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

  )

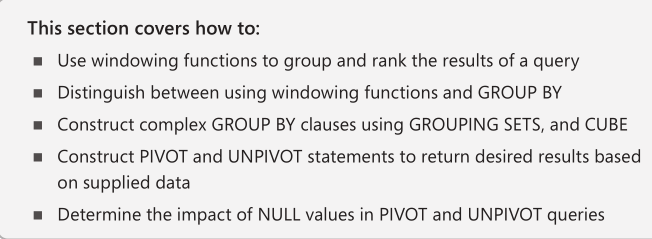
  SELECT empid, mgrid, firstname, lastname, distance

  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:

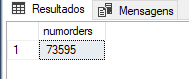
use

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

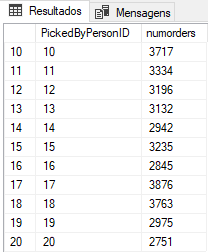
WideWorldImporters

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

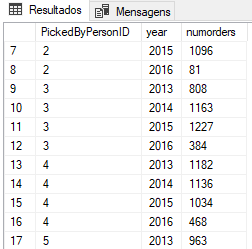
WideWorldImporters

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

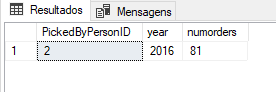
FROM Sales.Orders

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 )

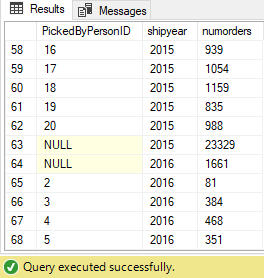
,( year(OrderDate) )

,( )

);

**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

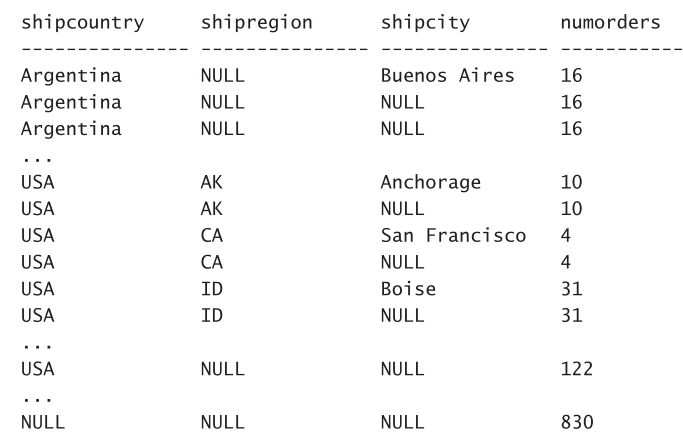
,shipcity

,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

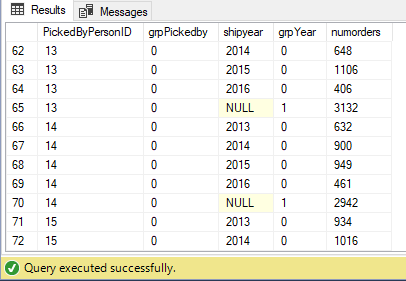
,COUNT(\*) AS numorders

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

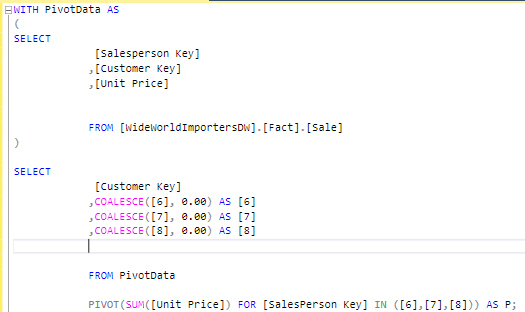
PIVOT( SUM (freight) --no caso frete

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

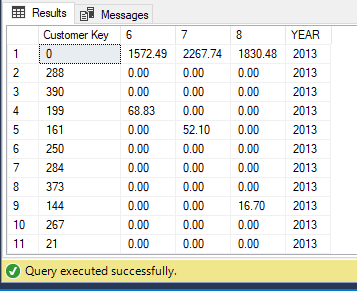
A forma geral recomendada é devido aos seguintes pontos:

* 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:



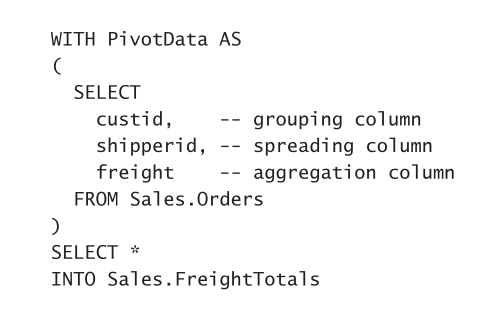
O operador PIVOT encontra por eliminação o elemento de agrupamento; na query acima indicamos o elemento de colunas (SalespersonKey) e o elemento de dados (o que recebe a agregação, no caso Unit Price), desta forma, o PIVOT entende que o elemento de agrupamento é aquele que sobrou da query. Por isso,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

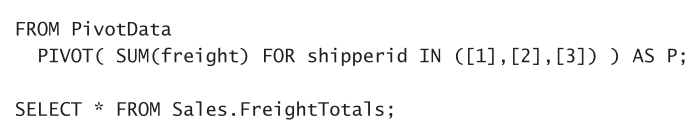
**Algumas limitações do operador PIVOT:**

* 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

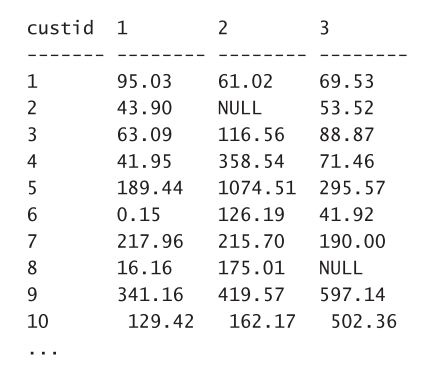
**Despivoteando (Unpivoting) Dados –** UNPIVOT pode ser considerado o contrário de PIVOT; UNPIVOT é rotacionar o dado de um estado de colunas para um estado de linhas. Assim como PIVOT, UNPIVOT é utilizado como um operador junto da cláusula FROM. O operador atua na tabela de input, à esquerda do operador.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

Exemplos abaixo:





A query acima resulta nisto aqui:



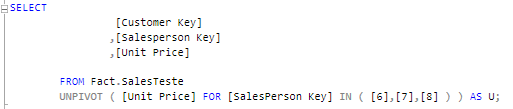
O resultado apresenta uma linha para cada cliente, uma coluna para cada fornecedor e o valor de intersecção de linha e coluna representa a soma de frete. A tarefa de unpivotear o resultado é retornar uma linha para cada cliente e entregador, colocando cliente em uma coluna e entregador em outra e o valor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

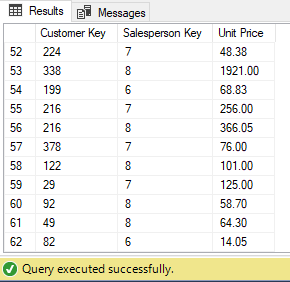
Em **TODA OPERAÇÃO DE UNPIVOTING** nós temos sempre 3 elementos envolvidos:

* 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])

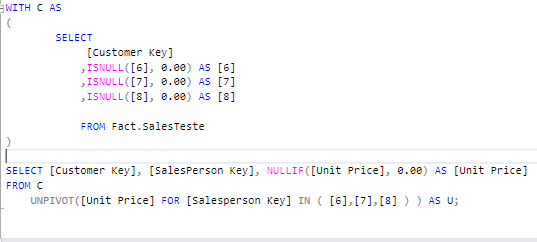
Após identificar esses três elementos, usamos a seguinte sintaxe para executarmos a tarefa de unpivotear:



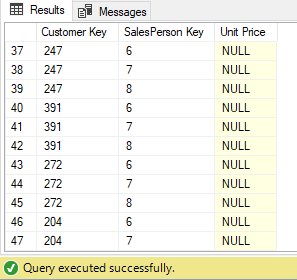
A query acima retorna o seguinte resultado:



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:



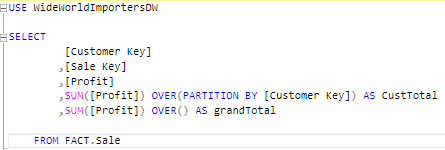
* **Usando funções de janela –** Assim como funções de grupo, funções de janela permitem que façamos analises de dados. A diferença entre os dois tipos de funções é como você define o conjunto de linhas para a função trabalhar. Com funções de grupo você usa queries agrupadas para organizar as linhas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

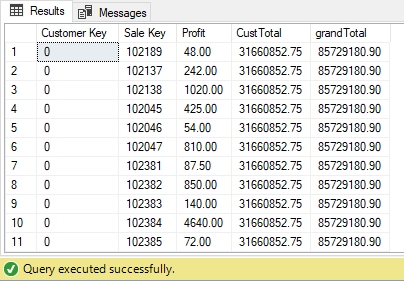
Essa seção cobre três tipos de funções de janela: **aggregate**, **rank** e **offset**.

**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:

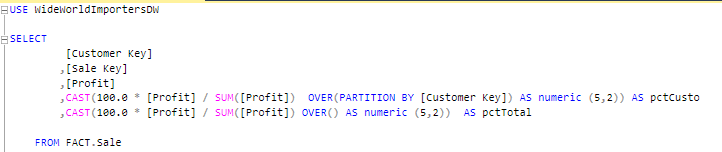


A query acima traz o seguinte resultado:

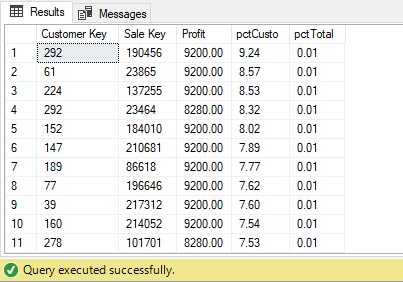


O grandTotal é o mesmo para todas as linhas; o total por cliente é o mesmo para todas as linhas com mesmo customer key.

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:

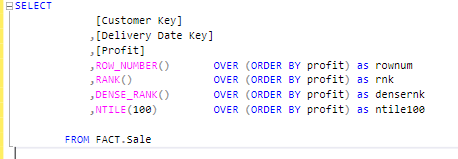


Funções de janela de agregação permitem delimitar o resultado definindo o *window frame*. A cláusula window frame requer uma ordem dentro da função de janela. No window frame você indica a **UNIDADE** (linhas ou range) e a **EXTENSÃO** do window frame (os delimitadores). Com ROWS como unidade do window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

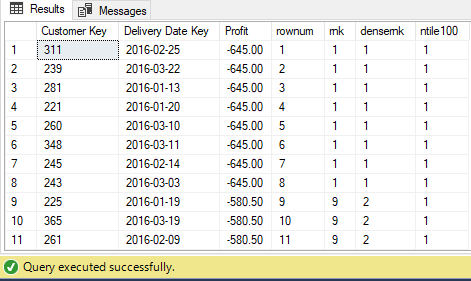
* **UNBOUNDED PRECEDING ou FOLLOWING** – Significando o começo ou o final da partição, respectivamente
* **CURRENT ROW**, representando, obviamente a primeira linha
* <n> **ROWS PRECEDING** ou **FOLLOWING** – Significando n numero de linhas antes ou depois da linha atual, respectivamente

**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:



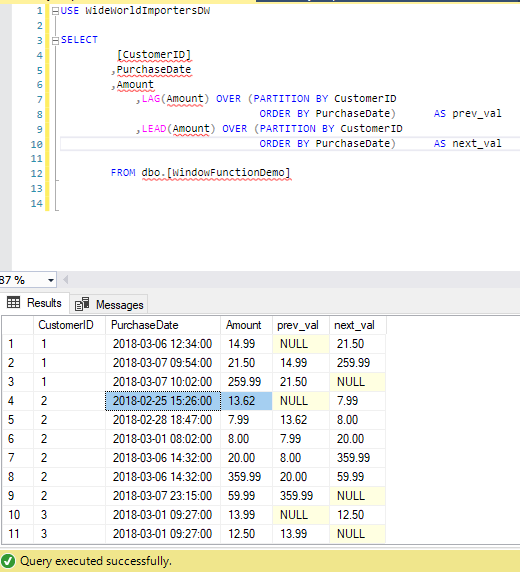
A função **ROW\_NUMBER** computa o incremento de inteiros únicos começando do 1, dentro da janela definida pela função de janela. Como na query de exemplo não temos uma partição (PARTITION BY) indicada, a função considera a query toda como um único conjunto de dados.

