SQL - Funkcje okna (Window functions)

Lab 1-2

Imię i nazwisko: Dariusz Piwowarski, Wojciech Przybytek

Celem ćwiczenia jest zapoznanie się z działaniem funkcji okna (window functions) w SQL, analiza wydajności zapytań i porównanie z rozwiązaniami przy wykorzystaniu "tradycyjnych" konstrukcji SQL

Swoje odpowiedzi wpisuj w miejsca oznaczone jako:

```
-- wyniki ...
```

Ważne/wymagane są komentarze.

Zamieść kod rozwiązania oraz zrzuty ekranu pokazujące wyniki, (dołącz kod rozwiązania w formie tekstowej/źródłowej)

Zwróć uwagę na formatowanie kodu

Oprogramowanie - co jest potrzebne?

Do wykonania ćwiczenia potrzebne jest następujące oprogramowanie:

- MS SQL Server - wersja 2019, 2022
- PostgreSQL - wersja 15/16
- SQLite
- Narzędzia do komunikacji z bazą danych
  - SSMS - Microsoft SQL Managment Studio
  - DtataGrip lub DBeaver
- Przykładowa baza Northwind
  - W wersji dla każdego z wymienionych serwerów

Oprogramowanie dostępne jest na przygotowanej maszynie wirtualnej

Dokumentacja/Literatura

- Kathi Kellenberger, Clayton Groom, Ed Pollack, Expert T-SQL Window Functions in SQL Server 2019, Apres 2019
- Itzik Ben-Gan, T-SQL Window Functions: For Data Analysis and Beyond, Microsoft 2020
- Kilka linków do materiałów które mogą być pomocne - https://learn.microsoft.com/en-us/sql/t-sql/queries/select-over-clause-transact-sql?view=sql-server-ver16
  - https://www.sqlservertutorial.net/sql-server-window-functions/
  - https://www.sqlshack.com/use-window-functions-sql-server/
  - https://www.postgresql.org/docs/current/tutorial-window.html
  - https://www.postgresqltutorial.com/postgresql-window-function/
  - https://www.sqlite.org/windowfunctions.html
  - https://www.sqlitetutorial.net/sqlite-window-functions/
- Ikonki używane w graficznej prezentacji planu zapytania w SSMS opisane są tutaj:
  - https://docs.microsoft.com/en-us/sql/relational-databases/showplan-logical-and-physical-operators-reference

Zadanie 1 - obserwacja

Wykonaj i porównaj wyniki następujących poleceń.

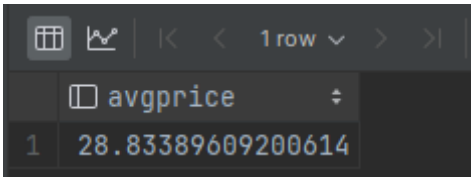
```
select avg(unitprice) avgprice
from products p;

select avg(unitprice) over () as avgprice
from products p;

select categoryid, avg(unitprice) avgprice
from products p
group by categoryid

select avg(unitprice) over (partition by categoryid) as avgprice
from products p;
```

Jaka jest są podobieństwa, jakie różnice pomiędzy grupowaniem danych a działaniem funkcji okna?

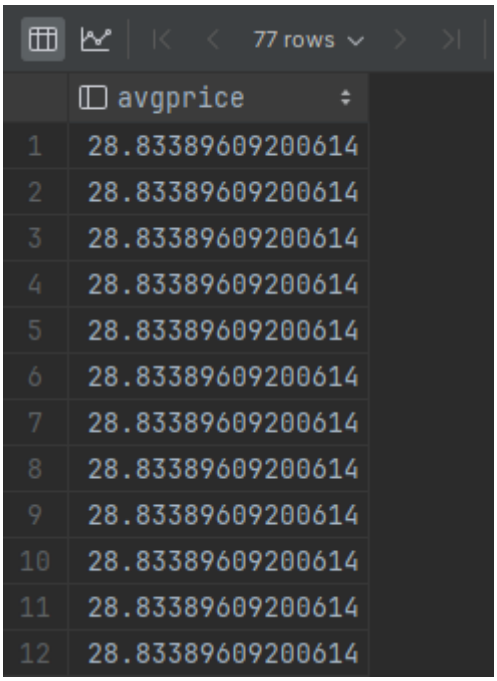
Podobieństwa - oba zwracają poprawne wyniki dla średniej ceny produktów i kategorii

Różnice - Grupowanie zwraca jeden wynik dla wszystkich zgrupowanych rekordów, funkcja okna zwraca wszystkie rekordy wraz z wynikiem

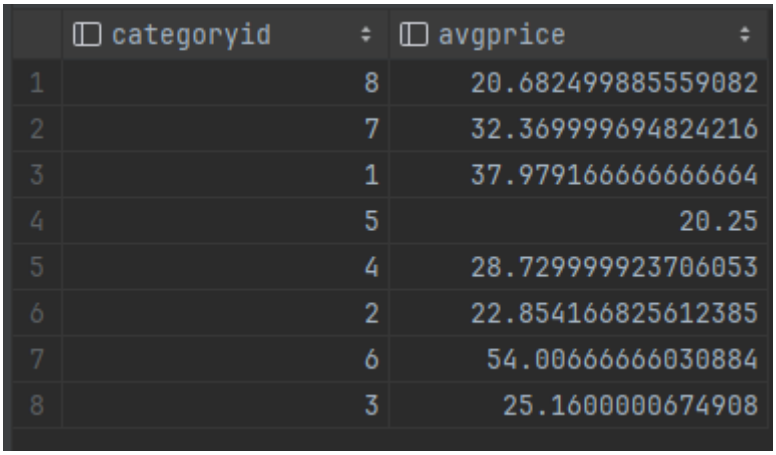
---



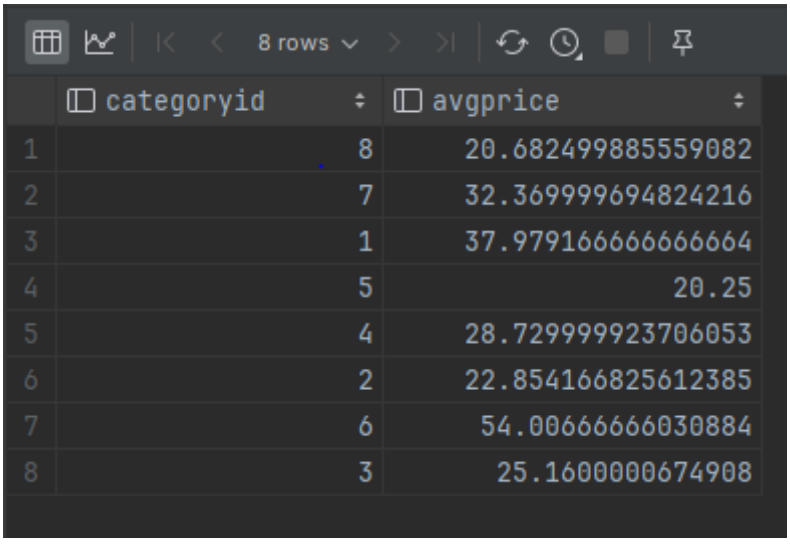
	avgprice
1	28.83389609200614



	avgprice
1	28.83389609200614
2	28.83389609200614
3	28.83389609200614
4	28.83389609200614
5	28.83389609200614
6	28.83389609200614
7	28.83389609200614
8	28.83389609200614
9	28.83389609200614
10	28.83389609200614
11	28.83389609200614
12	28.83389609200614



