

# Topic1

---

An overview of database and DBMS

From

Chapters 1 and 2 of Fundamentals of Database  
Systems, Authors: Elmasri and Navathe

Publisher: Addison Wesley -Pearson

By: A. Abhari

CPS510

Ryerson University

# Topics in this Section

---

- What is a Database System?
- What is a Database?
- Why Database?
- Data Independence
- Relational Systems, and Others
- What applications need DBMS

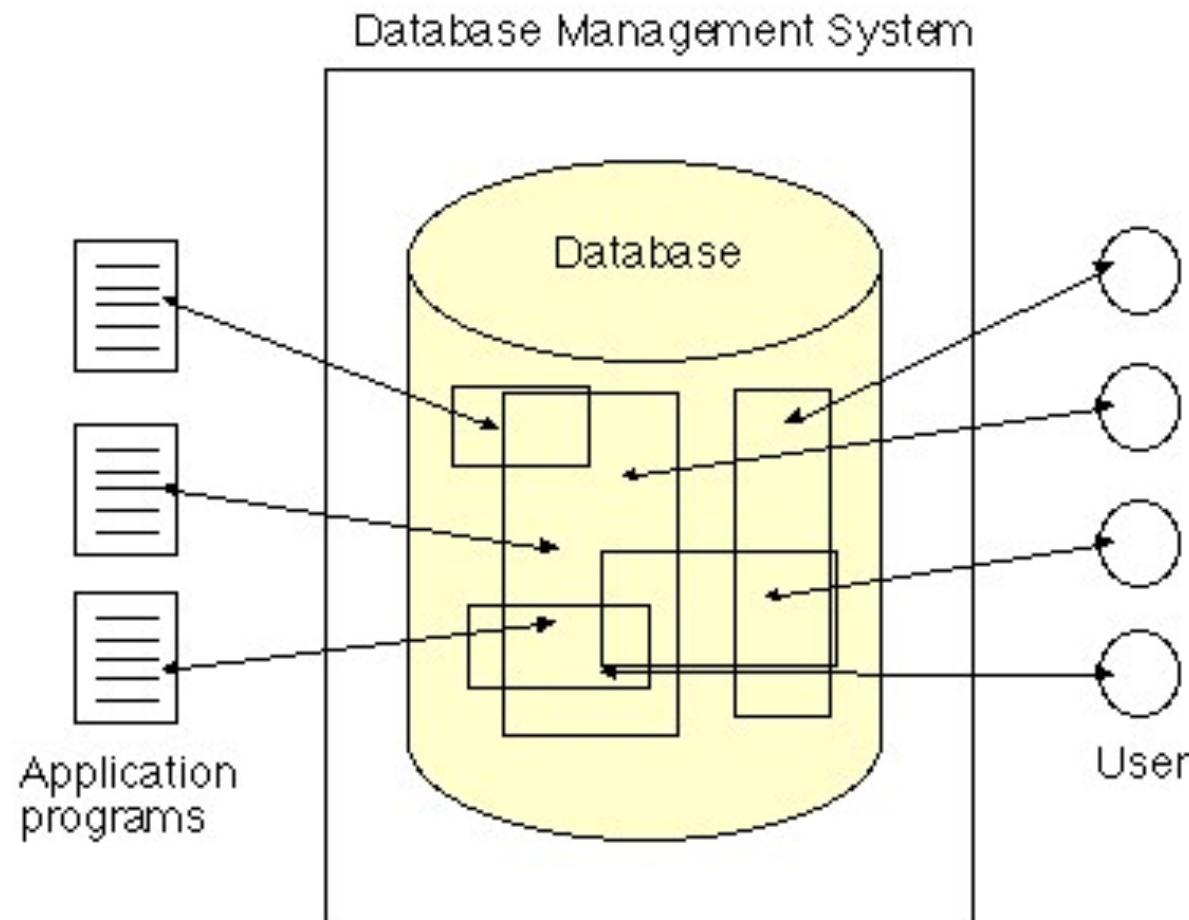
# Database System

---

- Computerized record-keeping system
- A collection of programs to create and maintain database
- Supports operations
  - \* Add or delete files to the database
  - \* Insert, retrieve, remove, or change data in database
- Components
  - \* Data, hardware, software, users

# A Simplified Database System

---



# Database System - Data

---

- Database system may support single user (for small machines) or many users
- When there are many users in organizations:
  - \* Data is integrated: database is unification of distinct files. Any redundancy among these files partly or wholly is eliminated.
  - \* Data is shared: Different users can have access to the same data
- Different users will require different views

# Database System - Data

---

- For example a given database can have EMPLOYEE file that shows the information of employees. Also this database can contain an ENROLLMENT file that shows the enrollment of employees in training courses.
- Personnel department uses EMPLOYEE and educational department uses ENROLLMENT files.

EMPLOYEE

NAME	ADDRESS	DEPARTMENT	SALARY	...
------	---------	------------	--------	-----

ENROLLMENT

NAME	COURSE	...
------	--------	-----

## Database System - Data

---

- ENROLMENT file does not need the department of employees who took a course because it will be redundant information (integrity).
- DEPARTMENT of employees can be used by the users in personnel and education departments (sharing)
- Although users in personnel and education departments share DEPARTMENT portion but they have different views on database.

## Database System - Hardware

---

- The hardware components of database system consist of the disks in which data are stored thus database system can perform
  - Direct access to subset portions of data
  - Rapid I/O
- Data operated on in the main memory

## Database System - Software

---

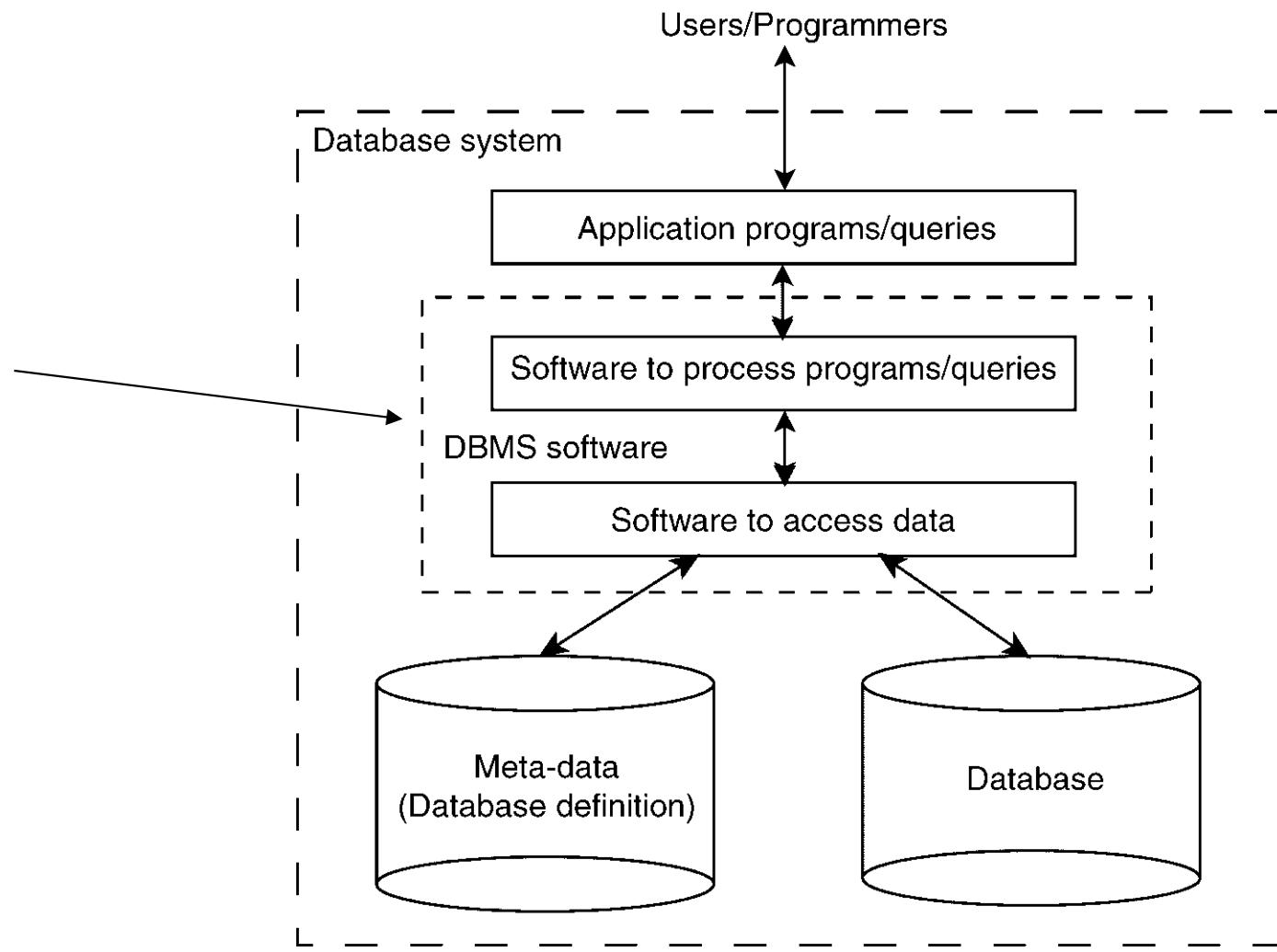
Between physically stored data and users of the systems there is a layer of software referred to as:

- Database manager
- Database server
- Database management system (DBMS)

DBMS shields database users from hardware details

- Note that DBMS is also referred to some products provided by specific vendor. For example BD2

# DBMS in a Database System Environment



# Database System - Software

---

- DBMS is not ( but may come with)
  - \* Application Development Tools
  - \* Application Software
  - \* Transaction Manager (TP Monitor)
  - \* Report Writer
  - \* System utilities
- Note that people often use the term *database* when they really mean *DBMS*. For example “Vendor X’s database” is wrong, it should be “Vendor X’s DBMS”

# Database System - Users

---

- Application programmers
- End users
- Database Administrators (DBA). DBA is a person or team of IT professional/s whose job is to create the database and put in place the technical controls needed to enforce the various policy decisions made by data administrator. Note that DBA is different from Data administrator (DA). DA's job is to decide what data should be stored and who can perform what operations on data (i.e., data security)

# What is a Database?

---

- Collection of persistent data that is used by the application systems of some given enterprise (enterprise is a self contained commercial, scientific, technical or other organization).
- Collection of true propositions: For example the fact “Supplier S1 is located in London” might be such a true proposition
- Made up of entities, relationships, properties (we will talk about it later)

# What is a Database? (persistent data)

---

- Persistent data is different from the data that last for a short time. For example the intermediate results are transient data that last for a short time.
- When persistent data has been accepted by DBMS for entry in database it can be removed from database only by some explicit request to DBMS.
- The earlier term for persistent data was operational data which reflected the original emphasis in database systems for production or operational databases. However databases are now increasingly used for other kind of applications too. For example database can offers decision support via operational data and data warehouse (i.e., summary information)

# What is a Database?

---

- For example here are the example of data that are used in database for following enterprises
  - \* Student data (for a university)
  - \* Patient data (for a hospital)
  - \* Product data (for a company)
- Data can be
  - \* Static (e.g., part#, SIN)
  - \* Dynamic (e.g., quantity, balance)
  - \* Quasi-static (e.g., salary)

# Why Database?

---

- **Shared data:** Not only for existing applications but also new ones can be developed to operate against the same data.
- **Reduced redundancy:** If we need to keep some redundancies it should be controlled. For example the updates should be propagated to all redundant data

# Why Database?

---

- Example of a situation in which redundancy is not completely eliminated

Saving account

account#	customer name	address	balance

Chequeing account

account#	customer name	address	balance

# Why Database?

---

- **Reduced inconsistent data:** Inconsistency happens when one of the redundant data has been updated and the other has not. Propagating updates is used to avoid inconsistency
- **Transaction support:** By putting several update operations in one transaction we can guarantee that either all of them are done or none of them are done. For example the operations for transferring cache from account A to account B can be put in one transaction.

## Why Database?

---

- **Support for data integrity:** Ensures that the data in database is accurate.

For example:

- We shouldn't have an employee working in non existing department.
- We shouldn't have number of hours entered as 400 instead of 40
- Inconsistency can lead to the lack of integrity.

# Why Database?

---

**Security enforcement:** Ensuring that the only means of access to a database is through proper channels: By:

- Restricting unauthorized data
- Different checks (security constraints) can be established for each type of access (retrieve, insert, delete, etc.)
  - \* Example: Course marks database
    - » A student can have access to his/her own mark
      - Should not be able to see other student's marks
    - » TA might have access to enter marks for the current assignment only
      - Should not be allowed to change marks for the other assignments/tests
    - » Instructor can have full access to the course database

# Why Database?

---

- **Support for standards**
  - \* Due to central control of database.
    - » Example
      - Address must be two lines
      - Each line 40 characters (maximum)

# Why Database?

---

- **Conflicting requirements can be met**
  - \* Knowledge of overall enterprise requirements as opposed to individual requirements
    - » System can be designed to provide overall service that is best for the enterprise
    - » Data representation can be selected that is good for most important applications (at the cost of some other applications).

# Data Independence

---

- Data independence
  - \* Traditional file processing is data-dependent
    - » Knowledge of data organization and access technique is built into application logic and code
- **Examples of situations in which the stored representation might be subject to change:**
  - » An application program written to search a student file in which records are sorted in ascending order by **student#** fails if the sort order is reversed
  - » Representation of numeric data
    - binary or decimal or BCD
    - fixed or floating point
    - real or complex

# Data Independence

---

- » Representation of characters
  - ASCII (uses 1 byte)
  - Unicode (uses 2 bytes)
    - used in JAVA
  - Universal character set (UCS)
    - UCS-2 (uses 2 bytes – essentially Unicode)
    - UCS-4 (uses 4 bytes)
- » Unit for numeric data
  - inches or centimeters
  - pounds or kilograms
- » Data encoding

Red = 1, Blue = 2, ...  
== changed to ==> Red = 0, Blue = 1, ...

# Data Independence

---

- In database systems DBMS immune applications to such changes
- In database systems the logical and physical representation of data are separated
- Using database allows changes to application programs without changing the structure of the underlying data and *vice versa*. So the database can grow without impairing existing applications. For example without requiring any changes to the existing applications a “unit cost” **field** can be added to the “part” **stored record** in the “parts” **stored file** of the database shown in Fig 1.7 of the text book.

# Relational Systems

---

- Introduction of relational model in 1969-70 was the most important innovation in database history
- Relational systems are based on logic and mathematics. *Relation* is basically a mathematical term for a table.
- In relational systems data is perceived as tables, only and operators derive new tables from existing
- Relational systems are not pointer based (to the user). Although they may use pointers at the level of physical implementation.

# Relational Systems-SUPPLIER Table

---

<b>supplier#</b>	<b>supplier_name</b>	<b>city</b>
1	Acme Supplies	Toronto
2	Sona Systems	Ottawa
3	AtoZ Systems	New York
4	Quality Supplies	London
5	Best Supplies	Saskatoon

# Relational Systems-PART Table

---

<b>part#</b>	<b>part_name</b>	<b>weight</b>
1	Tower case	2.5
2	Sony display	4.5
3	Mother board	0.6
4	Yamaha speakers	2
5	Power supply	3

# Relational Systems-PART-SUPPLIER Table

---

<b>part#</b>	<b>supplier#</b>	<b>quantity</b>
1	1	20
1	2	34
2	2	23
3	3	33
3	5	43
4	5	45
5	4	22

## Relational products

---

They appeared in the market in late 70s and mostly support SQL. Names of some of these products which are based on the relational system are:

- DB2 from IBM Corp.
- Ingres II from Computer Associate International Inc.
- Informix from Informix Software Inc.
- Microsoft SQL Server from Microsoft Corp.
- Oracle 11g from Oracle Corp.
- Sybase Adaptive Server from Sybase Corp.

## Not Relational Systems

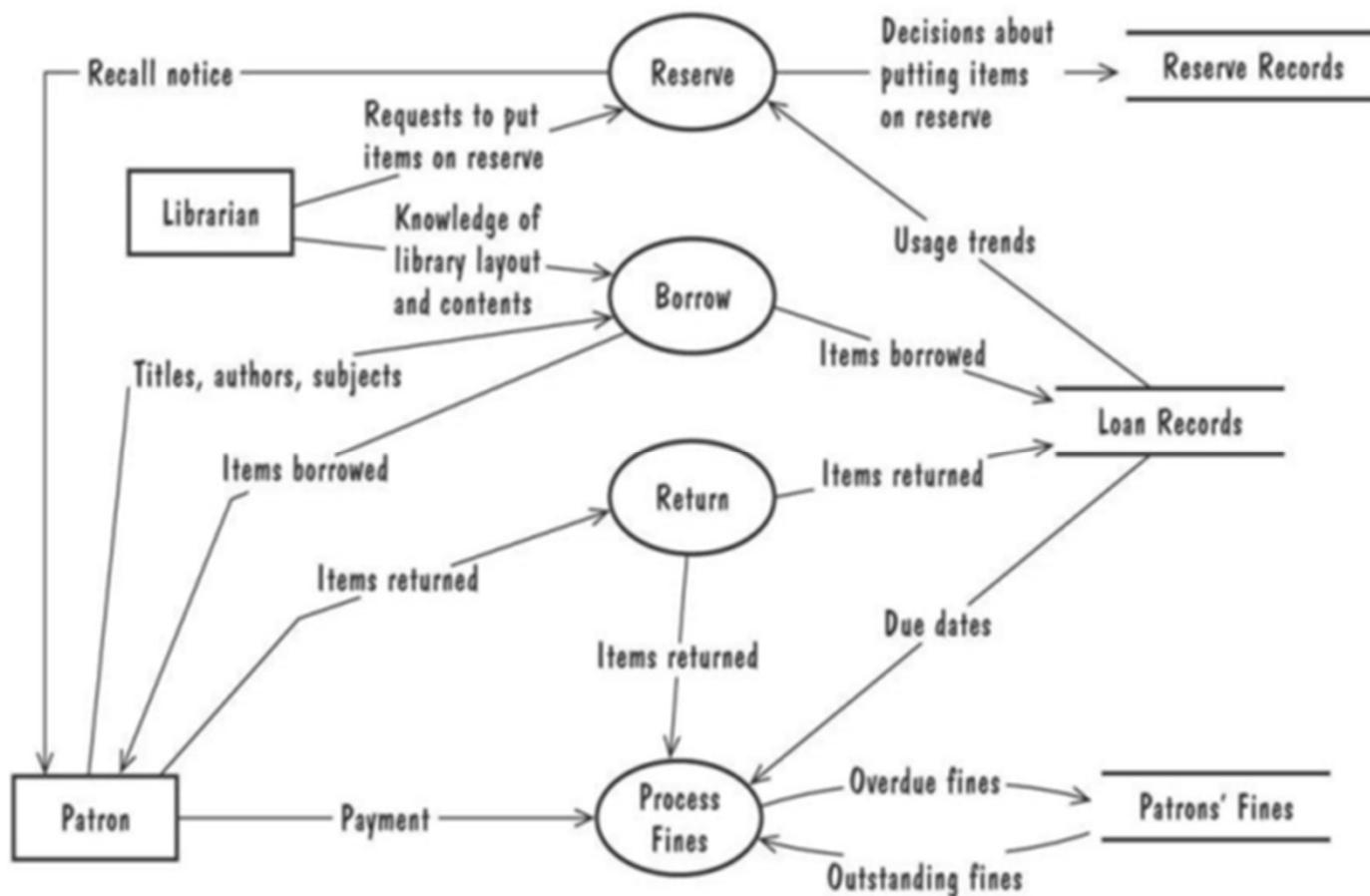
---

- Hierarchic
- Network
- Inverted List
- Object
- Object/Relational
- Multi-dimensional

We will talk about them later

# What applications need DBMS

## Library System: Data Flow Diagram



## What applications need DBMS

---

- How about creating an application that simulates the functions of a simple watch?
- See the UML diagrams of this application in the section called Review of Software Engineering in the Content Section.

# Topic 2

---

Database System Architecture  
From  
Chapters 1 and 2 of Fundamentals of Database  
Systems, Authors: Elmasri and Navathe  
Publisher: Addison Wesley - Pearson

By: Abdolreza Abhari  
CPS510  
Ryerson University

# Topics in this Section

---

- Three levels of architecture
- Mappings
- Database Administrator (DBA)
- Database Management System (DBMS)
- Database Communications
- Client/Server Architecture
- Utilities
- Distributed Processing

# Data Modeling: Schemas and Instances

---

- Before start to talk about database architecture note that in any data model, it is important to distinguish between
  - » description of the database (**database *schema***)
  - » database itself (***instance*** of a database)
- Database schema
  - \* Describes the database
  - \* Specified during the database design phase
    - » Not expected to change frequently
  - \* Most data models have a notation for graphical representation of schema

# Data Modeling: Schemas and Instances

---

Example schema: SUPPLIER-PARTS database

## SUPPLIER

supplier#	supplier_name	city
-----------	---------------	------

## PART

part#	part_name	weight
-------	-----------	--------

## PARTS\_SUPPLIER

part#	supplier#	quantity
-------	-----------	----------

# Data Modeling: Schemas and Instances

---

- Database instance
  - » Refers to the data in the database at a particular moment in time
  - » Many database instances can correspond to a particular schema
  - » Every time we insert, delete, update the value of a data item, we change one instance of database to another
  - » DBMS is partially responsible for ensuring that every instance satisfies
    - Structure and constraints specified in the database schema
  - » See the example instance of SUPPLIER-PARTS database shown before

# Three Levels of Architecture

---

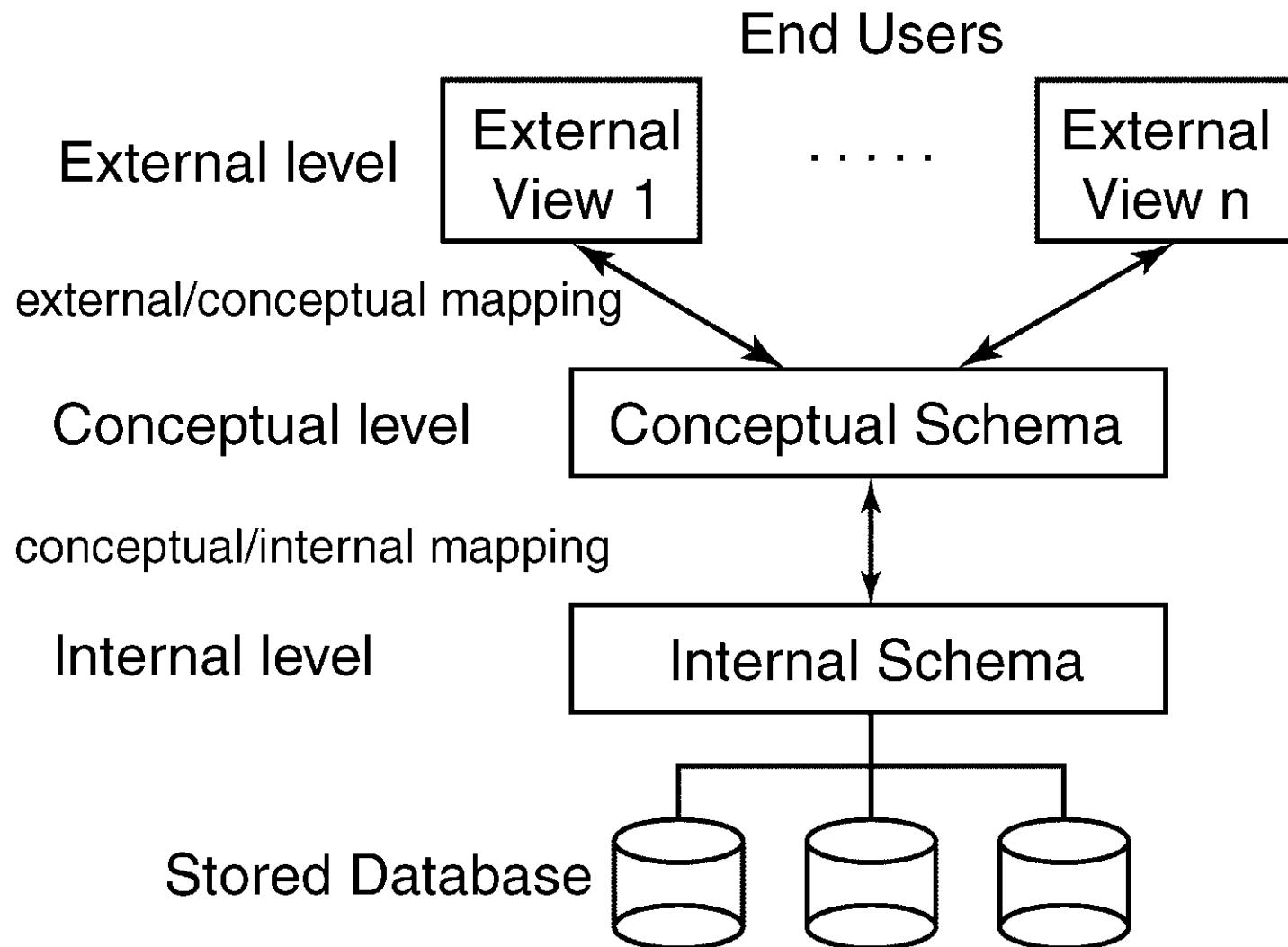
- Three level architecture is also called ANSI/SPARC architecture or three schema architecture
- This framework is used for describing the structure of specific database systems (small systems may not support all aspects of the architecture)
- In this architecture the database schemas can be defined at three levels explained in next slide

# Three Levels of Architecture

---

- Internal level: Shows how data are stored inside the system. It is the closest level to the physical storage. This level talks about database implementation and describes such things as file organization and access paths. Note that relational model has **nothing** explicit to say regarding the internal level
- Conceptual level: Deals with the modeling of the whole database. The conceptual schema of database is defined in this level
- External level: This level models a user oriented description of part of the database. The views for individual users are defined by means of external schemas in this level

# Three Levels of Architecture



# Three Levels of Architecture-Example

External view 1

E_no	F_name	L_name	Age	Salary
------	--------	--------	-----	--------

External view 2

Empl_No	L_name	B_no
---------	--------	------

Conceptual level

Empl_No	F_name	L_name	DOB	Salary	Branch_No
---------	--------	--------	-----	--------	-----------

Internal level

```
struct EMPLOYEE {
    int Empl_No;
    int Branch_No;
    char F_name [15];
    char L_name [15];
    struct date Date_of_Birth;
    float Salary;
    struct EMPLOYEE *next; //pointer to next employee record
}; index Empl_No; index Branch_No; //define indexes for employees
```

# Mapping

---

- Mapping is the key for providing data independence. Here is more explanation on data independence based on three-level architecture.
- Data independence is the capacity to change the schema at one level without having to change the schema at the next higher level
- Two types of data independence are
  - \* Logical data independence
  - \* Physical data independence

# Mapping - Data Independence

---

- Logical data independence (provided by external/conceptual mapping)
  - \* Ability to modify conceptual schema without changing
    - External views
    - Application programs
  - \* Changes to conceptual schema may be necessary
    - Whenever the logical structure of the database changes
      - Due to changed objectives
  - \* Examples
    - » Adding a data item to schema
      - Adding price of a part to PART table
    - » Adding PROJECT table to the SUPPLIER-PARTS database

# Mapping - Data Independence

---

- Physical data independence (provided by conceptual/internal mapping)
  - \* Ability to modify internal or physical schema without changing
    - Conceptual or view level schema
    - Application programs
  - \* Changes to physical schema may be necessary to
    - Improve performance of retrieval or update
  - » Example: Adding a new index structure on ***city***
- Achieving logical data independence is more difficult than physical data independence
  - » Because application programs heavily rely on the logical structure of the data they access

# Database Administrator

---

- Participates in conceptual database design
- Determines how to implement conceptual schema
- Teach users, and help them report
- Implement security and integrity
- Implement unload/reload utilities
- Monitor and tune database performance

# DBMS (Languages)

---

- Users interact with database with data sublanguage (embedded within a host language) which consists of at least two types of languages
  - \* DDL: To define the database
    - Used to define the database
      - For defining schemas at various levels
    - Required in building databases
    - DBA and database designers are typical users
    - Commonly referred to as *data definition language*
  - \* DML: To manipulate data
    - Used to construct and use the database
      - Facilitates retrieval, insertion, deletion and updates
    - Typical users are “End Users”
    - Referred to as *data manipulation language*

# Database Management System

---

- DDL processor / compiler
- DML processor / compiler
- Handle scheduled and *ad hoc* queries
- Optimizer and run-time manager
- Security and integrity
- Recovery and concurrency
- Data dictionary :The **data dictionary** is a system database that contains "data about the data". That is *definitions* of other objects in the system, also known as *metadata*
- Performance tuning utilities

# Support for System Processes

---

- Data Communications interface
- Client Server Architecture
- External tool support: query, reports, graphics, spreadsheets, statistics
- Utilities: unload/reload, stats, re-org
- Distributed processing

# Topic 3

---

ER Model and Deriving Relational Schema  
From  
Chapters 3 and 4 of Fundamentals of Database  
Systems, Authors: Elmasri and Navathe  
Publisher: Addison Wesley -Pearson

By: Abdolreza Abhari

CPS510

Ryerson University

# Topics in this Section

---

- Design of the conceptual schema
- Entity-relationship (ER) model
  - \* Entities
  - \* Relationships
  - \* Attributes
- ER diagrams
- Deriving relational schema from ER model

# Design of the Conceptual Schema

---

- *Stage one:* Choice of model: User requirements and real world concepts should go into the design model. If using E-R model, E-R diagram and relational schemas are the results of this stage.

# Design of the Conceptual Schema

---

- *Stage one:* Choice of model: User requirements and real world concepts should go into the design model. If using E-R model, E-R diagram and relational schemas are the results of this stage.
- *Stage two:* Normalization: Conceptual schema are then used in normalization. This further leads to adjusted diagrams and a normalized relational model. (will be discussed later)

# Design of the Conceptual Schema

---

- *Stage one:* Choice of model: User requirements and real world concepts should go into the design model. If using E-R model, E-R diagram and relational schemas are the results of this stage.
- *Stage two:* Normalization: Conceptual schema are then used in normalization. This further leads to adjusted diagrams and a normalized relational model. (will be discussed later)
- *Stage three:* Optimization: (will be discussed later). The outcome of this stage is the data dictionary and the database description.

# Entity-Relationship Model

---

- Entity-relationship (ER) model
  - \* Introduced by Peter Chen in 1976
- ER model consists of
  - » Entities
  - » Relationships
  - » Attributes
- No single, generally accepted notation
  - \* Note that
    - » You may find variants of the notation used here
    - » You may also find symbols different from the ones we use

# Entities

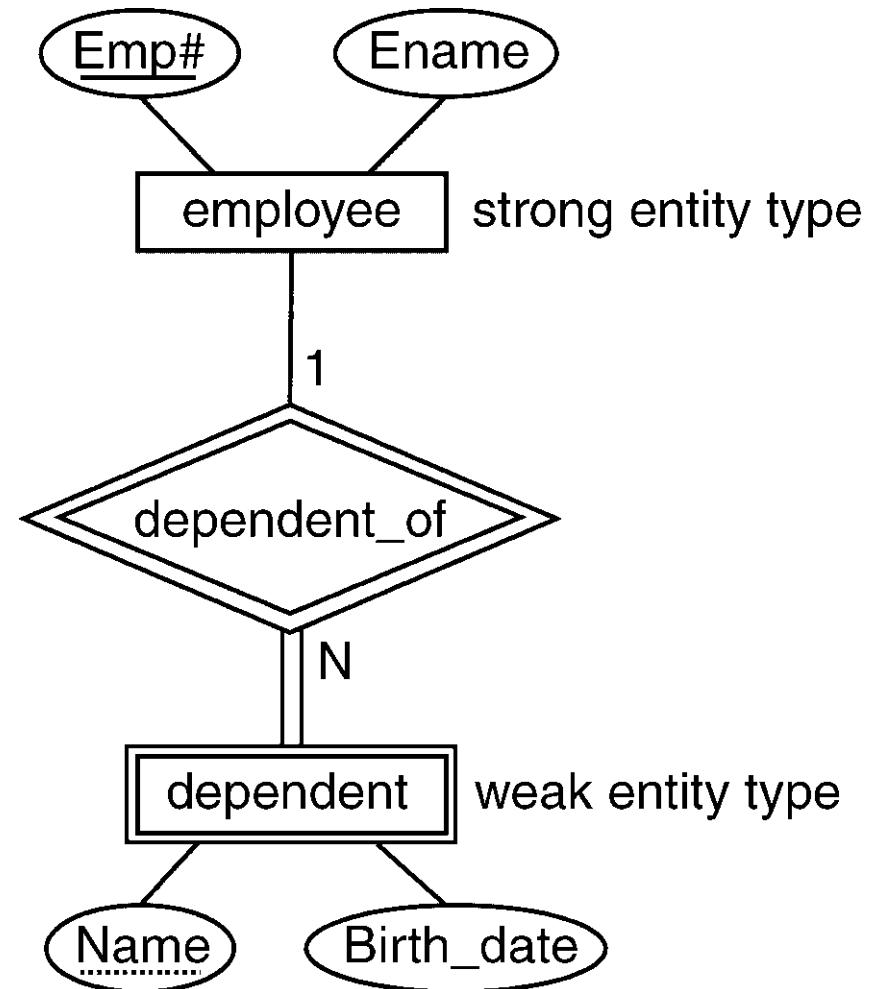
---

- Entity is a “thing in the real world with an independent existence”
  - \* Two types of entities
    - Strong/regular entity (simply called *entity*)
    - Weak entity
  - \* Strong entity types are called *owner* or *dominant* entity types
    - Exist on their own
  - \* Weak entity types are called *dependent* or *subordinate* entity types
    - Existence of weak entity depends on the existence of another strong entity

# Entities (cont'd)

## Weak Entity Example

- Every dependent must be associated with an employee
- If an employee is deleted from the database, dependent must also be deleted
- We can delete a dependent without affecting employee
- Weak entity type is indicated by double outlined box



# Relationships

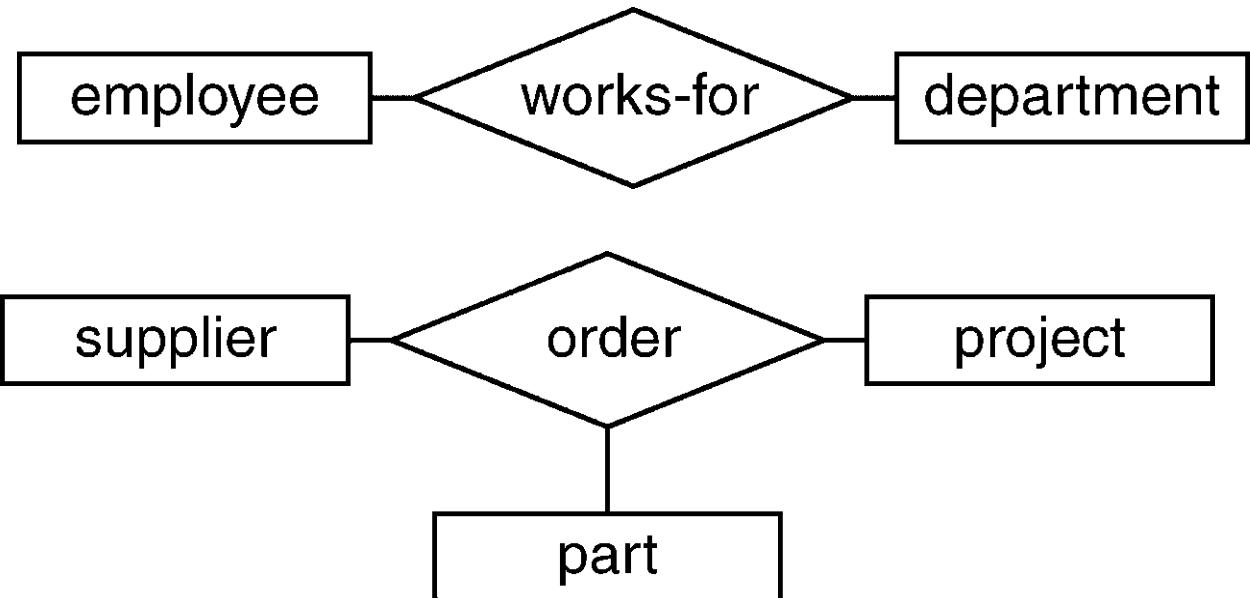
---

A relationship associates entities with one another

Degree of a relationship type is the number of participating entity types

## Examples

- Binary relationship
- Ternary relationship



# Relationships (cont'd)

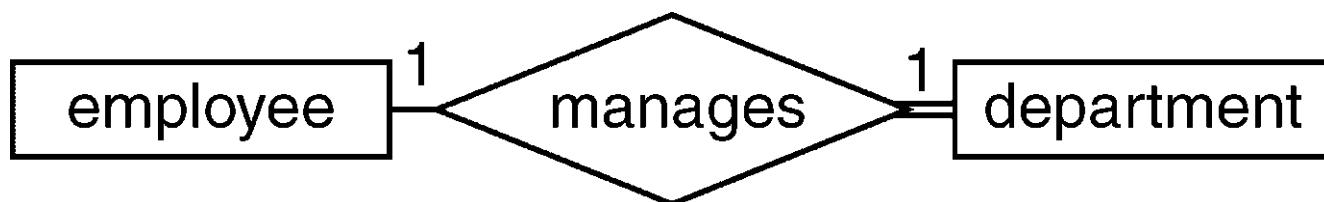
---

- Mapping constraints
  - \* one-to-one (1:1)
  - \* one-to-many (1:N)
  - \* many-to-many (M:N)

## Examples

### one-to-one

- » A manager can manage only one department
- » Each department can have only one manager

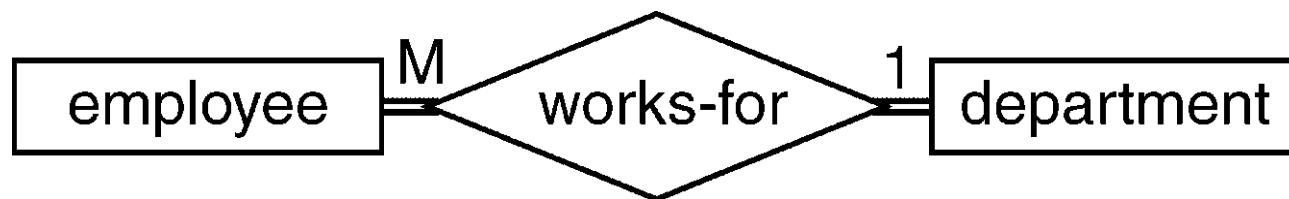


# Relationships (cont'd)

---

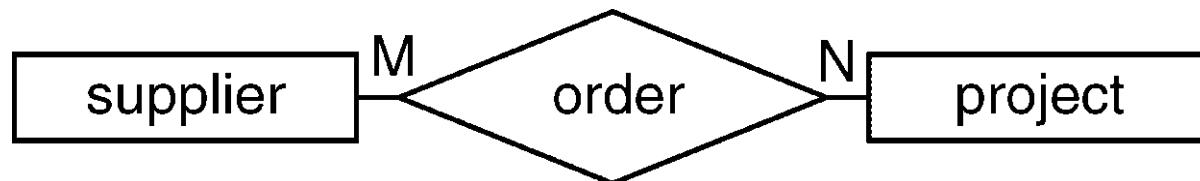
one-to-many

- » A department can have several employees
- » Each employee may work in only one department



many-to-many

- A supplier can supply parts to several projects
- A project can receive parts from several suppliers



# Relationships (cont'd)

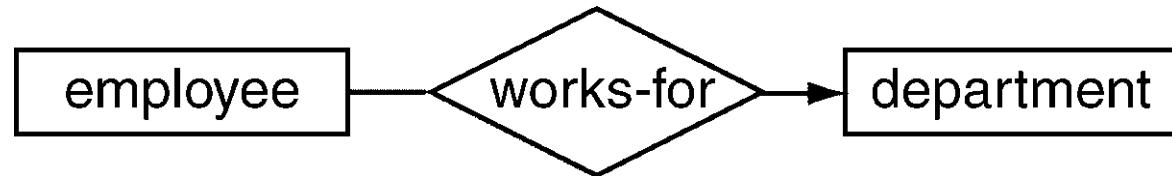
---

- Alternative notation for mapping constraints
  - \* Uses directed line to represent one-to-many or one-to-one mapping

- one-to-one example



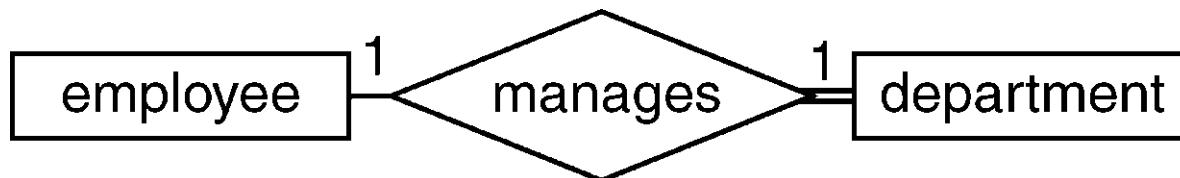
- one-to-many example



# Relationships (cont'd)

---

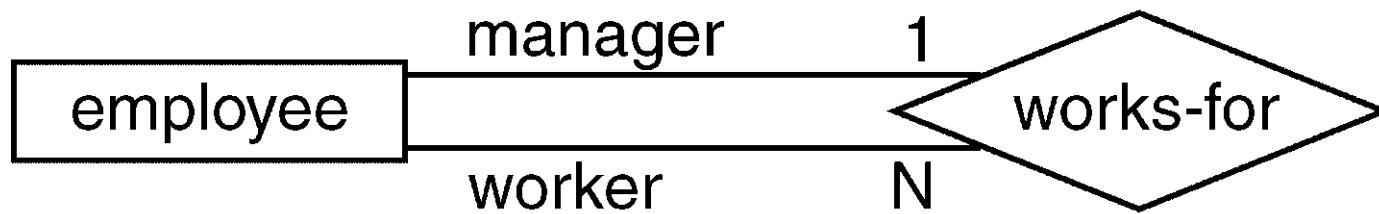
- Participation constraints
  - \* Two types
    - » Total participation (indicated by double lines)
      - Every department must be managed by a manager
      - **department** participation in **manages** relationship is total
    - » Partial participation (indicated by a single line)
      - Not every employee manages a **department**
      - **employee** participation in **manages** relationship is partial



# Relationships (cont'd)

---

- Recursive relationships
  - \* Each entity in a relationship plays a *role*
  - \* In a recursive relationship
    - » An entity participates more than once in different roles
- Example
  - \* 1:N recursive relationship on **employee** entity
    - » Each manager manages several workers
    - » Each worker may have only one manager



# Attributes/Properties

---

- Describe properties of entities and relationships
    - » Both entities and relationships can have attributes
  - Types of attributes
    - » Simple or composite
    - » Single-valued or multi-valued
    - » Stored/based or derived
  - \* An attribute can be a
    - Key or non-key
  - \* An attribute can have a
    - Null value
- in some circumstances

# Attributes (cont'd)

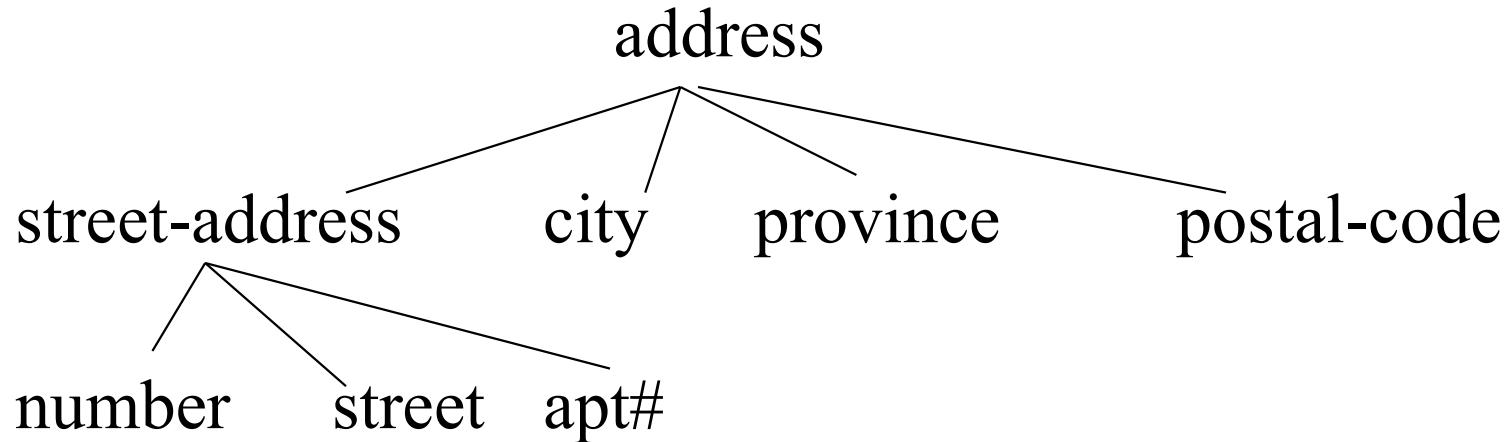
---

- Simple versus composite attributes
  - \* Simple attributes
    - Not divisible
      - called *atomic* attributes
    - Examples
      - **part#**, **weight**
  - \* Composite attributes
    - Consists of several simple attributes
    - Useful if the user refers it
      - sometimes as a unit
      - other times as individual components

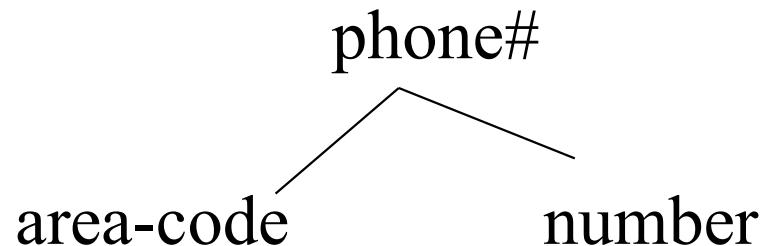
## Attributes (cont'd)

---

Example: Need a list of all suppliers located on Yonge street



Example: Require a list of all customers in 416 area code



# Attributes (cont'd)

---

- Single- versus multi-valued attributes
  - \* Single-valued
    - » Examples: **SIN**, **part-weight**
  - \* Multi-valued
    - » Examples: **college-degrees**, **skills**
- Stored versus derived attributes
  - \* Stored attribute
    - » Example: **date-of-birth**
  - \* Derived attribute
    - » Example: **age**

# Attributes (cont'd)

---

- Key or non-key attributes
  - \* Key attribute
    - » An attribute that is unique
      - distinct for each individual entity instance
      - Examples: **emp#**, **SIN**, **student#**
      - Can used to identify an entity
    - \* Key attributes are shown underlined in the ER diagram
      - » A key attribute may not be a single attribute
        - All attributes that form the key are shown underlined
        - We show only one key attribute
          - Different notation is used in the text (not recommended)

