



Chapter 1: Introduction

Database System Concepts, 5th Ed.

©Silberschatz, Korth and Sudarshan
See www.db-book.com for conditions on re-use





Chapter 1: Introduction

- Purpose of Database Systems
- View of Data
- Database Languages
- Relational Databases
- Database Design
- Object-based and semistructured databases
- Data Storage and Querying
- Transaction Management
- Database Architecture
- Database Users and Administrators
- Overall Structure
- History of Database Systems





Database Management System (DBMS)

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





Purpose of Database Systems

- In the early days, database applications were built directly on top of file systems
- Drawbacks of using file systems to store data:
 - Data redundancy and inconsistency
 - ▶ Multiple file formats, duplication of information in different files
 - Difficulty in accessing data
 - ▶ Need to write a new program to carry out each new task
 - Data isolation — multiple files and formats
 - Integrity problems
 - ▶ Integrity constraints (e.g. account balance > 0) become “buried” in program code rather than being stated explicitly
 - ▶ Hard to add new constraints or change existing ones





Purpose of Database Systems (Cont.)

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





Levels of Abstraction

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

```
type customer = record
    customer_id : string;
    customer_name : string;
    customer_street : string;
    customer_city : integer;
end
```

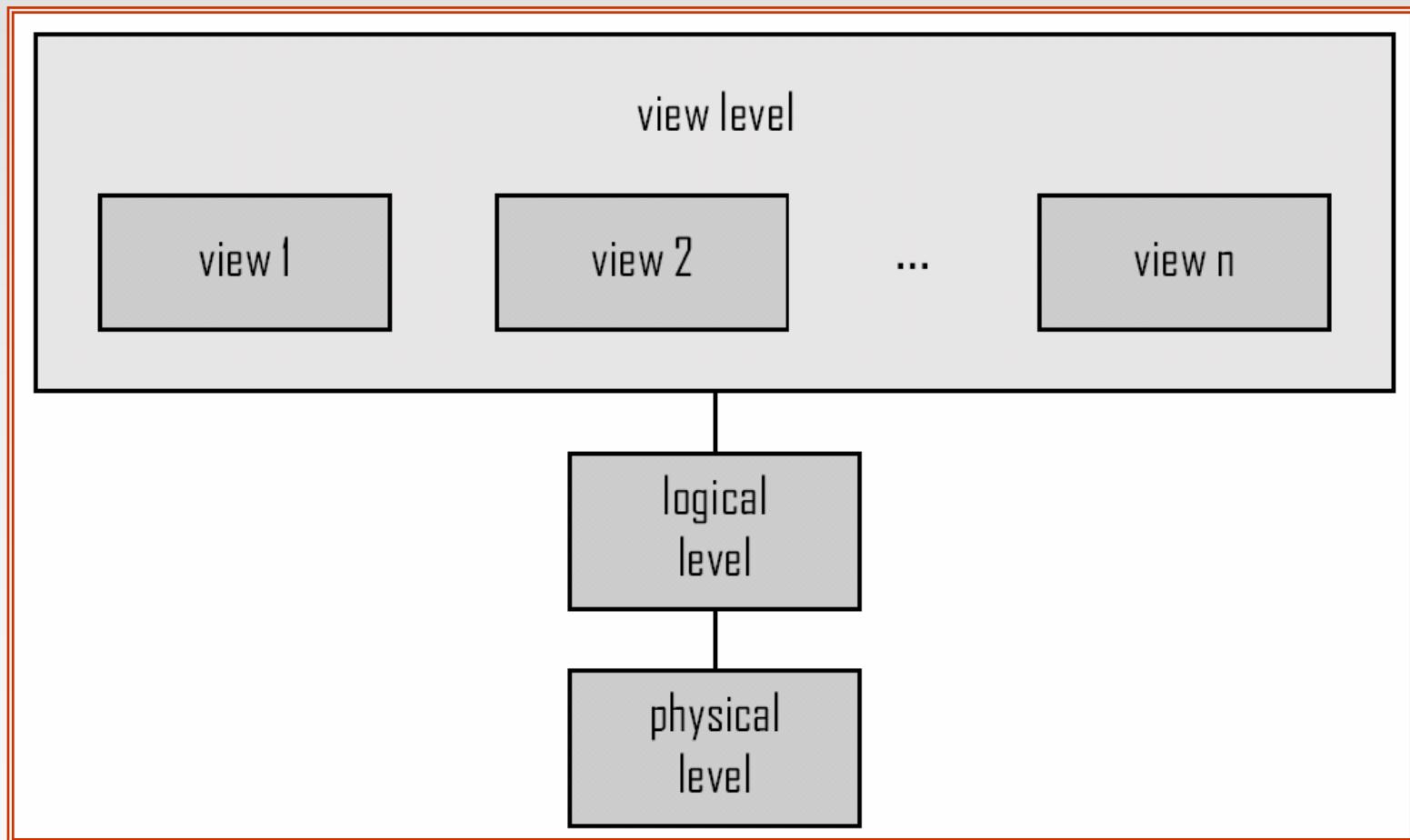
- **View level:** A way to hide: (a) details of data types and (b) information (such as an employee's salary) for security purposes.





View of Data

An architecture for a database system





Instances and Schemas

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





Data Models

- A collection of tools for describing
 - Data
 - Data relationships
 - Data semantics
 - Data constraints
- Relational model
- Entity-Relationship data model (mainly for database design)
- Object-based data models (Object-oriented and Object-relational)
- Semistructured data model (XML)
- Other older models:
 - Network model
 - Hierarchical model





Data Manipulation Language (DML)

- Language for accessing and manipulating the data organized by the appropriate data model
 - DML also known as **query language**
- Two classes of languages
 - **Procedural** – user specifies what data is required and how to get those data
 - **Declarative (nonprocedural)** – user specifies what data is required without specifying how to get those data
- SQL is the most widely used query language





Data Definition Language (DDL)

- Specification notation for defining the database schema
Example: `create table account (`
 `account-number char(10),`
 `balance integer)`
- DDL compiler generates a set of tables stored in a *data dictionary*
- Data dictionary contains metadata (i.e., data about data)
 - Database schema
 - Integrity constraints
 - ▶ Domain constraints
 - ▶ Referential integrity (**references** constraint in SQL)
 - ▶ Assertions
 - Authorization
- Data storage and definition language
 - Specifies the storage structure and access methods used





Relational Databases

- A relational database is based on the relational data model
- Data and relationships among the data is represented by a collection of tables
- Includes both a DML and a DDL
- Most commercial relational database systems employ the **SQL** query language.





Relational Model

- Example of tabular data in the relational model

Attributes

<i>customer_id</i>	<i>customer_name</i>	<i>customer_street</i>	<i>customer_city</i>	<i>account_number</i>
192-83-7465	Johnson	12 Alma St.	Palo Alto	A-101
192-83-7465	Johnson	12 Alma St.	Palo Alto	A-201
677-89-9011	Hayes	3 Main St.	Harrison	A-102
182-73-6091	Turner	123 Putnam St.	Stamford	A-305
321-12-3123	Jones	100 Main St.	Harrison	A-217
336-66-9999	Lindsay	175 Park Ave.	Pittsfield	A-222
019-28-3746	Smith	72 North St.	Rye	A-201





A Sample Relational Database

customer_id	customer_name	customer_street	customer_city
192-83-7465	Johnson	12 Alma St.	Palo Alto
677-89-9011	Hayes	3 Main St.	Harrison
182-73-6091	Turner	123 Putnam Ave.	Stamford
321-12-3123	Jones	100 Main St.	Harrison
336-66-9999	Lindsay	175 Park Ave.	Pittsfield
019-28-3746	Smith	72 North St.	Rye

(a) The *customer* table

account_number	balance
A-101	500
A-215	700
A-102	400
A-305	350
A-201	900
A-217	750
A-222	700

(b) The *account* table

customer_id	account_number
192-83-7465	A-101
192-83-7465	A-201
019-28-3746	A-215
677-89-9011	A-102
182-73-6091	A-305
321-12-3123	A-217
336-66-9999	A-222
019-28-3746	A-201

(c) The *depositor* table





SQL

- **SQL**: widely used non-procedural language
 - Example: Find the name of the customer with customer-id 192-83-7465

```
select customer.customer_name
from customer
where customer.customer_id = '192-83-7465'
```
 - Example: Find the balances of all accounts held by the customer with customer-id 192-83-7465

```
select account.balance
from depositor, account
where depositor.customer_id = '192-83-7465' and
      depositor.account_number = account.account_number
```
- Application programs generally access databases through one of
 - Language extensions to allow embedded SQL
 - Application program interface (e.g., ODBC/JDBC) which allow SQL queries to be sent to a database





Database Design

The process of designing the general structure of the database:

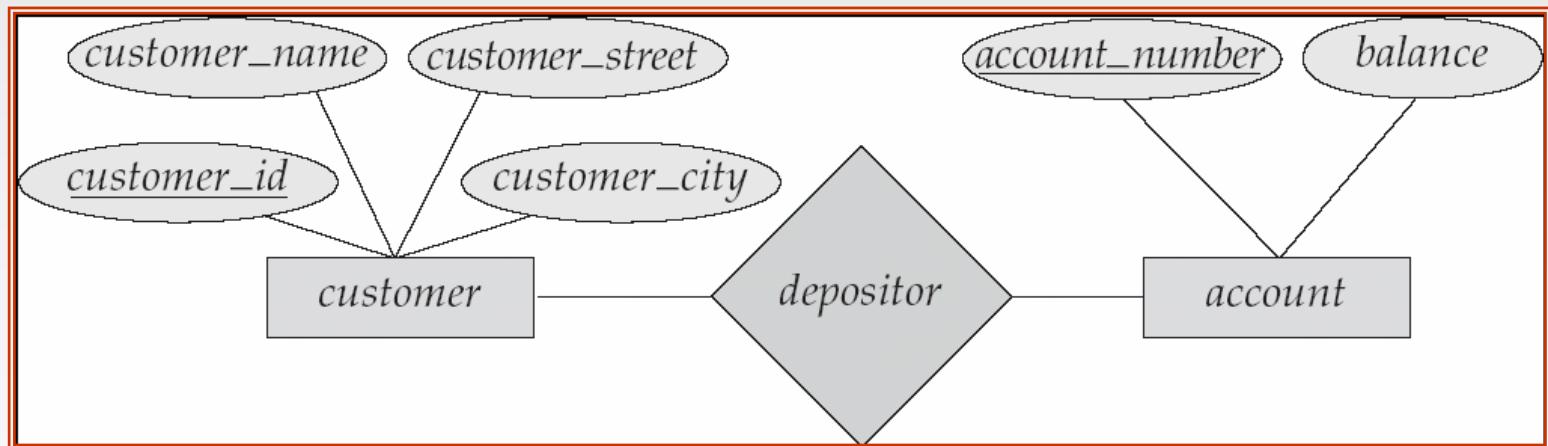
- Logical Design – Deciding on the database schema. Database design requires that we find a “good” collection of relation schemas.
 - Business decision – What attributes should we record in the database?
 - Computer Science decision – What relation schemas should we have and how should the attributes be distributed among the various relation schemas?
- Physical Design – Deciding on the physical layout of the database





The Entity-Relationship Model

- Models an enterprise as a collection of *entities* and *relationships*
 - Entity: a “thing” or “object” in the enterprise that is distinguishable from other objects
 - ▶ Described by a set of *attributes*
 - Relationship: an association among several entities
- Represented diagrammatically by an *entity-relationship diagram*:





Object-Relational Data Models

- Extend the relational data model by including object orientation and constructs to deal with added data types.
- Allow attributes of tuples to have complex types, including non-atomic values such as nested relations.
- Preserve relational foundations, in particular the declarative access to data, while extending modeling power.
- Provide upward compatibility with existing relational languages.





XML: Extensible Markup Language

- Defined by the WWW Consortium (W3C)
- Originally intended as a document markup language not a database language
- The ability to specify new tags, and to create nested tag structures made XML a great way to exchange **data**, not just documents
- XML has become the basis for all new generation data interchange formats.
- A wide variety of tools is available for parsing, browsing and querying XML documents/data





Storage Management

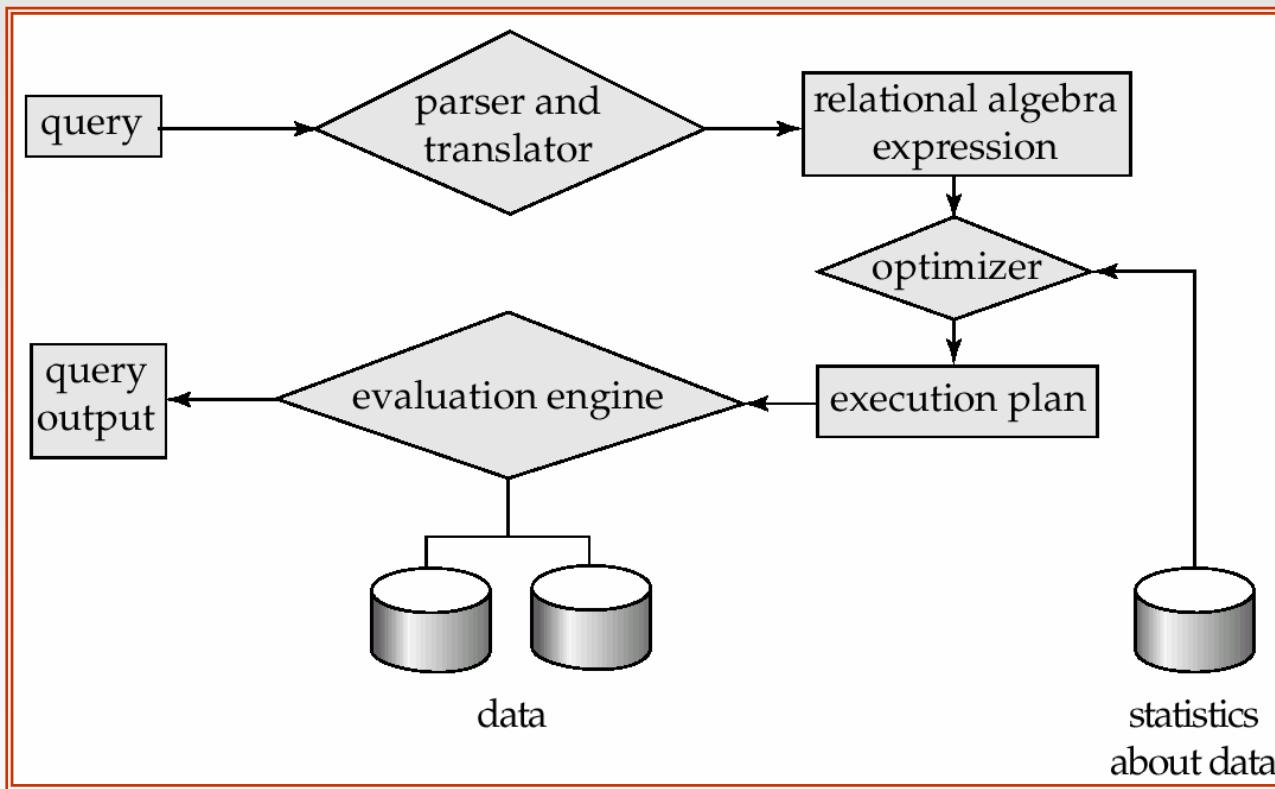
- **Storage manager** is a program module that provides the interface between the low-level data stored in the database and the application programs and queries submitted to the system.
- The storage manager is responsible to the following tasks:
 - Interaction with the file manager
 - Efficient storing, retrieving and updating of data
- Issues:
 - Storage access
 - File organization
 - Indexing and hashing





Query Processing

1. Parsing and translation
2. Optimization
3. 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





Transaction Management

- A **transaction** is a collection of operations that performs a single logical function in a database application
- **Transaction-management component** ensures that the database remains in a consistent (correct) state despite system failures (e.g., power failures and operating system crashes) and transaction failures.
- **Concurrency-control manager** controls the interaction among the concurrent transactions, to ensure the consistency of the database.





Database Architecture

The architecture of a database systems is greatly influenced by the underlying computer system on which the database is running:

- Centralized
- Client-server
- Parallel (multi-processor)
- Distributed





Database Users

Users are differentiated by the way they expect to interact with the system

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





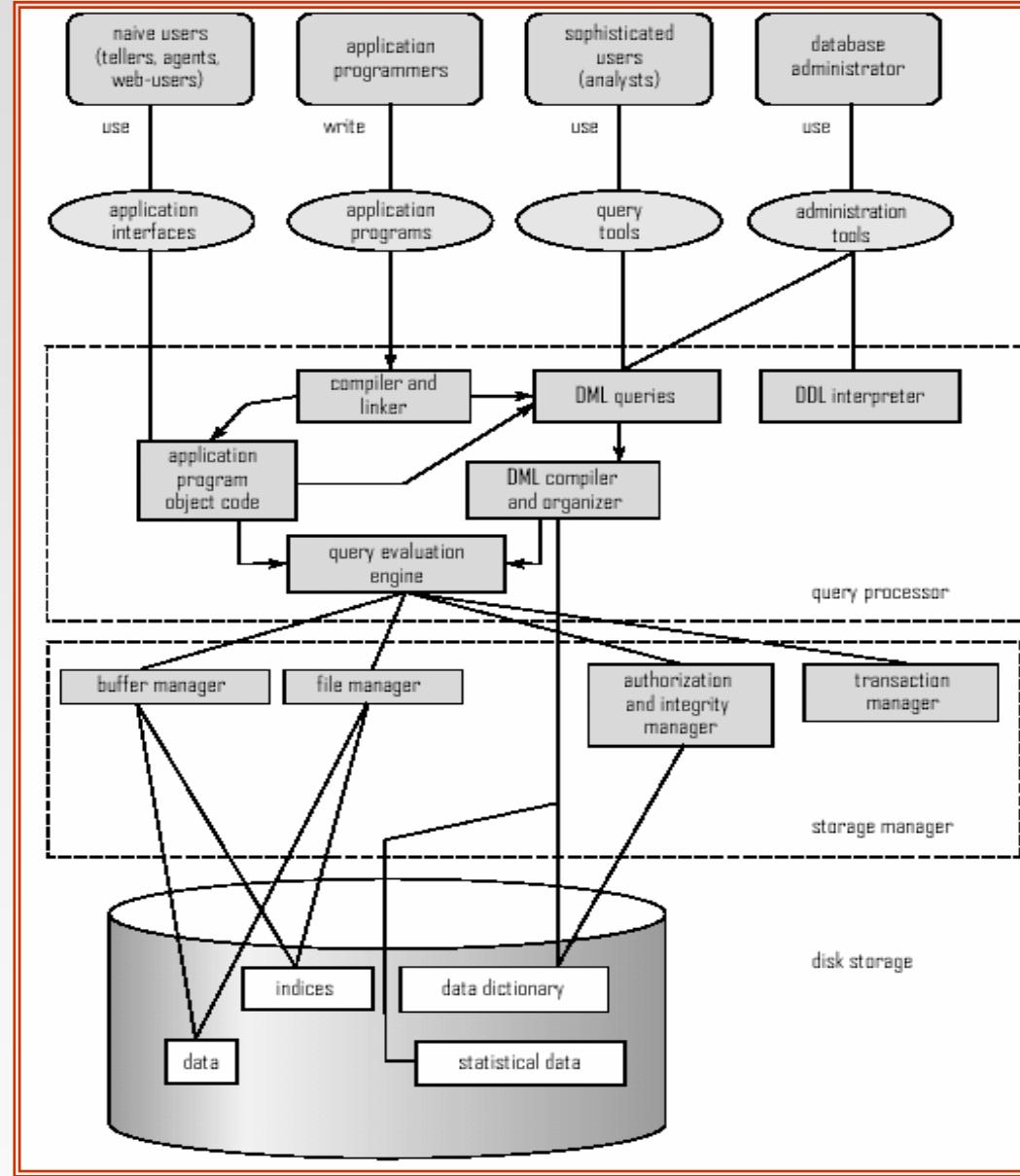
Database Administrator

- Coordinates all the activities of the database system; the database administrator has a good understanding of the enterprise's information resources and needs.
- Database administrator's duties include:
 - Schema definition
 - Storage structure and access method definition
 - Schema and physical organization modification
 - Granting user authority to access the database
 - Specifying integrity constraints
 - Acting as liaison with users
 - Monitoring performance and responding to changes in requirements





Overall System Structure





History of Database Systems

- 1950s and early 1960s:
 - Data processing using magnetic tapes for storage
 - ▶ Tapes provide only sequential access
 - Punched cards for input
- Late 1960s and 1970s:
 - Hard disks allow direct access to data
 - Network and hierarchical data models in widespread use
 - Ted Codd defines the relational data model
 - ▶ Would win the ACM Turing Award for this work
 - ▶ IBM Research begins System R prototype
 - ▶ UC Berkeley begins Ingres prototype
 - High-performance (for the era) transaction processing





History (cont.)

- 1980s:
 - Research relational prototypes evolve into commercial systems
 - ▶ SQL becomes industrial standard
 - Parallel and distributed database systems
 - Object-oriented database systems
- 1990s:
 - Large decision support and data-mining applications
 - Large multi-terabyte data warehouses
 - Emergence of Web commerce
- 2000s:
 - XML and XQuery standards
 - Automated database administration





End of Chapter 1

Database System Concepts, 5th Ed.

©Silberschatz, Korth and Sudarshan

See www.db-book.com for conditions on re-use





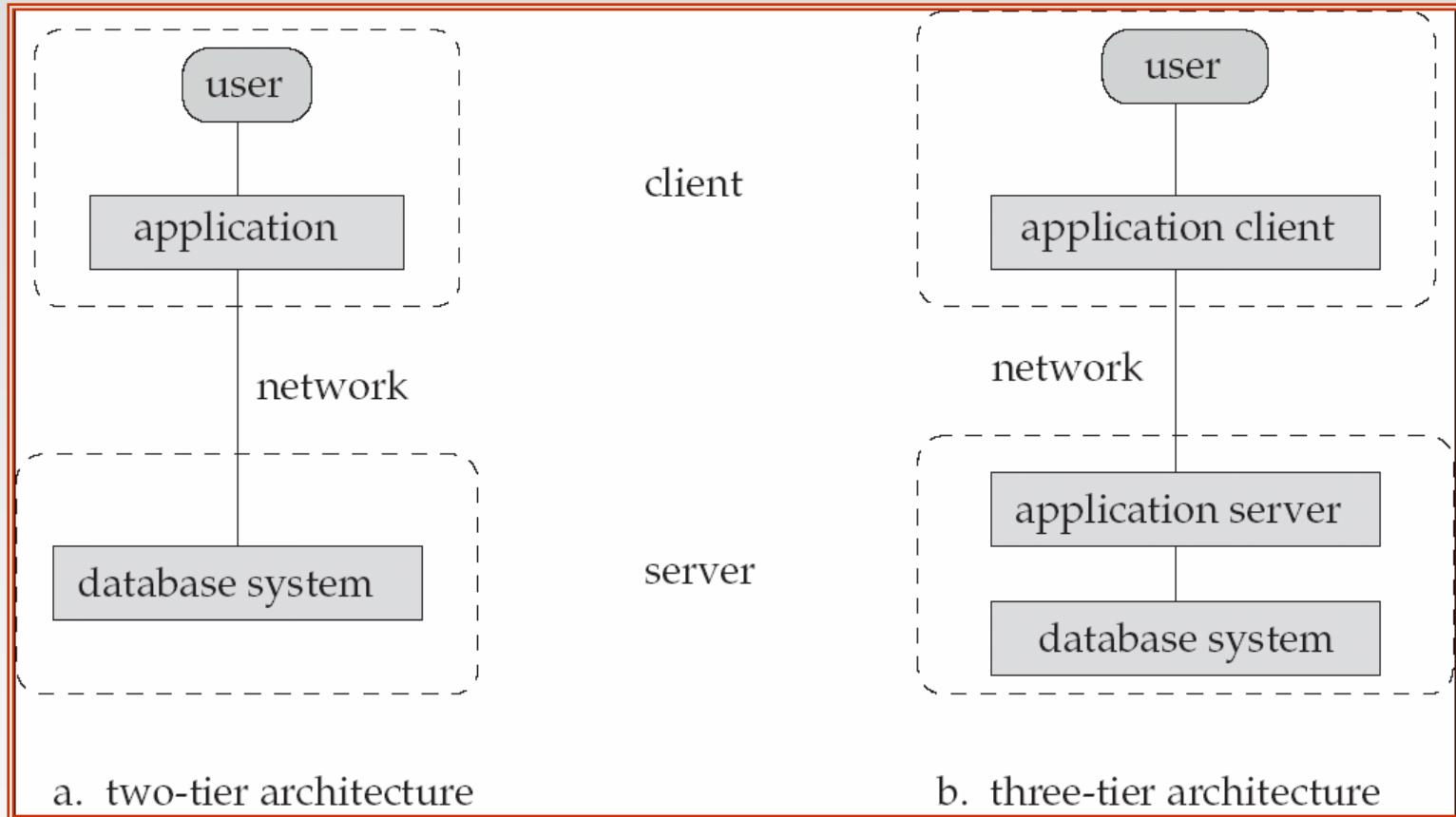
Figure 1.4

<i>customer_id</i>	<i>account_number</i>	<i>balance</i>
192-83-7465	A-101	500
192-83-7465	A-201	900
019-28-3746	A-215	700
677-89-9011	A-102	400
182-73-6091	A-305	350
321-12-3123	A-217	750
336-66-9999	A-222	700
019-28-3746	A-201	900





Figure 1.7





Chapter 6: Entity-Relationship Model

Database System Concepts, 5th Ed.

©Silberschatz, Korth and Sudarshan

See www.db-book.com for conditions on re-use





Chapter 6: Entity-Relationship Model

- Design Process
- Modeling
- Constraints
- E-R Diagram
- Design Issues
- Weak Entity Sets
- Extended E-R Features
- Design of the Bank Database
- Reduction to Relation Schemas
- Database Design
- UML





Modeling

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





Entity Sets *customer* and *loan*

customer_id	customer_name	customer_street	customer_city	loan_number	amount
321-12-3123	Jones	Main	Harrison	L-17	1000
019-28-3746	Smith	North	Rye	L-23	2000
677-89-9011	Hayes	Main	Harrison	L-15	1500
555-55-5555	Jackson	Dupont	Woodside	L-14	1500
244-66-8800	Curry	North	Rye	L-19	500
963-96-3963	Williams	Nassau	Princeton	L-11	900
335-57-7991	Adams	Spring	Pittsfield	L-16	1300

customer

loan





Relationship Sets

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

Example:

Hayes depositor A-102
customer entity relationship set account entity

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

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

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

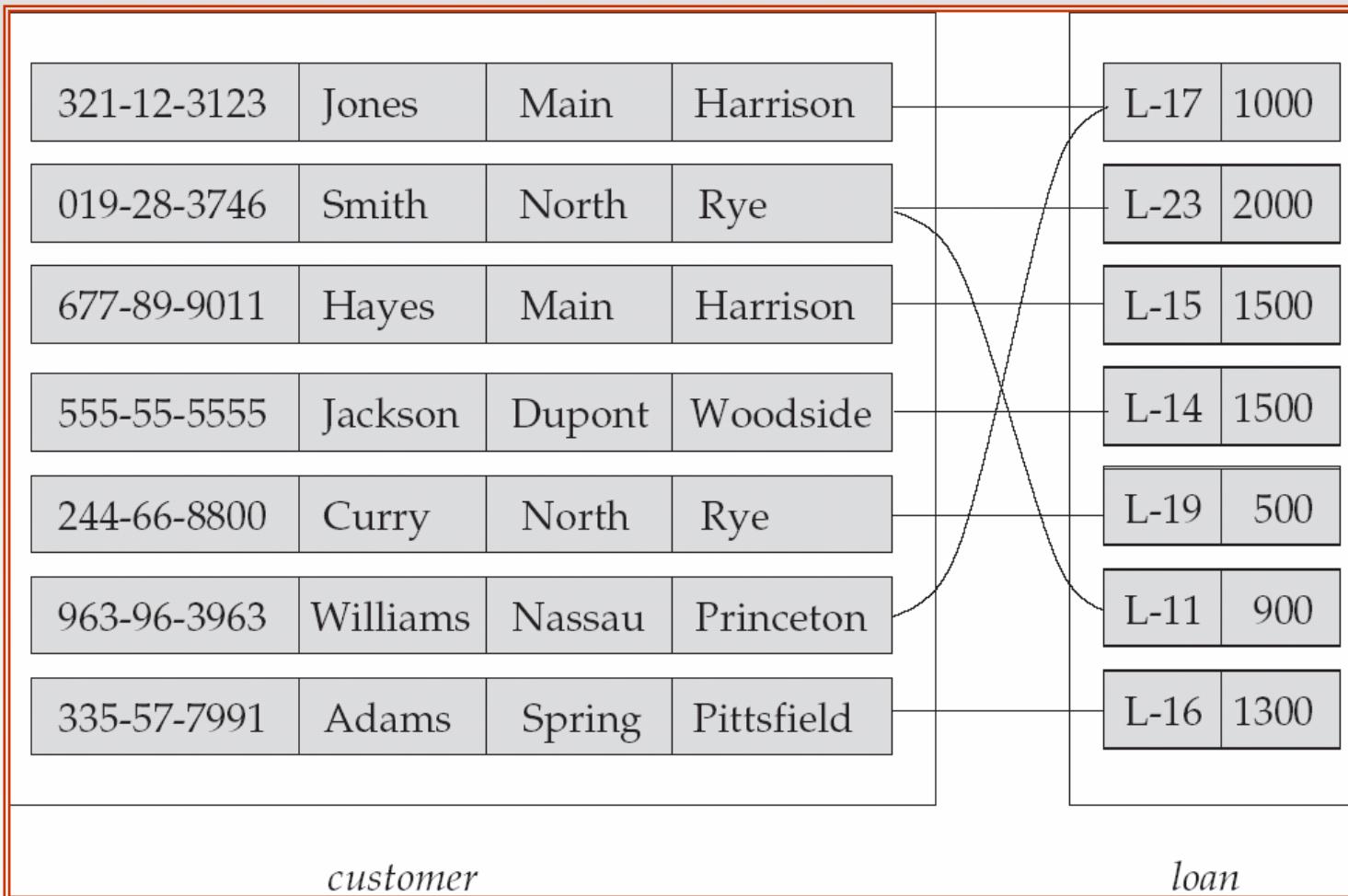
- Example:

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





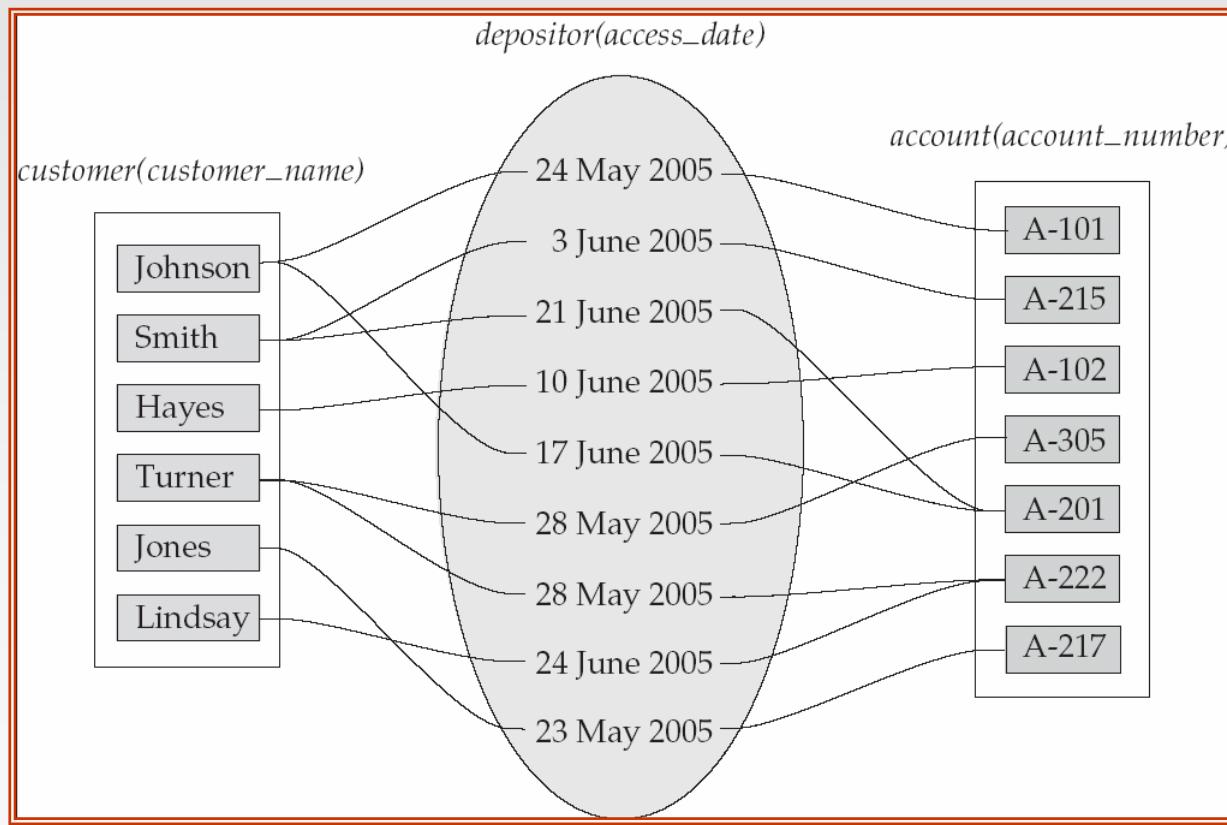
Relationship Set *borrower*





Relationship Sets (Cont.)

