

T-SQL supports two “not equal to” operators: <> and !=. The former is standard and the latter is not. T-SQL supports multiple functions that convert a source value to a target type. Among them are the CAST and CONVERT functions. The former is standard

and the latter isn’t. The nonstandard CONVERT function has a style argument that CAST doesn’t support. Because CAST is standard, you should consider it your default choice for conversions. You should consider using CONVERT only when you need to rely on the style

argument. Yet another example of choosing the standard form is in the termination of T-SQL statements. According to standard SQL, you should terminate your statements with a semicolon. T-SQL currently doesn’t make this a requirement for all statements, only in cases where there would otherwise be ambiguity of code elements, such as in the WITH clause of a common table expression (CTE). You should still follow the standard and terminate all of your statements even where it is currently not required.

A relation in the relational model is what SQL calls a table. The two are not synonymous. A predicate is an expression that when attributed to some object, makes a proposition either true or false. For example, “salary greater than $50,000” is a predicate. You can evaluate

this predicate for a specific employee, in which case you have a proposition. For example, suppose that for a particular employee, the salary is $60,000. When you evaluate the proposition for that employee, you get a true proposition. In other words, a predicate is a parameterized proposition.

Remember that a relation has a heading and a body. The heading is a set of attributes and the body is a set of tuples. Remember from the definition of a set that a set is supposed to be considered as a whole. What this translates to in T-SQL is that you’re supposed to write queries that interact with the tables as a whole. You should try to avoid using iterative constructs like cursors and loops that iterate through the rows one at a time. You should also try to avoid thinking in iterative terms because this kind of thinking is what leads to iterative solutions. For people with a procedural programming background, the natural way to interact with data (in a file, record set, or data reader) is with iterations. So using cursors and other iterative constructs in T-SQL is, in a way, an extension to what they already know. However, the correct way from the relational model’s perspective is not to interact with the rows one at a time; rather, use relational operations and return a relational result. This, in T-SQL, translates to writing queries.

When you issue a query with SELECT \*, you are guaranteed to get the columns in the result based on definition order. Also, T-SQL allows referring to ordinal positions of columns from the result in the ORDER BY clause, as follows.

SELECT empid, lastname

FROM HR.Employees

ORDER BY 1;

Tatt: the order of rows is not guaranteed but order of columns is.

T-SQL has another deviation from the relational model in that it allows defining result columns based on an expression without assigning a name to the target column. For example, the following query is valid in T-SQL.

SELECT empid, firstname + ' ' + lastname

FROM HR.Employees;

T-SQL allows a SELECT list that looks like the following.

SELECT T1.keycol, T2.keycol ...

For the result to be relational, all attributes must have unique names, so you would need to use different aliases for the result attributes, as in the following.

SELECT T1.keycol AS key1, T2.keycol AS key2 ...

T-SQL attempts to represent a relation with a table, a tuple with a row, and an attribute with a column;

Can you identify what the nonrelational aspects of the query are? Answer: The query doesn’t alias the expression YEAR(orderdate), so there’s no name for the result attribute. The query can return duplicates(but set should not have duplicates). The query forces certain presentation ordering to the result and uses ordinal positions in the ORDER BY clause( but set should not have any ordering).

SELECT custid, YEAR(orderdate)

FROM Sales.Orders

ORDER BY 1, 2;

Removing the non-relational aspects from the query you get:

SELECT distinct custid, YEAR(orderdate) AS ORderYear

FROM Sales.Orders;

Following are the main query clauses specified in the order that you are supposed to type them (known as “keyed-in order”):

1. SELECT

2. FROM

3. WHERE

4. GROUP BY

5. HAVING

6. ORDER BY

Here is the logical query processingorder:

1. FROM

2. WHERE

3. GROUP BY

4. HAVING

5. SELECT

1. Evaluate expressions

2. DISTINCT

6. ORDER BY

7. TOP/OFFSET-FETCH

Consider the following query as an example.

SELECT country, YEAR(hiredate) AS yearhired, COUNT(\*) AS numemployees

FROM HR.Employees

WHERE hiredate >= '20030101'

GROUP BY country, YEAR(hiredate)

HAVING COUNT(\*) > 1

ORDER BY country , yearhired DESC;

This query is issued against the HR.Employees table. It filters only employees that were hired in or after the year 2003. It groups the remaining employees by country and the hire year. It keeps only groups with more than one employee. For each qualifying group, the

query returns the hire year and count of employees, sorted by country and hire year, in descending order.

Attempting to refer in the WHERE clause to a column alias defined in the SELECT clause:

SELECT country, YEAR(hiredate) AS yearhired

FROM HR.Employees WHERE yearhired >= 2003;

This query fails with the following error.

Msg 207, Level 16, State 1, Line 3

Invalid column name 'yearhired'.

Group Phase: All expressions processed in subsequent phases must guarantee a single value per group. If you refer to an element from the GROUP BY list (for example, country), you already have such a guarantee, so such a reference is allowed. However, if you want to

refer to an element that is not part of your GROUP BY list (for example, empid), it must be contained within an aggregate function like MAX or SUM or COUNT etc.

The WHERE clause is evaluated before rows are grouped, and therefore is evaluated per row. The HAVING clause is evaluated after rows are grouped, and therefore is evaluated per group.

According to the relational theory, a set should be unordered, the rows distinct (no duplicates in result set), every column has distinct name.

Select Alias is only available in order by phase (which comes last after select).Alias created by the SELECT phase isn’t even visible to other expressions that appear in the same SELECT list. For example, the following query isn’t valid.

SELECT empid, country, YEAR(hiredate) AS yearhired, yearhired - 1 AS prevyear

FROM HR.Employees;

This query generates the following error.

Msg 207, Level 16, State 1, Line 1

Invalid column name 'yearhired'.

The reason that this isn’t allowed is that, conceptually, T-SQL evaluates all expressions that appear in the same logical query processing phase in an all-at-once manner. Note the use of the word conceptually. SQL Server won’t necessarily physically process all expressions at the same point in time, but it has to produce a result as if it did. This behavior is different than many other programming languages where expressions usually get evaluated in a left-toright order, making a result produced in one expression visible to the one that appears to its

right. But T-SQL is different.

A query may specify the TOP or OFFSET-FETCH filtering options. If it does, the same ORDER BY clause that is normally used to define presentation ordering also defines which rows to filter for these options. Tatt: TOP or OFFSET-FETCH need not necessarily be used with ORDER BY??? Even though they only make sense when used with ORDER BY as otherwise the order of rows in result set is not guaranteed. So doing a TOP 10 might result different result sets each time.

Tatt: DISTINCT clause has to appear before the first column is referred to in a SELECT list. So you can’t do SELECT E.Name, DISTINCT E.Age…; ????

Tatt: predicate is the condition and proposition is the instantiation of the condition. Means, proposition refers to the rows which satisfy that predicate.

In the FROM clause, you can alias the queried tables with your chosen names. You can use the form <table> <alias>, as in HR.Employees E, or <table> AS <alias>, as in HR.Employees AS E. The latter form is more readable.

Note that if you assign an alias to a table, you basically rename the table for the duration of the query. The original table name isn’t visible anymore; only the alias is. So the following won’t work:

SELECT H.firstname, HR.Employees.lastname

FROM HR.Employees AS H

But this will work:

SELECT H.firstname, lastname

FROM HR.Employees AS H

Normally, you can prefix a column name you refer to in a query with the table name, as in Employees.empid.However, if you aliased the Employees table as E, the reference Employees.empid is invalid; you have to use E.empid or just empid.

Using an \* is just a matter of laziness. You send more data than is needed over the network, and this can have a negative impact on the system’s performance.

In the SELECT clause, you can assign your own aliases to the expressions that define the result attributes. There are a number of supported forms of aliasing: <expression> AS <alias> as in empid AS employeeid, <expression> <alias> as in empid employeeid, and <alias> = <expression> as in employeeid = empid. Use the first form with the AS clause because it’s both standard and we find it to be the most readable.

A result with duplicates is considered nonrelational because relations—being sets—are not supposed to have duplicates. Therefore,

if duplicates are possible in the result, and you want to eliminate them in order to return a relational result, you can do so by adding a DISTINCT clause, as in the following.

SELECT DISTINCT country, region, city FROM HR.Employees;

There’s an interesting difference between standard SQL and T-SQL in terms of minimalSELECT query requirements. According to standard SQL, a SELECT query must have at minimum FROM and SELECT clauses. Conversely, T-SQL supports a SELECT query with only a SELECT clause and without a FROM clause. Such a query is as if issued against an imaginary table that has only one row. For example, the following query is invalid according to standard SQL but is valid according to T-SQL.

SELECT 10 AS col1, 'ABC' AS col2;

The output of this query is a single row with attributes resulting from the expressions with

names assigned using the aliases.

col1 col2

----------- ----

10 ABC

Tatt: Remember SELECT 1 can be used to ping SQL server to test network performance. So that is non-standard SQL.

T-SQL supports both a standard form to delimit identifiers using double quotation marks, as in "Sales"."Orders", as well as a proprietary form using square brackets, as in [Sales].[Orders]. For example, an attribute called 2006 is considered an irregular identifier because it starts with a digit, and therefore must be delimited as "2006" or [2006].

When defining columns in tables, parameters in procedures and functions, and variables in T-SQL batches, you need to choose a data type for those. Much like a type is a constraint, NOT NULL is a constraint as well. If an attribute isn’t supposed to allow NULLs, it’s important to enforce a NOT NULL constraint as part of its definition.

Suppose you have an attribute representing test scores, which are integers in the range 0 to 100. Using an INT data type for this purpose is overkill. It would use 4 bytes per value, whereas a TINYINT would use only 1 byte, and is therefore the more appropriate type in this case. If the value is supposed to represent both date and time, you should consider DATETIME2 or SMALLDATETIME. The former requires storage between 6 to 8 bytes (depending on precision), and as an added value, provides a wider range of dates and improved, controllable precision. Be very careful with the imprecise types FLOAT and REAL.