# Attributes (cont'd)

---

## Keys and Identifiers

- Each entity in an entity type needs to be identified uniquely
  - \* Sometimes artificial attributes are created to facilitate
    - » E.g. **student#**, **employee#**
  - \* One or more attributes can be used as an entity identifier
    - » For **marks** entity type, **student#** and **course#** are required to find the **grade**

# Attributes (cont'd)

---

- \* Candidate key
  - » Minimal subset of attributes that uniquely identifies an entity
    - Example: **employee#**  
**SIN**
- \* Primary key
  - » The candidate key chosen by the designer to access each entity
    - Example: **employee#**
  - » Can be defined for strong entities
  - » Weak entities may not have primary keys associated with them
  - » Note:
    - Strong and weak only from a particular application point of view
    - Not inherent in the physical world

# Attributes (cont'd)

---

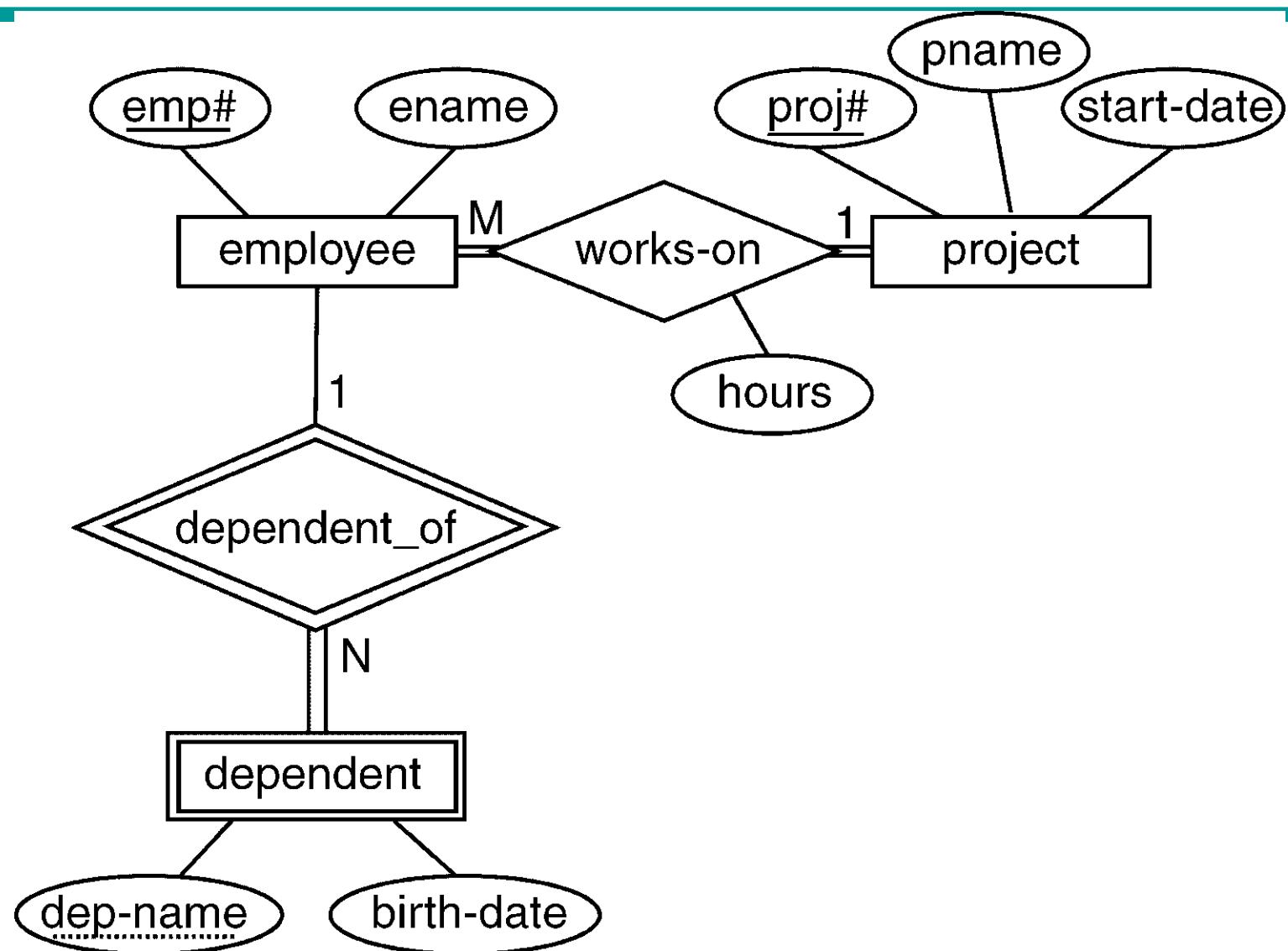
- Primary key for weak entity types
  - \* The entity **dependent** cannot be identified uniquely
    - » Several people may have the same name
    - » We need to identify different dependents of a particular employee
  - \* Primary key of a weak entity type is formed by the primary key of the associated strong entity plus the weak entity discriminator
  - \* Example
    - » **Emp#**, **dep-name** may serve as a primary key for the weak entity type **dependent**

# Attributes (cont'd)

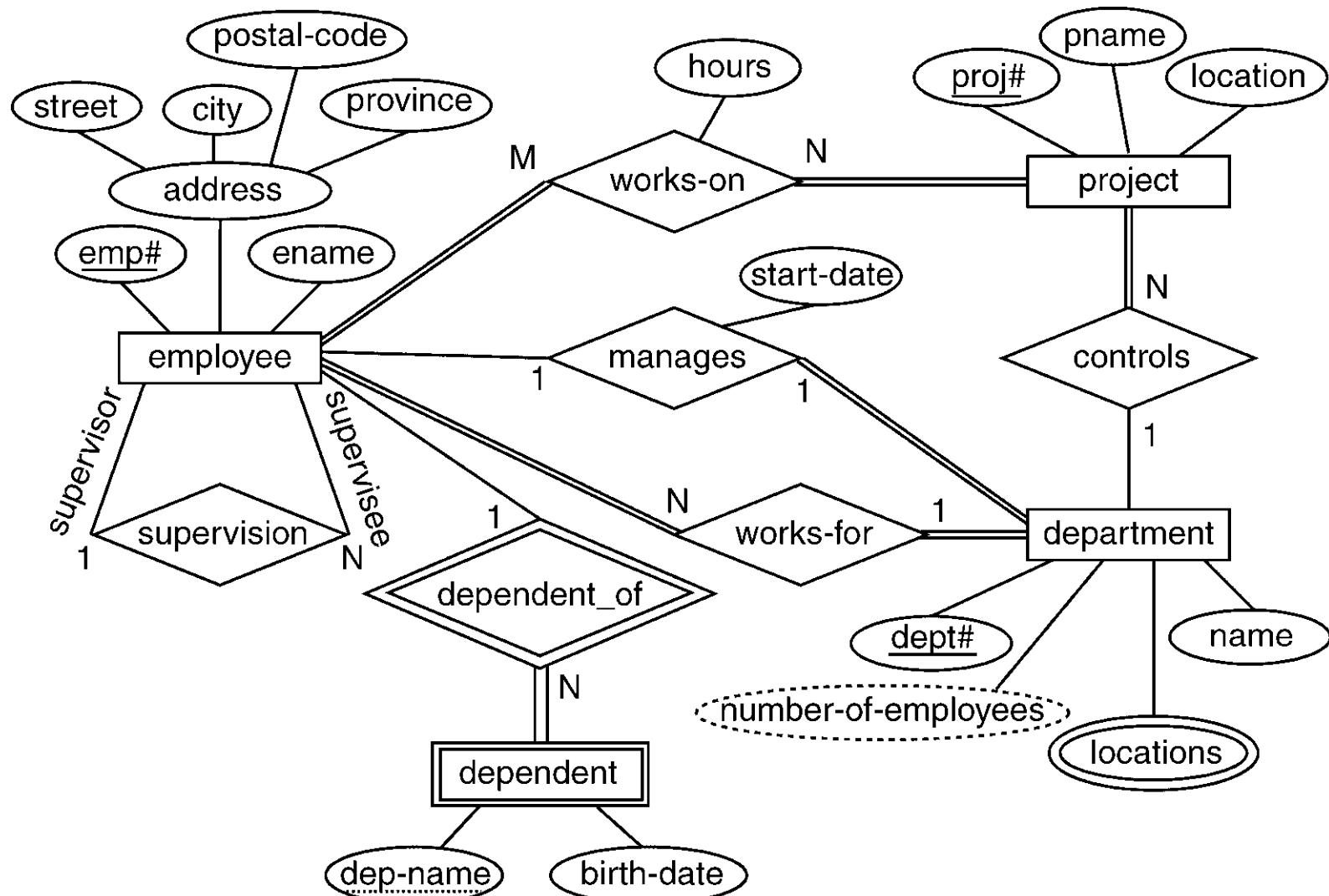
---

- Null values
  - \* A special attribute value NULL is created to represent various things
    - » Not applicable
      - A single-family home may not have **apt#** attribute
    - » Unknown
      - missing information
        - Not known at this time
        - Examples: **citizenship, grade**
      - not known
        - We don't know if the attribute value exists
        - Example: **email-address**

# ER Diagram Example - 1



# ER Diagram Example - 2

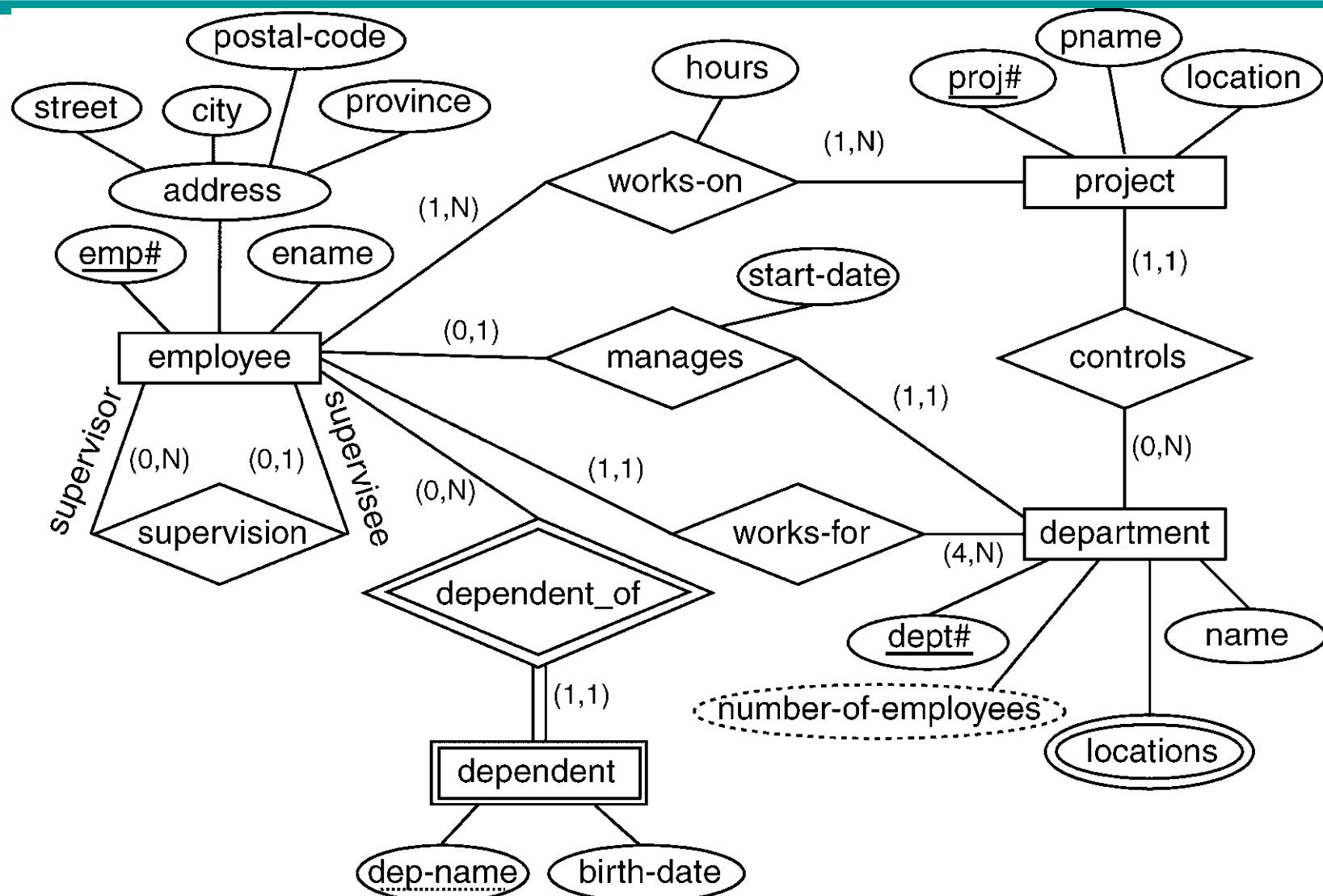


# Alternative Notation for Structural Constraints

---

- Associate a pair of integer numbers
  - \*  $(\text{min}, \text{max})$  where  $0 \leq \text{min} \leq \text{max}$
  - \* Each entity must participate in at least  $\text{min}$  at most  $\text{max}$  relationship instance *at all times*
- More flexible mapping constraints than the three types described before
  - \* Can easily be applied to relationships of any degree
- Participation constraints can also be specified
  - \*  $\text{min} = 0$  implies partial participation
  - \*  $\text{min} > 0$  implies total participation

# ER Diagram Example with (min, max)



# Subclass and Superclass

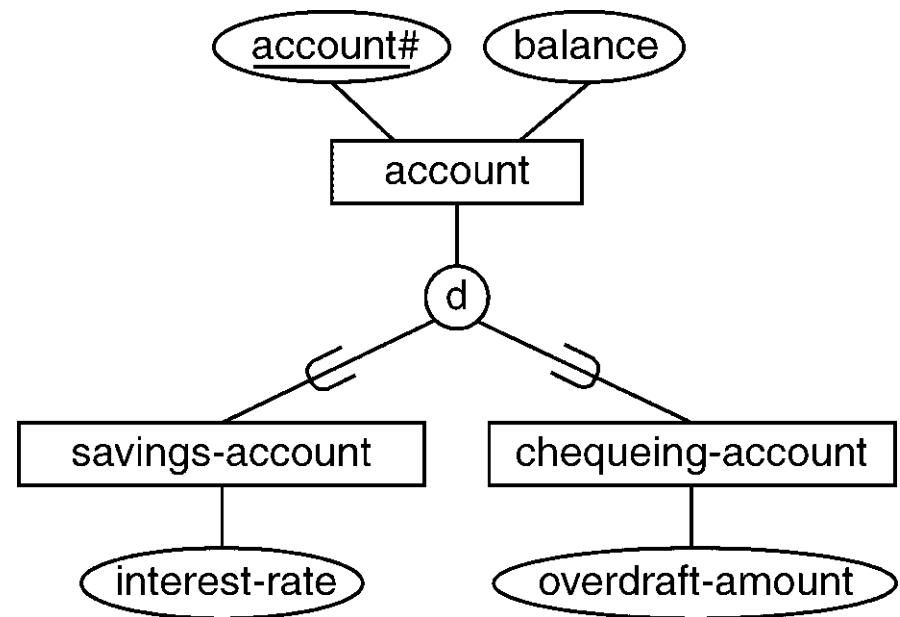
---

- Subclass(Subtypes) and superclass(Supertypes)
  - \* Subclass allows sub-groupings of entities
  - \* **student** entity type can have **part-time** and **full-time** student subclasses
  - \* **student** is said to be *superclass*
  - \* Attribute inheritance
    - » Member of a subclass inherits all the attribute of its superclass
    - » Each subclass can have its own attributes
      - in addition to the inherited attributes

# Specialization

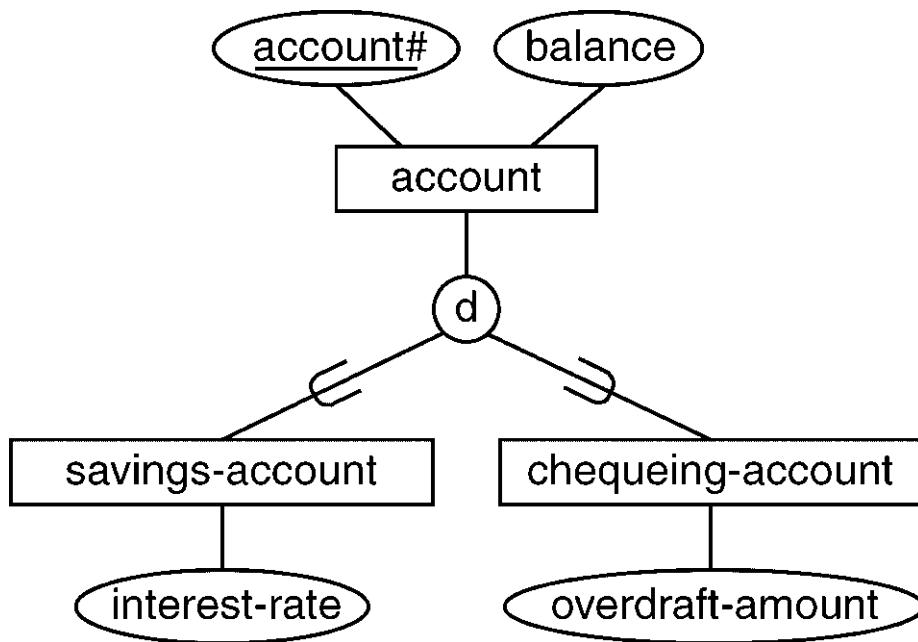
---

- Specialization
  - \* Process of defining a set of subclasses of an entity type
    - » Usually based on some distinguishing characteristic of the entity type
    - » Multiple specializations can be defined on a single entity type
  - \* Example
    - » **account** can be specialized into **savings-account** and **chequeing-account**

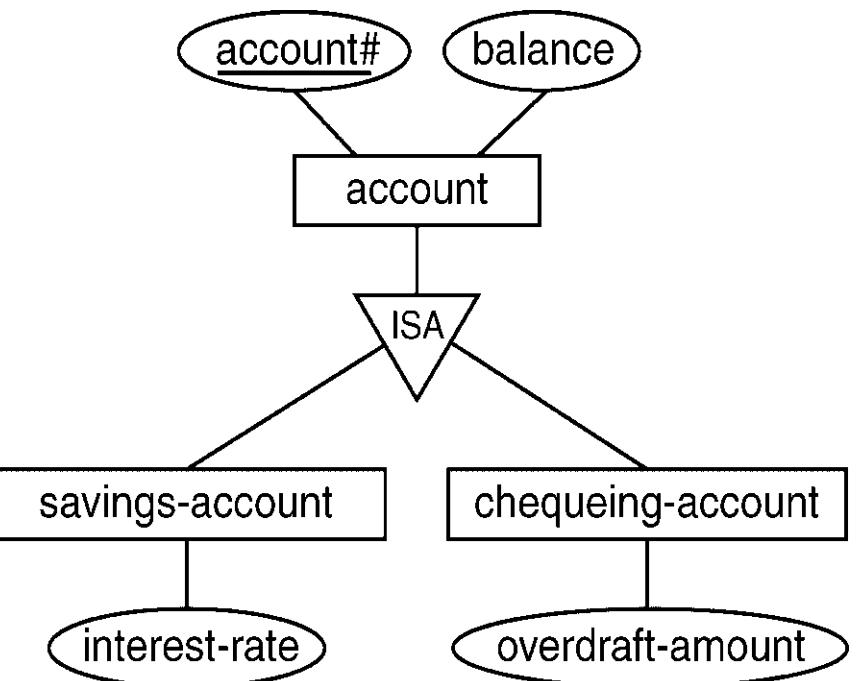


# Specialization (cont'd)

Our notation



ISA notation



# Specialization (cont'd)

---

- Two constraints
  - » Disjointness constraint
  - » Completeness constraint
- Disjointness constraint
  - \* Disjoint
    - » An entity can be a member of at most one of the subclasses of the specialization
    - » We use “d” in ER diagrams to represent disjoint constraint
  - \* Overlapping
    - » The same entity can be a member of more than one subclass of the specialization
    - » We use “o” in ER diagrams to represent overlapping constraint

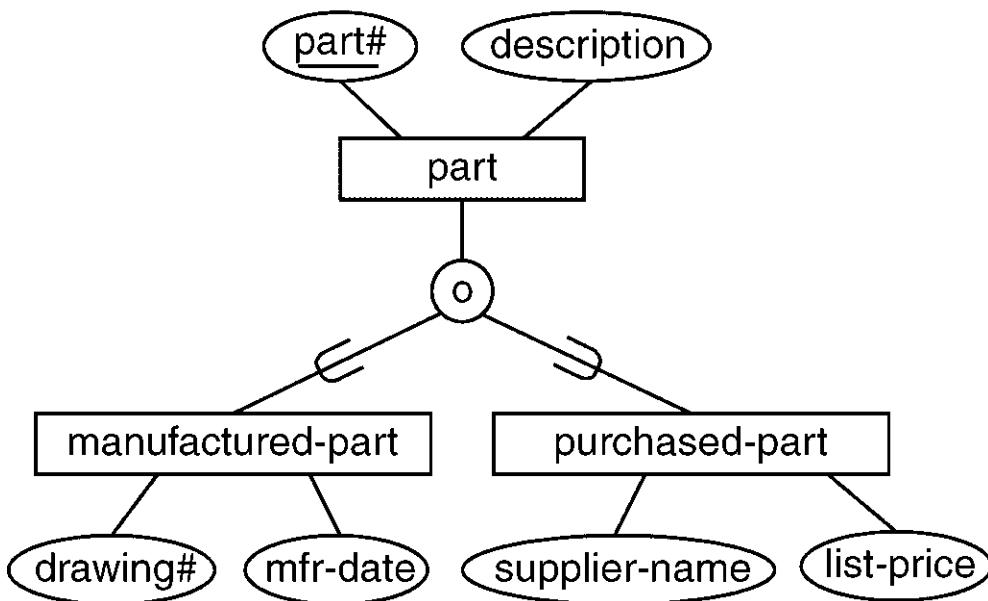
# Specialization (cont'd)

---

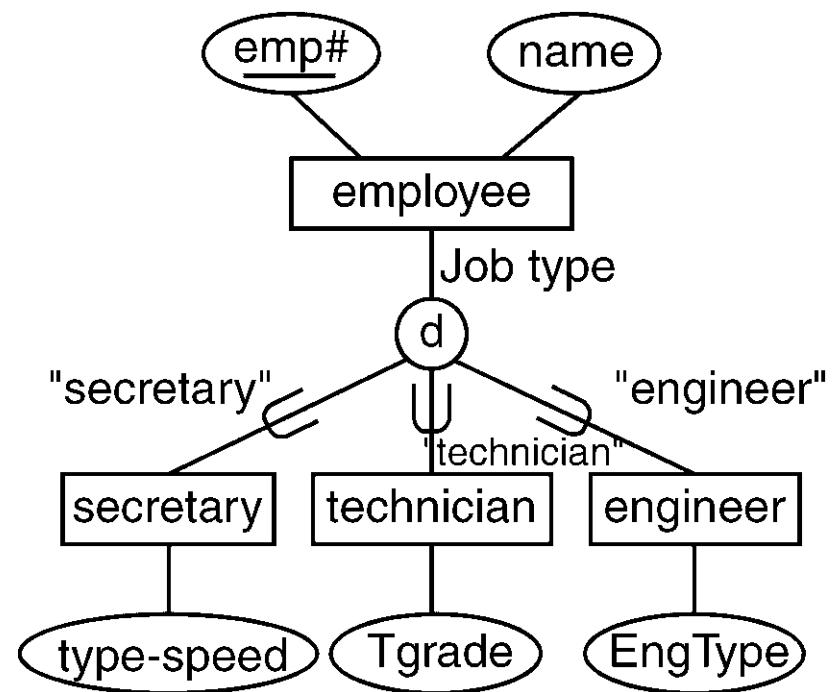
- Completeness constraint
  - \* Total
    - » Every entity in the superclass must be a member of some subclass in the specialization
  - \* Partial
    - » An entity may not belong to any of the subclasses in the specialization
- This leads to four types of specialization
  - » disjoint, total
  - » disjoint, partial
  - » overlapping, total
  - » overlapping, partial

# Specialization (cont'd)

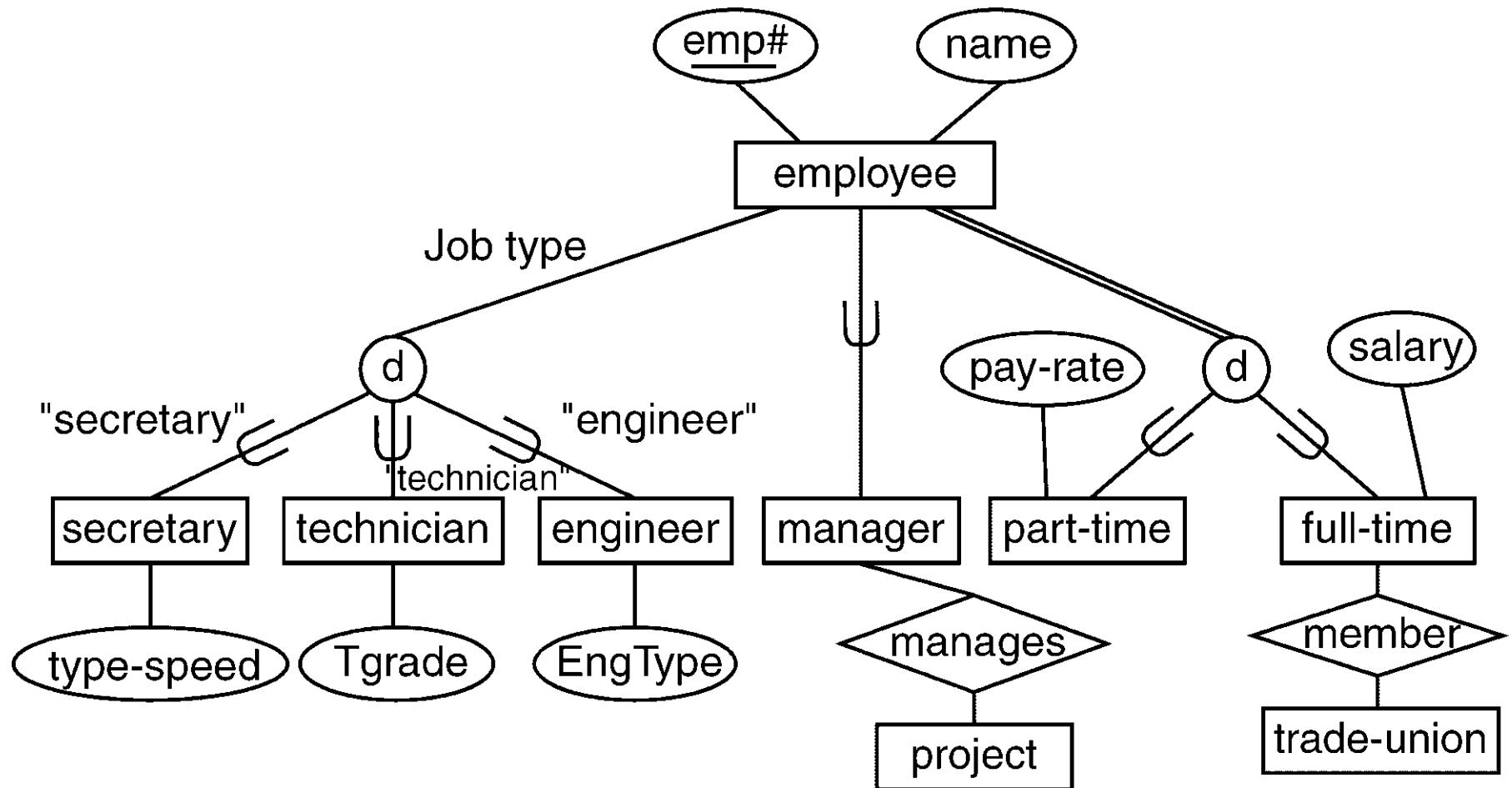
Specialization with overlapping subclasses



Attribute-defined specialization



# Specialization (cont'd)

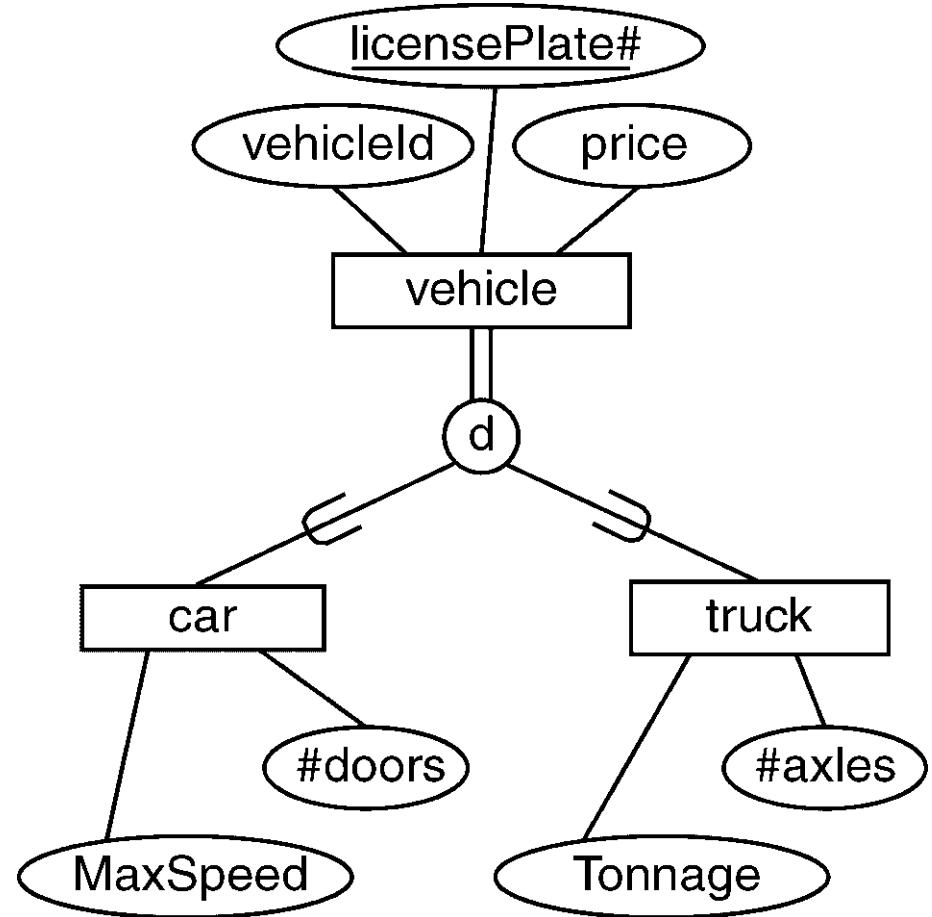
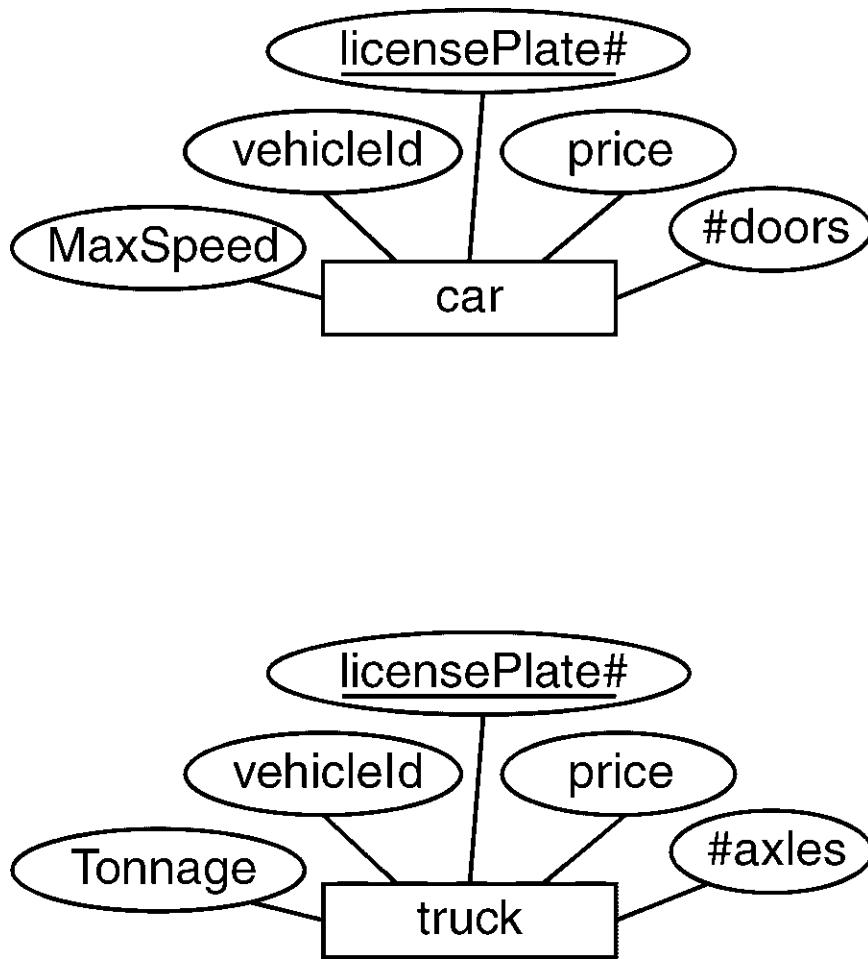


# Generalization

---

- Generalization
  - \* Result of taking the union of two or more lower-level entity types to produce a higher-level entity type
    - The original entity types are special subclasses and the new higher-level entity type is the superclass
    - Functionally the inverse of the specialization process
  - \* We don't use any special notation for generalization
  - \* The original entities that are used in generalization are special subclasses.
  - \* In other words in generalization every higher-level entity must also be a lower-level entity but specialization does not have this constraint.

# Generalization



# Deriving Relational Schema

---

- Fairly straightforward to derive relational schema from the ER diagrams

## Strong Entity

- \* An entity type E with attributes  $A_1, A_2, \dots, A_K$  is represented as a k-degree relation

$$E(A_1, A_2, \dots, A_K)$$

- » Each tuple of the relation represents one entity in the entity type
- » Include only simple components of a composite attribute

# Deriving Relational Schema (cont'd)

---

## Relationship

- \* A relationship R among entity types  $E_1, E_2, \dots, E_K$ 
  - » Let  $P_1, P_2, \dots, P_K$  be the primary keys of the entity sets  $E_1, E_2, \dots, E_K$  respectively
- \* Relationship R has attributes  $A_1, A_2, \dots, A_R$
- \* The relationship R is represented as a  $(k+r)$ -degree relation

$$R(P_1, P_2, \dots, P_K, A_1, A_2, \dots, A_R)$$

# Deriving Relational Schema (cont'd)

---

## Weak entity

- \* A weak entity type W has attributes  $A_1, A_2, \dots, A_W$
- \* Depends on strong entity type S with primary key  $P_S$
- \* The weak entity is represented as

$$W(P_S, A_1, A_2, \dots, A_W)$$

## Multi-valued attribute

- \* A multi-valued attribute  $A_M$  of entity type E (or relationship type R) with primary key  $A_K$  is represented by

$$M(A_K, A_M)$$

- \*  $A_K$  and  $A_M$  together form the primary key to M

# Deriving Relational Schema (cont'd)

---

## Example

- \* The project-employee ER diagram (Example 1) is converted to the following five relations:

**EMPLOYEE** (emp#, ename)

**PROJECT** (proj#, pname, start-date)

**WORKS-ON** (emp#, proj#, hours)

**DEPENDENT** (emp#, dep-name, birth-date)

**DEPENDENT-OF** (emp#, dep-name)

- \* Primary key shown underlined
- \* The last relation is redundant

# Deriving Relational Schema (cont'd)

---

## Example (cont'd)

- \* Problems in representing the weak entity type
  - » Using dep-name as the key means if two dependents of the same employee have the same name we have duplicated keys.
  - » Multiple occurrences of a dependent may be avoided by giving the dependent its own unique identifier
  - » The modified schema is

**EMPLOYEE (emp#, ename)**

**PROJECT (proj#, pname, start-date)**

**WORKS-ON (emp#, proj#, hours)**

**DEPENDENT (dep-id, dep-name, birth-date)**

**DEPENDENT-OF (emp#, dep-id)**

# Deriving Relational Schema (cont'd)

---

For 1:1 and 1:M Relations

- \* We can avoid a separate relation by adding attributes to the associated entity
  - » Reduces redundancy
- \* Example revisited
  - » The revised schema is

**EMPLOYEE** (emp#, proj#, ename, hours)

**PROJECT** (proj#, pname, start-date)

**WORKS-ON** (emp#, proj#, hours)

**DEPENDENT** (dep-id, dep-name, birth-date)

**DEPENDENT-OF** (emp#, dep-id)

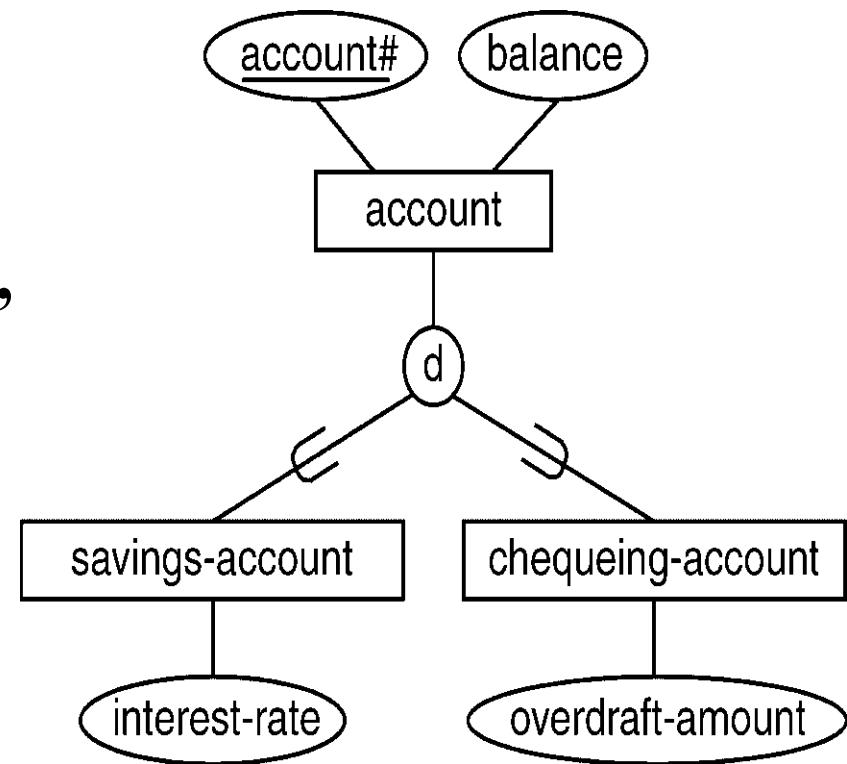
# Deriving Relational Schema (cont'd)

---

- Two methods for deriving relational schema from an ER diagram with specialization/generalization
  - \* Method 1
    - » Create a table for the higher-level entity
    - » For each lower-level entity, create a table which includes a column for each of its attributes plus for primary key of the higher-level entity
  - \* Method 2
    - » Do not create a table for the higher-level entity
    - » For each lower-level entity, create a table which includes a column for each of its attributes plus a column for each attribute of the higher-level entity

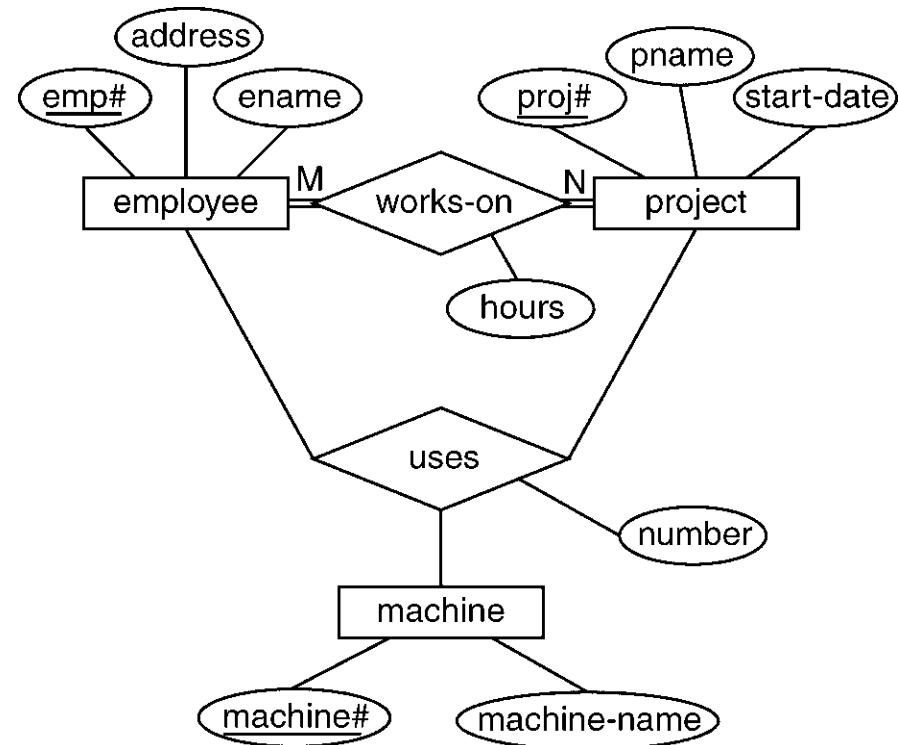
# Deriving Relational Schema (cont'd)

- Method 1
  - \* **account (account#, balance)**
  - \* **savings-account (account#, interest-rate)**
  - \* **chequeing-account (account#, overdraft-amount)**
- Method 2
  - \* **savings-account (account#, balance, interest-rate)**
  - \* **chequeing-account (account#, balance, overdraft-amount)**



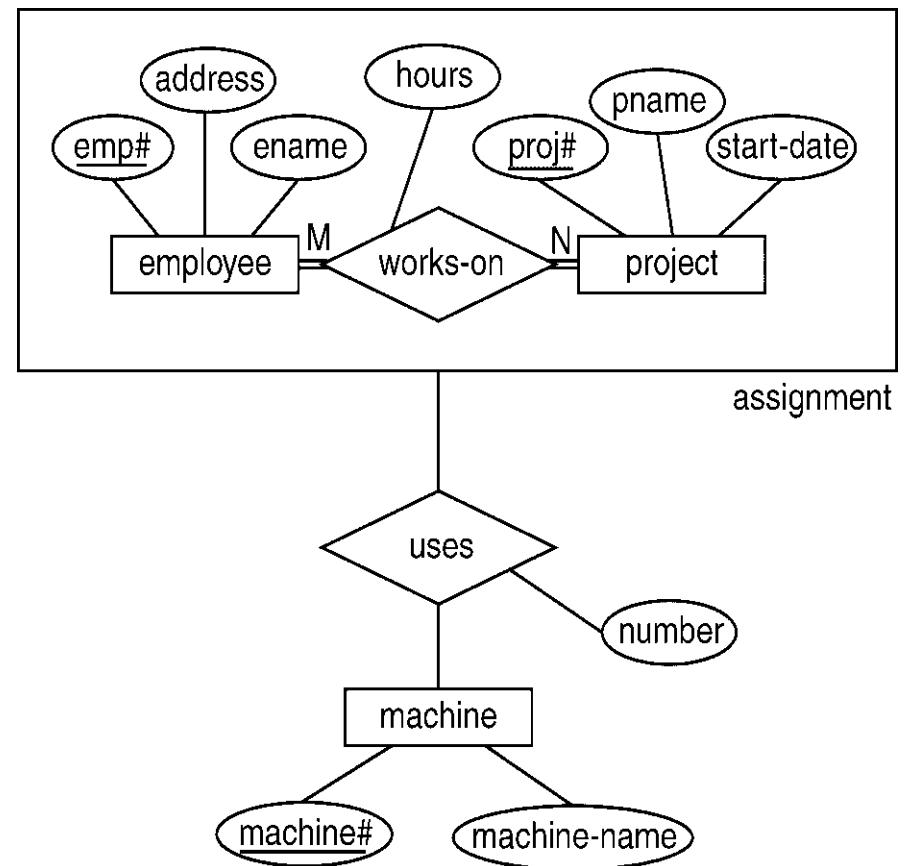
# Aggregation

- Motivation
  - \* A limitation of the ER model
    - » Not possible to express relationship among relationships
  - \* We may have to use two or more relationships
    - » **works-on** and **uses** relationships are independent
    - » But it is complicated because we just wanted to show that when employee works on a project, he/she uses a machine...



