (Cont.)

- Worst case estimate:  $b_r * b_s + b_r$  block accesses.
  - Each block in the inner relation  $s$  is read once for each *block* in the outer relation (instead of once for each tuple in the outer relation)
- Best case:  $b_r + b_s$  block accesses.
- Improvements to nested loop and block nested loop algorithms:
  - In block nested-loop, use  $M - 2$  disk blocks as blocking unit for outer relations, where  $M$  = memory size in blocks; use remaining two blocks to buffer inner relation and output

# Indexed Nested-Loop Join

- Index lookups can replace file scans if
  - join is an equi-join or natural join and
  - an index is available on the inner relation's join attribute
    - Can construct an index just to compute a join.
- For each tuple  $t_r$  in the outer relation  $r$ , use the index to look up tuples in  $s$  that satisfy the join condition with tuple  $t_r$ .
- Worst case: buffer has space for only one page of  $r$ , and, for each tuple in  $r$ , we perform an index lookup on  $s$ .
- Cost of the join:  $b_r + n_r * c_s$

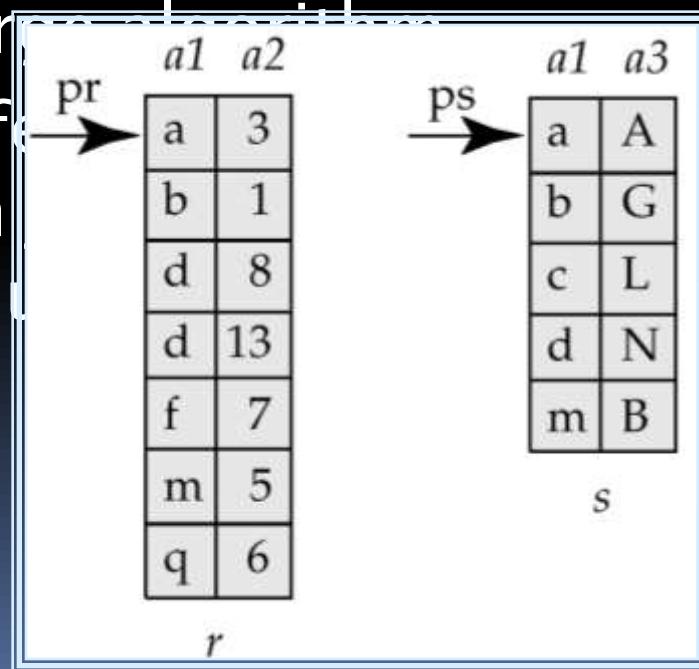
# Example of Nested-Loop Join Costs



- Compute *depositor* *customer*, with *depositor* as the outer relation.
- Let *customer* have a primary B<sup>+</sup>-tree index on the join attribute *customer-name*, which contains 20 entries in each index node.
- Since *customer* has 10,000 tuples, the height of the tree is 4, and one more access is needed to find the actual data
  - *depositor* has 5000 tuples

# Merge-Join

1. Sort both relations on their join attribute (if not already sorted on the join attributes).
2. Merge the sorted relations to join them
  1. Join step is similar to the merge stage of the sort-merge algorithm.
  2. Main difference is that if duplicate key pair with same value must be matched



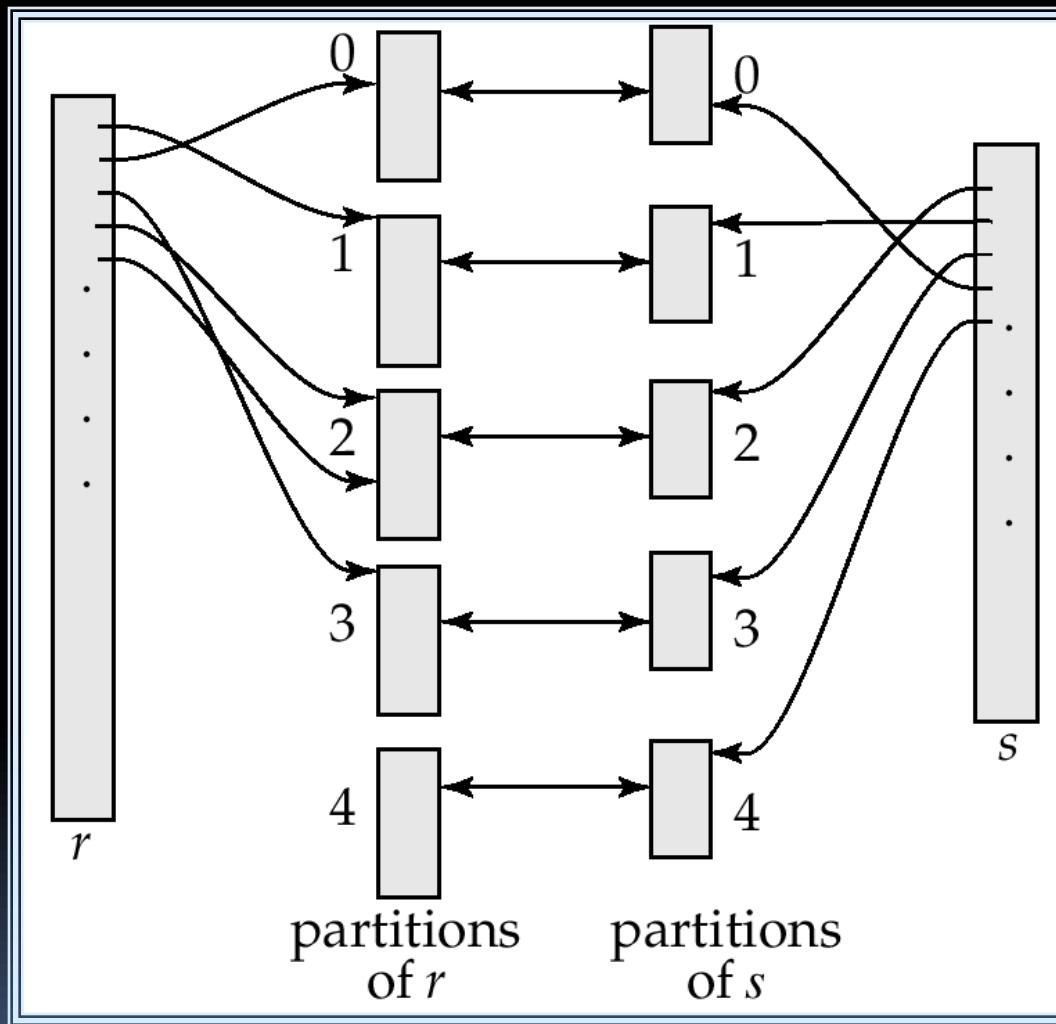
# Merge-Join (Cont.)

- Can be used only for equi-joins and natural joins
- Each block needs to be read only once (assuming all tuples for any given value of the join attributes fit in memory)
- Thus number of block accesses for merge-join is
$$b_r + b_s + \text{the cost of sorting}$$
if relations are unsorted.
- hybrid merge-join: If one relation is sorted, and the other has a secondary B<sup>+</sup>-tree index on the join attribute

# Hash-Join

- Applicable for equi-joins and natural joins.
- A hash function  $h$  is used to partition tuples of both relations
- $h$  maps  $JoinAttrs$  values to  $\{0, 1, \dots, n\}$ , where  $JoinAttrs$  denotes the common attributes of  $r$  and  $s$  used in the natural join.
  - $r_0, r_1, \dots, r_n$  denote partitions of  $r$  tuples
    - Each tuple  $t_r \in r$  is put in partition  $r_i$  where  $i = h(t_r[JoinAttrs])$ .
  - $r_0, r_1, \dots, r_n$  denotes partitions of  $s$  tuples
    - Each tuple  $t_s \in s$  is put in partition  $r_j$  where  $j = h(t_s[JoinAttrs])$ .

# Hash-Join (Cont.)



# Hash-Join (Cont.)

- $r$  tuples in  $r_i$  need only to be compared with  $s$  tuples in  $s_i$ . Need not be compared with  $s$  tuples in any other partition, since:
  - an  $r$  tuple and an  $s$  tuple that satisfy the join condition will have the same value for the join attributes.
  - If that value is hashed to some value  $i$ , the  $r$  tuple has to be in  $r_i$  and the  $s$  tuple in  $s_i$ .

# Hash-Join Algorithm

The hash-join of  $r$  and  $s$  is computed as follows.

1. Partition the relation  $s$  using hashing function  $h$ . When partitioning a relation, one block of memory is reserved as the output buffer for each partition.
2. Partition  $r$  similarly.
3. For each  $i$ :
  - (a) Load  $s_i$  into memory and build an in-memory hash index on it using the join attribute. This hash index uses a different hash function than the earlier one  $h$ .

# Hash-Join algorithm (Cont.)

- The value  $n$  and the hash function  $h$  is chosen such that each  $s_i$  should fit in memory.
  - Typically  $n$  is chosen as  $\lceil b_s/M \rceil * f$  where  $f$  is a “fudge factor”, typically around 1.2
  - The probe relation partitions  $s_i$  need not fit in memory
- Recursive partitioning required if number of partitions  $n$  is greater than number of pages  $M$  of memory.
  - instead of partitioning  $n$  ways, use  $M - 1$  partitions for  $s$
  - Further partition the  $M - 1$  partitions

# Handling of Overflows

- Hash-table overflow occurs in partition  $s_i$  if  $s_i$  does not fit in memory. Reasons could be
  - Many tuples in  $s_i$  with same value for join attributes
  - Bad hash function
- Partitioning is said to be skewed if some partitions have significantly more tuples than some others
- Overflow resolution can be done in build phase
  - Partition  $s_i$  is further partitioned using different hash function.
  - Partition  $s_i$  must be similarly partitioned

## Cost of Hash-Join

- If recursive partitioning is not required: cost of hash join is
$$3(b_r + b_s) + 2 * n_h$$
- If recursive partitioning required, number of passes required for partitioning  $s$  is  $\lceil \log_{M-1}(b_s) - 1 \rceil$ . This is because each final partition of  $s$  should fit in memory.
- The number of partitions of probe relation  $r$  is the same as that for build relation  $s$ ; the number of passes for partitioning of  $r$  is also the same as for  $s$ .
- Therefore it is best to choose the

# Example of Cost of Hash-Join

*customer*  $\bowtie$  *depositor*

- Assume that memory size is 20 blocks
- $b_{depositor} = 100$  and  $b_{customer} = 400$ .
- *depositor* is to be used as build input. Partition it into five partitions, each of size 20 blocks. This partitioning can be done in one pass.
- Similarly, partition *customer* into five partitions, each of size 80. This is also done in one pass.
- Therefore total cost:  $3(100 + 400) = 1500$  block transfers

# Hybrid Hash-Join

- Useful when memory sizes are relatively large, and the build input is bigger than memory.
- Main feature of hybrid hash join:  
Keep the first partition of the build relation in memory.
- E.g. With memory size of 25 blocks, *depositor* can be partitioned into five partitions, each of size 20 blocks.
- Division of memory:
  - The first partition occupies 20 blocks of memory  $\sqrt{b_s}$
  - 1 block is used for input, and 1 block each for buffering the other 4 partitions

# Complex Joins

- Join with a conjunctive condition:

$$r \underset{\theta_1 \wedge \theta_2 \wedge \dots \wedge \theta_n}{\bowtie} s$$

- Either use nested loops/block nested loops, or
- Compute the result of one of the simpler joins  $r \underset{\theta_i}{\bowtie} s$ 
  - final result comprises those tuples in the intermediate  $\bowtie$  result that satisfy the remaining conditions

$$\theta_1 \bowtie \dots \wedge \underset{\theta_{i+1}}{\bowtie} \wedge \theta_{i+1} \wedge \dots \bowtie \wedge \theta_n$$

- Join with a disjunctive condition

$$r \underset{\theta_1 \vee \theta_2 \vee \dots \vee \theta_n}{\bowtie} s$$

- Either use nested loops/block nested

# Other Operations

- Duplicate elimination can be implemented via hashing or sorting.
  - On sorting duplicates will come adjacent to each other, and all but one set of duplicates can be deleted. *Optimization:* duplicates can be deleted during run generation as well as at intermediate merge steps in external sort-merge.
  - Hashing is similar – duplicates will come into the same bucket.

# Other Operations : Aggregation

- Aggregation can be implemented in a manner similar to duplicate elimination.
  - Sorting or hashing can be used to bring tuples in the same group together, and then the aggregate functions can be applied on each group.
  - *Optimization:* combine tuples in the same group during run generation and intermediate merges, by computing partial aggregate values
    - For count, min, max, sum: keep aggregate

# Other Operations : Set

- **Operations**: Set operations ( $\cup$ ,  $\cap$  and  $-$ ): can either use variant of merge-join after sorting, or variant of hash-join.
- E.g., Set operations using hashing:
  1. Partition both relations using the same hash function, thereby creating,  $r_1, \dots, r_n$  and  $s_1, s_2, \dots, s_n$
  2. Process each partition  $i$  as follows.  
Using a different hashing function, build an in-memory hash index on  $r_i$  after it is brought into memory.
  3.  $- r \cup s$ : Add tuples in  $s_i$  to the hash index if they are not already in it. At end of partition processing, join the hash index with  $r$ .

# Other Operations : Outer Join

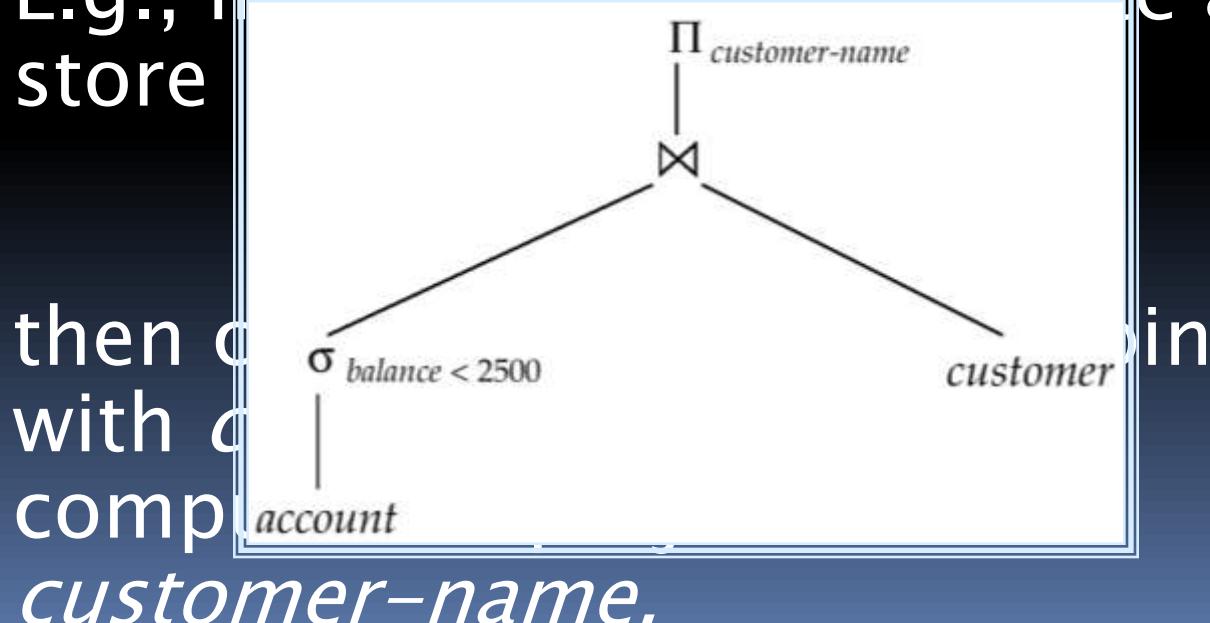
- Outer join can be computed either as
  - A join followed by addition of null-padded non-participating tuples.
  - by modifying the join algorithms.  $\bowtie$
- Modifying merge join to compute  $r \bowtie s$ 
  - In  $r \bowtie s$ , non participating tuples are those in  $r - \Pi_R(r \bowtie s)$   $\supseteq$
  - Modify merge-join to compute  $r \bowtie s$ : During merging, for every tuple  $t_r$  from  $r$  that do not match any tuple in  $s$ , output  $t_r$  padded with nulls.
  - Right outer-join and full outer-join can be

# Evaluation of Expressions

- So far: we have seen algorithms for individual operations
- Alternatives for evaluating an entire expression tree
  - **Materialization:** generate results of an expression whose inputs are relations or are already computed, **materialize** (store) it on disk. Repeat.
  - **Pipelining:** pass on tuples to parent operations even as an operation is being executed

# Materialization

- Materialized evaluation: evaluate one operation at a time, starting at the lowest-level. Use intermediate results materialized into temporary relations to evaluate next-level operations.
- E.g., in figure below compute and store



# Materialization (Cont.)

- Materialized evaluation is always applicable
- Cost of writing results to disk and reading them back can be quite high
  - Our cost formulas for operations ignore cost of writing results to disk, so
    - Overall cost = Sum of costs of individual operations +  
cost of writing intermediate results to disk
- Double buffering: use two output buffers for each operation when one

# Pipelining

- Pipelined evaluation : evaluate several operations simultaneously, passing the results of one operation on to the next.
- E.g., in previous expression tree, don't store result of
  - instead, pass tuples directly to the join.. Similarly, don't store result of join, pass tuples directly to projection.
- Much cheaper than materialization: no need to store a temporary relation to disk

# Pipelining (Cont.)

- In demand driven or lazy evaluation
  - system repeatedly requests next tuple from top level operation
  - Each operation requests next tuple from children operations as required, in order to output its next tuple
  - In between calls, operation has to maintain “state” so it knows what to return next
  - Each operation is implemented as an **iterator** implementing the following operations
    - `open()`
      - E.g. file scan: initialize file scan, store pointer to beginning of file as state
      - E.g. merge join: sort relations and store pointers to beginning of sorted relations

# Pipelining (Cont.)

- In produce-driven or eager pipelining
  - Operators produce tuples eagerly and pass them up to their parents
    - Buffer maintained between operators, child puts tuples in buffer, parent removes tuples from buffer
    - if buffer is full, child waits till there is space in the buffer, and then generates more tuples
  - System schedules operations that have space in output buffer and can process more input tuples

# Evaluation Algorithms for

- Some algorithms are not able to output results even as they get input tuples
  - E.g. merge join, or hash join
  - These result in intermediate results being written to disk and then read back always
- Algorithm variants are possible to generate (at least some) results on the fly, as input tuples are read in
  - E.g. hybrid hash join generates output tuples even as probe relation tuples in the in-memory partition (partition 0) are read in
  - **Pipelined join technique:** Hybrid hash join, modified to buffer partition 0 tuples of both relations in memory, and then process them sequentially.

# Complex Joins

- Join involving three relations:  $\text{loan} \bowtie \text{depositor} \bowtie \text{customer}$
- Strategy 1. Compute  $\text{depositor} \bowtie \text{customer}$ ; use result to compute  $\text{loan} \bowtie (\text{depositor} \bowtie \text{customer})$
- Strategy 2. Compute  $\text{loan} \bowtie \text{depositor}$  first, and then join the result with  $\text{customer}$ .
- Strategy 3. Perform the pair of joins at once. Build and index on  $\text{loan}$  for  $\text{loan-number}$ , and on  $\text{customer}$  for  $\text{customer-name}$ .

# CHAPTER 14

# QUERY OPTIMIZATION

# Chapter 14: Query Optimization

- Introduction
- Catalog Information for Cost Estimation
- Estimation of Statistics
- Transformation of Relational Expressions
- Dynamic Programming for Choosing Evaluation Plans

# Introduction

Alternative ways of evaluating a given query

- Equivalent expressions
- Different algorithms for each operation  
(Chapter 13)

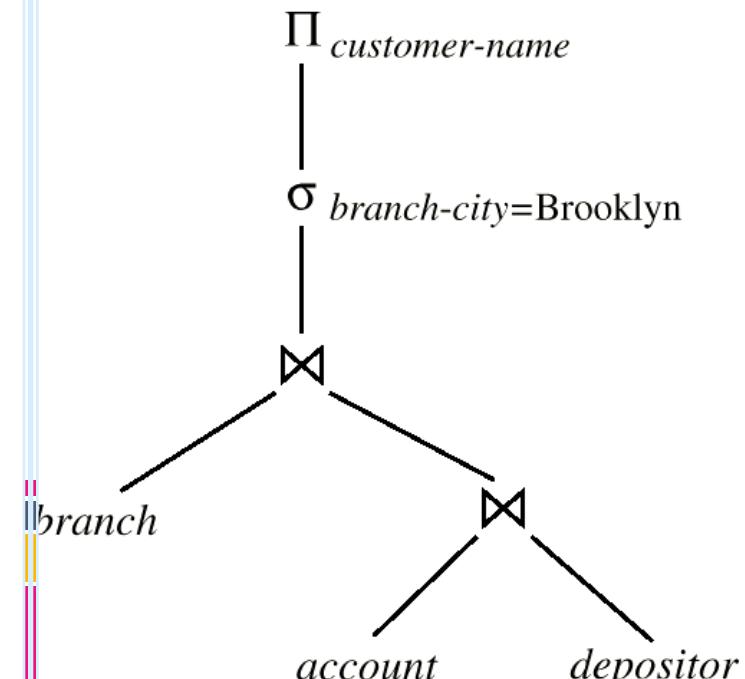
Cost difference between a good and a bad way of evaluating a query can be enormous

- Example: performing a  $r \times s$  followed by a selection  $r.A = s.B$  is much slower than performing a join on the same condition

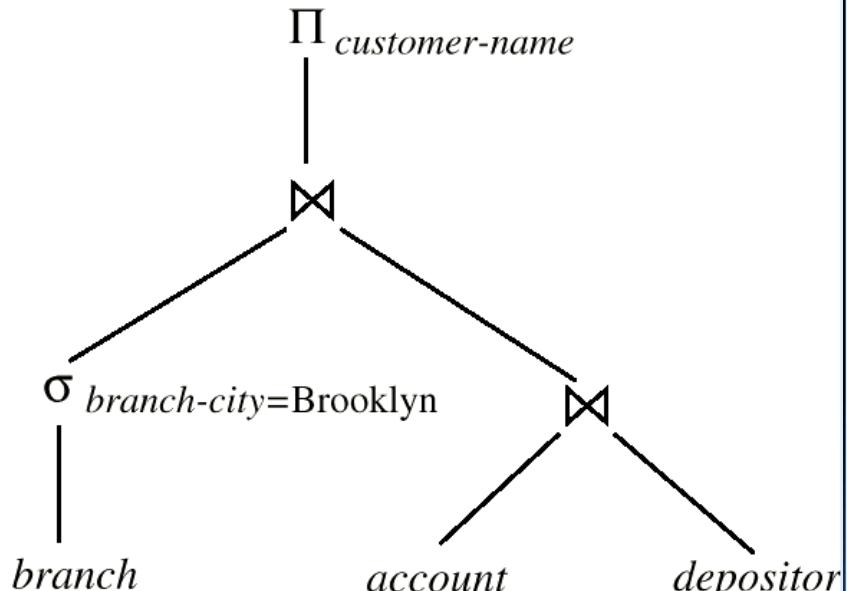
Need to estimate the cost of operations

# Introduction (Cont.)

Relations generated by two equivalent expressions have the same set of attributes and contain the same set of



(a) Initial Expression Tree



(b) Transformed Expression Tree

# Introduction (Cont.)

- Generation of query-evaluation plans for an expression involves several steps:
  1. Generating logically equivalent expressions
    - Use **equivalence rules** to transform an expression into an equivalent one.
  2. Annotating resultant expressions to get alternative query plans
  3. Choosing the cheapest plan based on estimated cost
- The overall process is called **cost-based optimization**.

# Overview of chapter

- Statistical information for cost estimation
- Equivalence rules
- Cost-based optimization algorithm
- Optimizing nested subqueries
- Materialized views and view maintenance

# Statistical Information for Cost Estimation

$n_r$ : number of tuples in a relation  $r$ .

- $b_r$ : number of blocks containing tuples of  $r$ .
- $s_r$ : size of a tuple of  $r$ .
- $f_r$ : blocking factor of  $r$  — i.e., the number of tuples of  $r$  that fit into one block.
- $V(A, r)$ : number of distinct values that appear in  $r$  for attribute  $A$ ; same as the size of  $\Pi_A(r)$ .
- $SCA(r)$ : selection cardinality of

# Catalog Information about Indices

- $f_i$ : average fan-out of internal nodes of index  $i$ , for tree-structured indices such as B+-trees.
- $HT_i$ : number of levels in index  $i$ 
  - i.e., the height of  $i$ .
    - For a balanced tree index (such as B+-tree) on attribute  $A$  of relation  $r$ ,  $HT_i = \lceil \log_{f_i}(N(A, r)) \rceil$ .
    - For a hash index,  $HT_i$  is 1.
    - $LB_i$ : number of lowest-level index blocks

# Measures of Query Cost

- Recall that
  - Typically disk access is the predominant cost, and is also relatively easy to estimate.
  - The *number of block transfers from disk* is used as a measure of the actual cost of evaluation.
  - It is assumed that all transfers of blocks have the same cost.
    - Real life optimizers do not make this assumption, and distinguish between sequential and random disk access
- We do not include cost to writing output to disk

# Selection Size Estimation

- Equality selection  $\sigma_{A=\nu}(r)$ 
  - $SC(A, r)$  : number of records that will satisfy the selection
  - $\lceil SC(A, r)/f_r \rceil$  — number of blocks that these records will occupy
  - E.g. Binary search cost estimate becomes
- Equality condition on a key attribute:  
 $SC(A, r) = 1$

# Statistical Information for Examples

- $f_{account} = 20$  (20 tuples of *account* fit in one block)
- $V(branch-name, account) = 50$  (50 branches)
- $V(balance, account) = 500$  (500 different *balance* values)
- $\pi_{account} = 10000$  (*account* has 10,000 tuples)
- Assume the following indices exist on *account*:

# Selections Involving

**Comparisons** Selections of the form  $\sigma_{A \leq v}(r)$  (case of  $\sigma_{A \geq v}(r)$  is symmetric)

- Let  $c$  denote the estimated number of tuples satisfying the condition.
  - If  $\min(A, r)$  and  $\max(A, r)$  are available in catalog
    - $C = 0$  if  $v < \min(A, r)$
    - $C =$
  - In absence of statistical information  $c$  is assumed to be  $n_r / 2$ .

# Implementation of Complex

~~Selections~~ Selectivity of a condition  $\theta_i$  is the probability that a tuple in the relation  $r$  satisfies  $\theta_i$ . If  $s_i$  is the number of satisfying tuples in  $r$ , the selectivity of  $\theta_i$  is given by  $s_i / n_r$ .

- Conjunction:  $\sigma_{\theta_1 \wedge \theta_2 \wedge \dots \wedge \theta_n}(r)$ . The estimate for number of

tuples in the result is:  $\left( \frac{s_1}{n_r} \right) * \left( \frac{s_2}{n_r} \right) * \dots * \left( \frac{s_n}{n_r} \right)$

- Disjunction:  $\sigma_{\theta_1 \vee \theta_2 \vee \dots \vee \theta_n}(r)$ . Estimated number of tuples:

# Join Operation: Running

Running example:

$depositor \bowtie customer$

Catalog information for join examples:

- $n_{customer} = 10,000$ .
- $f_{customer} = 25$ , which implies that  
 $b_{customer} = 10000/25 = 400$ .
- $n_{depositor} = 5000$ .
- $f_{depositor} = 50$ , which implies that  
 $b_{depositor} = 5000/50 = 100$ .
- $V(customer-name, depositor) = 2500$ , which implies that , on average, each customer has two accounts

# Estimation of the Size of Joins

- The Cartesian product  $r \times s$  contains  $n_r \cdot n_s$  tuples; each tuple occupies  $s_r + s_s$  bytes.
- If  $R \cap S = \emptyset$ , then  $r \bowtie s$  is the same as  $r \times s$ .
- If  $R \cap S$  is a key for  $R$ , then a tuple of  $s$  will join with at most one tuple from  $r$ 
  - therefore, the number of tuples in  $r \bowtie s$  is no greater than the number of tuples in  $s$ .
- If  $R \cap S$  in  $S$  is a foreign key in  $S$  referencing  $R$ , then the number of tuples in  $r \bowtie s$  is exactly the same as

# Estimation of the Size of Joins (Cont.)



- If  $R \cap S = \{A\}$  is not a key for  $R$  or  $S$ .  
If we assume that every tuple  $t$  in  $R$  produces tuples in  $R \times S$ , the number of tuples in  $R \times S$  is estimated to be:

If the reverse is true, the estimate obtained will be:

# Estimation of the Size of Joins (Cont.)



- Compute the size estimates for *depositor* *customer* without using information about foreign keys:
  - $V(\text{customer-name}, \text{depositor}) = 2500$ , and  
 $V(\text{customer-name}, \text{customer}) = 10000$
  - The two estimates are  $5000 * 10000/2500 - 20,000$  and  $5000 * 10000/10000 = 5000$
  - We choose the lower estimate, which in this case, is the same as our earlier computation using foreign keys.

# Size Estimation for Other Operations

- Projection: estimated size of  $\Pi_A(r) = V(A, r)$
- Aggregation : estimated size of  $_A g_F(r) = V(A, r)$
- Set operations
  - For unions/intersections of selections on the same relation: rewrite and use size estimate for selections
    - E.g.  $\sigma_{\theta_1}(r) \cup \sigma_{\theta_2}(r)$  can be rewritten as  $\sigma_{\theta_1} \sigma_{\theta_2}(r)$
    - For operations on different relations:

# Size Estimation (Cont.)

- Outer join:

- Estimated size of  $r \bowtie s = \text{size of } r + \text{size of } s$ 
  - Case of right outer join is symmetric
- Estimated size of  $r \bowtie s = \text{size of } r + \text{size of } r + \text{size of } s$

# Estimation of Number of Distinct Values

Selections:  $\sigma_\theta(r)$

- If  $\theta$  forces  $A$  to take a specified value:  $\mathcal{N}(A, \sigma_\theta(r)) = 1$ .
  - e.g.,  $A = 3$
- If  $\theta$  forces  $A$  to take on one of a specified set of values:  
 $\mathcal{N}(A, \sigma_\theta(r)) = \text{number of specified values.}$ 
  - (e.g.,  $(A = 1 \vee A = 3 \vee A = 4)$ ),
- If the selection condition  $\theta$  is of the form  $A \ op \ r$   
estimated  $\mathcal{N}(A, \sigma_\theta(r)) = \mathcal{N}(A, r) * s$

# Estimation of Distinct Values (Cont.)

Joins:  $r \times s$

- If all attributes in  $A$  are from  $r$  estimated  $\mathcal{V}(A, r \times s) = \min_{\times} (\mathcal{V}(A, r), n_r \cdot n_s)$
- If  $A$  contains attributes  $A1$  from  $r$  and  $A2$  from  $s$ , then estimated  $\mathcal{V}(A, r \times s) = \min(\mathcal{V}(A1, r) * \mathcal{V}(A2 - A1, s), \mathcal{V}(A1 - A2, r) * \mathcal{V}(A2, s), n_r \cdot n_s)$ 
  - More accurate estimate can be got using probability theory, but this one works fine

# Estimation of Distinct Values

- (Cont.) Estimation of distinct values are straightforward for projections.
  - They are the same in  $\Pi_A(r)$  as in  $r$ .
- The same holds for grouping attributes of aggregation.
- For aggregated values
  - For  $\min(A)$  and  $\max(A)$ , the number of distinct values can be estimated as  $\min(V(A, r), \mathcal{N}(G, r))$  where  $G$  denotes grouping attributes
  - For other aggregates, assume all values are distinct, and use  $\mathcal{N}(G, r)$

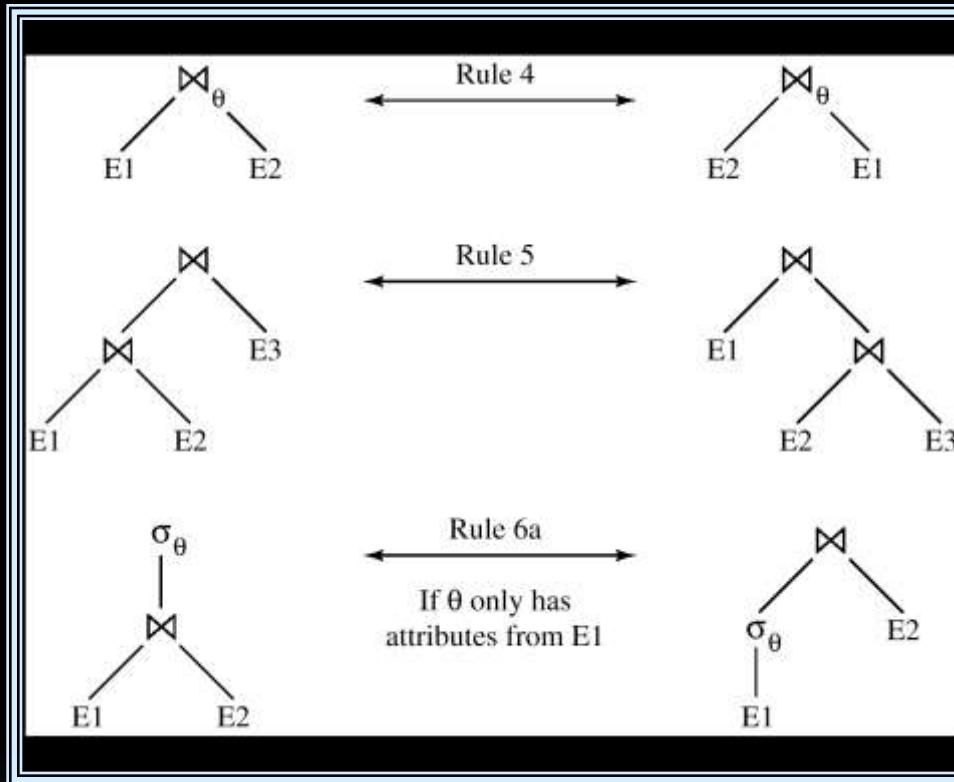
# Transformation of Relational Expressions

- Two relational algebra expressions are said to be **equivalent** if on every legal database instance the two expressions generate the same set of tuples
  - Note: order of tuples is irrelevant
- In SQL, inputs and outputs are multisets of tuples
  - Two expressions in the multiset version of the relational algebra are said to be equivalent if on every legal database instance the two expressions generate the same multiset of tuples

# Equivalence Rules

1. Conjunctive selection operations can be deconstructed into a sequence of individual selections.
2. Selection operations are commutative.
3. Only the last ~~⊗~~ projection operation is needed, the others can be omitted.

# Pictorial Depiction of Equivalence Rules



# Equivalence Rules (Cont.)

$\bowtie$        $\bowtie$

5. Theta-join operations (and natural joins) are ~~not~~ commutative.

$$E_1 \underset{\theta}{\bowtie} E_2 = E_2 \underset{\theta}{\bowtie} E_1$$

6. (a) Natural join operations are associative:

$$(E_1 \underset{\bowtie}{\bowtie} E_2) \underset{\bowtie}{\bowtie} E_3 = E_1 \underset{\bowtie}{\bowtie} (E_2 \underset{\bowtie}{\bowtie} E_3)$$

(b) Theta joins are associative in the following manner:

$$(E_1 \underset{\theta_1}{\bowtie} E_2) \underset{\theta_2}{\bowtie} \dots \underset{\theta_n}{\bowtie} E_n = E_1 \underset{\theta_1}{\bowtie} (E_2 \underset{\theta_2}{\bowtie} \dots \underset{\theta_n}{\bowtie} E_n)$$

# Equivalence Rules (Cont.)

7. The selection operation distributes over the theta join operation under the following two conditions:
- (a) When all the attributes in  $\theta_0$  involve only the attributes of one of the expressions ( $E_1$ ) being joined.

$$\sigma_{\theta_0}(E_1 \ \theta \ E_2) = (\sigma_{\theta_0}(E_1)) \ \theta \ E_2$$

# Equivalence Rules (Cont.)

8. The projections operation distributes over the theta join operation as follows:

(a) if  $\square$  involves only  $\bowtie$  attributes from  $L_1 \cup L_2$ :

(b) Consider a join  $E_1 \theta E_2$ .

- Let  $L_1 \bowtie L_2$  be sets of attributes from  $E_1$  and  $E_2$ , respectively.
- Let  $L_3$  be attributes of  $E_1$  that are involved in join condition  $\theta$ , but are not in  $L_1 \cup L_2$ .

# Equivalence Rules (Cont.)

9. The set operations union and intersection are commutative

$$\begin{aligned}E_1 \cup E_2 &= E_2 \cup E_1 \\E_1 \cap E_2 &= E_2 \cap E_1\end{aligned}$$

■ (set difference is not commutative).

10. Set union and intersection are associative.

$$\begin{aligned}(E_1 \cup E_2) \cup E_3 &= E_1 \cup (E_2 \cup E_3) \\(E_1 \cap E_2) \cap E_3 &= E_1 \cap (E_2 \cap E_3)\end{aligned}$$

11. The selection operation distributes over  $\cup$ ,  $\cap$  and  $-$ .

$$\sigma_{\theta}(E_1 - E_2) = \sigma_{\theta}(E_1) - \sigma_{\theta}(E_2)$$

and similarly for  $\cup$  and  $\cap$

in place of

# Transformation Example

- Query: Find the names of all customers who have an account at some branch located in Brooklyn.

$$\Pi_{customer-name}(\sigma_{branch-city = "Brooklyn"}(branch \ \ \ account depositor))$$

- Transformation using rule 7a.

$$\Pi_{customer-name}((\sigma_{branch-city = "Brooklyn"}(branch)) \ \ \ account \ \ \ depositor)$$

- Performing the selection as early as

# Example with Multiple

- **Query:** Find the names of all customers with an account at a Brooklyn branch whose account balance is over \$1000.

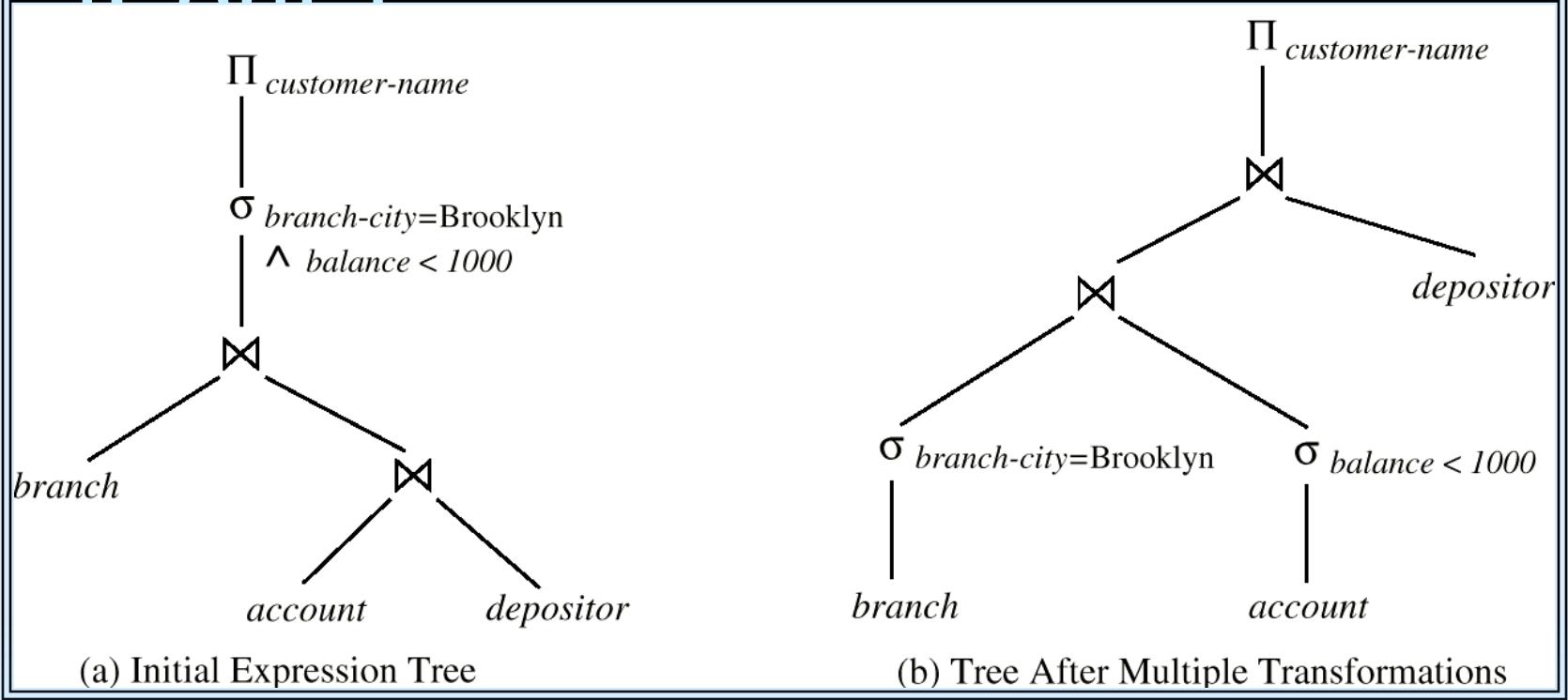
$$\Pi_{customer-name}((\sigma_{branch-city = "Brooklyn"} \wedge balance > 1000 \\ \bowtie \\ (branch \bowtie account \\ depositor)))$$

- Transformation using  $\bowtie$  join associatively (Rule 6a):

$$\Pi_{customer-name}((\sigma_{branch-city = "Brooklyn"} \wedge balance > 1000 \\ (branch \bowtie account))$$

# Multiple Transformations

(Cont.)



# Projection Operation Example

$\Pi_{customer-name}((\sigma_{branch-city} = "Brooklyn" (branch) \bowtie account) \bowtie depositor)$

- When we compute

$(\sigma_{branch-city} = "Brooklyn" (branch) \bowtie account)$

we obtain a relation whose schema is:

$(branch-name, branch-city, assets,$   
 $account-number, balance)$

- Push  $\bowtie$  projections using equivalence rules 8a and 8b; eliminate unneeded attributes from intermediate results to get:

$\Pi_{customer-name} (($

$\Pi_{account-number} ((\sigma_{branch-city} = "Brooklyn"$

# Join Ordering Example

- For all relations  $r_1, r_2$ , and  $r_3$ ,  
$$(r_1 \bowtie r_2) \bowtie r_3 = r_1 \bowtie (r_2 \bowtie r_3)$$
- If  $r_2 \bowtie r_3$  is quite large and  $r_1 \bowtie r_2$  is small, we choose

$$(r_1 \bowtie r_2) \bowtie r_3$$

so that we compute and store a smaller temporary relation.

# Join Ordering Example (Cont.)

- Consider the expression

$$\Pi_{\text{customer-name}} ((\sigma_{\text{branch-city} = \text{"Brooklyn"} (\text{branch})} \text{account} \bowtie \text{depositor})$$

- Could compute *account depositor* first, and join result with

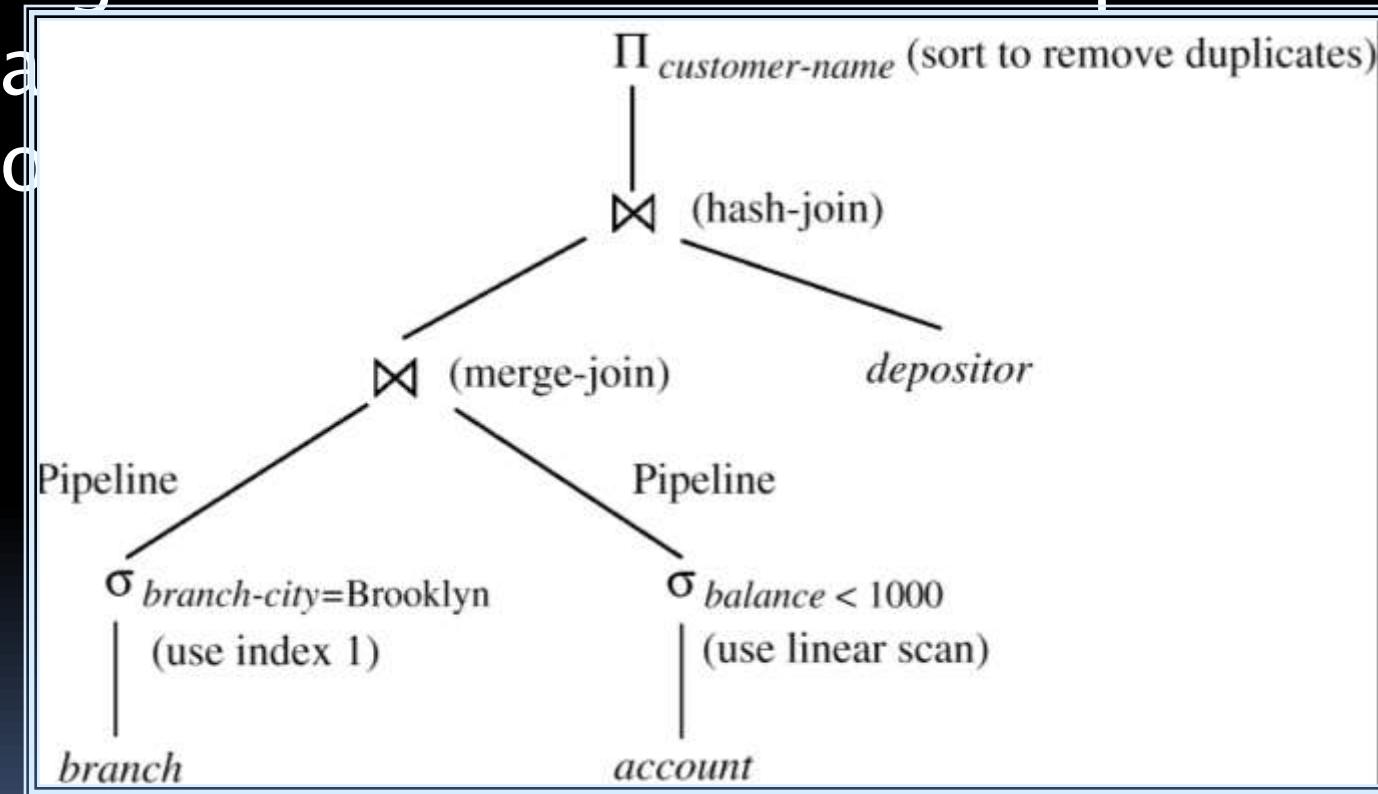
$\sigma_{\text{branch-city} = \text{"Brooklyn"} (\text{branch})}$   
but *account depositor* is likely  
to be a large relation.

# Enumeration of Equivalent

- Query optimizers use equivalence rules to systematically generate expressions equivalent to the given expression
- Conceptually, generate all equivalent expressions by repeatedly executing the following step until no more expressions can be found:
  - for each expression found so far, use all applicable equivalence rules, and add newly generated expressions to the set of expressions found so far
- The above approach is very expensive in space and time
- Space requirements reduced by sharing

# Evaluation Plan

- An evaluation plan defines exactly what algorithm is used for each operation,



# Choice of Evaluation Plans

- Must consider the interaction of evaluation techniques when choosing evaluation plans: choosing the cheapest algorithm for each operation independently may not yield best overall algorithm. E.g.
  - merge-join may be costlier than hash-join, but may provide a sorted output which reduces the cost for an outer level aggregation.
  - nested-loop join may provide opportunity for pipelining
- Practical query optimizers incorporate

# Cost-Based Optimization



- Consider finding the best join-order for  $r_1 \ r_2 \ \dots \ r_n$ .
- There are  $(2(n - 1))!/(n - 1)!$  different join orders for above expression.  
With  $n = 7$ , the number is 665280,  
with  $n = 10$ , the number is greater  
than 176 billion!
- No need to generate all the join  
orders. Using dynamic programming,  
the least-cost join order for any  
subset of

# Dynamic Programming in Optimization

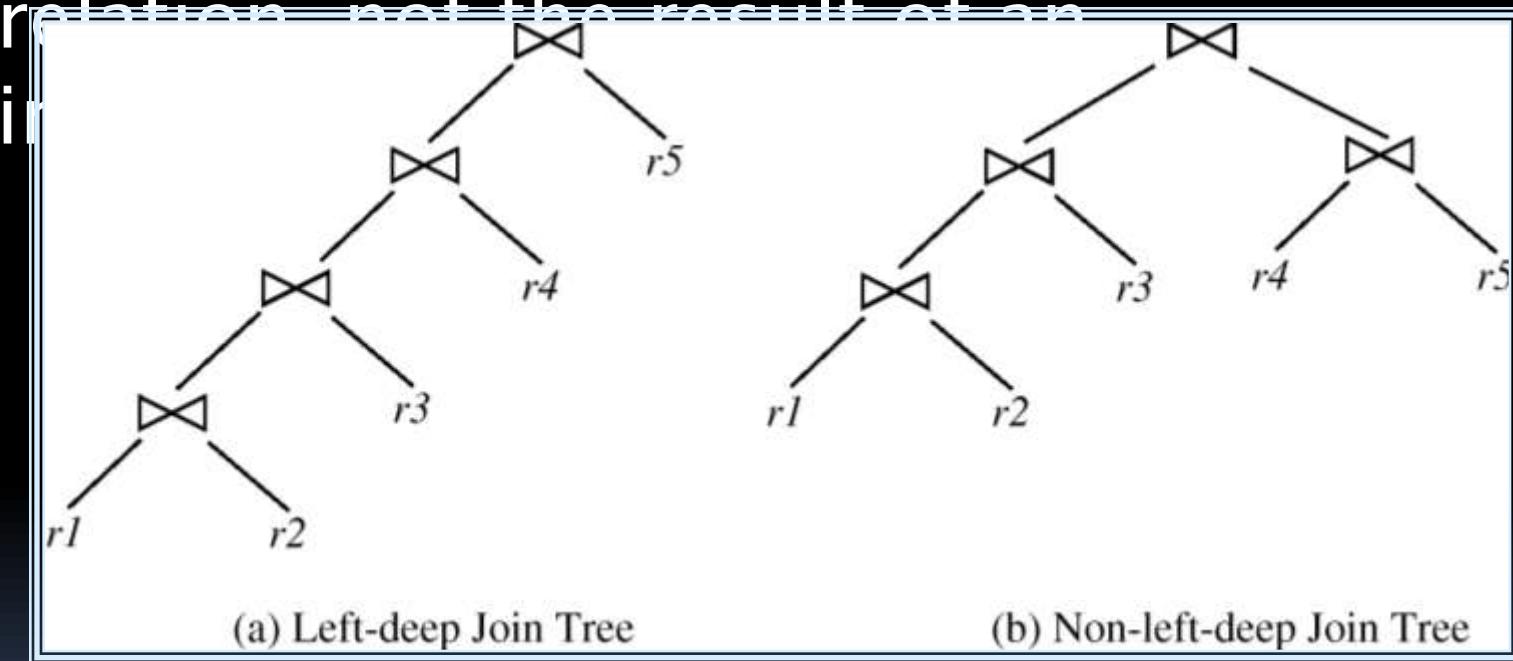
- To find best join tree for a set of  $n$  relations:
  - To find best plan for a set  $S$  of  $n$  relations, consider all possible plans of the form:  $S_1 \times (S - S_1)$  where  $S_1$  is any non-empty subset of  $S$ .
  - Recursively compute costs for joining subsets of  $S$  to find the cost of each plan. Choose the cheapest of the  $2^n - 1$  alternatives.
  - When plan for any subset is computed, store it and reuse it when it is required again, instead of recomputing it.

# Join Order Optimization

```
procedure findbestplan(S)
Algorithm
 if ($bestplan[S].cost \neq \infty$)
 return $bestplan[S]$
 // else $bestplan[S]$ has not been
 computed earlier, compute it now
 for each non-empty subset $S1$ of S such
 that $S1 \neq S$
 $P1 = findbestplan(S1)$
 $P2 = findbestplan(S - S1)$
 $A =$ best algorithm for joining results
 of $P1$ and $P2$
 $cost = P1.cost + P2.cost + \text{cost of } A$
 if $cost < bestplan[S].cost$
```

# Left Deep Join Trees

- In left-deep join trees, the right-hand-side input for each join is a relation not the result of another join.



# Cost of Optimization

- With dynamic programming time complexity of optimization with bushy trees is  $O(3^n)$ .
  - With  $n = 10$ , this number is 59000 instead of 176 billion!
- Space complexity is  $O(2^n)$
- To find best left-deep join tree for a set of  $n$  relations:
  - Consider  $n$  alternatives with one relation as right-hand side input and the other relations as left-hand side input.
  - Using (recursively computed and stored)

# Interesting Orders in Cost-Based Optimization

- Consider the expression  $r_1 \bowtie r_2 \bowtie r_3$   
 $r_4 \quad r_5$
- An interesting sort order is a particular sort order of tuples that could be useful for a later operation.
  - Generating the result of  $r_1 \bowtie r_2 \bowtie r_3$  sorted on the attributes common with  $r_4$  or  $r_5$  may be useful, but generating it sorted on the attributes common only  $r_1$  and  $r_2$  is not useful.
  - Using merge-join to compute  $r_1 \bowtie r_2 \bowtie r_3$  may be costlier, but may provide an output sorted in an interesting order.

# Heuristic Optimization

- Cost-based optimization is expensive, even with dynamic programming.
- Systems may use *heuristics* to reduce the number of choices that must be made in a cost-based fashion.
- Heuristic optimization transforms the query-tree by using a set of rules that typically (but not in all cases) improve execution performance:

# Steps in Typical Heuristic

1. Deconstruct conjunctive selections into a sequence of single selection operations (Equiv. rule 1.).
2. Move selection operations down the query tree for the earliest possible execution (Equiv. rules 2, 7a, 7b, 11).
3. Execute first those selection and join operations that will produce the smallest relations (Equiv. rule 6).
4. Replace Cartesian product operations that are followed by a selection condition by join operations (Equiv. rule 4a).

# Structure of Query Optimizers

- The System R/Starburst optimizer considers only left-deep join orders. This reduces optimization complexity and generates plans amenable to pipelined evaluation.  
System R/Starburst also uses heuristics to push selections and projections down the query tree.
- Heuristic optimization used in some versions of Oracle:
  - Repeatedly pick “best” relation to join

# Structure of Query Optimizers (Cont.)

Some query optimizers integrate heuristic selection and the generation of alternative access plans.

- System R and Starburst use a hierarchical procedure based on the nested-block concept of SQL: heuristic rewriting followed by cost-based join-order optimization.
- Even with the use of heuristics, cost-based query optimization imposes a substantial overhead.

- Optimizing Nested Subqueries
  - SQL conceptually treats nested subqueries in the where clause as functions that take parameters and return a single value or set of values
    - Parameters are variables from outer level query that are used in the nested subquery; such variables are called **correlation variables**
  - E.g.

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

# Optimizing Nested Subqueries

(Cont.)

- Correlated evaluation may be quite inefficient since
  - a large number of calls may be made to the nested query
  - there may be unnecessary random I/O as a result
- SQL optimizers attempt to transform nested subqueries to joins where possible, enabling use of efficient join techniques
- E.g.: earlier nested query can be rewritten as

# Optimizing Nested Subqueries

In general, SQL queries of the form below can be rewritten as shown  
*(Cont.)*

- Rewrite: select ...

from  $L_1$   
where  $P_1$  and exists (select \*  
                  from  $L_2$   
                  where  $P_2$ )

- To:

create table  $t_1$  as  
                  select distinct  $V$

from  $L_2$   
                  where  $P_2^1$

select ...

from  $L_1, t_1$

where  $P_1$  and  $P_2^2$

# Optimizing Nested Subqueries

- (Cont.) For example, the original nested query would be transformed to create table  $t_1$  as
  - select distinct *customer-name* from *depositor*
  - select *customer-name* from *borrower*,  $t_1$  where  $t_1.\text{customer-name} = \text{borrower.customer-name}$
- The process of replacing a nested query by a query with a join (possibly with a temporary relation) is called de-nesting.