Another important aspect in choosing a type has to do with choosing fixed types (CHAR, NCHAR, BINARY) vs. dynamic ones (VARCHAR, NVARCHAR, VARBINARY). Fixed types use the storage for the indicated size; for example, CHAR(30) uses storage for 30 characters, whether you actually specify 30 characters or less. This means that updates will not require the row to physically expand, and therefore no data shifting is required. So for attributes that get updated frequently, where the update performance is a priority, you should consider fixed types. Note that when compression is used—specifically row compression—SQL Server stores fixed types like variable ones, but with less overhead.

Variable types use the storage for what you enter, plus a couple of bytes for offset information (or 4 bits with row compression). So for widely varying sizes of strings, if you use variable types you can save a lot of storage. As already mentioned, the less storage used, the less there is for a query to read, and the faster the query can perform. So variable length types are usually preferable in such cases when read performance is a priority.

With character strings, there’s also the question of using regular character types (CHAR, VARCHAR) vs. Unicode types (NCHAR, NVARCHAR). The former use 1 byte of storage per character and support only one language (based on collation properties) besides English. The latter use 2 bytes of storage per character (unless compressed) and support multiple languages. When using types that can have a length associated with them, such as CHAR and VARCHAR, T-SQL supports omitting the length and then uses a default length. However, in different contexts, the defaults can be different. It is considered a best practice to always be explicit about the length, as in CHAR(1) or VARCHAR(30).

literals of regular character strings are delimited with single quotation marks, as in 'abc', whereas literals of Unicode character strings are delimited with a capital N and then single quotation marks, as in N'abc'. When an expression involves elements with different types, SQL Server needs to apply implicit conversion when possible, and this may result in performance penalties. In order to force a literal to be of a certain type, you may need to apply explicit conversion with functions like CAST, CONVERT, PARSE, or TRY\_CAST, TRY\_CONVERT, and TRY\_PARSE. As an example, the literal 1 is considered an INT by SQL Server in any context. If you need the literal 1 to be considered, for example, a BIT, you need to convert the literal’s type explicitly, as in CAST(1 AS BIT). Similarly, the literal 4000000000 is considered NUMERIC and not BIGINT. If you need the literal to be the latter, use CAST(4000000000 AS BIGINT). The difference between the functions without the TRY and their counterparts with the TRY is that those without the TRY fail if the value isn’t convertible, whereas those with the TRY return a NULL in such a case. For example, the following code fails.

SELECT CAST('abc' AS INT);

Conversely, the following code returns a NULL.

SELECT TRY\_CAST('abc' AS INT);

As for the difference between CAST, CONVERT, and PARSE, with CAST, you indicate the expression and the target type; with CONVERT, there’s a third argument representing the style for the conversion, which is supported for some conversions, like between character strings and date and time values. For example, CONVERT(DATE, '1/2/2012', 101) converts the literal character string to DATE using style 101 representing the United States standard. With PARSE, you can indicate the culture by using any culture supported by the Microsoft .NET Framework. For example, PARSE('1/2/2012' AS DATE USING 'en-US') parses the input literal as a DATE by using a United States English culture.

Tatt: cast is the standard not convert.

When using expressions that involve operands of different types, SQL Server usually converts the one that has the lower data type precedence to the one with the higher. Consider the expression 1 + '1' as an example. One operand is INT and the other is VARCHAR. If you look in Books Online for SQL Server 2012, under “Data Type Precedence (Transact-SQL),” at *http://msdn.microsoft.com/en-us/library/ms190309.aspx*, you will find that INT precedes VARCHAR; hence, SQL Server implicitly converts the VARCHAR value '1' to the INT value 1, and the result of the expression is therefore 2 and not the string '11'. Of course, you can always take control by using explicit conversion. If all operands of the expression are of the same type, that’s also going to be the type of the result, and you might not want it to be the case. For example, the result of the expression 5 / 2 in T-SQL is the INT value 2 and not the NUMERIC value 2.5, because both operands are integers, and therefore the result is an integer. If you were dealing with two integer columns, like col1 / col2, and wanted the division to be NUMERIC, you would need to convert the columns explicitly, as in CAST(col1 AS NUMERIC(12, 2)) / CAST(col2 AS NUMERIC(12, 2)).

The typical options people use to generate surrogate keys are:

■■ **The identity column property** A property that automatically generates keys in an attribute of a numeric type with a scale of 0; namely, any integer type (TINYINT, SMALLINT, INT, BIGINT) or NUMERIC/DECIMAL with a scale of 0.

■■ **The sequence object** An independent object in the database from which you can obtain new sequence values. Like identity, it supports any numeric type with a scale of 0. Unlike identity, it’s not tied to a particular column; instead, as mentioned, it is an independent object in the database. You can also request a new value from a sequence object before using it. There are a number of other advantages over identity that will be covered in Chapter 11.

■■ **Nonsequential GUI Ds** You can generate nonsequential global unique identifiers to be stored in an attribute of a UNIQUEIDENTIFIER type. You can use the T-SQL function NEWID to generate a new GUID, possibly invoking it with a default expression attached to the column. You can also generate one from anywhere—for example, the client— by using an application programming interface (API) that generates a new GUID. The GUIDs are guaranteed to be unique across space and time.

■■ **Sequential GUI Ds** You can generate sequential GUIDs within the machine by using the T-SQL function NEWSEQUENTIALID.

■■ **Custom solutions** If you do not want to use the built-in tools that SQL Server provides to generate keys, you need to develop your own custom solution. The data type for the key then depends on your solution

TODO: chapter 2, starting around page 40, it goes into effects of types of keys on indexing details which need to be reread.

One important category of functions is the category that returns the current date and time. The functions in this category are GETDATE, CURRENT\_TIMESTAMP, GETUTCDATE, SYSDATETIME, SYSUTCDATETIME, and SYSDATETIMEOFFSET.

GETDATE is T-SQL–specific, returning the current date and time in the SQL Server instance you’re connected to as a DATETIME data type**. CURRENT\_TIMESTAMP is the same, only it’s standard, and hence the recommended one to use**. SYSDATETIME and SYSDATETIMEOFFSET are similar, only returning the values as the more precise DATETIME2 and DATETIMEOFFSET types (including offset), respectively. Note that there are no built-in functions to return the current date or the current time; to get such information, simply cast the SYSDATETIME function to DATE or TIME, respectively. For example, to get the current date, use CAST(SYSDATETIME() AS DATE). The GETUTCDATE function returns the current date and time in UTC terms as a DATETIME type, and SYSUTCDATE does the same, only returning the result as the more precise DATETIME2 type.

Using the DATEPART function, you can extract from an input date and time value a desired part, such as a year, minute, or nanosecond, and return the extracted part as an integer. For example, the expression DATEPART(month, '20120212') returns 2. T-SQL provides the functions YEAR, MONTH, and DAY as abbreviations to DATEPART, not requiring you to specify the part. The DATENAME function is similar to DATEPART, only it returns the name of the part as a character string, as opposed to the integer value. Note that the function is languagedependent. That is, if the effective language in your session is us\_english, the expression DATENAME(month, '20120212') returns 'February', but for Italian, it returns 'febbraio'.

T-SQL provides a set of functions that construct a desired date and time value from its numeric parts. You have such a function for each of the six available date and time types: DATEFROMPARTS, DATETIME2FROMPARTS, DATETIMEFROMPARTS, DATETIMEOFFSETFROMPARTS, SMALLDATETIMEFROMPARTS, and TIMEFROMPARTS. For example, to build a DATE value from

its parts, you would use an expression such as DATEFROMPARTS(2012, 02, 12). Finally, the EOMONTH function computes the respective end of month date for the input date and time value. For example, suppose that today was February 12, 2012. The expression

EOMONTH(SYSDATETIME()) would then return the date '2012-02-29'. This function supports a second optional input indicating how many months to add to the result.

DATEADD is a very commonly used function. With it, you can add a requested number of units of a specified part to a specified date and time value. For example, the expression DATEADD(year, 1, '20120212') adds one year to the input date February 12, 2012. DATEDIFF is another commonly used function; it returns the difference in terms of a requested part between two date and time values. For example, the expression DATEDIFF(day,'20110212', '20120212') computes the difference in days between February 12, 2011 and February 12, 2012, returning the value 365. Note that this function looks only at the parts from the requested one and above in the date and time hierarchy—not below. For example, the expression DATEDIFF(year, '20111231', '20120101') looks only at the year part, and hence returns 1. It doesn’t look at the month and day parts of the values.

T-SQL supports two functions related to date and time values with an offset: SWITCHOFFSET and TODATETIMEOFFSET.

With the SWITCHOFFSET function, you can return an input DATETIMEOFFSET value in a requested offset term. For example, consider the expression SWITCHOFFSET(SYSDATETIMEOF FSET(), '-08:00'). Regardless of the offset of the instance you are connected to, you request to present the current date and time value in terms of offset '-08:00'. If the system’s offset is, say, '-05:00', the function will compensate for this by subtracting three hours from the input value. The TODATETIMEOFFSET function is used for a different purpose. You use it to construct a DATETIMEOFFSET value from two inputs: the first is a date and time value that is not offsetaware, and the second is the offset. You can use this function when migrating from data that is not offset-aware, where you keep the local date and time value in one attribute, and the offset in another, to offset-aware data. Say you have the local date and time in an attribute called dt, and the offset in an attribute called theoffset. You add an attribute called dto of a DATETIMEOFFSET type to the table. You then update the new attribute to the expression TODATETIMEOFFSET(dt, theoffset), and then drop the original attributes dt and theoffset from the table.

The following code demonstrates using both functions.

SELECT

SWITCHOFFSET('20130212 14:00:00.0000000 -08:00', '-05:00') AS [SWITCHOFFSET],

TODATETIMEOFFSET('20130212 14:00:00.0000000', '-08:00') AS [TODATETIMEOFFSET];

Here’s the output of this code.

SWITCHOFFSET TODATETIMEOFFSET

---------------------------------- ----------------------------------

2013-02-12 17:00:00.0000000 -05:00 2013-02-12 14:00:00.0000000 -08:00

T-SQL supports two ways to concatenate strings—one with the plus (+) operator, and another with the CONCAT function. Here’s an example for concatenating strings in a query by using the + operator.