# Aggregation (cont'd)

- Aggregation is an abstraction through which relationships are treated as higher-level entities
- Example
  - \* We create a new higher-level entity called **assignment**
  - \* Now we can establish relationships by treating this new entity as a regular entity



# Aggregation (cont'd)

---

- Deriving relational schema
  - \* Transform the higher-level entity
    - » Use the procedure described before
  - \* Transform the aggregate relationship
    - » Entity types participating in the higher-level entity H:  $E_1, E_2, \dots, E_{K-1}$
    - » Let  $P_1, P_2, \dots, P_K$  be the primary keys of  $E_1, E_2, \dots, E_K$  respectively
    - » Attributes of relationship R between entity types H and  $E_K$ :  $A_1, A_2, \dots, A_R$
    - » The relationship is represented by
$$R(P_1, P_2, \dots, P_{K-1}, P_K, A_1, A_2, \dots, A_R)$$

# Aggregation (cont'd)

Example

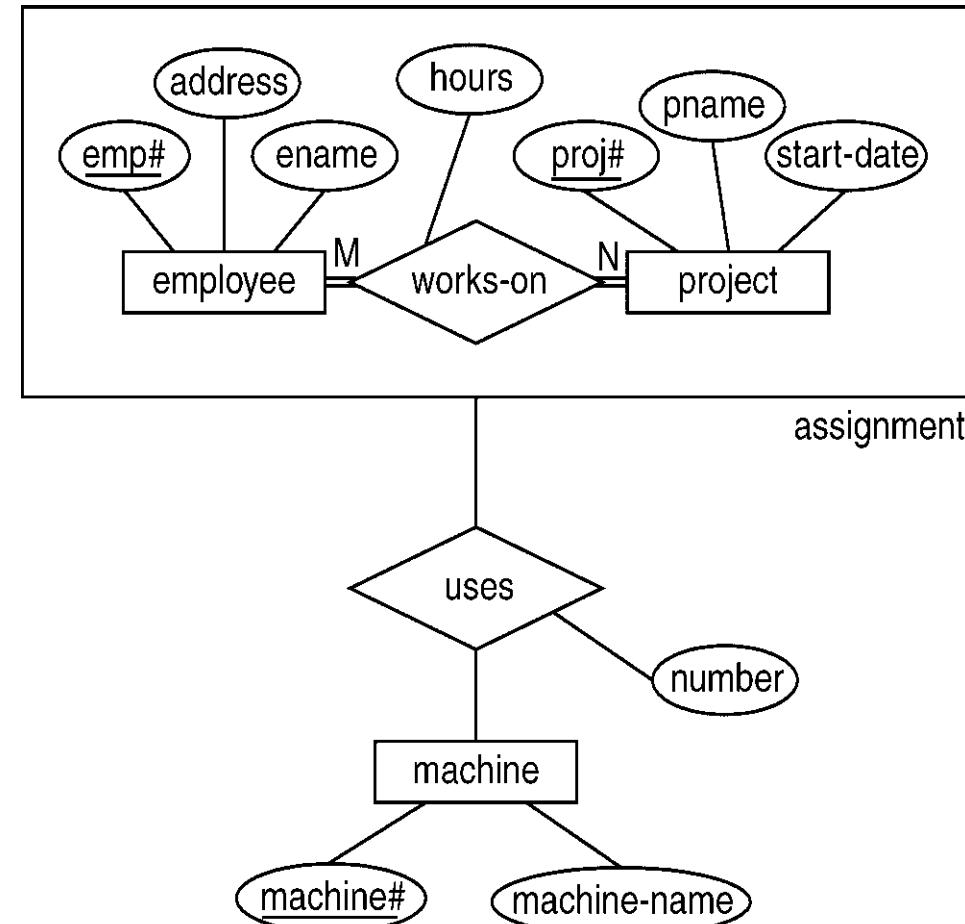
**EMPLOYEE** (emp#,  
ename, address)

**PROJECT** (proj#,  
pname, start-date)

**WORKS-ON** (emp#, proj#,  
hours)

**MACHINE** (machine#,  
machine-name)

**USES** (emp#, proj#,  
machine#, number)



# E/R Diagram and Data Dictionary

---

- As mentioned before, data dictionary is the database designer's database
- The results of E/R diagram can be used to identify the kinds of objects the dictionary needs to support
- For example a weak or strong entity, total or partial participation in a relationship and a supertype or subtype entity and etc., all can be explained in a data dictionary.

# Project: University Database

---

Consider the following requirements for a university database

- The university keeps track of each student's name, address, student number, social insurance number, and the courses they have registered.
- In addition, for undergraduate and graduate students the degree program (BA, BCS, MSc, PhD) they are in is also maintained. (For other students such as special students, exchange students etc. this information is not needed.)

# University Database

---

- For graduate students, a list of degrees held (degree, university, and the year degree was awarded) and their office in the department and phone number are included in the database.
- All graduate students are financially supported either by a teaching assistantship (TA) or by a research assistantship (RA). For the TAs we would like to keep the number of hours per week they are working and for the RAs the research project they are associated with (just research project name).

# University Database

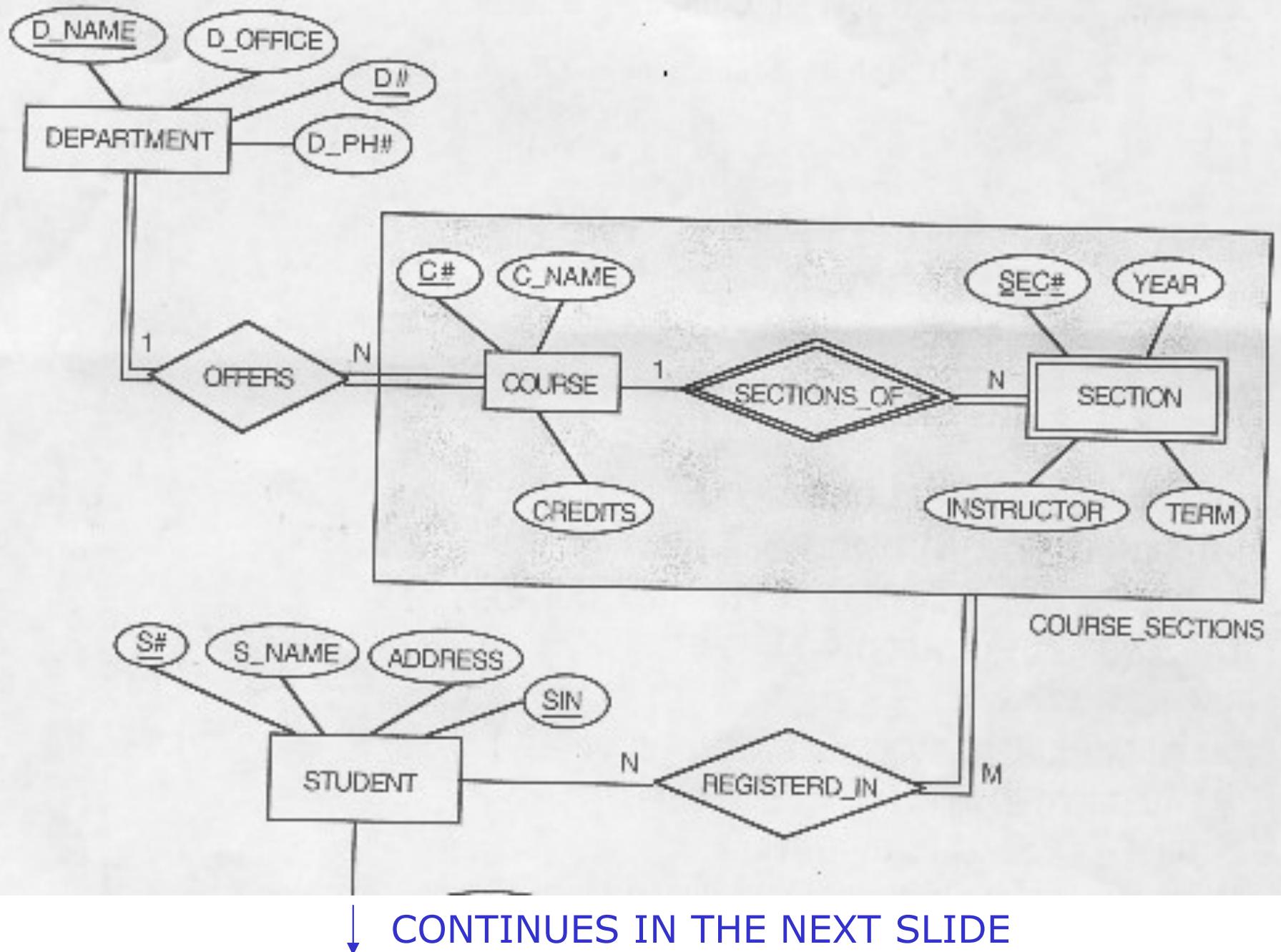
---

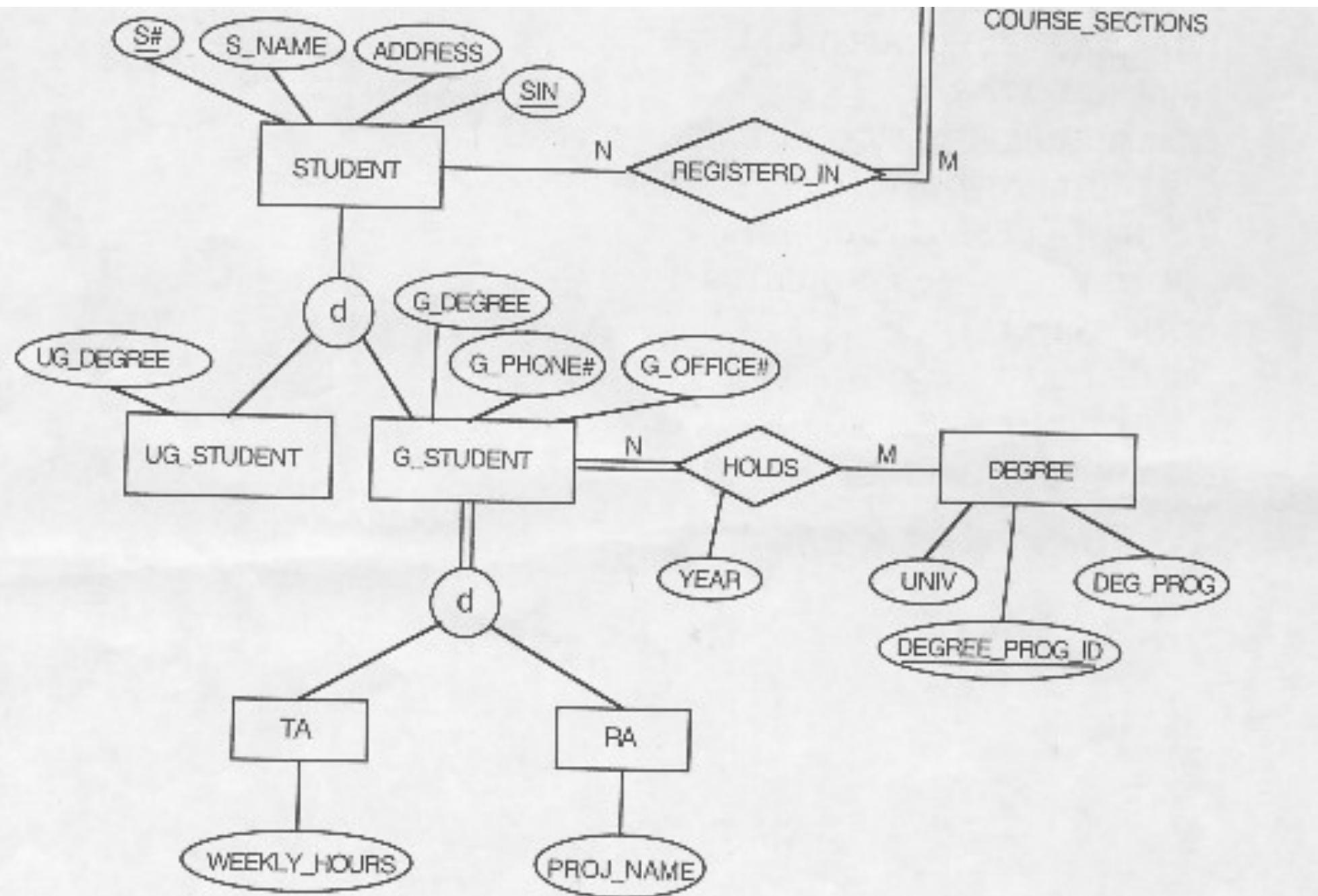
- Each department is represented. The data about departments are its name, department code, office number, and office phone. Both name and code have unique values for each department.
- Each course has a course number, course name, number of credits and the offering department. The value of course number is unique for each course.

# University Database

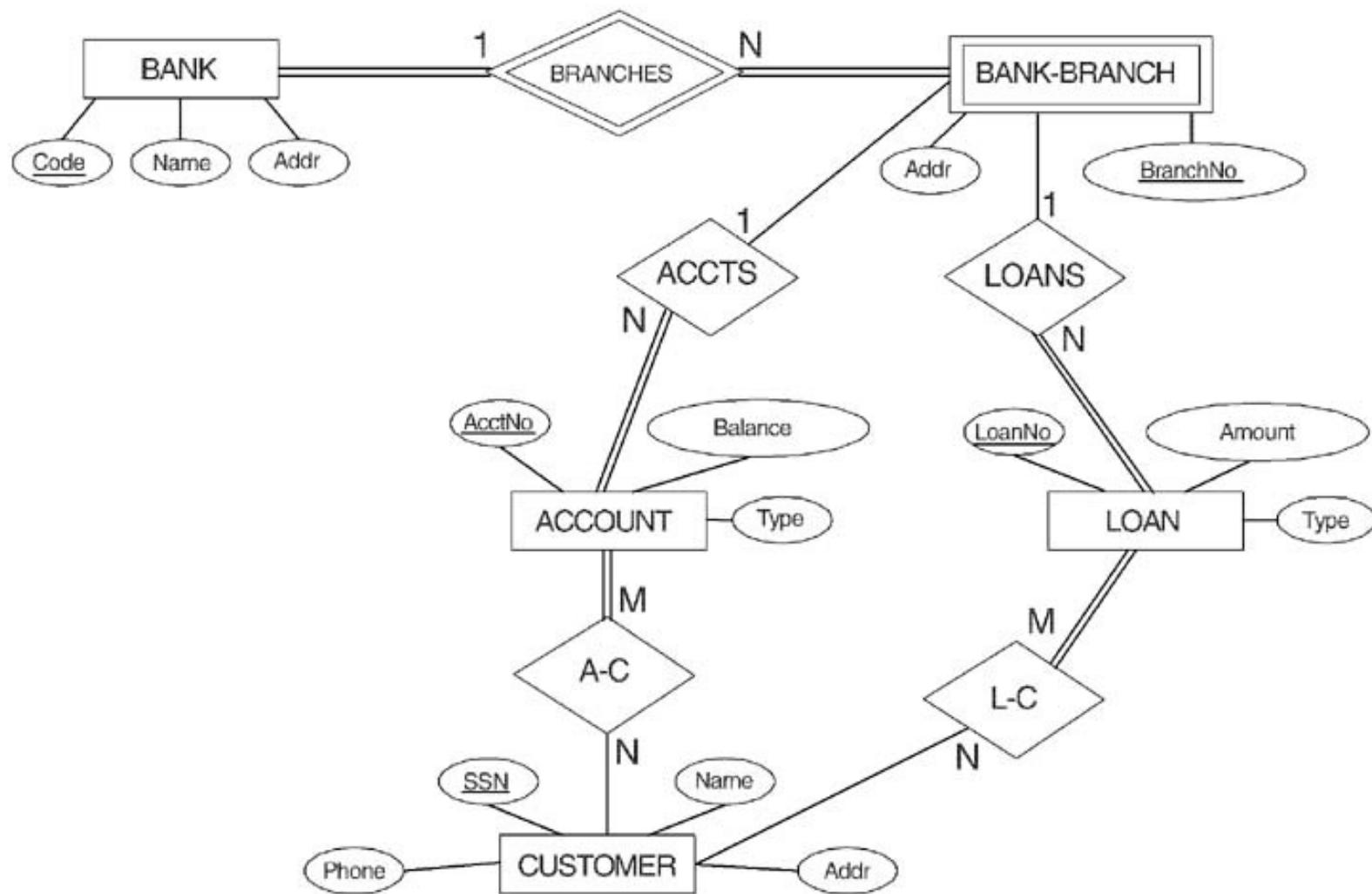
---

- Each section has an instructor, term, year, course, and section number. The section number distinguishes different sections of the same course that are taught during the same year; its values are 1, 2, 3,..., up to the number of sections taught during each year.
- The ER diagram is shown in the next two slides
- Specify a preliminary relational database schema for ER diagram





# Bank Database



# Topic 4

---

## Relational Databases Basic Terms

By: A. Abhari

CPS510,CP8203

Ryerson University

# Topics in this Section

---

- Relational Model
- Relations and relvars
- Optimization
- The catalog
- Transactions

# Relational Model

---

- Relational model uses *table* (called *relation*) to represent a collection of related data values
  - \* Tables and columns are identified by means of names
    - » Rows are called *tuples* and columns are called *attributes*
  - \* The number of attributes (i.e., number of columns) is called the *degree*
    - Degree is also called *arity*
  - \* The data type describing the type of values that can appear in each column is called a *domain*
  - \* A domain is a set of *atomic* values
    - Each value is *indivisible* as far as the relational model is concerned

# Relational Model

---

- Ordering of tuples
  - \* Theoretically, tuples in a relation do not have any particular order
  - \* In practice, relations are stored as files
    - » There is always an order among the records
- Ordering of values
  - \* Since we give names to columns, this ordering is not important
  - \* When a relation is implemented as a file, the attributes are ordered

# Relational Model

---

- A relation schema R of degree n is represented as

$$R(A_1, A_2, \dots, A_n)$$

- An n-tuple t in a relation r(R) is denoted by

$$t = < v_1, v_2, \dots, v_n >$$

where  $V_i$  is the value of attribute  $A_i$

»  $t[A_i]$  refers to the value of  $V_i$  in t

»  $t[A_u, A_v, \dots, A_z]$  refers to the subtuple of values

$$< v_u, v_v, \dots, v_z > \text{ in } t$$

- Further notation

» Q, R, S represent relation names

» q, r, s represent relation instances

» t, u, v represent tuples

# Relational Model

---

- Each entity in an entity type needs to be identified uniquely
  - \* Sometimes artificial attributes are created to facilitate this identification
    - » E.g. **student#**, **employee#**
  - \* One or more attributes can be used as an entity identifier
    - » For **marks** entity type, **student#** and **course#** are required to find the **grade**
  - \* Superkey
    - » key attribute + other attributes of relation
      - Example: **employee#**, **SIN**

# Relations and Relvars

---

- Suppose we say in some programming language:  
DECLARE N INTEGER ... ;  
N here is **not** an integer; it's an integer *variable* whose *values* are integers. It is different integers at different times (that's what *variable* means).
- In exactly the same way, if we say in SQL:  
CREATE TABLE T ... ;  
T here is **not** a table. It is a relation (table) *variable* whose *values* are relations (tables). It is different relations (tables) at different times

# Relations and Relvars

---

- *Relations* are **values**; they can thus be "read" but not updated, by definition. (The one thing you can't do to *any* value is update it. E.g., consider the value that's the integer 3.)
- *Relvars* are **variables**; they can thus be "read" **and** updated, by definition. (In fact, "variable" really *means* "updatable." To say that something is a variable is to say, precisely, that that something can be used as the target of an assignment operation.)

# Optimization

---

- The optimizer is a system component that determines how to implement user request
- Relational systems are responsible for locating the desired data by automatic navigation
- **Optimizer** has to do some "smart thinking" in order to support such automatic navigation

For example for simple search for a tuple (record) optimizer can do simple sequential search or use index (if there is an index)

# Catalog

---

- Catalog in a relational system will itself consist of relvars
- The **catalog** is a set of system relvars whose purpose is to contain *descriptors* regarding the various objects that are of interest to the system itself, such as base relvars, views, indexes, users, integrity constraints, security constraints, and so on. For example:

TABLE

TABNAME	COLCOUNT	ROWCOUNT
DEPT	3	3

COLUMN

TABNAME	COLNAME
DEPT	DEPT#

# Transaction

---

- A **transaction** is a logical unit of work involving several operations that begins by BEGIN TRANSACTION and terminates normally or abnormally.
- **Atomicity** means that transactions are guaranteed either to execute in their entirety or not to execute at all, even if the system fails halfway through the process.
- **Durability** means that once a transaction successfully commits, its updates are guaranteed to be applied to the database, even if the system subsequently fails at any point.
- **Isolation** means that database updates made by a given transaction  $T_1$  are kept hidden from all distinct transactions  $T_2$  until and unless  $T_1$  successfully commits.

# Transaction

---

- **Serializability** means that the interleaved execution of a set of concurrent transactions is guaranteed to produce the same result as executing those same transactions one at a time in some (unspecified) serial order.
- **Commit** (normal termination) is the operation that signals successful end-of-transaction. Any updates made to the database by the transaction in question are now "made permanent" and become visible to other transactions.
- **Rollback** (abnormal termination) is the operation that signals unsuccessful end-of-transaction. Any updates made to the database by the transaction in question are "rolled back" (undone) and are never made visible to other transactions

# Topic 5

---

SQL  
From  
Chapters 5, 6 and 7 of Fundamentals of Database  
Systems, Authors: Elmasri and Navathe  
Publisher: Addison Wesley -Pearson

By: Abdolreza Abhari

CPS510

Ryerson University

# Topics in this Section

---

- SQL Introduction
- Basic SQL data types
- DDL statements
  - » Create, delete, update tables
  - » Specifying constraints, keys, ...
- Example schema definition
- DML statements
  - » Querying the database
  - » Modifying the database
    - Insert, delete, and update
- Views

# Introduction

---

- Developed by IBM in 1970's
- Early prototype called System R
- In 1986, ANSI (American National Standards Institute) published first standard (SQL-86)
- An extended standard SQL in 1989 (SQL-89)
- ANSI/ISO version is SQL-92 Also known as SQL2
- Standard version is SQL:1999
- SQL:2003 and SQL:2006 added XML
- SQL:2008 added object database feature
- All major database vendors support SQL
- Note that:

Both relvar and relation = table in SQL

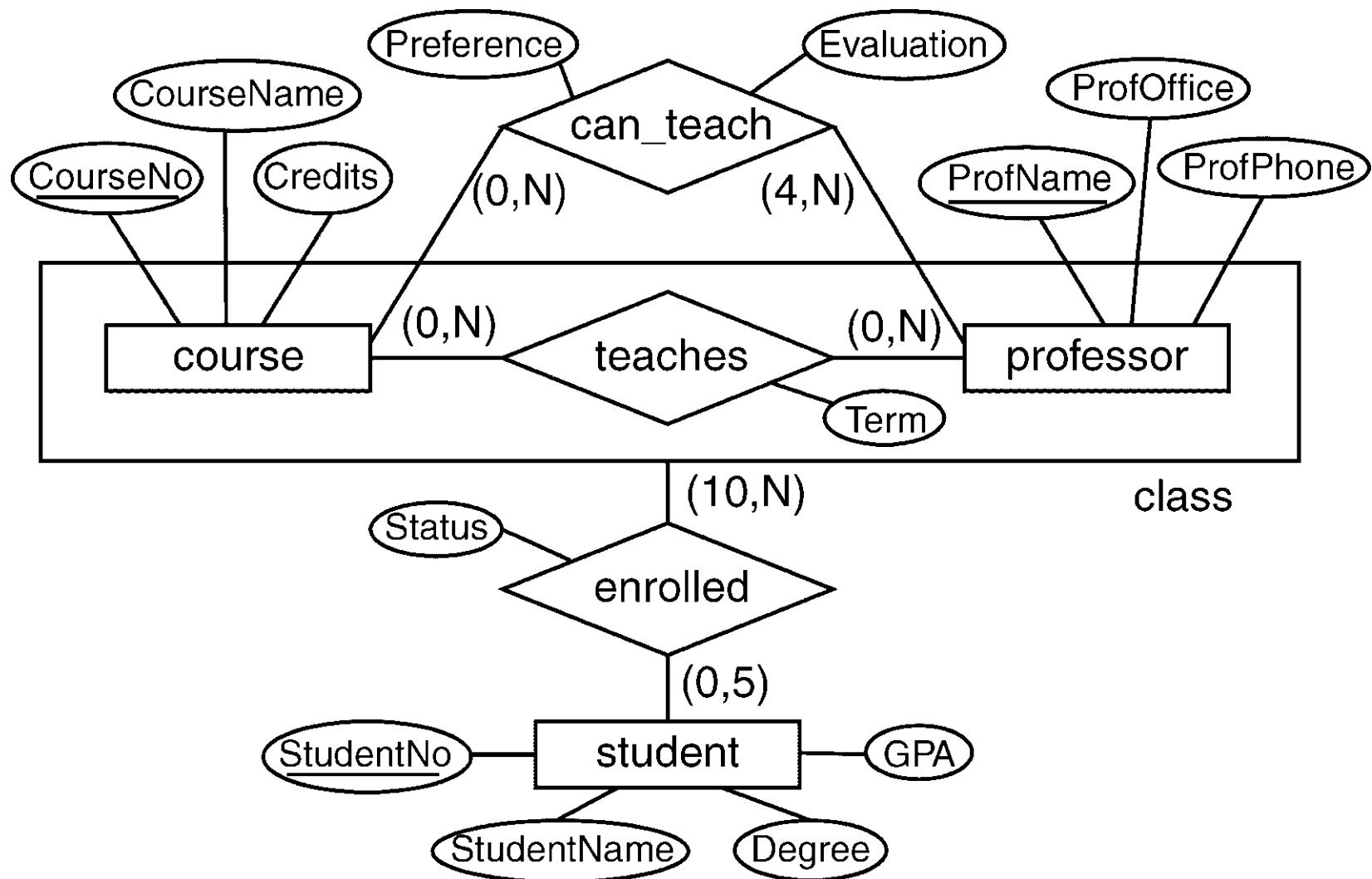
Tuple = row in SQL and Attribute = column in SQL

# Introduction

---

- SQL (Structures Query Language)
  - \* Non-procedural language
  - \* Aims to express most database operations
    - » Queries, updates, creating tables, etc.
  - \* Stand-alone SQL may not be able express everything you want
    - » Embedded SQL gives more flexibility and expressive power
      - Example: You can insert SQL statements to retrieve data from the database into a C or Java program
  - \* We will use SQL commands with the examples of Oracle and db2:
    - » To use Oracle SQL Command Line Processor: first login to Oracle server
    - » Then with using *sqlplus* write SQL commands as:
      - ... “CREATE TABLE .....”

# Example



# Basic Data Types

---

- SQL supports several data types
  - » Numeric
    - Integer and floating-point numbers supported
  - » Character string
    - Fixed- and variable-length supported
  - » Bit string
    - Not supported in DB2
    - We will not discuss bit strings
  - » Date
  - » Time
    - Date and time are combined in Oracle
    - Date and time formats in DB2 are depending on the country code of application. For example: YYYY-MM-DD for date and HH.MM.SS for time

# Basic Data Types (cont'd)

---

- Some exact numeric data types present in SQL-92 and/or Oracle
  - \* Integer values
    - » INTEGER
      - SQL-92 leaves precision to implementation
      - Oracle provides INTEGER(size) 40 digits
      - DB2 uses 4 bytes (-2,147,483,648 to 2,147,483,647)
    - » SMALLINT
      - SQL-92 allows for smaller storage space
        - Again precision is implementation-dependent
      - Oracle use NUMBER type for this
      - DB2 uses 2 bytes for SMALLINT (-32,768 to 32,767)

# Basic Data Types (cont'd)

---

- \* Fractional numbers

- » SQL-92 provides

- NUMERIC: accept default precision and scale
    - NUMERIC(*size*): specify precision but with default scale
    - NUMERIC(*size, d*): Specify both precision and scale

- » Oracle supports them with NUMBER

- » DB2 also supports them

- » DECIMAL

- Same as NUMBER in Oracle
    - Same as NUMERIC in DB2
    - SQL-92 provides 40 digits for this and (*size* and *d* can be specified)

# Basic Data Types (cont'd)

---

- Approximate numeric (floating point) data types
  - \* SQL-92 supports three data types
    - » REAL
      - Single-precision floating point with implementation-dependent precision
    - » DOUBLE PRECISION
      - Implementation-dependent double-precision number
    - » FLOAT( $p$ )
      - Provides binary precision greater than or equal to  $p$
  - \* Oracle provides FLOAT data type
  - \* DB2 provides all of them

# Basic Data Types (cont'd)

---

- Character strings
  - \* Fixed-size string
    - » CHAR(*size*): *size* characters long
      - Pads on the right with blanks for shorter strings
    - » Use CHAR for a single character (equivalent to CHAR(1))
  - \* Variable-length string
    - » VARCHAR(*size*): No blank padding is done
      - In Oracle VARCHAR2(*size*)
      - In DB2 can be from 1 to 32,672 bytes
    - » *size* is the integer value that shows the maximum length

# Basic Data Types (cont'd)

---

- Date representation
  - \* SQL-92 format
    - » DATE
      - Year is exactly 4 digits: 0001 to 9999
      - Month is exactly 2 digits in the range: 01 to 12
      - Day is exactly 2 digits in the range: 01 to 31
        - Month value may restrict this range to 28, 29, or 30
- Time representation
  - \* SQL-92 format
    - » TIME
      - Hour is exactly 2 digits: 00 to 23
      - Minutes is exactly 2 digits in the range: 00 to 59
      - Seconds is again 2 digits (but a fractional value is optional) in the range: 00 to 61.999....

# Basic DDL Statements

---

- Three basic DDL statements
  - \* CREATE TABLE
    - » To create a new table
    - » Can be quite complex
      - Takes various types of constraints into consideration
  - \* ALTER TABLE
    - » To update/modify an existing table
      - Adding/deleting a column
      - Updating an existing column (e.g. changing its data type)
  - \* DROP TABLE
    - » To delete a table
    - » Much simpler than the other two statements

# Creating Tables

---

- Tables can be created using CREATE TABLE statement
  - \* example

```
CREATE TABLE professor(  
    ProfName      VARCHAR2(25) ,  
    ProfOffice    VARCHAR2(10) ,  
    ProfPhone     VARCHAR2(12)  
);
```

- \* Case does not matter
  - » Enclosing table and column names in double quotes makes the names case-sensitive
    - Disastrous for users and developers

# Creating Tables (cont'd)

---

- CREATE TABLE statement allows specification of a variety of constraints on a table
  - » NULL and default values
  - » Candidate keys
  - » Primary keys
  - » Foreign keys
  - » Check conditions
    - e.g. simple range check ( $0 \leq \text{mark} \leq 100$ )
- \* The more constraints you specify
  - » the more work for the DMBS to maintain the data
    - takes more time to update the table
  - » less work for applications to maintain the data

# Creating Tables (cont'd)

---

- NULL values
  - \* NULL values are used to represent information that is out of bounds
  - \* NULL values alleviate the need to use blanks, 0, -1 to indicate
    - not available
    - not applicable
    - unknown
  - \* By default, NULL values are allowed
    - Specify NOT NULL if null values are not allowed for a particular column/attribute
    - NOT NULL is typically used with key attributes

# Creating Tables (cont'd)

---

- \* We can modify our previous example as

```
CREATE TABLE professor(  
    ProfName      VARCHAR2(25) NOT NULL,  
    ProfOffice    VARCHAR2(10),  
    ProfPhone     VARCHAR2(12)  
);
```

- \* **ProfName** is the key to the relation
  - » We *do not* allow entry of tuples with a NULL value for this field
    - Semantically, it means we should know the professor's name before we enter his/her other details
- \* We allow NULL values for the other two columns

# Creating Tables (cont'd)

---

- DEFAULT values
  - \* For attributes, we can also specify a default value
    - Used when no value is given when a tuple is inserted into the table
  - \* We can modify our previous example as

```
CREATE TABLE professor(  
    ProfName      VARCHAR2(25) NOT NULL,  
    ProfOffice    VARCHAR2(10) DEFAULT 'ENG',  
    ProfPhone     VARCHAR2(12) DEFAULT '416-979-5000'  
) ;
```

- \* **ProfOffice** and **ProfPhone** will have the specified default values

# Creating Tables (cont'd)

---

- Candidate keys
  - \* Can be specified using UNIQUE clause
  - \* Example:

```
CREATE TABLE professor(  
    ProfName      VARCHAR2(25) UNIQUE,  
    ProfOffice    VARCHAR2(10) DEFAULT 'ENG',  
    ProfPhone     VARCHAR2(12) DEFAULT '416-979-5000'  
)
```

- \* **ProfName** is a candidate key
  - » Since NOT NULL is not specified, one NULL tuple is allowed
    - Not recommended
    - » Should include NOT NULL (recommended practice)
    - » DB2 requires NOT NULL

# Creating Tables (cont'd)

---

- Rewriting the previous example:

```
CREATE TABLE professor(  
    ProfName      VARCHAR2(25) NOT NULL UNIQUE,  
    ProfOffice    VARCHAR2(10) DEFAULT 'ENG',  
    ProfPhone     VARCHAR2(12) DEFAULT '416-979-5000'  
) ;
```

- \* In SQL-92, we can write
  - » NOT NULL UNIQUE or
  - » UNIQUE NOT NULL
- \* SQL-89 allowed only
  - » NOT NULL UNIQUE

# Creating Tables (cont'd)

---

- We can write the previous statement as:

```
CREATE TABLE professor (
    ProfName      VARCHAR2(25) NOT NULL,
    ProfOffice    VARCHAR2(10) DEFAULT 'ENG',
    ProfPhone     VARCHAR2(12) DEFAULT '416-979-5000',
    UNIQUE (ProfName)
);
```

- \* This form uses **UNIQUE** as a *table constraint* instead of specifying it as a *column constraint*
  - » Useful to specify candidate keys with multiple columns
- \* Specification of candidate keys is useful to enforce uniqueness of the attribute values

# Creating Tables (cont'd)

---

- Primary key
  - \* One of the candidate keys
    - » Attach special significance/characteristics
    - » Only one primary key per table
    - » No NULL values are allowed in primary key column(s)
      - No need for NOT NULL in SQL
      - DB2 requires NOT NULL for primary key
  - \* Specification is similar to candidate key specification
    - » Use PRIMARY KEY instead of UNIQUE
    - » Column and table constraints forms can be used

# Creating Tables (cont'd)

---

## Example 1: Uses column constraint form

```
CREATE TABLE professor (
    ProfName      VARCHAR2(25) PRIMARY KEY,
    ProfOffice    VARCHAR2(10) DEFAULT 'ENG',
    ProfPhone     VARCHAR2(12) DEFAULT '416-979-
5000' );
```

## Example 2: Uses table constraint form

```
CREATE TABLE teaches (
    CourseNo      NUMBER,
    ProfName      VARCHAR2(25),
    Term          CHAR NOT NULL,
    PRIMARY KEY (CourseNo, ProfName) );
```

# Creating Tables (cont'd)

---

- Foreign key
  - \* A combination of columns of one relation that references primary key attributes of a second relation
    - » A tuple in the first table can exist only if there is a tuple in the second table with the corresponding primary key
  - \* Also known as *referential integrity constraint*

```
CREATE TABLE teaches (
    CourseNo      NUMBER REFERENCES course(CourseNo) ,
    ProfName      VARCHAR2(25) REFERENCES professor(ProfName) ,
    Term          CHAR NOT NULL,
    PRIMARY KEY (CourseNo, ProfName)) ;
```

# Creating Tables (cont'd)

---

## Another example

```
CREATE TABLE enrolled (
    StudentNo  NUMBER REFERENCES student(StudentNo) ,
    CourseNo   NUMBER,
    ProfName   VARCHAR2(25),
    Status      CHAR NOT NULL,
    PRIMARY KEY (StudentNo, CourseNo, ProfName),
    FOREIGN KEY (CourseNo, ProfName)
        REFERENCES teaches(CourseNo, ProfName)
) ;
```

- \* No need to establish **CourseNo** and **ProfName** as foreign keys
  - Taken care of by **teaches**

# Creating Tables (cont'd)

---

- Referential integrity actions in SQL2
  - \* On delete or update
    - » SET DEFAULT
      - The attribute value is set to its default value
      - Typically used with delete
    - » SET NULL
      - The attribute value is set to NULL value
      - Typically used with delete
    - » CASCADE
      - Updates are propagated (attribute value is updated)
      - Tuple is deleted (when the other tuple is deleted)
    - » NO ACTION

# Creating Tables (cont'd)

---

- Oracle supports only delete-based integrity actions
  - » Supports only CASCADE option on delete

Example

```
CREATE TABLE can_teach (
    CourseNo      INTEGER REFERENCES course(CourseNo)
                  ON DELETE CASCADE,
    ProfName      VARCHAR2(25) REFERENCES
                  professor(ProfName)
                  ON DELETE CASCADE,
    Preference    NUMBER DEFAULT 0,
    Evaluation    NUMBER(2,1) DEFAULT NULL,
    PRIMARY KEY (CourseNo, ProfName));
```

# Creating Tables (cont'd)

---

- Constraint names
  - \* We can assign names to constraints
  - \* Example

**PRIMARY KEY (CourseNo, ProfName)**

can be written as

**CONSTRAINT teaches\_pk PRIMARY KEY  
(CourseNo, ProfName)**

- \* We can refer to this constraint by its name  
**teaches\_pk**
- \* Useful if we want disable certain constraints at a later time

**CONSTRAINT teaches\_pk DISABLE**

# Creating Tables (cont'd)

---

- Check constraint
  - \* Can be used to ensure that every row in the table satisfies the condition
  - \* Format
    - CHECK condition**
      - » Can use only values in a single row of the table
        - Cannot refer to values in other rows
      - » **condition** can be any valid expression that evaluates to TRUE or FALSE
        - Can contain functions, any columns from this table, and literals
      - » Use column constraint form for single column constraints
      - » Use table constraint form for multi-column constraints

# Creating Tables (cont'd)

---

## Example

- » Suppose we know that course number ranges from 100 to 9999
- » We can create a CHECK constraint to ensure this

```
CREATE TABLE course (
    CourseNo      NUMBER CHECK (CourseNo BETWEEN
                           100 AND 9999),
    CourseName    VARCHAR2(25) NOT NULL,
    Credits       NUMBER NOT NULL,
    PRIMARY KEY (CourseNo)
) ;
```

# Dropping Tables

---

- To delete tables, use **DROP TABLE**

```
DROP TABLE professor;
```

- \* Oracle will not drop a table if deleting the table violates an integrity constraint

- » In our example, **ProfName** column participates in several integrity constraints

- **DROP TABLE** fails

- » To successfully delete tables, start with the table at the bottom of the dependency chain

- For example,

```
DROP TABLE enrolled;
```

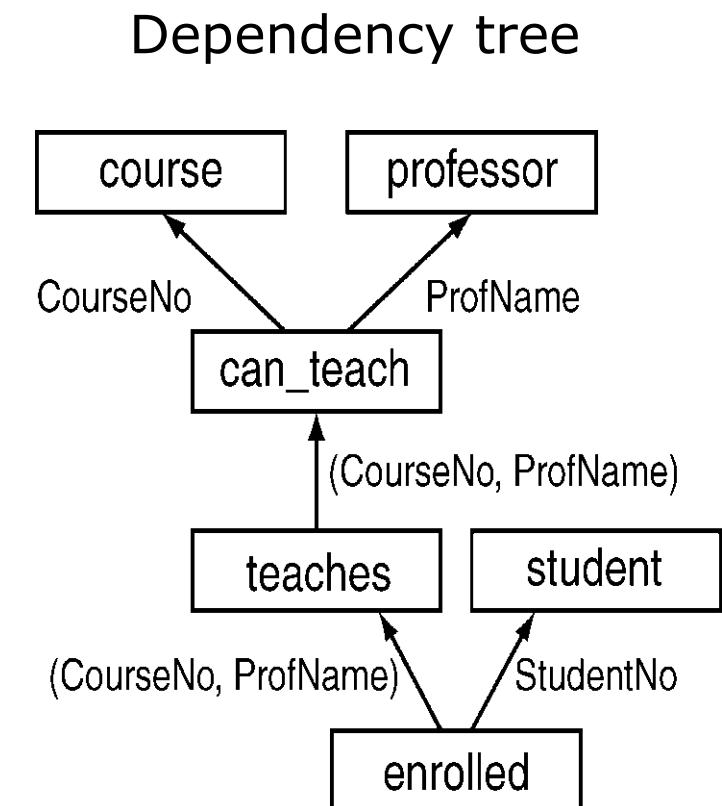
succeeds

# Dropping Tables (cont'd)

- We can use DROP TABLE in the following order for successful deletion of tables

```
DROP TABLE enrolled;  
DROP TABLE teaches;  
DROP TABLE can_teach;  
DROP TABLE student;  
DROP TABLE course;  
DROP TABLE professor
```

- \* This is only a partial ordering
  - » e.g. **student** table can be dropped immediately after dropping **enrolled** table



# Dropping Tables (cont'd)

---

- You can also force deletion of a table by dropping all integrity constraints that refer to the columns in the dropped table
  - \* Use CASCADE CONSTRAINTS option
- Example:

The DROP TABLE statement

```
DROP TABLE professor CASCADE CONSTRAINTS;
```

deletes the **professor** table successfully

» Side effect:

- Drops all referential integrity constraints that refer to the columns in **professor**

# Dropping Tables (cont'd)

---

```
DROP TABLE professor CASCADE CONSTRAINTS;
```

```
CREATE TABLE professor (
    ProfName          VARCHAR2(25) PRIMARY KEY,
    ProfOffice        VARCHAR2(10) DEFAULT
                           'VIC',
    ProfPhone         VARCHAR2(12) DEFAULT
                           '416-999-9999'
) ;
```

# Altering Tables

---

- Two ways of altering a table
  - \* Modify a column's definition
  - \* Add/delete a column
    - » Oracle does not support deletion of an existing column
- Use ALTER TABLE statement
  - \* Can also be used for modifying constraints
- To add a column to represent the rank of a professor, we can use

```
ALTER TABLE professor ADD (
    Rank      CHAR
) ;
```

# Altering Tables (cont'd)

---

- Modifying a column's definition can be done by using MODIFY in place of ADD

## Example

- \* Change **Rank** from one character to two characters

```
ALTER TABLE professor MODIFY (
    Rank      CHAR(2)
);
```

# Altering Tables (cont'd)

---

- Adding columns
  - \* You can add a column at any time if NOT NULL is not specified
  - \* You *cannot* use NOT NULL as in

```
ALTER TABLE professor ADD (
    Rank      CHAR NOT NULL
);
```
- Adding a NOT NULL column is done in three steps:
  - » Add the column *without specifying* NOT NULL
  - » Fill every row in the new column with proper data
  - » Modify the column to be NOT NULL

## Altering Tables (cont'd)

---

Example: Assume **professor** table has two rows

Step 1: Do not specify NOT NULL

```
ALTER TABLE professor MODIFY (
    Rank      CHAR(2)
);
```

Step 2: Fill the new column

```
UPDATE professor SET Rank = 'AP'
WHERE ProfName = 'Post';
```

```
UPDATE professor SET Rank = 'P'
WHERE ProfName = 'Peters';
```

Step 3: Modify to include NOT NULL

```
ALTER TABLE professor MODIFY (
    Rank      CHAR(2) NOT NULL
);
```

# Altering Tables (cont'd)

---

- Modifying a column
  - \* In general, you can
    - Increase a character column width at any time
    - Increase the number of digits in a NUMBER column at any time
    - Increase/decrease the number of decimal places of a NUMBER column at any time
  - \* Only if a column has NULL for *all* rows, you can
    - Change its data type
    - Decrease a character column width
    - Decrease the number of digits in a NUMBER column

# SQL Queries

---

student

<u>StudentNo</u>	StudentName	Degree	GPA
------------------	-------------	--------	-----

professor

<u>ProfName</u>	ProfOffice	ProfPhone
-----------------	------------	-----------

course

<u>CourseNo</u>	CourseName	Credits
-----------------	------------	---------

can\_teach

<u>CourseNo</u>	<u>ProfName</u>	Preference	Evaluation
-----------------	-----------------	------------	------------

teaches

<u>CourseNo</u>	<u>ProfName</u>	Term
-----------------	-----------------	------

enrolled

<u>CourseNo</u>	<u>ProfName</u>	<u>StudentNo</u>	Status
-----------------	-----------------	------------------	--------

# SQL Queries (cont'd)

---

- Uses SELECT statement to query the database
- A simple form of SELECT statement is

**SELECT A<sub>1</sub>, A<sub>2</sub>, . . . , A<sub>n</sub>**

**FROM r<sub>1</sub>, r<sub>2</sub>, . . . , r<sub>m</sub>**

**WHERE cond**

# SQL Queries (cont'd)

---

Q1: List all attributes of all students

```
SELECT *
FROM student;
```

- » We can use \* to list all attributes
- » WHERE clause is optional
  - When not specified, all tuples are selected

Q2: List student names and their GPAs of all students

```
SELECT StudentName, 'overall GPA is: ', GPA
FROM student;
```

- » SELECT statement can be used to output a string
- » When an attribute list is given, only the listed attributes are displayed

# SQL Queries (cont'd)

---

Q3: List all attributes of students with a  $\text{GPA} \geq 10$

```
SELECT *
FROM   student
WHERE  GPA >= 10;
```

- You can use

>	greater than
$\geq$	greater than or equal to
<	less than
$\leq$	less than or equal to
$\neq$	not equal to

You can also use  $\neq$  and  $\neq$  for *not equal to* in ORACLE

# SQL Queries (cont'd)

---

Q4: List all attributes of students with a  $\text{GPA} \geq 10$  (sort the output in descending order by GPA)

```
SELECT *
FROM student
WHERE GPA >= 10
ORDER BY GPA DESC;
```

- \* You can replace DESC by ASC to sort in ascending order
- \* Ascending order is the default
  - » If you do not want to depend on this default
    - specify ASC explicitly

# SQL Queries (cont'd)

---

Q4b: List (only course# and student#) the enrollments in courses (sort the output in descending order by course #, within each course#, sort the student# by ascending order)

```
SELECT CourseNo, StudentNo  
FROM enrolled  
ORDER BY CourseNo DESC, StudentNo ASC;
```

- \* The output is first ordered by **CourseNo** and within a course, ordered by **StudentNo**
- \* No need to include ASC as it is the default
  - » But including ASC improves query readability

# SQL Queries (cont'd)

---

Q5: List the professors teaching a course in the winter term  
(sort the output in ascending order by name)

```
SELECT ProfName AS Winter_Professors
FROM   teaches
WHERE  Term = 'W'
ORDER BY ProfName;
```

- \* The output uses *Winter\_professors* heading instead of *ProfName*
- \* Output contains duplicates if a professor is teaching more than one winter course
- \* Use DISTINCT to eliminate duplicates

```
SELECT DISTINCT ProfName AS Winter_Professors
FROM   teaches
WHERE  Term = 'W'
ORDER BY ProfName;
```

# SQL Queries (cont'd)

---

Q6: List all students (student name and GPA only) of B.C.S. students with a GPA  $\geq 10$

```
SELECT StudentName, GPA  
FROM student  
WHERE GPA >= 10  
      AND Degree = 'B.C.S';
```

- \* Logical operators AND, OR, and NOT can be used to combine several simple conditions
- \* Precedence:

NOT	highest
AND	middle
OR	lowest

» Parentheses can be used to override the default precedence

# SQL Queries (cont'd)

---

Q7: List all students (student name and GPA only) in the B.C.S. program with a  $\text{GPA} \geq 10$  or those in the B.A. program with a  $\text{GPA} \geq 10.5$

```
SELECT StudentName, GPA
FROM student
WHERE (GPA >= 10
      AND Degree = 'B.C.S')
OR
      (GPA >= 10.5
      AND Degree = 'B.A');
```

- \* Works without the parentheses ( ) because AND has a higher precedence than OR

# SQL Queries (cont'd)

---

Q8: List all students (student name and degree only) who are not in the B.C.S. program

```
SELECT StudentName, Degree  
FROM student  
WHERE Degree <> 'B.C.S';
```

- We can also use logical NOT operator

```
SELECT StudentName, Degree  
FROM student  
WHERE NOT (Degree = 'B.C.S');
```

# SQL Queries (cont'd)

---

Q9: List all students (student number and name only) who are enrolled in Prof. Smith's 8203 class

```
SELECT student.StudentNo, StudentName  
FROM enrolled, student  
WHERE ProfName = 'Smith'  
    AND CourseNo = 8203  
    AND enrolled.StudentNo =  
                      student.StudentNo  
ORDER BY StudentNo ASC;
```

