Le funzioni WINDOW in DAX

PUBBLICATO GENNAIO 29, 2023 *DI* FRANCESCO BERGAMASCHI

2023-01-31_Le_Funzioni_WINDOW_in_DAXDownload

Nell'aggiornamento di dicembre 2022 di Power BI Desktop, sono state annunciate in *preview* nuove funzioni DAX, dette *WINDOW function*. Si tratta di funzioni che si avvicinano ai calcoli visuali, semplificando la sintassi DAX in molti casi (non tutti) e dando la sensazione (solo la sensazione) di avvicinarsi ad Excel. Ci sono molte ragioni per accogliere con soddisfazione questo rilascio, tra cui il fatto che abbiano applicazioni in molteplici contesti, più di quanto si possa immaginare in prima istanza. In questo articolo daremo una descrizione applicata di ognuna delle tre funzioni WINDOW ad oggi rilasciate – *OFFSET*, *INDEX* e *WINDOW* – e mostreremo un'applicazione particolare.

Nota: la funzione *OFFSET* era stata già rilasciata in modo ufficioso nei mesi precedenti, e ne avevamo discusso in un precedente articolo. In questo articolo, nella parte finale, preciseremo un punto rispetto al precedente: l'uso di una misura come criterio di ordinamento.

Useremo, come siamo soliti fare, modelli dati semplici, in particolare due, che sono osservabili in figura 1 ed in figura 2. Il primo modello è in modalità *Import* (le sorgenti sono dei file Excel), e lo useremo per la descrizione delle funzioni; il secondo è in modalità *DirectQuery* (la sorgente è un Database SQL Server *on-premises)* e servirà per mostrare la citata applicazione particolare.

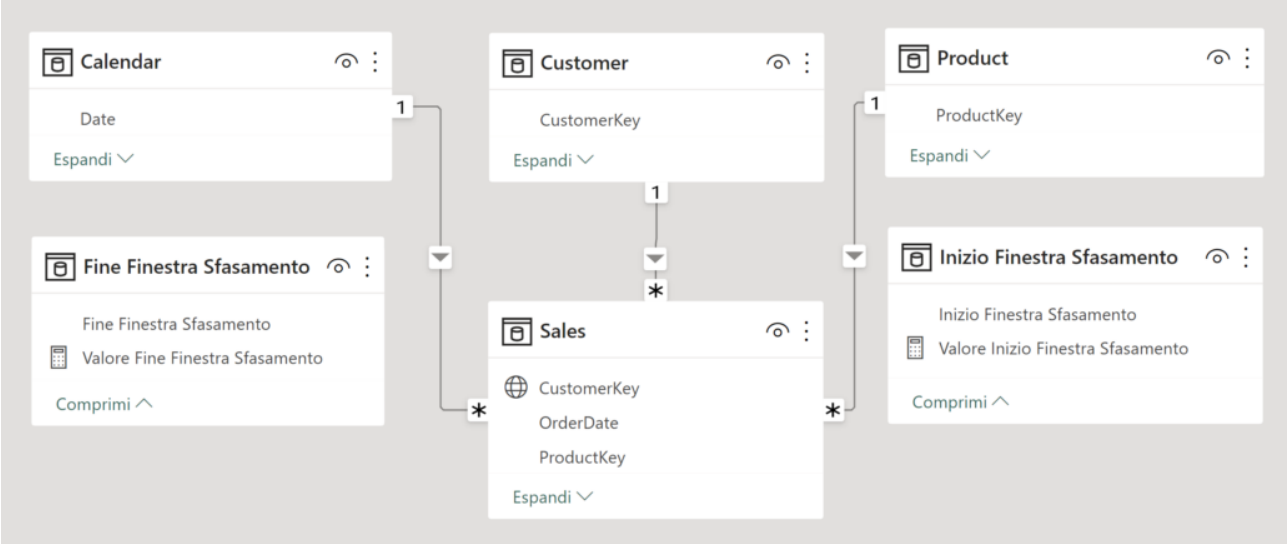

Figura 1 – Modello in *Import*

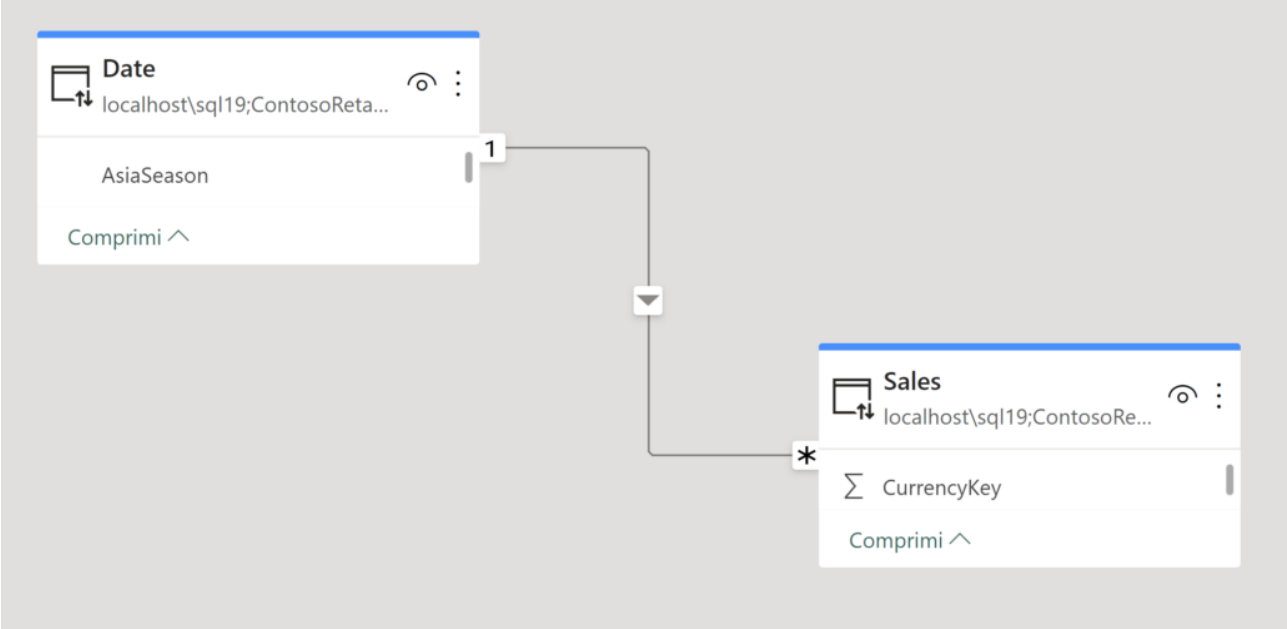

Figura 2 – Modello in *DirectQuery* su Database SQL Server *on-premises*

In figura 1, le due tabelle *Inizio Finestra Sfasamento* e *Fine Finestra Sfasamento* sono state create in Power BI Desktop tramite due *parametri campo* (*field parameter*). In questo articolo potrete trovare una breve descrizione dei parametri campo, insieme ad un'interessante customizzazione dei report che essi rendono possibile.

