Chapter 12: Common Table Expressions (WITH)

Section 12.1: Common Table Expressions in SELECT Queries

Common table expressions support extracting portions of larger queries. For example:

```
WITH sales AS (
  SELECT
    orders.ordered_at,
    orders.user_id,
    SUM(orders.amount) AS total
  FROM orders
  GROUP BY orders.ordered_at, orders.user_id
)
SELECT
  sales.ordered_at,
  sales.total,
  users.NAME
FROM sales
JOIN users USING (user_id)
```

Section 12.2: Traversing tree using WITH RECURSIVE

```
CREATE TABLE empl (
    NAME TEXT PRIMARY KEY,
    boss TEXT NULL
        REFERENCES NAME
            ON UPDATE CASCADE
            ON DELETE CASCADE
        DEFAULT NULL
);

INSERT INTO empl VALUES ('Paul',NULL);
INSERT INTO empl VALUES ('Luke','Paul');
INSERT INTO empl VALUES ('Kate','Paul');
INSERT INTO empl VALUES ('Marge','Kate');
INSERT INTO empl VALUES ('Edith','Kate');
INSERT INTO empl VALUES ('Pam','Kate');
INSERT INTO empl VALUES ('Carol','Luke');
INSERT INTO empl VALUES ('John','Luke');
INSERT INTO empl VALUES ('Jack','Carol');
INSERT INTO empl VALUES ('Alex','Carol');

WITH RECURSIVE t(LEVEL,path,boss,NAME) AS (
        SELECT 0,NAME,boss,NAME FROM empl WHERE boss IS NULL
    UNION
        SELECT
            LEVEL + 1,
            path || ' > ' || empl.NAME,
            empl.boss,
            empl.NAME
        FROM
            empl JOIN t
                ON empl.boss = t.NAME
) SELECT * FROM t ORDER BY path;
```

Chapter 13: Window Functions

Section 13.1: generic example

Preparing data:

```
CREATE TABLE wf_example(i INT, t TEXT,ts timestamptz,b BOOLEAN);
INSERT INTO wf_example SELECT 1,'a','1970.01.01',TRUE;
INSERT INTO wf_example SELECT 1,'a','1970.01.01',FALSE;
INSERT INTO wf_example SELECT 1,'b','1970.01.01',FALSE;
INSERT INTO wf_example SELECT 2,'b','1970.01.01',FALSE;
INSERT INTO wf_example SELECT 3,'b','1970.01.01',FALSE;
INSERT INTO wf_example SELECT 4,'b','1970.02.01',FALSE;
INSERT INTO wf_example SELECT 5,'b','1970.03.01',FALSE;
INSERT INTO wf_example SELECT 2,'c','1970.03.01',TRUE;
```

Running:

```
SELECT *
  , DENSE_RANK() OVER (ORDER BY i) dist_by_i
  , LAG(t) OVER () prev_t
  , NTH_VALUE(i, 6) OVER () nth
  , COUNT(TRUE) OVER (PARTITION BY i) num_by_i
  , COUNT(TRUE) OVER () num_all
  , NTILE(3) over() ntile
FROM wf_example
;
```

Result:

```
 i | t |           ts           | b | dist_by_i | prev_t | nth | num_by_i | num_all | ntile
---+---+------------------------+---+-----------+--------+-----+----------+---------+-------
1 | a | 1970-01-01 00:00:00+01 | f |         1 |        |   3 |        3 |       8 |     1
1 | a | 1970-01-01 00:00:00+01 | t |         1 | a      |   3 |        3 |       8 |     1
1 | b | 1970-01-01 00:00:00+01 | f |         1 | a      |   3 |        3 |       8 |     1
2 | c | 1970-03-01 00:00:00+01 | t |         2 | b      |   3 |        2 |       8 |     2
2 | b | 1970-01-01 00:00:00+01 | f |         2 | c      |   3 |        2 |       8 |     2
3 | b | 1970-01-01 00:00:00+01 | f |         3 | b      |   3 |        1 |       8 |     2
4 | b | 1970-02-01 00:00:00+01 | f |         4 | b      |   3 |        1 |       8 |     3
5 | b | 1970-03-01 00:00:00+01 | f |         5 | b      |   3 |        1 |       8 |     3
(8 rows)
```

Explanation:

dist_by_i: `DENSE_RANK() OVER (ORDER BY i)` is like a row_number per distinct values. Can be used for the number of distinct values of i (`COUNT(DISTINCT i)` wold not work). Just use the maximum value.

prev_t: `LAG(t) OVER ()` is a previous value of t over the whole window. mind that it is null for the first row.

nth: `NTH_VALUE(i, 6) OVER ()` is the value of sixth rows column i over the whole window

num_by_i: `COUNT(TRUE) OVER (PARTITION BY i)` is an amount of rows for each value of i

num_all: `COUNT(TRUE) OVER ()` is an amount of rows over a whole window

ntile: `NTILE(3) over()` splits the whole window to 3 (as much as possible) equal in quantity parts

Section 13.2: column values vs dense_rank vs rank vs row_number

here you can find the functions.

With the table wf_example created in previous example, run:

```
SELECT i
  , DENSE_RANK() OVER (ORDER BY i)
  , ROW_NUMBER() OVER ()
  , RANK() OVER (ORDER BY i)
FROM wf_example
```

The result is:

```
 i | dense_rank | row_number | rank
---+------------+------------+------
1 |          1 |          1 |    1
1 |          1 |          2 |    1
1 |          1 |          3 |    1
2 |          2 |          4 |    4
2 |          2 |          5 |    4
3 |          3 |          6 |    6
4 |          4 |          7 |    7
5 |          5 |          8 |    8
```

- *dense_rank* orders **VALUES** of **i** by appearance in window. `i=1` appears, so first row has dense_rank, next and third i value does not change, so it is `dense_rank` shows *1* - FIRST value not changed. fourth row `i=2`, it is second value of *i* met, so `dense_rank` shows 2, andso for the next row. Then it meets value `i=3` at 6th row, so it show 3. Same for the rest two values of *i*. So the last value of `dense_rank` is the number of distinct values of *i*.
- *row_number* orders **ROWS** as they are listed.
- *rank* Not to confuse with `dense_rank` this function orders **ROW NUMBER** of **i** values. So it starts same with three ones, but has next value 4, which means `i=2` (new value) was met at row 4. Same `i=3` was met at row 6. Etc..