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

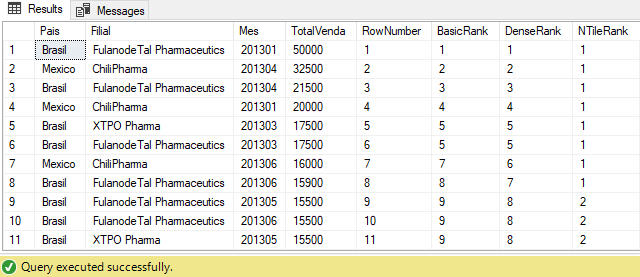
,ROW\_NUMBER() OVER (ORDER BY TotalVenda DESC) AS RowNumber

,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

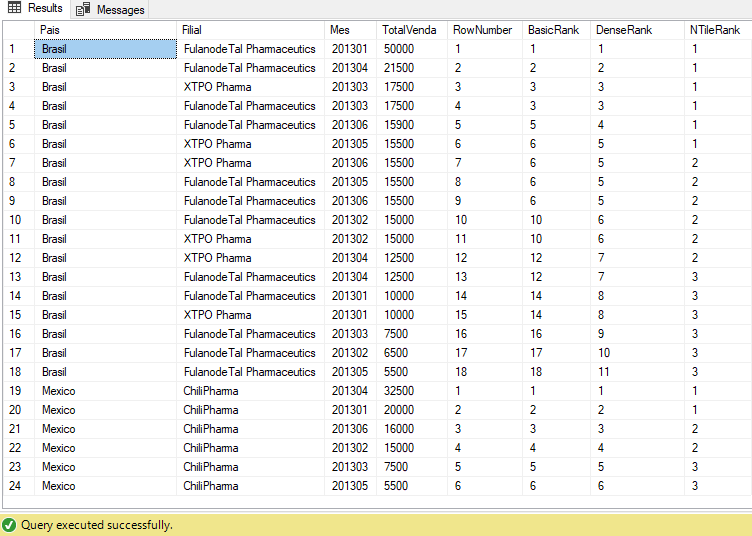
,NTILE(3) OVER (

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

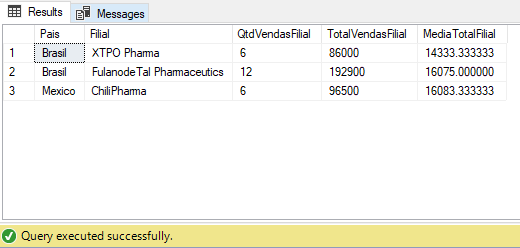
,AVG(TotalVenda) OVER(

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

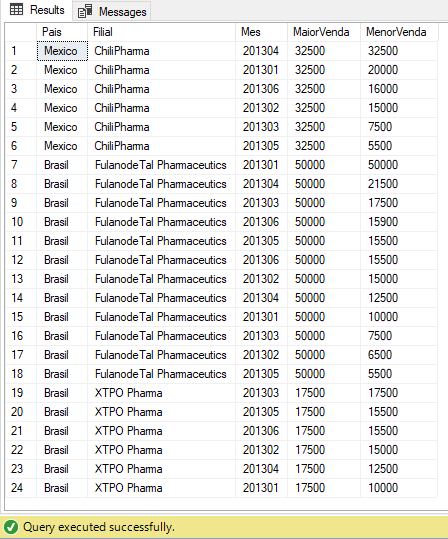
,LAST\_VALUE(TotalVenda) OVER (

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 (

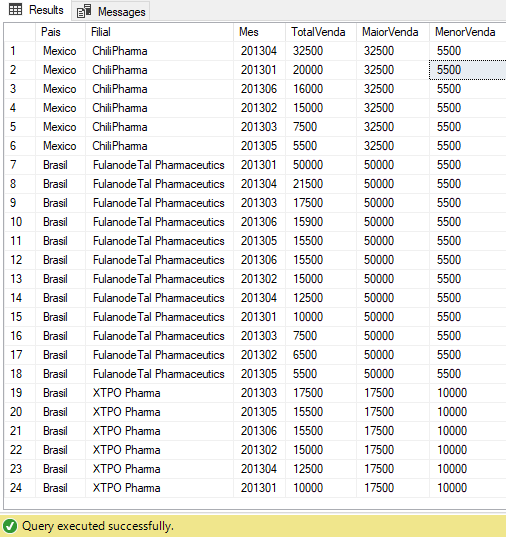
PARTITION BY Filial

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

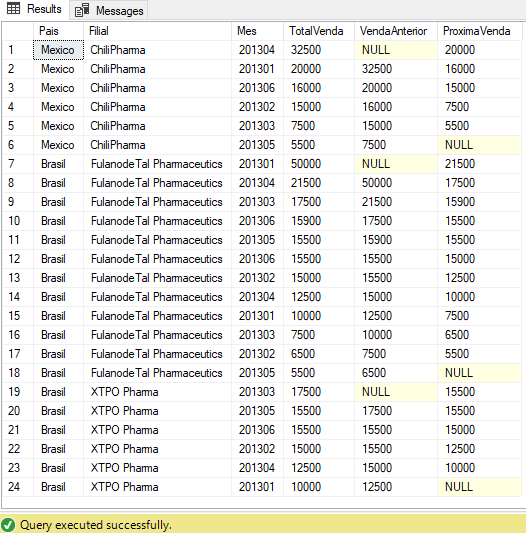
,LEAD(TotalVenda,1) OVER (

PARTITION BY Filial

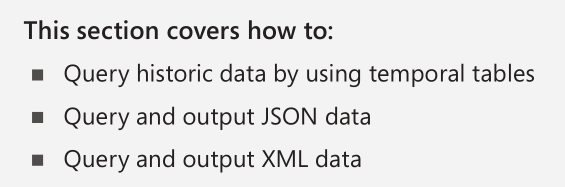
ORDER BY TotalVenda DESC) AS ProximaVenda

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)

)

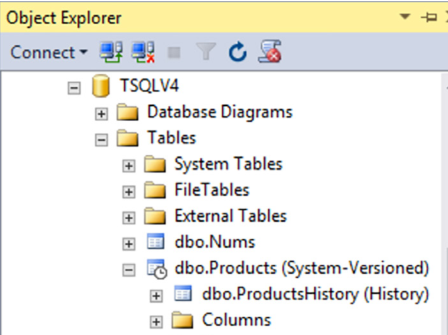
WITH ( SYSTEM\_VERSIONING = ON ( HISTORY\_TABLE = dbo.ProductsHistory ) );

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.

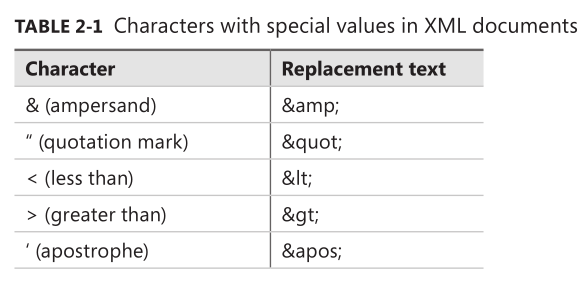
**Consultando e apresentando resultados com dados XML**

Nesta seção aprenderemos, de forma introdutória, como trabalhar com arquivos XML dentro do SQL Server e quais são os diferentes tipos de consultas que podemos criar utilizando XML. No final desta seção, veremos como misturamos XML e tabelas relacionais usando a função **OPENXML rowset**.

XML utiliza tags dentro das partes de um documento. Essas partes são chamadas de elementos; cada tag de início de um elemento tem uma tag de final para esse elemento.

Documentos XML são ordenados. Não no sentido de ordenação do valor de um elemento; em outras palavras a **posição** de um elemento importa.

XML é case-sensitive, importante lembrar disso. Alguns caracteres especiais como **<** são lidos como abertura de tag, se for necessário incorporar esse caractere (e outros) no corpo do documento, precisamos de uma formatação especial. Tabela abaixo serve de referência:



Documentos XML podem ter uma “introdução” no topo do documento, algo como *<?xml version=”1.0” encoding=”ISO-8859-15”?>*.

Além de documentos XML, podemos ter fragmentos XML. A única diferença entre um fragmento e um documento é que o documento tem um único *root node*. No exemplo abaixo, temos um único root node (CustomerOrders), se deletarmos esse root node, teremos um fragmento:

<CustomersOrders>

<Customer custid="1" companyname="Customer NRZBB">

<Order orderid="10692" orderdate="2007-10-03T00:00:00" />

<Order orderid="10702" orderdate="2007-10-13T00:00:00" />

<Order orderid="10952" orderdate="2008-03-16T00:00:00" />

</Customer>

<Customer custid="2" companyname="Customer MLTDN">

<Order orderid="10308" orderdate="2006-09-18T00:00:00" />

<Order orderid="10926" orderdate="2008-03-04T00:00:00" />

</Customer>

</CustomersOrders>

Elementos tem os seus próprios atributos, e cada atributo tem o seu próprio nome e valores, valores esses que sempre estão entre aspas “”.

**Produzindo e usando XML nas queries**

A primeira opção para gerar um XML a partir das queries é a cláusula **RAW**, onde, como o nome sugere, é retorna um resultado cru, sem muita formatação e divisão entre elementos e atributos. Fica tudo meio amontoado:

<row Pais="Brasil" Filial="XTPO Pharma" Mes="201301" TotalVenda="10000" />

<row Pais="Brasil" Filial="XTPO Pharma" Mes="201302" TotalVenda="15000" />

<row Pais="Brasil" Filial="XTPO Pharma" Mes="201303" TotalVenda="17500" />

<row Pais="Brasil" Filial="XTPO Pharma" Mes="201304" TotalVenda="12500" />

<row Pais="Brasil" Filial="XTPO Pharma" Mes="201305" TotalVenda="15500" />

<row Pais="Brasil" Filial="XTPO Pharma" Mes="201306" TotalVenda="15500" />

<row Pais="Brasil" Filial="FulanodeTal Pharmaceutics" Mes="201301" TotalVenda="10000" />

<row Pais="Brasil" Filial="FulanodeTal Pharmaceutics" Mes="201302" TotalVenda="15000" />

<row Pais="Brasil" Filial="FulanodeTal Pharmaceutics" Mes="201303" TotalVenda="17500" />

<row Pais="Brasil" Filial="FulanodeTal Pharmaceutics" Mes="201304" TotalVenda="12500" />

<row Pais="Brasil" Filial="FulanodeTal Pharmaceutics" Mes="201305" TotalVenda="15500" />

<row Pais="Brasil" Filial="FulanodeTal Pharmaceutics" Mes="201306" TotalVenda="15500" />

<row Pais="Mexico" Filial="ChiliPharma" Mes="201301" TotalVenda="20000" />

<row Pais="Mexico" Filial="ChiliPharma" Mes="201302" TotalVenda="15000" />

<row Pais="Mexico" Filial="ChiliPharma" Mes="201303" TotalVenda="7500" />

<row Pais="Mexico" Filial="ChiliPharma" Mes="201304" TotalVenda="32500" />

<row Pais="Mexico" Filial="ChiliPharma" Mes="201305" TotalVenda="5500" />

<row Pais="Mexico" Filial="ChiliPharma" Mes="201306" TotalVenda="16000" />

<row Pais="Brasil" Filial="FulanodeTal Pharmaceutics" Mes="201301" TotalVenda="50000" />

<row Pais="Brasil" Filial="FulanodeTal Pharmaceutics" Mes="201302" TotalVenda="6500" />

<row Pais="Brasil" Filial="FulanodeTal Pharmaceutics" Mes="201303" TotalVenda="7500" />

<row Pais="Brasil" Filial="FulanodeTal Pharmaceutics" Mes="201304" TotalVenda="21500" />

<row Pais="Brasil" Filial="FulanodeTal Pharmaceutics" Mes="201305" TotalVenda="5500" />

<row Pais="Brasil" Filial="FulanodeTal Pharmaceutics" Mes="201306" TotalVenda="15900" />

A outra opção é a cláusula **AUTO**, neste caso o documento é mais organizado e é um documento de fato e não um fragmento como o exemplo acima:

<dbo.TesteWindowFunction Pais="Brasil" Filial="XTPO Pharma" Mes="201301" TotalVenda="10000" />

<dbo.TesteWindowFunction Pais="Brasil" Filial="XTPO Pharma" Mes="201302" TotalVenda="15000" />

<dbo.TesteWindowFunction Pais="Brasil" Filial="XTPO Pharma" Mes="201303" TotalVenda="17500" />

<dbo.TesteWindowFunction Pais="Brasil" Filial="XTPO Pharma" Mes="201304" TotalVenda="12500" />

<dbo.TesteWindowFunction Pais="Brasil" Filial="XTPO Pharma" Mes="201305" TotalVenda="15500" />

