Querying – Thought Experiments – Capítulo 1

1. How come you cannot use an alias you define in the SELECT list in the WHERE clause,

or even the same SELECT clause? Where can you use such an alias?

Não é possível utilizar um alias que vc definiu no SELECT na lista de WHERE pq o SELECT é lido depois do WHERE, ou seja, quando a máquina processa o WHERE, o SELECT ainda não foi executado e portanto o alias não foi criado

1. What are the differences between joins and set operators?

Os operadores UNION colocam as linhas das consultas em seguida uma da outra; os operadores JOIN fazem uma espécie de produto cartesiano, gerando o resultado baseado nos cruzamentos iguais das duas ou mais consultas

1. What could prevent SQL Server from treating a query filter optimally, meaning, from

using an index efficiently to support the filter? What other query elements could also

be affected in a similar manner and what can you do to get optimal treatment?

1. What is the difference between the ON and WHERE clauses?

O ON deve ser usado preferencialmente cruzarmos dados de tabelas diferentes, utilizamos o ON para juntarmos duas tabelas diferentes, sem controle sobre o que será retornado, podendo trazer dados das duas tabelas cruzadas; o WHERE deve ser usado para filtrarmos dados. Apesar de podermos usar ON para juntar **E** filtrar dados, a boa prática indica que devemos utilizar o WHERE para filtrar os dados e utilizar o JOIN para juntarmos os dados.

1. Explain what function determinism means and what are the implications of using non-

deterministic functions?

Funções determinísticas são aquelas que sempre retornam o mesmo resultado, se utilizadas com os mesmos parâmetros. Exemplo: a função DATEDIFF. Essa função retornará sempre o mesmo valor quando utilizados parâmetros constantes. Por exclusão, funções não determinísticas são aquelas que retornam resultados diferentes quando executadas. Exemplo de função não determinística é a função GETDATE(), que **não retornará** o mesmo resultado se executada em momentos diferentes.

O impacto de utilizar uma função que retorna valores diferentes ao longo do tempo é que esse valor será sempre diferente, mesmo que o banco continue o mesmo, outra implicação é a criação de índices. Se você cria uma coluna calculada que se utiliza de uma função não determinística, você não pode criar um índice nesta coluna, uma vez que os valores desta coluna serão diferentes ao longo do tempo

1. What are the differences between DELETE and TRUNCATE TABLE?

O DELETE é utilizado para apagar dados de um banco; é criado log de todas as linhas excluídas, facilitando assim a restauração, caso necessária. DELETE é uma clausula DML (DML, data manipulation language), ou seja, trata diretamente da manipulação, neste caso a exclusão, de linhas. A cláusula pode (e deve) ser utilizada com um filtro para a exclusão.

O TRUNCATE é mais rápido na sua execução, pois armazena **menos** logs, tornando-se mais difícil de restaurar as linhas excluídas. TRUNCATE é uma instrução DDL (DDL, data definition language) e opera sem a necessidade de filtros.

1. You need to perform a multi-row insert into a target table that has a column with an

identity property. You need to capture the newly generated identity values for further

processing. How can you achieve this?

Os valores novos, gerados pela coluna identity pode ser armazenado em uma variável para processamento posterior. A variável mais indicada neste caso seria uma variável do tipo TABLE, onde seria armazenado os índices recém gerados ou qualquer informação necessária para processamento posterior.

1. When should you use the WITH VALUES clause explicitly as part of adding a column to

a table?

Quando a nova coluna criada aceitar NULL, você deve especificar um valor em WITH VALUES, para que a as linhas existentes da coluna nova recebam o valor definido em DEFAULT; se a coluna nova não aceitar NULL, mesmo que você não especifique valores para WITH VALUES, as linhas existentes da coluna nova receberá os valores informados em DEFAULT.

Querying – Thought Experiments – Capítulo 2

* Pontos de destaque deste capítulo: **O operador APPLY, nos modos CROSS APPLY ou OUTER APPLY:**

**O operador CROSS APPLY** avalia expressões do lado direito e do lado esquerdo e, caso o retorno da tabela da direita retorne uma linha ou conjunto de linhas vazias para a tabela da esquerda, a linha/conjunto não é retornada. O motivo do nome CROSS APPLY tem a ver com o fato deste operador funcionar como uma espécie de CROSS JOIN entre duas tabelas. As figuras abaixo ilustram bem:



Na esquerda as letras X, Y, Z funcionam como valores chaves da tabela da esquerda. F representa a expressão de tabela passadas como input pela tabela da direita e, entre os parênteses, o elemento que serve de correlação entre as duas tabelas.

Na direita, nós vemos o resultado retornado pela expressão da tabela da direita, para cada linha da tabela da esquerda.

Abaixo nós vemos o resultado do operador CROSS APPLY, onde cada linha da tabela da esquerda é combinada com o resultado retornado pela tabela da direita. Perceba que, onde linhas da tabela da esquerda que não retornaram nada na tabela da direita não são retornadas pelo CROSS APPLY.

* **Expressões de tabela** são construções em T-SQL que, basicamente, são queries nomeadas. Você escreve uma query interna que retorna um resultado em tabela, você nomeia esse resultado e consulta o **resultado** para uma query externa. Os tipos de expressões de tabela são **CTE, tabelas derivadas, views e funções que retornam tabelas (table valued functions)**.

As queries internas não podem ter a cláusula ORDER BY, todas colunas precisam possuir nomes, ou seja, os seus resultados precisam ser relacionais.

**Expressões de tabela (CTE) ou tabelas temporárias?** Essa dúvida depende da sua necessidade. Se você escreve uma query e quer interagir com o resultado desta query e escreve outra query para interagir com o resultado da primeira, CTE é mais indicado pois é mais performático devido ao processamento físico da engine do SQL lidar melhor; agora, se você precisa guardar os valores da primeira query para utilizar depois, em outro contexto etc. guardar esses resultados em uma tabela temporária é mais adequado.

**Tipos de expressões de tabela que o T-SQL suporta** são os seguintes:

* **Tabelas derivadas** – Tabelas derivadas são queries nomeadas que retornam uma tabela. Você especifica a tabela derivada dentro da cláusula FROM entre parênteses e após os parênteses você dá o nome. Problemas deste approach: quando você começa a referenciar diversas tabelas derivadas uma dentro das outras. Esse encadeamento dificulta a leitura e compreensão do código. Outro problema são os JOINS, que são avaliados de uma vez só, tratando query interna e externa como um único conjunto
* **Common table expressions (CTE) –** Semelhante a tabela derivada no sentido de que é uma query nomeada visível apenas para a declaração que a utiliza; inclusive a estrutura é a mesma:
  + - * Query interna
      * Nome que você dá para a query
      * Query externa

Apesar da estrutura semelhante, as CTEs utilizam os três componentes de forma diferente:

WITH <nome\_CTE>

AS

(

  <query\_interna>

)

<query\_externa>;

Exemplo de CTE:

WITH C AS

(

  SELECT

    ROW\_NUMBER() OVER(PARTITION BY categoryid

                      ORDER BY unitprice, productid) AS rownum,

    categoryid, productid, productname, unitprice

  FROM Production.Products

)

SELECT categoryid, productid, productname, unitprice

FROM C

WHERE rownum <= 2

O conceito e organização é similar a uma tabela derivada, mas com a diferença de que a query interna não é definida no meio da query externa dentro da cláusula FROM; ao invés você **NOMEIA A CTE 🡪 DEFINE A QUERY INTERNA 🡪 DEFINE A QUERY EXTERNA**

Você não encadeia várias CTEs da forma que faz com tabelas derivadas, você as separa por vírgulas, muito mais simples de seguir a lógica e entender o que o seu código está fazendo, exemplo de várias CTEs, uma lendo da outra:

WITH C1 AS

(

  SELECT ...

