Chapter 1: Introduction

- Purpose of Database Systems
- View of Data
- Data Models
- Data Definition Language
- Data Manipulation Language
- Transaction Management
- Storage Management
- Database Administrator
- Database Users
- Overall System Structure
- DBMS Vs. IRS

Database Management System (DBMS)

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

Purpose of Database System

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

Purpose of Database Systems (Cont.)

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

Is the WWW a DBMS?

- Fairly sophisticated search available
  - crawler indexes pages for fast search
- But, currently
  - data is mostly unstructured and untyped
  - can't manipulate the data
  - few guarantees provided for freshness of data, consistency across data items, fault tolerance, …
  - Web sites typically have a DBMS in the background to provide these functions.
- The picture is quickly changing
  - New standards like XML can help data modeling
  - Research groups are working on providing some of this functionality across multiple web sites.

Is a File System a DBMS?

- Thought Experiment 1:
  - You and your project partner are editing the same file.
  - You both save it at the same time.
  - Whose changes survive?

  A) Yours B) Partner's C) Both D) Neither E) ???

- Thought Experiment 2:
  - You're updating a file.
  - The power goes out.
  - Which of your changes survive?

  A) All B) None C) All Since last save D) ???

Levels of Abstraction

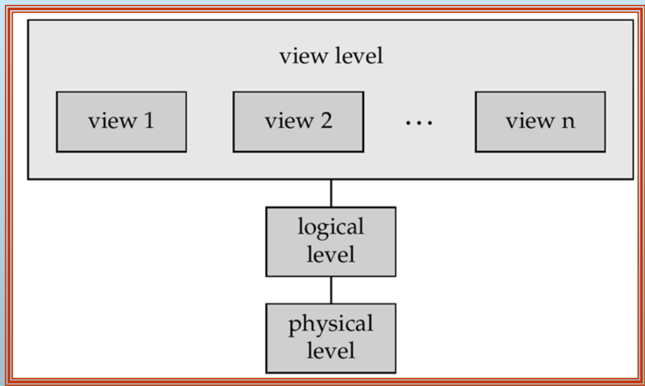
- Physical level describes how a record (e.g., customer) is stored.
- Logical level: describes data stored in database, and the relationships among the data.

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

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

View of Data

An architecture for a database system



Instances and Schemas

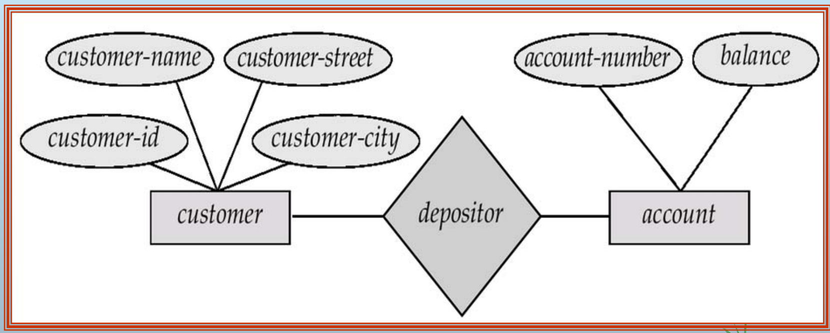
- Similar to types and variables in programming languages
- **Schema** – the logical structure of the database
  - e.g., the database consists of information about a set of customers and accounts and the relationship between them)
  - Analogous to type information of a variable in a program
  - **Physical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

Data Models

- A collection of tools for describing
  - data
  - data relationships
  - data semantics
  - data constraints
- Entity-Relationship model
- Relational model
- Other models:
  - object-oriented model
  - semi-structured data models
  - Older models: network model and hierarchical model

Entity-Relationship Model

Example of schema in the entity-relationship model



Entity Relationship Model (Cont.)

- E-R model of real world
  - Entities (objects)
    - E.g. customers, accounts, bank branch
  - Relationships between entities
    - E.g. Account A-101 is held by customer Johnson
    - Relationship set *depositor* associates customers with accounts
- Widely used for database design
  - Database design in E-R model usually converted to design in the relational model (coming up next) which is used for storage and processing

Relational Model

- Example of tabular data in the relational model

Attributes

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

A Sample Relational Database

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

(a) The *customer* table

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

Data Definition Language (DDL)

- Specification notation for defining the database schema
  - E.g.
    create table *account* (
    account-number **char**(10),
    balance **integer**)
- DDL compiler generates a set of tables stored in a *data dictionary*
- Data dictionary contains metadata (i.e., data about data)
  - database schema
  - Data *storage and definition* language
    - language in which the storage structure and access methods used by the database system are specified
    - Usually an extension of the data definition language

Data Manipulation Language (DML)

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

SQL

- SQL: widely used non-procedural language
  - E.g. find the name of the customer with customer-id 192-83-7465

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

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

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

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

Database Users

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

Database Administrator

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

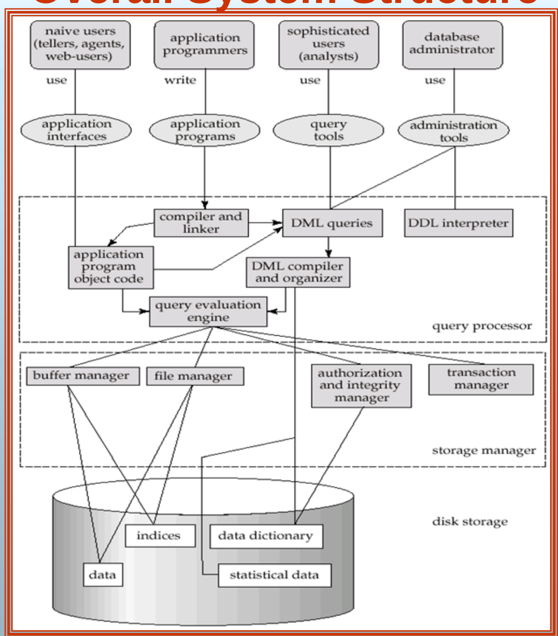
Transaction Management

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

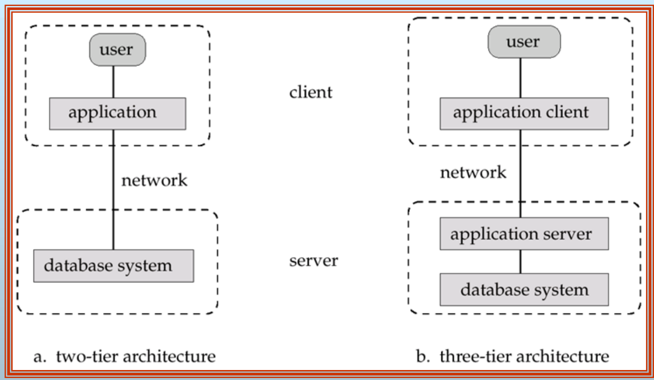
Storage Management

- Storage manager is a program module that provides the interface between the low-level data stored in the database and the application programs and queries submitted to the system.
- The storage manager is responsible to the following tasks:
  - interaction with the file manager
  - efficient storing, retrieving and updating of data

Overall System Structure



Application Architectures



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

Advantages of a DBMS

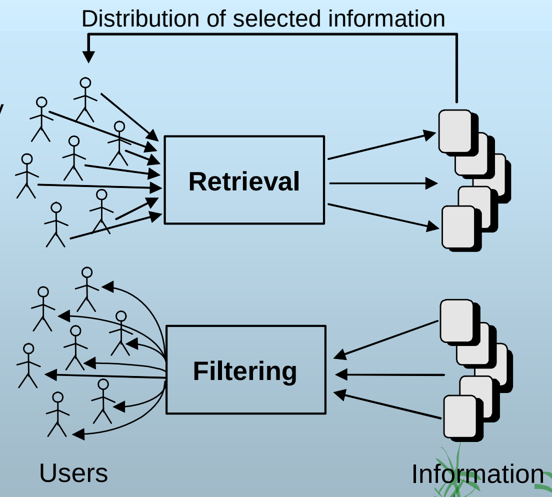
- **Data independence**
- **Efficient data access**
- **Data integrity & security**
- **Data administration**
- **Concurrent access, crash recovery**
- **Reduced application development time**

- **So why not use them always?**
  - Expensive/complicated to set up & maintain
  - This cost & complexity must be offset by need
  - General-purpose, not suited for special-purposetasks (e.g. text search!)

DBMS vs. IRS

DBMS: The entities are uniquely and completely described by its attributes.

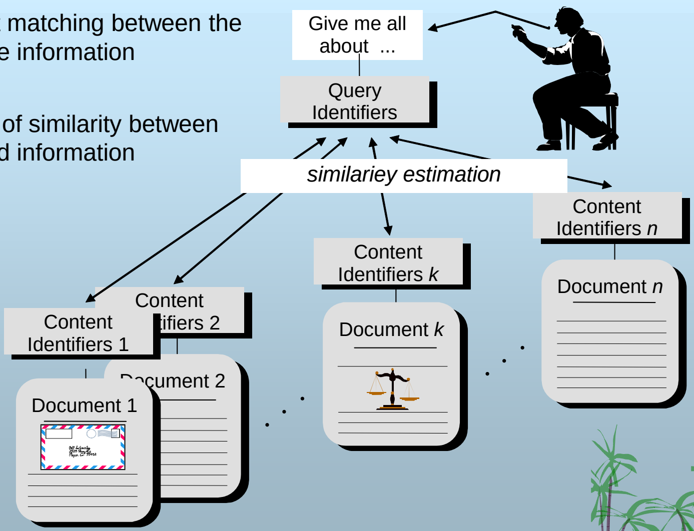
IRS: The number of content identifiers can be very large and they do not describe the information uniquely and completely.



Search vs. Retrieval

DBMS: Strict matching between the query and the information identifiers.

IRS: Degree of similarity between the query and information identifiers.



Chapter 2: Entity-Relationship Model

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

Entity Sets

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

Entity Sets *customer* and *loan*

customer

customer-id	customer-name	customer-street	customer-city
321-12-3123	Jones	Main	Harrison
019-28-3746	Smith	North	Rye
677-89-9011	Hayes	Main	Harrison
555-55-5555	Jackson	Dupont	Woodside
244-66-8800	Curry	North	Rye
963-96-3963	Williams	Nassau	Princeton
335-57-7991	Adams	Spring	Pittsfield

loan

loan-number	amount
L-17	1000
L-23	2000
L-15	1500
L-14	1500
L-19	500
L-11	900
L-16	1300

Attributes

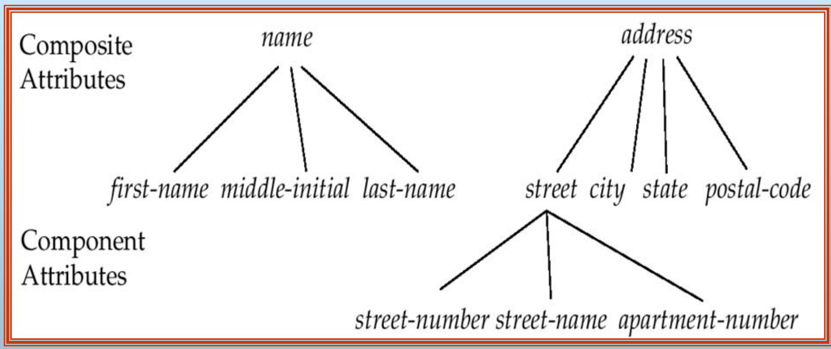
- An entity is represented by a set of attributes, that is descriptive properties possessed by all members of an entity set.

  Example:

  customer = (customer-id, customer-name,
  customer-street, customer-city)
  loan = (loan-number, amount)

- *Domain* – the set of permitted values for each attribute
- Attribute types:
  - *Simple* and *composite* attributes.
  - *Single-valued* and *multi-valued* attributes
    - E.g. multivalued attribute: *phone-numbers*
  - *Derived* attributes
    - Can be computed from other attributes
    - E.g. *age*, given date of birth

Composite Attributes



Relationship Sets

- A relationship is an association among several entities

  Example:

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

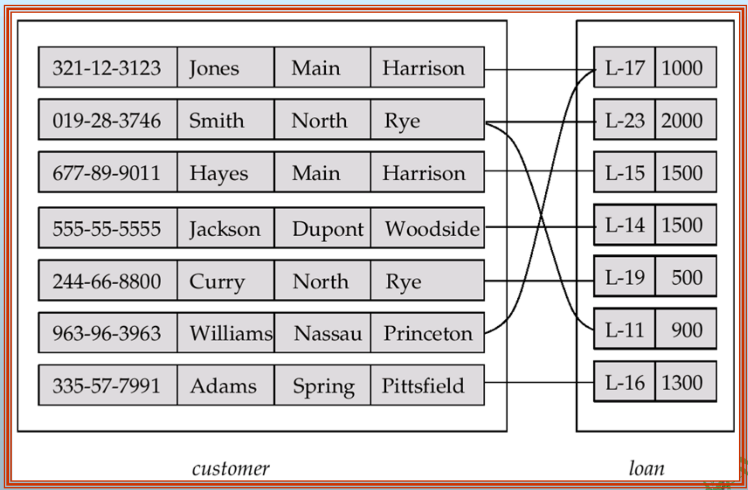
- A *relationship set* is a mathematical relation among $n \geq 2$ entities, each taken from entity sets

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

  where $(e_1, e_2, \ldots, e_n)$ is a relationship
  - Example:

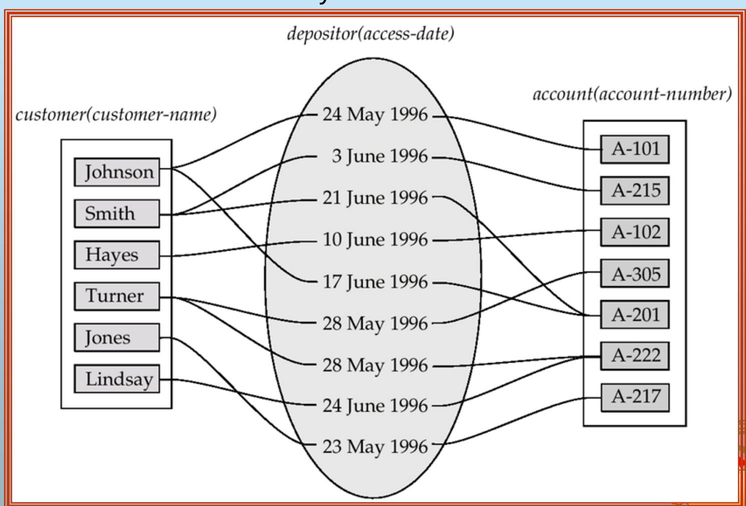
    (Hayes, A-102) $\in$ *depositor*

Relationship Set *borrower*



Relationship Sets (Cont.)

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



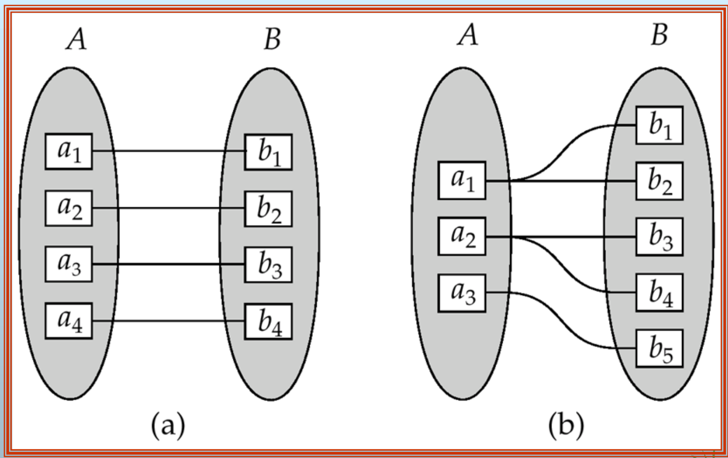
Degree of a Relationship Set

- 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.
  - E.g. Suppose employees of a bank may have jobs (responsibilities) at multiple branches, with different jobs at different branches. Then there is a ternary relationship set between entity sets *employee, job and branch*
- Relationships between more than two entity sets are rare. Most relationships are binary. (More on this later.)

Mapping Cardinalities

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

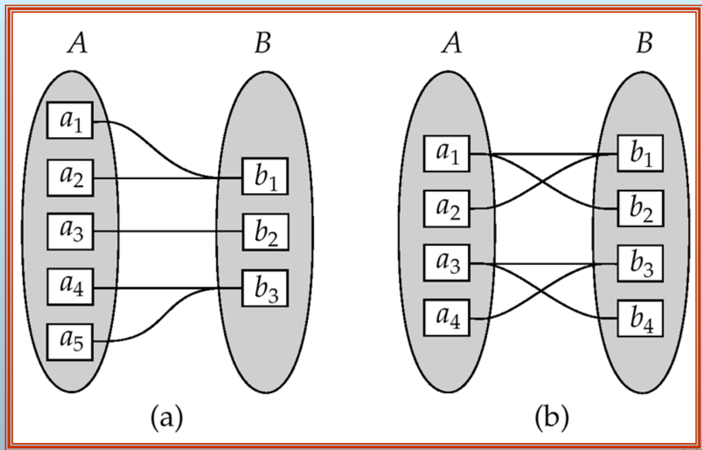
Mapping Cardinalities



One to one　　　　One to many

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

Mapping Cardinalities

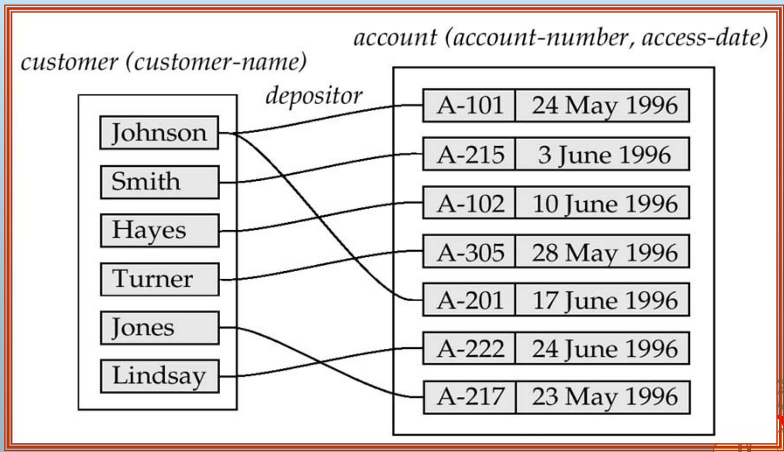


Many to one　　　　Many to many

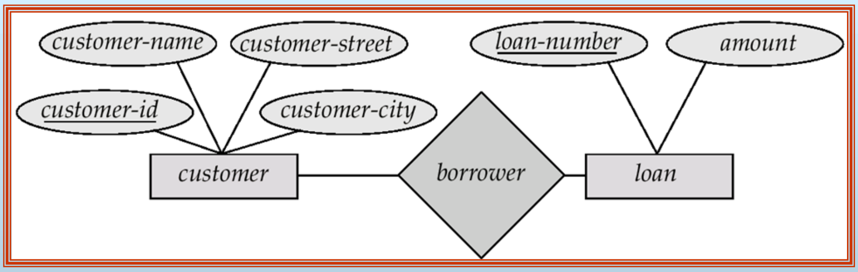
Note: Some elements in A and B may not be mapped to any elements in the other set

Mapping Cardinalities affect ER Design

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

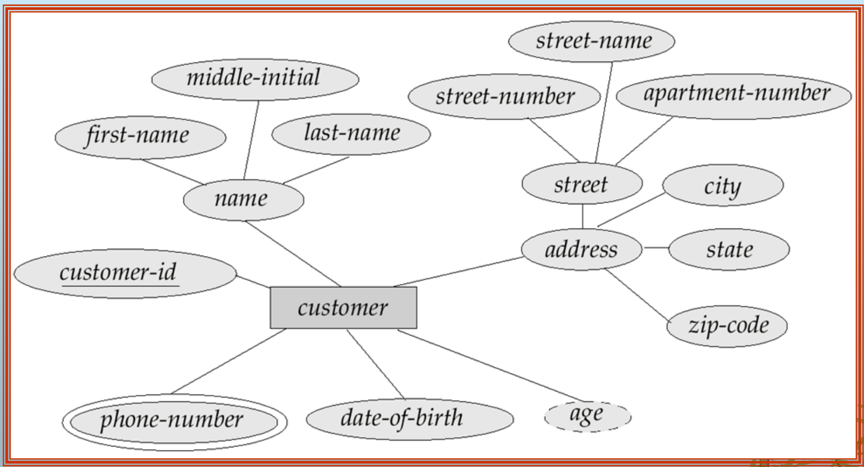


E-R Diagrams

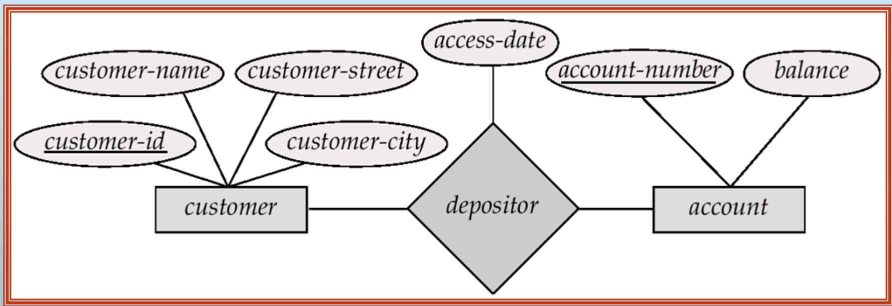


- **Rectangles** represent entity sets.
- **Diamonds** represent relationship sets.
- **Lines** link attributes to entity sets and entity sets to relationship sets.
- **Ellipses** represent attributes
  - **Double ellipses** represent multivalued attributes.
  - **Dashed ellipses** denote derived attributes.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