» We need to join two tables

→ Last condition specifies the join condition

» We can use the same attribute name in different tables

» Use the table name to identify the attribute as in

**student.StudentNo**

→ Unique attributes do not need the table prefix

# SQL Queries (cont'd)

---

Q9b: List all students (student number and name only) who are enrolled in Prof. Smith's 8203 class

```
SELECT s.StudentNo, StudentName  
FROM enrolled e, student s  
WHERE ProfName = 'Smith'  
      AND CourseNo = 8203  
      AND e.StudentNo = s.StudentNo  
ORDER BY StudentNo ASC;
```

- \* We can use table alias (e for enrolled and s for student)
- \* SQL-92 syntax uses

```
    FROM enrolled AS e, student AS s
```

- \* After aliasing, you have to use alias names (not the original names)

# SQL Queries (cont'd)

---

Q10: List all students (student names only) who are enrolled in Prof. Post's "Introduction to Database Systems" course

```
SELECT StudentName  
FROM   course c, enrolled e, student s  
WHERE  c.CourseName =  
                   'Introduction to Database Systems'  
      AND ProfName = 'Post'  
      AND c.CourseNo = e.CourseNo  
      AND e.StudentNo = s.StudentNo;
```

- » We have to join three tables
- » Last two conditions give the join conditions
  - **course** and **enrolled** on **CourseNo**
  - **enrolled** and **student** on **StudentNo**

# SQL Queries (cont'd)

---

- SQL supports three set operations
  - » Intersection
  - » Difference (EXCEPT or MINUS)
  - » UNION

But they are not widely supported
- For example we can use
  - » EXISTS to implement intersection
  - » NOT EXIST for EXCEPT or MINUS

# SQL Queries (cont'd)

---

Q11: List of students concurrently taking 305 & 403 in Prof. Peters' 403 class

```
SELECT s.StudentNo, s.StudentName
  FROM student s
 WHERE EXISTS
 (SELECT e1.StudentNo
   FROM teaches t1,teaches t2,enrolled e1,enrolled e2
  WHERE e2.ProfName='Peters'
    AND e2.CourseNo = 403
    AND e1.StudentNO = e2.StudentNo
    AND t1.CourseNo= 403
    AND t1.ProfName = 'Peters'
    AND t2.Term = t1.Term
    AND t2.CourseNo = 305
    AND e1.ProfName = t2.ProfName
    AND e1.CourseNo= 305
    AND s.StudentNo = e1.StudentNo) ;
```

# SQL Queries (cont'd)

---

Q12: Give a list of professors who can teach 102 but are not assigned to teach this course

```
SELECT ProfName  
      FROM can_teach ct  
     WHERE CourseNo = 102  
           AND NOT EXISTS  
                  (SELECT *  
                     FROM teaches t  
                    WHERE t.CourseNo = 102  
                      AND ct.ProfName= t.ProfName) ;
```

\* Oracle uses **MINUS**

# SQL Queries (cont'd)

---

Q13: List of professors who are not teaching any course or teaching only summer courses

```
SELECT ProfName
      FROM professor p
 WHERE NOT EXISTS
 (SELECT *
      FROM teaches t
     WHERE p.ProfName= t. ProfName)
UNION
 (SELECT ProfName
      FROM teaches t
     WHERE Term = 'S'
 AND NOT EXISTS
 (SELECT ProfName
      FROM teaches t1
     WHERE Term <> 'S'
 AND t.Profname = t1.ProfName));
```

# SQL Queries (cont'd)

---

Q14: List of courses not offered in the summer term

```
SELECT *
  FROM course
 WHERE CourseNo NOT IN
       (SELECT CourseNo
        FROM teaches
       WHERE Term = 'S');
```

- Can also be written using MINUS operator as

```
(SELECT *
  FROM course)
MINUS
(SELECT c.*
  FROM course c, teaches t
 WHERE c.CourseNo = t.CourseNo
   AND Term = 'S');
```

# SQL Queries (cont'd)

---

Q15: List all courses that are first courses (course title starts with *Introduction* or *principles*)

```
SELECT *  
FROM course  
WHERE CourseName LIKE 'Introduction%'  
      OR  
      CourseName LIKE 'Principles%';
```

- \* Case sensitive matching
- \* % wildcard
  - » Matches 0 or more characters
- \* Underscore (\_)
  - » Matches a single character

# SQL Queries (cont'd)

---

Q16: List of students whose GPA is between 10 and 12

```
SELECT *
FROM student
WHERE GPA BETWEEN 10 AND 12;
```

- It is simply a shorthand for range restriction
  - \* Can be done with relational operators ( $\geq$ ,  $\leq$ ) and the logical AND operator
- We can rewrite the above query without using BETWEEN as

```
SELECT *
FROM student
WHERE GPA >= 10
AND GPA <= 12;
```

# SQL Queries (cont'd)

---

Q17: List all professors who teaches a student who is also taking Prof. Peters' 100

```
SELECT DISTINCT ProfName
FROM enrolled e1
WHERE EXISTS
    (SELECT *
     FROM enrolled e2
     WHERE e2.ProfName = 'Peters'
           AND e2.CourseNo = 100
           AND e2.StudentNo = e1.StudentNo
           AND e1.ProfName <> 'Peters') ;
```

- Format is

**EXISTS (Subquery)**

---

# Aggregate Functions

---

- *Aggregate functions* take a set/multiset of values and return a single value
- SQL provides five aggregate functions
  - » Average: AVG
  - » Minimum: MIN
  - » Maximum: MAX
  - » Total: SUM
  - » Count: COUNT
- ORACLE provides some more:
  - » Variance: VARIANCE
  - » Standard deviation: STDDEV

# Aggregate Functions (cont'd)

---

Q18: Find the average GPA of all students

```
SELECT 'Average GPA is ', AVG(GPA)  
FROM student;
```

- » NULL tuples are excluded from the average computation (as if they didn't exist)

Q19: Find the minimum, maximum, average, variance, and standard deviation of the GPA of all students

```
SELECT MIN(GPA), MAX(GPA), AVG(GPA),  
       VARIANCE(GPA), STDDEV(GPA)  
FROM student;
```

- \* VARIANCE and STDDEV are not part of SQL-92
  - » Available in ORACLE

# Grouping in SELECT Statement

---

- SELECT statement may contain up to six clauses

**SELECT A<sub>1</sub>, A<sub>2</sub>, . . . , A<sub>n</sub>**

**FROM r<sub>1</sub>, r<sub>2</sub>, . . . , r<sub>m</sub>**

**[WHERE cond]**

**[GROUP BY <group attributes>]**

**[HAVING <group conditions>]**

**[ORDER BY <order attributes>]**

- \* The clauses are specified in the given order
- \* Clauses in [ ] are optional

# Grouping in SELECT Statement (cont'd)

---

- Order of Execution
  - \* Chooses rows using the WHERE clause
  - \* Groups these rows based on the GROUP BY clause
  - \* Calculates the results of the aggregate functions for each group
  - \* Uses HAVING clause to choose/eliminate groups
  - \* Uses ORDER BY to order groups
    - » ORDER BY must be either
      - An aggregate function, or
      - A column in the GROUP BY clause

# Grouping in SELECT Statement (cont'd)

---

Q20: List for each course, the number of students registered for the course

```
SELECT CourseNo, COUNT(StudentNo) AS  
                               Number_Enrolled  
FROM    enrolled  
GROUP BY CourseNo;
```

- To eliminate duplicates in the count, use DISTINCT

```
SELECT CourseNo, COUNT(DISTINCT StudentNo) AS  
                               Number_Enrolled  
FROM    enrolled  
GROUP BY CourseNo;
```

# Grouping in SELECT Statement (cont'd)

---

Q21: List the number of non-first term students in each degree program

» First term students have NULL as their GPA

```
SELECT Degree, COUNT(*) AS registered  
FROM student  
WHERE GPA IS NOT NULL /* cannot write GPA<>NULL */  
GROUP BY Degree  
ORDER BY registered;
```

\* We can use **IS NULL** and **IS NOT NULL** to test NULL conditions of a row

## Grouping in SELECT Statement (cont'd)

---

Q22: For each section of a course (identified by course number and professor name), give the enrollment. Output result grouped by course name and professor name

```
SELECT CourseName, ProfName, COUNT (*)
FROM course, enrolled
WHERE course.CourseNo = enrolled.CourseNo
GROUP BY CourseName, ProfName;
```

- \* We can use joins to get results from more than one table
- \* The GROUP BY clause can have more than one attribute

# Grouping in SELECT Statement (cont'd)

---

Q23: Give a list of degree programs that maintain above average GPA

```
SELECT Degree, AVG(GPA)
FROM student
GROUP BY Degree
HAVING AVG(GPA) > (SELECT AVG(GPA)
                      FROM student);
```

- \* The sub-query should produce a single row
  - » The sub-query here

```
SELECT AVG(GPA)
FROM student
```

produces the average GPA for all degree programs

# NULL Values (cont'd)

---

- NULLs in Logic expressions
  - \* Use three values:
    - TRUE, FALSE, and UNKNOWN

NOT operation

X	NOT(X)
TRUE	FALSE
FALSE	TRUE
UNKNOWN	UNKNOWN

# NULL Values (cont'd)

---

AND operation

<b>AND</b>	<b>TRUE</b>	<b>FALSE</b>	<b>UNKNOWN</b>
<b>TRUE</b>	<b>TRUE</b>	<b>FALSE</b>	<b>UNKNOWN</b>
<b>FALSE</b>	<b>FALSE</b>	<b>FALSE</b>	<b>FALSE</b>
<b>UNKNOWN</b>	<b>UNKNOWN</b>	<b>FALSE</b>	<b>UNKNOWN</b>

OR operation

<b>OR</b>	<b>TRUE</b>	<b>FALSE</b>	<b>UNKNOWN</b>
<b>TRUE</b>	<b>TRUE</b>	<b>TRUE</b>	<b>TRUE</b>
<b>FALSE</b>	<b>TRUE</b>	<b>FALSE</b>	<b>UNKNOWN</b>
<b>UNKNOWN</b>	<b>TRUE</b>	<b>UNKNOWN</b>	<b>UNKNOWN</b>

# NULL Values (cont'd)

---

- NULLs and JOINs

- \* Expression

- $x <\text{compare op}> y$

- returns UNKNOWN if

- x is NULL, or
    - y is NULL, or
    - both NULL

- Note:

- » UNKNOWN is a logical value whereas NULL is a data value
  - » Use **x IS [NOT] NULL**
    - rather than **x <> NULL** or **x = NULL**

# NULL Values (cont'd)

---

- NULLs and Aggregate Functions
  - » MIN, MAX, SUM, AVG --- ignore the NULL rows
  - » COUNT --- includes NULL rows in the count
- NULLs and arithmetic operators
  - » Operators like $\ast$ , /, -, + propagate NULL values
- NULLs and Functions
  - » Propagate NULL values

# NULL Values (cont'd)

---

- Design advice
  - » Try avoiding NULLs (confuses people)
    - Make all columns NOT NULL
  - » Use default values
    - Can automatically do this by DEFAULT clause
      - Zero for numeric columns
      - Blank for character columns
  - » Design your own encoding to show missing values
    - Example: Gender codes
      - 0 = unknown
      - 1 = male
      - 2 = female
      - 9 = not applicable (such as corporations, etc.)

# Modifying the Database

---

- SQL provides three basic ways to change the database
  - \* **INSERT:**
    - » Adds a new row to the selected table
      - Direct entry of a record
      - Result of a query
  - \* **DELETE:**
    - » Removes a row or a set of rows from the selected table
  - \* **UPDATE:**
    - » Changes the values of an existing row in the selected table

# Modifying the Database (cont'd)

---

- Insertion of records into tables can be done in two basic ways:
  - » Explicitly specify the record to be inserted
  - » Implicitly specify the record set by specifying a query
    - Result record set of the query is inserted into the table
- \* We can use two formats for the first type of insertion

**INSERT INTO table**

**VALUES (value<sub>1</sub>, value<sub>2</sub>, . . . , value<sub>n</sub>)**

Example

```
INSERT INTO professor VALUES  
( 'Post' , '4528' , '416-979-5000' ) ;
```

- » Useful to insert values for *all attributes* of the table

# Modifying the Database (cont'd)

---

- \* If we want to insert values for only a subset of attributes, we have to specify the attribute names

```
INSERT INTO table(attribute1, ..., attributen)  
VALUES (value1, value2, . . . , valuen)
```

- » Attributes can be listed in any order
  - No need to correspond to the order of the attributes in the table
  - One-to-one correspondence should be there between the attributes and the values specified

## Example

```
INSERT INTO professor (ProfName) VALUES  
( 'Peters' ) ;
```

# Modifying the Database (cont'd)

---

- We can insert the results of a query into a table

```
CREATE TABLE honour_student (
    StudentNo          INTEGER PRIMARY KEY,
    StudentName        VARCHAR2(30) NOT NULL,
    Degree             VARCHAR2(10) ,
    GPA                NUMBER CHECK
                           (GPA BETWEEN 10 and 12)
);
```

```
INSERT INTO honour_student
    SELECT *
    FROM   student
    WHERE  GPA >= 10;
```

# Modifying the Database (cont'd)

---

- DELETION

- \* The basic format is

```
DELETE FROM table  
WHERE <cond>
```

- \* Example 1: Delete 305 course from course table

```
DELETE FROM course  
WHERE CourseNo = 305;
```

» Due to the referential integrity constraints specified for the database, all tuples that refer to 305 in **professor**, **can\_teach**, **teaches** and **enrolled** tables are also deleted

# Modifying the Database (cont'd)

---

- We can also use complex predicates
  - \* Example 2: Delete all students who are enrolled in 102

```
DELETE FROM student
WHERE StudentNo IN
  (SELECT StudentNo
   FROM enrolled
   WHERE CourseNo = 102) ;
```

- » Again leads to cascading deletes due to the referential integrity constraints specified for the database

# Modifying the Database (cont'd)

---

- To modify the database records, use UPDATE
- The format is

**UPDATE table**

**SET attribute<sub>1</sub>= expression<sub>1</sub>, . . . ,**

**attribute<sub>n</sub>= expression<sub>n</sub>**

**WHERE <cond>**

- \* Example 1: Convert GPA from 12-point to 4-point system

**UPDATE student**

**SET GPA = GPA \* 4/12;**

# Modifying the Database (cont'd)

---

- \* Example 2: Change the department code (assuming to be the first two digits of the course no) from 14 to 13

```
UPDATE course
```

```
SET CourseNo = CourseNo - 1000
```

```
WHERE CourseNo BETWEEN 14000 AND 14999;
```

- \* Example 3: Promote to B.C.S all B.Sc students with a GPA at least equal to the average B.C.S student GPA

```
UPDATE student
```

```
SET degree = 'B.C.S'
```

```
WHERE Degree = 'B.Sc'
```

```
AND GPA >= (SELECT AVG(GPA)
```

```
FROM Student
```

```
WHERE Degree = 'B.C.S');
```

# Views

---

- View defines a *virtual table* (as opposed to *base tables*)
  - » For most part, these virtual tables can be used just like the base tables
- Suppose that only a B.C.S student with a GPA at least 10 is qualified to apply for TA
- We can create a **potential\_TA** view as

```
CREATE VIEW potential_TA AS
  (SELECT *
   FROM student
   WHERE Degree = 'B.C.S'
     AND GPA >= 10);
```

## View (cont'd)

---

- You can specify attributes that the created view would have
- Since we know all students in **potential\_TA** view are B.C.S student, we may drop the **Degree** attribute and give appropriate attribute names

```
CREATE VIEW potential_TA1(TA_StudentNo,  
TA_name, GPA) AS  
    (SELECT StudentNo, StudentName, GPA  
     FROM student  
     WHERE Degree = 'B.C.S'  
          AND GPA >= 10);
```

## Views (cont'd)

---

- Views can be deleted by using DROP VIEW  
`DROP VIEW potential_TA;`
- Views can be used just like base tables in queries
- Examples

```
SELECT *
FROM potential_TA;
```

```
SELECT *
FROM    potential_TA1
WHERE   GPA >= 11;
```

## Views (cont'd)

---

- Insertions/updates: For most part, views can be treated like base tables
  - \* Examples: Successful inserts (all update the **student** table by inserting records)

```
INSERT INTO potential_TA VALUES  
    (13334, 'John Majors', 'B.C.S', 11.8);
```

» Insertions into potential\_TA1 view

```
INSERT INTO potential_TA1 VALUES  
    (13243, 'Connie Smith', 11.3);
```

```
INSERT INTO potential_TA1(TA_StudentNo,  
TA_name) VALUES (43243, 'Major Smith');
```

## Views (cont'd)

---

- The following insertion into the view is unsuccessful

```
INSERT INTO potential_TA(TA_StudentNo) VALUES  
(41243)
```

\*

ERROR at line 1:

ORA-01400: cannot insert NULL into

- \* The reason: The base table **student** has NOT NULL for **StudentName** column

## Views (cont'd)

---

- We can create read-only views using

**WITH READ ONLY**

- Example

```
CREATE VIEW potential_TA2 (TA_StudentNo,  
                           TA_name, GPA) AS  
(SELECT StudentNo, StudentName, GPA  
   FROM student  
 WHERE Degree = 'B.C.S'  
   AND GPA >= 10)  
WITH READ ONLY;
```

# Views (cont'd)

---

- Views versus creating tables
  - \* View is not created at definition
    - » It materializes up on first use
  - \* View need not create a physical table
    - » Instead, DBMS can derive the table through some means
      - Implementation-dependent
  - \* View is updated automatically if the base table data is updated
    - » Creating a table and inserting records is not dynamic
- Create table should be used for the base tables
- Views can be used to provide a specific view of the database to a group of users

# Views (cont'd)

---

- Views serve two important purposes
  - \* Performance optimization
    - » Create a view if certain sub-queries or expressions are repeated in existing queries
      - These sub-queries/expressions can be computed efficiently
  - \* Security
    - » Users can be provided only with the data that is needed for their applications
      - This data can be derived from various tables
      - Certain data can be hidden by providing only statistical values (as opposed to individual values)
    - » Users can be provided read-only access

# Summary (cont'd)

---

- DDL statements
  - » Create table statement can be used to specify
    - Default and NULL values
    - Referential integrity constraints
    - Simple domain check constraints
    - Candidate and primary keys
    - Foreign keys
  - » Delete table (Drop table)
  - » Update table (Alter table)
    - Modify a column's definition
      - Certain restrictions apply
    - Add/delete columns
      - Oracle allows only addition of columns

# Summary (cont'd)

---

- DML statements
  - » Querying the database
    - SELECT statement
    - Join conditions will have to be specified if more than one table is involved
    - Provides grouping (GROUP BY), group qualifier (HAVING) and sorting (ORDER BY) facilities
  - » Modifying the database
    - Insert, delete, and update
- Views
  - » Virtual tables
  - » Two main uses
    - Performance optimization
    - Security

Last slide

# Topic 6

---

## Relational Database Design

Chapters 14 and 15 of Fundamentals of Database Systems, Authors: Elmasri and Navathe  
Publisher: Addison Wesley -Pearson

By: Abdolreza Abhari

CPS510, Database Systems  
Ryerson University

# Topics in this Section

---

- Problems with bad database design
- Concept of decomposition
- Functional dependencies
- Normalization theory
  - \* Normal forms (BCNF and 3NF)
  - \* Deriving normal forms
- Multivalued dependnecies
  - \* 4NF

# Introduction

---

- When designing a relational database schema
  - \* Several schemas are possible
    - » Some are better and convenient
    - » Some are bad and not desirable
- Database design process involves selecting an appropriate schema
  - \* Uses the concept of *data dependency*
    - » Simple functional dependency  
 $\text{StudentNo} \rightarrow \text{StudentName}$
    - » More complex multivalued dependency
  - \* Data dependencies obtain semantic information from the application

# Problems with Bad Database Design (cont'd)

---

- Consider the following schema

```
student(StudentNo, StudentName, Degree, GPA,  
          CourseNo, CourseName, Credits)
```

- \* This is a combined version of **student** and **course** relations
- What is wrong with this schema?
  - \* Redundancy
    - » Since a student can take several courses,
      - student information (**StudentName**, **Degree**, and **GPA**) is repeated
      - course information (**CourseName** and **Credits**) is repeated as well

# Problems with Bad Database Design (cont'd)

---

- \* Update anomalies
  - » Direct consequence of redundancy
  - » We could update the GPA in one tuple while leaving as is in other tuples for the same student
- \* Insertion anomalies
  - » We cannot insert a new course unless at least one student is taking it
  - » Could use NULL values but introduce problems of their own
    - In general, use NULL values only when unavoidable
- \* Deletion anomalies
  - » Inverse of the last problem
  - » If we delete all students taking a course, we also lose unintentionally information on the course

# Decomposition

---

- What is the solution?
  - \* Split the relation into two or more relations
    - » This process is called *decomposition*
    - » We can use *join* to get back the original information
    - » The whole theory of relational database design deals with how to do this decomposition so that the resulting schema satisfies desirable properties
  - \* Decomposition can also lead to problems if not done properly

**student (StudentNo, StudentName, Degree, GPA)**  
**course (CourseNo, CourseName, Credits, Degree)**

    - » This leads to loss of information (spurious tuples) when joined

# Decomposition (cont'd)

---

Example  
Original relation

StudentNo	StudentName	Degree	GPA	CourseNo	CourseName	Credits
12345	Cooper	BCS	10.9	100	Intro to CS	3
12345	Cooper	BCS	10.9	102	C Course	3
12355	Smith	BCS	11.9	100	Intro to CS	3
12355	Smith	BCS	11.9	102	C Course	3
12355	Smith	BCS	11.9	203	Comp. Org.	3
12377	Miles	BCS	10.1	100	Intro to CS	3

# Decomposition (cont'd)

---

Decomposed relations

student

StudentNo	StudentName	Degree	GPA
12345	Cooper	BCS	10.9
12355	Smith	BCS	11.9
12377	Miles	BCS	10.1

course

CourseNo	CourseName	Credits	Degree
100	Intro to CS	3	BCS
102	C Course	3	BCS
203	Comp. Org.	3	BCS

# Decomposition (cont'd)

---

Join of student and course

StudentNo	StudentName	Degree	GPA	CourseNo	CourseName	Credits
12345	Cooper	BCS	10.9	100	Intro to CS	3
12345	Cooper	BCS	10.9	102	C Course	3
12345	Cooper	BCS	10.9	203	Comp. Org.	3
12355	Smith	BCS	11.9	100	Intro to CS	3
12355	Smith	BCS	11.9	102	C Course	3
12355	Smith	BCS	11.9	203	Comp. Org.	3
12377	Miles	BCS	10.1	100	Intro to CS	3
12377	Miles	BCS	10.1	102	C Course	3
12377	Miles	BCS	10.1	203	Comp. Org.	3

It is wrong see the next slide

# Decomposition (cont'd)

---

- \* The join of student and course generates 3 spurious tuples
- \* A good decomposition should be *lossless* or non-additive join property
  - Ensures that no spurious tuples are generated when a natural join is applied to the relations in the decomposition

# Functional Dependencies

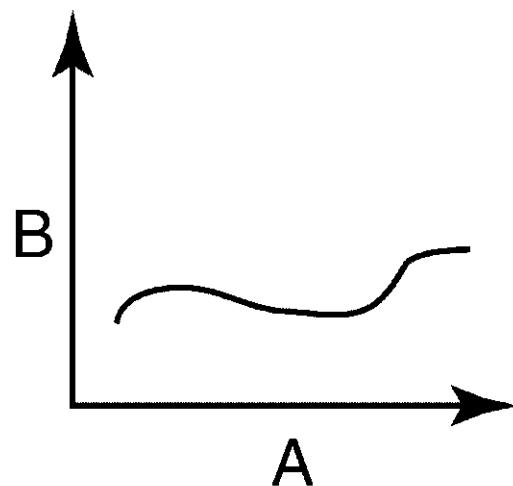
---

- Functional dependency expresses semantic information on two set of attributes
- Notation:  $\mathbf{A}_1 \mathbf{A}_2 \dots \mathbf{A}_n \rightarrow \mathbf{B}_1 \mathbf{B}_2 \dots \mathbf{B}_m$ 
  - » Read:  $\mathbf{A}_1 \mathbf{A}_2 \dots \mathbf{A}_n$  are functionally determines  $\mathbf{B}_1 \mathbf{B}_2 \dots \mathbf{B}_m$
  - » Read:  $\mathbf{B}_1 \mathbf{B}_2 \dots \mathbf{B}_m$  are functionally dependent on  $\mathbf{A}_1 \mathbf{A}_2 \dots \mathbf{A}_n$
- Examples:
  - $\text{CourseNo} \rightarrow \text{Credits}$
  - $\text{StudentNo} \rightarrow \text{Degree}$
  - $\text{CourseNo} \rightarrow \text{CourseName}$If  $\text{CourseName}$  is unique for the application, then
  - $\text{CourseName} \rightarrow \text{CourseNo}$

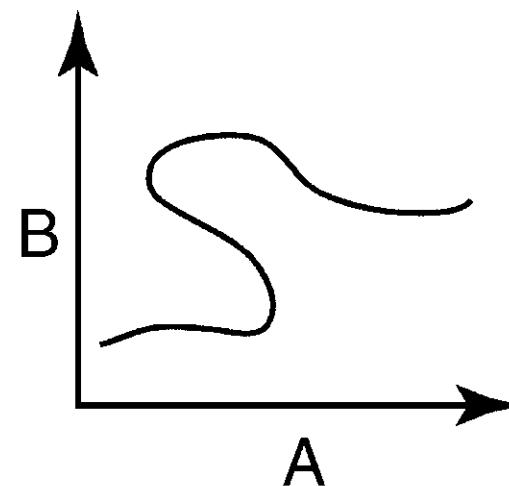
# Functional Dependencies (cont'd)

Functional dependency

$X \rightarrow Y$  holds on  $R$  if in any legal relation  $r \in R$ , for all pairs of tuples  $t_1$  and  $t_2$  in  $r$  such that  $t_1.X = t_2.X$ , it is also the case that  $t_1.Y = t_2.Y$



$$A \rightarrow B$$



$$A \not\rightarrow B$$

# Functional Dependencies (cont'd)

- The following FDs *appear* to hold:
  - »  $W \rightarrow Y$
  - »  $WX \rightarrow Z$
- We cannot look at an instance of a relation and deduce what FDs hold
  - » Example: If the relation is empty, all FDs appear to hold
- We can look at an instance of a relation and say what FDs do not hold

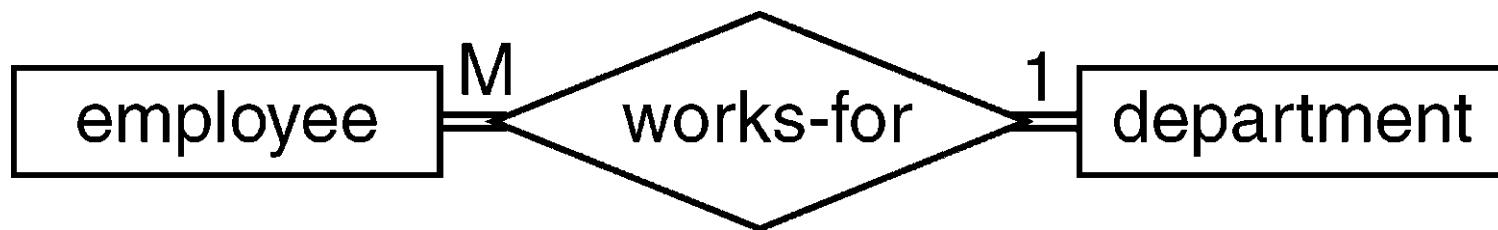
W	X	Y	Z
a1	b1	c1	d1
a1	b2	c1	d2
a2	b2	c2	d2
a2	b3	c2	d3
a3	b3	c2	d4

- FDs are assertions about the real world/application
  - They cannot be proved
  - We can enforce them in the database by proper design

# Functional Dependencies (cont'd)

---

- FDs can capture relationship types one-to-many relationship
  - One FD captures this Example
    - A department can have several employees. Each employee may work in only one department



FDs that express this relationship:  
 $\text{emp\#} \rightarrow \text{dept\#}$

# Functional Dependencies (cont'd)

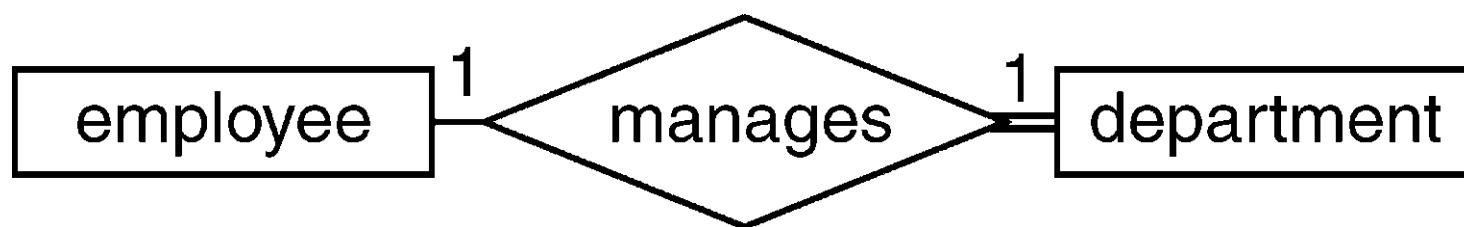
---

one-to-one relationship

Two FDs capture this Example

A manager can manage only one department

Each department can have only one manager



FDs that express this in **manages** relationship:

$\text{emp\#} \rightarrow \text{dept\#}$

$\text{dept\#} \rightarrow \text{emp\#}$

# Functional Dependencies (cont'd)

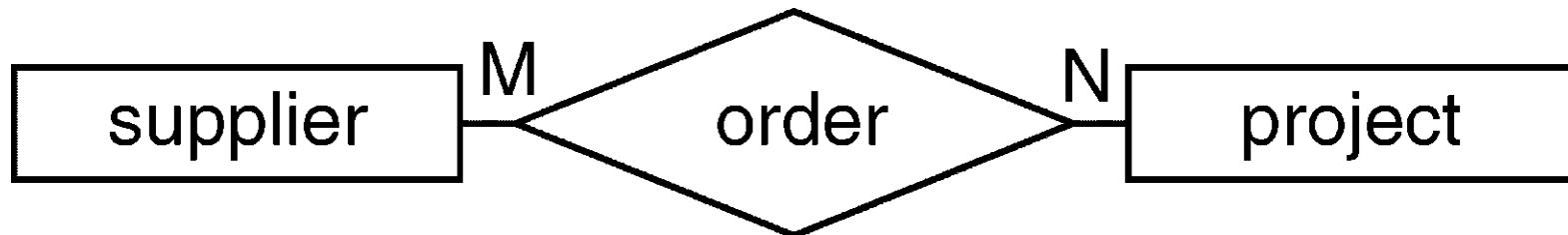
---

many-to-many relationship

No FDs to capture this Example

A supplier can supply parts to several projects

A project can receive parts from several suppliers



It is clear that no functional dependencies hold for this relationship

# Normalization Theory

---

- Normalization theory is built around the concept of normal forms
  - » A relation is said to be in a normal if it satisfies a certain set of specified constraints
- Several normal forms have been defined
  - \* 1NF, 2NF, 3NF, BCNF
    - » Based on functional dependency concept
  - \* 4NF
    - » Based on multivalued dependencies
  - \* 5NF, ...
- In practical terms, focus is on BCNF/3NF designs

# Normalization Theory (cont'd)

universe of relations  
(normalize and unnormalized)

1NF relations (normalized relations)

2NF relations

3NF relations

BCNF relations

4NF relations

PJNF (5NF)  
relations

# Normalization Theory (cont'd)

---

A relation is said to be in 1NF if it satisfies the constraint that it contains only atomic (simple, indivisible) values

This relation is not in 1NF

ProfName	CourseNo
Smith	{100,102, 203}
Post	{100,102, 305}
Graham	{102, 203}

This relation is in 1NF

ProfName	CourseNo
Smith	100
Smith	102
Smith	203
Post	100
Post	102
Post	305
Graham	102
Graham	203

# Normalization Theory (cont'd)

---

## Second Normal Form (2NF)

- Some definitions
  - \* Full Functional Dependency
    - » A functional dependency  $X \rightarrow Y$  is a full dependency if removal of any attribute from X means that the dependency does not hold
  - \* Partial dependency
    - » A functional dependency  $X \rightarrow Y$  is a partial dependency if there is some attribute  $A \in X$  that can be removed from X and dependency will still hold

# Normalization Theory (cont'd)

---

- Examples
  - \* For the **enrolled** relation,  
 $\{\text{CourseNo}, \text{ProfName}, \text{StudentNo}\} \rightarrow \text{Status}$   
is a *full dependency* as removing any of the three attributes will lead to the dependency not holding
  - \* For the **student** relation  
 $\{\text{StudentNo}, \text{Studentname}\} \rightarrow \text{Degree}$   
is a *partial dependency* because  
 $\{\text{StudentNo}\} \rightarrow \text{Degree}$   
holds

# Normalization Theory (cont'd)

---

- 2NF definition
  - » A relation schema R is in 2NF if it is in 1NF and every non-key attribute is fully functionally dependent on the primary key of R
  - \* The relation schema  
**student (StudentNo, StudentName, Degree, GPA)**  
is in 2NF
  - \* The relation schema  
**enrolled (StudentNo, ProfName, CourseNo,  
Degree, Status)**  
is not in 2NF because of  
**{StudentNo} → Degree**

# Normal Forms

---

- Two normal forms
  - \* BCNF (Boyce-Codd Normal Form) (will be discussed later)
  - \* 3NF (Third Normal Form)
    - BCNF is stronger than 3NF
- Our goal is to design a relational schema that is
  - \* In BCNF
  - \* Lossless-join type
  - \* Dependency preserving (discussed later)
- If not possible, we will settle for 3NF (instead of BCNF)

# 3NF

---

- A relation schema R with FDs F is in 3NF if it is in 2nd normal form and every nonkey attributes is nontransitively dependent on the primary key.
- Note that this definition assuming only one candidate key which further become the primary key. Later we will see the situations with we have more than one candidate key.
- For now we assume the FDs and primary and candidate keys are all identified based on appropriate information about the meaning of attributes and their relationships. Later on we will use formulas to identify the keys a set of FDs.

# 3NF

---

- For example:

**PropertyOwner (Propertyno, Paddress, Rent,  
Ownerno, Oname)**

With following FDs

$\{ \text{Propertyno} \} \rightarrow \text{Paddress, Rent, Ownerno, Oname}$

$\{ \text{Ownerno} \} \rightarrow \text{Oname}$

is not 3NF because all non-primary-key attributes are functionally dependent on primary key, with the exception of Oname, which is transitively dependent on Ownerno (see the second FD)

---

## 3NF

---

- Note that if Oname was part of primary key, Propertyowner was 3NF although it had transitive FD. But here since Oname is a nonkey attribute Propertyowner is not 3NF.
- To transform Propertyowner relation into 3NF we remove the transitive dependency by creating two new relations as the follows:

**Propertyforrent (Propertyno, Paddress, Rent,  
Ownerno)**

**Owner (Ownerno, Oname)**

## General Definition of 2NF and 3NF

---

The more general definitions for 2NF and 3NF that consider more than one candidate key for a relation are defined as follows:

- 2NF: A relation that is in 1NF and every non-candidate key attribute is fully functionally dependent on any candidate key.
- 3NF: A relation that is in 1NF and 2NF and in which no non-candidate-key attribute is transitively dependent on any candidate key

# BCNF

---

- *Boyce/Codd normal form* (BCNF) considers the general situations in which relation has super key or candidate keys.
- BCNF: A relation is in BCNF if and only if every nontrivial, left irreducible FD has a super key ( for simplicity we consider candidate key) as its determinant.
- Superkeys are the set of all combination of attributes in a table until removing one of the attributes makes it to be non-key. Candidate keys are the super keys with the minimized number of attributes.
- Note that *determinant* refer to the left side of a FD and in *trivial* FD, left side is a superset of the right side.

# BCNF

---

- Note that the difference between BCNF and 3NF is a functional dependency  $A \rightarrow B$  is allowed in 3NF if B is primary key attribute and A is not a candidate key, whereas BCNF insists that A must be candidate key (which is also a super key).
- Therefore, BCNF is stronger than 3NF because every relation in BCNF is also in 3NF. However a relation in 3NF is not necessarily in BCNF.

# BCNF

---

- Example of BCNF relations:

**Client**(Client#, cname, address, cphone#)

**Client#** → cname, address, cphone#

**Cphone#** → cname, address, client#

Or

**Storelocations**(Storeid, Storelocations)

**Storeid** → Storelocations

# More on Functional Dependencies

---

- Functional dependencies
  - \* Not sufficient to consider the given set
  - \* We need to consider all functional dependencies
- Example
  - \*  $X \rightarrow Y$  and  $Y \rightarrow Z$  logically implies  $X \rightarrow Z$
- Closure
  - \* Let  $F$  be the set of functional dependencies
  - \*  $F^+ = \text{closure of } F$ 
    - »  $F^+$  is the set of all FDs
      - original + logically implied by  $F$

# More on Functional Dependencies (cont'd)

---

- Two methods to compute the closure
- Method 1
  - \* Uses inference rules
  - \* Three basic inference rules
    - (1) Reflexivity rule

If  $X$  is a set of attributes and  $Y \subseteq X$ ,  
then  $X \rightarrow Y$  holds
    - (2) Augmentation rule

If  $X \rightarrow Y$  holds and  $W$  is a set of attributes,  
then  $WX \rightarrow WY$  holds
    - (3) Transitivity rule

$X \rightarrow Y, Y \rightarrow Z \Rightarrow X \rightarrow Z$

# More on Functional Dependencies (cont'd)

---

- The three rules are sound and complete
  - \* Sound
    - » They generate only the *correct* FDs
  - \* Complete
    - » They generate *all* FDs
- These three rules are known as *Armstrong's axioms*
  - » More appropriately as *Armstrong's inference rules*
- However, it is more convenient to consider other rules to simplify computing  $F^+$ 
  - \* Define three more rules

# More on Functional Dependencies (cont'd)

---

- Three additional rules
  - (4) Union rule

If  $X \rightarrow Y$  and  $X \rightarrow Z$  hold  
then  $X \rightarrow YZ$  holds
  - (5) Decomposition rule

If  $X \rightarrow YZ$  holds  
then  $X \rightarrow Y$  and  $X \rightarrow Z$  hold
  - (6) Pseudo-Transitivity rule

If  $X \rightarrow Y$  and  $WY \rightarrow Z$  hold  
then  $WX \rightarrow Z$  holds
- These three rules can be proved by using Armstrong's inference rules

# More on Functional Dependencies (cont'd)

---

- Method 2: Closure X under F

Algorithm to compute  $X^+$

```
x+ := X;  
repeat  
    old x+ := x+;  
    for each FD Y → Z in F do  
        if Y ⊆ x+  
            then x+ := x+ ∪ Z;  
until (old x+ = x+);
```

# More on Functional Dependencies (cont'd)

---

## Example

Given:

$$F = \{name \rightarrow street, city, province \\ name, date\_donated \rightarrow amount\_donated\}$$

Find:

All attributes that are functionally dependent on name,  
date\_donated

Answer:

$$(name, date\_donated)^+ = \{name, date\_donated, \\ amount\_donated, street, city, province\}$$

# Desirable Properties of Decomposition

---

- Two important properties
  - \* Lossless-join decomposition
    - » Required for semantic *correctness*
    - » When we join the decomposed relations, we should get back exact original relation contents
      - No spurious tuples
  - \* Dependency preservation
    - » Required for *efficiency*
    - » If checking a dependency requires joining two or more relations, it is *very inefficient* to enforce this FD .
      - Requires join when inserting a tuple

# Finding Keys By FDs

---

- We can define keys in terms of FDs
- If R is a relation schema with
  - » Attributes A<sub>1</sub>, A<sub>2</sub>, ..., A<sub>n</sub>
  - » Functional dependencies F
    - X is a subset of A<sub>1</sub>, A<sub>2</sub>, ..., A<sub>n</sub>
- \* X is a key if
  - »  $X \rightarrow A_1, A_2, \dots, A_n$  is in  $F^+$ , **and**
  - » For no proper subset Y ⊂ X  
 $Y \rightarrow A_1, A_2, \dots, A_n$  is in  $F^+$
- \* For a given relation schema, there may be several keys
  - » There are called *candidate keys*
  - » *Primary key*: Candidate key selected by the designer as the key

# Finding Keys By FDs (cont'd)

---

## Example

- \* Consider the relation schema

$R(C, S, P)$

with FDs

$\{CS \rightarrow P, P \rightarrow C\}$

- \* This schema has two candidate keys:

$\{CS\}$  and  $\{SP\}$

- \*  $\{CS\}$  is a candidate key because

$\{CS\}^+ \rightarrow \{C, S, P\}$

- \*  $\{SP\}$  is a candidate key because

$\{SP\}^+ \rightarrow \{S, P, C\}$

# Deriving 3NF: Bernstein's Algorithm

---

- Derives 3NF schema that is lossless and dependency preserving
- Outline
  - \* There are 4 steps
    - Step 1:
      - Find out facts about the real world
        - Difficult step but must be done in the design of a database
        - Probably takes more time than all the other steps put together
      - Result is a list of attributes and FDs

# Bernstein's Algorithm (cont'd)

---

Step 2:

- Reduce the list of functional dependencies
  - There is a straightforward polynomial algorithm (discussed later)
  - This step can be done manually for a small list
  - Algorithm can be programmed for a large list
- Result is a minimal list of FDs

Step 3:

- Find the keys
  - Difficult step (details given later)
- Result is a list of candidate keys

# Bernstein's Algorithm (cont'd)

---

Step 4:

- Derive the final schema
  - Combine FDs with the same left hand side
  - Make a new table for each FD
  - Add a key relation if no relation contains a key
  - Eliminate relations that contained in other relations
- Result is 3NF schema that is
  - Lossless
  - Dependency preserving
- Adding a key relation in this step is necessary to guarantee lossless-join type decomposition

# Bernstein's Algorithm (cont'd)

---

## Details of Step 2

Objective: Minimizing the list of FDs

- Consists of three sub-steps

### Sub-step 1

- » Rewrite FDs so that right hand side each FD is exactly one attribute

- Left hand side may have a set of attributes

FD  $X \rightarrow Y, Z$  is written as

$$X \rightarrow Y$$

$$X \rightarrow Z$$

- » Essentially applying the decomposition rule

# Bernstein's Algorithm (cont'd)

---

## Sub-step 2

- » Get rid of redundant FDs
- » Procedure
  - For each FD do the following
    - Consider FD  $X \rightarrow Y$
    - Take this FD from the list of all FDs
    - Find  $X^+$  in the reduced list
    - If  $X^+$  contains  $Y$ , then  $X \rightarrow Y$  is redundant
  - Note:
    - Here Sub-step 1 is necessary to eliminate some subset of right hand attributes

# Bernstein's Algorithm (cont'd)

---

Example for sub-step 2

`pharmacy_account# → patient_id`

`patient_id → doctor_id`

`pharmacy_account#, drug → doctor_id`

» Suppose we want to see if the last FD is redundant

» Reduced list of FDs

`pharmacy_account# → patient_id`

`patient_id → doctor_id`

» Compute  $\{pharmacy\_account\#, drug\}^+$

$$\begin{aligned} &= \{pharmacy\_account\#, drug, \\ &\quad patient\_id, doctor\_id\} \end{aligned}$$

» Since this closure includes `doctor_id`, the last FD

`pharmacy_account#, drug → doctor_id`

is redundant

# Bernstein's Algorithm (cont'd)

---

## Sub-step 3

- » Minimize left hand side
- » This is a tedious but straightforward process
- » For each FD, apply the following procedure:
  - Eliminate an attribute A on the LHS of one of the FDs
  - Look at the remainder Q of attributes on the LHS for that FD
  - Find  $Q^+$  in the *original set* of FDs
  - If  $Q^+$  contains the RHS of the FD in question  
then the attribute A is redundant

# Bernstein's Algorithm (cont'd)

---

Example for sub-step 3

$\text{last\_name}, \text{SIN} \rightarrow \text{first\_name}$   
 $\text{SIN} \rightarrow \text{last\_name}$

- » Suppose we want to see if **last\_name** in the first FD is redundant

$A = \text{last\_name}$

$Q = \text{SIN}$

$\text{SIN}^+ = \{\text{SIN}, \text{last\_name}, \text{first\_name}\}$

- » Since this closure contains the RHS (i.e., **first\_name**), **last\_name** is redundant
- » Minimal dependencies are:

$\text{SIN} \rightarrow \text{first\_name}$

$\text{SIN} \rightarrow \text{last\_name}$

# Bernstein's Algorithm (cont'd)

---

## Details of Step 3

### Objective: Finding keys

- \* To determine whether or not a given set of attributes  $X$  is a key
  - » Find  $X^+$  and see that it contains all attributes of the relation
    - All attributes of the relation are dependent on  $X$
  - » If  $X$  has more than one attribute, make sure that no proper subset of  $X$  has this property
    - This can be done by eliminating one attribute at a time
  - » Tedium process
    - For  $k$  attributes, we need to check  $2^k - 1$  possibilities

# Bernstein's Algorithm (cont'd)

---

- \* Two observations to simplify the amount of work
  - » If an attribute is never on the left hand side of a dependency, it is not in any key, unless it is also never on the right hand side
  - » If an attribute is never on the right hand side of any FD, it is in every key

Example for Step 3

**SIN → first\_name**

**SIN → last\_name**

**SIN → date\_of\_birth**

**last\_name, first\_name → SIN**

- \* Attributes on RHS but not on LHS

**date\_of\_birth** (it will not in any key)

# Bernstein's Algorithm (cont'd)

---

- \* Attributes not on RHS
  - » None for this example (in every key)
- \* Check the subsets of attributes:
  - SIN, last\_name, first\_name**
  - » Only four possibilities

<b>SIN</b>	is a key
<b>last_name</b>	not a key
<b>first_name</b>	not a key
<b>first_name, last_name</b>	is a key
- \* We have only two candidate keys:  
 $\{\text{SIN}\}$  and  $\{\text{first\_name}, \text{last\_name}\}$

# Bernstein's Algorithm (cont'd)

---

## Details of Step 4

**Objective:** Derive the final schema

- » Combine FDs with the same left hand side
  - $X \rightarrow Y$  and  $X \rightarrow Z$  are combined into  $X \rightarrow Y Z$
  - Applying the union rule
- » Make one relation for each FD, containing all attributes on both sides of the FD
  - Provides dependency preservation property
- » If no candidate key to the original schema is in any of these new relations, add a relation with all attributes of *some* key
  - Provides lossless-join property
- » If some relation contains all the attributes of some other relation, eliminate the smaller relation

