Using SQL to Solve Common Problems

Tamar E. Granor
Tomorrow's Solutions, LLC
Voice: 215-635-1958
Website: www.tomorrowssolutionsllc.com
Email: tamar@tomorrowssolutionsllc.com

Whether you're working in VFP, SQL Server, or MySQL, some problems come up pretty regularly. In this session, we'll look at how to solve some frequent problems using SQL. In some cases, we'll see how differences in what portion of the SQL standard is implemented make a problem easier to solve in some variants than others.

Introduction

The SQL language is powerful. A single SQL query can often replace tens or hundreds of lines of code. Knowing how to use SQL can make your code more readable and maintainable. In this session, we'll look at how to use SQL for some of the problems that come up frequently in applications, and see how different SQL versions handle them.

For each problem, we'll look at how to solve it in VFP's fairly limited SQL, in SQL Server, and in MySQL. The materials for this session include a separate folder for each, containing the examples for that version.

To make the results easier to compare, the examples use the Chinook database, which was created specifically to make such comparisons easier. You can download code to create Chinook for SQL Server and MySQL at https://github.com/lerocha/chinook-database. The materials for this session contain code to create a VFP version of Chinook as Chinook_VisualFoxPro_AutoincrementPKs.PRG, as well as the VFP version of the database. (Because I ran into some issues running the downloaded code to create Chinook for SQL Server and MySQL, the materials also include the code for those.)

Chinook contains information for a fictitious online music-selling service. It tracks artists, albums and tracks as well as customers and invoices. **Figure 1** shows the database structure; the diagram was generated by SQL Server Management Studio 2014.

The examples in this session were tested in VFP 9 SP2, SQL Server 2014 and 2017, and MySQL 5.7 and (where appropriate) 8.0.

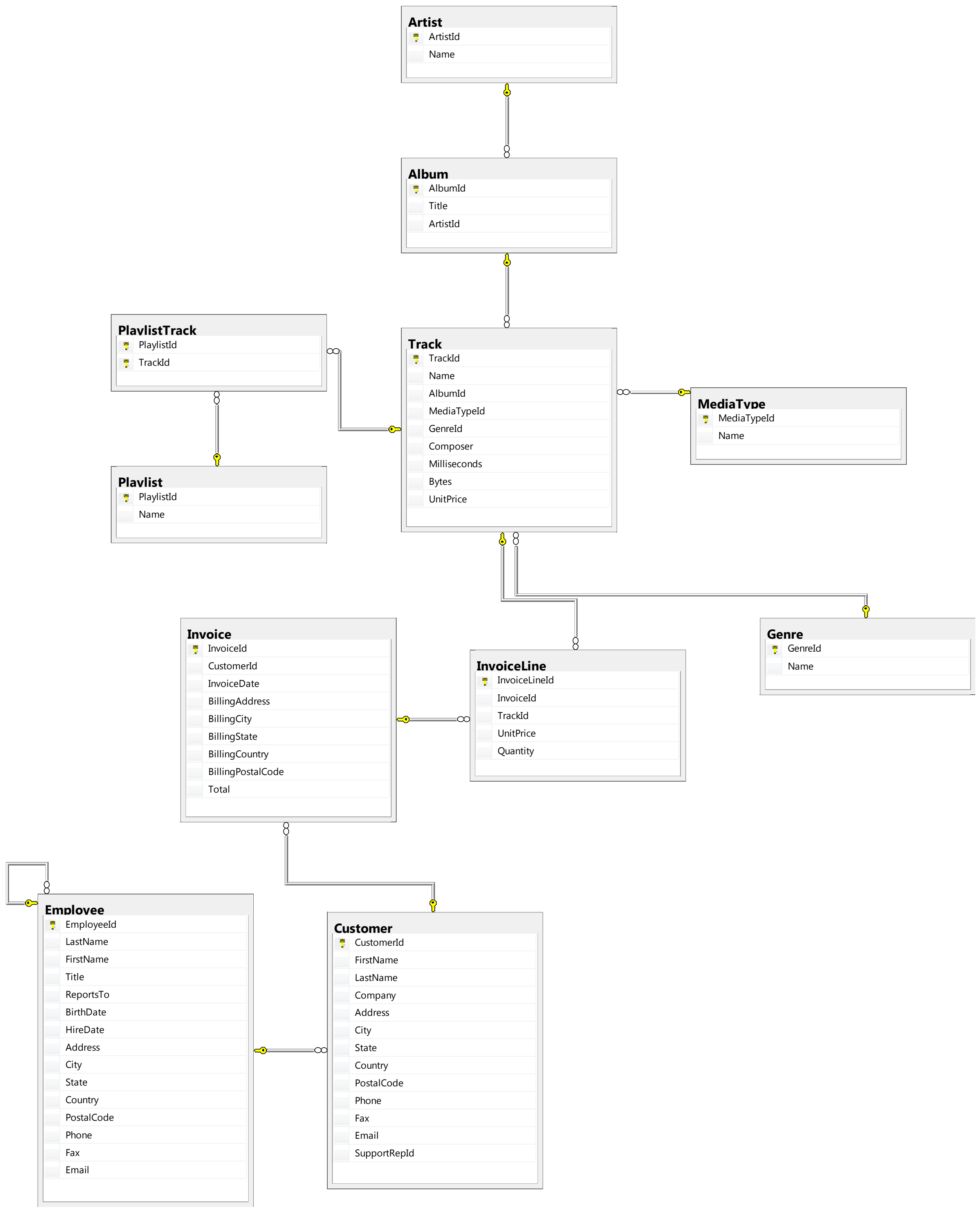


Figure 1. The Chinook database has data on artists, albums, tracks and playlists, as well as about customers and sales.

Introducing CTEs

A number of the solutions in this paper use CTEs, *Common Table Expressions*. CTEs have been available since SQL Server 2005. MySQL introduced them in the most recent version, 8.0.

A CTE is a query executed before the main query, in order to collect some data to be used in the main query. It's very similar to a derived table (that is, a query in the FROM clause), but easier to read and more useful. A CTE is easier to read because it's isolated from the main query rather than embedded in it. It's more useful because you can refer to the same CTE multiple times in the main query.

Listing 1 shows the syntax of a query with a CTE. The key elements are the WITH clause that names the CTE, the AS clause that contains the CTE query, and the main query that presumably uses the CTE.

Listing 1. A CTE is analogous to a derived table, but more useful.

```
WITH CTEName [(list of field names)]
AS
(SELECT <rest of query>)

SELECT <field list>
  FROM <tables, presumably including CTEName, and join conditions>
  <rest of query>
```

Listing 2 shows a fairly simple use of a CTE; it's included in the MySQL and SQL Server folders of the materials for this session as SalesByTrackCTE.SQL. The CTE groups data and the main query joins the grouped data to an underlying look-up table to provide descriptions.

Listing 2. Here the CTE computes annual sales totals for each track, and the main query adds the track name.

```
WITH csrSalesByTrack (TrackID, nYear, TotalSales)
AS
(SELECT TrackID, YEAR(InvoiceDate), SUM(UnitPrice * Quantity)
  FROM Invoice
    JOIN InvoiceLine
      ON Invoice.InvoiceId = InvoiceLine.InvoiceId
  GROUP BY TrackID, YEAR(InvoiceDate))

SELECT SBT.TrackID, Name, nYear, TotalSales
  FROM csrSalesByTrack SBT
    JOIN Track
      ON SBT.TrackID = Track.TrackId
  ORDER BY nYear, Name;
```

You can have multiple CTEs in a single query. Separate them with a comma following the terminating parenthesis for the preceding CTE definition. Any CTE can list any preceding CTE in its own FROM clause.

Both SQL Server and MySQL 8 support recursive CTEs; MySQL requires the keyword *RECURSIVE* following WITH. A recursive CTE joins two queries with UNION. The first query is an anchor; it provides one or more records to start with. The second query in the UNION references the CTE itself to build the complete result. Recursive CTEs are used in several of the solutions here; see the section "Filling in missing values," later in this document for the first such example and a little more explanation.

Both SQL Server and MySQL limit the number of levels of recursion. Each provides a way to override the default limit. In SQL Server, you set the limit by adding OPTION (MAXRECURSION n) to the query, where n is the maximum number of levels you want to allow. Set n to 0 for unlimited recursion.

In MySQL, you can't specify this at the query level. It's a global or session setting. The default is 1000; to change it for the current session, set the variable @@cte_max_recursion_depth, as in **Listing 3**.

Listing 3. To increase the depth allowed in recursive CTEs for the current session, set @@cte_max_recursion_depth.

```
SET @@cte_max_recursion_depth = 10000;
```

Introducing OVER

A number of the SQL Server and MySQL solutions in this paper use what are officially called *window functions*, but are widely known as the *OVER clause*. Window functions were introduced in SQL Server 2005, and enhanced in SQL Server 2012. They were added in MySQL 8.

The basic idea with window functions is that you can define a set of records and apply a function to only that set of records in order to specify a field in a query. The OVER clause works with about two dozen functions, including the familiar aggregate functions.

There are several ways to specify the set of records and those ways can be combined. The two ways you're most likely to use are with ORDER BY and PARTITION BY clauses. The basic structure for most of the window functions is shown in **Listing 4**.

Listing 4. Most of the window functions use this syntax.

```
<window function> OVER (
  [PARTITION BY <list of expressions>]
  [ORDER BY <list of <expression> ASC | DESC>>]
  [ROWS | RANGE <window frame>])
```

The PARTITION BY clause lets you divide the data into groups, much like GROUP BY. However, GROUP BY consolidates all the records with matching values into a single result record. PARTITION BY simply indicates the groups of records to which the specified function should be applied. The original records still appear in the result set.

The ORDER BY clause indicates the order in which records are processed by the specified function.

See the next section, "Numbering and ranking records," for a fairly simple use of window functions.

Window frame specification using RANGE and ROWS lets you apply a function to a subset of a partition; it was added to SQL Server 2012 and is available in MySQL 8. RANGE lets you limit the calculation to a group of rows based on their values for the ordering expression, while ROWS lets you limit the calculation to a set number of rows around the current row.

None of the solutions in this paper use the ROWS specification, but RANGE appears in one problem. RANGE lets you specify how far backward and/or forward in the current group to apply the function, based on the order of the records. It accepts only a few keywords and can combine any two with BETWEEN/AND. The list is:

- UNBOUNDED PRECEDING—go back to the first record of this group;
- CURRENT ROW—the record we're not processing;
- UNBOUNDED FOLLOWING—go all the way to the last record of this group.

So you can write something like RANGE BETWEEN CURRENT ROW AND UNBOUNDED FOLLOWING, to apply a function to every record from the current record to the end of the partition.

See "Matching values when aggregating," later in this paper for an example that requires a RANGE expression.

I've written at length about the OVER clause; my paper that covers the SQL SERVER 2014 version in depth is available at http://tomorrowssolutionsllc.com/ConferenceSessions/Going%20OVER%20and%20above%20with%20SQL.pdf.

Numbering and ranking records

Although SQL is set-oriented, it's not at all unusual to want to number the records in a result, in much the same way the VFP RECNO() function does. Most commonly, you have a sorted result set and you want to include the ranking of the item in the record itself.

Numbering and ranking across the whole result

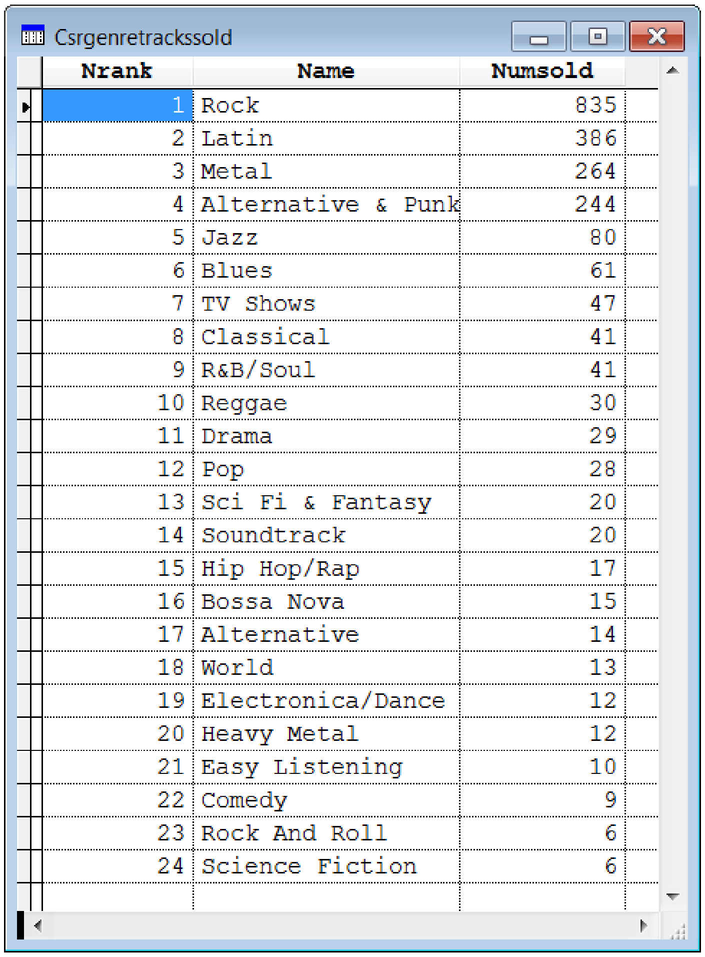
The simplest case is ranking all records in the result in a single list. That is, get the result, put it in the right order and then number from 1 to however many records there are.

In VFP, you can use RECNO() for this purpose, but there's a twist. When a query executes, VFP opens each table, whether or not it's already open. If the table is not open, it uses its normal alias and VFP leaves it open after executing the query. However, if the table is already open, VFP opens it again with a different alias and closes it afterwards. So, you can't be sure what alias the query uses to address a particular table.

This means that you can't pass the optional alias parameter to RECNO(), which in turn means you can't use RECNO() in a multi-table query. (In fact, in a multi-table query, it's risky to pass the optional alias parameter to any VFP function that accepts it.) The solution is to use a derived table (a query in the FROM clause) to get the results you want and put them in order, and use RECNO() against the derived table. **Listing 5** (RankGenreSold.PRG in the materials for this session) demonstrates, with a query that ranks the genres by the number of tracks sold. **Figure 2** shows the results.

Listing 5. In this query, the derived table calculates the number of tracks sold in each genre. Then, the main query assigns the ranks.

```
SELECT RECNO() AS nRank, * ;
   FROM (SELECT Genre.Name, ;
         SUM(Quantity) AS NumSold ;
      FROM Genre ;
        JOIN Track ;
          ON Genre.GenreID = Track.GenreID ;
        JOIN InvoiceLine ;
          ON Track.TrackID = InvoiceLine.TrackID ;
      GROUP BY Genre.Name ;
      ORDER BY NumSold DESC, 1) GTS;
   ORDER BY nRank ;
   INTO CURSOR csrGenreTracksSold
```



Nrank	Name	Numsold
1	Rock	835
2	Latin	386
3	Metal	264
4	Alternative & Punk	244
5	Jazz	80
6	Blues	61
7	TV Shows	47
8	Classical	41
9	R&B/Soul	41
10	Reggae	30
11	Drama	29
12	Pop	28
13	Sci Fi & Fantasy	20
14	Soundtrack	20
15	Hip Hop/Rap	17
16	Bossa Nova	15
17	Alternative	14
18	World	13
19	Electronica/Dance	12
20	Heavy Metal	12
21	Easy Listening	10
22	Comedy	9
23	Rock And Roll	6
24	Science Fiction	6

Figure 2. You can rank results in VFP by using a derived table to do the calculations.

In SQL Server and MySQL 8, the OVER clause provides a more direct way to get the result. To solve this problem, we use the RANK() function, which as its name suggests, ranks results, and the ORDER BY clause. **Listing 6** (RankGenreSold.sql in the SQLServer and MySQL folders of the materials for this session) shows the query. Like the VFP version, it uses SUM() to calculate the tracks sold for each genre, but rather than putting that calculation in a derived table, it's in the main query. The first field in the query uses OVER to compute the rank. You can read the expression as saying "put the records in descending order by the computed quantity, and assign them ranks, starting with 1."

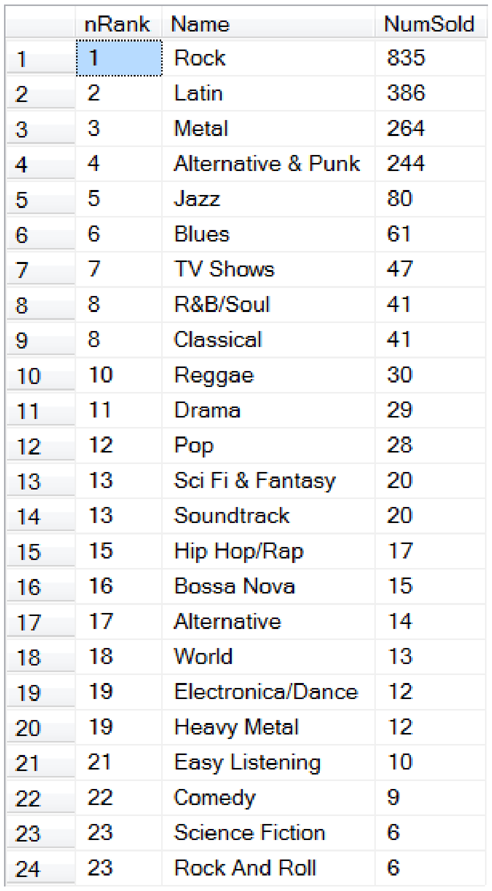
Listing 6. In SQL Server and MySQL 8, you can use the RANK() function with the OVER clause to number result records.

```
SELECT RANK() OVER (ORDER BY SUM(Quantity) DESC) AS nRank,
       Genre.Name,
       SUM(Quantity) AS NumSold
      FROM Genre
        JOIN Track
          ON Genre.GenreID = Track.GenreID
        JOIN InvoiceLine
          ON Track.TrackID = InvoiceLine.TrackID
```

```
	GROUP BY Genre.Name
	ORDER BY nRank
```

There is one significant difference between the SQL Server/MySQL results and the VFP results. In **Figure 3**, note that rows 8 and 9 both have nRank = 8. The RANK() function assigns ties the same rank. What determines a tie? The value of the expression in the ORDER BY clause. Here, it's SUM(Quantity). Since 41 tracks have been sold in both the R&B/Soul and Classical genres, they get the same rank. In keeping with normal practice, the next record has rank 10.