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

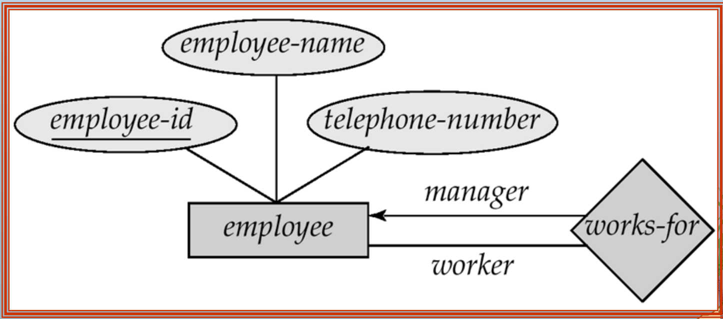


Relationship Sets with Attributes



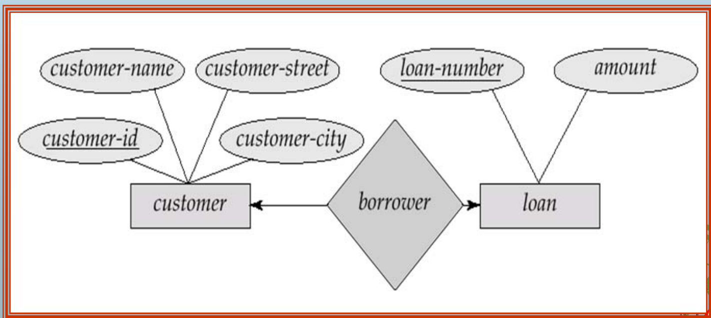
Roles

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



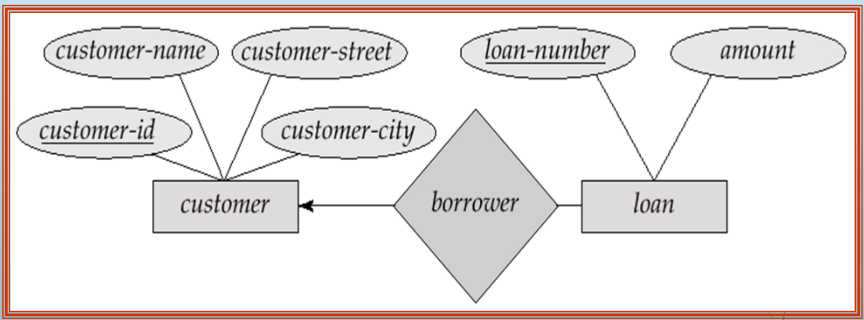
Cardinality Constraints

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



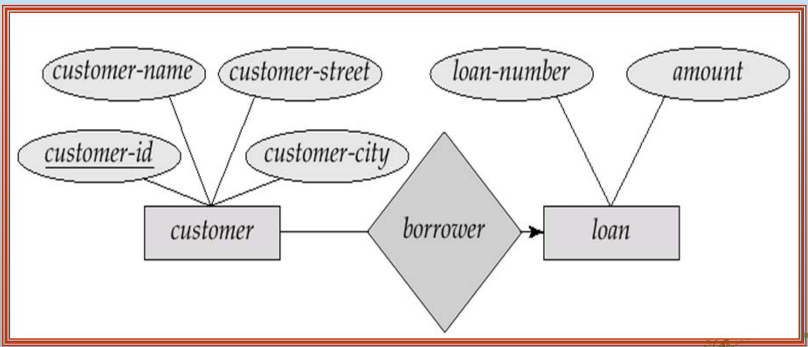
One-To-Many Relationship

- 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*

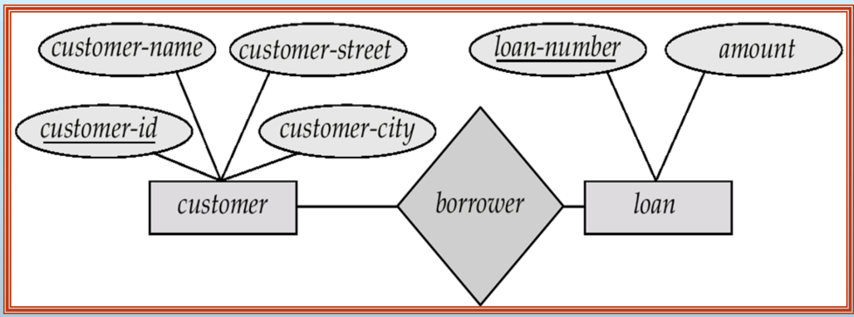


Many-To-One Relationships

- 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*



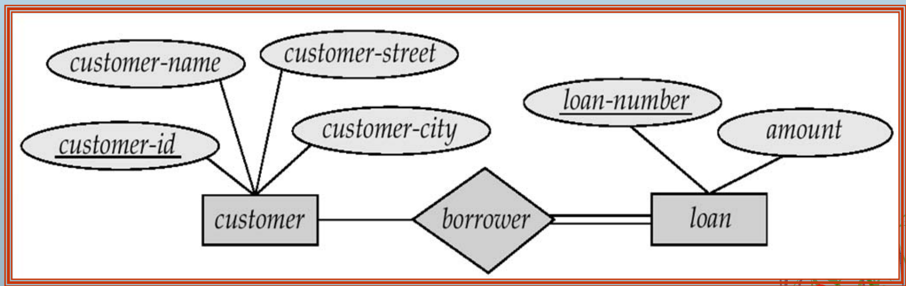
Many-To-Many Relationship



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

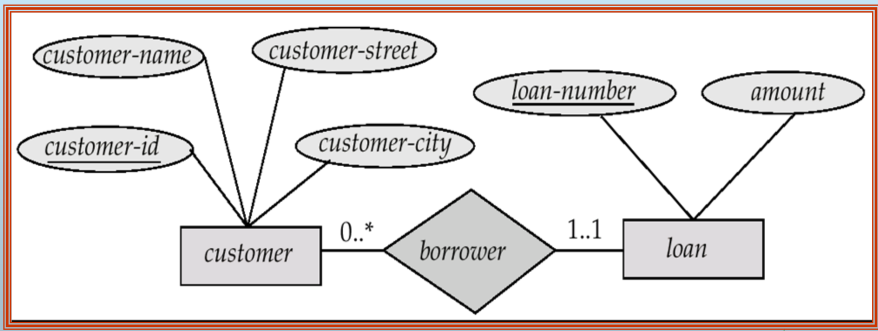
Participation of an Entity Set in a Relationship Set

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



Alternative Notation for Cardinality Limits

- Cardinality limits can also express participation constraints



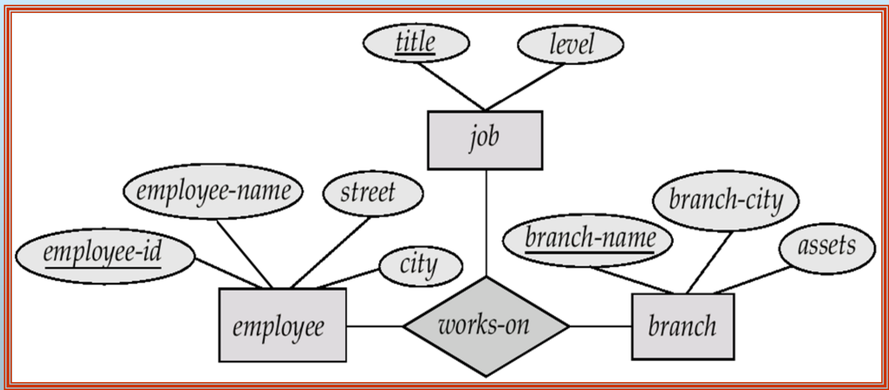
Keys

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

Keys for Relationship Sets

- The combination of primary keys of the participating entity sets forms a super key of a relationship set.
  - (*customer-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.*
    - E.g. if we wish to track all access-dates to each account by each customer, we cannot assume a relationship for each access. We can use a multivalued attribute though
- Must consider the mapping cardinality of the relationship set when deciding the 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 Diagram with a Ternary Relationship



Cardinality Constraints on Ternary Relationship

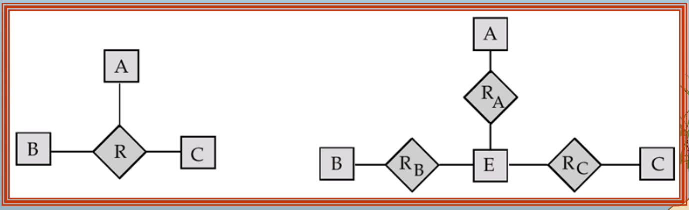
- We allow at most one arrow out of a ternary (or greater degree) relationship to indicate a cardinality constraint
- E.g. an arrow from *works-on* to *job* indicates each employee works on at most one job at any branch.
- If there is more than one arrow, there are two ways of defining the meaning.
  - 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

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 know)
  - But there are some relationships that are naturally non-binary
    - E.g. *works-on*

Converting Non-Binary Relationships to Binary Form

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

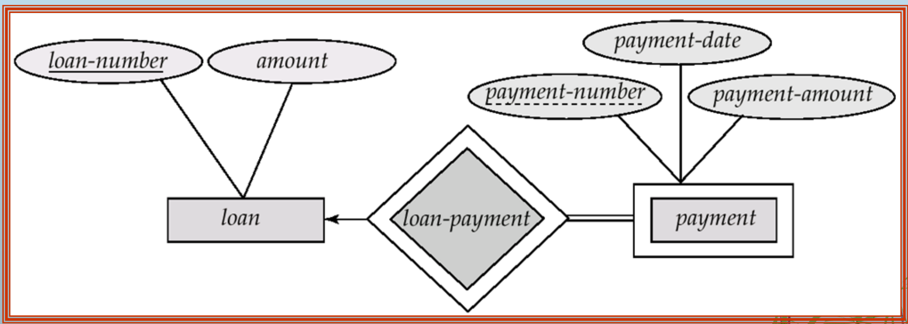
- Also need to translate constraints
  - Translating all constraints may not be possible
  - There may be instances in the translated schema that cannot correspond to any instance of *R*
    - *Exercise: add constraints to the relationships* R_A, R_B *and* R_C to ensure that a newly created entity corresponds 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

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*)



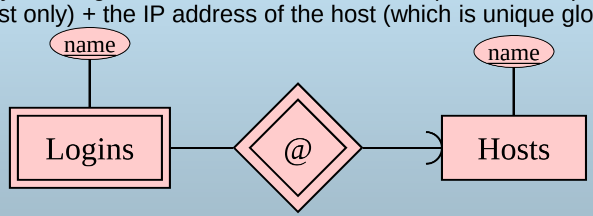
Weak Entity Sets (Cont.)

- 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*

Example: Logins (Email Addresses)

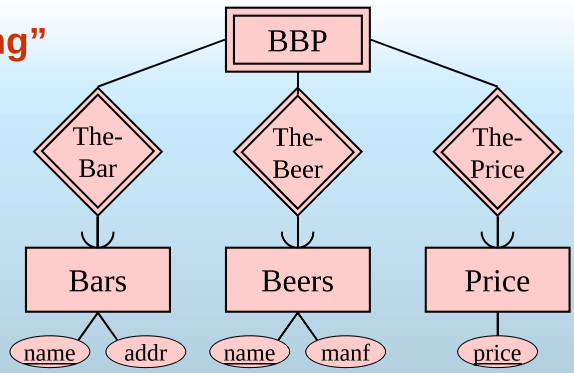
Login name = user name + host name, e.g.,
`ark@soe.ucsc.edu`.

- A “login” entity corresponds to a user name on a particular host, but the passwd table doesn’t record the host, just the user name, e.g., `ark`.
- Key for a login = the user name at the host (which is unique for that host only) + the IP address of the host (which is unique globally).



- Design issue: *Under what circumstances could we simply make login-name and host-name be attributes of logins, and dispense with the weak E.S.?*

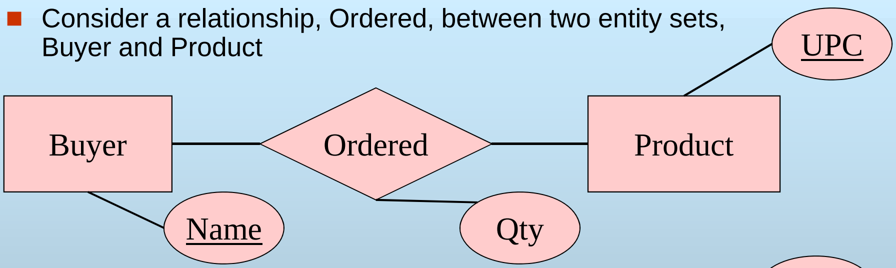
All "Connecting" Entity Sets Are Weak



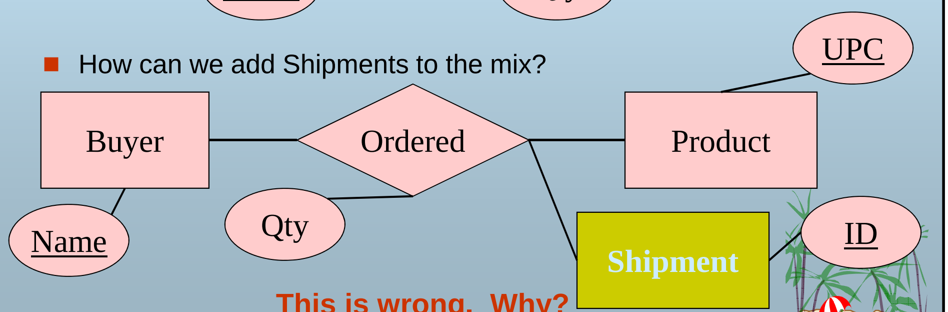
- In this special case, where bar and beer determine a price, we can omit `price` from the key, and remove the double diamond from `ThePrice`.
- Better: `price` is attribute of `BBP`.

Relationship To Weak Entities

- Consider a relationship, Ordered, between two entity sets, Buyer and Product

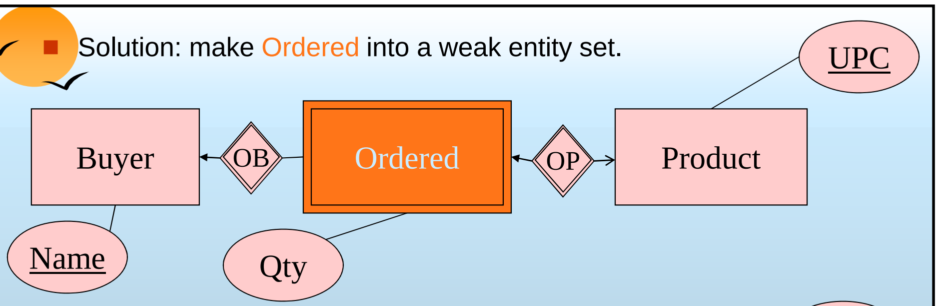


- How can we add Shipments to the mix?

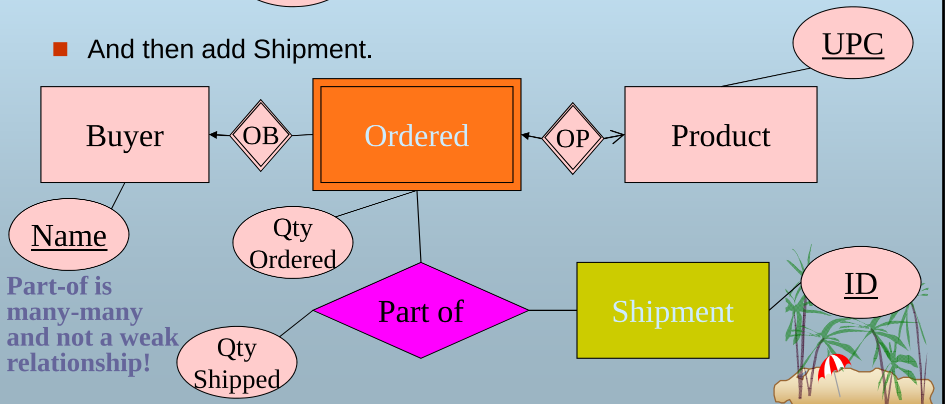


This is wrong. Why?

- Solution: make Ordered into a weak entity set.



- And then add Shipment.



Design Issues

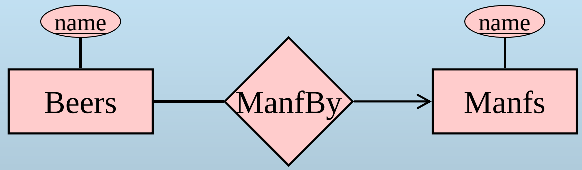
- Use of entity sets vs. attributes
  Choice mainly depends on the structure of the enterprise being modeled, and on the semantics associated with the attribute in question.
- Use of entity sets vs. relationship sets
  Possible guideline is to designate a relationship set to describe an action that occurs 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.
- Avoid redundancy
  Redudancy wastes space and encourages inconsistency.
- Don't overuse weak entity sets

Entity Sets Vs. Attributes

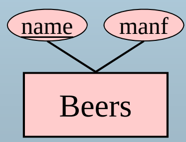
You may be unsure which concepts are worthy of being entity sets, and which are handled more simply as attributes.

- Especially tricky for the class design project, since there is a temptation to create needless entity sets to make project "larger."

Wrong:



Right:



Make an entity set only if it either:

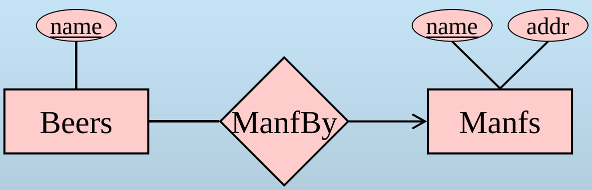
Is more than a name of something; *i.e.*, it has nonkey attributes or relationships with a number of different entity sets,

or

Is the "many" in a many-one relationship.

Example

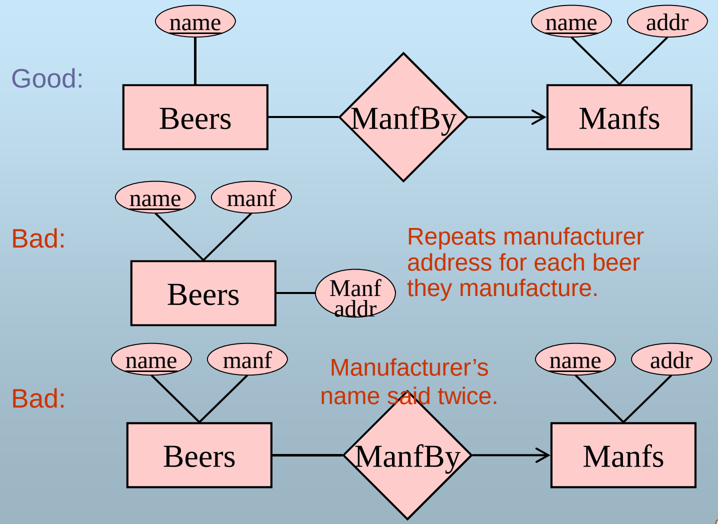
The following design illustrates both points:



- *Manfs* deserves to be an E.S. because we record *addr*, a nonkey attribute.
- *Beers* deserves to be an E.S. because it is at the "many" end.
  - If not, we would have to make "set of beers" an attribute of *Manfs* – something we avoid doing, although some may tell you it is OK in E/R model.

Avoid redundancy

Setting: client has (possibly vague) idea of what he/she wants. You must design a database that represents these thoughts and only these thoughts.



Don't Overuse Weak E.S.

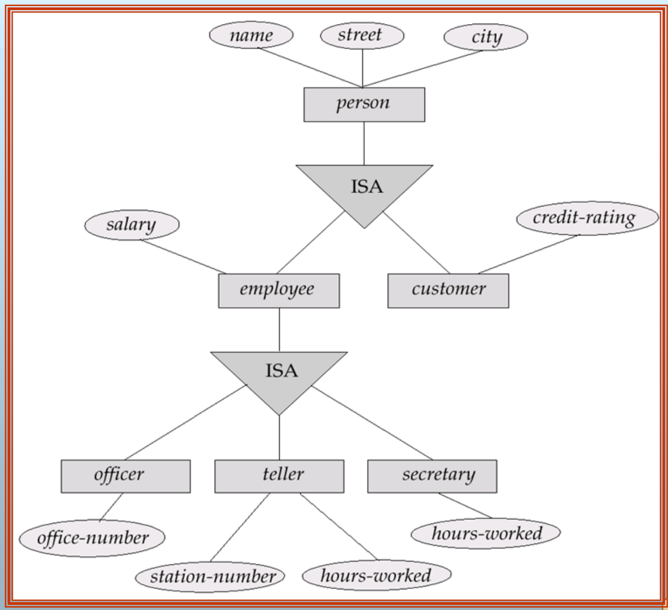
- There is a tendency to feel that no E.S. has its entities uniquely determined without following some relationships.
- However, in practice, we almost always create unique ID's to compensate: social-security numbers, VIN's, etc.
- The only times weak E.S.'s seem necessary are when:
  - We can't easily create such ID's; e.g., no one is going to accept a “species ID” as part of the standard nomenclature (species is a weak E.S. supported by membership in a genus).
  - There is no global authority to create them, *e.g.*, crews and studios.

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

Specialization

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

Specialization Example



Generalization

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

Specialization and Generalization (Contd.)

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

Design Constraints on a Specialization/Generalization

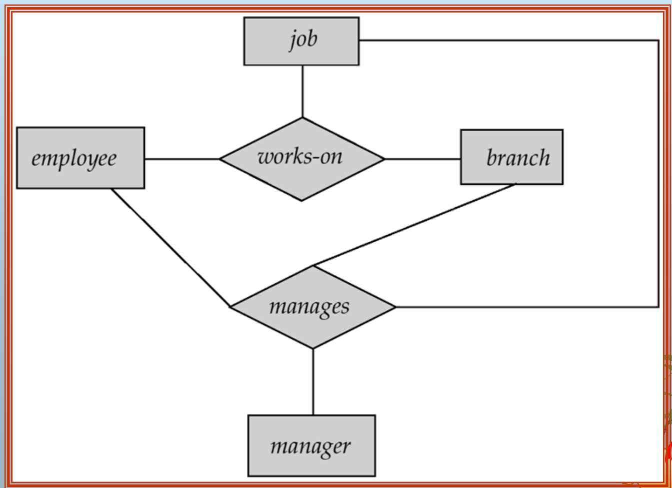
- Constraint on which entities can be members of a given lower-level entity set.
  - condition-defined
    - E.g. all customers over 65 years are members of *senior-citizen* entity set; *senior-citizen* ISA *person*.
  - user-defined
- Constraint on whether or not entities may belong to more than one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

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

Aggregation

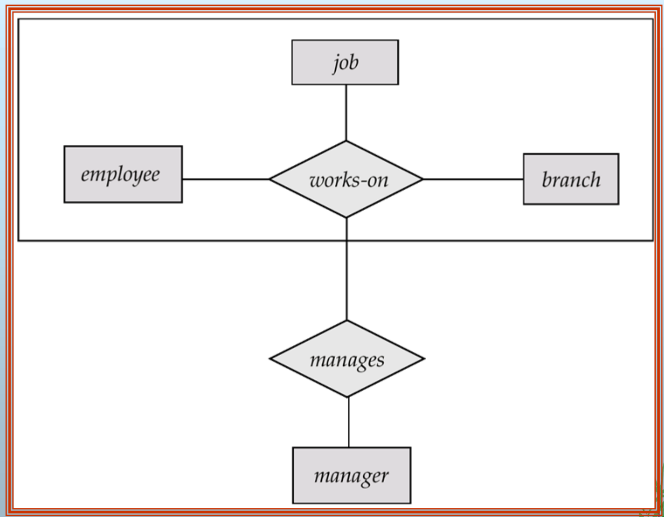
- Consider the ternary relationship *works-on*, which we saw earlier
- Suppose we want to record managers for tasks performed by an employee at a branch



Aggregation (Cont.)

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

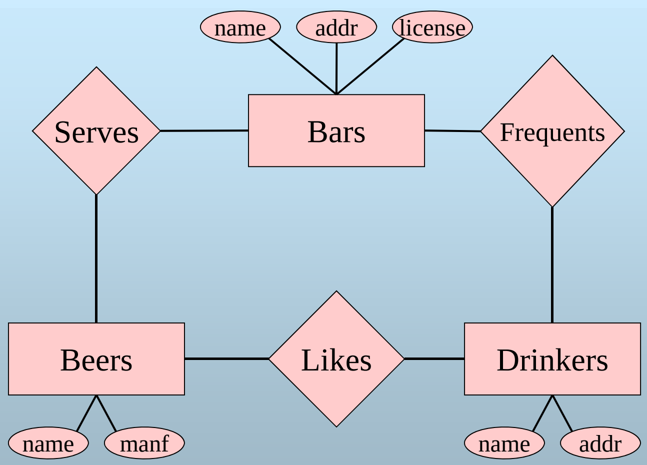
E-R Diagram With Aggregation



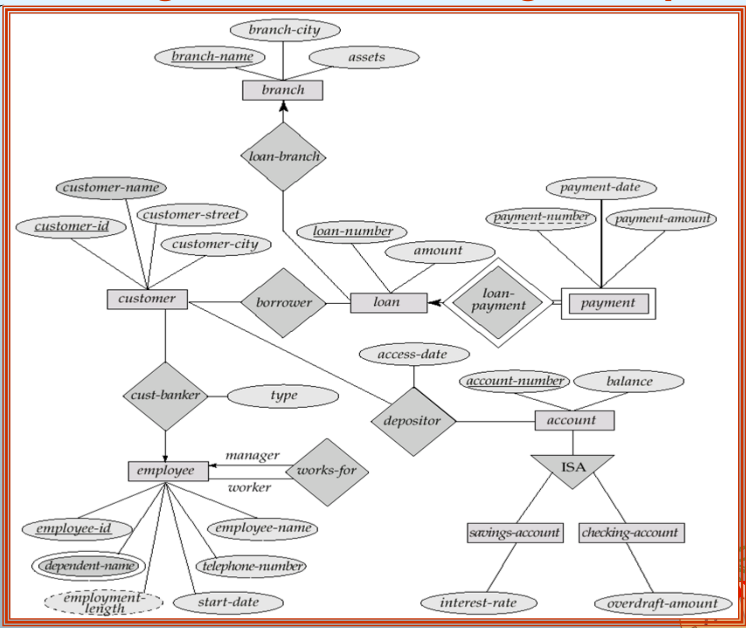
E-R Design Decisions

- 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.

