1. Elementy OLAP w bazie PostgreSQL

W ramach bazy danych PostgreSQL - relacyjnej bazy danych typu OLTP (ang. Online Transaction Processing) pojawiły się funkcjonalności dotąd dostępne w systemach OLAP (ang. Online Analitycal Processing). Dodatkowe możliwości dostępne bazach danych umożliwiają bardziej wydajną analizę danych bez konieczności przenoszenia danych do systemów hurtowni danych z wbudowanymi technologiami do analizy OLAP czy zewnętrznych aplikacji np. arkuszy kalkulacyjnych. Funkcje analityczne są wykorzystywane wyłącznie w klauzulach SELECT oraz ORDER BY i nie mogą być używane w klauzulach WHERE, GROUP BY, HAVING. Działają wyłącznie na wierszach będących wynikiem zapytania i nie odrzuconych przez WHERE lub HAVING.W ramach zajęć zostaną omówione dodatkowe operatory grupujące, partycje obliczeniowe oraz funkcje rankingowe dostępne w bazie danych PostgreSQL 9.6.

1. Operatory grupujące w bazie danych PostgreSQL.

Podstawową konstrukcją wykorzystującą funkcje agregujące SUM(), AVG(), COUNT(), MIN() czy MAX() w języku SQL jest konstrukcja **"SELECT .... funkcja agregująca FROM relacje GROUP BY atrybuty "** po których grupujemy. Do realizacji kilku agregacji na tym samym zestawie danych w ramach jednego zapytania SQL możemy wykorzystać operator sumy zbiorów **UNION**. Do realizacji powyższego zadania zostały udostępnione w ramach języka SQL dodatkowe konstrukcje umożliwiające realizację powyższych zadań wydajniej w ramach systemu bazodanowego: **GROUPING SETS**, **ROLLUP** i **CUBE.**
Przykładowe polecenie SQL realizujące opisany powyżej przypadek. Należy wyznaczyć liczbę rekordów i całkowitą wartość sprzedaży dla regionu i prowincji, regionu, prowincji i całkowitą liczbę wszystkich sprzedaży i całkowity koszt sprzedaży.

1.1. Operator GROUPING SETS.

Konstrukcja **GROUPING SETS** rozszerza funkcjonalność klauzuli GROUP BY o możliwość wyboru zbiorów grup, które chcemy uzyskać w wyniku zapytania. Poniżej konstrukcja zapytania SQL wykorzystująca operację **GROUPING SETS**.

```
SELECT  {lista atrybutów}
FROM  {lista relacji}
...
GROUP  BY  GROUPING  SETS(({lista wyrażeń grupujących 1}),
                ({lista wyrażeń grupujących 2}),
                ({lista wyrażeń grupujących 3})
 ... )
```

Przykładowe polecenie SQL realizujące opisany powyżej przypadek. Należy wyznaczyć liczbę rekordów i całkowitą wartość sprzedaży dla regionu i prowincji, regionu, prowincji i całkowitą liczbę wszystkich sprzedaży i całkowity koszt sprzedaży.

```
-- sumowanie po regionach i prowincjach
select region, province, count(*),sum(sales) from sample group by region,
province
union
-- sumowanie po regionach
select region, null, count(*),sum(sales) from sample group by region
union
-- sumowanie po prowincjach
select null, province, count(*),sum(sales) from sample group by province
union
-- sumowanie po wszystkich rekordach
select null, null, count(*),sum(sales) from sample
order by region, province;
```

Zapytanie SQL realizujące przedstawione powyżej zadanie z wykorzystaniem operacji **GROUPING SETS** przedstawia poniższy przykład.

```
select region, province, count(*),sum(sales) from sample
group by grouping sets ((region, province), (region), (province), ())
order by region, province ;
```

1.2.Operator GROUP BY ROLLUP.

Kolejną operacją umożliwiająca bardziej wydajne grupowanie danych jest półkostka danych realizowana przez **ROLLUP**. Konstrukcja zapytania SQL wykorzystująca operację **ROLLUP** przedstawiona została poniżej.

```
SELECT {lista atrybutów} FROM {lista relacji}
...
GROUP BY ROLLUP({lista wyrażeń grupujących})
...;
```

W trakcie realizacji tego zapytania zostaną wyznaczone wyrażenia grupujące dla grup tworzonych na podstawie pierwszych n, n-1, n-2, ... , 0 wyrażeń

grupujących wymienionych w klauzuli GROUP BY ROLLUP. Dla każdej grupy zwracany jest jeden rekord podsumowania.

.. GROUP BY ROLLUP (aa, bb, cc) ..
-- zwracane wartości
aa bb cc (wartość zagregowana)
aa bb (wartość zagregowana)
aa (wartość zagregowana)
(wartość zagregowana)

Zapytanie SQL realizujące przedstawione powyżej zadanie z wykorzystaniem operacji GROUP BY ROLLUP przedstawia poniższy przykład.

```
select region, province, count(*),sum(sales) from sample
group by rollup (region, province) order by region, province ;
```

1.3. Operator GROUP BY CUBE. Ostatni operator **CUBE** tworzy pełną kostkę danych. Poniżej konstrukcja zapytania SQL.

SELECT {lista atrybutów} FROM {lista relacji}
...
GROUP BY CUBE({lista wyrażeń grupujących})
...;

W ramach tej konstrukcji tworzone są grupy na podstawie wszystkich kombinacji wartości wyrażeń wymienionych w klauzuli **GROUP BY CUBE**, tj. 2^N kombinacji dla N wyrażeń. Dla każdej kombinacji zwracany jest jeden rekord podsumowania.

Zapytanie SQL realizujące przedstawione powyżej zadanie z wykorzystaniem operacji **GROUP BY CUBE** przedstawia poniższy przykład.

```
select region, province, count(*),sum(sales) from sample
group by cube (region, province) order by region, province ;
```

1.4. Funkcjonalność PIVOT.

Ciekawą konstrukcją w dzisiejszych systemach relacyjnych jest realizacja funkcji PIVOT (MS SQL Server czy ORACLE). W ramach bazy danych PostgreSQL funkcjonalność ta została zrealizowana z wykorzystaniem funkcji crosstab(). Na początek przygotujemy zestaw danych do realizacji funkcji PIVOT() w bazie danych. Opracowany zestaw danych zawiera informację o wartościach zagregowanych sprzedaży dla każdego miesiąca plus rok, każdego miesiąca (klauzula GROUP BY ROLLUP) oraz każdego miesiąca plus rok, każdego miesiąca, każdego roku (klauzula GROUP BY CUBE).

Na podstawie danych utworzonych dla klauzuli **GROUP BY CUBE** utworzymy tabelę tymczasową zawierającą otrzymany wynik.

```
create temp table temp1 as
select extract(month from ship_date) as m, extract(year from ship_date) as
y, sum(sales) as s
from sample group by cube ( extract(month from ship_date), extract(year from
ship_date) ) order by 2, 1 ;
```

Poniżej zapytanie realizujące utworzenie tabeli przestawnej w bazie danych PostgreSQL z wykorzystaniem funkcji crosstab().

```
select * from crosstab ( 'select y::text, m::text, s from temp1' )
as fr ( Rok text, "Styczen" numeric, "Luty" numeric,
"Marzec" numeric, "Kwiecien" numeric, "Maj" numeric, "Czerwiec" numeric,
"Lipiec" numeric, "Sierpien" numeric, "Wrzesien" numeric, "Pazdziernik"
numeric,"Listopad" numeric, "Grudzien" numeric, "All" numeric) ;
```

2. Partycje obliczeniowe i funkcje rankingowe

Tradycyjne wykorzystanie funkcji agregujących wymaga wykorzystania klauzuli GROUP BY grupującej rekordy. Dla każdej grupy rekordów wyznaczana jest jedna wartość zagregowana. W ramach funkcjonalności partycji obliczeniowej możliwe jest wyznaczenie wartości zagregowanej oddzielnie dla każdego rekordu grupy (a nie raz dla wszystkich rekordów). W celu skorzystania z tego rozwiązania należy wykorzystać wyrażenie PARTITION osadzone w wyrażeniu OVER(). Realizacja przedstawionej funkcjonalności realizowana jest w następującej strukturze:

```
funkcja_agregująca() OVER ( PARTITION BY atrybut )
```

gdzie:
funkcja_agregująca() - tradycyjna funkcja grupująca,
atrybut - atrybut (wyrażenie) grupujący rekordy w celu wyliczenia funkcji.

Funkcje rankingowe umożliwiają wyznaczenie położenia rekordu w odniesieniu do pozostałych rekordów grupy względem wybranej funkcji porządkującej. Implementacja SQL w bazie danych PostgreSQL realizuje sześć funkcji rankingowych: RANK(), DENSE_RANK(), CUME_RANK(), PERCENT_RANK(), NTILE() i ROW_NUMBER(). Składnia wyrażenia rankingowego przedstawia się następująco:

```
funkcja() over( [partition by wyrażenie1] order by wyrażenie2 [desc] )
```

gdzie:
funkcja() - funkcja rankingowa.
wyrażenie1 - wyrażenie grupujące rekordy w rankingu. Brak wyrażenie oznacza, że w rankingu biorą udział wszystkie rekordy. Parametr opcjonalny.
wyrażenie2 - wyrażenie porządkujące rekordy wewnątrz grupy, parametr wymagany.
desc - parametr zmieniający porządek rankingu, od wartości największej do najmniejszej.

Funkcjonalność realizowana przez poszczególne funkcje rankingowe:

RANK() i DENSE_RANK() - ranking zwykły,
CUME_DIST - ranking względny,
PERCENT_RANK - ranking procentowy,
ROW_NUMBER - numer rekordu,

NTILE - podział partycji na grupy.

Poniżej przykład wykorzystania funkcji ROW_NUMBER() do numeracji rekordów.

```
with cte as ( select region, province, sum(sales) as sum from sample
group by region, province )
select row_number() over(order by region) as nr,
region, province, sum, sum(sum) over ( partition by region ),
round(100*sum/(sum(sum) over (partition by region))) as udzial
from cte;
```

Przykład przedstawia numerację wszystkich rekordów i rekordów w ramach każdego regionu oddzielnie.

```
with cte as ( select region, province, sum(sales) as sum from sample
group by region, province )
select row_number() over(order by region) as nr,
row_number() over (partition by region order by region ) as nl,
region, province, sum, sum(sum) over ( partition by region ),
round(100*sum/(sum(sum) over (partition by region))) as udzial
from cte;
```

Funkcja RANK().

```
with cte as (select product_category, product_sub_category, count(*)
from sample
group by product_category, product_sub_category order by 1,2)
select product_category, product_sub_category,
rank() over( order by product_category ) from cte;
```

Funkcja DENSE_RANK().

```
with cte as (select product_category, product_sub_category, count(*)
from sample
group by product_category, product_sub_category order by 1,2)
select product_category, product_sub_category,
dense_rank() over( order by product_category ) from cte;
```

Prezentacja funkcji RANK(), DENSE_RANK() i ROW_NUMBER().

```
with cte as (select product_category, product_sub_category, sum(sales) suma
from sample
group by product_category, product_sub_category order by 1,2)
select product_category, product_sub_category,
dense_rank() over( order by product_category ),
rank() over(order by product_category),
row_number() over( partition by product_category),
row_number() over(),
suma, sum(suma) over( partition by product_category)
from cte;
```

Funkcja rankingowa CUME_DIST() oblicza skumulowany rozkład wartości w partycji, wyznacza jaki procent rekordów w partycji poprzedza w rankingu bieżący rekord (tzw. percentyl). Wyznacza wartość w przedziale (0,1>. Do wyliczonej wartości dodawany jest bieżący rekord.

```
select product_sub_category, sum(sales) as suma,
cume_dist() over ( order by sum(sales) desc ) as cume_dist
from sample
group by product_sub_category
order by suma desc, product_sub_category;
```

Ostatnią funkcją rankingową jest funkcja PERCENT_RANK(), która wylicza procentowy ranking z bieżącego rekordu w stosunku do zbioru wartości w partycji. Wyznaczona wartość zawiera się w przedziale (0,1>, działa podobnie jak CUME_DIST(), ale pomija bieżący rekord.

```
select product_sub_category, sum(sales) as suma,
percent_rank() over ( order by sum(sales) desc ) as perc_rank
from sample
group by product_sub_category
order by suma desc, product_sub_category;
```

Funkcje analityczne w bazie PostgreSQL

1. *PostgreSQL - Table Expressions*
2. *PostgreSQL - Window Functions*
3. *PostgreSQL - Window Functions*
4. *Waiting for 9.5 - Support GROUPING SETS, CUBE and ROLLUP*
5. *T-SQL Programming Part 12 - Using the ROLLUP, CUBE, and GROUPING SETS Operators*
6. *Pivot Tables in PostgreSQL Using the Crosstab Function*
7. *PostgreSQL - tablefunc*
8. *PostgreSQL - Date/Time Functions and Operators*

Funkcje rankingowe

9. *The Difference Between ROW_NUMBER(), RANK(), and DENSE_RANK()*
10. *SQL - FUNKCJA RANKINGOWA RANK()*
11. *Wyszukiwanie ciągłych zakresów, dziur, problemy wysp w SQL*