Functions

We've used functions in PostgreSQL before.

Simple math functions like SUM(), AVG(), MAX(), MIN() etc

In this document we look at important SQL functions you should know about.

Then we create functions based on SELECT statements.

- how to create a new SQL function
- difference in function volatility
- function overloading
- benefits of functions over simple queries

SQL functions use simple queries. They are efficient but lack any logic control such as LOOPs and variables.

For that we need a procedural language such as PL/pgSQL.

Important SQL functions

COALESCE

see https://www.postgresql.org/docs/current/functions-conditional.html#FUNCTIONS-COALESCE-NVL-IFNULL

> The **COALESCE** function returns **the first of its arguments that is not null**.

If all arguments are null, **COALESCE** returns null.

- `select COALESCE(1,2,3,4,5,6) -> 1`
- `select COALESCE(NULL,2) -> 2`
- `select COALESCE(NULL,NULL) -> NULL`

If the last column is a string COALESCE will always return that string if the other arguments are NULL.

So COALESCE is often used to substitute a default value for null values when data is retrieved for display, for example:

This query returns `description` if it is not null, otherwise `short_description` if it is not null, otherwise `(none)`.

```SQL
SELECT COALESCE(description, short_description, '(none)')
```

This query returns `unknown` when `domain` is null

```SQL
SELECT COALESCE(domain, 'unknown' ) from trees;  --
```

Practice

We're working with the normalized version of the trees database. `treesdb_v03`

Using the trees table. select a random number of trees, (`id` , `height` , `remarkable`). If the column `remarkable` is null, display `unknown` instead

> Note: all arguments of `COALESCE` need to have the same type. To cast a Boolean into a text use `name_of_boolean_column::text` .

[solution]

```SQL
<your query>
```

CONCAT function

To concatenate string you can use the `CONCAT` function which concatenates the text representations of all the arguments and ignores NULL arguments.

```SQL
concat ( val1 "any" [, val2 "any" [, ...] ] ) → text
```

For instance:

```SQL
concat('abcde', 2, NULL, 22) → abcde222
```

the `||` symbol

You can also use the `||` symbol to concatenate strings.

```
text || text → text
```

Concatenates the two strings.

```SQL
'Post' || 'greSQL' → PostgreSQL
```

Exercise

Using the `treesdb_v03`, concatenate multiple columns into one

columns are :

- name
- genre
- species
- variety
- domain
- arrondissement

also

- if one value is null, replace with `UNK`, using `COALESCE`.
- use `concat_ws` to add a '-' (upper dash) between all the columns
- replace all the spaces with a column value with '_' with `REPLACE()`

see https://www.postgresql.org/docs/current/functions-string.html

Note: you don't have to return all the rows each time. You can limit to N random() rows;

Note: first build the main query that returns all the columns, then coalesce, then concatenate, then replace

```SQL
<your query>
```

Create your own SQL function

You can create a SQL function with:

```SQL
CREATE [OR REPLACE] FUNCTION function_name(parameter1 type, parameter2 
RETURNS return_type
AS $$
    SQL_statement()
$$ LANGUAGE SQL;
```

where `SQL_STATEMENT` is a SQL query.

For instance

```SQL
CREATE OR REPLACE FUNCTION add_numbers(a integer, b integer)
RETURNS integer
AS $$
    SELECT a + b;
$$ LANGUAGE SQL;
```

which you can then use directly with :

```SQL
SELECT add_numbers(5, 3);
```

Key points about SQL vs PL/pgSQL functions

- Performance: SQL functions can often be inlined by the query planner, potentially leading to better performance than PL/pgSQL functions for simple operations.
- Simplicity: They're easier to write and understand for straightforward database operations.
- Limitations: SQL functions have limited procedural capabilities compared to PL/pgSQL. They can't use variables, loops, or complex control structures.

Why use Functions instead of SQL statement

If SQL functions are just simple queries, why use them ? Why not write the SQL query instead ?