Beers-Bars-Drinkers Example



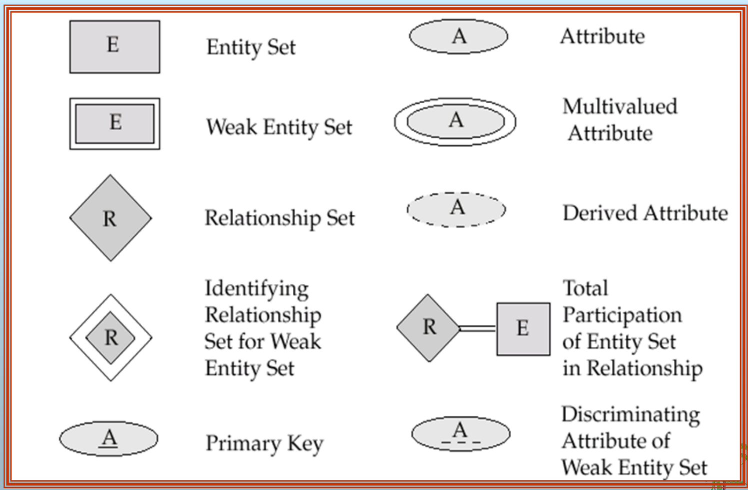
E-R Diagram for a Banking Enterprise



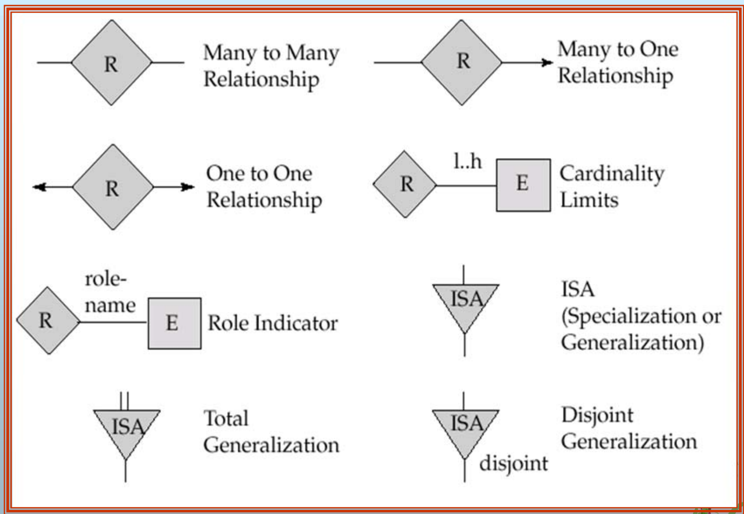
Database System Concepts 2.55 ©Silberschatz, Korth and Sudarshan

How about doing another ER design interactively on the board?

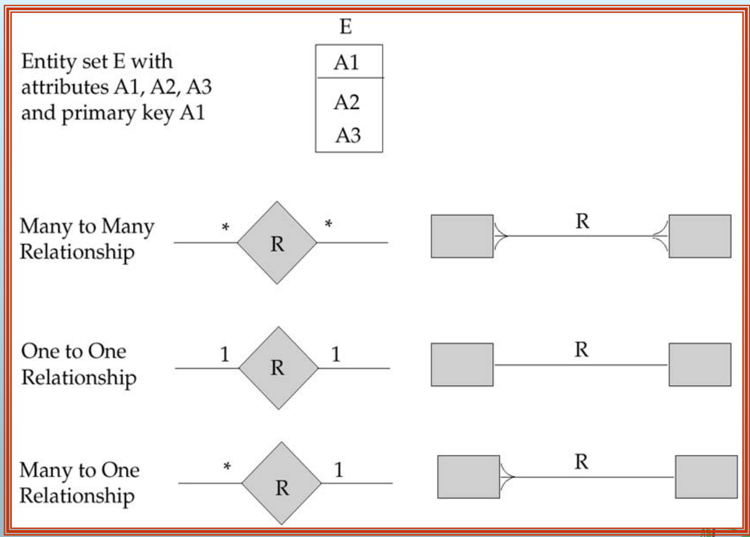
Summary of Symbols Used in E-R Notation



Summary of Symbols (Cont.)



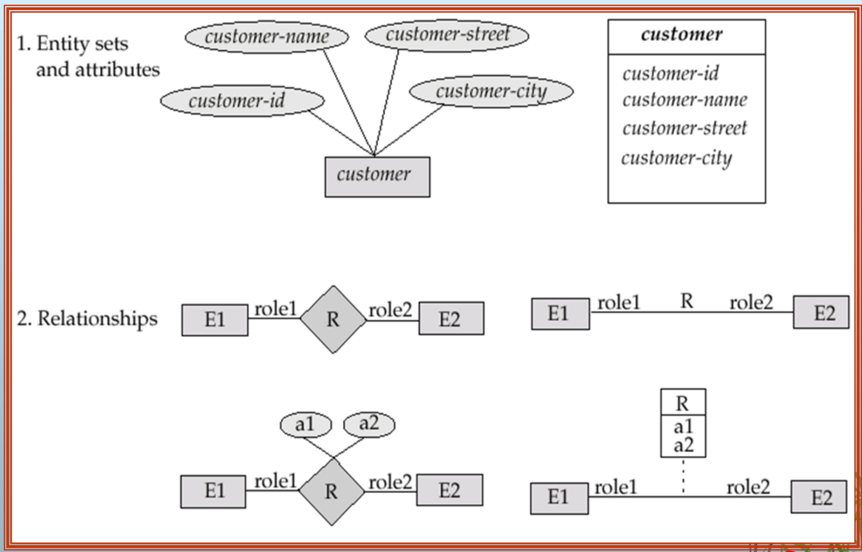
Alternative E-R Notations



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.

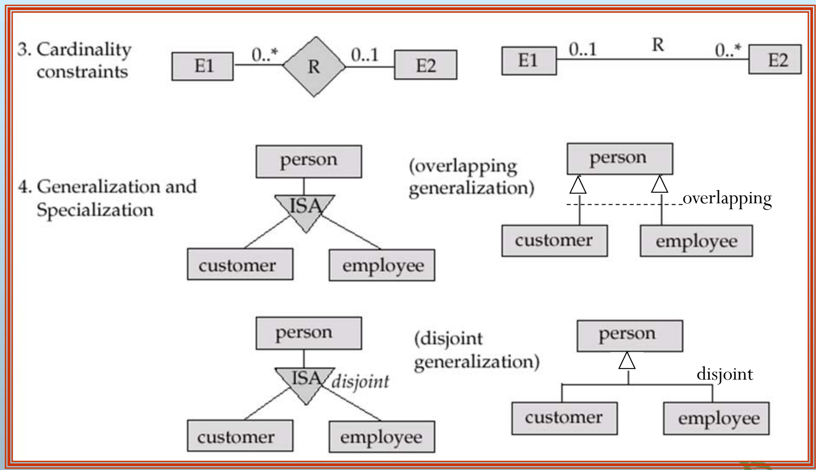
Summary of UML Class Diagram Notation



UML Class Diagrams (Contd.)

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



*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 *E*2 side and 0..1 on the *E*1 side means that each *E*2 entity can participate in at most one relationship, whereas each *E*1 entity can participate in many relationships; in other words, the relationship is many to one from *E*2 to *E*1.
- 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..*.

Reduction of an E-R Schema to Tables

- Primary keys allow entity sets and relationship sets to be expressed uniformly as *tables* which represent the contents of the database.
- A database which conforms to an E-R diagram can be represented by a collection of tables.
- For each entity set and relationship set there is a unique table which is assigned the name of the corresponding entity set or relationship set.
- Each table has a number of columns (generally corresponding to attributes), which have unique names.
- Converting an E-R diagram to a table format is the basis for deriving a relational database design from an E-R diagram.

Representing Entity Sets as Tables

- A strong entity set reduces to a table with the same attributes.

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

Composite and Multivalued Attributes

- Composite attributes are flattened out by creating a separate attribute for each component attribute
  - E.g. given entity set *customer* with composite attribute *name* with component attributes *first-name* and *last-name* the table corresponding to the entity set has two attributes
    name.first-name and *name.last-name*
- A multivalued attribute M of an entity E is represented by a separate table EM
  - Table EM has attributes corresponding to the primary key of E and an attribute corresponding to multivalued attribute M
  - E.g. Multivalued attribute *dependent-names* of *employee* is represented by a table
    employee-dependent-names(*employee-id, dname*)
  - Each value of the multivalued attribute maps to a separate row of the table EM
    - E.g., an employee entity with primary key John and dependents Johnson and Johndotir maps to two rows: (John, Johnson) and (John, Johndotir)

Representing Weak Entity Sets

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

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

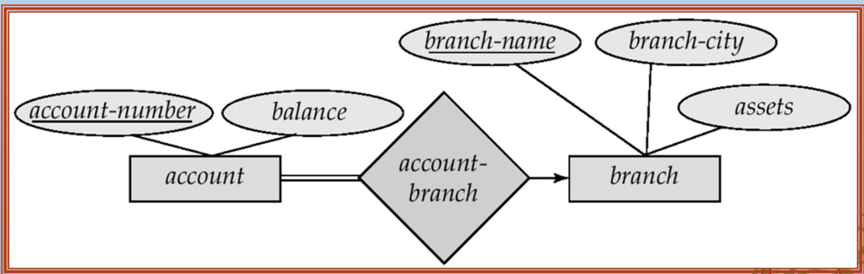
Representing Relationship Sets as Tables

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

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

Redundancy of Tables

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



Redundancy of Tables (Cont.)

- For one-to-one relationship sets, either side can be chosen to act as the "many" side
  - That is, extra attribute can be added to either of the tables corresponding to the two entity sets
- If participation is *partial* on the many side, replacing a table by an extra attribute in the relation corresponding to the "many" side could result in null values
- The table corresponding to a relationship set linking a weak entity set to its identifying strong entity set is redundant.
  - E.g. The *payment* table already contains the information that would appear in the *loan-payment* table (i.e., the columns loan-number and *payment-number*).

Representing Specialization as Tables

- Method 1:
  - Form a table for the higher level entity
  - Form a table for each lower level entity set, include primary key of higher level entity set and local attributes

table	table attributes
person	*name, street, city*
customer	*name, credit-rating*
employee	*name, salary*

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

Representing Specialization as Tables (Cont.)

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

table	table attributes
person	*name, street, city*
customer	*name, street, city, credit-rating*
employee	*name, street, city, salary*

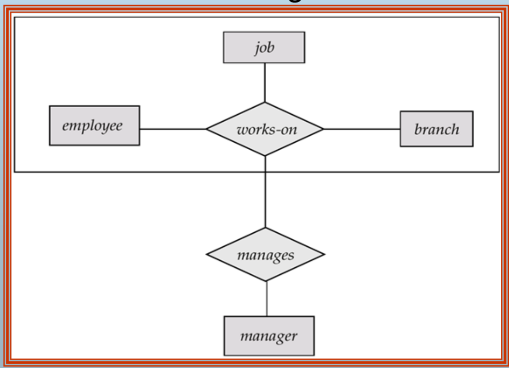
  - If specialization is total, table for generalized entity (*person*) not required to store information
    - Can be defined as a "view" relation containing union of specialization tables
    - But explicit table may still be needed for foreign key constraints
  - Drawback: street and city may be stored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

Relations Corresponding to Aggregation

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

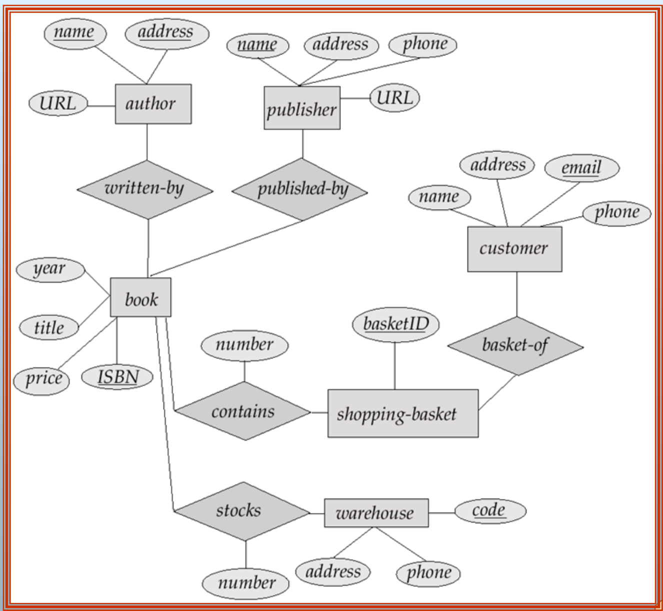
Relations Corresponding to Aggregation (Cont.)

- E.g. to represent aggregation *manages* between relationship *works-on* and entity set *manager*, create a table
  manages(*employee-id, branch-name, title, manager-name*)
- Table *works-on* is redundant **provided** we are willing to store null values for attribute *manager-name* in table *manages*

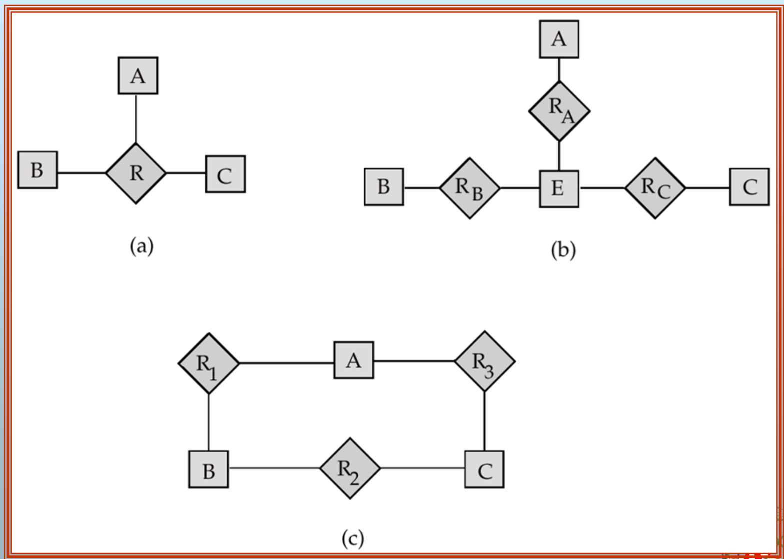


End of Chapter 2

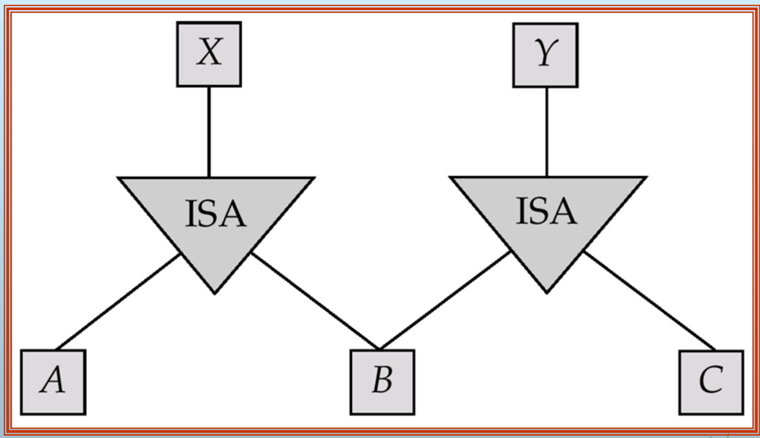
E-R Diagram for Exercise 2.10



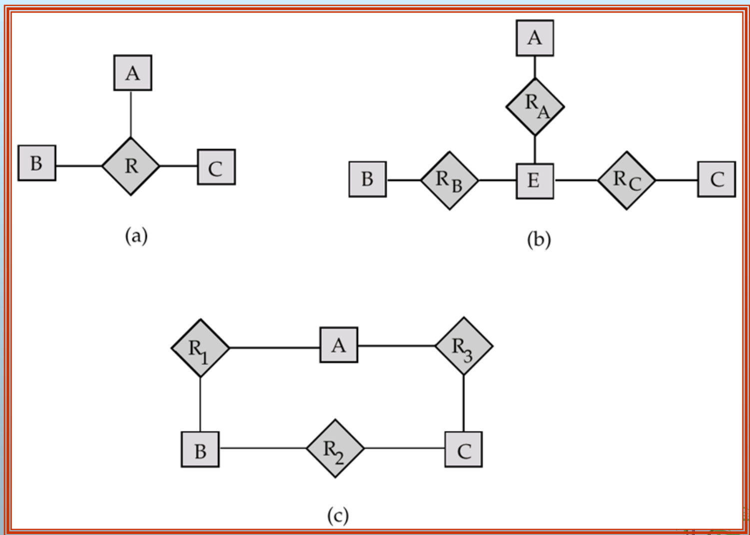
E-R Diagram for Exercise 2.15



E-R Diagram for Exercise 2.22

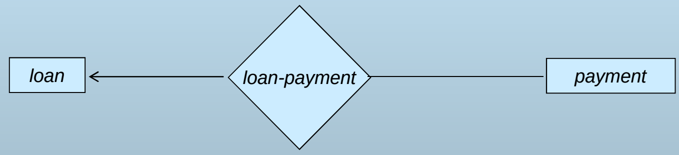


E-R Diagram for Exercise 2.15



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.

Chapter 3: Relational Model

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

Example of a Relation

account-number	*branch-name*	*balance*
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, …. D_n a **relation** r is a subset of $D_1 \times D_2 \times \ldots \times D_n$
  Thus a relation is a set of n-tuples $(a_1, a_2, \ldots, a_n)$ where each $a_i \in D_i$
- Example: if

  customer-name = {Jones, Smith, Curry, Lindsay}
  customer-street = {Main, North, Park}
  customer-city = {Harrison, Rye, Pittsfield}

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

  is a relation over *customer-name x customer-street x customer-city*

Attribute Types

- Each attribute of a relation has a name
- The set of allowed values for each attribute is called the **domain** of the attribute
- Attribute values are (normally) required to be **atomic**, that is, indivisible
  - E.g. multivalued attribute values are not atomic
  - E.g. composite attribute values are not atomic
- The special value *null* is a member of every domain
- The null value causes complications in the definition of many operations
  - we shall ignore the effect of null values in our main presentation and consider their effect later

Relation Schema

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

  E.g. *Customer-schema* =
  (*customer-name, customer-street, customer-city*)
- $r(R)$ is a *relation* on the *relation schema R*

  E.g. *customer (Customer-schema)*

Relation Instance

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

attributes
(or columns)

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

tuples
(or rows)

customer

Relations are Unordered

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

account-number	*branch-name*	*balance*
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

Why Relations?

- Very simple model.
- *Often* a good match for the way we think about our data.
- Abstract model that underlies SQL, the most important language in DBMS's today.
  - But SQL uses "bags" while the abstract relational model is set-oriented.
- All ingenious ideas are simple !

Database

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

  E.g.: *account* : stores information about accounts
  depositor : stores information about which customer owns which account
  customer : stores information about customers
- Storing all information as a single relation such as
  bank(*account-number, balance, customer-name*, ..)
  results in
  - repetition of information (e.g. two customers own an account)
  - the need for null values (e.g. represent a customer without an account)
- Normalization theory (Chapter 7) deals with how to design relational schemas

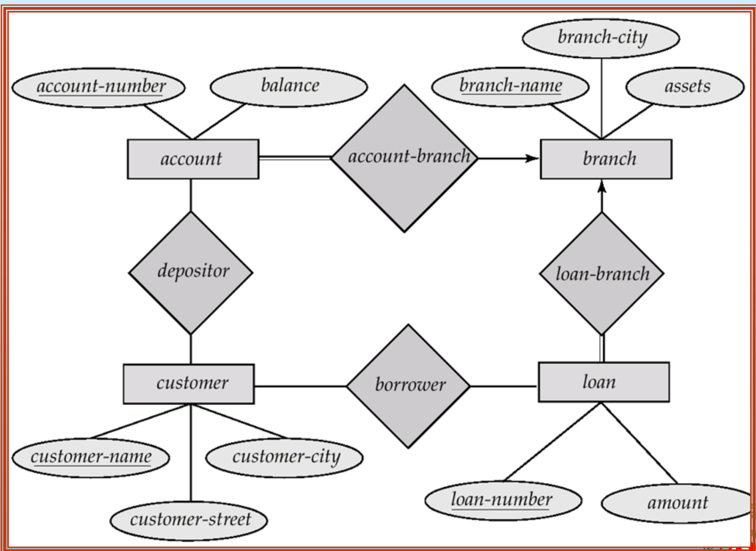
The *customer* Relation

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

The *depositor* Relation

customer-name	*account-number*
Hayes	A-102
Johnson	A-101
Johnson	A-201
Jones	A-217
Lindsay	A-222
Smith	A-215
Turner	A-305

E-R Diagram for the Banking Enterprise



Keys

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

Example 1

Drinkers(<u>name</u>, addr, <u>beersLiked</u>, manf, favoriteBeer)

- {name, beersLiked} FD's all attributes, as seen.
  - Shows {name, beersLiked} is a superkey.
- name $\rightarrow$ beersLiked is false, so name is not a superkey.
- beersLiked $\rightarrow$ name also false, so beersLiked is not a superkey.
- Thus, {name, beersLiked} is a key.
- No other keys in this example.
  - Neither name nor beersLiked is on the right of any observed FD, so they must be part of *any* superkey.
- Important point: "key" in a relation refers to tuples, not the entities they represent. If an entity is represented by several tuples, then entity-key will not be the same as relation-key.

Example 2

Lastname Firstname Student ID Major

$\underbrace{\text{Lastname Firstname}}_{\text{Key (2 attributes)}}$ $\underbrace{\underbrace{\text{Student ID}}_{\text{Key}}\ \text{Major}}_{\text{Superkey}}$

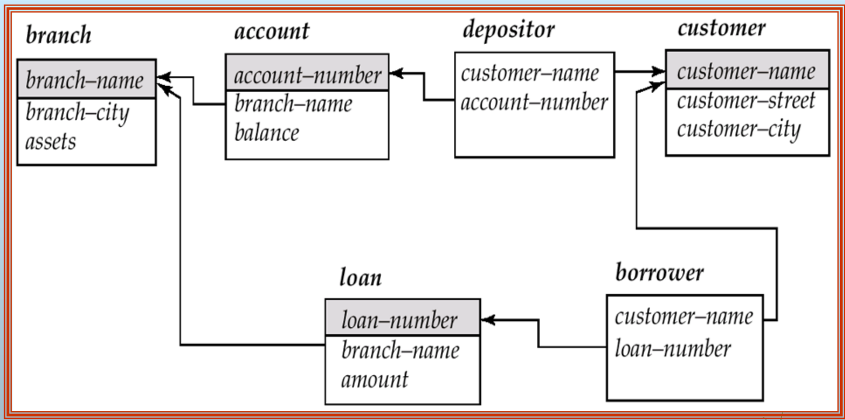
Note: There are <u>alternate</u> keys

- Keys are {Lastname, Firstname} and {StudentID}

Determining Keys from E-R Sets

- **Strong entity set**. The primary key of the entity set becomes the primary key of the relation.
- **Weak entity set**. The primary key of the relation consists of the union of the primary key of the strong entity set and the discriminator of the weak entity set.
- **Relationship set**. The union of the primary keys of the related entity sets becomes a super key of the relation.
  - For binary many-to-one relationship sets, the primary key of the "many" entity set becomes the relation's primary key.
  - For one-to-one relationship sets, the relation's primary key can be that of either entity set.
  - For many-to-many relationship sets, the union of the primary keys becomes the relation's primary key

Schema Diagram for the Banking Enterprise



Query Languages

- Language in which user requests information from the database.
- Categories of languages
  - procedural
  - non-procedural
- "Pure" languages:
  - Relational Algebra
  - Tuple Relational Calculus
  - Domain Relational Calculus
- Pure languages form underlying basis of query languages that people use.