SELECT empid, country, region, city,

country + N',' + region + N',' + city AS location

FROM HR.Employees;

Observe that when any of the inputs is NULL, the + operator returns a NULL. That’s standard behavior that can be changed by turning off a session option called CONCAT\_NULL\_YIELDS\_NULL\_INPUT, though it’s not recommended to rely on nonstandard behavior. If you

want to substitute a NULL with an empty string, there are a number of ways for you to do this programmatically. One option is to use COALESCE(<expression>, ''). For example, in this data, only region can be NULL, so you can use the following query to replace a comma plus region with an empty string when region is NULL.

SELECT empid, country, region, city,

country + COALESCE( N',' + region, N'') + N',' + city AS location

FROM HR.Employees;

Another option is to use the CONCAT function which, unlike the + operator, substitutes a NULL input with an empty string. Here’s how the query looks.

SELECT empid, country, region, city,

CONCAT(country, N',' + region, N',' + city) AS location

FROM HR.Employees;

the expression SUBSTRING('abcde', 1, 3) returns 'abc'. LEFT('abcde', 3) returns 'abc' and RIGHT('abcde', 3) returns 'cde'. CHARINDEX(' ','Itzik Ben-Gan') looks for the first occurrence of a space in the second input, returning 6 in this example. LEFT(fullname, CHARINDEX(' ', fullname) - 1)

But whereas with CHARINDEX you’re looking for a constant string, with PATINDEX you’re looking for a pattern. expression PATINDEX('%[0-9]%', 'abcd123efgh') looks for the first occurrence of a digit (a character in the range 0–9) in the second input, returning the position 6 in this case. Tatt: I think it should have been 5

T-SQL provides two functions that you can use to measure the length of an input value— LEN and DATALENGTH.The LEN function returns the length of an input string in terms of the number of characters. Note that it returns the number of characters, not bytes, whether the input is a regular character or Unicode character string. For example, the expression LEN(N'xyz') returns 3. If there are any trailing spaces, LEN removes them. The DATALENGTH function returns the length of the input in terms of number of bytes. This means, for example, that if the input is a Unicode character string, it will count 2 bytes per character. For example, the expression DATALENGTH(N'xyz') returns 6. Note also that, unlike LEN, the DATALENGTH function doesn’t remove trailing spaces

T-SQL supports a number of functions that you can use to apply alterations to an input string. Those are REPLACE, REPLICATE, and STUFF. the expression REPLACE('.1.2.3.', '.', '/') substitutes all occurrences of a dot (.) with a slash (/), returning the string '/1/2/3/'. the expression REPLICATE('0', 10) replicates the string '0' ten times, returning '0000000000'. the expression STUFF(',x,y,z', 1, 1, '') removes the first character from the input string, returning 'x,y,z'.

Apply formatting options to an input string using UPPER, LOWER, LTRIM, RTRIM, and FORMAT functions. The expression FORMAT(

1759, '000000000') formats the input number as a character string with a fixed size of 10 characters with leading zeros, returning '0000001759'.

The CASE expression has two forms—the *simple* form and the *searched* form. Here’s an example of the simple CASE form issued against the sample database TSQL2012.

SELECT productid, productname, unitprice, discontinued,

CASE discontinued

WHEN 0 THEN 'No'

WHEN 1 THEN 'Yes'

ELSE 'Unknown'

END AS discontinued\_desc

FROM Production.Products;

The simple form compares an *input expression* (in this case the attribute discontinued) to multiple possible scalar *when expressions* (in this case, 0 and 1), and returns the *result expression* (in this case, 'No' and 'Yes', respectively) associated with the first match. If there’s no match and an ELSE clause is specified, the *else expression* (in this case, 'Unknown') is returned. If there’s no ELSE clause, the default is ELSE NULL.

The searched form of the CASE expression is more flexible. Instead of comparing an input expression to multiple possible expressions, it uses predicates in the WHEN clauses, and the first predicate that evaluates to true determines which when expression is returned. If none is true, the CASE expression returns the else expression. Here’s an example.

SELECT productid, productname, unitprice,

CASE

WHEN unitprice < 20.00 THEN 'Low'

WHEN unitprice < 40.00 THEN 'Medium'

WHEN unitprice >= 40.00 THEN 'High'

ELSE 'Unknown'

END AS pricerange

FROM Production.Products;

T-SQL supports a number of functions that can be considered as abbreviates of the CASE expression. **Those are the standard COALESCE and NULLIF functions, and the nonstandard ISNULL, IIF, and CHOOSE**.

The COALESCE function accepts a list of expressions as input and returns the first that is not NULL, or NULL if all are NULLs. For example, the expression COALESCE(NULL, 'x', 'y') returns 'x'. More generally, the expression:

COALESCE(<exp1>, <exp2>, …, <expn>)

is similar to the following.

CASE

WHEN <exp1> IS NOT NULL THEN <exp1>

WHEN <exp2> IS NOT NULL THEN <exp2>

…

WHEN <expn> IS NOT NULL THEN <expn>

ELSE NULL

END

A typical use of COALESCE is to substitute a NULL with something else. For example, the expression COALESCE(region, '') returns region if it’s not NULL and returns an empty string if it is NULL. T-SQL supports a nonstandard function called ISNULL that is similar to the standard COALESCE, but it’s a bit more limited in the sense that it supports only two inputs. Like COALESCE, it returns the first input that is not NULL. So, instead of COALESCE(region, ''), you could use ISNULL(region, ''). **But we should stick to COALESCE as ISNULL is nonstandard while** **COALESCE is standard**.

There are a couple of subtle differences between COALESCE and ISNULL that you might be interested in. One difference is in which input determines the type of the output. Consider the following code.

DECLARE

@x AS VARCHAR(3) = NULL,

@y AS VARCHAR(10) = '1234567890';

SELECT COALESCE(@x, @y) AS [COALESCE], ISNULL(@x, @y) AS [ISNULL];

Here’s the output of this code.

COALESCE ISNULL

---------- ------

1234567890 123

Observe that the type of the COALESCE expression is determined by the returned element, whereas the type of the ISNULL expression is determined by the first input. Tatt: this is not very logical behaviour. Should stick to coalesce.

The other difference between COALESCE and ISNULL is when you are using SELECT INTO, which is discussed in more detail in Chapter 11. Suppose the SELECT list of a SELECT INTO statement contains the expressions COALESCE(col1, 0) AS newcol1 vs. ISNULL(col1, 0) AS newcol1. If the source attribute col1 is defined as NOT NULL, both expressions will produce an attribute in the result table defined as NOT NULL. However, if the source attribute col1 is defined as allowing NULLs, COALESCE will create a result attribute allowing NULLs, whereas ISNULL will create one that disallows NULLs.

COALESCE and ISNULL can impact performance when you are combining sets; for example, with joins or when you are filtering data. Consider an example where you have two tables T1 and T2 and you need to join them based on a match between T1.col1 and T2.col1. The attributes do allow NULLs. Normally, a comparison between two NULLs yields unknown, and this causes the row to be discarded. You want to treat two NULLs as equal. What some do in such a case is use COALESCE or ISNULL to substitute a NULL with a value that they know cannot appear in the data. For example, if the attributes are integers, and you know that you have only positive integers in your data (you can even have constraints that

ensure this), you might try to use the predicate COALESCE(T1.col1, -1) = COALESCE(T2. col1, -1), or ISNULL(T1.col1, -1) = ISNULL(T2.col1, -1). The problem with this form is that, because you apply manipulation to the attributes you’re comparing, SQL Server will not rely on index ordering. This can result in not using available indexes efficiently. Instead, it is recommended to use the longer form: T1.col1 = T2.col1 OR (T1.col1 IS NULL AND T2.col1 IS NULL), which SQL Server understands as just a comparison that considers NULLs as equal. With this form, SQL Server can efficiently use indexing.

**T-SQL also supports the standard NULLIF function**. Expression NULLIF(col1, col2): If col1 is equal to col2, the function returns a

NULL; otherwise, it returns the col1 value. As for IIF and CHOOSE, these are nonstandard T-SQL functions that were added to simplify

migrations from Microsoft Access platforms. Because these functions aren’t standard and there are simple standard alternatives with CASE expressions, it is not usually recommended that you use them.

With the IIF function, you can return one value if an input predicate is true and another value otherwise. The function has the following form.

IIF(<predicate>, <true\_result>, <false\_or\_unknown\_result>)

This expression is equivalent to the following.

CASE WHEN <predicate> THEN <true\_result> ELSE <false\_or\_unknown\_result> END

The expression IIF(orderyear = 2012, qty, 0) returns the value in the qty attribute when the orderyear attribute is equal to 2012, and zero otherwise.

CHOOSE function allows you to provide a position and a list of expressions, and returns the expression in the indicated position. The function takes the following form.

CHOOSE(<pos>, <exp1>, <exp2>, …, <expn>)

For example, the expression CHOOSE(2, 'x', 'y', 'z') returns 'y'.

The NEWID function generates GUID values in random order, whereas the NEWSEQUENTIAL ID function generates GUIDs that increase in a sequential order.

The + operator by default yields a NULL result on NULL input, whereas the CONCAT function treats NULLs as empty strings. Tatt**: Use CONCAT most of the times**. **USE SYSDATETIME most of the times as it returns as DATETIME2**.

Tatt: The NEWID function creates GUIDs in random order. You would consider it when the size overhead is not a major issue and the ability to generate a unique value across time and space, from anywhere, in random order is a higher priority. The NEWSEQUENTIALID function generates GUIDs in increasing order within the machine. It helps reduce fragmentation and works well when a single

session loads the data, and the number of drives is small. However, you should carefully consider an alternative using another key generator, like a sequence object, with a smaller type when possible.

We have our own custom table partitioning solution because we’re using the Standard edition of SQL Server. We use a surrogate key of a UNIQUEIDENTIFIER type with the NEWID function invoked by a default constraint expression as the primary key for the tables. We chose this approach because we do not want keys to conflict across the different tables. This primary key is also our clustered index key. Do you have any recommendations concerning our choice of a key?