  FROM T1

  WHERE ...

),

C2 AS

(

  SELECT

  FROM  C1

  WHERE ...

)

SELECT ...

FROM C2

WHERE ...;

* **Views e funções de valor de tabela –** Como vimos acima, CTEs e tabelas derivadas são declarações que são visíveis apenas para o segmento que as executa, não sendo possível referenciar nenhum dos dois tipos fora da declaração que elas foram criadas. Para reusabilidade, precisamos criar um objeto dentro do banco de dados capaz de fornecer isso e pra isso podemos usar as **views** e **funções de valor tabela**.

A diferença básica entre os dois é que as views não aceitam parâmetros e as funções sim. Por exemplo: suponhamos que criamos uma view que atenda as necessidades da primeira query que montamos:

DROP VIEW IF EXISTS Sales.RankedProducts;

GO

CREATE VIEW Sales.RankedProducts

AS

SELECT

  ROW\_NUMBER() OVER(PARTITION BY categoryid

                    ORDER BY unitprice, productid) AS rownum,

  categoryid, productid, productname, unitprice

FROM Production.Products;

GO

O que criamos aqui não foi o resultado dessa declaração SELECT, mas sim a lógica da VIEW, como um objeto fixo no banco, agora toda vez que precisamos acessar o resultado do SELECT, nós referenciamos a view

Sobre as funções de valor de tabela, suponhamos que você queira acessar aquela query que mostra os gerentes acima de um certo funcionário, mas não queremos esse funcionário como uma constante, queremos passar o idfuncionario como um parâmetro.

A lógica é parecida com a criação da view, mas criamos uma função que retorne isso de forma dinâmica:

DROP FUNCTION IF EXISTS HR.GetManagers;

GO

CREATE FUNCTION HR.GetManagers(@empid AS INT) RETURNS TABLE

AS

RETURN

  WITH EmpsCTE AS

  (

    SELECT empid, mgrid, firstname, lastname, 0 AS distance

    FROM HR.Employees

    WHERE empid = @empid

    UNION ALL

    SELECT M.empid, M.mgrid, M.firstname, M.lastname, S.distance + 1 AS distance

    FROM EmpsCTE AS S

      JOIN HR.Employees AS M

        ON S.mgrid = M.empid

  )

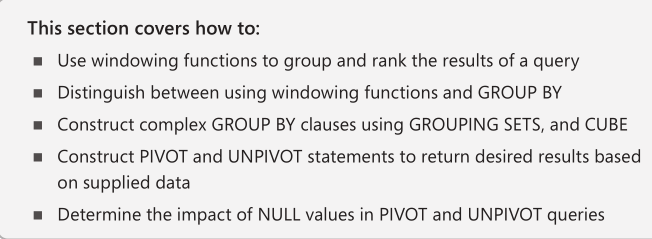
  SELECT empid, mgrid, firstname, lastname, distance

  FROM EmpsCTE;

GO

Perceba que o cabeçalho define o nome da função (HR.GetManagers), especifica o tipo de parâmetro aceito (@empid, INTEIRO) e indica o que a função retorna um resultado de tabela (RETURNS TABLE) especificada na query abaixo

* Pontos de destaque deste capítulo: **agrupando e pivoteando dados utilizando queries:**



Este capítulo cobre uma área importante do acesso e manipulação de dados: agrupamentos de resultados em diversas formas (GROUP BY, GROUP BY com CUBE, ROLLUP e GROUPING SETS) e as diferentes formas de rotação de dados com PIVOT e UNPIVOT nas queries.

* **Trabalhando com um único tipo agrupamento**: Uma query se torna uma query agrupada quando especificada uma função de agrupamento, ou uma cláusula como GROUP BY, ou ambos. Considere a query abaixo:

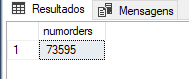
use

WideWorldImporters

SELECT COUNT(\*) AS numorders

FROM Sales.Orders

O resultado apresentado é uma única linha, já que não foi especificado uma função de agrupamento ou uma clausula para tal:



Como não tem um GROUP BY explicito, todas as linhas da tabela Sales.Orders foram agrupadas em um único conjunto e então foi feita a contagem das linhas deste único grupo.

Usando uma cláusula GROUP BY você pode agrupar as linhas em grupos determinados dentro da cláusula. Por exemplo a query abaixo, em que agrupamos por PickedbypersonID, algo como o recebedor do pedido:

USE

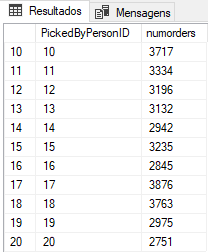
WideWorldImporters

SELECT PickedByPersonID, COUNT(\*) AS numorders

FROM Sales.Orders

GROUP BY PickedByPersonID

A query acima produz o seguinte resultado:



A query identifica e separa o número de pedidos de acordo com cada um dos PickedbypersonID.

Podemos especificar mais de um grupo dentro da clausula GROUP BY, por exemplo na query abaixo onde especificamos o ano:

USE

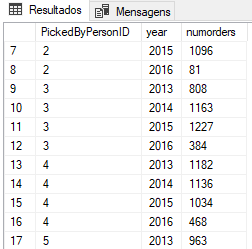
WideWorldImporters

SELECT PickedByPersonID, YEAR(Orderdate) as year, COUNT(\*) AS numorders

FROM Sales.Orders

GROUP BY PickedByPersonID, YEAR(OrderDate)

A query produz o seguinte resultado:



Se você quiser filtrar grupos inteiros, o T-SQL oferece a cláusula HAVING. Assim como na cláusula WHERE, o HAVING avalia e filtra um atributo, mas avalia o atributo em relação a um grupo e não a uma linha como o WHERE.

Suponhamos que você tenha que agrupar número de pedidos e ano de pedido, mas filtrar esse agrupamento para retornar um grupo que tenha um número de pedidos menor que 100; o HAVING faz esse trabalho. A query abaixo explicita:

USE

WideWorldImporters

SELECT PickedByPersonID, YEAR(Orderdate) as year, COUNT(\*) AS numorders

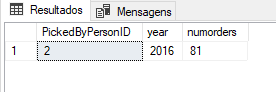
FROM Sales.Orders

WHERE YEAR(OrderDate) IS NOT NULL

GROUP BY PickedByPersonID, YEAR(OrderDate)

HAVING COUNT(\*) < 100

A query produz o seguinte resultado:



O que quer dizer que, o filtro WHERE é processado ao nível de linha antes do agrupamento ser realizado; depois, o agrupamento por pickedbypersonid e ano é realizado; depois o HAVING conta o numero de linhas (pedidos) que são menor que 100; e após isso o SELECT traz o pickedbypersonID, ano e número de pedidos.