	categoryid	avgprice
1	8	20.682499885559082
2	7	32.369999694824216
3	1	37.979166666666664
4	5	20.25
5	4	28.729999923706053
6	2	22.854166825612385
7	6	54.00666666030884
8	3	25.1600000674908



	categoryid	avgprice
1	8	20.682499885559082
2	7	32.369999694824216
3	1	37.979166666666664
4	5	20.25
5	4	28.729999923706053
6	2	22.854166825612385
7	6	54.00666666030884
8	3	25.1600000674908

Zadanie 2 - obserwacja

Wykonaj i porównaj wyniki następujących poleceń.

```
--1)

select p.productid, p.ProductName, p.unitprice,
       (select avg(unitprice) from products) as avgprice
from products p
where productid < 10

--2)
select p.productid, p.ProductName, p.unitprice,
       avg(unitprice) over () as avgprice
from products p
where productid < 10
```

Jaka jest różnica? Czego dotyczy warunek w każdym z przypadków? Napisz polecenie równoważne

- i. z wykorzystaniem funkcji okna. Napisz polecenie równoważne
- ii. z wykorzystaniem podzapytania

W 1) wybieramy średnią cenę wszystkich produktów a w 2) średnią cenę produktów o id < 10

i.

```
select p.productid,
       p.ProductName,
       p.unitprice,
       (select avg(unitprice) over () from products limit 1) as avgprice
from products p
where productid < 10;
```

ii.

```
select p.productid, p.ProductName, p.unitprice,
       (select avg(unitprice) from products pr where pr.productid < 10) as avgprice
from products p
where productid < 10;
```

Zadanie 3

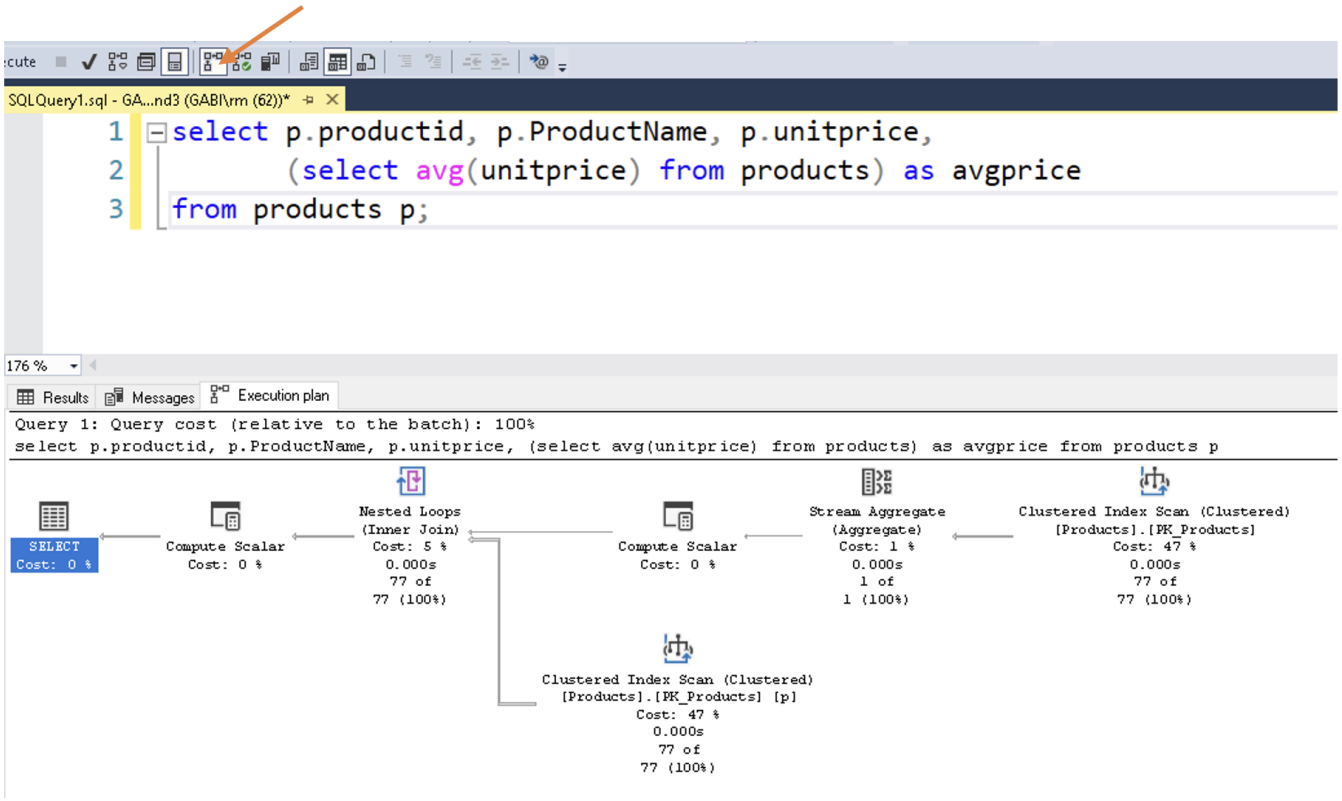
Baza: Northwind, tabela: products

Napisz polecenie, które zwraca: id produktu, nazwę produktu, cenę produktu, średnią cenę wszystkich produktów.

