Bangladesh Govt. & UGC Approved

UNIVERSITY OF GLOBAL VILLAGE (UGV), BARISHAL.

THE FIRST SKILL BASED HI-TECH UNIVERSITY IN BANGLADESH

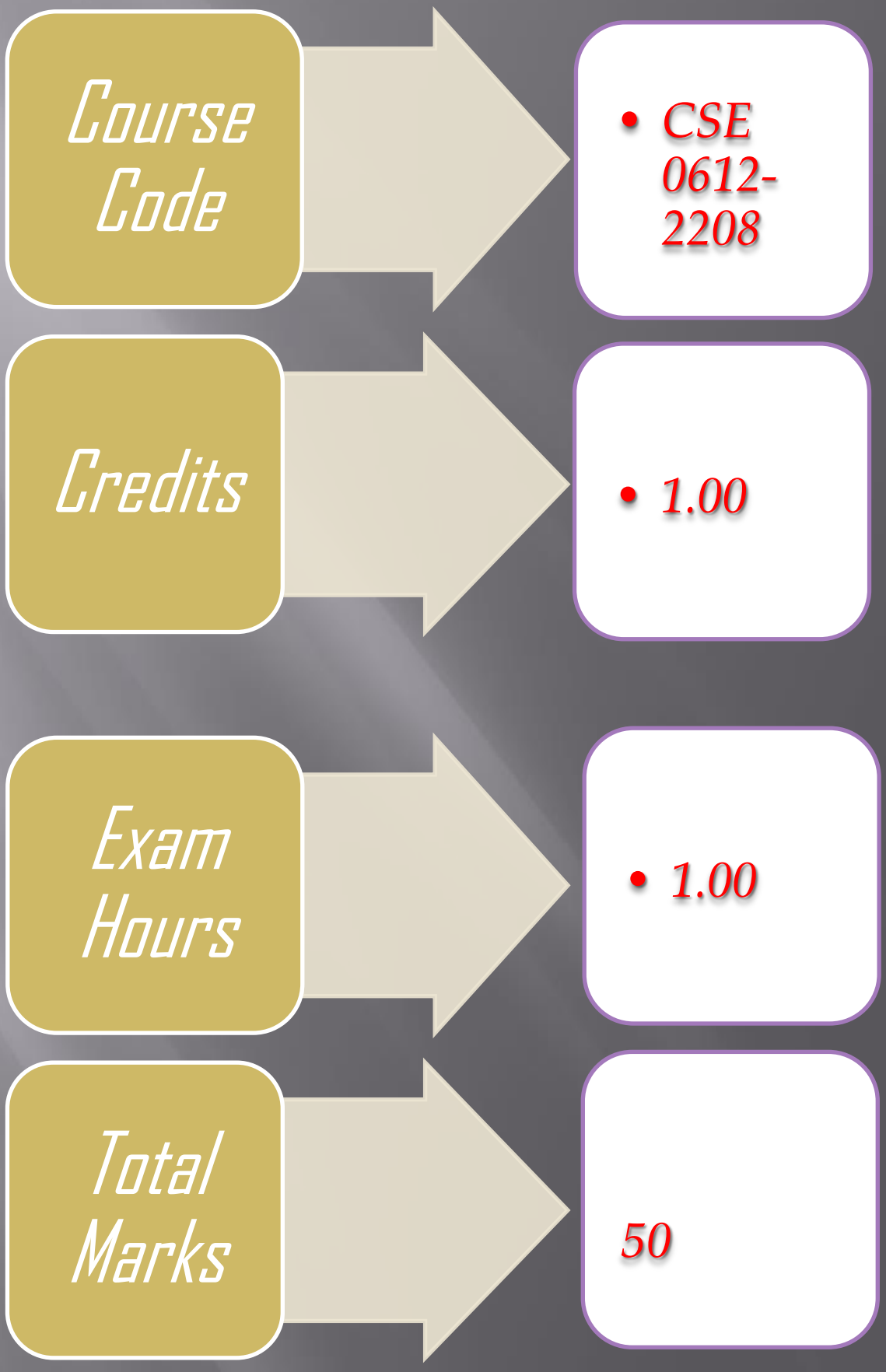
DBMS Sessional

Prepared by:

Md. Abdur Razzak

Asst. Professor.

Department of CSE


Course Code
CSE 0612-2208
Credits
1.00
Exam Hours
1.00
Total Marks
50

Index:

Sl. No	Name of The Sessional	Supervisor's Signature With Date	Instructor's Signature With Date
1	Introduction to SQL and SQL*Plus		
2	Entity-Relationship (ER) Diagrams		
3	Various Data Types		
4	Tables		
5	MySQL Installation		
6	DDL and DML Commands with Examples		
7	Key Constraints and Normalization		
8	Aggregate Functions		
9	Joins		
10	Views		
11	Index		
12	PL/SQL		
13	Exception Handling		
14	Triggers		
15	Cursors		
16	Subprograms (Procedures in PL/SQL)		
17	Functions in PL/SQL		

Week 1

Sessional -01

Introduction to SQL and SQL*Plus

Objective: To familiarize students with SQL*Plus, its interface, and basic commands.

Activities:

- Connect to the database using SQL*Plus.
- Learn and execute basic commands like `SELECT`, `INSERT`, `UPDATE`, and `DELETE`.
- Understand the significance of `DESC` to view table structures.

Task: Practice connecting to SQL*Plus and running basic SQL commands.

Practice Problems:

1. Create a table `Students` with fields `StudentID`, `Name`, and `Department`.
2. Insert sample data into the `Students` table.
3. Write a query to retrieve all student names from the table.

Instructor Signature (with date) **Supervisor's signature (with date)**

Why SQL?

- SQL is a high-level language.
  - Expresses "what to do" rather than "how to do it."
  - Avoid a lot of data-manipulation details needed in procedural languages like C++ or Java.

- Database management system figures out "best" way to execute query.
  - Called *query optimization*
- SQL is primarily a query language, for getting information from a database.
  - But SQL also includes a *data-definition* component for describing database schemas

Data Definition Language

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

- The schema for each relation.
- The domain of values associated with each attribute.
- Integrity constraints
- The set of *indices* to be maintained for each relations.
- Security and authorization information for each relation.
- The physical storage structure of each relation on disk. (Not covered in 354)

Creating (Declaring) a Relation

- Simplest form is:

```
CREATE TABLE <name> (
        <list of elements>
);
```

- To delete a relation:

```
DROP TABLE <name>;
```

Elements of Table Declarations

- Most basic element: an attribute and its type.
- The most common types are:
  - INT or INTEGER (synonyms).
  - REAL or FLOAT (synonyms).
  - CHAR(*n*) = fixed-length string of *n* characters.
  - VARCHAR(*n*) = variable-length string of up to *n* characters.

Declaring Keys

- An attribute or list of attributes may be declared PRIMARY KEY or UNIQUE.
- Either says that no two distinct tuples of the relation may agree in all the attribute(s) on the list.
- So keys provide a means of uniquely identifying tuples.
- There can be only one PRIMARY KEY for a relation, but possibly several UNIQUE lists of attributes.
- No attribute of a PRIMARY KEY can ever be NULL. (Why?)

Declaring Single-Attribute Keys

- Can place PRIMARY KEY or UNIQUE after the type in the declaration of the attribute.
- Example: Declare *branch_name* as the primary key for a bank's *branch*

```
CREATE TABLE branch (
                    branch_name     CHAR (15)
PRIMARY KEY,
                    branch_city     CHAR (30),
                    assets          INTEGER
);
```

Week 2

Sessional -02

Entity-Relationship (ER) Diagrams

Objective: To model real-world problems into database diagrams.

Activities:

- Design an ER diagram for a library management system with entities like `Book`, `Member`, and `Borrow`.
- Identify primary keys and relationships for the entities.

Task: Create an ER diagram for an online shopping system.

Practice Problems:

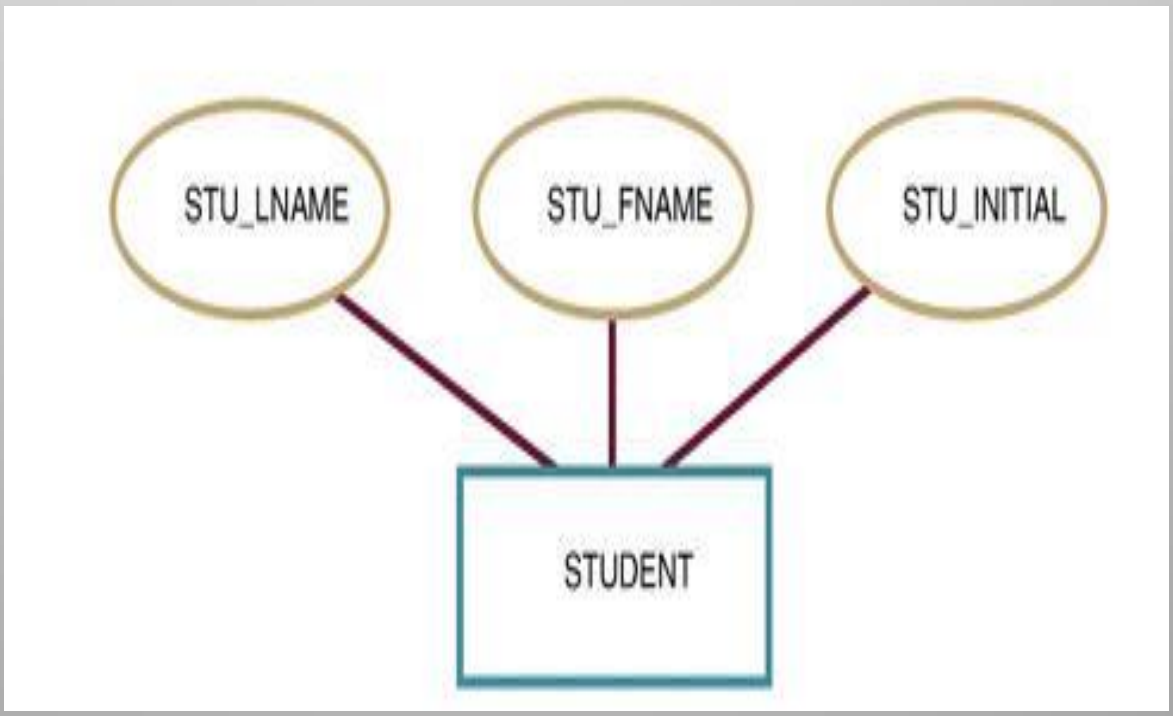
1. Identify entities and attributes for a hospital management system.
2. Draw an ER diagram for a university registration system.

Instructor Signature (with date) **Supervisor's signature (with date)**

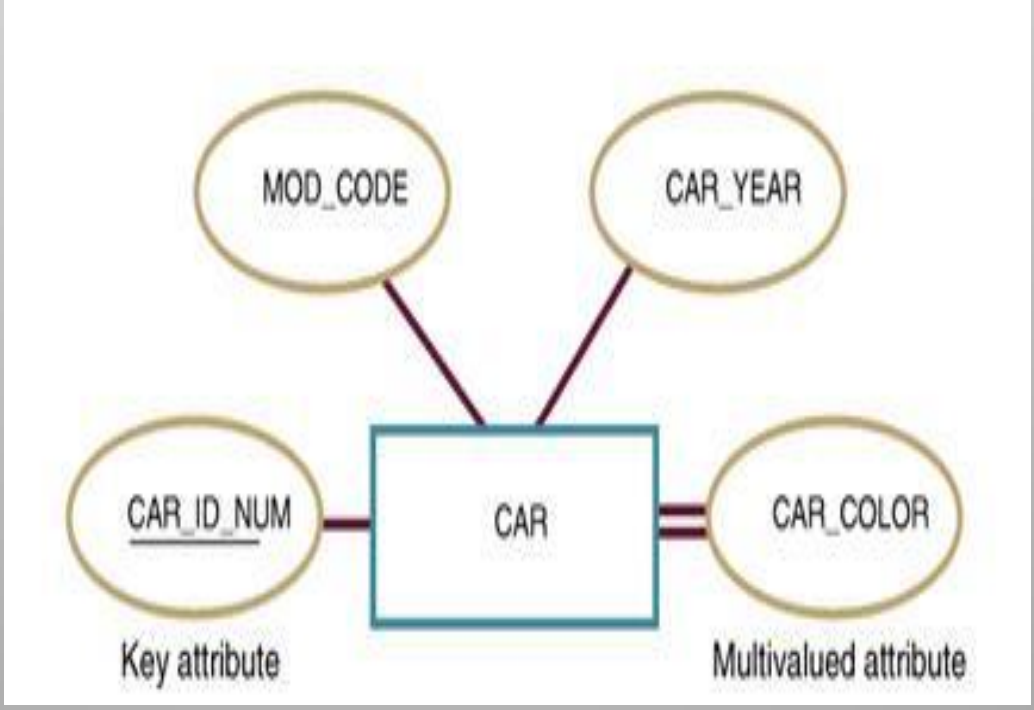
The Entity Relationship (E-R) Model

- E-R Model Components
  - Entities
    - In E-R models an entity refers to the entity set.
    - An entity is represented by a rectangle containing the entity's name.
  - Attributes
    - Attributes are represented by ovals and are connected to the entity with a line.
    - Each oval contains the name of the attribute it represents.
    - Attributes have a domain -- the attribute's set of possible values.
    - Attributes may share a domain.
    - Primary keys are underlined.
  - Relationships

The Attributes of the STUDENT Entity



Basic E-R Model Entity Presentation



The Entity Relationship (E-R) Model

Classes of Attributes

- A **simple attribute** cannot be subdivided.
  - Examples: Age, Sex, and Marital status
- A **composite attribute** can be further subdivided to yield additional attributes.
  - Examples:
    - ADDRESS ↘ Street, City, State, Zip
    - PHONE NUMBER ↘ Area code, Exchange number

The Entity Relationship (E-R) Model

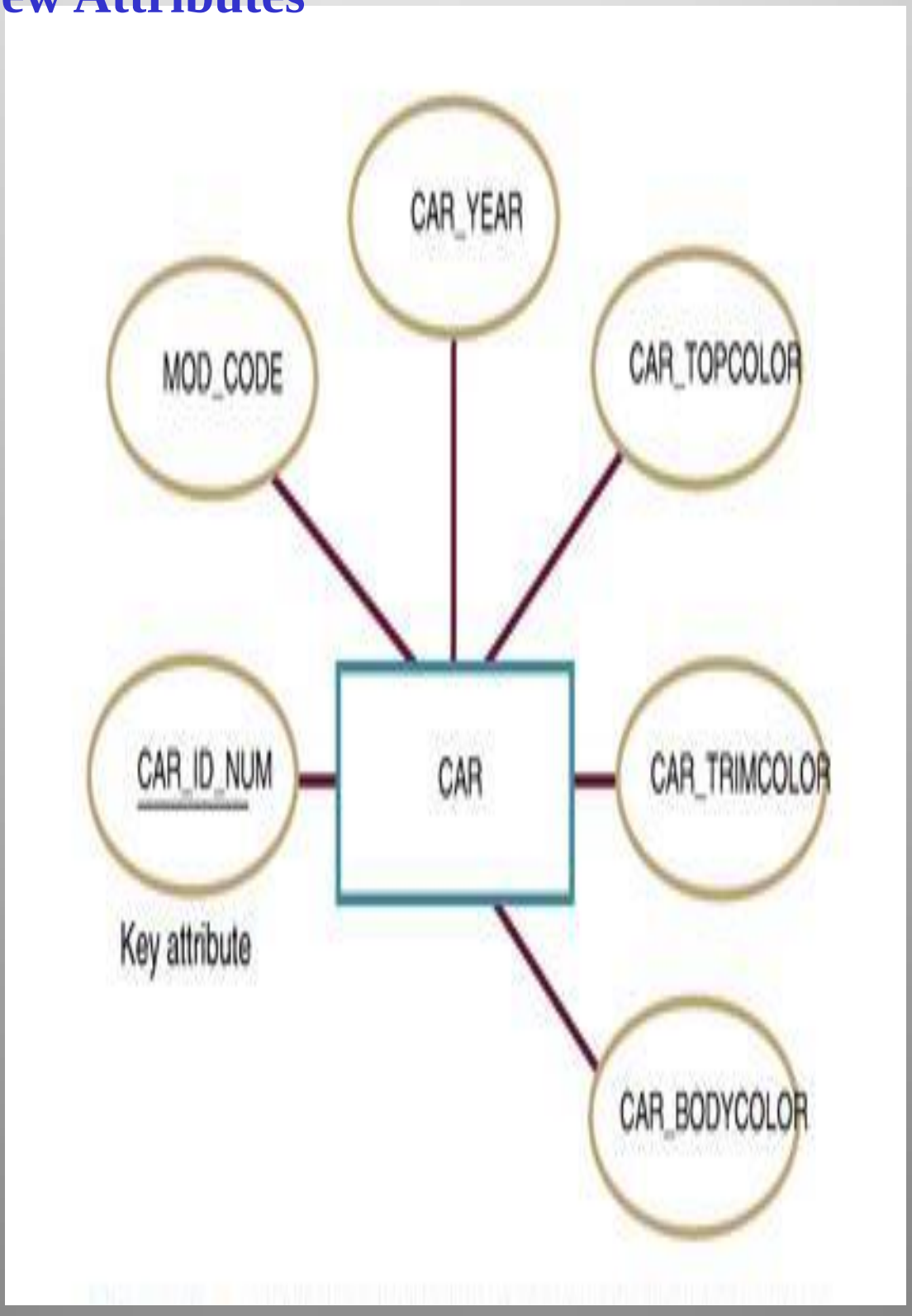
Classes of Attributes

- A single-valued attribute can have only a single value.
  - Examples:
    - A person can have only one social security number.
    - A manufactured part can have only one serial number.
- Multivalued attributes can have many values.
  - Examples:
    - A person may have several college degrees.
    - A household may have several phones with different numbers
  - Multivalued attributes are shown by a double line connecting to the entity.

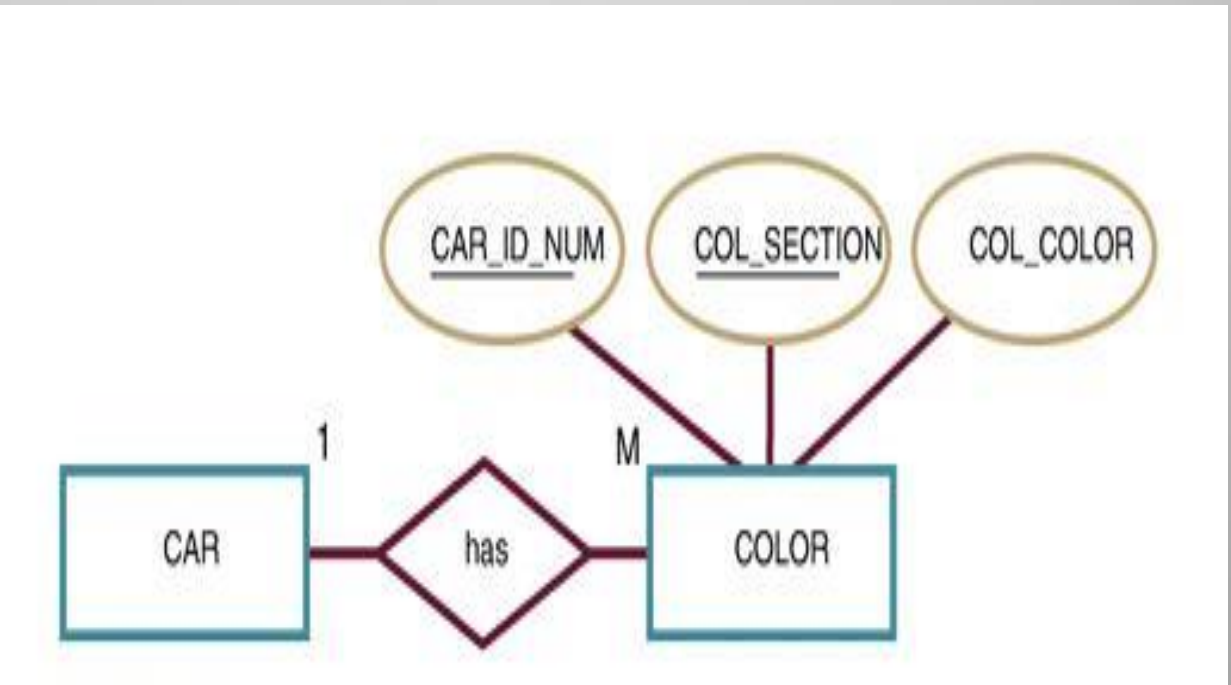
The Entity Relationship (E-R) Model

- Multivalued Attribute in Relational DBMS
  - The relational DBMS cannot implement multivalued attributes.
  - Possible courses of action for the designer
    - Within the original entity, create several new attributes, one for each of the original multivalued attribute's components.
    - Create a new entity composed of the original multivalued attribute's components

Splitting the Multivalued Attributes into New Attributes



A New Entity Set Composed of Multivalu Attribute's Components



The Entity Relationship (E-R) Model

- A derived attribute is not physically stored within the database; instead, it is derived by using an algorithm.
  - Example: AGE can be derived from the data of birth and the current date.