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





Degree of a Relationship Set

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





Attributes

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

Example:

$$\begin{aligned}customer &= (customer_id, customer_name, \\&\quad customer_street, customer_city) \\loan &= (loan_number, amount)\end{aligned}$$

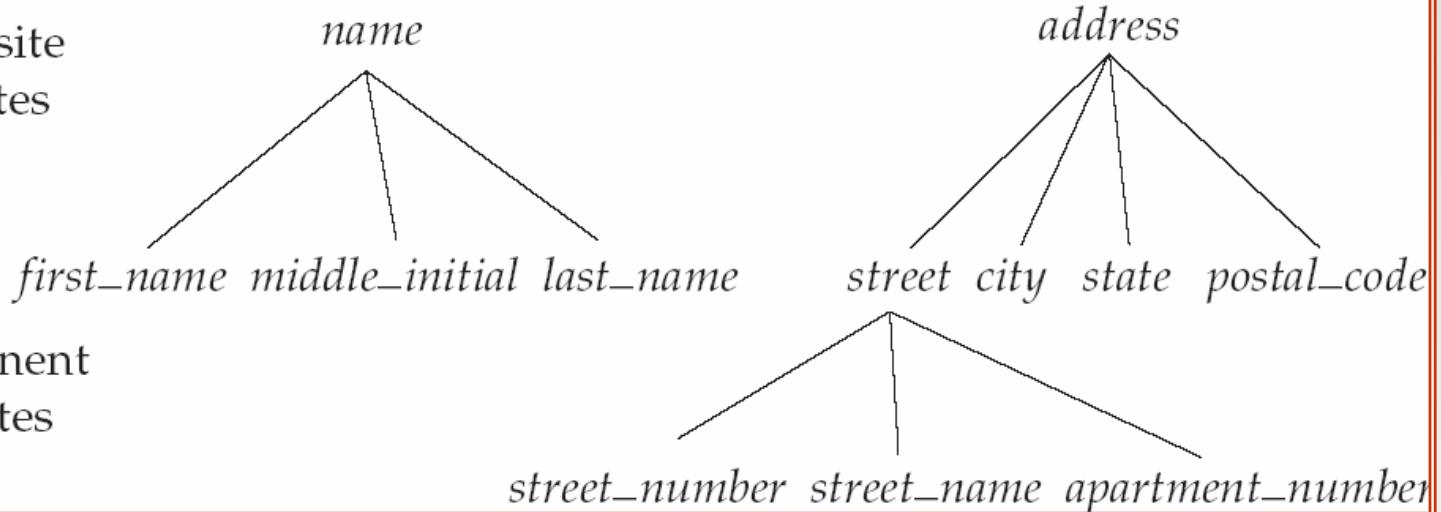
- Domain** – the set of permitted values for each attribute
- Attribute types:
 - Simple and composite attributes.
 - Single-valued and multi-valued attributes
 - Example: multivalued attribute: *phone_numbers*
 - Derived attributes
 - Can be computed from other attributes
 - Example: age, given date_of_birth





Composite Attributes

Composite
Attributes



Component
Attributes





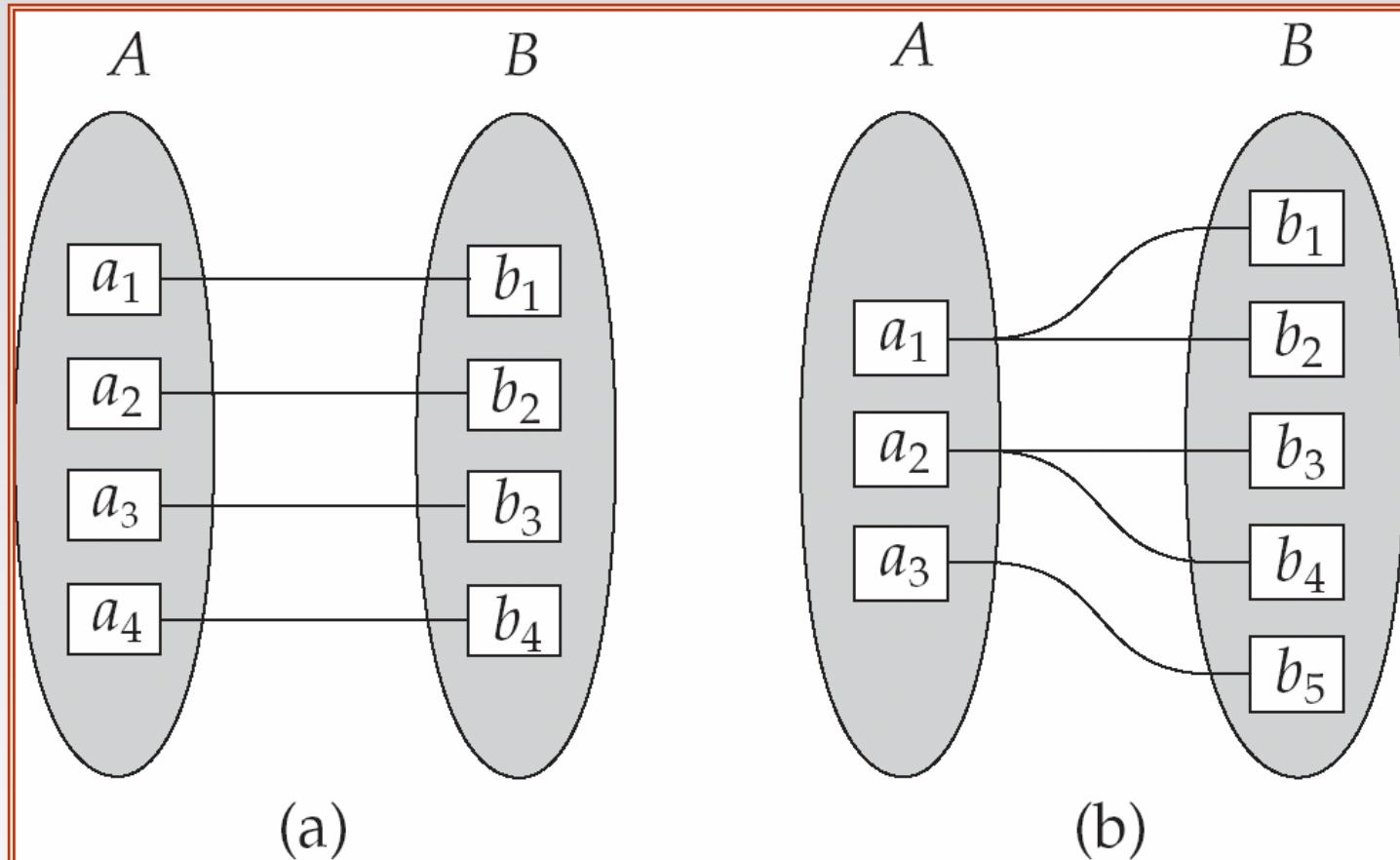
Mapping Cardinality Constraints

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





Mapping Cardinalities



One to one

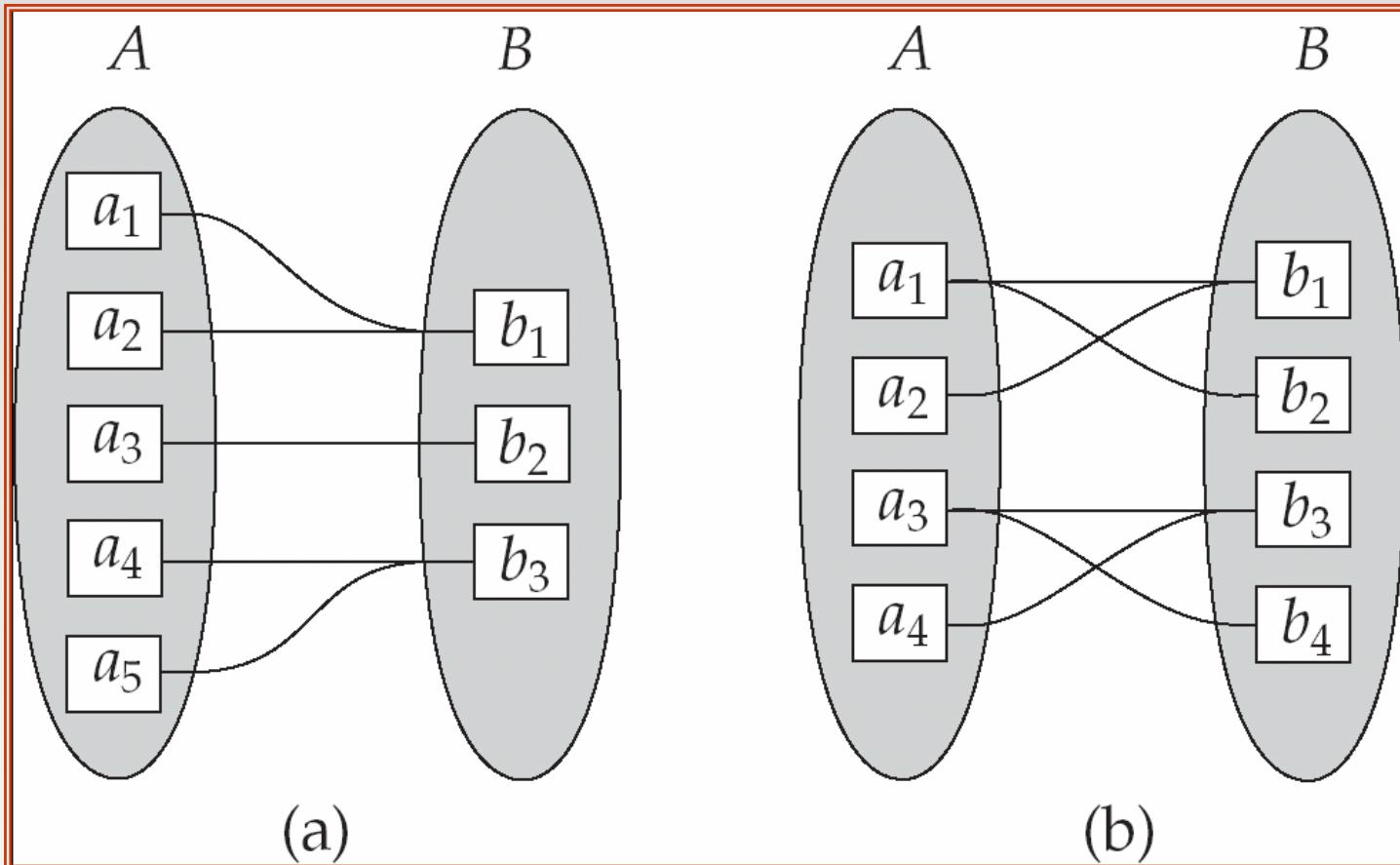
One to many

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





Mapping Cardinalities



Many to one

Many to many

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





Keys

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





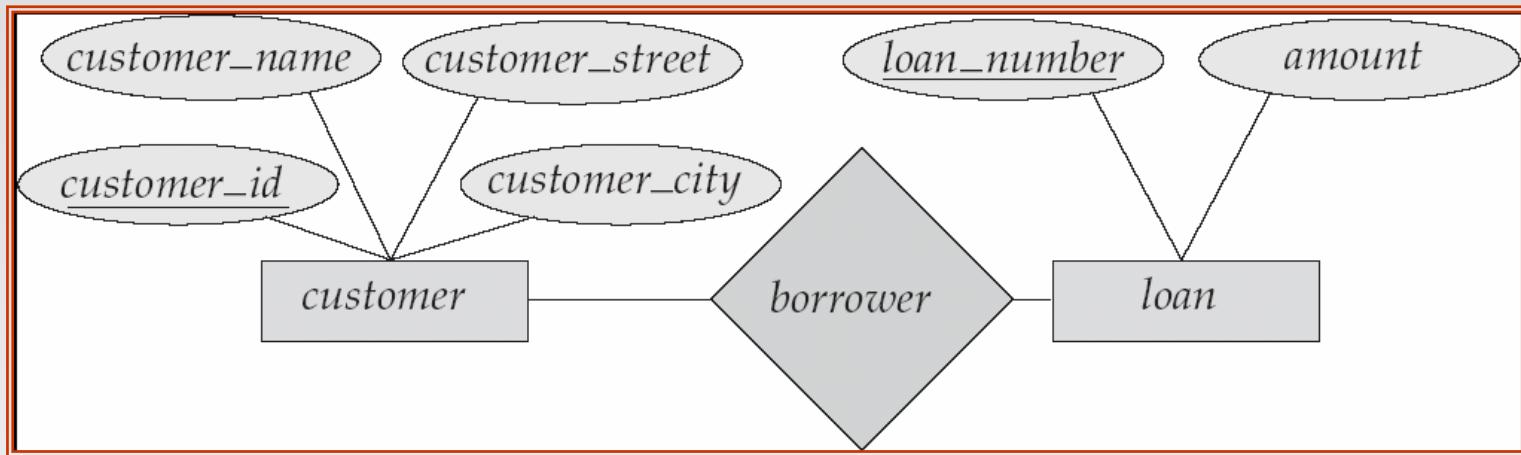
Keys for Relationship Sets

- The combination of primary keys of the participating entity sets forms a super key of a relationship set.
 - $(customer_id, account_number)$ is the super key of *depositor*
 - *NOTE: this means a pair of entity sets can have at most one relationship in a particular relationship set.*
 - ▶ Example: if we wish to track all access_dates to each account by each customer, we cannot assume a relationship for each access. We can use a multivalued attribute though
- Must consider the mapping cardinality of the relationship set when deciding what are the candidate keys
- Need to consider semantics of relationship set in selecting the *primary key* in case of more than one candidate key





E-R Diagrams

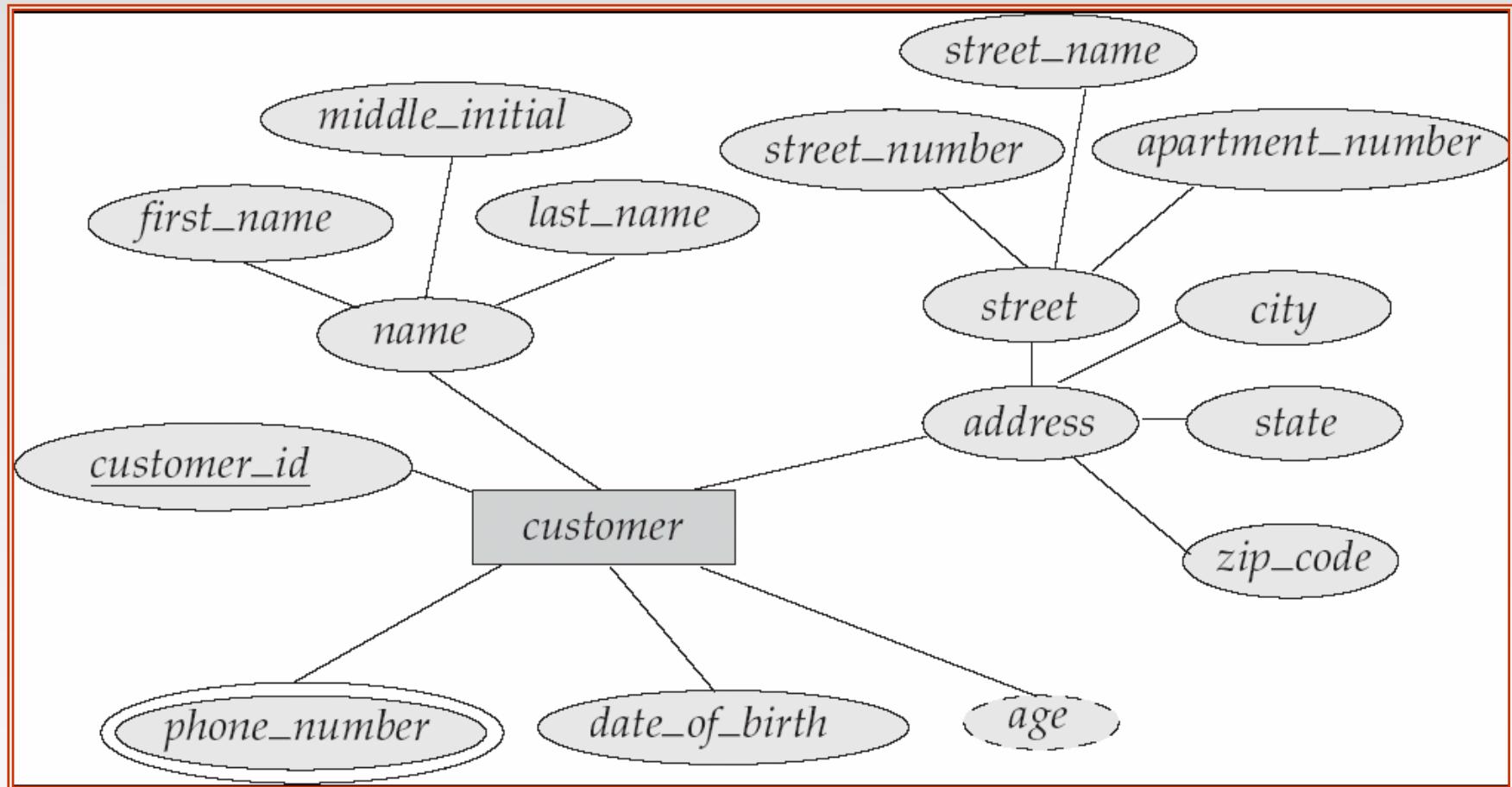


- Rectangles represent entity sets.
- Diamonds represent relationship sets.
- Lines link attributes to entity sets and entity sets to relationship sets.
- Ellipses represent attributes
 - Double ellipses represent multivalued attributes.
 - Dashed ellipses denote derived attributes.
- Underline indicates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



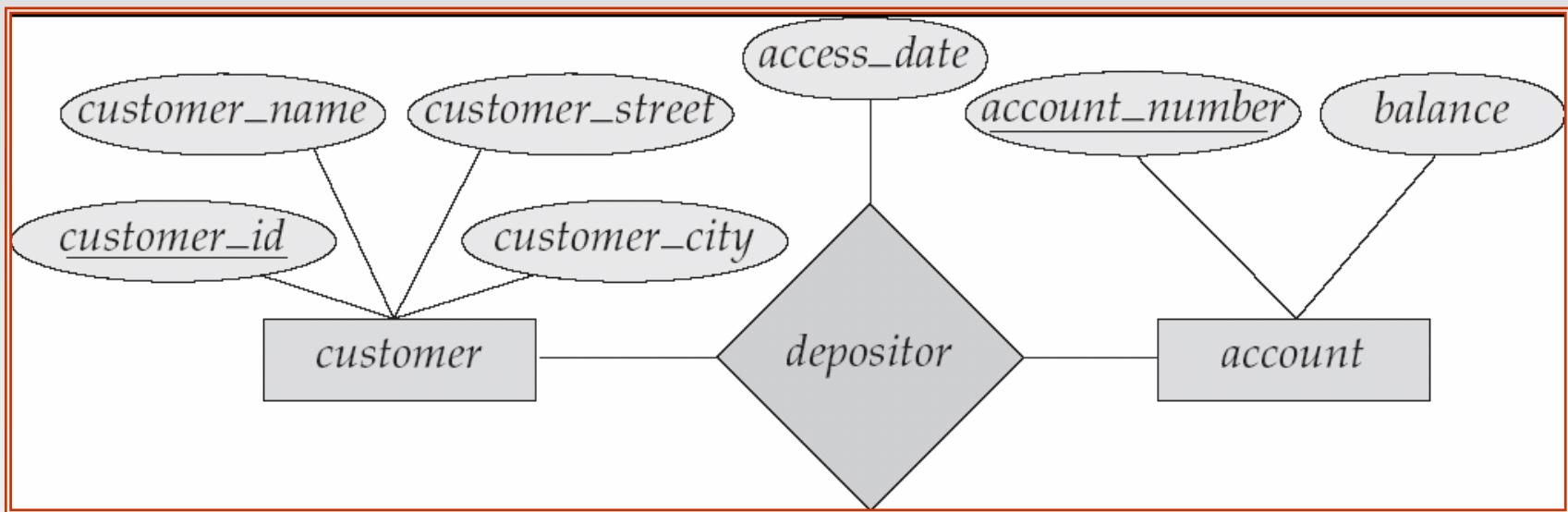


E-R Diagram With Composite, Multivalued, and Derived Attributes





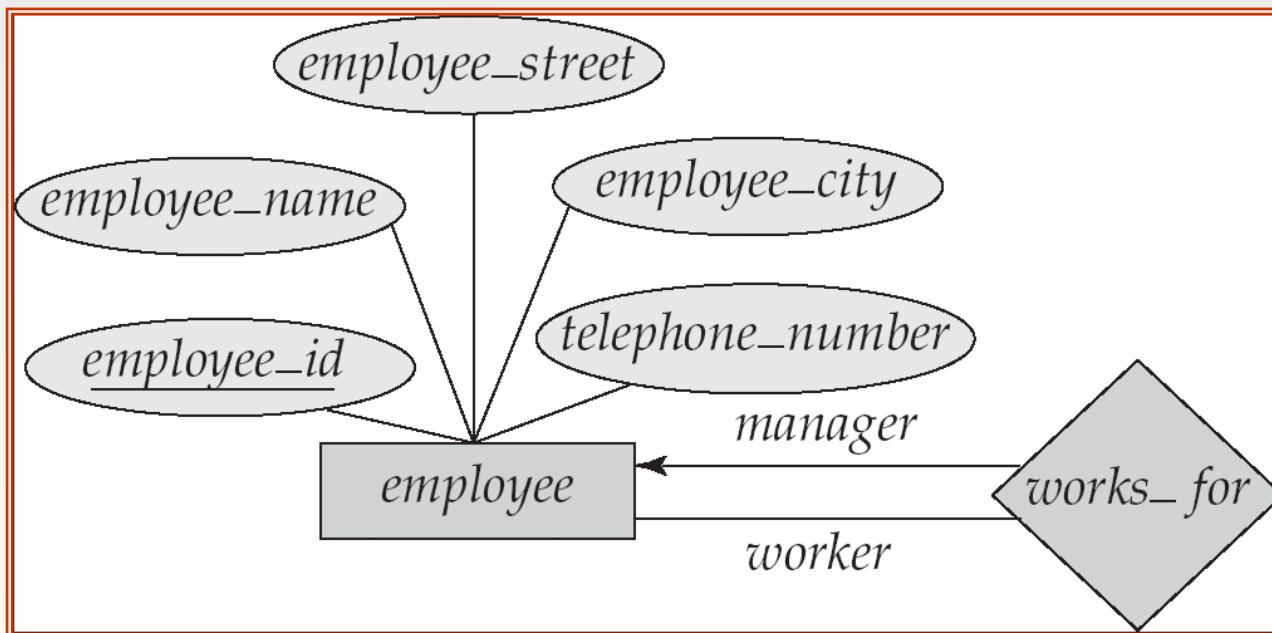
Relationship Sets with Attributes





Roles

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





Cardinality Constraints

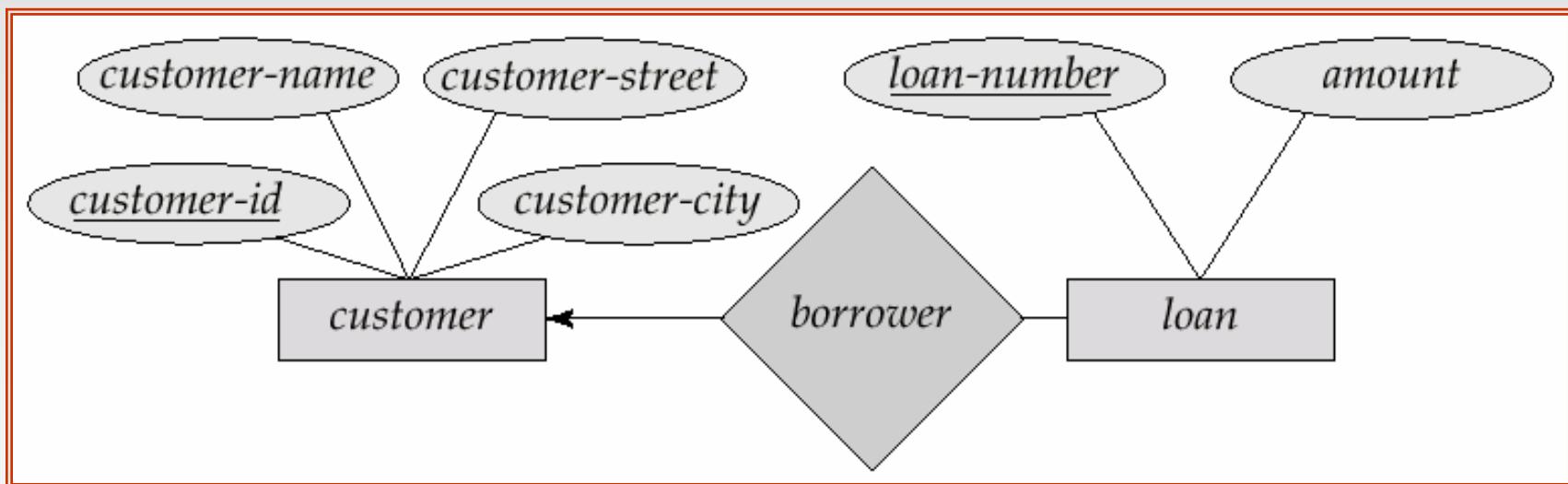
- We express cardinality constraints by drawing either a directed line (\rightarrow), signifying “one,” or an undirected line ($-$), signifying “many,” between the relationship set and the entity set.
- One-to-one relationship:
 - A customer is associated with at most one loan via the relationship *borrower*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





One-To-Many Relationship

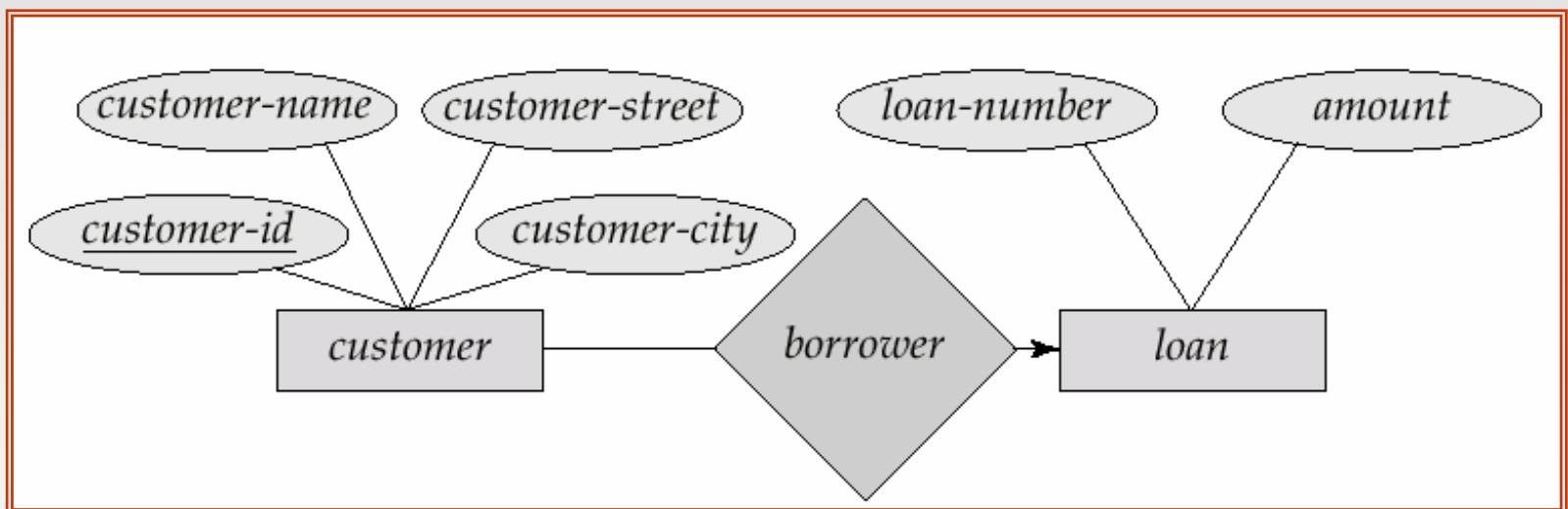
- In the one-to-many relationship a loan is associated with at most one customer via *borrower*, a customer is associated with several (including 0) loans via *borrower*





Many-To-One Relationships

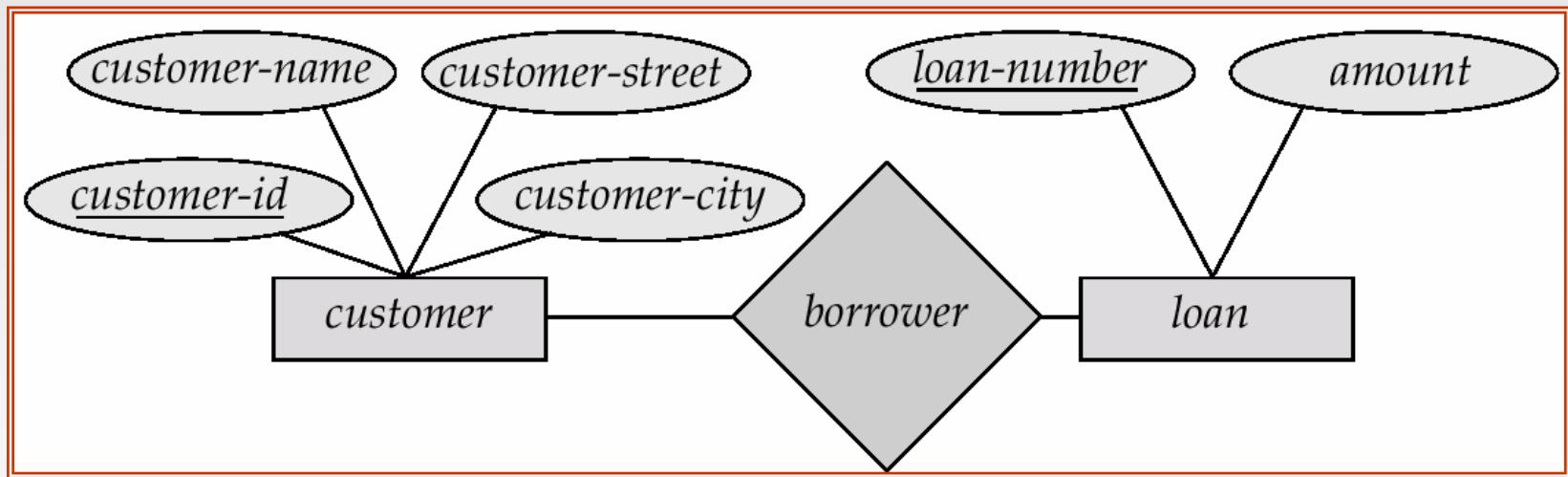
- In a many-to-one relationship a loan is associated with several (including 0) customers via *borrower*, a customer is associated with at most one loan via *borrower*





Many-To-Many Relationship

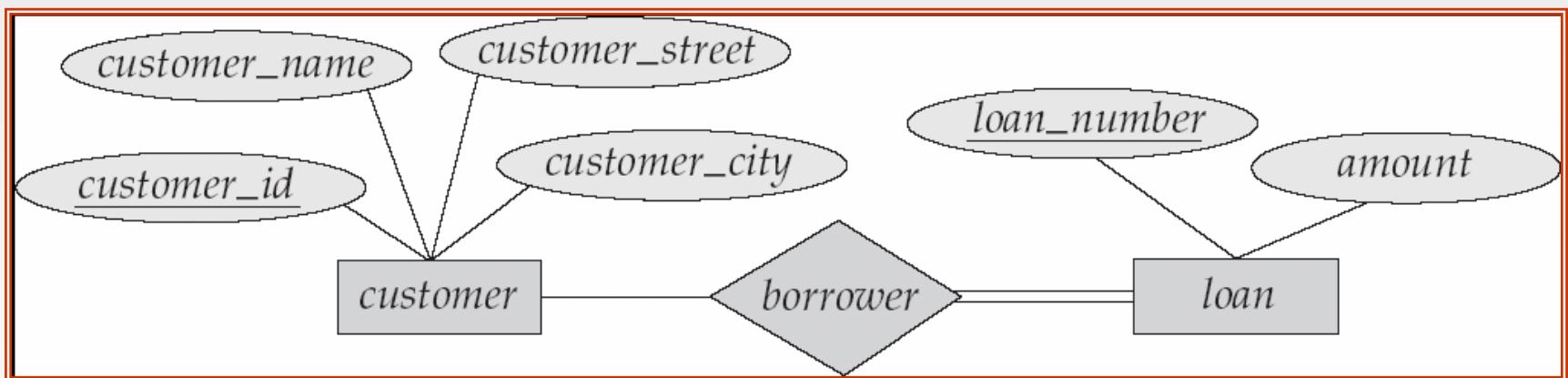
- A customer is associated with several (possibly 0) loans via borrower
- A loan is associated with several (possibly 0) customers via borrower





Participation of an Entity Set in a Relationship Set

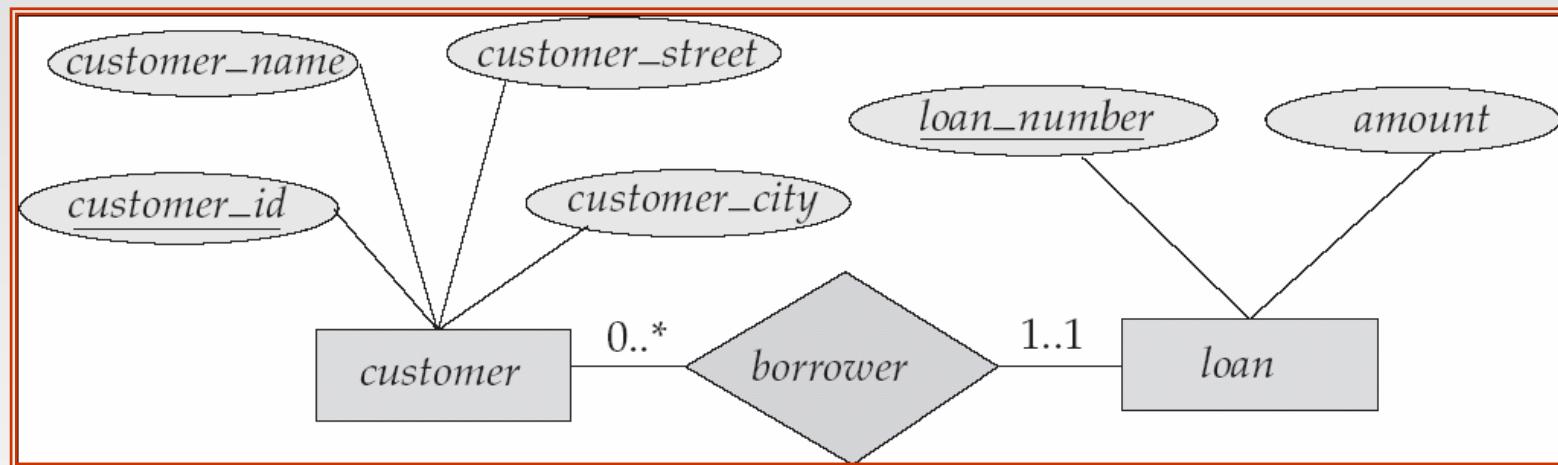
- Total participation (indicated by double line): every entity in the entity set participates in at least one relationship in the relationship set
 - E.g. participation of loan in borrower is total
 - ▶ every loan must have a customer associated to it via borrower
- Partial participation: some entities may not participate in any relationship in the relationship set
 - Example: participation of customer in borrower is partial





