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

  )

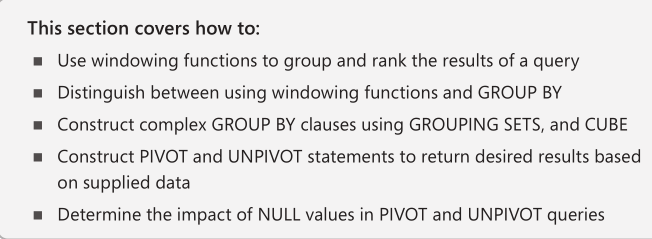
  SELECT empid, mgrid, firstname, lastname, distance

  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:

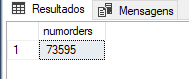
use

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

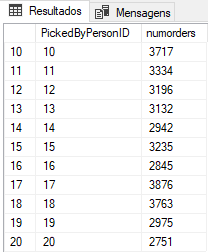
WideWorldImporters

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

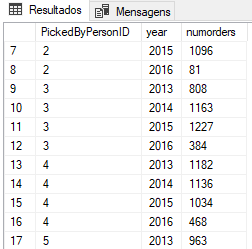
WideWorldImporters

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

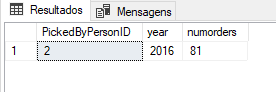
FROM Sales.Orders

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 )

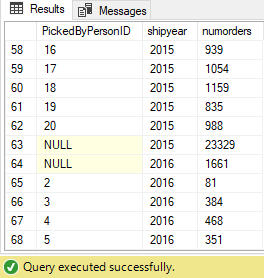
,( year(OrderDate) )

,( )

);

**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

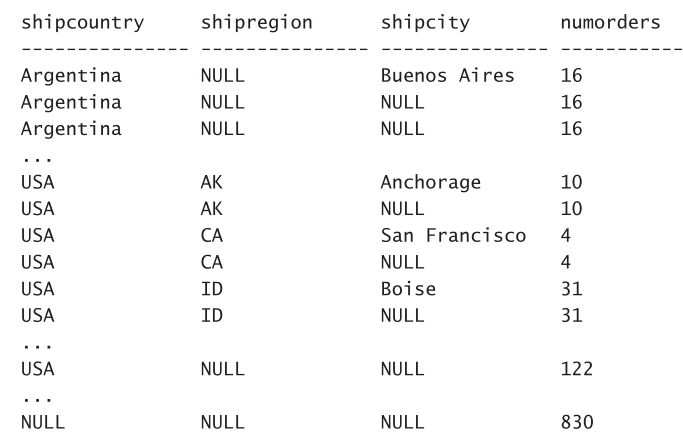
,shipcity

,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

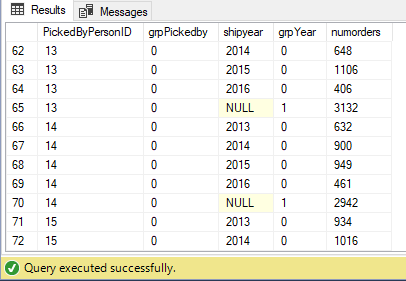
,COUNT(\*) AS numorders

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**

--------

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?