🗒️

DBS MID I NOTES

Discussion - 18 August 2025

- **What is Data?**

Data = raw facts, figures, or observations.

Examples:

Student's name = "Ali"

Flight number = "PK-304"

Mobile location = "24.9° N, 67.1° E"

👉 Alone, data has little meaning. When processed, it becomes information.

- **What is Metadata?**

Metadata = "data about data."

It describes the structure, format, or meaning of data.

Examples:

In a Student database:

Data = "Ali, 20, CS"

Metadata = Student_Name (String), Age (Integer), Department (Varchar)

In a photo:

Data = the picture pixels

Metadata = Resolution, Date Taken, Camera Type

👉 Metadata is like a label/instruction manual for data.

- **Where Do We See Databases in Real Life?**

Everywhere!

Flight booking system ✈️ → passenger records, tickets, schedules

University portal 🎓 → student records, courses, grades

Mobile phone 📱 → call logs, contacts, GPS location

E-commerce apps 🛒 → products, prices, customer history

Banking 🏦 → transactions, balances, accounts

👉 Almost every digital service uses a database behind the scenes.

- **Frontend vs Backend vs Database**

Frontend → what the user sees (website, app UI, forms, dashboards).

Backend → application logic (Java, Python, C#, etc.), connects frontend to DB.

Database → stores the actual data (SQL, Oracle, MySQL, PostgreSQL).

📌 **Example:**
When booking a flight:

Frontend = you select date & flight on a website.

Backend = processes request, checks available seats.

Database = stores passenger info, seat availability, payments.

- **What is DBMS?**

Database Management System (DBMS) = software to create, manage, and interact with databases.

Examples: MySQL, Oracle, SQL Server, PostgreSQL.

Provides:

Efficient storage

Security & access control

Querying (SQL)

Backup & recovery

- **Introduction to Relational Database (RDBMS)**

A Relational Database organizes data into tables (rows & columns).

Each table represents an entity (like Student, Course, Flight).

Each row = a record (tuple)

Each column = a field (attribute)

Tables are related by keys (Primary Key, Foreign Key).

📌 **Example (Student Table):**

Student_ID	Name	Age	Department
101	Ali	20	CS
102	Sara	21	EE

Primary Key: Student_ID

Relation: Student_ID in Student table can connect to Student_ID in Grades table.

- **Why Use RDBMS?**

1. Structured, organized data
2. Easy to query using SQL
3. Reduces redundancy (no duplicate data)
4. Ensures consistency & reliability.

- **Main categories of SQL commands.**

1. **DDL (Data Definition Language)**

👉 Commands that define or change the structure of a database (tables, schemas, indexes).

Examples:

CREATE → create database objects (tables, views, schemas)

ALTER → modify structure of a table (add/remove columns)

DROP → delete a table, database, or view

TRUNCATE → remove all rows from a table but keep the structure

📌 Think of DDL as blueprints (designing and modifying database structure).

2. **DML (Data Manipulation Language)**

👉 Commands that manipulate or change the data inside tables.

Examples:

SELECT → retrieve data

INSERT → add new rows

UPDATE → modify existing rows

DELETE → remove rows

📌 DML = day-to-day data operations.

3. **DCL (Data Control Language)**

👉 Commands that control access and permissions in the database.

Examples:

GRANT → give privileges to users (e.g., read, write, execute)

REVOKE → take back given privileges

📌 DCL = security & access control.

4. **DRL (Data Retrieval Language)**

👉 Sometimes considered part of DML, but some books/courses separate it.

Mainly refers to SELECT (retrieving/querying data).

Includes use of clauses like **WHERE, GROUP BY, ORDER BY, HAVING, JOIN.**

📌 DRL = asking questions from your database.

Quick Analogy

DDL → Building the house (CREATE, ALTER, DROP).

DML → Living in the house (INSERT, UPDATE, DELETE).

DCL → Locking/unlocking rooms (GRANT, REVOKE).

DRL → Looking inside rooms to find stuff (SELECT).

In short,

DDL = structure

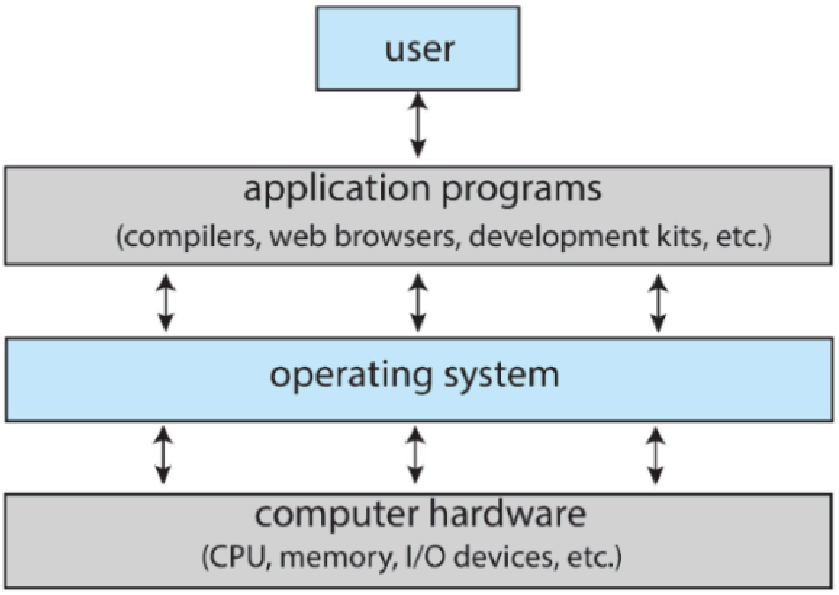
DML = data

DCL = permissions

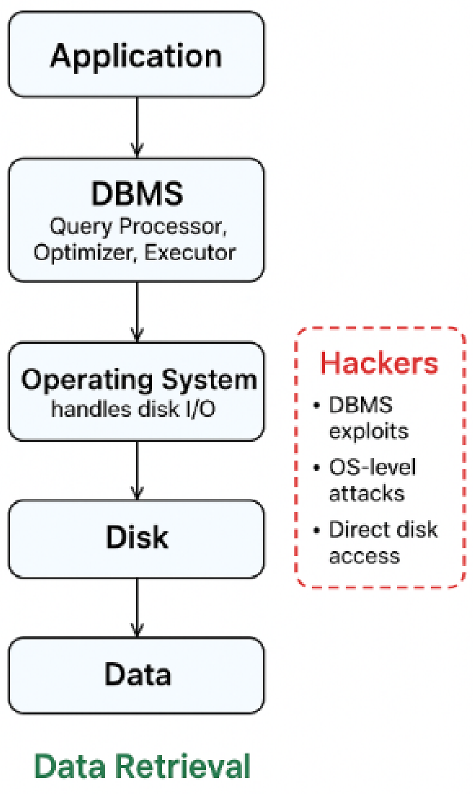
DRL = retrieval (SELECT).

Discussion - 19 August 2025

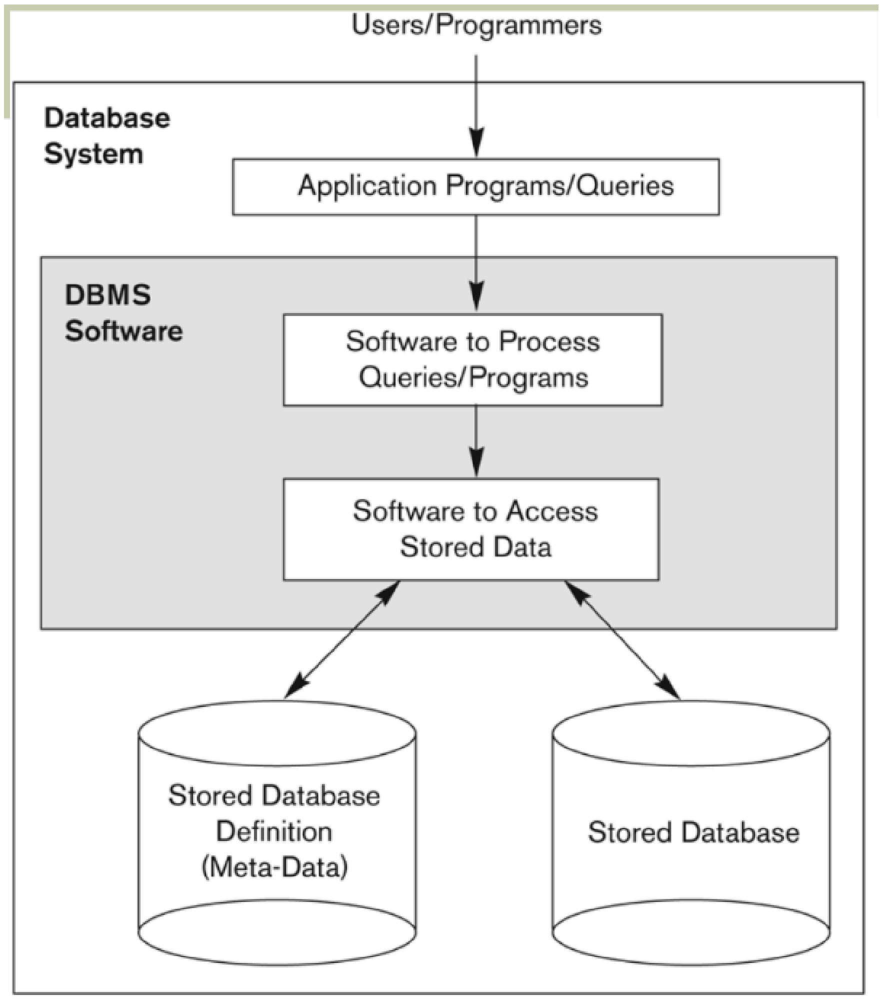
Recap of OS:


user
application programs
(compilers, web browsers, development kits, etc.)
operating system
computer hardware
(CPU, memory, I/O devices, etc.)

- **DBMS is software** → interacts with **OS** and **disk** to manage data.
- **Disk stores database files** → actual data lives here.
- **OS acts as a mediator** between DBMS(which is a software application) and hardware.
- **Data retrieval** involves DBMS query optimization + disk I/O.
- **Hackers target disks** because that's **where sensitive data resides**.
- **Encryption, access control, and backups** are critical for security.



- **What is Database System Environment?**


Users/Programmers
Database System
Application Programs/Queries
DBMS Software
Software to Process Queries/Programs
Software to Access Stored Data
Stored Database Definition (Meta-Data)
Stored Database

- **What is Database System?**

A database system is comprised of four components:

- Users
- Database Application
- Database Management System (DBMS)
- Database

- **What is Three Schema Architecture?**

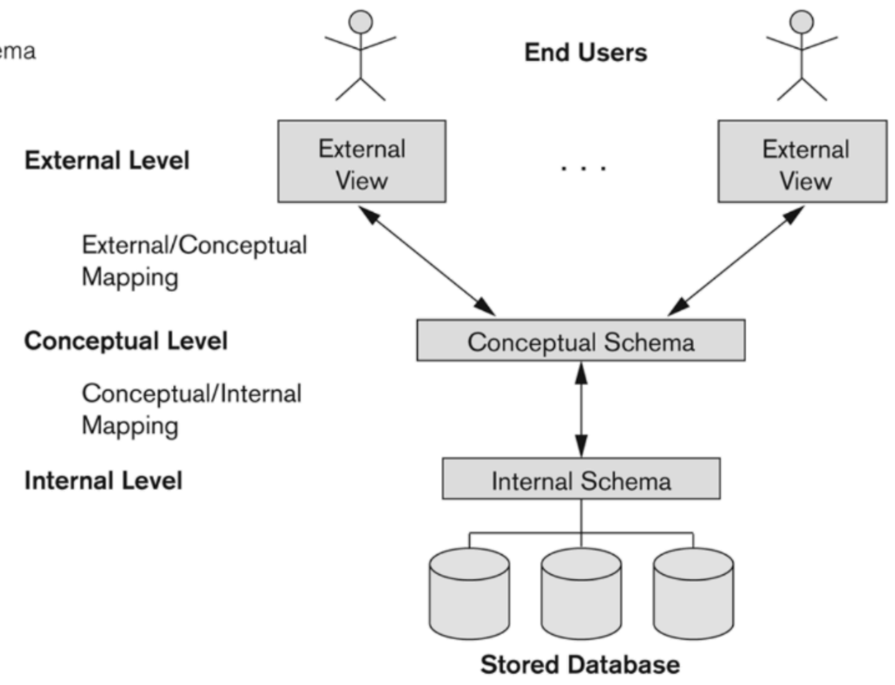


Figure 2.2
The three-schema architecture.

