DATABASE

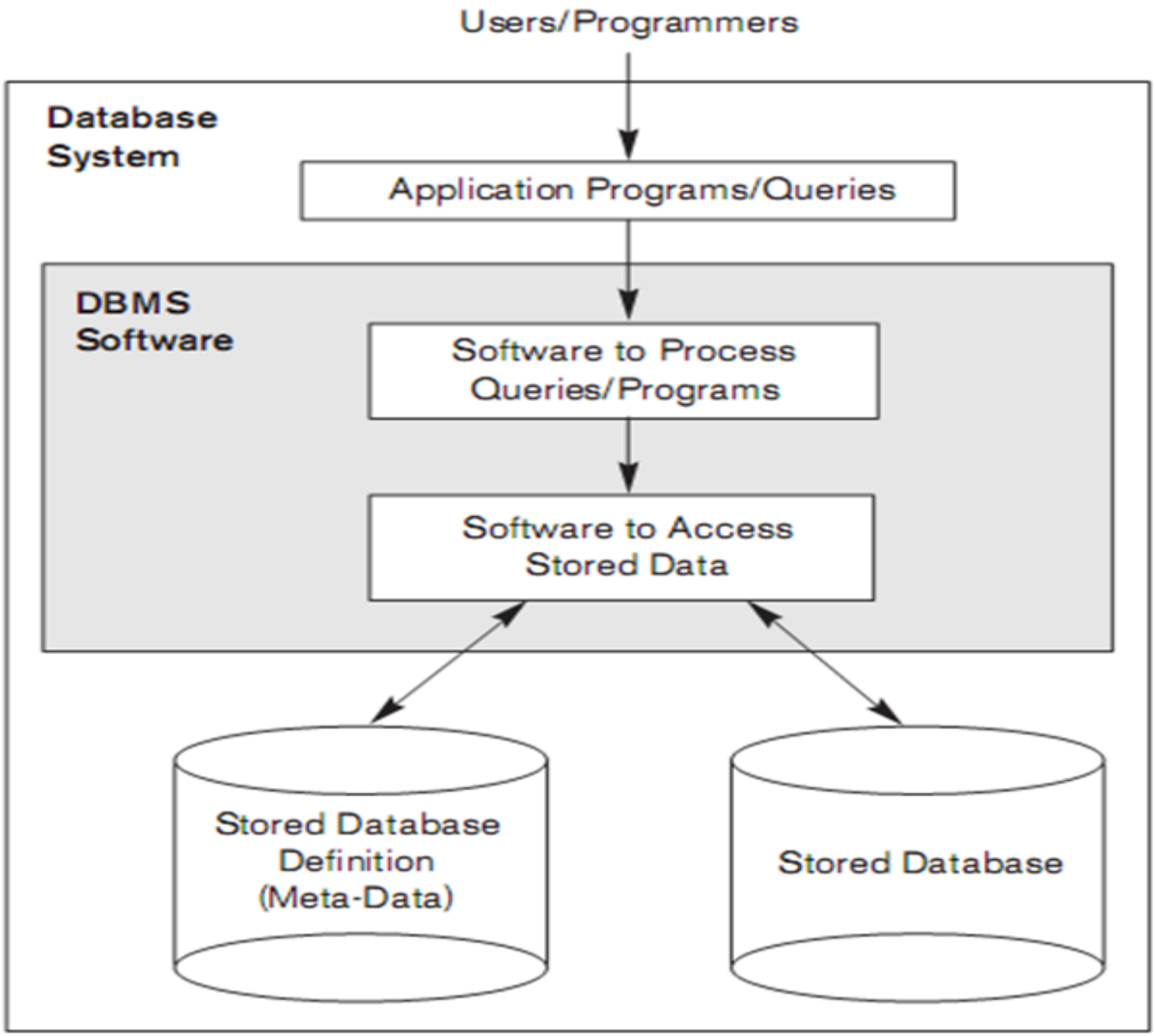
Introduction

- Database has become an essential component of life in modern days
- Eg. Airplane reservation, bank deposit or withdraw, purchase book online,etc..
- All these activities involve some interaction with the database
- In general, Database is a collection of related data. It can be generated & maintained manually or it may be computerised.

Introduction contd…

- A database management system (DBMS) is a collection of programs that enables users to create and maintain a database.
- DBMS includes the following steps:
  - Defining (specifying data types,structure,constraints.)
  - Constructing (process of storing the data on storage medium)
  - Manipulating(querying the database to retrieve specific data , updating the database to reflect changes)
  - Sharing (multiple users & programs access data simultaneously)

Database system environment



Characteristic of Database Approach

- Database Approach Vs Traditional File Processing
- In traditional file processing, each user maintains separate files of record.
- They have to maintain a separate copy of record

  i.e redundant effort in maintaining the record
- In database approach, a single repository maintains data that is defined once and then accessed by various users

Characteristics contd…

- Self-Describing Nature of a Database System
  - a complete definition or description of the database structure and constraints is stored in a catalog called meta-data.
  - DBMS software package is not written for a specific database

Characteristics contd..

- Support of Multiple Views of the Data
- A database typically has many users, each of whom may require a different perspective or view of the database. A view may be a subset of the database or it may contain virtual data that is derived from the database files but is not explicitly stored

- Sharing of Data and Multiuser Transaction Processing
- A fundamental role of multiuser DBMS software is to ensure that concurrent transactions operate correctly and efficiently
- The DBMS must include concurrency control software to ensure that several users trying to update the same data do so in a controlled manner so that the result of the updates is correct

Advantages of dbms

- Controlling Redundancy
- Restricting Unauthorized Access
- Providing Storage Structures and Search techniques for Efficient Query Processing
- Providing Backup and Recovery
- Providing Multiple User Interfaces
- Representing Complex Relationships among Data

Advantages contd..

- Enforcing Integrity Constraints

Problems In File Processing approach

- Data redundancy and inconsistency
  - Same information may be duplicated in several places.
  - All copies may not be updated properly
- Difficulty in accessing data
  - May have to write a new application program to satisfy an unusual request.
  - E.g. find all customers with the same postal code.
  - Could generate this data manually, but a long job...
- Data isolation
  - Data in different files.
  - Data in different formats.
- Multiple users
  - Want concurrency for faster response time.
  - Need protection for concurrent updates.
  - E.g. two customers withdrawing funds from the same account at the same time. account has Rs. 500 in it, and they withdraw Rs. 100 and Rs. 50. The result could be Rs. 350, Rs.400 or Rs. 450 if no protection

Problems In File Processing approach

- Security problems
  - Every user of the system should be able to access only the data they are permitted to see.
  - E.g. payroll people only handle employee records, and cannot see customer accounts; tellers only access account data and cannot see payroll data.
  - Difficult to enforce this with file processing programs.
- Integrity problems
  - Data may be required to satisfy constraints.
  - E.g. no account balance below Rs 25.00.
  - Again, difficult to enforce or to change constraints with the file-processing approach.

These problems and others led to the development of database management system.

Data Abstraction

- The major purpose of a database system is to provide users with an *abstract view* of the system.
- The system hides certain details of how data is stored and maintained
- Complexity is hidden from database users.
- These are the levels of abstraction
  - A) Physical Level
  - B) Conceptual Level
  - C) View Level

Data abstraction cont..

- A) Physical Level
  - Lowest level of abstraction.
  - How the data are stored.
    - E.g. index, B-tree, hashing
  - Complex low-level structures described in detail.
- B) Conceptual Level
  - Next highest level of abstraction.
  - Describes what data are stored.
  - Describes the relationships among data.
  - Database administrator level.

Data Abstraction contd…

- C) View Level
  - Highest level.
  - Describes part of the database for a particular group of users.
  - Can be many different views of a database.
  - E.g. tellers in a bank get a view of customer accounts, but not of payroll data.

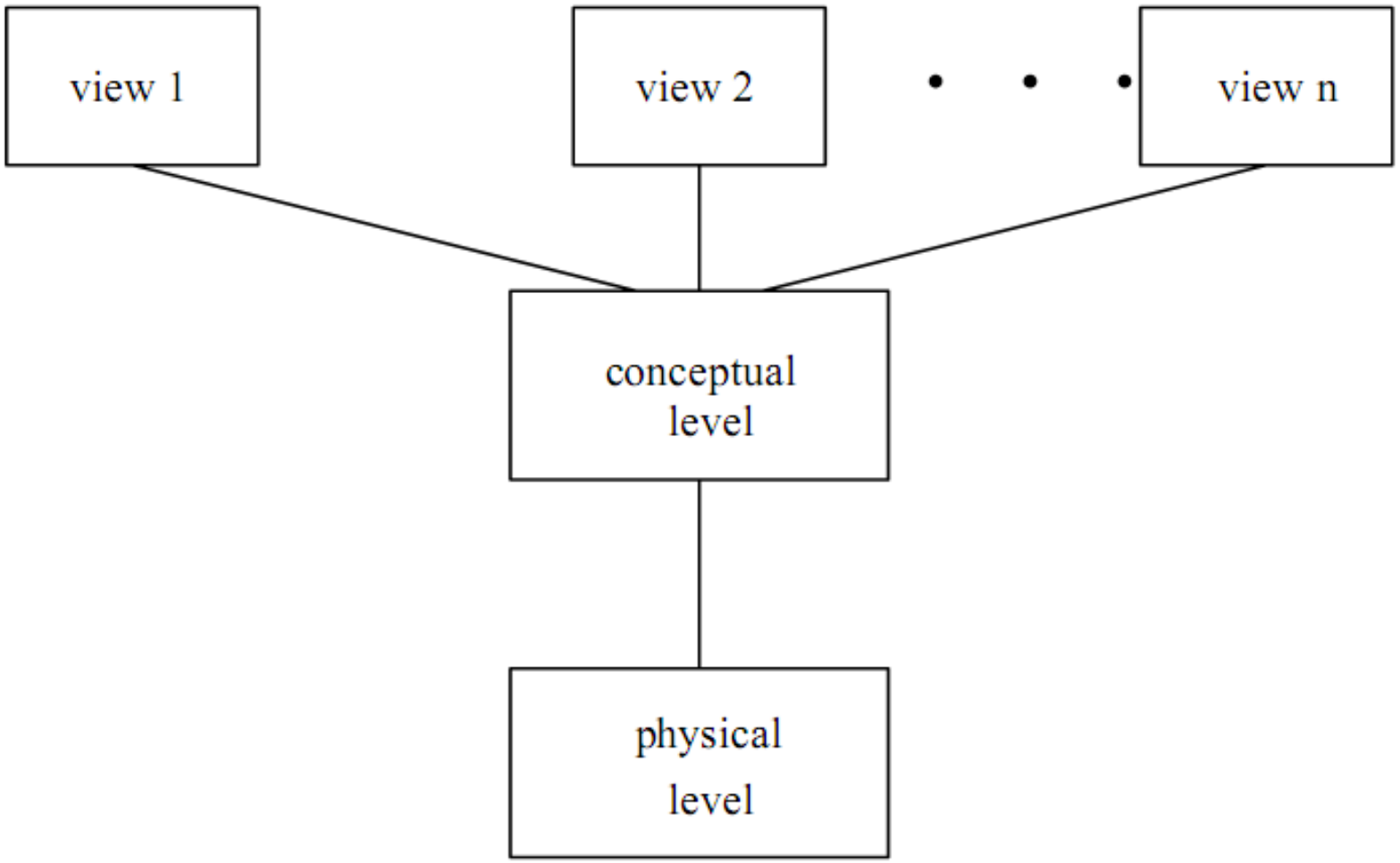


Figure 1.1: The three levels of data abstraction

Data Modelling

- It is the process of creating a representation of the structure of the database. The result of the process is a "*data model*".
- Data models are a collection of conceptual tools for describing data, data relationships, data semantics and data constraints
- So a data model
  - Identifies what is to be stored in the database.
  - It also defines structure and relationship
  - It is the basis for all subsequent database development

Data Model cont..

There are different groups:

- A) Object-based Logical Models.
  - Describe data at the conceptual and view levels.
  - Over 30 models including
    - Entity Relationship model (ER model)
    - Object Oriented model
    - Binary model
    - Semantic data model, etc.
- B)Physical Data Models

  Are used to describe data at the lowest level.
  - Very few models, e.g.
  - Unifying model.
  - Frame memory.
- We will only cover ER Modelling
- We will not cover physical models

E-R Model

- The entity-relationship model is based on a perception of the world as consisting of a collection of basic objects entities and relationships among these objects.

- An entity is a distinguishable object that exists.
- Each entity has associated with it a set of attributes describing it. E.g. number and balance for an account entity.

- A relationship is an association among several entities.
- e.g. A cust acct relationship associates a customer with each account he or she has.

- The set of all entities or relationships of the same type is called the entity set or relationship set.

E-R Model contd..

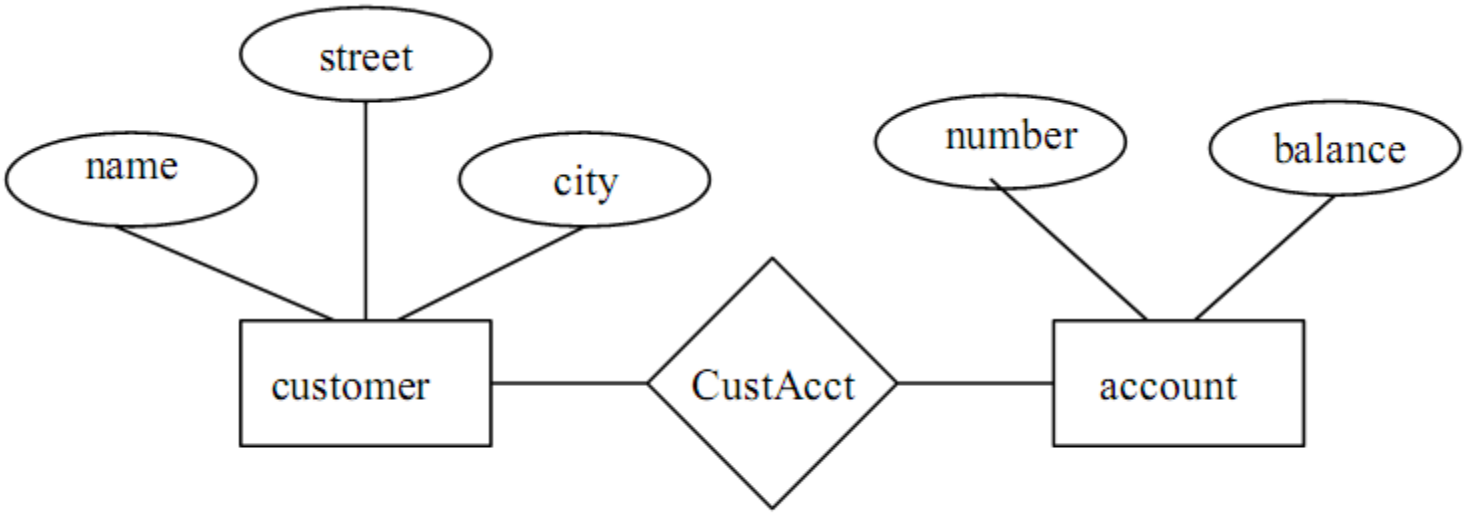


Figure 1.2: A sample E-R diagram.

Instances and Schemes

- Database changes over time
- The information in a database at a particular point in time is called an *instance* of the database.
- The overall design of the database is called the *database scheme.*
- Analogy with programming language:
  - Data type definition – scheme
  - Value of a variable – instance
- There are several schemes, corresponding to levels of abstraction:
  - Physical scheme
  - Conceptual scheme

Data independence

- The ability to allow users to take a logical view of the database which is independent of the way that the data is stored.

- The ability to modify a scheme definition in one level without affecting a scheme definition in a higher level is called data independence.

Summary of previous lect.

- Advantage
- Data Abstraction
- Data Modelling
- Instance & Schema

Data independence

- The ability to modify a scheme definition in one level without affecting a scheme definition in a higher level is called *data independence*.
- They are of 2 types:
  - Physical Data Independence
    - The ability to modify the physical scheme without causing application programs to be rewritten
    - Modifications at this level are usually to improve performance

Data independence cont..

- Logical Data Independence
  - The ability to modify the conceptual scheme without causing application programs to be rewritten
  - Usually done when logical structure of database is altered
  - *harder to achieve* as the application programs are usually heavily dependent on the logical structure of the data

Database Languages

- Database system provide
  - Data Definition Language (DDL)
    - Specify database schema
  - Data Manipulation Language (DML)
    - Database queries & updates
  - DDL & DML are not two separate languages, they are part of a single database language
  - SQL (widely used database language)

DDL Example…

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

- Execution of the above DDL statement creates the account table

Data Manipulation Language

- Data Mainpulation is:
  - *Retrieval , Insertion , Deletion ,Modification*
- DML is a language which enables users to access and manipulate data
- A query is a statement requesting the retrieval of information.
- The portion of DML that involves information retrieval is called *query language*
- DML & query language are often used simultaneouly

DML Example…

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

- This query in SQL Language finds the name of the customer whose customer-id is specified after = equal sign.

What’s Next!

- Database Administrator
- Database Users
- Overall System Architecture….

Database Manager

- A program module which provides the interface between the low-level data stored in the database and the application programs and queries submitted to the system.

- Databases typically require lots of storage space gigabytes. This must be stored on disks. Data is moved between disk and main memory MM as needed.

Database Manager cont..

- Database Manager module is responsible for
  - Interaction with the file manager
    - Storing raw data on disk using file system
  - Integrity enforcement
    - Do not violate consistency constraints
  - Security enforcement
    - Ensure user access to information they are permitted to
  - Backup and Recovery
    - Detecting failures due to power failure, disk crash,etc and restoring the database to its state before the failure
  - Concurrency control
    - Preserving data consistency when there are concurrent users

Database Administrator

- Is person having central control over data and program accessing that data
- Duties of administrator
  - Database design
  - Scheme definition
  - User authorization

Database Users

- Application programmers
  - Computer professionals
  - Interacts with the system through programming commonly using host language (java,c#,etc)
  - Host language compiler generates object code

- Sophisticated users
  - Interacts with the system without writing programs
  - Requests by writing queries in a database query language
  - Query submitted to query processor that breaks DML statement down into instructions for database manager module

Database Users contd…

- Specialized users
  - Are sophisticated users writing special database application programs.
  - Complex database systems (audio/video), Knowledge-based experts systems, etc

- Naive users
  - Unsophisticated users who interacts with the system by using permanent application programs (eg. ATM)

Overall System Structure

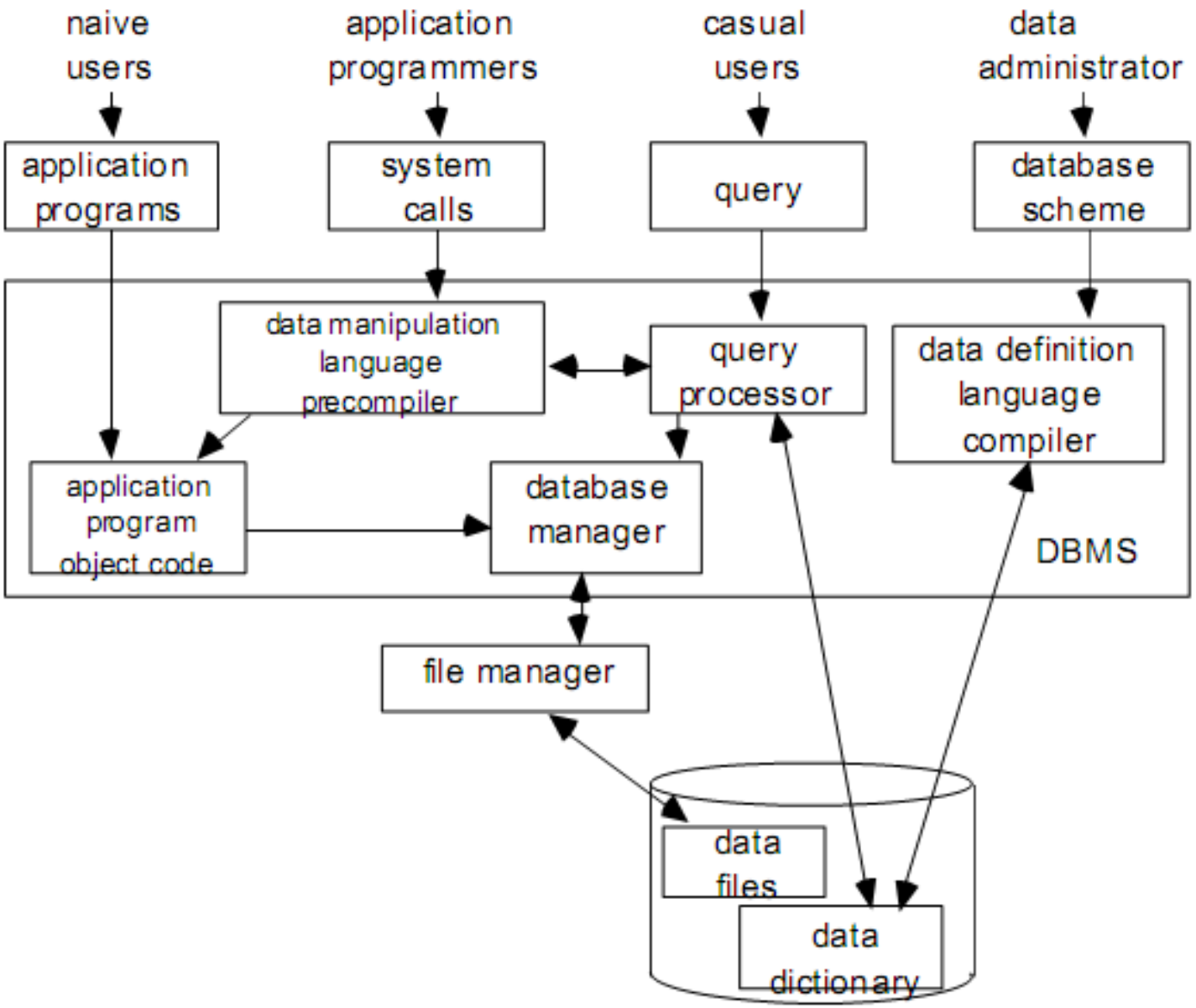
- File Manager
  - Allocation of disk space & data structure on disk
- Database manager
  - Interface between low-level data & application program and queries
- Query Processor
  - Translates statements in a query language to low-level instruction that db manager understands
- DML precompiler
  - Converts DML statement in application program to normal procedure calls in host language

Overall System Structure cont..