# Materialized Views\*\*

- A **materialized view** is a view whose contents are computed and stored.
- Consider the view  
*create view branch-total-loan(branch-name, total-loan) as  
select branch-name, sum(amount)  
from loan  
groupby branch-name*
- Materializing the above view would be very useful if the total loan amount is required frequently

# Materialized View Maintenance

- The task of keeping a materialized view up-to-date with the underlying data is known as **materialized view maintenance**
- Materialized views can be maintained by recomputation on every update
- A better option is to use **incremental view maintenance**
  - Changes to database relations are used to compute changes to materialized view, which is then updated

# Incremental View Maintenance

- The changes (inserts and deletes) to a relation or expressions are referred to as its **differential**
  - Set of tuples inserted to and deleted from  $r$  are denoted  $i_r$  and  $d_r$
- To simplify our description, we only consider inserts and deletes
  - We replace updates to a tuple by deletion of the tuple followed by insertion of the update tuple
- We describe how to compute the

# Join Operation

$\bowtie$

- Consider the materialized view  $v = r \bowtie s$  and an update to  $r \bowtie s$
- Let  $r^{old}$  and  $r^{new}$  denote the old and new states of relation  $r \bowtie s$
- Consider the case of an insert to  $r$ :
  - We can write  $r^{new} \bowtie s$  as  $(r^{old} \cup i_r) \bowtie s$
  - And rewrite the above to  $(r^{old} \bowtie s) \cup (i_r \bowtie s)$
  - But  $(r^{old} \bowtie s)$  is simply the old value of the materialized view, so the incremental change to the view is just  $i_r \bowtie s$

# Selection and Projection

- Selection: Consider a view  $v = \sigma_{\theta}(r)$ .  
**Operations**
  - $v^{new} = v^{old} \cup \sigma_{\theta}(i_r)$
  - $v^{new} = v^{old} - \sigma_{\theta}(d_r)$
- Projection is a more difficult operation
  - $R = (A, B)$ , and  $r(R) = \{ (a, 2), (a, 3) \}$
  - $\Pi_A(r)$  has a single tuple  $(a)$ .
  - If we delete the tuple  $(a, 2)$  from  $r$ , we should not delete the tuple  $(a)$  from  $\Pi_A(r)$ , but if we then delete  $(a, 3)$  as well, we should delete the tuple
- For each tuple in a projection  $\Pi_A(r)$ , we will keep a count of how many times it was derived
  - On insert of a tuple to  $r$ , if the resultant

# Aggregation Operations

- $\text{count} : V = \text{Ag}_{\text{count}(B)}^{(n)}$ .
  - When a set of tuples  $i_r$  is inserted
    - For each tuple  $r$  in  $i_r$ , if the corresponding group is already present in  $V$ , we increment its count, else we add a new tuple with count = 1
  - When a set of tuples  $d_r$  is deleted
    - for each tuple  $t$  in  $i_r$ , we look for the group  $t.A$  in  $V$ , and subtract 1 from the count for the group.
    - If the count becomes 0, we delete from  $V$  the tuple for the group  $t.A$
- $\text{sum}: V = \text{Ag}_{\text{sum}(B)}^{(n)}$ 
  - We maintain the sum in a manner similar to count, except we add/subtract the  $B$  value instead of adding/subtracting 1 for the count

# Aggregate Operations (Cont.)

- **min, max:**  $v = Ag_{\min(B)}(r)$ .
  - Handling insertions on  $r$  is straightforward.
  - Maintaining the aggregate values **min** and **max** on deletions may be more expensive. We have to look at the other tuples of  $r$  that are in the same group to find the new minimum

# Other Operations

- Set intersection:  $v = r \cap s$ 
  - when a tuple is inserted in  $r$  we check if it is present in  $s$ , and if so we add it to  $v$ .
  - If the tuple is deleted from  $r$ , we delete it from the intersection if it is present.
  - Updates to  $s$  are symmetric
  - The other set operations, *union* and *set difference* are handled in a similar fashion.
- Outer joins are handled in much the same way as joins but with some extra

# Handling Expressions

- To handle an entire expression, we derive ~~expressions~~ for computing the incremental change to the result of each sub-expressions, starting from the smallest sub-expressions.
- E.g. consider  $E_1 \quad E_2$  where each of  $E_1$  and  $E_2$  may be a complex expression
  - Suppose the set of tuples to be inserted into  $E_1$  is given by  $D_1$ 
    - Computed earlier, since smaller sub-

# Query Optimization and Materialized Views

- Rewriting queries to use materialized views:
  - A materialized view  $v = r \bowtie s$  is available
  - A user submits a query  $r \bowtie s \bowtie t$
  - We can rewrite the query as  $v \bowtie t$ 
    - Whether to do so depends on cost estimates for the two alternative
- Replacing a use of a materialized view by the view definition:
  - A materialized view  $v = r \bowtie s$  is available, but without any index on it
  - User submits a query  $\sigma_{A=10}(v)$ .

# Materialized View Selection

- Materialized view selection: “What is the best set of views to materialize?”.
  - This decision must be made on the basis of the system **workload**
- Indices are just like materialized views, problem of **index selection** is closely related, to that of materialized view selection, although it is simpler.
- Some database systems, provide tools to help the database administrator with index and materialized view

# END OF CHAPTER

(Extra slides with details of selection cost  
estimation follow)

# Selection Cost Estimate Example

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

- Number of blocks is  $b_{account} = 500$ : 10,000 tuples in the relation; each block holds 20 tuples.
- Assume  $account$  is sorted on  $branch-name$ .
  - $V(branch-name, account)$  is 50
  - $10000/50 = 200$  tuples of the  $account$  relation pertain to Perryridge branch
  - $200/20 = 10$  blocks for these

# Selections Using Indices

- Index scan - search algorithms that use an index; condition is on search-key of index.
- A3 (*primary index on candidate key, equality*). Retrieve a single record that satisfies the corresponding equality condition  $E_{A3} = HT_i + 1$
- A4 (*primary index on nonkey, equality*) Retrieve multiple records. Let the search-key attribute be  $A$ .

# Cost Estimate Example (Indices)

Consider the query is  $\sigma_{branch-name = \text{``Perryridge''}}(account)$ , with the primary index on *branch-name*.

- Since  $V(branch-name, account) = 50$ , we expect that  $10000/50 = 200$  tuples of the *account* relation pertain to the Perryridge branch.
- Since the index is a clustering index,  $200/20 = 10$  block reads are required to read the *account* tuples.
- Several index blocks must also be read to support the query.

# Selections Involving Comparisons

selections of the form  $\sigma_{A \leq Y}(r)$  or  $\sigma_{A \geq Y}(r)$  by using a linear file scan or binary search, or by using indices in the following ways:

- A6 (*primary index, comparison*).  
The cost estimate is:

where  $c$  is the estimated number of tuples satisfying the condition. In absence of statistical information  $c$  is assumed to be  $n_r/2$ .

# Example of Cost Estimate for Complex Selection

- Consider a selection on *account* with the following condition: where *branch-name* = “Perryridge” and *balance* = 1200
- Consider using algorithm A8:
  - The *branch-name* index is clustering, and if we use it the cost estimate is 12 block reads (as we saw before).
  - The *balance* index is non-clustering, and  $V(balance, account = 500)$ , so the selection would retrieve  $10,000/500 = 20$  accounts. Adding the index block

# Example (Cont.)

- Consider using algorithm A10:
  - Use the index on *balance* to retrieve set  $S_1$  of pointers to records with *balance* = 1200.
  - Use index on *branch-name* to retrieve-set  $S_2$  of pointers to records with *branch-name* = “Perryridge”.
  - $S_1 \cap S_2$  = set of pointers to records with *branch-name* = “Perryridge” and *balance* = 1200.
  - The number of pointers retrieved (20 and 200), fit into a single leaf page; we read four index blocks to retrieve the two sets of pointers and

# Chapter 15: Transactions

- Transaction Concept
- Transaction State
- Implementation of Atomicity and Durability
- Concurrent Executions
- Serializability
- Recoverability
- Implementation of Isolation
- Transaction Definition in SQL
- Testing for Serializability.

# 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 inconsistent.
- When the transaction is committed, the database must be consistent.
- Two main issues to deal with:
  - Failures of various kinds, such as hardware failures and system crashes
  - Concurrent execution of multiple transactions

# ACID Properties

To preserve 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.
- **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.
  - That is, for every pair of transactions  $T_i$  and  $T_j$ , it appears to  $T_i$  that either  $T_j$  finished execution before  $T_i$  started, or  $T_j$  started execution after  $T_i$  finished.
- **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 \$50 from account  $A$  to account  $B$ :
  1. **read( $A$ )**
  2.  $A := A - 50$
  3. **write( $A$ )**
  4. **read( $B$ )**
  5.  $B := B + 50$
  6. **write( $B$ )**
- Consistency requirement – the sum of  $A$  and  $B$  is unchanged by the execution of the transaction.
- 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.

# Example of Fund Transfer

(Cont.)

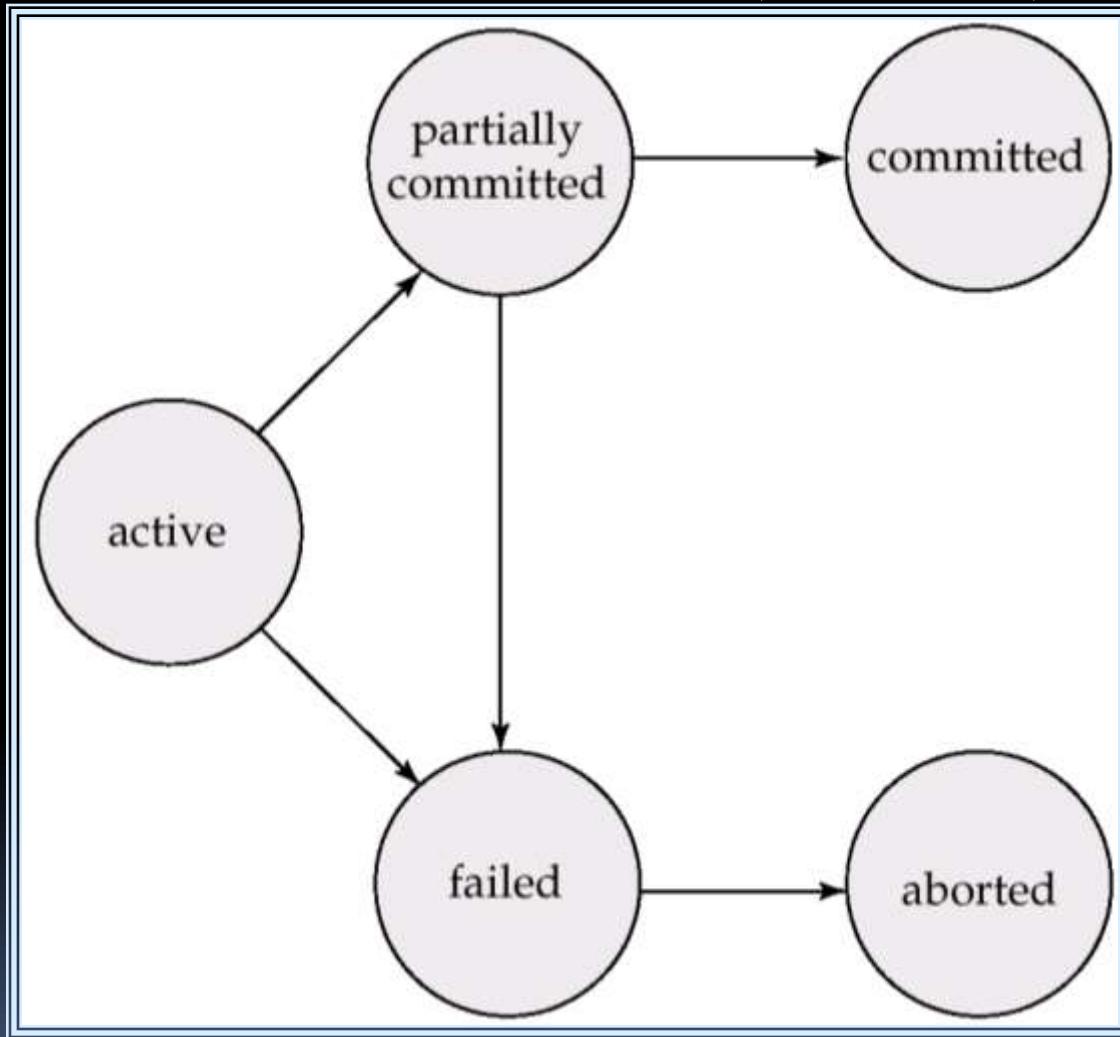
Durability requirement — once the user has been notified that the transaction has completed (i.e., the transfer of the \$50 has taken place), the updates to the database by the transaction must persist despite failures.

- 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). 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.

# 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 – only if no internal logical error
  - kill the transaction
- Committed, after *successful completion*.

# Transaction State (Cont.)



# Implementation of Atomicity

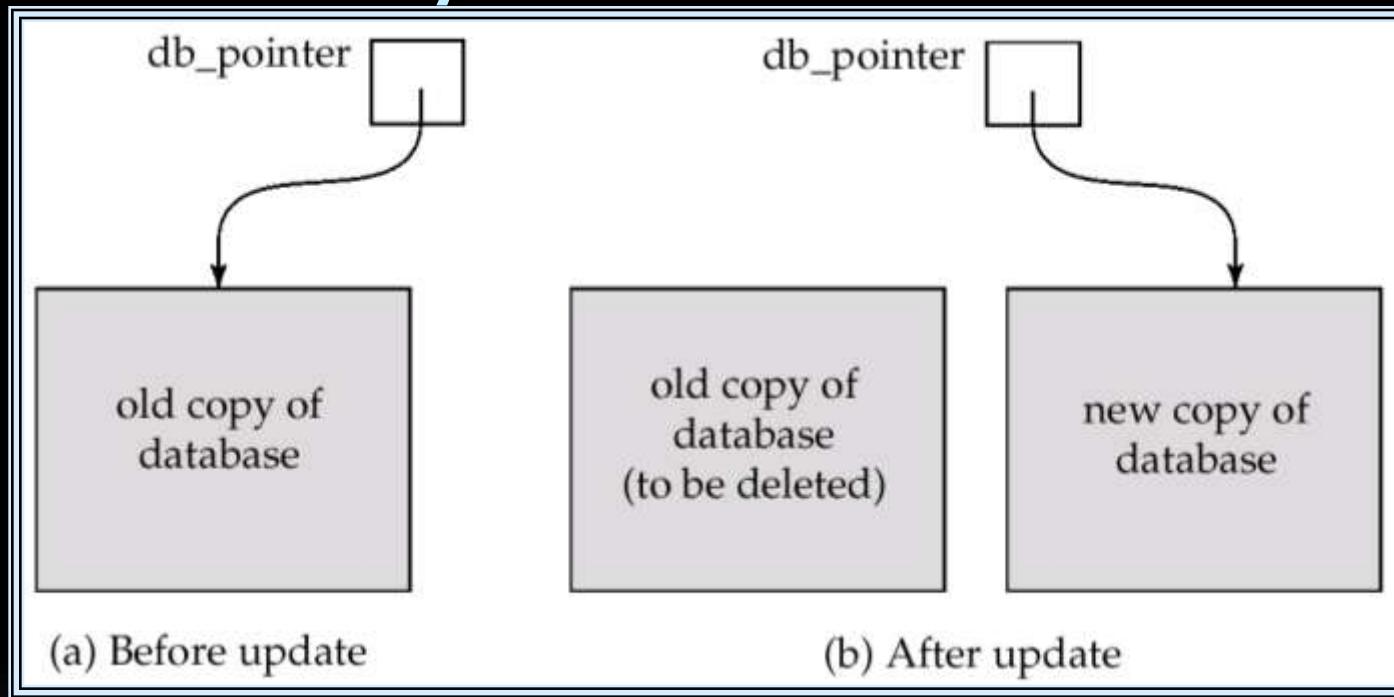
The recoverability-management

component of a database system  
implements the support for  
atomicity and durability.

- The *shadow-database* scheme:
  - assume that only one transaction is active at a time.
  - a pointer called db\_pointer always points to the current consistent copy of the database.
  - all updates are made on a *shadow copy* of the database, and db\_pointer is made to point to the updated

# Implementation of Atomicity and Durability (Cont.)

The shadow-database scheme:



- Assumes disks to not fail
- Useful for text editors, but extremely inefficient for large databases: executing a single transaction requires copying the *entire* database

# Concurrent Executions

- Multiple transactions are allowed to run concurrently in the system.  
Advantages are:
  - increased processor and disk utilization, leading to better transaction *throughput*: one transaction can be using the CPU while another is reading from or writing to the disk
  - reduced average response time for transactions: short transactions need not wait behind long ones.
- *Concurrency control schemes* – mechanisms to achieve isolation, i.e. to control the interaction

# Schedules

- *Schedules* – sequences that indicate the chronological order in which instructions of concurrent transactions are executed
  - a schedule for a set of transactions must consist of all instructions of those transactions
  - must preserve the order in which the instructions appear in each individual transaction.

# Example Schedules

Let  $T_1$  transfer \$50 from  $A$  to  $B$ ,  
and  $T_2$  transfer 10% of the  
balance from  $A$  to  $B$ . The  
following is a serial schedule

(Schedule  
 $T_1$  is fo

n which

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

# Example Schedule (Cont.)

Let  $T_1$  and  $T_2$  be the transactions defined previously.

The following schedule (Schedule 3 in the text) is not a serial schedule, but it is *equivalent* to Schedule 1.

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

In both Schedule 1 and 3, the sum  $A + B$  is preserved.

# Example Schedules (Cont.)

The following concurrent schedule (Schedule 4 in the text)

does not have a value of  
the shared variable  $A$ .

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

# Serializability

- Basic Assumption – Each transaction preserves database consistency.
- Thus serial execution of a set of transactions preserves database consistency.
- A (possibly concurrent) schedule is serializable if it is equivalent to a serial schedule. Different forms of schedule equivalence give rise to the notions of:
  1. conflict serializability
  2. view serializability

# Conflict Serializability

- Instructions  $I_i$  and  $I_j$  of transactions  $T_i$  and  $T_j$  respectively, conflict if and only if there exists some item  $Q$  accessed by both  $I_i$  and  $I_j$ , and at least one of these instructions wrote  $Q$ .
  1.  $I_i = \text{read}(Q)$ ,  $I_j = \text{read}(Q)$ .  $I_i$  and  $I_j$  don't conflict.
  2.  $I_i = \text{read}(Q)$ ,  $I_j = \text{write}(Q)$ . They conflict.
  3.  $I_i = \text{write}(Q)$ ,  $I_j = \text{read}(Q)$ . They conflict

# Conflict Serializability (Cont.)

- If a schedule  $S$  can be transformed into a schedule  $S'$  by a series of swaps of non-conflicting instructions, we say that  $S$  and  $S'$  are conflict equivalent.
- We say that a schedule  $S$  is conflict serializable if it is conflict equivalent to a serial schedule
- Example of a schedule that is not conflict serializable:

$T_3 \quad T_4$   
read( $Q$ )

# Conflict Serializability (Cont.)

- Schedule 3 below can be transformed into Schedule 1, a serial schedule where  $T_2$  follows  $T_1$ , by series of conflict-free operations.  
Schedule 3:  

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

Therefore serializable.

## View Serializability

- Let  $S$  and  $S'$  be two schedules with the same set of transactions.  $S$  and  $S'$  are view equivalent if the following three conditions are met:
  - For each data item  $Q$ , if transaction  $T_i$  reads the initial value of  $Q$  in schedule  $S$ , then transaction  $T_i$  must, in schedule  $S'$ , also read the initial value of  $Q$ .
  - For each data item  $Q$  if transaction  $T_i$  executes **read**( $Q$ ) in schedule  $S$ , and that value was produced by transaction  $T_j$  (if any), then transaction  $T_i$  must in schedule  $S'$  also read the value of  $Q$  that was produced by transaction  $T_j$ .
  - For each data item  $Q$ , the transaction (if any) that performs the final **write**( $Q$ )

# View Serializability (Cont.)

- A schedule  $S$  is view serializable if it is view equivalent to a serial schedule.
- Every conflict serializable schedule is also view serializable.

- Schedule which is not conflict serializable

	$T_3$	$T_4$	$T_6$
read( $Q$ )		write( $Q$ )	
write( $Q$ )			write( $Q$ )

# Other Notions of Serializability

Schedule 8 (from text) given below produces same outcome

as the  
yet is  
view e

$T_1$	$T_5$
read( $A$ )	
$A := A - 50$	
write( $A$ )	
	read( $B$ )
	$B := B - 10$
	write( $B$ )
read( $B$ )	
$B := B + 50$	
write( $B$ )	
	read( $A$ )
	$A := A + 10$
	write( $A$ )

$\langle T_1, T_5 \rangle$ ,  
valent or

# Recoverability

Need to address the effect of transaction failures on concurrently running transactions.

- Recoverable schedule — if a transaction  $T_j$  reads a data items previously written by a transaction  $T_i$ , the commit operation of  $T_i$  appears before the commit operation of  $T_j$ .
- The following schedule is not recoverable (Schedule 11)
  - commutes
  - immediately after the read

$T_8$	$T_9$
read( $A$ )	
write( $A$ )	
read( $B$ )	read( $A$ )

# Recoverability (Cont.)

- Cascading rollback – a single transaction failure leads to a series of transaction rollbacks. Consider the following timeline of the committal schedule where none of the transactions have yet recovered:
  - $T_{10}$ : read( $A$ ), read( $B$ ), write( $A$ )
  - $T_{11}$ : read( $A$ ), write( $A$ )
  - $T_{12}$ : read( $A$ )

$T_{10}$	$T_{11}$	$T_{12}$
read( $A$ )		
read( $B$ )		
write( $A$ )		
	read( $A$ )	
	write( $A$ )	
		read( $A$ )

# Recoverability (Cont.)

- Cascadeless schedules — cascading rollbacks cannot occur; for each pair of transactions  $T_i$  and  $T_j$  such that  $T_j$  reads a data item previously written by  $T_i$ , the commit operation of  $T_i$  appears before the read operation of  $T_j$ .
- Every cascadeless schedule is also recoverable
- It is desirable to restrict the schedules to those that are cascadeless

# Implementation of Isolation

- Schedules must be conflict or view serializable, and recoverable, for the sake of database consistency, and preferably cascadeless.
- A policy in which only one transaction can execute at a time generates serial schedules, but provides a poor degree of concurrency..
- Concurrency-control schemes tradeoff between the amount of concurrency they allow and the

# Transaction Definition in SQL

- Data manipulation language must include a construct for specifying the set of actions that comprise a transaction.
- In SQL, a transaction begins implicitly.
- A transaction in SQL ends by:
  - **Commit work** commits current transaction and begins a new one.
  - **Rollback work** causes current transaction to abort.
- Levels of consistency specified by SQL-92:
  - **Serializable** – default

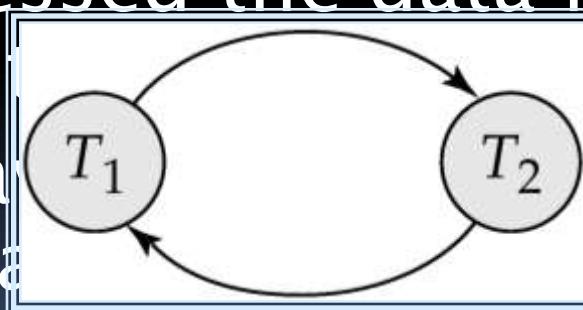
# Levels of Consistency in SQL-92

- Serializable — default
- Repeatable read — only committed records to be read, repeated reads of same record must return same value. However, a transaction may not be serializable – it may find some records inserted by a transaction but not find others.

Lower degrees of consistency useful for gathering approximate information about the database, e.g., statistics for query optimizer.  
**Read committed** only committed records can be read, but successive reads of record may return different (but committed) values.

# Testing for Serializability

- Consider some schedule of a set of transactions  $T_1, T_2, \dots, T_n$
- Precedence graph — a directed graph where the vertices are the transactions (names).
- We draw an arc from  $T_i$  to  $T_j$  if  $T_i$  accessed the data item on which  $T_j$  depends earlier.
- We make sure that we do not have cycles by the item that was modified.
- Example 1

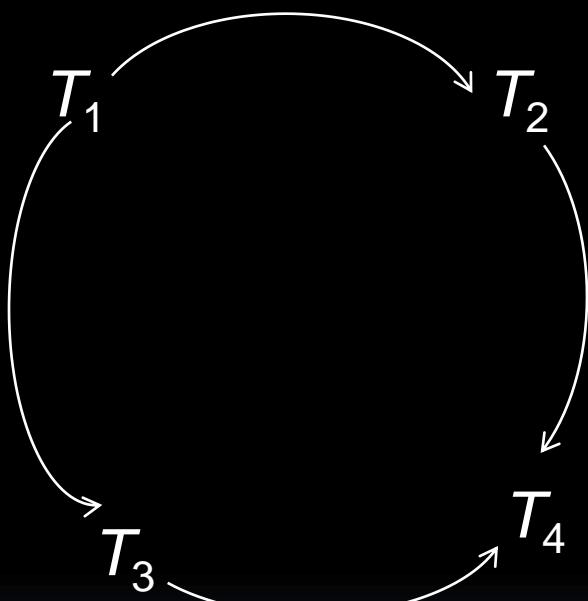


# Example Schedule (Schedule A)

	$T_1$	$T_2$	$T_3$	$T_4$	$T_5$
		read(X)			
	read(Y)				read(V)
	read(Z)				read(W)
					read(W)
		read(Y)			
		write(Y)			
			write(Z)		
	read(U)				read(Y)

# Precedence Graph for Schedule

A



# Test for Conflict Serializability

- A schedule is conflict serializable if and only if its precedence graph is acyclic.
- Cycle-detection algorithms exist which take order  $n^2$  time, where  $n$  is the number of vertices in the graph. (Better algorithms take order  $n + e$  where  $e$  is the number of edges.)
- If precedence graph is acyclic, the serializability order can be obtained by a *topological sorting* of the graph

# Test for View Serializability

- The precedence graph test for conflict serializability must be modified to apply to a test for view serializability.
- The problem of checking if a schedule is view serializable falls in the class of *NP*-complete problems. Thus existence of an efficient algorithm is unlikely.

However practical algorithms that just check some *sufficient conditions* for view serializability can still be used.

# Concurrency Control vs. Serializability Tests

- Testing a schedule for serializability *after* it has executed is a little too late!
- Goal – to develop concurrency control protocols that will assure serializability. They will generally not examine the precedence graph as it is being created; instead a protocol will impose a discipline that avoids nonserializable schedules.  
Will study such protocols in Chapter 16.

END OF CHAPTER



# Schedule 2 -- A Serial Schedule in Which $T_2$ is Followed by $T_1$

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

# Schedule 5 -- Schedule 3 After Swapping A Pair of Instructions

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

# Schedule 6 -- A Serial Schedule That is Equivalent to Schedule 3

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

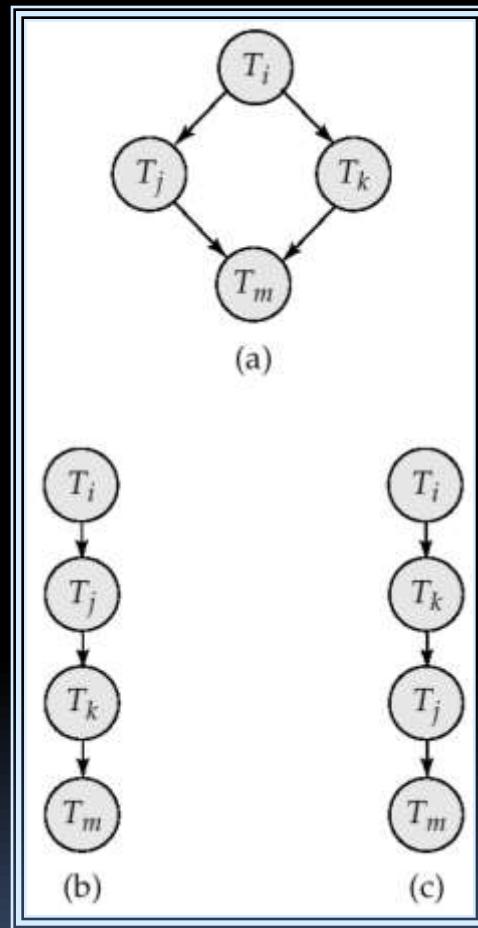
# Schedule 7

$T_3$	$T_4$
<b>read(<math>Q</math>)</b>	
<b>write(<math>Q</math>)</b>	<b>write(<math>Q</math>)</b>

# Precedence Graph for (a) Schedule 1 and (b) Schedule 2



# Illustration of Topological Sorting



# Precedence Graph

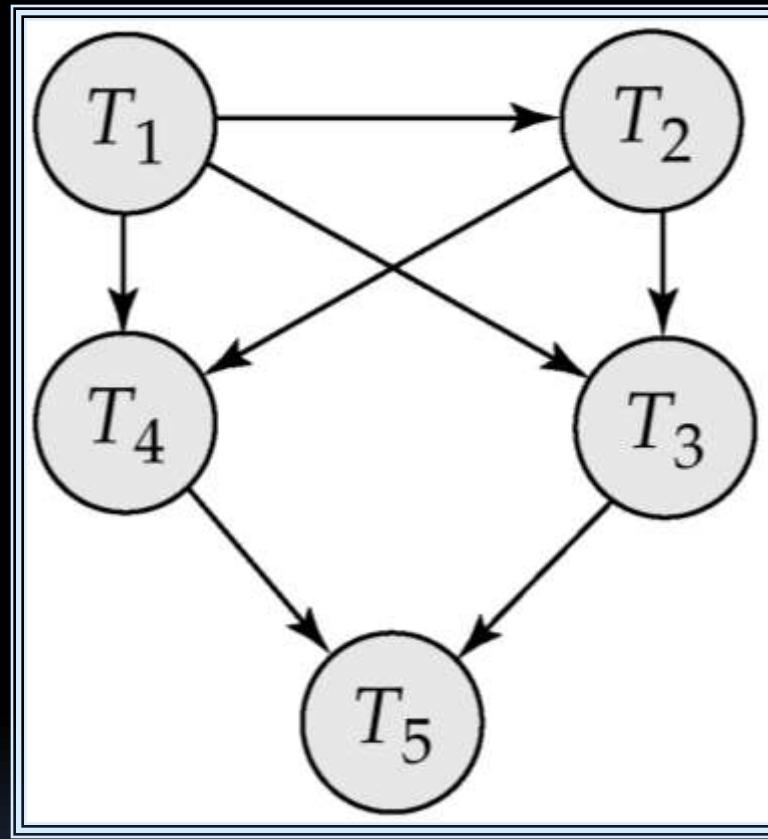


fig. 15.21

$T_3$	$T_4$	$T_7$
read( $Q$ )	write( $Q$ )	
write( $Q$ )		read( $Q$ ) write( $Q$ )

# Chapter 16: Concurrency

## Control

- Lock-Based Protocols
- Timestamp-Based Protocols
- Validation-Based Protocols
- Multiple Granularity
- Multiversion Schemes
- Deadlock Handling
- Insert and Delete Operations
- Concurrency in Index Structures

# Lock-Based Protocols

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

# Lock-Based Protocols (Cont.)

- Lock-compatibility matrix

	S	X
S	true	false
X	false	false

- A transaction may be granted a lock on an item if the requested lock is compatible with locks already held on the item by other transactions
- Any number of transactions can hold shared locks on an item, but if any transaction holds an exclusive on the item no other transaction may hold

# Lock-Based Protocols (Cont.)

- Example of a transaction performing locking:

```
 T_2 : lock-S(A);
 read (A);
 unlock(A);
 lock-S(B);
 read (B);
 unlock(B);
 display($A+B$)
```

- Locking as above is not sufficient to guarantee serializability — if  $A$  and  $B$  get updated in-between the read of  $A$  and  $B$ , the displayed sum would be wrong.
- A locking protocol is a set of rules followed by all transactions while requesting and releasing locks. Locking protocols restrict the set of possible schedules.

# Pitfalls of Lock-Based Protocols

Consider the partial schedule

$T_3$	$T_4$
lock-X( $B$ ) read( $B$ ) $B := B - 50$ write( $B$ )    lock-X( $A$ )	lock-S( $A$ ) read( $A$ ) lock-S( $B$ )

- Neither  $T_3$  nor  $T_4$  can make progress — executing lock-S( $B$ ) causes  $T_4$  to wait for  $T_3$  to release its lock on  $B$ , while executing lock-X( $A$ ) causes  $T_3$  to wait for  $T_4$  to release its lock on  $A$ .
- Such a situation is called a deadlock.
  - To handle a deadlock one of  $T_3$  or  $T_4$  must be rolled back and its locks released.

# Pitfalls of Lock-Based Protocols (Cont.)

- The potential for deadlock exists in most locking protocols. Deadlocks are a necessary evil.
- Starvation is also possible if concurrency control manager is badly designed. For example:
  - A transaction may be waiting for an X-lock on an item, while a sequence of other transactions request and are granted an S-lock on the same item.
  - The same transaction is repeatedly rolled back due to deadlocks.

# The Two-Phase Locking Protocol

- This is a protocol which ensures conflict-serializable schedules.
- Phase 1: Growing Phase
  - transaction may obtain locks
  - transaction may not release locks
- Phase 2: Shrinking Phase
  - transaction may release locks
  - transaction may not obtain locks
- The protocol assures serializability. It can be proved that the transactions

# The Two-Phase Locking Protocol (Cont.)

- Two-phase locking *does not* ensure freedom from deadlocks
- Cascading roll-back is possible under two-phase locking. To avoid this, follow a modified protocol called strict two-phase locking. Here a transaction must hold all its exclusive locks till it commits/aborts.
- Rigorous two-phase locking is even stricter: here *all* locks are held till

# The Two-Phase Locking Protocol (Cont.)

- There can be conflict serializable schedules that cannot be obtained if two-phase locking is used.
- However, in the absence of extra information (e.g., ordering of access to data), two-phase locking is needed for conflict serializability in the following sense:

Given a transaction  $T_i$  that does not follow two-phase locking, we can find a transaction  $T_j$  that uses two-phase

# Lock Conversions

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

# Automatic Acquisition of Locks

- A transaction  $T_i$  issues the standard read/write instruction, without explicit locking calls.
- The operation  $\text{read}(D)$  is processed as:
  - if  $T_i$  has a lock on  $D$
  - then
    - $\text{read}(D)$
  - else
    - $\text{begin}$
    - if necessary wait
  - until no other

# Automatic Acquisition of Locks

- $\text{write}(D)$  is processed as:

if  $T_i$  has a lock-X on  $D$

then

$\text{write}(D)$

else

begin

if necessary wait until no other trans. has any lock on  $D$ ,

if  $T_i$  has a lock-S on  $D$

then

upgrade lock on  $D$  to lock-X

else

grant  $T_i$  a lock-X on  $D$

$\text{write}(D)$

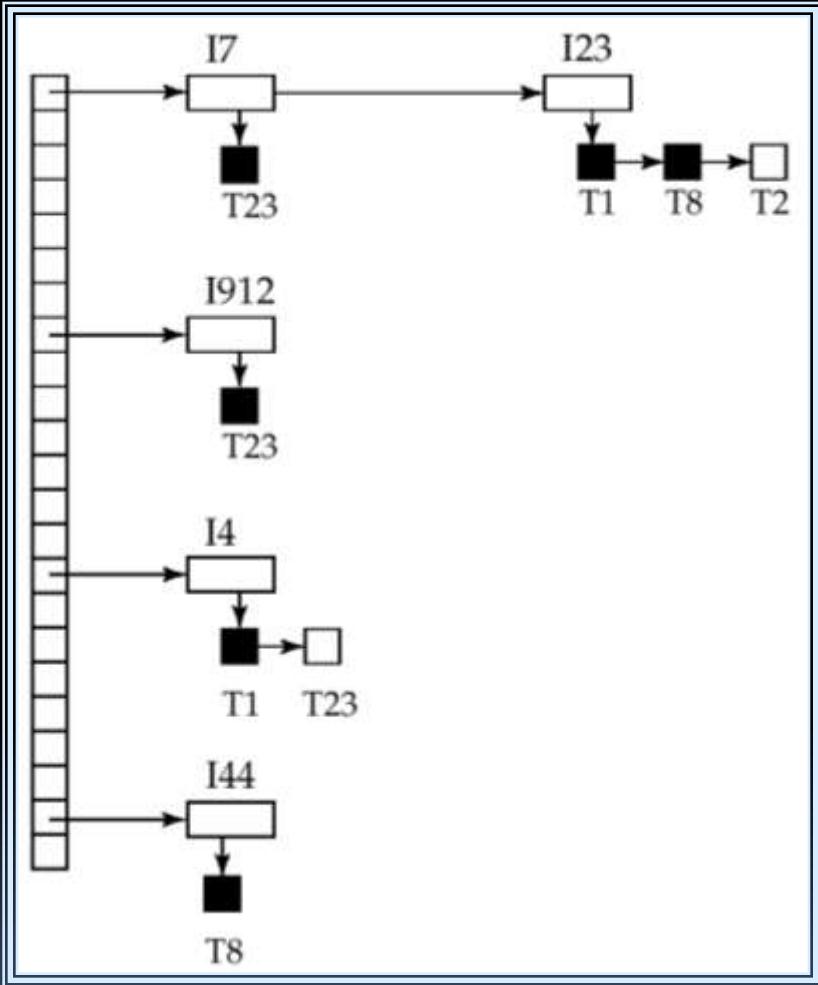
end;

- All locks are released after commit or abort

# Implementation of Locking

- A Lock manager can be implemented as a separate process to which transactions send lock and unlock requests
- The lock manager replies to a lock request by sending a lock grant messages (or a message asking the transaction to roll back, in case of a deadlock)
- The requesting transaction waits until its request is answered

# Lock Table

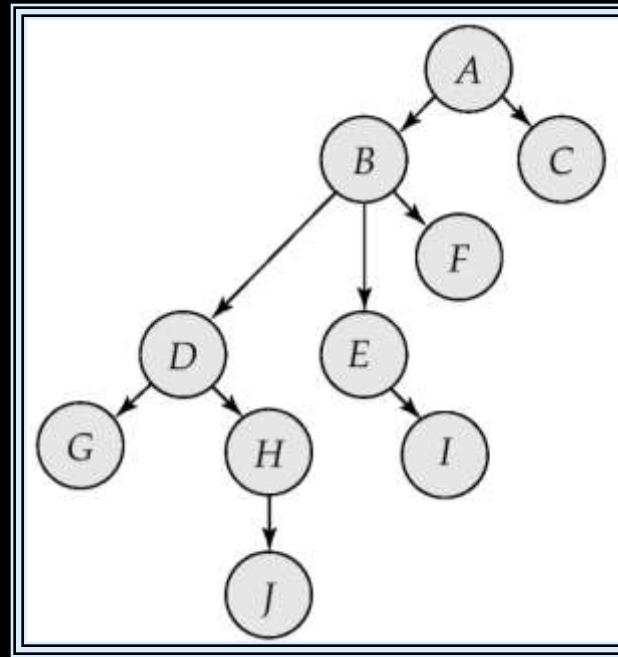


- Black rectangles indicate granted locks, white ones indicate waiting requests
- Lock table also records the type of lock granted or requested
- New request is added to the end of the queue of requests for the data item, and granted if it is compatible with all earlier locks
- Unlock requests result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

# Graph-Based Protocols

- Graph-based protocols are an alternative to two-phase locking
- Impose a partial ordering  $\rightarrow$  on the set  $D = \{d_1, d_2, \dots, d_h\}$  of all data items.
  - If  $d_i \rightarrow d_j$  then any transaction accessing both  $d_i$  and  $d_j$  must access  $d_i$  before accessing  $d_j$ .
  - Implies that the set D may now be viewed as a directed acyclic graph, called a *database graph*.
- The *tree-protocol* is a simple kind of graph protocol

# Tree Protocol



- Only exclusive locks are allowed.
- The first lock by  $T_i$  may be on any data item. Subsequently, a data  $Q$  can be locked by  $T_i$  only if the parent of  $Q$  is currently locked by  $T_i$ .

Data items may be unlocked at any

# Graph-Based Protocols (Cont.)

- The tree protocol ensures conflict serializability as well as freedom from deadlock.
- Unlocking may occur earlier in the tree-locking protocol than in the two-phase locking protocol.
  - shorter waiting times, and increase in concurrency
  - protocol is deadlock-free, no rollbacks are required
  - the abort of a transaction can still lead to cascading rollbacks.  
(this correction has to be made in the book also.)

# Timestamp-Based Protocols

- Each transaction is issued a timestamp when it enters the system. If an old transaction  $T_i$  has time-stamp  $\text{TS}(T_i)$ , a new transaction  $T_j$  is assigned time-stamp  $\text{TS}(T_j)$  such that  $\text{TS}(T_i) < \text{TS}(T_j)$ .
- The protocol manages concurrent execution such that the time-stamps determine the serializability order.
- In order to assure such behavior, the protocol maintains for each data  $Q$  two timestamp values:

# Timestamp-Based Protocols (Cont.)

- The timestamp ordering protocol ensures that any conflicting read and write operations are executed in timestamp order.
- Suppose a transaction  $T_i$  issues a  $\text{read}(Q)$ 
  - If  $\text{TS}(T_i) \leq \text{W-timestamp}(Q)$ , then  $T_i$  needs to read a value of  $Q$  that was already overwritten. Hence, the read operation is

# Timestamp-Based Protocols (Cont.)

- Suppose that transaction  $T_i$  issues  $\text{write}(Q)$ .
- If  $\text{TS}(T_i) < \text{R-timestamp}(Q)$ , then the value of  $Q$  that  $T_i$  is producing was needed previously, and the system assumed that that value would never be produced. Hence, the write operation is rejected, and  $T_i$  is rolled back.
- If  $\text{TS}(T_i) < \text{W-timestamp}(Q)$ , then  $T_i$  is attempting to write an obsolete value

# Example Use of the Protocol

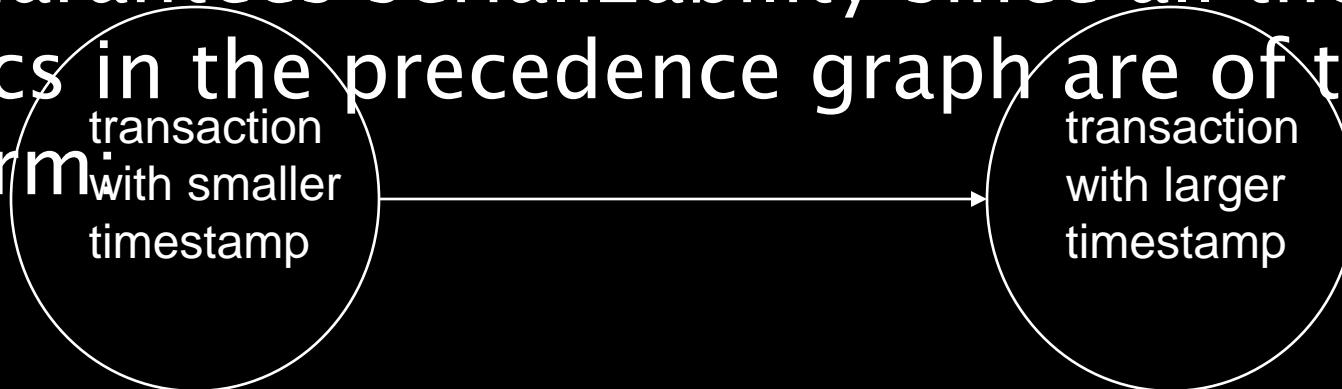
A partial schedule for several data items  
for transactions with  
timestamps  $\tau_1, \tau_2, \tau_3, \tau_4, \tau_5$

	$\tau_1$	$\tau_2$	$\tau_3$	$\tau_4$	$\tau_5$
	read(Y)	read(Y)			read(X)
	read(X)	read(X) abort	write(Y) write(Z)		read(Z)
			write(Z) abort		write(Y) write(Z)

# Correctness of Timestamp-

## Ordering Protocol

The timestamp-ordering protocol guarantees serializability since all the arcs in the precedence graph are of the form:



Thus, there will be no cycles in the precedence graph

Timestamp protocol ensures freedom

# Recoverability and Cascade Freedom

- Problem with timestamp-ordering protocol:
  - Suppose  $T_i$  aborts, but  $T_j$  has read a data item written by  $T_i$
  - Then  $T_j$  must abort; if  $T_j$  had been allowed to commit earlier, the schedule is not recoverable.
  - Further, any transaction that has read a data item written by  $T_j$  must abort
  - This can lead to cascading rollback --- that is, a chain of rollbacks

Solution:

# Thomas' Write Rule

- Modified version of the timestamp-ordering protocol in which obsolete write operations may be ignored under certain circumstances.
- When  $T_i$  attempts to write data item  $Q$ , if  $\text{TS}(T_i) < \text{W-timestamp}(Q)$ , then  $T_i$  is attempting to write an obsolete value of  $\{Q\}$ . Hence, rather than rolling back  $T_i$  as the timestamp ordering protocol would have done, this {write} operation can be ignored.

# Validation-Based Protocol

- Execution of transaction  $T_i$  is done in three phases.

1. Read and execution phase:

Transaction  $T_i$  writes only to temporary local variables

2. Validation phase: Transaction  $T_i$  performs a ``validation test''

to determine if local variables can be written without violating serializability.

3. Write phase: If  $T_i$  is validated, the updates are applied to the database; otherwise,  $T_i$  is rolled back.

The three phases of concurrently executing transactions can be

# Validation-Based Protocol (Cont.)

- Each transaction  $T_i$  has 3 timestamps
  - $\text{Start}(T_i)$  : the time when  $T_i$  started its execution
  - $\text{Validation}(T_i)$ : the time when  $T_i$  entered its validation phase
  - $\text{Finish}(T_i)$  : the time when  $T_i$  finished its write phase
- Serializability order is determined by timestamp given at validation time, to increase concurrency. Thus  $TS(T_i)$  is given the value of  $\text{Validation}(T_i)$

# Validation Test for Transaction

$T_j$

- If for all  $T_i$  with  $\text{TS}(T_i) < \text{TS}(T_j)$  either one of the following condition holds:
  - $\text{finish}(T_i) < \text{start}(T_j)$
  - $\text{start}(T_j) < \text{finish}(T_i) < \text{validation}(T_j)$  and the set of data items written by  $T_i$  does not intersect with the set of data items read by  $T_j$ .

then validation succeeds and  $T_j$  can be committed. Otherwise, validation fails and  $T_j$  is aborted.

■ *Justification:* Either first condition is

# Schedule Produced by Validation

- Example of schedule produced using validation

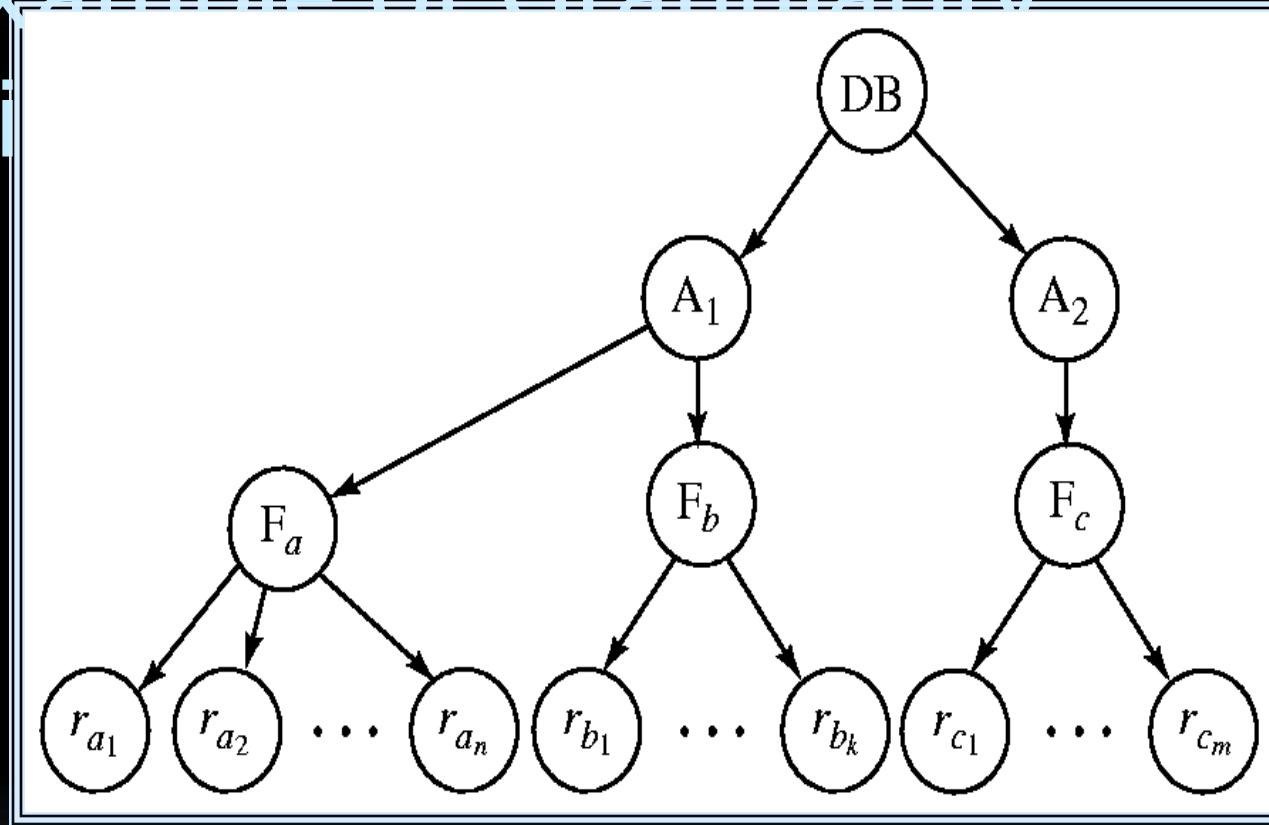
$T_{14}$	$T_{15}$
<b>read(<math>B</math>)</b>	<b>read(<math>B</math>)</b> $B \leftarrow B - 50$
<b>read(<math>A</math>)</b> <i>(validate)</i> <b>display (<math>A+B</math>)</b>	<b>read(<math>A</math>)</b> $A \leftarrow A + 50$  <i>(validate)</i> <b>write (<math>B</math>)</b> <b>write (<math>A</math>)</b>

# Multiple Granularity

- Allow data items to be of various sizes and define a hierarchy of data granularities, where the small granularities are nested within larger ones
- Can be represented graphically as a tree (but don't confuse with tree-locking protocol)
- When a transaction locks a node in the tree *explicitly*, it *implicitly* locks all the node's descendants in the same mode.

# Example of Granularity

Hi



The highest level in the example

# Intention Lock Modes

- In addition to S and X lock modes, there are three additional lock modes with multiple granularity:
  - *intention-shared* (IS): indicates explicit locking at a lower level of the tree but only with shared locks.
  - *intention-exclusive* (IX): indicates explicit locking at a lower level with exclusive or shared locks
  - *shared and intention-exclusive* (SIX): the subtree rooted by that node is locked explicitly in shared mode and explicit

# Compatibility Matrix with Intention Lock Modes

- The compatibility matrix for all lock modes is:

	IS	IX	S	S IX	X
IS	✓	✓	✓	✓	✗
IX	✓	✓	✗	✗	✗
S	✓	✗	✓	✗	✗
S IX	✓	✗	✗	✗	✗
X	✗	✗	✗	✗	✗

- **Multiple Granularity Locking Scheme**
  - Transaction  $T_i$  can lock a node  $Q$ , using the following rules:
    1. The lock compatibility matrix must be observed.
    2. The root of the tree must be locked first, and may be locked in any mode.
    3. A node  $Q$  can be locked by  $T_i$  in S or IS mode only if the parent of  $Q$  is currently locked by  $T_i$  in either IX or IS mode.
    4. A node  $Q$  can be locked by  $T_i$  in X, SIX, or IX mode only if the parent of  $Q$  is currently locked by  $T_i$  in either IX or SIX mode.
    5.  $T_i$  can lock a node only if it has not previously unlocked any node (that is,  $T_i$  is two-phase).
    6.  $T_i$  can unlock a node  $Q$  only if none of the children of  $Q$  are currently locked by  $T_i$ .
  - Observe that locks are acquired in root-to-leaf order, whereas they are released in leaf-to-root order.

# Multiversion Schemes

- Multiversion schemes keep old versions of data item to increase concurrency.
  - Multiversion Timestamp Ordering
  - Multiversion Two-Phase Locking
- Each successful write results in the creation of a new version of the data item written.
- Use timestamps to label versions.
- When a  $\text{read}(Q)$  operation is issued, select an appropriate version of  $Q$ .

# Multiversions Timestamp Ordering

- Each data item  $Q$  has a sequence of versions  $\langle Q_1, Q_2, \dots, Q_m \rangle$ . Each version  $Q_k$  contains three data fields:
  - **Content** -- the value of version  $Q_k$ .
  - **W-timestamp( $Q_k$ )** -- timestamp of the transaction that created (wrote) version  $Q_k$
  - **R-timestamp( $Q_k$ )** -- largest timestamp of a transaction that successfully read version  $Q_k$
- when a transaction  $T_i$  creates a new version  $Q_k$  of  $Q$ ,  $Q_k$ 's W-timestamp and R-timestamp are initialized to  $TS(T)$

# Multiversions Timestamp

The multiversions timestamp scheme presented next ensures serializability.

- Suppose that transaction  $T_i$  issues a  $\text{read}(Q)$  or  $\text{write}(Q)$  operation. Let  $Q_k$  denote the version of  $Q$  whose write timestamp is the largest write timestamp less than or equal to  $\text{TS}(T_i)$ .
  1. If transaction  $T_i$  issues a  $\text{read}(Q)$ , then the value returned is the content of version  $Q_k$ .
  2. If transaction  $T_i$  issues a  $\text{write}(Q)$ , and if  $\text{TS}(T_i) < \text{R-}$  timestamp( $Q$ ), then transaction  $T_i$  is

# Multiversioп Two-Phase

- **Locking** differentiates between read-only transactions and update transactions
- *Update transactions* acquire read and write locks, and hold all locks up to the end of the transaction. That is, update transactions follow rigorous two-phase locking.
  - Each successful **write** results in the creation of a new version of the data item written.
  - each version of a data item has a single timestamp whose value is obtained from a counter  $c$ , counter that is incremented

# Multiversions Two-Phase

## Locking (Cont.)

- When an update transaction wants to read a data item, it obtains a shared lock on it, and reads the latest version.
- When it wants to write an item, it obtains X lock on; it then creates a new version of the item and sets this version's timestamp to  $\infty$ .
- When update transaction  $T_i$  completes, commit processing occurs:
  - $T_i$  sets timestamp on the versions it has created to **ts-counter + 1**
  - $T_i$  increments **ts-counter** by 1
- Read-only transactions that start after  $T_i$  increments **ts-counter** will see the

# Deadlock Handling

- Consider the following two transactions:

$T_1$ :	write ( $X$ )	$T_2$ :
write( $Y$ )	$T_1$	$T_2$
	lock-X on $X$	lock-X on $Y$
write( $X$ )	write ( $Y$ )	write ( $X$ )
Schedule with deadlock		wait for lock-X on $X$
		wait for lock-X on $Y$

# Deadlock Handling

- System is deadlocked if there is a set of transactions such that every transaction in the set is waiting for another transaction in the set.
- *Deadlock prevention* protocols ensure that the system will *never* enter into a deadlock state. Some prevention strategies :
  - Require that each transaction locks all its data items before it begins execution (predeclaration).

# More Deadlock Prevention

## Strategies

- Following schemes use transaction timestamps for the sake of deadlock prevention alone.
- wait-die scheme — non-preemptive
  - older transaction may wait for younger one to release data item. Younger transactions never wait for older ones; they are rolled back instead.
  - a transaction may die several times before acquiring needed data item
- wound-wait scheme — preemptive
  - older transaction *wounds* (forces rollback)

# Deadlock prevention (Cont.)

- Both in *wait-die* and in *wound-wait* schemes, a rolled back transaction is restarted with its original timestamp. Older transactions thus have precedence over newer ones, and starvation is hence avoided.

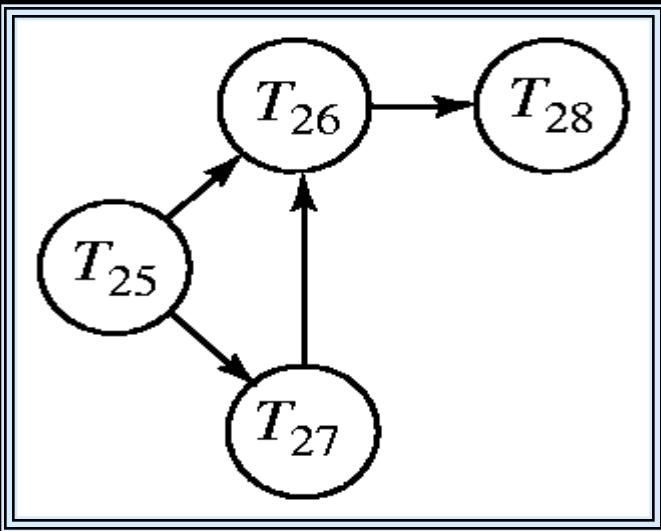
## Timeout-Based Schemes :

- a transaction waits for a lock only for a specified amount of time. After that, the wait times out and the transaction is rolled back.