- **What is Two-tier vs Three-tier?**

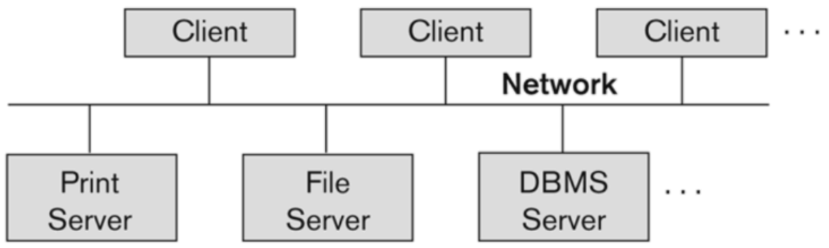
Two-Tier Architecture

- **Structure:**
  1. **Client (Tier 1):** Runs the application (UI + business logic).
  2. **Database Server (Tier 2):** Stores and manages the database (DBMS).
- **How it works:**
  - The client directly communicates with the database.
  - Application logic may reside partly on the client and partly in the database (stored procedures).
- **Advantages:**
  - Simple, fast for small systems.
  - Easy to implement.
- **Disadvantages:**
  - Poor scalability (too many clients overload DB server).
  - Security risk (direct client–DB connection).

Example:

A desktop application (like MS Access) directly connected to an Oracle/MySQL database.

Figure 2.5
Logical two-tier client/server architecture.



Three-Tier Architecture

- **Structure:**
  1. **Client (Tier 1 – Presentation Layer):** User interface (web browser, mobile app, desktop app).
  2. **Application Server (Tier 2 – Business Logic Layer):** Handles application logic, processing, rules, authentication.
  3. **Database Server (Tier 3 – Data Layer):** Stores and manages data.
- **How it works:**
  - Client sends request → Application Server processes → Database responds → Result goes back via Application Server.
- **Advantages:**
  - Better security (clients never directly access DB).
  - Scalability (app server can handle many users).
  - Easier maintenance (logic separated from UI and DB).
- **Disadvantages:**
  - More complex to design and maintain.
  - Slightly higher latency than two-tier.

Example:

Web application:

- Browser (client) → Django/Java/.NET app server → MySQL/Oracle DB server.

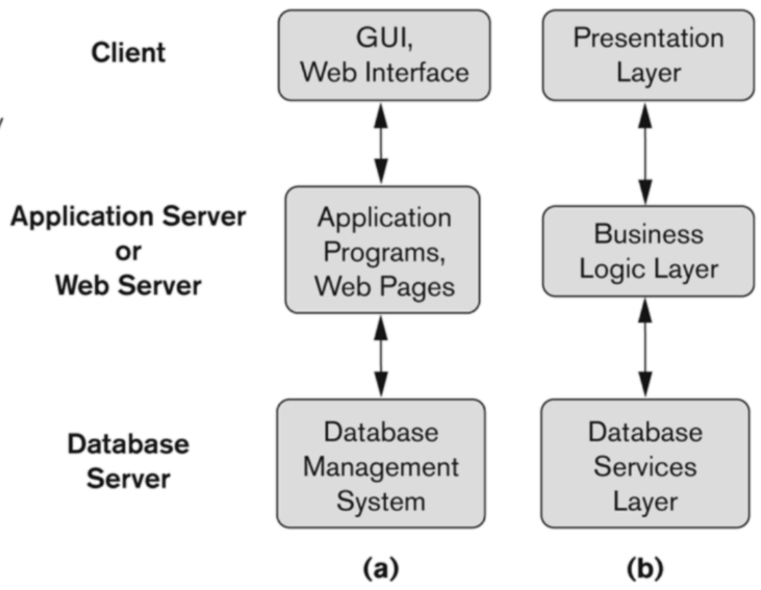


Figure 2.7
Logical three-tier client/server architecture, with a couple of commonly used nomenclatures.

In short:

- **Two-Tier:** Client ↔ Database.
- **Three-Tier:** Client ↔ Application Server ↔ Database.

PAST PAPER QS:

Question: Consider the following Database schema:

Company (name, city, country)
Dancer (did, name, birthyear, country)
Show (sid, title, choreographer, composer, year)
Role (did, sid, role, company)
Where,
• Company stores information about dance companies. The attributes name, city, and country are all string; we assume that all companies have unique names.
• Dancer stores information about individual dancers. did is a unique integer id for each dancer. name is a string with the dancer's name, birthyear is an integer, and the dancer's native country is a string.
• Show stores information about ballet shows (dances). Each show has a unique integer id sid, string attributes for the show title, choreographer, and composer, and an integer year in which the show was created.
• Role stores information about which dancers have been in which shows, the name of the role (part) they danced, and the company where they danced that part in that particular show. The dancer and show id's are integers, the role and company names are strings. A dancer may have danced multiple roles in the same show at the same company, or danced the same role in the same show for different companies, and so forth.
• Several attributes in Role are foreign keys: did references did in Dancer, sid references sid in

Show, and company references name in Company.
Write the SQL queries for each of the following:
a. Write the CREATE TABLE command needed to create the Dancer and Role tables.
b. Write the SELECT query to List the dancer ids (did) and names of all dancers born on or before 1950 and who have danced in at least three different shows. If a dancer has danced different roles in the same show, it still only counts once in the total number of shows. Each dancer/did pair should only be listed once.

Company (name, city, country) → All are STRING , name UNIQUE
Dancer (did, name, birthyear, country) → did is UNIQUE , Name String, birthyear INT, country STRING
Show (sid, title, choreographer, composer, year) → sid INT UNIQUE, year INT, rest are STRINGS
Role (did, sid, role, company) , did and sid INT, role and company are STRING

did FOREIGN KEY REFERENCES Dancer(did)

sid FOREIGN KEY REFERENCES Show(sid)

company FOREIGN KEY REFERENCES Company(name)

```
CREATE TABLE Company(
        name VARCHAR(30) PRIMARY KEY,
        city VARCHAR(30),
        country VARCHAR(30)
        );

CREATE TABLE Dancer(
        did INT PRIMARY KEY,
        name VARCHAR(30),
        birthyear INT,
        country VARCHAR(30)
        );

CREATE TABLE Show(
        sid INT PRIMARY KEY,
        title VARCHAR(30),
        choreographer VARCHAR(30),
        composer VARCHAR(30),
        year INT
        );

CREATE TABLE Role(
        sid INT,
        did INT,
        role VARCHAR(30),
        company VARCHAR(30),
        FOREIGN KEY(sid) REFERENCES Show(sid),
        FOREIGN KEY(did) REFERENCES Dancer(did),
        FOREIGN KEY(company) REFERENCES Company(name),
        PRIMARY KEY(sid,did,company)
        );

SELECT d.did,
       d.name
FROM Dancer d
JOIN Role   r
ON d.did = r.did
WHERE d.birthyear <= 1950
GROUP BY d.did, d.name
HAVING COUNT(DISTINCT r.sid) >= 3;
```

Question: In some small organization with 200 employees', attendance is collected daily through hand punch machines, which is then collected by HR department on monthly basis to generate salaries. Once salaries for a month have been drawn, the monthly attendance is of no more use for that small organization and therefore flushed (deleted). Identify the main concerns (drawbacks) of this system and discuss whether or not it will be beneficial to
go for DBMS in the light of these concerns.

Cost, domain of application, concurrency, usage and security are the main concerns on which we decide the system to be developed using DBMS or not. As it is already mentioned in the statement that the data of the hand punch machine is only useful till the salary is not being generated. After that they flush the data. In that case we gave priority to a file base system on any DBMS.

Question: Unilever Pakistan maintains their record of Products sale in File Based System at Regional Offices. Now they want to develop a Mobile App to monitor the production and to compete the demand and supply issue in market for their products. Briefly explain how the DBMS will prove beneficial with respect to their business demands, also discuss the issue(s) in their current record keeping system?

Benefits / Advantages of Dbms

1. Controlling Redundancy
2. Restricting Unauthorized Access
3. Providing Persistent Storage for Program Objects
4. Providing Storage Structures and Search Techniques for Efficient Query Processing
5. Providing Backup and Recovery
6. Providing Multiple User Interfaces

7. Representing Complex Relationships among Data
8. Enforcing Integrity Constraints

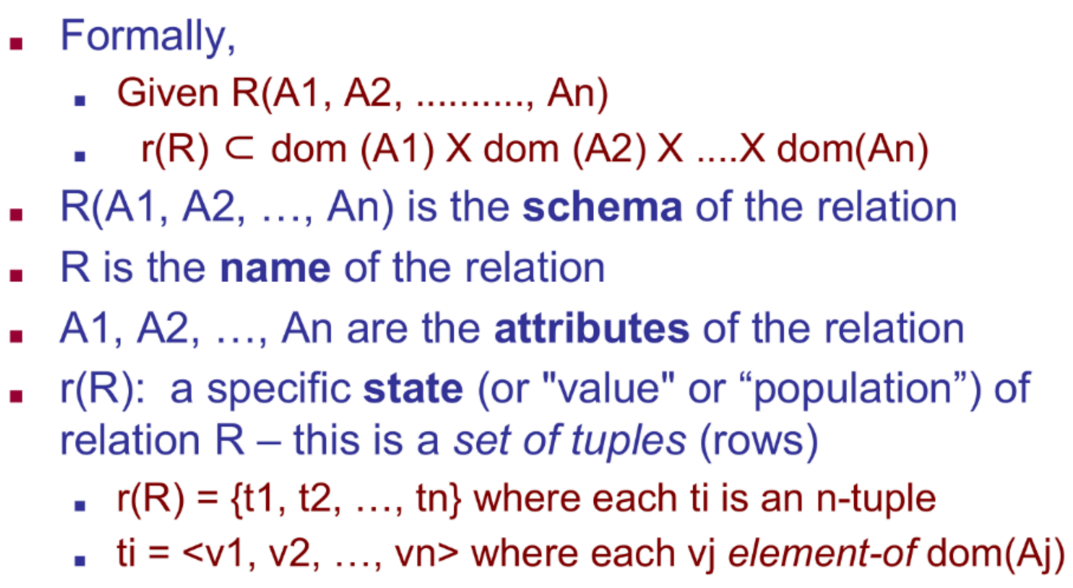
Demerits/ Limitation of File Based System

1. Separation & Isolation of Data.
2. Duplication of records /data.
3. Data Security Issue.
4. File accessing Issue
5. Fixed Queries.

Question: Define Data, Information, Database, DBMS and Database System?

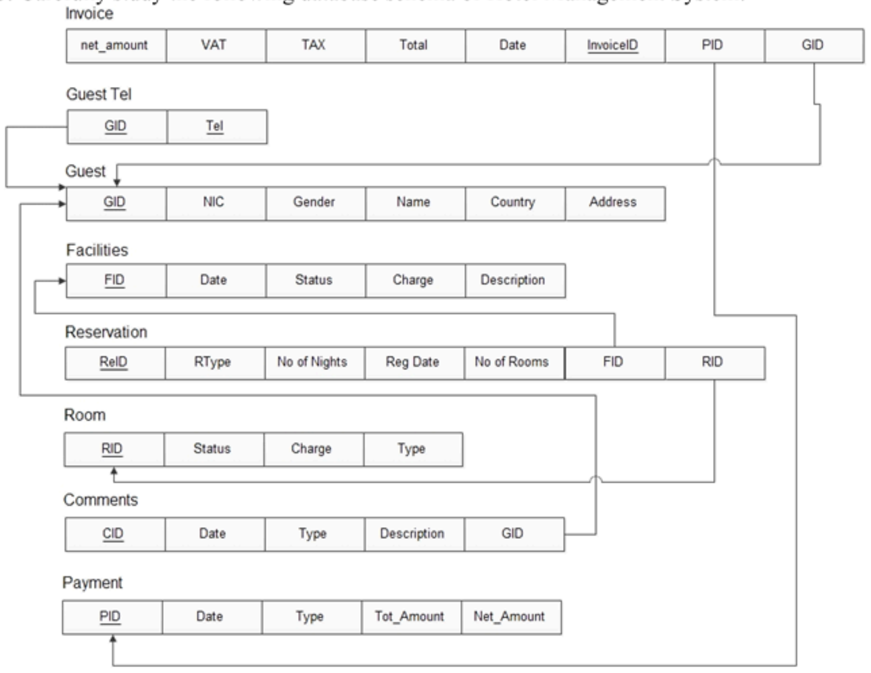
Data: Data is raw, unorganized facts that need to be processed
Information: When data is processed, organized, structured or presented in a given context so as to make it useful, it is called information.
Database: A database is a collection of related data.
Database Management System: A software package/ system to facilitate the creation and maintenance of a computerized database.
Database System: The DBMS software together with the data itself. Sometimes, the applications are also included.

