# Conditionals: Advanced ORDER BY

Complementary SQL: Solutions

May 11, 2025

## Contents

# Dataset for PostgreSQL

The following SQL code creates and populates the necessary tables for the exercises. This dataset is identical to the one provided in the exercises document.

```sql
-- Drop tables if they exist to ensure a clean setup
DROP TABLE IF EXISTS employee_projects CASCADE;
DROP TABLE IF EXISTS employees CASCADE;
DROP TABLE IF EXISTS departments CASCADE;

-- Create departments table
CREATE TABLE departments (
    department_name VARCHAR(50) PRIMARY KEY,
    location VARCHAR(50),
    budget_allocation NUMERIC(12,2)
);

-- Populate departments table
INSERT INTO departments (department_name, location, budget_allocation) VALUES
('Engineering', 'New York', 500000.00),
('Marketing', 'San Francisco', 300000.00),
('Sales', 'Chicago', 400000.00),
('Human Resources', 'New York', 250000.00),
('Product', 'San Francisco', 350000.00),
('Support', 'Remote', 150000.00);

-- Create employees table
CREATE TABLE employees (
    id SERIAL PRIMARY KEY,
    first_name VARCHAR(50),
    last_name VARCHAR(50),
    department VARCHAR(50) REFERENCES departments(department_name),
    salary NUMERIC(10, 2),
    hire_date DATE,
    bonus_percentage NUMERIC(3, 2), -- Nullable
    manager_id INTEGER REFERENCES employees(id) -- Nullable
);

-- Populate employees table
INSERT INTO employees (first_name, last_name, department, salary, hire_date,
    bonus_percentage, manager_id) VALUES
('Alice', 'Smith', 'Engineering', 90000.00, '2020-03-15', 0.10, NULL),
('Bob', 'Johnson', 'Engineering', 95000.00, '2019-07-01', 0.12, 1),
('Charlie', 'Williams', 'Marketing', 70000.00, '2021-01-10', 0.08, NULL),
('David', 'Brown', 'Sales', 80000.00, '2020-11-05', NULL, NULL), -- Null bonus
('Eve', 'Jones', 'Engineering', 90000.00, '2021-05-20', 0.10, 1),
('Frank', 'Garcia', 'Marketing', 72000.00, '2022-02-01', NULL, 3), -- Null bonus
('Grace', 'Miller', 'Sales', 82000.00, '2019-05-20', 0.09, 4),
('Heidi', 'Davis', 'Human Resources', 65000.00, '2023-01-15', 0.05, NULL),
('Ivan', 'Rodriguez', 'Product', 110000.00, '2020-08-24', 0.15, NULL),
('Judy', 'Martinez', 'Product', 105000.00, '2021-06-10', NULL, 9), -- Null bonus
('Kevin', 'Hernandez', 'Engineering', 88000.00, '2023-03-01', 0.07, 2),
('Linda', 'Lopez', 'Marketing', 68000.00, '2023-04-10', 0.06, 3),
('Mike', 'Gonzalez', 'Sales', 78000.00, '2022-07-18', 0.11, 4),
('Nancy', 'Wilson', 'Human Resources', 67000.00, '2022-09-01', NULL, 8), -- Null bonus
('Olivia', 'Anderson', 'Engineering', 90000.00, '2020-03-15', NULL, 1), -- Null bonus,
    same salary/hire_date as Alice
('Peter', 'Lee', 'Product', 100000.00, '2021-08-15', 0.12, 9),
('Zoe', 'King', 'Engineering', 92000.00, '2022-05-01', 0.11, 1),
('Yasmin', 'Scott', 'Marketing', 75000.00, '2021-11-20', NULL, 3),
('Eva', 'Taylor', 'Engineering', 90000.00, '2021-05-20', NULL, 1); -- Same salary/
    hire_date as Eve, but NULL bonus

-- Create employee_projects table
CREATE TABLE employee_projects (
    employee_id INTEGER REFERENCES employees(id),
    project_name VARCHAR(100),
    project_role VARCHAR(50),
    hours_assigned INTEGER,
    PRIMARY KEY (employee_id, project_name)
);

