**Functions for Exploring Categorical Data in PostgreSQL**

COUNT()

Description: Counts the number of rows that match a specific condition or belong to a certain category.

Keywords/Indicators: "total number", "frequency", "count occurrences".

Syntax:

SELECT category\_column, COUNT(\*)

FROM table

GROUP BY category\_column;

GROUP BY

Description: Groups rows that have the same values in specified columns into summary rows.

Keywords/Indicators: "group data", "aggregate by category", "summarize by".

Syntax:

SELECT category\_column, aggregate\_function(column)

FROM table

GROUP BY category\_column;

DISTINCT

Description: Returns only distinct (unique) values in the result set, often used to find the number of unique categories.

Keywords/Indicators: "unique values", "remove duplicates", "distinct categories".

Syntax:

SELECT DISTINCT category\_column

FROM table;

CASE

Description: Provides conditional logic within SQL queries, useful for creating new categorical variables or recoding existing ones.

Keywords/Indicators: "conditional logic", "if-else", "categorize conditionally".

Syntax:

SELECT category\_column,

CASE

WHEN condition1 THEN 'Category1'

WHEN condition2 THEN 'Category2'

ELSE 'Other'

END as new\_category

FROM table;

MODE()

Description: Returns the most frequent value in a column, which can help identify the mode of a categorical dataset.

Keywords/Indicators: "most frequent", "mode", "common category".

Syntax:

SELECT MODE() WITHIN GROUP (ORDER BY category\_column) AS mode\_category

FROM table;

PERCENTILE\_CONT()

Description: Computes a value based on a specified percentile within a sorted dataset, useful for understanding the distribution of categorical data.

Keywords/Indicators: "percentile", "cumulative distribution", "categorical distribution".

Syntax:

SELECT PERCENTILE\_CONT(0.5) WITHIN GROUP (ORDER BY column)

FROM table;

CROSSTAB() (requires the tablefunc extension)

Description: Converts categorical data into a matrix (or pivot table) format, useful for contingency tables.

Keywords/Indicators: "pivot table", "contingency table", "cross-tabulation".

Syntax:

SELECT \* FROM CROSSTAB(

'SELECT category1, category2, COUNT(\*)

FROM table

GROUP BY category1, category2'

) AS ct(row\_name text, category1\_name text, category2\_name text, ...);

**Functions for Exploring Unstructured Text in PostgreSQL**

LOWER() / UPPER()

Description: Converts text to lowercase or uppercase, respectively, useful for standardizing text data.

Keywords/Indicators: "standardize text", "case-insensitive comparison", "convert to lowercase/uppercase".

Syntax:

SELECT LOWER(text\_column), UPPER(text\_column)

FROM table;

LENGTH()

Description: Returns the number of characters in a text string, useful for analyzing the length of unstructured text.

Keywords/Indicators: "text length", "number of characters", "length of string".

Syntax:

SELECT LENGTH(text\_column)

FROM table;

SUBSTRING()

Description: Extracts a portion of a string, useful for isolating specific parts of unstructured text.

Keywords/Indicators: "extract part of text", "substring", "portion of string".

Syntax:

SELECT SUBSTRING(text\_column FROM start\_position FOR length)

FROM table;

POSITION()

Description: Finds the position of a substring within a string, helpful for locating specific text within unstructured data.

Keywords/Indicators: "find substring", "location in text", "position of text".

Syntax:

SELECT POSITION('substring' IN text\_column)

FROM table;

TRIM()

Description: Removes specified characters from the beginning and end of a string, commonly used to clean up whitespace.

Keywords/Indicators: "remove whitespace", "clean text", "trim characters".

Syntax:

SELECT TRIM(BOTH ' ' FROM text\_column)

FROM table;

RIGHT()

Description: Extracts a specified number of characters from the right side of a string, which can be useful when analyzing or isolating the ending portion of categorical strings or unstructured text.

Keywords/Indicators: "extract end", "last characters", "rightmost part of string".

Syntax:

SELECT RIGHT(column\_name, number\_of\_characters)

FROM table;

LEFT()

Description: Extracts a specified number of characters from the left side of a string, useful for analyzing or isolating the beginning portion of categorical strings or unstructured text.

Keywords/Indicators: "extract start", "first characters", "leftmost part of string".

