More on PIVOT

You can use PIVOT even when you don't know the list of possible values, and you can unpivot in order to normalize unnormalized data.

Tamar E. Granor, Ph.D.

In my last article, I explored the PIVOT keyword that lets you create crosstabs in SQL Server using a single query. This article looks at how to use PIVOT even when you don't know the list of values in the pivot column, and covers the UNPIVOT keyword that lets you undo a pivot (in some cases) and, more broadly, normalize data.

I'll start with a quick review. A crosstab is a result table or cursor where the set of columns is based on data values in the source. In SQL Server, you create crosstabs using the PIVOT keyword. You need at least three columns to do a pivot: one to specify the rows, one to specify the columns, and one that contains the data to be aggregated.

Listing 1 shows a simple example. It produces one row for each salesperson and one column for each year from 2011 to 2014. The data at the intersection of the row and the column is the total sales (in dollars) for that salesperson for that year. Partial results are shown in **Figure 1**; the code is included in this month's downloads as SalesPersonAnnualSalesCTE.SQL.

Listing 1. This simple example of PIVOT computes sales by salesperson by year.

```
WITH SalesByYear
  (SalesPersonID, SalesYear, SubTotal)
AS
(SELECT SalesPersonID, YEAR(OrderDate),
        SubTotal
   FROM Sales.SalesOrderHeader
   WHERE SalesPersonID IS NOT NULL)

SELECT *
  FROM SalesByYear
  PIVOT(SUM(SubTotal)
    FOR SalesYear
     IN ([2011], [2012], [2013], [2014]))
    AS TotalSales
  ORDER BY SalesPersonID
```

My last article shows how to do more complex pivots, including using multiple fields to specify the rows, and doing more than one aggregation.

SalesPersonID	2011	2012	2013	2014
274	28926.2465	453524.5233	431088.7238	178584.3625
275	875823.8318	3375456.8947	3985374.8995	1057247.3786
276	1149715.3253	3834908.674	4111294.9056	1271088.5216
277	1311627.2918	4317306.5741	3396776.2674	1040093.4071
278	500091.8202	1283569.6294	1389836.8101	435948.9551
279	1521289.1881	2674436.3518	2188082.7813	787204.4289
280	648485.5862	1208264.3834	963420.5805	504932.044
281	967597.2899	2294210.5506	2387256.0616	777941.6519
282	1175007.4753	1835715.8705	1870884.182	1044810.8277
283	599987.9444	1288068.7236	1351422.362	490466.319
284	NULL	441639.5961	1269908.9235	600997.1704
285	NULL	NULL	151257.1152	21267.336
286	NULL	NULL	836055.1236	585755.8006
287	NULL	116029.652	560091.7843	56637.7478

Figure 1. The query in Listing 1 produces this result, showing each salesperson's annual sales.

Handling unknown data

The biggest issue to me with the previous example and all of the other examples in my last article is the need to actually list out the values in the field that determines the result columns. It means that you have to know what data to expect and more importantly, that query results may be wrong if there's data you didn't expect.

Fortunately, there's a solution, using dynamic SQL. You can store a query in a string and then execute it; it's similar to using the & macro operator or the ExecScript() function in VFP. Full details on dynamic SQL are beyond the scope of this article, but you don't have to know much to use it for pivoting without knowing the values of the field you want to pivot on. I'll cover it in the context of the example above, seeing total sales by salesperson for each year; the complete code is in **Listing 2** and included as SalesPersonAnnualSalesDynamic.SQL in this month's downloads.

You need two variables, one to hold the list of values and one to hold the query you construct. In the example, they're called @years and @query. Step one is to run a query to store the list of distinct values in the first variable. It takes advantage of SQL's ability to populate a variable via a query. The QUOTENAME() function converts

to character (actually varchar) and adds delimiters to make sure the result can serve as an identifier. In this example, @years is assigned the value '[2011],[2012],[2013],[2014]', that is, exactly the list we've been hard-coding in the previous examples.

Step two is to build a string that contains the query we want to execute. The SET command here simply fills @query with the same query we've been using, except that instead of hard-coding the list of years, we plug in the computed value in @years.