Per una descrizione molto più approfondita, rispetto a questo articolo, delle funzioni WINDOW, rimandiamo a questo articolo (prima di due parti, la seconda parte è qui) di Jeffrey Wang, uno dei principali architetti del DAX.

Sviluppo

Lo scopo di queste funzioni è permettere di fare calcoli che coinvolgono diverse righe di una tabella. I calcoli che ne derivano sono interamente possibili anche senza le funzioni WINDOW, ma l'espressione DAX si semplifica molto grazie ad esse.

Consideriamo una semplice matrice creata sul modello dati di figura 1, mostrata in figura 3. La matrice ha, in *Righe*, le colonne *Customer[CustomerKey]* e *'Calendar'[Date]* e, in *Valori*, la misura *Sales* il cui codice è a seguire.

Customer Key	Date	Sales
12055	25/08/2003	30 €
12055	07/09/2003	5 €
12055	02/07/2004	24 €
12055	19/07/2004	45 €
12300	24/11/2001	3.578 €
12300	24/07/2003	2.498 €
12300	10/09/2003	2.393 €
12300	20/11/2003	2.320 €
12300	19/04/2004	2.453 €
Totale		**13.347 €**

Figura 3

```
Sales =
SUMX ( Sales, Sales[UnitPrice] * Sales[OrderQuantity] )
```

Inoltre, sulla matrice in figura 3, è stato applicato un filtro basato su una misura che calcola il numero di date di acquisto di ogni cliente, *Sales Dates*, che ne limita i valori tra 4 e 6. In questo modo, i clienti in matrice sono soltanto quelli che hanno tra 4 e 6 diverse date di ordine. Tra i clienti con questa caratteristica, ne sono poi stati filtrati soltanto due per praticità (*Customer[CustomerKey]* = 12055 o *Customer[CustomerKey]* = 12300). In figura 4 sono mostrati i due filtri applicati alla matrice. Il codice di *Sales Dates* è a seguire.

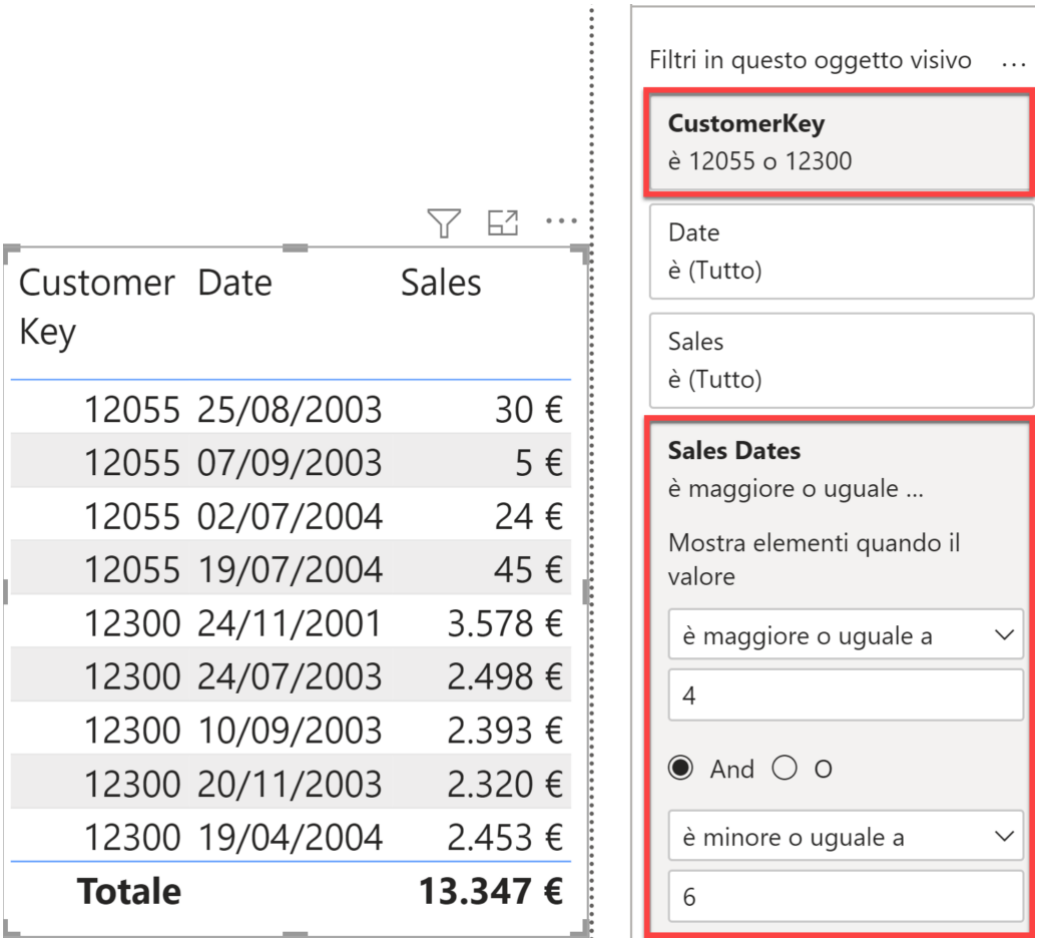

Figura 4

```
Sales Dates =
IF (
    [Sales],
    CALCULATE ( DISTINCTCOUNT ( Sales[OrderDate] ), REMOVEFILTERS ( 'Calendar' ) )
)
```

Si supponga adesso, di volere effettuare 3 diversi tipi di calcolo, ognuno dei quali sarà risolto usando una delle tre funzioni WINDOW.

