OLAP Functions in SQL iSeries per un'Analisi Dati Avanzata: Un Focus sulle Windows Functions

9 Febbraio 2024

Nel regno dell'analisi e della gestione dei dati sui sistemi IBM iSeries, SQL offre un potente insieme di strumenti progettati per migliorare l'efficienza e la profondità dell'interrogazione dei dati.

Tra questi strumenti, le funzioni OLAP (Online Analytical Processing) spiccano per la loro capacità di eseguire query analitiche complesse direttamente nell'ambiente SQL.

Questo post esplora la categoria specializzata di funzioni OLAP note come Funzioni di Finestra, che forniscono un meccanismo sofisticato per accedere e analizzare i punti dati in un dataset senza la necessità di join su sé stessi ingombranti.

Le Funzioni di Finestra consentono agli analisti e agli sviluppatori di eseguire confronti e analisi di dati complessi con maggiore semplicità ed eleganza.

Definendo una "finestra" o un sottoinsieme di righe in un set di risultati, queste funzioni permettono l'esecuzione di calcoli su queste righe, offrendo intuizioni che sono sia ampie che profondamente contestualizzate nel panorama dei dati.

Tra le Funzioni di Finestra più importanti troviamo:

- **LEAD e LAG**: Queste funzioni sono fondamentali nella navigazione dei dati temporali o sequenziali, permettendo agli utenti di accedere ai dati da una riga successiva (LEAD) o da una riga precedente (LAG) con una facilità senza precedenti. Questa capacità è essenziale per l'analisi delle tendenze, le previsioni e l'esame delle progressioni dei dati nel tempo senza la complessità dei join su sé stessi.
- **FIRST_VALUE e LAST_VALUE**: Consentendo un accesso diretto al primo o all'ultimo valore all'interno di una finestra specificata, queste funzioni facilitano analisi che richiedono la comprensione del punto di inizio o di conclusione dell'intervallo di valori di un dataset. Tali intuizioni sono cruciali per identificare i punti di partenza, i punti finali e i cambiamenti nelle tendenze dei dati.
- **NTH_VALUE()**: Questa funzione estende la versatilità delle funzioni di finestra estraendo il valore di una colonna specificata alla n-esima posizione all'interno della finestra, offrendo un controllo preciso sulla selezione dei punti dati all'interno di un dataset. Apre possibilità per analisi mirate e studi comparativi attraverso segmenti di dati vari.

L'adozione delle Funzioni di Finestra in SQL iSeries non solo semplifica compiti di analisi dati complessi ma migliora significativamente la capacità di trarre conclusioni significative da dataset grandi e diversificati.

Sfruttando queste funzioni, i professionisti possono sbloccare un livello superiore di precisione analitica ed efficienza, spingendo i limiti di ciò che può essere realizzato nell'analisi dei dati sulla piattaforma IBM iSeries.

Questo post mira a fornire una comprensione completa di queste potenti funzioni OLAP, illustrando il loro potenziale attraverso esempi e best practices, per potenziare gli utenti nell'utilizzo delle loro complete capacità analitiche all'interno dell'ambiente SQL iSeries.

Questa introduzione prepara il terreno per un'esplorazione più profonda delle funzioni di finestra, sottolineando la loro importanza e utilità nell'analisi dei dati all'interno del framework SQL iSeries.

Di seguito gli esempi che evidenziano l'uso delle funzioni OLAP in iSeries SQL, concentrandosi sulle funzioni finestra LEAD, LAG, FIRST_VALUE, LAST_VALUE e NTH_VALUE(). Creeremo un set di dati di esempio per testare queste funzioni utilizzando una Common Table Expression (CTE) con un'istruzione VALUES.

Questo approccio simula uno scenario pratico, fornendo un contesto chiaro per l'applicazione di queste funzioni e ci consente di applicare direttamente le nostre funzioni di finestra su un set di dati di esempio, senza la necessità di avere accesso a una base dati reale.

Esempio 1: LEAD e LAG OLAP Functions

In questo esempio, esploreremo l'utilizzo avanzato delle funzioni OLAP in SQL, specificatamente su un sistema iSeries, per analizzare i dati di vendita e scoprire pattern interessanti nei comportamenti d'acquisto dei clienti.