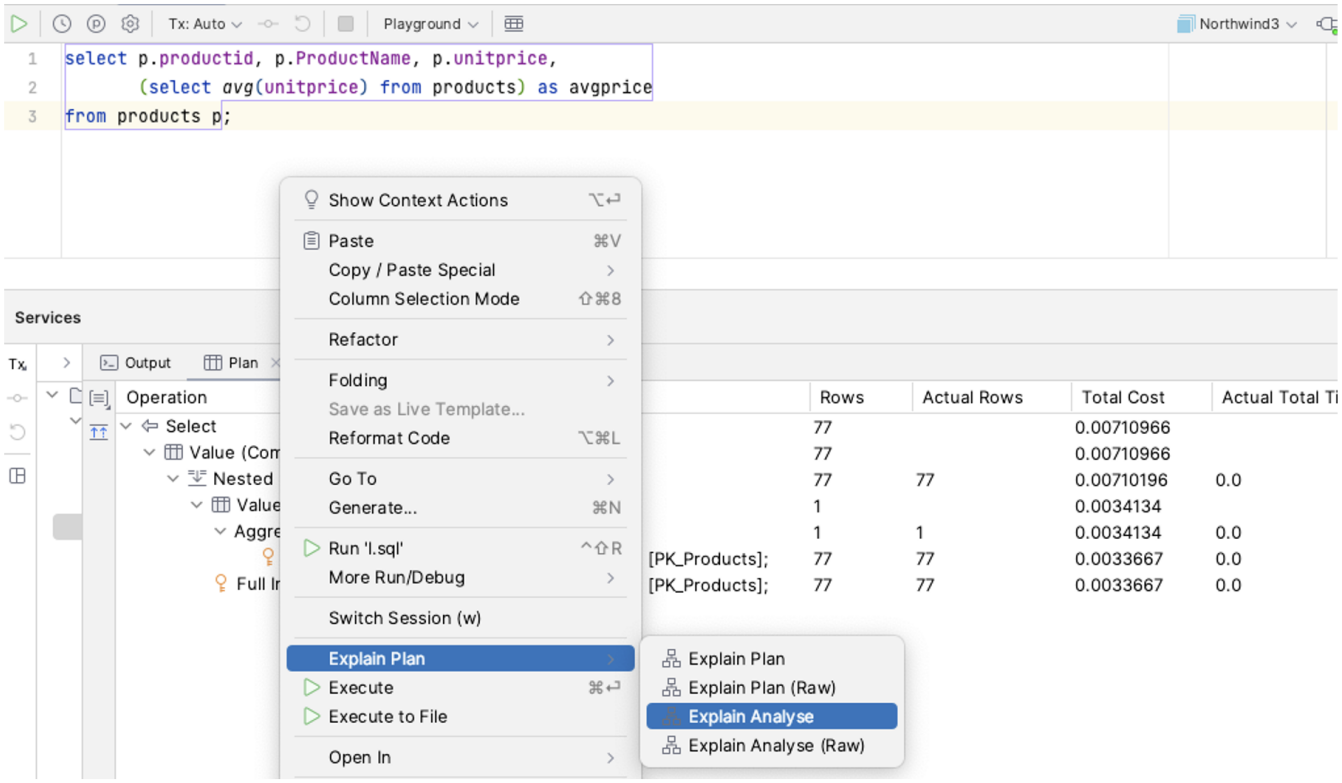
Napisz polecenie z wykorzystaniem z wykorzystaniem podzapytania, join'a oraz funkcji okna. Porównaj czasy oraz plany wykonania zapytań.

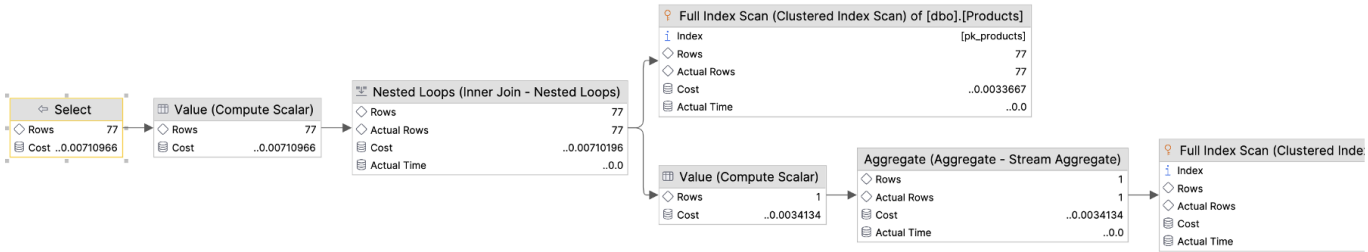
Przetestuj działanie w różnych SZBD (MS SQL Server, PostgreSql, SQLite)

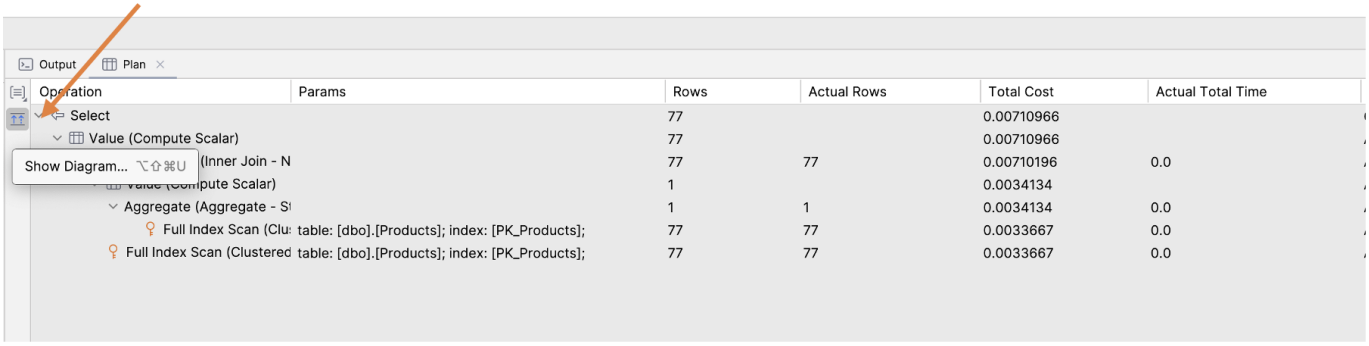
W SSMS włącz dwie opcje: Include Actual Execution Plan oraz Include Live Query Statistics



W DataGrip użyj opcji Explain Plan/Explain Analyze








```
select productid, productname, unitprice,
       (select avg(p.unitprice) from products p) as avg_price
from products;

select p.productid, p.productname, p.unitprice, avg(p2.unitprice) as avg_price
from products p
```

```
cross join products p2
group by p.productid, p.productname, p.unitprice;

select productid, productname, unitprice, avg(unitprice) over () as avg_price
from products

-- POSTGRES:
-- Podzapytanie i funkcja okna mają praktycznie taki sam koszt / czas wykonania, przy czym zap
-- funkcji okna jest krótsze i bardziej czytelne w zapisie.
-- Aby zrobić join musimy użyć CROSS JOIN, bo jak zrobimy zwykłego joina ON p.productid = p2.p
-- to do każdego wiersza dołączymy ten sam wiersz, więc nie damy rady zrobić avg wszystkich pr
-- Z kolei wykonanie CROSS JOINA powoduje że dla n wierszy w kolumnie mamy n^2 operacji
--
-- MS SQL:
-- W przypadku MS SQL koszt wszystskich sposobów jest porównywalny. Nie mamy n^2 w przypadku (
-- Zapytanie z window function ma w planie tylko jedno wykonanie Full Index Scan (który jest r
-- natomiast pozostałe dwa wykonuję Full Index Scan dwa razy i z tego powodu są prawie 2 razy
-- według planu.
--
-- SQLITE:
-- Biorąc pod uwagę ubogi schemat planu w Sqlite wiemy tylko że każde zapytanie wykonuje 2 raz
-- (A przynajmniej znajdują się 2 takie bloki w schemacie) sposób ich ułożenie sugeruje że Sql
-- sobie z CROSS JOINEM i nie zrobił n^2
```

Zadanie 4

Baza: Northwind, tabela products

Napisz polecenie, które zwraca: id produktu, nazwę produktu, cenę produktu, średnią cenę produktów w kategorii, do której należy dany produkt. Wyświetl tylko pozycje (produkty) których cena jest większa niż średnia cena.

