Wykład 5: Zaawansowany SQL

SQL się zmienia

SQL-86	pierwsza wersja
SQL-89	więzy spójności
SQL-92	to, co większość ludzi rozumie jako SQL (*vanilla SQL*)
SQL:1999	WITH RECURSIVE, wyzwalacze, integracja z Javą, LATERAL
SQL:2003	SQL/XML, funkcje okienkowe
SQL:2006	SQL/XQL: integracja z XQuery
SQL:2008	wyzwalacze INSTEAD OF, klauzula FETCH
SQL:2011	obsługa danych temporalnych
SQL:2016	JSON
SQL:2018	SQL/MDA: tablice wielowymiarowe
SQL:202?	SQL/PGQ: obsługa grafowych baz danych (w stylu Neo4j)

https://modern-sql.com/ https://www.postgresql.org/about/featurematrix/#sql

LATERAL

Dla każdego szefa wypisz wszystkich podwładnych. Następujące zapytanie jest nielegalne:

```
SELECT m.ename, e.ename
FROM emp m, (SELECT * FROM emp WHERE emp.mgr = m.empno) e;
```

Dzięki temu silnik może niezależnie (równolegle) wyliczać podzapytania w klauzuli FROM.

Poza tym, jaki byłby z tego pożytek. Przecież można krócej tak:

```
SELECT m.ename, e.ename
FROM emp m JOIN emp e ON emp.mgr = m.empno;
```

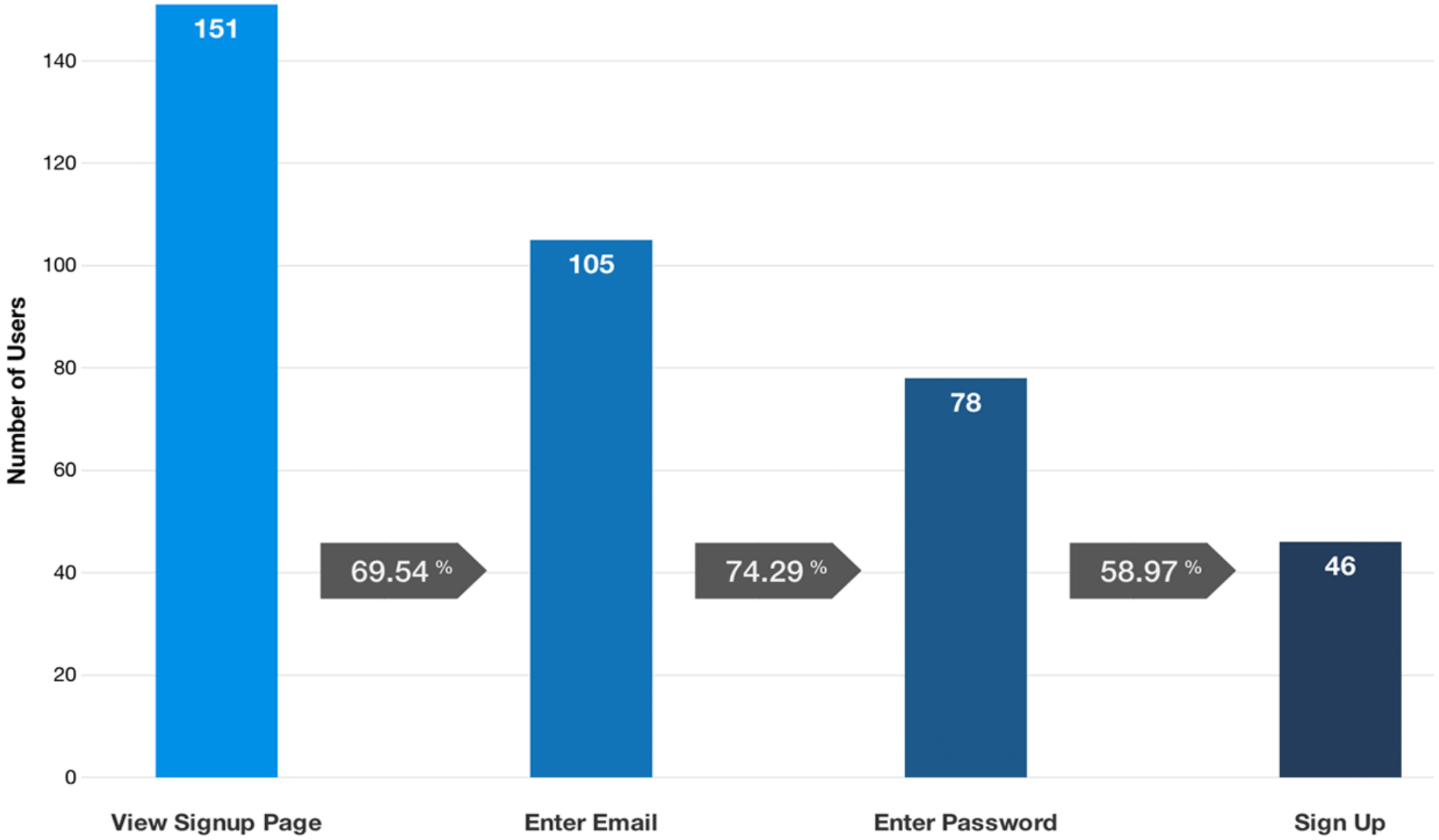
Ale co jeśli mamy wypisać tylko najlepiej zarabiającego podwładnego?

```
SELECT m.ename, e.ename, e.sal
FROM emp m, LATERAL (SELECT * FROM emp WHERE emp.mgr = m.empno
                     ORDER BY emp.sal DESC LIMIT 1) e;
```

