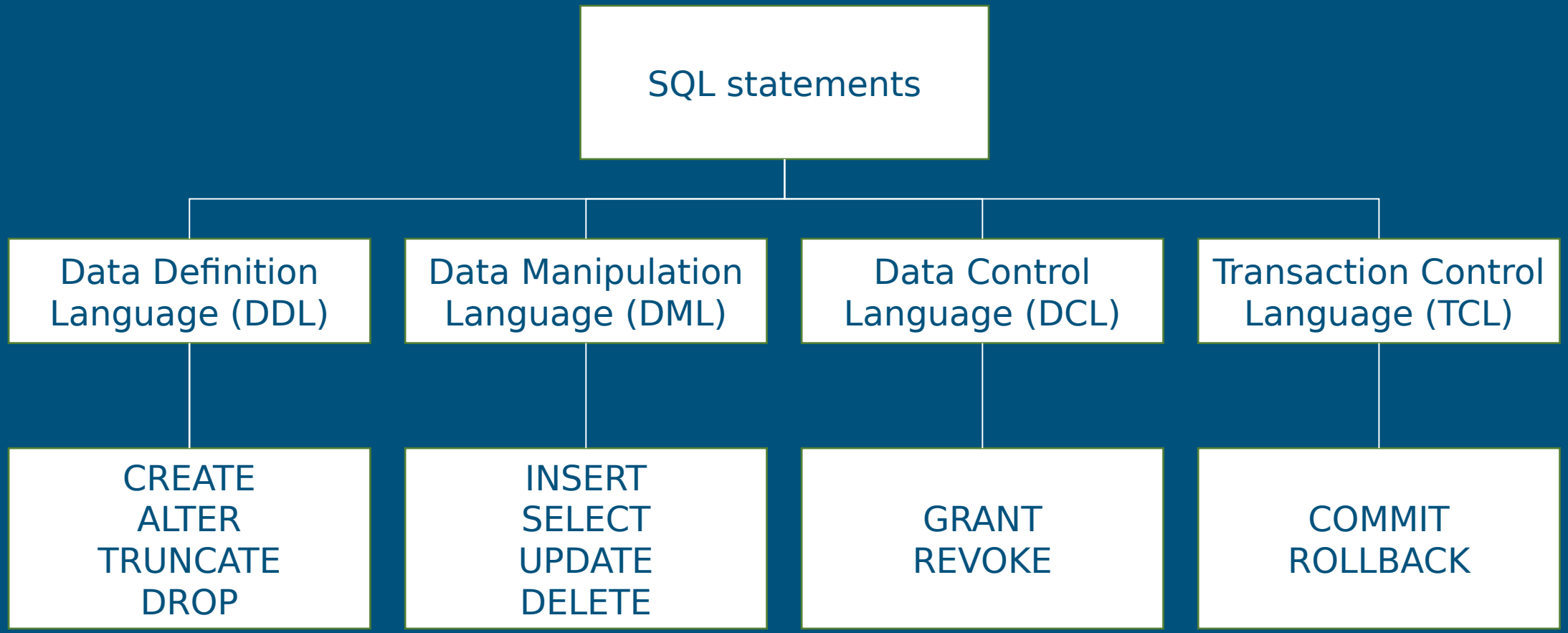
Databases

Lecture 7
DCL, Advanced SQL(CTE, recursion, window functions)

I. DCL

Groups of SQL operators



Why do we need DCL?