SQL Server and MySQL can handle this differently. The ROW_NUMBER() function assigns numbers sequentially without regard to ties, and the DENSE_RANK() function handles ties the way RANK() does, but continues with the next number rather than skipping ahead.



	nRank	Name	NumSold
1	1	Rock	835
2	2	Latin	386
3	3	Metal	264
4	4	Alternative & Punk	244
5	5	Jazz	80
6	6	Blues	61
7	7	TV Shows	47
8	8	R&B/Soul	41
9	8	Classical	41
10	10	Reggae	30
11	11	Drama	29
12	12	Pop	28
13	13	Sci Fi & Fantasy	20
14	13	Soundtrack	20
15	15	Hip Hop/Rap	17
16	16	Bossa Nova	15
17	17	Alternative	14
18	18	World	13
19	19	Electronica/Dance	12
20	19	Heavy Metal	12
21	21	Easy Listening	10
22	22	Comedy	9
23	23	Science Fiction	6
24	23	Rock And Roll	6

Figure 3. The RANK() function used with the OVER clause assigns the same value to ties, so for example, both R&B/Soul and Classical have rank 8 here.

While this query works in MySQL 8, that's the first version of MySQL where you can do this. For earlier versions of MySQL, you can use a trick based on the ability to assign values to variables in a query. You create a variable set to 0 in the FROM clause, and then increment it in the field list. To do so, the data needs to be in the desired order, so the query to compute the sales becomes a derived table. **Listing 7** (RankGenreSoldPre8.sql in the MySQL folder of the materials for this session) shows the complete query. The first query in the FROM clause computes the number of tracks sold for each genre and sorts the genres into descending order based on that calculation. The second query in the FROM clause creates the variable @nRank and initializes it to 0. Since that query produces only a single result, the implied join between the two tables has one record per genre. In the main query, the assignment @nRank := @nRank + 1 computes a new value for nRank for each record in the result.

Listing 7. In MySQL versions earlier than 8, you can use rank records by incrementing a variable created in the query.

```
SELECT @nRank := @nRank + 1 AS nRank,
       SalesDesc.*
   FROM (SELECT Genre.Name,
                SUM(Quantity) AS NumSold
         FROM Genre
           JOIN Track
             ON Genre.GenreID = Track.GenreID
           JOIN InvoiceLine
             ON Track.TrackID = InvoiceLine.TrackID
         GROUP BY Genre.Name
         ORDER BY NumSold Desc) SalesDesc,
      (SELECT @nRank := 0) Rank
   ORDER BY nRank;
```

As with the VFP query, genres with the same number of sales are assigned different ranks. If you care about the order in which tied genres are ranked, add to the ORDER BY clause of the subquery. For example, if you want tied genres to be assigned ranks in alphabetical order, add Name to the ORDER BY clause of the first subquery. (The VFP solution already does this.)

Numbering and ranking within groups

The problem gets harder when you want to number groups in the result separately. Here, we'll look at the length of tracks in each genre, ranking them from longest to shortest.

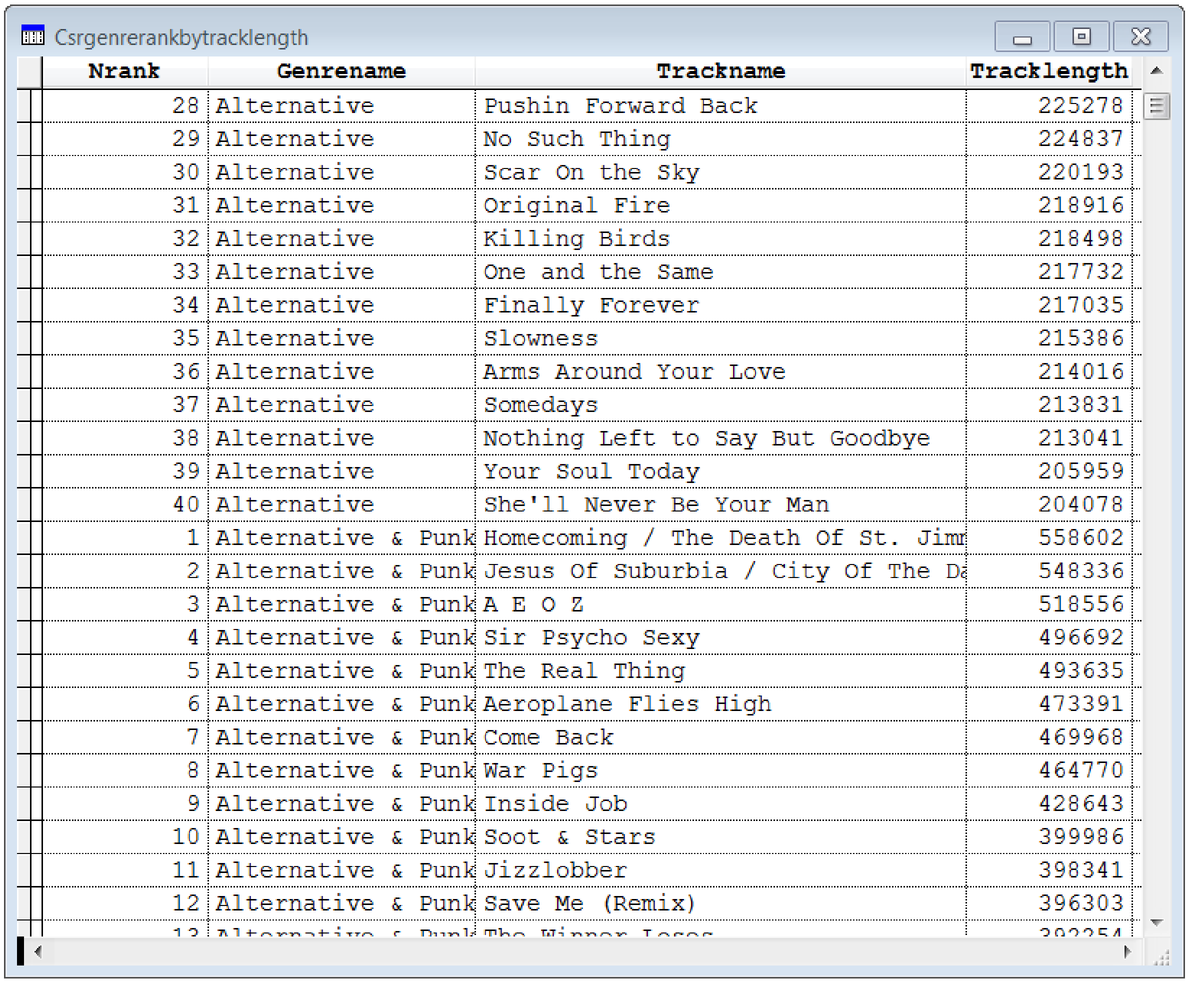
In VFP, you can't do this in a single query. The easiest way to do it is with a mix of SQL and Xbase code. First, collect the track lengths by genre. Then, loop through the genres and use the previous technique to rank the genres separately.

Listing 8 (RankByTrackLength.prg in the VFP folder of the materials for this session) demonstrates. First, we create a cursor to hold the final result. Then, we run a query that extracts track length and genre information. A follow-up query puts a list of genres into a cursor. Then we loop through that cursor of genres and, for each, use the technique from the previous example to attach a rank to each record for that genre. The query is included in an INSERT command that adds it to the overall result. (We could, of course, run the query that adds the ranks and put that result into another cursor and then append that cursor to the final result cursor. But INSERT INTO … SELECT lets us do it in one step rather than two.) **Figure 4** shows partial results.

Listing 8. In VFP, you can't rank records within groups in a single query. Instead, you need a query to collect the raw data and then a loop to handle each group.

```
CREATE CURSOR csrGenreRankByTrackLength ;
    (nRank I, GenreName C(120), TrackName C(200), TrackLength I)

SELECT Track.Name AS TrackName, ;
       Milliseconds AS TrackLength, ;
       Genre.Name AS GenreName ;
    FROM Track ;
      JOIN Genre ;
        ON Track.GenreId = Genre.GenreId ;
    ORDER BY GenreName, TrackLength DESC ;
    INTO CURSOR csrGenreLengthOrder

SELECT DISTINCT GenreName ;
    FROM csrGenreLengthOrder ;
    ORDER BY GenreName ;
    INTO CURSOR csrGenres


LOCAL cGenreName

SCAN
    cGenreName = csrGenres.GenreName
    INSERT INTO csrGenreRankByTrackLength ;
        SELECT RECNO() AS nRank, * ;
            FROM (SELECT GenreName, TrackName, TrackLength ;
                    FROM csrGenreLengthOrder ;
                    WHERE GenreName == m.cGenreName ;
                    ORDER BY TrackLength DESC) csrOneGenre
ENDSCAN
```

In this example, you could actually omit the first query and extract the data directly from the original tables in the innermost query inside the loop. When you need to calculate or aggregate data along the way, running a separate query first is likely to be more readable.



Csrgenrerankbytracklength

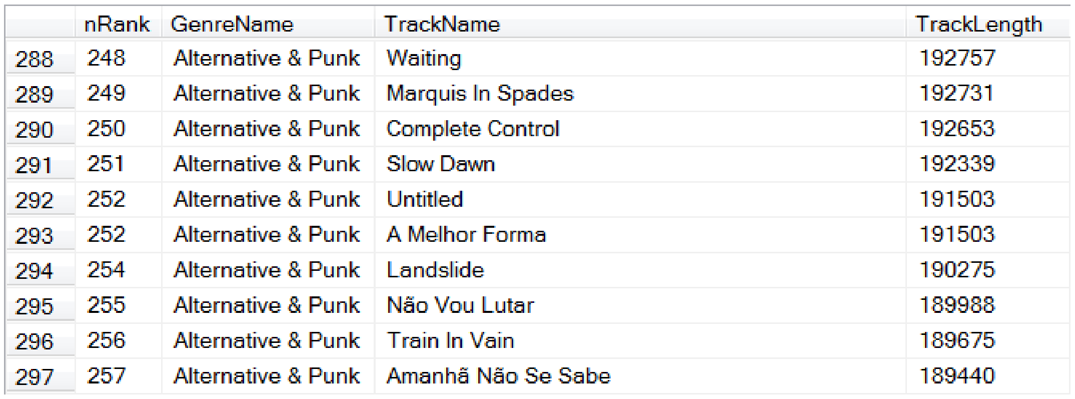
Nrank	Genrename	Trackname	Tracklength
28	Alternative	Pushin Forward Back	225278
29	Alternative	No Such Thing	224837
30	Alternative	Scar On the Sky	220193
31	Alternative	Original Fire	218916
32	Alternative	Killing Birds	218498
33	Alternative	One and the Same	217732
34	Alternative	Finally Forever	217035
35	Alternative	Slowness	215386
36	Alternative	Arms Around Your Love	214016
37	Alternative	Somedays	213831
38	Alternative	Nothing Left to Say But Goodbye	213041
39	Alternative	Your Soul Today	205959
40	Alternative	She'll Never Be Your Man	204078
1	Alternative & Punk	Homecoming / The Death Of St. Jimr	558602
2	Alternative & Punk	Jesus Of Suburbia / City Of The Da	548336
3	Alternative & Punk	A E O Z	518556
4	Alternative & Punk	Sir Psycho Sexy	496692
5	Alternative & Punk	The Real Thing	493635
6	Alternative & Punk	Aeroplane Flies High	473391
7	Alternative & Punk	Come Back	469968
8	Alternative & Punk	War Pigs	464770
9	Alternative & Punk	Inside Job	428643
10	Alternative & Punk	Soot & Stars	399986
11	Alternative & Punk	Jizzlobber	398341
12	Alternative & Punk	Save Me (Remix)	396303
13	Alternative & Punk	The Winner Loses	392254

Figure 4. With a little creativity, you can rank records within groups in VFP.

The problem is simpler in SQL Server and MySQL 8 because OVER's PARTITION BY clause lets you do it in a single step. In **Listing 9** (RankByTrackLength.sql in the SQL Server and MySQL folders of the materials for this session, respectively), the expression for nRank says to divide the records into groups based on GenreID and apply the RANK() function separately within each group, in descending order based on Milliseconds. **Figure 5** shows partial results; as in the previous example, ties are acknowledged by assigning the same rank.

Listing 9. OVER's PARTITION BY clause lets you apply functions like RANK() to groups within a query's results, not just to the results as a whole.

```
SELECT RANK() OVER (PARTITION BY Track.GenreID ORDER BY Milliseconds DESC) AS nRank,
       Track.Name AS TrackName,
       Milliseconds AS TrackLength,
       Genre.Name AS GenreName
    FROM Track
      JOIN Genre
        ON Track.GenreId = Genre.GenreId
    ORDER BY GenreName, nRank;
```



	nRank	GenreName	TrackName	TrackLength
288	248	Alternative & Punk	Waiting	192757
289	249	Alternative & Punk	Marquis In Spades	192731
290	250	Alternative & Punk	Complete Control	192653
291	251	Alternative & Punk	Slow Dawn	192339
292	252	Alternative & Punk	Untitled	191503
293	252	Alternative & Punk	A Melhor Forma	191503
294	254	Alternative & Punk	Landslide	190275
295	255	Alternative & Punk	Não Vou Lutar	189988
296	256	Alternative & Punk	Train In Vain	189675
297	257	Alternative & Punk	Amanhã Não Se Sabe	189440

Figure 5. Ranking within groups is easy in SQL Server and MySQL 8 because of the PARTITION BY clause of OVER.

In earlier versions of MySQL, you can again use variables to accomplish the task. In this case, you need both the @nRank variable of the previous example and another to track the genre. **Listing 10** demonstrates; it's included as RankByTrackLengthPre8.sql in the MySQL folder of the materials for this session. The first subquery collects and orders the genre and track length information. The second subquery initializes the variables @nRank and @GenreName. In the main query, the assignment to @nRank is a CASE statement. The first case handles records from the same genre as the previous record. The second case handles changes in genre, saving the new genre to the @GenreName variable as well as setting nRank to 1. The whole thing depends on @GenreName getting set after being compared to the GenreName field in the current record. (There's a well-known technique using variables in the VFP Report Designer that's quite similar, in fact.)

Listing 10. Variable assignment in the query can be extended to allow ranking records within groups in versions of MySQL before 8.

```
SELECT @nRank := CASE WHEN @GenreName = GenreName THEN @nRank + 1
                      WHEN (@GenreName := GenreName) IS NOT NULL THEN 1 END AS nRank,
       GenreTrackLength.*
    FROM (SELECT Genre.Name AS GenreName,
                 Track.Name AS TrackName,
                 Milliseconds AS TrackLength
              FROM Track
                JOIN Genre
                  ON Track.GenreId = Genre.GenreId
              ORDER BY GenreName, TrackLength DESC) GenreTrackLength,
         (SELECT @nRank := 0,
                 @GenreName := '') Vars
    ORDER BY GenreName, nRank;
```

Top N in each group

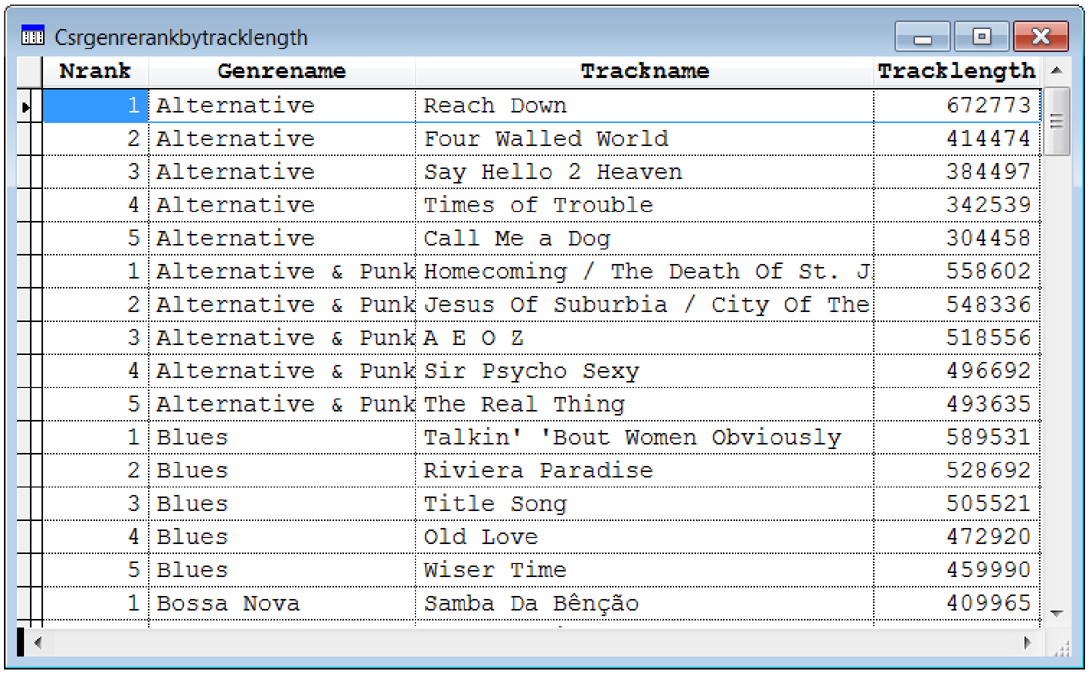
Once you know how to rank records in groups, you're almost all the way to finding the top N records in a group. Finding the top N in a group is a common problem, used for everything from identifying the top students in each class to determining the bestselling products for each department in a store to finding the least productive employees in each department of a company. To demonstrate, we'll find the five longest tracks in each genre in the Chinook data.