Primo calcolo: si vuole ottenere il valore di *Sales*, cliente per cliente, alla data di acquisto subito precedente quella nel contesto (come vedremo, in generale si può scegliere una data sfasata di un numero di righe a piacere). In figura 4 il calcolo dovrà fornire, per il cliente 12055, il valore BLANK () alla data del 25/08/2003 (non ci sono acquisti precedenti, per quel cliente), il valore 30 alla data del 07/09/2003, il valore 5 alla data del 02/07/2004 e il valore 24 alla data del 19/07/2004. In modo analogo, verranno calcolati i valori per l'altro cliente, 12300. Il codice è a seguire e il risultato è mostrato in figura 5.

```
Sales OFFSET Customer Date =
IF (
    HASONEVALUE ( Customer[CustomerKey] ),
    CALCULATE (
        [Sales],
        OFFSET (
            – [Valore Inizio Finestra Sfasamento],
            SUMMARIZE ( ALLSELECTED ( Sales ), Customer[CustomerKey], 'Calendar'[D
ate] ),
            ORDERBY ( 'Calendar'[Date], ASC ),
            KEEP,
            PARTITIONBY ( Customer[CustomerKey] )
        )
    )
)
```

Inizio Finestra Sfasamento

- (●) 1
- ( ) 2
- ( ) 3

Customer Key	Date	Sales	Sales OFFSET Customer Date
12055	25/08/2003	30 €	
12055	07/09/2003	5 €	30 €
12055	02/07/2004	24 €	5 €
12055	19/07/2004	45 €	24 €
12300	24/11/2001	3.578 €	
12300	24/07/2003	2.498 €	3.578 €
12300	10/09/2003	2.393 €	2.498 €
12300	20/11/2003	2.320 €	2.393 €
12300	19/04/2004	2.453 €	2.320 €
Totale		**13.347 €**	

Figura 5

Il codice usa la funzione *OFFSET*, progettata allo scopo di creare, appunto, uno sfasamento, già descritta in questo articolo in un momento in cui i parametri erano meno perché non era ancora *generally available*, e che ha oggi i seguenti input:

1. **l'entità dello sfasamento**, cioè il numero di righe da saltare e la relativa direzione (segno + o -), nel caso in esame è stato usato il valore selezionato del parametro campo *Inizio Finestra Sfasamento*, 1 in figura 5, cambiato di segno per indicare di volere andare indietro, in modo da rendere il tutto dinamicamente selezionabile dall'utente se l'entità dello sfasamento desiderato fosse diversa da uno;
2. **la tabella su cui creare il contesto nel quale calcolare lo sfasamento**, nel caso in esame una tabella di due colonne recuperate,

tramite *SUMMARIZE*, a partire da *ALLSELECTED (Sales): Customer[CustomerKey]* e *'Calendar'[Date]*;

3. **l'ordinamento dei valori della tabella del punto** 2, attraverso la chiamata a *ORDERBY ()* in modo da chiarire cosa si intenda per precedente/successivo, nel caso in esame *'Calendar'[Date]* in ordine crescente;
4. **un settaggio su come comportarsi in caso di valori *BLANK ()* coinvolti nell'ordinamento del punto 3**, nel caso in esame il valore *KEEP* che posiziona i valori BLANK () tra il valore zero e i valori negativi nel caso si tratti di numeri e prima di qualunque stringa, incluse le vuote, nel caso di stringhe;
5. **un settaggio di partizionamento della tabella del punto 2**, attraverso la chiamata a *PARTITIONBY ()*, nel caso in esame si vuole procedere cliente per cliente in modo da ordinare le date, appunto, cliente per cliente e non tutte insieme. Senza la chiamata a *PARTITIONBY*, in figura 5, il calcolo per ogni cliente potrebbe coinvolgere date di un altro.

Per ottimizzare il calcolo, nel caso in esame, si potrebbe avere come secondo parametro la tabella *SUMMARIZE (CALCULATETABLE (Sales, REMOVEFILTERS ('Calendar'[Date])))*, invece di *SUMMARIZE (ALLSELECTED (Sales): Customer[CustomerKey]* e *'Calendar'[Date])*, in modo da considerare solo le date del cliente nel contesto. La scelta *ALLSELECTED*, tuttavia, è quella di default delle funzioni WINDOW nel caso il secondo parametro sia omesso (non tutti i parametri sono obbligatori) – nel qual caso è obbligatorio indicare i parametri 3 e 5 dalle cui colonne verrà creata la chiamata a *ALLSELECTED (Colonna1 ORDERBY, … , ColonnaN ORDERBY, Colonna1 PARTITIONBY, … , ColonnaN PARTITIONBY)* con il vincolo, solo in questo caso, che le colonne appartengano alla stessa tabella. Ecco il codice modificato:

```
Sales OFFSET Customer Date Ottimizzata =
IF (
    HASONEVALUE ( Customer[CustomerKey] ),
    CALCULATE (
        [Sales],
        OFFSET (
            – [Valore Inizio Finestra Sfasamento],
            SUMMARIZE (
                CALCULATETABLE ( Sales, REMOVEFILTERS ( 'Calendar'[Date] ) ),
                Customer[CustomerKey],
                'Calendar'[Date]
            ),
```

```
        ORDERBY ( 'Calendar'[Date], ASC ),
        KEEP,
        PARTITIONBY ( Customer[CustomerKey] )
    )
  )
)
```

In figura 6, le due misure a confronto, che danno gli stessi risultati.

Inizio Finestra Sfasamento

- (●) 1
- ( ) 2
- ( ) 3

Customer Key	Date	Sales	Sales OFFSET Customer Date	Sales OFFSET Customer Date Ottimizzata
12055	25/08/2003	30 €		
12055	07/09/2003	5 €	30 €	30 €
12055	02/07/2004	24 €	5 €	5 €
12055	19/07/2004	45 €	24 €	24 €
12300	24/11/2001	3.578 €		
12300	24/07/2003	2.498 €	3.578 €	3.578 €
12300	10/09/2003	2.393 €	2.498 €	2.498 €
12300	20/11/2003	2.320 €	2.393 €	2.393 €
12300	19/04/2004	2.453 €	2.320 €	2.320 €
Totale		**13.347 €**		

Figura 6

Secondo calcolo: si vuole ottenere il valore di *Sales*, cliente per cliente, in un intervallo di date attorno a quella nel contesto. Come estremi dell'intervallo su useranno, rispettivamente, il valore selezionato del parametro campo *Inizio Finestra Sfasamento* (1, in figura 7) e quello del parametro campo *Fine Finestra Sfasamento* (2, in figura 7). Il calcolo dovrà fornire, per il cliente 12055, il valore 59 alla data del 25/08/2003 (BLANK () + 30 + 5 + 24), il valore 104 alla data del 07/09/2003 (30 + 5 + 24 + 45), il valore 74 alla data del 02/07/2004 (5 + 24 + 45 + BLANK ()) e il valore 69 alla data del 19/07/2004 (24 + 45 + BLANK () + BLANK ()). In

modo analogo, verranno calcolati i valori per l'altro cliente, 12300. Il codice della versione ottimizzata è a seguire e il risultato è mostrato in figura 7.

```
Sales WINDOW Customer Date Ottimizzata =
IF (
    HASONEVALUE ( Customer[CustomerKey] ),
    CALCULATE (
        [Sales],
        WINDOW (
            – [Valore Inizio Finestra Sfasamento],
            REL,
            [Valore Fine Finestra Sfasamento],
            REL,
            SUMMARIZE (
                CALCULATETABLE ( Sales, REMOVEFILTERS ( 'Calendar'[Date] ) ),
                Customer[CustomerKey],
                'Calendar'[Date]
            ),
            ORDERBY ( 'Calendar'[Date] ),
            KEEP,
            PARTITIONBY ( Customer[CustomerKey] )
        )
    )
)
```

