User-Defined Data Types

- PostgreSQL provides two methods to implement user-defined data types, via the following commands:
  - `CREATE DOMAIN`: The `CREATE DOMAIN` command allows developers to create a user-defined data type with constraints. This helps to make the source code more modular.
  - `CREATE TYPE`: The `CREATE TYPE` command is often used to create a composite type, which is useful in procedural languages, and is used as the return data type. Also, we can use the `CREATE TYPE` to create the `ENUM` type, which is useful to decrease the number of joins, specifically for lookup tables.

- Often, developers decide not to use user-defined data types and to use flat tables instead, due to a lack of support on the driver side, such as JDBC and ODBC.
- Nonetheless, in JDBC, the composite data types can be retried as Java objects and parsed manually.

- Domain objects, as with other database objects, should have a unique name within the schema scope.
- The first use case of domains is to use them for common patterns.
- For example, a text type that does not allow for null values and does not contain spaces is a common pattern.
- In the web car portal, the `first_name` and `last_name` columns in the account table are not null.
- They also should not contain spaces, and they can be defined as follows:
  - ```
    first_name TEXT NOT NULL,
    last_name TEXT NOT NULL,
    CHECK(first_name !~ '\s' AND last_name !~ '\s'),
    ```
    ```

- You can replace the text data type and the constraints by creating a domain and using it to define the `first_name` and `last_name` data types, as follows:

```
car_portal=> CREATE DOMAIN text_without_space_and_null AS TEXT NOT NULL CHECK (value !~ '\s');
CREATE DOMAIN
```

- In order to test `text_without_space_and_null` domain, let's use it in a table definition and execute several `INSERT` statements, as follows:

```
car_portal=> CREATE TABLE test_domain (
car_portal(>   test_att text_without_space_and_null
car_portal(> );
CREATE TABLE
car_portal=> INSERT INTO test_domain values ('hello');
INSERT 0 1
car_portal=> INSERT INTO test_domain values ('hello world');
ERROR:  value for domain text_without_space_and_null violates check constraint "text_without_space_and_null_check"
car_portal=> INSERT INTO test_domain values (null);
ERROR:  domain text_without_space_and_null does not allow null values
```

- Another good use case for creating domains is to create distinct identifiers across several tables, since some people tend to use numbers instead of names to retrieve information.
- You can do this by creating a sequence and wrapping it with a domain, as follows:

```
car_portal=> CREATE SEQUENCE global_id_seq;
CREATE SEQUENCE
car_portal=> CREATE DOMAIN global_serial INT DEFAULT NEXTVAL('global_id_seq') NOT NULL;
CREATE DOMAIN
```

- Finally, you can alter the domain by using the `ALTER DOMAIN` command.
- If a new constraint is added to a domain, it will cause all of the attributes using that domain to be validated against the new constraint.
- You can control this by suppressing the constraint validation on old values and then cleaning up the tables individually.
- For example, let's suppose that we would like to have a constraint on the text length of `text_without_space_and_null` domain; this can be done as follows:

```
car_portal=> INSERT INTO test_domain values ('abcdefghijklmnopqrstuvwxyz');
INSERT 0 1
car_portal=> ALTER DOMAIN text_without_space_and_null
car_portal-> ADD CONSTRAINT text_without_space_and_null_length_chk check (length(value)<=15);
ERROR:  column "test_att" of table "test_domain" contains values that violate the new constraint
```

- The preceding SQL statement will fail due to a data violation if an attribute is using this domain and the attribute value length is more than 15 characters.

- So, to force the newly created data to adhere to the domain constraints and to leave the old data without validation, you can still create it, as follows:

```
car_portal=> ALTER DOMAIN text_without_space_and_null
car_portal-> ADD CONSTRAINT text_without_space_and_null_length_chk check (length(value)<=15) NOT VALID;
ALTER DOMAIN
```

- After data cleanup, you can also validate the constraint for old data by invoking the `ALTER DOMAIN ... VALIDATE CONSTRAINT` statement.

```
car_portal=> UPDATE test_domain
car_portal-> SET test_att = 'abcd'
car_portal-> WHERE test_att = 'abcdefghijklmnopqrstuvwxyz';
UPDATE 1
car_portal=> ALTER DOMAIN text_without_space_and_null
car_portal-> VALIDATE CONSTRAINT text_without_space_and_null_length_chk;
ALTER DOMAIN
```

- Finally, the `\dD+` psql metacommand can be used to describe the domain.

```
car_portal=> \dD+
```

```
                                                                                                   List of domains
 Schema |            Name             |  Type   | Collation | Nullable |                Default                 |                       Check
                         | Access privileges | Description
--------+-----------------------------+---------+-----------+----------+----------------------------------------+-----------------------------------
-------------------------+-------------------+-------------
 public | global_serial               | integer |           | not null | nextval('global_id_seq'::regclass) |
                         |                   |
 public | text_without_space_and_null | text    |           | not null |                                        | CHECK (VALUE !~ '\s'::text) CHEC
K (length(VALUE) <= 15) |                   |
(2 rows)
```

- Composite data types are very useful for creating functions, especially when the return type is a row of several values.
- For example, let's suppose that we would like to have a function that returns `seller_id`, `seller_name`, the number of advertisements, and the total rank for a certain customer account.
- The first step is to create `TYPE`, as follows:

```
car_portal=# CREATE TYPE car_portal_app.seller_information AS
car_portal-# (seller_id INT, seller_name TEXT,number_of_advertisements BIGINT, total_rank float);
CREATE TYPE
```

- Then, we can use the newly created data `TYPE` as the return type of the function, as follows:

```
car_portal=# CREATE OR REPLACE FUNCTION car_portal_app.seller_information (account_id INT ) RETURNS car_portal_app.seller_information AS $$
car_portal$# SELECT seller_account.seller_account_id, first_name || last_name as seller_name, count(*), sum(rank)::float/count(*)
car_portal$# FROM car_portal_app.account
car_portal$#      INNER JOIN  car_portal_app.seller_account ON account.account_id = seller_account.account_id
car_portal$#      LEFT JOIN  car_portal_app.advertisement ON advertisement.seller_account_id = seller_account.seller_account_id
car_portal$#      LEFT JOIN  car_portal_app.advertisement_rating ON advertisement.advertisement_id = advertisement_rating.advertisement_id
car_portal$# WHERE account.account_id = $1
car_portal$# GROUP BY seller_account.seller_account_id, first_name, last_name
car_portal$# $$
car_portal-# LANGUAGE SQL;
CREATE FUNCTION
```

```
car_portal=# SELECT (car_portal_app.seller_information(35)).*;
 seller_id | seller_name  | number_of_advertisements | total_rank
-----------+--------------+--------------------------+------------
         1 | AlishiaSergi |                        1 |
(1 row)
```

- `CREATE TYPE` can also be used to define `ENUM`; an `ENUM` type is a special data type that enables an attribute to be assigned one of the predefined constants.
- The usage of the `ENUM` data types reduces the number of joins needed to create some queries; hence, it makes SQL code more compact and easier to understand.

- In the `advertisement_rating` table, we have a column with the rank name, which is defined as follows:
  - `-- This is a part of advertisement_rating table def.`
    ```
    rank INT NOT NULL,
    CHECK (rank IN (1,2,3,4,5)),
    ```
- In the preceding example, the given code is not semantically clear.
- For example, some people might consider `1` to be the highest rank, while others might consider `5` to be the highest rank.

- To solve this, you can use the lookup table, as follows:
  - ```
    CREATE TABLE rank (
        rank_id SERIAL PRIMARY KEY,
        rank_name TEXT NOT NULL
    );
    INSERT INTO rank
    VALUES (1, 'poor') , (2, 'fair'), (3, 'good') ,
    (4,'very good') ,( 5, 'excellent');
    ```
    ```

- In the preceding approach, the user can explicitly see the rank table entries.
- Moreover, the rank table entries can be changed to reflect new business needs (for example, to make the ranking from `1` to `10`).
- Additionally, in this approach, changing the rank table entries will not lock the `advertisement_rating` table, since the `ALTER TABLE` command will not be needed to change the check constraint, `CHECK (rank IN (1, 2, 3, 4, 5))`.
- The disadvantage of this approach lies in retrieving the information of a certain table that's linked to several lookup tables, since the tables need to be joined together.
- In our example, you need to join `advertisement_rating` and the `rank` table to get the semantics of `rank_id`.
- The more lookup tables, the more lengthy the queries are.

- Another approach to modeling the rank is to use the `ENUM` data types, as follows:

```
car_portal=# CREATE TYPE rank AS ENUM ('poor', 'fair', 'good', 'very good','excellent');
CREATE TYPE
```

- The psql `\dT` meta-command is used to describe the `ENUM` data type.

```
car_portal=# \dT
                List of data types
 Schema |            Name             | Description
--------+-----------------------------+-------------
 public | global_serial               |
 public | rank                        |
 public | text_without_space_and_null |
(3 rows)
```

- You can also use the `enum_range` function, as follows:

```
car_portal=# SELECT enum_range(null::rank);
               enum_range
-----------------------------------------
 {poor,fair,good,"very good",excellent}
(1 row)
```

- The ENUM data type order is determined by the order of the values in ENUM at the time of its creation:

```
car_portal=# SELECT unnest(enum_range(null::rank)) order by 1 desc;
  unnest
-----------
 excellent
 very good
 good
 fair
 poor
(5 rows)
```

- The `ENUM` PostgreSQL data types are type-safe, and different `ENUM` data types cannot be compared with each other.
- Moreover, `ENUM` data types can be altered and new values can be added.
- Note that adding a new value to `ENUM` might lock user activities, since this operation is blocking.

```
car_portal=# CREATE TABLE article_test (
car_portal(#   article_id INT,
car_portal(#   rating rank
car_portal(# );
CREATE TABLE
car_portal=# INSERT INTO article_test(article_id, rating) VALUES (1, 'poor');
INSERT 0 1
car_portal=# ALTER TYPE rank ADD VALUE 'exceptional' AFTER 'excellent';
ALTER TYPE
car_portal=# INSERT INTO article_test(article_id, rating) VALUES (2, 'exceptional');
INSERT 0 1
car_portal=# SELECT   *
car_portal-# FROM     article_test
car_portal-# ORDER BY rating DESC;
 article_id |   rating
------------+-------------
          2 | exceptional
          1 | poor
(2 rows)
```