Querying Data with SELECT Statements

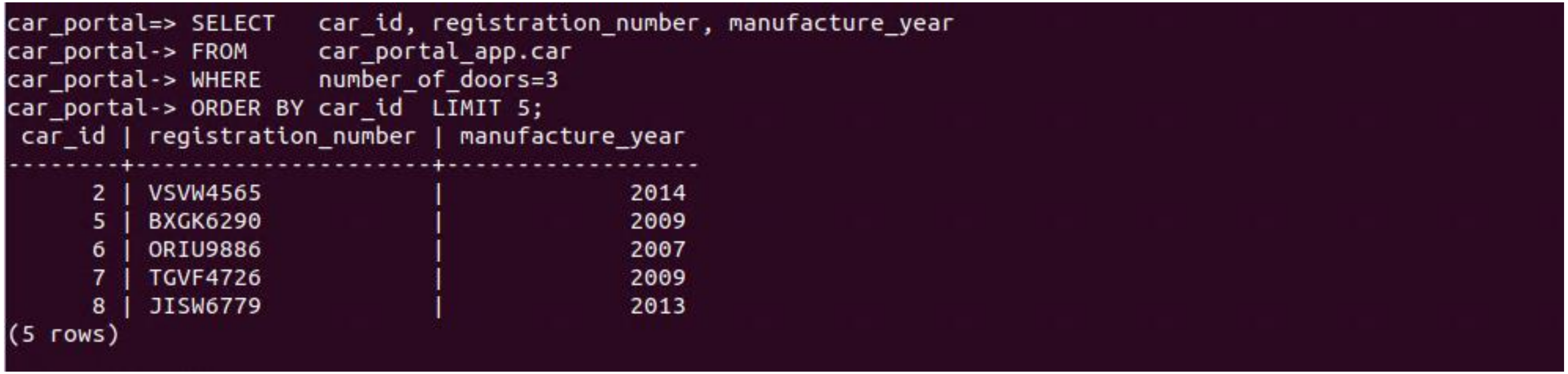
Part 1

- SELECT statements, SELECT queries, or just *queries* are used to retrieve data from a database.
- SELECT queries can have different sources: tables, views, functions, or the VALUES command.
- All of them are relations or can be treated as relations or return relations, which functions can do.
- The output of SELECT is also a relation that, in general, can have multiple columns and contain many rows.
- Since the result and the source of a query have the same nature in SQL, it is possible to use one SELECT query as a source for another statement.
- In this case, both queries are considered parts of one bigger query.
- The source of the data, output format, filters, grouping, ordering, and required transformations of the data are specified in the code of the query.

- In general, `SELECT` queries do not change the data in the database and could be considered read-only, but there is an exception.
- If a volatile function is used in the query, then the data can be changed by the function.

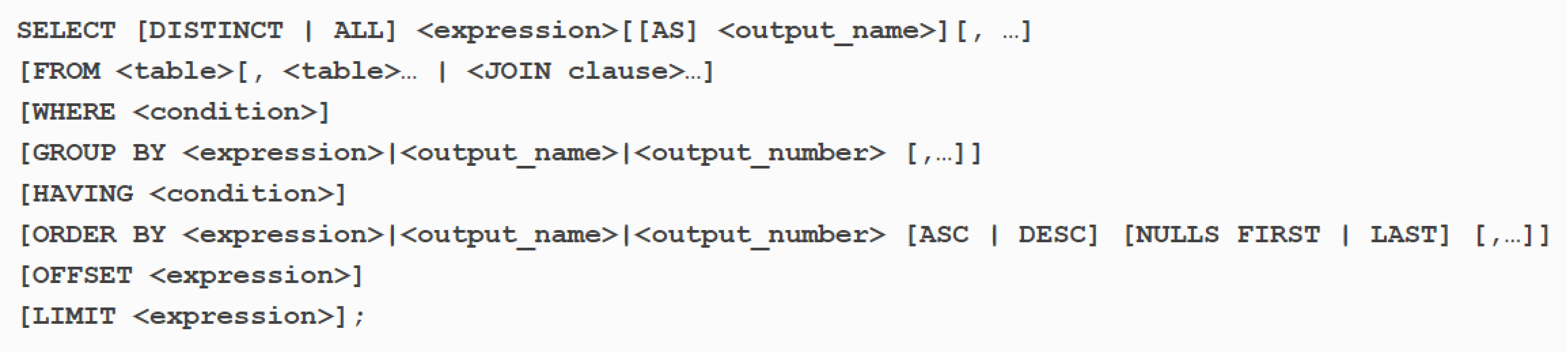
Structure of a SELECT Query

- Imagine the `car_portal` application needs to query the database to get information about cars that have three doors (counting the boot door).
- They should be sorted by their ID.
- The output should be limited to five records due to pagination in the user interface.



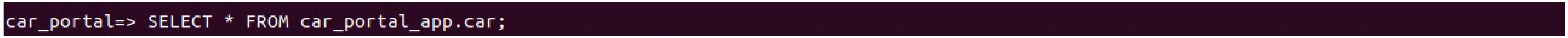
```
car_portal=> SELECT   car_id, registration_number, manufacture_year
car_portal-> FROM     car_portal_app.car
car_portal-> WHERE    number_of_doors=3
car_portal-> ORDER BY car_id  LIMIT 5;
 car_id | registration_number | manufacture_year
--------+---------------------+------------------
      2 | VSVW4565            |             2014
      5 | BXGK6290            |             2009
      6 | ORIU9886            |             2007
      7 | TGVF4726            |             2009
      8 | JISW6779            |             2013
(5 rows)
```

- The simplified syntax diagram for the `SELECT` statement is as follows:

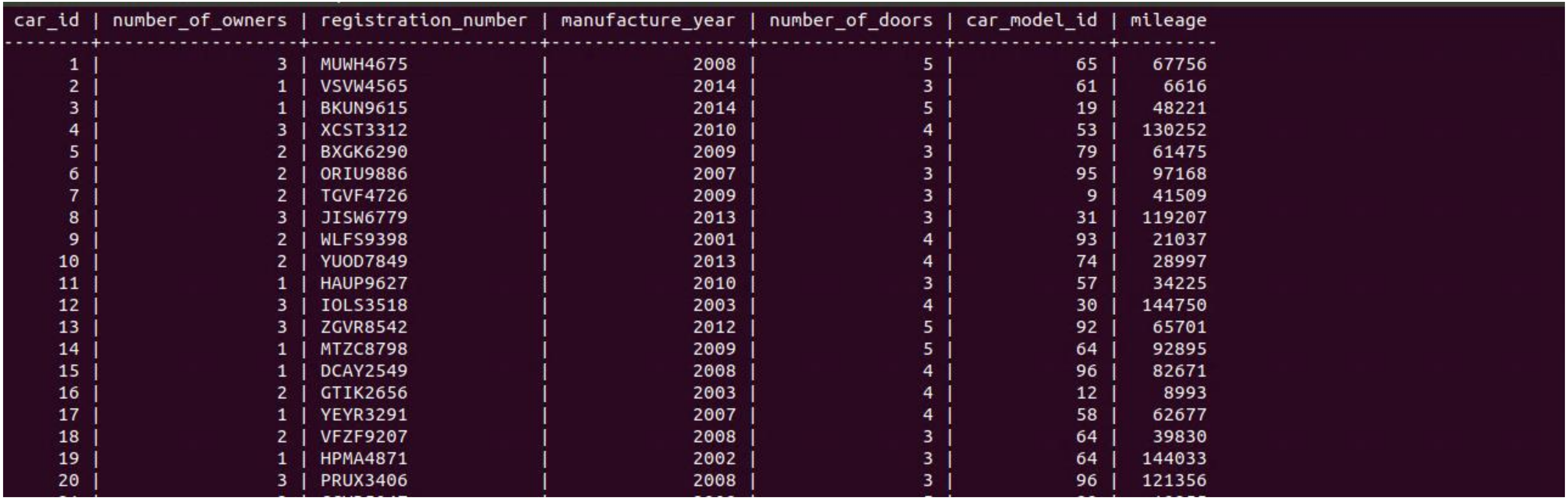


```
SELECT [DISTINCT | ALL] <expression>[[AS] <output_name>][, …]
[FROM <table>[, <table>… | <JOIN clause>…]
[WHERE <condition>]
[GROUP BY <expression>|<output_name>|<output_number> [,…]]
[HAVING <condition>]
[ORDER BY <expression>|<output_name>|<output_number> [ASC | DESC] [NULLS FIRST | LAST] [,…]]
[OFFSET <expression>]
[LIMIT <expression>];
```

- There is no part of the SELECT statement that is always mandatory.
- For example, the query might be simpler if no ordering or filtering is needed:

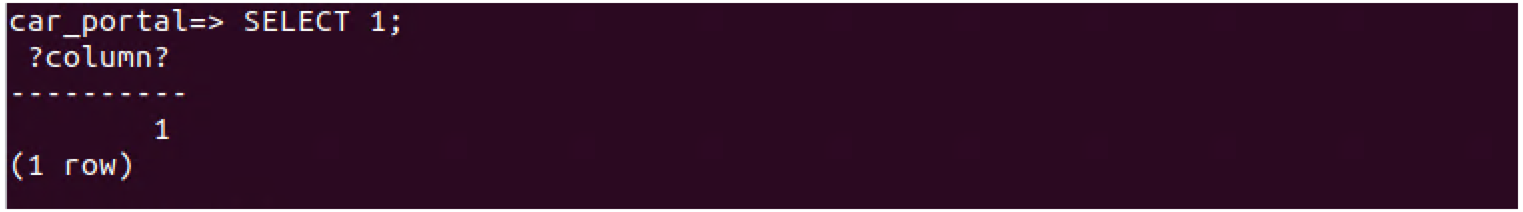


```
car_portal=> SELECT * FROM car_portal_app.car;
```



```
 car_id | number_of_owners | registration_number | manufacture_year | number_of_doors | car_model_id | mileage
--------+------------------+---------------------+------------------+-----------------+--------------+---------
      1 |                3 | MUWH4675            |             2008 |               5 |           65 |   67756
      2 |                1 | VSVW4565            |             2014 |               3 |           61 |    6616
      3 |                1 | BKUN9615            |             2014 |               5 |           19 |   48221
      4 |                3 | XCST3312            |             2010 |               4 |           53 |  130252
      5 |                2 | BXGK6290            |             2009 |               3 |           79 |   61475
      6 |                2 | ORIU9886            |             2007 |               3 |           95 |   97168
      7 |                2 | TGVF4726            |             2009 |               3 |            9 |   41509
      8 |                3 | JISW6779            |             2013 |               3 |           31 |  119207
      9 |                2 | WLFS9398            |             2001 |               4 |           93 |   21037
     10 |                2 | YUOD7849            |             2013 |               4 |           74 |   28997
     11 |                1 | HAUP9627            |             2010 |               3 |           57 |   34225
     12 |                3 | IOLS3518            |             2003 |               4 |           30 |  144750
     13 |                3 | ZGVR8542            |             2012 |               5 |           92 |   65701
     14 |                1 | MTZC8798            |             2009 |               5 |           64 |   92895
     15 |                1 | DCAY2549            |             2008 |               4 |           96 |   82671
     16 |                2 | GTIK2656            |             2003 |               4 |           12 |    8993
     17 |                1 | YEYR3291            |             2007 |               4 |           58 |   62677
     18 |                2 | VFZF9207            |             2008 |               3 |           64 |   39830
     19 |                1 | HPMA4871            |             2002 |               3 |           64 |  144033
     20 |                3 | PRUX3406            |             2008 |               3 |           96 |  121356
```

- Even the FROM clause isn't mandatory.
- When you need to evaluate an expression that doesn't take any data from the database, the query takes this form:



```
car_portal=> SELECT 1;
 ?column?
----------
        1
(1 row)
```

- The FROM clause is optional in PostgreSQL, but in other RDBMSes, such as Oracle, the FROM keyword may be required.