Napisz polecenie z wykorzystaniem podzapytania, join'a oraz funkcji okna. Porównaj zapytania. Porównaj czasy oraz plany wykonania zapytań.

Przetestuj działanie w różnych SZBD (MS SQL Server, PostgreSql, SQLite)

```
-- SUBQUERY
select p1.productid,
       p1.productname,
       p1.unitprice,
       (select avg(p2.unitprice) from products p2 where p1.categoryid = p2.categoryid) as avg_
from products p1
where p1.unitprice > (select avg(p2.unitprice) from products p2 where p1.categoryid = p2.categ

-- WINDOW FUNCTION
with products1 as (select productid,
```

```
                       productname,
                       unitprice,
                       avg(unitprice) over (partition by categoryid) as avg_category_price
                from products)

select productid, productname, unitprice, avg_category_price
from products1
where unitprice > avg_category_price;

-- JOIN
with products1 as (select p.productid, p.productname, p.unitprice, avg(p1.unitprice) as avg_ca
                   from products p
                            cross join products p1
                   where p.categoryid = p1.categoryid
                   group by p.productid, p.productname, p.unitprice)

select productid, productname, unitprice, avg_category_price
from products1
where unitprice > avg_category_price

-- W PostgreSQL i SQLite podzapytanie i join wykonuje Full Scan na tabeli products dwukrotnie,
-- okna tylko jeden raz. W PostgresSQL CROSS JOIN powoduje wykonanie pętli w pętli czyli mamy
-- niekorzystne dla kosztu zapytania. W MSSQL wyniki są podobne, ale dla joina wykonywana jest
-- plan wykonania jest identyczny jak w przypadku funkcji okna (czyli bez n^2). Wynika stąd, ż
-- dla powyższego problemu jest zastosowanie funkcji okna, ponieważ daje najlepsze wyniki.
```

Zadanie 5 - przygotowanie

Baza: Northwind

Tabela products zawiera tylko 77 wiersz. Warto zaobserwować działanie na większym zbiorze danych.

Wygeneruj tabelę zawierającą kilka milionów (kilkaset tys.) wierszy

Stwórz tabelę o następującej strukturze:

Skrypt dla SQL Srerver

```
create table product_history(
   id int identity(1,1) not null,
   productid int,
   productname varchar(40) not null,
   supplierid int null,
   categoryid int null,
   quantityperunit varchar(20) null,
   unitprice decimal(10,2) null,
   quantity int,
   value decimal(10,2),
```

```
    date date,
 constraint pk_product_history primary key clustered
    (id asc )
)
```

Wygeneruj przykładowe dane:

Dla 30000 iteracji, tabela będzie zawierała nieco ponad 2mln wierszy (dostostu ograniczenie do możliwości swojego komputera)

Skrypt dla SQL Srerver

```
declare @i int
set @i = 1
while @i <= 30000
begin
    insert product_history
    select productid, ProductName, SupplierID, CategoryID,
         QuantityPerUnit,round(RAND()*unitprice + 10,2),
         cast(RAND() * productid + 10 as int), 0,
         dateadd(day, @i, '1940-01-01')
    from products
    set @i = @i + 1;
end;

update product_history
set value = unitprice * quantity
where 1=1;
```

Skrypt dla Postgresql

```
create table product_history(
   id int generated always as identity not null
       constraint pkproduct_history
            primary key,
   productid int,
   productname varchar(40) not null,
   supplierid int null,
   categoryid int null,
   quantityperunit varchar(20) null,
   unitprice decimal(10,2) null,
   quantity int,
   value decimal(10,2),
   date date
);
```

Wygeneruj przykładowe dane:

Skrypt dla Postgresql

```
do $$
begin
  for cnt in 1..30000 loop
    insert into product_history(productid, productname, supplierid,
           categoryid, quantityperunit,
           unitprice, quantity, value, date)
    select productid, productname, supplierid, categoryid,
           quantityperunit,
           round((random()*unitprice + 10)::numeric,2),
           cast(random() * productid + 10 as int), 0,
           cast('1940-01-01' as date) + cnt
    from products;
  end loop;
end; $$;

update product_history
set value = unitprice * quantity
where 1=1;
```

Wykonaj polecenia: `select count(*) from product_history` , potwierdzające wykonanie zadania

```
-- Obie bazy mają po 2310000 wierszy. Ciekawym spostrzeżeniem jest to, że w przypadku SQL Serv
-- trwało tylko 100 ms, a w przypadku PostgresSQL aż 3 sekundy na Macbook-u z M2, natomiast na
-- te same zapytania zachowywały się w odwrotny sposób. W przypadku SQL Servera zapytanie trwa
-- PostgresSQL tylko 400 ms. Używaliśmy tego samego dockerfile'a, więc różnice w czasie wykona
-- z różnic w implementacji bazy danych dla różnej architektury procesora.
```

Zadanie 6

Baza: Northwind, tabela product_history

To samo co w zadaniu 3, ale dla większego zbioru danych

Napisz polecenie, które zwraca: id pozycji, id produktu, nazwę produktu, cenę produktu, średnią cenę produktów w kategorii do której należy dany produkt. Wyświetl tylko pozycje (produkty) których cena jest większa niż średnia cena.

Napisz polecenie z wykorzystaniem podzapytania, join'a oraz funkcji okna. Porównaj zapytania. Porównaj czasy oraz plany wykonania zapytań.