Question: Describe the relationship between mathematical relations and relations in the relational data model?



- Formally,
  - Given R(A1, A2, ..., An)
  - $r(R) \subset dom(A1) \times dom(A2) \times \ldots \times dom(An)$
- R(A1, A2, …, An) is the **schema** of the relation
- R is the **name** of the relation
- A1, A2, …, An are the **attributes** of the relation
- r(R): a specific **state** (or "value" or “population”) of relation R – this is a *set of tuples* (rows)
  - r(R) = {t1, t2, …, tn} where each ti is an n-tuple
  - ti = <v1, v2, …, vn> where each vj *element-of* dom(Aj)

Question 6: Carefully study the following database schema of Hotel Management System:


Invoice
net_amount | VAT | TAX | Total | Date | InvoiceID | PID | GID
Guest Tel
GID | Tel
Guest
GID | NIC | Gender | Name | Country | Address
Facilities
FID | Date | Status | Charge | Description
Reservation
ReID | RType | No of Nights | Reg Date | No of Rooms | FID | RID
Room
RID | Status | Charge | Type
Comments
CID | Date | Type | Description | GID
Payment
PID | Date | Type | Tot_Amount | Net_Amount

NOTE: The possible values (domain) for the RType attribute in the Reservation table are: (1, 2, 3), where 1 means Pending, 2 means Guaranteed, and 3 means cancelled. [3 marks]

Answer the following questions, in the order specified:

a. Write an SQL statement to add a new attribute named *GID* in the Reservation table, and create it as foreign key constraint to refer to GID attribute in the Guest table, so that every reservation should hold a guest ID.

b. After adding a new attribute as in part (a) above, now Insert a new record with the following values in the reservation table: *(R-123, 1, 2, 29-09-2018, 1, NULL, R-109, G123)*

c. List all the possible integrity constraint violations which may occur during the insert operation in part (b) above.

a.

```
Alter table Reservation
Add (GID varchar(10),
Constraint Guest_ID
Foreign Key (GID) References Guest (GID));
```

b.

```
Insert into Reservation values(R-123, 1, 2, 29-09-2018, 1, NULL, R-109, G123);
```

c.

Key Constraint violation (If primary key already exists)

Entity Integrity Constraint Violation (If primary key is NULL)

Referential Integrity Constraint Violation (If Inserted RID doesn't belong to PK)
Domain Constraint Violation

Question: Compare and contrast the two-tier client-server architecture for traditional DBMSs with the three-tier client-server architecture. Why is the three-tier architecture more appropriate for the Web?

Two-Tier Architecture (Traditional DBMS)

- **Structure:**
    - **Client ↔ Database Server**
- **Client Role**: Handles both **presentation** and **application logic**.
- **Server Role**: Manages **data storage and retrieval**.
- **Pros**: Simple, fast for small-scale applications.
- **Cons**:
    - Tight coupling between client and server.
    - Poor scalability.
    - Difficult to maintain or update logic across multiple clients.

🏗️ **Three-Tier Architecture (Modern/Web-Based)**

- **Structure**:
    - **Client (Presentation Layer) ↔ Application Server (Business Logic Layer) ↔ Database Server (Data Layer)**
- **Client Role**: Handles **UI/UX** only.
- **Application Server**: Processes business rules and logic.
- **Database Server**: Stores and manages data.
- **Pros**:
    - Better **scalability**, **security**, and **maintainability**.
    - Easier to update logic without touching client code.
    - Supports multiple client types (web, mobile, desktop).

Why Three-Tier Is Better for the Web

- **Separation of concerns**: Keeps UI, logic, and data independent.
- **Scalability**: Can handle thousands of users via load-balanced servers.
- **Security**: Sensitive logic and data access are kept away from the client.
- **Flexibility**: Supports diverse platforms and devices.

5.1 Define the following terms:

- **Domain:** The set of permissible values for an attribute (e.g., Age domain = integers 0–120).
- **Attribute:** A column in a relation; describes a property of an entity.
- **n-tuple (or tuple):** A row in a relation; represents a record with values for each attribute.
- **Relation schema:** The structure of a relation, including its name and attributes (e.g., STUDENT(SSN, Name, Major)).
- **Relation state:** The actual content (set of tuples) in a relation at a given time.
- **Degree of a relation:** The number of attributes in a relation schema.
- **Relational database schema:** The collection of relation schemas (the overall logical design).
- **Relational database state:** The collection of all current relation states (the actual data at a moment).

5.2 Why are tuples in a relation not ordered?

Because a relation is defined mathematically as a **set of tuples**, and sets are unordered. Order has no meaning unless explicitly specified in a query.

5.3 Why are duplicate tuples not allowed in a relation?

Relations are sets, and sets do not allow duplicates. This ensures that each tuple is unique and represents distinct information.

5.4 What is the difference between a key and a superkey?

- **Superkey:** Any set of attributes that uniquely identifies a tuple.
- **Key (Candidate Key):** A minimal superkey (no attribute can be removed without losing uniqueness).

Example: In STUDENT(SSN, Name),

- `{SSN, Name}` is a superkey.
- `{SSN}` is a key (minimal).

5.5 Why do we designate one candidate key as the primary key?

Because multiple candidate keys may exist, but the **primary key** is chosen as the main identifier for tuples. It provides consistency and is used in constraints and foreign keys.

5.6 Characteristics of relations that make them different from ordinary tables and files:

1. No duplicate tuples.
2. No ordering of tuples or attributes.
3. Every tuple has exactly one value for each attribute (unless NULL).
4. Attributes have atomic (indivisible) values.
5. A schema defines structure and constraints clearly.

5.7 Reasons for NULL values in relations:

1. Value is **unknown** (not yet available).
2. Value is **not applicable** (e.g., MiddleName for someone with none).
3. Value is **undefined** (not recorded).

5.8 Entity integrity and referential integrity constraints:

- **Entity Integrity:** Primary key attributes cannot be NULL; ensures each tuple is uniquely identifiable.
- **Referential Integrity:** A foreign key must match a primary key in another relation or be NULL; ensures consistency across relations.

  👉 Important because they prevent **orphan records**, maintain **data accuracy**, and enforce **logical consistency.**

5.9 Define foreign key. What is it used for?

- **Foreign key:** An attribute (or set of attributes) in one relation that refers to the primary key of another relation.
- **Use:** To represent **relationships** between relations (e.g., ENROLL.SSN → STUDENT.SSN).

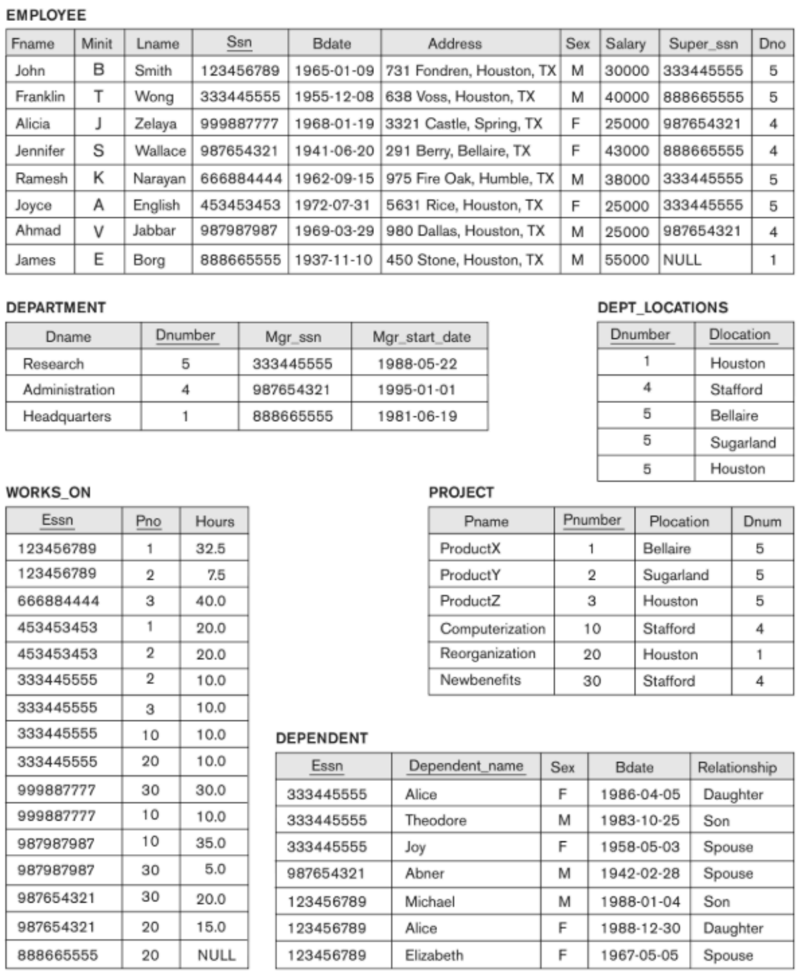
5.10 What is a transaction? How does it differ from an Update operation?

- **Transaction:** A logical unit of work consisting of one or more operations (insert, update, delete, select). It must satisfy **ACID properties** (Atomicity, Consistency, Isolation, Durability).
- **Update operation:** A single modification (e.g., changing a salary value).

  👉 A transaction may contain **multiple update operations** and ensures they are executed safely as one unit.

5.11. Suppose that each of the following Update operations is applied directly to the database state shown in Figure 5.6. Discuss all integrity constraints violated by each operation, if any, and the different ways of enforcing these constraints.

Figure 5.6
One possible database state for the COMPANY relational database schema.



EMPLOYEE

Fname	Minit	Lname	Ssn	Bdate	Address	Sex	Salary	Super_ssn	Dno
John	B	Smith	123456789	1965-01-09	731 Fondren, Houston, TX	M	30000	333445555	5
Franklin	T	Wong	333445555	1955-12-08	638 Voss, Houston, TX	M	40000	888665555	5
Alicia	J	Zelaya	999887777	1968-01-19	3321 Castle, Spring, TX	F	25000	987654321	4
Jennifer	S	Wallace	987654321	1941-06-20	291 Berry, Bellaire, TX	F	43000	888665555	4
Ramesh	K	Narayan	666884444	1962-09-15	975 Fire Oak, Humble, TX	M	38000	333445555	5
Joyce	A	English	453453453	1972-07-31	5631 Rice, Houston, TX	F	25000	333445555	5
Ahmad	V	Jabbar	987987987	1969-03-29	980 Dallas, Houston, TX	M	25000	987654321	4
James	E	Borg	888665555	1937-11-10	450 Stone, Houston, TX	M	55000	NULL	1

DEPARTMENT

Dname	Dnumber	Mgr_ssn	Mgr_start_date
Research	5	333445555	1988-05-22
Administration	4	987654321	1995-01-01
Headquarters	1	888665555	1981-06-19

DEPT_LOCATIONS

Dnumber	Dlocation
1	Houston
4	Stafford
5	Bellaire
5	Sugarland
5	Houston

WORKS_ON

Essn	Pno	Hours
123456789	1	32.5
123456789	2	7.5
666884444	3	40.0
453453453	1	20.0
453453453	2	20.0
333445555	2	10.0
333445555	3	10.0
333445555	10	10.0
333445555	20	10.0
999887777	30	30.0
999887777	10	10.0
987987987	10	35.0
987987987	30	5.0
987654321	30	20.0
987654321	20	15.0
888665555	20	NULL

PROJECT

Pname	Pnumber	Plocation	Dnum
ProductX	1	Bellaire	5
ProductY	2	Sugarland	5
ProductZ	3	Houston	5
Computerization	10	Stafford	4
Reorganization	20	Houston	1
Newbenefits	30	Stafford	4

DEPENDENT

Essn	Dependent_name	Sex	Bdate	Relationship
333445555	Alice	F	1986-04-05	Daughter
333445555	Theodore	M	1983-10-25	Son
333445555	Joy	F	1958-05-03	Spouse
987654321	Abner	M	1942-02-28	Spouse
123456789	Michael	M	1988-01-04	Son
123456789	Alice	F	1988-12-30	Daughter
123456789	Elizabeth	F	1967-05-05	Spouse

a. Insert <'Robert', 'F', 'Scott', '943775543', '1972-06-21', '2365 Newcastle Rd, Bellaire, TX', M, 58000, '888665555', 1> into EMPLOYEE.

b. Insert <'ProductA', 4, 'Bellaire', 2> into PROJECT.

c. Insert <'Production', 4, '943775543', '2007-10-01'> into DEPARTMENT.

d. Insert <'677678989', NULL, '40.0'> into WORKS_ON.

e. Insert <'453453453', 'John', 'M', '1990-12-12', 'spouse'> into DEPENDENT.

f. Delete the WORKS_ON tuples with Essn = '333445555'.

g. Delete the EMPLOYEE tuple with Ssn = '987654321'. h. Delete the PROJECT tuple with Pname = 'ProductX'. i. Modify the Mgr_ssn and Mgr_start_date of the DEPARTMENT tuple with Dnumber = 5 to '123456789' and '2007-10-01', respectively.

j. Modify the Super_ssn attribute of the EMPLOYEE tuple with Ssn = '999887777' to '943775543'.

k. Modify the Hours attribute of the WORKS_ON tuple with Essn = '999887777' and Pno = 10 to '5.0'.

