**Bones of SQL - The Calendar Table (1 of 2)**

**Introduction**

This article is for relative newcomers to SQL. But, please keep reading. If you are reading this because of the words “Calendar Table” in the title, that means you probably don’t have one in place already. But if you work with dates in your data, calculating turnaround times or periodic totals, you should have one. Properly constructed calendar tables can turn complicated date calculations into simple, efficient queries. This article will show you how to create a sample Calendar table and then illustrate how easy queries become for otherwise odd questions.

**The Tally Table**

A Tally table (or numbers table) is a source of sequentially-numbered rows (usually 1 through 1,000,000 when implemented as a physical table.) It is a powerful tool for writing set-based code, instead of using procedural loops. One of the more well-known methods of generating these sequential numbers is attributed to Itzik Ben-Gan. For convenience sake, I created a view named vTally using this technique, and all the code examples will reference this view.

CREATE VIEW [dbo].[vTally]

AS

WITH L0 AS (SELECT 1 AS C UNION ALL SELECT 1) --2 rows

,L1 AS (SELECT 1 AS C FROM L0 AS A, L0 AS B)--4 rows

,L2 AS (SELECT 1 AS C FROM L1 AS A, L1 AS B)--16 rows

,L3 AS (SELECT 1 AS C FROM L2 AS A, L2 AS B)--256 rows

,L4 AS (SELECT 1 AS C FROM L3 AS A, L3 AS B)--65536 rows

,L5 AS (SELECT 1 AS C FROM L4 AS A, L4 AS B)--WHOA!

,Tally as (SELECT ROW\_NUMBER() OVER(ORDER BY (SELECT NULL)) as N from L5)

SELECT TOP(1000000) N FROM Tally

GO

How the above code works is the subject for another article. For now, *what* it does is more important. This code can generate 100,000,000 rows in a few seconds or 100,000 rows almost instantly. The rows will have one column [N] , which will contain sequential values between 1 and 100 million. We will use these rows, in conjunction with a function called DATEADD(), to increment a starting date and produce a result set of thousands of sequential dates.

**Basic Calendar**

The simplest form of Calendar table is a set of rows containing sequential dates. To create one, we simply pick a starting date and add a day to it over and over. The DATEADD() function is used to increment the starting date by a value of N-1 from vTally. DATEADD() works with all date/time datatypes and works with any increment of time from seconds up to years. It knows how many days are in each month and even handles adding a 29th day to February during leap years.

WITH Dates (N, CalendarDate) as

(SELECT TOP(1000) N, DATEADD(DAY,N-1, CONVERT(DATE,'1/1/2016'))

FROM vTally)

SELECT N, CalendarDate FROM Dates;

The simple calendar is fairly limited, but still might be used for identifying dates when employees didn’t clock in for work. Below we’ll test a three-day period of time (2/15/2016 – 2/17/2016), so we only have to pull the TOP(3) rows from vTally

CREATE TABLE #EmployeeTime (TimeID int primary key identity(1,1), EmployeeID int, WorkDate date, InTime time, OutTime time)

INSERT INTO #EmployeeTime

VALUES (1,'2016-02-15','8:00','17:00')

,(2,'2016-02-15','8:00','17:00')

,(3,'2016-02-15','8:00','17:00')

,(1,'2016-02-16','8:00','17:00')

,(3,'2016-02-16','8:00','17:00')

,(2,'2016-02-17','8:00','17:00')

,(3,'2016-02-17','8:00','17:00')

SELECT \* from #EmployeeTime

GO

WITH Dates (N,CalendarDate) as

(SELECT TOP(3) N, DATEADD(DAY,N-1, CONVERT(DATE,'2/15/2016'))

FROM vTally)

,Employees as (SELECT DISTINCT EmployeeID from #EmployeeTime)

SELECT e.EmployeeID, CalendarDate as MissedDate

FROM Dates d

CROSS APPLY Employees e

LEFT JOIN #EmployeeTime t ON t.WorkDate = d.CalendarDate

AND t.EmployeeID = e.EmployeeID

WHERE t.WorkDate IS NULL

order by EmployeeID, MissedDate

Produces the following result:

|  |  |
| --- | --- |
| EmployeeID | MissedDate |
| 1 | 2016-02-17 |
| 2 | 2016-02-16 |

This query using the basic calendar is somewhat useful, but it begs the question of whether or not the missing days fell on a weekend or a holiday. To perform more powerful and flexible queries, we need to expand the simple calendar and store it as a physical table.