Finally, we call the built-in stored procedure sp_executesql to execute the query we've built. The first parameter to sp_executesql is always the statement to execute. It can accept additional parameters to be used in executing that statement, but in this case, we don't need any.

Listing 2. When you don't know the list of values for the column you want to pivot on, you can retrieve a list of values and use dynamic SQL to do the actual pivot.

```
DECLARE @query AS NVARCHAR(max);
DECLARE @years AS NVARCHAR(max);

-- Get the list of years
WITH DistinctYear (nYear)
AS
(SELECT DISTINCT YEAR(OrderDate)
  FROM Sales.SalesOrderHeader)

SELECT @years = ISNULL(@years + ', ', '')
       + QUOTENAME(nYear)
  FROM DistinctYear
  ORDER BY nYear;

-- Build the query including the list of years
SET @query =
  'WITH SalesByYear
   AS
   (SELECT SalesPersonID,
           YEAR(OrderDate) AS SalesYear,
           SubTotal
      FROM Sales.SalesOrderHeader
      WHERE SalesPersonID IS NOT NULL)

   SELECT *
    FROM SalesByYear
     PIVOT(SUM(SubTotal)
       FOR SalesYear IN (' + @years + '))
       AS TotalSales
    ORDER BY SalesPersonID';

-- Run the query
EXEC sp_executesql @query;
```

Not surprisingly, this example produces the same results as the one where the years are hard-coded; they're shown in Figure 1.

The query that produces the list of years probably deserves a little more attention, since it does something you can't do with a single query in VFP. First, as noted above, rather than storing its result in some kind of table, it puts the result into a variable (@years); that's what `SELECT @years =` does. The more interesting piece is that @years appears on the right-hand side of the equals sign, as well. So the result is built up one record at a time. For the first record, @years is null, and `ISNULL(@years + ', ', '')` returns the empty string. After that, it returns the string so far with a trailing comma and space. For each record, we then add the bracketed version of the year.

As noted in my last article, the one big advantage of having to list each value in PIVOT's IN section is that you can limit the result to a specified subset of the data. To do that when generating the list of values dynamically, there has to be a rule you can apply in the query that assembles the list of values. For example, if you're only interested in sales in 2013 and later, you can add a WHERE clause to the CTE of the query that populates @years, as in Listing 3. If you want to see sales for the last three years, you could specify YEAR(GETDATE())-2 rather than 2013 in the WHERE clause.

Listing 3. You can limit the column list by filtering the query that collects the list of values.

```
-- Get the list of years
WITH DistinctYear (nYear)
AS
(SELECT DISTINCT YEAR(OrderDate)
  FROM Sales.SalesOrderHeader
  WHERE YEAR(OrderDate) >= 2013)

SELECT @years = ISNULL(@years + ', ', '')
       + QUOTENAME(nYear)
  FROM DistinctYear
  ORDER BY nYear;
```

Undoing PIVOT

SQL Server lets you undo pivots through the UNPIVOT keyword, though you can't always get back to the original data (because of the aggregation performed as part of the pivot process). The syntax for UNPIVOT is quite similar to the syntax for PIVOT; it's shown in Listing 4.

Listing 4. The syntax for UNPIVOT is pretty much the same as for PIVOT, except there's no aggregation function involved.

```
SELECT <row identifier columns>,
       [ <column identifier column>, ]
       <column to extract>
FROM <source table>
UNPIVOT
(<column to extract>
  FOR [<generic name for group of columns>]
    IN (<list of columns to unpivot>)
) AS <alias>
```

For UNPIVOT, you generally list out the columns you want. Depending on the data you're unpivoting, you may or may not want to turn the column names into data. In addition, the columns you want in the result can be listed in any order.

The key items are to provide a name for the column to contain the data you're unpivoting (shown as "<column to extract>" in the syntax diagram), and to provide a list of the columns that contain data following the IN keyword. The latter is the same information you provide to PIVOT, but here

it's the column names rather than the values in a particular column. The item that follows the FOR keyword is simply a name to let you refer to that list as a group; that lets you include them as data in the result. As you'll see later in this article, it also allows you to operate on those column names as data.

