SQL Server DateTime Best Practices

By: Aaron Bertrand | Updated: 2017-11-15 | Comments (7) | Related: More > Dates

Problem

Dealing with date and time data in SQL Server can be tricky. I see a handful of the same issues present themselves over and over again out in the forums, and also have many interesting conversations during my speaking engagements throughout the community. Some of the symptoms can just be confusion and ambiguity, but in other cases, you can end up with incorrect output, unexpected errors, or data loss.

Solution

A few minor changes to how you handle dates in SQL Server can go a long way toward avoiding these issues, and in this tip, I'll address several best practices that I follow – usually after learning the hard way that the method I was using before was vulnerable to problems.

Avoid Regional Date Formats

Here in the United States, we love our format of m/d/y for expressing dates. In most other parts of the world, they prefer the more logical d/m/y format. You can argue that one makes more sense than the other, but we can all agree that neither format is safe to use as a string literal in SQL Server. There are other formats that are also unsafe to use as literals. Consider the following code; would you expect all of these dates to occur in February?

```
SET LANGUAGE us_english;
SELECT CONVERT(datetime, '02/09/2017');
SELECT CONVERT(datetime, '2017-02-09');
SELECT CONVERT(datetime, '2017-02-09 01:23:45.678');
```

Now, let's change the language setting to something else. Do you think all of these dates are still in February?

```
SET LANGUAGE français;
SELECT CONVERT(datetime, '02/09/2017');
SELECT CONVERT(datetime, '2017-02-09');
SELECT CONVERT(datetime, '2017-02-09 01:23:45.678');
```

All three dates are now interpreted by SQL Server as September 2nd instead of February 9th. That's not intuitive at all, but the truth is, this is how these three formats are interpreted in 24 of the 34 languages currently supported by SQL Server. Never mind that, if you don't know the intention of the user (and you can't base this solely on their language settings or geography), you don't even know whether they meant September 2nd or February 9th.

The following formats are the only ones that are safe from being interpreted based on language or regional settings, and will yield February 9th regardless of language:

```
SET LANGUAGE français;
SELECT CONVERT(datetime, '20170209');
SELECT CONVERT(datetime, '2017-02-09T01:23:45.678');
SELECT CONVERT(datetime, '20170209 01:23:45.678');
```

The lesson here is that, while you are perfectly able to use formats like yyyy-mm-dd on your local machine or in an isolated environment, you should consistently use one of the safer formats – because you never know where your code will eventually be deployed, or who will learn from reviewing your application. This is even more true if you are writing code samples for blog posts or forum answers, where people will certainly take your samples, deploy them in their own environments, and come looking for you when things go wrong.

Last but not least, don't ever let users enter free-text date strings into a form, because again, you can never be sure if a user meant September 2nd or February 9th. There are hundreds of calendar controls and other libraries that will let your users pick a date, and then you can have ultimate control over the string format before it gets anywhere near SQL Server.

Stay away from shorthand

On any given day, I can pick up a question on Stack Overflow involving SQL Server and DATEPART(), DATENAME(), or DATEADD(), and find usage of datepart shorthand like d, dd, m, and yyyy. I often wonder why people will use shorthand like this, and I have to assume it's because they've only every used one of those three, and not some of the others – which are less intuitive.

The dateparts d and dd are obviously day, and it would be hard to confuse them with anything else, but it would be even harder to confuse if you just typed DAY. The abbreviation for month, m, can be a little trickier, and I catch people all the time in a quiz I run in my live sessions, making them think twice about whether

this actually stands for minute. “Shortening” year to yyyy makes really no sense to me at all – in my sessions I joke that people are saving all the productivity that it takes to move your fingers between separate keys on the keyboard.

There are other abbreviations that cause a lot more confusion, and that I catch people with during my little quiz. Let’s see how you do, and be honest: what would the output be for the following query? I’ll even give you multiple choice options:

```
-- Christmas 2017 falls on a Monday
SELECT [w] = DATEPART(w, '20171225'),   --  (a) 53   (b)
2      (c) 1
       [y] = DATEPART(y, '20171225');   --  (a) 17   (b)
2017   (c) 359
```

Surprising results, right? Depending on your DATEFIRST setting, [w] is either going to be 1 or 2, since this abbreviation is actually for day of week, not week number. If you want to find the week, spell out WEEK (or ISO_WEEK). Similarly, y stands for day of year, not year, so both 17 and 2017 are incorrect. Again, spell our YEAR (not YYYY) if that’s what you’re after, and to be consistent, always spell out the datepart component you mean – even in cases where it’s not ambiguous.

Another form of shorthand you need to be aware of is “date math.” A lot of people write code like this, to find yesterday, tomorrow, or a week ago:

```
SELECT GETDATE()-1, GETDATE()+1, GETDATE()-7;
```

Now suppose you have a table full of orders, and you have a bunch of code that references columns in that table. Triggers, check constraints, and procedures, all that make calculations against things like order dates and ship dates. You might have something like this:

```
UPDATE dbo.table SET ExpectedShipDate = OrderDate + 4;
```

Now, try that if you change OrderDate to date; you’ll get:

```
Msg 206, Level 16, State 2, Line 34
Operand type clash: datetime2 is incompatible with int
```

This is because the new types introduced in 2008 (date, datetime2, datetimeoffset) do not support this ambiguous calculation. The lesson? Always

use explicit DATEADD() syntax (and good luck finding all the +n/-n references in your codebase).

