Common Table Expressions

- The same sample database, `car_portal`, is used in the code examples here.
- It's recommended to recreate the sample database in order to get the same results as shown in the code examples.

- Although SQL is a declarative language, it provides a way of implementing the logic of the sequential execution of code or reusing code.

- **Common table expressions (CTEs)** are parts of a SQL statement that produce result sets, defined once, with a view to reuse it, possibly several times, in other parts of the statement.
- The simplified syntax diagram for a CTE is as follows:
  - ```
    WITH <subquery name> AS (<subquery code>) [, ...]
    SELECT <Select list> FROM <subquery name>;
    ```
- In the preceding syntax, subquery code is a query whose results will be used later in the primary query, as if it were a real table.
- The subquery in parentheses after the `AS` keyword is a CTE.
- It can also be called a sub-statement or an auxiliary statement.
- The query after the `WITH` block is the primary or main query.
- The whole statement itself is called a `WITH` query.
    ```

- It's possible to use not only the `SELECT` statements in a CTE, but also the `INSERT`, `UPDATE`, and `DELETE` statements.

- It is also possible to use several CTEs in one `WITH` query.
- Every CTE has its name defined before the `AS` keyword.
- The main query can reference a CTE by its name.
- A CTE can also refer to another CTE by the name.
- A CTE can refer only to the CTEs that were defined before the referencing one.

- The references to CTEs in the primary query can be treated as table names.
- In fact, PostgreSQL executes CTEs only once, caches the results, and reuses them instead of executing subqueries each time they occur in the main query.
- This makes them similar to tables.

- CTEs can help developers to organize SQL code.
- Suppose we want to find the models of the cars in the sample database that were built after 2010, and have the least number of owners.

```
car_portal=> WITH
car_portal-> pre_select AS (
car_portal(>   SELECT car_id, number_of_owners, car_model_id
car_portal(>   FROM   car_portal_app.car
car_portal(>   WHERE  manufacture_year >= 2010),
car_portal-> joined_data AS (
car_portal(>   SELECT car_id, make, model, number_of_owners
car_portal(>   FROM   pre_select INNER JOIN car_portal_app.car_model
car_portal(>          ON pre_select.car_model_id = car_model.car_model_id),
car_portal-> minimal_owners AS (
car_portal(>   SELECT min(number_of_owners) AS min_number_of_owners
car_portal(>   FROM   pre_select)
car_portal-> SELECT car_id, make, model, number_of_owners
car_portal-> FROM   joined_data INNER JOIN minimal_owners
car_portal->        ON joined_data.number_of_owners = minimal_owners.min_number_of_owners;
 car_id |     make      |   model    | number_of_owners
--------+---------------+------------+------------------
      2 | Opel          | Corsa      |                1
      3 | Citroen       | C3         |                1
     11 | Nissan        | GT-R       |                1
     36 | KIA           | Magentis   |                1
     43 | Audi          | A6         |                1
     54 | Alfa Romeo    | Mito       |                1
     73 | Toyota        | RAV4       |                1
     81 | BMW           | 1er        |                1
     88 | Volkswagen    | Scirocco   |                1
     92 | Mercedes Benz | A klasse   |                1
    114 | Mercedes Benz | S klasse   |                1
    118 | BMW           | 7er        |                1
    131 | BMW           | X5         |                1
    137 | Audi          | A8         |                1
    139 | Ford          | S-Max      |                1
    145 | Daewoo        | Matiz      |                1
    157 | Mercedes Benz | S klasse   |                1
    168 | Peugeot       | 308        |                1
    169 | Jeep          | Compass    |                1
    170 | Peugeot       | 407        |                1
    173 | Citroen       | C3         |                1
    180 | Ferrari       | 458 Italia |                1
    182 | Jeep          | Wrangler   |                1
    202 | Volvo         | XC70       |                1
    205 | KIA           | Cerato     |                1
(25 rows)
```

- The order of execution of the CTEs is not defined.
- PostgreSQL aims to execute only the main query.
- If the main query contains references to the CTEs, then PostgreSQL will execute them first.
- If a SELECT sub-statement is not referenced by the main query, directly or indirectly, then it isn't executed at all.
- Data-changing CTEs are always executed.

Reusing SQL Code with CTE

- When the execution of a subquery takes a lot of time, and the subquery is used in the whole SQL statement more than once, it makes sense to put it into a `WITH` clause to reuse its results.
- This makes the query faster because PostgreSQL executes the subqueries from the `WITH` clause only once, caches the results in memory or on disk—depending on their size—and then reuses them.

- For example, let's take the `car_portal` database.
- Suppose it's required to find newer car models.
- This would require it to calculate the average age of the cars of each model and then select the models with an average age lower than the average age of all the models.

