01 External Queries with EXTERNAL_QUERY

The EXTERNAL_QUERY function in Google BigQuery allows you to run a query on an external data source, such as a database connected through BigQuery's federated query feature. Here, I'll demonstrate how to use EXTERNAL_QUERY to pull data from an external database and create a new table in BigQuery.

Example: Using EXTERNAL_QUERY to Create a Table in BigQuery

Scenario Suppose you have a connected external database, such as a MySQL or PostgreSQL database, and you want to pull data from it into a BigQuery table.

- **External Source (e.g., MySQL Database)**:
  - Table: `external_employees`
  - Columns: `id`, `name`, `role`

Requirement Create a new table `imported_employees` in BigQuery, which imports data from the `external_employees` table in the external database.

SQL Command

```
CREATE OR REPLACE TABLE imported_employees AS
EXTERNAL_QUERY("connection_id",
               "SELECT id, name, role FROM external_employees;");
```

- `connection_id` is the identifier for the connection to the external database.
- The SELECT statement is used to specify the data to be imported from the external database.

Resulting Action

- BigQuery runs the specified query on the external database using the provided connection.
- The result of the query (data from `external_employees`) is used to create or replace the `imported_employees` table in BigQuery.

Post-Execution State of `imported_employees` Table

- The `imported_employees` table in BigQuery will contain the data imported from the external database's `external_employees` table.
- Columns will include `id`, `name`, and `role`, as selected in the EXTERNAL_QUERY.

Note

- Ensure that the connection to the external data source is correctly set up in BigQuery.
- Permissions must be appropriately configured to allow BigQuery to access the external data source.
- `EXTERNAL_QUERY` is specific to Google BigQuery and is used for federated querying, allowing BigQuery to directly query external databases without the need to import data.

This feature is particularly useful for scenarios where you want to integrate or migrate data from various external databases into BigQuery for advanced analytics and data warehousing purposes.

02 SQL JOIN: Combining Data from Multiple Tables

Example: SQL JOIN Operation

- **Source Tables**:
  1. **`employees`**: | employee_id | name | department_id | |————-|————|————| | 1 | Alice | 101 | | 2 | Bob | 102 | | 3 | Charlie | 101 |
  2. **`departments`**: | department_id | department_name | |————|————-| | 101 | Sales | | 102 | Marketing | | 103 | HR |
- **SQL JOIN Query**:

```sql
SELECT
    e.employee_id,
    e.name AS EmployeeName,
    d.department_name AS Department
FROM
    employees e
JOIN
    departments d ON e.department_id = d.department_id;
```

- **Result**: | employee_id | EmployeeName | Department | |————-|————-|————| | 1 | Alice | Sales | | 2 | Bob | Marketing | | 3 | Charlie | Sales |

In this example, the JOIN operation is used to combine rows from two tables: `employees` and `departments`. The JOIN is based on a common column between these two tables, which is `department_id`.

- The `employees` table contains employee details along with their associated department IDs.

- The `departments` table lists department IDs and their names.

The SQL query joins these tables on their `department_id` columns. The result is a new table that includes the employee ID, employee name, and the corresponding department name for each employee. This kind of JOIN is particularly useful when you need to combine related information from multiple tables into a single, comprehensive dataset.

03 SQL JOIN: Creating Wide Tables

Expanded Source Tables:

1. **`employees`**: | employee_id | name | department_id | email | hire_date | |————-|———-|————|—————-|————| | 1 | Alice | 101 | alice@example.com | 2020-01-10 | | 2 | Bob | 102 | bob@example.com | 2019-06-23 | | 3 | Charlie | 103 | charlie@example.com| 2021-03-15 |
2. **`departments`**: | department_id | department_name | location | |————|————-|———— | | 101 | Sales | New York | | 102 | Marketing | San Francisco | | 103 | HR | Seattle |

Requirement:

Create a new table `employee_details` that includes detailed information about each employee, combining data from both `employees` and `departments`.

SQL Command:

```sql
CREATE OR REPLACE TABLE employee_details AS
SELECT
    e.employee_id,
    e.name AS employee_name,
    e.email,
    e.hire_date,
    d.department_name,
    d.location AS department_location
FROM
    employees e
JOIN
    departments d ON e.department_id = d.department_id;
```

Resulting Action:

- The command creates or replaces the `employee_details` table.
- The new table is populated with a selection of columns from both `employees` and `departments`, joined on `department_id`.
- The result includes employee ID, name, email, hire date, department name, and department location.

Post-Execution State of `employee_details` Table:

- The `employee_details` table now contains a more comprehensive set of data for each employee, reflecting information from both source tables.
- Example content of `employee_details`: | employee_id | employee_name | email | hire_date | department_name | department_location | |————-|————|————————|————|————————|————————| | 1 | Alice | alice@example.com | 2020-01-10 | Sales | New York | | 2 | Bob | bob@example.com | 2019-06-23 | Marketing | San Francisco | | 3 | Charlie | charlie@example.com| 2021-03-15 | HR | Seattle |

04 SQL JOIN: Complex example with Common Table Expressions (CTEs)

Certainly! Let's provide example data for each of the source tables (`employees`, `departments`, `roles`) and show what the resulting data would look like in the `combined_employee_data` view created by the CTE in BigQuery.

Source Tables Example Data:

1. **`employees`**: | employee_id | name | department_id | role_id | |————-|————|————|————| | 1 | Alice | 101 | 201 | | 2 | Bob | 102 | 202 | | 3 | Charlie | 103 | 203 |
2. **`departments`**: | department_id | department_name | |————|————-| | 101 | Sales | | 102 | Marketing | | 103 | HR |
3. **`roles`**: | role_id | role_name | |———|————-| | 201 | Manager | | 202 | Team Leader | | 203 | Specialist |

SQL Command

```
CREATE OR REPLACE VIEW `[YOUR
 ↪  PROJECT].adventureworks.combined_employee_data` AS
WITH joined_data AS (
  SELECT
    e.employee_id,
    e.name AS employee_name,
    d.department_name,
    r.role_name
  FROM
    `adventureworks.employees` e
  JOIN
    `adventureworks.departments` d ON e.department_id = d.department_id
  JOIN
    `adventureworks.roles` r ON e.role_id = r.role_id
)
SELECT
  *
FROM
  joined_data;
```

Resulting View (`combined_employee_data`) Data:

After executing the SQL command to create the view, the `combined_employee_data` would present the following combined data:

employee_id	employee_name	department_name	role_name
1	Alice	Sales	Manager
2	Bob	Marketing	Team Leader
3	Charlie	HR	Specialist

In this example:

- The `employees` table provides the core employee details, including their associated department and role IDs.
- The `departments` table lists each department's ID and name.
- The `roles` table details each role's ID and name.

- The CTE in the SQL query joins these tables on `department_id` and `role_id`, creating a comprehensive view (`combined_employee_data`) that includes each employee's name, department name, and role name.

This view is useful for getting a quick overview of employee roles across different departments, and it demonstrates the power of combining data from multiple tables into a single, more informative dataset.

05 SQL SELECT COUNT: Counting Rows in a Table

Source Table Example Data (`employees`):

employee_id	name	department_id
1	Alice	101
2	Bob	102
3	Charlie	101
4	Dana	103
5	Edward	101

SQL Command to Count All Employees:

```sql
SELECT COUNT(*) AS TotalEmployees
FROM employees;
-- Expected Result: 5
```

Result:

- This query counts the total number of rows (employees) in the `employees` table.
- **Expected Result**:

  - `TotalEmployees`: 5

SQL Command to Count Employees in a Specific Department:

```sql
SELECT COUNT(*) AS EmployeesInSales
FROM employees
```

```sql
WHERE department_id = 101;
-- Expected Result: 3
```

Result:

- This query counts the number of employees in the Sales department (department_id = 101).
- **Expected Result**:
  - `EmployeesInSales`: 3

The `COUNT(*)` function in SQL is very useful for getting the total number of records in a table or a subset of records that match certain criteria. In the first example, it provides the total count of employees, while in the second, it gives the count of employees working in a specific department. This function is essential in data analysis for getting quick insights into the volume of data.

06 SQL SUM and GROUP BY: Aggregating Data

These commands are typically used to aggregate data. We'll use an expanded version of the `employees` table, which includes a `salary` column, to illustrate these concepts.

Source Table Example Data (`employees`):

employee_id	name	department_id	salary
1	Alice	101	70000
2	Bob	102	80000
3	Charlie	101	55000
4	Dana	103	50000
5	Edward	101	65000

SQL Command to Sum Salaries for All Employees:

```sql
SELECT SUM(salary) AS TotalSalaries
FROM employees;
-- Expected Result: 320000
```

Result:

- This query sums the `salary` column for all employees in the `employees` table.
- **Expected Result**:

  - `TotalSalaries`: 320000

SQL Command to Sum Salaries by Department:

```
SELECT department_id, SUM(salary) AS TotalSalaries
FROM employees
GROUP BY department_id;
```

Result:

- This query sums the salaries for each department, grouping the results by `department_id`.
- **Expected Result**: | department_id | TotalSalaries | |————|————| | 101 | 190000 | | 102 | 80000 | | 103 | 50000 |

The SUM function in SQL is used to calculate the total of a numeric column. When used in conjunction with GROUP BY, it allows for the aggregation of this total based on different groupings, such as departments in this example. This is essential for data analysis and reporting where summarization of data is required.

07 SQL CAST: Converting Data Types

This function is particularly useful when you have numerical data stored as strings and you need to perform numerical operations on them. We'll use a hypothetical `orders` table for this illustration, where the `price` column is stored as a string.

Source Table Example Data (`orders`):

order_id	product_name	price
1	Widget	'15.99'
2	Gadget	'21.50'
3	Gizmo	'10.75'

SQL Command to Convert 'price' from String to Numeric and Calculate Total:

```
SELECT SUM(CAST(price AS NUMERIC)) AS TotalSales
FROM orders;
-- Expected Result: 48.24 (as NUMERIC)
```

Result:

- This query first converts the `price` column from a string to a numeric type using `CAST(price AS NUMERIC)`.
- Then, it sums up these values to provide the total sales amount.
- **Expected Result**:

  - `TotalSales`: 48.24

SQL Command to Convert 'price' and Show Individual Prices as Numeric:

```
SELECT order_id, product_name, CAST(price AS NUMERIC) AS NumericPrice
FROM orders;
```

Result:

- This query converts the `price` column for each order from a string to a numeric type.
- It displays the order ID, product name, and the converted numeric price.
- **Expected Result**: | order_id | product_name | NumericPrice | |———-|———–|———–| | 1 | Widget | 15.99 | | 2 | Gadget | 21.50 | | 3 | Gizmo | 10.75 |

The CAST function is essential for data type conversion in SQL. In this scenario, converting the `price` from a string to a numeric type allows for numerical calculations, such as summing up the total sales. This is a common requirement in data processing where the data type in the source data might not be aligned with the needs of your analysis or calculations.

08 SQL EXTRACT: Getting Specific Parts of a Date

This function is used to extract a specific part of a date, such as the year, from a date column. For this illustration, we'll use a hypothetical `employees` table that includes a `hire_date` column.

Source Table Example Data (`employees`):

employee_id	name	hire_date
1	Alice	2020-01-15
2	Bob	2019-06-23
3	Charlie	2021-03-12

SQL Command to Extract Year from Hire Dates:

```
SELECT employee_id, name, EXTRACT(YEAR FROM hire_date) AS hire_year
FROM employees;
```

Result:

- This query extracts the year part from the `hire_date` column for each employee.
- **Expected Result**: | employee_id | name | hire_year | |————-|———-|———-| | 1 | Alice | 2020 | | 2 | Bob | 2019 | | 3 | Charlie | 2021 |

SQL Command to Count Employees Hired Each Year:

```
SELECT EXTRACT(YEAR FROM hire_date) AS hire_year, COUNT(*) AS
 ↪  number_of_hires
FROM employees
GROUP BY hire_year;
```

Result:

- This query groups the employees based on the year they were hired and counts the number of hires in each year.
- **Expected Result**: | hire_year | number_of_hires | |————-|————-| | 2019 | 1 | | 2020 | 1 | | 2021 | 1 |

The `EXTRACT(YEAR FROM ...)` function in SQL is a powerful tool for working with date fields. It allows you to isolate specific components of a date, such as the year, which can be essential for time-based analysis, such as understanding trends over time or grouping data by specific time periods.

09 SQL DATE_DIFF and TIMESTAMP_DIFF: Calculating Time Differences

These functions are used to calculate the difference between two dates or timestamps. We'll create an example using a hypothetical `project_deadlines` table for `DATE_DIFF` and a `meeting_schedule` table for `TIMESTAMP_DIFF`.

Example for DATE_DIFF:

Source Table Example Data (`project_deadlines`):

project_id	start_date	end_date
101	2021-01-10	2021-03-15
102	2021-02-01	2021-04-20
103	2021-03-05	2021-03-30

SQL Command to Calculate Project Durations in Days:

```
SELECT project_id, DATE_DIFF(end_date, start_date, DAY) AS duration_days
FROM project_deadlines;
```

Result:

- This query calculates the number of days between `start_date` and `end_date` for each project.
- **Expected Result**: | project_id | duration_days | |———|———| | 101 | 63 | | 102 | 78 | | 103 | 25 |

Example for TIMESTAMP_DIFF:

Source Table Example Data (`meeting_schedule`):

meeting_id	start_timestamp	end_timestamp
201	2021-01-10T09:00:00Z	2021-01-10T11:00:00Z
202	2021-02-01T14:00:00Z	2021-02-01T15:30:00Z

meeting_id	start_timestamp	end_timestamp
203	2021-03-05T08:30:00Z	2021-03-05T10:15:00Z

SQL Command to Calculate Meeting Durations in Minutes:

```
SELECT meeting_id, TIMESTAMP_DIFF(end_timestamp, start_timestamp, MINUTE) AS
 ↪  duration_minutes
FROM meeting_schedule;
```

Result:

- This query calculates the duration in minutes between `start_timestamp` and `end_timestamp` for each meeting.
- **Expected Result**: | meeting_id | duration_minutes | |———|————| | 201 | 120 | | 202 | 90 | | 203 | 105 |

`DATE_DIFF` is used for date fields and typically returns the difference in days, while `TIMESTAMP_DIFF` is used for timestamp fields and can return the difference in various units such as seconds, minutes, or hours. Both functions are invaluable for calculating durations and differences in times or dates in SQL databases.

09 SQL LENGTH and LEFT: Working with Strings

LEFT is used to extract a specified number of characters from the left side of a string.

Source Table Example Data (`sales_data`):

sale_id	amount
1	'100.50CHF'
2	'200.75CHF'
3	'50.00CHF'

SQL Command to Extract and Convert CHF Currency Amounts:

```
SELECT sale_id,
       CAST(LEFT(amount, LENGTH(amount) - 3) AS NUMERIC) AS numeric_amount
FROM sales_data;
```

Result:

- This query uses `LEFT` and `LENGTH` to remove the "CHF" suffix from the `amount` string. `LENGTH(amount) - 3` calculates the length of the numeric part of the string, excluding the last three characters ("CHF").
- CAST is then used to convert this string to a numeric data type.
- **Expected Result**: | sale_id | numeric_amount | |———|————-| | 1 | 100.50 | | 2 | 200.75 | | 3 | 50.00 |

In this scenario, `LEFT` and `LENGTH` functions are used together to isolate and extract the numeric portion of the amount string, removing the currency symbol which appears at the end. This technique is essential for data cleaning and preparation, especially when dealing with financial data stored in mixed-format strings.

10 SQL ARRAY_AGG and UNNEST: Working with Arrays

Example for ARRAY_AGG(STRUCT...):

Source Table Example Data (`employee_skills`):

employee_id	skill	proficiency_level
1	SQL	Advanced
1	Python	Intermediate
2	Java	Beginner
2	SQL	Advanced
3	Python	Expert

SQL Command to Aggregate Skills and Proficiency per Employee:

```
SELECT employee_id, ARRAY_AGG(STRUCT(skill, proficiency_level)) AS
 ↪  skills_info
FROM employee_skills
GROUP BY employee_id;
```

Result:

- **Expected Result**: | employee_id | skills_info | |————-|————————————-| | 1 | [{“skill”: “SQL”, “proficiency_level”: “Advanced”}, {“skill”: “Python”, “proficiency_level”: “Intermediate”}] | | 2 | [{“skill”: “Java”, “proficiency_level”: “Beginner”}, {“skill”: “SQL”, “proficiency_level”: “Advanced”}] | | 3 | [{“skill”: “Python”, “proficiency_level”: “Expert”}] |

Example for UNNEST with STRUCT:

Source Table Example Data (`projects`):

project_id	team_members
101	[{“employee_id”: 1, “role”: “Developer”}, {“employee_id”: 2, “role”: “Analyst”}]
102	[{“employee_id”: 2, “role”: “Manager”}, {“employee_id”: 3, “role”: “Developer”}]

SQL Command to Unnest Team Members:

```
SELECT project_id, member.employee_id, member.role
FROM projects, UNNEST(team_members) AS member;
```

Result:

- **Expected Result**: | project_id | employee_id | role | |————|————-|————| | 101 | 1 | Developer | | 101 | 2 | Analyst | | 102 | 2 | Manager | | 102 | 3 | Developer |

In this representation, we’re using a format similar to JSON to illustrate the nested and structured data. It’s a straightforward way to visualize complex data structures like arrays of records, which are common in BigQuery and other advanced SQL environments. This format is helpful for understanding how data is grouped and expanded in queries involving `ARRAY_AGG(STRUCT...)` and `UNNEST`.

11 SQL MIN, MAX and AVG: Calculating Aggregates

These functions are used for calculating the average, minimum, and maximum of a numeric column in a set of rows.

Source Table Example Data (`employee_salaries`):

employee_id	name	salary
1	Alice	70000
2	Bob	80000
3	Charlie	55000
4	Dana	65000
5	Edward	72000

Example for AVG (Average):

SQL Command to Calculate Average Salary:

```sql
SELECT AVG(salary) AS average_salary
FROM employee_salaries;
-- Expected Result: 68400
```

Result:

- **Expected Result**: | average_salary | |————-| | 68400 |

Example for MIN (Minimum):

SQL Command to Find Minimum Salary:

```sql
SELECT MIN(salary) AS minimum_salary
FROM employee_salaries;
-- Expected Result: 55000
```

Result:

- **Expected Result**: | minimum_salary | |————-| | 55000 |

Example for MAX (Maximum):

SQL Command to Find Maximum Salary:

```
SELECT MAX(salary) AS maximum_salary
FROM employee_salaries;
-- Expected Result: 80000
```

Result:

- **Expected Result**: | maximum_salary | |————-| | 80000 |

In these examples, AVG is used to calculate the average salary across all employees, MIN finds the lowest salary, and MAX identifies the highest salary in the `employee_salaries` table. These aggregate functions are crucial for summarizing and analyzing numerical data, providing insights into trends, distributions, and extremes in datasets.

Other Scenario Using MIN, MAX, and DATE_DIFF:

Source Table Example Data (`product_sales`):

product_id	product_name	sale_start_date	sale_end_date
101	Widget	2021-01-01	2021-01-15
102	Gadget	2021-02-01	2021-03-01
103	Gizmo	2021-02-15	2021-02-20
104	Doodad	2021-03-01	2021-04-15

SQL Command to Calculate Shortest and Longest Sales Periods:

```
SELECT
  MIN(DATE_DIFF(sale_end_date, sale_start_date, DAY)) AS
   ↪  shortest_sales_period_days,
  MAX(DATE_DIFF(sale_end_date, sale_start_date, DAY)) AS
   ↪  longest_sales_period_days
```

```
FROM
  product_sales
WHERE
  sale_end_date IS NOT NULL;
```

Result:

- This query calculates the shortest and longest sales periods (in days) among all products that have a defined sale end date.
- **Expected Result**: | shortest_sales_period_days | longest_sales_period_days | |———————————-|———————————| | 5 | 75 |

In this scenario, we use the `MIN` and `MAX` functions along with `DATE_DIFF` to analyze the sales period durations. `DATE_DIFF` calculates the number of days between the start and end dates of each product's sale period. `MIN` finds the shortest sales period, and `MAX` identifies the longest sales period across all products in the `product_sales` table. This kind of analysis is useful for understanding sales dynamics, such as how long products tend to stay on sale.

12 SQL +,*,/: Basic Arithmetic Operations

For this scenario, we'll use a hypothetical `order_items` table, which contains data about various items in customer orders, including quantity, unit price, and discount.

Scenario for Arithmetic Calculations:

Source Table Example Data (`order_items`):

order_id	item_id	quantity	unit_price	discount
101	A1	10	20.00	0.10
102	B2	5	40.00	0.05
103	C3	15	15.00	0.00
104	D4	8	30.00	0.20

SQL Command to Calculate Total Price for Each Item:

```
SELECT
    order_id,
    item_id,
    quantity,
    unit_price,
    discount,
    (quantity * unit_price) * (1 - discount) AS total_price
FROM
    order_items;
```

Result:

- This query calculates the total price for each item, considering the quantity, unit price, and discount.
- The total price is calculated as `(quantity * unit_price) * (1 - discount)`.
- **Expected Result**: | order_id | item_id | quantity | unit_price | discount | total_price | |———-|———|———-|————|————-|————-| | 101 | A1 | 10 | 20.00 | 0.10 | 180.00 | | 102 | B2 | 5 | 40.00 | 0.05 | 190.00 | | 103 | C3 | 15 | 15.00 | 0.00 | 225.00 | | 104 | D4 | 8 | 30.00 | 0.20 | 192.00 |

In this example, the arithmetic operations (* for multiplication and − for subtraction) are used to calculate the total price of each item after applying the discount. These basic calculations are fundamental in SQL for data analysis, especially in financial or sales data processing, where such computations are frequently required to derive meaningful insights from raw data.

13 SQL CASE: Conditional Logic

The CASE statement is a conditional expression that provides a way to perform complex checks and return different values based on specific conditions. We'll use a hypothetical `employee_performance` table for this example.

Scenario for CASE:

Source Table Example Data (`employee_performance`):

employee_id	name	sales_amount	customer_feedback_score
1	Alice	5000	4.5
2	Bob	7000	3.8
3	Charlie	4000	4.2
4	Dana	8000	4.7
5	Edward	3000	3.5

SQL Command to Categorize Employees Based on Performance:

```
SELECT
    employee_id,
    name,
    sales_amount,
    customer_feedback_score,
    CASE
        WHEN sales_amount > 6000 AND customer_feedback_score >= 4 THEN 'High
        ↪  Performer'
        WHEN sales_amount BETWEEN 4000 AND 6000 AND customer_feedback_score
        ↪  >= 3.5 THEN 'Moderate Performer'
        ELSE 'Needs Improvement'
    END AS performance_category
FROM
    employee_performance;
```

Result:

- This query categorizes employees into 'High Performer', 'Moderate Performer', or 'Needs Improvement' based on their sales amount and customer feedback score.
- **Expected Result**: | employee_id | name | sales_amount | customer_feedback_score | performance_category | |————-|———-|————-|——————————-|——————————-| | 1 | Alice | 5000 | 4.5 | Moderate Performer | | 2 | Bob | 7000 | 3.8 | High Performer | | 3 | Charlie | 4000 | 4.2 | Moderate Performer | | 4 | Dana | 8000 | 4.7 | High Performer | | 5 | Edward | 3000 | 3.5 | Needs Improvement |

In this example, the CASE statement evaluates each employee's sales amount and customer feedback score to categorize their performance. This approach is highly useful for dynamically classifying

data based on multiple criteria and is widely used in data analysis and reporting for segmentation, categorization, and more nuanced insights into data sets.

14 SQL LAG: Comparing Values in Adjacent Rows

LAG is often used in time-series data or any scenario where you need to compare rows sequentially. For this example, we'll use a hypothetical `monthly_sales` table.

Scenario for LAG:

Source Table Example Data (`monthly_sales`):

month	sales_amount
2021-01	10000
2021-02	15000
2021-03	13000
2021-04	16000
2021-05	14000

SQL Command to Compare Current Month Sales with Previous Month:

```
SELECT
    month,
    sales_amount,
    LAG(sales_amount) OVER (ORDER BY month) AS previous_month_sales
FROM
    monthly_sales;
```

Result:

- This query uses LAG to access the sales amount of the previous month for each row.
- The `OVER (ORDER BY month)` clause specifies the order in which the rows are processed.
- **Expected Result**: | month | sales_amount | previous_month_sales | |———-|———-|———— ——-| | 2021-01 | 10000 | NULL | | 2021-02 | 15000 | 10000 | | 2021-03 | 13000 | 15000 | | 2021-04 | 16000 | 13000 | | 2021-05 | 14000 | 16000 |

In this example, the LAG function is used to retrieve the sales amount from the previous month for each record in the `monthly_sales` table. This is particularly useful for comparing sequential data, like time-series analysis, where you need to compare a value with its predecessor. The first row has a `NULL` for `previous_month_sales` since there is no preceding row for the first month.

15 SQL PERCENT_RANK: Calculating Percentile Rank

The PERCENT_RANK function in SQL is used to calculate the relative rank of a row within a partition of a result set. The value returned is the percentile rank of the row, ranging from 0 to 1. This function is particularly useful in statistical analysis and reporting. Let's demonstrate it with an example using a hypothetical `student_grades` table.

Scenario for PERCENT_RANK:

Source Table Example Data (`student_grades`):

student_id	test_score
1	85
2	92
3	75
4	88
5	90

SQL Command to Calculate Percentile Rank of Test Scores:

```
SELECT
    student_id,
    test_score,
    PERCENT_RANK() OVER (ORDER BY test_score) AS percentile_rank
FROM
    student_grades;
```

Result:

- This query calculates the percentile rank of each student's test score within the group.

- The `OVER (ORDER BY test_score)` clause determines the order of the test scores for percentile calculation.
- **Expected Result** (the exact values may vary slightly depending on the database system): | student_id | test_score | percentile_rank | |————|————|————————| | 3 | 75 | 0.00 | | 1 | 85 | 0.25 | | 4 | 88 | 0.50 | | 5 | 90 | 0.75 | | 2 | 92 | 1.00 |

In this example, `PERCENT_RANK` is used to rank students based on their test scores. The function assigns a percentile rank to each score, with 0 representing the lowest score and 1 representing the highest score. This is useful in educational data analysis for understanding how a student's performance compares relative to peers, or in any other domain where relative rankings are important.

16 BigQueryML ARIMA: Time Series Forecasting

For illustration, we'll assume a dataset that tracks monthly sales for a retail store.

Hypothetical Scenario: Monthly Sales Forecasting

Sample Data (`retail_sales_data`): Imagine we have a table with historical monthly sales data:

date	sales_amount
2020-01	30000
2020-02	34000
2020-03	32000
2020-04	36000
…	…

SQL Query to Create an ARIMA Model:

```
CREATE OR REPLACE MODEL `your_project.your_dataset.retail_sales_arima_model`
OPTIONS
  (model_type = 'ARIMA_PLUS',
   time_series_timestamp_col = 'month',
   time_series_data_col = 'monthly_sales',
   auto_arima = TRUE,
```

```
    data_frequency = 'AUTO_FREQUENCY',
    decompose_time_series = TRUE
  ) AS
SELECT
  PARSE_DATE("%Y-%m", date) AS month,
  SUM(sales_amount) AS monthly_sales
FROM
  `your_project.your_dataset.retail_sales_data`
GROUP BY month;
```

Expected Output:

- After running this SQL query, BigQuery ML will create an ARIMA model named `retail_sales_arima_model` in the specified dataset.
- The model learns from the historical monthly sales data and is capable of forecasting future sales.

Forecasting Future Sales: Once the model is created, you can use it to forecast future sales. For example:

```
SELECT
  month,
  forecast_value
FROM
  ML.FORECAST(MODEL `your_project.your_dataset.retail_sales_arima_model`,
              STRUCT(12 AS horizon, 0.8 AS confidence_level))
```

Expected Forecast Output: This query would return a forecast for the next 12 months, with expected sales amounts for each month. The output might look like this:

month	forecast_value
2021-05	37000
2021-06	38000
2021-07	39000
…	…

Notes:

- Replace `your_project.your_dataset.retail_sales_data` and `your_project.your_datase` with your actual project and dataset names.
- The forecast values are hypothetical and for illustrative purposes.
- ARIMA models in BigQuery ML are great for time series forecasting, especially when you have historical data over regular time intervals and want to predict future trends.

This scenario showcases how you can leverage SQL and BigQuery ML to build and use time series forecasting models, bringing powerful statistical modeling capabilities to analysts familiar with SQL.

17 BigQueryML Logistic Regression: Predicting Binary Outcomes

In this example, we'll imagine we're building a model to predict a binary outcome based on a dataset. For instance, let's say we want to predict whether individuals fall into a certain income bracket based on various demographic and economic factors.

Hypothetical Scenario: Predicting Income Bracket

Sample Data (`demographic_data`): Imagine we have a table with demographic and economic data:

age	education_num	marital_status	occupation	hours_per_week	income_bracket
39	13	Never-married	Adm-clerical	40	'<=50K'
50	13	Married-civ-spouse	Exec-managerial	13	'>50K'
38	9	Divorced	Handlers-cleaners	40	'<=50K'
…	…	…	…	…	…

Creating the Logistic Regression Model:

```
CREATE OR REPLACE MODEL `your_project.your_dataset.census_model`
OPTIONS
  ( model_type='LOGISTIC_REG',
    auto_class_weights=TRUE,
    input_label_cols=['income_bracket']
```

```
  ) AS
SELECT
  * EXCEPT(dataframe)
FROM
  `your_project.your_dataset.demographic_data`
WHERE
  dataframe = 'training';
```

Explanation:

- **`CREATE OR REPLACE MODEL`**: Creates a new logistic regression model or replaces an existing one.
- **`OPTIONS`**:
  - `model_type='LOGISTIC_REG'`: Specifies that the model is a logistic regression model, suitable for binary classification problems.
  - `auto_class_weights=TRUE`: Automatically adjusts class weights inversely proportional to class frequencies in the input data.
  - `input_label_cols=['income_bracket']`: Specifies the column to predict.
- **`SELECT ... FROM ... WHERE`**: Selects the training data while excluding the `dataframe` column.

Expected Output:

- This query creates a logistic regression model named `census_model` in the specified dataset.
- The model learns from the training data to predict the `income_bracket` based on features like age, education, marital status, occupation, and hours per week.

Using the Model for Prediction: After the model is created, you can use it to predict income brackets. For example:

```
SELECT
  predicted_income_bracket,
  probability,
  *
FROM
  ML.PREDICT(MODEL `your_project.your_dataset.census_model`,
             (SELECT * FROM `your_project.your_dataset.demographic_data`
↪  WHERE dataframe = 'evaluation'))
```

Expected Prediction Output: This query would use the model to predict income brackets for the evaluation dataset. The output might look like this:

predicted_income_bracket	probability	age	education_num	marital_status	occupation	hours_per_week
'<=50K'	0.75	28	10	Never-married	Sales	40
'>50K'	0.60	42	16	Married	Exec-managerial	50
…	…	…	…	…	…	…

Notes:

- Replace `your_project.your_dataset.demographic_data` and `your_project.your_dataset` with your actual project and dataset names.
- The prediction results are hypothetical and for illustrative purposes.
- Logistic regression in BigQuery ML is a powerful tool for binary classification tasks, allowing analysts to perform predictive analytics directly in the database environment.

This scenario illustrates how you can use SQL and BigQuery ML for logistic regression, a common technique in statistical modeling and machine learning for binary classification problems.