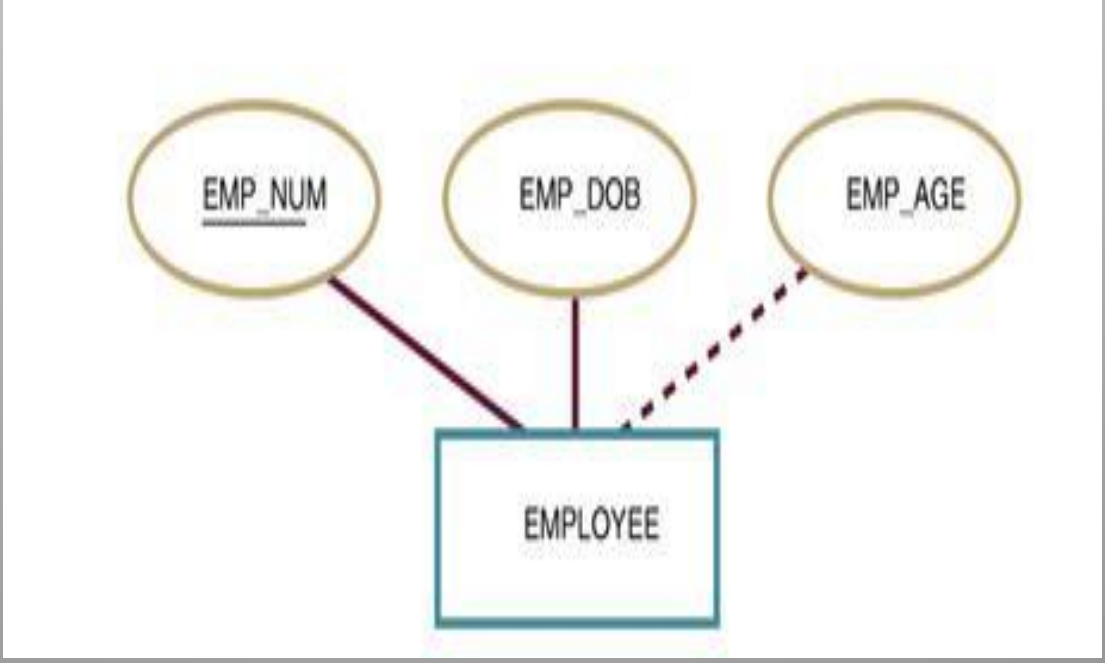


Figure: A Derived Attribute

The Entity Relationship (E-R) Model

- Relationships
  - A **relationship** is an association between entities.
  - Relationships are represented by diamond-shaped symbols.

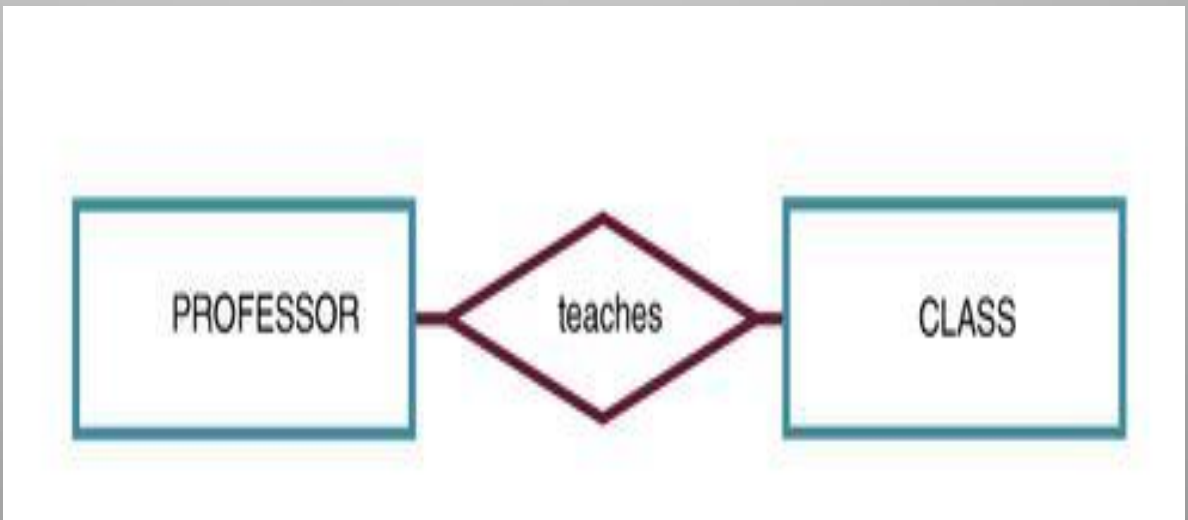
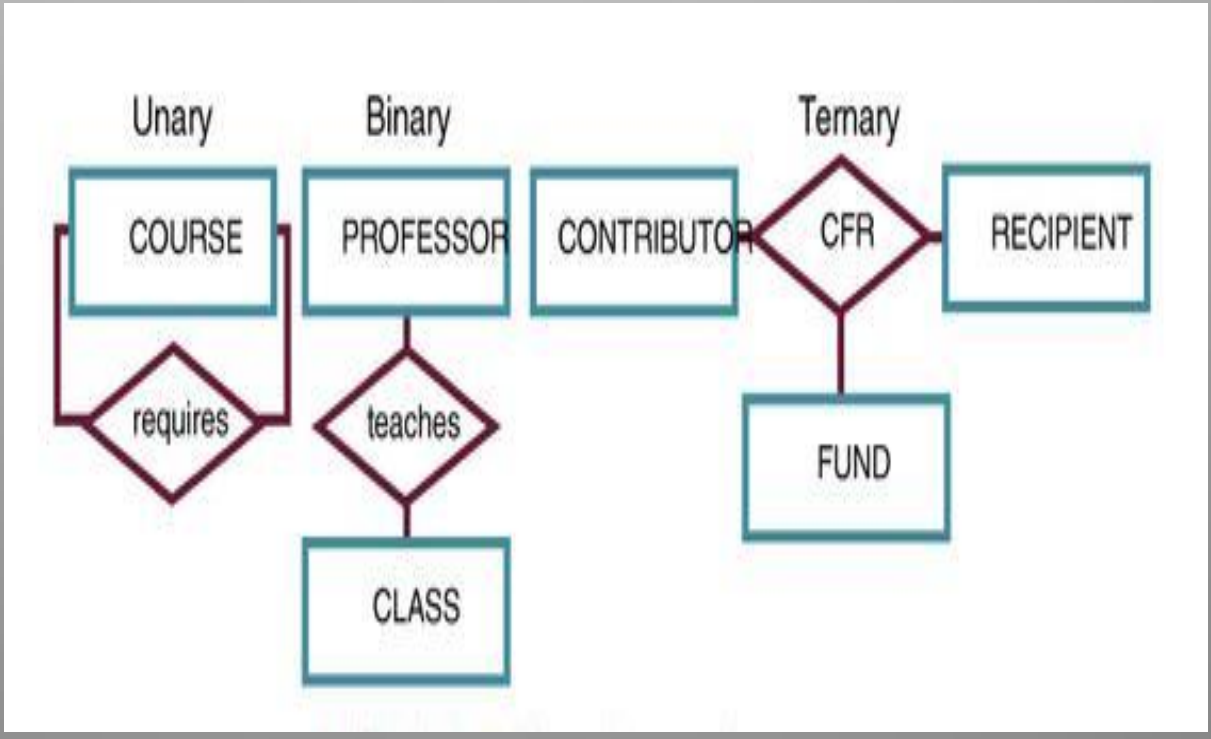


Figure :An Entity Relationship

The Entity Relationship (E-R) Model

- A relationship's degree indicates the number of associated entities or participants.
  - A unary relationship exists when an association is maintained within a single entity.
  - A binary relationship exists when two entities are associated.
  - A ternary relationship exists when three entities are associated.



The Entity Relationship (E-R) Model

- Connectivity
  - The term connectivity is used to describe the relationship classification (e.g., one-to-one, one-to-many, and many-to-many).

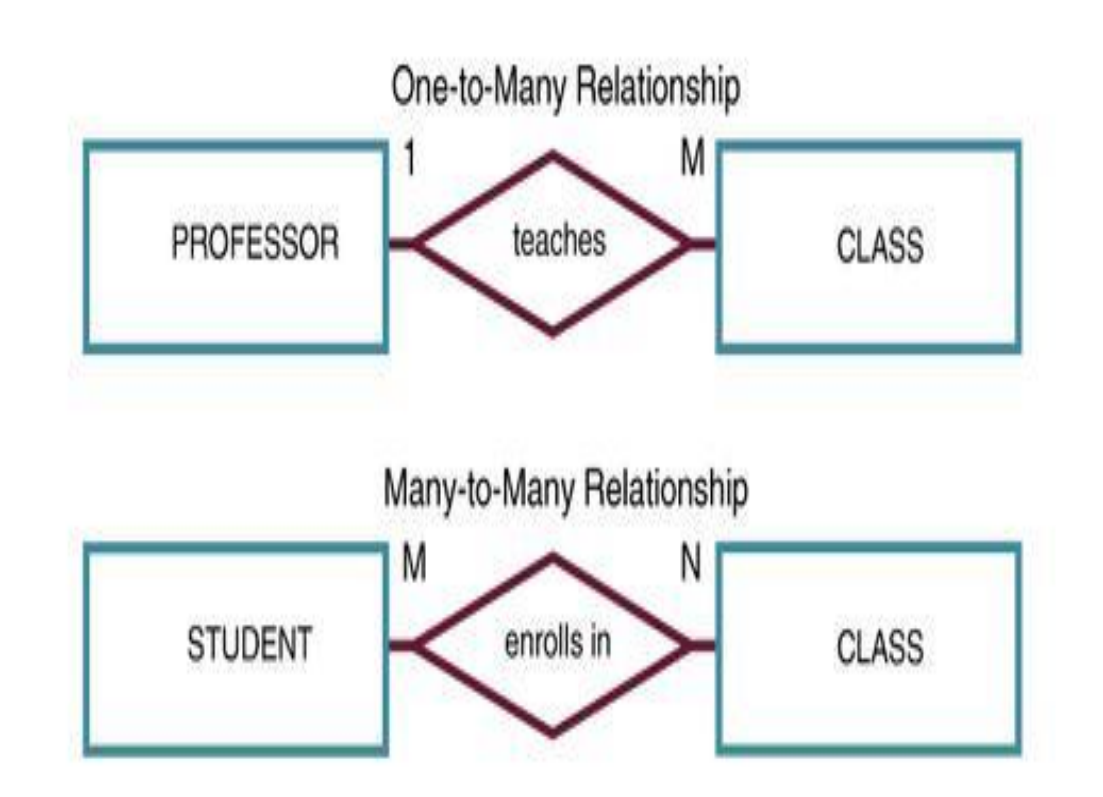


Figure :Connectivity in an ERD

The Entity Relationship (E-R) Model

- **Cardinality**
  - **Cardinality expresses the specific number of entity occurrences associated with one occurrence of the related entity.**

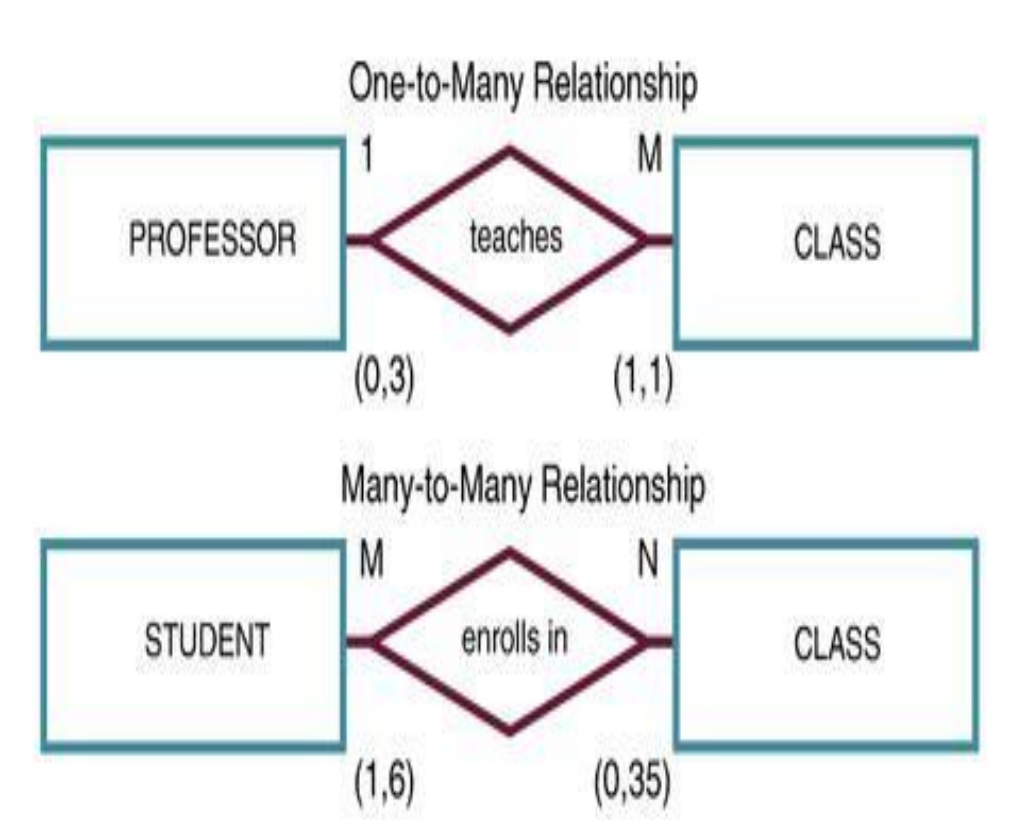


Figure :Cardinality in an ERD

The Entity Relationship (E-R) Model

- Relationship Participation
  - The participation is optional if one entity occurrence does not require a corresponding entity occurrence in a particular relationship.
  - An optional entity is shown by a small circle on the side of the optional entity.

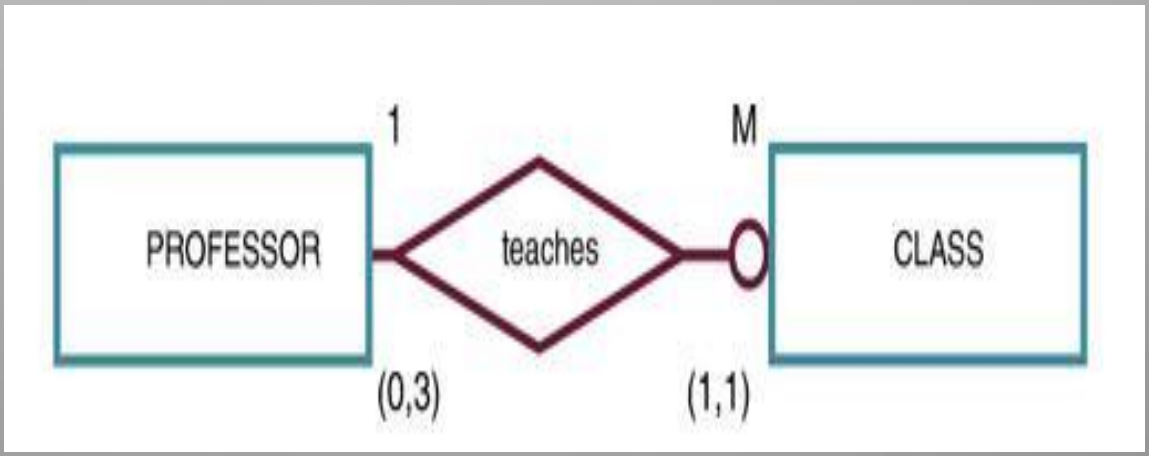


Figure : An ERD With An Optional Entity

Figure : CLASS is Optional to COURSE

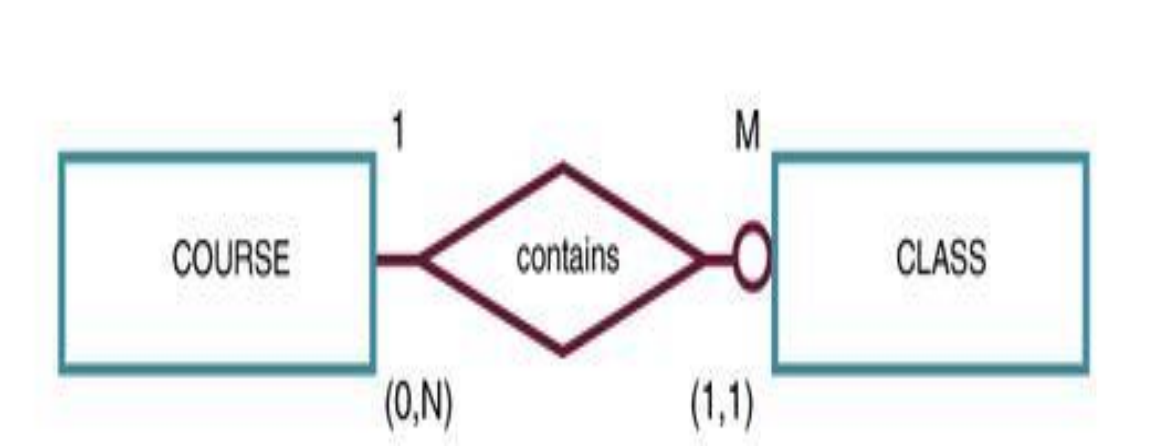
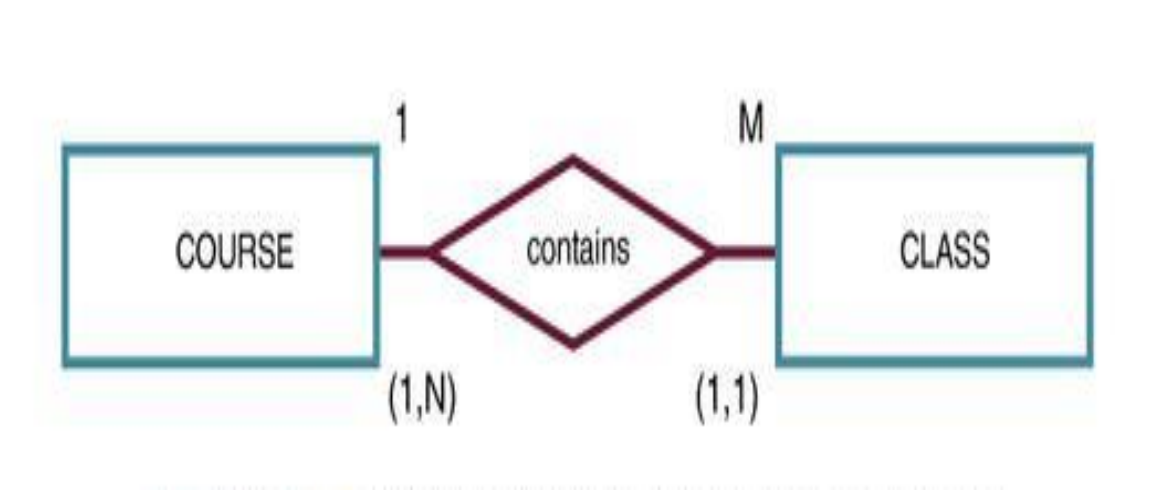


Figure : COURSE and CLASS in a Mandatory Relation

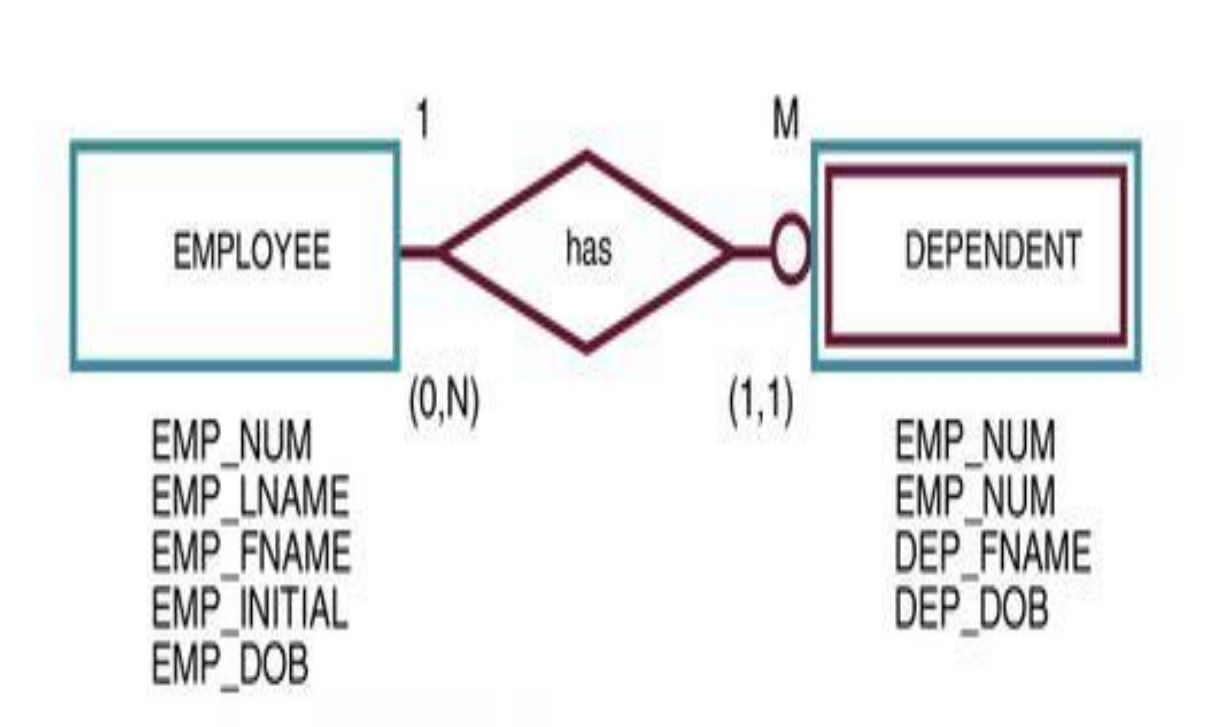


The Entity Relationship (E-R) Model

Weak Entities

- A **weak entity** is an entity that
  - Is existence-dependent and
  - Has a primary key that is partially or totally derived from the parent entity in the relationship.
- The existence of a weak entity is indicated by a double rectangle.
- The weak entity inherits all or part of its primary key from its strong counterpart.