# Bernstein's Algorithm (cont'd)

---

## Example 1

**R(account#, patient\_id, doctor\_id, drug, qty)**

FDs: {  $\text{account\#} \rightarrow \text{patient\_id}$ ,  
 $\text{patient\_id} \rightarrow \text{doctor\_id}$ ,  
 $\text{account\#, drug} \rightarrow \text{doctor\_id}$ ,  
 $\text{patient\_id, drug} \rightarrow \text{qty}$  }

## Step 2

- » FD are in the desired form
- » The FD  $\text{account\#, drug} \rightarrow \text{doctor\_id}$  is redundant
- » Left hand side of the remaining FDs are all minimal

# Bernstein's Algorithm (cont'd)

---

## Step 3

- » Only on key:

**{account#, drug}**

## Step 4:

- » Results in the following relation schema:

**R1 (account#, patient\_id)**

**R2 (patient\_id, doctor\_id)**

**R3 (patient\_id, drug, qty)**

**R4 (account#, drug)**

- » Note:

- The last relation is added as the other three (R1, R2 and R3) do not contain the key

# Bernstein's Algorithm (cont'd)

---

## Example 2

**R(SIN, last\_name, first\_name, DOB)**

FDs: {**last\_name, SIN → first\_name,**  
**SIN → last\_name,**  
**SIN → DOB,**  
**last\_name, first\_name → SIN}**}

### Step 2

- » FD are in the desired form
- » No redundant FD in this example
- » The FD **last\_name, SIN → first\_name** can be replaced with **SIN → first\_name**

# Bernstein's Algorithm (cont'd)

---

## Step 3

- » Two candidate keys:

{SIN}

{last\_name, first\_name}

## Step 4:

- » We get two relations:

R1 (SIN, last\_name, first\_name, DOB)

R2 (last\_name, first\_name, SIN)

- » Since attributes of R2 are a subset of R1, we can eliminate R2

- » Final schema:

R1 (SIN, last\_name, first\_name, DOB)

# Deriving BCNF

---

## Example

- \* Consider the relation schema

$R(C, S, P)$

with FDs

$\{CS \rightarrow P, P \rightarrow C\}$

- \* This schema has two candidate keys:

$\{CS\}$  and  $\{SP\}$

- \* This schema is *not* in BCNF because

- The dependency  $P \rightarrow C$  holds on  $R$
  - But  $P$  is not a key nor does it contain a key

## Deriving BCNF (cont'd)

---

Algorithm: Lossless decomposition into BCNF

1. set  $D := \{R\}$
2. while there is a schema in  $D$  that is not in BCNF do
  - choose a schema  $Q$  that is not in BCNF;
  - Find a dependency  $X \rightarrow Y$  in  $Q$  that violates BCNF;
  - Replace  $Q$  in  $D$  by two schemas  $(Q - Y)$  and  $(X \cup Y)$

## Deriving BCNF (cont'd)

---

Example: Consider the previous example

- \* Here  $R = Q = (C, S, P)$ 
  - » The FD that violates BCNF is  $P \rightarrow C$
- \* We replace R by R1 and R2
  - $R1 = (P, C)$
  - $R2 = (S, P)$
- \* This decomposition is in BCNF because
  - » For R1:  $F1 = \{P \rightarrow C\}$  and we know P is a key for R1
    - Thus, R1 is in BCNF
  - » For R2:  $F2 = \{ \}$ 
    - $\{S, P\}$  is the key to R2
    - Thus R2 is in BCNF too

# Deriving BCNF (cont'd)

---

- A Problem: The resulting BCNF schema is lossless-join type
  - » But not guaranteed to be dependency preserving
    - Proof: The last example BCNF schema is not dependency preserving
- Our goal is to design a relational schema that is
  - In BCNF
  - Lossless-join type
  - Dependency preserving
- If not possible, we settle for 3NF that guarantees a decomposition that is both
  - Lossless-join type
  - Dependency preserving

## 3NF versus BCNF

---

One way to distinguish between 3NF and BCNF relations is by considering *Zaniolo's* definition of 3NF, which is not based on transitivity. It is :

- A relation schema R with FDs F is in 3NF if *one of the following holds* for all FDs that hold on R:
  - \*  $X \rightarrow Y$  is a trivial FD
    - » That is,  $Y \subseteq X$
  - \*  $X$  is a superkey for R
    - » Super key is candidate key + some other attributes
  - \*  $Y$  is contained in a candidate key
- First two are the same conditions as in BCNF
- 3NF adds the last condition
  - » Thus, every schema that is in BCNF is also in 3NF
  - » But, every schema that is in 3NF is not necessarily in BCNF

# 4NF

---

## Fourth normal form (4NF)

- Deals with a new type of dependency called *multi-valued dependency*. Consider the following example:

CourseNo	ProfName	Text
203	Smith	Assembly lang. text
	Post	Comp. Org. text
305	Post	DBMS text
	Graham	Oracle text
	Smith	

## 4NF (cont'd)

---

Normalized table instance of CPT table is

CourseNo	ProfName	Text
203	Smith	Assembly lang. text
203	Smtih	Comp. Org. text
203	Post	Assembly lang. text
203	Post	Comp. Org. text
305	Post	DBMS text
305	Post	Oracle text
305	Graham	DBMS text
305	Graham	Oracle text
305	Smith	DBMS text
305	Smith	Oracle text

## 4NF (cont'd)

---

- Note that there is no functional dependencies for the data in CPT schema
- CPT schema is in BCNF
  - \* All three attributes form the key to CPT relation
- Redundancy in the data
  - \* Causes update anomalies
  - \* Inserting a new professor for 95305 requires two tuples
- Intuitively, we know it is better if we split CPT into two relations by knowing that course was the key in the original relation

**R1 (C, P)**

**R2 (C, T)**

## 4NF (cont'd)

---

CP instance:

CourseNo	ProfName
203	Smith
203	Post
305	Post
305	Graham
305	Smith

CT instance:

CourseNo	Text
203	Assembly lang. text
203	Comp. Org. text
305	DBMS text
305	Oracle text

## 4NF (cont'd)

---

- If we join CP and CT, we get back the original relation instance
- Problem: The theory we discussed so far will not lead us to this kind of design
- 4NF allows us to come up with this design
  - \* Uses multivalued dependencies
- In our CPT example, there are no FDs
  - \* But there are two multivalued dependencies (MVDs)

**CourseNo**  $\text{--->>}$  **ProfName**

**CourseNo**  $\text{--->>}$  **Text**

## 4NF (cont'd)

---

- We use the symbol

$X \text{ --->> } Y$

for multivalued dependency

» Read: X multidetermines y, or  
Y is multidependent on X

- MVDs always occur in pairs

MVD  $X \text{ --->> } Y$

also implies

MVD  $X \text{ --->> } (R - (XY))$

- Every FD is an MVD

\* Converse is not true

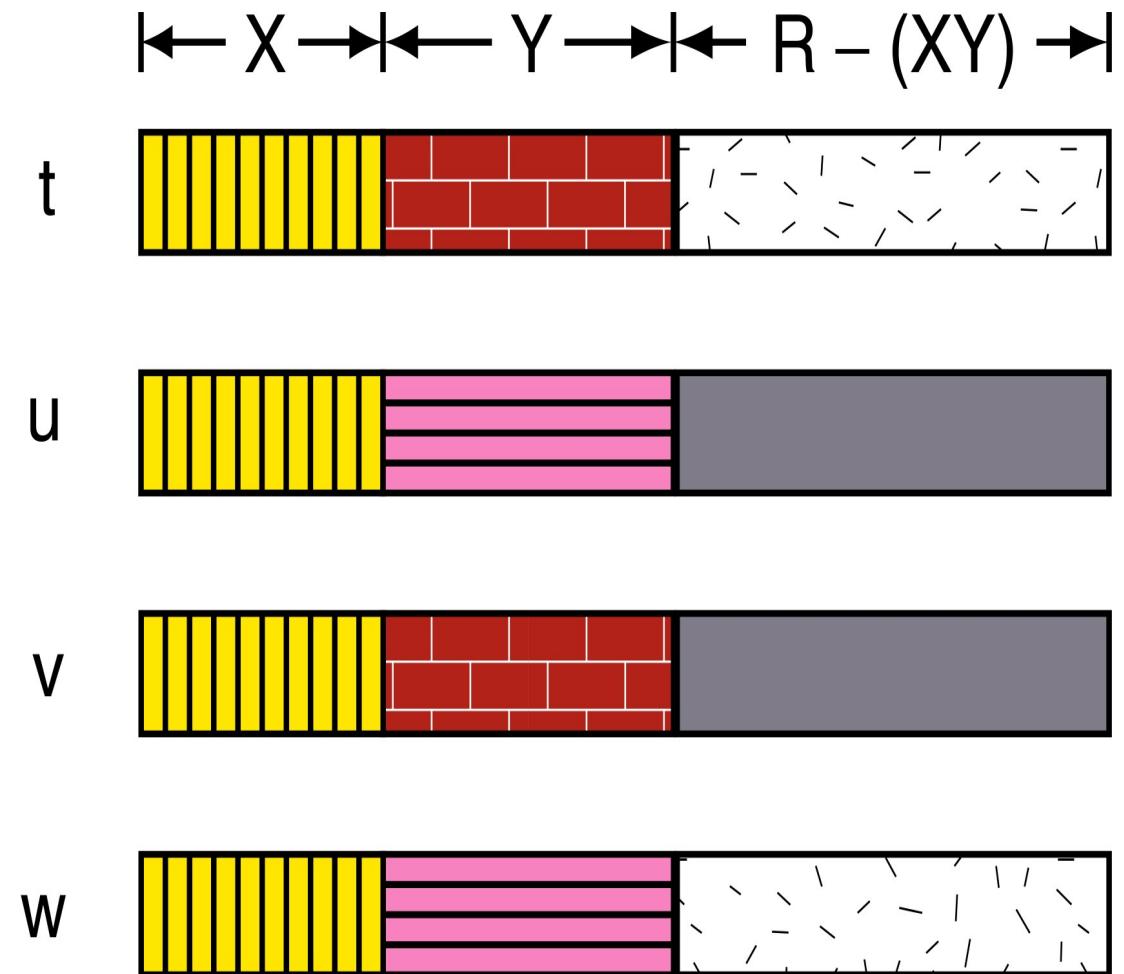
## 4NF (cont'd)

MVD definition:

The MVD  $X \rightarrow\!\!\!> Y$  holds on R if for each pair of tuples t and u of R that agree on X, we can find some other tuple v that agrees:

- With both t and u on X
- With t on Y, and
- With u on  $R - (XY)$

This definition implies (interchanging t and u) the existence of another tuple w that agrees with u on Y and t on  $R - (XY)$



## 4NF (cont'd)

---

- 4NF definition

- \* A relation R is in 4NF whenever

$X \dashrightarrow Y$

holds on R, one of the following holds:

- $X \dashrightarrow Y$  is a trivial MVD, or
  - X is a superkey for R

- \* Non-trivial MVD

- » A MVD  $X \dashrightarrow Y$  is non-trivial if

- None of the attributes in Y are in X
    - Not all attributes of R are in X and Y

- \* 4NF is essentially BCNF with not more than one

---

~~MVDs (note more than one multi-valued attribute in the table)~~

## 4NF (cont'd)

---

Algorithm: Lossless decomposition into 4NF

1. set  $D := \{R\}$
2. while there is a schema in  $D$  that is not in 4NF do
  - choose a schema  $Q$  that is not in 4NF;
  - Find a nontrivial MVD  $X \rightarrow\!\!\!> Y$  in  $Q$  that violates 4NF;
  - Replace  $Q$  in  $D$  by two schemas  $(Q-Y)$  and  $(X \cup Y)$

## 4NF (cont'd)

---

- Example
  - \* Consider the CPT relations

$R(C, P, T)$

with MVDs:  $C \dashrightarrow\!\!\!> P$ ,  $C \dashrightarrow\!\!\!> T$

» This is not in 4NF because for the nontrivial MVDs

$C \dashrightarrow\!\!\!> P$  and  $C \dashrightarrow\!\!\!> T$

$C$  is not a superkey of  $R$

- \* Using the algorithm, we derive the following 4NF schema: (note there is only one MVD in  $R1$  and  $R2$ )

$R1(C, P)$

$R2(C, T)$

# Additional Normal Forms (5NF)

---

- Additional normal forms also exist. For example 5NF
  - » 5NF or PJNF (project-join normal form): deals with the relations that require to decompose into more than two relations to have lossless-join property

It means if R is in 4NF (with no or only one MVDs) then if we can decompose it into the tables being lossless join (no extra or reduced records) and without join dependency (being dependent on the order of join) it will be 5NF. For example the 4NF **R1 (C, P)** and **R2 (C, T)** are also 5 NF tables.
  - » See Sections 13.3 of the text (Date's book) for details.

# Testing Decomposition

---

- BCNF/3NF decomposition
  - » Eliminates most of the anomalies
  - » Allows for efficient checking of desired FDs
  - » Sometimes BCNF decomposition is not dependency preserving
- For checking 5NF decomposition two things should be checked
  - » If decomposition has join dependency
  - » If decomposition is loss less decomposition

Next slides explain how to test lossless join decomposition and dependency preserving

# Testing Lossless-Join Decomposition

---

## Lossless Join Decomposition Test

- Assume
  - » Relational schema R is decomposed into k relations  $\{R_1, R_2, \dots, R_k\}$
  - » R has n attributes  $\{A_1, A_2, \dots, A_n\}$

(1) Construct a table with

- n columns
  - column j corresponds to attribute  $A_j$
- k rows
  - row i corresponds to  $R_i$

(2) Initialize the table

- Place  $a_j$  in row i and column j if  $A_j$  is in  $R_i$
- otherwise, place  $b_{ij}$

# Testing Lossless-Join Decomposition (cont'd)

---

(3) Repeat until no changes can be made to the table

» For each FD  $X \rightarrow Y$

- Look for all rows that agree in all the columns for the attributes of  $X$
- If we find two such rows, equate the symbols of those rows for the attributes of  $Y$
- Special rules for equating:
  - If one of them is  $a_j$ , make the other  $a_j$
  - If they are  $b_{ij}$  and  $b_{lj}$ , make them both  $b_{ij}$  or  $b_{lj}$  arbitrarily
- If we discover some row with all  $a$  entries ( $a_1, a_2, \dots, a_n$ )
  - The decomposition is lossless
- Else
  - The decomposition is lossy

# Testing Lossless-Join Decomposition (cont'd)

---

Example:

- \* Consider schema R decomposed into R1 and R2

$R(S, A, I, P)$

$R1(S, A)$

$R2(S, I, P)$

with FDs  $\{S \rightarrow A, SI \rightarrow P\}$

- \* The initial table is

S	A	I	P
a <sub>1</sub>	a <sub>2</sub>	b <sub>13</sub>	b <sub>14</sub>
a <sub>1</sub>	b <sub>22</sub>	a <sub>3</sub>	a <sub>4</sub>

# Testing Lossless-Join Decomposition

---

- \* Since  $S \rightarrow A$  holds and the two rows agree on  $S$ , we can equate their symbols for  $A$

→  $b_{22}$  becomes  $a_2$

- \* The resulting table is

S	A	I	P
$a_1$	$a_2$	$b_{13}$	$b_{14}$
$a_1$	$a_2$	$a_3$	$a_4$

- Since the second row has all  $a$  entries, the decomposition is lossless

# Testing Dependency Preserving

---

- Some times it is easy and if we follow the algorithm of BCNF we can easily notice when a FD is missing in the decomposition
- But in some cases we have to find the FDs in the decomposed tables for example:
  - \* Consider schema R decomposed into R1 and R2

**R (A, B, C, D, E)**

**R1 (A, B, C, D)**

**R2 (D, E)**

With the original FDs {**A →BC**, **C →DE**, **D→E**}

# Testing Dependency Preserving

---

- The question is if it is dependency preserving or not. First we have to find FDs of R1 and R2 from FDs  $\{A \rightarrow BC, C \rightarrow DE, D \rightarrow E\}$  by looking at attributes:

R1 (A, B, C, D)	FDs $\{A \rightarrow BC, C \rightarrow D\}$
R2 (D, E)	FDs $\{D \rightarrow E\}$
- If the union FDs of R1 and R2 equals to FDs of R it is dependency preserving
- FDs of  $(R1 \cup R2)$  is  $\{A \rightarrow BC, C \rightarrow D, D \rightarrow E\}$  so it looks like it is missing  $C \rightarrow E$ . But if we look at closure of C it includes E and  $C \rightarrow E$  and then  $C \rightarrow DE$  can be achieved for FDs of  $(R1 \cup R2)$ . Thus it is dependency preserving

# Summary

---

- Problems with bad database design
  - \* Redundancy
    - » Anomalies (insertion, deletion, and update)
- Introducing Normal Forms
- Functional dependency
  - \* Brings semantic information from the application into database design phase
  - \* Some times ER diagram can not capture all the rules or it is not easy to find entity and their relations in the applications

# Summary

---

- Normalization theory
  - \* Several normal forms have been defined
    - » Most popular normal forms are
      - BCNF
        - Desired and stricter than 3NF
        - But BCNF does not always result in a dependency preserving decomposition
      - 3NF
        - Always gives a lossless-join and dependency preserving decomposition
  - Multivalued dependencies, 4NF and 5NF
  - Testing of Lossless-join and dependency preserving characteristics in decomposition

# Topic 7

---

## Relational Model Query Languages

CPS510  
Database Systems  
Edited by Abdolreza Abhari  
From Chapter 8 of Almasri's book  
Department of Computer Science  
Ryerson University

# Topics in this Section

---

- Formal query languages
  - \* Relational algebra
  - \* Relational calculus
- Commercial query languages
  - \* SQL (already discussed)
  - \* QBE

# Formal Query Languages

---

- Three principal approaches to design languages for expressing queries about relations
  - \* Two broad classes
    - » Algebraic languages
      - Queries are formed by applying specialized operators to relations
    - » Predicate calculus languages
      - Queries describe a desired set of tuples by specifying a predicate the tuples must satisfy
      - Two types of languages
        - Primitive objects are tuples
        - Primitive objects are elements of the domain of an attribute

# Formal Query Languages (cont'd)

---

- Three formal query languages
  - \* Relational algebra
  - \* Tuple relational calculus
  - \* Domain relational calculus
- Two commercial query languages
  - \* SQL (Structured Query Language)
    - » Uses features of relational algebra and relational calculus
  - \* QBE (Query-By-Example)
    - » Domain relational calculus based language

# Relational Algebra

---

- Six fundamental operations
  - Selection
  - Projection
  - Cartesian product
  - Union
  - Difference
  - Renaming
- » All operations are on tables and produce a *single* result table
- » Selection, projection, and renaming are *unary* operations
  - Need only one table
- » The other three are *binary* operators
  - Each operation requires two tables

# Relational Algebra (cont'd)

---

- Selection operation
  - \* Notation:  $\sigma_P(R)$
  - \* Selects tuples in relation R that satisfy predicate P
  - \* Conditions are expressed in the form
    - <attribute> <operator> <value>
    - <attribute> <operator> <attribute>
  - » The operators for
    - Ordered domains:  $\{=, <, \leq, >, \geq, \neq\}$
    - Unordered domains:  $\{=, \neq\}$
  - \* Selection is also called *restriction*
  - \* result relation degree = original relation degree

# Relational Algebra (cont'd)

---

- PART

part#	part_name	weight
1	Tower case	2.5
2	Sony display	4.5
3	Mother board	0.6
4	Yamaha speakers	2
5	Power supply	1

- $\sigma_{\text{weight} > 2}$  (PART)

part#	part_name	weight
1	Tower case	2.5
2	Sony display	4.5

# Relational Algebra (cont'd)

---

- \* Logical operators
  - AND, OR, and NOTcan be used to specify compound conditions
- \* Selection is commutative

$$\sigma_{<\text{cond1}>}(\sigma_{<\text{cond2}>}(\text{R})) = \sigma_{<\text{cond2}>}(\sigma_{<\text{cond1}>}(\text{R}))$$

- \* A cascade of selects can be combined into a single select with a conjunctive (AND) condition

$$\begin{aligned}\sigma_{<\text{cond1}>}(\sigma_{<\text{cond2}>}(\dots (\sigma_{<\text{condn}>}(\text{R})) \dots)) = \\ \sigma_{<\text{cond1}>} \text{ AND } <\text{cond2}> \text{ AND } \dots \text{ AND } <\text{condn}> (\text{R})\end{aligned}$$

# Relational Algebra (cont'd)

---

- Projection
  - \* Notation:  $\Pi_A(R)$
  - \* Selects only the columns of relation R specified by attribute list A
    - Other columns are discarded
  - \* The resulting relation has the attributes specified in A and in the *same order* as they appear in A
    - » degree of the result relation = # of attributes in A
  - \* Implicitly removes duplicate tuples from the result relation
    - » Duplicates might exist due to the deletion of certain columns from the input relation

# Relational Algebra (cont'd)

PART

part#	part_name	weight
1	Tower case	2.5
2	Sony display	4.5
3	Mother board	0.6
4	Yamaha speakers	2
5	Power supply	1

$\Pi_{\text{part}\#}, \text{part\_name} (\text{PART})$

part#	part_name
1	Tower case
2	Sony display
3	Mother board
4	Yamaha speakers
5	Power supply

- Projection is not commutative
- Cascading project operations are valid only under certain conditions

- $\Pi_{\text{list-1}} (\Pi_{\text{list-2}} (R)) = \Pi_{\text{list-1}} (R)$
- provided  $\text{list-2}$  contains attributes of  $\text{list-1}$

# Relational Algebra (cont'd)

---

- Cartesian product (or simply *product*)
  - \* Notation:  $R \times S$
  - \* Provides the basic capability to combine data from several relations
  - \* If  $r \in R$  and  $s \in S$ ,  $t = r s$  belongs to  $R \times S$ 
    - » The attributes of  $R$  precede the attributes of  $S$
  - \* Attributes with identical names in different relations are identified by prefixing the relation name
  - \* Result relation degree = degree of  $R$  + degree of  $S$
  - \* Number of tuples in the result relation =  
number of tuples in  $R$  \* number of tuples in  $S$

# Relational Algebra (cont'd)

---

student

student#	student_name
12345	John
12346	Margaret

enrolled

student#	course#
12345	95100
12345	95305
12346	95305

student  $\times$  enrolled

student.student#	student_name	enrolled.student#	course#
12345	John	12345	95100
12345	John	12345	95305
12345	John	12346	95305
12346	Margaret	12345	95100
12346	Margaret	12345	95305
12346	Margaret	12346	95305

# Relational Algebra (cont'd)

---

- Union
  - \* Notation:  $R \cup S$ 
    - » Similar to set union operation
  - \* Result relation will have tuples that are in R or S or both
    - » Duplicates are eliminated in the result table
  - \* We can apply this operation only if R and S are *union compatible*
  - \* Two relations R and S are union compatible if
    - » R and S are of the same degree (i.e., same number of attributes)
    - » The domains of *i*th attribute of R and the *i*th attribute of S are the same

# Relational Algebra (cont'd)

---

- Difference
  - \* Notation:  $R - S$ 
    - » Similar to set difference operation
  - \* Result relation will have tuples that are in R but not in S
  - \* We can apply this operation only if R and S are *union compatible*
  - \* Note:  $R - S$  is not the same as  $S - R$ 
    - » Difference is not commutative
  - \* In both union and difference operations, the result will have the same attribute names as in the source relations
    - » If the two source relations have different attribute names, use the rename operation to give same attribute names

# Relational Algebra (cont'd)

- Example

student	
student#	student_name
12345	John
12346	Margaret

TA	
student#	student_name
12345	John
32456	Janet
23456	Jim

Student  $\cup$  TA

student#	student_name
12345	John
12346	Margaret
32456	Janet
23456	Jim

TA - student

student#	student_name
32456	Janet
23456	Jim

# Relational Algebra (cont'd)

---

- Rename
  - \* Notation:  $\rho_{S(A_1, A_2, \dots, A_n)}(R)$
  - \* Renames a degree  $n$  relation  $R$  to relation  $S$  with attribute names changed to  $A_1, A_2, \dots, A_n$ 
    - » If you don't want to rename the attributes, use
$$\rho_S(R)$$
to rename the relation only
      - Renames the relation to  $S$  but uses the same attributes names as in  $R$
    - » Example: If the attribute name for student name in TA relation is **TA\_name**, we could use renaming to union the two relations
$$\text{student} \cup \rho_{\text{TA}(\text{student}\#, \text{student\_name})}(TA)$$

# Additional RA Operations

---

- The previous six operations are sufficient to express relational algebra queries
  - » These operations are *not sufficient* to do any computation on relations
    - They are quite limited in their expressive power
    - However, they are enough to express the computations we really want on relations
      - These operations form the basis for the SQL
- These are sometimes too cumbersome to use
  - » Cartesian product is not particularly useful by itself
- Some additional operations are defined to simplify
  - » These additional operations can be expressed using only the basic six operations discussed

# Additional RA Operations (cont'd)

---

- Intersection
  - \* Notation:  $R \cap S$ 
    - » Similar to set intersection
  - \* Result relation contains tuples that are in both  $R$  and  $S$ 
    - » Duplicates are eliminated in the result table
  - \* Like union and difference,  $R$  and  $S$  must be *union compatible*
  - \* Intersection can be expressed as
$$R \cap S = R - (R - S)$$

# Additional RA Operations (cont'd)

---

- Intersection

student

<b>student#</b>	<b>student_name</b>
12345	John
12346	Margaret

TA

<b>student#</b>	<b>student_name</b>
12345	John
32456	Janet
23456	Jim

TA  $\cap$  student

<b>student#</b>	<b>student_name</b>
12345	John

# Additional RA Operations (cont'd)

---

- Division
  - \* Notation:  $R \div S$
  - \* Let  $R$  and  $S$  be relations with degree  $r$  and  $s$ 
    - » The following conditions must be met:
      - $r > s$
      - $S \neq \text{null}$
    - \* The result is the set of  $(r-s)$  degree tuples  $t$  such that for all  $s$ -tuples  $u$  in  $S$ , the tuple  $t u$  is in  $R$

# Additional RA Operations (cont'd)

- Division

A	B	C	D
1	2	3	4
1	2	5	6
2	3	5	6
5	4	3	4
5	4	5	6
1	2	4	5

C	D
3	4
5	6

$R \div S$

A	B
1	2
5	4

## Additional RA Operations (cont'd)

---

- We can express the division operation using the basic relational algebra operations

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

$r-s$  = attributes that are in R but not in S

» Attributes A and B represent  $r-s$  in our last example

- We will see an example query that uses the division operation
- Commercial query language SQL does not support this operation

# Additional RA Operations (cont'd)

---

- Natural Join
  - \* Notation:  $R \triangleright\triangleleft S$ 
    - » We use  $\triangleright\triangleleft$  to represent
    - » Natural join applies a selection operation and a projection operation on the output of  $R \times S$
    - » Retrieves only those tuples that have matching values for the common attributes  $A_1, A_2, \dots, A_c$  in  $R$  and  $S$

$R \triangleright\triangleleft S =$

$$\Pi_{r \cup s} (\sigma_{r.A_1=s.A_1 \text{ AND } r.A_2=s.A_2 \text{ AND } \dots \text{ r}.A_c=s.A_c} (R \times S))$$

- \*  $r \cup s$  represents the union of attributes in  $R$  and  $S$

# Additional RA Operations (cont'd)

- Natural Join Example

Student

student#	student_name
12345	John
12346	Margaret

enrolled

student#	course#
12345	95100
12345	95305
12346	95305

Student  $\bowtie$  enrolled

student#	student_name	course#
12345	John	95100
12345	John	95305
12346	Margaret	95305

# Additional RA Operations (cont'd)

---

- Cartesian product “blindly” joins two tables
  - » Very expensive to compute
  - » Often times not really needed
- Natural join takes the semantic information into account
  - » In the previous example it makes sense to join only those tuples that have a matching **StudentNo** attribute values
  - » Some tuples in the Cartesian product result do not make sense
- Natural join reverses the decomposition phase of the relational database design process

# Additional RA Operations (cont'd)

---

- Theta-Join
  - \* Natural join uses one specific condition (no options)
    - » Useful and required in most cases
  - \* Sometimes it is required to join on some other condition than the one used by the natural join
  - \* Theta-join allows for specification of the condition on which the join is to be performed
    - » Theta represents the arbitrary join condition
  - \* Most common use: joining two relations on attributes with different names
    - » Example: **employee#** and **manager#**
    - » To apply natural join, the attributes should have the same name

# Additional RA Operations (cont'd)

---

- \* Notation:  $R \triangleright\triangleleft_{\theta} S$ 
  - »  $\theta$  represents the join condition
  - » Retrieves only those tuples that satisfy the join condition  $\theta$

$$R \triangleright\triangleleft_{\theta} S = \Pi_{r,s}(\sigma_{\theta}(R \times S))$$

- \* The degree of the result relation is
$$\text{degree}(R) + \text{degree}(S)$$
- \* Note: Attributes, even with same name, are not filtered
  - » Because of the general  $\theta$  condition
    - Values need not be equal as in the natural join case
- \* Equi-join
  - » It is theta-join with equality condition
  - » Not equal to natural join as attributes are repeated in equi-join

# Additional RA Operations (cont'd)

employee

employee#	emp_name	phone
123456	John	520-4321
123457	Barbara	520-4433
123458	Jennifer	520-7654

manager

manager#	dept_name
123456	Toys
123458	Electronics

employee  $\bowtie$  manager

employee#	emp_name	phone	manager#	dept_name
123456	John	520-4321	123456	Toys
123458	Jennifer	520-7654	123458	Electronics

duplication

# Additional RA Operations (cont'd)

---

- Assignment
  - \* Notation:  $R \leftarrow S$
  - \* Creates relation R with tuples from relation S
    - » We use this to simplify writing relation algebra expressions
    - » No need to write one long, complex expression
      - We can divide the expression into more convenient and meaningful pieces
  - \* Example:  $R \div S = \Pi_{r-s}(R) - \Pi_{r-s}((\Pi_{r-s}(R) \times S) - \Pi_{r-s,s}(R))$   
can be written as
    - temp1  $\leftarrow \Pi_{r-s}(R)$
    - temp2  $\leftarrow \Pi_{r-s}((\text{temp1} \times S) - \Pi_{r-s,s}(R))$
    - result  $\leftarrow \text{temp1} - \text{temp2}$

# Additional RA Operations (cont'd)

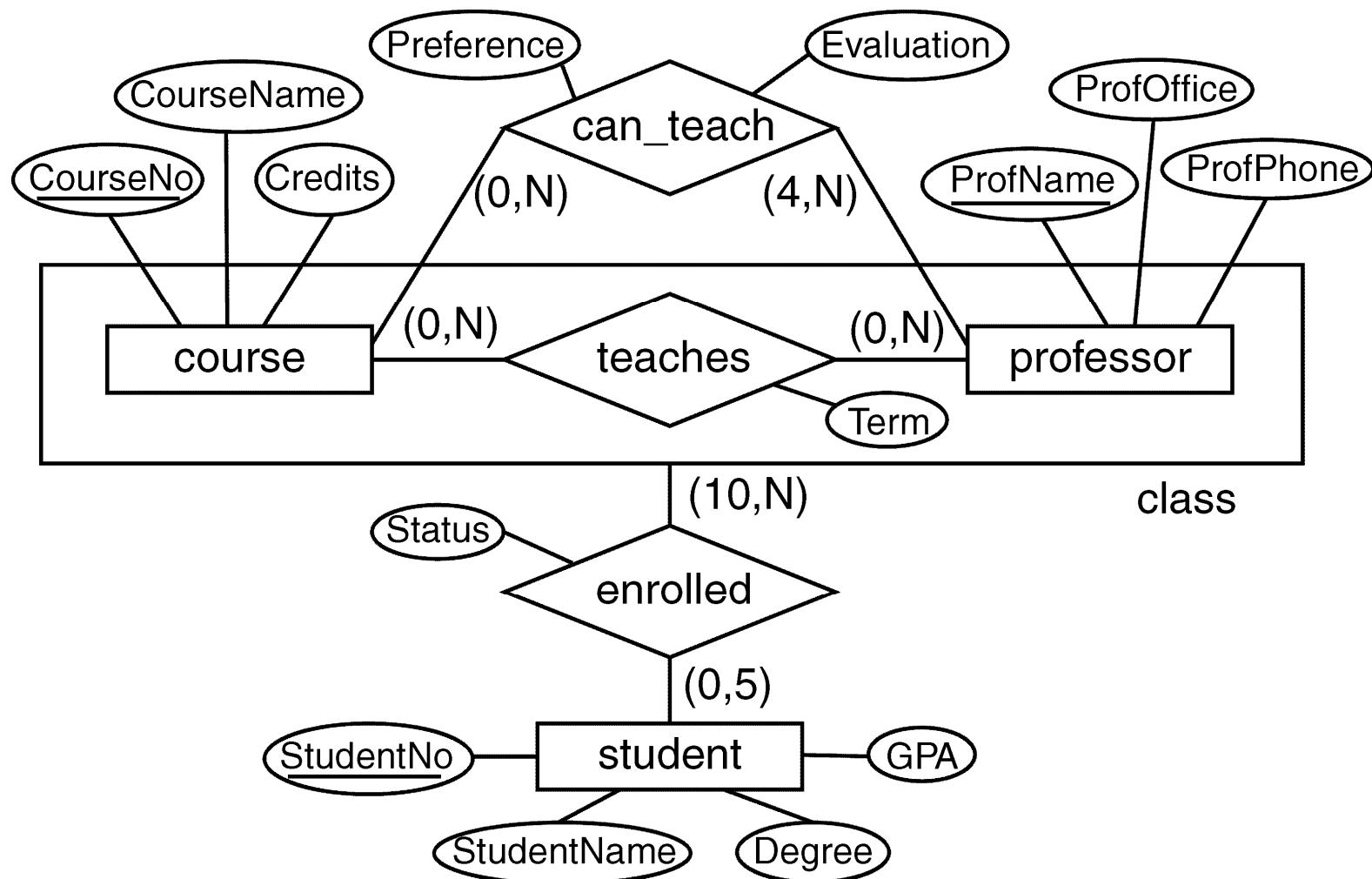
---

- Aggregate functions
  - \* No standard notation
    - We use the notation given in the text
  - \* Notation:  $\langle \text{grouping attributes} \rangle \text{ } F \langle \text{function list} \rangle (R)$   
 $\langle \text{function list} \rangle$  is a list of ( $\langle \text{function} \rangle \langle \text{attribute} \rangle$ ) pairs
  - \* Example

$R(\text{Degree}, \text{NoRegistered}, \text{AvgGPA}) \leftarrow$

**Degree  $F$  COUNT StudentNo, AVERAGE GPA (student)**

# Example RA Queries



# Example RA Queries (cont'd)

---

student	<u>StudentNo</u>	StudentName	Degree	GPA
---------	------------------	-------------	--------	-----

professor	<u>ProfName</u>	ProfOffice	ProfPhone
-----------	-----------------	------------	-----------

course	<u>CourseNo</u>	CourseName	Credits
--------	-----------------	------------	---------

can_teach	<u>CourseNo</u>	<u>ProfName</u>	Preference	Evaluation
-----------	-----------------	-----------------	------------	------------

teaches	<u>CourseNo</u>	<u>ProfName</u>	Term
---------	-----------------	-----------------	------

enrolled	<u>CourseNo</u>	<u>ProfName</u>	<u>StudentNo</u>	Status
----------	-----------------	-----------------	------------------	--------

## Example RA Queries (cont'd)

---

Q1: List all attributes of students with a  $\text{GPA} \geq 10$

$\sigma_{\text{GPA} \geq 10} (\text{student})$

Q2: List student number and student name of students with a  $\text{GPA} \geq 10$

$\Pi_{\text{StudentNo}, \text{ StudentName}} (\sigma_{\text{GPA} \geq 10} (\text{student}))$

Q3: List student number and student name of B.C.S. students with a  $\text{GPA} \geq 10$

$\Pi_{\text{StudentNo}, \text{ StudentName}} (\sigma_{\text{GPA} \geq 10 \text{ AND } \text{Degree} = 'B.C.S'} (\text{student}))$

## Example RA Queries (cont'd)

---

Q4: List student number, name, and GPA of the students in the B.C.S program with a  $\text{GPA} \geq 10$  or in the B.Sc program with a  $\text{GPA} \geq 10.5$

$$\begin{aligned} & \Pi_{\text{StudentNo}, \text{ StudentName}, \text{ GPA}} ( \\ & \quad \sigma_{(\text{GPA} \geq 10 \text{ AND } \text{Degree} = 'B.C.S') \text{ OR} \\ & \quad (\text{GPA} \geq 10.5 \text{ AND } \text{Degree} = 'B.Sc')) \\ & \quad (\text{student})) \end{aligned}$$

- \* Parentheses in the selection condition are not needed here because AND has a higher precedence than OR

## Example RA Queries (cont'd)

---

Q5: List all attributes of students who are not in the B.C.S program

$\sigma_{\text{Degree} \neq 'B.C.S'} (\text{student})$

This query can also be written using logical NOT as follows

$\sigma_{\text{NOT } (\text{Degree} = 'B.C.S')} (\text{student})$

## Example RA Queries (cont'd)

---

Q6: List student number and student name of students enrolled in 95.305

$$\Pi_{\text{StudentNo}, \text{ StudentName}} (\sigma_{\text{CourseNo} = 95305} \\ (\text{student} \triangleright\!\!\!< \text{enrolled}))$$

Q7: List student number and student name of students enrolled in “Introduction to Database Systems”

$$\Pi_{\text{StudentNo}, \text{ StudentName}} (\sigma_{\text{CourseName} = \text{'Introduction to} \\ \text{Database Systems'}} \\ (\text{course} \triangleright\!\!\!< \text{enrolled} \triangleright\!\!\!< \text{student}))$$

## Example RA Queries (cont'd)

---

Q8: Give a list of all professors who can teach 95102 but are not assigned to teach this course

```
canTeach102 ← ΠProfName (  
    σCourseNo = 95102 (can_teach) )
```

```
teaching102 ← ΠProfName (   
    σCourseNo = 95102 (teaches) )
```

```
result ← canTeach102 – teaching102
```

## Example RA Queries (cont'd)

---

Q9: Give a list of all professors who are not teaching any course or teaching only summer courses

```
notTeaching ←  $\Pi_{\text{ProfName}}(\text{professor}) - \Pi_{\text{ProfName}}(\text{teaches})$ 
```

```
sumTeacher ←  $\Pi_{\text{ProfName}}(\sigma_{\text{Term} = 'S'}(\text{teaches})) - \Pi_{\text{ProfName}}(\sigma_{\text{Term} \neq 'S'}(\text{teaches}))$ 
```

```
result ← notTeaching  $\cup$  sumTeacher
```

---

## Example RA Queries (cont'd)

---

Q10: Give a list of all students (student number and name) who are taking all courses taught by Prof. Post

```
PostCourses ←  $\Pi_{\text{CourseNo}, \text{Profname}} (\sigma_{\text{ProfName} = 'Post'} (\text{teaches}))$ 
```

```
result ←  $\Pi_{\text{StudentNo}, \text{StudentName}, \text{CourseNo}, \text{ProfName}}$   
 $(\text{enrolled} \bowtie \text{student}) \div \text{PostCourses}$ 
```

## Example RA Queries (cont'd)

---

Q11: Find the average GPA of all students

$\text{F AVERAGE GPA (student)}$

Q12: For each course, find the enrollments

$\text{CourseNo F COUNT StudentNo (enrolled)}$

Q13: For each section of a course (identified by course number and professor name), give the enrollment

$\text{CourseName, ProfName F COUNT StudentNo (\textbf{enrolled} \bowtie \textbf{course})}$

# Relational Calculus

---

- Relational algebra is a procedural language
  - » Order among the operations is explicitly specified
- Relational calculus is non-procedural
  - » Specifies *what* is to be retrieved
    - Does not specify *how* to retrieve
- Expressive power of relational calculus and relational algebra is identical
- Two types of relational calculus languages:
  - \* Tuple relational calculus
    - variables represent tuples
  - \* Domain relational calculus
    - variables represent values of attributes

# Tuple Relational Calculus (cont'd)

---

- Tuple Relational Calculus (TRC) is based on specifying a number of tuple variables
- Each tuple variable ranges over a particular relation
  - » The variable can take any tuple in the relation as its value
- Expressions in TRC are of the form

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

where  $t$  is a tuple variable and  $F(t)$  is a predicate (called *formula*) involving  $t$

# Tuple Relational Calculus (cont'd)

---

## Examples

Q1: List all attributes of all students in the B.C.S program

```
{t | student(t) AND  
      t.Degree = 'B.C.S' }
```

Q2: List only student numbers and names of all students in the B.C.S program

```
{t.StudentNo, t.StudentName |  
  student(t) AND t.Degree = 'B.C.S' }
```

# Tuple Relational Calculus (cont'd)

---

## Formulas

- \* A formula is built from *atoms* and a collection of *operators*
- Atoms
  - \* Three types of atoms
    - 1)  $R(s)$  is an atom
      - $R$  is a relation name
      - $s$  is a tuple variable
        - Stands for “ $s$  is a tuple in relation  $R$ ”

# Tuple Relational Calculus (cont'd)

---

2)  $s.A \theta u.B$  is an atom

→ s and u are tuple variables

→ A is an attribute of the relation on which s ranges

→ B is an attribute of the relation on which u ranges

– θ is a comparison operator

→ one of  $\{=, \neq, <, \leq, >, \geq\}$

3)  $s.A \theta a$  and  $a \theta s.A$  are atoms

→ s is a tuple variable

→ A is an attribute of the relation on which s ranges

→ a is a constant value

– θ is a comparison operator

→ one of  $\{=, \neq, <, \leq, >, \geq\}$

---

# Tuple Relational Calculus (cont'd)

---

- Two quantifiers can appear in formulas
    - » universal quantifier  $\forall$  (read “for all”)
    - » existential quantifier  $\exists$  (read “there exists”)
  - Two types of tuple variables
    - \* Bound variable
      - » A variable introduced by a  $\forall$  or  $\exists$  quantifier
      - » Analogous to a local variable defined in a procedure
        - Cannot be accessed outside its scope
    - \* Free variable
      - » A variable that is not bound
      - » Analogous to a global variable
  - Quantifiers in relational calculus play the role of declarations in a programming language
-

# Tuple Relational Calculus (cont'd)

---

## Composition of formulas

- \* A formula can be made up several atoms
  - 1) Every atom is a formula
    - All occurrences of tuple variables mentioned in the atom are free in this formula
  - 2) If  $F_1$  and  $F_2$  are formulas, the following are also formulas:
    - **$F_1$  AND  $F_2$**
    - **$F_1$  OR  $F_2$**
    - **NOT  $F_1$**
    - **NOT  $F_2$**

# Tuple Relational Calculus (cont'd)

---

3) If F is a formula, so is

$$(\exists t)(F)$$

where t is a tuple variable

– Formula  $(\exists t)(F)$  is

→ TRUE if F evaluates to TRUE for *at least one tuple* assigned to free occurrences of t in F

→ FALSE otherwise

4) If F is a formula, so is

$$(\forall t)(F)$$

where t is a tuple variable

– Formula  $(\forall t)(F)$  is

→ TRUE if F evaluates to TRUE for *every tuple* assigned to free occurrences of t in F

→ FALSE otherwise

# Tuple Relational Calculus (cont'd)

---

- Expressions in TRC are of the form

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

where  $t$  is the only free tuple variable in  $F$

- There can be more than one free tuple variable
  - \* All of them must appear to the left of bar |
- We will look at several examples next

## TRC Examples (cont'd)

---

Q3: List all B.C.S students (student number and name) with  $\text{GPA} \geq 10$

```
{ t.StudentNo, t.StudentName | student(t)  
    AND t.Degree = 'B.C.S' AND t.GPA ≥ 10 }
```

Q4: List all students (student number and name) who are enrolled in Prof. Smith's 95.100 class

```
{ t.StudentNo, t.StudentName | student(t)  
AND (exists u) (enrolled(u) AND u.CourseNo = 95100  
    AND u.ProfName = 'Smith'  
    AND t.StudentNo = u.StudentNo) }
```

» In this expression, t is a free variable and u is a bound variable

## TRC Examples (cont'd)

---

Q5: List all professors who can teach 95.102 but are not assigned to teach this course

```
{ t.ProfName | can_teach(t)
    AND NOT (∃u) (teaches(u)
        AND t.CourseNo = 95102
        AND t.ProfName = u.ProfName
        AND u.CourseNo = 95102) }
```

## TRC Examples (cont'd)

---

Q6: List all professors who are not teaching any course

```
{ t.ProfName | can_teach(t) AND  
          (NOT (exists u) (teaches(u)  
          AND u.ProfName = t.ProfName) ) }
```

- We can also use “for all” quantifier

```
{ t.ProfName | can_teach(t) AND  
          (forall u) (NOT teaches(u)  
          OR u.ProfName ≠ t.ProfName) }
```

## TRC Examples (cont'd)

---

Q7: List all professors who are not teaching any course or teaching only summer courses

```
{ t.ProfName | can_teach(t) AND  
NOT (∃u) (teaches(u)  
AND u.ProfName = t.ProfName  
AND u.Term ≠ 'S' ) }
```

- “For all” version is better for understanding

```
{ t.ProfName | can_teach(t) AND  
(∀ u) (NOT teaches(u)  
OR u.ProfName ≠ t.ProfName  
OR u.Term = 'S' ) }
```

# Domain Relational Calculus

---

- For most part, similar to tuple relational calculus
- Variables range over values from domain of attributes
  - \* In TRC, variables represent tuples
    - » One variable for each relation
  - \* In DRC, variables represent attributes
    - » We need  $n$  variables for a degree  $n$  relation
      - One domain variable for each attribute
  - \* We need more variables in DRC than in TRC

# Domain Relational Calculus (cont'd)

---

- An expression in DRC is of the form

$$\{ \mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_n \mid F(\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_n, \mathbf{x}_{n+1}, \dots, \mathbf{x}_{n+m}) \}$$

where

$$\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_n, \mathbf{x}_{n+1}, \dots, \mathbf{x}_{n+m}$$

are domain variables that range over (not necessarily distinct) domains of attributes and F is a function

## Example

Q: List the names and student numbers of all B.C.S students

$$\{ m, n \mid (\exists o) (\text{student}(m, n, o, p) \text{ AND } o = 'B.C.S' )$$

# Domain Relational Calculus (cont'd)

---

- A DRC formula is made up of *atoms* and a collections of *operators* as in TRC
- Atoms
  - \* Three types of atoms
    - 1)  $R(\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_n)$  is an atom
      - $R$  is the name of a relation of degree  $n$
      - $\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_n$  are domain variables
        - States that a list of values  $\langle \mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_n \rangle$
        - must be a tuple in the database whose name is  $R$
        - $\mathbf{x}_i$  is the value of the  $i^{\text{th}}$  attribute value of that tuple

# Domain Relational Calculus (cont'd)

---

2)  $x_i \theta x_j$  is an atom

→  $x_i$  and  $x_j$  are domain tuple variables

–  $\theta$  is a comparison operator

→ one of  $\{=, \neq, <, \leq, >, \geq\}$

3)  $x_i \theta a$  and  $a \theta x_i$  are atoms

→  $x_i$  is a domain tuple variable

→ A is an attribute of the relation on which s ranges

→ a is a constant value

–  $\theta$  is a comparison operator

→ one of  $\{=, \neq, <, \leq, >, \geq\}$

- These atoms can be used to make up formulas as in the tuple calculus
-

# DRC Examples

---

Q1: List all B.C.S students (student number and name) with  $\text{GPA} \geq 10$

```
{m, n | (exists o) (exists p) (student(m, n, o, p)  
                                AND o = 'B.C.S' AND p ≥ 10) }
```

» In this expression, m and n are free variables and o and p are bound variables

Q2: List all students (student number and name) who are enrolled in Prof. Smith's 95.100 class

```
{m n | student(m n o p)  
                                AND (exists q) (exists r) (exists s) (enrolled(q r s t)  
                                AND q = 95100 AND r = 'Smith'  
                                AND s = m) }
```

# Query-By-Example

---

- Query-by-example (QBE)
  - » Developed in the early 1970s by IBM at T. J. Watson Research Center
    - Available as a part of QMF in DB2
    - Variants of QBE are used in personal computer DBMSs
      - e.g., Microsoft Access
  - » Based on domain relational calculus
  - » Uses table templates to express queries
  - » Queries are expressed by using domain variables and constants
  - » QBE queries are expressed by example
    - User gives an example of what is desired
    - System generalizes this example to find the answer to the query

# QBE Examples

---

Q1: Give a list of B.C.S students (all attributes)

student	StudentNo	StudentName	Degree	GPA
P.			B.C.S	

Q2: Give a list of B.C.S students (only name and student number)

student	StudentNo	StudentName	Degree	GPA
	P.	P.	B.C.S	

## QBE Examples (cont'd)

---

Q3: Give a list of students (student number and names)  
taking Prof. Smith's 95.100 class

student	StudentNo	StudentName	Degree	GPA
	P._SN	P.		

enrolled	CourseNo	ProfName	StudentNo	Status
	95100	Smith	_SN	

## QBE Examples (cont'd)

---

Q4: Give a list of professors who can teach 95102 but are not teaching this course

can_teach	CourseNo	ProfName	Preference	Evaluation
	95102	P._PN		

teaches	CourseNo	ProfName	Term
⊤	95102	_PN	

## QBE Examples (cont'd)

---

Q5: Give a list of professors who are not teaching any course or teaching only summer courses

can_teach	CourseNo	ProfName	Preference	Evaluation
		P._PN1		
		P._PN2		

teaches	CourseNo	ProfName	Term
⊤		_PN1	
⊤		_PN2	≠ S

## QBE Examples (cont'd)

---

Q6: Give a list of B.C.S students with  $\text{GPA} \geq 10$   
and(means union here) B.Sc students with a  $\text{GPA} \geq 11$

student	StudentNo	StudentName	Degree	GPA
	P.	P.	B.C.S	$\geq 10$
	P.	P.	B.Sc	$\geq 11$

## QBE Examples (cont'd)

---

Q7: Give a list of professors who are teaching both 95.100 and 95.102 in the fall term

teaches	CourseNo	ProfName	Term
	95100	P._PN	F
	95102	P._PN	F

- This is wrong answer
  - This query retrieves professors who teach either 95.100 or 95.102 in the fall term
  - Right answer is on the next slide

## QBE Examples (cont'd)

---

Some time it is difficult to express all the constraints on the variables using table templates. *Condition box* allows the expression of general constraints

teaches	CourseNo	ProfName	Term
	95100	P._PN1	F
	95102	P._PN2	F

conditions

$_PN1 = _PN2$

# QBE Examples (cont'd)

---

Q8: Give a list of students taking a fall term course

student	StudentNo	StudentName	Degree	GPA
	P._SN	P.		

enrolled	CourseNo	ProfName	StudentNo	Status
	_CN	_PN	_SN	

teaches	CourseNo	ProfName	Term
	_CN	_PN	F

## QBE Examples (cont'd)

---

Q9: Give a list of students who are not taking any course offered by Prof. Post in the fall term

student	StudentNo	StudentName	Degree	GPA
	P._SN	P.		

enrolled	CourseNo	ProfName	StudentNo	Status
¬	_CN	Post	_SN	

teaches	CourseNo	ProfName	Term
	_CN	Post	F

## QBE Examples (cont'd)

Q10: Give a list of students who are taking both summer term courses offered Prof. Smith

student	StudentNo	StudentName	Degree	GPA
	P._SN1	P.		

enrolled	CourseNo	ProfName	StudentNo	Status
	_CN1	Smith	_SN1	
	_CN2	Smith	_SN2	

teaches	CourseNo	ProfName	Term
	_CN1	Smtih	S
	_CN2	Smith	S

conditions

$_SN1 = _SN2$

## QBE Examples (cont'd)

---

- Q11: Give a list of fall courses
  - sort the answer first on course# (ascending order) and professor name (descending order) within course#

teaches	CourseNo	ProfName	Term
	P.AO(1).	P.DO(2).	F

- To sort use AO. (ascending order) or DO. (descending order)
- Sort order can be specified by attaching a number AO(1), DO(2)

## QBE Examples (cont'd)

---

- QBE supports the standard aggregate functions SUM, MIN, MAX, AVG, CNT (for count)
  - Must use ALL. prefix so that duplicates are not eliminated
    - Example: P.SUM.ALL.
- If you want to eliminate duplicates, use UNQ. as in P.CNT.UNQ.ALL.
- Use G. to group results
- Q12: For each degree, give the average GPA

student	StudentNo	StudentName	Degree	GPA
			P.G.	P.AVG.ALL.