- DDL Compiler
  - Converts DDL statements to a set of tables containing metadata stored in a data dictionary

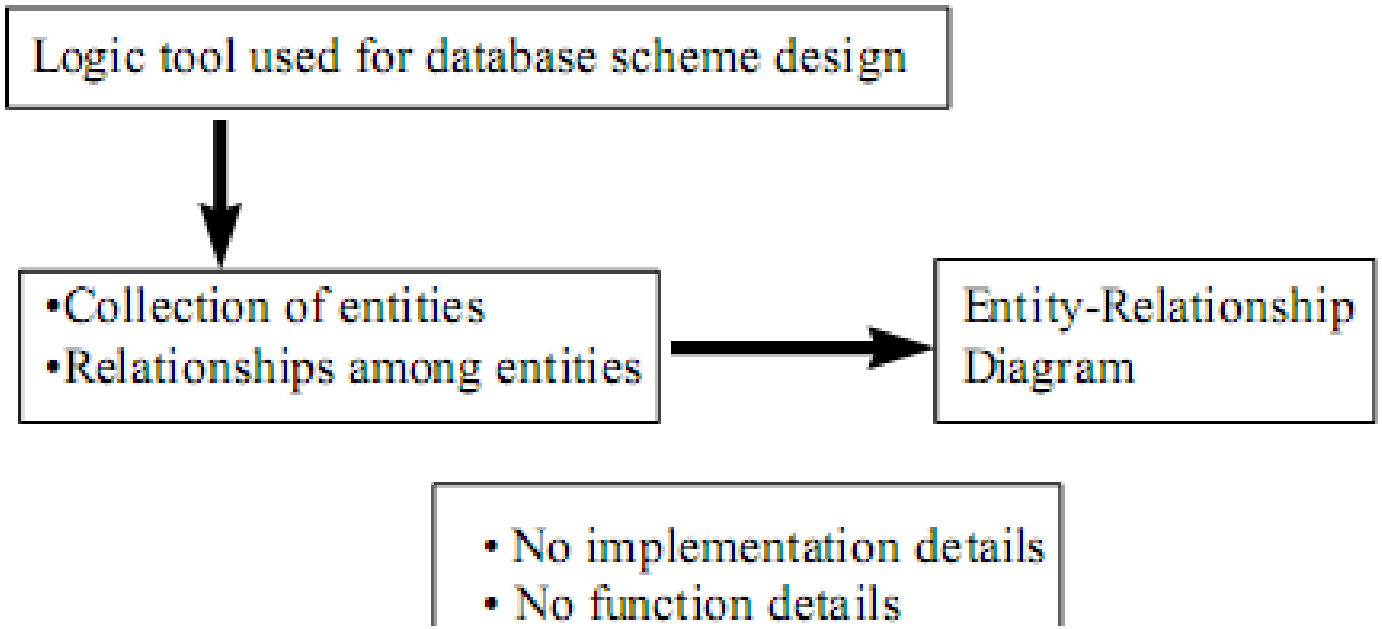
- Data Files – stores the data itself
- Data Dictionary – stores information about the structure of the database. It is used heavily
- Indices – provides fast access to data items holding particular values.

Overall System Structure



Entity-Relationship Model

- E-R Model views the real world as a set of basic objects (entities) and relationship among these objects.
- Represents overall logical structure of the DB



Entity

- An Entity is an object (something) that exists and is distinguishable from other objects (things)
  - Independent existence
  - Described by its attributes (set of properties)
  - Determined by particular value of its attributes
  - Can be concrete or abstract
- Example
  - A book
    - Title="Database system concept",pages=820,ISBN=xxxxxxxx
  - A person
    - Name="Suman Ghimire",age=28,Citizen No=xxxxxx
  - A holiday
    - Name="Christmas",date=December 25

Entity Set

- An entity set is a set of entities of the same type; entities that share the same properties
- Example
  - A set of books
  - A set of people taking DBMS
- Entity set need not be disjoint.
  - For eg. The entity set *employee* all employees of a bank an the entity set *customer* all customers of the bank may have members in common.

Attributes

- An entity is represented by a set of attributes i.e descriptive properties possessed by all members of an entity set
- Example
  - Student (StudentID,Name,Address,Class,Grade)
  - Course(Number,Title,Classroom,instructor)
- Domain
  - Domain of an attribute is a set of permitted values for each attribute
  - Example
    - Attribute Age is a number between 0 and 200
    - Color is one of { blue,green,black,…)

Attribute types

- Single value vs Multi valued
  - Single : Social Insurance Number, student number
  - Multi: Telephone Numbers, Lectures of a course,..
- Simple vs Composite
  - Composite: Address = Apt No, Street, City, Postal Code
- Null Value
  - No corresponding values (contain no value)
- Stored vs Derived
  - Stored: Individual mark of a student
  - Derived: Overall grade of a student

Relationships & Relationship Sets

- A relationship is an association between several entities.
- A relationship set is a set of relationships of the same type.

If $E_1, E_2, \ldots, E_n$ are entity sets, then a relationship set R is a **subset** of

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

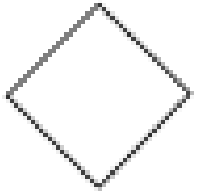
where $(e_1, e_2, \ldots, e_n)$ is a relationship.

- Eg. CustAcc relationship between customer & Account
- CustAcc is a subset of all possible customer & account pairings

Notation

rectangle: entity Set

ellipse: attribute



diamond: relationship

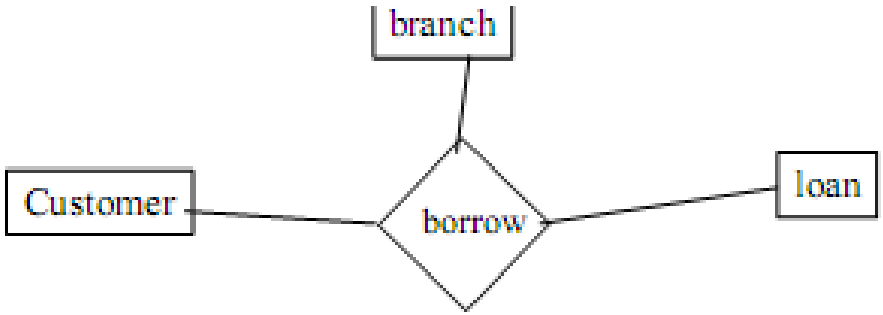
line: link attributes to entity sets and entity sets to relationship sets

double ellipse: multivalued attributes

dashed ellipse: derived attributes

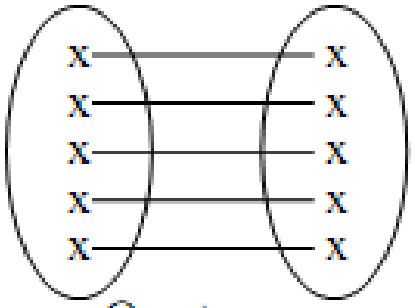
Degree of relationship set

- Refers to the number of entity sets that participate in a relationship set
- Relationship that involve two entity sets are called binary
- Relationship may involve more than 2 entity sets
- The entity set customer, loan and branch may be linked by a ternary relationship.

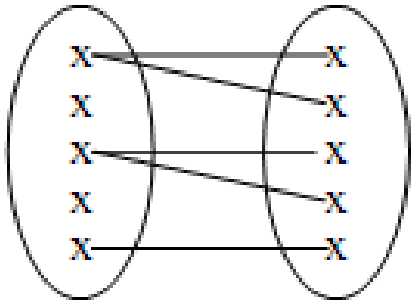


Mapping cardinalities

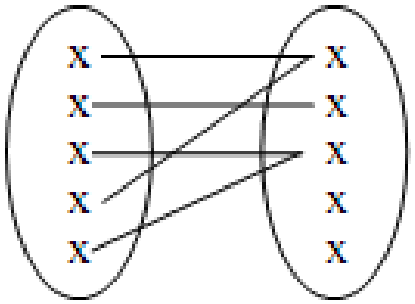
- Express the number of entities to which another entity can be associated via a relationship set
- One to one
- One to many
- Many to one
- Many to many



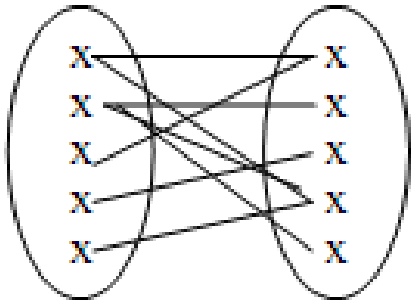
One to one



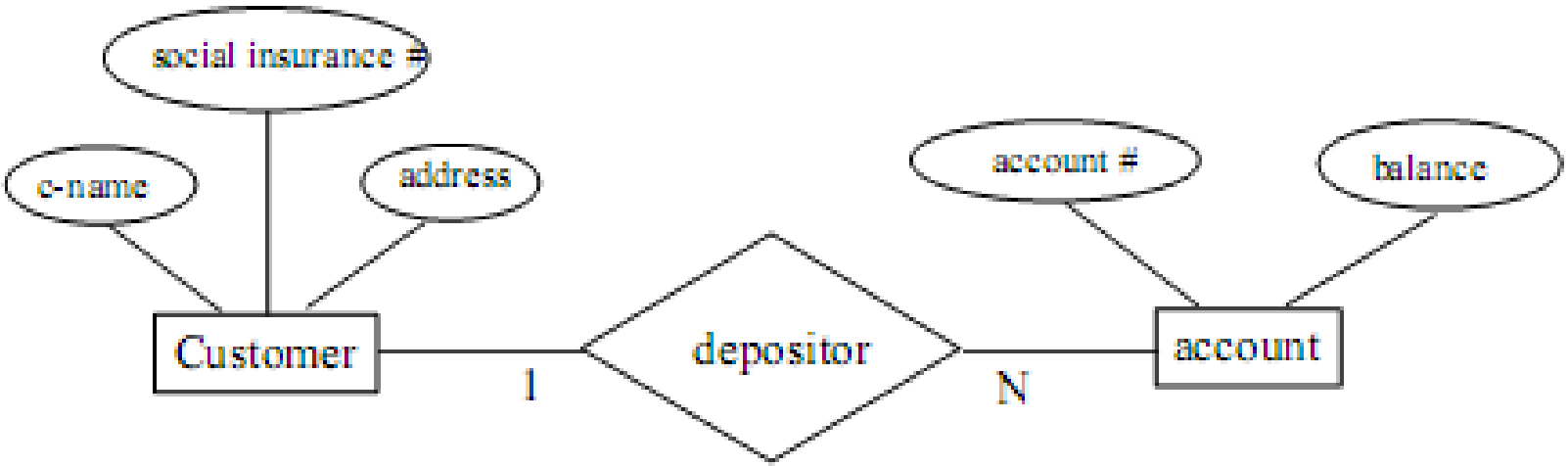
One to many



Many to one



Many to many



1:1 each customer has at most one account and each account is owned by at most one customer.

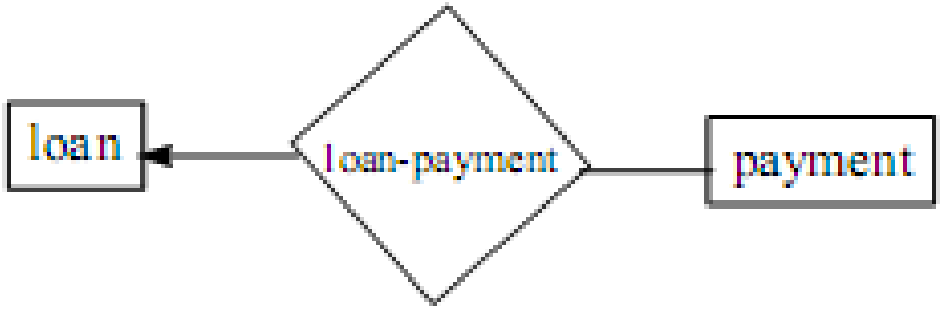
1:N each customer may have any number of accounts but each account is owned by at most one customer.

M:N each customer may have any number of accounts and each account may be owned by any number of customers.

Existence Dependencies

If the existence of an entity **A** depends on the existence of entity **B**, then **A** is said to be ***existence dependent*** on **B**.

- B is called ***dominant*** entity
- A is called ***subordinate*** entity



If a loan entity (dominant) is deleted, then all its associated payment entities (subordinates) must be deleted too.

Entity set Keys

Super keys: Sets of attributes that allow us to identify uniquely an entity in the entity set.

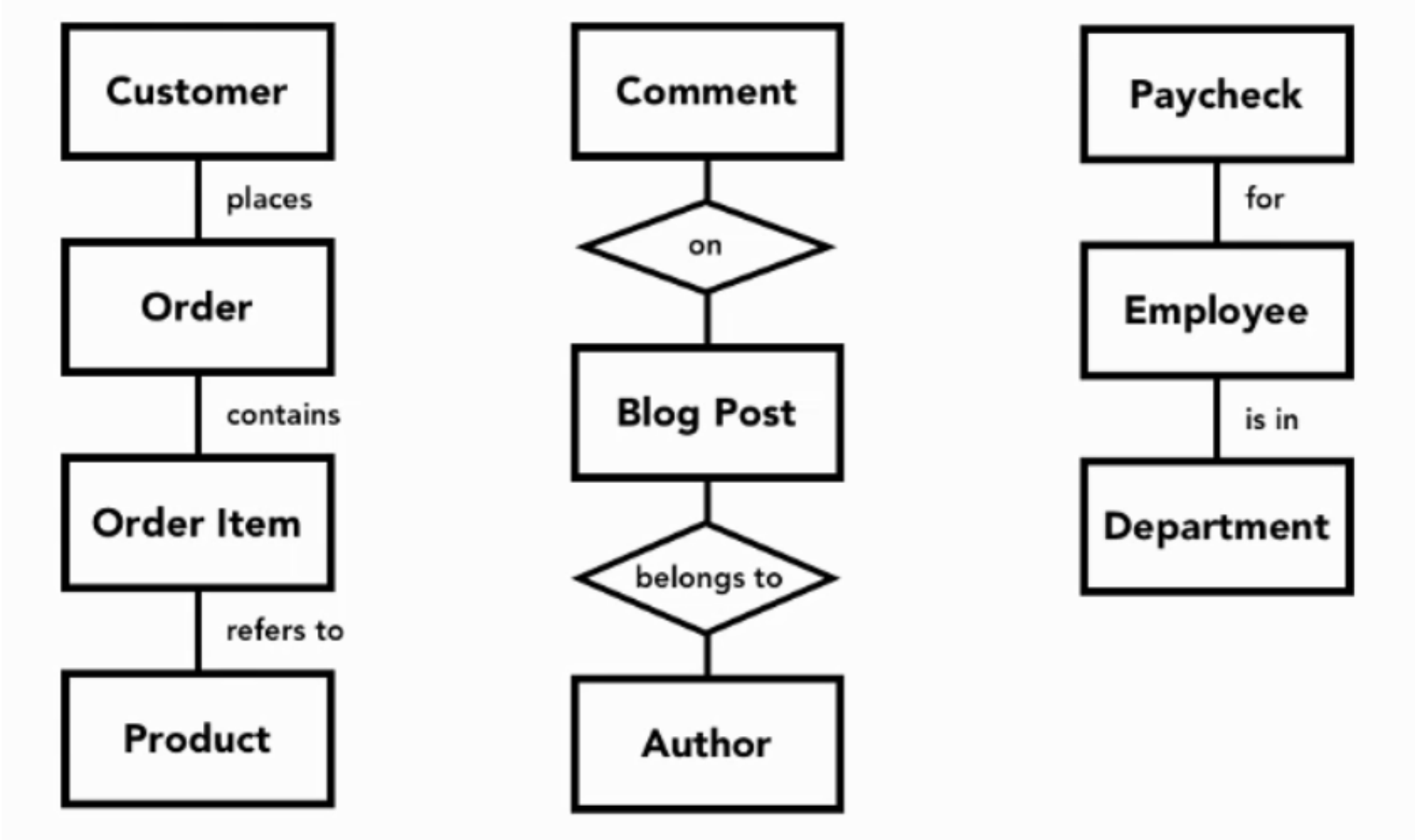
Candidate key: Minimal sets of such attributes.

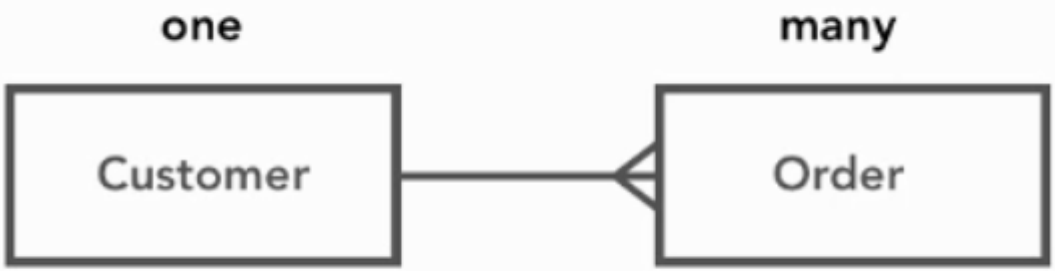
Primary key: Candidate key chosen by database designer.

- For eg. Entity set Customer, customer-name & SIN is a super key.
  - Customer name alone is not, as two customers could have same name
- In above example SIN is a candidate key, as it is minimal, and uniquely identifies a customer entity

- Primary key is a candidate key chosen by the db designer to identify entities in an entity set.
- Weak Entity Sets
  - An entity set that does not have a primary key is referred to as a weak entity set.
- Strong Entity Set
  - One that does have a primary key is called strong entity set

Reducing ER Diagram to table..