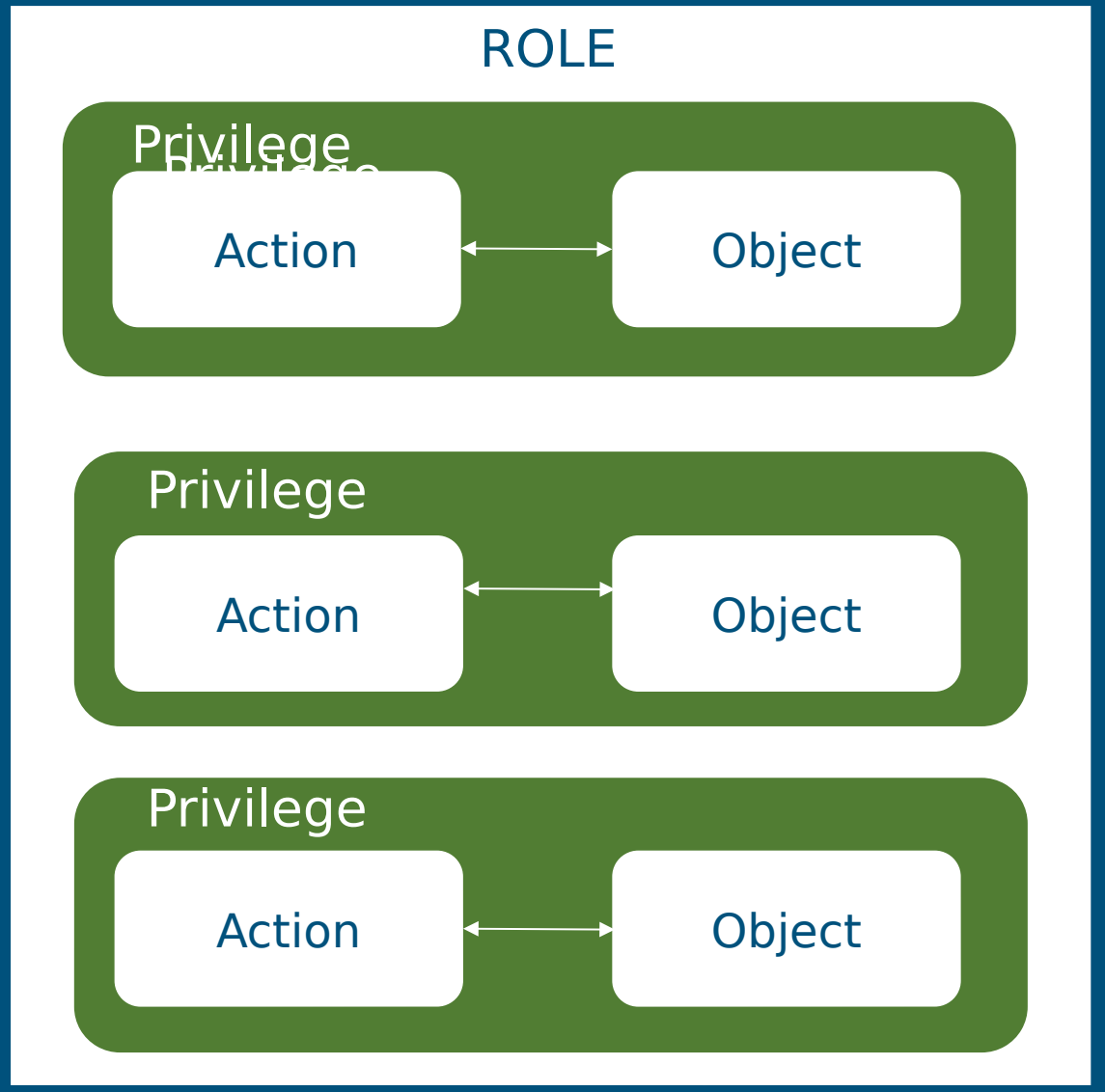
- It is a subset of the SQL language that is used to manage data access rights and user rights (roles) in databases
- In turn, this is necessary to ensure data security
- In Postgres, the main data security model is the role based access control (RBAC) model.
- An alternative model has also been partially implemented and supported – the attribute based access control (ABAC) model.

RBAC

- PostgreSQL uses the concept of roles to manage database access permissions.
- A role can be thought of as a database user or a group of users, depending on how the role is configured. Roles can own database objects and grant other roles permission to access those objects, controlling who has access to which objects. You can also grant one role membership in another role, so one role can use the rights of other roles.