Using a simple **SQL function** to insert a new row versus using a direct `INSERT` query has a few potential benefits and trade-offs, depending on your use case. Here are some points to consider:

Benefits of using a SQL function for INSERT operations:

1. **Reusability:**

- If you frequently need to insert rows with similar logic (such as setting default values or checking conditions), you can encapsulate that logic in a function. This avoids writing the same `INSERT` query multiple times in different parts of your application.
- Example: If you always insert a customer and automatically assign a default discount rate, this can be done inside a function.

2. **Abstraction:**

   - A function can abstract the underlying structure of the table. If you ever change the schema of the table (e.g., adding new columns), you can modify the function rather than updating every `INSERT` query across your application.
   - Example: If a column is added to a table, the function can take care of inserting default or derived values without needing to update application code.

3. **Encapsulation of Logic:**

   - Functions can enforce certain business logic, such as constraints or transformations, before performing the insert. For example, a function could check conditions or modify input data before inserting it, even though you're using a simple SQL function.
   - Example: A function could perform an `INSERT` only if a certain condition is met, like checking if the record already exists in the table.

4. **Parameter Handling:**

   - A function provides a simple interface where you pass parameters, and the function takes care of handling them, making the code cleaner when used from an external application.

5. **Security and Permissions:**

   - You can grant users or roles permission to execute the function without giving them direct `INSERT` permissions on the table. This can be useful for controlling access and enforcing specific behavior.
   - Example: You allow users to call the `insert_customer` function but do not give them direct `INSERT` privileges on the `customers` table.

6. **Transaction Control:**

   - Functions can be part of larger transactions. When used in combination with other database operations, a function call can contribute to an atomic transaction (all changes succeed or fail together), even in simple SQL functions.

Trade-offs of using a SQL function instead of a direct INSERT

query:

1. **Overhead:**
    - There might be a slight performance overhead when calling a function compared to running a direct `INSERT` query, as the function adds an extra layer of abstraction.
    - For simple use cases, a direct `INSERT` query might be faster.
2. **Complexity:**
    - If the function is very simple (e.g., just a one-liner `INSERT` without additional logic), the benefit of using a function might be minimal, and a direct `INSERT` query could suffice.
3. **Less Flexibility (in simple SQL functions):**
    - Simple SQL functions are limited in terms of logic and error handling compared to a direct query that you can wrap in more complex procedural code (e.g., using PL/pgSQL or an external programming language).
4. **Debugging and Transparency:**
    - If an error occurs, tracking down issues might be more complex when they are wrapped inside a function, compared to a direct `INSERT` where you can see exactly what query is being executed.

Example

Using a function:

```SQL
CREATE OR REPLACE FUNCTION insert_customer(c_name TEXT, c_email TEXT)
RETURNS VOID AS $$
    INSERT INTO customers (name, email)
    VALUES (c_name, c_email);
$$ LANGUAGE sql;

-- Then you can call it:
SELECT insert_customer('John Doe', 'john@example.com');
```

Direct query:

```sql
INSERT INTO customers (name, email)
VALUES ('John Doe', 'john@example.com');
```

When to prefer functions over direct queries

- If you plan to reuse the logic multiple times across your codebase.
- If you need to enforce business rules or perform additional transformations.
- If you want to abstract the underlying table structure or schema.
- If you need to provide controlled access through permissions.

When to prefer direct queries

- For one-off or very simple inserts.
- When you want maximum performance and have no additional logic to handle.

In summary, functions provide reusability, encapsulation, and the ability to enforce logic, but if your insert operation is straightforward with no additional logic, a direct `INSERT` query is sufficient.

Optimize SQL functions by setting their volatility

In PostgreSQL, function volatility categories (VOLATILE, STABLE, and IMMUTABLE) are important concepts that inform the query planner about the behavior of functions.

Setting the volatility of the SQL function helps the query planner optimize it.