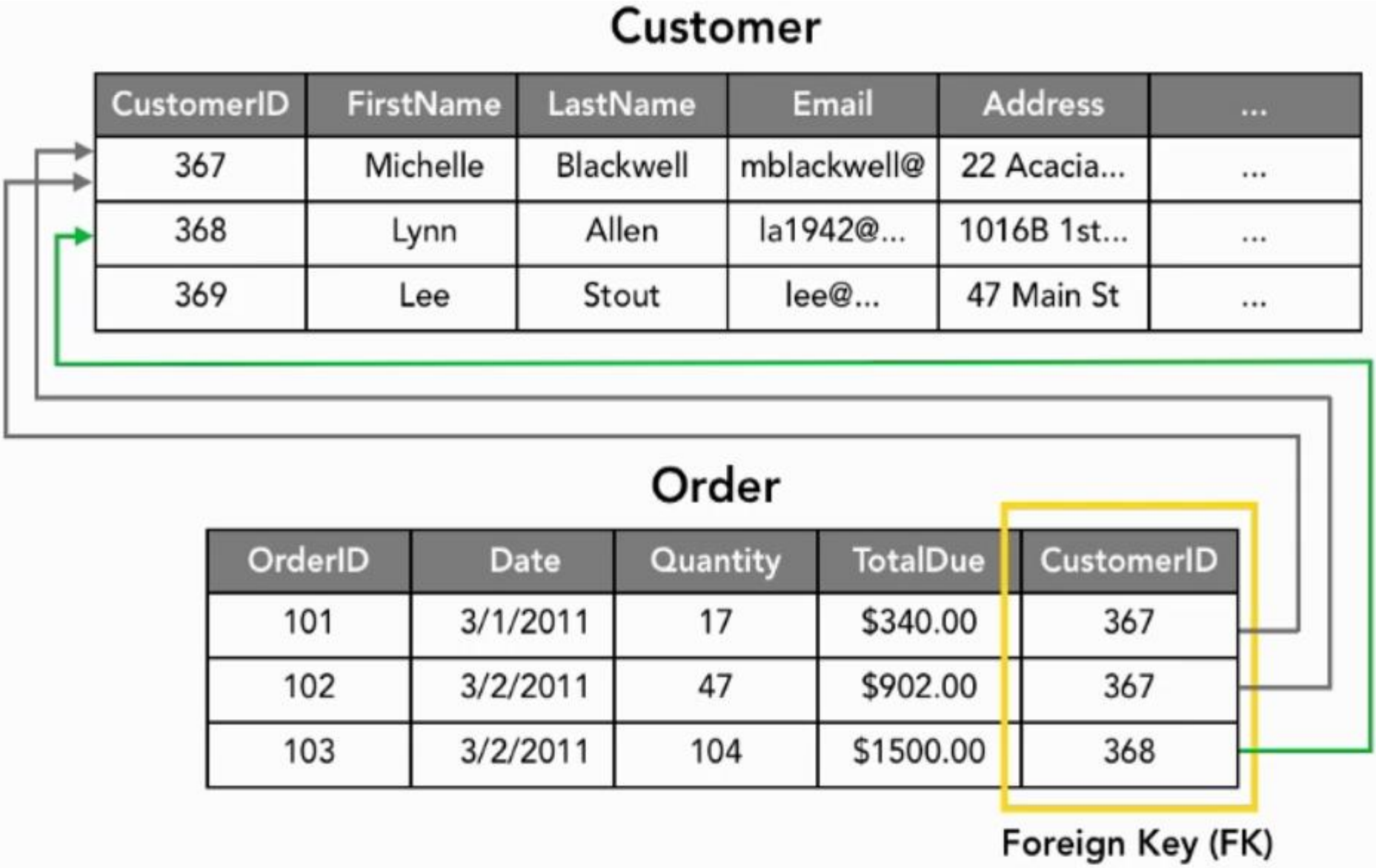


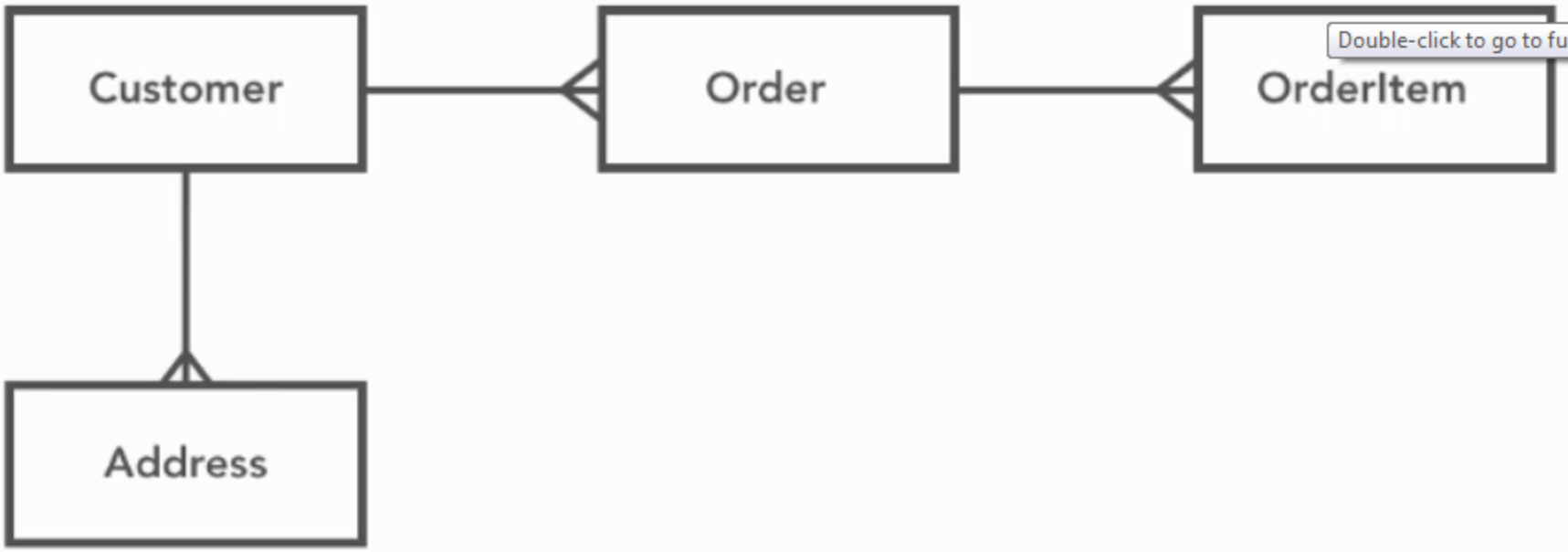
Customer

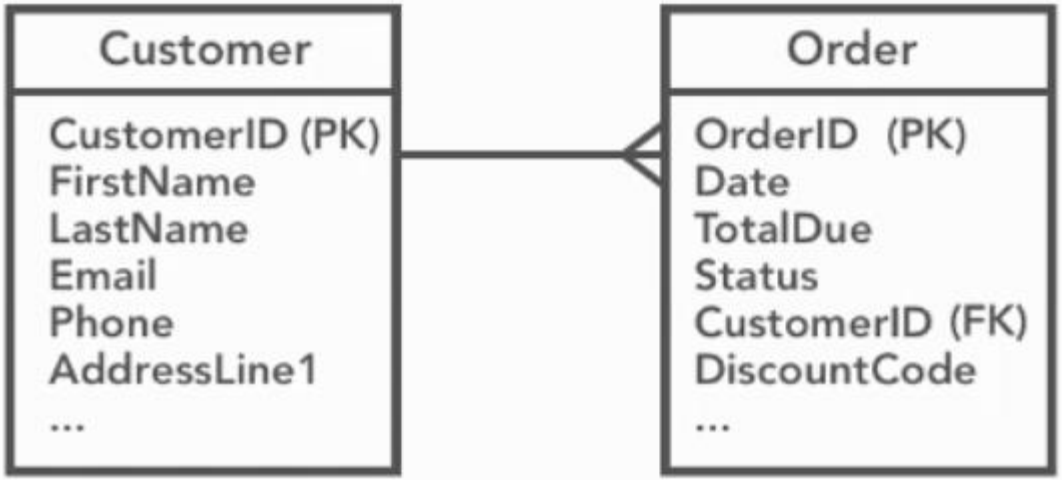
CustomerID	FirstName	LastName	Email	Address	...
367	Michelle	Blackwell	mblackwell@	22 Acacia...	...
368	Lynn	Allen	la1942@...	1016B 1st...	...
369	Lee	Stout	lee@...	47 Main St	...

Order

OrderID	Date	Quantity	TotalDue
101	3/1/2011	17	$340.00
102	3/2/2011	47	$902.00
103	3/2/2011	104	$1500.00


Customer
CustomerID
FirstName
LastName
Email
Address
...
367
Michelle
Blackwell
mblackwell@
22 Acacia...
...
368
Lynn
Allen
la1942@...
1016B 1st...
...
369
Lee
Stout
lee@...
47 Main St
...
Order
OrderID
Date
Quantity
TotalDue
CustomerID
101
3/1/2011
17
$340.00
367
102
3/2/2011
47
$902.00
367
103
3/2/2011
104
$1500.00
368
Foreign Key (FK)

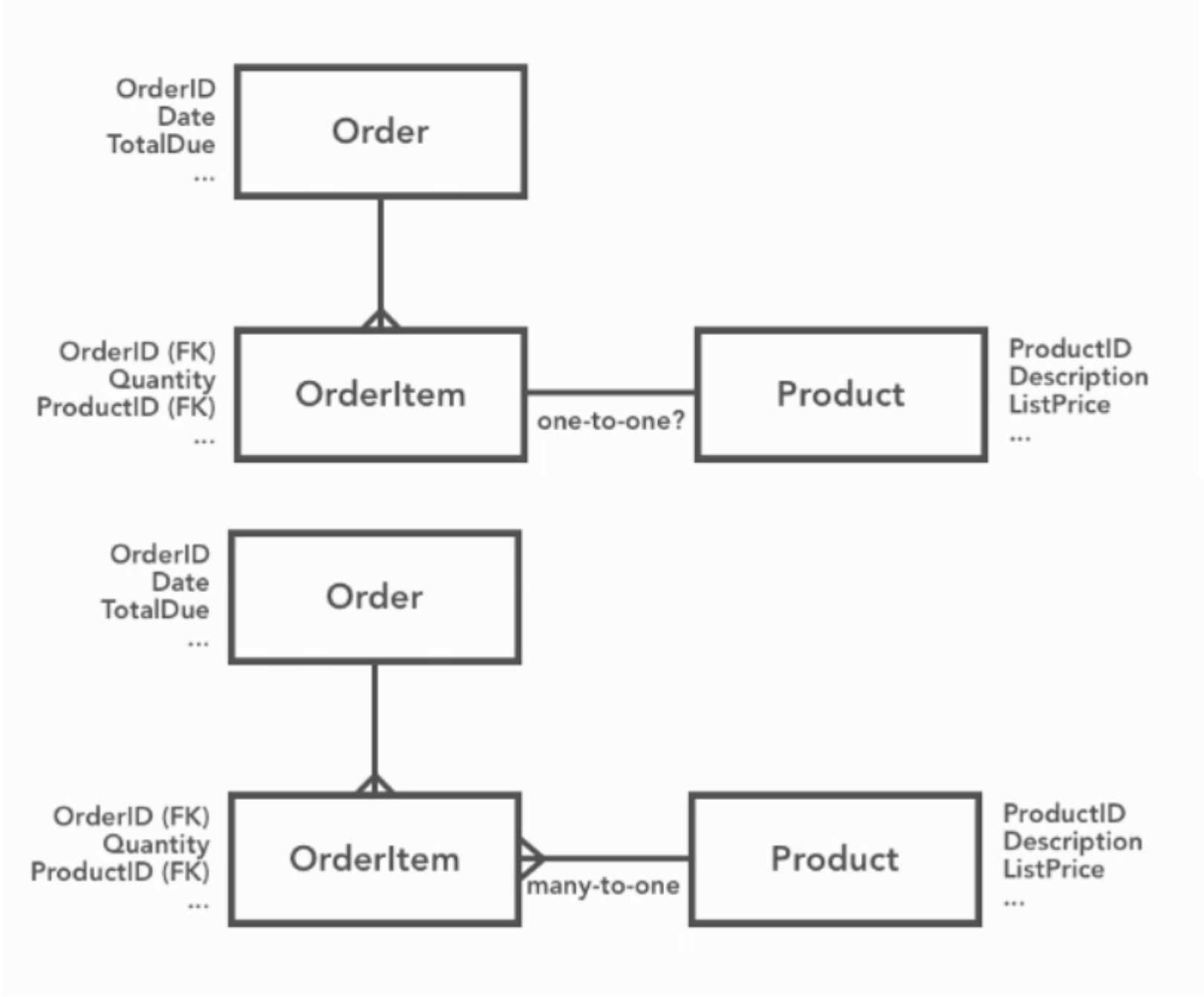




ID	FirstName	LastName	...
101	Jim	McDonald	
102	Marion	Robinson	
103	Angela	Gonzalez	

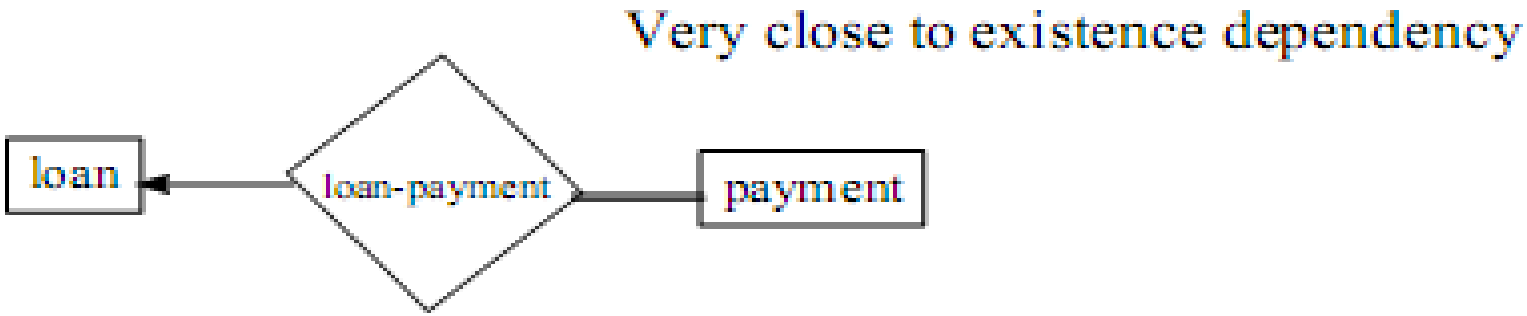
ID	LicenseNumber	State	Expires
101	1243235	AZ	11/2033
102	5434234	CA	11/2014
103	34234521	CA	5/2013

ID	FirstName	LastName	...	LicenseNumber	State	Expires
101	Jim	McDonald		1243235	AZ	11/2033
102	Marion	Robinson		5434234	CA	11/2014
103	Angela	Gonzalez		34234521	CA	5/2013


OrderID
Date
TotalDue
...
Order
OrderID (FK)
Quantity
ProductID (FK)
...
OrderItem
one-to-one?
Product
ProductID
Description
ListPrice
...
OrderID
Date
TotalDue
...
Order
OrderID (FK)
Quantity
ProductID (FK)
...
OrderItem
many-to-one
Product
ProductID
Description
ListPrice
...

Total Participation

- The participation of an entity set E in a relationship R is said to be *total* if every entity in E participates in at least one relationship in R.
- If only some entities in E participate in relationships in R, the participation on entity set E in relationship R is said to be *partial*.



- Notation in E-R diagram: double lines indicate total participation

Extended E-R Features

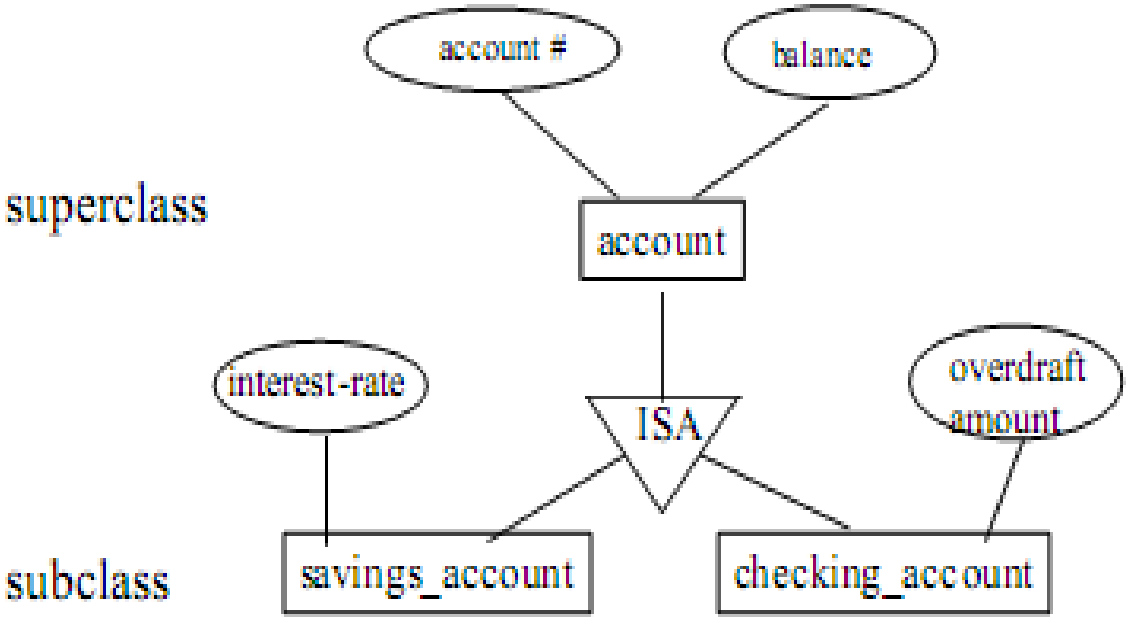
- Specialization
- Generalization
- Attribute inheritance
- Aggregation

Specialization

- An entity set may contain subgroupings of entities
- Such Entity Set is a high-level entity set that can be broken down into one or more lower-level entity set.