* **Trabalhando com vários conjuntos de agrupamento –** É a maneira oferecida pelo T-SQL de agrupar a query (ou queries) em mais de um jeito. As cláusulas utilizadas para este propósito são **CUBE, ROLLUP** e **GROUPING SETS.** É possível utilizar a cláusula GROUPING SETS para especificar quais agrupamentos você quer fazer. A query abaixo define quatro conjuntos de agrupamentos:

USE WideWorldImporters

SELECT

PickedByPersonID

,YEAR(OrderDate) AS shipyear

,COUNT(\*) AS numorders

FROM Sales.Orders

WHERE YEAR(OrderDate) IS NOT NULL --tira os pedidos nao entregues, que tem OrderDate NULL

GROUP BY GROUPING SETS

(

( PickedByPersonID,year(OrderDate) )

,( PickedByPersonID )

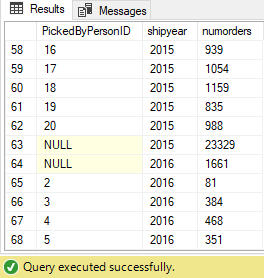
,( year(OrderDate) )

,( )

);

**Os GROUPING SETS** são listados entre parênteses e separados por vírgula. Na query acima listamos 4 agrupamentos: PickedbypersonID e Ano; PickedbypersonID; Ano; e um agrupamento sem parâmetro, que significa um agrupamento com todas as linhas e agregações possíveis.

A query produz o seguinte resultado:



O resultado combina os resultados de agrupamentos e agregações do dado de 4 maneiras diferentes. Os resultados NULL são espaços reservados nas linhas em que o elemento em questão não é parte do GROUPING SET.

Por exemplo, as linhas de resultado que estão associadas com o GROUPING SET YEAR(OrderDate), o PickedbypersonID é NULL. De forma análoga, as linhas associadas com Year(OrderDate) possuem PickedbypersonID NULL.

É possível obter o mesmo resultado escrevendo 4 queries distintas e utilizar o operador UNION ALL para chegarmos no mesmo resultado; obviamente a quantidade de código e compreensão do que está sendo feito é uma desvantagem.

Além do GROUPING SETS, T-SQL aceita outras duas cláusulas adicionais chamadas **CUBE** e **ROLLUP**. A cláusula CUBE aceita uma lista de expressões como inputs e define todos os tipos de agrupamentos possíveis. Por exemplo, a query abaixo é uma equivalente lógica da query feita acima, com GROUPING SETS:

USE WideWorldImporters

SELECT

PickedByPersonID

,YEAR(OrderDate) AS shipyear

,COUNT(\*) AS numorders

FROM Sales.Orders

WHERE YEAR(OrderDate) IS NOT NULL --tira os pedidos nao entregues, que tem OrderDate NULL

GROUP BY CUBE ( year(OrderDate), PickedByPersonID );

**A cláusula CUBE** define todos os 4 conjuntos de agrupamentos possíveis:

* ( PickedByPersonID,year(OrderDate) )
* ,( PickedByPersonID )
* ,( year(OrderDate) )
* ,( )

**A cláusula ROLLUP** é, também, uma abreviação da nossa primeira query com GROUPING SETS, mas é utilizada quando existe uma hierarquia natural dos campos passados como parâmetros. Por exemplo, uma hierarquia geográfica em que exista os campos **country**, **region** e **city**. Com a cláusula ROLLUP poderíamos escrever a seguinte query:

USE WideWorldImporters

SELECT

shipcountry

,shipregion

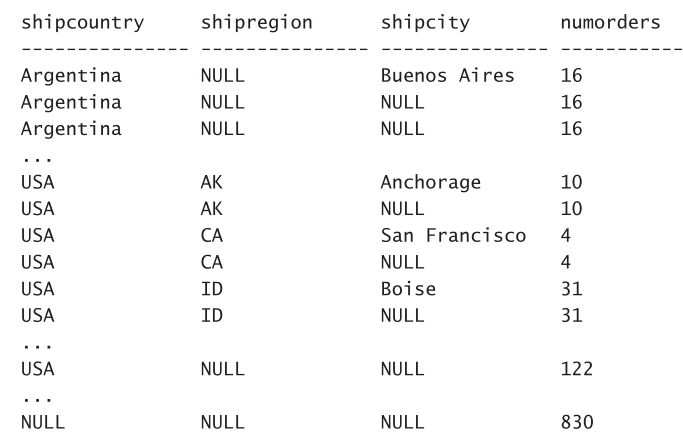
,shipcity

,COUNT(\*) AS numorders

FROM Sales.Orders

GROUP BY ROLLUP( shipcountry, shipregion, shipcity );

O resultado obtido seria o seguinte:



Como mencionado nos exemplos acima, os campos NULL servem como espaço reservado para colunas que não estão associadas com os parâmetros de agrupamento da cláusula **ROLLUP**.

Existe um problema de identificar quais são os agrupamentos feitos quando uma coluna permite valores NULL. Como identificar onde é um valor NULL da coluna ou representa um espaço reservado da cláusula de agrupamento? T-SQL oferece duas alternativas para solucionar esse problema: **GROUPING** e **GROUPING\_ID**.

A função GROUPING aceita um único elemento como input e retorna 0 quando o elemento é parte de um conjunto de agrupamento e retorna 1 quando não é. Em outras palavras, 0 define um elemento que é parte do agrupamento e 1 define um hiperagregado. A query abaixo demonstra a utilização da função GROUPING:

USE WideWorldImporters

SELECT

PickedByPersonID

,GROUPING(PickedByPersonID) AS grpPickedby

,YEAR(OrderDate) AS shipyear

,GROUPING(year(OrderDate)) AS grpYear

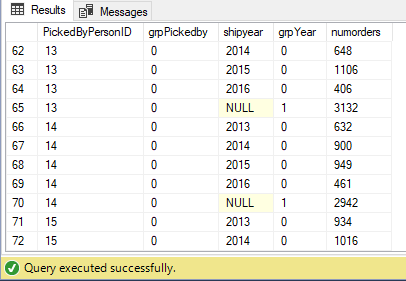
,COUNT(\*) AS numorders

FROM Sales.Orders

WHERE YEAR(OrderDate) IS NOT NULL --tira os pedidos nao entregues, que tem OrderDate NULL

GROUP BY CUBE ( year(OrderDate), PickedByPersonID );

A query produz o seguinte resultado:



Agora é só procurar pelos 0 e identificar quais elementos são partes do conjunto de agrupamento e 1 para o resto.

Além do GROUPING, T-SQL oferece a função **GROUPING\_ID** como alternativa para contornar o problema de identificar colunas utilizadas em conjuntos de agrupamento. GROUPING\_ID aceita vários parâmetros como input e retorna um inteiro representando um bitmap.

**Pivoteando dados utilizando PIVOT e UNPIVOT**, pivot os dados é um método específico de agregar os dados. Unpivot, de certo modo, é o inverso de pivot. Primeiro vamos descrever o operador PIVOT e depois UNPIVOT.