<dbo.TesteWindowFunction Pais="Brasil" Filial="XTPO Pharma" Mes="201306" TotalVenda="15500" />

<dbo.TesteWindowFunction Pais="Brasil" Filial="FulanodeTal Pharmaceutics" Mes="201301" TotalVenda="10000" />

<dbo.TesteWindowFunction Pais="Brasil" Filial="FulanodeTal Pharmaceutics" Mes="201302" TotalVenda="15000" />

<dbo.TesteWindowFunction Pais="Brasil" Filial="FulanodeTal Pharmaceutics" Mes="201303" TotalVenda="17500" />

<dbo.TesteWindowFunction Pais="Brasil" Filial="FulanodeTal Pharmaceutics" Mes="201304" TotalVenda="12500" />

<dbo.TesteWindowFunction Pais="Brasil" Filial="FulanodeTal Pharmaceutics" Mes="201305" TotalVenda="15500" />

<dbo.TesteWindowFunction Pais="Brasil" Filial="FulanodeTal Pharmaceutics" Mes="201306" TotalVenda="15500" />

<dbo.TesteWindowFunction Pais="Mexico" Filial="ChiliPharma" Mes="201301" TotalVenda="20000" />

<dbo.TesteWindowFunction Pais="Mexico" Filial="ChiliPharma" Mes="201302" TotalVenda="15000" />

<dbo.TesteWindowFunction Pais="Mexico" Filial="ChiliPharma" Mes="201303" TotalVenda="7500" />

<dbo.TesteWindowFunction Pais="Mexico" Filial="ChiliPharma" Mes="201304" TotalVenda="32500" />

<dbo.TesteWindowFunction Pais="Mexico" Filial="ChiliPharma" Mes="201305" TotalVenda="5500" />

<dbo.TesteWindowFunction Pais="Mexico" Filial="ChiliPharma" Mes="201306" TotalVenda="16000" />

<dbo.TesteWindowFunction Pais="Brasil" Filial="FulanodeTal Pharmaceutics" Mes="201301" TotalVenda="50000" />

<dbo.TesteWindowFunction Pais="Brasil" Filial="FulanodeTal Pharmaceutics" Mes="201302" TotalVenda="6500" />

<dbo.TesteWindowFunction Pais="Brasil" Filial="FulanodeTal Pharmaceutics" Mes="201303" TotalVenda="7500" />

<dbo.TesteWindowFunction Pais="Brasil" Filial="FulanodeTal Pharmaceutics" Mes="201304" TotalVenda="21500" />

<dbo.TesteWindowFunction Pais="Brasil" Filial="FulanodeTal Pharmaceutics" Mes="201305" TotalVenda="5500" />

<dbo.TesteWindowFunction Pais="Brasil" Filial="FulanodeTal Pharmaceutics" Mes="201306" TotalVenda="15900" />

Podemos trabalhar com as cláusulas WITH NAMESPACES, ELEMENTS etc. dentro do arquivo XML.

Além de transformarmos queries em arquivos XML, também podemos fazer o caminho oposto: transformar um arquivo XML em tabelas relacionais com a função **OPENXML** rowset.

A função OPENXML oferece uma apresentação em linhas na memória; o método que o XML é transformado em tabelas é utilizando uma apresentação em **DOM (Document Object Model).** Antes de adequar o DOM, é necessário prepará-lo. Para preparar a apresentação do XML em DOM, é necessário executar a stored procedure **sys.sp\_xml\_preparedocument**. Depois disso é necessário remover a apresentação DOM utilizando a stored\_procedure **sys.sp\_xml\_removedocument.**

A função **OPENXML** usa os seguintes parâmetros:

* Um documento XML DOM manuseável, retornado pela sp\_xml\_preparedocument
* Uma expressão XPath para localizar os root nodes que você quer mapear e retornar
* Uma descrição do conjunto de linas retornadas
* Mapeamento entre os nodes do XML e o conjunto de linhas retornadas

**Consultando dados XML com XQuery**

XQuery é a linguagem padrão para consultar dados XML. Faz as mesmas coisas do que as expressões XPath, mas oferece várias outras possibilidades.

O poder do XQuery está nas expressões FLWOR, acrônimo para **FOR**, **LET**, **WITH**, **ORDER BY** e **RETURN**. Esse tipo de expressão permite a iteração sobre nodes do documento XML.

Exemplo de uma expressão FLWOR:

DECLARE @x AS XML = N'

<CustomersOrders>

<Customer custid="1">

<!-- Comment 111 -->

<companyname>Customer NRZBB</companyname>

<Order orderid="10692">

<orderdate>2015-10-03T00:00:00</orderdate>

</Order>

<Order orderid="10702">

<orderdate>2015-10-13T00:00:00</orderdate>

</Order>

<Order orderid="10952">

<orderdate>2016-03-16T00:00:00</orderdate>

</Order>

</Customer>

<Customer custid="2">

<!-- Comment 222 -->

<companyname>Customer MLTDN</companyname>

<Order orderid="10308">

<orderdate>2014-09-18T00:00:00</orderdate>

</Order>

<Order orderid="10952">

<orderdate>2016-03-04T00:00:00</orderdate>

</Order>

200 Chapter 2 Query data with advanced Transact-SQL components

</Customer>

</CustomersOrders>';

SELECT @x.query('for $i in CustomersOrders/Customer/Order

let $j := $i/orderdate

where $i/@orderid < 10900

order by ($j)[1]

return

<Order-orderid-element>

<orderid>{data($i/@orderid)}</orderid>

{$j}

</Order-orderid-element>')

AS [Filtered, sorted and reformatted orders with let clause];

E o retorno é um fragmento/documento em XML:

<Order-orderid-element>

<orderid>10308</orderid>

<orderdate>2014-09-18T00:00:00</orderdate>

</Order-orderid-element>

<Order-orderid-element>

<orderid>10692</orderid>

<orderdate>2015-10-03T00:00:00</orderdate>

</Order-orderid-element>

<Order-orderid-element>

<orderid>10702</orderid>

<orderdate>2015-10-13T00:00:00</orderdate>

</Order-orderid-element>

**Consultando e exibindo dados JSON**

JSON é um formato parecido com XML, mas com uma leitura mais simplificada e focado no compartilhamento de dados.

A formatação de arquivos JSON no SSMS não é muito boa. Para driblar essa dificuldade, sugere-se que os alias sejam separados por **.**, pq no momento da apresentação dos dados, a formatação fica mais legível.

Na cláusula **FOR JSON** podemos adicionar além das cláusulas **PATH** e **AUTO**, três cláusulas adicionais, são elas:

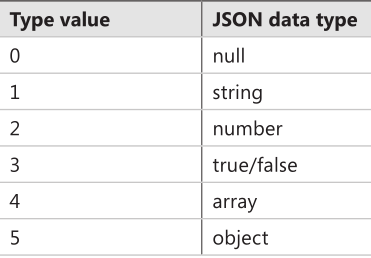
* **ROOT** adiciona um único membro superior
* **INCLUDE\_NULL\_VALUES** inclui valores nulos na apresentação. Nulos são, por padrão, excluídos da apresentação JSON
* **WITHOUT\_ARRAY\_WRAPPER** remove os colchetes na apresentação final do arquivo

**Converter dados JSON para formato tabular**

Assim como XML, podemos ler arquivos JSON e transforma-los para o modelo tabular, para isso utilizamos a função OPENJSON.

Quando não especificamos um schema, a função OPENJSON retorna uma tabela com três colunas:

* **KEY** o nome da propriedade JSON
* **VALUE** o valor real da propriedade
* **TYPE** tipo de dado JSON de um valor como um tiny int. Tabela abaixo mostra os valores possíveis para essa coluna e o seu significado



**Resumo deste Capítulo**

* Queries internas são totalmente independentes das queries externas. Elas são convenientes para identificarmos problemas pois podemos sempre executar a query interna de forma independente
* Subqueries relacionadas fazem referencias a colunas de tabelas da query externa e são, geralmente, mais complexas para trabalhar
* Existem casos em que o SQL Server trabalha melhor com subqueries e existem casos em que os joins são mais efetivos. É importante saber que existe essas duas possibilidades e testar ambos métodos para identificar o mais performático
* Usar o operador APPLY permite que você aplique uma expressão de tabela para cada linha de outra tabela. O operador CROSS APPLY não retorna a linha da esquerda se o lado direito é um conjunto vazio; nestes casos usar o operador OUTER APPLY é o correto
* Expressões de tabela são queries nomeadas que ajudam a simplificar e a reutilizar código. T-SQL oferece 4 tipos de expressões de tabela: CTE, tabelas derivadas, variáveis que retornam tabelas e views
* Usar tabelas derivadas e CTEs quando houver a necessidade de utilizar uma tabela em apenas uma declaração. Usar views e funções que retornam tabela quando você precisa reutilizar a expressão de tabela; nesses dois últimos casos são criados objetos dentro do banco e que permite a reutilização
* Use expressões de tabela (CTEs, tabelas derivadas) quando você não quiser interagir com o resultado da query interna; use tabelas temporárias ou variáveis de tabela quando quiser
* T-SQL mantém ambos, queries agrupadas de forma tradicional, que definem um conjunto de resultados, e queries agrupadas que definem diversos grupos de resultados utilizando as cláusulas GROUPING SETS, CUBE e ROLLUP. Usar a função GROUPING\_ID para computar identificadores de grupo
* Usar PIVOT para rotacionar dados do estado de linhas para colunas, e UNPIVOT para rotacionar dados de colunas para linhas. Com PIVOT, certifique-se de utilizar uma CTE que projete os elementos envolvidos, para evitar agrupamentos implícitos de colunas indesejadas. Com UNPIVOT, lembre-se que o operador remove as linhas NULL
* Funções de janela permitem que você execute operações de agregação e análise de dados em um intervalo específico de linhas. T-SQL mantém agregações, ranking, deslocamento e estatísticas
* Tabelas temporais (system-versioned) permitem que você mantenha um histórico de mudanças nos seus dados por longos períodos de tempo. Você permite que o sistema crie versões da tabela e conecte a uma tabela histórico correspondente. Tabelas temporais usam duas colunas do tipo DATETIME2 para representar o inicio e o final da validade do período de cada linha
* Para ler os dados, você consulta a tabela com a cláusula FOR SYSTEM\_TIME. Usando a sub cláusula AT você retorna o estado do dado em um período específico. Usando sub cláusulas FROM, BETWEEN, CONTAINED IN, você retorna o estado do dado quando for válido (em um intervalo específico de tempo)
* Você pode criar dados XML direto como saída de um SELECT com a cláusula FOR XML. Pode transformar dados XML em tabelas com a função OPENXML. Pode guardar dados XML em uma tabela com uma coluna com o tipo de dado XML. Pode usar expressões do XQuery como parâmetros para os métodos aplicáveis ao tipo de dado XML
* Não existe um tipo nativo JSON para o SQL Server. Entretanto, podemos criar documentos JSON direto das queries utilizando a função FOR JSON. Podemos criar tabelas a partir de arquivos do tipo JSON, extrair dados de segmentos JSON com as funções JSON\_VALUE e JSON\_QUERY, modificar arquivos JSON com a função JSON\_MODIFY e testar a validade com a função ISJSON.

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?

1. From a performance perspective, is it better to use table expressions or temporary tables?

Em termos de performance, expressões de tabela funcionam melhor, pois a leitura e processamento é feita em memória, sem necessidade de criação de uma tabela física. Entao, se você for interagir com um conjunto de dados produto de uma query UMA ÚNICA VEZ, é muito melhor fazer isso usando expressões de tabela; caso exista a necessidade de interagir várias vezes com esse conjunto de resultados, é recomendado criar uma tabela temporária e acessar os dados a partir dessa tabela temporária.

1. What are the limitations of the PIVOT operator in T-SQL?

- Não podemos usar a função COUNT() como função agregadora no PIVOT;

- Podemos utilizar apenas UMA função agregadora;

- Não podemos utilizar campos que são resultados de uma expressão na tabela fonte, nas cláusulas do PIVOT

- Não permite a parametrização dinâmica das colunas a serem rotacionadas

1. What are the limitations of the UNPIVOT operator in T-SQL?

- O UNPIVOT só permite realizar a rotação de uma única coluna com valores

- Não retorna valores NULL

- Não permite a parametrização dinâmica das colunas a serem rotacionadas

1. What are the best ways in T-SQL to compute totals and what are the performance considerations that you need to be aware of when using such computations?