**Expanded Calendar Table**

To get more use out of our calendar, we can use DATEPART(), and other date functions to break out various attributes of each CalendarDate.

CREATE TABLE #Holidays2016 (HolidayDate DATE, HolidayName VARCHAR(50));

INSERT INTO #Holidays2016 -- partial list of American holidays for 2016

VALUES ('1/1/2016', 'New Years Day')

, ('7/4/2016', 'Independence Day')

, ('12/25/2016', 'Christmas Day');

WITH Dates (N, CalendarDate) as

(SELECT TOP(366) N, DATEADD(DAY,N-1, CONVERT(DATE,'1/1/2016'))

FROM vTally)

SELECT N as DateID, CalendarDate

,DATEPART(day,CalendarDate) as CDay

,DATEPART(month,CalendarDate) as MonthNo

,DATEPART(year,CalendarDate) as YearNo

,DATEPART(DW,CalendarDate) as [DayofWeek]

,CASE WHEN EOMONTH(CalendarDate) = CalendarDate

THEN 1 ELSE 0 END as EndOfMonth

,CASE WHEN DATEPART(DW,CalendarDate) IN (1,7)

THEN 1 ELSE 0 END as Weekend

,CASE WHEN HolidayDate IS NOT NULL

THEN 1 ELSE 0 END as Holiday

,CASE WHEN DATEPART(DW,CalendarDate) NOT IN (1,7)

AND HolidayDate IS NULL

THEN 1 ELSE 0 END as WorkDay

FROM Dates d

LEFT JOIN #Holidays2016 h ON d.CalendarDate = h.HolidayDate

Produces:

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| DateID | CalendarDate | CDay | MonthNo | YearNo | DayofWeek | EndOfMonth | Weekend | Holiday | WorkDay |
| 1 | 2016-01-01 | 1 | 1 | 2016 | 6 | 0 | 0 | 1 | 0 |
| 2 | 2016-01-02 | 2 | 1 | 2016 | 7 | 0 | 1 | 0 | 0 |
| 3 | 2016-01-03 | 3 | 1 | 2016 | 1 | 0 | 1 | 0 | 0 |
| 4 | 2016-01-04 | 4 | 1 | 2016 | 2 | 0 | 0 | 0 | 1 |
| 5 | 2016-01-05 | 5 | 1 | 2016 | 3 | 0 | 0 | 0 | 1 |
| 6 | 2016-01-06 | 6 | 1 | 2016 | 4 | 0 | 0 | 0 | 1 |
| 7 | 2016-01-07 | 7 | 1 | 2016 | 5 | 0 | 0 | 0 | 1 |
| 8 | 2016-01-08 | 8 | 1 | 2016 | 6 | 0 | 0 | 0 | 1 |
| 9 | 2016-01-09 | 9 | 1 | 2016 | 7 | 0 | 1 | 0 | 0 |
| 10 | 2016-01-10 | 10 | 1 | 2016 | 1 | 0 | 1 | 0 | 0 |

Some of the most common date-oriented questions involve identifying the number of workdays between dates. The expanded calendar flags days as workdays, weekends, or holidays. On my system, each week begins with Sunday (day 1) and ends with Saturday (day 7), so the last CASE expression flags each day of the week (DW) as a weekend date when it’s day 1 or day 7.

A date is flagged as a holiday when a value for HolidayDate is returned from the LEFT JOIN to #Holidays2016 which contains a sample list of holidays. Holidays may fall on weekends and if so, both flags will be set. If a date is neither a holiday, nor a weekend day, it is a workday.

Before storing the expanded table, we use the CONVERT() function to produce smaller datatypes in order to minimize the disk space required. We’ll also add columns for more DATEPART() values, and a few unusual columns ([DoWAsc] and [LastDowInMonth) which will be explained in a moment.

