



FUNDAMENTALS OF DATABASE MANAGEMENT SYSTEMS

EEE | MECH
VII SEM



Prepared by: K Radhika
Assistant Professor
CSE

UNIT- I



Introduction to file and database systems: Database system structure, data models, introduction to network and hierarchical models, ER model, relational model.

Database



Data:

- Raw facts; building blocks of information
- Unprocessed information

Information:

- Data processed to reveal meaning

Database—shared, integrated computer structure that stores:

- End user data (raw facts)
- Metadata (data about data)

Database management system



DBMS (Database management system):

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

Advantages of the DBMS

End users have better access to more and better-managed data

- Promotes integrated view of organization's operations
- Probability of data inconsistency is greatly reduced
- Possible to produce quick answers to ad hoc queries

Database Applications



Database Applications:

- Banking: transactions
- Airlines: reservations, schedules
- Universities: registration, grades
- Sales: customers, products, purchases

Database Applications(contd.)



- Online retailers: order tracking, customized recommendations
- Manufacturing: production, inventory, orders, supply chain
- Human resources: employee records, salaries, tax deductions
- Databases can be very large.
- Databases touch all aspects of our lives

University Database Example



Application program examples

- Add new students, instructors, and courses
- Register students for courses, and generate class rosters
- Assign grades to students, compute grade point averages

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

Various Databases



Single-user:

- Supports only one user at a time

Desktop:

- Single-user database running on a personal computer

Multi-user:

- Supports multiple users at the same time

Various Databases(contd.)



Workgroup:

- Multi-user database that supports a small group of users or a single department

Enterprise:

- Multi-user database that supports a large group of users or an entire organization

Various Databases(contd.)

- Can be classified by location:
- Centralized:
 - Supports data located at a single site
- Distributed:
 - Supports data distributed across several sites

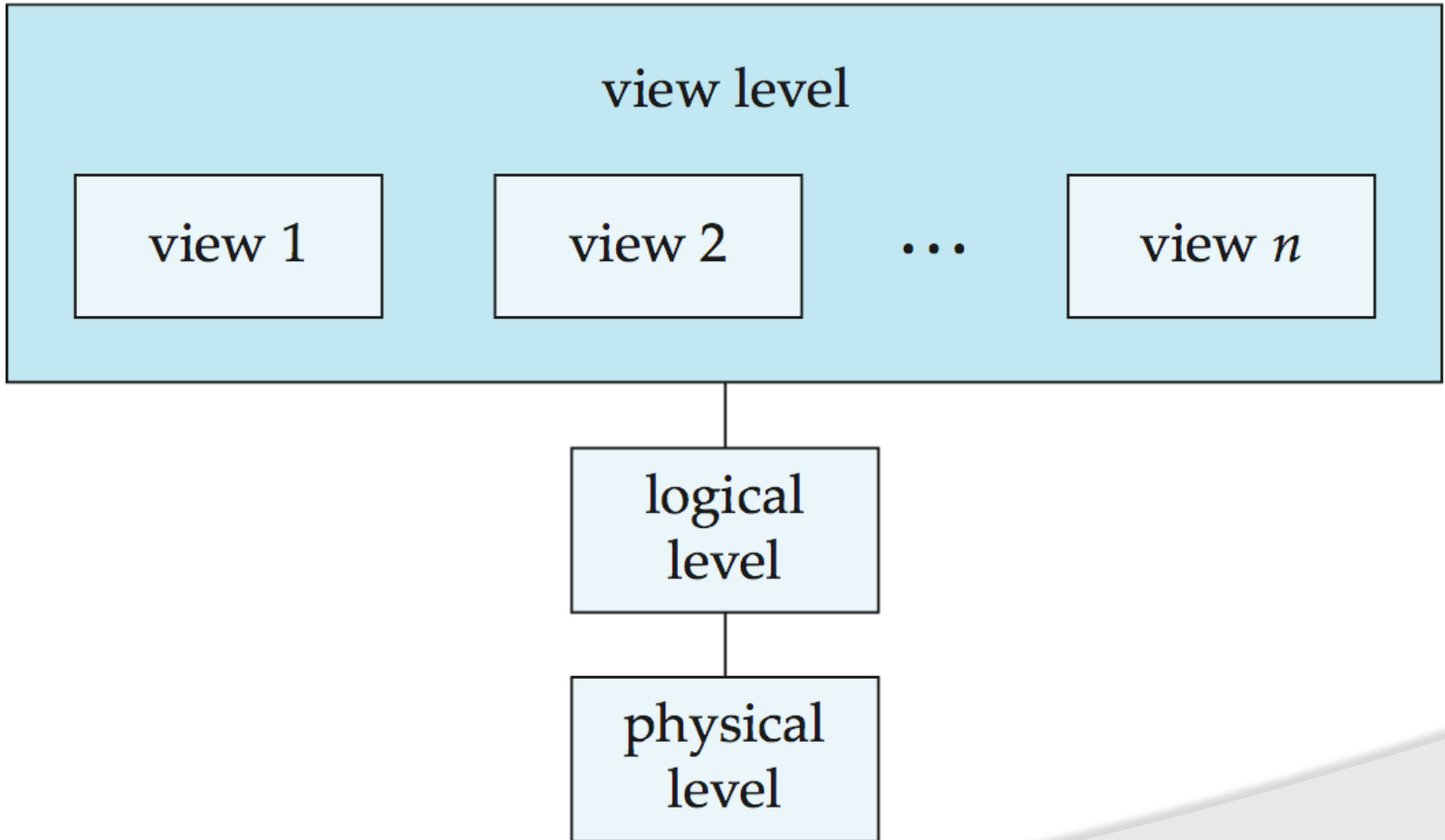
Levels of Abstraction



- Physical level: describes how a record (e.g., instructor) is stored.
- Logical level: describes data stored in database, and the relationships among the data.
 type instructor = record
 ID : string;
 name : string;
 dept_name : string;
 salary : integer;
- end;
- View level: application programs hide details of data types. Views can also hide information (such as an employee's salary) for security purposes.

View of Data

An architecture for a database system



Instances and Schemas

Similar to types and variables in programming languages

- **Logical Schema** – the overall logical structure of the database

Example: The database consists of information about a set of customers and accounts in a bank and the relationship between them

- Analogous to type information of a variable in a program
- **Physical schema**– the overall physical structure of the database
- **Instance** – the actual content of the database at a particular point in time
- Analogous to the value of a variable

Instances and Schemas(contd.)



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
- Relational model
- Entity-Relationship data model (mainly for database design)
- Object-based data models (Object-oriented and Object-relational)
- Semi structured data model (XML)
- Other older models:
 - Network model
 - Hierarchical model

Relational Model

All the data is stored in various tables.

Example of tabular data in the relational model

<i>ID</i>	<i>name</i>	<i>dept_name</i>	<i>salary</i>
22222	Einstein	Physics	95000
12121	Wu	Finance	90000
32343	El Said	History	60000
45565	Katz	Comp. Sci.	75000
98345	Kim	Elec. Eng.	80000
76766	Crick	Biology	72000
10101	Srinivasan	Comp. Sci.	65000
58583	Califieri	History	62000
83821	Brandt	Comp. Sci.	92000
15151	Mozart	Music	40000
33456	Gold	Physics	87000
76543	Singh	Finance	80000

(a) The *instructor* table

A Sample Relational Database

<i>ID</i>	<i>name</i>	<i>dept_name</i>	<i>salary</i>
22222	Einstein	Physics	95000
12121	Wu	Finance	90000
32343	El Said	History	60000
45565	Katz	Comp. Sci.	75000
98345	Kim	Elec. Eng.	80000
76766	Crick	Biology	72000
10101	Srinivasan	Comp. Sci.	65000
58583	Califieri	History	62000
83821	Brandt	Comp. Sci.	92000
15151	Mozart	Music	40000
33456	Gold	Physics	87000
76543	Singh	Finance	80000

(a) The *instructor* table

<i>dept_name</i>	<i>building</i>	<i>budget</i>
Comp. Sci.	Taylor	100000
Biology	Watson	90000
Elec. Eng.	Taylor	85000
Music	Packard	80000
Finance	Painter	120000
History	Painter	50000
Physics	Watson	70000

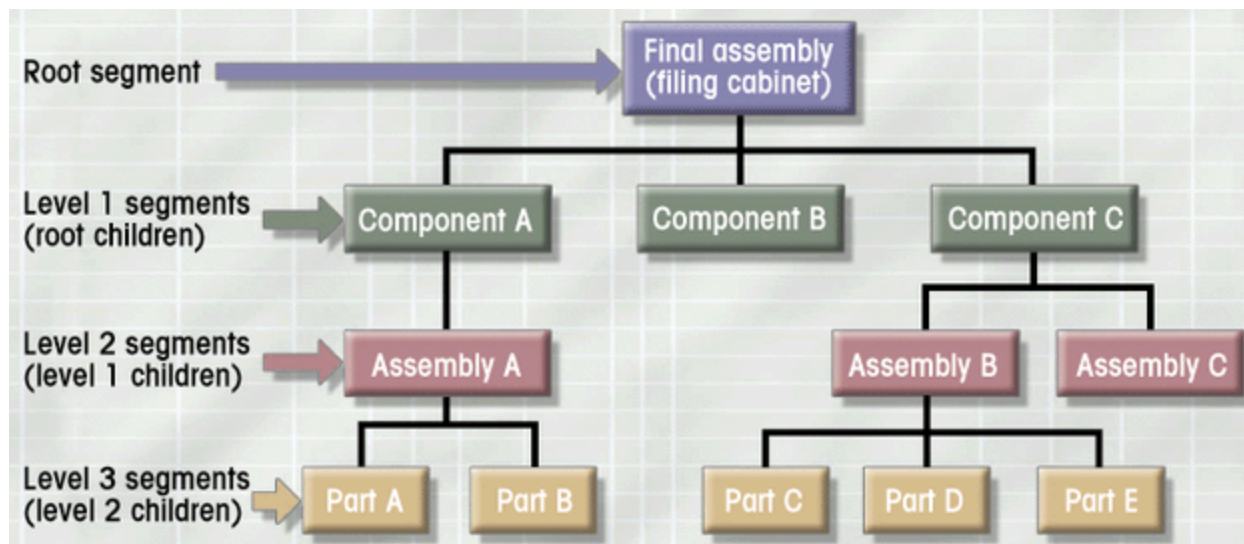
(b) The *department* table

Hierarchical model

Hierarchical Database Model

Assumes data relationships are hierarchical

- One-to-Many (1:M) relationships
- Each parent can have many children
- Each child has only one parent
- Logically represented by an upside down tree

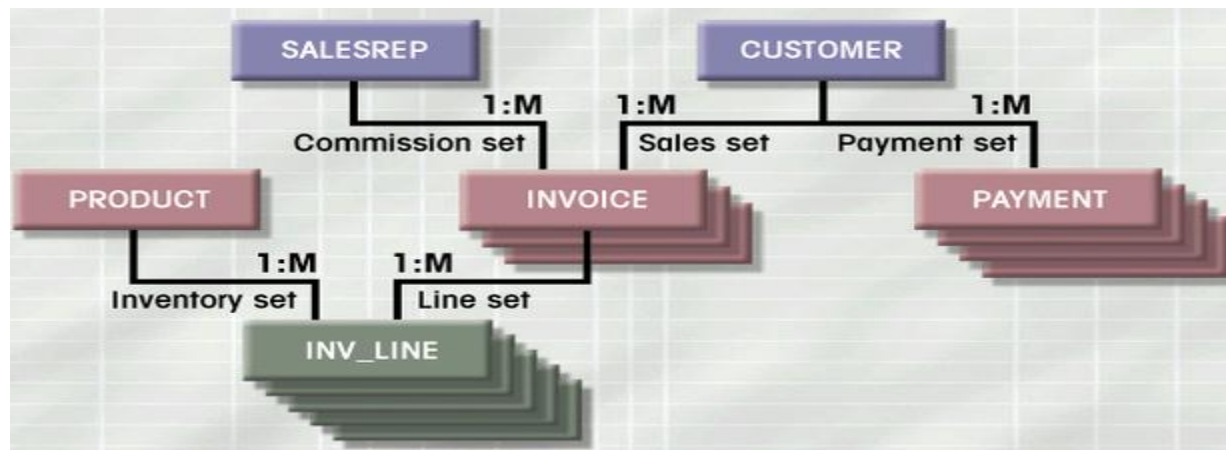


Network model

Network Database Model

Similar to Hierarchical Model

- Records linked by **pointers**
- Composed of sets
- Each set consists of owner (parent) and member (child)
- Many-to-Many (**M:N**) relationships representation
- Each owner can have multiple members (1:M)
- A member may have several owners



Entity Relationship Model

Entity Relationship (ER) Model

Based on **Entity, Attributes & Relationships**

- Entity is a **thing** about which data are to be collected and stored
e.g. EMPLOYEE
- Attributes are **characteristics** of the entity
e.g. SSN, last name, first name
- Relationships describe an **associations** between entities
i.e. 1:M, M:N, 1:1

Represented in an Entity Relationship Diagram (ERD)

Formalizes a way to describe relationships between groups of data

E-R Diagram:

A one-to-many (1:M) relationship: a PAINTER can paint many PAINTINGS; each PAINTING is painted by one PAINTER



A many-to-many (M:N) relationship: an EMPLOYEE can learn many SKILLS; each SKILL can be learned by many EMPLOYEES



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



- Entity
 - represented by a rectangle with its **name in capital** letters.
- Relationships
 - represented by an active or passive **verb inside the diamond** that connects the related entities.
- Connectivities
 - i.e., types of relationship
 - written next to each entity box.

Data Definition Language (DDL)



Specification notation for defining the database schema

Example: create table instructor (
 ID char(5),
 name varchar(20),
 dept_name varchar(20),
 salary numeric(8,2))

DDL compiler generates a set of table templates stored in a **data Dictionary**

- Data dictionary contains metadata (i.e., data about data)
- Database schema
- Integrity constraints
- Primary key (ID uniquely identifies instructors)
- Authorization
- Who can access what

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

- **Pure** – used for proving properties about computational power and for optimization

Relational Algebra - Tuple relational calculus & Domain relational calculus

- **Commercial** – used in commercial systems

SQL is the most widely used commercial language

Database Users and Administrators:



- **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** – Interact with the system without writing programs.

Database Administrator



Having central control over the system is called a 'database administrator (DBA)'.

The functions of DBA includes:

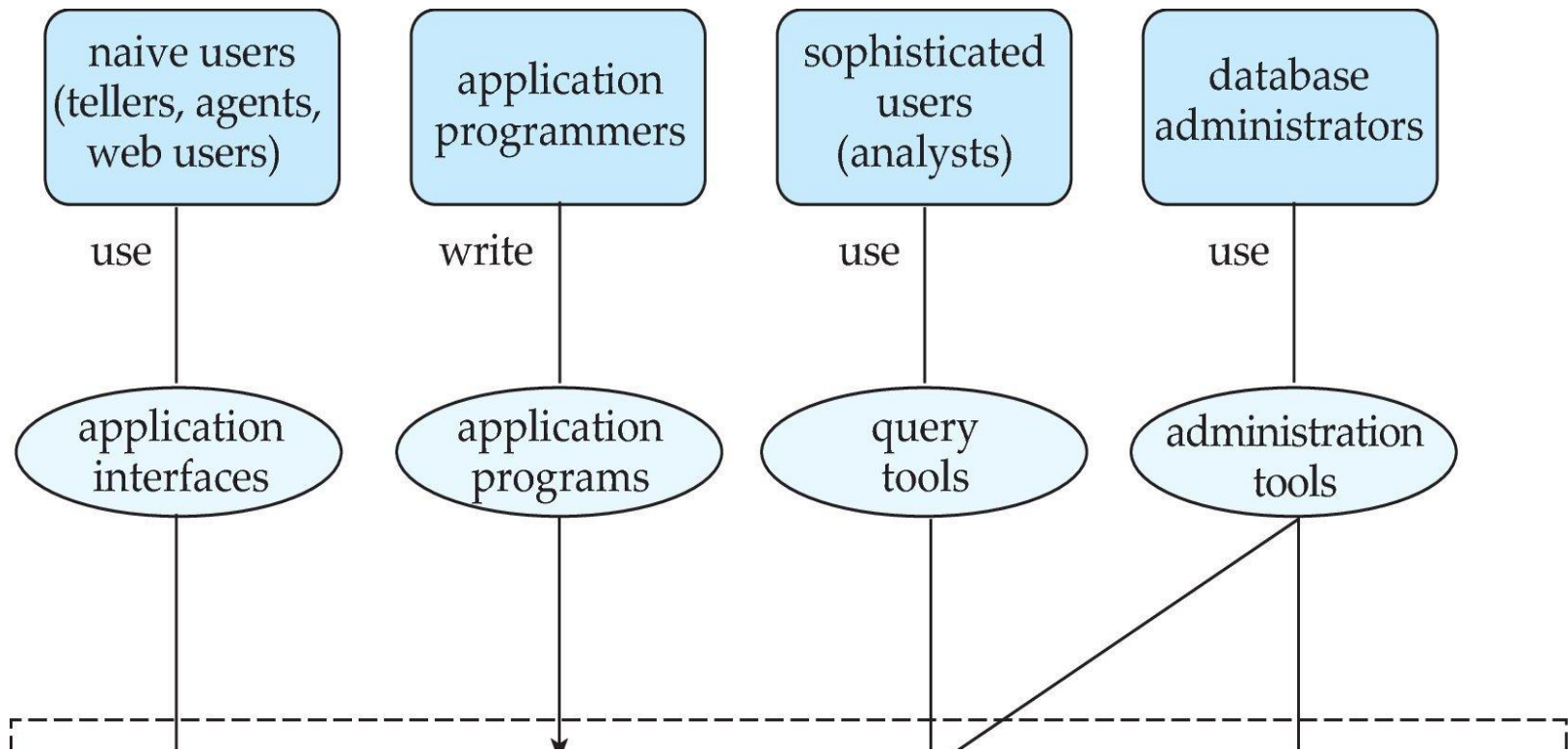
- **Schema Definition:** Creates the original database schema by executing a set of DDL statements a good understanding of the enterprise's information resources and needs.
- **Storage structure and access method definition**

Database Administrator(contd.)



- Schema and physical organization modification
- Granting users authority to access the database
- Backing up data
- Monitoring performance and responding to changes
- Database tuning.

Database Users and Administrators



Database

Storage Manager

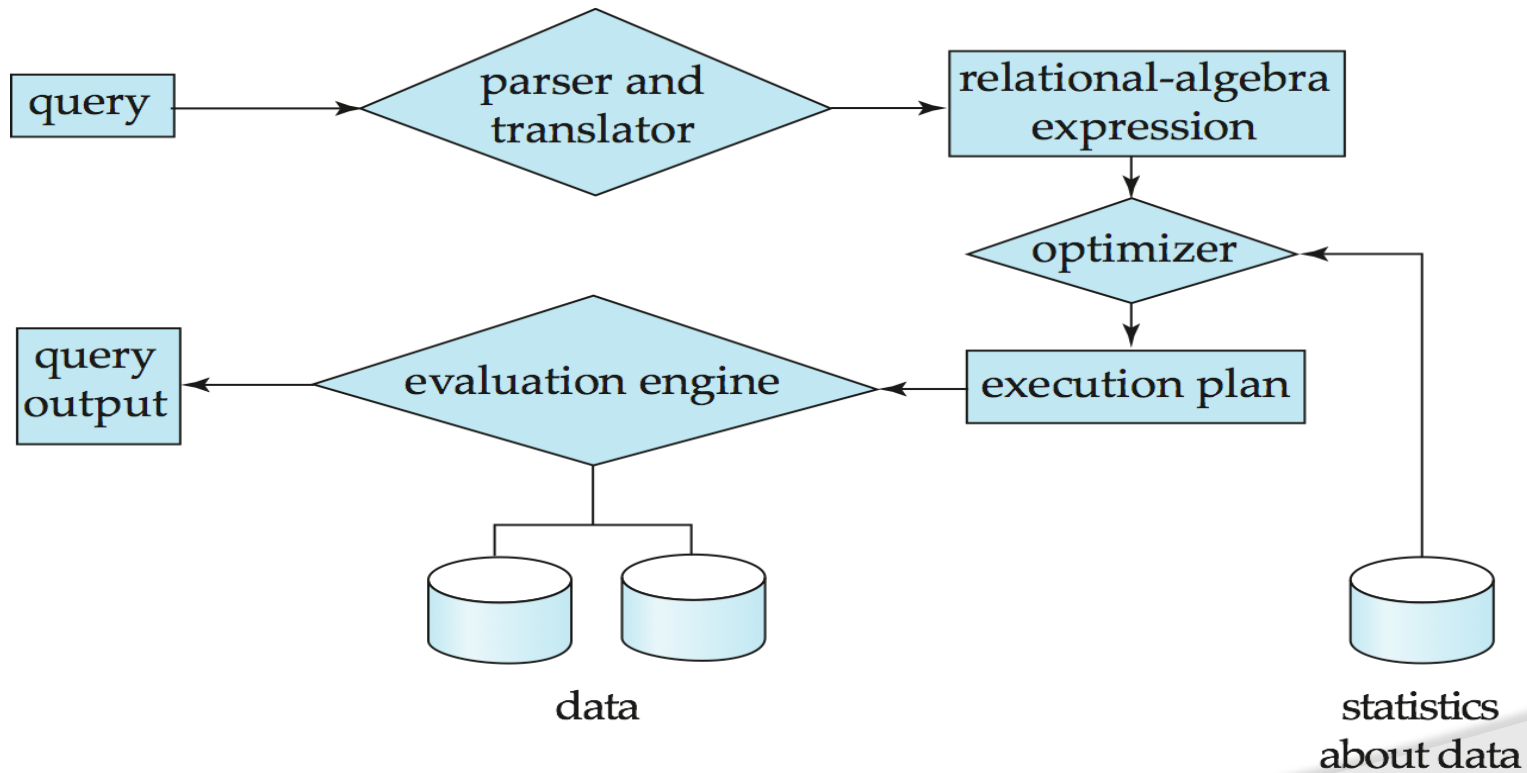
Storage manager is a program module that provides the interface between the low-level data stored in the database and the application programs and queries submitted to the system.