Le funzioni OLAP, come **LEAD**, **LAG**, ci permettono di esaminare le transazioni in maniera sequenziale, evidenziando la frequenza delle vendite e le variazioni degli importi di vendita tra transazioni consecutive.

Utilizzeremo una Common Table Expression (CTE) per simulare un dataset di transazioni di vendita, contenente informazioni come ID transazione, ID cliente, data transazione e importo vendita.

Attraverso la query seguente, dimostreremo come calcolare la differenza in giorni tra transazioni consecutive per ogni cliente, la variazione percentuale dell'importo di vendita da una transazione all'altra, e come accedere agli importi di vendita precedenti e successivi.

L'obiettivo è fornire intuizioni dettagliate sul comportamento d'acquisto dei clienti e sulla frequenza delle transazioni, utilizzando potenti strumenti di analisi dati integrati in SQL iSeries.

Vediamo la query in azione!

```
WITH
  SalesData (TrnID, CustID, TrnDate, SaleAmount) AS (
    VALUES
        (1,  'C1', DATE('2022-01-03'), 100)
      , (2,  'C1', DATE('2022-02-06'), 150)
      , (3,  'C1', DATE('2022-03-06'), 120)
      , (4,  'C1', DATE('2022-05-29'), 170)
      , (5,  'C1', DATE('2022-07-11'), 200)
      , (6,  'C1', DATE('2022-10-12'), 100)
      , (7,  'C2', DATE('2022-01-14'), 150)
      , (8,  'C2', DATE('2022-01-16'), 120)
      , (9,  'C2', DATE('2022-01-16'), 170)
      , (10, 'C2', DATE('2022-01-17'), 200)
      , (11, 'C2', DATE('2022-01-22'), 170)
      , (12, 'C2', DATE('2022-01-26'), 200)
)  -- select * from SalesData;
SELECT TrnID "Trn"
      , CustID "Cust"
      , TrnDate "Date"
      , SaleAmount "Amount"
      , DAYS(TrnDate) -
           DAYS(LAG(TrnDate) OVER (PARTITION BY CustID
                ORDER BY TrnDate)) "DaysPrev"
      , DAYS(LEAD(TrnDate) OVER (PARTITION BY CustID
             ORDER BY TrnDate)) - DAYS(TrnDate)
             "DaysNext"
      , LAG(SaleAmount) OVER (PARTITION BY CustID
                ORDER BY TrnDate) "PrevAmount"
      , 100 * DECIMAL(DECIMAL(SaleAmount, 10, 4)
        / DECIMAL(LAG(SaleAmount)
                     OVER (PARTITION BY CustID
                     ORDER BY TrnDate), 10, 3),
                 10, 4) "PercAmount"
      , LEAD(SaleAmount) OVER (PARTITION BY CustID
                ORDER BY TrnDate) "NextAmount"
  FROM  SalesData;
```

Risultato:

Trn	Cust	Date	Amount	DaysPrev	DaysNext	PrevAmount	PercAmount	NextAmount
1	C1	2022-01-03	100	[NULL]	34	[NULL]	[NULL]	150
2	C1	2022-02-06	150	34	28	100	150	120
3	C1	2022-03-06	120	28	84	150	80	170
4	C1	2022-05-29	170	84	43	120	141,66	200
5	C1	2022-07-11	200	43	93	170	117,64	100
6	C1	2022-10-12	100	93	[NULL]	200	50	[NULL]
7	C2	2022-01-14	150	[NULL]	2	[NULL]	[NULL]	120
8	C2	2022-01-16	120	2	0	150	80	170
9	C2	2022-01-16	170	0	1	120	141,66	200
10	C2	2022-01-17	200	1	5	170	117,64	170
11	C2	2022-01-22	170	5	4	200	85	200
12	C2	2022-01-26	200	4	[NULL]	170	117,64	[NULL]

Spiegazione dettagliata della Query

La nostra query SQL utilizza le funzioni OLAP **LEAD** e **LAG**, e calcola la differenza in giorni tra le date di transazione consecutive (**TrnDate**) per ogni cliente (**CustID**), oltre a calcolare la variazione percentuale dell'**SaleAmount** tra le transazioni consecutive.