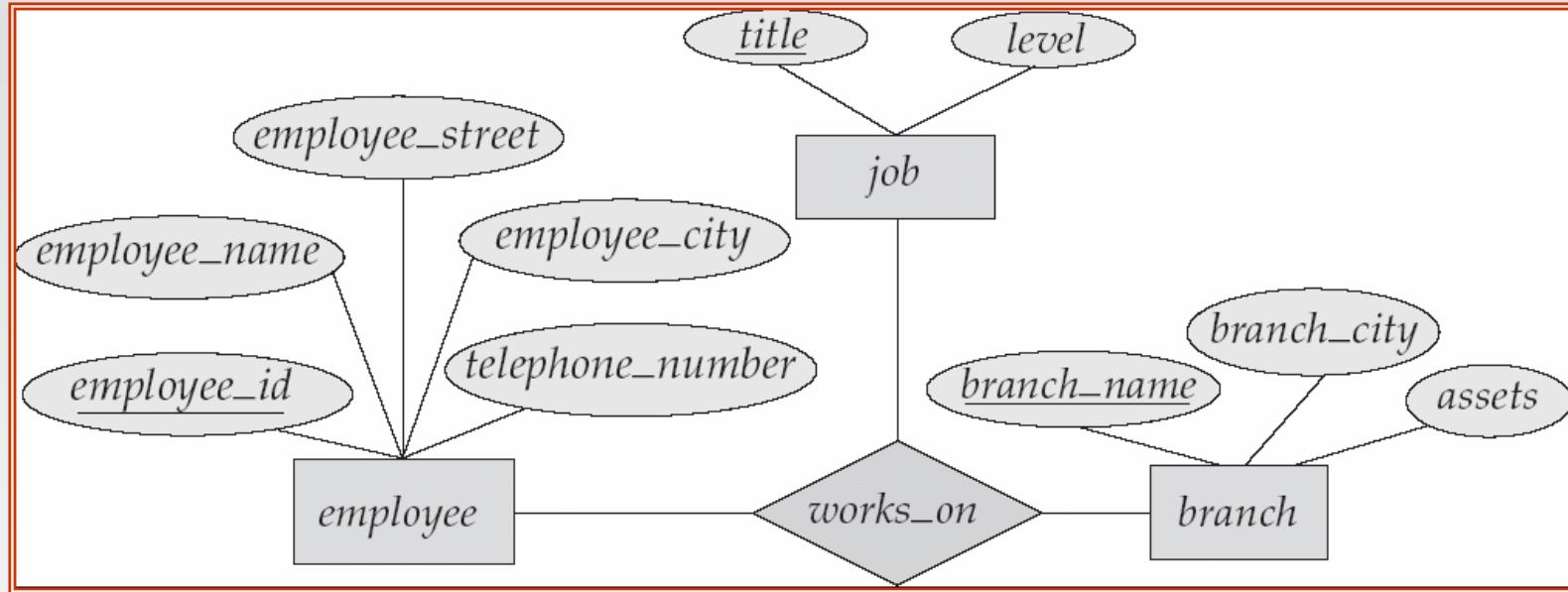
Alternative Notation for Cardinality Limits

- Cardinality limits can also express participation constraints





E-R Diagram with a Ternary Relationship





Cardinality Constraints on Ternary Relationship

- We allow at most one arrow out of a ternary (or greater degree) relationship to indicate a cardinality constraint
- E.g. an arrow from *works_on* to *job* indicates each employee works on at most one job at any branch.
- If there is more than one arrow, there are two ways of defining the meaning.
 - E.g a ternary relationship R between A , B and C with arrows to B and C could mean
 1. each A entity is associated with a unique entity from B and C or
 2. each pair of entities from (A, B) is associated with a unique C entity, and each pair (A, C) is associated with a unique B
 - Each alternative has been used in different formalisms
 - To avoid confusion we outlaw more than one arrow





Design Issues

■ Use of entity sets vs. attributes

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

■ Use of entity sets vs. relationship sets

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

■ Binary versus n-ary relationship sets

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

■ Placement of relationship attributes





Binary Vs. Non-Binary Relationships

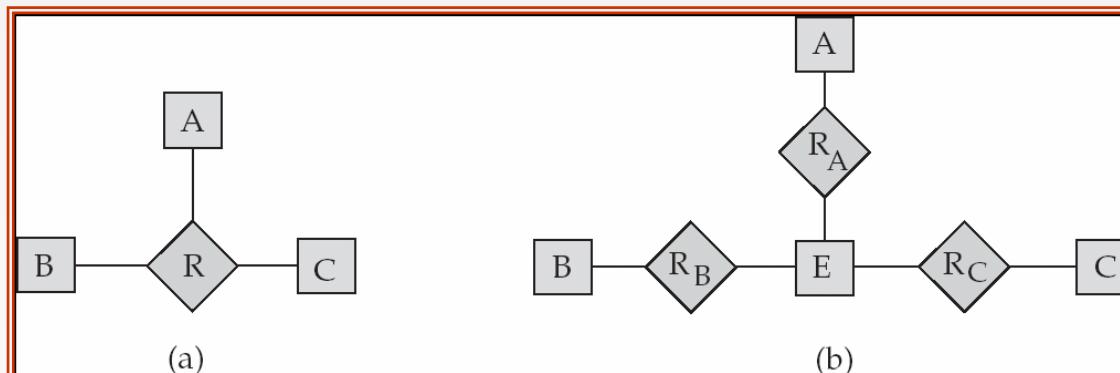
- Some relationships that appear to be non-binary may be better represented using binary relationships
 - E.g. A ternary relationship *parents*, relating a child to his/her father and mother, is best replaced by two binary relationships, *father* and *mother*
 - ▶ Using two binary relationships allows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





Converting Non-Binary Relationships to Binary Form

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





Converting Non-Binary Relationships (Cont.)

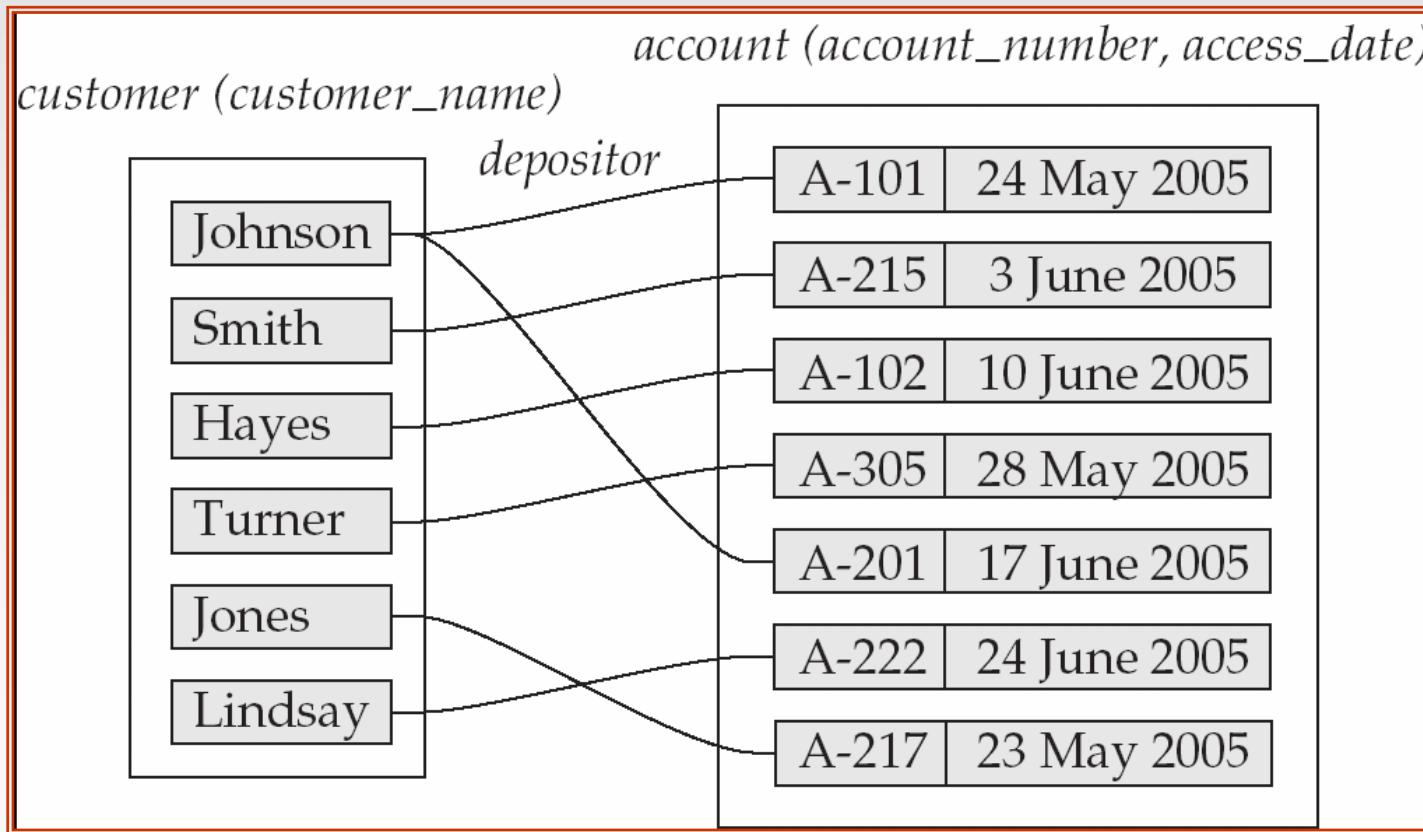
- Also need to translate constraints
 - Translating all constraints may not be possible
 - There may be instances in the translated schema that cannot correspond to any instance of R
 - ▶ Exercise: *add constraints to the relationships R_A , R_B and R_C to ensure that a newly created entity corresponds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





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
 - That is, the relationship from account to customer is many to one, or equivalently, customer to account is one to many





**How about doing an ER design
interactively on the board?
Suggest an application to be modeled.**

Database System Concepts, 5th Ed.

©Silberschatz, Korth and Sudarshan

See www.db-book.com for conditions on re-use





Weak Entity Sets

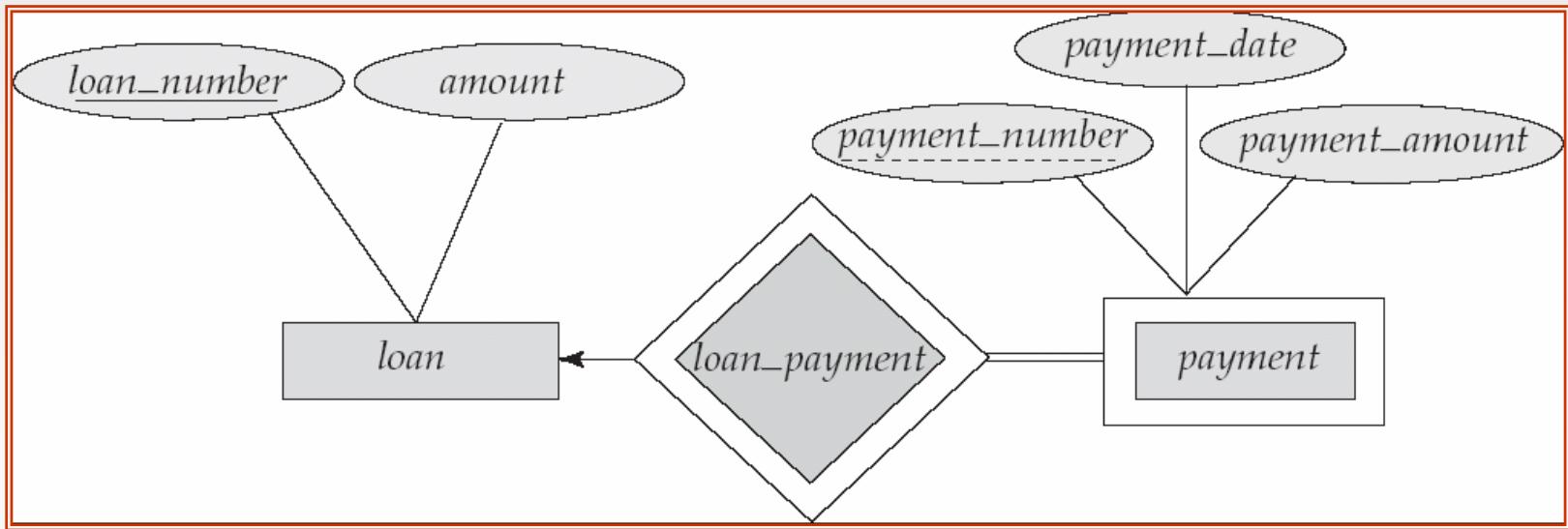
- An entity set that does not have a primary key is referred to as a **weak entity set**.
- The existence of a weak entity set depends on the existence of a **identifying entity set**
 - it must relate to the identifying entity set via a total, one-to-many relationship set from the identifying to the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





Weak Entity Sets (Cont.)

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





Weak Entity Sets (Cont.)

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





More Weak Entity Set Examples

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

Then the relationship with *course* would be implicit in the *course_number* attribute





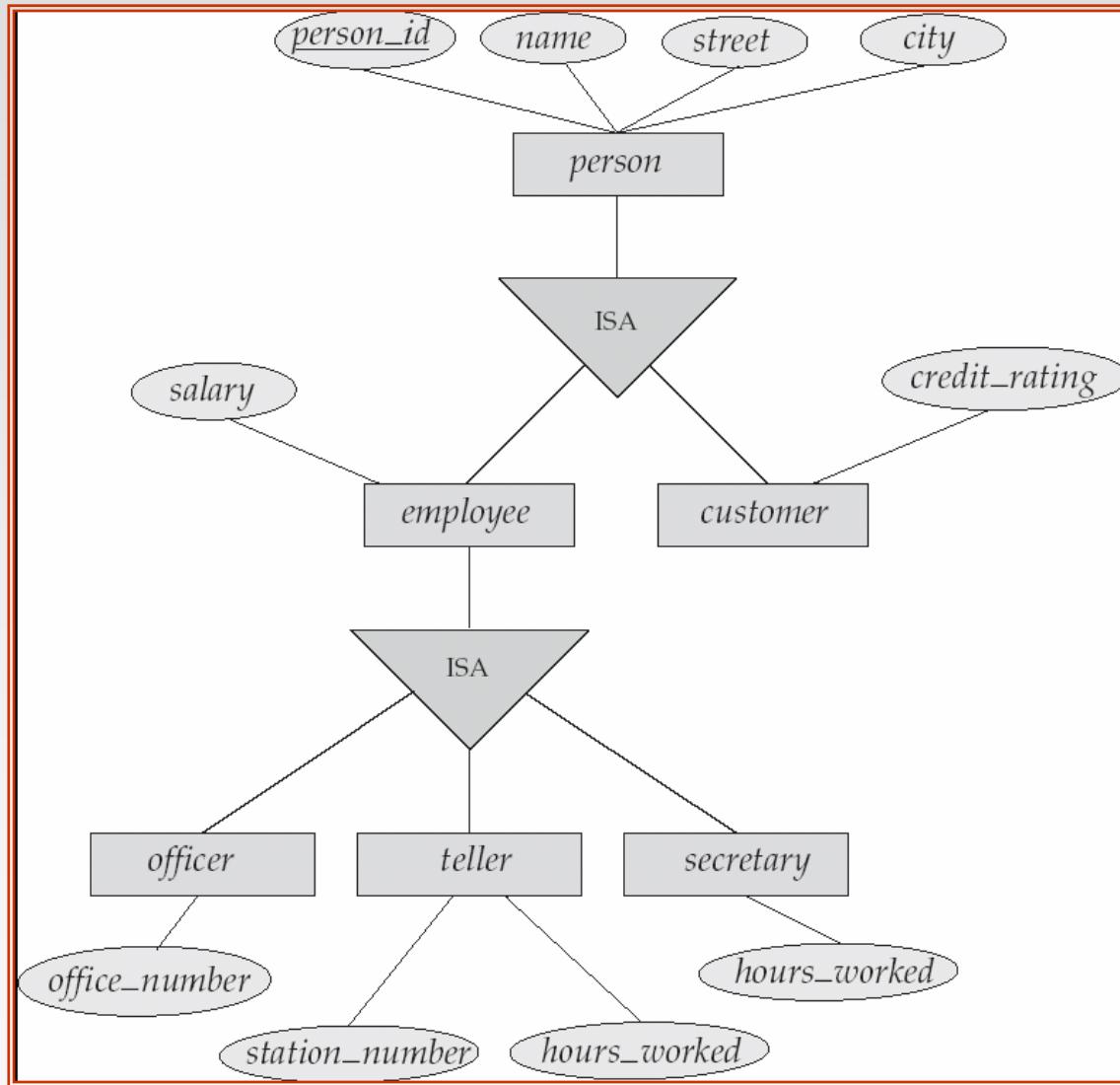
Extended E-R Features: Specialization

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





Specialization Example





Extended ER Features: Generalization

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





Specialization and Generalization (Cont.)

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





Design Constraints on a Specialization/Generalization

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





Design Constraints on a Specialization/Generalization (Cont.)

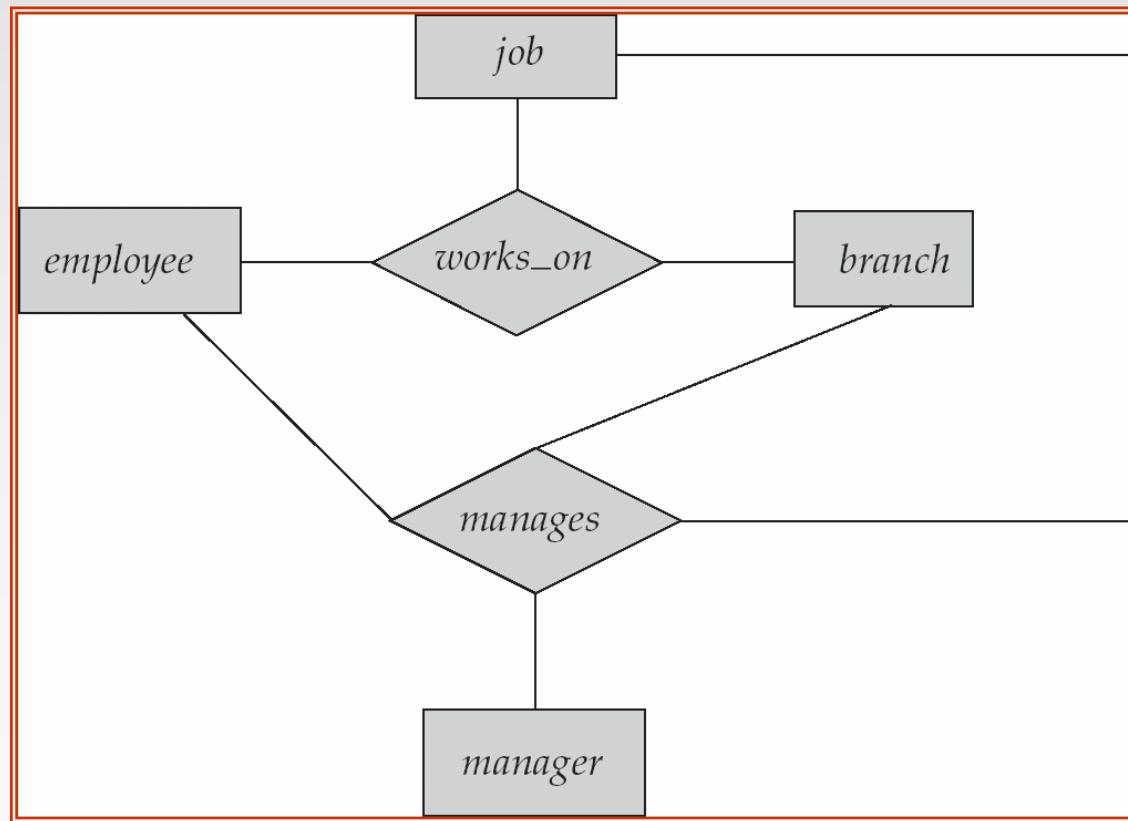
- **Completeness constraint** -- specifies whether or not an entity in the higher-level entity set must belong to at least one of the lower-level entity sets within a generalization.
 - **total** : an entity must belong to one of the lower-level entity sets
 - **partial**: an entity need not belong to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





Aggregation

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





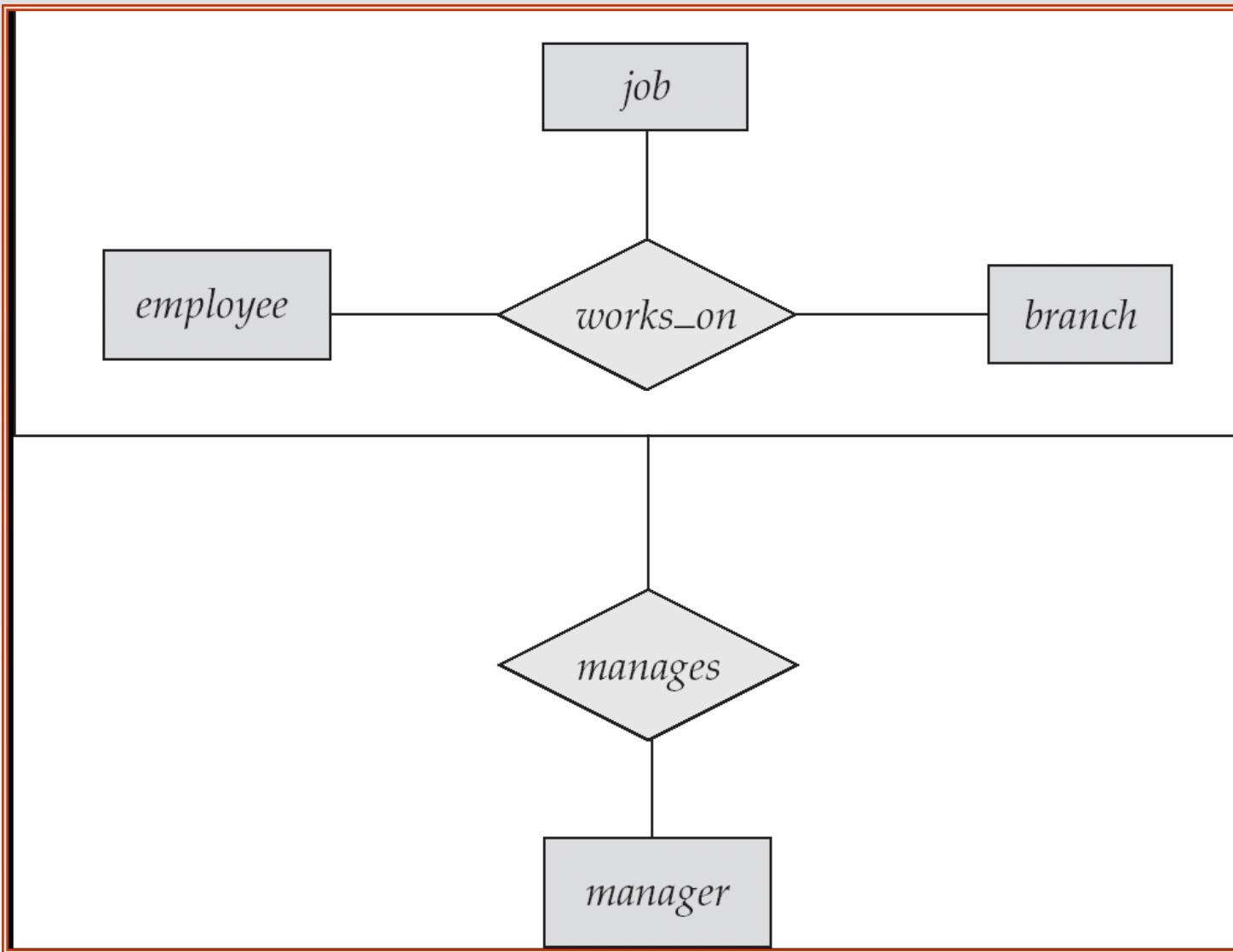
Aggregation (Cont.)

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





E-R Diagram With Aggregation





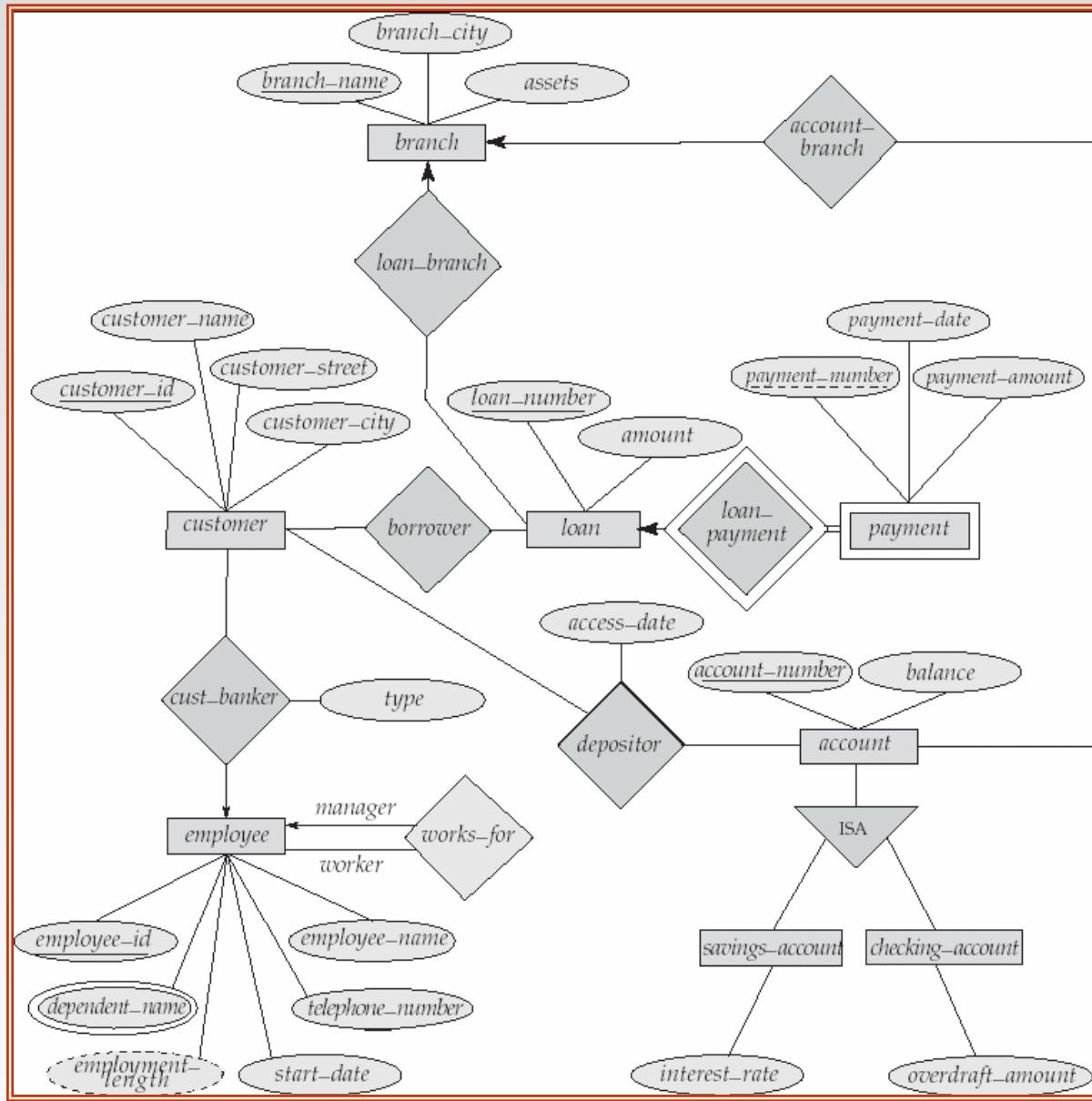
E-R Design Decisions

- The use of an attribute or entity set to represent an object.
- Whether a real-world concept is best expressed by an entity set or a relationship set.
- The use of a ternary relationship versus a pair of binary relationships.
- The use of a strong or weak entity set.
- The use of specialization/generalization – contributes to modularity in the design.
- The use of aggregation – can treat the aggregate entity set as a single unit without concern for the details of its internal structure.





E-R Diagram for a Banking Enterprise





How about doing another ER design interactively on the board?

Database System Concepts, 5th Ed.

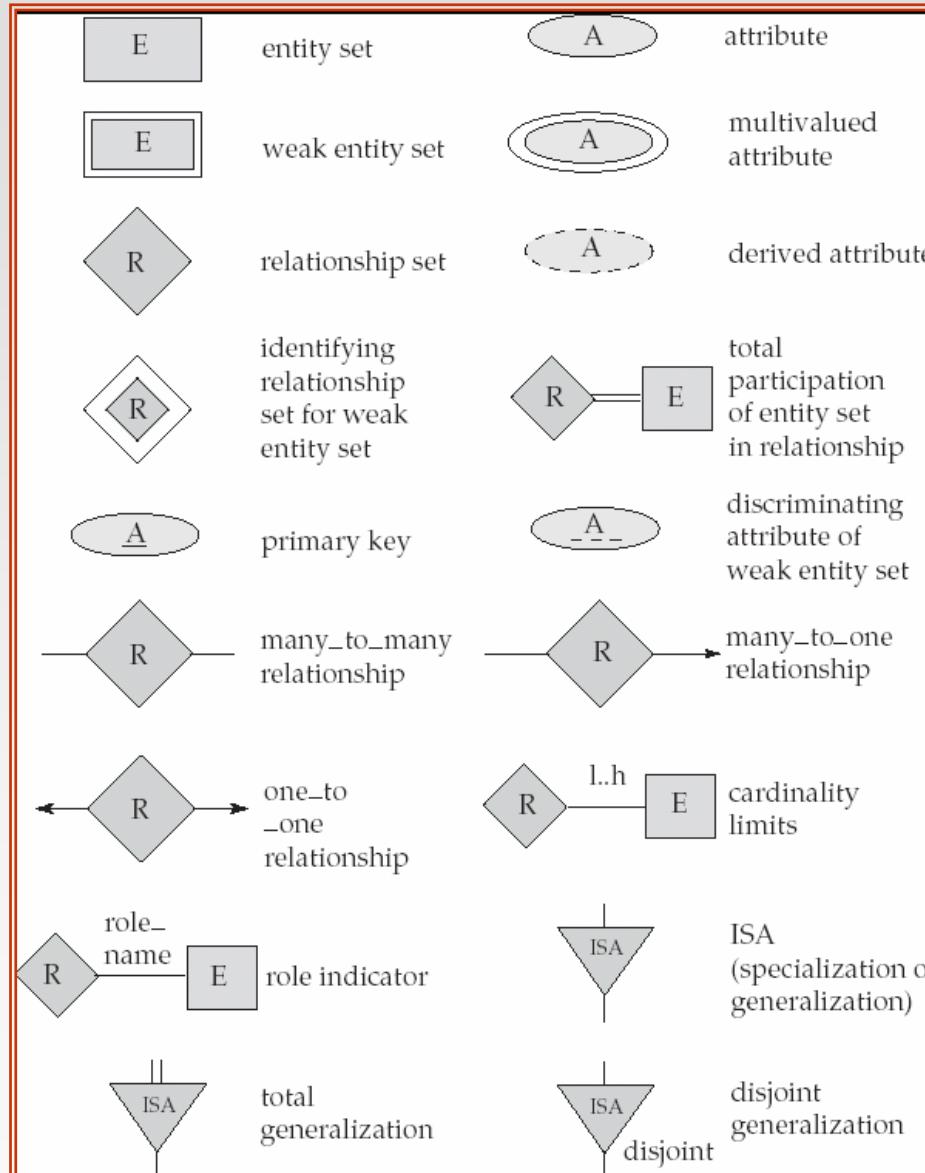
©Silberschatz, Korth and Sudarshan

See www.db-book.com for conditions on re-use



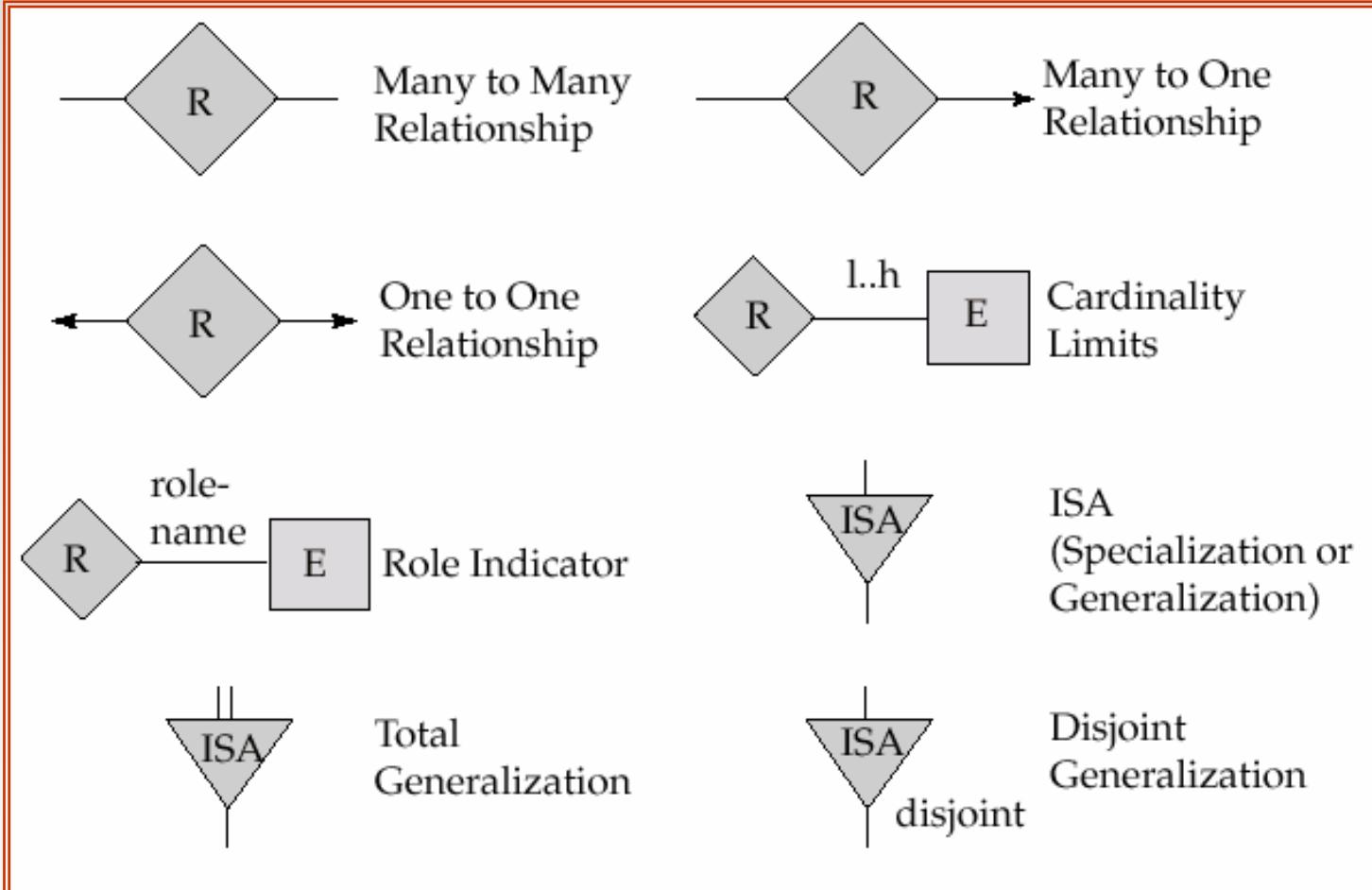


Summary of Symbols Used in E-R Notation





Summary of Symbols (Cont.)





