Lab_01 Report

Jakub Barber

09.03.2025

Konfiguracja środowiska

Projekt konfiguruje trzy systemy bazodanowe - PostgreSQL, Microsoft SQL Server oraz SQLite - przy użyciu kontenerów Docker, a ich konfiguracja jest zrealizowana przy pomocy docker-compose.yml. Każda baza jest automatycznie inicjalizowana przykładową bazą danych Northwind. Konfiguracja oparta jest na Docker Compose, a zadania (setup i clear dla poszczególnych baz) są definiowane w pliku Taskfile.yml. Szczegółowe informacje, w tym dane połączenia (host, port, nazwa bazy, użytkownik, hasło), struktura plików oraz instrukcje rozwiązywania problemów, znajdują się w pliku README.md. Do uruchomienia projektu wymagane jest posiadanie zainstalowanego Docker oraz Docker Compose. Automatyczny setup baz danych bazą Northwind jest realizowany przy pomocy skryptów uruchamianych przy pomocy task.

Plik docker-compose.yml

```
version: '3.8'

services:
  mssql-db:
    image: mcr.microsoft.com/azure-sql-edge:latest
    platform: linux/arm64
    environment:
      - ACCEPT_EULA=Y
      - MSSQL_SA_PASSWORD=lab_01_password
    ports:
      - "1433:1433"
    volumes:
      - mssql_data:/var/opt/mssql

  mssql-tools:
    image: mcr.microsoft.com/mssql-tools
    volumes:
      - ./Northwind_mssql:/mssql_init
```

```
    depends_on:
      - mssql-db

  postgres-db:
    image: postgres:16-alpine
    environment:
      - POSTGRES_PASSWORD=lab01_password
      - POSTGRES_USER=lab01_user
      - POSTGRES_DB=lab01_db
    ports:
      - "5432:5432"
    volumes:
      - postgres_data:/var/lib/postgresql/data
      - ./Northwind_psql:/psql_init

  sqlite-db:
    image: keinos/sqlite3:latest
    volumes:
      - ./sqlite_data:/data
      - ./Northwind_sqlite:/sqlite_init
    command: [ "sh", "-c", "sqlite3␣/data/lab01.db␣'.
       databases'␣&&␣tail␣-f␣/dev/null" ]

volumes:
  mssql_data:
  postgres_data:
```

Zadanie 1

Poniżej analiza zapytanń SQL dotyczących obliczania średniej oraz omówienie rezultatów:

1. **Zapytanie:**

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

   Rezultat: To zapytanie wykorzystuje funkcję agregującą `AVG()` bez funkcji okna czy grupowania. Wynikiem jest jeden wiersz zawierający średnią cenę dla całej tabeli `products`.

2. **Zapytanie:**

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

Rezultat: Użycie funkcji okna (`OVER()`) powoduje, że dla każdego wiersza tabeli zostanie obliczona ta sama wartość - średnia cena dla całej tabeli. Rezultatem jest tyle wierszy, ile jest rekordów w tabeli, przy czym każda z nich zawiera tę samą wartość średniej. **(77 wierszy w przypadku bazy Northwind)**

3. **Zapytanie:**

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

Rezultat: To zapytanie grupuje dane według `categoryid` i oblicza średnią cenę jednostkową dla każdej kategorii. Wynikiem jest jeden wiersz na każdą kategorię z przypisaną wartością średniej. **(8 wierszy w przypadku bazy Northwind)**

4. **Zapytanie:**

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

Rezultat: W tym przypadku funkcja okna z klauzulą `PARTITION BY` dzieli dane na grupy według `categoryid`, ale nie redukuje liczby wierszy. Każdy wiersz tabeli otrzymuje wartość średniej ceny jednostkowej swojej kategorii - efekt jest podobny do poprzedniego zapytania, jednak zamiast jednego wiersza na kategorię, otrzymujemy wszystkie rekordy, z których każdy zawiera średnią dla swojej grupy. **(77 wierszy w przypadku bazy Northwind)**

Wszystkie powyższe zapytania wykonałem dla każdej z baz danych. Wyniki w każdym przypadku były takie same i zgodne z oczekiwaniami - ilość wierszy oraz wyniki zapytań pokrywały się. Jedyną różnicą była dokładność wyświetlanych wyników - dla bazy mssql wyniki były wyświetlane z mniejszą dokładnością (4 miejsca po przecinku), w porównaniu do pozostałych baz (kilkanaście miejsc po przecinku).