You might think the TOP n clause, which allows you to include in the result only the first n records that match a query's filter conditions, would let you solve these problems. But TOP n by itself doesn't work when what you really want is the TOP n for each group in the query.

In VFP, like ranking within a group, finding the top N in each group requires a hybrid solution using both SQL and xBase. Again, we collect the data and get a list of genres, then loop through them. This time, we use TOP n to keep only the top five records within each group, as shown in **Listing 11** (TopNTrackLengthByGenre.prg in the VFP folder of the materials for this session). Results are shown in **Figure 6**.

Listing 11. To find the top N for each group in a dataset, you use code very similar to that used for ranking within groups, but apply TOP n to keep only the records you want.

```
CREATE CURSOR csrGenreRankByTrackLength ;
    (nRank I, GenreName C(120), TrackName C(200), TrackLength I)

SELECT Track.Name AS TrackName, ;
       Milliseconds AS TrackLength, ;
       Genre.Name AS GenreName ;
    FROM Track ;
      JOIN Genre ;
        ON Track.GenreId = Genre.GenreId ;
    ORDER BY GenreName, TrackLength DESC ;
    INTO CURSOR csrGenreLengthOrder

SELECT DISTINCT GenreName ;
    FROM csrGenreLengthOrder ;
    ORDER BY GenreName ;
    INTO CURSOR csrGenres


LOCAL cGenreName

SCAN
    cGenreName = csrGenres.GenreName
    INSERT INTO csrGenreRankByTrackLength ;
        SELECT RECNO() AS nRank, * ;
            FROM (SELECT TOP 5 GenreName, TrackName, TrackLength ;
                    FROM csrGenreLengthOrder ;
                    WHERE GenreName == m.cGenreName ;
                    ORDER BY TrackLength DESC) csrOneGenre
ENDSCAN
```



Nrank	Genrename	Trackname	Tracklength
1	Alternative	Reach Down	672773
2	Alternative	Four Walled World	414474
3	Alternative	Say Hello 2 Heaven	384497
4	Alternative	Times of Trouble	342539
5	Alternative	Call Me a Dog	304458
1	Alternative & Punk	Homecoming / The Death Of St. J	558602
2	Alternative & Punk	Jesus Of Suburbia / City Of The	548336
3	Alternative & Punk	A E O Z	518556
4	Alternative & Punk	Sir Psycho Sexy	496692
5	Alternative & Punk	The Real Thing	493635
1	Blues	Talkin' 'Bout Women Obviously	589531
2	Blues	Riviera Paradise	528692
3	Blues	Title Song	505521
4	Blues	Old Love	472920
5	Blues	Wiser Time	459990
1	Bossa Nova	Samba Da Bênção	409965

Figure 6. A combination of SQL and Xbase lets you find the top N for each group in a result set.

Once again, the problem is easier in SQL Server and MySQL 8. We can simply use the previous code as a CTE and filter on the newly-computed rank field. **Listing 12** (TopNTrackLengthByGenre.sql in the appropriate folder of the materials for this session) shows the same code as before (except for the ORDER BY clause), but it's now in a CTE. The main query has a WHERE clause to filter on nRank and the ORDER BY clause has been promoted by one level. The results are identical to those obtained in VFP, except for the handling of any ties.

Listing 12. In SQL Server and MySQL 8, we can simply use the code to rank within groups as a CTE and filter based on the computed rank of each record.

```
WITH RankByTrackLength (nRank, TrackName, TrackLength, GenreName)
AS
(SELECT RANK() OVER (PARTITION BY Track.GenreID ORDER BY Milliseconds DESC) AS nRank,
        Track.Name AS TrackName,
        Milliseconds AS TrackLength,
        Genre.Name AS GenreName
    FROM Track
      JOIN Genre
        ON Track.GenreId = Genre.GenreId)

SELECT *
    FROM RankByTrackLength
    WHERE nRank <= 5
    ORDER BY GenreName, nRank;
```

The same approach provides a solution for MySQL 5.7 and earlier. Just turn the previous solution into a derived table and filter on the new nRank field, and move the ORDER BY clause into the new main query. **Listing 13** (TopNGenreByYearPre8.sql in the MySql folder of the materials for this session) shows the complete query.

Listing 13. The hard part of getting the top N for each group in MySQL 5.7 and earlier is ranking the records within the group, as shown in the preceding section. Once that's done, a simple filter keeps only the top N records in each group.

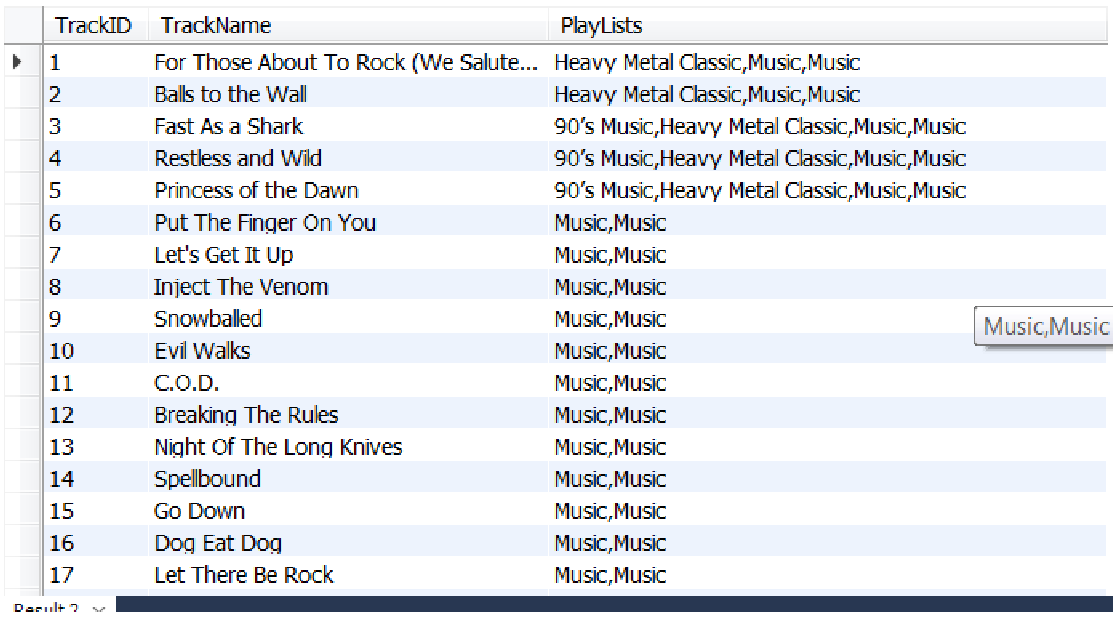
```
SELECT *
  FROM (SELECT @nRank := CASE WHEN @GenreName = GenreName THEN @nRank + 1
                             WHEN (@GenreName := GenreName) IS NOT NULL THEN 1 END
                  AS nRank,
               GenreTrackLength.*
          FROM (SELECT Genre.Name AS GenreName,
                       Track.Name AS TrackName,
                       Milliseconds AS TrackLength
                    FROM Track
                      JOIN Genre
                        ON Track.GenreId = Genre.GenreId
                    ORDER BY GenreName, TrackLength DESC) GenreTrackLength,
               (SELECT @nRank := 0,
                       @GenreName := '') Vars) RankedLengths
    WHERE nRank <= 5
    ORDER BY GenreName, nRank;
```

Consolidate data from a field into a list

One of the most common questions I see in online VFP forums is how to group data, consolidating the data from a particular field. If the consolidation you want is counting, summing, or averaging, the task is simple; just use GROUP BY with the corresponding aggregate function.

But if you want to create a comma-separated list of all the values or something like that, the solution isn't so simple. In VFP, you need a mix of SQL and Xbase. Both SQL Server and MySQL have language elements to handle the request directly.

The problem we'll solve here is producing a comma-separated list of playlists for each track. **Figure 7** shows part of the results we're after (in MySQL). (For some reason, the Playlist table includes several repeated playlist names, so the results for many of the tracks include some strings twice, most noticeably "Music.")



TrackID	TrackName	PlayLists
1	For Those About To Rock (We Salute...	Heavy Metal Classic,Music,Music
2	Balls to the Wall	Heavy Metal Classic,Music,Music
3	Fast As a Shark	90's Music,Heavy Metal Classic,Music,Music
4	Restless and Wild	90's Music,Heavy Metal Classic,Music,Music
5	Princess of the Dawn	90's Music,Heavy Metal Classic,Music,Music
6	Put The Finger On You	Music,Music
7	Let's Get It Up	Music,Music
8	Inject The Venom	Music,Music
9	Snowballed	Music,Music
10	Evil Walks	Music,Music
11	C.O.D.	Music,Music
12	Breaking The Rules	Music,Music
13	Night Of The Long Knives	Music,Music
14	Spellbound	Music,Music
15	Go Down	Music,Music
16	Dog Eat Dog	Music,Music
17	Let There Be Rock	Music,Music

Figure 7. Each language offers a different approach to creating a comma-separated list of data from different records.

VFP

In VFP, you have to collect the data you want and then loop through it to create the list of playlists for each track. **Listing 14** (TrackPlaylists.prg in the VFP folder of the materials for this session) shows the code. The initial query creates one record for each track/playlist combination. Then, we create a cursor to hold the final result. The loop goes through the initial cursor, building up the list for the current track. When we get to the next track, we save the track we were working on and set a few variables to indicate the current track. After the loop, we need to save the data for the last track.

Listing 14. In VFP, creating this kind of list requires a loop.

```
* Get the list with one record per combination
SELECT Track.TrackID, Track.Name AS TrackName, Playlist.Name AS PlayListName ;
   FROM Track ;
     JOIN PlaylistTrack ;
       ON Track.TrackID = PlaylistTrack.TrackID ;
     JOIN Playlist ;
       ON PlaylistTrack.PlaylistID = Playlist.PlaylistID ;
   ORDER BY 1, 3 ;
   INTO CURSOR csrTrackAndPlaylists

LOCAL cPlayLists, iTrackID, cTrackName

* Create a cursor to hold results
CREATE CURSOR csrTrackPlayLists (TrackID I, TrackName VARCHAR(200), PlayLists M)

SELECT csrTrackAndPlayLists
iTrackID = csrTrackAndPlaylists.TrackID
cTrackName = csrTrackAndPlaylists.TrackName
cPlayLists = ''
```

```
* Loop through to gather data
SCAN
   IF csrTrackAndPlaylists.TrackID <> m.iTrackID
      * Finished current track
      INSERT INTO csrTrackPlayLists ;
         VALUES (m.iTrackID, m.cTrackName, m.cPlayLists)
      iTrackId = csrTrackAndPlaylists.TrackID
      cTrackName = csrTrackAndPlaylists.TrackName
      cPlayLists = ''
   ENDIF

   cPlayLists = IIF(EMPTY(cPlayLists), '',  m.cPlayLists + ', ') + ;
     ALLTRIM(csrTrackAndPlaylists.PlaylistName)

ENDSCAN

* Save last record
INSERT INTO csrTrackPlayLists ;
   VALUES (m.iTrackID, m.cTrackName, m.cPlayLists)

SELECT csrTrackPlayLists
```

MySQL

MySQL has an easy way to get this result. The GROUP_CONCAT() function works with GROUP BY and lets you create a comma-separated list from the data in each record in the group.

GROUP_CONCAT() is versatile. It includes a number of optional clauses, including DISTINCT (to let you cut the list of values down to unique values), ORDER BY (to let you specify the order in which the values are concatenated), and SEPARATOR (to let you specify a separator other than comma).

Listing 15 (TrackPlaylists.sql in the MySQL folder of the materials for this session) shows the MySQL solution. We apply GROUP_CONCAT to the playlist name, using ORDER BY to make the list alphabetical.

Listing 15. Consolidating a list of values is easy in MySQL because of the GROUP_CONCAT() function.

```
SELECT Track.TrackID, Track.Name AS TrackName,
       GROUP_CONCAT(Playlist.Name ORDER BY 1) AS PlayLists
FROM Track
  JOIN PlaylistTrack
    ON Track.TrackId = PlaylistTrack.TrackId
  JOIN PlayList
    ON PlaylistTrack.PlaylistId = Playlist.PlaylistId
GROUP BY TrackID, TrackName
ORDER BY 1;
```

SQL Server

Prior to SQL Server 2017, this problem was complex, but that version added the STRING_AGG() function that's analogous to MySQL's GROUP_CONCAT(). The details are a little different, but the capabilities are quite similar. You pass the expression to consolidate and the separator to use. Optionally, you can specify the order for the data using the WITHIN GROUP clause. If the field contains anything other than the field using STRING_AGG(), the query requires a GROUP BY clause.

Listing 16 (TrackPlaylists.SQL in the SQL Server folder of the materials for this session) shows the SQL Server 2017 solution for this problem. The parameters to STRING_AGG() say to combine the Playlist.Name field with comma separators, and the WITHIN GROUP clause sorts the data on the Playlist.Name field before combining.

Listing 16. The new STRING_AGG() function in SQL Server 2017 makes consolidating data from multiple records easy.

```
SELECT Track.TrackID, Track.Name AS TrackName,
       STRING_AGG(Playlist.Name, ',') WITHIN GROUP (ORDER BY Playlist.Name)
         AS PlayLists
FROM Track
  JOIN PlaylistTrack
    ON Track.TrackId = PlaylistTrack.TrackId
  JOIN PlayList
    ON PlaylistTrack.PlaylistId = Playlist.PlaylistId
GROUP BY Track.TrackID, Track.Name
ORDER BY 1;
```

In earlier versions of SQL Server, there are two ways to solve this problem: using the FOR XML clause and using a stored function. This paper demonstrates only the FOR XML solution. My paper http://tomorrowssolutionsllc.com/ConferenceSessions/Go%20Beyond%20VFPs%20SQL%20with%20SQL%20Server.pdf works through the stored function solution, as well.

In general, FOR XML allows you to convert SQL results to XML. There are four variations; three of them produce XML results and vary only in how much control you have over the format of the result.

The fourth version of FOR XML, using the PATH keyword, provides what we need to consolidate the data into a single record. FOR XML PATH treats columns as XPath expressions. XPath, which stands for XML Path language, lets you select items in an XML document. (The full details are beyond the scope of this paper.)

What you need to know to solve the problem of creating a comma-separated list is that if you specify FOR XML PATH(''), the expression you specify in the query is consolidated into a single list, rather than one record per value.

Listing 17 (TracksAndPlaylistsPre2017.SQL in the SQLServer folder of the materials for this session) shows the complete solution to creating a comma-separated list of playlists for each track. After the listing, we'll work through the query from the inside out.

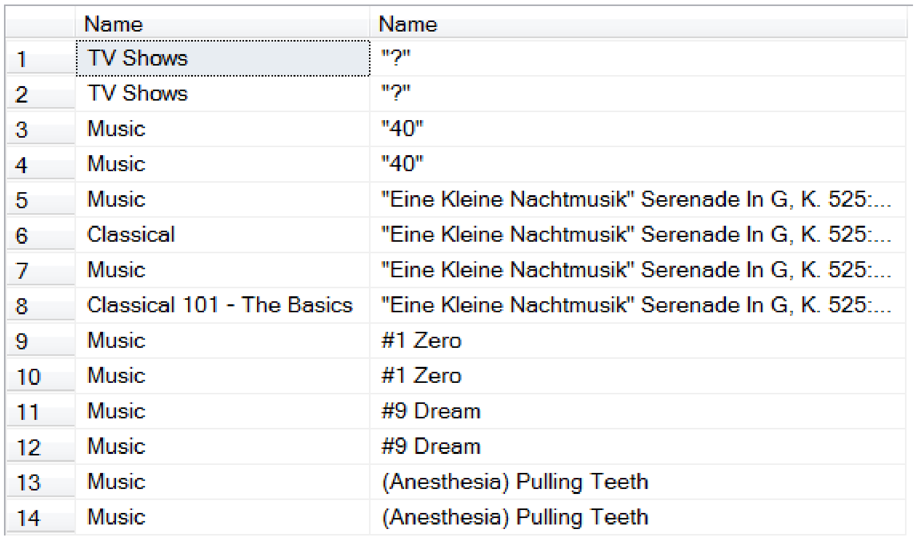
Listing 17. This complex query results in one record per track, with a comma-separated list of playlists that include that track.

```
SELECT Track.TrackID AS TrackID, Track.Name AS TrackName,
       STUFF((SELECT ', ' + Name
               FROM (SELECT PlayList.Name
                       FROM Track T1
                         JOIN PlaylistTrack
                           ON T1.TrackId = PlaylistTrack.TrackId
                         JOIN PlayList
                           ON PlaylistTrack.PlaylistId = Playlist.PlaylistId
                       WHERE T1.TrackID = Track.TrackID) AllPLs
               ORDER BY Name
               FOR XML PATH('')), 1, 2, '') AS TrackPlaylists
   FROM Track
   ORDER BY 1;
```

The innermost query (that begins with SELECT PlayList.Name) gets the list of playlists for a single track, with one record per playlist. If we extract that query and drop its WHERE clause, we can get a list of the playlists for each track. **Listing 18** (AllTracksAndPlaylists.SQL in the SQLServer folder of the materials for this session) shows that innermost query, without the WHERE clause, with the track name added to the field list and the result sorted by track name. **Figure 8** shows the results. In the final query, this subquery is correlated to the main query on the track id, so pulls data for one track at a time (at least in theory—I suspect the engine is smarter and does the whole thing at once for optimization purposes).