Função de janela com recorte de ROWS UNBOUNDED PRECEDING. Muito mais eficiente do que utilizar a função RANGE, muito menos performática

1. Why can’t you place a row number calculation in the WHERE clause if you want to filter a range of row numbers?

Porque as funções de janela são executadas apenas no SELECT ou ORDER BY, ou seja, na quinta fase e após o FROM, WHERE, GROUP BY e HAVING. Ou seja, se você quiser utilizar uma função de janela na cláusula WHERE, você tem que criar essa query de forma externa, atribuir um alias e AÍ ENTÃO utilizar esse alias no WHERE.

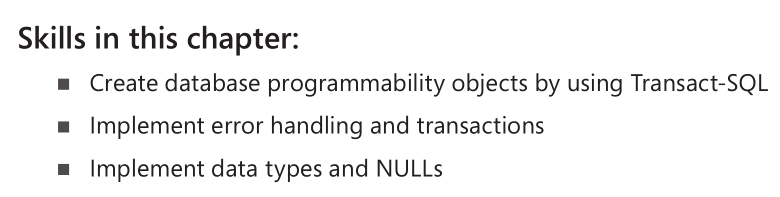
1. How would you quickly create, with minimal effort, na element-centric XML document from a T-SQL SELECT result set?

Usando a função FOR XML AUTO, combinando com as sub cláusulas ELEMENTS, ROOT; caso contrário teríamos um fragmento XML e não um documento XML.

1. What’s the difference between JSON\_VALUE and JSON\_QUERY functions?

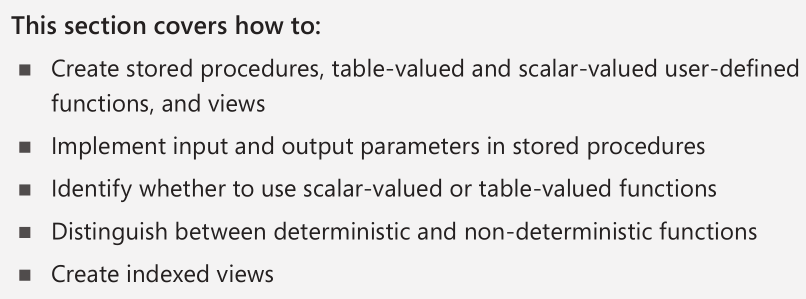
A função JSON\_VALUE retorna um valor escalar; a função JSON\_QUERY retorna um objeto ou um array

**Capítulo 3 – Programando banco de dados usando Transact-SQL**

****

**Criando objetos de programação usando Transact-SQL**

Foco aqui é criar objetos de programação usando T-SQL. Este capitulo cobre a simplificação e reutilização de lógicas aplicadas em query utilizando **views**; encapsulamento de expressões, queries e diversas declarações em **funções criadas pelo usuário**; e, por último, trabalhar com **stored procedures**.



**Views – Visualizações**

Views são queries nomeadas e que são reutilizáveis, ou *expressões de tabela*, que, por definição, estão armazenadas dentro do banco de dados. É acessível a usuários que têm permissão para consulta-las.

As views são especificamente uteis quando é necessário consultar queries complexas que já foram construídas. Ao invés de repetir o código da query, você consulta diretamente a view, que é uma representação da query; além disso, é possível unir dados das views com outras queries. Views também podem ser utilizadas como uma camada de segurança, autorizando usuários com acesso a view, mas não as tabelas subjacentes.

**Trabalhando com Views**

Essa view retorna o total de pedidos e valores de lucro juntando a tabela Orders e Sales (Pedidos e Vendas).

A opção **SCHEMABINDING** evita mudanças estruturais em tabelas dependentes e colunas enquanto essa View existir. Essa opção não é aplicada por padrão, mas existem situações que são mandatórias. Algumas pessoas enxergam essa opção como boa prática, já que aumenta a estabilidade do sistema e ajuda a evitar objetos no banco que dependem de objetos não existentes ou alterados, no banco.

Views são exemplos de expressões de tabela disponíveis no T-SQL. A query interna deve seguir esses três pré-requisitos:

* Todas colunas devem ter nomes
* Todas colunas devem ser únicas
* Não é permitido ORDER BY na query interna

Views podem ser criadas olhando para tabela/tabelas do banco, ou podem ser produtos de operações com CTEs encadeadas.

**Atributos das Views**

A opção SCHEMABINDING garante que uma view não tenha a sua estrutura alterada. Se for realizada a tentativa de excluir/alterar uma view com SCHEMABINDING, a seguinte mensagem será apresentada:

**ALTER TABLE DROP COLUMN address failed because one or more objects access this column.**

Quando criamos ou alteramos uma VIEW, seja com CREATE ou com CREATE ALTER VIEW, temos que especificar novamente qualquer atributo que você quer manter. O benefício em alterar a View, em oposição a dropar e criar, é manter as permissões.

O atributo **ENCRYPTION** faz com o que o SQL Server ofusque a definição do objeto que é guardada internamente. Sem esse atributo, você consegue acessar a definição do objeto usando a função OBJECT\_DEFINITION.

**Modificando dados com Views**

Existem restrições sensíveis em relação a alterar dados por meio de expressões de tabela. Uma delas é que se uma query junta varias tabelas, INSERT e UPDATE são permitidos afetar apenas uma tabela de destino por vez.

Normalmente você conseguiria incluir e atualizar linhas por meio de uma view mesmo que as modificações contradizem os filtros da query interior. No exemplo do livro, nós podemos incluir registros que tenham país diferente de USA, ou que tenham valor nulo.

Isso é proibido se, ao criarmos a view, incluirmos a opção **WITH CHECK** **OPTION**, que evita a inserção ou atualização de linhas pela view se os INSERT/UPDATE contradizem as condições da query interior à view. Entretanto, a opção WITH CHECK OPTION não previne a inserção de valores NULL, esses valores são permitidos.

**Views indexadas**

Criar um índice na view segue os mesmos requisitos de criar um índice em uma tabela normal. Algumas restrições estão presentes, como a view precisa ter uma coluna com COUNT\_BIG(\*) e não pode conter operações de conversão de dados CAST/CONVERT etc.

CREATE VIEW [dbo].[OrderTotals]

WITH SCHEMABINDING

AS

SELECT

S.[Sale Key] AS orderid

,S.[Customer Key] AS custid

,S.[Salesperson Key] AS empid

,S.[Delivery Date Key] AS orderdate

,S.[Invoice Date Key] AS requireddate

,SUM(S.[Quantity]) AS qty

,SUM(S.[Quantity] \* S.[Unit Price]) AS val

,COUNT\_BIG(\*) AS numorderlines

FROM Fact.Sale AS S

INNER JOIN Fact.[Order] O

ON S.[Sale Key] = O.[Order Key]

GROUP BY

S.[Sale Key], S.[Customer Key], S.[Salesperson Key], S.[Delivery Date Key], S.[Invoice Date Key]

GO

Criando o índice, o primeiro índice precisa ser um índice único e clusterizado

CREATE UNIQUE CLUSTERED INDEX idx\_cl\_orderid ON [dbo].[OrderTotals](orderid)

Uma vez criado, podemos criar índices diversos na view:

CREATE NONCLUSTERED INDEX idx\_nc\_custid ON [dbo].[OrderTotals](custid);

CREATE NONCLUSTERED INDEX idx\_nc\_empid ON [dbo].[OrderTotals](empid);

CREATE NONCLUSTERED INDEX idx\_nc\_orderdate ON [dbo].[OrderTotals](orderdate);

CREATE NONCLUSTERED INDEX idx\_nc\_shippeddate ON [dbo].[OrderTotals](shippeddate);

Para continuar a utilizar a função de CAST/CONVERT, você pode criar uma view intermediária que tenha sem a função, mas que tenha os índices. Após isso, você altera essa view e adiciona a função de CAST/CONVERT, o SQL Server permite esse ajuste.

* **Funções definidas pelo usuário**

Funções definidas pelo usuário são rotinas que executam cálculos, aceitam parâmetros e retornam valores escalares ou valores em formato de tabela. Essas rotinas podem ser escritas tanto em T-SQL quanto em CLR. O foco do exame são as funções escritas usando T-SQL.

Funções podem ser utilizadas em espaços em que você precise retornar um valor ou uma tabela. Podem aparecer tanto em uma query, quanto em uma coluna computada. Funções podem ser utilizadas para substituir uma procedure; para consultar o resultado de uma procedure; como alternativa para views, já que views não aceitam parâmetros; pode utilizar função para implementar uma segurança a nível de linha e várias outras funcionalidades.

Existem algumas restrições quando trabalhamos com funções, algumas dessas limitações são:

* Usar manipulador de erros (TRY...CATCH)
* Modificar dados (apenas variáveis de tabela)
* Usar DDL (data definition language)
* Usar tabelas temporárias
* Usar SQL dinâmico

T-SQL mantém 3 tipos de funções definidas pelo usuário:

* **Escalar** onde o tipo de retorno é um valor único; função pode ser chamada em um SELECT, WHERE, GROUP BY, HAVING. É preciso mencionar o tipo de dado retornado pela função. Exemplo:

CREATE FUNCTION GetDiseasedPatientsNumber (@DiseaseCode INT)

RETURNS INT

AS

BEGIN

DECLARE @Number\_of\_Patients INT

SELECT @Number\_of\_Patients = COUNT(PatientID)

FROM [dbo].[Patients]

WHERE DiseaseCode = @DiseaseCode

RETURN @Number\_of\_Patients

END

GO

* **Table valued function** onde o resultado retornado é um conjunto de linhas ao invés de um único valor. Pode ser chamada em FROM ou JOIN. Podem ser boas alternativas às Views, já que as Views não aceitam parâmetros.
* **Inline table valued function** essa função retorna APENAS tabelas e o corpo da função é uma única declaração de RETURN. Exemplo:

CREATE FUNCTION CustomerProductDetails (@CustomerID NCHAR(5))

RETURNS TABLE

AS

RETURN (

SELECT C.CustomerID

,C.CompanyName

,C.City

,O.OrderID

,O.OrderDate

,P.ProductName

,OD.UnitPrice

,OD.Quantity

,OD.Discount

FROM [Products] P

INNER JOIN [Order Details] OD ON P.ProductID = OD.ProductID

INNER JOIN Orders O ON O.OrderID = OD.OrderID

INNER JOIN Customers C ON C.CustomerID = O.CustomerID

WHERE C.CustomerID = @CustomerID

)

* **Multi statement table valued function** o corpo dessa função define uma declaração de RETURN junto com a declaração de uma variável do tipo TABLE. O corpo da função pode ter uma ou várias partes com código e um desses blocos de código irá popular a variável tabela. O escopo dessa função é restrito à função apenas. Exemplo:

CREATE FUNCTION CustomerOrderDetails (@CustomerID NCHAR(5))

RETURNS @CustomerOrders TABLE (

CustomerID NCHAR(5)

,CompanyName NVARCHAR(40)

,OrderID INT

,OrderDate DATETIME

)

AS

BEGIN

INSERT INTO @CustomerOrders

SELECT C.CustomerID

,C.CompanyName

,O.OrderID

,O.OrderDate

FROM Customers C

INNER JOIN Orders O ON C.CustomerID = C.CustomerID

WHERE C.CustomerID = @CustomerID

IF @@ROWCOUNT = 0

BEGIN

INSERT INTO @CustomerOrders

VALUES (

''

,'No Orders Found'

,0

,GETDATE()

)

END

RETURN

END

GO

**Diferenças entre funções Inline e Multi-statement**

**Inline:**

* O banco trata esse tipo de função como uma VIEW. Ou seja, o plano de execução para este tipo de função tem as mesmas estatísticas de consulta às tabelas utilizadas na função
* Por causa da razão acima, essas funções possuem melhor resultado de performance
* Não existe o peso extra de criar uma variável TABLE e popular

**Multi-statement**

* O banco precisa criar a variável TABLE, inserir dados e torna-la disponível para retorno. As estatísticas de performance aqui são diferentes do primeiro exemplo, pois o escopo não está limitado às tabelas utilizadas pela função
* Por causa da razão acima, esse tipo de função não é tão performático assim
* Necessário criar variável TABLE para esse tipo de função

Mais exemplos de funções de valor escalar:

CREATE OR ALTER FUNCTION dbo.SubtreeTotalSalaries(@mgr AS INT)

RETURNS MONEY

WITH SCHEMABINDING

AS

BEGIN

DECLARE @totalsalary AS MONEY;

WITH EmpsCTE AS

(

SELECT empid, salary

FROM dbo.Employees

WHERE empid = @mgr

UNION ALL

SELECT S.empid, S.salary

FROM EmpsCTE AS M

INNER JOIN dbo.Employees AS S

ON S.mgrid = M.empid

)

SELECT @totalsalary = SUM(salary)

FROM EmpsCTE;

RETURN @totalsalary;

END;

GO

A função acima aceita como parâmetro a variável mgr (managerID) e tem como retorno um valor MONEY. A opção SCHEMABINDING evita mudanças estruturais nas tabelas de origem (no caso do exemplo dbo.Employees) e a função em si tem o seu corpo definido entre as cláusulas BEGIN...END.