Let's start with a simple example that reverses (sort of) the annual sales pivot. Assume that rather than simply returning the result of the query in Listing 1, we've stored it in a temporary table called #SalesByYearCT. Listing 5 shows how to unpivot that result and get one row per salesperson per year; it's included in this month's downloads as UnpivotSales.SQL. The query lists the three fields we want in the result: the salesperson's ID, the year, and the total sales for that salesperson in that year. Inside the UNPIVOT section, we first specify that the data in the columns we're unpivoting should go into a column called AnnualSales. Then, we indicate that those columns are the ones named [2011], [2012], [2013] and [2014]. The names SalesYear is assigned to that group of fields, and matches up with the name in the field list, so that the year appears in the result. The alias for the UNPIVOT (SalesPersonYear) is required, but doesn't actually add anything.

Listing 5. The query sort of unpivots the result of the query in Listing 1.

```
SELECT SalesPersonID, SalesYear, AnnualSales
  FROM #SalesByYearCT
  UNPIVOT (AnnualSales
    FOR SalesYear
    IN ([2011], [2012], [2013], [2014]))
    AS SalesPersonYear;
```

Figure 2 shows partial results and explains why I said this query "sort of" reverses the original. The query in Listing 1 starts with the raw sales data, one record per sales order. But the original query aggregates those records. UNPIVOT has no way to disaggregate that aggregated data.

SalesPersonID	SalesYear	AnnualSales
284	2012	441639.5961
284	2013	1269908.9235
284	2014	600997.1704
278	2011	500091.8202
278	2012	1283569.6294
278	2013	1389836.8101
278	2014	435948.9551
281	2011	967597.2899
281	2012	2294210.5506
281	2013	2387256.0616
281	2014	777941.6519
275	2011	875823.8318
275	2012	3375456.8947
275	2013	3985374.8995
275	2014	1057247.3786

Figure 2. The unpivoted annual sales have one record per salesperson per year.

Using UNPIVOT to normalize data

UNPIVOT is useful for normalizing data. It's quite common to find databases that are not normalized, particularly tables that use multiple columns for essentially the same data. UNPIVOT lets you gather that data into a single column.

The AdventureWorks database doesn't have any such examples. So to demonstrate this use of UNPIVOT, we'll have to create our own example data. (These examples are inspired by and adapted from Aaron Bertrand's article on UNPIVOT at http://tinyurl.com/grlkfyf.)

One of the most common examples of multiple columns rather than normalized data is the use of separate fields for different phone numbers, that is, having fields named Phone1, Phone2, etc., or Home, Mobile and Work. Listing 6 creates and populates a simplified table that uses the latter approach. (To make it easy to see that the normalization code works, the phone numbers here use a pattern. The exchange for all home numbers is 555; for mobile numbers, it's 666; and for work numbers, it's 777. In addition, each number for an individual has the same last four digits. None of this makes a difference in how the code works; it simply makes checking the results easier.) Figure 3 shows the data in the #Person table.

Listing 6. This code creates and populates a temporary table that uses separate named fields for home, mobile and work phone numbers.

```
CREATE TABLE #Person
(First varchar(15), Last varchar(20),
 Home varchar(10), Mobile varchar(10),
 Work varchar(10))
INSERT INTO #Person VALUES
  ('Jane', 'Smith',
   '5555551234', '5556661234', '5557771234'),
  ('Andrew', 'Jones',
   '5555557890', '5556667890', '5557777890'),
  ('Deborah', 'Cohen', NULL,
   '5556667474', '5557777474');
```

First	Last	Home	Mobile	Work
Jane	Smith	5555551234	5556661234	5557771234
Andrew	Jones	5555557890	5556667890	5557777890
Deborah	Cohen	NULL	5556667474	5557777474

Figure 3. This approach to storing phones is not normalized and means the data structure has to change to accommodate each new type of phone.

With the proliferation of phone numbers for an individual, it makes much more sense to use a separate table to contain all phone numbers with one record per phone per person. The query in Listing 7 extracts the data from the #Person

table in the desired format. The result, shown in Figure 4, includes a column to indicate the type of phone; it's populated based on the column names in the original. Note also that there's no home phone record for Deborah Cohen, since the Home field was NULL. This month's downloads include NormalizePhones.SQL, which creates the temporary #Person table and has the query to normalize the data.