Syntax:

SELECT LEFT(column\_name, number\_of\_characters)

FROM table;

COALESCE()

Description: Returns the first non-null value in a list, often used to handle missing or null values in categorical data or text fields, providing a default value when necessary.

Keywords/Indicators: "handle nulls", "replace null values", "default value".

Syntax:

SELECT COALESCE(column\_name, 'Default Value')

FROM table;

REPLACE()

Description: Replaces all occurrences of a specified substring within a string with another substring. Useful for cleaning up or modifying text data.

Keywords/Indicators: "replace substring", "modify text", "string cleanup".

Syntax:

SELECT REPLACE(text\_column, 'old\_substring', 'new\_substring')

FROM table;

CONCAT()

Description: Combines two or more strings into one, useful for merging text data.

Keywords/Indicators: "concatenate text", "merge strings", "combine columns".

Syntax:

SELECT CONCAT(string1, string2, ...)

FROM table;

SPLIT\_PART()

Description: Splits a string into parts based on a delimiter and returns the specified part, useful for parsing unstructured text.

Keywords/Indicators: "split text", "delimiter", "extract part of string".

Syntax:

SELECT SPLIT\_PART(text\_column, delimiter, part\_number)

FROM table;

**Full-Text Search in PostgreSQL**

PostgreSQL provides full-text search capabilities that allow you to search for text within documents, handling linguistic complexities like stemming and ranking relevance. Full-text search is more advanced than simple pattern matching with LIKE and is optimized for querying large text fields.

Key Concepts of Full-Text Search

TO\_TSVECTOR()

Description: Converts a document (text) into a tsvector, which is a sorted list of words and their positions in the document. This function prepares text for full-text search.

Keywords/Indicators: "convert text", "prepare for search", "index text".

Syntax:

SELECT to\_tsvector('english', column)

FROM table;

Use When:

You want to convert a text column into a searchable vector. This is the first step in preparing your text for full-text search.

Example:

SELECT to\_tsvector('english', 'PostgreSQL provides full-text search capabilities');

Explanation: Converts the sentence into a tsvector for indexing.

TO\_TSQUERY()

Description: Converts a query string into a tsquery for searching a tsvector. You can combine search terms using logical operators like & (AND), | (OR), and ! (NOT).

Keywords/Indicators: "create search query", "define search terms", "search with logic".

Syntax:

SELECT to\_tsquery('english', 'search\_term');

Use When:

You want to construct a full-text search query to search for words in your document.

Example:

SELECT to\_tsquery('english', 'PostgreSQL & search');

Explanation: Creates a query that searches for documents containing both "PostgreSQL" and "search".

@@ (Text Search Match Operator)

Description: The match operator (@@) checks if a tsvector matches a tsquery.

Keywords/Indicators: "match text", "find documents with terms", "search document".

Syntax:

SELECT column

FROM table

WHERE to\_tsvector('english', column) @@ to\_tsquery('english', 'search\_term');

Use When:

You want to filter rows in a table based on whether they contain specific search terms.

Example:

SELECT document

FROM documents

WHERE to\_tsvector('english', document) @@ to\_tsquery('english', 'PostgreSQL & search');

Explanation: This query returns documents that contain both "PostgreSQL" and "search".

TS\_RANK()

Description: Ranks documents based on their relevance to the search terms. The higher the rank, the more relevant the document is.

Keywords/Indicators: "rank results", "order by relevance", "relevance score".

Syntax:

SELECT column, ts\_rank(to\_tsvector('english', column), to\_tsquery('english', 'search\_term')) AS rank

FROM table

ORDER BY rank DESC;

Use When:

You want to rank search results based on how closely they match the search terms.

Example:

SELECT document, ts\_rank(to\_tsvector('english', document), to\_tsquery('english', 'PostgreSQL & search')) AS rank

FROM documents

ORDER BY rank DESC;

Explanation: This query returns documents that match the search terms, ordered by how relevant they are.

PLAINTO\_TSQUERY()

Description: Converts plain text into a tsquery by automatically parsing the input string. This function is useful when you don't need complex logical operators.

Keywords/Indicators: "convert plain text to query", "simple search terms", "basic search".

Syntax:

SELECT plainto\_tsquery('english', 'search phrase');

Use When:

You want to perform a basic full-text search using simple words or phrases.