RBAC


USER


ROLE
Privilege
Action
Object
Privilege
Action
Object
Privilege
Action
Object

RBAC: the main abstractions

- the concepts of **object** and **action** are proposed to differentiate the capabilities of different users. Actions can be either associated with an object (for example, methods of an object) or with a class of objects, or not associated
- Any user who creates an object becomes its owner. The owner of any object has the right to perform any actions associated with that object, and can grant (possibly limited) access rights to his objects and actions on them to other users. The rights to access objects and use actions are called **privileges**.
- The concept of a **role** is proposed to make it easier to manage the transfer of privileges to users. Each role is given the privileges necessary to perform all operations associated with that role.
- **Each user receives the right (privilege) to perform a certain role or several roles.**

RBAC in Postgres

Basic Postgres objects:

- tables
- columns
- representation
- sequences
- Database
- functions
- procedures
- scheme
- tablespaces

DCL – the syntax

- CREATE ROLE name [[WITH] parameter [...]]
  - CREATE USER davide WITH PASSWORD 'jw8s0F4’;
- The role parameter determines its powers and interaction with the client authentication system
- ALTER ROLE is used for changing the role attributes, DROP ROLE – for deleting the role. All the attributes described in CREATE ROLE can be changed later with the commands ALTER ROLE.
- Command CREATE USER is now a synonym of CREATE ROLE

DCL – syntax

```
GRANT { { SELECT | INSERT | UPDATE | DELETE |
TRUNCATE | REFERENCES | TRIGGER }
    [, ...] | ALL [ PRIVILEGES ] }
    ON { [ TABLE ] table_name [, ...]
         | ALL TABLES IN SCHEMA schema_name [, ...] }
    TO role_identifier [, ...] [ WITH GRANT OPTION ]
    [ GRANTED BY role_identifier ]
```

DCL – syntax

- Command GRANT has two main varieties:
  - the first one assigns rights to access database objects
  - the second one assigns some roles as members of others
- If WITH GRANT OPTION is specified, then the grantee of the right can grant it to others. Without this instruction he will not be able to use his right
- Examples:
  - GRANT INSERT ON films TO PUBLIC;
  - GRANT ALL PRIVILEGES ON kinds TO manuel;
  - Grant role «admins» to user joe:
    - GRANT admins TO joe;

DCL – syntax

```
REVOKE [ GRANT OPTION FOR ]
    { { SELECT | INSERT | UPDATE | DELETE | TRUNCATE |
REFERENCES | TRIGGER }
    [, ...] | ALL [ PRIVILEGES ] }
    ON { [ TABLE ] table_name [, ...]
         | ALL TABLES IN SCHEMA schema_name [, ...] }
    FROM role_identifier [, ...]
    [ GRANTED BY role_identifier ]
    [ CASCADE | RESTRICT ]
```

DCL – synatx

- The REVOKE command revokes one or more roles' rights that were previously assigned
- If GRANT OPTION FOR is specified, only the right to transfer the right is revoked, not the right itself. Without this instruction, both the right and the right to dispose of it are revoked.
- Examples:
  - REVOKE INSERT ON films FROM PUBLIC;
  - REVOKE ALL PRIVILEGES ON kinds FROM manuel;
  - REVOKE admins FROM joe;

II. CTE

CTE – common table expressions

- The WITH clause provides a way to write additional statements for use in larger queries. In particular, the main purpose of SELECT in a WITH clause is to **break complex queries with subqueries into simpler parts**
- These statements, also called Common Table Expressions (CTE), can be thought of as temporary table definitions that exist for only one query.
- The additional statement in the WITH clause can be SELECT, INSERT, UPDATE, or DELETE
- The WITH clause itself is attached to a main statement, which can be a SELECT, INSERT, UPDATE, DELETE, or MERGE.