Listing 18. This query collects the paired list of playlists and tracks. It's an expanded version of the innermost query in the overall solution.

```
SELECT PlayList.Name, T1.Name
   FROM Track T1
     JOIN PlaylistTrack
       ON T1.TrackId = PlaylistTrack.TrackId
     JOIN PlayList
       ON PlaylistTrack.PlaylistId = Playlist.PlaylistId
    ORDER BY T1.Name;
```



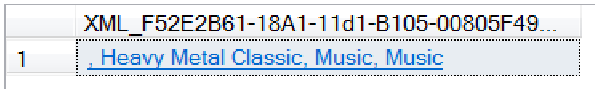
	Name	Name
1	TV Shows	"?"
2	TV Shows	"?"
3	Music	"40"
4	Music	"40"
5	Music	"Eine Kleine Nachtmusik" Serenade In G, K. 525:...
6	Classical	"Eine Kleine Nachtmusik" Serenade In G, K. 525:...
7	Music	"Eine Kleine Nachtmusik" Serenade In G, K. 525:...
8	Classical 101 - The Basics	"Eine Kleine Nachtmusik" Serenade In G, K. 525:...
9	Music	#1 Zero
10	Music	#1 Zero
11	Music	#9 Dream
12	Music	#9 Dream
13	Music	(Anesthesia) Pulling Teeth
14	Music	(Anesthesia) Pulling Teeth

Figure 8. To get a comma-separated list of playlists for each track, we first need to collect the right data.

The next query in **Listing 17**, moving outward, applies FOR XML PATH. Together with the field expression (', ' + Name), the query says to combine a comma, a space and the Name field from each record into a single string. **Listing 19** (TrackPlaylistOneRecord.SQL in the SQLServer folder of the materials for this session) contains a modified version of this query that selects data for a single track (the one with TrackID = 1) and applies this transformation. **Figure 9** shows the result, a single XML string.

Listing 19. When you use FOR XML PATH(''), the specified expression is consolidated into a single record.

```
SELECT ', ' + Name
   FROM (SELECT PlayList.Name, T1.Name AS TrackName
           FROM Track T1
             JOIN PlaylistTrack
               ON T1.TrackId = PlaylistTrack.TrackId
             JOIN PlayList
               ON PlaylistTrack.PlaylistId = Playlist.PlaylistId
           WHERE T1.TrackId = 1) AllPLs
   ORDER BY Name
   FOR XML PATH('');
```



	XML_F52E2B61-18A1-11d1-B105-00805F49...
1	, Heavy Metal Classic, Music, Music

Figure 9. The result of using FOR XML PATH is an XML string.

The result is very close to what we want, but we need to remove the leading comma and space from the string, and of course, in the final result, we want some other fields. The STUFF() function lets us replace the first two characters in the XML string with the empty string, giving us the comma-separated list of playlists. That whole expression, with the call to STUFF() containing the nested queries that produce the result we want, is in the field list of the final query (in **Listing 17**) with the track ID and name.

Finding duplicates

The repeated playlist names we noticed in the previous example raise the question of how to find duplicate records. The first issue is deciding what constitutes a duplicate record. This is a business question, not a programming question, and the answer depends on the situation. In a customer table, it might be matching addresses, while in a table of people, it might require an exact match of first and last names, plus social security number.

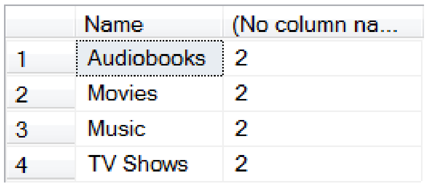
In Chinook, for the duplicated playlist names, we'd probably want to know whether the actual lists are the same; perhaps it's just unfortunate naming. There are also some track names that appear more than once. That's not surprising, as many song titles have been repeated over time. However, a deeper search shows that some track names appear more than once on the same album. Are those true duplicates? Maybe, but knowing that sometimes the same song appears twice on the same album (for example, as a reprise), we might further want to compare the lengths or the file sizes.

Once you know what constitutes a duplicate, identifying duplicate values can be easy. Even better, the code is the same in all three versions of SQL. **Listing 20** shows a query that finds the duplicated playlist names. (Each of the language-specific folders in the materials for this session includes a file named FindDups, with the appropriate extension.) **Figure 10** shows the results (in SQL Server, but they're the same in each database).

The idea is to group by the field or fields that define a record as a duplicate and keep only those that appear more than once. So, in this example, we group on the Name field. (Here, we're focusing only on the duplicated Name; you'd need additional code to see whether the list of tracks on the identically-named playlists is the same.)

Listing 20. Checking whether there are duplicates is fairly easy, but this query doesn't find the duplicated records for you.

```
SELECT Name, COUNT(*)
   FROM PlayList
   GROUP BY Name
   HAVING COUNT(*) > 1;
```



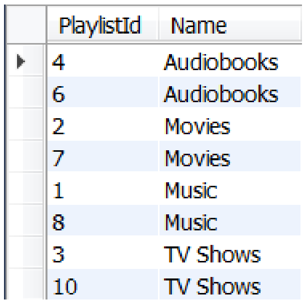
	Name	(No column na...
1	Audiobooks	2
2	Movies	2
3	Music	2
4	TV Shows	2

Figure 10. It's easy to get a list of duplicated values, along with the number of records containing that value (or set of values).

However, this approach doesn't tell you what specific records are duplicated, just which identifying values they contain. To find the actual records, you need to use the results of the previous query and grab all the records that match, as in **Listing 21** (FindDupRecords in each of the language-specific folders in the materials for this session). **Figure 11** shows the results (this time in MySQL).

Listing 21. To find the actual duplicate records, join the list of duplicates to the original data.

```
SELECT PlayList.*
   FROM PlayList
     JOIN (SELECT Name, COUNT(*) AS PLCount
             FROM PlayList
             GROUP BY Name
             HAVING COUNT(*) > 1) Dups
       ON PlayList.Name = Dups.Name
   ORDER BY Name;
```



PlaylistId	Name
4	Audiobooks
6	Audiobooks
2	Movies
7	Movies
1	Music
8	Music
3	TV Shows
10	TV Shows

Figure 11. You can use the list of duplicated values to find the records that contain them.

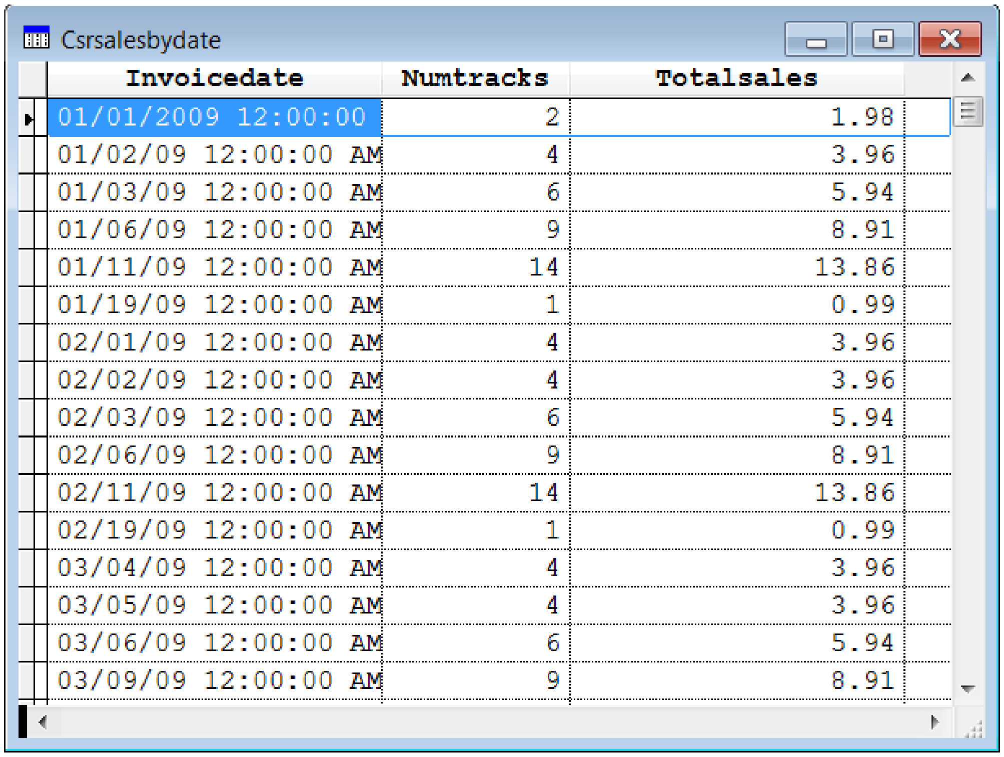
Once you've identified duplicate records, what to do with them is also a business problem, not a technical problem. Often, the solution is to show them to a user and let the user decide what to do.

Filling in missing values

When aggregating data, especially over periods of times (days, months, years, etc.), it's not unusual to run into cases where some periods are missing. For example, the code in **Listing 22** (SalesByDay in the language-specific folders of the materials for this session) computes the daily sales for Chinook, that is, for each date, the number of tracks sold and the revenue for those tracks. (The listing is the VFP version.) The partial results shown in **Figure 12** let you see that there are dates on which there were no sales. (While that seems unlikely for a production application, imagine aggregating sales by day for each artist. It's not at all unlikely that on some days, some artists didn't sell at all.)

Listing 22. This VFP query computes daily sales.

```
SELECT InvoiceDate, ;
       SUM(Quantity) AS NumTracks, ;
       SUM(Quantity * InvoiceLine.UnitPrice) AS TotalSales ;
   FROM Invoice ;
     JOIN InvoiceLine ;
       ON Invoice.InvoiceID = InvoiceLine.InvoiceID ;
     JOIN Track ;
       ON InvoiceLine.TrackID = Track.TrackID ;
   GROUP BY InvoiceDate ;
   ORDER BY 1 ;
   INTO CURSOR csrSalesByDate
```



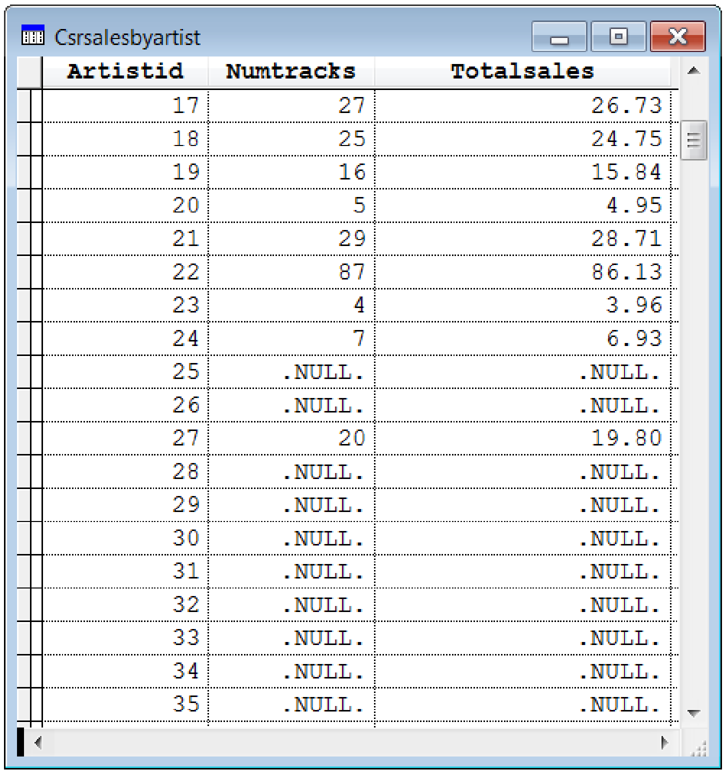
Invoicedate	Numtracks	Totalsales
01/01/2009 12:00:00	2	1.98
01/02/09 12:00:00 AM	4	3.96
01/03/09 12:00:00 AM	6	5.94
01/06/09 12:00:00 AM	9	8.91
01/11/09 12:00:00 AM	14	13.86
01/19/09 12:00:00 AM	1	0.99
02/01/09 12:00:00 AM	4	3.96
02/02/09 12:00:00 AM	4	3.96
02/03/09 12:00:00 AM	6	5.94
02/06/09 12:00:00 AM	9	8.91
02/11/09 12:00:00 AM	14	13.86
02/19/09 12:00:00 AM	1	0.99
03/04/09 12:00:00 AM	4	3.96
03/05/09 12:00:00 AM	4	3.96
03/06/09 12:00:00 AM	6	5.94
03/09/09 12:00:00 AM	9	8.91

Figure 12. When you aggregate data, some groups may be missing. Here, there's no data for January 4, January 5, or a number of other dates.

In general, in SQL, the way we make sure to include rows that represent missing data is using outer joins. For example, the query in **Listing 23** (SalesByArtist in the language-specific folders of the materials for this session) computes total sales by artist. The RIGHT JOIN to the Artist table ensures that every artist shows up in the result, even if no tracks by that artist have been sold; you can see that in the partial results shown in **Figure 13**.

Listing 23. Using an OUTER JOIN lets you include data for which there are no matches.

```
SELECT Artist.ArtistID, ;
      SUM(Quantity) AS NumTracks, ;
       SUM(Quantity * InvoiceLine.UnitPrice) AS TotalSales ;
   FROM Invoice ;
     JOIN InvoiceLine ;
       ON Invoice.InvoiceID = InvoiceLine.InvoiceID ;
     JOIN Track ;
       ON InvoiceLine.TrackID = Track.TrackID ;
     JOIN Album ;
       ON Track.AlbumId = Album.AlbumId ;
     RIGHT JOIN Artist ;
       ON Album.ArtistId = Artist.ArtistId ;
   GROUP BY Artist.ArtistId ;
   ORDER BY 1 ;
   INTO CURSOR csrSalesByArtist
```


Csrsalesbyartist

Artistid	Numtracks	Totalsales
17	27	26.73
18	25	24.75
19	16	15.84
20	5	4.95
21	29	28.71
22	87	86.13
23	4	3.96
24	7	6.93
25	.NULL.	.NULL.
26	.NULL.	.NULL.
27	20	19.80
28	.NULL.	.NULL.
29	.NULL.	.NULL.
30	.NULL.	.NULL.
31	.NULL.	.NULL.
32	.NULL.	.NULL.
33	.NULL.	.NULL.
34	.NULL.	.NULL.
35	.NULL.	.NULL.

Figure 13. When you use an outer join, you can include records from one table that have no matches in other tables.

But the key to using an outer join this way is that we have a list of all artists we can start with. In the previous query, we were missing days and the database doesn't contain a list of those. We could add a table of dates to the database, but we'd have to add data periodically to ensure it continues to cover the full period we need.

It's better to create a list of days on the fly. In VFP, we do that by creating a cursor and populating it. In SQL Server and MySQL 8, there's a way to do it that doesn't require the (admittedly minor) clean-up a cursor does.

Listing 24 (SalesByDayFull.prg in the materials for this session) shows one solution for VFP. We find the earliest and latest date in the Invoice table and then create a cursor containing all dates from the earliest to the latest. (Of course, you might prefer to specify the range of dates in some other way, perhaps passing the start and end dates as parameters.) In the daily sales query, we use an outer join to the new cursor to ensure every date is included.

Listing 24. In VFP, to include missing dates, figure out the range you need and populate a cursor. Then use an outer join.

```
SELECT MIN(InvoiceDate) AS MinDate, ;
       MAX(InvoiceDate) AS MaxDate ;
   FROM Invoice ;
   INTO CURSOR csrMinMax
```

```
CREATE CURSOR csrAllDates (tDate T)

LOCAL nCount, nNumDates, dStart
nNumDates = TTOD(csrMinMax.MaxDate) - TTOD(csrMinMax.MinDate)
dStart = TTOD(csrMinMax.MinDate)

FOR nCount = 0 TO m.nNumDates
   INSERT INTO csrAllDates VALUES (dStart + m.nCount)
ENDFOR

SELECT tDate, ;
      SUM(Quantity) AS NumTracks, ;
       SUM(Quantity * InvoiceLine.UnitPrice) AS TotalSales ;
   FROM Invoice ;
     JOIN InvoiceLine ;
       ON Invoice.InvoiceID = InvoiceLine.InvoiceID ;
     JOIN Track ;
       ON InvoiceLine.TrackID = Track.TrackID ;
     RIGHT JOIN csrAllDates ;
       ON InvoiceDate = tDate ;
   GROUP BY tDate ;
   ORDER BY 1 ;
   INTO CURSOR csrSalesByDate
```

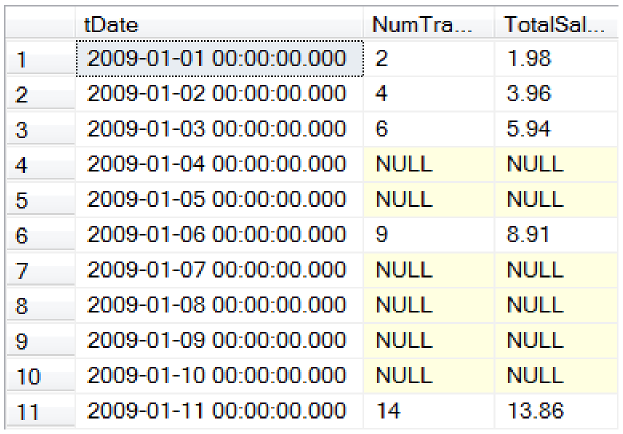
For SQL Server, we can use a recursive CTE to create the list of dates. The solution is shown in **Listing 25**. As in the VFP version, we determine the earliest and latest dates needed with an initial query. Then, the AllDates portion of the CTE creates the complete list of dates, which is used in an outer join in the main query. The first query in the CTE's UNION establishes the value of @StartDate as the anchor for the recursive CTE. The second query in the UNION then adds records by adding one day to the previous record until we reach @EndDate. The final line of the overall query removes any limit on the levels of recursion.