Słowo kluczowe LATERAL przez podzapytaniem w klauzuli FROM pozwala się odwoływać do kolumn poprzednich tabel/podzapytań. Semantyka: podzapytanie jest wyliczane oddzielnie dla każdej kombinacji krotek z poprzednich tabel/podzapytań (tutaj: dla każdej krotki w `m`).

UWAGA: W standardzie i w Oracle'u zamiast `LIMIT n` jest `FETCH FIRST n ROWS ONLY` .

Większy przykład: *Conversion funnel*



Źródło: https://heap.io/blog/postgresqls-powerful-new-join-type-lateral

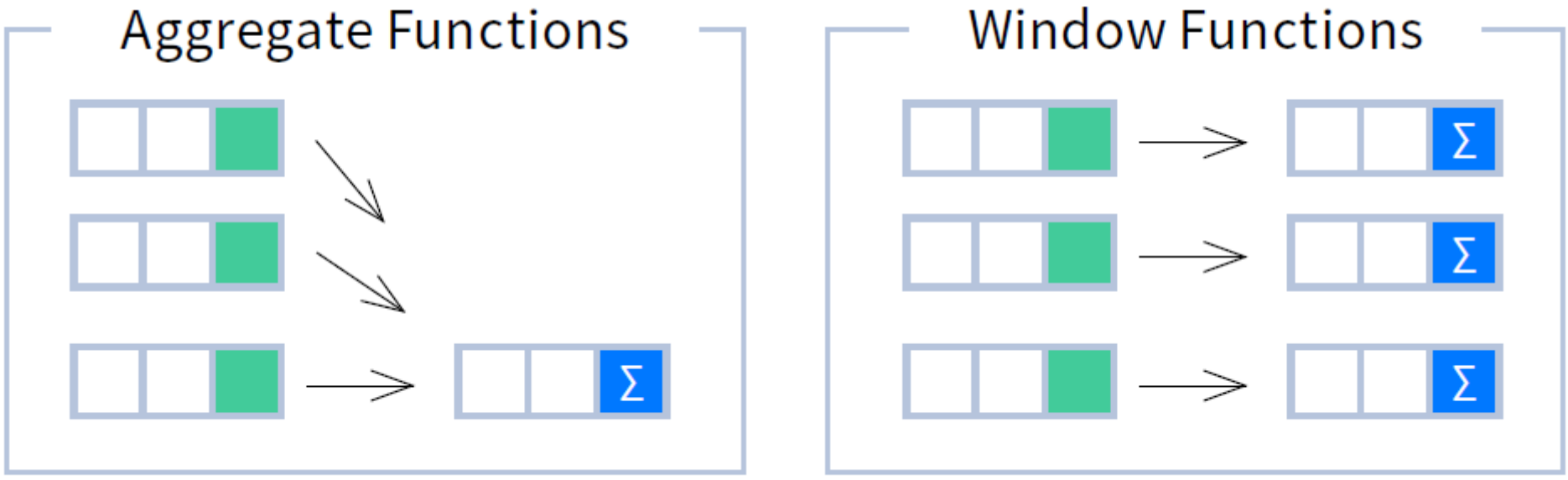
Większy przykład: *Conversion funnel*

```
SELECT
  count(view_homepage_time) AS viewed_homepage,
  count(enter_credit_card_time) AS entered_credit_card
FROM (
    SELECT                          -- Get the first time each user viewed the homepage.
      user_id,
      min(time) AS view_homepage_time
    FROM event
    WHERE
      event_type = 'view_homepage'
    GROUP BY user_id
) e1 LEFT JOIN LATERAL (
    SELECT                                  -- For each (user_id, view_homepage_time),
      time AS enter_credit_card_time        -- get the first time that user triggered
    FROM event                               -- the enter_credit_card event,
    WHERE                                   -- if one exists within two weeks.
      user_id = e1.user_id AND
      event_type = 'enter_credit_card' AND
      time BETWEEN view_homepage_time AND (view_homepage_time + 1000*60*60*24*14)
    ORDER BY time LIMIT 1
) e2 ON true;
```

Źródło: https://heap.io/blog/postgresqls-powerful-new-join-type-lateral

Funkcje okienkowe a agregacja

Inaczej niż zwykła agregacja, funkcje okienkowe nie kolapsują wierszy w grupie.



Źródło: https://learnsql.com/blog/sql-window-functions-cheat-sheet/

Składnia

```
SELECT city, month,
  sum(sold) OVER (
    PARTITION BY city
    ORDER BY month
    RANGE UNBOUNDED PRECEDING) total
FROM sales;
```

```
SELECT <column_1>, <column_2>,
  <window_function> OVER (
    PARTITION BY <...>
    ORDER BY <...>
    <window_frame>) <window_column_alias>
FROM <table_name>;
```

```
SELECT country, city,
  rank() OVER country_sold_avg
FROM sales
WHERE month BETWEEN 1 AND 6
GROUP BY country, city
HAVING sum(sold) > 10000
WINDOW country_sold_avg AS (
  PARTITION BY country
  ORDER BY avg(sold) DESC)
ORDER BY country, city;
```

```
SELECT <column_1>, <column_2>,
  <window_function>() OVER <window_name>
FROM <table_name>
WHERE <...>
GROUP BY <...>
HAVING <...>
WINDOW <window_name> AS (
  PARTITION BY <...>
  ORDER BY <...>
  <window_frame>)
ORDER BY <...>;
```

`PARTITION BY`, `ORDER BY`, `<window_frame>` są opcjonalne.

Źródło: https://learnsql.com/blog/sql-window-functions-cheat-sheet/

PARTITION BY

`PARTITION BY` dzieli wiersze na grupy, nazywane **partycjami**, do których stosuje się funkcję okienkową.

month	city	sold
1	Rome	200
2	Paris	500
1	London	100
1	Paris	300
2	Rome	300
2	London	400
3	Rome	400

PARTITION BY city

month	city	sold	sum
1	Paris	300	800
2	Paris	500	800
1	Rome	200	900
2	Rome	300	900
3	Rome	400	900
1	London	100	500
2	London	400	500

Domyślnie: Jeśli nie ma `PARTITION BY`, to cały zbiór krotek jest jedną partycją.

Źródło: https://learnsql.com/blog/sql-window-functions-cheat-sheet/

ORDER BY

`ORDER BY` specyfikuje porządek na wierszach w obrębie partycji.

sold	city	month
200	Rome	1
500	Paris	2
100	London	1
300	Paris	1
300	Rome	2
400	London	2
400	Rome	3

`PARTITION BY city ORDER BY month`

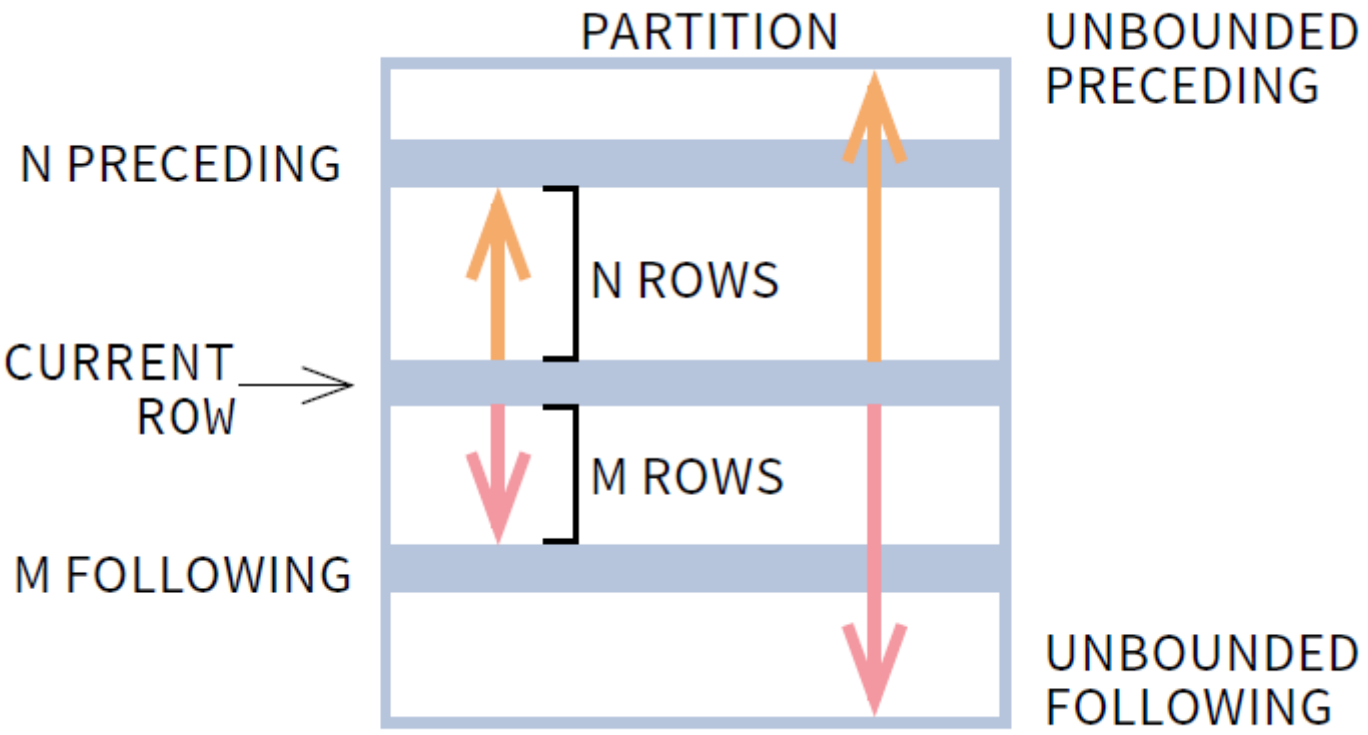
sold	city	month
300	Paris	1
500	Paris	2
200	Rome	1
300	Rome	2
400	Rome	3
100	London	1
400	London	2

Domyślnie: Jeśli nie ma `ORDER BY`, to porządek w obrębie partycji jest dowolny.

Źródło: https://learnsql.com/blog/sql-window-functions-cheat-sheet/

Ramka okienka `<window_frame>`

Ramka okienka to zbiór wierszy danej partycji używany do ewaluacji funkcji okienkowej, określony względem bieżącego wiersza. (W każdej partycji jest wyliczany niezależnie.)



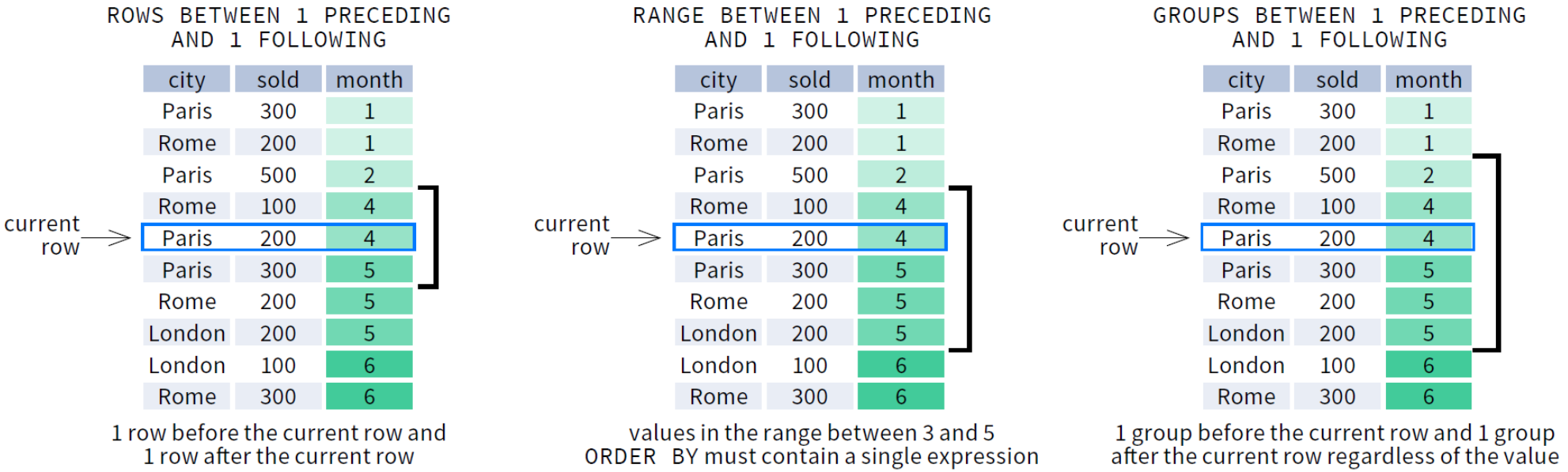
Źródło: https://learnsql.com/blog/sql-window-functions-cheat-sheet/

Ramka okienka `<window_frame>` (2)

```
ROWS | RANGE | GROUPS BETWEEN lower_bound AND upper_bound
```

(`GROUPS` tylko PostgreSQL), gdzie `lower_bound` i `upper_bound` można wybrać spośród

```
UNBOUNDED PRECEDING | n PRECEDING | CURRENT ROW | n FOLLOWING | UNBOUNDED FOLLOWING
```



Domyślnie: `RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW` z `ORDER BY`;

`ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING` bez `ORDER BY`.

Źródło: https://learnsql.com/blog/sql-window-functions-cheat-sheet/

Logiczna kolejność operacji w SQLu

1. `FROM`, `JOIN`
2. `WHERE`
3. `GROUP BY`
4. funkcje agregujące
5. `HAVING`
6. **funkcje okienkowe**
7. `SELECT`
8. `DISTINCT`
9. `UNION`/`INTERSECT`/`EXCEPT`
10. `ORDER BY`
11. `OFFSET`
12. `LIMIT`/`FETCH`/`TOP`

Funkcji okienkowych można używać w klauzulach `SELECT` i `ORDER BY`. Nie można ich używać w klauzulach `FROM`, `WHERE`, `GROUP BY`, ani `HAVING`.

Źródło: https://learnsql.com/blog/sql-window-functions-cheat-sheet/

Lista funkcji okienkowych

- Funkcje rankingowe
  - `row_number()`
  - `rank()`
  - `dense_rank()`
- Funkcje dystrybucji (rozkładu)
  - `percent_rank()`
  - `cume_dist()`
- Funkcje analityczne
  - `lead()`
  - `lag()`
  - `ntile()`
  - `first_value()`
  - `last_value()`
  - `nth_value()`
- Funkcje agregujące
  - `avg()`
  - `count()`
  - `max()`
  - `min()`
  - `sum()`

Źródło: https://learnsql.com/blog/sql-window-functions-cheat-sheet/

Funkcje rankingu

- `row_number()` - unikatowy numer dla każdego wiersza partycji, różne numery w przypadku remisu.
- `rank()` - ranking w obrębie partycji, w przypadku remisu ten sam, z lukami.
- `dense_rank()` - ranking w obrębie partycji, w przypadku remisu ten sam, bez luk.

city	price	row_number	rank	dense_rank
		over(order by price)		
Paris	7	1	1	1
Rome	7	2	1	1
London	8.5	3	3	2
Berlin	8.5	4	3	2
Moscow	9	5	5	3
Madrid	10	6	6	4
Oslo	10	7	6	4

ORDER BY i ramki: `rank()` i `dense_rank()` wymagają `ORDER BY`, a `row_number()` nie wymaga.
Specyfikacja ramki (`ROWS`, `RANGE`, `GROUPS`) niedozwolona.

Źródło: https://learnsql.com/blog/sql-window-functions-cheat-sheet/

Funkcje dystrybucji (rozkładu)

- `percent_rank()` - (rank-1) / (liczba wierszy - 1), liczba z przedziału [0, 1]:
- `cume_dist()` - dystrybuanta empiryczna: liczba wierszy nie większych podzielona przez łączną liczbę wierszy; liczba z przedziału (0, 1]

`cume_dist() OVER(ORDER BY sold)`

city	sold	cume_dist
Paris	100	0.2
Berlin	150	0.4
Rome	200	0.8
Moscow	200	0.8
London	300	1

80% of values are less than or equal to this one

`percent_rank() OVER(ORDER BY sold)`

city	sold	percent_rank
Paris	100	0
Berlin	150	0.25
Rome	200	0.5
Moscow	200	0.5
London	300	1

without this row 50% of values are less than this row's value

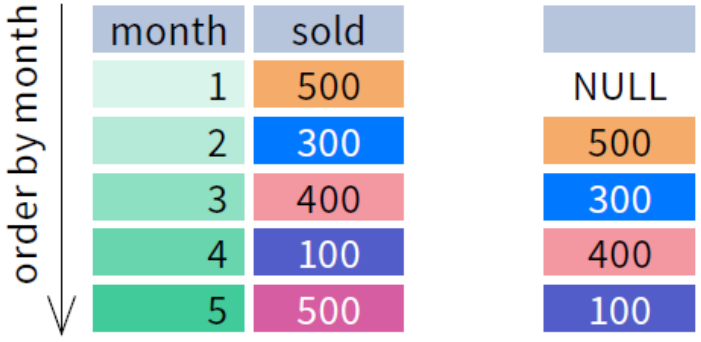
ORDER BY i ramki: `ORDER BY` wymagane. Specyfikacja ramki (`ROWS`, `RANGE`, `GROUPS`) niedozwolona.