Example:

SELECT document

FROM documents

WHERE to\_tsvector('english', document) @@ plainto\_tsquery('english', 'PostgreSQL search');

Explanation: This query searches for documents containing the words "PostgreSQL" and "search" without the need for explicit logical operators.

TS\_HEADLINE()

Description: Highlights the matched words in a document by wrapping them in HTML tags or other formatting markers.

Keywords/Indicators: "highlight terms", "show matched terms", "format search results".

Syntax:

SELECT ts\_headline('english', column, to\_tsquery('english', 'search\_term'))

FROM table;

Use When:

You want to display the results of your search with the matching terms highlighted.

Example:

SELECT ts\_headline('english', document, to\_tsquery('english', 'PostgreSQL & search'))

FROM documents;

Explanation: This query highlights the words "PostgreSQL" and "search" in the result set.

**Pattern Matching in PostgreSQL: LIKE, ILIKE, NOT LIKE, and Wildcards**

PostgreSQL provides pattern matching using the LIKE, ILIKE, and NOT LIKE operators, often in conjunction with wildcard characters (% and \_). These allow flexible string searches that can match portions of text.

LIKE

Description: The LIKE operator is used to perform case-sensitive pattern matching. You can use wildcards to define the pattern to match.

Keywords/Indicators: "case-sensitive search", "partial string match", "exact pattern match".

Wildcards:

% (percent): Matches any sequence of characters (including zero characters).

\_ (underscore): Matches any single character.

Syntax:

SELECT column

FROM table

WHERE column LIKE 'pattern';

Use When:

You need to perform a case-sensitive search for a string that partially or exactly matches a pattern.

Example:

SELECT employee\_name

FROM employees

WHERE employee\_name LIKE 'A%';

Explanation: This query returns employees whose names start with the letter "A".

ILIKE

Description: The ILIKE operator is used for case-insensitive pattern matching. This is helpful when you don't care about the case (uppercase/lowercase) of the text being matched.

Keywords/Indicators: "case-insensitive search", "ignore case in string match", "partial string match regardless of case".

Wildcards:

% (percent): Matches any sequence of characters (including zero characters).

\_ (underscore): Matches any single character.

Syntax:

SELECT column

FROM table

WHERE column ILIKE 'pattern';

Use When:

You want to match a string regardless of case, e.g., matching both "Apple" and "apple".

Example:

SELECT employee\_name

FROM employees

WHERE employee\_name ILIKE 'a%';

Explanation: This query returns employees whose names start with the letter "A" or "a", ignoring case.

NOT LIKE

Description: The NOT LIKE operator is the opposite of LIKE, used to exclude rows that match a certain pattern.

Keywords/Indicators: "exclude pattern", "not matching string", "doesn't match condition".

Wildcards:

% (percent): Matches any sequence of characters (including zero characters).

\_ (underscore): Matches any single character.

Syntax:

SELECT column

FROM table

WHERE column NOT LIKE 'pattern';

Use When:

You want to exclude rows that match a specific pattern or text.

Example:

SELECT employee\_name

FROM employees

WHERE employee\_name NOT LIKE 'A%';

Explanation: This query excludes employees whose names start with the letter "A".

Wildcards in LIKE/ILIKE

% (Percent Wildcard): Matches any sequence of characters, including none.

Use When: You want to match anything before, after, or in between certain characters.

Example:

SELECT product\_name

FROM products

WHERE product\_name LIKE '%phone%';

Explanation: This query finds all product names containing the word "phone" anywhere in the text.

\_ (Underscore Wildcard): Matches exactly one character.

Use When: You want to match strings with exactly one character difference at a certain position.

Example:

SELECT product\_name

FROM products

WHERE product\_name LIKE '\_phone';

Explanation: This query matches product names where the first character is any character, followed by the word "phone".

**Types of Joins**

INNER JOIN

Description: Retrieves only the rows that have matching values in both tables. If a row in either table doesn’t have a match in the other table, it will not be included in the result set.

Keywords/Indicators: "only matching rows", "common records", "both tables".

Syntax:

SELECT columns

FROM table1

INNER JOIN table2 ON table1.column = table2.column;

LEFT JOIN (or LEFT OUTER JOIN)

Description: Retrieves all rows from the left table, and the matched rows from the right table. If there is no match, NULL values are returned for columns from the right table.

