Date Functions, Cases, and Null Space

Complementary SQL: Solutions

May 13, 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 leave_requests CASCADE;
DROP TABLE IF EXISTS employee_projects CASCADE;
DROP TABLE IF EXISTS projects CASCADE;
DROP TABLE IF EXISTS employees CASCADE;
DROP TABLE IF EXISTS departments CASCADE;

-- Departments Table
CREATE TABLE departments (
    dept_id SERIAL PRIMARY KEY,
    dept_name VARCHAR(100) NOT NULL UNIQUE,
    creation_date DATE NOT NULL,
    location VARCHAR(50)
);

-- Employees Table
CREATE TABLE employees (
    emp_id SERIAL PRIMARY KEY,
    emp_name VARCHAR(100) NOT NULL,
    hire_date DATE NOT NULL,
    salary NUMERIC(10, 2),
    dept_id INT REFERENCES departments(dept_id),
    manager_id INT REFERENCES employees(emp_id), -- Self-reference for manager
    termination_date DATE, -- NULL if currently employed
    email VARCHAR(100) UNIQUE,
    performance_rating INT CHECK (performance_rating BETWEEN 1 AND 5 OR performance_rating IS NULL)
);

-- Projects Table
CREATE TABLE projects (
    project_id SERIAL PRIMARY KEY,
    project_name VARCHAR(100) NOT NULL UNIQUE,
    start_date DATE NOT NULL,
    planned_end_date DATE NOT NULL,
    actual_end_date DATE, -- NULL if not completed
    budget NUMERIC(12, 2),
    lead_emp_id INT REFERENCES employees(emp_id),
    CONSTRAINT check_project_dates CHECK (planned_end_date >= start_date)
);

-- Employee_Projects Table (Junction table)
CREATE TABLE employee_projects (
    emp_project_id SERIAL PRIMARY KEY,
    emp_id INT REFERENCES employees(emp_id) ON DELETE CASCADE,
    project_id INT REFERENCES projects(project_id) ON DELETE CASCADE,
    assigned_date DATE NOT NULL,
    role VARCHAR(50),
    hours_billed NUMERIC(6, 2) DEFAULT 0.00,
    billing_rate NUMERIC(8, 2),
    completion_date DATE, -- Date employee completed their part
    UNIQUE(emp_id, project_id) -- An employee has one role per project
);

-- Leave_Requests Table
CREATE TABLE leave_requests (
    leave_id SERIAL PRIMARY KEY,
    emp_id INT REFERENCES employees(emp_id) ON DELETE CASCADE,
    request_date TIMESTAMP WITHOUT TIME ZONE DEFAULT CURRENT_TIMESTAMP,
    leave_start_date DATE NOT NULL,
    leave_end_date DATE NOT NULL,
    status VARCHAR(20) CHECK (status IN ('Pending', 'Approved', 'Rejected', 'Cancelled')),
    approved_by_manager_id INT REFERENCES employees(emp_id),
    reason TEXT,
    CONSTRAINT check_leave_dates CHECK (leave_end_date >= leave_start_date)
);
```

```sql

-- Populate Departments
INSERT INTO departments (dept_name, creation_date, location) VALUES
('Human Resources', '2010-01-15', 'New York'),
('Engineering', '2010-03-01', 'San Francisco'),
('Sales', '2010-02-01', 'Chicago'),
('Marketing', '2011-05-20', 'New York'),
('Finance', '2010-01-20', 'New York');

-- Populate Employees (managers first, then their reports)
INSERT INTO employees (emp_name, hire_date, salary, dept_id, manager_id, termination_date, email, performance_rating) VALUES
('Alice Wonderland', '2015-06-01', 90000.00, 2, NULL, NULL, 'alice@example.com', 5), -- Eng Manager
('David Copperfield', '2015-03-10', 120000.00, 3, NULL, NULL, 'david@example.com', 4), -- Sales Manager
('Frankenstein Monster', '2019-11-01', 95000.00, 1, NULL, NULL, 'frank@example.com', 3), -- HR Manager
('Ivy Poison', '2022-09-15', 110000.00, 5, NULL, NULL, 'ivy@example.com', 4); -- Finance Head