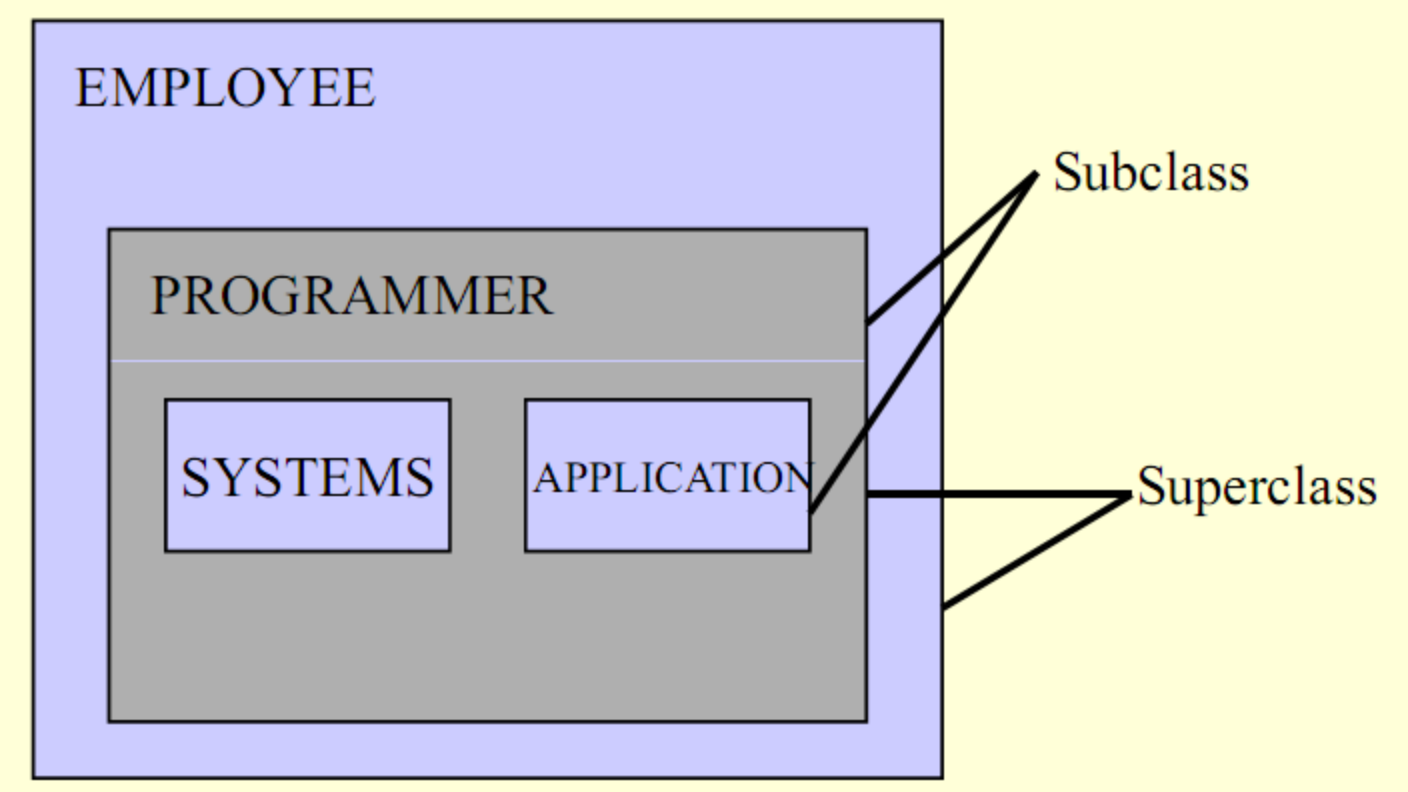
- Example:
  - Course => undergraduate course/ graduate course/ or CMPT/ENG
  - Account => saving account/ fixed deposit account

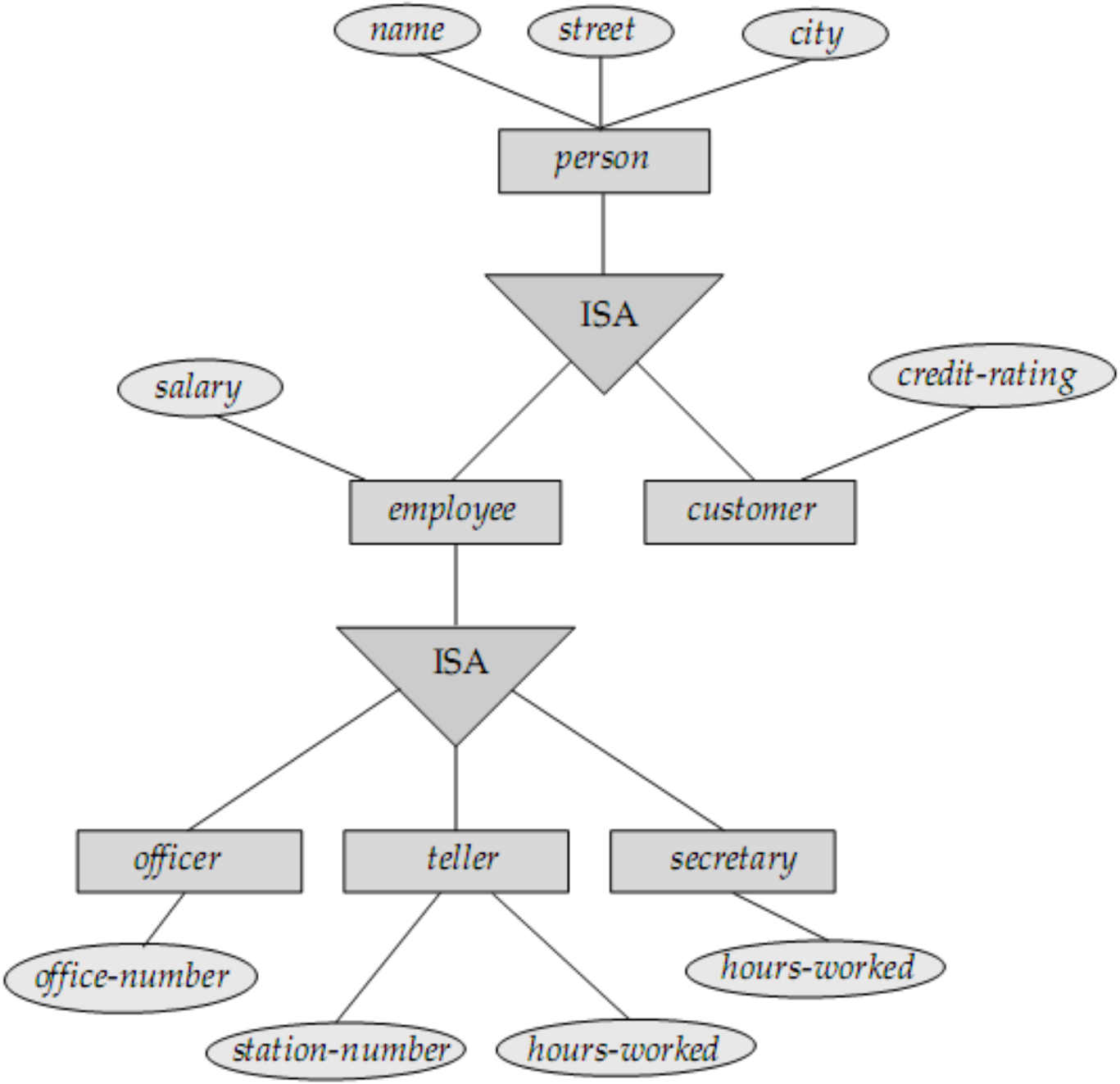
- An entity type E1 is a specialization of another entity type E2 if E1 has the same properties of E2 and perhaps even more
- E1 IS-A E2
- specialization is depicted by a triable labeled ISA.



Analogy to super class & sub class

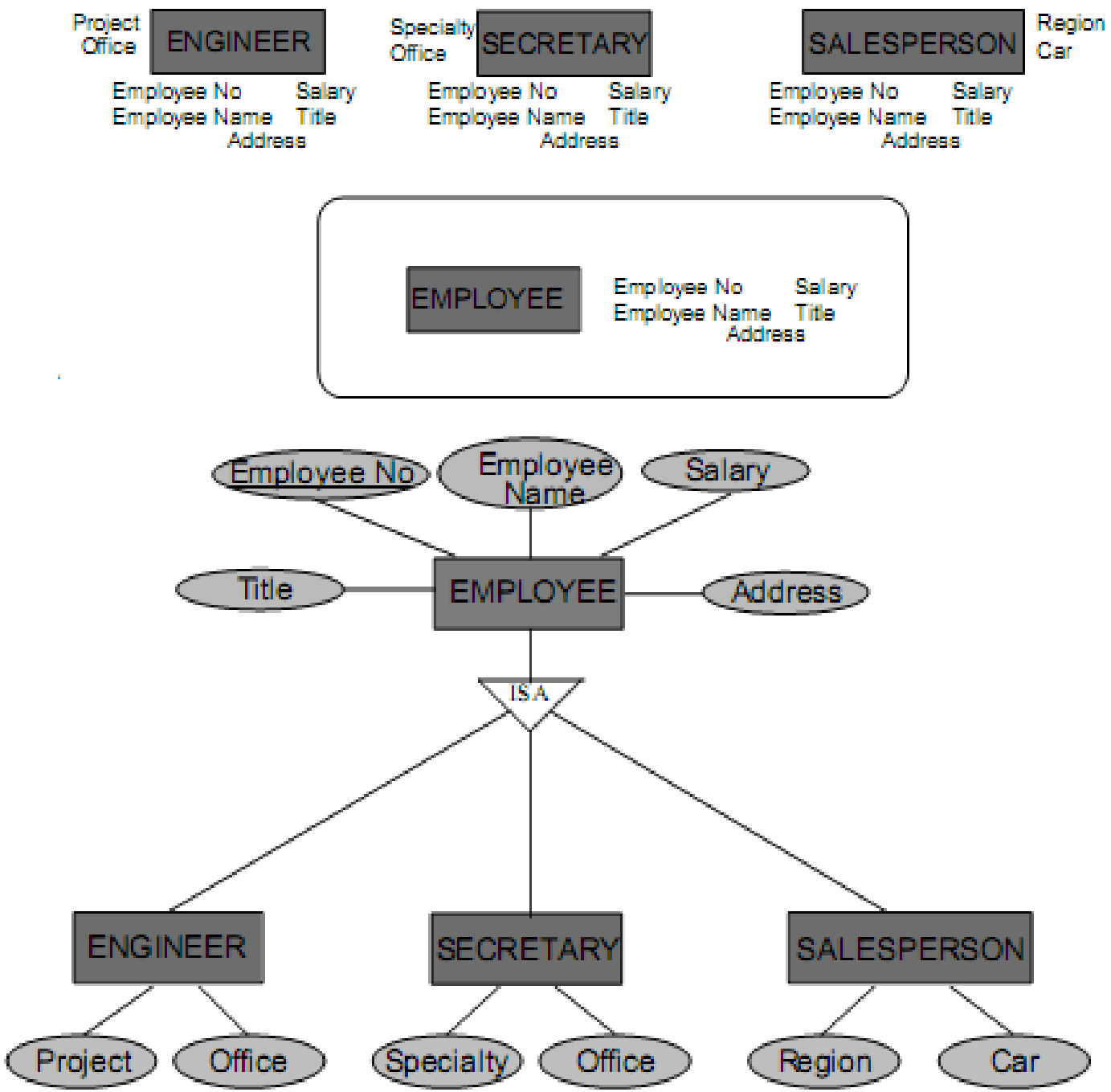
- Systems and Application are Subclasses of Programmer




name
street
city
person
ISA
salary
credit-rating
employee
customer
ISA
officer
teller
secretary
office-number
station-number
hours-worked
hours-worked

Generalization

- Specialization => top-down design
- Generalization => bottom-up design
- Abstracting the common properties of two or more entities to produce a “higher-level” entity.


Project
Office
ENGINEER
Employee No
Employee Name
Address
Salary
Title
Specialty
Office
SECRETARY
Employee No
Employee Name
Address
Salary
Title
SALESPERSON
Employee No
Employee Name
Address
Salary
Title
Region
Car
EMPLOYEE
Employee No
Employee Name
Address
Salary
Title
Employee No
Employee Name
Salary
Title
EMPLOYEE
Address
ISA
ENGINEER
SECRETARY
SALESPERSON
Project
Office
Specialty
Office
Region
Car

Create Table

```
CREATE TABLE Persons
(
Id int NOT NULL PRIMARY KEY,
LastName nvarchar(255) NOT NULL,
FirstName nvarchar(255),
Address nvarchar(255),
City nvarchar(255)
)
```

NOT NULL constraint enforces a column to NOT accept NULL values.

Primary key contraint name (Multiple columns)

```
CREATE TABLE Persons
(
Id int NOT NULL,
LastName varchar(255) NOT NULL,
FirstName varchar(255),
Address varchar(255),
City varchar(255),
CONSTRAINT pk_PersonID PRIMARY KEY
(Id,LastName)
)
```

SQL PRIMARY KEY Constraint on ALTER TABLE

- ALTER TABLE Persons ADD PRIMARY KEY (Id)

- ALTER TABLE Persons
  ADD CONSTRAINT pk_PersonID PRIMARY KEY (Id,LastName)

DROP a PRIMARY KEY Constraint

- ALTER TABLE Persons
  DROP CONSTRAINT pk_PersonID

Foreign Key

```
CREATE TABLE Orders
(
Id int NOT NULL,
OrderNo int NOT NULL,
PId int,
PRIMARY KEY (Id),
CONSTRAINT fk_PerOrders FOREIGN KEY (PId)
REFERENCES Persons(Id)
)
```

```
CREATE TABLE Orders
(
Id int NOT NULL PRIMARY KEY,
OrderNo int NOT NULL,
PId int FOREIGN KEY REFERENCES Persons(Id)
)
```

- ALTER TABLE Orders
  ADD CONSTRAINT fk_PerOrders
  FOREIGN KEY (PId)
  REFERENCES Persons(Id)

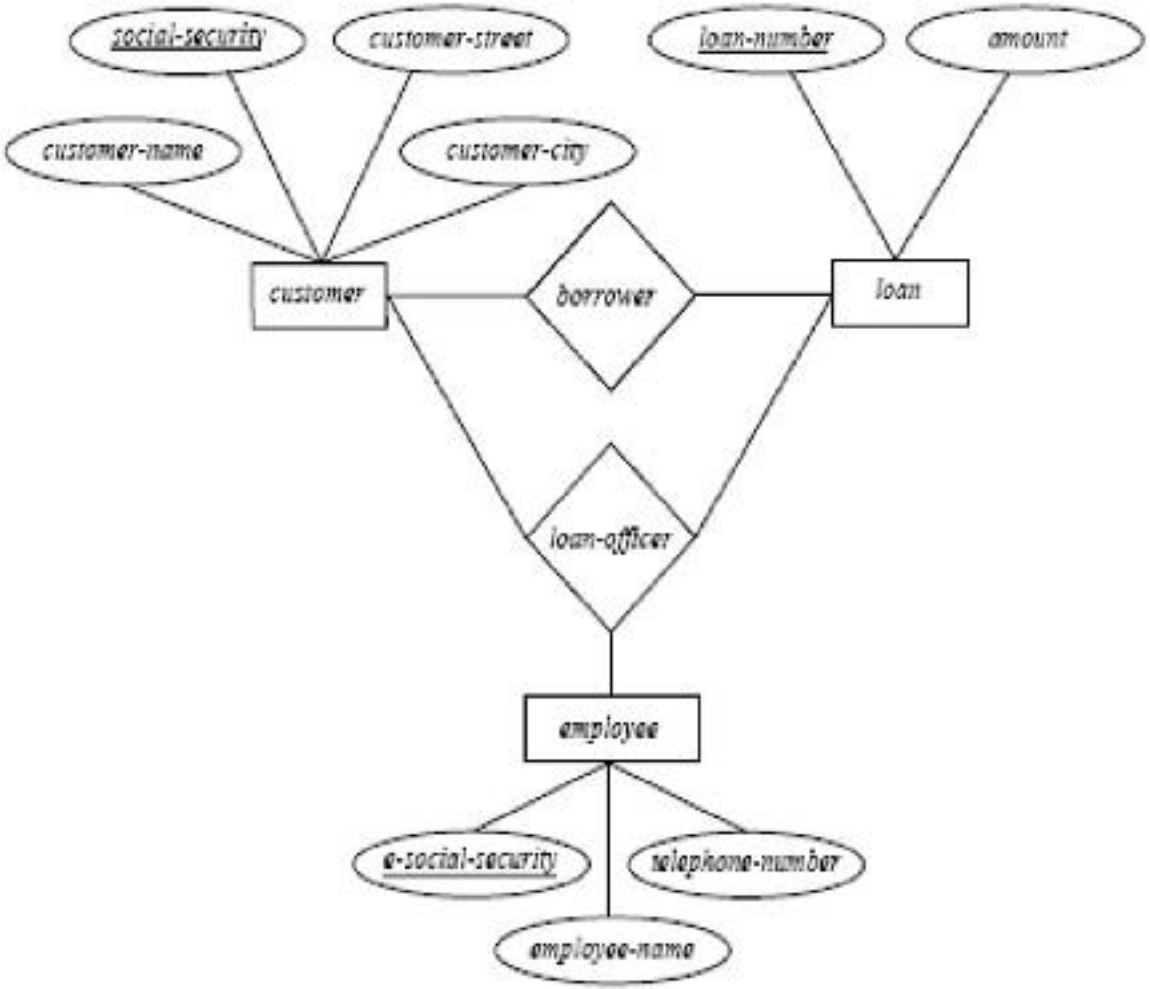
- ALTER TABLE Orders
  DROP CONSTRAINT fk_PerOrders

Sys object

- sp_help “Table-Name”
- SELECT * FROM sys.key_constraints
- SELECT * FROM sys.tables
- SELECT * FROM sys.key_constraints

WHERE [type] = 'PK‘ AND [parent_object_id] = Object_id('dbo.iProductCategory');

Aggregation



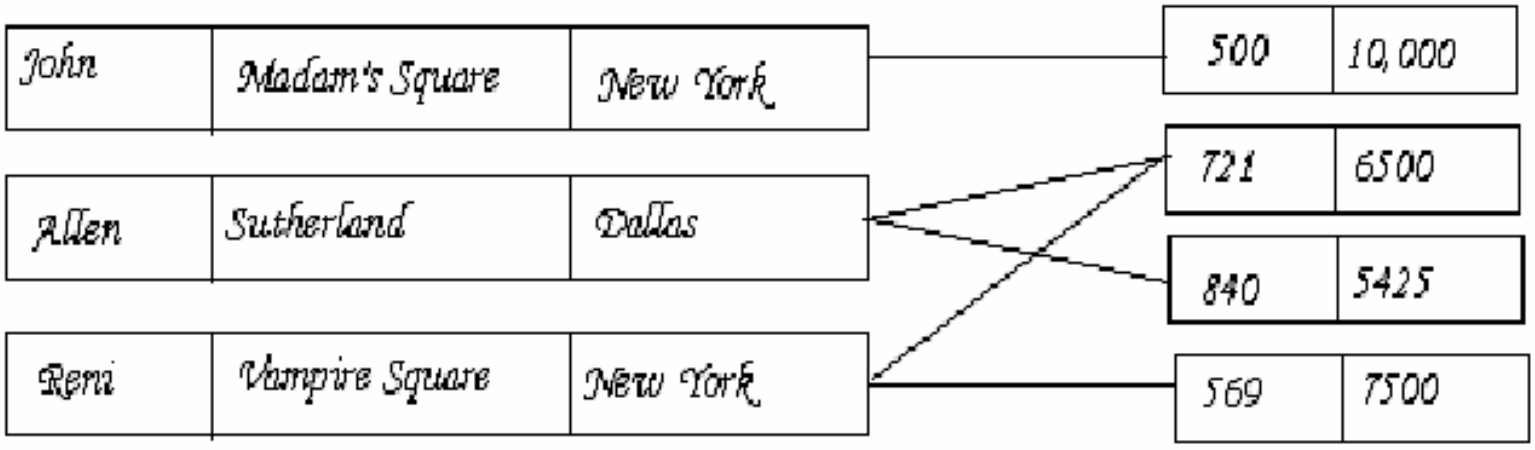
- Relationship sets borrower and loan-officer represent the same information
- - Eliminate this redundancy via aggregation as shown in Figure

- Without introducing redundancy, the above diagram represents that:
- A customer takes out a loan
- An employee may be a loan officer for a customer-loan pair

- Points in Aggregation
  - Treat relationship as an abstract entity.
  - Allows relationships between relationships.
  - Abstraction of relationship into new entity.

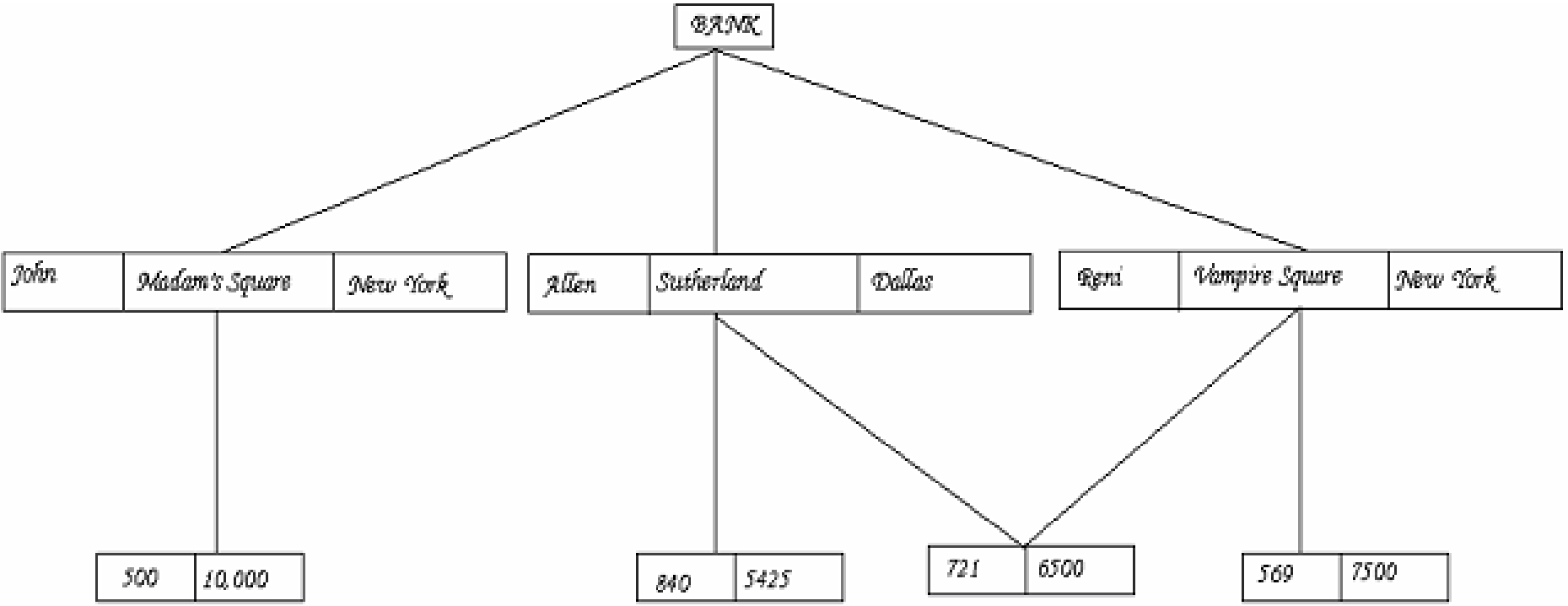
Network Model

- Data are represented by collections of records.
- Relationships among data are represented by links.
- Organization is that of an arbitrary graph and represented by Network diagram.



