Set Operations and Subqueries

Complementary SQL: Solutions

May 15, 2025

Contents

Global Dataset for PostgreSQL

The following SQL code creates and populates the necessary tables for the exercises. This dataset should be loaded into your PostgreSQL environment before running the solutions.

```sql
-- Drop tables if they exist to ensure a clean setup
DROP TABLE IF EXISTS EmployeeProjects CASCADE;
DROP TABLE IF EXISTS Sales CASCADE;
DROP TABLE IF EXISTS Products CASCADE;
DROP TABLE IF EXISTS OldEmployees CASCADE;
DROP TABLE IF EXISTS CandidateEmployees CASCADE;
DROP TABLE IF EXISTS OnLeaveEmployees CASCADE;
DROP TABLE IF EXISTS Employees CASCADE;
DROP TABLE IF EXISTS Departments CASCADE;
DROP TABLE IF EXISTS Projects CASCADE;

-- Create Tables
CREATE TABLE Departments (
    departmentId INT PRIMARY KEY,
    departmentName VARCHAR(100) NOT NULL UNIQUE,
    locationCity VARCHAR(50)
);

CREATE TABLE Employees (
    employeeId INT PRIMARY KEY,
    firstName VARCHAR(50) NOT NULL,
    lastName VARCHAR(50) NOT NULL,
    email VARCHAR(100) UNIQUE,
    phoneNumber VARCHAR(20),
    hireDate DATE NOT NULL,
    jobId VARCHAR(20),
    salary NUMERIC(10, 2) NOT NULL CHECK (salary > 0),
    commissionPct NUMERIC(4, 2) CHECK (commissionPct >= 0 AND commissionPct <= 1),
    managerId INT REFERENCES Employees(employeeId),
    departmentId INT REFERENCES Departments(departmentId)
);

CREATE TABLE Projects (
    projectId INT PRIMARY KEY,
    projectName VARCHAR(100) NOT NULL UNIQUE,
    departmentId INT REFERENCES Departments(departmentId),
    startDate DATE,
    endDate DATE,
    budget NUMERIC(12, 2) CHECK (budget >= 0)
);

CREATE TABLE EmployeeProjects (
    employeeId INT REFERENCES Employees(employeeId),
    projectId INT REFERENCES Projects(projectId),
    assignedRole VARCHAR(50),
    hoursWorked INT CHECK (hoursWorked >= 0),
    PRIMARY KEY (employeeId, projectId)
);

CREATE TABLE OldEmployees (
    employeeId INT PRIMARY KEY,
    firstName VARCHAR(50),
    lastName VARCHAR(50),
    lastDepartmentId INT,
    terminationDate DATE NOT NULL,
    finalSalary NUMERIC(10, 2),
    reasonForLeaving VARCHAR(255)
);

CREATE TABLE CandidateEmployees (
    candidateId INT PRIMARY KEY,
    firstName VARCHAR(50),
    lastName VARCHAR(50),
    appliedPosition VARCHAR(100),
    expectedSalary NUMERIC(10, 2),
    applicationDate DATE
);
```

```sql
CREATE TABLE OnLeaveEmployees (
    employeeId INT PRIMARY KEY REFERENCES Employees(employeeId),
    leaveStartDate DATE,
    leaveEndDate DATE,
    leaveReason VARCHAR(100)
);

CREATE TABLE Products (
    productId INT PRIMARY KEY,
    productName VARCHAR(100) NOT NULL,
    productCategory VARCHAR(50),
    unitPrice NUMERIC(10, 2) CHECK (unitPrice > 0)
);

CREATE TABLE Sales (
    saleId INT PRIMARY KEY,
    employeeId INT REFERENCES Employees(employeeId),
    productId INT REFERENCES Products(productId),
    saleDate TIMESTAMP NOT NULL,
    quantitySold INT CHECK (quantitySold > 0),
    saleAmount NUMERIC(10, 2)
);

-- Populate Tables
INSERT INTO Departments (departmentId, departmentName, locationCity) VALUES
(1, 'Human Resources', 'New York'),
(2, 'Engineering', 'San Francisco'),
(3, 'Sales', 'Chicago'),
(4, 'Marketing', 'New York'),
(5, 'Finance', 'London'),
(6, 'Research', 'San Francisco'),
(7, 'Customer Support', 'Austin');

INSERT INTO Employees (employeeId, firstName, lastName, email, phoneNumber, hireDate,
    jobId, salary, commissionPct, managerId, departmentId) VALUES
(101, 'John', 'Smith', 'john.smith@example.com', '555-1234', '2018-06-01', 'HR_REP',
    60000, NULL, NULL, 1),
(102, 'Alice', 'Johnson', 'alice.j@example.com', '555-5678', '2019-03-15', 'ENG_LEAD',
    90000, NULL, NULL, 2),
(103, 'Bob', 'Williams', 'bob.w@example.com', '555-8765', '2019-07-20', 'SALES_MGR',