The storage manager is responsible to the following tasks:

- Interaction with the OS file manager
- Efficient storing, retrieving and updating of data Issues:
 - Storage access
 - File organization
 - Indexing and hashing

Query Processing

- Parsing and translation
- Optimization
- Evaluation

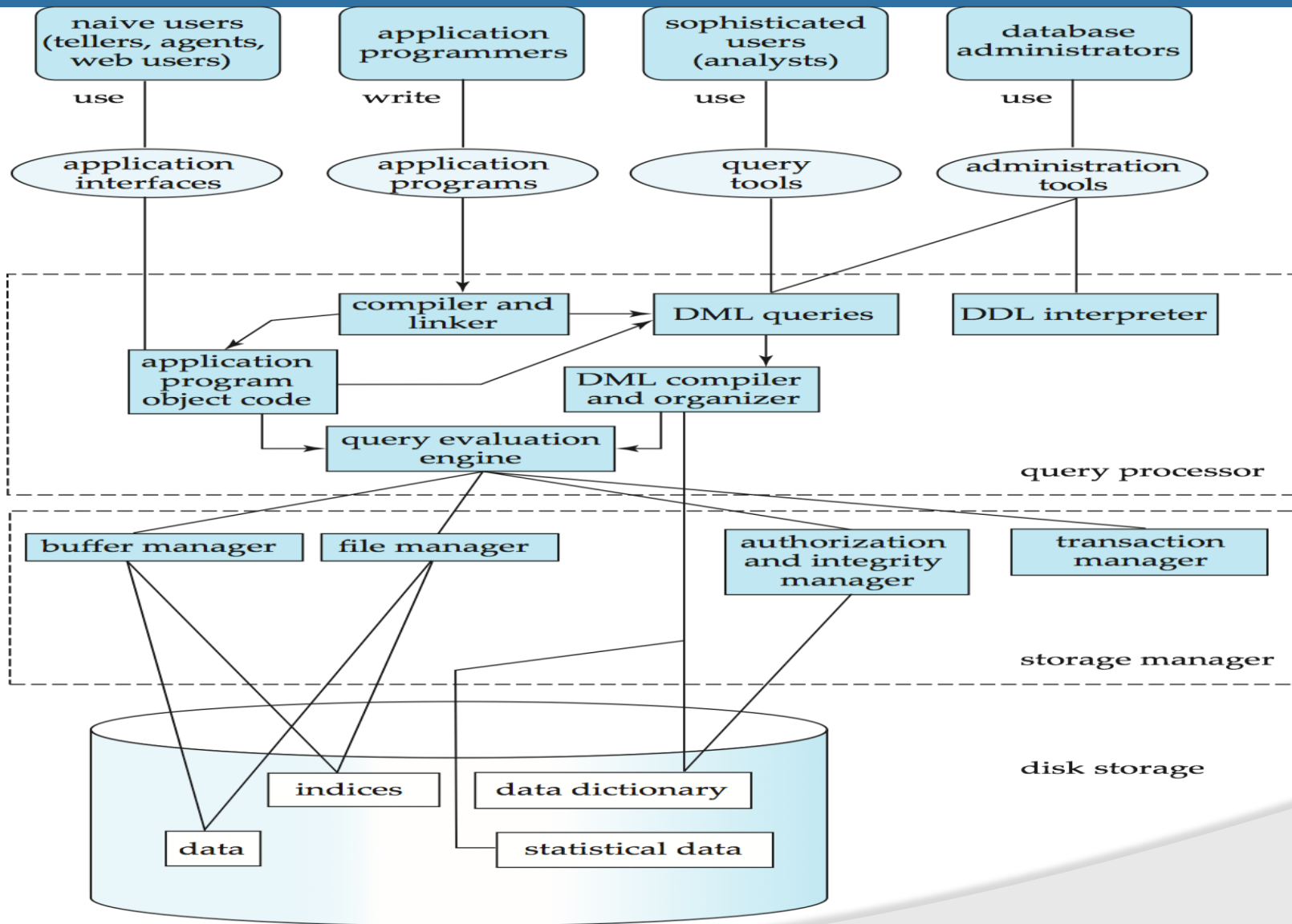


Query Processing (Cont.)

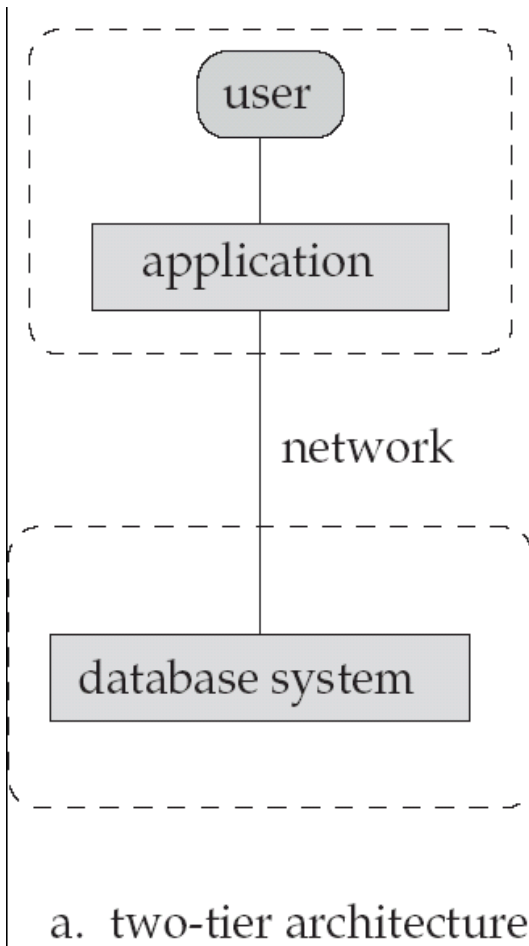
Alternative ways of evaluating a given query

- Equivalent expressions
- Different algorithms for each operation
- Cost difference between a good and a bad way of evaluating a query can be enormous
- Need to estimate the cost of operations
- Depends critically on statistical information about relations which the database must maintain
- Need to estimate statistics for intermediate results to compute cost of complex expressions

Database System Internals

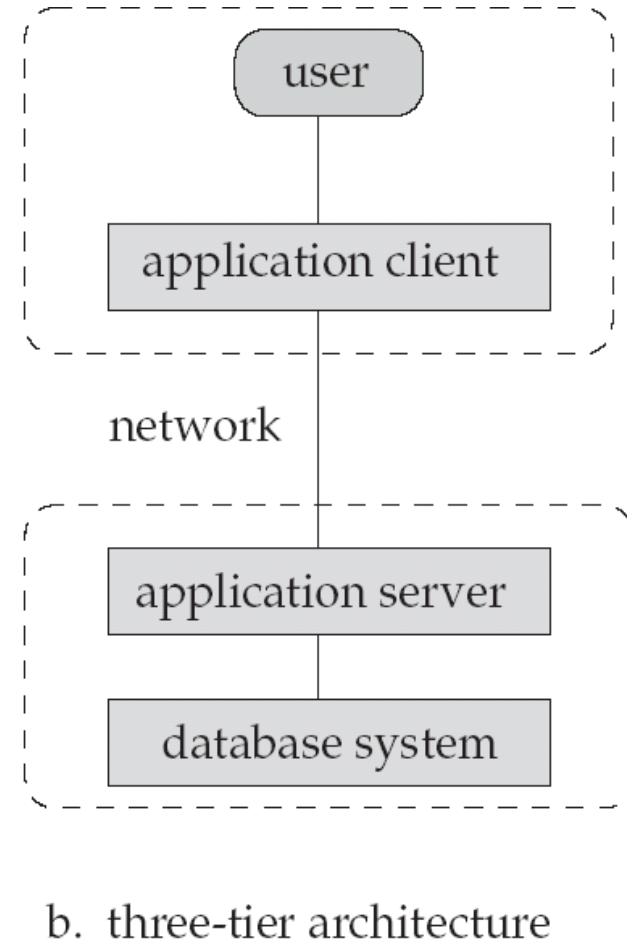


Database Application Architectures:



client

server



Storage Management

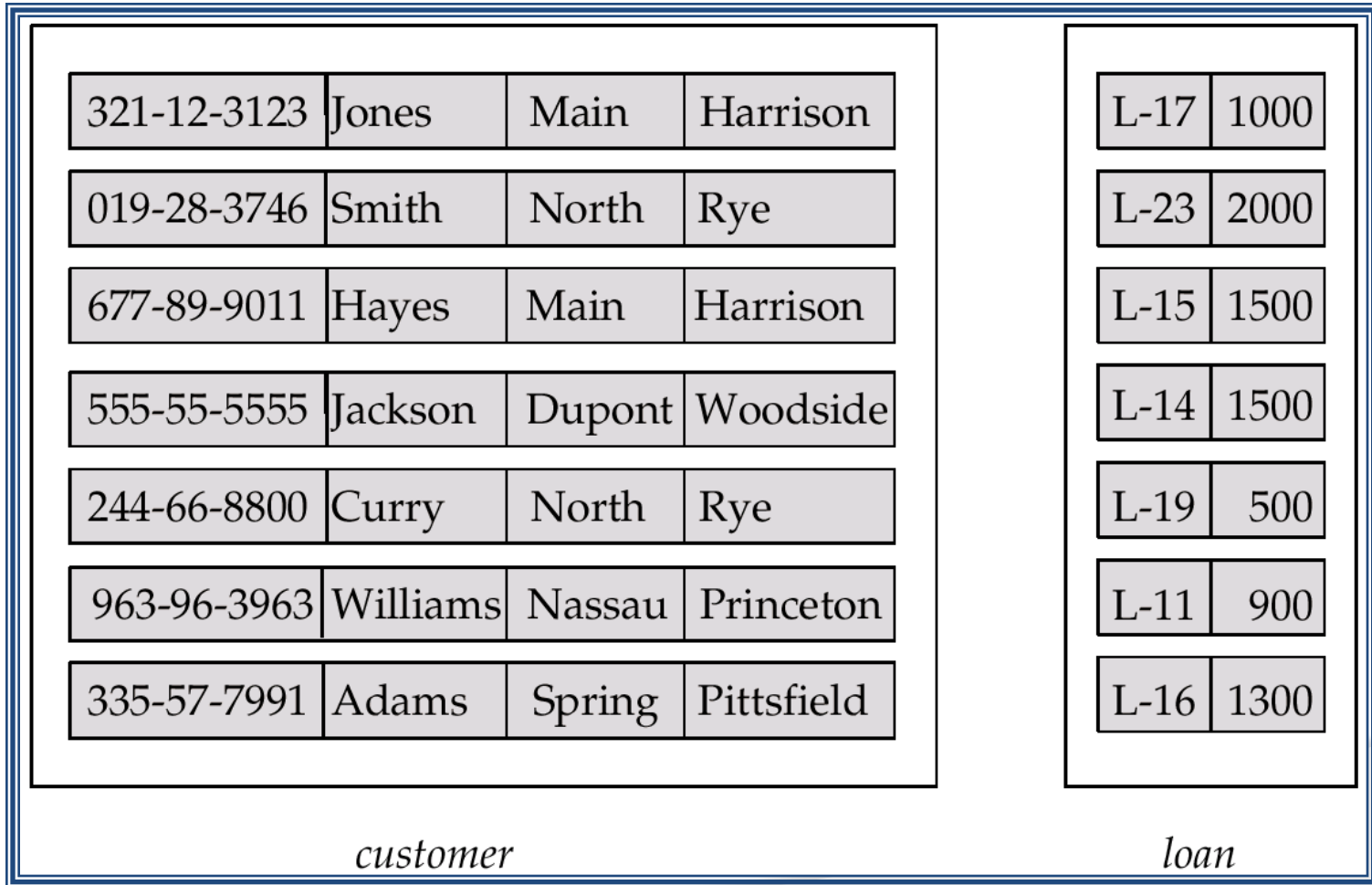
Storage manager is a program module that provides the interface between the low-level data stored in the database and the application programs and queries submitted to the system

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



Attributes

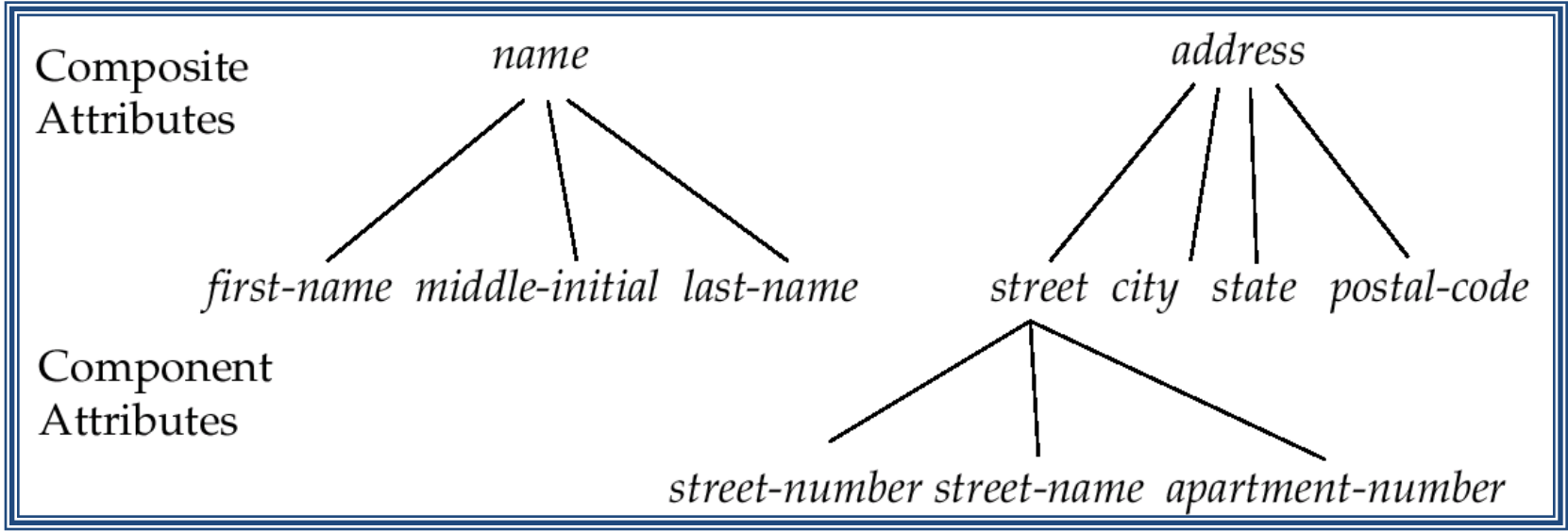
- An entity is represented by a set of attributes, that is descriptive properties possessed by all members of an entity set.

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:

<u>Hayes</u>	<u>depositor</u>	<u>A-102</u>
<i>customer</i> entity	relationship set	<i>account</i> entity

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

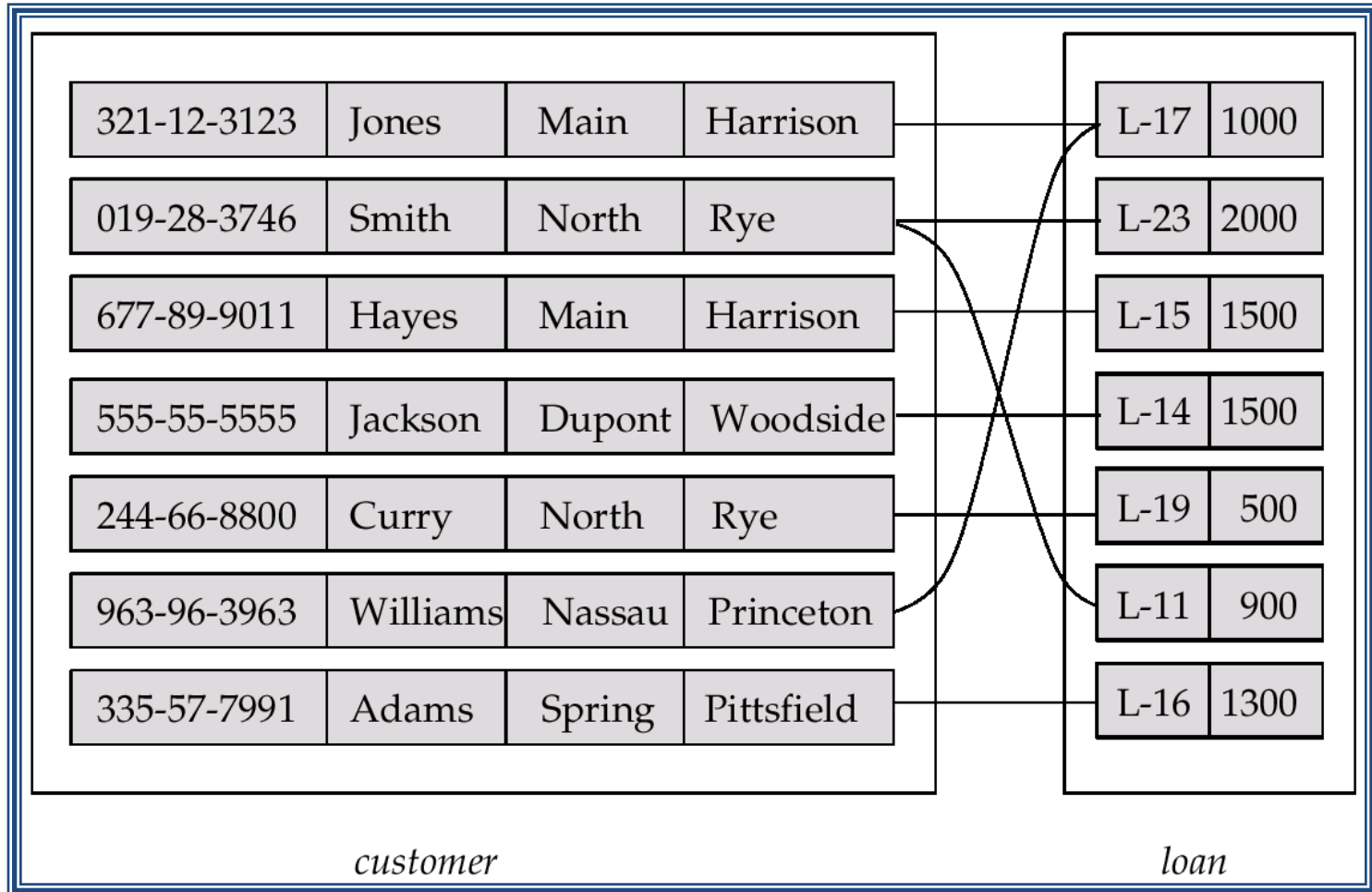
$$\{(e_1, e_2, \dots, e_n) \mid e_1 \in E_1, e_2 \in E_2, \dots, e_n \in E_n\}$$

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

Example:

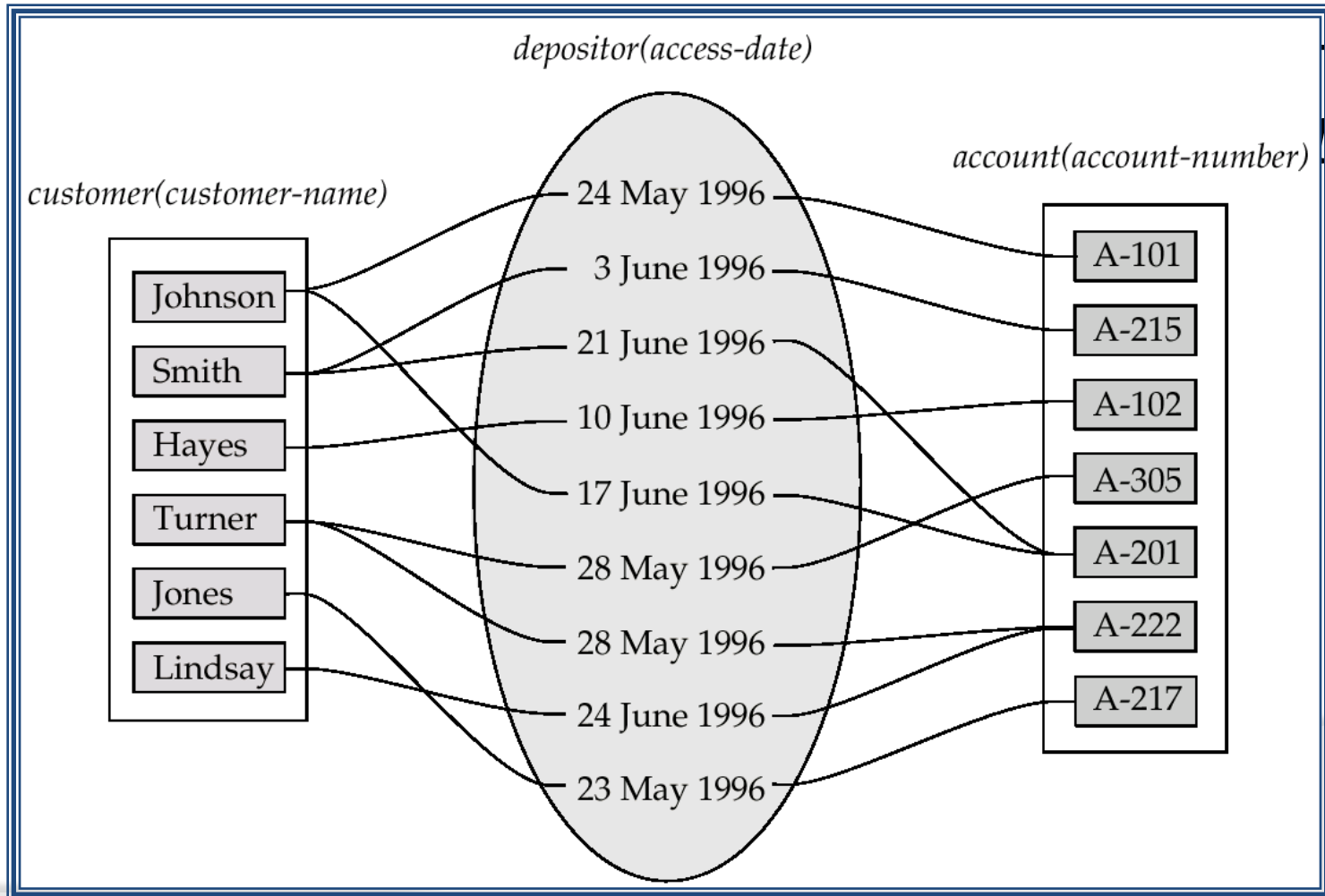
$$(Hayes, A-102) \in depositor$$

Relationship Set *borrower*



Relationship Sets (Cont.)

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



Degree of a Relationship Set

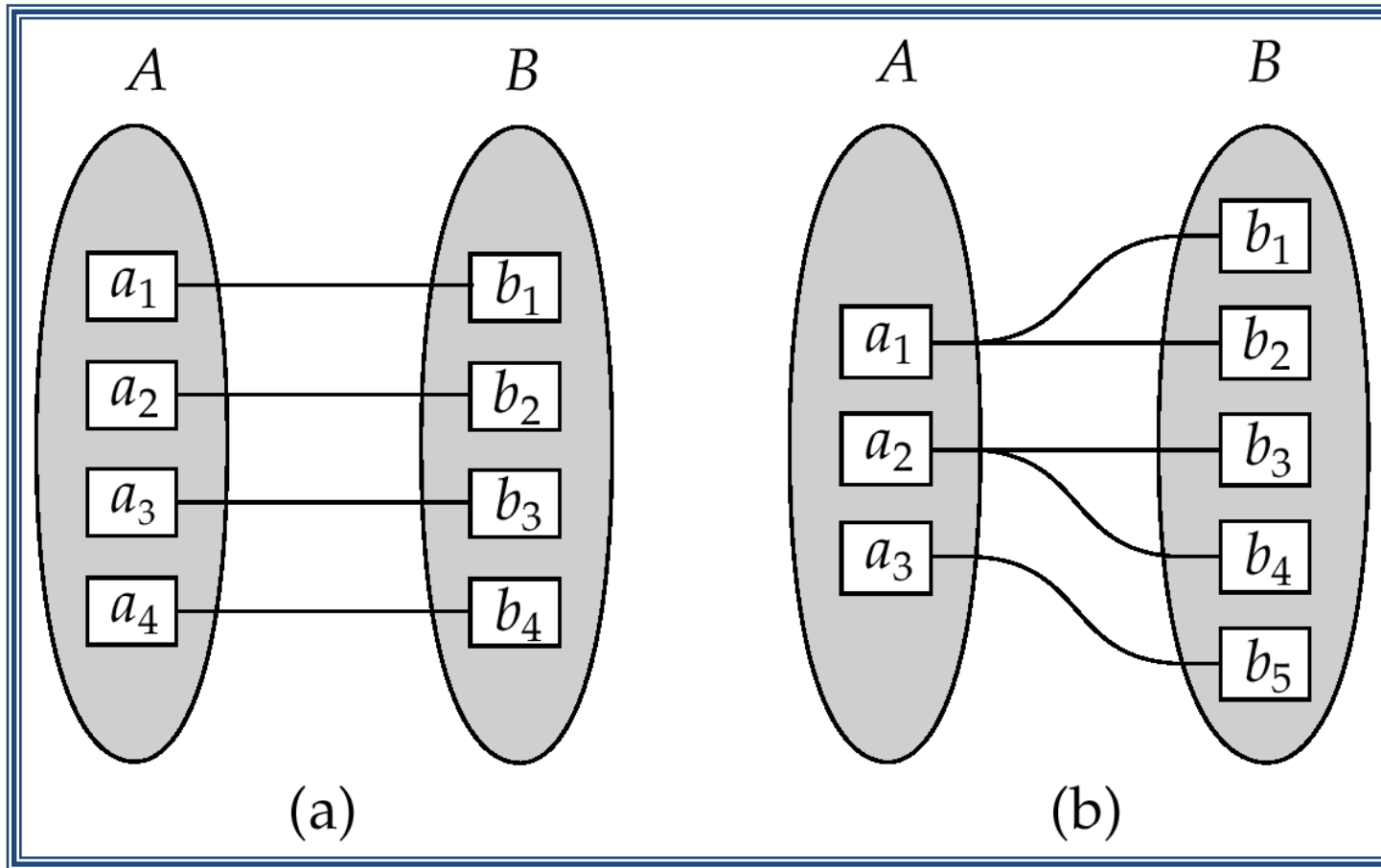
- Refers to number of entity sets that participate in a relationship set.
- Relationship sets that involve two entity sets are *binary* (or degree two). Generally, most relationship sets in a database system are binary.
- Relationship sets may involve more than two entity sets.

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

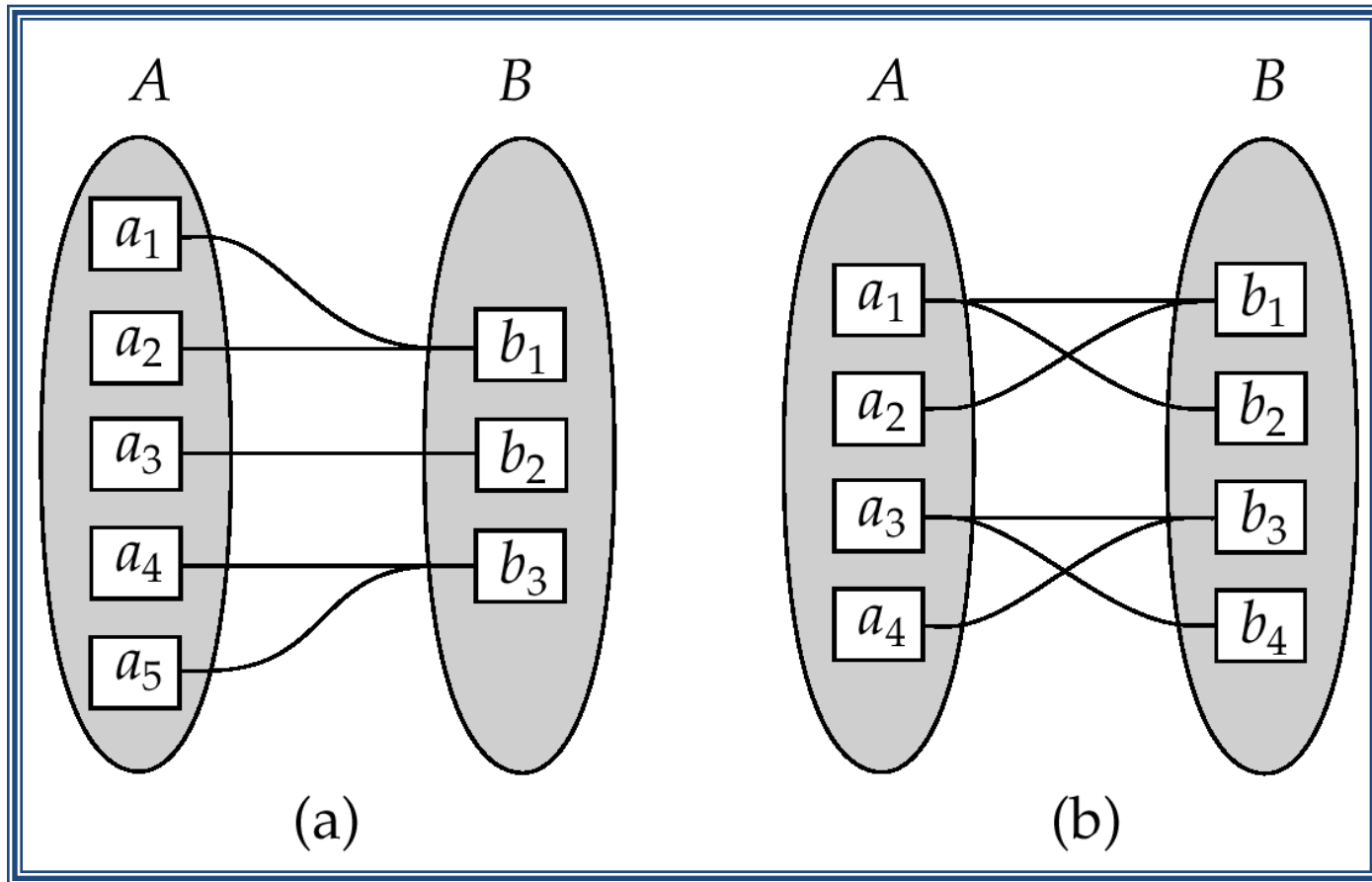


One to one

One to many

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

Mapping Cardinalities



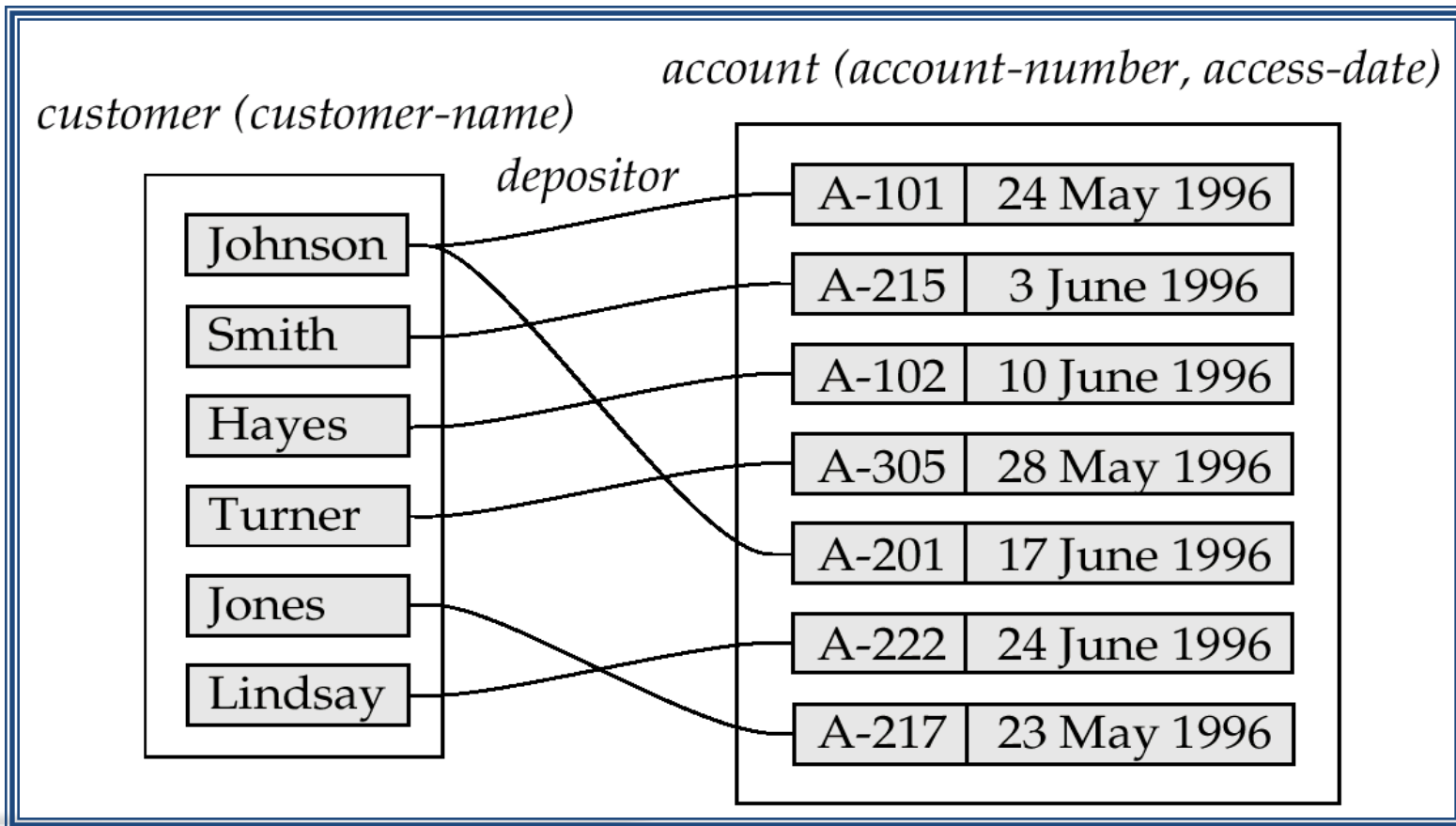
Many to one

Many to many

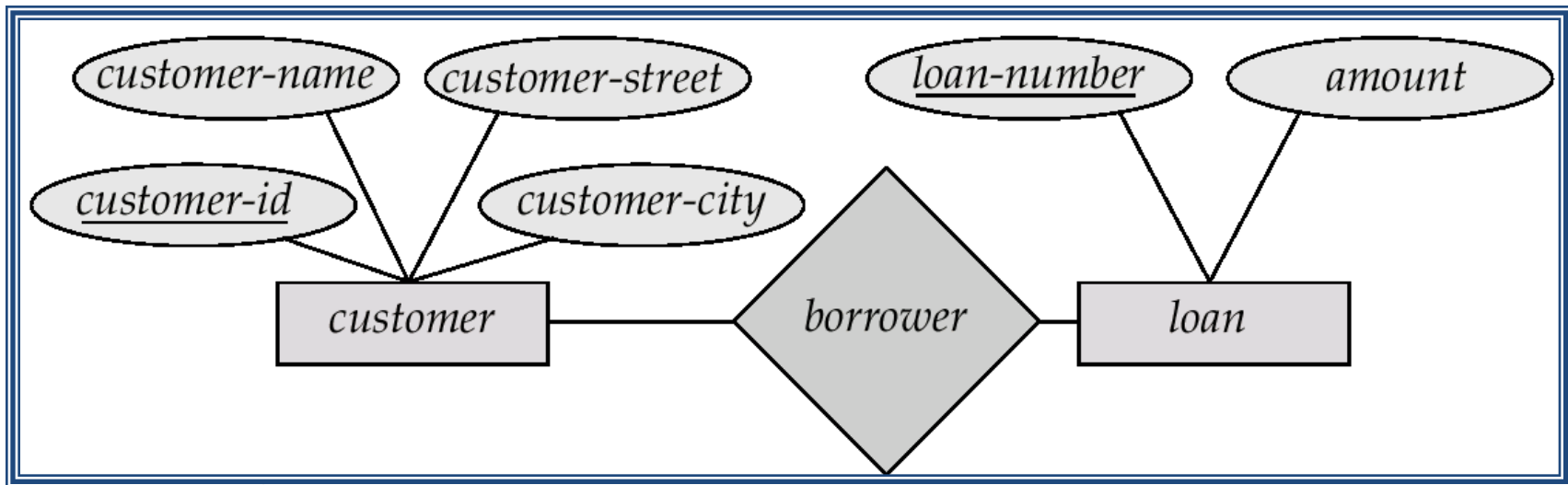
Note: Some elements in A and B may not be mapped to any elements in the other set