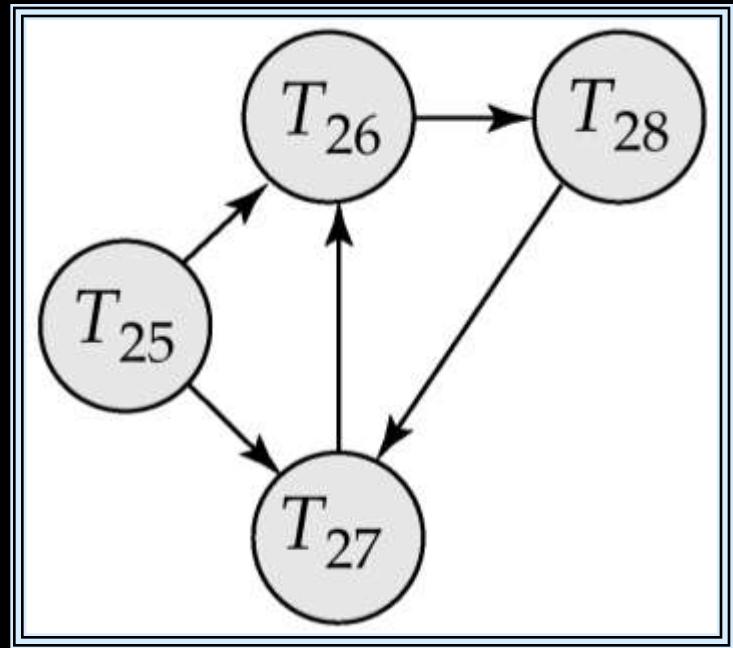
# Deadlock Detection

- Deadlocks can be described as a *wait-for graph*, which consists of a pair  $G = (V, E)$ ,
  - $V$  is a set of vertices (all the transactions in the system)
  - $E$  is a set of edges; each element is an ordered pair  $T_i \rightarrow T_j$ .
- If  $T_i \rightarrow T_j$  is in  $E$ , then there is a directed edge from  $T_i$  to  $T_j$ , implying that  $T_i$  is waiting for  $T_j$  to release a data item.

# Deadlock Detection (Cont.)



Wait-for graph without a cycle



Wait-for graph with a cycle

# Deadlock Recovery

- When deadlock is detected :
  - Some transaction will have to rolled back (made a victim) to break deadlock. Select that transaction as victim that will incur minimum cost.
  - Rollback -- determine how far to roll back transaction
    - Total rollback: Abort the transaction and then restart it.
    - More effective to roll back transaction only as far as necessary to break deadlock.
  - Starvation happens if same transaction is chosen as victim again and again.

# Insert and Delete Operations

- If two-phase locking is used :
  - A **delete** operation may be performed only if the transaction deleting the tuple has an exclusive lock on the tuple to be deleted.
  - A transaction that inserts a new tuple into the database is given an X-mode lock on the tuple

■ Insertions and deletions can lead to the phantom phenomenon.

- A transaction that scans a relation (e.g., find all accounts in Perryridge) and a transaction that inserts a tuple in the

# Insert and Delete Operations

- The transaction scanning the relation is reading information that indicates what tuples the relation contains, while a transaction inserting a tuple updates the same information.

- One solution:
  - The information should be locked.
  - Associate a data item with the relation, to represent the information about what tuples the relation contains.
  - Transactions scanning the relation acquire a shared lock in the data item,
  - Transactions inserting or deleting a tuple acquire an exclusive lock on the data item.  
(Note: locks on the data item do not conflict)

# Index Locking Protocol

- Every relation must have at least one index. Access to a relation must be made only through one of the indices on the relation.
- A transaction  $T_i$  that performs a lookup must lock all the index buckets that it accesses, in S-mode.
- A transaction  $T_i$  may not insert a tuple  $t_i$  into a relation  $r$  without updating all indices to  $r$ .
- $T_i$  must perform a lookup on every

# Weak Levels of Consistency

- Degree-two consistency: differs from two-phase locking in that S-locks may be released at any time, and locks may be acquired at any time
  - X-locks must be held till end of transaction
  - Serializability is not guaranteed, programmer must ensure that no erroneous database state will occur]
- Cursor stability:
  - For reads, each tuple is locked, read, and lock is immediately released
  - X-locks are held till end of transaction

# Weak Levels of Consistency in

- **SQL** allows non-serializable executions
  - **Serializable**: is the default
  - **Repeatable read**: allows only committed records to be read, and repeating a read should return the same value (so read locks should be retained)
    - However, the phantom phenomenon need not be prevented
      - T1 may see some records inserted by T2, but may not see others inserted by T2
  - **Read committed**: same as degree two consistency, but most systems implement it as cursor-stability
  - **Read uncommitted**: allows even

# Concurrency in Index Structures

- Indices are unlike other database items in that their only job is to help in accessing data.
- Index-structures are typically accessed very often, much more than other database items.
- Treating index-structures like other database items leads to low concurrency. Two-phase locking on an index may result in transactions executing practically one-at-a-time.
- It is acceptable to have nonserializable concurrent access to an index as long as the accuracy of the index is maintained.

# Concurrency in Index Structures

- Example of index concurrency protocol:
  - Use crabbing instead of two-phase locking on the nodes of the B<sup>+</sup>-tree, as follows. During search/insertion/deletion:
    - First lock the root node in shared mode.
    - After locking all required children of a node in shared mode, release the lock on the node.
    - During insertion/deletion, upgrade leaf node locks to exclusive mode.

END OF CHAPTER



# Partial Schedule Under Two-Phase Locking

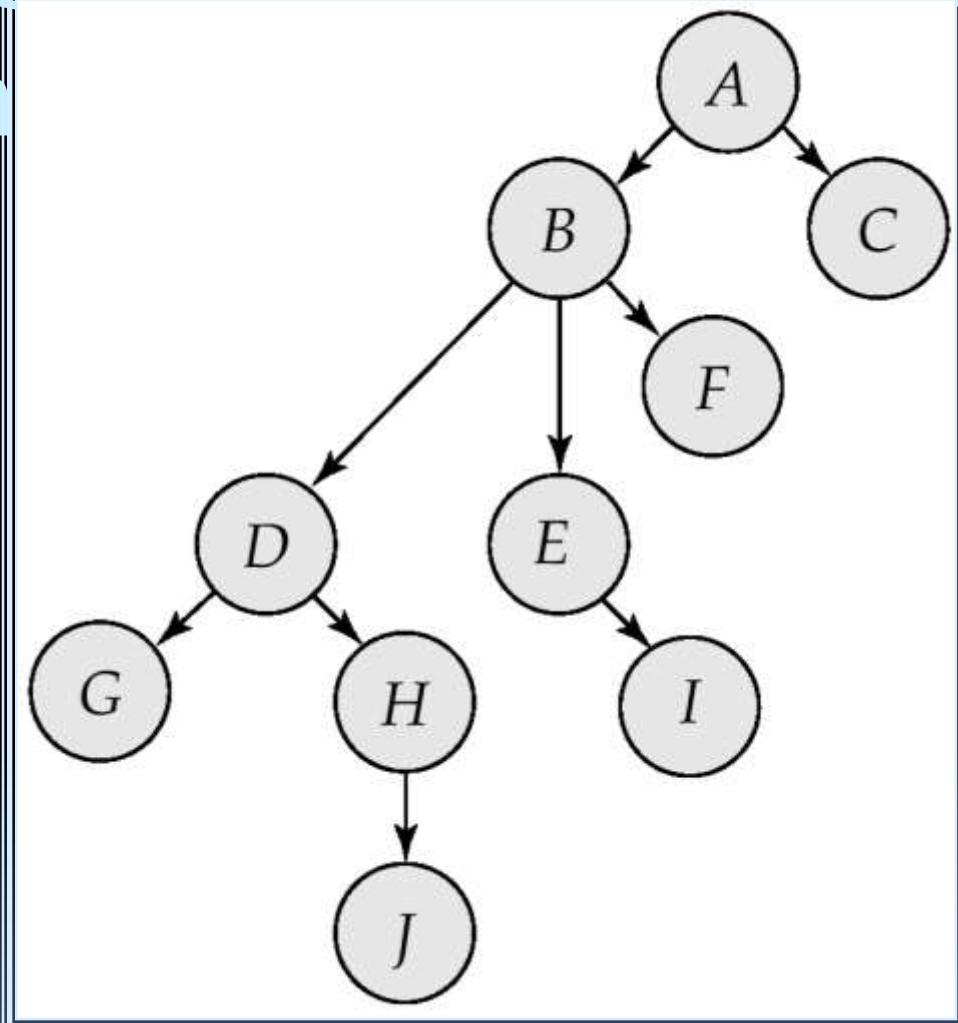
$T_5$	$T_6$	$T_7$
lock-X( $A$ )		
read( $A$ )		
lock-S( $B$ )		
read( $B$ )		
write( $A$ )		
unlock( $A$ )		
	lock-X( $A$ )	
	read( $A$ )	
	write( $A$ )	
	unlock( $A$ )	
		lock-S( $A$ )
		read( $A$ )

# Incomplete Schedule With a Lock Conversion

$T_8$	$T_9$
lock-S( $a_1$ )	
lock-S( $a_2$ )	lock-S( $a_1$ )
lock-S( $a_3$ )	lock-S( $a_2$ )
lock-S( $a_4$ )	
	unlock( $a_1$ )
	unlock( $a_2$ )
lock-S( $a_n$ )	
upgrade( $a_1$ )	

# Lock Table

# Tree-Structured Database Graph



# Serializable Schedule Under the Tree Protocol

$T_{10}$	$T_{11}$	$T_{12}$	$T_{13}$
lock-X( $B$ )			
lock-X( $E$ ) lock-X( $D$ ) unlock( $B$ ) unlock( $E$ )	lock-X( $D$ ) lock-X( $H$ ) unlock( $D$ )		
lock-X( $G$ ) unlock( $D$ )	unlock( $H$ )	lock-X( $B$ ) lock-X( $E$ )	
unlock ( $G$ )		unlock( $E$ ) unlock( $B$ )	lock-X( $D$ ) lock-X( $H$ ) unlock( $D$ ) unlock( $H$ )

# Schedule 3

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

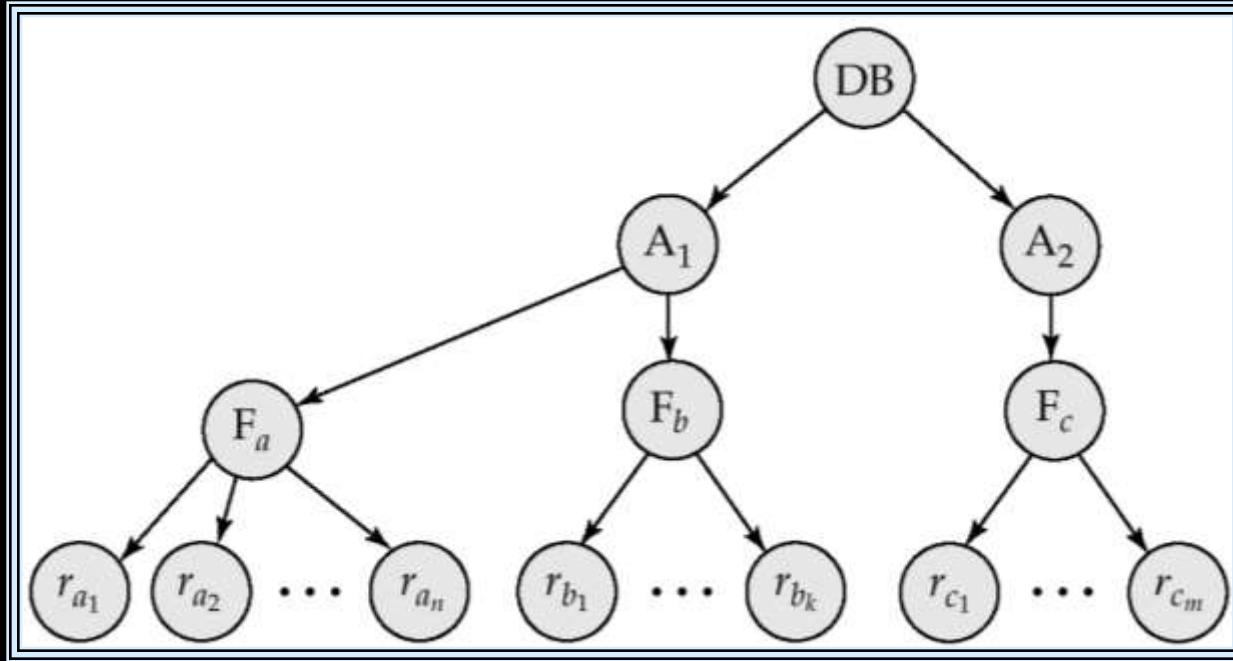
# Schedule 4

$T_{16}$	$T_{17}$
<b>read(<math>Q</math>)</b>	
<b>write(<math>Q</math>)</b>	<b>write(<math>Q</math>)</b>

# Schedule 5, A Schedule Produced by Using Validation

$T_{14}$	$T_{15}$
$\text{read}(B)$	$\text{read}(B)$ $B := B - 50$ $\text{read}(A)$ $A := A + 50$
$\text{read}(A)$ <i>&lt;validate &gt;</i> $\text{display}(A + B)$	<i>&lt;validate &gt;</i> $\text{write}(B)$ $\text{write}(A)$

# Granularity Hierarchy



# Compatibility Matrix

	IS	IX	S	SIX	X
IS	true	true	true	true	false
IX	true	true	false	false	false
S	true	false	true	false	false
SIX	true	false	false	false	false
X	false	false	false	false	false

# Wait-for Graph With No Cycle



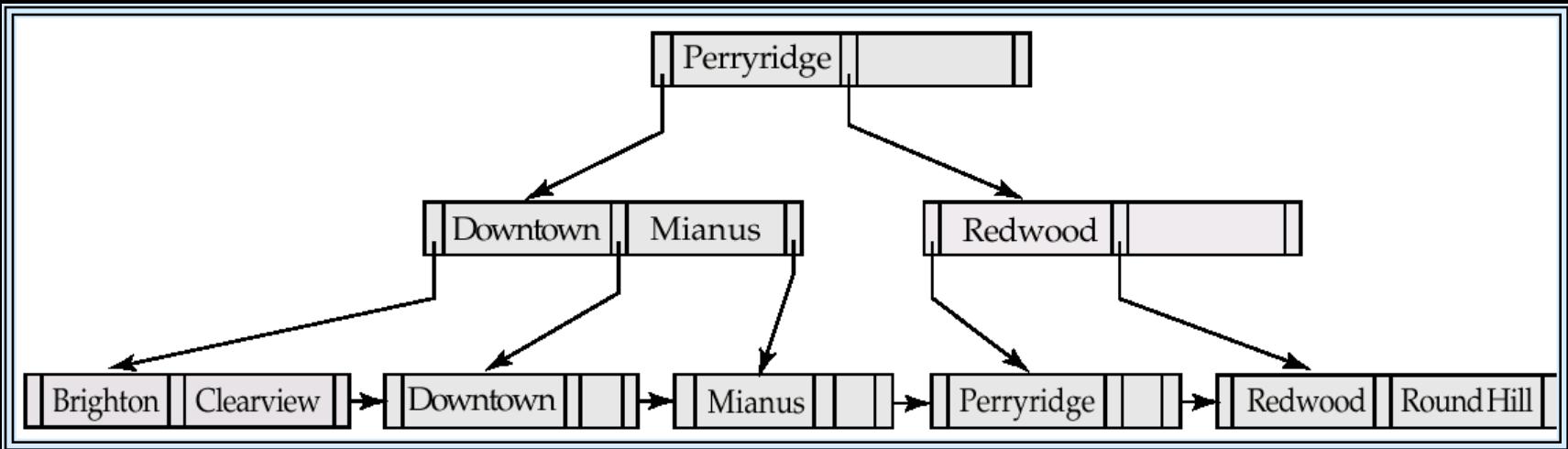
# Wait-for-graph With A Cycle



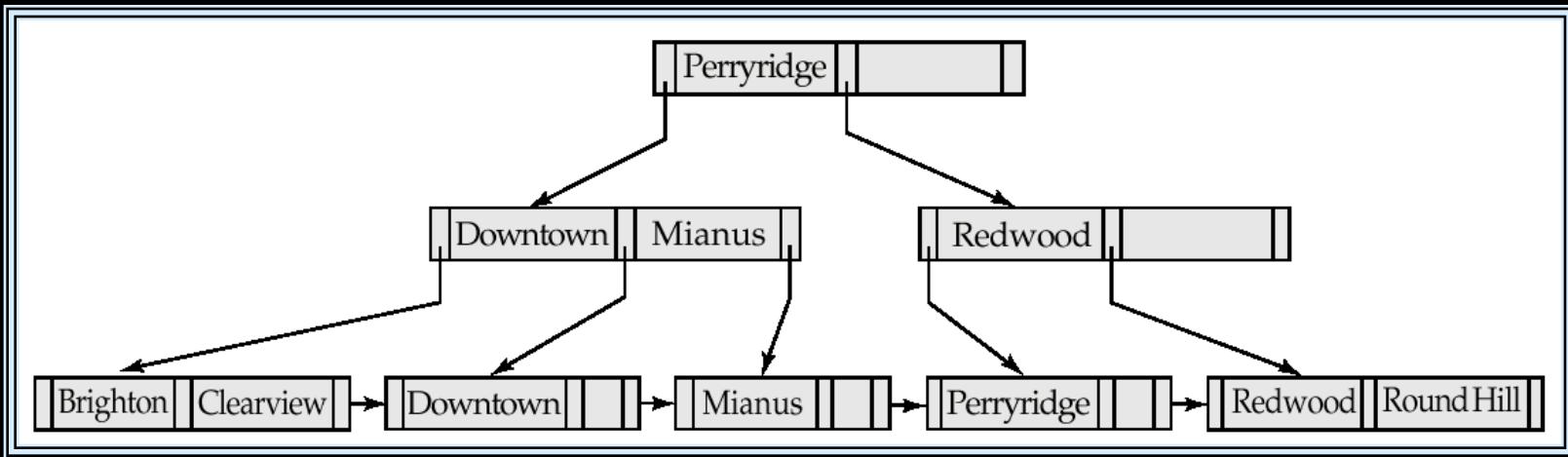
# Nonserializable Schedule with Degree-Two Consistency

$T_3$	$T_4$
lock-S( $Q$ ) read( $Q$ ) unlock( $Q$ )	lock-X( $Q$ ) read( $Q$ ) write( $Q$ ) unlock( $Q$ )
lock-S( $Q$ ) read( $Q$ ) unlock( $Q$ )	

# B<sup>+</sup>-Tree For *account* File with $n = 3$ .



# Insertion of “Clearview” Into the B<sup>+</sup>-Tree of Figure 16.21



# Lock-Compatibility Matrix

	S	X	I
S	true	false	false
X	false	false	false
I	false	false	true

# Chapter 17: Recovery System

- Failure Classification
- Storage Structure
- Recovery and Atomicity
- Log-Based Recovery
- Shadow Paging
- Recovery With Concurrent Transactions
- Buffer Management
- Failure with Loss of Nonvolatile Storage
- Advanced Recovery Techniques
- ARIES Recovery Algorithm

# Failure Classification

- Transaction failure :
  - **Logical errors:** transaction cannot complete due to some internal error condition
  - **System errors:** the database system must terminate an active transaction due to an error condition (e.g., deadlock)
- System crash: a power failure or other hardware or software failure causes the system to crash.
  - **Fail-stop assumption:** non-volatile storage contents are assumed to not be corrupted by system crash
    - Database systems have numerous integrity checks to prevent corruption of disk data

# Recovery Algorithms

- Recovery algorithms are techniques to ensure database consistency and transaction atomicity and durability despite failures
  - Focus of this chapter
- Recovery algorithms have two parts
  1. Actions taken during normal transaction processing to ensure enough information exists to recover from failures
  2. Actions taken after a failure to recover the database contents to a state that ensures

# Storage Structure

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

# Stable-Storage Implementation

- Maintain multiple copies of each block on separate disks
  - copies can be at remote sites to protect against disasters such as fire or flooding.
- Failure during data transfer can still result in inconsistent copies:  
Block transfer can result in
  - Successful completion
  - Partial failure: destination block has incorrect information
  - Total failure: destination block was never updated
- Protecting storage media from failure during data transfer (one solution):
  - Execute output operation as follows (assuming two copies of each block):
    1. Write the information onto the first physical

# Stable-Storage Implementation (Cont.)

- Protecting storage media from failure during data transfer (cont.):
- Copies of a block may differ due to failure during output operation. To recover from failure:
  1. First find inconsistent blocks:
    1. *Expensive solution:* Compare the two copies of every disk block.
    2. *Better solution:*
      - Record in-progress disk writes on non-volatile storage (Non-volatile RAM or special area of disk).
      - Use this information during recovery to find blocks that may be inconsistent, and only

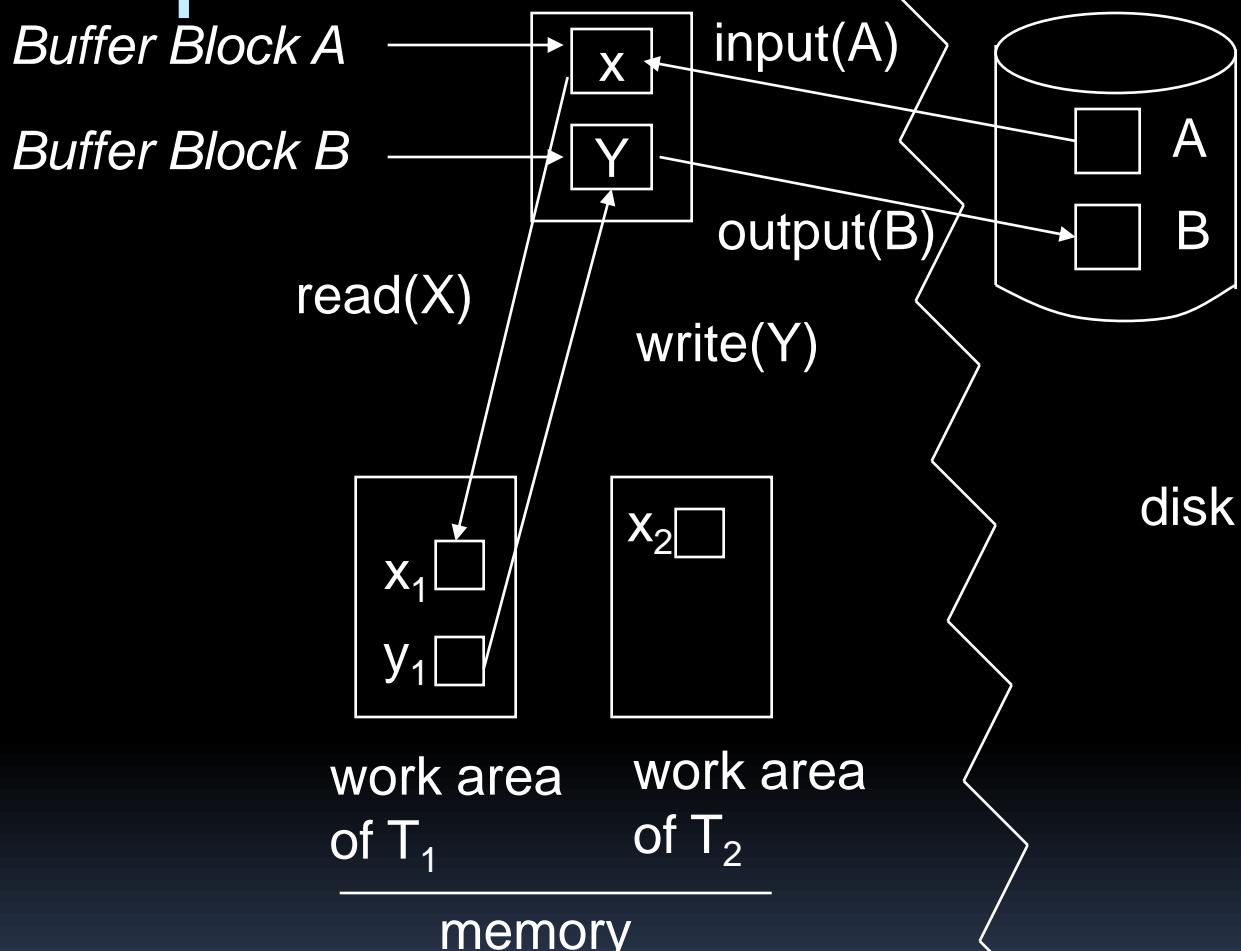
# Data Access

- Physical blocks are those blocks residing on the disk.
- Buffer blocks are the blocks residing temporarily in main memory.
- Block movements between disk and main memory are initiated through the following two operations:
  - **input( $B$ )** transfers the physical block  $B$  to main memory.
  - **output( $B$ )** transfers the buffer block  $B$  to the disk, and replaces the appropriate physical block there.
- Each transaction  $T_i$  has its private work-area in which local copies of all

# Data Access (Cont.)

- Transaction transfers data items between system buffer blocks and its private work-area using the following operations :
  - **read( $X$ )** assigns the value of data item  $X$  to the local variable  $x_i$ .
  - **write( $X$ )** assigns the value of local variable  $x_i$  to data item  $\{X\}$  in the buffer block.
  - both these commands may necessitate the issue of an **input( $B_x$ )** instruction before the assignment, if the block  $B_x$  in which  $X$  resides is not already in memory.
- Transactions
  - Perform **read( $X$ )** while accessing  $X$  for the first time.

# Example of Data Access



# Recovery and Atomicity

- Modifying the database without ensuring that the transaction will commit may leave the database in an inconsistent state.
- Consider transaction  $T_i$  that transfers \$50 from account  $A$  to account  $B$ ; goal is either to perform all database modifications made by  $T_i$  or none at all.
- Several output operations may be required for  $T_i$  (to output  $A$  and  $B$ ). A failure may occur after one of these

# Recovery and Atomicity (Cont.)

- To ensure atomicity despite failures, we first output information describing the modifications to stable storage without modifying the database itself.
- We study two approaches:
  - **log-based recovery**, and
  - **shadow-paging**
- We assume (initially) that transactions run serially, that is, one after the other.

# Log-Based Recovery

- A log is kept on stable storage.

- The log is a sequence of **log records**, and maintains a record of update activities on the database.

- When transaction  $T_j$  starts, it registers itself by writing a  $\langle T_j \text{ start} \rangle$  log record

- Before  $T_j$  executes  $\text{write}(X)$ , a log record  $\langle T_j, X, V_1, V_2 \rangle$  is written, where  $V_1$  is the value of  $X$  before the write, and  $V_2$  is the value to be written to  $X$ .

- Log record notes that  $T_j$  has performed a write on data item  $X_j$ .  $X_j$  had value  $V_1$  before the write, and will have value  $V_2$  after the write.

- When  $T_j$  finishes its last statement, the log record  $\langle T_j \text{ commit} \rangle$  is written.

- We assume for now that log records are written directly to stable storage (that is, they are not buffered).

# Deferred Database Modification

- The deferred database modification scheme records all modifications to the log, but defers all the writes to after partial commit.
- Assume that transactions execute serially
- Transaction starts by writing  $\langle T_i, start \rangle$  record to log.
- A  $\text{write}(X)$  operation results in a log record  $\langle T_i, X, V \rangle$  being written, where  $V$  is the new value for  $X$ 
  - Note: old value is not needed for this

# Deferred Database Modification

- During recovery after a crash, a transaction needs to be redone if and only if both  $\langle T_i, \text{start} \rangle$  and  $\langle T_i, \text{commit} \rangle$  are there in the log.
- Redoing a transaction  $T_i$  (redo  $T_i$ ) sets the value of all data items updated by the transaction to the new values.
- Crashes can occur while
  - the transaction is executing the original updates, or
  - while recovery action is being taken
- example transactions  $T_0$  and  $T_1$  ( $T_0$  executes before  $T_1$ ):

# Deferred Database Modification

Below we show the log as it appears at three times.

$\langle T_0 \text{ start} \rangle$	$\langle T_0 \text{ start} \rangle$	$\langle T_0 \text{ start} \rangle$
$\langle T_0, A, 950 \rangle$	$\langle T_0, A, 950 \rangle$	$\langle T_0, A, 950 \rangle$
$\langle T_0, B, 2050 \rangle$	$\langle T_0, B, 2050 \rangle$	$\langle T_0, B, 2050 \rangle$
	$\langle T_0 \text{ commit} \rangle$	$\langle T_0 \text{ commit} \rangle$
	$\langle T_1 \text{ start} \rangle$	$\langle T_1 \text{ start} \rangle$
	$\langle T_1, C, 600 \rangle$	$\langle T_1, C, 600 \rangle$
		$\langle T_1 \text{ commit} \rangle$

(a)

(b)

(c)

If log on stable storage at time of crash  
is as in case:

# Immediate Database

- The **modification** scheme allows database updates of an uncommitted transaction to be made as the writes are issued
  - since undoing may be needed, update logs must have both old value and new value
- Update log record must be written *before* database item is written
  - We assume that the log record is output directly to stable storage
  - Can be extended to postpone log record output, so long as prior to execution of an **output( $B$ )** operation for a data block  $B$ , all

# Immediate Database Modification Example

log

$T_0$  start>

$T_0, A, 1000, 950>$

, B, 2000, 2050

Write

Output

$$A = 950$$

$$B = 2050$$

$T_0$  commit>

$T_1$  start>

$T_1, C, 700, 600>$

$$C = 600$$

$$B_B, B_C$$

$$B_A$$

Note:  $B_X$  denotes block containing  $X$ .

# Immediate Database

Recovery procedure has two operations instead of one:

- **undo( $T_i$ )** restores the value of all data items updated by  $T_i$  to their old values, going backwards from the last log record for  $T_i$ ,
- **redo( $T_i$ )** sets the value of all data items updated by  $T_i$  to the new values, going forward from the first log record for  $T_i$ ,

Both operations must be idempotent

- That is, even if the operation is executed multiple times the effect is the same as if it is executed once
  - Needed since operations may get re-executed during recovery

# Immediate DB Modification Recovery Example

Below we show the log as it appears at three instances of time.

< $T_0$  start>

< $T_0$ , A, 1000, 950>

< $T_0$ , B, 2000, 2050>

< $T_0$  start>

< $T_0$ , A, 1000, 950>

< $T_0$ , B, 2000, 2050>

< $T_0$  commit>

< $T_1$  start>

< $T_1$ , C, 700, 600>

< $T_0$  start>

< $T_0$ , A, 1000, 950>

< $T_0$ , B, 2000, 2050>

< $T_0$  commit>

< $T_1$  start>

< $T_1$ , C, 700, 600>

< $T_1$  commit>

(a)

(b)

(c)

- (a) undo ( $T_0$ ): B is restored to 2000 and A to 1000.
- (b) undo ( $T_1$ ) and redo ( $T_0$ ): C is restored to 700, and then A and B are set to 950 and 2050 respectively.
- (c) redo ( $T_0$ ) and redo ( $T_1$ ): A and B are set to 950 and 2050 respectively. Then C is set to 600

# Checkpoints

- Problems in recovery procedure as discussed earlier :
  1. searching the entire log is time-consuming
  2. we might unnecessarily redo transactions which have already
  3. output their updates to the database.

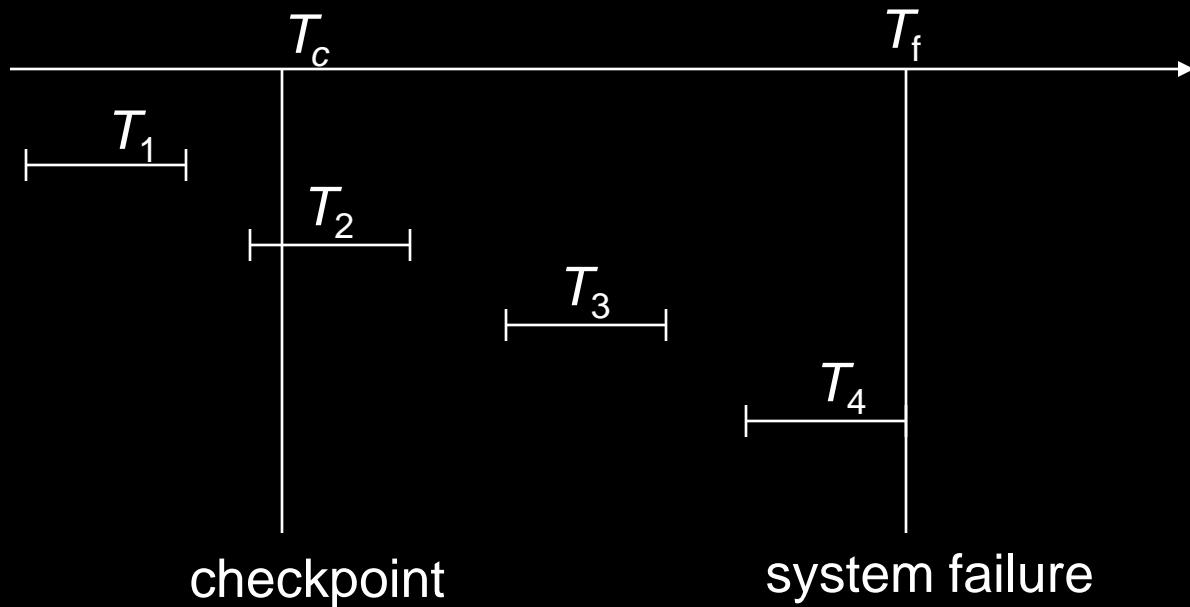
- Streamline recovery procedure by periodically performing checkpointing

1. Output all log records currently residing in main memory onto stable storage.
2. Output all modified buffer blocks to the disk

# Checkpoints (Cont.)

- During recovery we need to consider only the most recent transaction  $T_i$  that started before the checkpoint, and transactions that started after  $T_i$ .
  - Scan backwards from end of log to find the most recent <checkpoint> record
  - Continue scanning backwards till a record  $\langle T_i \text{ start} \rangle$  is found.
  - Need only consider the part of log following above start record. Earlier part of log can be ignored during recovery, and can be erased whenever desired.
  - For all transactions (starting from  $T_i$  or

# Example of Checkpoints

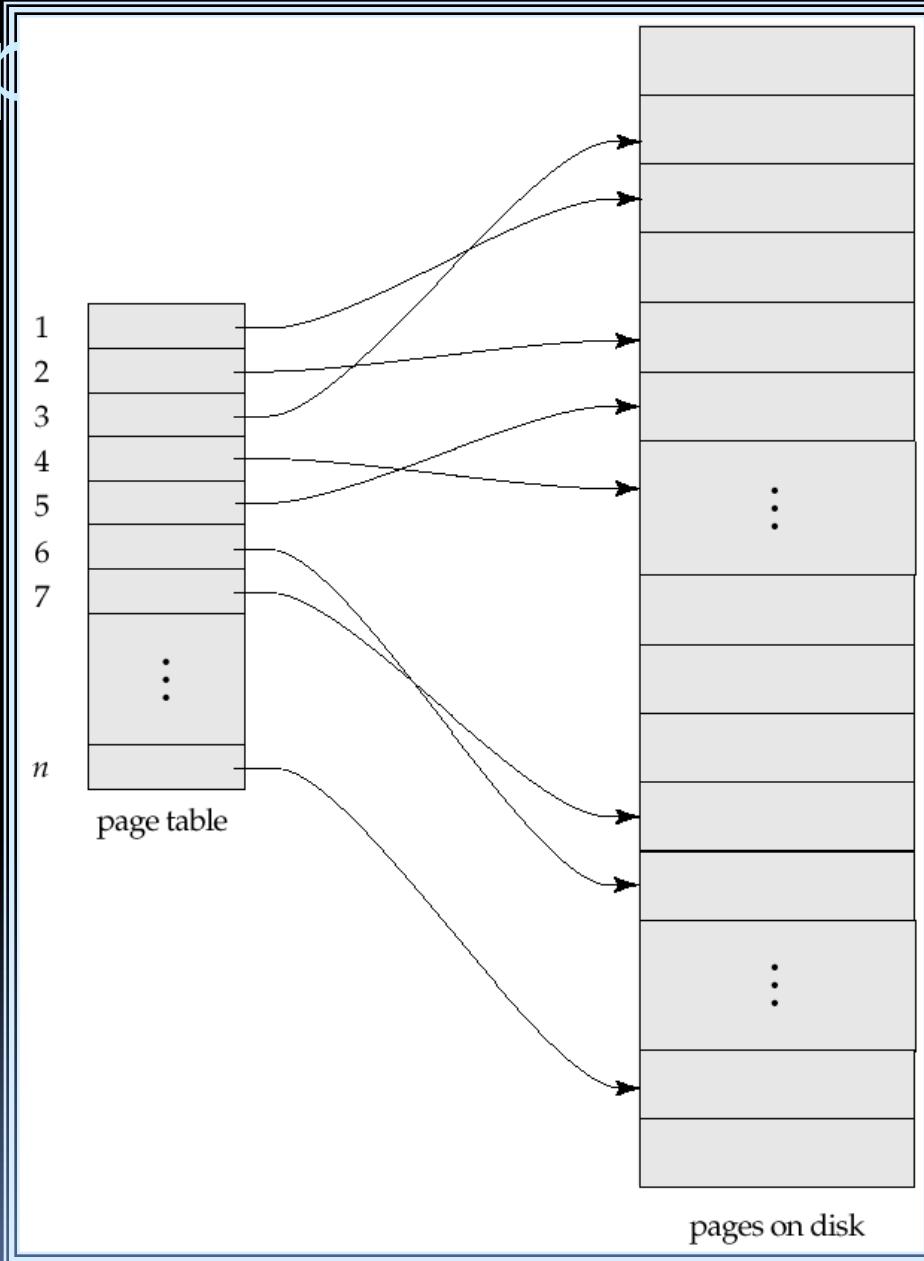


- $T_1$  can be ignored (updates already output to disk due to checkpoint)
- $T_2$  and  $T_3$  redo

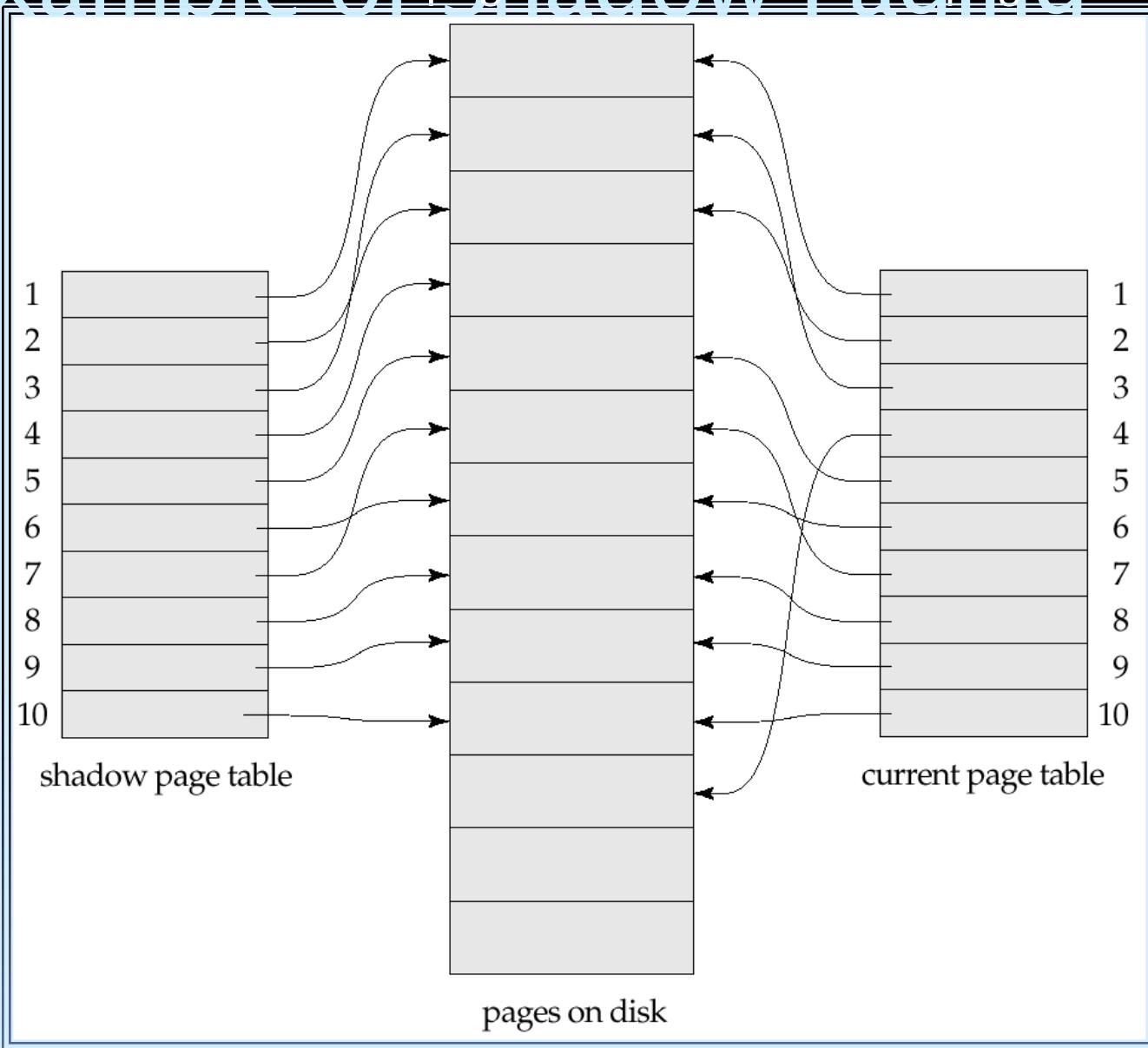
# Shadow Paging

- Shadow paging is an alternative to log-based recovery; this scheme is useful if transactions execute serially
- Idea: maintain *two* page tables during the lifetime of a transaction -the current page table, and the shadow page table
- Store the shadow page table in nonvolatile storage, such that state of the database prior to transaction execution may be recovered.
  - Shadow page table is never modified during execution
- To start with, both the page tables are

# Sampling



# Example of Shadow Paging



# Shadow Paging (Cont.)

To commit a transaction :

1. Flush all modified pages in main memory to disk
2. Output current page table to disk
3. Make the current page table the new shadow page table, as follows:
  - keep a pointer to the shadow page table at a fixed (known) location on disk.
  - to make the current page table the new shadow page table, simply update the pointer to point to current page table on disk
- Once pointer to shadow page table has been written, transaction is committed.
- No recovery is needed after a crash — new transactions can start right away, using the shadow page table.
- Pages not pointed to from current/shadow page table should be freed (garbage collected)

# Shadow Paging (Cont.)

## Advantages of shadow-paging over log-based schemes

- no overhead of writing log records
- recovery is trivial

## Disadvantages :

- Copying the entire page table is very expensive
  - Can be reduced by using a page table structured like a B<sup>+</sup>-tree
    - No need to copy entire tree, only need to copy paths in the tree that lead to updated leaf nodes
- Commit overhead is high even with above extension
  - Need to flush every updated page, and page table

# Recovery With Concurrent Transactions

- We modify the log-based recovery schemes to allow multiple transactions to execute concurrently.
  - All transactions share a single disk buffer and a single log
  - A buffer block can have data items updated by one or more transactions
- We assume concurrency control using strict two-phase locking;
  - i.e. the updates of uncommitted transactions should not be visible to other transactions
    - Otherwise how to perform undo if T1 updates A, then T2 updates A and commits, and finally T1 has to abort?
- Logging is done as described earlier.

# Recovery With Concurrent Transactions (Cont.)

Checkpoints are performed as before, except that the checkpoint log record is now of the form

<checkpoint  $L$ >

where  $L$  is the list of transactions active at the time of the checkpoint

- We assume no updates are in progress while the checkpoint is carried out (will relax this later)
- When the system recovers from a crash, it first does the following:
  1. Initialize *undo-list* and *redo-list* to empty
  2. Scan the log backwards from the end, stopping when the first <checkpoint  $L$ >

# Recovery With Concurrent Transactions (Cont.)

- At this point *undo-list* consists of incomplete transactions which must be undone, and *redo-list* consists of finished transactions that must be redone.
- Recovery now continues as follows:
  1. Scan log backwards from most recent record, stopping when  $\langle T_i, \text{start} \rangle$  records have been encountered for every  $T_i$  in *undo-list*.
    - During the scan, perform **undo** for each log record that belongs to a transaction in *undo-list*.

# Example of Recovery

- Go over the steps of the recovery algorithm on the following log:

```
< T_0 start>
< T_0 , A, 0, 10>
< T_0 commit>
< T_1 start>
< T_1 , B, 0, 10>
< T_2 start> /* Scan in Step 4 stops here
*//
< T_2 , C, 0, 10>
< T_2 , C, 10, 20>
<checkpoint { T_1 , T_2 }>
< T_3 start>
< T_3 , A, 10, 20>
< T_3 , D, 0, 10>
< T_3 commit>
```

# Log Record Buffering

- Log record buffering: log records are buffered in main memory, instead of being output directly to stable storage.
  - Log records are output to stable storage when a block of log records in the buffer is full, or a **log force** operation is executed.
- Log force is performed to commit a transaction by forcing all its log records (including the commit record) to stable storage.
- Several log records can thus be output using a single output operation, reducing the I/O cost

# Log Record Buffering (Cont.)

- The rules below must be followed if log records are buffered:
  - Log records are output to stable storage in the order in which they are created.
  - Transaction  $T_i$  enters the commit state only when the log record  $\langle T_i \text{ commit} \rangle$  has been output to stable storage.
  - Before a block of data in main memory is output to the database, all log records pertaining to data in that block must have been output to stable storage.

# Database Buffering

- Database maintains an in-memory buffer of data blocks
  - When a new block is needed, if buffer is full an existing block needs to be removed from buffer
  - If the block chosen for removal has been updated, it must be output to disk
- As a result of the write-ahead logging rule, if a block with uncommitted updates is output to disk, log records with undo information for the updates are output to the log on stable storage first.
- No updates should be in progress on a block when it is output to disk. Can be ensured as follows.
  - Before writing a data item, transaction acquires exclusive lock on block containing the data item
  - Lock can be released once the write is

# Buffer Management (Cont.)

- Database buffer can be implemented either
  - in an area of real main-memory reserved for the database, or
  - in virtual memory
- Implementing buffer in reserved main-memory has drawbacks:
  - Memory is partitioned before-hand between database buffer and applications, limiting flexibility.
  - Needs may change, and although operating system knows best how memory should be divided up at any time, it cannot

# Buffer Management (Cont.)

- Database buffers are generally implemented in virtual memory in spite of some drawbacks:
  - When operating system needs to evict a page that has been modified, to make space for another page, the page is written to swap space on disk.
  - When database decides to write buffer page to disk, buffer page may be in swap space, and may have to be read from swap space on disk and output to the database on disk, resulting in extra I/O!
    - Known as dual paging problem.

# Failure with Loss of Nonvolatile Storage

- So far we assumed no loss of non-volatile storage
- Technique similar to checkpointing used to deal with loss of non-volatile storage
  - Periodically **dump** the entire content of the database to stable storage
  - No transaction may be active during the dump procedure; a procedure similar to checkpointing must take place
    - Output all log records currently residing in main memory onto stable storage.
    - Output all buffer blocks onto the disk.
    - Copy the contents of the database to stable storage.
    - Output a record <dump> to log on stable storage.
  - To recover from disk failure

# ADVANCED RECOVERY ALGORITHM



# Advanced Recovery Techniques

- Support high-concurrency locking techniques, such as those used for B<sup>+</sup>-tree concurrency control
- Operations like B<sup>+</sup>-tree insertions and deletions release locks early.
  - They cannot be undone by restoring old values (**physical undo**), since once a lock is released, other transactions may have updated the B<sup>+</sup>-tree.
  - Instead, insertions (resp. deletions) are undone by executing a deletion (resp. insertion) operation (known as **logical undo**).

For such operations, undo log records should contain the undo operation to

# Advanced Recovery Techniques

Operation Logging is done as follows:

(Cont.)

1. When operation starts, log  $\langle T_i, O_j, \text{operation-begin} \rangle$ . Here  $O_j$  is a unique identifier of the operation instance.
  2. While operation is executing, normal log records with physical redo and physical undo information are logged.
  3. When operation completes,  $\langle T_i, O_j, \text{operation-end}, U \rangle$  is logged, where  $U$  contains information needed to perform a logical undo operation.
- If crash/rollback occurs before operation completes:
    - the operation-end log record is not found, and

# Advanced Recovery Techniques

Rollback(~~Commit~~) transaction  $T_i$  is done as follows:

- Scan the log backwards
  1. If a log record  $\langle T_i, X, V_1, V_2 \rangle$  is found, perform the undo and log a special **redo-only log record**  $\langle T_i, X, V_i \rangle$ .
  2. If a  $\langle T_i, O_j, \text{operation-end}, U \rangle$  record is found
    - Rollback the operation logically using the undo information  $U$ .
      - Updates performed during roll back are logged just like during normal operation execution.

# Advanced Recovery Techniques

## Scan the log backwards (cont.): *(Cont.)*

3. If a redo-only record is found ignore it
4. If a  $\langle T_i, O_j, \text{operation-abort} \rangle$  record is found:
  - ☛ skip all preceding log records for  $T_i$ , until the record  $\langle T_i, O_j, \text{operation-begin} \rangle$  is found.
5. Stop the scan when the record  $\langle T_i, \text{start} \rangle$  is found
6. Add a  $\langle T_i, \text{abort} \rangle$  record to the log

Some points to note:

- Cases 3 and 4 above can occur only if the database crashes while a

# Advanced Recovery

The following actions are taken when recovering from system crash  
**Techniques(Cont.)**

1. Scan log forward from last  $<\text{checkpoint } L>$  record
  1. Repeat history by physically redoing all updates of all transactions,
  2. Create an undo-list during the scan as follows
    - *undo-list* is set to  $L$  initially
    - Whenever  $<T_i \text{ start}>$  is found  $T_i$  is added to *undo-list*
    - Whenever  $<T_i \text{ commit}>$  or  $<T_i \text{ abort}>$  is found,  $T_i$  is deleted from *undo-list*

# Advanced Recovery Techniques

## Recovery from system crash (cont.)

2. Scan log backwards, performing undo on log records of transactions found in *undo-list*.

- Transactions are rolled back as described earlier.
- When  $\langle T_i, \text{start} \rangle$  is found for a transaction  $T_i$  in *undo-list*, write a  $\langle T_i, \text{abort} \rangle$  log record.
- Stop scan when  $\langle T_i, \text{start} \rangle$  records have been found for all  $T_i$  in *undo-list*

▪ This undoes the effects of incomplete transactions (those with neither

# Advanced Recovery Techniques

- Checkpointing is done as follows:  
**(Cont.)**

1. Output all log records in memory to stable storage
2. Output to disk all modified buffer blocks
3. Output to log on stable storage a < checkpoint > record.

Transactions are not allowed to perform any actions while checkpointing is in progress.

- Fuzzy checkpointing allows transactions to progress while the most time consuming parts of the checkpointing process are in progress.

# Advanced Recovery Techniques

Fuzzy checkpointing is done as follows:

1. Temporarily stop all updates by transactions
2. Write a **<checkpoint L>** log record and force log to stable storage
3. Note list  $M$  of modified buffer blocks
4. Now permit transactions to proceed with their actions
5. Output to disk all modified buffer blocks in list  $M$ 
  - ❖ blocks should not be updated while being output
  - ❖ Follow WAL: all log records pertaining to a block must be output before the block is output

ARIES RECOVERY  
ALGORITHM



# ARIES

- ARIES is a state of the art recovery method
  - Incorporates numerous optimizations to reduce overheads during normal processing and to speed up recovery
  - The “advanced recovery algorithm” we studied earlier is modeled after ARIES, but greatly simplified by removing optimizations
- Unlike the advanced recovery algorithm, ARIES
  1. Uses log sequence number (LSN) to identify log records

# ARIES Optimizations

- Physiological redo
  - Affected page is physically identified, action within page can be logical
    - Used to reduce logging overheads
      - e.g. when a record is deleted and all other records have to be moved to fill hole
        - Physiological redo can log just the record deletion
        - Physical redo would require logging of old and new values for much of the page
    - Requires page to be output to disk atomically
      - Easy to achieve with hardware RAID, also supported by some disk systems
      - Incomplete page output can be detected by checksum techniques,
        - But extra actions are required for recovery

# ARIES Data Structures

- Log sequence number (LSN) identifies each log record
  - Must be sequentially increasing
  - Typically an offset from beginning of log file to allow fast access
    - Easily extended to handle multiple log files
- Each page contains a PageLSN which is the LSN of the last log record whose effects are reflected on the page
  - To update a page:
    - X-latch the pag, and write the log record
    - Update the page
    - Record the LSN of the log record in PageLSN
    - Unlock page

# ARIES Data Structures (Cont.)

- Each log record contains LSN of previous log record of the same transaction

LSN	TransID	PrevLSN	RedoInfo	UndoInfo
-----	---------	---------	----------	----------

- LSN in log record may be implicit
- Special redo-only log record called compensation log record (CLR) used to log actions taken during recovery that never need to be undone

LSN	TransID	UndoNextLSN	RedoInfo
-----	---------	-------------	----------

- Also serve the role of operation-abort log records used in advanced recovery

# ARIES Data Structures (Cont.)

- **DirtyPageTable**
  - List of pages in the buffer that have been updated
  - Contains, for each such page
    - **PageLSN** of the page
    - **RecLSN** is an LSN such that log records before this LSN have already been applied to the page version on disk
      - Set to current end of log when a page is inserted into dirty page table (just before being updated)
      - Recorded in checkpoints, helps to minimize redo work
- **Checkpoint log record**
  - Contains:

# ARIES Recovery Algorithm

ARIES recovery involves three passes

- Analysis pass: Determines
  - Which transactions to undo
  - Which pages were dirty (disk version not up to date) at time of crash
  - RedoLSN: LSN from which redo should start
- Redo pass:
  - Repeats history, redoing all actions from RedoLSN
    - RecLSN and PageLSNs are used to avoid redoing actions already reflected on page
- Undo pass:
  - Rolls back all incomplete transactions

# ARIES Recovery: Analysis

## Analysis pass

- Starts from last complete checkpoint log record
  - Reads in DirtyPageTable from log record
  - Sets RedoLSN = min of RecLSNs of all pages in DirtyPageTable
    - In case no pages are dirty, RedoLSN = checkpoint record's LSN
  - Sets undo-list = list of transactions in checkpoint log record
  - Reads LSN of last log record for each transaction in the undo-list

# ARIES Recovery: Analysis (Cont.)

## Analysis pass (cont.)

- Scans forward from checkpoint
  - If any log record found for transaction not in undo-list, adds transaction to undo-list
  - Whenever an update log record is found
    - If page is not in DirtyPageTable, it is added with RecLSN set to LSN of the update log record
  - If transaction end log record found, delete transaction from undo-list
  - Keeps track of last log record for each transaction in undo-list
    - May be needed for later undo

# ARIES Redo Pass

Redo Pass: Repeats history by replaying every action not already reflected in the page on disk, as follows:

- Scans forward from RedoLSN. Whenever an update log record is found:
  1. If the page is not in DirtyPageTable or the LSN of the log record is less than the RecLSN of the page in DirtyPageTable, then skip the log record
  2. Otherwise fetch the page from disk. If the Page is dirty, then update it.