The Hierarchical Model

- Similar to the network model and the concepts are derived from the earlier systems. (older models)
- Organization of the records is as a collection of trees, rather than arbitrary graphs.



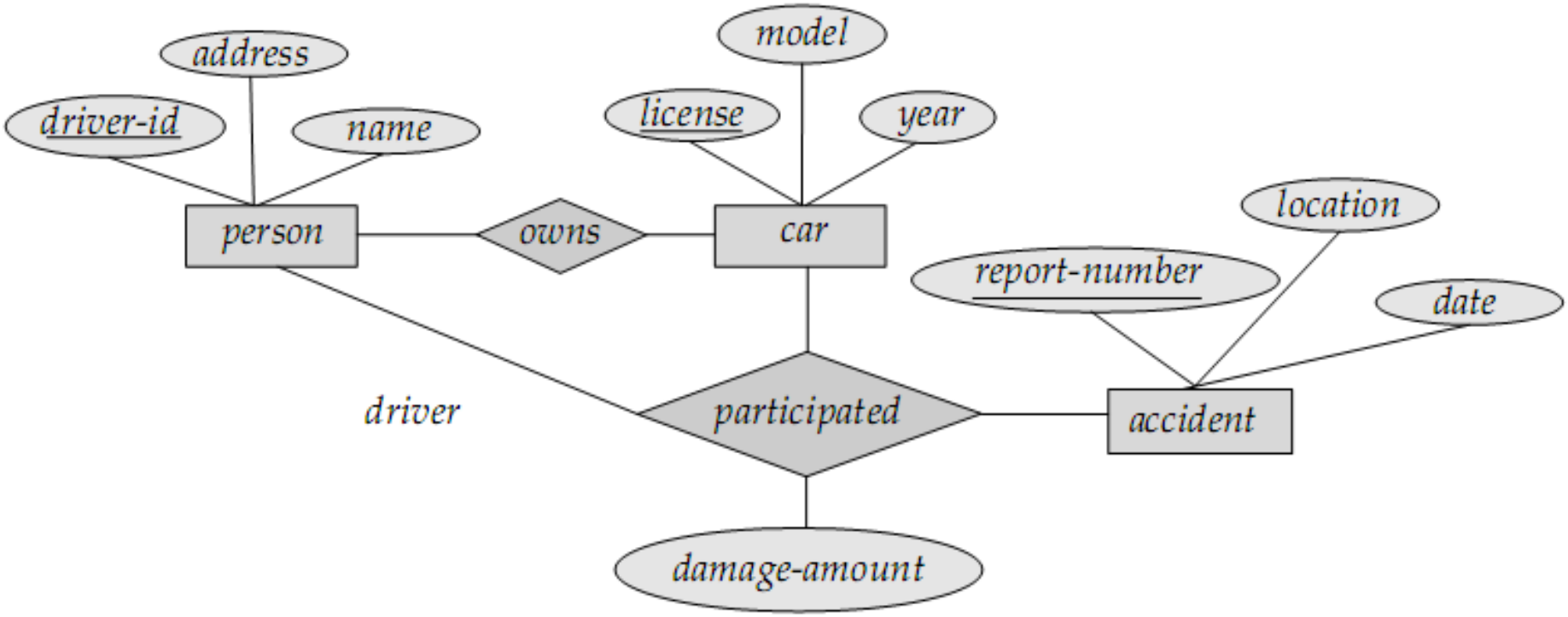
- In the hierarchical model, a Schema is represented by a Hierarchical Diagram
- One record type, called Root, does not participate as a child record type.
- Every record type except the root participates as a child record type in exactly one type.
- Leaf is a record that does not participate in any record types.
- A record can act as a Parent for any number of records.

Class work

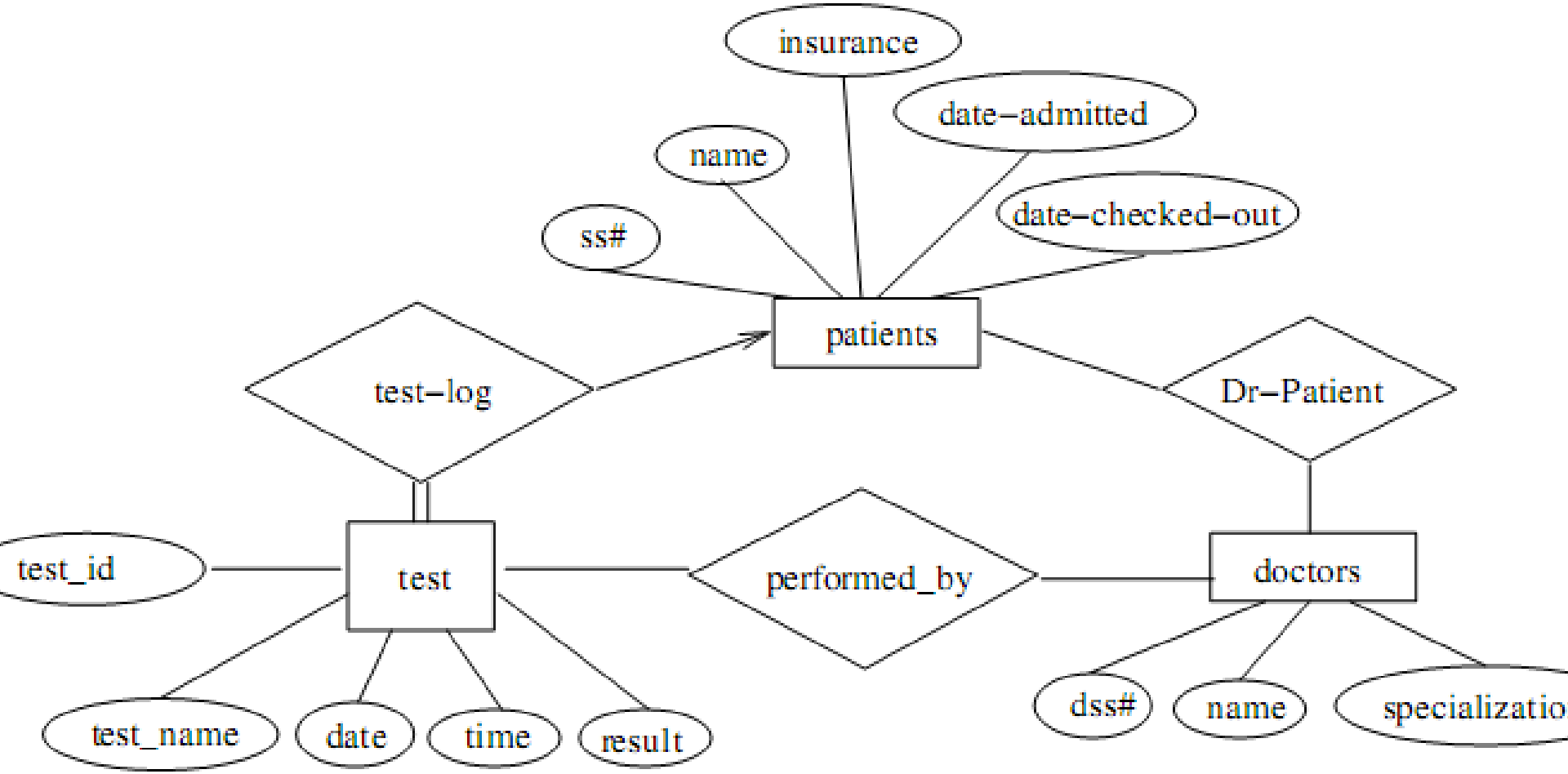
- Construct an E-R diagram for a car-insurance company whose customers own one or more cars each. Each car has associated with it zero to any number of recorded accidents.

- Construct an E-R diagram for a hospital with a set of patients and a set of medical doctors. Associate with each patient a log of the various tests and examinations conducted.

Insurance company



Hospital


insurance
date–admitted
name
date–checked–out
ss#
patients
test–log
Dr–Patient
test_id
test
performed_by
doctors
test_name
date
time
result
dss#
name
specializatio

Relational Model

Relational Model

- The first database systems were based on the network and hierarchical models
- Relational Model was proposed by Edgar. F. Codd (1923-2003) in the early seventies.
- Most of the modern DBMS are relational.
- Simple and elegant model with a mathematical basis.
- Led to the development of a theory of data dependencies and database design.
- Relational algebra operations –
  - crucial role in query optimization and execution.
- Laid the foundation for the development of
  - Tuple relational calculus and then
  - Database standard SQL

Basic structure

bname	account#	cname	balance
Downtown	101	Johnson	500
Lougheed_Mall	215	Smith	700
SFU	102	Hayes	400
SFU	304	Adams	1300

- It has four attributes.
- For each attribute there is a permitted set of values, called the domain of that attribute.
- E.g. the domain of bname is the set of all branch names.
- Let D1 denote the domain of bname, and D2, D3 and $D_4$the remaining attributes' domains respectively.
- Then, any row of deposit consists of a four-tuple v_1;v2;v3;v4 where
- v_1E D1;v2 E D2;v3 E D3;v4 E D4

- In general, deposit contains a subset of the set of all possible rows.
- That is, deposit is a subset of D_1 X D2XD3XD4; or; abbreviated to; $\times_{i=1}^{4} D_i$
- In general, a table of n columns must be a subset of all possible rows $\times_{i=1}^{n} D_i$ (all possible rows)

- Mathematicians define a relation to be a subset of a Cartesian product of a list of domains
- We will use the terms relation and tuple in place of table and row from now on.

- let the tuple variable t refer to a tuple of the relation r.
- We say t E r to denote that the tuple t is in relation r.
- Then t[bname] = t[1] = the value of t on the bname attribute.
- So t[bname] = t[1] = “Downtown",
- and t[cname] = t[3] = “Johnson".

Relational Moel

- Tables are called as *relation*.
- Column headers are known as attributes.
- Rows are called as *tuples*
- Relation schema = name(attributes) + other structure information
  - E.g., keys, other constraints.
- Order of attributes is arbitrary, but in practice we need to assume the order given in the relation schema.
- Relation instance is current set of rows for a relation schema.
- Database schema = collection of relation schemas.

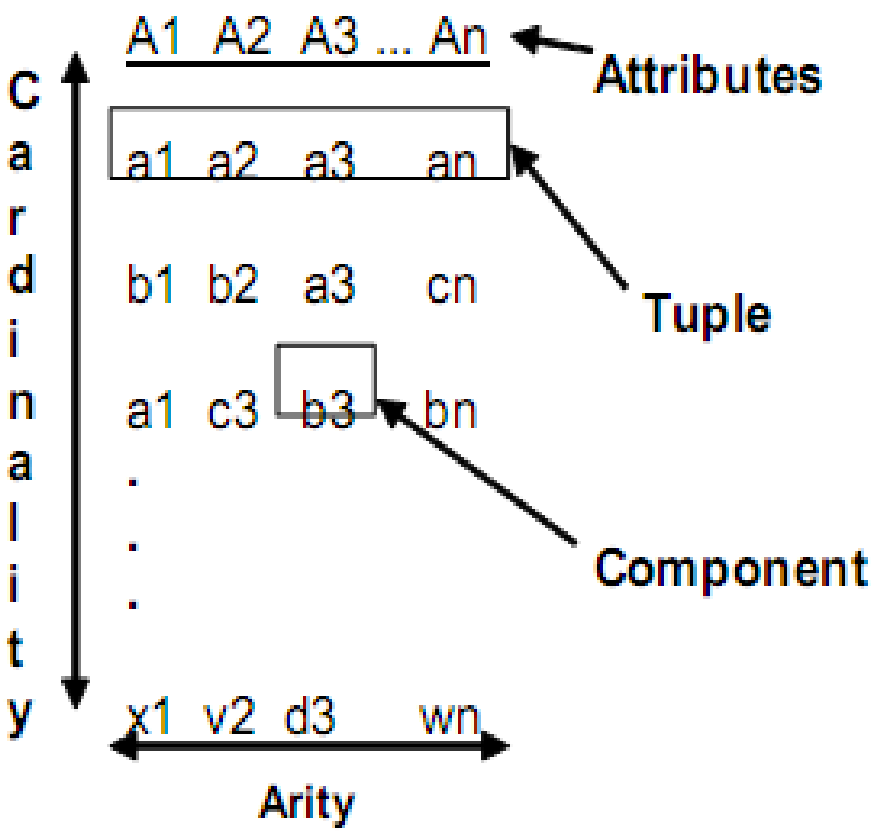
Basic Structure

- Formally, given sets D1, D2,... Dn a relation r is a subset of D1x D2x … x Dn .Thus a relation is a set of n-tuples (a1, a2, …, an) where each ai ∈ Di
- 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

Relation as table

- Rows = tuples
- Columns = components
- Names of columns = attributes
- Set of attribute names = schema
  - REL (A1,A2,...,An)



Set theoretic

- Domain — set of values
  - like a data type
- Cartesian product (or product)
  - D1 □D2 □... □ Dn
  - n-tuples (V1,V2,...,Vn)
  - s.t., V1 □D1, V2 □D2,...,Vn □Dn
- Relation-subset of cartesian product
  - of one or more domains
  - FINITE only; empty set allowed
- Tuples = members of a relation inst.
- Arity = number of domains
- Components = values in a tuple
- Domains — corresp. with attributes
- Cardinality = number of tuples

Query Language

- is a language in which a user requests information from a database.
- These are typically
- Procedural, where the user instructs the system to perform a sequence of operations on the database.
  This will compute the desired information.
- Nonprocedural, where the user specifies the information desired without giving a procedure for obtaining the information.
- A complete query language also contains facilities to insert and delete tuples as well as to modify parts of existing tuples.

RELATIONAL ALGEBRA

- A set of operators (unary and binary) that take relation instances as arguments and return new relations.
- Gives a procedural method of specifying a retrieval query.
- Forms the core component of a relational query engine.
- SQL queries are internally translated into RA expressions.
- Provides a framework for query optimization.

RA Operations

- Six basic operators are fundamental in relational algebra.
- They are

  select (σ),

  project (π),

  cross product (×),

  union (∪),

  intersection (∩),

  difference (−),

  join (⋈)
- The operators take two or more relations as inputs and give a new relation as a result.

The select Operator

- Unary operator.
- Can be used to select those tuples of a relation that

satisfy a given condition (predicate).

- Select is denoted by a lowercase Greek sigma, with the predicate appearing as a subscript.

Notation: σ_{θ} (r)

- σ : select operator (read as sigma)
- θ : selection condition
- r : relation name

Result: a relation with the same schema as r consisting of the tuples in r that satisfy condition θ

Examples of Select Operator

1. Obtain information about a professor with name "giridhar"
   - σ_{name} = "giridhar" (professor)

2. Obtain information about professors who joined the university between 1980 and 1985

$\sigma_{startYear \geq 1980 \wedge startYear < 1985}$ (professor)

3. Suppose there is one more relation, client with the scheme

- Client scheme = (cname,banker)

We might write $\sigma_{cname=banker}(client)$

to find clients who have the same name as their banker.

The Project Operation

- Project copies its argument relation for the specified attributes only. Since a relation is a set, duplicate rows are eliminated. Projection is denoted by the Greek capital letter pi.
- For example, to obtain a relation showing customers and branches, but ignoring amount and loan,

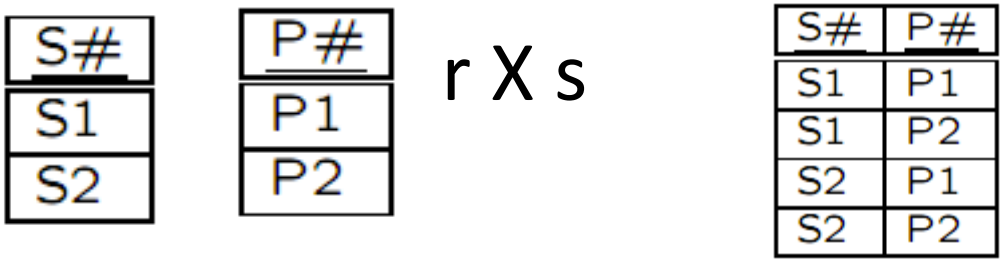
$$\Pi_{bname,cname}(borrow)$$

- To get the names of customers having the same name as their bankers,

$$\Pi_{cname}(\sigma_{cname=banker}(client))$$

- Think of select as taking rows of a relation, and project as taking columns of a relation.

The Cartesian Product Operation

- The Cartesian product of two relations is denoted by a cross X written as
  - r1 X r2 for relations r1 & r2
- The result is a new relation with a tuple for each possible pairing of tuples from r1 and r_2.
- 

- Here r and s share no common attribute names
- What happens when r and s do share common properties ?

The Cartesian Product Operation

- For example, T1(A,B) X T2(B,C)
- By the very definition, We get new relation with T3(A,B,B,C) where the two B's have same name but different values.

- What we will do is thus renaming such attributes. This operator, not a basic one, can take the following syntax: expression[A1, · · · ,An]

where A1, · · · ,An are the new names of the original relational expression, for the corresponding positions.

Example of Cartesian Product

$$(\pi_{\text{Id,Name}}(Student) \times \pi_{Id,DeptId}(Professor))$$
$$[Student.Id, Name, Professor.Id, DeptId])$$

Id	Name
111223344	Smith, Mary
023456789	Simpson, Homer
987654321	Simpson, Bart

A subset of $\pi_{\text{Id,Name}}(\text{STUDENT})$

Id	DeptId
555666777	CS
101202303	CS

A subset of $\pi_{\text{Id,DeptId}}(\text{PROFESSOR})$

STUDENT.Id	Name	PROFESSOR.Id	DeptId
111223344	Smith, Mary	555666777	CS
111223344	Smith, Mary	101202303	CS
023456789	Simpson, Homer	555666777	CS
023456789	Simpson, Homer	101202303	CS
987654321	Simpson, Bart	555666777	CS
987654321	Simpson, Bart	101202303	CS

Their Cartesian product

Rename Operator

- rename operator, denoted by the Greek letter rho (ρ)

$$\rho_x(r)$$

- Suppose we want to find the names of all the customers who live on the same street and in the same city as Smith.

$$\Pi_{street,ccity}(\sigma_{cname=\text{“}Smith\text{”}}(customer))$$

- To find other customers with the same information, we need to reference the customer relation again

$$\sigma_P(customer \times (\Pi_{street,ccity}\ (\sigma_{cname=\text{“}Smith\text{”}}(customer))))$$

- If we use this to rename one of the two customer relations we are using, the ambiguities will disappear.

$$\Pi_{customer.cname}(\sigma_{cust2.street=customer.street \wedge cust2.ccity=customer.ccity}(customer \times (\Pi_{street,ccity}(\sigma_{cname=\text{“}Smith\text{”}}(\rho_{cust2}(customer))))))$$

- We are renaming to cust2.

The Union Operation

- The union operation is denoted U as in set theory. It returns the union set union of two compatible relations.
- For a union operation r U s to be legal, we require that
  - r and s must have the same number of attributes.
  - The domains of the corresponding attributes must be the same.