Przetestuj działanie w różnych SZBD (MS SQL Server, PostgreSql, SQLite)

```
select id, productid, productname, unitprice,
       (select avg(p.unitprice) from product_history p) as avg_price
```

```
from product_history;

select p.id, p.productid, p.productname, p.unitprice, avg(p2.unitprice) as avg_price
from product_history p
cross join product_history p2
group by p.id, p.productid, p.productname, p.unitprice;

select id, productid, productname, unitprice, avg(unitprice) over () as avg_price
from product_history;

-- POSTGRES:
-- Podzapytanie - 2 full scany jeden pełny drugi trochę zoptymalizowany przez postrgresa przez
-- Join - dalej nie zoptymalizował CROSS JOINA i robi n^2, według planu, bo z mierzeniem rzecz
-- jeśli nie mamy kilku dni.
-- Window function - według analizy przechodzi tylko raz full scanem, a jednak jest około 4 ra
-- jeśli chodzi o rzeczywisty czas wykonania.
--
-- MSSQL:
-- Dla podzapytania i window function plan jest podobny jak w w przypadku PostgresSQL, czyli 2
-- oraz 1 full scan dla window function, przy czym tutaj rzeczywisty czas wykonania jest niezr
-- Join natomiast został zoptymalizowany i nie robi n^2, tylko 2 full scany, ale jest trochę w
--
-- SQLITE:
-- W przypadku Sqlite jednak nie ma co się sugerować planem wykonania, bo jest on bardzo ubogi
-- żeby CROSS JOIN robił n^2. W rzeczywistości CROSS JOIN musi robić n^2 bo w skończonej ilośc
-- go wykonać dla 2310000 wierszy. Poza tym subquery i window function mają porównywalne czas
```

Zadanie 7

Baza: Northwind, tabela product_history

Lekka modyfikacja poprzedniego zadania

Napisz polecenie, które zwraca: id pozycji, id produktu, nazwę produktu, cenę produktu oraz

- średnią cenę produktów w kategorii do której należy dany produkt.
- łączną wartość sprzedaży produktów danej kategorii (suma dla pola value)
- średnią cenę danego produktu w roku którego dotyczy dana pozycja
- łączną wartość sprzedaży produktów danej kategorii (suma dla pola value)

Napisz polecenie z wykorzystaniem podzapytania, join'a oraz funkcji okna. Porównaj zapytania. W przypadku funkcji okna spróbuj użyć klauzuli WINDOW.

Porównaj czasy oraz plany wykonania zapytań.

Przetestuj działanie w różnych SZBD (MS SQL Server, PostgreSql, SQLite)

```sql
-- SUBQUERY
select p.id,
       p.productid,
       p.productname,
       p.unitprice,
       (select avg(p1.unitprice) from product_history p1 where p.categoryid = p1.categoryid) a
       (select sum(p1.value) from product_history p1 where p.categoryid = p1.categoryid)     a
       (select avg(p1.unitprice)
        from product_history p1
        where p.productid = p1.productid
          and date_part('year', p.date) = date_part('year', p1.date))                        a
from product_history p;

-- WINDOW FUNCTION
select p.id,
       p.productid,
       p.productname,
       p.unitprice,
       avg(unitprice) over (category_window)                                  as avg_category_p
       sum(value) over (category_window)                                      as sum_category_v
       avg(unitprice) over (partition by productid, date_part('year', date)) as avg_year_price
from product_history p
window category_window as (
        partition by categoryid
        );

-- JOIN
select p.id,
       p.productid,
       p.productname,
       p.unitprice,
       avg(p1.unitprice) as avg_category_price,
       sum(p1.value)     as sum_category_value,
       avg(p2.unitprice) as avg_year_price
from product_history p
         cross join product_history p1
         cross join product_history p2
where p.categoryid = p1.categoryid
  and p.productid = p2.productid
  and date_part('year', p.date) = date_part('year', p2.date)
group by p.id, p.productid, p.productname, p.unitprice;

/*
Plan wykonania dla poszczególnych typów zapytań nie różnił się zbytnio
pomiędzy poszczególnymi bazami danych. Najgorzej wypadało zawsze zapytanie
z joinem, liczba przetworzonych wierszy to między n^2 a n^3, planer szacuje,
że będzie ich około 10^14. Nieco lepiej wypada wersja z podzapytaniem, tu liczba
przetworzonych wierszy jest rzędu n^2, około 10^11. Zdecydowanie najlepiej wypada
wersja z funkcją okna, Full Scan wykonywany jest tam tylko raz.
 */
```

Zadanie 8 - obserwacja

Funkcje rankingu, `row_number()` , `rank()` , `dense_rank()`

Wykonaj polecenie, zaobserwuj wynik. Porównaj funkcje row_number(), rank(), dense_rank()

```
select productid, productname, unitprice, categoryid,
    row_number() over(partition by categoryid order by unitprice desc) as rowno,
    rank() over(partition by categoryid order by unitprice desc) as rankprice,
    dense_rank() over(partition by categoryid order by unitprice desc) as denserankprice
from products;
```

```
-- Polecenie row_number() zwraca w obrębie każdej kategorii numer wiersza, posortowany po ceni
-- W przypadku remisów, funkcja row_number() zwraca kolejne numery wierszy w sposób arbitralny
-- kolejnych numerów ile jest wierszy w danej kategorii i każdy numer występuje tylko raz.
-- Polecenie rank() zwraca w obrębie każdej kategorii ranking ceny produktu, posortowany po ce
-- Różni się od row_number() tym, że przypadku remisów, funkcja rank() zwraca ten sam numer dl
-- samej cenie, a następny numer po serii remisów jest taki jakby był jego row_number(), czyli
-- Polecenie dense_rank() różni się od rank() tym, że nie ma przerw w numeracji, czyli jeśli m
-- ten sam numer, ale po serii remisów kolejny numer jest o 1 większy niż poprzedni, czyli np.
```

Zadanie

Spróbuj uzyskać ten sam wynik bez użycia funkcji okna

```
select productid,
       productname,
       unitprice,
       categoryid,
       (select count(*) + 1
        from products p2
        where p2.categoryid = p.categoryid
          and (p2.unitprice > p.unitprice or (p2.unitprice = p.unitprice and p2.productid < p.
       (select count(*) + 1
        from products p2
        where p2.categoryid = p.categoryid
          and p2.unitprice > p.unitprice)
       (select count(distinct p2.unitprice) + 1
        from products p2
        where p2.categoryid = p.categoryid
          and p2.unitprice > p.unitprice)
from products p
order by p.unitprice desc;
```

Zadanie 9

Baza: Northwind, tabela product_history

Dla każdego produktu, podaj 4 najwyższe ceny tego produktu w danym roku. Zbiór wynikowy powinien zawierać:

- rok
- id produktu
- nazwę produktu
- cenę
- datę (datę uzyskania przez produkt takiej ceny)
- pozycję w rankingu

Uporządkuj wynik wg roku, nr produktu, pozycji w rankingu

```
with ranking as (select date_part('year', date) as year,
                        productid,
                        productname,
                        unitprice,
                        date,
                        dense_rank() over (partition by productid, date_part('year', date) orc
                 from product_history)
select *
from ranking
where rank < 5
order by year, productid, rank
```