# ARIES Undo Actions

- When an undo is performed for an update log record
    - Generate a CLR containing the undo action performed (actions performed during undo are logged physically or physiologically).
      - CLR for record  $n$  noted as  $n'$  in figure below
    - Set UndoNextLSN of the CLR to the PrevLSN value of the update log record
      - Arrows indicate UndoNextLSN value
  - ARIES supports partial rollback

- Used e.g. to handle deadlocks<sup>6</sup> by rolling back just enough to release req'd. locks

- Figure indicates forward actions after partial rollbacks
    - records 3 and 4 initially, later 5 and 6, then full rollback

# ARIES: Undo Pass

## Undo pass

- Performs backward scan on log undoing all transaction in undo-list
  - Backward scan optimized by skipping unneeded log records as follows:
    - Next LSN to be undone for each transaction set to LSN of last log record for transaction found by analysis pass.
    - At each step pick largest of these LSNs to undo, skip back to it and undo it
    - After undoing a log record
      - For ordinary log records, set next LSN to be undone for transaction to PrevLSN noted in the log record

# Other ARIES Features

- Recovery Independence
  - Pages can be recovered independently of others
    - E.g. if some disk pages fail they can be recovered from a backup while other pages are being used
- Savepoints:
  - Transactions can record savepoints and roll back to a savepoint
    - Useful for complex transactions
    - Also used to rollback just enough to release locks on deadlock

# Other ARIES Features (Cont.)

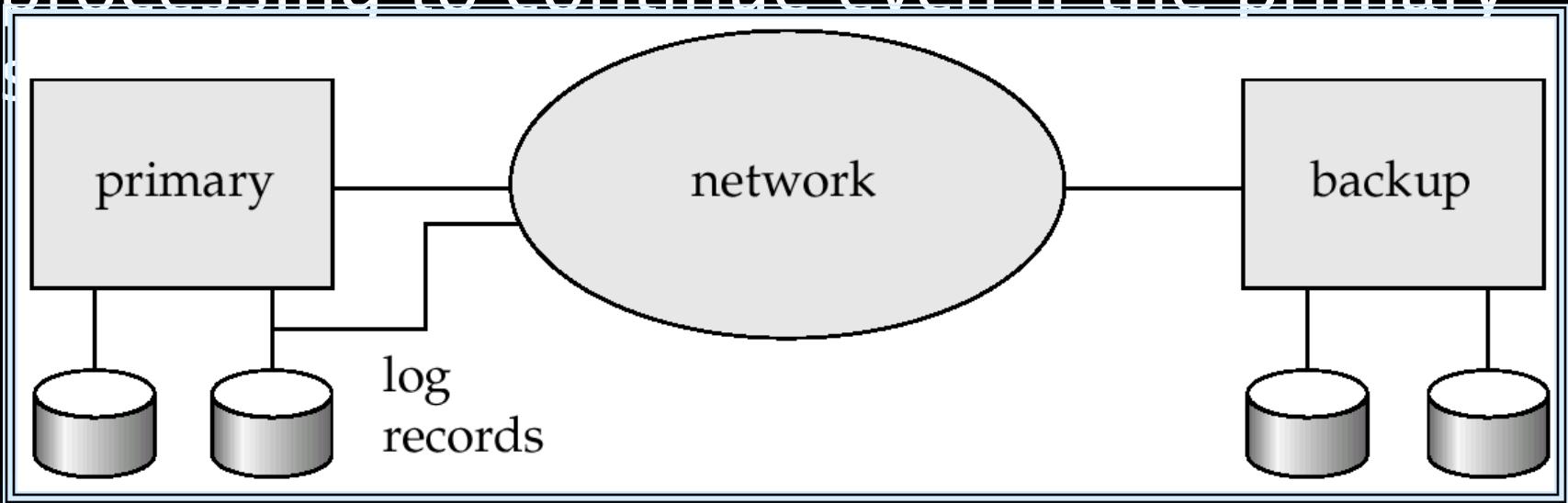
- Fine-grained locking:
  - Index concurrency algorithms that permit tuple level locking on indices can be used
    - These require logical undo, rather than physical undo, as in advanced recovery algorithm
- Recovery optimizations: For example:
  - Dirty page table can be used to prefetch pages during redo
  - Out of order redo is possible:
    - redo can be postponed on a page being fetched from disk and

# REMOTE BACKUP SYSTEMS



# Remote Backup Systems

- Remote backup systems provide high availability by allowing transaction processing to continue even if the primary



# Remote Backup Systems (Cont.)

- Detection of failure: Backup site must detect when primary site has failed
  - to distinguish primary site failure from link failure maintain several communication links between the primary and the remote backup.
- Transfer of control:
  - To take over control backup site first perform recovery using its copy of the database and all the long records it has received from the primary.
    - Thus, completed transactions are redone and incomplete transactions are rolled back.
    - When the backup site takes over processing it becomes the new primary.

# Remote Backup Systems (Cont.)

- Time to recover: To reduce delay in takeover, backup site periodically processes the redo log records (in effect, performing recovery from previous database state), performs a checkpoint, and can then delete earlier parts of the log.
- Hot-Spare configuration permits very fast takeover:
  - Backup continually processes redo log record as they arrive, applying the updates locally.
  - When failure of the primary is detected the

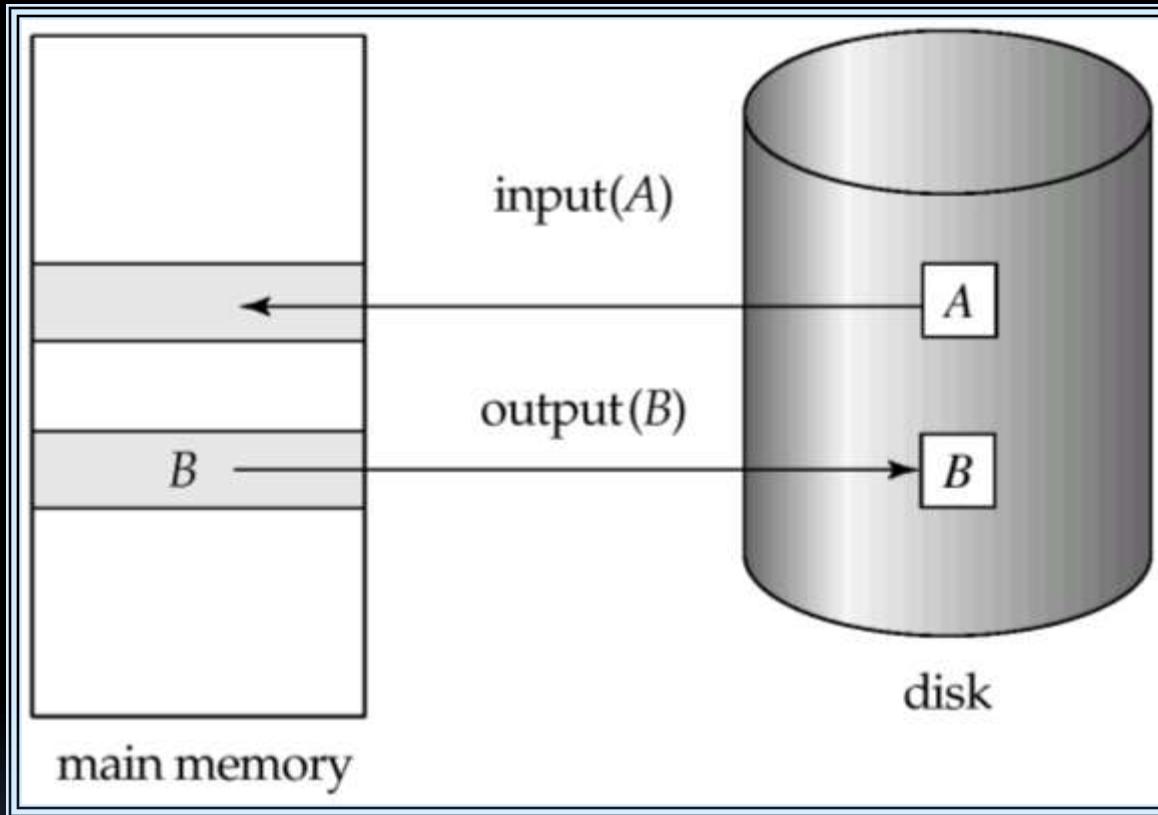
# Remote Backup Systems (Cont.)

- Ensure durability of updates by delaying transaction commit until update is logged at backup; avoid this delay by permitting lower degrees of durability.
- One-safe: commit as soon as transaction's commit log record is written at primary
  - Problem: updates may not arrive at backup before it takes over.
- Two-very-safe: commit when transaction's commit log record is written at primary and backup
  - Reduces availability since transactions cannot commit if either site fails.

END OF CHAPTER



# Block Storage Operations



# Portion of the Database Log Corresponding to $T_0$ and $T_1$

```
< T_0 start>
< T_0 , A, 950>
< T_0 , B, 2050>
< T_0 commit>
< T_1 start>
< T_1 , C, 600>
< T_1 commit>
```

# State of the Log and Database Corresponding to $T_0$ and $T_1$

Log	Database
$\langle T_0 \text{ start} \rangle$	
$\langle T_0, A, 950 \rangle$	
$\langle T_0, B, 2050 \rangle$	
$\langle T_0 \text{ commit} \rangle$	$A = 950$ $B = 2050$
$\langle T_1 \text{ start} \rangle$	
$\langle T_1, C, 600 \rangle$	
$\langle T_1 \text{ commit} \rangle$	$C = 600$

# Portion of the System Log Corresponding to $T_0$ and $T_1$

```
< T_0 start>
< T_0 , A, 1000, 950>
< T_0 , B, 2000, 2050>
< T_0 commit>
< T_1 start>
< T_1 , C, 700, 600>
< T_1 commit>
```

# State of System Log and Database Corresponding to $T_0$ and $T_1$

Log	Database
$\langle T_0 \text{ start} \rangle$	
$\langle T_0, A, 1000, 950 \rangle$	
$\langle T_0, B, 2000, 2050 \rangle$	
	$A = 950$
	$B = 2050$
$\langle T_0 \text{ commit} \rangle$	
$\langle T_1 \text{ start} \rangle$	
$\langle T_1, C, 700, 600 \rangle$	
	$C = 600$
$\langle T_1 \text{ commit} \rangle$	

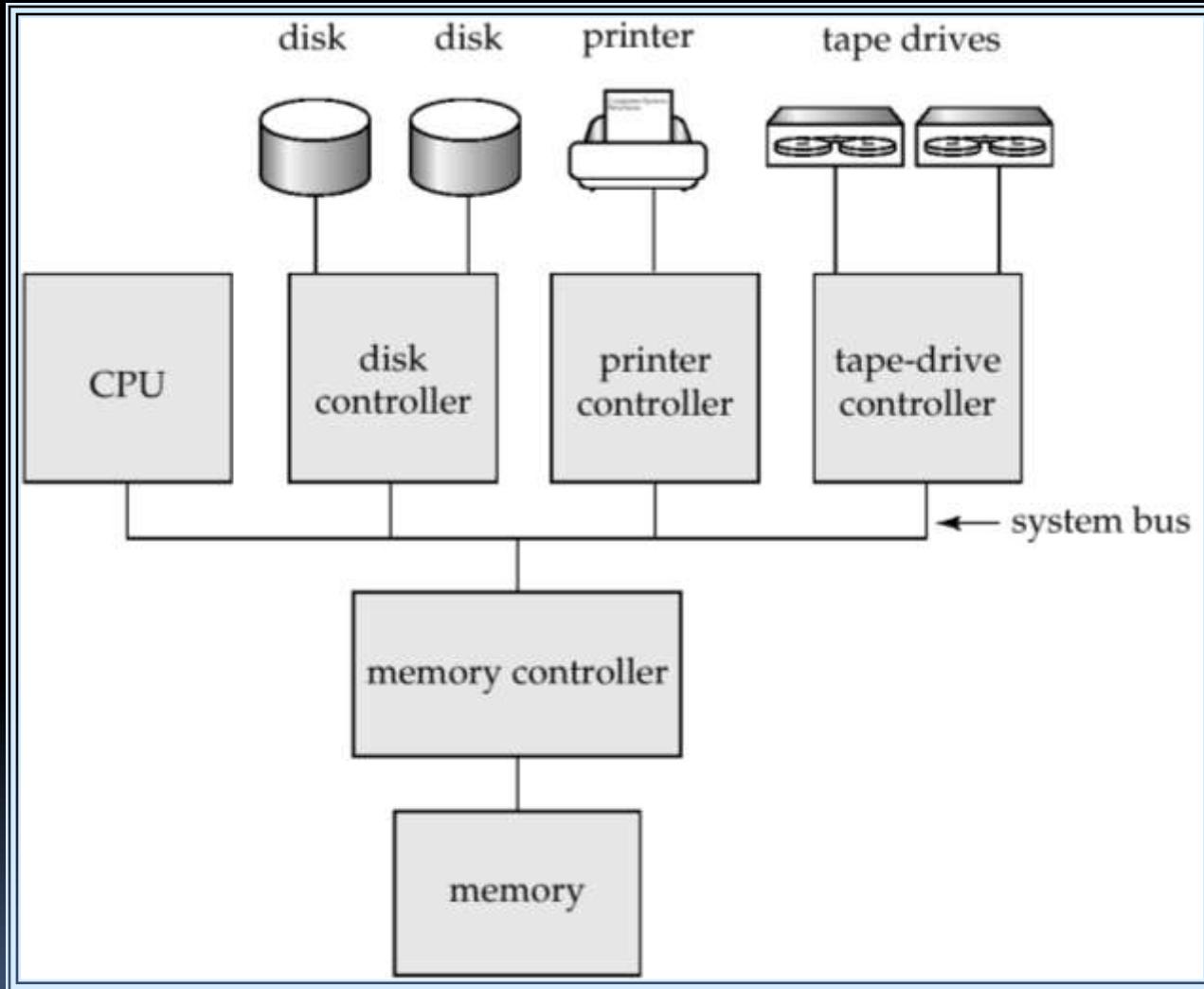
# Chapter 18: Database System Architectures

- Centralized Systems
- Client--Server Systems
- Parallel Systems
- Distributed Systems
- Network Types

# Centralized Systems

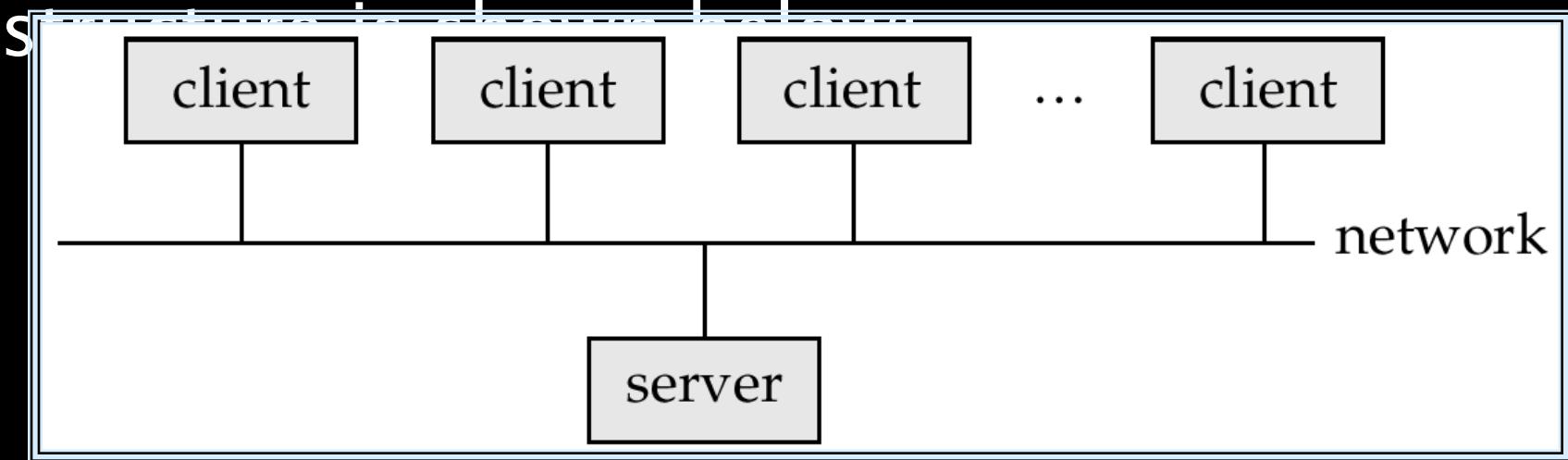
- Run on a single computer system and do not interact with other computer systems.
- General-purpose computer system: one to a few CPUs and a number of device controllers that are connected through a common bus that provides access to shared memory.
- Single-user system (e.g., personal computer or workstation): desk-top unit, single user, usually has only one CPU and one or two hard disks; the OS may support only one user.
- Multi-user system: more disks, more memory, multiple CPUs, and a multi-user OS. Serve a large number of users who are connected to the system via terminals. Often called *server* systems.

# A Centralized Computer System



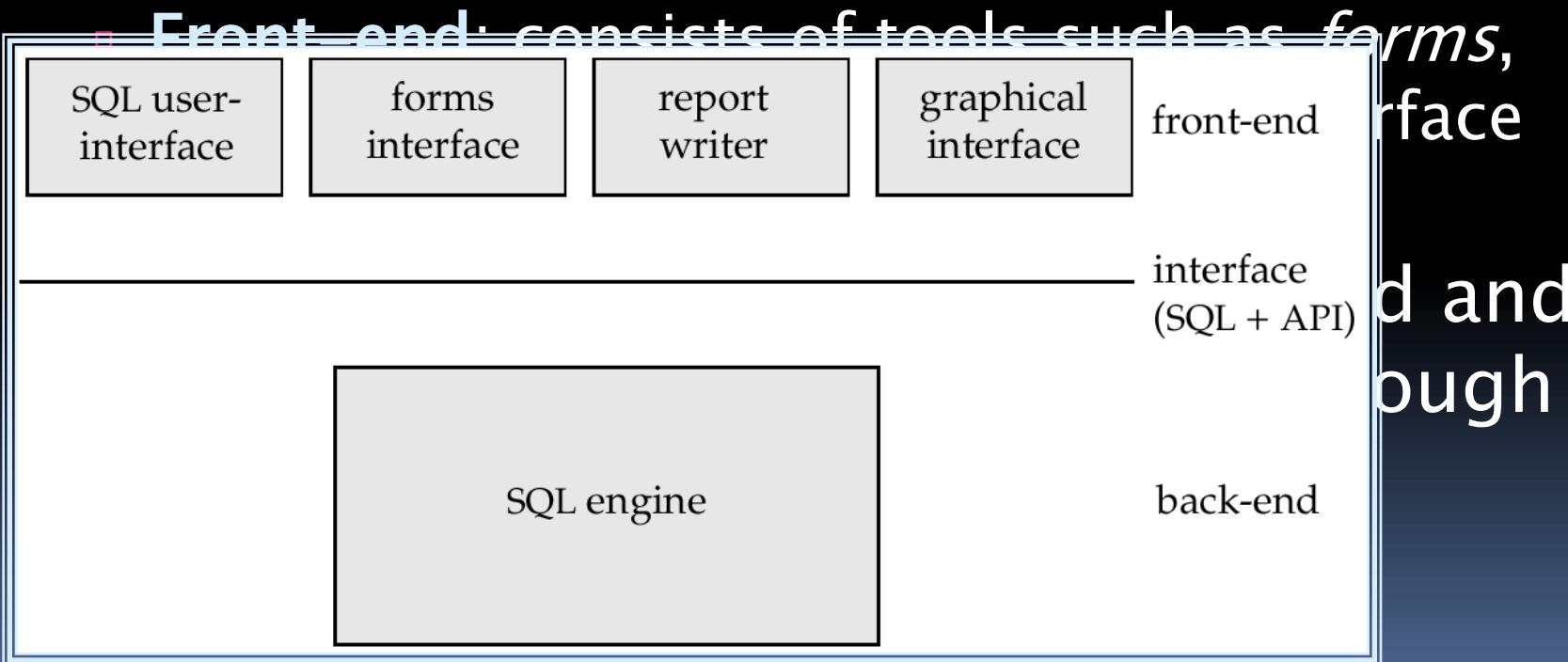
# Client-Server Systems

- Server systems satisfy requests generated at  $m$  client systems, whose general



# Client-Server Systems (Cont.)

- Database functionality can be divided into:
  - **Back-end:** manages access structures, query evaluation and optimization, concurrency control and recovery.



# Client-Server Systems (Cont.)

- Advantages of replacing mainframes with networks of workstations or personal computers connected to back-end server machines:
  - better functionality for the cost
  - flexibility in locating resources and expanding facilities
  - better user interfaces
  - easier maintenance
- Server systems can be broadly categorized into two kinds:
  - transaction servers which are widely used

# Transaction Servers

- Also called query server systems or SQL *server* systems; clients send requests to the server system where the transactions are executed, and results are shipped back to the client.
- Requests specified in SQL, and communicated to the server through a *remote procedure call (RPC)* mechanism.
- Transactional RPC allows many RPC calls to collectively form a transaction.

# Transaction Server Process Structure

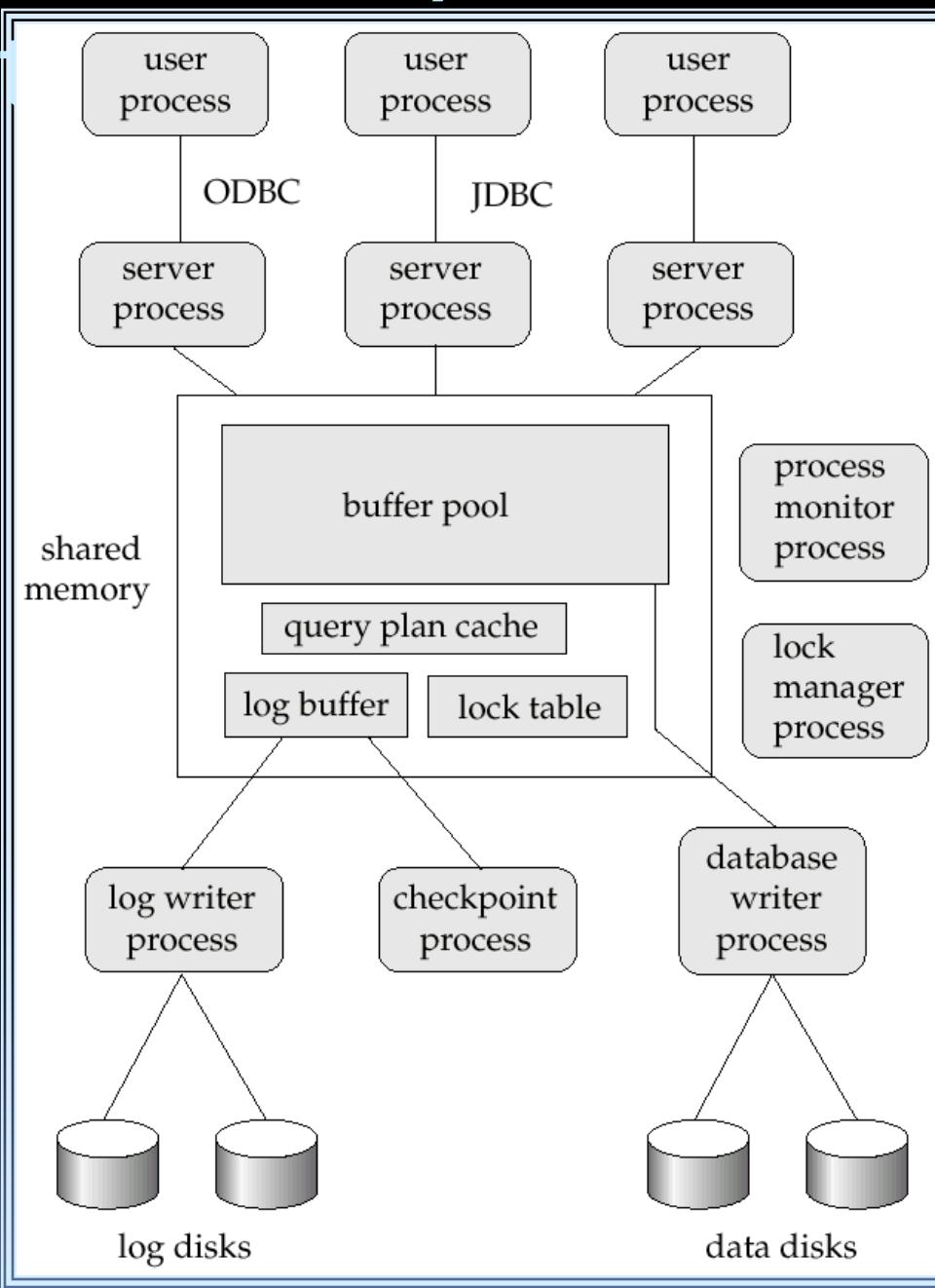
- A typical transaction server consists of multiple processes accessing data in shared memory.
- Server processes
  - These receive user queries (transactions), execute them and send results back
  - Processes may be **multithreaded**, allowing a single process to execute several user queries concurrently
  - Typically multiple multithreaded server processes

# Transaction Server Processes (Cont.)

- Log writer process
  - Server processes simply add log records to log record buffer
  - Log writer process outputs log records to stable storage.
- Checkpoint process
  - Performs periodic checkpoints
- Process monitor process
  - Monitors other processes, and takes recovery actions if any of the other processes fail
    - E.g. aborting any transactions being executed by a server process and restarting it

# Transaction System Processes

## (Con)



# Transaction System Processes (Cont.)

- Shared memory contains shared data
  - Buffer pool
  - Lock table
  - Log buffer
  - Cached query plans (reused if same query submitted again)
- All database processes can access shared memory
- To ensure that no two processes are accessing the same data structure at the same time, databases systems

# Transaction System Processes

- To avoid overhead of interprocess communication for lock request/grant, each database process operates directly on the lock table data structure (Section 16.1.4) instead of sending requests to lock manager process
  - Mutual exclusion ensured on the lock table using semaphores, or more commonly, atomic instructions
  - If a lock can be obtained, the lock table is updated directly in shared memory
  - If a lock cannot be immediately obtained, a lock request is noted in the lock table and the process (or thread) then waits for lock to be granted
  - When a lock is released, releasing process updates lock table to record release of lock, as well as grant of lock to waiting requests (if

# Data Servers

- Used in LANs, where there is a very high speed connection between the clients and the server, the client machines are comparable in processing power to the server machine, and the tasks to be executed are compute intensive.
- Ship data to client machines where processing is performed, and then ship results back to the server machine.

This architecture requires full back-

# Data Servers (Cont.)

- Page-Shipping versus Item-Shipping
  - Smaller unit of shipping  $\Rightarrow$  more messages
  - Worth **prefetching** related items along with requested item
  - Page shipping can be thought of as a form of prefetching
- Locking
  - Overhead of requesting and getting locks from server is high due to message delays
  - Can grant locks on requested and prefetched items; with page shipping, transaction is granted lock on whole page.
  - Locks on a prefetched item can be P{called back} by the server and returned by client

# Data Servers (Cont.)

- Data Caching

- Data can be cached at client even in between transactions
- But check that data is up-to-date before it is used (**cache coherency**)
- Check can be done when requesting lock on data item

- Lock Caching

- Locks can be retained by client system even in between transactions
- Transactions can acquire cached locks locally, without contacting server

# Parallel Systems

- Parallel database systems consist of multiple processors and multiple disks connected by a fast interconnection network.
- A coarse-grain parallel machine consists of a small number of powerful processors
- A massively parallel or fine grain parallel machine utilizes thousands of smaller processors.
- Two main performance measures:

# Speed-Up and Scale-Up

- Speedup: a fixed-sized problem executing on a small system is given to a system which is  $N$ -times larger.

- Measured by:

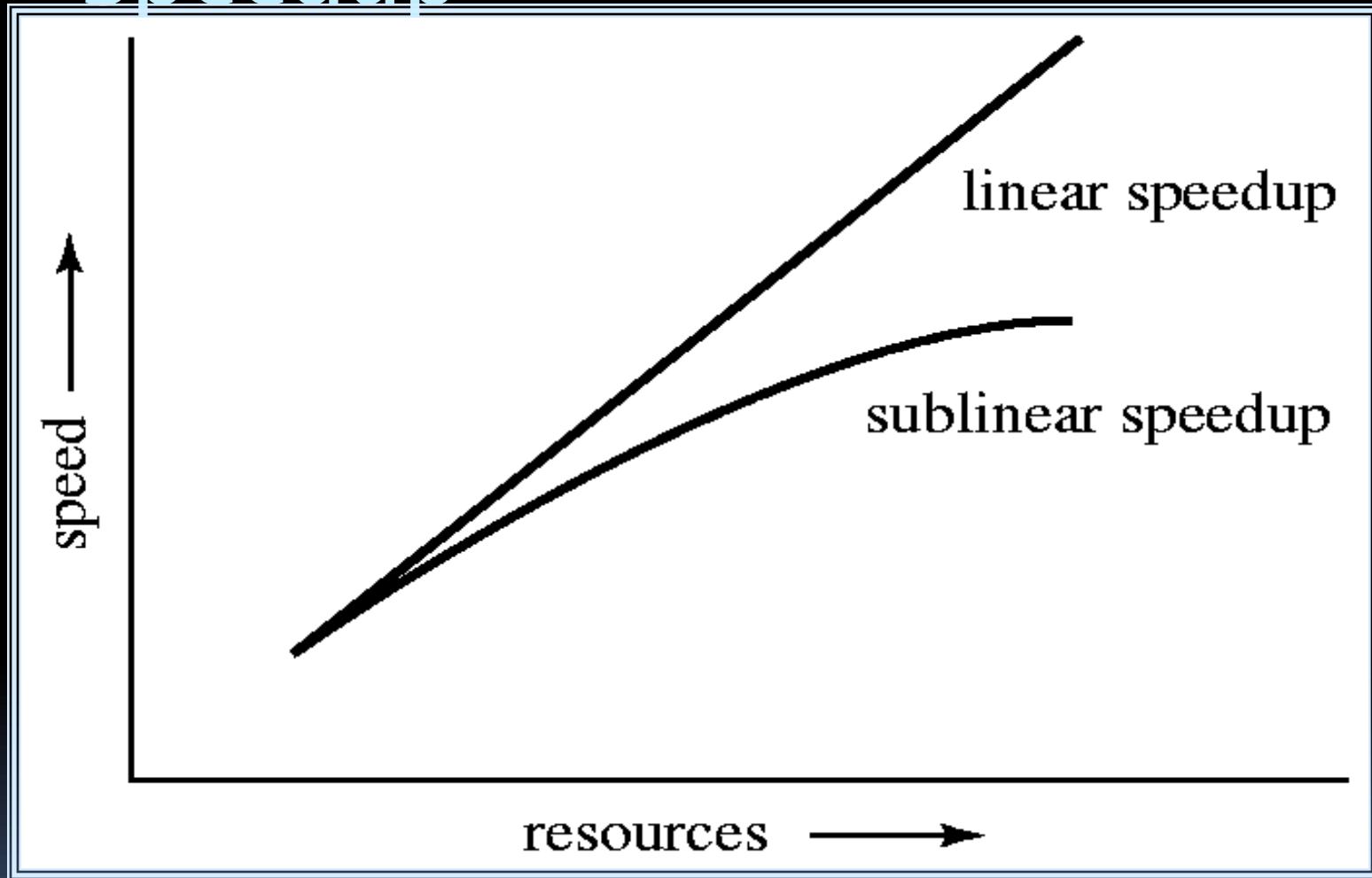
$$\text{speedup} = \frac{\text{small system elapsed time}}{\text{large system elapsed time}}$$

- Speedup is linear if equation equals  $N$ .

■ Scaleup: increase the size of both the problem and the system

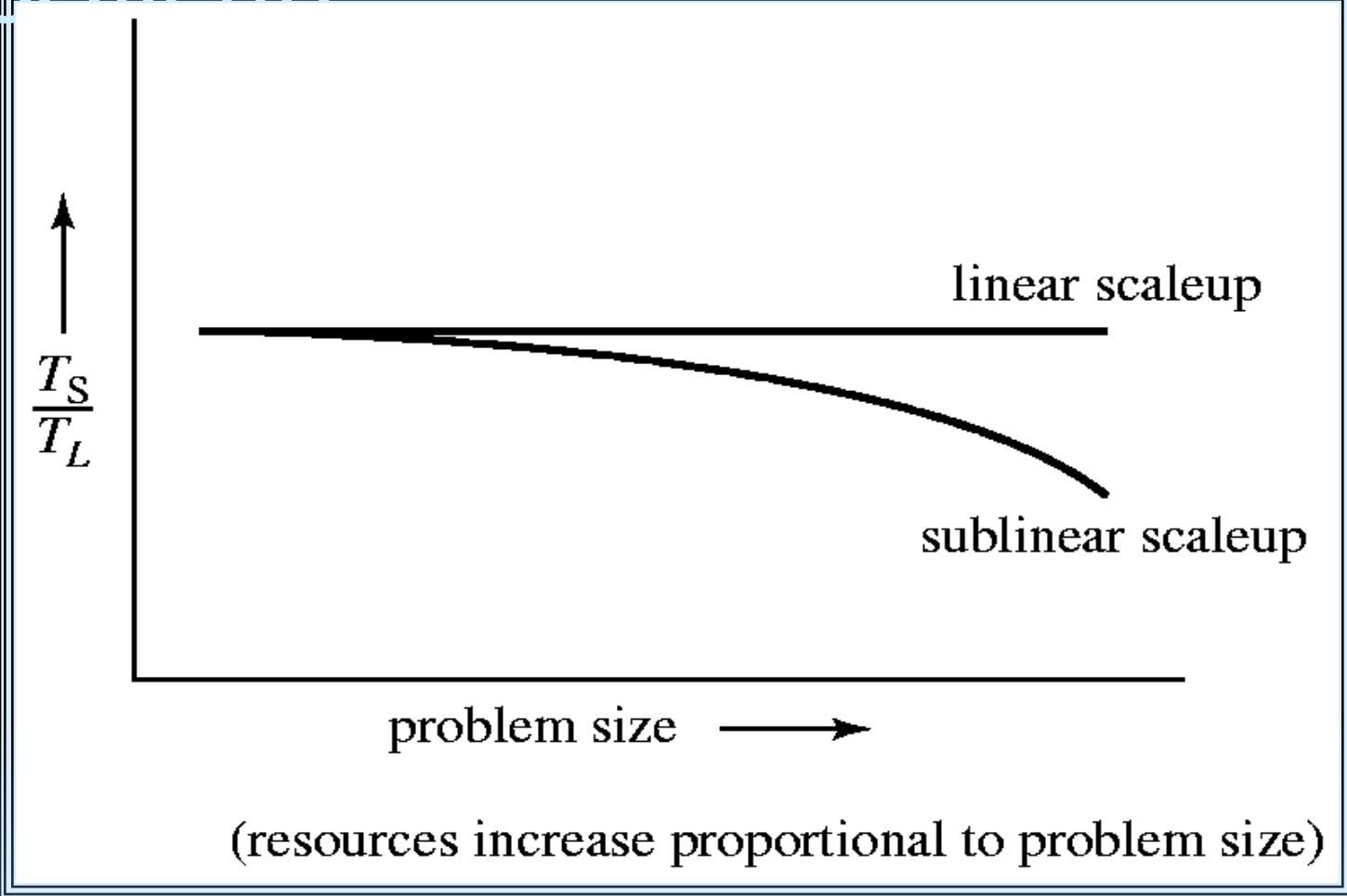
- $N$ -times larger system used to perform  $N$ -times larger job
  - Measured by:

# Speedup



Speedup

# Scaleup



Scaleup

# Batch and Transaction Scaleup

- Batch scaleup:
  - A single large job; typical of most database queries and scientific simulation.
  - Use an  $N$ -times larger computer on  $N$ -times larger problem.
- Transaction scaleup:
  - Numerous small queries submitted by independent users to a shared database; typical transaction processing and timesharing systems.
  - $N$ -times as many users submitting requests (hence,  $N$ -times as many requests) to an  $N$ -times larger database.

# Factors Limiting Speedup and Scaleup

Speedup and scaleup are often sublinear due to:

- Startup costs: Cost of starting up multiple processes may dominate computation time, if the degree of parallelism is high.
- Interference: Processes accessing shared resources (e.g., system bus, disks, or locks) compete with each other, thus spending time waiting on other processes, rather than performing useful work

# Interconnection Network

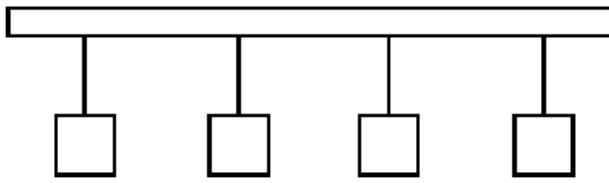
## Architectures

- Bus. System components send data on and receive data from a single communication bus;
  - Does not scale well with increasing parallelism.

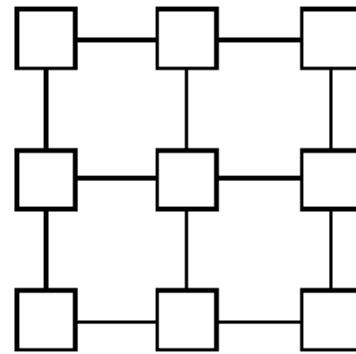
- Mesh. Components are arranged as nodes in a grid, and each component is connected to all adjacent components

- Communication links grow with growing number of components, and so scales better.
- But may require  $2\sqrt{n}$  hops to send message to a node (or  $\sqrt{n}$  with some assumptions)

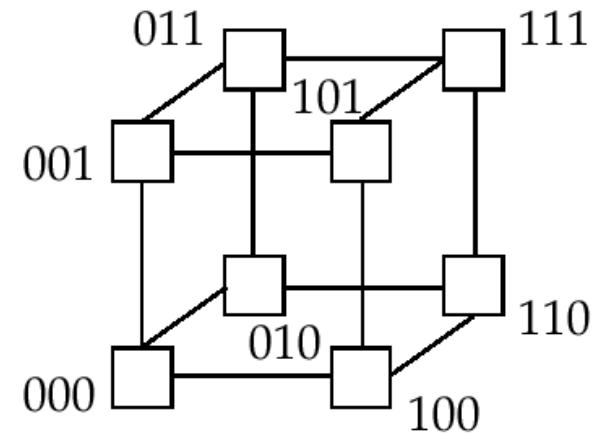
# Interconnection Architectures



(a) bus



(b) mesh

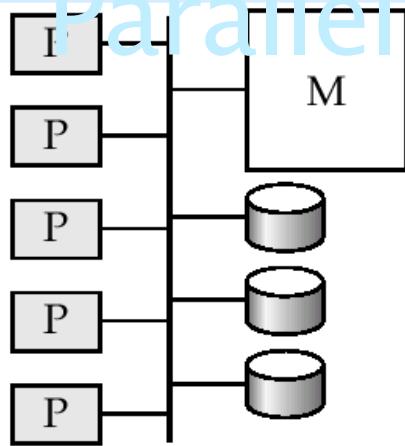


(c) hypercube

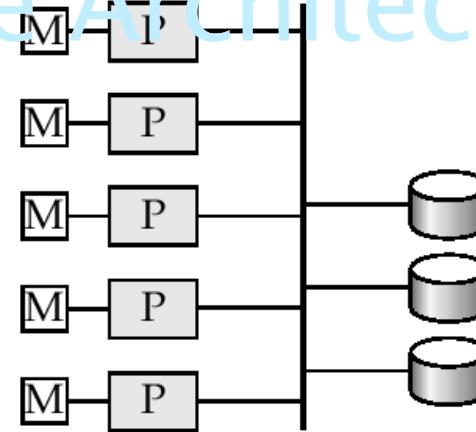
# Parallel Database Architectures

- Shared memory -- processors share a common memory
- Shared disk -- processors share a common disk
- Shared nothing -- processors share neither a common memory nor common disk
- Hierarchical -- hybrid of the above architectures

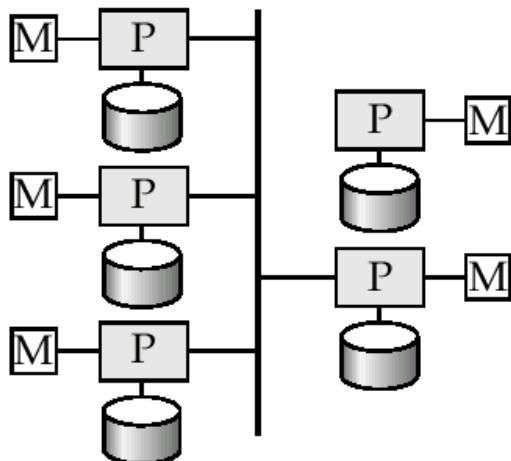
# Parallel Database Architectures



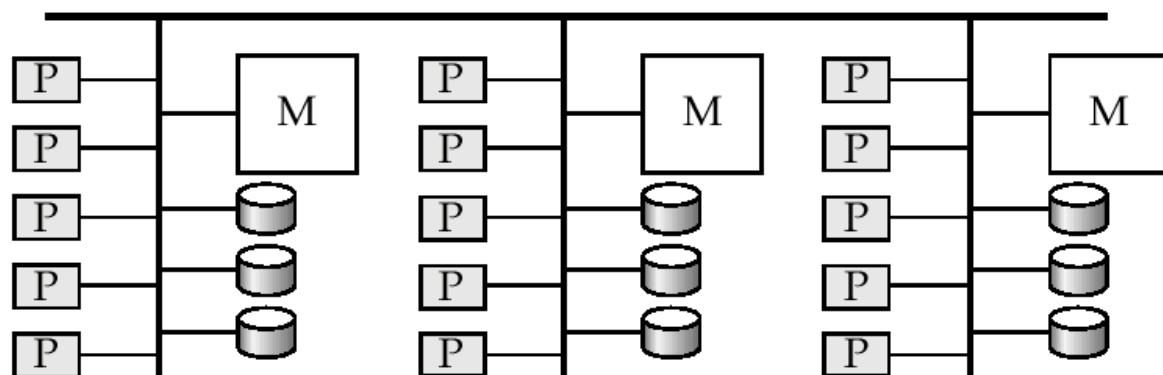
(a) shared memory



(b) shared disk



(c) shared nothing



(d) hierarchical

# Shared Memory

- Processors and disks have access to a common memory, typically via a bus or through an interconnection network.
- Extremely efficient communication between processors — data in shared memory can be accessed by any processor without having to move it using software.
- Downside - architecture is not scalable beyond 32 or 64 processors since the bus or the interconnection network

# Shared Disk

- All processors can directly access all disks via an interconnection network, but the processors have private memories.
  - The memory bus is not a bottleneck
  - Architecture provides a degree of **fault-tolerance** — if a processor fails, the other processors can take over its tasks since the database is resident on disks that are accessible from all processors.

Examples: IBM Sysplex and DEC clusters (now part of Compaq) running Rdb (now Oracle Rdb) were early

# Shared Nothing

- Node consists of a processor, memory, and one or more disks. Processors at one node communicate with another processor at another node using an interconnection network. A node functions as the server for the data on the disk or disks the node owns.
- Examples: Teradata, Tandem, Oracle-n CUBE
- Data accessed from local disks (and local memory accesses) do not pass through interconnection network

# Hierarchical

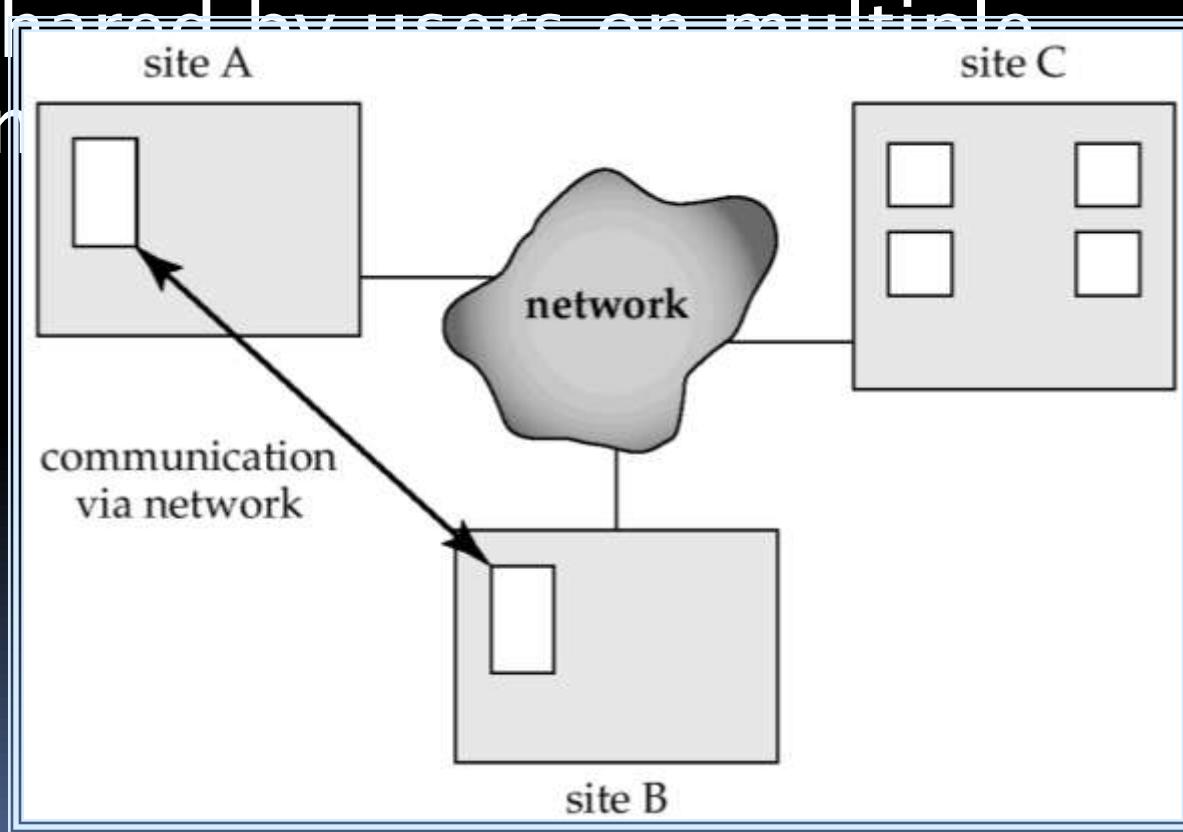
- Combines characteristics of shared-memory, shared-disk, and shared-nothing architectures.
- Top level is a shared-nothing architecture - nodes connected by an interconnection network, and do not share disks or memory with each other.
- Each node of the system could be a shared-memory system with a few processors.
- Alternatively, each node could be a

# Distributed Systems

- Data spread over multiple machines  
(also referred to as sites or nodes).

- Network interconnects the machines

- Data shared between multiple machines



# Distributed Databases

- Homogeneous distributed databases
  - Same software/schema on all sites, data may be partitioned among sites
  - Goal: provide a view of a single database, hiding details of distribution
- Heterogeneous distributed databases
  - Different software/schema on different sites
  - Goal: integrate existing databases to provide useful functionality
- Differentiate between *local* and *global*

# Trade-offs in Distributed

## Systems

- Sharing data – users at one site able to access the data residing at some other sites.
- Autonomy – each site is able to retain a degree of control over data stored locally.
- Higher system availability through redundancy — data can be replicated at remote sites, and system can function even if a site fails.  
Disadvantage: added complexity required to ensure proper coordination

# Implementation Issues for Distributed Databases

Atomicity needed even for transactions that update data at multiple site

- Transaction cannot be committed at one site and aborted at another
- The two-phase commit protocol (2PC) used to ensure atomicity
  - Basic idea: each site executes transaction till just before commit, and the leaves final decision to a coordinator
  - Each site must follow decision of coordinator: even if there is a failure while waiting for coordinators decision
    - To do so, updates of transaction are logged to stable storage and transaction is recorded as “waiting”

# Network Types

- Local-area networks (LANs) – composed of processors that are distributed over small geographical areas, such as a single building or a few adjacent buildings.
- Wide-area networks (WANs) – composed of processors distributed over a large geographical area.
- *Discontinuous connection* – WANs, such as those based on periodic dial-up (using, e.g., UUCP), that are connected only for part of the time

# Networks Types (Cont.)

- WANs with continuous connection are needed for implementing distributed database systems
- Groupware applications such as Lotus notes can work on WANs with discontinuous connection:
  - Data is replicated.
  - Updates are propagated to replicas periodically.
  - No global locking is possible, and copies of data may be independently updated.
  - Non-serializable executions can thus

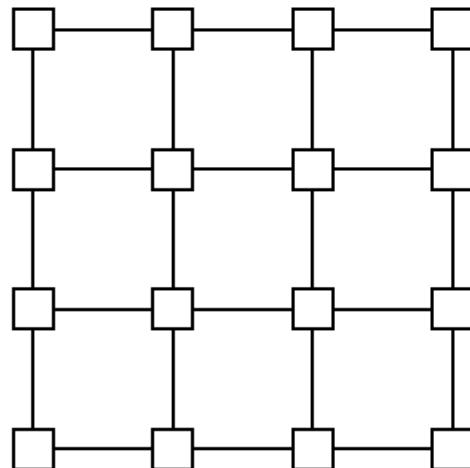
END OF CHAPTER



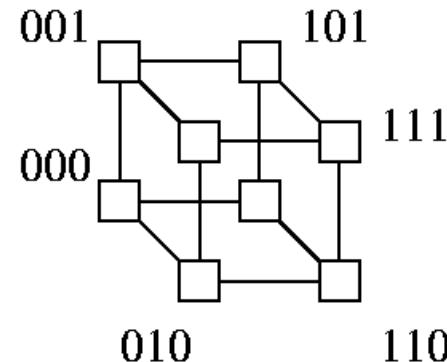
# Interconnection Networks



Bus Interconnection



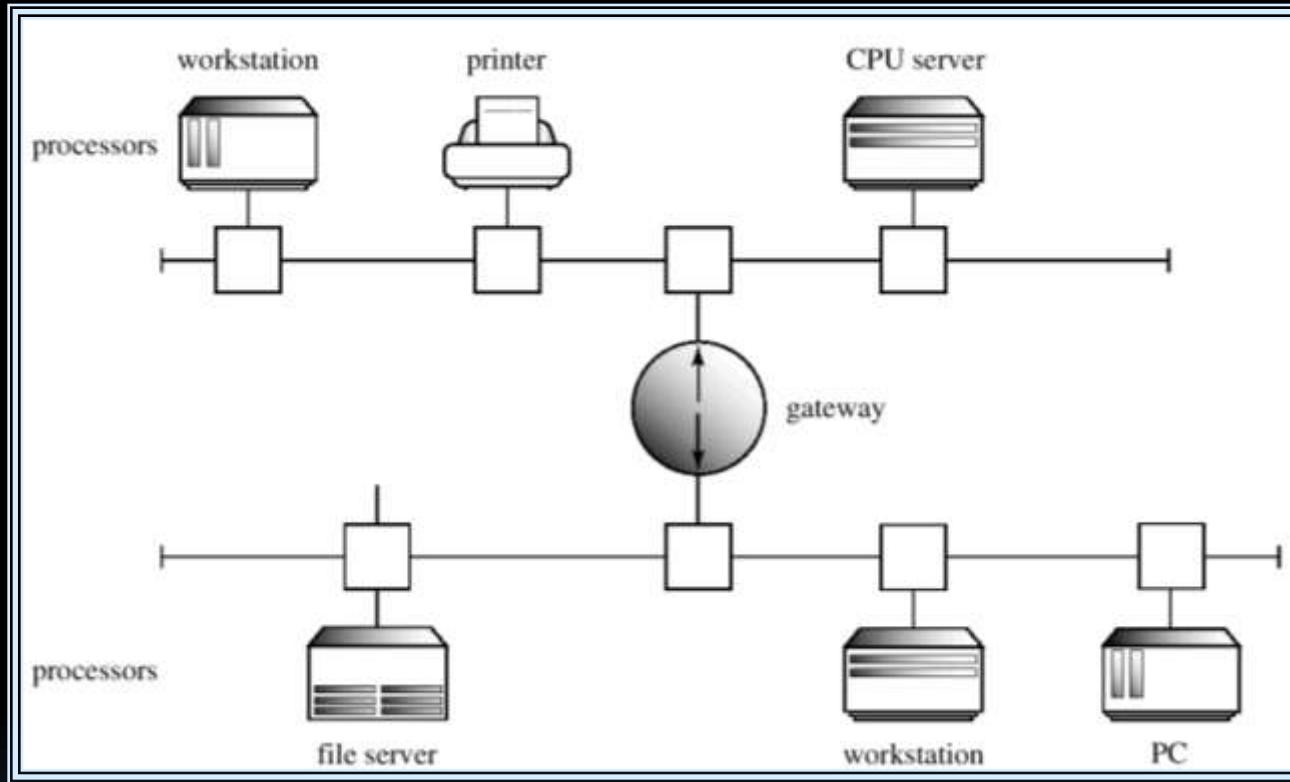
Mesh Interconnection



Hypercube Interconnection

# A Distributed System

# Local-Area Network



# Chapter 19: Distributed Databases

- Heterogeneous and Homogeneous Databases
- Distributed Data Storage
- Distributed Transactions
- Commit Protocols
- Concurrency Control in Distributed Databases
- Availability
- Distributed Query Processing
- Heterogeneous Distributed Databases
- Directory Systems

# Distributed Database System

- A distributed database system consists of loosely coupled sites that share no physical component
- Database systems that run on each site are independent of each other
- Transactions may access data at one or more sites

# Homogeneous Distributed

- **Databases** homogeneous distributed database
  - All sites have identical software
  - Are aware of each other and agree to cooperate in processing user requests.
  - Each site surrenders part of its autonomy in terms of right to change schemas or software
  - Appears to user as a single system
- In a heterogeneous distributed database
  - Different sites may use different schemas and software
    - Difference in schema is a major problem for

# Distributed Data Storage

- Assume relational data model
- Replication
  - System maintains multiple copies of data, stored in different sites, for faster retrieval and fault tolerance.
- Fragmentation
  - Relation is partitioned into several fragments stored in distinct sites
- Replication and fragmentation can be combined
  - Relation is partitioned into several fragments: system maintains several

# Data Replication

- A relation or fragment of a relation is replicated if it is stored redundantly in two or more sites.
- Full replication of a relation is the case where the relation is stored at all sites.
- Fully redundant databases are those in which every site contains a copy of the entire database.

# Data Replication (Cont.)

## ■ Advantages of Replication

- **Availability:** failure of site containing relation  $r$  does not result in unavailability of  $r$  if replicas exist.
- **Parallelism:** queries on  $r$  may be processed by several nodes in parallel.
- **Reduced data transfer:** relation  $r$  is available locally at each site containing a replica of  $r$ .

## ■ Disadvantages of Replication

- Increased cost of updates: each replica of relation  $r$  must be updated.
- Increased complexity of concurrency control: concurrent updates to distinct replicas may lead to inconsistent data over time.

# Data Fragmentation

- Division of relation  $r$  into fragments  $r_1, r_2, \dots, r_n$  which contain sufficient information to reconstruct relation  $r$ .
- Horizontal fragmentation: each tuple of  $r$  is assigned to one or more fragments
- Vertical fragmentation: the schema for relation  $r$  is split into several smaller schemas
  - All schemas must contain a common candidate key (or superkey) to ensure lossless join property.

# Horizontal Fragmentation of *account* Relation

<i>branch-name</i>	<i>account-number</i>	<i>balance</i>
Hillside	A-305	500
Hillside	A-226	336
Hillside	A-155	62

$account_1 = \sigma_{branch-name="Hillside"}(account)$

<i>branch-name</i>	<i>account-number</i>	<i>balance</i>
Valleyview	A-177	205
Valleyview	A-402	10000
Valleyview	A-408	1123
Valleyview	A-639	750

$account_2 = \sigma_{branch-name="Valleyview"}(account)$

# Vertical Fragmentation of *employee-info* Relation

<i>branch-name</i>	<i>customer-name</i>	<i>tuple-id</i>
Hillside	Lowman	1
Hillside	Camp	2
Valleyview	Camp	3
Valleyview	Kahn	4
Hillside	Kahn	5
Valleyview	Kahn	6
Valleyview	Green	7

$\text{deposit}_1 = \Pi_{\text{branch-name}, \text{customer-name}, \text{tuple-id}}(\text{employee-info})$

<i>account number</i>	<i>balance</i>	<i>tuple-id</i>
A-305	500	1
A-226	336	2
A-177	205	3
A-402	10000	4
A-155	62	5
A-408	1123	6
A-639	750	7

$\text{deposit}_2 = \Pi_{\text{account-number}, \text{balance}, \text{tuple-id}}(\text{employee-info})$

# Advantages of Fragmentation

- Horizontal:
  - allows parallel processing on fragments of a relation
  - allows a relation to be split so that tuples are located where they are most frequently accessed
- Vertical:
  - allows tuples to be split so that each part of the tuple is stored where it is most frequently accessed
  - tuple-id attribute allows efficient joining of vertical fragments
  - allows parallel processing on a relation

# Data Transparency

- Data transparency: Degree to which system user may remain unaware of the details of how and where the data items are stored in a distributed system
- Consider transparency issues in relation to:
  - Fragmentation transparency
  - Replication transparency
  - Location transparency

# Naming of Data Items – Criteria

- . Every data item must have a system-wide unique name.
- . It should be possible to find the location of data items efficiently.
- . It should be possible to change the location of data items transparently.
- . Each site should be able to create new data items autonomously.