**PIVOT de dados** é a técnica que agrupa e agrega dados, transportando da forma de **linhas** para um estado de **colunas**. Em todas queries que utilizam PIVOT os seguintes elementos são necessários:

* O que você quer ver em **linhas**? Esse elemento é conhecido como *on rows*, ou *grouping elements*
* O que você quer ver em **colunas**? Esse elemento é conhecido como *on cols*, ou *spreading element*
* O que você quer ver na **intersecção** no valor distinto de cada linha e coluna? Esse elemento é conhecido como *data*, ou *aggregation element.*

Suponhamos que você deseje ver como resultado de uma query **nas linhas** (*on rows*) todos IDs distintos para clientes; **nas colunas** (*on cols*)todos IDs distintos dos entregadores; e na intersecção dessas linhas e colunas a **soma** do valor de frete (*data, aggregation*). Com T-SQL e o operador PIVOT é possível realizar essa tarefa. A forma recomendável para montar uma query de PIVOT é a seguinte:

WITH PivotData AS

(

SELECT

custid, --clientes

shipperid, --entregadores

freight --frete

FROM Sales.Order

) AS SourceTable

SELECT

select list

FROM PivotData

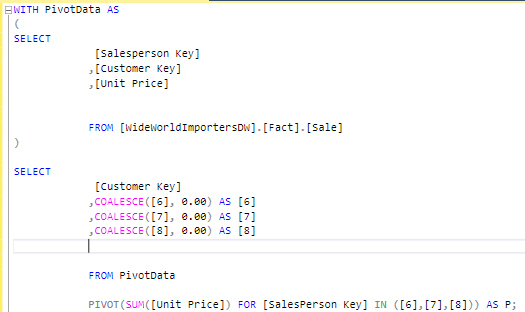
PIVOT( SUM (freight) --no caso frete

FOR < spreading column > IN (< distinct spreading values >) ) AS P;

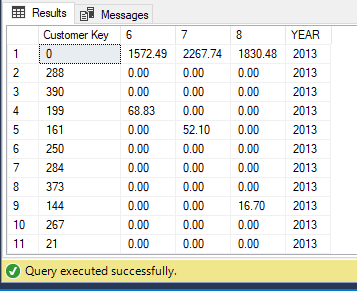
A forma geral recomendada é devido aos seguintes pontos:

* Você define uma CTE (no caso acima chamada PivotData) que retorna 3 elementos que estão envolvidos na operação de PIVOT, no caso são **clientes**, **entregadores**, **frete**, todos da tabela **Sales.Order**
* Você executa a query externa contra a CTE e aplica o operador **PIVOT** a essa CTE. O operador PIVOT retorna uma tabela como resultado. Você precisa atribuir um alias a essa tabela, no caso P.
* A especificação para o operador PIVOT começa por indicar a função de agregação aplicada ao elemento agregador (qual o dado, valor, numero a ser pivoteado), nesse caso seria **SUM(freight)**
* Então você especifica o parâmetro da cláusula **FOR**, seguido por aquilo que você quer ver em **colunas**
* Então você especifica na cláusula **IN** seguido da lista de valores distintos que aparecem no elemento de colunas, separados por vírgulas. O que eram valores no elemento de colunas (shipperID), se tornam nomes de colunas. Portanto, os itens na lista devem ser expressos como identificadores de colunas. Importante lembrar que, se um identificador de coluna é irregular, ele tem de ser delimitado. Como os shipperID são inteiros, devem ser expressos entre colchetes: [1], [2], [3] etc.

Seguindo a recomendação de sintaxe indicada acima, a query que pivoteamos ficaria da seguinte forma:



A query produz o seguinte resultado:



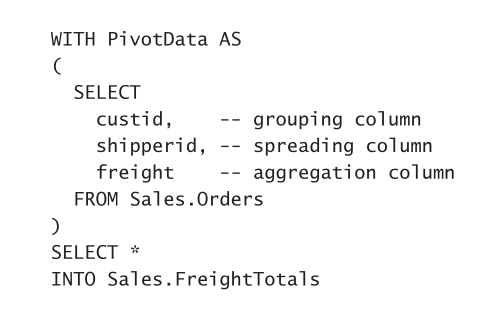
O operador PIVOT encontra por eliminação o elemento de agrupamento; na query acima indicamos o elemento de colunas (SalespersonKey) e o elemento de dados (o que recebe a agregação, no caso Unit Price), desta forma, o PIVOT entende que o elemento de agrupamento é aquele que sobrou da query. Por isso, indicamos sempre montar uma CTE para PIVOT com os três elementos apenas.

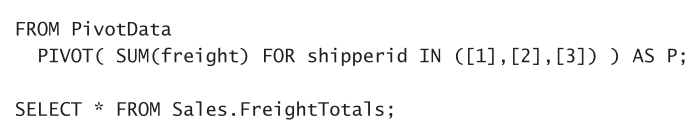
**Algumas limitações do operador PIVOT:**

* A agregação e o elemento de colunas não podem ser resultados diretos das expressões; ao invés disso, esses elementos devem ser nomes de colunas da tabela fonte. Podemos, entretanto, utilizar expressões na tabela fonte, atribuir aliases para a expressões e usar esses alias no operador PIVOT
* O operador COUNT(\*) não pode ser utilizado como a função agregadora do PIVOT. Se precisar de uma contagem, precisamos de uma alternativa; a mais simples é criar uma coluna com valor 1 e na função agregadora do PIVOT, realizar o SUM() dessa coluna que criamos
* É possível utilizar somente UMA função agregadora no PIVOT
* A cláusula IN aceita uma lista de valores como *spreading values*, ou seja, aquilo que você quer transformar em colunas; é preciso saber quais os valores únicos que serão passados como parâmetro para essa cláusula, podemos utilizar T-SQL dinâmico para construir e executar uma query dinâmica nesta cláusula

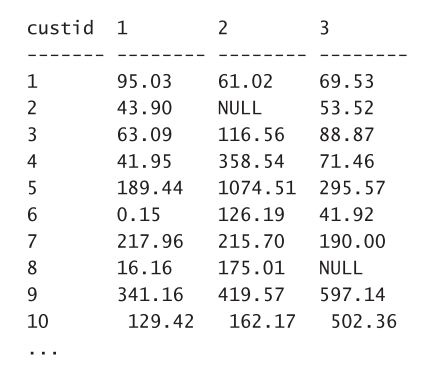
**Despivoteando (Unpivoting) Dados –** UNPIVOT pode ser considerado o contrário de PIVOT; UNPIVOT é rotacionar o dado de um estado de colunas para um estado de linhas. Assim como PIVOT, UNPIVOT é utilizado como um operador junto da cláusula FROM. O operador atua na tabela de input, à esquerda do operador. O resultado é uma tabela que pode ser utilizada como input para outros operadores de tabela.

Exemplos abaixo:





A query acima resulta nisto aqui:



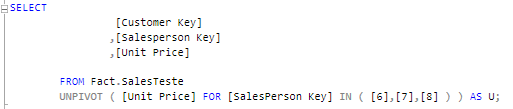
O resultado apresenta uma linha para cada cliente, uma coluna para cada fornecedor e o valor de intersecção de linha e coluna representa a soma de frete. A tarefa de unpivotear o resultado é retornar uma linha para cada cliente e entregador, colocando cliente em uma coluna e entregador em outra e o valor de frete um uma terceira coluna.

Unpivotear sempre pega um conjunto de colunas fonte e rotaciona para várias linhas, gerando duas colunas como objetivo: uma que guarda os **valores** fonte e outra que segura os **nomes** das colunas. No nosso primeiro exemplo o nome da primeira coluna nós já sabemos, os nomes das outras colunas [6], [7], [8] ainda precisa ser decidido.