UNIQUEIDENTIFIER type is large—16 bytes. And because it’s also the clustered index key, it is copied to all nonclustered indexes. Also, due to the random order in which the NEWID function generates values, there’s probably a high level of fragmentation in the index. A different approach to consider (and test!) is switching to an integer type and using the sequence object to generate keys that do not

conflict across tables. Due to the reduced size of the type, with the multiplied effect on nonclustered indexes, performance of reads will likely improve. The values will be increasing, and as a result, there will be less fragmentation, which will also likely have a positive effect on reads.

Tatt: **use COALESCE and not ISNULL, use CURRENT\_TIMESTAMP and not GETDATE, and use CASE and not IIF.**

Tatt: a predicate with a null will always be false. More specifically the result will be unknown and both false and unknown cases are excluded from the results of the query (unless the predicate is testing for nulls like ‘IS NULL’). The following query, intuitively, should have returned those regions as well which have region as ‘NULL’.

SELECT empid, firstname, lastname, country, region, city

FROM HR.Employees

WHERE region <> N'WA';

Fix it by:

SELECT empid, firstname, lastname, country, region, city

FROM HR.Employees

WHERE region <> N'WA'

OR region IS NOT NULL;

Based on the query filters that appear in the query, SQL Server can evaluate the option to use indexes to get to the data efficiently without requiring a full scan of the table. It’s important to note, though, that the predicate needs to be of a form known as a *search argument* (SARG) to allow efficient use of the index. A predicate in the form *column operator value* or *value operator column* can be a search argument. For example, predicates like col1 = 10, and col1 > 10 are search arguments. Applying manipulation to the filtered column in most cases prevents the predicate from being a search argument. An example for manipulation of the filtered column is applying a function

to it, as in F(col1) = 10, where F is some function. There are some exceptions to this rule, but they are very uncommon.

Tatt: so using the functions in predicates makes the query slower as index can’t be used in that case. (I think also making function calls are more expensive as compared to inline code).

Suppose you have a stored procedure that accepts an input parameter @dt representing an input shipped date.

SELECT orderid, orderdate, empid

FROM Sales.Orders

WHERE shippeddate = @dt;

It does not handle nulls. So you might do this:

SELECT orderid, orderdate, empid

FROM Sales.Orders

WHERE COALESCE(shippeddate, '19000101') = COALESCE(@dt, '19000101');

Now the predicate is not in the form of a search argument as we are using functions. So the index on the ‘shippeddate’ column can’t be efficiently used. So we should get the predicate in the ‘search argument’ form by **avoiding manipulation of filtered column**:

SELECT orderid, orderdate, empid

FROM Sales.Orders

WHERE shippeddate = @dt

OR (shippeddate IS NULL AND @dt IS NULL);

Another **example for manipulation involves the filtered column** in an expression; for example, col1 - 1 <= @n. Sometimes, you can rewrite the predicate to a form that is a search argument, and then allow efficient use of indexing. The last predicate, for example, can be rewritten using simple math as col1 <= @n + 1.

For example, consider the following filter predicate.

WHERE propertytype = 'INT' AND CAST(propertyval AS INT) > 10

Due to the all-at-once concept in the language (same reason why u can’t refer to column alias in SELECT clause), it is not necessarily going to evaluate the expressions in left-to-right order. It could decide, based on cost-related reasons, to start with the second expression, and then if the second expression evaluates to true, to evaluate the first expression as well. This means that if there are rows in the table where propertytype is different than 'INT', and in those rows propertyval isn’t convertible to INT, the query can fail due to a conversion error.

A simple option is to use the TRY\_CAST function instead of CAST.

WHERE propertytype = 'INT' AND TRY\_CAST(propertyval AS INT) > 10

Tatt: So do not depend on assumed execution order (it could be right to left OR left to right or all at once…sql server decides)??

If you write an expression that involves operands of different types, SQL Server will have to apply implicit conversion to align the types. The right form for a **Unicode character string literal is to prefix the literal with a capital N and delimit the literal with single quotation marks**; for example, N'literal'. For a regular character string literal, you just delimit the literal with single quotation marks; for example, 'literal'. It’s a very typical bad habit to specify a regular character string literal when the filtered column is of a Unicode type, as in the following example.

SELECT empid, firstname, lastname

FROM HR.Employees

WHERE lastname = 'Davis';

T-SQL provides the LIKE predicate, which you can use to filter character string data (regular and Unicode) based on pattern matching. The form of a predicate using LIKE is as follows.

<column> LIKE <pattern>

If you want to look for a character that is considered a wildcard, you can indicate it after a character that you designate as an escape character by using the ESCAPE keyword. For example, the expression col1 LIKE '!\_%' ESCAPE '!' looks for strings that start with an underscore (\_) by using an exclamation point (!) as the escape character.

When the LIKE pattern starts with a known prefix—for example**, col LIKE 'ABC%'— SQL Server can potentially efficiently use an index on the filtered column; in other words, SQL Server can rely on index ordering. When the pattern starts with a wildcard—for example, col LIKE '%ABC%'—SQL Server cannot rely on index ordering anymore**. Also, when looking for a string that starts with a known prefix (say, ABC) make sure you use the LIKE predicate, as in col LIKE 'ABC%', because this form is considered a search argument. Recall that applying manipulation to the filtered column prevents the predicate from being a search argument. For example, the form LEFT(col, 3) = 'ABC' isn’t a search argument and

will prevent SQL Server from being able to use an index efficiently.

SELECT orderid, orderdate, empid, custid

FROM Sales.Orders

WHERE orderdate = '02/12/07';

If you’re an American, this form probably means February 12, 2007, to you. However, if you’re British, this form probably means December 2, 2007. If you’re Japanese, it probably means December 7, 2002. The question is, when SQL Server converts this character string to a

date and time type to align it with the filtered column’s type, how does it interpret the value? As it turns out, it depends on the language of the logon that runs the code. Each logon has a default language associated with it, and the default language sets various session options on the logon’s behalf, including one called DATEFORMAT. A logon with us\_english will have the DATEFORMAT setting set to mdy, British to dmy, and Japanese to ymd. The problem is, how do you as a developer express a date if you want it to be interpreted the way you intended, regardless of who runs your code? There are two main approaches. One is to use a form that is considered language-neutral.

For example, **the form '20070212' is always interpreted as ymd, regardless of your language**. Note that the form '2007-02-12' is considered language-neutral only for the data types DATE, DATETIME2, and DATETIMEOFFSET. Unfortunately, due to historic reasons, this form is considered language-dependent for the types DATETIME and SMALLDATETIME. **The advantage of the form without the separators is that it is language-neutral for all date and time types**. So the recommendation is to write the query like the following.

SELECT orderid, orderdate, empid, custid

FROM Sales.Orders

WHERE orderdate = '20070212';

The filtered column orderdate is of a DATETIME data type representing both date and time. Yet the literal specified in the filter contains only a date part. When SQL Server converts the literal to the filtered column’s type, it assumes midnight when a time part isn’t indicated. If you want such a filter to return all rows from the specified date, you need to ensure that you store all values with midnight as the time.

Tatt: so far what I have seen, it is the literal that is converted to columns’ type and not the other way around (for implicit coversion).

Another important aspect of filtering date and time data is trying whenever possible to use search arguments. For example, suppose that you need to filter only orders placed in February 2007. You can use the YEAR and MONTH functions, as in the following.

SELECT orderid, orderdate, empid, custid

FROM Sales.Orders

WHERE YEAR(orderdate) = 2007 AND MONTH(orderdate) = 2;

However, because here you apply manipulation to the filtered column, the predicate is not considered a search argument, and therefore, SQL Server won’t be able to rely on index ordering. You could revise your predicate as a range, like the following.

SELECT orderid, orderdate, empid, custid

FROM Sales.Orders

WHERE orderdate >= '20070201' AND orderdate < '20070301';

Now that you don’t apply manipulation to the filtered column, the predicate is considered a search argument, and there’s the potential for SQL Server to rely on index ordering. If you’re wondering why this code expresses the date range by using greater than or equal

to (>=) and less than (<) operators as opposed to using BETWEEN, there’s a reason for this. When you are using BETWEEN and the column holds both date and time elements, what do you use as the end value? As you might realize, for different types, there are different precisions. What’s more, suppose that the type is DATETIME, and you use the following predicate.

WHERE orderdate BETWEEN '20070201' AND '20070228 23:59:59.999'

This type’s precision is three and a third milliseconds. The milliseconds part of the end point 999 is not a multiplication of the precision unit, so SQL Server ends up rounding the value to midnight of March 1, 2007. As a result, you may end up getting some orders that you’re

not supposed to see. In short, instead of BETWEEN, use >= and <, and this form will work correctly in all cases, with all date and time types, whether the time portion is applicable or not.

Tatt: try to use datetime2 for columns. Use date format ymd. Do not use BETWEEN.

Tatt: about the search argument form of predicates (when filter column is not manipulated), is it also helpful when the column we are filtering on is not used in an index??? I do not think so.

Tatt: since SQL is based on set theory and in sets, order does not matter, so sql results, unless explicitly, ordered can’t be counted upon to be in a particular order (insertion order or index order or primary key order).

With T-SQL, you can sort by ordinal positions of columns in the SELECT list, but it is considered a bad practice. Consider the following query as an example.

SELECT empid, firstname, lastname, city, MONTH(birthdate) AS birthmonth

FROM HR.Employees

WHERE country = N'USA' AND region = N'WA'

ORDER BY 4, 1;

Note that you can order the result rows by elements that you’re not returning. For example, the following query returns, for each qualifying employee, the employee ID and city, ordering the result rows by the employee birth date.

SELECT empid, city

FROM HR.Employees

WHERE country = N'USA' AND region = N'WA'

ORDER BY birthdate;

This rule changes when the DISTINCT clause is also specified—and for a good reason. When DISTINCT is used, duplicates are removed; then the result rows don’t necessarily map to source rows in a one-to-one manner, rather than oneto-many. For example, try to reason why the following query isn’t valid.