Listing 7. This query normalizes the phone data, using the column names to indicate the type of phone.

```
SELECT First, Last, PhoneType, Phone
  FROM #Person
  UNPIVOT (Phone
    FOR PhoneType
    IN (Home, Mobile, Work)) AS Phones;
```

First	Last	PhoneType	Phone
Jane	Smith	Home	5555551234
Jane	Smith	Mobile	5556661234
Jane	Smith	Work	5557771234
Andrew	Jones	Home	5555557890
Andrew	Jones	Mobile	5556667890
Andrew	Jones	Work	5557777890
Deborah	Cohen	Mobile	5556667474
Deborah	Cohen	Work	5557777474

Figure 4. The normalized phone data is much more flexible

The PhoneType column in this result is optional. If, for some reason, you don't want to include it, you can just leave it out of the field list, as in Listing 8. In this example, it's hard to see why you'd want to do that, but if the original columns were simply Phone1, Phone2 and Phone3, it would make sense.

Listing 8. This version of the query normalizes the phone data without including the phone type information from the column name.

```
SELECT First, Last, Phone
  FROM #Person
  UNPIVOT (Phone
    FOR PhoneType
    IN (Home, Mobile, Work)) AS Phones;
```

First	Last	Phone
Jane	Smith	5555551234
Jane	Smith	5556661234
Jane	Smith	5557771234
Andrew	Jones	5555557890
Andrew	Jones	5556667890
Andrew	Jones	5557777890
Deborah	Cohen	5556667474
Deborah	Cohen	5557777474

Figure 5. You can normalize without including the name of the column where the data originated.

Normalizing multiple columns

What if rather than a single set of columns that need to be normalized, you have multiple related sets? Again, telephone numbers provide a simple example. Suppose that instead of having one column per phone number, named with the phone type, you have a pair of columns for each phone number, one indicating the type and the second containing the phone number. As long as the column names follow a pattern, UNPIVOT lets you normalize without losing any of the data.

Listing 9 creates and populates a different version of the #Person table. In this version, there are three pairs of columns, named Phone*n* and Phone*n*Type. Each holds one phone number, along with its type, and a person can have up to three. The data here is the same as in the previous version of the table, except that I've consciously changed the order of the phone numbers in one record. As in the previous example, Deborah Cohen has no home number. Figure 6 shows the table's contents.

Listing 9. This version of the #Person table uses two columns for each phone number, to specify the number and the type.

```
CREATE TABLE #Person
  (First varchar(15), Last varchar(20),
   Phone1 varchar(10), Phone1Type varchar(6),
   Phone2 varchar(10), Phone2Type varchar(6),
   Phone3 varchar(10), Phone3Type varchar(6));
INSERT INTO #Person VALUES
  ('Jane', 'Smith',
   '5555551234', 'Home',
   '5556661234', 'Mobile',
   '5557771234', 'Work'),
  ('Andrew', 'Jones',
   '5556667890', 'Mobile',
   '5557777890', 'Work',
   '5555557890', 'Home');
INSERT INTO #Person
  (First, Last, Phone1, Phone1Type,
   Phone2, Phone2Type) Values
  ('Deborah', 'Cohen',
   '5556667474', 'Mobile',
   '5557777474', 'Work');
```

First	Last	Phone1	Phone1Type	Phone2	Phone2Type	Phone3	Phone3Type
Jane	Smith	5555551234	Home	5556661234	Mobile	5557771234	Work
Andrew	Jones	5556667890	Mobile	5557777890	Work	5555557890	Home
Deborah	Cohen	5556667474	Mobile	5557777474	Work	NULL	NULL

Figure 6. In this version of the #Person table, each phone number has separate columns for number and type.

Surprisingly, given that doing multiple pivots requires separate queries (as discussed in my last article), you can do multiple unpivots in a single query. That's what's required to normalize this data, as shown in Listing 10 (included as NormalizePhoneAndType.SQL in this month's downloads); the result is shown in Figure 7.

Listing 10. You can unpivot multiple related fields in a single query.

```
WITH AllPhones AS
```

```
(SELECT First, Last, Phone, PhoneType,
        RIGHT(Phones, 1) AS nPhone,
        SUBSTRING(PhoneTypes, 6, 1)
          AS nPhoneType
  FROM #Person
    UNPIVOT
     (Phone FOR Phones
        IN (Phone1, Phone2, Phone3))
        AS PhoneList
    UNPIVOT
     (PhoneType FOR PhoneTypes
        IN (Phone1Type, Phone2Type,
            Phone3Type))
        AS PhoneTypeList)

SELECT First, Last, Phone, PhoneType
  FROM AllPhones
  WHERE nPhone = nPhoneType;
```

First	Last	Phone	PhoneType
Jane	Smith	5555551234	Home
Jane	Smith	5556661234	Mobile
Jane	Smith	5557771234	Work
Andrew	Jones	5556667890	Mobile
Andrew	Jones	5557777890	Work
Andrew	Jones	5555557890	Home
Deborah	Cohen	5556667474	Mobile
Deborah	Cohen	5557777474	Work

Figure 7. Using a pair of UNPIVOTs and a little more work, we can normalize a table involving multiple related fields.

The actual unpivots are done in a CTE. First, we unpivot the phone number fields, specifying Phone as the field to hold the data, Phones as the collective name for the existing fields and listing them as Phone1, Phone2, and Phone3. The second unpivot does the same thing for the phone types, indicating that the data goes into PhoneType, the collective name for the existing fields is PhoneTypes and specifying the list of those fields as Phone1Type, Phone2Type and Phone3Type.

However, the pair of UNPIVOT clauses operate like a cross-join (also known as a Cartesian join); each item unpivoted from the first is joined to each item unpivoted from the second. We need a way to match up the corresponding numbers and types. The two extra fields in the CTE give us what we need to do that. They're also the reason we can only do this is there's a pattern to the field names. The first pulls out the digit from the phone number field, while the second does the same for the phone type field. Each of those expressions uses the collective name for the list of fields as the way to find the name of the original field for that data item.

Figure 8 shows partial results from running just the query in the CTE. For Jane Smith, with three phone numbers, there are 9 records, one for each match of each phone number with each phone type. But the only valid records are those where nPhone and nPhoneType match.

First	Last	Phone	PhoneType	nPhone	nPhoneType
Jane	Smith	5555551234	Home	1	1
Jane	Smith	5555551234	Mobile	1	2
Jane	Smith	5555551234	Work	1	3
Jane	Smith	5556661234	Home	2	1
Jane	Smith	5556661234	Mobile	2	2
Jane	Smith	5556661234	Work	2	3
Jane	Smith	5557771234	Home	3	1
Jane	Smith	5557771234	Mobile	3	2
Jane	Smith	5557771234	Work	3	3
Andrew	Jones	5556667890	Mobile	1	1
Andrew	Jones	5556667890	Work	1	2
Andrew	Jones	5556667890	Home	1	3
Andrew	Jones	5557777890	Mobile	2	1
Andrew	Jones	5557777890	Work	2	2
Andrew	Jones	5557777890	Home	2	3

Figure 8. The two UNPIVOTs in the CTE in Listing 10 do a cross-join of the unpivoted data. The additional fields nPhone and nPhoneType help us see where each item originated, so that the main query can filter out the mismatches.

With that data available, the main query in Listing 10 keeps only those records for corresponding phone numbers and phone types.

Unpivoting unknown columns

It seems less likely to me that you might need to unpivot without knowing the column names, but I guess there can situations where data comes in from multiple sources with different numbers of unnormalized columns. Fortunately, you can build the list of columns for the unpivot and use dynamic SQL here, too. As in the matched columns example, doing so depends on the relevant column names following a pattern.

The key to doing this is to use the system columns table (that is, a system table named Columns that contains information about the columns in the database) to extract the relevant list. The columns of interest in that table are Name, which contains the field name, and Object_ID, which identifies the table to which the field belongs. As with the dynamic pivot, we need QUOTENAME() to wrap the field names to ensure they're valid. Listing 11 shows the code to create and run the query from Listing 10; it's included in this month's downloads as DynamicUnpivot.SQL. (This example is inspired by and adapted from Aaron Bertrand's article at http://tinyurl.com/z464ll9.)