```

```sql
-- Populate employee_projects table
INSERT INTO employee_projects (employee_id, project_name, project_role, hours_assigned)
    VALUES
(1, 'Alpha Platform', 'Developer', 120),
(1, 'Beta Feature', 'Lead Developer', 80),
(2, 'Alpha Platform', 'Senior Developer', 150),
(3, 'Campaign X', 'Coordinator', 100),
(5, 'Gamma Initiative', 'Developer', 90),
(5, 'Alpha Platform', 'Tester', 40),
(6, 'Campaign Y', 'Analyst', 110),
(7, 'Client Outreach', 'Manager', 60),
(9, 'Omega Product', 'Product Owner', 180),
(10, 'Omega Product', 'UX Designer', 70),
(11, 'Gamma Initiative', 'Junior Developer', 100),
(13, 'Client Retention', 'Specialist', 50);
```

Listing 1: Dataset for Advanced ORDER BY Exercises

# 1 Solutions: Practice Meanings, Values, Relations, and Advantages

## 1.1 Solution 1.1: Ordering by Multiple Columns

```sql
SELECT first_name, last_name, department, salary
FROM employees
ORDER BY department ASC, salary DESC;
```

Listing 2: Solution to Exercise 1.1

*Problem solved with the given dataset:* This query retrieves employee names, their department, and salary from the `employees` table. The results are structured to first group employees by the `department` column, and then, within each department, arrange them from the highest to the lowest `salary`. This demonstrates the advantage of multi-column ordering in creating a more detailed and hierarchical view of the data, allowing for precise control over data presentation.

## 1.2 Solution 1.2: Using `NULLS FIRST`

```sql
SELECT first_name, last_name, department, bonus_percentage
FROM employees
ORDER BY bonus_percentage ASC NULLS FIRST;
```

Listing 3: Solution to Exercise 1.2

*Problem solved with the given dataset:* This query lists employees along with their department and bonus percentages from the `employees` table. The specific requirement is to prioritize employees with `NULL` values in the `bonus_percentage` column by showing them first. They are followed by other employees sorted by their `bonus_percentage` from lowest to highest. This highlights the advantage and value of `NULLS FIRST` in explicitly handling and positioning records with missing data at the beginning of the sorted result set, which can be useful for review or special handling.

## 1.3 Solution 1.3: Using `NULLS LAST` and Multiple Columns

```sql
SELECT first_name, last_name, hire_date, bonus_percentage
FROM employees
WHERE department = 'Engineering'
ORDER BY hire_date ASC, bonus_percentage DESC NULLS LAST;
```

Listing 4: Solution to Exercise 1.3

*Problem solved with the given dataset:* This query focuses on employees in the 'Engineering' `department`. The sorting criteria are: primarily by `hire_date` (earliest first). For ties in `hire_date`, employees who have a value in their `bonus_percentage` column are prioritized (and sorted by the highest bonus first), while those with `NULL` in `bonus_percentage` are placed at the end of that `hire_date` group. This demonstrates combining multi-column sort with `NULLS LAST` for refined ordering, where records with missing data in a secondary sort key are deprioritized within the primary sort groups.

# 2 Solutions: Practice Disadvantages

## 2.1 Solution 2.1: Disadvantage of Overly Complex Sorting (Readability/Maintainability)

```sql
-- This is a conceptual question. The query below is for context.
SELECT first_name, last_name, department, hire_date, salary,
   bonus_percentage
FROM employees
ORDER BY
    department ASC,
    (CASE WHEN hire_date < '2021-01-01' THEN 0 ELSE 1 END) ASC, --
   Custom logic for hire_date grouping
    salary DESC NULLS LAST,                                    -- Salary
   with NULLs at the end
    (bonus_percentage IS NULL) ASC,                            --
   Prioritizes non-NULL bonus_percentage
    last_name ASC;
```

Listing 5: Conceptual Query for Exercise 2.1

*Problem illustrated by the example:* The provided `ORDER BY` clause sorts employee data from the `employees` table using five distinct criteria, including `CASE` expressions to create synthetic sort keys and explicit `NULLS LAST`. The main disadvantage is significantly reduced readability and maintainability. Such complexity makes the query's intent hard to grasp quickly, increases the likelihood of errors during modification, and makes it difficult for other developers to understand and debug. While powerful, excessive complexity in `ORDER BY` can outweigh the benefits if simpler, more declarative constructs (like direct use of `NULLS FIRST/LAST` where appropriate instead of `CASE` for null handling) could achieve similar results with greater clarity.