Inizio Finestra Sfasamento

- (●) 1
- ( ) 2
- ( ) 3

Fine Finestra Sfasamento

- ( ) 1
- (●) 2
- ( ) 3
- ( ) 4
- ( ) 5

Customer Key	Date	Sales	Sales WINDOW Customer Date	Sales WINDOW Customer Date Ottimizzata
12055	25/08/2003	30 €	59 €	59 €
12055	07/09/2003	5 €	104 €	104 €
12055	02/07/2004	24 €	74 €	74 €
12055	19/07/2004	45 €	69 €	69 €
12300	24/11/2001	3.578 €	8.470 €	8.470 €
12300	24/07/2003	2.498 €	10.790 €	10.790 €
12300	10/09/2003	2.393 €	9.664 €	9.664 €
12300	20/11/2003	2.320 €	7.166 €	7.166 €
12300	19/04/2004	2.453 €	4.773 €	4.773 €
Totale		**13.347 €**		

Figura 7

Il codice usa la funzione *WINDOW*, progettata allo scopo di creare, appunto, una finestra attorno alla riga corrente, che ha i seguenti input:

1. **gli estremi della finestra**, cioè il numero di righe da saltare, rispetto alla riga nel contesto, per individuare il primo estremo e quello per individuare l'ultimo estremo con la relativa direzione (segno + o -), nel caso in esame sono stati usati, rispettivamente, il valore selezionato del parametro campo *Inizio Finestra Sfasamento*, 1 in figura 7, cambiato di segno per indicare di volere andare indietro, e il valore selezionato del parametro campo *Fine Finestra Sfasamento*, 2 in figura 7. I due valori possono essere indicati come relativi (*REL*), cioè calcolati a partire dalla riga corrente o assoluti (*ABS*), cioè calcolati a partire dalla posizione assoluta delle righe della finestra della partizione. Mentre per *OFFSET* e, come vedremo, per *INDEX*, il primo parametro è un

singolo valore, nel caso di *WINDOW* si arriva, dunque, fino a quattro valori (si possono omettere i valori di *ABS* e *REL*, nel caso *REL* è il default;

2. **la tabella su cui creare il contesto nel quale determinare la finestra di righe**, nel caso in esame una tabella di due colonne recuperate, tramite *SUMMARIZE*, a partire da *CALCULATETABLE (Sales, REMOVEFILTERS ('Calendar'[Date]))* in modo da considerare tutte le sole date del cliente nel contesto;
3. **l'ordinamento dei valori della tabella del punto** 2, attraverso la chiamata a *ORDERBY ()* in modo da chiarire cosa si intenda per precedente/successivo, nel caso in esame *'Calendar'[Date]* in ordine crescente;
4. **un settaggio su come comportarsi in caso di valori *BLANK ()* coinvolti nell'ordinamento del punto 3**, nel caso in esame il valore *KEEP* che posiziona i valori BLANK () tra il valore zero e i valori negativi nel caso si tratti di numeri e prima di qualunque stringa, incluse le vuote, nel caso di stringhe;
5. **un settaggio di partizionamento della tabella del punto 2**, attraverso la chiamata a *PARTITIONBY ()*, nel caso in esame si vuole procedere cliente per cliente in modo da ordinare le date, appunto, cliente per cliente e non tutte insieme. Senza la chiamata a *PARTITIONBY*, in figura 7, il calcolo per ogni cliente potrebbe coinvolgere date di un altro.

Anche in questo caso, non tutti i parametri sono obbligatori.

Terzo calcolo: si vuole ottenere il valore di *Sales*, cliente per cliente, in corrispondenza di una data specifica all'interno dell'insieme di tutte le date di ogni cliente, specificandone la posizione assoluta. Come posizione assoluta si userà il valore selezionato del parametro campo *Inizio Finestra Sfasamento* (1 in figura 8). Il calcolo dovrà fornire, per il cliente 12055, il valore 30 in tutte le relative date (25/08/2003, 07/09/2003, 02/07/2004 e 19/07/2004), visto che quello è il valore alla prima data in senso assoluto del cliente (25/08/2003), ordinando le date in senso crescente. In modo analogo, verranno calcolati i valori per l'altro cliente, 12300. Il codice della versione ottimizzata è a seguire e il risultato è mostrato in figura 8.

```
Sales INDEX Customer Date Ottimizzata =
IF (
   HASONEVALUE ( Customer[CustomerKey] ),
   CALCULATE (
      [Sales],
```

```
    INDEX (
        [Valore Inizio Finestra Sfasamento],
        SUMMARIZE (
            CALCULATETABLE ( Sales, REMOVEFILTERS ( 'Calendar'[Date] ) ),
            Customer[CustomerKey],
            'Calendar'[Date]
        ),
        ORDERBY ( 'Calendar'[Date] ),
        KEEP,
        PARTITIONBY ( Customer[CustomerKey] )
    )
  )
)
```

Inizio Finestra Sfasamento

- (●) 1
- ( ) 2
- ( ) 3

Customer Key	Date	Sales	Sales INDEX Customer Date	Sales INDEX Customer Date Ottimizzata
12055	25/08/2003	30 €	30 €	€ 30
12055	07/09/2003	5 €	30 €	€ 30
12055	02/07/2004	24 €	30 €	€ 30
12055	19/07/2004	45 €	30 €	€ 30
12300	24/11/2001	3.578 €	3.578 €	€ 3.578
12300	24/07/2003	2.498 €	3.578 €	€ 3.578
12300	10/09/2003	2.393 €	3.578 €	€ 3.578
12300	20/11/2003	2.320 €	3.578 €	€ 3.578
12300	19/04/2004	2.453 €	3.578 €	€ 3.578
Totale		**13.347 €**		

Figura 8

Il codice usa la funzione *INDEX*, progettata allo scopo di restituire una riga specifica all'interno dell'insieme delle righe da considerare, indicandone la posizione in modo assoluto. La funzione ha i seguenti input:

1. **la posizione assoluta della riga** e la relativa direzione (segno + o -), nel caso in esame è stato usato il valore selezionato del parametro campo *Inizio Finestra Sfasamento*, 1 in figura 8. Nel caso di un numero negativo, il conteggio parte dall'ultima posizione dell'insieme delle righe (-1 è l'ultima riga, -2 la penultima e così via);
2. **la tabella su cui creare il contesto nel quale individuare la posizione assoluta della riga da restituire**, nel caso in esame una tabella di due colonne recuperate, tramite *SUMMARIZE*, a partire da *CALCULATETABLE (Sales, REMOVEFILTERS ('Calendar'[Date]))*, in modo da considerare tutte le sole date del cliente nel contesto;
3. **l'ordinamento dei valori della tabella del punto** 2, attraverso la chiamata a *ORDERBY ()* in modo da chiarire cosa si intenda per precedente/successivo, nel caso in esame *'Calendar'[Date]* in ordine crescente;
4. **un settaggio su come comportarsi in caso di valori *BLANK ()* coinvolti nell'ordinamento del punto 3**, nel caso in esame il valore *KEEP* che posiziona i valori BLANK () tra il valore zero e i valori negativi nel caso si tratti di numeri e prima di qualunque stringa, incluse le vuote, nel caso di stringhe;
5. **un settaggio di partizionamento della tabella del punto 2**, attraverso la chiamata a *PARTITIONBY ()*, nel caso in esame si vuole procedere cliente per cliente in modo da ordinare le date, appunto, cliente per cliente e non tutte insieme. Senza la chiamata a *PARTITIONBY*, in figura 8, il calcolo per ogni cliente potrebbe coinvolgere date di un altro.

Anche qui, non tutti i parametri sono obbligatori. Si rimanda alla ottima DAX guide per ulteriori dettagli su ognuna delle tre funzioni e i relativi sviluppi (ricordiamo che queste funzioni sono ancora in *preview*, il team DAX sta ancora lavorando per sviluppare funzionalità aggiuntive e risolvere bachi).

Passiamo adesso al **secondo modello**, questa volta in *DirectQuery* su un Database SQL Server *on-premises*. La ragione dell'uso di questo secondo modello è proprio la connessione in *DirectQuery* che rende sconsigliato vivamente l'uso della *Time-Intelligence* con *CALCULATE*. Le query generate da *Power View* verso *Tabular*, in questo caso, devono essere tradotte in parte in SQL, il linguaggio della sorgente. Data la natura piuttosto complessa delle query, esse diventano estremamente lunghe in SQL, restituendo prestazioni estremamente degradate. In particolare,

visto che la *Time-Intelligence* in *Tabular* lavora con granularità giornaliera, SQL riceverà calcoli da effettuare data per data. Ciò rende più complesso per Power BI usare delle aggregazioni per migliorare le prestazioni. Infatti, quando si è in *DirectQuery*, si crea spesso in Power Query, ed importa, una tabella aggregata al livello di categorie di prodotto e/o di anni e/o di categorie di clienti in modo che, per quanto possibile, Power BI indirizzi le query a quelle tabelle che, essendo in *Import*, risponderanno molto velocemente, riducendo le probabilità di indirizzare le query a SQL. Il riferimento è la figura 2. Prima di mostrare il tutto, ringraziamo Chris Webb per il suo illuminante articolo da cui traiamo liberamente ispirazione.

Si supponga, dunque, di volere calcolare il valore una misura un anno indietro rispetto al periodo correntemente nel *filter context*. La scelta classica, in questo caso, è usare come filtro di *CALCULATE* la tabella di date, costituita da una singola colonna, restituita dalla funzione *DATEADD ('Date'[Datekey], -1, YEAR)* che, per brevità è chiarezza, ha un alias in *SAMEPERIODLASTYEAR ('Date'[Datekey])*. Useremo l'acronimo *SPLY* per indicare la misura che usa questo filtro di *CALCULATE*.

In figura 9 è mostrata una matrice che riporta, in *Righe*, l'anno di calendario e, in *Valori*, le misure *Transazioni* e *Transazioni AP SPLY*, dove AP indica l'anno precedente. Il codice delle misure è a seguire.

CalendarYear	Transazioni	Transazioni AP SPLY
2007	4.000.025	
2008	3.626.523	4.000.025
2009	5.001.060	3.626.523
2010		5.001.060
Totale	**12.627.608**	

Figura 9

```
Transazioni =
COUNTROWS ( Sales )
Transazioni AP SPLY =
IF (
    HASONEVALUE ( 'Date'[CalendarYear] ),
```

```
    CALCULATE ( [Transazioni], SAMEPERIODLASTYEAR ( 'Date'[Datekey] ) )
)
```

Tramite l'Analizzatore prestazioni di Power BI, abbiamo catturato la query generata da *Power View* verso *Tabular*, che è in minima parte scritta in DAX e, per la maggior parte, in SQL. Copiata la parte SQL, abbiamo eseguito la query in SQL Server Management Studio, per verificarne l'output.

Per prima cosa, la query generata è, come preannunciato, estremamente complessa. Inoltre, lavora al livello delle singole date, risultando molto lunga: circa 2.500 righe di codice SQL. In figura 10 è mostrato l'output della query dove le singole date sono evidenziate.