Relational Algebra

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

Select Operation – 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:

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

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:

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

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:

$$\prod_{A1,\ A2,\ \ldots,\ Ak} (r)$$

  where A_1, A_2 are attribute names and *r* is a relation name.
- The result is defined as the relation of *k* columns obtained by erasing the columns that are not listed
- Duplicate rows removed from result, since relations are sets
- E.g. To eliminate the *branch-name* attribute of *account*

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

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* (e.g., 2nd column
     of r deals with the same type of values as does the 2nd column of s)

- E.g. to find all customers with either an account or a loan
  $\Pi_{customer\text{-}name}$ (*depositor*) $\cup$ $\Pi_{customer\text{-}name}$ (*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:

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

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

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 x *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* x *s*
- Defined as:

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

- 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 x 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	20	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

$$\rho_{X\ (A1,\ A2,\ \ldots,\ An)} (E)$$

returns the result of expression *E* under the name *X*, and with the attributes renamed to *A1, A2, …., An*.

Banking Example

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

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

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

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

depositor (customer-name, account-number)

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

$$\Pi_{loan\text{-}number}\ (\sigma_{amount\ >\ 1200}\ (loan))$$

Example Queries

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

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

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

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

Example Queries

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

$$\Pi_{customer\text{-}name}\ (\sigma_{branch\text{-}name=\text{“Perryridge”}}\ (\sigma_{borrower.loan\text{-}number\ =\ loan.loan\text{-}number}(borrower \times loan)))$$

- 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.

$$\Pi_{customer\text{-}name}\ (\sigma_{branch\text{-}name\ =\ \text{“Perryridge”}}\ (\sigma_{borrower.loan\text{-}number\ =\ loan.loan\text{-}number}(\text{borrower x loan}))) - \Pi_{customer\text{-}name}(\text{depositor})$$

Example Queries

- Find the names of all customers who have a loan at the Perryridge branch.
  - Query 1

$$\Pi_{\text{customer-name}}(\sigma_{\text{branch-name = “Perryridge”}}\ (\sigma_{\text{borrower.loan-number = loan.loan-number}}(\text{borrower x loan})))$$

  - Query 2

$$\Pi_{\text{customer-name}}(\sigma_{\text{loan.loan-number = borrower.loan-number}}(\ (\sigma_{\text{branch-name = “Perryridge”}}(\text{loan})) \text{ x borrower}))$$

Example Queries

Find the largest account balance

- Rename *account* relation as *d*
- 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:
- $r \cap s = \{ t \mid t \in r$ **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:

r

A	B
α	1
α	2
β	1

s

A	B
α	2
β	3

- $r \cap s$

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:

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

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	$\in$

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

$$r \div s$$

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

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

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

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

Division Operation – Example

Relations *r*, *s*:

r

A	B
α	1
α	2
α	3
β	1
γ	1
δ	1
δ	3
δ	4
$\in$	6
$\in$	1
β	2

s

B
1
2

$r \div s$:

A
α
β

Another Division Example

Relations *r*, *s*:

r

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

s

D	E
a	1
b	1

$r \div s$:

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 $S \subseteq R$

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

To see why

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

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

Assignment Operation

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

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

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

  $result = temp1 - temp2$

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

Example Queries

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

Query 1

$$\Pi_{CN}(\sigma_{BN=\text{“Downtown”}}(depositor \bowtie account)) \cap$$

$$\Pi_{CN}(\sigma_{BN=\text{“Uptown”}}(depositor \bowtie account))$$

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

Query 2

$$\Pi_{customer\text{-}name,\ branch\text{-}name}\ (depositor \bowtie account)$$
$$\div\ \rho_{temp(branch\text{-}name)}\ (\{(\text{“Downtown”}),\ (\text{“Uptown”})\})$$

Example Queries

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

$$\Pi_{customer\text{-}name,\ branch\text{-}name}\ (depositor \bowtie account)$$
$$\div\ \Pi_{branch\text{-}name}\ (\sigma_{branch\text{-}city\ =\ \text{“Brooklyn”}}\ (branch))$$

Extended Relational-Algebra-Operations

- Generalized Projection
- Outer Join
- Aggregate Functions

Generalized Projection

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

$$\Pi_{F1, F2, \ldots, Fn}(E)$$

- E is any relational-algebra expression
- Each of F_1, F_2, …, 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:

$$\Pi_{customer\text{-}name,\ limit\ -\ credit\text{-}balance}\ (credit\text{-}info)$$

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

$$_{G1,\ G2,\ \ldots,\ Gn}\ \mathcal{G}_{\ F1(\ A1),\ F2(\ A2),\ldots,\ Fn(\ An)}\ (E)$$

  - E is any relational-algebra expression
  - $G_1, G_2 \ldots, G_n$ 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

$\mathcal{G}_{\mathbf{sum(c)}}\ (\mathrm{r})$

sum-C
27

Aggregate Operation – Example

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

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

$$_{branch\text{-}name}\mathcal{G}_{sum(balance)}\ (account)$$

branch-name	*balance*
Perryridge	1300
Brighton	1500
Redwood	700

Aggregate Functions (Cont.)

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

$$_{branch\text{-}name}\mathcal{G}_{\mathbf{sum}(balance)\ \mathbf{as}\ sum\text{-}balance}\ (account)$$

Outer Join

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

Outer Join – Example

- Relation *loan*

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

- Relation *borrower*

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

Outer Join – Example

- **Inner Join**

loan ⋈ *Borrower*

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

- **Left Outer Join**

loan ⟕ *Borrower*

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

Outer Join – Example

- **Right Outer Join**

loan ⟖ *borrower*

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

- **Full Outer Join**

loan ⟗ *borrower*

loan-number	*branch-name*	*amount*	*customer-name*
L-170	Downtown	3000	Jones
L-230	Redwood	4000	Smith
L-260	Perryridge	1700	*null*
L-155	*null*	*null*	Hayes

Null Values

- It is possible for tuples to have a null value, denoted by *null*, for some of their attributes
- *null* signifies an unknown value or that a value does not exist.
- The result of any arithmetic expression involving *null* is *null.*
- Aggregate functions simply ignore null values
  - Is an arbitrary decision. Could have returned null as result instead.
  - We follow the semantics of SQL in its handling of null values
- For duplicate elimination and grouping, null is treated like any other value, and two nulls are assumed to be the same
  - Alternative: assume each null is different from each other
  - Both are arbitrary decisions, so we simply follow SQL

Null Values

- Comparisons with null values return the special truth value *unknown*
  - If *false* was used instead of *unknown*, then *not (A < 5)* would not be equivalent to *A >= 5*
- 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*
  - 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.

$$account \leftarrow account - \sigma_{\ branch\text{-}name\ =\ \text{“Perryridge”}}\ (account)$$

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

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

- Delete all accounts at branches located in Needham.

$$r_1 \leftarrow \sigma_{\ branch\text{-}city\ =\ \text{“Needham”}}\ (account \bowtie branch)$$
$$r_2 \leftarrow \Pi_{branch\text{-}name,\ account\text{-}number,\ balance}\ (r_1)$$
$$r_3 \leftarrow \Pi_{\ customer\text{-}name,\ account\text{-}number}\ (r_2 \bowtie \text{depositor})$$
$$account \leftarrow \text{account} - r_2$$
$$depositor \leftarrow \text{depositor} - r_3$$

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.

  account ← *account* ∪ {("Perryridge", A-973, 1200)}

  depositor ← *depositor* ∪ {("Smith", A-973)}

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

  $r_1 \leftarrow (\sigma_{branch\text{-}name\ =\ \text{"Perryridge"}}\ (borrower \bowtie loan))$

  $account \leftarrow account \cup \Pi_{branch\text{-}name,\ account\text{-}number,200}\ (r_1)$

  $depositor \leftarrow depositor \cup \Pi_{customer\text{-}name,\ loan\text{-}number}(r_1)$

Updating

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

$$r \leftarrow \Pi_{F1,\ F2,\ \ldots,\ Fl,}\ (r)$$

- 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.

  $account \leftarrow \Pi_{AN,\ BN,\ BAL\ *\ 1.05}\,(account)$

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

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

$$account \leftarrow \Pi_{AN,\ BN,\ BAL\ *\ 1.06}\,(\sigma_{BAL\ >\ 10000}\,(account)) \cup \Pi_{AN,\ BN,\ BAL\ *\ 1.05}\,(\sigma_{BAL\ \leq\ 10000}\,(account))$$

Views

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

  $\Pi_{customer\text{-}name,\ loan\text{-}number}\,(borrower \bowtie loan)$

- 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 relational algebra query expression. The view name is represented by *v.*
- Once a view is defined, the view name can be used to refer to the virtual relation that the view generates.
- View definition is not the same as creating a new relation by evaluating the query expression
  - Rather, a view definition causes the saving of an expression; the expression is substituted into queries using the view.

View Examples

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

  create view *all-customer* **as**

  $$\Pi_{branch\text{-}name,\ customer\text{-}name}\ (depositor \bowtie account)$$
  $$\cup\ \Pi_{branch\text{-}name,\ customer\text{-}name}\ (borrower \bowtie loan)$$

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

  $$\Pi_{customer\text{-}name}$$
  $$(\sigma_{branch\text{-}name = \text{“Perryridge”}}\ (all\text{-}customer))$$

Updates Through View

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

  create view *branch-loan* **as**

  $\Pi_{\textit{branch-name, loan-number}}\,(\textit{loan})$
- Since we allow a view name to appear wherever a relation name is allowed, the person may write:

  $\textit{branch-loan} \leftarrow \textit{branch-loan} \cup \{(\text{“Perryridge”, L-37})\}$

Updates Through Views (Cont.)

- The previous insertion must be represented by an insertion into the actual relation *loan* from which the view *branch-loan* is constructed.
- An insertion into *loan* requires a value for *amount*. The insertion can be dealt with by either.
  - rejecting the insertion and returning an error message to the user.
  - inserting a tuple (“L-37”, “Perryridge”, *null*) into the *loan* relation
- Some updates through views are impossible to translate into database relation updates
  - create view v as $\sigma_{\textit{branch-name} = \text{“Perryridge”}}\,(\textit{account}))$

    $v \leftarrow v \cup (\text{L-99, Downtown, 23})$
- Others cannot be translated uniquely
  - $\textit{all-customer} \leftarrow \textit{all-customer} \cup \{(\text{“Perryridge”, “John”})\}$
    - Have to choose loan or account, and create a new loan/account number!

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

Tuple Relational Calculus

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

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

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

Predicate Calculus Formula

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

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

5. Set of quantifiers:
   - $\exists\, t \in r\, (Q(t)) \equiv$ "there exists" a tuple in *t* in relation *r* such that predicate *Q(t)* is true
   - $\forall t \in r\, (Q(t)) \equiv Q$ is true "for all" tuples *t* in relation *r*

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 the *loan-number, branch-name,* and *amount* for loans of over $1200

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

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

$$\{t \mid \exists\, s \in \text{loan}\ (t[loan\text{-}number] = s[loan\text{-}number] \wedge s\,[amount] > 1200)\}$$

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

Example Queries

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

$$\{t \mid \exists s \in borrower(\ t[customer\text{-}name] = s[customer\text{-}name]) \\ \lor \exists u \in depositor(\ t[customer\text{-}name] = u[customer\text{-}name])$$

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

$$\{t \mid \exists s \in borrower(\ t[customer\text{-}name] = s[customer\text{-}name]) \\ \land \exists u \in depositor(\ t[customer\text{-}name] = u[customer\text{-}name])$$

Example Queries

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

$$\{t \mid \exists s \in borrower(t[customer\text{-}name] = s[customer\text{-}name] \\ \land \exists u \in loan(u[branch\text{-}name] = \text{“Perryridge”} \\ \land\ u[loan\text{-}number] = s[loan\text{-}number]))\}$$

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

$$\{t \mid \exists s \in borrower(\ t[customer\text{-}name] = s[\text{customer-name}] \\ \land \exists u \in loan(u[branch\text{-}name] = \text{“Perryridge”} \\ \land\ u[loan\text{-}number] = s[\text{loan-}number])) \\ \land \textbf{not } \exists v \in depositor\ (v[customer\text{-}name] = \\ t[\text{customer-name}])\ \}$$

Example Queries

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

{*t* | ∃s ∈ *loan*(*s*[*branch-name*] = "Perryridge"
∧ ∃*u* ∈ *borrower* (*u*[*loan-number*] = *s*[*loan-number*]
∧ *t* [*customer-name*] = *u*[*customer-name*])
∧ ∃ *v* ∈ *customer* (*u*[*customer-name*] = *v*[*customer-name*]
∧ *t*[*customer-city*] = *v*[*customer-city*])))}

Example Queries

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

{*t* | ∃ c ∈ customer (*t*[customer.name] = c[customer-name]) ∧
∀ *s* ∈ *branch*(*s*[*branch-city*] = "Brooklyn" ⇒
∃ *u* ∈ *account* (s[*branch-name*] = *u*[branch-name]
∧ ∃ *s* ∈ *depositor* (*t*[*customer-name*] = *s*[customer-name]
∧ s[*account-number*] = *u*[account-number])))}

Safety of Expressions

- It is possible to write tuple calculus expressions that generate infinite relations.
- For example, $\{t \mid \neg\, t \in r\}$ results in an infinite relation if the domain of any attribute of relation *r* is infinite
- To guard against the problem, we restrict the set of allowable expressions to safe expressions.
- An expression $\{t \mid P(t)\}$ in the tuple relational calculus is *safe* if every component of *t* appears in one of the relations, tuples, or constants that appear in *P*
  - NOTE: this is more than just a syntax condition.
    - E.g. $\{ t \mid t[A]=5 \vee$ **true** $\}$ is not safe --- it defines an infinite set with attribute values that do not appear in any relation or tuples or constants in *P*.

Domain Relational Calculus

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

$$\{ < x_1, x_2, \ldots, x_n > \mid P(x_1, x_2, \ldots, x_n)\}$$

  - $x_1, x_2, \ldots, x_n$ represent domain variables
  - *P* represents a formula similar to that of the predicate calculus

Example Queries

- Find the *loan-number, branch-name,* and *amount* for loans of over $1200

$$\{< l, b, a > \mid < l, b, a > \in loan \wedge a > 1200\}$$

- Find the names of all customers who have a loan of over $1200

$$\{< c > \mid \exists\ l, b, a\ (< c, l > \in borrower \wedge < l, b, a > \in loan \wedge a > 1200)\}$$

- Find the names of all customers who have a loan from the Perryridge branch and the loan amount:

$$\{< c, a > \mid \exists\ l\ (< c, l > \in borrower \wedge \exists b(< l, b, a > \in loan \wedge b = \text{“Perryridge”}))\}$$

or $\{< c, a > \mid \exists\ l\ (< c, l > \in borrower \wedge < l, \text{“Perryridge”}, a > \in loan)\}$

Example Queries

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

$$\begin{aligned}\{< c > \mid \exists\ l\ (\{< c, l > \in borrower\\ &\wedge \exists\ b,a(< l, b, a > \in loan \wedge b = \text{“Perryridge”}))\\ \vee \exists\ a(< c, a > \in depositor\\ &\wedge \exists\ b,n(< a, b, n > \in account \wedge b = \text{“Perryridge”}))\}\end{aligned}$$

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

$$\begin{aligned}\{< c > \mid \exists\ s, n\ (< c, s, n > \in \text{customer}) \wedge\\ \forall\ x,y,z(< x, y, z > \in branch \wedge y = \text{“Brooklyn”}) \Rightarrow\\ \exists\ a,b(< x, y, z > \in account \wedge < c,a > \in depositor)\}\end{aligned}$$

Safety of Expressions

$$\{ < x_1, x_2, \ldots, x_n > \mid P(x_1, x_2, \ldots, x_n)\}$$

is safe if all of the following hold:

1. All values that appear in tuples of the expression are values from *dom*(P) (that is, the values appear either in P or in a tuple of a relation mentioned in P).
2. For every "there exists" subformula of the form $\exists\, x\, (P_1(x))$, the subformula is true if and only if there is a value of x in *dom*(P_1) such that $P_1(x)$ is true.
3. For every "for all" subformula of the form $\forall_x\, (P_1\, (x))$, the subformula is true if and only if $P_1(x)$ is true for all values x from *dom* (P_1).

End of Chapter 3

Result of $\sigma_{\textit{branch-name} = \text{“Perryridge”}}$ (*loan*)

loan-number	*branch-name*	*amount*
L-15	Perryridge	1500
L-16	Perryridge	1300

Loan Number and the Amount of the Loan

loan-number	*amount*
L-11	900
L-14	1500
L-15	1500
L-16	1300
L-17	1000
L-23	2000
L-93	500

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

customer-name
Adams
Curry
Hayes
Jackson
Jones
Smith
Williams
Lindsay
Johnson
Turner

Customers With An Account But No Loan

customer-name
Johnson
Lindsay
Turner

Result of *borrower* × *loan*

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

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

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

Result of $\Pi_{customer\text{-}name}$

customer-name
Adams
Hayes

Result of the Subexpression

balance
500
400
700
750
350

Largest Account Balance in the Bank

balance
900

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

customer-name
Curry
Smith

Customers With Both an Account and a Loan at the Bank

customer-name
Hayes
Jones
Smith

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

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

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

branch-name
Brighton
Perryridge

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

branch-name
Brighton
Downtown

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

customer-name	*branch-name*
Hayes	Perryridge
Johnson	Downtown
Johnson	Brighton
Jones	Brighton
Lindsay	Redwood
Smith	Mianus
Turner	Round Hill

The *credit-info* Relation

customer-name	*branch-name*
Hayes	Perryridge
Johnson	Downtown
Johnson	Brighton
Jones	Brighton
Lindsay	Redwood
Smith	Mianus
Turner	Round Hill

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

customer-name	*credit-available*
Curry	250
Jones	5300
Smith	1600
Hayes	0

The *pt-works* Relation

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

The *pt-works* Relation After Grouping

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

Result of $_{branch\text{-}name}\mathcal{G}_{sum(salary)}$ *(pt-works)*

branch-name	*sum of salary*
Austin	3100
Downtown	5300
Perryridge	8100

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

branch-name	*sum-salary*	*max-salary*
Austin	3100	1600
Downtown	5300	2500
Perryridge	8100	5300

The *employee* and *ft-works* Relations

employee-name	*street*	*city*
Coyote	Toon	Hollywood
Rabbit	Tunnel	Carrotville
Smith	Revolver	Death Valley
Williams	Seaview	Seattle

employee-name	*branch-name*	*salary*
Coyote	Mesa	1500
Rabbit	Mesa	1300
Gates	Redmond	5300
Williams	Redmond	1500

The Result of *employee* ⋈ *ft-works*

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

The Result of *employee* ⟕ *ft-works*

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

Result of *employee* ⟖ *ft-works*

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

Result of *employee* ⟗ *ft-works*

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

Tuples Inserted Into *loan* and *borrower*

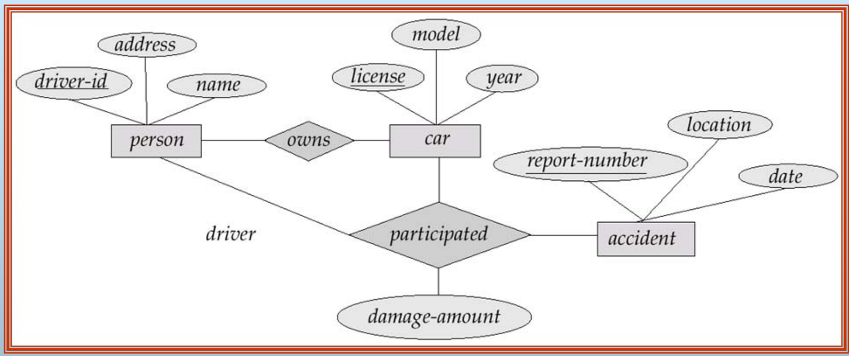
loan-number	*branch-name*	*amount*
L-11	Round Hill	900
L-14	Downtown	1500
L-15	Perryridge	1500
L-16	Perryridge	1300
L-17	Downtown	1000
L-23	Redwood	2000
L-93	Mianus	500
null	*null*	1900

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
Johnson	*null*

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

customer-name
Adams
Hayes

E-R Diagram



The *branch* Relation

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

The *loan* Relation

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

The *borrower* Relation

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

Chapter 7: SQL

- Basic Structure
- Simple Queries
- Nested Subqueries
- Aggregate Functions
- Set Operations
- With Clause
- Views
- Modification of the Database
- Joined Relations
- Data Definition Language
- Embedded SQL, ODBC and JDBC

Basic Structure

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

  select $A_1, A_2, ..., A_n$
  from $r_1, r_2, ..., r_m$
  where *predicate*

  - A_is represent attributes
  - r_is represent relations (tables)
  - *predicate* is any predicate.
- This query is equivalent to the relational algebra expression.

  $$\Pi_{A1,\ A2,\ ...,\ An}(\sigma_P (r_1 \times r_2 \times ... \times r_m))$$
- The result of an SQL query is a relation.
- NOTE: SQL does not permit the '-' character in names. SQL names are case insensitive, i.e. you can use capital or small letters.