## 2.2 Solution 2.2: Disadvantage of Potentially Misleading Prioritization with `NULLS FIRST/LAST`

```sql
-- This is a conceptual question. The query below is for context.
SELECT first_name, last_name, bonus_percentage
FROM employees
ORDER BY bonus_percentage ASC NULLS FIRST;
```

Listing 6: Conceptual Query for Exercise 2.2

*Problem illustrated by the example:* The query sorts employees by `bonus_percentage` ascending, placing `NULL` values first. If a large portion of employees have `NULL` for `bonus_percentage` in the `employees` table, these `NULL` records will dominate the top of the report. The disadvantage is that this might obscure the employees with actual low bonus percentages, or it might give undue prominence to records where the bonus is simply unknown or not applicable. If the intent was to focus on those with *defined* low bonuses, `NULLS FIRST` could be counterproductive, making the initial part of the report less informative about the actual data distribution of non-null bonuses. The choice of `NULLS FIRST` vs. `NULLS LAST` must align with the semantic meaning of `NULL` in that context.

# 3 Solutions: Practice Cases of Lost Advantages

## 3.1 Solution 3.1: Inefficient Simulation of NULLS FIRST

```sql
-- Less direct / potentially less clear approach using CASE:
SELECT first_name, last_name, department, bonus_percentage
FROM employees
ORDER BY (CASE WHEN bonus_percentage IS NULL THEN 0 ELSE 1 END) ASC,
    bonus_percentage ASC;

-- Preferred Advanced ORDER BY approach:
SELECT first_name, last_name, department, bonus_percentage
FROM employees
ORDER BY bonus_percentage ASC NULLS FIRST;
```

Listing 7: Solution to Exercise 3.1 - Comparing Approaches

*Problem solved by comparing approaches:* The first query uses a `CASE` statement on the `bonus_percentage` column from the `employees` table. It assigns a sort key (0 for `NULL`, 1 for non-`NULL`) to bring `NULL` bonus percentages to the top, then sorts by the actual `bonus_percentage`. The second query achieves the identical result using `bonus_percentage ASC NULLS FIRST`. The `NULLS FIRST` approach is preferred because it is more concise, more declarative (clearly states the intent of `NULL` handling), and generally easier to read and understand. It directly leverages a built-in SQL feature designed for this purpose, potentially allowing the database optimizer to handle the sorting more efficiently than a generic `CASE` expression. This avoids the common mistake of using a more verbose method when a specialized, simpler one exists.

## 3.2 Solution 3.2: Inefficient Custom Sort Order Implementation

```sql
-- Superior Advanced ORDER BY Approach:
SELECT first_name, last_name, department, salary
FROM employees
ORDER BY
    CASE department
        WHEN 'Sales' THEN 1
        WHEN 'Engineering' THEN 2
        ELSE 3
    END ASC,           -- Primary sort: Custom department order
    department ASC,    -- Secondary sort: Alphabetical for 'other'
    departments
    salary DESC;       -- Tertiary sort: Salary within each department
    group
```

Listing 8: Solution to Exercise 3.2 - Efficient Approach

*Problem solved by the efficient approach:* The query sorts employees from the `employees` table. It first establishes a custom order for `department` values ('Sales', then 'Engineering', then others) using a `CASE` expression. If multiple departments fall into the 'ELSE 3' category, they are then sorted alphabetically by `department` name. Finally, within each of these defined department groupings, employees are sorted by `salary` in descending order. This single-query solution is superior to manual multi-query `UNION ALL` approaches because it is significantly more efficient (requires only one scan or pass over the data

by the database engine), is much cleaner to write and maintain, and directly utilizes the power of the `ORDER BY` clause for complex sorting logic. This prevents the loss of performance and clarity that comes from avoiding advanced `ORDER BY` capabilities.