(a) Insert `('Robert','F','Scott','943775543','1972-06-21','2365 Newcastle Rd, Bellaire, TX', M, 58000, '888665555', 1)` into EMPLOYEE

Check against Figure 5.6

- `Ssn = 943775543` — **not present** in current EMPLOYEE table.
- `Super_ssn = 888665555` — **present** (James E Borg, Ssn 888665555).
- `Dno = 1` — **present** (DEPARTMENT Dnumber = 1 exists: Headquarters).

Violations: None of the standard integrity constraints are violated.

- PK uniqueness: OK (943775543 not used).
- FK Super_ssn: OK (888665555 exists).
- FK Dno: OK (1 exists).
- Domain values look valid.

Enforcement / action: Insert succeeds. (No remedial action required.)

(b) Insert `('ProductA', 4, 'Bellaire', 2)` into PROJECT

(assumed order: `Pname`, `Pnumber`, `Plocation`, `Dnum`)

Check

- `Pnumber = 4` — **not present** among current project numbers (1,2,3,10,20,30) → OK for PK.
- `Dnum = 2` — **NOT present** in DEPARTMENT (departments are 5,4,1).

Violations

- **Referential integrity**: PROJECT.Dnum → DEPARTMENT.Dnumber fails because department `2` does not exist.

Ways to enforce / remediate

- **Reject** insertion (default).
- **Create DEPARTMENT Dnumber = 2 first**, then insert PROJECT.
- **Allow NULL / SET DEFAULT** for Dnum (only if schema allows it) — not typical.

- **Defer FK check** until end of transaction and insert department in same transaction (DB must support deferred constraints).
- **Fix 1 (create missing department):**

```
INSERT INTO DEPARTMENT VALUES ('TempDept', 2, NULL, NULL);
```

Then insert project.

- **Fix 2 (assign to existing dept):**

```
INSERT INTO PROJECT VALUES ('ProductA', 4, 'Bellaire', 5);
```

(c) Insert `('Production', 4, '943775543', '2007-10-01')` into DEPARTMENT

(assumed order: `Dname`, `Dnumber`, `Mgr_ssn`, `Mgr_start_date`)

Check

- `Dnumber = 4` — **already exists** (Administration has Dnumber = 4).
- `Mgr_ssn = 943775543` — **does not exist** in EMPLOYEE (figure's SSNs: 123456789,333445555,999887777,987654321,666884444,454354353,987987987,888665555).

Violations

1. **Primary-key (Dnumber) duplication** — cannot insert duplicate Dnumber = 4.
2. **Referential integrity** on Mgr_ssn — `943775543` is not an existing EMPLOYEE.Ssn.

- **Fix:** Assign a new Dnumber and a valid existing manager.

```
INSERT INTO DEPARTMENT VALUES ('Production', 6, '123456789', '2007-10-01');
```

(d) Insert `('677678989', NULL, '40.0')` into WORKS_ON

(assumed order: `Essn`, `Pno`, `Hours`)

Check

- `Essn = 677678989` — **not present** in EMPLOYEE.
- `Pno = NULL` — problematic because `(Essn,Pno)` is the primary key in WORKS_ON; PK attributes cannot be NULL. Also Pno must reference an existing PROJECT.Pnumber.

Violations

- **Referential integrity (Essn):** Essn value does not reference any EMPLOYEE.Ssn.
- **Primary-key / NOT NULL constraint:** `Pno` is part of the PK and cannot be NULL.

- **Referential integrity (Pno)**: even if NULL were allowed, Pno must (when non-NULL) reference PROJECT; here it's NULL so it cannot reference a project.
- **Fix:** Use an existing employee & project.

```
INSERT INTO WORKS_ON VALUES ('123456789', 1, 40.0);
```

e) Insert `('453453453', 'John', 'M', '1990-12-12', 'spouse')` into DEPENDENT

(assumed order: `Essn`, `Dependent_name`, `Sex`, `Bdate`, `Relationship`)

No violation

(f) Delete the WORKS_ON tuples with `Essn = '333445555'`

Check

- Figure shows WORKS_ON rows for `Essn = 333445555` (several project assignments).

Violations

- **None** of the core FKs are violated by deleting WORKS_ON rows: WORKS_ON references PROJECT and EMPLOYEE; deleting these WORKS_ON rows removes references, not create orphans elsewhere.
- No other table references WORKS_ON in the figure, so no cascading problem.

Ways to enforce / action

- The delete **succeeds** by default.
- Consider business/archival rules: if you must keep history, you might **archive** rather than delete.
- If the system had a rule "employee must have at least one project", deletion could violate business logic (enforce via triggers or reject).

(g) Delete the EMPLOYEE tuple with `Ssn = '987654321'`

Check

- `987654321` exists (Jennifer S Wallace).
- This SSN is referenced elsewhere in the figure:
  - DEPENDENT has `Essn = 987654321` (Dependent: Abner).
  - WORKS_ON has tuples with Essn = 987654321 (e.g., Pno 30 and Pno 20 rows).
  - DEPARTMENT: Administration (Dnumber = 4) has `Mgr_ssn = 987654321` — **department manager**.

    - Some employees' `Super_ssn` may reference 987654321 (e.g., Ahmad V Jabbar's Super_ssn = 987654321 in the figure).

Violations

- Deleting this EMPLOYEE would **violate referential integrity** for:
    - DEPENDENT rows (orphans),
    - WORKS_ON rows (orphans),
    - DEPARTMENT.Mgr_ssn (department would refer to non-existent manager),
    - Super_ssn references (subordinates would refer to non-existent supervisor).

Option 1: using ON DELETE CASCADE

If your foreign keys are defined with `ON DELETE CASCADE`, a **single query** will delete the employee and all related rows automatically:

```
DELETE FROM EMPLOYEE
WHERE Ssn = '987654321';
```

Requirements for this to work:

```
-- For dependents
ALTER TABLE DEPENDENT
ADD CONSTRAINT fk_dependent_emp
FOREIGN KEY (Essn) REFERENCES EMPLOYEE(Ssn)
ON DELETE CASCADE;

-- For supervisor references
ALTER TABLE EMPLOYEE
ADD CONSTRAINT fk_supervisor
FOREIGN KEY (Super_ssn) REFERENCES EMPLOYEE(Ssn)
ON DELETE CASCADE;

-- For department manager, if you want to delete the department automatically (not usually desir
ed)
ALTER TABLE DEPARTMENT
ADD CONSTRAINT fk_mgr
FOREIGN KEY (Mgr_ssn) REFERENCES EMPLOYEE(Ssn)
ON DELETE CASCADE;
```

Option 2: Manual handling (without cascading)

If your foreign keys are not set to cascade:

```
-- Step 1: Reassign department manager(s)
UPDATE DEPARTMENT
SET Mgr_ssn = '123456789'
WHERE Mgr_ssn = '987654321';

-- Step 2: Delete dependents
DELETE FROM DEPENDENT
WHERE Essn = '987654321';

-- Step 3: Reassign subordinates if needed
UPDATE EMPLOYEE
SET Super_ssn = NULL
WHERE Super_ssn = '987654321';

-- Step 4: Delete employee
DELETE FROM EMPLOYEE
WHERE Ssn = '987654321';
```

(h) Delete the PROJECT tuple with `Pname = 'ProductX'`

(ProductX has `Pnumber = 1`, `Dnum = 5`)

Check

- WORKS_ON currently references `Pno = 1` (e.g., WORKS_ON row `123456789, 1, 32.5`).

Violations

- **Referential integrity**: Deleting PROJECT `Pnumber = 1` would orphan WORKS_ON rows that reference that project.
- **Fix:** Delete those WORKS_ON first.

```
DELETE FROM WORKS_ON WHERE Pno = 1;
DELETE FROM PROJECT WHERE Pname = 'ProductX';
```

(i) Modify DEPARTMENT (Dnumber = 5) `Mgr_ssn` and `Mgr_start_date` → `'123456789'`, `'2007-10-01'`

Check

- Dept 5 (Research) currently has `Mgr_ssn = 333445555`.
- `123456789` **exists** in EMPLOYEE (John B Smith).

Violations

- **No FK violation:** `Mgr_ssn = 123456789` references an existing EMPLOYEE.Ssn → OK.
- **Possible business-rule violation**: Many schemas / business rules require that the manager **belong to the department they manage** (i.e., EMPLOYEE.Dno = DEPARTMENT.Dnumber). Here John has Dno = 1 while the department Dnumber = 5 — that would violate that business rule if enforced.

```
UPDATE DEPARTMENT SET Mgr_ssn = '333445555', Mgr_start_date = '2007-10-01' WHERE Dnumber = 5;
```

(j) Modify EMPLOYEE (Ssn = '999887777') `Super_ssn` → `'943775543'`

Check

- Employee `999887777` exists (Alicia J Zelaya).
- `943775543` **does not exist** among EMPLOYEE SSNs in figure.

Violations

- **Referential integrity**: `Super_ssn` is an FK referencing EMPLOYEE.Ssn. Setting it to 943775543 violates the FK (no such supervisor exists).

(k) Modify WORKS_ON (Essn = '999887777', Pno = 10) `Hours` → `5.0`

Check

- Figure shows a WORKS_ON row for `Essn = 999887777` and `Pno = 10` with Hours = 10.0 (and a second row for same emp on Pno = 30 with Hours = 30.0).
- After change, `Hours` for the two rows would be 30.0 + 5.0 = **35.0** total for that employee.

Violations

- **None** of the basic constraints are violated:
  - Hours is numeric and within typical bounds.
  - Row exists and is updated (not creating or breaking FKs).
- Only possible **business rule** to check: some examples restrict total hours across projects ≤ 40. Current total after update = 35.0 ≤ 40, so **no violation**.

```
UPDATE WORKS_ON SET Hours = 5.0 WHERE Essn = '999887777' AND Pno = 10;
```

5.13. Consider the relation CLASS(Course#, Univ_Section#, Instructor_name, Semester, Building_code, Room#, Time_period, Weekdays, Credit_hours). This represents classes taught in a university, with unique Univ_section#s. Identify what you think should be various candidate keys, and write in your own words the conditions or assumptions under which each candidate key would be valid.

Step 1: Obvious Candidate Key

1. `Univ_Section#`
    - Assumption: Each university section is uniquely assigned an ID.
    - Condition: This ID is unique across all courses and semesters.
    - ✅ This is the simplest candidate key.