The Union Operation

- To find all customers of the SFU branch, we must find everyone who has a loan or an account or both at the branch. We need both borrow and deposit relations for this

$$\Pi_{cname}(\sigma_{bname=\text{“}SFU\text{”}}(borrow)) \cup \Pi_{cname}(\sigma_{bname=\text{“}SFU\text{”}}(deposit))$$

- As in all set operations, duplicates are eliminated

The Set Difference Operation

- Set difference is denoted by the minus sign (-)
- It finds tuples that are in one relation, but not in another.
- Thus R – S results in a relation containing tuples that are in R but not is S.
- To find customers of the SFU branch who have an account there but no loan, we write

$$\Pi_{cname}(\sigma_{bname=\text{“}SFU\text{”}}(deposit)) \; - \; \Pi_{cname}(\sigma_{bname=\text{“}SFU\text{”}}(borrow))$$

The Set Intersection Operation

- Set Intersection is denoted by $\cap$
- It returns a relation that contains tuples that are in both of its argument relations.

$$r \cap s = r - (r - s)$$

- To find all customers having both a loan and an account at the SFU branch, we write

$$\Pi_{cname}(\sigma_{bname=\text{“}SFU\text{”}}(borrow)) \cap \Pi_{cname}(\sigma_{bname=\text{“}SFU\text{”}}(deposit))$$

The Natural Join Operation

- Often we want to simplify queries on a cartesian product
- For example, to find all customers having a loan at the bank and the cities in which they live, we need borrow and customer relations

$$\Pi_{borrow.cname,ccity}\ (\sigma_{borrow.cname=customer.cname}\ (borrow \times customer$$

- This type of operation is very common, so we have the natural join, denoted by a $\bowtie$ sign

Natural Join..

- Natural join combines a Cartesian product and a selection into one operation.
- It performs a selection forcing equality on those attributes that appear in both relation schemes.
- Duplicates are removed as in all relation operations.

$$\Pi_{cname,ccity}(borrow \bowtie customer)$$

Basic Query Structure

- The SQL **data-manipulation language (DML)** provides the ability to query information, and insert, delete and update tuples
- A typical SQL query has the form:

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

  - A_i represents an attribute
  - R_i represents a relation
  - P is a predicate.
- The result of an SQL query is a relation.

The select Clause

- The **select** clause list the attributes desired in the result of a query
  - corresponds to the projection operation of the relational algebra
- Example: find the names of all instructors:

  select *name*

  from *instructor*
- NOTE: SQL names are case insensitive (i.e., you may use upper- or lower-case letters.)
  - E.g. *Name* ≡ *NAME* ≡ *name*
  - Some people use upper case wherever we use bold font.

The select Clause (Cont.)

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

select distinct *dept_name*
from *instructor*

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

select all *dept_name*
from *instructor*

The select Clause (Cont.)

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

select *

from *instructor*

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

- The query:

select *ID, name, salary/12*

from *instructor*

would return a relation that is the same as the *instructor* relation, except that the value of the attribute *salary* is divided by 12.

The where Clause

- The **where** clause specifies conditions that the result must satisfy
  - Corresponds to the selection predicate of the relational algebra.
- To find all instructors in Comp. Sci. dept with salary > 80000

  select *name*
  from *instructor*
  where *dept_name = '*Comp. Sci.' **and** *salary* > 80000
- Comparison results can be combined using the logical connectives

  and, or, and **not.**
- Comparisons can be applied to results of arithmetic expressions.

The from Clause

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

select ∗

from *instructor, teaches*

  - generates every possible instructor – teaches pair, with all attributes from both relations
- Cartesian product not very useful directly, but useful combined with where-clause condition (selection operation in relational algebra)

Cartesian Product: *instructor* X *teaches*

instructor

ID	*name*	*dept_name*	*salary*
10101	Srinivasan	Comp. Sci.	65000
12121	Wu	Finance	90000
15151	Mozart	Music	40000
22222	Einstein	Physics	95000
32343	El Said	History	60000

teaches

ID	*course_id*	*sec_id*	*semester*	*year*
10101	CS-101	1	Fall	2009
10101	CS-315	1	Spring	2010
10101	CS-347	1	Fall	2009
12121	FIN-201	1	Spring	2010
15151	MU-199	1	Spring	2010
22222	PHY-101	1	Fall	2009

inst.ID	*name*	*dept_name*	*salary*	*teaches.ID*	*course_id*	*sec_id*	*semester*	*year*
10101	Srinivasan	Comp. Sci.	65000	10101	CS-101	1	Fall	2009
10101	Srinivasan	Comp. Sci.	65000	10101	CS-315	1	Spring	2010
10101	Srinivasan	Comp. Sci.	65000	10101	CS-347	1	Fall	2009
10101	Srinivasan	Comp. Sci.	65000	12121	FIN-201	1	Spring	2010
10101	Srinivasan	Comp. Sci.	65000	15151	MU-199	1	Spring	2010
10101	Srinivasan	Comp. Sci.	65000	22222	PHY-101	1	Fall	2009
...	...	...	...	...	...	...	...	...
...	...	...	...	...	...	...	...	...
12121	Wu	Finance	90000	10101	CS-101	1	Fall	2009
12121	Wu	Finance	90000	10101	CS-315	1	Spring	2010
12121	Wu	Finance	90000	10101	CS-347	1	Fall	2009
12121	Wu	Finance	90000	12121	FIN-201	1	Spring	2010
12121	Wu	Finance	90000	15151	MU-199	1	Spring	2010
12121	Wu	Finance	90000	22222	PHY-101	1	Fall	2009
...	...	...	...	...	...	...	...	...
...	...	...	...	...	...	...	...	...

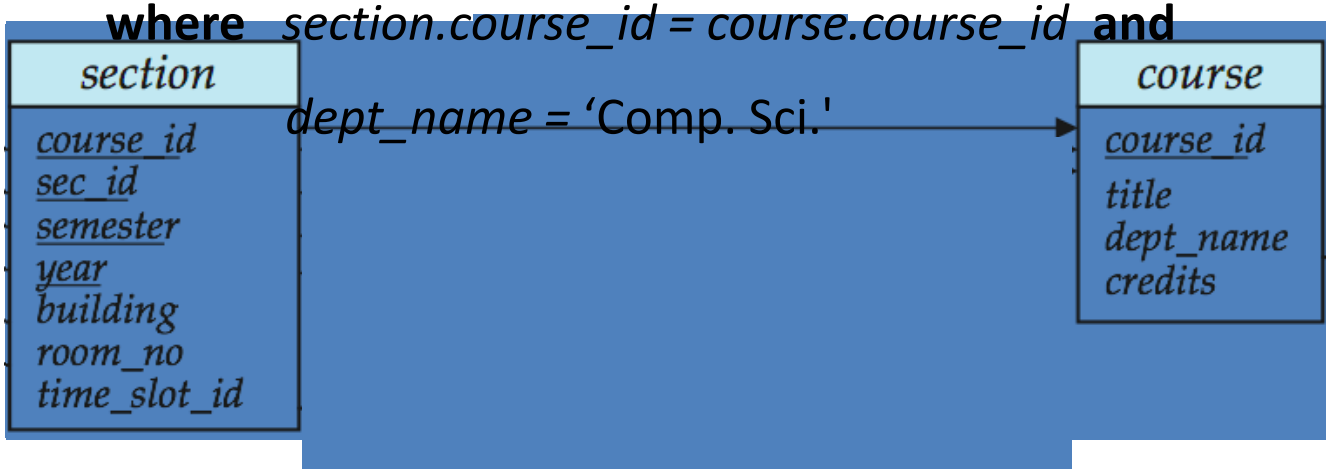
Joins

- For all instructors who have taught some course, find their names and the course ID of the courses they taught.

```
select name, course_id
from instructor, teaches
where instructor.ID = teaches.ID
```

- Find the course ID, semester, year and title of each course offered by the Comp. Sci. department

```
select section.course_id, semester, year, title
from section, course
where section.course_id = course.course_id and
dept_name = 'Comp. Sci.'
```



Natural Join

- Natural join matches tuples with the same values for all common attributes, and retains only one copy of each common column
- **select** *
  from *instructor* **join** *teaches*;

ID	*name*	*dept_name*	*salary*	*course_id*	*sec_id*	*semester*	*year*
10101	Srinivasan	Comp. Sci.	65000	CS-101	1	Fall	2009
10101	Srinivasan	Comp. Sci.	65000	CS-315	1	Spring	2010
10101	Srinivasan	Comp. Sci.	65000	CS-347	1	Fall	2009
12121	Wu	Finance	90000	FIN-201	1	Spring	2010
15151	Mozart	Music	40000	MU-199	1	Spring	2010
22222	Einstein	Physics	95000	PHY-101	1	Fall	2009
32343	El Said	History	60000	HIS-351	1	Spring	2010
45565	Katz	Comp. Sci.	75000	CS-101	1	Spring	2010
45565	Katz	Comp. Sci.	75000	CS-319	1	Spring	2010
76766	Crick	Biology	72000	BIO-101	1	Summer	2009
76766	Crick	Biology	72000	BIO-301	1	Summer	2010

Natural Join Example

- List the names of instructors along with the course ID of the courses that they taught.

  - **select** *name*, *course_id*
    from *instructor, teaches*
    where *instructor.ID = teaches.ID*;

  - **select** *name*, *course_id*
    from *instructor* **natural join** *teaches*;

Natural Join (Cont.)

- Danger in natural join: beware of unrelated attributes with same name which get equated incorrectly
- List the names of instructors along with the the titles of courses that they teach
  - Incorrect version (makes course.dept_name = instructor.dept_name)
    - **select** *name*, *title*
      from *instructor* **join** *teaches* **join** *course*;
  - Correct version
    - **select** *name*, *title*
      from *instructor* **join** *teaches*, *course*
      where *teaches*.*course_id* = *course*.*course_id*;

The Rename Operation

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

  old-name **as** *new-name*

- E.g.
  - **select** *ID, name, salary/12* **as** *monthly_salary*
    from *instructor*

- Find the names of all instructors who have a higher salary than some instructor in 'Comp. Sci'.
  - **select distinct** *T. name*
    from *instructor* **as** *T, instructor* **as** *S*
    where *T.salary > S.salary* **and** *S.dept_name = 'Comp. Sci.'*
- Keyword **as** is optional and may be omitted

  instructor **as** *T* ≡ *instructor T*

String Operations

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

```
select name
from instructor
where name like '%dar%'
```

- Match the string “100 %”

```
like ‘100 \%'  escape  '\'
```

String Operations (Cont.)

- Patters are case sensitive.
- Pattern matching examples:
  - 'Intro%' matches any string beginning with "Intro".
  - '%Comp%' matches any string containing "Comp" as a substring.
  - '_ _ _' matches any string of exactly three characters.
  - '_ _ _ %' matches any string of at least three characters.

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

Ordering the Display of Tuples

- List in alphabetic order the names of all instructors

  select distinct *name*
  from *instructor*
  order by *name*

- We may specify **desc** for descending order or **asc** for ascending order, for each attribute; ascending order is the default.
  - Example: **order by** *name* **desc**
- Can sort on multiple attributes
  - Example: **order by** *dept_name, name*

Where Clause Predicates

- SQL includes a **between** comparison operator
- Example: Find the names of all instructors with salary between $90,000 and $100,000 (that is, ≥ $90,000 and ≤ $100,000)
  - **select** *name*
    from *instructor*
    where *salary* **between** 90000 **and** 100000
- Tuple comparison
  - **select** *name*, *course_id*
    from *instructor*, *teaches*
    where (*instructor*.*ID*, *dept_name*) = (*teaches*.*ID*, 'Biology');

Duplicates

- In relations with duplicates, SQL can define how many copies of tuples appear in the result.
- **Multiset** versions of some of the relational algebra operators – given multiset relations r_1 and r_2:

1. $\sigma_\theta(r_1)$**:** If there are c_1 copies of tuple t_1 in r_1, and t_1 satisfies selections σ_θ, then there are c_1 copies of t_1 in $\sigma_\theta(r_1)$.
2. $\Pi_A(r)$**:** For each copy of tuple t_1 in r_1, there is a copy of tuple $\Pi_A(t_1)$ in $\Pi_A(r_1)$ where $\Pi_A(t_1)$ denotes the projection of the single tuple t_1.
3. $r_1 \times r_2$ **:** If there are c_1 copies of tuple t_1 in r_1 and c_2 copies of tuple t_2 in r_2, there are $c_1 \times c_2$ copies of the tuple $t_1 . t_2$ in $r_1 \times r_2$

Duplicates (Cont.)

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

  r_1 = {(1, *a*) (2,*a*)} r_2 = {(2), (3), (3)}

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

  {(*a*,2), (*a*,2), (*a*,3), (*a*,3), (*a*,3), (*a*,3)}

- SQL duplicate semantics:

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

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

$$\Pi_{A_1,A_2,\ldots,A_n}(\sigma_P(r_1 \times r_2 \times \ldots \times r_m))$$

Set Operations

- Find courses that ran in Fall 2009 or in Spring 2010

(**select** *course_id* **from** *section* **where** *sem* = 'Fall' **and** *year* = 2009)
union
(**select** *course_id* **from** *section* **where** *sem* = 'Spring' **and** *year* = 2010)

- Find courses that ran in Fall 2009 and in Spring 2010

(**select** *course_id* **from** *section* **where** *sem* = 'Fall' **and** *year* = 2009)
intersect
(**select** *course_id* **from** *section* **where** *sem* = 'Spring' **and** *year* = 2010)

- Find courses that ran in Fall 2009 but not in Spring 2010

(**select** *course_id* **from** *section* **where** *sem* = 'Fall' **and** *year* = 2009)
except
(**select** *course_id* **from** *section* **where** *sem* = 'Spring' **and** *year* = 2010)

Set Operations

- Set operations **union, intersect,** and **except**
  - Each of the above operations automatically eliminates duplicates
- To retain all duplicates use the corresponding multiset versions **union all, intersect all** and **except all.**

  Suppose a tuple occurs *m* times in *r* and *n* times in *s,* then, it occurs:
  - *m + n* times in *r* **union all** *s*
  - min(*m,n)* times in *r* **intersect all** *s*
  - max(0, *m – n)* times in *r* **except all** *s*

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*
  - Example: 5 + *null* returns null
- The predicate **is null** can be used to check for null values.
  - Example: Find all instructors whose salary is null.

select *name*
from *instructor*
where *salary* **is null**

Functional Dependency

Overview

- Domain; Ref. Integrity constraints
- Assertions and Triggers
- Security
- Functional dependencies
  - why
  - definition
  - Armstrong's "axioms"
  - closure and cover

Functional dependencies

takes1 (ssn, c-id, grade, name, address)

Ssn	c-id	Grade	Name	Address
123	413	A	smith	Main
123	415	B	smith	Main
123	211	A	smith	Main

Functional Dependencies

- Redundancy
  - space
  - inconsistencies
  - insertion/deletion anomalies
- What caused the problem?

Functional dependencies

Definition: $a \rightarrow b$

'a' functionally determines 'b'

Ssn	c-id	Grade	Name	Address
123	413	A	smith	Main
123	415	B	smith	Main
123	211	A	smith	Main

Functional dependencies

formally:

$$X \rightarrow Y \qquad \Rightarrow \quad (t1[x] = t2[x] \Rightarrow t1[y] = t2[y])$$

if two tuples agree on the 'X' attribute,
they *must* agree on the 'Y' attribute, too
(e.g., if ssn is the same, so should address)

… a functional dependency is a generalization of the notion of a *key*

Functional dependencies

K is a **superkey** for relation R iff K -> R

K is a **candidate key** for relation R iff:

K -> R

for no a $\subset$ K, a -> R

Functional dependencies

Closure of a set of FD: all implied FDs – e.g.:

ssn -> name, address

ssn, c-id -> grade

imply

ssn, c-id -> grade, name, address

ssn, c-id -> ssn

FDs - Armstrong's axioms

Closure of a set of FD: all implied FDs – e.g.:

ssn -> name, address

ssn, c-id -> grade

how to find all the implied ones, systematically?

FDs - Armstrong's axioms

"Armstrong's axioms" guarantee **soundness** and **completeness**:

- Reflexivity: $Y \subseteq X \Rightarrow X \rightarrow Y$

  e.g., ssn, name -> ssn

- Augmentation $X \rightarrow Y \Rightarrow XW \rightarrow YW$

  e.g., ssn->name then ssn,grade-> ssn,grade

FDs - Armstrong's axioms

- Transitivity $\left.\begin{array}{l} X \rightarrow Y \\ Y \rightarrow Z \end{array}\right\} \Rightarrow X \rightarrow Z$

  ssn->address

  address-> county-tax-rate

  THEN:

  ssn-> county-tax-rate

FDs - Armstrong's axioms

Reflexivity: $Y \subseteq X \Rightarrow X \rightarrow Y$

Augmentation: $X \rightarrow Y \Rightarrow XW \rightarrow YW$

Transitivity: $\left.\begin{array}{l} X \rightarrow Y \\ Y \rightarrow Z \end{array}\right\} \Rightarrow X \rightarrow Z$

'sound' and 'complete'

Trivial FDs

The FD X→Y is ***trivial*** if set {Y} is a subset of set {X}

Examples: If A and B are attributes of R,
{A}→{A}
{A,B} →{A}
{A,B} →{B}
{A,B} →{A,B}

are all trivial FDs.

Full functional Dependency

- A full functional dependency occurs when you already meet the requirements for a functional dependency and the set of attributes on the left side of the functional dependency statement cannot be reduced any farther.

- For example, “{SSN, age} -> name” is a functional dependency, but it is not a full functional dependency because you can remove age from the left side of the statement without impacting the dependency relationship

Closure of a set of attributes

- Closure of a set of attributes X with respect to is the set X^+ of all attributes that are functionally determined by X using $^+$
- Imagine the following list of FD's. We are going to calculate a closure for A from this relationship.

  $A \rightarrow B$ $B \rightarrow C$ $AB \rightarrow D$
- The closure would be as follows:

  a) A → A (by Armstrong's reflexivity)
  b) A → AB (by 1. and (a))
  c) A → ABD (by (b), 3, and Armstrong's transitivity)
  d) A → ABCD (by (c), and 2)
- The closure is therefore A → ABCD. By calculating the closure of A, we have validated that A is also a good candidate key as its closure is every single data value in the relationship.

FDs – finding the closure 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

- We can further simplify manual computation of F^+ by using the following additional rules →

FDs - Armstrong's axioms

Additional rules:

- Union $\left.\begin{array}{l} X \to Y \\ X \to Z \end{array}\right\} \Rightarrow X \to YZ$
- Decomposition $X \to YZ \Rightarrow \left.\begin{array}{l} X \to Y \\ X \to Z \end{array}\right\}$
- Pseudo-transitivity $\left.\begin{array}{l} X \to Y \\ YW \to Z \end{array}\right\} \Rightarrow XW \to Z$

FDs - Closure A+

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

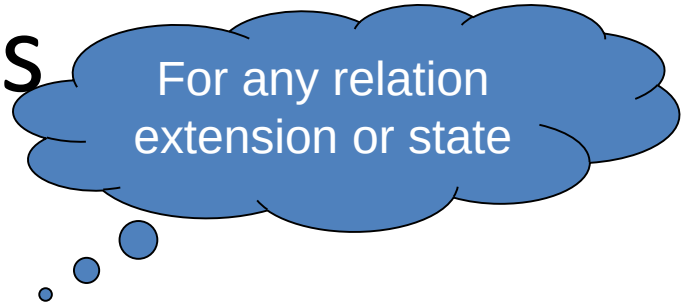
  result := α;
  while (changes to *result*) **do**
  for each $\beta \rightarrow \gamma$ **in** *F* **do**
  begin
  if $\beta \subseteq$ *result* **then** *result* := *result* $\cup \gamma$
  end

FDs - Closure A+ (example)

- $R = (A, B, C, G, H, I)$
- $F = \{A \rightarrow B,\ A \rightarrow C,\ CG \rightarrow H,\ CG \rightarrow I,\ B \rightarrow H\}$
- $(AG)^+$
  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 $AG \rightarrow R$?**
  2. **Is any subset of AG a superkey?**
     1. **Does $A^+ \rightarrow R$?**
     2. **Does $G^+ \rightarrow R$?**

Normalization

Definitions



- **Superkey:** a set of attributes uniquely (but not necessarily minimally!) identifying a tuple of a relation.
- **Key:** A ***set of attributes*** that uniquely and minimally identifies a tuple of a relation.
- **Candidate key:** If there is more than one ***key*** in a relation, the keys are called candidate keys.
- **Primary key:** One ***candidate key*** is chosen to be the primary key.
- **Prime attribute:** An attribute ***A*** that is part of a ***candidate*** key ***X*** (vs. nonprime attribute)

Normal Forms

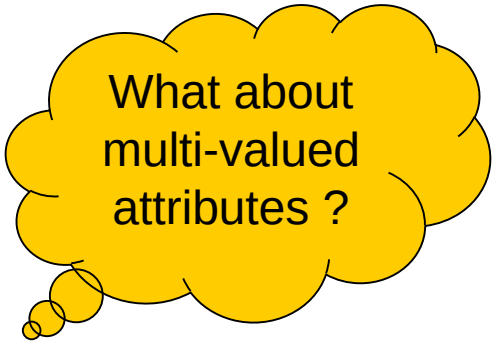
- 1NF, 2NF, 3NF, BCNF (4NF, 5NF)
- ***Minimize redundancy***
- ***Minimize update anomalies***
- Normal form ↑ = redundancy and update anomalies ↓ and relations become smaller.
- Join operation to recover original relations.