SELECT DISTINCT city

FROM HR.Employees

WHERE country = N'USA' AND region = N'WA'

ORDER BY birthdate;

You can have multiple employees—each with a different birth date—from the same city. But you’re returning only one row for each distinct city in the result. So given one city (say, Seattle) with multiple employees, which of the employee birth dates should apply as the ordering value? The query won’t just pick one; rather, it simply fails.

But the following works:

SELECT DISTINCT city, birthdate

FROM HR.Employees

WHERE country = N'USA' AND region = N'WA'

ORDER BY birthdate;

How NULLs should behave in terms of sorting? Should they all sort together? If so, should they sort before or after non-NULL values? Standard SQL says that NULLs should sort together, but leaves it to the implementation to decide whether to sort them before or after non-NULL values. In SQL Server the decision was to sort them before non-NULLs (when using an ascending direction).

SELECT orderid, shippeddate

FROM Sales.Orders

WHERE custid = 20

ORDER BY shippeddate;

Remember that unshipped orders have a NULL in the shippeddate column; hence, they sort before shipped orders, as the query output shows.

orderid shippeddate

----------- -----------------------

11008 NULL

11072 NULL

10258 2006-07-23 00:00:00.000

10263 2006-07-31 00:00:00.000

10351 2006-11-20 00:00:00.000

Standard SQL supports the options NULLS FIRST and NULLS LAST to control how NULLs sort, but T-SQL doesn’t support this option.

Creating the right indexes can help SQL Server avoid the need to actually sort the data to address an ORDER BY request. Without good indexes, SQL Server needs to sort the data, and sorting can be expensive, especially when a large set is involved

Tatt: When DISTINCT is used in SELECT list, then the col used in ORDER BY clause has to be in the SELECT list.

If the filter involves some ordering specification and a requested number of rows, then T-SQL provides two options to handle such filtering needs: one is the **proprietary TOP option and the other is the standard OFFSET-FETCH option** that was introduced in SQL Server 2012.

SELECT TOP (3) orderid, orderdate, custid, empid

FROM Sales.Orders

ORDER BY orderdate DESC;

Tatt: T-SQL supports specifying the number of rows to filter using the TOP option in SELECT queries without parentheses, but that’s only for backward-compatibility reasons. The correct syntax is with parentheses.

You can also specify a percent of rows to filter instead of a number. To do so, specify a FLOAT value in the range 0 through 100 in the parentheses, and the keyword PERCENT after the parentheses, as follows.

SELECT TOP (1) PERCENT orderid, orderdate, custid, empid

FROM Sales.Orders

ORDER BY orderdate DESC;

The PERCENT option puts a ceiling on the resulting number of rows if it’s not whole. In this example, without the TOP option, the number of rows in the result is 830. Filtering 1 percent gives you 8.3, and then the ceiling of this value gives you 9; hence, the query returns 9 rows.

The TOP option isn’t limited to a constant input; instead, it allows you to specify a selfcontained expression. From a practical perspective, this capability is especially important when you need to pass a parameter or a variable as input, as the following code demonstrates.

DECLARE @n AS BIGINT = 5;

SELECT TOP (@n) orderid, orderdate, custid, empid

FROM Sales.Orders

ORDER BY orderdate DESC;

Tatt: you can use TOP without ORDER BY but then there is no guarantee that you will get the same rows back. **The query won’t be deterministic**.

If you are really after three arbitrary rows, it might be a good idea to add an ORDER BY clause with the expression (SELECT NULL) to let people know that your choice is intentional and not an oversight. Here’s how your query would look.

SELECT TOP (3) orderid, orderdate, custid, empid

FROM Sales.Orders

ORDER BY (SELECT NULL);

But what if there are other rows in the result without TOP that have the same order date as in the last row here? You don’t always care about guaranteeing deterministic or repeatable results; but if you do, two options are available to you. One option is to ask to include all ties with the last row by adding the WITH TIES option, as follows.

SELECT TOP (3) WITH TIES orderid, orderdate, custid, empid

FROM Sales.Orders

ORDER BY orderdate DESC;

Of course, this could result in returning more rows than you asked for. The other option to guarantee determinism is to break the ties by adding a tiebreaker that makes the ordering unique

SELECT TOP (3) WITH TIES orderid, orderdate, custid, empid

FROM Sales.Orders

ORDER BY orderdate DESC, orderid DESC;

TOP option can also be used in modification statements to limit how many rows get modified.

The OFFSET-FETCH option is a filtering option that, like TOP, you can use to filter data based on a specified number of rows and ordering. **But unlike TOP, it is standard, and also has a skipping capability, making it useful for ad-hoc paging purposes**. The OFFSET and FETCH clauses appear right after the ORDER BY clause, and in fact, in T-SQL, they require an ORDER BY clause to be present.

The OFFSET and FETCH clauses appear right after the ORDER BY clause, and in fact, in T-SQL, they require an ORDER BY clause to be present. You first specify the OFFSET clause indicating how many rows you want to skip (0 if you don’t want to skip any); you then optionally specify the FETCH clause indicating how many rows you want to filter. For example, the following query defines ordering based on order date descending, followed by order ID descending; it then skips 50 rows and fetches the next 25 rows.

SELECT orderid, orderdate, custid, empid

FROM Sales.Orders

ORDER BY orderdate DESC, orderid DESC

OFFSET 50 ROWS FETCH NEXT 25 ROWS ONLY;

In order to make the syntax intuitive, **you can use the keywords NEXT or FIRST interchangeably**. When skipping some rows, it might be more intuitive to you to use the keywords FETCH NEXT to indicate how many rows to filter; but when not skipping any rows, it might be

more intuitive to you to use the keywords FETCH FIRST, as follows.

SELECT orderid, orderdate, custid, empid

FROM Sales.Orders

ORDER BY orderdate DESC, orderid DESC

OFFSET 0 ROWS FETCH FIRST 25 ROWS ONLY;

For similar reasons, **you can use the singular form ROW or the plural form ROWS interchangeably**, both for the number of rows to skip and for the number of rows to filter.

**While in T-SQL, a FETCH clause requires an OFFSET clause, and the OFFSET clause doesn’t require a FETCH clause**. In other words, by indicating an OFFSET clause, you’re requesting to skip some rows; then by not indicating a FETCH clause, you’re requesting to return all remaining rows. For example, the following query requests to skip 50 rows, returning all the rest.

SELECT orderid, orderdate, custid, empid

FROM Sales.Orders

ORDER BY orderdate DESC, orderid DESC

OFFSET 50 ROWS;

As mentioned earlier, the OFFSET-FETCH option requires an ORDER BY clause. But what if you need to filter a certain number of rows based on arbitrary order? To do so, you can specify the expression (SELECT NULL) in the ORDER BY clause, as follows.

SELECT orderid, orderdate, custid, empid

FROM Sales.Orders

ORDER BY (SELECT NULL)

OFFSET 0 ROWS FETCH FIRST 3 ROWS ONLY;

With both the OFFSET and the FETCH clauses, you can use expressions as inputs. This is very handy when you need to compute the input values dynamically. For example, suppose that you’re implementing a paging concept where you return to the user one page of rows at a time. The user passes as input parameters to your procedure or a function the page number they are after (@pagenum parameter) and page size (@pagesize parameter). This means that you need to skip as many rows as @pagenum minus one times @pagesize, and fetch the next @pagesize rows. This can be implemented using the following code (using local variables for simplicity).

DECLARE @pagesize AS BIGINT = 25, @pagenum AS BIGINT = 3;

SELECT orderid, orderdate, custid, empid

FROM Sales.Orders

ORDER BY orderdate DESC, orderid DESC

OFFSET (@pagesize - 1) \* @pagesize ROWS FETCH NEXT @pagesize ROWS ONLY;

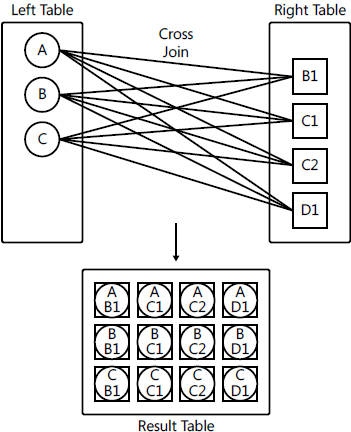
Tatt: Because **the OFFSET-FETCH option is standard and TOP isn’t**, in cases where they are logically equivalent, it’s recommended to stick to the former However, for now, **OFFSET-FETCH does not support options similar to TOP’s PERCENT and WITH TIES**. From a performance standpoint**, you should evaluate indexing the ORDER BY columns to support the TOP and OFFSET-FETCH options. Such indexing serves a very similar purpose to indexing filtered columns and can help avoid scanning unnecessary data as well as sorting**.

**How do you guarantee deterministic results with TOP**? By either returning all ties by using the WITH TIES option or by defining unique ordering to break ties (for example by using the primary key or unique key in the ORDER BY list).

**Performance scenario**: You trace a typical workload submitted to the system and observe very slow query run times. You see a lot of network traffic. You see that many queries return all rows to the client and then the client handles the filtering. Queries that do filter data often manipulate the filtered columns. All queries have ORDER BY clauses, and when you inquire about this, you are told that it’s not really needed, but the developers got accustomed to doing so—just in case. You identify a lot of expensive sort operations.

Tatt: if there are ties, then the result would be non-deterministic.

Cross Join: *Cartesian product* of the two input tables. If you have *m* rows in table T1 and *n* rows in table T2, the result of a cross join between T1 and T2 is a virtual table with *m* × *n* rows.



SELECT

D.n AS DAY, S.n AS SHIFT

FROM DBO.Nums AS D CROSS JOIN dbo.Nums AS S

WHERE D.n<=7 AND S.n<=3;

If n is a large number say 10000, then doing a Cartesian product in the FROM clause would result a very large table. And only then after the CROSS JOIN we apply the filter to restrict rows to 21. That is logical query processing phases as in theory. But SQL server is smarter and its query optimizer applies the filter first and then does the CROSS JOIN. I could have written the query such I created 2 tables first with filtered rows (maybe using CTE or other such mechanisms) and then did the CROSS JOIN but query optimizer will do it for me.