Schema Used in Examples

S#	Sname	Status	City
S1	Smith	20	London
S2	Jones	10	Paris
S3	Blake	30	Paris
S4	Clark	20	London
S5	Adams	30	Athens

S#	P#	QTY			
S1	P1	300	S2	P1	300
S1	P2	200	S2	P2	400
S1	P3	400	S3	P2	200
S1	P4	200	S4	P2	200
S1	P5	100	S4	P4	300
S1	P6	100	S4	P5	400

P#	Pname	Color	Weight	City
P1	Nut	Red	12	London
P2	Bolt	Green	17	Paris
P3	Screw	Blue	17	Rome
P4	Screw	Red	14	London
P5	Cam	Blue	12	Paris
P6	Cog	Red	19	London

Suppliers
S (S#, Sname, Status, City)

Shipments
SP (S#, P#, QTY)

Parts
P (P#, Pname, Color, Weight, City)

Simple Queries (1)

S#	Sname	Status	City
S1	Smith	20	London
S2	Jones	10	Paris
S3	Blake	30	Paris
S4	Clark	20	London
S5	Adams	30	Athens

P#	Pname	Color	Weight	City
P1	Nut	Red	12	London
P2	Bolt	Green	17	Paris
P3	Screw	Blue	17	Rome
P4	Screw	Red	14	London
P5	Cam	Blue	12	Paris
P6	Cog	Red	19	London

S#	P#	QTY			
S1	P1	300	S2	P1	300
S1	P2	200	S2	P2	400
S1	P3	400	S3	P2	200
S1	P4	200	S4	P2	200
S1	P5	100	S4	P4	300
S1	P6	100	S4	P5	400

Get part numbers for all parts supplied.

select *P#*
from *SP* ;

Result:

P#
P1
P2
P3
P4
P5
P6
P1
P2
P2
P2
P4
P5

Get part numbers for all parts supplied (no duplicates).

select distinct *P#*
from *SP* ;

Result:

P#
P1
P2
P3
P4
P5
P6

Get supplier numbers from Paris with Status above 20.

select *S#*
from *S*
where *City* = 'Paris' **and** *Status* > 25;

Result:

S#
S3

Simple Queries (2)

S#	Sname	Status	City
S1	Smith	20	London
S2	Jones	10	Paris
S3	Blake	30	Paris
S4	Clark	20	London
S5	Adams	30	Athens

P#	Pname	Color	Weight	City
P1	Nut	Red	12	London
P2	Bolt	Green	17	Paris
P3	Screw	Blue	17	Rome
P4	Screw	Red	14	London
P5	Cam	Blue	12	Paris
P6	Cog	Red	19	London

S#	P#	QTY			
S1	P1	300	S2	P1	300
S1	P2	200	S2	P2	400
S1	P3	400	S3	P2	200
S1	P4	200	S4	P2	200
S1	P5	100	S4	P4	300
S1	P6	100	S4	P5	400

Get supplier numbers and status for suppliers in Paris in desceding order of status.

select *S#, Status*
from *S*
where *City* = 'Paris'
order by *Status* **desc** ;

Result:

S#	Status
S3	30
S2	10

For all blue parts, get the weights in grams.

select *P#, Weight*454*
from *P*
where *Color* = 'Blue'
order by *2, P#* ;

Result:

P#	Weight
P5	5448
P3	7718

Include constatnt in select clause.

select *P#*, 'Weights in grams = ', *Weight*454*
from *P*
where *Color* = 'Blue' ;

Result:

P#	
P3	Weights in grams = 7718
P5	Weights in grams = 5448

Simple Queries (between)

S#	Sname	Status	City
S1	Smith	20	London
S2	Jones	10	Paris
S3	Blake	30	Paris
S4	Clark	20	London
S5	Adams	30	Athens

P#	Pname	Color	Weight	City
P1	Nut	Red	12	London
P2	Bolt	Green	17	Paris
P3	Screw	Blue	17	Rome
P4	Screw	Red	14	London
P5	Cam	Blue	12	Paris
P6	Cog	Red	19	London

S#	P#	QTY			
S1	P1	300	S2	P1	300
S1	P2	200	S2	P2	400
S1	P3	400	S3	P2	200
S1	P4	200	S4	P2	200
S1	P5	100	S4	P4	300
S1	P6	100	S4	P5	400

Get parts whose weight is in range 16 to 19 (inclusive).

select *
from *P*
where *Weight* **between** 16 **and** 19 ;

Result:

P#	Pname	Color	Weight	City
P2	Bolt	Green	17	Paris
P3	Screw	Blue	17	Rome
P6	Cog	Red	19	London

Get parts whose weight is not in range 16 to 19.

select *P#, Pname, Color, Weight, City*
from *P*
where *Weight* **not between** 16 **and** 19 ;

Result:

P#	Pname	Color	Weight	City
P1	Nut	Red	12	London
P4	Screw	Red	14	London
P5	Cam	Blue	12	Paris

Simple Queries (in)

S#	Sname	Status	City
S1	Smith	20	London
S2	Jones	10	Paris
S3	Blake	30	Paris
S4	Clark	20	London
S5	Adams	30	Athens

P#	Pname	Color	Weight	City
P1	Nut	Red	12	London
P2	Bolt	Green	17	Paris
P3	Screw	Blue	17	Rome
P4	Screw	Red	14	London
P5	Cam	Blue	12	Paris
P6	Cog	Red	19	London

S#	P#	QTY			
S1	P1	300	S2	P1	300
S1	P2	200	S2	P2	400
S1	P3	400	S3	P2	200
S1	P4	200	S4	P2	200
S1	P5	100	S4	P4	300
S1	P6	100	S4	P5	400

Get parts whose weight is in range 16 to 19 (inclusive).

select *
from *P*
where *Weight* **in** {12, 16, 17} ;

Result:

P#	Pname	Color	Weight	City
P1	Nut	Red	12	London
P2	Bolt	Green	17	Paris
P3	Screw	Blue	17	Rome
P5	Cam	Blue	12	Paris

Get parts whose weight is not in range 16 to 19.

select *P#, Pname, Color, Weight, City*
from *P*
where *Weight* **not in** {12, 16, 17} ;

Result:

P#	Pname	Color	Weight	City
P4	Screw	Red	14	London
P6	Cog	Red	19	London

Simple Queries (like)

S#	Sname	Status	City
S1	Smith	20	London
S2	Jones	10	Paris
S3	Blake	30	Paris
S4	Clark	20	London
S5	Adams	30	Athens

P#	Pname	Color	Weight	City
P1	Nut	Red	12	London
P2	Bolt	Green	17	Paris
P3	Screw	Blue	17	Rome
P4	Screw	Red	14	London
P5	Cam	Blue	12	Paris
P6	Cog	Red	19	London

S#	P#	QTY			
S1	P1	300	S2	P1	300
S1	P2	200	S2	P2	400
S1	P3	400	S3	P2	200
S1	P4	200	S4	P2	200
S1	P5	100	S4	P4	300
S1	P6	100	S4	P5	400

Get parts whose names begin with the letter C.

select *
from *P*
where *Pname* **like** ‘C*’ ;

Result:

P#	Pname	Color	Weight	City
P5	Cam	Blue	12	Paris
P6	Cog	Red	19	London

% *stands for any string,* ? *stands for any character*

Sname **like** ‘?la*’ – all supplier names with second character l and third characer a.

Pname **like** ‘????’ – all part names 4 character long.

City **not like** ‘*o*’ – all city names which does not contain characer o.

like ‘Main*’ **escape** ‘\’ – match Main*

SQL supports a variety of string operations such as: concatenation (“||”), converting from upper to lower case (and vice versa), finding string length, extracting substrings, etc.

Simple Queries (null values)

S#	Sname	Status	City
S1	Smith	20	London
S2	Jones	10	Paris
S3	Blake	30	Paris
S4	Clark	20	London
S5	Adams	30	Athens

P#	Pname	Color	Weight	City
P1	Nut	Red	12	London
P2	Bolt	Green	17	Paris
P3	Screw	Blue	17	Rome
P4	Screw	Red	14	London
P5	Cam	Blue	12	Paris
P6	Cog	Red	19	London

S#	P#	QTY			
S1	P1	300	S2	P1	300
S1	P2	200	S2	P2	400
S1	P3	400	S3	P2	200
S1	P4	200	S4	P2	200
S1	P5	100	S4	P4	300
S1	P6	100	S4	P5	400

Get parts whose color is not null.

select *
from *P*
where *Color* **is not null** ;

Result:

P#	Pname	Color	Weight	City
P1	Nut	Red	12	London
P2	Bolt	Green	17	Paris
P3	Screw	Blue	17	Rome
P4	Screw	Red	14	London
P5	Cam	Blue	12	Paris
P6	Cog	Red	19	London

null signifies an unknown value or that a value does not exist.
The result of any arithmetic expression involving *null* is *null* (E.g. 5 + null returns null).

Any comparison with *null* returns *unknown* (E.g. 5 < *null* or *null* <> *null* or *null* = *null*).

Simple Queries (natural join)

S#	Sname	Status	City
S1	Smith	20	London
S2	Jones	10	Paris
S3	Blake	30	Paris
S4	Clark	20	London
S5	Adams	30	Athens

P#	Pname	Color	Weight	City
P1	Nut	Red	12	London
P2	Bolt	Green	17	Paris
P3	Screw	Blue	17	Rome
P4	Screw	Red	14	London
P5	Cam	Blue	12	Paris
P6	Cog	Red	19	London

S#	P#	QTY			
S1	P1	300	S2	P1	300
S1	P2	200	S2	P2	400
S1	P3	400	S3	P2	200
S1	P4	200	S4	P2	200
S1	P5	100	S4	P4	300
S1	P6	100	S4	P5	400

Get all combination suppliers - parts located in the same city.

select *S*.*, *P*.*
from *S*, *P*
where *S*.*City* = *P*.*City* ;

Result:

S#	Sname	Status	S.City	P#	Pname	Color	Weight	P.City
S1	Smith	20	London	P1	Nut	Red	12	London
S1	Smith	20	London	P4	Screw	Red	14	London
S1	Smith	20	London	P6	Cog	Red	19	London
S2	Jones	10	Paris	P2	Bolt	Green	17	Paris
S2	Jones	10	Paris	P5	Cam	Blue	12	Paris
S3	Blake	30	Paris	P2	Bolt	Green	17	Paris
S3	Blake	30	Paris	P5	Cam	Blue	12	Paris
S4	Clark	20	London	P5	Nut	Red	12	London
S4	Clark	20	London	P5	Screw	Red	14	London
S4	Clark	20	London	P5	Cog	Red	19	London

How conceptualy join is constructed:

- Form the *cartesian* product of the tables listed in **from** clause (in our example the new table will have 5·6 = 30 rows)
- Eliminate from the cartesian product all those rows that do not satisfy join predicate (**where** clause)

Simple Queries (natural join)

S#	Sname	Status	City
S1	Smith	20	London
S2	Jones	10	Paris
S3	Blake	30	Paris
S4	Clark	20	London
S5	Adams	30	Athens

P#	Pname	Color	Weight	City
P1	Nut	Red	12	London
P2	Bolt	Green	17	Paris
P3	Screw	Blue	17	Rome
P4	Screw	Red	14	London
P5	Cam	Blue	12	Paris
P6	Cog	Red	19	London

S#	P#	QTY			
S1	P1	300	S2	P1	300
S1	P2	200	S2	P2	400
S1	P3	400	S3	P2	200
S1	P4	200	S4	P2	200
S1	P5	100	S4	P4	300
S1	P6	100	S4	P5	400

Same, but suplier city follows part city (alphabetically).

select *S.∗, P.∗*
from *S, P*
where *S.City > P.City* ;

Result:

S#	Sname	Status	S.City	P#	Pname	Color	Weight	P.City
S2	Jones	10	Paris	P1	Nut	Red	12	London
S2	Jones	10	Paris	P4	Screw	Red	14	London
S2	Jones	10	Paris	P6	Cog	Red	19	London
S3	Blake	30	Paris	P1	Nut	Red	12	London
S3	Blake	30	Paris	P4	Screw	Red	14	London
S3	Blake	30	Paris	P6	Cog	Red	19	London

Get all combination suppliers - parts located in the same city, without suppliers that have status 20.

select *S.∗, P.∗*
from *S, P*
where *S.City = P.City* **and** *S.Status* <> 20 ;

Result:

S#	Sname	Status	S.City	P#	Pname	Color	Weight	P.City
S2	Jones	10	Paris	P2	Bolt	Green	17	Paris
S2	Jones	10	Paris	P5	Cam	Blue	12	Paris
S3	Blake	30	Paris	P2	Bolt	Green	17	Paris
S3	Blake	30	Paris	P5	Cam	Blue	12	Paris

Simple Queries (natural join)

S#	Sname	Status	City
S1	Smith	20	London
S2	Jones	10	Paris
S3	Blake	30	Paris
S4	Clark	20	London
S5	Adams	30	Athens

P#	Pname	Color	Weight	City
P1	Nut	Red	12	London
P2	Bolt	Green	17	Paris
P3	Screw	Blue	17	Rome
P4	Screw	Red	14	London
P5	Cam	Blue	12	Paris
P6	Cog	Red	19	London

S#	P#	QTY			
S1	P1	300	S2	P1	300
S1	P2	200	S2	P2	400
S1	P3	400	S3	P2	200
S1	P4	200	S4	P2	200
S1	P5	100	S4	P4	300
S1	P6	100	S4	P5	400

Get all pairs of city names such that a supplier located in the first city supplies a part stored in the second city.

For example, supplier S1 supplies part P1; suppliers S1 is located in London, and part P1 is stored in London; so 'London, London' is a pair of cities in the result.

select distinct *S.City, P.City*
from *S, SP, P*
where *S.S# = SP.S#* **and** *SP.P# = P.P#* ;

Result:

S.City	P.City
London	London
London	Paris
London	Rome
Paris	London
Paris	Paris

This example shows join of 3 tables.

Simple Queries (join a table with itself)

S#	Sname	Status	City
S1	Smith	20	London
S2	Jones	10	Paris
S3	Blake	30	Paris
S4	Clark	20	London
S5	Adams	30	Athens

P#	Pname	Color	Weight	City
P1	Nut	Red	12	London
P2	Bolt	Green	17	Paris
P3	Screw	Blue	17	Rome
P4	Screw	Red	14	London
P5	Cam	Blue	12	Paris
P6	Cog	Red	19	London

S#	P#	QTY			
S1	P1	300	S2	P1	300
S1	P2	200	S2	P2	400
S1	P3	400	S3	P2	200
S1	P4	200	S4	P2	200
S1	P5	100	S4	P4	300
S1	P6	100	S4	P5	400

Get all pairs of supplier numbers such that the two suppliers are co-located.

select *Sup1.S#, Sup2.S#*
from *S* **as** *Sup1, S* **as** *Sup2*
where *Sup1.City* = *Sup2.City* ;

Result:

S#	S#
S1	S1
S1	S4
S2	S2
S2	S3
S3	S2
S3	S3
S4	S1
S4	S4
S5	S5

This result can be cleared up as follows:

select *Sup1.S#, Sup2.S#*
from *S* **as** *Sup1, S* **as** *Sup2*
where *Sup1.City* = *Sup2.City* **and** *Sup1.S#* > *Sup2.S#* ;

Result:

S#	S#
S1	S4
S2	S3

SubQueries

S#	Sname	Status	City
S1	Smith	20	London
S2	Jones	10	Paris
S3	Blake	30	Paris
S4	Clark	20	London
S5	Adams	30	Athens

P#	Pname	Color	Weight	City
P1	Nut	Red	12	London
P2	Bolt	Green	17	Paris
P3	Screw	Blue	17	Rome
P4	Screw	Red	14	London
P5	Cam	Blue	12	Paris
P6	Cog	Red	19	London

S#	P#	QTY			
S1	P1	300	S2	P1	300
S1	P2	200	S2	P2	400
S1	P3	400	S3	P2	200
S1	P4	200	S4	P2	200
S1	P5	100	S4	P4	300
S1	P6	100	S4	P5	400

Get suppliers names for suppliers who supplies part P2.

select *S.Sname*
from *S*
where *S.S#* **in** (**select** *SP.S#*
from *SP*
where *SP.P#* = 'P2') ;

Result:

Sname
Smith
Jones
Blake
Clark

The nested subqueries are evaluated first. So, our query is equivalent to:
select *S.Sname*
from *S*
where *S.S#* **in** ('S1', 'S2', 'S3', 'S4') ;

The same using join.

select *S.Sname*
from *S, SP*
where *S.S#* = *SP.S#* **and** *SP.P#* = 'P2' ;

The join of S and SP over supplier numbers is a table of 12 rows from which we select those 4 rows that have the part number P2.

SubQueries (correlated)

S#	Sname	Status	City
S1	Smith	20	London
S2	Jones	10	Paris
S3	Blake	30	Paris
S4	Clark	20	London
S5	Adams	30	Athens

P#	Pname	Color	Weight	City
P1	Nut	Red	12	London
P2	Bolt	Green	17	Paris
P3	Screw	Blue	17	Rome
P4	Screw	Red	14	London
P5	Cam	Blue	12	Paris
P6	Cog	Red	19	London

S#	P#	QTY			
S1	P1	300	S2	P1	300
S1	P2	200	S2	P2	400
S1	P3	400	S3	P2	200
S1	P4	200	S4	P2	200
S1	P5	100	S4	P4	300
S1	P6	100	S4	P5	400

Get suppliers names for suppliers who supplies part P2.

select *Sname*
from *S*
where ‘P2’ **in** (**select** *P#*
from *SP*
where *S#* = *S.S#*) ;

Result:

Sname
Smith
Jones
Blake
Clark

In the last line the unqualified reference *S#* is implicitl qualified by *SP*. Here, inner subquery cannot be evaluated once and for all before the outher query is evaluated (variable *S.S#* is uknown). Such subqueries are called *correlated*. The system examines one by one rows of table *S* and each time evaluate the subquery.

Some people prefer to use aliases in correlated subqueries.

select *SX.Sname*
from *S* **as** *SX*
where ‘P2’ **in** (**select** *P#*
from *SP*
where *S#* = *SX.S#*) ;

SubQueries (more nesting)

S#	Sname	Status	City
S1	Smith	20	London
S2	Jones	10	Paris
S3	Blake	30	Paris
S4	Clark	20	London
S5	Adams	30	Athens

P#	Pname	Color	Weight	City
P1	Nut	Red	12	London
P2	Bolt	Green	17	Paris
P3	Screw	Blue	17	Rome
P4	Screw	Red	14	London
P5	Cam	Blue	12	Paris
P6	Cog	Red	19	London

S#	P#	QTY			
S1	P1	300	S2	P1	300
S1	P2	200	S2	P2	400
S1	P3	400	S3	P2	200
S1	P4	200	S4	P2	200
S1	P5	100	S4	P4	300
S1	P6	100	S4	P5	400

Get suppliers names for suppliers who supplie at least one red part.

select *Sname*
from *S*
where *S#* **in** (**select** *S#*
from *SP*
where *P#* **in** (**select** *P#*
from *P*
where *Color* = ‘Red’));

Result:

Sname
Smith
Jones
Clark

The innermost subquery evaluates to the set {‘P1’, ‘P4’, ‘P6’}. The next subquery evaluates in turn to the set {‘S1’, ‘S2’, ‘S4’}. Last, the outermost select evaluates to the final result. In general, subqueries can be nested to any depth.

The same using join.

select distinct *S.Sname*
from *S, SP, P*
where *S.S#* = *SP.S#* **and** *SP.P#* = *P.P#*
and *P.Color* = ‘Red’ ;

SubQueries (with same table)

S#	Sname	Status	City
S1	Smith	20	London
S2	Jones	10	Paris
S3	Blake	30	Paris
S4	Clark	20	London
S5	Adams	30	Athens

P#	Pname	Color	Weight	City
P1	Nut	Red	12	London
P2	Bolt	Green	17	Paris
P3	Screw	Blue	17	Rome
P4	Screw	Red	14	London
P5	Cam	Blue	12	Paris
P6	Cog	Red	19	London

S#	P#	QTY
S1	P1	300
S1	P2	200
S1	P3	400
S1	P4	200
S1	P5	100
S1	P6	100
S2	P1	300
S2	P2	400
S3	P2	200
S4	P2	200
S4	P4	300
S4	P5	400

Get supplier numbers for suppliers who supply at least one part supplied by supplier S2.

select distinct *S#*
from *SP*
where *P#* **in** (**select** *P#*
from *SP*
where *S#* = 'S2');

Result:

S#
S1
S2
S3
S4

The reference *SP* in the subquery does not mean the same thing as reference to *SP* in the outher query. They are different variables. Using aliases will make this fact explicit.

The same using join.

select distinct *SP1.S#*
from *SP* **as** *SP1*, *SP* **as** *SP2*
where *SP1.P#* = *SP2.P#*
and *SP2.S#* = 'S2' ;

SubQueries (correlated with same table)

S#	Sname	Status	City
S1	Smith	20	London
S2	Jones	10	Paris
S3	Blake	30	Paris
S4	Clark	20	London
S5	Adams	30	Athens

P#	Pname	Color	Weight	City
P1	Nut	Red	12	London
P2	Bolt	Green	17	Paris
P3	Screw	Blue	17	Rome
P4	Screw	Red	14	London
P5	Cam	Blue	12	Paris
P6	Cog	Red	19	London

S#	P#	QTY
S1	P1	300
S1	P2	200
S1	P3	400
S1	P4	200
S1	P5	100
S1	P6	100
S2	P1	300
S2	P2	400
S3	P2	200
S4	P2	200
S4	P4	300
S4	P5	400

Get part numbers for all parts supplied by more than one supplier.

select distinct *SP1.P#*
from *SP* **as** *SP1*
where *SP1.P#* **in** (**select** *SP2.P#*
from *SP* **as** *SP2*
where SP2.*S#* = SP1.*S#*);

Result:

P#
P1
P2
P4
P5

Operation of this query: For each row in turn, *SP1* of table *SP*, extract the P# value, iff that P# value appears in some row *SP2* of table *SP* whose *S#* value is not that in row *SP1*. Note that at least one alias must be used, but not both.

Get supplier numbers for suppliers who are located in the same city as supplier S1.

select *S#*
from *S*
where *City* = (**select** *City*
from *S*
where *S#* = 'S1');

Result:

S#
S1
S4

SubQueries (exists)

S#	Sname	Status	City
S1	Smith	20	London
S2	Jones	10	Paris
S3	Blake	30	Paris
S4	Clark	20	London
S5	Adams	30	Athens

P#	Pname	Color	Weight	City
P1	Nut	Red	12	London
P2	Bolt	Green	17	Paris
P3	Screw	Blue	17	Rome
P4	Screw	Red	14	London
P5	Cam	Blue	12	Paris
P6	Cog	Red	19	London

S#	P#	QTY			
S1	P1	300	S2	P1	300
S1	P2	200	S2	P2	400
S1	P3	400	S3	P2	200
S1	P4	200	S4	P2	200
S1	P5	100	S4	P4	300
S1	P6	100	S4	P5	400

Get suppliers names for suppliers who supplies part P2.

```
select Sname
from S
where exists  ( select *
                from SP
                where S# = S.S# and P# = 'P2' );
```

Result:

Sname
Smith
Jones
Blake
Clark

Predicate **exists x (predicate-involving-x)** is true iff **predicate-involving-x** is true for some **x**. For example if **x**=1,2,…,10 then **exists x (x<5)** is true, while **exists x (x<0)** is false.

Get suppliers names for suppliers who do not supply part P2.