INSERT INTO employees (emp_name, hire_date, salary, dept_id, manager_id, termination_date, email, performance_rating) VALUES
('Bob The Builder', '2016-08-15', 75000.00, 2, (SELECT emp_id from employees WHERE email='alice@example.com'), NULL, 'bob@example.com', 4),
('Carol Danvers', '2017-01-20', 80000.00, 2, (SELECT emp_id from employees WHERE email='alice@example.com'), NULL, 'carol@example.com', 5),
('Eve Harrington', '2018-07-01', 65000.00, 3, (SELECT emp_id from employees WHERE email='david@example.com'), '2023-12-31', 'eve@example.com', 2),
('Grace Hopper', '2020-02-10', 70000.00, 1, (SELECT emp_id from employees WHERE email='frank@example.com'), NULL, 'grace@example.com', NULL),
('Henry Jekyll', '2021-05-01', 50000.00, 4, (SELECT emp_id from employees WHERE email='david@example.com'), NULL, 'henry@example.com', 3), -- Marketing, reports to Sales head for now
('Jack Sparrow', '2023-01-20', 60000.00, 5, (SELECT emp_id from employees WHERE email='ivy@example.com'), '2023-08-15', 'jack@example.com', 1),
('Kevin McCallister', '2018-09-01', 72000.00, 2, (SELECT emp_id from employees WHERE email='alice@example.com'), NULL, 'kevin@example.com', NULL),
('Laura Croft', '2019-04-15', 85000.00, 4, (SELECT emp_id from employees WHERE email='david@example.com'), NULL, 'laura@example.com', 5); -- Marketing, reports to Sales head

-- Populate Projects
INSERT INTO projects (project_name, start_date, planned_end_date, actual_end_date, budget, lead_emp_id) VALUES
('Project Alpha', '2023-01-01', '2023-06-30', '2023-07-15', 100000.00, (SELECT emp_id from employees WHERE email='alice@example.com')),
('Project Beta', '2023-03-01', '2023-09-30', NULL, 150000.00, (SELECT emp_id from employees WHERE email='bob@example.com')),
('Project Gamma', '2022-09-01', '2023-03-31', '2023-03-20', 80000.00, (SELECT emp_id from employees WHERE email='alice@example.com')),
('Project Delta', '2023-08-01', '2024-02-29', NULL, 200000.00, (SELECT emp_id from employees WHERE email='david@example.com')),
('Project Epsilon', '2023-05-01', '2023-08-31', '2023-09-05', 60000.00, NULL),
('Project Zeta', '2024-01-01', '2024-01-30', NULL, 120000.00, (SELECT emp_id from employees WHERE email='bob@example.com')), -- Ends Jan 30, 2024
('Project Omega', '2023-10-01', '2023-12-20', '2023-12-20', 50000.00, (SELECT emp_id from employees WHERE email='carol@example.com')),
('Critical Eng Proj 1', '2024-01-02', '2024-01-25', NULL, 5000, (SELECT emp_id from employees WHERE email='alice@example.com')), -- Critical for report
('Critical Eng Proj 2', '2024-01-05', '2024-02-10', NULL, 5000, (SELECT emp_id from employees WHERE email='bob@example.com')); -- Critical for report

-- Populate Employee_Projects
INSERT INTO employee_projects (emp_id, project_id, assigned_date, role, hours_billed, billing_rate, completion_date) VALUES
((SELECT emp_id from employees WHERE email='alice@example.com'), (SELECT project_id from projects WHERE project_name='Project Alpha'), '2023-01-01', 'Project Manager', 200, 150.00, '2023-07-15'),
((SELECT emp_id from employees WHERE email='alice@example.com'), (SELECT project_id from projects WHERE project_name='Project Gamma'), '2022-09-01', 'Project Manager', 180, 150.00, '2023-03-20'),
((SELECT emp_id from employees WHERE email='bob@example.com'), (SELECT project_id from projects WHERE project_name='Project Alpha'), '2023-01-05', 'Developer', 300, 120.00, '2023-07-10'),
((SELECT emp_id from employees WHERE email='bob@example.com'), (SELECT project_id from projects WHERE project_name='Project Beta'), '2023-03-01', 'Lead Developer', 250, 130.00, NULL),
((SELECT emp_id from employees WHERE email='bob@example.com'), (SELECT project_id from projects WHERE project_name='Project Zeta'), '2024-01-01', 'Lead Developer', 50, 135.00, NULL),
((SELECT emp_id from employees WHERE email='carol@example.com'), (SELECT project_id from projects WHERE project_name='Project Beta'), '2023-03-05', 'Developer', 220, 120.00, NULL),
((SELECT emp_id from employees WHERE email='carol@example.com'), (SELECT project_id from projects WHERE project_name='Project Epsilon'), '2023-05-01', 'Consultant', 100, 200.00, '2023-09-05'),
((SELECT emp_id from employees WHERE email='carol@example.com'), (SELECT project_id from projects WHERE project_name='Project Omega'), '2023-10-01', 'Developer', 80, 125.00, '2023-12-20'),
((SELECT emp_id from employees WHERE email='eve@example.com'), (SELECT project_id from projects WHERE project_name='Project Delta'), '2023-08-01', 'Sales Rep', 150, 100.00, '2023-12-31'),
((SELECT emp_id from employees WHERE email='grace@example.com'), (SELECT project_id from projects WHERE project_name='Project Alpha'), '2023-02-01', 'HR Coordinator', 80, NULL, '2023-07-15'),
((SELECT emp_id from employees WHERE email='kevin@example.com'), (SELECT project_id from projects WHERE project_name='Project Beta'), '2023-03-15', 'QA Engineer', 180, 110.00, NULL),
((SELECT emp_id from employees WHERE email='laura@example.com'), (SELECT project_id from projects WHERE project_name='Project Delta'), '2023-08-05', 'Marketing Lead', 200, 140.00, NULL);