# Centralized Scheme - Name

## Server Structure:

- name server assigns all names
- each site maintains a record of local data items
- sites ask name server to locate non-local data items

## Advantages:

- satisfies naming criteria 1–3

## Disadvantages:

- does not satisfy naming criterion 4
- name server is a potential performance bottleneck

# Use of Aliases

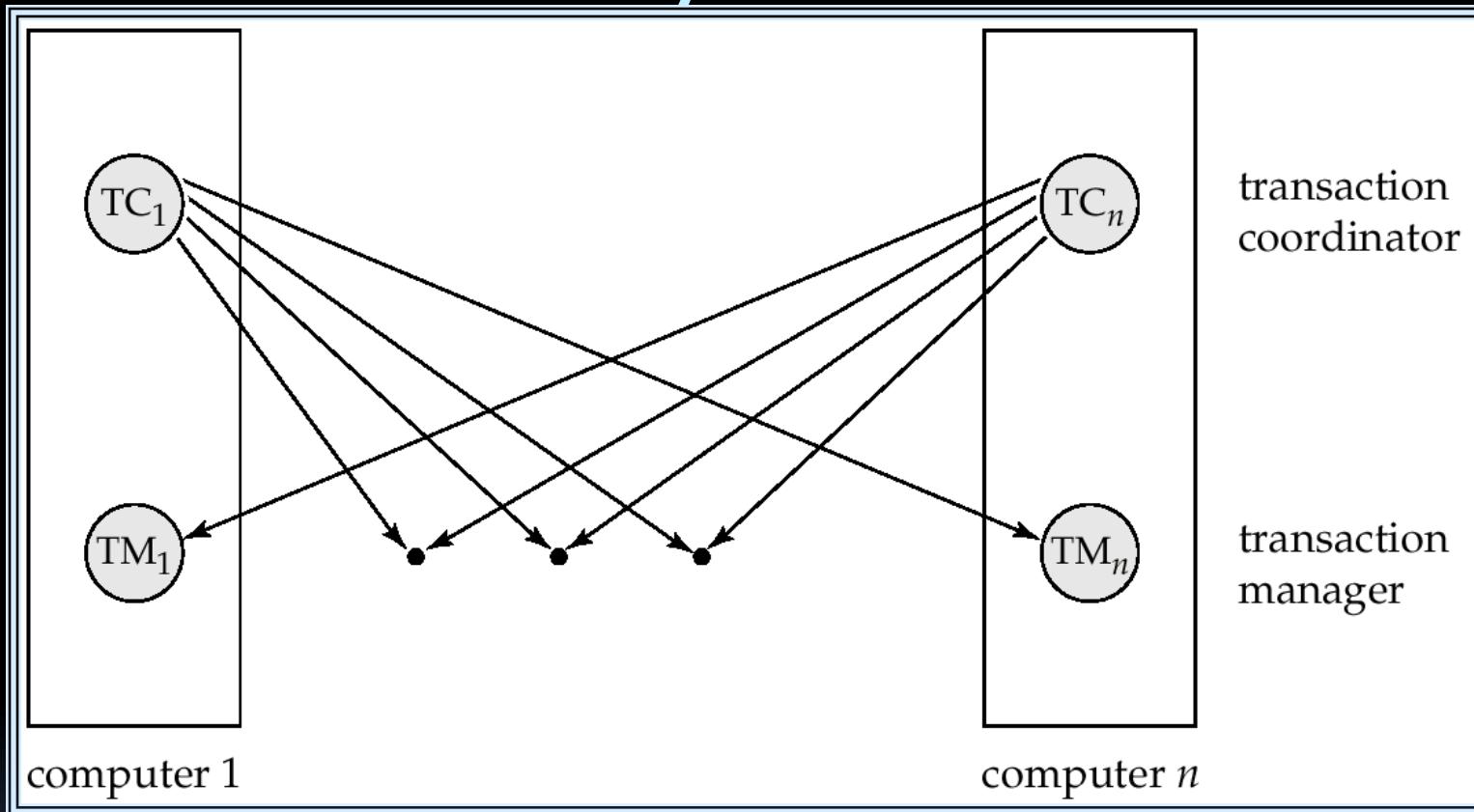
- Alternative to centralized scheme: each site prefixes its own site identifier to any name that it generates i.e., *site 17.account*.
  - Fulfils having a unique identifier, and avoids problems associated with central control.
  - However, fails to achieve network transparency.
- Solution: Create a set of aliases for data items; Store the mapping of aliases to the real names at each site.

# DISTRIBUTED TRANSACTIONS

# Distributed Transactions

- Transaction may access data at several sites.
- Each site has a local transaction manager responsible for:
  - Maintaining a log for recovery purposes
  - Participating in coordinating the concurrent execution of the transactions executing at that site.
- Each site has a transaction coordinator, which is responsible for:
  - Starting the execution of transactions that originate at the site

# Transaction System Architecture



# System Failure Modes

- Failures unique to distributed systems:
  - Failure of a site.
  - Loss of messages
    - Handled by network transmission control protocols such as TCP–IP
  - Failure of a communication link
    - Handled by network protocols, by routing messages via alternative links
  - Network partition
    - A network is said to be **partitioned** when it has been split into two or more subsystems that lack any connection between them
      - Note: a subsystem may consist of a single node

# Commit Protocols

- Commit protocols are used to ensure atomicity across sites
  - a transaction which executes at multiple sites must either be committed at all the sites, or aborted at all the sites.
  - not acceptable to have a transaction committed at one site and aborted at another
- The *two-phase commit* (2 PC) protocol is widely used
- The *three-phase commit* (3 PC) protocol is more complicated and

# Two Phase Commit Protocol

(2PC)

- Assumes fail-stop model – failed sites simply stop working, and do not cause any other harm, such as sending incorrect messages to other sites.
- Execution of the protocol is initiated by the coordinator after the last step of the transaction has been reached.
- The protocol involves all the local sites at which the transaction executed
- Let  $T$  be a transaction initiated at site  $S_i$ , and let the transaction coordinator

# Phase 1: Obtaining a Decision

- Coordinator asks all participants to *prepare* to commit transaction  $T_i$ ,
  - $C_i$  adds the records <prepare  $T$ > to the log and forces log to stable storage
  - sends **prepare  $T$**  messages to all sites at which  $T$  executed
- Upon receiving message, transaction manager at site determines if it can commit the transaction
  - if not, add a record <no  $T$ > to the log and send **abort  $T$**  message to  $C_i$ ,
  - if the transaction can be committed, then:

# Phase 2: Recording the Decision

- $T$  can be committed if  $C$ , received a ready  $T$  message from all the participating sites: otherwise  $T$  must be aborted.
- Coordinator adds a decision record,  $\langle \text{commit } T \rangle$  or  $\langle \text{abort } T \rangle$ , to the log and forces record onto stable storage. Once the record stable storage it is irrevocable (even if failures occur)
- Coordinator sends a message to each participant informing it of the decision (commit or abort)

# Handling of Failures – Site

When a site  $S$  recovers, it examines its log to determine the fate of transactions active at the time of the failure.

- Log contains  $\langle \text{commit } T \rangle$  record: site executes redo ( $T$ )
- Log contains  $\langle \text{abort } T \rangle$  record: site executes undo ( $T$ )
- Log contains  $\langle \text{ready } T \rangle$  record: site must consult  $C$ , to determine the fate of  $T$ .
  - If  $T$  committed, redo ( $T$ )

# Handling of Failures–

If coordinator fails while the commit protocol for  $T$  is executing then participating sites must decide on  $T$ 's fate:

1. If an active site contains a **<commit  $T$ >** record in its log, then  $T$  must be committed.
2. If an active site contains an **<abort  $T$ >** record in its log, then  $T$  must be aborted.
3. If some active participating site does not contain a **<ready  $T$ >** record in its log, then the failed coordinator  $C_i$  cannot have decided to commit  $T$ . Can therefore abort<sup>119</sup>  
 $T$ .

# Handling of Failures – Network Partition

- If the coordinator and all its participants remain in one partition, the failure has no effect on the commit protocol.
- If the coordinator and its participants belong to several partitions:
  - Sites that are not in the partition containing the coordinator think the coordinator has failed, and execute the protocol to deal with failure of the coordinator.
    - No harm results, but sites may still have to

# Recovery and Concurrency

## Control

- In-doubt transactions have a  $\langle \text{ready } T \rangle$ , but neither a  $\langle \text{commit } T \rangle$ , nor an  $\langle \text{abort } T \rangle$  log record.
- The recovering site must determine the commit–abort status of such transactions by contacting other sites; this can slow and potentially block recovery.
- Recovery algorithms can note lock information in the log.

# Three Phase Commit (3PC)

- Assumptions:
  - No network partitioning
  - At any point, at least one site must be up.
  - At most K sites (participants as well as coordinator) can fail
- Phase 1: Obtaining Preliminary Decision: Identical to 2PC Phase 1.
  - Every site is ready to commit if instructed to do so
- Phase 2 of 2PC is split into 2 phases, Phase 2 and Phase 3 of 3PC
  - In phase 2 coordinator makes a decision as in 2PC (called the pre-commit decision) and records it in multiple (at least K) sites
  - In phase 3, coordinator sends commit/abort message to all participating sites,
- Under 3PC, knowledge of pre-commit decision can be used to commit despite coordinator failure

# Alternative Models of Transaction Processing

- Notion of a single transaction spanning multiple sites is inappropriate for many applications
  - E.g. transaction crossing an organizational boundary
  - No organization would like to permit an externally initiated transaction to block local transactions for an indeterminate period
- Alternative models carry out transactions by sending messages
  - Code to handle messages must be carefully designed to ensure atomicity and

# Alternative Models (Cont.)

- Motivating example: funds transfer between two banks
  - Two phase commit would have the potential to block updates on the accounts involved in funds transfer
  - Alternative solution:
    - Debit money from source account and send a message to other site
    - Site receives message and credits destination account
  - Messaging has long been used for distributed transactions (even before computers were invented!)
- Atomicity issue
  - Once transaction sending a message is

# Error Conditions with Persistent Messaging

- Code to handle messages has to take care of variety of failure situations (even assuming guaranteed message delivery)
  - E.g. if destination account does not exist, failure message must be sent back to source site
  - When failure message is received from destination site, or destination site itself does not exist, money must be deposited back in source account
    - Problem if source account has been closed
      - get humans to take care of problem

# Persistent Messaging and Workflows

- Workflows provide a general model of transactional processing involving multiple sites and possibly human processing of certain steps
  - E.g. when a bank receives a loan application, it may need to
    - Contact external credit-checking agencies
    - Get approvals of one or more managers
  - and then respond to the loan application
  - We study workflows in Chapter 24 (Section 24.2)

# Implementation of Persistent Messaging

Sendup to SiteProtocols

1. Sending transaction writes message to a special relation *messages-to-send*. The message is also given a unique identifier.
  - ¶ Writing to this relation is treated as any other update, and is undone if the transaction aborts.
  - ¶ The message remains locked until the sending transaction commits
2. A message delivery process monitors the *messages-to-send* relation
  - ¶ When a new message is found, the message is sent to its destination
  - ¶ When an acknowledgment is received from a destination, the message is deleted from

# Implementation of Persistent Messaging

- Receiving site protocol
  - When a message is received

1. it is written to a *received-messages* relation if it is not already present (the message id is used for this check). The transaction performing the write is committed
2. An acknowledgement (with message id) is then sent to the sending site.

■ There may be very long delays in message delivery coupled with repeated messages

- Could result in processing of duplicate messages if we are not careful!
  - Option 1: messages are never deleted from *received-messages*
  - Option 2: messages are given timestamps
    - Messages older than some cut-off are deleted

# CONCURRENCY CONTROL IN DISTRIBUTED DATABASES

# Concurrency Control

- Modify concurrency control schemes for use in distributed environment.
- We assume that each site participates in the execution of a commit protocol to ensure global transaction atomicity.
- We assume all replicas of any item are updated
  - Will see how to relax this in case of site failures later

# Single-Lock-Manager Approach

- System maintains a *single* lock manager that resides in a *single* chosen site, say S,
- When a transaction needs to lock a data item, it sends a lock request to S, and lock manager determines whether the lock can be granted immediately
  - If yes, lock manager sends a message to the site which initiated the request
  - If no, request is delayed until it can be granted, at which time a message is sent to the initiator.

# Single-Lock-Manager Approach (Cont.)

- The transaction can read the data item from *any* one of the sites at which a replica of the data item resides.
- Writes must be performed on all replicas of a data item
- Advantages of scheme:
  - Simple implementation
  - Simple deadlock handling
- Disadvantages of scheme are:
  - Bottleneck: lock manager site becomes a bottleneck

# Distributed Lock Manager

- In this approach, functionality of locking is implemented by lock managers at each site
  - Lock managers control access to local data items
    - But special protocols may be used for replicas
- Advantage: work is distributed and can be made robust to failures
- Disadvantage: deadlock detection is more complicated
  - Lock managers cooperate for deadlock

# Primary Copy

- Choose one replica of data item to be the primary copy.
  - Site containing the replica is called the **primary site** for that data item
  - Different data items can have different primary sites
- When a transaction needs to lock a data item  $Q$ , it requests a lock at the primary site of  $Q$ .
  - Implicitly gets lock on all replicas of the data item

# Majority Protocol

- Local lock manager at each site administers lock and unlock requests for data items stored at that site.
- When a transaction wishes to lock an unreplicated data item  $Q$  residing at site  $S_i$ , a message is sent to  $S_i$ 's lock manager.
  - If  $Q$  is locked in an incompatible mode, then the request is delayed until it can be granted.
  - When the lock request can be granted, the lock manager sends a message back to the

# Majority Protocol (Cont.)

- In case of replicated data
  - If  $Q$  is replicated at  $n$  sites, then a lock request message must be sent to more than half of the  $n$  sites in which  $Q$  is stored.
  - The transaction does not operate on  $Q$  until it has obtained a lock on a majority of the replicas of  $Q$ .
  - When writing the data item, transaction performs writes on *all* replicas.
- Benefit
  - Can be used even when some sites are unavailable
  - details on how handle writes in the presence

# Biased Protocol

- Local lock manager at each site as in majority protocol, however, requests for shared locks are handled differently than requests for exclusive locks.
- Shared locks. When a transaction needs to lock data item  $Q$ , it simply requests a lock on  $Q$  from the lock manager at one site containing a replica of  $Q$ .
- Exclusive locks. When transaction needs to lock data item  $Q$ , it requests

# Quorum Consensus Protocol

- A generalization of both majority and biased protocols
- Each site is assigned a weight.
  - Let  $S$  be the total of all site weights
- Choose two values read quorum  $Q_r$  and write quorum  $Q_w$ 
  - Such that  $Q_r + Q_w > S$  and  $2 * Q_w > S$
  - Quorums can be chosen (and  $S$  computed) separately for each item
- Each read must lock enough replicas

# Deadlock Handling

Consider the following two transactions and history, with item X and transaction  $T_1$  at site 1, and item Y and transaction  $T_2$  at site 2:

X-lock on X  
write (X)

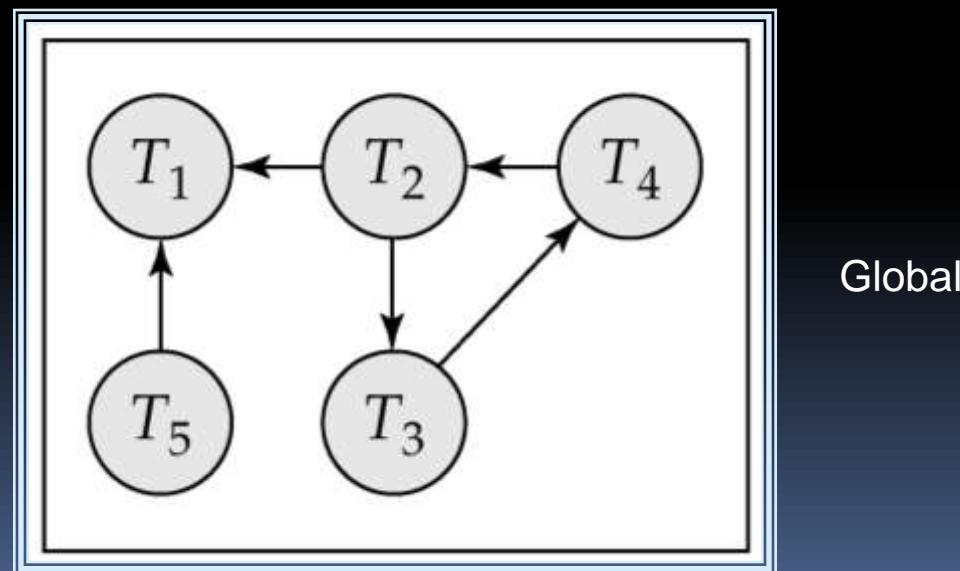
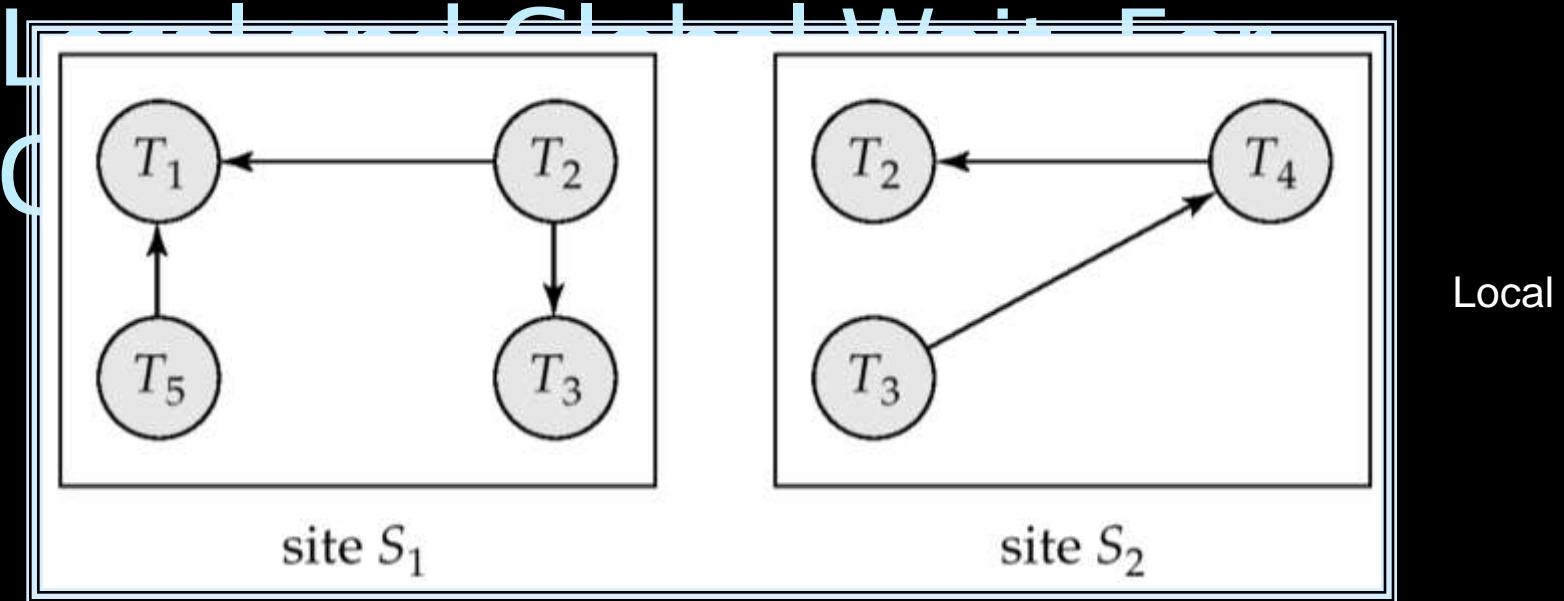
Wait for X-lock on Y

X-lock on Y  
write (Y)  
wait for X-lock on X

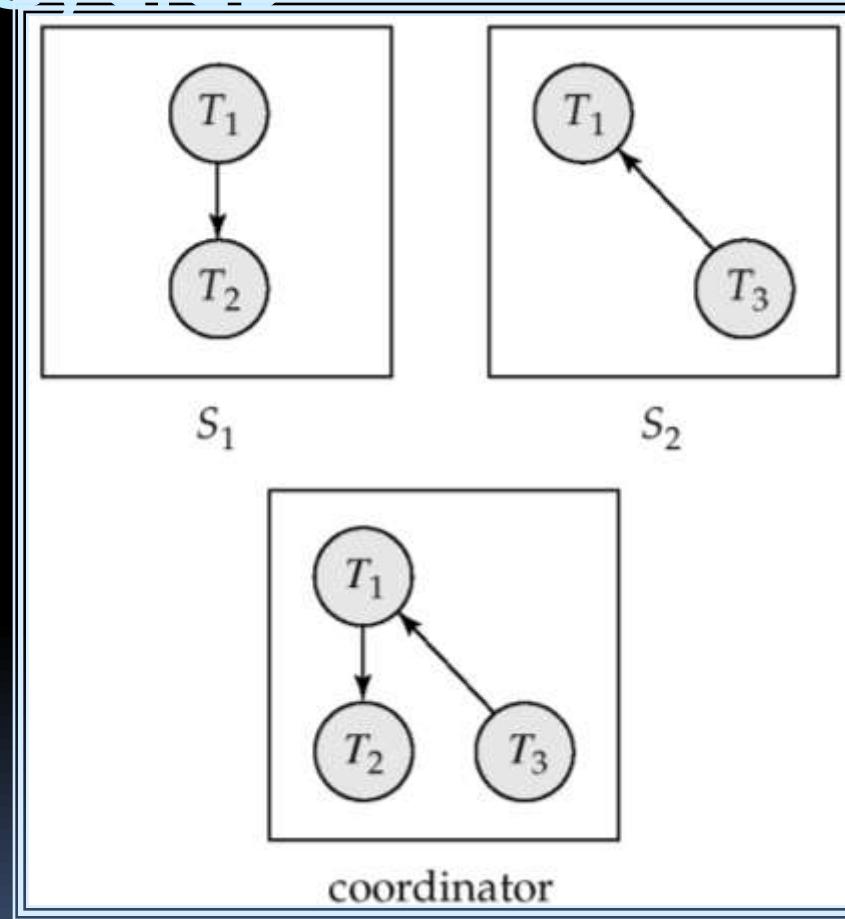
Result: deadlock which cannot be detected locally at either site

# Centralized Approach

- A global wait-for graph is constructed and maintained in a *single* site; the deadlock-detection coordinator
  - *Real graph*: Real, but unknown, state of the system.
  - *Constructed graph*: Approximation generated by the controller during the execution of its algorithm .
- the global wait-for graph can be constructed when:
  - a new edge is inserted in or removed from one of the local wait-for graphs.



# Example Wait-For Graph for Initial state: False Cycles



# False Cycles (Cont.)

- Suppose that starting from the state shown in figure,

- $T_2$  releases resources at  $S_1$ 
  - resulting in a message remove  $T_1 \rightarrow T_2$  message from the Transaction Manager at site  $S_1$  to the coordinator)
- And then  $T_2$  requests a resource held by  $T_3$  at site  $S_2$ 
  - resulting in a message insert  $T_2 \rightarrow T_3$  from  $S_2$  to the coordinator

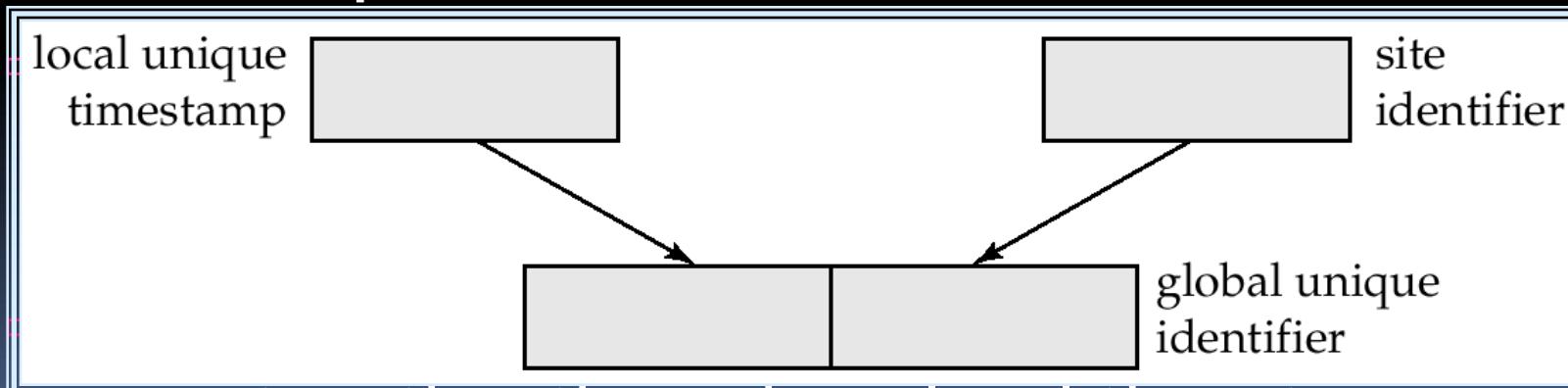
- Suppose further that the insert message reaches before the delete message
  - this can happen due to network delays

# Unnecessary Rollbacks

- Unnecessary rollbacks may result when deadlock has indeed occurred and a victim has been picked, and meanwhile one of the transactions was aborted for reasons unrelated to the deadlock.
- Unnecessary rollbacks can result from false cycles in the global wait-for graph; however, likelihood of false cycles is low.

# Timestamping

- Timestamp based concurrency-control protocols can be used in distributed systems
- Each transaction must be given a unique timestamp
- Main problem: how to generate a timestamp in a distributed fashion



concatenating the unique local timestamp with the unique identifier.

# Timestamping (Cont.)

- A site with a slow clock will assign smaller timestamps
  - Still logically correct: serializability not affected
  - But: “disadvantages” transactions
- To fix this problem
  - Define within each site  $S_i$ , a *logical clock* ( $LC_i$ ), which generates the unique local timestamp
  - Require that  $S_i$  advance its logical clock whenever a request is received from a transaction  $T_i$  with timestamp  $\langle x, y \rangle$  and

# Replication with Weak Consistency

- Many commercial databases support replication of data with weak degrees of consistency (i.e., without a guarantee of serializability)
- E.g.: master-slave replication:  
updates are performed at a single “master” site, and propagated to “slave” sites.
  - Propagation is not part of the update transaction: it is decoupled
    - May be immediately after transaction commits

# Replication with Weak Consistency (Cont.)

- Replicas should see a transaction-consistent snapshot of the database
  - That is, a state of the database reflecting all effects of all transactions up to some point in the serialization order, and no effects of any later transactions.
- E.g. Oracle provides a `create snapshot` statement to create a snapshot of a relation or a set of relations at a remote site
  - snapshot refresh either by recomputation  
or by a periodic update

# Multimaster Replication

- With multimaster replication (also called update-anywhere replication) updates are permitted at any replica, and are automatically propagated to all replicas
  - Basic model in distributed databases, where transactions are unaware of the details of replication, and database system propagates updates as part of the same transaction
    - Coupled with 2 phase commit
    - Many systems support lazy propagation

# Lazy Propagation (Cont.)

- Two approaches to lazy propagation
  - Updates at any replica translated into update at primary site, and then propagated back to all replicas
    - Updates to an item are ordered serially
    - But transactions may read an old value of an item and use it to perform an update, result in non-serializability
  - Updates are performed at any replica and propagated to all other replicas
    - Causes even more serialization problems:
      - Same data item may be updated concurrently at multiple sites!
- Conflict detection is a problem
  - Some conflicts due to lack of distributed

# AVAILABILITY

# Availability

- High availability: time for which system is not fully usable should be extremely low (e.g. 99.99% availability)
- Robustness: ability of system to function spite of failures of components
- Failures are more likely in large distributed systems
- To be robust, a distributed system must

# Reconfiguration

- Reconfiguration:
  - Abort all transactions that were active at a failed site
    - Making them wait could interfere with other transactions since they may hold locks on other sites
    - However, in case only some replicas of a data item failed, it may be possible to continue transactions that had accessed data at a failed site (more on this later)
  - If replicated data items were at failed site, update system catalog to remove them from the list of replicas.
    - This should be reversed when failed site

# Reconfiguration (Cont.)

- Since network partition may not be distinguishable from site failure, the following situations must be avoided
  - Two or more central servers elected in distinct partitions
  - More than one partition updates a replicated data item
- Updates must be able to continue even if some sites are down
- Solution: majority based approach
  - Alternative of “read one write all available”

# Majority-Based Approach

- The majority protocol for distributed concurrency control can be modified to work even if some sites are unavailable
  - Each replica of each item has a **version number** which is updated when the replica is updated, as outlined below
  - A lock request is sent to at least  $\frac{1}{2}$  the sites at which item replicas are stored and operation continues only when a lock is obtained on a majority of the sites
  - Read operations look at all replicas

# Majority-Based Approach

- Majority protocol (Cont.)
  - Write operations
    - find highest version number like reads, and set new version number to old highest version + 1
    - Writes are then performed on all locked replicas and version number on these replicas is set to new version number
  - Failures (network and site) cause no problems as long as
    - Sites at commit contain a majority of replicas of any updated data items
    - During reads a majority of replicas are available to find version numbers
    - Subject to above, 2 phase commit can be used to update replicas

# Read One Write All (Available)

- Biased protocol is a special case of quorum consensus
  - Allows reads to read any one replica but updates require all replicas to be available at commit time (called **read one write all**)
- Read one write all available (ignoring failed sites) is attractive, but incorrect
  - If failed link may come back up, without a disconnected site ever being aware that it was disconnected
  - The site then has old values, and a read from that site would return an old value

# Site Reintegration

- When failed site recovers, it must catch up with all updates that it missed while it was down
  - Problem: updates may be happening to items whose replica is stored at the site while the site is recovering
  - Solution 1: halt all updates on system while reintegrating a site
    - Unacceptable disruption
  - Solution 2: lock all replicas of all data items at the site, update to latest version, then release locks

# Comparison with Remote Backup

- Remote backup (hot spare) systems (Section 17.10) are also designed to provide high availability
- Remote backup systems are simpler and have lower overhead
  - All actions performed at a single site, and only log records shipped
  - No need for distributed concurrency control, or 2 phase commit
- Using distributed databases with replicas of data items can provide

# Coordinator Selection

## Backup coordinators

- site which maintains enough information locally to assume the role of coordinator if the actual coordinator fails
- executes the same algorithms and maintains the same internal state information as the actual coordinator fails
- executes state information as the actual coordinator
- allows fast recovery from coordinator failure but involves overhead during normal processing.

# Bully Algorithm

- If site  $S_i$  sends a request that is not answered by the coordinator within a time interval  $T$ , assume that the coordinator has failed  $S_i$  tries to elect itself as the new coordinator.
- $S_i$  sends an election message to every site with a higher identification number,  $S_i$  then waits for any of these processes to answer within  $T$ .
- If no response within  $T$ , assume that all sites with number greater than  $i$  have failed  $S_i$  elects itself the new

# Bully Algorithm (Cont.)

- If no message is sent within  $T'$ , assume the site with a higher number has failed;  $S_i$  restarts the algorithm.
- After a failed site recovers, it immediately begins execution of the same algorithm.
- If there are no active sites with higher numbers, the recovered site forces all processes with lower numbers to let it become the coordinator site, even if there is a currently active coordinator with a lower number.

# DISTRIBUTED QUERY PROCESSING

# Distributed Query Processing

- For centralized systems, the primary criterion for measuring the cost of a particular strategy is the number of disk accesses.
- In a distributed system, other issues must be taken into account:
  - The cost of a data transmission over the network.
  - The potential gain in performance from having several sites process parts of the query in parallel.

# Query Transformation

- Translating algebraic queries on fragments.
  - It must be possible to construct relation  $r$  from its fragments
  - Replace relation  $r$  by the expression to construct relation  $r$  from its fragments
- Consider the horizontal fragmentation of the *account* relation into

$account_1 = \sigma_{branch-name = "Hillside"} (account)$

$account_2 = \sigma_{branch-name = "Valleyview"} (account)$

- The query  $\sigma_{branch-name = "Hillside"} (account)$  becomes

# Example Query (Cont.)

- Since  $account_1$  has only tuples pertaining to the Hillside branch, we can eliminate the selection operation.

- Apply the definition of  $account_2$  to obtain

$$\sigma_{branch-name = "Hillside"} (\sigma_{branch-name = "Valleyview"} (account))$$

- This expression is the empty set regardless of the contents of the  $account$  relation.
- Final strategy is for the Hillside site to return  $account_1$  as the result of the query.

# Simple Join Processing

- Consider the following relational algebra expression in which the three relations are neither replicated nor fragmented

*account depositor branch*

- account* is stored at site  $S_1$

*depositor* at  $S_2$

*branch* at  $S_3$

- For a query issued at site  $S_1$ , the system needs to produce the result at site  $S_1$

# Possible Query Processing

- Strategies
  - Ship copies of all three relations to site  $S_1$  and choose a strategy for processing the entire locally at site  $S_1$ .
  - Ship a copy of the account relation to site  $S_2$  and compute  $\text{temp}_1 = \text{account depositor}$  at  $S_2$ . Ship  $\text{temp}_1$  from  $S_2$  to  $S_3$ , and compute  $\text{temp}_2 = \text{temp}_1 \text{ branch}$  at  $S_3$ . Ship the result  $\text{temp}_2$  to  $S_1$ .
  - Devise similar strategies, exchanging the roles  $S_1, S_2, S_3$

# Semijoin Strategy

- Let  $r_1$  be a relation with schema  $R_1$  stores at site  $S_1$   
Let  $r_2$  be a relation with schema  $R_2$  stores at site  $S_2$
- Evaluate the expression  $r_1 \bowtie r_2$  and obtain the result at  $S_1$ .
  - Compute  $temp_1 \leftarrow \Pi_{R1 \cap R2}(r1)$  at  $S_1$ .
  - Ship  $temp_1$  from  $S_1$  to  $S_2$ .
  - Compute  $temp_2 \leftarrow r_2 \bowtie temp_1$  at  $S_2$
  - Ship  $temp_2$  from  $S_2$  to  $S_1$ .
  - Compute  $r_1 \bowtie temp_2$  at  $S_1$ . This is the result.

# Formal Definition

- The semijoin of  $r_1$  with  $r_2$ , is denoted by:

$$r_1 \bowtie r_2$$

- it is defined by:
  - $\prod_{R1} (r_1 \bowtie r_2)$
- Thus,  $r_1 \bowtie r_2$  selects those tuples of  $r_1$  that contributed to  $r_1 \bowtie r_2$ .
- In step 3 above,  $temp_2 = r_2 \bowtie r_1$ .
- For joins of several relations, the above strategy can be extended to a series of semijoin steps.

# Join Strategies that Exploit Parallelism

- Consider  $r_1 \bowtie r_2 \bowtie r_3 \bowtie r_4$  where relation  $r_i$  is stored at site  $S_i$ . The result must be presented at site  $S_1$ .
  - $r_1$  is shipped to  $S_2$  and  $r_1 \bowtie r_2$  is computed at  $S_2$ ; simultaneously  $r_3$  is shipped to  $S_4$  and  $r_3 \bowtie r_4$  is computed at  $S_4$
  - $S_2$  ships tuples of  $(r_1 \bowtie r_2)$  to  $S_1$  as they produced;
  - $S_4$  ships tuples of  $(r_3 \bowtie r_4)$  to  $S_1$

# Heterogeneous Distributed Databases

- Many database applications require data from a variety of preexisting databases located in a heterogeneous collection of hardware and software platforms
- Data models may differ (hierarchical, relational , etc.)
- Transaction commit protocols may be incompatible
- Concurrency control may be based on different techniques

# Advantages

- Preservation of investment in existing
  - hardware
  - system software
  - Applications
- Local autonomy and administrative control
- Allows use of special-purpose DBMSs
- Step towards a unified homogeneous DBMS
  - Full integration into a homogeneous DBMS faces

# Unified View of Data

- Agreement on a common data model
  - Typically the relational model
- Agreement on a common conceptual schema
  - Different names for same relation/attribute
  - Same relation/attribute name means different things
- Agreement on a single representation of shared data
  - E.g. data types, precision,
  - Character sets
    - ASCII vs EBCDIC

# Query Processing

- Several issues in query processing in a heterogeneous database
- Schema translation
  - Write a **wrapper** for each data source to translate data to a global schema
  - Wrappers must also translate updates on global schema to updates on local schema
- Limited query capabilities
  - Some data sources allow only restricted forms of selections
    - E.g. web forms, flat file data sources
  - Queries have to be broken up and processed partly at the source and partly at a different

# Mediator Systems

- Mediator systems are systems that integrate multiple heterogeneous data sources by providing an integrated global view, and providing query facilities on global view
  - Unlike full fledged multidatabase systems, mediators generally do not bother about transaction processing
  - But the terms mediator and multidatabase are sometimes used interchangeably
  - The term virtual database is also used to refer to mediator/multidatabase systems

# DISTRIBUTED DIRECTORY SYSTEMS

# Directory Systems

- Typical kinds of directory information
  - Employee information such as name, id, email, phone, office addr, ..
  - Even personal information to be accessed from multiple places
    - e.g. Web browser bookmarks
- White pages
  - Entries organized by name or identifier
    - Meant for forward lookup to find more about an entry
- Yellow pages
  - Entries organized by properties
    - For reverse lookup to find entries matching

# Directory Access Protocols

- Most commonly used directory access protocol:
  - LDAP (Lightweight Directory Access Protocol)
  - Simplified from earlier X.500 protocol
- Question: Why not use database protocols like ODBC/JDBC?
- Answer:
  - Simplified protocols for a limited type of data access, evolved parallel to ODBC/JDBC
  - Provide a nice hierarchical naming

# LDAP: Lightweight Directory Access Protocol

- LDAP Data Model
- Data Manipulation
- Distributed Directory Trees

# LDAP Data Model

- LDAP directories store entries
  - Entries are similar to objects
- Each entry must have unique distinguished name (DN)
- DN made up of a sequence of relative distinguished names (RDNs)
- E.g. of a DN
  - cn=Silberschatz, ou=Bell Labs, o=Lucent, c=USA
  - Standard RDNs (can be specified as part of schema)

# LDAP Data Model (Cont.)

- Entries can have attributes
  - Attributes are multi-valued by default
  - LDAP has several built-in types
    - Binary, string, time types
    - Tel: telephone number              PostalAddress:  
postal address
- LDAP allows definition of object classes
  - Object classes specify attribute names and types
  - Can use inheritance to define object

# LDAP Data Model (cont.)

- Entries organized into a directory information tree according to their DNs
  - Leaf level usually represent specific objects
  - Internal node entries represent objects such as organizational units, organizations or countries
  - Children of a node inherit the DN of the parent, and add on RDNs
    - E.g. internal node with DN `c=USA`
      - Children nodes have DN starting with `c=USA`

# LDAP Data Manipulation

- Unlike SQL, LDAP does not define DDL or DML
- Instead, it defines a network protocol for DDL and DML
  - Users use an API or vendor specific front ends
  - LDAP also defines a file format
    - LDAP Data Interchange Format (LDIF)
- Querying mechanism is very simple: only selection & projection

# LDAP Queries

- LDAP query must specify
  - Base: a node in the DIT from where search is to start
  - A search condition
    - Boolean combination of conditions on attributes of entries
      - Equality, wild-cards and approximate equality supported
  - A scope
    - Just the base, the base and its children, or the entire subtree from the base
  - Attributes to be returned

# LDAP URLs

- First part of URL specifies server and DN of base
  - `ldap://aura.research.bell-labs.com/o=Lucent,c=USA`
- Optional further parts separated by ? symbol
  - `ldap://aura.research.bell-labs.com/o=Lucent,c=USA??sub?cn=Korth`
  - Optional parts specify
    1. attributes to return (empty means all)
    2. Scope (sub indicates entire subtree)
    3. Search condition (`cn=Korth`)

# C Code using LDAP API

```
#include <stdio.h>
#include <ldap.h>
main() {
 LDAP *ld;
 LDAPMessage *res, *entry;
 char *dn, *attr, *attrList [] =
{“telephoneNumber”, NULL};
 BerElement *ptr;
 int vals, i;
 // Open a connection to server
 ld = ldap_open(“aura.research.bell-
labs.com” LDAP_PORT);
```

# C Code using LDAP API (Cont.)

```
ldap_search_s(id, "o=Lucent, c=USA", LDAP_SCOPE_SUBTREE,
 "cn=Korth", attrList, /* attrsonly*/
0, &res);
/*attrsonly = 1 => return only schema
not actual results*/ printf("found%d
entries", ldap_count_entries(id, res));
for (entry=ldap_first_entry(id, res); entry != NULL;
 entry=ldap_next_entry(id, entry)) {
 dn = ldap_get_dn(id, entry);
 printf("dn: %s", dn); /* dn: DN of matching
entry */
 ldap_memfree(dn);
 for(attr = ldap_first_attribute(id, entry, &ptr):
```

# LDAP API (Cont.)

- LDAP API also has functions to create, update and delete entries
- Each function call behaves as a separate transaction
  - LDAP does not support atomicity of updates

# Distributed Directory Trees

- Organizational information may be split into multiple directory information trees
  - Suffix of a DIT gives RDN to be tagged onto to all entries to get an overall DN
    - E.g. two DITs, one with suffix o=Lucent, c=USA and another with suffix o=Lucent, c=India
  - Organizations often split up DITs based on geographical location or by organizational structure
  - Many LDAP implementations support replication (master-slave or multi-master replication) of DITs (not part of LDAP 3 standard)

# END OF CHAPTER EXTRA SLIDES (MATERIAL NOT IN BOOK)

- 1. 6-phase frommit
- 2. Fully distributed deadlock detection
- 3. Naming transparency
- 4. Network topologies

# Three Phase Commit (3PC)

- Assumptions:
  - No network partitioning
  - At any point, at least one site must be up.
  - At most K sites (participants as well as coordinator) can fail
- Phase 1: Obtaining Preliminary Decision:  
Identical to 2PC Phase 1.
  - Every site is ready to commit if instructed to do so
  - Under 2 PC each site is obligated to wait for decision from coordinator

# Phase 2. Recording the Preliminary Decision

- Coordinator adds a decision record ( $\langle \text{abort } T \rangle$  or  $\langle \text{precommit } T \rangle$ ) in its log and forces record to stable storage.
- Coordinator sends a message to each participant informing it of the decision
- Participant records decision in its log
  - If abort decision reached then participant aborts locally
  - If pre-commit decision reached then participant replies with  $\langle \text{acknowledge} \rangle$

# Phase 3. Recording Decision in the Database

Executed only if decision in phase 2 was to precommit

- Coordinator collects acknowledgements. It sends  $\langle \text{commit } T \rangle$  message to the participants as soon as it receives  $K$  acknowledgements.
- Coordinator adds the record  $\langle \text{commit } T \rangle$  in its log and forces record to stable storage.

# Handling Site Failure

- Site Failure. Upon recovery, a participating site examines its log and does the following:
  - Log contains **<commit  $T$ >** record: site executes **redo ( $T$ )**
  - Log contains **<abort  $T$ >** record: site executes **undo ( $T$ )**
  - Log contains **<ready  $T$ >** record, but no **<abort  $T$ >** or **<precommit  $T$ >** record: site consults  $C_i$  to determine the fate of  $T$ .
    - if  $C_i$  says  $T$  aborted, site executes **undo ( $T$ )** (and writes **<abort  $T$ >** record)

# Handling Site Failure (Cont.)

- Log contains  $\langle \text{precommit } T \rangle$  record, but no  $\langle \text{abort } T \rangle$  or  $\langle \text{commit } T \rangle$ : site consults  $C_i$  to determine the fate of  $T$ .
  - if  $C_i$  says  $T$  aborted, site executes **undo** ( $T$ )
  - if  $C_i$  says  $T$  committed, site executes **redo** ( $T$ )
  - if  $C_i$  says  $T$  still in precommit state, site resumes protocol at this point
- Log contains no  $\langle \text{ready } T \rangle$  record for a transaction  $T$ : site executes **undo** ( $T$ ) writes  $\langle \text{abort } T \rangle$  record.

# Coordinator – Failure Protocol

1. The active participating sites select a new coordinator,  $C_{new}$
2.  $C_{new}$  requests local status of  $T$  from each participating site
3. Each participating site including  $C_{new}$  determines the local status of  $T$ :
  - **Committed.** The log contains a **<commit  $T$ >** record
  - **Aborted.** The log contains an **<abort  $T$ >** record.
  - **Ready.** The log contains a **<ready  $T$ >** record but no **<abort  $T$ >** or **<precommit  $T$ >** record
  - **Precommitted.** The log contains a **<precommit  $T$ >** record but no **<abort  $T$ >** or **<commit  $T$ >** record.
  - **Not ready.** The log contains neither a

# Coordinator Failure Protocol

(Cont.)

5.  $C_{new}$  decides either to commit or abort  $T$ , or to restart the three-phase commit protocol:

- Commit state for any one participant  $\Rightarrow$  commit
- Abort state for any one participant  $\Rightarrow$  abort.
- Precommit state for any one participant and above 2 cases do not hold  $\Rightarrow$   
A precommit message is sent to those participants in the uncertain state.  
Protocol is resumed from that point.

# Fully Distributed Deadlock Detection Scheme

- Each site has local wait-for graph; system combines information in these graphs to detect deadlock
- Local Wait-for Graphs

Site 1

Site 2

Site 3

$T_1 \rightarrow T_2$

$T_3 \rightarrow T_4$

$T_5 \rightarrow T_1$



- Global Wait-for Graphs

$$T_1 \rightarrow T_2 \rightarrow T_3 \rightarrow T_4 \rightarrow T_5$$

# Fully Distributed Approach

(Cont.)

- System model: a transaction runs at a single site, and makes requests to other sites for accessing non-local data.
- Each site maintains its own local wait-for graph in the normal fashion: there is an edge  $T_i \rightarrow T_j$  if  $T_i$  is waiting on a lock held by  $T_j$  (note:  $T_i$  and  $T_j$  may be non-local).
- Additionally, arc  $T_i \rightarrow T_{ex}$  exists in the graph at site  $S_k$  if

# Fully Distributed Approach

(Cont.)

- Centralized Deadlock Detection – all graph edges sent to central deadlock detector
- Distributed Deadlock Detection – “path pushing” algorithm
- Path pushing initiated when a site detects a local cycle involving  $T_{ex}$ , which indicates possibility of a deadlock.
- Suppose cycle at site  $S_i$  is

$$T_{ex} \rightarrow T_i \rightarrow T_j \rightarrow \dots \rightarrow T_n \rightarrow T_{ex}$$

# Fully Distributed Approach:

## Example

Site 1

$\text{EX}(3) \rightarrow T_1 \rightarrow T_2 \rightarrow T_3 \rightarrow \text{EX}(2)$

Site 2

$\text{EX}(1) \rightarrow T_3 \rightarrow T_4 \rightarrow T_5 \rightarrow \text{EX}(3)$

Site 3

$\text{EX}(2) \rightarrow T_5 \rightarrow T_1 \rightarrow T_3 \rightarrow \text{EX}(1)$

$\text{EX } (i)$ : Indicates Tex, plus wait is on/by a transaction at Site  $i$

# Fully Distributed Approach

## Example (Cont.)

Site passes wait-for information along path in graph:

- Let  $\text{EX}(j) \rightarrow T_i \rightarrow \dots T_n \rightarrow \text{EX}(k)$  be a path in local wait-for graph at Site  $m$
- Site  $m$  “pushes” the path information to site  $k$  if  $i > n$

Example:

- Site 1 does not pass information :  $1 > 3$
- Site 2 does not pass information :  $3 > 5$
- Site 3 passes  $(T_5, T_1)$  to Site 1 because:
  - $5 > 1$
  - $T_1$  is waiting for a data item at site 1

# Fully Distributed Approach

## (Cont.)

- After the path  $\text{EX}(2) \rightarrow T_5 \rightarrow T_1 \rightarrow \text{EX}(1)$  has been pushed to Site 1 we have:

Site 1

$\text{EX}(2) \rightarrow T_5 \rightarrow T_1 \rightarrow T_2 \rightarrow T_3 \rightarrow \text{EX}(2)$

Site 2

$\text{EX}(1) \rightarrow T_3 \rightarrow T_4 \rightarrow T_5 \rightarrow \text{EX}(3)$

Site 3

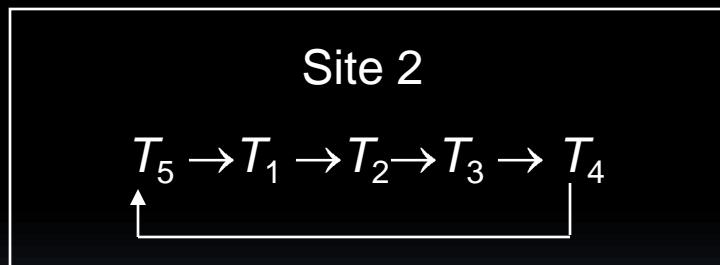
$\text{EX}(2) \rightarrow T_5 \rightarrow T_1 \rightarrow \text{EX}(1)$

# Fully Distributed Approach

- After the push, only Site 1 has new edges. Site 1 passes  $(T_5, T_1, T_2, T_3)$  to site 2 since  $5 > 3$  and  $T_3$  is waiting for a data item, at site 2



- The new state of the local wait-for graph:



Deadlock Detected

Site 3

$\text{EX}(2) \rightarrow T_5 \rightarrow T_1 \rightarrow \text{EX}(1)$

# NAMING OF ITEMS

# Naming of Replicas and Fragments

- Each replica and each fragment of a data item must have a unique name.

- Use of postscripts to determine those replicas that are replicas of the same data item, and those fragments that are fragments of the same data item.
- fragments of same data item: “. $f_1$ ”, “. $f_2$ ”, …, “. $f_n$ ”
- replicas of same data item: “. $r_1$ ”, “. $r_2$ ”, …, “. $r_n$ ”

*sitel7.account.f<sub>3</sub>.r<sub>2</sub>*

refers to replica 2 of fragment 3 of

# Name – Translation Algorithm

if *name* appears in the alias table

    then *expression* := *map* (*name*)

    else *expression* := *name*;

function *map* (*n*)

    if *n* appears in the replica table

        then *result* := name of replica of *n*;

    if *n* appears in the fragment table

        then begin

*result* := expression to construct fragment;

            for each *n'* in *result* do begin

                replace *n'* in *result* with *map* (*n'*);

            end

        end

    return *result*,

# Example of Name - Translation Scheme

- A user at the Hillside branch (site  $S_1$ ), uses the alias *local-account* for the local fragment *account.f1* of the *account* relation.
- When this user references *local-account*, the query-processing subsystem looks up *local-account* in the alias table, and replaces *local-account* with  $S_1.account.f_1$ .
- If  $S_1.account.f_1$  is replicated, the system must consult the replica table in order to choose a replica.

# Transparency and Updates

- Must ensure that all replicas of a data item are updated and that all affected fragments are updated.
- Consider the *account* relation and the insertion of the tuple:  
    (“Valleyview”, A-733, 600)
- Horizontal fragmentation of account
  - $account_1 = \sigma branch-name = “Hillside” (account)$
  - $account_2 = \sigma branch-name = “Valleyview” (account)$

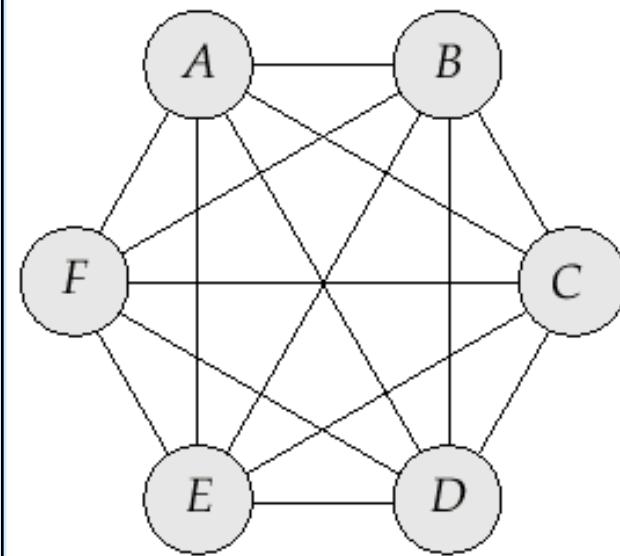
# Transparency and Updates

(Cont.) Vertical fragmentation of *deposit* into *deposit*<sub>1</sub> and *deposit*<sub>2</sub>

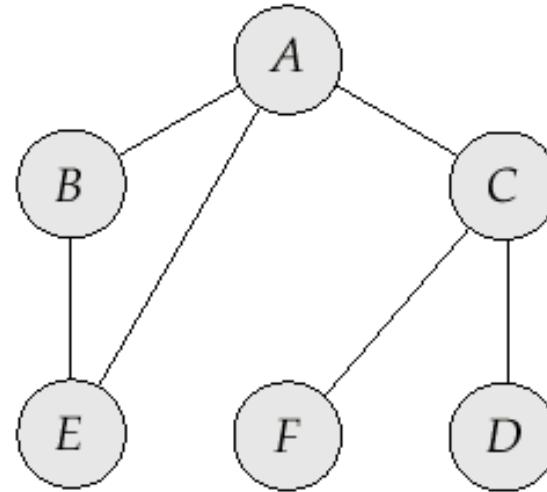
- The tuple (“Valleyview”, A-733, ‘Jones’, 600) must be split into two fragments:
  - one to be inserted into *deposit*<sub>1</sub>
  - one to be inserted into *deposit*<sub>2</sub>
- If *deposit* is replicated, the tuple (“Valleyview”, A-733, “Jones” 600) must be inserted in all replicas
- Problem: If *deposit* is accessed concurrently it is possible that one replica will be updated earlier than

# NETWORK TOPOLOGIES

# Network Topologies

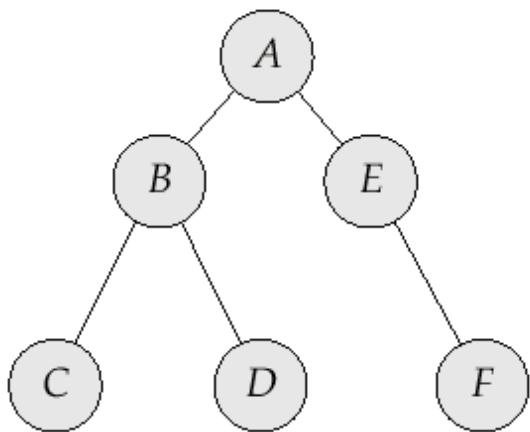


fully connected network

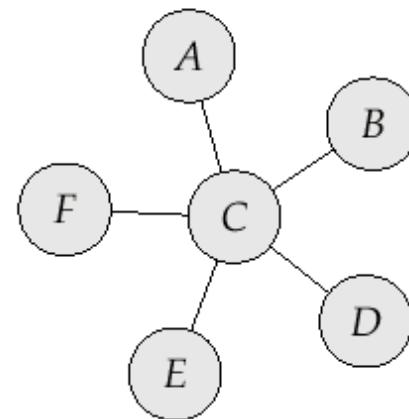


partially connected network

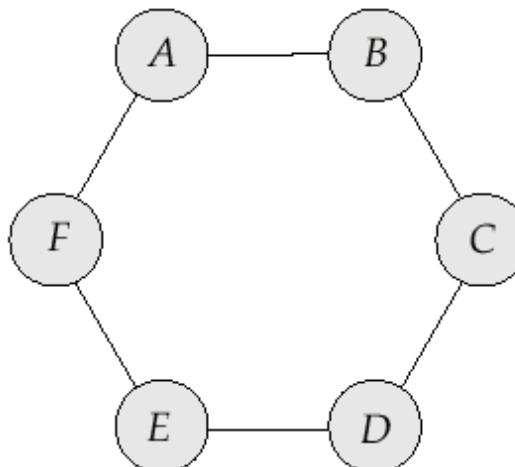
## Network Topologies (Cont.)



tree structured network



star network



ring network

# Network Topology (Cont.)

- A *partitioned* system is split into two (or more) subsystems (*partitions*) that lack any connection.
- Tree-structured: low installation and communication costs; the failure of a single link can partition network
- Ring: At least two links must fail for partition to occur; communication cost is high.
- Star:
  - the failure of a single link results in a partition

# Robustness

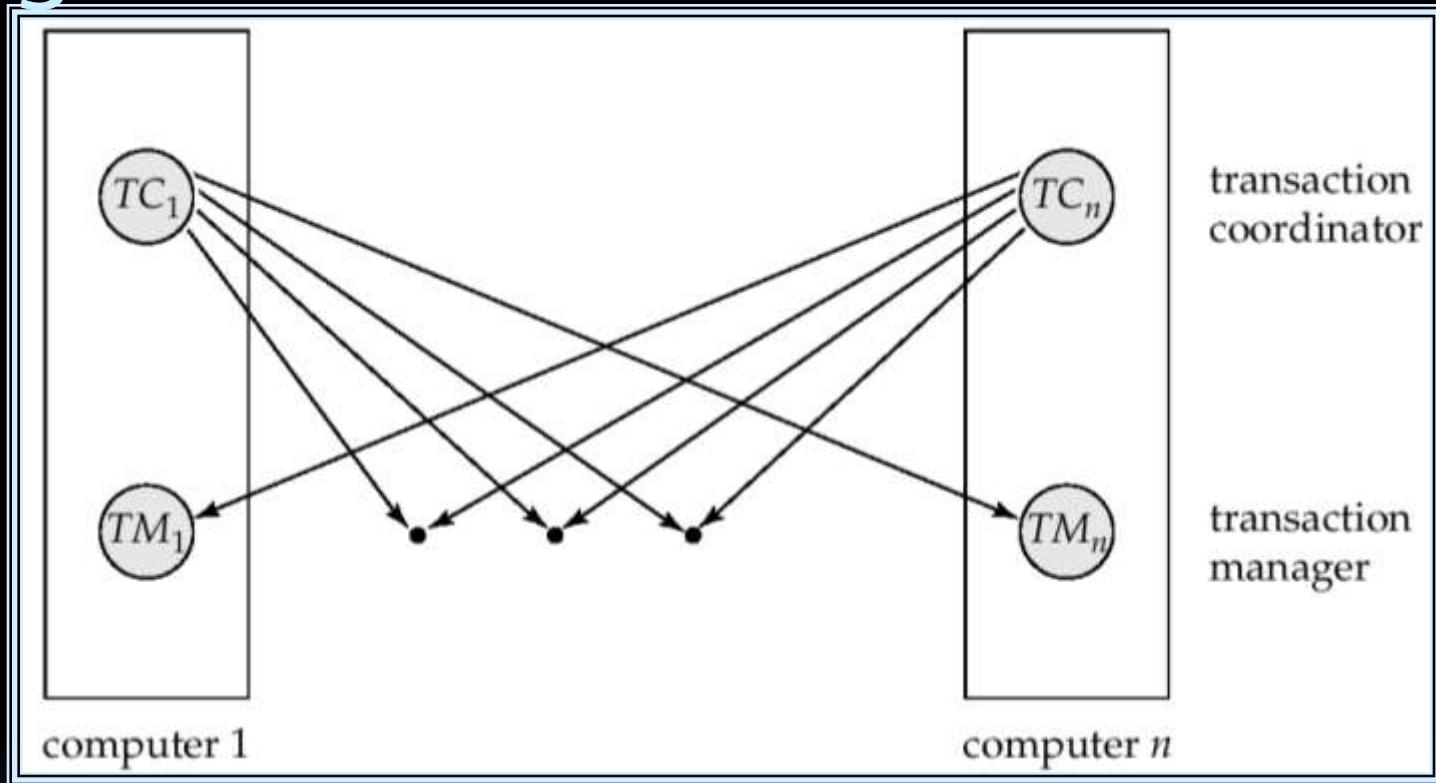
- A robustness system must:
  - Detect site or link failures
  - Reconfigure the system so that computation may continue.
  - Recover when a processor or link is repaired
- Handling failure types:
  - Retransmit lost messages
  - Unacknowledged retransmits indicate link failure; find alternative route for message.
  - Failure to find alternative route is a symptom of network partition.