Step 2: Other possible candidate keys

Since `Univ_Section#` is unique, technically we could also consider **combinations of attributes** that can uniquely identify a class. Some possibilities:

1. **Combination of** `Course# + Semester + Time_period + Room#`
    - Assumption:
        - A course in a particular semester has only **one section per time period per room**.
        - No two courses share the same room and time period in the same semester.
    - Condition: This combination uniquely identifies a specific class.
2. **Combination of** `Course# + Instructor_name + Semester + Time_period`
    - Assumption:
        - An instructor teaches a specific course at a specific time only once per semester.
    - Condition: This uniquely identifies the class without needing `Room#` if rooms don't duplicate for a course at the same time.
3. **Combination of** `Course# + Semester + Building_code + Room# + Time_period`
    - Assumption:
        - A course in a semester is scheduled in **one room** at a specific time.
        - Two different courses cannot occupy the same room at the same time in the same semester.
    - Condition: Can uniquely identify the class, independent of instructor.

5.15. Consider the following relations for a database that keeps track of business trips of salespersons in a sales office:
SALESPERSON(Ssn, Name, Start_year, Dept_no)

TRIP(Ssn, From_city, To_city, Departure_date, Return_date, Trip_id)
EXPENSE(Trip_id, Account#, Amount)
A trip can be charged to one or more accounts. Specify the foreign keys for this schema, stating any assumptions you make.

```
-- TRIP: each trip is associated with a salesperson
ALTER TABLE TRIP
ADD CONSTRAINT fk_trip_salesperson
FOREIGN KEY (Ssn) REFERENCES SALESPERSON(Ssn);

-- EXPENSE: each expense belongs to a trip
ALTER TABLE EXPENSE
ADD CONSTRAINT fk_expense_trip
FOREIGN KEY (Trip_id) REFERENCES TRIP(Trip_id);
```

1. `Ssn` in `SALESPERSON` is unique (primary key).
2. `Trip_id` in `TRIP` is unique.
3. A trip can be charged to **multiple accounts**, so `(Trip_id, Account#)` is used as the primary key in `EXPENSE`.
4. Deletion or updates on SALESPERSON or TRIP may optionally use `ON DELETE CASCADE` if you want trips and expenses to be removed automatically when the salesperson or trip is deleted.

5.16. Consider the following relations for a database that keeps track of student enrollment in courses and the books adopted for each course:
STUDENT(Ssn, Name, Major, Bdate)
COURSE(Course#, Cname, Dept)
ENROLL(Ssn, Course#, Quarter, Grade)
BOOK_ADOPTION(Course#, Quarter, Book_isbn)
TEXT(Book_isbn, Book_title, Publisher, Author)
Specify the foreign keys for this schema, stating any assumptions you make.

```
-- ENROLL: student enrollment
ALTER TABLE ENROLL
ADD CONSTRAINT fk_enroll_student
FOREIGN KEY (Ssn) REFERENCES STUDENT(Ssn);

ALTER TABLE ENROLL
ADD CONSTRAINT fk_enroll_course
FOREIGN KEY (Course#) REFERENCES COURSE(Course#);

-- BOOK_ADOPTION: books adopted for courses
```

```
ALTER TABLE BOOK_ADOPTION
ADD CONSTRAINT fk_bookadoption_course
FOREIGN KEY (Course#) REFERENCES COURSE(Course#);

ALTER TABLE BOOK_ADOPTION
ADD CONSTRAINT fk_bookadoption_text
FOREIGN KEY (Book_isbn) REFERENCES TEXT(Book_isbn);
```

Assumptions

1. `Ssn` uniquely identifies a student.
2. `Course#` uniquely identifies a course.
3. A student can enroll in the same course in different quarters.
4. Each course can adopt multiple books per quarter.
5. All adopted books exist in the `TEXT` table.
6. `Quarter` is assumed to be consistent between `ENROLL` and `BOOK_ADOPTION`, but no separate table is used to validate quarters.

5.12. Consider the AIRLINE relational database schema shown in Figure 5.8, which describes a database for airline flight information. Each FLIGHT is identified by a Flight_number, and consists of one or more FLIGHT_LEGs with Leg_numbers 1, 2, 3, and so on. Each FLIGHT_LEG has scheduled arrival and departure times, airports, and one or more LEG_INSTANCEs, one for each Date on which the flight travels. FAREs are kept for each FLIGHT. For each FLIGHT_LEG instance,SEAT_RESERVATIONs are kept, as are the AIRPLANE used on the leg and the actual arrival and departure times and airports. An AIRPLANE is identified by an Airplane_id and is of a particular AIRPLANE_TYPE. CAN_LAND relates AIRPLANE_TYPEs to the AIRPORTs at which they can land. An AIRPORT is identified by an Airport_code. Consider an update for the AIRLINE database to enter a reservation on a particular flight or flight leg on a given date.

a. Give the operations for this update.
b. What types of constraints would you expect to check?

c. Which of these constraints are key, entity integrity, and referential integrity constraints, and which are not?

d. Specify all the referential integrity constraints that hold on the schema shown in Figure 5.8.

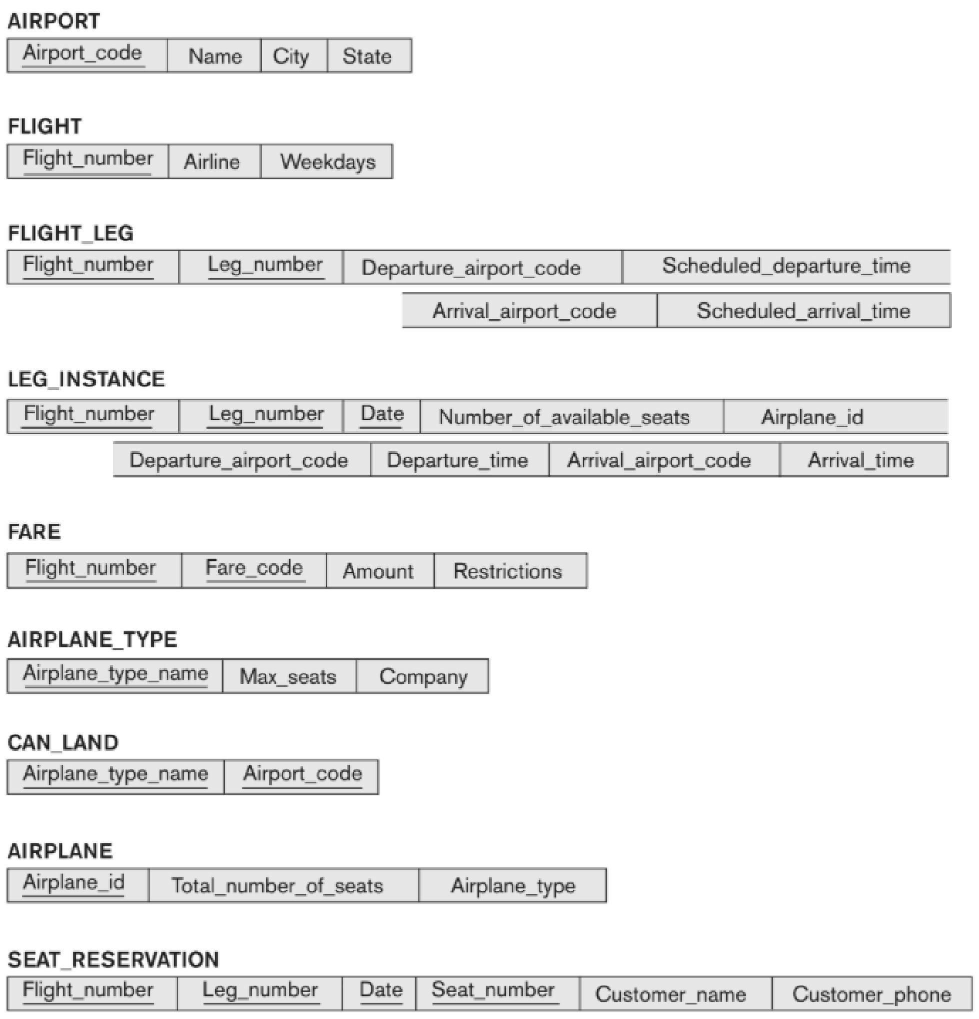

AIRPORT
Airport_code | Name | City | State
FLIGHT
Flight_number | Airline | Weekdays
FLIGHT_LEG
Flight_number | Leg_number | Departure_airport_code | Scheduled_departure_time
Arrival_airport_code | Scheduled_arrival_time
LEG_INSTANCE
Flight_number | Leg_number | Date | Number_of_available_seats | Airplane_id
Departure_airport_code | Departure_time | Arrival_airport_code | Arrival_time
FARE
Flight_number | Fare_code | Amount | Restrictions
AIRPLANE_TYPE
Airplane_type_name | Max_seats | Company
CAN_LAND
Airplane_type_name | Airport_code
AIRPLANE
Airplane_id | Total_number_of_seats | Airplane_type
SEAT_RESERVATION
Flight_number | Leg_number | Date | Seat_number | Customer_name | Customer_phone

Figure 5.8
The AIRLINE relational database schema.

5.12 (a) Operations for entering a reservation

To enter a new reservation on a particular flight leg on a given date:

1. **Check that the flight leg exists**

```
SELECT *
FROM LEG_INSTANCE
WHERE Flight_number = X AND Leg_number = Y AND Date = '2025-09-21';
```

2. **Check seat availability**
   - Ensure that `Number_of_available_seats > 0` in the corresponding `LEG_INSTANCE`.
3. **Insert new reservation**

```
INSERT INTO SEAT_RESERVATION
(Flight_number, Leg_number, Date, Seat_number, Customer_name, Customer_phone)
VALUES (X, Y, '2025-09-21', '12A', 'John Doe', '123-456-7890');
```

4. **Update available seats**

```
UPDATE LEG_INSTANCE
SET Number_of_available_seats = Number_of_available_seats - 1
WHERE Flight_number = X AND Leg_number = Y AND Date = '2025-09-21';
```

5.12 (b) Constraints to check

When entering the reservation, we should check:

1. **Entity existence**
   - Does the `FLIGHT_NUMBER` exist?
   - Does the `(Flight_number, Leg_number, Date)` exist in `LEG_INSTANCE`?
2. **Seat availability**
   - Ensure the seat number is valid (`<= total seats of airplane`)
   - Ensure the seat is not already reserved in `SEAT_RESERVATION`.
3. **Data consistency**
   - `Number_of_available_seats` must not go negative.
4. **Customer info validity**
   - `Customer_name` and `Customer_phone` should be non-null.

5.12 (c) Types of constraints

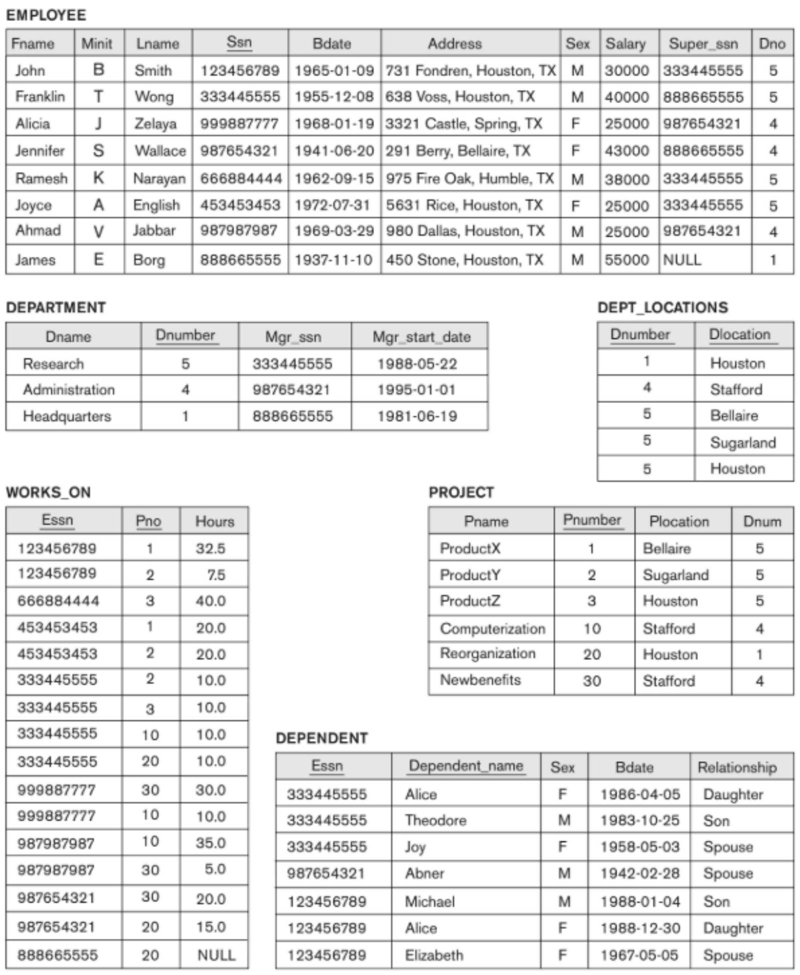
1. **Key constraints**
   - Primary keys:
     - `AIRPORT(Airport_code)`
     - `FLIGHT(Flight_number)`
     - `FLIGHT_LEG(Flight_number, Leg_number)`
     - `LEG_INSTANCE(Flight_number, Leg_number, Date)`
     - `FARE(Flight_number, Fare_code)`
     - `AIRPLANE_TYPE(Airplane_type_name)`
     - `CAN_LAND(Airplane_type_name, Airport_code)`
     - `AIRPLANE(Airplane_id)`
     - `SEAT_RESERVATION(Flight_number, Leg_number, Date, Seat_number)`