Źródło: https://learnsql.com/blog/sql-window-functions-cheat-sheet/

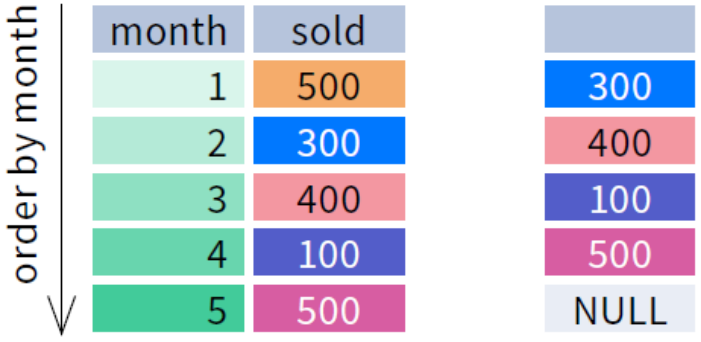
Funkcje analityczne

- `lead(expr,offset,default)` - wartość dla wiersza *offset* za bieżącym;
- `lag(expr,offset,default)` - wartość dla wiersza *offset* przed bieżącym;
  - domyślnie *offset* = 1, *default* = `NULL`.

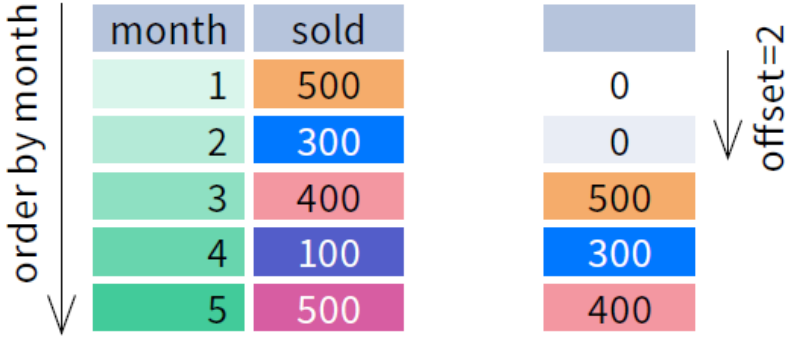
`lag(sold) OVER(ORDER BY month)`



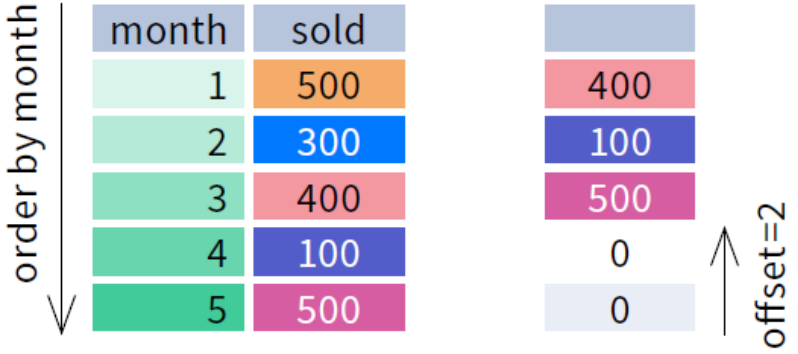
`lead(sold) OVER(ORDER BY month)`



`lag(sold, 2, 0) OVER(ORDER BY month)`



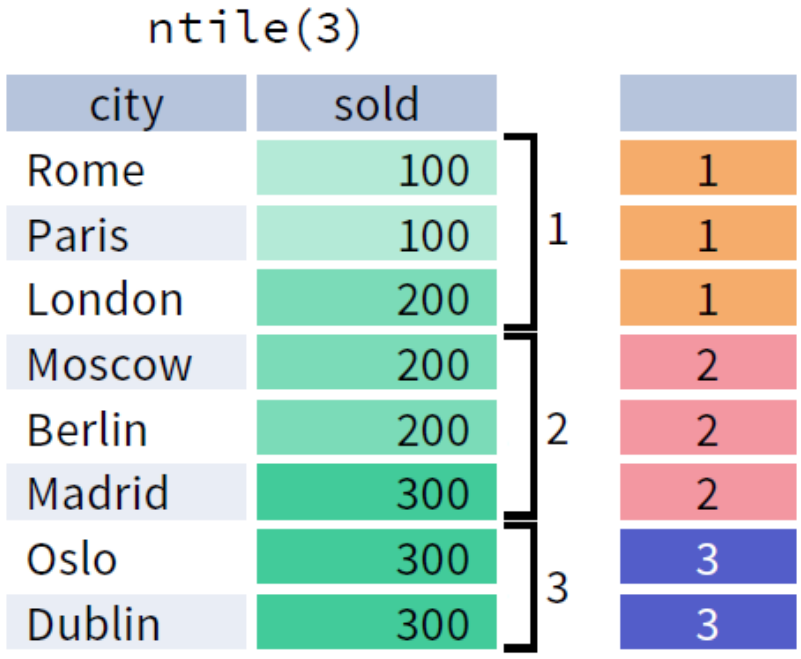
`lead(sold, 2, 0) OVER(ORDER BY month)`



Źródło: https://learnsql.com/blog/sql-window-functions-cheat-sheet/

Funkcje analityczne (2)

- `ntile(n)` - podziel wiersze partycji na *n* równych grup, przypisz każdemu wierszowi numer grupy.



ORDER BY i ramki: `ntile()`, `lead()` i `lag()` wymagają `ORDER BY`. Specyfikacja ramki (`ROWS`, `RANGE`, `GROUPS`) niedozwolona.

Źródło: https://learnsql.com/blog/sql-window-functions-cheat-sheet/

Funkcje analityczne (3)

- `first_value(expr)` - wartość `expr` dla pierwszego rzędu w ramce.
- `last_value(expr)` - wartość `expr` dla ostatniego rzędu w ramce.

```
first_value(sold) OVER
(PARTITION BY city ORDER BY month)
```

city	month	sold	first_value
Paris	1	500	500
Paris	2	300	500
Paris	3	400	500
Rome	2	200	200
Rome	3	300	200
Rome	4	500	200

```
last_value(sold) OVER
(PARTITION BY city ORDER BY month
RANGE BETWEEN UNBOUNDED PRECEDING