    75000, 0.10, NULL, 3),
(104, 'Eva', 'Brown', 'eva.b@example.com', '555-4321', '2020-01-10', 'MKT_SPEC', 65000,
    NULL, 101, 4),
(105, 'Charlie', 'Davis', 'charlie.d@example.com', '555-1122', '2018-11-05', 'ENG_SR',
    85000, NULL, 102, 2),
(106, 'Diana', 'Miller', 'diana.m@example.com', '555-6543', '2021-05-25', 'SALES_REP',
    55000, 0.05, 103, 3),
(107, 'Frank', 'Wilson', 'frank.w@example.com', '555-7890', '2022-08-01', 'ENG_JR',
    70000, NULL, 102, 2),
(108, 'Grace', 'Moore', 'grace.m@example.com', '555-2109', '2019-09-01', 'FIN_ANALYST',
    72000, NULL, NULL, 5),
(109, 'Henry', 'Taylor', 'henry.t@example.com', '555-1098', '2023-02-15', 'ENG_JR',
    68000, NULL, 105, 2),
(110, 'Ivy', 'Anderson', 'ivy.a@example.com', '555-8076', '2020-11-30', 'MKT_MGR',
    80000, NULL, 101, 4),
(111, 'Jack', 'Thomas', 'jack.t@example.com', '555-7654', '2017-07-14', 'RES_SCI',
    95000, NULL, NULL, 6),
(112, 'Karen', 'Jackson', 'karen.j@example.com', '555-6547', '2021-10-01', 'HR_ASSIST',
    50000, NULL, 101, 1),
(113, 'Leo', 'White', 'leo.w@example.com', '555-5438', '2023-05-20', 'SALES_REP', 58000,
     0.06, 103, 3),
(114, 'Mia', 'Harris', 'mia.h@example.com', '555-4329', '2019-02-18', 'FIN_MGR', 92000,
    NULL, 108, 5),
(115, 'Noah', 'Martin', 'noah.m@example.com', '555-3210', '2022-06-10', 'RES_ASSIST',
    60000, NULL, 111, 6),
(116, 'Olivia', 'Garcia', 'olivia.g@example.com', '555-1987', '2018-09-01', 'ENG_SR',
    88000, NULL, 102, 2),
(117, 'Paul', 'Martinez', 'paul.m@example.com', '555-8760', '2023-01-05', 'SALES_INTERN'
    , 40000, 0.02, 106, 3),
(118, 'Quinn', 'Robinson', 'quinn.r@example.com', '555-7651', '2020-07-07', 'MKT_INTERN'
    , 42000, NULL, 104, 4),
```

```sql
(119, 'Ruby', 'Clark', 'ruby.c@example.com', '555-6542', '2022-03-03', 'HR_SPEC', 62000, NULL, 101, 1),
(120, 'Sam', 'Rodriguez', 'sam.r@example.com', '555-5433', '2021-11-11', 'ENG_TECH', 72000, NULL, 105, 2),
(121, 'Tom', 'Lee', 'tom.lee@example.com', '555-1122', '2023-08-15', 'FIN_ANALYST', 73000, NULL, 114, 5),
(122, 'Ursula', 'Walker', 'ursula.w@example.com', '555-2233', '2019-01-20', 'RES_HEAD', 120000, NULL, NULL, 6),
(123, 'Victor', 'Hall', 'victor.h@example.com', '555-3344', '2020-05-10', 'ENG_LEAD', 95000, NULL, NULL, 2),
(124, 'Wendy', 'Allen', 'wendy.a@example.com', '555-4455', '2021-09-01', 'MKT_COORD', 63000, NULL, 110, 4),
(125, 'Xavier', 'Young', 'xavier.y@example.com', '555-5566', '2022-12-12', 'SALES_LEAD', 78000, 0.08, 103, 3),
(126, 'Yara', 'King', 'yara.k@example.com', '555-6677', '2018-04-04', 'HR_MGR', 85000, NULL, NULL, 1),
(127, 'Zack', 'Wright', 'zack.w@example.com', '555-7788', '2023-07-01', 'ENG_INTERN', 45000, NULL, 107, 2),
(128, 'Laura', 'Palmer', 'laura.p@example.com', '555-1111', '2023-01-15', 'SUPPORT_REP', 52000, NULL, NULL, 7),
(129, 'Dale', 'Cooper', 'dale.c@example.com', '555-2222', '2023-02-20', 'SUPPORT_LEAD', 65000, NULL, NULL, 7);

UPDATE Employees SET managerId = 102 WHERE employeeId IN (105, 107, 116, 120);
UPDATE Employees SET managerId = 123 WHERE employeeId IN (109, 127);
UPDATE Employees SET managerId = 103 WHERE employeeId IN (106, 113, 117, 125);
UPDATE Employees SET managerId = 110 WHERE employeeId IN (104, 118, 124);
UPDATE Employees SET managerId = 101 WHERE employeeId IN (112, 119);
UPDATE Employees SET managerId = 126 WHERE employeeId = 101;
UPDATE Employees SET managerId = 111 WHERE employeeId = 115;
UPDATE Employees SET managerId = 122 WHERE employeeId = 111;
UPDATE Employees SET managerId = 114 WHERE employeeId IN (108, 121);
UPDATE Employees SET managerId = 129 WHERE employeeId = 128;

INSERT INTO Projects (projectId, projectName, departmentId, startDate, endDate, budget) VALUES
(1, 'Alpha Launch', 4, '2023-01-01', '2023-06-30', 150000.00),
