**PART1**

```
create database unionDB
use unionDB

-- Trainees Table
CREATE TABLE Trainees (
TraineeID INT PRIMARY KEY,
FullName VARCHAR(100),
Email VARCHAR(100),
Program VARCHAR(50),
GraduationDate DATE
);

-- Job Applicants Table
CREATE TABLE Applicants (
ApplicantID INT PRIMARY KEY,
FullName VARCHAR(100),
Email VARCHAR(100),
Source VARCHAR(20), -- e.g., "Website", "Referral"
AppliedDate DATE
);

-- Insert into Trainees
INSERT INTO Trainees VALUES
(1, 'Layla Al Riyami', 'layla.r@example.com', 'Full Stack .NET', '2025-04-30'),
(2, 'Salim Al Hinai', 'salim.h@example.com', 'Outsystems', '2025-03-15'),
(3, 'Fatma Al Amri', 'fatma.a@example.com', 'Database Admin', '2025-05-01');
```

-- Insert into Applicants

INSERT INTO Applicants VALUES

(101, 'Hassan Al Lawati', 'hassan.l@example.com', 'Website', '2025-05-02'),

(102, 'Layla Al Riyami', 'layla.r@example.com', 'Referral', '2025-05-05'), -- same person as

(103, 'Aisha Al Farsi', 'aisha.f@example.com', 'Website', '2025-04-28');

-- **1.** Unique people who either trained or applied

SELECT FullName, Email FROM Trainees

UNION

SELECT FullName, Email FROM Applicants;

**-- Observation:**

-- UNION removes duplicates based on all selected columns.

| | FullName | Email |
|---|---|---|
| 1 | Aisha Al Farsi | aisha.f@example.com |
| 2 | Fatma Al Amri | fatma.a@example.com |
| 3 | Hassan Al Lawati | hassan.l@example.com |
| 4 | Layla Al Riyami | layla.r@example.com |
| 5 | Salim Al Hinai | salim.h@example.com |

-- **2**. Using UNION ALL to show all SELECT FullName, Email FROM Trainees UNION ALL SELECT FullName, Email FROM Applicants;

-- **Observation:**

-- Duplicate entries appear if the same person is in both tables.

-- Explanation: UNION ALL does not eliminate duplicates.

| | FullName | Email |
|---|---|---|
| 1 | Layla Al Riyami | layla.r@example.com |
| 2 | Salim Al Hinai | salim.h@example.com |
| 3 | Fatma Al Amri | fatma.a@example.com |
| 4 | Hassan Al Lawati | hassan.l@example.com |
| 5 | Layla Al Riyami | layla.r@example.com |
| 6 | Aisha Al Farsi | aisha.f@example.com |

-- **3.** People in both tables (INTERSECT or JOIN) -- If your DB supports INTERSECT: SELECT FullName, Email FROM Trainees INTERSECT SELECT FullName, Email FROM Applicants;

**-- Simulated using JOIN:**

SELECT t.FullName, t.Email FROM Trainees t INNER JOIN Applicants a ON t.Email = a.Email;

| | FullName | Email |
|---|---|---|
| 1 | Layla Al Riyami | layla.r@example.com |

**PART2**

-- **1.** DELETE test DELETE FROM Trainees WHERE Program = 'Outsystems';

-- **Observation:**

-- Rows where Program is 'Outsystems' are deleted. -- Table and structure still exist.

```
(1 row affected)

Completion time: 2025-05-27T10:01:05.2151889+04:00
```

-- **2.** TRUNCATE test TRUNCATE TABLE Applicants;

-- **Observation:**

-- All data is removed instantly. -- Cannot be rolled back in most databases (non-transactional).

-- Table structure remains intact.

-- **3**. DROP test DROP TABLE Applicants;

**-- Observation:**

-- Table is deleted completely.

-- SELECT on Applicants now results in "Table does not exist" error.

**PART3**

- What is SQL Transaction?

  A **SQL transaction** is a sequence of operations performed as a single logical unit of work. These operations can include reading, writing, updating, or deleting data from a database. Transactions ensure that the database remains in a consistent state even in the event of errors or failures. They are fundamental to maintaining data integrity and are governed by the ACID properties: Atomicity, Consistency, Isolation, and Durability
  -- A transaction is a group of SQL statements that are executed as a single unit.
  -- It follows ACID properties to ensure reliability.
  -- BEGIN TRANSACTION starts.
  -- COMMIT saves all changes.
  -- ROLLBACK undoes changes if an error occurs.
- Transaction Script with failure

```
BEGIN TRANSACTION;
INSERT INTO Applicants VALUES
(104, 'Zahra Al Amri', 'zahra.a@example.com', 'Referral', '2025-05-10');

-- This will fail due to duplicate ID (if 104 already exists)
INSERT INTO Applicants VALUES
(104, 'Error User', 'error@example.com', 'Website', '2025-05-11');
```

-- COMMIT:  Will not be reached if error occurs

-- ROLLBACK: Rollback must be handled in application or manually

The **ACID properties** are a foundational concept in database management systems that ensure reliable processing of transactions. ACID stands for **Atomicity, Consistency, Isolation, and Durability**—each addressing a critical aspect of how data changes should be handled to preserve accuracy, integrity, and stability, especially in multi-user or error-prone environments.

**Atomicity** means that a transaction is treated as a single, indivisible unit. It must either be fully completed or not executed at all. If any part of the transaction fails, the entire transaction is rolled back so that the database remains unchanged. For example, in a bank transfer, if money is withdrawn from one account but fails to be credited to another, the withdrawal is reversed to maintain balance integrity. This prevents partial updates from corrupting data.

**Consistency** ensures that a transaction transforms the database from one valid state to another. It enforces all defined rules, such as constraints, cascades, or triggers. If a transaction would violate

any integrity rule (e.g., inserting an order with a non-existent customer ID), it will be rejected. This guarantees that the database never contains invalid or contradictory information.

**Isolation** refers to the ability of the database to allow multiple transactions to occur simultaneously without interfering with each other. Even when transactions run at the same time, each one should execute independently, as if it were alone. This avoids problems like dirty reads, where one transaction reads uncommitted data from another, leading to incorrect results. For example, two users trying to purchase the last unit of a product will not end up both buying it—only one will succeed.

**Durability** means that once a transaction has been committed, it becomes permanent. Even if the system crashes or loses power immediately afterward, the committed data will not be lost. This is typically ensured through mechanisms like write-ahead logs, backups, and redundant storage systems.

Together, these four principles—**Atomicity, Consistency, Isolation, and Durability**—guarantee that database transactions are processed safely and reliably. They are essential for any system that handles critical data, such as banking, e-commerce, or health records.