Spróbuj uzyskać ten sam wynik bez użycia funkcji okna, porównaj wyniki, czasy i plany zapytań. Przetestuj działanie w różnych SZBD (MS SQL Server, PostgreSql, SQLite)

```
with ranking as (select date_part('year', date)            as year,
                        productid,
                        productname,
                        unitprice,
                        date,
                        (select count(distinct p2.unitprice) + 1
                         from product_history p2
                         where p2.productid = p.productid
                           and date_part('year', p2.date) = date_part('year', p.date)
                           and p2.unitprice > p.unitprice) as rank
                 from product_history p)
select *
from ranking
where rank < 5
order by year, productid, rank;
```

```
-- Jeśli chodzi o PostgresSQL to zapytanie z funkcją okna ma sensowny czas wykonania (około 10
-- z podzapytaniem nie udało mi się w skończonym czasie wykonać nawet dla pojedynczego product
-- Po analizie planu wykonania zapytania z podzapytaniem, okazało się że jest ono bardzo koszt
-- window function koszt to jedynie 1185340). Z grafu planu zapytania oraz wartości kosztu wni
-- zoptymalizował go i wykonuje ~n^2 operacji.
-- W MSSQL jest podobnie, ale zapytanie z funkcją okna wykonuje się w około 3 sekund, natomias
-- znowu nie udało się wykonać. Analiza planu wykonania ponownie wskazuje na nested loop join,
-- W przypadku Sqlite zapytanie z funkcją okna podobnie jak w MSSQL wykonuje się w około 3 sek
-- się nie wykonało w sensownym czasie. Niestety tutaj plan wykonania jest dość ubogi więc cie
-- jest tak kosztowne, ale prawdopodobnie z tych samych przyczyn co w przypadku PostgresSQL i
```

Zadanie 10 - obserwacja

Funkcje `lag()` , `lead()`

Wykonaj polecenia, zaobserwuj wynik. Jak działają funkcje `lag()` , `lead()`

```sql
select productid, productname, categoryid, date, unitprice,
       lag(unitprice) over (partition by productid order by date)
as previousprodprice,
       lead(unitprice) over (partition by productid order by date)
as nextprodprice
from product_history
where productid = 1 and year(date) = 2022
order by date;

with t as (select productid, productname, categoryid, date, unitprice,
                  lag(unitprice) over (partition by productid
order by date) as previousprodprice,
                  lead(unitprice) over (partition by productid
order by date) as nextprodprice
           from product_history
           )
select * from t
where productid = 1 and year(date) = 2022
order by date;



/*
Funkcja 'lag(x)' zwraca wartość kolumny x poprzedniego rekordu w kolejności, a dla pierwszego
Funkcja 'lead(x)' zwraca wartość kolumny x następnego rekordu w kolejności, a dla ostatnego re
 */
```

Zadanie

Spróbuj uzyskać ten sam wynik bez użycia funkcji okna, porównaj wyniki, czasy i plany zapytań. Przetestuj działanie w różnych SZBD (MS SQL Server, PostgreSql, SQLite)

```
-- POSTGRESQL VERSION
select p.productid,
       p.productname,
       p.categoryid,
       p.date,
       p.unitprice,
       (select p1.unitprice
        from product_history p1
        where p1.productid = 1
          and date_part('year', p1.date) = 2022
          and p1.date < p.date
        order by p1.date desc
        limit 1)
           as previousprodprice,
       (select p1.unitprice
        from product_history p1
        where p1.productid = 1
          and date_part('year', p1.date) = 2022
          and p1.date > p.date
        order by p1.date
        limit 1)
           as nextprodprice
from product_history p
where p.productid = 1
  and date_part('year', p.date) = 2022
order by p.date;

/*
Plany wykonania dla wszystkich baz danych są podobne. Zapytanie bez funkcji lag i lead wykorzy
wykonuje liczbe operacji rzędu n^2. Znacznie lepiej prezentuje się zapytanie wykorzystujące fu
wykonywany jest tylko jeden Full Scan, co jest zauważalne w dużo szybszym czasie wykonania.
 */
```

Zadanie 11

Baza: Northwind, tabele customers, orders, order details

Napisz polecenie które wyświetla inf. o zamówieniach

Zbiór wynikowy powinien zawierać:

- nazwę klienta, nr zamówienia,
- datę zamówienia,
- wartość zamówienia (wraz z opłatą za przesyłkę),

- nr poprzedniego zamówienia danego klienta,
- datę poprzedniego zamówienia danego klienta,
- wartość poprzedniego zamówienia danego klienta.

```
with order_values as (select c.companyname,
                             o.customerid,
                             o.orderid,
                             o.orderdate,
                             round((sum((od.unitprice * od.quantity) * (1 - od.discount)) + o.
                                   2) as order_total
                      from orders o
                               inner join orderdetails od on o.orderid = od.orderid
                               inner join customers c on o.customerid = c.customerid
                      group by o.orderid, c.customerid)

select companyname,
       orderid,
       orderdate,
       order_total,
       lag(orderid) over other_customer_orders as prev_order_id,
       lag(orderdate) over other_customer_orders as prev_order_date,
       lag(order_total) over other_customer_orders as prev_order_value
from order_values
window other_customer_orders as (partition by customerid order by orderdate);
```

Zdjęcie tabeli wynikowej, aby udowodnić poprawność zapytania:

	companyname	orderid	orderdate	order_value	prev_order_id	prev_order_date	prev_order_value
1	Alfreds Futterkiste	10643	1997-08-25	843.96	<null>	<null>	<null>
2	Alfreds Futterkiste	10692	1997-10-03	939.02	10643	1997-08-25	843.96
3	Alfreds Futterkiste	10702	1997-10-13	353.94	10692	1997-10-03	939.02
4	Alfreds Futterkiste	10835	1998-01-15	915.33	10702	1997-10-13	353.94
5	Alfreds Futterkiste	10952	1998-03-16	511.62	10835	1998-01-15	915.33
6	Alfreds Futterkiste	11011	1998-04-09	934.71	10952	1998-03-16	511.62
7	Ana Trujillo Emparedados y helados	10308	1996-09-18	90.41	<null>	<null>	<null>
8	Ana Trujillo Emparedados y helados	10625	1997-08-08	523.65	10308	1996-09-18	90.41
9	Ana Trujillo Emparedados y helados	10759	1997-11-28	331.99	10625	1997-08-08	523.65
10	Ana Trujillo Emparedados y helados	10926	1998-03-04	554.32	10759	1997-11-28	331.99
11	Antonio Moreno Taquería	10365	1996-11-27	425.2	<null>	<null>	<null>
12	Antonio Moreno Taquería	10507	1997-04-15	796.51	10365	1996-11-27	425.2
13	Antonio Moreno Taquería	10535	1997-05-13	1956.49	10507	1997-04-15	796.51
14	Antonio Moreno Taquería	10573	1997-06-19	2166.84	10535	1997-05-13	1956.49
15	Antonio Moreno Taquería	10677	1997-09-22	817.39	10573	1997-06-19	2166.84
16	Antonio Moreno Taquería	10682	1997-09-25	411.63	10677	1997-09-22	817.39
17	Antonio Moreno Taquería	10856	1998-01-28	718.43	10682	1997-09-25	411.63

