Views

- A view can be considered a named query or a wrapper around a SELECT statement.
- Views can be used for the following purposes:
  - Simplifying complex queries and increasing code modularity
  - Tuning performance by caching the view results for later use
  - Decreasing the amount of SQL code
  - Bridging the gap between relational databases and object-oriented languages, especially updatable views
  - Implementing authorization at the row level, by leaving out rows that do not meet a certain predicate
  - Implementing interfaces and the abstraction layer between high-level languages and relational databases
  - Implementing last-minute changes

- A view should meet the current business needs, instead of potential future business needs.
- It should be designed to provide certain functionality or service.
- Note that the more attributes there are in a view, the more effort will be required to refactor the view.
- In addition to that, when a view aggregates data from many tables and is used as an interface, there might be a degradation in performance, due to many factors (for example, bad execution plans due to outdated statistics for some tables, execution plan time generation, and so on).

- When implementing complex business logic in a database using views and stored procedures, database refactoring, especially for base tables, might turn out to be very expensive.
- To solve this issue, consider migrating the business logic to the application business tier.

- Some frameworks, such as object-relational mappers, might have specific needs, such as a unique key.
- This limits the usage of views in these frameworks; however, we can mitigate these issues by faking the primary keys, via window functions such as `row_number`.

- In PostgreSQL, a view is internally modeled as a table with an `_RETURN` rule.
- So, in theory, we can create a table and convert it into a view.
- However, this is not a recommended practice.

- The `VIEW` dependency tree is well maintained; this means that we cannot drop a view or amend its structure if another view depends on it, as follows:

```
postgres=# CREATE VIEW test AS SELECT 1 as v;
CREATE VIEW
postgres=# CREATE VIEW test2 AS SELECT v FROM test;
CREATE VIEW
postgres=# CREATE OR REPLACE VIEW test AS SELECT 1 as val;
ERROR:  cannot change name of view column "v" to "val"
```

View synopsis

- In the following view synopsis, the `CREATE VIEW` statement is used to create a view; if the `REPLACE` keyword is used, the view will be replaced (if it already exists).
- View attribute names can be given explicitly or they can be inherited from the `SELECT` statement:
-
```
CREATE [ OR REPLACE ] [ TEMP | TEMPORARY ] [ RECURSIVE ] VIEW name
[ ( column_name [, ...] ) ]
    [ WITH ( view_option_name [= view_option_value] [, ... ] ) ]
    AS query    [ WITH [ CASCADED | LOCAL ] CHECK OPTION ]
```

- The following example shows how to create a view that only lists the user information, without the password.
- This might be useful for implementing data authorization to restrict applications from accessing the password.
- Note that the view column names are inherited from the `SELECT` list, as shown by the `\d` meta-command:

```
postgres=# \c car_portal
You are now connected to database "car_portal" as user "postgres".
car_portal=# SET search_path to car_portal_app;
SET
car_portal=# SET role car_portal_app;
SET
car_portal=> CREATE VIEW car_portal_app.account_information AS
car_portal->  SELECT account_id, first_name, last_name, email FROM car_portal_app.account;
CREATE VIEW
car_portal=> \d car_portal_app.account_information
       View "car_portal_app.account_information"
   Column   |  Type   | Collation | Nullable | Default
------------+---------+-----------+----------+---------
 account_id | integer |           |          |
 first_name | text    |           |          |
 last_name  | text    |           |          |
 email      | text    |           |          |
```

- The view column names can be assigned explicitly, as shown in the following example.
- This might be useful when we need to change the view column names:

```
car_portal=> CREATE OR REPLACE VIEW car_portal_app.account_information (account_id, first_name, last_name, email)
car_portal->  AS SELECT account_id, first_name, last_name, email FROM car_portal_app.account;
CREATE VIEW
```