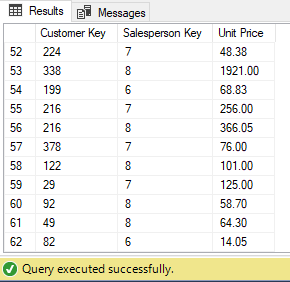
Em **TODA OPERAÇÃO DE UNPIVOTING** nós temos sempre 3 elementos envolvidos:

* O nome que você quer atribuir para as colunas de valor (no nosso caso Unit Price)
* O nome que você quer atribuir para as colunas de nomes (no nosso caso SalesPersonKey)
* O conjunto de colas fonte que você tá unpivoteando (no nosso caso [6],[7],[8])

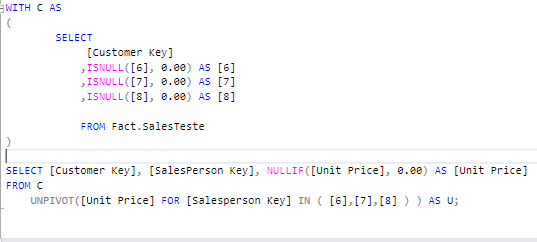
Após identificar esses três elementos, usamos a seguinte sintaxe para executarmos a tarefa de unpivotear:



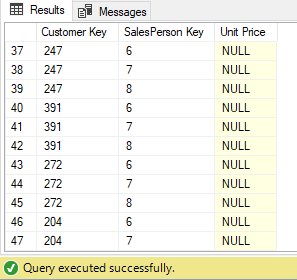
A query acima retorna o seguinte resultado:



Além de rotacionar os dados, o operador UNPIVOT filtra as linhas com valor NULL na coluna de valores, no nosso caso [Unit Price]. Se você quiser retornar linhas com valor NULL, precisamos alterar a query e utilizar as funções ISNULL/COALESCE para trazer os valores NULL. A query ficaria assim:



Trazendo como resultados linhas NULL na origem:



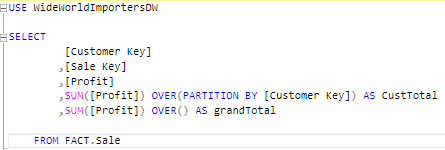
* **Usando funções de janela –** Assim como funções de grupo, funções de janela permitem que façamos analises de dados. A diferença entre os dois tipos de funções é como você define o conjunto de linhas para a função trabalhar. Com funções de grupo você usa queries agrupadas para organizar as linhas em grupos e então as funções de grupo são aplicadas a cada grupo.

Com funções de janela, você define o conjunto de linhas por função – e então retorna um valor de resultado por linha e função. Você define o conjunto de linhas para cada função trabalhar usando uma cláusula chamada **OVER**.

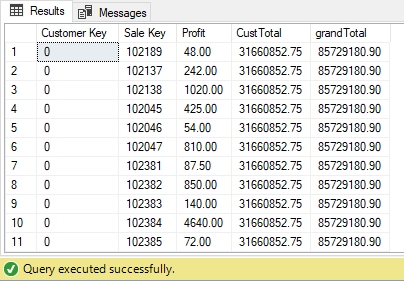
Essa seção cobre três tipos de funções de janela: **aggregate**, **rank** e **offset**.

**Função de janela – AGREGAÇÃO –** A função de janela de agregação são as mesmas que já conhecemos (SUM, AVG, MAX, MIN, COUNT), exceto que funções de janela são aplicadas a **janelas** de linhas, ou seja, a intervalo específicos de linhas, definidas pela cláusula OVER.

Aqui um exemplo de uma query olhando para uma view retornando, para cada pedido, o customerID, orderID e o valor do pedido; usando funções de janela, a query retorna também, o valor total e o total para todos os clientes:

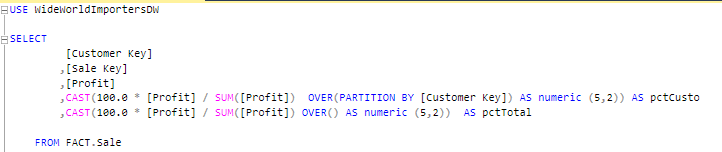


A query acima traz o seguinte resultado:

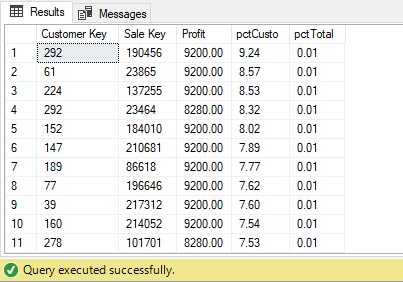


O grandTotal é o mesmo para todas as linhas; o total por cliente é o mesmo para todas as linhas com mesmo customer key.

É possível misturar elementos e agregações por janela na mesma expressão. Por exemplo a query abaixo computa, para cada pedido o percentual de cada pedido em relação ao total por clientes, e também em relação ao percentual total:



A query traz o seguinte resultado:

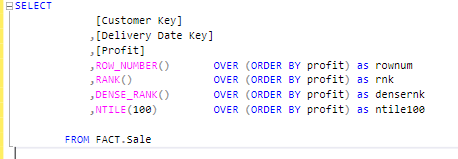


Funções de janela de agregação permitem delimitar o resultado definindo o *window frame*. A cláusula window frame requer uma ordem dentro da função de janela. No window frame você indica a **UNIDADE** (linhas ou range) e a **EXTENSÃO** do window frame (os delimitadores). Com ROWS como unidade do window frame, você pode indicar delimitadores de três tipos:

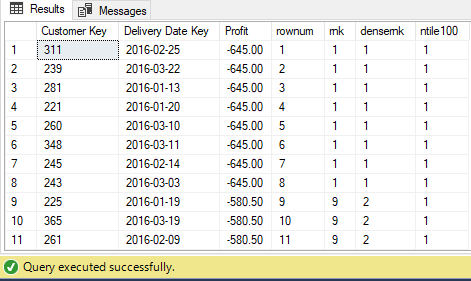
* **UNBOUNDED PRECEDING ou FOLLOWING** – Significando o começo ou o final da partição, respectivamente
* **CURRENT ROW**, representando, obviamente a primeira linha
* <n> **ROWS PRECEDING** ou **FOLLOWING** – Significando n numero de linhas antes ou depois da linha atual, respectivamente

**Funções de janela – RANKING –** Com as funções de janela de ranking, você consegue ranquear linhas dentro de uma partição baseado em uma ordenação específica. Como as outras funções de janela, se você não indicar uma cláusula de partição, o resultado todo é identificado como uma única partição.