In general, **exists** is one of the most important SQL feature. In fact, any query expresssed using **in** can be formulated using **exists**. The converse is not true.

```
select Sname
from S
where not exists  ( select *
                    from SP
                    where S# = S.S# and P# = 'P2' );
```

Result:

Sname
Adams

SubQueries (not exists)

S#	Sname	Status	City
S1	Smith	20	London
S2	Jones	10	Paris
S3	Blake	30	Paris
S4	Clark	20	London
S5	Adams	30	Athens

P#	Pname	Color	Weight	City
P1	Nut	Red	12	London
P2	Bolt	Green	17	Paris
P3	Screw	Blue	17	Rome
P4	Screw	Red	14	London
P5	Cam	Blue	12	Paris
P6	Cog	Red	19	London

S#	P#	QTY			
S1	P1	300	S2	P1	300
S1	P2	200	S2	P2	400
S1	P3	400	S3	P2	200
S1	P4	200	S4	P2	200
S1	P5	100	S4	P4	300
S1	P6	100	S4	P5	400

Get supplier names for suppliers who supply all parts.

```
select Sname
from S
where not exists
        ( select *
          from P
          where not exists
                ( select *
                  from SP
                  where S# = S.S# and P# = p.pp );
```

Result:

Sname
Smith

The query can be paraphrased according to the above SQL statement: *Select supplier names for supplier such that there does not exists a part that they do not supply.*

SubQueries (all, some)

S#	Sname	Status	City
S1	Smith	20	London
S2	Jones	10	Paris
S3	Blake	30	Paris
S4	Clark	20	London
S5	Adams	30	Athens

P#	Pname	Color	Weight	City
P1	Nut	Red	12	London
P2	Bolt	Green	17	Paris
P3	Screw	Blue	17	Rome
P4	Screw	Red	14	London
P5	Cam	Blue	12	Paris
P6	Cog	Red	19	London

S#	P#	QTY			
S1	P1	300	S2	P1	300
S1	P2	200	S2	P2	400
S1	P3	400	S3	P2	200
S1	P4	200	S4	P2	200
S1	P5	100	S4	P4	300
S1	P6	100	S4	P5	400

Get the all part numbers that have greater shipment quantity than all parts located in London.

select *P#*
from *SP*
where *QTY* > **all**
(**select** *QTY*
from *SP, P*
where *City* = 'London') ;

Result:

P#
P3
P2
P5

Get the all part numbers that have greater shipment quantity than some part located in London.

select *P#*
from *SP*
where *QTY* > **some**
(**select** *QTY*
from *SP, P*
where *City* = 'London') ;

Result:

P#
P1
P2
P3
P4
P5

Definition of Some and All Clauses

$(5 < \textbf{some} \begin{array}{|c|}\hline 0\\\hline 5\\\hline 6\\\hline\end{array}) = \text{true}$ $\quad$ $(5 < \textbf{some} \begin{array}{|c|}\hline 0\\\hline 5\\\hline\end{array}) = \text{false}$

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

$(= \textbf{some}) \equiv \textbf{in}.$ However, $(\neq \textbf{some}) \not\equiv \textbf{not in}$

$(5 < \textbf{all} \begin{array}{|c|}\hline 0\\\hline 5\\\hline 6\\\hline\end{array}) = \text{false}$ $\quad$ $(5 < \textbf{all} \begin{array}{|c|}\hline 6\\\hline 10\\\hline\end{array}) = \text{true}$

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

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

Aggregate Functions (count, sum, max)

S#	Sname	Status	City
S1	Smith	20	London
S2	Jones	10	Paris
S3	Blake	30	Paris
S4	Clark	20	London
S5	Adams	30	Athens

P#	Pname	Color	Weight	City
P1	Nut	Red	12	London
P2	Bolt	Green	17	Paris
P3	Screw	Blue	17	Rome
P4	Screw	Red	14	London
P5	Cam	Blue	12	Paris
P6	Cog	Red	19	London

S#	P#	QTY			
S1	P1	300	S2	P1	300
S1	P2	200	S2	P2	400
S1	P3	400	S3	P2	200
S1	P4	200	S4	P2	200
S1	P5	100	S4	P4	300
S1	P6	100	S4	P5	400

Get the number of shipments for part P2.

select count(∗)
from *SP*
where *P#* **=** ‘P2’ ;

Result:

4

Get the total quantity of part P2 supplied.

select sum(*QTY*)
from *SP*
where *P#* **=** ‘P2’ ;

Result:

1000

Get supplier numbers for suppliers with status less then current maximum status.

select *S#*
from *S*
where *Status* **<**
(**select max**(*Status*)
from *S*) ;

Result:

S#
S1
S2
S4

Aggregate Functions (min, avg)

S#	Sname	Status	City
S1	Smith	20	London
S2	Jones	10	Paris
S3	Blake	30	Paris
S4	Clark	20	London
S5	Adams	30	Athens

P#	Pname	Color	Weight	City
P1	Nut	Red	12	London
P2	Bolt	Green	17	Paris
P3	Screw	Blue	17	Rome
P4	Screw	Red	14	London
P5	Cam	Blue	12	Paris
P6	Cog	Red	19	London

S#	P#	QTY			
S1	P1	300	S2	P1	300
S1	P2	200	S2	P2	400
S1	P3	400	S3	P2	200
S1	P4	200	S4	P2	200
S1	P5	100	S4	P4	300
S1	P6	100	S4	P5	400

Get the all part names for parts with minimum weights.

select *Pname*
from *P*
where *Weight* **=**
(**select min**(*Weight*)
from *P*) ;

Result:

Pname
Nut
Cam

Get supplier numbers, status nad city for all suppliers whose status is greater than or equal to the average for their city.

select *S#, Status, City*
from *S* **as** *S1*
where *Status* **>=**
(**select avg**(*Status*)
from *S* **as** *S2*
where S2.City = *S1.City*) ;

Result:

S#	Status	City
S1	20	London
S3	30	Paris
S4	30	London
S5	30	Athens

Aggregate Functions (group by)

S#	Sname	Status	City
S1	Smith	20	London
S2	Jones	10	Paris
S3	Blake	30	Paris
S4	Clark	20	London
S5	Adams	30	Athens

P#	Pname	Color	Weight	City
P1	Nut	Red	12	London
P2	Bolt	Green	17	Paris
P3	Screw	Blue	17	Rome
P4	Screw	Red	14	London
P5	Cam	Blue	12	Paris
P6	Cog	Red	19	London

S#	P#	QTY			
S1	P1	300	S2	P1	300
S1	P2	200	S2	P2	400
S1	P3	400	S3	P2	200
S1	P4	200	S4	P2	200
S1	P5	100	S4	P4	300
S1	P6	100	S4	P5	400

Get the total quantity supplied for each part.

select *P#*, **sum**(*QTY*)
from *SP*
group by *P#* ;

Result:

P#	
P1	600
P2	1000
P3	400
P4	500
P5	500
P6	100

For each part supplied, get the part number and the total quantity supplied of that part, excluding shipment from supplier S1.

select *P#*, **sum**(*QTY*)
from *SP*
where *S#* **<>** 'S1'
group by *P#* ;

Result:

P#	
P1	300
P2	800
P4	300
P5	400

Aggregate Functions (having)

S#	Sname	Status	City
S1	Smith	20	London
S2	Jones	10	Paris
S3	Blake	30	Paris
S4	Clark	20	London
S5	Adams	30	Athens

P#	Pname	Color	Weight	City
P1	Nut	Red	12	London
P2	Bolt	Green	17	Paris
P3	Screw	Blue	17	Rome
P4	Screw	Red	14	London
P5	Cam	Blue	12	Paris
P6	Cog	Red	19	London

S#	P#	QTY			
S1	P1	300	S2	P1	300
S1	P2	200	S2	P2	400
S1	P3	400	S3	P2	200
S1	P4	200	S4	P2	200
S1	P5	100	S4	P4	300
S1	P6	100	S4	P5	400

Get part numbers for all parts supplied by more than one supplier.

select *P#*
from *SP*
group by *P#*
having count(∗) > 1 ;

Result:

P#
P1
P2
P4
P5

Having is to groups what **where** is to rows. (If **having** is specified, **group by** should be also specified). **Having** is used to eliminate groups just as **where** is used to eliminate rows.

The same without **group by/having**.

select *P#*,
from *P*
where 1 < (**select count**(*S#*)
from *SP*
where *P#* = *P.P#*);

Set Operations (union)

S#	Sname	Status	City
S1	Smith	20	London
S2	Jones	10	Paris
S3	Blake	30	Paris
S4	Clark	20	London
S5	Adams	30	Athens

P#	Pname	Color	Weight	City
P1	Nut	Red	12	London
P2	Bolt	Green	17	Paris
P3	Screw	Blue	17	Rome
P4	Screw	Red	14	London
P5	Cam	Blue	12	Paris
P6	Cog	Red	19	London

S#	P#	QTY			
S1	P1	300	S2	P1	300
S1	P2	200	S2	P2	400
S1	P3	400	S3	P2	200
S1	P4	200	S4	P2	200
S1	P5	100	S4	P4	300
S1	P6	100	S4	P5	400

Get part numbers for parts with weight more than 16 pounds or are supplied by supplier S2.

select *P#*
from *P*
where *Weight* **> 16 union select** *P#*
from *SP*
where *S#* = 'S2' ;

Result:

<u>*P#*</u>
P1
P2
P3
P6

Since a relation is set of rows, it is possible to construct union, intersection and difference between them. However, to be result a relation the two original relation must be set-compatable:

1. *to have the same number of columns.*
2. *the i-th column of both relations must have the same data type.*

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.**

Set Operations (intersect, except)

S#	Sname	Status	City
S1	Smith	20	London
S2	Jones	10	Paris
S3	Blake	30	Paris
S4	Clark	20	London
S5	Adams	30	Athens

P#	Pname	Color	Weight	City
P1	Nut	Red	12	London
P2	Bolt	Green	17	Paris
P3	Screw	Blue	17	Rome
P4	Screw	Red	14	London
P5	Cam	Blue	12	Paris
P6	Cog	Red	19	London

S#	P#	QTY			
S1	P1	300	S2	P1	300
S1	P2	200	S2	P2	400
S1	P3	400	S3	P2	200
S1	P4	200	S4	P2	200
S1	P5	100	S4	P4	300
S1	P6	100	S4	P5	400

Get supplier numbers for suppliers who supply part P1 and are located in London.

select *S#*
from *SP*
where *P#* **=** 'P1' **intersect select** *S#*
from *S*
where *City* = 'London' ;

Result:

<u>*S#*</u>
S1

Get supplier numbers for suppliers who supply part P2 and are not located in London.

select *S#*
from *SP*
where *P#* **=** 'P2' **except select** *S#*
from *S*
where *City* = 'London' ;

Result:

<u>*S#*</u>
S2
S3

A Comprehensive Example

S#	*Sname*	*Status*	*City*
S1	Smith	20	London
S2	Jones	10	Paris
S3	Blake	30	Paris
S4	Clark	20	London
S5	Adams	30	Athens

P#	*Pname*	*Color*	*Weight*	*City*
P1	Nut	Red	12	London
P2	Bolt	Green	17	Paris
P3	Screw	Blue	17	Rome
P4	Screw	Red	14	London
P5	Cam	Blue	12	Paris
P6	Cog	Red	19	London

S#	*P#*	*QTY*			
S1	P1	300	S2	P1	300
S1	P2	200	S2	P2	400
S1	P3	400	S3	P2	200
S1	P4	200	S4	P2	200
S1	P5	100	S4	P4	300
S1	P6	100	S4	P5	400

For all red and blue parts such that the total quantity supplied is greater than 350 (excluding from the total all shipments for which the quantity is less than or equal to 200), get the part number, the weight in grams, the color, and the maximum supplied of that part. Order the result by decreasing part number within asceding values of that maximum.

select *P.P#,* 'Weight in grams = ', *P.Weight**454,
P.Color, 'MSQuantity = ', **max** (*SP.QTY*)
from *P, SP*
where *P.P#* = *SP.P#*
and *P.Color* **in** ('Red', 'Blue')
and *SP.QTY* **>** 200
group by *P.P#, P.Weight*; *P.Color*
having sum (*QTY*) > 350
order by 6, *P.P#,* **desc** ;

Result:

P#		*Color*	
P1	Weight in grams = 5448	Red	MSQuantity = 300
P5	Weight in grams = 5448	Blue	MSQuantity = 400
P3	Weight in grams = 7718	Blue	MSQuantity = 400

With Clause

S#	*Sname*	*Status*	*City*
S1	Smith	20	London
S2	Jones	10	Paris
S3	Blake	30	Paris
S4	Clark	20	London
S5	Adams	30	Athens

P#	*Pname*	*Color*	*Weight*	*City*
P1	Nut	Red	12	London
P2	Bolt	Green	17	Paris
P3	Screw	Blue	17	Rome
P4	Screw	Red	14	London
P5	Cam	Blue	12	Paris
P6	Cog	Red	19	London

S#	*P#*	*QTY*			
S1	P1	300	S2	P1	300
S1	P2	200	S2	P2	400
S1	P3	400	S3	P2	200
S1	P4	200	S4	P2	200
S1	P5	100	S4	P4	300
S1	P6	100	S4	P5	400

Get all supplier names with maximum status.

with maxst(value) **as**
select max(*Status*)
from *S*
select *Sname*
from *S*
where *Status* = maxst.value;

Result:

Sname
Blake
Adams

With clause allows views to be defined locally to a query, rather than globally. Analogous to procedures in a programming language.

Get all part numbers where the total their shipments is greater than the average of the total supplier shipments at all suppliers.

with ptotal(*P#*, value) **as**
select *P#,* **sum**(*QTY*)
from *SP*
group by *P#*
with pavg(*S#*, value) **as**
select *S#*, **avg**(*QTY*)
from *SP*
group by *P#*
select *P#*
from ptotal, pavg
where ptotal.value > pavg.value;

Result:

P#
P1
P2

Derived Relations

S#	Sname	Status	City
S1	Smith	20	London
S2	Jones	10	Paris
S3	Blake	30	Paris
S4	Clark	20	London
S5	Adams	30	Athens

P#	Pname	Color	Weight	City
P1	Nut	Red	12	London
P2	Bolt	Green	17	Paris
P3	Screw	Blue	17	Rome
P4	Screw	Red	14	London
P5	Cam	Blue	12	Paris
P6	Cog	Red	19	London

S#	P#	QTY			
S1	P1	300	S2	P1	300
S1	P2	200	S2	P2	400
S1	P3	400	S3	P2	200
S1	P4	200	S4	P2	200
S1	P5	100	S4	P4	300
S1	P6	100	S4	P5	400

Get the average quantity of those supplier shipments where the average quantity is greater than 250.

select *S#, AvgShip*
from (select *S#*, **avg** (*QTY*)
from *SP*
group by *S#*)
as *result* (*S#, AvgShip*)
where *AvgShip* **>** 250

Result:

S#	AvgShip
S2	350
S4	300

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

Views

S#	Sname	Status	City
S1	Smith	20	London
S2	Jones	10	Paris
S3	Blake	30	Paris
S4	Clark	20	London
S5	Adams	30	Athens

P#	Pname	Color	Weight	City
P1	Nut	Red	12	London
P2	Bolt	Green	17	Paris
P3	Screw	Blue	17	Rome
P4	Screw	Red	14	London
P5	Cam	Blue	12	Paris
P6	Cog	Red	19	London

S#	P#	QTY			
S1	P1	300	S2	P1	300
S1	P2	200	S2	P2	400
S1	P3	400	S3	P2	200
S1	P4	200	S4	P2	200
S1	P5	100	S4	P4	300
S1	P6	100	S4	P5	400

Create view from good suppliers (with status greater than 15).

create view *GoodSup*
as select *S#, Status,City*
from *S*
where *Status* **>** 15 ;

S#	Status	City
S1	20	London
S3	30	Paris
S4	20	London
S5	30	Athens

GoodSup is in effect a "window" into real table S. The window is dynamic because changes of S is automatically visible through the window *GoodSup*. Some users may genuinely believe that *GoodSup* is a "real" table.

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

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

where:

- <query expression> is any legal expression
- the view name is represented by *v*

Views

S#	Sname	Status	City
S1	Smith	20	London
S2	Jones	10	Paris
S3	Blake	30	Paris
S4	Clark	20	London
S5	Adams	30	Athens

P#	Pname	Color	Weight	City
P1	Nut	Red	12	London
P2	Bolt	Green	17	Paris
P3	Screw	Blue	17	Rome
P4	Screw	Red	14	London
P5	Cam	Blue	12	Paris
P6	Cog	Red	19	London

S#	P#	QTY			
S1	P1	300	S2	P1	300
S1	P2	200	S2	P2	400
S1	P3	400	S3	P2	200
S1	P4	200	S4	P2	200
S1	P5	100	S4	P4	300
S1	P6	100	S4	P5	400

Query on view (suppliers not located in London).

select *S#, City*
from *GoodSup*
where *City* **<>** 'London' ;

Result:

S#	City
S3	Paris
S5	Athens

Create view of part numbers and names for parts with weight more than 16 pounds or are supplied by supplier S2.

select *P#, Pname*
from *P*
where *Weight* **> 16 union**
select distinct *P#, Pname*
from *P, SP*
where *P.P#* = *SP.P#*
and *S#* = 'S2' ;

Result:

P#	Pname
P1	Nut
P2	Bolt
P3	Screw
P6	Cog

Modification of the Database – Deletion

S#	Sname	Status	City
S1	Smith	20	London
S2	Jones	10	Paris
S3	Blake	30	Paris
S4	Clark	20	London
S5	Adams	30	Athens

P#	Pname	Color	Weight	City
P1	Nut	Red	12	London
P2	Bolt	Green	17	Paris
P3	Screw	Blue	17	Rome
P4	Screw	Red	14	London
P5	Cam	Blue	12	Paris
P6	Cog	Red	19	London

S#	P#	QTY			
S1	P1	300	S2	P1	300
S1	P2	200	S2	P2	400
S1	P3	400	S3	P2	200
S1	P4	200	S4	P2	200
S1	P5	100	S4	P4	300
S1	P6	100	S4	P5	400

Delete all suppliers in Paris.

delete *S#, City*
from *S*
where *City* **=** 'Paris' ;

Delete all shipments.

delete
from *SP* ;

Delete all shipments for suppliers in London.

delete
from *SP*
where 'London' **= (select** *City*
from *S*
where *S.S#* = *SP.S#* **)** ;

Modification of the Database – Deletion

S#	Sname	Status	City
S1	Smith	20	London
S2	Jones	10	Paris
S3	Blake	30	Paris
S4	Clark	20	London
S5	Adams	30	Athens

P#	Pname	Color	Weight	City
P1	Nut	Red	12	London
P2	Bolt	Green	17	Paris
P3	Screw	Blue	17	Rome
P4	Screw	Red	14	London
P5	Cam	Blue	12	Paris
P6	Cog	Red	19	London

S#	P#	QTY			
S1	P1	300	S2	P1	300
S1	P2	200	S2	P2	400
S1	P3	400	S3	P2	200
S1	P4	200	S4	P2	200
S1	P5	100	S4	P4	300
S1	P6	100	S4	P5	400

General form of **delete** statement:

delete
from *table*
[where *predicate*]

Delete all shipments with quantity below the average.

delete
from *SP*
where *QTY* **< (select avg**(*QTY*)
from *SP* **)** ;

Problem: **as we delete tuples from *SP*, the average quantity 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

S#	Sname	Status	City
S1	Smith	20	London
S2	Jones	10	Paris
S3	Blake	30	Paris
S4	Clark	20	London
S5	Adams	30	Athens

P#	Pname	Color	Weight	City
P1	Nut	Red	12	London
P2	Bolt	Green	17	Paris
P3	Screw	Blue	17	Rome
P4	Screw	Red	14	London
P5	Cam	Blue	12	Paris
P6	Cog	Red	19	London

S#	P#	QTY			
S1	P1	300	S2	P1	300
S1	P2	200	S2	P2	400
S1	P3	400	S3	P2	200
S1	P4	200	S4	P2	200
S1	P5	100	S4	P4	300
S1	P6	100	S4	P5	400

Add part P7 with unknown name and color.

insert
into *P* (*P#, City, Weight*)
values ('P7', 'Athens', 2) ;

Name and *color* will have **null** values.

Add part P8 to table P.

insert
into *P*
values ('P8', 'Sprocket', 'Pink', 14, 'Nice') ;

Add a new shipment with supplier S20, part number p20, and quantity 1000.

insert
into *SP* (*S#, P#, QTY*)
values ('S20', 'P20', 1000) ;

Modification of the Database – Insertion

S#	Sname	Status	City
S1	Smith	20	London
S2	Jones	10	Paris
S3	Blake	30	Paris
S4	Clark	20	London
S5	Adams	30	Athens

P#	Pname	Color	Weight	City
P1	Nut	Red	12	London
P2	Bolt	Green	17	Paris
P3	Screw	Blue	17	Rome
P4	Screw	Red	14	London
P5	Cam	Blue	12	Paris
P6	Cog	Red	19	London

S#	P#	QTY			
S1	P1	300	S2	P1	300
S1	P2	200	S2	P2	400
S1	P3	400	S3	P2	200
S1	P4	200	S4	P2	200
S1	P5	100	S4	P4	300
S1	P6	100	S4	P5	400

General form of **insert** statement:

insert
into *table* [(field$_1$, field$_2$, field$_3$, …)]
values (constant$_1$, constant$_2$, constant$_3$, …) ; or

insert
into *table* [(field$_1$, field$_2$, field$_3$, …)]
subquery ;

For each part supplied, get the part number and the total quantity, and save the result in the database.

create table *temp*
(*P#* **char**(6)
TOTQTY **integer**) ;

insert into *temp* (*P#, TOTQTY*)
select *P#,* sum(*QTY*)
from *SP*
group by *P#* ;

Modification of the Database – Updates

S#	Sname	Status	City
S1	Smith	40	London
S2	Jones	10	Paris
S3	Blake	30	Paris
S4	Clark	40	London
S5	Adams	30	Athens

P#	Pname	Color	Weight	City
P1	Nut	Red	12	London
P2	Bolt	Yellow	22	Paris
P3	Screw	Blue	17	Rome
P4	Screw	Red	14	London
P5	Cam	Blue	12	Paris
P6	Cog	Red	19	London

S#	P#	QTY			
S1	P1	300	S2	P1	0
S1	P2	200	S2	P2	0
S1	P3	400	S3	P2	0
S1	P4	200	S4	P2	200
S1	P5	100	S4	P4	300
S1	P6	100	S4	P5	400

Double status for all suppliers in London.

update *S*
set *status* = *status* ∗ *2*
where *city* = ‘London’ ;

Change the color and weight of part P2.

update *P*
set *color* = ‘Yellow’, *weight* = *weight* + 5
where *P#* = ‘P2’ ;

Set the shipment quantity to zero for all suppliers in Paris.

update *SP*
set *QTY* = 0
where ‘Paris’ = (**select** *city*
from *S*
where *S.S#* = *SP.S#*) ;

Modification of the Database – Updates

S#	Sname	Status	City
S1	Smith	20	London
S2	Jones	10	Paris
S3	Blake	30	Paris
S4	Clark	20	London
S5	Adams	30	Athens

P#	Pname	Color	Weight	City
P1	Nut	Red	12	London
P2	Bolt	Green	17	Paris
P3	Screw	Blue	17	Rome
P4	Screw	Red	14	London
P5	Cam	Blue	12	Paris
P6	Cog	Red	19	London

S#	P#	QTY			
S1	P1	318	S2	P1	318
S1	P2	210	S2	P2	424
S1	P3	424	S3	P2	210
S1	P4	210	S4	P2	210
S1	P5	105	S4	P4	318
S1	P6	105	S4	P5	424

General form of **update** statement:

update *table*
set field = expression
[, field = expression] …
where *predicate* ;

Increase all shipment quantities over 200 by 6%, and all others by 5%.

update *SP*
set *QTY* = *QTY* ∗ 1.06
where *QTY* **>** 200

update *account*
set *QTY* = *QTY* ∗ 1.05
where *QTY* **<=** 200

The order is important
Can be done better using the **case** statement

update *SP*
set *QTY* = **case**
when *QTY* **<=** 200 **then** *QTY* ∗ 1.05
else *QTY* ∗ 1.06
end

Modification of the Views

S#	Sname	Status	City
S1	Smith	20	London
S2	Jones	10	Paris
S3	Blake	30	Paris
S4	Clark	20	London
S5	Adams	30	Athens

P#	Pname	Color	Weight	City
P1	Nut	Red	12	London
P2	Bolt	Green	17	Paris
P3	Screw	Blue	17	Rome
P4	Screw	Red	14	London
P5	Cam	Blue	12	Paris
P6	Cog	Red	19	London

S#	P#	QTY			
S1	P1	300	S2	P1	300
S1	P2	200	S2	P2	400
S1	P3	400	S3	P2	200
S1	P4	200	S4	P2	200
S1	P5	100	S4	P4	300
S1	P6	100	S4	P5	400

Create a view of shipment relation (*SP*), hiding the *QTY* attribute.

create view *Ship* **as**
select *S#, P#*
from *SP*

Add a new shipment to ship.

insert
into *Ship*
values (‘S5’, ‘P6’) ;

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

Transactions

- A transaction is a sequence of queries and update statements executed as a single unit
  - Transactions are started implicitly and terminated by one of
    - **commit work:** makes all updates of the transaction permanent in the database
    - **rollback work:** undoes all updates performed by the transaction.
- Motivating example
  - Transfer of money from one account to another involves two steps:
    - deduct from one account and credit to another
  - If one steps succeeds and the other fails, database is in an inconsistent state
  - Therefore, either both steps should succeed or neither should
- If any step of a transaction fails, all work done by the transaction can be undone by **rollback work.**
- Rollback of incomplete transactions is done automatically, in case of system failures

Transactions (Cont.)