O código da função cria uma CTE de nome EmpsCTE, que seleciona empid e salario da tabela dbo.Employees, filtrando empid = @mgr; em seguinda o operador UNION ALL une a query acima com a EmpsCTE que por sua vez também está agrupada (JOIN) com a tabela Employees, igualadas por mgrid e empid.

O SELECT final cria uma variável @totalsalary para receber a SUM(salary) da EmpsCTE e retorna esse valor, finalizando, assim, o código da função retornando o valor armazenado em @totalsalary.

**Inline table-valued functions**

É uma função que tem uma estrutura parecida com uma View no sentido de que é baseada em uma única query e você interage como se fosse uma expressão de tabela, mas, ao contrario das views, a função permite um parâmetro de input.

O cabeçalho da função define os parâmetros de input e informa que retorna uma tabela (RETURN TABLE). Não existe um corpo da função (ausência de bloco BEGIN...END), ao invés disso, temos um único bloco RETURN em que a lógica da função se encontra. Abaixo um exemplo:

CREATE OR ALTER FUNCTION dbo.GetPage(@pagenum AS BIGINT, @pagesize AS BIGINT)

RETURNS TABLE

WITH SCHEMABINDING

AS

RETURN

WITH C AS

(

SELECT ROW\_NUMBER() OVER(ORDER BY orderdate, orderid) AS rownum,

orderid, orderdate, custid, empid

FROM Sales.Orders

)

SELECT rownum, orderid, orderdate, custid, empid

FROM C

WHERE rownum BETWEEN (@pagenum - 1) \* @pagesize + 1 AND @pagenum \* @pagesize;

GO

**Multistatement table-valued function**

Da mesma forma que a inline valued function acima, essa função também retorna uma tabela. Mas ao invés de ser baseada em um único bloco de código RETURN, essa função é responsável por criar uma tabela e popular por linhas em mais de um bloco de código.

Por debaixo dos panos o SQL Server cria uma variável do tipo TABLE (declarada no cabeçalho da função), executa o fluxo de código definido no corpo da função e assim quando o SQL executa o comando RETURN, retorna a variável de tabela para a query que chamou a função.

Por que implementar uma função multi-statement ao invés da inline? Com a função inline nós não temos controle sobre os índices da tabela retornada pela função. Com a função multi-statement nós controlamos as definições da variável do tipo TABLE, inclusive os índices. Além disso, a função do tipo inline retorna apenas uma tabela, já a função multi-statement pode ter um fluxo complexo com lógica extensa.

**Stored procedures**

Stored procedures são rotinas que permitem fluxos de código com várias etapas. Stored procedures aceitam vários elementos de T-SQL que uma função não aceitaria, por exemplo modificação de dados no banco de dados, modificação de dados e objetos do banco, uso de tabelas temporárias, SQL dinâmico. Essas são vantagens se comparados com as funções que vimos acima.

Procedures armazenam o cache do seu processamento, reutilizando em execuções subsequentes, otimizando uso de CPU e memória.

Importante entender quando usar views, funções ou stored procedure. **Views** são mais para dados consolidados, por exemplo, para garantir segurança as alterações não são feitas diretamente no dado, vc pode alterar a estrutura da query, mas não o que ela retorna; **Funções** podem ser uteis quando você precisa iterar sobre o resultado se a função retornar uma tabela, por exemplo. Além disso, podemos passar parâmetros para as funções, algo impossível de se fazer com views; **Stored procedures** são mais indicadas para modificar os dados, uso de T-SQL dinâmico, uso de tabelas temporárias etc.

**Trabalhando com stored procedures**

Por exemplo, precisamos criar uma rotina que varra uma tabela de acordo com certos parametros. O que podemos fazer é criar uma procedure, passar os parâmetros e executar a procedure. Exemplo:

CREATE OR ALTER PROC dbo.GetOrders

@orderid AS INT = NULL,

@orderdate AS DATE = NULL,

@custid AS INT = NULL,

@empid AS INT = NULL

AS

SET XACT\_ABORT, NOCOUNT ON;

SELECT orderid, orderdate, shippeddate, custid, empid, shipperid

FROM Sales.Orders

WHERE (orderid = @orderid OR @orderid IS NULL)

AND (orderdate = @orderdate OR @orderdate IS NULL)

AND (custid = @custid OR @custid IS NULL)

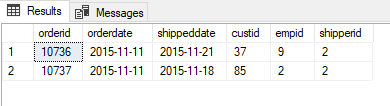
AND (empid = @empid OR @empid IS NULL);

GO

Executando a procedure podemos passar apenas um parâmetro, especificando com @parametro e aplicando a opção DEFAULT para os outros. Por exemplo podemos executar a proc assim:

EXEC dbo.GetOrders default, '20151111'

O resultado seria o seguinte:



Obtemos o mesmo resultado se criarmos uma função com a mesma estrutura da proc e as chamássemos passando os parâmetros:

CREATE FUNCTION dbo.GetOrdersFn (@orderid AS int, @orderdate AS date, @custid AS int, @empid AS int)

RETURNS TABLE

WITH SCHEMABINDING

AS

RETURN

SELECT orderid, orderdate, shippeddate, custid, empid, shipperid

FROM Sales.Orders

WHERE (orderid = @orderid OR @orderid IS NULL)

AND (orderdate = @orderdate or @orderdate IS NULL)

AND (custid = @custid or @custid IS NULL)

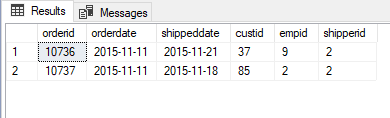
AND (empid = @empid or @empid IS NULL)

GO

Executando a função GetOrders:

SELECT \* FROM DBO.GetOrdersFn(default, '20151111', default, default)

Resultado:



Nestes casos procedure e função trazem os mesmos resultados.

**Stored procedures com SQL dinâmico**

A etapa anterior demonstrou o uso de procedure para tratar uma busca dinâmica para consultar dados da tabela Sales.Orders usando uma query estática.

Esta seção vai tratar de soluções alternativas usando SQL dinâmico, que é uma técnica que envolve construir um bloco de código como uma string, normalmente usando uma variável, e então dizendo para o SQL Server executar o código que está dentro da variável.

Vamos alterar a procedure que criamos acima para atender as condições de SQL dinâmico:

CREATE OR ALTER PROC dbo.GetOrders

@orderid AS INT = NULL,

@orderdate AS DATE = NULL,

@custid AS INT = NULL,

@empid AS INT = NULL

AS

SET XACT\_ABORT, NOCOUNT ON;

DECLARE @sql AS NVARCHAR(MAX) = N'SELECT orderid, orderdate, shippeddate, custid, empid,

shipperid

FROM Sales.Orders

WHERE 1 = 1'

+ CASE WHEN @orderid IS NOT NULL THEN N' AND orderid = @orderid ' ELSE N'' END

+ CASE WHEN @orderdate IS NOT NULL THEN N' AND orderdate = @orderdate' ELSE N'' END

+ CASE WHEN @custid IS NOT NULL THEN N' AND custid = @custid ' ELSE N'' END

+ CASE WHEN @empid IS NOT NULL THEN N' AND empid = @empid ' ELSE N'' END

+ N';'

EXEC sys.sp\_executesql

@stmt = @sql,

@params = N'@orderid AS INT, @orderdate AS DATE, @custid AS INT, @empid AS INT',

@orderid = @orderid,

@orderdate = @orderdate,

@custid = @custid,

@empid = @empid;

GO

O que fizemos acima foi: declarar uma variável do tipo nvarchar de nome **@sql** e nela guardamos a query com a lista de elementos do SELECT com a condição WHERE 1 = 1, onde sempre retorna true para não termos problemas na execução; após isso usamos expressões CASE...WHEN para cada parâmetro aplicável à query, o código concatena os atributos parametrizáveis com o filtro, de outra forma uma string vazia.

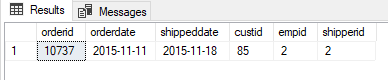
Então o código é executado dentro da procedure **sp\_executesql**.A primeira parte da procedure pega a variável @stmt e atribui a ela o valor da variável @sql, definida anteriormente.

A segunda parte da proc é com o parâmetro **@params**, onde vc declara todos os parâmetros do código dinâmico. No nosso caso nós declaramos um bloco de código com os parâmetros da procedure (orderid, orderdate, custid, empid).

A última parte é uma serie de atribuições dos parâmetros da procedure com os seus correspondentes no bloco de código:

EXEC dbo.GetOrders @orderdate = '20151111', @custid = 85;

Resultado:



**Usando parâmetros de output e modificando dados**

Suponhamos que seja necessário criar uma lista de numero de faturas distintos e sequenciais; poderíamos criar uma tabela que receberia esses valores com um acoluna IDENTITY, mas isso pode falhar.

A sugestão do livro é criar uma procedure que atualize a coluna de valores de numero de fatura e somente atualize esse valor quando uma nova transação for concluída.

Implementamos um bloco de tratamento de erros com IF ROWCOUNT = 0 (ou seja não foi inserido registro), utilizamos THROW 51001 e especificamos o erro.

A procedure ficaria dessa forma:

CREATE OR ALTER PROC dbo.GetSequenceValue

@seqname AS VARCHAR(128),

@val AS INT OUTPUT

AS

SET XACT\_ABORT, NOCOUNT ON;

UPDATE dbo.MySequences

SET @val = val += 1

WHERE seqname = @seqname;

IF @@ROWCOUNT = 0

THROW 51001, 'Specified sequence was not found.', 1;

GO

Para geramos um numero novo de fatura poderíamos seguir o script abaixo:

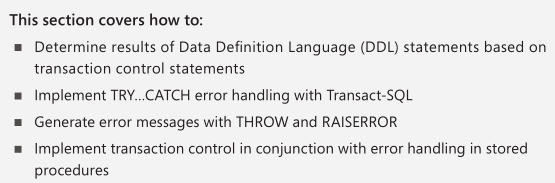
DECLARE @newinvoicenumber AS INT;

exec dbo.GetSequenceValue @seqname = 'SEQINVOICES', @val = @newinvoicenumber OUTPUT;

SELECT @newinvoicenumber as newinvoicenumber

O código acima usa uma variável local para receber o valor retornado pelo parâmetro de output; perceba que também temos que indicar a palavra chave OUTPUT novamente caso contrário o SQL Server entenderia a variável como um novo parâmetro.

**Implementando tratamento de erros e transações**



Uma transação deve seguir os princípios *ACID*. **A**tomicidade, **C**onsistência, **I**ntegridade, **D**urabilidade.

* **Uma transação deve ser atômica** no sentido de ou ela é completa em sua totalidade ou ela não é nem feita. Exemplo daquele caso de INSERT VALUES em duas declarações, não podemos garantir que OS DOIS ou NENHUM vão ser executados
* **Deve ser consistente** ao passo que uma transação deveria transferir o banco de um estado de consistência para outro estado de consistência em termos de aderência ao modelo de dados, constraints e triggers
* **A transação deve ser isolada** uma vez que estados de inconsistência intermediários supostamente não deveriam ser visíveis para outras transações além daquela que executa as mudanças.
* **Uma transação é durável** quando podemos garantir que os registros foram inseridos e commitados de forma correta. Isso significa que as alterações sobreviverão a uma falha de energia ou falha de algum componente

**Definindo transações**

Transações sempre estarão inseridas dentro de um bloco BEGIN...END TRANSACTION:

USE TSQLV4;

SET XACT\_ABORT, NOCOUNT ON;

-- start a new transaction

BEGIN TRAN;

-- declare a variable

DECLARE @neworderid AS INT;

-- insert a new order into the Sales.Orders table

INSERT INTO Sales.Orders

(custid, empid, orderdate, requireddate, shippeddate,

shipperid, freight, shipname, shipaddress, shipcity,

shippostalcode, shipcountry)

VALUES

(1, 1, '20170212', '20170301', '20170216',

1, 10.00, N'Shipper 1', N'Address AAA', N'City AAA',

N'11111', N'Country AAA');

-- save the new order id in the variable @neworderid

SET @neworderid = SCOPE\_IDENTITY();

PRINT 'Added new order header with order ID ' + CAST(@neworderid AS VARCHAR(10))

+ '. @@TRANCOUNT is ' + CAST(@@TRANCOUNT AS VARCHAR(10)) + '.';

-- insert order lines for new order into Sales.OrderDetails

INSERT INTO Sales.OrderDetails(orderid, productid, unitprice, qty, discount)

VALUES(@neworderid, 1, 10.00, 1, 0.000),

(@neworderid, 2, 10.00, 1, 0.000),

(@neworderid, 3, 10.00, 1, 0.000);

PRINT 'Added order lines to new order. @@TRANCOUNT is '

+ CAST(@@TRANCOUNT AS VARCHAR(10)) + '.';

-- commit the transaction

COMMIT TRAN

BEGIN TRAN 🡪 marca o início da transação

COMMIT TRAN 🡪 finaliza a transação

ROLLBACK TRAN 🡪 restaura a transação para o estado anterior

Podemos consultar @@TRANCOUNT para saber se a transação atual está aberta ou está finalizada. Se você está em uma transação ainda aberta o valor de TRANCOUNT vai ser maior que 0, de outra forma retorna 0.