# Procedure to Reconfigure

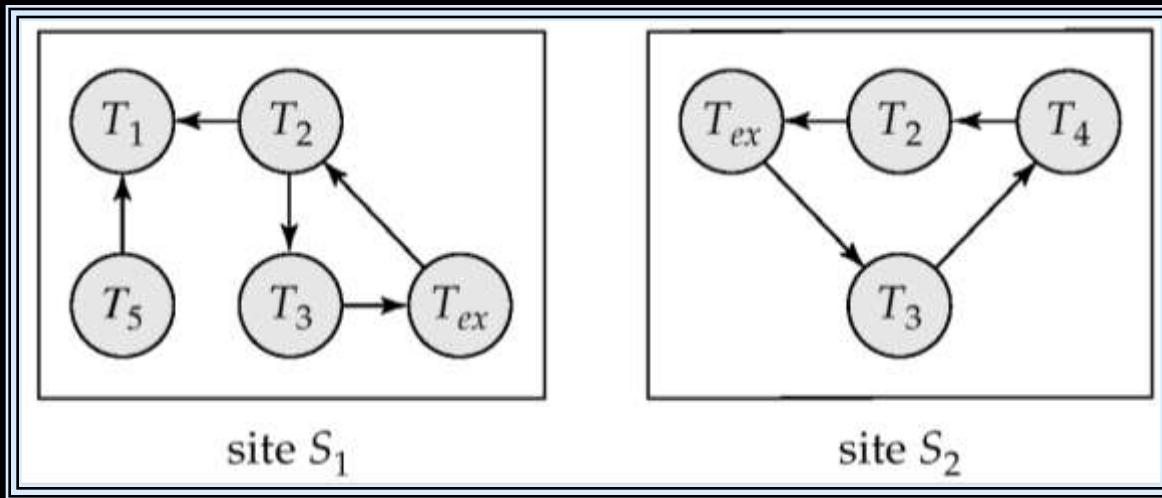
- **System**
  - If replicated data is stored at the failed site, update the catalog so that queries do not reference the copy at the failed site.
  - Transactions active at the failed site should be aborted.
  - If the failed site is a central server for some subsystem, an election must be held to determine the new server.
  - Reconfiguration scheme must work correctly in case of network partitioning.

END OF CHAPTER

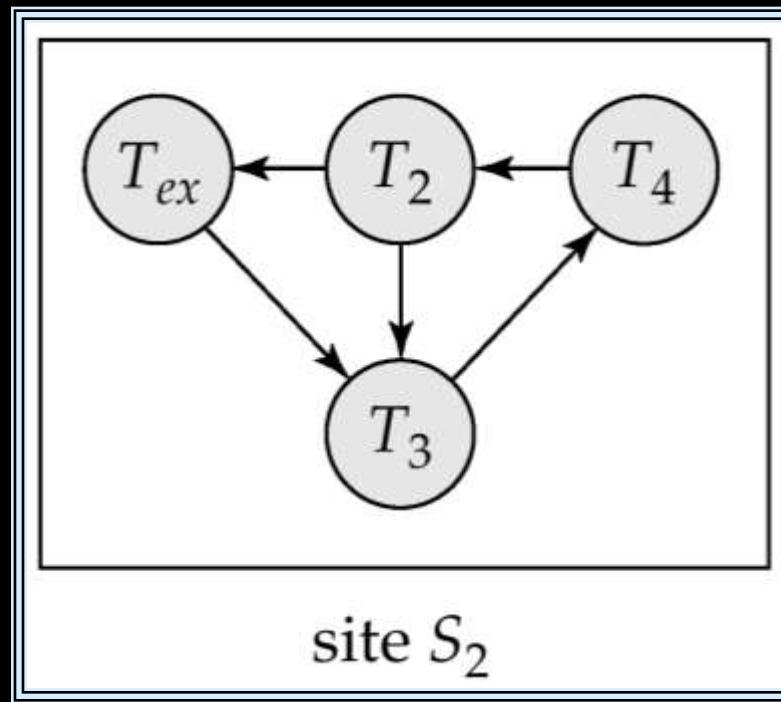
# Figure 19.7



# Figure 19.13



# Figure 19.14



# Chapter 20: Parallel Databases

- Introduction
- I/O Parallelism
- Interquery Parallelism
- Intraquery Parallelism
- Intraoperation Parallelism
- Interoperation Parallelism
- Design of Parallel Systems

# Introduction

- Parallel machines are becoming quite common and affordable
  - Prices of microprocessors, memory and disks have dropped sharply
- Databases are growing increasingly large
  - large volumes of transaction data are collected and stored for later analysis.
  - multimedia objects like images are increasingly stored in databases
- Large-scale parallel database systems increasingly used for:

# Parallelism in Databases

- Data can be partitioned across multiple disks for parallel I/O.
- Individual relational operations (e.g., sort, join, aggregation) can be executed in parallel
  - data can be partitioned and each processor can work independently on its own partition.
- Queries are expressed in high level language (SQL, translated to relational algebra)
  - makes parallelization easier.

# I/O Parallelism

- Reduce the time required to retrieve relations from disk by partitioning the relations on multiple disks.
- Horizontal partitioning – tuples of a relation are divided among many disks such that each tuple resides on one disk.

■ Partitioning techniques (number of disks =  $n$ ):

**Round-robin:**

Send the  $i^{\text{th}}$  tuple inserted in the relation to disk  $i \bmod n$ .

**Hash partitioning:**

# I/O Parallelism (Cont.)

- Partitioning techniques (cont.):
- Range partitioning:
  - Choose an attribute as the partitioning attribute.
  - A partitioning vector  $[v_0, v_1, \dots, v_{n-2}]$  is chosen.
  - Let  $v$  be the partitioning attribute value of a tuple. Tuples such that  $v_i \leq v_{i+1}$  go to disk  $i + 1$ . Tuples with  $v < v_0$  go to disk 0 and tuples with  $v \geq v_{n-2}$  go to disk  $n - 1$ .

E.g., with a partitioning vector [5,11], a tuple with partitioning attribute value of 2 will go to disk 0, a tuple with value 8 will

# Comparison of Partitioning

## Techniques

- Evaluate how well partitioning techniques support the following types of data access:

1. Scanning the entire relation.
2. Locating a tuple associatively - point queries.
  - E.g.,  $r.A = 25$ .
3. Locating all tuples such that the value of a given attribute lies within a specified range - range queries.
  - E.g.,  $10 \leq r.A < 25$ .

# Comparison of Partitioning Techniques (Cont.)

Round robin:

- Advantages
  - Best suited for sequential scan of entire relation on each query.
  - All disks have almost an equal number of tuples; retrieval work is thus well balanced between disks.
- Range queries are difficult to process
  - No clustering -- tuples are scattered across all disks

# Comparison of Partitioning Techniques(Cont.)

## Hash partitioning:

- Good for sequential access
  - Assuming hash function is good, and partitioning attributes form a key, tuples will be equally distributed between disks
  - Retrieval work is then well balanced between disks.
- Good for point queries on partitioning attribute
  - Can lookup single disk, leaving others available for answering other queries.

# Comparison of Partitioning

## Techniques (Cont.)

Range partitioning:

- Provides data clustering by partitioning attribute value.
- Good for sequential access
- Good for point queries on partitioning attribute: only one disk needs to be accessed.
- For range queries on partitioning attribute, one to a few disks may need to be accessed
- Remaining disks are available for other queries.

Good if result tuples are from one to a few partitions.

# Partitioning a Relation across Disks

- If a relation contains only a few tuples which will fit into a single disk block, then assign the relation to a single disk.
- Large relations are preferably partitioned across all the available disks.
- If a relation consists of  $m$  disk blocks and there are  $n$  disks available in the system, then the relation should be allocated  $\min(m, n)$  disks

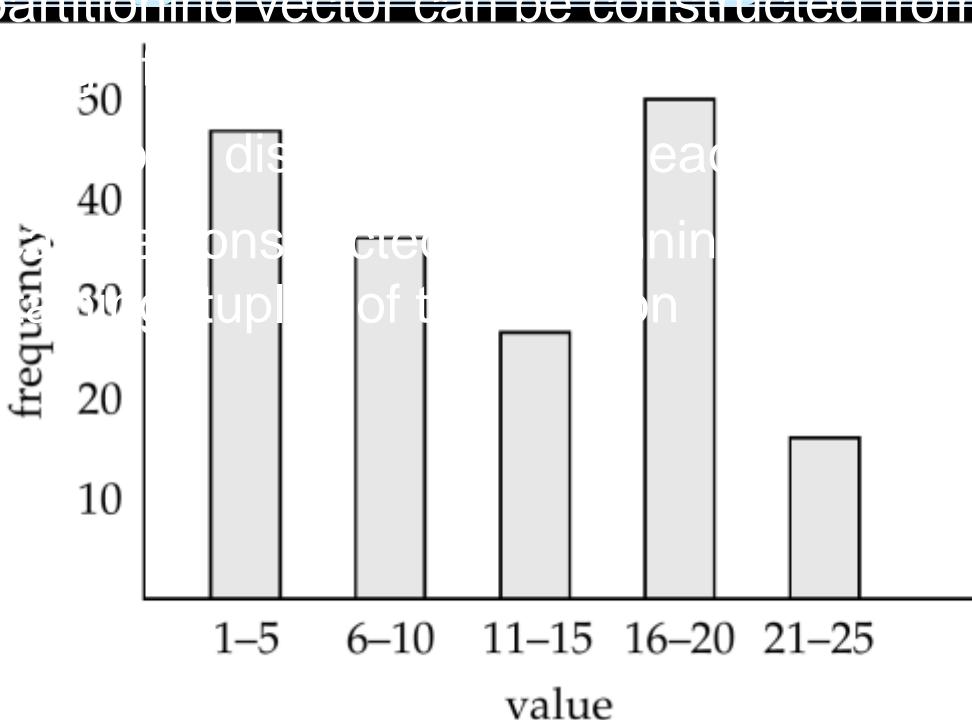
# Handling of Skew

- The distribution of tuples to disks may be skewed — that is, some disks have many tuples, while others may have fewer tuples.
- Types of skew:
  - **Attribute-value skew.**
    - Some values appear in the partitioning attributes of many tuples; all the tuples with the same value for the partitioning attribute end up in the same partition.
    - Can occur with range-partitioning and hash-partitioning.
  - **Partition skew.**
    - With range partitioning, badly chosen

# Handling Skew in Range-Partitioning

- To create a balanced partitioning vector (assuming partitioning attribute forms a key of the relation):
  - Sort the relation on the partitioning attribute.
  - Construct the partition vector by scanning the relation in sorted order as follows.
    - After every  $1/n^{th}$  of the relation has been read, the value of the partitioning attribute of the next tuple is added to the partition vector.
  - $n$  denotes the number of partitions to be constructed.

- Handling Skew using Histograms
  - Balanced partitioning vector can be constructed from histogram in a relatively simple manner
- Assumption: the data is skewed and the distribution is known
- Histograms are used to estimate the distribution of the data for sampling
- Histograms are used to estimate the distribution of the data for sampling (blocks containing)



# Handling Skew Using Virtual Processor Partitioning

- Skew in range partitioning can be handled elegantly using virtual processor partitioning:
  - create a large number of partitions (say 10 to 20 times the number of processors)
  - Assign virtual processors to partitions either in round-robin fashion or based on estimated cost of processing each virtual partition
- Basic idea:
  - If any normal partition would have been skewed, then a virtual partition will be skewed.

## Interquery Parallelism

- Queries/transactions execute in parallel with one another.
- Increases transaction throughput; used primarily to scale up a transaction processing system to support a larger number of transactions per second.
- Easiest form of parallelism to support, particularly in a shared-memory parallel database, because even sequential database systems support concurrent processing.

More complicated to implement on

# Cache Coherency Protocol

- Example of a cache coherency protocol for shared disk systems:
  - Before reading/writing to a page, the page must be locked in shared/exclusive mode.
  - On locking a page, the page must be read from disk
  - Before unlocking a page, the page must be written to disk if it was modified.
- More complex protocols with fewer disk reads/writes exist.
- Cache coherency protocols for shared-nothing systems are similar. Each

# Intraquery Parallelism

- Execution of a single query in parallel on multiple processors/disks; important for speeding up long-running queries.
- Two complementary forms of intraquery parallelism :
  - **Intraoperation Parallelism** – parallelize the execution of each individual operation in the query.
  - **Interoperation Parallelism** – execute the different operations in a query expression in parallel.

# Parallel Processing of Relational Operations

- Our discussion of parallel algorithms assumes:
  - *read-only* queries
  - shared-nothing architecture
  - $n$  processors,  $P_0, \dots, P_{n-1}$ , and  $n$  disks  $D_0, \dots, D_{n-1}$ , where disk  $D_i$  is associated with processor  $P_i$ .
- If a processor has multiple disks they can simply simulate a single disk  $D_i$ .
- Shared-nothing architectures can be efficiently simulated on shared-memory and shared-disk systems.

# Parallel Sort

## Range-Partitioning Sort

- Choose processors  $P_0, \dots, P_m$ , where  $m \leq n - 1$  to do sorting.
- Create range-partition vector with  $m$  entries, on the sorting attributes
- Redistribute the relation using range partitioning
  - all tuples that lie in the  $i^{\text{th}}$  range are sent to processor  $P_i$
  - $P_i$  stores the tuples it received temporarily on disk  $D_i$ .
  - This step requires I/O and communication overhead.
- Each processor  $P_i$  sorts its partition of the relation locally.
- Each processor executes same operation (sort) in parallel with other processors, without any interaction with the others (data parallelism).
- Final merge operation is trivial: range-partitioning ensures that,

# Parallel Sort (Cont.)

## Parallel External Sort-Merge

- Assume the relation has already been partitioned among disks  $D_0, \dots, D_{n-1}$  (in whatever manner).
- Each processor  $P_i$  locally sorts the data on disk  $D_i$ .
- The sorted runs on each processor are then merged to get the final sorted output.
- Parallelize the merging of sorted runs as follows:

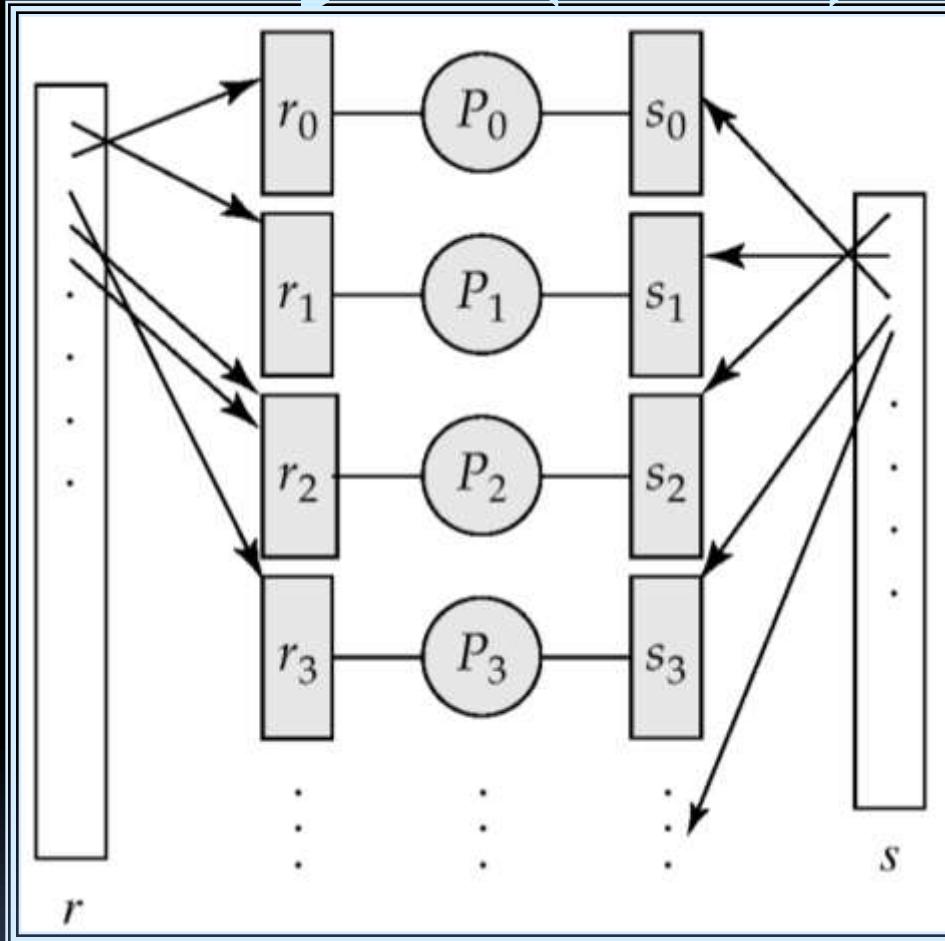
# Parallel Join

- The join operation requires pairs of tuples to be tested to see if they satisfy the join condition, and if they do, the pair is added to the join output.
- Parallel join algorithms attempt to split the pairs to be tested over several processors. Each processor then computes part of the join locally.
- In a final step, the results from each processor can be collected together to produce the final result

# Partitioned Join

- For equi-joins and natural joins, it is possible to *partition* the two input relations across the processors, and  $\bowtie$  compute the join locally at each processor.
- Let  $r$  and  $s$  be the input relations, and we want to compute  $r \underset{r.A=s.B}{\bowtie} S$ .
- $r$  and  $s$  each are partitioned into  $n$  partitions, denoted  $r_0, r_1, \dots, r_{n-1}$  and  $s_0, s_1, \dots, s_{n-1}$ .
- Can use either *range partitioning* or *hash partitioning*.
- $r$  and  $s$  must be partitioned on their join

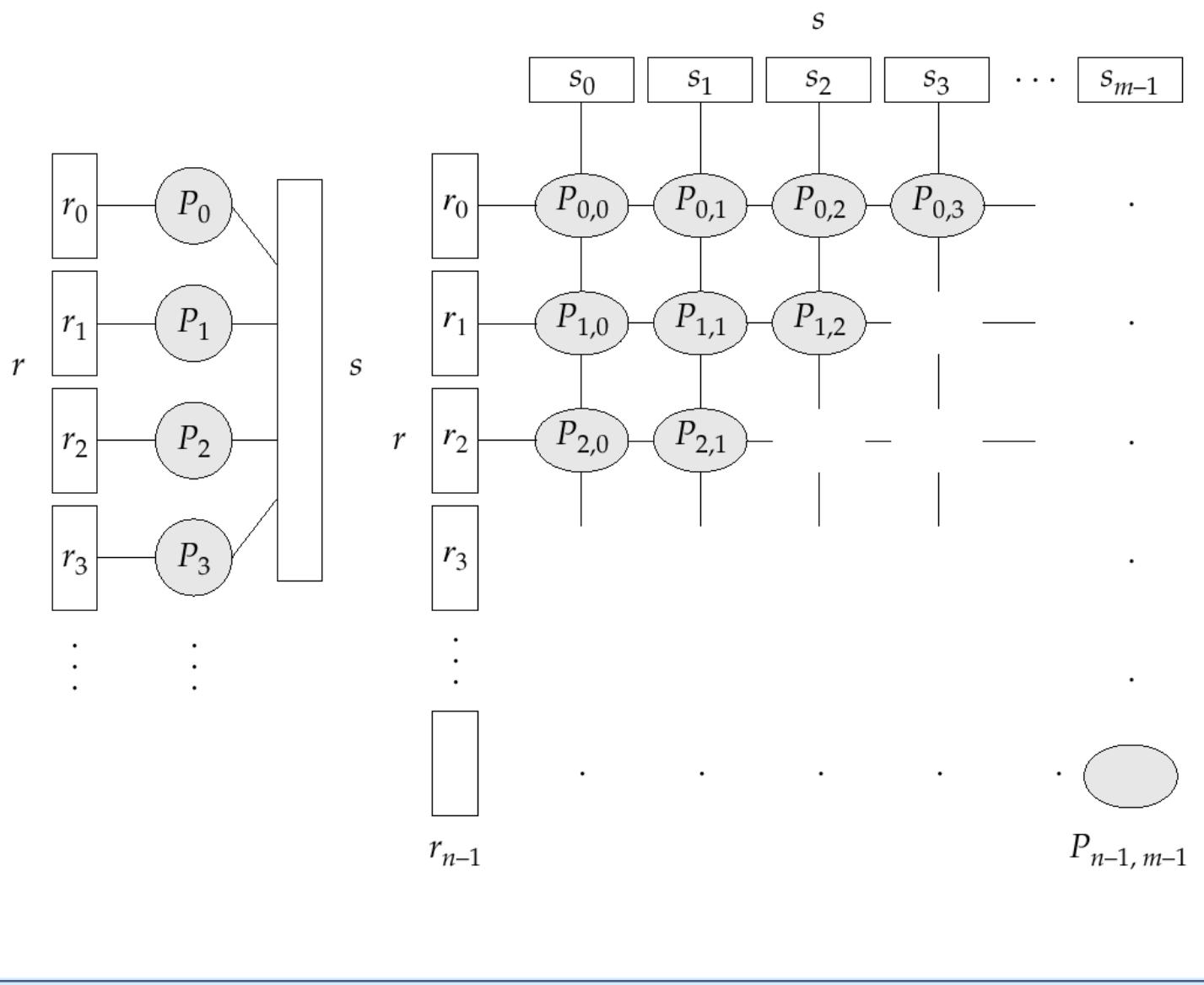
# Partitioned Join (Cont.)



# Fragment-and-Replicate Join

- Partitioning not possible for some join conditions
  - e.g., non-equijoin conditions, such as  $r.A > s.B$ .
- For joins where partitioning is not applicable, parallelization can be accomplished by fragment and replicate technique
  - Depicted on next slide
- Special case – asymmetric fragment-and-replicate:
  - One of the relations, say  $r$ , is partitioned; any partitioning technique can be used.

# Depiction of Fragment-and-Replicate links



# Fragment-and-Replicate Join

(Cont.)

General case: reduces the sizes of the relations at each processor.

- $r$  is partitioned into  $n$  partitions,  $r_0, r_1, \dots, r_{n-1}$ ;  $s$  is partitioned into  $m$  partitions,  $s_0, s_1, \dots, s_{m-1}$ .
- Any partitioning technique may be used.
- There must be at least  $m * n$  processors.
- Label the processors as
- $P_{0,0}, P_{0,1}, \dots, P_{0,m-1}, P_{1,0}, \dots, P_{n-1,m-1}$ .
- $P_{i,j}$  computes the join of  $r_i$  with  $s_j$ . In order to do so,  $r_i$  is replicated to  $P_{i,0}, P_{i,1}, \dots, P_{i,m-1}$ , while  $s_j$  is replicated to  $P_{0,j}, P_{1,j}, \dots, P_{n-1,j}$ .

# Fragment-and-Replicate Join

(Cont.)

- Both versions of fragment-and-replicate work with any join condition, since every tuple in  $r$  can be tested with every tuple in  $s$ .
- Usually has a higher cost than partitioning, since one of the relations (for asymmetric fragment-and-replicate) or both relations (for general fragment-and-replicate) have to be replicated.
- Sometimes asymmetric fragment-and-replicate is preferable even though partitioning could be used.

# Partitioned Parallel Hash-Join

Parallelizing partitioned hash join:

- Assume  $s$  is smaller than  $r$  and therefore  $s$  is chosen as the build relation.
- A hash function  $h_1$  takes the join attribute value of each tuple in  $s$  and maps this tuple to one of the  $n$  processors.
- Each processor  $P_i$  reads the tuples of  $s$  that are on its disk  $D_i$ , and sends each tuple to the appropriate processor based on hash function  $h_2$ . Let  $s$

# Partitioned Parallel Hash-Join

(Cont.)

Once the tuples of  $s$  have been distributed, the larger relation  $r$  is redistributed across the  $m$  processors using the hash function  $h_1$

- Let  $r_i$  denote the tuples of relation  $r$  that are sent to processor  $P_i$ .

As the  $r$  tuples are received at the destination processors, they are repartitioned using the function  $h_2$

- (just as the probe relation is partitioned in the sequential hash-join algorithm).

Each processor  $P_i$  executes the build

# Parallel Nested-Loop Join

- Assume that
  - relation  $s$  is much smaller than relation  $r$  and that  $r$  is stored by partitioning.
  - there is an index on a join attribute of relation  $r$  at each of the partitions of relation  $r$ .
- Use asymmetric fragment-and-replicate, with relation  $s$  being replicated, and using the existing partitioning of relation  $r$ .
- Each processor  $P_j$  where a partition of relation  $s$  is stored reads the tuples of

# Other Relational Operations

## Selection $\sigma_\theta(r)$

- If  $\theta$  is of the form  $a_i = v$ , where  $a_i$  is an attribute and  $v$  a value.
  - If  $r$  is partitioned on  $a_i$  the selection is performed at a single processor.
- If  $\theta$  is of the form  $l \leq a_i \leq u$  (i.e.,  $\theta$  is a range selection) and the relation has been range-partitioned on  $a_i$ 
  - Selection is performed at each processor whose partition overlaps with the specified range of values.

# Other Relational Operations (Cont.)

- Duplicate elimination
  - Perform by using either of the parallel sort techniques
    - eliminate duplicates as soon as they are found during sorting.
  - Can also partition the tuples (using either range- or hash- partitioning) and perform duplicate elimination locally at each processor.
- Projection
  - Projection without duplicate elimination

# Grouping/Aggregation

- Partition the relation on the grouping attributes and then compute the aggregate values locally at each processor.
- Can reduce cost of transferring tuples during partitioning by partly computing aggregate values before partitioning.
- Consider the sum aggregation operation:
  - Perform aggregation operation at each

# Cost of Parallel Evaluation of Operations

- If there is no skew in the partitioning, and there is no overhead due to the parallel evaluation, expected speed-up will be  $1/n$
- If skew and overheads are also to be taken into account, the time taken by a parallel operation can be estimated as

$$T_{\text{part}} + T_{\text{asm}} + \max(T_0, T_1, \dots, T_{n-1})$$

- $T_{\text{part}}$  is the time for partitioning the relations

# Interoperator Parallelism

- Pipelined parallelism
  - Consider a join of four relations
    - $r_1 \quad r_2 \quad r_3 \quad r_4$
  - Set up a pipeline that computes the three joins in parallel
    - Let P1 be assigned the computation of $\text{temp1} = r_1 \quad r_2$
    - And P2 be assigned the computation of $\text{temp2} = \text{temp1} \quad r_3$
    - And P3 be assigned the computation of $\text{temp2} \quad r_4$
  - Each of these operations can execute in

# Factors Limiting Utility of Pipeline Parallelism

- Pipeline parallelism is useful since it avoids writing intermediate results to disk
- Useful with small number of processors, but does not scale up well with more processors. One reason is that pipeline chains do not attain sufficient length.
- Cannot pipeline operators which do not produce output until all inputs have been accessed (e.g. aggregate and sort)

# Independent Parallelism

## Independent parallelism

- Consider a join of four relations

$$r_1 \quad r_2 \quad r_3 \bowtie r_4$$

- Let P1 be assigned the computation of  
 $\text{temp1} = r_1 \quad r_2 \quad \bowtie$
- And P2 be assigned the computation of  $\text{temp2} = r_3 \quad r_4$
- And P3 be assigned the computation of  $\text{temp1} \bowtie \text{temp2}$
- P1 and P2 can work **independently in parallel**
- P3 has to wait for input from P1 and P2
  - Can pipeline output of P1 and P2 to P3, combining independent parallelism and pipelined parallelism
- Does not provide a high degree of parallelism
  - Useful with a lower degree of parallelism

# Query Optimization

Query optimization in parallel databases is significantly more complex than query optimization in sequential databases.

Cost models are more complicated, since we must take into account partitioning costs and issues such as skew and resource contention.

When scheduling execution tree in parallel system, must decide:

- How to parallelize each operation and how many processors to use for it.
- What operations to pipeline, what operations to execute independently in parallel, and what operations to execute sequentially, one after the other.

Determining the amount of resources to allocate for each operation is a problem.

- E.g., allocating more processors than optimal can result in high communication overhead.

Long pipelines should be avoided as the final operation may wait a lot for inputs, while holding precious resources

# Query Optimization (Cont.)

- The number of parallel evaluation plans from which to choose from is much larger than the number of sequential evaluation plans.
  - Therefore heuristics are needed while optimization
- Two alternative heuristics for choosing parallel plans:
  - No pipelining and inter-operation pipelining; just parallelize every operation across all processors.
    - Finding best plan is now much easier --- use standard optimization technique, but with new cost model
    - Volcano parallel database popularize the **exchange-operator** model
      - exchange operator is introduced into query plans to partition and distribute tuples
      - each operation works independently on local

# Design of Parallel Systems

Some issues in the design of parallel systems:

- Parallel loading of data from external sources is needed in order to handle large volumes of incoming data.
- Resilience to failure of some processors or disks.
  - Probability of some disk or processor failing is higher in a parallel system.
  - Operation (perhaps with degraded performance) should be possible in spite

# Design of Parallel Systems (Cont.)

- On-line reorganization of data and schema changes must be supported.
  - For example, index construction on terabyte databases can take hours or days even on a parallel system.
  - Need to allow other processing (insertions/deletions/updates) to be performed on relation even as index is being constructed.
  - Basic idea: index construction tracks changes and ``catches up'' on changes at the end.

END OF CHAPTER





# CHAPTER 21: APPLICATION DEVELOPMENT AND ADMINISTRATION

# Overview

- Web Interfaces to Databases
- Performance Tuning
- Performance Benchmarks
- Standardization
- E-Commerce
- Legacy Systems

# The World Wide Web

- The Web is a distributed information system based on hypertext.
- Most Web documents are hypertext documents formatted via the HyperText Markup Language (HTML)
- HTML documents contain
  - text along with font specifications, and other formatting instructions
  - hypertext links to other documents, which can be associated with regions of the text.
  - forms, enabling users to enter data which can then be sent back to the Web server

# Web Interfaces to Databases

Why interface databases to the Web?

1. Web browsers have become the de-facto standard user interface to databases

- Enable large numbers of users to access databases from anywhere
- Avoid the need for downloading/installing specialized code, while providing a good graphical user interface
- E.g.: Banks, Airline/Car reservations, University course registration/grading, ...

# Web Interfaces to Database (Cont.)

## 2. Dynamic generation of documents

- Limitations of static HTML documents
  - Cannot customize fixed Web documents for individual users.
  - Problematic to update Web documents, especially if multiple Web documents replicate data.
- Solution: Generate Web documents dynamically from data stored in a database.
  - Can tailor the display based on user information stored in the database.
    - E.g. tailored ads, tailored weather and local news, ...
  - Displayed information is up-to-date, unlike the static Web pages

# Uniform Resources Locators

- In the Web, functionality of pointers is provided by Uniform Resource Locators (URLs).
- URL example:

<http://www.bell-labs.com/topics/book/db-book>

- The first part indicates how the document is to be accessed
  - “http” indicates that the document is to be accessed using the Hyper Text Transfer Protocol.
- The second part gives the unique name of a machine on the Internet.
- The rest of the URL identifies the document within the machine.

# HTML and HTTP

- HTML provides formatting, hypertext link, and image display features.
- HTML also provides input features
  - Select from a set of options
    - Pop-up menus, radio buttons, check lists
  - Enter values
    - Text boxes
  - Filled in input sent back to the server, to be acted upon by an executable at the server
- HyperText Transfer Protocol (HTTP) used for communication with the Web server

# Sample HTML Source Text

```
<html> <body>
<table border cols = 3>
 <tr> <td> A-101 </td> <td>
Downtown </td> <td> 500 </td> </tr>
...
</table>
<center> The <i>account</i> relation
</center>
```

```
<form action="BankQuery"
method=get>
 Select account/loan and enter number

 <select name="type">
```

# Display of Sample HTML Source

A-101	Downtown	500
A-102	Perryridge	400
A-201	Brighton	900

The *account* relation

Select account/loan and enter number

Account

# Client Side Scripting and

- ~~Applets~~ Browsers can fetch certain scripts (client-side scripts) or programs along with documents, and execute them in “safe mode” at the client site
  - Javascript
  - Macromedia Flash and Shockwave for animation/games
  - VRML
  - Applets
- Client-side scripts/programs allow documents to be active
  - E.g., animation by executing programs at the client site

# Client Side Scripting and

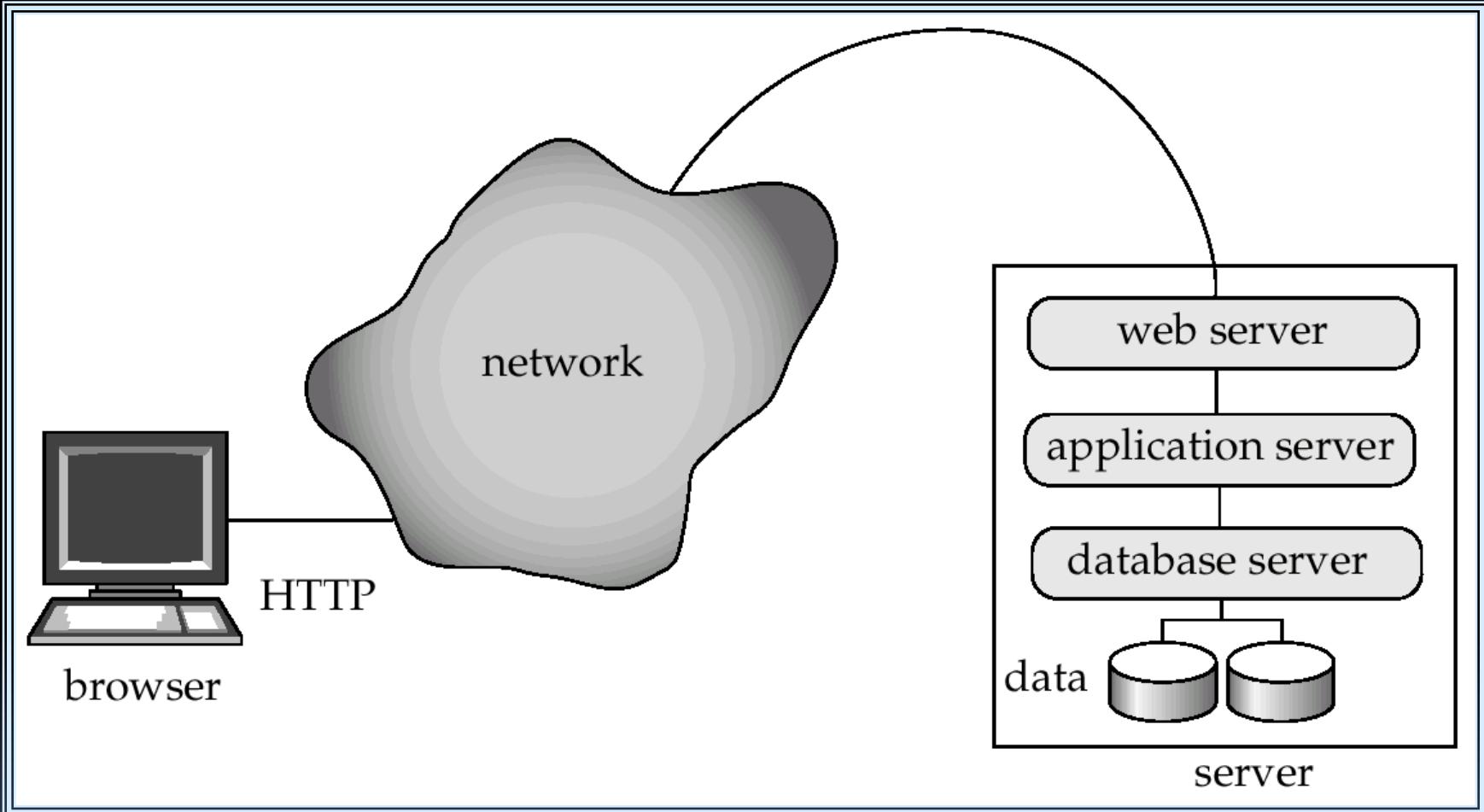
## Security

- Security mechanisms needed to ensure that malicious scripts do not cause damage to the client machine
  - Easy for limited capability scripting languages, harder for general purpose programming languages like Java
- E.g. Java's security system ensures that the Java applet code does not make any system calls directly
  - Disallows dangerous actions such as file writes
  - Notifies the user about potentially dangerous actions, and allows the option

# Web Servers

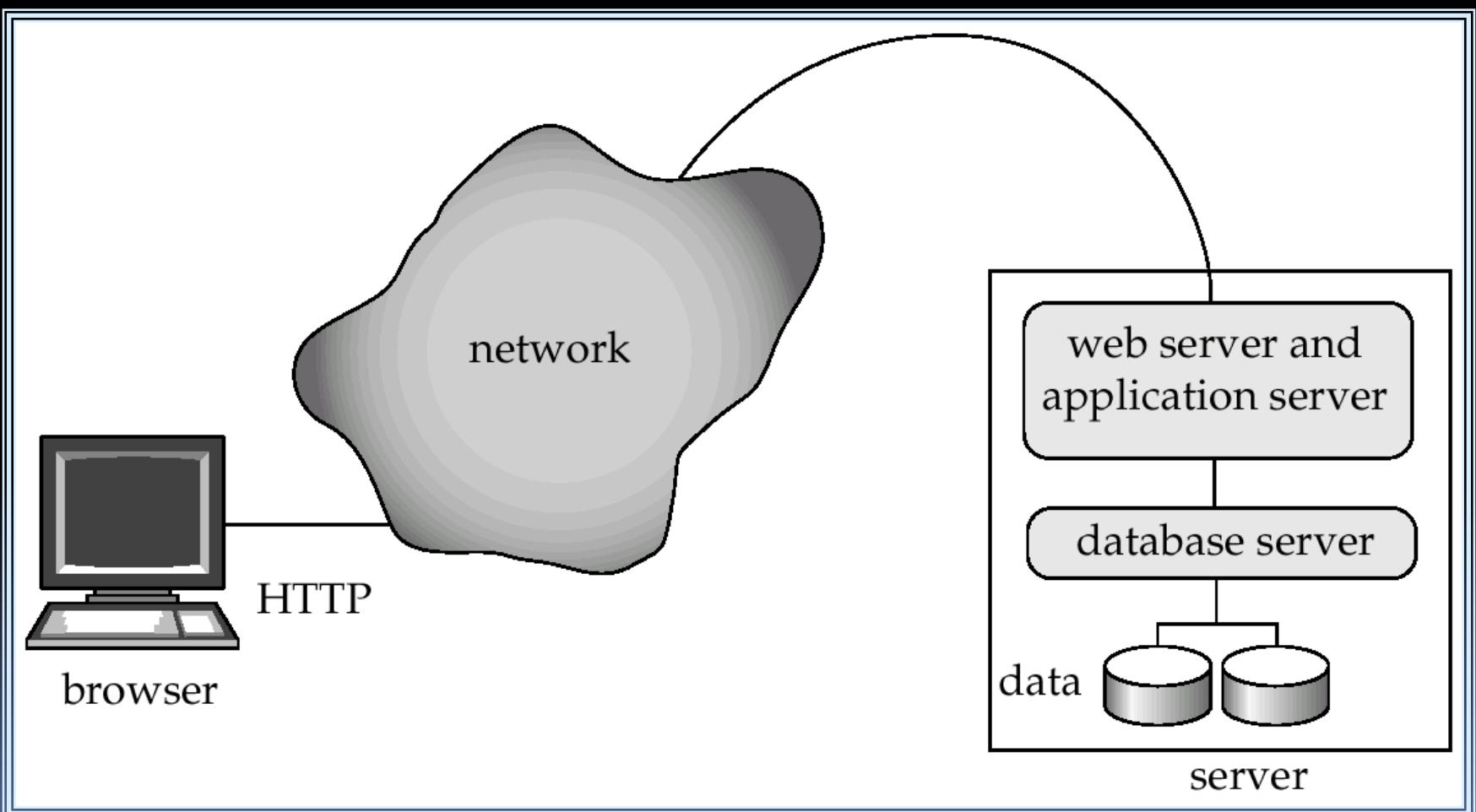
- A Web server can easily serve as a front end to a variety of information services.
- The document name in a URL may identify an executable program, that, when run, generates a HTML document.
  - When a HTTP server receives a request for such a document, it executes the program, and sends back the HTML document that is generated.
  - The Web client can pass extra arguments with the name of the document.

# Three-Tier Web Architecture



# Two-Tier Web Architecture

- Multiple levels of indirection have overheads
  - ☞ Alternative: two-tier architecture



# HTTP and Sessions

- The HTTP protocol is connectionless
  - That is, once the server replies to a request, the server closes the connection with the client, and forgets all about the request
  - In contrast, Unix logins, and JDBC/ODBC connections stay connected until the client disconnects
    - retaining user authentication and other information
  - Motivation: reduces load on server
    - operating systems have tight limits on

# Sessions and Cookies

- A cookie is a small piece of text containing identifying information
  - Sent by server to browser on first interaction
  - Sent by browser to the server that created the cookie on further interactions
    - part of the HTTP protocol
  - Server saves information about cookies it issued, and can use it when serving a request
    - E.g., authentication information, and user preferences

# Servlets

- Java Servlet specification defines an API for communication between the Web server and application program
  - E.g. methods to get parameter values and to send HTML text back to client
- Application program (also called a servlet) is loaded into the Web server
  - Two-tier model
  - Each request spawns a new thread in the Web server
    - thread is closed once the request is serviced
- Servlet API provides a getSession()

# Example Servlet Code

```
Public class BankQuery(Servlet extends
HttpServlet {
public void doGet(HttpServletRequest
request, HttpServletResponse result)
throws ServletException, IOException {
String type =
request.getParameter("type");
String number =
request.getParameter("number");
...code to find the loan
amount/account balance ...
...using JDBC to communicate with the
database..
...we assume the value is stored in the
```

# Server-Side Scripting

- Server-side scripting simplifies the task of connecting a database to the Web
  - Define a HTML document with embedded executable code/SQL queries.
  - Input values from HTML forms can be used directly in the embedded code/SQL queries.
  - When the document is requested, the Web server executes the embedded code/SQL queries to generate the actual HTML document.

# Improving Web Server Performance

- Performance is an issue for popular Web sites
  - May be accessed by millions of users every day, thousands of requests per second at peak time
- Caching techniques used to reduce cost of serving pages by exploiting commonalities between requests
  - At the server site:
    - Caching of JDBC connections between servlet requests

# PERFORMANCE TUNING

# Performance Tuning

- Adjusting various parameters and design choices to improve system performance for a specific application.
- Tuning is best done by
  1. identifying bottlenecks, and
  2. eliminating them.
- Can tune a database system at 3 levels:
  - **Hardware** -- e.g., add disks to speed up I/O, add memory to increase buffer hits, move to a faster processor.
  - **Database system parameters** -- e.g., set buffer size to avoid paging of buffer, set

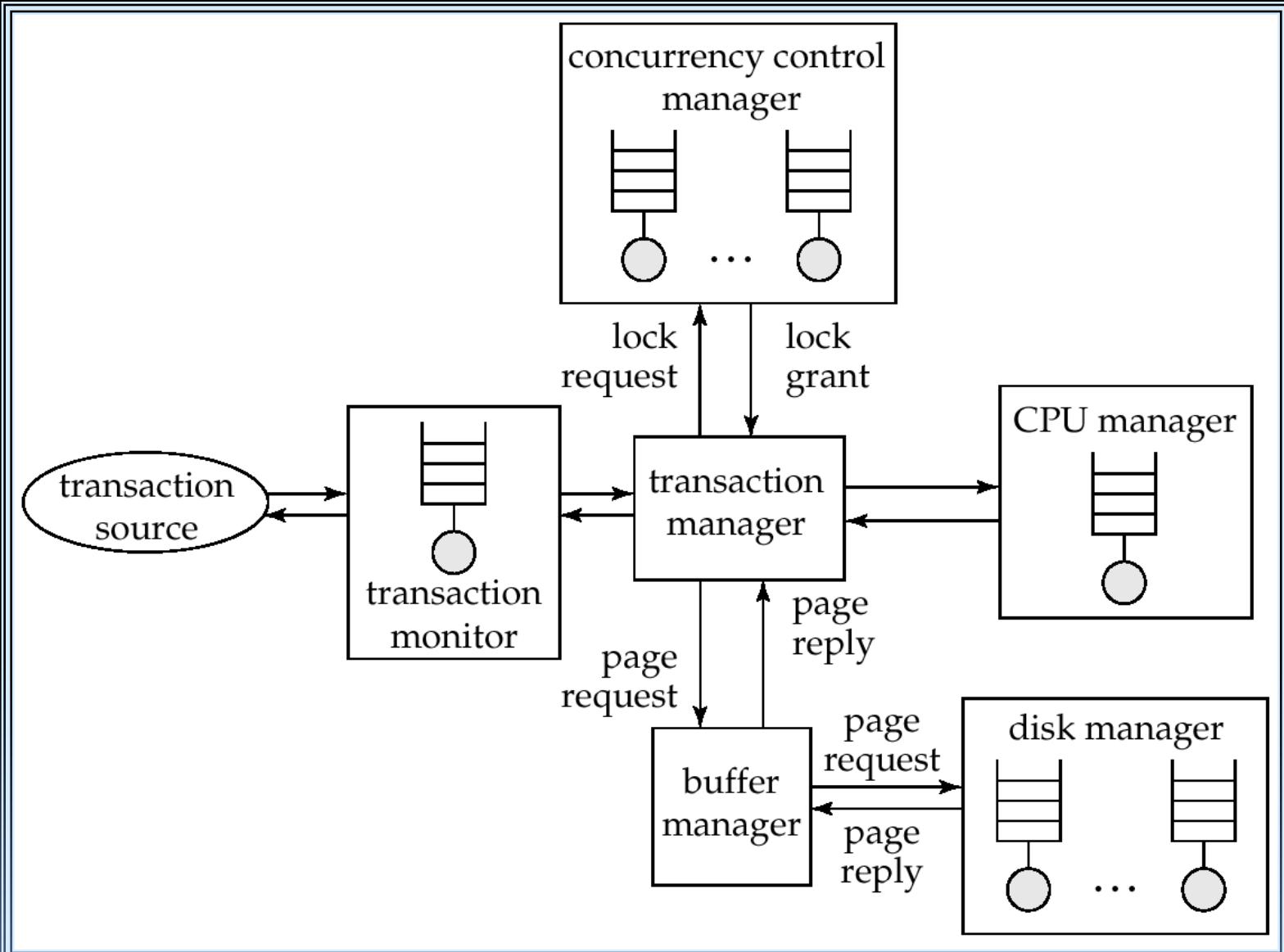
# Bottlenecks

- Performance of most systems (at least before they are tuned) usually limited by performance of one or a few components: these are called **bottlenecks**
  - E.g. 80% of the code may take up 20% of time and 20% of code takes up 80% of time
    - Worth spending most time on 20% of code that take 80% of time
- Bottlenecks may be in hardware (e.g. disk access, memory, CPU bottleneck), or in

# Identifying Bottlenecks

- Transactions request a sequence of services
  - e.g. CPU, Disk I/O, locks
- With concurrent transactions, transactions may have to wait for a requested service while other transactions are being served
- Can model database as a **queueing system** with a queue for each service
  - transactions repeatedly do the following
    - request a service, wait in queue for the service, and get serviced
- Bottlenecks in a database system typically show up as very high

# Queues In A Database System



# Tunable Parameters

- Tuning of hardware
- Tuning of schema
- Tuning of indices
- Tuning of materialized views
- Tuning of transactions

# Tuning of Hardware

- Even well-tuned transactions typically require a few I/O operations
  - Typical disk supports about 100 random I/O operations per second
  - Suppose each transaction requires just 2 random I/O operations. Then to support  $n$  transactions per second, we need to stripe data across  $n/50$  disks (ignoring skew)
- Number of I/O operations per transaction can be reduced by keeping more data in memory

# Hardware Tuning: Five-Minute

- Question: which data to keep in memory:

## Rule

- If a page is accessed  $n$  times per second, keeping it in memory saves
  - $n * \frac{\text{price-per-disk-drive}}{\text{accesses-per-second-per-disk}}$
- Cost of keeping page in memory
  - $\text{price-per-MB-of-memory} * \text{ages-per-MB-of-memory}$
- Break-even point: value of  $n$  for which above costs are equal
  - If accesses are more then saving is greater than cost
  - Solving above equation with current disk and memory prices leads to...

# Hardware Tuning: One-Minute Rule

- For sequentially accessed data, more pages can be read per second.  
Assuming sequential reads of 1 MB of data at a time:  
**1-minute rule:** sequentially accessed data that is accessed once or more in a minute should be kept in memory
- Prices of disk and memory have changed greatly over the years, but the ratios have not changed much

# Hardware Tuning: Choice of RAID Level

- To use RAID 1 or RAID 5?
  - Depends on ratio of reads and writes
    - RAID 5 requires 2 block reads and 2 block writes to write out one data block
- If an application requires  $r$  reads and  $w$  writes per second
  - RAID 1 requires  $r + 2w$  I/O operations per second
  - RAID 5 requires:  $r + 4w$  I/O operations per second
- For reasonably large  $r$  and  $w$ , this requires lots of disks to handle workload
  - RAID 5 may require more disks than RAID

# Tuning the Database Design

## ▪ Schema tuning

- Vertically partition relations to isolate the data that is accessed most often -- only fetch needed information.
  - E.g., split *account* into two, (*account-number*, *branch-name*) and (*account-number*, *balance*).
    - Branch-name need not be fetched unless required
- Improve performance by storing a **denormalized relation**
  - E.g., store join of *account* and *depositor*, branch-name and balance information is repeated for each holder of an account, but join need not be computed repeatedly.
    - Price paid: more space and more work for

# Tuning the Database Design (Cont.)

## Index tuning

- Create appropriate indices to speed up slow queries/updates
- Speed up slow updates by removing excess indices (tradeoff between queries and updates)
- Choose type of index (B-tree/hash) appropriate for most frequent types of queries.
- Choose which index to make clustered
- Index tuning wizards look at past history of queries and updates (the **workload**)  
and recommend which indices would be

# Tuning the Database Design

## Materialized Views (Cont.)

- Materialized views can help speed up certain queries
  - Particularly aggregate queries
- Overheads
  - Space
  - Time for view maintenance
    - Immediate view maintenance: done as part of update txn
      - time overhead paid by update transaction
    - Deferred view maintenance: done only when required
      - update transaction is not affected, but system time is spent on view maintenance

# Tuning the Database Design (Cont.)

- How to choose set of materialized views
  - Helping one transaction type by introducing a materialized view may hurt others
  - Choice of materialized views depends on costs
    - Users often have no idea of actual cost of operations
  - Overall, manual selection of materialized views is tedious
- Some database systems provide tools

# Tuning of Transactions

- Basic approaches to tuning of transactions
  - Improve set orientation
  - Reduce lock contention
- Rewriting of queries to improve performance was important in the past, but smart optimizers have made this less important
- Communication overhead and query handling overheads significant part of cost of each call
  - Combine multiple embedded SQL/ODBC/JDBC queries into a single set-oriented query

# Tuning of Transactions (Cont.)

- Reducing lock contention
- Long transactions (typically read-only) that examine large parts of a relation result in lock contention with update transactions
  - E.g. large query to compute bank statistics and regular bank transactions
- To reduce contention
  - Use multi-version concurrency control
    - E.g. Oracle “snapshots” which support multi-version 2PL

# Tuning of Transactions (Cont.)

- Long update transactions cause several problems
  - Exhaust lock space
  - Exhaust log space
    - and also greatly increase recovery time after a crash, and may even exhaust log space during recovery if recovery algorithm is badly designed!
- Use **mini-batch** transactions to limit number of updates that a single transaction can carry out. E.g., if a single large transaction updates every record of a very large relation, log may grow too big.