A Weak Entity in an ERD



The Entity Relationship (E-R) Model

- Recursive Entities
  - A **recursive entity** is one in which a relationship can exist between occurrences of the same entity set.
  - A recursive entity is found within a unary relationship.

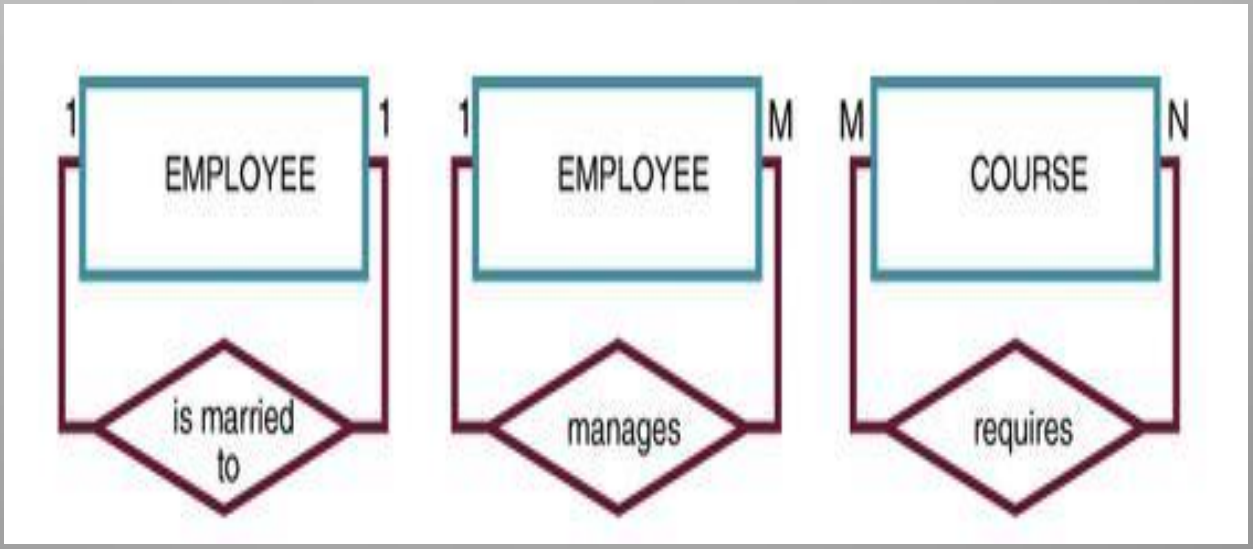


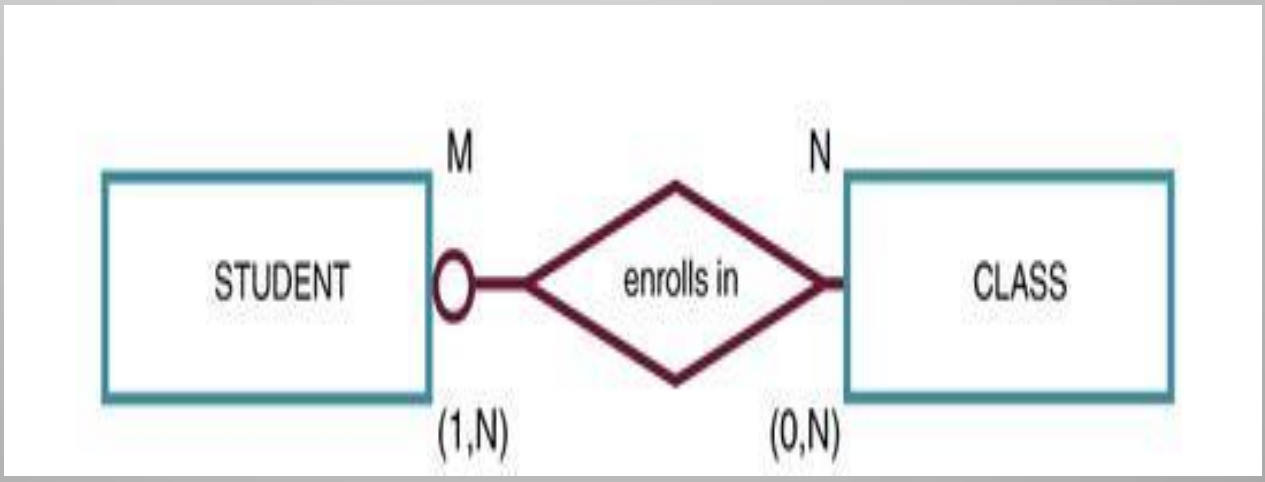
Figure : An E-R Representation of Recursive Relationshi

The Entity Relationship (E-R) Model

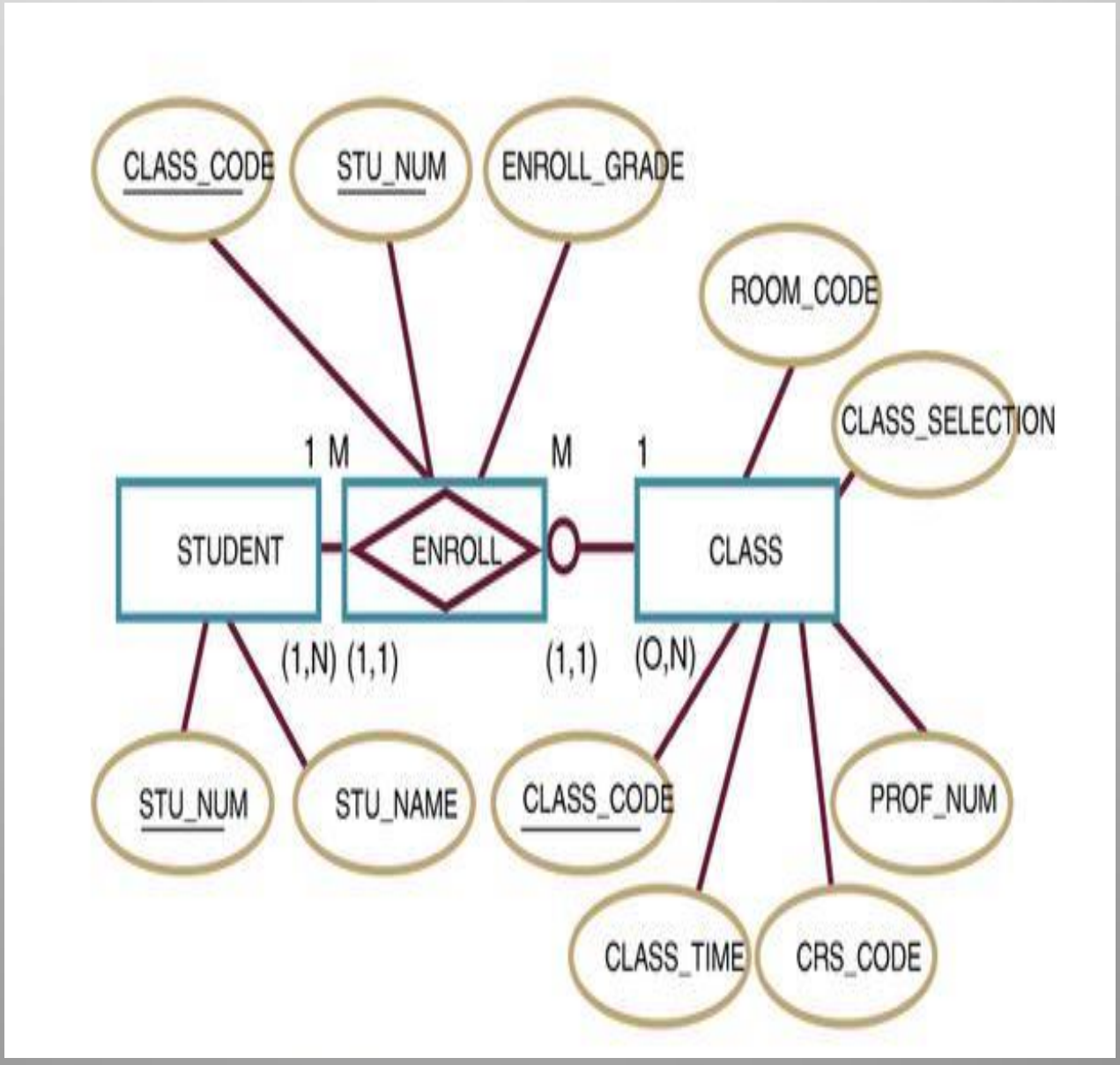
Composite Entities

- A **composite entity** is composed of the primary keys of each of the entities to be connected.
- The composite entity serves as a **bridge** between the related entities.
- The composite entity may contain additional attributes.

The M:N Relationship Between STUDENT and CLASS



A Composite Entity in the ERD



Developing an E-R Diagram

- The process of database design is an iterative rather than a linear or sequential process.
- It usually begins with a general narrative of the organization's operations and procedures.
- The basic E-R model is graphically depicted and presented for review.
- The process is repeated until the end users and designers agree that the E-R diagram is a fair representation of the organization's activities and functions.

Developing an E-R Diagram

- B.D. College Database (1)
  - College is divided into several schools. Each school is administered by a dean. A 1:1 relationship exists between DEAN and SCHOOL.
  - Each dean is a member of a group of administrators (ADMINISTRATOR). Deans also hold professorial rank and may teach a class (PROFESSOR). Administrators and professors are also Employees.

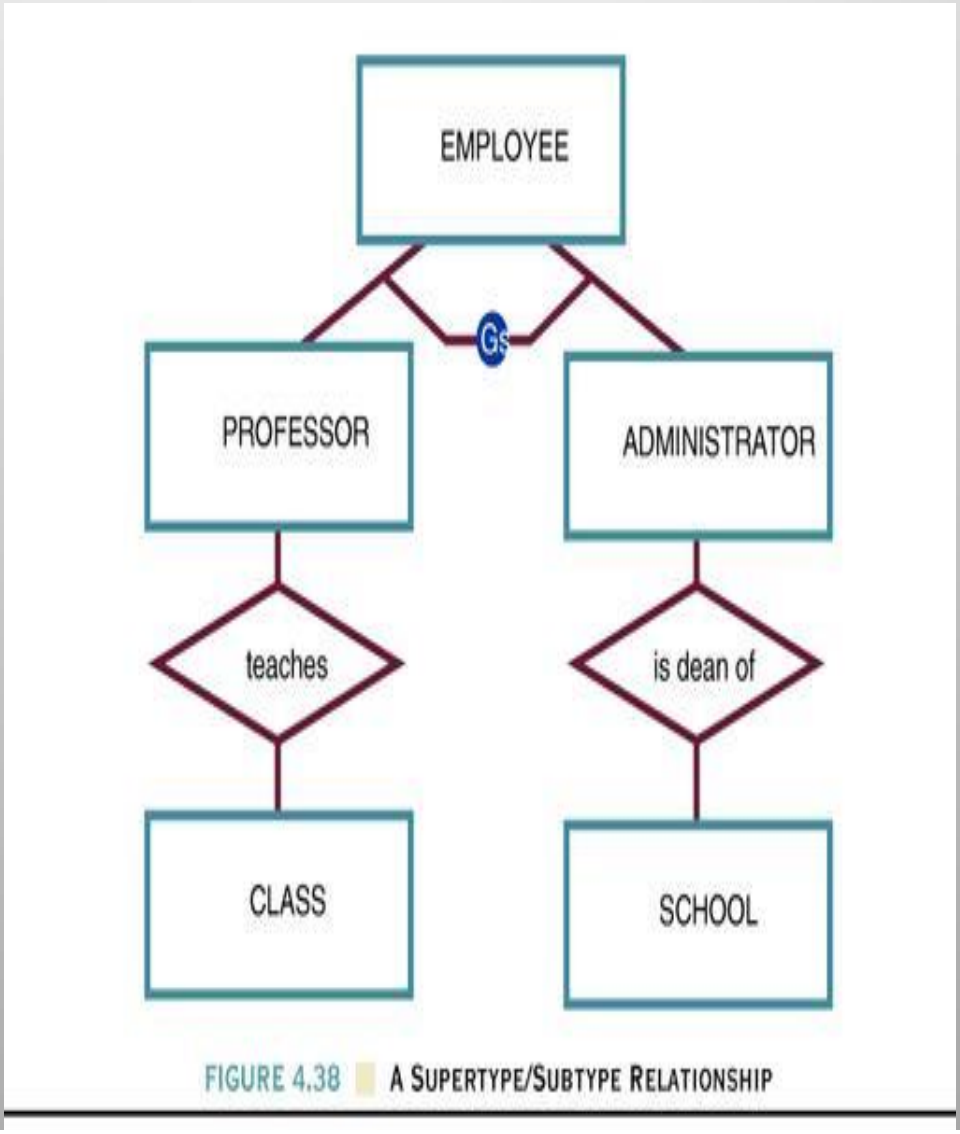
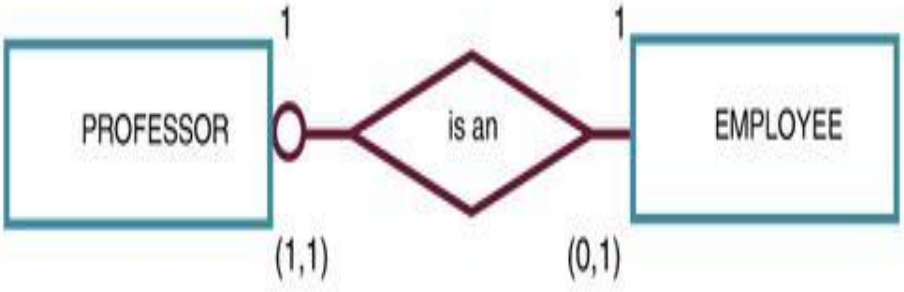


FIGURE 4.38 A SUPERTYPE/SUBTYPE RELATIONSHIP



Developing an E-R Diagram

B.D. College Database (2)

- Each school is composed of several departments.
- The smallest number of departments operated by a school is one, and the largest number of departments is indeterminate (N).
- Each department belongs to only a single school.

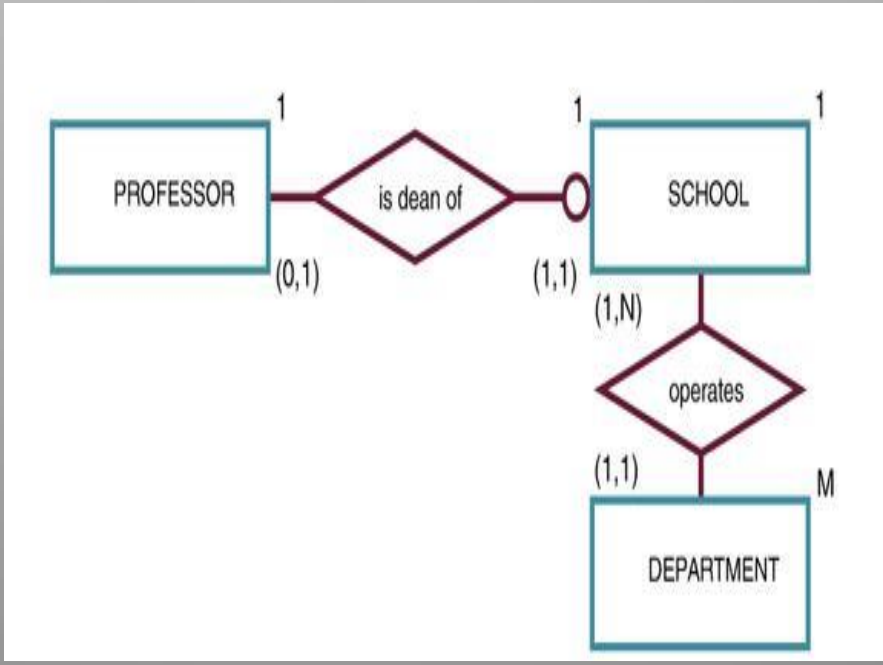


Figure : The First B.D. College ERD Segment

Developing an E-R Diagram

- B.D. College Database (3)
  - Each department offers several courses.

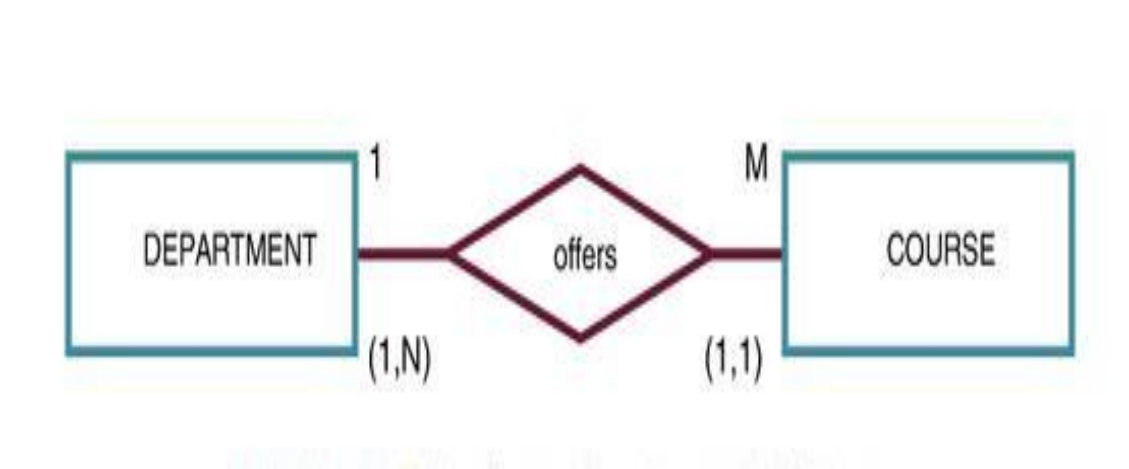


Figure : The Second B.D. College ERD Segmen

Developing an E-R Diagram

B.D. College Database (4)

- A department may offer several sections (classes) of the same course.
- A 1:M relationship exists between COURSE and CLASS.
- CLASS is optional to COURSE

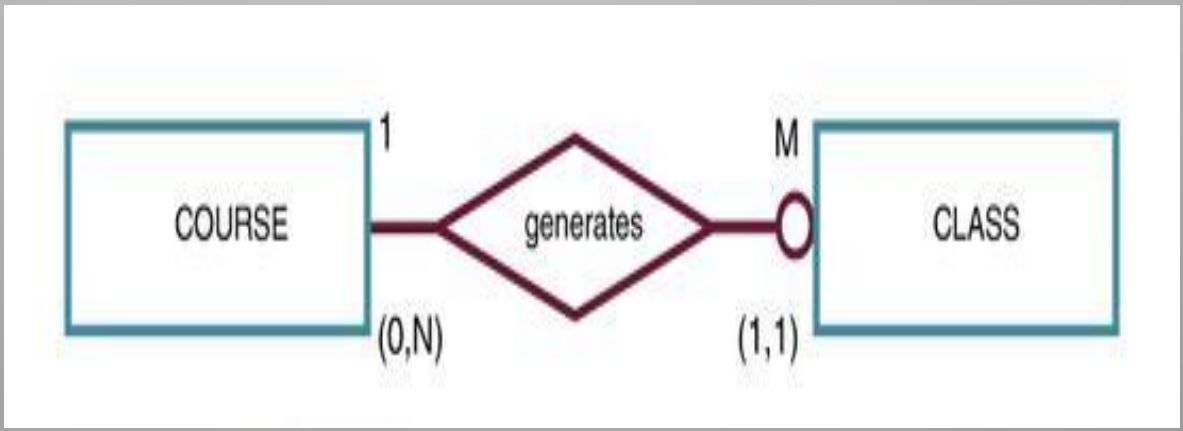


Figure : The Third B.D. College ERD Segment

Developing an E-R Diagram

B.D. College Database (5)

- Each department has many professors assigned to it.
- One of those professors chairs the department. Only one of the professors can chair the department.
- DEPARTMENT is optional to PROFESSOR in the “chairs” relationship.

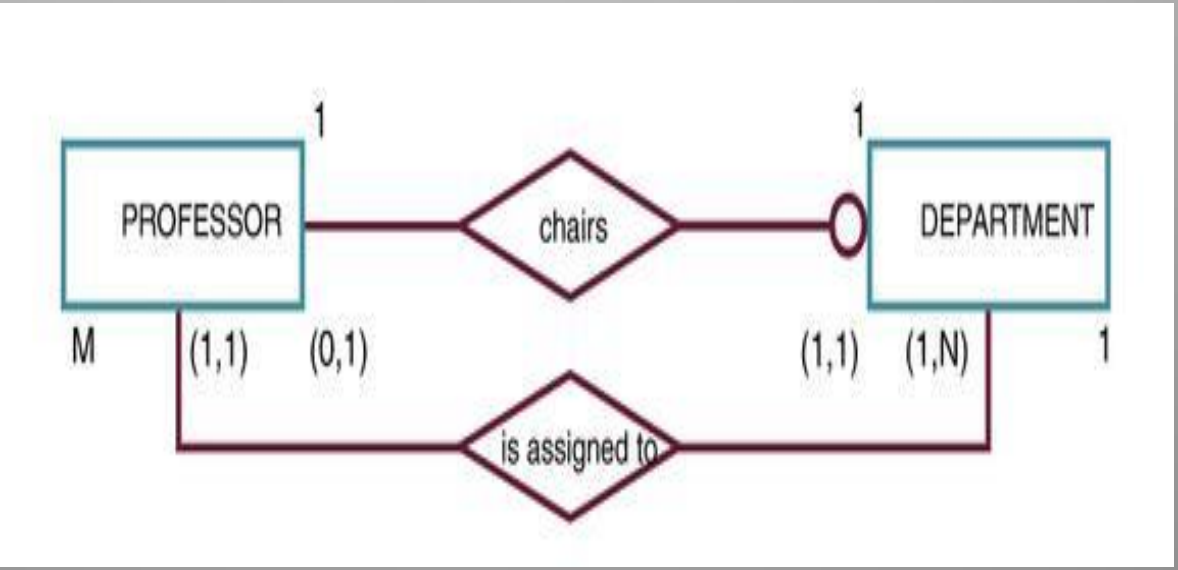


Figure : The Fourth B.D. College ERD Segmer

Developing an E-R Diagram

- B.D. College Database (6)
  - Each professor may teach up to four classes, each one a section of a course.
  - A professor may also be on a research contract and teach no classes.