CTE – common table expressions

- **Important!** The order of executing a query with WITH: first, all additional queries “within” WITH are executed, only then the FROM clause in the main statement begins to be executed. Temporary tables are created if necessary when WITH is executed, then they can be accessed in FROM
- According to the SQL standard, queries containing CTEs must be executed as if each CTE had been evaluated once. In the PostgreSQL system, this was implemented literally: all CTEs are executed as separate queries, the result is materialized (written) into temporary memory and then used when executing the entire query containing the CTE

CTE – syntax

- [WITH [RECURSIVE] query_WITH [, ...]]
  SELECT...

- Where **query_WITH**:
  - name_of_query_WITH [(column_name [, ...])] AS [[NOT] MATERIALIZED] (data | values | insert | update | delete)

CTE – Example

```
WITH regional_sales AS (
    SELECT region, SUM(amount) AS total_sales
    FROM orders
    GROUP BY region
), top_regions AS (
    SELECT region
    FROM regional_sales
    WHERE total_sales > (SELECT SUM(total_sales)/10 FROM regional_sales)
)
```

```
SELECT region,
       product,
       SUM(quantity) AS product_units,
       SUM(amount) AS product_sales
FROM orders
WHERE region IN (SELECT region FROM top_regions)
GROUP BY region, product;
```

CTE – Example

- The query displays sales totals only for leading regions
- The WITH clause defines two additional operators regional_sales and top_regions so that the result of regional_sales is used in top_regions and the result of top_regions is used in the main SELECT query
- This example could be rewritten without WITH, but then we would need two levels of nested SELECT subqueries

```
WITH regional_sales AS (
    SELECT region, SUM(amount) AS total_sales
    FROM orders
    GROUP BY region
), top_regions AS (
    SELECT region
    FROM regional_sales
    WHERE total_sales > (SELECT
SUM(total_sales)/10 FROM regional_sales)
)
SELECT region,
        product,
        SUM(quantity) AS product_units,
        SUM(amount) AS product_sales
FROM orders
WHERE region IN (SELECT region FROM
top_regions)
GROUP BY region, product;
```

III. Recursion

Recursion

- A special variant of CTE are “recursive” queries
- In fact, there is no recursion, iteration occurs. (Research in the 80s, 90s gave the following result: recursive queries cannot be expressed within relational languages)
- Syntax:

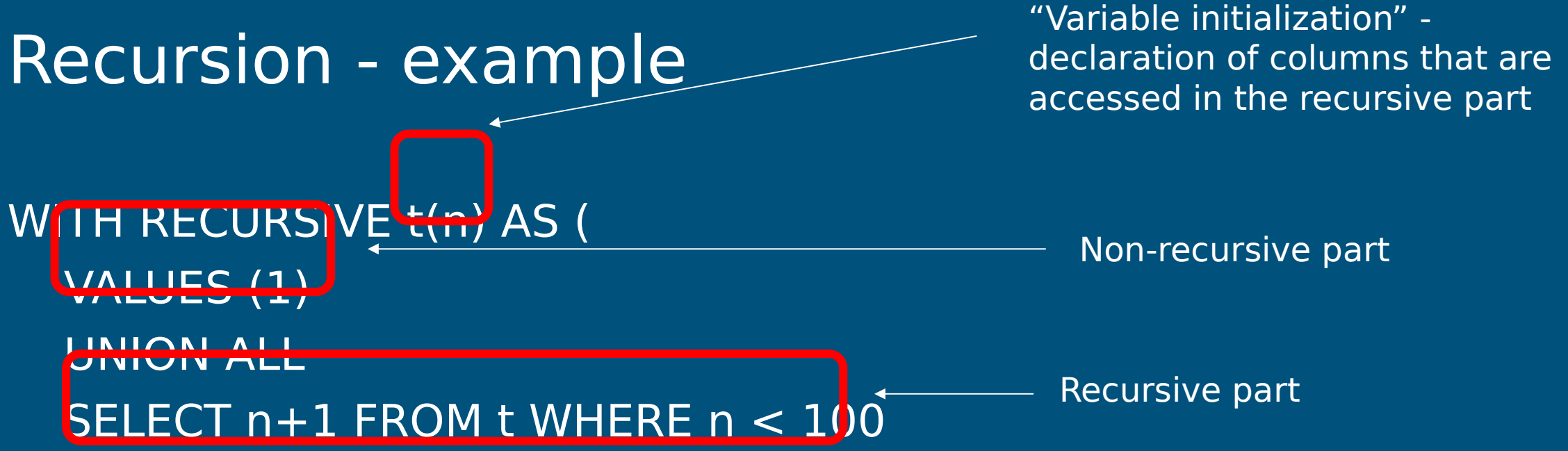
  [WITH [***RECURSIVE***] query_WITH [, ...]]

