Dynamic SQL

- **Dynamic SQL** is used to build and execute queries on the fly.
- Unlike a static SQL statement, a dynamic SQL statement's full text is unknown and can change between successive executions.
- These queries can be **DDL**, **DCL**, and **DML** statements.
- Dynamic SQL is used to reduce repetitive tasks.
  - For example, you could use dynamic SQL to create table partitioning for a certain table on a daily basis, to add missing indexes to all foreign keys or to add data auditing capabilities to a certain table, without major coding effects.
- Another important use of dynamic SQL is to overcome the side effects of PL/pgSQL caching, as queries executed using the `EXECUTE` statement are not cached.

- Dynamic SQL is achieved via the `EXECUTE` statement.
- The `EXECUTE` statement accepts a string, and simply evaluates it.
- The synopsis to execute a statement is as follows:

```
EXECUTE command-string [ INTO [STRICT] target ]
[ USING expression [, ...] ];
```

- Dynamic SQL is a very sharp knife, and should be treated carefully.
- Dynamic SQL, if not written carefully, can expose the database to SQL injection attacks.

Executing DDL statements in dynamic SQL

- In some cases, you will need to perform operations at the database object level, such as tables, indexes, columns, and roles.
- For example, suppose that a database developer would like to vacuum and analyze a specific schema object, which is a common task after the deployment, in order to update the statistics.

- To analyze the `car_portal_app` schema tables, we could write the following script:

```
car_portal=> DO $$
car_portal$> DECLARE
car_portal$>   table_name text;
car_portal$> BEGIN
car_portal$>   FOR table_name IN SELECT tablename FROM pg_tables WHERE schemaname ='car_portal_app' LOOP
car_portal$>     RAISE NOTICE 'Analyzing %', table_name;
car_portal$>     EXECUTE 'ANALYZE car_portal_app.' || table_name;
car_portal$>   END LOOP;
car_portal$> END;
car_portal$> $$;
NOTICE:  Analyzing account
NOTICE:  Analyzing account_history
NOTICE:  Analyzing advertisement
NOTICE:  Analyzing seller_account
NOTICE:  Analyzing car_model
NOTICE:  Analyzing car
NOTICE:  Analyzing advertisement_picture
NOTICE:  Analyzing advertisement_rating
NOTICE:  Analyzing favorite_advertisement
NOTICE:  Analyzing a
DO
```

Executing DML statements in dynamic SQL

- Some applications might interact with data in an interactive manner.
- For example, we might have billing data generated on a monthly basis.
- Also, some applications filter data on different criteria, as defined by the user.
- In such cases, dynamic SQL is very convenient.

- For example, in the car portal application, the search functionality is needed to get accounts using the dynamic predicate, as follows:

```
car_portal=> CREATE OR REPLACE FUNCTION car_portal_app.get_account (predicate TEXT) RETURNS SETOF car_portal_app.account AS
car_portal-> $$
car_portal$> BEGIN
car_portal$>   RETURN QUERY EXECUTE 'SELECT * FROM car_portal_app.account WHERE ' || predicate;
car_portal$> END;
car_portal$> $$ LANGUAGE plpgsql;
CREATE FUNCTION
car_portal=>
```

- We can test how the function can be used to evaluate several predicates, as follows:

```
car_portal=> SELECT * FROM car_portal_app.get_account ('true') limit 1;
 account_id | first_name | last_name |      email      |             password
------------+------------+-----------+-----------------+----------------------------------
          1 | James      | Butt      | jbutt@gmail.com | 1b9ef408e82e38346e6ebebf2dcc5ece
(1 row)

car_portal=> SELECT * FROM car_portal_app.get_account (E'first_name=\'James\'');
 account_id | first_name | last_name |      email      |             password
------------+------------+-----------+-----------------+----------------------------------
          1 | James      | Butt      | jbutt@gmail.com | 1b9ef408e82e38346e6ebebf2dcc5ece
(1 row)
```

Dynamic SQL and the caching effect

- As we mentioned earlier, PL/pgSQL caches execution plans.
- This is quite good if the generated plan is expected to be static.
- For example, the following statement is expected to use an index scan, due to selectivity.
- In this case, caching the plan saves some time and increases performance:

```
SELECT * FROM account WHERE account_id =<INT>
```

- In other scenarios, however, this is not true.
- For example, let's suppose that we have an index on the `advertisement_date` column, and we would like to get the number of advertisements since a certain date, as follows:

```
SELECT count (*)
FROM   car_portal_app.advertisement
WHERE  advertisement_date >= <certain_date>;
```

- In the preceding query, the entries from the `advertisement` table can be fetched from the hard disk, either by using the index scan or by using the sequential scan based on selectivity, which depends on the provided `certain_date` value.
- Caching the execution plan of such a query will cause serious problems; hence, writing the function as follows is not a good idea:

```
CREATE OR REPLACE FUNCTION car_portal_app.get_advertisement_count
(some_date timestamptz ) RETURNS BIGINT AS $$
BEGIN
  RETURN (SELECT count (*) FROM car_portal_app.advertisement WHERE
advertisement_date >=some_date)::bigint;
END;
$$ LANGUAGE plpgsql;
```