**Tratando erros com bloco TRY-CATCH**

Mesma lógica do TRY-CATCH em C#. Você coloca o seu código dentro do bloco **TRY** e a rotina a ser executada caso o código falhe, dentro do bloco **CATCH**.

A procedure abaixo ilustra esse tipo de construção:

[dbo].[AddRowToT1]

@keycol INT,

@col1 INT,

@col2 VARCHAR(10)

AS

SET NOCOUNT ON;

BEGIN TRY

INSERT INTO dbo.T1(keycol, col1, col2)

VALUES(@keycol, @col1, @col2);

PRINT 'Got to the end of the outer TRY block.';

END TRY

BEGIN CATCH

PRINT 'Error occurred in outer TRY block. Entering outer CATCH block.';

BEGIN TRY

INSERT INTO dbo.ErrorLog(errormessage) VALUES(ERROR\_MESSAGE());

PRINT 'Got to the end of the inner TRY block.';

END TRY

BEGIN CATCH

PRINT 'Error occurred in inner TRY block. Entering inner CATCH block. Error message:

' + ERROR\_MESSAGE();

END CATCH;

END CATCH;

Criando blocos TRY-CATCH aninhados pode ser útil se for necessário se existe a possibilidade de um primeiro CATCH falhar. Você coloca um TRY-CATCH dentro de um CATCH.

Exemlo de tratamento de erros com TRY-CATCH:

BEGIN TRY

INSERT INTO dbo.T1(keycol, col1, col2)

VALUES(1, 10, 'AAA');

INSERT INTO dbo.T1(keycol, col1, col2)

VALUES(2, -20, 'BBB');

PRINT 'Got to the end of the TRY block.';

END TRY

BEGIN CATCH

PRINT 'Error occurred. Entering CATCH block. Error message: ' + ERROR\_MESSAGE();

END CATCH;

GO

SELECT keycol, col1, col2

FROM dbo.T1;

Aqui o primeiro TRY insere valores na tabela T1. O primeiro INSERT é válido, mas o segundo não. O segundo fere uma condição de FK da tabela. O constructo CATCH trata desse erro imprimindo na tela o erro e uma mensagem.

O código abaixo também produz um erro, mas que também é tratado pela procedure exibindo uma mensagem de erro e em qual bloco CATCH está sendo feito o tratamento:

EXEC dbo.AddRowToT1

1,10,'AAA'

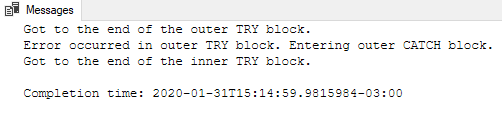
GO

EXEC dbo.AddRowToT1

1,11230,'BBBB'

GO

Resultado:



**Funções de erros**

T-SQL possui 6 funções de erros que oferecem informações sobre os erros e que você pode consultar dentro do bloco CATCH. Abaixo estão as funções e as suas descrições:

* **ERROR\_NUMBER()** – mostra o número do erro
* **ERROR\_MESSAGE()** – mostra a mensagem de erro
* **ERROR\_SEVERITY()** – mostra a severidade do erro. Dentro do bloco CATCH podemos tratar erros de severidade entre 11 e 19. Erros com severidade maior que isso causam o abort da conexão. Severidade entre 0 e 9 são mostradas no próprio client; SQL Server tratam essas como severidade apenas como informativa.
* **ERROR\_STATE() –** mostra o estado do erro. O estado do erro é um inteiro que varia entre 1 e 255. Pode ser usado para diferentes propósitos, o mais utilizado é mostrara onde o erro teve origem no código, facilitando o tratamento
* **ERROR\_LINE()** – mostra a linha em que o erro ocorreu
* **ERROR\_PROCEDURE() –** mostra o nome da procedure onde o erro ocorreu. Se o erro não aconteceu em uma procedure essa função retorna um NULL

Algumas ressalvas quando utilizamos essas funções:

* Se chamarmos essas funções **FORA DE UM BLOCO CATCH** todas elas retornam valor NULL
* Se aninharmos blocos TRY-CATCH, direta ou indiretamente quando chamamos uma procedure dentro de outra proc, essas funções retornam informações de erros sobre o erro mais interno no código
* Algumas falhas geram uma cadeia de erros. Essas funções retornam apenas informações sobre o ultimo erro na cadeia

Convenientemente nós podemos chamar essas funções dentro de uma procedure devido à reusabilidade do código. Ainda assim, a função ERROR\_PROCEDURE() vai retornar a procedure em que o erro ocorreu e não o nome da procedure que você chamou a função. Exemplo abaixo de como usar uma procedure para listar erros, dentro de outra procedure:

Procedure que exibe os erros:

CREATE OR ALTER PROC dbo.PrintErrorInfo

AS

PRINT 'Error Number : ' + CAST(ERROR\_NUMBER() AS VARCHAR(10));

PRINT 'Error Message : ' + ERROR\_MESSAGE();

PRINT 'Error Severity: ' + CAST(ERROR\_SEVERITY() AS VARCHAR(10));

PRINT 'Error State : ' + CAST(ERROR\_STATE() AS VARCHAR(10));

PRINT 'Error Line : ' + CAST(ERROR\_LINE() AS VARCHAR(10));

PRINT 'Error Proc : ' + COALESCE(ERROR\_PROCEDURE(), 'Not within proc');

GO

Chamando essa procedure dentro de um bloco CATCH de outra procedure:

CREATE OR ALTER PROC dbo.AddRowToT1 -- proc name

@keycol INT,

@col1 INT,

@col2 VARCHAR(10)

AS

SET NOCOUNT ON;

BEGIN TRY

INSERT INTO dbo.T1(keycol, col1, col2) -- line 11

VALUES(@keycol, @col1, @col2);

END TRY

BEGIN CATCH

EXEC dbo.PrintErrorInfo;

END CATCH;

GO

EXEC dbo.AddRowToT1 @KEYCOL = 1, @COL1 = -10, @COL2 = 'AAA'

A execução da procedure acima causa o seguinte erro:

Error Number : 2627

Error Message : Violation of PRIMARY KEY constraint 'PK\_T1'. Cannot insert duplicate key in object 'dbo.T1'. The duplicate key value is (1).

Error Severity: 14

Error State : 1

Error Line : 8

Error Proc : dbo.AddRowToT1

Completion time: 2020-02-04T09:43:00.6054337-03:00

Mostrando o numero, a mensagem, a severidade, o estado, a linha e em qual procedure o erro ocorreu.

**Os comandos THROW e RAISERROR**

Os dois comandos permitem que erros sejam evidenciados. Ambos permitem mostrar um erro definido pelo usuário, mas somente o THROW permite que repassamos um erro original que foi pego dentro de um bloco TRY-CATCH.

**O comando THROW**

O comando THROW aceita duas sintaxes possíveis. Uma sem parâmetros, que você utiliza dentro de um bloco CATCH para repassar esse erro que originalmente foi capturado dentro de bloco CATCH. O exemplo abaixo mostra como utilizamos o THROW sem os parâmetros:

CREATE OR ALTER PROC dbo.Divide

@dividend AS INT,

@divisor AS INT

AS

SET NOCOUNT ON;

BEGIN TRY

SELECT @dividend / @divisor AS quotient, @dividend % @divisor AS remainder;

END TRY

BEGIN CATCH

PRINT 'Error occurred when trying to compute the division '

+ CAST(@dividend AS VARCHAR(11)) + ' / ' + CAST(@divisor AS VARCHAR(11)) + '.';

THROW;

PRINT 'This doesn''nt execute.';

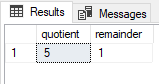
END CATCH;

GO

Essa procedure usa o bloco TRY para executar uma divisão e mostrar o resultado e o resto. Se tudo der certo, o resultado é mostrado na tela e o bloco CATCH não é nem mostrado em tela. Caso algum erro ocorra, o bloco CATCH é ativado; a primeira tarefa que o bloco CATCH é imprimir uma mensagem indicando o erro com a tentativa de cálculo e então é repassada a mensagem original do erro. O comando THROW aborta a execução do lote de código. Podemos ver que ainda teria uma instrução PRINT para ser executada, mas o comando THROW passa por cima.

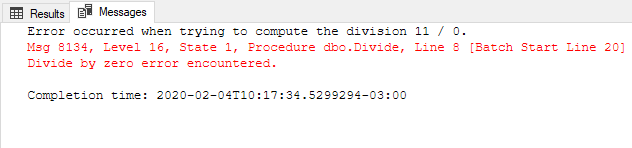
EXEC dbo.Divide @dividend = 11, @divisor = 2;

A execução acima traz o seguinte resultado:



EXEC dbo.Divide @dividend = 11, @divisor = 0;

A execução acima traz como resultado as seguintes mensagens:



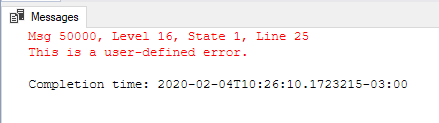
A segunda sintaxe possível para o comando THROW é com parâmetros. Funciona da seguinte forma: usamos essa sintaxe para mostrarmos erros definidos pelo usuário. Podemos utilizar essa função dentro e fora de blocos CATCH. A forma geral de THROW com parâmetros é a seguinte:

THROW errornumber, message, state;

Todos os parâmetros podem ser constantes ou variveis. Aqui estmos usando com parâmetros:

THROW 50000,'This is a user-defined error.', 1;

O código acima gera o seguinte resultado:



O primeiro parâmetro é um inteiro, especificado pelo usuário e obrigatoriamente maior que 50000. O numero do erro não precisa ser gravado em um lugar previamente.

O segundo parâmetro é uma mensagem especificada pelo usuário. O segundo argumento não aceita parâmetros no meio da mensagem direto. Se quisermos inserirmos valores de input como parte da mensagem, precisamos construir a msg em uma variável previamente ou usar a função FORMATMESSAGE.

O terceiro parâmetro é um inteiro, também especificado pelo usuário, variando entre 1 e 255. Esse valor é utilizado como forma de indicar onde o erro foi capturado.

Abaixo listamos como exemplo a utilização de uma variável junto com a função FORMATMESSAGE para utilizarmos na hora de usarmos o THROW:

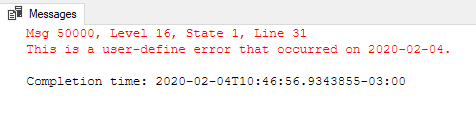
DECLARE @msg AS NVARCHAR(2048) =

FORMATMESSAGE('This is a user-define error that occurred on %s.',

CONVERT(CHAR(10), SYSDATETIME(), 121));

THROW 50000, @msg, 1;

Esse código gera o seguinte resultado:



Se executarmos o THROW fora de um bloco TRY-CATCH, sempre abortará a execução do nosso código. Se você executar dentro do bloco TRY, isso afeta o bloco CATCH correspondente. O exemplo abaixo mostra o comportamento de abortar o bloco de código:

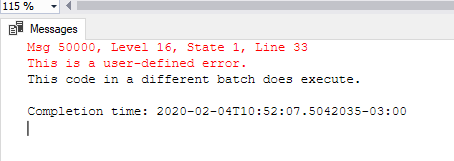
THROW 50000, 'This is a user-defined error.', 1;

PRINT 'This code in the same batch doesn''t execute.';

GO

PRINT 'This code in a different batch does execute.';

O resultado é o seguinte



Percebemos que o primeiro PRINT, logo após o comando THROW não é executado. Mas o segundo PRINT é executado, pois está fora do primeiro corpo de código.

Quando usamos o THROW em uma transação ainda aberta e com o XACT\_ABORT OFF a transação continua aberta e passível de commit.