CREATE TABLE #Holidays2016 (HolidayDate DATE, HolidayName VARCHAR(50));

INSERT INTO #Holidays2016

-- partial list of American holidays

VALUES ('1/1/2016', 'New Years Day')

, ('7/4/2016', 'Independence Day')

, ('12/25/2016', 'Christmas Day');

GO

WITH Dates (N, CalendarDate) as

(SELECT TOP(75000) N, DATEADD(DAY,N-1,CONVERT(DATE,'1/1/1900')) FROM vTally)

,ExpandedCalendar as

(SELECT N, CalendarDate

,CONVERT(smallint,DATEPART(day,CalendarDate)) as DayNo

,CONVERT(tinyint,DATEPART(week,CalendarDate)) as WeekNo

,CONVERT(tinyint,DATEPART(month,CalendarDate)) as MonthNo

,CONVERT(tinyint,DATEPART(quarter,CalendarDate)) as QuarterNo

,CONVERT(smallint,DATEPART(year,CalendarDate)) as YearNo

,CONVERT(tinyint,DATEPART(DW,CalendarDate)) as [DayofWeek]

,CONVERT(bit, CASE WHEN EOMONTH(CalendarDate) = CalendarDate

THEN 1 ELSE 0 END) as EndOfMonth

,CONVERT(bit, CASE WHEN DATEPART(DW,CalendarDate) IN (1,7)

THEN 1 ELSE 0 END) as Weekend

,CONVERT(bit, CASE WHEN HolidayDate IS NOT NULL

THEN 1 ELSE 0 END) as Holiday

,CONVERT(bit, CASE WHEN DATEPART(DW,CalendarDate) NOT IN (1,7)

AND HolidayDate IS NULL

THEN 1 ELSE 0 END) as WorkDay

FROM Dates d

LEFT JOIN #Holidays2016 h ON d.CalendarDate = h.HolidayDate)

SELECT \*,CONVERT(bit, CASE WHEN ROW\_NUMBER() OVER(PARTITION BY YearNo,MonthNo,[DayofWeek]

ORDER BY N DESC) = 1

THEN 1 ELSE 0 END) as LastDowInMonth

,CONVERT(tinyint, ROW\_NUMBER() OVER(PARTITION BY YearNo,MonthNo,[DayofWeek]

ORDER BY N)) as DoWAsc

,CONVERT(tinyint,((DayNo-1)/7)+1) as WeekNoAlt

INTO dbo.Calendar

FROM ExpandedCalendar;

GO

CREATE UNIQUE CLUSTERED INDEX PK\_Calendar on dbo.Calendar(CalendarDate);

GO

CREATE NONCLUSTERED INDEX X1\_Calendar on dbo.Calendar(YearNo,MonthNo);

GO

SELECT TOP(10) \*

FROM dbo.Calendar

WHERE CalendarDate >= '2016-1-1'

ORDER BY CalendarDate

The results of the above query are too wide to display here, so we’re just going to have to discuss the uses of the columns. YearNo, QuarterNo, MonthNo, and WeekNo are obviously useful for GROUP BY columns in summary queries. [DayOfWeek] allows selection of just certain days (Tuesdays and Thursdays) when required. The Weekend, Holiday, and Workday flags help filter out unwanted days from consideration. Now for the seemingly odd columns.

The column [DoWAsc] is used to number days with the same day of week within a month. This is handy for identifying descriptive dates such as “the third Sunday in March.”

The [LastDowInMonth] flag identifies dates like “the last Tuesday in April.”

The [WeekNoAlt] column is an alternative to the [WeekNo] column generated by DATEPART(). It represents the first 7 days starting with January 1st, the second 7 days etc. This probably bears illustrating.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| CalendarDate | DayofWeek | WeekNo | DayNo | WeekNoAlt |
| 2016-01-01 | 6 | 1 | 1 | 1 |
| 2016-01-02 | 7 | 1 | 2 | 1 |
| 2016-01-03 | 1 | 2 | 3 | 1 |
| 2016-01-04 | 2 | 2 | 4 | 1 |
| 2016-01-05 | 3 | 2 | 5 | 1 |
| 2016-01-06 | 4 | 2 | 6 | 1 |
| 2016-01-07 | 5 | 2 | 7 | 1 |
| 2016-01-08 | 6 | 2 | 8 | 2 |
| 2016-01-09 | 7 | 2 | 9 | 2 |
| 2016-01-10 | 1 | 3 | 10 | 2 |

As you can see, the [WeekNo] column starts with two-day period and increments each time [DayOfWeek] = 1. But [WeekNoAlt] counts 7 days starting with January 1st and increments every 8th day. The calculation ((DayNo-1)/7)+1) can be easily modified to count periods of 30 days (or 5 or 10 or 12 or whatever you need) by simply replacing the 7 with the number of your choice.