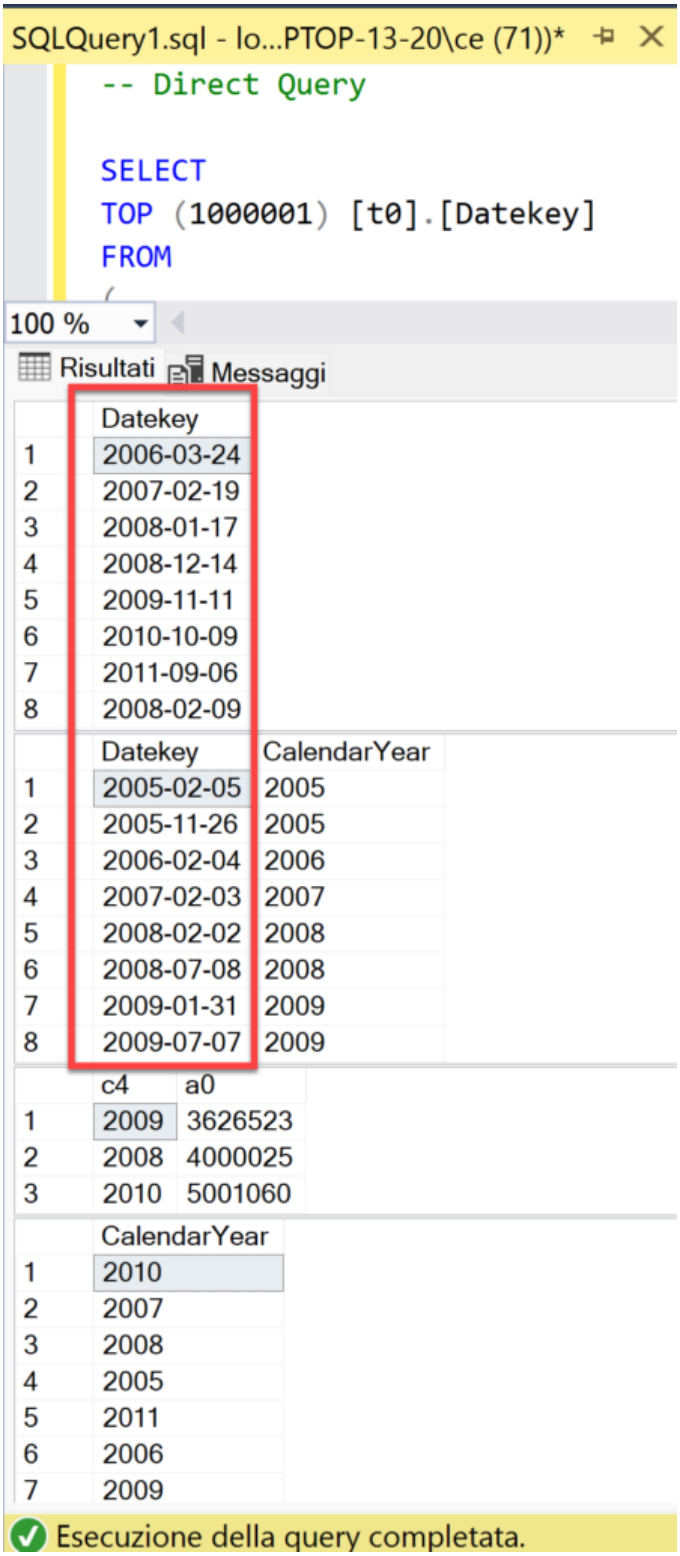

Figura 10

L'idea di Chris Webb è di verificare il codice SQL generato dalla query originata dall'uso di *OFFSET* come filtro di *CALCULATE* per ottenere lo stesso calcolo in modo visuale. La misura è chiamata *Transazioni AP OFFSET*, il cui risultato, identico a quello di *Transazioni AP SPLY*, è mostrato in figura 11 e il cui codice è a seguire.

CalendarYear	Transazioni	Transazioni AP SPLY	Transazioni AP OFFSET
2007	4.000.025		
2008	3.626.523	4.000.025	4.000.025
2009	5.001.060	3.626.523	3.626.523
2010		5.001.060	5.001.060
Totale	**12.627.608**		

Figura 11

```
Transazioni AP OFFSET =
IF (
    HASONEVALUE ( 'Date'[CalendarYear] ),
    CALCULATE (
        [Transazioni],
        OFFSET (
            -1,
            ALLSELECTED ( 'Date'[CalendarYear] ),
            ORDERBY ( 'Date'[CalendarYear] )
        )
    )
)
```

La query, questa volta, è molto semplice e consta di circa 120 righe soltanto. In figura 12 è mostrato l'output della query, dove si nota che la granularità è al livello di interi anni, dunque estremamente più efficiente.

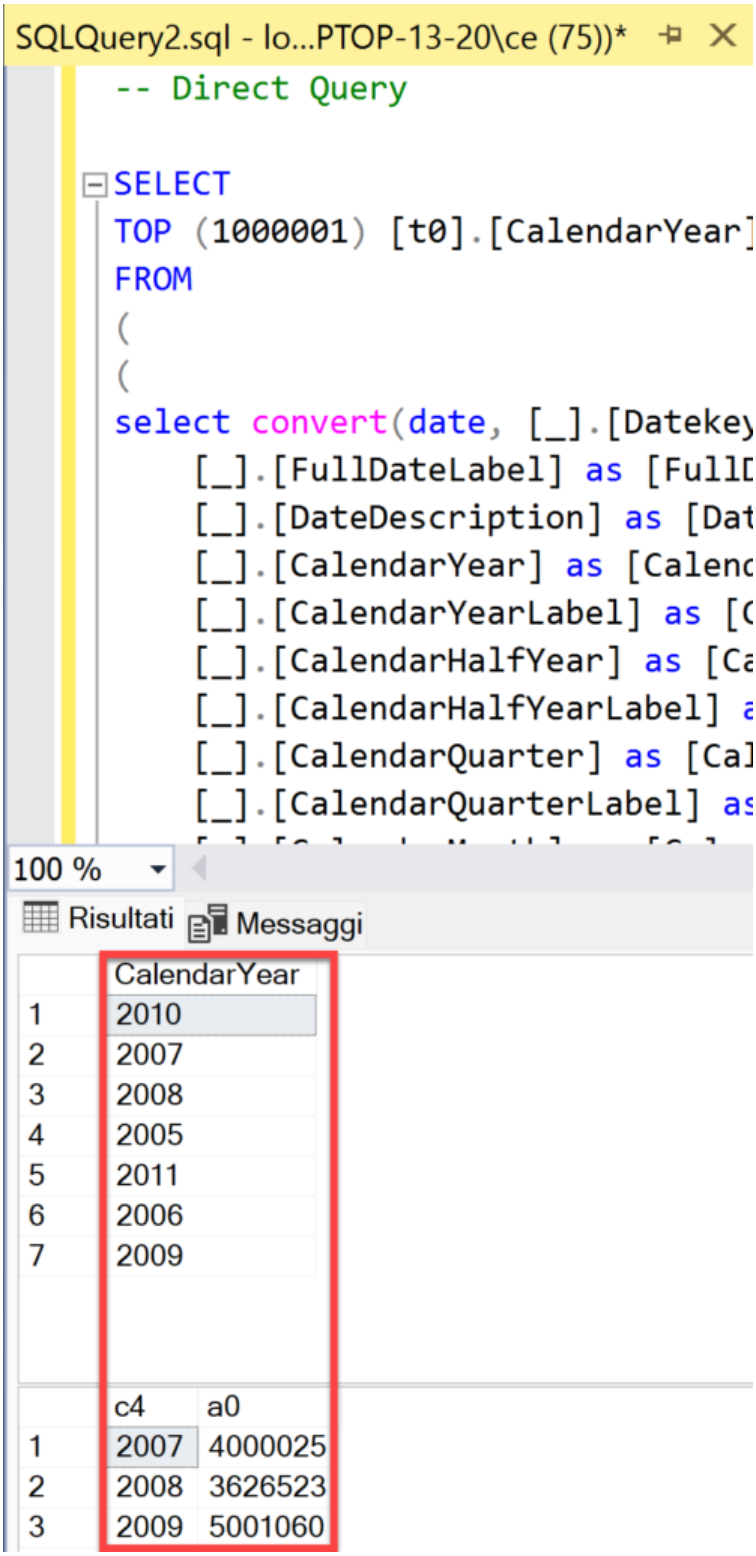

Figura 12

Per concludere questa digressione, che dire delle prestazioni delle due misure nel caso di uso di aggregazioni? DAX Studio ci offre una misura, che purtroppo però non è precisa quando i tempi di esecuzione sono molto rapidi, come capita quando viene usata un'aggregazione e dunque la query è interamente in DAX. Per apprezzare le differenze bisognerebbe usare un modello più esteso, ci riserviamo di farlo in futuro. Tuttavia, ecco i risultati applicati allo stesso modello della figura 2, ma a cui è stata aggiunta una tabella di aggregazione in Import, *Sales Aggregata*, creata con Power Query sulla base della tabella *Sales*, quest'ultima sempre in *DirectQuery*. La tabella *'Date'* è stata posta in modalità *Doppia*. Il modello è osservabile in figura 13.