(2, 'Beta Platform', 2, '2022-09-01', '2024-03-31', 500000.00),
(3, 'Gamma Sales Drive', 3, '2023-03-01', '2023-09-30', 80000.00),
(4, 'Delta HR System', 1, '2023-02-01', '2023-12-31', 120000.00),
(5, 'Epsilon Research', 6, '2022-05-01', '2024-05-30', 300000.00),
(6, 'Zeta Finance Tool', 5, '2023-07-01', '2024-06-30', 200000.00),
(7, 'Omega Security Update', 2, '2023-10-01', '2024-01-31', 250000.00),
(8, 'Sigma Marketing Campaign', 4, '2024-01-15', '2024-07-15', 180000.00),
(9, 'Kappa Efficiency Audit', 5, '2022-11-01', '2023-04-30', 75000.00),
(10, 'New Support Portal', 7, '2023-05-01', '2023-11-30', 90000.00);

INSERT INTO EmployeeProjects (employeeId, projectId, assignedRole, hoursWorked) VALUES
(102, 2, 'Project Lead', 500), (105, 2, 'Senior Developer', 600), (107, 2, 'Junior Developer', 450), (116, 2, 'Senior Developer', 550), (120, 2, 'Technician', 400), (123, 7, 'Project Lead', 200),
(104, 1, 'Marketing Specialist', 300), (110, 1, 'Campaign Manager', 250), (124, 1, 'Coordinator', 320),
(106, 3, 'Sales Representative', 400), (113, 3, 'Sales Representative', 380), (125, 3, 'Lead Sales', 350),
(101, 4, 'HR Lead', 200), (112, 4, 'HR Assistant', 300), (119, 4, 'HR Specialist', 280),
(111, 5, 'Lead Scientist', 700), (115, 5, 'Research Assistant', 650), (122, 5, 'Principal Investigator', 500),
(108, 6, 'Financial Analyst', 300), (114, 6, 'Finance Lead', 250), (121, 6, 'Analyst', 320),
(105, 7, 'Security Consultant', 150), (107, 7, 'Developer', 180),
(108, 9, 'Auditor', 200), (114, 9, 'Audit Lead', 150),
(128, 10, 'Support Analyst', 350), (129, 10, 'Project Manager', 280);

INSERT INTO OldEmployees (employeeId, firstName, lastName, lastDepartmentId, terminationDate, finalSalary, reasonForLeaving) VALUES
(201, 'Gary', 'Oldman', 2, '2022-12-31', 82000, 'Retired'),
(202, 'Helen', 'Hunt', 3, '2023-03-15', 70000, 'New Opportunity'),
(203, 'Mike', 'Myers', 2, '2021-08-20', 90000, 'Relocation'),
(204, 'Olivia', 'Garcia', 2, '2023-11-01', 88000, 'New Opportunity');

INSERT INTO CandidateEmployees (candidateId, firstName, lastName, appliedPosition, expectedSalary, applicationDate) VALUES
```

```
(301, 'Peter', 'Pan', 'ENG_JR', 65000, '2023-10-01'),
(302, 'Wendy', 'Darling', 'MKT_SPEC', 68000, '2023-09-15'),
(303, 'John', 'Smith', 'HR_REP', 60000, '2023-11-01'),
(304, 'Alice', 'Wonder', 'ENG_LEAD', 92000, '2023-08-20'),
(305, 'Bruce', 'Wayne', 'FIN_ANALYST', 75000, '2023-11-05');

INSERT INTO OnLeaveEmployees (employeeId, leaveStartDate, leaveEndDate, leaveReason)
    VALUES
(104, '2023-11-01', '2024-02-01', 'Maternity Leave'),
(111, '2023-09-15', '2023-12-15', 'Sabbatical');

INSERT INTO Products (productId, productName, productCategory, unitPrice) VALUES
(1, 'AlphaWidget', 'Electronics', 49.99),
(2, 'BetaGear', 'Software', 199.00),
(3, 'GammaCore', 'Hardware', 120.50),
(4, 'DeltaService', 'Services', 75.00),
(5, 'EpsilonPlus', 'Electronics', 89.90);

INSERT INTO Sales (saleId, employeeId, productId, saleDate, quantitySold, saleAmount)
    VALUES
(1, 106, 1, '2023-04-10 10:30:00', 2, 99.98),
(2, 106, 3, '2023-04-12 14:00:00', 1, 120.50),
(3, 113, 2, '2023-05-05 11:15:00', 1, 199.00),
(4, 106, 1, '2023-05-20 16:45:00', 3, 149.97),
(5, 117, 4, '2023-06-01 09:00:00', 10, 750.00),
(6, 125, 5, '2023-07-10 12:30:00', 5, 449.50),
(7, 113, 1, '2023-07-15 15:00:00', 2, 99.98),
(8, 103, 2, '2023-08-01 10:00:00', 2, 398.00),
(9, 106, 3, '2023-08-18 13:20:00', 1, 120.50),
(10, 125, 2, '2023-09-05 17:00:00', 1, 199.00),
(11, 113, 5, '2023-11-10 09:30:00', 3, 269.70),
(12, 106, 4, '2023-11-15 11:45:00', 5, 375.00),
(13, 117, 1, '2023-11-20 14:15:00', 1, 49.99),
(14, 125, 3, '2023-12-01 10:00:00', 2, 241.00),
(15, 103, 5, '2023-12-05 16:30:00', 4, 359.60),
(16, 128, 2, '2023-06-15 10:00:00', 1, 199.00),
(17, 105, 3, '2023-07-20 11:00:00', 1, 120.50);
```