  SELECT...
  - Query WITH can use its own result by using RECURSIVE
  - What are they needed for: for example, to search for data that has a hierarchical organization, but is stored in the form of a table

Recursion - example

```
WITH RECURSIVE t(n) AS (
    VALUES (1)
    UNION ALL
    SELECT n+1 FROM t WHERE n < 100
)
SELECT sum(n) FROM t;
```

Recursion - example



Recursion - explanation

```
WITH RECURSIVE t(n) AS (
    VALUES (1)
  UNION ALL
    SELECT n+1 FROM t WHERE n <
100
)
SELECT sum(n) FROM t;
```

1. The non-recursive part is calculated. For UNION (but not UNION ALL), duplicate rows are discarded. All remaining rows are included in the result of the recursive query and are also placed in a temporary work table.
   (The number of columns in the WITH RECURSIVE t(...), VALUES (in this case) clause and in the recursive part must match!)
2. As long as the worktable is not empty, the following steps are repeated:
   - The recursive part is evaluated so that the recursive reference to the query itself accesses the current contents of the worktable. For UNION (but not UNION ALL), duplicate rows and rows that duplicate previously received ones are discarded. Any remaining rows are included in the result of the recursive query and are also placed in a temporary staging table.
   - The contents of the work table are replaced with the contents of the staging table, and then the staging table is cleared.

Recursion - explanation

```
WITH RECURSIVE t(n) AS (
    VALUES (1)
    UNION ALL
    SELECT n+1 FROM t
WHERE n < 100
)
SELECT...
```

```
CREATE TABLE t (n
INTEGER);
INSERT INTO t VALUES (1);

SELECT n FROM t
        UNION ALL
SELECT n+1 from t
WHERE n < 100;
```

To be completely identical to a recursive query at this stage, we would have to “add” the results into another table, and from it we would then request data for the next SELECT... UNION ALL SELECT...

Recursion: example

```
WITH RECURSIVE t(n) AS (
    VALUES (1)
    UNION ALL
    SELECT n+1 FROM t WHERE n < 100
)
SELECT sum(n) FROM t;
```

Result: 5050

Recursion: example

```
WITH RECURSIVE t(n) AS (
    VALUES (1)
    UNION ALL
    SELECT n+1 FROM t WHERE n < 100
)
SELECT * FROM t;
```

n
1
2
…
99
100

Recursion: more complex example

We have a parts table and we want to count how many parts we need for our product our_product

sub_part	part	quantity
bolt	our_product	4
nut	our_product	4
washer	our_product	8
screw	bolt	2
metal	screw	1
rubber	washer	1

Expected result

sub_part	total_quantity
bolt	4
metal	8
nut	4
rubber	8
screw	8
washer	8

Recursion: more complex example

```
WITH RECURSIVE included_parts(sub_part, part, quantity) AS (
    SELECT sub_part, part, quantity FROM parts WHERE part =
'our_product’
    UNION ALL
    SELECT p.sub_part, p.part, p.quantity * pr.quantity
    FROM included_parts pr, parts p
    WHERE p.part = pr.sub_part
)
SELECT sub_part, SUM(quantity) as total_quantity
FROM included_parts
GROUP BY sub_part
```