```
car_portal=> SELECT make, model, avg_age
car_portal-> FROM   (SELECT car_model_id, avg(EXTRACT(YEAR FROM now())-manufacture_year) AS avg_age
car_portal(>         FROM car_portal_app.car
car_portal(>         GROUP BY car_model_id) age_subq1
car_portal->        INNER JOIN car_portal_app.car_model
car_portal->        ON car_model.car_model_id = age_subq1.car_model_id
car_portal-> WHERE  avg_age < (SELECT avg(avg_age)
car_portal(>                   FROM   (SELECT   avg(EXTRACT(YEAR FROM now()) - manufacture_year) avg_age
car_portal(>                           FROM     car_portal_app.car
car_portal(>                           GROUP BY car_model_id  ) age_subq2);
```

make	model	avg_age
Audi	A3	11.5
Audi	A5	7
Audi	A8	11.5
BMW	1er	6
BMW	5er	6
BMW	7er	10.5
BMW	z4	9
BMW	X3	11
BMW	X5	9.25
BMW	X6	7.5
Citroen	C1	8
Citroen	C3	6.5
Citroen	C4	11.5
Daewoo	Nexia	5
Eagle	Wagon	5
Eagle	Talon	10.5
Ford	Mondeo	8.5
Ford	C-Max	8
KIA	Sportage	10.6666666666667
KIA	Magentis	9
Lincoln	Towncar	11
Mercedes Benz	S klasse	8
Mercedes Benz	A klasse	6.5
Nissan	Skyline	10
Nissan	GT-R	10.6666666666667
Nissan	Murano	10
Opel	Corsa	11
Peugeot	308	8.66666666666667
Peugeot	508	11.5
Renault	Symbol	11
Skoda	Fabia	10
Skoda	Superb	10.5
Toyota	Land Cruiser	11.3333333333333
Toyota	Auris	11.3333333333333
Toyota	RAV4	10.5
Volvo	S60	5
Volkswagen	Passat	7
Volkswagen	Phaeton	7.5
Volkswagen	Scirocco	6
Ferrari	458 Spider	11
Alfa Romeo	Mito	7.5

(41 rows)

- There are two subqueries that are almost the same, `age_subq1` and `age_subq2`: the same table is queried and the same grouping and aggregation are performed.
- This makes it possible to use the same subquery twice by using a CTE:

```
car_portal=> WITH
car_portal-> age_subq AS (
car_portal(>   SELECT   car_model_id, avg(EXTRACT(YEAR FROM now())-manufacture_year) AS avg_age
car_portal(>   FROM     car_portal_app.car
car_portal(>   GROUP BY car_model_id)
car_portal-> SELECT make, model, avg_age
car_portal-> FROM   age_subq INNER JOIN car_portal_app.car_model
car_portal->        ON car_model.car_model_id = age_subq.car_model_id
car_portal-> WHERE  avg_age < (SELECT avg(avg_age)
car_portal(>                   FROM   age_subq);
```

make	model	avg_age
Audi	A3	11.5
Audi	A5	7
Audi	A8	11.5
BMW	1er	6
BMW	5er	6
BMW	7er	10.5
BMW	z4	9
BMW	X3	11
BMW	X5	9.25
BMW	X6	7.5
Citroen	C1	8
Citroen	C3	6.5
Citroen	C4	11.5
Daewoo	Nexia	5
Eagle	Wagon	5
Eagle	Talon	10.5
Ford	Mondeo	8.5
Ford	C-Max	8
KIA	Sportage	10.6666666666667
KIA	Magentis	9
Lincoln	Towncar	11
Mercedes Benz	S klasse	8
Mercedes Benz	A klasse	6.5
Nissan	Skyline	10
Nissan	GT-R	10.6666666666667
Nissan	Murano	10
Opel	Corsa	11
Peugeot	308	8.66666666666667
Peugeot	508	11.5
Renault	Symbol	11
Skoda	Fabia	10
Skoda	Superb	10.5
Toyota	Land Cruiser	11.3333333333333
Toyota	Auris	11.3333333333333
Toyota	RAV4	10.5
Volvo	S60	5
Volkswagen	Passat	7
Volkswagen	Phaeton	7.5
Volkswagen	Scirocco	6
Ferrari	458 Spider	11
Alfa Romeo	Mito	7.5

(41 rows)

- The result of both of the queries is the same.
- However, on the test system used to prepare the code samples, the first query took 1.9 milliseconds to finish and the second one took 1.0 milliseconds.
- Of course, in absolute values, the difference is nothing, but relatively, the `WITH` query is almost twice as fast.
- If the number of records in the tables was in the range of millions, the absolute difference would be substantial.

- Another advantage of using a CTE, in this case, is that the code became shorter and easier to understand.
- That's another use case for the `WITH` clause.
- Long and complicated subqueries can be formatted as CTEs in order to make the whole query shorter and more understandable, even if it doesn't affect the performance.

- Sometimes, though, it's better not to use a CTE.
- For example, you could try to preselect some columns from the table, thinking it would help the database to perform the query faster because of the reduced amount of information to process.
- In that case, the query could be as follows:
  - ```
    WITH car_subquery AS (
      SELECT number_of_owners, manufacture_year,
             number_of_doors
      FROM   car_portal_app.car)
    SELECT number_of_owners, number_of_doors
    FROM   car_subquery
    WHERE  manufacture_year = 2008;
    ```
    ```