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

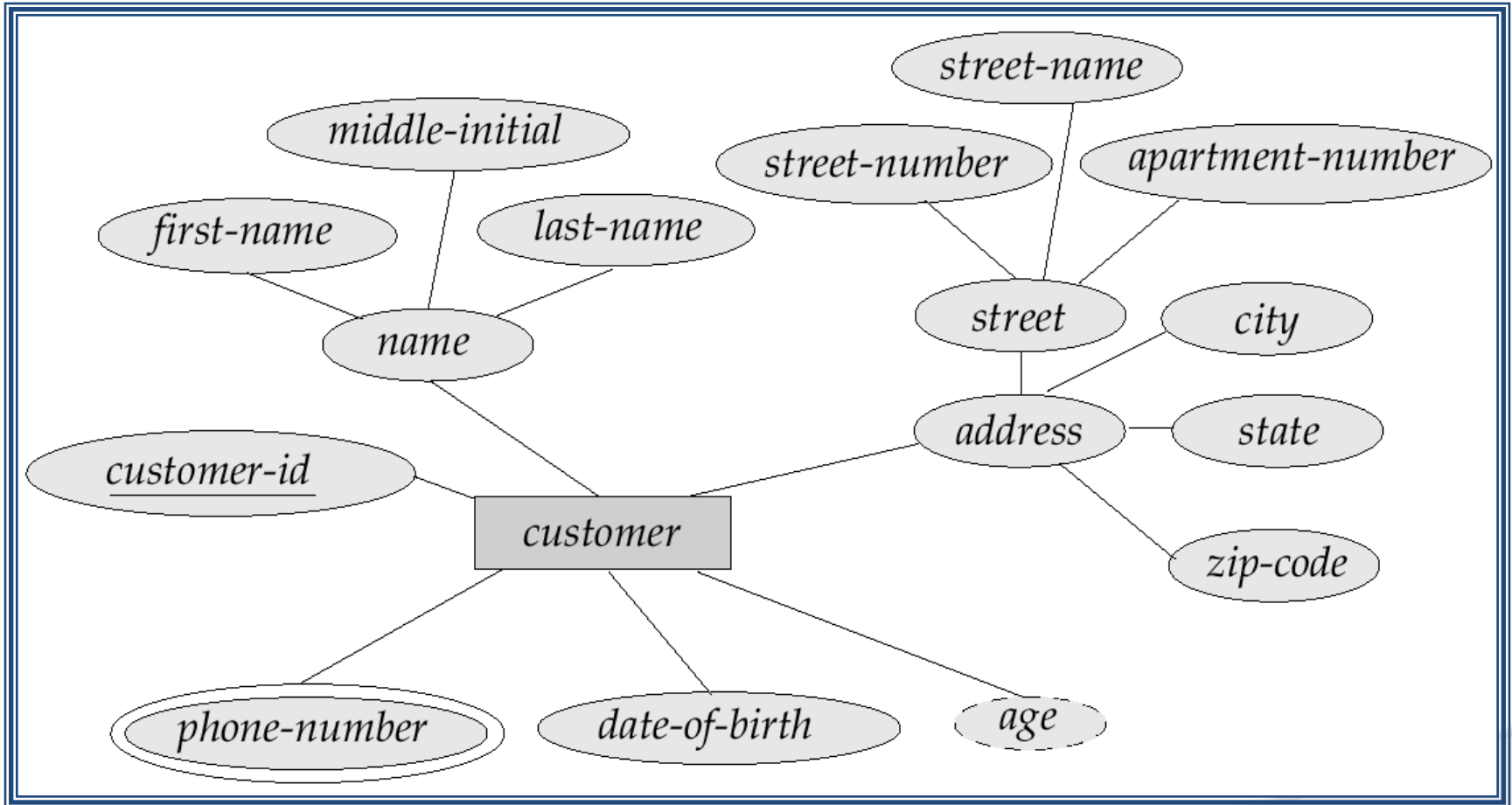


E-R Diagrams



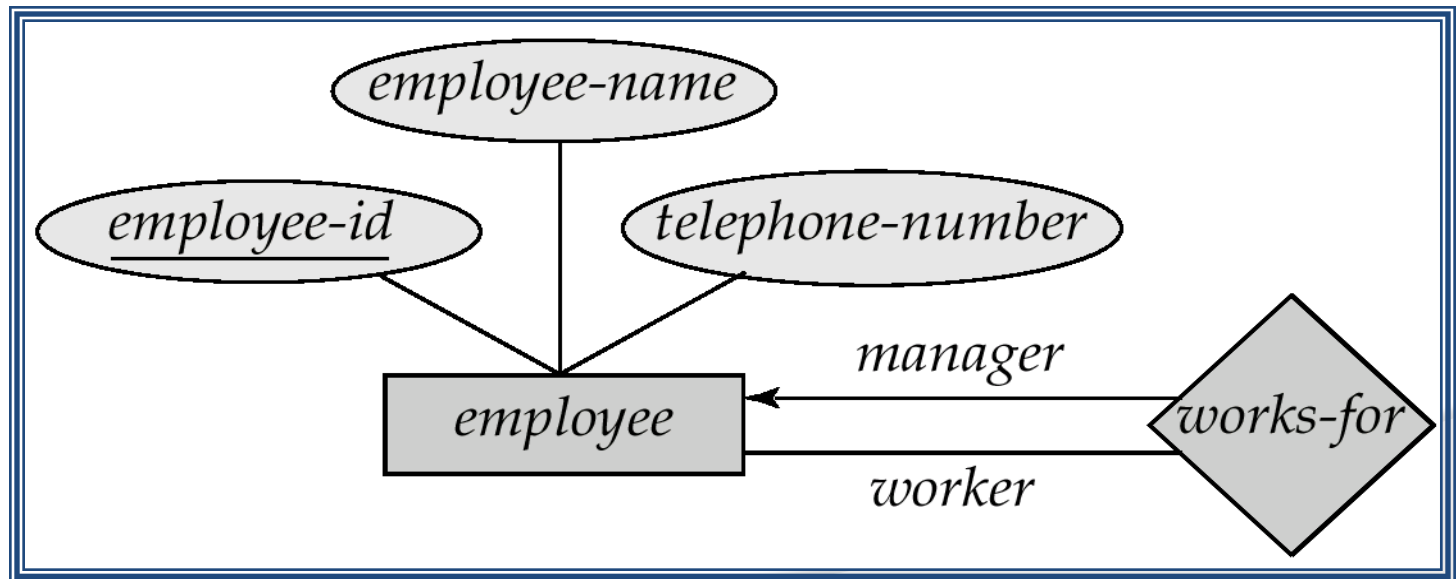
- **Rectangles** represent entity sets.
- **Diamonds** represent relationship sets.
- **Lines** link attributes to entity sets and entity sets to relationship sets.
- **Ellipses** represent attributes
 - **Double ellipses** represent multivalued attributes.
 - **Dashed ellipses** denote derived attributes.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

Composite, Multivalued, Derived



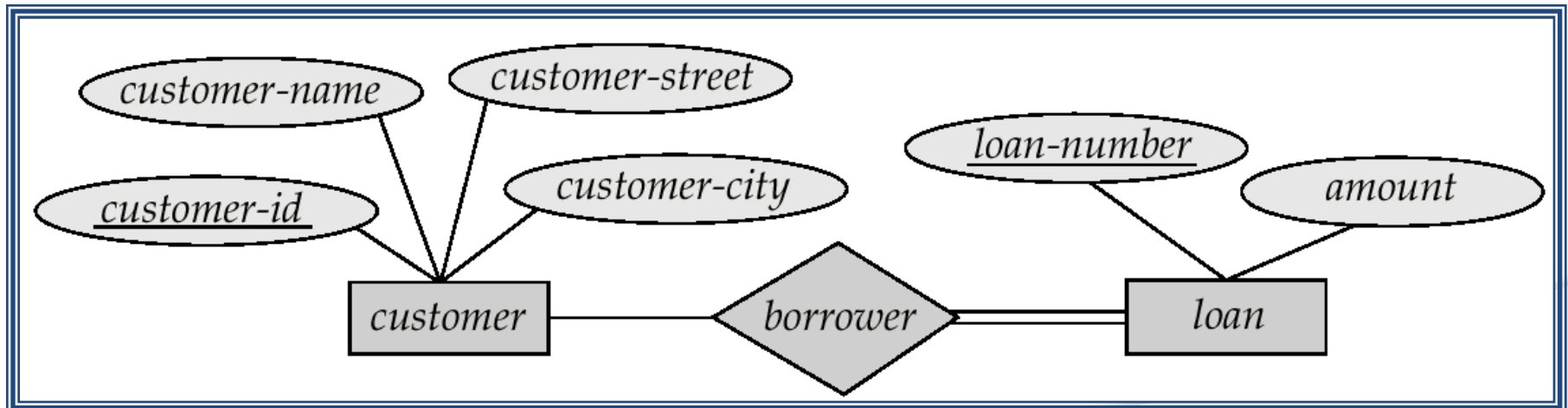
Roles

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



Entity Set in a Relationship

- **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



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**

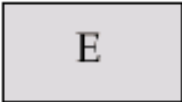
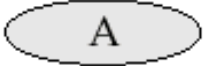
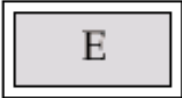
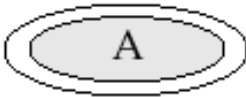



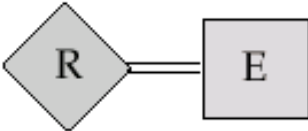


Possible guideline is to designate a relationship set to describe an action that occurs between entities

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

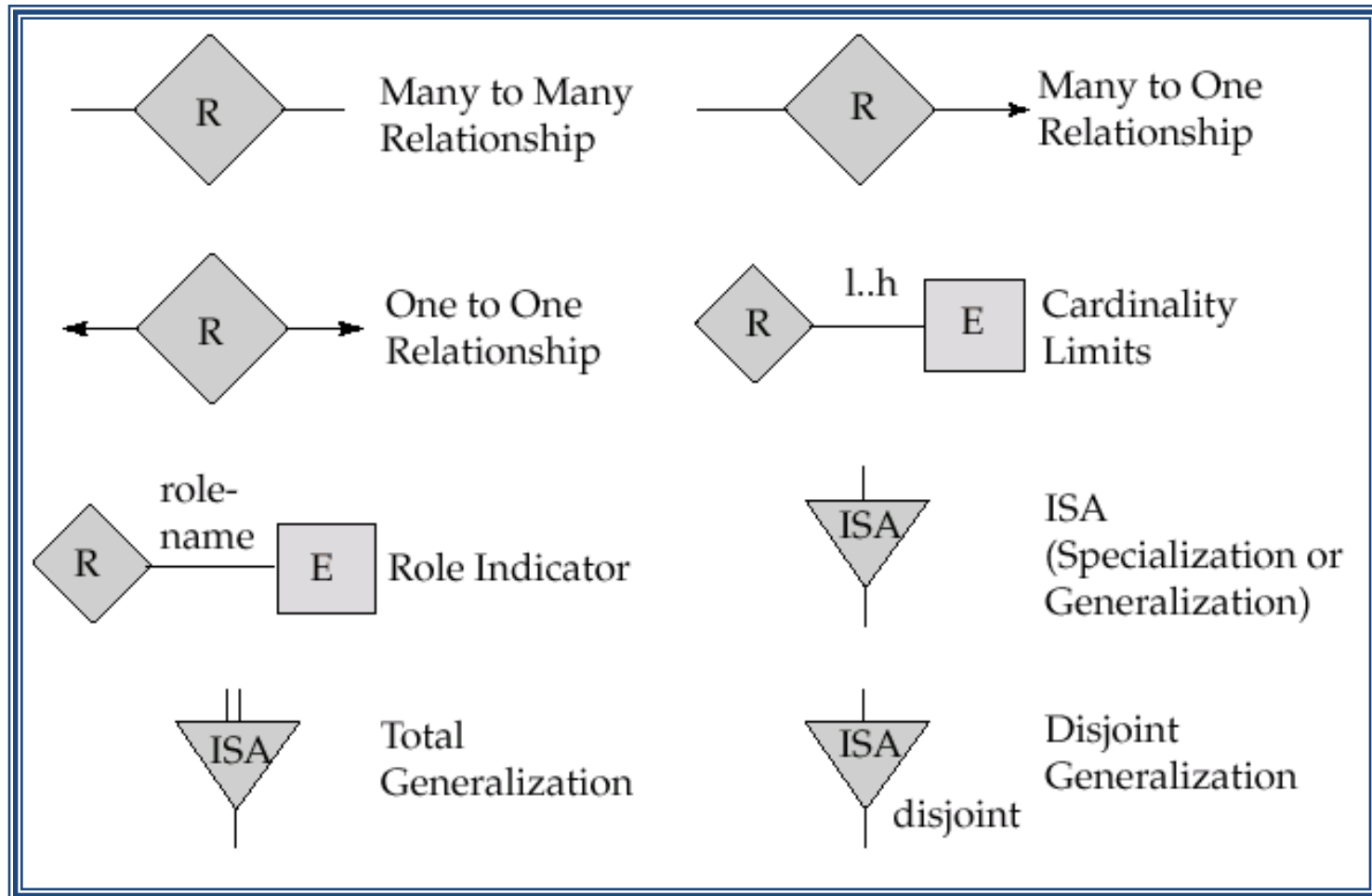
Although it is possible to replace any non binary (n -ary, for $n > 2$) relationship set by a number of distinct binary relationship sets, a n -ary relationship set shows more clearly that several entities participate in a single relationship.

- Placement of relationship attributes

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.)

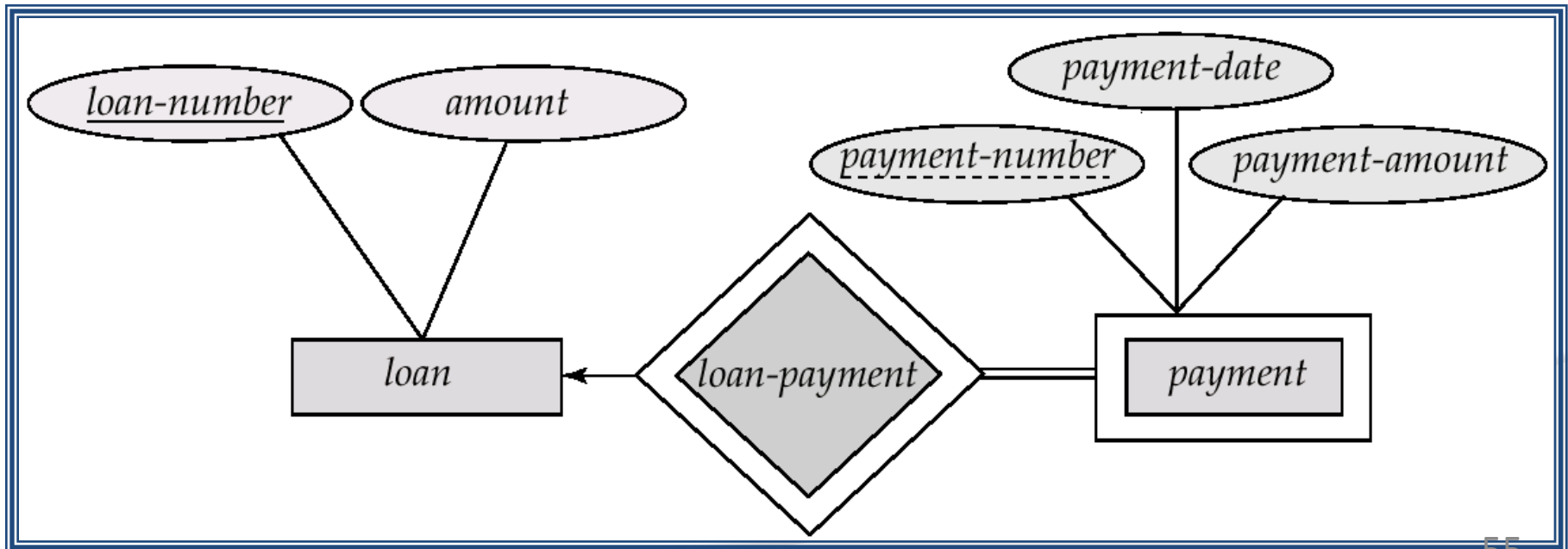


Weak Entity Sets

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

Weak Entity Sets (Cont.)

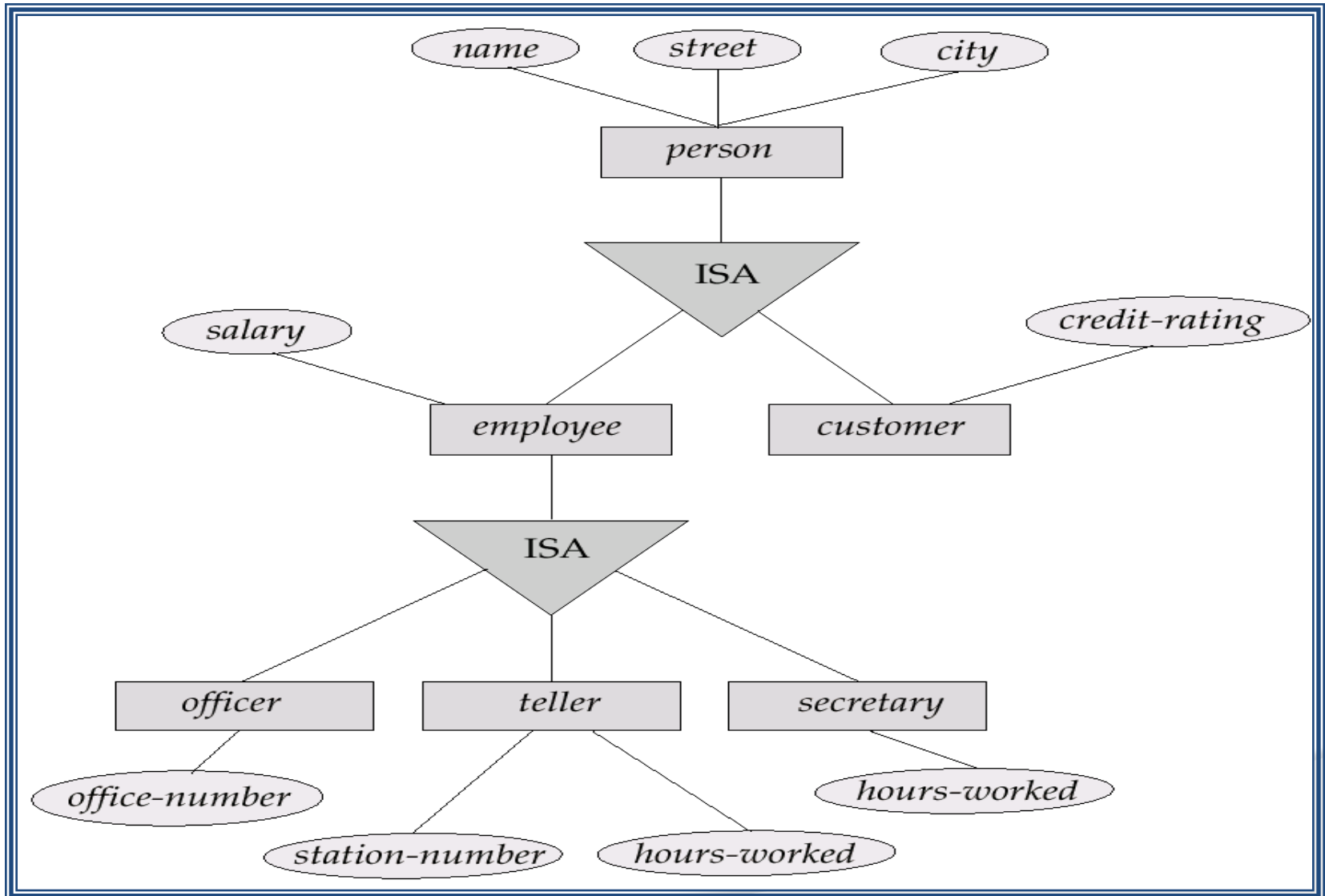
- We depict a weak entity set by double rectangles.
- We underline the discriminator of a weak entity set with a dashed line.
- *payment-number* – discriminator of the *payment* entity set
- Primary key for *payment* – (*loan-number*, *payment-number*)