Quando usamos THROW com o XACT\_ABORT ON e em uma transação aberta, se não estamos usando TRY-CATCH, o SQL Server aborta a transação. Se estamos utilizando TRY-CATCH, a transação fica marcada como ***doom***, vamos ver o que é isso mais pra frente.

O exemplo abaixo mostra o que acontece quando usamos o THROW com o XACT\_ABORT OFF:

SET XACT\_ABORT OFF;

BEGIN TRAN;

THROW 50000, 'Hello from THROW.', 1;

PRINT 'This doesn''t execute.';

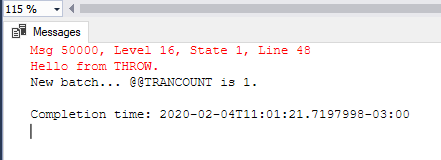
GO

PRINT 'New batch... @@TRANCOUNT is ' + CAST(@@TRANCOUNT AS VARCHAR(10)) + '.';

IF @@TRANCOUNT > 0

ROLLBACK TRAN;

O resultado é o seguinte:



A utilização do THROW no primeiro lote **ENCERRA** o lote, mas não a transação. Se olharmos, tem um PRINT dentro do primeiro lote que não é executado; mas o segundo PRINT (que está em outro lote) é executado e a transação ainda é marcada como ativa.

Agora com o XACT\_ABORT ON:

SET XACT\_ABORT ON;

BEGIN TRAN;

THROW 50000, 'Hello from THROW.', 1;

PRINT 'This doesn''t execute.';

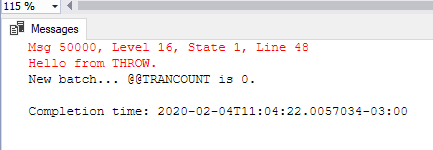
GO

PRINT 'New batch... @@TRANCOUNT is ' + CAST(@@TRANCOUNT AS VARCHAR(10)) + '.';

IF @@TRANCOUNT > 0

ROLLBACK TRAN;

Temos o seguinte resultado:



O que esse resultado tem de diferente do outro? Dessa vez a execução do THROW, encerra o primeiro bloco de código e inicia o segundo; este, por sua vez, por estarmos com o XACT\_ABORT ON, não commita a transação, causando assim no @@TRANCOUNT = 0.

**O comando RAISERROR**

O comando RAISERROR é o predecessor do THROW. A sintaxe é a seguinte:

RAISERROR ( { message | messageid }, severity, state [, arguments for parameter markers

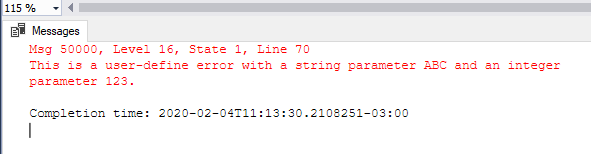
] ) [ WITH options ];

Exemplo abaixo de utilização do RAISERROR:

RAISERROR( 'This is a user-define error with a string parameter %s and an integer

parameter %d.', 16, 1, 'ABC', 123 )

O resultado é o seguinte:



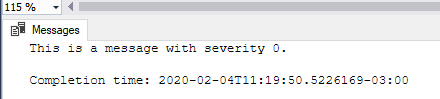
O primeiro parâmetro pode ser uma mensagem de texto com um parâmetro opcional (com sintaxe parecida com aquele esquema do C; de %s, para parâmetro tipo string, %d, para parâmetro tipo int etc). Nesse caso o numero do erro vai ser sempre 50000. O parâmetro da quarta posição pra frente vai ser sempre os valores que serão substituídos. No nosso caso ‘ABC’ e 123.

O segundo parâmetro é a severidade do erro. No nosso caso 16. O resultado de elevar um erro com uma certa severidade é semelhante ao resultado de um erro gerado pelo sistema com a mesma severidade. Quer dizer, mensagens com severidade entre 0 e 10 são consideradas informativas apenas. Com severidade de 0 e 10 a mensagem é printada sem cabeçalho; com severidade entre 1 e 9, a mensagem é printada com cabeçalho.

Exemplo de erro com severidade 0:

RAISERROR( 'This is a message with severity 0.', 0, 1 );

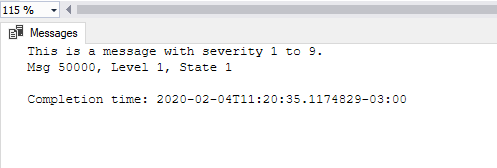
Resultado:



O exemplo abaixo mostra um erro com severidade entre 1 e 9:

RAISERROR( 'This is a message with severity 1 to 9.', 1, 1 )

Resultado:



Erros com severidade menor que 11 não são capturáveis. Erros com severidade entre 11 e 19 são capturáveis. Erros com severidade entre 20 e 25 finalizam a conexão. Para elevar um erro com severidade >19 é necessário utilizar a opção **WITH LOG**, que faz com que seja registrado log do erro no SQL e também no log de aplicações do Windows. Para usarmos a opção WITH LOG, precisamos ter a role sysadmin ou ter a permissão **ALTER TRACE**

O terceiro parâmetro é um inteiro com valor entre 1 e 255, similar ao valor de estado utilizado no THROW.

Existem parâmetros adicionais que podemos especificar como parte da clausula WITH, os mais utilizados são o NOW e o **NOWAIT**. O **NOWAIT** faz com que o SQL Server mande uma mensagem imediatamente pelo client sem esperar pelo buffer interno. Desenvolvedores geralmente usam essa opção quando executam scripts muito grandes para reportar o avanço do código nas suas marcações intermediarias.

Exemplo abaixo mostra a utilização da opção **NOWAIT**:

RAISERROR( 'First message.', 0, 1 ) WITH NOWAIT;

WAITFOR DELAY '00:00:05';

RAISERROR( 'Second message.', 0, 1 ) WITH NOWAIT;

Nesse código, a primeira mensagem é impressa na tela mas a segunda só é impressa 5 segundos depois.

A opção **WITH LOG** registra informações sobre a mensagem de erro e encerra a conexão:

RAISERROR( 'This is a message with severity 20.', 20, 1 ) WITH LOG;

O comando THROW não oferece opções de clausula como o LOG e NOWAIT, então pode ser uma vantagem neste sentido utilizar o RAISERROR.

Apenas erros com severidade >= 20 encerram a conexão. Se elevarmos um erro com severidade menor que 20, isso não encerra a conexão, ou encerra a execução do bloco de código. O código a seguir demonstra isso:

SET XACT\_ABORT ON;

BEGIN TRAN;

RAISERROR( 'This is a user-defined error.', 16, 1 );

PRINT 'This code in the same batch executes. @@TRANCOUNT is ' + CAST(@@TRANCOUNT AS

VARCHAR(10)) + '.';

IF @@TRANCOUNT > 0

ROLLBACK TRAN;

Na prova o que pode ser perguntado é quais são as diferenças entre THROW e RAISERROR. A tabela abaixo tenta evidenciar isso de uma forma mais clara:

|  |  |  |
| --- | --- | --- |
| **PROPRIEDADE** | **THROW** | **RAISERROR** |
| Consegue repassar erros originais do sistema? | SIM | NÃO |
| Ativa o bloco CATCH | SIM | SIM, para severidade > 10 e severidade < 20 |
| Sempre aborta o lote de código quando não utilizamos TRY-CATCH? | SIM | NÃO |
| Aborta transação se XACT\_ABORT OFF? | NÃO | NÃO |
| Aborta transação se XACT\_ABORT ON? | SIM | NÃO |
| Se o numero de erro é passado (como parametro), erro deve ser definido em sys.messages | NÃO | SIM |
| Aceita parâmetros com sintaxe tipo printf (linguagem C) | NÃO | SIM |
| Aceita indicar a severidade | NÃO | SIM |
| Aceita opções WITH LOG e NOWAIT? | NÃO | SIM |
| Aceita que WITH NOWAIT mande mensagens direto pelo client | NÃO | SIM |
| Declarações anteriores precisam ser terminadas? | SIM | NÃO |

**Tratando erros dentro de transações**

Existem diversas possibilidades de erros dentro do SQL. A maioria delas foi abordada na ultima seção, mas existem especificidades para erros gerados dentro de uma transação. Recomenda-se sempre utilizarmos a opção XACT\_ABORT ON e usarmos blocos TRY-CATCH.

O exemplo abaixo demonstra um erro de divisão por zero, o que normalmente não encerra o bloco de código e não encerra a conexão:

SET XACT\_ABORT OFF;

BEGIN TRAN;

DECLARE @i AS INT = 10/0;

PRINT 'Batch wasn’t aborted.';

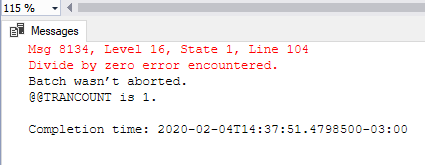
GO

PRINT '@@TRANCOUNT is ' + CAST(@@TRANCOUNT AS VARCHAR(10)) + '.';

IF @@TRANCOUNT > 0

ROLLBACK TRAN

Resultado:



O que esse resultado quer dizer? O bloco de código não foi encerrado devido ao erro e a transação também não foi. Olhando para a função PRINT executada, vemos que foi executada de forma normal e a transação também foi executada normalmente.

O código abaixo exemplifica:

BEGIN TRAN;

SET XACT\_ABORT ON;

DECLARE @i AS INT = CAST('1,759' AS INT);

PRINT 'Batch wasn''t aborted.';

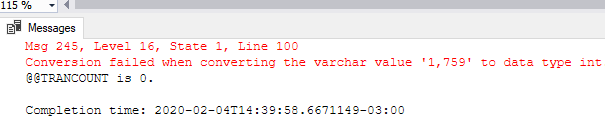
GO

PRINT '@@TRANCOUNT is ' + CAST(@@TRANCOUNT AS VARCHAR(10)) + '.';

IF @@TRANCOUNT > 0

ROLLBACK TRAN;

Resultado:



Como setamos a opção XACT\_ABORT ON, nós temos um comportamento mais consistente no sentido de que a maioria dos erros que não encerrariam a transação/código são tratados; com essa opção nós encerramos o bloco de código e damos ROLLBACK na transação.

Dessa forma vimos que com a opção XACT\_ABORT ON, nós encerramos o código e a transação é abortada. Quando usamos blocos TRY-CATCH para tratamento de erros, podemos ver o status da transação (se está aberta ou fechada) e se queremos commitar ou dar rollback. Casos onde as transações são **ABORTADAS** também são perdidos os dados que foram gerados durante a transação. Para casos assim, onde não estamos utilizando **TRY-CATCH**, a transação fica em um estado INCOMMITÁVEL, aquele doom que nós falamos acima.

Então, quando estamos usando TRY-CATCH, existem 3 resultados possíveis de erros em termos de estado da transação:

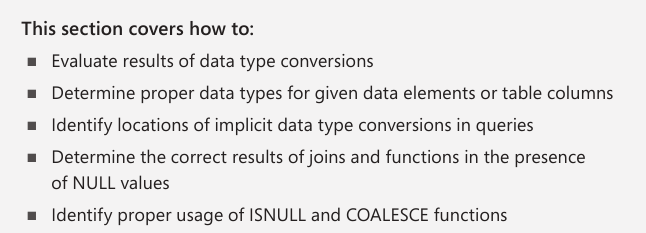
* A transação pode estar aberta e commitável
* A transação pode estar aberta e incommitável (doom)
* A transação não está aberta

Para tratarmos os erros de uma forma mais efetiva, não basta só checarmos @@TRANCOUNT, pois apenas nos informa se a transação está aberta, não diz se está commitável ou incommitável. T-SQL oferece outra função chamada XACT\_STATE que faz a distinção entre os 3 estados listados acima.

**Retorna 0** quando nenhuma transação está aberta (igual a @@TRANCOUNT = 0); **Retorna 1** quando a transação está aberta e commitável e **Retorna -1** quando a transação está incommitável.

**Implementando dados de vários tipos e NULLs**

Alguns tópicos dessa seção são familiares e já vimos anteriormente. Por exemplo funções COALESCE, ISNULL. Aqui é mais uma chance de revisarmos esses tópicos



**Trabalhando com tipos de dados**

Os tipos de dados é um tópico importantíssimo para os nossos dados. Tudo que fazemos em um banco opera em cima dos dados, então os tipos de dados não é algo trivial, é algo para olharmos com carinho e atenção.

Esta seção oferece recomendações para escolher o tipo apropriado de dados para diferentes elementos do banco, escolhendo tipos de dados para chaves, tipos de dados para conversões etc.

**Escolhendo o tipo apropriado para os dados**

Essa seção aponta a importância da consistência de dados como uma característica intrínseca ao banco de dados. Ter os tipos de dados definidos de forma clara é uma garantia de consistência de dados. Por exemplo uma coluna de datas só aceitará o tipo data, algo como ‘abc’ ou ‘20191010’ não será aceito como data válida.

