SQL Set Operations

Database Setup

```sql
-- Delete tables if they exist
DROP TABLE IF EXISTS Completions;
DROP TABLE IF EXISTS Players;
DROP TABLE IF EXISTS Levels;

-- Create tables
CREATE TABLE Players (PlayerId INT, Name VARCHAR(50));
CREATE TABLE Levels (LevelId INT, LevelName VARCHAR(50));
CREATE TABLE Completions (PlayerId INT, LevelId INT);

-- Insert 7 players
INSERT INTO Players VALUES
(1, 'Alex'),
(2, 'Beth'),
(3, 'Carl'),
(4, 'Diana'),
(5, 'Erik'),
(6, 'Fiona'),
(7, 'Greg');

-- Insert 3 levels
INSERT INTO Levels VALUES
(1, 'Forest'),
(2, 'Cave'),
(3, 'Castle');

-- Insert completions: 3 players complete all levels, 4 players with different
patterns
INSERT INTO Completions VALUES
-- Alex, Beth, Carl: Completed all 3 levels
(1, 1), (1, 2), (1, 3),
(2, 1), (2, 2), (2, 3),
(3, 1), (3, 2), (3, 3),

-- Diana: Only completed Forest
(4, 1),
```

```
-- Erik: Completed Forest and Cave
(5, 1), (5, 2),

-- Fiona: Only completed Castle
(6, 3),

-- Greg: Completed Forest and Castle
(7, 1), (7, 3);
```

Data Overview

We have seven players and three game levels. The Completions table tracks which players have completed which levels.

Here's what makes our data interesting:

- **Alex, Beth, and Carl** are our completionists - they've finished all three levels
- **Diana** only tackled the Forest level
- **Erik** completed Forest and Cave
- **Fiona** only conquered the Castle
- **Greg** finished Forest and Castle

---

1. UNION/UNION ALL (+/∪) - Addition/Combining Sets

Mathematical Definition: `A ∪ B = {x | x ∈ A or x ∈ B}`

SQL Definition: Find records that appear in either set A or set B or both.

Key Tips:

- **UNION:** Removes duplicates
- **UNION ALL:** Keeps duplicates

Example Question

Find all players who completed either Forest **OR** Cave levels

```sql
SELECT PlayerId FROM Completions WHERE LevelId = 1  -- Forest completers
UNION
SELECT PlayerId FROM Completions WHERE LevelId = 2; -- Cave completers
-- Note: Fiona (6) is excluded - Only completed Castle (Level=3)
```

---

2. CROSS JOIN (×) - Multiplication

Mathematical Definition: `A × B = {(x, y) | x ∈ A, y ∈ B}`

SQL Definition: Creates all possible combinations (Cartesian product)

Example Question

Find all possible Player-Level combinations

```sql
SELECT p.PlayerId, p.Name, l.LevelId, l.LevelName
FROM Players p CROSS JOIN Levels l
ORDER BY p.PlayerId, l.LevelId;
-- Result: 7 players × 3 levels = 21 total combinations
-- Shows every player paired with every level (whether completed or not)
```

---

3. INTERSECT (∩) - Set Intersection/Common Elements

Mathematical Definition: `A ∩ B = {x | x ∈ A and x ∈ B}`

SQL Definition: Find records that appear in both set A and set B.

Example Question

Find players who completed Forest **AND** Cave levels

```sql
SELECT PlayerId FROM Completions WHERE LevelId = 1  -- Forest completers
INTERSECT
SELECT PlayerId FROM Completions WHERE LevelId = 2; -- Cave completers
-- Excluded players:
-- Diana (4): Only completed Forest (Level=1)
-- Fiona (6): Only completed Castle (Level=3)
-- Greg (7): Completed Forest and Castle (Levels=1,3)
```

---

4. EXCEPT/MINUS (-) - Subtraction/Difference

Mathematical Definition: `A - B = {x | x ∈ A and x ∉ B}`

SQL Definition: Find records that are in set A but not in set B.

Example Question

Find players who completed Forest **but NOT** Cave

Solution 1: EXCEPT (MINUS in Oracle)

```sql
SELECT PlayerId FROM Completions WHERE LevelId = 1  -- Forest completers
EXCEPT
SELECT PlayerId FROM Completions WHERE LevelId = 2; -- Cave completers
-- Excluded players:
-- Alex (1), Beth (2), Carl (3), Erik (5) have completed Level=2
-- Fiona (6) has not completed Level=1
```

> **Note:** Using EXCEPT is easy and makes more sense, but there are other ways to simulate it.

Solution 2: NOT IN

```sql
SELECT PlayerId
FROM Completions
WHERE LevelId = 1
  AND PlayerId NOT IN (
    SELECT PlayerId FROM Completions WHERE LevelId = 2
  );
```

Solution 3: LEFT JOIN with WHERE NULL

```sql
SELECT c1.PlayerId
FROM (SELECT DISTINCT PlayerId FROM Completions WHERE LevelId = 1) c1
LEFT JOIN (SELECT DISTINCT PlayerId FROM Completions WHERE LevelId = 2) c2
  ON c1.PlayerId = c2.PlayerId
WHERE c2.PlayerId IS NULL;
-- This filters for players who exist in Forest (c1) but have NO match in Cave
(c2)
-- LEFT JOIN keeps all c1 records, setting c2 fields to NULL when no match
exists
```

Solution 4: NOT EXISTS

```sql
SELECT DISTINCT PlayerId
FROM Completions c1
WHERE c1.LevelId = 1  -- Forest completers
  AND NOT EXISTS (
    SELECT 1 -- The existence of a row matters not the value of it
    FROM Completions c2
    WHERE c2.PlayerId = c1.PlayerId
      AND c2.LevelId = 2  -- Cave completers
  );
```

Why Use SELECT 1?