- When replacing the view definition using the `REPLACE` keyword, the column list should be identical before and after the replacement, including the column type, name, and order.
  - The new query may add additional columns to the end of the list.
  - The calculations giving rise to the output columns may be completely different.
- The following example shows what happens when you try to change the view column order:

```
car_portal=> CREATE OR REPLACE VIEW account_information AS
car_portal->  SELECT account_id, last_name, first_name, email FROM car_portal_app.account;
ERROR:  cannot change name of view column "first_name" to "last_name"
```

```
car_portal=> drop view car_portal_app.account_information;
DROP VIEW
car_portal=> CREATE OR REPLACE VIEW car_portal_app.account_information AS
car_portal->  SELECT account_id, last_name, first_name, email FROM car_portal_app.account;
CREATE VIEW
```

View Categories

- Views in PostgreSQL can be categorized into one of the following categories on the basis of their uses:
  - **Temporary views**: A temporary view is automatically dropped at the end of a user session. If the `TEMPORARY` or `TEMP` keywords are not used, then the life cycle of the view starts with the view creation and ends with the action of dropping it.
  - **Recursive views**: A recursive view is similar to the recursive functions in high-level languages. The view column list should be specified in recursive views. The recursion in relational databases, such as in recursive views or recursive **Common Table Expressions (CTEs)**, can be used to write very complex queries, specifically for hierarchical data.
  - **Updatable views**: Updatable views allow the user to see the view as a table. This means that the developer can perform `INSERT`, `UPDATE`, and `DELETE` operations on views, similar to tables. Updatable views can help to bridge the gap between an object model and a relational model (to some extent), and they can help to overcome problems such as polymorphism.
  - **Materialized views**: A materialized view is a table whose contents are periodically refreshed, based on a certain query. Materialized views are useful for boosting the performance of queries that require a longer execution time and are executed frequently on static data. We could perceive materialized views as a caching technique.

Materialized Views

- The materialized view synopsis differs a little bit from the normal view synopsis.
- Materialized views are a PostgreSQL extension, but several databases, such as Oracle, support them.

- As shown in the following synopsis, a materialized view can be created in a certain `TABLESPACE`, as well as a `storage_parameter`, which is logical, since materialized views are physical objects:
  - `CREATE MATERIALIZED VIEW [ IF NOT EXISTS ] table_name`
    `[ (column_name [, ...] ) ]`
    `[ WITH ( storage_parameter [= value] [, ... ] ) ]`
    `[ TABLESPACE tablespace_name ]`
    `AS query [ WITH [ NO ] DATA ]`

- At the time of the creation of a materialized view, it can be populated with data or left empty.
- If it is not populated, retrieving data from the unpopulated materialized view will raise `ERROR`.
- The `REFRESH MATERIALIZED VIEW` statement can be used to populate a materialized view.
- The synopsis for the `REFRESH` command is as follows:
  - `REFRESH MATERIALIZED VIEW [ CONCURRENTLY ] name [ WITH [ NO ] DATA ]`

- The following example shows an attempt to retrieve data from an unpopulated materialized view:

```
car_portal=> CREATE MATERIALIZED VIEW test_mat AS SELECT 1 WITH NO DATA;
CREATE MATERIALIZED VIEW
car_portal=> TABLE test_mat;
ERROR:  materialized view "test_mat" has not been populated
HINT:  Use the REFRESH MATERIALIZED VIEW command.
```

- To refresh the view, as the hint suggested, need to refresh the view, as follows:

```
car_portal=> REFRESH MATERIALIZED VIEW test_mat;
REFRESH MATERIALIZED VIEW
car_portal=> TABLE test_mat;
 ?column?
----------
        1
(1 row)
```

- Refreshing materialized view is a blocking statement; this means that concurrent selects will be blocked for a specific time period until the refresh is done.
- This can be solved by refreshing the materialized view concurrently.
- To be able to use this option, the materialized view should have a unique index.

- Materialized views are often used with **data warehousing**.
- In data warehousing, several queries are required for business analysis and decision support.
- The data in these kinds of applications does not usually change, but the calculation and aggregation of that data is often a costly operation.
- In general, a materialized view can be used for the following:
  - Generating summary reports
  - Caching the results of recurring queries
  - Optimizing performance by processing data only once

- Since materialized views are tables, they can also be indexed, leading to a great performance boost.