- Logically, the sequence of the operations performed by the SELECT query is as follows:
  1. Take all the records from all the source tables. If there are subqueries in the `FROM` clause, they are evaluated beforehand.
  2. Build all possible combinations of those records and discard the combinations that do not follow the `JOIN` conditions or set some fields to `NULL` in the case of outer joins.
  3. Filter out the combinations that don't match the condition of the `WHERE` clause.
  4. Build groups based on the values of the expressions of the `GROUP BY` list.
  5. Filter the groups that match the `HAVING` conditions.
  6. Evaluate expressions of the `SELECT`-list.
  7. Eliminate duplicated rows if `DISTINCT` is specified.
  8. Apply the `UNION`, `EXCEPT`, or `INTERSECT` set operations.
  9. Sort rows according to the `ORDER BY` clause.
  10. Discard records according to `OFFSET` and `LIMIT`.

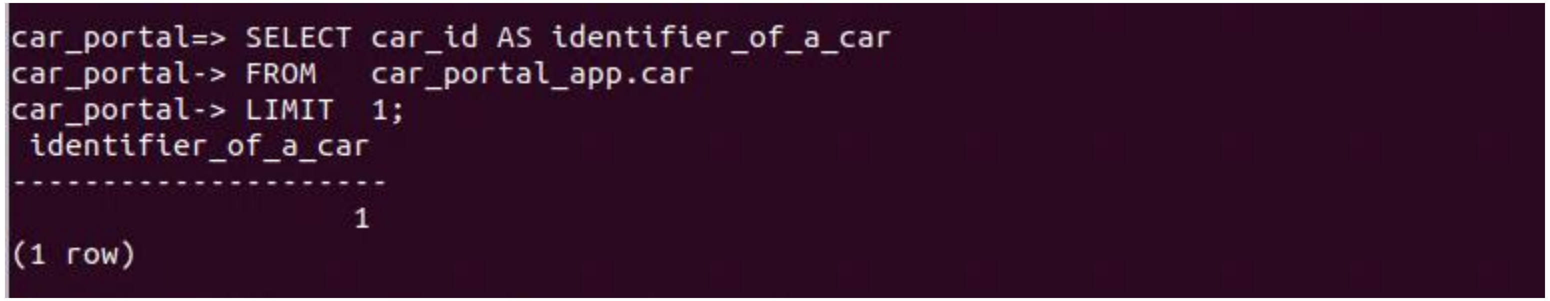
- In fact, PostgreSQL optimizes that algorithm by performing the steps in a different order or even simultaneously.
- For example, if `LIMIT 1` is specified, then it doesn't make sense to retrieve all the rows from the source tables, but only the first one that matches the `WHERE` condition.
- In this case, PostgreSQL would scan the rows one by one and evaluate the condition for each of them in a loop, and stop once a matching row is found.

SELECT-List

- After the `SELECT` keyword, you should specify the list of fields (or expressions) to retrieve from the database.
- This list is called SELECT-list.
- It defines the structure of the query result: the number, names, and types of the selected values.

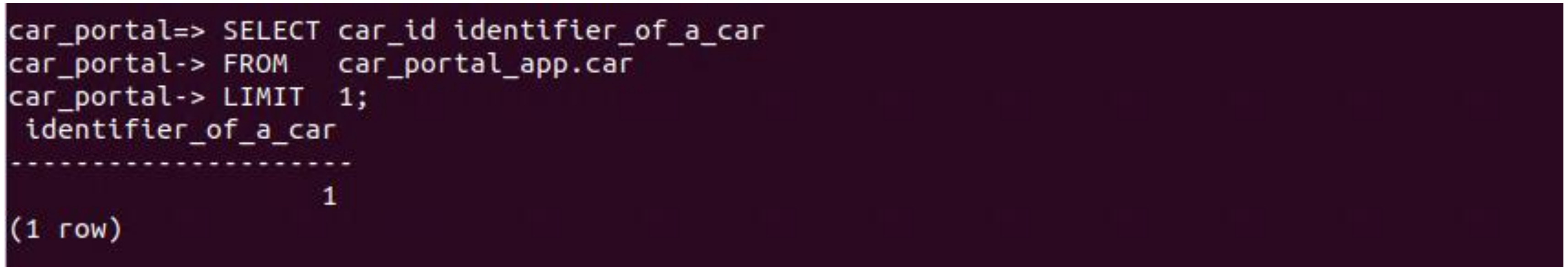
- Every expression in SELECT-list has a name in the output of the query.
- The names, when not provided by the user, are assigned automatically by the database and, in most cases, the name reflects the source of the data: a name of a column when a field from a table is selected, or the name of a function when one is used.
- In other cases, the name will look like `?column?`.

