SQL Challenge: Optimal Hiring Under Budget

Problem Statement

You are given a table containing `id`, `position`, and `salary` columns. You have a fixed budget of **50,000**, and your goal is to **hire as many people as possible**:

- Prioritize **"senior"** positions first.
- Then hire **"junior"** positions with the remaining budget.

Write a SQL query that returns the number of juniors and seniors hired without exceeding the total salary budget.

Table Schema

```
CREATE TABLE Employees (
  id INT,
  position VARCHAR(10),
  salary INT
);
```

Sample Inputs & Expected Outputs

Table 1

id	position	salary
1	junior	10000
2	senior	15000
3	senior	30000

Expected Output:

juniors	seniors
0	2

Table 2

id	position	salary
1	junior	5000
2	junior	7000
3	junior	7000
4	senior	10000
5	junior	10000
6	senior	20000
7	senior	30000

Expected Output:

juniors	seniors
3	2

 Table 3

id	position	salary
1	junior	15000
2	junior	15000
3	junior	20000
4	senior	60000

Expected Output:

juniors	seniors
3	0

 SQL Solution (for MySQL or similar dialects):

```
WITH ranked_seniors AS (
    SELECT *,
           ROW_NUMBER() OVER (ORDER BY salary) AS rn
    FROM Employees
    WHERE position = 'senior'
),
seniors_accum AS (
    SELECT *,
           SUM(salary) OVER (ORDER BY salary) AS total_senior_cost
    FROM ranked_seniors
    WHERE salary <= 50000
),
max_seniors AS (
    SELECT COUNT(*) AS senior_count,
           MAX(total_senior_cost) AS used_budget
    FROM seniors_accum
    WHERE total_senior_cost <= 50000
),
ranked_juniors AS (
    SELECT *,
           ROW_NUMBER() OVER (ORDER BY salary) AS rn
    FROM Employees
    WHERE position = 'junior'
),
juniors_accum AS (
    SELECT *,
           SUM(salary) OVER (ORDER BY salary) AS total_junior_cost
```

```
    FROM ranked_juniors
),
final_juniors AS (
    SELECT COUNT(*) AS junior_count
    FROM juniors_accum, max_seniors
    WHERE total_junior_cost <= 50000 - COALESCE(used_budget, 0)
)
SELECT
    COALESCE(junior_count, 0) AS juniors,
    COALESCE(senior_count, 0) AS seniors
FROM final_juniors
CROSS JOIN max_seniors;
```

Understanding the Problem

We need to:

1. Hire as many people as possible with a fixed budget of $50,000
2. Prioritize hiring seniors first
3. Use any remaining budget to hire juniors
4. Return the count of juniors and seniors hired

Breaking Down the SQL Query

This solution uses Common Table Expressions (CTEs) to build the logic in steps. Let's walk through each part:

```
WITH ranked_seniors AS (
    SELECT *,
           ROW_NUMBER() OVER (ORDER BY salary) AS rn
    FROM Employees
    WHERE position = 'senior'
),
seniors_accum AS (
    SELECT *,
           SUM(salary) OVER (ORDER BY salary) AS total_senior_cost
    FROM ranked_seniors
    WHERE salary <= 50000
),
max_seniors AS (
    SELECT COUNT(*) AS senior_count,
           MAX(total_senior_cost) AS used_budget
    FROM seniors_accum
    WHERE total_senior_cost <= 50000
),
ranked_juniors AS (
    SELECT *,
           ROW_NUMBER() OVER (ORDER BY salary) AS rn
    FROM Employees
    WHERE position = 'junior'
),
juniors_accum AS (
    SELECT *,
```

```
            SUM(salary) OVER (ORDER BY salary) AS total_junior_cost
    FROM ranked_juniors
),
final_juniors AS (
    SELECT COUNT(*) AS junior_count
    FROM juniors_accum, max_seniors
    WHERE total_junior_cost <= 50000 - COALESCE(used_budget, 0)
)
SELECT
    COALESCE(junior_count, 0) AS juniors,
    COALESCE(senior_count, 0) AS seniors
FROM final_juniors
CROSS JOIN max_seniors;
```

1. First CTE: Ranking Senior Employees by Salary

```
ranked_seniors AS (
    SELECT *,
           ROW_NUMBER() OVER (ORDER BY salary) AS rn
    FROM Employees
    WHERE position = 'senior'
)
```

This first step:

- Filters the table to include only senior positions
- Assigns a row number to each senior based on salary (lowest to highest)
- This helps us prioritize hiring cheaper seniors first to maximize our headcount

2. Second CTE: Calculating Cumulative Cost for Seniors

```
seniors_accum AS (
    SELECT *,
           SUM(salary) OVER (ORDER BY salary) AS total_senior_cost
    FROM ranked_seniors
    WHERE salary <= 50000
)
```

This step:

- Takes the ranked seniors from the previous CTE
- Adds a running total of salaries using the window function `SUM() OVER (ORDER BY salary)`
- Filters out any individual senior whose salary exceeds our total budget
- The result shows how the budget gets consumed as we hire seniors in order of increasing salary

3. Third CTE: Finding Maximum Seniors We Can Hire

```
max_seniors AS (
    SELECT COUNT(*) AS senior_count,
           MAX(total_senior_cost) AS used_budget
    FROM seniors_accum
```

```
    WHERE total_senior_cost <= 50000
)
```

Here we:

- Count how many seniors we can hire without exceeding the budget
- Calculate the total budget used for seniors
- The `WHERE total_senior_cost <= 50000` ensures we stay within budget

4. Fourth CTE: Ranking Junior Employees by Salary

```
ranked_juniors AS (
    SELECT *,
           ROW_NUMBER() OVER (ORDER BY salary) AS rn
    FROM Employees
    WHERE position = 'junior'
)
```

Similar to the first step:

- Filters to only junior positions
- Assigns a row number based on salary (lowest to highest)
- This helps us prioritize hiring cheaper juniors first with our remaining budget

5. Fifth CTE: Calculating Cumulative Cost for Juniors

```
juniors_accum AS (
    SELECT *,
           SUM(salary) OVER (ORDER BY salary) AS total_junior_cost
    FROM ranked_juniors
)
```

This step:

- Takes the ranked juniors from the previous CTE
- Adds a running total of junior salaries
- The result shows how the remaining budget gets consumed as we hire juniors

6. Sixth CTE: Determining How Many Juniors We Can Hire

```
final_juniors AS (
    SELECT COUNT(*) AS junior_count
    FROM juniors_accum, max_seniors
    WHERE total_junior_cost <= 50000 - COALESCE(used_budget, 0)
)
```

This step:

- Calculates how many juniors we can hire with the remaining budget
- `50000 - COALESCE(used_budget, 0)` determines how much money we have left after hiring seniors
- `COALESCE(used_budget, 0)` handles the case where we don't hire any seniors
- We count juniors whose cumulative cost fits within our remaining budget

7. Final Query: Combining the Results

```sql
SELECT
    COALESCE(junior_count, 0) AS juniors,
    COALESCE(senior_count, 0) AS seniors
FROM final_juniors
CROSS JOIN max_seniors
```

Finally:

- We join the junior count and senior count results
- `COALESCE` ensures we return 0 instead of NULL if no juniors or seniors can be hired
- `CROSS JOIN` simply combines our two CTEs (since each contains only one row)

Walking Through Examples

Let's apply this to Example 2:

id	position	salary
1	junior	5000
2	junior	7000
3	junior	7000
4	senior	10000
5	junior	10000
6	senior	20000
7	senior	30000

Step 1-3: Process Senior Employees

First, we filter and rank seniors by salary:

id	position	salary	rn
4	senior	10000	1
6	senior	20000	2
7	senior	30000	3

Next, we calculate cumulative costs:

id	position	salary	rn	total_senior_cost
4	senior	10000	1	10000
6	senior	20000	2	30000
7	senior	30000	3	60000

The third senior would bring us to 60000, which exceeds our budget. So we can only hire 2 seniors:

senior_count = 2 used_budget = 30000

Step 4-6: Process Junior Employees

Filter and rank juniors:

id	position	salary	rn
1	junior	5000	1
2	junior	7000	2
3	junior	7000	3
5	junior	10000	4

Calculate cumulative costs:

id	position	salary	rn	total_junior_cost
1	junior	5000	1	5000
2	junior	7000	2	12000
3	junior	7000	3	19000
5	junior	10000	4	29000

Our remaining budget is 50000 - 30000 = 20000. We can hire 3 juniors because:

- First 3 juniors cost 19000 (under our 20000 remaining)
- Adding the 4th junior would cost 29000 (exceeds our remaining budget)

Final Result

juniors = 3 seniors = 2

This matches the expected output!

Key Concepts for Beginners

1. **Common Table Expressions (CTEs)**: Break complex problems into manageable steps
2. **Window Functions**: `ROW_NUMBER()` and `SUM() OVER()` help with ranking and cumulative calculations
3. **Prioritization Logic**: Handling seniors first, then juniors
4. **COALESCE Function**: Handles NULL values by providing a default value
5. **CROSS JOIN**: Combines results from different CTEs

Alternative Approaches

For databases without window function support, you could use:

```
-- Calculate total cost of hiring all seniors whose salary is <= 50000
SELECT @senior_cost := IFNULL(SUM(salary), 0)
FROM Employees
WHERE position = 'senior' AND salary <= 50000;
```

```sql
-- Get count of seniors hired
SELECT @senior_count := COUNT(*)
FROM Employees
WHERE position = 'senior' AND salary <= 50000;

-- Calculate remaining budget after hiring seniors
SELECT @remaining := 50000 - @senior_cost;

-- Create temp table of juniors ordered by salary
CREATE TEMPORARY TABLE juniors_ordered
SELECT id, salary,
       (@running_total := @running_total + salary) AS running_total
FROM Employees, (SELECT @running_total := 0) r
WHERE position = 'junior'
ORDER BY salary;

-- Get count of juniors we can hire with remaining budget
SELECT @junior_count := COUNT(*)
FROM juniors_ordered
WHERE running_total <= @remaining;

-- Final result
SELECT @junior_count AS juniors, @senior_count AS seniors;
```

However, the window function approach is more elegant and generally easier to understand once you're familiar with it.

This problem demonstrates important concepts in budget allocation and prioritization, which are common in real-world business scenarios.