AND UNBOUNDED FOLLOWING)
```

city	month	sold	last_value
Paris	1	500	400
Paris	2	300	400
Paris	3	400	400
Rome	2	200	500
Rome	3	300	500
Rome	4	500	500

Uwaga: Zwykle dla `last_value()` chodzi o `RANGE BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING` i trzeba to napisać jeśli jest `ORDER BY` bo domyślna ramka do kończy się na bieżącym wierszu.

Źródło: https://learnsql.com/blog/sql-window-functions-cheat-sheet/

Funkcje analityczne (4)

- `nth_value(expr, n)` - wartość `expr` dla `n`-tego wiersza w ramce

```
nth_value(sold, 2) OVER
(PARTITION BY city ORDER BY month)
```

city	month	sold	nth_value
Paris	1	500	300
Paris	2	300	300
Paris	3	400	300
Rome	2	200	300
Rome	3	300	300
Rome	4	500	300
Rome	5	300	300
London	1	100	NULL

ORDER BY i ramki: `first_value()`, `last_value()` i `nth_value()` nie wymagają `ORDER BY`.
Specyfikacja ramki (`ROWS`, `RANGE`, `GROUPS`) jest dozwolona.

Źródło: https://learnsql.com/blog/sql-window-functions-cheat-sheet/

Funkcje agregujące

- `avg(expr)` - średnia wartość dla wierszy w ramce
- `count(expr)` - liczba wartości dla wierszy w ramce
- `max(expr)` - maksymalna wartość dla wierszy w ramce
- `min(expr)` - minimalna wartość dla wierszy w ramce
- `sum(expr)` - suma wartości dla wierszy w ramce

ORDER BY i ramki: Funkcje agregujące nie wymagają `ORDER BY`. Specyfikacja ramki (`ROWS`, `RANGE`, `GROUPS`) dozwolona.

Źródło: https://learnsql.com/blog/sql-window-functions-cheat-sheet/