2. **Entity integrity constraints**
    - Primary keys must not be NULL.
3. **Referential integrity constraints**
    - Foreign key dependencies (see part d).
4. **Other constraints (not key/entity/referential)**
    - `Number_of_available_seats >= 0`
    - `Seat_number ≤ Airplane.total_number_of_seats`

5.12 (d) Referential Integrity Constraints

```
-- AIRPORT
CREATE TABLE AIRPORT (
    Airport_code CHAR(5) PRIMARY KEY,
    Name VARCHAR(50),
    City VARCHAR(50),
    State VARCHAR(50)
);

-- FLIGHT
CREATE TABLE FLIGHT (
    Flight_number CHAR(10) PRIMARY KEY,
    Airline VARCHAR(50),
    Weekdays VARCHAR(20)
);

-- FLIGHT_LEG
CREATE TABLE FLIGHT_LEG (
    Flight_number CHAR(10),
    Leg_number INT,
    Departure_airport_code CHAR(5),
    Scheduled_departure_time TIME,
    Arrival_airport_code CHAR(5),
    Scheduled_arrival_time TIME,
    PRIMARY KEY (Flight_number, Leg_number),
    FOREIGN KEY (Flight_number) REFERENCES FLIGHT(Flight_number)
);

-- LEG_INSTANCE
CREATE TABLE LEG_INSTANCE (
    Flight_number CHAR(10),
    Leg_number INT,
    Date DATE,
    Number_of_available_seats INT,
    Airplane_id CHAR(10),
    Departure_airport_code CHAR(5),
    Departure_time TIME,
    Arrival_airport_code CHAR(5),
    Arrival_time TIME,
    PRIMARY KEY (Flight_number, Leg_number, Date),
    FOREIGN KEY (Flight_number, Leg_number) REFERENCES FLIGHT_LEG(Flight_number, Leg_nu
mber),
    FOREIGN KEY (Flight_number)
    REFERENCES FLIGHT(Flight_number)
);

-- FARE
CREATE TABLE FARE (
    Flight_number CHAR(10),
    Fare_code CHAR(5),
    Amount DECIMAL(8,2),
    Restrictions VARCHAR(100),
    PRIMARY KEY (Flight_number, Fare_code),
    FOREIGN KEY (Flight_number) REFERENCES FLIGHT(Flight_number)
);

-- AIRPLANE_TYPE
CREATE TABLE AIRPLANE_TYPE (
    Airplane_type_name VARCHAR(30) PRIMARY KEY,
    Max_seats INT,
    Company VARCHAR(50)
);

-- AIRPLANE
CREATE TABLE AIRPLANE (
    Airplane_id CHAR(10) PRIMARY KEY,
    Total_number_of_seats INT,
    Airplane_type VARCHAR(30),
    FOREIGN KEY (Airplane_type) REFERENCES AIRPLANE_TYPE(Airplane_type_name)
);

-- CAN_LAND
CREATE TABLE CAN_LAND (
    Airplane_type_name VARCHAR(30),
    Airport_code CHAR(5),
    PRIMARY KEY (Airplane_type_name, Airport_code),
    FOREIGN KEY (Airplane_type_name) REFERENCES AIRPLANE_TYPE(Airplane_type_name),
    FOREIGN KEY (Airport_code) REFERENCES AIRPORT(Airport_code)
);

-- SEAT_RESERVATION
CREATE TABLE SEAT_RESERVATION (
    Flight_number CHAR(10),
    Leg_number INT,
    Date DATE,
    Seat_number CHAR(5),
    Customer_name VARCHAR(50),
    Customer_phone VARCHAR(20),
    PRIMARY KEY (Flight_number, Leg_number, Date, Seat_number),
    FOREIGN KEY (Flight_number, Leg_number, Date)
        REFERENCES LEG_INSTANCE(Flight_number, Leg_number, Date)
    FOREIGN KEY (Flight_number)
    REFERENCES FLIGHT(Flight_number)
);
```

Figure 5.6
One possible database state for the COMPANY relational database schema.



EMPLOYEE

Fname	Minit	Lname	Ssn	Bdate	Address	Sex	Salary	Super_ssn	Dno
John	B	Smith	123456789	1965-01-09	731 Fondren, Houston, TX	M	30000	333445555	5
Franklin	T	Wong	333445555	1955-12-08	638 Voss, Houston, TX	M	40000	888665555	5
Alicia	J	Zelaya	999887777	1968-01-19	3321 Castle, Spring, TX	F	25000	987654321	4
Jennifer	S	Wallace	987654321	1941-06-20	291 Berry, Bellaire, TX	F	43000	888665555	4
Ramesh	K	Narayan	666884444	1962-09-15	975 Fire Oak, Humble, TX	M	38000	333445555	5
Joyce	A	English	453453453	1972-07-31	5631 Rice, Houston, TX	F	25000	333445555	5
Ahmad	V	Jabbar	987987987	1969-03-29	980 Dallas, Houston, TX	M	25000	987654321	4
James	E	Borg	888665555	1937-11-10	450 Stone, Houston, TX	M	55000	NULL	1

DEPARTMENT

Dname	Dnumber	Mgr_ssn	Mgr_start_date
Research	5	333445555	1988-05-22
Administration	4	987654321	1995-01-01
Headquarters	1	888665555	1981-06-19

DEPT_LOCATIONS

Dnumber	Dlocation
1	Houston
4	Stafford
5	Bellaire
5	Sugarland
5	Houston

WORKS_ON

Essn	Pno	Hours
123456789	1	32.5
123456789	2	7.5
666884444	3	40.0
453453453	1	20.0
453453453	2	20.0
333445555	2	10.0
333445555	3	10.0
333445555	10	10.0
333445555	20	10.0
999887777	30	30.0
999887777	10	10.0
987987987	10	35.0
987987987	30	5.0
987654321	30	20.0
987654321	20	15.0
888665555	20	NULL

PROJECT

Pname	Pnumber	Plocation	Dnum
ProductX	1	Bellaire	5
ProductY	2	Sugarland	5
ProductZ	3	Houston	5
Computerization	10	Stafford	4
Reorganization	20	Houston	1
Newbenefits	30	Stafford	4

DEPENDENT

Essn	Dependent_name	Sex	Bdate	Relationship
333445555	Alice	F	1986-04-05	Daughter
333445555	Theodore	M	1983-10-25	Son
333445555	Joy	F	1958-05-03	Spouse
987654321	Abner	M	1942-02-28	Spouse
123456789	Michael	M	1988-01-04	Son
123456789	Alice	F	1988-12-30	Daughter
123456789	Elizabeth	F	1967-05-05	Spouse

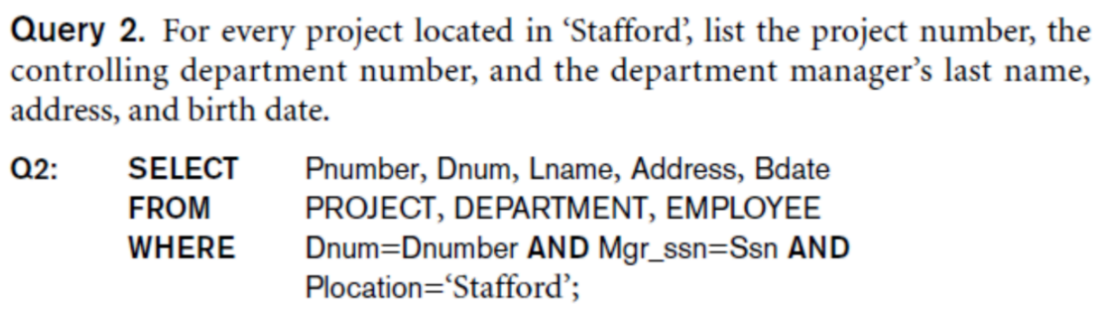
BASIC RETRIEVAL QUERIES:

Query 0. Retrieve the birth date and address of the employee(s) whose name is 'John B. Smith'.

```
Q0:   SELECT   Bdate, Address
      FROM     EMPLOYEE
      WHERE    Fname='John' AND Minit='B' AND Lname='Smith';
```

Query 1. Retrieve the name and address of all employees who work for the 'Research' department.

```
Q1:   SELECT   Fname, Lname, Address
      FROM     EMPLOYEE, DEPARTMENT
      WHERE    Dname='Research' AND Dnumber=Dno;
```



Query 2. For every project located in 'Stafford', list the project number, the controlling department number, and the department manager's last name, address, and birth date.

```
Q2:     SELECT  Pnumber, Dnum, Lname, Address, Bdate
        FROM    PROJECT, DEPARTMENT, EMPLOYEE
        WHERE   Dnum=Dnumber AND Mgr_ssn=Ssn AND
                Plocation='Stafford';
```

Q: For each employee, retrieve the employee's first and last name and the first and last name of his or her immediate supervisor.

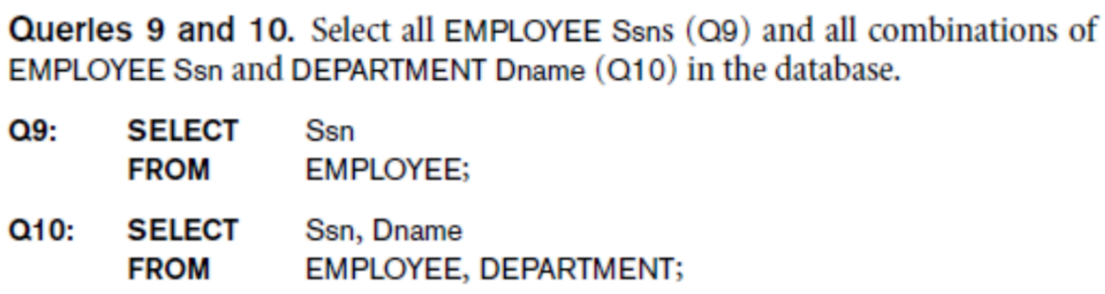
```
SELECT E.Fname, E.Lname, S.Fname, S.Lname
FROM EMPLOYEE AS E, EMPLOYEE AS S
WHERE E.Super_ssn=S.Ssn;
```

Missing WHERE clause

Indicates no condition on tuple selection

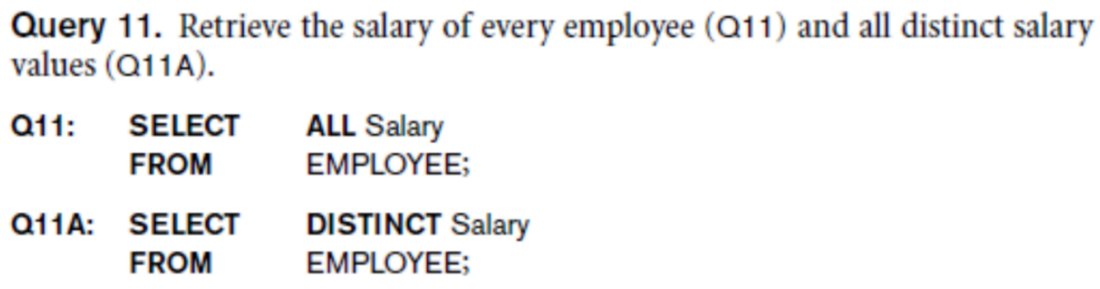
Effect is a CROSS PRODUCT

Result is all possible tuple combinations (or the Algebra operation of Cartesian Product– see Ch.8)



Queries 9 and 10. Select all EMPLOYEE Ssns (Q9) and all combinations of EMPLOYEE Ssn and DEPARTMENT Dname (Q10) in the database.

```
Q9:     SELECT  Ssn
        FROM    EMPLOYEE;

Q10:    SELECT  Ssn, Dname
        FROM    EMPLOYEE, DEPARTMENT;
```



Query 11. Retrieve the salary of every employee (Q11) and all distinct salary values (Q11A).

```
Q11:    SELECT  ALL Salary
        FROM    EMPLOYEE;

Q11A:   SELECT  DISTINCT Salary
        FROM    EMPLOYEE;
```

Bulk Loading of Tables

Normally, we use `INSERT` to add **one tuple (row)** or a few rows at a time into a table. But if we want to **copy many rows at once**, SQL provides variations like **bulk loading** using `CREATE TABLE ... AS SELECT ...` (sometimes with `LIKE` and `WITH DATA`).