Tatt: The other thing that I should repeat here I that the filter columns should be indexed for faster processing.

Both standard SQL and T-SQL support an older syntax where you specify a comma between the table names, as in FROM T1, T2. However, for a number of reasons, it is recommended to stick to the newer syntax; it is less prone to errors and allows for more consistent code.

INNER JOIN: you can match rows from two tables based on a predicate—usually one that compares a primary key value in one side to a foreign key value in another side. Assuming the join is an equijoin (using a predicate with an equality operator such as lefttable.keycol = righttable.keycol), the inner join returns only matching rows for which the predicate evaluates to true. Rows for which the predicate evaluates to false or unknown are discarded.

SELECT

S.companyname AS supplier, S.country,

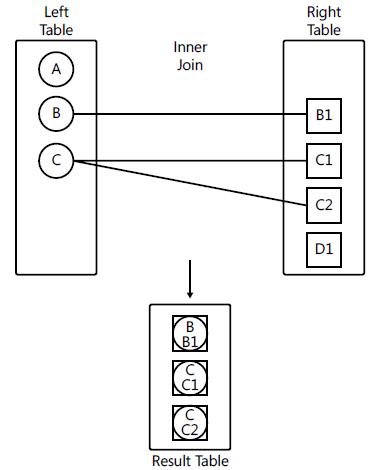
P.productid, P.productname, P.unitprice

FROM Production.Suppliers AS S

INNER JOIN Production.Products AS P

ON S.supplierid = P.supplierid

WHERE S.country = N'Japan';



Rows from either side that don’t find a match in the other are discarded. For example, suppliers from Japan with no related products aren’t returned.

Tatt: Often, when joining tables, you join them based on a foreign key–unique key relationship. It’s also important to note that when you define a primary key or unique constraint, SQL Server creates a unique index on the constraint columns to enforce the constraint’s uniqueness property. But when you define a foreign key, SQL Server doesn’t create any indexes on the foreign key columns. Such indexes could improve the performance of joins based on those relationships. Because SQL Server doesn’t create such indexes automatically, it’s your responsibility to identify the cases where they can be useful and create them. So when working on index tuning, one interesting area to examine is foreign key columns, and evaluating the benefits of creating indexes on those.

A very common question is, “What’s the difference between the ON and the WHERE clauses, and does it matter if you specify your predicate in one or the other?” The answer is that for inner joins it doesn’t matter. Both clauses perform the same filtering purpose. Both filter only rows for which the predicate evaluates to true and discard rows for which the predicate evaluates to false or unknown. In terms of logical query processing, the WHERE is evaluated right after the FROM, so conceptually it is equivalent to concatenating the predicates with an AND operator. SQL Server knows this, and therefore can internally rearrange the order in which it evaluates the predicates in practice, and it does so based on cost estimates.

For these reasons, if you wanted, you could rearrange the placement of the predicates from the previous query, specifying both in the ON clause, and still retain the original meaning, as follows.

SELECT

S.companyname AS supplier, S.country,

P.productid, P.productname, P.unitprice

FROM Production.Suppliers AS S

INNER JOIN Production.Products AS P

ON S.supplierid = P.supplierid

AND S.country = N'Japan';

Tatt: You have to have atleast one ON predicate and then the rest of the predicates in the WHERE clause can be AND’ed to it thereby removing the need for the WHERE clause. But for outer joins ON and WHERE play different roles; you need to figure out, according to your needs, which is the appropriate clause for each of your predicates.

Typical example for INNER JOIN is extract emp and manager from emp table:

SELECT

CONCAT(E.firstname, N' ',E.lastname) AS EMPNAME, CONCAT(M.firstname,N' ', M.lastname) AS MANAGERNAME

FROM HR.Employees AS E INNER JOIN HR.Employees AS M

ON E.mgrid = M.empid

Now this does not include the row where mgrid is NULL(remember to think of nulls in predicates).

OUTER JOIN: preserve all rows from one or both sides of the join, never mind if there are matching rows in the other side based on the ON predicate. By using the keywords LEFT OUTER JOIN (or LEFT JOIN for short), you ask to preserve the left table. The join returns what an inner join normally would—that is, matches (call those inner rows). In addition, the join also returns rows from the left that had no matches in the right table (call those outer rows), with NULLs used as placeholders in the right side. So from query above, since we have to include the row corresponding to the CEO, this is how we would do it:

SELECT

CONCAT(E.firstname, ' ',E.lastname) AS EMPNAME, CONCAT(M.firstname,' ', M.lastname) AS MANAGERNAME

FROM HR.Employees AS E LEFT OUTER JOIN HR.Employees AS M

ON E.mgrid = M.empid;

As an example, the following query returns suppliers from Japan and the products they supply, including suppliers from Japan that don’t have related products.

SELECT

S.companyname AS supplier, S.country,

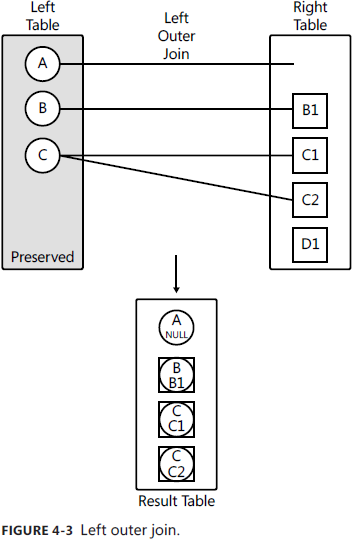
P.productid, P.productname, P.unitprice

FROM Production.Suppliers AS S

LEFT OUTER JOIN Production.Products AS P

ON S.supplierid = P.supplierid

WHERE S.country = N'Japan';



With outer joins, the ON and WHERE clauses play very different roles, and therefore, they aren’t interchangeable. The WHERE clause still plays a simple filtering role—namely, it keeps true cases and discards false and unknown cases. However, the ON clause doesn’t play a simple filtering role; rather, it’s more a *matching* role. In other words, a row in the preserved side will be returned whether the ON predicate

finds a match for it or not. **So the ON predicate only determines which rows from the nonpreserved side get matched to rows from the preserved side—not whether to return the rows from the preserved side. In other words, ON is not final with respect to the preserved side of the join. WHERE is final**. So when in doubt whether to specify the predicate in the ON or WHERE clauses, ask yourself: Is the predicate used to filter or match? Is it supposed to be final or nonfinal?

SELECT

S.companyname AS supplier, S.country,

P.productid, P.productname, P.unitprice

FROM Production.Suppliers AS S

LEFT OUTER JOIN Production.Products AS P

ON S.supplierid = P.supplierid

AND S.country = N'Japan';

Observe what’s different in the result (shown here in abbreviated form) and see if you can

explain in your own words what the query returns now.

supplier country productid productname unitprice

--------------- -------- ---------- -------------- ----------

Supplier SWRXU UK NULL NULL NULL

Supplier VHQZD USA NULL NULL NULL

Supplier STUAZ USA NULL NULL NULL

Supplier QOVFD Japan 9 Product AOZBW 97.00

Supplier QOVFD Japan 10 Product YHXGE 31.00

Supplier QOVFD Japan 74 Product BKAZJ 10.00

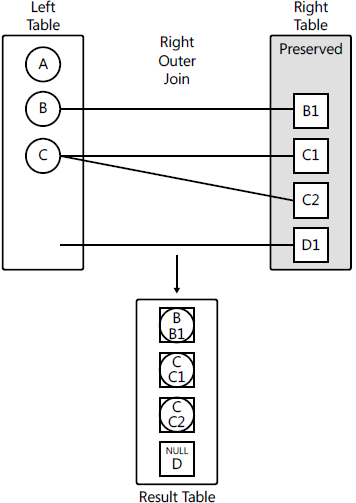
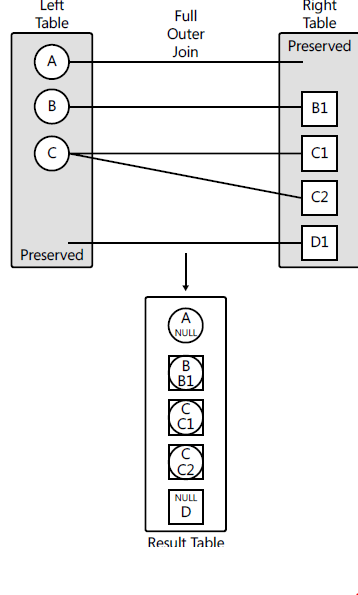
Supplier EQPNC Spain NULL NULL NULL

...

**Now that both predicates appear in the ON clause, both serve a matching purpose. What this means is that all suppliers are returned—even those that aren’t from Japan. But in order to match a product to a supplier, the supplier IDs in both sides need to match, and the supplier country needs to be Japan**.

Tatt: think in terms of matching and filtering. Matching preserves the rows in the right side table (for left outer join) while filtering filters rows from left side table

T-SQL also supports a full outer join (FULL OUTER JOIN, or FULL JOIN in short), that preserves both sides. A full outer join returns the inner rows that are normally returned from an inner join; plus rows from the left that don’t have matches in the right, with NULLs used as placeholders in the right side; plus rows from the right that don’t have matches in the left, with NULLs used as placeholders in the left side.

MULTI JOIN query: A multi-join query evaluates the joins conceptually from left to right. So the result of one join is used as the left input to the next join. If you don’t understand this, you can end up with logical bugs, especially when outer joins are involved. (With inner and cross joins, the order cannot affect the meaning.)

Suppose that you wanted to return all suppliers from Japan, and matching products where relevant. For this, you need an outer join between Production.Suppliers and Production.Products, preserving Suppliers. But you also want to include product category information, so you add an inner join to Production.Categories, as follows.