## QBE Examples (cont'd)

---

Q13: List all fall courses (course# and the professor teaching the course) with an enrollment > 100

teaches	CourseNo	ProfName	Term
	P._CN	P._PN	F

enrolled	CourseNo	ProfName	StudentNo	Status
	G._CN	_PN	CNT.ALL._SN	

conditions

CNT.ALL._SN > 100
-------------------

---

# **CPS510**

## **Database Management Systems (DBMS)**

### **Topic 8: Physical Database Organization and Indexing**

From Elmasri and Navathe, *Fundamentals of  
Database book*  
Chapters 16 and 17

**Edited By Abdolreza Abhari  
Department of Computer Science  
Ryerson University**

---

# Topic 8 Reading List

---

Elmasri and Navathe, *Fundamentals of Database Systems*, Seventh edition

- \* Chapter 16: *Disk Storage, Basic File Structures, and Hashing*
  - » All sections
- \* Chapter 17: *Index Structures for Files*
  - » All sections

# Roadmap

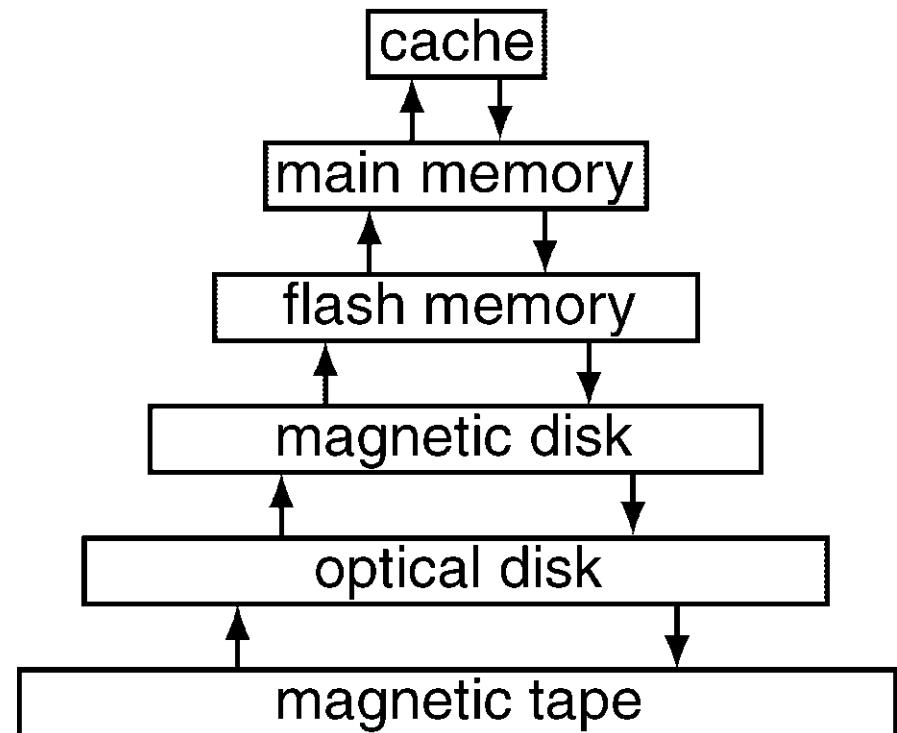
---

- Physical storage
  - \* Disk characteristics, Disk parameters
- File organizations
  - \* Unordered and ordered files, Hashing
- Index structures
  - \* Single-level indexes
    - » Primary, clustering, secondary
  - \* Multilevel indexes
    - » Static, dynamic
    - » Search tree, B-tree, B+-tree, B\*-tree

# Physical Storage

---

- Three main categories of storage medium
  - \* Primary storage
    - » Main memory
    - » Cache memory
  - \* Secondary storage
    - » Primarily magnetic disk storage
    - » Flash memories for small volume
    - » Solid State Storage
  - \* Tertiary storage
    - » Optical disks
    - » Magnetic tapes



# Physical Storage (cont'd)

---

- Primary memory
  - \* Includes main memory and cache memory
  - \* Fast access
    - » Expensive
    - » Limited capacity
    - » Volatile
      - Persistency is important for database applications
    - » Data in primary memory can be processed directly
      - All data will have to be placed in main memory buffers before processing
    - » Performance improvement with cache memory
      - Nothing special to databases

# Physical Storage (cont'd)

---

- Secondary storage
  - \* Mainly disk storage
  - \* Larger capacity
    - » Lower cost
    - » Slower
    - » Non-volatile
      - Disk failures can still cause data problems
    - » Must be copied into primary storage for processing
  - \* Flash memory
    - » Solid State Drives, USB Sticks
    - » It is an EEPROM (Electrically Erasable Programmable Read Only Memory)

# Physical Storage (cont'd)

---

- Tertiary storage
  - \* Mainly magnetic tapes
    - » Primarily used for backup and archival data
    - » Used for recovery from certain types of disk failures
    - » Slower access
      - Provides sequential access (no random access)
    - » Removable media
      - Jukebox type systems used for large capacity
  - \* Optical disks
    - » Similar to CD-ROMs
    - » Not as popular due to writing problems
    - » Removable media (jukebox type systems for large capacity)

# Physical Storage (cont'd)

---

- Most databases are stored on disks
    - \* Databases are too large to fit in main memory although memory based database are growing
    - \* Disks provide a non-volatile storage
      - » Sometimes called *stable* or *persistent* storage
    - \* Less expensive
      - » An order of magnitude less expensive than main memory
  - Main problem
    - \* Too slow
      - » Performance can be improved by
        - System design (e.g., disk cache)
        - Better access paths (e.g., indexing)
        - Better file organizations (e.g., block clustering)
-

# Disk Device Types

---

- Two main types of disk drives are common
  - \* Electrical: Solid State Drives
    - » Are still expensive
    - » Faster access
  - \* Mechanical: Moving-head drive
    - » Disks are made up of thin circular-shaped magnetic material
    - » Each disk surface is divided into concentric circles called *tracks*
    - » Tracks with the same diameter on various surfaces are called a *cylinder*
    - » Each track is divided into *sectors* or *blocks*
      - Basic unit of transfer between main memory and disk

# Disk Parameters

---

- In mechanical HD most important parameter is
  - \* Access time
    - » Given a block number, it is the time required to transfer the block to main memory buffer
  - \* Consists of three main components
    - » Seek time
    - » Rotational delay
    - » Block transfer time

Access time = Seek time + Rotational delay +  
Block transfer time

---

# Disk Parameters (cont'd)

---

- Seek time,  $s$ 
  - \* Movable-head disks
    - » Time needed to place the read/write head on the target track
    - » Time varies depending on the distance between the current position and target track
    - » Usually average seek time is specified (for a random track/cylinder)
    - » This is the main component of delay

# Disk Parameters (cont'd)

---

- Rotational delay,  $rd$ 
  - » After the R/W head is on the correct track, we have to wait for the beginning of the block to rotate into a position under the R/W head
  - » On average, it takes half a revolution time
  - » If the disk rotates at  $p$  RPM (revolutions per minute)  
$$rd = 1/2p \text{ min} = 30,000/p \text{ msec}$$

## Disk Parameters (cont'd)

---

- Block transfer time, *btt*
  - \* Time needed to transfer data in the block
  - \* It depends on
    - » Block size
    - » Track size
    - » Rotational speed (RPM)
  - \* If disk transfer rate is *tr* bytes/sec and block size is *B* bytes
$$btt = B/tr \text{ sec}$$
  - \* *btt* for a 512-byte block: 100 - 25  $\mu$ sec

# Disk Parameters (cont'd)

---

- How do we reduce the access time?
  - \* Improvements in disk technology
  - \* Smart data placement
    - » Seek time reduction
      - Transfer several blocks stored on one or more tracks of the *same cylinder*
      - Seek time needed only for the first block
      - Time to transfer consecutively k non-contiguous blocks on the same cylinder =  $s + k * (rd + btt)$  msec
    - » Seek time and rotational delay reduction
      - Transfer consecutive blocks on the same track or cylinder
      - Time =  $s + rd + k * btt$  msec

# Cost Effectiveness & Reliability

---

- Disk drive densities are improving
  - \* Getting smaller, cheaper, larger capacity
- Cost-effective to use
  - \* Large number of smaller, cheaper disks
    - Improves reliability as well
  - \* Rather than small number of large disks
- RAID is proposed to exploit this
  - \* Redundant Arrays of Inexpensive Disks (RAID)
  - \* Various levels
    - RAID 0 to RAID 6

# Cost Effectiveness & Reliability (cont'd)

---

- Three fundamental concepts
  - \* Mirroring
    - » Provides high reliability
    - » Expensive
  - \* Striping
    - » Spreading blocks, for example, across the array of disks
    - » Provides high data transfer rates
    - » Does not improve reliability
  - \* Parity
    - » Used to detect errors
    - » Can also be used to recover data (error correction)
    - » In addition, block level error checking is done at the disk

# File Records & Their Placement

---

- File is a collection of records
  - » Each tuple of a relation is stored as a record in the file representing the relation
- Record characteristics and their placement of disk is important for performance and efficiency
  - \* Record types
    - » Fixed-length
    - » Variable-length
  - \* Mapping of records to disk blocks
    - » Spanned records
    - » Unspanned records

# File Records & Their Placement (cont'd)

---

- Record types
  - \* Fixed-length records
    - » All records of the file are of the same size
      - Efficient processing
      - Efficient storage
  - \* Variable-length records
    - » Different records in the file have different sizes
      - Even if all records have same number and type of fields
        - One or more fields may have variable length data
          - E.g. VARCHAR data type in SQL
        - Optional field
      - Different fields

## File Records & Their Placement (cont'd)

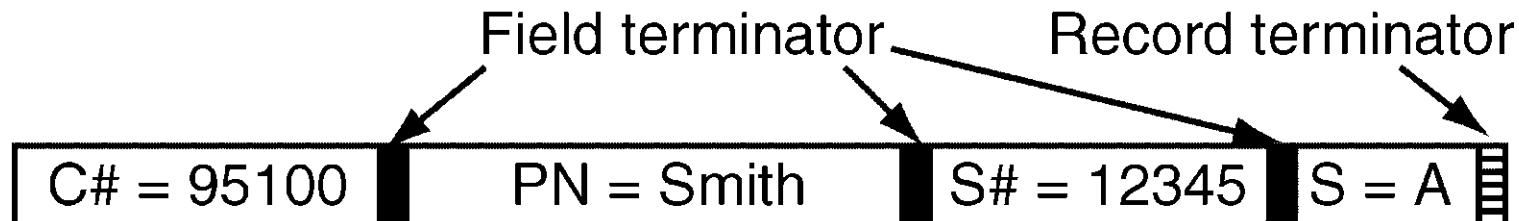
CourseNo	ProfName	StudetnNo	Status
95100	Smith	12345	A

Fixed-length fields for all fields

Field terminator

CourseNo	ProfName	StudetnNo	Status
95100	Smith	12345	A

ProfName = variable-length field, others = fixed-length



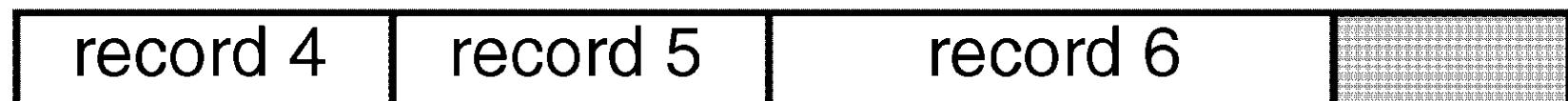
Record stored under different types of fields assumption

# File Records & Their Placement (cont'd)

---

- Blocking records
  - \* Blocking factor  $bfr$  depends on
    - Block size B
    - Record size R
  - \* Unspanned records
    - » A record is not split between blocks
    - » Fixed records:
      - $bfr = \lfloor B/R \rfloor$  records/block
      - Wasted storage space/block =  $B - (bfr * R)$
    - » Variable records: Use average value
  - \* Spanned records
    - » Record is split between two blocks to save space

# File Records & Their Placement (cont'd)



**Unspanned**



**Spanned**

rest of record 4

# Evaluation Metrics

---

- \* Access types
  - » Types of access supported efficiently
    - Exact match, range search, ...
- \* Access time
  - Time to find a record, or a set of records
- \* Insertion time
  - Time to insert a new record
    - Includes time to find the correct place + index update
- \* Deletion time
  - Time to delete a record (similar to insertion time)
- \* Space overhead
  - Additional overhead for indexes
    - Space versus time tradeoff

# File Organizations

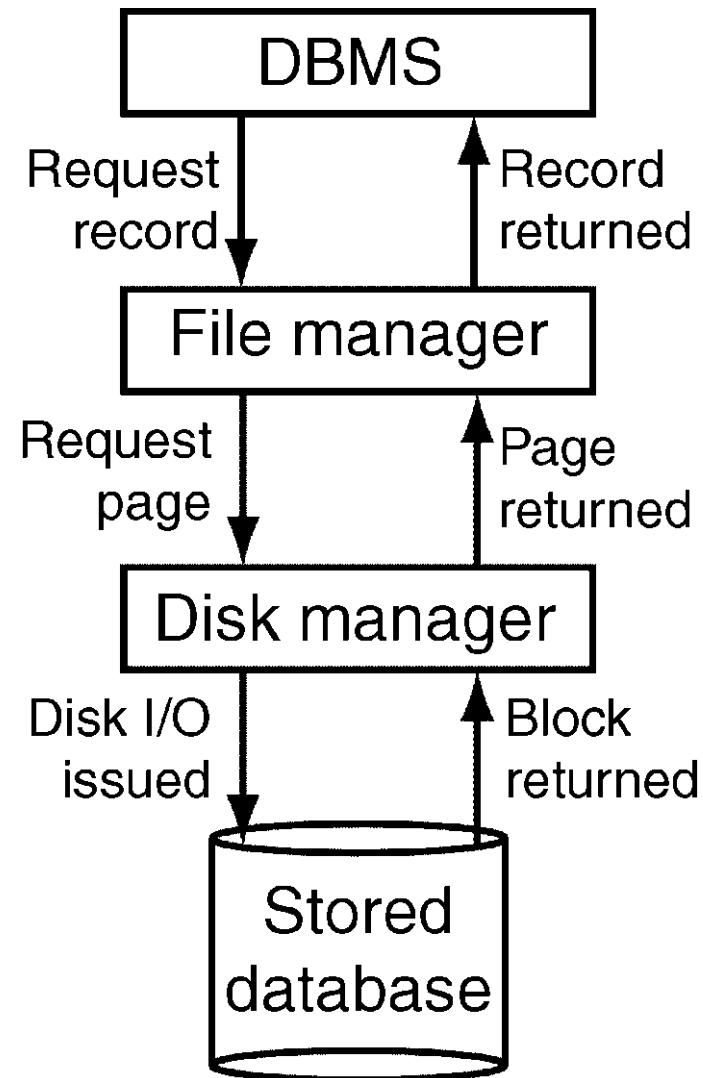
---

- Records are stored in blocks
  - »  $bfr$  gives the number of records/block
  - » In principle, we can retrieve records based on the value of any attribute
  - » Typically, only a subset of attributes (called *keys*) are used to retrieve records
- Files are organized so that retrieval based on keys is efficient
  - » *Primary key*: is a set of attributes that *uniquely* identify a record
  - » *Secondary key*: Identify a *set* of records
- Index structures can be added to speedup access of records

# File Organizations

## Accessing Data

- Details vary from system to system, but the basic aspects are the same
  - \* DBMS
    - » Determines what record is required
  - \* File manager
    - » Determines the page that contains the record
  - \* Disk manager
    - » Determines the physical location
    - » Issues necessary I/O command to get it



# Unordered Records (or Heap) Files

---

- Term sequential files has been used instead of unordered files
- Records are placed in the order they are inserted
- Insertion
  - \* Very efficient
    - » Last disk block is copied into main memory buffer
    - » Add the new record at the end
    - » Write the block back to disk
- Search
  - \* Linear search block-by-block
    - » If search is on a primary key (at most one record exists), requires half the blocks on average
    - » Multiple records require searching all blocks

# Unordered File (cont'd)

---

- Deletion
  - \* Search for the record, then delete
    - » First search for the record
    - » Read the block into main memory buffer
    - » Delete the record from the buffer
    - » Write the buffer back to disk
  - \* Leaves unused space (“holes”) in the disk block
    - » Periodic reorganization is required to claim the space
    - » File becomes unavailable during reorganization
- Storage overhead
  - \* No additional overhead

# Ordered Sequential File

---

- Records are stored in order
  - \* Ordered by *ordering field*
    - » Typically primary key is used as the ordering field
  - \* Advantages of ordered file over unordered file
    - » No additional sorting is required
      - Already in sorted order
    - » Finding next record usually does not require additional block accesses
    - » Search conditions involving  $<$ ,  $>$ ,  $\geq$ , and  $\leq$  on ordering field are efficient
    - » Binary search technique can be applied to speedup search
      - Requires  $\log_2 b$  block accesses in all cases

# Ordered Sequential File (cont'd)

---

## \* Problems

- » Binary search is not used due to seek time and rotational delay
  - Indexing techniques are better suited
- » Inserting and deleting records are expensive
  - Due to physical order requirement
- » Inserting involves shifting records forward
  - Can be improved by
    - Keeping some unused space in each block for future insertions
    - Creating a temporary overflow file (“transaction” file)
- » Deletion requires compression
  - Can be simplified by marking the record deleted
  - Periodic reorganization compresses file

# Hashing

---

- Basic idea:
  - » Use a formula instead of a search tree to get the block pointer of the record
- We get the record address by computing a hash function on the search key value
  - » We use the term bucket to represent storage that can store one or more records
  - » Typically, a bucket is a disk block
    - But a bucket can be smaller or larger than a disk block
- A hash function  $h$  is a mapping from K to B
  - where K = set of all search key values
  - B = set of all bucket addresses

# Hashing (cont'd)

---

- Searching
  - \* To look for a record with search key value  $K_i$ 
    - find the bucket address =  $h(K_i)$
    - search the bucket for the record
- Insertion
  - \* To insert a record with search key value  $K_i$ 
    - find the bucket address =  $h(K_i)$
    - insert the record into the bucket
- Deletion
  - \* To delete a record with search key value  $K_i$ 
    - find the bucket address =  $h(K_i)$
    - delete the record (if exists) in the bucket

# Hashing (cont'd)

---

- Properties of good hash functions
  - \* Two basic properties
    - Uniform distribution
    - Random distribution
  - » Guarantee equal utilization of all buckets
  - \* Uniform distribution
    - » Search key value distribution to buckets should be uniform
    - » Each bucket should be assigned the same number of search key values from the set of *all possible key values*
    - » Example: If E represents **employee#**, hash function
$$E \bmod B$$
distributes key values uniformly *if there are no gaps in E*

# Hashing (cont'd)

---

- \* Random distribution
  - » In the average case, all buckets will have nearly the same number of values
  - » Implies “uniform distribution” on whatever we inserted
    - Not only when all the records are inserted
  - » Example: The previous hash function ( $E \bmod B$ ) does not have the randomness property
    - Suppose  $B = 7$
    - Inserting records with **employee#** that is a multiple of 7
      - such as      0, 7, 14, 70, 49, ...
- \* Bucket overflow need to be handled even with good hash functions

# Bucket Overflow

---

- Several causes for bucket overflow
  - \* Insufficient number of buckets
    - » To provide sufficient storage space using a good hash function (i.e., with uniform and random distribution properties)
      - we have to assign a number of buckets that can take the largest anticipated number of records
      - If the actual number turned out to be much smaller than anticipated
        - we waste storage space
        - If we assign fewer, buckets will be full
          - Bucket overflow causes performance deterioration
          - Significant increase in search insertion/deletion times

# Bucket Overflow (cont'd)

---

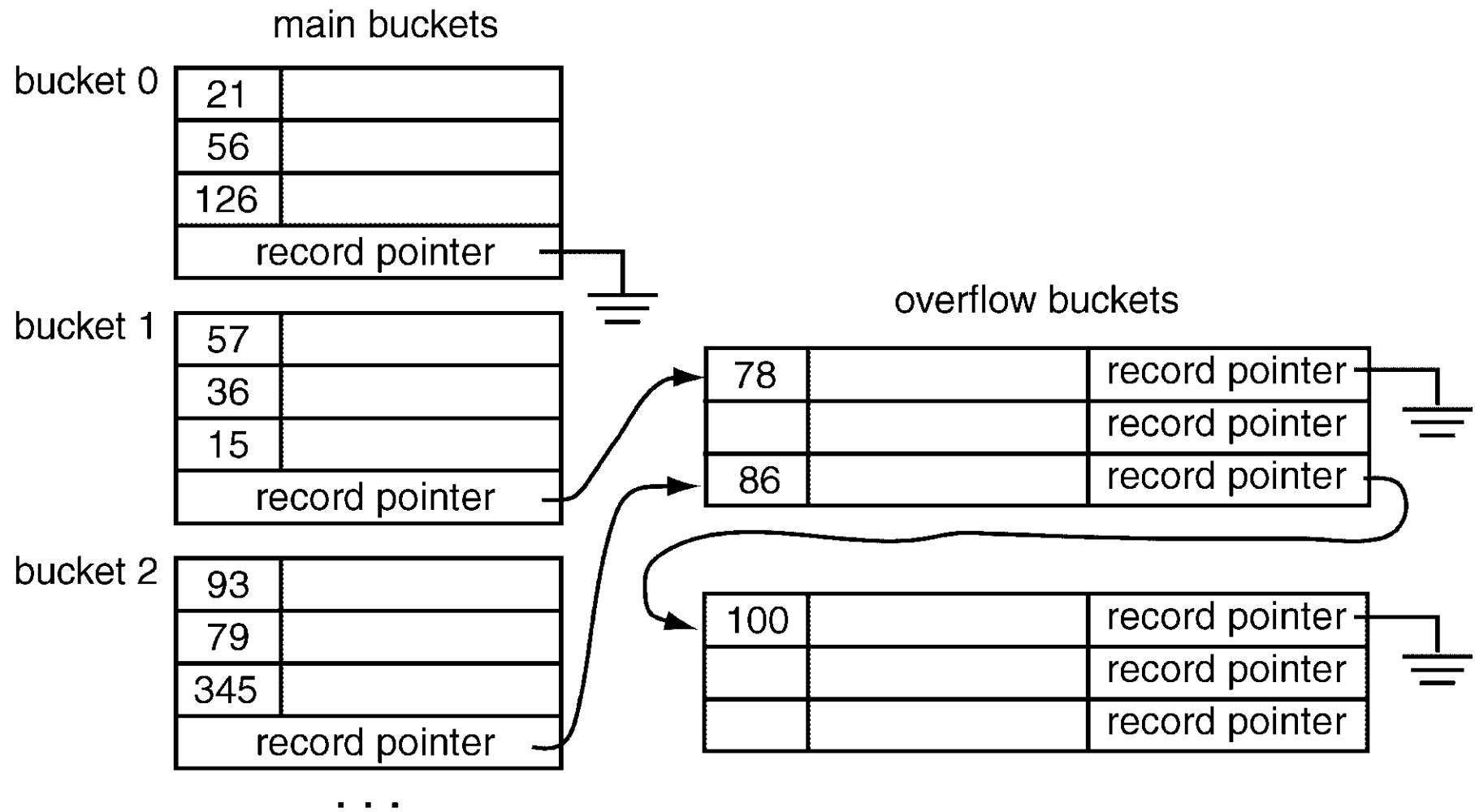
- \* Bucket skew
  - » Search key distribution may be non-uniform
    - Several records with same key value
    - Even a good hash function cannot help
  - » Hash function distributes key values non-uniformly
    - Even a hash function with uniform property may distribute records non-randomly for a particular insertion order
- Possibility of buck overflow can be reduced by providing some extra space
  - \* We can allocate, say, 10-20% more space than the anticipated requirement

# Bucket Overflow (cont'd)

---

- Several techniques to handle bucket overflows
  - \* Open addressing
    - » Also called open hashing
    - » When a bucket is full, look for a next bucket (in cyclic order) that has space for the record
  - \* Overflow chaining
    - » A separate overflow area is kept for all bucket overflows
    - » A linked list of overflow records for each hash address is maintained
  - \* Multiple hashing
    - » A second hash function is applied to locate another bucket
    - » If not successful, a third hash function can be applied

## Bucket Overflow (cont'd)



# Dynamic Hashing

---

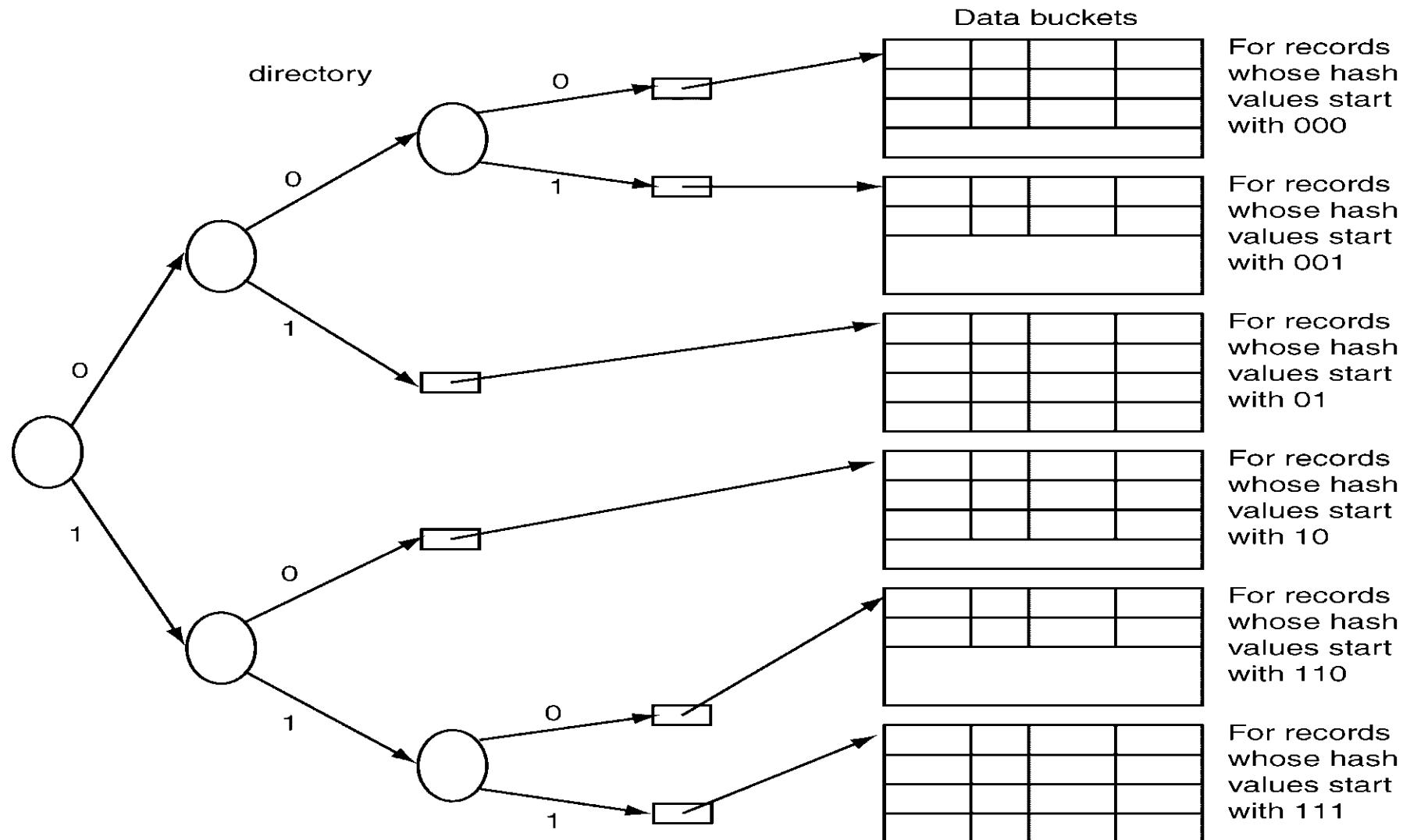
- Previous scheme is called *static hashing*
  - \* Problem: Hash address space is fixed
    - » Difficult to expand or shrink the file size dynamically
  - \* Three options if using static hashing
    - » Choose a hash function based on the current file size
      - Leads to performance degradation as database grows
    - » Choose a hash function based on the anticipated size
      - Performance degradation is avoided
      - Significant storage space wasted initially
    - » Periodic reorganization in response to file growth
      - Choose a new hash function at reorganization time
      - Reorganization is very time-consuming
      - Database cannot be accessed during reorganization

# Dynamic Hashing (cont'd)

---

- In dynamic hashing
  - \* Number of buckets is not fixed
  - \* Starts with a single bucket
  - \* Once full, the bucket is split into two buckets
    - » Records are distributed between the two buckets based on the first (leftmost) bit of their hash value
    - » Results in a directory with two types of nodes:
      - Internal nodes
        - Guide the search process
      - Leaf nodes
        - Pointers to buckets
    - » Similar to split, buckets can be combined to shrink

# Dynamic Hashing (cont'd)

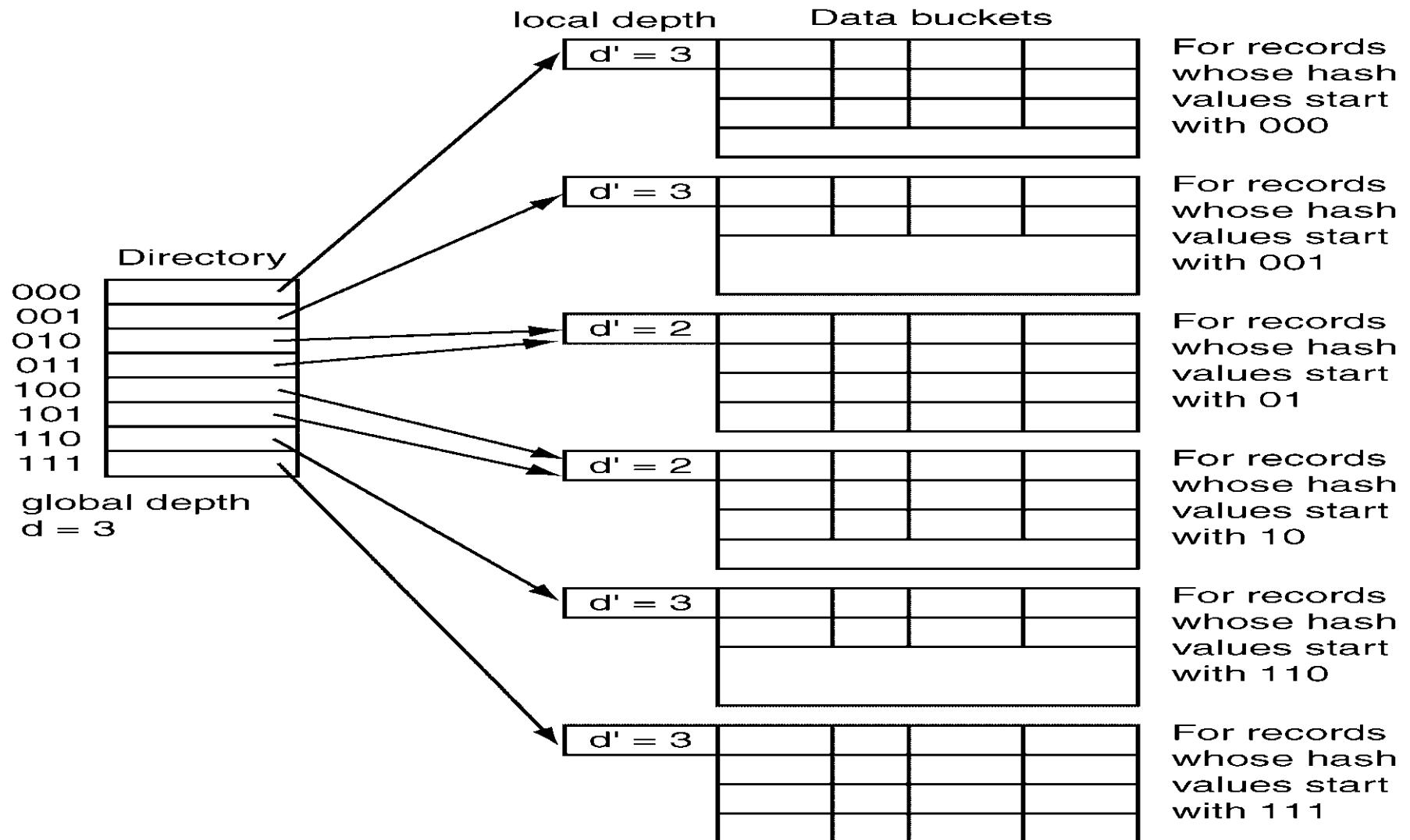


# Dynamic Hashing (cont'd)

---

- Extendible hashing
  - \* Uses a very large bucket address space
    - » Bucket address is of  $b$  bits
    - » Number of buckets =  $2^b$  (usually,  $b = 32$ )
  - \* A directory with leftmost  $d$  bits of hash value is built
    - »  $d$  is called *global depth*
    - »  $2^d$  entries in the directory
    - » There need not be  $2^d$  buckets
    - » A *local depth*  $d'$  for each bucket specifies the number of higher-order bits on which its contents are based
  - \* The value of  $d$  can be increased/decreased one at a time
    - » Doubling: when a bucket with  $d' = d$  overflows
    - » Halving: if  $d > d'$  for all buckets after some deletions

# Dynamic Hashing (cont'd)



# Index Structures

---

- An index is a set of <key, address> pairs
  - \* A sequential file that is indexed is called indexed-sequential file
    - » Index provides efficient access for random access of records
    - » Sequential nature of file provides efficient access for sequential processing
  - \* Indexing allows us to look at logical rather than physical order of records
    - » Leads to more efficient insertions and deletions
  - \* It is also important to take disk characteristics into account
    - » E.g. clustering of records based on access patterns

# Index Structures (cont'd)

---

- Types of indexes
  - \* Single-level index
    - » Primary index (index on the ordering field)
      - Can use sparse indexing
    - » Clustering index (index on the ordering field)
      - For files with records ordered on a non-key field
    - » Secondary index (index on a non-ordering field)
      - May use dense or sparse indexing
        - Depends on the indexing field
  - \* Multilevel indexes
    - » Static multilevel indexes
    - » Dynamic multilevel indexes
      - B-tree, B\*-tree, B<sup>+</sup>-tree

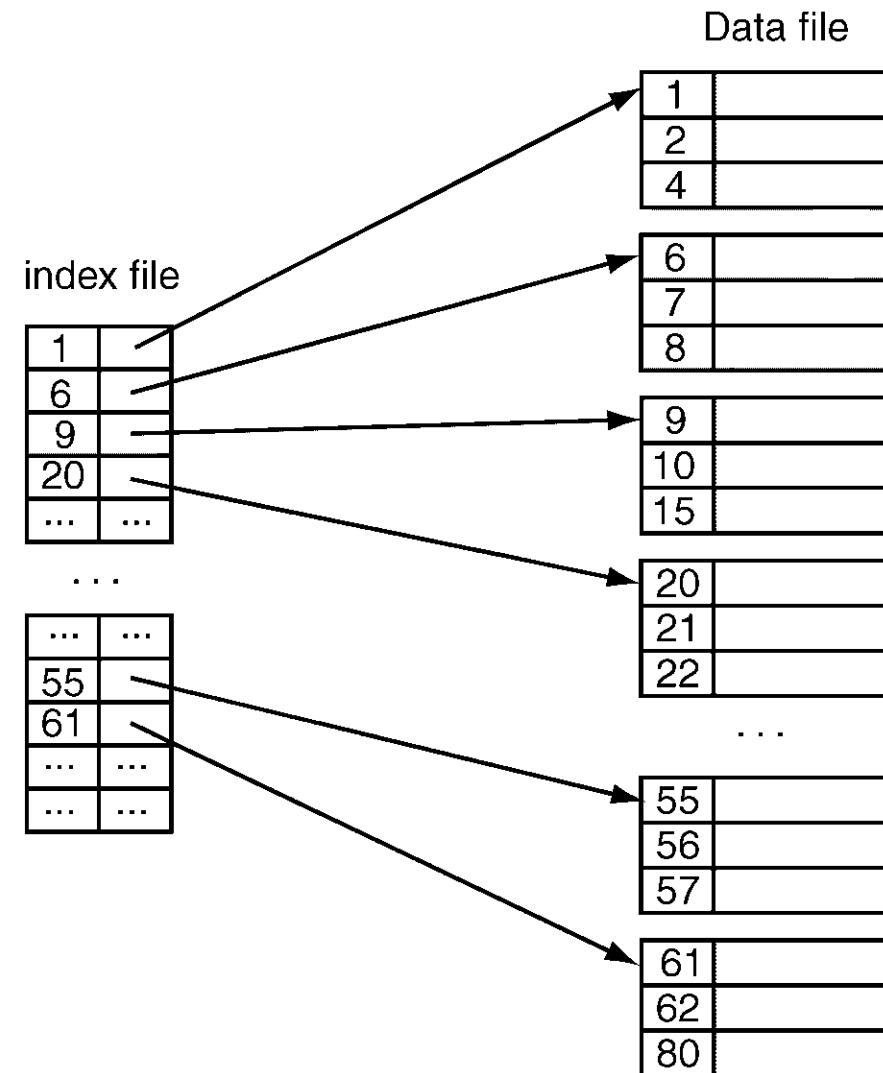
# Index Structures (cont'd)

---

- Primary index
  - \* A primary index is an ordered file of fixed-length records with two fields:
    - Ordering field, disk block address
  - \* One index entry for *each block* of data file
    - » Called sparse or nondense index
      - Dense index is one with one index entry for *each data file record*
    - » First record in each data file block is called *anchor record* of the block (simply *block anchor*)
  - \* Insertion/deletion is a major problem

# Index Structures (cont'd)

Primary index



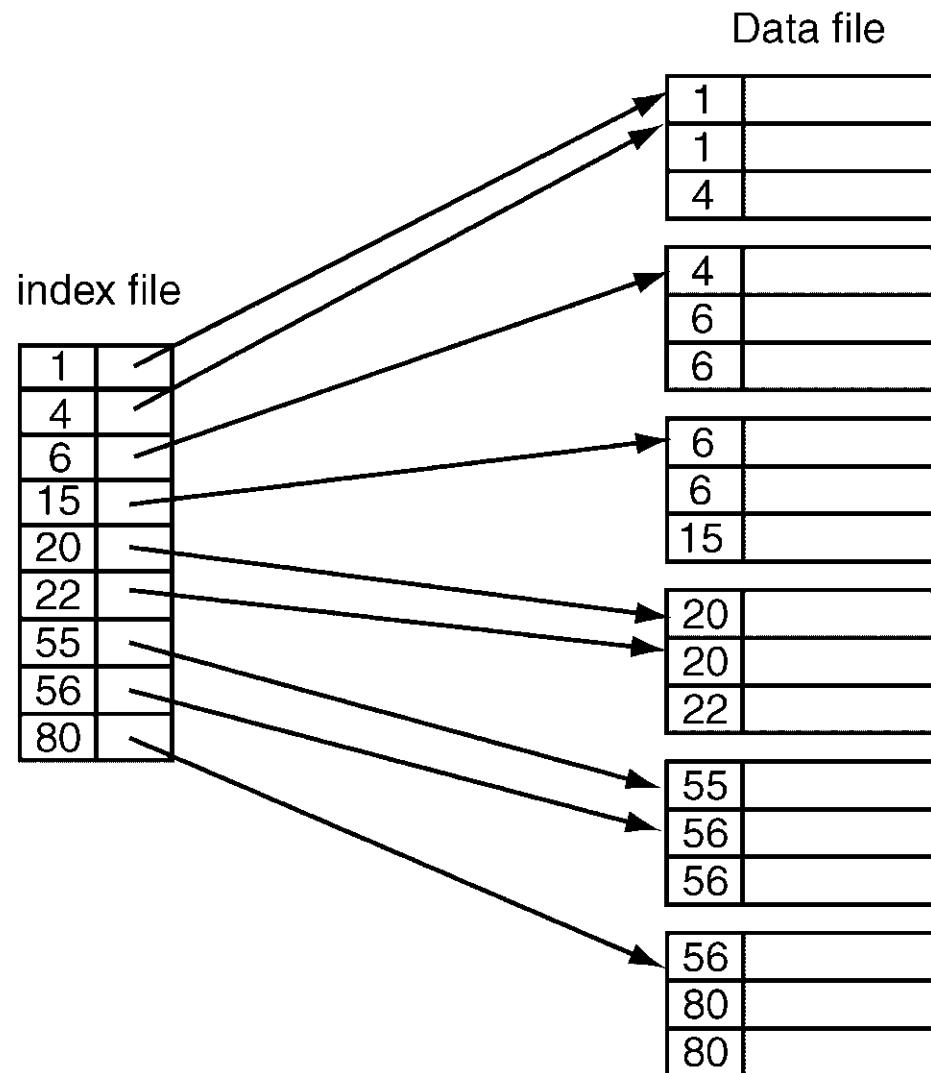
# Index Structures (cont'd)