Zadanie 12 - obserwacja

Funkcje `first_value()` , `last_value()`

Wykonaj polecenia, zaobserwuj wynik. Jak działają funkcje `first_value()` , `last_value()` . Skomentuj uzyskane wyniki. Czy funkcja `first_value` pokazuje w tym przypadku najdroższy produkt w danej kategorii, czy funkcja `last_value()` pokazuje najtańszy produkt? Co jest przyczyną takiego działania funkcji `last_value` . Co trzeba zmienić żeby funkcja last_value pokazywała najtańszy produkt w danej kategorii

```
select productid, productname, unitprice, categoryid,
    first_value(productname) over (partition by categoryid
order by unitprice desc) first,
    last_value(productname) over (partition by categoryid
order by unitprice desc) last
from products
order by categoryid, unitprice desc;
```

Funkcje okna przyjmują parametr `frame_clause`. Jak możemy przeczytać np. w dokumentacji PostgreSQL:

> The default framing option is RANGE UNBOUNDED PRECEDING, which is the same as RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW. With ORDER BY, this sets the frame to be all rows from the partition start up through the current row's last ORDER BY peer.

W podanym przykładzie oznacza to, że ramka dla danego rekordu zawiera wszystkie rekordy z tej samej kategorii od pierwszego z najwyższą ceną do ostatniego z ceną równej cenie tego rekordu. Dlatego funkcja `first_value()` zwróci najdroższy produkt w danej kategorii a `last_value()` dla każdego rekordu zwróci ostatni w kolejności rekord o tej samej cenie. Aby funkcja `last_value()` zwróciła najtańszy rekord z kategorii należy zmodyfikować domyślną ramkę:

```
select productid,
       productname,
       unitprice,
       categoryid,
       first_value(productname) over (partition by categoryid order by unitprice desc) first,
       last_value(productname)
       over (partition by categoryid order by unitprice desc rows between unbounded preceding
from products
order by categoryid, unitprice desc;
```

Zadanie

Spróbuj uzyskać ten sam wynik bez użycia funkcji okna, porównaj wyniki, czasy i plany zapytań. Przetestuj działanie w różnych SZBD (MS SQL Server, PostgreSql, SQLite)

```
select productid,
       productname,
       unitprice,
       categoryid,
       (select productname
        from products p1
        where p.categoryid = p1.categoryid
        order by p1.unitprice desc
        limit 1) first,
       (select productname
        from products p1
        where p.categoryid = p1.categoryid
```

```
          and p.unitprice = p1.unitprice
        order by p1.unitprice
        limit 1) last
from products p
order by categoryid, unitprice desc;

/*
Wynik porównania jest podobny co w poprzednich przypadkach, wybór SZDB nie ma większego wpływu
na plan wykonania, zapytanie bez funkcji okna ma złożoność n^2, a z funkcją okna tylko n. Stąd
iż użycie funkcji okna do takiego przypdaku jest lepszym rozwiązaniem.
 */
```

Zadanie 13

Baza: Northwind, tabele orders, order details

Napisz polecenie które wyświetla inf. o zamówieniach

Zbiór wynikowy powinien zawierać:

- Id klienta,
- nr zamówienia,
- datę zamówienia,
- wartość zamówienia (wraz z opłatą za przesyłkę),
- dane zamówienia klienta o najniższej wartości w danym miesiącu
  - nr zamówienia o najniższej wartości w danym miesiącu
  - datę tego zamówienia
  - wartość tego zamówienia
- dane zamówienia klienta o najwyższej wartości w danym miesiącu
  - nr zamówienia o najniższej wartości w danym miesiącu
  - datę tego zamówienia
  - wartość tego zamówienia

```
with order_values as (select o.customerid,
                             o.orderid,
                             o.orderdate,
                             round((sum((od.unitprice * od.quantity) * (1 - od.discount)) + o.
                                   2) as order_total
                      from orders o
                               inner join orderdetails od on o.orderid = od.orderid
                      group by o.orderid)

select customerid,
       orderid,
       orderdate,
```

```
       order_total,
       first_value(orderid) over asc_monthly_orders      as lowest_total_monthly_id,
       first_value(orderdate) over asc_monthly_orders    as lowest_total_monthly_date,
       first_value(order_total) over asc_monthly_orders  as lowest_total_monthly_value,
       first_value(orderid) over desc_monthly_orders     as highest_total_monthly_id,
       first_value(orderdate) over desc_monthly_orders   as highest_total_monthly_date,
       first_value(order_total) over desc_monthly_orders as highest_total_monthly_value
from order_values
window asc_monthly_orders as ( partition by customerid, date_part('year', orderdate), date_par
        order by order_total ),
       desc_monthly_orders as ( partition by customerid, date_part('year', orderdate), date_pa
              order by order_total desc );
```

Wynik zapytania

	customerid	orderid	orderdate	order_total	lowest_monthly_id	lowest_monthly_date	lowest_monthly_value	highest_monthly_id	highest_monthly_date	highest_monthly_value
1	ALFKI	10643	1997-08-25	843.96	10643	1997-08-25	843.96	10643	1997-08-25	843.96
2	ALFKI	10702	1997-10-13	353.94	10702	1997-10-13	353.94	10692	1997-10-03	939.02
3	ALFKI	10692	1997-10-03	939.02	10702	1997-10-13	353.94	10692	1997-10-03	939.02
4	ALFKI	10835	1998-01-15	915.33	10835	1998-01-15	915.33	10835	1998-01-15	915.33
5	ALFKI	10952	1998-03-16	511.62	10952	1998-03-16	511.62	10952	1998-03-16	511.62
6	ALFKI	11011	1998-04-09	934.71	11011	1998-04-09	934.71	11011	1998-04-09	934.71
7	ANATR	10308	1996-09-18	90.41	10308	1996-09-18	90.41	10308	1996-09-18	90.41
8	ANATR	10625	1997-08-08	523.65	10625	1997-08-08	523.65	10625	1997-08-08	523.65
9	ANATR	10759	1997-11-28	331.99	10759	1997-11-28	331.99	10759	1997-11-28	331.99
10	ANATR	10926	1998-03-04	554.32	10926	1998-03-04	554.32	10926	1998-03-04	554.32
11	ANTON	10365	1996-11-27	425.2	10365	1996-11-27	425.2	10365	1996-11-27	425.2
12	ANTON	10507	1997-04-15	796.51	10507	1997-04-15	796.51	10507	1997-04-15	796.51

Zadanie 14

Baza: Northwind, tabela product_history

Napisz polecenie które pokaże wartość sprzedaży każdego produktu narastająco od początku każdego miesiąca. Użyj funkcji okna

Zbiór wynikowy powinien zawierać:

- id pozycji
- id produktu
- datę
- wartość sprzedaży produktu w danym dniu
- wartość sprzedaży produktu narastające od początku miesiąca

```
select id,
       productid,
       date,
       sum(value) over (partition by productid, date)
       sum(value)
       over (partition by productid, date_part('year', date), date_part('month', date) order b
from product_history
order by productid, date;
```

Wynik zapytania

	id	productid	date	daily_value	monthly_value_to_date
1	1	1	1940-01-02	158.5	158.5
2	78	1	1940-01-03	256.7	415.2
3	155	1	1940-01-04	274.2	689.4
4	232	1	1940-01-05	140.3	829.7
5	309	1	1940-01-06	265.8	1095.5
6	386	1	1940-01-07	202.5	1298
7	463	1	1940-01-08	212.3	1510.3
8	540	1	1940-01-09	187.4	1697.7
9	617	1	1940-01-10	233.1	1930.8
10	694	1	1940-01-11	210.2	2141
11	771	1	1940-01-12	178.7	2319.7
12	848	1	1940-01-13	125.5	2445.2

Spróbuj wykonać zadanie bez użycia funkcji okna. Spróbuj uzyskać ten sam wynik bez użycia funkcji okna, porównaj wyniki, czasy i plany zapytań. Przetestuj działanie w różnych SZBD (MS SQL Server, PostgreSql, SQLite)

```
select id,
       productid,
       date,
       (select sum(ph1.value)
        from product_history ph1
        where ph.productid = ph1.productid
          and ph.date = ph1.date) as daily_value,
       (select sum(ph1.value)
        from product_history ph1
        where ph.productid = ph1.productid
          and date_part('year', ph1.date) = date_part('year', ph.date)
          and date_part('month', ph1.date) = date_part('month', ph.date)
          and ph1.date <= ph.date) as monthly_value_to_date
from product_history ph
order by productid, date;

/*
W każdym z SZBD wykonanie zapytania skutkuje wykonaniem dwóch zagnieżdżonych Full Scanów tabel
od 3 do 5 minut. Jest to dużo gorszy wynik niż w przypadku funkcji okna, tu wykonywany jest je
a całośc trwa zaledwie kilka sekund.
 */
```

Zadanie 15

Wykonaj kilka "własnych" przykładowych analiz. Czy są jeszcze jakieś ciekawe/przydatne funkcje okna (z których nie korzystałeś w ćwiczeniu)? Spróbuj ich użyć w zaprezentowanych przykładach.

```
-- Łącząc ze sobą funkcje lag i max można wykonać zapytanie które dla każdego produktu zwróci
-- w których cena produktu była najwyższa (wraz z tą ceną). Dla danych w product_history wyger
-- te okresy wynoszą zawsze 1 dzień ale dla rzeczywistych zbiorów danych takie zapytanie mogło

with p as (select productid,
                  productname,
                  lag(date) over (partition by productid order by date) + 1 as date_from,
                  date                                                      as date_to,
                  unitprice,
                  max(unitprice) over (partition by productid)              as maxprice
           from product_history
           order by productid)
select productid, productname, date_from, date_to, unitprice from p
where p.unitprice = p.maxprice;



-- Funkcja która nie była jeszcze użyta w ćwiczeniu to nth_value. Funkcja ta przyjmuje dwa arg
-- z której chcemy pobrać wartość, a drugi to numer wiersza w ramce okna. Korzystając z tej fu
-- zapytanie z zadania 13 tak by zwracała:
-- - Id klienta,
-- - nr zamówienia,
-- - datę zamówienia,
-- - wartość zamówienia (wraz z opłatą za przesyłkę),
-- - dane zamówienia klienta o najwyższej wartości w danym miesiącu
--      - nr tego zamówienia
--      - datę tego zamówienia
--      - wartość tego zamówienia
-- - dane zamówienia klienta o drugiej najwyższej wartości w danym miesiącu
--      - nr tego zamówienia
--      - datę tego zamówienia
--      - wartość tego zamówienia

select customerid,
       orderid,
       orderdate,
       order_total,
       first_value(orderid) over desc_monthly_orders      as highest_total_monthly_id,
       first_value(orderdate) over desc_monthly_orders    as highest_total_monthly_date,
       first_value(order_total) over desc_monthly_orders  as highest_total_monthly_value,
       nth_value(orderid, 2) over desc_monthly_orders     as second_highest_total_monthly_id,
       nth_value(orderdate, 2) over desc_monthly_orders   as second_highest_total_monthly_date
       nth_value(order_total, 2) over desc_monthly_orders as second_highest_total_monthly_valu
from order_values
window desc_monthly_orders as ( partition by customerid, date_part('year', orderdate), date_pa
        order by order_total desc );



-- Kolejne dwie funkcje które nie były użyte w ćwiczeniu to percent_rank i cume_dist. Funkcja
-- zwraca percentyl dla danego wiersza. Funkcja cume_dist zwraca natomiasto wartość kumulatywn
-- danego wiersza. Funkcje zwracająca percentyl można wykorzystać na przykład do określenia wy
-- gdzie oprócz wyniku mamy też właśnie podany percentyl. Funkcję możemy porównać z funkcjami
-- row_number z ćwiczenia 8.
```

```sql
select productid, productname, unitprice, categoryid,
    row_number() over(desc_price_by_category) as rowno,
    rank() over(desc_price_by_category) as rankprice,
    dense_rank() over(desc_price_by_category) as denserankprice,
    percent_rank() over(desc_price_by_category) as percentrankprice,
    cume_dist() over(desc_price_by_category) as cumedistprice
from products
window desc_price_by_category as (partition by categoryid order by unitprice desc);
```

```sql
-- Ostatnia funkcja której jeszcze nie użyliśmy to ntile. Funkcja ta przyjmuje jeden argument
-- dzieli wiersze wewnątrz okna na n możliwie równych grup i zwraca numer grupy do której nale
-- Naturalnym zastosowaniem takiej funkcji jest podzielenie studentów z danego roku na grupy.
-- mamy do dyspozycji bazę Northwind to możemy podzielić klientów z danego kraju na 2 grupy.

select companyname, country, ntile(2) over (partition by country)
from customers;
```

Punktacja

zadanie	pkt
1	0,5
2	0,5
3	1
4	1
5	0,5
6	2
7	2
8	0,5
9	2
10	1
11	2
12	1
13	2
14	2

15	2
razem	20