Reduction to Relation Schemas

- Primary keys allow entity sets and relationship sets to be expressed uniformly as *relation schemas* that represent the contents of the database.
- A database which conforms to an E-R diagram can be represented by a collection of schemas.
- For each entity set and relationship set there is a unique schema that is assigned the name of the corresponding entity set or relationship set.
- Each schema has a number of columns (generally corresponding to attributes), which have unique names.





Representing Entity Sets as Schemas

- A strong entity set reduces to a schema with the same attributes.
- A weak entity set becomes a table that includes a column for the primary key of the identifying strong entity set

payment =

(loan_number, payment_number, payment_date, payment_amount)





Representing Relationship Sets as Schemas

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

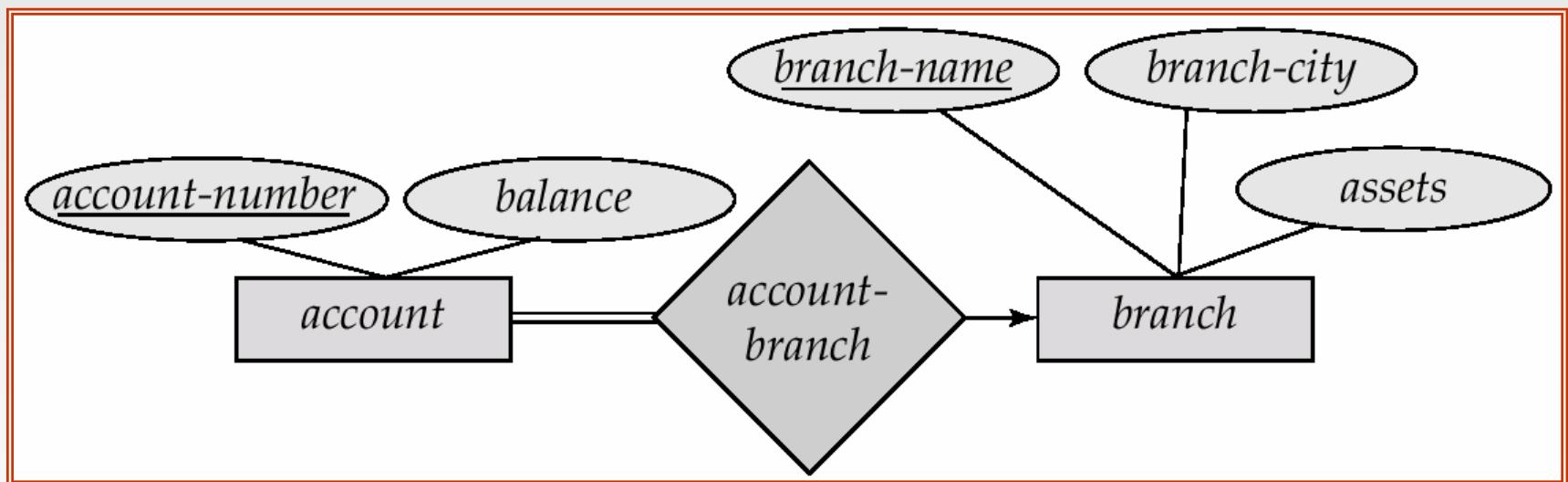
borrower = (customer_id, loan_number)





Redundancy of Schemas

- Many-to-one and one-to-many relationship sets that are total on the many-side can be represented by adding an extra attribute to the “many” side, containing the primary key of the “one” side
- Example: Instead of creating a schema for relationship set *account_branch*, add an attribute *branch_name* to the schema arising from entity set *account*





Redundancy of Schemas (Cont.)

- For one-to-one relationship sets, either side can be chosen to act as the “many” side
 - That is, extra attribute can be added to either of the tables corresponding to the two entity sets
- If participation is *partial* on the “many” side, replacing a schema by an extra attribute in the schema corresponding to the “many” side could result in null values
- The schema corresponding to a relationship set linking a weak entity set to its identifying strong entity set is redundant.
 - Example: The *payment* schema already contains the attributes that would appear in the *loan_payment* schema (i.e., *loan_number* and *payment_number*).





Composite and Multivalued Attributes

- Composite attributes are flattened out by creating a separate attribute for each component attribute
 - Example: given entity set *customer* with composite attribute *name* with component attributes *first_name* and *last_name* the schema corresponding to the entity set has two attributes
name.first_name and *name.last_name*
- A multivalued attribute *M* of an entity *E* is represented by a separate schema *EM*
 - Schema *EM* has attributes corresponding to the primary key of *E* and an attribute corresponding to multivalued attribute *M*
 - Example: Multivalued attribute *dependent_names* of *employee* is represented by a schema:
employee_dependent_names = (*employee_id*, *dname*)
 - Each value of the multivalued attribute maps to a separate tuple of the relation on schema *EM*
 - ▶ For example, an employee entity with primary key 123-45-6789 and dependents Jack and Jane maps to two tuples:
(123-45-6789 , Jack) and (123-45-6789 , Jane)





Representing Specialization via Schemas

Method 1:

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

schema	attributes
<i>person</i>	<i>name, street, city</i>
<i>customer</i>	<i>name, credit_rating</i>
<i>employee</i>	<i>name, salary</i>

- Drawback: getting information about, an *employee* requires accessing two relations, the one corresponding to the low-level schema and the one corresponding to the high-level schema





Representing Specialization as Schemas (Cont.)

Method 2:

- Form a schema for each entity set with all local and inherited attributes

schema	attributes
<i>person</i>	<i>name, street, city</i>
<i>customer</i>	<i>name, street, city, credit_rating</i>
<i>employee</i>	<i>name, street, city, salary</i>

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





Schemas Corresponding to Aggregation

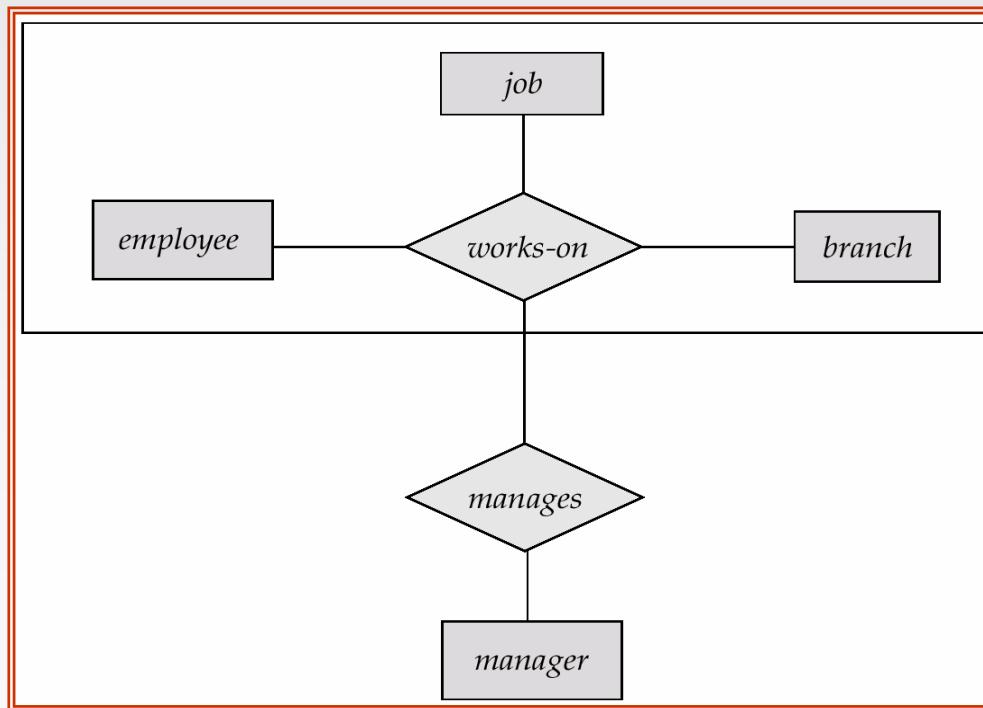
- To represent aggregation, create a schema containing
 - primary key of the aggregated relationship,
 - the primary key of the associated entity set
 - any descriptive attributes





Schemas Corresponding to Aggregation (Cont.)

- For example, to represent aggregation managers between relationship *works_on* and entity set *manager*, create a schema
 - manages* (*employee_id*, *branch_name*, *title*, *manager_name*)
- Schema *works_on* is redundant provided we are willing to store null values for attribute *manager_name* in relation on schema *manages*





UML

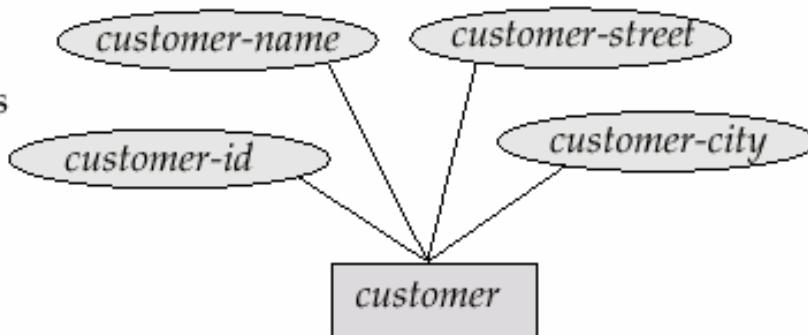
- **UML**: Unified Modeling Language
- UML has many components to graphically model different aspects of an entire software system
- UML Class Diagrams correspond to E-R Diagram, but several differences.





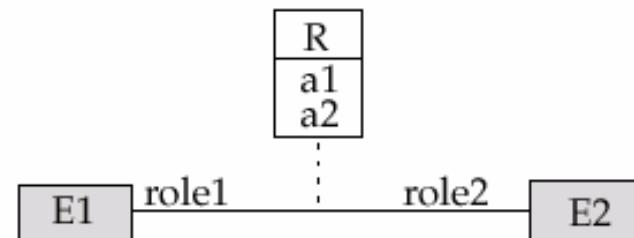
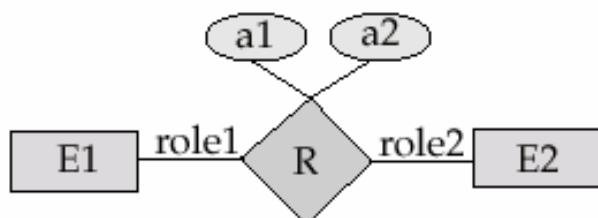
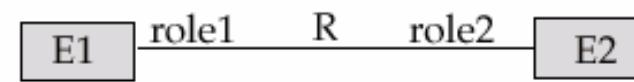
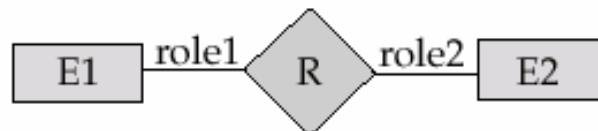
Summary of UML Class Diagram Notation

1. Entity sets
and attributes



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

2. Relationships





UML Class Diagrams (Cont.)

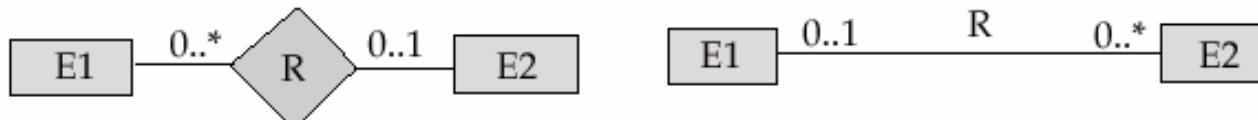
- Entity sets are shown as boxes, and attributes are shown within the box, rather than as separate ellipses in E-R diagrams.
- Binary relationship sets are represented in UML by just drawing a line connecting the entity sets. The relationship set name is written adjacent to the line.
- The role played by an entity set in a relationship set may also be specified by writing the role name on the line, adjacent to the entity set.
- The relationship set name may alternatively be written in a box, along with attributes of the relationship set, and the box is connected, using a dotted line, to the line depicting the relationship set.
- Non-binary relationships drawn using diamonds, just as in ER diagrams



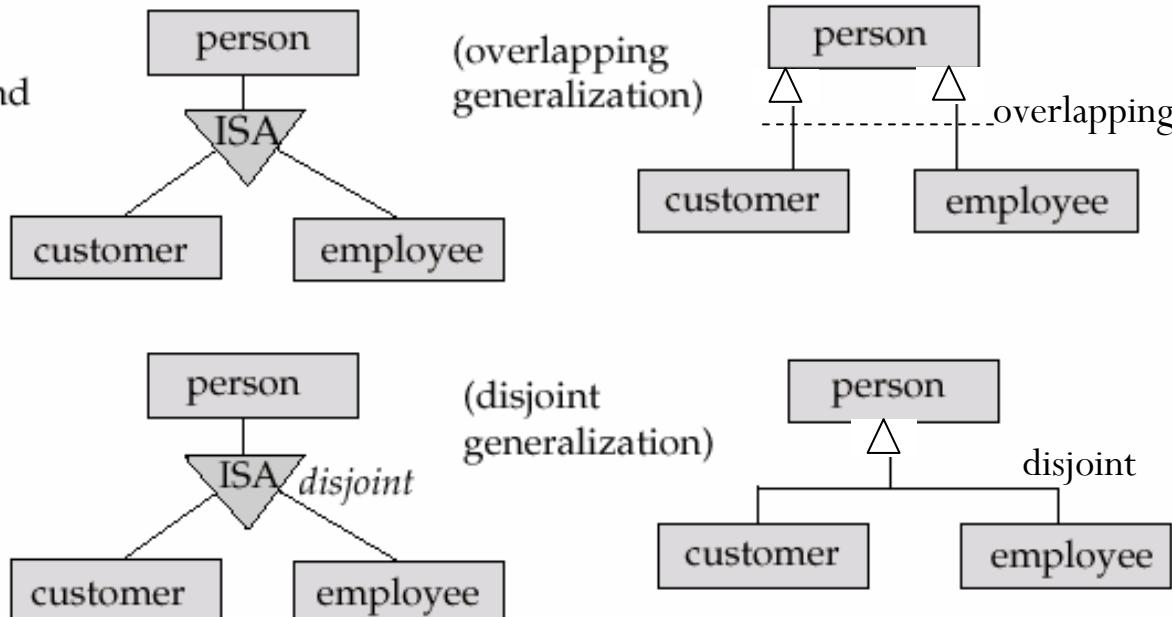


UML Class Diagram Notation (Cont.)

3. Cardinality constraints



4. Generalization and Specialization



*Note reversal of position in cardinality constraint depiction

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





UML Class Diagrams (Contd.)

- Cardinality constraints are specified in the form $l..h$, where l denotes the minimum and h the maximum number of relationships an entity can participate in.
- Beware: the positioning of the constraints is exactly the reverse of the positioning of constraints in E-R diagrams.
- The constraint $0..*$ on the $E2$ side and $0..1$ on the $E1$ side means that each $E2$ entity can participate in at most one relationship, whereas each $E1$ entity can participate in many relationships; in other words, the relationship is many to one from $E2$ to $E1$.
- Single values, such as 1 or $*$ may be written on edges; The single value 1 on an edge is treated as equivalent to $1..1$, while $*$ is equivalent to $0..*$.





End of Chapter 2

Database System Concepts, 5th Ed.

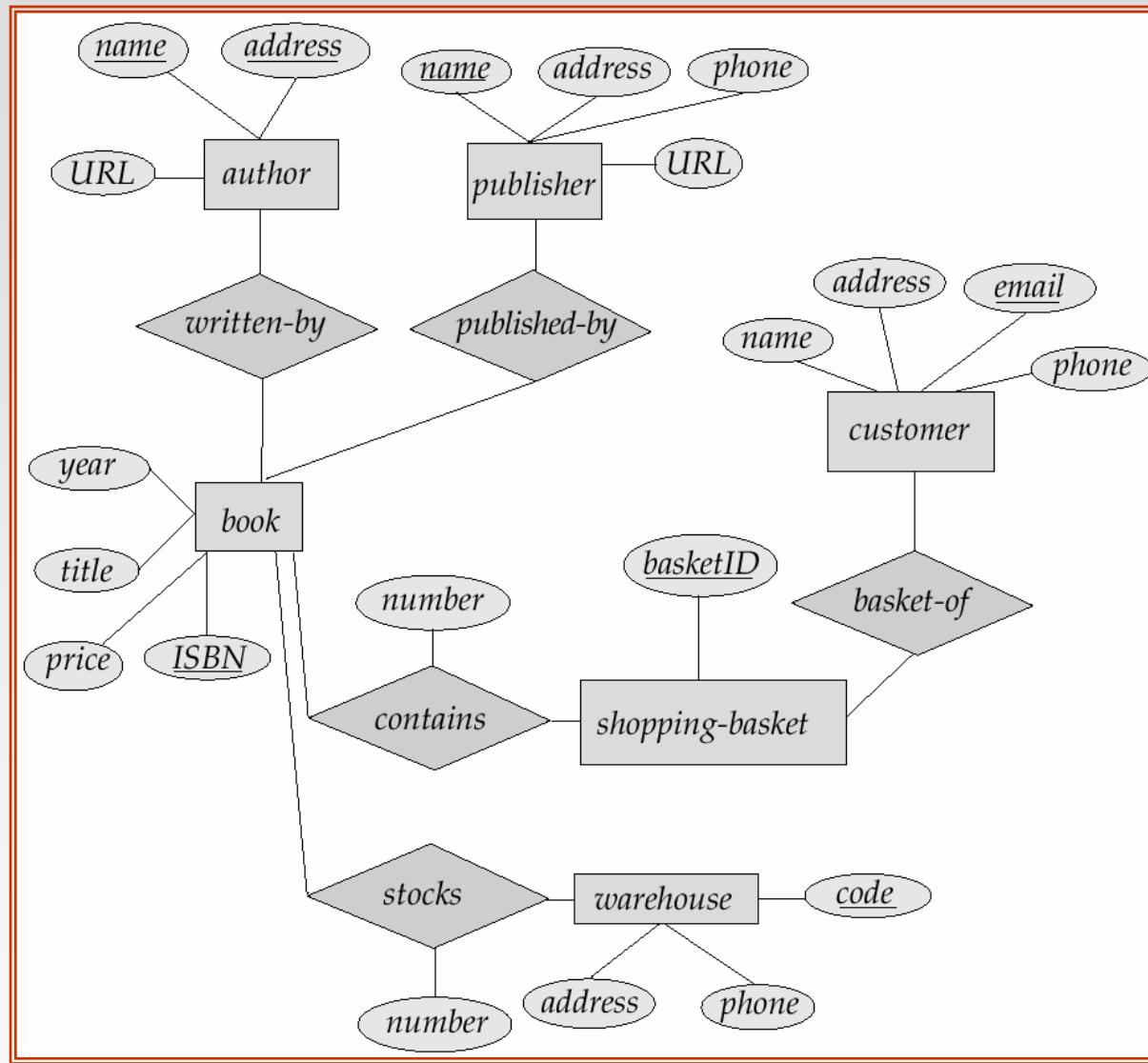
©Silberschatz, Korth and Sudarshan

See www.db-book.com for conditions on re-use



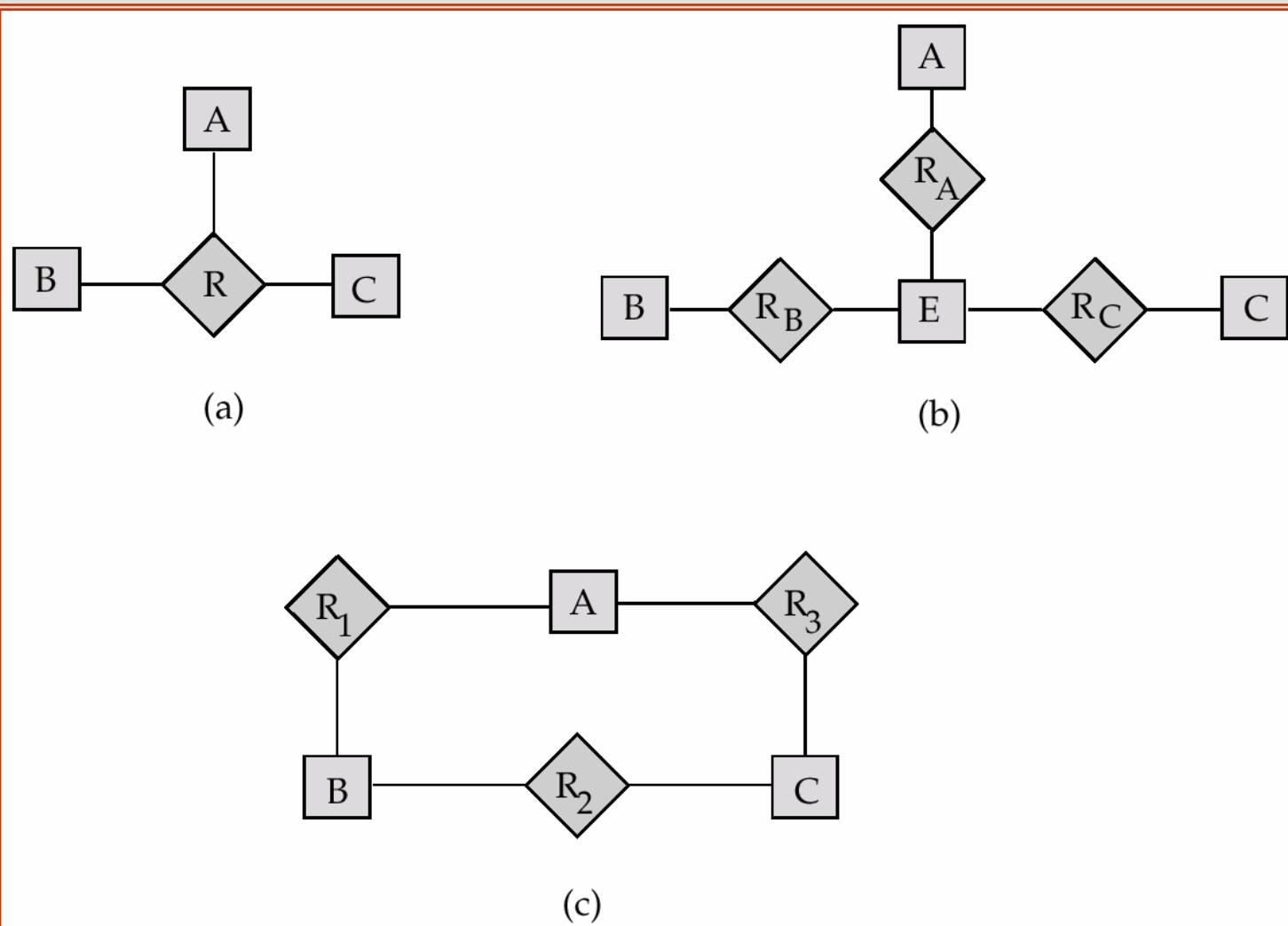


E-R Diagram for Exercise 2.10



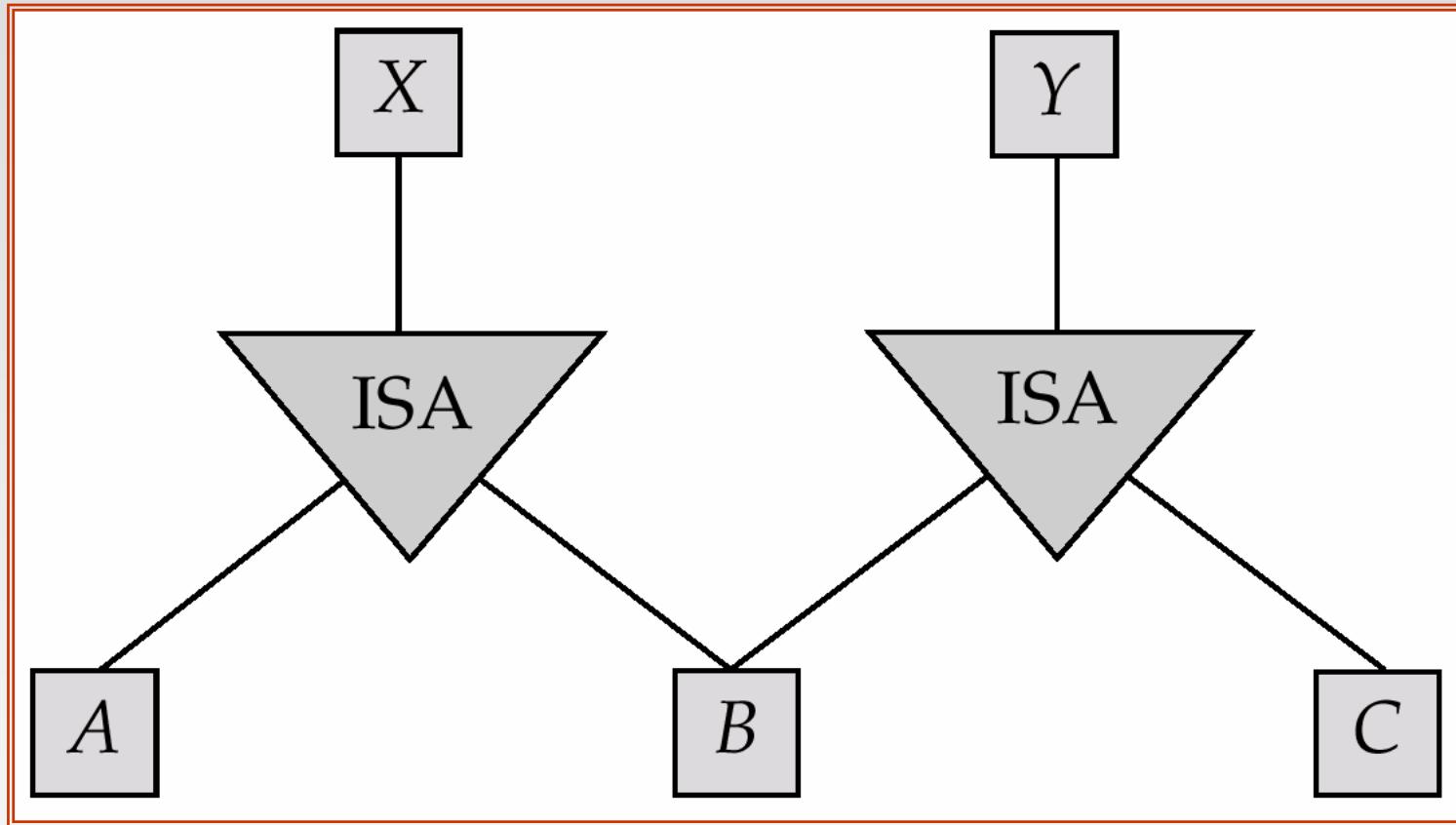


E-R Diagram for Exercise 2.15



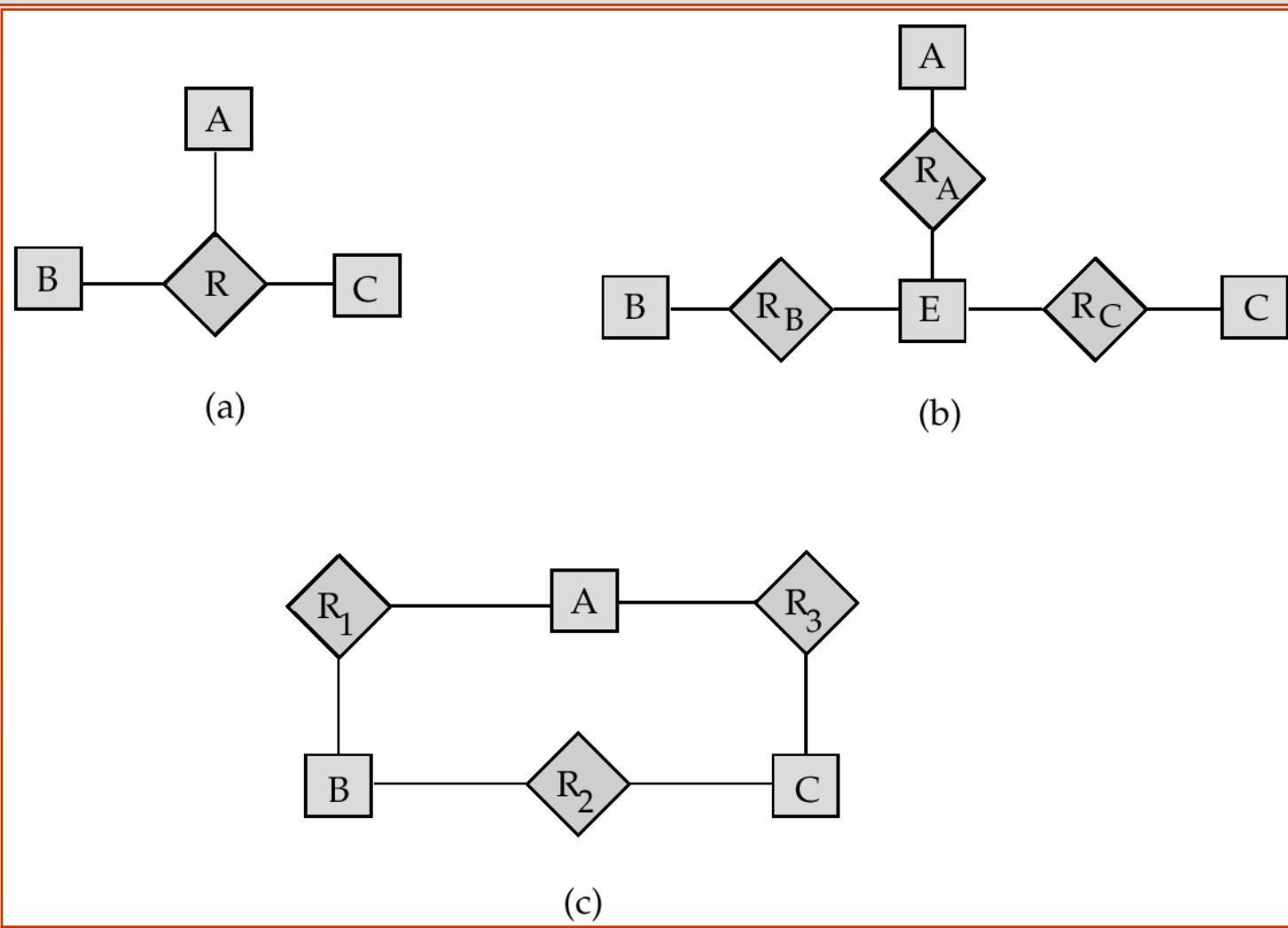


E-R Diagram for Exercise 2.22





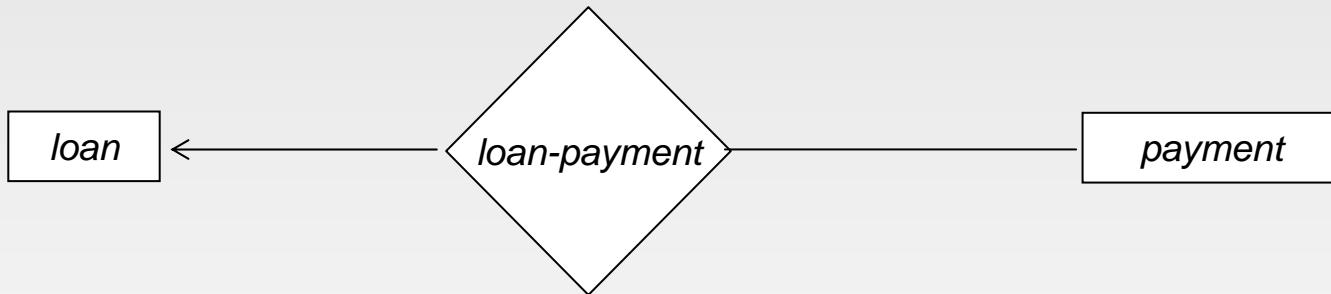
E-R Diagram for Exercise 2.15





Existence Dependencies

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



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





Figure 6.8

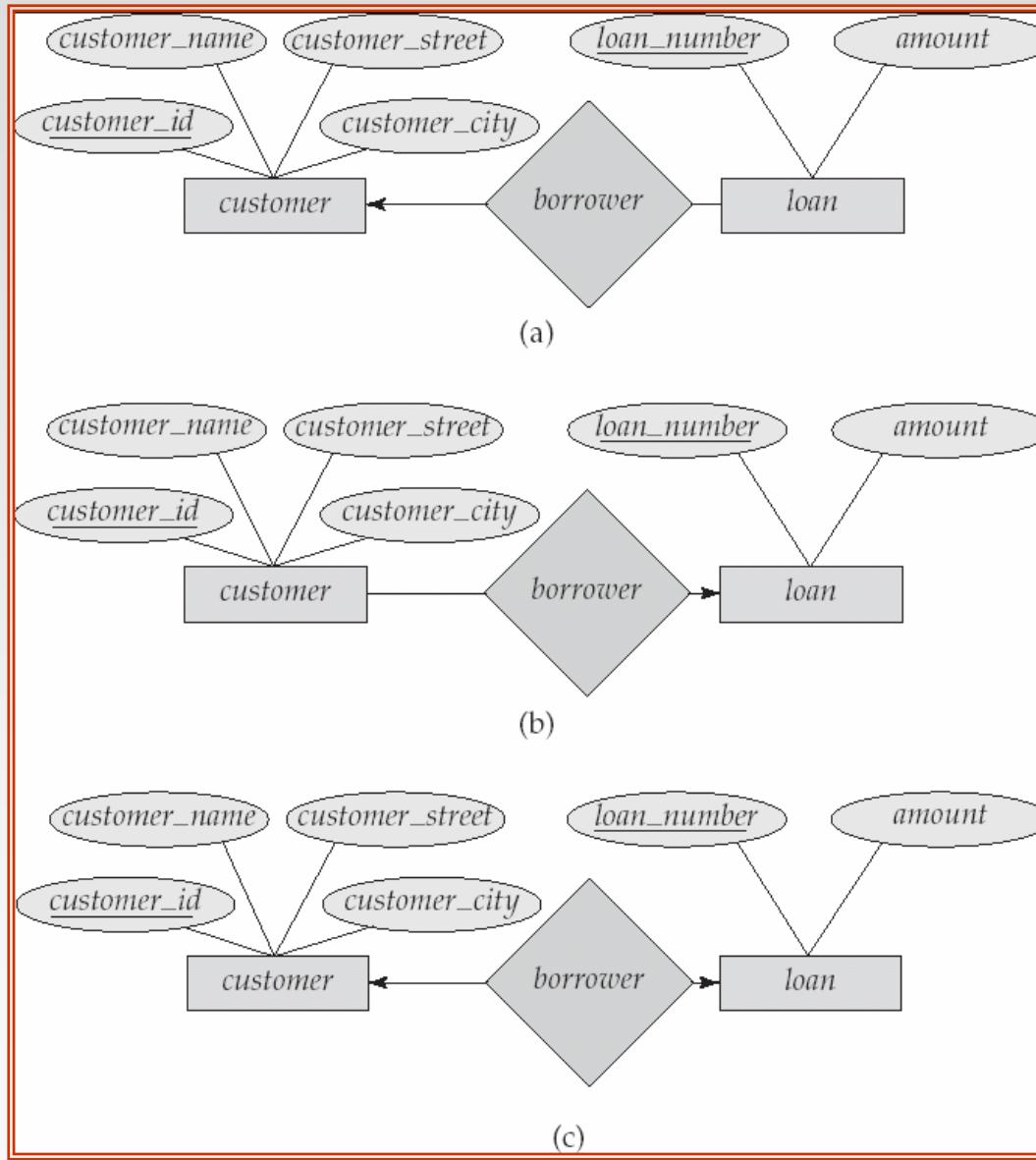
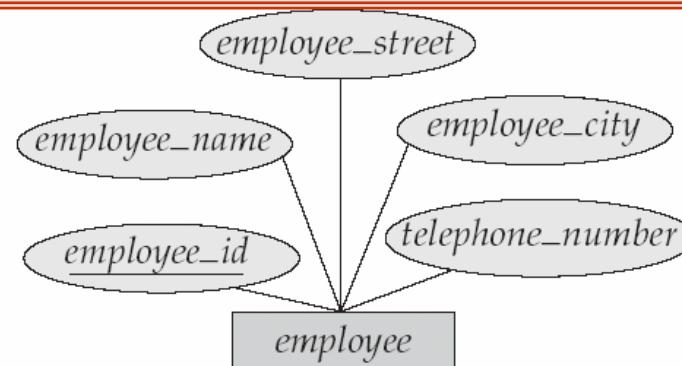
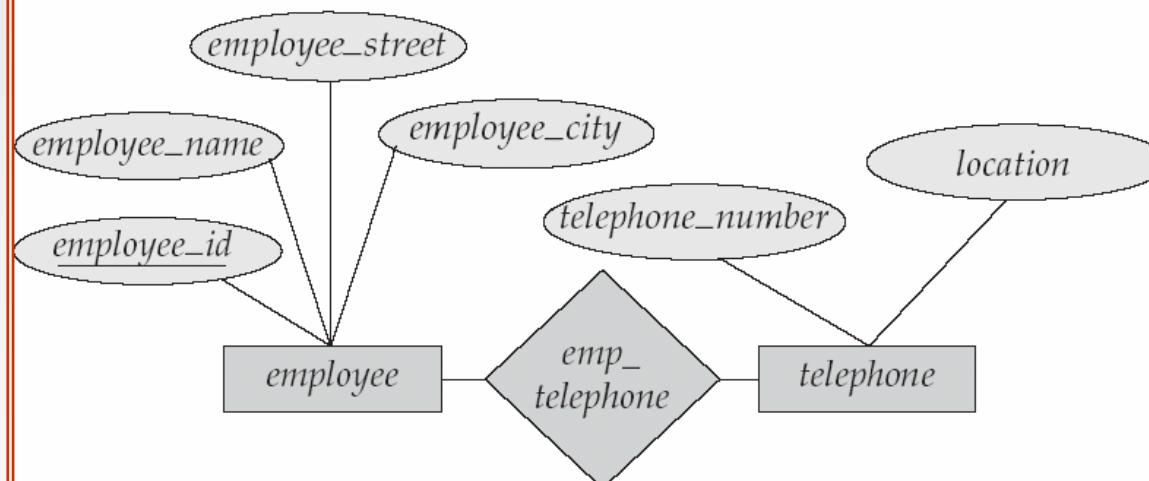