- It's possible, and in many cases it totally makes sense, to provide a different name for a selected expression.
- This is done using the `AS` keyword, like this:



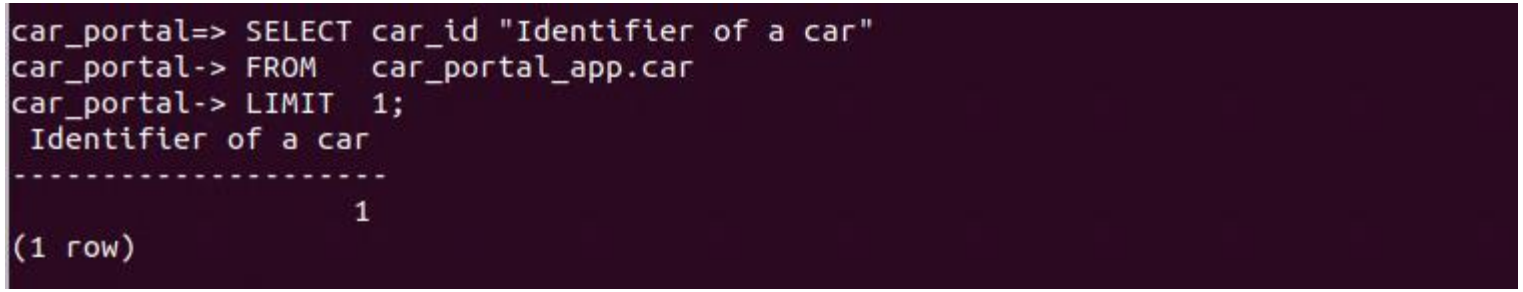
```
car_portal=> SELECT car_id AS identifier_of_a_car
car_portal-> FROM   car_portal_app.car
car_portal-> LIMIT  1;
 identifier_of_a_car
---------------------
                   1
(1 row)
```

- The AS keyword is optional.



```
car_portal=> SELECT car_id identifier_of_a_car
car_portal-> FROM   car_portal_app.car
car_portal-> LIMIT  1;
 identifier_of_a_car
---------------------
                   1
(1 row)
```

- Double-quoted names could be used, for example, when a report is generated by a `SELECT` query without any further processing.
- In that case, it may make sense to use more human-readable column names:



```
car_portal=> SELECT car_id "Identifier of a car"
car_portal-> FROM   car_portal_app.car
car_portal-> LIMIT  1;
 Identifier of a car
---------------------
                   1
(1 row)
```

- In many cases, it is convenient to use an asterisk (`*`) instead of a SELECT-list.
- An asterisk represents all the fields from all the tables specified in the `FROM` clause.
- It's possible to use an asterisk for each table separately, like this:
  - `SELECT car.*, car_model.make ...`
- In this example, all fields are selected from the `car` table and only one `make` field from `car_model`.

- It's considered a bad practice to use * in situations where the query is used in other code, such as in applications, stored procedures, and view definitions.
- It isn't recommended because in the case of using *, the output format depends not on the code of the query but on the structure of the data.
- If the data structure changes, the output format also changes, which will break the application using it.
- However, if you explicitly specify all the output fields in the query and add another column to the input table afterward, this will not change the output of the query and will not break the application.

- So, in our example, instead of `SELECT * . . .`, it would be safer to use the following:

  -
    ```
    SELECT car_id, number_of_owners,
           registration_number, number_of_doors,
           car_model_id, mileage ...
    ```

SQL Expressions

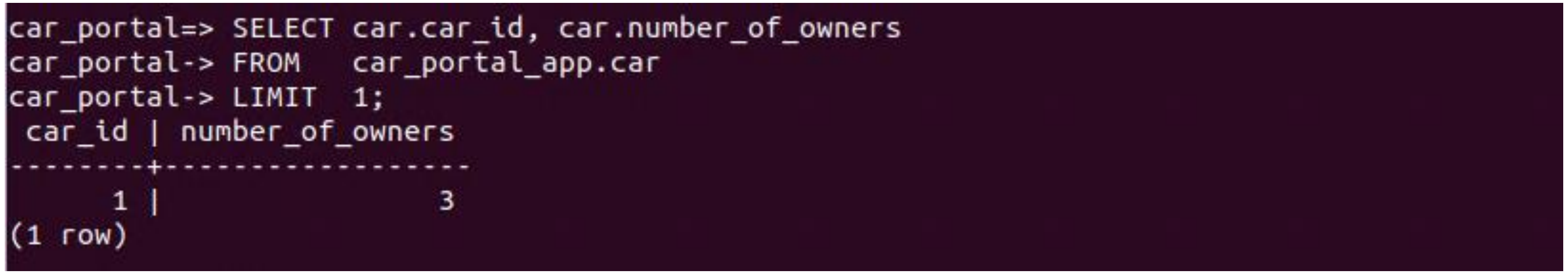
- Expressions in the SELECT-list are called **value expressions** or **scalar expressions**.
- This is because each expression in the SELECT-list always returns only one value (though the value can be an array).

- Scalar expressions can also be called SQL expressions or simply expressions.
- Each expression in SQL has its data type.
- It's determined by the data type(s) of the input.
- In many cases, it's possible to explicitly change the type of the data.
- Each item of the SELECT-list becomes a column in the output dataset, of a type that the corresponding expression has.

- SQL expressions can contain the following:
  - Column names (in most cases)
  - Constants
  - Operator invocations
  - Parentheses to control operations/precedence
  - Function calls
  - Aggregate expressions
  - Scalar subqueries
  - Type casts
  - Conditional expressions
- This list is not complete.

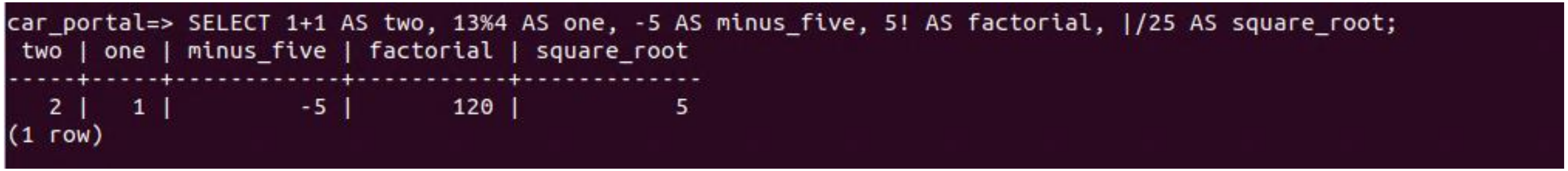
- Column names can be qualified and unqualified.
- **Qualified** means that the name of the column is preceded by the table name, and optionally, the schema name, all separated by a period, (**.**), symbol.
- **Unqualified** indicates just the names of the fields without table references.
- Qualified column names must be used when several tables in the `FROM` clause have columns with the same name.
- Unqualified naming in this case will cause an error: `column reference is ambiguous`.
- This means that the database can't understand which column is being referred to.
- It's possible to use a table alias instead of a table name, and in the case of using subqueries or functions, the alias must be used.

- An example of using qualified names in a SELECT-list is as follows:



```
car_portal=> SELECT car.car_id, car.number_of_owners
car_portal-> FROM   car_portal_app.car
car_portal-> LIMIT  1;
 car_id | number_of_owners
--------+------------------
      1 |                3
(1 row)
```

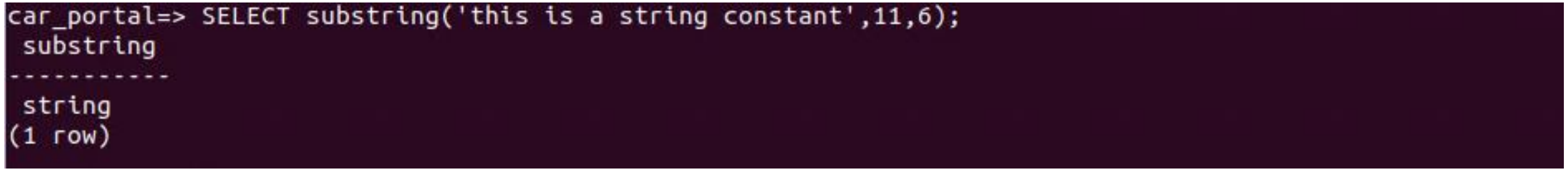
- SQL supports all common operators as most of the other programming languages do, such as logical, arithmetic, string, binary, and date/time.
- An example of using arithmetic operators in expressions would be as follows:



```
car_portal=> SELECT 1+1 AS two, 13%4 AS one, -5 AS minus_five, 5! AS factorial, |/25 AS square_root;
 two | one | minus_five | factorial | square_root
-----+-----+------------+-----------+-------------
   2 |   1 |         -5 |       120 |           5
(1 row)
```

- In PostgreSQL, it is also possible to create user-defined operators.

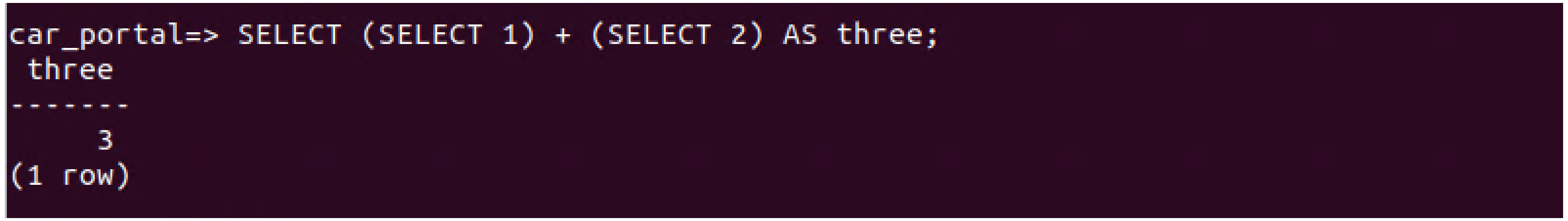
- Function calls can also be part of a SQL expression.
- To call a SQL function, use its name and the arguments in parentheses:



```
car_portal=> SELECT substring('this is a string constant',11,6);
 substring
-----------
 string
(1 row)
```

- If a function has no arguments, it's still necessary to use parentheses to indicate that it's a function name and not a field name or another identifier or keyword.

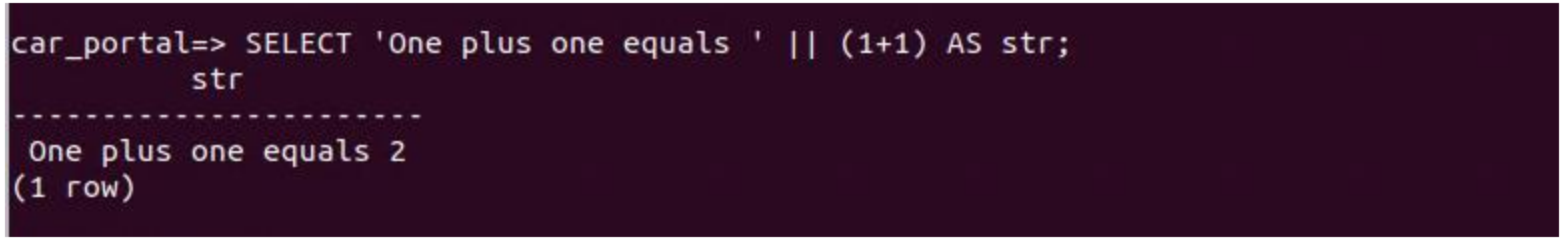
- Another thing that makes SQL very flexible and powerful is scalar subqueries, which can be used as part of a value expression.
- This allows the developer to combine the results of different queries.
- **Scalar subqueries** or **scalar queries** are queries that return exactly one column and one or zero records.
- They have no special syntax and their difference from non-scalar queries is nominal.
- Consider the following example:



```
car_portal=> SELECT (SELECT 1) + (SELECT 2) AS three;
 three
-------
     3
(1 row)
```

- Type-casting means changing the data type of a value.
- Type casts have several syntax patterns, which all have the same meaning:
  - CAST (<value> AS <type>)
  - <value>::<type>
  - <type> '<value>`
  - <type> (<value>)
- The first is a common SQL syntax that is supported in most databases.
- The second is PostgreSQL-specific.
- The third is only applicable for string constants and is usually used to define constants of other types apart from string or numeric.
- The last is function-like and can be applied only to types whose names are also existing function names, which is not very convenient.
  - That's why this syntax is not widely used.

- In many cases, PostgreSQL can do implicit type conversion.
- For example, the concatenation operator, || (double vertical bar), takes two operands of the string type.
- If one tries to concatenate a string with a number, PostgreSQL will convert the number to a string automatically:



```
car_portal=> SELECT 'One plus one equals ' || (1+1) AS str;
          str
-----------------------
 One plus one equals 2
(1 row)
```

- A conditional expression is an expression that returns different results depending on some condition.
- It's similar to an `IF – THEN – ELSE` statement in other programming languages.
- The syntax is as follows:

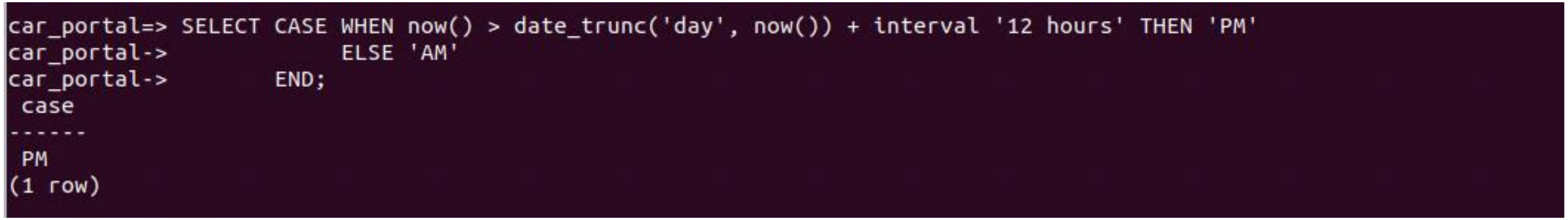
  -
    ```
    CASE WHEN <condition1> THEN <expression1>
        [WHEN <condition2> THEN <expression2> ...]
        [ELSE <expression n>]
    END
    ```

- `CASE` can be used in any place where an SQL expression is used.
- `CASE` expressions can be nested, that is, they can be put inside each other as both a condition part or an expression part.
- The order of evaluating conditions is the same as specified in the expression.
- This means, for any condition, it's known that all preceding conditions are evaluated as false.
- If any condition returns true, subsequent conditions are not evaluated at all.