Keywords/Indicators: "all rows from the left table", "include unmatched rows from the left", "left table's rows regardless of matching".

Syntax:

SELECT columns

FROM table1

LEFT JOIN table2 ON table1.column = table2.column;

RIGHT JOIN (or RIGHT OUTER JOIN)

Description: Retrieves all rows from the right table, and the matched rows from the left table. If there is no match, NULL values are returned for columns from the left table.

Keywords/Indicators: "all rows from the right table", "include unmatched rows from the right", "right table's rows regardless of matching".

Syntax:

SELECT columns

FROM table1

RIGHT JOIN table2 ON table1.column = table2.column;

FULL JOIN (or FULL OUTER JOIN)

Description: Retrieves all rows when there is a match in either the left or right table. It combines the result of both LEFT JOIN and RIGHT JOIN.

Keywords/Indicators: "all rows from both tables", "include unmatched rows from both", "union of LEFT JOIN and RIGHT JOIN".

Syntax:

SELECT columns

FROM table1

FULL JOIN table2 ON table1.column = table2.column;

CROSS JOIN

Description: Produces a Cartesian product of the two tables, meaning every row from the first table is combined with every row from the second table.

Keywords/Indicators: "Cartesian product", "all combinations of rows", "multiply tables".

Syntax:

SELECT columns

FROM table1

CROSS JOIN table2;

SELF JOIN

Description: Joins a table with itself. This is useful for finding relationships between rows in the same table.

Keywords/Indicators: "compare rows within the same table", "self-reference", "rows in relation to themselves".

Syntax:

SELECT a.columns, b.columns

FROM table1 a

INNER JOIN table1 b ON a.column = b.column;

**Set Operations in PostgreSQL**

UNION

Description: Combines the result sets of two SELECT statements and removes duplicate rows.

Keywords/Indicators: "combine results", "remove duplicates", "union of queries".

Syntax:

SELECT columns FROM table1

UNION

SELECT columns FROM table2;

UNION ALL

Description: Combines the result sets of two SELECT statements, but includes all duplicates.

Keywords/Indicators: "combine results", "keep duplicates", "all records from both queries".

Syntax:

SELECT columns FROM table1

UNION ALL

SELECT columns FROM table2;

INTERSECT

Description: Returns the common rows that appear in both result sets of two SELECT statements.

Keywords/Indicators: "common records", "intersection of queries", "records in both".

Syntax:

SELECT columns FROM table1

INTERSECT

SELECT columns FROM table2;

**Types of Subqueries in PostgreSQL**

Basic Subquery

Description: A subquery is a query nested inside another query. It can be used in various parts of the main query, such as the SELECT list, FROM clause, or WHERE clause.

Keywords/Indicators: "nested query", "inner query", "query within a query".

Syntax:

SELECT column1, column2

FROM table

WHERE column3 = (SELECT column3 FROM another\_table WHERE condition);

Subquery in WHERE Clause

Description: A subquery in the WHERE clause is used to filter results based on the outcome of another query.

Keywords/Indicators: "filter by another query", "compare result from another query", "conditional filtering".

Syntax:

SELECT column

FROM table

WHERE column IN (SELECT column FROM another\_table WHERE condition);

Use When:

You want to filter rows in a query based on values returned from another query. For example, selecting users who belong to departments listed in another table.

Example:

SELECT employee\_id, employee\_name

FROM employees

WHERE department\_id IN (SELECT department\_id FROM departments WHERE location = 'New York');

Explanation: This query returns employees who work in departments located in New York.

Subquery in FROM Clause

Description: A subquery in the FROM clause (also called a derived table) allows the result of the subquery to act as a table that can be queried further.

Keywords/Indicators: "use query as table", "derived table", "temporary result for main query".

Syntax:

SELECT column

FROM (SELECT column FROM table WHERE condition) AS subquery\_alias;

Use When:

You need to create a temporary result set that can be queried further in the main query. For example, if you need to aggregate or manipulate data before performing additional calculations or joins.

Example:

SELECT subquery.employee\_name, subquery.department\_id

FROM (SELECT employee\_name, department\_id FROM employees WHERE salary > 5000) AS subquery;

Explanation: This query creates a temporary table of employees with a salary greater than 5000, then selects the employee name and department from that result set.