Figure 6.15



(a)



(b)





Figure 6.16

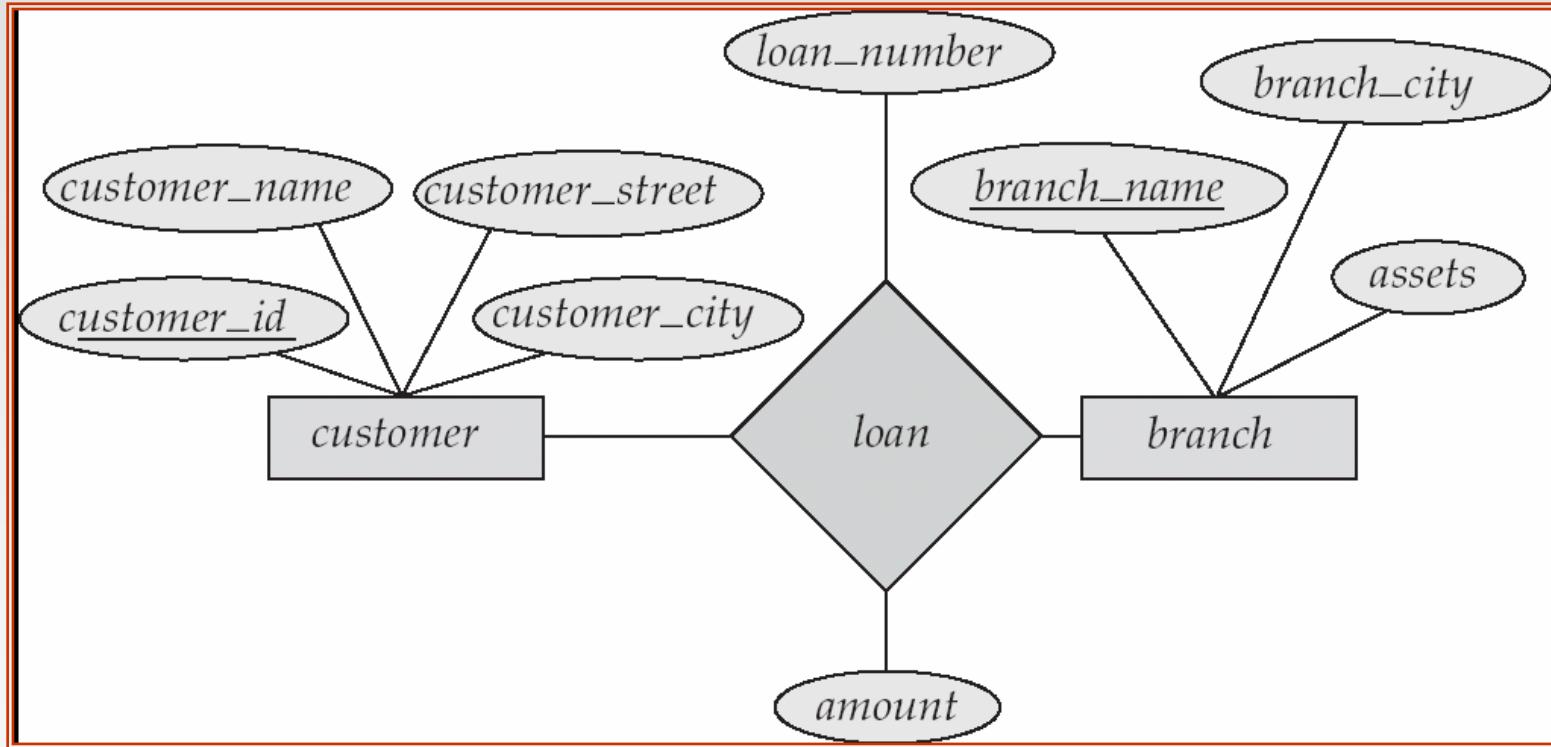




Figure 6.26

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





Figure 6.27

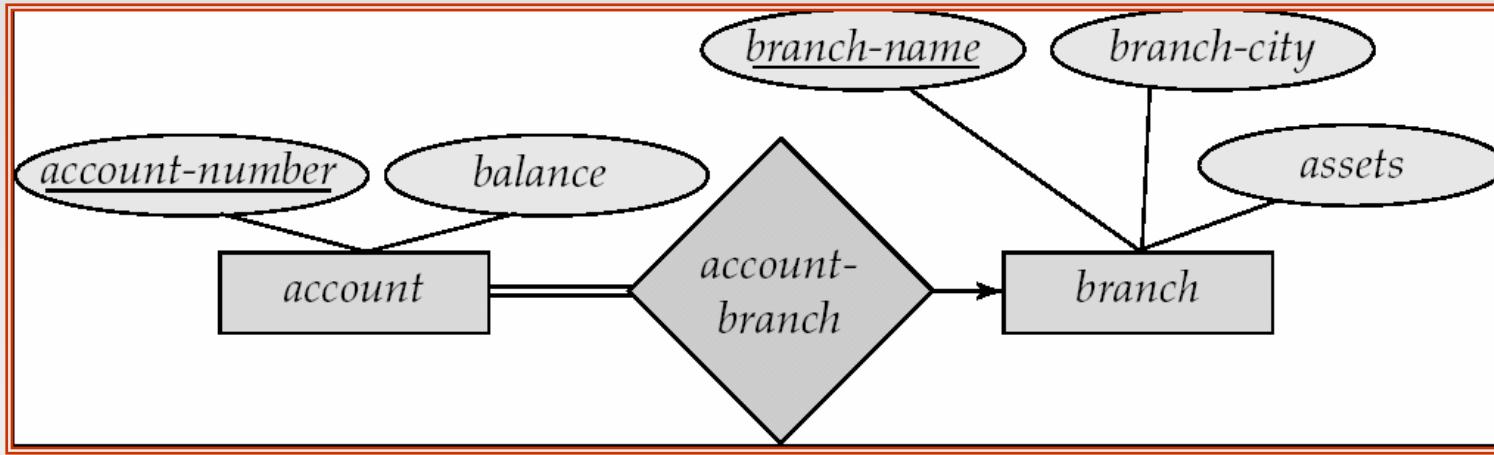




Figure 6.28

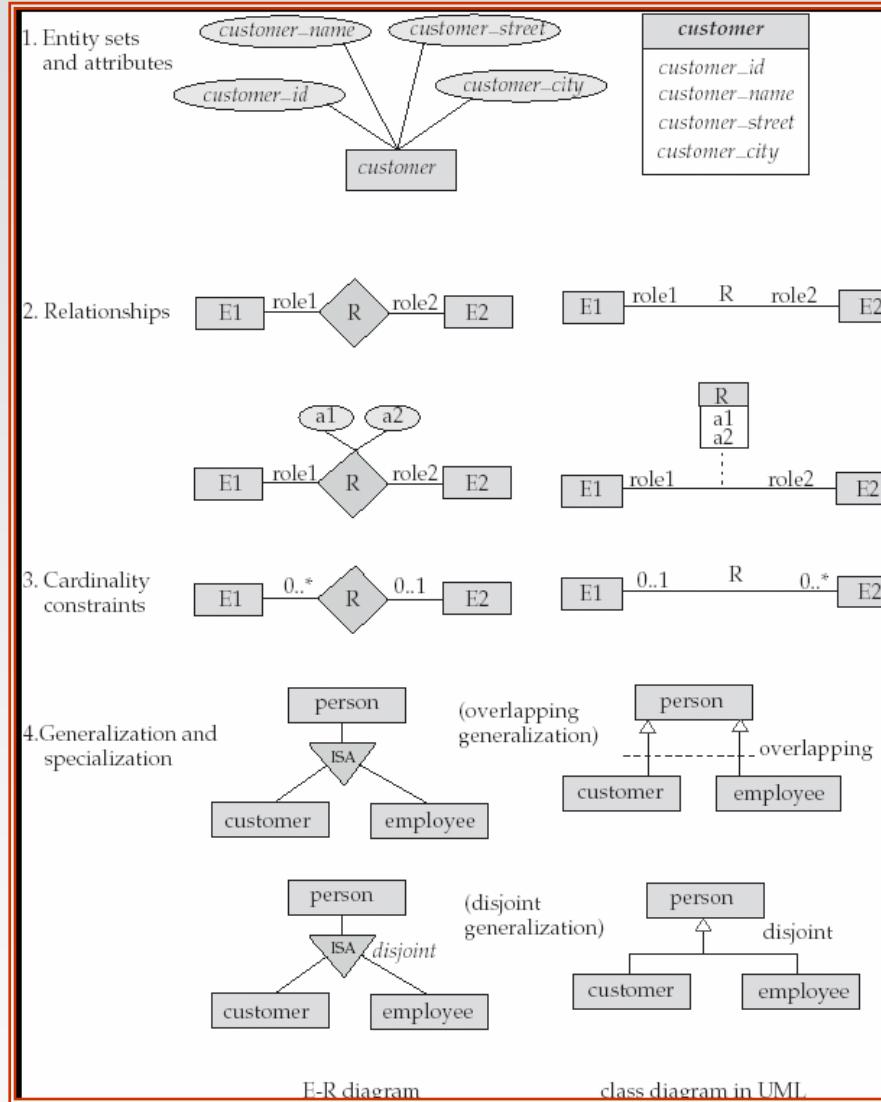




Figure 6.29

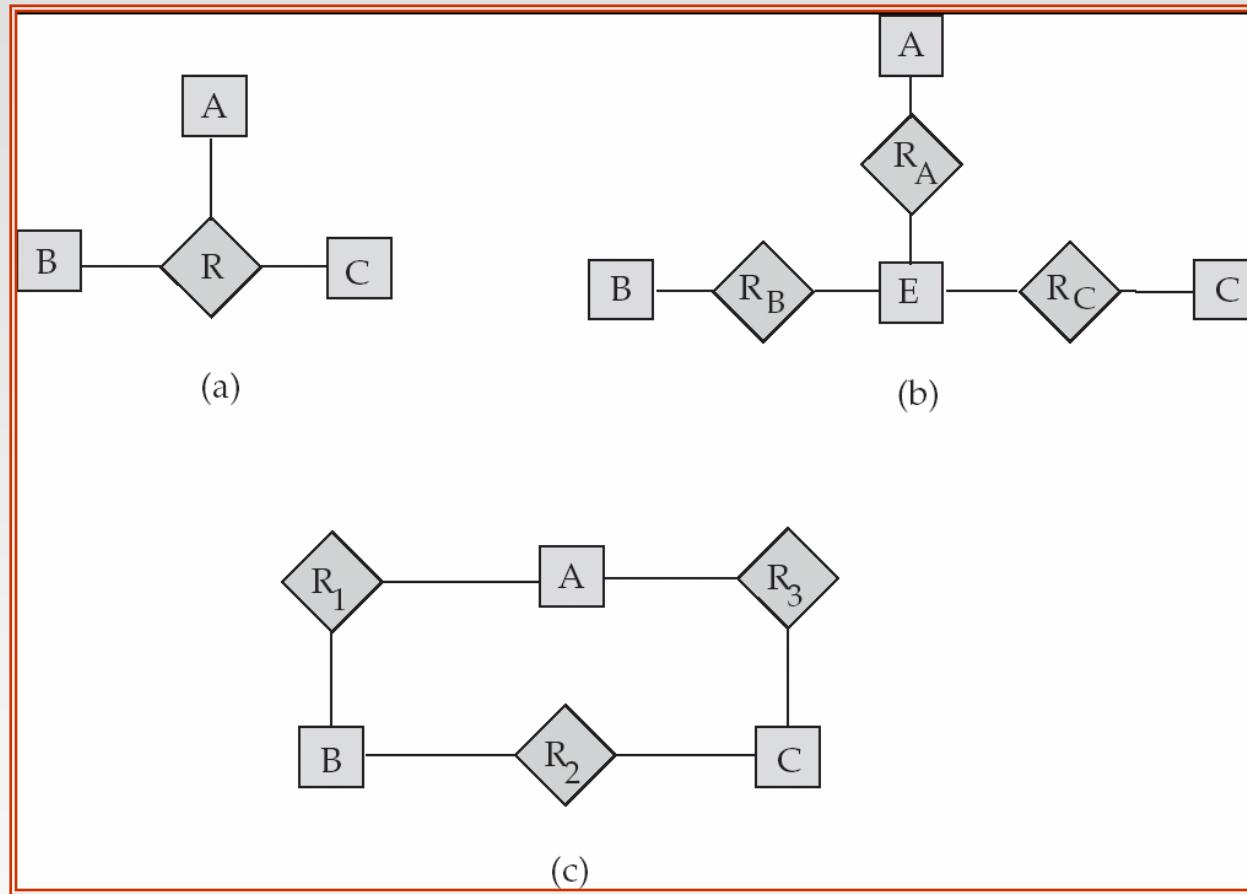




Figure 6.30

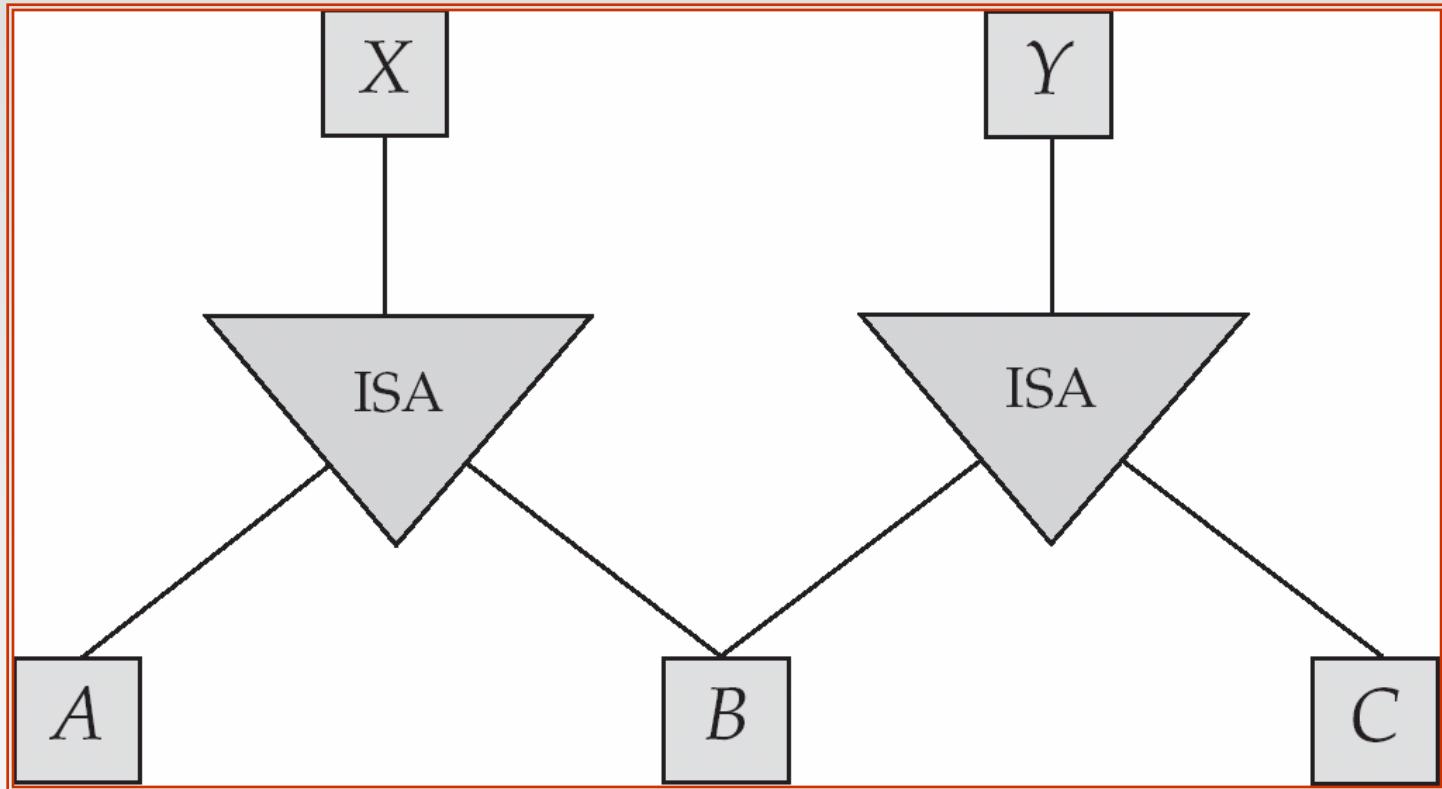
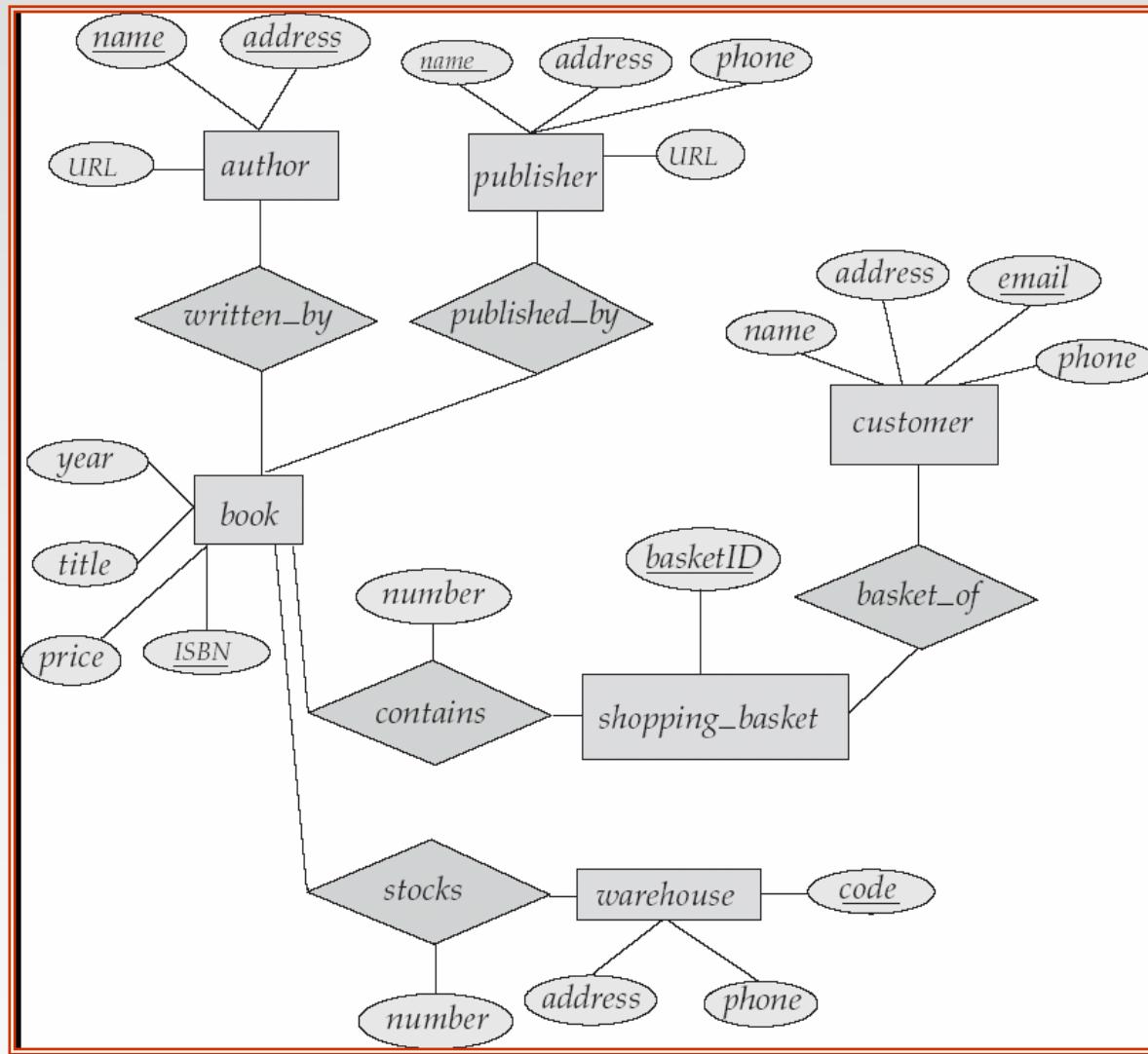




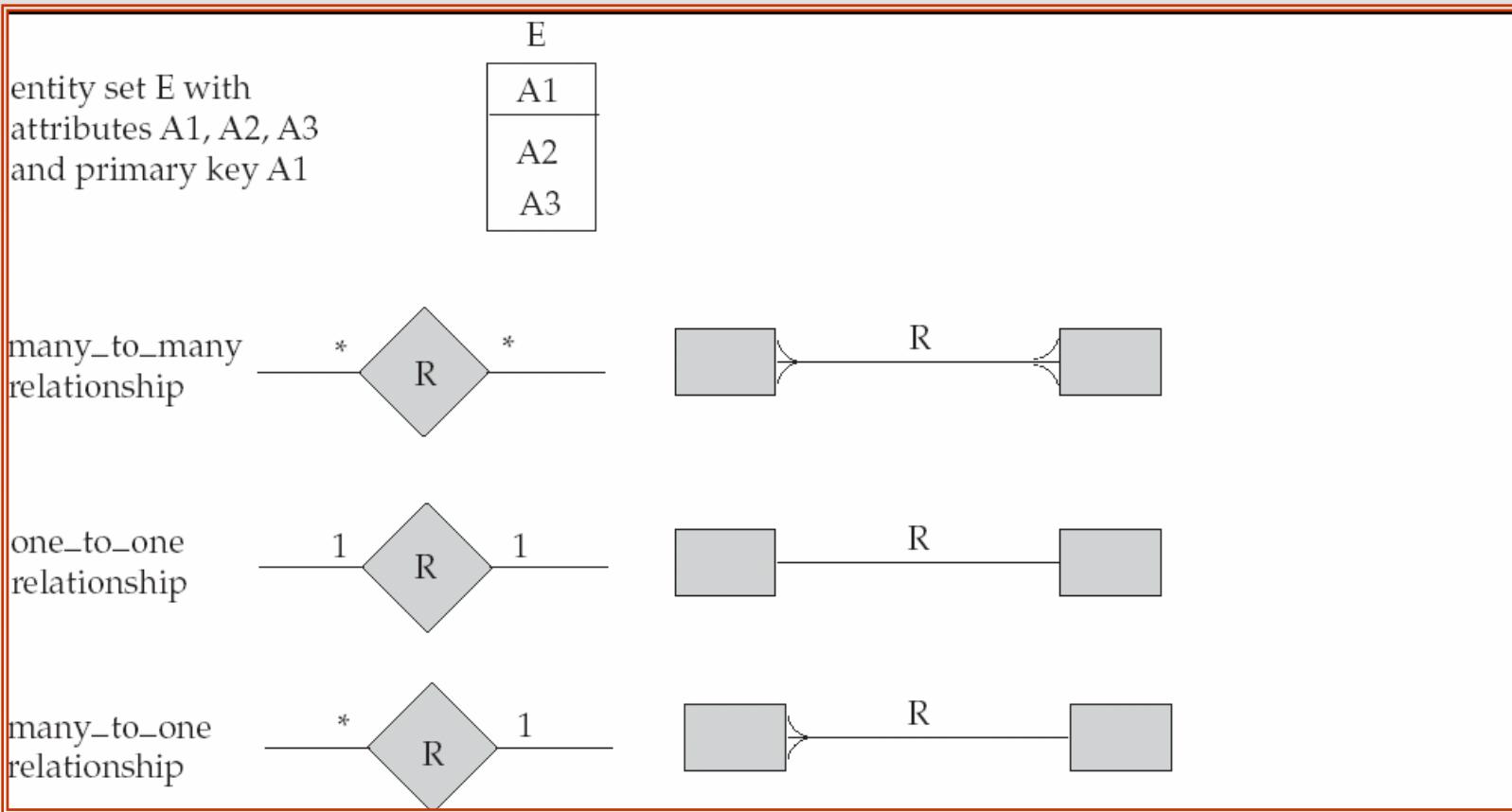
Figure 6.31





Alternative E-R Notations

Figure 6.24





Chapter 2: Relational Model

Database System Concepts, 5th Ed.

©Silberschatz, Korth and Sudarshan
See www.db-book.com for conditions on re-use





Chapter 2: Relational Model

- Structure of Relational Databases
- Fundamental Relational-Algebra-Operations
- Additional Relational-Algebra-Operations
- Extended Relational-Algebra-Operations
- Null Values
- Modification of the Database





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

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

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

is a relation over

$\text{customer_name} \times \text{customer_street} \times \text{customer_city}$





Attribute Types

- Each attribute of a relation has a name
- The set of allowed values for each attribute is called the **domain** of the attribute
- Attribute values are (normally) required to be **atomic**; that is, indivisible
 - E.g. the value of an attribute can be an account number, but cannot be a set of account numbers
- Domain is said to be atomic if all its members are atomic
- The special value *null* is a member of every domain
- The null value causes complications in the definition of many operations
 - We shall ignore the effect of null values in our main presentation and consider their effect later





Relation Schema

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

Example:

Customer_schema = (*customer_name*, *customer_street*, *customer_city*)

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

Example:

customer (*Customer_schema*)





Relation Instance

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

The diagram shows a table with three columns: *customer_name*, *customer_street*, and *customer_city*. The table has four rows of data. Arrows point from the column headers to the label "attributes (or columns)" and from the row data to the label "tuples (or rows)".

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

customer





Relations are Unordered

- Order of tuples is irrelevant (tuples may be stored in an arbitrary order)
- Example: *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
 - account* : stores information about accounts
 - depositor* : stores information about which customer owns which account
 - customer* : stores information about customers
- Storing all information as a single relation such as
bank(account_number, balance, customer_name, ..)
results in
 - repetition of information
 - ▶ e.g., if two customers own an account (What gets repeated?)
 - the need for null values
 - ▶ e.g., to represent a customer without an account
- Normalization theory (Chapter 7) deals with how to design relational schemas





The *customer* Relation

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





The *depositor* Relation

<i>customer_name</i>	<i>account_number</i>
Hayes	A-102
Johnson	A-101
Johnson	A-201
Jones	A-217
Lindsay	A-222
Smith	A-215
Turner	A-305





Keys

- Let $K \subseteq R$
- K is a **superkey** of R if values for K are sufficient to identify a unique tuple of each possible relation $r(R)$
 - by “possible r ” we mean a relation r that could exist in the enterprise we are modeling.
 - Example: $\{customer_name, customer_street\}$ and $\{customer_name\}$ are both superkeys of *Customer*, if no two customers can possibly have the same name
 - ▶ In real life, an attribute such as $customer_id$ would be used instead of $customer_name$ to uniquely identify customers, but we omit it to keep our examples small, and instead assume customer names are unique.





Keys (Cont.)

- K is a **candidate key** if K is minimal

Example: $\{customer_name\}$ is a candidate key for *Customer*, since it is a superkey and no subset of it is a superkey.

- **Primary key:** a candidate key chosen as the principal means of identifying tuples within a relation

- Should choose an attribute whose value never, or very rarely, changes.
- E.g. email address is unique, but may change

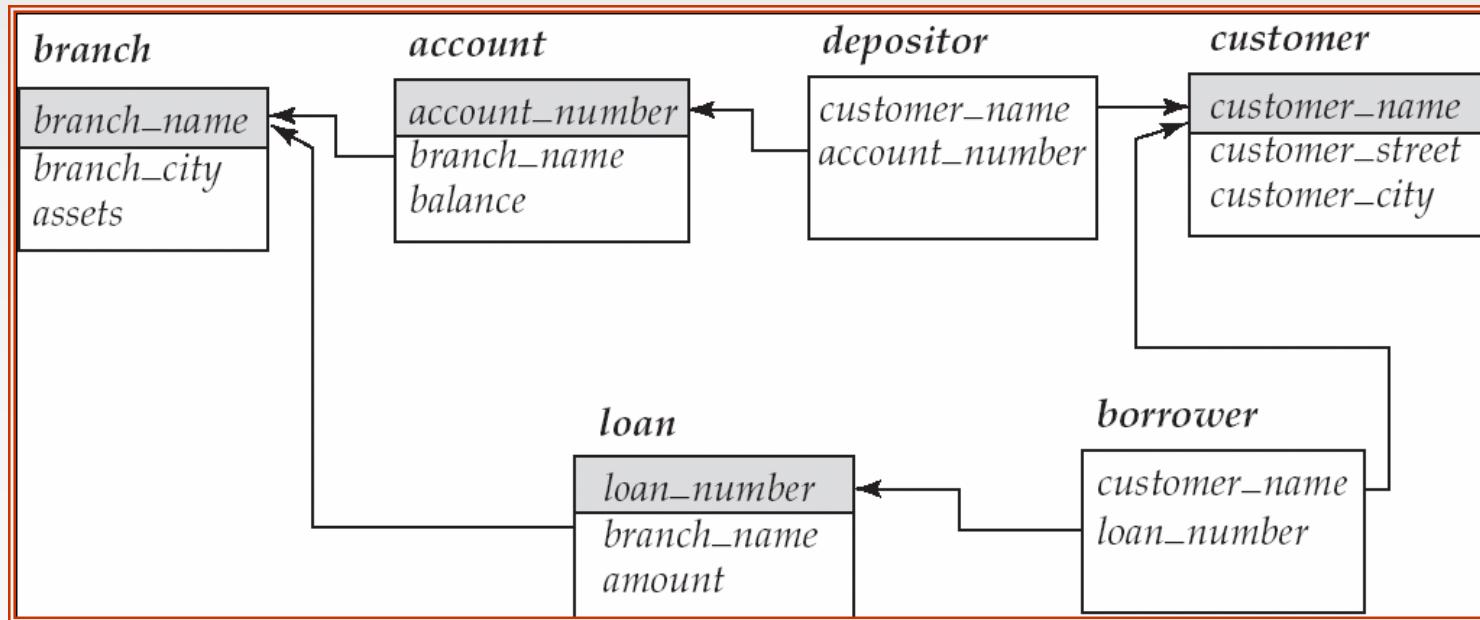




Foreign Keys

- A relation schema may have an attribute that corresponds to the primary key of another relation. The attribute is called a **foreign key**.
 - E.g. *customer_name* and *account_number* attributes of *depositor* are foreign keys to *customer* and *account* respectively.
 - Only values occurring in the primary key attribute of the **referenced relation** may occur in the foreign key attribute of the **referencing relation**.

■ Schema diagram





Query Languages

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





Relational Algebra

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





Select Operation – Example

- Relation r

A	B	C	D
α	α	1	7
α	β	5	7
β	β	12	3
β	β	23	10

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

A	B	C	D
α	α	1	7
β	β	23	10





Select Operation

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

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

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

Each **term** is one of:

<attribute> op <attribute> or <constant>

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

- Example of selection:

$\sigma_{branch_name='Perryridge'}(account)$





Project Operation – Example

- Relation r :

	A	B	C
α	10	1	
α	20	1	
β	30	1	
β	40	2	

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

	A	C
α	1	
α	1	
β	1	
β	2	

=

	A	C
α	1	
β	1	
β	2	





Project Operation

- Notation:

$$\prod_{A_1, A_2, \dots, A_k} (r)$$

where A_1, A_2 are attribute names and r is a relation name.