- There is a simplified syntax for CASE expressions.
- When all conditions check the same expression, whether it is equal to certain values, it is possible to do it like this:

  -
    ```
    CASE <checked_expression>
        WHEN <value1> THEN <result1>
       [WHEN <value2> THEN <result2> ...]
       [ELSE <result_n>]
    END
    ```

- Here is an example of using a CASE expression:



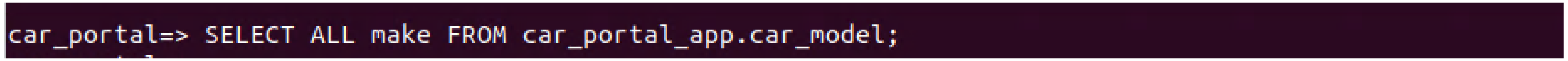
```
car_portal=> SELECT CASE WHEN now() > date_trunc('day', now()) + interval '12 hours' THEN 'PM'
car_portal->               ELSE 'AM'
car_portal->          END;
 case
------
 PM
(1 row)
```

- A single SQL expression can have many operators, functions, type casts, and so on.
- The length of a SQL expression has no limit in language specification.
- The SELECT-list is not the only place where SQL expressions can be used.
- In fact, they are used almost everywhere in SQL statements.
- For example, you can order the results of a query based on a SQL expression, as a sorting key.
- In an `INSERT` statement, they are used to calculate values of the fields for newly-inserted records.
- SQL expressions that return Boolean values are often used as conditions in the `WHERE` clause.

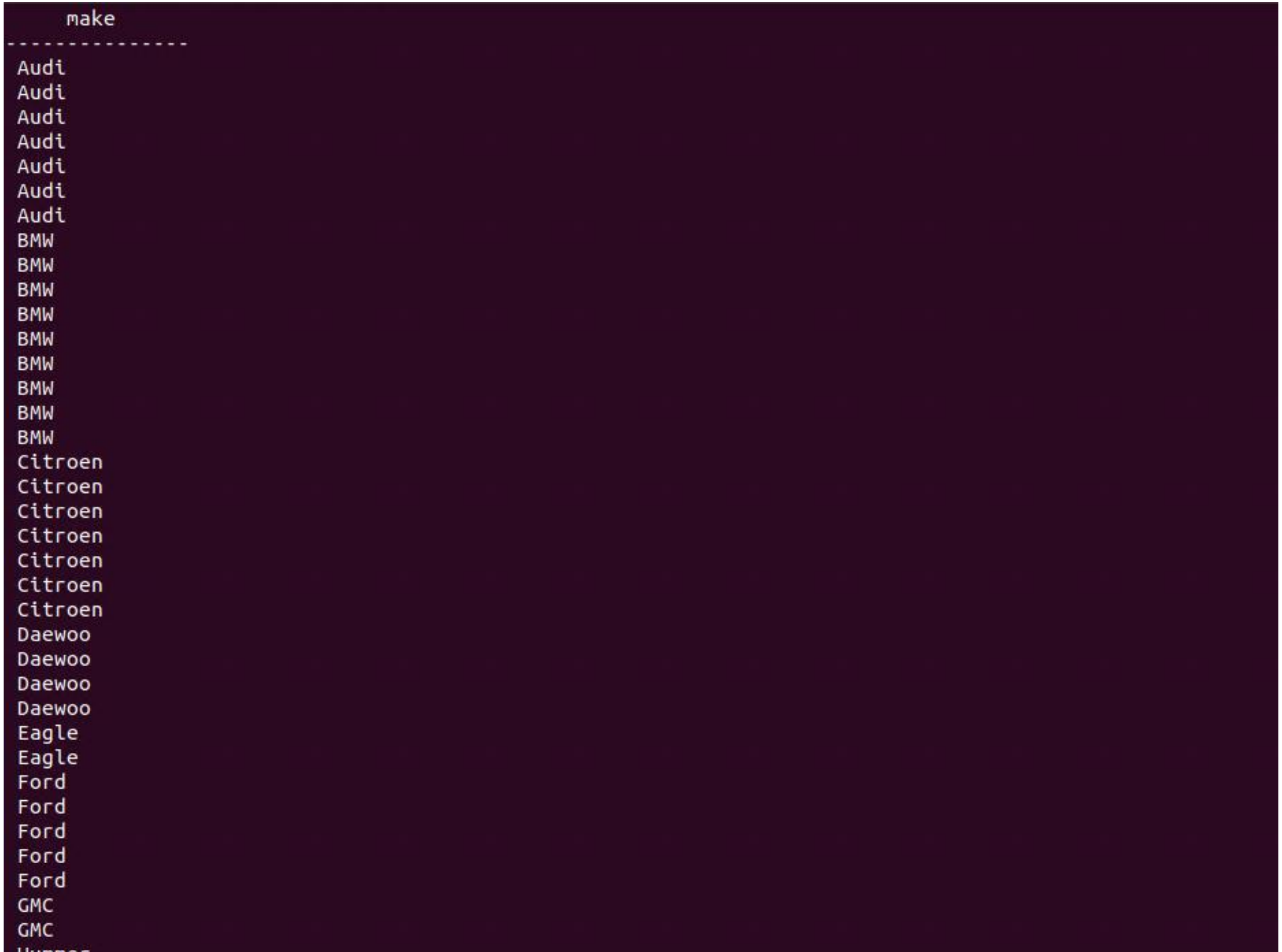
- PostgreSQL supports the short-circuit evaluation of expressions and, sometimes, it skips the evaluation of some parts of an expression when they don't affect the result.
- For example, when evaluating the `false AND z()` expression, PostgreSQL will not call the `z()` function because the result of the `AND` operator is determined by its first operand, the `false` constant, and it is always false, regardless of what the `z()` function returns.

DISTINCT

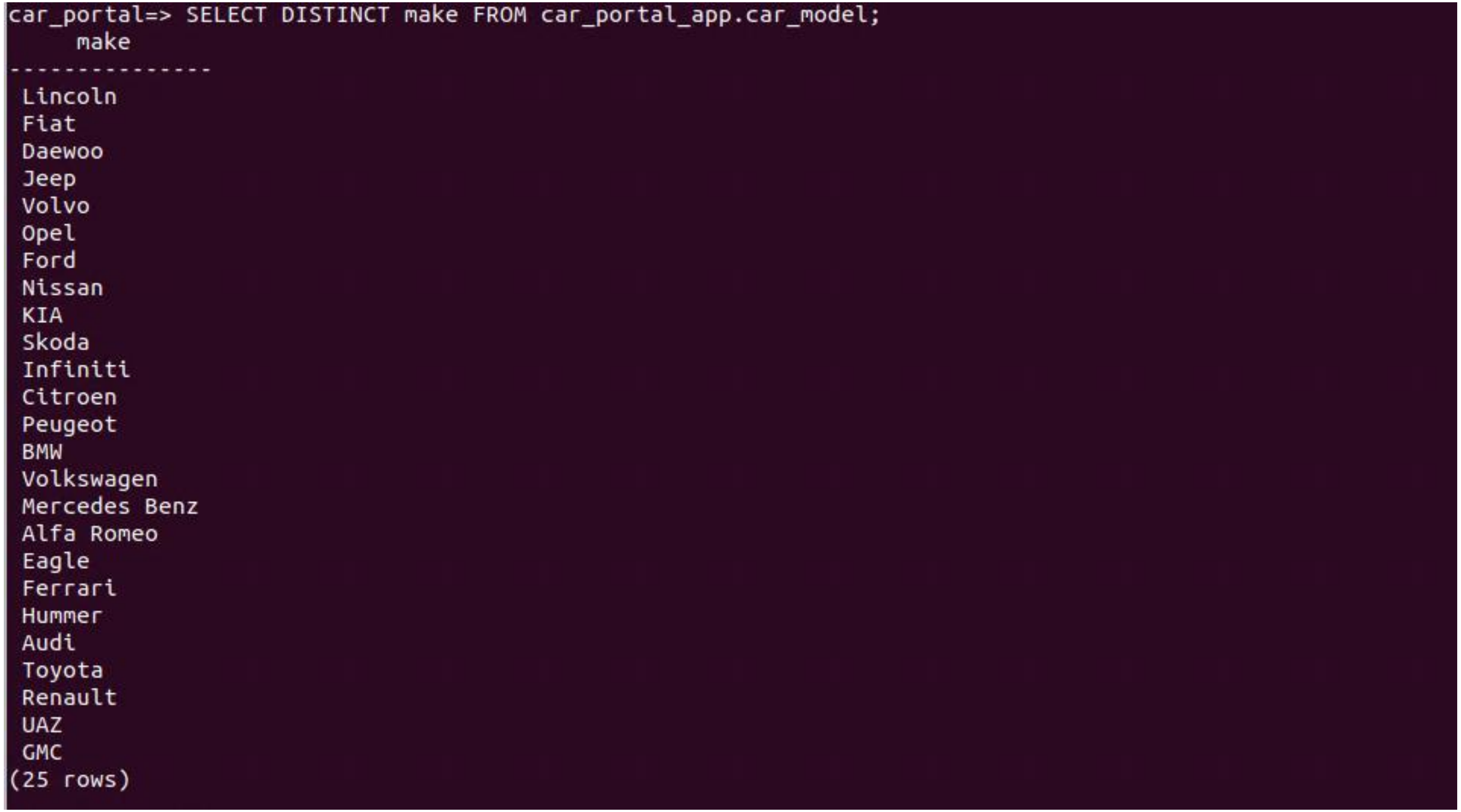
- Also related to the SELECT-list is a pair of keywords, `DISTINCT` and `ALL`, that can be used right after the `SELECT` keyword.
- When `DISTINCT` is specified, only unique rows from the input dataset will be returned.
- `ALL` returns all the rows—this is the default.



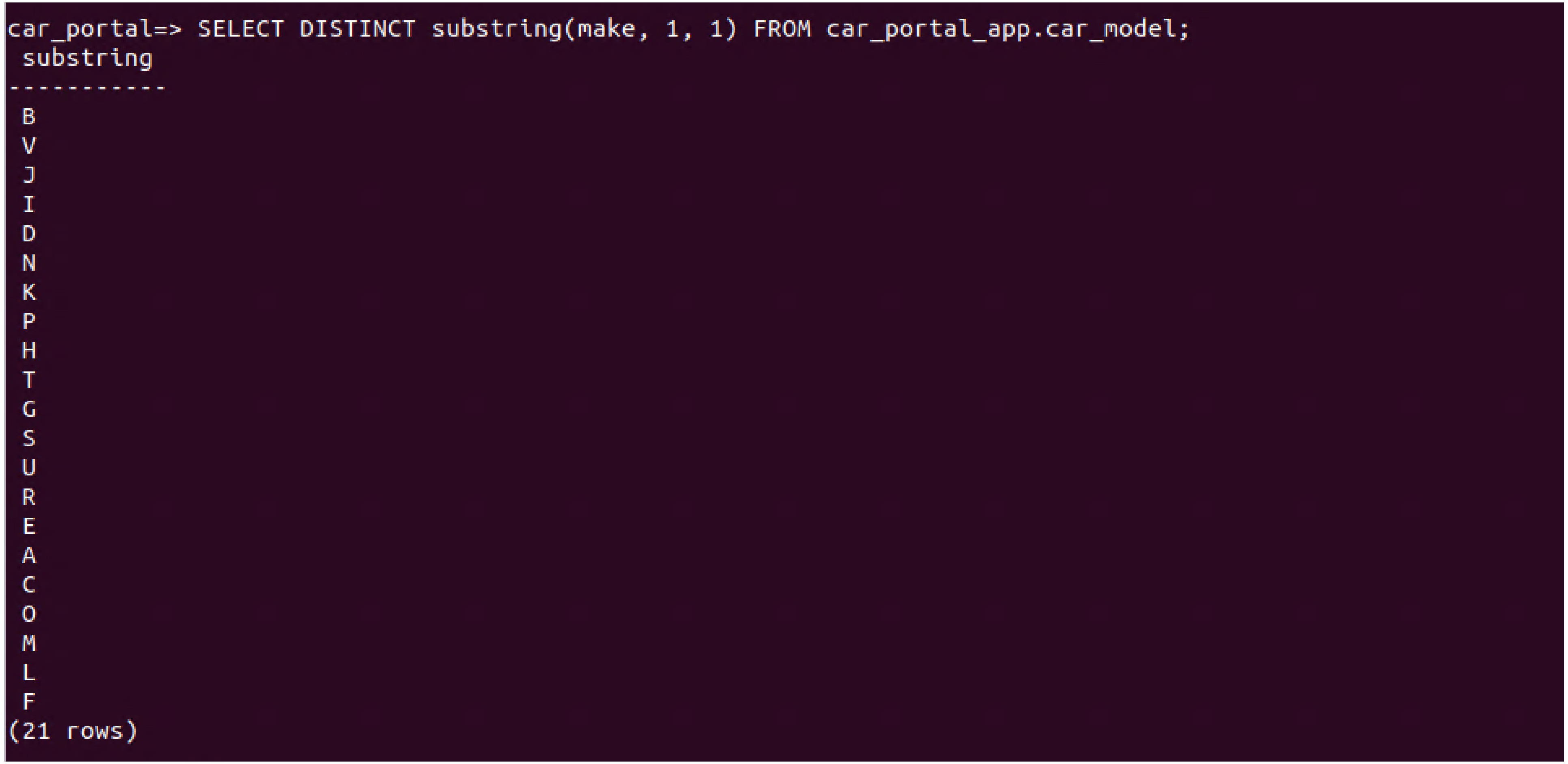
```
car_portal=> SELECT ALL make FROM car_portal_app.car_model;
```