# Performance Simulation

- Performance simulation using queuing model useful to predict bottlenecks as well as the effects of tuning changes, even without access to real system
- Queuing model as we saw earlier
  - Models activities that go on in parallel
- Simulation model is quite detailed, but usually omits some low level details
  - Model service time, but disregard details of service
  - E.g. approximate disk read time by using an average disk read time

# PERFORMANCE BENCHMARKS

# Performance Benchmarks

- Suites of tasks used to quantify the performance of software systems
- Important in comparing database systems, especially as systems become more standards compliant.
- Commonly used performance measures:
  - **Throughput** (transactions per second, or tps)
  - **Response time** (delay from submission of transaction to return of result)
  - **Availability** or mean time to failure

# Performance Benchmarks

- Suites of tasks used to characterize performance
  - single task not enough for complex systems
- Beware when computing average throughput of different transaction types
  - E.g., suppose a system runs transaction type A at 99 tps and transaction type B at 1 tps.
  - Given an equal mixture of types A and B, throughput is not  $(99+1)/2 = 50$  tps.
  - Running one transaction of each type takes time  $1/.01$  seconds, giving a throughput of 1.98 tps.
  - To compute average throughput, use

# Database Application Classes

- **Online transaction processing (OLTP)**
  - requires high concurrency and clever techniques to speed up commit processing, to support a high rate of update transactions.
- **Decision support applications**
  - including **online analytical processing**, or **OLAP** applications
  - require good query evaluation algorithms and query optimization.
- Architecture of some database

# Benchmarks Suites

- The Transaction Processing Council (TPC) benchmark suites are widely used.
  - TPC-A and TPC-B: simple OLTP application modeling a bank teller application with and without communication
    - Not used anymore
  - TPC-C: complex OLTP application modeling an inventory system
    - Current standard for OLTP benchmarking

# Benchmarks Suites (Cont.)

- TPC benchmarks (cont.)
  - TPC-D: complex decision support application
    - Superceded by TPC-H and TPC-R
  - TPC-H: (H for ad hoc) based on TPC-D with some extra queries
    - Models ad hoc queries which are not known beforehand
      - Total of 22 queries with emphasis on aggregation
    - prohibits materialized views
    - permits indices only on primary and foreign

# TPC Performance Measures

- TPC performance measures
  - **transactions-per-second** with specified constraints on response time
  - **transactions-per-second-per-dollar** accounts for cost of owning system
- TPC benchmark requires database sizes to be scaled up with increasing transactions-per-second
  - reflects real world applications where more customers means more database size and more transactions-per-second
- External audit of TPC performance numbers mandatory

# TPC Performance Measures

- Two types of tests for TPC-H and TPC-R
  - **Power test**: runs queries and updates sequentially, then takes mean to find queries per hour
  - **Throughput test**: runs queries and updates concurrently
    - multiple streams running in parallel each generates queries, with one parallel update stream
  - **Composite query per hour metric**: square root of product of power and throughput

# Other Benchmarks

- OODB transactions require a different set of benchmarks.
  - OO7 benchmark has several different operations, and provides a separate benchmark number for each kind of operation
  - Reason: hard to define what is a typical OODB application
- Benchmarks for XML being discussed

# STANDARDIZATION

# Standardization

- The complexity of contemporary database systems and the need for their interoperation require a variety of standards.
  - syntax and semantics of programming languages
  - functions in application program interfaces
  - data models (e.g. object oriented/object relational databases)
- Formal standards are standards developed by a standards organization (ANSI, ISO) or by industry groups

# Standardization (Cont.)

- Anticipatory standards lead the market place, defining features that vendors then implement
  - Ensure compatibility of future products
  - But at times become very large and unwieldy since standards bodies may not pay enough attention to ease of implementation (e.g.,SQL-92 or SQL:1999)
- Reactionary standards attempt to standardize features that vendors have already implemented, possibly in different ways.

# SQL Standards History

- SQL developed by IBM in late 70s/early 80s
- SQL-86 first formal standard
- IBM SAA standard for SQL in 1987
- SQL-89 added features to SQL-86 that were already implemented in many systems
  - Was a reactionary standard
- SQL-92 added many new features to SQL-89 (anticipatory standard)
  - Defines levels of compliance (*entry, intermediate and full*)

# SQL Standards History (Cont.)

- SQL:1999

- Adds variety of new features --- extended data types, object orientation, procedures, triggers, etc.
- Broken into several parts
  - SQL/Framework (Part 1): overview
  - SQL/Foundation (Part 2): types, schemas, tables, query/update statements, security, etc
  - SQL/CLI (Call Level Interface) (Part 3): API interface
  - SQL/PSM (Persistent Stored Modules) (Part

# SQL Standards History (Cont.)

- More parts undergoing standardization process
  - Part 7: SQL/Temporal: temporal data
  - Part 9: SQL/MED (Management of External Data)
    - Interfacing of database to external data sources
      - Allows other databases, even files, can be viewed as part of the database
  - Part 10 SQL/OLB (Object Language Bindings): embedding SQL in Java
  - Missing part numbers 6 and 8 cover

# Database Connectivity

**Standards** Open DataBase Connectivity (ODBC)  
standard for database interconnectivity

- based on *Call Level Interface* (CLI) developed by X/Open consortium
- defines application programming interface, and SQL features that must be supported at different levels of compliance
- JDBC standard used for Java
- X/Open XA standards define transaction management standards for supporting distributed 2-phase commit
- OLE-DB: API like ODBC, but intended to support non-database sources of data such as flat files

# Object Oriented Databases

- Object Database **Standards** Management Group (ODMG) standard for object-oriented databases

- version 1 in 1993 and version 2 in 1997, version 3 in 2000
- provides language independent *Object Definition Language* (ODL) as well as several language specific *bindings*

- Object Management Group (OMG) standard for distributed software based on objects

- Object Request Broker (ORB) provides transparent message dispatch to

# XML-Based Standards

- Several XML based Standards for E-commerce
  - E.g. RosettaNet (supply chain), BizTalk
  - Define catalogs, service descriptions, invoices, purchase orders, etc.
  - XML wrappers are used to export information from relational databases to XML
- Simple Object Access Protocol (SOAP): XML based remote procedure call standard



# E-COMMERCE

# E-Commerce

- E-commerce is the process of carrying out various activities related to commerce through electronic means
- Activities include:
  - Presale activities: catalogs, advertisements, etc
  - Sale process: negotiations on price/quality of service
  - Marketplace: e.g. stock exchange, auctions, reverse auctions
  - Payment for sale

# E-Catalogs

- Product catalogs must provide searching and browsing facilities
  - Organize products into intuitive hierarchy
  - Keyword search
  - Help customer with comparison of products
- Customization of catalog
  - Negotiated pricing for specific organizations
  - Special discounts for customers based on past history

# Marketplaces

- Marketplaces help in negotiating the price of a product when there are multiple sellers and buyers
- Several types of marketplaces
  - Reverse auction
  - Auction
  - Exchange
- Real world marketplaces can be quite complicated due to product differentiation
- Database issues:
  - Authenticate bidders
  - Record buy/sell bids securely

# Types of Marketplace

- Reverse auction system: single buyer, multiple sellers.
  - Buyer states requirements, sellers bid for supplying items. Lowest bidder wins. (also known as tender system)
  - Open bidding vs. closed bidding
- Auction: Multiple buyers, single seller
  - Simplest case: only one instance of each item is being sold
  - Highest bidder for an item wins
  - More complicated with multiple copies, and buyers bid for specific number of copies

# Order Settlement

- Order settlement: payment for goods and delivery
- Insecure means for electronic payment: send credit card number
  - Buyers may present some one else's credit card numbers
  - Seller has to be trusted to bill only for agreed-on item
  - Seller has to be trusted not to pass on the credit card number to unauthorized people

# Secure Payment Systems

- All information must be encrypted to prevent eavesdropping
  - Public/private key encryption widely used
- Must prevent person-in-the-middle attacks
  - E.g. someone impersonates seller or bank/credit card company and fools buyer into revealing information
    - Encrypting messages alone doesn't solve this problem
    - More on this in next slide
- Three-way communication between seller, buyer and credit-card company to

# Secure Payment Systems (Cont.)

- Digital certificates are used to prevent impersonation/man-in-the middle attack
  - Certification agency creates digital certificate by encrypting, e.g., seller's public key using its own private key
    - Verifies sellers identity by external means first!
  - Seller sends certificate to buyer
  - Customer uses public key of certification agency to decrypt certificate and find sellers public key

# Digital Cash

- Credit-card payment does not provide anonymity
  - The SET protocol hides buyers identity from seller
  - But even with SET, buyer can be traced with help of credit card company
- Digital cash systems provide anonymity similar to that provided by physical cash
  - E.g. DigiCash
  - Based on encryption techniques that make it impossible to find out who purchased digital cash from the bank

# LEGACY SYSTEMS

# Legacy Systems

- Legacy systems are older-generation systems that are incompatible with current generation standards and systems but still in production use
  - E.g. applications written in Cobol that run on mainframes
    - Today's hot new system is tomorrow's legacy system!
- Porting legacy system applications to a more modern environment is problematic
  - Very expensive, since legacy system may involve millions of lines of code, written over decades

# Legacy Systems (Cont.)

- Rewriting legacy application requires a first phase of understanding what it does
  - Often legacy code has no documentation or outdated documentation
  - **reverse engineering:** process of going over legacy code to
    - Come up with schema designs in ER or OO model
    - Find out what procedures and processes are implemented, to get a high level view of system

# Legacy Systems (Cont.)

- Switching over from old to new system is a major problem
  - Production systems are in every day, generating new data
  - Stopping the system may bring all of a company's activities to a halt, causing enormous losses
- Big-bang approach:
  1. Implement complete new system
  2. Populate it with data from old system
  1. No transactions while this step is executed

# Legacy Systems (Cont.)

- Chicken-little approach:
  - Replace legacy system one piece at a time
  - Use wrappers to interoperate between legacy and new code
    - E.g. replace front end first, with wrappers on legacy backend
      - Old front end can continue working in this phase in case of problems with new front end
    - Replace back end, one functional unit at a time
      - All parts that share a database may have to be replaced together, or wrapper is needed on database also
  - Drawback: significant extra development



END OF CHAPTER

# Chapter 22: Advanced Querying and Information Retrieval

## Decision-Support Systems

- Data Analysis
  - OLAP
  - Extended aggregation features in SQL
    - Windowing and ranking
- Data Mining
- Data Warehousing
- Information-Retrieval Systems
  - Including Web search

# Decision Support Systems

- Decision-support systems are used to make business decisions often based on data collected by on-line transaction-processing systems.
- Examples of business decisions:
  - what items to stock?
  - What insurance premium to change?
  - Who to send advertisements to?
- Examples of data used for making decisions
  - Retail sales transaction details

# Decision-Support Systems: Overview

- Data analysis tasks are simplified by specialized tools and SQL extensions
  - Example tasks
    - For each product category and each region, what were the total sales in the last quarter and how do they compare with the same quarter last year
    - As above, for each product category and each customer category
- Statistical analysis packages (e.g., : S++) can be interfaced with databases
  - Statistical analysis is a large field will not study it here
- Data mining seeks to discover knowledge automatically in the form of statistical rules and patterns from Large databases.
- A data warehouse archives information gathered from multiple sources, and stores it under a unified schema, at a single site.
  - Important for large businesses which

# Data Analysis and OLAP

- Aggregate functions summarize large volumes of data
- Online Analytical Processing (OLAP)
  - Interactive analysis of data, allowing data to be summarized and viewed in different ways in an online fashion (with negligible delay)
- Data that can be modeled as dimension attributes and measure attributes are called multidimensional data.
  - Given a relation used for data analysis, we can identify some of its attributes as

# Cross Tabulation of *sales* by item

size: all

item-name	color			Total
	dark	pastel	white	
skirt	8	35	10	53
dress	20	10	5	35
shirt	14	7	28	49
pant	20	2	5	27
Total	62	54	48	164

- The table above is an example of a cross-tabulation (cross-tab), also referred to as a pivot-table.
- A cross-tab is a table where
  - values for one of the dimension attributes form the row headers, values for another dimension attribute form the column headers

# Relational Representation of Crosstabs

Crosstabs can be represented as relations

The value **all** is used to represent aggregates

The SQL:1999 standard actually uses null values in place of **all**

More on this later....

item-name	color	number
skirt	dark	8
skirt	pastel	35
skirt	white	10
skirt	<b>all</b>	53
dress	dark	20
dress	pastel	10
dress	white	5
dress	<b>all</b>	35
shirt	dark	14
shirt	pastel	7
shirt	white	28
shirt	<b>all</b>	49
pant	dark	20
pant	pastel	2
pant	white	5
pant	<b>all</b>	27
<b>all</b>	dark	62
<b>all</b>	pastel	54
<b>all</b>	white	48
<b>all</b>	<b>all</b>	164

# Three-Dimensional Data Cube

A **data cube** is a multidimensional generalization of a crosstab

Cannot view a three-dimensional object in its entirety  
but crosstabs can be used as views on a data cube

		item name					size			
		skirt	dress	shirts	pant	all	all	large	medium	small
color		dark	20	14	20	62	34	4	18	16
pastel	dark	8	20	14	20	62	34	4	18	16
	pastel	35	10	7	2	54	21	9	45	42
white		dark	10	8	28	5	48	77	42	45
all		dark	53	35	49	27	164	all	large	medium

# Online Analytical Processing

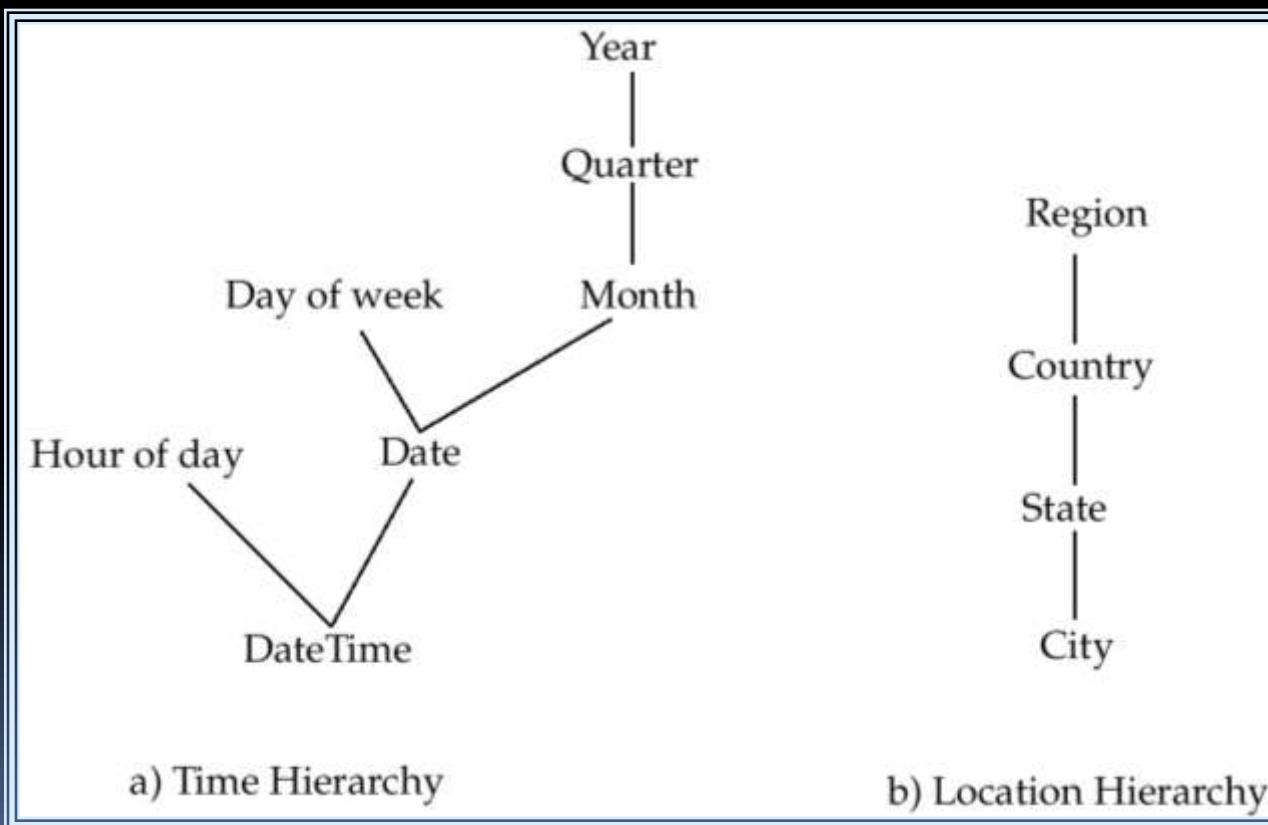
- The operation of changing the dimensions used in a cross-tab is called pivoting
- Suppose an analyst wishes to see a cross-tab on *item-name* and *color* for a fixed value of *size*, for example, large, instead of the sum across all sizes.
  - Such an operation is referred to as **slicing**.
    - The operation is sometimes called **dicing**, particularly when values for multiple dimensions are fixed.

The operation of moving from finer-

# Hierarchies on Dimensions

**Hierarchy** on dimension attributes: lets dimensions to be viewed at different levels of detail

- E.g. the dimension DateTime can be used to aggregate by hour of day, date, day of week, month, quarter or year



# Cross Tabulation With Hierarchy

Crosstabs can be easily extended to deal with hierarchies

- ❖ Can drill down or roll up on a hierarchy

<i>category</i>	<i>item-name</i>	dark	pastel	white	total	
womenswear	skirt	8	8	10	53	
	dress	20	20	5	35	
	subtotal	28	28	15		88
menswear	pants	14	14	28	49	
	shirt	20	20	5	27	
	subtotal	34	34	33		76
total		62	62	48		164

# OLAP Implementation

- The earliest OLAP systems used multidimensional arrays in memory to store data cubes, and are referred to as multidimensional OLAP (MOLAP) systems.
- OLAP implementations using only relational database features are called relational OLAP (ROLAP) systems
- Hybrid systems, which store some summaries in memory and store the base data and other summaries in a relational database, are called hybrid

# OLAP Implementation (Cont.)

- Early OLAP systems precomputed *all* possible aggregates in order to provide online response
  - Space and time requirements for doing so can be very high
    - $2^n$  combinations of **group by**
  - It suffices to precompute some aggregates, and compute others on demand from one of the precomputed aggregates
    - Can compute aggregate on (*item-name, color*) from an aggregate on (*item-name, color, size*)
      - For all but a few “non-decomposable” aggregates such as *median*
      - is cheaper than computing it from scratch
- Several optimizations available for

# Extended Aggregation

- SQL-92 aggregation quite limited
  - Many useful aggregates are either very hard or impossible to specify
    - Data cube
    - Complex aggregates (median, variance)
    - binary aggregates (correlation, regression curves)
    - ranking queries (“assign each student a rank based on the total marks”)
- SQL:1999 OLAP extensions provide a variety of aggregation functions to address above limitations

# Extended Aggregation in

**SQL:1999**

- The cube operation computes union of group by's on every subset of the specified attributes
- E.g. consider the query

```
select item-name, color, size,
sum(number)
from sales
group by cube(item-name, color, size)
```

This computes the union of eight different groupings of the *sales* relation:

{ (*item-name*, *color*, *size*), (*item-name*,  
*color*),

# Extended Aggregation (Cont.)

- Relational representation of crosstab that we saw earlier, but with *null* in place of all, can be computed by

```
select item-name, color, sum(number)
from sales
group by cube(item-name, color)
```

- The function grouping() can be applied on an attribute
  - Returns 1 if the value is a null value representing all, and returns 0 in all other cases.

```
select item-name, color, size, sum(number),
grouping(item-name) as item-name-flag,
grouping(color) as color-flag,
grouping(size) as size-flag,
from sales
group by cube(item-name, color, size)
```

- Can use the function decode() in the select clause to replace such nulls by a value such as all
  - E.g. replace *item-name* in first query by  
`decode( grouping(item-name), 1, 'all',`

# Extended Aggregation (Cont.)

- The rollup construct generates union on every prefix of specified list of attributes
- E.g.

```
select item-name, color, size, sum(number)
from sales
group by rollup(item-name, color, size)
```

- Generates union of four groupings:  
{ (*item-name*, *color*, *size*), (*item-name*, *color*), (*item-name*), () }

- Rollup can be used to generate aggregates at multiple levels of a hierarchy.
- E.g., suppose table *itemcategory*(*item-name*, *category*) gives the category of each item. Then

```
select category, item-name, sum(number)
from sales, itemcategory
where sales.item-name = itemcategory.item-name
group by rollup(category, item-name)
```

would give a hierarchical summary by *item-name* and by *category*.

# Extended Aggregation (Cont.)

- Multiple rollups and cubes can be used in a single group by clause
  - Each generates set of group by lists, cross product of sets gives overall set of group by lists
- E.g.,

```
select item-name, color, size,
sum(number)
from sales
group by rollup(item-name),
rollup(color, size)
generates the groupings
```

# Ranking

- Ranking is done in conjunction with an order by specification.
- Given a relation student-marks(*student-id*, marks) find the rank of each student.  
*select student-id, rank( ) over (order by marks desc) as s-rank  
from student-marks*
- An extra order by clause is needed to get them in sorted order  
*select student-id, rank ( ) over (order by marks desc) as s-rank  
from student-marks*

# Ranking (Cont.)

- Ranking can be done within partition of the data.
- “Find the rank of students within each section.”

```
select student-id, section,
 rank () over (partition by section order by marks desc)
 as sec-rank
from student-marks, student-section
where student-marks.student-id = student-section.student-id
order by section, sec-rank
```

- Multiple rank clauses can occur in a single select clause
- Ranking is done *after* applying group by clause/aggregation
- Exercises:
  - Find students with top n ranks
    - Many systems provide special (non-standard) syntax for “top-n” queries
  - Rank students by sum of their marks in different courses

# Ranking (Cont.)

- Other ranking functions:
  - **percent\_rank** (within partition, if partitioning is done)
  - **cume\_dist** (cumulative distribution)
    - fraction of tuples with preceding values
  - **row\_number** (non-deterministic in presence of duplicates)
- SQL:1999 permits the user to specify nulls first or nulls last

select *student-id*,  
rank () over (order by *marks*

# Ranking (Cont.)

- For a given constant  $n$ , the ranking function `ntile( $n$ )` takes the tuples in each partition in the specified order, and divides them into  $n$  buckets with equal numbers of tuples. For instance, we can sort employees by salary, and use `ntile(3)` to find which range (bottom third, middle third, or top third) each employee is in, and compute the total salary earned by employees in each range:

```
select threetile, sum(salary)
```

# Windowing

- E.g.: “Given sales values for each date, calculate for each date the average of the sales on that day, the previous day, and the next day”
- Such *moving average* queries are used to smooth out random variations.
- In contrast to group by, the same tuple can exist in multiple windows
- Window specification in SQL:
  - Ordering of tuples, size of window for each tuple, aggregate function
  - E.g. given relation *sales(date, value)*

```
select date, sum(value) over
 (order by date between rows 1 preceding and 1 following)
 from sales
```
- Examples of other window specifications:
  - **between rows unbounded preceding and current**
  - **rows unbounded preceding**

# Windowing (Cont.)

- Can do windowing within partitions
- E.g. Given a relation  
*transaction(account-number, date-time, value)*, where value is positive for a deposit and negative for a withdrawal
  - “Find total balance of each account after each transaction on the account”  
**select account-number, date-time,  
sum(value) over  
(partition by account-number**

# DATA MINING

# Data Mining

- Broadly speaking, data mining is the process of semi-automatically analyzing large databases to find useful patterns
- Like knowledge discovery in artificial intelligence data mining discovers statistical rules and patterns
- Differs from machine learning in that it deals with large volumes of data stored primarily on disk.
- Some types of knowledge discovered from a database can be represented

# Applications of Data Mining

- Prediction based on past history
  - Predict if a credit card applicant poses a good credit risk, based on some attributes (income, job type, age, ..) and past history
  - Predict if a customer is likely to switch brand loyalty
  - Predict if a customer is likely to respond to “junk mail”
  - Predict if a pattern of phone calling card usage is likely to be fraudulent
- Some examples of prediction mechanisms:
  - **Classification**
    - Given a training set consisting of items belonging to different classes, and a new

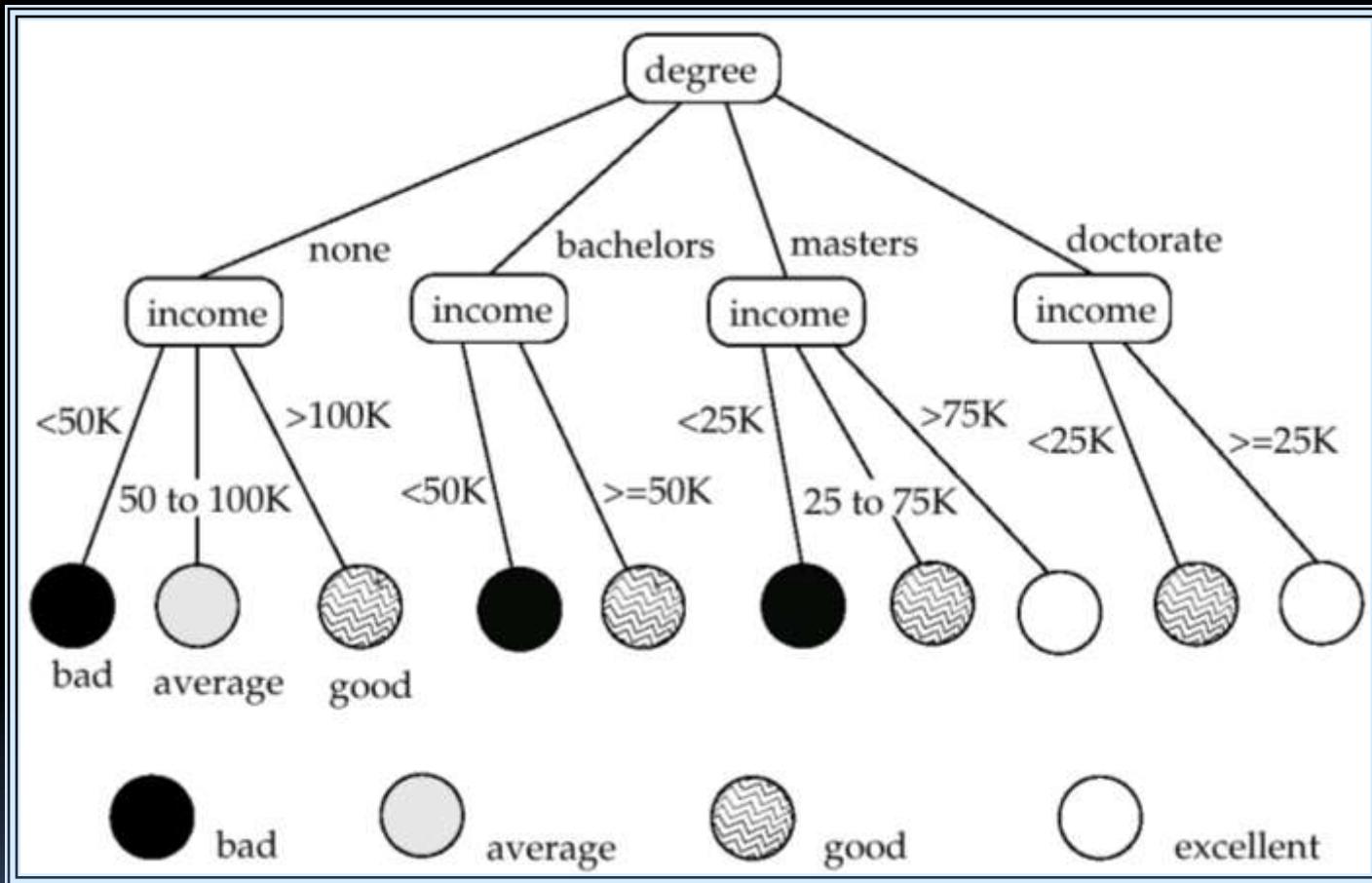
# Applications of Data Mining (Cont.)

- Descriptive Patterns
  - Associations
    - Find books that are often bought by the same customers. If a new customer buys one such book, suggest that he buys the others too.
    - Other similar applications: camera accessories, clothes, etc.
  - Associations may also be used as a first step in detecting causation
    - E.g. association between exposure to chemical X and cancer, or new medicine and disease Y

# Classification Rules

- Classification rules help assign new objects to a set of classes. E.g., given a new automobile insurance applicant, should he or she be classified as low risk, medium risk or high risk?
- Classification rules for above example could use a variety of knowledge, such as educational level of applicant, salary of applicant, age of applicant, etc.
  - $\forall$  person P, P.degree = masters and P.income > 75,000

# Decision Tree



# Construction of Decision Trees

- Training set: a data sample in which the grouping for each tuple is already known.
- Consider credit risk example: Suppose *degree* is chosen to partition the data at the root.
  - Since *degree* has a small number of possible values, one child is created for each value.
- At each child node of the root, further classification is done if required. Here, partitions are defined by *income*.
  - Since *income* is a continuous attribute, some number of intervals are chosen, and one child created for each interval.
- Different classification algorithms use different ways of choosing which attribute to partition on at each node, and what the intervals, if any, are.
- In general

# Construction of Decision Trees

(Cont.)

Greedy top down generation of decision trees.

- Each internal node of the tree partitions the data into groups based on a **partitioning attribute**, and a **partitioning condition** for the node
  - More on choosing partitioning attribute/condition shortly
  - Algorithm is greedy: the choice is made once and not revisited as more of the tree is constructed
- The data at a node is not partitioned further if either
  - all (or most) of the items at the node

# Best Splits

Idea: evaluate different attributes and partitioning conditions and pick the one that best improves the “purity” of the training set examples

- The initial training set has a mixture of instances from different classes and is thus relatively impure
- E.g. if degree exactly predicts credit risk, partitioning on degree would result in each child having instances of only one class
  - I.e., the child nodes would be *pure*

- The purity of a set  $S$  of training instances can be measured quantitatively in several ways.

- Notation: number of classes  $\kappa = k$ , number of instances  $= |S|$ , fraction of instances in class  $i = p_i^2$

- The Gini measure of purity is defined as

$$\text{Gini}(S) = 1 - \sum$$

- When all instances are in a single class, the

# Best Splits (Cont.)

*k*

- Another measure of purity is the entropy measure, which is defined as

$$\text{entropy}(S) = - \sum_{i=1}^k p_i \log_2 p_i$$

- When a set  $S$  is split into multiple sets,  $S_i, i=1, 2, \dots, r$ , we can measure the purity of the resultant set of sets as:

$$\text{purity}(S_1, S_2, \dots, S_r) = \sum_{i=1}^r \frac{|S_i|}{|S|} \text{purity}(S_i)$$

- The information gain due to particular split of  $S$  into  $S_i, i = 1, 2, \dots, r$

$$\begin{aligned}\text{Information-gain}(S, \{S_1, S_2, \dots, S_r\}) &= \\ &\text{purity}(S) - \text{purity}(S_1, S_2, \dots, S_r)\end{aligned}$$

# Best Splits (Cont.)

- Measure of “cost” of a split:

$$\text{Information-content}(S, \{S_1, S_2, \dots, S_r\}) =$$

$$\frac{\sum_{i=1}^r |S_i|}{|S|} \log_2 \frac{|S|}{\sum_{i=1}^r |S_i|}$$

- Information-gain ratio =  $\frac{\text{Information-gain}(S, \{S_1, S_2, \dots, S_r\})}{\text{Information-content}(S, \{S_1, S_2, \dots, S_r\})}$
- The best split for an attribute is the one that gives the maximum information gain ratio
- Continuous valued attributes
  - Can be ordered in a fashion meaningful to classification
  - e.g. integer and real values
- Categorical attributes
  - Cannot be meaningfully ordered (e.g. country, school/university, item-color, .): 144

# Finding Best Splits

- Categorical attributes:
  - Multi-way split, one child for each value
    - may have too many children in some cases
  - Binary split: try all possible breakup of values into two sets, and pick the best
- Continuous valued attribute
  - Binary split:
    - Sort values in the instances, try each as a split point
      - E.g. if values are 1, 10, 15, 25, split at  $\leq 1$ ,  $\leq 10$ ,  $\leq 15$
    - Pick the value that gives best split

# Decision-Tree Construction Algorithm

```
Procedure Grow Tree(S)
 Partition(S);
```

```
Procedure Partition (S)
 if ($purity(S) > \delta_p$ or $|S| < \delta_s$) then
 return;
 for each attribute A
 evaluate splits on attribute A ;
 Use best split found (across all attributes) to partition
 S into S_1, S_2, \dots, S_r
 for $i = 1, 2, \dots, r$
 Partition(S_i);
```

# Decision Tree Constr'n Algo's. (Cont.)

- Variety of algorithms have been developed to
  - Reduce CPU cost and/or
  - Reduce IO cost when handling datasets larger than memory
  - Improve accuracy of classification
- Decision tree may be overfitted, i.e., overly tuned to given training set
  - Pruning of decision tree may be done on branches that have too few training instances
    - When a subtree is pruned, an internal node becomes a leaf

# Other Types of Classifiers

- Further types of classifiers
  - Neural net classifiers
  - Bayesian classifiers
- Neural net classifiers use the training data to train artificial neural nets
  - Widely studied in AI, won't cover here
- Bayesian classifiers use Bayes theorem, which says

$$p(c_j | d) = \frac{p(d | c_j) p(c_j)}{p(d)}$$

where

$p(c_j | d)$  = probability of instance  $d$  being in class  $c_j$ ,

# Naïve Bayesian Classifiers

- Bayesian classifiers require
  - computation of  $p(d | c_j)$
  - precomputation of  $p(c_j)$
  - $p(d)$  can be ignored since it is the same for all classes
- To simplify the task, naïve Bayesian classifiers assume attributes have independent distributions, and thereby estimate

$$p(d | c_j) = p(d_1 | c_j) * p(d_2 | c_j) * \dots * \\ (p(d_n | c_j))$$

# Regression

- Regression deals with the prediction of a value, rather than a class.
  - Given values for a set of variables,  $X_1, X_2, \dots, X_n$ , we wish to predict the value of a variable  $Y$ .
- One way is to infer coefficients  $a_0, a_1, a_2, \dots, a_n$  such that

$$Y = a_0 + a_1 * X_1 + a_2 * X_2 + \dots + a_n * X_n$$

- Finding such a linear polynomial is called linear regression.
  - In general, the process of finding a curve that fits the data is also called curve fitting.

# Association Rules

- Retail shops are often interested in associations between different items that people buy.
  - Someone who buys bread is quite likely also to buy milk
  - A person who bought the book *Database System Concepts* is quite likely also to buy the book *Operating System Concepts*.

- Associations information can be used in several ways.

- E.g. when a customer buys a particular book, an online shop may suggest associated books.

Association rules:

# Association Rules (Cont.)

- Rules have an associated support, as well as an associated confidence.
- Support is a measure of what fraction of the population satisfies both the antecedent and the consequent of the rule.
  - E.g. suppose only 0.001 percent of all purchases include milk and screwdrivers. The support for the rule is  $milk \Rightarrow screwdrivers$  is low.
  - We usually want rules with a reasonably high support
    - Rules with low support are usually not very useful

Confidence is a measure of how often

# Finding Association Rules

- We are generally only interested in association rules with reasonably high support (e.g. support of 2% or greater)
- Naïve algorithm
  1. Consider all possible sets of relevant items.
  2. For each set find its support (i.e. count how many transactions purchase all items in the set).
    - ❖ Large itemsets: sets with sufficiently high support
  3. Use large itemsets to generate association rules

# Finding Support

- Few itemsets: determine support of all itemsets via a single pass on set of transactions
  - A count is maintained for each itemset, initially set to 0.
  - When a transaction is fetched, the count is incremented for each set of items that is contained in the transaction.
  - Large itemsets: sets with a high count at the end of the pass
- Many itemsets: If memory not enough to hold all counts for all itemsets use multiple passes, considering only some itemsets in each pass.
- Optimization: Once an itemset is eliminated because its count (support) is too small none of its supersets needs to be considered.
- The a priori technique to find large itemsets:
  - Pass 1: count support of all sets with just 1

# Other Types of Associations

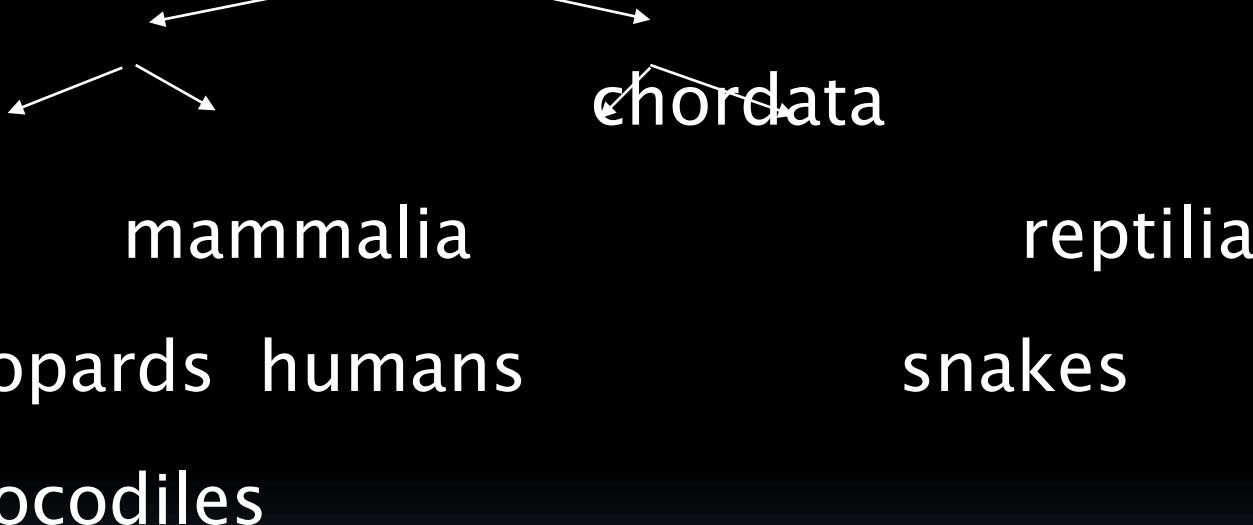
- Basic association rules have several limitations
- Deviations from the expected probability are more interesting
  - E.g. if many people purchase bread, and many people purchase cereal, quite a few would be expected to purchase both ( $\text{prob1} * \text{prob2}$ )
  - We are interested in positive as well as negative correlations between sets of items
    - Positive correlation: co-occurrence is higher than predicted
    - Negative correlation: co-occurrence is lower than predicted

# Clustering

- Clustering: Intuitively, finding clusters of points in the given data such that similar points lie in the same cluster
- Can be formalized using distance metrics in several ways
- E.g. Group points into  $k$  sets (for a given  $k$ ) such that the average distance of points from the centroid of their assigned group is minimized
  - Centroid: point defined by taking average of coordinates in each dimension.
  - Another metric: minimize average distance to the nearest centroid

# Hierarchical Clustering

- Example from biological classification
  - (the word classification here does not mean a prediction mechanism)



- Other examples: Internet directory systems (e.g. Yahoo, more on this later)
- Agglomerative clustering algorithms
  - Build small clusters, then cluster small

# Clustering Algorithms

- Clustering algorithms have been designed to handle very large datasets
- E.g. the Birch algorithm
  - Main idea: use an in-memory R-tree to store points that are being clustered
  - Insert points one at a time into the R-tree, merging a new point with an existing cluster if it is less than some  $\delta$  distance away
  - If there are more leaf nodes than fit in memory, merge existing clusters that are close to each other

# Collaborative Filtering

- Goal: predict what movies/books/... a person may be interested in, on the basis of
  - Past preferences of the person
  - Other people with similar past preferences
  - The preferences of such people for a new movie/book/...
- One approach based on repeated clustering
  - Cluster people on the basis of preferences for movies
  - Then cluster movies on the basis of being

# Other Types of Mining

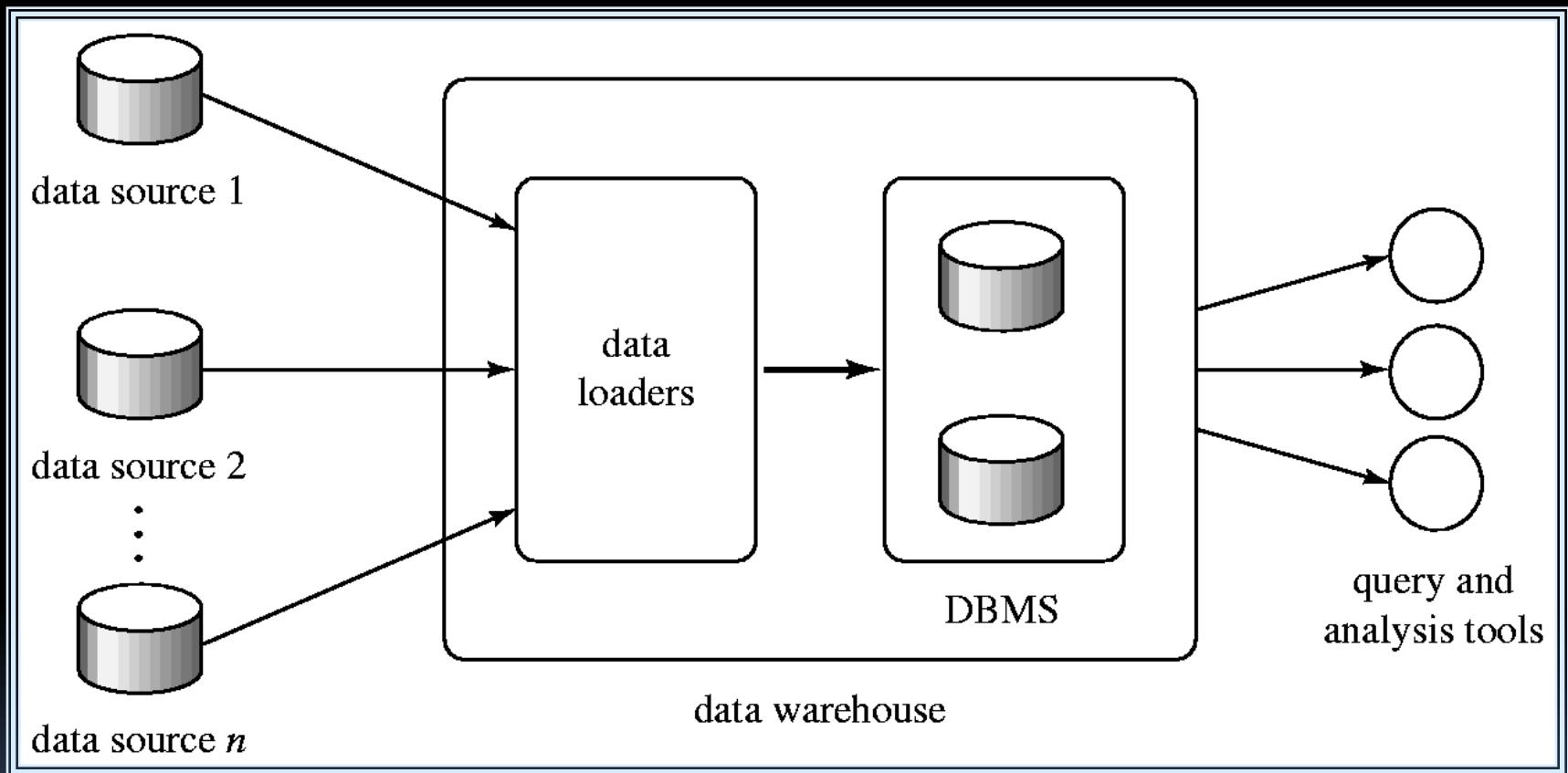
- Text mining: application of data mining to textual documents
  - E.g. cluster Web pages to find related pages
  - E.g. cluster pages a user has visited to organize their visit history
  - E.g. classify Web pages automatically into a Web directory
- Data visualization systems help users examine large volumes of data and detect patterns visually

# DATA WAREHOUSING

# Data Warehousing

- Large organizations have complex internal organizations, and have data stored at different locations, on different operational (transaction processing) systems, under different schemas
- Data sources often store only current data, not historical data
- Corporate decision making requires a unified view of all organizational data, including historical data

# Data Warehousing



# Components of Data Warehouse

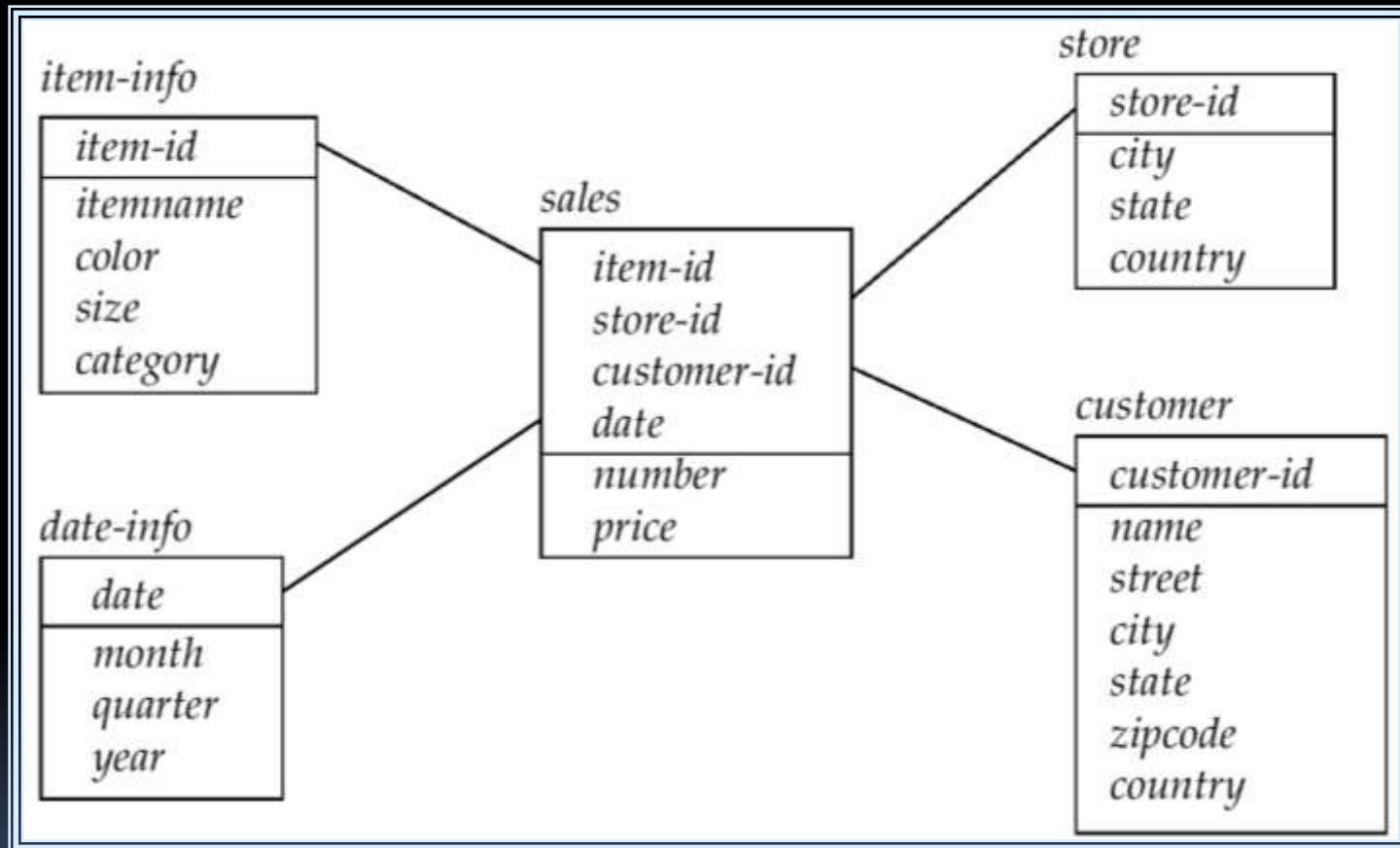
## *When and how to gather data*

- Source driven architecture: data sources transmit new information to warehouse, either continuously or periodically (e.g. at night)
- Destination driven architecture: warehouse periodically requests new information from data sources
- Keeping warehouse exactly synchronized with data sources (e.g. using two-phase commit) is too expensive
  - Usually OK to have slightly out-of-date data at warehouse
  - Data/updates are periodically downloaded

# Components of Data Warehouse (Cont.)

- *Data cleansing*
  - E.g. correct mistakes in addresses
    - E.g. misspellings, zip code errors
  - Merge address lists from different sources and purge duplicates
    - Keep only one address record per household (“householding”)
- *How to propagate updates*
  - Warehouse schema may be a (materialized) view of schema from data sources
  - Efficient techniques for update of materialized views

# Data Warehouse Schemas



# Warehouse Schemas

- Typically warehouse data is multidimensional, with very large fact tables
  - Examples of dimensions: item-id, date/time of sale, store where sale was made, customer identifier
  - Examples of measures: number of items sold, price of items
- Dimension values are usually encoded using small integers and mapped to full values via dimension tables

# INFORMATION RETRIEVAL

# Information Retrieval Systems

- Information retrieval (IR) systems use a simpler data model than database systems
  - Information organized as a collection of documents
  - Documents are unstructured, no schema
- Information retrieval locates relevant documents, on the basis of user input such as keywords or example documents
  - e.g., find documents containing the words “database systems”
- Can be used even on textual

# Information Retrieval Systems (Cont.)

- Differences from database systems
  - IR systems don't deal with transactional updates (including concurrency control and recovery)
  - Database systems deal with structured data, with schemas that define the data organization
  - IR systems deal with some querying issues not generally addressed by database systems
    - Approximate searching by keywords
    - Ranking of retrieved answers by estimated

# Keyword Search

- In full text retrieval, all the words in each document are considered to be keywords.
  - We use the word **term** to refer to the words in a document
- Information-retrieval systems typically allow query expressions formed using keywords and the logical connectives *and*, *or*, and *not*
  - *Ands* are implicit, even if not explicitly specified
- Ranking of documents on the basis of estimated relevance to a query is critical
  - Relevance ranking is based on factors such as
    - Term frequency
      - Frequency of occurrence of query keyword in document
    - Inverse document frequency

# Relevance Ranking Using Terms

- TF-IDF (Term frequency/Inverse Document frequency) ranking:
  - Let  $n(d)$  = number of terms in the document  $d$
  - $n(d, t)$  = ~~number of occurrences of term  $t$  in the document  $d$ .~~  $\log\left(\frac{n(d, t)}{n(d)}\right)$
  - Then relevance of a document  $d$  to a term  $t$

$$r(d, Q) = \sum_{t \in Q} \frac{r(d, t)}{n(t)}$$

- The log factor is to avoid excessive weightage to frequent terms

# Relevance Ranking Using Terms (Cont.)

- Most systems add to the above model
  - Words that occur in title, author list, section headings, etc. are given greater importance
  - Words whose first occurrence is late in the document are given lower importance
  - Very common words such as “a”, “an”, “the”, “it” etc are eliminated
    - Called **stop words**
  - **Proximity:** if keywords in query occur close together in the document, the document has higher importance than if

# Relevance Using Hyperlinks

- When using keyword queries on the Web, the number of documents is enormous (many billions)
  - Number of documents relevant to a query can be enormous if only term frequencies are taken into account
- Using term frequencies makes “spamming” easy
  - E.g. a travel agent can add many occurrences of the words “travel agent” to his page to make its rank very high
- Most of the time people are looking for pages from popular sites

# Relevance Using Hyperlinks

- Solution: use number of hyperlinks to a site as a measure of the popularity or prestige of the site

□ Count only one hyperlink from each site (why?)

□ Popularity measure is for site, not for individual page

- Most hyperlinks are to root of site

- Site–popularity computation is cheaper than page popularity computation

- Refinements

- When computing prestige based on links to a site, give more weightage to links from sites that are themselves highly popular

# Relevance Using Hyperlinks

- (~~Conn.~~) Connections to social networking theories that ranked prestige of people
  - E.g. the president of the US has a high prestige since many people know him
  - Someone known by multiple prestigious people has high prestige
- Hub and authority based ranking
  - A **hub** is a page that stores links to many pages (on a topic)
  - An **authority** is a page that contains actual information on a topic
  - Each page gets a **hub prestige** based on prestige of authorities that it points to

# Similarity Based Retrieval

- Similarity based retrieval – retrieve documents similar to a given document
  - Similarity may be defined on the basis of common words
    - E.g. find  $k$  terms in A with highest  $r(d, t)$  and use these terms to find relevance of other documents; each of the terms carries a weight of  $r(d, t)$
  - Similarity can be used to refine answer set to keyword query
    - User selects a few relevant documents

# Synonyms and Homonyms

## ■ Synonyms

- E.g. document: “motorcycle repair”, query: “motorcycle maintenance”
  - need to realize that “maintenance” and “repair” are synonyms
- System can extend query as “motorcycle and (repair or maintenance)”

## ■ Homonyms

- E.g. “object” has different meanings as noun/verb
- Can disambiguate meanings (to some extent) from the context

## ■ Extending queries automatically using

# Indexing of Documents

- An inverted index maps each keyword  $K_i$  to a set of documents  $S_i$  that contain the keyword
  - Documents identified by identifiers
- Inverted index may record
  - Keyword locations within document to allow proximity based ranking
  - Counts of number of occurrences of keyword to compute TF
- and operation: Finds documents that contain all of  $K_1, K_2, \dots, K_n$ .
  - Intersection  $S_1 \cap S_2 \cap \dots \cap S_n$
- or operation: documents that contain at least one of  $K_1, K_2, \dots, K_n$

# Measuring Retrieval

Effectiveness systems save space by using index structures that support only approximate retrieval. May result in:

- **false negative (false drop)** – some relevant documents may not be retrieved.
- **false positive** – some irrelevant documents may be retrieved.
- For many applications a good index should not permit any false drops, but may permit a few false positives.
- Relevant performance metrics:
  - Precision – what percentage of the retrieved documents are relevant to the query?

# Measuring Retrieval

- Ranking order can also result in false positives/false negatives
- Effectiveness (Cont.)
- Recall vs. precision tradeoff:
  - Can increase recall by retrieving many documents (down to a low level of relevance ranking), but many irrelevant documents would be fetched, reducing precision
- Measures of retrieval effectiveness:
  - Recall as a function of number of documents fetched, or
  - Precision as a function of recall
    - Equivalently, as a function of number of documents fetched
  - E.g. “precision of 75% at recall of 50%, and 60% at a recall of 75%”
    - In general: draw a graph of precision vs recall

# Web Crawling

- Web crawlers are programs that locate and gather information on the Web
  - Recursively follow hyperlinks present in known documents, to find other documents
    - Starting from a *seed* set of documents
  - Fetched documents
    - Handed over to an indexing system
    - Can be discarded after indexing, or store as a *cached* copy
- Crawling the entire Web would take a very large amount of time

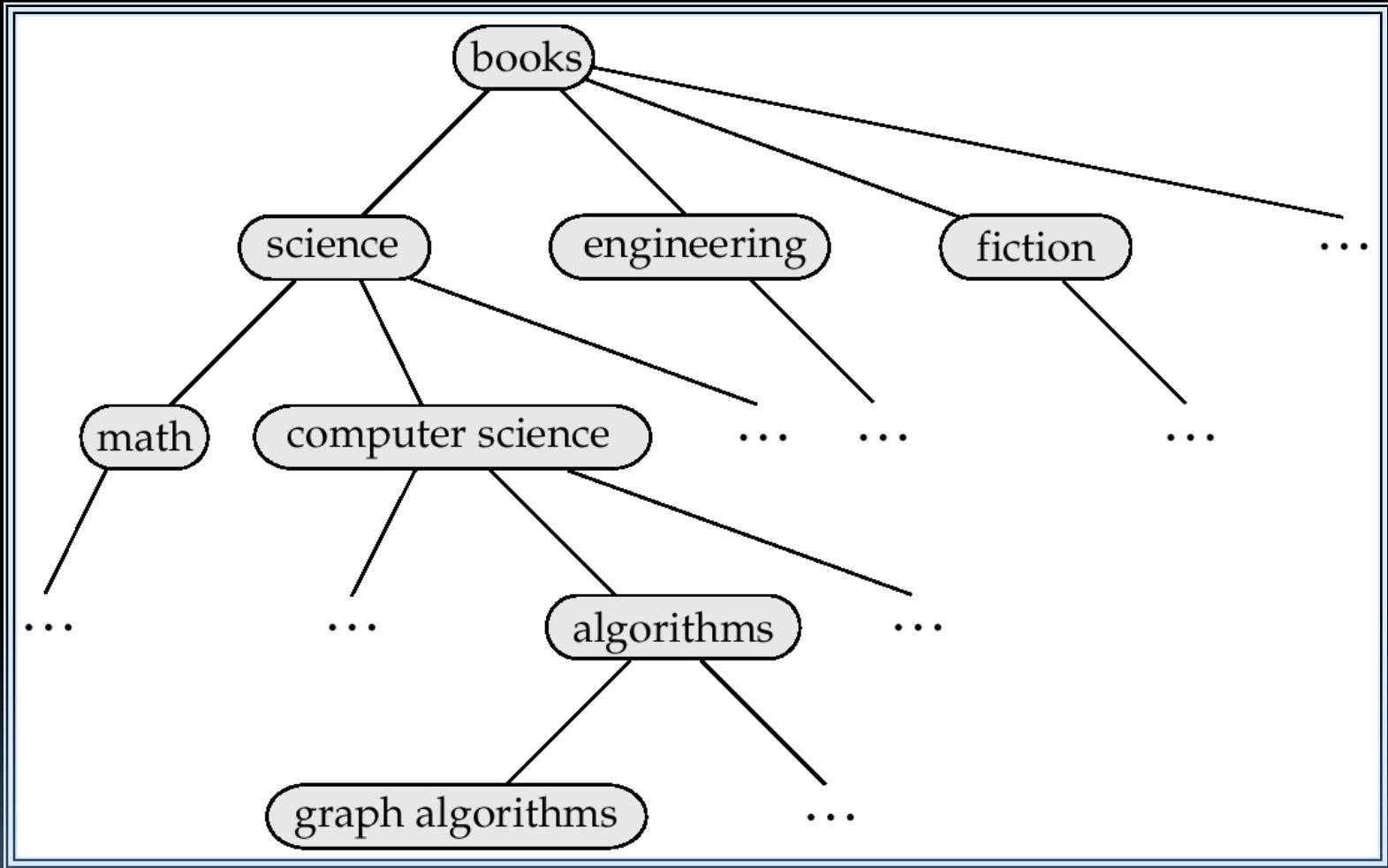
# Web Crawling (Cont.)