-- Populate Leave_Requests
INSERT INTO leave_requests (emp_id, request_date, leave_start_date, leave_end_date, status, approved_by_manager_id, reason) VALUES
((SELECT emp_id from employees WHERE email='bob@example.com'), '2023-04-01 10:00:00', '2023-04-10', '2023-04-12', 'Approved', (SELECT emp_id from employees WHERE email='alice@example.com'), 'Vacation'),
((SELECT emp_id from employees WHERE email='carol@example.com'), '2023-05-10 14:30:00', '2023-06-01', '2023-06-05', 'Approved', (SELECT emp_id from employees WHERE email='alice@example.com'), 'Personal Leave'),
((SELECT emp_id from employees WHERE email='eve@example.com'), '2023-11-01 09:00:00', '2023-11-10', '2023-11-15', 'Pending', (SELECT emp_id from employees WHERE email='david@example.com'), 'Sick Leave'),
((SELECT emp_id from employees WHERE email='grace@example.com'), '2023-06-15 11:00:00', '2023-07-01', '2023-07-03', 'Rejected', (SELECT emp_id from employees WHERE email='frank@example.com'), NULL),
((SELECT emp_id from employees WHERE email='bob@example.com'), '2023-08-01 16:00:00', '2023-08-20', '2023-08-25', 'Approved', (SELECT emp_id from employees WHERE email='alice@example.com'), 'Family event'),
((SELECT emp_id from employees WHERE email='kevin@example.com'), '2023-09-01 08:00:00', '2023-09-10', '2023-09-11', 'Pending', (SELECT emp_id from employees WHERE email='alice@example.com'), ''),
((SELECT emp_id from employees WHERE email='laura@example.com'), '2024-01-10 10:00:00', '2024-02-01', '2024-02-05', 'Pending', (SELECT emp_id from employees WHERE email='david@example.com'), 'Conference'),
((SELECT emp_id from employees WHERE email='alice@example.com'), '2023-12-01 09:00:00', '2023-12-20', '2023-12-28', 'Approved', NULL, 'Holiday Season'),
((SELECT emp_id from employees WHERE email='bob@example.com'), '2024-01-10 10:00:00', '2024-01-14', '2024-01-16', 'Approved', (SELECT emp_id from employees WHERE email='alice@example.com'), 'Short break'); -- Bob on leave on Jan 15 2024
```

Listing 1: Global Dataset for Exercises

1 Date Functions (Complementary SQL) - Solutions

Concepts: Date arithmetic, OVERLAPS operator.

(i) Meaning, values, relations, advantages of unique usage

Exercise 1.1: Project Timeline Extension and Next Month Check

```sql
SELECT
    project_name,
    planned_end_date AS original_planned_end,
    planned_end_date + INTERVAL '2 months' + INTERVAL '15 days' AS
  extended_planned_end,
    (EXTRACT(YEAR FROM planned_end_date) = 2024 AND EXTRACT(MONTH FROM
  planned_end_date) = 2) AS is_planned_feb_2024
FROM
    projects;
```

Listing 2: Solution for Exercise 1.1

Exercise 1.2: Identifying Concurrent Project Assignments for Employees

```sql
SELECT
    e.emp_name,
    p1.project_name AS project1_name,
    ep1.assigned_date AS p1_assigned,
    ep1.completion_date AS p1_completed,
    p2.project_name AS project2_name,
    ep2.assigned_date AS p2_assigned,
    ep2.completion_date AS p2_completed
FROM
    employee_projects ep1
JOIN
    employees e ON ep1.emp_id = e.emp_id
JOIN
    projects p1 ON ep1.project_id = p1.project_id
JOIN
    employee_projects ep2 ON ep1.emp_id = ep2.emp_id AND ep1.project_id
    < ep2.project_id -- Different projects for the same employee
JOIN
    projects p2 ON ep2.project_id = p2.project_id
WHERE
    (ep1.assigned_date, COALESCE(ep1.completion_date, DATE '9999-12-31'
  )) OVERLAPS
    (ep2.assigned_date, COALESCE(ep2.completion_date, DATE '9999-12-31'
  ));
```

Listing 3: Solution for Exercise 1.2

(ii) Disadvantages of all its technical concepts

Exercise 1.3: OVERLAPS with identical start/end points *Explanation:* In PostgreSQL, when using `(date, date)` with `OVERLAPS` where both dates are identical, this defines an empty interval because the end is not after the start. For example, `(DATE '2023-01-01', DATE '2023-01-01') OVERLAPS (DATE '2023-01-01', DATE '2023-01-05')` will be `FALSE`. This is because an empty interval cannot overlap anything. If the intention is to check if a specific *day* falls within a period, you should ensure the "day" is represented as a one-day interval, e.g., `(start_date, start_date + INTERVAL '1 day')` if your periods are exclusive of the end date, or ensure the comparison handles points correctly. For checking if a single date falls within a range `[S, E]`, it's usually clearer and more direct to use: `single_date >= S AND single_date <= E`.