Listing 1: Global Dataset for Exercises

1 Set Operations - Solutions

1.1 Type (i): Meaning, values, relations, advantages

1.1.1 SO-1.1: Unified List of All Current and Potential Engineering Staff

```sql
SELECT E.firstName || ' ' || E.lastName AS fullName, 'Current Employee' AS sourceType
FROM Employees E
JOIN Departments D ON E.departmentId = D.departmentId
WHERE D.departmentName = 'Engineering'
UNION
SELECT C.firstName || ' ' || C.lastName AS fullName, 'Candidate' AS sourceType
FROM CandidateEmployees C
WHERE C.appliedPosition IN ('ENG_LEAD', 'ENG_JR')
ORDER BY fullName;
```

This exercise shows **UNION** *'s ability to merge results from different queries with compatible columns and automatically remove duplicates. The advantage is a clean, consolidated list without manual deduplication.*

1.1.2 SO-1.2: Log of All Employee-Project Assignments and Recent Terminations

```sql
SELECT employeeId, CAST(projectId AS VARCHAR) AS activityIdentifier, 'Project Assignment' AS activityType
FROM EmployeeProjects
UNION ALL
SELECT employeeId, CAST(terminationDate AS VARCHAR) AS activityIdentifier, 'Termination' AS activityType
FROM OldEmployees
WHERE EXTRACT(YEAR FROM terminationDate) = 2023
ORDER BY employeeId, activityType;
```