Updatable Views

- By default, simple PostgreSQL views are auto-updatable.
- **Auto-updatable** means that you can use the view with the `DELETE`, `INSERT`, and `UPDATE` statements, in order to manipulate the data of the underlying table.
- If the view is not updatable (which is not simple) due to the violation of one of the following constraints, the trigger and rule systems can be used to make it updatable.

- The view is automatically updatable if the following conditions are met:
  - The view must be built on top of one table or an updatable view.
  - The view definition must not contain the following clauses and set operators at the top level: `DISTINCT`, `WITH`, `GROUP BY`, `OFFSET`, `HAVING`, `LIMIT`, `UNION`, `EXCEPT`, and `INTERSECT`.
  - The view's select list must be mapped to the underlying table directly, without using functions and expressions. Moreover, the columns in the select list shouldn't be repeated.
  - The `security_barrier` property must not be set. The preceding conditions promise that the view attributes can be mapped directly to the underlying table attributes.

- In the web car portal, let's assume that we have an updatable view that only shows the accounts that are not seller accounts, as follows:

```
car_portal=> CREATE VIEW car_portal_app.user_account AS
car_portal->  SELECT account_id, first_name, last_name, email, password
car_portal->  FROM car_portal_app.account
car_portal->  WHERE account_id NOT IN (SELECT account_id FROM car_portal_app.seller_account);
CREATE VIEW
```

- To test the `user_account` view, let's insert a row, as follows:

```
car_portal=> INSERT INTO car_portal_app.user_account
car_portal->  VALUES (default,'first_name1','last_name1','test@email.com','password');
INSERT 0 1
```

- In the case of an auto-updatable view, we cannot modify data that isn't returned by the view.
- For example, let's insert an account with a seller account and then try to delete it, as follows:

```
car_portal=> WITH account_info AS (
car_portal(>  INSERT INTO car_portal_app.user_account
car_portal(>  VALUES (default,'first_name2','last_name2','test2@email.com','password')
car_portal(>  RETURNING account_id)
car_portal-> INSERT INTO  car_portal_app.seller_account (account_id, street_name, street_number, zip_code, city)
car_portal->  SELECT account_id, 'street1', '555', '555', 'test_city'
car_portal->  FROM account_info;
INSERT 0 1
```

- In the preceding example, notice that the `INSERT` command to the user account was successful.
- However, even though the account was inserted successfully, we cannot delete it using the updatable view, since the check constraint will be effective, as follows:

```
car_portal=> DELETE FROM car_portal_app.user_account WHERE first_name = 'first_name2';
DELETE 0
car_portal=> SELECT * FROM account where first_name like 'first_name%';
 account_id | first_name  | last_name  |      email      | password
------------+-------------+------------+-----------------+----------
        482 | first_name1 | last_name1 | test@email.com  | password
        483 | first_name2 | last_name2 | test2@email.com | password
(2 rows)
```

- The `WITH CHECK OPTION` view is used to control the behavior of automatically updatable views.
- If `WITH CHECK OPTION` is not specified, we can `UPDATE` or `INSERT` a record, even if it is not visible in the view, which might cause a security risk.

- To demonstrate `WITH CHECK OPTION`, let's create a table, as follows:

```
car_portal=> CREATE TABLE check_option (val INT);
CREATE TABLE
car_portal=> CREATE VIEW test_check_option
car_portal->  AS SELECT *
car_portal->  FROM check_option
car_portal->  WHERE val > 0
car_portal->  WITH CHECK OPTION;
CREATE VIEW
```

- To test `WITH CHECK OPTION`, let's insert a row that violates the `check` condition, as follows:

```
car_portal=> INSERT INTO test_check_option VALUES (-1);
ERROR:  new row violates check option for view "test_check_option"
DETAIL:  Failing row contains (-1).
```

- If you are uncertain whether a view is auto-updatable, you can verify this information by using `information_schema`, checking the value of the `is_insertable_into` flag, as follows:

```
car_portal=> SELECT table_name, is_insertable_into
car_portal-> FROM   information_schema.tables
car_portal-> WHERE  table_name = 'user_account';
  table_name  | is_insertable_into
--------------+--------------------
 user_account | YES
(1 row)
```