Listing 25. In SQL Server, a recursive CTE lets us generate the list of dates to use in an outer join.

```
DECLARE @StartDate DATETIME;
DECLARE @EndDate DATETIME;

SELECT @StartDate = MIN(InvoiceDate),
       @EndDate = MAX(InvoiceDate)
   FROM Invoice;

WITH AllDates (tDate)
AS
(SELECT @StartDate
 UNION ALL
 SELECT DATEADD(DAY, 1, tDate)
   FROM AllDates
   WHERE tDate < @EndDate)

SELECT tDate,
       SUM(Quantity) AS NumTracks,
       SUM(Quantity * InvoiceLine.UnitPrice) AS TotalSales
   FROM AllDates
```

```
     LEFT JOIN Invoice
       ON AllDates.tDate = Invoice.InvoiceDate
     LEFT JOIN InvoiceLine
       ON Invoice.InvoiceID = InvoiceLine.InvoiceID
     LEFT JOIN Track
       ON InvoiceLine.TrackID = Track.TrackID
   GROUP BY tDate
   ORDER BY 1
   OPTION (MAXRECURSION 0);
```

In the partial results shown in **Figure 14**, the dates with no sales are included with NULL for the number of tracks sold and the total sales.



	tDate	NumTra...	TotalSal...
1	2009-01-01 00:00:00.000	2	1.98
2	2009-01-02 00:00:00.000	4	3.96
3	2009-01-03 00:00:00.000	6	5.94
4	2009-01-04 00:00:00.000	NULL	NULL
5	2009-01-05 00:00:00.000	NULL	NULL
6	2009-01-06 00:00:00.000	9	8.91
7	2009-01-07 00:00:00.000	NULL	NULL
8	2009-01-08 00:00:00.000	NULL	NULL
9	2009-01-09 00:00:00.000	NULL	NULL
10	2009-01-10 00:00:00.000	NULL	NULL
11	2009-01-11 00:00:00.000	14	13.86

Figure 14. With the outer join, all dates are included, even those with no sales.

MySQL 8 also uses a recursive CTE for this problem, but the syntax is a little different. **Listing 26** shows the code. The first line sets the number of recursion levels permitted; since the data may cover many years, the default of 1000 isn't sufficient. Then, as in the SQL Server example, we find the earliest and latest invoice dates, but MySQL uses := for assignment rather than =. The second half of the UNION is also a little different than the SQL Server version; rather than the DATEADD() function (or its MySQL equivalent, DATE_ADD()), we can just use the + operator to add one day. The main query is identical to the SQL Server version.

Listing 26. As in SQL Server, in MySQL 8, you can use a recursive CTE to generate the full set of values you need to do a query that contains all dates.

```
SET @@cte_max_recursion_depth = 10000;

SELECT @StartDate := MIN(InvoiceDate),
       @EndDate := MAX(InvoiceDate)
   FROM Invoice;

WITH RECURSIVE AllDates (tDate)
AS
(SELECT @StartDate
 UNION ALL
 SELECT tDate + INTERVAL 1 DAY
   FROM AllDates
   WHERE tDate < @EndDate)

SELECT tDate,
       SUM(Quantity) AS NumTracks,
       SUM(Quantity * InvoiceLine.UnitPrice) AS TotalSales
   FROM AllDates
     LEFT JOIN Invoice
       ON AllDates.tDate = Invoice.InvoiceDate
     LEFT JOIN InvoiceLine
       ON Invoice.InvoiceID = InvoiceLine.InvoiceID
     LEFT JOIN Track
       ON InvoiceLine.TrackID = Track.TrackID
   GROUP BY tDate
   ORDER BY 1;
```

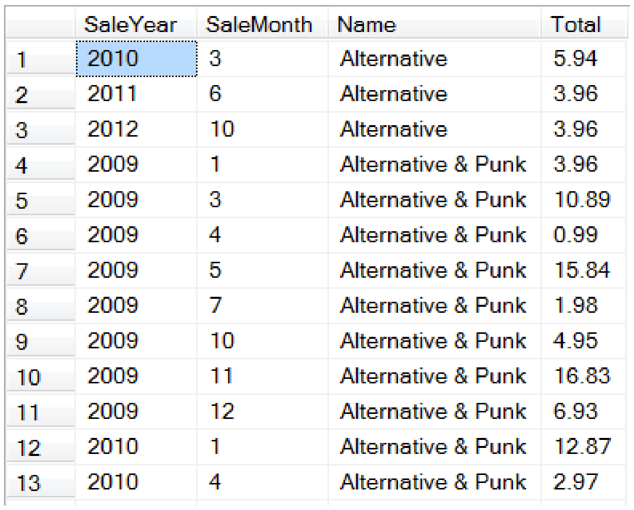
In earlier versions of MySQL, the easiest solution is to use a table of dates.

For all three languages, you can substitute 0 for the nulls, using the appropriate function: NVL() in VFP, ISNULL() in SQL Server or IFNULL() in MySQL.

The problem gets more challenging when there are two columns with values that might be missing. Suppose we want to find sales for each month for each genre. You can use a query like the one in **Listing 27** (SalesByMonthAndGenre.SQL in the SQLServer folder of the materials for this session; there are analogous versions for VFP and MySQL in their respective folders), but as the partial results in **Figure 15** show, some month/genre combinations are missing.

Listing 27. This SQL Server query finds sales by month for each genre, but it only includes the months for each genre where there were sales.

```
SELECT YEAR(InvoiceDate) as SaleYear,
       MONTH(InvoiceDate) as SaleMonth,
       Genre.Name,
       SUM(Quantity * InvoiceLine.UnitPrice) AS Total
   FROM Invoice
     JOIN InvoiceLine
       ON Invoice.InvoiceID = InvoiceLine.InvoiceID
     JOIN Track
       ON InvoiceLine.TrackID = Track.TrackID
     JOIN Genre
       ON Track.GenreID = Genre.GenreID
   GROUP BY YEAR(InvoiceDate), Month(InvoiceDate), Genre.Name
   ORDER BY 3, 1, 2;
```



	SaleYear	SaleMonth	Name	Total
1	2010	3	Alternative	5.94
2	2011	6	Alternative	3.96
3	2012	10	Alternative	3.96
4	2009	1	Alternative & Punk	3.96
5	2009	3	Alternative & Punk	10.89
6	2009	4	Alternative & Punk	0.99
7	2009	5	Alternative & Punk	15.84
8	2009	7	Alternative & Punk	1.98
9	2009	10	Alternative & Punk	4.95
10	2009	11	Alternative & Punk	16.83
11	2009	12	Alternative & Punk	6.93
12	2010	1	Alternative & Punk	12.87
13	2010	4	Alternative & Punk	2.97

Figure 15. When you get sales by month by genre, some combinations of month and genre are missing.

To get all months for all genres, we need to create those combinations and then do an outer join. To do that, we use something we usually try to avoid, a cross join (also known as a Cartesian join), in which every record from one table is matched with every record from another.

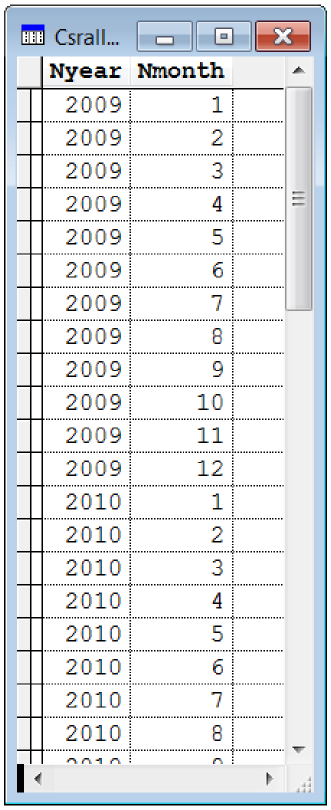
We have the complete list of genres, so we don't have to construct it, but as with days in the previous example, we need to create a list of all the months (technically, month/year pairs) that fall in the period of interest. We can do that by having separate fields for month and year or by storing a single date for each month into one field. I found that in VFP, separate month and year fields made the task easiest, while in SQL Server and MySQL, it was easier to work with a single datetime field containing the first of each month.

Listing 28 (SalesByMonthAndGenreFull.prg in the materials for this session) shows the complete VFP solution. As in the previous example, we start by finding the first and last date in the Invoice table. We then find the first day of the month for each of them and use those dates to drive a loop that fills a cursor with the list of month/year combinations we want. **Figure 16** shows the first 20 rows of that cursor.

Listing 28. To get a list of sales by month and genre that includes those months where a particular genre wasn't sold, we need to construct all the combinations of month and genre first.

```
SELECT MIN(InvoiceDate) AS MinDate, ;
       MAX(InvoiceDate) AS MaxDate ;
   FROM Invoice ;
   INTO CURSOR csrMinMax

CREATE CURSOR csrAllMonths (nYear N(4), nMonth N(2))

LOCAL nCount, dStart, dEnd, dDate
dStart = TTOD(csrMinMax.MinDate) - DAY(csrMinMax.MinDate) + 1
dEnd = TTOD(csrMinMax.MaxDate) - DAY(csrMinMax.MaxDate) + 1

dDate = dStart
```

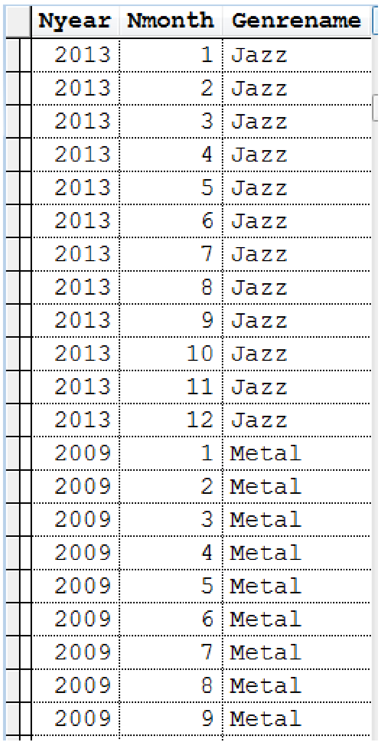
```
DO WHILE m.dDate <= m.dEnd
   INSERT INTO csrAllMonths VALUES (YEAR(m.dDate), MONTH(m.dDate))
   dDate = GOMONTH(m.dDate, 1)
ENDDO

SELECT nYear, nMonth, Name AS GenreName ;
   FROM csrAllMonths, Genre ;
   INTO CURSOR csrAllMonthsAndGenres

SELECT nYear as SaleYear, ;
       nMonth as SaleMonth, ;
       GenreName, ;
       SUM(Quantity * InvoiceLine.UnitPrice) AS Total ;
   FROM Invoice ;
     JOIN InvoiceLine ;
       ON Invoice.InvoiceID = InvoiceLine.InvoiceID ;
     JOIN Track ;
       ON InvoiceLine.TrackID = Track.TrackID ;
     JOIN Genre ;
       ON Track.GenreID = Genre.GenreID ;
     RIGHT JOIN csrAllMonthsAndGenres ;
       ON nYear = YEAR(InvoiceDate) ;
       AND nMonth = MONTH(InvoiceDate) ;
       AND GenreName == Genre.Name ;
   GROUP BY 1, 2, 3;
   ORDER BY 3, 1, 2 ;
   INTO CURSOR csrSalesByMonth
```



Nyear	Nmonth
2009	1
2009	2
2009	3
2009	4
2009	5
2009	6
2009	7
2009	8
2009	9
2009	10
2009	11
2009	12
2010	1
2010	2
2010	3
2010	4
2010	5
2010	6
2010	7
2010	8

Figure 16. The first part of the code in **Listing 28** creates this cursor of month/year combinations.

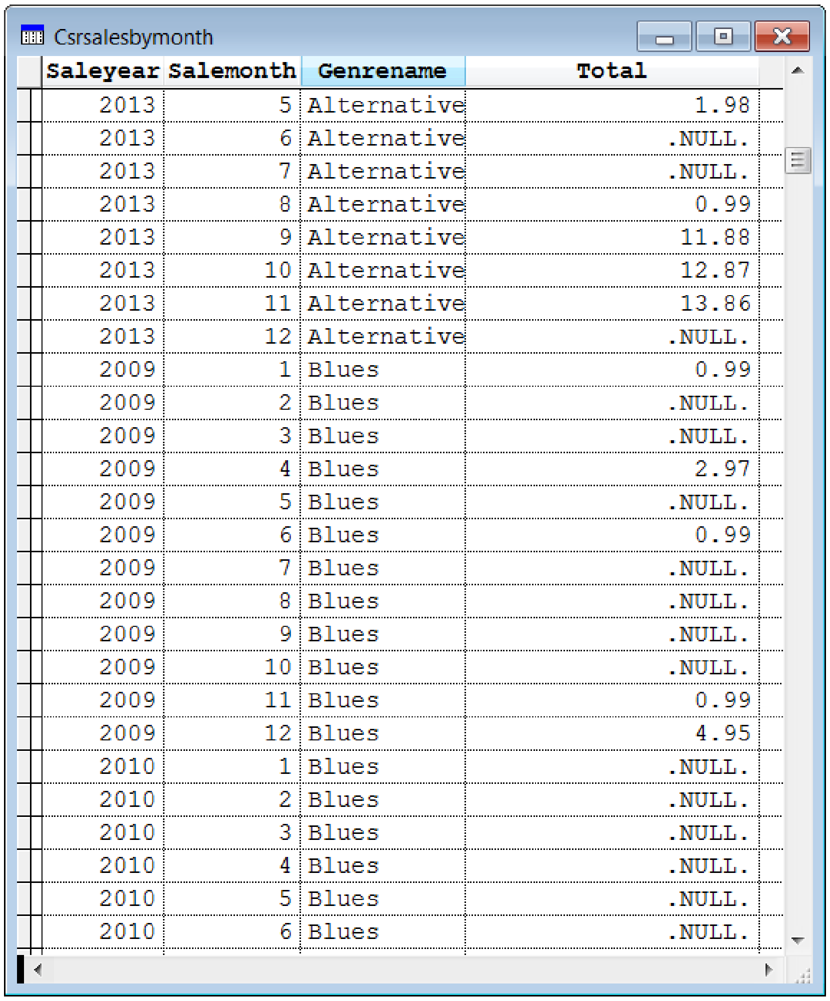
The query after the loop uses the old-style FROM clause to specify a cross-join. By listing the two tables and not providing a join condition, every record in csrAllMonths is matched with each record in Genre. **Figure 17** shows a few rows from the middle of the cursor created by this query; the complete cursor has 1500 rows. (There are 60 month/year combinations in the data and 25 genres.)



Nyear	Nmonth	Genrename
2013	1	Jazz
2013	2	Jazz
2013	3	Jazz
2013	4	Jazz
2013	5	Jazz
2013	6	Jazz
2013	7	Jazz
2013	8	Jazz
2013	9	Jazz
2013	10	Jazz
2013	11	Jazz
2013	12	Jazz
2009	1	Metal
2009	2	Metal
2009	3	Metal
2009	4	Metal
2009	5	Metal
2009	6	Metal
2009	7	Metal
2009	8	Metal
2009	9	Metal

Figure 17. Once we have the list of month/year combinations, a cross join gives us a record match every genre with each month and year.

Finally, we use an outer join in the query that computes monthly sales to ensure that every month/year/genre combination appears in the final results. **Figure 18** shows partial results; the month/year/genre combinations with no sales show nulls in the Total field. (As in the previous example, you can use NVL() to replace those nulls with zeroes, if you wish.)


Csrsalesbymonth

Saleyear	Salemonth	Genrename	Total
2013	5	Alternative	1.98
2013	6	Alternative	.NULL.
2013	7	Alternative	.NULL.
2013	8	Alternative	0.99
2013	9	Alternative	11.88
2013	10	Alternative	12.87
2013	11	Alternative	13.86
2013	12	Alternative	.NULL.
2009	1	Blues	0.99
2009	2	Blues	.NULL.
2009	3	Blues	.NULL.
2009	4	Blues	2.97
2009	5	Blues	.NULL.
2009	6	Blues	0.99
2009	7	Blues	.NULL.
2009	8	Blues	.NULL.
2009	9	Blues	.NULL.
2009	10	Blues	.NULL.
2009	11	Blues	0.99
2009	12	Blues	4.95
2010	1	Blues	.NULL.
2010	2	Blues	.NULL.
2010	3	Blues	.NULL.
2010	4	Blues	.NULL.
2010	5	Blues	.NULL.
2010	6	Blues	.NULL.

Figure 18. An outer join combines the complete list of month/year/genre combinations with the computed monthly sales totals to get us the full results.

The SQL Server version, shown in **Listing 29** (SalesByMonthAndGenreFull.SQL in the SQLServer folder of the materials for this session) uses a series of CTEs. The first uses the same technique as in **Listing 25** to build the list of months. In this case, each record contains a single datetime field containing the first day of the particular month.