Don't use BETWEEN

When I ask an audience if they use BETWEEN for date range queries, I'm always overwhelmed at the number of hands that shoot up. For me, BETWEEN is fine for querying a range of integers, as there's no ambiguity about either end of the range. "Between one and ten" means 1 to 10, inclusive. What does "between February and March" mean? What does "between February and the end of February" mean? What is "the end of February," exactly, even if you ignore leap year problems? A lot of people will say it's 3 milliseconds before the beginning of March:

```
SELECT DATEADD(MILLISECOND, -3, '20170301');   -- 20170228
23:59:59.997
```

So they then run a query that asks for sales or events that happened "between February 1st and 3 milliseconds before midnight on March 1st." This logic is okay, as long as you only ever deal with columns, parameters, and variables that use the datetime type. For other types (including future changes you can't predict today), this becomes a problem.

In the following code, we have a table with seven sales events, six of them happening in February. Our goal is to count all of the rows in February using BETWEEN, but let's say we have no control over the data types of the parameters. To demonstrate this unknown, we make a simple stored procedure that takes our start date (as date), and then our end date as multiple individual data types.

```
CREATE TABLE dbo.SalesOrders
(
  OrderDate datetime2
);
INSERT dbo.SalesOrders(OrderDate)    -- 6 rows in February, 1
row in March
  VALUES ('20170201 00:00:00.000'),
         ('20170211 01:00:00.000'),('20170219 00:00:00.000'),
         ('20170228 04:00:00.000'),('20170228 13:00:27.000'),
         ('20170228 23:59:59.999'),('20170301 00:00:00.000');
GO
CREATE PROCEDURE dbo.GetMonthlyOrders
  @start             date,
```

```
  @end_datetime      datetime,
  @end_smalldatetime smalldatetime,
  @end_date          date
AS
BEGIN
  SET NOCOUNT ON;
  SELECT
    [datetime] = (SELECT COUNT(*) FROM dbo.SalesOrders
      WHERE OrderDate BETWEEN @start AND @end_datetime),
    [smalldatetime] = (SELECT COUNT(*) FROM dbo.SalesOrders
      WHERE OrderDate BETWEEN @start AND @end_smalldatetime),
    [date] = (SELECT COUNT(*) FROM dbo.SalesOrders
      WHERE OrderDate BETWEEN @start AND @end_date);
END
GO
DECLARE
  @start datetime = '20170201',
  @end   datetime = DATEADD(MILLISECOND, -3, '20170301');

EXEC dbo.GetMonthlyOrders
  @start             = @start,
  @end_datetime      = @end,
  @end_smalldatetime = @end,
  @end_date          = @end;
```

The counts should be 6 in each case, but they aren't:

```
datetime    smalldatetime date
----------- ------------- -----------
5           7             3
```

Why? Well, the first one cuts off at 23:59:59.997, but notice that the datetime2 column is capable of holding a row at 23:59:59.998 or 23:59:59.999 – whether this is statistically likely is a different issue. The second one, when converting to smalldatetime, actually rounds up, so now the report for February includes any rows from March 1st at midnight (and this could be significant). The last one, when converting to date, rounds **down**, so all rows from the last day of the month after midnight are excluded. (As an aside, the same thing happens if you use EOMONTH().) None of these reports would be right, and the only thing that changed was an innocuous input parameter.

The problem here isn't so much with the use of the BETWEEN keyword explicitly; after all, you could just as easily write BETWEEN as a closed-ended

range (>= and <=), and end up with the same incorrect results.

The problem really is trying to hit a moving target: the "end" of a period. Instead of trying to find the end of a period, you can simply find the beginning of the **next** period, and use an open-ended range. For example, all of the above queries were written like this (pseudo):

```
>= Beginning of February and <= End of February
```

You could, instead, write them as follows:

```
>= Beginning of February and < Beginning of March
```

In fact, if the stored procedure is designed to get monthly counts only, you could rewrite it to only take a single parameter:

```
CREATE PROCEDURE dbo.GetMonthlyOrders2
  @month date
AS
BEGIN
  SET NOCOUNT ON;
  -- ensure always at start of month
  SET @month = DATEFROMPARTS(YEAR(@month), MONTH(@month), 1);

  SELECT [one param] = COUNT(*)
      FROM dbo.SalesOrders
      WHERE OrderDate >= @month
        AND OrderDate <  DATEADD(MONTH, 1, @month);
END
GO

-- can pass any date/time within the desired month
DECLARE @month datetime = '20170227 02:34:12.762';
EXEC dbo.GetMonthlyOrders2 @month = @month;
```

Results:

```
one param
-----------
6
```

In this case, no matter what data types you pass into the procedure, you're always guaranteed to get all the rows from February and **only** the rows from

February. So always use an open-ended range to prevent erroneously including or excluding rows. As a bonus, it's much less complex to find the beginning of the next period than the end of the current period.

Next Steps

Just try to keep these little issues in mind when working with date and time data in SQL Server. "Works on my machine" can get you far enough in small, localized projects, but beyond that, it can lead to big problems. There are some other tips and resources to check out, too: