Google Cloud

Techniques used to transform data, part 1

Data transformation is the process of taking raw or inconsistent data, and converting it into a usable format for analysis and visualization. In this reading, you'll learn more about data transformation techniques for aggregation, deduplication, and filtering, and how you can perform them in SQL.

Data aggregation

Data aggregation is the process of combining data from multiple sources and summarizing it to provide general insights, or report overall statistics. Common aggregation functions in SQL are **SUM**, **COUNT**, **AVG**, **MAX**, and **MIN**.

SUM and COUNT

The **SUM** function returns the total sum of a numeric column, like sales or expenditure for the month, year, or quarter.

This query uses the **SUM** function to calculate the sum of the values in the sales_amount column. The **SELECT** statement returns the total sales amount, which is aliased as total_sales.

```
Unset
SELECT SUM(sales_amount) AS total_sales_amount
FROM total_sales;
```

If the total_sales table contains these rows of data:

```
Unset
| sales_amount |
|--------------|
|      100000  |
|      200000  |
|      300000  |
|      400000  |
```

The query will return this result:

```
Unset
| total_sales_amount |
|--------------------|
|            1000000 |
```

The **COUNT** function returns the number of rows that match a specific criteria. These rows can contain numeric and non-numeric values.

If you want to count the number of customers you've served in the last quarter to make a projection for the next quarter. The customer_id column contains strings and numeric values. The **SELECT** statement counts the rows in the customer_id column from the customers table, and the result will be aliased as number_of_customers.

```
Unset
SELECT COUNT(customer_id) AS number_of_customers
FROM customers;
```

If the customers table contains these rows of data:

```
Unset
| customer_id |
|-------------|
|   CUST001   |
|   CUST002   |
|   CUST003   |
|   CUST004   |
|   CUST005   |
```

The query will return this result:

```
Unset
| number_of_customers |
|---------------------|
|          5          |
```

SUM and **COUNT** perform a "count" in different ways. The **SUM** statement adds numerical values and returns a numerical value. The **COUNT** statement returns a numeric value, too, but

it's the total number of rows counted, not a total sum of the values in their cells.

SUM and **COUNT** statements can be used together. Consider a scenario where you want to know how many orders customers made, and the total revenue from those orders. The **SELECT** statement counts the rows in the order_id column aliased as number_of_orders, and sums the rows in the total_price column aliased as total_revenue.

Unset

```
SELECT
    COUNT(order_id) AS number_of_orders,
    SUM(total_price) AS total_revenue
FROM orders;
```

If the table orders contains these rows of data:

Unset

```
| order_id | customer_id | total_price |
|----------|-------------|-------------|
|    1     |   CUST001   |     100     |
|    2     |   CUST002   |     150     |
|    3     |   CUST003   |     200     |
|    4     |   CUST001   |     50      |
|    5     |   CUST004   |     250     |
```

The query will return this result:

Unset

```
| number_of_orders | total_revenue |
|------------------|---------------|
|        5         |      750      |
```

AVG

The **AVG** function returns the mean or average value of a numeric column. The **AVG** function makes comparisons, identifies trends and patterns, and sets benchmarks.

An analyst might use the **AVG** function to calculate a student satisfaction score for different class sessions. The **SELECT** statement calculates the average of the student_rating column from the students table, and aliases the result as student_satisfaction.

```
Unset
SELECT AVG(student_rating) AS student_satisfaction
FROM students;
```

If the table contains these rows of data:

```
Unset
| student_id | student_rating |
|------------|----------------|
|  ANSM1995  |       5        |
|  JOLE1998  |       4        |
|  SARO2000  |       4        |
|  MATO1997  |       3        |
|  KIST2002  |       5        |
```

The query will return this result:

```
Unset
| student_satisfaction |
|----------------------|
|         4.2          |
```

MAX & MIN

The **MAX** and **MIN** functions in SQL return the largest and smallest values in a column. They can be used on numeric and non-numeric values. For numeric values, the **MAX** and **MIN** function will return the largest and smallest numeric values in the column, whether they are integers, floats, or dates. The **SELECT** statement calculates the maximum and minimum grade received by students to determine the range of scores, aliased as highest_score and lowest_score.

```
Unset
SELECT MAX(grade) as highest_score, MIN(grade) as lowest_score
FROM students;
```

If the table contains these rows of data:

```
Unset
| student_id | grade |
|------------|-------|
|  ANSM1995  |  85   |
|  JOLE1998  |  90   |
|  SARO2000  |  78   |
|  MATO1997  |  88   |
|  KIST2002  |  82   |
```

The query will return this result:

```
Unset
| highest_score | lowest_score |
|---------------|--------------|
|      90       |      78      |
```

Note: **MAX** and **MIN** can be used to find the first or last string values using alphabetical order.