- This has the opposite effect.
- PostgreSQL does not push the `WHERE` clause from the primary query to the sub-statement.
- The database will retrieve all the records from the table, take three columns from them, and store this temporary dataset in memory.
- Then, the temporary data will be queried using the `manufacture_year = 2008` predicate.
- If there was an index on `manufacture_year`, it would not be used because the temporary data is being queried and not the real table.

- For this reason, the following query is executed five times faster than the preceding one, even though it seems to be almost the same:
  - ```
    SELECT number_of_owners, manufacture_year,
           number_of_doors
    FROM   car_portal_app.car
    WHERE  manufacture_year = 2008;
    ```
    ```

Recursive and Hierarchical Queries

- It's possible to refer to the name of a CTE from the code of that CTE itself.
- These statements are called **recursive queries**.
- Recursive queries must have a special structure that tells the database that the subquery is recursive.
- The structure of a recursive query is as follows:
  - ```
    WITH RECURSIVE <subquery_name> (<field list>) AS
    (
      <non-recursive term>
      UNION [ALL|DISTINCT]
      <recursive term>
    )
    [,...]
    <main query>
    ```
    ```

- Both non-recursive and recursive terms are subqueries that must return the same number of fields of the same types.
- The names of the fields are specified in the declaration of the whole recursive query; therefore, it does not matter which names are assigned to the fields in the subqueries.