This exercise highlights **UNION ALL***. Its advantage is speed, as it doesn't perform duplicate checking, which is suitable when all records from all sources are desired, like in a log.*

1.1.3 SO-1.3: Employees in Sales and Marketing Departments

```sql
-- Employees in 'Sales' for 'Gamma Sales Drive'
SELECT E.employeeId, E.firstName, E.lastName
FROM Employees E
JOIN EmployeeProjects EP ON E.employeeId = EP.employeeId
JOIN Projects P ON EP.projectId = P.projectId
WHERE P.projectName = 'Gamma Sales Drive'
INTERSECT
-- Employees in 'Marketing' department
SELECT E.employeeId, E.firstName, E.lastName
FROM Employees E
JOIN Departments D ON E.departmentId = D.departmentId
WHERE D.departmentName = 'Marketing'
ORDER BY lastName, firstName;
```

This exercise shows **INTERSECT** *for finding records present in both result sets. Its advantage is clearly expressing the need for common elements.*

1.1.4 SO-1.4: Active Engineers Not Assigned to 'Omega Security Update' Project

```sql
-- All Engineers
SELECT E.employeeId, E.firstName, E.lastName
FROM Employees E
JOIN Departments D ON E.departmentId = D.departmentId
WHERE D.departmentName = 'Engineering'
EXCEPT
-- Engineers on 'Omega Security Update'
SELECT E.employeeId, E.firstName, E.lastName
FROM Employees E
JOIN EmployeeProjects EP ON E.employeeId = EP.employeeId
JOIN Projects P ON EP.projectId = P.projectId
WHERE P.projectName = 'Omega Security Update'
ORDER BY lastName, firstName;
```

This exercise uses `EXCEPT` to subtract one set of employees from another. Its advantage is a concise way to find elements unique to the first query's result set.

1.2 Type (ii): Disadvantages

1.2.1 SO-2.1: Mismatched Column Data Types in Union

```sql
-- This query will likely cause an error or implicit cast with potential issues:
-- SELECT employeeId, hireDate FROM Employees
-- UNION
-- SELECT candidateId, expectedSalary FROM CandidateEmployees;

-- Corrected version (if the goal was to show IDs and a numeric value):
SELECT employeeId AS id, salary AS amount, 'Employee' as source FROM Employees
WHERE departmentId = 2 -- Example filter
UNION
SELECT candidateId AS id, expectedSalary AS amount, 'Candidate' as source FROM CandidateEmployees
WHERE appliedPosition LIKE 'ENG%';
```

Disadvantage: Set operations require that corresponding columns in the `SELECT` lists have compatible data types. A direct `UNION` of `hireDate` (DATE) and `expectedSalary` (NUMERIC) would fail or lead to undesirable implicit conversions. The solution shows how to select compatible types (or cast explicitly if combining conceptually different but type-compatible data).

1.2.2 SO-2.2: Performance of UNION vs UNION ALL with Large Datasets (Conceptual)

```sql
-- Potentially faster if duplicates are acceptable or known to be few
SELECT firstName FROM Employees
UNION ALL
SELECT firstName FROM CandidateEmployees;

-- Potentially slower due to deduplication, especially with large tables
```

```sql
SELECT firstName FROM Employees
UNION
SELECT firstName FROM CandidateEmployees;
```

Disadvantage: `UNION` performs a distinct sort/hash operation to remove duplicates, which can be resource-intensive (CPU, memory, I/O) on large datasets compared to `UNION ALL` which simply concatenates results. If duplicates are acceptable or if the queries are designed such that no duplicates are possible, `UNION ALL` is generally more performant.

1.3 Type (iii): Inefficient alternatives

1.3.1 SO-3.1: Simulating EXCEPT with NOT IN (and its NULL pitfall)

Inefficient/Potentially Risky (if subquery could return NULLs for the compared column):

```sql
SELECT E.employeeId, E.firstName, E.lastName
FROM Employees E
JOIN Departments D ON E.departmentId = D.departmentId
WHERE D.departmentName = 'Engineering'
  AND E.employeeId NOT IN (SELECT OL.employeeId FROM OnLeaveEmployees
    OL);
```

More Efficient/Robust (using EXCEPT):

```sql
SELECT E.employeeId, E.firstName, E.lastName
FROM Employees E
JOIN Departments D ON E.departmentId = D.departmentId
WHERE D.departmentName = 'Engineering'
EXCEPT
SELECT OL.employeeId, E.firstName, E.lastName
FROM OnLeaveEmployees OL
JOIN Employees E ON OL.employeeId = E.employeeId;
```

Explanation: The `NOT IN` approach is common but has a pitfall: if the subquery for `NOT IN` returns even one NULL value, the entire `NOT IN` condition will evaluate to UNKNOWN or FALSE for all rows, potentially yielding incorrect results. `EXCEPT` (or `NOT EXISTS`) is generally safer and often more efficient for "anti-join" patterns. Here, `EXCEPT` clearly expresses the set difference and handles NULLs predictably.