```sql
-- Example: Project starts on a day an employee is on approved leave.
-- Incorrectly using (start_date, start_date) for OVERLAPS
SELECT
    p.project_name,
    p.start_date,
    e.emp_name,
    lr.leave_start_date,
    lr.leave_end_date,
    (p.start_date, p.start_date) OVERLAPS (lr.leave_start_date, lr.leave_end_date) AS overlaps_test_empty_interval -- Likely FALSE
FROM projects p
JOIN employee_projects ep ON p.project_id = ep.project_id
JOIN employees e ON ep.emp_id = e.emp_id
JOIN leave_requests lr ON e.emp_id = lr.emp_id
WHERE lr.status = 'Approved' AND p.project_name = 'Project Alpha' AND e.emp_name = 'Alice Wonderland' LIMIT 1; -- Example with data

-- Correct way to check if a single date is within a range (inclusive end date for leave)
SELECT
    p.project_name,
    p.start_date,
    e.emp_name,
    lr.leave_start_date,
    lr.leave_end_date,
    p.start_date >= lr.leave_start_date AND p.start_date <= lr.leave_end_date AS day_is_within_leave
FROM projects p
JOIN employee_projects ep ON p.project_id = ep.project_id
JOIN employees e ON ep.emp_id = e.emp_id
JOIN leave_requests lr ON e.emp_id = lr.emp_id
WHERE lr.status = 'Approved' AND p.project_name = 'Project Alpha' AND e.emp_name = 'Alice Wonderland' LIMIT 1;
```

Listing 4: Solution for Exercise 1.3 - Demonstrating OVERLAPS and correct check

Disadvantage Summary: The main disadvantage is potential misinterpretation of `OVERLAPS` with point-in-time events or zero-duration intervals if the database's specific handling (e.g., empty intervals) isn't understood. Direct comparisons are often clearer for single point-in-range checks.

Exercise 1.4: Time Zone Issues in Date Arithmetic without Explicit Time Zone Handling *Discussion of Disadvantage:*

1. `TIMESTAMP WITHOUT TIME ZONE`**:** If `CURRENT_TIMESTAMP` returns this type, it represents a local time. Adding an interval is straightforward mathematically but the resulting timestamp's meaning is relative to an *assumed* time zone, often the server's. If data or users are in different time zones, this can lead to off-by-several-hours errors when interpreting these timestamps.

2. `TIMESTAMP WITH TIME ZONE`**:** This type stores UTC or a timestamp with zone information. `CURRENT_TIMESTAMP` usually returns this. Arithmetic is generally con-

sistent (e.g., adding 24 hours). However, when displaying or converting this to local times, issues arise if the target local time zone isn't specified or correctly applied. An interval like `INTERVAL '1 day'` means exactly 24 hours. Adding `INTERVAL '1 day'` across a Daylight Saving Time (DST) transition might result in a local time that isn't the "same time next day" but is 23 or 25 hours later in local wall-clock time.

3. **Ambiguity of "Day":** `INTERVAL '1 day'` typically means 24 hours. If you need "next calendar day at the same local time", especially across DST, more complex logic involving `DATE_TRUNC`, local time zone conversions, and then adding intervals might be needed.

The disadvantage is that not being mindful of time zones when performing date/timestamp arithmetic can lead to data that is misinterpreted or incorrectly processed.

```sql
-- Example (behavior depends on session time zone for display)
SELECT
    TIMESTAMP WITH TIME ZONE '2024-03-10 01:00:00 America/New_York' AS ts_before_dst,
    TIMESTAMP WITH TIME ZONE '2024-03-10 01:00:00 America/New_York' + INTERVAL '24 hours' AS ts_plus_24h;
    -- In many parts of US, March 10 2024 is a DST spring forward. 2 AM becomes 3 AM.
    -- 1:00 AM EST + 24 hours = 2:00 AM EDT the next day (which is 24 actual hours later).