Subquery in SELECT Clause

Description: A subquery in the SELECT clause allows you to compute additional columns based on the result of another query.

Keywords/Indicators: "compute column with query", "additional column from subquery", "dynamic calculation in select".

Syntax:

SELECT column, (SELECT expression FROM another\_table WHERE condition) AS new\_column

FROM table;

Use When:

You want to include a computed column in your query, where the value of the new column is determined by a subquery. For example, fetching the highest salary per department for each employee.

Example:

SELECT employee\_name,

(SELECT MAX(salary) FROM employees AS e2 WHERE e2.department\_id = e1.department\_id) AS max\_salary\_in\_dept

FROM employees AS e1;

Explanation: This query returns the name of each employee along with the maximum salary in their department, as calculated by the subquery.

Common Table Expressions (CTE)

Description: A CTE is a named temporary result set that can be referenced within a SELECT, INSERT, UPDATE, or DELETE statement. It makes complex queries easier to read and maintain.

Keywords/Indicators: "with clause", "temporary result set", "improve readability".

Syntax:

WITH cte\_name AS (

SELECT column1, column2

FROM table

WHERE condition

)

SELECT \*

FROM cte\_name

WHERE another\_condition;

Temporary Table

Description: A temporary table stores intermediate results temporarily during the session. It is useful for breaking down complex operations into manageable steps.

Keywords/Indicators: "intermediate results", "session-specific", "complex operations".

Syntax:

CREATE TEMPORARY TABLE temp\_table AS

SELECT column1, column2

FROM table

WHERE condition;

SELECT \*

FROM temp\_table

WHERE another\_condition;

Correlated Subquery

Description: A correlated subquery is a subquery that references columns from the outer query. It is executed once for each row processed by the outer query.

Keywords/Indicators: "reference outer query", "row-by-row", "nested with dependencies".

Syntax:

SELECT column1, column2

FROM table1

WHERE column3 > (SELECT AVG(column3)

FROM table2

WHERE table2.column4 = table1.column4);

**Window Functions in PostgreSQL**

ROW\_NUMBER()

Description: Assigns a unique sequential integer to rows within a partition of a result set, starting at 1 for the first row.

Keywords/Indicators: "row numbering", "sequence", "rank rows".

Syntax:

SELECT column1, column2,

ROW\_NUMBER() OVER (PARTITION BY partition\_column ORDER BY order\_column) AS row\_num

FROM table;

RANK()

Description: Assigns a rank to each row within a partition of a result set, with gaps in ranking values for ties.

Keywords/Indicators: "ranking", "rank with gaps", "ordered ranking".

Syntax:

SELECT column1, column2,

RANK() OVER (PARTITION BY partition\_column ORDER BY order\_column) AS rank

FROM table;

PARTITION BY

Description: The PARTITION BY clause divides the result set into partitions, and the window function is applied to each partition separately.

Keywords/Indicators: "group rows for window function", "calculate within partitions", "apply function per group".

Syntax:

SELECT column,

window\_function() OVER (PARTITION BY partition\_column ORDER BY order\_column)

FROM table;

Use When:

You need to calculate something like a rank, running total, or average, but only within certain groups or categories.

Examples include ranking employees within each department or calculating the cumulative sales for each region.

Example:

SELECT employee\_name, department\_id,

RANK() OVER (PARTITION BY department\_id ORDER BY salary DESC) AS rank\_in\_dept

FROM employees;

Explanation: This query ranks employees by salary within each department. The ranking restarts for each department (PARTITION BY department\_id).

Sliding Windows (Using ROWS or RANGE)

Description: Sliding windows define a frame of rows over which the window function operates, moving along the result set. The frame is defined using ROWS or RANGE.

ROWS: Refers to a specific number of rows before or after the current row.

RANGE: Refers to a logical range of values before or after the current row, based on the order column.

Keywords/Indicators: "sliding window", "moving average", "running total", "define frame".

Syntax:

SELECT column,

window\_function() OVER (ORDER BY order\_column

ROWS BETWEEN start\_boundary AND end\_boundary)

FROM table;

Use When:

You need to calculate metrics like moving averages, cumulative sums, or any other calculation that should consider a sliding window of rows around the current row.