A seguinte query demonstra a utilização dessas funções:



A query acima produz o seguinte resultado:



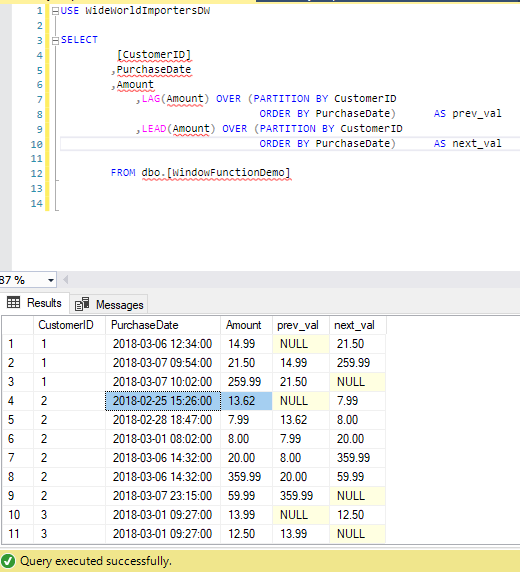
A função **ROW\_NUMBER** computa o incremento de inteiros únicos começando do 1, dentro da janela definida pela função de janela. Como na query de exemplo não temos uma partição (PARTITION BY) indicada, a função considera a query toda como um único conjunto de dados.

A função **RANK** e **DENSE\_RANK** se diferenciam de ROW\_NUMBER ao passo que elas atribuem o mesmo valor de ranking para todas as linhas que partilham do mesmo valor de ordenação. A função RANK retorna o número de linhas na partição que possuem valor menor que o atual, mais 1. Por exemplo, considere as linhas que trazem resultado que tem um valor de ordenação igual a 45.00. Nove linhas tem o valor de ordenação menor do que 45; portanto, essas linhas possuem rank 10 (9 + 1)

A função **DENSE\_RANK** retorna o numero de valores de ordenação distintos que são menores do que o atual mais 1. Por exemplo, as mesmas linhas que possuem rank 10, possuem o rank\_dense de 9. Isso por que essas linhas tem um valor de ordenação de 45.00, e existem 8 valores de ordenação distintos menor que 45.00. Rank considera as linhas e dense\_rank considera as linhas distintas.

Com a função **NTILE** você consegue organizar as linhas dentro de uma partição em um numero de compartimentos de igual tamanho. Na query de exemplo, fizemos N / 100, onde N é o numero total de linhas da query. Dividimos o result set em 100 partes iguais; caso a divisão não possua resto 0, as linhas excedentes são distribuídas entre os compartimentos.

**Função de janela – OFFSET –** Funções do tipo OFFSET retornam um único elemento de uma linha dado um deslocamento da linha atual dentro da partição, ou na forma de primeira/última linha no window frame. Um exemplo de query utilizando a função **LAG e LEAD:**



O resultado tem a ver com a linha imediatamente anterior ou posterior da linha atual.

Além de LAG e LEAD, as funções de janela offset oferecem mais dois tipos principais: **FIRST\_VALUE e LAST\_VALUE.**

Mais exemplos de **FUNÇÕES DE JANELA**, utilizando e-mail enviado pelo Julio em 27-01-2020, tomando a seguinte query/resultado como amostra:

SELECT

Pais

,Filial

,Mes

,TotalVenda

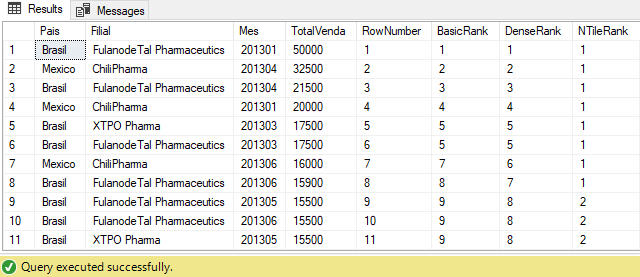
,ROW\_NUMBER() OVER (ORDER BY TotalVenda DESC) AS RowNumber

,RANK() OVER (ORDER BY TotalVenda DESC) AS BasicRank

,DENSE\_RANK() OVER (ORDER BY TotalVenda DESC) AS DenseRank

,NTILE(3) OVER (ORDER BY TotalVenda DESC) AS NTileRank

FROM dbo.TesteWindowFunction



A função **ROW\_NUMBER()** retorna o número de registros ordenados pelo campo TotalVenda. Como não utilizamos a cláusula PARTITION BY, o SQL Server entende todo o result set como uma única partição, portanto os dados retornados nessa coluna têm o intervalo de 1 a 24.

A função **RANK()** olha para os registros que têm valores iguais no campo TotalVenda (passado como parâmetro dentro de OVER() ) e os classifica analisando apenas os valores distintos. No exemplo acima, existem 4 linhas com TotalVenda 15500, todas ranqueadas como 9º, o próximo registro nessa coluna seria o 13º.

A função **DENSE\_RANK()** tem basicamente o mesmo comportamento, mas o próximo registro distinto recebe um número sequencial. Se levarmos em conta o exemplo acima, após a 9ª posição, teríamos a 10ª e não a 13ª.

A função **NTILE()** recebe um parâmetro e classifica todos os registros de acordo com o número informado como parâmetro. Se temos um result set com 100 linhas e passamos o NTILE(5), ele vai dividir as 100 linhas em 5 grupos de registros com 20 linhas cada. Na query de amostra, temos 24 linhas e passamos o NTILE(3), ou seja, a função dividiu o nosso result set (24), em 3 grupos iguais de 8 registros cada.

Agora utilizaremos a cláusula **PARTITION BY** na nossa query e analisaremos o resultado para cada função aplicada:

SELECT

Pais

,Filial

,Mes

,TotalVenda

,ROW\_NUMBER() OVER (

PARTITION BY Pais

ORDER BY TotalVenda DESC) AS RowNumber

,RANK() OVER (

PARTITION BY Pais

ORDER BY TotalVenda DESC) AS BasicRank

,DENSE\_RANK() OVER (

PARTITION BY Pais

ORDER BY TotalVenda DESC) AS DenseRank

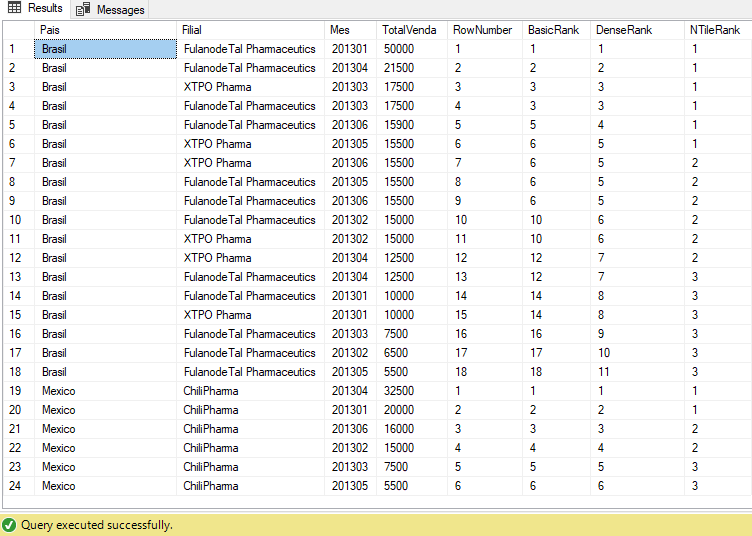
,NTILE(3) OVER (

PARTITION BY Pais

ORDER BY TotalVenda DESC) AS NTileRank

FROM dbo.TesteWindowFunction

Resultado:



Agora os rankings são reiniciados cada vez que um novo país é lido na consulta. A lógica se mantém a mesma descrita acima, mas os números são reiniciados quando um país novo é retornado da consulta.