```
     make
---------------
 Audi
 Audi
 Audi
 Audi
 Audi
 Audi
 Audi
 BMW
 BMW
 BMW
 BMW
 BMW
 BMW
 BMW
 BMW
 BMW
 Citroen
 Citroen
 Citroen
 Citroen
 Citroen
 Citroen
 Citroen
 Daewoo
 Daewoo
 Daewoo
 Daewoo
 Eagle
 Eagle
 Ford
 Ford
 Ford
 Ford
 Ford
 GMC
 GMC
 Hummer
```



```
car_portal=> SELECT DISTINCT make FROM car_portal_app.car_model;
     make
---------------
 Lincoln
 Fiat
 Daewoo
 Jeep
 Volvo
 Opel
 Ford
 Nissan
 KIA
 Skoda
 Infiniti
 Citroen
 Peugeot
 BMW
 Volkswagen
 Mercedes Benz
 Alfa Romeo
 Eagle
 Ferrari
 Hummer
 Audi
 Toyota
 Renault
 UAZ
 GMC
(25 rows)
```



```
car_portal=> SELECT DISTINCT substring(make, 1, 1) FROM car_portal_app.car_model;
 substring
-----------
 B
 V
 J
 I
 D
 N
 K
 P
 H
 T
 G
 S
 U
 R
 E
 A
 C
 O
 M
 L
 F
(21 rows)
```