- To solve the caching issue, we could rewrite the previous function, using either the SQL language function or the PL/pgSQL `execute` command, as follows:

```
car_portal=> CREATE OR REPLACE FUNCTION car_portal_app.get_advertisement_count (some_date timestamptz ) RETURNS BIGINT AS $$
car_portal$> DECLARE
car_portal$>   count BIGINT;
car_portal$> BEGIN
car_portal$>   EXECUTE 'SELECT count (*) FROM car_portal_app.advertisement WHERE advertisement_date >= $1' USING some_date INTO count;
car_portal$>   RETURN count;
car_portal$> END;
car_portal$> $$ LANGUAGE plpgsql;
CREATE FUNCTION
car_portal=>
```

Recommended practices for dynamic SQL usage

- Dynamic SQL can cause security issues if it is not handled carefully; dynamic SQL is vulnerable to the SQL injection technique.
- SQL injection is used to execute SQL statements that reveal secure information or even destroy data in a database.

- A very simple example of a PL/pgSQL function that's vulnerable to SQL injection is as follows:

```
car_portal=> CREATE OR REPLACE FUNCTION car_portal_app.can_login (email text, pass text) RETURNS BOOLEAN AS $$
car_portal$> DECLARE
car_portal$>   stmt TEXT;
car_portal$>   result bool;
car_portal$> BEGIN
car_portal$>   stmt = E'SELECT COALESCE (count(*)=1, false) FROM car_portal_app.account WHERE email = \''|| $1 || E'\' and password = \
''||$2||E'\'';
car_portal$>   RAISE NOTICE '%' , stmt;
car_portal$>   EXECUTE stmt INTO result;
car_portal$>   RETURN result;
car_portal$> END;
car_portal$> $$ LANGUAGE plpgsql;
CREATE FUNCTION
car_portal=>
```

- The preceding function returns `true` if the email and the password match.
- To test this function, let's insert a row and try to inject some code, as follows:

```
car_portal=> SELECT car_portal_app.can_login('jbutt@gmail.com', md5('jbutt@gmail.com'));
NOTICE:  SELECT COALESCE (count(*)=1, false) FROM car_portal_app.account WHERE email = 'jbutt@gmail.com' and password = '1b9ef408e82e38
346e6ebebf2dcc5ece'
 can_login
-----------
 t
(1 row)

car_portal=> SELECT car_portal_app.can_login('jbutt@gmail.com', md5('jbutt@yahoo.com'));
NOTICE:  SELECT COALESCE (count(*)=1, false) FROM car_portal_app.account WHERE email = 'jbutt@gmail.com' and password = '37eb43e4d43958
9d274b6f921b1e4a0d'
 can_login
-----------
 f
(1 row)

car_portal=> SELECT car_portal_app.can_login(E'jbutt@gmail.com\'--', 'Do not know password');
NOTICE:  SELECT COALESCE (count(*)=1, false) FROM car_portal_app.account WHERE email = 'jbutt@gmail.com'--' and password = 'Do not know
 password'
 can_login
-----------
 t
(1 row)
```

- Notice that the function returns `true` even when the password does not match the password stored in the table.
- This is simply because the predicate was commented, as shown by the `raise` notice:

```
SELECT COALESCE (count(*)=1, false) FROM
account WHERE email = 'jbutt@gmail.com'--' and
password = 'Do not know password'
```

- To protect code against this technique, follow these practices:
  - For parameterized dynamic SQL statements, use the `USING` clause.
  - Use the `format` function with the appropriate interpolation to construct your queries. Note that `%I` escapes the argument as an identifier and `%L` as a literal.
  - Use `quote_ident()`, `quote_literal()`, and `quote_nullable()` to properly format your identifiers and literal.

- One way to write the preceding function is as follows:

```
car_portal=> CREATE OR REPLACE FUNCTION car_portal_app.can_login (email text, pass text) RETURNS BOOLEAN AS
car_portal-> $$
car_portal$> DECLARE
car_portal$>   stmt TEXT;
car_portal$>   result bool;
car_portal$> BEGIN
car_portal$>   stmt = format('SELECT COALESCE (count(*)=1, false) FROM car_portal_app.account WHERE email = %L and password = %L', $1,
$2);
car_portal$>   RAISE NOTICE '%' , stmt;
car_portal$>   EXECUTE stmt INTO result;
car_portal$>   RETURN result;
car_portal$> END;
car_portal$> $$ LANGUAGE plpgsql;
CREATE FUNCTION
car_portal=> SELECT car_portal_app.can_login(E'jbutt@gmail.com\'--', 'Do not know password');
NOTICE:  SELECT COALESCE (count(*)=1, false) FROM car_portal_app.account WHERE email = 'jbutt@gmail.com''--' and password = 'Do not kno
w password'
 can_login
-----------
 f
(1 row)
```