V. Views

Views - description

- Views are virtual tables that represent the result of executing SQL queries. The view does not contain the actual data, but only the query definition.
- The view stores the SQL query, allowing it to be accessed as a table. Views can include data derived from one or more tables, and even other views
- Purposes of using views:
  - Abstraction: Views allow you to hide the complexity of SQL queries from end users
  - Security: Views can be used to restrict user access to certain data
  - Logical separation of data: Views can help organize data so that it is presented to the user in the most logical manner without changing the physical structure of the database
- Postgres supports Materialized Views. Unlike conventional ones, they store query results in a physical table, which is updated periodically or manually. This is useful for speeding up complex queries

Views - updatable and non-updatable

- Views can be mutable (**updatable views**) or immutable (**non-updatable**).
- Mutable views allow you to not only perform read operations (SELECT), but also modify data (INSERT, UPDATE, DELETE) through the view, as if these operations were performed directly on the underlying tables.
- By default, "simple" views are **automatically updatable views**. For “complex” views, this effect can be achieved, for example, using INSTEAD OF triggers

Which views are “simple”

- A representation is considered "simple" if it meets the following conditions:
  - The FROM list in a query defining a view must contain exactly one element, and it must be a table or other mutable view.
  - The view definition must not contain WITH, DISTINCT, GROUP BY, HAVING, LIMIT, or OFFSET clauses (at the top level of the query).
  - The view definition must not contain set operations (UNION, INTERSECT, and EXCEPT) (at the top level of the query).
  - The selection list (...SELECT selection_list FROM...) in the query must not contain aggregate, window, or set-returning functions.

Views - syntax

```
CREATE [ OR REPLACE ] [ TEMP | TEMPORARY ] [ RECURSIVE ] VIEW
name [ ( column_name [, ...] ) ]
    [ WITH ( name_of_view_parameter [=value_of_view_parameter]
[, ... ] ) ]
    AS query
    [ WITH [ CASCADED | LOCAL ] CHECK OPTION ]
```

Views - examples

```
CREATE VIEW comedies AS
    SELECT *
    FROM films
    WHERE kind = 'Comedy’;
```

- The command will create a view with the columns that were contained in the film table at the time the command was executed. Although * was specified when creating the view, columns added to the table later will not be part of the view

Views - examples

```
CREATE VIEW universal_comedies AS
    SELECT *
    FROM comedies
    WHERE classification = 'U'
    WITH LOCAL CHECK OPTION;
```

- This command will create a view based on the comedies view, returning only comedies (kind = 'Comedy') of the universal age category classification = 'U'. Any attempt s to do INSERT or UPDATE on a view with a row that does not satisfy classification = 'U' will be rejected, but the kind constraint will not be checked.

Views - examples

```
CREATE VIEW pg_comedies AS
    SELECT *
    FROM comedies
    WHERE classification = 'PG'
    WITH CASCADED CHECK OPTION;
```

- this view will check if new rows satisfy both the kind column and the classification column.

Views - examples

```
CREATE VIEW comedies AS
    SELECT f.*,
           country_code_to_name(f.country_code) AS country,
           (SELECT avg(r.rating)
            FROM user_ratings r
            WHERE r.film_id = f.id) AS avg_rating
    FROM films f
    WHERE f.kind = 'Comedy';
```

- This view will support INSERT, UPDATE and DELETE operations. All columns from the films table will be mutable, while the calculated columns country and avg_rating will be read-only.