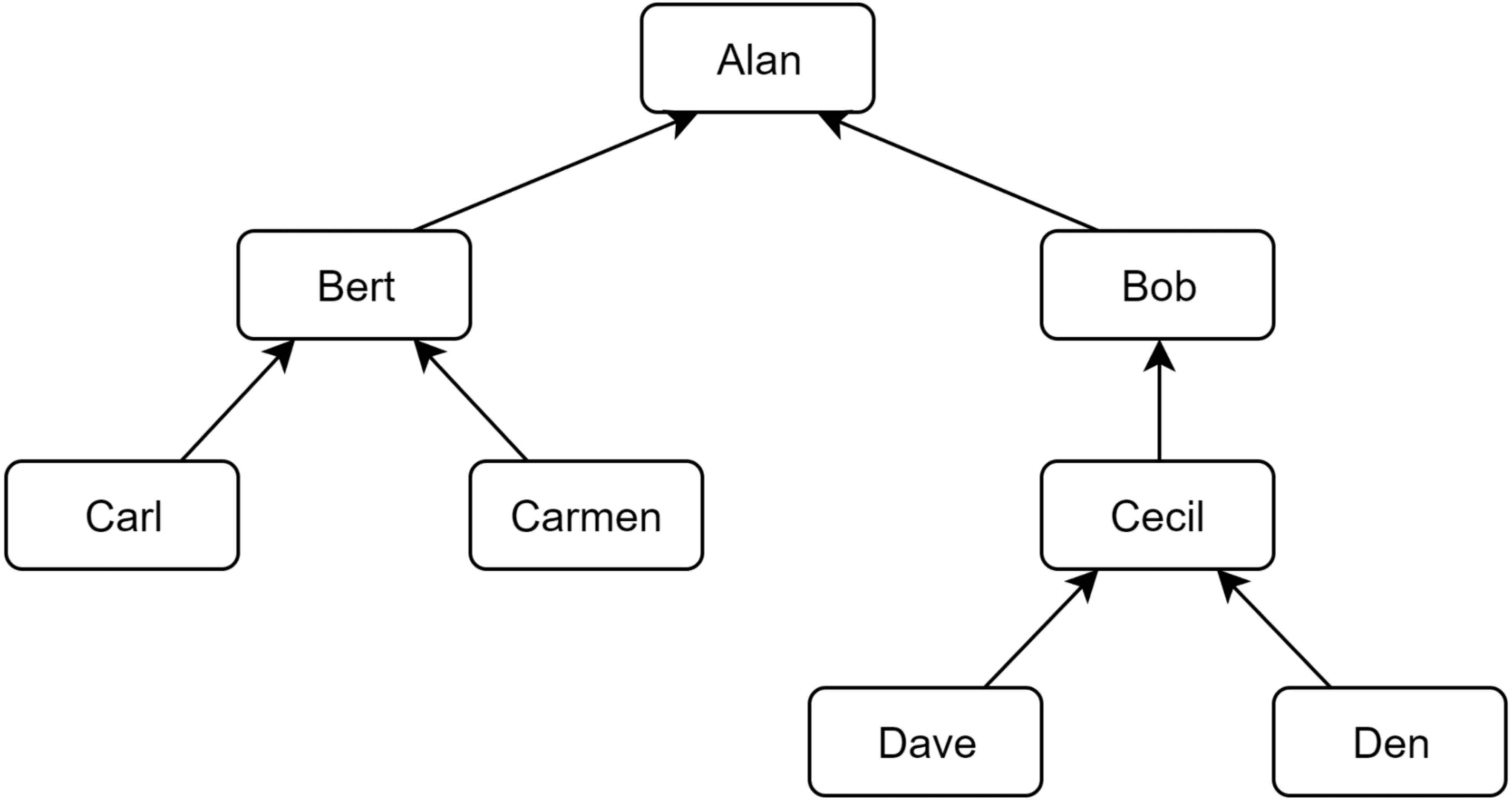
- A non-recursive term is also called an **anchor subquery**, while a recursive term is also known as an **iterating subquery**.

- A non-recursive or anchor subquery is a starting point for the execution of a recursive query.
- It cannot refer to the name of the recursive subquery.
- It's executed only once.
- The results of the non-recursive term are passed again to the same CTE and then only the recursive term is executed.
- It can reference the recursive subquery.
- If the recursive term returns rows, they are passed to the CTE again.
- This is called iteration.
- Iteration is repeated as long as the result of the recursive term is not empty.
- The result of the whole query is all the rows returned by the non-recursive term and all the iterations of the recursive term.
- If the `UNION ALL` keywords are used, all the rows are returned.
- If `UNION DISTINCT` or just `UNION` is used, the duplicated rows are removed from the result set.

- For example, the following recursive query can be used to calculate the factorial values of numbers up to 5:

```
car_portal=> WITH
car_portal-> RECURSIVE subq (n, factorial) AS (
car_portal(>   SELECT 1, 1
car_portal(>   UNION ALL
car_portal(>   SELECT n + 1, factorial * (n + 1)
car_portal(>   FROM   subq
car_portal(>   WHERE  n < 5)
car_portal-> SELECT * FROM subq;
 n | factorial
---+-----------
 1 |         1
 2 |         2
 3 |         6
 4 |        24
 5 |       120
(5 rows)
```

- The preceding example is quite easy to implement without recursive queries.
- PostgreSQL provides a way to generate a series of numeric values and use them in subqueries.
- However, there's a task that cannot be solved without recursive queries: querying a hierarchy of objects.