1.3.2 SO-3.2: Simulating INTERSECT with Multiple Joins/WHERE conditions

Using `INTERSECT`:

```sql
SELECT E.employeeId, E.firstName
FROM Employees E
JOIN Departments D ON E.departmentId = D.departmentId
WHERE D.departmentName = 'Engineering'
INTERSECT
SELECT E.employeeId, E.firstName
FROM Employees E
JOIN EmployeeProjects EP ON E.employeeId = EP.employeeId
JOIN Projects P ON EP.projectId = P.projectId
WHERE P.projectName = 'Beta Platform'
ORDER BY employeeId;
```

Alternative using JOINs:

```sql
SELECT DISTINCT E.employeeId, E.firstName
FROM Employees E
JOIN Departments D ON E.departmentId = D.departmentId
JOIN EmployeeProjects EP ON E.employeeId = EP.employeeId
JOIN Projects P ON EP.projectId = P.projectId
WHERE D.departmentName = 'Engineering' AND P.projectName = 'Beta
    Platform'
ORDER BY E.employeeId;
```

Explanation: In this case, the JOIN approach is quite common and often efficient. However, if the conditions for being in each "set" were much more complex and involved different tables or logic, ***INTERSECT*** *can make the query more readable by clearly separating the criteria for each set. The "inefficiency" of avoiding* ***INTERSECT*** *isn't always about raw performance but can be about code clarity and maintainability when set logic is primary.*

1.4 Type (iv): Hardcore problem

1.4.1 SO-4.1: Consolidated List of High-Value Personnel Not On Leave, Ranked

```sql
WITH ValuablePersonnel AS (
    SELECT
        E.employeeId,
        E.firstName,
        E.lastName,
        E.salary AS amount,
        'Current Employee' AS personnelType
    FROM Employees E
    WHERE E.salary > 70000
    UNION ALL
    SELECT
        OE.employeeId,
        OE.firstName,
        OE.lastName,
        OE.finalSalary AS amount,
        'Former Employee' AS personnelType
    FROM OldEmployees OE
    WHERE OE.finalSalary > 75000 AND OE.reasonForLeaving IN ('New
    Opportunity', 'Retired')
),
ActiveValuablePersonnel AS (
    SELECT vp.employeeId, vp.firstName, vp.lastName, vp.amount, vp.
    personnelType
    FROM ValuablePersonnel vp
    EXCEPT
    SELECT ole.employeeId, vp2.firstName, vp2.lastName, vp2.amount, vp2
    .personnelType
    FROM OnLeaveEmployees ole
    JOIN ValuablePersonnel vp2 ON ole.employeeId = vp2.employeeId
)
SELECT
    avp.employeeId,
    avp.firstName,
    avp.lastName,
    avp.amount,
```

```
    avp.personnelType,
    RANK() OVER (PARTITION BY avp.personnelType ORDER BY avp.amount
   DESC) as salaryRankByType
FROM ActiveValuablePersonnel avp
ORDER BY avp.personnelType, salaryRankByType, avp.lastName;
```

*This hardcore problem uses **`UNION ALL`** to combine two distinct sets of personnel, **`EXCEPT`** to filter out those on leave, Common Table Expressions (CTEs) for readability and modularity, and a **`RANK()`** window function to rank them as required. It also involves basic joins and filtering conditions (**`WHERE`**).*

2 Subqueries - Solutions

2.1 Type (i): Meaning, values, relations, advantages

2.1.1 SQ-1.1: Employees Earning More Than Average (Scalar Subquery)

```sql
SELECT employeeId, firstName, lastName, salary
FROM Employees
WHERE salary > (SELECT AVG(salary) FROM Employees)
ORDER BY salary DESC;
```

Advantage: Scalar subqueries allow dynamic comparison values. The average salary is calculated at runtime and used for filtering.