Data deduplication

The purpose of data deduplication is to identify and remove duplicates from a dataset. Duplicates in a dataset negatively impact analysis by skewing results, making outcomes or insights inaccurate. Analysts use a combination of the **WHERE**, **DISTINCT**, and **GROUP BY** functions in SQL for deduplication:

WHERE

The **WHERE** clause is used to filter rows based on specific condition(s). You can use **WHERE** to locate all instances of duplicates based on the condition set for the query.

An analyst has a list of employee data containing information about everyone in the organization. The analyst would like to check for duplicate employees in the sales department to ensure the most up-to-date information. To be sure the entries are actually duplicates, the **SELECT** statement queries both the first and last name of employees, and the employee_id in the sales department.

```
Unset
SELECT employee_id, first_name, last_name
```

```
FROM employees
WHERE department = 'Sales';
```

If the table contains these rows of data:

Unset

```
| employee_id | first_name | last_name | department |
|-------------|------------|-----------|------------|
|      E001   | John       | Doe       | Sales      |
|      E002   | Jane       | Smith     | HR         |
|      E003   | Alice      | Johnson   | Sales      |
|      E004   | Bob        | White     | Marketing  |
|      E001   | John       | Doe       | Sales      |
|      E006   | Charlie    | Brown     | Sales      |
|      E007   | John       | Thompson  | IT         |
```

The query will return the output that contains duplicates for John Doe:

Unset

```
| employee_id | first_name | last_name |
|-------------|------------|-----------|
| E001        | John       | Doe       |
| E003        | Alice      | Johnson   |
| E001        | John       | Doe       |
```

The analyst can then delete the duplicates and update the table.

DISTINCT

DISTINCT is used to remove duplicate rows from a result set. Analysts use **DISTINCT** in combination with **WHERE** to drill down on their data and identify duplicates.

An analyst is working with this employees table. There are 4 employees in the sales department, but there's a duplicate by the name of John Doe.

Unset

```
| employee_id | first_name | last_name | department |
```

```
|-------------|------------|-----------|------------|
|      E001   | John       | Doe       | Sales      |
|      E002   | Jane       | Smith     | HR         |
|      E003   | Alice      | Johnson   | Sales      |
|      E004   | Bob        | White     | Marketing  |
|      E001   | John       | Doe       | Sales      |
|      E006   | Charlie    | Brown     | Sales      |
|      E007   | John       | Thompson  | IT         |
```

To query the results with no duplicates, the analyst utilizes the **DISTINCT** function to select employees by first and last name.

Unset

```
SELECT DISTINCT employee_id, first_name, last_name
FROM employees
WHERE department = 'Sales';
```

The query returns this output:

Unset

```
| employee_id | first_name | last_name |
|-------------|------------|-----------|
| E001        | John       | Doe       |
| E003        | Alice      | Johnson   |
| E006        | Charlie    | Brown     |
```

Note that this is the result of a specific query. The analyst will need to update the table to remove the duplicate permanently.

GROUP BY

GROUP BY groups rows that have the same values in specified columns into summary rows. **GROUP BY** is ideal for identifying duplicates within groups. If an analyst has this employees table, and wants to count the number of employees by department, they can use the **GROUP BY** function.

```
Unset
SELECT department, COUNT(DISTINCT employee_id) AS
unique_employee_count
FROM employees
GROUP BY department;
```

The **SELECT** statement includes a **COUNT** and **DISTINCT** function to ensure that duplicates are not counted.

If the table contains these rows of data:

```
Unset
| employee_id | first_name | last_name | department |
|-------------|------------|-----------|------------|
|      E001   | John       | Doe       | Sales      |
|      E002   | Jane       | Smith     | HR         |
|      E003   | Alice      | Johnson   | Sales      |
|      E004   | Bob        | White     | Marketing  |
|      E001   | John       | Doe       | Sales      |
|      E005   | Charlie    | Brown     | Sales      |
|      E006   | John       | Thompson  | IT         |
```

The query will result in this output:

```
Unset
| department | unique_employee_count |
|------------|-----------------------|
| Sales      |           3           |
| HR         |           1           |
| Marketing  |           1           |
| IT         |           1           |
```

Data derivation

Data derivation is the process of extrapolating data from other existing data. For data derivation, you can use the **CASE** statement, a conditional expression that allows you to derive values based on a specific criteria.

CASE can be used in a variety of ways to extrapolate data by: creating new columns based on existing column values, customizing the sorting of results, and replacing or transforming data values based on specific criteria.