- The result is defined as the relation of k columns obtained by erasing the columns that are not listed
- Duplicate rows removed from result, since relations are sets
- Example: To eliminate the *branch_name* attribute of *account*

$$\prod_{\text{account_number}, \text{balance}} (\text{account})$$





Union Operation – Example

- Relations r, s :

A	B
α	1
α	2
β	1

r

A	B
α	2
β	3

s

- $r \cup s$:

A	B
α	1
α	2
β	1
β	3





Union Operation

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

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

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

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





Set Difference Operation – Example

- Relations r , s :

A	B
α	1
α	2
β	1

r

A	B
α	2
β	3

s

- $r - s$:

A	B
α	1
β	1





Set Difference Operation

- Notation $r - s$
- Defined as:

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

- Set differences must be taken between **compatible** relations.
 - r and s must have the **same** arity
 - attribute domains of r and s must be compatible





Cartesian-Product Operation – Example

- Relations r, s :

A	B
α	1
β	2

r

C	D	E
α	10	a
β	10	a
β	20	b
γ	10	b

s

- $r \times s$:

A	B	C	D	E
α	1	α	10	a
α	1	β	10	a
α	1	β	20	b
α	1	γ	10	b
β	2	α	10	a
β	2	β	10	a
β	2	β	20	b
β	2	γ	10	b





Cartesian-Product Operation

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

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

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





Composition of Operations

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

A	B	C	D	E
α	1	α	10	a
α	1	β	10	a
α	1	β	20	b
α	1	γ	10	b
β	2	α	10	a
β	2	β	10	a
β	2	β	20	b
β	2	γ	10	b

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

A	B	C	D	E
α	1	α	10	a
β	2	β	10	a
β	2	β	20	b





Rename Operation

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

$$\rho_x(E)$$

returns the expression E under the name X

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

$$\rho_{x(A_1, A_2, \dots, A_n)}(E)$$

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





Banking Example

branch (branch_name, branch_city, assets)

customer (customer_name, customer_street, customer_city)

account (account_number, branch_name, balance)

loan (loan_number, branch_name, amount)

depositor (customer_name, account_number)

borrower (customer_name, loan_number)





Example Queries

- Find all loans of over \$1200

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

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

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

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

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




Example Queries

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

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

- 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.

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




Example Queries

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

- Query 1

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

- Query 2

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




Example Queries

customer_name	loan_number
Adams	L-16
Curry	L-93
Hayes	L-15
Jackson	L-14
Jones	L-17
Smith	L-11
Smith	L-23
Williams	L-17

loan_number	branch_name	amount
L-11	Round Hill	900
L-14	Downtown	1500
L-15	Perryridge	1500
L-16	Perryridge	1300
L-17	Downtown	1000
L-23	Redwood	2000
L-93	Mianus	500

customer_name	loan_number
Adams	L-16
Curry	L-93
Hayes	L-15
Jackson	L-14
Jones	L-17
Smith	L-11
Smith	L-23
Williams	L-17

loan_number	branch_name	amount
L-11	Round Hill	900
L-14	Downtown	1500
L-15	Perryridge	1500
L-16	Perryridge	1300
L-17	Downtown	1000
L-23	Redwood	2000
L-93	Mianus	500





Example Queries

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

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




Formal Definition

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





Additional Operations

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

- Set intersection
- Natural join
- Division
- Assignment





Set-Intersection Operation

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





Set-Intersection Operation – Example

- Relation 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

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

$$\prod_{r.A, r.B, r.C, r.D, s.E} (\sigma_{r.B = s.B \wedge r.D = s.D} (r \times s))$$





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





Division Operation

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

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

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

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

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





Division Operation – Example

- Relations r, s :

A	B
α	1
α	2
α	3
β	1
γ	1
δ	1
δ	3
δ	4
ϵ	6
ϵ	1
β	2

B
1
2

s

- $r \div s$:

A
α
β

r





Another Division Example

- Relations r , s :

A	B	C	D	E
α	a	α	a	1
α	a	γ	a	1
α	a	γ	b	1
β	a	γ	a	1
β	a	γ	b	3
γ	a	γ	a	1
γ	a	γ	b	1
γ	a	β	b	1

r

D	E
a	1
b	1

s

- $r \div s$:

A	B	C
α	a	γ
γ	a	γ





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
- $\Pi_{R-S}(r) \times s$: **all possible pairs** between $R-S(r)$ and s
- $\Pi_{R-S}(r) \times s - \Pi_{R-S,S}(r)$: those pairs **that do not qualify, i.e., do not appear in $r(R)$** .
- $\Pi_{R-S}(\Pi_{R-S}(r) \times s) - \Pi_{R-S,S}(r)$ gives those tuples t in $\Pi_{R-S}(r)$ such that for some tuple $u \in s$, **$tu \notin r$** .





Assignment Operation

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

$$\begin{aligned}temp1 &\leftarrow \Pi_{R-S}(r) \\temp2 &\leftarrow \Pi_{R-S}((temp1 \times s) - \Pi_{R-S,S}(r)) \\result &= temp1 - temp2\end{aligned}$$

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





Bank Example Queries

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

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

- Find the name of all customers who have a loan at the bank and the loan amount

$$\Pi_{customer_name, loan_number, amount} (borrower \bowtie loan)$$




Bank Example Queries

- Find all customers who have an account from at least the “Downtown” **and** the Uptown” branches.
 - Query 1

$$\Pi_{customer_name} (\sigma_{branch_name = \text{``Downtown''}} (depositor \bowtie account)) \cap$$
$$\Pi_{customer_name} (\sigma_{branch_name = \text{``Uptown''}} (depositor \bowtie account))$$

- Query 2

$$\Pi_{customer_name, branch_name} (depositor \bowtie account)$$
$$\div \rho_{temp(branch_name)} (\{(\text{``Downtown''}), (\text{``Uptown''})\})$$

Note that Query 2 uses a constant relation.





Bank Example Queries

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

$$\begin{aligned} & \prod_{customer_name, branch_name} (depositor \bowtie account) \\ & \div \prod_{branch_name} (\sigma_{branch_city = "Brooklyn"} (branch)) \end{aligned}$$




Extended Relational-Algebra-Operations

- Generalized Projection
- Aggregate Functions
- Outer Join





Generalized Projection

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

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

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

$$\Pi_{customer_name, limit - credit_balance}(credit_info)$$





Aggregate Functions and Operations

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

avg: average value

min: minimum value

max: maximum value

sum: sum of values

count: number of values

- **Aggregate operation** in relational algebra

$$G_1, G_2, \dots, G_n \mathcal{V}_{F_1(A_1), F_2(A_2), \dots, F_n(A_n)}(E)$$

E is any relational-algebra expression

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





Aggregate Operation – Example

- Relation r :

A	B	C
α	α	7
α	β	7
β	β	3
β	β	10

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

sum(c)
27





Aggregate Operation – Example

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

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

branch_name $\text{g sum}(\text{balance})$ (*account*)

<i>branch_name</i>	$\text{sum}(\text{balance})$
Perryridge	1300
Brighton	1500
Redwood	700





Aggregate Functions (Cont.)

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

branch_name $\text{g sum(balance) as sum_balance (account)}$





Outer Join

- An extension of the join operation that **avoids loss of information**.
- **Computes the join** and then **adds tuples from one relation that does not match tuples** in the other relation to the result of the join.
- 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

$loan \bowtie borrower$

<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

$loan \text{ }\square\bowtie \text{ } borrower$

<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

$loan \bowtie^R borrower$

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

- Full Outer Join

$loan \bowtie^L borrower$

<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>
L-155	<i>null</i>	<i>null</i>	Hayes





Null Values

- It is possible for tuples to have a null value, denoted by *null*, for some of their attributes
- *null* signifies an **unknown** value or that a value **does not exist**.
- The result of any **arithmetic expression involving *null*** is ***null***.
- **Aggregate functions** simply **ignore null values** (as in SQL)
- For duplicate elimination and grouping, null is treated like any other value, and **two nulls** are **assumed to be the same** (as in SQL)





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: $(\text{unknown} \text{ or } \text{true}) = \text{true}$,
 $(\text{unknown} \text{ or } \text{false}) = \text{unknown}$
 $(\text{unknown} \text{ or } \text{unknown}) = \text{unknown}$
 - 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* evaluates to *unknown*
- Result of select predicate is treated as **false** if it evaluates to *unknown*





Modification of the Database

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





Deletion

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

$$r \leftarrow r - E$$

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





Deletion Examples

- Delete all account records in the Perryridge branch.

$$\text{account} \leftarrow \text{account} - \sigma_{\text{branch_name} = \text{"Perryridge"}(\text{account})}$$

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

$$\text{loan} \leftarrow \text{loan} - \sigma_{\text{amount} \geq 0 \text{ and } \text{amount} \leq 50}(\text{loan})$$

- Delete all accounts at branches located in Needham.

$$r_1 \leftarrow \sigma_{\text{branch_city} = \text{"Needham"}(\text{account} \bowtie \text{branch})}$$
$$r_2 \leftarrow \Pi_{\text{account_number}, \text{branch_name}, \text{balance}}(r_1)$$
$$r_3 \leftarrow \Pi_{\text{customer_name}, \text{account_number}}(r_2 \bowtie \text{depositor})$$
$$\text{account} \leftarrow \text{account} - r_2$$
$$\text{depositor} \leftarrow \text{depositor} - r_3$$




Insertion

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

$$r \leftarrow r \cup E$$

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

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





Insertion Examples

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

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

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

$$r_1 \leftarrow (\sigma_{\text{branch_name} = \text{"Perryridge"}} (\text{borrower} \bowtie \text{loan}))$$
$$\text{account} \leftarrow \text{account} \cup \Pi_{\text{loan_number}, \text{branch_name}, 200} (r_1)$$
$$\text{depositor} \leftarrow \text{depositor} \cup \Pi_{\text{customer_name}, \text{loan_number}} (r_1)$$




Updating

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

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

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





Update Examples

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

$$\text{account} \leftarrow \prod_{\text{account_number}, \text{branch_name}, \text{balance}} \text{balance} * 1.05 (\text{account})$$

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

$$\begin{aligned} \text{account} \leftarrow & \prod_{\text{account_number}, \text{branch_name}, \text{balance}} \text{balance} * 1.06 (\sigma_{\text{BAL} > 10000}(\text{account})) \\ & \cup \prod_{\text{account_number}, \text{branch_name}, \text{balance}} \text{balance} * 1.05 (\sigma_{\text{BAL} \leq 10000}(\text{account})) \end{aligned}$$




End of Chapter 2

Database System Concepts, 5th Ed.

©Silberschatz, Korth and Sudarshan
See www.db-book.com for conditions on re-use





Figure 2.3. The *branch* relation

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





Figure 2.6: The *loan* relation

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





Figure 2.7: The *borrower* relation

<i>customer_name</i>	<i>loan_number</i>
Adams	L-16
Curry	L-93
Hayes	L-15
Jackson	L-14
Jones	L-17
Smith	L-11
Smith	L-23
Williams	L-17





Figure 2.9

Result of $\sigma_{\text{branch_name} = \text{"Perryridge"}}(\text{loan})$

<i>loan_number</i>	<i>branch_name</i>	<i>amount</i>
L-15	Perryridge	1500
L-16	Perryridge	1300





Figure 2.10: Loan number and the amount of the loan

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





Figure 2.11: Names of all customers who have either an account or an loan

<i>customer_name</i>
Adams
Curry
Hayes
Jackson
Jones
Smith
Williams
Lindsay
Johnson
Turner





Figure 2.12: Customers with an account but no loan

customer_name

Johnson
Lindsay
Turner





Figure 2.13: Result of *borrower* |X| *loan*

<i>customer_name</i>	<i>borrower_loan_number</i>	<i>loan_loan_number</i>	<i>branch_name</i>	<i>amount</i>
Adams	L-16	L-11	Round Hill	900
Adams	L-16	L-14	Downtown	1500
Adams	L-16	L-15	Perryridge	1500
Adams	L-16	L-16	Perryridge	1300
Adams	L-16	L-17	Downtown	1000
Adams	L-16	L-23	Redwood	2000
Adams	L-16	L-93	Mianus	500
Curry	L-93	L-11	Round Hill	900
Curry	L-93	L-14	Downtown	1500
Curry	L-93	L-15	Perryridge	1500
Curry	L-93	L-16	Perryridge	1300
Curry	L-93	L-17	Downtown	1000
Curry	L-93	L-23	Redwood	2000
Curry	L-93	L-93	Mianus	500
Hayes	L-15	L-11		900
Hayes	L-15	L-14		1500
Hayes	L-15	L-15		1500
Hayes	L-15	L-16		1300
Hayes	L-15	L-17		1000
Hayes	L-15	L-23		2000
Hayes	L-15	L-93		500
...
...
...
Smith	L-23	L-11	Round Hill	900
Smith	L-23	L-14	Downtown	1500
Smith	L-23	L-15	Perryridge	1500
Smith	L-23	L-16	Perryridge	1300
Smith	L-23	L-17	Downtown	1000
Smith	L-23	L-23	Redwood	2000
Smith	L-23	L-93	Mianus	500
Williams	L-17	L-11	Round Hill	900
Williams	L-17	L-14	Downtown	1500
Williams	L-17	L-15	Perryridge	1500
Williams	L-17	L-16	Perryridge	1300
Williams	L-17	L-17	Downtown	1000
Williams	L-17	L-23	Redwood	2000
Williams	L-17	L-93	Mianus	500





Figure 2.14

<i>customer_name</i>	<i>borrower_loan_number</i>	<i>loan_loan_number</i>	<i>branch_name</i>	<i>amount</i>
Adams	L-16	L-15	Perryridge	1500
Adams	L-16	L-16	Perryridge	1300
Curry	L-93	L-15	Perryridge	1500
Curry	L-93	L-16	Perryridge	1300
Hayes	L-15	L-15	Perryridge	1500
Hayes	L-15	L-16	Perryridge	1300
Jackson	L-14	L-15	Perryridge	1500
Jackson	L-14	L-16	Perryridge	1300
Jones	L-17	L-15	Perryridge	1500
Jones	L-17	L-16	Perryridge	1300
Smith	L-11	L-15	Perryridge	1500
Smith	L-11	L-16	Perryridge	1300
Smith	L-23	L-15	Perryridge	1500
Smith	L-23	L-16	Perryridge	1300
Williams	L-17	L-15	Perryridge	1500
Williams	L-17	L-16	Perryridge	1300

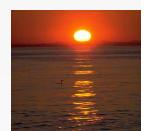




Figure 2.15

<i>customer_name</i>
Adams
Hayes





Figure 2.16

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





Figure 2.17

Largest account balance in the bank

<i>balance</i>
900





Figure 2.18: Customers who live on the same street and in the same city as Smith

<i>customer_name</i>
Curry Smith





Figure 2.19: Customers with both an account and a loan at the bank

<i>customer_name</i>
Hayes
Jones
Smith





Figure 2.20

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





Figure 2.21

branch_name

Brighton
Perryridge





Figure 2.22

branch_name

Brighton
Downtown





Figure 2.23

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





Figure 2.24: The *credit_info* relation

<i>customer_name</i>	<i>limit</i>	<i>credit_balance</i>
Curry	2000	1750
Hayes	1500	1500
Jones	6000	700
Smith	2000	400





Figure 2.25

<i>customer_name</i>	<i>credit_available</i>
Curry	250
Jones	5300
Smith	1600
Hayes	0





Figure 2.26: The *pt_works* relation

<i>employee_name</i>	<i>branch_name</i>	<i>salary</i>
Adams	Perryridge	1500
Brown	Perryridge	1300
Gopal	Perryridge	5300
Johnson	Downtown	1500
Loreena	Downtown	1300
Peterson	Downtown	2500
Rao	Austin	1500
Sato	Austin	1600





Figure 2.27
The *pt_works* relation after regrouping

<i>employee_name</i>	<i>branch_name</i>	<i>salary</i>
Rao	Austin	1500
Sato	Austin	1600
Johnson	Downtown	1500
Loreena	Downtown	1300
Peterson	Downtown	2500
Adams	Perryridge	1500
Brown	Perryridge	1300
Gopal	Perryridge	5300





Figure 2.28

<i>branch_name</i>	<i>sum of salary</i>
Austin	3100
Downtown	5300
Perryridge	8100





Figure 2.29

<i>branch_name</i>	<i>sum_salary</i>	<i>max_salary</i>
Austin	3100	1600
Downtown	5300	2500
Perryridge	8100	5300





Figure 2.30

The *employee* and *ft_works* relations

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

<i>employee_name</i>	<i>branch_name</i>	<i>salary</i>
Coyote	Mesa	1500
Rabbit	Mesa	1300
Gates	Redmond	5300
Williams	Redmond	1500





Figure 2.31

<i>employee_name</i>	<i>street</i>	<i>city</i>	<i>branch_name</i>	<i>salary</i>
Coyote	Toon	Hollywood	Mesa	1500
Rabbit	Tunnel	Carrotville	Mesa	1300
Williams	Seaview	Seattle	Redmond	1500





Figure 2.32

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





Figure 2.33

<i>employee_name</i>	<i>street</i>	<i>city</i>	<i>branch_name</i>	<i>salary</i>
Coyote	Toon	Hollywood	Mesa	1500
Rabbit	Tunnel	Carrotville	Mesa	1300
Williams	Seaview	Seattle	Redmond	1500
Gates	<i>null</i>	<i>null</i>	Redmond	5300





Figure 2.34

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





Chapter 3: SQL

Database System Concepts, 5th Ed.

©Silberschatz, Korth and Sudarshan

See www.db-book.com for conditions on re-use





Chapter 3: SQL

- Data Definition
- Basic Query Structure
- Set Operations
- Aggregate Functions
- Null Values
- Nested Subqueries
- Complex Queries
- Views
- Modification of the Database
- Joined Relations**





History

- IBM Sequel language developed as part of System R project at the IBM San Jose Research Laboratory
- Renamed Structured Query Language (SQL)
- ANSI and ISO standard SQL:
 - SQL-86
 - SQL-89
 - SQL-92
 - SQL:1999 (language name became Y2K compliant!)
 - SQL:2003
- Commercial systems offer most, if not all, SQL-92 features, plus varying feature sets from later standards and special proprietary features.
 - Not all examples here may work on your particular system.





Data Definition Language

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

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





Domain Types in SQL

- **char(n).** Fixed length character string, with user-specified length n .
- **varchar(n).** Variable length character strings, with user-specified maximum length n .
- **int.** Integer (a finite subset of the integers that is machine-dependent).
- **smallint.** Small integer (a machine-dependent subset of the integer domain type).
- **numeric(p,d).** Fixed point number, with user-specified precision of p digits, with n digits to the right of decimal point.
- **real, double precision.** Floating point and double-precision floating point numbers, with machine-dependent precision.
- **float(n).** Floating point number, with user-specified precision of at least n digits.
- More are covered in Chapter 4.





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 A_i is an attribute name in the schema of relation r
- D_i is the data type of values in the domain of attribute A_i

- Example:

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





Integrity Constraints in Create Table

- **not null**
- **primary key** (A_1, \dots, A_n)

Example: Declare *branch_name* as the primary key for *branch*

```
create table branch
  (branch_name char(15),
   branch_city  char(30),
   assets       integer,
   primary key (branch_name))
```

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





Drop and Alter Table Constructs

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

alter table r add $A D$

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

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





Basic Query Structure

- SQL is based on set and relational operations with certain modifications and enhancements
- A typical SQL query has the form:

```
select A1, A2, ..., An  
from r1, r2, ..., rm  
where P
```

- A_i represents an attribute
 - R_j represents a relation
 - P is a predicate.
- This query is equivalent to the relational algebra expression.

$$\prod_{A_1, A_2, \dots, A_n} (\sigma_P(r_1 \times r_2 \times \dots \times r_m))$$

- The result of an SQL query is a relation.





The select Clause

- The **select** clause lists the attributes desired in the result of a query
 - corresponds to the projection operation of the relational algebra

- Example: find the names of all branches in the *loan* relation:

```
select branch_name  
from loan
```

- In the relational algebra, the query would be:

$$\Pi_{branch_name} (loan)$$

- NOTE: SQL names are case insensitive (i.e., you may use upper- or lower-case letters.)

- E.g. *Branch_Name* \equiv *BRANCH_NAME* \equiv *branch_name*
- Some people use upper case wherever we use bold font.





The select Clause (Cont.)

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

```
select distinct branch_name  
from loan
```

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

```
select all branch_name  
from loan
```





The select Clause (Cont.)

- An asterisk in the select clause denotes “all attributes”

```
select *
from /loan
```

- The **select** clause can contain arithmetic expressions involving the operation, +, −, *, and /, and operating on constants or attributes of tuples.
- The query:

```
select /loan_number, /branch_name, /amount * 100
from /loan
```

would return a relation that is the same as the */loan* relation, except that the value of the attribute *amount* is multiplied by 100.





The where Clause

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

```
select loan_number  
from loan  
where branch_name = 'Perryridge' and amount > 1200
```

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





The where Clause (Cont.)

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

```
select loan_number
      from loan
     where amount between 90000 and 100000
```





The from Clause

- The **from** clause lists the relations involved in the query
 - Corresponds to the Cartesian product operation of the relational algebra.

- Find the Cartesian product *borrower X loan*

```
select *
  from borrower, loan
```

- Find the name, loan number and loan amount of all customers having a loan at the Perryridge branch.

```
select customer_name, borrower.loan_number, amount
      from borrower, loan
     where borrower.loan_number = loan.loan_number and
           branch_name = 'Perryridge'
```





The Rename Operation

- The SQL allows renaming relations and attributes using the **as** clause:
old-name as new-name
- Find the name, loan number and loan amount of all customers; rename the column name *loan_number* as *loan_id*.

```
select customer_name, borrower.loan_number as loan_id, amount  
from borrower, loan  
where borrower.loan_number = loan.loan_number
```





Tuple Variables

- Tuple variables are defined in the **from** clause via the use of the **as** clause.
- Find the customer names and their loan numbers for all customers having a loan at some branch.

```
select customer_name, T.loan_number, S.amount  
      from borrower as T, loan as S  
     where T.loan_number = S.loan_number
```

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

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

- Keyword **as** is optional and may be omitted

borrower as T ≡ borrower T





String Operations

- SQL includes a string-matching operator for comparisons on character strings. The operator “like” uses patterns that are described using two special characters:
 - percent (%). The % character matches any substring.
 - underscore (_). The _ character matches any character.
- Find the names of all customers whose street includes the substring “Main”.

```
select customer_name  
from customer  
where customer_street like '% Main%'
```

- Match the name “Main%”

```
like 'Main\%' escape '\'
```
- SQL supports a variety of string operations such as
 - concatenation (using “||”)
 - converting from upper to lower case (and vice versa)
 - finding string length, extracting substrings, etc.





Ordering the Display of Tuples

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

```
select distinct customer_name  
from   borrower, loan  
where borrower loan_number = loan.loan_number and  
      branch_name = 'Perryridge'  
order by customer_name
```

- We may specify **desc** for descending order or **asc** for ascending order, for each attribute; ascending order is the default.
 - Example: **order by customer_name desc**





Duplicates

- In relations with duplicates, SQL can define how many copies of tuples appear in the result.
- **Multiset** versions of some of the relational algebra operators – given multiset relations r_1 and r_2 :
 1. $\sigma_\theta(r_1)$: If there are c_1 copies of tuple t_1 in r_1 , and t_1 satisfies selections σ_θ , then there are c_1 copies of t_1 in $\sigma_\theta(r_1)$.
 2. $\Pi_A(r)$: For each copy of tuple t_1 in r_1 , there is a copy of tuple $\Pi_A(t_1)$ in $\Pi_A(r_1)$ where $\Pi_A(t_1)$ denotes the projection of the single tuple t_1 .
 3. $r_1 \times r_2$: If there are c_1 copies of tuple t_1 in r_1 and c_2 copies of tuple t_2 in r_2 , there are $c_1 \times c_2$ copies of the tuple $t_1 \cdot t_2$ in $r_1 \times r_2$





Duplicates (Cont.)

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

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

- Then $\Pi_B(r_1)$ would be $\{(a), (a)\}$, while $\Pi_B(r_1) \times r_2$ would be $\{(a, 2), (a, 2), (a, 3), (a, 3), (a, 3), (a, 3)\}$
- SQL duplicate semantics:

```
select A1, A2, ..., An
from r1, r2, ..., rm
where P
```

is equivalent to the *multiset* version of the expression:

$$\prod_{A_1, A_2, \dots, A_n} (\sigma_P (r_1 \times r_2 \times \dots \times r_m))$$





Set Operations

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

Suppose a tuple occurs m times in r and n times in s , then, it occurs:

- $m + n$ times in r **union all** s
- $\min(m,n)$ times in r **intersect all** s
- $\max(0, m - n)$ times in r **except all** s





Set Operations

- Find all customers who have a loan, an account, or both:

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

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

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

- Find all customers who have an account but no loan.

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





Aggregate Functions

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

avg: average value

min: minimum value

max: maximum value

sum: sum of values

count: number of values





Aggregate Functions (Cont.)

- Find the average account balance at the Perryridge branch.

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

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

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

- Find the number of depositors in the bank.

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





Aggregate Functions – Group By

- Find the number of depositors for each branch.

```
select branch_name, count (distinct customer_name)
  from depositor, account
 where depositor.account_number = account.account_number
   group by branch_name
```

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





Aggregate Functions – Having Clause

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

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

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





Null Values

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

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

- The result of any arithmetic expression involving *null* is *null*
 - Example: $5 + \text{null}$ returns null
- However, aggregate functions simply ignore nulls
 - More on next slide





Null Values and Three Valued Logic

- Any comparison with *null* returns *unknown*
 - Example: $5 < \text{null}$ or $\text{null} < > \text{null}$ or $\text{null} = \text{null}$
- Three-valued logic using the truth value *unknown*:
 - OR: (*unknown or true*) = *true*,
(*unknown or false*) = *unknown*
(*unknown or unknown*) = *unknown*
 - AND: (*true and unknown*) = *unknown*,
(*false and unknown*) = *false*,
(*unknown and unknown*) = *unknown*
 - NOT: (*not unknown*) = *unknown*
 - “*P is unknown*” evaluates to *true* if predicate *P* evaluates to *unknown*
- Result of **where** clause predicate is treated as *false* if it evaluates to *unknown*





Null Values and Aggregates

- Total all loan amounts

```
select sum (amount )
      from loan
```

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





Nested Subqueries

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





Example Query

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

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

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

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





Example Query

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

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

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





Set Comparison

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

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

- Same query using > **some** clause

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





Definition of Some Clause

- $F \text{ <comp> } \text{some } r \Leftrightarrow \exists t \in r \text{ such that } (F \text{ <comp> } t)$
Where <comp> can be: <, \leq , $>$, $=$, \neq

$(5 < \text{some} \begin{array}{|c|} \hline 0 \\ \hline 5 \\ \hline 6 \\ \hline \end{array}) = \text{true}$ (read: 5 < some tuple in the relation)

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

$(5 = \text{some} \begin{array}{|c|} \hline 0 \\ \hline 5 \\ \hline \end{array}) = \text{true}$

$(5 \neq \text{some} \begin{array}{|c|} \hline 0 \\ \hline 5 \\ \hline \end{array}) = \text{true}$ (since $0 \neq 5$)

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

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





Example Query

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

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





Definition of all Clause

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

(5 < all) = false

0
5
6

(5 < all) = true

6
10

(5 = all) = false

4
5

(5 ≠ all) = true (since $5 \neq 4$ and $5 \neq 6$)

4
6

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





Test for Empty Relations

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





Example Query

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

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

- Note that $X - Y = \emptyset \Leftrightarrow X \subseteq Y$
- Note: Cannot write this query using = all and its variants





Test for Absence of Duplicate Tuples

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

```
select T.customer_name
from depositor as T
where unique (
    select R.customer_name
    from account, depositor as R
    where T.customer_name = R.customer_name and
          R.account_number = account.account_number and
          account.branch_name = 'Perryridge')
```





Example Query

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

```
select distinct T.customer_name
from depositor as T
where not unique (
    select R.customer_name
    from account, depositor as R
    where T.customer_name = R.customer_name and
          R.account_number = account.account_number and
          account.branch_name = 'Perryridge')
```

- Variable from outer level is known as a **correlation variable**





Derived Relations

- SQL allows a subquery expression to be used in the **from** clause
- Find the average account balance of those branches where the average account balance is greater than \$1200.

```
select branch_name, avg_balance
  from (select branch_name, avg (balance)
         from account
        group by branch_name )
       as branch_avg ( branch_name, avg_balance )
  where avg_balance > 1200
```

Note that we do not need to use the **having** clause, since we compute the temporary (view) relation *branch_avg* in the **from** clause, and the attributes of *branch_avg* can be used directly in the **where** clause.





With Clause

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

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





Complex Queries using With Clause

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

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





Views

- In some cases, it is not desirable for all users to see the entire logical model (that is, all the actual relations stored in the database.)
- Consider a person who needs to know a customer's name, loan number and branch name, but has no need to see the loan amount. This person should see a relation described, in SQL, by

```
(select customer_name, borrower.loan_number, branch_name  
      from borrower, loan  
     where borrower.loan_number = loan.loan_number )
```

- A **view** provides a mechanism to hide certain data from the view of certain users.
- Any relation that is not of the conceptual model but is made visible to a user as a “virtual relation” is called a **view**.





View Definition

- A view is defined using the **create view** statement which has the form

```
create view v as < query expression >
```

where <query expression> is any legal SQL expression. The view name is represented by *v*.

- Once a view is defined, the view name can be used to refer to the virtual relation that the view generates.
- When a view is created, the query expression is stored in the database; the expression is substituted into queries using the view.





Example Queries

- A view consisting of branches and their customers

```
create view all_customer as
    (select branch_name, customer_name
     from depositor, account
      where depositor.account_number =
            account.account_number )
    union
    (select branch_name, customer_name
     from borrower, loan
      where borrower.loan_number = loan.loan_number )
```

- Find all customers of the Perryridge branch

```
select customer_name
      from all_customer
     where branch_name = 'Perryridge'
```





Views Defined Using Other Views

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





View Expansion

- A way to define the meaning of views defined in terms of other views.
- Let view v_1 be defined by an expression e_1 that may itself contain uses of view relations.
- View expansion of an expression repeats the following replacement step:

repeat

 Find any view relation v_i in e_1

 Replace the view relation v_i by the expression defining v_i

until no more view relations are present in e_1

- As long as the view definitions are not recursive, this loop will terminate





Modification of the Database – Deletion

- Delete all account tuples at the Perryridge branch

```
delete from account
where branch_name = 'Perryridge'
```

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

```
delete from account
where branch_name in (select branch_name
from branch
where branch_city = 'Needham')
```





Example Query

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

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

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





Modification of the Database – Insertion

- Add a new tuple to *account*

```
insert into account  
values ('A-9732', 'Perryridge', 1200)
```

or equivalently

```
insert into account (branch_name, balance, account_number)  
values ('Perryridge', 1200, 'A-9732')
```

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

```
insert into account  
values ('A-777','Perryridge', null )
```





Modification of the Database – Insertion

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

insert into account

```
select loan_number, branch_name, 200  
from loan  
where branch_name = 'Perryridge'
```

insert into depositor

```
select customer_name, loan_number  
from loan, borrower  
where branch_name = 'Perryridge'  
and loan.account_number = borrower.account_number
```

- The **select from where** statement is evaluated fully before any of its results are inserted into the relation (otherwise queries like

insert into table1 select * from table1

would cause problems)





Modification of the Database – Updates

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

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

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

- The order is important
- Can be done better using the **case** statement (next slide)





Case Statement for Conditional Updates

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

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





Update of a View

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

```
create view loan_branch as  
    select loan_number, branch_name  
        from loan
```

- Add a new tuple to *branch_loan*

```
insert into branch_loan  
    values ('L-37', 'Perryridge')
```

This insertion must be represented by the insertion of the tuple

('L-37', 'Perryridge', *null*)

into the *loan* relation





Updates Through Views (Cont.)

- Some updates through views are impossible to translate into updates on the database relations
 - **create view v as**

```
select loan_number, branch_name, amount  
from loan  
where branch_name = 'Perryridge'
```

insert into v values ('L-99','Downtown', '23')
- Others cannot be translated uniquely
 - **insert into all_customer values ('Perryridge', 'John')**
 - ▶ Have to choose loan or account, and create a new loan/account number!
- Most SQL implementations allow updates only on simple views (without aggregates) defined on a single relation





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.