# 4 Solution: Hardcore Problem Combining Concepts

## 4.1 Solution 4.1: Comprehensive Employee Ranking and Reporting

```sql
WITH DepartmentTotalHours AS (
    -- Calculate total project hours per department
    -- Uses: Aggregation (SUM), GROUP BY, HAVING, LEFT JOIN (to include employees/departments even if no projects assigned to specific employees)
    SELECT
        e.department,
        SUM(COALESCE(ep.hours_assigned, 0)) AS total_department_hours
    FROM employees e
    LEFT JOIN employee_projects ep ON e.id = ep.employee_id
    GROUP BY e.department
    HAVING SUM(COALESCE(ep.hours_assigned, 0)) > 200
),
RankedEmployees AS (
    -- Select and rank employees based on criteria
    -- Uses: Joins (INNER JOIN with departments), WHERE (date, list, subquery),
    -- Window Function (RANK() with PARTITION BY and complex ORDER BY including NULLS LAST),
    -- String concatenation
    SELECT
        e.first_name || ' ' || e.last_name AS full_name,
        e.department,
        d.location,
        e.salary,
        e.hire_date,
        e.bonus_percentage,
        RANK() OVER (
            PARTITION BY e.department
            ORDER BY
                e.salary DESC,
                e.bonus_percentage DESC NULLS LAST, -- Key: Handles tie-breaking with bonus, NULLs last
                e.hire_date ASC
        ) AS department_rank
    FROM employees e
    JOIN departments d ON e.department = d.department_name
    WHERE e.hire_date >= '2021-01-01'                                    -- Basic SQL: WHERE condition
      AND d.location IN ('New York', 'San Francisco')                    -- Basic SQL: WHERE IN
      AND e.department IN (SELECT department FROM DepartmentTotalHours)  -- Complementary SQL: Subquery in WHERE
)
-- Final selection and ordering
-- Uses: COALESCE (Null Space), final ORDER BY (multi-column)
SELECT
    full_name,
    department,
    salary,
    hire_date,
    COALESCE(bonus_percentage::TEXT, 'N/A') AS bonus_percentage_display
```

```
    , -- Intermediate SQL: COALESCE, Casters
45     department_rank
46 FROM RankedEmployees
47 ORDER BY department ASC, department_rank ASC; -- Basic and Advanced
    ORDER BY
```

Listing 9: Solution to Exercise 4.1

*Problem solved with the given dataset:* This query addresses a complex reporting requirement by integrating multiple SQL concepts.

1. It first identifies `departments` that are active enough (total project hours ¿ 200) using a Common Table Expression (CTE `DepartmentTotalHours`) that joins `employees` and `employee_projects`, aggregates `hours_assigned` using `SUM()`, and filters with `HAVING`. `COALESCE` handles employees with no projects.

2. The main CTE (`RankedEmployees`) then filters `employees` based on `hire_date` (from `employees`), `location` (from `departments` via a `JOIN`), and membership in the previously identified active departments (using a subquery with `IN`).

3. Crucially, for these filtered employees, it calculates a `department_rank` using the `RANK()` window function. This rank is partitioned by `department`. The `ORDER BY` clause within `RANK()` implements the specified multi-level tie-breaking logic: `salary DESC` first, then `bonus_percentage DESC NULLS LAST` (which correctly places non-NULL bonuses first, then sorts them descending, and places all NULL bonuses after non-NULL ones), and finally `hire_date ASC`. This demonstrates a sophisticated use of "Advanced ORDER BY" concepts (`NULLS LAST`, multi-column ordering) within an analytical function.

4. The final `SELECT` statement formats the `bonus_percentage` for display (using `COALESCE` and `CAST`) and orders the entire result set by `department` and then by the calculated `department_rank`.

This exercise showcases how "Advanced ORDER BY" features are essential not only for final result sorting but also within window functions to achieve complex analytical rankings, combined with various other SQL techniques from basic to intermediate levels. The dataset, with its `NULL`able `bonus_percentage`, varied salaries, hire dates, and linked project data, provides the necessary complexity to make such a problem meaningful.