Step-by-Step Explanation of the Example

```
CREATE TABLE D5EMPS LIKE EMPLOYEE
(SELECT E.*
 FROM EMPLOYEE AS E
 WHERE E.Dno = 5)
WITH DATA;
```

1. **`CREATE TABLE D5EMPS LIKE EMPLOYEE`**
    - Creates a **new table** called `D5EMPS`.
    - It has the **same structure (attributes/columns)** as the `EMPLOYEE` table.
2. **`(SELECT E.* FROM EMPLOYEE AS E WHERE E.Dno = 5)`**
    - Selects all rows (`E.*`) from the `EMPLOYEE` table where the department number `Dno = 5` .
    - This is the data we want to bulk-load into the new table.
3. **`WITH DATA`**
    - Ensures that not only the structure is created but also the **selected data is copied** into `D5EMPS` .
    - If we had used `WITH NO DATA` , it would only create the empty table `D5EMPS` with the same schema, but no rows inside.

What This Does

- It creates a new table `D5EMPS` .
- The new table has **the same columns as EMPLOYEE**.
- It is filled with all employees who work in **Department 5** (`Dno = 5`).

SELECT <attribute and function list>
FROM <table list>
[WHERE <condition>]
[GROUP BY <grouping attribute(s)>]
[HAVING <group condition>]
[ORDER BY <attribute list>];

- FROM → take table **Employee**
- WHERE → keep only employees with Salary > 50000

- GROUP BY → group remaining rows by Deptno
- HAVING → keep only those groups where count > 3
- SELECT → output Deptno and the count
- ORDER BY → sort by NumEmployees descending

🔷 Referential Integrity Options

When we create a **foreign key**, we tell the DBMS what should happen to rows in the **child table** when a referenced row in the **parent table** is **deleted or updated**.

The options are:

1. ON DELETE CASCADE

- **Meaning:** If a row in the parent table is deleted, all referencing rows in the child table are automatically deleted.
- **Use case:** When child records are meaningless without the parent (e.g., order details without an order).

✅ Example:

```
CREATE TABLE Department (
    DeptID INT PRIMARY KEY,
    DeptName VARCHAR(50)
);

CREATE TABLE Employee (
    EmpID INT PRIMARY KEY,
    EmpName VARCHAR(50),
    DeptID INT,
    FOREIGN KEY (DeptID) REFERENCES Department(DeptID)
        ON DELETE CASCADE
);
```

👉 Scenario: If you delete a department, all employees in that department are automatically deleted.

2. ON DELETE SET NULL

- **Meaning:** If a parent row is deleted, the foreign key in child rows is set to `NULL`.
- **Use case:** When you want to keep child records but indicate that they are no longer linked to a parent.

✅ Example:

```
CREATE TABLE Employee (
    EmpID INT PRIMARY KEY,
    EmpName VARCHAR(50),
    DeptID INT,
    FOREIGN KEY (DeptID) REFERENCES Department(DeptID)
        ON DELETE SET NULL
);
```

👉 Scenario: If a department is deleted, employees remain in the table, but their `DeptID` becomes `NULL` (no department assigned).

3. ON DELETE RESTRICT (or NO ACTION)

- **Meaning:** Prevents deletion of a parent row if there are related child rows.
- **Use case:** To enforce that parent cannot be removed while it still has dependents.

✅ Example:

```
CREATE TABLE Employee (
    EmpID INT PRIMARY KEY,
    EmpName VARCHAR(50),
    DeptID INT,
    FOREIGN KEY (DeptID) REFERENCES Department(DeptID)
        ON DELETE RESTRICT
);
```

👉 Scenario: If you try to delete a department that still has employees, the DBMS will reject the delete.

4. ON UPDATE CASCADE

- **Meaning:** If a parent row's key value is updated, the foreign keys in child rows are automatically updated too.
- **Use case:** Useful when primary keys are natural keys that can change (like flight numbers or product codes).

✅ Example:

```
CREATE TABLE Employee (
    EmpID INT PRIMARY KEY,
    EmpName VARCHAR(50),
    DeptID INT,
    FOREIGN KEY (DeptID) REFERENCES Department(DeptID)
        ON UPDATE CASCADE
);
```

👉 Scenario: If you update a department's ID from `10` to `20`, all employees with `DeptID = 10` will automatically be updated to `20`.

5. ON UPDATE SET NULL

- **Meaning:** If the parent's key changes, child foreign key values are set to `NULL`.
- **Use case:** When you want to keep child rows but indicate they are no longer linked to a parent after the update.

6. ON UPDATE RESTRICT (or NO ACTION)

- **Meaning:** Prevents updating of the parent's key if child rows exist.
- **Use case:** When foreign keys must remain consistent and should not automatically change.

🔷 Comparison Table

Action	On Delete Behavior	On Update Behavior	Typical Use Case
CASCADE	Delete child rows automatically	Update child rows automatically	Strong dependency
SET NULL	Keep child row, set FK = NULL	Keep child row, set FK = NULL	Optional relationship
RESTRICT / NO ACTION	Prevent delete if child exists	Prevent update if child exists	Preserve strict consistency

🔷 1. `= ANY` (or `= SOME`)

- **Meaning:** `v = ANY (subquery)` is the same as `v IN (subquery)`.
- Returns `TRUE` if `v` matches at least one value in the set.

✅ Example:

```
SELECT Name
FROM Employee
WHERE DeptID = ANY (
    SELECT DeptID
    FROM Department
    WHERE Location = 'Karachi'
);
```

👉 This finds employees who work in **any department located in Karachi**.

Equivalent to `DeptID IN (SELECT DeptID...)`.

🔷 2. `> ANY`

- **Meaning:** `v > ANY (subquery)` means `v` is greater than at least one value in the set.
- Equivalent to: `v > MIN(set)`.

✅ Example:

```
SELECT Name
FROM Employee
WHERE Salary > ANY (
    SELECT Salary
    FROM Employee
    WHERE DeptID = 5
);
```

👉 Returns employees whose salary is greater than **at least one employee in Dept 5**.

🔷 3. `> ALL`

- **Meaning:** `v > ALL (subquery)` means `v` is greater than every value in the set.
- Equivalent to: `v > MAX(set)`.

✅ Example:

```
SELECT Name
FROM Employee
WHERE Salary > ALL (
    SELECT Salary
    FROM Employee
    WHERE DeptID = 5
);
```

👉 Returns employees whose salary is greater than **every salary in Dept 5** (i.e., higher than the highest salary in Dept 5).

🔷 4. `< ANY` and `< ALL`

- **`< ANY`**: `v` is less than at least one value (so, `v < MAX(set)`).
- **`< ALL`**: `v` is less than every value (so, `v < MIN(set)`).

✅ Example:

```
-- Salary less than at least one in Dept 5
SELECT Name
FROM Employee
WHERE Salary < ANY (
    SELECT Salary FROM Employee WHERE DeptID = 5
);

-- Salary less than all in Dept 5 (minimum salary)
```

```
SELECT Name
FROM Employee
WHERE Salary < ALL (
    SELECT Salary FROM Employee WHERE DeptID = 5
);
```

🔷 5. `<> ALL` vs `<> ANY`

- `<> ALL`: means `v` is not equal to **any** value in the set (i.e., `v` is different from all values → safe).

  ```
  -- Not in subquery (same as NOT IN)
  SELECT Name
  FROM Employee
  WHERE DeptID <> ALL (
      SELECT DeptID FROM Department WHERE Location = 'Lahore'
  );
  ```

- `<> ANY`: means `v` is not equal to at least one value in the set → usually **always true**, unless the set has only one element.

✅ Example:

🔷 Quick Comparison Table

Operator Form	Equivalent To	Meaning
`= ANY` / `= SOME`	`IN`	True if equal to at least one value
`> ANY`	`> MIN(set)`	Greater than at least one value
`> ALL`	`> MAX(set)`	Greater than all values
`< ANY`	`< MAX(set)`	Less than at least one value
`< ALL`	`< MIN(set)`	Less than all values
`<> ALL`	`NOT IN`	Not equal to any value in the set

7.3. Discuss how NULLs are treated in comparison operators in SQL. How are NULLs treated when aggregate functions are applied in an SQL query? How are NULLs treated if they exist in grouping attributes?

1. NULLs in Comparison Operators

- In SQL, `NULL` means **unknown / missing value**.
- Any comparison with `NULL` (`=`, `<`, `>`, `<>`) returns **UNKNOWN**, not `TRUE` or `FALSE`.
  - Example: `5 = NULL` → UNKNOWN

    - `NULL = NULL` → UNKNOWN (not true, because both are unknown values).
- To test for NULL explicitly, we use:

```
IS NULL
IS NOT NULL
```

2. NULLs in Aggregate Functions

- Aggregate functions **ignore NULL values** (except `COUNT(*)`).
    - `COUNT(column)` → counts only non-NULL values.
    - `SUM`, `AVG`, `MIN`, `MAX` → computed only over non-NULL values.
- If all values are `NULL`, the result of the aggregate is `NULL` (except `COUNT(*)`, which returns total row count).

3. NULLs in Grouping Attributes

- When grouping with `GROUP BY`, all `NULL` values are treated as **equal** and placed into the same group.
- Example:

```
SELECT DeptID, COUNT(*)
FROM Employee
GROUP BY DeptID;
```

    - All rows with `DeptID = NULL` will be grouped together as one group.

Summary:

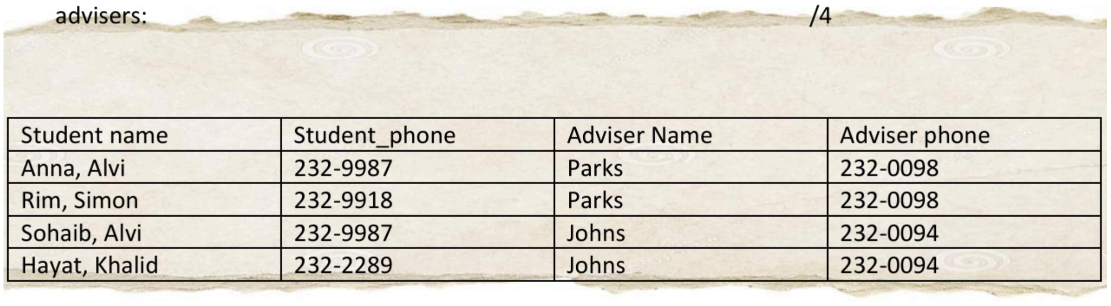
- Comparisons with `NULL` → UNKNOWN, must use `IS NULL`.
- Aggregates ignore `NULL` (except `COUNT(*)`).
- In grouping, all `NULL`s are treated as equal and fall into a single group.

Q: Specify the following queries on the database in Figure 5.5 in SQL. Show the query results if each query is applied to the database state in Figure 5.6.
a. For each department whose average employee salary is more than $30,000, retrieve the department name and the number of employees working for that department.

```
SELECT D.Dname,
       COUNT(E.SSN) AS NumEmployees
FROM Department D
JOIN Employee E
```

```
    ON D.Dnumber = E.Dno
GROUP BY D.Dname
HAVING AVG(E.Salary) > 30000;
```

a. Given is a glimpse of a table in a legacy system for maintaining records of students and their advisers: /4



Student name	Student_phone	Adviser Name	Adviser phone
Anna, Alvi	232-9987	Parks	232-0098
Rim, Simon	232-9918	Parks	232-0098
Sohaib, Alvi	232-9987	Johns	232-0094
Hayat, Khalid	232-2289	Johns	232-0094

- Write any 3 problems that you find out in this design which make manipulating data difficult.
- Suggest the improvements to remove those flaws you found in the design.

