Microsoft

Querying Data with Transact-SQL

Exam Ref 70-761

Itzik Ben-Gan

Microsoft

Exam Ref 70-761 Querying Data with Transact-SQL

Itzik Ben-Gan

Yet another common example is when filtering a NULLable column. Consider the following query:

```
DECLARE @dt AS DATE = '20150212';

SELECT orderid, shippeddate
FROM Sales.Orders
WHERE shippeddate = @dt;
```

There's an index defined on the shippeddate column, with the orderid column implicitly included. Clearly, this filter is sargable, but there is a bug in this query. Unshipped orders are marked with a NULL in the shippeddate column, so when you want to see unshipped orders you pass a NULL as input as follows:

```
DECLARE @dt AS DATE = NULL;

SELECT orderid, shippeddate
FROM Sales.Orders
WHERE shippeddate = @dt;
```

This query uses an equality-based comparison, where a comparison between anything and a NULL, including between two NULLs, yields unknown, and therefore the result is an empty set.

One of the common techniques that people use to cope with such cases is to use the COALESCE or ISNULL function to replace a NULL in both sides with a value that can't normally appear in the data as follows:

```
DECLARE @dt AS DATE = NULL;

SELECT orderid, shippeddate
FROM Sales.Orders
WHERE ISNULL(shippeddate, '99991231') = ISNULL(@dt, '99991231');
```

You do get the correct result, but the filter isn't sargable, so the plan for this query is similar to the one shown earlier in Figure 1-15, only with the current index and query filter predicate.

One recommended solution that is considered sargable is to use the IS NULL predicate to check for NULLs as follows:

```
DECLARE @dt AS DATE = NULL;

SELECT orderid, shippeddate
FROM Sales.Orders
WHERE shippeddate = @dt
   OR (shippeddate IS NULL AND @dt IS NULL);
```

The plan for this query is similar to the one shown earlier in Figure 1-16.

This solution is correct, but if you have a conjunction of multiple predicates based on NULLable columns that you need to filter by, your WHERE clause will end up being long and convoluted. Recall the trick you used in the joins section when comparing NULLable columns using the EXISTS predicate and the INTERSECT set operator. You can apply the same trick here as follows:

```
DECLARE @dt AS DATE = NULL;

SELECT orderid, shippeddate
FROM Sales.Orders
WHERE EXISTS (SELECT shippeddate INTERSECT SELECT @dt);
```

With multiple columns you simply extend the SELECT lists in both sides. Set operators use a distinctness-based comparison and not an equality-based one, giving you the desired meaning without the need for special handling of NULLs. What's more, remarkably this form is sargable and therefore the plan for this query is also similar to the one shown earlier in Figure 1-16.

In the same way that manipulation of a filtered column prevents the sargability of a filter, such manipulation breaks the ordering property of the data. This means that even if there's an index on a column, SQL Server cannot rely on the index order to support an order-based algorithm for presentation ordering, window-function ordering, joining, grouping, distinctness, and so on.

Function determinism

Function determinism is a characteristic that indicates whether the function is guaranteed to return the same result given the same set of input values (including an empty set) in different invocations. If the function provides such a guarantee, it is said to be *deterministic*; otherwise, it is said to be *nondeterministic*. I am not going to go over the full list of functions and their determinism quality here. You can find the details at *https://msdn.microsoft.com/en-us/library/ms178091.aspx*. Here I am going to describe the different categories of determinism and illustrate each with a couple of examples. I also describe the limitations that nondeterministic functions impose.

There are three main categories of function determinism. There are functions that are always deterministic, those that are deterministic when invoked in a certain way, and those that are always nondeterministic.

Examples for functions that are always deterministic: all string functions, COALESCE, ISNULL, ABS, SQRT and many others. For instance, the expression ABS(-1759) always returns 1759.