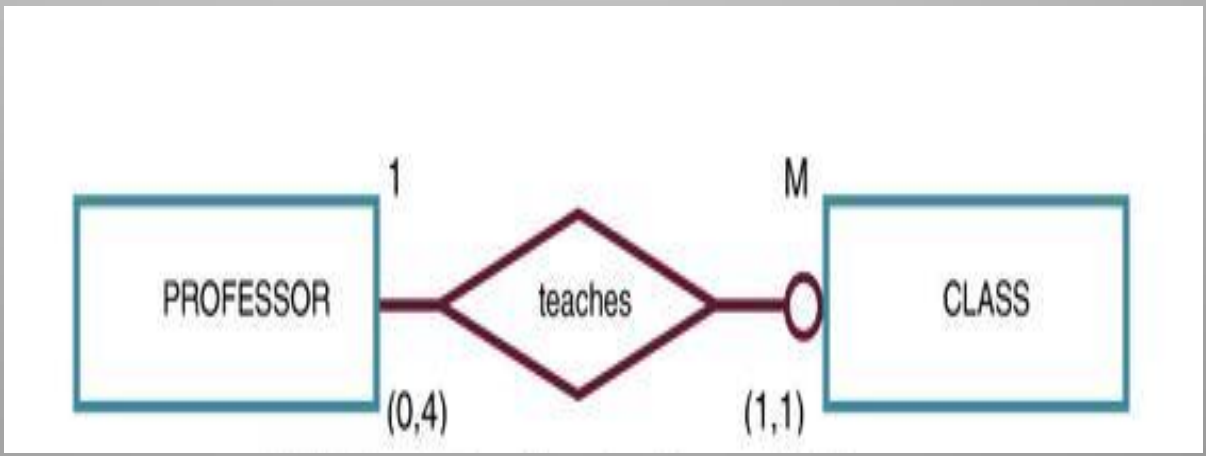


Figure : The Fifth B.D. College ERD Segment

Developing an E-R Diagram

B.D. College Database (7)

- A student may enroll in several classes, but (s)he takes each class only once during any given enrollment period.
- Each student may enroll in up to six classes and each class may have up to 35 students in it.
- STUDENT is optional to CLASS.

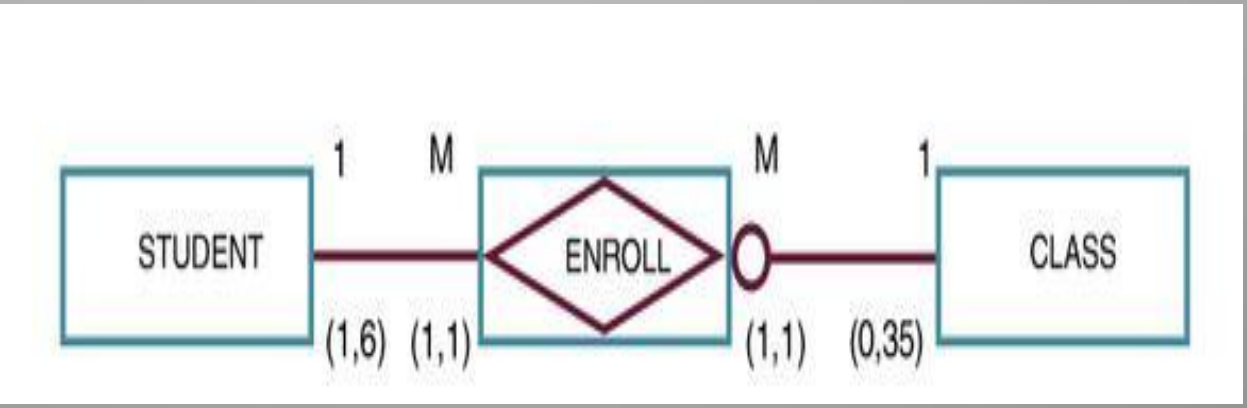


Figure : The Sixth B.D. College ERD Segment

Developing an E-R Diagram

B.D. College Database (8)

- Each department has several students whose major is offered by that department.
- Each student has only a single major and associated with a single department.

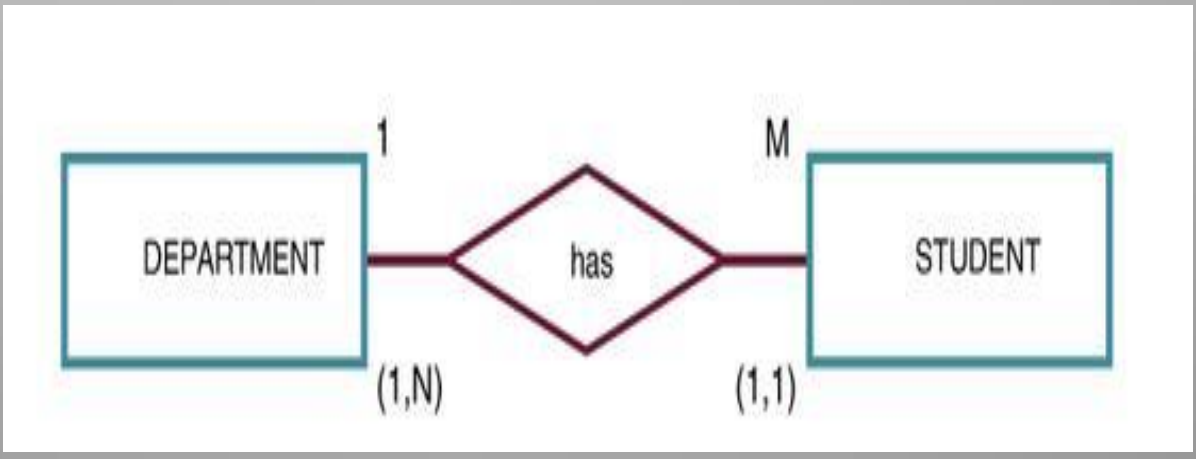


Figure : The Seventh B.D. College ERD Segme

Developing an E-R Diagram

B.D. College Database (9)

- Each student has an advisor in his or her department; each advisor counsels several students.
- An advisor is also a professor, but not all professors advise students.

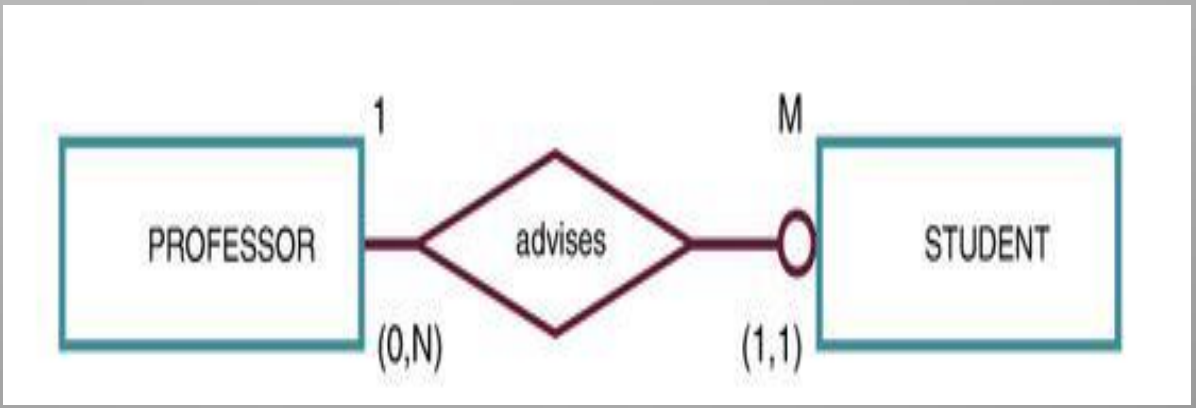


Figure : The Eighth B.D. College ERD Segmen

Developing an E-R Diagram

Entities for the B.D. College Database

- SCHOOL
- DEPARMENT
- EMPLOYEE
- PROFESSOR
- COURSE
- CLASS
- ENROLL (Bridge between STUDENT and CLASS)
- STUDENT

Week 3-4

Sessional -03

Various Data Types

Objective: To explore and utilize data types in SQL for efficient table creation.

Activities:

- Create a table to store employee details, using
- appropriate data types.
- Insert data and observe storage behaviors.

Sample Query:

```
CREATE TABLE Employee ( EmpID
   NUMBER(5), EmpName
   VARCHAR2(50),
   JoinDate DATE
);
```

Task: Experiment with different data types by creating and populating tables.

Practice Problems:

1. Create a `Products` table with fields `ProductID`, `ProductName`, `Price`, and `ManufactureDate`.
2. Insert at least 5 records into the `Products` table.
3. Write a query to retrieve products priced above 500.

Instructor Signature (with date)

Supervisor's signature (with date)

Sessional-04

Tables

Objective: To create and manipulate tables in SQL.

Activities:

- Design a `Reservation` table for a travel system.
- Practice inserting, updating, and deleting records.

Sample Query:

```
CREATE TABLE Reservation (
   PNR_NO NUMBER(9) PRIMARY KEY,
   PassengerName VARCHAR2(50),
   No_of_Seats NUMBER(3)
);
```

Task: Create a relational schema for a library system, including tables for books, members, and loans.

Practice Problems:

1. Add a column `BookingDate` to the `Reservation` table.
2. Delete all records where `No_of_Seats` is less than 2.
3. Retrieve all reservations made for more than 4 seats.

Instructor Signature (with date) **Supervisor's signature (with date)**

DDL

Introduction

- To understand the SQL Data Definition Language
  - Create
  - Insert
  - Delete
  - Drop
  - Truncate
  - Alter

DDL

Creating a Database

- To initialize a new database:
- **Syntax:**

  CREATE DATABASE database_name
- There are numerous arguments that go along with this command but are database specific
- Only some databases require database to be created and space to be allocated prior to creation of tables.
- Some databases provide graphical user interfaces to create databases and allocate space.
  - Access only allows database to be created using User Interface

DDL

Creating a Table

- **Syntax**

```
CREATE TABLE table_name
(Column_name     datatype[(size)],
 Column_name     datatype[(size)],
)
```

- **Example**

```
CREATE TABLE books
(ISBN                char(20),
Title                        char(50),
AuthorID             Integer,
Price                float)
```

- Creates a table with four columns

DDL

Data Types

- Following broad categories of data types exist in most databases:
  - String Data
  - Numeric Data
  - Temporal Data
  - Large Objects

DDL

Data Types

- Following broad categories of data types exist in most databases:
  - String Data
  - Numeric Data
  - Temporal Data
  - Large Objects

DDL

String Data

- **Fixed Length**:
- Occupies the same length of space in memory no matter how much data is stored in them.
- **Syntax:**

  char(n) where n is the length of the String

  e.g. name char(50)
- If the variable stored for name is ‘Sanjay’ the extra 43 fields are padded with blanks

DDL

String Data

- **Variable Length** string is specified with maximum length of characters possible in the string, however, the allocation is sized to the size of the data stored in memory.
- **Syntax:**

  Varchar(n) – n is the maximum length of data possible for the type
- There may be a restriction in the maximum length of the data that you can specify in the declaration which will vary according to the database.
- All character data has to be enclosed in single quotes during specification.

DDL

Numeric Data Types

- Store all the data related to purely numeric data.
- Some numeric data may also be stored as a character field e.g. zip codes
- **Common Numeric Types**:
  - Decimal — Floating point number
  - Float — Floating point number
  - Integer(size) — Integer of specified length
  - Money — A number which contains exactly two digits after the decimal point
  - Number — A standard number field that can hold a floating point data

Note: Different databases name their numeric fields differently and may not support all numeric types. They may also support additional numeric types.

DDL

Temporal Data Types

- **These represent the dates and time:**
- Three basic types are supported:
  - Dates
  - Times
  - Date-Time Combinations

DDL

Large Data Objects

- These are used for storing data objects like files and images:
- **There are two types**:
  - Character Large Objects (clobs)
  - Binary Large Objects (blobs)

DDL

Specifying Keys- Introduction

- Unique keyword is used to specify keys.
  - This ensures that duplicate rows are not created in the database.
- Both Primary keys and Candidate Keys can be specified in the database.
- Once a set of columns has been declared unique any data entered that duplicates the data in these columns is rejected.
- **Specifying a single column as unique**:
- Example

```
CREATE TABLE Studios
(studio_id          Number,
name                char(20),
city                varchar(50),
state               char(2),
UNIQUE (name))
```

- Here the name column has been declared as a candidate key

DDL

Specifying Keys- Multiple Columns

- Specifying multiple columns as unique:
- **Example:**

```
CREATE TABLE Studios
(studio_id          Number,
name                char(20),
city                varchar(50),
state                       char(2),
UNIQUE (name),
UNIQUE(city, state))
```

- Here both name & city/state combination are declared as candidate keys

DDL

Specifying Keys- Primary Key

- Specifying multiple columns as unique:
- To specify the Primary Key the **Primary Key** clause is used

- **Example:**

```
CREATE TABLE Studios
(studio_id      Number,
name                char(20),
city            varchar(50),
state           char(2),
PRIMARY KEY (studio_id),
UNIQUE (name),
UNIQUE(city, state)
)
```

DDL

Specifying Keys- Foreign Keys

- References clause is used to create a relationship between a set of columns in one table and a candidate key in the table that is being referenced.

- **Example:**

```
CREATE TABLE Movies
(movie_title      varchar(40),
studio_id         Number REFERENCES
      Studios(studio_id))
```

- Creates a relationship from the Movies table to the Studios table

Modifying Records

Insert Statement

- **Insert:**
  - Allows you to add new records to the Table
- **Syntax:**
  - Insert into table_name[(column_list)] values (value_list)
- **Example:**

INSERT INTO studios
VALUES (1, 'Giant', 'Los Angeles', 'CA')

INSERT INTO studios
(studio_city, studio_state, studio_name, studio_id)
VALUES ('Burbank', 'CA', 'MPM', 2)

- **Notes1:** If the columns are not specified as in the first example the data goes in the order specified in the table
- **Notes2:** There are two ways of inserting Null values
  1. If the field has a default value of Null, you can use an Insert statement that ignores the column where the value is to be Null.
  2. You can specify the column in the column list specification and assign a value of Null to the corresponding value field.

Modifying Records

Select & Insert

- **Select & Insert:**
  - A select query can be used in the insert statement to get the values for the insert statement

- **Example:**

```
INSERT INTO city_state
SELECT studio_city, studio_state FROM studios
```

- This selects the corresponding fields from the studios table and inserts them into the city_state table.

- **Example:**

```
INSERT INTO city_state
SELECT Distinct studio_city, studio_state FROM studios
```

- This selects the corresponding fields from the studios table, deletes the duplicate fields and inserts them into the city_state table. Thus the final table has distinct rows

Modifying Records

Delete Statement

- **Delete Statement:**
  - is used to remove records from a table of the database. The where clause in the syntax is used to restrict the rows deleted from the table otherwise all the rows from the table are deleted.

- **Syntax:** DELETE FROM table_name [WHERE Condition]

- **Example:**

```
DELETE FROM City_State
WHERE state = 'TX'
```

- Deletes all the rows where the state is Texas keeps all the other rows.

Modifying Records

Update Statement

- **Update Statement:**
  - used to make changes to existing rows of the table. It has three parts. First, you ,must specify which table is going to be updated. The second part of the statement is the set clause, in which you should specify the columns that will be updated as well as the values that will be inserted. Finally, the where clause is used to specify which rows will be updated.

- **Syntax:**

UPDATE table_name
SET column_name1 = value1, column_name2 = value2, ...
[WHERE Condition]

- **Example:**

UPDATE studios
SET studio_city = 'New York', studio_state = 'NY'
WHERE studio_id = 1

- **Notes1**: If the condition is dropped then all the rows are updated.

Modifying Records

Truncate Statement

- **Truncate Statement:**
  - used to delete all the rows of a table. Delete can also be used to delete all the rows from the table. The difference is that delete performs a delete operation on each row in the table and the database performs all attendant tasks on the way. On the other had the Truncate statement simply throws away all the rows at once and is much quicker. The note of caution is that truncate does not do integrity checks on the way which can lead to inconsistencies on the way. If there are dependencies requiring integrity checks we **should** use delete.

- **Syntax:** TRUNCATE TABLE table_name

- **Example:**

TRUNCATE TABLE studios

- This deletes all the rows of the table studios

Modifying Records

Drop Statement

- **Drop Statement:**
  - used to remove elements from a database, such as tables, indexes or even users and databases. Drop command is used with a variety of keywords based on the need.

- **Drop Table Syntax:** DROP TABLE table_name
- **Drop Table Example:** DROP TABLE studios

- **Drop Index Syntax:** DROP INDEX table_name
- **Drop Index Example:** DROP INDEX movie_index

Modifying Records

Alter Statement

- **Alter Statement:**
  - used to make changes to the schema of the table. Columns can be added and the data type of the columns changed as long as the data in those columns conforms to the data type specified.

- **Syntax:**

ALTER TABLE table_name
ADD (column datatype [Default Expression])
[REFERENCES table_name (column_name)'
[CHECK condition]

- **Example:**

ALTER TABLE studios
ADD (revenue Number DEFAULT 0)

Modifying Records

Alter Statement

Add table level constraints:

- **Syntax:**

ALTER TABLE table_name
ADD ([CONSTRAINT constraint_name CHECK comparison]
[columns REFERENCES table_name (columns)]

- **Example:**

ALTER TABLE studios
ADD (CONSTRAINT check_state CHECK (studio_state in ('TX', 'CA', 'WA'))

Modify Columns:

- **Syntax:**

ALTER TABLE table_name
MODIFY column [data type]
[Default Expression]
[REFERENCES table_name (column_name)'
[CHECK condition]

- **Example:**

ALTER TABLE People
MODIFY person_union varchar(10)

- **Notes1:** Columns can not be removed from the table using alter. If you want to remove columns you have to drop the table and then recreate it without the column that you want to discard

Modifying Records

Alter Statement

- **Alter Statement:**
  - used to make changes to the schema of the table. Columns can be added and the data type of the columns changed as long as the data in those columns conforms to the data type specified.

- **Syntax:**

ALTER TABLE table_name
ADD (column datatype [Default Expression])
[REFERENCES table_name (column_name)'
[CHECK condition]

- **Example:**

ALTER TABLE studios
ADD (revenue Number DEFAULT 0)

DDL

Specifying Keys- Single and MultiColumn Keys

- Single column keys can be defined at the column level instead of at the table level at the end of the field descriptions.
- MultiColumn keys still need to be defined separately at the table level

```
CREATE TABLE Studios
(studio_id    Number      PRIMARY KEY,
name          char(20)    UNIQUE,
city          varchar(50),
state               char(2),
Unique(city, state))
```

- Note: Some databases require the use of Unique Index for specification of keys.

Week 5-6

Sessional -05

MySQL Installation

Objective: To set up MySQL on a local machine for hands-on practice.

Details:

- **Installation Steps:**
  - Download the MySQL installer from the official website.
  - Run the installer and select the appropriate setup type (e.g., Developer Default).
  - Configure server options, including root password and port.
  - Test the installation by connecting to the MySQL server using the CLI.

Task: Successfully set up MySQL and create a new database named `College`.

Practice Problems:

1. Verify MySQL installation and check server status.
2. Create a new user with specific privileges in MySQL.
3. Create a database `Library` and switch to it.

Instructor Signature (with date) **Supervisor's signature (with date)**

Sessional -06

DDL and DML Commands with Examples

Objective: To practice defining and manipulating database structures and records.

Details:

- **DDL (Data Definition Language):** Commands like `CREATE`, `ALTER`, `DROP`.
- **DML (Data Manipulation Language):** Commands like `INSERT`, `UPDATE`, `DELETE`, `SELECT`.

Activities:

- Create a table for storing product details.
- Insert sample data and retrieve specific records using `WHERE` conditions.

Sample Queries:

```
CREATE TABLE Products (
    ProductID NUMBER(5),
    ProductName VARCHAR2(50),
    Price NUMBER(10,2)
);

INSERT INTO Products VALUES (1, 'Laptop', 75000.00);
SELECT * FROM Products WHERE Price > 50000;
```

Task: Practice creating and modifying tables and inserting data into them.

Practice Problems:

1. Write a query to update the price of a product in the `Products` table.
2. Delete a record with `ProductID` 3.
3. Retrieve all products with a name starting with 'L'.