-- For TIMESTAMP WITHOUT TIME ZONE (assuming server time is local and not UTC)
SELECT
    '2024-01-15 10:00:00'::TIMESTAMP WITHOUT TIME ZONE AS current_ts_no_tz,
    ('2024-01-15 10:00:00'::TIMESTAMP WITHOUT TIME ZONE) + INTERVAL '1 day' AS next_day_no_tz;
    -- Result: 2024-01-16 10:00:00. Meaning is relative to unspecified time zone.
```

Listing 5: Solution for Exercise 1.4 - Conceptual Examples

(iii) Practice entirely cases where people in general does not use these approaches losing their advantages, relations and values because of the easier, basic, common or easily understandable but highly inefficient solutions

Exercise 1.5: Inefficiently Finding Projects Active During a Specific Period

```sql
SELECT project_name, start_date, COALESCE(actual_end_date, planned_end_date) AS relevant_end_date
FROM projects
WHERE
  (start_date BETWEEN '2023-01-01' AND '2023-03-31') OR
  (COALESCE(actual_end_date, planned_end_date) BETWEEN '2023-01-01' AND '2023-03-31') OR
  (start_date < '2023-01-01' AND COALESCE(actual_end_date, planned_end_date) > '2023-03-31');
```

Listing 6: Solution for Exercise 1.5 - Inefficient Method

```sql
SELECT project_name, start_date, COALESCE(actual_end_date,
    planned_end_date) AS relevant_end_date
FROM projects
WHERE (start_date, COALESCE(actual_end_date, planned_end_date, DATE '
    9999-12-31'))
      OVERLAPS (DATE '2023-01-01', DATE '2023-03-31');
-- Note: planned_end_date is NOT NULL in projects table.
-- If actual_end_date is NULL, planned_end_date is used.
-- If actual_end_date is present, it's used.
-- (start_date, COALESCE(actual_end_date, planned_end_date)) OVERLAPS (
    DATE '2023-01-01', DATE '2023-03-31');
-- The above is also correct given planned_end_date is NOT NULL.
-- The DATE '9999-12-31' handles cases where both could be NULL if
    schema allowed.
```

Listing 7: Solution for Exercise 1.5 - Efficient Method (using OVERLAPS)

Comparison: The `OVERLAPS` version is more concise, directly expresses the intent, and is less prone to logical errors.

2 Cases (Complementary SQL) - Solutions

Concepts: Searched CASE expressions, CASE in ORDER BY, CASE in GROUP BY.

(i) Meaning, values, relations, advantages of unique usage

Exercise 2.1: Project Status Categorization

```sql
SELECT
    project_name,
    start_date,
    planned_end_date,
    actual_end_date,
    CASE
        WHEN start_date > DATE '2024-01-15' THEN 'Upcoming'
        WHEN start_date <= DATE '2024-01-15' AND (actual_end_date IS NULL OR actual_end_date > DATE '2024-01-15') THEN 'Ongoing'
        WHEN actual_end_date IS NOT NULL AND actual_end_date <= planned_end_date THEN 'Completed Early/On-Time'
        WHEN actual_end_date IS NOT NULL AND actual_end_date > planned_end_date THEN 'Completed Late'
        ELSE 'Status Unknown' -- e.g. completed project but actual_end_date is null (bad data)
    END AS project_status
FROM
    projects;
```

Listing 8: Solution for Exercise 2.1

Exercise 2.2: Sorting Employees by Custom Priority

```sql
SELECT
    emp_name,
    salary,
    hire_date,
    CASE
        WHEN emp_id IN (SELECT DISTINCT manager_id FROM employees WHERE manager_id IS NOT NULL) OR manager_id IS NULL THEN 'Manager/Top Level'
        ELSE 'Non-Manager'
    END AS employee_category
FROM
    employees
ORDER BY
    CASE
        WHEN emp_id IN (SELECT DISTINCT manager_id FROM employees WHERE manager_id IS NOT NULL) OR manager_id IS NULL THEN 1 -- Managers first
        ELSE 2 -- Non-managers second
    END ASC,
    CASE
        WHEN emp_id IN (SELECT DISTINCT manager_id FROM employees WHERE manager_id IS NOT NULL) OR manager_id IS NULL THEN salary
        ELSE NULL
    END DESC NULLS LAST,
    CASE
        WHEN NOT (emp_id IN (SELECT DISTINCT manager_id FROM employees WHERE manager_id IS NOT NULL) OR manager_id IS NULL) THEN hire_date
        ELSE NULL
    END ASC NULLS LAST;
```

Listing 9: Solution for Exercise 2.2

Exercise 2.3: Grouping Projects by Budget Ranges