**AGREGAÇÃO:**

SELECT DISTINCT

Pais

,Filial

,COUNT(TotalVenda) OVER (

PARTITION BY Pais, Filial) AS QtdVendasFilial

,SUM(TotalVenda) OVER(

PARTITION BY Pais, Filial) AS TotalVendasFilial

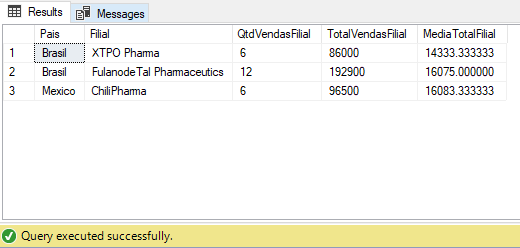
,AVG(TotalVenda) OVER(

PARTITION BY Pais, Filial) AS MediaTotalFilial

FROM dbo.TesteWindowFunction

ORDER BY Pais, Filial DESC;

Resultado:



As funções de agregação foram aplicadas para cada país e filial, conforme informado na cláusula **PARTITION BY**

**ANÁLISANDO:**

As funções **FIRST\_VALUE / LAST\_VALUE** retornam o primeiro e o último valor, respectivamente, de acordo com o particionamento e ordenação informados.

Exemplo de query:

SELECT

Pais

,Filial

,Mes

,FIRST\_VALUE(TotalVenda) OVER (

PARTITION BY Filial

ORDER BY TotalVenda DESC) AS MaiorVenda

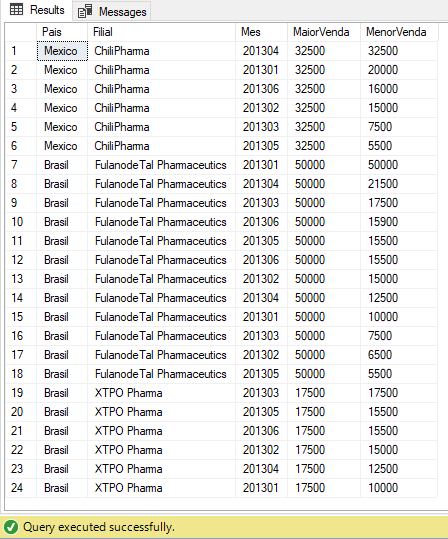
,LAST\_VALUE(TotalVenda) OVER (

PARTITION BY Filial

ORDER BY TotalVenda DESC) AS MenorVenda

FROM dbo.TesteWindowFunction

Resultado:



Aqui a cláusula PARTITION BY divide os dados por filial; aliado com a cláusula ORDER BY aplicada ao campo TotalVenda, conseguimos definir na coluna MaiorVenda, o maior valor de venda para cada mês e na coluna MenorVenda, o menor valor de venda para cada mês.

Ainda dentro de FIRST/LAST\_VALUE, podemos passar como parâmetros o intervalo de linhas em que as cláusulas PARTITION BY/ORDER BY serão aplicadas, com os parâmetros **RANGE** e **ROWS**. Quando não especificamos nada para RANGE/ROWS, o padrão para o parâmetro é **RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW**, ou seja, a função é aplicada entre o registro anterior (UNBOUNDED PRECEDING) e o registro atual (CURRENT ROW) apenas

Exemplo de query parametrizada para trazer o maior e o menor valor para TotalVenda por filial e ordenado por TotalVenda:

SELECT

Pais

,Filial

,Mes

,TotalVenda

,FIRST\_VALUE(TotalVenda) OVER (

PARTITION BY Filial

ORDER BY TotalVenda DESC) AS MaiorVenda

,LAST\_VALUE(TotalVenda) OVER (

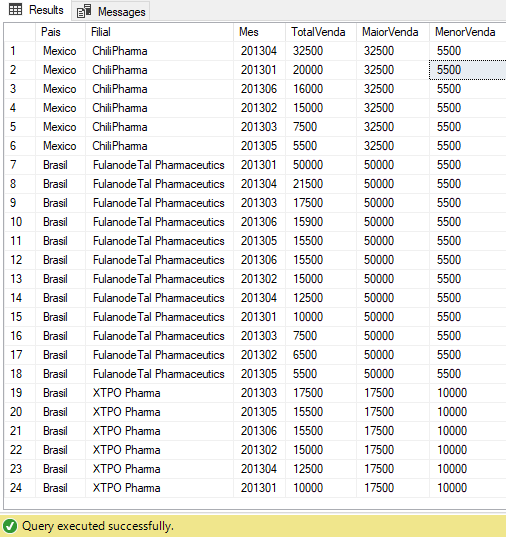
PARTITION BY Filial

ORDER BY TotalVenda DESC

ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING) AS MenorVenda

FROM dbo.TesteWindowFunction

Resultado:



**Funções LAG/LEAD** retornam o valor acima ou abaixo da linha atual, de acordo com o valor passado como parâmetro. Na query abaixo informamos 1 como valor para LAG/LEAD, portanto as funções retornarão os valores **imediatamente** acima ou abaixo da linha atual:

SELECT

Pais

,Filial

,Mes

,TotalVenda

,LAG(TotalVenda,1) OVER (

PARTITION BY Filial

ORDER BY TotalVenda DESC) AS VendaAnterior

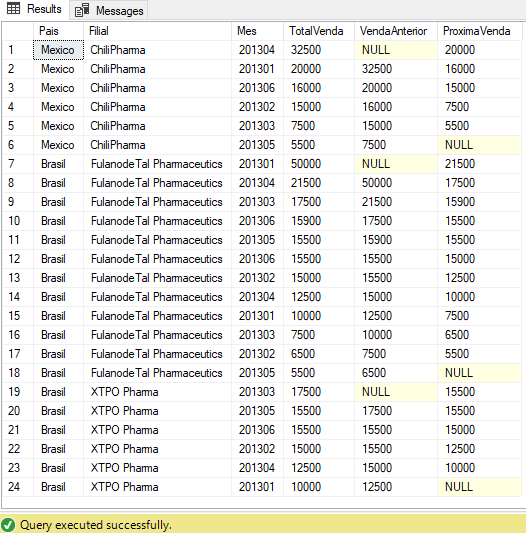
,LEAD(TotalVenda,1) OVER (

PARTITION BY Filial

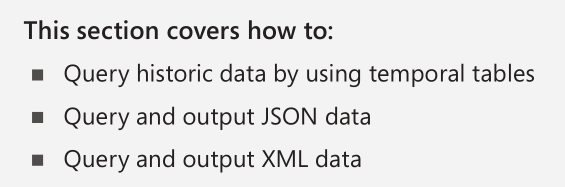
ORDER BY TotalVenda DESC) AS ProximaVenda

FROM dbo.TesteWindowFunction

Resultado:



* **Consultando dados temporais e dados não relacionais (XML, JSON)**



Tabelas temporais com versionamento controlado pelo sistema. Essas tabelas, que nós vamos chamar a partir de agora apenas de *tabelas temporais*, tem como finalidade restaurar estados do banco após alguma mudança, acidental ou não.

SQL oferece a alternativa de fazer uma tabela temporal usando a opção **SYSTEM\_VERSIONING** e conectando a uma tabela histórica. Quando você modifica dados, você está interagindo apenas com a tabela atual e o SQL Server, por debaixo dos panos, escreve estados históricos das linhas modificadas na tabela histórica.