Specialization

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

Specialization Example



Generalization

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

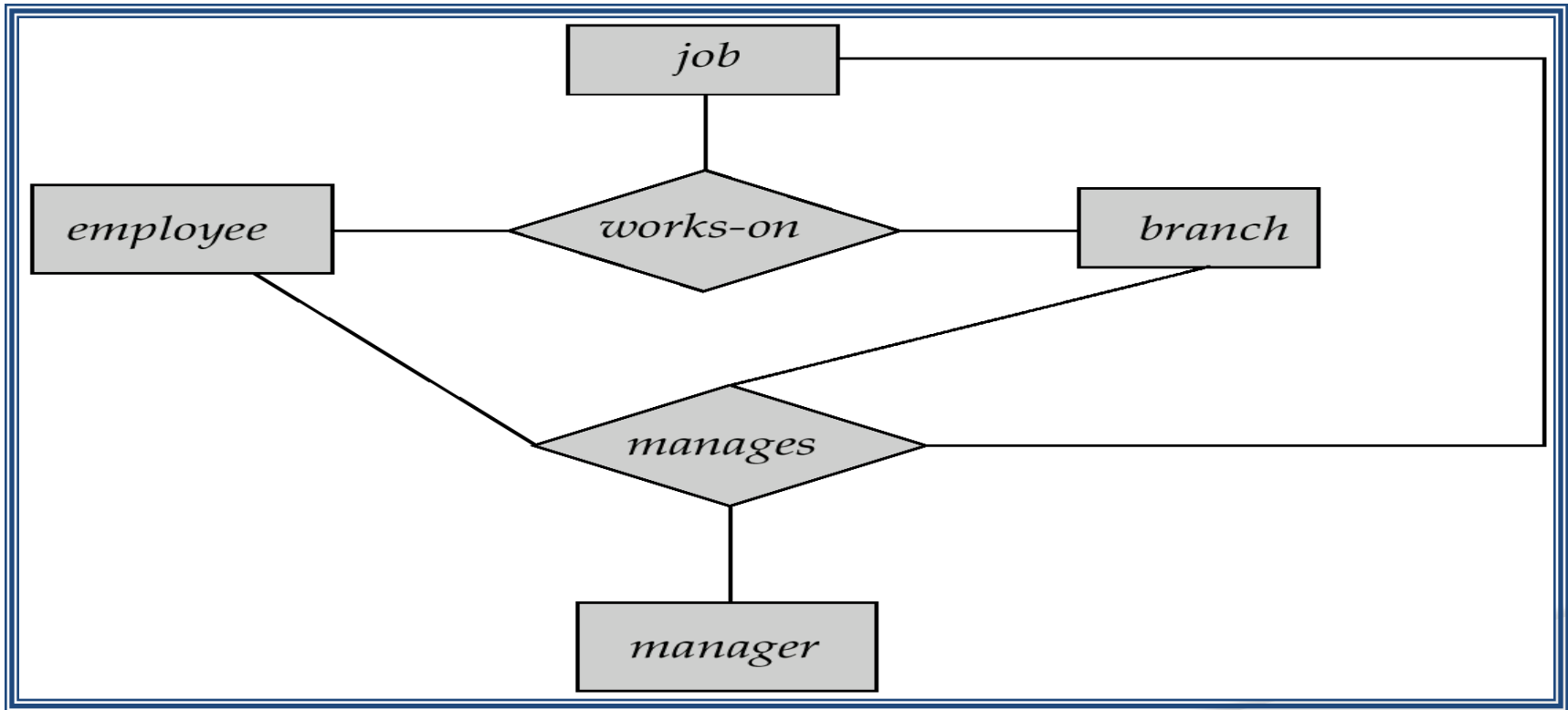
Specialization and Generalization



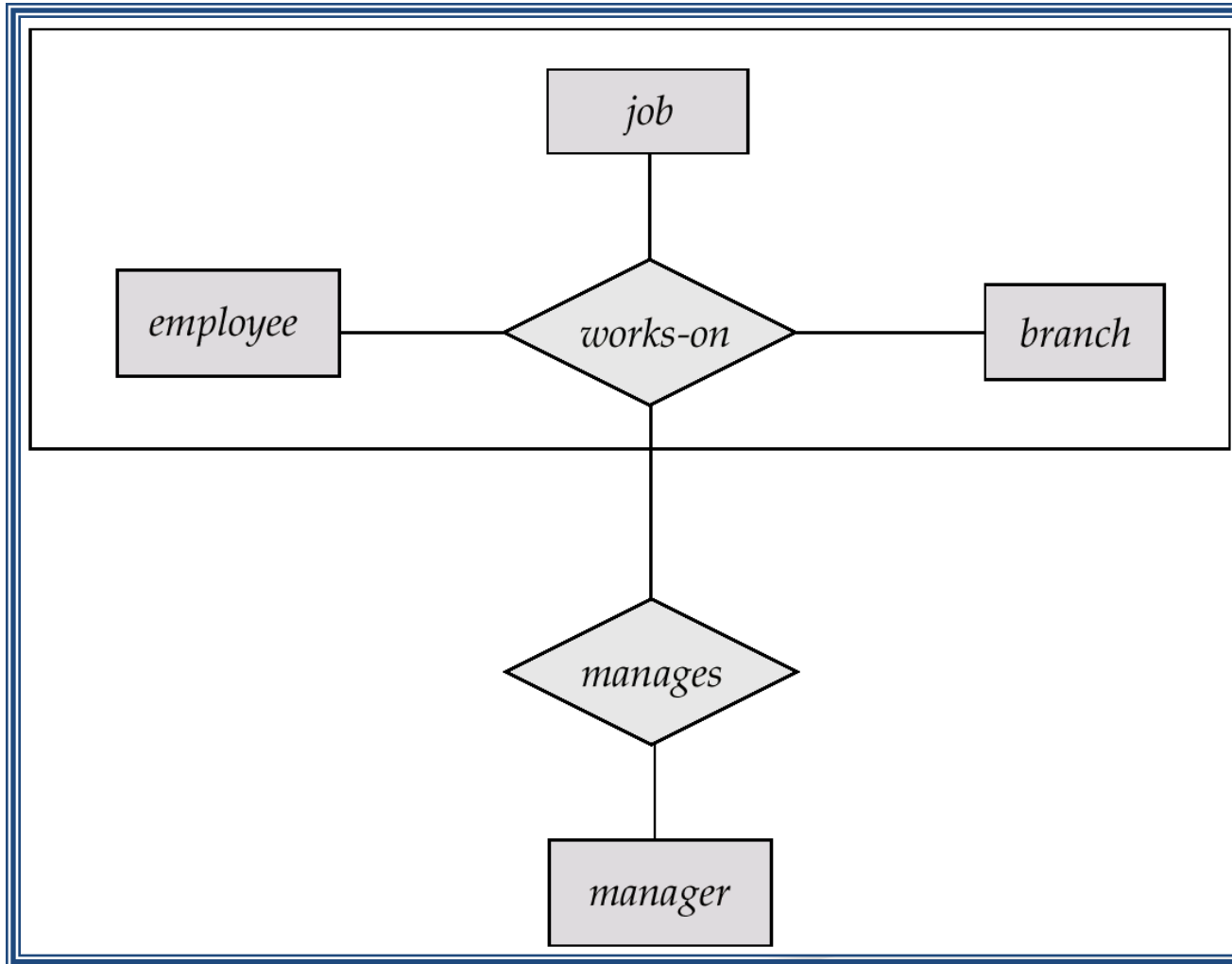
- Can have multiple specializations of an entity set based on different features.
- E.g. *permanent-employee* vs. *temporary-employee*, in addition to *officer* vs. *secretary* vs. *teller*
- Each particular employee would be
 - a member of one of *permanent-employee* or *temporary-employee*,
 - and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

Aggregation

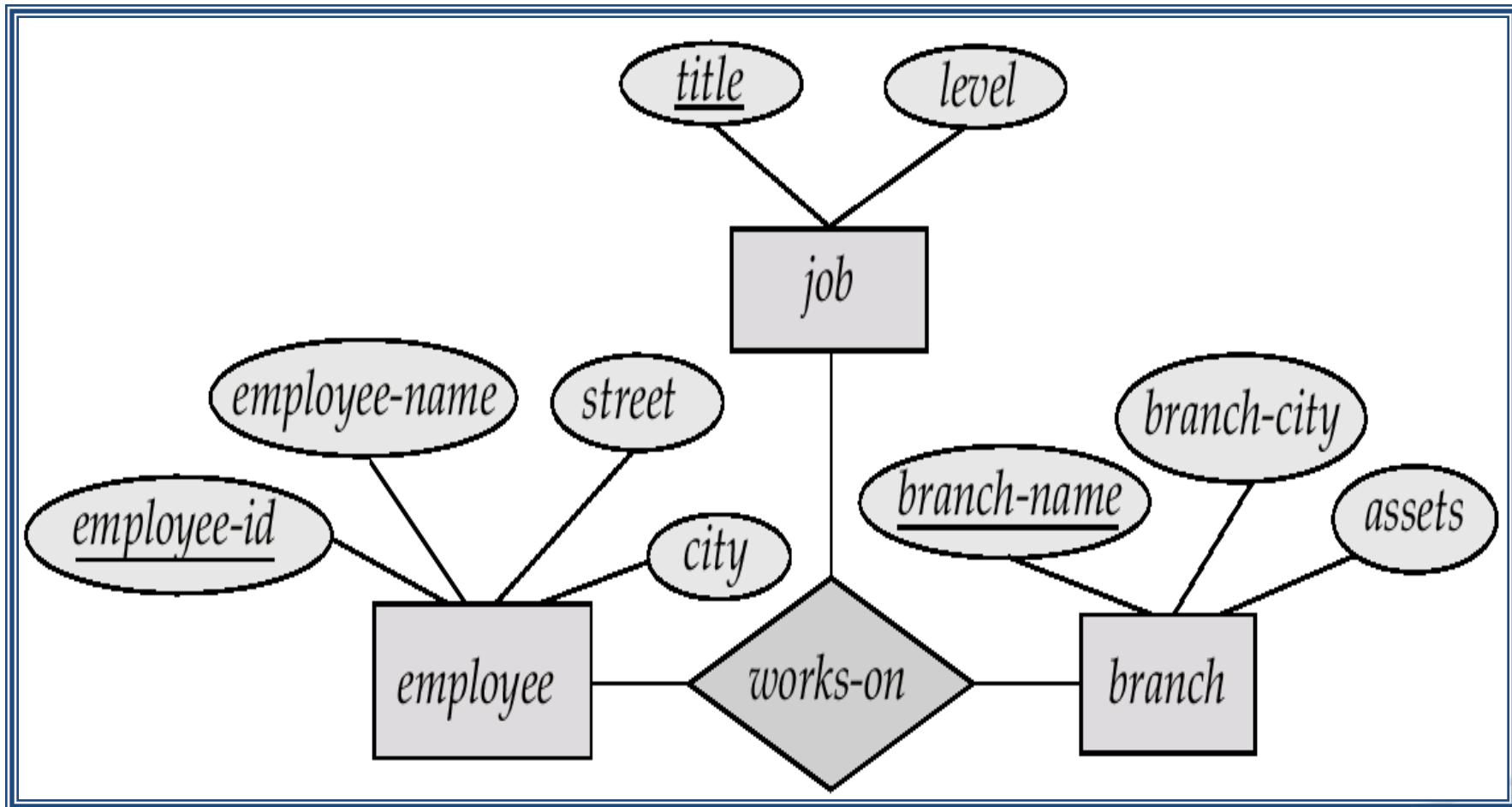
- Consider the ternary relationship *works-on*, which we saw earlier
- Suppose we want to record managers for tasks performed by an employee at a branch



E-R Diagram With Aggregation



Ternary Relationship



Relational Database Approach

Relational algebra and calculus: Relational algebra, selection and projection, set operations, renaming, joins, division, examples of algebra queries, relational calculus, tuple relational calculus

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

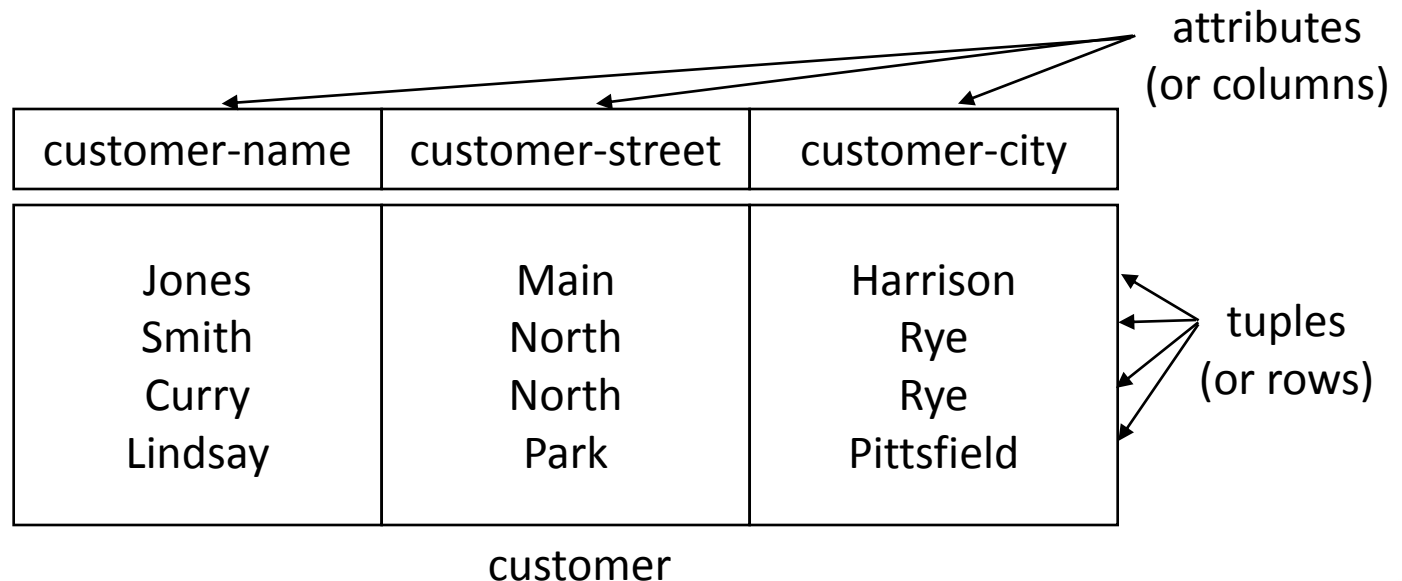
- Each attribute of a relation has a name
- The set of allowed values for each attribute is called the **domain** of the attribute
- Attribute values are (normally) required to be **atomic**, that is, indivisible
 - E.g. multivalued attribute values are not atomic
 - E.g. composite attribute values are 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



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 deals with how to design relational schemas

Relational Model

- Relational model
 - First commercial implementations available in early 1980s
 - Has been implemented in a large number of commercial system
 - Hierarchical and network models
 - Preceded the relational model
 - Represents data as a collection of relations
 - Table of values
 - Row
 - Represents a collection of related data values
 - Fact that typically corresponds to a real-world entity or relationship
 - *Tuple*
- Table name and column names
- Interpret the meaning of the values in each row attribute

Relational Model

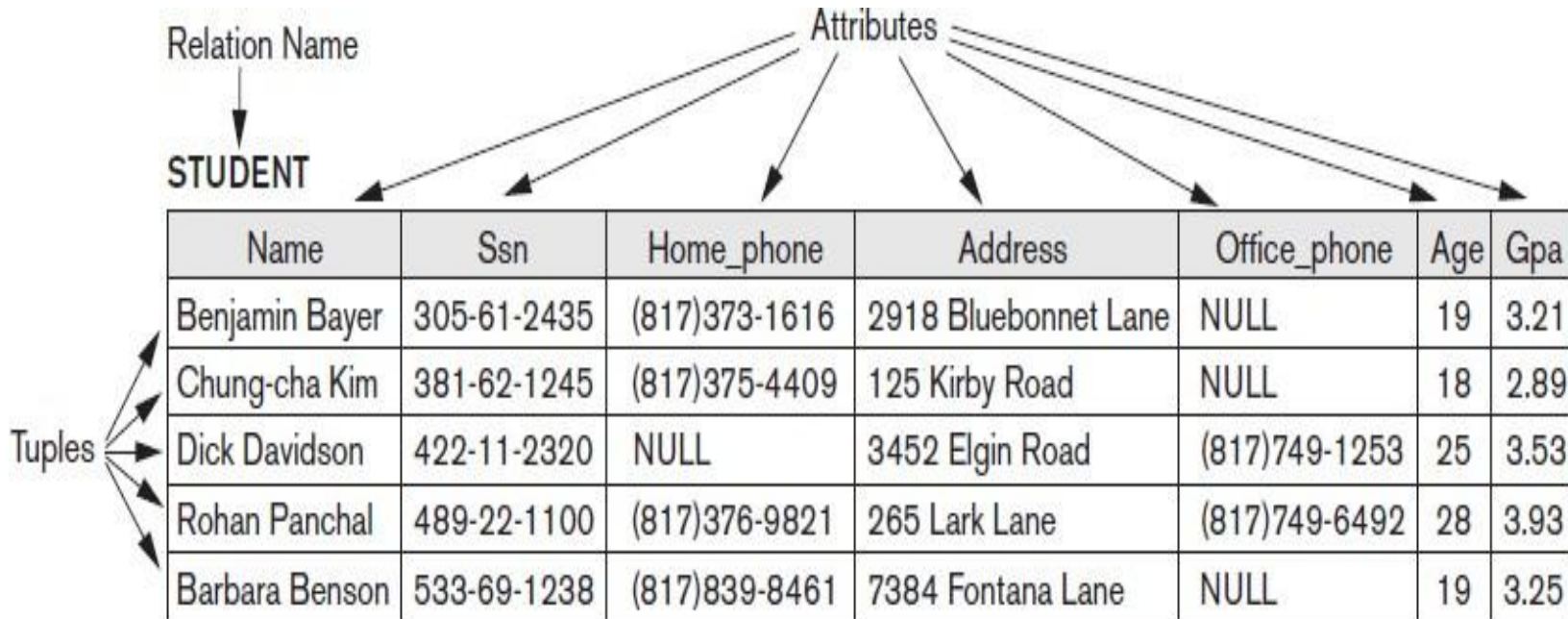


Figure 3.1

The attributes and tuples of a relation STUDENT.

Domains, Attributes, Tuples Relations

- Domain D
 - Set of atomic values
- Atomic
 - Each value indivisible
- Specifying a domain
 - Data type specified for each domain
- Relation schema R
 - Denoted by $R(A_1, A_2, \dots, A_n)$
 - Made up of a relation name R and a list of attributes, A_1, A_2, \dots, A_n
- Attribute A_i
 - Name of a role played by some domain D in the relation schema R
- Degree (or arity) of a relation
 - Number of attributes n of its relation schema

Domains, Attributes, Tuples Relations

- Relation (or relation state)
 - Set of n -tuples $r = \{t_1, t_2, \dots, t_m\}$
 - Each n -tuple t
 - Ordered list of n values $t = \langle v_1, v_2, \dots, v_n \rangle$
 - Each value v_i , $1 \leq i \leq n$, is an element of $\text{dom}(A_i)$ or is a special NULL value

- Relation (or relation state) $r(R)$
 - Mathematical relation of degree n on the domains $\text{dom}(A_1)$, $\text{dom}(A_2)$, ..., $\text{dom}(A_n)$
 - Subset of the Cartesian product of the domains that define R:
 - $r(R) (\text{dom}(A_1) \times \text{dom}(A_2) \times \dots \times \text{dom}(A_n))$

Relational Model Notation

- Interpretation (meaning) of a relation
 - Assertion
 - Each tuple in the relation is a fact or a particular instance of the assertion
 - Predicate
 - Values in each tuple interpreted as values that satisfy predicate
- Relation schema R of degree n
 - Denoted by $R(A_1, A_2, \dots, A_n)$
- Uppercase letters Q, R, S
 - Denote relation names
- Lowercase letters q, r, s
 - Denote relation states
- Letters t, u, v
 - Denote tuples

Relational Model Notation

- Name of a relation schema: STUDENT
 - Indicates the current set of tuples in that relation
- Notation: STUDENT(Name, Ssn, ...)
 - Refers only to relation schema
- Attribute A can be qualified with the relation name R to which it belongs
 - Using the dot notation $R.A$
- n -tuple t in a relation $r(R)$
 - Denoted by $t = \langle v_1, v_2, \dots, v_n \rangle$
 - v_i is the value corresponding to attribute A_i
- Component values of tuples:
 - $t[A_i]$ and $t.A_i$ refer to the value v_i in t for attribute A_i
 - $t[A_u, A_w, \dots, A_z]$ and $t.(A_u, A_w, \dots, A_z)$ refer to the subtuple of values $\langle v_u, v_w, \dots, v_z \rangle$ from t corresponding to the attributes specified in the list

Query languages

- Query Languages are categorized as Pure Query languages and Commercial Query languages
- Languages which are defined theoretically and mathematically are known as Pure query languages
Example: Relational Algebra

Commercial Query languages are developed based on Pure query languages for implementation purpose

Example : SQL

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 of query languages that people use.