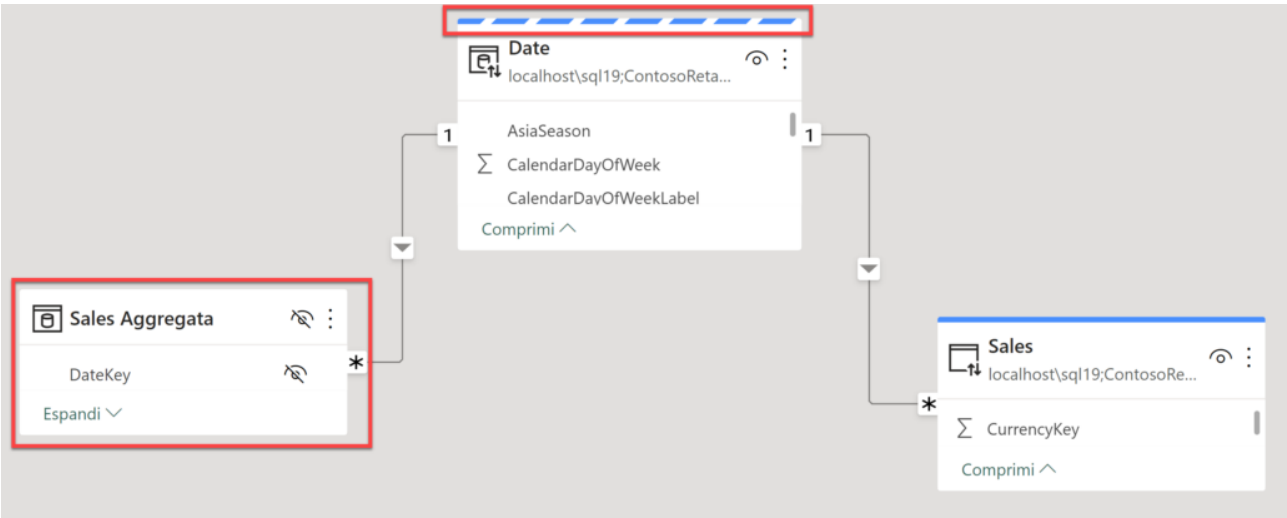

Figura 13

In figura 14, le prestazioni della misura *Transazioni AP SPLY*, in figura 15 quelle della misura *Transazioni AP OFFSET*. Entrambe usano l'aggregazione, ma nuovamente, come era atteso, il numero di righe coinvolte dalla query di *Transazioni AP OFFSET* è molto inferiore a quello della query di *Transazioni AP SPLY*, il che è un bene e porta a considerare l'uso di queste funzioni una validissima alternativa alla classica *Time-Intelligence*, specialmente in *DirectQuery*.