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

begin atomic
...
end

Joined Relations

- Join operations take two relations and return as a result another relation.
- These additional operations are typically used as subquery expressions in the **from** clause
- Join condition – defines which tuples in the two relations match, and what attributes are present in the result of the join.
- Join type – defines how tuples in each relation that do not match any tuple in the other relation (based on the join condition) are treated.

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

Join Conditions
natural
on <predicate>
using $(A_1, A_2, ..., A_n)$

Joined Relations – Datasets for Examples

- Relation *loan*

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

- Relation *borrower*

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

- 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*

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

- loan **natural right outer join** *borrower*

loan-number	*branch-name*	*amount*	*customer-name*
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)*

loan-number	*branch-name*	*amount*	*customer-name*
L-170	Downtown	3000	Jones
L-230	Redwood	4000	Smith
L-260	Perryridge	1700	*null*
L-155	null	null	Hayes

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

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

Data Definition Language (DDL)

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

- The schema for each relation.
- The domain of values associated with each attribute.
- Integrity constraints
- The set of indices to be maintained for each relations.
- Security and authorization information for each relation.
- The physical storage structure of 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.
- Null values are allowed in all the domain types. Declaring an attribute to be **not null** prohibits null values for that attribute.
- **create domain** construct in SQL-92 creates user-defined domain types

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

Date/Time Types in SQL (Cont.)

- **date.** Dates, containing a (4 digit) year, month and date
    - E.g. **date** '2001-7-27'
- **time.** Time of day, in hours, minutes and seconds.
    - E.g. **time** '09:00:30' **time** '09:00:30.75'
- **timestamp**: date plus time of day
    - E.g. **timestamp** '2001-7-27 09:00:30.75'
- **Interval**: period of time
    - E.g. Interval '1' day
    - Subtracting a date/time/timestamp value from another gives an interval value
    - Interval values can be added to date/time/timestamp values
- Can extract values of individual fields from date/time/timestamp
    - E.g. **extract** (**year from** r.starttime)
- Can cast string types to date/time/timestamp
    - E.g. **cast** <string-valued-expression> **as date**

Create Table Construct

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

  create table *r* (A_1 D_1, A_2 D_2, ..., A_n D_n,
  (integrity-constraint$_1$),
  ...,
  (integrity-constraint$_k$))

  - *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 *S*
  (*S#* **char**(5) **not null,**
  Sname **char**(20),
  Status **smallint**,
  City **char**(15)) ;

Integrity Constraints in Create Table

- **not null**
- **primary key** (A_1, ..., A_n)
- **check** *(Predicate),* where *Predicate* is a predicate

Example: Declare table *P* (Parts).

create table *P*
(*P#* **char**(6) **not null,**
Pname **char**(20)
Color **char**(10),
Weight **smallint**,
City **char**(15),
primary key (*P#*),
check (*Weight* **>=** 0))

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

Drop and Alter Table Constructs

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

  alter table *r* **add** *A D*

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

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

Chapter 5: Relational Database Design

Chapter 5: Relational Database Design

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

First Normal Form

- Domain is atomic if its elements are considered to be indivisible units
  - Examples of non-atomic domains:
    - Set of names, composite attributes
    - Identification numbers like CS101 that can be broken up into parts
- A relational schema R is in first normal form if the domains of all attributes of R are atomic
- Non-atomic values complicate storage and encourage redundant (repeated) storage of data
  - E.g. Set of accounts stored with each customer, set of children stored with each person, etc.

First Normal Form (Contd.)

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

Pitfalls in Relational Database Design

- Relational database design requires that we find a "good" collection of relation schemas. A bad design may lead to
  - Repetition of Information.
  - Inability to represent certain information.
- Design Goals:
  - Avoid redundant data
  - Ensure that relationships among attributes are represented
  - Facilitate the checking of updates for violation of database integrity constraints.

Example

- Consider the relation schema:

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

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

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

Decomposition

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

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

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

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

$$R = R_1 \cup R_2$$

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

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

Example of Non Lossless-Join Decomposition

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

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

A	B
α	1
α	2
β	1

r

A
α
β

$\Pi_A(r)$

B
1
2

$\Pi_B(r)$

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

A	B
α	1
α	2
β	1
β	2

Goal — Devise a Theory for the Following

- Decide whether a particular relation R is in "good" form.
- In the case that a relation R is not in "good" form, decompose it into a set of relations $\{R_1, R_2, ..., R_n\}$ 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 $\alpha \subseteq R$ and $\beta \subseteq R$
- The functional dependency

$$\alpha \rightarrow \beta$$

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

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

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

1	4
1	5
3	7

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

Functional Dependencies (Cont.)

- K is a superkey for relation schema R if and only if $K \rightarrow R$
- K is a candidate key for R if and only if
  - $K \rightarrow R$, and
  - for no $\alpha \subset K$, $\alpha \rightarrow R$
- Functional dependencies allow us to express constraints that cannot be expressed using superkeys. Consider the schema:

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

  We expect this set of functional dependencies to hold:

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

  but would not expect the following to hold:

  loan-number $\rightarrow$ *customer-name*

Example

Drinkers(<u>name</u>, addr, <u>beersLiked</u>, manf, favoriteBeer)

name	addr	beersLiked	manf	favoriteBeer
Janeway	Voyager	Bud	A.B.	WickedAle
Janeway	Voyager	WickedAle	Pete's	WickedAle
Spock	Enterprise	Bud	A.B.	Bud

- Reasonable FD's to assert:
  1. name $\rightarrow$ addr
  2. name $\rightarrow$ favoriteBeer
  3. beersLiked $\rightarrow$ manf
- Sometimes, several attributes jointly determine another attribute, although neither does by itself. Example:

  beer bar $\rightarrow$ price

Functional Dependencies

- *A* functional dependency is trivial if it is satisfied by all instances of a relation
  - *E.g.*
    - *customer-name, loan-number* $\rightarrow$ *customer-name*
    - *customer-name* $\rightarrow$ *customer-name*
  - In general, $\alpha \rightarrow \beta$ is trivial if $\beta \subseteq \alpha$
- "Nontrivial" = right-side attribute not in left side

Closure of a Set of Functional Dependencies

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

Example

- $R = (A, B, C, G, H, I)$
  $F = \{ A \rightarrow B$
  $A \rightarrow C$
  $CG \rightarrow H$
  $CG \rightarrow I$
  $B \rightarrow H\}$
- some members of F^+
  - $A \rightarrow H$
    - by transitivity from $A \rightarrow B$ *and* $B \rightarrow H$
  - $AG \rightarrow I$
    - by augmenting $A \rightarrow C$ with G, to get $AG \rightarrow CG$ and then transitivity with $CG \rightarrow I$
  - $CG \rightarrow HI$
    - from $CG \rightarrow H$ *and* $CG \rightarrow I$: "union rule" can be inferred from
      - definition of functional dependencies, or
      - Augmentation of $CG \rightarrow I$ to infer $CG \rightarrow CGI$, augmentation 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 will 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 $\gamma\beta \rightarrow \delta$ holds, then $\alpha\gamma \rightarrow \delta$ holds **(pseudotransitivity)**

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

Closure of Attribute Sets

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

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

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

  result := α;
  while (changes to *result*) **do**
      for each $\beta \rightarrow \gamma$ **in** F **do**
        begin
          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*   ($A \rightarrow C$ and $A \rightarrow B$)
  3. *result* = *ABCGH*   ($CG \rightarrow H$ and $CG \subseteq AGBC$)
  4. *result* = *ABCGHI*   ($CG \rightarrow I$ and $CG \subseteq AGBCH$)
- Is *AG* a candidate key?
  1. Is AG a super key?
     1. Does $AG \rightarrow R$? == Is $(AG)^+ \supseteq R$
  2. Is any subset of AG a superkey?
     1. Does $A \rightarrow R$? == Is $(A)^+ \supseteq R$
     2. Does $G \rightarrow R$? == 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 $\alpha^{+,}$ 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 $\gamma \rightarrow S$.

Goals of Normalization

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

Decomposition

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

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

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

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

$$R = R_1 \cup R_2$$

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

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

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

Normalization Using Functional Dependencies

- When we decompose a relation schema R with a set of functional dependencies F into $R_1, R_2, .., R_n$ we want
  - Lossless-join decomposition: Otherwise decomposition would result in information loss.
  - No redundancy: The relations R_i preferably should be in either Boyce-Codd Normal Form or Third Normal Form.
  - Dependency preservation: Let F_i be the set of dependencies F^+ that include only attributes in R_i.
    - Preferably the decomposition should be dependency preserving, that is, $(F_1 \cup F_2 \cup \ldots \cup F_n)^+ = F^+$
    - Otherwise, checking updates for violation of functional dependencies may require computing joins, which is expensive.

Example

- $R = (A, B, C)$
  $F = \{A \rightarrow B, B \rightarrow C)$
  - Can be decomposed in two different ways
- $R_1 = (A, B), \quad R_2 = (B, C)$
  - Lossless-join decomposition:
    $R_1 \cap R_2 = \{B\}$ and $B \rightarrow BC$
  - Dependency preserving
- $R_1 = (A, B), \quad R_2 = (A, C)$
  - Lossless-join decomposition:
    $R_1 \cap R_2 = \{A\}$ and $A \rightarrow AB$
  - Not dependency preserving
    (cannot check $B \rightarrow C$ without computing $R_1 \bowtie R_2$)

Second Normal Form

A relation schema R is in 2NF respect to a set F of functional dependencies if for all nonkey set of attributes β holds:

- $\alpha \rightarrow \beta$ *where* α is a superkey for R

  A relation is said to be in Second Normal Form when every nonkey attribute is fully functionally dependent on the primary key. (No attribute dependent on a portion of primary key)
  - That is, every nonkey attribute needs the full primary key for unique identification
  - It is important only in cases of keys containing more than one attribute

Example

Drinkers (<u>name</u>, addr, <u>beersLiked</u>, manf, favoriteBeer)

name	addr	beersLiked	manf	favoriteBeer
Janeway	Voyager	Bud	A.B.	WickedAle
Janeway	Voyager	WickedAle	Pete's	WickedAle
Spock	Enterprise	Bud	A.B.	Bud

FD's: name $\rightarrow$ addr, name $\rightarrow$ favoriteBeer, beersLiked $\rightarrow$ manf
violates 2NF.

Lending-schema (*<u>branch-name</u>*, *<u>branch-city</u>*, *assets*, *customer-name*, *<u>loan-number</u>*, *amount)*

FD's: *<u>branch-name</u>* *<u>branch-city</u>* $\rightarrow$ *assets* violates 2NF.

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*

R is in BCNF if for every nontrivial FD for *R*, say $X \rightarrow A$, then *X* is a superkey.

Follow from the idea "key $\rightarrow$ everything."

1. Guarantees no redundancy due to FD's.
2. Guarantees no *update anomalies* = one occurrence of a fact is updated, not all.
3. Guarantees no *deletion anomalies* = valid fact is lost when tuple is deleted.

Example

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

Example of Problems

Drinkers(<u>name</u>, addr, <u>beersLiked</u>, manf, favoriteBeer)

name	addr	beersLiked	manf	favoriteBeer
Janeway	Voyager	Bud	A.B.	WickedAle
Janeway	???	WickedAle	Pete's	???
Spock	Enterprise	Bud	???	Bud

FD's:

1. name $\rightarrow$ addr
2. name $\rightarrow$ favoriteBeer
3. beersLiked $\rightarrow$ manf

- ???'s are redundant, since we can figure them out from the FD's.
- Update anomalies: If Janeway gets transferred to the *Intrepid*, will we change addr in each of her tuples?
- Deletion anomalies: If nobody likes Bud, we lose track of Bud's manufacturer.

Each of the given FD's is a BCNF violation:

- Key = {name, beersLiked}
  - Each of the given FD's has a left side that is a proper subset of the key.

Another Example

Beers(<u>name</u>, manf, manfAddr). (Note: 2NF is satisfied)

- FD's = name $\rightarrow$ manf, manf $\rightarrow$ manfAddr.
- Only key is name.
  - manf $\rightarrow$ manfAddr violates BCNF with a left side unrelated to any key.

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
  - E.g. Consider R (A, B, C, D), with F = { $A \rightarrow B$, $B \rightarrow C$}
    - Decompose R into $R_1(A,B)$ and $R_2(A,C,D)$
    - Neither of the dependencies in F contain only attributes from (A,C,D) so we might be mislead into thinking R_2 satisfies BCNF.
    - In fact, dependency $A \rightarrow C$ in F^+ shows R_2 is not in BCNF.

BCNF Decomposition Algorithm (1)

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

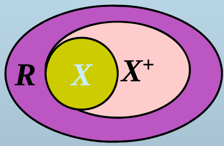
- *R = (branch-name, branch-city, assets, customer-name, loan-number, amount)*
  F = {branch-name $\rightarrow$ *assets branch-city*
  loan-number $\rightarrow$ *amount branch-name}*
  Key = *{loan-number, customer-name}*
- Decomposition
  - R_1 = *(branch-name, branch-city, assets)*
  - R_2 = *(branch-name, customer-name, loan-number, amount)*
  - R_3 = *(branch-name, loan-number, amount)*
  - R_4 = *(customer-name, loan-number)*
- Final decomposition
  R_1, R_3, R_4

BCNF Decomposition Algorithm (2)

Setting: relation R, given FD's F.

Suppose relation R has BCNF violation $X \rightarrow B$.

1. Compute X^+.
   - Cannot be all attributes – why?
2. Decompose R into X^+ and $(R–X^+) \cup X$.



3. Find the FD's for the decomposed relations.
   - Project the FD's from F = calculate all consequents of F that involve only attributes from X^+ or only from $(R–X^+) \cup X$.

Example

R = Drinkers(<u>name</u>, addr, <u>beersLiked</u>, manf, favoriteBeer)

F =

1. name → addr
2. name → favoriteBeer
3. beersLiked → manf

Pick BCNF violation name → addr.

- Close the left side: name$^+$ = name addr favoriteBeer.
- Decomposed relations:

  Drinkers1(<u>name</u>, addr, favoriteBeer)

  Drinkers2(<u>name</u>, <u>beersLiked</u>, manf)
- Projected FD's (skipping a lot of work that leads nowhere interesting):
  - For Drinkers1: name → addr and name → favoriteBeer.
  - For Drinkers2: beersLiked → manf.

(Repeating)

- Decomposed relations:
  Drinkers1(name, addr, favoriteBeer)
  Drinkers2(name, beersLiked, manf)
- Projected FD's:
  - For Drinkers1: name $\rightarrow$ addr and name $\rightarrow$ favoriteBeer.
  - For Drinkers2: beersLiked $\rightarrow$ manf.
- BCNF violations?
  - For Drinkers1, name is key and all left sides of FD's are superkeys.
  - For Drinkers2, {name, beersLiked} is the key, and beersLiked $\rightarrow$ manf violates BCNF.

Decompose Drinkers2

- First set of decomposed relations:
  Drinkers1(name, addr, favoriteBeer)
  Drinkers2(name, beersLiked, manf)
- Close beersLiked$^+$ = beersLiked, manf.
- Decompose Drinkers2 into:
  Drinkers3(beersLiked, manf)
  Drinkers4(name, beersLiked)
- Resulting relations are all in BCNF:
  Drinkers1(name, addr, favoriteBeer)
  Drinkers3(beersLiked, manf)
  Drinkers4(name, beersLiked)

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 and Dependency Preservation

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

- $R = (J, K, L)$
  $F = \{JK \rightarrow L$
  $L \rightarrow K\}$
  Two candidate keys = JK and JL
- R is not in BCNF
- Any decomposition of R will fail to preserve

$$JK \rightarrow L$$

Third Normal Form: Motivation

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

Example

One FD structure causes problems:

- If you decompose, you can't check all the FD's only in the decomposed relations.
- If you don't decompose, you violate BCNF.

  Abstractly: $R = (A, B, C)$, $F = \{AB \rightarrow C, C \rightarrow B.\}$

  Example: street city $\rightarrow$ zip, zip $\rightarrow$ city.

  Keys: $\{A, B\}$ and $\{A, C\}$, but $C \rightarrow B$ has a left side that is not a superkey.

- Suggests decomposition into $\{B, C\}$ and $\{A, C\}$.
  - But you can't check the FD: $AB \rightarrow C$ in only these relations (requires a join)
- Equivalent to example in book:

  Banker-schema = (branch-name, customer-name, banker-name)

  banker-name $\rightarrow$ branch name

  branch name customer-name $\rightarrow$ banker-name

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).

Testing for 3NF

- Optimization: Need to check only FDs in F, need not check all FDs in F^+.
- Use attribute closure to check for each dependency $\alpha \rightarrow \beta$, if α is a superkey.
- If α is not a superkey, we have to verify if each attribute in β is contained in a candidate key of R
  - this test is rather more expensive, since it involve finding candidate keys
  - testing for 3NF has been shown to be NP-hard
  - 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 $(R_1, R_2, ..., R_i)$

What 3NF Gives You

There are two important properties of a decomposition:

1. We should be able to recover from the decomposed relations the data of the original.
   - Recovery involves projection and join.
2. We should be able to check that the FD's for the original relation are satisfied by checking the projections of those FD's in the decomposed relations.

- Without proof, we assert that it is always possible to decompose into BCNF and satisfy (1).
- Also without proof, we can decompose into 3NF and satisfy both (1) and (2).
- But it is not possible to decompose into BNCF and get both (1) and (2).
  - Street-city-zip is an example of this point.

Example

- Relation schema:

  Banker-info-schema = (branch-name, customer-name, banker-name, office-number)

- The functional dependencies for this relation schema are:

  banker-name $\rightarrow$ *branch-name office-number*
  customer-name branch-name $\rightarrow$ *banker-name*

- The key is:

  {*customer-name, branch-name*}

Applying 3NF to *Banker-info-schema*

- The **for** loop in the algorithm causes us to include the following schemas in our decomposition:

  Banker-office-schema = (banker-name, branch-name, office-number)
  Banker-schema = (customer-name, branch-name, banker-name)

- Since *Banker-schema* contains a candidate key for *Banker-info-schema*, we are done with the decomposition process.

Comparison of BCNF and 3NF

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

Comparison of BCNF and 3NF (Cont.)

- Example of problems due to redundancy in 3NF
  - $R = (A, B, C)$
    $F = \{AB \rightarrow C, C \rightarrow B\}$

A	B	C
a_1	b_1	c_1
a_2	b_1	c_1
a_3	b_1	c_1
null	b_2	c_2

A schema that is in 3NF but not in BCNF has the problems of

- repetition of information (e.g., the relationship b_1, c_1)
- need to use null values (e.g., to represent the relationship b_2, c_2 where there is no corresponding value for A).

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

- There are database schemas in BCNF that do not seem to be sufficiently normalized
- Consider a database

  classes(course, teacher, book)

  such that $(c,t,b) \in$ *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).

Multivalued Dependencies (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	Shaw
operating systems	Jim	OS Concepts
operating systems	Jim	Shaw

classes

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

  (database, Sara, DB Concepts)
  (database, Sara, Ullman)

Multivalued Dependencies (Cont.)

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

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

teaches

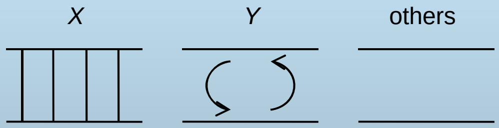
course	*book*
database	DB Concepts
database	Ullman
operating systems	OS Concepts
operating systems	Shaw

text

We shall see that these two relations are in Fourth Normal Form (4NF)

Multivalued Dependencies Def.

The *multivalued dependency* $X \rightarrow\rightarrow Y$ holds in a relation R if whenever we have two tuples of R that agree in all the attributes of X, then we can swap their Y components and get two new tuples that are also in R.



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.

Example

Drinkers(name, addr, phones, beersLiked)

with MVD Name $\rightarrow\rightarrow$ phones. If Drinkers has the two tuples:

name	addr	phones	beersLiked
sue	*a*	*p*1	*b*1
sue	*a*	*p*2	*b*2

it must also have the same tuples with phones components swapped:

name	addr	phones	beersLiked
sue	*a*	*p*2	*b*1
sue	*a*	*p*1	*b*2

Note: we must check this condition for *all* pairs of tuples that agree on name, not just one pair.

MVD Rules

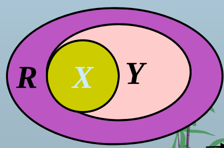
1. Every FD is an MVD.
   - Because if $X \rightarrow Y$, then swapping Y's between tuples that agree on X doesn't create new tuples.
   - Example, in `Drinkers`: `name` $\rightarrow\rightarrow$ `addr`.
2. *Complementation*: if $X \rightarrow\rightarrow Y$, then $X \rightarrow\rightarrow Z$, where Z is all attributes not in X or Y.
   - Example: since `name` $\rightarrow\rightarrow$ `phones` holds in `Drinkers`, so does `name` $\rightarrow\rightarrow$ `addr beersLiked`.

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

4NF eliminates redundancy due to multiplicative effect of MVD's.

- Formally: R is in Fourth Normal Form if whenever MVD $X \rightarrow\rightarrow Y$ is *nontrivial* (Y is not a subset of X, and $X \cup Y$ is not all attributes), then X is a superkey.
  - Remember, $X \rightarrow Y$ implies $X \rightarrow\rightarrow Y$, so 4NF is more stringent than BCNF.



- Decompose R, using NF violation $X \rightarrow\rightarrow Y$, into XY and $X \cup (R—Y)$.

4NF Decomposition Algorithm

result: = {R};
done := false;
compute D^+;
Let D_i denote the restriction of D^+ to R_i

while (**not** *done*)
  if (there is a schema $\mathbf{R}_i$ in *result* that is not in 4NF) **then**
    begin
      let $\alpha \rightarrow\rightarrow \beta$ be a nontrivial multivalued dependency that holds on R_i such that $\alpha \rightarrow R_i$ is not in D_i, and $\alpha \cap \beta = \phi$;
      result := (*result* - R_i) $\cup$ (R_i - β) $\cup$ (α, β);
    end
  else *done*:= true;

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

Splitting Doesn't Hold

Sometimes you need to have several attributes on the right of an MVD. For example:

Drinkers(name, areaCode, phones, beersLiked, beerManf)

name	areaCode	phones	beersLiked	beerManf
Sue	831	555-1111	Bud	A.B.
Sue	831	555-1111	Wicked Ale	Pete's
Sue	408	555-9999	Bud	A.B.
Sue	408	555-9999	Wicked Ale	Pete's

- name →→ areaCode phones holds, but neither name →→ areaCode nor name →→ phones do.