Zadanie 2

W ramach tego zadania porównano dwa zapytania:

```
-- Zapytanie 1
SELECT p.productid, p.ProductName, p.unitprice, (
   SELECT AVG(unitprice) FROM products) AS avgprice
FROM products p
WHERE productid < 10;
```

```
-- Zapytanie 2
SELECT p.productid, p.ProductName, p.unitprice, AVG(
   unitprice) OVER () AS avgprice
FROM products p
WHERE productid < 10;
```

Różnice między zapytaniami:

- **Zapytanie 1:** Wykorzystuje podzapytanie skalarnie, które oblicza średnią cenę jednostkową (`unitprice`) dla wszystkich produktów w tabeli `products`. Wynik tego podzapytania jest identyczny dla każdego wiersza wyniku głównego zapytania.
- **Zapytanie 2:** Używa funkcji okna `AVG` z klauzulą `OVER ()`, która również oblicza średnią cenę jednostkową dla wszystkich produktów. W tym przypadku średnia jest obliczana jako funkcja okna, ale bez podziału na partycje, co skutkuje tym samym wynikiem dla każdego wiersza.

Zakres działania warunku `WHERE`:

W obu zapytaniach warunek `WHERE productid < 10` ogranicza zestaw wyników do produktów o `productid` mniejszym niż 10. Jednak:

- **W Zapytaniu 1:** Podzapytanie obliczające średnią cenę jednostkową (`AVG(unitprice)`) jest niezależne od warunku `WHERE` w zapytaniu głównym. Oznacza to, że średnia jest obliczana na podstawie wszystkich produktów w tabeli, niezależnie od tego, czy spełniają one warunek `productid < 10`.
- **W Zapytaniu 2:** Funkcja okna `AVG(unitprice) OVER ()` jest obliczana po zastosowaniu warunku `WHERE`, co oznacza, że średnia cena jednostkowa jest obliczana tylko dla produktów o `productid` mniejszym niż 10.

Zapytanie 1 z wykorzystaniem funkcji okna:

```
SELECT *
FROM (
  SELECT p.productid, p.ProductName, p.unitprice,
    AVG(unitprice) OVER () AS avgprice
  FROM products p
) subquery
WHERE productid < 10;
```

Zapytanie 2 z wykorzystaniem podzapytania:

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

Wyniki powyższych zapytań są identyczne z ich wcześniejszymi odpowiednikami. W przypadku przepisania 1 zapytania przy pomocy funkicji okna, użyłem podzapytania, które następnie filtruje wyniki przy pomocy warunku `WHERE`.

Zadanie 3

Poniżej znajdują się trzy różne podejścia do uzyskania rezultatu zadania: przy użyciu podzapytania, JOIN-a oraz funkcji okna.

1. **Zapytanie z wykorzystaniem podzapytania**

Listing 1: Subquery

```
SELECT p.productid, p.ProductName, p.unitprice,
       (SELECT AVG(unitprice) FROM products) AS
          avgprice
FROM products p;
```

2. **Zapytanie z wykorzystaniem JOIN-a**

Listing 2: JOIN

```
SELECT p.productid, p.ProductName, p.unitprice, ap
   .avgprice
FROM products p
JOIN (SELECT AVG(unitprice) AS avgprice FROM
   products) ap
    ON 1=1;
```

3. **Zapytanie z wykorzystaniem funkcji okna**

Listing 3: Funkcja okna

```
SELECT p.productid, p.ProductName, p.unitprice,
       AVG(unitprice) OVER () AS avgprice
FROM products p;
```

Następnie porównałem wyniki operacji Explain Analyze dla każdego z tych zapytań dla każdej z bas danych:

MSSQL

W poniższej sekcji przedstawiamy wyniki `EXPLAIN PLAN` dla zapytania w systemie bazodanowym MSSQL.