Alan
Bert
Bob
Carl
Carmen
Cecil
Dave
Den

```
car_portal=> CREATE TABLE family (
car_portal(>   person text PRIMARY KEY,
car_portal(>   parent text REFERENCES family
car_portal(> );
CREATE TABLE
car_portal=> INSERT INTO family
car_portal-> VALUES ('Alan', NULL),
car_portal->        ('Bert', 'Alan'),
car_portal->        ('Bob', 'Alan'),
car_portal->        ('Carl', 'Bert'),
car_portal->        ('Carmen', 'Bert'),
car_portal->        ('Cecil', 'Bob'),
car_portal->        ('Dave', 'Cecil'),
car_portal->        ('Den', 'Cecil');
INSERT 0 8
```

```
car_portal=> WITH
car_portal-> RECURSIVE genealogy (bloodline, person, level) AS (
car_portal(>   SELECT person, person, 0
car_portal(>   FROM   family
car_portal(>   WHERE  person = 'Alan'
car_portal(>   UNION ALL
car_portal(>   SELECT g.bloodline || ' -> ' || f.person, f.person, g.level + 1
car_portal(>   FROM   family f, genealogy g
car_portal(>   WHERE  f.parent = g.person)
car_portal-> SELECT bloodline, level
car_portal-> FROM   genealogy;
          bloodline          | level
-----------------------------+-------
 Alan                        |     0
 Alan -> Bert                |     1
 Alan -> Bob                 |     1
 Alan -> Bert -> Carl        |     2
 Alan -> Bert -> Carmen      |     2
 Alan -> Bob -> Cecil        |     2
 Alan -> Bob -> Cecil -> Dave |     3
 Alan -> Bob -> Cecil -> Den  |     3
(8 rows)
```

- There is a potential problem with such hierarchical queries.
- If the data contained cycles, the recursive query would never stop if used in the same way as the preceding code.
- For example, let's change the starting record in the family table, as follows:

```
car_portal=> UPDATE family
car_portal-> SET    parent = 'Bert'
car_portal-> WHERE  person = 'Alan';
UPDATE 1
```

- Now there's a cycle in the data: `Alan` is a child of his own child.
- If we run the previous bloodline query as is, it would process this cycle until it would eventually fail.
- To use the query, it's necessary to somehow make the query stop.
- This can be done by checking whether the person that is being processed by the recursive term was already included in the bloodline.

```
car_portal=> WITH
car_portal-> RECURSIVE genealogy (bloodline, person, level, processed) AS (
car_portal(>   SELECT person, person, 0, ARRAY[person]
car_portal(>   FROM   family
car_portal(>   WHERE  person = 'Alan'
car_portal(>   UNION ALL
car_portal(>   SELECT g.bloodline || ' -> ' || f.person, f.person, g.level + 1, processed || f.person
car_portal(>   FROM   family f, genealogy g
car_portal(>   WHERE  f.parent = g.person AND
car_portal(>          NOT f.person = ANY(processed))
car_portal-> SELECT bloodline, level
car_portal-> FROM   genealogy;
          bloodline           | level
------------------------------+-------
 Alan                         |     0
 Alan -> Bert                 |     1
 Alan -> Bob                  |     1
 Alan -> Bert -> Carl         |     2
 Alan -> Bert -> Carmen       |     2
 Alan -> Bob -> Cecil         |     2
 Alan -> Bob -> Cecil -> Dave |     3
 Alan -> Bob -> Cecil -> Den  |     3
(8 rows)
```

- There are some limitations to the implementation of recursive queries.
- The use of aggregation is not allowed in the recursive term.
- Moreover, the name of the recursive subquery can be referenced only once in the recursive term.

Changing Data in Multiple Tables at a Time

- Another very useful application of CTEs is performing several data-changing statements at once.
- This is done by including the `INSERT`, `UPDATE`, and `DELETE` statements in CTEs.
- The results of any of these statements can be passed to the following CTEs or to the primary query by specifying the `RETURNING` clause.

- For example, suppose you want to add a new car to the car portal database and there is no corresponding car model in the `car_model` table.
- To do this, you need to enter a new record in the `car_model` table, take the ID of the new record, and use this ID to insert the data into the car table:

```
car_portal=> INSERT INTO car_portal_app.car_model (make, model)
car_portal-> VALUES      ('Ford','Mustang')
car_portal-> RETURNING   car_model_id;
 car_model_id
--------------
          100
(1 row)

INSERT 0 1
car_portal=> INSERT INTO car_portal_app.car (number_of_owners, registration_number, manufacture_year,
car_portal(>                                 number_of_doors, car_model_id, mileage)
car_portal-> VALUES      (1, 'GTR1231', 2014, 4, 100, 10423);
INSERT 0 1
```