2.1.2 SQ-1.2: Departments with Higher Than Average Project Budgets (Subquery in FROM)

```sql
SELECT
    D.departmentName,
    DeptAvgBudgets.avgBudget
FROM Departments D
JOIN (
    SELECT departmentId, AVG(budget) AS avgBudget
    FROM Projects
    WHERE departmentId IS NOT NULL
    GROUP BY departmentId
) AS DeptAvgBudgets ON D.departmentId = DeptAvgBudgets.departmentId
WHERE DeptAvgBudgets.avgBudget > (SELECT AVG(budget) FROM Projects)
ORDER BY DeptAvgBudgets.avgBudget DESC;
```

*Advantage: Subqueries in the **FROM** clause allow for pre-aggregation or transformation of data into an intermediate result set that can then be joined or filtered.*

2.1.3 SQ-1.3: Employee's Project Count (Subquery in SELECT - Correlated)

```sql
SELECT
    E.employeeId,
    E.firstName,
    E.lastName,
    (SELECT COUNT(EP.projectId)
     FROM EmployeeProjects EP
     WHERE EP.employeeId = E.employeeId) AS projectCount
FROM Employees E
ORDER BY projectCount DESC, E.lastName;
```

*Advantage: Correlated subqueries in **SELECT** can compute a specific value for each row of the outer query, useful for fetching related summary data.*

2.1.4 SQ-1.4: Employees in Departments Located in 'New York' (Subquery in WHERE with IN)

```sql
SELECT employeeId, firstName, lastName
FROM Employees
```

```sql
WHERE departmentId IN (SELECT departmentId FROM Departments WHERE
    locationCity = 'New York')
ORDER BY lastName, firstName;
```

Advantage: `IN` with a subquery is concise for checking membership in a dynamically generated list of values.

2.1.5 SQ-1.5: Departments with at Least One Project (Subquery in WHERE with EXISTS)

```sql
SELECT D.departmentId, D.departmentName
FROM Departments D
WHERE EXISTS (SELECT 1 FROM Projects P WHERE P.departmentId = D.
    departmentId)
ORDER BY D.departmentName;
```

Advantage: `EXISTS` is efficient for checking the existence of related rows. It stops processing the subquery as soon as a match is found.

2.1.6 SQ-1.6: Employees Earning More Than Any Sales Intern (Subquery in WHERE with ANY/SOME)

```sql
SELECT employeeId, firstName, lastName, salary
FROM Employees
WHERE salary > ANY (SELECT salary FROM Employees WHERE jobId = '
    SALES_INTERN')
  AND jobId != 'SALES_INTERN' -- Exclude sales interns themselves if
    desired
ORDER BY salary;
```

Advantage: `ANY` (or `SOME`) allows comparison against a set of values. `> ANY` means greater than the minimum value in the set returned by the subquery.

2.1.7 SQ-1.7: Employees Earning More Than All Sales Interns (Subquery in WHERE with ALL)

```sql
SELECT employeeId, firstName, lastName, salary
FROM Employees
WHERE salary > ALL (SELECT salary FROM Employees WHERE jobId = '
    SALES_INTERN')
ORDER BY salary;
```

Advantage: `ALL` allows comparison against a set of values. `> ALL` means greater than the maximum value in the set returned by the subquery.

2.2 Type (ii): Disadvantages

2.2.1 SQ-2.1: Performance of Correlated Subquery in SELECT

Solution (Correlated Subquery):

```sql
SELECT
    E.employeeId,
    E.firstName,
    E.departmentId,
    (SELECT P.projectName
```

```sql
    FROM Projects P
    WHERE P.departmentId = E.departmentId -- Correlation
    ORDER BY P.budget DESC
    LIMIT 1) AS mostExpensiveDeptProject
FROM Employees E
WHERE E.departmentId IS NOT NULL;
```

Disadvantage: Correlated subqueries, especially in the **SELECT** *list or* **WHERE** *clause, execute repeatedly for each row of the outer query. If the subquery is complex or operates on large tables, this can lead to poor performance.*

2.2.2 SQ-2.2: Scalar Subquery Returning Multiple Rows (Error Scenario)

```sql
-- This will error if the subquery returns more than one row:
-- SELECT employeeId, firstName, salary
-- FROM Employees
-- WHERE salary = (SELECT DISTINCT salary FROM Employees E JOIN
   Departments D ON E.departmentId = D.departmentId WHERE D.
   departmentName = 'Sales');

-- Corrected version using IN:
SELECT E.employeeId, E.firstName, E.salary
FROM Employees E
WHERE E.salary IN (SELECT DISTINCT Em.salary FROM Employees Em JOIN
   Departments Dp ON Em.departmentId = Dp.departmentId WHERE Dp.
   departmentName = 'Sales');
```

Disadvantage: A scalar subquery (one used where a single value is expected) must return exactly one row and one column. If it returns more than one row, it will result in a runtime error.