- Crawling is done by multiple processes on multiple machines, running in parallel
  - Set of links to be crawled stored in a database
  - New links found in crawled pages added to this set, to be crawled later
- Indexing process also runs on multiple machines
  - Creates a new copy of index instead of modifying old index

# Browsing

- Storing related documents together in a library facilitates browsing
  - users can see not only requested document but also related ones.
- Browsing is facilitated by classification system that organizes logically related documents together.
- Organization is hierarchical: classification hierarchy

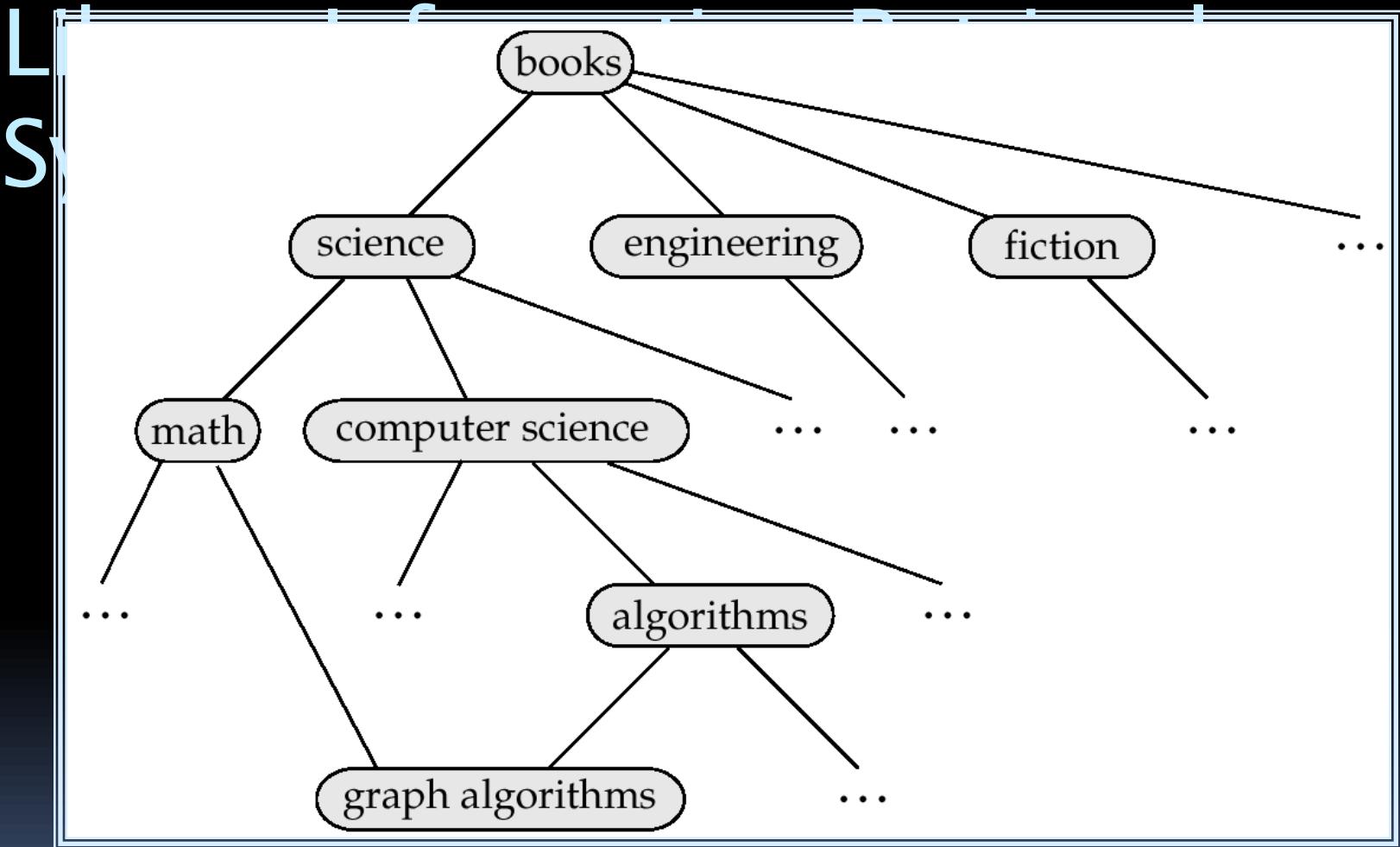
# A Classification Hierarchy For A Library System



# Classification DAG

- Documents can reside in multiple places in a hierarchy in an information retrieval system, since physical location is not important.
- Classification hierarchy is thus Directed Acyclic Graph (DAG)

# A Classification DAG For A



# Web Directories

- A Web directory is just a classification directory on Web pages
  - E.g. Yahoo! Directory, Open Directory project
  - Issues:
    - What should the directory hierarchy be?
    - Given a document, which nodes of the directory are categories relevant to the document
  - Often done manually
    - Classification of documents into a hierarchy may be done based on term similarity

# CHAPTER 23: ADVANCED DATA TYPES AND NEW APPLICATIONS

# Overview

- Temporal Data
- Spatial and Geographic Databases
- Multimedia Databases
- Mobility and Personal Databases

# Time In Databases

- While most databases tend to model reality at a point in time (at the ``current'' time), temporal databases model the states of the real world across time.
- Facts in temporal relations have associated times when they are *valid*, which can be represented as a union of intervals.
- The transaction time for a fact is the time interval during which the fact is current within the database system.

# Time In Databases (Cont.)

- Example of a temporal relation:

<i>account-number</i>	<i>branch-name</i>	<i>balance</i>	<i>from</i>		<i>to</i>	
A-101	Downtown	500	1999/1/1	9:00	1999/1/24	11:30
A-101	Downtown	100	1999/1/24	11:30	*	
A-215	Mianus	700	2000/6/2	15:30	2000/8/8	10:00
A-215	Mianus	900	2000/8/8	10:00	2000/9/5	8:00
A-215	Mianus	700	2000/9/5	8:00	*	
A-217	Brighton	750	1999/7/5	11:00	2000/5/1	16:00

- Temporal query languages have been proposed to simplify modeling of time as

# Time Specification in SQL-92

- date: four digits for the year (1--9999), two digits for the month (1--12), and two digits for the date (1--31).
- time: two digits for the hour, two digits for the minute, and two digits for the second, plus optional fractional digits.
- timestamp: the fields of date and time, with six fractional digits for the seconds field.
- Times are specified in the *Universal*

# Temporal Query Languages

- Predicates *precedes*, *overlaps*, and *contains* on time intervals.
- *Intersect* can be applied on two intervals, to give a single (possibly empty) interval; the union of two intervals may or may not be a single interval.
- A snapshot of a temporal relation at time t consists of the tuples that are valid at time t, with the time-interval attributes projected out.
- Temporal selection: involves time

# Temporal Query Languages

- Functional dependencies must be used with care: adding a time field may invalidate functional dependency  
*(Cont.)*
- A temporal functional dependency  $X \rightarrow Y$  holds on a relation schema R if, for all legal instances r of R, all snapshots of r satisfy the functional dependency  $X \rightarrow Y$ .
- SQL:1999 Part 7 (SQL/Temporal) is a proposed extension to SQL:1999 to improve support of temporal data.

# SPATIAL AND GEOGRAPHIC DATABASES

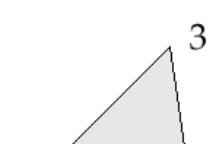
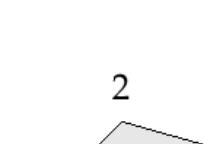
# Spatial and Geographic Databases

- Spatial databases store information related to spatial locations, and support efficient storage, indexing and querying of spatial data.
- Special purpose index structures are important for accessing spatial data, and for processing spatial join queries.
- Computer Aided Design (CAD) databases store design information about how objects are constructed  
E.g.: designs of buildings, aircraft, ...

# Represented of Geometric Information

- Various geometric constructs can be represented in a database in a normalized fashion.
- Represent a line segment by the coordinates of its endpoints.
- Approximate a curve by partitioning it into a sequence of segments
  - Create a list of vertices in order, or
  - Represent each segment as a separate tuple that also carries with it the identifier of the curve (2D features such as roads).
- Closed polygons
  - List of vertices in order, starting vertex is the same as the ending vertex, or
  - Represent boundary edges as separate tuples, with each containing identifier of the polygon. or

# Representation of Geometric Con

line segment	 $\{(x_1, y_1), (x_2, y_2)\}$
triangle	 $\{(x_1, y_1), (x_2, y_2), (x_3, y_3)\}$
polygon	 $\{(x_1, y_1), (x_2, y_2), (x_3, y_3), (x_4, y_4), (x_5, y_5)\}$
polygon	 $\{(x_1, y_1), (x_2, y_2), (x_3, y_3), \text{ID1}\}$ $\{(x_1, y_1), (x_3, y_3), (x_4, y_4), \text{ID1}\}$ $\{(x_1, y_1), (x_4, y_4), (x_5, y_5), \text{ID1}\}$

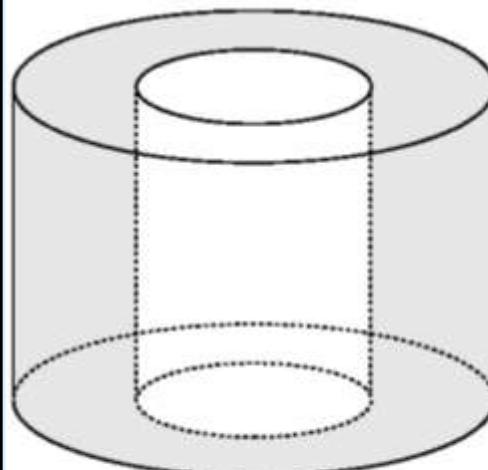
# Representation of Geometric Information (Cont.)

- Representation of points and line segment in 3-D similar to 2-D, except that points have an extra z component
- Represent arbitrary polyhedra by dividing them into tetrahedrons, like triangulating polygons.
- Alternative: List their faces, each of which is a polygon, along with an indication of which side of the face is inside the polyhedron.

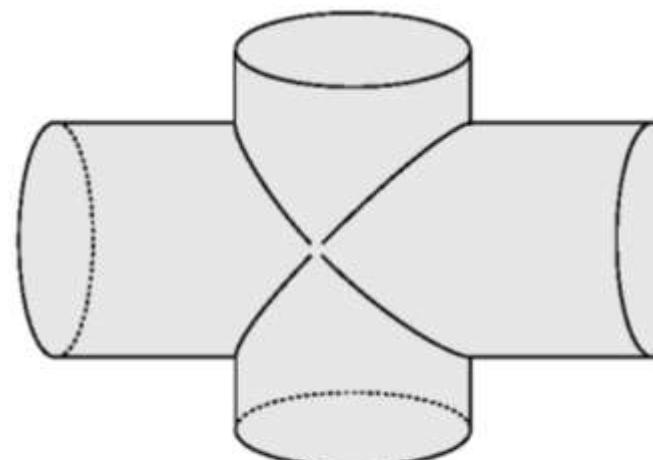
# Design Databases

- Represent design components as objects (generally geometric objects); the connections between the objects indicate how the design is structured.
- Simple two-dimensional objects: points, lines, triangles, rectangles, polygons.
- Complex two-dimensional objects: formed from simple objects via union, intersection, and difference operations.

# Representation of Geometric Constructs



(a) Difference of cylinders



(b) Union of cylinders

(a) Difference of cylinders

(b) Union of cylinders

Design databases also store non-spatial information about objects (e.g., construction material, color, etc.)

Spatial integrity constraints are important.

E.g. pipes should not intersect wires

# Geographic Data

- Raster data consist of bit maps or pixel maps, in two or more dimensions.
  - Example 2-D raster image: satellite image of cloud cover, where each pixel stores the cloud visibility in a particular area.
  - Additional dimensions might include the temperature at different altitudes at different regions, or measurements taken at different points in time.
- Design databases generally do not store raster data

# Geographic Data (Cont.)

- Vector data are constructed from basic geometric objects: points, line segments, triangles, and other polygons in two dimensions, and cylinders, spheres, cuboids, and other polyhedrons in three dimensions.
- Vector format often used to represent map data.
  - Roads can be considered as two-dimensional and represented by lines and curves.

# Applications of Geographic Data

- Examples of geographic data
  - map data for vehicle navigation
  - distribution network information for power, telephones, water supply, and sewage
- Vehicle navigation systems store information about roads and services for the use of drivers:
  - **Spatial data:** e.g., road/restaurant/gas-station coordinates
  - **Non-spatial data:** e.g., one-way streets, speed limits, traffic congestion
- Global Positioning System (GPS)

# Spatial Queries

- Nearness queries request objects that lie near a specified location.
- Nearest neighbor queries, given a point or an object, find the nearest object that satisfies given conditions.
- Region queries deal with spatial regions. e.g., ask for objects that lie partially or fully inside a specified region.
- Queries that compute intersections or unions of regions.

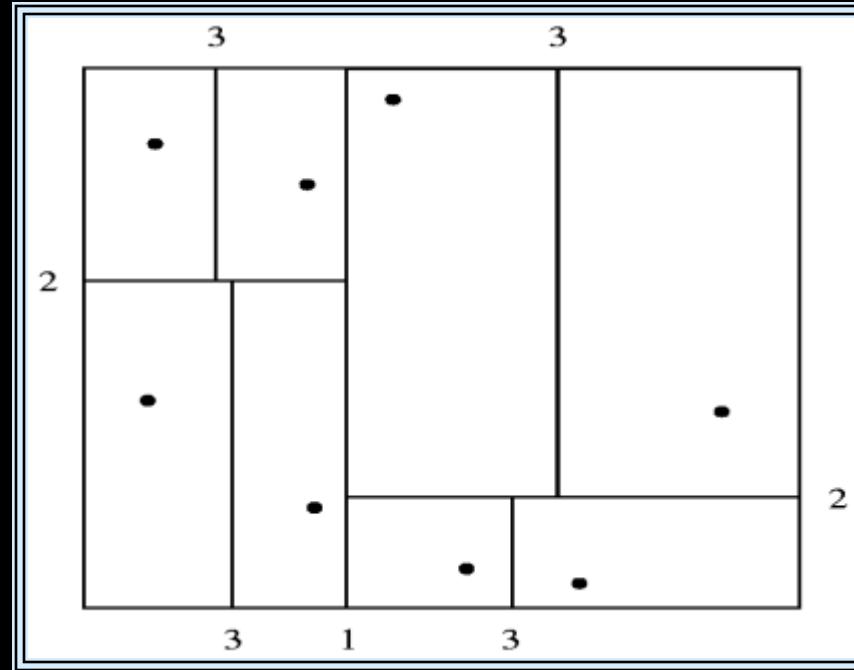
# Spatial Queries (Cont.)

- Spatial data is typically queried using a graphical query language; results are also displayed in a graphical manner.
- Graphical interface constitutes the front-end
- Extensions of SQL with abstract data types, such as lines, polygons and bit maps, have been proposed to interface with back-end.
  - allows relational databases to store and retrieve spatial information

# Indexing of Spatial Data

- k-d tree – early structure used for indexing in multiple dimensions.
- Each level of a *k-d* tree partitions the space into two.
  - choose one dimension for partitioning at the root level of the tree.
  - choose another dimensions for partitioning in nodes at the next level and so on, cycling through the dimensions.
- In each node, approximately half of the points stored in the sub-tree fall on one side and half on the other.

# Division of Space by a $k$ - $d$ Tree



- Each line in the figure (other than the outside box) corresponds to a node in the  $k$ - $d$  tree

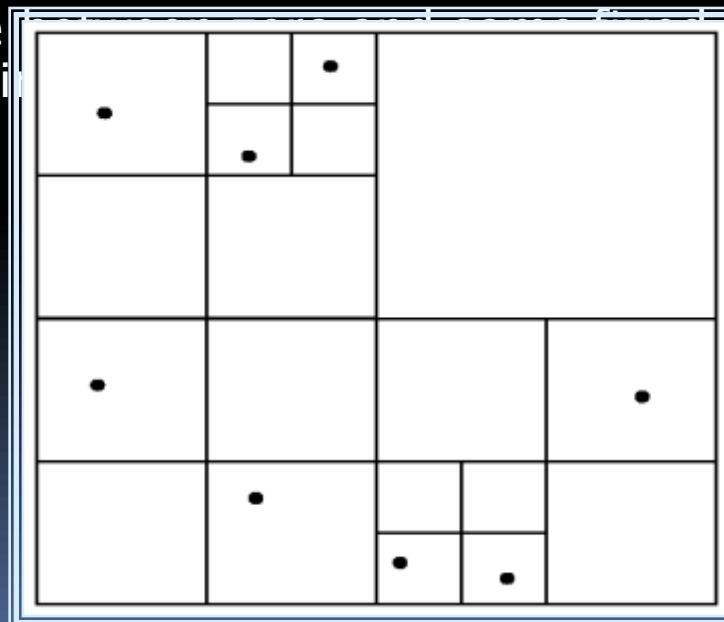
- the maximum number of points in a leaf node has been set to 1.

- The numbering of the lines in the figure indicates the level of the tree

# Division of Space by Quadtrees

## Quadtrees

- Each node of a quadtree is associated with a rectangular region of space; the top node is associated with the entire target space.
- Each non-leaf nodes divides its region into four equal sized quadrants
  - correspondingly each such node has four child nodes corresponding to the four quadrants and so on
- Leaf nodes have maximum number of points (set to 1 in this diagram)



# Quadtrees (Cont.)

- PR quadtree: stores points; space is divided based on regions, rather than on the actual set of points stored.
- Region quadtrees store array (raster) information.
  - A node is a leaf node if all the array values in the region that it covers are the same. Otherwise, it is subdivided further into four children of equal area, and is therefore an internal node.
  - Each node corresponds to a sub-array of values.
  - The sub-arrays corresponding to leaves either contain just a single array element, or have multiple array

# R-Trees

- R-trees are a N-dimensional extension of B<sup>+</sup>-trees, useful for indexing sets of rectangles and other polygons.
- Supported in many modern database systems, along with variants like R<sup>+</sup> - trees and R<sup>\*</sup>-trees.
- Basic idea: generalize the notion of a one-dimensional interval associated with each B+ -tree node to an N-dimensional interval, that is, an N-dimensional rectangle.

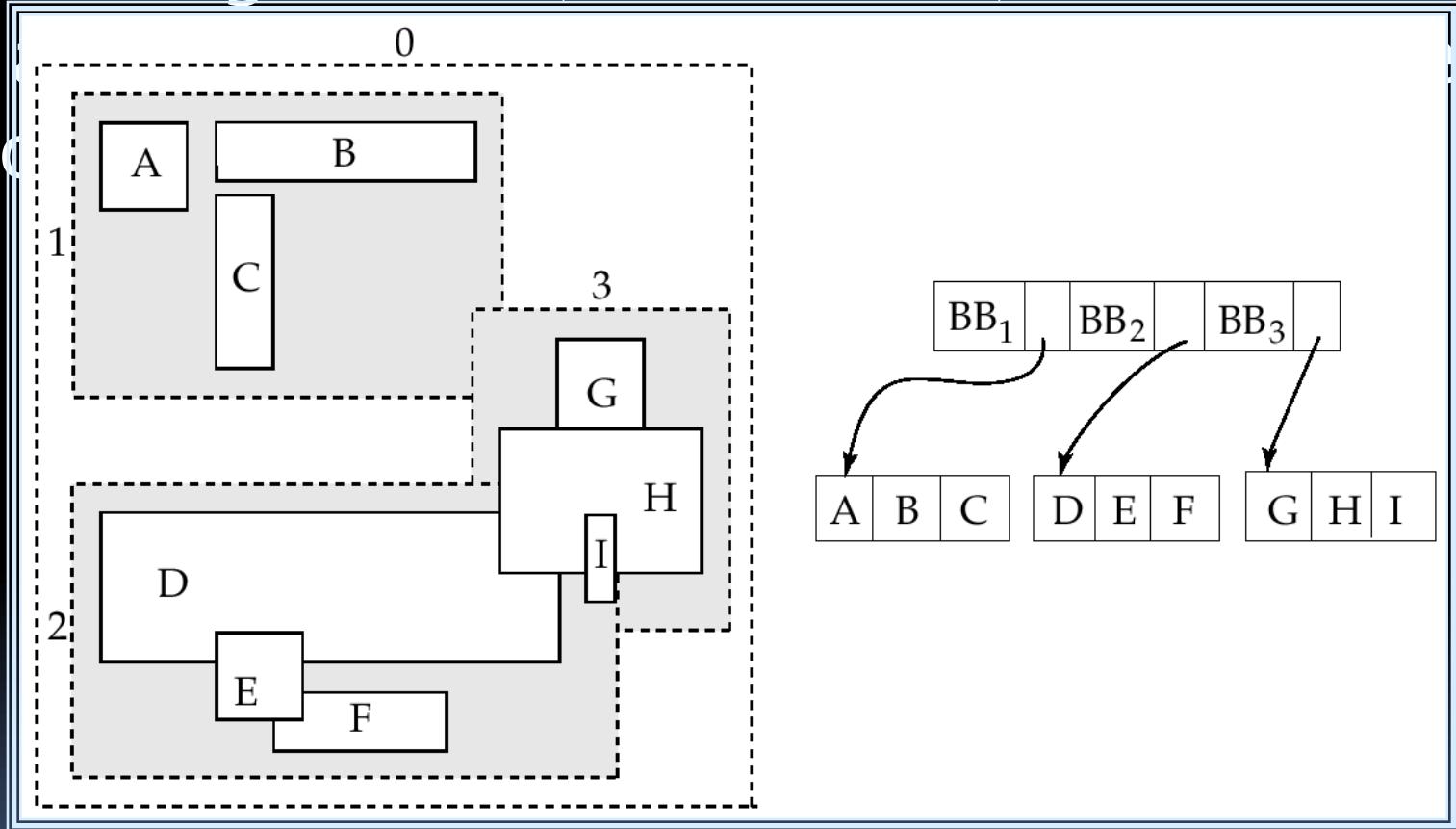
# R Trees (Cont.)

- A rectangular bounding box is associated with each tree node.
  - Bounding box of a leaf node is a minimum sized rectangle that contains all the rectangles/polygons associated with the leaf node.
  - The bounding box associated with a non-leaf node contains the bounding box associated with all its children.
  - Bounding box of a node serves as its key in its parent node (if any)
  - *Bounding boxes of children of a node are allowed to overlap*

A polygon is stored only in one node

# Example R-Tree

- A set of rectangles (solid line) and the bounding boxes (dashed line) of the nodes of the R-tree is shown.



# Search in R-Trees

- To find data items (rectangles/polygons) intersecting (overlaps) a given query point/region, do the following, starting from the root node:
  - If the node is a leaf node, output the data items whose keys intersect the given query point/region.
  - Else, for each child of the current node whose bounding box overlaps the query point/region, recursively search the child
- Can be very inefficient in worst case since multiple paths may need to be

# Insertion in R-Trees

- To insert a data item:
  - Find a leaf to store it, and add it to the leaf
    - To find leaf, follow a child (if any) whose bounding box contains bounding box of data item, else child whose overlap with data item bounding box is maximum
  - Handle overflows by splits (as in B+ - trees)
    - Split procedure is different though (see below)
  - Adjust bounding boxes starting from the leaf upwards

# Splitting an R-Tree Node

- Quadratic split divides the entries in a node into two new nodes as follows
  1. Find pair of entries with “maximum separation”
    - that is, the pair such that the bounding box of the two would have the maximum wasted space (area of bounding box - sum of areas of two entries)
  2. Place these entries in two new nodes
  3. Repeatedly find the entry with “maximum preference” for one of the two new nodes, and assign the entry to that node
    - ❖ Preference of an entry to a node is the increase in area of bounding box if the entry is added to the *other* node

# Deleting in R-Trees

- Deletion of an entry in an R-tree done much like a  $B^+$ -tree deletion.
  - In case of underfull node, borrow entries from a sibling if possible, else merging sibling nodes
  - Alternative approach removes all entries from the underfull node, deletes the node, then reinserts all entries

# MULTIMEDIA DATABASES

# Multimedia Databases

- To provide such database functions as indexing and consistency, it is desirable to store multimedia data in a database
  - rather than storing them outside the database, in a file system
- The database must handle large object representation.
- Similarity-based retrieval must be provided by special index structures.
- Must provide guaranteed steady

# Multimedia Data Formats

- Store and transmit multimedia data in compressed form
  - JPEG and GIF the most widely used formats for image data.
  - MPEG standard for video data use commonalities among a sequence of frames to achieve a greater degree of compression.
- MPEG-1 quality comparable to VHS video tape.
  - stores a minute of 30-frame-per-second video and audio in approximately 12.5 MB

# Continuous-Media Data

- Most important types are video and audio data.
- Characterized by high data volumes and real-time information-delivery requirements.
  - Data must be delivered sufficiently fast that there are no gaps in the audio or video.
  - Data must be delivered at a rate that does not cause overflow of system buffers.
  - Synchronization among distinct data streams must be maintained
    - video of a person speaking must show lips

# Video Servers

- Video-on-demand systems deliver video from central video servers, across a network, to terminals
  - Must guarantee end-to-end delivery rates
- Current video-on-demand servers are based on file systems; existing database systems do not meet real-time response requirements.
- Multimedia data are stored on several disks (RAID configuration), or on tertiary storage for less frequently accessed data.

# Similarity-Based Retrieval

Examples of similarity based retrieval

- Pictorial data: Two pictures or images that are slightly different as represented in the database may be considered the same by a user.
  - E.g., identify similar designs for registering a new trademark.
- Audio data: Speech-based user interfaces allow the user to give a command or identify a data item by speaking.
  - E.g., test user input against stored

# MOBILITY

# Mobile Computing Environments

- A mobile computing environment consists of mobile computers, referred to as mobile hosts, and a wired network of computers.
- Mobile host may be able to communicate with wired network through a *wireless digital communication network*
  - Wireless local-area networks (within a building)
    - E.g. Avaya's Orinoco Wireless LAN
  - Wide areas networks
    - Cellular digital packet networks

# Mobile Computing Environments (Cont.)

- A model for mobile communication
  - Mobile hosts communicate to the wired network via computers referred to as **mobile support (or base) stations**.
  - Each mobile support station manages those mobile hosts within its **cell**.
  - When mobile hosts move between cells, there is a **handoff** of control from one mobile support station to another.
- Direct communication, without going through a mobile support station is also possible between nearby mobile

# Database Issues in Mobile Computing

- New issues for query optimization.
  - Connection time charges and number of bytes transmitted
  - Energy (battery power) is a scarce resource and its usage must be minimized
- Mobile user's locations may be a parameter of the query
  - GIS queries
  - Techniques to track locations of large numbers of mobile hosts
- Broadcast data can enable any number of clients to receive the same data at no extra cost
  - leads to interesting querying and data caching issues.
- Users may need to be able to perform database updates even while the mobile computer is disconnected.

# Routing and Query Processing

- Must consider these competing costs:
  - User time.
  - Communication cost
    - Connection time – used to assign monetary charges in some cellular systems.
    - Number of bytes, or packets, transferred – used to compute charges in digital cellular systems
    - Time-of-day based charges – vary based on peak or off-peak periods
  - Energy – optimize use of battery power by minimizing reception and transmission of data

# Broadcast Data

- Mobile support stations can broadcast frequently-requested data
  - Allows mobile hosts to wait for needed data, rather than having to consume energy transmitting a request
  - Supports mobile hosts without transmission capability
- A mobile host may optimize energy costs by determining if a query can be answered using only cached data
  - If not then must either;
    - Wait for the data to be broadcast
    - Transmit a request for data and must know when the relevant data will be broadcast.
- Broadcast data may be transmitted according to a fixed schedule or a changeable schedule.

# Disconnectivity and Consistency

- A mobile host may remain in operation during periods of disconnection.
- Problems created if the user of the mobile host issues queries and updates on data that resides or is cached locally:
  - **Recoverability:** Updates entered on a disconnected machine may be lost if the mobile host fails. Since the mobile host represents a single point of failure, stable storage cannot be simulated well.

# Mobile Updates

- Partitioning via disconnection is the normal mode of operation in mobile computing.
- For data updated by only one mobile host, simple to propagate update when mobile host reconnects
  - in other cases data may become invalid and updates may conflict.
- When data are updated by other computers, **invalidation reports** inform a reconnected mobile host of out-of-date cache entries
  - however, mobile host may miss a report.
- **Version-numbering**-based schemes guarantee only that if two hosts independently update the same version of a document, the clash will be detected eventually, when the hosts exchange information either directly or through a common host.
  - More on this shortly
- Automatic reconciliation of inconsistent copies of data is

# Detecting Inconsistent Updates

- Version vector scheme used to detect inconsistent updates to documents at different hosts (sites).
- Copies of document  $d$  at hosts  $i$  and  $j$  are **inconsistent** if
  1. the copy of document  $d$  at  $i$  contains updates performed by host  $k$  that have not been propagated to host  $j$  ( $k$  may be the same as  $i$ ), and
  2. the copy of  $d$  at  $j$  contains updates performed by host  $/$  that have not been propagated to host  $i$  ( $/$  may be the same as  $j$ )

# Detecting Inconsistent Updates (Cont.)

- When two hosts  $i$  and  $j$  connect to each other they check if the copies of all documents  $d$  that they share are consistent:
  1. If the version vectors are the same on both hosts (that is, for each  $k$ ,  $V_{d,i}[k] = V_{d,j}[k]$ ) then the copies of  $d$  are identical.
  2. If, for each  $k$ ,  $V_{d,i}[k] \leq V_{d,j}[k]$ , and the version vectors are not identical, then the copy of document  $d$  at host  $i$  is older than the one at host  $j$ 
    - That is, the copy of document  $d$  at host  $i$  was obtained by one or more

# Handling Inconsistent Updates

- Dealing with inconsistent updates is hard in general. Manual intervention often required to merge the updates.
- Version vector schemes
  - were developed to deal with failures in a distributed file system, where inconsistencies are rare.
  - are used to maintain a unified file system between a fixed host and a mobile computer, where updates at the two hosts have to be merged periodically.
    - Also used for similar purposes in groupware systems.
    - are used in database systems where

# END OF CHAPTER

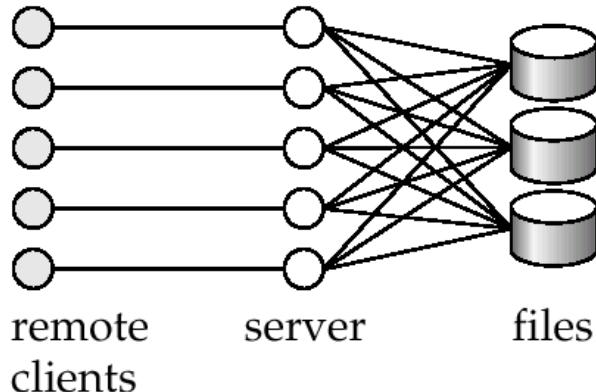
# Chapter 24: Advanced Transaction Processing

- Transaction-Processing Monitors
- Transactional Workflows
- High-Performance Transaction Systems
  - Main memory databases
  - Real-Time Transaction Systems
- Long-Duration Transactions
- Transaction management in multidatabase systems

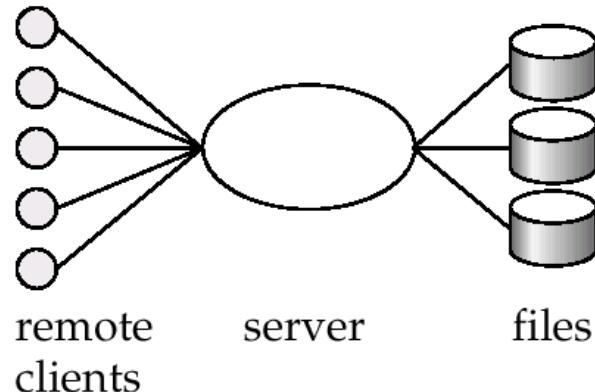
# Transaction Processing Monitors

- TP monitors initially developed as multithreaded servers to support large numbers of terminals from a single process.
- Provide infrastructure for building and administering complex transaction processing systems with a large number of clients and multiple servers.
- Provide services such as:
  - Presentation facilities to simplify creating user interfaces
  - Persistent queuing of client requests and

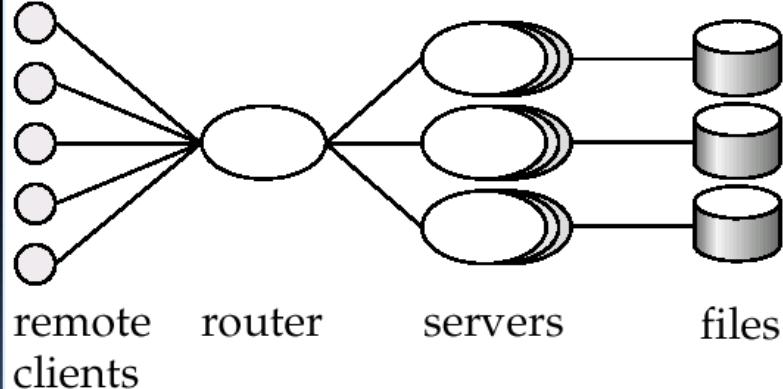
# TP Monitor Architectures



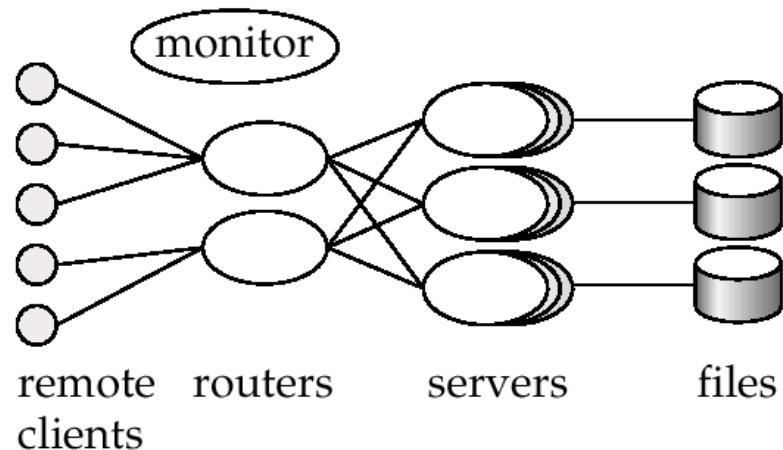
(a) Process-per-client model



(b) Single-process model



(c) Many-server, single-router model



(d) Many-server, many-router model

# TP Monitor Architectures (Cont.)

■ Process per client model – instead of individual login session per terminal, server process communicates with the terminal, handles authentication, and executes actions.

- Memory requirements are high
- Multitasking– high CPU overhead for context switching between processes

■ Single process model – all remote terminals connect to a single server process.

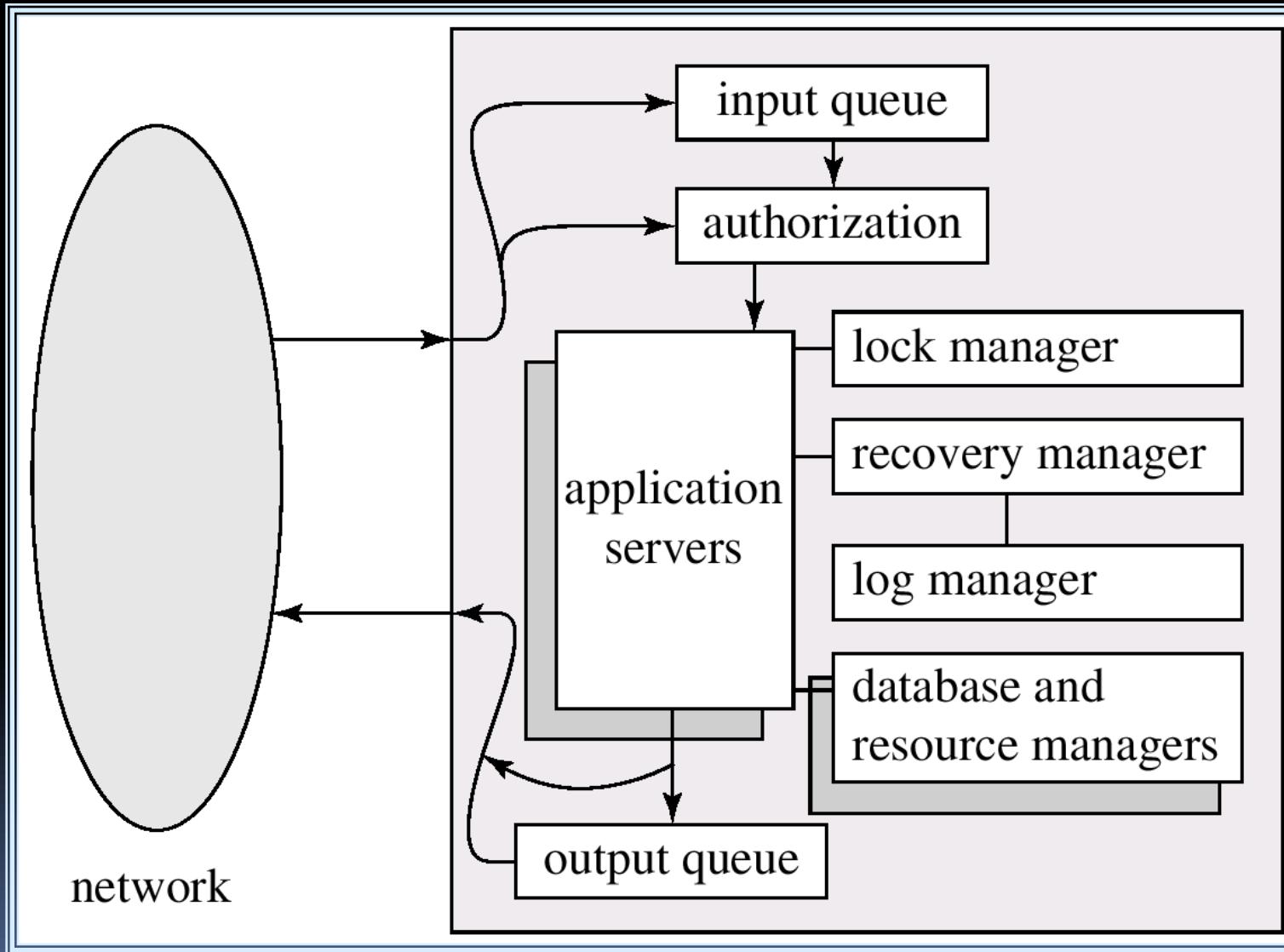
- Used in client-server environments
- Server process is multi-threaded· low

# TP Monitor Architectures (Cont.)

- Many-server single-router model
  - multiple application server processes access a common database; clients communicate with the application through a single communication process that routes requests.
    - Independent server processes for multiple applications
    - Multithread server process
    - Run on parallel or distributed database

■ Many server many-router model –

# Detailed Structure of a TP Monitor



# Detailed Structure of a TP Monitor

- Queue manager handles incoming messages
- Some queue managers provide persistent or durable message queueing contents of queue are safe even if systems fails.
- Durable queueing of outgoing messages is important
  - application server writes message to durable queue as part of a transaction
  - once the transaction commits, the TP monitor guarantees message is eventually delivered, regardless of

# Application Coordination Using TP Monitors

- A TP monitor treats each subsystem as a resource manager that provides transactional access to some set of resources.
- The interface between the TP monitor and the resource manager is defined by a set of transaction primitives
- The resource manager interface is defined by the X/Open Distributed Transaction Processing standard.
- TP monitor systems provide a transactional remote procedure call (transactional RPC) interface to support distributed transaction processing.

# WORKFLOW SYSTEMS

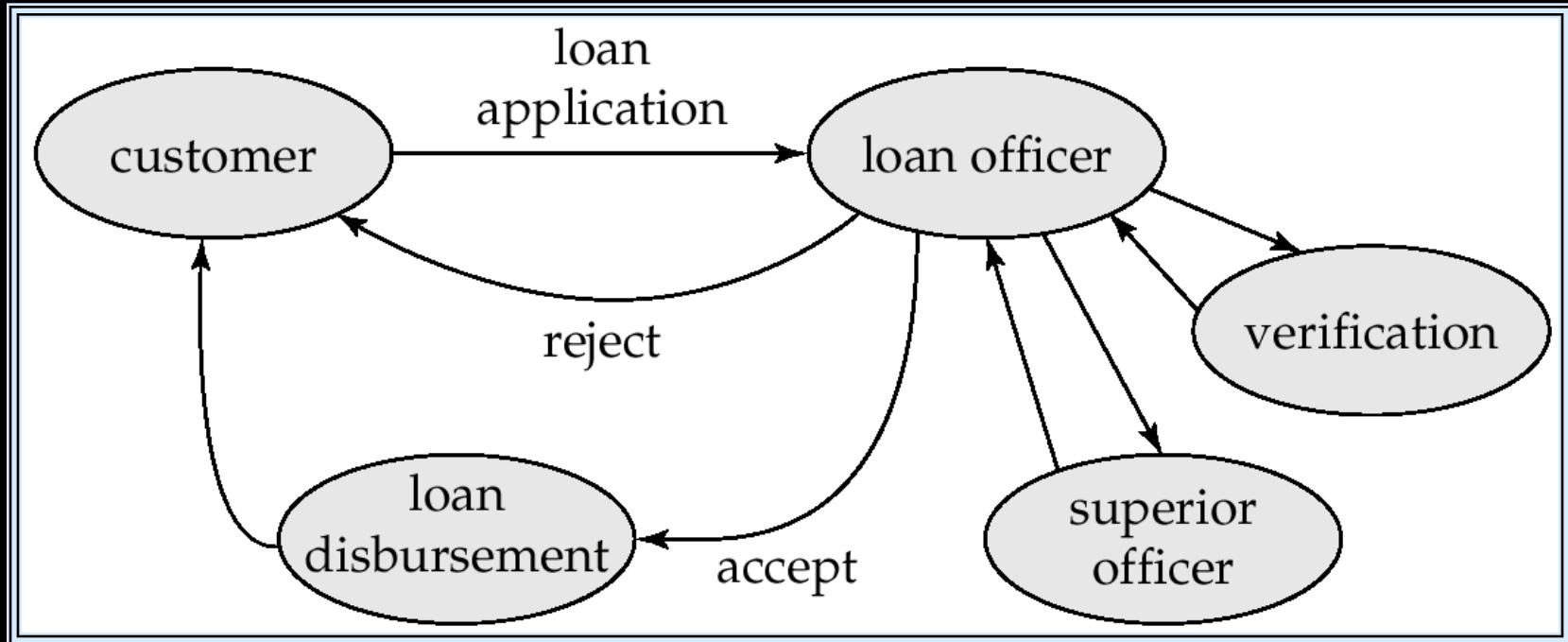
# Transactional Workflows

- Workflows are activities that involve the coordinated execution of multiple tasks performed by different processing entities.
- With the growth of networks, and the existence of multiple autonomous database systems, workflows provide a convenient way of carrying out tasks that involve multiple systems.
- Example of a workflow delivery of an email message, which goes through several mails systems to reach destination

# Examples of Workflows

Workflow application	Typical task	Typical processing entity
electronic-mail routing	electronic-mail message	mailers
loan processing	form processing	humans, application software
purchase-order processing	form processing	humans, application software, DBMSs

# Loan Processing Workflow



- In the past, workflows were handled by creating and forwarding paper forms

- Computerized workflows aim to automate many of the tasks. But the

# Transactional Workflows

- Must address following issues to computerize a workflow.
  - Specification of workflows – detailing the tasks that must be carried out and defining the execution requirements.
  - Execution of workflows – execute transactions specified in the workflow while also providing traditional database safeguards related to the correctness of computations, data integrity, and durability.
  - E.g.: Loan application should not get lost even if system fails.
- Extend transaction concepts to the context of workflows

# Workflow Specification

- Static specification of task coordination:
  - Tasks and dependencies among them are defined before the execution of the workflow starts.
  - Can establish preconditions for execution of each task: tasks are executed only when their preconditions are satisfied.
  - Defined preconditions through dependencies:
    - Execution states of other tasks.  
“task  $t_i$  cannot start until task  $t_j$  has ended”
    - Output values of other tasks.  
“task  $t_i$  can start if task  $t_j$  returns a value”<sup>155</sup>



# Workflow Specification (Cont.)

- Dynamic task coordination  
E.g. Electronic mail routing system in which the text to be schedule for a given mail message depends on the destination address and on which intermediate routers are functioning.

# Failure-Automaticity Requirements

- Usual ACID transactional requirements are too strong/unimplementable for workflow applications.
- However, workflows must satisfy some limited transactional properties that guarantee a process is not left in an inconsistent state.
- Acceptable termination states – every execution of a workflow will terminate in a state that satisfies the failure-atomicity requirements defined by the designer.

# Execution of Workflows

Workflow management systems include:

- Scheduler – program that processes workflows by submitting various tasks for execution, monitoring various events, and evaluation conditions related to intertask dependencies
- Task agents – control the execution of a task by a processing entity.
- Mechanism to query to state of the workflow system.

# Workflow Management System Architectures

- Centralized – a single scheduler schedules the tasks for all concurrently executing workflows.
  - used in workflow systems where the data is stored in a central database.
  - easier to keep track of the state of a workflow.
- Partially distributed – has one (instance of a ) scheduler for each workflow.
- Fully distributed – has no scheduler, but the task agents coordinate their execution by communicating with each other.

# Workflow Scheduler

- Ideally scheduler should execute a workflow only after ensuring that it will terminate in an acceptable state.
- Consider a workflow consisting of two tasks  $S_1$  and  $S_2$ . Let the failure-atomicity requirement be that either both or neither of the subtransactions should be committed.
  - Suppose systems executing  $S_1$  and  $S_2$  do not provide prepared-to-commit states and  $S_1$  or  $S_2$  do not have compensating transactions.

# Recovery of a Workflow

- Ensure that if a failure occurs in any of the workflow-processing components, the workflow eventually reaches an acceptable termination state.
- Failure-recovery routines need to restore the state information of the scheduler at the time of failure, including the information about the execution states of each task. Log status information on stable storage.

# Recovery of a Workflow (Cont.)

- Persistent messages: messages are stored in permanent message queue and therefore not lost in case of failure.
  - Described in detail in Chapter 19 (Distributed Databases)
- Before an agent commits, it writes to the persistent message queue whatever messages need to be sent out.
- The persistent message system must make sure the messages get delivered eventually if and only if the transaction commits.
- The message system needs to resend a message when the site recovers if the

# HIGH PERFORMANCE TRANSACTION SYSTEMS

# High-Performance Transaction Systems

High-performance hardware and parallelism help improve the rate of transaction processing, but are insufficient to obtain high performance:

- Disk I/O is a bottleneck — I/O time (10 milliseconds) has not decreased at a rate comparable to the increase in processor speeds.
- Parallel transactions may attempt to read or write the same data item, resulting in data conflicts that reduce effective parallelism

# Main-Memory Database

- Commercial 64-bit systems can support main memories of tens of gigabytes.
- Memory resident data allows faster processing of transactions.
- Disk-related limitations:
  - Logging is a bottleneck when transaction rate is high.
  - Use group-commit to reduce number of output operations (Will study two slides ahead.)
  - If the update rate for modified buffer blocks is high, the disk data-transfer

# Main-Memory Database Optimizations

- To reduce space overheads, main-memory databases can use structures with pointers crossing multiple pages. In disk databases, the I/O cost to traverse multiple pages would be excessively high.
- No need to pin buffer pages in memory before data are accessed, since buffer pages will never be replaced.
- Design query-processing techniques to minimize space overhead – avoid exceeding main

# Group Commit

- Idea: Instead of performing output of log records to stable storage as soon as a transaction is ready to commit, wait until
  - log buffer block is full, or
  - a transaction has been waiting sufficiently long after being ready to commit
- Results in fewer output operations per committed transaction, and correspondingly a higher throughput.
- However, commits are delayed until a sufficiently large group of transactions are ready to commit, or a

- In systems with real-time constraints, correctness of execution involves both database consistency and the satisfaction of deadlines.
  - **Hard deadline** – Serious problems may occur if task is not completed within deadline
  - **Firm deadline** – The task has zero value if it completed after the deadline.
  - **Soft deadline** – The task has diminishing value if it is completed after the deadline.
- The wide variance of execution times for read and write operations on disks complicates the transaction management problem for time-constrained systems

# Long Duration Transactions

- Traditional concurrency control techniques do not work well when user interaction is required:
- Long duration: Design edit sessions are very long
  - Exposure of uncommitted data: E.g., partial update to a design
  - Subtasks: support partial rollback
  - Recoverability: on crash state should be restored even for yet-to-be committed data, so user

# Long-Duration Transactions

- Represent as a nested transaction
  - atomic database operations (read/write) at a lowest level.
- If transaction fails, only active short-duration transactions abort.
- Active long-duration transactions resume once any short duration transactions have recovered.
- The efficient management of long-duration waits, and the possibility of aborts.
- Need alternatives to waits and aborts to continue a transaction.

# Concurrency Control

- Correctness without serializability:
  - Correctness depends on the specific consistency constraints for the databases.
  - Correctness depends on the properties of operations performed by each transaction.
- Use database consistency constraints as to split the database into subdatabases on which concurrency can be managed separately.
- Treat some operations besides

# Concurrency Control (Cont.)

non-conflict-  
serializable schedule  
that preserves the  
sum of A + B

$T_1$	$T_2$
<b>read(A)</b>	
$A := A - 50$	
<b>write(A)</b>	
	<b>read(B)</b>
	$B := B - 10$
	<b>write(B)</b>
<b>read(B)</b>	
$B := B + 50$	
<b>write(B)</b>	
	<b>read(A)</b>
	$A := A + 10$
	<b>write(A)</b>

# Nested and Multilevel Transactions

- A nested or multilevel transaction  $T$  is represented by a set  $T = \{t_1, t_2, \dots, t_n\}$  of subtransactions and a partial order  $P$  on  $T$ .
- A subtransaction  $t_i$  in  $T$  may abort without forcing  $T$  to abort.
- Instead,  $T$  may either restart  $t_i$ , or simply choose not to run  $t_i$ .
- If  $t_i$  commits, this action does not make  $t_i$  permanent (unlike the situation in Chapter 15). Instead,  $t_i$  commits to  $T$ , and may still abort (or be restarted) if  $T$  is rolled back.

# Nested and Multilevel Transactions (Cont.)

- Subtransactions can themselves be nested/multilevel transactions.
  - Lowest level of nesting: standard read and write operations.
- Nesting can create higher-level operations that may enhance concurrency.
- Types of nested/ multilevel transactions:
  - **Multilevel transaction:** subtransaction of  $T$  is permitted to release locks on completion.
  - **Saga:** multilevel long-duration transaction.
  - **Nested transaction:** locks held by a subtransaction  $t_i$  of  $T$  are automatically assigned to  $T$  on completion of  $t_i$ .

# Example of Nesting

Rewrite transaction  $T_1$  using subtransactions  $T_a$  and  $T_b$  that perform increment or decrement operations:

- $T_1$  consists of
  - $T_{1,1}$ , which subtracts 50 from  $A$
  - $T_{1,2}$ , which adds 50 to  $B$

Rewrite transaction  $T_2$  using subtransactions  $T_c$  and  $T_d$  that perform increment or decrement operations:

- $T_2$  consists of
  - $T_{2,1}$ , which subtracts 10 from  $B$

# Compensating Transactions

- Alternative to undo operation; compensating transactions deal with the problem of cascading rollbacks.
- Instead of undoing all changes made by the failed transaction, action is taken to “compensate” for the failure.
- Consider a long-duration transaction  $T_i$  representing a travel reservation, with subtransactions  $T_{i,1}$ , which makes airline reservations,  $T_{i,2}$  which reserves rental cars, and  $T_{i,3}$  which reserves a hotel room.
  - Hotel cancels the reservation.
  - Instead of undoing all of  $T_i$ , the failure of

# Implementation Issues

- For long-duration transactions to survive system crashes, we must log not only changes to the database, but also changes to internal system data pertaining to these transactions.
- Logging of updates is made more complex by physically large data items (CAD design, document text); undesirable to store both old and new values.
- Two approaches to reducing the overhead of ensuring the recoverability of large data items:
  - 157  
6

# Transaction Management in Multidatabase Systems

- Transaction management is complicated in multidatabase systems because of the assumption of autonomy
  - **Global 2PL** -each local site uses a strict 2PL (locks are released at the end); locks set as a result of a global transaction are released only when that transaction reaches the end.
    - Guarantees global serializability
  - Due to autonomy requirements, sites cannot cooperate and execute a common

# Transaction Management

- Local transactions are executed by each local DBMS, outside of the MDBS system control.
- Global transactions are executed under multidatabase control.
- Local autonomy – local DBMSs cannot communicate directly to synchronize global transaction execution and the multidatabase has no control over local transaction execution.
  - local concurrency control scheme needed to ensure that DBMS's schedule is valid

# Two-Level Serializability

- DBMS ensures local serializability among its local transactions, including those that are part of a global transaction.
- The multidatabase ensures serializability among global transactions alone- *ignoring the orderings induced by local transactions.*
- 2LSR does not ensure global serializability, however, it can fulfill requirements for strong correctness.

# Two-Level Serializability (Cont.)

- Local-read protocol : Local transactions have read access to global data; disallows all access to local data by global transactions.
- A transaction has a value dependency if the value that it writes to a data item at one site depends on a value that it read for a data item on another site.
- For strong correctness: No transaction may have a value dependency.
- Global-read-write/local-read protocol;  
Local transactions have read access to

# Global Serializability

- Even if no information is available concerning the structure of the various concurrency control schemes, a very restrictive protocol that ensures serializability is available.
- Transaction-graph : a graph with vertices being global transaction names and site names.
- An undirected edge  $(T_i, S_k)$  exists if  $T_i$  is active at site  $S_k$ .
- Global serializability is assured if transaction-graph contains no

# Ensuring Global Serializability

- Each site  $S_i$  has a special data item, called ticket
- Every transaction  $T_j$  that runs at site  $S_k$  writes to the ticket at site  $S_i$
- Ensures global transactions are serialized at each site, regardless of local concurrency control method, so long as the method guarantees local serializability
- Global transaction manager decides serial ordering of global transactions by controlling order in which tickets

END OF CHAPTER

# Weak Levels Consistency

- Use alternative notions of consistency that do not ensure serializability, to improve performance.
- Degree-two consistency avoids cascading aborts without necessarily ensuring serializability.
  - Unlike two-phase locking, S-locks may be released at any time, and locks may be acquired at any time.
  - X-locks be released until the transaction either commits or aborts.

# Example Schedule with Degree-Two Consistency

## Consistency

Nonserializable schedule with degree-two consistency (Figure 20.5) where  $T_3$  reads the value if  $Q$  before and after that value is written<sup>7</sup> by  $T_4$ .

$T_4$
<b>lock-S (Q)</b> <b>read (Q)</b> <b>unlock (Q)</b>
<b>lock-X (Q)</b> <b>read (Q)</b> <b>write (Q)</b> <b>unlock (Q)</b>

# Cursor Stability

- Form of degree-two consistency designed for programs written in general-purpose, record-oriented languages (e.g., Pascal, C, Cobol, PL/I, Fortran).
- Rather than locking the entire relation, cursor stability ensures that
  - The tuple that is currently being processed by the iteration is locked in shared mode.
  - Any modified tuples are locked in exclusive mode until the transaction