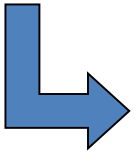
1NF

- 1NF: The relation should have no non-atomic values.

R_{non1NF}

ID	**Name**	**LivesIn**
100	Pettersson	{Stockholm, Linköping}
101	Andersson	{Linköping}
102	Svensson	{Ystad, Hjo, Berlin}





Normalization

$R1_{1NF}$

ID	**Name**
100	Pettersson
101	Andersson
102	Svensson

$R2_{1NF}$

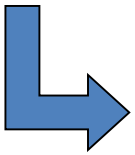
ID	***LivesIn***
100	*Stockholm*
100	*Linköping*
101	*Linköping*
102	*Ystad*
102	*Hjo*
102	*Berlin*

2NF

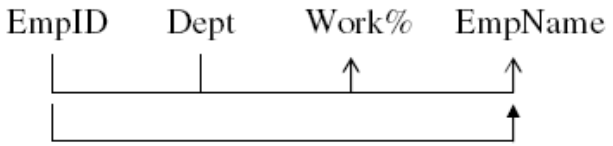
- 2NF: no **nonprime** attribute should be functionally dependent on a **part** of a candidate key.

R_{non2NF}

EmpID	***Dept***	**Work%**	**EmpName**
100	*Dev*	50	Baker
100	*Support*	50	Baker
200	*Dev*	80	Miller



Normalization



$R1_{2NF}$

EmpID	**EmpName**
100	Baker
200	Miller

$R2_{2NF}$

EmpID	***Dept***	**Work%**
100	*Dev*	50
100	*Support*	50
200	*Dev*	80

2NF

- No 2NF: A part of a candidate key can have repeated values in the relation and, thus, so can have the nonprime attribute, i.e. redundancy + insertion and modification anomalies.

- An FD X→Y is a **full functional dependency (FFD)** if removal of any attribute A_i from X means that the dependency does not hold any more.

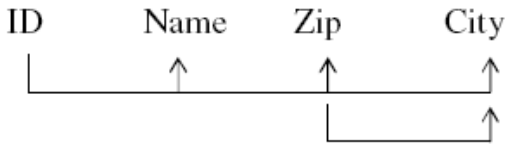
- 2NF: Every **nonprime** attribute is fully functionally dependent on every candidate key.

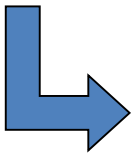
3NF

- 3NF: **2NF +** no **nonprime** attribute should be functionally dependent on a set of attributes that is not a candidate key

R_{non3NF}

ID	**Name**	**Zip**	**City**
100	Andersson	58214	Linköping
101	Björk	10223	Stockholm
102	Carlsson	58214	Linköping





Normalization

$R1_{3NF}$

ID	**Name**	**Zip**
100	Andersson	58214
101	Björk	10223
102	Carlsson	58214

$R2_{3NF}$

Zip	**City**
58214	Linköping
10223	Stockholm

3NF

- No 3NF (but 2NF): A set of attributes that is not a candidate key can have repeated values in the relation and, thus, so can have the nonprime attribute, i.e. redundancy + insertion and modification anomalies.

- An FD X→Y is a **transitive dependency** if there is a set of attributes Z that is not a candidate key and such that both X→Z and Z→Y hold.

- 3NF: 2NF + no **nonprime** attribute is transitively dependent on any candidate key.

Boyce-Codd Normal Form

- BCNF: **Every determinant is a candidate key**

- BCNF = decompose if X → A is such that X is not a candidate key and A is a prime attribute.

- Example: Given R(A,B,C,D) and AB→CD, C→B. Then R is in 3NF but not in BCNF
  - C is a determinant but not a candidate key.
  - Decompose into R1(A,C,D) with AC → D and R2(C,B) with C → B.

Properties of decomposition

- Keep all attributes from the universal relation R.
- Preserve the identified functional dependencies.
- Lossless join
  - It must be possible to join the smaller tables to arrive at composite information without spurious tuples.

Normalization: Example

Given universal relation

**R(PID, PersonName,
Country, Continent, ContinentArea,
NumberVisitsCountry)**

- Functional dependencies?
- Keys?

Normalization: Example

PID → PersonName

PID, Country → NumberVisitsCountry

Country → Continent

Continent → ContinentArea

- Based on FDs, what are keys for R?
- Use inference rules

Normalization: Example

Country → Continent, Continent → ContinentArea,
then
Country → Continent, ContinentArea (transitive + aditive rules)

PID, Country → Continent, ContinentArea (augmentation + decomposition rules),
PID, Country → PersonName (augmentation + decomposition rules),
PID, Country → NumberVisitsCountry,
then
PID, Country → Continent, ContinentArea, PersonName, NumberVisitsCountry (additive rule)

<u>PID, Country</u> is the key for R.

Normalization: Example

Is

R (PID,Country,Continent,ContinentArea,PersonName,NumberVisitsCountry)

in 2NF?

No, *PersonName* depends on a part of the candidate key (*PID*), then

R1(PID, PersonName)

R2(PID, Country, Continent, ContinentArea, NumberVisitsCountry)

Is R2 in 2NF?

No, *Continent* and *ContinentArea* depend on a part of the candidate key (*Country*), then

R1(PID, PersonName)

R21(Country, Continent, ContinentArea)

R22(PID, Country, NumberVisitsCountry)

→ R1, R21, R22 are in 2NF

2NF: *no nonprime attribute* should be functionally dependent on a ***part*** of a candidate key.

Are R1, R21, R22

3NF: 2NF + no nonprime attribute should be functionally dependent on a set of attributes that is not a candidate key

R22(<u>PID, Country</u>, NumberVisitsCountry),
R1(<u>PID</u>, PersonName):

Yes, a single nonprime attribute, no transitive dependencies.

R21(<u>Country</u>, Continent, ContinentArea):

No, Continent defines ContinentArea, then
R211(<u>Country</u>, Continent)
R212(<u>Continent</u>, ContinentArea)

→ R1, R22, R211, R212 are in 3NF

Are R1, R22, R211, R212 in BCNF?

BCNF: Every determinant is a candidate key

R22(PID, Country, NumberVisitsCoun...)

R1(PID, PersonName):

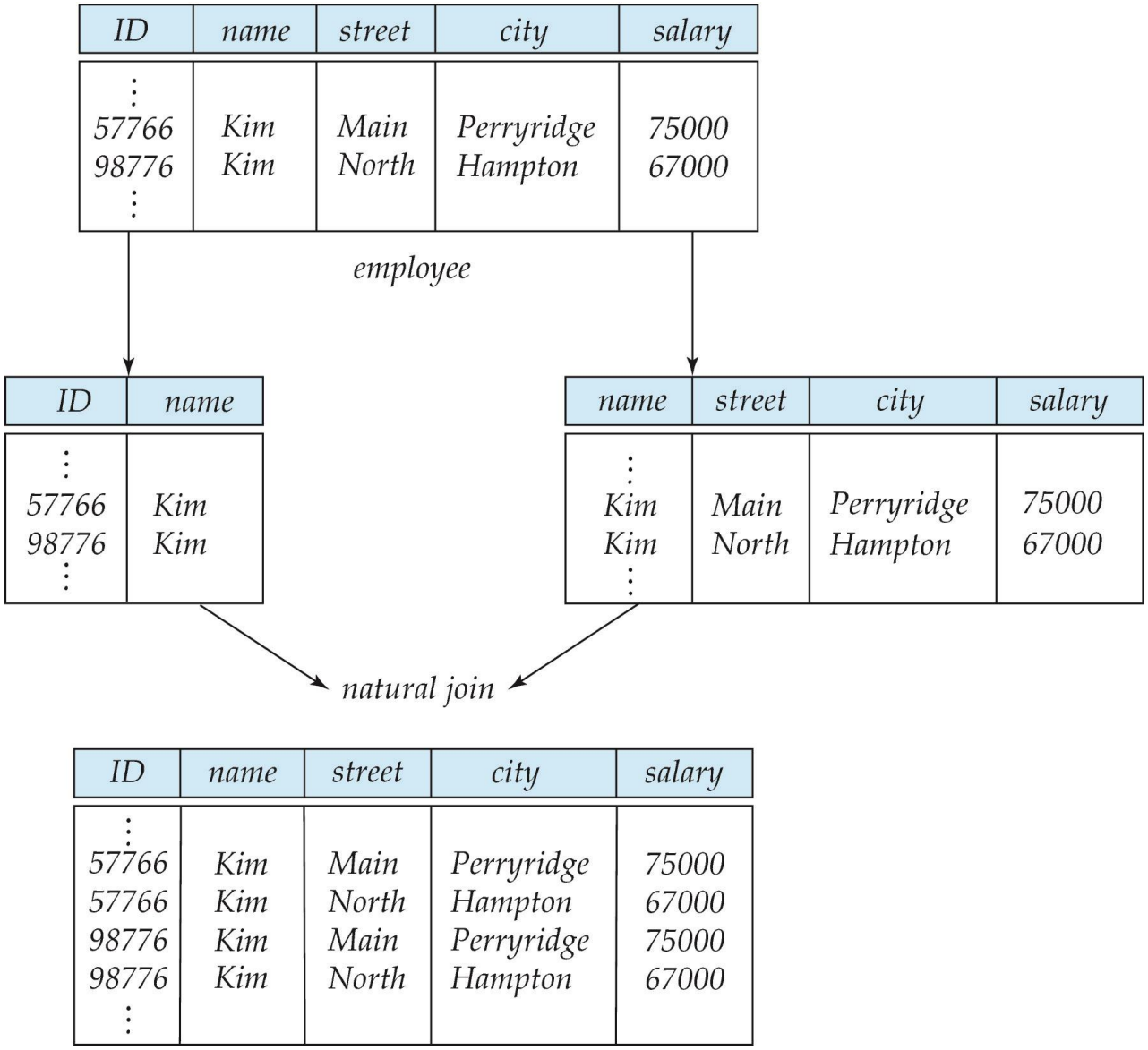
R211(Country, Continent)

R212(Continent, ContinentArea)

→ Yes

Can the universal relation R be reproduced from R1, R22, R211 and R212 without spurious tuples?

A Lossy Decomposition



- **Lossless join decomposition**
- Decomposition of *R = (A, B, C)*

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

A	*B*	*C*
α	1	A
β	2	B

r

A	*B*
α	*1*
β	*2*

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

B	*C*
1	A
2	B

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

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

A	*B*	*C*
α	1	A
β	2	B

Sufficient Condition for Lossless Join

- Lossless Join means:
  - Let { R_1 , R_2 } be a decomposition of R (meaning that $R_1 \cup R_2 = R$);
- Prove that for all legal instances r:

  $r \subseteq \Pi_{R1}(r) \rhd\lhd \Pi_{R2}(r)$
- Prove that this decomposition is lossless if

  $R_1 \cap R_2 \rightarrow R_1$ or $R_1 \cap R_2 \rightarrow R_2$
  - Give an example of a lossless join decomposition (instance) when neither

    $R_1 \cap R_2 \rightarrow R_1$ nor $R_1 \cap R_2 \rightarrow R_2$ hold?

Canonical Cover

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

Dependency Preservation

- Let F_i be the set of dependencies F^+ that include only attributes in R_i.
  - A decomposition is **dependency preserving**, if

    $(F_1 \cup F_2 \cup \ldots \cup F_n)^+ = F^+$
  - If it is not, then checking updates for violation of functional dependencies may require computing joins, which is expensive.
- In simple term, any functional dependency can be tested in one relation.(join not required)

Comparison of BCNF and 3NF

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

Summary and open issues

- Good design: informal and formal properties of relations
- Functional dependencies, and thus normal forms, are about attribute *semantics* (= real-world knowledge), normalization can only be automated if FDs are given.

- Are high normal forms good design when it comes to performance?
  - No, denormalization may be required.

Chapter 15: Transactions

- Transaction Concept
- Transaction State
- Implementation of Atomicity and Durability
- Concurrent Executions
- Serializability
- Recoverability
- Implementation of Isolation
- Transaction Definition in SQL
- Testing for Serializability.

Transaction Concept

- A ***transaction*** is a *unit* of program execution that accesses and possibly updates various data items.
- A transaction must see a consistent database.
- During transaction execution the database may be inconsistent.
- When the transaction is committed, the database must be consistent.
- Two main issues to deal with:
  - Failures of various kinds, such as hardware failures and system crashes
  - Concurrent execution of multiple transactions

ACID Properties

To preserve integrity of data, the database system must ensure:

- **Atomicity.** Either all operations of the transaction are properly reflected in the database or none are.
- **Consistency.** Execution of a transaction in isolation preserves the consistency of the database.
- **Isolation.** Although multiple transactions may execute concurrently, each transaction must be unaware of other concurrently executing transactions. Intermediate transaction results must be hidden from other concurrently executed transactions.
  - That is, for every pair of transactions T_i and T_j, it appears to T_i that either T_j, finished execution before T_i started, or T_j started execution after T_i finished.
- **Durability.** After a transaction completes successfully, the changes it has made to the database persist, even if there are system failures.

Example of Fund Transfer

- Transaction to transfer $50 from account *A* to account *B*:
  1. **read**(*A*)
  2. *A* := *A* – 50
  3. **write**(*A*)
  4. **read**(*B*)
  5. *B* := *B* + 50
  6. **write**(*B)*
- Consistency requirement – the sum of *A* and *B* is unchanged by the execution of the transaction.
- Atomicity requirement — if the transaction fails after step 3 and before step 6, the system should ensure that its updates are not reflected in the database, else an inconsistency will result.

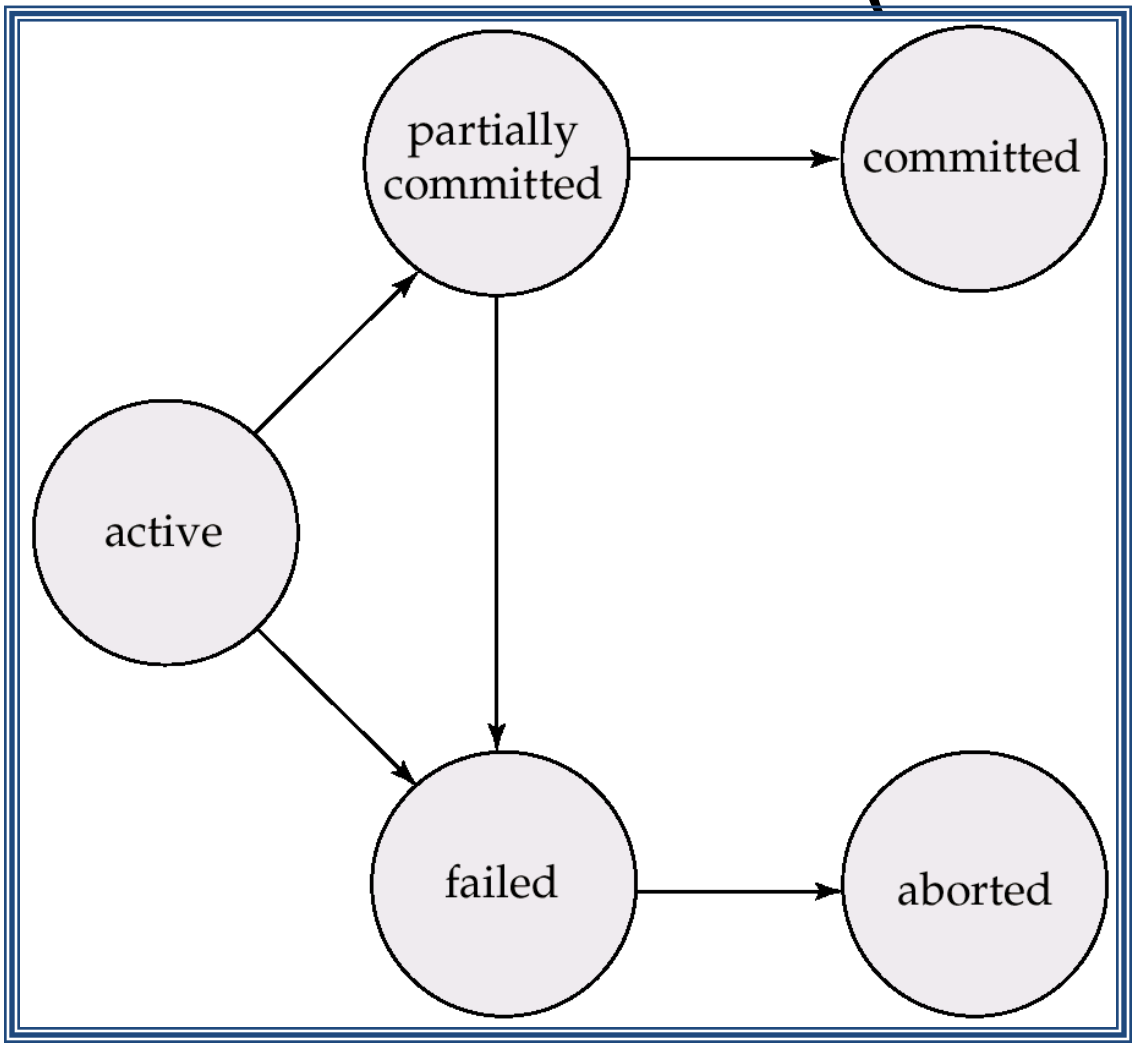
Example of Fund Transfer (Cont.)

- Durability requirement — once the user has been notified that the transaction has completed (i.e., the transfer of the $50 has taken place), the updates to the database by the transaction must persist despite failures.
- Isolation requirement — if between steps 3 and 6, another transaction is allowed to access the partially updated database, it will see an inconsistent database (the sum $A + B$ will be less than it should be).
  Can be ensured trivially by running transactions ***serially,*** that is one after the other. However, executing multiple transactions concurrently has significant benefits, as we will see.

Transaction State

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

Transaction State (Cont.)

