Worksheet: Using Functions in PL/pgSQL

In this exercise, the goal is to generate a unique `sha256` hash string for each tree entry in the `treesdb_v03` database. The SHA hash string that will be generated takes multiple columns describing tree characteristics as input.

`sha256` in PostgreSQL

The `sha256` function in PostgreSQL is used to generate a cryptographic hash of data,. It is part of the `pgcrypto` extension, which provides various cryptographic functions, including hashing algorithms.

To use the `sha256` function, you utilize the `digest()` function provided by `pgcrypto`. It requires two input parameters:

1. **Data**: The string or data to be hashed.
2. **Algorithm**: The name of the hash algorithm, in this case, `'sha256'`.

Before using the function, you must install the `pgcrypto` extension, which can be done by running:

```sql
CREATE EXTENSION pgcrypto;
```

Once installed, you can hash a string as shown in the following example:

```sql
SELECT encode(digest('example_string', 'sha256'), 'hex') AS sha256_hash
```

This example generates the SHA-256 hash of the string `example_string` and returns it in a human-readable hexadecimal format.

```sql
SELECT encode(digest('example_string', 'sha256'), 'hex') AS sha256_hash
                             sha256_hash
------------------------------------------------------------------
 9bfb55a8406617ff3e6767ec5d27fc6b5682c3a79c415ef6e084bf7d050273e6
```

Benefits of sha256

Hashing the columns that fully describe a tree (such as `name`, `domain`, `genre`, `species`, `variety`, `arrondissement`) has several advantages:

1. Unique Identifier Creation

- By concatenating all descriptive columns and applying a hash function like `sha256`, you create a **unique, fixed-length identifier** (hash) for each tree. This ensures that even with varying column lengths, the hash size remains constant.

Very useful for content that varies a lot in length like articles, blog posts, etc

2. Efficient Comparison

- Instead of comparing multiple columns to check if two trees are the same, you can compare a single hash value. This speeds up querying and comparisons, especially with large datasets.

Hashing a record also helps with:

- **Data Integrity**: Hashing ensures that even a small change in any of the descriptive fields results in a completely different hash. This helps maintain **data integrity** and detect changes or tampering.
- **Data Handling**: Hashes are useful when you need a compact way to reference records without needing to expose or transmit all individual columns. This simplifies **indexing** and referencing trees in other operations.

In summary, hashing provides a compact, efficient, and secure way to uniquely identify and handle records in a database.

Your task

1. Concatenate Categorical Columns

- Write a **SQL query** that concatenates the following columns for each tree: `name`, `domain`, `genre`, `species`, `variety`, `arrondissement`.
  - Use `COALESCE` to replace any `NULL` values with `'UNK'` in the concatenated result.
  - Example structure: `COALESCE(column_name, 'UNK')`.

2. Generate SHA256 Hash

- display the `digest` and `encode` functions definition

You can use `\df+ digest` to find which version of the function you want to inspect. and then

```sql
SELECT pg_catalog.pg_get_functiondef('digest(bytea, text)'::regprocedur
```

To see the definition for `digest(bytea, text)` .

- Extend the SQL query to pass the concatenated string into the `sha256` function.
- The query should return the resulting `sha256` hash string.

3. Create SQL Function for Tree Hash

- Write a SQL function that:
  - Accepts a tree's `id` as input.
  - Returns the `sha256` hash generated from the concatenated columns of that tree.

4. Add `sha_id` Column

- Add a new column `sha_id` of type `text` to the `trees` table .

This column will store the hash values for each tree.

5. Modify Function: Insert Hash Value

- Modify the function from task 3 so that it:
  - Inserts the hash value into the `sha_id` column.
  - Includes exception handling to catch any errors (e.g., inserting a duplicate hash).

6. Test the Function on Several Trees

- Run the modified function for several rows in the database to insert their hash values into the `sha_id` column.

7. Update Function: Handle Subset of Trees

- Extend the function to handle a set of trees, rather than just a single tree.
  - Input could be a list of `id` s or a query that selects a subset of trees.

8. Ensure Uniqueness of Hash Values

- Add a database constraint to ensure that the `sha_id` column only contains unique values.

9. PL/pgSQL Function: Generate Hash for All Trees

- Write a `PL/pgSQL` function that:

  - Runs the `sha256` hash on all rows in the database.
  - Inserts the hash values into the `sha_id` column.
  - Checks for duplicates and ensures that no tree is processed more than once.

Additional Notes

- Make sure to test your functions thoroughly.
- Consider edge cases like:
  - Trees with missing or `NULL` values in categorical columns.
  - Duplicates in `sha_id` values.

- You can use `RAISE NOTICE` statements within PL/pgSQL to help with debugging.