Example 1: Cumulative Total with ROWS:

SELECT date, sales,

SUM(sales) OVER (ORDER BY date

ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW) AS running\_total

FROM sales\_data;

Explanation: This query calculates a running total of sales, considering all previous rows (UNBOUNDED PRECEDING) up to the current row.

Example 2: Moving Average with ROWS:

SELECT date, sales,

AVG(sales) OVER (ORDER BY date

ROWS BETWEEN 2 PRECEDING AND CURRENT ROW) AS moving\_avg

FROM sales\_data;

Explanation: This query calculates a 3-day moving average of sales, using the current row and the two preceding rows (ROWS BETWEEN 2 PRECEDING AND CURRENT ROW).

Example 3: Moving Average with RANGE:

SELECT date, sales,

AVG(sales) OVER (ORDER BY date

RANGE BETWEEN INTERVAL '3 DAYS' PRECEDING AND CURRENT ROW) AS range\_moving\_avg

FROM sales\_data;

Explanation: This query calculates a moving average using a 3-day range (RANGE BETWEEN INTERVAL '3 DAYS' PRECEDING), which groups rows by date range, not by the number of rows.

DENSE\_RANK()

Description: Assigns a rank to each row within a partition of a result set, without gaps in ranking values for ties.

Keywords/Indicators: "dense ranking", "rank without gaps", "continuous ranking".

Syntax:

SELECT column1, column2,

DENSE\_RANK() OVER (PARTITION BY partition\_column ORDER BY order\_column) AS dense\_rank

FROM table;

NTILE()

Description: Divides the result set into a specified number of roughly equal parts and assigns a bucket number to each row.

Keywords/Indicators: "bucket distribution", "percentile division", "quantiles".

Syntax:

SELECT column1, column2,

NTILE(number\_of\_buckets) OVER (PARTITION BY partition\_column ORDER BY order\_column) AS bucket

FROM table;

SUM()

Description: Calculates the sum of a specified column's values over a specified window frame.

Keywords/Indicators: "running total", "cumulative sum", "windowed aggregation".

Syntax:

SELECT column1, column2,

SUM(column\_to\_sum) OVER (PARTITION BY partition\_column ORDER BY order\_column ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW) AS running\_total

FROM table;

AVG()

Description: Calculates the average of a specified column's values over a specified window frame.

Keywords/Indicators: "moving average", "average over window", "windowed average".

Syntax:

SELECT column1, column2,

AVG(column\_to\_avg) OVER (PARTITION BY partition\_column ORDER BY order\_column ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW) AS moving\_avg

FROM table;

MIN()

Description: Finds the minimum value of a specified column's values over a specified window frame.

Keywords/Indicators: "minimum value", "windowed min", "rolling minimum".

Syntax:

SELECT column1, column2,

MIN(column\_to\_min) OVER (PARTITION BY partition\_column ORDER BY order\_column ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW) AS min\_value

FROM table;

MAX()

Description: Finds the maximum value of a specified column's values over a specified window frame.

Keywords/Indicators: "maximum value", "windowed max", "rolling maximum".

Syntax:

SELECT column1, column2,

MAX(column\_to\_max) OVER (PARTITION BY partition\_column ORDER BY order\_column ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW) AS max\_value

FROM table;

LEAD()

Description: Accesses data from the next row in the result set, useful for comparing values between consecutive rows.

Keywords/Indicators: "next row value", "future row comparison", "look ahead".

Syntax:

SELECT column1, column2,

LEAD(column\_to\_access, offset) OVER (PARTITION BY partition\_column ORDER BY order\_column) AS next\_value

FROM table;

LAG()

Description: Accesses data from the previous row in the result set, useful for comparing values between the current and previous rows.

Keywords/Indicators: "previous row value", "past row comparison", "look back".

Syntax:

SELECT column1, column2,

LAG(column\_to\_access, offset) OVER (PARTITION BY partition\_column ORDER BY order\_column) AS previous\_value

FROM table;

FIRST\_VALUE()

Description: Returns the first value in the result set within the specified window frame.

Keywords/Indicators: "first value", "initial value in window", "first row value".

Syntax:

SELECT column1, column2,

FIRST\_VALUE(column\_to\_access) OVER (PARTITION BY partition\_column ORDER BY order\_column) AS first\_value

FROM table;

LAST\_VALUE()

Description: Returns the last value in the result set within the specified window frame.