Ter uma coluna de datas como INT não garante que um valor como 999999 seja inserido. Os tipos de dados representam uma **LIMITAÇÃO**, mas que garante a integridade dos formatos dos dados.

OS tipos de dados string (varchar, char, nvarchar, nchar) são usados dependendo da necessidade. Para casos em que os campos não variam de tamanho, ter um tamanho fixo é mais negócio para a leitura. Se temos um campo que varia muito de tamanho, podemos nos beneficiar de um campo variável.

**Escolhendo tipos de dados para as chaves**

Quando criamos uma chave tendo em conta uma regra de negócio ou um atributo já existente no banco nem temos muita opção de escolher além do que é possível e necessário. Mas quando criamos uma chave a partir de novos atributos, precisamos escolher o tipo de dado apropriado.

Precisamos sempre ver o tradeoff de cada solução. Onde uma é melhor que a outra, pode ser pior em outro ponto. Temos como opções típicas para criação de chaves as seguintes opções:

* **Propriedade identity em uma coluna** gera de forma automática chaves em um atributo/coluna de tipo numérico com uma escala de 0. Qualquer **NUMERAL** pode ser identity (tinyint, int, bigint, numeric, decimal)
* **The sequence object** são objetos independentes dentro do banco no qual você pode obter valores sequenciais novos. Assim como identity, aceita qualquer tipo numérico com escala de 0. Diferentemente de identity, não está atrelado a uma coluna particular; ao invés disso, como mencionado, é um objeto independente do banco. É possível
* **Nonsequential Global Unique Identifiers** são gerados a partir da função NEWID(), que gera um novo GUID. Vantagem dos GUIDs é que é garantido que serão únicos durante o tempo e espaço em que existirem
* **Sequential GUIDs** são gerados a partir da função NEWSEQUENTIALID(). Essa função só é permitida dentro do valor DEFAULT de uma constraint que está associada a uma coluna
* **Soluções personalizadas** quando não queremos utilizar ferramentas nativas do SQL para gerar as chaves, precisamos desenvolver a nossa própria solução. Um exemplo seria guardar o valor utilizado em uma tabela e toda vez que precisarmos de um novo valor, faríamos um UPDATE na tabela e usaríamos o valor novo que foi inserido como chave

Importante levar em conta, além do tipo de dado utilizado como chave, é o **tamanho** do campo utilizado. INT utiliza 4 bytes, BIGINT 8, UNIQUEIDENTIFIER utiliza 16 e assim por diante. Se utilizarmos chaves sequenciais, todas as linhas são direcionadas para a parte direita inferior, como no final de uma página. Dessa forma os procedimentos de inserção e leitura são otimizados.

Isso pode ser um problema quando estamos realizando cargas em varias etapas de trabalho (o que é a regra em um banco OLTP). Esse gargalo se dá porque existem várias requisições para obter o índice da página mais a direita. Como só uma requisição é aceita por vez, acontece o enfileiramento dessas requisições, causando assim uma perda de performance com esse tipo de chave em ambientes que exigem várias cargas de trabalho.

Se considerarmos índices não sequenciais (NEWID(), UPDATE em valores) nós temos a divisão das páginas pelo SQL, quando atingimos a paginação limite. Isso implica diretamente na divisão de metade das linhas em uma página e metade na outra; por um lado a fragmentação dos índices em páginas diferentes dificulta a leitura, mas em cenários de várias cargas simultâneas de dados os índices não sequenciais são mais performáticos.

**Tipos de conversão de dados**

Revisão de conteúdo já visto em capítulos anteriores. Operadores CAST, CONVERT e PARSE. Em CAST nós especificamos o tipo destino de dados; com CONVERT temos que informar o formato (para alguns tipos específicos, exemplo datas); com PARSE especificamos a cultura que desejamos utilizar na conversão, exemplo de datas novamente.

Os três operadores suportam a opção de TRY\_CAST, TRY\_CONVERT, TRY\_PARSE. Com a opção TRY, caso a conversão não seja bem-sucedida é retornado NULL.

**Lidando com NULLs**

No primeiro capítulo já vimos os detalhes do tipo NULL, nesta seção trataremos especificamente das funções ISNULL e COALESCE() e as suas diferenças.

**As funções COALESCE e ISNULL**

As duas funções são utilizadas primordialmente para achar os primeiros valores não nulos em uma serie de inputs. A primeira grande diferença entre ISNULL e COALESCE() é que o primeiro aceita apenas 2 parâmetros de input; o último aceita mais de 2. A query abaixo demonstra o uso:

SET NOCOUNT ON;

USE TSQLV4;

DECLARE

 @x AS INT = NULL,

 @y AS INT = 1759,

 @z AS INT = 42;

SELECT COALESCE(@x, @y, @z);

SELECT ISNULL(@x, @y);

As duas queries retornam o mesmo valor (1759), pois em ambas este é o primeiro valor não nulo.

A função ISNULL é uma propriedade do SQL; a função COALESCE é definida pelo padrão ISO/ANSI, então em um cenário de políticas de conversão dentro de uma empresa, a preferencia de é sempre para a função COALESCE.

Um exemplo de diferença entre as duas funções:

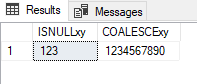
DECLARE

 @x AS VARCHAR(3) = NULL,

 @y AS VARCHAR(10) = '1234567890';

SELECT ISNULL(@x, @y) AS ISNULLxy, COALESCE(@x, @y) AS COALESCExy;

A consulta acima retorna o seguinte:



No primeiro SELECT o primeiro input é do tipo VARCHAR(3) e NULL, quando isso acontece (primeiro input NULL) o tipo do resultado é o tipo do primeiro input. Como o primeiro input é NULL e o segundo não, o SQL converte de forma implícita o segundo input para o tipo do primeiro, VARCHAR(3) e trunca os valores restantes.

Com a função COALESCE o tipo do resultado é o tipo do input com maior precedência, quer dizer, VARCHAR(10) no nosso exemplo, então o valor retornado não é truncado.

Outro exemplo de diferença entre ISNULL e COALESCE pode ser explicitada por essa query:

SELECT ISNULL('1a2b', 1234) AS ISNULLstrnum;

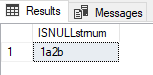
GO

SELECT COALESCE('1a2b', 1234, NULL) AS COALESCEstrnum;

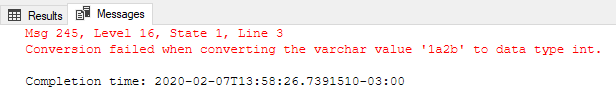
GO

A query acima falha se executada em um único lote.

Rodando separados, como a função ISNULL define o tipo de resultado baseado no primeiro input, ele retorna o primeiro input com o tipo original:



Como a função COALESCE olha para a precedência dos tipos, no caso o tipo de maior precedência é INT, então a função tenta converter o primeiro input para INT e aí ocorre a falha:



Outra diferença entre ISNULL e COALESCE é quando executamos um conjunto SELECT INTO com as duas funções na subquery. Com ISNULL, a coluna resultado é definida como NOT NULLs se um dos inputs tiver essa característica; somente se todos inputs forem ALLOW NULLs então o resultado será NULL.

Com COALESCE() a coluna resultado só é definida como NOT NULL se todos inputs tiverem essa característica, caso contrário a coluna de resultado será ALLOW NULL.

Na prova se existirem perguntas que envolvam ISNULL e COALESCE() provavelmente teremos que escolher entre as duas baseadas nas diferenças entre elas. A tabela abaixo ajuda nisso:

|  |  |  |
| --- | --- | --- |
| **aspect** | **isnull** | **coalesce** |
| Number of supported parameters | 2 | > 2 |
| Standard | No | Yes |
| Data type of result | 1. If first input has a type, that’s the type of result.  2. Otherwise, if second input has a type, that’s the tupe of the result.  3. If both inputs are untyped NULL literals, the result type is INT | 1. If at least one input has a type, the result type is the type with the highest precedence  2. If all inputs are untyped NULL literals, you get an error |
| the type of the result | If any input is nonnullable, result is defined as NOT NULL, otherwise as NULL | If all inputs are nonnullable, result is defined as NOT NULL, otherwise as NULL |
| Might execute subquery more than once | No | Yes |

**Resumo do capítulo**

* Views são expressões de tabela que são reutilizáveis, mas não aceitam parâmetros; funções que retornam tabelas (TBF) são como as views, mas aceitam parâmetros, devemos pensar nas TBF (table valued functions) como **VIEWS COMO PARÂMETROS**
* Funções definidas pelo usuário que retornam valores escalares (User Defined Functions) aceitam parâmetros e retornam um único valor Você não pode mudar dados dados retornados pela função (só alterando a definição dela). T-SQL não dá suporte para tratamento de erros dentro de UDF
* Stored procedures são rotinas reutilizáveis que permitem o encapsulamento da lógica. Diferentemente das funções, stored procedures podem alterar dados no banco de dados, usar SQL dinâmico e aceitam tratamento de erros. Procedures ajudam a mascarar a complexidade das tarefas. Também ajudam com a segurança uma vez que é necessário a declaração EXECUTE, essa declaração pode ser concedida a um grupo especifico de usuários
* SQL Server aceita o trabalho de transações dentro do paradigma ACID (atomicidade, consistência, isolamento e durabilidade)
* T-SQL aceita construção de blocos TRY-CATCH para tratamento de erros. Usando esses blocos de forma apropriada garante que não fique transações abertas em casos de erros, permitindo o tratamento de forma mais consistente
* Usando o XACT\_ABORT ON nós conseguimos resultados mais consistentes em termos de transações quando erros ocorrem. Quando usamos TRY-CATCH, uma transação aberta é revertida (roll back) e o bloco é abortado. Quando usamos TRY-CATCH, a transação fica em estado de *doom* e o controle passa para o bloco CATCH
* As funções ISNULL e COALESCE() retornam o primeiro valor não nulo de uma serie de inputs
* Quando combinamos dados de varias tabelas por meio de joins e subqueries temos que estar atento às possibilidades de valores NULL nas chaves que estamos combinando. Precisamos pensar no resultado da comparação das chaves. Se quisermos combinar comparando NULLs por meio de joins e subqueries precisamos adicionar um tratamento especial, usando um operador (INTERSECT, por exemplo) não é necessário o tratamento especial

1. Some developers argue against implementing logic in database routines like stored procedures. Do you have na arguments in favor of using those?

Encapsulamento da lógica, reutilização do código, segurança ao mascarar a complexibilidade do código.

1. Some developers argue that there’s no need for error handling in T-SQL, rather all error handling should be done in the application. Can you provide any arguments in favor of handling erros in T-SQL, and any recommendations for consistente error handling?

O tratamento de erros usando T-SQL pode garantir mais consistência no tratamento dos erros utilizando TRY-CATCH e XACT\_ABORT ON, dessa maneira, transações com erros não seriam commitadas nunca e sempre seriam revertidas, garantindo segurança nas transações.

1. There’s some inconsistency in the choice of data types for columns that represent the same thing in different tables. Making a change in the type of a column involves some refactoring, and in some cases downtime. What are the benefits and arguments in favor of creating a plan for more consistent use of data types for attributes that represent the same thing?

Por exemplo se utilizamos numeric(18,2) em uma coluna e INT em uma coluna, os números essencialmente representam a mesma coisa, mas inconsistência dos dados pode ser um problema devido ao arredondamento, perdendo informações de uma para a outra. Definir tipos de dados iguais para tabelas/colunas que dizem as mesmas coisas é essencial para garantir a integridade.

1. Currently there’s a lot of inconsistency in using ISNULL and COALESCE. If the dev team had to pick one of the two to be consistent, which of the two should it be? Also, are there special cases where an exception should be made?

COALESCE. COALESCE segue o padrão ISO/ANSI e é mais indicado quando é necessário uma padronização e consistência maior. A exceção de quando temos apenas 2 inputs o ISNULL é mais indicado; também é mais indicado em subqueries pois tem uma performance melhor

1. When joining data with keys that support NULLs, developers regularly use the ISNULL function in both sides of the join to replace a NULL with an alternative value, and this way get a match when both sides are NULL. However, users complain about bad performance for those queries. Can you suggest an alternative solution?

Quando usamos ISNULL ou COALESCE em joins, os dados perdem a sua ordem original. É preferível fazer as operações desta forma

COALESCE(T1.key1, <algumvalor>) = COALESCE(T2.key1, <algumvalor>),

ou ainda

T1.key1 = T2.key1 OR (T1.key1 IS NULL AND T2.key1 IS NULL)