To specify if a function should be considered as VOLATILE, STABLE or IMMUTABLE, you add the keyword after `$$ LANGUAGE SQL <VOLATILE | STABLE | IMMUTABLE>;`

VOLATILE Functions

- Default category if not specified
- Behavior:
  - Can modify the database
  - Can return different results on successive calls with the same arguments
- Examples:
  - Functions that use random number generation
  - Functions that modify database state
  - Functions that depend on current time

Here's an example of a VOLATILE function:

```SQL
CREATE FUNCTION get_random_number() RETURNS integer AS $$
    SELECT floor(random() * 100)::integer;
$$ LANGUAGE SQL VOLATILE;
```

STABLE Functions

- Behavior:
  - Cannot modify the database
  - For the same input arguments, will return the same result within a single table scan
  - Results might change across SQL statements

- Examples:
  - Functions that depend on database content (but not on values that change within a single query)
  - Functions that depend on configuration settings
  - Current timestamp functions (within a transaction)

Here's an example of a STABLE function:

```SQL
CREATE FUNCTION get_current_timestamp() RETURNS timestamp AS $$
    SELECT current_timestamp;
$$ LANGUAGE SQL STABLE;

-- Usage:
SELECT get_current_timestamp();
```

IMMUTABLE Functions

- Behavior:
  - Cannot modify the database
  - Always return the same result for the same input arguments
  - Results do not depend on any database content or external state

- Examples:
  - Mathematical functions
  - String manipulation functions that don't depend on database settings

Here's an example of an IMMUTABLE function:

```sql
CREATE FUNCTION add_numbers(a integer, b integer) RETURNS integer AS $$
    SELECT a + b;
$$ LANGUAGE SQL IMMUTABLE;
```

Key Points:

1. Optimization:
    - The query planner can optimize STABLE and IMMUTABLE functions more effectively than VOLATILE functions.
    - IMMUTABLE function calls can be pre-computed during query planning if their arguments are constant.
    - STABLE functions can be optimized in WHERE clauses and joins, as the planner knows their value won't change within a query.
2. Default Behavior:
    - Functions are considered VOLATILE by default unless specified otherwise.
3. Performance Impact:
    - Correctly categorizing your functions can lead to significant performance improvements in complex queries.
4. Incorrect Categorization:
    - Marking a function as STABLE or IMMUTABLE when it's actually VOLATILE can lead to incorrect query results.
    - It's safer to mark a function as more volatile than it actually is (e.g., STABLE instead of IMMUTABLE) if you're unsure.
5. Usage in Indexes:
    - Only IMMUTABLE functions can be used directly in index definitions.

Here's an example demonstrating how volatility affects query planning:

```sql
CREATE TABLE items (id serial PRIMARY KEY, price numeric);

-- VOLATILE function (default)
CREATE FUNCTION get_tax_rate() RETURNS numeric AS $$
    SELECT 0.1; -- Assume this could change
$$ LANGUAGE SQL;

-- IMMUTABLE function
CREATE FUNCTION calculate_tax(price numeric) RETURNS numeric AS $$
    SELECT price * 0.1;
$$ LANGUAGE SQL IMMUTABLE;

-- This query will recalculate get_tax_rate() for every row
EXPLAIN ANALYZE SELECT id, price * get_tax_rate() FROM items;

-- This query can optimize the calculate_tax() call
EXPLAIN ANALYZE SELECT id, calculate_tax(price) FROM items;
```

In the second query, the planner might be able to compute `calculate_tax` just once if it determines that this optimization is beneficial.

Rule of thumb:

- **VOLATILE**: Use when the function can modify database state or its result can change even with the same inputs (e.g., random(), current_timestamp).
- **STABLE**: Use when the function doesn't modify the database and returns consistent results for the same inputs within a single table scan, but may change across statements (e.g., now(), current_date).
- **IMMUTABLE**: Use when the function always returns the same result for the same arguments, regardless of any database or external state (e.g., abs(), lower()).

It's safer to choose the more volatile option. Incorrectly marking a function as less volatile than it actually is can lead to incorrect query results, while being more conservative only potentially misses some optimization opportunities.