Certain functions are either deterministic or not depending on how they're used. For example, the CAST function is not deterministic when converting from a character string to a date and time type or the other way around because the interpretation of the value might depend on the login's language. For instance, the expression CAST('02/12/17' AS DATE)

converts to February 12, 2017 under us_english, December 2, 2017 under British, and December 17, 2002 under Swedish and Japanese. The same applies to CONVERT when using certain styles. Another example is the RAND function. This function returns a float in the range 0 through 1. When using it with a seed, it is deterministic. For instance, run the following code several times:

```
SELECT RAND(1759);
```

You keep getting the same result, 0.746348756684839.

When you invoke the function without a seed, SQL Server computes a new seed based on the previous invocation and you get a pseudo random value. Run the following code several times and notice that you keep getting different results:

```
SELECT RAND();
```

In *pseudo*, I mean that even though technically without a seed the function is considered nondeterministic, if invoked right after an invocation with a seed, the result is repeatable. Run the following code several times:

```
SELECT RAND(1759);
SELECT RAND();
```

You repeatedly get the following output:

```
----------------------
0.746348756684839

----------------------
0.201391138037653
```

Certain functions are always nondeterministic, for example SYSDATETIME and NEWID. The former returns the current date and time value as a DATETIME2 typed value, and the latter returns a globally unique identifier as a UNIQUEIDENTIFIER typed value. NEWID returns a fairly random value, but its type is awkward to work with. To get a random integer value in a certain range, for instance, 1 through 10, use the following expression:

```
SELECT 1 + ABS(CHECKSUM(NEWID())) % 10;
```

By applying CHECKSUM to the result of NEWID you get a random integer. The absolute value modulo 10 gives you a random value in the range 0 through 9. Adding the result to 1 gives you a random value in the range 1 through 10.

Most nondeterministic functions are invoked once per query. That's the case for instance with SYSDATETIME and RAND (when invoked without a seed). The NEWID function is an exception in the sense that it gets invoked per row. Consider the following query:

```
SELECT empid, SYSDATETIME() AS dtnow, RAND() AS rnd, NEWID() AS newguid
FROM HR.Employees;
```

In one of the executions of this query on my system I got the following result (formatted as two outputs to fit on the page).

```
empid  dtnow                       rnd
------ --------------------------- ------------------
2      2016-10-02 09:35:46.8024874 0.980769010450262
7      2016-10-02 09:35:46.8024874 0.980769010450262
1      2016-10-02 09:35:46.8024874 0.980769010450262
5      2016-10-02 09:35:46.8024874 0.980769010450262
6      2016-10-02 09:35:46.8024874 0.980769010450262
8      2016-10-02 09:35:46.8024874 0.980769010450262
3      2016-10-02 09:35:46.8024874 0.980769010450262
9      2016-10-02 09:35:46.8024874 0.980769010450262
4      2016-10-02 09:35:46.8024874 0.980769010450262

empid  newguid
------ ------------------------------------
2      F2EC6CC7-E986-4A43-9ED9-1A08C56600D2
7      74A018CF-5E95-4C7F-BA8F-D707A3DD5177
1      B894EAAF-07C8-4CC1-AD20-4CE7DA4AFB81
5      10B86C1F-BD18-4E7E-8A70-195A239B54EA
6      20514C7A-4F3C-4C3F-BFBB-014E5FAC6B85
8      86C29355-16B2-4A48-9068-039ABBD56B85
3      D923AEBB-0FD7-424D-93C2-E68E199A24C1
9      E2940155-3C66-4F1A-BBF9-0D001AF9A4AB
4      2B3BE2DD-00CE-494E-B52C-989CA71A36E9
```

Keep this in mind, for instance, if you want to return the results ordered randomly, or select a random set of rows. For instance, suppose that you need to return a random set of three employees, and you use the following query in attempt to achieve this:

```
SELECT TOP (3) empid, firstname, lastname
FROM HR.Employees
ORDER BY RAND();
```

Run this code repeatedly and you probably keep getting the same result. Because the RAND function returns the same value in all rows, the ORDER BY is meaningless here. To get different random values in the different rows, order by NEWID, or for even better random distribution, apply CHECKSUM to NEWID as follows:

```
SELECT TOP (3) empid, firstname, lastname
FROM HR.Employees
ORDER BY CHECKSUM(NEWID());
```

Note that the use of a nondeterministic function in a computed column prevents the ability to create an index on the column. Similarly, the use of a nondeterministic function in a view prevents the ability to create a clustered index on the view. That's the case whether the function is always nondeterministic, or nondeterministic in certain cases.