Instructor Signature (with date) **Supervisor's signature (with date)**

Introduction of MySQL

- MySQL is an SQL (Structured Query Language) based relational database management system (DBMS)
- MySQL is compatible with standard SQL
- MySQL is frequently used by PHP and Perl
- Commercial version of MySQL is also provided (including technical support)

Resource

- MySQL and GUI Client can be downloaded from

  http://dev.mysql.com/downloads/

- The SQL script for creating database 'bank' can be found at
  - http://www.cs.kent.edu/~mabuata/DB10_lab/bank_db.sql
  - http://www.cs.kent.edu/~mabuata/DB10_lab/bank_data.sql

Command for accessing MySQL

- Access from DB server

  >ssh dbdev.cs.kent.edu

  Start MySQL

  >mysql –u [username] –p

  >Enter password:[password]

- From a departmental machine

  >mysql -u [username] -h dbdev.cs.kent.edu –p

  >Enter password:[password]

Entering & Editing commands

- Prompt mysql>
  - issue a command
  - Mysql sends it to the server for execution
  - displays the results
  - prints another mysql>
- a command could span multiple lines
- A command normally consists of SQL statement followed by a semicolon

Command prompt

prompt	meaning
mysql>	Ready for new command.
->	Waiting for next line of multiple-line command.
'>	Waiting for next line, waiting for completion of a string that began with a single quote ("'").
">	Waiting for next line, waiting for completion of a string that began with a double quote (""").
`>	Waiting for next line, waiting for completion of an identifier

MySQL commands

- help \h
- Quit/exit \q
- Cancel the command \c
- Change database use
- …etc

Info about databases and tables

- Listing the databases on the MySQL server host
  - >show databases;
- Access/change database
  - >Use [database_name]
- Showing the current selected database
  - > select database();
- Showing tables in the current database
  - >show tables;
- Showing the structure of a table
  - > describe [table_name];

Banking Example

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

customer (customer-name, customer-street, customer-city)

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

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

depositor (customer-name, account-number)

borrower (customer-name, loan-number)

employee (employee-name, branch-name, salary)

CREATE DATABASE

- An SQL relation is defined using the **CREATE DATABASE** command:
  - **create database** [*database name*]

- Example
  - **create database** *mydatabase*

SQL Script for creating tables

- The SQL script for creating database 'bank' can be found at

http://www.cs.kent.edu/~mabuata/DB10_lab/bank_db.sql
http://www.cs.kent.edu/~mabuata/DB10_lab/bank_data.sql

Notice: we do not have permission to create database, so you have to type command "use [your_account]" to work on your database.

Query

- To find all loan number for loans made at the Perryridge branch with loan amounts greater than $1100.

  select *loan_number* **from** *loan*
  where *branch_name* = 'Perryridge' **and** *amount*>1100;

- Find the loan number of those loans with loan amounts between $1,000 and $1,500 (that is, ≥$1,000 and ≤$1,500)

  select *loan_number* **from** *loan*
  where *amount* **between** 1000 **and** 1500;

Query

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

  select distinct *T.branch_name*

  from *branch* **as** *T, branch* **as** *S*
  where *T.assets > S.assets* **and** *S.branch_city* = 'Brooklyn';

- Find the customer names and their loan numbers for all customers having a loan at some branch.

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

Set Operation

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

  (select *customer_name* **from** *depositor*)
  union
  (select *customer_name* **from** *borrower);*

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

  (no **minus** operator provided in mysql)

  select *customer_name* **from** *depositor*

  where *customer_name* **not in**
  (select *customer_name* **from** *borrower);*

Aggregate function

- Find the number of depositors for each branch.

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

- Find the names of all branches where the average account balance is more than $500.

select *branch_name,* **avg** *(balance)*
from *account*
group by *branch_name*
having avg*(balance)* > 500;

Nested Subqueries

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

  select distinct *customer_name*
  from *borrower*
  where *customer_name* **in**

  (select *customer_name* **from** depositor);

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

  select distinct *customer_name*
  from *borrower*
  where *customer_name* **not in**

  (select *customer_name* **from** *depositor);*

Nested Subquery

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

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

Create View (new feature in mysql 5.0)

- A view consisting of branches and their customers

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

Joined Relations

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

Join types	*Join Conditions*
inner join	natural
left outer join	on <predicate>
right outer join	using $(A_1, A_1, \ldots, A_n)$
full outer join	

Joined Relations – Datasets for Examples

- Relation *loan*

loan_number	*branch_name*	*amount*
L-170	Downtown	3000
L-230	Redwood	4000
L-260	Perryridge	1700

loan

- Relation *borrower*

customer_name	*loan_number*
Jones	L-170
Smith	L-230
Hayes	L-155

borrower

- Note: borrower information missing for L-260 and loan information missing for L-155

Joined Relations – Examples

loan_number	*branch_name*	*amount*
L-170	Downtown	3000
L-230	Redwood	4000
L-260	Perryridge	1700

loan

customer_name	*loan_number*
Jones	L-170
Smith	L-230
Hayes	L-155

borrower

- *Select * from loan* **inner join** *borrower* **on** *loan.loan-number = borrower.loan-number*

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

Example

loan_number	*branch_name*	*amount*
L-170	Downtown	3000
L-230	Redwood	4000
L-260	Perryridge	1700

loan

customer_name	*loan_number*
Jones	L-170
Smith	L-230
Hayes	L-155

borrower

- *Select * from loan* **left join** *borrower* **on** *loan.loan-number = borrower.loan-number*

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

Modification of Database

- Increase all accounts with balances over $800 by 7%, all other accounts receive 8%.

```
update account
set balance = balance * 1.07
where balance > 800;
```

```
update account
set balance = balance * 1.08
where balance ≤ 800;
```

Modification of Database

- Increase all accounts with balances over $700 by 6%, all other accounts receive 5%.

```
update account
set balance =case
            when balance <= 700 then balance
*1.05
                        else   balance * 1.06
            end;
```

Modification of Database

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

  delete from *account*
  where *balance* < (**select avg** *(balance)* **from** *account);*

- Add a new tuple to *account*

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

Week 7-8

Sessional -07

Key Constraints and Normalization

Objective: To ensure data integrity and eliminate redundancy.

Details:

- **Key Constraints:** PRIMARY KEY, FOREIGN KEY, UNIQUE, CHECK.
- **Normalization:** Organizing data into tables to reduce redundancy and improve integrity.
    - **1NF:** Eliminate repeating groups by ensuring each column contains atomic values.
    - **2NF:** Remove partial dependencies by ensuring non-primary attributes are fully dependent on the primary key.
    - **3NF:** Remove transitive dependencies by ensuring non-primary attributes depend only on the primary key.

Activities:

- Normalize a student database to 3NF.
- Apply constraints to enforce rules.

Sample Query:

```
ALTER TABLE Reservation ADD CONSTRAINT FK_Passenger FOREIGN KEY (PassengerID)
REFERENCES Passenger(ID);
```

Task:

- Normalize an `Orders` database to 3NF and highlight the process step-by-step.

Practice Problems:

1. Identify redundancy in a `Sales` table and normalize it to 1NF, 2NF, and 3NF.
2. Apply appropriate constraints to ensure data integrity in a `Customers` table.
3. Design a schema for a college database and normalize it.

- The figure below demonstrates normalization from 1NF to 3NF for a `Student Course Enrollment` table:

4. **Unnormalized Table (UNF):** | StudentID | Name | Courses | |-----------|-------|-------------| | 101 | Alice | Math, Physics| | 102 | Bob | Chemistry |
2. **1NF:** Separate courses into atomic values. | StudentID | Name | Course | |-----------|-------|-----------| | 101 | Alice | Math | | 101 | Alice | Physics | | 102 | Bob | Chemistry |

3. **2NF:** Ensure non-key attributes depend entirely on the primary key. | StudentID | Name | |-----------|-------| | 101 | Alice | | 102 | Bob |

StudentID	Course
101	Math
101	Physics
102	Chemistry

4. **3NF:** Remove transitive dependencies. | StudentID | Name | |-----------|-------| | 101 | Alice | | 102 | Bob |

CourseID	CourseName
C01	Math
C02	Physics
C03	Chemistry

StudentID	CourseID
101	C01
101	C02
102	C03

Instructor Signature (with date) **Supervisor's signature (with date)**

Keys: Fundamental Constraint

- In the CREATE TABLE statement, use:
  - PRIMARY KEY, UNIQUE

```
CREATE TABLE MovieStar (
    name CHAR(30) PRIMARY KEY,
    address VARCHAR(255),
    gender CHAR(1));
```

- Or, list at end of CREATE TABLE

```
PRIMARY KEY (name)
```

Keys...

- Can use the UNIQUE keyword in same way
  - …but for any number of attributes
  - foreign keys, which reference attributes of a second relation, only reference PRIMARY KEY
- Indexing Keys

CREATE UNIQUE INDEX YearIndex ON Movie(year)

- Makes insertions easier to check for key constraints

Referential Integrity Constraints

- 2 rules for Foreign Keys:

  Movies(MovieName, year)

  ActedIn(ActorName, MovieName)

1) Foreign Key must be a reference to a valid value in the referenced table.

2) … must be a PRIMARY KEY in the referenced table.

Declaring FK Constraints

- FOREIGN KEY <attributes> REFERENCES <table> (<attributes>)

```
CREATE TABLE ActedIn (
    Name CHAR(30) PRIMARY KEY,
    MovieName CHAR(30)
        REFERENCES
    Movies(MovieName));
```

- Or, summarize at end of CREATE TABLE

```
FOREIGN KEY MovieName
    REFERENCES
    Movies(MovieName)
```

- MovieName must be a PRIMARY KEY

How to Maintain?

- Given a change to DB, there are several possible violations:
  - Insert new tuple with bogus foreign key value
  - Update a tuple to a bogus foreign key value
  - Delete a tuple in the referenced table with the referenced foreign key value
  - Update a tuple in the referenced table that changes the referenced foreign key value

How to Maintain?

- Recall, ActedIn has FK MovieName...

  Movies(MovieName, year)

  (Fatal Attraction, 1987)

  ActedIn(ActorName, MovieName)

  (Michael Douglas, Fatal Attraction)

  insert: (Rick Moranis, Strange Brew)

How to Maintain?

- Policies for handling the change…
  - Reject the update (default)
  - Cascade (example: cascading deletes)
  - Set NULL
- Can set update and delete actions independently in CREATE TABLE

```
MovieName CHAR(30)
        REFERENCES
    Movies(MovieName))
                ON DELETE SET NULL
                ON UPDATE CASCADE
```

Constraining Attribute Values

- Constrain invalid values
  - NOT NULL
  - gender CHAR(1)
    CHECK (gender IN ('F', 'M'))
  - MovieName CHAR(30)
    CHECK (MovieName IN
    (SELECT MovieName FROM Movies))
- Last one not the same as REFERENCE
  - The check is invisible to the Movies table!

Constraining Values with User Defined 'Types'

- Can define new domains to use as the attribute type...

```
CREATE DOMAIN GenderDomain
    CHAR(1)
    CHECK (VALUE IN ('F', 'M'));
```

- Then update our attribute definition...

```
gender GenderDomain
```

More Complex Constraints...

- …Among several attributes in one table
  - Specify at the end of CREATE TABLE

CHECK (gender = ‘F’ OR name NOT LIKE ‘Ms.%’)

Declaring Assertions

- CREATE ASSERTION <name> CHECK (<condition>)

```
CREATE ASSERTION RichPres
   CHECK
   (NOT EXISTS
        (SELECT *
         FROM Studio, MovieExec
         WHERE presC# = cert#
              AND netWorth < 10000000))
```

Different Constraint Types

Type Guaranteed to hold?	Where Declared	When activated
Attribute not if CHECK subquery	with attribute	on insertion or update
Tuple not if CHECK subquery	relation schema	insertion or update to relation
Assertion Yes	database schema	on change to any relation mentioned

Giving Names to Constraints

Why give names? In order to be able to alter constraints.

Add the keyword CONSTRAINT and then a name:

```
ssn CHAR(50) CONSTRAINT  ssnIsKey PRIMARY KEY

CREATE DOMAIN ssnDomain  INT
  CONSTRAINT  ninedigits CHECK  (VALUE >= 100000000
                                 AND VALUE <= 999999999

CONSTRAINT  rightage
    CHECK  (age >= 0  OR status = “dead”)
```

Altering Constraints

```
ALTER TABLE  Product  DROP CONSTRAINT
positivePrice

ALTER TABLE  Product ADD CONSTRAINT
      positivePrice  CHECK  (price >= 0)

ALTER DOMAIN  ssn ADD CONSTRAINT no-
leading-1s
    CHECK  (value >= 200000000)

DROP ASSERTION  assert1.
```

NORMALIZATION

Objective

- Normalization presents a set of rules that tables and databases must follow to be well structured.
- Historically presented as a sequence of normal forms

First Normal From

- A table is in the first normal form iff
  - The domain of each attribute contains only ***atomic values***, and
  - The value of each attribute contains only a ***single value*** from that domain.

In layman's terms. it means every column of your table should only contain *single values*

Example

- For a library

Patron ID	Borrowed books
C45	B33, B44, B55
C12	B56

1-NF Solution

Patron ID	Borrowed book
C45	B33
C45	B44
C45	B33
C12	B56

Example

- For an airline

Flight	Weekdays
UA59	Mo We Fr
UA73	Mo Tu We Th Fr

1NF Solution

Flight	Week day
UA59	Mo
UA59	We
UA59	Fr
UA73	Mo
UA73	We
...	...

Implication for the ER model

- Watch for entities that can have multiple values for the same attribute
  - Phone numbers, …
- What about course schedules?
  - MW 5:30-7:00pm
    - Can treat them as *atomic time slots*

Functional dependency

Let X and Y be *sets* of attributes in a table *T*

- Y is **functionally dependent** on *X* in *T* **iff** for each set x $\in$ R.X there is precisely one corresponding set y$\in$ R.Y
- Y is **fully functional dependent** on *X* in *T* if *Y* is functional dependent on X and Y is not functional dependent on any proper subset of *X*

Example

- Book table

Book No	Title	Author	Year
B1	Moby Dick	H. Melville	1851
B2	Lincoln	G. Vidal	1984

Author attribute is:

- ***functionally dependent*** on the pair
  { BookNo, Title}
- ***fully functionally dependent*** on BookNo

Why it matters

- table BorrowedBooks

Book No	Patron	Address	Due
B1	J. Fisher	101 Main Street	3/2/15
B2	L. Perez	202 Market Street	2/28/15

Address attribute is

- ***functionally dependent*** on the pair
  { BookNo, Patron}
- ***fully functionally dependent*** on Patron

Problems

- Cannot insert new patrons in the system until they have borrowed books
  - ***Insertion anomaly***
- Must update all rows involving a given patron if he or she moves.
  - ***Update anomaly***
- Will lose information about patrons that have returned all the books they have borrowed
  - ***Deletion anomaly***

Armstrong inference rules (1974)

- ***Axioms:***
  - Reflexivity: if $Y \subseteq X$, then $X \rightarrow Y$
  - Augmentation: if $X \rightarrow Y$, then $WX \rightarrow WY$
  - Transitivity: if $X \rightarrow Y$ and $Y \rightarrow Z$, then $X \rightarrow Z$
- ***Derived Rules:***
  - Union: if $X \rightarrow Y$ and $X \rightarrow Z$, the $X \rightarrow YZ$
  - Decomposition: if $X \rightarrow YZ$, then $X \rightarrow Y$ and $X \rightarrow Z$
  - Pseudotransitivity: if $X \rightarrow Y$ and $WY \rightarrow Z$, then $XW \rightarrow Z$

Armstrong inference rules (1974)

- Axioms are both
  - ***Sound:***
    when applied to a set of functional dependencies they only produce dependency tables that belong to the transitive closure of that set
  - ***Complete:***
    can produce all dependency tables that belong to the transitive closure of the set

Armstrong inference rules (1974)

- Three last rules can be derived from the first three (the axioms)
- Let us look at the ***union rule***:
  if X→Y and X→Z, the X→YZ
- Using the first three axioms, we have:
  - if X→Y, then XX→XY same as X→XY (2nd)
  - if X→Z, then YX→YZ same as XY→YZ (2nd)
  - if X→XY and XY→YZ, then X→YZ (3rd)

Second Normal Form

- A table is in 2NF iff
  - It is in 1NF and
  - no non-prime attribute is dependent on any proper subset of any candidate key of the table
- A ***non-prime attribute*** of a table is an attribute that is not a part of any candidate key of the table
- A ***candidate key*** is a minimal superkey

Example

- Library allows patrons to request books that are currently out

Book No	Patron	PhoneNo
B3	J. Fisher	555-1234
B2	J. Fisher	555-1234
B2	M.	555-4321

Example

- Candidate key is {BookNo, Patron}
- We have
  - Patron → PhoneNo
- Table is not 2NF
  - Potential for
    - Insertion anomalies
    - Update anomalies
    - Deletion anomalies

2NF Solution

- Put telephone number in separate Patron table

Book No	Patron
B3	J. Fisher
B2	J. Fisher
B2	M.

Patron	PhoneNo
J. Fisher	555-1234
M. Ame	555-4321

Third Normal Form

- A table is in 3NF iff
  - it is in 2NF and
  - all its attributes are determined only by its candidate keys and not by any non-prime attributes

Example

- Table BorrowedBooks

Book No	Patron	Address	Due
B1	J. Fisher	101 Main Street	3/2/15
B2	L. Perez	202 Market Street	2/28/15

  - Candidate key is BookNo
  - Patron → Address

3NF Solution

- Put address in separate Patron table

Book No	Patron	Due
B1	J. Fisher	3/2/15
B2	L. Perez	2/28/15

Patron	Address
J. Fisher	101 Main Street
L. Perez	202 Market Street

Week 9-10

Sessional -08

Aggregate Functions

Objective: To summarize data using built-in SQL functions.

Details:

- **Functions:** `SUM`, `AVG`, `COUNT`, `MAX`, `MIN`.
- **GROUP BY:** Group data for aggregation.
- **HAVING:** Filter aggregated data.

Activities:

- Calculate the total sales for each product category.
- Find the average salary of employees in each department.

Sample Query:

```
SELECT DeptID, AVG(Salary) AS AvgSalary FROM Employee GROUP BY DeptID HAVING
AVG(Salary) > 50000;
```

Task:

- Write a query to calculate the maximum, minimum, and average price of products in a `Products` table.

Practice Problems:

1. Write a query to count the number of employees in each department.
2. Find the total and average marks of students grouped by subject.
3. Retrieve the highest and lowest sale amounts in a `Sales` table.

Instructor Signature (with date) **Supervisor's signature (with date)**

Aggregate Functions

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

  avg: average value
  min: minimum value
  max: maximum value
  sum: sum of values
  count: number of values

©Silberschatz, Korth and Sudarsha

Aggregate Functions (Cont.)

- Find the average account balance at the Perryridge branch.

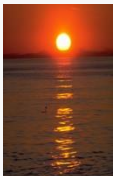
  select avg *(balance)*
  from *account*
  where *branch_name* = ‘Perryridge’

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

  select count (*)
  from *customer*

- Find the number of depositors in the bank.

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



Aggregate Functions – Group By

- Find the number of depositors for each branch.

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

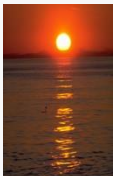
Note: Attributes in **select** clause outside of aggregate functions must
appear in **group by** list

Aggregate Functions – Having Clause

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

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

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



©Silberschatz, Korth and Sudarsha

Null Values

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

    select *loan_number*
    from *loan*
    where *amount* **is null**
- The result of any arithmetic expression involving *null* is *null*
  - Example: 5 + *null* returns null
- However, aggregate functions simply ignore nulls
  - More on next slide

Null Values and Three Valued Logic

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

©Silberschatz, Korth and Sudarsha

Null Values and Aggregates

- Total all loan amounts

  select sum (*amount*)
  from *loan*

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

Week 11-12

Experiment -09

Joins

Objective: To combine data from multiple tables.

Details:

- **INNER JOIN:** Retrieve matching rows.
- **LEFT JOIN:** Retrieve all rows from the left table.
- **RIGHT JOIN:** Retrieve all rows from the right table.
- **FULL JOIN:** Retrieve all matching rows and unmatched rows from both tables.

Activities:

- Practice joins on `Employee` and `Department` tables.

Sample Query:

```
SELECT e.EmpID, e.Name, d.DeptName
FROM Employee e
INNER JOIN Department d ON e.DeptID = d.DeptID;
```

Task:

- Write a query using a LEFT JOIN to retrieve all employees and their corresponding department names, even if the department is not assigned.

Practice Problems:

1. Write a query to retrieve students and their enrolled courses using a JOIN on `Students` and `Courses` tables.
2. Use a FULL JOIN to find records in both `Products` and `Sales` tables.
3. Perform an INNER JOIN on `Orders` and `Customers` tables to retrieve orders with customer details.

Let me know if you'd like more topics expanded or additional illustrations!

more topic

Instructor Signature (with date) **Supervisor's signature (with date)**

Week 13-14

Experiment -10

Views

Objective:
To simplify complex queries and abstract data by using virtual tables.

Details:

- **View:** A virtual table created by a `SELECT` statement. It doesn't store data physically but provides an abstraction for ease of querying.
- **Advantages:**
  - Simplifies complex queries.
  - Enhances security by restricting access to specific data.
  - Improves maintainability.

Activities:

- Create and manipulate views for specific datasets.
- Use views to abstract data and provide tailored information for different user groups.

Sample Query:

```
CREATE VIEW EmpDept AS
SELECT e.EmpID, e.Name, d.DeptName
FROM Employee e
INNER JOIN Department d ON e.DeptID = d.DeptID;
```

Task:

- Create a view that displays the total salary of employees by department.