```sql
SELECT
    CASE
        WHEN budget IS NULL THEN 'Undefined'
        WHEN budget <= 50000 THEN 'Low'
        WHEN budget > 50000 AND budget <= 150000 THEN 'Medium'
        WHEN budget > 150000 THEN 'High'
        ELSE 'Other' -- Should not happen with these conditions
    END AS budget_category,
    COUNT(project_id) AS number_of_projects,
    SUM(budget) AS total_budget_in_category
FROM
    projects
GROUP BY
    budget_category -- Can group by alias in PostgreSQL
ORDER BY
    MIN(budget) NULLS FIRST; -- Order categories by their typical budget size
```

Listing 10: Solution for Exercise 2.3

(ii) Disadvantages of all its technical concepts

Exercise 2.4: Overly Nested CASE Expressions for Readability

```sql
SELECT
    emp_name,
    salary,
    dept_id,
    performance_rating,
    CASE
        WHEN salary > 100000 THEN
            'High Earner' ||
            CASE
                WHEN dept_id = 2 THEN
                    ', Key Engineer' ||
                    CASE WHEN performance_rating = 5 THEN ', Top Performer' ELSE '' END
                ELSE ''
            END ||
            CASE -- Example: add something if not Eng but still high earner and rating 5
                WHEN dept_id != 2 AND performance_rating = 5 THEN ', Non-Eng Top Performer'
                ELSE ''
            END
        ELSE 'Standard Profile'
    END AS profile_string
FROM
    employees;
```

Listing 11: Solution for Exercise 2.4

Discussion of Disadvantage: Deeply nested `CASE` statements or very long flat ones can significantly reduce query readability and maintainability. Debugging becomes harder. Alternatives might involve CTEs or user-defined functions for very complex, reused logic.

Exercise 2.5: CASE in GROUP BY Causing Performance Issues with Non-SARGable Conditions

```sql
SELECT
    CASE
        WHEN email LIKE '%@example.com' THEN 'Internal Email'
        ELSE 'External Email'
    END AS email_category,
    COUNT(*) AS number_of_employees
FROM
    employees
GROUP BY
    email_category; -- Can group by alias in PostgreSQL
```

Listing 12: Solution for Exercise 2.5

Discussion of Disadvantage: If `GROUP BY CASE WHEN email LIKE '%@example.com' ... END` is used:

1. **Index Inusability:** A standard B-tree index on `email` is unlikely to be used efficiently for a `LIKE '%suffix'` pattern. The database would likely scan all emails.

2. **Computational Cost:** Evaluating the `CASE` for every row before grouping adds overhead.

For better performance, functional indexes or specific text search index types might be needed. The disadvantage is potential poor performance on large tables without such optimizations.

(iii) Practice entirely cases where people in general does not use these approaches losing their advantages, relations and values because of the easier, basic, common or easily understandable but highly inefficient solutions

Exercise 2.6: Multiple UNION ALL Queries vs. CASE in GROUP BY for Segmented Counts

```sql
SELECT 'Engineering' AS department_segment, COUNT(*) AS employee_count
   FROM employees WHERE dept_id = 2
UNION ALL
SELECT 'Sales' AS department_segment, COUNT(*) AS employee_count FROM
   employees WHERE dept_id = 3
UNION ALL
SELECT 'Other Departments' AS department_segment, COUNT(*) AS
   employee_count FROM employees WHERE dept_id NOT IN (2, 3) OR dept_id
    IS NULL;
```

Listing 13: Solution for Exercise 2.6 - Inefficient Method (UNION ALL)

```sql
SELECT
    CASE
        WHEN dept_id = 2 THEN 'Engineering'
        WHEN dept_id = 3 THEN 'Sales'
        ELSE 'Other Departments'
    END AS department_segment,
    COUNT(*) AS employee_count
FROM
    employees
GROUP BY
    department_segment; -- Can group by alias in PostgreSQL
```

Listing 14: Solution for Exercise 2.6 - Efficient Method (CASE in GROUP BY)

Comparison: The `CASE in GROUP BY` approach requires only one pass over the table, generally more efficient and concise than multiple `UNION ALL` operations.

3 Null Space (Complementary SQL) - Solutions

Concepts: *`NULLIF`, `NULL handling in aggregations`, `NULL handling in sorting`.*

(i) Meaning, values, relations, advantages of unique usage

Exercise 3.1: Safe Bonus Calculation Using NULLIF

```sql
SELECT
    emp_name,
    salary,
    performance_rating,
    (salary * 0.10) / NULLIF(performance_rating, 1) AS
   bonus_avoiding_rating_1
FROM
    employees
WHERE
    salary IS NOT NULL AND performance_rating IS NOT NULL;
```

Listing 15: Solution for Exercise 3.1

Exercise 3.2: Average Billing Rate Excluding Internal/Non-Billable Roles