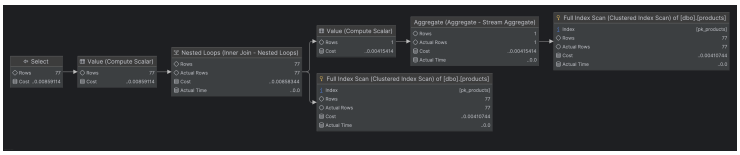

Figure 1: Plan zapytania dla MSSQL - zapytanie z subquery

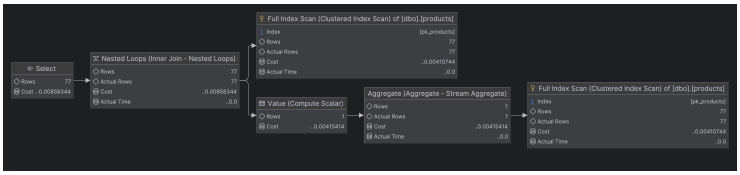

Figure 2: Plan zapytania dla MSSQL - zapytanie z JOIN

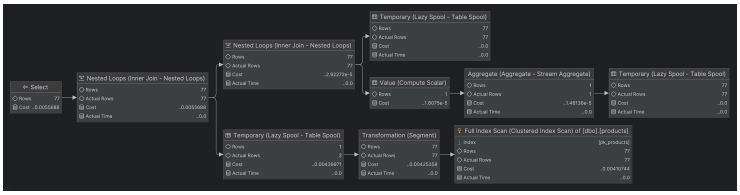

Figure 3: Plan zapytania dla MSSQL - zapytanie z funckją okna

Różnice w planach wykonania MSSQL

Analiza wyników dla zapytań z **podzapytaniem**, **joinem** oraz **funkcją okna** w MSSQL ujawnia kilka kluczowych różnic w planach wykonania.

1. **Podzapytanie (Subquery)**: W przypadku podzapytania widoczna jest dodatkowa operacja **Compute Scalar**, która oblicza wartości na podstawie warunków. Dodatkowo, zapytanie wykorzystuje **Stream Aggregate** do agregowania wyników przed dołączeniem, co wprowadza dodatkowy koszt związany z agregowaniem danych w osobnym kroku. Ponadto, zapytanie korzysta z **Clustered Index Scan**, co wskazuje na pełne skanowanie tabeli.

2. **Join**: Plan wykonania zapytania z **joinem** jest zbliżony do zapytania z podzapytaniem, jednak różni się brakiem osobnej agregacji. **Compute Scalar** jest nadal obecny, ale agregacja (COUNT i SUM) wykonywana jest bezpośrednio w trakcie dołączenia. Również w tym przypadku wykorzystano **Clustered Index Scan**, co sprawia, że zapytanie może być nieco bardziej wydajne niż zapytanie z podzapytaniem, ponieważ nie ma osobnego etapu agregacji.

3. **Funkcja okna (Window Function)**: Plan zapytania z funkcją okna jest najbardziej złożony. Zawiera dodatkowe operacje takie jak **Table Spool** i **Segment**, które pomagają w przechowywaniu wyników pośrednich. Zapytanie

wykonuje **Clustered Index Scan** na tabeli, a agregacja z funkcją okna jest realizowana w obrębie samego zapytania, co może zwiększać złożoność. Jednak zastosowanie tych mechanizmów może prowadzić do bardziej efektywnego przetwarzania danych w przypadku dużych zbiorów.

Podsumowanie

W przypadku MSSQL zapytania z **funkcją okna** okazują się być najbardziej złożone w wykonaniu, z dodatkowymi mechanizmami buforowania wyników i segmentowania danych. Z kolei zapytania z **podzapytaniem** i **joinem** są mniej skomplikowane, ale podzapytanie może wiązać się z wyższym kosztem z powodu osobnej agregacji. Ostateczna wydajność zależy od konkretnego przypadku i charakterystyki danych, jednak w ogólności zapytania z **funkcją okna** mogą zapewniać lepszą optymalizację przy dużych zbiorach danych.

PostgreSQL

Teraz przeanalizujemy plan zapytania w systemie PostgreSQL.

Figure 4: Plan zapytania dla PostgreSQL - zapytanie z subquery

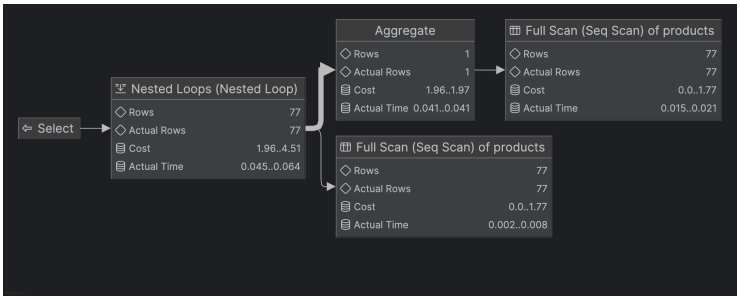

Figure 5: Plan zapytania dla PostgreSQL - zapytanie z JOIN

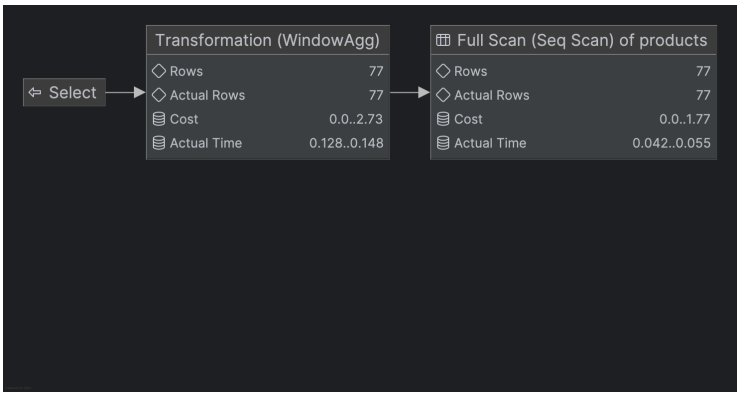

Figure 6: Plan zapytania dla PostgreSQL - zapytanie z funkcją okna

Różnice w planach wykonania PSQL

Analiza wyników `EXPLAIN ANALYZE` dla zapytań z podzapytaniem, `JOIN` oraz funkcją okna w PostgreSQL ujawnia różnice w metodzie przetwarzania danych oraz czasie wykonania:

- **Subquery**: Zapytanie z podzapytaniem wykonuje pełne skanowanie tabeli `products` oraz agregację. Czas wykonania całkowity wynosi 0.383 ms, a proces agregacji jest względnie szybki (0.139..0.140 ms). Niemniej jednak, pełne skanowanie jest mniej efektywne w przypadku dużych danych.
- **Join**: W przypadku `JOIN` wykorzystano metodę `Nested Loop`, która wykonuje wielokrotne skanowanie tabeli. Czas wykonania wynosi 0.099 ms, ale przy większych zbiorach danych może być mniej skalowalne. Koszt planu jest wyższy niż w przypadku podzapytania.
- **Window**: Funkcja okna (`WindowAgg`) wykonuje się najwydajniej, chociaż koszt analizy jest wyższy niż w przypadku pozostałych zapytań. Czas wykonania wynosi 0.160 ms, co jest wynikiem przetwarzania danych w jednym przebiegu. Funkcja okna okazuje się być najbardziej wydajna w tym przypadku.