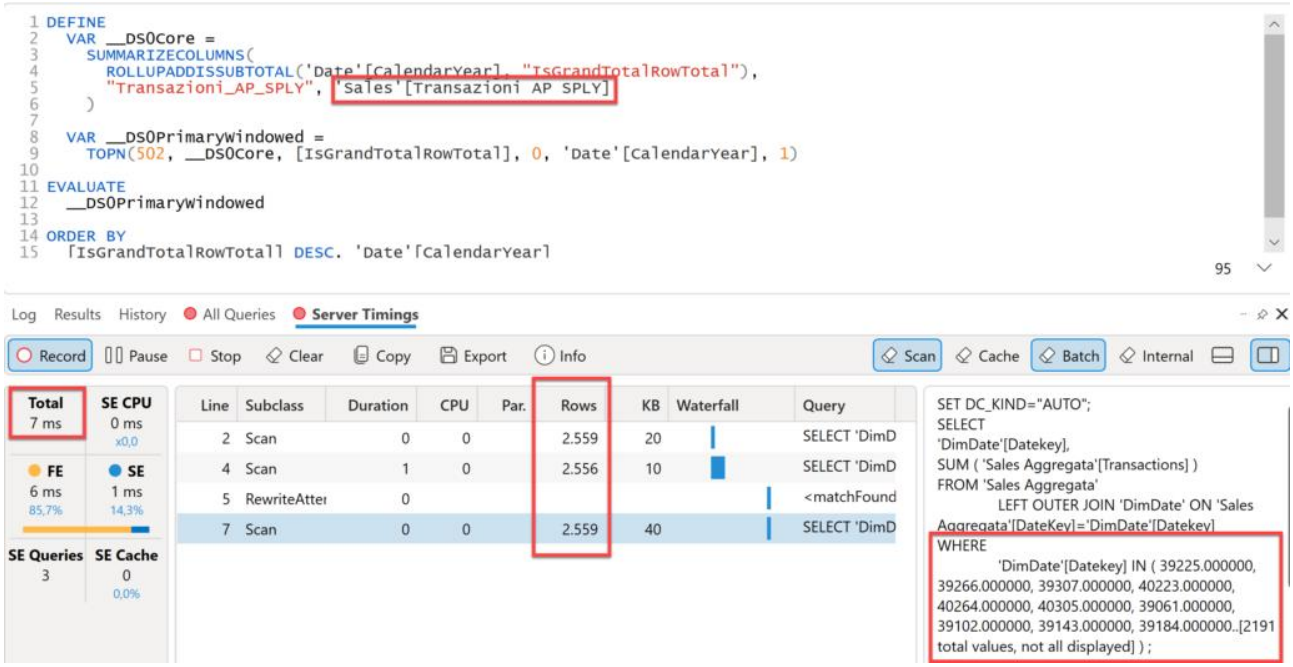

Figura 14

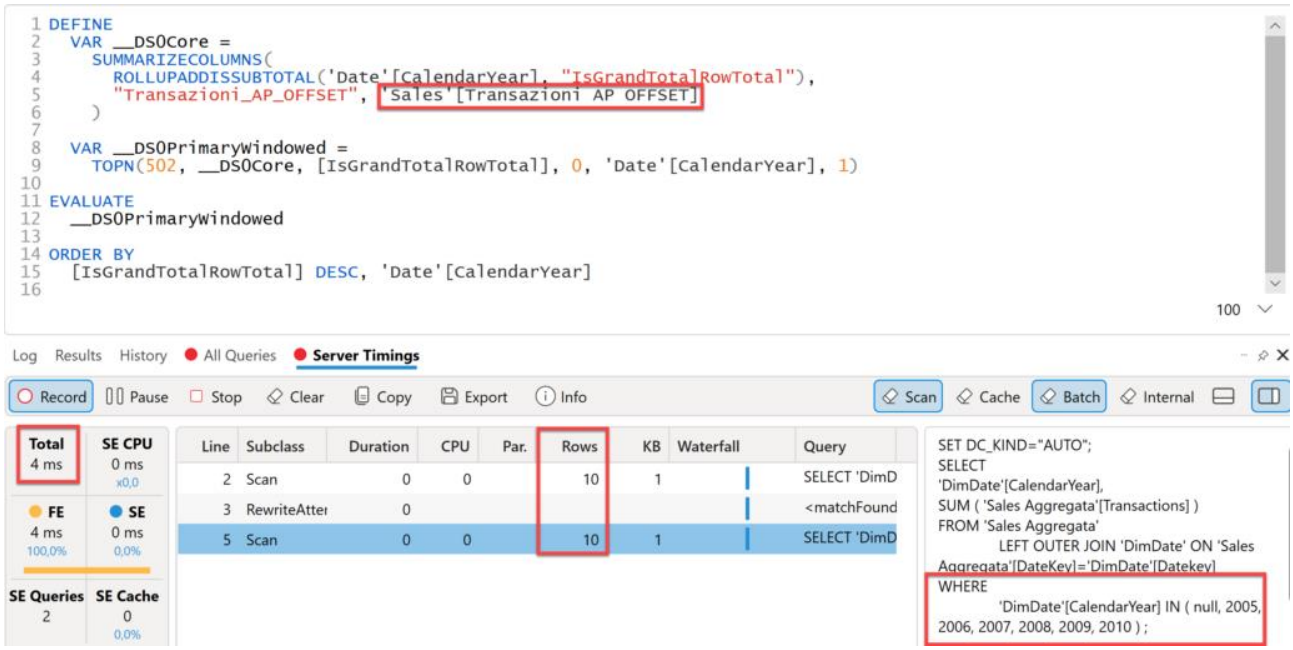

Figura 15

Infine, come accennato ad inizio articolo, precisiamo un punto rispetto al precedente articolo: l'uso di una misura come criterio di ordinamento è, in effetti, possibile; tuttavia Intellisense non riconosce la sintassi (e mostra, dunque, un falso errore) e la possibilità è limitata all'uso di posizioni assolute, cioè all'uso

di *WINDOW* con posizioni *ABS* e di *INDEX* (*OFFSET* non offre questa possibilità in quanto progettata per sfasamenti relativi). Ecco a seguire due esempi relativi al modello in *Import* (figura 16) e, a seguire, il codice delle misure.

Inizio Finestra Sfasamento
1
2
3

Fine Finestra Sfasamento
1
2
3
4
5

Customer Key	Date	Sales	Sales WINDOW Customer Date Ottimizzata_Misura	Sales INDEX Customer Date Ottimizzata_Misura
12300	24/11/2001	3.578 €	4.713 €	2.498 €
12300	24/07/2003	2.498 €	4.713 €	2.498 €
12300	19/04/2004	2.453 €	4.713 €	2.498 €
12300	10/09/2003	2.393 €	4.713 €	2.498 €
12300	20/11/2003	2.320 €	4.713 €	2.498 €
12055	19/07/2004	45 €	29 €	30 €
12055	25/08/2003	30 €	29 €	30 €
12055	02/07/2004	24 €	29 €	30 €
12055	07/09/2003	5 €	29 €	30 €
Totale		**13.347 €**		

Figura 16

```
Sales WINDOW Customer Date Ottimizzata_Misura =
IF (
    HASONEVALUE ( Customer[CustomerKey] ),
    CALCULATE (
        [Sales],
        WINDOW (
            – [Valore Inizio Finestra Sfasamento],
            ABS,
            [Valore Fine Finestra Sfasamento],
            ABS,
            ADDCOLUMNS (
                SUMMARIZE (
                    CALCULATETABLE ( Sales, REMOVEFILTERS ( 'Calendar'[Date] ) ),
```

```
            Customer[CustomerKey],
            'Calendar'[Date]
        ),
        "@Sales", [Sales]
    ),
    ORDERBY ( [@Sales], DESC ),
    KEEP,
    PARTITIONBY ( Customer[CustomerKey] )
  )
 )
)


Sales INDEX Customer Date Ottimizzata_Misura =
IF (
  HASONEVALUE ( Customer[CustomerKey] ),
  CALCULATE (
    [Sales],
    INDEX (
      [Valore Inizio Finestra Sfasamento],
      ADDCOLUMNS (
        SUMMARIZE (
          CALCULATETABLE ( Sales, REMOVEFILTERS ( 'Calendar'[Date] ) ),
          Customer[CustomerKey],
          'Calendar'[Date]
        ),
        "@Sales", [Sales]
      ),
      ORDERBY ( [@Sales], DESC ),
      KEEP,
      PARTITIONBY ( Customer[CustomerKey] )
    )
  )
)
```

Conclusioni

Per sintetizzare, *OFFSET* determina uno sfasamento relativo alla riga nel contesto, *WINDOW* un insieme di righe attorno ad essa, i cui estremi possono essere indicati come relativi alla riga nel contesto o come assoluti, *INDEX* individua una particolare riga, con posizione assoluta, nell'insieme delle righe disponibili. *WINDOW*, se usata con posizioni assolute, e *INDEX* possono usare anche

una misura come criterio di ordinamento. Queste funzioni rendono più semplici i calcoli intra-riga, evitando di dovere ricorrere a *RANKX*, estremamente ottimizzata ma piuttosto complessa da usare. Inoltre, hanno applicazioni in molteplici contesti, per esempio la *Time-Intelligence*, dove rappresentano una valida alternativa alla classica tecnica (*CALCULATE* e *DATEADD*), molto inefficiente in quella modalità di archiviazione in quanto genera query lunghe e complesse in SQL. Seguiranno altri articolo con esempi di comparazione tra le vecchie tecniche in DAX e le nuove possibilità aperte dalle funzioni WINDOW. Non sempre queste ultime rappresentano una semplificazione, tuttavia più tecniche esistono a disposizione del progettista che studia, meglio è.

Nota: non abbiamo allegato i file .pbix in *DirectQuery* (con o senza aggregazioni) in quanto non funzionerebbero, in assenza della corrispondente istanza di SQL Server *on-premises*.

file. pbixDownload