2.3 Type (iii): Inefficient alternatives

2.3.1 SQ-3.1: Finding Max Salary Without Scalar Subquery (Inefficient Application Logic)

Inefficient (conceptual application-level two-step): 1. `SELECT MAX(salary) FROM Employees;` (Result e.g., 120000) 2. `SELECT employeeId, firstName, salary FROM Employees WHERE salary = 120000;`

Efficient (using a scalar subquery):

```sql
SELECT employeeId, firstName, lastName, salary
FROM Employees
WHERE salary = (SELECT MAX(salary) FROM Employees);
```

Explanation: Performing two separate queries from the application adds network latency and complexity. A scalar subquery allows the database to handle this in a single, optimized operation.

2.3.2 SQ-3.2: Checking Existence of Sales by Engineers (Inefficient: Fetching All Sales Data)

Inefficient (conceptual: fetching lots of data then checking): `SELECT * FROM Sales;` (and then process in application to join with Engineers)

Efficient (using `EXISTS` with a correlated subquery):

```sql
SELECT D.departmentName
FROM Departments D
WHERE D.departmentName = 'Engineering'
  AND EXISTS (SELECT 1
              FROM Sales S
              JOIN Employees E ON S.employeeId = E.employeeId
              WHERE E.departmentId = D.departmentId);
-- To list the engineers:
SELECT E.employeeId, E.firstName
FROM Employees E
JOIN Departments D ON E.departmentId = D.departmentId
WHERE D.departmentName = 'Engineering'
  AND EXISTS (SELECT 1 FROM Sales S WHERE S.employeeId = E.employeeId);
```

*Explanation: The inefficient alternative involves fetching potentially large amounts of sales data. The **EXISTS** subquery is highly efficient because the database can stop searching as soon as the first matching sale is found.*

2.4 Type (iv): Hardcore problem

2.4.1 SQ-4.1: Departmental High Earners Analysis

```sql
WITH OverallAvgSalaryPost2020 AS (
    SELECT AVG(salary) AS globalAvgSalary
    FROM Employees
    WHERE hireDate >= '2020-01-01'
),
DepartmentalStats AS (
    SELECT
        D.departmentId,
        D.departmentName,
        AVG(E.salary) AS avgDeptSalary,
        SUM(P.budget) AS totalDeptBudget
    FROM Departments D
    JOIN Employees E ON D.departmentId = E.departmentId
    LEFT JOIN Projects P ON D.departmentId = P.departmentId
    GROUP BY D.departmentId, D.departmentName
    HAVING AVG(E.salary) > (SELECT globalAvgSalary FROM OverallAvgSalaryPost2020)
),
RankedEmployeesInQualifiedDepts AS (
    SELECT
        E.employeeId,
        E.firstName,
        E.lastName,
        E.salary,
        E.departmentId,
        DS.departmentName,
        DS.avgDeptSalary,
        DS.totalDeptBudget,
        ROW_NUMBER() OVER (PARTITION BY E.departmentId ORDER BY E.salary DESC) AS rn
    FROM Employees E
    JOIN DepartmentalStats DS ON E.departmentId = DS.departmentId
)
SELECT
    RE.departmentName,
```

```sql
    RE.firstName,
    RE.lastName,
    RE.salary,
    (RE.salary - RE.avgDeptSalary) AS salaryDiffFromDeptAvg,
    COALESCE(RE.totalDeptBudget, 0) AS departmentTotalProjectBudget
FROM RankedEmployeesInQualifiedDepts RE
WHERE RE.rn <= 2
ORDER BY RE.departmentName, RE.salary DESC;
```

This hardcore problem utilizes: Scalar Subquery, Subquery in FROM (via CTEs), Aggregators, Joins, Window Functions, Basic SQL, and Null Handling to solve a multi-step analytical problem.