Keywords/Indicators: "last value", "final value in window", "last row value".

Syntax:

SELECT column1, column2,

LAST\_VALUE(column\_to\_access) OVER (PARTITION BY partition\_column ORDER BY order\_column ROWS BETWEEN CURRENT ROW AND UNBOUNDED FOLLOWING) AS last\_value

FROM table;

Working with Dates and Timestamps in PostgreSQL

CURRENT\_DATE

Description: Returns the current date (without the time part).

Keywords/Indicators: "current date", "today's date", "current day".

Syntax:

SELECT CURRENT\_DATE;

CURRENT\_TIME

Description: Returns the current time (without the date part).

Keywords/Indicators: "current time", "current hour", "current minute".

Syntax:

SELECT CURRENT\_TIME;

CURRENT\_TIMESTAMP

Description: Returns the current date and time, including timezone information.

Keywords/Indicators: "current timestamp", "current date and time", "current datetime".

Syntax:

SELECT CURRENT\_TIMESTAMP;

DATE\_TRUNC()

Description: Truncates a timestamp or date to a specified precision (e.g., day, month, year).

Keywords/Indicators: "truncate date", "round down to", "reset time part".

Syntax:

SELECT DATE\_TRUNC('precision', timestamp\_column)

FROM table;

Examples:

'day' for truncating to the start of the day.

'month' for truncating to the start of the month.

EXTRACT()

Description: Extracts a specific part of a date or timestamp (e.g., year, month, day).

Keywords/Indicators: "extract part", "get year/month/day", "date component".

Syntax:

SELECT EXTRACT(part FROM date\_column)

FROM table;

Examples:

EXTRACT(YEAR FROM timestamp\_column) for the year.

EXTRACT(MONTH FROM timestamp\_column) for the month.

DATE\_PART()

Description: Similar to EXTRACT(), it returns a specific part of a date or timestamp.

Keywords/Indicators: "part of date", "date component", "date portion".

Syntax:

SELECT DATE\_PART('part', timestamp\_column)

FROM table;

Examples:

'year' for extracting the year.

'month' for extracting the month.

AGE()

Description: Calculates the difference between two timestamps or between a timestamp and the current time, returning the result as an interval.

Keywords/Indicators: "calculate age", "time difference", "interval".

Syntax:

SELECT AGE(timestamp1, timestamp2)

FROM table;

Examples:

AGE(CURRENT\_DATE, birth\_date) for the age of a person.

INTERVAL

Description: Represents a span of time, useful for adding or subtracting time from a timestamp or date.

Keywords/Indicators: "time span", "add/subtract time", "duration".

Syntax:

SELECT timestamp\_column + INTERVAL '1 day'

FROM table;

Examples:

INTERVAL '1 day' to add one day.

INTERVAL '1 month' to subtract one month.

TO\_DATE()

Description: Converts a string to a date type, given a specified format.

Keywords/Indicators: "convert string to date", "parse date", "string to date".

Syntax:

SELECT TO\_DATE('string\_date', 'format')

FROM table;

Examples:

TO\_DATE('2024-08-29', 'YYYY-MM-DD') for converting a string to a date.

TO\_TIMESTAMP()

Description: Converts a string to a timestamp type, given a specified format.

Keywords/Indicators: "convert string to timestamp", "parse timestamp", "string to timestamp".

Syntax:

SELECT TO\_TIMESTAMP('string\_timestamp', 'format')

FROM table;

Examples:

TO\_TIMESTAMP('2024-08-29 14:30:00', 'YYYY-MM-DD HH24:MI:SS') for converting a string to a timestamp.

NOW()

Description: Returns the current date and time including timezone information, similar to CURRENT\_TIMESTAMP.

Keywords/Indicators: "current datetime", "current timestamp", "now".

Syntax:

SELECT NOW();

DATE\_PART()

Description: Returns a specified part of a date or timestamp as a number.

Keywords/Indicators: "extract part", "date component", "get specific date part".

Syntax:

SELECT DATE\_PART('part', date\_column)

FROM table;

Examples:

DATE\_PART('year', timestamp\_column) for the year.

DATE\_PART('month', timestamp\_column) for the month.