Podsumowanie

Podsumowując, w analizie zapytań w PostgreSQL:

- **Subquery** jest najprostsze w implementacji, ale może być mniej wydajne przy większych danych z powodu pełnego skanowania tabeli.
- **Join** wykorzystuje `Nested Loop`, co może być nieefektywne w przypadku dużych zbiorów danych. Mimo to, zapytanie wykonuje się szybciej niż podzapytanie.

- **Window** okazało się najwydajniejsze w przypadku mniejszych danych. Mimo wyższego kosztu planu, czas wykonania jest najniższy, ponieważ zapytanie przetwarza dane w jednym przebiegu, co daje lepsze rezultaty w porównaniu do `JOIN` i `Subquery`.

SQLite

W przypadku SQLite, wyniki zapytania w narzędziu EXPLAIN PLAN wyglądają następująco.

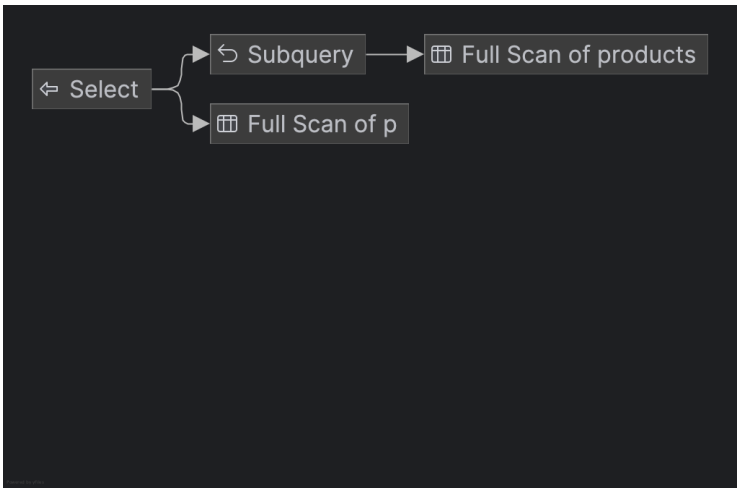

Figure 7: Plan zapytania dla SQLite - zapytanie z subquery

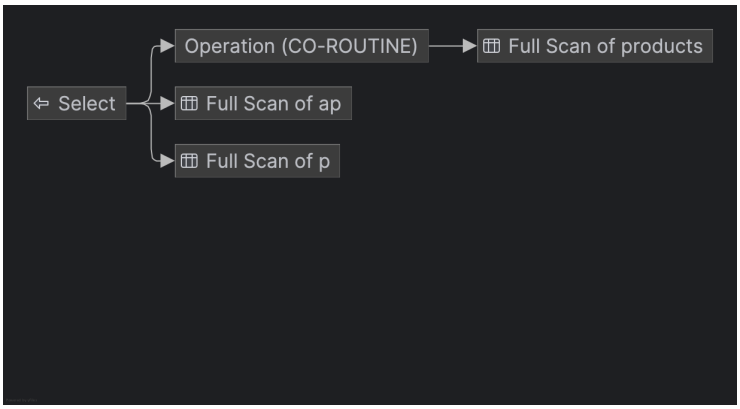

Figure 8: Plan zapytania dla SQLite - zapytanie z JOIN

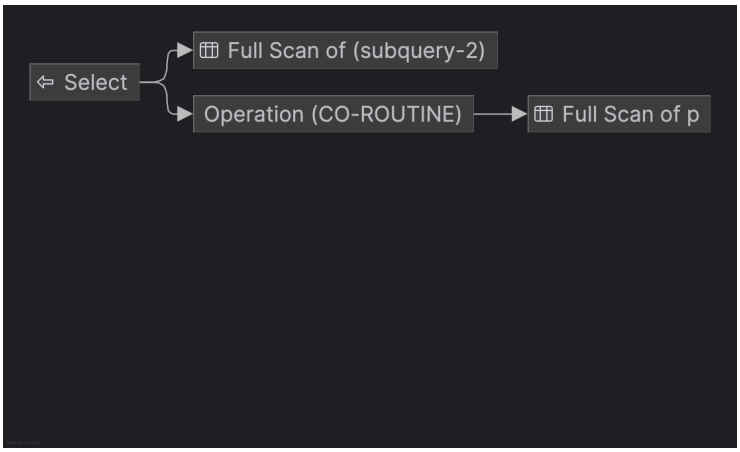

Figure 9: Plan zapytania dla SQLite - zapytanie z funkcją okna

Różnice w planach wykonania SQLite

- **Subquery:** Plan wykonania subquery składa się z dwóch skanów tabeli `products`. Pierwszy jest dla głównego zapytania, a drugi dla podzapytania. Jest to stosunkowo prosta operacja, choć koszt związany z oddzielnym przetwarzaniem podzapytania jest wyższy.
- **Join:** Plan dla zapytania z `JOIN` obejmuje cztery operacje skanowania: jedną dla tabeli `products`, jedną dla agregacji (alias `ap`), oraz dwie dla tabeli `products` (z powodu działania `JOIN`). Plan jest bardziej złożony w porównaniu do subquery.
- **Window Function:** Zapytanie z funkcją okna jest również złożone, ponieważ SQLite używa `CO-ROUTINE` do obliczenia agregacji w ramach funkcji okna. Skanowanie tabeli `products` i wyników funkcji okna wprowadza większą złożoność w porównaniu do prostych zapytań z `JOIN` i subquery.

Podsumowanie

Porównując zapytania z `subquery`, `JOIN` oraz `Window Function` w SQLite, można zauważyć, że:

- Zapytania z `subquery` są najprostsze, ponieważ wymagają tylko dwóch skanów tej samej tabeli.