<i>Join types</i>
inner join
left outer join
right outer join
full outer join

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





Joined Relations – Datasets for Examples

- Relation *loan*
- Relation *borrower*

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

loan *borrower*

- 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*

loan_number	branch_name	amount	customer_name	loan_number
L-170	Downtown	3000	Jones	L-170
L-230	Redwood	4000	Smith	L-230

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

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





Joined Relations – Examples

- *loan natural inner join borrower*

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

- *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
```





End of Chapter 3

Database System Concepts, 5th Ed.

©Silberschatz, Korth and Sudarshan

See www.db-book.com for conditions on re-use





Figure 3.1: Database Schema

branch (branch_name, branch_city, assets)

customer (customer_name, customer_street, customer_city)

loan (loan_number, branch_name, amount)

borrower (customer_name, loan_number)

account (account_number, branch_name, balance)

depositor (customer_name, account_number)





Figure 3.3: Tuples inserted into *loan* and *borrower*

loan_number	branch_name	amount	customer_name	loan_number
L-11	Round Hill	900	Adams	L-16
L-14	Downtown	1500	Curry	L-93
L-15	Perryridge	1500	Hayes	L-15
L-16	Perryridge	1300	Jackson	L-14
L-17	Downtown	1000	Jones	L-17
L-23	Redwood	2000	Smith	L-11
L-93	Mianus	500	Smith	L-23
null	null	1900	Williams	L-17
<i>loan</i>			<i>borrower</i>	





Figure 3.4:

The *loan* and *borrower* relations

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





Chapter 4: Advanced SQL

Database System Concepts, 5th Ed.

©Silberschatz, Korth and Sudarshan

See www.db-book.com for conditions on re-use





Chapter 4: Advanced SQL

- SQL Data Types and Schemas
- Integrity Constraints
- Authorization
- Embedded SQL
- Dynamic SQL
- Functions and Procedural Constructs**
- Recursive Queries**
- Advanced SQL Features**





Built-in Data Types in SQL

- **date:** Dates, containing a (4 digit) year, month and date
 - Example: **date** '2005-7-27'
- **time:** Time of day, in hours, minutes and seconds.
 - Example: **time** '09:00:30' **time** '09:00:30.75'
- **timestamp:** date plus time of day
 - Example: **timestamp** '2005-7-27 09:00:30.75'
- **interval:** period of time
 - Example: **interval** '1' day
 - Subtracting a date/time/timestamp value from another gives an interval value
 - Interval values can be added to date/time/timestamp values





Build-in Data Types in SQL (Cont.)

- Can extract values of individual fields from date/time/timestamp
 - Example: **extract (year from r.starttime)**
- Can cast string types to date/time/timestamp
 - Example: **cast <string-valued-expression> as date**
 - Example: **cast <string-valued-expression> as time**





User-Defined Types

- **create type** construct in SQL creates user-defined type

```
create type Dollars as numeric (12,2) final
```

- **create domain** construct in SQL-92 creates user-defined domain types

```
create domain person_name char(20) not null
```

- Types and domains are similar. Domains can have constraints, such as **not null**, specified on them.





Domain Constraints

- **Domain constraints** are the most elementary form of integrity constraint. They test values inserted in the database, and test queries to ensure that the comparisons make sense.
- New domains can be created from existing data types
 - Example: `create domain Dollars numeric(12, 2)`
`create domain Pounds numeric(12,2)`
- We cannot assign or compare a value of type Dollars to a value of type Pounds.
 - However, we can convert type as below
`(cast r.A as Pounds)`
(Should also multiply by the dollar-to-pound conversion-rate)





Large-Object Types

- Large objects (photos, videos, CAD files, etc.) are stored as a *large object*.
 - **blob**: binary large object -- object is a large collection of uninterpreted binary data (whose interpretation is left to an application outside of the database system)
 - **clob**: character large object -- object is a large collection of character data
 - When a query returns a large object, a pointer is returned rather than the large object itself.





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.
 - A checking account must have a balance greater than \$10,000.00
 - A salary of a bank employee must be at least \$4.00 an hour
 - A customer must have a (non-null) phone number





Constraints on a Single Relation

- **not null**
- **primary key**
- **unique**
- **check** (P), where P is a predicate





Not Null Constraint

- Declare *branch_name* for *branch* is **not null**
branch_name **char(15) not null**

- Declare the domain *Dollars* to be **not null**

```
create domain Dollars numeric(12,2) not null
```





The Unique Constraint

- **unique** (A_1, A_2, \dots, A_m)
- The unique specification states that the attributes A_1, A_2, \dots, A_m form a candidate key.
- Candidate keys are permitted to be null (in contrast to primary keys).





The check clause

- **check (P)**, where P is a predicate

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

```
create table branch
  (branch_name    char(15),
   branch_city     char(30),
   assets          integer,
   primary key (branch_name),
   check (assets >= 0))
```





The check clause (Cont.)

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

```
create domain hourly_wage numeric(5,2)
constraint value_test check(value > = 4.00)
```
 - The domain has a constraint that ensures that the hourly_wage is greater than 4.00
 - The clause **constraint** *value_test* is optional; useful to indicate which constraint an update violated.





Referential Integrity

- Ensures that a value that appears in one relation for a given set of attributes also appears for a certain set of attributes in another relation.
 - Example: If “Perryridge” is a branch name appearing in one of the tuples in the *account* relation, then there exists a tuple in the *branch* relation for branch “Perryridge”.
- 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.





Referential Integrity in SQL – Example

```
create table customer
  (customer_name    char(20),
   customer_street  char(30),
   customer_city    char(30),
   primary key (customer_name))

create table branch
  (branch_name      char(15),
   branch_city       char(30),
   assets            numeric(12,2),
   primary key (branch_name))
```





Referential Integrity in SQL – Example (Cont.)

```
create table account
```

```
(account_number char(10),  
branch_name    char(15),  
balance        integer,  
primary key (account_number),  
foreign key (branch_name) references branch )
```

```
create table depositor
```

```
(customer_name  char(20),  
account_number  char(10),  
primary key (customer_name, account_number),  
foreign key (account_number) references account,  
foreign key (customer_name) references customer )
```





Assertions

- An **assertion** is a predicate expressing a condition that we wish the database always to satisfy.
- An assertion in SQL takes the form
 - create assertion <assertion-name> check <predicate>
- When an assertion is made, the system tests it for validity, and tests it again on every update that may violate the assertion
 - This testing may introduce a significant amount of overhead; hence assertions should be used with great care.
- Asserting
 - for all X , $P(X)$is achieved in a round-about fashion using
 - not exists X such that not $P(X)$





Assertion Example

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

```
create assertion balance_constraint check
```

```
(not exists (
```

```
    select *
```

```
    from loan
```

```
    where not exists (
```

```
        select *
```

```
        from borrower, depositor, account
```

```
        where loan.loan_number = borrower.loan_number
```

```
            and borrower.customer_name = depositor.customer_name
```

```
            and depositor.account_number = account.account_number
```

```
            and account.balance >= 1000)))
```





Assertion Example

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

create assertion *sum_constraint* check

```
(not exists (select *
             from branch
             where (select sum(amount )
                    from loan
                    where loan.branch_name =
                          branch.branch_name )
                   >= (select sum (amount )
                        from account
                        where loan.branch_name =
                              branch.branch_name )))
```





Authorization

Forms of authorization on parts of the database:

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

Forms of authorization to modify the database schema (covered in Chapter 8):

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





Authorization Specification in SQL

- The **grant** statement is used to confer authorization

```
grant <privilege list>
```

```
on <relation name or view name> to <user list>
```

- <user list> is:

- a user-id
- **public**, which allows all valid users the privilege granted
- A role (more on this in Chapter 8)

- Granting a privilege on a view does not imply granting any privileges on the underlying relations.
- The grantor of the privilege must already hold the privilege on the specified item (or be the database administrator).





Privileges in SQL

- **select**: allows read access to relation, or the ability to query using the view
 - Example: grant users U_1 , U_2 , and U_3 **select** authorization on the *branch* relation:

`grant select on branch to U_1 , U_2 , U_3`
- **insert**: the ability to insert tuples
- **update**: the ability to update using the SQL update statement
- **delete**: the ability to delete tuples.
- **all privileges**: used as a short form for all the allowable privileges
- more in Chapter 8





Revoking Authorization in SQL

- The **revoke** statement is used to revoke authorization.

revoke <privilege list>

on <relation name or view name> **from** <user list>

- Example:

revoke select on branch from U_1, U_2, U_3

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





Embedded SQL

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

```
EXEC SQL <embedded SQL statement> END_EXEC
```

Note: this varies by language (for example, the Java embedding uses

```
# SQL { .... }; )
```





Example Query

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

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

EXEC SQL

```
declare c cursor for
select depositor.customer_name, customer_city
from depositor, customer, account
where depositor.customer_name = customer.customer_name
      and depositor.account_number = account.account_number
      and account.balance > :amount
```

END_EXEC





Embedded SQL (Cont.)

- The **open** statement causes the query to be evaluated

```
EXEC SQL open c END_EXEC
```

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

```
EXEC SQL fetch c into :cn, :cc END_EXEC
```

Repeated calls to **fetch** get successive tuples in the query result

- A variable called SQLSTATE in the SQL communication area (SQLCA) gets set to '02000' to indicate no more data is available
- The **close** statement causes the database system to delete the temporary relation that holds the result of the query.

```
EXEC SQL close c END_EXEC
```

Note: above details vary with language. For example, the Java embedding defines Java iterators to step through result tuples.





Updates Through Cursors

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

```
declare c cursor for
  select *
    from account
   where branch_name = 'Perryridge'
  for update
```

- To update tuple at the current location of cursor *c*

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





Dynamic SQL

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

```
char * sqlprog = "update account
                  set balance = balance * 1.05
                  where account_number = ?"
EXEC SQL prepare dynprog from :sqlprog;
char account [10] = "A-101";
EXEC SQL execute dynprog using :account;
```

- The dynamic SQL program contains a ?, which is a place holder for a value that is provided when the SQL program is executed.





ODBC and JDBC

- API (application-program interface) for a program to interact with a database server
- Application makes calls to
 - Connect with the database server
 - Send SQL commands to the database server
 - Fetch tuples of result one-by-one into program variables
- ODBC (Open Database Connectivity) works with C, C++, C#, and Visual Basic
- JDBC (Java Database Connectivity) works with Java





ODBC

- Open DataBase Connectivity(ODBC) standard
 - standard for application program to communicate with a database server.
 - application program interface (API) to
 - ▶ open a connection with a database,
 - ▶ send queries and updates,
 - ▶ get back results.
- Applications such as GUI, spreadsheets, etc. can use ODBC





ODBC (Cont.)

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





ODBC Code

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

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





ODBC Code (Cont.)

- Program sends SQL commands to the database by using SQLExecDirect
- Result tuples are fetched using SQLFetch()
- SQLBindCol() binds C language variables to attributes of the query result
 - When a tuple is fetched, its attribute values are automatically stored in corresponding C variables.
 - Arguments to SQLBindCol()
 - ▶ ODBC stmt variable, attribute position in query result
 - ▶ The type conversion from SQL to C.
 - ▶ The address of the variable.
 - ▶ For variable-length types like character arrays,
 - The maximum length of the variable
 - Location to store actual length when a tuple is fetched.
 - Note: A negative value returned for the length field indicates null value
- Good programming requires checking results of every function call for errors; we have omitted most checks for brevity.





ODBC Code (Cont.)

- Main body of program

```
char branchname[80];
float balance;
int lenOut1, lenOut2;
HSTMT stmt;

SQLAllocStmt(conn, &stmt);
char * sqlquery = "select branch_name, sum (balance)
                    from account
                    group by branch_name";

error = SQLExecDirect(stmt, sqlquery, SQL_NTS);

if (error == SQL_SUCCESS) {
    SQLBindCol(stmt, 1, SQL_C_CHAR, branchname , 80,
&lenOut1);
    SQLBindCol(stmt, 2, SQL_C_FLOAT, &balance,      0 ,
&lenOut2);

    while (SQLFetch(stmt) >= SQL_SUCCESS) {
        printf (" %s %g\n", branchname, balance);
    }
}
SQLFreeStmt(stmt, SQL_DROP);
```





More ODBC Features

■ Prepared Statement

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

■ Metadata features

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

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

- Can turn off automatic commit on a connection
 - ▶ `SQLSetConnectOption(conn, SQL_AUTOCOMMIT, 0)`
- transactions must then be committed or rolled back explicitly by
 - ▶ `SQLTransact(conn, SQL_COMMIT)` or
 - ▶ `SQLTransact(conn, SQL_ROLLBACK)`





ODBC Conformance Levels

- Conformance levels specify subsets of the functionality defined by the standard.
 - Core
 - Level 1 requires support for metadata querying
 - Level 2 requires ability to send and retrieve arrays of parameter values and more detailed catalog information.
- SQL Call Level Interface (CLI) standard similar to ODBC interface, but with some minor differences.





JDBC

- JDBC is a Java API for communicating with database systems supporting SQL
- JDBC supports a variety of features for querying and updating data, and for retrieving query results
- JDBC also supports metadata retrieval, such as querying about relations present in the database and the names and types of relation attributes
- Model for communicating with the database:
 - Open a connection
 - Create a “statement” object
 - Execute queries using the Statement object to send queries and fetch results
 - Exception mechanism to handle errors





JDBC Code

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





JDBC Code (Cont.)

- Update to database

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

- Execute query and fetch and print results

```
ResultSet rset = stmt.executeQuery( "select branch_name,  
    avg(balance)  
        from account  
        group by branch_name");  
  
while (rset.next()) {  
    System.out.println(  
        rset.getString("branch_name") + " " + rset.getFloat(2));  
  
}
```





JDBC Code Details

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

```
int a = rs.getInt("a");
if (rs.wasNull()) System.out.println("Got null value");
```





Procedural Extensions and Stored Procedures

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





Functions and Procedures

- SQL:1999 supports functions and procedures
 - Functions/procedures can be written in SQL itself, or in an external programming language
 - Functions are particularly useful with specialized data types such as images and geometric objects
 - ▶ Example: functions to check if polygons overlap, or to compare images for similarity
 - Some database systems support **table-valued functions**, which can return a relation as a result
- SQL:1999 also supports a rich set of imperative constructs, including
 - Loops, if-then-else, assignment
- Many databases have proprietary procedural extensions to SQL that differ from SQL:1999





SQL Functions

- Define a function that, given the name of a customer, returns the count of the number of accounts owned by the customer.

```
create function account_count (customer_name varchar(20))
returns integer
begin
    declare a_count integer;
    select count (*) into a_count
    from depositor
    where depositor.customer_name = customer_name
    return a_count;
end
```

- Find the name and address of each customer that has more than one account.

```
select customer_name, customer_street, customer_city
from customer
where account_count (customer_name ) > 1
```





Table Functions

- SQL:2003 added functions that return a relation as a result
- Example: Return all accounts owned by a given customer

```
create function accounts_of(customer_name char(20))
    returns table (    account_number char(10),
                      branch_name char(15)
                      balance numeric(12,2))

return table
    (select account_number, branch_name, balance
     from account A
     where exists (
         select *
         from depositor D
         where D.customer_name = accounts_of.customer_name
             and D.account_number = A.account_number ))
```





Table Functions (cont'd)

■ Usage

```
select *  
from table (accounts_of('Smith'))
```





SQL Procedures

- The *author_count* function could instead be written as procedure:

```
create procedure account_count_proc (in title varchar(20),
                                     out a_count integer)
begin
    select count(author) into a_count
    from depositor
    where depositor.customer_name = account_count_proc.customer_name
end
```

- Procedures can be invoked either from an SQL procedure or from embedded SQL, using the **call** statement.

```
declare a_count integer;
call account_count_proc( 'Smith', a_count);
```

Procedures and functions can be invoked also from dynamic SQL

- SQL:1999 allows more than one function/procedure of the same name (called name **overloading**), as long as the number of arguments differ, or at least the types of the arguments differ





Procedural Constructs

- Compound statement: **begin ... end**,
 - May contain multiple SQL statements between **begin** and **end**.
 - Local variables can be declared within a compound statements
- **While** and **repeat** statements:

```
declare n integer default 0;
```

```
while n < 10 do
```

```
    set n = n + 1
```

```
end while
```

```
repeat
```

```
    set n = n - 1
```

```
until n = 0
```

```
end repeat
```





Procedural Constructs (Cont.)

■ For loop

- Permits iteration over all results of a query
- Example: find total of all balances at the Perryridge branch

```
declare n integer default 0;  
for r as  
    select balance from account  
        where branch_name = 'Perryridge'  
do  
    set n = n + r.balance  
end for
```





Procedural Constructs (cont.)

- Conditional statements (**if-then-else**)

E.g. To find sum of balances for each of three categories of accounts (with balance <1000, >=1000 and <5000, >= 5000)

```
if r.balance < 1000
  then set l = l + r.balance
elseif r.balance < 5000
  then set m = m + r.balance
else set h = h + r.balance
end if
```

- SQL:1999 also supports a **case** statement similar to C case statement

- Signaling of exception conditions, and declaring handlers for exceptions

```
declare out_of_stock condition
declare exit handler for out_of_stock
begin
  ...
  .. signal out-of-stock
end
```

- The handler here is **exit** -- causes enclosing **begin..end** to be exited
- Other actions possible on exception





External Language Functions/Procedures

- SQL:1999 permits the use of functions and procedures written in other languages such as C or C++
- Declaring external language procedures and functions

```
create procedure account_count_proc(in customer_name varchar(20),  
                                     out count integer)
```

language C

external name '/usr/avi/bin/account_count_proc'

```
create function account_count(customer_name varchar(20))
```

returns integer

language C

external name '/usr/avi/bin/author_count'





External Language Routines (Cont.)

- Benefits of external language functions/procedures:
 - more efficient for many operations, and more expressive power
- Drawbacks
 - Code to implement function may need to be loaded into database system and executed in the database system's address space
 - ▶ risk of accidental corruption of database structures
 - ▶ security risk, allowing users access to unauthorized data
 - There are alternatives, which give good security at the cost of potentially worse performance
 - Direct execution in the database system's space is used when efficiency is more important than security





Security with External Language Routines

- To deal with security problems
 - Use **sandbox** techniques
 - ▶ that is use a safe language like Java, which cannot be used to access/damage other parts of the database code
 - Or, run external language functions/procedures in a separate process, with no access to the database process' memory
 - ▶ Parameters and results communicated via inter-process communication
- Both have performance overheads
- Many database systems support both above approaches as well as direct executing in database system address space





Recursion in SQL

- SQL:1999 permits recursive view definition
- Example: find all employee-manager pairs, where the employee reports to the manager directly or indirectly (that is manager's manager, manager's manager's manager, etc.)

```
with recursive empl (employee_name, manager_name ) as (
    select employee_name, manager_name
    from   manager
  union
    select manager.employee_name, empl.manager_name
    from   manager, empl
    where  manager.manager_name = empl.employee_name)
select *
from   empl
```

This example view, `empl`, is called the *transitive closure* of the `manager` relation





The Power of Recursion

- Recursive views make it possible to write queries, such as transitive closure queries, that cannot be written without recursion or iteration.
 - Intuition: Without recursion, a non-recursive non-iterative program can perform only a fixed number of joins of *manager* with itself
 - ▶ This can give only a fixed number of levels of managers
 - ▶ Given a program we can construct a database with a greater number of levels of managers on which the program will not work
- Computing transitive closure
 - The next slide shows a *manager* relation
 - Each step of the iterative process constructs an extended version of *empl* from its recursive definition.
 - The final result is called the *fixed point* of the recursive view definition.
- Recursive views are required to be *monotonic*. That is, if we add tuples to *manger* the view contains all of the tuples it contained before, plus possibly more





Example of Fixed-Point Computation

<i>employee_name</i>	<i>manager_name</i>
Alon	Barinsky
Barinsky	Estovar
Corbin	Duarte
Duarte	Jones
Estovar	Jones
Jones	Klinger
Rensal	Klinger

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





Advanced SQL Features**

- Create a table with the same schema as an existing table:
`create table temp_account like account`
- SQL:2003 allows subqueries to occur *anywhere* a value is required provided the subquery returns only one value. This applies to updates as well
- SQL:2003 allows subqueries in the **from** clause to access attributes of other relations in the **from** clause using the **lateral** construct:

```
select C.customer_name, num_accounts
  from customer C,
       lateral (select count(*)
                  from account A
                 where A.customer_name = C.customer_name )
               as this_customer (num_accounts )
```





Advanced SQL Features (cont'd)

- Merge construct allows batch processing of updates.
- Example: relation *funds_received* (*account_number*, *amount*) has batch of deposits to be added to the proper account in the *account* relation

```
merge into account as A  
using (select *  
        from funds_received as F)  
on (A.account_number = F.account_number)  
when matched then  
    update set balance = balance + F.amount
```





End of Chapter

Database System Concepts, 5th Ed.

©Silberschatz, Korth and Sudarshan

See www.db-book.com for conditions on re-use





Chapter 7: Relational Database Design

Database System Concepts, 5th Ed.

©Silberschatz, Korth and Sudarshan

See www.db-book.com for conditions on re-use





Chapter 7: Relational Database Design

- Features of Good Relational Design
- Atomic Domains and First Normal Form
- Decomposition Using Functional Dependencies
- Functional Dependency Theory
- Algorithms for Functional Dependencies
- Decomposition Using Multivalued Dependencies
- More Normal Form
- Database-Design Process
- Modeling Temporal Data





The Banking Schema

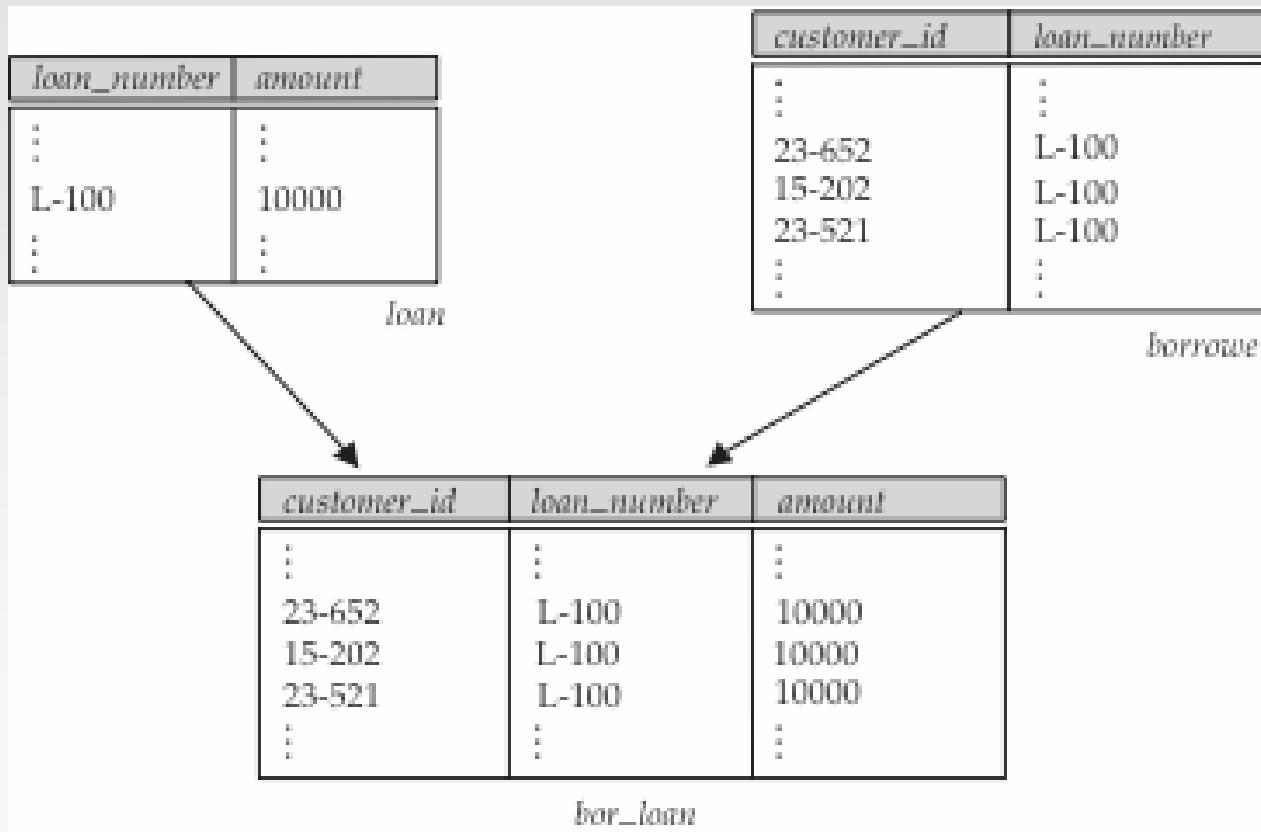
- $\text{branch} = (\underline{\text{branch_name}}, \underline{\text{branch_city}}, \text{assets})$
- $\text{customer} = (\underline{\text{customer_id}}, \underline{\text{customer_name}}, \underline{\text{customer_street}}, \underline{\text{customer_city}})$
- $\text{loan} = (\underline{\text{loan_number}}, \text{amount})$
- $\text{account} = (\underline{\text{account_number}}, \text{balance})$
- $\text{employee} = (\underline{\text{employee_id}}, \underline{\text{employee_name}}, \underline{\text{telephone_number}}, \underline{\text{start_date}})$
- $\text{dependent_name} = (\underline{\text{employee_id}}, \underline{\text{dname}})$
- $\text{account_branch} = (\underline{\text{account_number}}, \underline{\text{branch_name}})$
- $\text{loan_branch} = (\underline{\text{loan_number}}, \underline{\text{branch_name}})$
- $\text{borrower} = (\underline{\text{customer_id}}, \underline{\text{loan_number}})$
- $\text{depositor} = (\underline{\text{customer_id}}, \underline{\text{account_number}})$
- $\text{cust_banker} = (\underline{\text{customer_id}}, \underline{\text{employee_id}}, \text{type})$
- $\text{works_for} = (\underline{\text{worker_employee_id}}, \underline{\text{manager_employee_id}})$
- $\text{payment} = (\underline{\text{loan_number}}, \underline{\text{payment_number}}, \underline{\text{payment_date}}, \text{payment_amount})$
- $\text{savings_account} = (\underline{\text{account_number}}, \text{interest_rate})$
- $\text{checking_account} = (\underline{\text{account_number}}, \text{overdraft_amount})$





Combine Schemas?

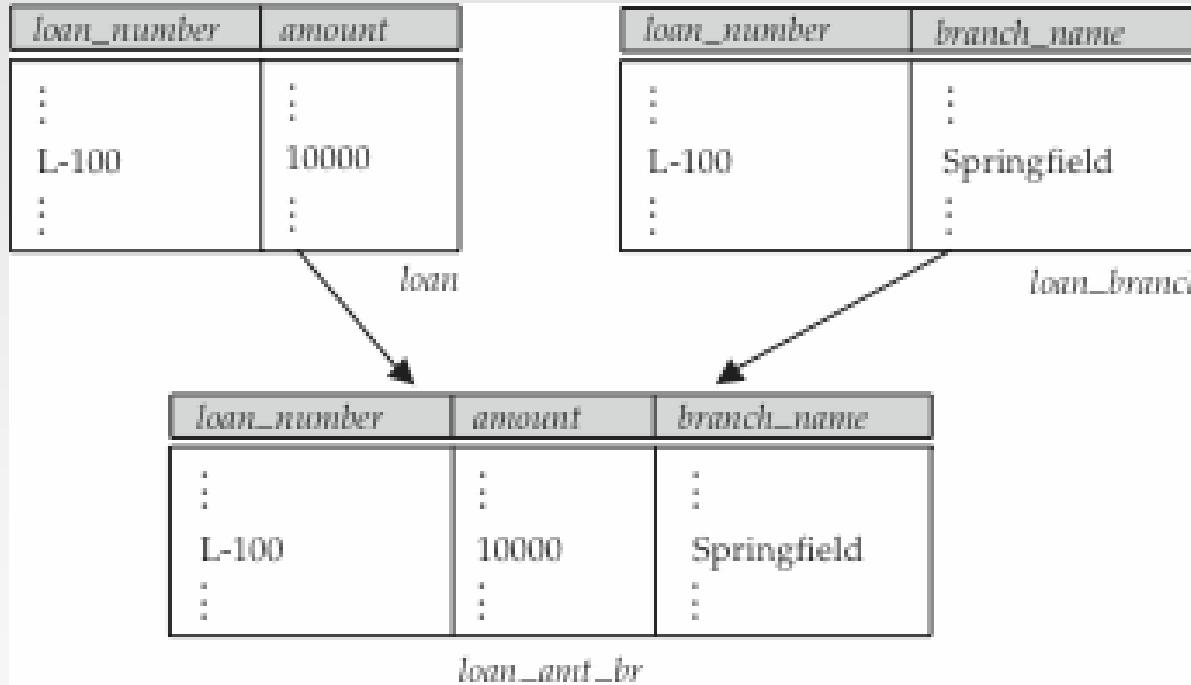
- Suppose we combine *borrower* and *loan* to get
 $bor_loan = (customer_id, loan_number, amount)$
- Result is possible repetition of information (L-100 in example below)





A Combined Schema Without Repetition

- Consider combining *loan_branch* and *loan*
 $\text{loan_amt_br} = (\text{loan_number}, \text{amount}, \text{branch_name})$
- No repetition (as suggested by example below)





What About Smaller Schemas?

- Suppose we had started with *bor_loan*. How would we know to split up (**decompose**) it into *borrower* and *loan*?
- Write a rule “if there were a schema (*loan_number, amount*), then *loan_number* would be a candidate key”
- Denote as a **functional dependency**:

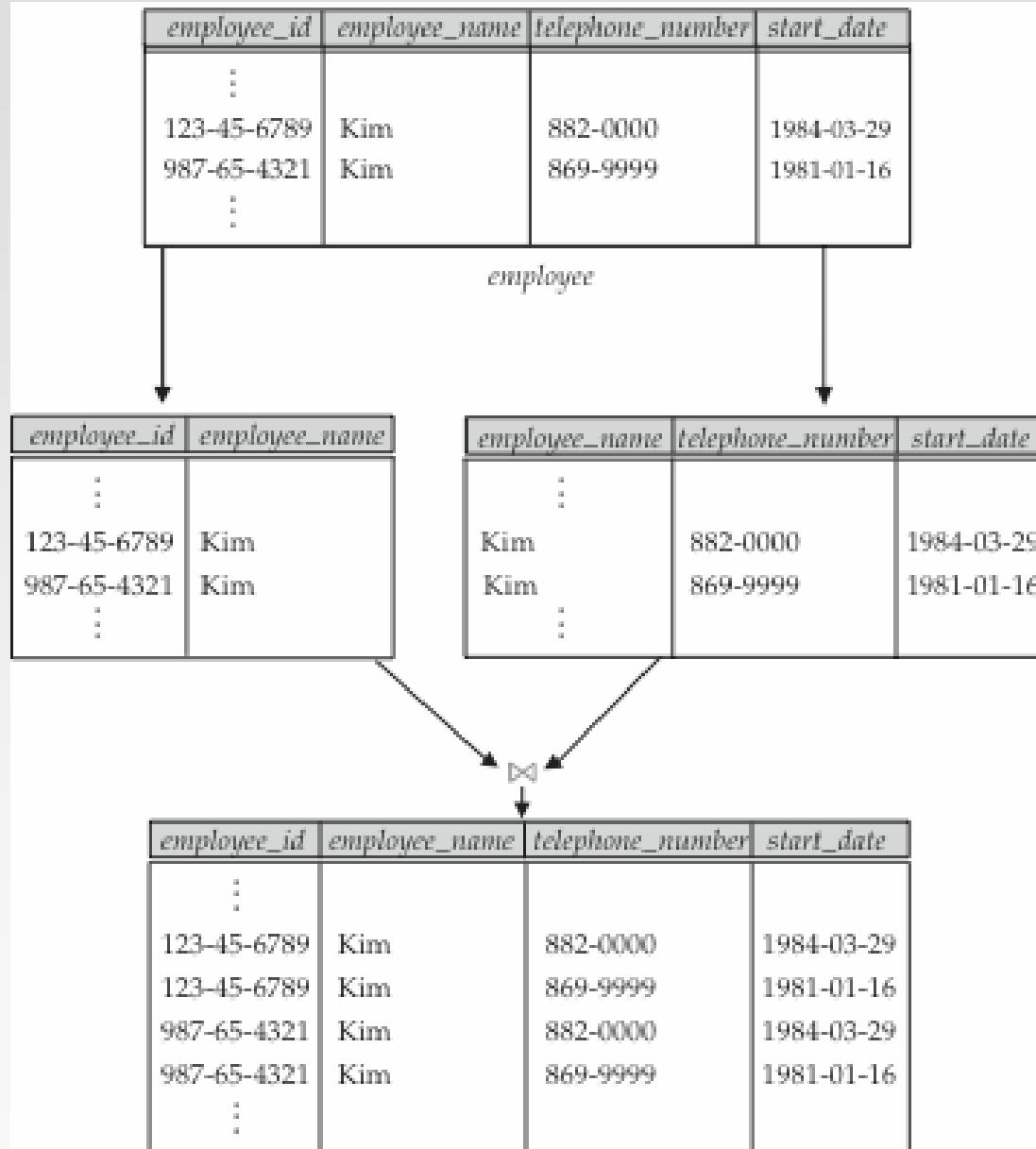
$$\text{loan_number} \rightarrow \text{amount}$$

- In *bor_loan*, because *loan_number* is not a candidate key, the amount of a loan may have to be repeated. This indicates the need to decompose *bor_loan*.
- Not all decompositions are good. Suppose we decompose *employee* into
 - employee1* = (*employee_id, employee_name*)
 - employee2* = (*employee_name, telephone_number, start_date*)
- The next slide shows how we lose information -- we cannot reconstruct the original *employee* relation -- and so, this is a lossy decomposition.





A Lossy Decomposition





First Normal Form

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





First Normal Form (Cont'd)

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





Goal — Devise a Theory for the Following

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





Functional Dependencies

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





Functional Dependencies (Cont.)

- 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:

bor_loan = (customer_id, loan_number, amount).

We expect this functional dependency to hold:

loan_number \rightarrow amount

but would not expect the following to hold:

amount \rightarrow customer_name





Use of Functional Dependencies

- We use functional dependencies to:
 - test relations to see if they are legal under a given set of functional dependencies.
 - ▶ If a relation r is legal under a set F of functional dependencies, we say that r **satisfies** F .
 - specify constraints on the set of legal relations
 - ▶ We say that F **holds on** R if all legal relations on R satisfy the set of functional dependencies F .
- Note: A specific instance of a relation schema may satisfy a functional dependency even if the functional dependency does not hold on all legal instances.
 - For example, a specific instance of *loan* may, by chance, satisfy $amount \rightarrow customer_name$.





Functional Dependencies (Cont.)

- A functional dependency is **trivial** if it is satisfied by all instances of a relation
 - Example:
 - ▶ $\text{customer_name}, \text{loan_number} \rightarrow \text{customer_name}$
 - ▶ $\text{customer_name} \rightarrow \text{customer_name}$
 - In general, $\alpha \rightarrow \beta$ is trivial if $\beta \subseteq \alpha$





Closure of a Set of Functional Dependencies

- Given a set F of functional dependencies, there are certain other functional dependencies that are logically implied by F .
 - For example: If $A \rightarrow B$ and $B \rightarrow C$, then we can infer that $A \rightarrow C$
- The set of all functional dependencies logically implied by F is the *closure* of F .
- We denote the *closure* of F by F^+ .
- F^+ is a superset of F .





Boyce-Codd Normal Form

A relation schema R is in BCNF with respect to a set F of functional dependencies if for all functional dependencies in F^+ of the form

$$\alpha \rightarrow \beta$$

where $\alpha \subseteq R$ and $\beta \subseteq R$, at least one of the following holds:

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

Example schema *not* in BCNF:

bor_loan = (*customer_id*, *loan_number*, *amount*)

because $\text{loan_number} \rightarrow \text{amount}$ holds on *bor_loan* but *loan_number* is not a superkey





Decomposing a Schema into BCNF

- Suppose we have a schema R and a non-trivial dependency $\alpha \rightarrow \beta$ causes a violation of BCNF.

We decompose R into:

- $(\alpha \cup \beta)$
- $(R - (\beta - \alpha))$

- In our example,

- $\alpha = \text{loan_number}$
- $\beta = \text{amount}$

and bor_loan is replaced by

- $(\alpha \cup \beta) = (\text{loan_number}, \text{amount})$
- $(R - (\beta - \alpha)) = (\text{customer_id}, \text{loan_number})$





BCNF and Dependency Preservation

- Constraints, including functional dependencies, are costly to check in practice unless they pertain to only one relation
- If it is sufficient to test only those dependencies on each individual relation of a decomposition in order to ensure that *all* functional dependencies hold, then that decomposition is *dependency preserving*.
- Because it is not always possible to achieve both BCNF and dependency preservation, we consider a weaker normal form, known as *third normal form*.





Third Normal Form

- A relation schema R is in third normal form (3NF) if for all:

$$\alpha \rightarrow \beta \text{ in } F^+$$

at least one of the following holds:

- $\alpha \rightarrow \beta$ is trivial (i.e., $\beta \in \alpha$)
- α is a superkey for R
- Each attribute A in $\beta - \alpha$ is contained in a candidate key for R .

(**NOTE**: each attribute may be in a different candidate key)

- If a relation is in BCNF it is in 3NF (since in BCNF one of the first two conditions above must hold).
- Third condition is a minimal relaxation of BCNF to ensure dependency preservation (will see why later).





Goals of Normalization

- Let R be a relation scheme with a set F of functional dependencies.
- Decide whether a relation scheme R is in “good” form.
- In the case that a relation scheme R is not in “good” form, decompose it into a set of relation scheme $\{R_1, R_2, \dots, R_n\}$ such that
 - each relation scheme is in good form
 - the decomposition is a lossless-join decomposition
 - Preferably, the decomposition should be dependency preserving.





How good is BCNF?

- 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).





How good is BCNF? (Cont.)

course	teacher	book
database	Avi	DB Concepts
database	Avi	Ullman
database	Hank	DB Concepts
database	Hank	Ullman
database	Sudarshan	DB Concepts
database	Sudarshan	Ullman
operating systems	Avi	OS Concepts
operating systems	Avi	Stallings
operating systems	Pete	OS Concepts
operating systems	Pete	Stallings

classes

- There are no non-trivial functional dependencies and therefore the relation is in BCNF
- Insertion anomalies – i.e., if Marilyn is a new teacher that can teach database, two tuples need to be inserted

(database, Marilyn, DB Concepts)
(database, Marilyn, Ullman)





How good is BCNF? (Cont.)

- Therefore, it is better to decompose *classes* into:

course	teacher
database	Avi
database	Hank
database	Sudarshan
operating systems	Avi
operating systems	Jim

teaches

course	book
database	DB Concepts
database	Ullman
operating systems	OS Concepts
operating systems	Shaw

text

This suggests the need for higher normal forms, such as Fourth Normal Form (4NF), which we shall see later.





Functional-Dependency Theory

- We now consider the formal theory that tells us which functional dependencies are implied logically by a given set of functional dependencies.
- We then develop algorithms to generate lossless decompositions into BCNF and 3NF
- We then develop algorithms to test if a decomposition is dependency-preserving





Closure of a Set of Functional Dependencies

- Given a set F of functional dependencies, there are certain other functional dependencies that are logically implied by F .
 - For example: If $A \rightarrow B$ and $B \rightarrow C$, then we can infer that $A \rightarrow C$
- The set of all functional dependencies logically implied by F is the *closure* of F .
- We denote the *closure* of F by F^+ .
- We can find all of F^+ by applying Armstrong's Axioms:
 - if $\beta \subseteq \alpha$, then $\alpha \rightarrow \beta$ **(reflexivity)**
 - if $\alpha \rightarrow \beta$, then $\gamma \alpha \rightarrow \gamma \beta$ **(augmentation)**
 - if $\alpha \rightarrow \beta$, and $\beta \rightarrow \gamma$, then $\alpha \rightarrow \gamma$ **(transitivity)**
- These rules are
 - **sound** (generate only functional dependencies that actually hold) and
 - **complete** (generate all functional dependencies that hold).





Example

- $R = (A, B, C, G, H, I)$

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

- some members of F^+

- $A \rightarrow H$

- ▶ by transitivity from $A \rightarrow B$ and $B \rightarrow H$

- $AG \rightarrow I$

- ▶ by augmenting $A \rightarrow C$ with G , to get $AG \rightarrow CG$ and then transitivity with $CG \rightarrow I$

- $CG \rightarrow HI$

- ▶ by augmenting $CG \rightarrow I$ to infer $CG \rightarrow CGI$, and augmenting of $CG \rightarrow H$ to infer $CGI \rightarrow HI$, and then transitivity





Procedure for Computing F^+

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

$$F^+ = F$$

repeat

for each functional dependency f in F^+

 apply reflexivity and augmentation rules on f

 add the resulting functional dependencies to F^+

for each pair of functional dependencies f_1 and f_2 in F^+

if f_1 and f_2 can be combined using transitivity

then add the resulting functional dependency to F^+

until F^+ does not change any further

NOTE: We shall see an alternative procedure for this task later





Closure of Functional Dependencies (Cont.)

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

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





Closure of Attribute Sets

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