Other columns are possible, such as FiscalYear, FiscalQuarter, and FiscalMonth where an organization’s fiscal year does not follow the calendar year. Flags might be created for holidays of different nations or workdays of different nations. Don’t hesitate to add columns which are beneficial to you. The entire table can be recreated in seconds, and it will save you time and effort ever after.

Since I just mentioned holidays, I might as well address the fact that only three days in 2016 were flagged as Holidays. The next article will discuss creating a Holidays table that spans multiple years. Otherwise, dates have to be entered into a Holiday table manually.

**USING the Enhanced Calendar Table**

Now that you have a Calendar table in place, you can do basic calculations like finding the number of days between dates.

SELECT Count(\*)-1 as Date\_Diff   
FROM Calendar   
WHERE CalendarDate BETWEEN '2/27/2016' and '3/1/2016'

To be honest, there is already a DATEDIFF() function that can produce that same result. And there are long, nested, sometimes convoluted, strings of functions that answer more complicated date-related questions. But the enhanced calendar table can answer such questions easily.

-- How many workdays fall between June 27 and July 19th, 2016.

SELECT SUM(1) as Workdays

FROM dbo.Calendar

WHERE CalendarDate BETWEEN '6/27/2016' and '7/19/2016'

AND WorkDay = 1

-- Using a six-day workweek (including Saturdays),

-- how many workdays fall between June 27 and July 19

SELECT SUM(1) as Workdays

FROM dbo.Calendar

WHERE CalendarDate BETWEEN '6/27/2016' and '7/19/2016'

AND [DayOfWeek] > 1

AND Holiday = 0

-- Running a restaurant that closes on Sunday and Monday

-- how many workdays fall between June 27 and July 19

SELECT SUM(1) as Workdays

FROM dbo.Calendar

WHERE CalendarDate BETWEEN '6/27/2016' and '7/19/2016'

AND [DayOfWeek] not in (1,2)

AND Holiday = 0

-- We're scheduling meetings for the second half of 2016

-- Which dates fall on every 2nd and 4th Wednesday?

SELECT \*

FROM dbo.Calendar

WHERE YearNo = 2016

and MonthNo >6

and [DayofWeek] = 4

and DoWAsc in (2,4)

ORDER BY CalendarDate

-- We changed our mind about the meetings.

-- Which dates fall on the last Friday of every month?

SELECT \*

FROM dbo.Calendar

WHERE YearNo = 2016

and MonthNo >6

and [DayofWeek] = 5

and LastDowInMonth = 1

ORDER BY CalendarDate

-- Using the #employeeTime table, total up hours worked

-- For each employee per alternate week

SELECT t.EmployeeID, c.WeekNoAlt, SUM(DATEDIFF(hour,InTime,OutTime)-1) as Hours

FROM #employeetime t

JOIN dbo.Calendar c on t.WorkDate = c.CalendarDate

GROUP BY T.EmployeeID, WeekNoAlt

-- What time period is currently covered by this table>

SELECT MIN(CalendarDate) as [From], MAX(CalendarDate) as [To] FROM dbo.Calendar

|  |  |
| --- | --- |
| From | To |
| 1900-01-01 | 2105-05-05 |

-- How much disk space is all this costing me?

EXEC sp\_spaceused Calendar

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| name | Rows | reserved | Data | index\_size | Unused |
| Calendar | 75000 | 3472 KB | 2376 KB | 928 KB | 168 KB |

**Conclusion**

As the preceding examples illustrated, a properly constructed calendar table can simplify complicated calculations. Its columns can be easily expanded and the entire table regenerated in a matter of seconds. The flexibility it provides makes this table a must for any developer who has to work with data containing dates.