Practice Problems:

1. Create a view to show products and their total sales.
2. Write a query to fetch data from a view and filter it by a specific range.
3. Update data through an updatable view and observe the impact on the base table.

Instructor Signature (with date) **Supervisor's signature (with date)**

Nested Subqueries

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

©Silberschatz, Korth and Sudarsha

Example Query

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

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

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

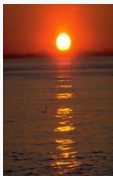
select distinct *customer_name*
from *borrower*
where *customer_name* **not in** (**select** *customer_name*
from *depositor*)

Example Query

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

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

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



©Silberschatz, Korth and Sudarsha

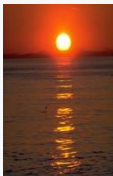
Set Comparison

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

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

- Same query using > **some** clause

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



Definition of Some Clause

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

(5 < **some** | 0 / 5 / 6 |) = true (read: 5 < some tuple in the relation)

(5 < **some** | 0 / 5 |) = false

(5 = **some** | 0 / 5 |) = true

(5 ≠ **some** | 0 / 5 |) = true (since 0 ≠ 5)

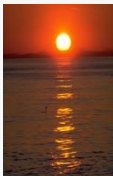
(= **some**) ≡ **in**

However, (≠ **some**) ≡ **not in**

Example Query

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

select *branch_name*
from *branch*
where *assets* > **all**
(select *assets*
from *branch*
where *branch_city* = ‘Brooklyn’)



©Silberschatz, Korth and Sudarsha

Definition of all Clause

- F <comp> **all** $r \Leftrightarrow \forall\ t \in r$ (F <comp> *t)*

(5 < **all** {0, 5, 6}) = false

(5 < **all** {6, 10}) = true

(5 = **all** {4, 5}) = false

(5 ≠ **all** {4, 6}) = true (since 5 ≠ 4 and 5 ≠ 6)

(≠ **all**) ≡ **not in**

However, (= **all**) ≡ **in**

Test for Empty Relations

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

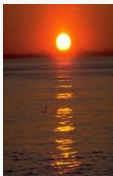
©Silberschatz, Korth and Sudarsha

Example Query

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

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

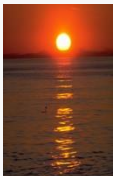
- Note that $X - Y = \emptyset \Leftrightarrow X \subseteq Y$
- *Note:* Cannot write this query using = **all** and its variants



Test for Absence of Duplicate Tuples

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

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

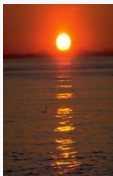


©Silberschatz, Korth and Sudarsha

Example Query

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

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



Derived Relations

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

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

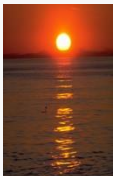
Note that we do not need to use the **having** clause, since we compute the temporary (view) relation *branch_avg* in the **from** clause, and the attributes of *branch_avg* can be used directly in the **where** clause.

©Silberschatz, Korth and Sudarsha

With Clause

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

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



Complex Query using With Clause

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

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

©Silberschatz, Korth and Sudarsha

Views

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

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

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

©Silberschatz, Korth and Sudarsha

View Definition

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

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

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

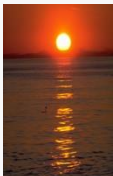
Example Queries

- A view consisting of branches and their customers

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

- Find all customers of the Perryridge branch

select *customer_name*
from *all_customer*
where *branch_name* = ‘Perryridge’



Views Defined Using Other Views

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

©Silberschatz, Korth and Sudarsha

View Expansion

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

  repeat
  Find any view relation v_i in e_1
  Replace the view relation v_i by the expression defining v_i
  until no more view relations are present in e_1

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

©Silberschatz, Korth and Sudarsha

Week 15

Experiment -11

Index

Objective:
To enhance the performance of database queries by creating and using indexes.

Details:

- **Index:** A database object used to optimize the speed of query execution.
- **Types of Indexes:**
  - **Single-column Index:** Speeds up searches on one column.
  - **Composite Index:** Optimizes searches on multiple columns.
  - **Unique Index:** Ensures no duplicate values in a column.
- **Trade-offs:** Improves query performance but may slow down insert and update operations.

Activities:

- Create indexes on large tables to improve query performance.
- Drop unnecessary indexes and analyze execution times.

Sample Query:

```
CREATE INDEX idx_EmployeeName ON Employee(Name);
```

Task:

- Add an index to the `Products` table for optimizing search queries by `ProductName`.

Practice Problems:

1. Create a composite index for the `Orders` table using `OrderDate` and `OrderStatus`.
2. Drop an index and analyze its effect on query performance.
3. Write a query to display all indexes created for a table.

Instructor Signature (with date) **Supervisor's signature (with date)**

Representing Data

- Attributes are represented in fixed or variable length collections called “fields”
- Fields in turn are put into fixed or variable length collections called records.
- Records are stored in physical blocks.
- A collection of records that forms a relation is stored as a collection of blocks called a file.
  - This file different than OS file. How?
  - Organization is different.
  - Extra indices to accommodate easy search and access.

Basic Concepts (indexing)

- Indexing works the same way as a catalog for a book in a library.
- Indexing needs to be efficient to allow fast access to records.
- Two types of indices:
  - ordered indices and
  - hash indices

Techniques and Evaluation

- Access types : types of accesses that are supported efficiently. Search by specific value or by range.
- Access time: Time sit takes to find a particular data or a set of data.
- Insertion time: Time it takes to insert a new item.
- Deletion time: Time it takes to delete an item.
- Space overhead : Additional space occupied by the index structure.

Ordered Indices

- To gain fast access to records in a file we can use an index structure.
- If the file containing the records is sequentially ordered, the index whose search key specifies the sequential order of the file is the primary key index.
- Primary key indices are also called clustering indices.

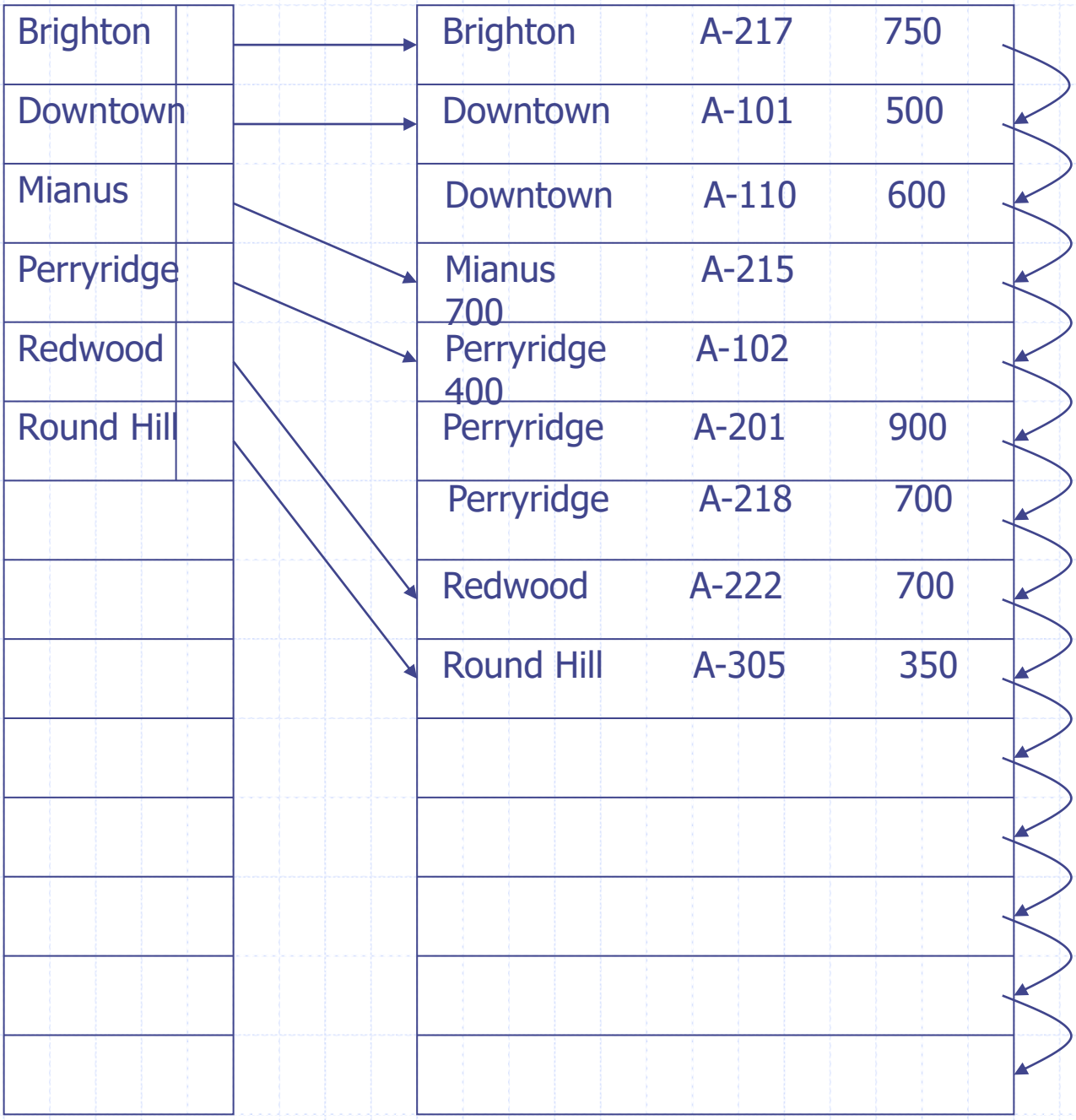
Primary Index

- Assume that all files are ordered sequentially on some search key.
- Such files, with primary key on the search key, are called index-sequential files.
- These files accommodate both sequential and random access to individual records.

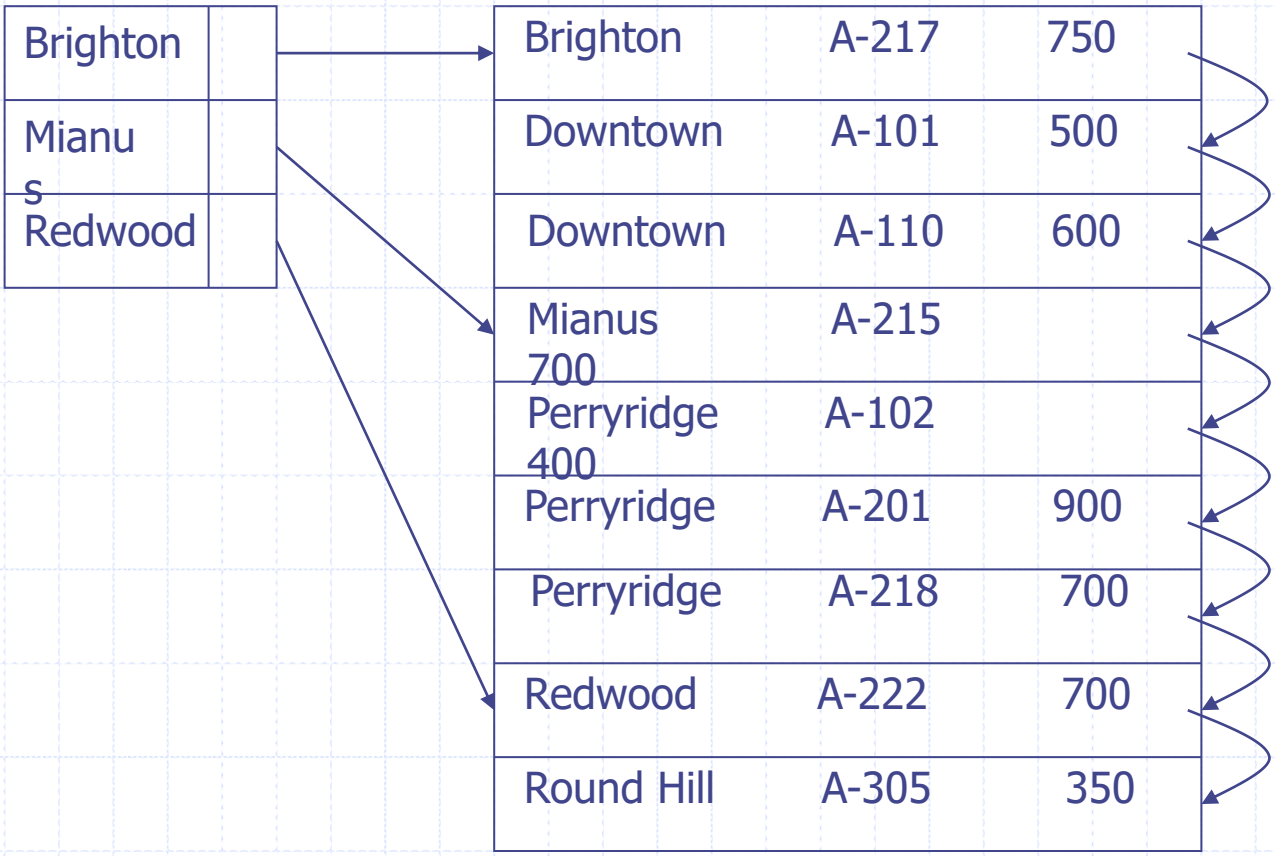
Dense and Sparse Index

- Dense index:
  - An index record appears for every search key value in the file.
  - The index record contains the search key and a pointer to the first data record with that search-key value.
- Sparse index:
  - An index is created only for a few values. Each index contains a value and pointer to first record that contains that value.

Dense Index


Brighton
Downtown
Mianus
Perryridge
Redwood
Round Hill
Brighton A-217 750
Downtown A-101 500
Downtown A-110 600
Mianus A-215 700
Perryridge A-102 400
Perryridge A-201 900
Perryridge A-218 700
Redwood A-222 700
Round Hill A-305 350

Sparse Index



Which one is better? Dense or sparse? It is a trade o
Between access time and space overhead.

Multi-level Indices

- Indices themselves may become too large for efficient processing.
- Example:
  - Consider file with 100000 records with 10 records in a block.
  - With sparse index and one index per block we have about 10,000 indices.
  - Assuming 100 indices fit into a block we need about 100 blocks.
  - It is desirable to keep the index file in the main memory.
  - Problem: Searching a large index file becomes expensive.

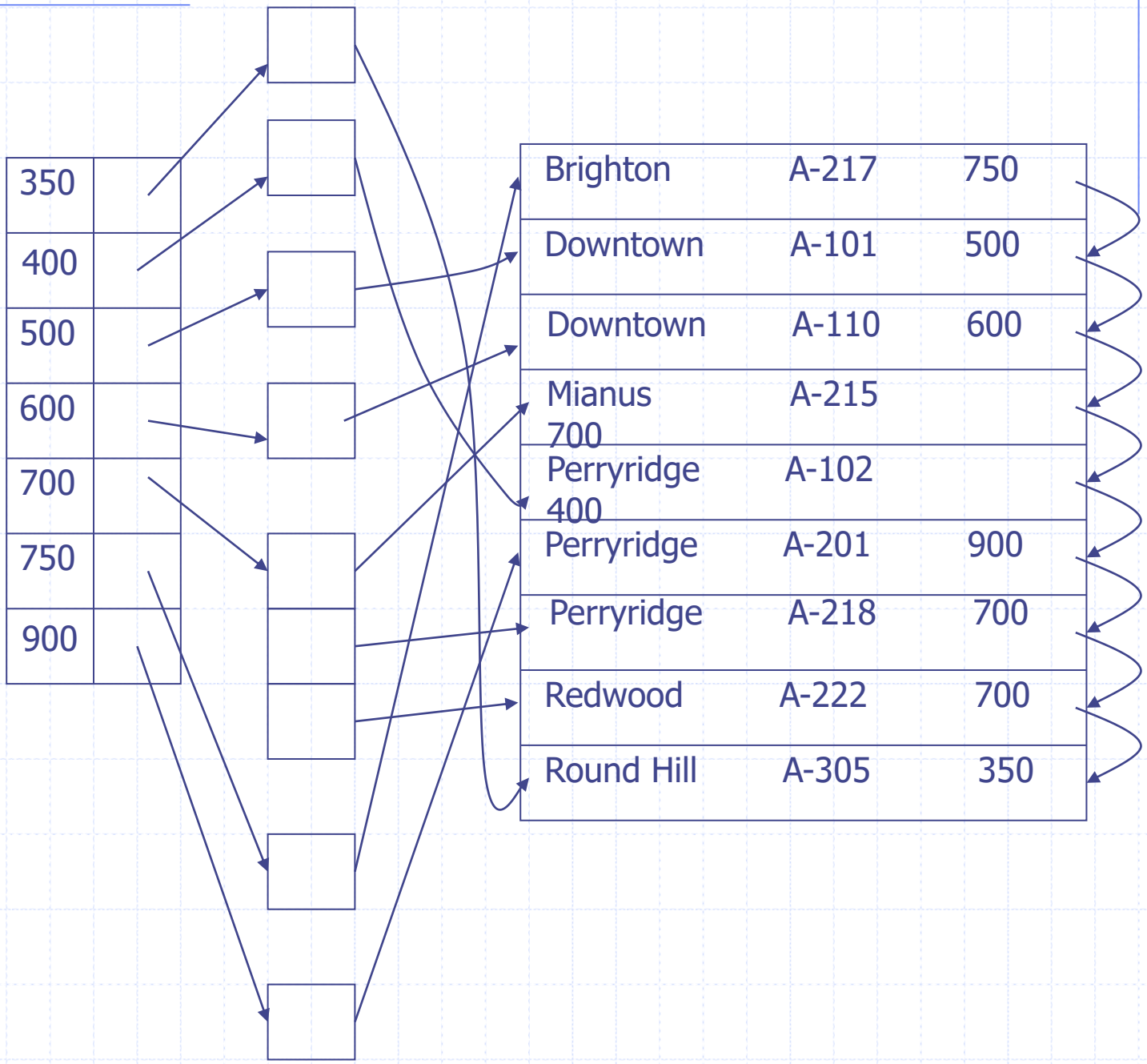
Multi-level Index

- Solution: Index the index file. We treat the index as we would treat any other sequential file and construct a sparse index on the primary index.
- We binary-search the outer level index to find the largest search key less than or equal to the one we desire.
- Two-level sparse index ; Figure 11.4

Secondary Index

- Secondary index is on attributes whose values are not stored sequentially.
- If the search key of a secondary index is not a candidate key, the index needs to be dense too.
- We can use an extra level of indirection with buckets at the second level.
- See fig.11.5

Secondary Index



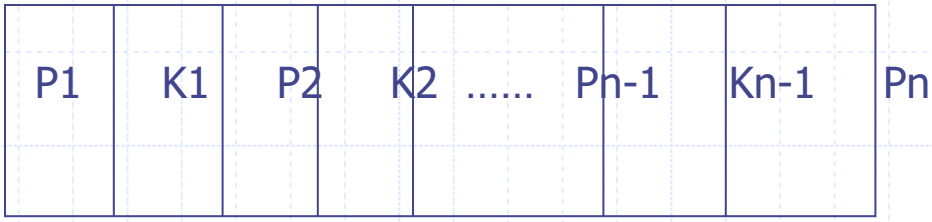
B+ Tree Index Files

- Main disadvantage of the index-sequential file organization is that performance degrades as the file grows both for index lookups and sequential scans.
- B+ tree index structure is most widely used of several index structures that maintain their efficiency despite insertion and deletion of data.

B+ Tree Index files

- A B+ index tree is a balanced tree in which every path from root to leaf is of same length and each non-leaf node has between ceiling(n/2) and n nodes where n is fixed.
- Typical node is a B+ tree:
  - n-1 search keys K1, K2,… Kn-1
  - n pointers P1, P2, …Pn

B+ Tree Node



B+ Tree (contd.)

- Structure of a B+ tree
- Queries on B+ trees
- Updates on B+ trees (insertion , deletion)
- B+ file organization
- B Tree variation of B+ tree : avoiding redundancy

Hashing

- Can we avoid the IO operations that the result from accessing the index file?
- Hashing offers a way.
- It also provides a way of constructing indices (which need nor be sequential).
- We will study static and dynamic hashing.

Hash File Organization

- Address of the disk block containing a desired record is computed using a function (hash function) and the search key.
- Let K denote set of all search keys, B denote set of all bucket addresses. Hash function h is a function that maps K to B.
- Bucket is typically a disk block.

Operations

- To insert a record with Ki as key, compute h(Ki) which gives the address of the bucket for the record. If there is space in the bucket then it is stored that bucket. (else chaining?)
- To lookup a record with key Ki, compute h(ki). Check with every record in the bucket to obtain the record.
- To delete a similar hash, find and delete is followed.

Hash Functions

- Hash function should be chosen so that
  - The distribution of records is uniform.
  - The distribution is random.
- Handling bucket overflows:
  - May occur due to insufficient number of buckets.
  - Due to bucket skew.
  - Solution: Overflow buckets, chaining, double hashing, linear probing, quadratic probing

Hash Indices

- Hashing can be used for organizing indices.Hash index organizes search keys with their associated pointers.
- See Fig.11.22
- Typically only secondary indices need to be organized using hashing.

Dynamic Hashing

- Many of today's databases grow very large in (a short) time.
- If you use static hash function we have three option:
  - Choose hash function based on current size,
  - Choose hash function based on anticipated size.
  - Periodically restructure the hash file in response to growth.
- Another solution: dynamic hashing.