```
result :=  $\alpha$ ;  
while (changes to result) do  
  for each  $\beta \rightarrow \gamma$  in  $F$  do  
    begin  
      if  $\beta \subseteq result$  then result := result  $\cup$   $\gamma$   
    end
```





Example of Attribute Set Closure

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





Uses of Attribute Closure

There are several uses of the attribute closure algorithm:

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





Canonical Cover

- Sets of functional dependencies may have redundant dependencies that can be inferred from the others
 - For example: $A \rightarrow C$ is redundant in: $\{A \rightarrow B, B \rightarrow C\}$
 - Parts of a functional dependency may be redundant
 - ▶ E.g.: on RHS: $\{A \rightarrow B, B \rightarrow C, A \rightarrow CD\}$ can be simplified to
$$\{A \rightarrow B, B \rightarrow C, A \rightarrow D\}$$
 - ▶ E.g.: on LHS: $\{A \rightarrow B, B \rightarrow C, AC \rightarrow D\}$ can be simplified to
$$\{A \rightarrow B, B \rightarrow C, A \rightarrow D\}$$
- Intuitively, a canonical cover of F is a “minimal” set of functional dependencies equivalent to F , having no redundant dependencies or redundant parts of dependencies





Extraneous Attributes

- Consider a set F of functional dependencies and the functional dependency $\alpha \rightarrow \beta$ in F .
 - Attribute A is **extraneous** in α if $A \in \alpha$ and F logically implies $(F - \{\alpha \rightarrow \beta\}) \cup \{(\alpha - A) \rightarrow \beta\}$.
 - Attribute A is **extraneous** in β if $A \in \beta$ and the set of functional dependencies $(F - \{\alpha \rightarrow \beta\}) \cup \{\alpha \rightarrow (\beta - A)\}$ logically implies F .
- Note: implication in the opposite direction is trivial in each of the cases above, since a “stronger” functional dependency always implies a weaker one
- Example: Given $F = \{A \rightarrow C, AB \rightarrow C\}$
 - B is extraneous in $AB \rightarrow C$ because $\{A \rightarrow C, AB \rightarrow C\}$ logically implies $A \rightarrow C$ (i.e. the result of dropping B from $AB \rightarrow C$).
- Example: Given $F = \{A \rightarrow C, AB \rightarrow CD\}$
 - C is extraneous in $AB \rightarrow CD$ since $AB \rightarrow C$ can be inferred even after deleting C





Testing if an Attribute is Extraneous

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





Canonical Cover

- A *canonical cover* for F is a set of dependencies F_c such that
 - F logically implies all dependencies in F_c , and
 - F_c logically implies all dependencies in F , and
 - No functional dependency in F_c contains an extraneous attribute, and
 - Each left side of functional dependency in F_c is unique.
- To compute a canonical cover for F :
repeat
 - Use the union rule to replace any dependencies in F
 $\alpha_1 \rightarrow \beta_1$ and $\alpha_1 \rightarrow \beta_2$ with $\alpha_1 \rightarrow \beta_1 \beta_2$
 - Find a functional dependency $\alpha \rightarrow \beta$ with an
 extraneous attribute either in α or in β
 - If an extraneous attribute is found, delete it from $\alpha \rightarrow \beta$**until** F does not change
- Note: Union rule may become applicable after some extraneous attributes have been deleted, so it has to be re-applied





Computing a Canonical Cover

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





Lossless-join Decomposition

- For the case of $R = (R_1, R_2)$, we require that for all possible relations r on schema R

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

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





Example

- $R = (A, B, C)$
 $F = \{A \rightarrow B, B \rightarrow C\}$
 - Can be decomposed in two different ways

- $R_1 = (A, B), R_2 = (B, C)$
 - Lossless-join decomposition:

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

- $R_1 = (A, B), R_2 = (A, C)$
 - Lossless-join decomposition:

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

- Not dependency preserving
(cannot check $B \rightarrow C$ without computing $R_1 \bowtie R_2$)





Dependency Preservation

- Let F_i be the set of dependencies F^+ that include only attributes in R_i
 - ▶ A decomposition is **dependency preserving**, if
$$(F_1 \cup F_2 \cup \dots \cup F_n)^+ = F^+$$
 - ▶ If it is not, then checking updates for violation of functional dependencies may require computing joins, which is expensive.





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 test (with attribute closure done with respect to 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 $(F_1 \cup F_2 \cup \dots \cup F_n)^+$





Example

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





Testing for BCNF

- To check if a non-trivial dependency $\alpha \rightarrow \beta$ causes a violation of BCNF
 1. compute α^+ (the attribute closure of α), and
 2. verify that it includes all attributes of R , that is, it is a superkey of R .
- **Simplified test:** To check if a relation schema R is in BCNF, it suffices to check only the dependencies in the given set F for violation of BCNF, rather than checking all dependencies in F^+ .
 - If none of the dependencies in F causes a violation of BCNF, then none of the dependencies in F^+ will cause a violation of BCNF either.
- However, using only F is **incorrect** when testing a relation in a decomposition of R
 - Consider $R = (A, B, C, D, E)$, with $F = \{ A \rightarrow B, BC \rightarrow D \}$
 - ▶ Decompose R into $R_1 = (A, B)$ and $R_2 = (A, C, D, E)$
 - ▶ Neither of the dependencies in F contain only attributes from (A, C, D, E) so we might be misled into thinking R_2 satisfies BCNF.
 - ▶ In fact, dependency $AC \rightarrow D$ in F^+ shows R_2 is not in BCNF.





Testing Decomposition for BCNF

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





BCNF Decomposition Algorithm

```
result := {R };
done := false;
compute  $F^+$ ;
while (not done) do
  if (there is a schema  $R_i$  in result that is not in BCNF)
    then begin
      let  $\alpha \rightarrow \beta$  be a nontrivial functional dependency that holds on  $R_i$ 
      such that  $\alpha \rightarrow R_i$  is not in  $F^+$ ,
      and  $\alpha \cap \beta = \emptyset$ ;
      result := (result -  $R_i$ )  $\cup$  ( $R_i - \beta$ )  $\cup$  ( $\alpha, \beta$ );
    end
  else done := true;
```

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





Example of BCNF Decomposition

- $R = (A, B, C)$
 $F = \{A \rightarrow B$
 $\quad B \rightarrow C\}$
Key = {A}
- R is not in BCNF ($B \rightarrow C$ but B is not superkey)
- Decomposition
 - $R_1 = (B, C)$
 - $R_2 = (A, B)$





Example of BCNF Decomposition

- Original relation R and functional dependency F

$$R = (\text{branch_name}, \text{branch_city}, \text{assets}, \\ \text{customer_name}, \text{loan_number}, \text{amount})$$
$$F = \{\text{branch_name} \rightarrow \text{assets} \text{ branch_city} \\ \text{loan_number} \rightarrow \text{amount} \text{ branch_name}\}$$

Key = {loan_number, customer_name}

- Decomposition

- $R_1 = (\text{branch_name}, \text{branch_city}, \text{assets})$
- $R_2 = (\text{branch_name}, \text{customer_name}, \text{loan_number}, \text{amount})$
- $R_3 = (\text{branch_name}, \text{loan_number}, \text{amount})$
- $R_4 = (\text{customer_name}, \text{loan_number})$

- Final decomposition

$$R_1, R_3, R_4$$




BCNF and Dependency Preservation

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

- $R = (J, K, L)$

$$F = \{JK \rightarrow L$$

$$L \rightarrow K\}$$

Two candidate keys = JK and JL

- R is not in BCNF

- Any decomposition of R will fail to preserve

$$JK \rightarrow L$$

This implies that testing for $JK \rightarrow L$ requires a join





Third Normal Form: Motivation

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





3NF Example

■ Relation R:

- $R = (J, K, L)$
 $F = \{JK \rightarrow L, L \rightarrow K\}$
- Two candidate keys: JK and JL
- R is in 3NF

$JK \rightarrow L$

JK is a superkey

$L \rightarrow K$

K is contained in a candidate key





Redundancy in 3NF

- There is some redundancy in this schema
- Example of problems due to redundancy in 3NF
 - $R = (J, K, L)$
 $F = \{JK \rightarrow L, L \rightarrow K\}$

J	L	K
j_1	l_1	k_1
j_2	l_1	k_1
j_3	l_1	k_1
<i>null</i>	l_2	k_2

- repetition of information (e.g., the relationship l_1, k_1)
- need to use null values (e.g., to represent the relationship l_2, k_2 where there is no corresponding value for J).





Testing for 3NF

- Optimization: Need to check only FDs in F , need not check all FDs in F^+ .
- Use attribute closure to check for each dependency $\alpha \rightarrow \beta$, if α is a superkey.
- If α is not a superkey, we have to verify if each attribute in β is contained in a candidate key of R
 - this test is rather more expensive, since it involves finding candidate keys
 - testing for 3NF has been shown to be NP-hard
 - Interestingly, decomposition into third normal form (described shortly) can be done in polynomial time





3NF Decomposition Algorithm

Let F_c be a canonical cover for F ;

$i := 0$;

for each functional dependency $\alpha \rightarrow \beta$ in F_c **do**

if none of the schemas R_j , $1 \leq j \leq i$ contains $\alpha \beta$

then begin

$i := i + 1$;

$R_i := \alpha \beta$

end

if none of the schemas R_j , $1 \leq j \leq i$ contains a candidate key for R

then begin

$i := i + 1$;

$R_i :=$ any candidate key for R ;

end

return (R_1, R_2, \dots, R_i)





3NF Decomposition Algorithm (Cont.)

- Above algorithm ensures:
 - each relation schema R_i is in 3NF
 - decomposition is dependency preserving and lossless-join
 - Proof of correctness is at end of this presentation ([click here](#))





3NF Decomposition: An Example

- Relation schema:

$\text{cust_banker_branch} = (\underline{\text{customer_id}}, \underline{\text{employee_id}}, \text{branch_name}, \text{type})$

- The functional dependencies for this relation schema are:

1. $\text{customer_id}, \text{employee_id} \rightarrow \text{branch_name}, \text{type}$
2. $\text{employee_id} \rightarrow \text{branch_name}$
3. $\text{customer_id}, \text{branch_name} \rightarrow \text{employee_id}$

- We first compute a canonical cover

- branch_name is extraneous in the r.h.s. of the 1st dependency
- No other attribute is extraneous, so we get $F_C =$

$\text{customer_id}, \text{employee_id} \rightarrow \text{type}$

$\text{employee_id} \rightarrow \text{branch_name}$

$\text{customer_id}, \text{branch_name} \rightarrow \text{employee_id}$





3NF Decomposition Example (Cont.)

- The **for** loop generates following 3NF schema:

$(customer_id, employee_id, type)$

$(\underline{employee_id}, branch_name)$

$(customer_id, branch_name, employee_id)$

- Observe that $(customer_id, employee_id, type)$ contains a candidate key of the original schema, so no further relation schema needs be added
- If the FDs were considered in a different order, with the 2nd one considered after the 3rd,
 $(\underline{employee_id}, branch_name)$
would not be included in the decomposition because it is a subset of
 $(customer_id, branch_name, employee_id)$
- Minor extension of the 3NF decomposition algorithm: at end of for loop, detect and delete schemas, such as $(\underline{employee_id}, branch_name)$, which are subsets of other schemas
 - result will not depend on the order in which FDs are considered
- The resultant simplified 3NF schema is:

$(customer_id, employee_id, type)$

$(customer_id, branch_name, employee_id)$





Comparison of BCNF and 3NF

- It is always possible to decompose a relation into a set of relations that are in 3NF such that:
 - the decomposition is lossless
 - the dependencies are preserved
- It is always possible to decompose a relation into a set of relations that are in BCNF such that:
 - the decomposition is lossless
 - it may not be possible to preserve dependencies.





Design Goals

- Goal for a relational database design is:
 - BCNF.
 - Lossless join.
 - Dependency preservation.
- If we cannot achieve this, we accept one of
 - Lack of dependency preservation
 - Redundancy due to use of 3NF
- Interestingly, SQL does not provide a direct way of specifying functional dependencies other than superkeys.
Can specify FDs using assertions, but they are expensive to test
- Even if we had a dependency preserving decomposition, using SQL we would not be able to efficiently test a functional dependency whose left hand side is not a key.





Multivalued Dependencies (MVDs)

- Let R be a relation schema and let $\alpha \subseteq R$ and $\beta \subseteq R$. The *multivalued dependency*

$$\alpha \rightarrow\!\!\!\rightarrow \beta$$

holds on R if in any legal relation $r(R)$, for all pairs of 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:

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





MVD (Cont.)

- Tabular representation of $\alpha \rightarrow\!\!\!\rightarrow \beta$

	α	β	$R - \alpha - \beta$
t_1	$a_1 \dots a_i$	$a_{i+1} \dots a_j$	$a_{j+1} \dots a_n$
t_2	$a_1 \dots a_i$	$b_{i+1} \dots b_j$	$b_{j+1} \dots b_n$
t_3	$a_1 \dots a_i$	$a_{i+1} \dots a_j$	$b_{j+1} \dots b_n$
t_4	$a_1 \dots a_i$	$b_{i+1} \dots b_j$	$a_{j+1} \dots a_n$





Example

- Let R be a relation schema with a set of attributes that are partitioned into 3 nonempty subsets.

Y, Z, W

- We say that $Y \rightarrow\!\!\rightarrow Z$ (Y multidetermines Z) if and only if for all possible relations $r(R)$

$$\langle y_1, z_1, w_1 \rangle \in r \text{ and } \langle y_1, z_2, w_2 \rangle \in r$$

then

$$\langle y_1, z_1, w_2 \rangle \in r \text{ and } \langle y_1, z_2, w_1 \rangle \in r$$

- Note that since the behavior of Z and W are identical it follows that
 $Y \rightarrow\!\!\rightarrow Z$ if $Y \rightarrow\!\!\rightarrow W$





Example (Cont.)

- In our example:

course $\rightarrow\!\!\!\rightarrow$ *teacher*

course $\rightarrow\!\!\!\rightarrow$ *book*

- The above formal definition is supposed to formalize the notion that given a particular value of *Y* (*course*) it has associated with it a set of values of *Z* (*teacher*) and a set of values of *W* (*book*), and these two sets are in some sense independent of each other.
- Note:
 - If $Y \rightarrow Z$ then $Y \rightarrow\!\!\!\rightarrow Z$
 - Indeed we have (in above notation) $Z_1 = Z_2$
The claim follows.





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 multivalued dependency, we can derive the following rule:
 - If $\alpha \rightarrow \beta$, then $\alpha \rightarrow\rightarrow \beta$
- That is, every functional dependency is also a multivalued dependency
- The **closure** D^+ of D is the set of all functional and multivalued dependencies logically implied by D .
 - We can compute D^+ from D , using the formal definitions of functional dependencies and multivalued dependencies.
 - We can manage with such reasoning for very simple multivalued 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 (see Appendix C).





Fourth Normal Form

- A relation schema R is in 4NF with respect to a set D of functional and multivalued dependencies if for all multivalued dependencies in D^+ of the form $\alpha \twoheadrightarrow \beta$, where $\alpha \subseteq R$ and $\beta \subseteq R$, at least one of the following hold:
 - $\alpha \twoheadrightarrow \beta$ is trivial (i.e., $\beta \subseteq \alpha$ or $\alpha \cup \beta = R$)
 - α is a superkey for schema R
- If a relation is in 4NF it is in BCNF





Restriction of Multivalued Dependencies

- The restriction of D to R_i is the set D_i consisting of
 - All functional dependencies in D^+ that include only attributes of R_i
 - All multivalued dependencies of the form
$$\alpha \twoheadrightarrow (\beta \cap R_i)$$
where $\alpha \subseteq R_i$ and $\alpha \twoheadrightarrow \beta$ is in D^+





4NF Decomposition Algorithm

```
result := {R};  
done := false;  
compute D+;  
Let Di denote the restriction of D+ to Ri  
while (not done)  
  if (there is a schema Ri in result that is not in 4NF) then  
    begin  
      let α →→ β be a nontrivial multivalued dependency that holds  
      on Ri such that α → Ri is not in Di, and α ∩ β = φ;  
      result := (result - Ri) ∪ (Ri - β) ∪ (α, β);  
    end  
  else done := true;
```

Note: each R_i is in 4NF, and decomposition is lossless-join





Example

- $R = (A, B, C, G, H, I)$

$$\begin{aligned} F = \{ & A \rightarrow\!\!\!\rightarrow B \\ & B \rightarrow\!\!\!\rightarrow HI \\ & CG \rightarrow\!\!\!\rightarrow HI \} \end{aligned}$$

- R is not in 4NF since $A \rightarrow\!\!\!\rightarrow B$ and A is not a superkey for R

- Decomposition

- a) $R_1 = (A, B)$ $(R_1$ is in 4NF)

- b) $R_2 = (A, C, G, H, I)$ $(R_2$ is not in 4NF)

- c) $R_3 = (C, G, H)$ $(R_3$ is in 4NF)

- d) $R_4 = (A, C, G, I)$ $(R_4$ is not in 4NF)

- Since $A \rightarrow\!\!\!\rightarrow B$ and $B \rightarrow\!\!\!\rightarrow HI$, $A \rightarrow\!\!\!\rightarrow HI$, $A \rightarrow\!\!\!\rightarrow I$

- e) $R_5 = (A, I)$ $(R_5$ is in 4NF)

- f) $R_6 = (A, C, G)$ $(R_6$ is in 4NF)





Further Normal Forms

- **Join dependencies** generalize multivalued dependencies
 - lead to **project-join normal form (PJNF)** (also called **fifth normal form**)
- A class of even more general constraints, leads to a normal form called **domain-key normal form**.
- Problem with these generalized constraints: are hard to reason with, and no set of sound and complete set of inference rules exists.
- Hence rarely used





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.





ER Model and Normalization

- When an E-R diagram is carefully designed, identifying all entities correctly, the tables generated from the E-R diagram should not need further normalization.
- However, in a real (imperfect) design, there can be functional dependencies from non-key attributes of an entity to other attributes of the entity
 - Example: an *employee* entity with attributes *department_number* and *department_address*, and a functional dependency $\text{department_number} \rightarrow \text{department_address}$
 - Good design would have made department an entity
- Functional dependencies from non-key attributes of a relationship set possible, but rare --- most relationships are binary





Denormalization for Performance

- May want to use non-normalized schema for performance
- For example, displaying *customer_name* along with *account_number* and *balance* requires join of *account* with *depositor*
- Alternative 1: Use denormalized relation containing attributes of *account* as well as *depositor* with all above attributes
 - faster lookup
 - extra space and extra execution time for updates
 - extra coding work for programmer and possibility of error in extra code
- Alternative 2: use a materialized view defined as
$$\text{account} \bowtie \text{depositor}$$
 - Benefits and drawbacks same as above, except no extra coding work for programmer and avoids possible errors





Other Design Issues

- Some aspects of database design are not caught by normalization
- Examples of bad database design, to be avoided:

Instead of *earnings* (*company_id*, *year*, *amount*), use

- *earnings_2004*, *earnings_2005*, *earnings_2006*, etc., all on the schema (*company_id*, *earnings*).
 - ▶ Above are in BCNF, but make querying across years difficult and needs new table each year
- *company_year*(*company_id*, *earnings_2004*, *earnings_2005*, *earnings_2006*)
 - ▶ Also in BCNF, but also makes querying across years difficult and requires new attribute each year.
 - ▶ Is an example of a **crosstab**, where values for one attribute become column names
 - ▶ Used in spreadsheets, and in data analysis tools





Modeling Temporal Data

- *Temporal data* have an association time interval during which the data are *valid*.
- A *snapshot* is the value of the data at a particular point in time
- Several proposals to extend ER model by adding valid time to
 - attributes, e.g. address of a customer at different points in time
 - entities, e.g. time duration when an account exists
 - relationships, e.g. time during which a customer owned an account
- But no accepted standard
- Adding a temporal component results in functional dependencies like
$$customer_id \rightarrow customer_street, customer_city$$
not to hold, because the address varies over time
- A *temporal functional dependency* $X \rightarrow Y$ holds on schema R if the functional dependency $X \rightarrow Y$ holds on all snapshots for all legal instances $r(R)$





Modeling Temporal Data (Cont.)

- In practice, database designers may add start and end time attributes to relations
 - E.g. $\text{course}(\text{course_id}, \text{course_title}) \rightarrow \text{course}(\text{course_id}, \text{course_title}, \text{start}, \text{end})$
 - ▶ Constraint: no two tuples can have overlapping valid times
 - Hard to enforce efficiently
- Foreign key references may be to current version of data, or to data at a point in time
 - E.g. student transcript should refer to course information at the time the course was taken





End of Chapter

Database System Concepts, 5th Ed.

©Silberschatz, Korth and Sudarshan

See www.db-book.com for conditions on re-use





Proof of Correctness of 3NF Decomposition Algorithm

Database System Concepts, 5th Ed.

©Silberschatz, Korth and Sudarshan

See www.db-book.com for conditions on re-use





Correctness of 3NF Decomposition Algorithm

- 3NF decomposition algorithm is dependency preserving (since there is a relation for every FD in F_c)
- Decomposition is lossless
 - A candidate key (C) is in one of the relations R_i in decomposition
 - Closure of candidate key under F_c must contain all attributes in R .
 - Follow the steps of attribute closure algorithm to show there is only one tuple in the join result for each tuple in R_i





Correctness of 3NF Decomposition Algorithm (Cont'd.)

Claim: if a relation R_i is in the decomposition generated by the above algorithm, then R_i satisfies 3NF.

- Let R_i be generated from the dependency $\alpha \rightarrow \beta$
- Let $\gamma \rightarrow B$ be any non-trivial functional dependency on R_i . (We need only consider FDs whose right-hand side is a single attribute.)
- Now, B can be in either β or α but not in both. Consider each case separately.





Correctness of 3NF Decomposition (Cont'd.)

■ Case 1: If B in β :

- If γ is a superkey, the 2nd condition of 3NF is satisfied
- Otherwise α must contain some attribute not in γ
- Since $\gamma \rightarrow B$ is in F^+ it must be derivable from F_c , by using attribute closure on γ .
- Attribute closure not have used $\alpha \rightarrow \beta$. If it had been used, α must be contained in the attribute closure of γ , which is not possible, since we assumed γ is not a superkey.
- Now, using $\alpha \rightarrow (\beta - \{B\})$ and $\gamma \rightarrow B$, we can derive $\alpha \rightarrow B$ (since $\gamma \subseteq \alpha \beta$, and $B \notin \gamma$ since $\gamma \rightarrow B$ is non-trivial)
- Then, B is extraneous in the right-hand side of $\alpha \rightarrow \beta$; which is not possible since $\alpha \rightarrow \beta$ is in F_c .
- Thus, if B is in β then γ must be a superkey, and the second condition of 3NF must be satisfied.





Correctness of 3NF Decomposition (Cont'd.)

■ Case 2: B is in α .

- Since α is a candidate key, the third alternative in the definition of 3NF is trivially satisfied.
- In fact, we cannot show that γ is a superkey.
- This shows exactly why the third alternative is present in the definition of 3NF.

Q.E.D.





Figure 7.5: Sample Relation r

A	B	C	D
a_1	b_1	c_1	d_1
a_1	b_2	c_1	d_2
a_2	b_2	c_2	d_2
a_2	b_3	c_2	d_3
a_3	b_3	c_2	d_4





Figure 7.6

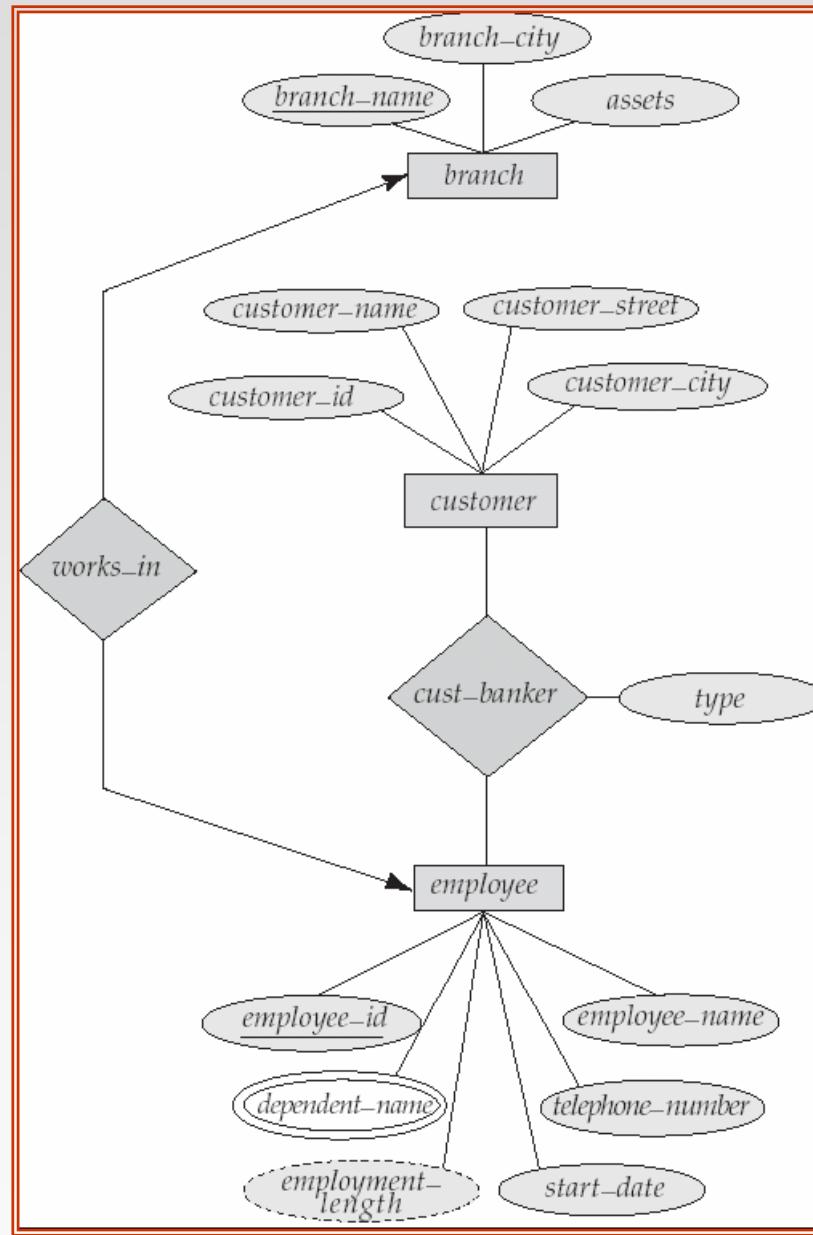




Figure 7.7

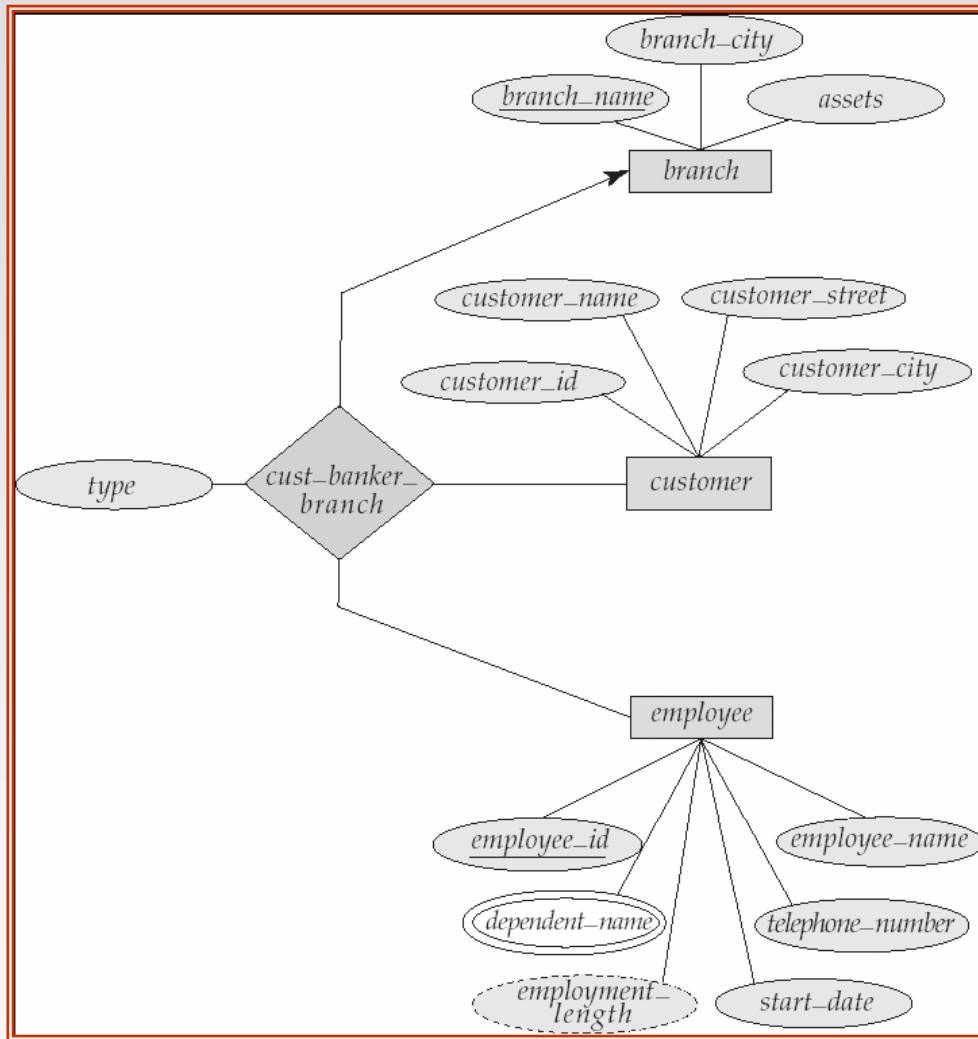




Figure 7.15: An Example of Redundancy in a BCNF Relation

<i>loan_number</i>	<i>customer_id</i>	<i>customer_street</i>	<i>customer_city</i>
L-23	99-123	North	Rye
L-23	99-123	Main	Manchester
L-93	15-106	Lake	Horseneck





Figure 7.16: An Illegal R_2 Relation

<i>loan_number</i>	<i>customer_id</i>	<i>customer_street</i>	<i>customer_city</i>
L-23	99-123	North	Rye
L-27	99-123	Main	Manchester





Figure 7.18: Relation of Practice

Exercise 7.2

A	B	C
a_1	b_1	c_1
a_1	b_1	c_2
a_2	b_1	c_1
a_2	b_1	c_3