You can use a **CASE** statement to create a grading scale for test scores. Instead of listing information by each individual score, the **CASE** statement enables you to create ranges of values, and assign the data to groups of your choosing. The **SELECT** statement groups first and last name, and grades of students into labeled groups based on their grade:

```
Unset
SELECT
  first_name,
  last_name,
  grade,
  CASE
    WHEN grade < 59 THEN 'F'
    WHEN grade BETWEEN 60 AND 69 THEN 'D'
    WHEN grade BETWEEN 70 AND 79 THEN 'C'
    WHEN grade BETWEEN 80 AND 89 THEN 'B'
    ELSE 'A'
  END AS grading_scale
FROM students;
```

If the table contains these rows of data:

```
Unset
| student_id | first_name | last_name | grade |
|------------|------------|-----------|-------|
|    S001    | John       | Doe       |   85  |
|    S002    | Jane       | Smith     |   58  |
|    S003    | Alice      | Johnson   |   78  |
|    S004    | Bob        | White     |   68  |
|    S005    | Charlie    | Brown     |   95  |
```

The query will result in this output:

```
Unset
| first_name | last_name | grade | grading_scale |
|------------|-----------|-------|---------------|
| John       | Doe       |   85  |       B       |
```

```
| Jane       | Smith     |   58  |       F       |
| Alice      | Johnson   |   78  |       C       |
| Bob        | White     |   68  |       D       |
| Charlie    | Brown     |   95  |       A       |
```

The **CASE** statement is a powerful tool for adding logic to your queries and deriving new information based on existing data.

Data filtering

Data filtering allows you to select specific portions of your dataset based on certain conditions. The **WHERE** clause is helpful for setting a condition in your query. You can filter your data even further by using **AND** or **OR** operators.

The **AND** operator requires both **WHERE** clause conditions to be true. **OR** requires that at least one condition is true. The **WHERE** clause condition you choose will determine the outcome of the query.

An analyst wants to determine which employees exceeded their sales quota and are in a senior position. The **SELECT** statement queries the first and last names, sales amount, and title from the employees table. The **WHERE** clause specifies that the result should only include records for employees who are in a senior position, and have surpassed their sales quota. Both conditions must be met in order to be included in the results.

Unset

```
SELECT first_name, last_name, sales, title
FROM employees
WHERE sales > quota AND title = 'Senior';
```

If the table contains these rows of data:

Unset

```
|-------------|------------|-----------|-------|-------|--------|
|    E001     | John       | Doe       |  5000 | 4500  | Senior
|
|    E002     | Jane       | Smith     |  4200 | 4500  | Junior
|
|    E003     | Alice      | Johnson   |  4800 | 4500  | Senior
|
```

```
|    E004     | Bob        | White     |  4600 | 4500  | Senior
|
|    E005     | Charlie    | Brown     |  4300 | 4200  | Junior
|
|    E006     | Daisy      | Green     |  4400 | 4500  | Senior
|
| employee_id | first_name | last_name | sales | quota | title
```

The query will output these results:

Unset

```
| first_name | last_name | sales | title  |
|------------|-----------|-------|--------|
| John       | Doe       |  5000 | Senior |
| Alice      | Johnson   |  4800 | Senior |
| Bob        | White     |  4600 | Senior |
```

The analyst modifies the **SELECT** statement by adjusting the **WHERE** clause to include employees that exceeded their sales quota, OR hold a senior position.

Unset

```
SELECT first_name, last_name, sales, title
FROM employees
WHERE sales > quota OR title = 'Senior';
```

The output is noticeably different. Only one condition needs to be true for the result to be included in the output.

Unset

```
| employee_id | first_name | last_name | sales | quota | title
|
|-------------|------------|-----------|-------|-------|--------|
|    E001     | John       | Doe       |  5000 | 4500  | Senior
|
|    E002     | Jane       | Smith     |  4200 | 4500  | Junior
|
|    E003     | Alice      | Johnson   |  4800 | 4500  | Senior
```

```
|
|    E004     | Bob         | White      |  4600 | 4500  | Senior
|
|    E005     | Charlie     | Brown      |  4300 | 4200  | Junior
|
|    E006     | Daisy       | Green      |  4400 | 4500  | Senior
```

The choice to choose **AND** or **OR** will depend on the final result you are trying to achieve.

Key takeaways

In this reading, you learned about data aggregation, deduplication, derivation and filtering in SQL, and explored specific examples of how to perform these functions. These data transformation techniques help analysts correct errors and reduce unnecessary details, making the data usable and accessible.