Dynamic Hash Techniques

- Dynamic hash techniques allow the hash function to be modified dynamically to accommodate the growth and shrinkage of the database.
- It is also known as extendable hashing.
- Extendable hashing copes with the growth in the database size by splitting and coalescing buckets as the database grows and shrinks.

Week 16

Instructor Signature (with date) **Supervisor's signature (with date)**

Experiment -13

Exception Handling

Objective:
To handle runtime errors in SQL and PL/SQL efficiently.

Details:

- **Types of Exceptions:**
  - **Predefined Exceptions:** Standard errors like `NO_DATA_FOUND` and `ZERO_DIVIDE`.
  - **User-defined Exceptions:** Custom errors declared by the user.
- **Structure:** Use the `EXCEPTION` block to handle errors gracefully.

Activities:

- Write PL/SQL blocks to handle common runtime errors.
- Create user-defined exceptions for specific scenarios.

Sample Code:

```
DECLARE
    my_error EXCEPTION;
BEGIN
    IF 1 = 1 THEN
        RAISE my_error;
    END IF;
EXCEPTION
    WHEN my_error THEN
        DBMS_OUTPUT.PUT_LINE('Custom exception raised.');
END;
/
```

Task:

- Write a PL/SQL block to handle both predefined and user-defined exceptions.

Practice Problems:

1. Create a program to catch `NO_DATA_FOUND` errors.
2. Write a PL/SQL block to handle `ZERO_DIVIDE` exceptions.
3. Define a custom exception for invalid user input and handle it.

EXCEPTIONS

Bordoloi and
Bock

Errors

- Two types of errors can be found in a program: compilation errors and runtime errors.
- There is a special section in a PL/SQL block that handles the runtime errors.
- This section is called the *exception-handling section*, and in it, runtime errors are referred to as *exceptions*.
- The exception-handling section allows programmers to specify what actions should be taken when a specific exception occurs.

Exception Handling

- In order to handle run time errors in the program, an exception handler must be added.
- The exception-handling section has the following structure:
  - EXCEPTION
  - WHEN EXCEPTION_NAME
  - THEN
  - ERROR-PROCESSING STATEMENTS;
- The exception-handling section is placed after the executable section of the block.

- The section of the example in bold letters shows the exception-handling section of the block.
- When this example is executed with values of 4 and 0 for variables v_num1 and v_num2, respectively, the following output is produced:

Enter value for sv_num1:
4
old 2: v_num1 integer :=
&sv_num1;
new 2: v_num1 integer :=
4;
Enter value for sv_num2:
0
old 3: v_num2 integer :=
&sv_num2;
new 3: v_num2 integer :=
0;
A number cannot be divided by zero.
PL/SQL procedure successfully completed.

Example

```
DECLARE
  v_num1 integer :=
    &sv_num1;
  v_num2 integer :=
    &sv_num2;
  v_result number;
BEGIN
  v_result := v_num1 /
    v_num2;
  DBMS_OUTPUT.PUT_LINE
    ('v_result: '| |v_result);
EXCEPTION
  WHEN ZERO_DIVIDE
  THEN

    DBMS_OUTPUT.PUT_LI
    NE
    ('A number cannot be
    divided by zero.');
END;
```

Exception Handling

- This output shows that once an attempt to divide v_num1 by v_num2 was made, the exception-handling section of the block was executed.
- Therefore, the error message specified by the exception-handling section was displayed on the screen.
- This example illustrates several advantages of using an exception-handling section.
- You have probably noticed that the output looks cleaner. Even though the error message is still displayed on the screen, the output is more informative.
- In short, it is oriented more toward a user than a programmer.

Exception Handling

- In addition, an exception-handling section allows a program to execute to completion, instead of terminating prematurely.
- Another advantage offered by the exception-handling section is isolation of error-handling routines. In other words, all error-processing code for a specific block is located in the single section. As a result, the logic of the program becomes easier to follow and understand.
- Finally, adding an exception-handling section enables event-driven processing of errors.
- In case of a specific exception event, the exception-handling section is executed.

Exception Handling

- Just like in the example shown earlier, in case of the division by 0, the exception-handling section was executed.
- In other words, the error message specified by the DBMS_OUTPUT.PUT_LINE statement was displayed on the screen.

BUILT-IN EXCEPTIONS

- When a built-in exception occurs, it is said to be raised implicitly.
- In other words, if a program breaks an Oracle rule, the control is passed to the exception-handling section of the block.
- At this point, the error processing statements are executed.
- It is important for you to realize that after the exception-handling section of the block has executed, the block terminates.
- Control will not return to the executable section of this block.

Example

```
DECLARE
  v_student_name
    VARCHAR2(50);
BEGIN
 SELECT first_name||'
    '||last_name
 INTO v_student_name
 FROM student
 WHERE student_id = 101;
   DBMS_OUTPUT.PUT_LINE
   ('Student name
    is'||v_student_name);
EXCEPTION
  WHEN NO_DATA_FOUND
  THEN
     DBMS_OUTPUT.PUT_LINE
    ('There is no such
    student');
END;
```

- This example produces the following output:

  There is no such student
  PL/SQL procedure successfully completed.

- Because there is no record in the STUDENT table with student ID 101, the SELECT INTO statement does not return any rows.
- As a result, control passes to the exception-handling section of the block, and the error message "There is no such student" is displayed on the screen.
- Even though there is a DBMS_OUTPUT.PUT_LINE statement right after the SELECT statement, it will not be executed because control has been transferred to the exception-handling section.

BUILT-IN EXCEPTIONS

- Control will never return to the executable section of this block, which contains the DBMS_OUTPUT.PUT_LINE statement.
- While every Oracle runtime error has a number associated with it, it must be handled by its name in the exception-handling section.
- One of the outputs from the previous example has the following error message:

  ORA-01476: divisor is equal to zero

  where ORA-01476 stands for error number.
- This error number refers to the error named ZERO_DIVIDE.
- So, some common Oracle runtime errors are predefined in the PL/SQL as exceptions.

BUILT-IN EXCEPTIONS

- The list shown below explains some commonly used predefined exceptions and how they are raised:
- **NO_DATA_FOUND** This exception is raised when a SELECT INTO statement, which makes no calls to group functions, such as SUM or COUNT, does not return any rows.
- For example, you issue a SELECT INTO statement against STUDENT table where student ID equals 101.
- If there is no record in the STUDENT table passing this criteria (student ID equals 101), the NO_DATA_FOUND exception is raised.

TOO_MANY_ROWS

- This exception is raised when a SELECT INTO statement returns more than one row.
- By definition, a SELECT INTO can return only single row.
- If a SELECT INTO statement returns more than one row, the definition of the SELECT INTO statement is violated.
- This causes the TOO_MANY_ROWS exception to be raised.
- For example, you issue a SELECT INTO statement against the STUDENT table for a specific zip code.
- There is a big chance that this SELECT statement will return more than one row because many students can live in the same zip code area.

ZERO_DIVIDE

- This exception is raised when a division operation is performed in the program and a divisor is equal to zero.
- Previous example in the illustrates how this exception is raised.

LOGIN_DENIED

- This exception is raised when a user is trying to login on to Oracle with invalid username or password.

PROGRAM_ERROR

- This exception is raised when a PL/SQL program has an internal problem.

VALUE_ERROR

- This exception is raised when conversion or size mismatch error occurs.
- For example, you select student's last name into a variable that has been defined as VARCHAR2(5).
- If student's last name contains more than five characters, VALUE_ERROR exception is raised.

DUP_VALUE_ON_INDEX

- This exception is raised when a program tries to store a duplicate value in the column or columns that have a unique index defined on them.
- For example, you are trying to insert a record into the SECTION table for the course number "25," section 1.
- If a record for the given course and section numbers already exists in the SECTION table, DUP_VAL_ON_INDEX exception is raised because these columns have a unique index defined on them.

HANDLING DIFFERENT EXCEPTIONS

- So far, you have seen examples of the programs able to handle a single exception only.
- For example, a PL/SQL contains an exception-handler with a single exception ZERO_DIVIDE.
- However, many times in the PL/SQL block you need to handle different exceptions.
- Moreover, often you need to specify different actions that must be taken when a particular exception is raised.

```
DECLARE
  v_student_id NUMBER :=
    &sv_student_id;
  v_enrolled VARCHAR2(3) := 'NO';
BEGIN
   DBMS_OUTPUT.PUT_LINE
  ('Check if the student is enrolled');
SELECT 'YES'
INTO v_enrolled
FROM enrollment
 WHERE student_id = v_student_id;
    DBMS_OUTPUT.PUT_LINE
    ('The student is enrolled into one
     course');
EXCEPTION
  WHEN NO_DATA_FOUND
  THEN
    DBMS_OUTPUT.PUT_LINE('The
     student is not enrolled');
WHEN TOO_MANY_ROWS
  THEN
    DBMS_OUTPUT.PUT_LINE
   ('The student is enrolled into
     many courses');
END;
```

- This example contains two exceptions in the single exception handling section.
- The first exception, NO_DATA_FOUND, will be raised if there are no records in the ENROLLMENT table for a particular student.
- The second exception, TOO_MANY_ROWS, will be raised if a particular student is enrolled into more than one course.

OTHERS Handler

- You have seen examples of exception-handling sections that have particular exceptions, such as NO_DATA_FOUND or ZERO_DIVIDE.
- However, you cannot always predict beforehand what exception might be raised by your PL/SQL block.
- In cases like this, there is a special exception handler called OTHERS.
- All predefined Oracle errors (exceptions) can be handled with the help of the OTHERS handler.

Example

```
DECLARE
  v_instructor_id NUMBER :=
    &sv_instructor_id;
  v_instructor_name
    VARCHAR2(50);
BEGIN
 SELECT first_name||'
    '||last_name
 INTO v_instructor_name
 FROM instructor
 WHERE instructor_id =
    v_instructor_id;
   DBMS_OUTPUT.PUT_LINE
   ('Instructor name is'
    ||v_instructor_name);
EXCEPTION
  WHEN OTHERS
  THEN

    DBMS_OUTPUT.PUT_LINE
    ('An error has occurred');
END;
```

- When run, this example produces the following output:

```
Enter value for sv_instructor_id:
    100
old 2: v_instructor_id NUMBER
    := &sv_instructor_id;
new 2: v_instructor_id NUMBER
    := 100;
An error has occurred
PL/SQL procedure successfully
    completed.
```

- This demonstrates not only the use of the OTHERS exception handler, but also a bad programming practice.
- The exception OTHERS has been raised because there is no record in the INSTRUCTOR table for instructor ID 100.

EXCEPTION SCOPE

- The scope of an exception is the portion of the block that is covered by this exception.
- Even though variables and exceptions serve different purposes, the same scope rules apply to them.

Example

```
DECLARE
      v_student_id NUMBER := &sv_student_id;
      v_name VARCHAR2(30);
      v_total NUMBER(1);
-- outer block
BEGIN
      SELECT RTRIM(first_name)||' '||RTRIM(last_name)
      INTO v_name
      FROM student
      WHERE student_id = v_student_id;
      DBMS_OUTPUT.PUT_LINE('Student name is '||v_name);
        -- inner block
        BEGIN
                SELECT COUNT(*)
                INTO v_total
```

Example

```
                FROM enrollment
                WHERE student_id = v_student_id;
                DBMS_OUTPUT.PUT_LINE
                ('Student is registered for '||v_total||'
    course(s)');
      EXCEPTION
                WHEN VALUE_ERROR OR
    INVALID_NUMBER
                THEN
                DBMS_OUTPUT.PUT_LINE('An error has
                                      occurred');
      END;
EXCEPTION
      WHEN NO_DATA_FOUND
      THEN
      DBMS_OUTPUT.PUT_LINE('There is no such student');
END;
```

Example explained

- The inner block has structure similar to the outer block.
- It has a SELECT INTO statement and an exception section to handle errors.
- When VALUE_ERROR or INVALID_NUMBER error occurs in the inner block, the exception is raised.
- It is important that you realize that exceptions VALUE_ERROR and INVALID_NUMBER have been defined for the inner block only.
- Therefore, they can be raised in the inner block only.
- If one of these errors occurs in the outer block, this program will be unable to terminate successfully.

Example explained

- The exception NO_DATA_FOUND has been defined in the outer block; therefore, it is global to the inner block.
- This example will never raise the exception NO_DATA_FOUND in the inner block as it contains a group function in the SELECT statement.
- It is important to note that if you define an exception in a block, it is local to that block.
- However, it is global to any blocks enclosed by that block.
- In other words, in the case of nested blocks, any exception defined in the outer block becomes global to its inner blocks.

User Defined Exceptions

- Often in your programs you may need to handle problems that are specific to the program you write.
- For example, your program asks a user to enter a value for student_id. This value is then assigned to the variable v_student_id that is used later in the program.
- Generally, you want a positive number for an id. By mistake, the user enters a negative number.
- However, no error has occurred because student_id has been defined as a number, and the user has supplied a legitimate numeric value.
- Therefore, you may want to implement your own exception to handle this situation.

User Defined Exceptions

- This type of an exception is called a *user-defined exception* because it is defined by the programmer.
- Before the exception can be used, it must be declared.
- A user-defined exception is declared in the declarative part of a PL/SQL block as shown below:

  - DECLARE
  - exception_name EXCEPTION;

- Once an exception has been declared, the executable statements associated with this exception are specified in the exception-handling section of the block.
- The format of the exception-handling section is the same as for built-in exceptions.

Example

- DECLARE
- e_invalid_id EXCEPTION;
- BEGIN
- ...
- EXCEPTION
- WHEN e_invalid_id
- THEN
- DBMS_OUTPUT.PUT_LINE ('An id cannot be negative');
- END;

Raising Exception

- A user-defined exception must be raised explicitly.
- In other words, you need to specify in your program under which circumstances an exception must be raised as shown :

```
DECLARE
  exception_name
      EXCEPTION;
BEGIN
  …
 IF CONDITION
 THEN
     RAISE exception_name;
 ELSE
     …
  END IF;
EXCEPTION
   WHEN exception_name
    THEN
      ERROR-
     PROCESSING
       STATEMENTS;
END;
```

EXCEPTION PROPAGATION

- A runtime error may occur in the executable section, declaration section of the block or in the exception-handling section of the block.
- The rules that govern how exceptions are raised in these situations are referred to as *exception propagation.*

EXCEPTION PROPAGATION

- When a runtime error occurs in the executable section of the PL/SQL block, If there is an exception specified associated with a particular error, the control is passed to the exception-handling section of the block.
- Once the statements associated with the exception are executed, the control is passed to the host environment or to the enclosing block.
- If there is no exception handler for this error, the exception is propagated to the enclosing block (outer block).
- Then, the steps described above are repeated again.
- If no exception handler is found, the execution of the program halts, and the control is transferred to the host environment.

EXCEPTION PROPAGATION

- When a runtime error occurs in the declaration section of the block and if there is no outer block, the execution of the program halts, and the control is passed to the host environment.
- When a runtime error occurs in the declaration section of the PL/SQL block, the exception-handling section of this block will not be able to catch the error.
- When a runtime error occurs in the declaration section of the inner block, the exception immediately propagates to the enclosing (outer) block.

EXCEPTION PROPAGATION

- When a run time error occurs in the exception-handling section, just like in the previous case, if there is no outer block, the execution of the program halts, and the control is passed to the host environment.
- When a runtime error occurs in the exception-handling section of the PL/SQL block, the exception-handling section of this block is not able to prevent the error.
- When a runtime error occurs in the exception-handling section of the inner block, the exception immediately propagates to the enclosing block.

EXCEPTION PROPAGATION

- Only one exception can be raised in the exception-handling section of the block.
- Only after one exception has been handled, another can be raised, but two or more exceptions cannot be raised simultaneously.

Example

```
--outer block
DECLARE
    e_exception1 EXCEPTION;
    e_exception2 EXCEPTION;
BEGIN
    -- inner block
    BEGIN
      RAISE e_exception1;
    EXCEPTION
    WHEN e_exception1
    THEN
      RAISE e_exception2;
```

Example contd.

```
    WHEN e_exception2
    THEN
      DBMS_OUTPUT.PUT_LINE ('An error has
    occurred in the          inner'|| 'block');
    END;
EXCEPTION
WHEN e_exception2
THEN
    DBMS_OUTPUT.PUT_LINE ('An error has
    occurred in the program');
END;
```

Output

An error has occurred in the program
PL/SQL procedure successfully completed.

- Here two exceptions are declared: e_exception1 and e_exception2.
- The exception e_exception1 is raised in the inner block via statement RAISE.
- In the exception-handling section of the block, the exception e_exception1 tries to raise e_exception2.
- Even though there is an exception handler for the exception e_exception2, the control is transferred to the outer block.
- This happens because only one exception can be raised in the exception-handling section of the block.

RERAISING AN EXCEPTION

- On some occasions you may want to be able to stop your program if a certain type of error occurs.
- In other words, you may want to handle an exception in the inner block and then pass it to the outer block.
- This process is called *reraising an exception*. The following example illustrates this point.

```
-- outer block
DECLARE
      e_exception
      EXCEPTION;
BEGIN
      -- inner block
      BEGIN
        RAISE e_exception;
      EXCEPTION
      WHEN e_exception
      THEN
        RAISE;
      END;
EXCEPTION
WHEN e_exception
THEN
DBMS_OUTPUT.PUT_LINE
      ('An error has
      occurred');
END;
```

Output

The error has occurred

PL/SQL procedure successfully completed.

The exception, e_exception, is declared in the outer block.

It is raised in the inner block.

As a result, the control is transferred to the exception handling section of the inner block.

The statement RAISE in the exception-handling section of the block causes the exception to propagate to the exception-handling section of the outer block.

RERAISING AN EXCEPTION

- It is important to note that when an exception is reraised in the block that is not enclosed by any other block, the program is unable to complete successfully

```
DECLARE
    e_exception EXCEPTION;
BEGIN
    RAISE e_exception;
EXCEPTION
    WHEN e_exception
    THEN
    RAISE;
END;
```

Output

- DECLARE
- *
- ERROR at line 1:
- ORA-06510: PL/SQL: unhandled user-defined exception
- ORA-06512: at line 8

Week 17

Instructor Signature (with date) **Supervisor's signature (with date)**

Experiment -14

Triggers

Objective:
To automate database operations using triggers.

Details:

- **Trigger:** A stored procedure that automatically executes in response to specific events.
- **Types:**
  - **Before Triggers:** Execute before an operation (e.g., `BEFORE INSERT`).
  - **After Triggers:** Execute after an operation (e.g., `AFTER UPDATE`).
- **Applications:**
  - Enforce business rules.
  - Audit database changes.
  - Maintain log tables.

Activities:

- Create triggers to log changes in critical tables.
- Implement triggers for data validation before inserts.

Sample Code:

```
CREATE OR REPLACE TRIGGER log_changes
AFTER INSERT ON Employee
FOR EACH ROW
BEGIN
    INSERT INTO AuditLog (EmpID, ChangeDate)
    VALUES (:NEW.EmpID, SYSDATE);
END;
/
```

Task:

- Create a trigger to automatically update stock quantities after a sale.

Practice Problems:

Triggers

- A **trigger** is a statement that is executed automatically by the system as a side effect of a modification to the database.
- To design a trigger mechanism, we must:
  - Specify the conditions under which the trigger is to be executed.
  - Specify the actions to be taken when the trigger executes.
- Triggers introduced to SQL standard in SQL:1999, but supported even earlier using non-standard syntax by most databases.