- Zapytania z `JOIN` i `Window Function` są bardziej złożone, ponieważ wymagają więcej operacji skanowania oraz wykorzystują dodatkowe techniki, takie jak `CO-ROUTINE`.
- Zapytanie z funkcją okna jest najbardziej złożone, ponieważ SQLite wykonuje równoległe przetwarzanie wyników za pomocą `CO-ROUTINE` i agregacji okna.

Zadanie 4

Poniżej znajdują się trzy różne podejścia do uzyskania rezultatu zadania: przy użyciu podzapytania, JOIN-a oraz funkcji okna.

1. **Zapytanie z wykorzystaniem podzapytania**

Listing 4: Subquery

```
        SELECT p.productid,
        p.productname,
        p.unitprice,
        avg_price
FROM products p
         JOIN (SELECT categoryid,
                      AVG(unitprice) AS avg_price
               FROM products
               GROUP BY categoryid) avg_table
              ON p.categoryid = avg_table.
                 categoryid
WHERE p.unitprice > avg_table.avg_price
ORDER BY 1;
```

2. **Zapytanie z wykorzystaniem JOIN-a**

Listing 5: JOIN

```
        SELECT p.productid,
        p.productname,
        p.unitprice,
        avg_price
FROM products p
         JOIN (SELECT categoryid,
                      AVG(unitprice) AS avg_price
               FROM products
               GROUP BY categoryid) avg_table
              ON p.categoryid = avg_table.
                 categoryid
WHERE p.unitprice > avg_table.avg_price
ORDER BY 1;
```

3. **Zapytanie z wykorzystaniem funkcji okna**

Listing 6: Funkcja okna

```
        WITH avg_prices AS (SELECT p.productid,
        p.productname,
        p.unitprice,
```

```
        AVG(p.unitprice) OVER (PARTITION BY p.
            categoryid) AS avg_price
 FROM products p)
SELECT productid,
productname,
unitprice,
avg_price
FROM avg_prices
WHERE unitprice > avg_price
ORDER BY 1;
```

Następnie porównałem wyniki operacji Explain Analyze dla każdego z tych zapytań dla każdej z bas danych. W sprawozdaniu przedstawiam wyniki tylko dla PostgreSQL, w pozostalych przypadkach obserwacje dotyczace efektywnosci zapytan sa bardzo podobne.

PostgreSQL

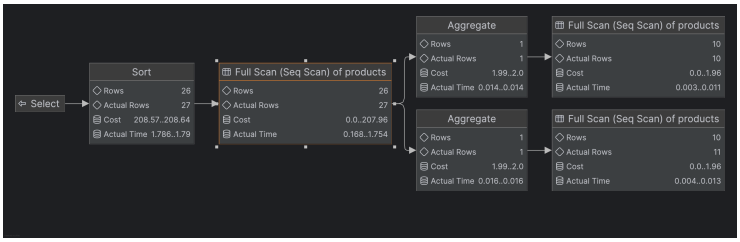

Figure 10: Plan zapytania dla PSQL - zapytanie z subquery

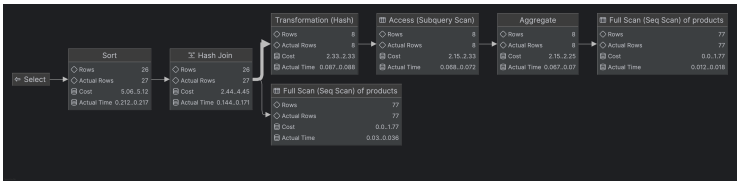

Figure 11: Plan zapytania dla PSQL - zapytanie z JOIN

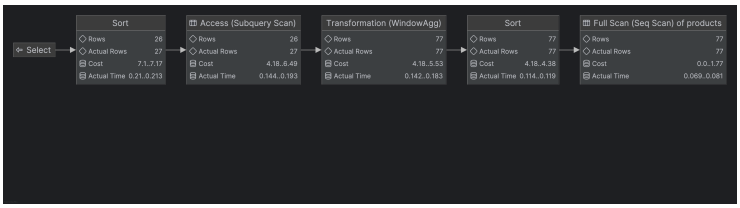

Figure 12: Plan zapytania dla PSQL - zapytanie z funckją okna

1 Analiza wydajności zapytań

1.1 Zapytanie z podzapytaniem (subquery)

Dla zapytania z podzapytaniem wykonanie planu obejmuje:

- Dla każdego wiersza w tabeli ‘products‘ wykonywane jest osobne podzapytanie, co prowadzi do wysokich kosztów wykonania, zwłaszcza przy dużych tabelach.
- Wydajność jest obniżona, ponieważ obliczenie średniej ceny dla każdej kategorii odbywa się wielokrotnie dla każdego produktu.

1.2 Zapytanie z JOIN

W rozwiązaniu z JOIN:

- Średnia cena dla kategorii obliczana jest tylko raz w podzapytaniu, a następnie dołączana do tabeli ‘products‘.
- Koszty wykonania są znacznie niższe, ponieważ średnia cena dla każdej kategorii obliczana jest raz i nie ma potrzeby wielokrotnego wykonywania podzapytań.

1.3 Zapytanie z funkcjami okna (window functions)

Zapytanie wykorzystujące funkcje okna (window functions):

- Funkcje okna pozwalają na obliczenie średniej ceny dla każdej kategorii w jednym przejściu przez dane, bez konieczności używania podzapytań.
- Jest to najwydajniejsze rozwiązanie, ponieważ agregacja odbywa się w jednym kroku, a następnie używane są tylko wyniki dla każdego wiersza.

1.4 Podsumowanie

Najbardziej wydajne rozwiązanie to zapytanie z funkcjami okna, które oblicza średnią cenę w ramach jednej operacji. Zapytanie z JOIN również oferuje znaczną poprawę wydajności w porównaniu do zapytania z podzapytaniem,

ponieważ średnia cena obliczana jest tylko raz. Zapytanie z podzapytaniem jest najwolniejsze, ponieważ wymaga obliczenia średniej ceny dla każdego produktu, co skutkuje dużym obciążeniem bazy danych przy większych ilościach obliczeń.

Zapytanie	**Koszt sortowania**	**Czas wykonania (ms)**
Podzapytanie	208.57	1.540
JOIN	5.06	0.289
Funkcje okna	7.10	0.334

Table 1: Porównanie zapytań pod względem kosztów i czasu wykonania