Problems

1. **Data Redundancy**
   - Adviser names and phone numbers are repeated multiple times for different students (e.g., "Parks, 232-0098" appears multiple times).
   - Leads to **inconsistency** and wastes storage.
2. **Difficulty in Updating Data**
   - If an adviser's phone number changes, it must be updated in **multiple rows**.
   - This increases the chance of errors.
3. **Poor Data Organization / Mixed Information**
   - Student and adviser information is stored in the same table.
   - Makes it hard to query just students or just advisers.

Improvements

1. **Separate Tables for Students and Advisers**
   - Create one table for `Student(StudentID, Name, Phone)` and one for `Adviser(AdviserID, Name, Phone)`.
2. **Use a Relationship Table**
   - Use a table `StudentAdviser(StudentID, AdviserID)` to link students and advisers.
   - This removes redundancy and allows multiple advisers per student if needed.
3. **Enforce Primary Keys and Foreign Keys**

- Ensures data consistency and easier updates.

Resulting Design (Simplified)

- **Student:** `(StudentID, Name, Phone)`
- **Adviser:** `(AdviserID, Name, Phone)`
- **StudentAdviser:** `(StudentID, AdviserID)`

Data Independence in Three-Schema Architecture

- **Data Independence** is the ability to **change the schema at one level without affecting other levels**.
- **Three Schema Architecture:**
  1. **Internal Schema** → Physical storage details
  2. **Conceptual Schema** → Logical structure of the database
  3. **External Schema** → User views
- **Types:**
  - **Logical Data Independence:** Changes in conceptual schema (e.g., adding a new attribute) do **not affect external/user views**.
  - **Physical Data Independence:** Changes in internal storage (e.g., file structure, indexes) do **not affect conceptual schema**.

Superkey, Key, and Default Superkey

1. **Superkey**
   - A set of one or more attributes that **uniquely identifies a tuple** in a relation.
2. **Key (Candidate Key)**
   - A **minimal superkey**; no attribute can be removed without losing uniqueness.
3. **Default Superkey**
   - The **set of all attributes in a relation**.
   - Always uniquely identifies tuples, but usually not minimal.

Example:

STUDENT(SSN, Name, Major, DOB)

- Default Superkey: {SSN, Name, Major, DOB}
- Candidate Key: {SSN}

Usage of ALL and ANY

1. `ANY` (or `SOME`)

- Compares a value with **any value in a subquery result**.
- Returns `TRUE` if the condition matches **at least one value** from the subquery.

Example:

```
SELECT employee_ID, First_Name, job_ID
FROM EMPLOYEES
WHERE SALARY < ANY (
    SELECT salary
    FROM EMPLOYEES
    WHERE JOB_ID = 'PU_CLERK'
);
```

Explanation:

- Subquery: `SELECT salary FROM EMPLOYEES WHERE JOB_ID = 'PU_CLERK'` → gets all salaries of PU_CLERKs.
- Main query: selects employees whose salary is **less than at least one PU_CLERK salary**.

Equivalent:

- You can also write `SOME` instead of `ANY`.

2. `ALL`

- Compares a value with **all values in a subquery result**.
- Returns `TRUE` only if the condition matches **every value** from the subquery.

Example:

```
SELECT employee_ID, First_Name, job_ID
FROM EMPLOYEES
WHERE SALARY > ALL (
    SELECT salary
    FROM EMPLOYEES
    WHERE JOB_ID = 'PU_CLERK'
)
AND job_ID <> 'PU_CLERK';
```

Explanation:

- Subquery: all PU_CLERK salaries.

- Main query: selects employees whose salary is **greater than every PU_CLERK salary** and who are **not PU_CLERK**.

✅ Key Points:

Operator	Meaning	Returns TRUE when
`ANY`	Less/greater than **any** value in subquery	At least one value satisfies condition
`ALL`	Less/greater than **all** values in subquery	Every value satisfies condition

1. Employee(s) with salary greater than any employee in department 80

```
SELECT first_name, last_name
FROM EMPLOYEES
WHERE salary > ANY (
    SELECT salary
    FROM EMPLOYEES
    WHERE department_id = 80
);
```

2. Employees hired before employee with ID = 150

```
SELECT first_name, last_name, hire_date
FROM EMPLOYEES
WHERE hire_date < (
    SELECT hire_date
    FROM EMPLOYEES
    WHERE employee_id = 150
);
```

3. Job title of employee with maximum salary

```
SELECT job_title
FROM EMPLOYEES e
JOIN JOBS j ON e.job_id = j.job_id
WHERE salary = (
    SELECT MAX(salary)
    FROM EMPLOYEES
);
```

4. Employees in the same department as “Neena”

```
SELECT first_name, last_name
FROM EMPLOYEES
WHERE department_id = (
   SELECT department_id
   FROM EMPLOYEES
   WHERE first_name = 'Neena'
);
```

5. Department where the highest-paid employee works

```
SELECT department_name
FROM DEPARTMENTS
WHERE department_id = (
   SELECT department_id
   FROM EMPLOYEES
   WHERE salary = (SELECT MAX(salary) FROM EMPLOYEES)
);
```

6. Employees earning more than their department's average salary

```
SELECT e.first_name, e.last_name, e.salary
FROM EMPLOYEES e
JOIN (
   SELECT department_id, AVG(salary) AS avg_salary
   FROM EMPLOYEES
   GROUP BY department_id
) d_avg ON e.department_id = d_avg.department_id
WHERE e.salary > d_avg.avg_salary;
```

7. Employees in the same department as the employee with the lowest salary

```
SELECT first_name, last_name, department_id
FROM EMPLOYEES
WHERE department_id = (
   SELECT department_id
   FROM EMPLOYEES
   WHERE salary = (SELECT MIN(salary) FROM EMPLOYEES)
);
```

8. Employees earning less than "Lex De Haan"

```
SELECT first_name, last_name
FROM EMPLOYEES
WHERE salary < (
   SELECT salary
   FROM EMPLOYEES
   WHERE first_name = 'Lex' AND last_name = 'De Haan'
);
```

9. Department managed by the manager with the lowest salary

```
SELECT department_name, manager_id
FROM DEPARTMENTS
WHERE manager_id = (
   SELECT employee_id
   FROM EMPLOYEES
   WHERE salary = (
      SELECT MIN(salary)
      FROM EMPLOYEES
      WHERE employee_id IN (
         SELECT manager_id
         FROM DEPARTMENTS
         WHERE manager_id IS NOT NULL
      )
   )
);
```

10. Employees earning more than average salary of employees hired after 01-JAN-2005

```
SELECT employee_id, first_name, last_name, salary
FROM EMPLOYEES
WHERE salary > (
   SELECT AVG(salary)
   FROM EMPLOYEES
   WHERE hire_date > TO_DATE('01-JAN-2005', 'DD-MON-YYYY')
);
```

11. Departments in the same city as department 20

```
SELECT department_id, department_name, location_id
FROM DEPARTMENTS
WHERE location_id = (
    SELECT location_id
    FROM DEPARTMENTS
    WHERE department_id = 20
);
```

12. Employees hired after the most recently hired employee in department 90

```
SELECT first_name, last_name, hire_date
FROM EMPLOYEES
WHERE hire_date > (
    SELECT MAX(hire_date)
    FROM EMPLOYEES
    WHERE department_id = 90
);
```

13. Employees who are not managers

```
SELECT employee_id
FROM EMPLOYEES
WHERE employee_id NOT IN (
    SELECT manager_id
    FROM DEPARTMENTS
    WHERE manager_id IS NOT NULL
);
```

14. Employees with second highest salary in their department

```
SELECT first_name, last_name, salary, department_id
FROM EMPLOYEES e
WHERE salary = (
    SELECT MAX(salary)
    FROM EMPLOYEES
    WHERE department_id = e.department_id
      AND salary < (
          SELECT MAX(salary)
          FROM EMPLOYEES
          WHERE department_id = e.department_id
      )
);
```

15. Departments with more employees than David’s department

```
SELECT department_name
FROM DEPARTMENTS
WHERE department_id IN (
   SELECT department_id
   FROM EMPLOYEES
   GROUP BY department_id
   HAVING COUNT(*) > (
      SELECT COUNT(*)
      FROM EMPLOYEES
      WHERE first_name = 'David'
      GROUP BY department_id
   )
);
```

16. Employees with the same job as employee 176

```
SELECT first_name, last_name, job_id
FROM EMPLOYEES
WHERE job_id = (
   SELECT job_id
   FROM EMPLOYEES
   WHERE employee_id = 176
);
```

17. City of employee who earns minimum salary

```
SELECT l.city
FROM EMPLOYEES e
JOIN DEPARTMENTS d ON e.department_id = d.department_id
JOIN LOCATIONS l ON d.location_id = l.location_id
WHERE e.salary = (SELECT MIN(salary) FROM EMPLOYEES);
```

18. Employees earning more than department 60’s average salary

```
SELECT first_name, last_name
FROM EMPLOYEES
WHERE salary > (
   SELECT AVG(salary)
   FROM EMPLOYEES
   WHERE department_id = 60
);
```

19. Employees in departments with no manager assigned

```
SELECT employee_id, first_name, last_name
FROM EMPLOYEES
WHERE department_id IN (
  SELECT department_id
  FROM DEPARTMENTS
  WHERE manager_id IS NULL
);
```

20. Department with maximum number of employees

```
SELECT department_name
FROM DEPARTMENTS
WHERE department_id = (
  SELECT department_id
  FROM EMPLOYEES
  GROUP BY department_id
  ORDER BY COUNT(*) DESC
  FETCH FIRST 1 ROW ONLY
);
```

Tables Recap

```
Matches(match_id, match_date, venue_id, team1_id, team2_id, winner_id)
Teams(team_id, team_name, country)
Players(player_id, player_name, team_id)
Match_Scores(match_id, player_id, score)
Venues(venue_id, venue_name, city)
```

1️⃣ Players who scored more than 100 in any single match

```
SELECT DISTINCT p.player_id, p.player_name
FROM Players p, Match_Scores ms
WHERE p.player_id = ms.player_id
GROUP BY p.player_id, ms.match_id
HAVING SUM(ms.score) > 100;
```

- ✅ Groups by `player_id` and `match_id` to correctly sum scores per match.
- `DISTINCT` ensures each player appears only once, even if they exceeded 100 in multiple matches.

2️⃣ Venues where the Pakistan team has won a match

```
SELECT DISTINCT v.venue_name
FROM Venues v, Matches m, Teams t
WHERE v.venue_id = m.venue_id
  AND t.team_name = 'Pakistan'
  AND (t.team_id = m.team1_id OR t.team_id = m.team2_id)
  AND m.winner_id = t.team_id;
```

- Classic join style.
- Simple `DISTINCT` to avoid repeating the same venue multiple times.

3️⃣ Players who have a score of exactly 50 in any match

```
SELECT DISTINCT p.player_id, p.player_name
FROM Players p, Match_Scores ms
WHERE p.player_id = ms.player_id
  AND ms.score = 50;
```

- No grouping needed here because you're checking individual rows for a score of 50.

4️⃣ Players whose average score across all matches is over 75

```
SELECT p.player_id, p.player_name
FROM Players p, Match_Scores ms
WHERE p.player_id = ms.player_id
GROUP BY p.player_id
HAVING AVG(ms.score) > 75;
```

- ✅ Grouped only by `player_id` (primary key), not name.
- Selecting `player_name` is safe because it's functionally dependent on `player_id`.

5️⃣ Players who scored more than 100 in a match played at Gaddafi Stadium

```
SELECT DISTINCT p.player_id, p.player_name
FROM Players p, Match_Scores ms, Matches m, Venues v
WHERE p.player_id = ms.player_id
  AND ms.match_id = m.match_id
  AND m.venue_id  = v.venue_id
  AND v.venue_name = 'Gaddafi Stadium'
GROUP BY p.player_id, ms.match_id
HAVING SUM(ms.score) > 100;
```

- Same aggregation logic as query #1, but restricted to a specific venue.
- `DISTINCT` ensures players are not repeated if they scored >100 in multiple matches at Gaddafi Stadium.