  - Syntax illustrated here may not work exactly on your database system; check the system manuals

Trigger Example

- E.g. *time_slot_id* is not a primary key of *timeslot,* so we cannot create a foreign key constraint from *section* to *timeslot.*
- Alternative: use triggers on *section* and *timeslot* to enforce integrity constraints

```
create trigger timeslot_check1 after insert on section
referencing new row as nrow
for each row
when (nrow.time_slot_id not in (
                select time_slot_id
                from time_slot)) /* time_slot_id not present in
time_slot */
begin
    rollback
end;
```

Trigger Example Cont.

```
create trigger timeslot_check2 after delete on timeslot
    referencing old row as orow
    for each row
    when (orow.time_slot_id not in (
                select time_slot_id
                from time_slot)
                /* last tuple for time slot id deleted from time slot */
            and orow.time_slot_id in (
                select time_slot_id
                from section)) /* and time_slot_id still referenced
    from section*/
    begin
       rollback
    end;
```

Triggering Events and Actions in SQL

- Triggering event can be **insert**, **delete** or **update**
- Triggers on update can be restricted to specific attributes
  - **E.g., after update of** *takes* **on** *grade*
- Values of attributes before and after an update can be referenced
  - **referencing old row as : ** for deletes and updates
  - **referencing new row as : ** for inserts and updates
- Triggers can be activated before an event, which can serve as extra constraints. E.g. convert blank grades to null.

```
create trigger setnull_trigger before update of
takes
    referencing new row as nrow
    for each row
    when (nrow.grade = ' ')
     begin atomic
            set nrow.grade = null;
     end;
```

Trigger to Maintain credits_earned value

- **create trigger** *credits_earned* **after update of** *takes*
  on (*grade*)
  referencing new row as *nrow*
  referencing old row as *orow*
  for each row
  when *nrow.grade* <> 'F' **and** *nrow.grade* **is not null**
  and (*orow.grade* = 'F' **or** *orow.grade* **is null**)
  begin atomic
  update *student*
  set *tot_cred= tot_cred* +
  (**select** *credits*
  from *course*
  where *course.course_id= nrow.course_id*)
  where *student.id = nrow.id*;
  end;

Statement Level Triggers

- Instead of executing a separate action for each affected row, a single action can be executed for all rows affected by a transaction
  - Use **for each statement** instead of **for each row**
  - Use **referencing old table** or **referencing new table** to refer to temporary tables (called ***transition tables***) containing the affected rows
  - Can be more efficient when dealing with SQL statements that update a large number of rows

When Not To Use Triggers

- Triggers were used earlier for tasks such as
  - maintaining summary data (e.g., total salary of each department)
  - Replicating databases by recording changes to special relations (called **change** or **delta** relations) and having a separate process that applies the changes over to a replica
- There are better ways of doing these now:
  - Databases today provide built in materialized view facilities to maintain summary data
  - Databases provide built-in support for replication
- Encapsulation facilities can be used instead of triggers in many cases
  - Define methods to update fields
  - Carry out actions as part of the update methods instead of
    through a trigger

When Not To Use Triggers

- Risk of unintended execution of triggers, for example, when
  - loading data from a backup copy
  - replicating updates at a remote site
  - Trigger execution can be disabled before such actions.
- Other risks with triggers:
  - Error leading to failure of critical transactions that set off the trigger
  - Cascading execution

Oracle Example 1

A trigger that corrects mixed-case state names

```
CREATE OR REPLACE TRIGGER vendors_before_update_state
BEFORE INSERT OR UPDATE OF vendor_state
ON vendors
FOR EACH ROW
WHEN (NEW.vendor_state != UPPER(NEW.vendor_state))
BEGIN
  :NEW.vendor_state := UPPER(:NEW.vendor_state);
END;
/
```

Oracle Example 1

An UPDATE statement that fires the trigger

```
UPDATE vendors
SET vendor_state = 'wi'
WHERE vendor_id = 1;
```

A SELECT statement that shows the new row

```
SELECT vendor_name, vendor_state
FROM vendors
WHERE vendor_id = 1;
```

The result set

	VENDOR_NAME	VENDOR_STATE
1	US Postal Service	WI

Oracle Example 2

A trigger that validates line item amounts

```
CREATE OR REPLACE TRIGGER invoices_before_update_total
BEFORE UPDATE OF invoice_total
ON invoices
FOR EACH ROW
DECLARE
  sum_line_item_amount NUMBER;
BEGIN
  SELECT SUM(line_item_amt)
  INTO sum_line_item_amount
  FROM invoice_line_items
  WHERE invoice_id = :new.invoice_id;

  IF sum_line_item_amount != :new.invoice_total THEN
    RAISE_APPLICATION_ERROR(-20001,
      'Line item total must match invoice total.');
  END IF;
END;
/
```

Oracle Example 2

An UPDATE statement that fires the trigger

```
UPDATE invoices
SET invoice_total = 600
WHERE invoice_id = 100;
```

The response from the system

```
ORA-20001: Line item total must match invoice total.
```

Oracle Example 3

An AFTER trigger that inserts rows into the table

```
CREATE OR REPLACE TRIGGER invoices_after_dml
AFTER INSERT OR UPDATE OR DELETE
ON invoices
FOR EACH ROW
BEGIN
  IF INSERTING THEN
    INSERT INTO invoices_audit VALUES
    (:new.vendor_id, :new.invoice_number,
     :new.invoice_total, 'INSERTED', SYSDATE);
  ELSIF UPDATING THEN
    INSERT INTO invoices_audit VALUES
    (:old.vendor_id, :old.invoice_number,
     :old.invoice_total, 'UPDATED', SYSDATE);
  ELSIF DELETING THEN
    INSERT INTO invoices_audit VALUES
    (:old.vendor_id, :old.invoice_number,
     :old.invoice_total, 'DELETED', SYSDATE);
  END IF;
END;
/
```

Oracle Example 3

An INSERT statement that fires the trigger

```
INSERT INTO invoices VALUES
(115, 34, 'ZXA-080', '30-AUG-14', 14092.59, 0, 0, 3,
'30-SEP-14', NULL);
```

A DELETE statement that fires the trigger

```
DELETE FROM invoices WHERE invoice_number = 'ZXA-080';
```

A statement that retrieves the audit table rows

```
SELECT * FROM invoices_deleted;
```

The result set

	VENDOR_ID	INVOICE_NUMBER	INVOICE_TOTAL	ACTION_TYPE	ACTION_DATE
1	34	ZXA-080	14092.59	INSERTED	24-JUN-14
2	34	ZXA-080	14092.59	DELETED	24-JUN-14

Week 18

- Create a cursor to list students with grades above 75 in a `Marks` table.

Practice Problems:

1. Write a cursor to display employee details from the `Employee` table.
2. Use a cursor to calculate total sales for each product.
3. Develop a cursor to fetch and display student details in a loop.

Instructor Signature (with date) **Supervisor's signature (with date)**

Experiment -16

Transactions

Objective:
To ensure database consistency and integrity by grouping operations into transactions.

Details:

- **ACID Properties:**
  - **Atomicity:** All operations in a transaction are completed, or none are.
  - **Consistency:** Data remains in a valid state before and after a transaction.
  - **Isolation:** Transactions are executed independently of others.
  - **Durability:** Changes are permanent after a transaction is committed.
- **Commands:**
  - `COMMIT`: Save changes permanently.
  - `ROLLBACK`: Undo changes in a transaction.
  - `SAVEPOINT`: Create intermediate checkpoints in a transaction.

Activities:

- Write queries to implement transactions using `COMMIT` and `ROLLBACK`.
- Practice using `SAVEPOINT` to manage complex transactions.

Sample Code:

Outline

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

Transaction Concept

- A **transaction** is a *unit* of program execution that accesses and possibly updates various data items.
- E.g., transaction to transfer $50 from account A to account B:
  1. **read**(*A*)
  2. *A* := *A* – 50
  3. **write**(*A*)
  4. **read**(*B*)
  5. *B* := *B* + 50
  6. **write**(*B)*
- Two main issues to deal with:
  - Failures of various kinds, such as hardware failures and system crashes
  - Concurrent execution of multiple transactions

©Silberschatz, Korth and Sudarsha

Example of Fund Transfer

- Transaction to transfer $50 from account A to account B:
  1. **read**(*A*)
  2. *A* := *A* – 50
  3. **write**(*A*)
  4. **read**(*B*)
  5. *B* := *B* + 50
  6. **write**(*B*)
- **Atomicity requirement**
  - If the transaction fails after step 3 and before step 6, money will be "lost" leading to an inconsistent database state
    - Failure could be due to software or hardware
  - The system should ensure that updates of a partially executed transaction are not reflected in the database
- **Durability requirement** — once the user has been notified that the transaction has completed (i.e., the transfer of the $50 has taken place), the updates to the database by the transaction must persist even if there are software or hardware failures.

Example of Fund Transfer (Cont.)

- **Consistency requirement** in above example:
  - The sum of A and B is unchanged by the execution of the transaction
- In general, consistency requirements include
  - Explicitly specified integrity constraints such as primary keys and foreign keys
  - Implicit integrity constraints
    - e.g., sum of balances of all accounts, minus sum of loan amounts must equal value of cash-in-hand
  - A transaction must see a consistent database.
  - During transaction execution the database may be temporarily inconsistent.
  - When the transaction completes successfully the database must be consistent
    - Erroneous transaction logic can lead to inconsistency

Example of Fund Transfer (Cont.)

- **Isolation requirement** — if between steps 3 and 6, another transaction T2 is allowed to access the partially updated database, it will see an inconsistent database (the sum $A + B$ will be less than it should be).

T1	T2
1. **read**(A)	
2. $A := A - 50$	
3. **write**(A)	
	read(A), read(B), print(A+B)
4. **read**(B)	
5. $B := B + 50$	
6. **write**(B	

- Isolation can be ensured trivially by running transactions **serially**
  - That is, one after the other.
- However, executing multiple transactions concurrently has significant benefits, as we will see later.

ACID Properties

A **transaction** is a unit of program execution that accesses and possibly updates various data items. To preserve the integrity of data the database system must ensure:

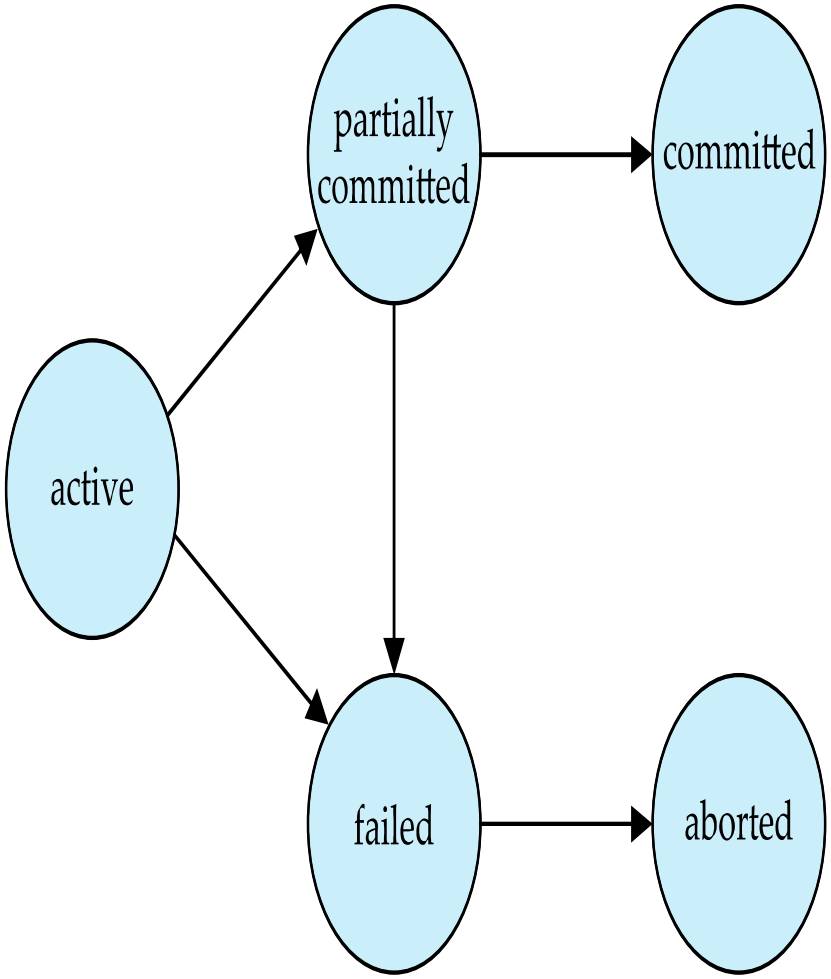
- **Atomicity.** Either all operations of the transaction are properly reflected in the database or none are.
- **Consistency.** Execution of a transaction in isolation preserves the consistency of the database.
- **Isolation.** Although multiple transactions may execute concurrently, each transaction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

Transaction State

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

 ©Silberschatz, Korth and Sudarsha

Transaction State (Cont.)



Concurrent Executions

- Multiple transactions are allowed to run concurrently in the system. Advantages are:
  - **Increased processor and disk utilization**, leading to better transaction *throughput*
    - E.g., one transaction can be using the CPU while another is reading from or writing to the disk
  - **Reduced average response time** for transactions: short transactions need not wait behind long ones.
- **Concurrency control schemes** – mechanisms to achieve isolation
  - That is, to control the interaction among the concurrent transactions in order to prevent them from destroying the consistency of the database
    - Will study in Chapter 15, after studying notion of correctness of concurrent executions.

Schedules

- **Schedule** – a sequences of instructions that specify the chronological order in which instructions of concurrent transactions are executed
  - A schedule for a set of transactions must consist of all instructions of those transactions
  - Must preserve the order in which the instructions appear in each individual transaction.
- A transaction that successfully completes its execution will have a commit instructions as the last statement
  - By default transaction assumed to execute commit instruction as its last step
- A transaction that fails to successfully complete its execution will have an abort instruction as the last statement

Schedule 1

- Let T_1 transfer \$50 from A to B, and T_2 transfer 10% of the balance from A to B.
- A serial schedule in which T_1 is followed by T_2 :

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

©Silberschatz, Korth and Sudarsha

Schedule 2

- A serial schedule where T_2 is followed by T_1

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

Schedule 3

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

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

- In Schedules 1, 2 and 3, the sum A + B is preserved.

Schedule 4

- The following concurrent schedule does not preserve the value of ($A + B$).

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

Serializability

- **Basic Assumption** – Each transaction preserves database consistency.
- Thus, serial execution of a set of transactions preserves database consistency.
- A (possibly concurrent) schedule is serializable if it is equivalent to a serial schedule. Different forms of schedule equivalence give rise to the notions of:
  1. **Conflict serializability**
  2. **View serializability**

Simplified view of transactions

- We ignore operations other than **read** and **write** instructions
- We assume that transactions may perform arbitrary computations on data in local buffers in between reads and writes.
- Our simplified schedules consist of only **read** and **write** instructions.

Conflicting Instructions

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

Conflict Serializability

- If a schedule *S* can be transformed into a schedule *S'* by a series of swaps of non-conflicting instructions, we say that *S* and *S'* are **conflict equivalent**.
- We say that a schedule *S* is **conflict serializable** if it is conflict equivalent to a serial schedule

Conflict Serializability (Cont.)

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

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

Schedule 3

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

Schedule 6

Conflict Serializability (Cont.)

- Example of a schedule that is not conflict serializable:

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

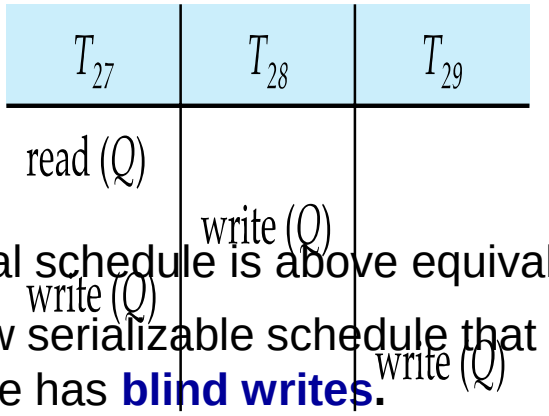
- We are unable to swap instructions in the above schedule to obtain either the serial schedule < T_3, T_4 >, or the serial schedule < T_4, T_3 >.

View Serializability

- Let *S* and *S'* be two schedules with the same set of transactions. *S* and *S'* are **view equivalent** if the following three conditions are met, for each data item *Q,*
  1. If in schedule S, transaction T_i reads the initial value of *Q*, then in schedule *S'* also transaction T_i must read the initial value of *Q.*
  2. If in schedule S transaction T_i executes **read**(*Q*), and that value was produced by transaction T_j (if any), then in schedule *S'* also transaction T_i must read the value of *Q* that was produced by the same **write**(Q) operation of transaction T_j .
  3. The transaction (if any) that performs the final **write**(*Q*) operation in schedule *S* must also perform the final **write**(*Q*) operation in schedule *S'.*
- As can be seen, view equivalence is also based purely on **reads** and **writes** alone.

View Serializability (Cont.)

- A schedule S is **view serializable** if it is view equivalent to a serial schedule.
- Every conflict serializable schedule is also view serializable.
- Below is a schedule which is view-serializable but *not* conflict serializable.



- What serial schedule is above equivalent to?
- Every view serializable schedule that is not conflict serializable has **blind writes**.

Other Notions of Serializability

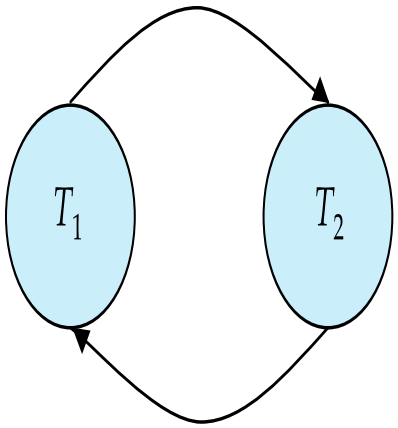
- The schedule below produces same outcome as the serial schedule < T_1, T_5 >, yet is not conflict equivalent or view equivalent to it.

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

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

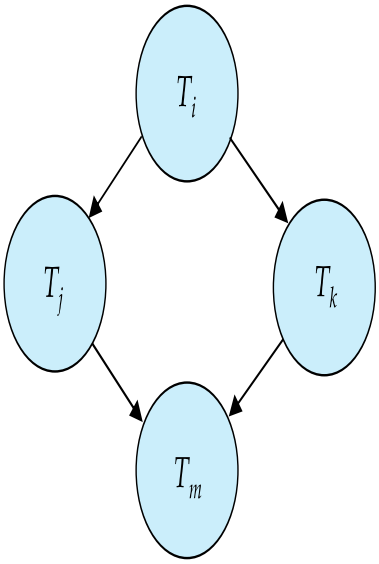
Testing for Serializability

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

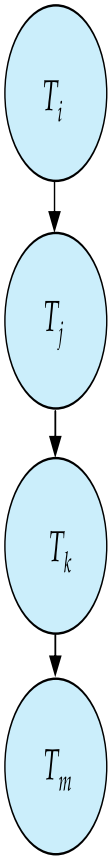


Test for Conflict Serializability

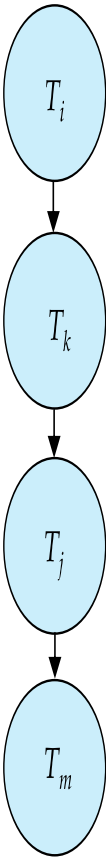
- A schedule is conflict serializable if and only if its precedence graph is acyclic.
- Cycle-detection algorithms exist which take order n^2 time, where n is the number of vertices in the graph.
  - (Better algorithms take order n + e where e is the number of edges.)
- If precedence graph is acyclic,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



(a)



(b)



(c)

Test for View Serializability

- The precedence graph test for conflict serializability cannot be used directly to test for view serializability.
  - Extension to test for view serializability has cost exponential in the size of the precedence graph.
- The problem of checking if a schedule is view serializable falls in the class of *NP*-complete problems.
  - Thus, existence of an efficient algorithm is *extremely* unlikely.
- However practical algorithms that just check some **sufficient conditions** for view serializability can still be used.

Recoverable Schedules

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

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

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

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

Cascading Rollbacks

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

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

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

- Can lead to the undoing of a significant amount of work

Cascadeless Schedules

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

Concurrency Control

- A database must provide a mechanism that will ensure that all possible schedules are
  - either conflict or view serializable, and
  - are recoverable and preferably cascadeless
- A policy in which only one transaction can execute at a time generates serial schedules, but provides a poor degree of concurrency
  - Are serial schedules recoverable/cascadeless?
- Testing a schedule for serializability *after* it has executed is a little too late!
- **Goal** – to develop concurrency control protocols that will assure serializability.

Concurrency Control (Cont.)

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

Concurrency Control vs. Serializability Tests

- Concurrency-control protocols allow concurrent schedules, but ensure that the schedules are conflict/view serializable, and are recoverable and cascadeless .
- Concurrency control protocols (generally) do not examine the precedence graph as it is being created
  - Instead a protocol imposes a discipline that avoids non-serializable schedules.
  - We study such protocols in Chapter 16.
- Different concurrency control protocols provide different tradeoffs between the amount of concurrency they allow and the amount of overhead that they incur.
- Tests for serializability help us understand why a concurrency control protocol is correct.

Weak Levels of Consistency

- Some applications are willing to live with weak levels of consistency, allowing schedules that are not serializable
  - E.g., a read-only transaction that wants to get an approximate total balance of all accounts
  - E.g., database statistics computed for query optimization can be approximate (why?)
  - Such transactions need not be serializable with respect to other transactions
- Tradeoff accuracy for performance

Levels of Consistency in SQL-92

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

Levels of Consistency

- Lower degrees of consistency useful for gathering approximate information about the database
- Warning: some database systems do not ensure serializable schedules by default
- E.g., Oracle (and PostgreSQL prior to version 9) by default support a level of consistency called snapshot isolation (not part of the SQL standard)

Transaction Definition in SQL

- In SQL, a transaction begins implicitly.
- A transaction in SQL ends by:
  - **Commit work** commits current transaction and begins a new one.
  - **Rollback work** causes current transaction to abort.
- In almost all database systems, by default, every SQL statement also commits implicitly if it executes successfully
  - Implicit commit can be turned off by a database directive
    - E.g., in JDBC -- connection.setAutoCommit(false);
- Isolation level can be set at database level
- Isolation level can be changed at start of transaction
    - E.g. In SQL **set transaction isolation level serializable**
    - E.g. in JDBC -- connection.setTransactionIsolation(

      Connection.TRANSACTION_SERIALIZABLE)

Implementation of Isolation Levels

- Locking
  - Lock on whole database vs lock on items
  - How long to hold lock?
  - Shared vs exclusive locks
- Timestamps
  - Transaction timestamp assigned e.g. when a transaction begins
  - Data items store two timestamps
    - Read timestamp
    - Write timestamp
  - Timestamps are used to detect out of order accesses
- Multiple versions of each data item
  - Allow transactions to read from a “snapshot” of the database

Transactions as SQL Statements

- E.g., Transaction 1:
  select *ID, name* **from** *instructor* **where** *salary* > 90000
- E.g., Transaction 2:
  insert into *instructor* **values** ('11111', 'James', 'Marketing', 100000)
- Suppose
  - T1 starts, finds tuples salary > 90000 using index and locks them
  - And then T2 executes.
  - Do T1 and T2 conflict? Does tuple level locking detect the conflict?
  - Instance of the **phantom phenomenon**
- Also consider T3 below, with Wu's salary = 90000
  update *instructor*
  set *salary* = *salary* * 1.1
  where *name* = 'Wu'
- Key idea: Detect "**predicate**" conflicts, and use some form of "**predicate locking**"

©Silberschatz, Korth and Sudarsha