Vediamo nel dettaglio cosa fa la nostra query:

1. **WITH SalesData**: Definisce una Common Table Expression (CTE) chiamata **SalesData** che simula una tabella di dati di vendita con colonne per l'ID transazione (**TrnID**), ID cliente (**CustID**), data transazione (**TrnDate**) e importo vendita (**SaleAmount**). Questo set di dati temporaneo viene utilizzato per dimostrare le funzioni di finestra.
2. **SELECT Clause**: Recupera varie informazioni per ogni transazione:
   - **TrnID** come "Trn",
   - **CustID** come "Cust",
   - **TrnDate** come "Date",
   - **SaleAmount** come "Amount".
3. **DAYS(TrnDate) – DAYS(LAG(TrnDate)...) "DaysPrev"**: Calcola la differenza in giorni tra la data di transazione corrente e la data di transazione precedente per lo stesso cliente. Questo mostra quanti giorni sono passati dall'ultima transazione per ogni cliente.
4. **DAYS(LEAD(TrnDate)...) – DAYS(TrnDate) "DaysNext"**: Calcola la differenza in giorni tra la data di transazione successiva e la data di transazione corrente per lo stesso cliente. Indica quanti giorni passeranno fino alla prossima transazione per ogni cliente.
5. **LAG(SaleAmount) OVER... "PrevAmount"**: Recupera l'importo di vendita dalla transazione precedente per lo stesso cliente. Questo permette di confrontare direttamente l'importo di vendita corrente con quello precedente.
6. **100 * DECIMAL(...) "PercAmount"**: Calcola la variazione percentuale dell'importo di vendita tra la transazione corrente e la precedente per lo stesso cliente. Si tratta di una misura di quanto l'importo di vendita sia aumentato o diminuito in termini percentuali da una transazione all'altra.

7. **LEAD(SaleAmount) OVER... "NextAmount"**: Recupera l'importo di vendita per la transazione successiva per lo stesso cliente. Simile a "PrevAmount", ma per la transazione successiva.

Riepilogo

Questa query dimostra efficacemente il potere delle funzioni di finestra in SQL per eseguire analisi complesse, come il calcolo delle differenze e delle variazioni percentuali tra le righe in un dataset partizionato senza la necessità di unire la tabella con se stessa.

È un approccio molto utile per analizzare dati di serie temporali, modelli di transazione dei clienti o qualsiasi scenario in cui è necessario confrontare i record in sequenza.

Esempio 2: FIRST_VALUE, LAST_VALUE e NTH_VALUE() OLAP Functions

Per dimostrare l'utilizzo delle funzioni finestra **FIRST_VALUE, LAST_VALUE** e **NTH_VALUE()** in iSeries SQL, creiamo uno scenario in cui queste funzioni possono essere applicate in modo significativo.

Simuleremo un set di dati di vendita utilizzando un'espressione di tabella comune (CTE) con **VALUES** per preparare i dati per il test.

Questa configurazione ci aiuterà a esplorare queste funzioni analizzando le transazioni di vendita, identificando modelli o estraendo approfondimenti specifici senza la necessità di auto-join.

Scenario: analisi dei dati di vendita mensili

Supponiamo di avere un set di dati che rappresenta gli importi delle vendite mensili per diversi prodotti in un anno. Vogliamo trovare il primo, l'ultimo e l'importo delle vendite di un mese specifico (ad esempio, il sesto mese) per ciascun prodotto all'interno di questo set di dati.