Listing 11. You can build the list of fields and use dynamic SQL to unpivot when you're not sure how many fields you have.

```
-- Use variables here. In production,
-- these might be parameters
DECLARE @table AS NVARCHAR(max) =
  N'tempdb..#Person';
DECLARE @phonenames AS NVARCHAR(max) =
  N'Phone[0-9]';
DECLARE @typenames AS NVARCHAR(max) =
  N'Phone[0-9]Type';

DECLARE @phonecols AS NVARCHAR(max);
DECLARE @typecols AS NVARCHAR(max);
DECLARE @query AS NVARCHAR(max);

SELECT @phonecols =
       ISNULL(@phonecols + ', ', '') +
       QUOTENAME(Name)
  FROM tempdb.sys.Columns
  WHERE Object_ID = OBJECT_ID(@table)
    AND Name LIKE @phonenames;

SELECT @typecols =
       ISNULL(@typecols + ', ', '') +
       QUOTENAME(Name)
  FROM tempdb.sys.columns
  WHERE object_id = OBJECT_ID(@table)
    AND name LIKE @typenames;

SET @query =
'WITH AllPhones AS
  (SELECT First, Last, Phone, PhoneType,
          RIGHT(Phones, 1) AS nPhone,
          SUBSTRING(PhoneTypes, 6, 1)
            AS nPhoneType
  FROM #Person
    UNPIVOT
      (Phone FOR Phones
       IN (' + @phonecols +')) AS PhoneList
    UNPIVOT
      (PhoneType FOR PhoneTypes
       IN (' + @typecols +'))
       AS PhoneTypeList)

SELECT First, Last, Phone, PhoneType
  FROM AllPhones
    WHERE nPhone = nPhoneType';

-- Run the query
EXEC sp_executesql @query;
```

First, we declare several variables that hold information about the query we need to build. As the comment says, in real code, these might be parameters to a stored procedure. The first, @table, identifies the table containing the data we want to normalize. Because we're working with a temporary table here, it needs the "tempdb.." prefix. The next two, @phonenames and @typenames, specify the pattern for the two sets of columns we're interested in.

Next, we declare variables to hold the two lists of columns and the query to be executed.

Then, two consecutive, quite similar queries populate the @phonecols and @typecols variables by extracting the list from the system Columns table. (Here again, we specify tempdb because we're interested in the columns of a temporary table. For a permanent table, you can omit that.) The strategy for building each list is the same as in the dynamic pivot example; start with a null string and build it up. The OBJECT_ID() function looks up the unique identifier for the table of interest, so that we consider only fields from that table.

Next, we populate the @query variable with the query of interest, plugging in the field names from the two variables @phonecols and @typecols.

Finally, we call sp_executesql to run the query we've built. Not surprisingly, in this case, we get the same results as for the static query, shown in Figure 7.

Summing up

You may not need to pivot or unpivot data every day, but when you do, SQL Server gives you a straightforward way to do so. The ability to use such queries even when you don't know all the data or all the fields you're interested in makes this feature even more useful.

Author Profile

Tamar E. Granor, Ph.D. is the owner of Tomorrow's Solutions, LLC. She has developed and enhanced numerous Visual FoxPro applications for businesses and other organizations. Tamar is author or co-author of a dozen books including the award winning Hacker's Guide to Visual FoxPro, Microsoft Office Automation with Visual FoxPro *and* Taming Visual FoxPro's SQL. *Her latest collaboration is* VFPX: Open Source Treasure for the VFP Developer, *available at www.foxrockx.com. Her other books are available from Hentzenwerke Publishing (www.hentzenwerke.com). Tamar was a Microsoft Support Most Valuable Professional from the program's inception in 1993 until 2011. She is one of the organizers of the annual Southwest Fox conference. In 2007, Tamar received the Visual FoxPro Community Lifetime Achievement Award. You can reach her at tamar@thegranors.com or through www.tomorrowssolutionsllc.com.*