Example

Drinkers(name, addr, phones, beersLiked)

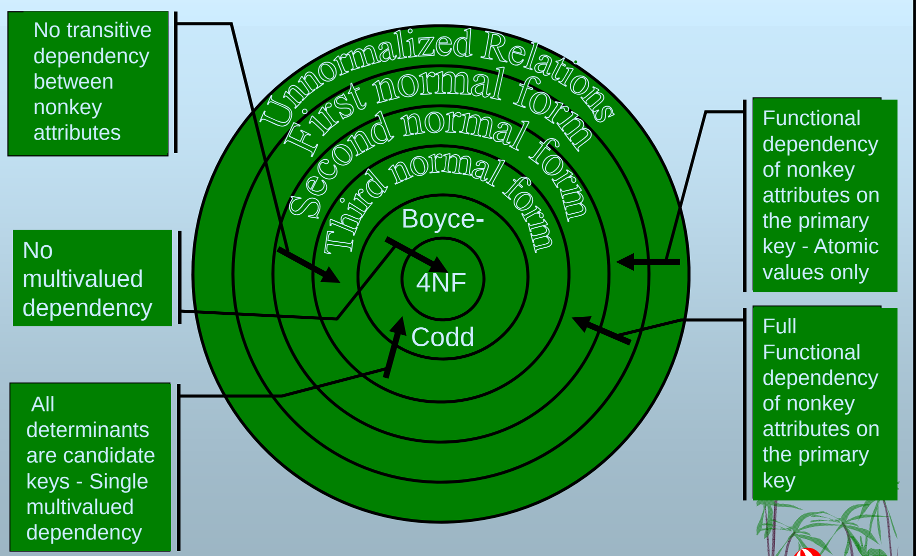
- FD: name → addr
- Nontrivial MVD's: name →→ phones and name →→ beersLiked.
- Only key: {name, phones, beersLiked}
- All three dependencies above violate 4NF.
- Successive decomposition yields 4NF relations:

  D1(<u>name</u>, addr)

  D2(<u>name</u>, <u>phones</u>)

  D3(<u>name</u>, <u>beersLiked</u>)

Normalization



Further Normal Forms

- **Join dependencies** generalize multivalued dependencies
  - lead to **project-join normal form (**PJNF) (also called **fifth normal form**)
  - A relation is in 5NF if every join dependency in the relation is implied by the keys of the relation
  - Implies that relations that have been decomposed in previous NF can be recombined via natural joins to recreate the original relation
- 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
- **The normalized relations grows in additive way while non-normalized relations grows in multiplicative way.**

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.

  - In practice, usually we start with more relations that intuitively satisfy some normal forms.

ER Model and Normalization

- When an E-R diagram is carefully designed, identifying all entities correctly, the tables generated from the E-R diagram should not need further normalization.
- However, in a real (imperfect) design there can be FDs from non-key attributes of an entity to other attributes of the entity
- E.g. *employee* entity with attributes *department-number* and *department-address*, and an FD *department-number* $\rightarrow$ *department-address*
  - Good design would have made department an entity
- FDs 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
- E.g. displaying *customer-name* along with *account-number* and *balance* requires join of *account* with *depositor*
- Alternative 1: Use denormalized relation containing attributes of *account* as well as *depositor* with all above attributes
  - faster lookup
  - Extra space and extra execution time for updates
  - extra coding work for programmer and possibility of error in extra code
- Alternative 2: use a materialized view defined as
  account $\bowtie$ 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-2000, earnings-2001, earnings-2002*, etc., all on the schema (*company-id, earnings*).
    - Above are in BCNF, but make querying across years difficult and needs new table each year
  - *company-year*(*company-id, earnings-2000, earnings-2001, earnings-2002*)
    - 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

End of Chapter

Sample *lending* Relation

branch-name	*branch-city*	*assets*	*customer-name*	*loan-number*	*amount*
Downtown	Brooklyn	9000000	Jones	L-17	1000
Redwood	Palo Alto	2100000	Smith	L-23	2000
Perryridge	Horseneck	1700000	Hayes	L-15	1500
Downtown	Brooklyn	9000000	Jackson	L-14	1500
Mianus	Horseneck	400000	Jones	L-93	500
Round Hill	Horseneck	8000000	Turner	L-11	900
Pownal	Bennington	300000	Williams	L-29	1200
North Town	Rye	3700000	Hayes	L-16	1300
Downtown	Brooklyn	9000000	Johnson	L-18	2000
Perryridge	Horseneck	1700000	Glenn	L-25	2500
Brighton	Brooklyn	7100000	Brooks	L-10	2200

Sample Relation *r*

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

The *customer* Relation

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

The *loan* Relation

loan-number	*branch-name*	*amount*
L-17	Downtown	1000
L-23	Redwood	2000
L-15	Perryridge	1500
L-14	Downtown	1500
L-93	Mianus	500
L-11	Round Hill	900
L-29	Pownal	1200
L-16	North Town	1300
L-18	Downtown	2000
L-25	Perryridge	2500
L-10	Brighton	2200

The *branch* Relation

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

The Relation *branch-customer*

branch-name	*branch-city*	*assets*	*customer-name*
Downtown	Brooklyn	9000000	Jones
Redwood	Palo Alto	2100000	Smith
Perryridge	Horseneck	1700000	Hayes
Downtown	Brooklyn	9000000	Jackson
Mianus	Horseneck	400000	Jones
Round Hill	Horseneck	8000000	Turner
Pownal	Bennington	300000	Williams
North Town	Rye	3700000	Hayes
Downtown	Brooklyn	9000000	Johnson
Perryridge	Horseneck	1700000	Glenn
Brighton	Brooklyn	7100000	Brooks

The Relation *customer-loan*

customer-name	*loan-number*	*amount*
Jones	L-17	1000
Smith	L-23	2000
Hayes	L-15	1500
Jackson	L-14	1500
Jones	L-93	500
Turner	L-11	900
Williams	L-29	1200
Hayes	L-16	1300
Johnson	L-18	2000
Glenn	L-25	2500
Brooks	L-10	2200

The Relation *branch-customer* ⋈ *customer-loan*

branch-name	*branch-city*	*assets*	*customer-name*	*loan-number*	*amount*
Downtown	Brooklyn	9000000	Jones	L-17	1000
Downtown	Brooklyn	9000000	Jones	L-93	500
Redwood	Palo Alto	2100000	Smith	L-23	2000
Perryridge	Horseneck	1700000	Hayes	L-15	1500
Perryridge	Horseneck	1700000	Hayes	L-16	1300
Downtown	Brooklyn	9000000	Jackson	L-14	1500
Mianus	Horseneck	400000	Jones	L-17	1000
Mianus	Horseneck	400000	Jones	L-93	500
Round Hill	Horseneck	8000000	Turner	L-11	900
Pownal	Bennington	300000	Williams	L-29	1200
North Town	Rye	3700000	Hayes	L-15	1500
North Town	Rye	3700000	Hayes	L-16	1300
Downtown	Brooklyn	9000000	Johnson	L-18	2000
Perryridge	Horseneck	1700000	Glenn	L-25	2500
Brighton	Brooklyn	7100000	Brooks	L-10	2200

An Instance of *Banker-schema*

customer-name	*banker-name*	*branch-name*
Jones	Johnson	Perryridge
Smith	Johnson	Perryridge
Hayes	Johnson	Perryridge
Jackson	Johnson	Perryridge
Curry	Johnson	Perryridge
Turner	Johnson	Perryridge

Tabular Representation of $\alpha \rightarrow\rightarrow \beta$

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

Relation *bc*: An Example of Reduncy in a BCNF Relation

loan-number	*customer-name*	*customer-street*	*customer-city*
L-23	Smith	North	Rye
L-23	Smith	Main	Manchester
L-93	Curry	Lake	Horseneck

An Illegal *bc* Relation

loan-number	*customer-name*	*customer-street*	*customer-city*
L-23	Smith	North	Rye
L-27	Smith	Main	Manchester

Decomposition of *loan-info*

branch-name	*loan-number*
Round Hill	L-58

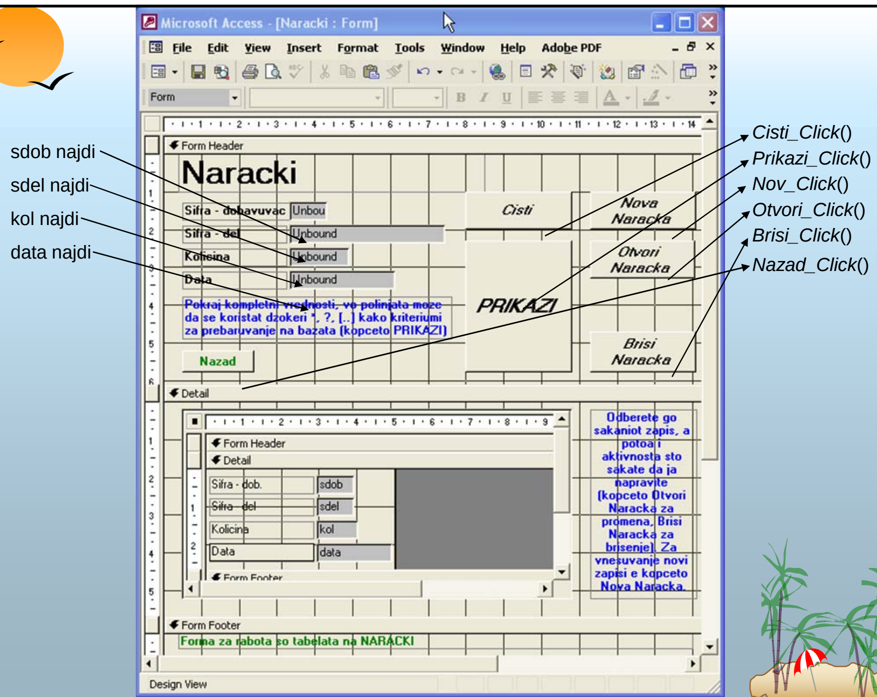
loan-number	*amount*

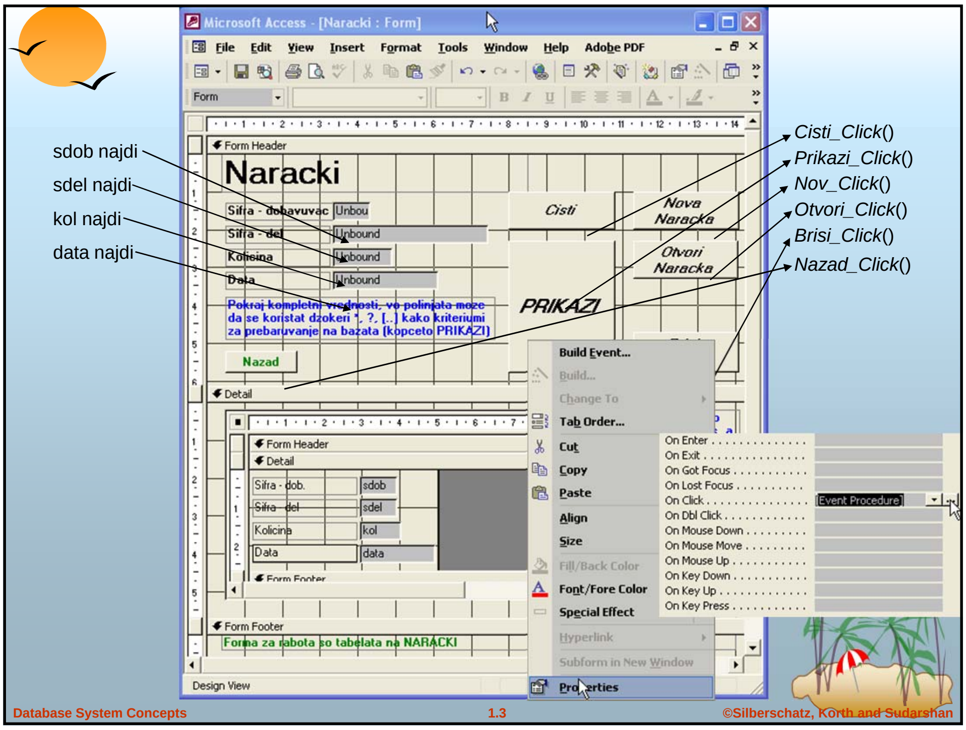
loan-number	*customer-name*
L-58	Johnson

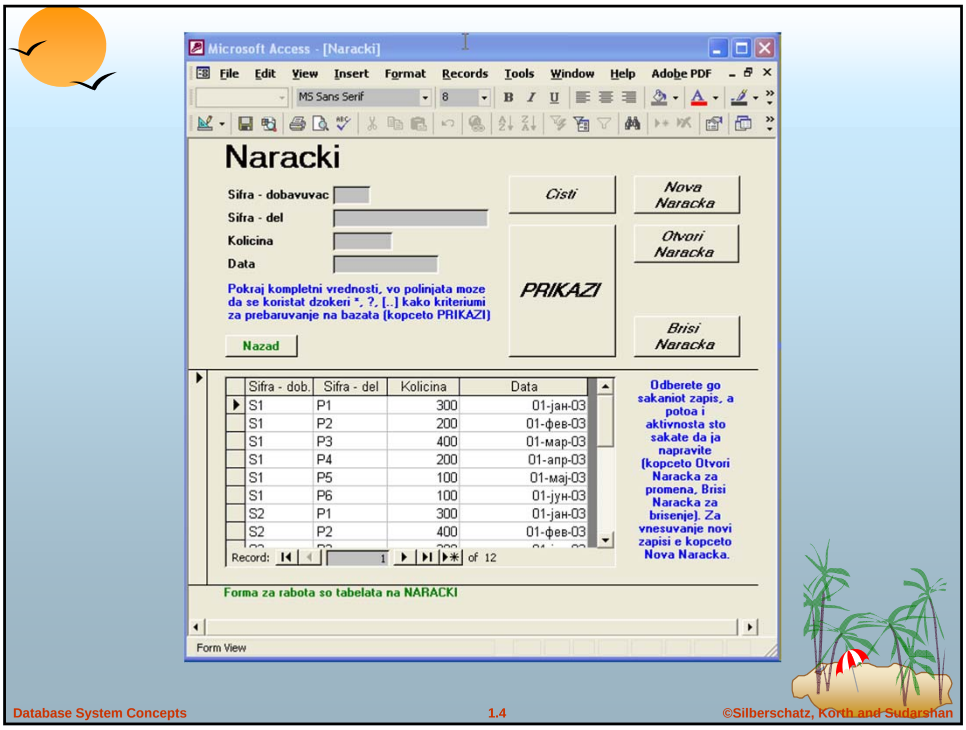
Relation of Exercise 7.4

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

A Form with Visual Basic




Microsoft Access - [Naracki : Form]
sdob najdi
sdel najdi
kol najdi
data najdi
Cisti_Click()
Prikazi_Click()
Nov_Click()
Otvori_Click()
Brisi_Click()
Nazad_Click()
Naracki
Cisti
Nova Naracka
Otvori Naracka
PRIKAZI
Nazad
Pokraj kompletni vrednosti, vo polinjata moze da se koristat dzokeri *, ?, [..] kako kriteriumi za prebaruvanje na bazata (kopceto PRIKAZI)
Forma za rabota so tabelata na NARACKI
Build Event...
Tab Order...
Cut
Copy
Paste
Align
Size
Font/Fore Color
Special Effect
Properties
On Enter
On Exit
On Got Focus
On Lost Focus
On Click
On Dbl Click
On Mouse Down
On Mouse Move
On Mouse Up
On Key Down
On Key Up
On Key Press
Event Procedure
Design View
Database System Concepts
1.3
©Silberschatz, Korth and Sudarshan


Microsoft Access - [Naracki]
Naracki
Sifra - dobavuvac
Sifra - del
Kolicina
Data
Cisti
Nova Naracka
Otvori Naracka
PRIKAZI
Brisi Naracka
Pokraj kompletni vrednosti, vo polinjata moze da se koristat dzokeri *, ?, [..] kako kriteriumi za prebaruvanje na bazata (kopceto PRIKAZI)
Nazad
Sifra - dob. Sifra - del Kolicina Data
S1 P1 300 01-јан-03
S1 P2 200 01-фев-03
S1 P3 400 01-мар-03
S1 P4 200 01-апр-03
S1 P5 100 01-мај-03
S1 P6 100 01-јун-03
S2 P1 300 01-јан-03
S2 P2 400 01-фев-03
Record: 1 of 12
Odberete go sakaniot zapis, a potoa i aktivnosta sto sakate da ja napravite (kopceto Otvori Naracka za promena, Brisi Naracka za brisenje). Za vnesuvanje novi zapisi e kopceto Nova Naracka.
Forma za rabota so tabelata na NARACKI
Form View
Database System Concepts
1.4
©Silberschatz, Korth and Sudarshan

Procedure for a Query Building

```
Option Compare Database
Option Explicit
Private Sub AddToWhere(FieldValue As Variant, FieldName As String,
    MyCriteria As String, ArgCount As Integer)
  ' Create criteria for WHERE clause.
  If FieldValue <> "" Then
    ' Add "and" if other criterion exists.
    If ArgCount > 0 Then
       MyCriteria = MyCriteria & " and "
    End If

   ' Append criterion to existing criteria.
   ' Enclose FieldValue and asterisk in quotation marks.
   MyCriteria = (MyCriteria & FieldName & " Like " & Chr(39) & Chr(42) &
  FieldValue & Chr(42) & Chr(39))

    ' Increase argument count.
    ArgCount = ArgCount + 1
  End If
End Sub
```

Chr(34) = “
Chr(39) = ‘
Chr(42) = *

Procedure for Field Cleaning in a Form

```
Private Sub Cisti_Click()
  Dim MySQL As String
  Dim Tmp As Variant
    MySQL = "SELECT * FROM NajdiNaracka WHERE False"
    ' Clear search text boxes.
    Me![sdob najdi] = Null
    Me![sdel najdi] = Null
    Me![kol najdi] = Null
    Me![data najdi] = Null
    ' Reset subform's RecordSource property to remove records.
    Me![Naracka subform].Form.RecordSource = MySQL

   ' Move insertion point to Look For Company text box.
   Me![sdob najdi].SetFocus
  ‘ Exit_Cisti_Click:
  '   Exit Sub

 ‘ Err_Cisti_Click:
 '   MsgBox "Greska-->" & Err.Description, vbInformation, "Greska"
 '  Resume Exit_Cisti_Click
End Sub
```

Procedure for Searching in Database

```
Private Sub Prikazi_Click()
On Error GoTo Err_Prikazi_Click
   Dim MySQL As String, MyCriteria As String, MyRecordSource As String
   Dim ArgCount As Integer
   Dim Tmp As Variant
    '  Initialize argument count.
    ArgCount = 0
    '  Initialize SELECT statement.
    MySQL = "SELECT * FROM NajdiNaracka WHERE "
    MyCriteria = ""
    '  Use values entered in text boxes in form header to create criteria for WHERE clause.
    AddToWhere [sdob najdi], "[NajdiNaracka.sdob]", MyCriteria, ArgCount
    AddToWhere [sdel najdi], "[NajdiNaracka.sdel]", MyCriteria, ArgCount
    AddToWhere [kol najdi], "[NajdiNaracka.kol]", MyCriteria, ArgCount
    AddToWhere [data najdi], "[NajdiNaracka.data]", MyCriteria, ArgCount

    '  If no criterion specifed, return all records.
    If MyCriteria = "" Then
       MyCriteria = "True"
    End If

    '  Create SELECT statement.
    MyRecordSource = MySQL & MyCriteria & " ORDER BY sdob"
```


Continues

```
'  Set RecordSource property of Find Customers Subform.
    Me![Naracka subform].Form.RecordSource = MyRecordSource

    '  If no records match criteria, display message.
    '  Move focus to Clear button.
    If Me![Naracka subform].Form.RecordsetClone.RecordCount = 0 Then
        MsgBox "Nema zapisi! ", 48, "Greska"
        Me!Cisti.SetFocus
    Else
         'Enable control in detail section.
      ' Me.Section(acDetail).Enabled = True
       'Tmp = EnableControls("Detail", True)
        '  Move insertion point to Find Customers Subform.
        Me![Naracka subform].SetFocus
    End If

Exit_Prikazi_Click:   Exit Sub

Err_Prikazi_Click:
   MsgBox "Greska-->" & Err.Description, vbInformation, "Greska"
  Resume Exit_Prikazi_Click
End Sub
```

Do almost Nothing

```
Private Sub Form_Activate()
   ' Used by Solutions to show toolbar that includes Show Me button.
   ' Hide built-in Form View toolbar.
   ' Show Custom Form View toolbar.
   '   DoCmd.ShowToolbar "Form View", A_TOOLBAR_NO
   '  DoCmd.ShowToolbar "Custom Form View", A_TOOLBAR_YES
End Sub

Private Sub Form_Deactivate()
   ' Used by Solutions to hide toolbar that includes Show Me button.
   ' Hide Custom Form View toolbar.
   ' Show built-in Form View toolbar.
   '   DoCmd.ShowToolbar "Custom Form View", A_TOOLBAR_NO
   '  DoCmd.ShowToolbar "Form View", A_TOOLBAR_WHERE_APPROP
End Sub

Private Sub Form_Open(Cancel As Integer)
   ' Move insertion point to sdob when form is opened.
   Me![sdob najdi].SetFocus
End Sub
```


Levels of Abstraction

```
Private Sub Nov_Click()
On Error GoTo Err_Nov_Click
   Dim stDocName As String
   Dim stLinkCriteria As String
     stDocName = "Naracka"
     DoCmd.OpenForm stDocName, , , stLinkCriteria, acFormAdd
Exit_Nov_Click:   Exit Sub
Err_Nov_Click:
   MsgBox Err.Description
   Resume Exit_Nov_Click
End Sub
```

```
Private Sub Otvori_Click()
On Error GoTo Err_Otvori_Click
   Dim stDocName As String
   Dim stLinkCriteria As String
   stDocName = "Naracka"
   If IsNull(Me![Naracka subform].Form![sdob]) Then
   MsgBox "-->Nemate sifra za ovoj zapis!", vbInformation, "Greska"
   Else
   stLinkCriteria = "[sdob]='" & Me![Naracka subform].Form![sdob] & "'"
   DoCmd.OpenForm stDocName, , , stLinkCriteria, acFormEdit
   End If
Exit_Otvori_Click:      Exit Sub
Err_Otvori_Click:
   MsgBox "Greska-->" & Err.Description, vbInformation, "Greska"
   Resume Exit_Otvori_Click
End Sub
```

Instances and Schemas

```
Private Sub Brisi_Click()
On Error GoTo Err_Brisi_Click

    Dim MySQL As String, MyCriteria As String

    MySQL = "DELETE FROM Naracka WHERE sdob LIKE "
    If IsNull(Me![Naracka subform].Form![sdob]) Then
    MsgBox "-->Nemate izbrano zapis!", vbInformation, "Greska"
    Else
    MySQL = (MySQL & Chr(34) & Me![Naracka subform].Form![sdob] & Chr(34) & " and sdel
       LIKE " & Chr(34) & Me![Naracka subform].Form![sdel] & Chr(34))
    DoCmd.RunSQL MySQL
    Prikazi_Click
    End If

Exit_Brisi_Click:    Exit Sub

Err_Brisi_Click:
    MsgBox "Greska-->" & Err.Description, vbInformation, "Greska"
    Resume Exit_Brisi_Click
End Sub
```


Procedure for Opening Form

```
Private Sub Nazad_Click()
On Error GoTo Err_Nazad_Click

    Dim stDocName As String
    Dim stLinkCriteria As String
    DoCmd.Close
    stDocName = "GlavnoMeni"
    DoCmd.OpenForm stDocName, , , stLinkCriteria

Exit_Nazad_Click:    Exit Sub

Err_Nazad_Click:
    MsgBox Err.Description
    Resume Exit_Nazad_Click
End Sub
```