Listing 29. In SQL Server, we can use a series of CTEs to pull together all the data we need, but the basic approach is quite similar.

```
DECLARE @StartDate DATETIME;
DECLARE @EndDate DATETIME;

SELECT @StartDate = MIN(InvoiceDate),
       @EndDate = MAX(InvoiceDate)
   FROM Invoice;

SET @StartDate = DATEADD(MONTH, DATEDIFF(MONTH, 0, @StartDate), 0);
SET @EndDate = DATEADD(MONTH, DATEDIFF(MONTH, 0, @EndDate), 0);

WITH AllMonths (tDate)
```

```
AS
(SELECT @StartDate AS tDate
 UNION ALL
 SELECT DATEADD(m, 1, tDate)
   FROM AllMonths
   WHERE tDate < @EndDate),

MonthsByGenres (tDate, GenreName)
AS
(SELECT tDate, Name
   FROM AllMonths
     CROSS JOIN Genre),

SalesByDay (InvoiceDate, Name, Total)
AS
(SELECT InvoiceDate,
        Genre.Name,
        SUM(Quantity * InvoiceLine.UnitPrice) AS Total
   FROM Invoice
     JOIN InvoiceLine
       ON Invoice.InvoiceID = InvoiceLine.InvoiceID
     JOIN Track
       ON InvoiceLine.TrackID = Track.TrackID
     JOIN Genre
       ON Track.GenreID = Genre.GenreID
   GROUP BY InvoiceDate, Genre.Name)

SELECT YEAR(tDate) as SaleYear,
       MONTH(tDate) as SaleMonth,
       MonthsByGenres.GenreName,
       SUM(Total) AS Total
   FROM MonthsByGenres
     LEFT JOIN SalesByDay
       ON InvoiceDate >= tDate
       AND InvoiceDate < DATEADD(MONTH, 1, tDate)
       AND MonthsByGenres.GenreName = SalesByDay.Name
   GROUP BY tDate, GenreName
   ORDER BY 3, 1, 2;
```

The second CTE does a cross join between the first and the Genre table. SQL Server lets you simply specify CROSS JOIN.

The third CTE isn't directly related to the first two; it aggregates the sales for each genre by day. Because the MonthsByGenres CTE contains the first day of each month, we need to keep the invoice date in SalesByDay in order to be able to do the join.

Finally, the main query joins MonthsByGenres with SalesByDay to get the final results.

The MySQL 8 version of the solution is quite similar to the SQL Server version, but as with the simpler problem, reflects some language differences. It's shown in **Listing 30** (and included as SalesByMonthAndGenreFull.sql in the MySQL folder of the materials for this session). In addition to the differences discussed for the simpler problem, the technique for finding the first day of the month of a specified date is a little different than the SQL Server version, reflecting that MySQL's Date_Add() function takes different parameters than SQL Server's DATEADD().

Listing 30. In MySQL, the solution to filling in missing data on multiple dimensions also uses a series of CTEs. It's analogous to the SQL Server solution, but with some syntactic differences.

```
SELECT @StartDate := MIN(InvoiceDate),
       @EndDate := MAX(InvoiceDate)
   FROM Invoice;

SET @StartDate := DATE_ADD(@StartDate, interval -day(@StartDate) + 1 day);
SET @EndDate := DATE_ADD(@EndDate, interval -day(@EndDate) + 1 day);

WITH RECURSIVE AllMonths (tDate)
AS
(SELECT @StartDate AS tDate
 UNION ALL
 SELECT tDate + interval 1 month
   FROM AllMonths
   WHERE tDate < @EndDate),

MonthsByGenres (tDate, GenreName)
AS
(SELECT tDate, Name
   FROM AllMonths
     CROSS JOIN Genre),

SalesByDay (InvoiceDate, Name, Total)
AS
(SELECT InvoiceDate,
        Genre.Name,
        SUM(Quantity * InvoiceLine.UnitPrice) AS Total
   FROM Invoice
     JOIN InvoiceLine
       ON Invoice.InvoiceID = InvoiceLine.InvoiceID
     JOIN Track
       ON InvoiceLine.TrackID = Track.TrackID
     JOIN Genre
       ON Track.GenreID = Genre.GenreID
   GROUP BY InvoiceDate, Genre.Name)

SELECT YEAR(tDate) as SaleYear,
       MONTH(tDate) as SaleMonth,
       MonthsByGenres.GenreName,
       SUM(Total) AS Total
   FROM MonthsByGenres
     LEFT JOIN SalesByDay
     ON InvoiceDate >= tDate
     AND InvoiceDate < tDate + interval 1 month
     AND MonthsByGenres.GenreName = SalesByDay.Name

   GROUP BY tDate, GenreName
    ORDER BY 3, 1, 2;
```

As with the previous example, the easiest way to do this in earlier versions of MySQL is to use a calendar table. Once that exists, the rest of the solution is quite similar to the MySQL 8 versions.

Running totals

It's easy to compute totals such as the total sales for a day or a month. But users often want to see a running total, for example, for each day of the month, the sales for the month through that day.

Suppose we want to see the daily sales for each genre, along with a running total for the month. Getting daily sales by genre is straightforward and is quite similar in all versions of SQL. **Listing 31** shows a version that runs in both MySQL and SQL Server. The VFP version differs only in punctuation and in adding INTO CURSOR. (The example is available as DailySalesByGenre in the language-specific folders of the materials for this session)

Listing 31. Totaling sales by day for each genre is simple.

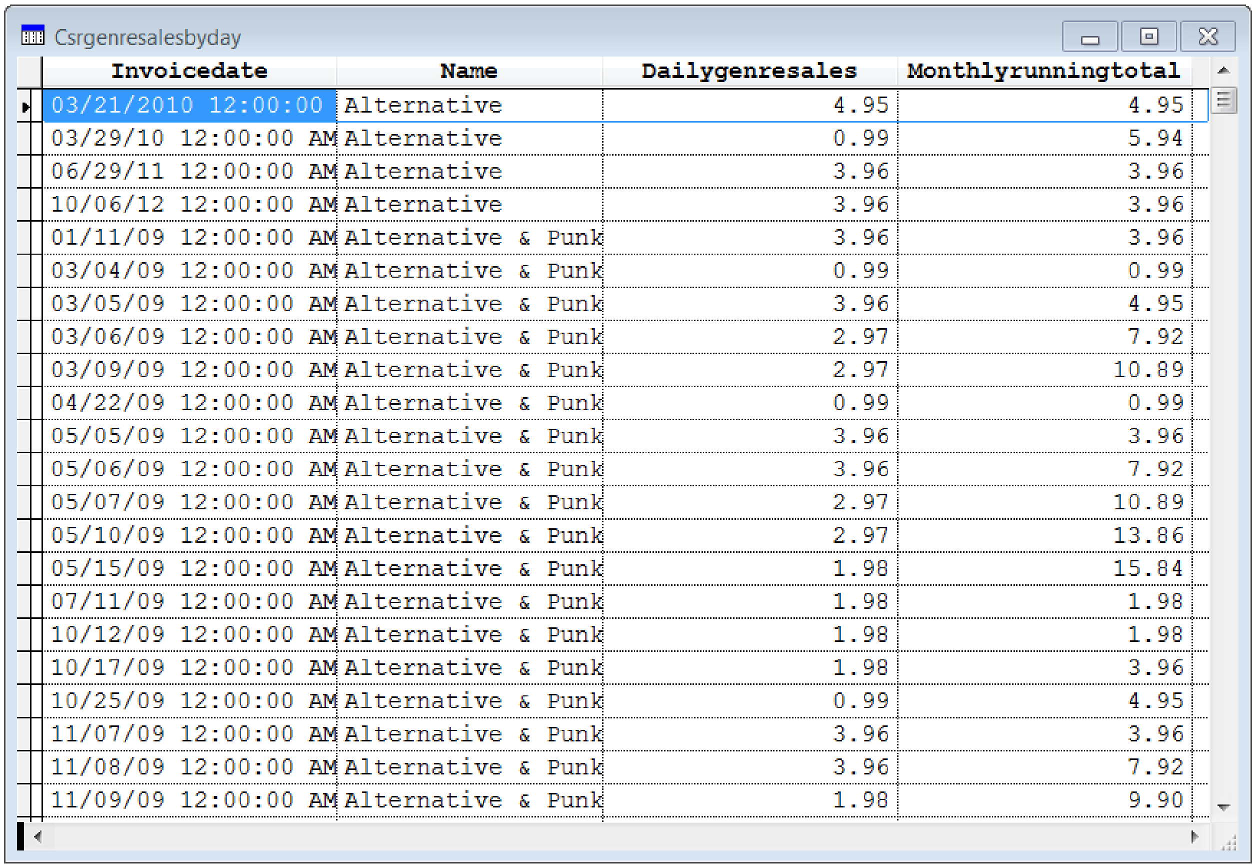
```
SELECT InvoiceDate, Genre.Name,
       SUM(Quantity * invoiceline.UnitPrice) AS DailySales
   FROM invoice
     JOIN invoiceline
       ON invoice.InvoiceId = invoiceline.InvoiceId
     JOIN track
       ON invoiceline.TrackId = track.TrackId
     JOIN genre
       ON track.GenreId = genre.GenreId
   GROUP BY InvoiceDate, Genre.Name
   ORDER BY Genre.Name, InvoiceDate;
```

Adding running totals is another problem that requires a mix of SQL and Xbase in VFP. First, we run a query much like the previous example to get the daily totals and then, we loop through the result to compute the running total for the current month. **Listing 32** (SalesWithMonthlyRunningTotal.PRG in the VFP folder of the materials for this session) shows the code. The only difference between the first query and one that computes only daily sales is the addition of an empty column to hold the running total and the use of READWRITE to allow us to modify that column later. The loop keeps track of what month and genre we're working on and when we reach the end of a month/genre combination, resets the running total to 0. Partial results are shown in **Figure 19**.

Listing 32. In VFP, computing running totals requires a loop.

```
SELECT InvoiceDate, PADR(Genre.Name, 120) as Name, ;
       SUM(Quantity * InvoiceLine.UnitPrice) AS DailyGenreSales, ;
       CAST(0 as N(20,2)) AS MonthlyRunningTotal ;
     FROM Invoice ;
     JOIN InvoiceLine ;
       ON Invoice.InvoiceID = InvoiceLine.InvoiceID ;
     JOIN Track ;
       ON InvoiceLine.TrackID = Track.TrackID ;
     JOIN Genre ;
```

```
        ON Track.GenreId = Genre.GenreId ;
   GROUP BY 1, 2 ;
   ORDER BY 2, 1 ;
   INTO CURSOR csrGenreSalesByDay READWRITE

* Compute running totals
LOCAL nYear, nMonth, nTotal, cGenre
STORE 0 TO nYear, nMonth, nTotal
cGenre = ''

SCAN
    IF nYear <> YEAR(InvoiceDate) OR ;
       nMonth <> MONTH(InvoiceDate) OR ;
       NOT (m.cGenre == UPPER(Name))

        nYear = YEAR(InvoiceDate)
        nMonth = MONTH(InvoiceDate)
        cGenre = UPPER(Name)
        nTotal = 0
    ENDIF
    nTotal = m.nTotal + DailyGenreSales
    REPLACE MonthlyRunningTotal WITH m.nTotal
ENDSCAN
```



Csrgenresalesbyday

Invoicedate	Name	Dailygenresales	Monthlyrunningtotal
03/21/2010 12:00:00	Alternative	4.95	4.95
03/29/10 12:00:00 AM	Alternative	0.99	5.94
06/29/11 12:00:00 AM	Alternative	3.96	3.96
10/06/12 12:00:00 AM	Alternative	3.96	3.96
01/11/09 12:00:00 AM	Alternative & Punk	3.96	3.96
03/04/09 12:00:00 AM	Alternative & Punk	0.99	0.99
03/05/09 12:00:00 AM	Alternative & Punk	3.96	4.95
03/06/09 12:00:00 AM	Alternative & Punk	2.97	7.92
03/09/09 12:00:00 AM	Alternative & Punk	2.97	10.89
04/22/09 12:00:00 AM	Alternative & Punk	0.99	0.99
05/05/09 12:00:00 AM	Alternative & Punk	3.96	3.96
05/06/09 12:00:00 AM	Alternative & Punk	3.96	7.92
05/07/09 12:00:00 AM	Alternative & Punk	2.97	10.89
05/10/09 12:00:00 AM	Alternative & Punk	2.97	13.86
05/15/09 12:00:00 AM	Alternative & Punk	1.98	15.84
07/11/09 12:00:00 AM	Alternative & Punk	1.98	1.98
10/12/09 12:00:00 AM	Alternative & Punk	1.98	1.98
10/17/09 12:00:00 AM	Alternative & Punk	1.98	3.96
10/25/09 12:00:00 AM	Alternative & Punk	0.99	4.95
11/07/09 12:00:00 AM	Alternative & Punk	3.96	3.96
11/08/09 12:00:00 AM	Alternative & Punk	3.96	7.92
11/09/09 12:00:00 AM	Alternative & Punk	1.98	9.90

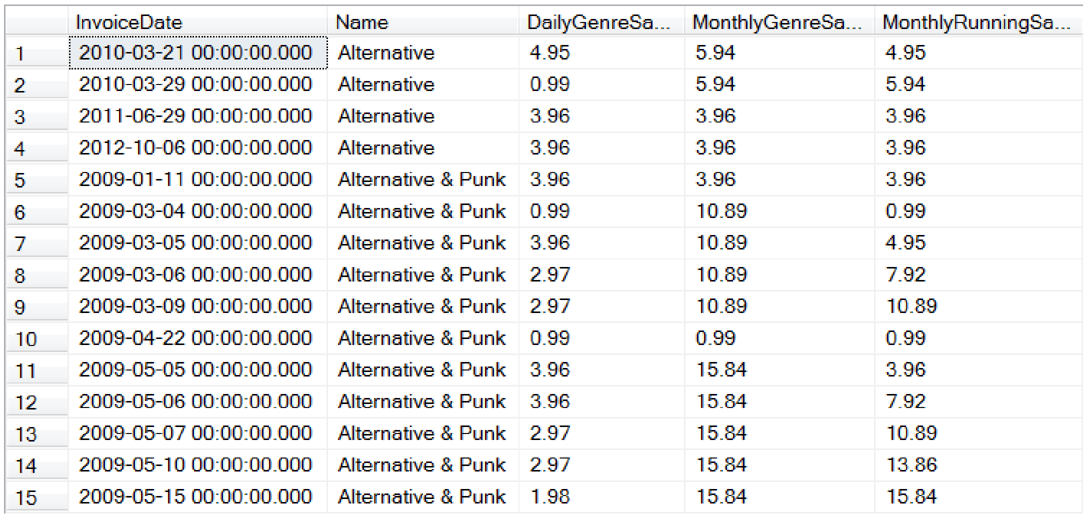
Figure 19. To compute running totals for each month, first collect the data and then loop through computing the running total.

The VFP folder in the materials for this session also includes SalesWithRunningTotal.PRG, a more complex program that includes the total for the month in each record.

In SQL Server and MySQL 8, this is another problem solved by OVER. Beginning in SQL Server 2012, when using OVER with an aggregate function, you can include an ORDER BY clause. MySQL 8 supports this, as well. Doing so computes a running total, running count or moving average (depending on which aggregate function you're using). **Listing 33** shows the SQL Server and MySQL solution (SalesWithRunningTotal.SQL in the appropriate folder of the materials for this session); it looks less like the daily sales by genre query than in the VFP solution. The query includes the monthly total to demonstrate the difference between including and omitting ORDER BY when aggregating with OVER. Although we casually say "monthly totals" or "monthly running totals," we need to include both month and year in the partition, so that we're totaling only data from one calendar month, not from the same month across all years. **Figure 20** shows partial results.

Listing 33. Computing running totals is easy in SQL Server, using OVER.

```
SELECT DISTINCT InvoiceDate, Genre.Name,
       SUM(Quantity * InvoiceLine.UnitPrice)
        OVER (PARTITION by Track.GenreID, InvoiceDate) AS DailyGenreSales,
       SUM(Quantity * InvoiceLine.UnitPrice)
        OVER (PARTITION by Track.GenreID, MONTH(InvoiceDate), YEAR(InvoiceDate))
        AS MonthlyGenreSales,
      SUM(Quantity * InvoiceLine.UnitPrice)
        OVER (PARTITION BY Track.GenreID, MONTH(InvoiceDate), YEAR(InvoiceDate)
              ORDER BY InvoiceDate) AS MonthlyRunningSales
   FROM Invoice
     JOIN InvoiceLine
       ON Invoice.InvoiceID = InvoiceLine.InvoiceID
     JOIN Track
       ON InvoiceLine.TrackID = Track.TrackID
     JOIN Genre
       ON Track.GenreId = Genre.GenreId
   ORDER BY 2, 1;
```



	InvoiceDate	Name	DailyGenreSa...	MonthlyGenreSa...	MonthlyRunningSa...
1	2010-03-21 00:00:00.000	Alternative	4.95	5.94	4.95
2	2010-03-29 00:00:00.000	Alternative	0.99	5.94	5.94
3	2011-06-29 00:00:00.000	Alternative	3.96	3.96	3.96
4	2012-10-06 00:00:00.000	Alternative	3.96	3.96	3.96
5	2009-01-11 00:00:00.000	Alternative & Punk	3.96	3.96	3.96
6	2009-03-04 00:00:00.000	Alternative & Punk	0.99	10.89	0.99
7	2009-03-05 00:00:00.000	Alternative & Punk	3.96	10.89	4.95
8	2009-03-06 00:00:00.000	Alternative & Punk	2.97	10.89	7.92
9	2009-03-09 00:00:00.000	Alternative & Punk	2.97	10.89	10.89
10	2009-04-22 00:00:00.000	Alternative & Punk	0.99	0.99	0.99
11	2009-05-05 00:00:00.000	Alternative & Punk	3.96	15.84	3.96
12	2009-05-06 00:00:00.000	Alternative & Punk	3.96	15.84	7.92
13	2009-05-07 00:00:00.000	Alternative & Punk	2.97	15.84	10.89
14	2009-05-10 00:00:00.000	Alternative & Punk	2.97	15.84	13.86
15	2009-05-15 00:00:00.000	Alternative & Punk	1.98	15.84	15.84

Figure 20. With the monthly total in the results, you can see the running total for the month approach the final value.

While MySQL 8.0 supports OVER, earlier versions did not. But there's still a way to compute running totals there. The way to do it is not that different from the VFP solution, except that it can all be done in the context of a single (complex) query.