```sql
SELECT
    AVG(billing_rate) AS average_actual_billing_rate,
    COUNT(emp_project_id) AS total_assignments,
    COUNT(billing_rate) AS assignments_with_billing_rate
FROM
    employee_projects;
```

Listing 16: Solution for Exercise 3.2

Discussion: `AVG(billing_rate)` calculates the sum of all non-NULL `billing_rate` values divided by the count of non-NULL `billing_rate` values. `NULL` billing rates are completely ignored. This is advantageous because:

1. It prevents `NULL`s from skewing the average (e.g., if `NULL` was treated as 0).

2. It reflects the average of roles/assignments that *are* billable.

If all `billing_rate`s were `NULL`, `AVG()` would return `NULL`.

Exercise 3.3: Listing Projects by Actual End Date, Undefined Last

```sql
SELECT
    project_name,
    actual_end_date
FROM
    projects
ORDER BY
    actual_end_date ASC NULLS LAST;
```

Listing 17: Solution for Exercise 3.3

(ii) Disadvantages of all its technical concepts

Exercise 3.4: NULLIF with Unintended Type Coercion or Comparison Issues

Discussion of Disadvantage & Example: If `performance_rating` is `VARCHAR` and we use `NULLIF(performance_rating, 0)`:

1. **Type Coercion Error:** Many databases will try to convert either the string `performance_rating` to an integer, or the integer `0` to a string. If `performance_rating` contains non-numeric strings (e.g., ’Good’, ’N/A’), converting it to an integer will fail.

2. **Unexpected Behavior (String Comparison):** If the integer `0` is converted to string ’0’, then `NULLIF(’N/A’, ’0’)` would not nullify ’N/A’.

3. **Portability:** Implicit type conversion rules can vary.

It’s best practice to ensure `expr1` and `expr2` in `NULLIF` are of compatible types, or to use explicit `CAST`.

```sql
-- Hypothetical table and query illustrating the issue
CREATE TABLE IF NOT EXISTS reviews_temp (id INT, rating_text VARCHAR(10));
INSERT INTO reviews_temp VALUES (1, '5'), (2, 'N/A'), (3, '0');

-- This query in PostgreSQL will cast 0 to TEXT '0' for comparison
SELECT id, rating_text, NULLIF(rating_text, 0::TEXT) FROM reviews_temp;
-- Output will NULLIF '0', but not 'N/A'

-- If we wanted to NULLIF 'N/A':
SELECT id, rating_text, NULLIF(rating_text, 'N/A') FROM reviews_temp;

DROP TABLE IF EXISTS reviews_temp; -- Cleanup
```

Listing 18: Solution for Exercise 3.4 - Illustrative Example

The disadvantage is that carelessness with types in `NULLIF` can lead to runtime errors or logically incorrect nullifications.

Exercise 3.5: Aggregates over Mostly NULL Data Yielding Misleading Results

```sql
-- For HR dept (Frankenstein Monster: rating 3, Grace Hopper: rating NULL)
SELECT
    d.dept_name,
    AVG(e.performance_rating) AS avg_rating_ignoring_nulls,
    COUNT(e.emp_id) AS total_employees,
    COUNT(e.performance_rating) AS rated_employees
FROM
    departments d
JOIN
    employees e ON d.dept_id = e.dept_id
WHERE
    d.dept_name = 'Human Resources'
GROUP BY
    d.dept_name;
```

Listing 19: Solution for Exercise 3.5

Discussion: The ”disadvantage” is potential for misinterpretation. For ’Human Resources’, `avg_rating_ignoring_nulls` will be 3 (from Frankenstein Monster), while Grace Hopper’s `NULL` rating is ignored. Reporting ”Average rating of 3” alone can be misleading if the context (1 out of 2 employees rated) isn’t provided. This calls for careful reporting, possibly including counts of total vs. contributing populations.

(iii) Practice entirely cases where people in general does not use these approaches losing their advantages, relations and values because of the easier, basic, common or easily understandable but highly inefficient solutions

Exercise 3.6: Using `CASE WHEN expr = val THEN NULL ELSE expr END` instead of `NULLIF(expr, val)`

```sql
SELECT
    leave_id,
    reason AS original_reason,
    CASE
        WHEN reason = '' THEN NULL
        ELSE reason
    END AS reason_via_case
FROM
    leave_requests
WHERE emp_id = (SELECT emp_id from employees WHERE email='kevin@example
    .com');
```

Listing 20: Solution for Exercise 3.6 - Verbose CASE method

```sql
SELECT
    leave_id,
    reason AS original_reason,
    NULLIF(reason, '') AS reason_via_nullif
FROM
    leave_requests
WHERE emp_id = (SELECT emp_id from employees WHERE email='kevin@example
    .com');
```

Listing 21: Solution for Exercise 3.6 - Concise NULLIF method

Comparison: `NULLIF(reason, '')` is more direct and less verbose than the `CASE` expression for this specific task.

4 Hardcore Problem - Solution

Exercise 4.1: Comprehensive Departmental Project Health and Employee Engagement Report