Skill 1.4: Modify data

The T-SQL support for data manipulation language (DML) includes both statements that retrieve data (SELECT) and statements that modify data (INSERT, UPDATE, DELETE, TRUNCATE TABLE, and MERGE). The previous skills focused on data retrieval; this skill focuses on data modification.

This section covers how to:

- Write INSERT, UPDATE, and DELETE statements
- Determine which statements can be used to load data to a table based on its structure and constraints
- Construct Data Manipulation Language (DML) statements using the OUTPUT statement
- Determine the results of Data Definition Language (DDL) statements on supplied tables and data

Inserting data

T-SQL supports a number of different methods that you can use to insert data into your tables. Those include statements like INSERT VALUES, INSERT SELECT, INSERT EXEC, and SELECT INTO. This section covers these statements and demonstrates how to use them through examples.

Some of the code examples in this section use a table called Sales.MyOrders. Use the following code to create such a table in the sample database TSQLV4:

```
USE TSQLV4;
DROP TABLE IF EXISTS Sales.MyOrders;
GO

CREATE TABLE Sales.MyOrders
(
  orderid INT NOT NULL IDENTITY(1, 1)
    CONSTRAINT PK_MyOrders_orderid PRIMARY KEY,
  custid  INT NOT NULL,
  empid   INT NOT NULL,
  orderdate DATE NOT NULL
    CONSTRAINT DFT_MyOrders_orderdate DEFAULT (CAST(SYSDATETIME() AS DATE)),
  shipcountry NVARCHAR(15) NOT NULL,
  freight MONEY NOT NULL
);
```

Observe that the orderid column has an identity property defined with a seed 1 and an increment 1. This property generates the values in this column automatically when rows are inserted. As an alternative to the identity property you can use a sequence object to gener-

ate surrogate keys. For details about the sequence object and a comparison between the two options, see the following articles:

- Sequences part 1 at *http://sqlmag.com/sql-server/sequences-part-1*
- Sequences part 2 at *http://sqlmag.com/sql-server/sequences-part-2*
- Sequence and identity performance at *http://sqlmag.com/sql-server/sequence-and-identity-performance*

Also observe that the orderdate column has a default constraint with an expression that returns the current system's date.

INSERT VALUES

With the INSERT VALUES statement, you can insert one or more rows into a target table based on value expressions. Here's an example for a statement inserting one row into the Sales.MyOrderValues table:

```
INSERT INTO Sales.MyOrders(custid, empid, orderdate, shipcountry, freight)
  VALUES(2, 19, '20170620', N'USA', 30.00);
```

Specifying the target column names after the table name is optional but considered a best practice. That's because it enables you to control the source value to target column association, irrespective of the order in which the columns were defined in the table.

Without the target column list, you must specify the values in column definition order. If the underlying table definition changes but the INSERT statements aren't modified accordingly, this can result in either errors, or worse, values written to the wrong columns.

The INSERT VALUES statement does not specify a value for a column with an identity property because the property generates the value for the column automatically. Observe that the previous statement doesn't specify the orderid column. If you do want to provide your own value instead of letting the identity property do it for you, you need to first turn on a session option called IDENTITY_INSERT, as follows:

```
SET IDENTITY_INSERT <table> ON;
```

When you're done, you need to remember to turn it off.

Note that in order to use this option, you need quite strong permissions; you need to be the owner of the table or have ALTER permissions on the table.

Besides using the identity property, there are other ways for a column to get its value automatically in an INSERT statement. A column can have a default constraint associated with it like the orderdate column in the Sales.MyOrders table. If the INSERT statement doesn't specify a value for the column explicitly, SQL Server will use the default expression to generate that value. For example, the following statement doesn't specify a value for orderdate, and therefore SQL Server uses the default expression:

```
INSERT INTO Sales.MyOrders(custid, empid, shipcountry, freight)
  VALUES(3, 11, N'USA', 10.00);
```