As in some earlier examples, the trick is to use variables in the query. In this case, they keep track of what month, year and genre you last saw, use them to decide whether the current record is from the same month, year and genre, and then update them. The solution relies on the fact that the MySQL engine evaluates the fields for each record in the order they're listed in the query. **Listing 34** (SalesWithRunningTotalMonthlyPre8.sql in the MySQL folder on the materials for this session; that folder also includes SalesWithRunningTotalDailyPre8.sql, which uses the same technique to compute a running total of sales on a daily basis) shows the complete solution; we'll break it down below.

Listing 34. In versions before MySQL 8, you can compute running totals using variables to track where you are.

```
SELECT InvoiceDate, Name, DailySales,
       IF(@PrevMonth = MONTH(InvoiceDate)
            and @PrevYear = YEAR(InvoiceDate)
            and @PrevGenre = Name,
          @RunningSales := @RunningSales + DailySales,
          @RunningSales := DailySales) AS MonthlyRunning,
          @PrevMonth := MONTH(InvoiceDate),
          @PrevYear := YEAR(InvoiceDate),
          @PrevGenre := Name
   FROM (SELECT InvoiceDate, Genre.Name,
                  SUM(Quantity * invoiceline.UnitPrice) AS DailySales
           FROM invoice
             JOIN invoiceline
               ON invoice.InvoiceId = invoiceline.InvoiceId
             JOIN track
               ON invoiceline.TrackId = track.TrackId
             JOIN genre
```

```
              ON track.GenreId = genre.GenreId
          GROUP BY InvoiceDate, Genre.Name
          ORDER BY Genre.Name, InvoiceDate) DailyTotals
      JOIN (SELECT @RunningSales := 0,
                   @PrevMonth := 0,
                   @PrevYear := 0,
                   @PrevGenre := '') RS
   ORDER BY Name, InvoiceDate;
```

The query joins two derived tables. The first is the query that computes daily totals; it's the same query as in Listing 31. The second creates and initializes four variables: @RunningSales holds the running total of sales for the current month; @PrevMonth holds the month for the prior record; @PrevYear holds the year for the prior record; @PrevGenre holds the genre name for the prior record. There's no ON clause, so the two are joined via a cross join. Since there's only one "record" in the second derived table, there's one result record for each record in DailyTotals.

In the field list, the running total is computed by the expression beginning with IF. MySQL's IF() is like VFP's IIF(). It evaluates the first expression you pass to determine whether to return the value of the second expression or the value of the third. The first expression compares the variables @PrevMonth, @PrevYear and @PrevGenre to the corresponding values for the current record. If they match, we're still on the same month and genre, and so the sales for this date should be added to the running total we've computed so far. If any of the three don't match, we've started a new month, so we reset the running total to just the DailySales value from the current record.

Once we've done that, we update the three variables (@PrevMonth, @PrevYear, @PrevGenre) to their values in the current record. Technically, we only need to do that when one of them has changed, but it's easier to just do it every time.

This solution depends on processing the records in the right order. Note that the first derived table (DailyTotals) sorts by genre name and invoice date to make sure that's the case.

Matching values when aggregating

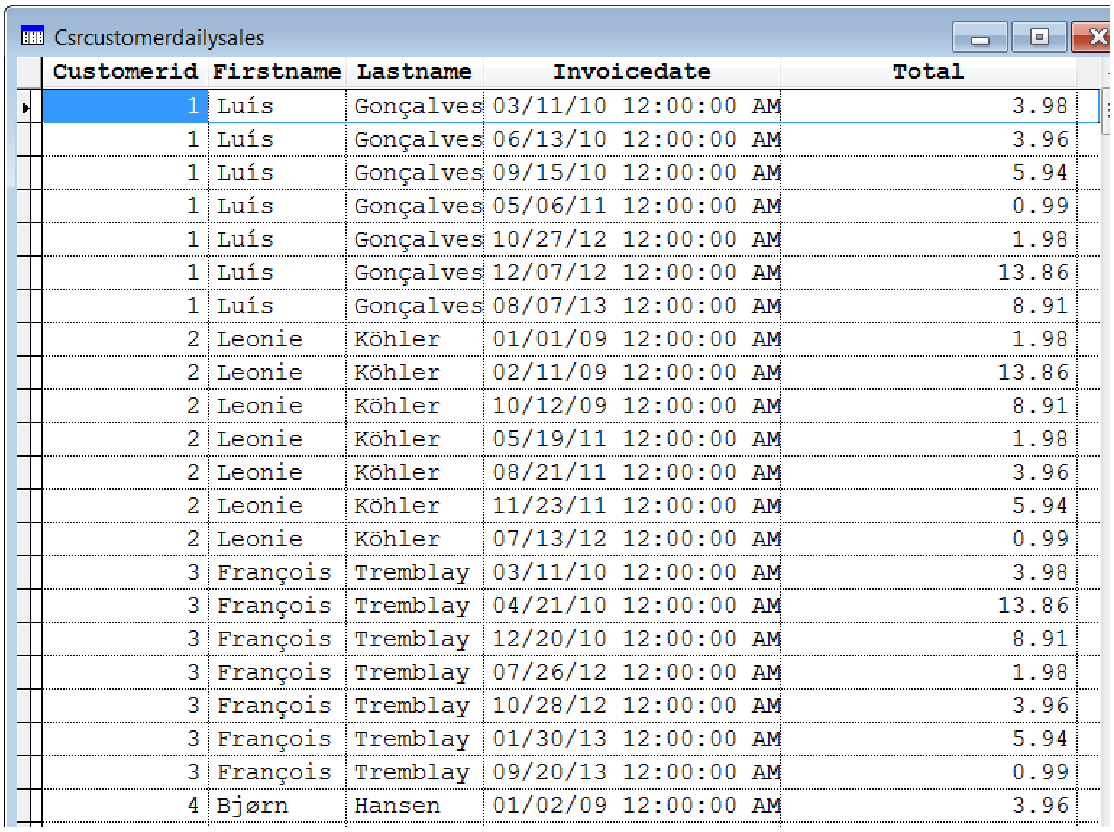
Another problem that's easy with SQL is finding the first or last or highest or lowest value for something. Just aggregate using the MIN() and MAX() values. But when you've done that, getting at the other values from the record that supplies the minimum or maximum value isn't straightforward.

To demonstrate, we'll find the most recent sale for each customer, including the total amount of that sale. (Technically, the solutions here actually find the most recent day on which a customer bought anything and the total spent on that day. The Chinook data doesn't include the time for each sale, only the date.) Finding out when each customer last bought something is easy. Attaching the total for that sale to it is the tricky part. (In fact, there's a quirk of VFP that means you can do this quite easily, but it relies on an undocumented feature, requires you to SET ENGINEBEHAVIOR 70, and works only if you're trying to find the most recent sale, but not if you're trying to find the first sale for each customer, so we're not going to look at it here. There's discussion of this issue in http://www.tomorrowssolutionsllc.com/Articles/Using%20SQL%20the%20VFP%208%20way.PDF.)

In VFP, while you can do it the right way with SQL alone, it takes a series of queries, as shown in **Listing 35** (CustomerMostRecentSale.prg in the VFP folder of the materials for this session). First, we consolidate the sale data to get one record per customer per day showing the total spent by that customer that day. **Figure 21** shows partial results from that query. The second query aggregates the data in the first to find the most recent date on which each customer bought something; partial results are shown in **Figure 22**. The key point here is that, while the second query tells us when each customer last bought something, it doesn't tell us how much they spent. You can't just add Total to this query because every field in a grouped query has to either be part of the GROUP BY expression or use one of the aggregate functions. On the other hand, while you could add MAX(Total), that wouldn't tell you the total for the most recent sale, but the most that customer ever spent in a day. In order to find the total for the most recent sale, we need to join the two cursors we've created so far, matching records on both CustomerID and on the date of the most recent sale. **Figure 23** shows partial results.

Listing 35. To find other data that goes along with the minimum or maximum value of a field, you need several steps.

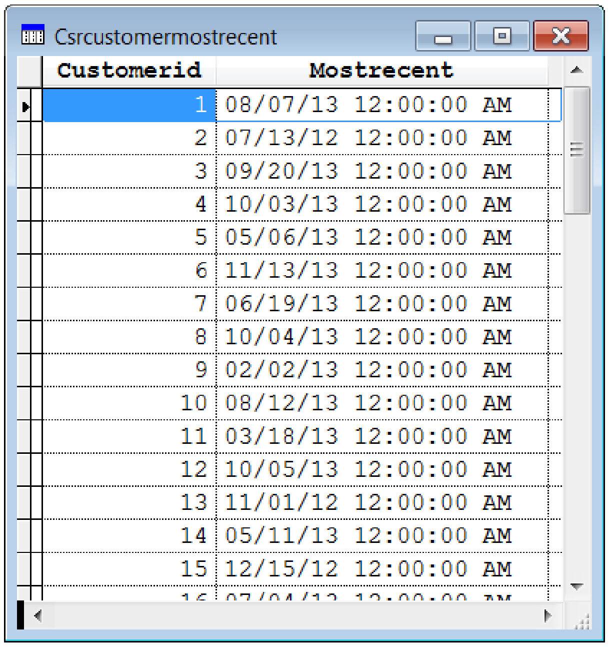
```
SELECT Customer.CustomerId, FirstName, LastName, ;
       InvoiceDate, SUM(Quantity * UnitPrice) AS Total;
   FROM Customer ;
     JOIN Invoice ;
       ON Customer.CustomerId = Invoice.CustomerId ;
     JOIN InvoiceLine ;
       ON Invoice.InvoiceId = InvoiceLine.InvoiceId ;
   GROUP BY Customer.CustomerId, FirstName, LastName, InvoiceDate ;
   INTO CURSOR csrCustomerDailySales

SELECT CustomerId, MAX(InvoiceDate) AS MostRecent;
   FROM csrCustomerDailySales ;
   GROUP BY CustomerID ;
   INTO CURSOR csrCustomerMostRecent

SELECT csrCustomerDailySales.CustomerId, FirstName, LastName, ;
       InvoiceDate, Total ;
   FROM csrCustomerDailySales ;
     JOIN csrCustomerMostRecent ;
       ON csrCustomerDailySales.CustomerID = csrCustomerMostRecent.CustomerID ;
       AND csrCustomerDailySales.InvoiceDate = csrCustomerMostRecent.MostRecent ;
   INTO CURSOR csrCustomerMostRecentSale
```



Csrcustomerdailysales

Customerid	Firstname	Lastname	Invoicedate	Total
1	Luís	Gonçalves	03/11/10 12:00:00 AM	3.98
1	Luís	Gonçalves	06/13/10 12:00:00 AM	3.96
1	Luís	Gonçalves	09/15/10 12:00:00 AM	5.94
1	Luís	Gonçalves	05/06/11 12:00:00 AM	0.99
1	Luís	Gonçalves	10/27/12 12:00:00 AM	1.98
1	Luís	Gonçalves	12/07/12 12:00:00 AM	13.86
1	Luís	Gonçalves	08/07/13 12:00:00 AM	8.91
2	Leonie	Köhler	01/01/09 12:00:00 AM	1.98
2	Leonie	Köhler	02/11/09 12:00:00 AM	13.86
2	Leonie	Köhler	10/12/09 12:00:00 AM	8.91
2	Leonie	Köhler	05/19/11 12:00:00 AM	1.98
2	Leonie	Köhler	08/21/11 12:00:00 AM	3.96
2	Leonie	Köhler	11/23/11 12:00:00 AM	5.94
2	Leonie	Köhler	07/13/12 12:00:00 AM	0.99
3	François	Tremblay	03/11/10 12:00:00 AM	3.98
3	François	Tremblay	04/21/10 12:00:00 AM	13.86
3	François	Tremblay	12/20/10 12:00:00 AM	8.91
3	François	Tremblay	07/26/12 12:00:00 AM	1.98
3	François	Tremblay	10/28/12 12:00:00 AM	3.96
3	François	Tremblay	01/30/13 12:00:00 AM	5.94
3	François	Tremblay	09/20/13 12:00:00 AM	0.99
4	Bjørn	Hansen	01/02/09 12:00:00 AM	3.96

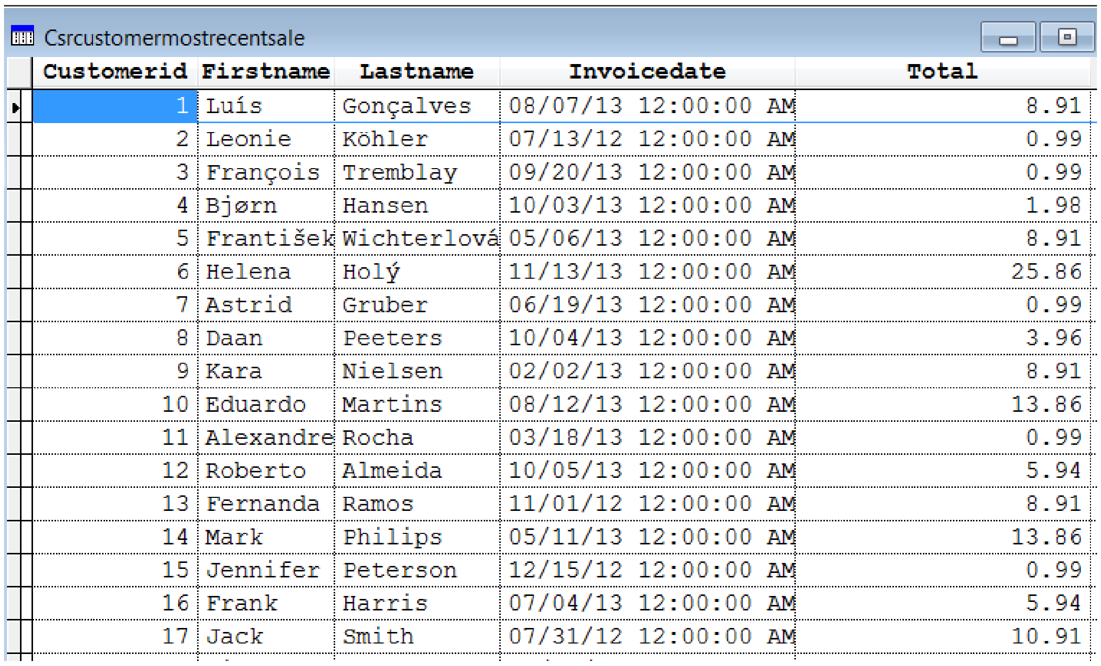
Figure 21. It's easy to compute the daily sales for each customer.



Csrcustomermostrecent

Customerid	Mostrecent
1	08/07/13 12:00:00 AM
2	07/13/12 12:00:00 AM
3	09/20/13 12:00:00 AM
4	10/03/13 12:00:00 AM
5	05/06/13 12:00:00 AM
6	11/13/13 12:00:00 AM
7	06/19/13 12:00:00 AM
8	10/04/13 12:00:00 AM
9	02/02/13 12:00:00 AM
10	08/12/13 12:00:00 AM
11	03/18/13 12:00:00 AM
12	10/05/13 12:00:00 AM
13	11/01/12 12:00:00 AM
14	05/11/13 12:00:00 AM
15	12/15/12 12:00:00 AM
16	07/04/13 12:00:00 AM

Figure 22. Once we have the total sales by day for each customer, a second query finds the most recent day that customer bought something, but it doesn't tell us how much they spent that day.



Customerid	Firstname	Lastname	Invoicedate	Total
1	Luís	Gonçalves	08/07/13 12:00:00 AM	8.91
2	Leonie	Köhler	07/13/12 12:00:00 AM	0.99
3	François	Tremblay	09/20/13 12:00:00 AM	0.99
4	Bjørn	Hansen	10/03/13 12:00:00 AM	1.98
5	František	Wichterlová	05/06/13 12:00:00 AM	8.91
6	Helena	Holý	11/13/13 12:00:00 AM	25.86
7	Astrid	Gruber	06/19/13 12:00:00 AM	0.99
8	Daan	Peeters	10/04/13 12:00:00 AM	3.96
9	Kara	Nielsen	02/02/13 12:00:00 AM	8.91
10	Eduardo	Martins	08/12/13 12:00:00 AM	13.86
11	Alexandre	Rocha	03/18/13 12:00:00 AM	0.99
12	Roberto	Almeida	10/05/13 12:00:00 AM	5.94
13	Fernanda	Ramos	11/01/12 12:00:00 AM	8.91
14	Mark	Philips	05/11/13 12:00:00 AM	13.86
15	Jennifer	Peterson	12/15/12 12:00:00 AM	0.99
16	Frank	Harris	07/04/13 12:00:00 AM	5.94
17	Jack	Smith	07/31/12 12:00:00 AM	10.91

Figure 23. To find the amount the customer spent most recently, we join the cursor of daily totals with the cursor showing the most recent sales.

SQL Server and MySQL 8 have a more direct route to the result, using a couple of CTEs and the OVER clause. OVER supports a pair of functions named FIRST_VALUE() and LAST_VALUE(). FIRST_VALUE() accepts an expression and returns the value of that expression in each partition based on the order you specify. In this case, it's exactly what we need. **Listing 36** shows the query (CustomerMostRecentSale.sql in the appropriate folder of the materials for this session). The first CTE is the same as the first query in the VFP solution; it computes the daily total for each customer. The second CTE uses both MAX() and FIRST_VALUE() with OVER to find the most recent date for each customer and the corresponding sales total. Note that the two uses of OVER require different details. For MAX(), we need just a PARTITION BY clause, telling us to do this calculation for each customer. With FIRST_VALUE(), we also need ORDER BY to indicate which record is first in the partition. The second CTE has one record for each customer for each day that customer made a purchase, but the records for each customer are identical. So the main query keeps just one record per customer.