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

- 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:

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

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

- Example of selection:

$$\sigma_{\text{branch-name}=\text{"Perryridge"}}(\text{account})$$

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
- E.g. To eliminate the *branch-name* attribute of *account*

$$\Pi_{\text{account-number, balance}} (\text{account})$$

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 values as does the 2nd column of s)
- E.g. to find all customers with either an account or a loan

$$\Pi_{customer-name} (depositor) \cup \Pi_{customer-name} (borrower)$$

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

RA - Operations Examples

Banking :

branch (branch_name, branch_city, assets)

*customer (customer_name, customer_street,
customer_city)*

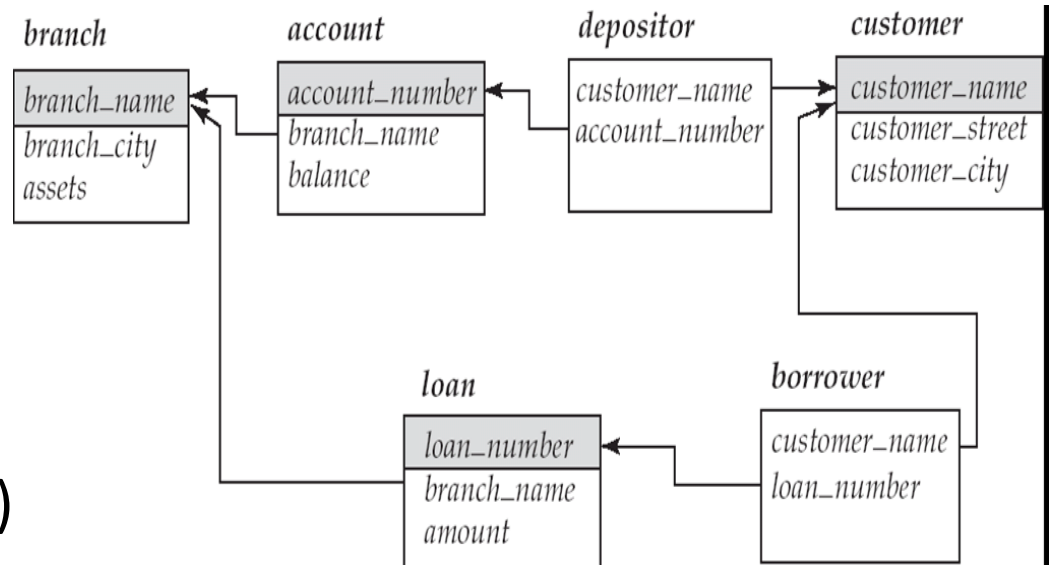
account (account_number, branch_name, balance)

loan (loan_number, branch_name, amount)

depositor (customer_name, account_number)

borrower (customer_name, loan_number)

RA - Operations Examples



Find all loans of over \$1200

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

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

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

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

$$\Pi_{customer_name} (borrower) \cup \Pi_{customer_name} (depositor)$$

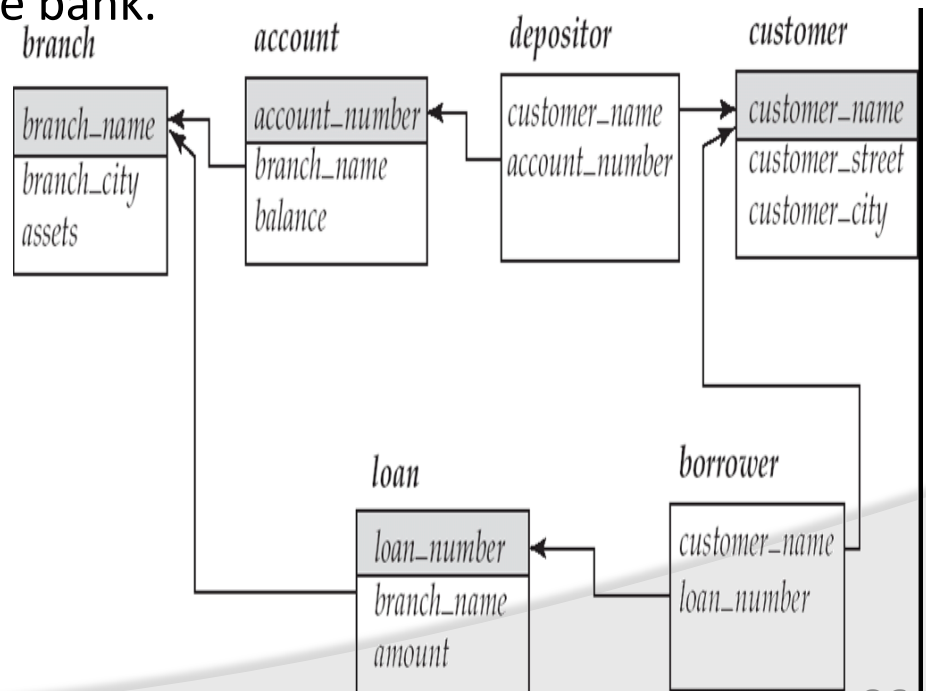
Example Queries

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

$$\Pi_{customer_name} (\sigma_{branch_name="Perryridge"} (\sigma_{borrower.loan_number = loan.loan_number} (borrower \times loan)))$$

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

$$\Pi_{customer_name} (\sigma_{branch_name = "Perryridge"} (\sigma_{borrower.loan_number = loan.loan_number} (borrower \times loan))) - \Pi_{customer_name} (depositor)$$



Outer Join – Example

Relation *loan*

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

- Relation *borrower*

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

Outer Join – Example

Join

loan ⋈ *borrower*

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

■ Left Outer Join

loan ⋈_L *borrower*

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

Outer Join – Example

- Right Outer Join

loan ⋈ *borrower*

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

- Full Outer Join

loan ⋈ *borrower*

<i>loan_number</i>	<i>branch_name</i>	<i>amount</i>	<i>customername</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

- Question:** can outerjoins be expressed using basic relational

algebra operations

Division Operation (Cont.)

Property

Let $q = r \div s$

Then q is the largest relation satisfying $q \times s \subseteq r$

Definition in terms of the basic algebra operation

Let $r(R)$ and $s(S)$ be relations, and let $S \subseteq R$

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

To see why

$\Pi_{R-S,S}(r)$ simply reorders attributes of r

$\Pi_{R-S}(\Pi_{R-S}(r) \times s) - \Pi_{R-S,S}(r)$ gives those tuples t in

$\Pi_{R-S}(r)$ such that for some tuple $u \in s$, $tu \notin r$.

RA - Advanced Operations



Advanced Operations

- Set intersection
- Natural join
- Aggregation
- Outer Join
- Division

All above, other than aggregation, can be expressed using basic operations we have seen earlier

Set-Intersection Operation – Example

Relation r, s :

A	B
α	1
α	2
β	1

r

A	B
α	2
β	3

s

$r \cap s$

A	B
α	2

Natural Join Operation – Example

Relations r , s :

A	B	C	D
α	1	α	a
β	2	γ	a
γ	4	β	b
α	1	γ	a
δ	2	β	b

r

B	D	E
1	a	α
3	a	β
1	a	γ
2	b	δ
3	b	ϵ

s

■ $r \bowtie s$

A	B	C	D	E
α	1	α	a	α
α	1	α	a	γ
α	1	γ	a	α
α	1	γ	a	γ
δ	2	β	b	δ

Aggregate Operation – Example

A	B	C
α	α	7
α	β	7
β	β	3
β	β	10

- $g_{\text{sum}(c)}(r)$

sum(c)
27

- Question: Which aggregate operations cannot be expressed using basic relational operations?

Aggregate Operation – Example

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

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

branch_name \mathcal{G} **sum**(*balance*) (*account*)

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

Outer Join

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

Outer Join – Example

Relation *loan*

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

- Relation *borrower*

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

Outer Join – Example

Join

loan ⋈ *borrower*

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

■ Left Outer Join

loan ⋈_L *borrower*

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

Outer Join – Example

- Right Outer Join

loan ⋈ *borrower*

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

- Full Outer Join

loan ⋈ *borrower*

<i>loan_number</i>	<i>branch_name</i>	<i>amount</i>	<i>customername</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

■ **Question:** can outerjoins be expressed using basic relational algebra operations

Null Values

- It is possible for tuples to have a null value, denoted by *null*
- *null* signifies an unknown value or that a value does not exist.
- The result of any arithmetic expression involving *null* is *null*.
- Aggregate functions simply ignore null values (as in SQL)

Null Values

- Comparisons with null values return the special truth value: *unknown*
- If *false* was used instead of *unknown*, then $\text{not } (A < 5)$ 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: $(\text{true and unknown}) = \text{unknown}$,
 $(\text{false and unknown}) = \text{false}$,
 $(\text{unknown and unknown}) = \text{unknown}$
 - NOT: $(\text{not unknown}) = \text{unknown}$
- In SQL “*P is unknown*” evaluates to true if predicate *P*

Division Operation

Notation:

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

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

$$r \div s = \{ t \mid t \in \Pi_{R-S}(r) \wedge \forall u \in s (tu \in r) \}$$

Where tu means the concatenation of tuples t and u to produce a single tuple

Division Operation (Cont.)

Property

Let $q = r \div s$

Then q is the largest relation satisfying $q \times s \subseteq r$

Definition in terms of the basic algebra operation

Let $r(R)$ and $s(S)$ be relations, and let $S \subseteq R$

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

To see why

$\Pi_{R-S,S}(r)$ simply reorders attributes of r

$\Pi_{R-S}(\Pi_{R-S}(r) \times s) - \Pi_{R-S,S}(r)$ gives those tuples t in

$\Pi_{R-S}(r)$ such that for some tuple $u \in s$, $tu \notin r$.

RA - Advanced Operations

branch (branch_name, branch_city, assets)

*customer (customer_name, customer_street,
customer_city)*

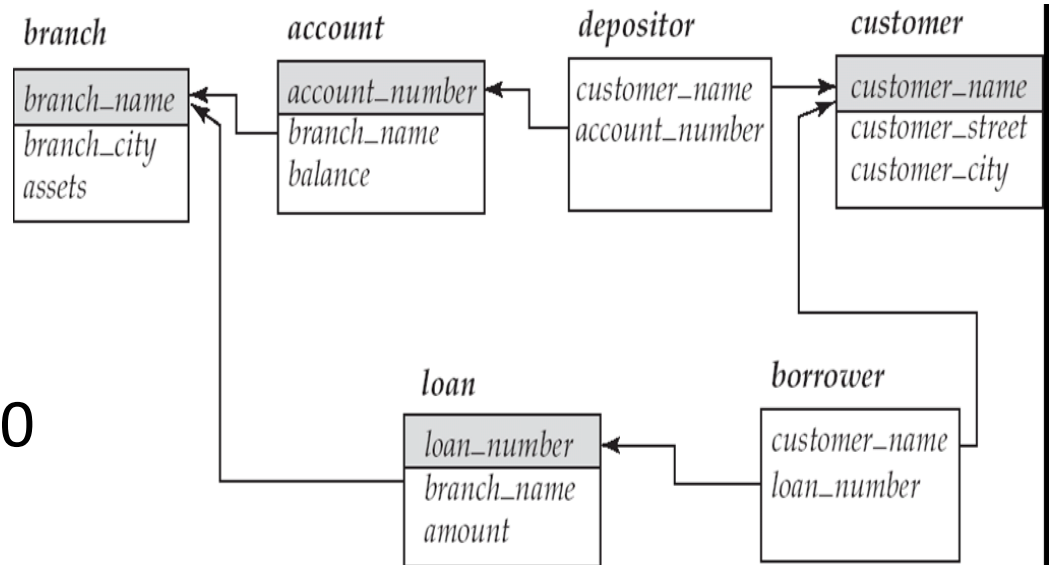
account (account_number, branch_name, balance)

loan (loan_number, branch_name, amount)

depositor (customer_name, account_number)

borrower (customer_name, loan_number)

Example Queries



Find all loans of over \$1200

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

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

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

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

$$\Pi_{customer_name} (borrower) \cup \Pi_{customer_name} (depositor)$$

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

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 is true, then y is true

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

Examples of TRC Queries



branch (branch_name, branch_city, assets)

*customer (customer_name, customer_street,
customer_city)*

account (account_number, branch_name, balance)

loan (loan_number, branch_name, amount)

depositor (customer_name, account_number)

borrower (customer_name, loan_number)

Example Queries

Find the *loan_number*, *branch_name*, and *amount* for loans of over \$1200

$$\{t \mid t \in loan \wedge t [amount] > 1200\}$$

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

$$\{t \mid \exists s \in loan (t [loan_number] = s [loan_number] \wedge s [amount] > 1200)\}$$

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

$$\{t \mid \exists s \in \text{borrower} (t [\text{customer_name}] = s [\text{customer_name}]) \vee \exists u \in \text{depositor} (t [\text{customer_name}] = u [\text{customer_name}])\}$$

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

$$\{t \mid \exists s \in \text{borrower} (t [\text{customer_name}] = s [\text{customer_name}]) \wedge \exists u \in \text{depositor} (t [\text{customer_name}] = u [\text{customer_name}])\}$$

Example Queries

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

$$\{t \mid \exists s \in \text{borrower} (t [\text{customer_name}] = s [\text{customer_name}] \\ \wedge \exists u \in \text{loan} (u [\text{branch_name}] = \text{"Perryridge"} \\ \wedge u [\text{loan_number}] = s [\text{loan_number}])))\}$$

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

$$\{t \mid \exists s \in \text{borrower} (t [\text{customer_name}] = s [\text{customer_name}] \\ \wedge \exists u \in \text{loan} (u [\text{branch_name}] = \text{"Perryridge"} \\ \wedge u [\text{loan_number}] = s [\text{loan_number}]))) \\ \wedge \text{not } \exists v \in \text{depositor} (v [\text{customer_name}] = \\ t [\text{customer_name}])\}$$

Example Queries

Find the names of all customers having a loan from the Perryridge branch, and the cities in which they live

$$\begin{aligned}
 & t \mid \exists s \in \text{loan} (s [\text{branch_name}] = \text{"Perryridge"}) \\
 \wedge & \exists u \in \text{borrower} (u [\text{loan_number}] = s [\text{loan_number}]) \\
 & \wedge t [\text{customer_name}] = u [\text{customer_name}]) \\
 \wedge & \exists v \in \text{customer} (u [\text{customer_name}] = v [\text{customer_name}] \\
 & \wedge t [\text{customer_city}] = v [\text{customer_city}]))))\}
 \end{aligned}$$

Example Queries

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

$$\begin{aligned} & t \mid \exists r \in \text{customer} (t [\text{customer_name}] = r [\text{customer_name}]) \wedge \\ & (\forall u \in \text{branch} (u [\text{branch_city}] = \text{"Brooklyn"} \Rightarrow \\ & \exists s \in \text{depositor} (t [\text{customer_name}] = s [\text{customer_name}] \\ & \wedge \exists w \in \text{account} (w[\text{account_number}] = s [\text{account_number}] \\ & \wedge (w [\text{branch_name}] = u [\text{branch_name}])))))) \end{aligned}$$

Cont...

Find the names of all employees who work for First Bank Corporation:-

- i. $\{t \mid \exists s \in works (t[person-name] = s[person-name] \wedge s[company-name] = \text{"First Bank Corporation"})\}$
- ii. $\{ \langle p \rangle \mid \exists c, s (\langle p, c, s \rangle \in works \wedge c = \text{"First Bank Corporation"})\}$

Example Queries

Find the names and cities of residence of all employees who work for First Bank Corporation:-

- i. $\{t \mid \exists r \in employee \exists s \in works (t[person-name] = r[person-name] \wedge t[city] = r[city] \wedge r[person-name] = s[person-name] \wedge s[company-name] = \text{"First Bank Corporation"})\}$
- ii. $\{ \langle p, c \rangle \mid \exists co, sa, st (\langle p, co, sa \rangle \in works \wedge \langle p, st, c \rangle \in employee \wedge co = \text{"First Bank Corporation"})\}$

Cont...

Find the names, street address, and cities of residence of all employees who work for First Bank Corporation and earn more than \$10,000 per annum:-

- i. $\{t \mid t \in \text{employee} \wedge (\exists s \in \text{works} (s[\text{person-name}] = t[\text{person-name}] \wedge s[\text{company-name}] = \text{"First Bank Corporation"} \wedge s[\text{salary}] > 10000))\}$
- ii. $\{\langle p, s, c \rangle \mid \langle p, s, c \rangle \in \text{employee} \wedge \exists co, sa (\langle p, co, sa \rangle \in \text{works} \wedge co = \text{"First Bank Corporation"} \wedge sa > 10000)\}$

Find the names of all employees in this database who live in the same city as the company for which they work:-

i. $\{t \mid \exists e \in \text{employee} \exists w \in \text{works} \exists c \in \text{company}$
 $(t[\text{person-name}] = e[\text{person-name}]$
 $\wedge e[\text{person-name}] = w[\text{person-name}]$
 $\wedge w[\text{company-name}] = c[\text{company-name}] \wedge e[\text{city}] = c[\text{city}])\}$

Domain Relational calculus- Queries



A nonprocedural query language equivalent in power to the tuple relational calculus

Each query is an expression of the form:

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

x_1, x_2, \dots, x_n represent domain variables

P represents a formula similar to that of the predicate calculus

BASIC SQL QUERY

SQL data definition; Queries in SQL: updates, views, integrity and security, relational database design.

Normal Forms: 1NF, 2NF, 3NF and BCNF

Create Table Construct

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

```
create table r (A1 D1, A2 D2, ..., An Dn,  
(integrity-constraint1),  
...,  
(integrity-constraintk))
```

r is the name of the relation

each *Ai* is an attribute name in the schema of relation *r*

Di is the data type of values in the domain of attribute *Ai*

Example:

```
create table branch  
(branch_name char(15) not null,  
branch_city char(30),  
assets integer)
```


CREATE TABLE

- In SQL2, can use the CREATE TABLE command for specifying the primary key attributes, secondary keys, and referential integrity constraints (foreign keys).
- Key attributes can be specified via the PRIMARY KEY and UNIQUE phrases

create table dept unique (dname),foreign key(mgrssn) references emp);

(Dname	Varchar(10)	NOT NULL,
Dnumber	Integer	NOT NULL,
Mgrssn	Char(9),	
Mgrstartdate	Char(9),	
Primary key	(Dnumber),	

DROP TABLE

Used to remove a relation (base table) *and its definition*

The relation can no longer be used in queries, updates, or any other commands since its description no longer exists

Example:

DROP TABLE DEPENDENT;

- 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.
- The **alter table** command can also be used to drop attributes of a relation:
alter table r drop A
- where A is the name of an attribute of relation r Dropping of attributes not supported by many databases

Referential Integrity

- Ensures that a value that appears in one relation for a given set of attributes also appears for a 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”.

- Primary and candidate keys and foreign keys can be specified as part of the SQL **create table** statement:
 - The **primary key** clause lists primary key (PK) attributes.
 - The **unique key** clause lists candidate key attributes
 - The **foreign key** clause lists foreign key (FK) attributes and the name of the relation referenced by the FK. By default, a FK references PK attributes of the referenced table.

The SQL DROP TABLE Statement



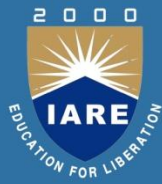
- The DROP TABLE statement is used to drop an existing table in a database.

Syntax

DROP TABLE *table_name*;

- Be careful before dropping a table. Deleting a table will result in loss of complete information stored in the table!

INSERT Into With Select Example



The Bigfoot Brewery supplier is also a customer.

Add a customer record with values from the supplier table

```
INSERT INTO Customer (FirstName, LastName, City, Country, Phone)
SELECT LEFT(ContactName, CHARINDEX(' ',ContactName) -1),
SUBSTRING(ContactName, CHARINDEX(' ',ContactName) + 1, 100),
City, Country, Phone FROM Supplier WHERE CompanyName = 'Bigfoot
Breweries';
```

UPDATE Statement



- The UPDATE statement updates data values in a database.
- UPDATE can update one or more records in a table.
- Use the WHERE clause to UPDATE only specific records.

The general syntax is:

UPDATE table-name SET column-name = value, column-name = value, ...

- To limit the number of records to UPDATE append a WHERE clause:
UPDATE table-name SET column-name = value, column-name = value, ...
WHERE condition
- UPDATE Examples
- Problem: discontinue all products in the database UPDATE Product SET IsDiscontinued = 1

```
UPDATE Product SET IsDiscontinued = 1 WHERE UnitPrice > 50
```

Integrity and Security



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

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)*

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 *account* relation, deleting the tuple that refers to the branch that was deleted
- Cascading updates are similar.

Cascading Actions in SQL (Cont.)



- 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 undone.

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***

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 attributes are present in the result of the join.
- Join type – defines how tuples in each relation that do not match any tuple in the other relation (based on the join condition) are treated.