SELECT

S.companyname AS supplier, S.country,

P.productid, P.productname, P.unitprice,

C.categoryname

FROM Production.Suppliers AS S

LEFT OUTER JOIN Production.Products AS P

ON S.supplierid = P.supplierid

INNER JOIN Production.Categories AS C

ON C.categoryid = P.categoryid

WHERE S.country = N'Japan';

Tatt: I would read this query as: I want all the suppliers who are from japan. And if those suppliers are producing any products, I want that info. And for those products get me the category info as well. **But it does not work**. It did not return a supplier from Japan which did not have any products.??? Conceptually, the first join included outer rows (suppliers with no products) but produced NULLs in the product attributes in those rows. Then the join to Production.Categories compared the NULL categoryid values in the outer rows to categoryid values in Production. Categories, and discarded those rows. In short, the inner join that followed the outer join nullified the outer part of the join.

There are a number of ways to address this problem, but probably the most natural is to use an interesting capability in the language—separate some of the joins to their own independent logical phase. What you’re after is a left outer join between Production.Suppliers and

the result of the inner join between Production.Products and Production.Categories. You can phrase your query exactly like this.

SELECT

S.companyname AS supplier, S.country,

P.productid, P.productname, P.unitprice,

C.categoryname

FROM Production.Suppliers AS S

LEFT OUTER JOIN

(Production.Products AS P

INNER JOIN Production.Categories AS C

ON C.categoryid = P.categoryid)

ON S.supplierid = P.supplierid

WHERE S.country = N'Japan';

Cross joins return a Cartesian product of the rows from both sides.

■■ Inner joins match rows based on a predicate and return only matches.

■■ Outer joins match rows based on a predicate and return both matches and nonmatches

from the tables marked as preserved.

■■ Multi-join queries involve multiple joins. They can have a mix of different join types.

You can control the logical join ordering by using parentheses or by repositioning the

ON clauses.

Tatt: use keyword ‘match rows’ for ON clause predicates. Use keyword ‘filter rows’ for WHERE clause predicates.

T-SQL supports nesting of queries. This is a convenient part of the language that you can use to refer to one query’s result from another. You do not need to store the result of one query in a variable in order to be able to refer to that result from another query.

Subqueries

*Subqueries* can be self-contained—namely, independent of the outer query; or they can be correlated—namely, having a reference to a column from the table in the outer query. In terms of the result of the subquery, it can be scalar, multi-valued, or table-valued.

Self-Contained Subqueries

Self-contained subqueries are subqueries that have no dependency on the outer query. If you want, you can highlight the inner query and run it independently. This makes the troubleshooting of problems with self-contained subqueries easier compared to correlated subqueries. As mentioned, a subquery can return different forms of results. It can return a single value, multiple values, or even an entire table result.

Subqueries that return a single value, or scalar subqueries, can be used where a singlevalued expression is expected, like in one side of a comparison. For example, the following query uses a self-contained subquery to return the products with the minimum unit price.

SELECT productid, productname, unitprice

FROM Production.Products

WHERE unitprice =

(SELECT MIN(unitprice)

FROM Production.Products);

Here’s the output of this query.

productid productname unitprice

---------- -------------- ----------

33 Product ASTMN 2.50

Note that if what’s supposed to be a scalar subquery returns in practice more than one value, the code fails at run time. If the scalar subquery returns an empty set, it is converted to a NULL. A subquery can also return multiple values in the form of a single column. Such a subquery can be used where a multi-valued result is expected—for example, when using the IN predicate. As an example, the following query uses a multi-valued subquery to return products supplied by suppliers from Japan. As with predicates in general, you can negate an IN predicate, so if you wanted to return products supplied by suppliers that are not from Japan, simply change IN to NOT IN.

SELECT productid, productname, unitprice

FROM Production.Products

WHERE supplierid IN

(SELECT supplierid

FROM Production.Suppliers

WHERE country = N'Japan');

Tatt: this could have been done with an INNER JOIN as well??

Correlated Subqueries

*Correlated subqueries* are subqueries where the inner query has a reference to a column from the table in the outer query. As an example, suppose that you need to return products with the minimum unit price per category. You can use a correlated subquery to return the minimum unit price out of the products where the category ID is equal to the one in the outer row (the correlation), as follows.

SELECT categoryid, productid, productname, unitprice

FROM Production.Products AS P1

WHERE unitprice =

(SELECT MIN(unitprice)

FROM Production.Products AS P2

WHERE P2.categoryid = P1.categoryid);

The following query returns customers who placed orders on February 12, 2007 (again I think this could have been done with an INNER JOIN as well):

SELECT custid, companyname

FROM Sales.Customers AS C

WHERE EXISTS

(SELECT \*

FROM Sales.Orders AS O

WHERE O.custid = C.custid

AND O.orderdate = '20070212');

The EXISTS predicate accepts a subquery as input and returns true when the subquery returns at least one row and false otherwise. So the outer query returns a customer only if there’s at least one order placed by that customer on the date in question. As with other predicates, you can negate the EXISTS predicate as well.

Tatt: IN and EXISTS are predicates. So they return true or false.

Table Expressions

***Table expressions* are named queries(views is a table expression type)**. You write an inner query that returns a relational result set, name it, and query it from an outer query. T-SQL supports four forms of table expressions:

■■ Derived tables

■■ Common table expressions (CTEs)

■■ Views

■■ Inline table-valued functions

The first two are visible only to the statement that defines them. As for the last two, you preserve the definition of the table expression in the database as an object; therefore, it’s reusable, and you can also control access to the object with permissions.

The first two are visible only to the statement that defines them. As for the last two, you preserve the definition of the table expression in the database as an object; therefore, it’s reusable, and you can also control access to the object with permissions. Note that because a table expression is supposed to represent a relation, the inner query defining it needs to be relational. This means that all columns returned by the inner query must have names (use aliases if the column is a result of an expression), and all column names must be unique. Also, the inner query is not allowed to have an ORDER BY clause. (Remember, a set has no order.) There’s an exception to the last rule: If you use the TOP or OFFSET-FETCH option in the inner query, the ORDER BY serves a meaning that is not related to presentation ordering; rather, it’s part of the filter’s specification. So if the inner query uses the TOP or OFFSET-FETCH option, it’s allowed to have an ORDER BY clause as well. But then the outer query has no presentation ordering guarantees if it doesn’t have its own

ORDER BY clause.

**Optimization of Table Expressions**: It’s important to note that, from a performance standpoint, when SQL Server optimizes queries involving table expressions, it first unnests the table expression’s logic, and therefore interacts with the underlying tables directly. It does not somehow persist the table expression’s result in an internal work table and then interact with that work table. This means that table expressions don’t have a performance side to them—neither good nor bad—just no side.

Tatt: so they would always be ‘inlined’ into the calling query (using lingo from procedural programming languages). So exactly as if you had written the table expression inside the query using it.

Derived Tables

A derived table is probably the form of table expression that most closely resembles a subquery—only a subquery that returns an entire table result. You define the derived table’s inner query in parentheses in the FROM clause of the outer query, and specify the name of the derived table after the parentheses.

Before demonstrating the use of derived tables, this section describes a query that returns a certain desired result. Then it explains a need that cannot be addressed directly in the query, and shows how you can address that need by using a derived table (or any other table

expression type for that matter). Consider the following query, which computes row numbers for products, partitioned by categoryid, and ordered by unitprice and productid.

SELECT

ROW\_NUMBER() OVER(PARTITION BY categoryid

ORDER BY unitprice, productid) AS rownum,

categoryid, productid, productname, unitprice

FROM Production.Products;

This query generates the following output, shown here in abbreviated form.

rownum categoryid productid productname unitprice

------- ----------- ---------- -------------- ----------

1 1 24 Product QOGNU 4.50

2 1 75 Product BWRLG 7.75

3 1 34 Product SWNJY 14.00

4 1 67 Product XLXQF 14.00

5 1 70 Product TOONT 15.00

...

1 2 3 Product IMEHJ 10.00

2 2 77 Product LUNZZ 13.00

3 2 15 Product KSZOI 15.50

4 2 66 Product LQMGN 17.00

5 2 44 Product VJIEO 19.45

Tatt: PARTITITON BY is a sort of GROUP BY but the groups need not be collapsed down to aggregates for display purposes.

The thing with the ROW\_NUMBER function—and window functions in general—is that they are only allowed in the SELECT and ORDER BY clauses of a query. So, what if you want to filter rows based on such a function’s result?

For example, suppose you want to return only the rows where the row number is less than or equal to 2; namely, in each category, you want to return the two products with the lowest unit prices, with the product ID used as a tiebreaker. You are not allowed to refer to the ROW\_NUMBER function in the query’s WHERE clause. Remember also that according to logical query processing, you’re not allowed to refer to a column alias that was assigned in the SELECT list in the WHERE clause, because the WHERE clause is conceptually evaluated before the SELECT clause. You can circumvent the restriction by using a table expression. You write a query such as the previous query that computes the window function in the SELECT clause, and assign a column alias to the result column. You then define a table expression based on that query, and refer to the column alias in the outer query’s WHERE clause, as follows.

SELECT categoryid, productid, productname, unitprice

FROM (SELECT

ROW\_NUMBER() OVER(PARTITION BY categoryid

ORDER BY unitprice, productid) AS rownum,

categoryid, productid, productname, unitprice

FROM Production.Products) AS D

WHERE rownum <= 2;

This query generates the following output, shown here in abbreviated form.

categoryid productid productname unitprice

----------- ---------- -------------- ----------

1 24 Product QOGNU 4.50

1 75 Product BWRLG 7.75

2 3 Product IMEHJ 10.00

2 77 Product LUNZZ 13.00

3 19 Product XKXDO 9.20

3 47 Product EZZPR 9.50

**The outer query is allowed to refer to column aliases that were assigned by the inner query. That’s a classic use of table expressions**.

Two column aliasing options are available to you when working with derived tables: *inline* and *external*. With the inline form, you specify the column alias as part of the expression, as in <*expression*> AS *alias*. The last query used the inline form to assign the alias rownum to the expression with the ROW\_NUMBER function. With the external aliasing form, you don’t specify result column aliases as part of the column expressions; instead, you name all target columns right after the derived table’s name, as in FROM (…) AS D(rownum, categoryid, productid, productname, unitprice). With the external form, you must specify all target column names and not just those that are results of computations.