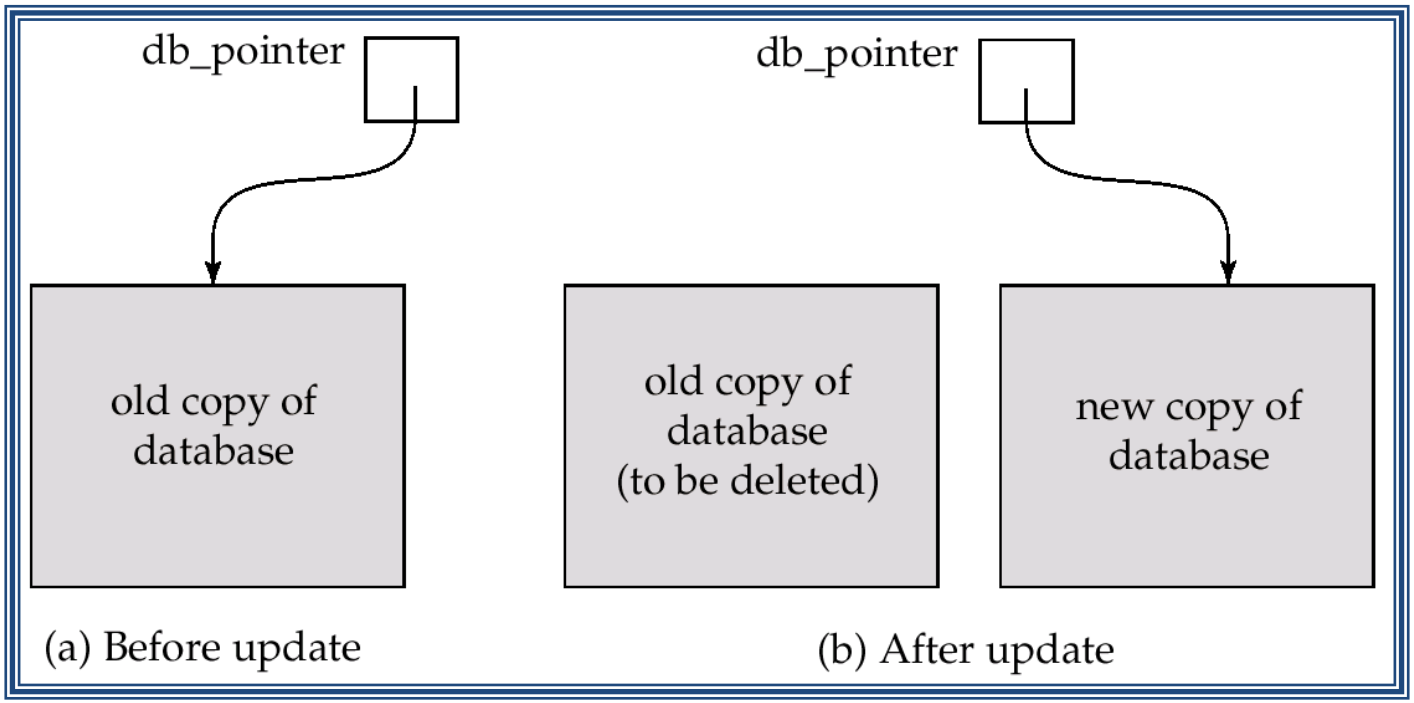


Implementation of Atomicity and Durability

- The recovery-management component of a database system implements the support for atomicity and durability.
- The *shadow-database* scheme:
  - assume that only one transaction is active at a time.
  - a pointer called db_pointer always points to the current consistent copy of the database.
  - all updates are made on a *shadow copy* of the database, and **db_pointer** is made to point to the updated shadow copy only after the transaction reaches partial commit and all updated pages have been flushed to disk.
  - in case transaction fails, old consistent copy pointed to by **db_pointer** can be used, and the shadow copy can be deleted.

Implementation of Atomicity and Durability (Cont.)

The shadow-database scheme:



- Assumes disks to not fail
- extremely inefficient for large databases: executing a single transaction requires copying the *entire* database..

Concurrent Executions

- Multiple transactions are allowed to run concurrently in the system. Advantages are:
  - **increased processor and disk utilization**, leading to better transaction *throughput:* one transaction can be using the CPU while another is reading from or writing to the disk
  - **reduced average response time** for transactions: short transactions need not wait behind long ones.
- *Concurrency control schemes* – mechanisms to achieve isolation, i.e., to control the interaction among the concurrent transactions in order to prevent them from destroying the consistency of the databas

Schedules

- *Schedules* – sequences that indicate the chronological order in which instructions of concurrent transactions are executed
  - a schedule for a set of transactions must consist of all instructions of those transactions
  - must preserve the order in which the instructions appear in each individual transaction.

Example Schedules

- Let T_1 transfer 50 from A to B, and T_2 transfer 10% of the balance from A to B. The following is a serial schedule (Schedule 1 in the text), in which T_1 is followed by T_2.

T_1	T_2
read(A)	
$A := A - 50$	
write (A)	
read(B)	
$B := B + 50$	
write(B)	
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
	read(B)
	$B := B + temp$
	write(B)

Example Schedule (Cont.)

- Let T_1 and T_2 be the transactions defined previously. The following schedule (Schedule 3 in the text) is not a serial schedule, but it is *equivalent* to Schedule 1.

T_1	T_2
read(A)	
$A := A - 50$	
write(A)	
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
read(B)	
$B := B + 50$	
write(B)	
	read(B)
	$B := B + temp$
	write(B)

In both Schedule 1 and 3, the sum A + B is preserved.

Example Schedules (Cont.)

- The following concurrent schedule (Schedule 4 in the text) does not preserve the value of the the sum $A + B$.

T_1	T_2
read(A)	
$A := A - 50$	
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
	read(B)
write(A)	
read(B)	
$B := B + 50$	
write(B)	
	$B := B + temp$
	write(B)

Serializability

- Basic Assumption – Each transaction preserves database consistency.
- Thus serial execution of a set of transactions preserves database consistency.
- A (possibly concurrent) schedule is serializable if it is equivalent to a serial schedule. Different forms of schedule equivalence give rise to the notions of:
  1. conflict serializability
  2. view serializability
- We ignore operations other than **read** and **write** instructions, and we assume that transactions may perform arbitrary computations on data in local buffers in between reads and writes. Our simplified schedules consist of only **read** and **write** instructions.

Conflict Serializability

- Instructions I_i and I_j of transactions T_i and T_j respectively, **conflict** if and only if there exists some item Q accessed by both I_i and I_j, and at least one of these instructions wrote Q.
  1. I_i = **read**(Q), I_j = **read**(Q). I_i and I_j don't conflict.
  2. I_i = **read**(Q), I_j = **write**(Q). They conflict.
  3. I_i = **write**(Q), I_j = **read**(Q). They conflict
  4. I_i = **write**(Q), I_j = **write**(Q). They conflict
- Intuitively, a conflict between I_i and I_j forces a (logical) temporal order between them. If I_i and I_j are consecutive in a schedule and they do not conflict, their results would remain the same even if they had been interchanged in the schedule.

Conflict Serializability (Cont.)

- If a schedule S can be transformed into a schedule S' by a series of swaps of non-conflicting instructions, we say that S and S' are **conflict equivalent**.
- We say that a schedule S is **conflict serializable** if it is conflict equivalent to a serial schedule
- Example of a schedule that is not conflict serializable:

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

We are unable to swap instructions in the above schedule to obtain either the serial schedule < T_3, T_4 >, or the serial schedule < T_4, T_3 >.

Conflict Serializability (Cont.)

- Schedule 3 below can be transformed into Schedule 1, a serial schedule where T_2 follows T_1, by series of swaps of non-conflicting [illegible] therefore Schedule [illegible] zable.

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

View Serializability

- Let S and S' be two schedules with the same set of transactions. S and S' are **view equivalent** if the following three conditions are met:
  1. For each data item Q, if transaction T_i reads the initial value of Q in schedule S, then transaction T_i must, in schedule S', also read the initial value of Q.
  2. For each data item Q if transaction T_i executes **read**(Q) in schedule S, and that value was produced by transaction T_j (if any), then transaction T_i must in schedule S' also read the value of Q that was produced by transaction T_j .
  3. For each data item Q, the transaction (if any) that performs the final **write**(Q) operation in schedule S must perform the final **write**(Q) operation in schedule S'.

As can be seen, view equivalence is also based purely on **reads**

and **writes** alone.

View Serializability (Cont.)

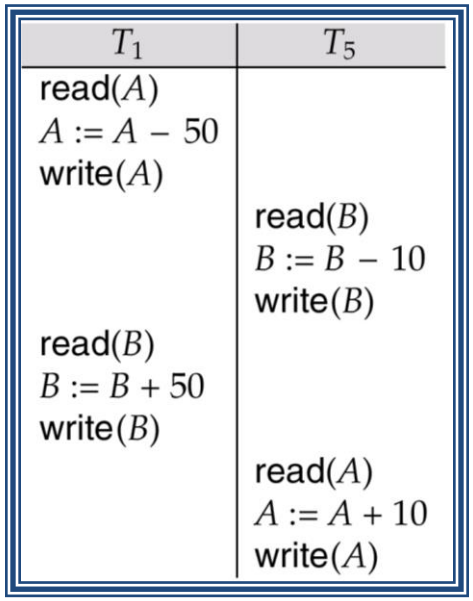
- A schedule *S* is **view serializable** it is view equivalent to a serial schedule.
- Every conflict serializable schedule is also view serializable.
- Schedule 9 (from text) — a schedule which is view-serializable but *not* conflict serializable.

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

- Every view serializable schedule that is not conflict serializable has **blind writes.**

Other Notions of Serializability

- Schedule 8 (from text) given below produces same outcome as the serial schedule < T_1, T_5 >, yet is not conflict equivalent or view equivalent to it.



T_1	T_5
read(A)	
$A := A - 50$	
write(A)	
	read(B)
	$B := B - 10$
	write(B)
read(B)	
$B := B + 50$	
write(B)	
	read(A)
	$A := A + 10$
	write(A)

- Determining such equivalence requires analysis of operations other than read and write.

Recoverability

Need to address the effect of transaction failures on concurrently running transactions.

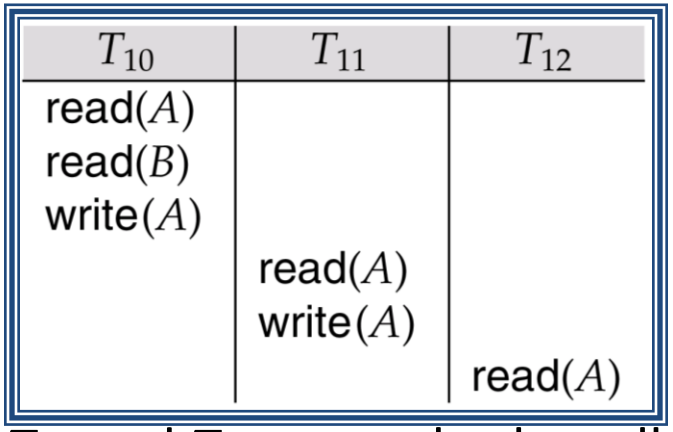
- **Recoverable schedule** — if a transaction T_j reads a data items previously written by a transaction T_i, the commit operation of T_i appears before the commit operation of T_j.
- The following schedule (Schedule 11) is not recoverable if T_9 commits immediately after the read

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

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

Recoverability (Cont.)

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



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

  If T_{10} fails, T_{11} and T_{12} must also be rolled back.
- Can lead to the undoing of a significant amount of work

Recoverability (Cont.)

- **Cascadeless schedules** — cascading rollbacks cannot occur; for each pair of transactions T_i and T_j such that T_j reads a data item previously written by T_i, the commit operation of T_i appears before the read operation of T_j.
- Every cascadeless schedule is also recoverable
- It is desirable to restrict the schedules to those that are cascadeless

Implementation of Isolation

- Schedules must be conflict or view serializable, and recoverable, for the sake of database consistency, and preferably cascadeless.
- A policy in which only one transaction can execute at a time generates serial schedules, but provides a poor degree of concurrency..
- Concurrency-control schemes tradeoff between the amount of concurrency they allow and the amount of overhead that they incur.
- Some schemes allow only conflict-serializable schedules to be generated, while others allow view-serializable schedules that are not conflict-serializable.

Transaction Definition in SQL

- Data manipulation language must include a construct for specifying the set of actions that comprise a transaction.
- In SQL, a transaction begins implicitly.
- A transaction in SQL ends by:
  - **Commit work** commits current transaction and begins a new one.
  - **Rollback work** causes current transaction to abort.
- Levels of consistency specified by SQL-92:
  - **Serializable** — default
  - **Repeatable read**
  - **Read committed**
  - **Read uncommitted**

Levels of Consistency in SQL-92

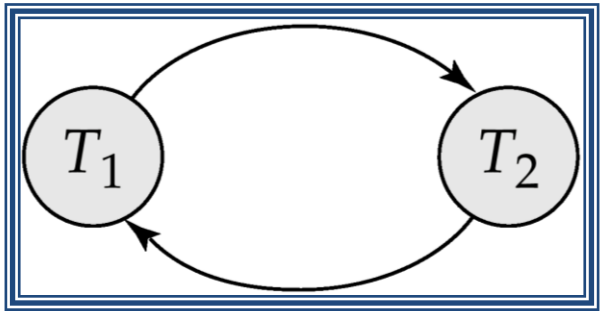
- **Serializable** — default
- **Repeatable read** — only committed records to be read, repeated reads of same record must return same value. However, a transaction may not be serializable – it may find some records inserted by a transaction but not find others.
- **Read committed** — only committed records can be read, but successive reads of record may return different (but committed) values.
- **Read uncommitted** — even uncommitted records may be read.

Lower degrees of consistency useful for gathering approximate information about the database, e.g., statistics for query optimizer.

Testing for Serializability

- Consider some schedule of a set of transactions T_1, T_2, …, T_n
- **Precedence graph** — a direct graph where the vertices are the transactions (names).
- We draw an arc from T_i to T_j if the two transaction conflict, and T_i accessed the data item on which the conflict arose earlier.
- We may label the arc by the item that was accessed.
- **Example 1**

x

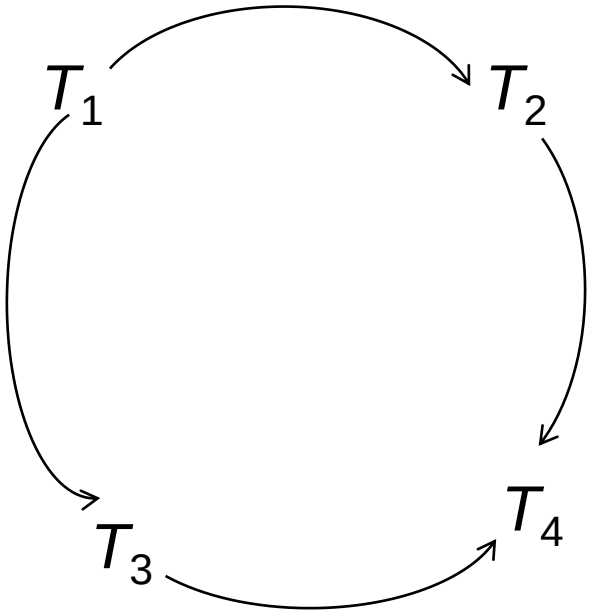


y

Example Schedule (Schedule A)

T_1	T_2	T_3	T_4	T_5
	read(X)			
read(Y)				
read(Z)				
				read(V)
				read(W)
				read(W)
	read(Y)			
	write(Y)			
		write(Z)		
read(U)				
			read(Y)	
			write(Y)	
			read(Z)	
			write(Z)	
read(U)				
write(U)				

Precedence Graph for Schedule A



Test for Conflict Serializability

- A schedule is conflict serializable if and only if its precedence graph is acyclic.
- Cycle-detection algorithms exist which take order n^2 time, where n is the number of vertices in the graph. (Better algorithms take order $n + e$ where e is the number of edges.)
- If precedence graph is acyclic, the serializability order can be obtained by a *topological sorting* of the graph. This is a linear order consistent with the partial order of the graph. For example, a serializability order for Schedule A would be $T_5 \rightarrow T_1 \rightarrow T_3 \rightarrow T_2 \rightarrow T_4$.

Test for View Serializability

- The precedence graph test for conflict serializability must be modified to apply to a test for view serializability.
- The problem of checking if a schedule is view serializable falls in the class of *NP*-complete problems. Thus existence of an efficient algorithm is unlikely.
  However practical algorithms that just check some *sufficient conditions* for view serializability can still be used.

Concurrency Control vs. Serializability Tests

- Testing a schedule for serializability *after* it has executed is a little too late!
- Goal – to develop concurrency control protocols that will assure serializability. They will generally not examine the precedence graph as it is being created; instead a protocol will impose a discipline that avoids nonseralizable schedules. Will study such protocols in Chapter 16.
- Tests for serializability help understand why a concurrency control protocol is correct.

End of Chapter

Schedule 2 -- A Serial Schedule in Which T_2 is Followed by T_1

T_1	T_2
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
	read(B)
	$B := B + temp$
	write(B)
read(A)	
$A := A - 50$	
write(A)	
read(B)	
$B := B + 50$	
write(B)	

Schedule 5 -- Schedule 3 After Swapping A Pair of Instructions

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

Schedule 6 -- A Serial Schedule That is Equivalent to Schedule 3

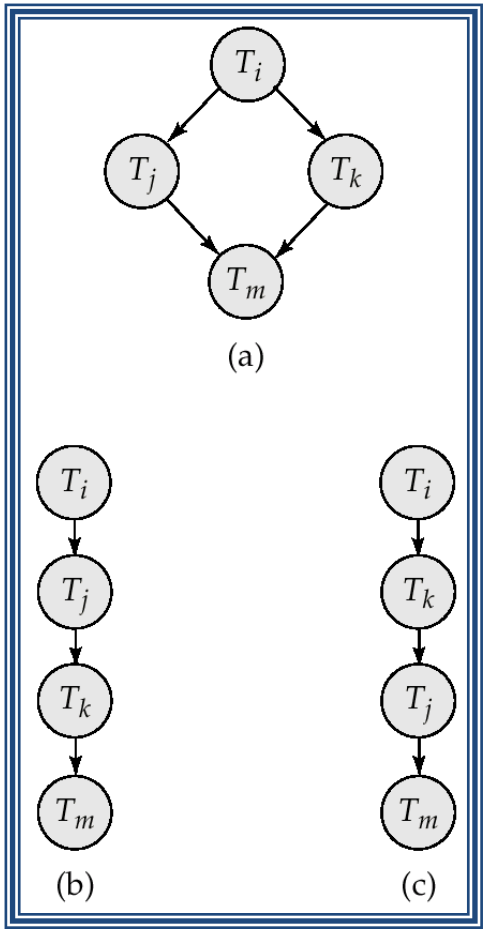
T_1	T_2
read(A)	
write(A)	
read(B)	
write(B)	
	read(A)
	write(A)
	read(B)
	write(B)

Schedule 7

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

Precedence Graph for
(a) Schedule 1 and (b) Schedule 2

Illustration of Topological Sorting



Precedence Graph

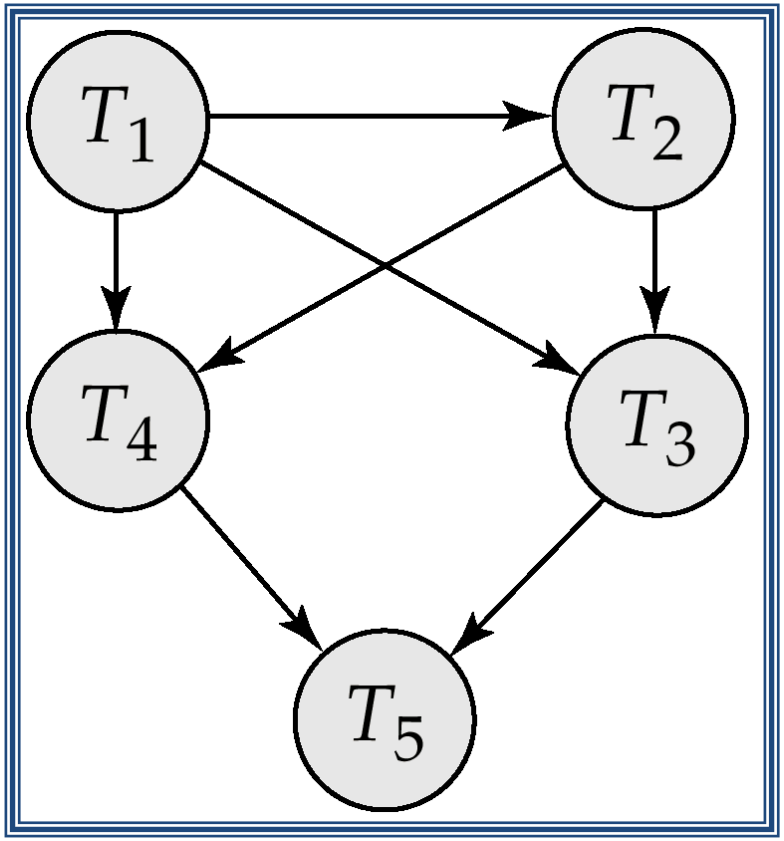


fig. 15.21

T_3	T_4	T_7
read(Q)		
	write(Q)	
		read(Q)
write(Q)		
		write(Q)