Join Types
inner join
left outer join
right outer join
full outer join

Join Conditions
natural
on <predicate>
using (A_1, A_2, \dots, A_n)

Join - Examples

Relation *loan*

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

■ Relation *borrower*

<i>customer-name</i>	<i>loan-number</i>
Jones	L-170
Smith	L-230
Hayes	L-155

■ 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 = borrower.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

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

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 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

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
```

Views

Provide a mechanism to hide certain data from the view of certain users.
To create a view we use the command:

create view v as <query expression>

where:

<query expression> is any legal expression

The view name is represented by v

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

Integrity 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.

Various Integrity Constraints In RDB :

- Domain Integrity Constraints
- Referential Integrity Constraints
- Assertions
- Triggers
- Functional 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 (Contd.)

- Let R be a relation schema

$$\alpha \subseteq R \text{ and } \beta \subseteq R$$

- 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)

We expect this set of functional dependencies to hold:

- loan-number \rightarrow amount
- loan-number \rightarrow branch-name

but would not expect the following to hold:

loan-number \rightarrow customer-name

Introduction to schema refinement

Problems caused by redundancy

- Redundant storage
- Update Anomalies
- Insert Anomalies
- Delete Anomalies

Example:

Project (Project-id, Name, Status, Budget, Dept-Id, Dept-name, Location)

- How To address Above Problems?
- NULL values can not address completelt

Pitfalls in Relational Database Design



Relational database design requires that we find a “good” collection 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*)

<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

- Redundancy:

- 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.

Design Guidelines - RDB

What is relational database design?

- The grouping of attributes to form "good" relation schemas

Two levels of relation schemas

- The logical "user view" level
- The storage "base relation" level

Design is concerned mainly with base relations

What are the criteria for "good" base relations?

Semantics of the Attributes



GUIDELINE 1: Informally, each tuple in a relation should represent one entity or relationship instance. (Applies to individual relations and their attributes).

- Attributes of different entities (EMPLOYEEs, DEPARTMENTs, PROJECTs) should not be mixed in the same relation
- Only foreign keys should be used to refer to other entities
- Entity and relationship attributes should be kept apart as much as possible.

Redundancy - Update Anomalies



Information is stored redundantly

- Wastes storage
- Causes problems with update anomalies
 - Insertion anomalies
 - Deletion anomalies
 - Modification anomalies

Insert Anomaly - Example



Consider the relation:

EMP_PROJ(Emp#, Proj#, Ename, Pname, No_hours)

Insert Anomaly:

Cannot insert a project unless an employee is assigned to it.

Conversely

Cannot insert an employee unless an he/she is assigned to a project.

Delete Anomaly- Example

Consider the relation:

EMP_PROJ(Emp#, Proj#, Ename, Pname, No_hours)

Delete Anomaly:

- When a project is deleted, it will result in deleting all the employees who work on that project.
- Alternately, if an employee is the sole employee on a project, deleting that employee would result in deleting the corresponding project.

Redundancy - Update Anomalies



GUIDELINE 2:

- Design a schema that does not suffer from the insertion, deletion and update anomalies.
- If there are any anomalies present, then note them so that applications can be made to take them into account.

Null Values in Tuples

GUIDELINE 3:

- Relations should be designed such that their tuples will have as few NULL values as possible
- Attributes that are NULL frequently could be placed in separate relations (with the primary key)

Reasons for nulls:

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

Spurious Tuples – avoid at any cost

- Bad designs for a relational database may result in erroneous results for certain JOIN operations
- The "lossless join" property is used to guarantee meaningful results for join operations

GUIDELINE 4:

- The relations should be designed to satisfy the lossless join condition.
- No spurious tuples should be generated by doing a natural-join of any relations.

Spurious Tuples

There are two important properties of decompositions:

- a) Non-additive or losslessness of the corresponding join
- b) Preservation of the functional dependencies.

Note that:

Property (a) is extremely important and cannot be sacrificed.

Property (b) is less stringent and may be sacrificed.

Overall Database Design Process

We have assumed schema R is given

- 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.

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 (Contd.)

Let R be a relation schema

$\alpha \subseteq R$ 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)

We expect this set of functional dependencies to hold:

loan-number \rightarrow amount
loan-number \rightarrow branch-name

but would not expect the following to hold:

loan-number \rightarrow customer-name

Decomposition - Problems

To Avoid Redundancy the given relations must be Decomposed into Sub Relations.

Problems related to Decomposition

- Lossy /Superfluous information
- Dependency Preservation

Solution: Lossless Join Decomposition

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

$$r = \Pi_{R_1}(r) \bowtie \Pi_{R_2}(r)$$

Testing for Dependency Preservation



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
 $t = (result \cap R_i)^+ \cap R_i$
 $result = result \cup t$
- If *result* contains all attributes in β , then the functional dependency $\alpha \rightarrow \beta$ is preserved.

We apply the test on all dependencies in F to check if a decomposition is dependency preserving

This procedure takes polynomial time, instead of the exponential time required to compute F^+ and $(F_1 \cup F_2 \cup \dots \cup F_n)^+$

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 functional dependencies may require computing joins, which is expensive.

Example

$$R = (A, B, C)$$

$$F = \{A \rightarrow B, B \rightarrow C\}$$

- Can be decomposed in two different ways

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

- Lossless-join decomposition:

$$R_1 \cap R_2 = \{B\} \text{ and } B \rightarrow BC$$

- Dependency preserving

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

- Lossless-join decomposition:

$$R_1 \cap R_2 = \{A\} \text{ and } A \rightarrow AB$$

- Not dependency preserving

(cannot check $B \rightarrow C$ without computing $R_1 \bowtie R_2$)

Normalization of Relations

Normalization:

The process of decomposing unsatisfactory "bad" relations by breaking up their attributes into smaller relations

Normal form:

Condition using keys and FDs of a relation to certify whether a relation schema is in a particular normal form

Normalization of Relations (2)



2NF, 3NF, BCNF

based on keys and FDs of a relation schema

4NF

based on keys, multi-valued dependencies : MVDs;

5NF

based on keys, join dependencies : JDs

Additional properties may be needed to ensure a good relational design
(lossless join, dependency preservation; see Chapter 15)

Practical Use of Normal Forms

Normalization is carried out in practice so that the resulting designs are of high quality and meet the desirable properties

- The practical utility of these normal forms becomes questionable when the constraints on which they are based are *hard to understand* or to *detect*
- The database designers *need not* normalize to the highest possible normal form
(usually up to 3NF and BCNF. 4NF rarely used in practice.)

Denormalization:

The process of storing the join of higher normal form relations as a base relation—which is in a lower normal form

Keys and Attributes

- A **superkey** of a relation schema $R = \{A_1, A_2, \dots, A_n\}$ is a set of attributes S *subset-of* R with the property that no two tuples t_1 and t_2 in any legal relation state r of R will have $t_1[S] = t_2[S]$
- A **key** K is a **superkey** with the *additional property* that removal of any attribute from K will cause K not to be a superkey any more.

Keys and Attributes

- If a relation schema has more than one key, each is called a **candidate key**.
- One of the candidate keys is *arbitrarily* designated to be the **primary key**, and the others are called **secondary keys**.
- A **Prime attribute** must be a member of *some* candidate key
- A **Nonprime attribute** is not a prime attribute—that is, it is not a member of any candidate key.

First Normal Form

Disallows

- Composite Attributes
- Multivalued Attributes
- **Nested Relations**; attributes whose values for an *individual tuple* are non-atomic
- Considered to be part of the definition of a relation
- Most RDBMSs allow only those relations to be defined that are in First Normal Form

Normalization into 1NF

(a)

DEPARTMENT

Dname	<u>Dnumber</u>	Dmgr_ssn	Dlocations

(b)

DEPARTMENT

Dname	<u>Dnumber</u>	Dmgr_ssn	Dlocations
Research	5	333445555	{Bellaire, Sugarland, Houston}
Administration	4	987654321	{Stafford}
Headquarters	1	888665555	{Houston}

(c)

DEPARTMENT

Dname	<u>Dnumber</u>	Dmgr_ssn	<u>Dlocation</u>
Research	5	333445555	Bellaire
Research	5	333445555	Sugarland
Research	5	333445555	Houston
Administration	4	987654321	Stafford
Headquarters	1	888665555	Houston

Second Normal Form (1)

- Uses the concepts of **FDs, primary key**
- Definitions

Prime attribute: An attribute that is member of the primary key K

Full functional dependency: a FD $Y \rightarrow Z$ where removal of any attribute from Y means the FD does not hold any more

Examples:

$\{SSN, PNUMBER\} \rightarrow HOURS$ is a full FD since neither $SSN \rightarrow HOURS$ nor $PNUMBER \rightarrow HOURS$ hold

$\{SSN, PNUMBER\} \rightarrow ENAME$ is not a full FD (it is called a partial dependency) since $SSN \rightarrow ENAME$ also holds

Second Normal Form (2)

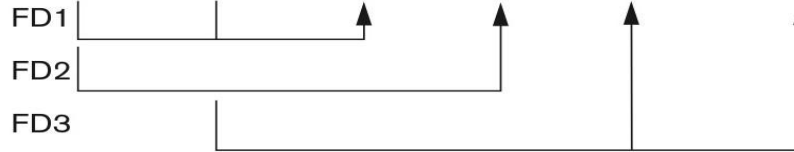
- A relation schema R is in **second normal form (2NF)** if every non-prime attribute A in R is fully functionally dependent on the primary key
- R can be decomposed into 2NF relations via the process of 2NF normalization or “second normalization”

Normalizing into 2NF and 3NF

(a)

EMP_PROJ

<u>Ssn</u>	<u>Pnumber</u>	Hours	Ename	Pname	Plocation
------------	----------------	-------	-------	-------	-----------



2NF Normalization

EP1

<u>Ssn</u>	<u>Pnumber</u>	Hours
------------	----------------	-------



EP2

<u>Ssn</u>	Ename
------------	-------



EP3

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



(b)

EMP_DEPT

Ename	<u>Ssn</u>	Bdate	Address	Dnumber	Dname	Dmgr_ssn
-------	------------	-------	---------	---------	-------	----------



3NF Normalization

ED1

Ename	<u>Ssn</u>	Bdate	Address	Dnumber
-------	------------	-------	---------	---------



ED2

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



Third Normal Form (1)

Definition:

Transitive functional dependency: a FD $X \rightarrow Z$ that can be derived from two FDs $X \rightarrow Y$ and $Y \rightarrow Z$

Examples:

SSN \rightarrow DMGRSSN is a **transitive** FD

Since SSN \rightarrow DNUMBER and DNUMBER \rightarrow DMGRSSN hold

SSN \rightarrow ENAME is **non-transitive**

Since there is no set of attributes X where SSN \rightarrow X and X \rightarrow ENAME

Third Normal Form (2)

A relation schema R is in **third normal form (3NF)** if it is in 2NF *and* no non-prime attribute A in R is transitively dependent on the primary key

R can be decomposed into 3NF relations via the process of 3NF normalization

NOTE:

- In $X \rightarrow Y$ and $Y \rightarrow Z$, with X as the primary key, we consider this a problem only if Y is not a candidate key.
- When Y is a candidate key, there is no problem with the transitive dependency .

E.g., Consider EMP (SSN, Emp#, Salary).

Here, $SSN \rightarrow Emp\# \rightarrow Salary$ and $Emp\#$ is a candidate key

Normal Forms

- 1st normal form
All attributes depend on **the key**
- 2nd normal form
All attributes depend on **the whole key**
- 3rd normal form
All attributes depend on **nothing but the key**

General Definition of 2NF

A relation schema R is in **second normal form (2NF)** if every non-prime attribute A in R is fully functionally dependent on *every* key of R

FD

County_name \rightarrow Tax_rate violates 2NF.

So second normalization converts LOTS into

LOTS1 (Property_id#, County_name, Lot#, Area, Price)

LOTS2 (County_name, Tax_rate)

Third Normal Form

DEFINITION of 3NF:

- A relation schema R is in **third normal form (3NF)** if every non-prime attribute in R meets both of these conditions:
- It is fully functionally dependent on every key of R
- It is non-transitively dependent on every key of R

Note that stated this way, a relation in 3NF also meets the requirements for 2NF.

BCNF (Boyce-Codd Normal Form)



- A relation schema R is in **Boyce-Codd Normal Form (BCNF)** if whenever an **FD $X \rightarrow A$** holds in R , then **X is a superkey** of R
- Each normal form is strictly stronger than the previous one
 - Every 2NF relation is in 1NF
 - Every 3NF relation is in 2NF
 - Every BCNF relation is in 3NF
- There exist relations that are in 3NF but not in BCNF
- Hence BCNF is considered a **stronger form of 3NF**
- The goal is to have each relation in BCNF (or 3NF)

BCNF - Example

TEACH

Student	Course	Instructor
Narayan	Database	Mark
Smith	Database	Navathe
Smith	Operating Systems	Ammar
Smith	Theory	Schulman
Wallace	Database	Mark
Wallace	Operating Systems	Ahamad
Wong	Database	Omiecinski
Zelaya	Database	Navathe
Narayan	Operating Systems	Ammar

BCNF by Decomposition (1)

- Two FDs exist in the relation TEACH:
 - fd1: { student, course} -> instructor
 - fd2: instructor -> course{student, course} is a candidate key for this relation and that the dependencies shown follow the pattern in Figure 14.13 (b).
So this relation is in 3NF *but not in* BCNF
- A relation **NOT** in BCNF should be decomposed so as to meet this property, while possibly forgoing the preservation of all functional dependencies in the decomposed relations.

Multi valued Dependencies

- There are database schemas in BCNF that do not seem to be sufficiently normalized
- Consider a database *classes(course, teacher, 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 instructor, and the set of books, all of which are required for the course (no matter who teaches it).

Multi valued Dependencies (Contd.)

<i>course</i>	<i>teacher</i>	<i>book</i>
Physics101	Green	Mechanics
Physics101	Green	Optics
Physics101	Brown	Mechanics
Physics101	Brown	Optics
Math301	Green	Mechanics
Math301	Green	Vectors
Math301	Green	Geometry

- 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 classes
- Insertion anomalies – i.e., if Sara is a new teacher that can teach the following two tuples need to be inserted

(Physics101, Sara, Mechanics)

(database, Sara, Optics)

Multi_valued Dependencies (MVDs)

- Let R 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:

$$\begin{aligned} t_1[\alpha] &= t_2[\alpha] = t_3[\alpha] = t_4[\alpha] \\ t_3[\beta] &= t_1[\beta] \\ t_3[R - \beta] &= t_2[R - \beta] \\ t_4[\beta] &= t_2[\beta] \\ t_4[R - \beta] &= t_1[R - \beta] \end{aligned}$$

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 multivalued dependency, we can construct a relations r' that does satisfy the multivalued dependency by adding tuples to r

Theory of MVDs



- From the definition of multi_valued dependency, we can derive the following rule:

If $\alpha \rightarrow \beta$, then $\alpha \twoheadrightarrow \beta$

That is, every functional dependency is also a multi_valued dependency

- The **closure** D^+ of D is the set of all functional and multi_valued dependencies logically implied by D .

We can compute D^+ from D , using the formal definitions of functional dependencies and multi_valued dependencies.

We can manage with such reasoning for very simple multi_valued dependencies, which seem to be most common in practice

For complex dependencies, it is better to reason about sets of dependencies using a system of inference rules .

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

Join Dependencies and 5NF

Definition:

- A **join dependency (JD)**, denoted by $JD(R_1, R_2, \dots, R_n)$, specified on relation schema R , specifies a constraint on the states r of R .
- The constraint states that every legal state r of R should have a non-additive join decomposition into R_1, R_2, \dots, R_n ; that is, for every such r we have

$$* (\pi_{R_1}(r), \pi_{R_2}(r), \dots, \pi_{R_n}(r)) = r$$

Note: an MVD is a special case of a JD where $n = 2$.

- A join dependency $JD(R_1, R_2, \dots, R_n)$, specified on relation schema R , is a **trivial JD** if one of the relation schemas R_i in $JD(R_1, R_2, \dots, R_n)$ is equal to R .

Join Dependencies & 5NF Contd..



Definition:

- A relation schema R is in **fifth normal form (5NF)** (or **Project-Join Normal Form (PJNF)**) with respect to a set F of functional, multivalued, and join dependencies if, for every nontrivial join dependency $JD(R_1, R_2, \dots, R_n)$ in F^+ (that is, implied by F), every R_i is a superkey of R .
- Discovering join dependencies in practical databases with hundreds of relations is next to impossible. Therefore, 5NF is rarely used in practice.

UNIT - IV

Transaction processing:

Introduction, need for concurrency control, desirable properties of transaction, schedule and recoverability, serializability and schedules

Transaction

Transaction:

A **transaction** is a *unit* of program execution that accesses and possibly updates various data items.

E.g., 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)

ACID Properties

- A **transaction** is a unit of program execution that accesses and possibly updates various data items. To preserve the integrity of data the database system must ensure:
- **Atomicity.** Either all operations of the transaction are properly reflected in the database or none are.
- **Consistency.** Execution of a transaction in isolation preserves the consistency of the database.

ACID Properties(cont.)

- **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.

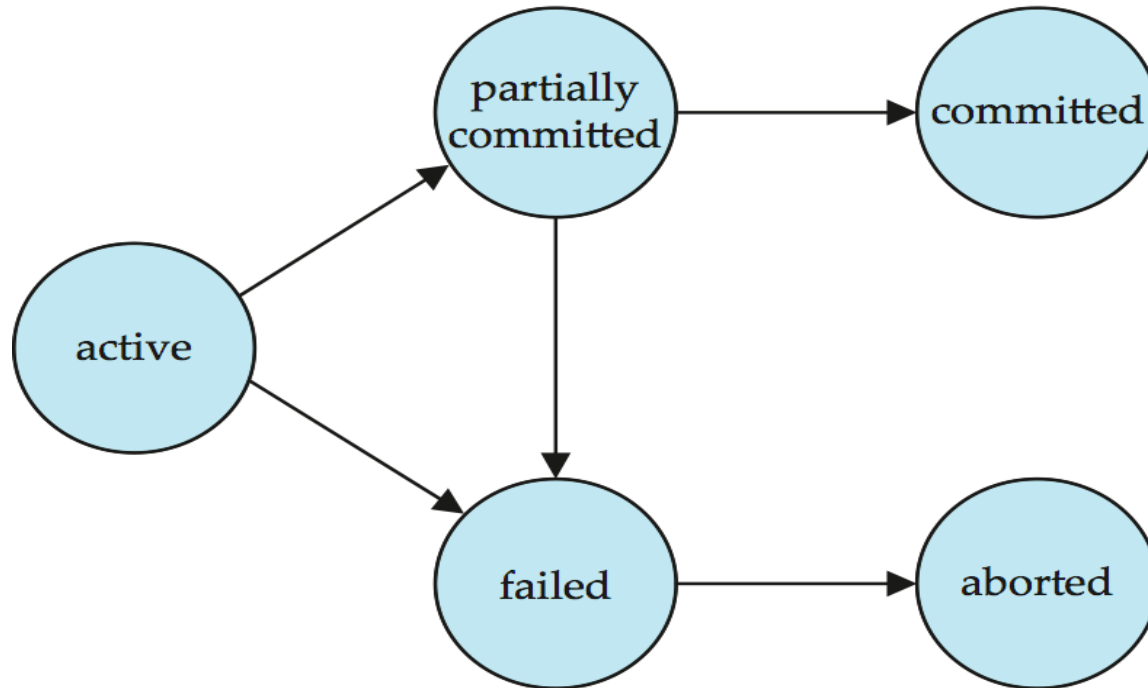
Transaction State

- **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.

Transaction State(cont.)

- **Aborted** – after the transaction has been rolled back and the database restored to its state prior to the start of the transaction.
Two options after it has been aborted:
 - Restart the transaction
 - can be done only if no internal logical error
 - Kill the transaction
- **Committed** – after successful completion.

Transaction State (Cont.)



Example of transaction

Transfer £50 from account A to account B

Read(A)

$A = A - 50$

Write(A)

Read(B)

$B = B + 50$

Write(B)

- **Atomicity** - shouldn't take money from A without giving it to B
- **Consistency** - money isn't lost or gained
- **Isolation** - other queries shouldn't see A or B change until completion
- **Durability** - the money does not go back to A

Concurrent Executions

Multiple transactions are allowed to run concurrently in the system.

Advantages are:

- **Increased processor and disk utilization**, leading to better transaction *throughput*
E.g. 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.

Concurrent Executions(cont.)



Concurrency control schemes – mechanisms to achieve isolation