As próximas páginas vão tratar de criar, modificar e consultar tabelas temporais.

**Criando tabelas temporais** – Você pode marcar uma tabela como temporal no momento da criação; em paralelo, você consegue criar com o SQL Server a tabela de histórico ou providenciar uma tabela de histórico já existente.

**Certos elementos que definem uma tabela temporal**:

* **Primary key**
* **Duas colunas DATETIME2** – Com a precisão da sua escolha para guardar o começo e o final da validade do período de cada linha. O período é expresso como um intervalo aberto-fechado, ou seja, o começo é inclusivo e o final é exclusivo.
* **A coluna de início** deve ser sempre marcada como GENERATED ALWAYS AS ROW START
* **A coluna de final** deve ser sempre marcada como GENERATED ALWAYS AS ROW END
* **A designação do par de colunas** que guardam a validade do período da linha com a cláusula PERIOD FOR SYSTEM\_TIME(start\_col, end\_col)
* **A opção de tabela** SYSTEM\_VERSIONING precisa estar setada como ON
* **Uma tabela histórica**, que o SQL Server consegue criar pra nós

Sintaxe básica para criação de uma tabela temporal:

CREATE TABLE dbo.Products

(

productid INT NOT NULL

CONSTRAINT PK\_dboProducts PRIMARY KEY(productid),

productname NVARCHAR(40) NOT NULL,

supplierid INT NOT NULL,

categoryid INT NOT NULL,

unitprice MONEY NOT NULL,

-- below are additions related to temporal table

validfrom DATETIME2(3)

GENERATED ALWAYS AS ROW START HIDDEN NOT NULL,

validto DATETIME2(3)

GENERATED ALWAYS AS ROW END HIDDEN NOT NULL,

PERIOD FOR SYSTEM\_TIME (validfrom, validto)

)

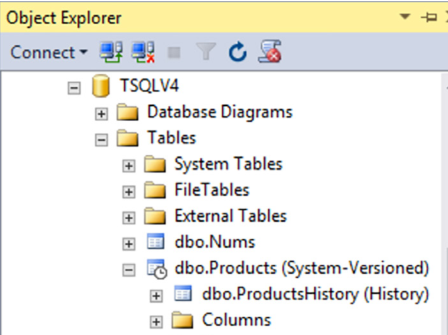
WITH ( SYSTEM\_VERSIONING = ON ( HISTORY\_TABLE = dbo.ProductsHistory ) );

A opção de HIDDEN NOT NULL faz com que as colunas validto, validfrom não aparecam em um SELECT \*; as colunas existem, mas só retornarão em uma consulta caso sejam explicitamente invocadas na lista de SELECT.

Em relação as tabelas históricas, se você não informar uma, o SQL cria uma de forma automática, seguindo a convenção e nomeando-a como MSSQL\_TemporalHistoriFor\_<object\_id>. Se especificarmos como no caso acima, o SQL cria a tabela com as seguintes características:

* Sem chave primária
* Índice clusterizado nas colunas <endcol, startcol> com comprensão de página
* As colunas de tempo não são marcadas com GENERATED ALWAYS AS ROW START/END ou HIDDEN
* Não existem designações de período para colunas com a cláusula PERIOD FOR SYSTEM\_TIME
* A tabela não é marcada com a opção SYSTEM\_VERSIONING

Conseguimos identificar uma tabela temporal no SSMS quando a tabela possui, dentro do Object Explorer, quando a tabela possui (System-Versioned) ao lado do seu nome:



Se precisarmos transformar uma tabela já existente em uma tabela temporal, podemos usar ALTER TABLE e passar as especificações mencionadas acima. A sintaxe para alterar uma tabela existente para uma tabela temporal é a seguinte:

BEGIN TRAN;

ALTER TABLE dbo.Products ADD

validfrom DATETIME2(3) GENERATED ALWAYS AS ROW START HIDDEN NOT NULL

CONSTRAINT DFT\_Products\_validfrom DEFAULT('19000101'),

validto DATETIME2(3) GENERATED ALWAYS AS ROW END HIDDEN NOT NULL

CONSTRAINT DFT\_Products\_validto DEFAULT('99991231 23:59:59.999'),

PERIOD FOR SYSTEM\_TIME (validfrom, validto);

ALTER TABLE dbo.Products

SET ( SYSTEM\_VERSIONING = ON ( HISTORY\_TABLE = dbo.ProductsHistory ) );

ALTER TABLE dbo.Products

DROP CONSTRAINT DFT\_Products\_validfrom, DFT\_Products\_validto;

COMMIT TRAN;

Para alterar uma tabela temporal também é possível utilizar ALTER TABLE e informarmos as alterações desejadas.

**Modificando dados em uma tabela temporal** segue a mesma forma de modificação em uma tabela normal. Algumas coisas para lembrar:

* SQL grava as mudanças de tempo dentro do fuso UTC
* Se várias mudanças forem aplicadas, o tempo de início da transação é considerado o tempo efetivo para *todas* alterações realizadas.

Quando você insere linhas na tabela atual, o SQL seta a startcol com o tempo do início da transação de INSERT. Se realizarmos uma consulta na tabela de histórico, ela estará vazia.

Agora, se realizarmos um DELETE em uma linha da tabela atual e consultarmos a tabela histórica, a transação estará presente no histórico; como o UPDATE é tratado como DELETE+INSERT, as operações de UPDATE também são registradas na tabela de histórico.

Quando modificamos dados em uma tabela temporal, o tempo inicial da transação é considerado o tempo para TODAS mudanças. Se varias mudanças foram aplicadas na mesma transação, essas mudanças terão o mesmo tempo de início.

**Consultando dados em uma tabela temporal** é possível consultarmos os dados em uma tabela temporal para acessarmos os estados históricos das linhas. O SQL Server facilita nossas vidas oferecendo a cláusula **FOR SYSTEM\_TIME**. Usando sub cláusulas diferentes, podemos passar um intervalo de tempo de validade para o SQL buscar as informações na tabela.

Uma sub cláusula comum de ser utilizada com o **FOR SYSTEM\_TIME** é o AS OF @dt, onde dt é uma variável DATETIME. Como o período de validade é um intervalo aberto, ou seja, você retornaria o intervalo de linhas onde @dt >= validfrom AND @dt < validto.

**Consultando e apresentando resultados com dados XML**

--------

1. Generally, when solving tasks with T-SQL, is it more efficient to use joins or subqueries?

Depende, existem casos em que o uso de JOIN é mais performático do que a subquery e vice-versa. Em uma consulta em que são feitas agregações e acessando os mesmos dados, o JOIN tem melhor performance que a subquery, pois na subquery SQL vai criar acessos separados para cada consulta e formar as agregações. Com JOIN o acesso aos dados é feito uma única vez e as agregações feitas a partir daí.

1. What is the difference between a self-contained subquery and a correlated one?

A subquery “auto-contida” é aquele caso em que a subquery roda de forma separada e independente da query externa que a contém; uma subquery correlacionada não roda de forma separada e independente, uma vez que ela referencia e utiliza colunas da query externa que a contém.

1. In what way is the APPLY operator different than joins and subqueries? Can you pro-

vide an example when it should be used?