SELECT categoryid, productid, productname, unitprice

FROM (SELECT

ROW\_NUMBER() OVER(PARTITION BY categoryid

ORDER BY unitprice, productid) AS rownum,

categoryid, productid, productname, unitprice

FROM Production.Products) AS D(rownum,

categoryid, productid, productname, unitprice)

WHERE rownum <= 2;

There are a couple of problematic aspects to working with derived tables that stem from the fact that a derived table is defined in the FROM clause of the outer query. One problem has to do with cases where you want to refer to one derived table from another. In such a

case, you end up nesting derived tables, and nesting often complicates the logic, making it hard to follow and increasing the likelihood for errors. Consider the following general form of nesting of derived tables.

SELECT ...

FROM (SELECT

FROM (SELECT ...

FROM T1

WHERE ...) AS D1

WHERE ...) AS D2

WHERE ...;

The other problem with derived tables has to do with the “all-at-once” property of the language. Remember that all expressions that appear in the same logical query processing phase are conceptually evaluated at the same point in time. This is true even for table expressions.

As a result, the name assigned to a derived table is not visible to other elements that appear in the same logical query processing phase where the derived table name was defined. This means that if you want to join multiple instances of the same derived table, you can’t. You

have no choice but to duplicate the code, defining multiple derived tables based on the same query. The general form of such a query looks like the following.

SELECT ...

FROM (SELECT ...

FROM T1) AS D1

INNER JOIN

(SELECT ...

FROM T1) AS D2

ON ...;

The derived tables D1 and D2 are based on the same query. This repetition of code increases the likelihood for errors when you need to make revisions to the inner queries.

CTEs

A *common table expression (CTE)* is a similar concept to a derived table in the sense that it’s a named table expression that is visible only to the statement that defines it. Like a query against a derived table, a query against a CTE involves three main parts:

■■ The inner query

■■ The name you assign to the query and its columns

■■ The outer query

However, with CTEs, the arrangement of the three parts is different. Recall that with derived tables the inner query appears in the FROM clause of the outer query—kind of in the middle of things. With CTEs, you first name the CTE, then specify the inner query, and then the outer query—a much more modular approach.

WITH <CTE\_name>

AS

(

<inner\_query>

)

<outer\_query>;

For each product category the two products with the lowest unit prices using 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;

As you can see, it’s a similar concept to derived tables, except the inner query is not defined in the middle of the outer query; instead, first you define the inner query—from start to end—then the outer query—from start to end. This design leads to much clearer code that is easier to understand. You don’t nest CTEs like you do derived tables. If you need to define multiple CTEs, you simply separate them by commas. Each can refer to the previously defined CTEs, and the outer query can refer to all of them. After the outer query terminates, all CTEs defined in that WITH statement are gone. The fact that you don’t nest CTEs makes it easier to follow the logic and therefore reduces the chances for errors.

For example, if you want to refer to one CTE from another, you can use the following general form.

WITH C1 AS

(

SELECT ...

FROM T1

WHERE ...

),

(

SELECT

FROM C1

WHERE ...

)

SELECT ...

FROM C2

WHERE ...;

Because the CTE name is assigned before the start of the outer query, you can refer to multiple instances of the same CTE name, unlike with derived tables. The general form looks like the following.

WITH C AS

(

SELECT ...

FROM T1

)

SELECT ...

FROM C AS C1

INNER JOIN C AS C2

ON ...;

CTEs also have a recursive form. The body of the recursive query has two or more queries, usually separated by a UNION ALL operator. At least one of the queries in the CTE body, known as the anchor member, is a query that returns a valid relational result. The anchor query is invoked only once. In addition, at least one of the queries in the CTE body, known as the recursive member, has a reference to the CTE name. This query is invoked repeatedly until it returns an empty result set. In each iteration, the reference to the CTE name from the recursive member represents the previous result set. Then the reference to the CTE name from the outer query represents the unified results of the invocation of the anchor member and all invocations of the recursive member.

As an example, the following code uses a recursive CTE to return the management chain leading all the way up to the CEO for a specified employee.

WITH EmpsCTE AS

(

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

FROM HR.Employees

WHERE empid = 9

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;

This code returns the following output.

empid mgrid firstname lastname distance

----------- ----------- ---------- -------------------- -----------

9 5 Zoya Dolgopyatova 0

5 2 Sven Buck 1

2 1 Don Funk 2

1 NULL Sara Davis 3

The anchor member returns the row for employee 9. Then the recursive member is invoked repeatedly, and in each round joins the previous result set with the HR.Employees table to return the direct manager of the employee from the previous round. The recursive query stops as soon as it returns an empty set—in this case, after not finding a manager of the CEO. Then the outer query returns the unified results of the invocation of the anchor member (the row for employee 9) and all invocations of the recursive member (all

managers above employee 9).

Views and Inline Table-Valued Functions

For reusability, you need to store the definition of the table expression as an object in the database, and for this you can use either views or inline table-valued functions. Because these are objects in the database, you can control access by using permissions. The main difference between views and inline table-valued functions is that the former doesn’t accept input parameters and the latter does. As an example, suppose you need to persist the definition of the query with the row number computation from the examples in the previous sections. To achieve this, you create the following view.

IF OBJECT\_ID('Sales.RankedProducts', 'V') IS NOT NULL DROP VIEW 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

Note that it’s not the result set of the view that is stored in the database; rather, only its definition is stored. Now that the definition is stored, the object is reusable. Whenever you need to query the view, it’s available to you, assuming you have the permissions to query it.

SELECT categoryid, productid, productname, unitprice

FROM Sales.RankedProducts

WHERE rownum <= 2;

As for inline table-valued functions, they are very similar to views in concept; however, as mentioned, they do support input parameters.

IF OBJECT\_ID('HR.GetManagers', 'IF') IS NOT NULL DROP FUNCTION 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

Observe that the header assigns the function with a name (HR.GetManagers), defines the input parameter (@empid AS INT), and indicates that the function returns a table result (defined by the returned query).

SELECT \*

FROM HR.GetManagers(9) AS M;

**APPLY**

The APPLY operator is a powerful operator that you can use to apply a table expression given to it as the right input to each row from a table expression given to it as the left input. What’s interesting about the APPLY operator as compared to a join is that the right table expression can be correlated to the left table; in other words, the inner query in the right table expression can have a reference to an element from the left table. So conceptually, the right table expression is evaluated separately for each left row. This means that you can replace the use of cursors in some cases with the APPLY operator.

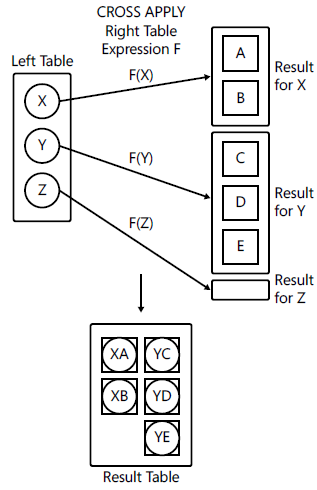
For example, suppose that you have a query that performs some logic for a particular customer. Suppose that you need to apply this query logic to each customer from the Sales .Customers table. You could use a cursor to iterate through the customers, and in each iteration

invoke the query for the current customer. Instead, you can use the APPLY operator, providing the Sales.Customers table as the left input, and a table expression based on your query as the right input. You can correlate the customer ID in the inner query of the right table expression to the customer ID from the left table. The two forms of the APPLY operator—CROSS and OUTER—are described in the next

sections.

CROSS APP LY

The CROSS APPLY operator operates on left and right table expressions as inputs. The right table expression can have a correlation to elements from the left table. The right table expression is applied to each row from the left input. What’s special about the CROSS APPLY operator as compared to OUTER APPLY is that if the right table expression returns an empty set for a left row, the left row isn’t returned.



The letters X, Y, and Z represent key values from the left table. F represents the table expression provided as the right input, and in parentheses, you can see the key value from the left row passed as the correlated element. On the right side of the illustration, you can see the result returned from the right table expression for each left row. Then at the bottom, you can see the result of the CROSS APPLY table operator, where left rows are matched with the respective right rows that were returned for them. Notice that a left row that gets an empty set back from the right table expression isn’t returned. Such is the case with the row with the key value Z.

As a more practical example, suppose that you write a query that returns the two products with the lowest unit prices for a specified supplier—say, supplier 1.

SELECT productid, productname, unitprice

FROM Production.Products

WHERE supplierid = 1

ORDER BY unitprice, productid

OFFSET 0 ROWS FETCH FIRST 2 ROWS ONLY;

This query generates the following output.

productid productname unitprice

---------- -------------- ----------

3 Product IMEHJ 10.00

1 Product HHYDP 18.00

Next, suppose that you need to apply this logic to each of the suppliers from Japan that you have in the Production.Suppliers table. You don’t want to use a cursor to iterate through the suppliers one at a time and invoke a separate query for each. Instead, you can use the

CROSS APPLY operator like in the following.

SELECT S.supplierid, S.companyname AS supplier, A.\*

FROM Production.Suppliers AS S

CROSS APPLY (SELECT productid, productname, unitprice

FROM Production.Products AS P

WHERE P.supplierid = S.supplierid

ORDER BY unitprice, productid

OFFSET 0 ROWS FETCH FIRST 2 ROWS ONLY) AS A

WHERE S.country = N'Japan';

This query generates the following output.

supplierid supplier productid productname unitprice

----------- --------------- ---------- -------------- ----------

4 Supplier QOVFD 74 Product BKAZJ 10.00

4 Supplier QOVFD 10 Product YHXGE 31.00

6 Supplier QWUSF 13 Product POXFU 6.00

6 Supplier QWUSF 15 Product KSZOI 15.50

As you can see in the query, the left input to the APPLY operator is the Production .Suppliers table, with only suppliers from Japan filtered. The right table expression is a correlated derived table returning the two products with the lowest prices for the left supplier. Because the APPLY operator applies the right table expression to each supplier from the left, you get the two products with the lowest prices per each supplier from Japan. Because the CROSS APPLY operator doesn’t return left rows for which the right table expression returns an empty set, suppliers from Japan who don’t have any related products aren’t returned.