That is, to control the interaction among the concurrent transactions in order to prevent them from destroying the consistency of the database

Recoverability

A schedule is said to be recoverable if a failed transaction is undone.

- If a transaction T_i fails we need to undo the effect of this transaction to ensure the atomicity property.
- In a concurrent execution it is necessary to ensure that transaction T_j that is dependent on T_i is also aborted.

Recoverable Schedules

Recoverable schedule — if a transaction T_j reads a data item previously written by a transaction T_i , then the commit operation of T_i **must** appear before the commit operation of T_j .

The following schedule is not recoverable if T_9 commits immediately after the read(A) operation.

T_8	T_9
read (A)	
write (A)	
	read (A)
	commit
read (B)	

If T_8 should abort, T_9 would have read (and possibly shown to the user) an inconsistent database state. Hence, database must ensure that schedules are recoverable.

Cascading Rollbacks

Cascading rollback – a single transaction failure leads to a series of transaction rollbacks. Consider the following schedule where none of the transactions has yet committed (so the schedule is recoverable)

T_{10}	T_{11}	T_{12}
read (A) read (B) write (A)	read (A) write (A)	read (A)
abort		

If T_{10} fails, T_{11} and T_{12} must also be rolled back.

Can lead to the undoing of a significant amount of work

Concurrency Control



A database must provide a mechanism that will ensure that all possible schedules are both:

- Conflict serializable.
- Recoverable and preferably cascadeless

Concurrency-control schemes tradeoff between the amount of concurrency they allow and the amount of overhead that they incur

Testing a schedule for serializability *after* it has executed is a little too late!

- Tests for serializability help us understand why a concurrency control protocol is correct

Goal – to develop concurrency control protocols that will assure serializability.

Conflict Serializability (Cont.)

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

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

Schedule 3

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

Schedule 6

Conflict Serializability (Cont.)

Example of a schedule that is not conflict serializable:

T_3	T_4
read (Q)	
	write (Q)
write (Q)	

We are unable to swap instructions in the above schedule to obtain either the serial schedule $\langle T_3, T_4 \rangle$, or the serial schedule $\langle T_4, T_3 \rangle$.

View Serializability



Let S and S' be two schedules with the same set of transactions. And S' are View equivalent if the following three conditions are met:

1. Initial Read
2. Write-read
3. Final write

View equivalence is purely based on reads and writes alone.

View Serializability(cont.)

A schedule S is view serializable if it is equivalent to a serial schedule.
 Every conflict serializable schedule is also a view serializable.
 Every view serializable schedule is not conflict serializable has blind writes.

T_3	T_4	T_6
read(Q)	write(Q)	
write(Q)		write(Q)

Above example is a view serializable but not conflict serializable.

Testing for Conflict Serializability



In order to determine a conflict serializable we need to construct a directed graph called precedence graph .

It is represented as $G=(V,E)$

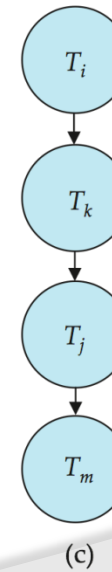
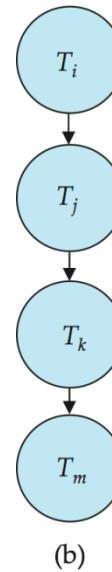
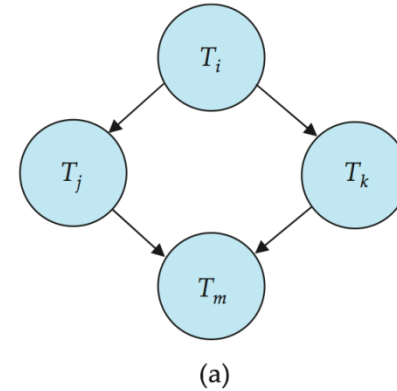
V-consists of transactions

E-consists of set of edges $T_i \rightarrow T_j$ for which one of the three conditions.

1. T_i executes **Write(Q)** before T_j executes **Read(Q)**.
2. T_i executes **Read(Q)** before T_j executes **Write(Q)**
3. T_i executes **Write(Q)** before T_j executes **Write(Q)**

Conflict Serializability

For example, a serializability order for the schedule (a) would be one of either (b) or (c)



Concurrency control

Types of locks: Two phases locking, deadlock, timestamp based concurrency control, recovery techniques, concepts, immediate update, deferred update, shadow paging.

Concurrency Control

- A database must provide a mechanism that will ensure that all possible schedules are both:
 - Conflict serializable.
 - Recoverable 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 amount of overhead that they incur.
- Testing a schedule for serializability *after* it has executed is a little too late!
- Tests for serializability help us understand why a concurrency control protocol is correct
- **Goal** – to develop concurrency control protocols that will assure serializability.

Weak Levels of Consistency



- Some applications are willing to live with weak levels of consistency, allowing schedules that are not serializable

E.g., a read-only transaction that wants to get an approximate total balance of all accounts

E.g., database statistics computed for query optimization can be approximate (why?)

Such transactions need not be serializable with respect to other transactions tradeoff accuracy for performance

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.
- **Read committed** — only committed records can be read, but successive reads of record may return different (but committed) values.
- **Read uncommitted** — even uncommitted records may be read.

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.
- In almost all database systems, by default, every SQL statement also commits implicitly if it executes successfully
- Implicit commit can be turned off by a database directive
E.g. in JDBC, `connection.setAutoCommit(false);`

Introduction to Locks

Introduction:

Locking is necessary in a concurrent environment to assure that one process should not retrieve or update a record which another process is updating. Failure to this would result in inconsistent and corrupted data.

Types of Locks

There are various modes to lock data items. They are:

- **Shared(S)**: If a transaction T_i has shared mode lock on data item Q then T_i can read but not write Q . **lock-S(Q)** instruction is used in shared mode.
- **Exclusive(X)**: If a transaction has obtained an exclusive mode lock on data item Q , then T_i can perform both read and write. **lock-X(Q)** instruction is used to lock in exclusive mode.

Lock-compatibility matrix

A lock is a mechanism to control concurrent access to a data item. Lock requests are made to concurrency-control manager. Transaction can proceed only after request is granted.

Lock-compatibility matrix

	S	X
S	true	false
X	false	false

Transaction Performing Locking:

<i>T1:</i>	<i>T2:</i>	<i>T3:</i>	<i>T4:</i>
lock-X(B); read (B); B:=B-50; write(B); unlock(B); lock-X(A); read (A); A:=A+50; write(A); unlock(A);	lock-S(A); read (A); unlock(A); lock-S(B); read (B); unlock(B); display (A+B)	lock-X(B); read (B); B:=B-50; write(B); lock-X(A); read (A); A:=A+50; write(A); unlock(B); unlock(A);	lock-S(A); read (A); lock-S(B); read (B); display (A+B); unlock(A); unlock(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.

Two-Phase Locking (2PL)

- A protocol that guarantees **serializability** but does not prevent deadlocks.
- A transaction obeying the **two-phase locking protocol (2PL)** if before operating on any object, the transaction first acquires a lock on that object (Growing Phase/locking phase) after releasing a lock, the transaction never acquires any more locks (Shrinking Phase/unlocking phase).
- 2PL can be shown to be conflict serializable in the order of '**lock point**'

Strict 2PL

- **Strict-2PL:** Transaction holds X-locks till it commit/aborts. After commit/aborted it releases the lock.
- Another variants of Two-Phase locking is Rigorous and conservative 2PL.

Variants of Two-Phase locking



- **Rigorous 2PL:** T holds S|X locks till it commit | Aborts transactions can be serialized in the order in which they commit.
- **Conservative 2PL:** Transaction gets all locks in an atomic manner i.e. no deadlocks.

Timestamp Ordering

Timestamp:

- a number generate by system.
- ticks of the computer's internal clock.
- no two transactions can have the same timestamp.

Timestamp-ordering Protocol



Case 1: Suppose that transaction T_i issues $\text{read}(Q)$.

- If $\text{TS}(T_i) \leq \mathbf{W}\text{-timestamp}(Q)$, then T_i needs to read a value of Q that was already overwritten. Hence, T_i should read value before W -timestamp. Therefore **read** operation is rejected, and T_i is rolled back.
- If $\text{TS}(T_i) \geq \mathbf{W}\text{-timestamp}(Q)$, then the **read** operation is executed, and $\text{R-timestamp}(Q)$ is set to $\mathbf{max}(\text{R-timestamp}(Q), \text{TS}(T_i))$. Suppose that transaction T_i issues **write**(Q).

Timestamp-ordering

Case: 2 Suppose T_i issues $\text{write}(Q)$.

- If $\text{TS}(T_i) < \text{R-timestamp}(Q)$, since it has read-write conflict the **write** operation is rejected, and T_i is rolled back.
- If $\text{TS}(T_i) < \text{W-timestamp}(Q)$, since it has write-write conflict the **write** operation is rejected, and T_i is rolled back. Otherwise, the **write** operation is executed, and $\text{W-timestamp}(Q)$ is set to $\text{TS}(T_i)$.
- If $\text{TS}(T_i) > \text{R-timestamp}(Q)$ and $\text{TS}(T_i) > \text{W-timestamp}(Q)$, since all read and write operations done before timestamp write operation is granted. Therefore w-timestamp need to be updated with max of w-TS & $\text{TS}(T_i)$.

Thomas's write rule

Thomas's write rule:

This rule states that if $TS(T_i) < W\text{-timestamp}(Q)$ then the operation is rejected & T_i is rolled back. Timestamp ordering rules can be modified to make the schedule view serializable. Instead of making T_i rolled back, the write operation itself is ignored.

Thomas's write rule(contd.)

Consider the given transactions

T1	T2
Read(Q)	
Write(Q)	Write(Q)

- For the condition $TS(T_i) < W\text{-timestamp}(Q)$ write of T_2 is having largest W-timestamp.
- In case of T_1 and T_2 write operation is updated by W-timestamp(Q) to $TS(T_1)$.
- Under the thomas's write rule, write(Q) on T_1 would be ignored.

Validation based protocol

Validation based protocol:

- No checking is done while the transaction is executing.
- Each transaction executes in three phases in its lifetime.

Read phase:

During this phase, the system executes transaction T_i . It reads the values of various data items and writes on temporary local variables without updating the actual database.

Validation phases

- **Validation phase:** Transaction T_i performs a validation test to determine the operation of read phase without violating the serializability.
- **Write phase:** If Transaction T_i succeeds in validation then actual updates are applied to the database otherwise the system rolls back T_i .

Validation based protocol(contd.)



To perform the validation test, we need to know when the various phases of transaction T_i took place. Therefore associate three different timestamps with transaction T_i . The validation scheme is called as optimistic concurrency-control.

Three timestamps of validation are:

Start (T): start of execution (T_i)

Validation (T): T_i finished its read phase & started its validation phase.

Finish (T): Time when T_i finished its write phase.

Serializability order by the timestamp-ordering technique is determined by using the value $TS(T_i) = \text{validation}(T_i)$.

Validation Test

Validation Test:

- For transaction $TS(T_i) < TS(T_j)$, one of the following condition must hold
- $Finish(T_i) < Start(T_j)$: Since T_i completes its execution before T_j started the serializability order is maintained.
- $Start(T_i) < Finish(T_i) < Validation(T_j)$:The validation phase of T_j should occur after T_i finishes.
- It ensures that writes of T_i & T_j do not overlap. Write (T_i) do not effect read(T_j) hence serializability order is maintained.

Recovery System

Recovery system is an integral part of database management system that can restore the database to the consistent state before failure. The failures are categorized as failure that does not result in loss of information and effects with loss of information.

Failures

Failures are classified as:

1. Transaction failure

The transaction may fail due to two errors. They are:

Logical error: The transaction further cannot continue with normal execution because of internal conditions as data not found, invalid input data, overflow or exceeded resource limits.

System error: The transaction further cannot continue with undesirable state like deadlock conditions.

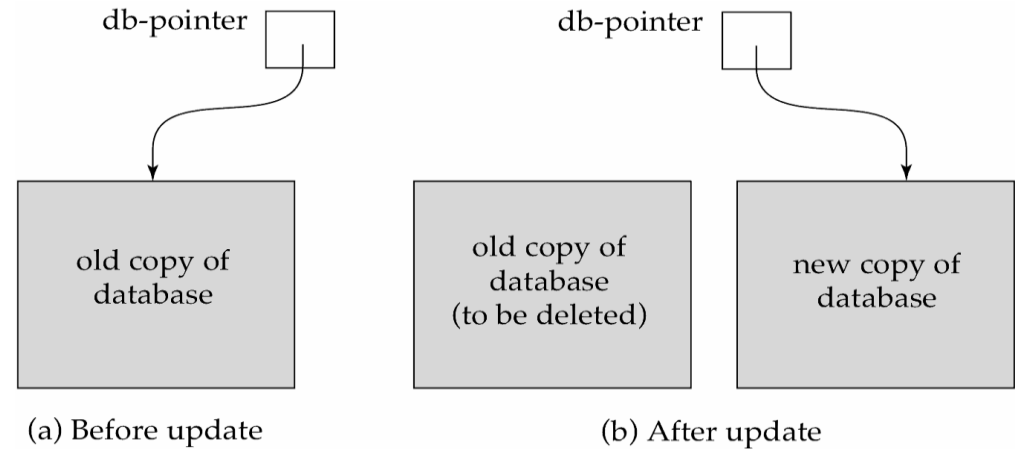
2. System crash: System crash causes loss of the content of volatile storage.

The reasons for this are: hardware problem; bug in the software or database software.

3. Disk failure: Disk crash leads to loss of information, which is due to failure due to data transfer or head crash. To recover from this failure backup on other disks, tapes.

Recovery From Failure

- Two approaches for recovery:
- Log-based recovery
- Shadow-paging



Recovery Algorithms

- Recovery algorithms are techniques to ensure database consistency and transaction atomicity and durability even with failures. Recovery algorithms have two parts:
- Actions taken during normal transaction processing to ensure enough information exists to recover from failures
- Actions taken after a failure to recover the database contents to a state that ensures atomicity, consistency and durability.

Two approaches using logs

- **Deferred database modification**
- **Immediate database modification**

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

Log record has 3 fields:

- **Transaction Identifier:** Unique identifier of the transaction that performed write operation.
- **Data item identifier:** Unique identification of the data item written
- **Old value:** Value of the item prior to the write
- **New value:** Value of the item after write transaction

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.

$\langle T_i \text{ start} \rangle$ transaction T_i started.

- **write**(X) operation results in a log record :

$\langle T_i, X, V \rangle$ where V is the new value for X

Note: old value is not needed for this scheme

- The write is not performed on X at this time, but is deferred.

When T_i partially commits,

$\langle T_i \text{ commit} \rangle$ is written to the log

Finally, the log records are read and used to actually execute the previously deferred writes. During recovery after a crash, a transaction needs to be **redo** 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

$\langle \text{redo } T_i \rangle$ sets the value of all data items updated by the transaction to the new values.

Example

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):

Let T_0 be a transaction that transfers ₹50 from Account A to B. T_1 be a transaction that withdraws ₹100 from Account C. Initially A, B and C have ₹1000, ₹2000 and ₹700 respectively.

T_0 : read (A) A: - A - 50 Write(A) read (B) B:- B + 50 write (B)	T_1 : read (C) C:-C- 100 write (C)
---	---

Log Record entries

Portion of database log for T_0 and T_1	Log	database
<T0 start>	< T0 start>	
<T0, A, 950>	< T0, A, 950>	
<T0, B, 2050>	< T0, B, 2050>	
<T0, commit>	< T0, commit>	
<T1 start>		A=950
<T1, C, 600>		B=2050
<T1, commit>	<T1, start>	
	< T1, C, 600>	
	< T1,commit>	
		C=600

Log updates at three instances of time



(a)	(b)	(c)
<T0 start>	< T0 start>	< T0 start>
<T0, A, 950>	< T0, A, 950>	< T0, A, 950>
<T0, B, 2050>	< T0, B, 2050>	< T0, B, 2050>
	< T0, commit>	< T0, commit>
	<T1, start>	<T1, start>
	< T1, C, 600>	< T1, C, 600>
		< T1,commit>

Recovery actions

<p>Case 1: Shown in (a)</p>	<p>Crash occurs just after log record for Write(B) of transaction T_0.</p>	<ul style="list-style-type: none"> • No redo action required due to no commit in log. • The accounts A and B remains with initial values. • Incomplete transaction T_0 can be deleted from the log
<p>Case 2: Shown in (b)</p>	<p>Crash occurs just after log record for Write(C) of transaction T_1.</p>	<ul style="list-style-type: none"> • Redo(T_0) is performed due to commit record ($\langle T_0, \text{commit} \rangle$) in log. • The accounts A and B has with ₹ 950 and ₹ 2050 respectively. • Incomplete transaction T_1 can be deleted from the log
<p>Case 3: Shown in (c)</p>	<p>Crash occurs just after log record ($\langle T_1, \text{commit} \rangle$) is written in stable storage.</p>	<ul style="list-style-type: none"> • The accounts A, B and C has with ₹ 950, ₹ 2050 and ₹ 600 respectively.

Immediate Database Modification



- The **immediate database 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. 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 log records corresponding to items *B* must be flushed to stable storage.
- Output of updated blocks can take place at any time before or after transaction commit
- Order in which blocks are output can be different from the order in which they are written.

Recovery procedure operations

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
- Transaction T_i needs to be **undone** if the log contains the record $\langle T_i \text{ start} \rangle$, but does not contain the record $\langle T_i \text{ commit} \rangle$.
- Transaction T_i needs to be **redone** if the log contains both the record $\langle T_i \text{ start} \rangle$ and the record $\langle T_i \text{ commit} \rangle$.
- Undo operations are performed first, then redo operations.

Example

Let accounts A, B and C initially has ₹ 1000, ₹ 2000 and ₹ 700 respectively. The log entry of both the transactions are:

Log	Database
<T ₀ start> <T ₀ , A, 1000, 950> <T ₀ , B, 2000, 2050>	
	A = 950 B = 2050
<T ₀ commit> <T ₁ start> <T ₁ , C, 700, 600>	
	C = 600
<T ₁ commit>	

Recovery Actions

Failure time slots	(a)	(b)	(c)
Log	<p><T₀ start> <T₀, A, 1000, 950> <T₀, B, 2000, 2050></p>	<p><T₀ start> <T₀, A, 1000, 950> <T₀, B, 2000, 2050> <T₀ commit> <T₁ start> <T₁, C, 700, 600></p>	<p><T₀ start> <T₀, A, 1000, 950> <T₀, B, 2000, 2050> <T₀ commit> <T₁ start> <T₁, C, 700, 600> <T₁ commit></p>
Recovery Scheme	Undo(T ₀)	Redo(T ₀) Undo(T ₀)	Redo(T ₀) Redo(T ₁)
Recovery Action	Account A and B with ₹1000 and ₹2000	Account A, B and C with ₹950, ₹2050 and ₹700.	Account A, B and C with ₹950, ₹2050 and ₹600.

Example

- 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):
- Let T_0 be a transaction that transfers ₹50 from Account A to B. T_1 be a transaction that withdraws ₹100 from Account C. Initially A, B and C have ₹1000, ₹2000 and ₹700 respectively.

T_0 : read (A) A: - A - 50 Write(A) read (B) B:- B + 50 write (B)	T_1 :
	read (C) C:-C- 100 write (C)

Log Record entries

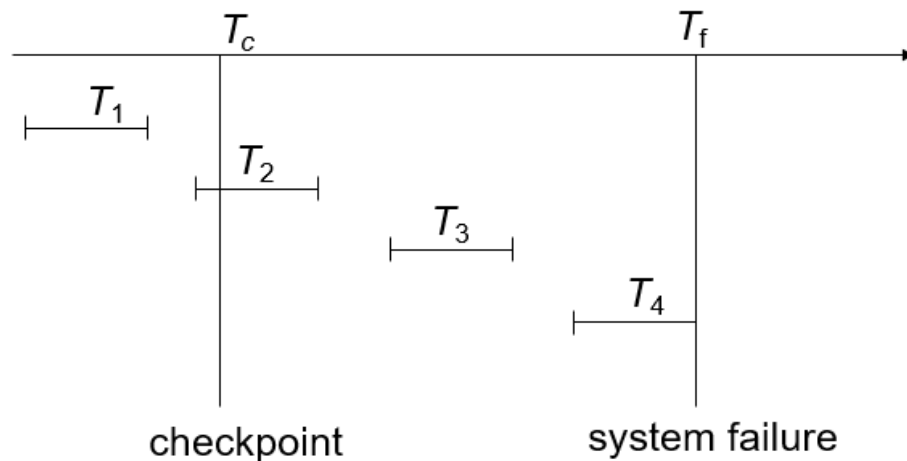
Portion of database log for T_0 and T_1	Log	database
<T0 start>	< T0 start>	
<T0, A, 950>	< T0, A, 950>	
<T0, B, 2050>	< T0, B, 2050>	
<T0, commit>	< T0, commit>	
<T1 start>		A=950
<T1, C, 600>		B=2050
<T1, commit>	<T1, start>	
	< T1, C, 600>	
	< T1,commit>	
		C=600

Log updates at three instances of time

(a)	(b)	(c)
<T0 start>	< T0 start>	< T0 start>
<T0, A, 950>	< T0, A, 950>	< T0, A, 950>
<T0, B, 2050>	< T0, B, 2050>	< T0, B, 2050>
	< T0, commit>	< T0, commit>
	<T1, start>	<T1, start>
	< T1, C, 600>	< T1, C, 600>
		< T1,commit>

Example

Example: Let $T_1, T_2, T_3,$ and T_4 are transaction recorded in log. T_c is checkpoint and T_t is the failure occurred.



- T_1 can be ignored (updates already output to disk due to checkpoint)
- T_2 and T_3 redone.
- T_4 undone

Shadow paging

- Shadow paging is an alternative to log-based recovery; this scheme is useful if transactions execute serially
- 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 identical. Only current page table is used for data item accesses during execution of the transaction.
- Whenever any page is about to be written for the first time:
 - Copy of this page is made onto an unused page.
 - Current page table is then made to point to the copy.
 - Update is performed on the copy

Example of Shadow paging

Example of Shadow Paging

Shadow and current page tables after write to page 4