IV. Window functions

Window functions - description

- The window function performs calculations on a set of rows that are related in some way to the current row. Its action can be compared to the calculation performed by an aggregate function
- With window functions, **strings are not grouped into one output string**
- A window function call always contains an OVER clause following the window function name and arguments. This syntactically distinguishes it from a regular, non-windowed aggregate function. The OVER clause specifies how exactly the query strings should be split for processing by the window function

Window functions - syntax

function_name ([expression]) OVER (window_definition)

window_definition can consist of the following components:

- [name_of_existing_window]
- [PARTITION BY expression [, ...]]
- [ORDER BY expression [ASC | DESC | USING operator] [NULLS { FIRST | LAST }] [, ...]]
- [frame_definition]

- A window function call always contains an OVER clause following the window function name and arguments. This syntactically distinguishes it from a regular, non-windowed aggregate function. The OVER clause specifies how exactly the query strings should be split for processing by the window function

Window functions - syntax

function_name ([expression]) OVER (window_definition)

window_definition can consist of the following components:

- [name_of_existing_window]
- [PARTITION BY expression [, ...]]
- [ORDER BY expression [ASC | DESC | USING operator] [NULLS { FIRST | LAST }] [, ...]]
- [frame_definition]

- The PARTITION BY clause, which complements OVER, separates rows into groups, or sections, by combining the same values of the PARTITION BY clauses. The window function is evaluated on rows that fall in the same section as the current row
- The ORDER BY clause allows you to specify the order in which rows will be processed by window functions
- The window frame parameter allows you to define a set of rows in its section (partition) that will be processed by the window function

Window function - examples

```
SELECT salary, sum(salary)
OVER () FROM empsalary;
```

Salary	Sum
5200	47100
5000	47100
3500	47100
4800	47100
3900	47100
4200	47100
4500	47100
4800	47100
6000	47100
5200	47100

Window function - examples

```
SELECT depname, empno, salary, avg(salary) OVER
(PARTITION BY depname)
        FROM empsalary;
```

Department	Employee Number	Salary	Average Salary
develop	11	5200	5020.0
develop	7	4200	5020.0
develop	9	4500	5020.0
develop	8	6000	5020.0
develop	10	5200	5020.0
personnel	5	3500	3700.0
personnel	2	3900	3700.0
sales	3	4800	4866.7
sales	1	5000	4866.7
sales	4	4800	4866.7

Window function - examples

```
SELECT depname, empno, salary, avg(salary) OVER
(PARTITION BY depname)
        FROM empsalary;
```

- The first three columns are retrieved directly from the empsalary table, with each row in the table having a result row. The fourth column contains the average value calculated over all rows that have the same depname value as the current row

Window function - examples

```
SELECT depname, empno, salary,
        rank() OVER (PARTITION BY depname ORDER BY salary
DESC)
        FROM empsalary;
```

Department	Employee Number	Salary	Rank
develop	8	6000	1
develop	10	5200	2
develop	11	5200	2
develop	9	4500	4
develop	7	4200	5
personnel	2	3900	1
personnel	5	3500	2
sales	1	5000	1
sales	4	4800	2
sales	3	4800	2

Window function - examples

```
SELECT depname, empno, salary,
        rank() OVER (PARTITION BY depname ORDER BY
salary DESC)
        FROM empsalary;
```

- Here, the rank function produces a rank number for each unique value in the current row section that the ORDER BY clause sorts on. The rank function has no parameters, since its behavior is entirely determined by the OVER clause.

Window function - examples

```
SELECT salary, sum(salary)
OVER (ORDER BY salary)
FROM empsalary;
```

Salary	Sum
3500	3500
3900	7400
4200	11600
4500	16100
4800	25700
4800	25700
5000	30700
5200	41100
5200	41100
6000	47100

Window function - examples

SELECT salary, sum(salary) OVER (ORDER BY salary) FROM empsalary

- Here, salaries are accumulated from the first (lowest) to the current one, including repeating current values (note the result in rows with the same salary)
- The fact is that by default, when specifying ORDER BY, the frame consists of all lines from the beginning of the section to the current line and lines equal to the current one in terms of the value of the ORDER BY expression