- Sometimes, it isn't convenient to perform two statements that store the intermediate ID number somewhere.
- `WITH` queries provide a way to make the changes in both tables at the same time.

```
car_portal=> DELETE FROM car_portal_app.car
car_portal-> WHERE       car_model_id = (SELECT car_model_id
car_portal(>                            FROM   car_portal_app.car_model
car_portal(>                            WHERE  make = 'Ford' AND
car_portal(>                                   model = 'Mustang')
car_portal-> RETURNING   *;
 car_id | number_of_owners | registration_number | manufacture_year | number_of_doors | car_model_id | mileage
--------+------------------+---------------------+------------------+-----------------+--------------+---------
    230 |                1 | GTR1231             |             2014 |               4 |          100 |   10423
(1 row)

DELETE 1
car_portal=> DELETE FROM car_portal_app.car_model
car_portal-> WHERE       make = 'Ford' AND
car_portal->             model = 'Mustang'
car_portal-> RETURNING   *;
 car_model_id | make |  model
--------------+------+---------
          100 | Ford | Mustang
(1 row)

DELETE 1
car_portal=> WITH
car_portal-> car_model_insert AS (
car_portal(>   INSERT INTO car_portal_app.car_model (make, model)
car_portal(>   VALUES ('Ford','Mustang')
car_portal(>   RETURNING car_model_id)
car_portal-> INSERT INTO car_portal_app.car (number_of_owners, registration_number, manufacture_year,
car_portal(>                                 number_of_doors, car_model_id, mileage)
car_portal-> SELECT      1, 'GTR1231', 2014, 4, car_model_id, 10423
car_portal-> FROM        car_model_insert;
INSERT 0 1
```

- CTEs that change the data are always executed.
- It doesn't matter whether they are referenced in the primary query directly or indirectly.
- However, the order of their execution isn't determined.
- You can influence that order by making them dependent on each other.

- What if several CTEs change the same table or use the results produced by each other?
- Here are some principles of their isolation and interaction.

- For sub-statements:
  - All sub-statements work with the data as it was at the time of the start of the whole `WITH` query.
  - They don't see the results of each other's work. For example, it isn't possible for the `DELETE` sub-statement to remove a row that was inserted by another `INSERT` sub-statement.
  - The only way to pass information about the processed records from a data-changing CTE to another CTE is with the `RETURNING` clause.

- For triggers defined on the tables being changed:
  - **For BEFORE triggers**: Statement-level triggers are executed just before the execution of each sub-statement. Row-level triggers are executed just before the changing of every record. This means that a row-level trigger for one sub-statement can be executed before a statement-level trigger for another sub-statement even if the same table is changed.
  - **For AFTER triggers**: Both statement-level and row-level triggers are executed after the whole WITH query. They are executed in groups per every sub-statement: first at the row level and then at the statement level. This means that a statement-level trigger for one sub-statement can be executed before a row-level trigger for another sub-statement, even if the same table is changed.
  - The statements inside the code of the triggers do see the changes in data that were made by other sub-statements.

- For the constraints defined on the tables being changed, assuming they are not set to `DEFERRED`:
  - `PRIMARY KEY` and `UNIQUE` constraints are validated for every record at the time the record is inserted or updated. They take into account the changes made by other sub-statements.
  - `CHECK` constraints are validated for every record at the time the record is inserted or updated. They don't take into account the changes made by other sub-statements.
  - `FOREIGN KEY` constraints are validated at the end of the execution of the whole `WITH` query.

- Here is a simple example of dependency and interaction between CTEs:

```
car_portal=> CREATE TABLE t (f int UNIQUE);
CREATE TABLE
car_portal=> INSERT INTO t VALUES (1);
INSERT 0 1
car_portal=> WITH
car_portal-> del_query AS (
car_portal(>   DELETE FROM t)
car_portal-> INSERT INTO t
car_portal-> VALUES      (1);
ERROR:  duplicate key value violates unique constraint "t_f_key"
DETAIL:  Key (f)=(1) already exists.
```

- The last query failed because PostgreSQL tried to execute the main query before the CTE.

- If you create a dependency that will make the CTE execute first, the record will be deleted and the new record will be inserted.
- In that case, the constraint will not be violated:

```
car_portal=> WITH
car_portal-> del_query AS (
car_portal(>   DELETE FROM t
car_portal(>   RETURNING f)
car_portal-> INSERT INTO t
car_portal-> SELECT 1
car_portal-> WHERE  (SELECT count(*)
car_portal(>         FROM   del_query) IS NOT NULL;
INSERT 0 1
```