---

- Clustering index
  - \* For files ordered on a non-key field
    - » Several records with the same ordering field value
  - \* One index entry for each distinct value of the indexing field
  - \* Typically results in a nondense index
    - » Depends on the degree of clustering
  - \* Insertion/deletion is a major problem as in the primary index
    - » Can be improved by allocating a block for each index value
    - » Additional needed blocks can be allocated later and linked

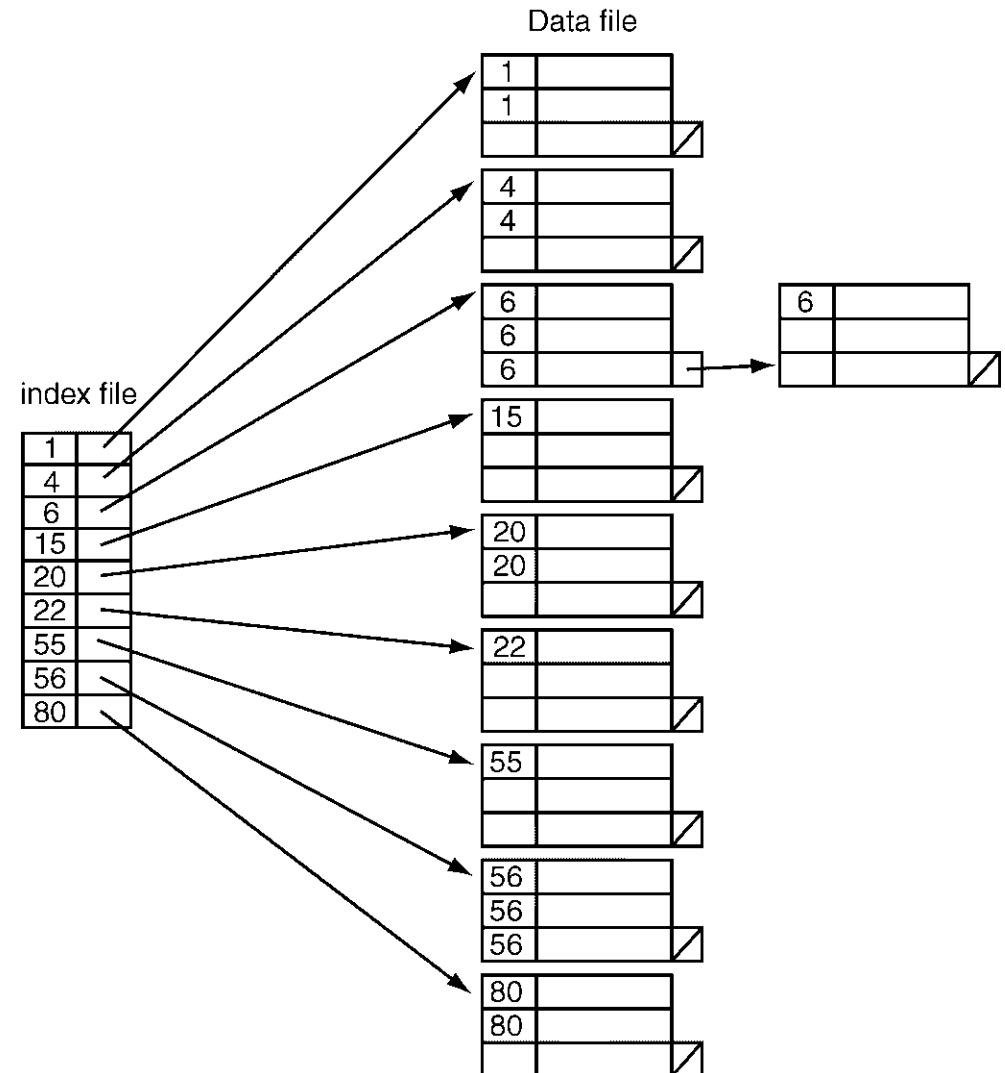
# Index Structures (cont'd)

## Clustering index



# Index Structures (cont'd)

Clustering index  
with links



# Index Structures (cont'd)

---

- Secondary indexes
  - \* Primary and clustering indexes are restricted to the file *ordering field*
    - » Primary index is restricted to key field
    - » Clustering can be a nonkey field
    - » There can only either a single primary index or a single clustering index (but not both)
  - \* Secondary indexes are not on the ordering field
    - » Several secondary indexes are possible
      - You can use any non-ordering field as an index field
    - » Often built in addition to primary/clustering index to improve access performance

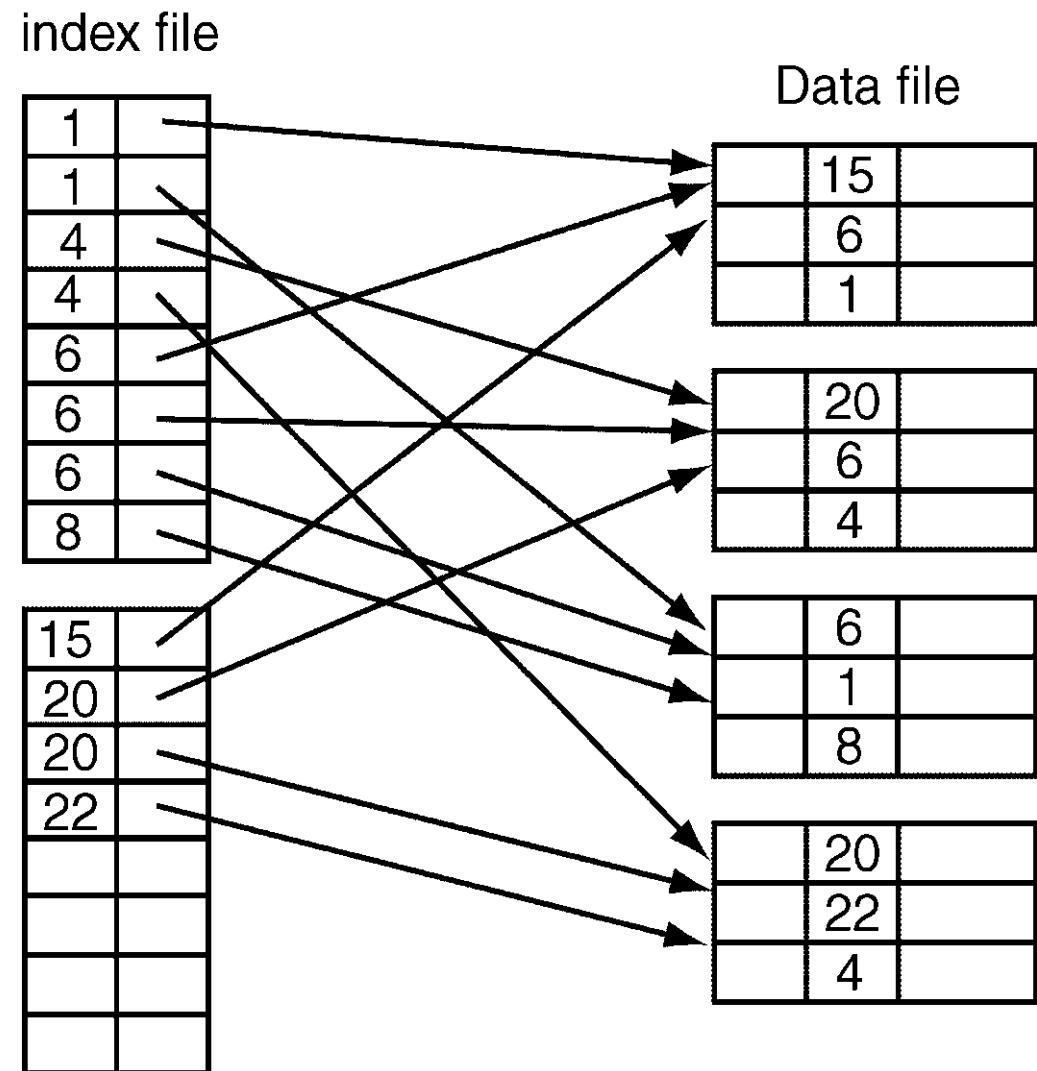
# Index Structures (cont'd)

---

- Secondary indexes
  - \* Two types
    - » Index field is a key field
      - Dense index
    - » Index field is a nonkey field
      - Several options
      - OPTION1
        - Several index entries with the same index values
        - Dense index
      - OPTION 2
        - Use variable length records for indexes
        - A repeating field for pointers

# Index Structures (cont'd)

# Dense secondary index



# Index Structures (cont'd)

---

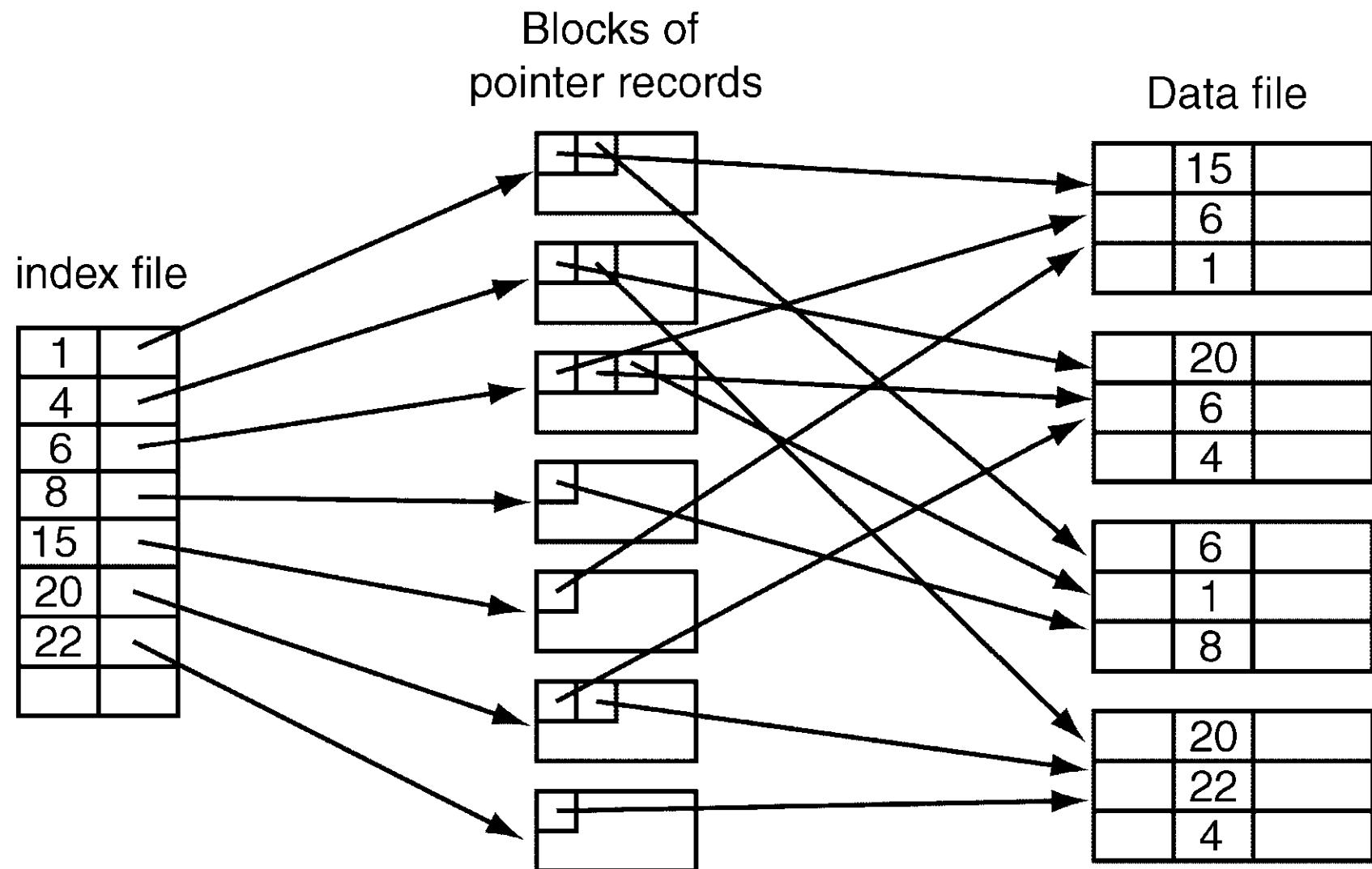
## – OPTION3

- Keep index entries fixed length
- Single entry for each distinct index value
- Use a level of indirection
- Keep a list of pointers in this
- Commonly used technique

## Summary of index types

	<b>Ordering field</b>	<b>Non-ordering field</b>
<b>Key field</b>	Primary index	Secondary index (key)
<b>Nonkey field</b>	Clustering index	Secondary index (nonkey)

# Index Structures (cont'd)



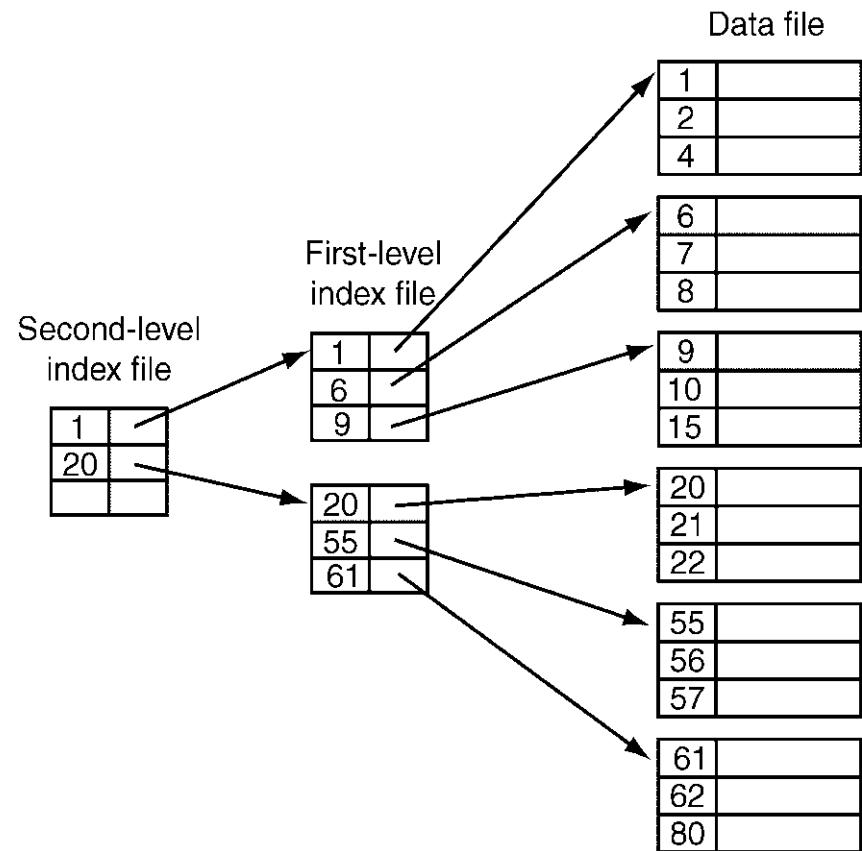
# Multilevel Index Structures

---

- Multilevel Indexes
  - \* Previous schemes use an ordered index file
    - » Binary search can be used to search the index file for a particular record
    - » Binary search requires  $\log_2 b_i$  block accesses  
 $b_i = \# \text{ of index blocks}$
  - \* Motivation for multilevel indexes
    - » Performance can be improved over binary search by using multiway branching  $fo$  than just 2
    - » Requires  $\log_{fo} b_i$  block accesses (better for  $fo > 2$ )
    - » For large data files, the index file may be large
      - May cause binary search to be inefficient
    - » Prefer an index file that is compact

# Multilevel Index Structures (cont'd)

- Static multilevel index
  - » Number of levels is a design parameter
  - » Does not change dynamically as the data file expands
  - » Suitable for files that do not change much in terms of file size
    - Infrequent insertions and deletions
- Dynamic indexes adjust the number of levels depending on the data file size

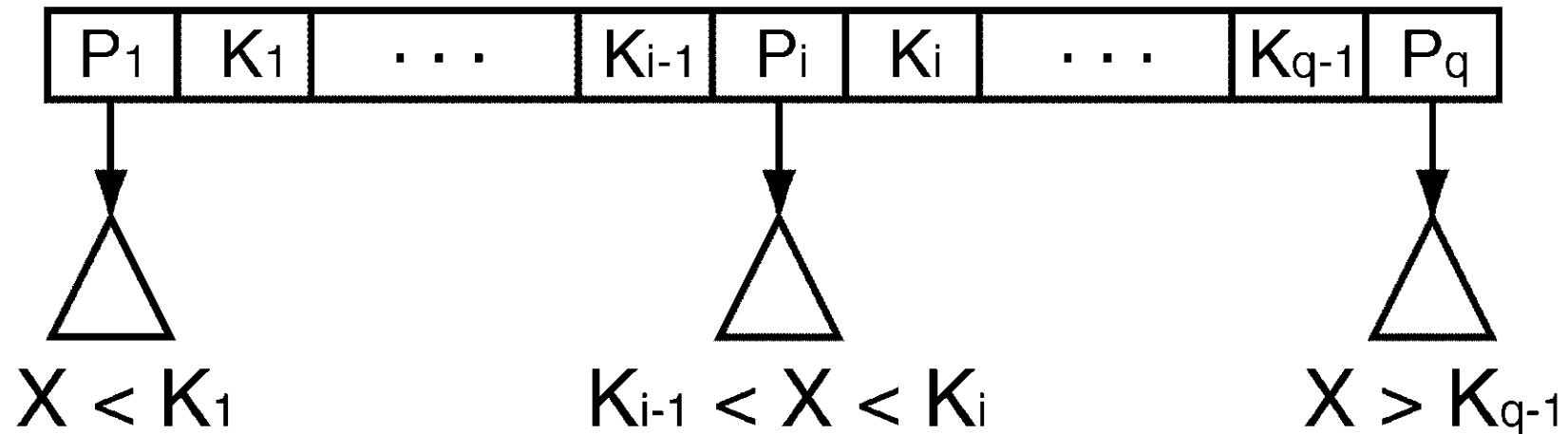


# Search Trees

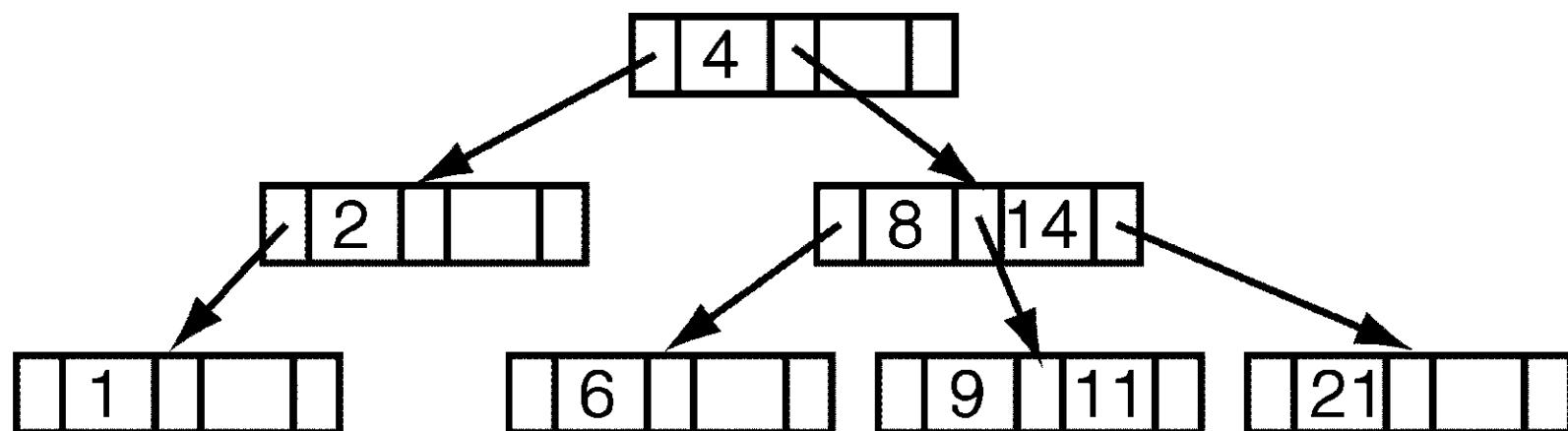
---

- Search trees
  - \* A search tree of order  $p$  is a tree with each node
    - » Contains at most  $p-1$  search values
    - »  $p$  pointers
    - » All search values are assumed to be unique
  - \* Each value in the tree also has a data pointer
    - » Points to the disk block that contains the record
  - \* Search tree itself can be stored on disk
    - » Assign each tree node to a disk block
    - » Order  $p$  is determined by the disk block size and the storage required for pointers and search values

## Search Trees (cont'd)



Data pointers are not shown



# Search Trees (cont'd)

---

- Problems with search trees
  - \* Search trees are not balanced
    - » All leaf nodes are not at the same level
      - Keeping the tree balanced is important
        - Guarantees that no nodes will be at very high levels
        - Limits the number of disk accesses required to search
    - \* Record deletions may leave some nodes in the tree nearly empty
      - » Results in performance deterioration
        - Wastage of storage space
        - Increase in number of levels
          - More disk accesses during search

# B-Tree

---

- B-tree (balanced tree) is a search tree with some additional constraints
  - » Solves the two problems to some extent
- Additional constraints ensure that
  - \* Tree is *always balanced*
  - \* Storage space wasted by deletions never becomes excessive
- Three variations
  - » B-tree
  - » B<sup>+</sup>-tree
  - » B<sup>\*</sup>-tree

# B-Tree (cont'd)

---

- A B-tree of order p is defined as follows:
  - \* Each internal node is of the form
$$\langle P_1, \langle k_1, P_{r1} \rangle, P_2, \langle k_2, P_{r2} \rangle, \dots, \langle P_{q-1}, \langle k_{q-1}, P_{rq-1} \rangle, P_q \rangle$$
where  $q \leq p$ 
    - $P_i$  is a tree pointer
    - $P_{ri}$  is a data pointer
  - \* Within each node
$$k_1 < k_2 < \dots < k_{q-1}$$
  - \* For all values  $x$  in the subtree of  $P_i$ 
$$k_{i-1} < x < k_i \text{ for } 1 < i < q$$
$$x < k_i \text{ for } i = 1$$
$$k_{i-1} < x \text{ for } i = q$$

# B-Tree (cont'd)

---

- \* Each node has at most  $p$  tree pointers
- \* Each node, except the root and leaf nodes, has at least  $\lceil p/2 \rceil$  tree pointers
  - » The root node has at least two pointers unless it is the only node
- \* A node with  $q$  tree pointers,  $q \leq p$ , has  $q-1$  search field values (and  $q-1$  data pointers)
- \* All leaf nodes are at the same level
  - » Leaf nodes have the same structure as internal nodes except that all their tree pointers are NULL.

## B-Tree (cont'd)

---

- Typically each node is stored in a disk block
- Order B-tree depends on
  - » Disk block size
  - » Search field size
  - » Pointer size

Example: Find the B-tree order for the following parameter values

Search field  $V = 9$  bytes

disk block size  $B = 1024$  bytes

Block pointer size  $P = 6$  bytes

# B-Tree (cont'd)

---

- \* An order  $p$  B-tree contains
  - »  $p$  block pointers (for tree pointers)
  - »  $p-1$  search field values
  - »  $p-1$  block pointers (for data pointers)

- \* We have

$$p * p + (p-1) * p + (p-1) * v \leq B$$

$$p * p + (p-1) * (p+v) \leq B$$

- \* In this example,

$$6p + 15(p-1) \leq 1024$$

$$21p \leq 1039$$

$$p \leq 49.48$$

We can build an order 49 B-tree

---

# B-Tree (cont'd)

---

- Searching
  - \* Follow appropriate tree pointer until
    - » the desired search value is found (“successful” case), or
    - » a NULL pointer is encountered (“failure” case)
  - \* A successful search costs at most  $(h+1)$  disk accesses in a level  $h$  B-tree
    - » The additional one is to access the actual data block
    - »  $h$  is  $O(\log_{\lceil p/2 \rceil} N)$  where  $N$  is the number of records
  - \* Unsuccessful search always requires  $h$  disk block accesses

# B-Tree (cont'd)

---

- Insertions
  - \* Place the search key value to be inserted in an appropriate leaf node
  - \* If that leaf node is full
    - » Split the node into two nodes at the same level
    - » Move the middle entry (i.e.  $\lceil p/2 \rceil$  entry) into the parent node
  - \* Splitting can propagate all the way to the root node
    - » This creates a new level
    - » Tree height increases by one whenever the root node splits
    - » This is the dynamic nature of the multilevel index represented by the B-tree

# B-Tree (cont'd)

---

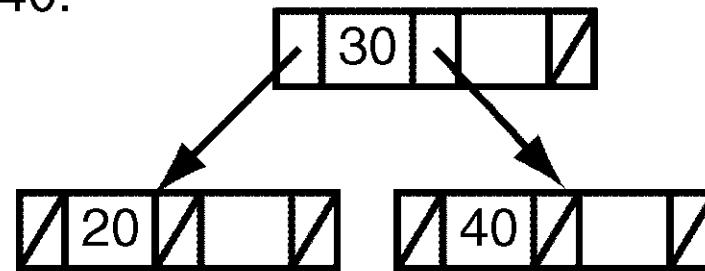
Example: Insert 20, 30, 40, 50, 60, 70, and 80 in this order into an order

$p=3$  B-tree

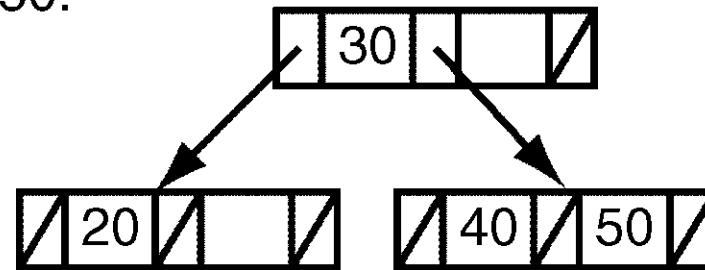
Insert 20, 30:



Insert 40:



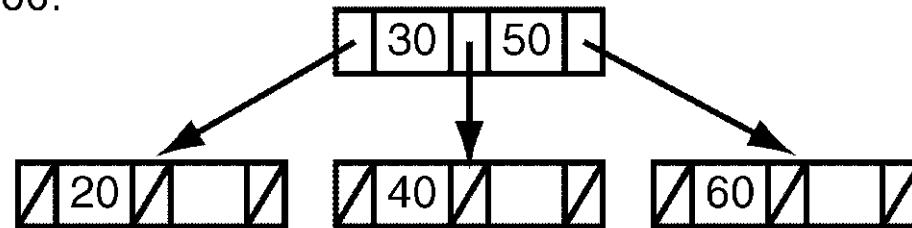
Insert 50:



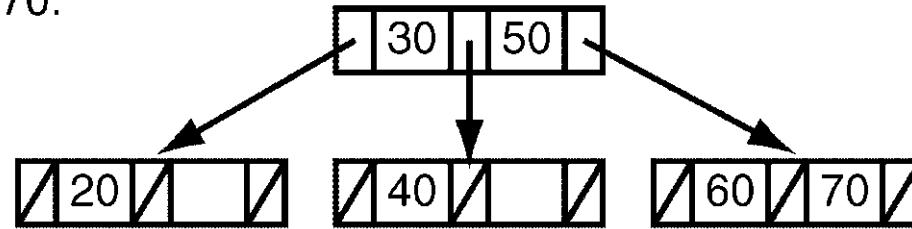
# B-Tree (cont'd)

---

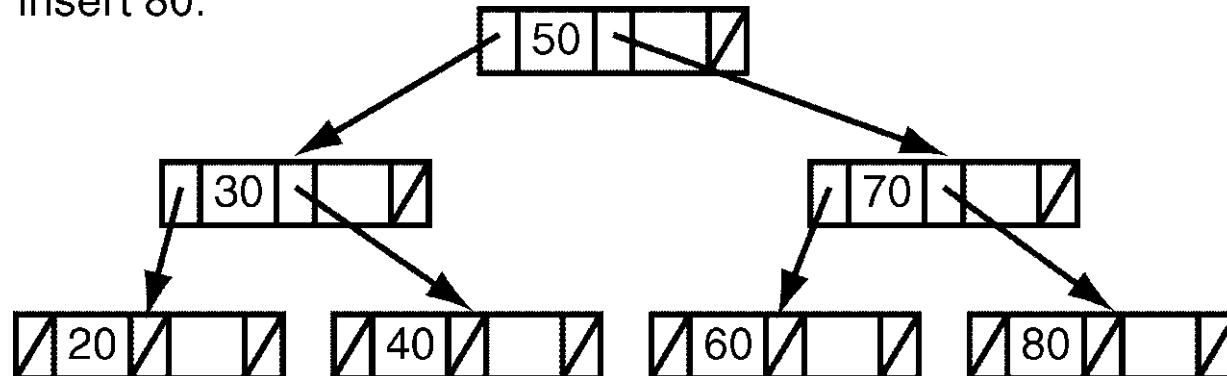
Insert 60:



Insert 70:



Insert 80:



# B-Tree (cont'd)

---

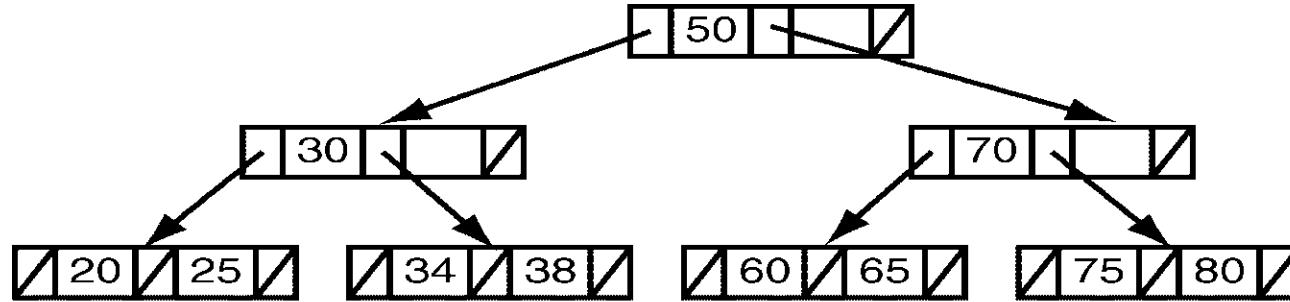
- Deletions
  - » The value to be deleted could be in a non-leaf or leaf node
- \* Deleting a value from an internal node
  - » If the key to be deleted is in an internal node
    - Its immediate predecessor or successor under the natural order of the keys is guaranteed to be in a leaf node
  - » Promote this predecessor/successor value from the leaf node into the place of the value to be deleted
  - » Delete the value promoted in the leaf node
    - Apply the leaf-node deletion procedure

# B-Tree (cont'd)

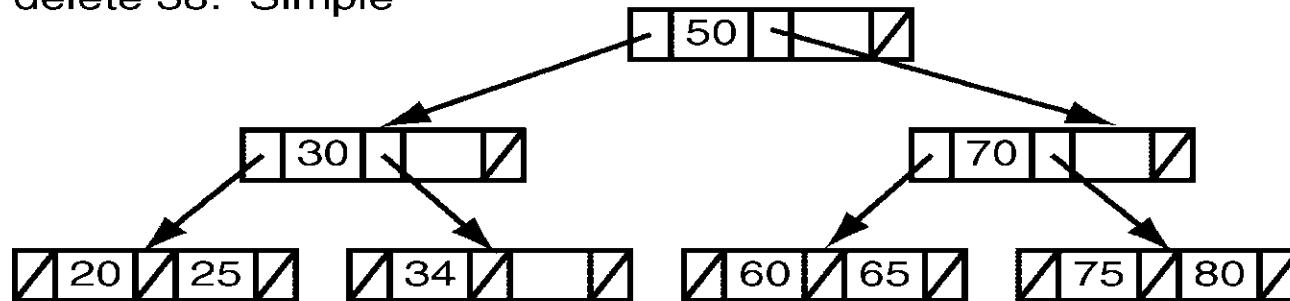
---

- \* Deleting a value from the leaf node
  - » Delete the value from the leaf node
  - » If the node contains more than the minimum number of keys ( $\lceil p/2 \rceil - 1$  keys) we are done
  - » Otherwise, look at the one/two leaf nodes that are adjacent and children of the same parent node
    - If one of them has more than *minimum number* of keys
    - Move an appropriate key from this node to its parent and move the parent key value into the original node
    - If not, combine two leaf nodes and the key from parent to form a single new leaf node
  - » Delete the key from the parent node (recursive step)

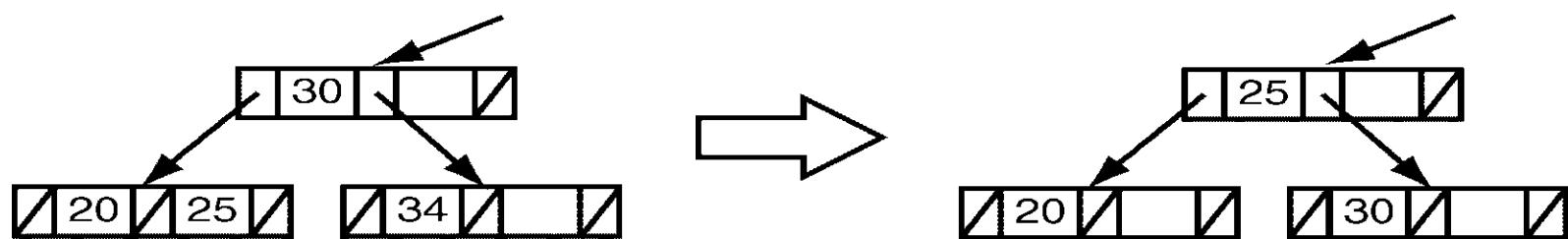
# B-Tree (cont'd)



delete 38: Simple

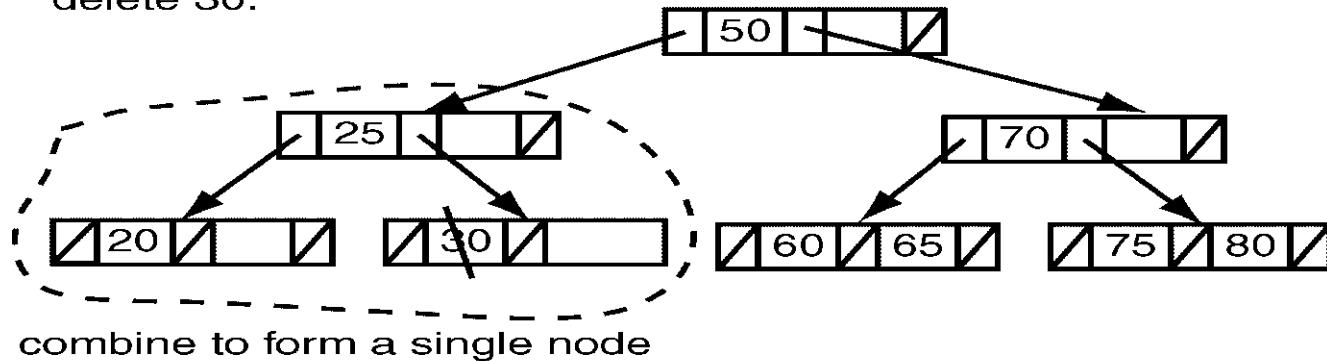


delete 34: Left neighbour is full, we can locally redistribute

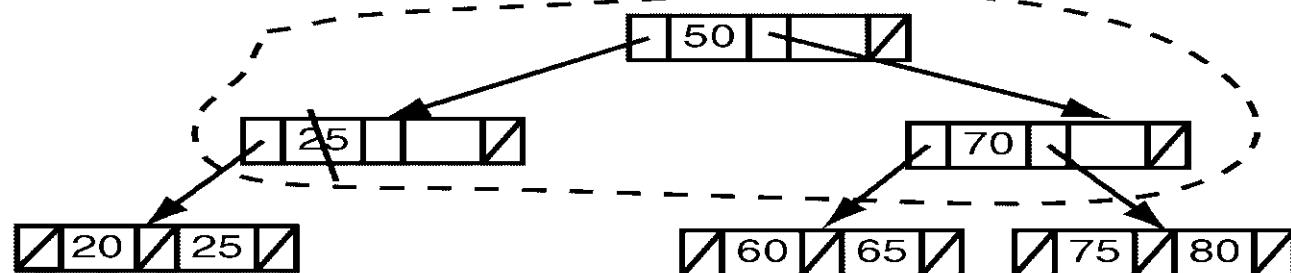


# B-Tree (cont'd)

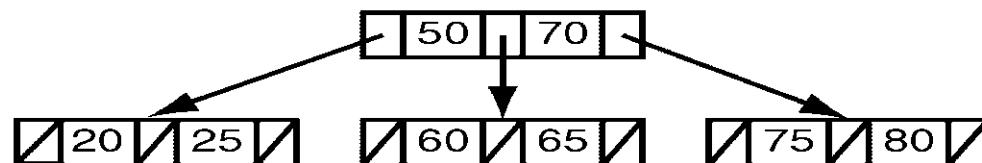
delete 30:



combine to form a single node

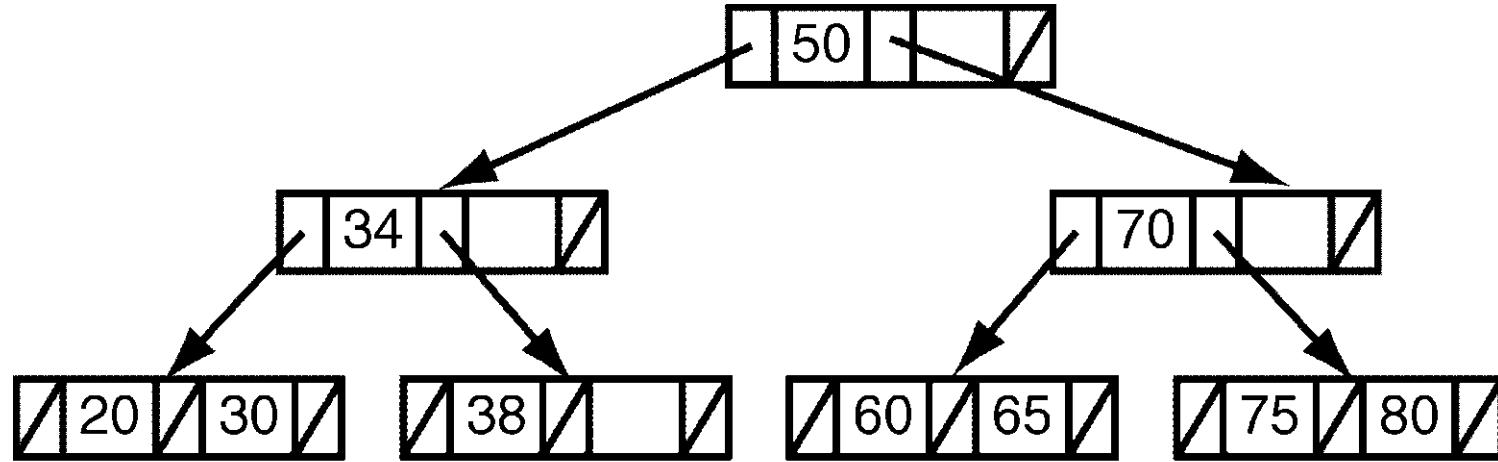


Final tree after deleting 30

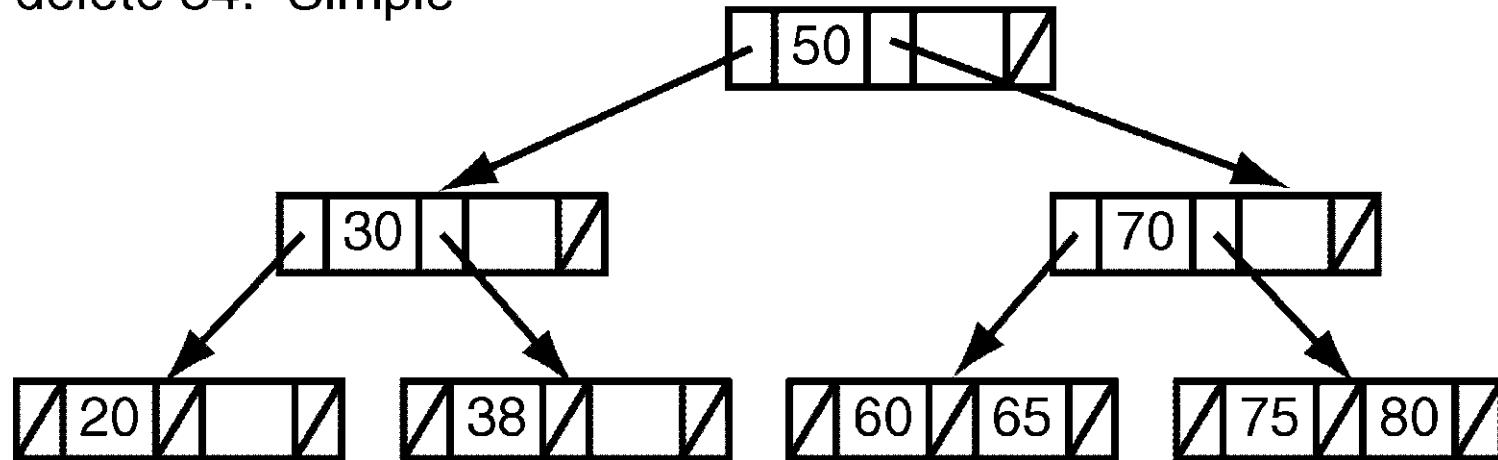


## B-Tree (cont'd)

---

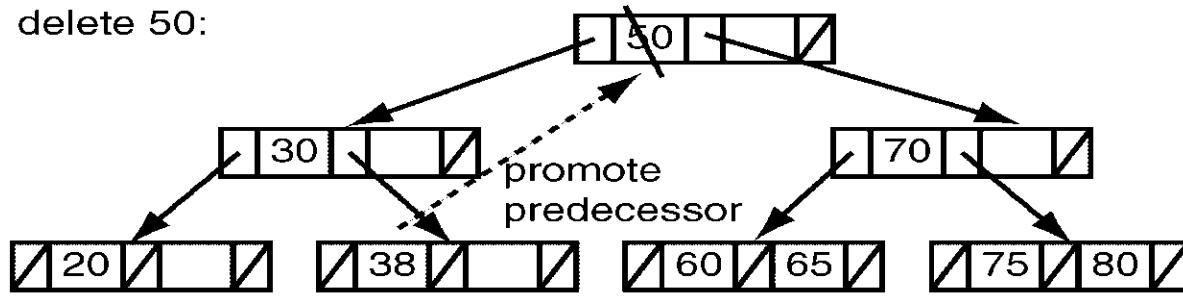


delete 34: Simple

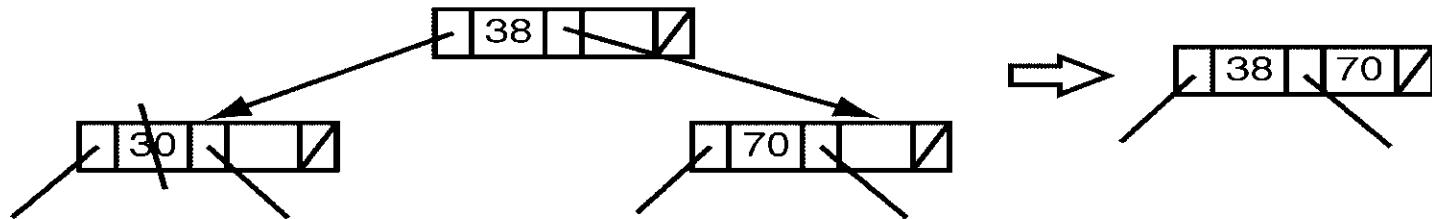
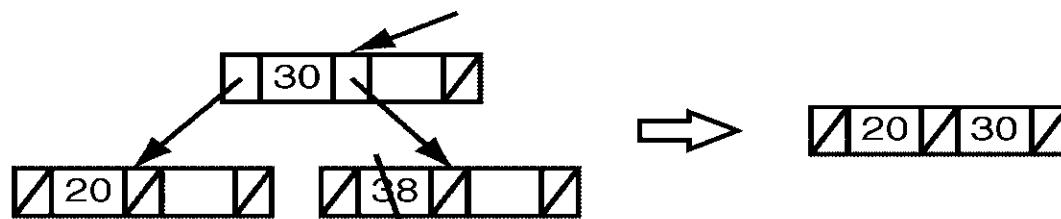


# B-Tree (cont'd)

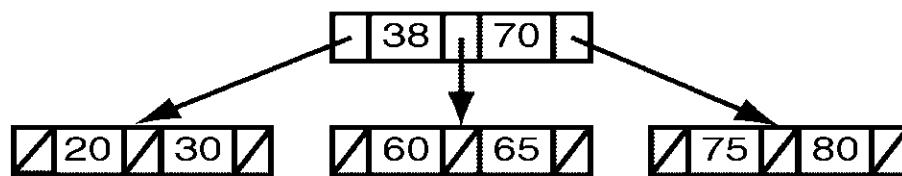
delete 50:



promote  
predecessor



Final tree after deleting 50



# B<sup>+</sup>-Tree

---

- Problems with B-trees
  - \* B-tree does not efficiently support sequential search
    - » Requires in-order traversal of the tree
      - Requires accessing several index blocks
  - \* B<sup>+</sup>-tree supports both indexed and sequential access efficiently
    - » In B<sup>+</sup>-tree, data pointers are stored only at the leaf nodes
      - Leaf node structure is different from that of internal node
      - Leaf nodes carry only data pointers (no tree pointers)
        - For non-key fields, the pointer can be to a block of actual data pointers
      - Internal node structure is similar to B-tree except no data pointers are stored

# B<sup>+</sup>-Tree (cont'd)

---

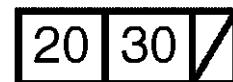
- Insertions
  - \* Similar to B-tree except
    - » When splitting a leaf node, we *copy* the middle value to the parent node (rather than moving it to the parent node)
  - \* Splitting an internal node is similar to B-tree
- Deletions
  - \* Similar to B-tree except when deleting from a leaf node
  - \* Splitting a leaf node
    - » The example and class discussion should clarify