```sql
WITH ReportContext AS (
    SELECT DATE '2024-01-15' AS current_report_date
),
ActiveEmployees AS (
    SELECT
        e.emp_id, e.emp_name, e.dept_id, e.hire_date, e.performance_rating, e.manager_id,
        (rc.current_report_date - e.hire_date) / 365.25 AS tenure_years_raw
    FROM employees e, ReportContext rc
    WHERE (e.termination_date IS NULL OR e.termination_date > rc.current_report_date)
),
ActiveManagersByDept AS (
    SELECT DISTINCT ae.dept_id, ae.emp_id AS manager_emp_id
    FROM ActiveEmployees ae
    WHERE ae.emp_id IN (SELECT DISTINCT manager_id FROM employees WHERE manager_id IS NOT NULL)
       OR ae.manager_id IS NULL
),
ActiveProjects AS (
    SELECT p.project_id, p.project_name, p.start_date, p.planned_end_date, p.lead_emp_id
    FROM projects p, ReportContext rc
    WHERE (p.actual_end_date IS NULL OR p.actual_end_date > rc.current_report_date)
),
DepartmentProjectAnalysis AS (
    SELECT
        d.dept_id,
        EXISTS (
            SELECT 1
            FROM ActiveProjects ap1
            JOIN ActiveManagersByDept amd1 ON ap1.lead_emp_id = amd1.manager_emp_id AND amd1.dept_id = d.dept_id
            JOIN ActiveProjects ap2 ON ap1.project_id < ap2.project_id
            JOIN ActiveManagersByDept amd2 ON ap2.lead_emp_id = amd2.manager_emp_id AND amd2.dept_id = d.dept_id
            WHERE (ap1.start_date, ap1.planned_end_date) OVERLAPS (ap2.start_date, ap2.planned_end_date)
        ) AS has_overlap_risk,
        COUNT(DISTINCT ap.project_id) FILTER (
            WHERE ap.planned_end_date BETWEEN rc.current_report_date AND (rc.current_report_date + INTERVAL '30 days')
              AND EXISTS (SELECT 1 FROM ActiveManagersByDept amd WHERE ap.lead_emp_id = amd.manager_emp_id AND amd.dept_id = d.dept_id)
        ) AS critical_deadline_projects_count
    FROM departments d
    LEFT JOIN ActiveProjects ap ON TRUE -- To allow access to rc and aggregate over d
    CROSS JOIN ReportContext rc -- Ensure rc.current_report_date is available
    GROUP BY d.dept_id, rc.current_report_date -- Group by rc.current_report_date for correlated subquery access
),
DepartmentAggregates AS (
    SELECT
        d.dept_id,
        d.dept_name,
        COUNT(DISTINCT ae.emp_id) AS total_active_employees,
        AVG(ae.tenure_years_raw) AS avg_employee_tenure_years_raw,
        AVG(COALESCE(ae.performance_rating, 2)) AS avg_rating_adjusted_raw,
        COUNT(DISTINCT CASE
                            WHEN lr.status = 'Approved' AND rc.current_report_date BETWEEN lr.leave_start_date AND lr.leave_end_date
                            THEN ae.emp_id
                            ELSE NULL
                        END
        ) AS distinct_employees_on_leave
    FROM departments d
    LEFT JOIN ActiveEmployees ae ON d.dept_id = ae.dept_id
    LEFT JOIN leave_requests lr ON ae.emp_id = lr.emp_id
    CROSS JOIN ReportContext rc
    GROUP BY d.dept_id, d.dept_name
)
SELECT
    da.dept_name,
    da.total_active_employees,
    ROUND(da.avg_employee_tenure_years_raw::numeric, 2) AS avg_employee_tenure_years,
    CASE
        WHEN dpa.has_overlap_risk THEN 'High Overlap Risk'
        WHEN dpa.critical_deadline_projects_count > 1 THEN 'Multiple Critical Deadlines'
        ELSE 'Normal Load'
    END AS projects_info,
    ROUND(COALESCE(da.avg_rating_adjusted_raw, 0)::numeric, 2) AS avg_rating_adjusted,
    ROUND(
        (da.distinct_employees_on_leave * 100.0) / NULLIF(da.total_active_employees, 0),
    2) AS employees_on_leave_percentage
FROM DepartmentAggregates da
JOIN DepartmentProjectAnalysis dpa ON da.dept_id = dpa.dept_id
ORDER BY
    CASE
        WHEN dpa.has_overlap_risk THEN 1
        WHEN dpa.critical_deadline_projects_count > 1 THEN 2
        ELSE 3
    END ASC,
    da.dept_name ASC
LIMIT 5;
```

Listing 22: Solution for Hardcore Exercise 4.1