Because NOT EXISTS only cares about: "Does the subquery return ANY rows?"

- If subquery returns 0 rows → NOT EXISTS = TRUE
- If subquery returns 1+ rows → NOT EXISTS = FALSE
- So, SELECT 1 returns the actual value 1, and it means row(s) exist.

---

5. DIVISION (÷) - The Most Important Part

> **Important:** SQL doesn't have a built-in division operator. We need to simulate it completely.

SQL Definition: Find records in set A that are related to **ALL** records in set B.

Alternative Definition: Find records in set A that are related to **EVERY** record in set B.

Example Questions

- Find players who completed **ALL** levels
- Find players who have completed **EVERY** level

Solution 1: Double NOT EXISTS (Most Common Approach)

For this approach, we should rephrase the question to something that has two NOs or two NOTs.

Original: "Find players who completed ALL levels"
Rephrased: "Find players where there does NOT exist a level that they did NOT complete." (Two NOTs)

```sql
-- Find players where there does NOT exist a level that they did NOT complete.
SELECT p.PlayerId, p.Name
FROM Players p
WHERE NOT EXISTS ( -- does NOT exist a level
    SELECT 1 -- The existence of a row matters not the value of it
    FROM Levels l
    WHERE NOT EXISTS ( -- did NOT complete
        SELECT 1
        FROM Completions c
        WHERE c.PlayerId = p.PlayerId
        AND c.LevelId = l.LevelId
    )
);
```

Understanding the Two NOT EXISTS

First NOT EXISTS (Outer)

- **Purpose:** Filters which items from the "ALL" set to consider
- **Can have conditions:** YES - to subset the items you're checking against

Second NOT EXISTS (Inner)

- **Purpose:** Checks if the action/relationship exists
- **Always has conditions:** YES - must link the entities and define the action

Example with Conditions

```sql
-- If we add 'Difficulty' column in our Level table, then we can address:
-- Find players who completed ALL HARD levels
SELECT p.PlayerId, p.Name
FROM Players p
WHERE NOT EXISTS ( -- First NOT EXISTS can filter levels
    SELECT 1
    FROM Levels l
    WHERE l.Difficulty = 'HARD'  -- CONDITION IN FIRST NOT EXISTS
    AND NOT EXISTS ( -- Second NOT EXISTS checks completion
        SELECT 1
        FROM Completions c
        WHERE c.PlayerId = p.PlayerId
        AND c.LevelId = l.LevelId
    )
);
```

Pattern Recognition

Let's try rephrasing approach with other examples to find a pattern:

- **Find students enrolled in ALL required courses**
  BECOMES: Find students where there is NO required course that they are NOT enrolled in. (NO & NOT)

- **Find customers who bought ALL sale items**
  BECOMES: Find customers where it's NOT true that there exists a sale item they didn't buy. (Two NOTs)

The Universal Pattern

```sql
-- Find entities that [ACTION] ALL [ITEMS]
-- BECOMES
-- Find entities where there is NO [ITEM] that they did NOT [ACTION]
--
-- So we have:
--
-- WHERE NOT EXISTS ( -- NO [ITEM] (first)
-- ... subset the items we're checking against (optional)
--
--     WHERE NOT EXISTS ( -- NOT [ACTION] (second)
--     ... link the entities and define the action (mandatory)
```

Solution 2: COUNT Comparison (Explicit JOIN)

If someone completed all levels, their completion count equals total levels.

```sql
SELECT p.PlayerId, p.Name
FROM Players p
JOIN Completions c ON p.PlayerId = c.PlayerId
JOIN Levels l ON c.LevelId = l.LevelId
GROUP BY p.PlayerId, p.Name
HAVING COUNT(DISTINCT c.LevelId) = (SELECT COUNT(*) FROM Levels);
-- Why DISTINCT matters: If completions table had duplicates
-- (player completed same level twice), we'd still count correctly
```

Solution 3: EXCEPT/MINUS

Player qualifies if (All Levels - Their Completions) = Empty Set

```sql
SELECT p.PlayerId, p.Name
FROM Players p
WHERE NOT EXISTS ( -- Empty Set
    SELECT LevelId FROM Levels -- All Levels
    EXCEPT
    SELECT LevelId FROM Completions c WHERE c.PlayerId = p.PlayerId -- Their
Completions
);
```

Solution 4: Filtered COUNT (Subquery)

First find all valid completions, then check if count is complete

```sql
SELECT PlayerId, Name
FROM Players
WHERE PlayerId IN (
    SELECT c.PlayerId
    FROM Completions c
    WHERE c.LevelId IN (SELECT LevelId FROM Levels) -- Filter valid levels
    GROUP BY c.PlayerId -- Group by player
    HAVING COUNT(DISTINCT c.LevelId) = (SELECT COUNT(*) FROM Levels) -- Count =
total
);
-- DISTINCT is crucial - prevents counting duplicate completions
```