# B<sup>+</sup>-Tree (cont'd)

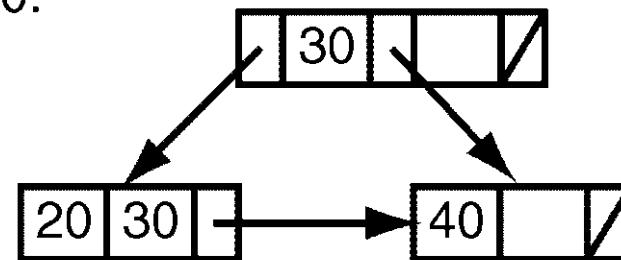
Example: Insert 20, 30, 40, 50, 60, 70, and 80 in this order into an order

p=3 B<sup>+</sup>-tree

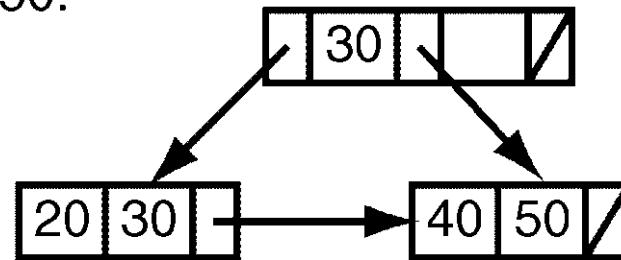
Insert 20, 30:



Insert 40:

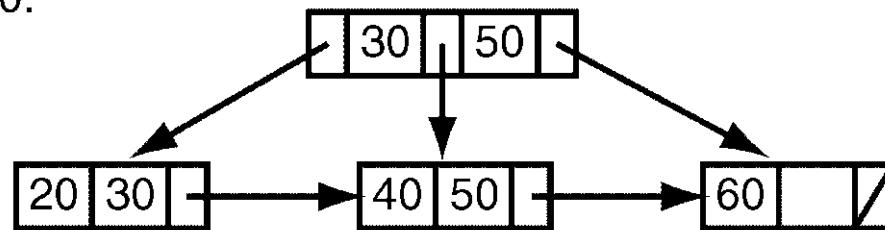


Insert 50:

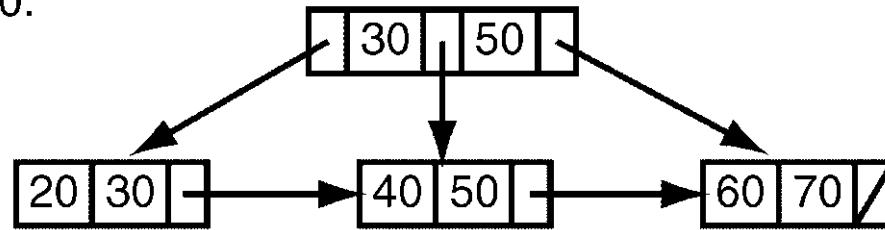


# B<sup>+</sup>-Tree (cont'd)

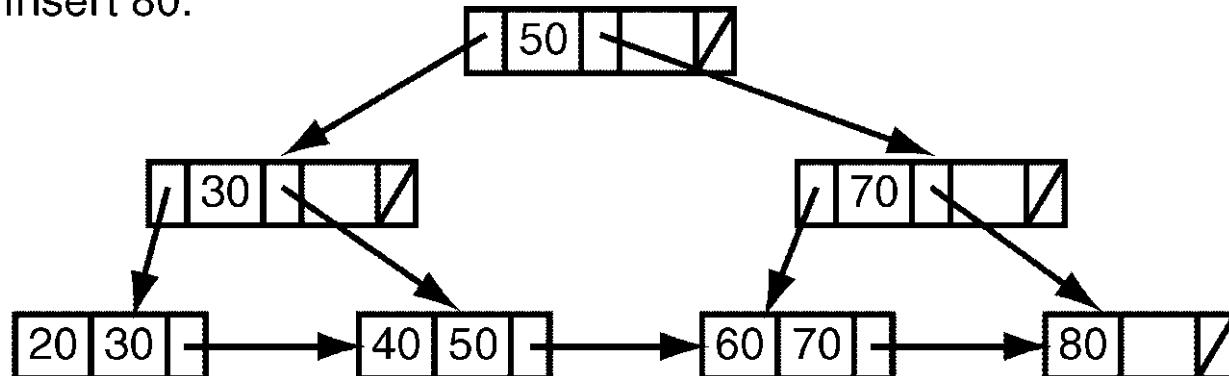
Insert 60:



Insert 70:

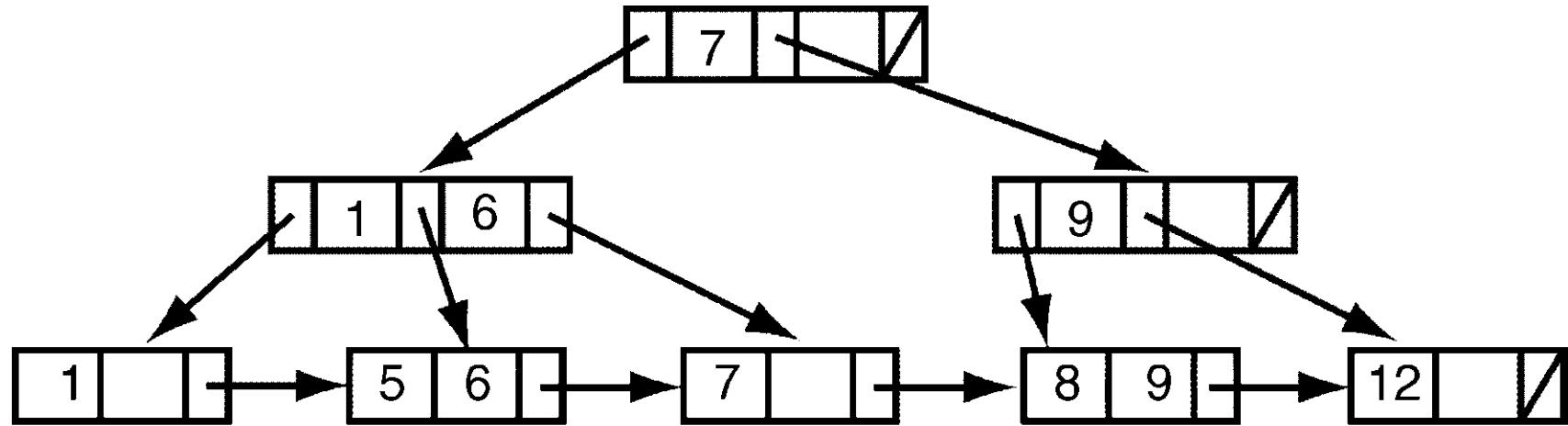


Insert 80:

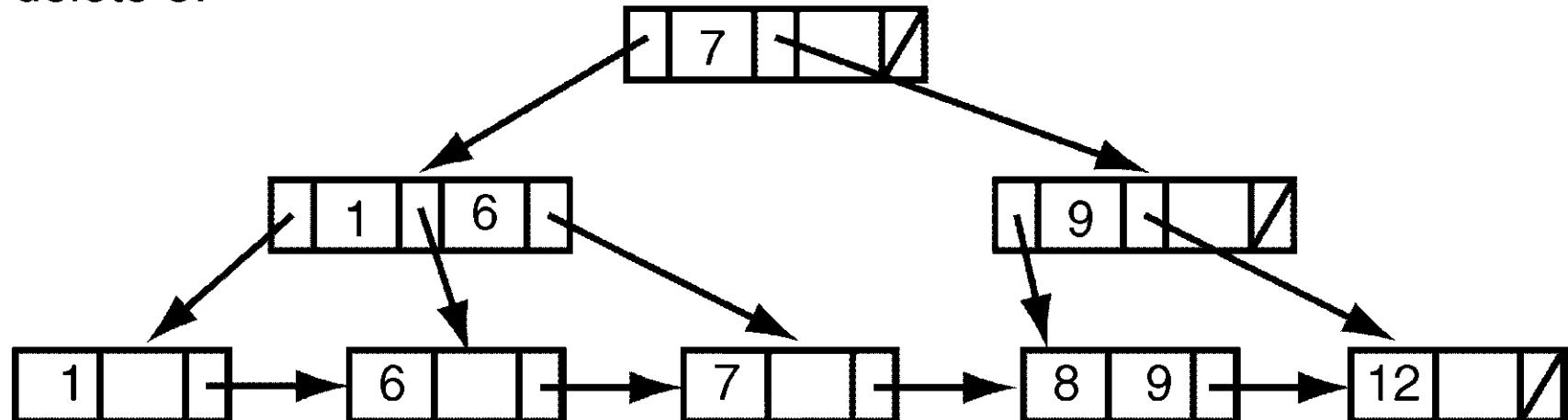


## B<sup>+</sup>-Tree (cont'd)

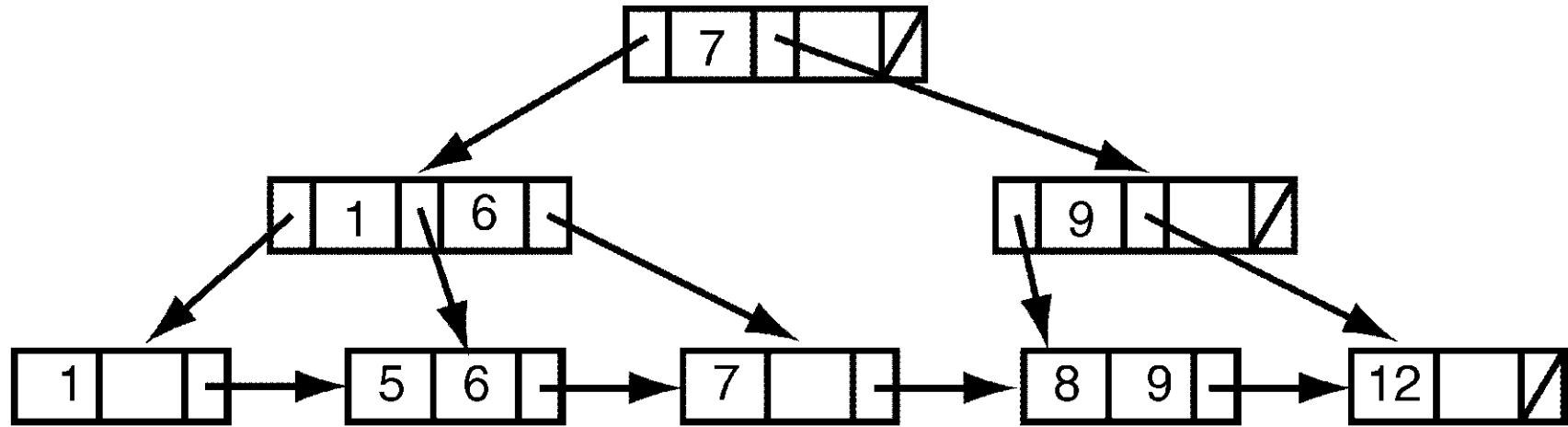
---



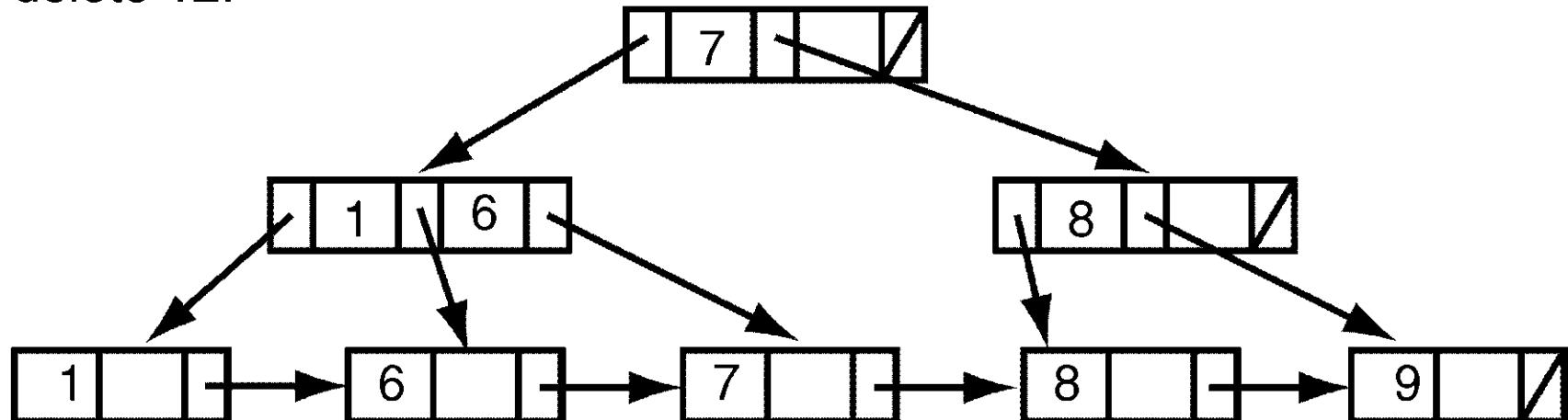
delete 5:



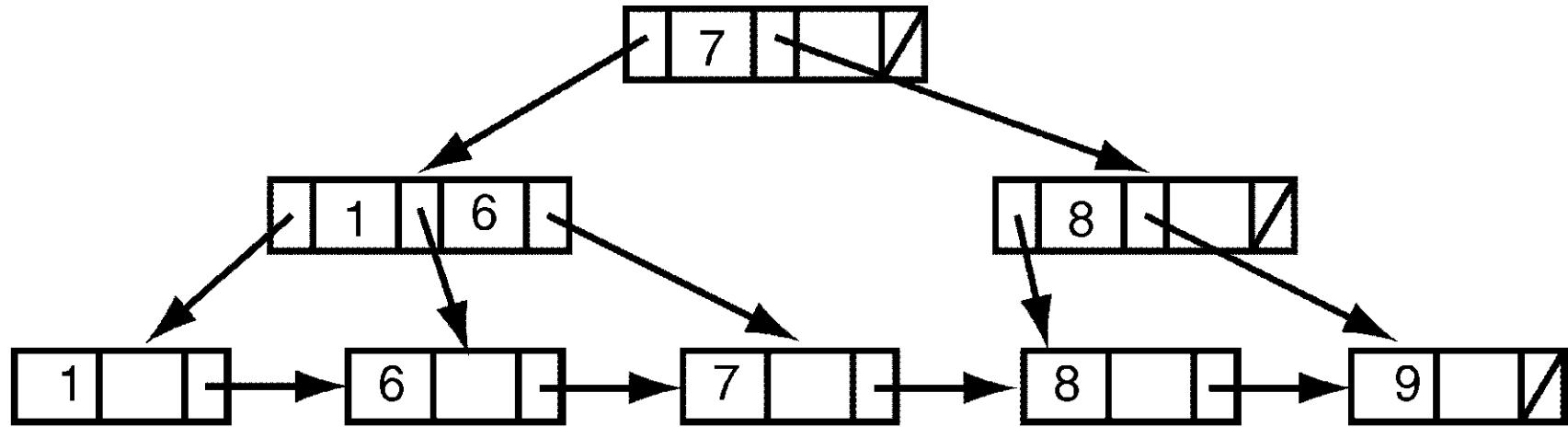
## B<sup>+</sup>-Tree (cont'd)



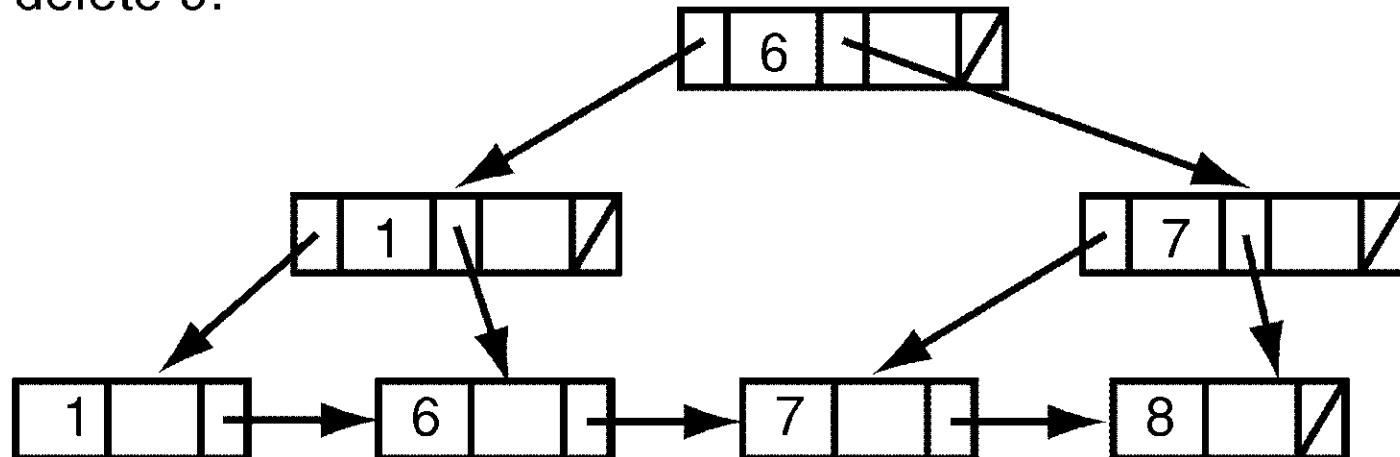
delete 12:



## B<sup>+</sup>-Tree (cont'd)

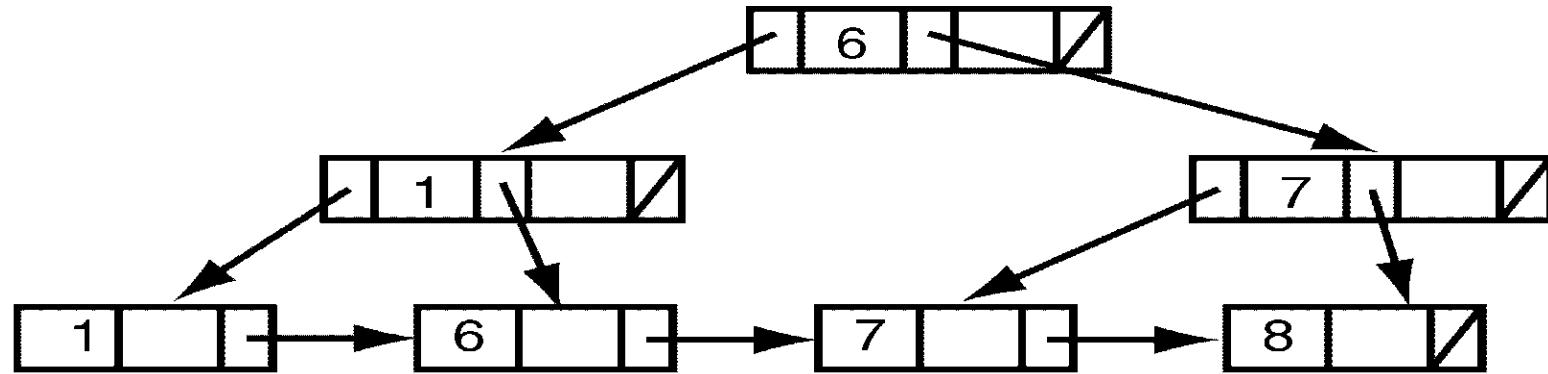


delete 9:

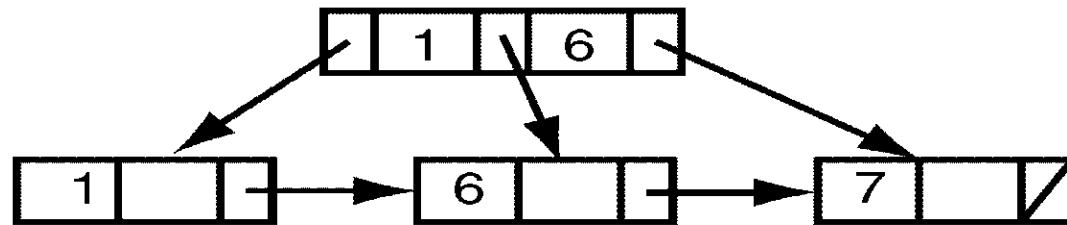


## B<sup>+</sup>-Tree (cont'd)

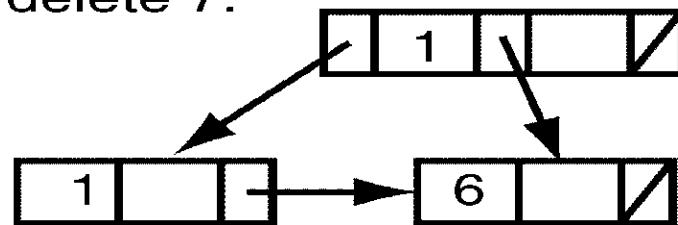
---



**delete 8:**



**delete 7:**

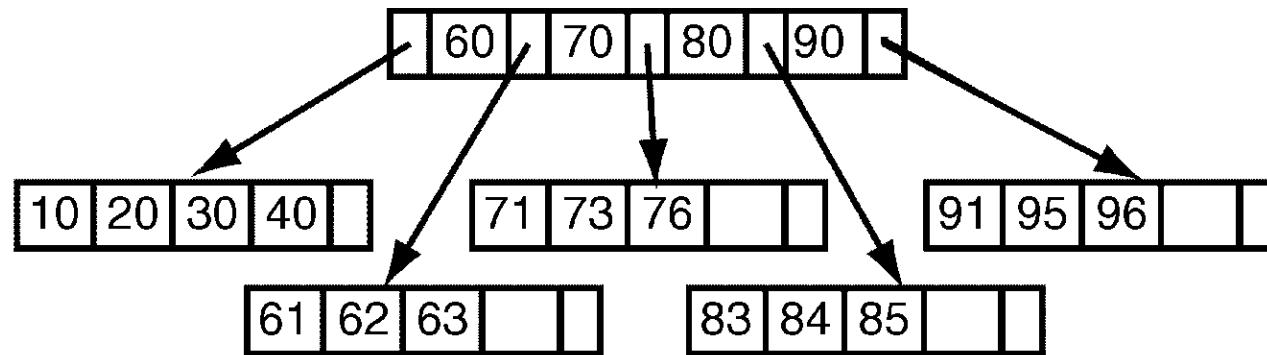


# B\*-Tree

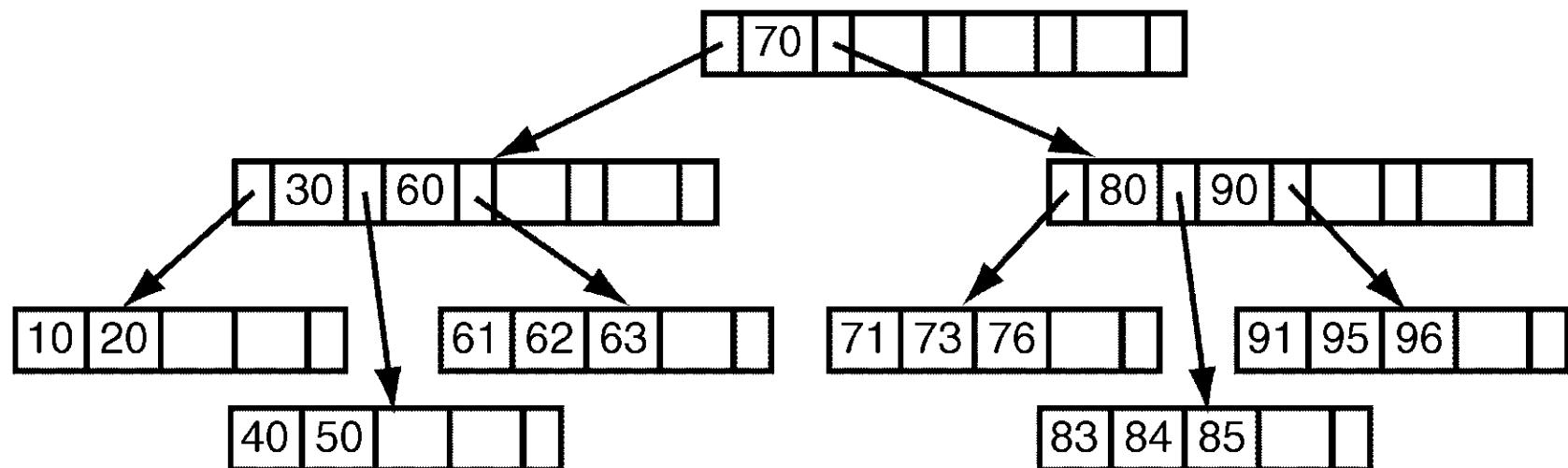
---

- In B-tree
  - \* Each node should be at least *half full*
- In B\*-tree
  - \* Each node should be at least  $2/3$  *full*
    - » Achieved by locally distributing values
- B\*-tree improves
  - » Storage efficiency
  - » Reduces the tree height
- Problem
  - » Creates too many node splits and merges with insertions and deletions

## B\*-Tree (cont'd)

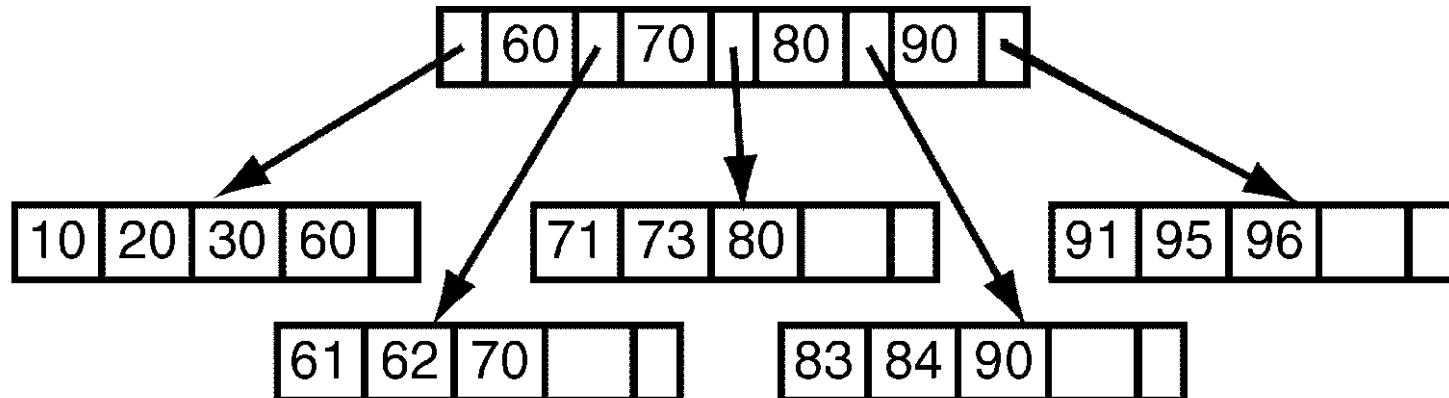


insert 50: B-Tree



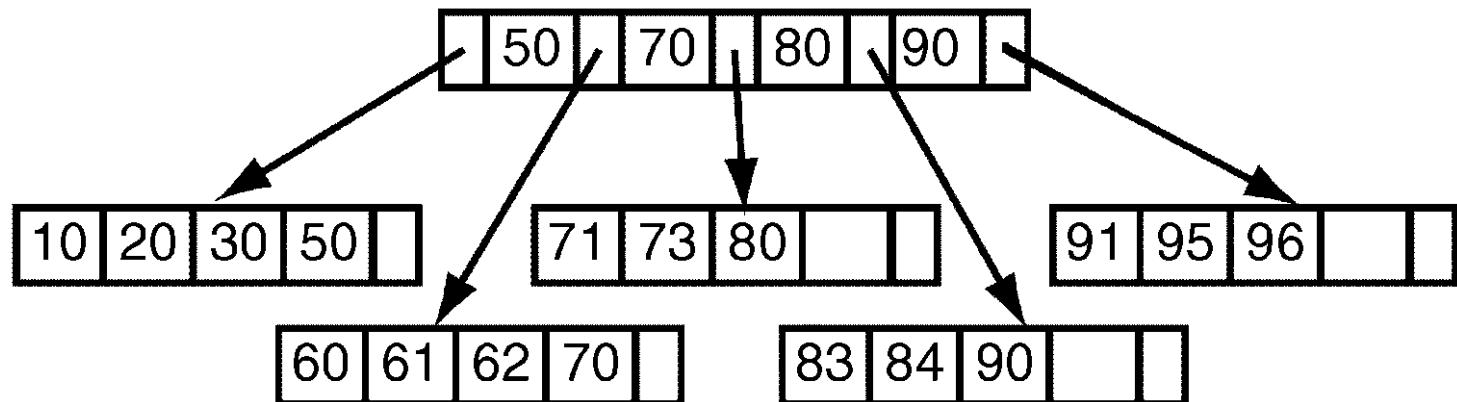
## B\*-Tree (cont'd)

---



Leaf node links are not shown for clarity

insert 50: B\*-Tree



## B\*-Tree (cont'd)

---

- ORACLE uses a more flexible version of B\*-tree as the default indexing structure
  - \* Using the CREATE INDEX statement, you can specify the amount of free space kept for new entries
    - » This is done by PCTFREE clause
- One more difference from our B\*-tree
  - \* ORACLE implementation uses doubly linked list for the leaf nodes
    - » Sequential search can be done forward or backward
      - Comes in handy in queries like

**WHERE CourseNo < 94234**

# B\*-Trees versus Hashing

---

- B\*-trees
  - \* Provide indexed as well as sequential access
    - » Sequential access is important for range queries
    - » Example:

```
SELECT StudentNo, StudentName
      FROM student
      WHERE StudentNo > 12345
```
- Hashing
  - \* Provides indexed access
    - » Does not provide sequential access

# B\*-Trees versus Hashing

---

- Use B\*-tree type of index
  - \* Default indexing scheme
    - » Unless you know in advance that range queries are infrequent
      - In this case, you can use hashing
- Use hashing
  - » Data is accessed primarily through exact match type queries
  - » Range queries and full table scans are infrequent
  - » Table size is static
    - Be prepared for the overhead for periodic reorganization
  - \* Hashing is useful for temporary files created during query processing
    - » Typically accessed using a key value

# Indexing in Oracle

---

- Oracle supports index
  - \* Creation
  - \* Modification
  - \* Deletion
- To create index, use

```
CREATE INDEX <index-name> ON  
          <table-name> (<col-name1, col-name2, ...)
```

- \* Two parameters (applicable to table storage)

**pctfree** – percent free in a block

**pctused** – percent used in a block

can also be specified

# Indexing in Oracle (cont'd)

---

## **pctfree**

- » Controls the amount of free space reserved in a block for updates that increase the row length
- » When there is only **pctfree**% free space is left, no more rows will be inserted into that block
- » Default is 10% (minimum 0%)

## **pctused**

- » Determines the point where a block that has reached **pctfree** will become re-eligible for inserts when deletes reduce the number of rows in the block
- » When the block is only **pctused**% full, new rows can be inserted into the block
- » Default is 40%

# Indexing in Oracle (cont'd)

---

- NULL entries are not indexed
- Possible to create concatenated indexes
  - \* For example, we can create a multicol column index on  
**CourseNo, ProfName, StudentNo**
    - » We can use the index to search based on any of the following:
      - **CourseNo, ProfName, StudentNo**
      - **CourseNo, ProfName**
      - **CourseNo**
    - » Only **CourseNo, StudentNo** cannot be used because  
**StudentNo** comes after **ProfName**, which is not specified
    - » Index is not useful in the following cases
      - **ProfName, StudentNo**

# Indexing in Oracle (cont'd)

---

- To remove an index

**DROP INDEX <index-name>**

- You can also modify an index using **ALTER INDEX**

» For example, you may want to change index from  
**CourseNo , ProfName , StudentNo** to **CourseNo , Profname**

» You can create this new index by modifying the existing one rather than creating a new one

- You can use **REBUILD** clause to change the storage characteristics of an index

# Summary

---

- Physical storage
  - » Disk characteristics, Disk parameters, RAID
- File organizations
  - » Unordered (heap) and ordered sequential files
- Hashing (Static and Dynamic)
- Index structures
  - \* Single-level indexes
    - » Primary, clustering, secondary
  - \* Multilevel indexes
    - » Static and dynamic (B-trees, B<sup>+</sup>-trees, and B\*-trees)

## ② Database System Basics:

- A database system is a computerized way to manage records, allowing for data creation, storage, retrieval, updating, and deletion.
- Consists of data, hardware, software, and user interfaces to facilitate these tasks.

## ③ Database Concepts:

- A database is a structured collection of persistent data used in applications, reflecting true propositions (like "Supplier S1 is in London").
- Data can be static (e.g., SIN), dynamic (e.g., account balance), or quasi-static (e.g., salary).

## ④ Why Use a Database?:

- Shared data access, reduced redundancy, consistency, transaction support (ensuring complete or no updates in data operations), data integrity, security, standardization, and balancing enterprise needs over individual requirements.
- Example: Inconsistent updates can lead to integrity issues, like updating one file but not its redundant counterparts.

## ⑤ Data Independence:

- Separates data from application code, allowing data to change without affecting applications and vice versa.
- Two layers: logical (user view) and physical (storage), with DBMS shielding users from hardware details.

## ⑥ Relational Systems:

- Introduced in 1969-70, they store data in tables and rely on relational algebra for processing.
- Use non-pointer-based logic, representing data solely as tables and allowing for SQL support.
- Examples include IBM's DB2, Microsoft SQL Server, and Oracle.

## ⑦ Non-Relational Systems:

- Other models include hierarchical, network, object-oriented, and multidimensional databases, each tailored to different data structures and applications.

## ⑧ DBMS Software Components:

- A DBMS provides a bridge between physical data storage and user access.
- It includes software like database managers and servers, excluding tools such as application development software and report writers.

## ⑨ Types of Users:

- Users include application programmers, end users, and database administrators (DBA), who implement technical controls based on policies by data administrators (DAs).

## ⑩ Applications Needing DBMS:

- Common in systems that handle complex data relationships and transactions, such as library and financial systems.

## **Database Architecture:**

- **Three-Level Architecture (ANSI/SPARC):**

- **Internal Level:** Closest to physical storage, detailing data storage and access paths.
- **Conceptual Level:** Represents the entire database, defining structure without focusing on physical storage.
- **External Level:** Provides tailored views of the database for different users, ensuring a user-oriented perspective.

## **Schemas and Instances:**

- **Schema:** Defines the database's structure (e.g., tables and relationships) and changes infrequently.
- **Instance:** Refers to the data in the database at a specific point in time.

## **Data Independence:**

- **Logical Data Independence:** Allows changes to the conceptual schema without altering external views or applications.
- **Physical Data Independence:** Enables changes to the internal schema without affecting the conceptual schema.

## **Database Administrator (DBA):**

- Responsible for conceptual design, user support, implementing security, and database performance.

## **DBMS Languages:**

- **Data Definition Language (DDL):** Used to define and modify schemas.
- **Data Manipulation Language (DML):** Used for data retrieval, insertion, updating, and deletion by end users.

## **System Processes Support:**

- Includes data communication interfaces, client-server architecture, external tools for queries and reports, and support for distributed processing.

## ② Designing the Conceptual Schema:

- **Stages:**

- **Model Selection:** Choose a model (e.g., ER model) to capture user requirements.
- **Normalization:** Refines the schema to eliminate redundancy.
- **Optimization:** Produces the final database structure and data dictionary.

## ③ ER Model Basics:

- **Entities:** Represent objects (e.g., Employee). Types include strong (independent) and weak (dependent on another entity).
- **Relationships:** Define connections between entities with cardinality constraints (one-to-one, one-to-many, many-to-many).
- **Attributes:** Describe entity properties and include keys, simple/composite, single/multi-valued, and stored/derived.

## ④ Converting ER Diagrams to Relational Schema:

- **Strong Entities:** Each entity type becomes a table with attributes.
- **Relationships:** Relationship types create tables linking primary keys of related entities.
- **Weak Entities:** Rely on strong entities for identification, combining primary keys of the strong entity and unique attributes of the weak entity.

## ⑤ Advanced Modeling Techniques:

- **Specialization/Generalization:** Organizes entities into hierarchies, allowing subclasses (specialization) or superclasses (generalization).
- **Aggregation:** Represents relationships among relationships as higher-level entities.

## ⑥ Constraints:

- **Disjoint Constraint:**

- **Disjoint:** Entity only member of one subclass
- **Overlapping:** Entity member of more than one subclass

- **Completeness:**

- **Total:** Every entity in the superclass must be a member of some subclass in the specialization
- **Partial:** An entity may not belong to any of the subclasses in the specialization

Two methods for deriving relational schema from an ER diagram with specialization/generalization

Method 1

Create a table for the higher-level entity

For each lower-level entity, create a table which includes a column for each of its attributes plus for primary key of the higher-level entity

Method 2

Do not create a table for the higher-level entity

For each lower-level entity, create a table which includes a column for each of its attributes plus a column for each attribute of the higher-level entity

## Relational Model Basics:

- Data is organized in tables (relations), with rows as **tuples** and columns as **attributes**.
- The number of columns (attributes) is called the **degree** or **arity** of the relation, and each column has a **domain** specifying its allowable values.

## Ordering and Notation:

- Tuples (rows) in a relation theoretically have no inherent order, although they may be stored with order for practical reasons.
- Columns are labeled with names, making their order less important.

## Relation Schema and Notation:

- A **relation schema** defines the structure of a table. For example, `Student(StudentID, Name, Age, Major)` specifies the columns of a Student table.
- Specific notation is used for relation names (e.g., R, S) and tuples (e.g., t, u) to standardize how we reference database components.

## Keys and Superkeys:

- **Primary keys** uniquely identify records within a table, often with identifiers like `studentID` or `employeeID`.
- **Superkeys** may include additional attributes but still uniquely identify records.

## Relations vs. Relvars:

- **Relations** represent specific, static snapshots of data (like a read-only view at a given time).
- **Relvars** are the actual tables (relation variables) in a database that can change as data is inserted, updated, or deleted.

## Optimization:

- The **optimizer** in a relational system decides the best way to execute queries, using techniques like indexing or sequential search to retrieve data efficiently.

## Catalog:

- The catalog is a metadata repository that stores information about database objects (tables, columns, indexes, constraints, etc.).
- It helps the DBMS manage data structure, enforce integrity, and optimize access and security.

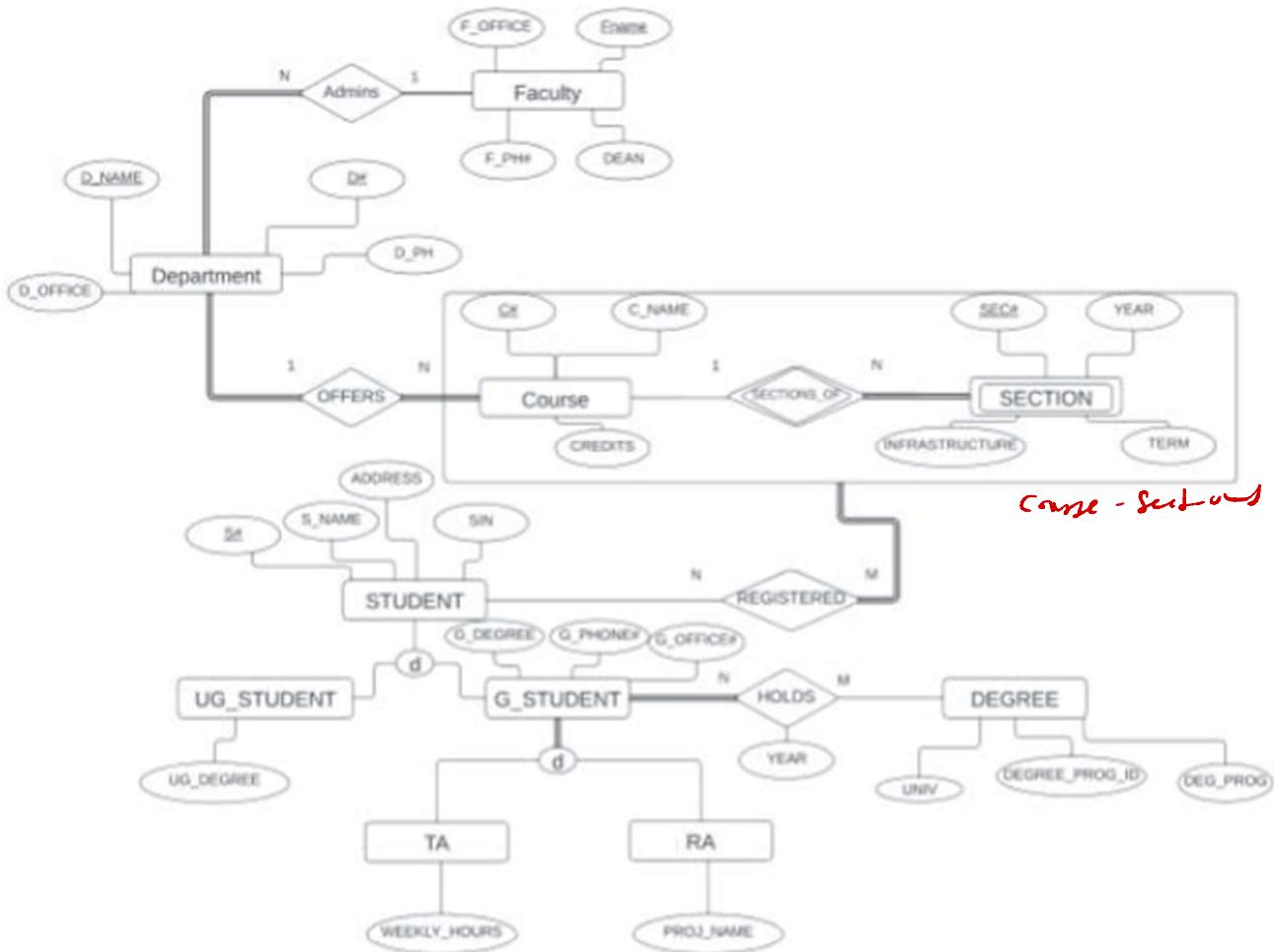
## Transactions:

- A **transaction** is a sequence of operations that represents a single unit of work. It is often used to group multiple database operations together to ensure data consistency.
- Properties of transactions include:
  - **Atomicity:** The transaction is all-or-nothing, meaning it either completes fully or not at all.
  - **Durability:** Once committed, the changes are permanently applied to the database.
  - **Isolation:** Transactions run independently, with changes hidden from others until committed.

- **Serializability:** The outcome of interleaved transactions is the same as if each transaction had executed one after the other.

## ② Commit and Rollback:

- **Commit:** A successful end-of-transaction operation that makes all changes permanent in the database. For instance, after a COMMIT, any updates made by the transaction are visible to other users and remain, even if the system crashes afterward.
- **Rollback:** An operation that undoes changes made by a transaction, effectively canceling it. If an error or failure occurs, a ROLLBACK reverts all database changes to the state before the transaction began, ensuring data integrity.



In evaluating the required ER Diagram, students should have accurately represented key database design elements: Specialization (Subclassing), Cardinality (Relationship Types), Participation (Entity Involvement), Weak/Strong Entities, Keys (Primary/Attributes Identifiers), Entities (Tables), and Relations (Entity Associations). Errors or omissions in these areas resulted in a 0.25 point deduction for each. For inaccuracies in identifying Weak/Strong entities the penalty is 0.5 point.

```

-- Department Table
CREATE TABLE Department (
    D_NAME VARCHAR(50) PRIMARY KEY,
    D_PH VARCHAR(15),
    D_OFFICE VARCHAR(20)
);

-- Faculty Table
CREATE TABLE Faculty (
    F_PH VARCHAR(15),
    F_OFFICE VARCHAR(20),
    Exams VARCHAR(50),
    Dean VARCHAR(50),
    D_NAME VARCHAR(50),
    PRIMARY KEY (F_PH),
    FOREIGN KEY (D_NAME) REFERENCES Department(D_NAME)
);

-- Course Table
CREATE TABLE Course (
    C_NAME VARCHAR(50) PRIMARY KEY,
    C_ID INT UNIQUE,
    CREDITS INT,
    D_NAME VARCHAR(50),
    FOREIGN KEY (D_NAME) REFERENCES Department(D_NAME)
);

-- Section Table
CREATE TABLE Section (
    SEC_ID INT PRIMARY KEY,
    YEAR INT,
    INFRASTRUCTURE VARCHAR(100),
    TERM VARCHAR(20),
    C_ID INT,
    FOREIGN KEY (C_ID) REFERENCES Course(C_ID)
);

-- Student Table
CREATE TABLE Student (
    S_ID INT PRIMARY KEY,
    S_NAME VARCHAR(50),
    SIN INT UNIQUE,
    ADDRESS VARCHAR(100),
    D_DEGREE VARCHAR(50)
);

-- UG_Student Table
CREATE TABLE UG_Student (
    S_ID INT PRIMARY KEY,
    UG_DEGREE VARCHAR(50),
    FOREIGN KEY (S_ID) REFERENCES Student(S_ID)
);

-- G_Student Table
CREATE TABLE G_Student (
    S_ID INT PRIMARY KEY,
    G_PHONE VARCHAR(15),
    G_OFFICE VARCHAR(20),
    FOREIGN KEY (S_ID) REFERENCES Student(S_ID)
);

-- Teaching Assistant (TA) Table
CREATE TABLE TA (

```

```

S_ID INT PRIMARY KEY,
WEEKLY_HOURS INT,
FOREIGN KEY (S_ID) REFERENCES G_Student(S_ID)
);

-- Research Assistant (RA) Table
CREATE TABLE RA (
S_ID INT PRIMARY KEY,
PROJ_NAME VARCHAR(50),
FOREIGN KEY (S_ID) REFERENCES G_Student(S_ID)
);

-- Degree Table
CREATE TABLE Degree (
DEG_PROG_ID INT PRIMARY KEY,
YEAR INT,
UNIV VARCHAR(50),
DEG_PROG VARCHAR(50)
);

-- Registered Table (Associative Entity between Student and Section)
CREATE TABLE Registered (
S_ID INT,
SEC_ID INT,
PRIMARY KEY (S_ID, SEC_ID),
FOREIGN KEY (S_ID) REFERENCES Student(S_ID),
FOREIGN KEY (SEC_ID) REFERENCES Section(SEC_ID)
);

-- Holds Table (Associative Entity between G_Student and Degree)
CREATE TABLE Holds (
S_ID INT,
DEG_PROG_ID INT,
PRIMARY KEY (S_ID, DEG_PROG_ID),
FOREIGN KEY (S_ID) REFERENCES G_Student(S_ID),
FOREIGN KEY (DEG_PROG_ID) REFERENCES Degree(DEG_PROG_ID)
);

```

## Explanation of Relationships

- Department:** Contains information about departments, including department name, phone number, and office.
- Faculty:** Linked to a department (`D_NAME`) and includes contact details like phone and office. The `Dean` and `Exams` fields capture other relevant info.
- Course:** Each course is related to a department and includes `CREDITS` and other details.
- Section:** A course can have multiple sections, each with details like `YEAR`, `INFRASTRUCTURE`, and `TERM`.
- Student:** Stores basic student details, which is the parent entity for undergraduate (`UG_Student`) and graduate (`G_Student`) students.
- UG\_Student and G\_Student:** Inherit from `Student`. Undergraduates have an `UG_DEGREE`, and graduates have `G_PHONE` and `G_OFFICE`.
- TA and RA:** Sub-entities of graduate students, representing teaching assistants and research assistants.
- Degree:** Stores degree information and relates to graduate students via the `Holds` table.
- Registered:** Connects students to sections, enabling a many-to-many relationship.