Listing 36. In SQL Server and MySQL 8, a couple of CTEs along with a couple of OVER clauses let you find both the date and the total of each customer's most recent purchase.

```
WITH csrCustomerDailySales (CustomerID, FirstName, LastName, InvoiceDate, Total)
AS (SELECT Customer.CustomerId, FirstName, LastName,
           InvoiceDate, SUM(Quantity * UnitPrice)
   FROM Customer
     JOIN Invoice
       ON Customer.CustomerId = Invoice.CustomerId
     JOIN InvoiceLine
       ON Invoice.InvoiceId = InvoiceLine.InvoiceId
   GROUP BY Customer.CustomerId, FirstName, LastName, InvoiceDate),
csrWithMax (CustomerID, FirstName, LastName, InvoiceDate, Total)
AS (SELECT CustomerId, FirstName, LastName,
           MAX(InvoiceDate) OVER (PARTITION BY CustomerId),
           FIRST_VALUE(Total)
             OVER (PARTITION BY CustomerId ORDER BY InvoiceDate DESC)
   FROM csrCustomerDailySales)

SELECT DISTINCT *
   FROM csrWithMax
```

While it's not an issue for this example, it's important to know that LAST_VALUE() does not behave the same as FIRST_VALUE(). By default, the function returns the "running last value," that is, the one you're up to. The secret to getting the actual last value in the partition is to use the window frame notation (described in "Introducing OVER," earlier in this paper). The default frame is RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW. To get a value from the last record of the partition, we need RANGE BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING.

Dealing with hierarchical data

Most databases we deal with include hierarchical data, with multiple tables containing different levels of the hierarchy. But certain kinds of hierarchical data are best stored by including multiple levels of the hierarchy in a single table. Typical examples include organizational charts for a business, where each employee record includes a pointer to the employee's supervisor, who is also an employee; family relationships, where each person may be related to other people in one of several ways; and bills of materials, where a part may be constructed of other parts, which in turn are constructed of other parts.

While this structure is handy for storing the data, extracting it is trickier. In fact, if there are more than two levels of data, you can't use a single query to extract the whole hierarchy.

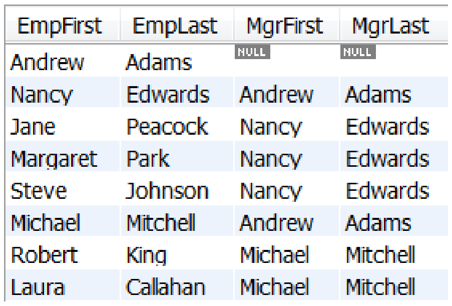
The Chinook employee table is an example of such data. Each record contains a ReportsTo field that points to another record in the same table, the one for that employee's manager. There are a number of questions you can ask in this situation.

Matching all employees with their managers

We'll start with the simplest question. How do we get the name of each employee and his or her supervisor? It takes a self-join, but is fairly straightforward. Other than punctuation (and the inclusion of an INTO clause in VFP), the query is identical in all three SQL versions. **Listing 37** (EmpWMgr in the appropriate folders of the materials for this session) shows the MySQL and SQL Server version. The query uses a self-join of the Employee table. The first instance is given the local alias Emp to indicate its use for employee records, while the second has the local alias Mgr to indicate it's the source for manager records. The LEFT JOIN ensures that the person at the top of the hierarchy is included in the results. The results are shown in **Figure 24**.

Listing 37. It's easy to match every employee with their own manager, using a self-join.

```
SELECT Emp.FirstName AS EmpFirst,
       Emp.LastName AS EmpLast,
       Mgr.FirstName AS MgrFirst,
       Mgr.LastName AS MgrLast
  FROM Employee Emp
    LEFT JOIN Employee Mgr
      ON Emp.ReportsTo = Mgr.EmployeeID ;
```



EmpFirst	EmpLast	MgrFirst	MgrLast
Andrew	Adams	NULL	NULL
Nancy	Edwards	Andrew	Adams
Jane	Peacock	Nancy	Edwards
Margaret	Park	Nancy	Edwards
Steve	Johnson	Nancy	Edwards
Michael	Mitchell	Andrew	Adams
Robert	King	Michael	Mitchell
Laura	Callahan	Michael	Mitchell

Figure 24. A simple query lets us match each employee to their supervisor.

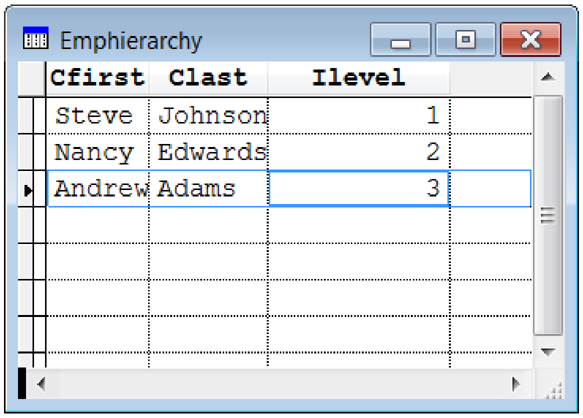
What's the management hierarchy for an employee?

Where things get harder is when you want to walk up or down the hierarchy. Walking up the hierarchy means, given a particular employee, retrieve the name of her manager and of the manager's manager and of the manager's manager's manager and so on up the line until you reach the person in charge.

VFP's SQL alone doesn't offer a solution for this problem. Instead, you need to combine a little bit of SQL with some Xbase code, as in **Listing 38**. This solution (EmpHierarchy.prg in the VFP folder of the materials for this session) is written as a function, with the primary key for the specified employee passed as a parameter. The strategy is to start with the employee you're interested in, insert her data into the result cursor, then grab the PK for her manager and repeat until you reach an employee whose ReportsTo field is empty. The results when the parameter is 5 are shown in **Figure 25**.

Listing 38. To climb the management hierarchy for an employee in VFP, you combine SQL and Xbase code, though it's mostly Xbase.

```
LPARAMETERS iEmpID

LOCAL iCurrentID , iLevel

CREATE CURSOR EmpHierarchy ;
  (cFirst C(15), cLast C(20) , iLevel I)

USE Employee IN 0 ORDER EmployeeID

iCurrentID = iEmpID
iLevel = 1

DO WHILE NOT EMPTY(iCurrentID)

   SEEK iCurrentID IN Employee
   INSERT INTO EmpHierarchy ;
      VALUES (Employee.FirstName, ;
              Employee.LastName, ;
              m.iLevel)

   iCurrentID = Employee.ReportsTo
   iLevel = m.iLevel + 1
ENDDO

USE IN Employee
SELECT EmpHierarchy
```



Cfirst	Clast	Ilevel
Steve	Johnson	1
Nancy	Edwards	2
Andrew	Adams	3

Figure 25. You can show each level of an employee's management hierarchy in one record.

In SQL Server and MySQL 8, you can use a recursive CTE to do the whole job in a single query. The queries differ only in the inclusion of the keyword RECURSIVE, required in MySQL. The SQL Server version is shown in **Listing 39**, while the MySQL version is shown in **Listing 40**. They're included as EmpHierarchy.sql in the respective folders of the materials for this session.

The anchor portion of the CTE selects the specified person (WHERE EmployeeID = @iEmpID), including that person's ManagerID in the result and setting up a field to track the level in the database. The recursive portion of the query joins the Employee table to the EmpHierarchy table-in-progress (that is, the CTE itself), matching the ManagerID from EmpHierarchy to Employee.EmployeeID. It also increments the EmpLevel field, so that the first time it executes, EmpLevel is 2, and the second time, it's 3, and so forth. Once the CTE is complete, the main query pulls the desired information from it.

Listing 39. In SQL Server, a recursive CTE makes it fairly easy to get the management hierarchy for an employee.

```
DECLARE @iEmpID INT = 5;

WITH EmpHierarchy (
  FirstName, LastName, ManagerID, EmpLevel)
AS
(
SELECT FirstName, LastName,
       ReportsTo, 1 AS EmpLevel
  FROM Employee
  WHERE EmployeeID = @iEmpID
UNION ALL
SELECT Employee.FirstName, Employee.LastName,
       Employee.ReportsTo,
       EmpHierarchy.EmpLevel + 1 AS EmpLevel
  FROM Employee
    JOIN EmpHierarchy
      ON Employee.EmployeeID = EmpHierarchy.ManagerID
)

SELECT FirstName, LastName, EmpLevel
  FROM EmpHierarchy;
```

Listing 40. The MySQL 8 version of the code to get the management hierarchy for an employee is almost identical to the SQL Server version.

```
SET @iEmpID = 5;

WITH RECURSIVE EmpHierarchy (
  FirstName, LastName, ManagerID, EmpLevel)
AS
(
SELECT FirstName, LastName,
       ReportsTo, 1 AS EmpLevel
  FROM Employee
  WHERE EmployeeID = @iEmpID
UNION ALL
SELECT Employee.FirstName, Employee.LastName,
       Employee.ReportsTo,
       EmpHierarchy.EmpLevel + 1 AS EmpLevel
  FROM Employee
    JOIN EmpHierarchy
      ON Employee.EmployeeID = EmpHierarchy.ManagerID
)

SELECT FirstName, LastName, EmpLevel
  FROM EmpHierarchy;
```

Who does an employee manage?

The other interesting question is how to walk down the hierarchy, how to find the list of all employees a particular person manages at all levels of the hierarchy. That is, not only those she manages directly, but people who report to those people, and so on down the line.

To make the results more meaningful, we want to include the name of the employee's direct manager in the results.

What makes this difficult in VFP is that at each level, you may (probably do) have multiple employees. You need not only to add each to the result, but to check who each of them manages. That means you need some way of keeping track of who you've checked and who you haven't.

The solution in **Listing 41** (MgrHierarchy.prg in the VFP folder of the materials for this session) uses two cursors. One (MgrHierarchy) holds the results, while the other (EmpsToProcess) holds the list of people to check.

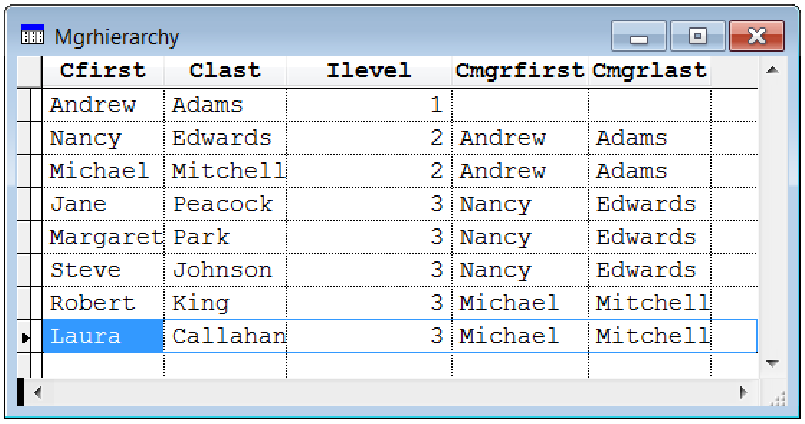
As in the previous example, the code is written as a function, with the primary key of the employee of interest passed as a parameter. To kick the process off, a single record is added to EmpsToProcess, with information about the specified employee. Then a loop through EmpsToProcess handles one employee at a time. A record is inserted into MgrHierarchy for that employee, and then records are added to EmpsToProcess for everyone directly managed by the employee currently being processed.

The most interesting bit of this code is that the SCAN loop has no problem with records being added to the cursor being scanned. We just have to keep track of the record pointer, and after adding records, move it back to the record we're currently processing.

Figure 26 shows the results when employee ID 1 is passed. That employee is the overall boss of the organization, so those results show the complete employee hierarchy.

Listing 41. In VFP, finding everyone an employee manages at all levels of the hierarchy requires two cursors and a mix of Xbase and SQL code.

```
LPARAMETERS iEmpID


LOCAL iCurrentID, iLevel, cFirst, cLast
LOCAL nCurRecNo, cMgrFirst, cMgrLast

CREATE CURSOR MgrHierarchy ;
  (cFirst C(15), cLast C(20), iLevel I, ;
   cMgrFirst C(15), cMgrLast C(15))
CREATE CURSOR EmpsToProcess ;
  (EmployeeID I, cFirst C(15), cLast C(20), ;
   iLevel I, cMgrFirst C(15), cMgrLast C(15))

INSERT INTO EmpsToProcess ;
  SELECT m.iEmpID, FirstName, LastName, 1, "", "" ;
    FROM Employee ;
    WHERE EmployeeID = m.iEmpID

SELECT EmpsToProcess

SCAN
  iCurrentID = EmpsToProcess.EmployeeID
  iLevel = EmpsToProcess.iLevel
  cFirst = EmpsToProcess.cFirst
  cLast = EmpsToProcess.cLast
  cMgrFirst = EmpsToProcess.cMgrFirst
  cMgrLast = EmpsToProcess.cMgrLast

  * Insert this records into result
  INSERT INTO MgrHierarchy ;
    VALUES (m.cFirst, m.cLast, m.iLevel, m.cMgrFirst, m.cMgrLast)
  * Grab the current record pointer
  nCurRecNo = RECNO("EmpsToProcess")

  INSERT INTO EmpsToProcess ;
    SELECT EmployeeID, FirstName, LastName, m.iLevel + 1, m.cFirst, m.cLast ;
      FROM Employee ;
      WHERE ReportsTo = m.iCurrentID

  * Restore record pointer
  GO m.nCurRecNo IN EmpsToProcess
ENDSCAN

SELECT MgrHierarchy
```



Cfirst	Clast	Ilevel	Cmgrfirst	Cmgrlast
Andrew	Adams	1		
Nancy	Edwards	2	Andrew	Adams
Michael	Mitchell	2	Andrew	Adams
Jane	Peacock	3	Nancy	Edwards
Margaret	Park	3	Nancy	Edwards
Steve	Johnson	3	Nancy	Edwards
Robert	King	3	Michael	Mitchell
Laura	Callahan	3	Michael	Mitchell

Figure 26. It takes a mix of Xbase and SQL code to trace downward through the management hierarchy in VFP.

In SQL Server and MySQL, tracing downward is no harder than tracing upward. The solutions in **Listing 42** for SQL Server and **Listing 43** for MySQL 8 (MgrHierarchy.sql in the appropriate folders of the materials for this session) differ from the previous examples only in the direction of the join between the CTE and the Employee table, and in including the additional fields to hold the manager's name for each employee.

Listing 42. To walk down the management hierarchy in SQL Server is no harder than walking up.

```
DECLARE @iEmpID INT = 1;

WITH EmpHierarchy
  (FirstName, LastName, EmployeeID, EmpLevel, MgrFirst, MgrLast)
AS
(
SELECT FirstName, LastName,
       EmployeeID, 1 AS EmpLevel,
       CAST('' AS NVARCHAR(20)) AS MgrFirst,
       CAST('' AS NVARCHAR(20)) AS MgrLast
  FROM Employee
  WHERE EmployeeID = @iEmpID
UNION ALL
SELECT Employee.FirstName, Employee.LastName,
       Employee.EmployeeID,
       EmpHierarchy.EmpLevel + 1 AS EmpLevel,
       EmpHierarchy.FirstName AS MgrFirst,
       EmpHierarchy.LastName AS MgrLast
  FROM Employee
    JOIN EmpHierarchy
      ON Employee.ReportsTo = EmpHierarchy.EmployeeID
)

SELECT FirstName, LastName, EmpLevel,
       MgrFirst, MgrLast
   FROM EmpHierarchy;
```

Listing 43. As in SQL Server, the MySQL code to walk down the management hierarchy is almost identical to the code to walk up the hierarchy.

```
SET @iEmpID = 1;

WITH RECURSIVE EmpHierarchy
  (FirstName, LastName, EmployeeID, EmpLevel, MgrFirst, MgrLast)
AS
(
SELECT FirstName, LastName,
       EmployeeID, 1 AS EmpLevel,
       CAST('' AS CHAR(20)) AS MgrFirst,
       CAST('' AS CHAR(20)) AS MgrLast
  FROM Employee
  WHERE EmployeeID = @iEmpID
UNION ALL
SELECT Employee.FirstName, Employee.LastName,
       Employee.EmployeeID,
       EmpHierarchy.EmpLevel + 1 AS EmpLevel,
       EmpHierarchy.FirstName AS MgrFirst,
       EmpHierarchy.LastName AS MgrLast
  FROM Employee
    JOIN EmpHierarchy
      ON Employee.ReportsTo = EmpHierarchy.EmployeeID
)

SELECT FirstName, LastName, EmpLevel,
       MgrFirst, MgrLast
   FROM EmpHierarchy;
```

The HierarchyID type

Starting in SQL Server 2008, there's a data type called HierarchyID, designed to make it easier to store information about this type of hierarchy. It allows each record to have a single field that encodes the entire hierarchy to reach that record. A set of methods, including GetAncestor and GetDescendant, let you use that data to retrieve other records in the hierarchy. However, queries to extract data from the hierarchy are not significantly different than the ones above.

For more discussion of this data type, see my article about hierarchical data at http://www.tomorrowssolutionsllc.com/Articles/Handling%20Hierarchical%20Data.pdf.

Resources

I've written a number of other papers about SQL. They're available at http://tomorrowssolutionsllc.com/conferencepapers.php; you can filter on the topics "SQL" and "SQL Server" to quickly find them.

Documentation for both SQL Server and MySQL is available online. The language reference for SQL Server is at https://docs.microsoft.com/en-us/sql/t-sql/language-reference, while the documentation for MySQL is at https://dev.mysql.com/doc/.

As with many other topics, StackOverflow is an excellent place to find peer support for SQL languages. W3Schools has a great deal of SQL information beginning at https://www.w3schools.com/sql/default.asp.

Summary

The SQL language is a powerful tool for working with data. It can be used to solve a wide range of problems, but different SQL implementations may require different solutions.

The problems are solutions in this paper are just a small sample of what can be accomplished with SQL. The techniques used here should help you approach other problems.