Vediamo la query in azione!

```
WITH MonthlySales (ProductID, Month, SalesAmount) AS (
    VALUES
        ('P1', '2022-01', 500), ('P1', '2022-02', 450)
      , ('P1', '2022-03', 600), ('P1', '2022-04', 550)
      , ('P1', '2022-05', 650), ('P1', '2022-06', 700)
```

```
      , ('P1', '2022-07', 550), ('P1', '2022-08', 480)
      , ('P1', '2022-09', 610), ('P1', '2022-10', 550)
      , ('P1', '2022-11', 660), ('P1', '2022-12', 720)
      , ('P2', '2022-01', 380), ('P2', '2022-02', 450)
      , ('P2', '2022-03', 600), ('P2', '2022-04', 550)
      , ('P2', '2022-05', 650), ('P2', '2022-06', 660)
      , ('P2', '2022-07', 550), ('P2', '2022-08', 480)
      , ('P2', '2022-09', 610), ('P2', '2022-10', 550)
      , ('P2', '2022-11', 660), ('P2', '2022-12', 620)
)
SELECT distinct
    ProductID "ProductID"
  , SUM(SalesAmount) OVER (PARTITION BY ProductID) "TotalAmount"
  , FIRST_VALUE(SalesAmount)
          OVER (PARTITION BY ProductID
          ORDER BY Month
               ROWS BETWEEN UNBOUNDED PRECEDING
               AND UNBOUNDED FOLLOWING)
          AS "FirstSaleAmount"
  , LAST_VALUE(SalesAmount)
          OVER (PARTITION BY ProductID
          ORDER BY Month
          ROWS BETWEEN UNBOUNDED PRECEDING
               AND UNBOUNDED FOLLOWING)
          AS "LastSaleAmount"
  , NTH_VALUE(SalesAmount, 6)
          OVER (PARTITION BY ProductID
          ORDER BY Month
          ROWS BETWEEN UNBOUNDED PRECEDING
               AND UNBOUNDED FOLLOWING)
          AS "SixthMonthSales"
FROM MonthlySales
order by 1;
```

Risultato:

ProductID	TotalAmount	FirstSaleAmount	LastSaleAmount	SixthMonthSales
P1	7.020	500	720	700
P2	6.760	380	620	660

Spiegazione dettagliata della Query

Questa query SQL utilizza una Common Table Expression (CTE) e diverse funzioni di finestra per analizzare i dati di vendita dei prodotti nel corso di un anno.

Ecco una ripartizione dettagliata di ciascun componente e funzione utilizzata nella query:

Common Table Expression (CTE): MonthlySales

- **WITH MonthlySales (ProductID, Month, SalesAmount) AS (...)**: Definisce una CTE chiamata **MonthlySales**. Questo insieme di risultati temporaneo include tre colonne: **ProductID** per l'identificativo del prodotto, **Month**

per il mese della vendita, e **SalesAmount** per l'importo venduto in quel mese. La clausola **VALUES** che segue questa definizione popola la CTE con i dati di vendita per i prodotti **P1** e **P2** attraverso vari mesi nel 2022.

SELECT Statement

- **SELECT distinct**: Questo comando seleziona righe uniche basate sulla combinazione di valori nelle colonne selezionate. Assicura che ogni **ProductID** appaia solo una volta nell'output, accompagnato dalle metriche calcolate per l'importo totale delle vendite, l'importo della prima vendita, l'importo dell'ultima vendita e l'importo delle vendite del sesto mese.

Window Functions

La query utilizza funzioni di finestra per calcolare metriche specifiche per ogni prodotto. Le funzioni di finestra eseguono calcoli su un insieme di righe della tabella che sono in qualche modo correlate alla riga corrente.

1. **SUM(SalesAmount) OVER (PARTITION BY ProductID)**: Calcola l'importo totale delle vendite per ogni prodotto (**ProductID**). La clausola **PARTITION BY ProductID** assicura che la somma sia calcolata separatamente per ogni prodotto.
2. **FIRST_VALUE(SalesAmount) OVER (PARTITION BY ProductID ORDER BY Month ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING)**: Determina l'importo della prima vendita all'interno di ogni partizione del prodotto, ordinata per mese. Il frame di finestra specificato da **ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING** assicura che la funzione consideri tutte le righe nella partizione dall'inizio alla fine. Tuttavia, poiché **FIRST_VALUE** guarda intrinsecamente al primo valore nella sequenza ordinata, la specificazione del frame è tecnicamente non necessaria per il suo funzionamento ma è sintatticamente richiesta da alcuni dialetti SQL.
3. **LAST_VALUE(SalesAmount) OVER (...)**: Trova l'importo dell'ultima vendita per ogni prodotto, utilizzando una clausola di partizionamento e ordinamento simile a **FIRST_VALUE**. La differenza chiave è nella sua applicazione: **LAST_VALUE** guarda all'ultimo valore nella sequenza. Il frame **ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING** è cruciale qui per assicurare che la funzione consideri correttamente l'intero intervallo della partizione, specialmente nei database che richiedono una specificazione esplicita del frame per restituire il vero ultimo valore attraverso l'intera sequenza.
4. **NTH_VALUE(SalesAmount, 6) OVER (...)**: Recupera l'importo delle vendite per il sesto mese per ogni prodotto. La funzione è partizionata per **ProductID** e ordinata per **Month**, simile alle funzioni precedenti. Il frame **ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING** è specificato, anche se per **NTH_VALUE**, la preoccupazione principale è l'ordinamento corretto e la posizione specifica (la sesta) all'interno di quell'ordine.

FROM Clause

- **FROM MonthlySales**: Indica che i dati per questa query provengono dalla CTE **MonthlySales** definita all'inizio.

ORDER BY Clause

- **ORDER BY 1**: Ordina i risultati per la prima colonna nella lista SELECT, che è **ProductID**. Questo assicura che l'output sia ordinato alfabeticamente o numericamente basato sui valori di **ProductID**.

Riepilogo

La query è strutturata per fornire una visualizzazione completa dei dati di vendita per ciascun prodotto, calcolando l'importo totale delle vendite e identificando gli importi delle vendite del primo, dell'ultimo e del sesto mese utilizzando le funzioni della finestra.

Queste funzioni consentono calcoli avanzati all'interno di ciascuna partizione **ProductID** mantenendo la leggibilità e l'efficienza nell'istruzione SQL. L'uso delle clausole **DISTINCT** e **ORDER BY** garantisce che l'output sia conciso e ben organizzato.

Conclusioni

Questi esempi illustrano la potenza delle funzioni finestra in SQL per iSeries, consentendo l'analisi complessa dei dati senza la necessità di self-join.

Utilizzando un CTE per simulare un set di dati reale, abbiamo mostrato applicazioni pratiche delle funzioni **LEAD, LAG, FIRST_VALUE, LAST_VALUE e NTH_VALUE()** per accedere a punti dati specifici, analizzare tendenze e ricavare informazioni da dati sequenziali.

Queste funzionalità sono preziose per gli analisti di dati e gli sviluppatori che lavorano con iSeries SQL, offrendo un approccio flessibile ed efficiente all'analisi dei dati.

Tenere presente, come sempre, che l'implementazione e le funzionalità effettive potrebbero variare a seconda della versione del sistema operativo iSeries e del database DB2.

È sempre buona norma fare riferimento alla documentazione ufficiale IBM per avere informazioni più precise e dettagliate.

Riferimenti utili

- Data Science: opportunità e sfide da affrontare
- Funzioni OLAP in Data Science: un approccio moderno alla Business Intelligence
- Sfruttare il Potere della Data Science e SQL con OLAP Functions su IBM iSeries

#OLAP Online Analytical Processing | #Window Functions | #SQL | #Data Science | #Business Intelligence

← PRECEDENTE
Sfruttare il Potere della Data Science e SQL con OLAP Functions su IBM iSeries

SEGUENTE →
OLAP Functions in SQL iSeries per un'Analisi Dati Avanzata: Un Focus sulle Ranking Functions

Articoli simili

Data Science

AI Artificial Intelligence

SkillUP

As400 Architettura

Gestire AS400 con SQL

Gestione approfondita Risorse Sistema

Spiccioli di tecnica

Windows

CERCA TESTO IN POST / TAGS

Cerca ...

SkillUp Via Marco Polo, 139 55049 Viareggio (LU)
Via Lucchese, 84 50019 Sesto Fiorentino (FI)
Tel: 055319212 - info@SkillUp.cloud

LinkedIn

Marco Del Corto
Founder/CEO Genesi srl
Visualizza profilo